Description: Using the Task Parallel Library (TPL)

Part 1 – Parallel.ForEach

Open the Lab4 solution provided.

The Lab4 project is a Windows Forms project. Included in the project is the MathStuff class which contains an algorithm for calculating a given number of digits of math constant pi. We will work with that class in part 2. For part one, we will be focusing on parallel loops.

Open the Form1 designer by double clicking on it in the Solution Explorer. Notice the top half of the form contains many ProgressBar controls. We will be using parallel loops to fill the bars.

Now, go to the *Form1.Desginer.cs* file. You can reach it by right-clicking on *Form1.cs* in the Solution Explorer and selecting "View Code".

Navigate to the Event Handlers region and expand it. Go to the *parallelGoButton_Click* method.

The first thing we need to do is construct a list of items to loop through. We can do this with a LINQ query. All of the ProgressBar controls are contained within a GroupBox called *parallelGroup*. A WinForms container controls has a *Controls* property that contains all of its children. Since we also have a Button control in the group our query will have to filter out unwanted controls:

```
var query =  from Control child in parallelGroup.Controls
             where child is ProgressBar
             select child as ProgressBar;
```

Next, we need to create an anonymous method that will update an individual ProgressBar. We could also use a standard named method, but it is always good to practice new techniques. The reason we need a method (anonymous or standard) is that we will need to execute the ProgressBar updates on the thread that created the controls. This is also known as synchronizing on the UI thread. If we try to update the controls from a separate thread without the synchronization exceptions would result.

```
Action<ProgressBar, int> updateProgressBarValue = delegate(ProgressBar pb, int value)
{
    pb.Value = value;
};
```

Notice that this anonymous method uses the Action delegate to store the method. The method itself accepts a reference to the ProgressBar to update and the new value. We are using Action instead of Func since we don't need a return value.

Now, the fun part… We will use the Parallel.ForEach statement to iterate through the ProgressBar controls returned from "query" in a parallel manner. This time we will use a lambda expression for the body of the loop. This lambda expression is executed once per item from "query". The TPL decides how to allocate threads for each iteration.

```
Parallel.ForEach(query, progressBar =>
    {
        for (int i = 0; i < progressBar.Maximum; i++)
        {
            if (i % 100 == 0)
            {
                progressBar.BeginInvoke(updateProgressBarValue, progressBar, i);
            }
        }
    }
);
```

Pay close attention to the inner for loop that iterates through the ProgressBar's maximum value. Notice that every time the value "i" is evenly divisible by 100 we use the BeginInvoke method of the ProgressBar to execute the anonymous method stored in *updateProgressBarValue*. This executes *updateProgressBarValue* on the thread that the ProgressBar was created on. Because we are using a UI control's BeginInvoke method, we are not required to call EndInvoke to complete the call. Typically, any time you call BeginInvoke you must call EndInvoke to prevent memory leaks. Also, we are using BeginInvoke instead of Invoke to avoid blocking on the call.

Now, run the project and click the Go button in the top GroupBox named "Parallel ForEach". The ProgressBars should fill in a relatively parallel manner. Various factors can effect if all bars are filled at the same rate or even start at relatively the same time. Running this on computers with different hardware can greatly affect the results.

Part 2 – Cancelling Tasks

For this part of the lab we will use TPL's Task objects to spawn a thread and provide a mechanism for cancellation.

Take a look at the MathStuff class. In here is an algorithm to calculate a desired number of digits of pi. Pi represents the ratio of a circle's circumference to its diameter and is frequently used to calculate the area or circumference of circles.

Math.PI in System.Math is defined as:

```
public const double PI = 3.14159;
```

If you want more precision, you can create a decimal that can hold up to 28 digits of pi:

```
decimal pi = 3.14159265358979323846264338 3M;
```

28 digits of pi should be more than enough for any realistic computations. From "Cool Science Facts":

"if you had a circle the size of the observable universe, and you wanted to compute its circumference with an accuracy equal to the size of a proton, the number of digits of pi that you'd need is only 43"

- http://www.coolsciencefacts.com/2006/pi.html

We are not interested in practicality, so we will calculate pi to a large number of digits. The function we are using is a slightly modified version of the algorithm located at:

- http://kashfarooq.wordpress.com/2011/07/23/calculating-pi-in-c-part-2-using-the-net-4-biginteger-class/

This algorithm is executed by calling MathStuff.Calculate and providing it the number of digits we want to calculate for pi. The way we will be using it is not the most efficient way to calculate digits of pi, but it is better some dummy formula.

First, we need to create a CancellationTokenSource object that will be monitored by our task. Create a class-level field in Form1 of that type called m_TokenSource and set it to null.

```
CancellationTokenSource m_TokenSource = null;
```

Next, we need a mechanism much like the one used in Part 1 to update a value on the UI thread from within our task. Instead of an anonymous method, this time we will use a standard method. Add a method with the following signature to the Form1 class:

```
private void UpdatePiTextBlock(string pi) {  }
```

Within this method we need to check if cross-thread synchronization is required. We can do this by checking the specific control's InvokeRequired property. This is one of the few properties we can access safely across threads.

```
if (piTextBox.InvokeRequired)
```

If this condition is true, then we need to recursively invoke this method on the UI thread. Just as before we will use the control's BeginInvoke method to do this.

```
piTextBox.BeginInvoke(new Action<string>(UpdatePiTextBlock), pi);
```

Notice how the method we are invoking is the same one that we are currently in. This is known as recursion (a method that directly or indirectly calls itself).

Now we need to add the "else" part of the InvokeRequired statement. Basically, if InvokeRequired returned false, simply update the piTextBox with the pi value given in the method:

```
else
    piTextBox.Text = pi;
```

So, if UpdatePiTextBlock is called from our task, the test for InvokeRequired required will return true. This will then asynchronously call UpdatePiTextBlock recursively. This time, however, since it was invoked upon the UI thread, InvokeRequired will be false so the else portion of the function executes, updating the TextBox.

The next step is to create the task that will calculate the digits of pi by calling MathStuff.Calculate(). Like standard threads, a Task is typically implemented as a method that accepts some number of parameters and can return a value. When the method exits, the Task is complete. We will start with the following method definition:

```
private void CalculatePiTask()
{
    try
    {
        // Task code will go here
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Cancelled.");
        throw;
    }
}
```

Strictly speaking, the try block is not necessary. However, we are going to monitor for an OperationCanceledException and output a message to the Console (Output window for WinForms) when the user cancels the Task. For a Task to monitor for cancellation, we need a reference to a CancellationTokenSource object. Remember that we have one that is defined at the class-level of Form1, so we'll use that. You could also pass it to the Task as a parameter, but this is the easiest way to handle it. So, inside the try block we will execute a for loop from 2 to 1,000,000. At every iteration we will check to see if cancellation was requested. If not we will calculate pi to "i" number of digits and update the UI:

```
for (int i = 2; i < 1000000; i++)
{
    m_TokenSource.Token.ThrowIfCancellationRequested();
    string pi = MathStuff.Calculate(i);
    UpdatePiTextBlock(pi);
}
```

Notice that I call ThrowIfCancellationRequested at every iteration of the loop. This is because that function basically encapsulates the following code:

```
if (m_TokenSource.Token.IsCancellationRequested) throw new OperationCanceledException();
```

It is a little more complicated than this, but it is a fair comparison. If your loops are very fast and you want to space out this check you could use a modulus operator to perform the check at periodic intervals (Part 1 demonstrates the modulus technique). To make the calls take a little longer, you can add a sleep statement. This allows you to see the digits being created one-by-one (especially for the first few hundred). The following code shows the CalculatePiTask method in its entirety:

```
private void CalculatePiTask()
{
    try
    {
        for (int i = 2; i < 1000000; i++)
        {
            // Call ThrowIfCancellationRequested periodically to test for cancellation request
            m_TokenSource.Token.ThrowIfCancellationRequested();
            DateTime start = DateTime.Now;

            string pi = MathStuff.Calculate(i);
            UpdatePiTextBlock(pi);

            DateTime end = DateTime.Now;

            double totalMs = end.Subtract(start).TotalMilliseconds;
            if (totalMs < 500)
            {
                // Slow down the task for visual purposes only
                Thread.Sleep(500 - (int)totalMs);
            }
        }
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Cancelled.");
        throw;
    }
}
```

Next, let's add code to the taskCancelButton event handler *taskCancelButton_Click*. This method is called when the user presses the Cancel button in the bottom GroupBox. The code is very trivial:

```
if (m_TokenSource != null)
{
   m_TokenSource.Cancel();
}
```

This basically says: If a valid token source is available then signal to any listeners that the Task has been cancelled.

Finally, we will add the code to *taskGoButton_Click* which is fired when the user clicks the Go button in the bottom GroupBox. This will start the Task to calculate the digits of pi.

Our first step is to set the buttons into the appropriate state:

```
taskGoButton.Enabled = false;
taskCancelButton.Enabled = true;
```

Then, set the CancelationTokenSource object to a new instance:

```
m_TokenSource = new CancellationTokenSource();
```

When Tasks complete, TPL provides a mechanism to automatically continue execution on a WinForm or WPF UI thread. To do this we need a handle to the UI thread which we'll use later:

```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
```

Now, start the Task, passing it the name of the method that will be the entry point for the Task and the token used for cancellation:

```
Task calculatePiTask = Task.Factory.StartNew(CalculatePiTask, m_TokenSource.Token);
```

The Task thread will automatically start after the StartNew call. We now can add some continuation methods to that task, so that when it completes it will automatically perform other actions:

```
var resultOK = calculatePiTask.ContinueWith(resultTask =>
    {
        MessageBox.Show("Calculation fininshed", "Task Complete",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        taskCancelButton.Enabled = false;
        taskGoButton.Enabled = true;
    },
    CancellationToken.None,
    TaskContinuationOptions.OnlyOnRanToCompletion,
    ui);

var resultCancel = calculatePiTask.ContinueWith(resultTask =>
    {
        MessageBox.Show("Calculation stopped by user", "Task Cancelled",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
        taskCancelButton.Enabled = false;
        taskGoButton.Enabled = true;
    },
    CancellationToken.None,
    TaskContinuationOptions.OnlyOnCanceled,
    ui);
```

The first continuation call will execute only when the Task completes successfully (see second to last parameter - TaskContinuationOptions.OnlyOnRanToCompletion). This is an unlikely scenario in our case as calculating pi to 1,000,000 digits incrementally will take a long time. But, if it does complete, it notifies the user by showing a message box then resetting the buttons to their default configuration. Notice that the last parameter to ContinueWith is "ui". This tells ContinueWith to execute the lambda statements to execute within the context of the UI thread. No Invoke/BeginInvoke needed here. The second continuation call only executes if the task is cancelled – again, check the second to last parameter. The rest of the code is very similar to the first continuation call. Please take note that this method, *taskGoButton_Click*, does not wait around for these continuation methods to call. In fact, the method terminates very quickly, allowing the UI to continue without interrupting the user experience.

Run the code and click the bottom Go button. When you have seen enough digits to suit your needs, click cancel and watch the cancellation continuation function fire and show you the message box.

Note: Don't worry if the debugger stops on the "throw" statement in *taskGoButton_Click*. This is most likely because your settings are set to break on all unhandled exceptions. Just hit F5 to continue.

Tasks can be a powerful addition to your library of knowledge – just be aware there is a somewhat steep learning curve to master all of its capabilities.