CHAPTER 26

Thread Basics

In this chapter:

In this chapter, I introduce the basic concepts concerning threads, and I offer a way for developers to conceptualize about them and their use. I'll explain why Windows introduced the concept of threads, CPU trends, the relationship between common language runtime (CLR) threads and Windows threads, the overhead associated with using threads, how Windows schedules threads, the Microsoft .NET Framework classes that expose thread properties, and much more.

The chapters in Part V, "Threading," of this book explain how Windows and the CLR work together to provide a threading architecture. It is my hope that after reading these chapters, you will take away a foundation of knowledge that will allow you to effectively use threads to design and build responsive, reliable, and scalable applications and components.

Why Does Windows Support Threads?

Back in the early days of computers, operating systems didn't offer the concept of a thread. In effect, there was just one thread of execution that ran throughout the entire system, which included both operating system code and application code. The problem with having only one thread of execution was that a long-running task would prevent other tasks from executing. For example, in the days of 16-bit Windows, it was very common for an application that was printing a document to stall the entire machine, causing the operating system and all other applications to stop responding. And, sometimes applications would have a bug in them, resulting in an infinite loop that also stopped the entire machine from operating.

At this point, the end user would have no choice but to reboot the computer by pressing the reset button or power switch. Of course, end users hated doing this (they still do, in fact) because all running applications terminated; more importantly, any data that these applications were processing was thrown out of memory and lost. Microsoft knew that 16-bit Windows would not be a good enough operating system to keep Microsoft relevant as the computer industry progressed, so they set out to build a new operating system to address the needs of corporations and individuals. This new operating system had to be robust, reliable, scalable, and secure, and it had to improve the many deficiencies of 16-bit Windows. This operating system kernel originally shipped in Windows NT. Over the years, this kernel has had many tweaks and features added to it. The latest version of this kernel ships in the latest versions of the Microsoft client and server Windows operating systems.

When Microsoft was designing this operating system kernel, they decided to run each instance of an application in what is called a *process*. A process is just a collection of resources that is used by a single instance of an application. Each process is given a virtual address space, ensuring that the code and data used by one process is not accessible to another process. This makes application instances robust because one process cannot corrupt code or data being used by another. In addition, the operating system's kernel code and data are not accessible to processes; therefore, it's not possible for application code to corrupt operating system code or data. So now, application code cannot corrupt other applications or the operating system itself, and the whole computing experience is much better for end users. In addition, the system is more secure because application code cannot access user names, passwords, credit card information, or other sensitive information that is in use by another application or the operating system itself.

This is all well and good, but what about the CPU itself? What if an application enters an infinite loop? Well, if there is only one CPU in the machine, then it executes the infinite loop and cannot execute anything else, so although the data cannot be corrupted and is more secure, the system could still stop responding to the end user. Microsoft needed to fix this problem, too, and threads were the answer. A *thread* is a Windows concept whose job is to virtualize the CPU. Windows gives each process its very own thread (which functions similar to a CPU), and if application code enters an infinite loop, the process associated with that code freezes up, but other processes (which have their own threads) are not frozen; they keep running!

Thread Overhead

Threads are awesome because they enable Windows to be responsive even when applications are executing long-running tasks. Also, threads allow the user to use one application (like Task Manager) to forcibly kill an application that appears frozen because it is executing a long-running task. But as with every virtualization mechanism, threads have space (memory consumption) and time (runtime execution performance) overhead associated with them.

Let's explore this overhead in more detail now. Every thread has one of each of the following:

- **Thread kernel object** The operating system allocates and initializes one of these data structures for each thread created in the system. The data structure contains a bunch of properties (discussed later in this chapter) that describe the thread. This data structure also contains what is called the thread's context. The context is a block of memory that contains a set of the CPU's registers. For the x86, x64, and ARM CPU architectures, the thread's context uses approximately 700, 1,240, or 350 bytes of memory, respectively.

- **Thread environment block (TEB)** The TEB is a block of memory allocated and initialized in user mode (address space that application code can quickly access). The TEB consumes 1 page of memory (4 KB on x86, x64 CPUs, and ARM CPUs). The TEB contains the head of the thread's exception-handling chain. Each `try` block that the thread enters inserts a node in the head of this chain; the node is removed from the chain when the thread exits the `try` block. In addition, the TEB contains the thread's thread-local storage data and some data structures for use by Graphics Device Interface (GDI) and OpenGL graphics.

- **User-mode stack** The user-mode stack is used for local variables and arguments passed to methods. It also contains the address indicating what the thread should execute next when the current method returns. By default, Windows allocates 1 MB of memory for each thread's user-mode stack. More specifically, Windows reserves the 1 MB of address space and sparsely commits physical storage to it as the thread actually requires it when growing the stack.

- **Kernel-mode stack** The kernel-mode stack is also used when application code passes arguments to a kernel-mode function in the operating system. For security reasons, Windows copies any arguments passed from user-mode code to the kernel from the thread's user-mode stack to the thread's kernel-mode stack. Once copied, the kernel can verify the arguments' values, and because the application code can't access the kernel-mode stack, the application can't modify the arguments' values after they have been validated and the operating system kernel code begins to operate on them. In addition, the kernel calls methods within itself and uses the kernel-mode stack to pass its own arguments, to store a function's local variables, and to store return addresses. The kernel-mode stack is 12 KB when running on a 32-bit Windows system and 24 KB when running on a 64-bit Windows system.

- **DLL thread-attach and thread-detach notifications** Windows has a policy that whenever a thread is created in a process, all unmanaged DLLs loaded in that process have their `DllMain` method called, passing a `DLL_THREAD_ATTACH` flag. Similarly, whenever a thread dies, all DLLs in the process have their `DllMain` method called, passing it a `DLL_THREAD_DETACH` flag. Some DLLs need these notifications to perform some special initialization or cleanup for each thread created/destroyed in the process. For example, the C-Runtime library DLL allocates some thread-local storage state that is required should the thread use functions contained within the C-Runtime library.

In the early days of Windows, many processes had maybe 5 or 6 DLLs loaded into them, but today, some processes have several hundred DLLs loaded into them. Right now, on my machine, Microsoft Visual Studio has about 470 DLLs loaded into its process address space! This means that whenever a new thread is created in Visual Studio, 470 DLL functions must get called before the thread is allowed to do what it was created to do. And these 470 functions must be called again whenever a thread in Visual Studio dies. Wow—this can seriously affect the performance of creating and destroying threads within a process.[1]

So now, you see all the space and time overhead that is associated with creating a thread, letting it sit around in the system, and destroying it. But the situation gets even worse—now we're going to start talking about *context switching*. A computer with only one CPU in it can do only one thing at a time. Therefore, Windows has to share the actual CPU hardware among all the threads (logical CPUs) that are sitting around in the system.

At any given moment in time, Windows assigns one thread to a CPU. That thread is allowed to run for a time-slice (sometimes referred to as a quantum). When the time-slice expires, Windows context switches to another thread. Every context switch requires that Windows performs the following actions:

1. Save the values in the CPU's registers to the currently running thread's context structure inside the thread's kernel object.
2. Select one thread from the set of existing threads to schedule next. If this thread is owned by another process, then Windows must also switch the virtual address space seen by the CPU before it starts executing any code or touching any data.
3. Load the values in the selected thread's context structure into the CPU's registers.

After the context switch is complete, the CPU executes the selected thread until its time-slice expires, and then another context switch happens again. Windows performs context switches about every 30 ms. Context switches are pure overhead; that is, there is no memory or performance benefit that comes from context switches. Windows performs context switching to provide end users with a robust and responsive operating system.

Now, if an application's thread enters into an infinite loop, Windows will periodically preempt that thread, assign a different thread to an actual CPU, and let this other thread run for a while. This other thread could be Task Manager's thread and now, the end user can use Task Manager to kill the process containing the thread that is in an infinite loop. When doing this, the process dies and all the data it was working on is destroyed, too, but all other processes in the system continue to run just fine without losing their data. Of course, the user doesn't have to reset the machine and reboot, so context switches are required to provide end users with a much better overall experience at the cost of performance.

[1] DLLs produced by C# and most other managed programming languages do not have a `DllMain` in them at all and so managed DLLs will not receive the `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH` notifications improving performance. In addition, unmanaged DLLs can opt out of these notifications by calling the Win32 `DisableThreadLibraryCalls` function. Unfortunately, many unmanaged developers are not aware of this function, so they don't call it.

In fact, the performance hit is much worse than you might think. Yes, a performance hit occurs when Windows context switches to another thread. But the CPU was executing another thread, and the previously running thread's code and data reside in the CPU's caches so that the CPU doesn't have to access RAM memory as much, which has significant latency associated with it. When Windows context switches to a new thread, this new thread is most likely executing different code and accessing different data that is not in the CPU's cache. The CPU must access RAM memory to populate its cache so it can get back to a good execution speed. But then, about 30 ms later, another context switch occurs.

The time required to perform a context switch varies with different CPU architectures and speed. And the time required to build up a CPU's cache depends on what applications are running in the system, the size of the CPU's caches, and various other factors. So it is impossible for me to give you an absolute figure or even an estimate as to what time overhead is incurred for each context switch. Suffice it to say that you want to avoid using context switches as much as possible if you are interested in building high-performing applications and components.

Important At the end of a time-slice, if Windows decides to schedule the same thread again (rather than switching to another thread), then Windows does not perform a context switch. Instead, the thread is allowed to just continue running. This improves performance significantly, and avoiding context switches is something you want to achieve as often as possible when you design your code.

Important A thread can voluntarily end its time-slice early, which happens quite frequently. Threads typically wait for I/O operations (keyboard, mouse, file, network, etc.) to complete. For example, Notepad's thread usually sits idle with nothing to do; this thread is waiting for input. If the user presses the J key on the keyboard, Windows wakes Notepad's thread to have it process the J keystroke. It may take Notepad's thread just 5 ms to process the key, and then it calls a Win32 function that tells Windows that it is ready to process the next input event. If there are no more input events, then Windows puts Notepad's thread into a wait state (relinquishing the remainder of its time-slice) so that the thread is not scheduled on any CPU until the next input stimulus occurs. This improves overall system performance because threads that are waiting for I/O operations to complete are not scheduled on a CPU and do not waste CPU time; other threads can be scheduled on the CPU instead.

In addition, when performing a garbage collection, the CLR must suspend all the threads, walk their stacks to find the roots to mark objects in the heap, walk their stacks again (updating roots to objects that moved during compaction), and then resume all the threads. So avoiding threads will greatly improve the performance of the garbage collector, too. And whenever you are using a debugger, Windows suspends all threads in the application being debugged every time a breakpoint is hit and resumes all the threads when you single-step or run the application. So the more threads you have, the slower your debugging experience will be.

From this discussion, you should conclude that you must avoid using threads as much as possible because they consume a lot of memory and they require time to create, destroy, and manage. Time is also wasted when Windows context switches between threads and when garbage collections occur. However, this discussion should also help you realize that threads must be used sometimes because they allow Windows to be robust and responsive.

I should also point out that a computer with multiple CPUs in it can actually run multiple threads simultaneously, increasing scalability (the ability to do more work in less time). Windows will assign one thread to each CPU core, and each core will perform its own context switching to other threads. Windows makes sure that a single thread is not scheduled on multiple cores at one time because this would wreak havoc. Today, computers that contain multiple CPUs, hyperthreaded CPUs, or multi-core CPUs are commonplace. But when Windows was originally designed, single-CPU computers were commonplace, and Windows added threads to improve system responsiveness and reliability. Today, threads are also being used to improve scalability, which can happen only on computers that have multiple cores in them.

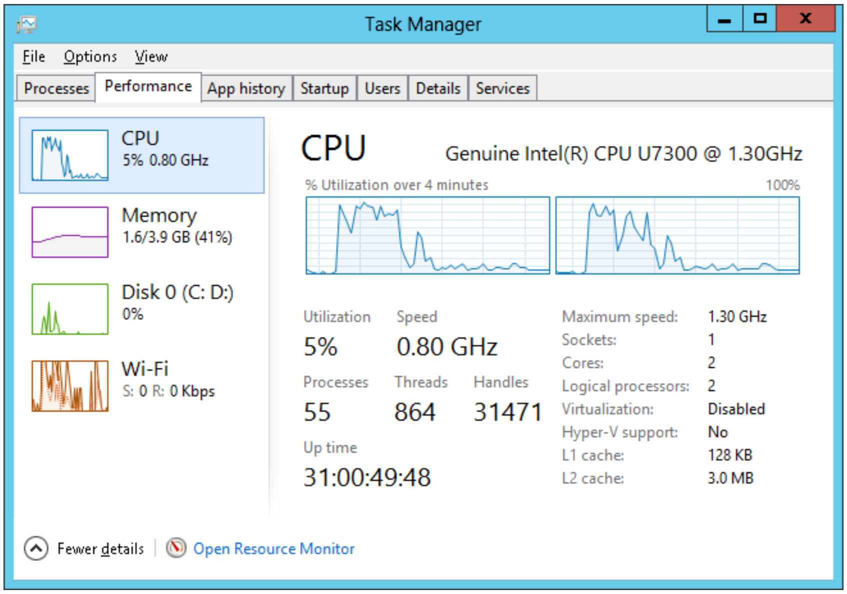
The remaining chapters in this book discuss the various Windows and CLR mechanisms that exist so that you can effectively wrestle with the tension of creating as few threads as possible, while still keeping your code responsive and allowing it to scale if your code is running on a machine with multiple cores.

Stop the Madness

If all we cared about was raw performance, then the optimum number of threads to have on any machine is identical to the number of CPUs on that machine. So a machine with one CPU would have only one thread, a machine with two CPUs would have two threads, and so on. The reason is obvious: if you have more threads than CPUs, then context switching is introduced and performance deteriorates. If each CPU has just one thread, then no context switching exists and the threads run at full speed.

However, Microsoft designed Windows to favor reliability and responsiveness as opposed to favoring raw speed and performance. And I commend this decision: I don’t think any of us would be using Windows or the .NET Framework today if applications could still stop the operating system and other applications. Therefore, Windows gives each process its own thread for improved system reliability and responsiveness. On my machine, for example, when I run Task Manager and select the Performance tab, I see the image shown in Figure 26-1.

It shows that my machine currently has 55 processes running on it, and so we’d expect that there were at least 55 threads on my machine, because each process gets at least 1 thread. But Task Manager also shows that my machine currently has 864 threads in it! All these threads end up allocating many megabytes of memory on my machine, which has only 4 GB of RAM in it. This also means that there is an average of approximately 15.7 threads per process, when I should ideally have only 2 threads per process on my dual core machine!



CPU Genuine Intel(R) CPU U7300 @ 1.30GHz

Utilization 5%
Speed 0.80 GHz
Processes 55
Threads 864
Handles 31471
Up time 31:00:49:48

Maximum speed:	1.30 GHz
Sockets:	1
Cores:	2
Logical processors:	2
Virtualization:	Disabled
Hyper-V support:	No
L1 cache:	128 KB
L2 cache:	3.0 MB

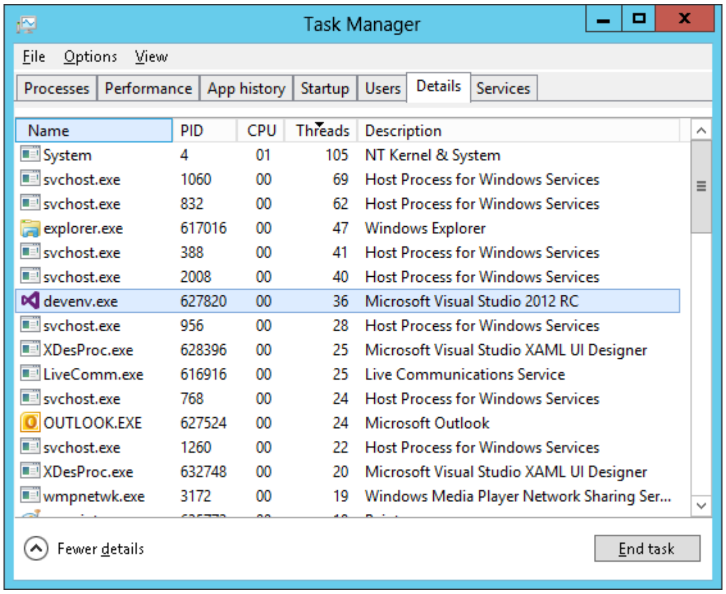
Figure 26-1 Task Manager showing system performance.

To make matters worse, when I look at the CPU Usage, it shows that my CPU is busy 5 percent of the time. This means that 95 percent of the time, these 864 threads have literally nothing to do—they are just soaking up memory that is definitely not being used when the threads are not running. You have to ask yourself: Do these applications need all these threads to do nothing 95 percent of the time? The answer to this question has to be “No.” Now, if you want to see which processes are the most wasteful, click the Task Manager’s Details tab, add the Threads column, and sort this column in descending order, as shown in Figure 26-2.[2]

As you can see here, System has created 105 threads and is using 1 percent of the CPU, Explorer has created 47 threads to use 0 percent of the CPU, Visual Studio (Devenv.exe) has created 36 threads to use 0 percent of the CPU, Microsoft Outlook has created 24 threads to use 0 percent of the CPU, and so on. What is going on here?

When developers were learning about Windows, they learned that a process in Windows is very, very expensive. Creating a process usually takes several seconds, a lot of memory must be allocated, this memory must be initialized, the EXE and DLL files have to load from disk, and so on. By comparison, creating a thread in Windows is very cheap, so developers decided to stop creating processes and start creating threads instead. So now we have lots of threads. But even though threads are cheaper than processes, they are still very expensive compared to most other system resources, so they should be used sparingly and appropriately.

[2] You add the column by right-clicking an existing column and selecting Select Columns.



Name	PID	CPU	Threads	Description
System	4	01	105	NT Kernel & System
svchost.exe	1060	00	69	Host Process for Windows Services
svchost.exe	832	00	62	Host Process for Windows Services
explorer.exe	617016	00	47	Windows Explorer
svchost.exe	388	00	41	Host Process for Windows Services
svchost.exe	2008	00	40	Host Process for Windows Services
devenv.exe	627820	00	36	Microsoft Visual Studio 2012 RC
svchost.exe	956	00	28	Host Process for Windows Services
XDesProc.exe	628396	00	25	Microsoft Visual Studio XAML UI Designer
LiveComm.exe	616916	00	25	Live Communications Service
svchost.exe	768	00	24	Host Process for Windows Services
OUTLOOK.EXE	627524	00	24	Microsoft Outlook
svchost.exe	1260	00	22	Host Process for Windows Services
XDesProc.exe	632748	00	20	Microsoft Visual Studio XAML UI Designer
wmpnetwk.exe	3172	00	19	Windows Media Player Network Sharing Ser...

Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

FIGURE 26-2 Task Manager showing details.

Well, without a doubt, we can say for sure that all of the applications we've just discussed are using threads inefficiently. There is just no way that all of these threads need to exist in the system. It is one thing to allocate resources inside an application; it's quite another to allocate them and then not use them. This is just wasteful, and allocating all the memory for thread stacks means that there is less memory for more important data, such as a user's document.[3]

To make matters worse, what if these were the processes running in a single user's Remote Desktop Services session—and what if there were actually 100 users on this machine? Then there would be 100 instances of Outlook, all creating 24 threads only to do nothing with them. That's 2,400 threads each with its own kernel object, TEB, user-mode stack, kernel-mode stack, etc. That is a lot of wasted resources. This madness has to stop, especially if Microsoft wants to give users a good experience when running Windows on netbook computers, many of which have only 1 GB of RAM. Again, the chapters in this part of the book will describe how to properly design an application to use very few threads in an efficient manner.

[3] I just can't resist sharing with you another demonstration of how bad this situation is. Try this: open Notepad.exe and use Task Manager to see how many threads are in it (you should see 1). Then select Notepad's File Open menu item to display the common File Open dialog box. After the dialog box opens, look at Task Manager to see how many new threads just got created. On my machine, 31 additional threads are created just by displaying this dialog box! In fact, every application that uses the common File Open or File Save dialog box will get many additional threads created inside it that sit idle most of the time. A lot of these threads aren't even destroyed when the dialog box is closed!

CPU Trends

In the past, CPU speeds used to increase with time, so an application that ran slowly on one machine would typically run faster on a newer machine. However, CPU manufacturers are unable to continue the trend of making CPUs faster. When you run CPUs at high speeds, they produce a lot of heat that has to be dissipated. A few years ago, I acquired a newly released notebook computer from a respected manufacturer. This computer had a bug in its firmware that made it not turn the fan on enough; as a result, after running the computer for a while, the CPU and the motherboard melted. The hardware manufacturer replaced the machine and then "improved" the firmware by making the fan run more frequently. Unfortunately, this had the effect of draining the battery faster, because fans consume a lot of power.

These are the kinds of problems that the hardware vendors face today. Because CPU manufacturers can't continuously produce higher-speed CPUs, they have instead turned their attention to making transistors smaller so that more of them can reside on a single chip. Today, we can have a single silicon chip that contains two or more CPU cores. The result is that our software only gets faster if we write our software to use the multiple cores. How do we do this? We use threads *in an intelligent fashion*.

Computers use three kinds of multi-CPU technologies today:

- **Multiple CPUs** Some computers just have multiple CPUs in them. That is, the motherboard has multiple sockets on it, with each socket containing a CPU. Because the motherboard must be bigger, the computer case is bigger as well, and sometimes these machines have multiple power supplies in them due to the additional power drain. These kinds of computers have been around for a few decades, but they are not as popular today due to their increased size and cost.

- **Hyperthreaded chips** This technology (owned by Intel) allows a single chip to look like two chips. The chip contains two sets of architectural states, such as CPU registers, but the chip has only one set of execution resources. To Windows, this looks like there are two CPUs in the machine, so Windows schedules two threads concurrently. However, the chip only executes one of the threads at a time. When one thread pauses due to a cache miss, branch misprediction, or data dependency, the chip switches to the other thread. This all happens in hardware, and Windows doesn't know that it is happening; Windows believes that both threads are running concurrently. Windows does know about hyperthreaded CPUs, and if you have multiple hyperthreaded CPUs in a single machine, Windows will first schedule one thread on each CPU so that the threads are truly running concurrently and then schedule other threads on the already-busy CPUs. Intel claims that a hyperthreaded CPU can improve performance by 10 percent to 30 percent.

- **Multi-core chips** A few years ago, single chips containing multiple CPU cores have entered the scene. As I write this, chips with two, three, and four cores are readily available. Even my notebook computer has two cores in it; our mobile phones now have multiple cores in them too. Intel has even been working on a single chip with 80 cores on it! Wow, this is a lot of computing power! And Intel even has hyperthreaded multi-core chips.

CLR Threads and Windows Threads

Today, the CLR uses the threading capabilities of Windows, so Part V of this book is really focusing on how the threading capabilities of Windows are exposed to developers who write code by using the CLR. I will explain about how threads in Windows work and how the CLR alters the behavior (if it does). However, if you'd like more information about threads, I recommend reading some of my earlier writings on the topic, such as *Windows via C/C++*, Fifth Edition, by myself and Christophe Nasarre (Microsoft Press, 2007).

Note Back in the early days of the .NET Framework, the CLR team felt that they would someday have the CLR offer logical threads, which did not necessarily map to Windows threads. However, around 2005, this was attempted unsuccessfully, causing the CLR team to give up on the idea. So today, a CLR thread is identical to a Windows thread. However, in the .NET Framework, you see remnants of the attempt. For example, the `System.Environment` class exposes a `CurrentManagedThreadId` property, which returns the CLR's ID for a thread, although the `System.Diagnostics.ProcessThread` class exposes an `Id` property that returns Windows's ID for the same thread. The `System.Thread` class's `BeginThreadAffintiy` and `EndThreadAffinity` methods were also introduced to deal with the notion that a CLR thread might not map to a Windows thread.

Note For Windows Store apps, Microsoft has removed some APIs related to threading, because the API encouraged bad programming practices (as discussed in the "Stop the Madness" section of this chapter), or because the APIs don't help achieve the goals that Microsoft has set out for Windows Store apps. For example, the whole `System.Thread` class is not available to Windows Store apps because it has many bad APIs (such as `Start`, `IsBackground`, `Sleep`, `Suspend`, `Resume`, `Join`, `Interrupt`, `Abort`, `BeginThreadAffinity`, and `EndThreadAffinity`). Personally, I think this is great, and I wish it had happened a long time ago. So, in Chapters 26 to 30, I do discuss some APIs and features that are available for desktop apps but are not available for Windows Store apps. As you read these chapters, you should be able to easily discern why certain APIs are unavailable for Windows Store apps.

Using a Dedicated Thread to Perform an Asynchronous Compute-Bound Operation

In this section, I will show you how to create a thread and have it perform an asynchronous compute-bound operation. Although I am going to walk you through this, I highly recommend that you avoid the technique I show you here. And, in fact, this technique is not even possible if you are building a Windows Store app because the `Thread` class is not available. Instead, you should use the thread pool to execute asynchronous compute-bound operations whenever possible. I go into the details about doing this in Chapter 27, "Compute-Bound Asynchronous Operations."

However, there are some very unusual occasions when you might want to explicitly create a thread dedicated to executing a particular compute-bound operation. Typically, you'd want to create a dedicated thread if you're going to execute code that requires the thread to be in a particular state that is not normal for a thread pool thread. For example, explicitly create your own thread if any of the following is true:

- You need the thread to run with a non-normal thread priority. All thread pool threads run at normal priority. Although you can change this, it is not recommended, and the priority change does not persist across thread pool operations.
- You need the thread to behave as a foreground thread, thereby preventing the application from dying until the thread has completed its task. For more information, see the "Foreground Threads vs. Background Threads" section later in this chapter. Thread pool threads are always background threads, and they may not complete their task if the CLR wants to terminate the process.
- The compute-bound task is extremely long-running; this way, I would not be taxing the thread pool's logic as it tries to figure out whether to create an additional thread.
- You want to start a thread and possibly abort it prematurely by calling Thread's Abort method (discussed in Chapter 22, "CLR Hosting and AppDomains").

To create a dedicated thread, you construct an instance of the System.Threading.Thread class, passing the name of a method into its constructor. Here is the prototype of Thread's constructor.

```
public sealed class Thread : CriticalFinalizerObject, ... {
   public Thread(ParameterizedThreadStart start);
   // Less commonly used constructors are not shown here
}
```

The start parameter identifies the method that the dedicated thread will execute, and this method must match the signature of the ParameterizedThreadStart delegate.[4]

```
delegate void ParameterizedThreadStart(Object obj);
```

Constructing a Thread object is a relatively lightweight operation because it does not actually create a physical operating system thread. To actually create the operating system thread and have it start executing the callback method, you must call Thread's Start method, passing into it the object (state) that you want passed as the callback method's argument.

[4] For the record, Thread also offers a constructor that takes a ThreadStart delegate that accepts no arguments and returns void. Personally, I recommend that you avoid this constructor and delegate because they are more limiting. If your thread method takes an Object and returns void, then you can invoke your method by using a dedicated thread or invoke it by using the thread pool (as shown in Chapter 27).

The following code demonstrates how to create a dedicated thread and have it call a method asynchronously.

```
using System;
using System.Threading;

public static class Program {
   public static void Main() {
      Console.WriteLine("Main thread: starting a dedicated thread " +
         "to do an asynchronous operation");
      Thread dedicatedThread = new Thread(ComputeBoundOp);
      dedicatedThread.Start(5);

      Console.WriteLine("Main thread: Doing other work here...");
      Thread.Sleep(10000);     // Simulating other work (10 seconds)

      dedicatedThread.Join();  // Wait for thread to terminate
      Console.WriteLine("Hit <Enter> to end this program...");
      Console.ReadLine();
   }

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

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

      // When this method returns, the dedicated thread dies
   }
}
```

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

```
Main thread: starting a dedicated thread to do an asynchronous operation
Main thread: Doing other work here...
In ComputeBoundOp: state=5
```

Sometimes when I run this code, I get the following output, because I can't control how Windows schedules the two threads.

```
Main thread: starting a dedicated thread to do an asynchronous operation
In ComputeBoundOp: state=5
Main thread: Doing other work here...
```

Notice that the `Main` method calls `Join`. The `Join` method causes the calling thread to stop executing any code until the thread identified by `dedicatedThread` has destroyed itself or been terminated.

Reasons to Use Threads

There are really two reasons to use threads:

- **Responsiveness (typically for client-side GUI applications)** Windows gives each process its own thread so that one application entering an infinite loop doesn't prevent the user from working with other applications. Similarly, within your client-side GUI application, you could spawn some work off onto a thread so that your GUI thread remains responsive to user input events. In this example, you are possibly creating more threads than available cores on the machine, so you are wasting system resources and hurting performance. However, the user is gaining a responsive user interface and therefore having a better overall experience with your application.
- **Performance (for client and server side applications)** Because Windows can schedule one thread per CPU and because the CPUs can execute these threads concurrently, your application can improve its performance by having multiple operations executing at the same time in parallel. Of course, you only get the improved performance if and only if your application is running on a machine with multiple CPUs in it. Today, machines with multiple CPUs in them are quite common, so designing your application to use multiple cores makes sense and is the focus of Chapter 27 and Chapter 28, "I/O-Bound Asynchronous Operations."

Now, I'd like to share with you a theory of mine. Every computer has an incredibly powerful resource inside it: the CPU itself. If someone spends money on a computer, then that computer should be working all the time. In other words, I believe that all the CPUs in a computer should be running at 100 percent utilization all the time. I will qualify this statement with two caveats. First, you may not want the CPUs running at 100 percent utilization if the computer is on battery power, because that may drain the battery too quickly. Second, some data centers would prefer to have 10 machines running at 50 percent CPU utilization rather than 5 machines running at 100 percent CPU utilization, because running CPUs at full power tends to generate heat, which requires cooling systems, and powering an HVAC cooling system can be more expensive than powering more computers running at reduced capacity. Although data centers find it increasingly expensive to maintain multiple machines, because each machine has to have periodic hardware and software upgrades and monitoring, this has to be weighed against the expense of running a cooling system.

Now, if you agree with my theory, then the next step is to figure out what the CPUs should be doing. Before I give you my ideas here, let me say something else first. In the past, developers and end users always felt that the computer was not powerful enough. Therefore, we developers would never just execute code unless the end users give us permission to do so and indicate that it is OK for the application to consume CPU resources via UI elements, such as menu items, buttons, and check boxes.

But now, times have changed. Computers ship with phenomenal amounts of computing power. Earlier in this chapter, I showed you how Task Manager was reporting that my CPU was busy just 5 percent of the time. If my computer contained a quad-core CPU in it instead of the dual-core CPU that it now has, then Task Manager will report 2 percent more often. When an 80-core processor comes out, the machine will look like it's doing nothing almost all the time. To computer purchasers, it looks like they're spending more money for more CPUs and the computer is doing less work!

This is the reason why the hardware manufacturers are having a hard time selling multi-core computers to users: the software isn't taking advantage of the hardware and users get no benefit from buying machines with additional CPUs. What I'm saying is that we now have an abundance of computing power available and more is on the way, so developers can aggressively consume it. That's right—in the past, we would never dream of having our applications perform some computation unless we knew the end user wanted the result of that computation. But now that we have extra computing power, we can dream like this.

Here's an example: when you stop typing in Visual Studio's editor, Visual Studio automatically spawns the compiler and compiles your code. This makes developers incredibly productive because they can see warnings and errors in their source code as they type and can fix things immediately. In fact, what developers think of today as the Edit-Build-Debug cycle will become just the Edit-Debug cycle, because building (compiling) code will just happen all the time. You, as an end user, won't notice this because there is a lot of CPU power available and other things you're doing will barely be affected by the frequent running of the compiler. In fact, I would expect that in some future version of Visual Studio, the Build menu item will disappear completely, because building will just become automatic. Not only does the application's UI get simpler, but the application also offers "answers" to the end user, making them more productive.

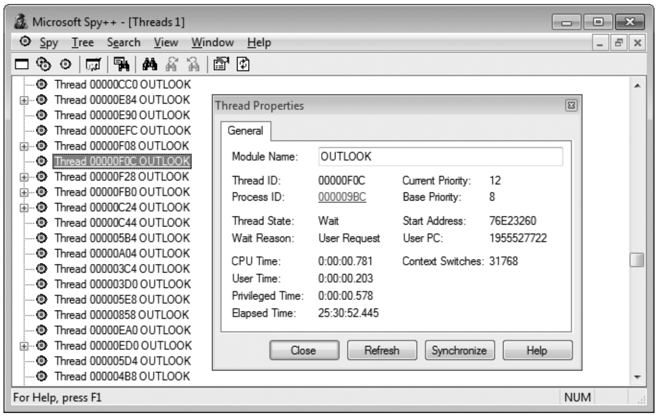
When we remove UI components like menu items, computers get simpler for end users. There are fewer options for them and fewer concepts for them to read and understand. It is the multi-core revolution that allows us to remove these UI elements, thereby making software so much simpler for end users that my grandmother might someday feel comfortable using a computer. For developers, removing UI elements usually results in less testing, and offering fewer options to the end user simplifies the code base. And if you currently localize the text in your UI elements and your documentation (like Microsoft does), then removing the UI elements means that you write less documentation and you don't have to localize this documentation anymore. All of this can save your organization a lot of time and money.

Here are some more examples of aggressive CPU consumption: spell checking and grammar checking of documents, recalculation of spreadsheets, indexing files on your disk for fast searching, and defragmenting your hard disk to improve I/O performance.

I want to live in a world where the UI is reduced and simplified, I have more screen real estate to visualize the data that I'm actually working on, and applications offer me information that helps me get my work done quickly and efficiently instead of me telling the application to go get information for me. I think the hardware has been there for software developers to use for the past few years. It's time for the software to start using the hardware creatively.

Thread Scheduling and Priorities

A preemptive operating system must use some kind of algorithm to determine which threads should be scheduled when and for how long. In this section, we'll look at the algorithm Windows uses. Earlier in this chapter, I mentioned how every thread's kernel object contains a context structure. The context structure reflects the state of the thread's CPU registers when the thread last executed. After a time-slice, Windows looks at all the thread kernel objects currently in existence. Of these objects, only the threads that are not waiting for something are considered schedulable. Windows selects one of the schedulable thread kernel objects and context switches to it. Windows actually keeps a record of how many times each thread gets context switched to. You can see this when using a tool such as Microsoft Spy++. Figure 26-3 shows the properties for a thread. Notice that this thread has been scheduled 31,768 times.[5]



```
Thread 00000CC0 OUTLOOK
Thread 00000E84 OUTLOOK
Thread 00000E90 OUTLOOK
Thread 00000EFC OUTLOOK
Thread 00000F08 OUTLOOK
Thread 00000F0C OUTLOOK
Thread 00000F28 OUTLOOK
Thread 00000FB0 OUTLOOK
Thread 00000C24 OUTLOOK
Thread 00000C44 OUTLOOK
Thread 000005B4 OUTLOOK
Thread 00000A04 OUTLOOK
Thread 000003C4 OUTLOOK
Thread 000003D0 OUTLOOK
Thread 000005E8 OUTLOOK
Thread 00000858 OUTLOOK
Thread 00000EA0 OUTLOOK
Thread 00000ED0 OUTLOOK
Thread 000005D4 OUTLOOK
Thread 000004B8 OUTLOOK
```

```
Module Name:     OUTLOOK

Thread ID:       00000F0C        Current Priority:   12
Process ID:      000009BC        Base Priority:      8

Thread State:    Wait            Start Address:      76E23260
Wait Reason:     User Request    User PC:            1955527722

CPU Time:        0:00:00.781     Context Switches:   31768
User Time:       0:00:00.203
Privileged Time: 0:00:00.578
Elapsed Time:    25:30:52.445
```

FIGURE 26-3 Spy++ showing a thread's properties.

At this point, the thread is executing code and manipulating data in its process's address space. After another time-slice, Windows performs another context switch. Windows performs context switches from the moment the system is booted and continues until the system is shut down.

[5] As a side note, you can also see that the thread has been in the system for more than 25 hours, but it actually used less than one second of CPU time, which wastes a lot of resources.

Windows is called a preemptive multithreaded operating system because a thread can be stopped at any time and another thread can be scheduled. As you'll see, you have some control over this, but not much. Just remember that you cannot guarantee that your thread will always be running and that no other thread will be allowed to run.

Note Developers frequently ask me how they can guarantee that their thread will start running within some time period after some event—for example, how can you ensure that a particular thread will start running within 1 ms of data coming from the network? I have an easy answer: You can't.

Real-time operating systems can make these promises, but Windows is not a real-time operating system. A real-time operating system requires intimate knowledge of the hardware it is running on so that it knows the latency associated with its hard disk controllers, keyboards, and other components. Microsoft's goal with Windows is to make it work on a wide variety of hardware: different CPUs, different drives, different networks, and so on. In short, Windows is not designed to be a real-time operating system. Let me also add that the CLR makes managed code behave even less in real time. There are many reasons for this, including just-in-time (JIT) loading of DLLs, JIT compiling of code, and the garbage collector kicking in at unpredictable times.

Every thread is assigned a priority level ranging from 0 (the lowest) to 31 (the highest). When the system decides which thread to assign to a CPU, it examines the priority 31 threads first and schedules them in a round-robin fashion. If a priority 31 thread is schedulable, it is assigned to a CPU. At the end of this thread's time-slice, the system checks to see whether there is another priority 31 thread that can run; if so, it allows that thread to be assigned to a CPU.

As long as priority 31 threads are schedulable, the system never assigns any thread with a priority of 0 through 30 to a CPU. This condition is called *starvation*, and it occurs when higher-priority threads use so much CPU time that they prevent lower-priority threads from executing. Starvation is much less likely to occur on a multiprocessor machine because a priority 31 thread and a priority 30 thread can run simultaneously on such a machine. The system always tries to keep the CPUs busy, and CPUs sit idle only if no threads are schedulable.

Higher-priority threads always preempt lower-priority threads, regardless of what the lower-priority threads are executing. For example, if a priority 5 thread is running and the system determines that a higher-priority thread is ready to run, the system immediately suspends the lower-priority thread (even if it's in the middle of its time-slice) and assigns the CPU to the higher-priority thread, which gets a full time-slice.

By the way, when the system boots, it creates a special thread called the *zero page thread*. This thread is assigned priority 0 and is the only thread in the entire system that runs at priority 0. The zero page thread is responsible for zeroing any free pages of RAM in the system when no other threads need to perform work.

Microsoft realized that assigning priority levels to threads was going to be too hard for developers to rationalize. Should this thread be priority level 10? Should this other thread be priority level 23? To resolve this issue, Windows exposes an abstract layer over the priority level system.

When designing your application, you should decide whether your application needs to be more or less responsive than other applications that may be running on the machine. Then you choose a process priority class to reflect your decision. Windows supports six process priority classes: Idle, Below Normal, Normal, Above Normal, High, and Realtime. Of course, Normal is the default and is therefore the most common priority class by far.

The Idle priority class is perfect for applications (like screen savers) that run when the system is all but doing nothing. A computer that is not being used interactively might still be busy (acting as a file server, for example) and should not have to compete for CPU time with a screen saver. Statistics-tracking applications that periodically update some state about the system usually should not interfere with more critical tasks.

You should use the High priority class only when absolutely necessary. You should avoid using the Realtime priority class if possible. Realtime priority is extremely high and can interfere with operating system tasks, such as preventing required disk I/O and network traffic from occurring. In addition, a Realtime process's threads could prevent keyboard and mouse input from being processed in a timely manner, causing the user to think that the system is completely frozen. Basically, you should have a good reason for using Realtime priority, such as the need to respond to hardware events with short latency or to perform some short-lived task.

Note To keep the overall system running smoothly, a process cannot run in the Realtime priority class unless the user has the Increase Scheduling Priority privilege. Any user designated as an administrator or a power user has this privilege by default.

After you select a priority class, you should stop thinking about how your application relates to other applications and just concentrate on the threads within your application. Windows supports seven relative thread priorities: Idle, Lowest, Below Normal, Normal, Above Normal, Highest, and Time-Critical. These priorities are relative to the process's priority class. Again, Normal relative thread priority is the default, and it is therefore the most common.

So, to summarize, your process is a member of a priority class and within that process you assign thread priorities that are relative to each other. You'll notice that I haven't said anything about priority levels 0 through 31. Application developers never work with priority levels directly. Instead, the system maps the process's priority class and a thread's relative priority to a priority level. Table 26-1 shows how the process's priority class and the thread's relative priority maps to priority levels.

TABLE 26-1 How Process Priority Class and Relative Thread Priorities Map to Priority Levels

Relative Thread Priority	**Process Priority Class**					
	Idle	Below Normal	Normal	Above Normal	High	Realtime
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above Normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below Normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

For example, a Normal thread in a Normal process is assigned a priority level of 8. Because most processes are of the Normal priority class and most threads are of Normal thread priority, most threads in the system have a priority level of 8.

If you have a Normal thread in a high-priority process, the thread will have a priority level of 13. If you change the process's priority class to Idle, the thread's priority level becomes 4. Remember that thread priorities are relative to the process's priority class. If you change a process's priority class, the thread's relative priority will not change, but its priority number will.

Notice that the table does not show any way for a thread to have a priority level of 0. This is because the 0 priority is reserved for the zero page thread and the system does not allow any other thread to have a priority of 0. Also, the following priority levels are not obtainable: 17, 18, 19, 20, 21, 27, 28, 29, or 30. If you are writing a device driver that runs in kernel mode, you can obtain these levels; a user-mode application cannot. Also note that a thread in the Realtime priority class can't be below priority level 16. Likewise, a thread in a priority class other than Realtime cannot be above 15.

Note The concept of a process priority class confuses some people. They think that this somehow means that Windows schedules processes. However, Windows never schedules processes; Windows only schedules threads. The process priority class is an abstract concept that Microsoft created to help you rationalize how your application compares with other running applications; it serves no other purpose.

Important It is best to lower a thread's priority instead of raising another thread's priority. You would normally lower a thread's priority if that thread was going to execute a long-running compute-bound task like compiling code, spell checking, spreadsheet recalculations, etc. You would raise a thread's priority if the thread needs to respond to something very quickly and then run for a very short period of time and go back to its wait state. High-priority threads should be waiting for something most of their life so that they do not affect the responsiveness of the whole system. The Explorer thread that responds to the user pressing the Windows key on the keyboard is an example of a high-priority thread. When the user presses this key, Explorer preempts other lower-priority threads immediately and displays its menu. As the user navigates the menu, Explorer's thread responds to each keystroke quickly, updates the menu, and then stops running until the user continues navigating the menu.

Normally, a process is assigned a priority class based on the process that starts it running. And most processes are started by Explorer, which spawns all its child processes in the Normal priority class. Managed applications are not supposed to act as though they own their own processes; they are supposed to act as though they run in an AppDomain, so managed applications are not supposed to change their process's priority class because this would affect all code running in the process. For example, many ASP.NET applications run in a single process, with each application in its own AppDomain. The same is true for Microsoft Silverlight applications, which run in an Internet browser process, and managed stored procedures, which run inside the Microsoft SQL Server process.

In addition, a Windows Store app is not able to create additional AppDomains, cannot change its process's priority class, or any of its threads' priorities. Furthermore, when a Windows Store app is not in the foreground, Windows automatically suspends all its threads. This serves two purposes. First, it prevents a background app from "stealing" CPU time away from the app the user is actively interacting with. This ensures that touch events like swipes are fast and fluid. Second, by reducing CPU usage, battery power is conserved, allowing the PC to run longer on a single charge.

On the other hand, your application can change the relative thread priority of its threads by setting `Thread`'s `Priority` property, passing it one of the five values (`Lowest`, `BelowNormal`, `Normal`, `AboveNormal`, or `Highest`) defined in the `ThreadPriority` enumerated type. However, just as Windows has reserved the priority level 0 and the real-time range for itself, the CLR reserves the Idle and Time-Critical priority levels for itself. Today, the CLR has no threads that run at Idle priority level, but this could change in the future. However, the CLR's finalizer thread, discussed in Chapter 21, "The Managed Heap and Garbage Collection," runs at the Time-Critical priority level. Therefore, as a managed developer, you really only get to use the five highlighted relative thread priorities listed in Table 26-1.

Important Today, most applications do not take advantage of thread priorities. However, in the world that I envision, where the CPUs are busy 100 percent of the time doing some kind of useful work, using thread priorities becomes critically important so that system responsiveness is unaffected. Unfortunately, end users have been trained to interpret a high-CPU usage number to mean that an application is out of control. In my new world, end users will need to be retrained to understand that high-CPU usage is a good thing—that it actually means that the computer is aggressively processing helpful pieces of information for users. The real problem would be if all the CPUs are busy running threads that are priority level 8 and above, because this would mean that applications are having trouble responding to end user input. Perhaps a future version of Task Manager will take thread priority levels into account when reporting CPU usage; this would be much more helpful in diagnosing a troubled system.

For desktop apps (non–Windows Store apps), I should point out that the `System.Diagnostics` namespace contains a `Process` class and a `ProcessThread` class. These classes provide the Windows view of a process and thread, respectively. These classes are provided for developers wanting to write utility applications in managed code or for developers who are trying to instrument their code to help them debug it. In fact, this is why the classes are in the `System.Diagnostics` namespace. Applications need to be running with special security permissions to use these two classes. You would not be able to use these classes from a Silverlight application or an ASP.NET application, for example.

On the other hand, applications can use the `AppDomain` and `Thread` classes, which expose the CLR's view of an AppDomain and thread. For the most part, special security permissions are not required to use these classes, although some operations are still considered privileged.

Foreground Threads vs. Background Threads

The CLR considers every thread to be either a foreground thread or a background thread. When all the foreground threads in a process stop running, the CLR forcibly ends any background threads that are still running. These background threads are ended immediately; no exception is thrown.

Therefore, you should use foreground threads to execute tasks that you really want to complete, like flushing data from a memory buffer out to disk. And you should use background threads for tasks that are not mission-critical, like recalculating spreadsheet cells or indexing records, because this work can continue again when the application restarts, and there is no need to force the application to stay active if the user wants to terminate it.

The CLR needed to provide this concept of foreground and background threads to better support AppDomains. You see, each AppDomain could be running a separate application and each of these applications would have its own foreground thread. If one application exits, causing its foreground thread to terminate, then the CLR still needs to stay up and running so that other applications continue to run. After all the applications exit and all their foreground threads terminate, the whole process can be destroyed.

The following code demonstrates the difference between foreground and background threads.

```
using System;
using System.Threading;

public static class Program {
   public static void Main() {
      // Create a new thread (defaults to foreground)
      Thread t = new Thread(Worker);

      // Make the thread a background thread
      t.IsBackground = true;

      t.Start(); // Start the thread
      // If t is a foreground thread, the application won't die for about 10 seconds
      // If t is a background thread, the application dies immediately
      Console.WriteLine("Returning from Main");
   }

   private static void Worker() {
      Thread.Sleep(10000);  // Simulate doing 10 seconds of work

      // The following line only gets displayed if this code is executed by a foreground thread
      Console.WriteLine("Returning from Worker");
   }
}
```

It is possible to change a thread from foreground to background and vice versa at any time during its lifetime. An application's primary thread and any threads explicitly created by constructing a Thread object default to being foreground threads. On the other hand, thread pool threads default to being background threads. Also, any threads created by native code that enter the managed execution environment are marked as background threads.

Important Try to avoid using foreground threads as much as possible. I was brought into a consulting job once where an application just wouldn't terminate. After I spent several hours researching the problem, it turned out that a UI component was explicitly creating a foreground thread (the default), and that was why the process wouldn't terminate. We changed the component to use the thread pool to fix the problem, and efficiency improved as well.

What Now?

In this chapter, I've explained the basics about threads, and I hope I've made it clear to you that threads are very expensive resources that should be used sparingly. The best way to accomplish this is by using the thread pool. The thread pool will manage thread creation and destruction for you automatically. The thread pool creates a set of threads that get reused for various tasks so your application requires just a few threads to accomplish all of its work.

In Chapter 27, I will focus on how to use the thread pool to perform compute-bound operations. Then, in Chapter 28, I will discuss how to use the thread pool to perform I/O-bound operations. In many scenarios, you can perform asynchronous compute-bound and I/O-bound operations in such a way that thread synchronization is not required at all. However, there are some scenarios where thread synchronization is required, and the way that the thread synchronization constructs work and the difference between these various constructs are discussed in Chapter 29, "Primitive Thread Synchronization Constructs," and Chapter 30, "Hybrid Thread Synchronization Constructs."

Before ending this discussion, I'd like to point out that I have been working extensively with threads because the first beta version of Windows NT 3.1 was available around 1992. And when .NET was in beta, I started producing a library of classes that can simplify asynchronous programming and thread synchronization. This library is called the *Wintellect Power Threading Library*, and it is freely downloadable and usable. Versions of the library exist for the desktop CLR, the Silverlight CLR, and the Compact Framework. The library, documentation, and sample code can be downloaded from *http://Wintellect.com/PowerThreading.aspx*. This website also contains links to a support forum, in addition to videos that show how to use various parts of the library.