Background Worker in .NET 2.0

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

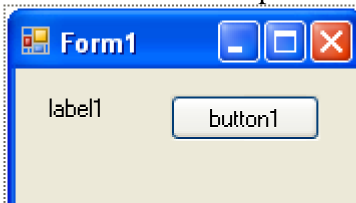
Abstract

Delegating a task to a separate thread is easy. However, updating user controls with result of those tasks was not, until .NET 2.0. BackgroundWorker in .NET 2.0 makes life easier. It allows you to delegate tasks to be executed in a separate thread, and provides a thread-safe way to update the user controls as well. In this article, we will explore the capabilities of BackgroundWorker component.

Multithreading Woes

Let's start with an example. Assume we have a windows application which, upon the press of a button, should perform an operation that may take some time. We certainly don't want to perform that task in the main (event handling) thread. This would make the application non-responsive. So, we decide to put that code in a separate thread. Here is the first attempt to do just that.

I have created a simple WinForm application with a Form as shown below:



I've added an event handler for the button (by double clicking on button1) as shown below:

```
private void button1_Click(object sender, EventArgs e)
{
    button1.Enabled = false;
    new Thread(Work).Start();
}
```

The method `Work()` is shown below:

```
private void Work()
{
    // Simulates a time consuming task
    int index = 10;
    int total = 0;
    for (int i = 0; i < index; i++)
    {
        total += i;
        Thread.Sleep(1000); //Simulates delay
    }


    label1.Text = total.ToString(); // Wrong, don't do this.
```

```
            button1.Enabled = true; // Wrong, don't do this here.
        }
```

If you execute the program, it may appear to run fine. However, there is a problem with this code. We will discuss this further in the next section.

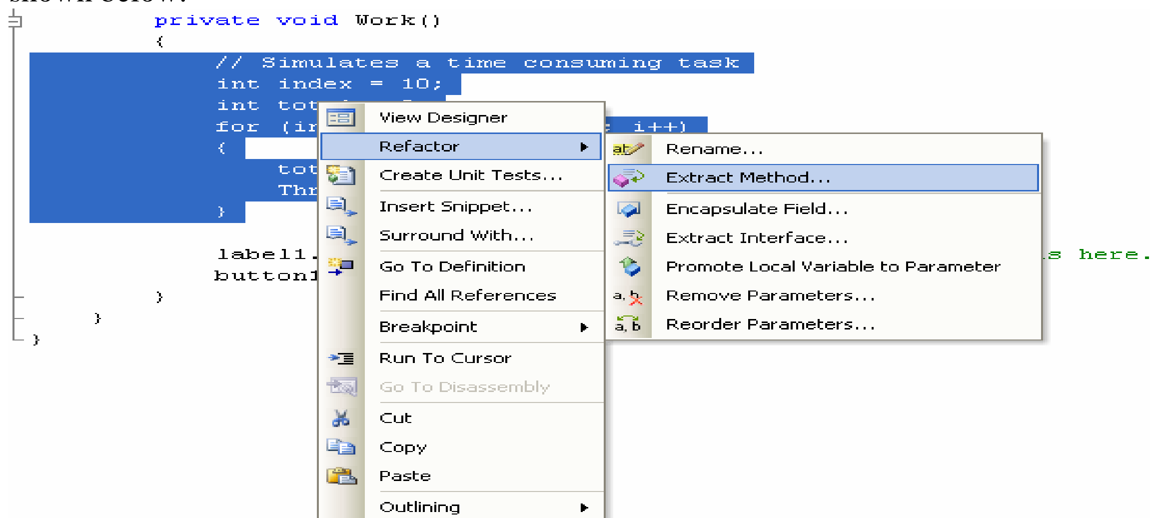
Thread-safety

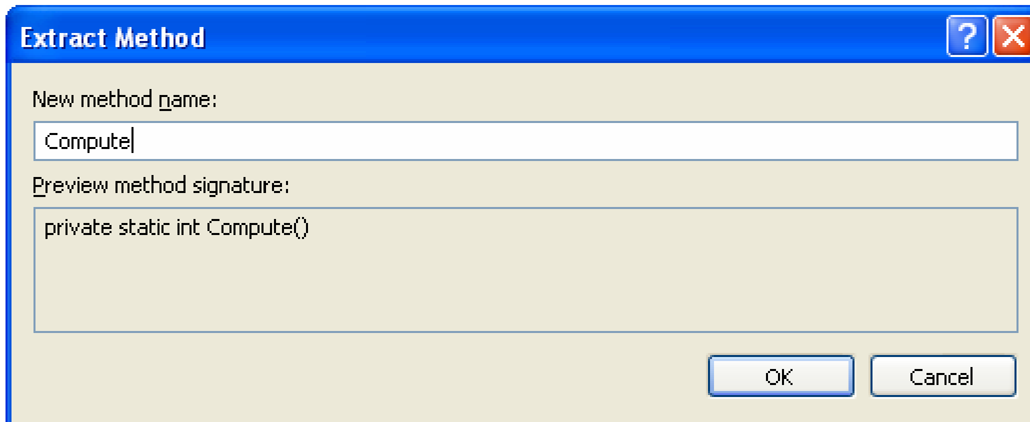
The `Work()` method, in the above example, executes in a separate thread from the main event handling thread. However, from this method we are updating the `Text` property of the label control and the `Enabled` property of the button as well. This is a no-no. Why? Because, only the methods `Invoke()`, `BeginInvoke()`, `EndInvoke()`, `CreateGraphics()`, and property `InvokeRequired` are thread-safe. Other methods and properties of control aren't thread-safe. What does that mean? If a method is thread-safe, it indicates that you are allowed to call that method from any thread. This means that either the method doesn't have a problem with contention (contention raises the possibility that multiple threads will collide over or overwrite some critical data) or it synchronizes the calls to avoid the contention.

It is good that most methods and properties of control aren't thread-safe. Why's that? Synchronization has a price. If your code has locks, then, even if you only have one thread running in your application, the thread has to take time to acquire the lock and then relinquish the lock. This can be an unnecessary overhead and can be avoided by agreeing to access the controls from only the event handling thread.

Pre .NET 2.0 Solution

The solution in pre .NET 2.0 is to jump threads[1]. But, before we try to do that, it will help to refactor the code so it is easier to work with separate concerns. If you take a closer look at the `Work()` method, you notice that it does two things. One, the computation itself (the time consuming task that we are emulating), and second, displaying the result. Let's refactor[2] this method using the *extract method* refactoring capability of VS 2005[3] as shown below:





After the above refactoring, the code looks like the following:

```
private void Work()
{
    int total = Compute();

    label1.Text = total.ToString(); // Wrong, don't do this.
    button1.Enabled = true; // Wrong, don't do this here.
}

private /* static */ int Compute()
    // Removed static placed by tool. We need this to be
    // instance methods for later use.
{
    // Simulates a time consuming task
    int index = 10;
    int total = 0;
    for (int i = 0; i < index; i++)
    {
        total += i;
        Thread.Sleep(1000); //Simulates delay
    }
    return total;
}
```

We can now refactor the code that accesses the controls in the `Work()` method as well. The resulting refactored code should look like the following:

```
private void Work()
{
    int total = Compute();

    Display(total);
}

private void Display(int total)
{
    label1.Text = total.ToString(); // Wrong, don't do this.
    button1.Enabled = true; // Wrong, don't do this here.
}
```

We make small modification to the code so the `Display()` method accepts a string instead of an `int` as shown below:

```
private void Work()
{
    int total = Compute();

    Display(total.ToString());
}

private void Display(string text)
{
    label1.Text = text;
    button1.Enabled = true;
}
```

Now, while the `Compute()` method can be executed in the worker thread, the `Display()` should not. The code to switch threads is shown below:

```
private delegate void DisplayDelegate(string text);
private void Display(string text)
{
    if (InvokeRequired)
    {
        Invoke(new DisplayDelegate(Display),
            new object[] { text });
    }
    else
    {
        label1.Text = text;
        button1.Enabled = true;
    }
}
```

This implementation has some nice features[1]. We don't have to worry about which thread is invoking the Display method. Within the method, we quietly switch to the appropriate thread. The code is succinct as well.

Let's take a look at the relevant code in entirety.

```
private void button1_Click(object sender, EventArgs e)
{
    button1.Enabled = false;
    new Thread(Work).Start();
}

private void Work()
{
    int total = Compute();

    Display(total.ToString());
```

```
        }

        private delegate void DisplayDelegate(string text);
        private void Display(string text)
        {
            if (InvokeRequired)
            {
                Invoke(new DisplayDelegate(Display),
                    new object[] { text });
            }
            else
            {
                label1.Text = text;
                button1.Enabled = true;
            }
        }

        private int Compute()
        {
            // Simulates a time consuming task
            int index = 10;
            int total = 0;
            for (int i = 0; i < index; i++)
            {
                total += i;
                Thread.Sleep(1000); //Simulates delay
            }
            return total;
        }
```

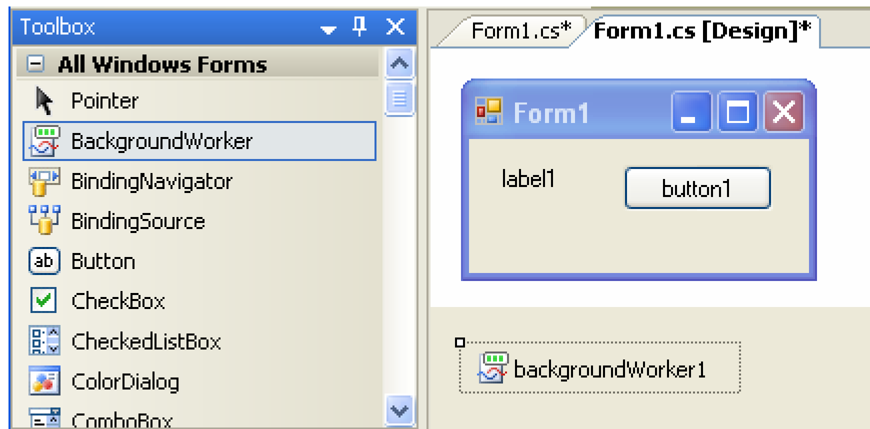
Once we understand what's going on, it is not too bad. However, can it be better? Can it be simpler? Let's ask for just one more features before answering that question. What if, when in the middle of the `Compute()` method, we want to report progress as to where we are? How do we do that? We have to write code again to switch threads and this can become tedious and unwieldy. Let's leave it at that and see how .NET 2.0 makes life easier.

BackgroundWorker in .NET 2.0

BackgroundWorker[4] is a component that allows you to delegate a long running task to a different thread. It doesn't stop with that. You can place the component on a windows form (it is a non-UI control, so it goes into the component tray). You can register event handlers with it. It takes care of running the long running task in separate thread while running the task to update control (to report result or progress) in the main event handling thread. These features make it very easy to deal with the problem like the example above has shown.

Delegating the task and reporting completion

Let's continue with our example. Let's drag and drop a BackgroundWorker class onto the Form as shown below:



The component we dragged and dropped appears in the component tray in the bottom (I've made the IDE window really small so we can see all relevant information here).

Right click on `backgroundWorker1` and select properties. In the properties window, go to the Events tab and double click on the DoWork event to add a handler. Similarly, double click on RunWorkerCompleted to add an event handler for that as well. We have added code to the generated handlers as shown below:

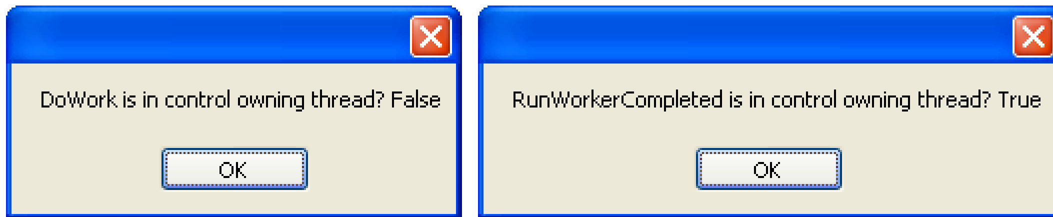
```
private void backgroundWorker1_DoWork(object sender,
                                      DoWorkEventArgs e)
{
        MessageBox.Show("DoWork is in control owning thread? "
               + !InvokeRequired);
}

private void backgroundWorker1_RunWorkerCompleted(
                object sender, RunWorkerCompletedEventArgs e)
{
    MessageBox.Show(
          "RunWorkerCompleted is in control owning thread? "
          + !InvokeRequired);
}
```

I've also added a line to the button event handler to delegate the task to the BackgroundWorker component, as shown below:

```
private void button1_Click(object sender, EventArgs e)
{
    button1.Enabled = false;
    //new Thread(Work).Start();
    backgroundWorker1.RunWorkerAsync();
}
```

I've commented out the code that created a new thread as we don't need that any more.

When we run the application and click on `button1`, we see the following displayed:



We see that the DoWork method is executed in a different thread and the RunWorker is executed in the main (control owning event handling) thread.

Enough studying, let's put the two methods to use in our example.

```
private void backgroundWorker1_DoWork(
                object sender, DoWorkEventArgs e)
{
    e.Result = Compute();
}

private void backgroundWorker1_RunWorkerCompleted(
                object sender, RunWorkerCompletedEventArgs e)
{
    Display(e.Result.ToString());
}
```

Our earlier effort to refactor the code came in handy. We are simply calling the `Compute()` method from within the `DoWork` event handler. We then stuff the result into the `DoWorkEventArgs` object. When the `DoWork` event completes, the `RunWorkerCompleted` event is executed in the main thread. In this method we pick up the result of the delegated task and display it. Go ahead and try the example. We still have one more small talks to do. We have to get rid of some code we don't need. After some cleaning up, our code looks like this:

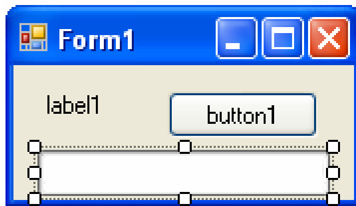
```
private void button1_Click(object sender, EventArgs e)
{
    button1.Enabled = false;
    backgroundWorker1.RunWorkerAsync();
}

private void Display(string text)
{
    label1.Text = text;
    button1.Enabled = true;
}

private int Compute()
{
    // Simulates a time consuming task
    int index = 10;
    int total = 0;
    for (int i = 0; i < index; i++)
    {
```

```
                total += i;
                Thread.Sleep(1000); //Simulates delay
            }
            return total;
        }

        private void backgroundWorker1_DoWork(
                                object sender, DoWorkEventArgs e)
        {
            e.Result = Compute();
        }

        private void backgroundWorker1_RunWorkerCompleted(
                                object sender, RunWorkerCompletedEventArgs e)
        {
            Display(e.Result.ToString());
        }
```

Reporting progress

Let's do one more thing before we call this done. How about reporting progress while we're in the middle of the `Compute()` method?

Let's add a progress bar to the form first as shown here:



Now, let's add the event handler for the `backgroundworker1`'s `ProgressChanged` event. Here is the code to update the progress bar:

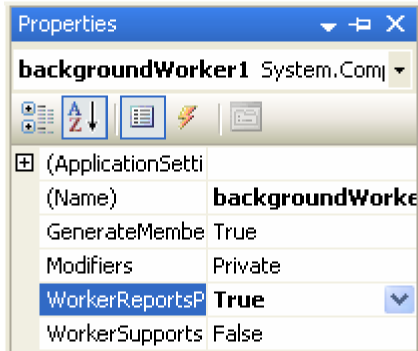
```
        private void backgroundWorker1_ProgressChanged(
                    object sender, ProgressChangedEventArgs e)
        {
            progressBar1.Value = e.ProgressPercentage;
        }
```

But, we've not asked the application to invoke this event handler, yet. So, here it is in the `Compute()` method.

```
        private int Compute()
        {
            // Simulates a time consuming task
            int index = 10;
            int total = 0;
            for (int i = 0; i < index; i++)
            {
                total += i;
                backgroundWorker1.ReportProgress(i * 10 + 10);
```

```
            Thread.Sleep(1000); //Simulates delay
        }
        return total;
    }
```

Before we go for a test drive, there is one last thing we need to turn on–we need to ask the `backgroundworker1` to report progress by setting the `WorkerReportsProgress` property to `true`.



Alright, now take it for a test drive and see what happens.

One disadvantage you may see here is that the `Compute()` method which was fairly cohesive by its focus on (nose into) the computation is now interested in reporting progress as well. As a purist we may break that part away from `Compute()` method into a separate method that deals with updating progress.

Conclusion

When delegating tasks in a windows application we need to be careful about accessing the control due to thread-safety issues. BackgroundWorker component provided in .NET 2.0 comes to rescue to address these concerns. It provides an easier way to delegate task, report progress, and update controls with result, while making sure the activities are carried out in the appropriate threads.

References

1. Venkat Subramaniam, ".NET Gotchas," O'Reilly.
2. Martin Fowler, et. al., "Refactoring: Improving the Design of Existing Code," Addison-Wesley.
3. Visual Studio 2005 – http://msdn.microsoft.com/vs2005.
4. MSDN Documentation – http://msdn.microsoft.com.