UNIVERSITY OF Hull

Threading

Rob Miles

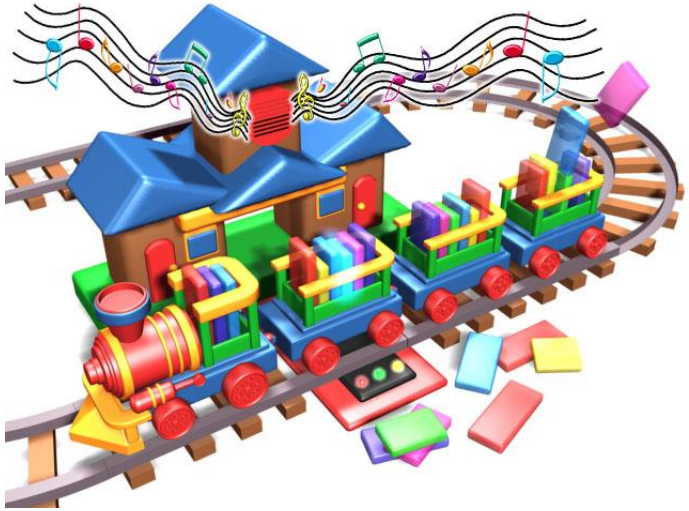
Department of Computer Science

Threads and Program Execution

- At the moment we have not thought much about how our programs actually run
- We have just said that the computer starts at the beginning of the `Main` method and executes statements until it reaches the end
- When we add decisions and loops we control the path of execution through the code
- We see this when we step through code

What is a Thread?

- A thread is a “unit of execution”
- This is probably not a helpful definition
- Think of a thread as a train on a track
- The track is your program, the train is a thread



Multiple Threads

- Just as you can have several trains on the same set of track, it is also possible to have more than one thread running at once
- This is very useful, but also somewhat dangerous
- Before we consider these aspects, lets see how we make and use threads

Consider a method:

```
static void CountToTen()
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine("Loop: " + i);
        System.Threading.Thread.Sleep(500);
    }
}
```

- The method simply prints ten messages, pausing for half a second between each one

Calling the CountToTen method

```
public static void Main ()
{
    CountToTen();
}
```

- We can use the method by just calling it
- This just causes the program to go into the method, run the contents and then return
- This is how we have called methods in the past

Managing Threads

- Instead of just calling `CountToTen`, I can create a thread which runs it
- This is equivalent to making a new train and putting it onto our track
- The thread is represented by an instance of the `Thread` class, which is in the `System.Threading` namespace:

```
using System.Threading;
```

Creating a Thread

- When we create a thread we need to tell it where to start running
- We do this by telling the thread to execute a particular method
- To do this we need to have a way of representing a reference to a method
- We have already seen how to do this
- We use *delegates*

Delegates

- We first saw delegates when we looked at buttons on Windows Presentation Foundation displays
- They provide a way of telling a `Button` which method to call when the button is pressed
- In this case we are telling a `Thread` which method it is to start with
- It is a different delegate, but it does a very similar job

ThreadStart Delegate

- **`ThreadStart`** is a delegate type that refers to the method that will be called when a thread is started
- We make the delegate refer to the method we want to use
- You can think of this as determining where on the track our train is to start running

A ThreadStart for CountToTen

```
ThreadStart countStart;
countStart = new ThreadStart(CountToTen);
```

- We first declare the delegate reference (in this case `countStart`)
- We then make an instance of a `ThreadStart`
- The constructor for `ThreadStart` is given the method to be used, in this case `CountToTen`
- We now have something we can use to tell a `Thread` where to start running

Creating a Thread

```
Thread countThread;
countThread = new Thread(countStart);
```

- We first create a reference to a `Thread`
- Then we create the `Thread` itself
- The constructor of a `Thread` is given the delegate that tells it where to start running
- Note that this does not start the `Thread`, it just creates the `Thread` object

Starting a Thread Running

```
countThread.Start();
```

- The `Thread` class provides a `Start` method that is used to start the thread running
- This is the point at which the "train" is placed on the track and set running
- The `Thread` will run until its method body finishes, then it will end

Fully Threaded

```
public static void Main ()
{
      ThreadStart countStart;
      countStart = new ThreadStart(CountToTen);


      Thread countThread;
      countThread = new Thread(countStart);
      countThread.Start();
}
```

- This creates the thread and starts it

Threads and Programs

- Normally your program finishes when the `Main` method is completed
- But if your program contains threads it will only finish when the last thread ends
- This is why the previous program prints out all the numbers, even though the `Main` method completes after the call to `Start`

Aborting a Thread

```
public static void Main ()
{
      // Create the thread here

      countThread.Start();


      Console.ReadLine();    // wait for the user
      countThread.Abort();   // Abort the count thread
}
```

- If I call the `Abort` method on a `Thread` instance it causes that thread to stop
- This would cause the program above to stop as all the threads in it have finished

Threads and Program Data

- All the threads in an application “share” the same objects
- Local variables are unique to each thread
- Contents of members of classes are shared amongst threads
- This can lead to lots of problems if two threads are “fighting” over the same data

Thread Fighting

```
count = count + 1;
```

- The above code looks sensible but it is not “thread safe”
  - Thread one starts performing the increment and fetches the value of count to add one to it
  - Thread one is then suspended to make way for thread two
  - Thread two runs performs an increment of count
  - Then Thread one continues and uses its “old” value of count
  - This results in one increment operation being lost

Thread Safety

- The .NET Framework provides ways that two threads can use a synchronising object to ensure that this kind of problem can't happen
- However, programmers must use them to avoid these issues
- Bugs caused by threading mistakes are really hard to fix, because they depend on precise timing and even hardware configuration

Making Thread Safe Code

- Even if you use synchronisation you can still have problems
  - Two threads waiting for each other would be locked forever in a “deadly embrace”
- If threads are either “producers” or “consumers” there is less likelihood of problems
  - One thread creates data and another displays it

Mutual exclusion locks (Mutex)

```
object syncObject = new object();
...
lock (syncObject ) // start of synchronised block
{
    // synchronised code
}
// end of synchronised block
```

- You can create code which can only be executed by one thread at a time
- The synchronisation is managed in relation to a particular object

Threads and Applications

- Threads are used a lot in applications
  - Web servers start a thread to deal with a page request
  - Windows starts a thread to deal with each Button press
- When Windows is running there are a great many threads active
  - Many are just waiting for a trigger to act

Threads and Windows Presentation Framework

- Often you want to start a thread and have it report back to a window on the desktop
- This means that it will be changing properties on a WPF page
- Unfortunately the page is single threaded, and doesn't like you changing display elements without it knowing

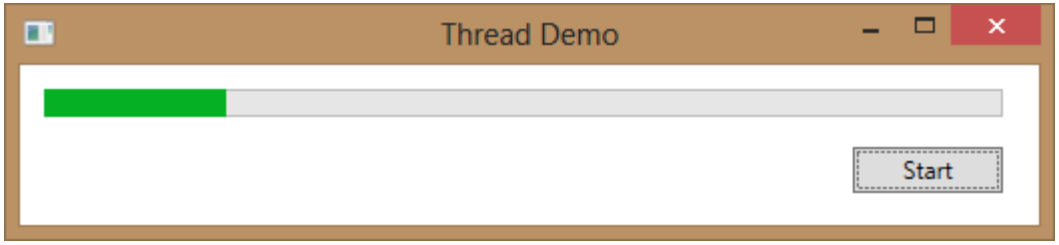
What You Want to Do

- Often a user will select an action which will take a long time to complete
  - Load a large document
  - Process lots of images
  - Create a network connection
  - Send something to a printer
- They will start the action off by pressing a button on a page

Buttons and Long Tasks

- When a user presses a button on a page the event hander for this button press should return in reasonable time
- Otherwise the Window Manager gets confused/upset
- Your application should therefore fire off a thread if the action will take a while to complete

Sample Application



- When the user presses the Start button this will fire off a thread that just makes the progress bar count up to 100
- In real-life this could load a file
- You should work like this, because button presses should return as soon as possible

Starting the Thread

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    loadThread = new Thread(new ThreadStart(loadMethod));
    loadThread.Start();
}
```

- This is an event handler for the Start button
- It creates a new load thread and starts it
  - It creates a ThreadStart delegate that refers to `loadMethod`
- The thread will perform our loading action which might take some time
- It will want to update the progress bar

Updating the Progress Bar

```
private void loadMethod()
{
    loadProgress = 0;
    while (loadProgress < 100)
    {
        loadProgress++;
        updateProgress();
        Thread.Sleep(100);
    }
}
```

- This is the `loadMethod` that I have created
- It actually does not load anything, but it counts the progress value up to 100, updating the progress bar each time

Progress Bar

```
<ProgressBar Height="14" HorizontalAlignment="Left"
Margin="12,12,0,0" Name="loadProgressBar"
VerticalAlignment="Top" Width="479" />
```

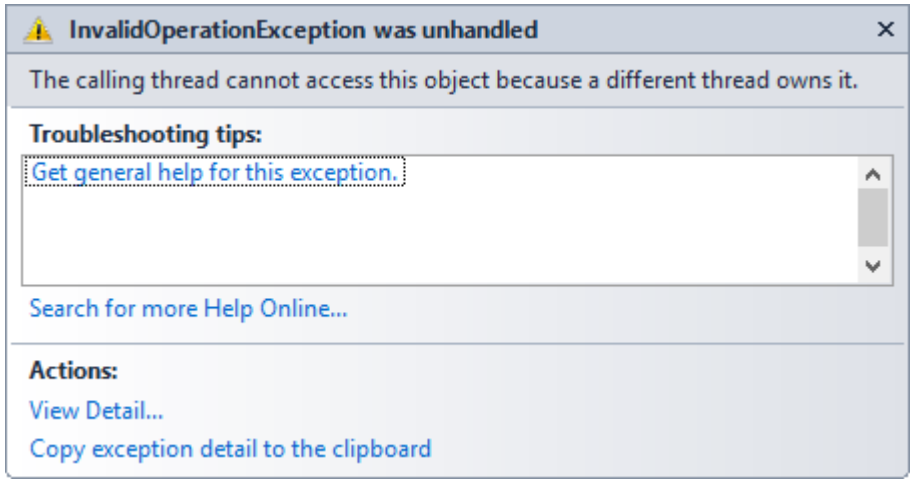
- The Progress Bar is a screen control that displays a bar of a particular length
  - You add it to a window as you would any other element
- This one is called `loadProgressBar`

Updating the Progress Bar

```
private int loadProgress = 0;

...

private void updateProgressBar()

{

    loadProgressBar.Value = loadProgress;

}
```

- You set the length displayed by setting the `Value` property in the range 0 to 100
- You can set other ranges if you need to
- The bar is automatically updated on the screen when the property is changed

Thread Safety



- Unfortunately a simple update like this will fail
- The Windows Presentation Foundation (WPF) run time system does not let other threads mess with screen components

Threads and fun and games

- If your system contains multiple threads it means that things can happen *asynchronously*
  - i.e. we can't tell exactly when, or in what sequence
- This is very bad news for the window management software
- It has to assemble a screen full of display elements and then pass that screen over to the graphics engine to be displayed
- It cannot allow changes to be made to screen components at any time, as this might corrupt the display
- So only one thread in the display engine is allowed to change the settings in display components

The Problem

- You started a thread to perform a task that would take a long time to complete
- The thread wants to update a component (the progress bar) on the page to show how it is getting on
- However, the thread is not allowed to directly manipulate display elements since only the WPF thread is allowed to do this
- We need to find a way to update the display elements at a time the WPF display thread is happy to do this

WPF Invoke Mechanism

- To get around this problem the WPF components provide a way an external thread can give a delegate to a element and say “Call this when you get round to it please”
- The element can then execute the delegate during its update behaviour on the page
- This is how we get the page to update the progress bar for us

Invoking WPF Methods

```
loadProgressBar.Dispatcher.Invoke(
    new UpdateTextCallback(this.updateProgressBar));
```

- All WPF elements provide a method called `Invoke`, that lets you ask them to run something for you in the context of the page
- You don't run the method yourself, you ask the component to run it for you
- This means that you have to provide the component with a reference to the method to be run – which means Delegates are back!

Delegate Re-Refresher

- We know what a reference is, it lets you refer to an object in memory
- A delegate is also a reference, but it refers to a method in an object
- We saw them when we used Windows Presentation Foundation, in that they are how we bind methods to events from display elements such as Buttons
- They are also how you start threads

Declaring a Delegate Type

```
delegate void SimpleMethod ();
```

- This creates a delegate that can refer to simple methods that are void and have no parameters
- This is the kind of method that we can ask a component to invoke
- Now that we have the delegate type we can create a delegate that refers to methods of that type

Creating a Delegate Variable

```
SimpleMethod barUpdate =
                new SimpleMethod(this.updateProgressBar);
```

- The variable `barUpdate` is a delegate instance that refers to the `updateProgressBar` method
- We can ask the progress bar to call this method, so that it gets run in the same Thread as the display
- Then our application will work correctly

Dispatching the update method

```
loadProgressBar.Dispatcher.Invoke(barUpdate);
```

- The variable `barUpdate` is a delegate instance that refers to the `updateProgressBar` method
- We can ask the progress bar to call this method, so that it gets run in the same Thread as the display
- Then our application will work correctly

Dispatching the update method

```
private void updateProgress()
{
    SimpleMethod barUpdate =
                        new SimpleMethod(this.updateProgressBar);
    loadProgressBar.Dispatcher.Invoke(barUpdate);
}
```

- This method will update the progress display
- It creates the delegate and then passes it to the Invoke mechanism on the progress bar

WPF Dispatcher.Invoke Method

- All Windows display elements have a `Dispatcher` property that provides an `Invoke` method
- You can use this to allow "background" threads to communicate with the user as methods executed by the `Dispatcher` run in the context of the display element
- There are also timers that you can create which will run code in the page context at regular intervals

Updating the Progress Bar during the Load

```
private void loadMethod()
{
    loadProgress = 0;
    while (loadProgress < 100)
    {
        loadProgress++;
        updateProgress();
        Thread.Sleep(100);
    }
}
```

- During the load process the program can update the progress bar as the requested task is performed
- The `loadProgress` variable is the means by which the different threads communicate

Spot the Error

```
private void startButton_Click(object sender, EventArgs e)
{
    loadThread = new Thread(new ThreadStart(loadMethod));
    loadThread.Start();
}
```

- There is a very serious error with the code above
- It doesn't cause the program to crash, but it does cause weird things to happen
- Any ideas?

Multiple Threads

```
private void startButton_Click(object sender, EventArgs e)
{
    loadThread = new Thread(new ThreadStart(loadMethod));
    loadThread.Start();
}
```

- Every time the button is pressed we get a new thread
- Repeated button presses will cause lots of threads to be created
- This causes the bar to move more quickly as each thread updates the shared progress value

Thread Management

```
private void startButton_Click(object sender, EventArgs e)
{
    if (loadThread != null)
    {
        if (loadThread.IsAlive)
        {
            return;
        }
    }
    loadThread = new Thread(new ThreadStart(loadMethod));
    loadThread.Start();
}
```

- This version checks to see if an existing thread is alive before starting a new one

Threads Summary

- A Thread is a “unit of execution” in a program
- A console application contains just one thread, which is the one that calls Main
- You can create threads of your own
- A thread is told where to start by using a delegate to refer to the method it is to run
- Threads can be controlled and synchronised
- WPF pages run on a separate thread