**c# wait for thread to finish without blocking**

In [Part 101](http://csharp-video-tutorials.blogspot.com/2016/10/async-and-await-in-c-example.html), we discussed creating a simple responsive windows forms application using Task, and async & await keywords. In this video we will discuss how to do the same using a Thread instead of Task. This is continuation to [Part 101](http://csharp-video-tutorials.blogspot.com/2016/10/async-and-await-in-c-example.html). Please watch [Part 101](http://csharp-video-tutorials.blogspot.com/2016/10/async-and-await-in-c-example.html) from [C# tutorial](https://www.youtube.com/playlist?list=PLAC325451207E3105) before proceeding.

To use a Thread instead of a Task we only need to change btnProcessFile\_Click() method as shown below.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() => { count = CountCharacters(); });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

    lblCount.Text = count.ToString() + " characters in file";

}

At this point the application does not work as expected. We have two problems with the above code.  
1. We do not see the message, "Processing file. Please wait." at all  
2. It displays "0 characters in file"  
  
**Why is this happening**  
The Main thread i.e the UI thread has created a worker thread which executes CountCharacters() function. The worker thread takes at least 5 seconds to complete. In the mean time the Main thread continues executing the following 2 lines of code.

lblCount.Text = "Processing file. Please wait...";

lblCount.Text = count.ToString() + " characters in file";

**But why didn't we see the message "Processing file. Please wait..."**

This is because, the UI thread executes the above 2 lines of code so fast that the second message overwrites the first message and at that speed it is impossible for a human eye to spot the overwriting.  
  
**How to solve the above two problems**  
It is very simple. The Main thread has to wait for the worker thread to finish it's work before the UI thread can display the second message. We achieve this by using Join() method on the worker thread.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() => { count = CountCharacters(); });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

    // Join() blocks the Main thread (UI Thread)

    thread.Join();

    lblCount.Text = count.ToString() + " characters in file";

}

At this point run the application and test it. We have fixed the above two problems but introduced a new problem. While the application is busy processning the file, the UI is blocked i.e we cannot move the form around or resize it.  
  
You may be thinking why can't we move the code that updates the label control Text property into the worker thread as shown below. This is dangerous because, the thread that has created the control must modify the control. In our case the Main thread (i.e UI Thread) is the thread that has created the label control so only the Main thread should set it's Text property and not the worker thread. If you run the application it may or may not work as expected. If it is working, it is only working by blind luck.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() =>

    {

        count = CountCharacters();

        // This is dangerous

        lblCount.Text = count.ToString() + " characters in file";

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

The right way to achieve this is by using BeginInvoke() method as shown below. BeginInvoke() method asks the UI thread to set the Text property of the label control in a type safe manner.

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    int count = 0;

    Thread thread = new Thread(() =>

    {

        count = CountCharacters();

        Action action = () => lblCount.Text = count.ToString() + " characters in file";

        this.BeginInvoke(action);

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

In the example above, notice that the Action delegate points to a piece of code. The Action delegate is then passed to the BeginInvoke() method which asks the UI thread to execute that piece of code asynchronously in a type safe manner. The above code can also be rewritten as shown below.

int characterCount = 0;

private void btnProcessFile\_Click(object sender, EventArgs e)

{

    Thread thread = new Thread(() =>

    {

        characterCount = CountCharacters();

        // Action delegate points to SetLabelTextProperty method

        // Signature of SetLabelTextProperty() method should match

        // with the signature of Action delegate

        Action action = new Action(SetLabelTextProperty);

        this.BeginInvoke(action);

    });

    thread.Start();

    lblCount.Text = "Processing file. Please wait...";

}

private void SetLabelTextProperty()

{

    lblCount.Text = characterCount.ToString() + " characters in file";

}

Asynchronous implementation is very easy with tasks, and async & await keywords. Though the above example is a very simple example, notice the code is already getting relatively complicated. Imagine if we have multiple threads, and we want to use the result of one thread from another thread and so on and so forth. It can get painful and complicated. In our previous video, we have seen how easy it is to achieve exactly the same thing using a Task.  
  
**Next video :** We will discuss the difference between a Thread and a Task