THE BOOK OF™ VISUAL BASIC 2005

.NET Insight for Classic VB Developers

by Matthew MacDonald

NO STARCH PRESS

San Francisco

11

THREADING

Threading is, from your application's point of view, a way of running various different pieces of code at the same time. Threading is also one of the more complex subjects examined in this book. That's not because it's difficult to use threading in your programs—as you'll see, Visual Basic 2005 makes it absurdly easy—but because it's difficult to use threading *correctly*. If you stick to the rules, keep your use of threads simple, or rely on the new all-in-one `BackgroundWorker` component, you'll be fine. If, however, you embark on a wild flight of multithreaded programming, you will probably commit one of the cardinal sins of threading, and wind up in a great deal of trouble. Many excellent developers have argued that the programming community has repeatedly become overexcited about threading in the past, and has misused it, creating endless headaches.

This chapter explains how to use threading and, more importantly, the guidelines you should follow to make sure you keep your programs free of such troubles as thread overload and synchronization glitches. Threading is a sophisticated subject with many nuances, so it's best to proceed carefully. However, a judicious use of carefully selected threads can make your applications appear faster, more responsive, and more sophisticated.

New in .NET

In Visual Basic 6, there was no easy way to create threads. Programmers who wanted to create truly multithreaded applications had to use the Windows API (or create and register separate COM components). Visual Basic 2005 provides these enhancements:

Integrated threads
The method of creating threads in Visual Basic 2005 is conceptually and syntactically similar to using the Windows API, but it's far less error-prone, and it's elegantly integrated into the language through the `System.Threading` namespace. The class library also contains a variety of tools to help implement synchronization and thread management.

Multithreaded debugging
The Visual Studio debugger now allows you to run and debug multi-threaded applications without forcing them to act as though they are single-threaded. You can even view a Threads window that shows all the currently active threads and allows you to pause and resume them individually.

The `BackgroundWorker`
As you'll learn in this chapter, multithreaded programming can be complicated. In .NET 2.0, Microsoft has added a `BackgroundWorker` component that can simplify the way you code a background task. All you need to do is handle the `BackgroundWorker` events and add your code—the `BackgroundWorker` takes care of the rest, making sure that your code executes on the correct thread. This chapter provides a detailed look at the `BackgroundWorker`.

An Introduction to Threading

Even if you've never tried to implement threading in your own code, you've already seen threads work in the modern Windows operating system. For example, you have probably noticed how you can work with a Windows application while another application is busy or in the process of starting up, because both applications run in separate processes and use separate *threads*. You have probably also seen that even when the system appears to be frozen, you can almost always bring up the Task Manager by pressing CTRL+ALT+DELETE. This is because the Task Manager runs on a thread that has an extremely high priority. Even if other applications are currently executing or frozen, trapping their threads in endless CPU-wasting cycles, Windows can usually wrest control away from them for a more important thread.

If you've used Windows 3.1, you'll remember that this has not always been the case. Threads really came into being with 32-bit Windows and the Windows 95 operating system.

Threads "Under the Hood"

Now that you have a little history, it's time to examine how threads really work.

Threads are created by the handful in Windows applications. If you open a number of different applications on your computer, you will quickly have several different processes and potentially dozens of different threads executing simultaneously. The Windows Task Manager can list all the active processes, which gives you an idea of the scope of the situation (Figure 11-1).

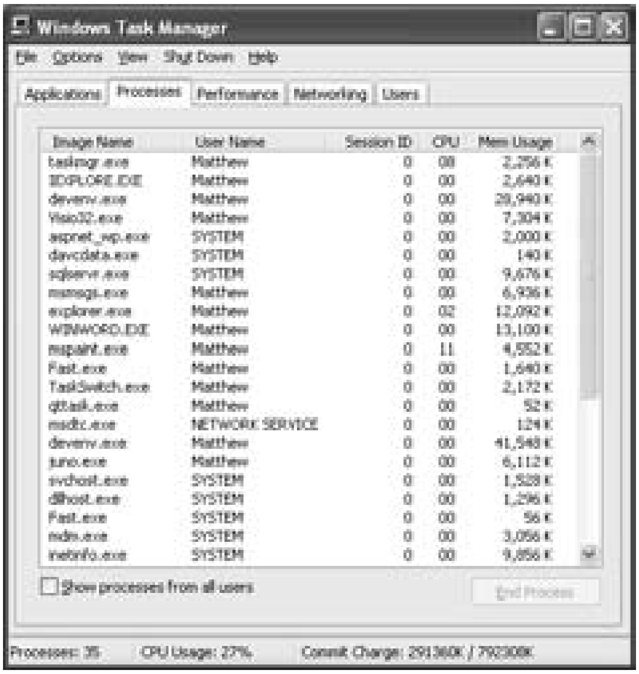

Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users
Show processes from all users
End Process

Image Name	User Name	Session ID	CPU	Mem Usage
taskmgr.exe	Matthew	0	00	2,256 K
IEXPLORE.EXE	Matthew	0	00	2,640 K
devenv.exe	Matthew	0	00	28,940 K
Visio32.exe	Matthew	0	00	7,304 K
aspnet_wp.exe	SYSTEM	0	00	2,000 K
davcdata.exe	SYSTEM	0	00	140 K
sqlservr.exe	SYSTEM	0	00	9,676 K
msmsgs.exe	Matthew	0	00	6,936 K
explorer.exe	Matthew	0	02	12,092 K
WINWORD.EXE	Matthew	0	00	13,100 K
mspaint.exe	Matthew	0	11	4,552 K
Fast.exe	Matthew	0	00	1,640 K
TaskSwitch.exe	Matthew	0	00	2,172 K
qttask.exe	Matthew	0	00	52 K
msdtc.exe	NETWORK SERVICE	0	00	124 K
devenv.exe	Matthew	0	00	41,548 K
juno.exe	Matthew	0	00	6,112 K
svchost.exe	SYSTEM	0	00	1,528 K
dllhost.exe	SYSTEM	0	00	1,296 K
Fast.exe	SYSTEM	0	00	56 K
mdm.exe	SYSTEM	0	00	3,056 K
inetinfo.exe	SYSTEM	0	00	9,856 K

Processes: 35 CPU Usage: 27% Commit Charge: 291360K / 792308K

Figure 11-1: Active processes in Task Manager

In all honesty, there is no way any computer, no matter how technologically advanced, can run dozens of different operations literally at once. If your system has two CPUs, it is technically possible for two instructions to be processed at the same time, and Windows is likely to send the instructions for different threads to different CPUs. At some point, however, you will still end up with many more threads than CPUs.

Windows handles this situation by switching rapidly between different threads. Each thread *thinks* it is running independently, but in reality it only runs for a little while, is suspended, and is then resumed a short while later for another brief interval of time. This switching is all taken care of by the Windows operating system and is called *preemptive multitasking*.

Comparing Single Threading and Multithreading

One consequence of thread switching is that multithreading usually doesn't result in a speed increase. Figure 11-2 shows why.

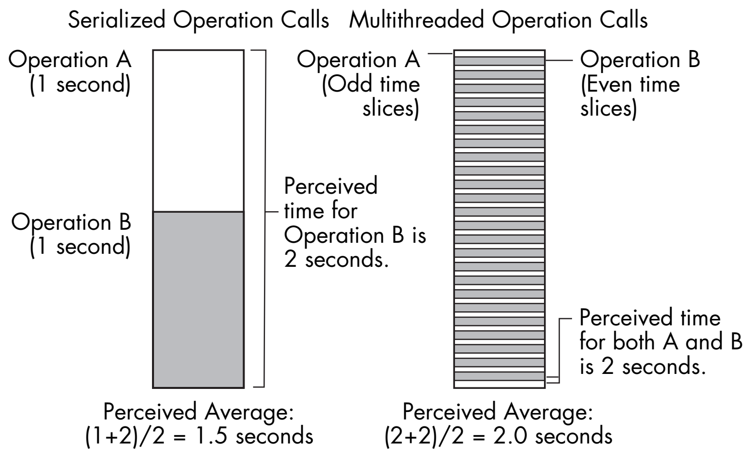

Serialized Operation Calls
Multithreaded Operation Calls
Operation A (1 second)
Operation B (1 second)
Perceived time for Operation B is 2 seconds.
Operation A (Odd time slices)
Operation B (Even time slices)
Perceived time for both A and B is 2 seconds.
Perceived Average: (1+2)/2 = 1.5 seconds
Perceived Average: (2+2)/2 = 2.0 seconds

Figure 11-2: Multithreading can make operations appear slower

This illustration compares a *single-threaded* and a *multithreaded* application. Both are performing the same two tasks, but the multithreaded program is working by dividing the two operations into numerous little intervals, and rapidly switching from one to the other. This switching introduces a small overhead, but overall, both applications will finish at about the same time. However, if a user is waiting for both tasks to end, they will both seem to be running more slowly, because both tasks will finish at more or less the same time—at the end of two seconds. With the single-threaded approach, Operation A will be completed sooner, after about a second of processing time.

So why use multithreading? Well, if you were running a short task and a long task simultaneously, the picture might change. For example, if Operation B took only a few time slices to complete, a user would perceive the multithreaded application as being much faster, because the user wouldn't have to wait for Operation A to finish before Operation B was started (in technical terms, with multithreading Operation B is not *blocked* by Operation A). In this case, Operation A would finish in a fraction of a second, rather than waiting the full one-second period (see Figure 11-3).

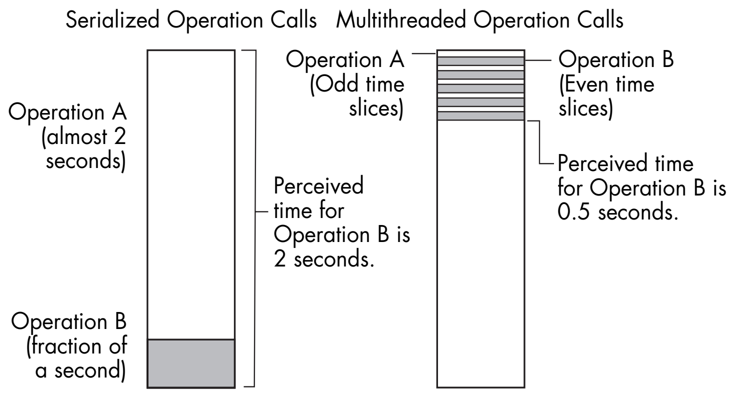

Serialized Operation Calls
Multithreaded Operation Calls
Operation A (almost 2 seconds)
Operation B (fraction of a second)
Perceived time for Operation B is 2 seconds.
Operation A (Odd time slices)
Operation B (Even time slices)
Perceived time for Operation B is 0.5 seconds.

Figure 11-3: Multithreading lets short tasks finish first

This is the basic principle of multithreading. Rather than speeding up tasks, it allows the quickest tasks to finish first; this makes an application appear more responsive and adds only a slight performance degradation (caused by all the required thread switching).

Multithreading works even better in applications where substantial waits are involved for certain tasks. For example, an application that spends a lot of time waiting for file I/O operations to complete could accomplish other useful tasks while waiting. In this case, multithreading can actually speed up the application, because it will not be forced to sit idle.

Scalability and Simplicity

There is one other reason to use threading: It makes program design much simpler for some common types of applications. For example, imagine you want to create an FTP server that can serve several simultaneous users. In a single-threaded application, you may find it very difficult to manage a variable number of users without hard-coding some preset limit on the number of users and implementing your own crude thread-switching logic.

With a multithreaded application, you can easily create a new thread to serve each client connection. Windows will take care of automatically assigning the processor time for each thread, and you can use exactly the same code to serve a hundred users as you would to serve one. Each thread uses the same code, but handles a different client. As the workload increases, all you need to do is add more threads.

Timers Versus Threads

You may have used `Timer` objects in previous versions of Visual Basic. `Timer` objects are still provided in Visual Basic 2005, and they are useful for a wide variety of tasks. Timers work differently than threads, however. From the program's standpoint, multiple threads execute simultaneously. In contrast, a timer works by interrupting your code in order to perform a single task at a "timed" interval. This task is then started, performed, and completed before control returns to the procedure in your application that was executing when the timer code launched.

This means that timers are not well suited for implementing long-running processes that perform a variety of independent, unpredictably scheduled tasks. To use a timer for this purpose, you would need to fake a multithreaded process by performing part of a task the first time a timer event occurs, a different part the next time, and so on.

To observe this problem, you can create a project with two timers and two labels, and add the following code.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
    Timer1.Enabled = True
    Timer2.Enabled = True
End Sub
```

```
Private Sub Timer1_Elapsed(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Timer1.Tick
    Dim i As Integer
    For i = 1 To 5000
        Label1.Text = i.ToString()
        Label1.Refresh()
    Next
    Timer1.Enabled = False
End Sub

Private Sub Timer2_Elapsed(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Timer2.Tick
    Dim i As Integer
    For i = 1 To 5000
        Label2.Text = i.ToString()
        Label2.Refresh()
    Next
    Timer2.Enabled = False
End Sub
```

When you run this program, one timer will take control, and one label will display the numbers from 1 to 5,000. The other timer will also perform the same process, but only after the first timer finishes. Even though both timers are scheduled to start at the same time, only one can work with the application window at a time. (Indeed, if Visual Basic 2005 were to allow timer events to execute simultaneously, it would lead programmers to encounter all the same synchronization issues that can occur with threads, as you'll see later this chapter.)

You'll also notice that while the timer is executing in this example (incrementing a label), the application as a whole won't be responsive. If you try to have perform another task with your application or drag its window around on the desktop, you'll find it performs very sluggishly.

Basic Threading with the BackgroundWorker

The simplest way to create a multithreaded application is to use the BackgroundWorker component, which is new in Visual Basic 2005. The BackgroundWorker handles all the multithreading behind the scenes and interacts with your code through events. Your code handles these events to perform the background task, track the progress of the background task, and deal with the final result. Because these events are automatically fired on the correct threads, you don't need to worry about thread synchronization and other headaches of low-level multithreaded programming.

Of course, the BackgroundWorker also has a limitation—namely, flexibility. Although the BackgroundWorker works well when you have a single, distinct task that needs to take place in the background, it isn't as well suited when you want to manage multiple background tasks, control thread priority, or maintain a thread for the lifetime of your application.

To use the `BackgroundWorker`, you begin by dragging it from the Components section of the Toolbox onto a form. (You can also create a `BackgroundWorker` in code, but the drag-and-drop approach is easiest.) The `BackgroundWorker` will then appear in the component tray (see Figure 11-4).

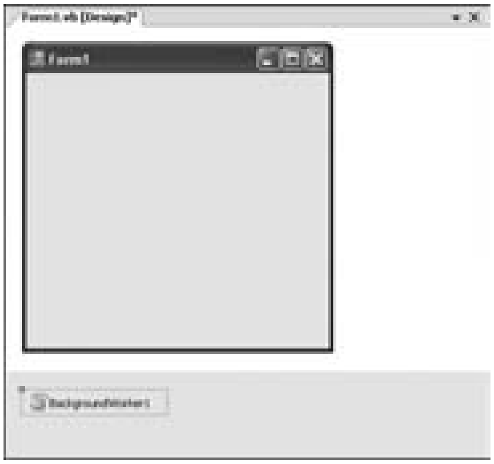


Figure 11-4: Adding the `BackgroundWorker` to a form

Once you have a `BackgroundWorker`, you can begin to use it by connecting it to the appropriate event handlers. A `BackgroundWorker` throws three events:

- The `DoWork` event fires when the `BackgroundWorker` begins its work. But here's the trick—this event is fired on a *separate* thread (which is temporarily borrowed from a thread pool that the Common Language Runtime maintains). That means your code can run freely without stalling the rest of your application. You can handle the `DoWork` event and perform your time-consuming task from start to finish.

NOTE *The code that responds to the `DoWork` event can't communicate directly with the rest of your application or try to manipulate a form, control, or member variable. If it did, it would violate the rules of thread safety (as you'll see later in this chapter), perhaps causing a fatal error.*

- The `ProgressChanged` event fires when you notify the `BackgroundWorker` (in your `DoWork` event handler) that the progress of the background task has changed. Your application can react to this event to update some sort of status display or progress meter.
- The `RunWorkerCompleted` event fires once the code in the `DoWork` handler has finished. Like the `ProgressChanged` event, the `RunWorkerCompleted` event fires on the main application thread, which allows you to take the result and display it in a control or store it in a member variable somewhere else in your application, without risking any problems. `RunWorkerCompleted` also fires when the background task is canceled (assuming you elect to support the Cancel feature).

To try out the BackgroundWorker, you can create a simple test. First, drop the BackgroundWorker component onto a form. Then attach the following DoWork event handler, which simply idles away ten seconds without doing anything. (We'll return to the Sleep() method later in the chapter.)

```
Private Sub BackgroundWorker1_DoWork(ByVal sender As System.Object, _
  ByVal e As System.ComponentModel.DoWorkEventArgs) _
  Handles BackgroundWorker1.DoWork

    ' This fires on a thread from the CLR thread pool.
    ' It's not safe to access the form here or any shared data
    ' (such as form-level variables).
    System.Threading.Thread.Sleep(TimeSpan.FromSeconds(10))

End Sub
```

WARNING *If you do break the rule in the above code and manipulate a control or form-level variable, you might not receive an error. But eventually you will cause a more serious problem under difficult-to-predict conditions, as described later in this chapter.*

Next you need to handle the RunWorkerCompleted event, in order to react when the background task is complete:

```
Private Sub BackgroundWorker1_RunWorkerCompleted( _
  ByVal sender As System.Object, _
  ByVal e As System.ComponentModel.RunWorkerCompletedEventArgs) _
  Handles BackgroundWorker1.RunWorkerCompleted

    ' This fires on the main application thread.
    ' It's now safe to update the form.
    MessageBox.Show("Time wasting completed!")

End Sub
```

The only thing remaining is to set the BackgroundWorker in motion when the form loads. To do this, call the BackgroundWorker.RunWorkerAsync() method. Here's the code that launches the BackgroundWorker when the form loads:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load

    BackgroundWorker1.RunWorkerAsync()

End Sub
```

When you run this application, you'll see a message box appear after ten seconds have elapsed.

Although this example is trivial, it should be clear that if you were doing something truly time-consuming in the DoWork event handler (like performing a database query or calling a web service on a remote computer), there would be a clear benefit: Your application would remain responsive as this work is taking place in the background.

In the next section, you'll see how to extend this pattern to use the BackgroundWorker in a more realistic application.

Transferring Data to and from the BackgroundWorker

One of the main challenges in multithreaded programming is exchanging information between threads. Fortunately, the BackgroundWorker includes a mechanism that lets you send initial information to the background thread and retrieve the result from it without any synchronization headaches.

To supply information to the BackgroundWorker you pass a single parameter to the RunWorkerAsync() method. This parameter can be any object type from a simple integer to a full-fledged object. However, you can only supply a single object. This object will be delivered to the DoWork event.

For example, imagine you want to calculate a series of cryptographically strong random digits. Cryptographically strong random numbers are random numbers that can't be predicted. Ordinarily, computers use relatively well-understood algorithms to generate random numbers. As a result, a malicious user can predict an upcoming "random" number based on recently generated numbers. This isn't necessarily a problem, but it is a risk if you need your random number to be secret.

For this operation, your code needs to specify the number of digits and the maximum and minimum value. In this case, you might create a class like this to encapsulate the input arguments:

```
Public Class RandomNumberGeneratorInput

    Private _NumberOfDigits As Integer
    Private _MinValue As Integer
    Private _MaxValue As Integer

    ' (Property procedures are omitted.)

    Public Sub New(ByVal numberOfDigits As Integer, _
      ByVal minValue As Integer, _
      ByVal maxValue As Integer)
        Me.NumberOfDigits = numberOfDigits
        Me.MinValue = minValue
        Me.MaxValue = maxValue
    End Sub

End Class
```

The form should provide text boxes for supplying this information and a button that can start the asynchronous background task. When the button is clicked, you'll launch the operation with the correct information. Here's the event handler that starts it all off:

```
Private Sub cmdDoWork_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdDoWork.Click
```

```
    ' Prevent two asynchronous tasks from being triggered at once.
    ' This is allowed but doesn't make sense in this application
    ' (because the form only has space to show one set of results
    ' at a time).
    cmdDoWork.Enabled = False
    ' Clear any previous results.
    txtResult.Text = ""

    ' Start the asynchronous task.
    Dim Input As New RandomNumberGeneratorInput( _
      Val(txtNumberOfDigits.Text), _
      Val(txtMin.Text), Val(txtMax.Text))
    BackgroundWorker1.RunWorkerAsync(Input)
End Sub
```

Once the `BackgroundWorker` acquires the thread, it fires a `DoWork` event. The `DoWork` event provides a `DoWorkEventArgs` object, which is the key ingredient for retrieving and returning information. You retrieve the input through the `DoWorkEventArgs.Argument` property, and return the result by setting the `DoWorkEventArgs.Result` property. Both properties can use any object.

Here's the implementation for a simple secure random number generator that's deliberately written to take almost 1,000 times longer than it should (and thereby make testing easier).

```
Private Sub BackgroundWorker1_DoWork(ByVal sender As System.Object, _
  ByVal e As System.ComponentModel.DoWorkEventArgs) _
  Handles BackgroundWorker1.DoWork

    ' Retrieve the input arguments.
    Dim Input As RandomNumberGeneratorInput = CType( _
      e.Argument, RandomNumberGeneratorInput)

    ' Create a StringBuilder to hold the generated random number sequence.
    Dim ResultString As New System.Text.StringBuilder()

    ' Start generating numbers.
    For i As Integer = 0 To Input.NumberOfDigits - 1
        ' Create a cryptographically secure random number.
        Dim RandomByte(1000) As Byte
        Dim Random As New _
          System.Security.Cryptography.RNGCryptoServiceProvider()

        ' Fill the byte array with random bytes. In this case,
        ' the byte array only needs a single byte.
        ' We fill it with 1000 just to make sure this is the world's slowest
        ' random number generator.
        Random.GetBytes(RandomByte)

        ' Convert the random byte into a decimal from MinValue to MaxValue.
        Dim RandomDigit As Integer
```

```
            RandomDigit = Int(RandomByte(0) / 256 * _
              (Input.MaxValue - Input.MinValue + 1)) + Input.MinValue

            ' Add the random number to the string.
            ResultString.Append(RandomDigit.ToString())
        Next

        ' Return the complete string.
        e.Result = ResultString.ToString()
    End Sub
```

TIP *In many cases, you'll want your `DoWork` event handler to call a method in another class to perform the actual work. This more extensively factored approach gives you greater flexibility—you can decide whether to perform the task synchronously, asynchronously, in multiple forms, or even in other applications (if you place the component in a separate class library assembly).*

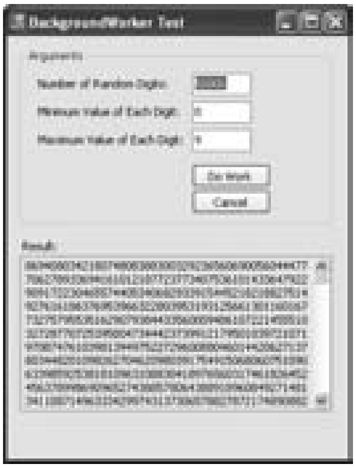
Once the handler completes, the `BackgroundWorker` fires the `RunWorkerCompletedEventArgs` on the user interface thread. At this point, you can retrieve the result from the `RunWorkerCompletedEventArgs.Result` property and update the interface and access form-level variables without worry:

```
Private Sub BackgroundWorker1_RunWorkerCompleted( _
  ByVal sender As System.Object, _
  ByVal e As System.ComponentModel.RunWorkerCompletedEventArgs) _
  Handles BackgroundWorker1.RunWorkerCompleted
    ' Show the results.
    txtResult.Text = CType(e.Result, String)

    ' Allow another operation to be started.
    cmdDoWork.Enabled = True
End Sub
```

This code completes the simple asynchronous random number generator shown in Figure 11-5. You can find this example (complete with the refinements for cancellation handling and progress tracking that you'll consider in the following sections) in the BackgroundWorkerTest project.

This simple application really demonstrates the power of threading. When you run it, you have no idea that any work is being carried out in the background. Best of all, the user interface remains responsive, which is not the case when timers are used. The user can click other buttons and perform other tasks while the time-consuming random number calculation is performed without any noticeable slowdown.

One reason multithreading works so well is that modern computers are so fast. Slowing down an application to execute several operations at once is a performance degradation that most applications can easily afford. Also, there's a little human psychology involved—in a user's experience, perception *is* reality.


BackgroundWorker Test

Arguments

Number of Random Digits: [illegible]

Minimum Value of Each Digit: 0

Maximum Value of Each Digit: 9

Result:

Figure 11-5: Generating random numbers asynchronously

Tracking Progress

There's no automatic way to report progress with the `BackgroundWorker` because it won't know how long your code will take to execute. However, the `BackgroundWorker` does provide built-in support for your `DoWork` code to read progress information and pass it to the rest of the application. This is useful for keeping a user informed about how much work has been completed in a long-running task.

To add support for progress reporting, you first need to set the `BackgroundWorker.WorkerReportsProgress` property to `True`. Then it's up to your code in the `DoWork` event handler to call the `BackgroundWorker.ReportProgresss()` method and provide an estimate of percentage complete (from 0% to 100%). You can do this as little or as often as you like. How is progress estimated? Each time you call `ReportProgress()`, the `BackgroundWorker` fires the `ProgressChanged` event. You can react to this event to read the new progress percentage and update the user interface. Because the `ProgressChanged` event fires on the user interface thread, there's no need for you to worry about marshalling your call to the correct thread.

Reporting progress usually involves a calculation, a call to another thread, an event, and a refresh of the form's user interface. Because of this overhead, you want to cut down the rate of progress reporting as much as possible. In our random number generator example, we do this by reporting progress in 1% increments only. Before the loop starts, a calculation is made to determine how many iterations must pass before a 1% progress change has occurred:

```
Dim ProgressIteration As Integer = Input.NumberOfDigits / 100
```

For example, if you are calculating 500 random numbers, progress will be reported every fifth iteration. This is a relatively quick calculation to make in the loop:

```
For i As Integer = 0 To Input.NumberOfDigits - 1
    If BackgroundWorker1.WorkerReportsProgress _
      And ProgressIteration > 0 Then
        If i Mod ProgressIteration = 0 Then
            BackgroundWorker1.ReportProgress(i / ProgressIteration)
        End If
    End If
    ...
Next
```

Now the only remaining step is to respond to the `ProgressChanged` event and update a `ProgressBar` control:

```
Private Sub BackgroundWorker1_ProgressChanged( _
  ByVal sender As System.Object, _
  ByVal e As System.ComponentModel.ProgressChangedEventArgs) _
  Handles BackgroundWorker1.ProgressChanged

    ProgressBar1.Value = e.ProgressPercentage
End Sub
```

Remember, you'll also need to reset the `ProgressBar.Value` at the beginning of each new operation. Figure 11-6 shows the revised program with a random number calculation in progress.

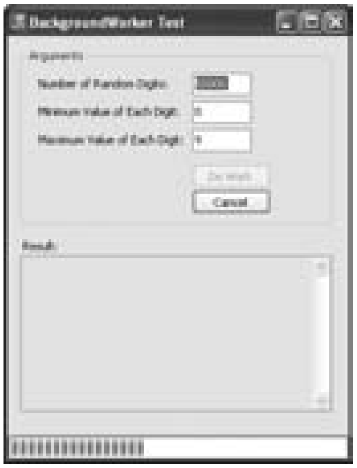


Figure 11-6: Tracking progress in an asynchronous task

Supporting a Cancel Feature

Another feature your users will appreciate is cancellation—the ability to halt an asynchronous task before it's complete if the information isn't needed. However, there's no generic way to support cancellation in the BackgroundWorker component. After all, your DoWork code might need to perform some cleanup before it can stop, or it might be in the middle of an operation that isn't safe to stop. However, the BackgroundWorker provides support for passing cancellation messages, which you can take advantage of. To enable this feature, first set the BackgroundWorker.WorkerSupportsCancellation property to True.

As long as WorkerSupportsCancellation is true, your form can call the BackgroundWorker.CancelAsync() method to request a cancellation. In this example, the cancellation is requested when a Cancel button is clicked:

```
Private Sub cmdCancel_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdCancel.Click

    BackgroundWorker1.CancelAsync()
End Sub
```

Nothing happens automatically when you call CancelAsync(). The code that's performing the task needs to explicitly check for a cancel request, set the DoWorkEventArgs.Cancel property to true, perform any required cleanup, and return. Here's how you can add this code to the loop in your DoWork code:

```
For i As Integer = 0 To Input.NumberOfDigits - 1
    If BackgroundWorker1.CancellationPending Then
         e.Cancel = true
        ' Return without doing any more work.
        Return
    End If
    ...
Next
```

Even when you cancel an operation, the RunWorkerCompleted event still fires. At this point, you can check whether the task was canceled and handle it accordingly.

```
Private Sub BackgroundWorker1_RunWorkerCompleted( _
  ByVal sender As System.Object, _
  ByVal e As System.ComponentModel.RunWorkerCompletedEventArgs) _
  Handles BackgroundWorker1.RunWorkerCompleted

    ' This fires on the main application thread.
    ' It's now safe to update the form.
    If e.Cancelled Then
        MessageBox.Show("Task canceled.")
    Else
        txtResult.Text = CType(e.Result, String)
    End If
    cmdDoWork.Enabled = True
    ProgressBar1.Visible = False
End Sub
```

Advanced Threading with the Thread Class

The BackgroundWorker is a great tool for implementing a single, straightforward background task. However, there are several situations in which you might want to use a more sophisticated (and more complex) approach. Here are some examples:

- You want to control the priority of a background task.
- You want to have the ability to suspend and resume a task (by suspending and resuming the thread that executes it).
- You want to reuse a thread over the lifetime of the application.
- You need to have a single thread perform multiple tasks and communicate with multiple forms or other classes.

Although there's no denying that the BackgroundWorker is a great tool for many common scenarios involving a single, asynchronously running background task, sooner or later nearly every Windows programmer is tempted to get his or her hands dirty with something a little more powerful. In the rest of this chapter, you'll get an overview of how you can use the Thread class from the System.Threading namespace to create and control threads at will.

A Simple Multithreaded Application

The first type of threaded program we will create using the Thread class is an *unsynchronized* multithreaded application. An unsynchronized application only spawns threads that perform independent tasks—that is, tasks that require no interaction with other parts of your application in order to do their work.

NOTE *The BackgroundWorker example obviously wasn't an example of unsynchronized multithreading—not only did it return a result after it finished its work, but it also reported progress along the way. However, in this scenario the distinction between unsynchronized and synchronized multithreading wasn't as important because the BackgroundWorker handles the messy plumbing without requiring any work from you.*

Before going any further, it makes sense to import the namespace that's used for threading so that its classes are easily accessible in your code:

```
Imports System.Threading
```

A thread runs a single method (technically, a procedure that takes no arguments and doesn't have a return value). Thus, before you can create a thread, you need to create a method and code the task you want to perform inside it. In this example, the method has a boringly simple task—every ten seconds, it writes a timestamp to a file named Alive.txt.

NOTE *There is no necessary relationship between a thread and an object. A single thread can run code that belongs to several objects. The methods of a single object can be executed by different threads.*

```
Private Sub WriteRegularTimeStamp()
    Dim LastUpdate As DateTime
    Do
        ' Write the file every 10 seconds.
        If DateTime.Now.Subtract(LastUpdate).TotalSeconds > 10 Then
            My.Computer.FileSystem.WriteAllText( _
              "c:\alive.txt", _
              DateTime.Now.ToLongTimeString + vbNewLine, True)
            LastUpdate = DateTime.Now
        End If

        ' You could use the following line of code to pause the thread,
        ' which makes the overall application more efficient.
        ' This example leaves it out, just to prove that
        ' multithreading works smoothly even with a CPU-intensive loop.
        'Thread.Sleep(TimeSpan.FromSeconds(10))
    Loop
End Sub
```

The thread is created and started in the handler for the `Click` event of a button:

```
Private Sub cmdStart_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdStart.Click

    Dim MyThread1 As New Thread(AddressOf WriteRegularTimeStamp)
    MyThread1.IsBackground = True
    MyThread1.Start()
End Sub
```

All of the threading logic is contained in just three lines. This code declares a thread for the appropriate procedure, and then sets its `IsBackground` property to `True`. This ensures that the thread will be a *background* thread. A background thread lasts only as long as another *foreground* thread is running (namely, the rest of your application). That way, the thread stops as soon as you exit the application. Another option would be to write some sort of termination condition in the loop, so that it only writes the timestamp a certain number of times before exiting. (Once the thread's method ends, the thread dies off.) Finally, you could also terminate a thread the hard way, by explicitly calling its Abort() method, as discussed later in this chapter. Generally, however, a background thread is a good choice for a noncritical task that can be shut down independently of any other application activity.

Once the thread is created, you simply need to call its Start() method to send it on its way.

TIP *The Start() method does not instantaneously start the thread. Instead, it notifies the Windows operating system, which then schedules the thread to be started. If your system is currently bogged down with a heavy task load, there could be a noticeable delay.*

You'll have no obvious indication that the thread is at work while your application is running. But after it's been at its work for a while, you'll find a list of timestamps in the Alive.txt file that look something like this:

```
3:28:22 PM
3:28:32 PM
3:28:42 PM
3:28:52 PM
3:29:02 PM
...
```

Sending Data to a Thread

There's an obvious drawback with this application the way it stands. Namely, it hard-codes the file path. What if you want to move the Alive.txt file to another directory, or you want to run two threads, each of which will be working with its own Alive.txt file? You need a way to pass some date to your thread—namely, the full path of the file you want to use.

You might think that this modification would work:

```
Private Sub WriteRegularTimeStamp(filePath As string)
```

Here, the WriteRegularTimeStamp() method is modified to accept a string argument. Unfortunately, this isn't allowed. Thread objects can only point to a method that takes no parameters.

The best way to get around this limitation is to create a class that encapsulates the procedure that you want to use and any data that it needs. In this case, the only data required is the path to the file to be modified. The following class works well for the desired purpose:

```
Public Class TimeStamper
    Private filePath As String

    Public Sub New(ByVal filePath As String)
        Me.filePath = filePath
    End Sub

    Public Sub WriteRegularTimeStamp()
```

```
        Dim LastUpdate As DateTime
            Do
            ' Write the file every 10 seconds.
            If DateTime.Now.Subtract(LastUpdate).TotalSeconds > 10 Then
                My.Computer.FileSystem.WriteAllText( _
                  filePath, DateTime.Now.ToLongTimeString + vbNewLine, True)
                LastUpdate = DateTime.Now
            End If
        Loop
    End Sub
End Class
```

As a nice touch, each instance of this class receives its path string as an argument in its constructor, rather than forcing you to set it through a property.

You can now modify the code in the click event handler. Here's an example that starts two threads off, each with a different file:

```
Private Sub cmdStart_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles cmdStart.Click

    Dim Stamper1 As New TimeStamper("c:\alive1.txt")
    Dim MyThread1 As New Thread(AddressOf Stamper1.WriteRegularTimeStamp)
    MyThread1.IsBackground = True

    Dim Stamper2 As New TimeStamper("c:\alive2.txt")
    Dim MyThread2 As New Thread(AddressOf Stamper2.WriteRegularTimeStamp)
    MyThread2.IsBackground = True

    ' Start both threads.
    MyThread1.Start()
    MyThread2.Start()
End Sub
```

If you run the program now, you'll find that it works more or less the same as before. Under the hood, however, the design is much more elegant and extensible. You can find this code in the ThreadTest project, with the samples for this chapter. (This sample project also uses the notification technique described in the following section, so it reports to the user every time the file is stamped.)

One of the reasons this works so well is that each thread has its own data. There's no need to worry about exchanging or synchronizing information, as each thread is independent. If you stick to this type of multithreading, you'll have little to worry about.

In many situations, unsynchronized multithreading can be very useful. For example, you might want to process a batch of data while waiting for the user to enter more information. Or, you might want to create something like a graphical arcade game, where the background music is handled by a separate thread that queues the appropriate music files.

Threading and the User Interface

One of the reasons our examples have worked so well is that the information the threads return is sent directly to the appropriate label control in the window. There is no need for the main program to determine whether a thread is finished, or to try to retrieve the result of its work. Most real-world programs don't work this way. It is far more common (and far better program design) for an application to use a thread to perform a series of calculations, retrieve the results once they are ready, and then format and display them in the user interface, if necessary.

This technique isn't as easy as it seems. For example, imagine you want to modify the previous threading example so that every time it writes a new timestamp to the Alive.txt file, it updates the text in a status bar. This seems like a trivial task—after all, you simply need to tweak the text in the `WriteRegularTimeStamp()` method, right?

Wrong. In fact, if you attempt to interact with a Windows control from another thread, the results could be disastrous. While you're debugging a prerelease version your application, most controls are kind enough to throw an exception to warn you when you've made this mistake, but when you compile a release version of your application, these checks disappear (for better performance). In this environment, you won't get an error, and your code might work fine in many cases. But under certain difficult-to-predict conditions, your application will lock up. Tricks that you might think would get you out of this mess won't help. For example, you might try to dodge the problem by firing an event from the code that's performing the background work (like the file stamping in the previous example). Then the form can handle that event and update the user interface safely, right? Not so fast. It turns out that it doesn't matter *where* you write the code—even if you place the event handler in your form, it's still going to be executed on the time stamper thread, because the time stamper thread fires the event. So this approach just creates the same problem in another location.

Fortunately, there is a solution. .NET provides a way to force a code routine to run on the user interface thread. You just need to follow these steps:

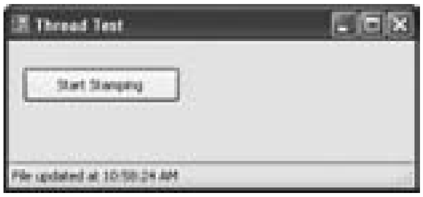
1. Put your otherwise unsafe code into a separate procedure.
2. Create an instance of the `MethodInvoker` delegate, and point it to this procedure.
3. Call the `Invoke()` method on any control or form in your application, and pass it the `MethodInvoker` delegate as an argument.

`Invoke()` is the only user interface method that's safe to call from another thread. It triggers the code you specify through the delegate to execute it on the safe user-interface thread. You can also check a control's `InvokeRequired` property to determine whether the current code is running on the user-interface thread or on another thread. This allows you to determine whether you need to call `Invoke()` when modifying the control.

Here is a revised TimeStamper example that uses the Invoke() solution and reports its last update in a label in a thread-safe manner (the changed lines are highlighted in bold):

```
Public Class TimeStamper
    Private filePath As String
    Private statusLabel As Control

    Public Sub New(ByVal filePath As String)
        Me.filePath = filePath
    End Sub

    Public Sub New(ByVal filePath As String, ByVal statusLabel As Control)
        Me.filePath = filePath
        Me.statusLabel = statusLabel
    End Sub

    Private LastUpdate As DateTime
    Public Sub WriteRegularTimeStamp()
        Do
            ' Write the file every 10 seconds.
            If DateTime.Now.Subtract(LastUpdate).TotalSeconds > 10 Then
                My.Computer.FileSystem.WriteAllText( _
                  filePath, DateTime.Now.ToLongTimeString + vbNewLine, True)
                LastUpdate = DateTime.Now

                ' Perform the update on the right thread.
                Dim method As New MethodInvoker(AddressOf UpdateStatusLabel)
                statusLabel.Invoke(method)
            End If
        Loop
    End Sub

    Private Sub UpdateStatusLabel()
        statusLabel.Text = "File updated at " + _
          LastUpdate.ToLongTimeString()
    End Sub
End Class
```

Figure 11-7 shows the revised application in action. This example is provided with the sample code as the ThreadTest project.



File updated at 10:58:24 AM

Figure 11-7: Updating the user interface from another thread (safely)

This example shows one way to get information out of a thread and back into the rest of your application. But directly updating a user interface element is not the only approach. In many cases, you want a way to transfer information to some sort of variable, where other code can access it later as needed. You'll see this issue later when we tackle synchronization.

Basic Thread Management

The previous example takes advantage of some simplifications. For one thing, it assumes that you can create a thread and then leave it to do its work without ever worrying about it again. In the real world, however, you often need to know when a thread has completed its work. You might even need to pause or kill a thread.

Thread Methods

You've already seen how to start a thread. You can also stop a thread by using the `Abort()` method, which will finish it off by raising a `ThreadAbortException`.

```
MyThread.Abort()
```

Your thread class can then handle this exception in order to try to end as gracefully as possible, performing any necessary cleanup in a `Finally` block. However, the `ThreadAbortException` can never be killed off. Even if you catch it, once the cleanup code finishes, the exception will be thrown again to end the code in the thread procedure.

Using the `Abort()` method is a relatively crude way to stop a thread. You might use it to reign in an otherwise unresponsive thread, but it's not an ideal mechanism. It's more typical for a long-running thread to take the responsibility of polling a variable that indicates whether or not it should continue, as shown below. This relies on the thread being well behaved, but it also allows processing to be interrupted at a natural stopping point, as opposed to being unpredictably interrupted with an exception. If you wrap your thread in a class, it makes sense for this to be a public class variable or property.

```
Private Sub ThreadFunction()
    Do Until ThreadStop = true
        ' Do some work here.
    Loop
End Sub
```

You can also pause and resume a thread with the `Suspend()` and `Resume()` methods:

```
MyThread.Suspend()
' Do something in the foreground that requires a lot of CPU work.
MyThread.Resume()
```

TIP *The `Suspend()` and `Resume()` methods generally aren't used much in multithreaded applications, because they can easily lead to deadlocks (as you'll see later in this chapter). If the suspended thread has a lock on a resource that another thread needs, the other thread will be forced to stop processing as well. If both threads are holding on to resources that the other needs, neither can continue. A better approach to managing shared resources is to use thread priorities, which are introduced in the next section.*

As you've already seen, you can pause a thread for a preset amount of time using the shared `Sleep()` method:

```
Thread.Sleep(TimeSpan.FromSeconds(1))
```

This is a common method to use in a CPU-intensive or disk-intensive process to provide a bit of time during which other threads can get their work done. The example here uses the `TimeSpan` class to send the thread to sleep for one second, which makes the resulting code very readable.

Another commonly used method is `Join()`. It waits for a thread to complete.

```
MyThread.Join()
```

When you use the `Join()` method, your code becomes *synchronous*, meaning that the thread executing the `Join()` will not progress until the waited-for thread is finished. In the above example, the code won't continue until `MyThread` finishes its work. The `Join()` method can also be used with a `TimeSpan` that specifies the maximum amount of time that you will wait before continuing.

TIP *When you abort a thread with the `Abort()` method, it does not necessarily terminate immediately, because the thread may be running exception-handling code. If you need to ensure that the thread has stopped before continuing, use the `Join()` method on the thread after calling the thread's `Abort()` method.*

And how do you know what a thread is up to? You can examine its `ThreadState` property and compare it against the possible enumerated values. Here's an example:

```
MyThread.Join(TimeSpan.FromSeconds(10))
If (MyThread.ThreadState And ThreadState.Stopped) = _
  ThreadState.Stopped Then
    MessageBox.Show("We waited with Thread.Join, and the thread finished.")
ElseIf (MyThread.ThreadState And ThreadState.Running) = _
  ThreadState.Running Then
    MessageBox.Show( _
      "We waited 10 seconds, but the Thread is still running.")
End If
```

Figure 11-8 shows the different stages in a thread's execution.

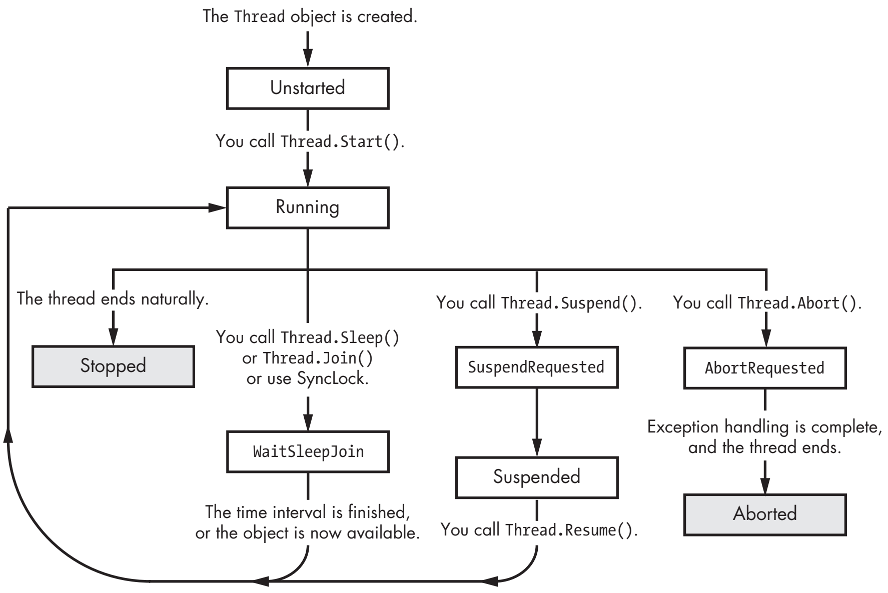

The Thread object is created.
Unstarted
You call Thread.Start().
Running
The thread ends naturally.
Stopped
You call Thread.Sleep() or Thread.Join() or use SyncLock.
WaitSleepJoin
The time interval is finished, or the object is now available.
You call Thread.Suspend().
SuspendRequested
Suspended
You call Thread.Resume().
You call Thread.Abort().
AbortRequested
Exception handling is complete, and the thread ends.
Aborted

Figure 11-8: The life cycle of a thread

Thread Priorities

All threads are created equal, but they don't have to stay that way. *Priorities* allow you to make sure that some threads are always executed preferentially. Threads with low priorities, on the other hand, may not do much work if the system is heavily bogged down with other, higher priority tasks.

You can set a thread's priority to various values: `AboveNormal`, `BelowNormal`, `Highest`, `Lowest`, and `Normal`, which is the default. These priorities are relative; they are significant only in the way that they compare with the priorities of other currently executing threads in your program or in other programs. If all of your application's threads have the same priority, it doesn't make much difference whether that priority is `Normal` or `Highest` (assuming, for the moment, that there aren't any other programs or processes competing for the CPU's attention).

Setting a thread's priority is straightforward:

```
MyThread.Priority = ThreadPriority.Lowest
```

A thread with a high priority may need to use the `Sleep()` method to allow other threads a chance to get their work done. Fine-tuning this sharing of the CPU is an art that requires significant trial-and-error experimentation.

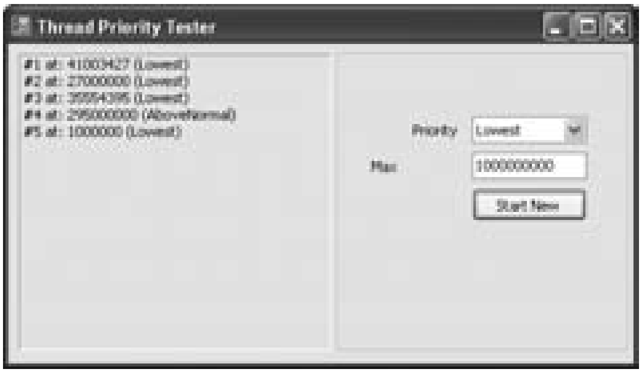
When Is Too Much Not Enough?

As just mentioned, you must be careful about using high priorities. If you have too many aggressive threads, some threads may not receive enough CPU time to be able to perform their work properly. The sorry state that results when too many threads compete for too few resources is called *thread starvation*, and it can make an application perform poorly, or render some functions inoperative.

TIP *When using threads, it's a good idea to test them on the minimum system configuration that your application will support.*

Thread Priority Example

The online chapter sample code provides a project named ThreadPriorities that allows you to satisfy your curiosity and create as many simultaneous threads as you want (see Figure 11-9). These threads "compete" to increment their individual counter variables. The ones that receive the most CPU time will increment their counters the fastest.



```
#1 at: 41003427 (Lowest)
#2 at: 270000000 (Lowest)
#3 at: 35554395 (Lowest)
#4 at: 295000000 (AboveNormal)
#5 at: 1000000 (Lowest)
```

Max: 1000000000

Figure 11-9: Testing threads

A separate thread class, `ThreadCounter`, provides this counter functionality and incorporates a Boolean "stop signal" variable named `ThreadStop`:

```
Public Class ThreadCounter

    Public LoopCount As Integer
    Public MaxValue As Integer
    Public Priority As String
    Public ThreadStop As Boolean

    Public Sub New(ByVal MaxValue As Integer, ByVal Priority As String)
        Me.MaxValue = MaxValue
        Me.Priority = Priority
    End Sub

    Public Sub Refresh()
        ' Increment the counter.
        For LoopCount = 0 To MaxValue - 1
            ' Check for the signal to stop abruptly.
            If ThreadStop = True Then Exit For
        Next
    End Sub

End Class
```

The interesting part about this program is that it uses a collection called ActiveCounters to store references to all the objects that are running on the various threads (and another collection called ActiveThreads to store references to the Thread objects). Periodically, a timer fires, and a routine in the form code loops through the ActiveCounters collection and prints out the status of every thread in a label.

```
Private Sub tmrThreadMonitor_Tick(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles tmrThreadMonitor.Tick

    lblThreads.Text = ""
    Dim Counter As ThreadCounter
    Dim i As Integer

    For Each Counter In ActiveCounters
        i += 1
        lblThreads.Text &= "#" & i.ToString() & " at: "
        lblThreads.Text &= Counter.LoopCount.ToString() & " ("
        lblThreads.Text &= Counter.Priority & ")"
        lblThreads.Text &= vbNewLine
    Next

End Sub
```

The ThreadPriorities program allows you to set the priority of each thread when you create it. This allows you to verify that a high-priority thread will increment its counter much faster than a low-priority one. You'll also notice that when you create a thread with a high priority, your application (and your computer) will become noticeably less responsive until the thread is finished incrementing its counter.

When you end the program, it performs some graceful cleanup by iterating through the ActiveThreads collection and stopping each thread. Rather than use the Abort() method, this program does things the nice way, setting the ThreadStop member variable of each ThreadCounter object, and then waiting on each thread's termination with the Join() method to verify that it has stopped. This is actually much faster than aborting each thread, because it avoids the overhead of wresting control of the thread and throwing an exception.

Here's the code:

```
Private Sub ThreadPriorityTester_Closing(ByVal sender As Object, _
  ByVal e As System.ComponentModel.CancelEventArgs) Handles MyBase.Closing

    ' Signal each thread to stop.
    Dim Counter As ThreadCounter
    For Each Counter In ActiveCounters
        Counter.ThreadStop = True
    Next

    ' Wait to verify that each thread has stopped.
    Dim CounterThread As Thread
    For Each CounterThread In ActiveThreads
        CounterThread.Join()
    Next

End Sub
```

Thread Debugging

One very useful technique when debugging a multithreaded project is to assign each thread a name. This allows you to distinguish one thread from another and to verify which thread is currently executing. It's not unusual when debugging a tricky problem to discover that the thread you thought was at work actually isn't responsible for the problem.

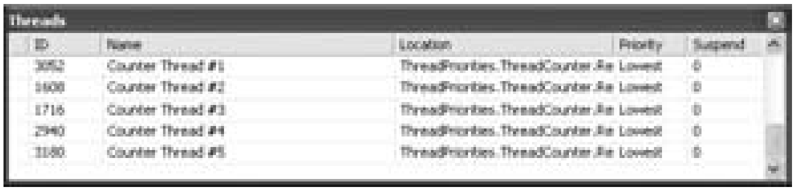
To name a thread with a descriptive string, use the `Thread.Name` property:

```
CounterThread.Name = "Counter 1"
```

To check which thread is running a given code procedure at a specific time, you can use code like this, which uses the shared `CurrentThread()` method of the `Thread` class:

```
MessageBox.Show(Thread.CurrentThread.Name)
```

Visual Studio also provides some help with a Threads debugging window (Figure 11-10). This window shows all the currently executing threads in your program and indicates the thread that currently has the processor's attention with a yellow arrow. The `Location` column even tells you what code the thread is running.


Threads

ID	Name	Location	Priority	Suspend
3052	Counter Thread #1	ThreadPriorities.ThreadCounter.Re	Lowest	0
1608	Counter Thread #2	ThreadPriorities.ThreadCounter.Re	Lowest	0
1716	Counter Thread #3	ThreadPriorities.ThreadCounter.Re	Lowest	0
2940	Counter Thread #4	ThreadPriorities.ThreadCounter.Re	Lowest	0
3180	Counter Thread #5	ThreadPriorities.ThreadCounter.Re	Lowest	0

Figure 11-10: Controlling threads at runtime

NOTE *To access the Threads window, you need to pause your program's execution and choose Debug ▸ Windows ▸ Threads. You can then use some advanced features for controlling threads. For example, you can set the active thread by right-clicking a thread and selecting Switch to Thread. You can also use the `Freeze` command to instruct the operating system to ignore a thread, giving it no processing time until you select the corresponding `Thaw` command to restore it to life. This fine-grained control is ideal for isolating problematic threads in a misbehaving application. The Threads window includes other threads that you haven't created, but are a part of .NET. As a general rule of thumb, you should assign names to all the threads you create so that you can identify each one in the list.*

Thread Synchronization

The mistake that most novice programmers make when they start creating multithreaded applications is simple: They assume that everything they want to do is thread-safe. In other words, they assume that any action that can be performed by a synchronous piece of code can be moved into a thread. This is a dangerous mistake that ignores the effects of concurrency.

Potential Thread Problems

Remember, threads work almost simultaneously as Windows switches from one thread to another. This means that each time you run the application, the relative order of execution of the multithreaded code may vary. Sometimes Thread A might perform a given action before Thread B, but at other times Thread B might take the lead. You can configure the priorities of individual threads, as you've seen, but you can never be absolutely sure when a thread will act, or what the order will be for operations on different threads.

What's more, if you have more than one thread manipulating the same object, eventually at least two of them will try to use it at once. Consider a situation where you have a global counter used by multiple threads for keeping track of the number of times an operation takes place. Sooner or later, Thread A will try to increment the value from, say, 10 to 11 at the same time that Thread B is also trying to increment the value from 10 to 11. The result? In this case, the count will be set to 11, even though the final value should really be 12. The more threads there are (and the greater the delay between reading and updating the counter variable), the worse the problem will become.

Concurrency and synchronization problems are particularly tricky because they often don't show up when an application is being tested, but only lead to bugs later in unpredictable situations, after the application has been deployed. If you neglect giving adequate consideration to synchronization issues at design time, there is no way to know when a problem could appear. Many programmers don't realize the dependencies of the objects they are using. Trying to use an object concurrently when that object has not been designed to be thread-safe is likely to cause a runtime exception in the best case, and a more insidious data (and more difficult to spot) data error in the worst case.

NOTE *Most classes in the .NET Framework are not thread-safe, because adding the required synchronization code would dramatically slow down their performance.*

Basic Synchronization

The best approach to avoiding data synchronization problems is often to refrain from modifying variables that are accessible to multiple threads. If this isn't possible, the next best thing is to use synchronization (which is also known as *locking*). The idea behind synchronization, as mentioned earlier, is to acquire a lock on a resource before you access it so that other threads that try to access the resource will be forced to wait. This process prevents collisions, but it also slows down performance.

All you need to do is place code that uses shared objects inside a SyncLock/End SyncLock block. The first line of this block identifies the data item that is being synchronized. This item must be a reference type, such as an object or an array; it can't be a simple value type.

When you use the SyncLock statement, your application waits until it has exclusive access to the object you've specified before performing the commands in the SyncLock block. While these commands are being executed, any other thread that tries to access the synchronized object will be temporarily suspended by the operating system. When the final End SyncLock statement is reached the lock is released and the operating system gives other threads the opportunity to access that object. Again, although this guarantees thread safety, performance can suffer, because all threads trying to access a locked object are blocked.

A Sample Synchronization Problem

To demonstrate how synchronization works, we will use a variant of a global counter program. There are many different ways to observe the effects of thread synchronization problems, but this one gives a quick demonstration of the potential hazards.

The first ingredient is a GlobalCounter class:

```
Public Class GlobalCounter
    Public Counter As Integer
End Class
```

An instance of this class is provided as a public variable in the form class:

```
Public MyGlobalCounter As New GlobalCounter()
```

There is also a class that wraps our threaded operations, as before:

```
Public Class IncrementThread

    Private Counter As GlobalCounter
    Private LocalCounter As Integer
    Private ThreadLabel, GlobalLabel As Label

    Public Sub New(ByVal Counter As GlobalCounter, _
      ByVal ThreadLabel As Label, ByVal GlobalLabel As Label)
        Me.Counter = Counter
        Me.ThreadLabel = ThreadLabel
        Me.GlobalLabel = GlobalLabel
    End Sub

    Public Sub Increment()
        Dim i As Integer
        Dim GlobalCounter As Integer
        For i = 1 To 1000
            LocalCounter = LocalCounter + 1
            GlobalCounter = Counter.Counter
            Thread.Sleep(TimeSpan.FromTicks(1))
            Counter.Counter = GlobalCounter + 1
         Next i

        ' Assume that ThreadLabel and GlobalLabel are on the same window.
        Dim Invoker As New MethodInvoker(AddressOf UpdateLabel)
        ThreadLabel.Invoke(Invoker)
    End Sub

    Private Sub UpdateLabel()
        ThreadLabel.Text = LocalCounter.ToString()
        GlobalLabel.Text = Counter.Counter.ToString()
    End Sub

End Class
```

This threading class wraps two counters—a local counter (stored in an integer) and a global counter, which is stored in an object so it can be shared between several different `IncrementThread` objects. When `IncrementThread` finishes its work (coded in the `Increment()` method), it uses the thread-safe `MethodInvoker` described earlier to update the user interface with information based on both its local and global counters.

A deliberate feature of this example is the way that the global counter is incremented. Instead of doing it in one instruction (`GlobalCounter = Counter.Counter + 1`), our example uses two lines, pausing the thread for one tick (a small interval of time equal to 100 nanoseconds) in between the time that the counter value is read and the time that the counter is updated. This pause is meant to simulate thread latency and therefore increase the likelihood that synchronization issues will occur when this example is run. In realistic scenarios, synchronization issues occur only when many more threads than are at work here compete for the CPU. (Remember, one of the most devious aspects of synchronization problems is that they often don't come out of the woodwork when you are testing under simple conditions.)

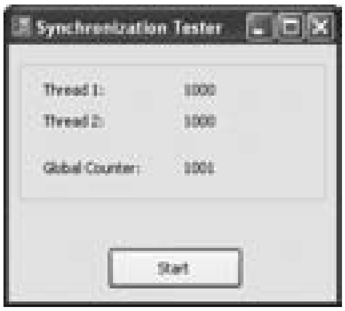
As before, the threads are created and started in a `Click` event handler for a button on the form. The following code is used:

```
Private Sub cmdStart_Click(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles cmdStart.Click
    MyGlobalCounter.Counter = 0
    Dim Increment1 As New IncrementThread(MyGlobalCounter, _
      lblThread1, lblGlobal)
    Dim Increment2 As New IncrementThread(MyGlobalCounter, _
      lblThread2, lblGlobal)
    Dim MyThread1 As New Thread(AddressOf Increment1.Increment)
    Dim MyThread2 As New Thread(AddressOf Increment2.Increment)
    MyThread1.Start()
    MyThread2.Start()
End Sub
```

The result is shown in Figure 11-11.

Each thread has kept track of its own private local counter value, so that much is accurate. However, the global counter is completely wrong. It should be 2,000, reflecting that each of the two threads incremented it 1,000 times. Instead, the competing threads performed overlapped edits that didn't take each other's actions into account. Here's an example of how the problem occurs (assuming the counter's current value was 12):

1. MyThread1 reads the value 12.
2. MyThread2 reads the value 12.
3. MyThread1 sets the value to 13.
4. MyThread2 sets the value to 13—instead of 14.



Thread 1: 1000
Thread 2: 1000
Global Counter: 1001

Figure 11-11: A flawed global counter

Using SyncLock to Fix the Problem

In this case, the fix is quite easy. Because `GlobalCounter` is a full-fledged object, you can use `SyncLock` while your thread is executing to gain exclusive access to the global counter. (If `GlobalCounter` was only a variable, this solution wouldn't be possible.)

In fact, all you need to do is to add two lines to the `Increment()` method, as shown here:

```
Public Sub Increment()
    Dim i As Integer
    Dim GlobalCounter As Integer
    For i = 1 To 1000
        LocalCounter = LocalCounter + 1
        SyncLock Counter
            GlobalCounter = Counter.Counter
            Thread.Sleep(TimeSpan.FromTicks(1))
            Counter.Counter = GlobalCounter + 1
        End SyncLock
    Next i

    Dim Invoker As New MethodInvoker(AddressOf UpdateLabel)
    ThreadLabel.Invoke(Invoker)
End Sub
```

Now the result, as shown in Figure 11-12, will be correct. You can see the complete code in the `ThreadingSynchronization` project.

If you timed the application, you might notice that it has slowed down. All of the automatic pausing and resuming of threads creates some overhead. But when you consider the frustrating problems that `SyncLock` can help you avoid, you'll be eager to put it to work in your applications.

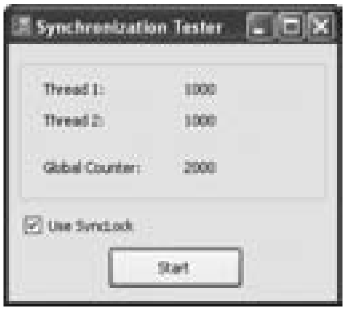


Figure 11-12: A successful global counter

What Comes Next?

This chapter has endeavored to give you a solid understanding of the fundamentals of threading, and a knowledge of the issues involved. Mastering all the aspects of threading could almost be a life's work, and many books and articles have been written on the subject.

If you're in search of more threading information, the best place to start is the documentation. Both the Visual Studio Help and the MSDN website provide white papers describing the technical details of threading, along with code examples that show it in action in live applications.