<div align="center">

Santiago Canyon College
Computer Science
The .Net Threading Model

</div>

Introduction

The purpose of this paper is to introduce you to multi-threading in Visual Studio. Learning how to take advantage of this technique can greatly improve the performance of your VB & C# apps. This discussion is limited to managed code applications and not C++ or C unmanaged native code solutions. Managed code is run by the CLR and unmanaged code is handled by the O/S.

When you create a Windows form, the controls and code are executed or handled in the order they are triggered. If I have two buttons and the first one starts a long loop or internet search, the form will appear to be unresponsive until that task completes. Only then will I be able to click on the second button and have it respond.

This behavior happens because when you have only one CPU, it can only process one stream of instructions at a time. This paper will try to show you how to prevent this type of behavior by running the code from the first button on one stream and the code for the second button on another stream. In this way, the O/S will be able to start one stream while remaining responsive to the UI and even starting another section of code.

What Happens When We Push F5 in VS?

- When you launch an app you create an **instance** of the application and the O/S creates a **process** within which our entire project will be managed.
- Inside this process, your app is loaded into an **App Domain** by the O/S and CLR and your assembly (MSIL) .exe. or .dll is in its own protected space in RAM. Several application domains can exist within one process.
- The CLR determines the entry/starting point for your app and assigns it one main **thread** called the UI thread.
    - A thread is the object that sends a stream of instructions to the CPU.
    - All code/controls created in the form will run on this UI thread.
    - Only one method can run at a time in this scenario.
- Each method/section of your code is now viewed as a **task**.
- In order to execute code, a thread of execution (just thread) is instantiated from the group of threads (**thread pool**) created by the CLR.
- Each task is added to a list of tasks we want the CPU to execute called the **task queue.**
- When a method is started the JIT (just-in-time compiler) assigns that method to a thread.

- When the O/S gives our app its attention, the thread sends the code to the CPU for processing.
  - Remember, in a multi-tasking O/S like Windows, the CPU processes code for a given thread for about 20ms (20 thousandths of a second) then it turns its attention to another task until it gets back to our task. It makes it look like it can process tasks simultaneously but in fact it cannot. By switching between tasks at short intervals, it gives us the illusion it is multi-processing. Only multi-core computers can actually do more than one thing at a time.
  - When the method ends, the thread is released from the task and becomes available to the thread pool again to be used over.
- You do not start a task directly. By default, the thread pool schedules a task, places it on a queue in the thread pool, and eventually executes the task on an available thread. *Starting* a task implies queuing the task first and later executing the task on an available thread from the thread pool.

Below is a visual summary of our previous discussion. The items inside the grey box are in our **assembly** which is contained inside an **App Domain** which is itself inside a Windows **Process which manages security/authentication/memory, etc**. The App Domain isolates our code for protection.
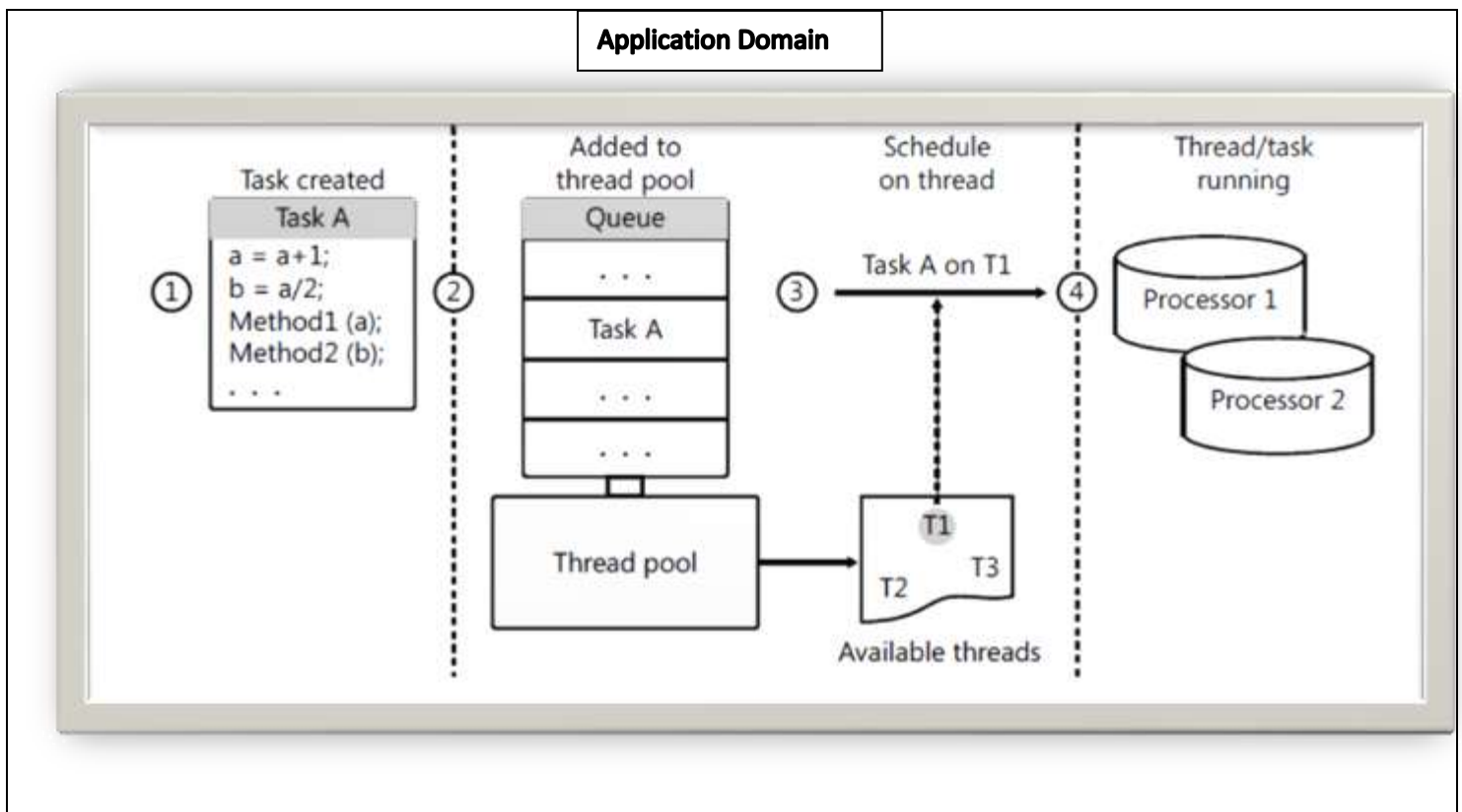
Applications, Processes, & Threads

Application Domain

| Task created | Added to thread pool | Schedule on thread | Thread/task running |

Figure 1: How Code is executed on Threads
From Parallel Programming with MS Visual Studio 2010 Step by Step

Multi-Threading: Using the Background Worker Class

Visual Studio contains a class called the BackgroundWorker (BW). There is also a control in the toolbox you can drag to a form to use this class (that is what you should do!). This control will let you run a section of code on a background thread. In this way, your UI remains responsive and free to do something else while the background tasks runs. This control will also tell you when it is finished. It can be cancelled (if you set the property) and it can report progress as it works if you want. This allows you to update a progress bar on the form, for example.
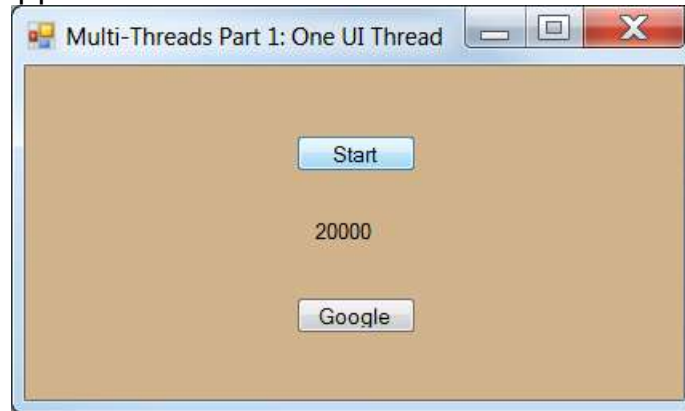
It is important to picture what happens on a normal VS application. The UI runs on the main thread. The background task runs on a separate thread. In my humble opinion, you never want to try to access code running on one thread from a different thread. Let this BW control handle this for you. Threading is very powerful and also very dangerous if you try to manage things yourself!

In the example code I will show you, you can see that the BW can update the UI thread while it is working on a background task. If you understand how it works, it will make threading an easy way for you to improve performance in your applications.

Several class members are fundamental to the operation of a BackgroundWorker object:

- You start the time-consuming operation in the DoWork event handler.
- You start the operation by calling RunWorkerAsync, which in turn raises the DoWork event.
- You receive updates of progress, if desired, by handling the ProgressChanged event.
- You receive a notification of completion, if desired, by handling the RunWorkerCompleted event.
- You cancel the operation by calling the CancelAsync method.
    - Properties you need to use:
        - BackgroundWorker1.WorkerReportsProgress = True
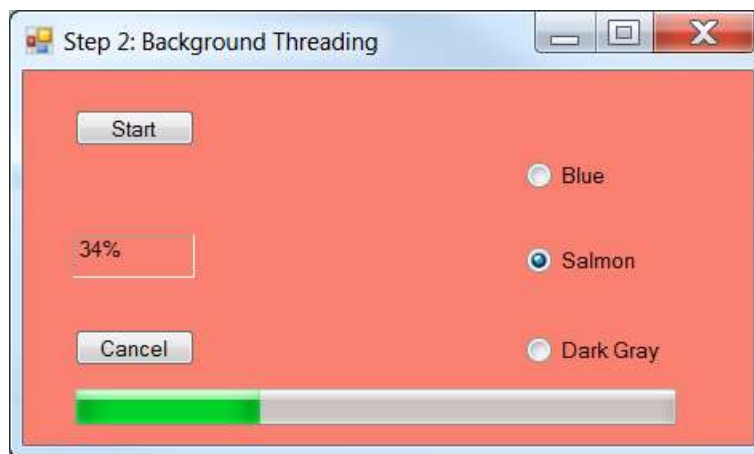   - BackgroundWorker1.WorkerSupportsCancellation = True

Let's take a look at an example. Here is a simple form and when you click start, a loop begins. While this loop executes, I cannot navigate to Google because the CPU is busy running my loop. As soon as the loop ends, the button responds. During the loop, I cannot even close the app!

Our Solution: Run Time-Intensive Code Asynchronously On a Background Thread

So let's say that you need to search a huge database or the internet for information and you know it could be a time-consuming task. You want the UI to be responsive while this search is taking place. If you make the search code run on a background thread, you will have a much nicer running app.

In the second form, I use the same loop but run it on a background thread. Now, the radio buttons respond while the loop is running and the progress bar works as expected.

Steps to build this app:

1. Create a new Windows app in VB.
2. Add the controls you see. The label with the 34% is lblCount. The radio buttons change the form.background color.
3. Add a progress bar control. Set its Step property to 1 and Style to "blocks".
4. Drag a background worker control to the form.
5. Add the following Imports/using statements
   Imports System.Threading

```vbnet
Imports System.ComponentModel
```

6. Add a constructor to your form to initialize the background worker control.

```vbnet
Public Sub New()

    InitializeComponent()              'leave this alone

    BackgroundWorker1.WorkerReportsProgress = True
    BackgroundWorker1.WorkerSupportsCancellation = True
    ProgressBar1.Step = 1

End Sub
```

7. Create a click event handler for the start button. This code starts your BW.

```vbnet
Private Sub btnStart_Click(sender As System.Object, e As System.EventArgs) Handles btnStart.Click
    If BackgroundWorker1.IsBusy = False Then
        BackgroundWorker1.RunWorkerAsync()

    End If
End Sub
```

8. Now you must make a method that contains the code to run on the BW control. It must be inside the DoWork method! Notice I make this thread sleep (pause) for 200ms to simulate a really long internet search. Make sure to remove it on a real app!

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

    Dim worker As BackgroundWorker = CType(sender, BackgroundWorker)

    For x As Integer = 1 To 100

        If (worker.CancellationPending = True) Then
            e.Cancel = True
            Exit For
        Else
            Thread.Sleep(200)                'this simulates a long internet search.
            worker.ReportProgress(x)
        End If
    Next
End Sub
```

9. To show progress and to notify when the BW finishes, add these methods:
The ProgressChanged event is the only place you should update the UI because it is running on another thread!

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

 '---update the label on the UI thread and also the Progressbar.
        lblCount.Text = e.ProgressPercentage.ToString() + "%"
        ProgressBar1.Value = e.ProgressPercentage
End Sub


Private Sub BackgroundWorker1_RunWorkerCompleted(sender As System.Object, e As System.ComponentModel.RunWorkerCompletedEventArgs) Handles BackgroundWorker1.RunWorkerCompleted
        MessageBox.Show("The background thread is done", "System Message", MessageBoxButtons.OK, MessageBoxIcon.Information)
End Sub
```

10. To complete the BW stuff, add this method to handle the Cancel button:

```vb
    Private Sub btnCancel_Click(sender As System.Object, e As System.EventArgs) Handles btnCancel.Click

        If BackgroundWorker1.WorkerSupportsCancellation = True Then
          ' Cancel the asynchronous operation.
          BackgroundWorker1.CancelAsync()
        End If

    End Sub
```

11. Finally, add code for the radio buttons:

```vb
        Private Sub radioButtons_Changed(sender As System.Object, e As System.EventArgs) Handles rdoBlue.CheckedChanged, rdoGray.CheckedChanged, rdoSalmon.CheckedChanged

            Dim whichControl As RadioButton = sender
            Select Case whichControl.Name

                Case "rdoBlue"
                    Me.BackColor = Color.Blue
                Case "rdoGray"
                    Me.BackColor = Color.DarkGray
                Case "rdoSalmon"
                    Me.BackColor = Color.Salmon
                Case Else
                    Me.BackColor = Color.Gray

            End Select
        End Sub
```

12. Run the app. You should notice that the loop counts from 1-100% and during the process, your UI controls work also!!

In conclusion, you may not realize it but every time you use Word, you are taking advantage of multi-threading. As you type on the main thread, a background thread is running the spell checker code! This is how you can type and get immediate spelling suggestions at the same time.