Comprehensive VB .NET Debugging

MARK PEARCE

ISBN (pbk): 1-59059-050-3

Printed and bound in the United States of America 12345678910

Technical Reviewer: Pamela Fanstill

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Managers: Sofia Marchant, Nicole LeClerc

Copy Editor: Nicole LeClerc

Compositor and Proofreader: Impressions Book and Journal Services, Inc.

Indexer: Ann Rogers

Artist and Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 14

Debugging Multithreaded Applications

"Multithreading" is a word that strikes fear into the hearts of many VB.Classic developers. Although VB 5.0 introduced a restricted type of multithreading called *apartment* threading, VB .NET is the first version of Visual Basic to have proper *free* threading. This means that you've been given a very powerful tool to make your application faster and more responsive to its users. As with any powerful technology, it requires that you have a good understanding of the benefits and drawbacks in order to use it properly and safely.

When I started skydiving many years ago, I never forgot the sign above the clubhouse door. It read simply "Knowledge dispels fear." In no area of software development is this saying more true than multithreading. Most Visual Basic developers are relatively unfamiliar with the subject, and even those programmers who tried VB.Classic development using apartment threading will be worried when faced with free threading. The key to avoiding threading bugs lies in a good knowledge of what can go wrong and why it goes wrong.

This chapter looks at how and why multithreading is so difficult, and gives you some knowledge and tools that will help you to tackle this subject safely. It starts with a quick look at how multithreading works and why it's so difficult to do correctly. Then it examines when multithreading is, and isn't, useful. Knowing when *not* to use multithreading is important because it can save you a lot of time in testing and debugging.

The next section of this chapter looks in more detail at the sorts of bugs that can be introduced when writing multithreaded programs. It demonstrates why it's necessary to design your multithreaded applications to avoid bugs rather than trying to remove the bugs later. The elusive nature of many multithreading bugs means that the normal code ➢ test ➢ debug cycle doesn't work well when writing multithreaded programs.

I'll show you the four main types of threading problems in some detail, using small example applications to illustrate the issues and some possible solutions. The final example application shows you how to use multithreading safely in a typical graphical user interface (GUI) program. It looks at debugging a relatively simple multithreaded application that uses messages to pass information between a user interface thread and a background thread, thus avoiding thread

synchronization problems. You'll also learn how to propagate an exception across thread boundaries, even when using asynchronous threads.

Multithreading Basics

Figure 14-1 shows a simple multithreaded program using two threads that access a single instance of three separate classes.

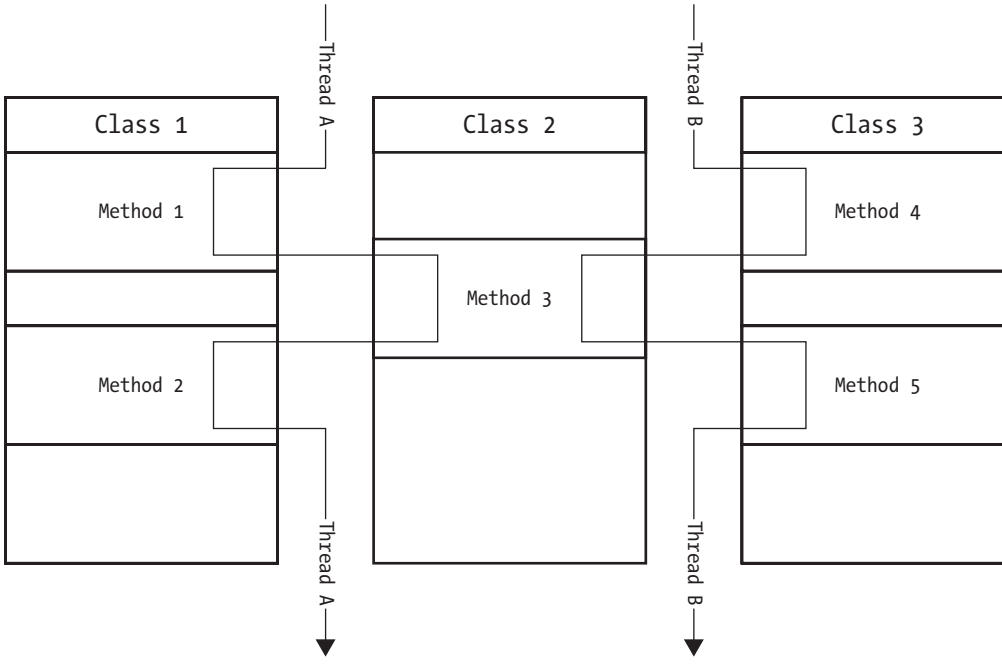

Figure 14-1. A simple multithreaded application

Thread A executes the code in class objects 1 and 2, and thread B runs through the code in class objects 2 and 3. Both threads access a common method in the instance of class 2, thereby sharing the code and data that this method uses.

Why Multithreading Is So Difficult

Of course, on a single-processor machine, the two threads don't actually execute simultaneously. What happens is that the operating system interleaves instructions from the two threads to give the impression that the threads are

executing together. Whenever control is switched from thread A to thread B, the processor saves the context of thread A, restores the context of thread B, and then starts running it. As soon as an instruction from thread A is reached, the same process happens in reverse. Because this all happens so fast, you receive the impression that both threads are executing simultaneously.

Say that you have two threads containing just ten source code instructions. How many ways can these ten instructions be interleaved together? The answer, in case you haven't got a calculator handy, is 184,756! This starts to look worrying. How can you possibly test 184,756 possible code paths? Unfortunately, the situation is actually much worse than this. Threads aren't interleaved together at the level of source code, or even CIL, but at the assembly code level. When a single source statement can translate to dozens of native code instructions, you can see the impossibility of using execution testing to verify that your multithreaded application is working correctly.

So if you can't test a multithread program using code coverage tools, how about trying to desk-check it? Is it possible to examine the source code thoroughly enough to be able to predict problems resulting from multiple threads? Well, it's a nice idea that combining lots of brainpower with lots of multithreading experience can help you to find and remove the problems. The truth is that hard-earned evidence gained from very experienced developers has demonstrated that this doesn't work. Developers' brains simply aren't equipped to cope with understanding how multiple execution threads can interact with each other.

Because multithreading involves a nonlinear process, and a developer is unable to establish the exact flow of execution because the interleaving of threaded code happens at the level of assembly code, many of the bugs occur rarely and seemingly at random. Even moving a multithreaded program from a machine with a slow processor to one with a much faster processor can cause bugs to appear and disappear, because the processor speed can prevent or cause "racing" between threads.

Before looking at "racing" and other bug types that can be caused by the use of multiple threads, it's worth examining when multithreading can be useful to you, and when it doesn't give you the benefits that you might think.

Multithreading Advantages

One of the best and most common uses of multithreading is to keep the user interface of your program responsive to the user while also performing one or more background tasks. For instance, the end user might ask your application to calculate the current market value of a sophisticated financial derivative instrument. During this lengthy calculation, you want the user to be able to cancel the operation if it's taking too long, perhaps by clicking a Cancel button. This

chapter's last example application shows a very similar scenario to this, and multithreading done properly works very well for this type of situation.

Another good use of multithreading is to keep the user interface updated with the intermediate results of a background task that's running. For instance, in the scenario just discussed, you might want to display intermediate results from the option valuation on the user interface while the calculation continues. If you don't use multithreading and the option calculation runs on the same thread as the user interface, you'll find that the application's window will go blank and won't be repainted. This is because the single thread can't cope with performing both tasks simultaneously.

An associated advantage of multithreading comes when you have a task that's going to take a long time to complete. In this case, you can fire off a background thread to perform the task and forget about it until the task completes at some time in the future. In the case of a background thread, this thread will be terminated automatically if the process that launched it is finished. Some dangers are associated with this automatic termination of a background thread because the termination is done through the CLR calling **Thread.Abort**. For a discussion of the dangers associated with **Thread.Abort**, please see the section titled "Terminating a Managed Thread" later in this chapter.

Yet another advantageous use of multithreading is to spawn a new thread for each user request to a server application. This allows multiple users to be serviced without the delay that might happen if the user requests are serialized and processed just one at a time. The built-in thread pool supplied by .NET is often an excellent solution to this situation.

If your application is doing input/output (I/O) work, such as accessing a disk, a printer, or the network, these resources can have unpredictable delays. Multiple threads can help to prevent I/O latency affecting other parts of your application.

You can use threads to isolate critical subsystems of your application from noncritical subsystems. Because most thread exceptions won't propagate out of the thread, this prevents an error in, say, the printing subsystem of your application affecting the radiation dosage monitoring subsystem.

A final reason for using multithreading is to establish the priority of an application's competing tasks. You can set the priority of a thread when it's created, so an important task can be assigned a high priority while less important tasks can be given a lower priority.

Multithreading Disadvantages

This section presents some of the disadvantages of writing multithreading code. There's certainly no need to use multithreading just because it's there and it's cool.

Using multithreading on a single-processor machine to process multiple tasks where each task takes approximately the same time isn't always very effective. For example, you might decide to spawn ten threads within your program in order to process ten separate tasks. If each task takes approximately 1 minute to process, and you use ten threads to do this processing, you won't have access to any of the task results for the whole 10 minutes. If instead you processed the same tasks using just a single thread, you would see the first result in 1 minute, the next result 1 minute later, and so on. If you can make use of each result without having to rely on all of the results being ready simultaneously, the single thread might be the better way of implementing the program.

If you launch a large number of threads within a process, the overhead of thread housekeeping and context switching can become significant. The processor will spend considerable time in switching between threads, and many of the threads won't be able to make progress. In addition, a single process with a large number of threads means that threads in other processes will be scheduled less frequently and won't receive a reasonable share of processor time.

If multiple threads have to share many of the same resources, you're unlikely to see performance benefits from multithreading your application. Many developers see multithreading as some sort of magic wand that gives automatic performance benefits. Unfortunately multithreading isn't the magic wand that it's sometimes perceived to be. If you're using multithreading for performance reasons, you should measure your application's performance very closely in several different situations, rather than just relying on some nonexistent magic.

Coordinating thread access to common data can be a big performance killer. Achieving good performance with multiple threads isn't easy when using a coarse locking plan, because this leads to low concurrency and threads waiting for access. Alternatively, a fine-grained locking strategy increases the complexity and can also slow down performance unless you perform some sophisticated tuning.

Using multiple threads to exploit a machine with multiple processors sounds like a good idea in theory, but in practice you need to be careful. To gain any significant performance benefits, you need to learn about thread balancing. For instance, imagine an application that receives incoming price information from the network, aggregates and sorts that information, and then displays the results on the screen for the end user. With a dual-processor machine, it makes sense to split the task into, say, three threads. The first thread deals with storing the incoming price information, the second thread processes the prices, and the final thread handles the display of the results. After implementing this solution, you find that the price processing is by far the longest stage, so you decide to rewrite that thread's code to improve its performance by a factor of three. Unfortunately, this performance benefit in a single thread may not be reflected across your whole application. This is because the other two threads may not be able to keep pace with the improved thread. If the user interface thread is unable to keep up

with the faster flow of processed information, the other threads now have to wait around for the new bottleneck in the system.

When you have a bug in multithreading code, it's really easy to blame the multiple threads and immediately start looking for data races and deadlocking. You should remember not to overlook the possibility of bugs in the single-threaded sequential code.

As I've already discussed, controlling code execution with multiple threads can be complex and is likely to result in hard-to-find software defects. To avoid these bugs by the use of good design, you need to understand them in some detail. The next section looks at typical bug types related to multithreading.

Multithreading Problems

Several bug types are associated with multithreading. The ones that you're most likely to meet are as follows:

- *Data races:* A data race occurs when multiple threads are allowed simultaneous access to read from and write to the same data area. This is likely to result in inconsistency or even corruption of that data. Using synchronization locks to serialize thread access to the common data area is the usual way of combating this problem.

- *Deadlock:* A process deadlock happens when two or more threads are unable to proceed because each is waiting for one of the others to proceed. The most common type of deadlock involves one thread issuing a synchronization lock on resource A and then trying to access resource B while another thread locks resource B and then tries to access resource A. The result is that the two threads are in a deadly embrace and each thread will wait forever for the opposing thread to relinquish its lock.

- *Livelock:* Process livelock occurs when two or more threads become caught in a circular loop. For example, thread A sends an error message to thread B, which responds by sending an error message back. This can result in a never-ending stream of error messages from one thread to the other.

- *Starvation:* Thread starvation happens when a thread grinds almost to a halt because of lack of processor time or a continuing failure to access some resource being used by other threads.

The next section looks at each of these problems in more detail and suggests ways of dealing with each problem.

Understanding Data Races

A data race happens when two or more threads race each other to read from and write to common data shared between the threads. The adverse effects of a data race happen when the reading and writing occur in a sequence not anticipated by the developer of the multithreaded code, and the common data then becomes corrupted.

The **ThreadSynch** console application demonstrates how a data race can occur. This program uses multiple worker threads to perform the very simple task of incrementing a shared counter from its starting value to a specified end value. Listing 14-1 shows the **Sub Main** of the application and the **CountCoordinator** class that launches the worker threads and coordinates the shared count.

Listing 14-1. The CountCoordinator Class

```
Option Strict On
Imports System.Threading

Module CountMonitor
    Sub Main()
        Dim CountTest As New CountCoordinator(5)
        Console.ReadLine()
    End Sub
End Module

Class CountCoordinator
    Private Const MAX_COUNT As Integer = 99
    Private m_Counter As Integer = 0

    Public Sub New(ByVal NumberOfCounters As Integer)
        Dim EachWorker As Integer, NewThread As Thread, _
            Worker As CountWorker
        'Show starting conditions
        Console.WriteLine( _
          "Count started at {0} with max value of {1}.", _
          CStr(Me.CurrentCount), CStr(Me.MaxCount))
        Console.WriteLine( _
          "{0} worker threads are doing the counting.", _
          CStr(NumberOfCounters))
        'Start specified number of worker threads
        For EachWorker = 1 To NumberOfCounters
            Worker = New CountWorker(Me, EachWorker)
            NewThread = New Thread(AddressOf _
                                   Worker.IncrementCount)
```

```
            NewThread.Start()
        Next EachWorker
    End Sub

    Public Property CurrentCount() As Integer
        Get
            Return m_Counter
        End Get
        Set(ByVal Value As Integer)
            m_Counter = Value
        End Set
    End Property

    Public ReadOnly Property MaxCount() As Integer
        Get
            Return MAX_COUNT
        End Get
    End Property

End Class
```

The application creates the **CountCoordinator** class and passes 5 as the number of worker threads required to the class constructor. The class constructor starts each of the specified number of threads, and these worker threads then compete for processor time to increment the shared counter from 0 to its maximum value, in this case 99. The worker threads are given access to this counter and its maximum value through the **CurrentCount** and **MaxCount** properties of the **CountCoordinator** class.

Listing 14-2 shows the **CountWorker** class, which represents each of the worker threads. This class stored the reference to the **CountCoordinator** class that it receives in its constructor and then uses this to increment the **CountCoordinator.CurrentCount** property.

Listing 14-2. The CountWorker Class

```
Class CountWorker
    Private m_Coordinator As CountCoordinator
    Private m_WorkerId As Integer

    Public Sub New(ByVal Coordinator As CountCoordinator, _
                   ByVal WorkerId As Integer)
        m_Coordinator = Coordinator
        m_WorkerId = WorkerId
    End Sub
```

```
    Public Sub IncrementCount()

        'Increment shared counter until equal to maximum allowed
        With m_Coordinator

            Do While .CurrentCount < .MaxCount
                Select Case .CurrentCount
                    Case Is < (.MaxCount - 10)
                        Thread.Sleep(0)
                        .CurrentCount += 10
                    Case Is < .MaxCount
                        Thread.Sleep(0)
                        .CurrentCount += 1
                    Case Else
                End Select
                'Show current thread and counter value
                Console.WriteLine( _
                   "Worker {0} current count {1}", _
                   CStr(m_WorkerId), CStr(.CurrentCount))
            Loop

        End With

    End Sub

End Class
```

The **IncrementCount** method takes one of three actions, depending on the current value of the shared counter. If the counter value is within 10 of the maximum, it adds 1 to the counter. If the counter value is not within 10 of the maximum, it adds 10 to the counter. If the counter value is equal to or greater than the maximum value, the thread simply finishes running, as no more counting is required.

Given this relatively simple code, it's hard to see what could go wrong. But if you run the **ThreadSynch** program, you can see that the counter is always incremented well beyond its maximum value before all of the worker threads cease working. Figure 14-2 shows an example where the counter starts at 0, has a maximum value of 99, and five worker threads have been allocated to perform the counting. The final counter value of 131 is much higher than you might expect from reading the code.

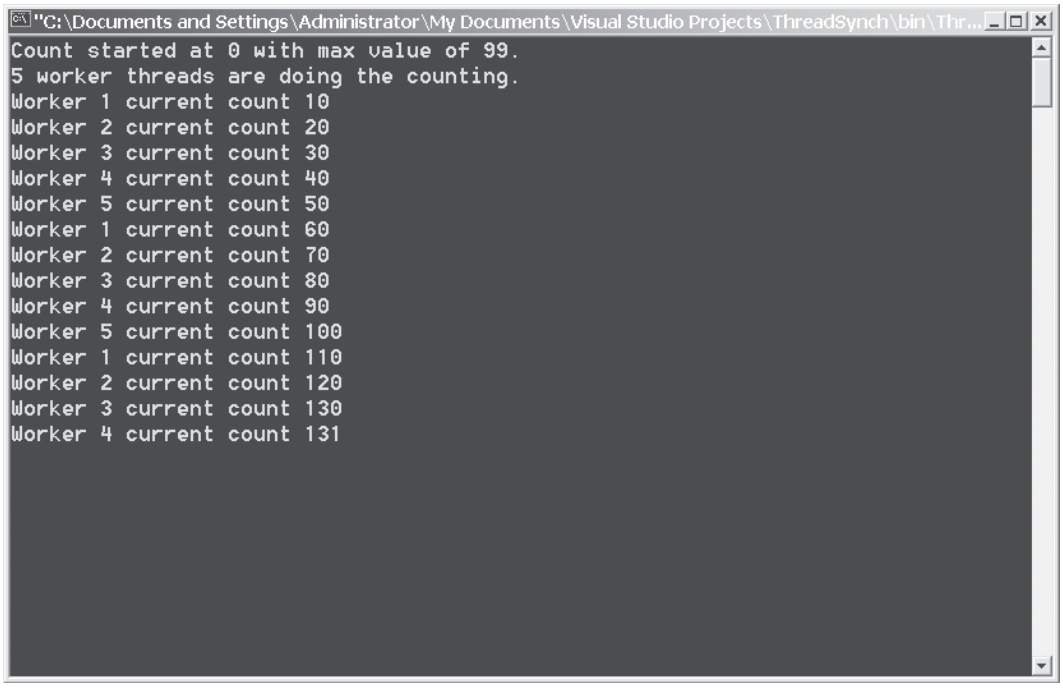

Figure 14-2. An example run of the ThreadSynch application

What's happening is that in between a thread reading the shared counter value and incrementing it, another thread has also incremented the value. The threads aren't all working with the same value of the counter, and therefore they race each other to read and increment this value.

The insidious evil is that in a real application, this final counter value could be almost any number equal to or greater than the maximum, depending on the number of allocated worker threads and when Windows decides to switch between each thread. In this example program, I deliberately made each thread give up its time slice by issuing a **Thread.Sleep(0)** instruction after reading the shared counter. This allows me to force a data race problem because sleeping one thread allows the processor to switch the execution of other threads. In the real world, where the **Thread.Sleep** instruction probably wouldn't be used in this manner, the code might work fine 99.99% of the time, and you're likely to see an inconsistent final number only when you're demonstrating the program to your boss or an important customer.

The most common way of solving a data race problem is to lock the data shared between multiple threads so that only one thread can access the shared data at any one time. One way of locking shared data is with the **SyncLock** statement. If you add the lines shown in bold in Listing 14-3 to the **CountWorker.IncrementCount** method, you'll find that the data race goes away completely.

Listing 14-3. Adding SyncLock to Synchronize Access to Shared Data

```
'Increment shared counter until equal to maximum allowed
With m_Coordinator

    Do While .CurrentCount < .MaxCount
        SyncLock(m_Coordinator)
            Select Case .CurrentCount
                Case Is < (.MaxCount - 10)
                    Thread.Sleep(0)
                    .CurrentCount += 10
                Case Is < .MaxCount
                    Thread.Sleep(0)
                    .CurrentCount += 1
                Case Else
            End Select
        End SyncLock

        'Show current thread and counter value
        Console.WriteLine("Worker {0} current count {1}", _
                    CStr(m_WorkerId), CStr(.CurrentCount))

    Loop

End With
```

In this example, **SyncLock** is used to lock access to the count coordinator object so that only one thread at a time can run code within this object for the duration of the lock. For finer synchronization, you can use the **Monitor** class. For high-performance addition or subtraction of a value type variable, you can use the **Interlocked** class. However, any locking strategy needs to beware of two potential problems. The first problem is that performance can be adversely affected if the locking is done too frequently or blocks too big a region of code from executing. The second problem is that locking exposes you to the classic deadlock situation, as discussed further in the next section.

Understanding Process Deadlock

A process deadlock happens when two or more threads are unable to proceed because each is waiting for one of the others to proceed. The most common type of deadlock occurs when one thread issues a synchronization lock on resource A and then tries to access resource B while another thread locks resource B and

then tries to access resource A. The result is that the two threads are in a deadly embrace and each thread will wait forever for the opposing thread to relinquish its lock. An alternative scenario involves a cyclic chain of dependencies where multiple threads become gridlocked because they're queuing up and waiting for other threads to relinquish one or more shared resources. This is similar to the way that road traffic can become gridlocked at a very busy intersection.

The **ThreadDeadlock** console application demonstrates how a process deadlock can occur. This program consists of a **Bank** object that spawns multiple **Cashier** threads to randomly debit and credit two **Account** objects owned by the **Bank** object. Listing 14-4 shows the **Sub Main** of the application and the **Bank** class that launches the cashier threads and keeps control over the two bank accounts.

Listing 14-4. The Bank Class

```
Option Strict On
Imports System.Threading

Module DeadlockTest

    Sub Main()
        Dim TransferTest As New Bank(2, 10000)
        Console.ReadLine()
    End Sub

End Module

Class Bank
    Private m_AccountOne As New Account(1000000)
    Private m_AccountTwo As New Account(1000000)

    Public Sub New(ByVal NumberOfCashiers As Integer, _
                   ByVal NumberOfTransfers As Integer)
        Dim EachWorker As Integer, NewThread As Thread, _
            Worker As Cashier

        'Show starting conditions
        Console.WriteLine( _
          "{0} cashiers are performing {1} transfers each.", _
          NumberOfCashiers.ToString, _
                      NumberOfTransfers.ToString)
```

```
        'Start specified number of worker threads
        For EachWorker = 1 To NumberOfCashiers
            Worker = New Cashier(Me, NumberOfTransfers)
            NewThread = New Thread(AddressOf Worker.TransferMoney)
            NewThread.Name = "Cashier" & EachWorker.ToString
            NewThread.Start()
        Next EachWorker

    End Sub

    Public ReadOnly Property AccountOne() As Account
        Get
            AccountOne = m_AccountOne
        End Get
    End Property

    Public ReadOnly Property AccountTwo() As Account
        Get
            AccountTwo = m_AccountTwo
        End Get
    End Property

End Class
```

The application creates the **Bank** class and passes 2 to the class constructor as the number of cashiers required. It also passes the number of account transfers to be made, in this case 10,000. This relatively large number is used to demonstrate that a process deadlock can happen infrequently and may require some extensive execution testing to find. The **Bank** class has two instances of the **Account** class, each given an opening account balance of £1 million. When the **Bank** object is created, it then instantiates the specified number of **Cashier** objects and starts a worker thread for each of the cashiers. It also passes a reference to itself to each of the cashiers so that they can transfer money to and from the bank accounts.

One interesting point is that each of the cashier threads is given a name when it's instantiated. This really helps with debugging because the thread name is shown in the IDE Threads window, which makes each thread easier to identify. I go into more detail about the Threads window shortly, when you start to run this application.

Listing 14-5 shows the associated **Cashier** class. Each cashier is instantiated with an instruction to perform a certain number of transfers on the bank's two accounts, in this case 10,000 transfers. For each transfer, the bank accounts to credit and debit are chosen randomly from the two available. Before performing

the transfer, the credit account is locked first followed by the debit account. This synchronization locking (shown in bold) prevents multiple cashiers from interfering with each other's transfers and causing a data race as seen in the previous example. Unlike the previous example, no thread sleeping is used. The sheer number of transfers happening is going to force a deadlock at some point, although thread sleeping (which allows other threads to run) will usually increase the speed at which a deadlock happens.

Listing 14-5. The Cashier Class

```
Class Cashier
    Private m_Bank As Bank, m_NumberOfTransfers As Integer

    Public Sub New(ByVal AnyBank As Bank, _
                   ByVal NumberOfTransfers As Integer)
        m_Bank = AnyBank
        m_NumberOfTransfers = NumberOfTransfers
    End Sub

    Public Sub TransferMoney()
        Dim CurrentTransfer As Integer

        With m_Bank

            For CurrentTransfer = 1 To m_NumberOfTransfers

                If TrueOrFalse() = True Then
                    SyncLock (.AccountOne)
                        SyncLock (.AccountTwo)
                            .AccountOne.CreditBalance(100)
                            .AccountTwo.DebitBalance(100)
                            Console.WriteLine( _
                                 "{0}: Transfer {1}", _
                                 Thread.CurrentThread.Name, _
                                 CurrentTransfer.ToString)
                        End SyncLock
                    End SyncLock
                Else
                    SyncLock (.AccountTwo)
                        SyncLock (.AccountOne)
                            .AccountOne.DebitBalance(100)
                            .AccountTwo.CreditBalance(100)
                            Console.WriteLine( _
```

```
                                    "{0}: Transfer {1}", _
                                    Thread.CurrentThread.Name, _
                                    CurrentTransfer.ToString)
                            End SyncLock
                        End SyncLock
                    End If
                Next CurrentTransfer

            End With

        End Sub

        Private Function TrueOrFalse() As Boolean
            Randomize()
            Dim Test As Single = (Int((2 * Rnd()) + 1))
            Return CBool(Test = 1)
        End Function

End Class
```

Finally, Listing 14-6 shows the **Account** class. This class offers functions for debiting and crediting the bank account, and a property for interrogating the current account balance.

Listing 14-6. The Account Class

```
Class Account
    Private m_AccountBalance As Decimal = 0

    Public Sub New(ByVal StartingBalance As Decimal)
        m_AccountBalance = StartingBalance
    End Sub

    Public ReadOnly Property AccountBalance() As Decimal
        Get
            AccountBalance = m_AccountBalance
        End Get
    End Property

    Public Function DebitBalance _
           (ByVal AmountToDebit As Decimal) As Decimal
        m_AccountBalance -= AmountToDebit
        Return m_AccountBalance
    End Function
```

```
    Public Function CreditBalance _
            (ByVal AmountToCredit As Decimal) As Decimal
        m_AccountBalance += AmountToCredit
        Return m_AccountBalance
    End Function

End Class
```

If you load the **ThreadDeadlock** solution into Visual Studio and run it using F5, you can see that it prints each transfer made by the two cashiers as it happens, along with the name of the cashier (thread) doing the transfer. At some unpredictable number of transfers, the application simply hangs, as shown in Figure 14-3. If you run the application many times, you should see that it hangs at a different point each time.

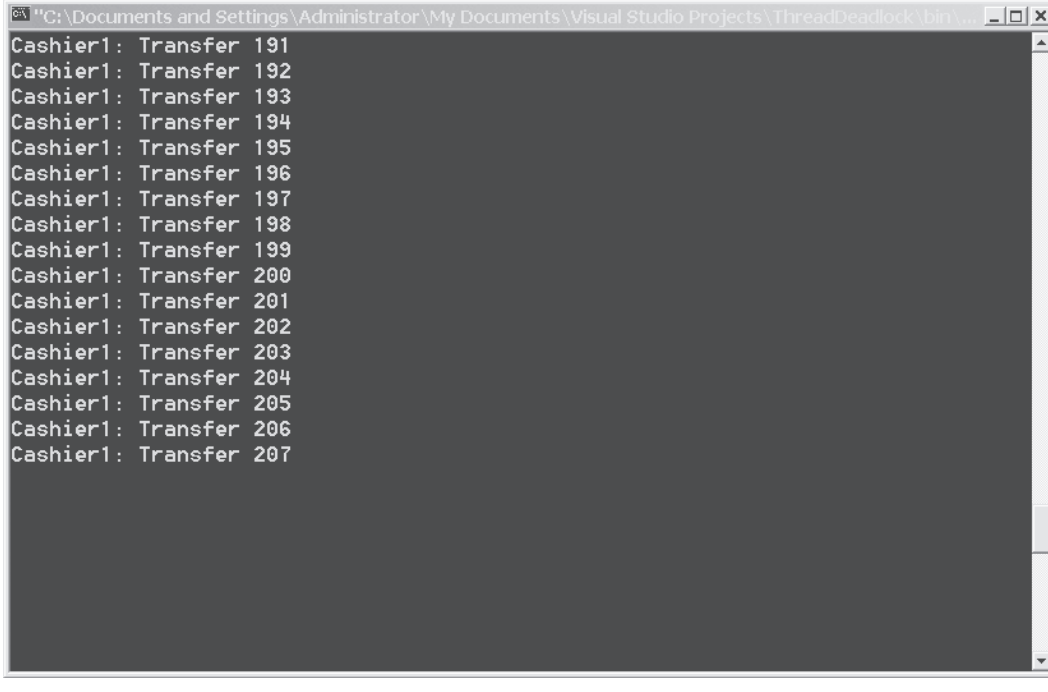

Figure 14-3. Process deadlock in the ThreadDeadlock application

To establish what's happening and why the program is hanging, press Ctrl+Break at the point where the application hangs. The debugger should break into the program at one of the four **SyncLock** statements. Select Debug ➢ Windows ➢ Threads to display the Threads window, and you can see the three managed threads within the program, as shown in Figure 14-4. The first thread is the main application thread, and the other two are the cashier threads.

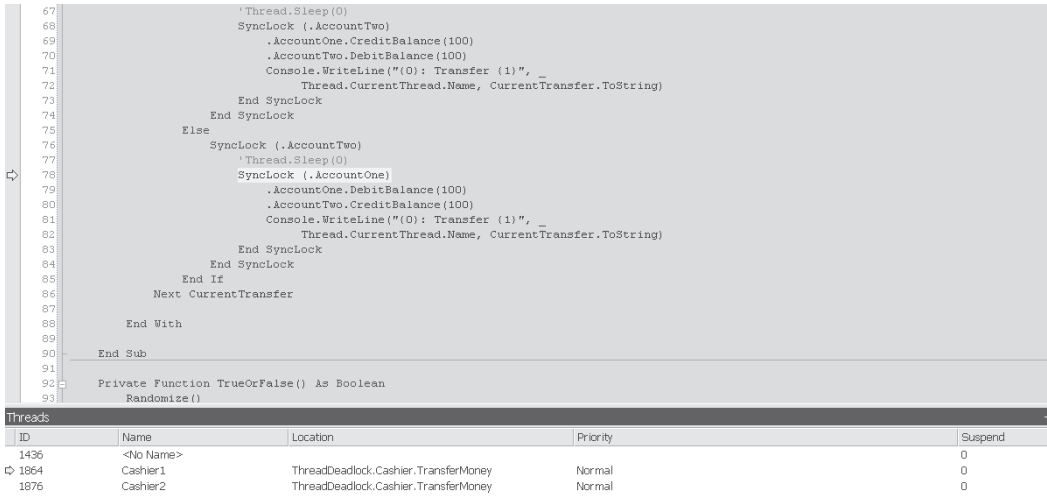

Figure 14-4. Using the Threads windows to investigate a process deadlock

The debugger should be paused on one of the **SyncLock** statements. The thread that was active when the program was suspended by the debugger is shown with a small yellow arrow next to it. If you right-click the nonactive cashier thread in the Threads window (the cashier thread without a yellow arrow next to it) and choose Switch to Thread from the context menu, you can see that the other thread is also paused at a **SyncLock** statement. If you switch between the two cashier threads in this manner, you can see that each of the threads is attempting to lock a different account. One cashier has a lock on the first account and is trying to lock the second account, and the other cashier has a lock on the second account and is trying to lock the first account. Hence the process is deadlocked and can't go any further, so the application hangs.

Notice that the program is randomly picking the order in which it locks the accounts. One way of avoiding this type of deadlock situation is always to lock your resources in exactly the same sequence within each thread. An identical sequence of synchronization locks ensures that multiple threads won't try to grab each other's resources in a nondeterministic order. This is usually easier said than done, because it's normally very hard to guarantee that code statements are executed in the same order in each thread. Depending upon thread entry, data, and timing conditions, each thread may follow a quite different execution path to its siblings. Even if you can guarantee that objects are locked in exactly the same order, you can still be caught by subtleties, as you'll see in a couple of paragraphs.

In this case, an alert developer would probably anticipate the deadlock because the synchronization is explicit, out in the open, and clearly happening in a random order. A more dangerous situation is when the synchronization locks that lead to a deadlock are implicit rather than explicit. In this example, the

synchronization lock might be placed inside a **Transfer** method as shown in Listing 14-7, and you might not have easy access to that method's source code. If your threads are calling another component's methods and those methods perform their own synchronization, you may not even be aware that this locking is taking place and that a potential deadlock is hovering over your program.

Listing 14-7. A Hypothetical Transfer Method

```
Public Sub Transfer(ByVal AccountToDebit As Account, _
                    ByVal AccountToCredit As Account, _
                    ByVal AmountToTransfer As Decimal)
    SyncLock (AccountToDebit)
        SyncLock (AccountToCredit)
                'Transfer happens here
```

And here's the subtlety that I talked about a couple of paragraphs ago. In Listing 14-7, the resources are always locked in the same order, so you might think that a deadlock can't occur. However, what happens when the first thread passes account A as the debit account and account B as the credit account, while the second thread does the opposite? If both threads enter this method simultaneously, and a context switch happens after the first lock, you might well see a deadlock.

To reduce the amount of time that your application is exposed to a potential deadlock, you should acquire your synchronization locks as late as possible and release them as early as possible. You should always try to avoid lengthy operations inside code that's locked, especially operations (such as I/O) that can block indefinitely.

If your code throws an exception in a region of code protected by a **SyncLock** statement, the synchronization lock will always be released. The VB .NET compiler automatically places any synchronized region of code inside an implicit **Try...Finally** block, where the **Finally** block releases the synchronization lock. This has one interesting side effect: You can't use **SyncLock** in a method that also uses unstructured exception handling (**On Error...**), because structured and unstructured exception handling can't be combined within the same method.

A final subtlety to remember is that threads can deadlock while waiting on events as well as while waiting to acquire resources locked by other threads. This is often overlooked, especially when a developer forgets that code in an event handler is executed by the thread that raises the event, not by the thread that owns the object within which the event handler resides.

Understanding Process Livelock

Process livelock is similar to process deadlock in its external appearance, in that both situations result in the process appearing to hang indefinitely. Internally, however, livelock is quite different from deadlock. A process is considered to be in a state of livelock when thread code is still executing, but two or more threads are in a never-ending cycle with each other and no useful work is being done. One example of this was mentioned earlier, a situation where one thread throws an error message, to which a second thread, not expecting this error, responds with an error message of its own. This can result in a continuous cycle of errors being thrown.

There's no easy way to prevent process livelock, although it's often possible to detect the livelock once it's happened. In the case of a livelock produced by a cycle of exception messages, Chapter 6 discusses performance counters that you can use from Performance Monitor (or from code) to detect an excessive number of exceptions being thrown within a certain time period. Detecting a livelock internally is much easier than detecting a deadlock because thread code can continue to execute even in the presence of a livelock. However, it requires some careful design to ensure that your detection code can adequately find and stop livelocks. The best solution is to try to anticipate potential livelock situations and then design your threads to prevent them or at least make them highly unlikely to happen.

Understanding Thread Starvation

To understand thread starvation, think about approaching a road tunnel in your car. The road tunnel has only a single lane, but it has to accommodate traffic traveling in both directions. The problem is that if the oncoming stream of cars is steady enough, you won't get a chance to go through the tunnel yourself. The analogy is that each car is a thread and the tunnel is a shared resource.

Just like your car can't get through the tunnel because of the oncoming cars, a thread can be starved of a resource by multiple other threads and be unable to execute in a timely fashion. The thread might not die of starvation—it just runs very slowly. This happens when the thread can get access to the resource, but only for a limited time before competing threads grab the resource back. Going back to the car/thread analogy, think of your car managing to enter the tunnel, but having to swerve into a turnout and stop every time it meets an oncoming car.

There are at least two common reasons for thread starvation. The first reason is when you assign a lower priority than normal to a specific thread. In this case, you've explicitly requested that the thread is less important than other threads with a normal priority, and you can solve the problem by juggling and tuning thread priorities as required. The second common reason for thread starvation is sometimes called the *writer-reader* problem, where a single writer thread tries to lock some data in order to update it but is repeatedly blocked by multiple reader threads with their own synchronization locks on the same data. This situation is common enough that the .NET Framework has constructs specifically designed to help you to avoid thread starvation when it occurs.

In the writer-reader situation, you need to make sure that a writer thread isn't prevented from doing its writing for lengthy periods of time by multiple competing reader threads. To do this, you can use the Framework's **ReaderWriterLock** synchronization class. This class enforces exclusive access to a region of code for any writer thread, but it allows nonexclusive access to a code region for any reader thread. The class also coordinates thread access so that once a writer thread has requested a lock, all subsequent lock requests by reader threads are queued until the writer lock has been granted. This prevents starvation of a writer thread through any inability to grab a shared resource away from multiple reader threads.

In effect, the **ReaderWriterLock** class switches between one writer thread and a group of reader threads. In any situation where the shared resource is being updated infrequently, this class has been designed to provide better throughput than a standard one-at-a-time lock such as **SyncLock** or **Monitor**.

Listing 14-8 demonstrates this by showing a class designed for reading and writing of some hypothetical shared data. The class is completely thread-safe in that multiple reader and writer threads can use it simultaneously. Its use of the **ReaderWriterLock** class also prevents starvation of any writer threads, even in the presence of a larger number of competing reader threads. It's worth examining the code closely because, although it looks simple, it has some subtleties that may not be immediately apparent from a first reading.

Listing 14-8. A Thread-Safe Class for Reading and Writing Shared Data

```
Option Strict On
Imports System.Threading

Class ReadWrite
    'This class is thread-safe in that its methods can
    'be called safely from multiple threads simultaneously.
    Private m_Lock As New Threading.ReaderWriterLock()
```

```
    Public Sub ReadData(ByVal MillisecondsToWait As Integer)
        'This procedure reads information from some source.
        'The read lock prevents data from being written until
        'the thread is done reading, while allowing other threads
        'to call ReadData.
        m_Lock.AcquireReaderLock(MillisecondsToWait)

        'We have a lock, so now try to read the data
        Try
            'Perform read operation here.
        Finally
            m_Lock.ReleaseReaderLock()
        End Try

    End Sub

    Public Sub WriteData(ByVal MillisecondsToWait As Integer)
        'This procedure writes information to some source.
        'The write lock prevents data from being read or
        'written until the thread has finished writing.
        m_Lock.AcquireWriterLock(MillisecondsToWait)

        'We have a lock, so now try to write the data
        Try
            'Perform write/update operation here.
        Finally
            m_Lock.ReleaseWriterLock()
        End Try

    End Sub

End Class
```

Acquiring a Reader Lock

When a thread enters the **ReadData** method shown in Listing 14-8, it attempts to acquire a reader lock using the **AcquireReaderLock** method of the **ReaderWriterLock** class. The **AcquireReaderLock** will block if a different thread has a writer lock or if any thread is waiting to acquire a writer lock. This latter point is one of the keys to avoiding thread starvation of a writer thread. Any block on the attempt to acquire a reader lock will last until either the reader lock is granted or the number of milliseconds specified by the **MillisecondsToWait**

parameter of the **ReadData** method has expired. If the time-out of the reader lock attempt does expire, the **AcquireReaderLock** statement will throw an **ApplicationException** exception that can be caught by the code calling the **ReadData** method. Using a time-out in this manner prevents any possible deadlock problems.

If the reader lock is granted, the **ReadData** method then attempts to read the data. This read attempt is placed within a **Try...End Try** block so that read lock will always be released, even if an error occurs. This is important because the number of reader locks acquired and reader locks released should always be matched.

There is one final subtlety of the **AcquireReaderLock** method of which you should be aware. If the current thread already has a writer lock and it then attempts to acquire a reader lock, the reader lock isn't granted. Instead, the writer lock count is incremented by one and the data read then proceeds as normal. This nuance has the advantage of preventing a thread from blocking on its own writer lock. The disadvantage is that instead of calling **ReleaseReaderLock** when the data read has completed, you need to call **ReleaseWriterLock** instead. This is because every writer lock has to be matched with a writer lock release, even if the writer lock has been acquired by using the **AcquireReaderLock** method. Fortunately, you can check if the current thread already has a writer lock by checking the **IsWriterLockHeld** property and then releasing the writer lock if necessary.

Acquiring a Writer Lock

When a thread enters the **WriteData** method shown in Listing 14-8, it attempts to acquire a writer lock using the **AcquireWriterLock** method of the **ReaderWriterLock** class. This method blocks if a different thread has a reader or writer lock. If the writer lock attempt is blocked, it's placed in a queue ahead of any reader locks that are blocked. This is the final key to avoiding thread starvation of a writer thread. Once again, a time-out period is specified on the attempt to acquire the writer lock to avoid any possibility of deadlocks. If the time-out expires, the **AcquireWriterLock** statement will throw an **ApplicationException** exception that can be caught by the code calling the **WriteData** method.

If the writer lock is granted, the **WriteData** method then tries to update the data. As with the **ReadData** method, this update is placed within a **Try...End Try** block so that writer lock will always be released, even if an error occurs. This is important because the number of writer locks acquired and writer locks released should always be matched.

As with the **AcquireReaderLock** method, **AcquireWriterLock** also has a final subtlety that can trip you up. If a thread calls **AcquireWriterLock** while it also has a reader lock, it will block on that reader lock. If an infinite time-out is specified for the writer lock attempt, the thread will then deadlock with itself. To prevent

this, you can use the **IsReaderLockHeld** property and then upgrade the reader lock to a writer lock using the **UpgradeToWriterLock** method if necessary. Remember, of course, that if you do upgrade a reader lock to a writer lock, you need to release the writer lock rather than the reader lock.

The ThreadMonitor Application

This section uses the **ThreadMonitor** application to investigate managed and unmanaged threads running within a process. If you load the **ThreadMonitor** solution into Visual Studio and execute it by pressing F5, you can see the application's user interface as shown in Figure 14-5.

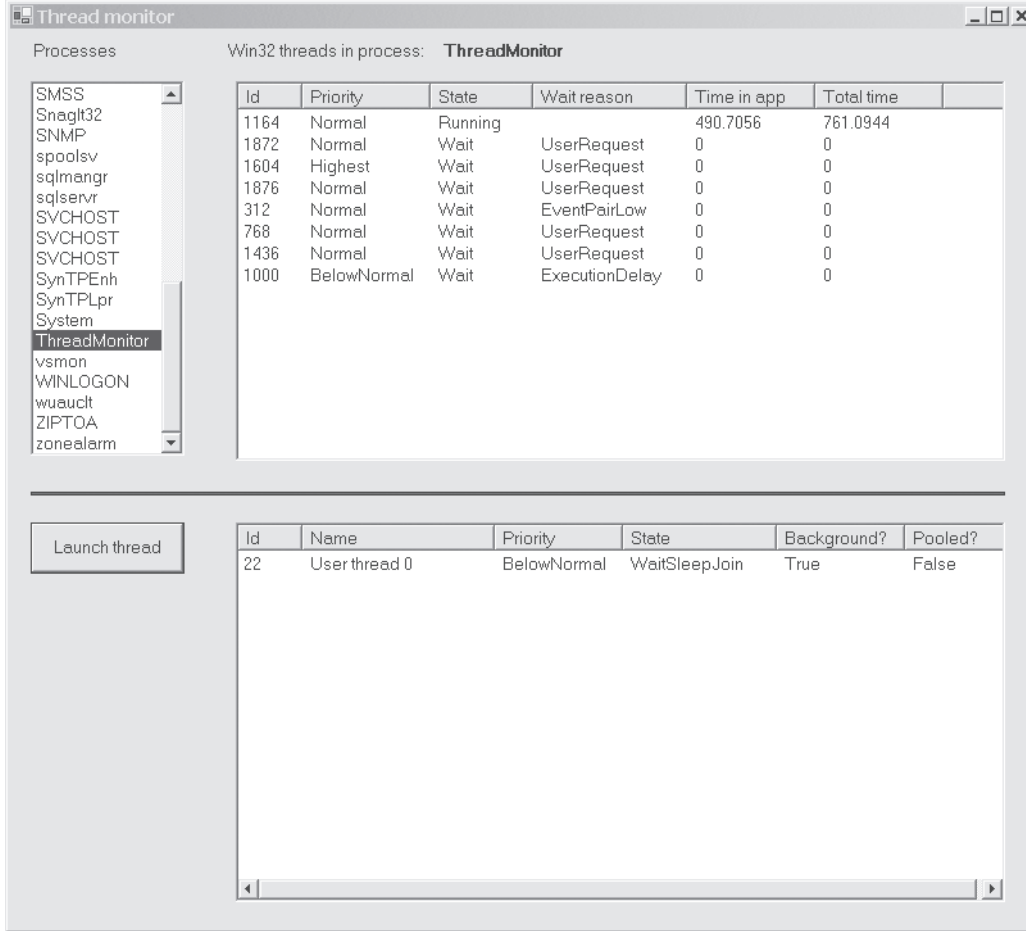

Figure 14-5. The user interface of the ThreadMonitor application

The list box on the left side of the form contains a list of the processes that are currently executing on the local machine. When you click any process, its managed and unmanaged threads are shown in the form's top list view. This list of threads is updated every second and shows details about each thread's priority, state, and processor time.

Clicking the command button underneath the list box allows you to launch a user thread. Each thread sleeps for 5 seconds and then executes a simple loop 30 million times before terminating. You can track the user threads because they're shown in the form's bottom list box as well as in the top list box.

Listing 14-9 shows the code that uses the **Process.Threads** collection to enumerate all of the Win32 threads within the selected process and display each of them in the top list box. Unfortunately, there's no direct way of knowing which of these process threads is managed and which isn't. There isn't even a guarantee that a single managed thread actually maps to a single Win32 thread, because the CLR explicitly declines to make any such guarantee. In the future, the use of thread fibers may allow multiple managed threads to run on a single Win32 thread.

Listing 14-9. Displaying All Win32 Threads Within a Specified Process

```
Private Sub UpdateWin32ThreadDisplay( _
            ByVal SelectedProcessId As Integer)
    Dim SelectedProcess As Process
    Dim ThisThread As ProcessThread, LV_item As ListViewItem

    'Get Win32 threads for this process and display them
    SelectedProcess = _
       Process.GetProcessById(SelectedProcessId)
    LabelThreadName.Text = SelectedProcess.ProcessName

    With Me.ThreadList
        .BeginUpdate()
        .Items.Clear()

        'Iterate through every Win32 thread in this process
        For Each ThisThread In SelectedProcess.Threads

            Try
                'Add thread id
                LV_item = _
                  New ListViewItem(ThisThread.Id.ToString)
                'Add thread details
                With LV_item.SubItems
```

```
                        'Thread priority
                        .Add(ThisThread.PriorityLevel.ToString)
                        'Thread state
                        .Add(ThisThread.ThreadState.ToString)
                        'Reason for thread wait
                        If ThisThread.ThreadState = _
                          Diagnostics.ThreadState.Wait Then
                            .Add(ThisThread.WaitReason.ToString)
                        Else
                            .Add(vbNullString)
                        End If
                        'Thread time in app
                        .Add _
(ThisThread.UserProcessorTime.TotalMilliseconds.ToString)
                        'Thread time in OS
                        .Add _
(ThisThread.TotalProcessorTime.TotalMilliseconds.ToString)
                    End With
                    'Display the thread
                    .Items.Add(LV_item)

                Catch Exc As InvalidOperationException
                    'Thread's disappeared - ignore

                End Try

            Next ThisThread

            .EndUpdate()
        End With

    End Sub
```

If you choose the **ThreadMonitor** process in the list box, you can see that it contains no less than eight Win32 threads. If, however, you use Ctrl+Break to break into the program and then examine the Threads window, you'll see only two threads displayed. Make a note of the thread ID of each of the two threads shown, and then resume program execution with F5. You can use the thread IDs to locate these two managed threads in the top list box. The first thread is usually the application's main thread and will normally be the first thread in the **Process.Threads** collection, although this is also not guaranteed. The second thread shown in the Threads window is the thread that runs the message pump for the Windows Form that's being displayed.

Listing 14-10 shows the code that uses the collection of managed user threads to display every user thread in the bottom list box. This collection is maintained by the code every time a new user thread is launched or terminates. Keeping this collection solves the problem of trying to figure out which Win32 thread corresponds to which user thread. The reason for keeping track of these threads is that a managed thread object has some useful information about the thread that the standard process thread object doesn't have. Some of this extra information is displayed in the list box, including the thread name, whether it's a foreground or background thread, and whether the thread is running in the managed thread pool.

Listing 14-10. Displaying All User Threads Within a Specified Process

```
Private Sub UpdateUserThreadDisplay()
    Dim ThisThread As Threading.Thread, _
        LV_item As ListViewItem

    'Iterate through managed threads for current process
    With Me.ManagedThreadList
        .BeginUpdate()
        .Items.Clear()

        'Iterate through every thread in this process
        For Each ThisThread In UserThreads

            If ThisThread.IsAlive Then

                Try
                    'Add thread id
                    LV_item = New ListViewItem _
                        (ThisThread.GetHashCode.ToString)
                    'Add thread details
                    With LV_item.SubItems
                        'Add thread name
                        .Add(ThisThread.Name)
                        'Thread priority
                        .Add(ThisThread.Priority.ToString)
                        'Thread state
                        .Add(ThisThread.ThreadState.ToString)
                        'Thread is alive?
                        .Add(ThisThread.IsAlive.ToString)
                        'Background thread?
                        .Add(ThisThread.IsBackground.ToString)
```

```
                    'Threadpool thread?
                    .Add _
                  (ThisThread.IsThreadPoolThread.ToString)
                End With
                'Display the thread
                .Items.Add(LV_item)

            Catch Exc As Threading.ThreadStateException
                'Thread's disappeared - ignore
                UserThreads.Remove _
                     (ThisThread.GetHashCode.ToString)

            End Try

        Else

            'Thread's dead - remove from collection
            UserThreads.Remove _
                 (ThisThread.GetHashCode.ToString)

        End If

    Next ThisThread

    .EndUpdate()
End With

End Sub
```

To launch and watch a user thread, click the "Launch thread" button once. A single managed thread will appear in the bottom list box, with its state set to **WaitSleepJoin**. After 5 seconds, the state will move to a state of **Running**, and then a few seconds later the thread will terminate and disappear from the display.

If you launch three threads in quick succession and then quickly press Ctrl+Break to pause the program, you can use the Threads window to examine these user threads. The thread name allocated by the code to each thread as it's launched is shown in the second column of the window. The active thread is shown with a yellow arrow next to it. To select another thread as the active thread, simply double-click it.

One interesting facility that the Threads window gives you is the ability to "freeze" or "thaw" a thread. To freeze a thread, right-click it in the Threads window and select the Freeze menu item. This prevents execution of that thread after

you resume the program, which can be very useful if you want to examine the behavior of a single thread without worrying about side effects caused by other threads. To thaw a thread, right-click the thread again and select the Thaw menu item. This allows execution of that thread once the program has been resumed. Two blue bars next to a thread in the Threads window means that the thread has been frozen. Of course, this thread freezing and thawing is just a debugger artifact, and it doesn't mean anything to Windows itself. If you took away the debugger from a frozen thread, it would continue execution normally.

Multithreading in Windows Forms

The **ThreadGui** application examines some of the threading issues that you have to face in a Windows Forms program. Multithreading can be very useful in this environment because it lets you keep the user interface responsive to the end user while you're doing intensive work, and you can also use it to let the end user cancel a long-running task. The drawback is that you need to respect the single-threaded nature of a Windows Form.

If you load the **ThreadGui** solution into Visual Studio and execute it by pressing F5, you'll see the application's user interface, as shown in Figure 14-6.

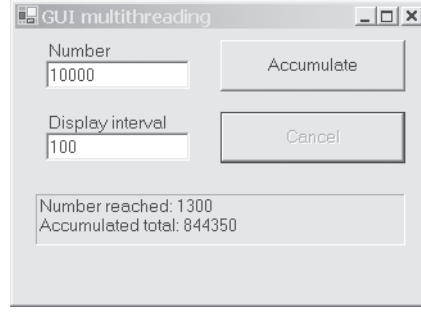

Figure 14-6. The user interface of the ThreadGui application

When you enter a number into the top text box and click the button marked Accumulate, the program accumulates a running total by adding every number together between 1 and the number entered in the text box. So if you entered **5** into the top text box, the accumulated total would be 15 (1 + 2 + 3 + 4 + 5). The Cancel button allows you to interrupt and stop the accumulation calculation at any time. The label at the bottom of the form displays the results of the calculation.

The second text box is used to allow the user to monitor progress of the calculation. After the accumulation is performed the number of times specified in this text box, the current running total is displayed in the label at the bottom of

the form and the display is paused for 0.1 second so that the user can glimpse this intermediate result. Then the calculation continues accumulating until either the next display interval is reached or the calculation finishes.

The challenge is to run the potentially lengthy calculation while keeping the user interface responsive so that the user can cancel the calculation and while displaying the intermediate results of the calculation in a safe manner.

The first major problem to overcome is that you should never, ever, update a control (in this case, the label at the bottom of the form) from a thread other than the thread that created the control. If you break this Windows Forms law, your program will experience strange and difficult multithreading bugs, and you won't be able to fix these bugs except by changing your program to obey the law. The first approach that many developers experiment with when faced with this prohibition is to tell the calculation thread to raise an event that the form can handle and use to update the user interface. Unfortunately, as mentioned earlier in this chapter, this won't work. An event handler always runs on the same thread that raised the event, so the event handler also isn't allowed to update the user interface.

The second problem is for the user interface thread to find a way of telling the calculation thread that the user has canceled the calculation request. Often a developer will think about setting a class-level variable that can be accessed and shared by both threads. This is, however, difficult to do without running into the thread synchronization issues that I discussed in the previous section.

Listing 14-11 shows the code that launches the worker thread to do the calculation once the user has clicked the Accumulate button. It uses an asynchronous delegate to spawn a work request that will be handled by the .NET thread pool. The delegate's **BeginInvoke** method is used to start the calculation thread asynchronously and pass it the specified number to accumulate.

Listing 14-11. Code to Launch the Calculation Thread

```
Option Strict On
Imports System.Threading

Public Class MainForm : Inherits System.Windows.Forms.Form

    Private Delegate Sub CalcDelegate(ByVal AnyNumber As Int32)
    Private Delegate Sub _
                ProgressDelegate(ByVal CurrentTotal As Decimal, _
                                 ByVal NumberReached As Int32, _
                                 ByRef CancelRequest As Boolean)
    Private m_CancelRequested As Boolean = False
```

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

        'Init calculation
        Me.cmdCalc.Enabled = False
        Me.cmdCancel.Enabled = True
        m_CancelRequested = False

        'Use asynch delegate to launch thread from thread pool
        Dim CalcAccumulation As CalcDelegate = New _
                    CalcDelegate(AddressOf CalculateAccumulation)
        CalcAccumulation.BeginInvoke( _
                    Convert.ToInt32(Me.txtNumber.Text), _
                    AddressOf CalcComplete, Nothing)

    End Sub
```

If you don't need the control that a manual thread gives you, such as setting the thread name or priority, using the thread pool spares you from the messy details of thread management and scales better in many multithreaded environments. Even better, as you'll see shortly, it's easy to propagate background thread exceptions back to the main thread when using the thread pool.

Listing 14-12 shows the method that runs the calculation thread. First place a breakpoint on line 162 (the line marked in bold in Listing 14-12) and then run the application by pressing F5. When you click the Accumulate button, the program will break as soon as it reaches your breakpoint. If you now look at the Threads window, you'll see two threads: the user interface thread and the calculation thread from the thread pool.

Listing 14-12. Performing the Accumulation Calculation

```
Private Sub CalculateAccumulation( _
            ByVal NumberToAccumulate As Int32)
    Dim CalcObject As New Calc(NumberToAccumulate), _
        CurrentTotal As Decimal = 0
    Dim CancelRequested As Boolean = False

    With CalcObject

        Do While .NumberReached <= NumberToAccumulate
            CurrentTotal = _
               .Accumulate(Convert.ToInt32(Me.txtInterval.Text))
            ShowProgress(CurrentTotal, .NumberReached, _
```

```
                        CancelRequested)
            If CancelRequested = True Then
                Exit Do
            End If
        Loop

    End With

End Sub
```

After performing each stage of the accumulation, the calculation thread calls the **ShowProgress** method, which is shown in Listing 14-13. This method is where the clever work happens. Remember that you should never update a user interface control from any thread except the one that created the control. To verify whether the user interface thread or the calculation thread is trying to update the user interface, the **ShowProgress** method checks **Me.InvokeRequired**. This will return **True** if the current thread isn't the user interface thread, and it will return **False** if it's the user interface thread. If **InvokeRequired** is **False**, then the thread is allowed to update the user interface directly, and therefore update the label with information about the progress of the calculation.

The interesting work happens when **InvokeRequired** is **True**, and therefore the user interface can't be updated directly. Every control has an **Invoke** method, and this is one control method that the CLR guarantees is safe to call from any thread. The arguments for the **Invoke** method include a delegate and a developer-defined set of arguments that are used to call the delegate's associated method. So this code calls **ShowProgress** recursively using the **ProgressDelegate** delegate. Using the **Invoke** method ensures that the recursive call happens on the user interface thread, where it's safe to update the user interface.

Listing 14-13. Updating the User Interface with Intermediate Calculation Results

```
Private Sub ShowProgress(ByVal CurrentAccumulation As Decimal, _
                         ByVal NumberReached As Int32, _
                         ByRef CancelRequest As Boolean)

    If Me.InvokeRequired = True Then

        'Transfer to GUI thread to show progress
        Dim CancelRequested As Object = False
        Dim SP As ProgressDelegate = _
            New ProgressDelegate(AddressOf ShowProgress)
        Dim Arguments() As Object = New Object() _
                {CurrentAccumulation, _
```

```
                    NumberReached, _
                    CancelRequested}
            Me.Invoke(SP, Arguments)
            CancelRequest = DirectCast(CancelRequested, Boolean)

        Else

            'We're on the GUI thread, so just show progress
            With Me.lblResult
                .Text = "Number reached: " & NumberReached.ToString
                .Text += Environment.NewLine
                .Text += "Accumulated total: " _
                      & CurrentAccumulation.ToString
            End With

            'Pause for a short time to allow user to read display
            Thread.CurrentThread.Sleep(100)

            'Return any cancellation request
            CancelRequest = m_CancelRequested

        End If

    End Sub
```

This technique for updating the user interface with progress information from the calculation thread works well and avoids any updating of the user interface from a nonuser interface thread. Instead of interacting directly, the user interface and calculation threads pass messages to each other, which means that you never have to worry about any thread synchronization or deadlock issues.

So the first problem of updating the user interface is solved, but you still need to tackle the problem of canceling the thread if the user clicks the Cancel button. The code behind the Cancel button is shown in Listing 14-14. It sets a class-level variable signifying that the user has issued a cancellation request. But how can you give the calculation thread access to this variable without running into thread synchronization issues?

Listing 14-14. Canceling the Calculation

```
Private Sub ButtonCancel_Click(ByVal sender As System.Object, _
                               ByVal e As System.EventArgs) _
                               Handles cmdCancel.Click
    'Request that calculation thread cancels itself
    m_CancelRequested = True
End Sub
```

The key to this puzzle lies in the **ShowProgress** method shown in Listing 14-13. This method has a **ByRef** argument called **CancelRequest**. When this method is called on the user interface thread, it sets this argument to the class-level request cancellation variable. Because the calculation thread regularly calls the **ShowProgress** method to update the user interface with its progress, it can read the cancellation request argument after it's called the **Invoke** method. This allows the request for cancellation to pass safely from the user interface thread to the calculation thread without having to worry about synchronization issues. Yet again, a message passing between the two threads prevents any problems that might arise if the two threads interacted directly.

Finally, once the calculation thread has completed, it invokes the callback that it was passed when it was started. This is the **CalcComplete** method shown in Listing 14-15, which simply resets the user interface so that another request can be started.

Listing 14-15. Completing the Calculation

```
Private Sub CalcComplete(ByVal CalcResult As System.IAsyncResult)
        'Called when asynch thread completes
        Me.cmdCalc.Enabled = True
        Me.cmdCancel.Enabled = False
End Sub
```

Dealing with Thread Failure

So far in this chapter, I've made the assumption that threads don't throw exceptions. Back in the real world, you need to be able to trap and deal with errors in threads launched by your applications.

Handling Thread Exceptions

An exception thrown by any thread that your application launches from its main thread is not propagated back to the main thread. The CLR simply swallows the exception and either returns the thread to the thread pool (if it came from thread pool) or just terminates the thread.

You can trap these thread exceptions by creating an unhandled exception filter and attaching this filter to the **Application.ThreadException** event. This process is described in detail in Chapter 13. An alternative possibility for threads that run in the thread pool is to add a **Try...End Try** block on the thread start delegate's **EndInvoke** method. **EndInvoke** on a thread delegate is the method used to block and wait for the thread to finish.

In the case of the **ThreadGui** application, the calculation thread is launched asynchronously, so it doesn't make sense to use **EndInvoke** after the thread has been launched. This would just force the user interface thread to block and wait for the calculation thread to finish, which defeats the point of using an asynchronous thread. Instead, you can call the **EndInvoke** method after the calculation thread has signaled its completion by calling the **CalcComplete** callback. Listing 14-16 shows you how you can modify the **CalcComplete** method shown in Listing 14-15 to catch any exception thrown by the calculation thread.

Listing 14-16. Trapping Any Calculation Thread Exception

```
Private Sub CalcComplete(ByVal CalcResult As System.IAsyncResult)
    Dim Result As AsyncResult = CType(CalcResult, AsyncResult)
    Dim MyDelegate As CalcDelegate = _
        CType(Result.AsyncDelegate, CalcDelegate)

    'Called when asynch thread completes
    Me.cmdCalc.Enabled = True
    Me.cmdCancel.Enabled = False

    Try
        'Find out if anything dodgy happened in the async thread
        MyDelegate.EndInvoke(CalcResult)
    Catch Exc As Exception
        MsgBox(Exc.Message, MsgBoxStyle.OKOnly, _
               "Async thread exception")
    End Try

End Sub
```

The first two lines in Listing 14-16 are a little confusing. Their job is to extract the original delegate from the asynchronous result returned by the calculation thread. Once the original delegate has been extracted, the line shown in bold calls **EndInvoke** on the original delegate. This has the effect of marshalling any exception that occurred in the calculation thread back to the user interface thread, where the **Catch** block shown here traps and displays the exception message.

If you throw a test exception from the **CalculateAccumulation** method shown in Listing 14-12, you should now see the displayed exception message. To throw a test exception, simply add a line such as

```
Throw ApplicationException("Test exception")
```

Terminating a Managed Thread

Explicitly terminating a managed thread should be done with some care. You can use the **Thread.Abort** method to terminate a thread, but when doing this you should be aware of exactly how the thread is terminated and the issues that this might cause. This section discusses these issues.

Using **Thread.Abort** doesn't end a thread immediately. It causes an exception of type **ThreadAbortException** to be generated in the thread to be aborted, which in turn unwinds any **Try...End Try** blocks in that thread's call stack. Code in related **Catch** and **Finally** blocks will be executed, and theoretically this code might perform long, or even infinite, calculations. This means that you can't guarantee that a thread will end when you call **Thread.Abort**.

Unlike a normal exception, an exception of type **ThreadAbortException** can't be suppressed by using a **Catch** block because it's always rethrown automatically at the end of each **Catch** block. However, a thread with sufficient privilege can call **Thread.AbortReset** to suppress this exception. This is another way in which a thread might resist being terminated.

To confirm that a thread really has terminated, you need to call **Thread.Join**. This joins your invoking thread to the thread that's been aborted and blocks your thread until either the joined thread has actually been aborted or the time-out that you specify in the **Thread.Join** has been exceeded.

If the **ThreadAbortException** caused by the call to **Thread.Abort** interrupts a thread during execution of a **Finally** block, that execution of that block of code won't be completed. This is one of the very few ways in which a **Finally** block can be bypassed.

Developers will often use **Try...Catch...Finally** to protect code where they anticipate that an exception might be thrown. The problem is that an exception of type **ThreadAbortException** can occur at any time and with no warning. This can complicate the process of writing really safe code that always unwinds itself when interrupted.

Aborting a thread with **Thread.Abort** unlocks any synchronization locks that the thread holds. This means that the data being protected by these locks may become inconsistent or corrupted, as discussed previously in the section on data race problems.

In some circumstances, attempting to abort a managed thread that's suspended by user code (as opposed to one suspended by the garbage collector or another system process) leads to that thread hanging forever and not terminating. This appears to be a known CLR bug at the time of this writing.

A better way of terminating a thread is to set a **PleaseStop** variable that the thread can check periodically. This allows a thread to terminate itself under controlled conditions. An example of using this technique safely without synchronization problems is discussed in the **ThreadGui** application earlier in this chapter.

The final fact to be aware of is that background threads are always terminated automatically when the process or thread from which they were launched is terminated. **Try...End Try** blocks are unwound normally when this happens.

Summary

Debugging a multithreaded application can be very messy and difficult. One developer memorably compared the difficulty of testing and debugging free-threaded code with performing a tonsillectomy while entering the patient from the wrong end. Understanding the common problems that can afflict a multithreaded application is key to creating a program design that avoids the need to perform heavy debugging. The examples of problem behavior discussed in this chapter are best avoided by designing your thread interactions very carefully. The **ThreadGui** example application presents one design pattern that avoids thread interaction problems.

INTERLUDE

THE 500-MILE E-MAIL BUG

In November 2002, a system administrator named Trey Harris posted the tale of a remarkable bug that he had diagnosed and fixed. He later posted a clarification of the bug's details and a FAQ about the bug, both of which you can find at `http://www.ibiblio.org/harris/500milemail.html`.

While working as a system administrator for a campus e-mail system at the University of North Carolina, Trey received a phone call from the chairman of the statistics department saying that nobody in the statistics department could send e-mail farther than 520 miles from the campus! After verifying that the call wasn't a practical joke, Trey ran some tests of his own on the e-mail system. Sure enough, when he sent test e-mails to Richmond, Virginia; Atlanta, Georgia; Washington D.C.; Princeton, New Jersey; and New York City, all of which are destinations within 520 miles, the e-mails were sent successfully. When he sent a test e-mail to Memphis, Tennessee, about 600 miles away, it failed to deliver. Likewise for Boston, Massachusetts; Detroit, Michigan; and Providence, Rhode Island, the latter being 580 miles away. A minor comfort was that when he sent an e-mail to a friend in North Carolina whose ISP was in Seattle, Washington, it also failed. If the problem had been related to the geographic location of the e-mail recipient rather than the mail server, there would have been some real explaining to do.

Having duplicated the problem, he now had to figure out what was causing it. After all, it's not every day that you find such an unusual bug. He knew that a consultant had recently patched the mail server to upgrade its SunOS operating system, but the consultant hadn't touched the mail system itself. The first obvious place to look for problems was the configuration file for the sendmail utility. But the sendmail.cf file on the offending mail server looked perfectly normal.

To investigate further, Trey telnetted into the SMTP port on the mail server, and was greeted with a SunOS sendmail banner…for sendmail version 5. At this point in time, Sun shipped the tried and trusted sendmail version 5 with its operating system, even though the version of sendmail used by the university had been standardized at version 8. So when the consultant patched the OS, the sendmail utility had been downgraded from version 8 to version 5, but the sendmail.cf configuration file had not been downgraded—this was the first clue.

Although the sendmail version 5 shipped by Sun had been tweaked to cope with a version 8 configuration file, it only did so by ignoring the configuration options that it couldn't understand. One of these options was the time-out to connect to the remote SMTP server, which sendmail set to zero because it couldn't understand the version 8 setting. Some experimentation revealed that under the typical load experienced by this particular mail server, and after accounting for router delays and transmission speeds across optic fiber, a zero time-out would abort a connect call to a remote mail server in approximately 3 milliseconds.

How far does light travel in 3 milliseconds? Slightly over 500 miles.