Taking a Dip in the Thread Pool

At 58 pages, the chapter on multithreading is possibly the longest chapter in the book. Yet now, looking back, I feel that I still have not done enough to demonstrate the principals of good multithreaded programming. In this bonus article, available only to readers of the book and Pinnacle's VB Developer newsletter, I'd like to walk you through the design of a simple, yet typical multithreaded application.
This example assumes that your main application or component has a set of clearly defined tasks that it needs to perform, and that the time to accomplish those tasks can range from very short to rather long. Examples of tasks that fall into this category range from database lookups, to retrieving information from a remote site or web site, to performing some operation on a legacy mainframe computer. Let's assume that these tasks are all synchronous – in other words, you start the task using a function or method call that does not return until the task is complete.

We'll model this task using the following class named Server defined in the server.vb file in the MTPool1 and MTPool2 sample applications:

```
Imports System.Threading
Public Class Server
    Public Sub ServerOp(ByVal Duration As Integer)
        Thread.Sleep(Duration)
    End Sub
End Class
```

The Sleep operation suspends the current thread, and represents a good simulation of a long synchronous operation – especially one that is not particularly CPU intensive.

Thread Pools – First Attempt

The MTPool1 sample application illustrates how you can use a thread pool to provide an efficient solution to performing these tasks in the background. Creating a separate thread each time a task executes can severely impact system performance if you have a great many tasks executing, as each thread requires time and system resources to create and tear down. A thread pool allows you to create a pool of threads that are assigned to tasks as needed.
The .NET Framework includes a thread pool class, but in many cases it may be more effective to create your own, since this approach allows you to completely customize the behavior of the thread pool. Even if you choose to use the .NET ThreadPool class, the lessons you learn from this example should help you with multithreaded programming in general (which is, ultimately, the whole idea of this article).

The frmControl contains a command button and a list box. The command button creates a new task that takes between one and five seconds to run. The list box is updated by a timer to display the number of tasks completed and the number of tasks pending. When you run the program, you should just click the button a dozen times or so to see how the application processes the tasks.

The code that follows, like the Threading1 example in the book, works most of the time. In fact, you may never actually see a problem with this code – like the Threading1 example, the likelihood of an error is probably one per millions of tasks executed (maybe more). Nevertheless, this code is flawed.
Your job, in addition to understanding the code that follows, is to try to anticipate every potential threading problem that may exist.

The ThreadPool consists of an array of 5 thread objects (the array is zero based). There is a first in first out queue in which tasks are stored in preparation for processing. The JobQueued variable is an AutoResetEvent object. This object is signaled any time a job is added to the queue. If there are any idle threads, one and only one thread will be activated when this object is signaled, and its signaled state will immediately return to False. The TotalServed variable keeps track of the total number of tasks completed.

```
Private ThreadPool(4) As Thread

' Queue containing all jobs
Private OperationQueue As New Collections.Queue()

' JobQueued event is signaled when new job is queued
Private JobQueued As New AutoResetEvent(False)

Private TotalServed As Integer
```

When the form loads, the thread pool is initialized and each thread is started. The ThreadFunction method is executed for each thread.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
    Dim ThisThread As Thread
    Dim x As Integer
    For x = 0 To UBound(ThreadPool)
        ThreadPool(x) = New Thread(AddressOf ThreadFunction)
        ThreadPool(x).Start()
    Next
End Sub
```

A task is queued when the button is clicked. In this example, a task is represented by an integer value from 1 to 5 (indicating a one to five second delay). The JobQueued AutoResetEvent object is signaled when the job is queued.

```
Private Sub cmdRequest_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles cmdRequest.Click
    Static RandomGenerator As New Random()
    Dim rnum As Integer
    rnum = RandomGenerator.Next(1, 5)
    OperationQueue.Enqueue(rnum)
    JobQueued.Set()
End Sub
```

The ThreadFunction routine is called for each thread. The thread runs in an infinite loop – we'll be using the exception mechanism to interrupt the idle thread from its wait condition and terminate the thread.
The thread begins by attempting to retrieve an entry from the queue. It is important to check if jobs are available at the beginning and after the completion of each job because of the nature of the JobQueued AutoResetEvent object. To understand why, let's say all threads are busy doing a job, and you add five new tasks to the queue. The JobQueued object will be signaled, which will allow the next thread to enter a wait state to execute. At that point the JobQueued object will no longer be signaled – however, there will still be four tasks waiting in the queue! Think of the JobQueued object as a signal to any idle threads that there is work to be done. If there are no idle threads, it serves no purpose. This approach does mean, however, that there may be times when a thread attempts to retrieve a task from the queue (using the Dequeue method), when the queue is already empty. In this case an InvalidOperationException exception is raised.
When this occurs, the thread enters a wait state, waiting for the JobQueued event to be signaled. This block is in its own Try block because a ThreadInterruptedException will be raised when the application terminates.

```
Private Sub ThreadFunction()
    Dim s As New Server()
    Dim i As Integer
    Do
        Try
            i = OperationQueue.Dequeue()
            s.ServerOp(i * 1000)
            TotalServed += 1
        Catch e As InvalidOperationException
            Try
                JobQueued.WaitOne()
            Catch ti As ThreadInterruptedException
                Exit Sub
            End Try
        End Try
    Loop While True

End Sub
```

Speaking of application termination. The TearDown method is called during the form's Dispose method (a call to TearDown is added to the form's Dispose method, but is not shown here).

The TearDown method starts by blocking any further tasks from being added to the queue. In this example, this is accomplished by disabling the command button.
The queue is then cleared of any existing tasks.
Finally, for each thread, the thread is interrupted. The Join method waits until the thread has terminated.

```
    Private Sub TearDown()
        Dim th As Thread
        ' Block further requests
        cmdRequest.Enabled = False
        ' Clear the queue
        OperationQueue.Clear()

        For Each th In ThreadPool
            th.Interrupt()
            th.Join()
        Next
    End Sub
```

When the TearDown method returns, all of the threads have been terminated. The Timer event simply displays the total tasks completed and the number of tasks pending.

```
    Private Sub Timer1_Tick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Timer1.Tick
        ListBox1.Items.Clear()
        ListBox1.Items.Add("Total served: " & CStr(TotalServed))
        ListBox1.Items.Add("Total pending: " & _
        CStr(OperationQueue.Count))
    End Sub
End Class
```

Thread Pools – Ready for swimming

Before you continue reading, I encourage you to take a few minutes and see if you can spot all of the flaws in the program.
Here's a hint: remember – the first essential step towards creating good multithreaded programming is to ruthlessly detect and synchronize any shared variables.
Ready?

There are actually three types of problems in this example. One is a simple bug. Another is a design flaw. The rest relate to synchronization of shared variables. Here's the list:

1. Simple Bug: If you close the application while tasks are in progress you'll see a runtime error. The program does not trap interruption of a thread that is performing a task.
2. Design Flaw: Interrupting a wait operation is perfectly reasonable. But interrupting a background task – especially one that may be communicating with

a remote server is a bad idea. The program should be redesigned to wait until all background tasks are finished before terminating.

3. The OperationQueue queue is accessed by multiple threads and is not synchronized. Remember: .NET Framework objects are NOT thread safe unless explicitly noted in the documentation.
4. The Total Served integer is accessed by multiple threads and is not synchronized.

To be thorough, we should examine every variable that might be shared. Let's take them one at a time.

- The ThreadPool array is only accessed by the main application thread. So it is safe.
- The JobQueued AutoResetEvent variable is accessed by multiple threads. The .NET documentation does not comment on the thread safety of this object. However, the only method of this object that is called by multiple threads is the WaitOne method, which is specifically designed to be called by multiple threads in the manner shown here.

Let's first look at how to synchronize the shared variables, and trap the runtime exception that occurs when interrupting a task. The ThreadFunction method is rewritten in the MTPool2 sample program as follows:

```
Private Sub ThreadFunction()
    Dim s As New Server()
    Dim i As Integer
    Do
        Try
            SyncLock (OperationQueue.SyncRoot)
                i = OperationQueue.Dequeue()
            End SyncLock
            ' Keep track of # of operations in progress
            WorkCounter.AddOne()
            s.ServerOp(i * 1000)
            WorkCounter.SubtractOne()
            Interlocked.Increment(TotalServed)
        Catch SInterrupted As ThreadInterruptedException
            ' If server operation was interrupted, we exit, _
              but this is probably bad
            Exit Sub
        Catch e As InvalidOperationException
            ' Queue was already empty
            Try
                ' Wait to be signaled
                JobQueued.WaitOne()
            Catch ti As ThreadInterruptedException
                ' This is legit interruption on teardown
                Exit Sub
            End Try
        End Try
    Loop While True

End Sub
```

Ignore the two lines that reference variable WorkCounter – that's part of the solution to the design flaw that I'll discuss shortly.
The TotalServed variable is simply incremented each time a task completes. The Interlocked.Increment method is perfect for this purpose.
Trapping the runtime error that occurred when a task was interrupted is a simple matter of adding another Catch statement to the main Try block.
There are two ways to synchronize the queue. One is to create a queue that is thread safe – you can do this by creating the queue using the shared System.Collections.Queue.Syncrhonized method (that returns a reference to a Queue object that is synchronized). Another, potentially more efficient approach, is to use a SyncLock block as shown in this example. Many .NET Framework objects have a SyncRoot property that allows you to retrieve a reference to the object that you should use for synchronization purposes.
It is important to get into the habit of using the SyncRoot property instead of the object itself. Why? Because of inheritance. What happens if you derive a new class from the Queue class? The base (Queue) class might perform synchronization based on some object in the base class. If those using your class try to synchronize on the derived class, you will see synchronization problems because the base class and those using your class will not be synchronizing on the same object. The SyncRoot property ensures that the base class and all those using derived classes always synchronize to the same object. Every reference to the OperationQueue object must be synchronized, not just the one in the ThreadFunction method (you'll see this in the sample code).

Waiting for Tasks To Complete

To fix the design flaw, we need a way of determining if all of the tasks are complete. The best way to accomplish this would be if we had an object that could keep track of how many tasks were in progress, and which would support a wait operation. In other words, you'd be able to simply wait until the number of tasks executing is equal to zero – preferably with an optional timeout.
This example uses a CounterEvent object to do this. The new TearDown method is defined as follows:

```
Private WorkCounter As New CounterEvent()

' May be called multiple times
Private Sub TearDown()
    Dim th As Thread
    ' Block further requests
    cmdRequest.Enabled = False
    ' Clear the queue
    SyncLock (OperationQueue.SyncRoot)
        OperationQueue.Clear()
    End SyncLock
    ' Allow up to 15 seconds for current operations to finish
    WorkCounter.WaitOne(15000, False)
    For Each th In ThreadPool
        th.Interrupt()
```

```
            th.Join()
        Next
    End Sub
```

The CounterEvent object has methods AddOne and SubtractOne that are called before and after a task begins in the ThreadFunction method. These methods track the number of tasks in progress. The CounterEvent object is signaled any time the number of tasks in progress is zero. The object's WaitOne method will return immediately if signaled, otherwise it will wait for up to 15 seconds for the number of tasks to equal zero.

There is one small catch to this solution. There is no CounterEvent object that works as described here in the .NET Framework class library.

Defining the CounterEvent object

If you are an experienced API programmer, you first thought to solve this problem might be to use a semaphore object. However, there is no semaphore object provided by the .NET Framework and (as you learned in chapter 15), it's better to avoid API calls if possible.

Your second thought, if you were familiar with synchronization objects, would be to somehow inherit from the ManualResetEvent object. A ManualResetEvent object is an object that supports all of the wait operations and is manually Set or Reset. There are two reasons why inheritance is the wrong choice. First: because the new CounterEvent object we're defining does not meet the design criteria for inheritance specified in chapter 5 – that an *is a* relationship exist between the base and derived class. A CounterEvent object is *not* a ManualResetEvent. Second: you can't inherit from ManualResetEvent anyway because the object is not inheritable!

However, you can and should inherit from the WaitHandle object. The WaitHandle object is the base class for all synchronization objects. It implements the WaitOne method that allows you to wait for the object to be signaled. A CounterEvent object *is a* WaitHandle, so it meets our design criteria as well.

The WaitHandle does, however, require that you provide a handle to an operating system synchronization object to use for synchronization. For our purposes what we need is a synchronization object that our class can explicitly signal whenever the internal counter is zero. In fact, a ManualResetEvent object would serve this purpose wonderfully.

As you can see in the following code, the CounterEvent object inherits from WaitHandle, and contains a ManualResetEvent object that is initially signaled (because the initial counter is set to zero as well):

```
' Thread Pool Implementation #2
' Copyright ©2001 by Desaware Inc. All Rights Reserved

Imports System.Threading
Public Class CounterEvent
    Inherits WaitHandle
    Private InternalEvent As New ManualResetEvent(True)
    Private m_Counter As Integer
```

The constructor has two jobs. First, it calls the base constructor – an essential step to making sure the object works correctly. Next, it sets the synchronization object to that of the contained InternalEvent ManualResetEvent object. Any time you signal the InternalEvent object; the WaitHandle (and thus the CounterEvent) object will also be signaled.

```
    Public Sub New()
        MyBase.New()
        MyBase.Handle = InternalEvent.Handle
    End Sub
```

The AddOne and SubtractOne methods simply increment or decrement the m_Counter counter variable. The InternalEvent object (and thus the CounterEvent object) is signaled any time the counter is zero. Note the use of the SyncLock block to make sure that this object is thread safe.

```
    Public Sub AddOne()
        SyncLock InternalEvent
            m_Counter += 1
            If m_Counter = 0 Then
               InternalEvent.Set()
            Else
               InternalEvent.Reset()
            End If
        End SyncLock
    End Sub

    Public Sub SubtractOne()
        SyncLock InternalEvent
            m_Counter -= 1
            If m_Counter = 0 Then
               InternalEvent.Set()
            Else
               InternalEvent.Reset()
            End If
        End SyncLock
    End Sub

End Class
```

Debugging Tricks

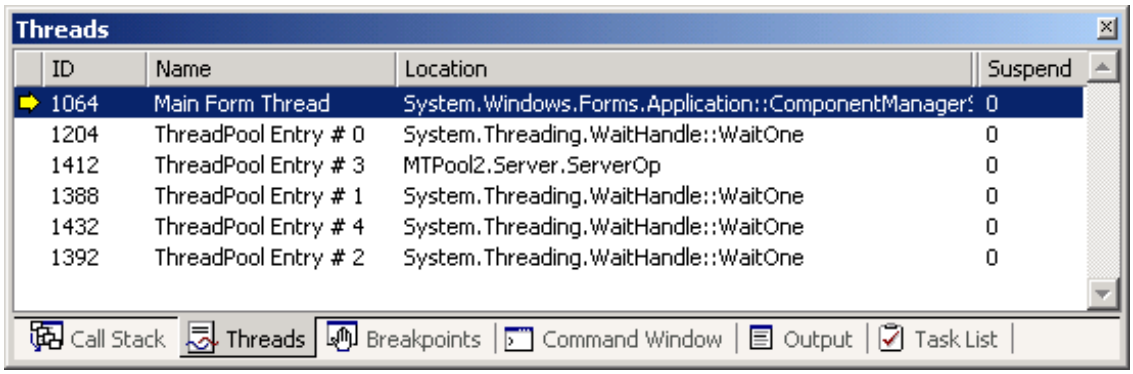
There's just one more trick to the MTPool2 example that I'd like to point out. The Form_Load event now includes the line:

```
ThreadPool(x).Name = "ThreadPool Entry # " & CStr(x)
```

after each thread is created, and the following line as well:

```
Thread.CurrentThread.Name = "Main Form Thread"
```

Giving names to threads provides no functional benefit, but can be a huge help when you start debugging your application. Try running the sample program, then using the Debug-Break command to pause operation. Then bring up the Threads window (under the Debug-Other Windows menu command). It will look something like this:



It is obviously much easier to track threads by name than by ID. In this case you can see exactly where each thread is executing. This window also allows you to freeze the operation of any thread – a helpful feature, since it can otherwise be very difficult to know exactly which thread you are debugging at any given time.

Conclusion

In terms of information, there is little in this article that is not already covered in chapter 7 of the book. But I hope that this in-depth analysis of a simple, yet typical, multithreaded application will help you to develop the habits you'll need to safely create multithreaded applications.

Above all, this example shows he approach that you must use in your design. The MTPool1 example mostly worked. Normal testing and debugging would have quickly identified the unhandled error, but almost certainly would have missed the subtle synchronization problems on the OperationQueue and TotalServed variables. Only a thorough look at every variable, and careful synchronization can ensure that the program will work properly. The good news is that once synchronized, this application should prove very reliable.