CHAPTER 27

Compute-Bound Asynchronous Operations

In this chapter:

In this chapter, I'll talk about the various ways that you can perform operations asynchronously. When performing an asynchronous compute-bound operation, you execute it using other threads. Here are some examples of compute-bound operations: compiling code, spell checking, grammar checking, spreadsheet recalculations, transcoding audio or video data, and producing a thumbnail of an image. Certainly, compute-bound operations are common in financial and engineering applications.

I would say that most applications do not spend the bulk of their time processing in-memory data or performing calculations. You can verify that this is true by opening Task Manager and selecting the Performance tab. If your CPU usage is below 100 percent (which it tends to be most of the time), then the processes you have running are not using all the processing power made available by your machine's CPU cores. When the CPU usage is less than 100 percent, then some (if not all) of the threads within their processes are not running at all. Instead, these threads are waiting for some input or output operation to occur. For example, these threads are waiting for a timer to come due, waiting for data to be read from or written to a database, web service, file, network, or other hardware device, or waiting for keystrokes, mouse movement, or mouse button clicks. When performing an I/O-bound operation, the Windows device driver has the hardware device do the work for you, and the CPU itself doesn't execute any threads that happen to exist in the system. Because threads are not running on a CPU, Task Manager indicates that CPU usage is low.

However, even in applications that are heavily I/O-bound, these applications perform some computation on data that has been received, and parallelizing this computation can greatly improve the application's throughput. This chapter introduces the common language runtime's (CLR's) thread pool and some basic concepts about how it works and how to use it. This information is critically useful, because the thread pool is the core technology that enables you to design and implement responsive and scalable applications and components. Then this chapter shows the various mechanisms available that allow you to perform compute-bound operations via the thread pool.

Introducing the CLR's Thread Pool

As stated in the previous chapter, creating and destroying a thread is an expensive operation in terms of time. In addition, having lots of threads wastes memory resources and also hurts performance due to the operating system having to schedule and context switch between the runnable threads. To improve this situation, the CLR contains code to manage its own thread pool. You can think of a thread pool as being a set of threads that are available for your application's own use. There is one thread pool per CLR; this thread pool is shared by all AppDomains controlled by that CLR. If multiple CLRs load within a single process, then each CLR has its own thread pool.

When the CLR initializes, the thread pool has no threads in it. Internally, the thread pool maintains a queue of operation requests. When your application wants to perform an asynchronous operation, you call some method that appends an entry into the thread pool's queue. The thread pool's code will extract entries from this queue and dispatch the entry to a thread pool thread. If there are no threads in the thread pool, a new thread will be created. Creating a thread has a performance hit associated with it (as already discussed). However, when a thread pool thread has completed its task, the thread is not destroyed; instead, the thread is returned to the thread pool, where it sits idle waiting to respond to another request. Because the thread doesn't destroy itself, there is no added performance hit.

If your application makes many requests of the thread pool, the thread pool will try to service all of the requests by using just this one thread. However, if your application is queuing up several requests faster than the thread pool thread can handle them, additional threads will be created. Your application will eventually get to a point at which all of its requests can be handled by a small number of threads, so the thread pool should have no need to create a lot of threads.

If your application stops making requests of the thread pool, the pool may have a lot of threads in it that are doing nothing. This is wasteful of memory resources. So when a thread pool thread has been idle with nothing to do for some period of time (subject to change with different versions of the CLR), the thread wakes itself up and kills itself to free up resources. As the thread is killing itself, there is a performance hit. However, this probably doesn't matter, because the thread is killing itself because it has been idle, which means that your application isn't performing a lot of work.

The great thing about the thread pool is that it manages the tension between having a few threads, to keep from wasting resources, and having more threads, to take advantage of multiprocessors, hyperthreaded processors, and multi-core processors. And the thread pool is heuristic. If

your application needs to perform many tasks and CPUs are available, the thread pool creates more threads. If your application's workload decreases, the thread pool threads kill themselves.

Performing a Simple Compute-Bound Operation

To queue an asynchronous compute-bound operation to the thread pool, you typically call one of the following methods defined by the `ThreadPool` class.

```
static Boolean QueueUserWorkItem(WaitCallback callBack);
static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);
```

These methods queue a "work item" and optional state data to the thread pool's queue, and then all of these methods return immediately. A work item is simply a method identified by the `callback` parameter that will be called by a thread pool thread. The method can be passed a single parameter specified via the `state` (the state data) argument. The version of `QueueUserWorkItem` without the `state` parameter passes `null` to the callback method. Eventually, some thread in the pool will process the work item, causing your method to be called. The callback method you write must match the `System.Threading.WaitCallback` delegate type, which is defined as follows.

```
delegate void WaitCallback(Object state);
```

Note The signatures of the `WaitCallback` delegate, the `TimerCallback` delegate (discussed in this chapter's "Performing a Periodic Compute-Bound Operation" section), and the `ParameterizedThreadStart` delegate (discussed in Chapter 26, "Thread Basics") are all identical. If you define a method matching this signature, the method can be invoked by using `ThreadPool.QueueUserWorkItem`, by using a `System.Threading.Timer` object, or by using a `System.Threading.Thread` object.

The following code demonstrates how to have a thread pool thread call a method asynchronously.

```
using System;
using System.Threading;

public static class Program {
   public static void Main() {
      Console.WriteLine("Main thread: queuing an asynchronous operation");
      ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);
      Console.WriteLine("Main thread: Doing other work here...");
      Thread.Sleep(10000);  // Simulating other work (10 seconds)
      Console.WriteLine("Hit <Enter> to end this program...");
      Console.ReadLine();
   }

   // This method's signature must match the WaitCallback delegate
   private static void ComputeBoundOp(Object state) {
      // This method is executed by a thread pool thread
```

```
        Console.WriteLine("In ComputeBoundOp: state={0}", state);
        Thread.Sleep(1000);  // Simulates other work (1 second)

        // When this method returns, the thread goes back
        // to the pool and waits for another task
    }
}
```

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

```
Main thread: queuing an asynchronous operation
Main thread: Doing other work here...
In ComputeBoundOp: state=5
```

And, sometimes when I run this code, I get this output.

```
Main thread: queuing an asynchronous operation
In ComputeBoundOp: state=5
Main thread: Doing other work here...
```

The difference in the order of the lines in the output is attributed to the fact that the two methods are running asynchronously with respect to one another. The Windows scheduler determines which thread to schedule first, or it may schedule them both simultaneously if the application is running on a multi-CPU machine.

Note If the callback method throws an exception that is unhandled, the CLR terminates the process (unless the host imposes its own policy). Unhandled exceptions are discussed in Chapter 20, "Exceptions and State Management."

Note For Windows Store apps, the `System.Threading.ThreadPool` class is not publicly exposed. However, it is used indirectly when you use types in the `System.Threading.Tasks` namespace (see the "Tasks" Section later in this chapter).

Execution Contexts

Every thread has an execution context data structure associated with it. The execution context includes things such as security settings (compressed stack, `Thread`'s `Principal` property, and Windows identity), host settings (see `System.Threading.HostExecutionContextManager`), and logical call context data (see `System.Runtime.Remoting.Messaging.CallContext`'s `LogicalSetData` and `LogicalGetData` methods). When a thread executes code, some operations are affected by the values of the thread's execution context settings. This is certainly true for the security settings. Ideally, whenever a thread uses another (helper) thread to perform tasks, the issuing thread's execution context should flow (be copied) to the helper thread. This ensures that any operations performed by helper thread(s) are executing with the same security settings and host settings. It also

ensures that any data stored in the initiating thread's logical call context is available to the helper thread.

By default, the CLR automatically causes the initiating thread's execution context to flow to any helper threads. This transfers context information to the helper thread, but it comes at a performance cost, because there is a lot of information in an execution context, and accumulating all of this information and then copying it for the helper thread takes a fair amount of time. If the helper thread then employs additional helper threads, then more execution context data structures have to be created and initialized as well.

In the `System.Threading` namespace, there is an `ExecutionContext` class that allows you to control how a thread's execution context flows from one thread to another. Here is what the class looks like.

```
public sealed class ExecutionContext : IDisposable, ISerializable {
   [SecurityCritical] public static AsyncFlowControl SuppressFlow();
   public static void RestoreFlow();
   public static Boolean IsFlowSuppressed();

   // Less commonly used methods are not shown
}
```

You can use this class to suppress the flowing of an execution context, thereby improving your application's performance. The performance gains can be quite substantial for a server application. There is not much performance benefit for a client application, and the `SuppressFlow` method is marked with the `[SecurityCritical]` attribute, making it impossible to call in some client applications (like Microsoft Silverlight). Of course, you should suppress the flowing of execution context only if the helper thread does not need to access the context information. If the initiating thread's execution context does not flow to a helper thread, the helper thread will use whatever execution context it last associated with it. Therefore, the helper thread really shouldn't execute any code that relies on the execution context state (such as a user's Windows identity).

Here is an example showing how suppressing the flow of execution context affects data in a thread's logical call context when queuing a work item to the CLR's thread pool.[1]

```
public static void Main() {
   // Put some data into the Main thread's logical call context
   CallContext.LogicalSetData("Name", "Jeffrey");

   // Initiate some work to be done by a thread pool thread
   // The thread pool thread can access the logical call context data
   ThreadPool.QueueUserWorkItem(
      state => Console.WriteLine("Name={0}", CallContext.LogicalGetData("Name")));


   // Now, suppress the flowing of the Main thread's execution context
   ExecutionContext.SuppressFlow();
```

[1] The items that you add to the logical call context must be serializable, as discussed in Chapter 24, "Runtime Serialization." Flowing an execution context that contains logical call context data items can hurt performance dramatically, because capturing the execution context requires serializing and deserializing all the data items.

```
   // Initiate some work to be done by a thread pool thread
   // The thread pool thread CANNOT access the logical call context data
   ThreadPool.QueueUserWorkItem(
      state => Console.WriteLine("Name={0}", CallContext.LogicalGetData("Name")));

   // Restore the flowing of the Main thread's execution context in case
   // it employs more thread pool threads in the future
   ExecutionContext.RestoreFlow();
   ...
   Console.ReadLine();
}
```

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

```
Name=Jeffrey
Name=
```

Although this discussion has focused on suppressing the flow of execution context when calling `ThreadPool.QueueUserWorkItem`, this technique is also useful when using `Task` objects (discussed in the "Tasks" section of this chapter) and is also useful when initiating asynchronous I/O operations (discussed in Chapter 28, "I/O-Bound Asynchronous Operations").

Cooperative Cancellation and Timeout

The Microsoft .NET Framework offers a standard pattern for canceling operations. This pattern is *cooperative*, meaning that the operation that you want to cancel has to explicitly support being canceled. In other words, the code performing the operation that you want to cancel and the code that attempts to cancel the operation must both use the types mentioned in this section. It is nice when long-running compute-bound operations offer cancellation, so you should consider adding cancellation to your own compute-bound operations. In this section, I'll explain how to accomplish this. But, first, let me explain the two main types provided in the Framework Class Library (FCL) that are part of the standard cooperative cancellation pattern.

To cancel an operation, you must first create a `System.Threading.CancellationTokenSource` object. This class looks like this.

```
public sealed class CancellationTokenSource : IDisposable {  // A reference type
   public CancellationTokenSource();

   public Boolean IsCancellationRequested { get; }
   public CancellationToken Token { get; }

   public void Cancel();  // Internally, calls Cancel passing false
   public void Cancel(Boolean throwOnFirstException);
   ...
}
```

This object contains all the states having to do with managing cancellation. After constructing a `CancellationTokenSource` (a reference type), one or more `CancellationToken` (a value type) instances can be obtained by querying its `Token` property and passed around to your operations that

allow themselves to be canceled. Here are the most useful members of the CancellationToken value type.

```
public struct CancellationToken {  // A value type
   public static CancellationToken None { get; }  // Very convenient

   public Boolean    IsCancellationRequested { get; } // Called by non-Task invoked operations
   public void       ThrowIfCancellationRequested();  // Called by Task-invoked operations

   // WaitHandle is signaled when the CancellationTokenSource is canceled
   public WaitHandle WaitHandle { get; }
   // GetHashCode, Equals, operator== and operator!= members are not shown

   public Boolean CanBeCanceled { get; }  // Rarely used

   public CancellationTokenRegistration Register(Action<Object> callback, Object state,
      Boolean useSynchronizationContext);  // Simpler overloads not shown
}
```

A CancellationToken instance is a lightweight value type because it contains a single private field: a reference to its CancellationTokenSource object. A compute-bound operation's loop can periodically call CancellationToken's IsCancellationRequested property to know if the loop should terminate early, thereby ending the compute-bound operation. Of course, the benefit here is that CPU time is no longer being wasted on an operation whose result you know you're not interested in. Now, let me put all this together with some sample code.

```
internal static class CancellationDemo {
   public static void Main() {
      CancellationTokenSource cts = new CancellationTokenSource();

      // Pass the CancellationToken and the number-to-count-to into the operation
      ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));

      Console.WriteLine("Press <Enter> to cancel the operation.");
      Console.ReadLine();
      cts.Cancel();  // If Count returned already, Cancel has no effect on it
      // Cancel returns immediately, and the method continues running here...

      Console.ReadLine();
   }

   private static void Count(CancellationToken token, Int32 countTo) {
      for (Int32 count = 0; count <countTo; count++) {
         if (token.IsCancellationRequested) {
            Console.WriteLine("Count is cancelled");
            break; // Exit the loop to stop the operation
         }

         Console.WriteLine(count);
         Thread.Sleep(200);   // For demo, waste some time
      }
      Console.WriteLine("Count is done");
   }
}
```

Note If you want to execute an operation and prevent it from being canceled, you can pass the operation the `CancellationToken` returned from calling `CancellationToken`'s static `None` property. This very convenient property returns a special `CancellationToken` instance that is not associated with any `CancellationTokenSource` object (its private field is `null`). Because there is no `CancellationTokenSource`, no code can call `Cancel`, and the operation that is querying the special `CancellationToken`'s `IsCancellationRequested` property will always return false. If you query `CancellationToken`'s `CanBeCanceled` property by using one of these special `CancellationToken` instances, the property will return false. The property returns true for all other `CancellationToken` instances obtained by querying a `CancellationTokenSource` object's `Token` property.

If you'd like, you can call `CancellationTokenSource`'s `Register` method to register one or more methods to be invoked when a `CancellationTokenSource` is canceled. To this method, you pass an `Action<Object>` delegate, a state value that will be passed to the callback via the delegate, and a `Boolean` indicating whether or not to invoke the delegate by using the calling thread's `SynchronizationContext`. If you pass `false` for the `useSynchronizationContext` parameter, then the thread that calls `Cancel` will invoke all the registered methods sequentially. If you pass `true` for the `useSynchronizationContext` parameter, then the callbacks are sent (as opposed to posted) to the captured `SynchronizationContext` object that governs which thread invokes the callback. The `SynchronizationContext` class is discussed more in the "Applications and Their Threading Models" section in Chapter 28.

Note If you register a callback method with a `CancellationTokenSource` after the `CancellationTokenSource` has already been canceled, then the thread calling `Register` invokes the callback (possible via the calling thread's `SynchronizationContext` if `true` is passed for the `useSynchronizationContext` parameter).

If `Register` is called multiple times, then multiple callback methods will be invoked. These callback methods could throw an unhandled exception. If you call `CancellationTokenSource`'s `Cancel`, passing it `true`, then the first callback method that throws an unhandled exception stops the other callback methods from executing, and the exception thrown will be thrown from `Cancel` as well. If you call `Cancel` passing it `false`, then all registered callback methods are invoked. Any unhandled exceptions that occur are added to a collection. After all callback methods have executed, if any of them threw an unhandled exception, then `Cancel` throws an `AggregateException` with its `InnerExceptions` property set to the collection of exception objects that were thrown. If no registered callback methods threw an unhandled exception, then `Cancel` simply returns without throwing any exception at all.

Important There is no way to correlate an exception object from `AggregateException`'s `InnerExceptions` collection to a particular operation; you are basically just being told that *some* operation failed and the exception type tells you what the failure was. To track down the specific location of the failure will require examining the exception object's `StackTrace` property and manually scanning your source code.

`CancellationToken`'s `Register` method returns a `CancellationTokenRegistration`, which looks like this.

```
public struct CancellationTokenRegistration :
   IEquatable<CancellationTokenRegistration>, IDisposable {
   public void Dispose();
   // GetHashCode, Equals, operator== and operator!= members are not shown
}
```

You can call `Dispose` to remove a registered callback from the `CancellationTokenSource` that it is associated with so that it does not get invoked when calling `Cancel`. Here is some code that demonstrates registering two callbacks with a single `CancellationTokenSource`.

```
var cts = new CancellationTokenSource();
cts.Token.Register(() => Console.WriteLine("Canceled 1"));
cts.Token.Register(() => Console.WriteLine("Canceled 2"));

// To test, let's just cancel it now and have the 2 callbacks execute
cts.Cancel();
```

When I run this code, I get the following output as soon as the `Cancel` method is called.

```
Canceled 2
Canceled 1
```

Finally, you can create a new `CancellationTokenSource` object by linking a bunch of other `CancellationTokenSource` objects. This new `CancellationTokenSource` object will be canceled when *any* of the linked `CancellationTokenSource` objects are canceled. The following code demonstrates.

```
// Create a CancellationTokenSource
var cts1 = new CancellationTokenSource();
cts1.Token.Register(() => Console.WriteLine("cts1 canceled"));

// Create another CancellationTokenSource
var cts2 = new CancellationTokenSource();
cts2.Token.Register(() => Console.WriteLine("cts2 canceled"));

// Create a new CancellationTokenSource that is canceled when cts1 or ct2 is canceled
var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(cts1.Token, cts2.Token);
linkedCts.Token.Register(() => Console.WriteLine("linkedCts canceled"));
```

```
// Cancel one of the CancellationTokenSource objects (I chose cts2)
cts2.Cancel();

// Display which CancellationTokenSource objects are canceled
Console.WriteLine("cts1 canceled={0}, cts2 canceled={1}, linkedCts canceled={2}",
   cts1.IsCancellationRequested, cts2.IsCancellationRequested,
linkedCts.IsCancellationRequested);
```

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

```
linkedCts canceled
cts2 canceled
cts1 canceled=False, cts2 canceled=True, linkedCts canceled=True
```

It is often valuable to cancel an operation after a period of time has elapsed. For example, imagine a server application that starts computing some work based on a client request. But the server application needs to respond to the client within two seconds, no matter what. In some scenarios, it is better to respond in a short period of time with an error or with partially computed results as opposed to waiting a long time for a complete result. Fortunately, `CancellationTokenSource` gives you a way to have it self-cancel itself after a period of time. To take advantage of this, you can either construct a `CancellationTokenSource` object by using one of the constructors that accepts a delay, or you can call `CancellationTokenSource`'s `CancelAfter` method.

```
public sealed class CancellationTokenSource : IDisposable {  // A reference type
   public CancellationTokenSource(Int32 millisecondsDelay);
   public CancellationTokenSource(TimeSpan delay);

   public void CancelAfter(Int32 millisecondsDelay);
   public void CancelAfter(TimeSpan delay);
   ...
}
```

Tasks

Calling `ThreadPool`'s `QueueUserWorkItem` method to initiate an asynchronous compute-bound operation is very simple. However, this technique has many limitations. The biggest problem is that there is no built-in way for you to know when the operation has completed, and there is no way to get a return value back when the operation completes. To address these limitations and more, Microsoft introduced the concept of *tasks*, and you use them via types in the `System.Threading.Tasks` namespace.

So, instead of calling `ThreadPool`'s `QueueUserWorkItem` method, you can do the same via tasks.

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5); // Calling QueueUserWorkItem
new Task(ComputeBoundOp, 5).Start();             // Equivalent of preceding using Task
Task.Run(() => ComputeBoundOp(5));               // Another equivalent
```

In the second line of preceding code, I am creating the `Task` object and then immediately calling `Start` to schedule the task to run. Naturally, you can create the `Task` object and then call `Start` on it later. You could imagine code that creates a `Task` object and then passes it to some other method that decides when to call `Start` to schedule the task. Because it is common to create a `Task` object and then immediately call `Start` on it, you can call `Task`'s convenient static `Run` method as shown on the last line of the preceding code.

When creating a `Task`, you call a constructor, passing it an `Action` or an `Action<Object>` delegate that indicates the operation that you want performed. If you pass a method that expects an `Object`, then you must also pass to `Task`'s constructor the argument that you ultimately want passed to the operation. When calling `Run`, you can pass it an `Action` or `Func<TResult>` delegate indicating the operation you want performed. When calling a constructor or when calling `Run`, you can optionally pass a `CancellationToken`, which allows the `Task` to be canceled before it has been scheduled (see the "Canceling a Task" section later in this chapter).

You can also optionally pass to the constructor some `TaskCreationOptions` flags that control how the `Task` executes. `TaskCreationOptions` is an enumerated type defining a set of flags that you can bitwise-OR together. It is defined as follows.

```
[Flags, Serializable]
public enum TaskCreationOptions {
   None             = 0x0000,// The default

   // Hints to the TaskScheduler that you want this task to run sooner than later.
   PreferFairness      = 0x0001,

   // Hints to the TaskScheduler that it should more aggressively create thread pool threads.
   LongRunning         = 0x0002,

   // Always honored: Associates a Task with its parent Task (discussed shortly)
   AttachedToParent    = 0x0004,

   // If a task attempts to attach to this parent task, it is a normal task, not a child task.
   DenyChildAttach     = 0x0008,

   // Forces child tasks to use the default scheduler as opposed to the parent's scheduler.
   HideScheduler       = 0x0010
}
```

Some of these flags are hints that may or may not be honored by the `TaskScheduler` that is being used to schedule a `Task`; the `AttachedToParent`, `DenyChildAttach`, and `HideScheduler` flags are always honored, because they have nothing to do with the `TaskScheduler` itself. `TaskScheduler` objects are discussed later in the "Task Schedulers" section.

Waiting for a Task to Complete and Getting Its Result

With tasks, it is also possible to wait for them to complete and then get their result. Let's say that we have a Sum method that is computationally intensive if n is a large value.

```
private static Int32 Sum(Int32 n) {
   Int32 sum = 0;
   for (; n > 0; n--)
      checked { sum += n; }   // if n is large, this will throw System.OverflowException
   return sum;
}
```

We can now construct a `Task<TResult>` object (which is derived from `Task`), and we pass for the generic `TResult` argument the compute-bound operation's return type. Now, after starting the task, we can wait for it to complete and then get its result by using the following code.

```
// Create a Task (it does not start running now)
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

// You can start the task sometime later
t.Start();

// Optionally, you can explicitly wait for the task to complete
t.Wait(); // FYI: Overloads exist accepting timeout/CancellationToken

// You can get the result (the Result property internally calls Wait)
Console.WriteLine("The Sum is: " + t.Result); // An Int32 value
```

Important When a thread calls the `Wait` method, the system checks if the `Task` that the thread is waiting for has started executing. If it has, then the thread calling `Wait` will block until the `Task` has completed running. But if the `Task` has not started executing yet, then the system *may* (depending on the `TaskScheduler`) execute the `Task` by using the thread that called `Wait`. If this happens, then the thread calling `Wait` does not block; it executes the `Task` and returns immediately. This is good in that no thread has blocked, thereby reducing resource usage (by not creating a thread to replace the blocked thread) while improving performance (no time is spent to create a thread and there is no context switching). But it can also be bad if, for example, the thread has taken a thread synchronization lock before calling `Wait` and then the `Task` tries to take the same lock, resulting in a deadlocked thread!

If the compute-bound task throws an unhandled exception, the exception will be swallowed, stored in a collection, and the thread pool thread is allowed to return to the thread pool. When the `Wait` method or the `Result` property is invoked, these members will throw a `System.AggregateException` object.

The AggregateException type is used to encapsulate a collection of exception objects (which can happen if a parent task spawns multiple child tasks that throw exceptions). It contains an InnerExceptions property that returns a ReadOnlyCollection<Exception> object. Do not confuse the InnerExceptions property with the InnerException property, which the AggregateException class inherits from the System.Exception base class. For the preceding example, element 0 of AggregateException's InnerExceptions property would refer to the actual System.OverflowException object thrown by the compute-bound method (Sum).

As a convenience, AggregateException overrides Exception's GetBaseException method. AggregateException's implementation returns the innermost AggregateException that is the root cause of the problem (assuming that there is just one innermost exception in the collection). AggregateException also offers a Flatten method that creates a new AggregateException, whose InnerExceptions property contains a list of exceptions produced by walking the original AggregateException's inner exception hierarchy. Finally, AggregateException also provides a Handle method that invokes a callback method for each exception contained in the AggregateException. The callback can then decide, for each exception, how to handle the exception; the callback returns true to consider the exception handled and false if not. If, after calling Handle, at least one exception is not handled, then a new AggregateException object is created containing just the unhandled exceptions and the new AggregateException object is thrown. Later in this chapter, I show examples using the Flatten and Handle methods.

Important If you never call Wait or Result or query a Task's Exception property, then your code never observes that this exception has occurred. This is not ideal, because your program has experienced an unexpected problem that you are not aware of. To help you detect unobserved exceptions, you can register a callback method with TaskScheduler's static UnobservedTaskException event. This event is raised by the CLR's finalizer thread whenever a Task with an unobserved exception is garbage collected. When raised, your event handler method will be passed an UnobservedTaskExceptionEventArgs object containing the unobserved AggregateException.

In addition to waiting for a single task, the Task class also offers two static methods that allow a thread to wait on an array of Task objects. Task's static WaitAny method blocks the calling thread until any of the Task objects in the array have completed. This method returns an Int32 index into the array indicating which Task object completed, causing the thread to wake and continue running. The method returns -1 if the timeout occurs and throws an OperationCanceledException if WaitAny is canceled via a CancellationToken.

Similarly, the Task class has a static WaitAll method that blocks the calling thread until all the Task objects in the array have completed. The WaitAll method returns true if all the Task objects complete and false if a timeout occurs; an OperationCanceledException is thrown if WaitAll is canceled via a CancellationToken.

Canceling a Task

Of course, you can use a `CancellationTokenSource` to cancel a `Task`. First, we must revise our `Sum` method so that it accepts a `CancellationToken`.

```
private static Int32 Sum(CancellationToken ct, Int32 n) {
   Int32 sum = 0;
   for (; n > 0; n--) {

      // The following line throws OperationCanceledException when Cancel
      // is called on the CancellationTokenSource referred to by the token
      ct.ThrowIfCancellationRequested();

      checked { sum += n; }   // if n is large, this will throw System.OverflowException
   }
   return sum;
}
```

In this code, the compute-bound operation's loop periodically checks to see if the operation has been canceled by calling `CancellationToken`'s `ThrowIfCancellationRequested` method. This method is similar to `CancellationToken`'s `IsCancellationRequested` property shown earlier in the "Cooperative Cancellation and Timeout" section. However, `ThrowIfCancellationRequested` throws an `OperationCanceledException` if the `CancellationTokenSource` has been canceled. The reason for throwing an exception is because, unlike work items initiated with `ThreadPool`'s `QueueUserWorkItem` method, tasks have the notion of having completed and a task can even return a value. So, there needs to be a way to distinguish a completed task from a canceled task, and having the task throw an exception lets you know that the task did not run all the way to completion.

Now, we will create the `CancellationTokenSource` and `Task` objects as follows.

```
CancellationTokenSource cts = new CancellationTokenSource();
Task<Int32> t = Task.Run(() => Sum(cts.Token, 1000000000), cts.Token);

// Sometime later, cancel the CancellationTokenSource to cancel the Task
cts.Cancel(); // This is an asynchronous request, the Task may have completed already

try {
   // If the task got canceled, Result will throw an AggregateException
   Console.WriteLine("The sum is: " + t.Result);   // An Int32 value
}
catch (AggregateException x) {
   // Consider any OperationCanceledException objects as handled.
   // Any other exceptions cause a new AggregateException containing
   // only the unhandled exceptions to be thrown
   x.Handle(e => e is OperationCanceledException);

   // If all the exceptions were handled, the following executes
   Console.WriteLine("Sum was canceled");
}
```

When creating a Task, you can associate a CancellationToken with it by passing it to Task's constructor (as shown in the preceding code). If the CancellationToken gets canceled before the Task is scheduled, the Task gets canceled and never executes at all.[2] But if the Task has already been scheduled (by calling the Start method), then the Task's code must explicitly support cancellation if it allows its operation to be canceled while executing. Unfortunately, while a Task object has a CancellationToken associated with it, there is no way to access it, so you must somehow get the *same* CancellationToken that was used to create the Task object into the Task's code itself. The easiest way to write this code is to use a lambda expression and "pass" the CancellationToken as a closure variable (as I've done in the previous code example).

Starting a New Task Automatically When Another Task Completes

In order to write scalable software, you must not have your threads block. This means that calling Wait or querying a task's Result property when the task has not yet finished running will most likely cause the thread pool to create a new thread, which increases resource usage and hurts performance. Fortunately, there is a better way to find out when a task has completed running. When a task completes, it can start another task. Here is a rewrite of the earlier code that doesn't block any threads.

```
// Create and start a Task, continue with another task
Task<Int32> t = Task.Run(() => Sum(CancellationToken.None, 10000));

// ContinueWith returns a Task but you usually don't care
Task cwt = t.ContinueWith(task => Console.WriteLine("The sum is: " + task.Result));
```

Now, when the task executing Sum completes, this task will start another task (also on some thread pool thread) that displays the result. The thread that executes the preceding code does not block waiting for either of these two tasks to complete; the thread is allowed to execute other code or, if it is a thread pool thread itself, it can return to the pool to perform other operations. Note that the task executing Sum could complete before ContinueWith is called. This will not be a problem because the ContinueWith method will see that the Sum task is complete and it will immediately start the task that displays the result.

Also, note that ContinueWith returns a reference to a new Task object (which my code placed in the cwt variable). Of course, you can invoke various members (like Wait, Result, or even ContinueWith) using this Task object, but usually you will ignore this Task object and will not save a reference to it in a variable.

I should also mention that Task objects internally contain a collection of ContinueWith tasks. So you can actually call ContinueWith several times using a single Task object. When the task completes, all the ContinueWith tasks will be queued to the thread pool. In addition, when calling ContinueWith, you can specify a bitwise OR'd set of TaskContinuationOptions. The first six flags—None, PreferFairness, LongRunning, AttachedToParent, DenyChildAttach, and

[2] By the way, if you try to cancel a task before it is even started, an InvalidOperationException is thrown.

HideScheduler—are identical to the flags offered by the TaskCreationOptions enumerated type shown earlier. Here is what the TaskContinuationOptions type looks like.

```
[Flags, Serializable]
public enum TaskContinuationOptions {
   None             = 0x0000,// The default

   // Hints to the TaskScheduler that you want this task to run sooner than later.
   PreferFairness      = 0x0001,

   // Hints to the TaskScheduler that it should more aggressively create thread pool threads.
   LongRunning        = 0x0002,

   // Always honored: Associates a Task with its parent Task (discussed shortly)
   AttachedToParent   = 0x0004,

   // If a task attempts to attach to this parent task, an InvalidOperationException is thrown.
   DenyChildAttach    = 0x0008,

   // Forces child tasks to use the default scheduler as opposed to the parent's scheduler.
   HideScheduler      = 0x0010,

   // Prevents completion of the continuation until the antecedent has completed.
   LazyCancellation   = 0x0020,

   // This flag indicates that you want the thread that executed the first task to also
   // execute the ContinueWith task. If the first task has already completed, then the
   // thread calling ContinueWith will execute the ContinueWith task.
   ExecuteSynchronously  = 0x80000,

   // These flags indicate under what circumstances to run the ContinueWith task
   NotOnRanToCompletion  = 0x10000,
   NotOnFaulted          = 0x20000,
   NotOnCanceled         = 0x40000,

   // These flags are convenient combinations of the above three flags
   OnlyOnCanceled        = NotOnRanToCompletion | NotOnFaulted,
   OnlyOnFaulted         = NotOnRanToCompletion | NotOnCanceled,
   OnlyOnRanToCompletion = NotOnFaulted         | NotOnCanceled,
}
```

When you call ContinueWith, you can indicate that you want the new task to execute only if the first task is canceled by specifying the TaskContinuationOptions.OnlyOnCanceled flag. Similarly, you have the new task execute only if the first task throws an unhandled exception using the TaskContinuationOptions.OnlyOnFaulted flag. And, of course, you can use the TaskContinuationOptions.OnlyOnRanToCompletion flag to have the new task execute only if the first task runs all the way to completion without being canceled or throwing an unhandled exception. By default, if you do not specify any of these flags, then the new task will run regardless of how the first task completes. When a Task completes, any of its continue-with tasks that do not run are automatically canceled. Here is an example that puts all of this together.

```
// Create and start a Task, continue with multiple other tasks
Task<Int32> t = Task.Run(() => Sum(10000));
// Each ContinueWith returns a Task but you usually don't care
```

```
t.ContinueWith(task => Console.WriteLine("The sum is: " + task.Result),
   TaskContinuationOptions.OnlyOnRanToCompletion);

t.ContinueWith(task => Console.WriteLine("Sum threw: " + task.Exception.InnerException),
   TaskContinuationOptions.OnlyOnFaulted);

t.ContinueWith(task => Console.WriteLine("Sum was canceled"),
   TaskContinuationOptions.OnlyOnCanceled);
```

A Task May Start Child Tasks

Finally, tasks support parent/child relationships, as demonstrated by the following code.

```
Task<Int32[]> parent = new Task<Int32[]>(() => {
   var results = new Int32[3];   // Create an array for the results

   // This tasks creates and starts 3 child tasks
   new Task(() => results[0] = Sum(10000), TaskCreationOptions.AttachedToParent).Start();
   new Task(() => results[1] = Sum(20000), TaskCreationOptions.AttachedToParent).Start();
   new Task(() => results[2] = Sum(30000), TaskCreationOptions.AttachedToParent).Start();

   // Returns a reference to the array (even though the elements may not be initialized yet)
   return results;
});

// When the parent and its children have run to completion, display the results
var cwt = parent.ContinueWith(
   parentTask => Array.ForEach(parentTask.Result, Console.WriteLine));

// Start the parent Task so it can start its children
parent.Start();
```

Here, the parent task creates and starts three `Task` objects. By default, `Task` objects created by another task are top-level tasks that have no relationship to the task that creates them. However, the `TaskCreationOptions.AttachedToParent` flag associates a `Task` with the `Task` that creates it so that the creating task is not considered finished until all its children (and grandchildren) have finished running. When creating a `Task` by calling the `ContinueWith` method, you can make the continue-with task be a child by specifying the `TaskContinuationOptions.AttachedToParent` flag.

Inside a Task

Each `Task` object has a set of fields that make up the task's state. There is an `Int32` ID (see `Task`'s read-only `Id` property), an `Int32` representing the execution state of the `Task`, a reference to the parent task, a reference to the `TaskScheduler` specified when the `Task` was created, a reference to the callback method, a reference to the object that is to be passed to the callback method (queryable via `Task`'s read-only `AsyncState` property), a reference to an `ExecutionContext`, and a reference to a `ManualResetEventSlim` object. In addition, each `Task` object has a reference to some supplementary state that is created on demand. The supplementary state includes a `CancellationToken`, a collection of `ContinueWithTask` objects, a collection of `Task` objects for child tasks that have thrown unhandled exceptions, and more. My point is that although tasks provide you a lot of

features, there is some cost to tasks because memory must be allocated for all this state. If you don't need the additional features offered by tasks, then your program will use resources more efficiently if you use `ThreadPool.QueueUserWorkItem`.

The `Task` and `Task<TResult>` classes implement the `IDisposable` interface, allowing you to call `Dispose` when you are done with the `Task` object. Today, all the `Dispose` method does is close the `ManualResetEventSlim` object. However, it is possible to define classes derived from `Task` and `Task<TResult>`, and these classes could allocate their own resources, which would be freed in their override of the `Dispose` method. I recommend that developers not explicitly call `Dispose` on a `Task` object in their code; instead, just let the garbage collector clean up any resources when it determines that they are no longer in use.

You'll notice that each `Task` object contains an `Int32` field representing a `Task`'s unique ID. When you create a `Task` object, the field is initialized to zero. Then the first time you query `Task`'s read-only `Id` property, the property assigns a unique `Int32` value to this field and returns it from the property. Task IDs start at 1 and increment by 1 as each ID is assigned. Just looking at a `Task` object in the Microsoft Visual Studio debugger will cause the debugger to display the `Task`'s ID, forcing the `Task` to be assigned an ID.

The idea behind the ID is that each `Task` can be identified by a unique value. In fact, Visual Studio shows you these task IDs in its Parallel Tasks and Parallel Stacks windows. But because you don't assign the IDs yourself in your code, it is practically impossible to correlate an ID number with what your code is doing. While running a task's code, you can query `Task`'s static `CurrentId` property, which returns a nullable `Int32` (`Int32?`). You can also call this from Visual Studio's Watch window or Immediate window while debugging to get the ID for the code that you are currently stepping through. Then you can find your task in the Parallel Tasks/Stacks windows. If you query the `CurrentId` property while a task is not executing, it returns `null`.

During a `Task` object's existence, you can learn where it is in its lifecycle by querying `Task`'s read-only `Status` property. This property returns a `TaskStatus` value that is defined as follows.

```
public enum TaskStatus {
   // These flags indicate the state of a Task during its lifetime:
   Created,             // Task created explicitly; you can manually Start() this task
   WaitingForActivation,// Task created implicitly; it starts automatically

   WaitingToRun,  // The task was scheduled but isn't running yet
   Running,       // The task is actually running

   // The task is waiting for children to complete before it considers itself complete
   WaitingForChildrenToComplete,

   // A task's final state is one of these:
   RanToCompletion,
   Canceled,
   Faulted
}
```

When you first construct a `Task` object, its status is `Created`. Later, when the task is started, its status changes to `WaitingToRun`. When the `Task` is actually running on a thread, its status changes

to Running. When the task stops running and is waiting for any child tasks, the status changes to WaitingForChildrenToComplete. When a task is completely finished, it enters one of three final states: RanToCompletion, Canceled, or Faulted. When a Task<TResult> runs to completion, you can query the task's result via Task<TResult>'s Result property. When a Task or Task<TResult> faults, you can obtain the unhandled exception that the task threw by querying Task's Exception property; which always returns an AggregateException object whose collection contains the set of unhandled exceptions.

For convenience, Task offers several read-only, Boolean properties: IsCanceled, IsFaulted, and IsCompleted. Note that IsCompleted returns true when the Task is in the RanToCompletion, Canceled, or Faulted state. The easiest way to determine if a Task completed successfully is to use code like the following.

```
if (task.Status == TaskStatus.RanToCompletion) ...
```

A Task object is in the WaitingForActivation state if that Task is created by calling one of these functions: ContinueWith, ContinueWhenAll, ContinueWhenAny, or FromAsync. A Task created by constructing a TaskCompletionSource<TResult> object is also created in the WaitingForActivation state. This state means that the Task's scheduling is controlled by the task infrastructure. For example, you cannot explicitly start a Task object that was created by calling ContinueWith. This Task will start automatically when its antecedent task has finished executing.

Task Factories

Occasionally, you might want to create a bunch of Task objects that share the same configuration. To keep you from having to pass the same parameters to each Task's constructor over and over again, you can create a task factory that encapsulates the common configuration. The System.Threading.Tasks namespace defines a TaskFactory type as well as a TaskFactory<TResult> type. Both of these types are derived from System.Object; that is, they are peers of each other.

If you want to create a bunch of tasks that return void, then you will construct a TaskFactory. If you want to create a bunch of tasks that have a specific return type, then you will construct a TaskFactory<TResult> where you pass the task's desired return type for the generic TResult argument. When you create one of these task factory classes, you pass to its constructor the defaults that you want the tasks that the factory creates to have. Specifically, you pass to the task factory the CancellationToken, TaskScheduler, TaskCreationOptions, and TaskContinuationOptions settings that you want factory-created tasks to have.

Here is some sample code demonstrating the use of a TaskFactory.

```
Task parent = new Task(() => {
   var cts = new CancellationTokenSource();
   var tf  = new TaskFactory<Int32>(cts.Token, TaskCreationOptions.AttachedToParent,
      TaskContinuationOptions.ExecuteSynchronously, TaskScheduler.Default);

   // This task creates and starts 3 child tasks
   var childTasks = new[] {
      tf.StartNew(() => Sum(cts.Token, 10000)),
```

```
      tf.StartNew(() => Sum(cts.Token, 20000)),
      tf.StartNew(() => Sum(cts.Token, Int32.MaxValue))  // Too big, throws OverflowException
   };

   // If any of the child tasks throw, cancel the rest of them
   for (Int32 task = 0; task < childTasks.Length; task++)
      childTasks[task].ContinueWith(
         t => cts.Cancel(), TaskContinuationOptions.OnlyOnFaulted);

   // When all children are done, get the maximum value returned from the
   // non-faulting/canceled tasks. Then pass the maximum value to another
   // task that displays the maximum result
   tf.ContinueWhenAll(
      childTasks,
      completedTasks =>
         completedTasks.Where(t => t.Status == TaskStatus.RanToCompletion).Max(t => t.Result),
      CancellationToken.None)
         .ContinueWith(t =>Console.WriteLine("The maximum is: " + t.Result),
            TaskContinuationOptions.ExecuteSynchronously);
});


// When the children are done, show any unhandled exceptions too
parent.ContinueWith(p => {
   // I put all this text in a StringBuilder and call Console.WriteLine just once
   // because this task could execute concurrently with the task above & I don't
   // want the tasks' output interspersed
   StringBuilder sb = new StringBuilder(
      "The following exception(s) occurred:" + Environment.NewLine);

   foreach (var e in p.Exception.Flatten().InnerExceptions)
      sb.AppendLine("   "+ e.GetType().ToString());
   Console.WriteLine(sb.ToString());
}, TaskContinuationOptions.OnlyOnFaulted);

// Start the parent Task so it can start its children
parent.Start();
```

With this code, I am creating a `TaskFactory<Int32>` object that I will use to create three `Task` objects. I want to configure the child tasks all the same way: each `Task` object shares the same `CancellationTokenSource` token, tasks are considered children of their parent, all continue-with tasks created by the `TaskFactory` execute synchronously, and all `Task` objects created by this `TaskFactory` use the default `TaskScheduler`.

Then I create an array consisting of the three child `Task` objects, all created by calling `TaskFactory`'s `StartNew` method. This method conveniently creates and starts each child task. In a loop, I tell each child task that throws an unhandled exception to cancel all the other child tasks that are still running. Finally, using the `TaskFactory`, I call `ContinueWhenAll`, which creates a `Task` that runs when all the child tasks have completed running. Because this task is created with the `TaskFactory`, it will also be considered a child of the parent task and it will execute synchronously using the default `TaskScheduler`. However, I want this task to run even if the other child tasks were canceled, so I override the `TaskFactory`'s `CancellationToken` by passing in `CancellationToken.None`, which prevents this task from being cancelable at all. Finally, when the task that processes all

the results is complete, I create another task that displays the highest value returned from all the child tasks.

Note When calling `TaskFactory`'s or `TaskFactory<TResult>`'s `ContinueWhenAll` and `ContinueWhenAny` methods, the following `TaskContinuationOption` flags are illegal: `NotOnRanToCompletion`, `NotOnFaulted`, and `NotOnCanceled`. And of course, the convenience flags (`OnlyOnCanceled`, `OnlyOnFaulted`, and `OnlyOnRanToCompletion`) are also illegal. That is, `ContinueWhenAll` and `ContinueWhenAny` execute the continue-with task regardless of how the antecedent tasks complete.

Task Schedulers

The task infrastructure is very flexible, and `TaskScheduler` objects are a big part of this flexibility. A `TaskScheduler` object is responsible for executing scheduled tasks and also exposes task information to the Visual Studio debugger. The FCL ships with two `TaskScheduler`-derived types: the thread pool task scheduler and a synchronization context task scheduler. By default, all applications use the thread pool task scheduler. This task scheduler schedules tasks to the thread pool's worker threads and is discussed in more detail in this chapter's "How the Thread Pool Manages Its Threads" section. You can get a reference to the default task scheduler by querying `TaskScheduler`'s static `Default` property.

The synchronization context task scheduler is typically used for applications sporting a graphical user interface, such as Windows Forms, Windows Presentation Foundation (WPF), Silverlight, and Windows Store applications. This task scheduler schedules all tasks onto the application's GUI thread so that all the task code can successfully update UI components like buttons, menu items, and so on. The synchronization context task scheduler does not use the thread pool at all. You can get a reference to a synchronization context task scheduler by querying `TaskScheduler`'s static `FromCurrentSynchronizationContext` method.

Here is a simple Windows Forms application that demonstrates the use of the synchronization context task scheduler.

```
internal sealed class MyForm : Form {
   private readonly TaskScheduler m_syncContextTaskScheduler;
   public MyForm() {
      // Get a reference to a synchronization context task scheduler
      m_syncContextTaskScheduler = TaskScheduler.FromCurrentSynchronizationContext();

      Text = "Synchronization Context Task Scheduler Demo";
      Visible = true; Width = 600; Height = 100;
   }

   private CancellationTokenSource m_cts;

   protected override void OnMouseClick(MouseEventArgs e) {
      if (m_cts != null) { // An operation is in flight, cancel it
         m_cts.Cancel();
```

```
            m_cts = null;
        } else { // An operation is not in flight, start it
            Text = "Operation running";
            m_cts = new CancellationTokenSource();

            // This task uses the default task scheduler and executes on a thread pool thread
            Task<Int32> t = Task.Run(() => Sum(m_cts.Token, 20000), m_cts.Token);

            // These tasks use the sync context task scheduler and execute on the GUI thread
            t.ContinueWith(task => Text = "Result: " + task.Result,
               CancellationToken.None, TaskContinuationOptions.OnlyOnRanToCompletion,
               m_syncContextTaskScheduler);

            t.ContinueWith(task => Text = "Operation canceled",
               CancellationToken.None, TaskContinuationOptions.OnlyOnCanceled,
               m_syncContextTaskScheduler);

            t.ContinueWith(task => Text = "Operation faulted",
               CancellationToken.None, TaskContinuationOptions.OnlyOnFaulted,
               m_syncContextTaskScheduler);
        }
        base.OnMouseClick(e);
    }
}
```

When you click in the client area of this form, a compute-bound task will start executing on a thread pool thread. This is good because the GUI thread is not blocked during this time and can therefore respond to other UI operations. However, the code executed by the thread pool thread should not attempt to update UI components or else an `InvalidOperationException` will be thrown.

When the compute-bound task is done, one of the three continue-with tasks will execute. These tasks are all issued against the synchronization context task scheduler corresponding to the GUI thread, and this task scheduler queues the tasks to the GUI thread, allowing the code executed by these tasks to update UI components successfully. All of these tasks update the form's caption via the inherited `Text` property.

Because the compute-bound work (`Sum`) is running on a thread pool thread, the user can interact with the UI to cancel the operation. In my simple code example, I allow the user to cancel the operation by clicking in the form's client area while an operation is running.

You can, of course, define your own class derived from `TaskScheduler` if you have special task scheduling needs. Microsoft has provided a bunch of sample code for tasks and includes the source code for a bunch of task schedulers in the Parallel Extensions Extras package, which can be downloaded from here: *http://code.msdn.microsoft.com/ParExtSamples*. Here are some of the task schedulers included in this package:

- **IOTaskScheduler** This task scheduler queues tasks to the thread pool's I/O threads instead of its worker threads.
- **LimitedConcurrencyLevelTaskScheduler** This task scheduler allows no more than *n* (a constructor parameter) tasks to execute simultaneously.

- **OrderedTaskScheduler** This task scheduler allows only one task to execute at a time. This class is derived from `LimitedConcurrencyLevelTaskScheduler` and just passes 1 for *n*.
- **PrioritizingTaskScheduler** This task scheduler queues tasks to the CLR's thread pool. After this has occurred, you can call `Prioritize` to indicate that a `Task` should be processed before all normal tasks (if it hasn't been processed already). You can call `Deprioritize` to make a `Task` be processed after all normal tasks.
- **ThreadPerTaskScheduler** This task scheduler creates and starts a separate thread for each task; it does not use the thread pool at all.

`Parallel`'s Static `For`, `ForEach`, and `Invoke` Methods

There are some common programming scenarios that can potentially benefit from the improved performance possible with tasks. To simplify programming, the static `System.Threading.Tasks.Parallel` class encapsulates these common scenarios while using `Task` objects internally. For example, instead of processing all the items in a collection like this.

```
// One thread performs all this work sequentially
for (Int32 i = 0; i < 1000; i++) DoWork(i);
```

you can instead get multiple thread pool threads to assist in performing this work by using the `Parallel` class's `For` method.

```
// The thread pool's threads process the work in parallel
Parallel.For(0, 1000, i => DoWork(i));
```

Similarly, if you have a collection, instead of doing this:

```
// One thread performs all this work sequentially
foreach (var item in collection) DoWork(item);
```

you can do this.

```
// The thread pool's threads process the work in parallel
Parallel.ForEach(collection, item => DoWork(item));
```

If you can use either `For` or `ForEach` in your code, then it is recommended that you use `For` because it executes faster.

And finally, if you have several methods that you need to execute, you could execute them all sequentially, like this:

```
// One thread executes all the methods sequentially
Method1();
Method2();
Method3();
```

or you could execute them in parallel, like this.

```
// The thread pool's threads execute the methods in parallel
Parallel.Invoke(
   () => Method1(),
   () => Method2(),
   () => Method3());
```

All of `Parallel`'s methods have the calling thread participate in the processing of the work, which is good in terms of resource usage because we wouldn't want the calling thread to just suspend itself while waiting for thread pool threads to do all the work. However, if the calling thread finishes its work before the thread pool threads complete their part of the work, then the calling thread will suspend itself until all the work is done, which is also good because this gives you the same semantics as you'd have when using a `for` or `foreach` loop: the thread doesn't continue running until all the work is done. Also note that if any operation throws an unhandled exception, the `Parallel` method you called will ultimately throw an `AggregateException`.

Of course, you should not go through all your source code replacing `for` loops with calls to `Parallel.For` and `foreach` loops with calls to `Parallel.ForEach`. When calling the `Parallel` method, there is an assumption that it is OK for the work items to be performed concurrently. Therefore, do not use the `Parallel` methods if the work must be processed in sequential order. Also, avoid work items that modify any kind of shared data because the data could get corrupted if it is manipulated by multiple threads simultaneously. Normally, you would fix this by adding thread synchronization locks around the data access, but if you do this, then one thread at a time can access the data and you would lose the benefit of processing multiple items in parallel.

In addition, there is overhead associated with the `Parallel` methods; delegate objects have to be allocated, and these delegates are invoked once for each work item. If you have lots of work items that can be processed by multiple threads, then you might gain a performance increase. Also, if you have lots of work to do for each item, then the performance hit of calling through the delegate is negligible. You will actually hurt your performance if you use the `Parallel` methods for just a few work items or for work items that are processed very quickly.

I should mention that `Parallel`'s `For`, `ForEach`, and `Invoke` methods all have overloads that accept a `ParallelOptions` object, which looks like this.

```
public class ParallelOptions{
   public ParallelOptions();

   // Allows cancellation of the operation
   public CancellationToken CancellationToken { get; set; } // Default=CancellationToken.None

   // Allows you to specify the maximum number of work items
   // that can be operated on concurrently
   public Int32 MaxDegreeOfParallelism { get; set; }     // Default=-1 (# of available CPUs)

   // Allows you to specify which TaskScheduler to use
   public TaskScheduler TaskScheduler { get; set; }     // Default=TaskScheduler.Default
}
```

In addition, there are overloads of the `For` and `ForEach` methods that let you pass three delegates:

- The task local initialization delegate (`localInit`) is invoked once for each task participating in the work. This delegate is invoked before the task is asked to process a work item.
- The body delegate (`body`) is invoked once for each item being processed by the various threads participating in the work.
- The task local finally delegate (`localFinally`) is invoked once for each task participating in the work. This delegate is invoked after the task has processed all the work items that will be dispatched to it. It is even invoked if the body delegate code experiences an unhandled exception.

Here is some sample code that demonstrates the use of the three delegates by adding up the bytes for all files contained within a directory.

```
private static Int64 DirectoryBytes(String path, String searchPattern,
   SearchOption searchOption) {
   var files = Directory.EnumerateFiles(path, searchPattern, searchOption);
   Int64 masterTotal = 0;

   ParallelLoopResult result = Parallel.ForEach<String, Int64>(
      files,

      () => { // localInit: Invoked once per task at start
         // Initialize that this task has seen 0 bytes
         return 0;   // Set taskLocalTotal initial value to 0
      },

      (file, loopState, index, taskLocalTotal) => { // body: Invoked once per work item
         // Get this file's size and add it to this task's running total
         Int64 fileLength = 0;
         FileStream fs = null;
         try {
            fs = File.OpenRead(file);
            fileLength = fs.Length;
         }
         catch (IOException) { /* Ignore any files we can't access */ }
         finally { if (fs != null) fs.Dispose(); }
         return taskLocalTotal + fileLength;
      },

      taskLocalTotal => { // localFinally: Invoked once per task at end
         // Atomically add this task's total to the "master" total
         Interlocked.Add(ref masterTotal, taskLocalTotal);
      });

   return masterTotal;
}
```

Each task maintains its own running total (in the `taskLocalTotal` variable) for the files that it is given. As each task completes its work, the master total is updated in a thread-safe way by calling the `Interlocked.Add` method (discussed in Chapter 29, "Primitive Thread Synchronization Constructs"). Because each task has its own running total, no thread synchronization is required during the processing of the items. Because thread synchronization would hurt performance, not requiring thread synchronization is good. It's only after each task returns that `masterTotal` has to be updated in a thread-safe way, so the performance hit of calling `Interlocked.Add` occurs only once per task instead of once per work item.

You'll notice that the body delegate is passed a `ParallelLoopState` object, which looks like this.

```
public class ParallelLoopState{
   public void Stop();
   public Boolean IsStopped { get; }

   public void Break();
   public Int64? LowestBreakIteration{ get; }

   public Boolean IsExceptional { get; }
   public Boolean ShouldExitCurrentIteration { get; }
}
```

Each task participating in the work gets its own `ParallelLoopState` object, and it can use this object to interact with the other task participating in the work. The `Stop` method tells the loop to stop processing any more work, and future querying of the `IsStopped` property will return `true`. The `Break` method tells the loop to stop processing any items that are beyond the current item. For example, let's say that `ForEach` is told to process 100 items and `Break` is called while processing the fifth item, then the loop will make sure that the first five items are processed before `ForEach` returns. Note, however, that additional items may have been processed. The `LowestBreakIteration` property returns the lowest item number whose processing called the `Break` method. The `LowestBreakIteration` property returns `null` if `Break` was never called.

The `IsException` property returns `true` if the processing of any item resulted in an unhandled exception. If the processing of an item takes a long time, your code can query the `ShouldExitCurrentIteration` property to see if it should exit prematurely. This property returns `true` if `Stop` was called, `Break` was called, the `CancellationTokenSource` (referred to by the `ParallelOption`'s `CancellationToken` property) is canceled, or if the processing of an item resulted in an unhandled exception.

`Parallel`'s `For` and `ForEach` methods both return a `ParallelLoopResult` instance, which looks like this.

```
public struct ParallelLoopResult {
   // Returns false if the operation was ended prematurely
   public Boolean IsCompleted { get; }
   public Int64? LowestBreakIteration{ get; }
}
```

You can examine the properties to determine the result of the loop. If `IsCompleted` returns `true`, then the loop ran to completion and all the items were processed. If `IsCompleted` is `false` and `LowestBreakIteration` is `null`, then some thread participating in the work called the `Stop` method. If `IsCompleted` is `false` and `LowestBreakIteration` is not `null`, then some thread participating in the work called the `Break` method and the `Int64` value returned from `LowestBreakIteration` indicates the index of the lowest item guaranteed to have been processed. If an exception is thrown, then you should catch an `AggregateException` in order to recover gracefully.

Parallel Language Integrated Query

Microsoft's Language Integrated Query (LINQ) feature offers a convenient syntax for performing queries over collections of data. Using LINQ, you can easily filter items, sort items, return a projected set of items, and much more. When you use LINQ to Objects, only one thread processes all the items in your data collection sequentially; we call this a *sequential query*. You can potentially improve the performance of this processing by using Parallel LINQ, which can turn your sequential query into a *parallel query*, which internally uses tasks (queued to the default `TaskScheduler`) to spread the processing of the collection's items across multiple CPUs so that multiple items are processed concurrently. Like `Parallel`'s methods, you will get the most benefit from Parallel LINQ if you have many items to process or if the processing of each item is a lengthy compute-bound operation.

The static `System.Linq.ParallelEnumerable` class (defined in `System.Core.dll`) implements all of the Parallel LINQ functionality, and so you must import the `System.Linq` namespace into your source code via C#'s `using` directive. In particular, this class exposes parallel versions of all the standard LINQ operators such as `Where`, `Select`, `SelectMany`, `GroupBy`, `Join`, `OrderBy`, `Skip`, `Take`, and so on. All of these methods are extension methods that extend the `System.Linq.ParallelQuery<T>` type. To have your LINQ to Objects query invoke the parallel versions of these methods, you must convert your sequential query (based on `IEnumerable` or `IEnumerable<T>`) to a parallel query (based on `ParallelQuery` or `ParallelQuery<T>`) using `ParallelEnumerable`'s `AsParallel` extension method, which looks like this.[3]

```
public static ParallelQuery<TSource> AsParallel<TSource>(this IEnumerable<TSource> source)
public static ParallelQuery          AsParallel(this IEnumerable source)
```

Here is an example of a sequential query that has been converted to a parallel query. This query returns all the obsolete methods defined within an assembly.

```
private static void ObsoleteMethods(Assembly assembly) {
   var query =
      from type in assembly.GetExportedTypes().AsParallel()

      from method in type.GetMethods(BindingFlags.Public |
         BindingFlags.Instance | BindingFlags.Static)

      let obsoleteAttrType = typeof(ObsoleteAttribute)
```

[3] The `ParallelQuery<T>` class is derived from the `ParallelQuery` class.

```
      where Attribute.IsDefined(method, obsoleteAttrType)

      orderby type.FullName

      let obsoleteAttrObj = (ObsoleteAttribute)
         Attribute.GetCustomAttribute(method, obsoleteAttrType)

      select String.Format("Type={0}\nMethod={1}\nMessage={2}\n",
            type.FullName, method.ToString(), obsoleteAttrObj.Message);

   // Display the results
   foreach (var result in query) Console.WriteLine(result);
}
```

Although uncommon, within a query you can switch from performing parallel operations back to performing sequential operations by calling `ParallelEnumerable`'s `AsSequential` method.

```
public static IEnumerable<TSource> AsSequential<TSource>(this ParallelQuery<TSource> source)
```

This method basically turns a `ParallelQuery<T>` back to an `IEnumerable<T>` so that operations performed after calling `AsSequential` are performed by just one thread.

Normally, the resulting data produced by a LINQ query is evaluated by having some thread execute a `foreach` statement (as shown earlier). This means that just one thread iterates over all the query's results. If you want to have the query's results processed in parallel, then you should process the resulting query by using `ParallelEnumerable`'s `ForAll` method.

```
static void ForAll<TSource>(this ParallelQuery<TSource> source, Action<TSource> action)
```

This method allows multiple threads to process the results simultaneously. I could modify my code earlier to use this method as follows.

```
// Display the results
query.ForAll(Console.WriteLine);
```

However, having multiple threads call `Console.WriteLine` simultaneously actually hurts performance, because the `Console` class internally synchronizes threads, ensuring that only one at a time can access the console window. This prevents text from multiple threads from being interspersed, making the output unintelligible. Use the `ForAll` method when you intend to perform calculations on each result.

Because Parallel LINQ processes items by using multiple threads, the items are processed concurrently and the results are returned in an unordered fashion. If you need to have Parallel LINQ preserve the order of items as they are processed, then you can call `ParallelEnumerable`'s `AsOrdered` method. When you call this method, threads will process items in groups and then the groups are merged back together, preserving the order; this will hurt performance. The following operators produce unordered operations: `Distinct`, `Except`, `Intersect`, `Union`, `Join`, `GroupBy`,

GroupJoin, and ToLookup. If you want to enforce ordering again after one of these operators, just call the AsOrdered method.

The following operators produce ordered operations: OrderBy, OrderByDescending, ThenBy, and ThenByDescending. If you want to go back to unordered processing again to improve performance after one of these operators, just call the AsUnordered method.

Parallel LINQ offers some additional ParallelEnumerable methods that you can call to control how the query is processed.

```
public static ParallelQuery<TSource> WithCancellation<TSource>(
   this ParallelQuery<TSource> source, CancellationTokencancellationToken)

public static ParallelQuery<TSource> WithDegreeOfParallelism<TSource>(
   this ParallelQuery<TSource> source, Int32degreeOfParallelism)

public static ParallelQuery<TSource> WithExecutionMode<TSource>(
   this ParallelQuery<TSource> source, ParallelExecutionModeexecutionMode)

public static ParallelQuery<TSource> WithMergeOptions<TSource>(
   this ParallelQuery<TSource> source, ParallelMergeOptionsmergeOptions)
```

Obviously, the WithCancellation method allows you to pass a CancellationToken so that the query processing can be stopped prematurely. The WithDegreeOfParallelism method specifies the maximum number of threads allowed to process the query; it does not force the threads to be created if not all of them are necessary. Usually you will not call this method, and, by default, the query will execute using one thread per core. However, you could call WIthDegreeOfParallelism, passing a number that is smaller than the number of available cores if you want to keep some cores available for doing other work. You could also pass a number that is greater than the number of cores if the query performs synchronous I/O operations because threads will be blocking during these operations. This wastes more threads but can produce the final result in less time. You might consider doing this in a client application, but I'd highly recommend against performing synchronous I/O operations in a server application.

Parallel LINQ analyzes a query and then decides how to best process it. Sometimes processing a query sequentially yields better performance. This is usually true when using any of these operations: Concat, ElementAt(OrDefault), First(OrDefault), Last(OrDefault), Skip(While), Take(While), or Zip. It is also true when using overloads of Select(Many) or Where that pass a position index into your selector or predicate delegate. However, you can force a query to be processed in parallel by calling WithExecutionMode, passing it one of the ParallelExecutionMode flags.

```
public enum ParallelExecutionMode {
   Default = 0,         // Let Parallel LINQ decide to best process the query
   ForceParallelism = 1 // Force the query to be processed in parallel
}
```

As mentioned before, Parallel LINQ has multiple threads processing items, and then the results must be merged back together. You can control how the items are buffered and merged by calling `WithMergeOptions`, passing it one of the `ParallelMergeOptions` flags.

```
public enum ParallelMergeOptions {
   Default       = 0,    // Same as AutoBuffered today (could change in the future)
   NotBuffered   = 1,    // Results are processed as ready
   AutoBuffered  = 2,    // Each thread buffers some results before processed
   FullyBuffered = 3     // Each thread buffers all results before processed
}
```

These options basically give you some control over speed versus memory consumption. `NotBuffered` saves memory but processes items slower. `FullyBuffered` consumes more memory while running fastest. `AutoBuffered` is the compromise in between `NotBuffered` and `FullyBuffered`. Really, the best way to know which of these to choose for any given query is to try them all and compare their performance results, or just accept the default, which tends to work pretty well for many queries. See the following blog posts for more information about how Parallel LINQ partitions work across CPU cores:

- *http://blogs.msdn.com/pfxteam/archive/2009/05/28/9648672.aspx*
- *http://blogs.msdn.com/pfxteam/archive/2009/06/13/9741072.aspx*

Performing a Periodic Compute-Bound Operation

The `System.Threading` namespace defines a `Timer` class, which you can use to have a thread pool thread call a method periodically. When you construct an instance of the `Timer` class, you are telling the thread pool that you want a method of yours called back at a future time that you specify. The `Timer` class offers several constructors, all quite similar to each other.

```
public sealed class Timer : MarshalByRefObject, IDisposable {
   public Timer(TimerCallback callback, Object state, Int32    dueTime, Int32    period);
   public Timer(TimerCallback callback, Object state, UInt32   dueTime, UInt32   period);
   public Timer(TimerCallback callback, Object state, Int64    dueTime, Int64    period);
   public Timer(TimerCallback callback, Object state, Timespan dueTime, TimeSpan period);
}
```

All four constructors construct a `Timer` object identically. The `callback` parameter identifies the method that you want called back by a thread pool thread. Of course, the callback method that you write must match the `System.Threading.TimerCallback` delegate type, which is defined as follows.

```
delegate void TimerCallback(Object state);
```

The constructor's `state` parameter allows you to pass state data to the callback method each time it is invoked; you can pass `null` if you have no state data to pass. You use the `dueTime` parameter to tell the CLR how many milliseconds to wait before calling your callback method for the very first time. You can specify the number of milliseconds by using a signed or unsigned 32-bit value, a signed 64-bit value, or a `TimeSpan` value. If you want the callback method called immediately, specify 0 for the `dueTime` parameter. The last parameter, `period`, allows you to specify how long, in milliseconds, to wait before each successive call to the callback method. If you pass `Timeout.Infinite` (–1) for this parameter, a thread pool thread will call the callback method just once.

Internally, the thread pool has just one thread that it uses for all `Timer` objects. This thread knows when the next `Timer` object's time is due. When the next `Timer` object is due, the thread wakes up, and internally calls `ThreadPool`'s `QueueUserWorkItem` to enter an entry into the thread pool's queue, causing your callback method to get called. If your callback method takes a long time to execute, the timer could go off again. This could cause multiple thread pool threads to be executing your callback method simultaneously. To work around this problem, I recommend the following: construct the `Timer` specifying `Timeout.Infinite` for the `period` parameter. Now, the timer will fire only once. Then, in your callback method, call the `Change` method specifying a new due time and again specify `Timeout.Infinite` for the `period` parameter. Here is what the `Change` method overloads look like.

```
public sealed class Timer : MarshalByRefObject, IDisposable {
   public Boolean Change(Int32    dueTime, Int32    period);
   public Boolean Change(UInt32   dueTime, UInt32   period);
   public Boolean Change(Int64    dueTime, Int64    period);
   public Boolean Change(TimeSpan dueTime, TimeSpan period);
}
```

The `Timer` class also offers a `Dispose` method that allows you to cancel the timer altogether and optionally signal the kernel object identified by the `notifyObject` parameter when all pending callbacks for the time have completed. Here is what the `Dispose` method overloads look like.

```
public sealed class Timer : MarshalByRefObject, IDisposable {
   public Boolean Dispose();
   public Boolean Dispose(WaitHandle notifyObject);
}
```

Important When a `Timer` object is garbage collected, its finalization code tells the thread pool to cancel the timer so that it no longer goes off. So when using a `Timer` object, make sure that a variable is keeping the `Timer` object alive or else your callback method will stop getting called. This is discussed and demonstrated in the "Garbage Collections and Debugging" section in Chapter 21, "The Managed Heap and Garbage Collection."

The following code demonstrates how to have a thread pool thread call a method starting immediately and then every two seconds thereafter.

```
internal static class TimerDemo {
   private static Timer s_timer;

   public static void Main() {
      Console.WriteLine("Checking status every 2 seconds");

      // Create the Timer ensuring that it never fires. This ensures that
      // s_timer refers to it BEFORE Status is invoked by a thread pool thread
      s_timer = new Timer(Status, null, Timeout.Infinite, Timeout.Infinite);

      // Now that s_timer is assigned to, we can let the timer fire knowing
      // that calling Change in Status will not throw a NullReferenceException
      s_timer.Change(0, Timeout.Infinite);

      Console.ReadLine();   // Prevent the process from terminating
   }

   // This method's signature must match the TimerCallback delegate
   private static void Status(Object state) {
      // This method is executed by a thread pool thread
      Console.WriteLine("In Status at {0}", DateTime.Now);
      Thread.Sleep(1000);  // Simulates other work (1 second)

      // Just before returning, have the Timer fire again in 2 seconds
      s_timer.Change(2000, Timeout.Infinite);

      // When this method returns, the thread goes back
      // to the pool and waits for another work item
   }
}
```

If you have an operation you want performed periodically, there is another way you can structure your code by taking advantage of `Task`'s static `Delay` method along with C#'s `async` and `await` keywords (discussed extensively in Chapter 28). Here is a rewrite of the preceding code demonstrating this.

```
internal static class DelayDemo {
   public static void Main() {
      Console.WriteLine("Checking status every 2 seconds");
      Status();
      Console.ReadLine();   // Prevent the process from terminating
   }

   // This method can take whatever parameters you desire
   private static async void Status() {
      while (true) {
         Console.WriteLine("Checking status at {0}", DateTime.Now);
         // Put code to check status here...

         // At end of loop, delay 2 seconds without blocking a thread
         await Task.Delay(2000); // await allows thread to return
         // After 2 seconds, some thread will continue after await to loop around
      }
   }
}
```

So Many Timers, So Little Time

Unfortunately, the FCL actually ships with several timers, and it is not clear to most programmers what makes each timer unique. Let me attempt to explain:

- **System.Threading's Timer class** This is the timer discussed in the previous section, and it is the best timer to use when you want to perform periodic background tasks on a thread pool thread.
- **System.Windows.Forms's Timer class** Constructing an instance of this class tells Windows to associate a timer with the calling thread (see the Win32 `SetTimer` function). When this timer goes off, Windows injects a timer message (`WM_TIMER`) into the thread's message queue. The thread must execute a message pump that extracts these messages and dispatches them to the desired callback method. Notice that all of the work is done by just one thread—the thread that sets the timer is guaranteed to be the thread that executes the callback method. This also means that your timer method will not be executed by multiple threads concurrently.
- **System.Windows.Threading's DispatcherTimer class** This class is the equivalent of the `System.Windows.Forms`'s `Timer` class for Silverlight and WPF applications.
- **Windows.UI.Xaml's DispatcherTimer class** This class is the equivalent of the `System.Windows.Forms`'s `Timer` class for Windows Store apps.
- **System.Timers's Timer class** This timer is basically a wrapper around `System.Threading`'s `Timer` class that causes the CLR to queue events into the thread pool when the timer comes due. The `System.Timers.Timer` class is derived from `System.ComponentModel`'s `Component` class, which allows these timer objects to be placed on a design surface in Visual Studio. Also, it exposes properties and events, allowing it to be used more easily from Visual Studio's designer. This class was added to the FCL years ago while Microsoft was still sorting out the threading and timer stuff. This class probably should have been removed so that everyone would be using the `System.Threading.Timer` class instead. In fact, I never use the `System.Timers.Timer` class, and I'd discourage you from using it, too, unless you really want a timer on a design surface.

How the Thread Pool Manages Its Threads

Now I'd like to talk about how the thread pool code manages worker and I/O threads. However, I don't want to go into a lot of detail, because the internal implementation has changed greatly over the years with each version of the CLR, and it will continue changing with future versions. It is best to think of the thread pool as a black box. The black box is not perfect for any one application, because it is a general purpose thread-scheduling technology designed to work with a large myriad of applications; it will work better for some applications than for others. It works very well today, and I highly recommend that you trust it, because it would be very hard for you to produce a thread pool that works better than the one shipping in the CLR. And, over time, most applications should improve as the thread pool code internally changes how it manages threads.

Setting Thread Pool Limits

The CLR allows developers to set a maximum number of threads that the thread pool will create. However, it turns out that thread pools should never place an upper limit on the number of threads in the pool because starvation or deadlock might occur. Imagine queuing 1,000 work items that all block on an event that is signaled by the 1,001st item. If you've set a maximum of 1,000 threads, the 1,001st work item won't be executed, and all 1,000 threads will be blocked forever, forcing end users to terminate the application and lose all their work. Also, it is very unusual for developers to artificially limit the resources that they have available to their application. For example, would you ever start your application and tell the system you'd like to restrict the amount of memory that the application can use or limit the amount of network bandwidth that your application can use? Yet, for some reason, developers feel compelled to limit the number of threads that the thread pool can have.

Because customers have had starvation and deadlock issues, the CLR team has steadily increased the default maximum number of threads that the thread pool can have. The default maximum is now about 1,000 threads, which is effectively limitless because a 32-bit process has at most 2 GB of usable address space within it. After a bunch of Win32 DLLs load, the CLR DLLs load, the native heap and the managed heap is allocated, there is approximately 1.5 GB of address space left over. Because each thread requires more than 1 MB of memory for its user-mode stack and thread environment block (TEB), the most threads you can get in a 32-bit process is about 1,360. Attempting to create more threads than this will result in an `OutOfMemoryException` being thrown. Of course, a 64-bit process offers 8 terabytes of address space, so you could theoretically create hundreds of thousands of threads. But allocating anywhere near this number of threads is really just a waste of resources, especially when the ideal number of threads to have is equal to the number of CPUs in the machine. What the CLR team should do is remove the limits entirely, but they can't do this now because doing so might break some applications that expect thread pool limits to exist.

The `System.Threading.ThreadPool` class offers several static methods that you can call to manipulate the number of threads in the thread pool: `GetMaxThreads`, `SetMaxThreads`, `GetMinThreads`, `SetMinThreads`, and `GetAvailableThreads`. I highly recommend that you do not call any of these methods. Playing with thread pool limits usually results in making an application perform worse, not better. If you think that your application needs hundreds or thousands of threads, there is something seriously wrong with the architecture of your application and the way that it's using threads. This chapter and Chapter 28 demonstrate the proper way to use threads.

How Worker Threads Are Managed

Figure 27-1 shows the various data structures that make up the worker threads' part of the thread pool. The `ThreadPool.QueueUserWorkItem` method and the `Timer` class always queue work items to the global queue. Worker threads pull items from this queue using a first-in-first-out (FIFO) algorithm and process them. Because multiple worker threads can be removing items from the global queue simultaneously, all worker threads contend on a thread synchronization lock to ensure that two or more threads don't take the same work item. This thread synchronization lock can become a bottleneck in some applications, thereby limiting scalability and performance to some degree.

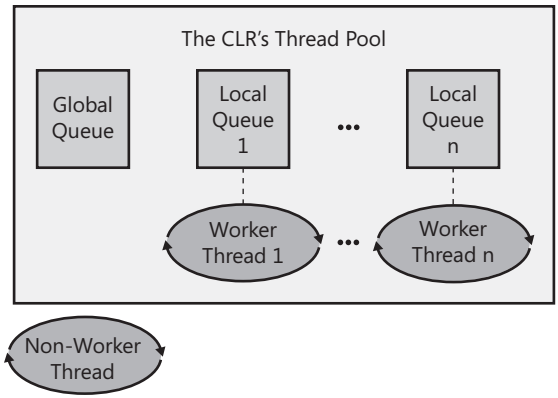

FIGURE 27-1 The CLR's thread pool.

Now let's talk about `Task` objects scheduled using the default `TaskScheduler` (obtained by querying `TaskScheduler`'s static `Default` property).[4] When a non-worker thread schedules a `Task`, the `Task` is added to the global queue. But, each worker thread has its own local queue, and when a worker thread schedules a `Task`, the `Task` is added to calling the thread's local queue.

When a worker thread is ready to process an item, it always checks its local queue for a `Task` first. If a `Task` exists, the worker thread removes the `Task` from its local queue and processes the item. Note that a worker thread pulls tasks from its local queue by using a last-in-first-out (LIFO) algorithm. Because a worker thread is the only thread allowed to access the head of its own local queue, no thread synchronization lock is required and adding and removing `Task`s from the queue is very fast. A side effect of this behavior is that `Task`s are executed in the reverse order that they were queued.

Important Thread pools have never guaranteed the order in which queued items are processed, especially because multiple threads could be processing items simultaneously. However, this side effect exacerbates the problem. You must make sure that your application has no expectations about the order in which queued work items or tasks execute.

If a worker thread sees that its local queue is empty, then the worker thread will attempt to steal a `Task` from another worker thread's local queue. `Task`s are stolen from the tail of a local queue and require that a thread synchronization lock be taken, which hurts performance a little bit. Of course, the hope is that stealing rarely occurs, so this lock is taken rarely. If all the local queues are empty, then the worker thread will extract an item from the global queue (taking its lock) using the FIFO algorithm. If the global queue is empty, then the worker thread puts itself to sleep waiting for something to show up. If it sleeps for a long time, then it will wake itself up and destroy itself, allowing the system to reclaim the resources (kernel object, stacks, TEB) that were used by the thread.

[4] Other `TaskScheduler`-derived objects may exhibit behavior different from what I describe here.

The thread pool will quickly create worker threads so that the number of worker threads is equal to the value pass to `ThreadPool`'s `SetMinThreads` method. If you never call this method (and it's recommended that you never call this method), then the default value is equal to the number of CPUs that your process is allowed to use as determined by your process's affinity mask. Usually your process is allowed to use all the CPUs on the machine, so the thread pool will quickly create worker threads up to the number of CPUs on the machine. After this many threads have been created, the thread pool monitors the completion rate of work items and if items are taking a long time to complete (the meaning of which is not documented), it creates more worker threads. If items start completing quickly, then worker threads will be destroyed.