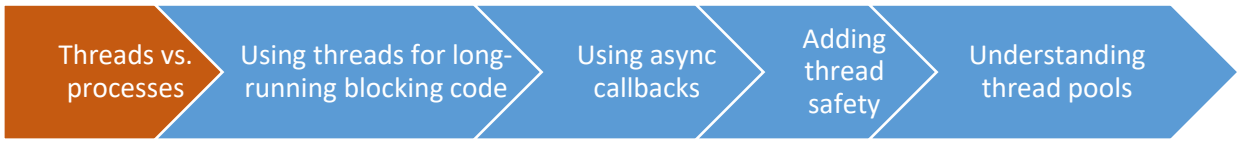
Dist. Sys. Threads

Objectives

The aim of this tutorial is to provide a practical introduction to use of multiple threads in applications to improve performance or scalability.



Introduction

Threads

Every program has one at least one thread of control. Essentially a thread of control (or thread for short) is a section of code that executes its statements one after another, independently of other threads of control, within a single program. Most of the programs you have written previously will have been single-threaded programs, with a single thread of control which starts at the first statement of the entrypoint and executes all subsequent statements one after another until the program completes. A program can have multiple threads of control which operate simultaneously. Such a program is said to be multithreaded and has the following characteristics

- Each thread begins executing at a pre-defined location and executes all subsequent code in an ordered sequence. When a statement completes, the thread always executes the next statement in sequence.
- Each thread executes its own code independently of the other threads in the program. However, threads can cooperate with each other if the programmer so chooses. We will examine various cooperation methods in later sessions.
- All the threads *appear* to execute simultaneously due to the multitasking implemented by the virtual machine. The degree of simultaneity is affected by various factors including the priority of the threads, the state of the threads and the scheduling scheme used by the Common Language Runtime (CLR).
- All the threads execute in the same virtual address space; if two threads access memory address 100 they will both be accessing the same real memory address and the data it contains.
- All the threads have access to a global variables in a program but may also define their own local variables.

Processes

Operating systems have traditionally implemented single-threaded processes which have the following characteristics

- Each process begins executing at a predefined location (usually the first statement of the entrypoint) and executes all subsequent code in an ordered sequence. When a statement completes, the process always executes the next statement in sequence.
- All processes appear to execute simultaneously due to the multitasking implemented by the operating system.

- Processes execute independently but may cooperate if the programmer so chooses using the interprocess communication mechanisms provided by the operating system.
- Each process executes in its own virtual address space; two processes which access memory address 100 will not access the same real memory address and will therefore not see the same data.
- All variables declared by a process are local to that process and not available from other processes.

Comparison of Threads and Processes

Threads and processes essentially do the same job; they are concurrency constructs that allow the programmer to construct an application with parts that execute simultaneously. If the system provides both threads and processes the programmer can use whichever he prefers to implement a concurrent application. In most applications though, threads are the preferred method of implementing concurrent activities as they impose a much lower overhead on the system during a context switch and therefore execute faster. Although systems built using processes incur greater overheads they are inherently safer. Because each process runs in its own virtual address space, a fault in one process cannot affect the state of other processes. In a multithreaded application, erroneous behaviour of one thread can cause erroneous behaviour of other threads. A single unhandled error in one of the threads may be sufficient to terminate the entire application.

.NET enables the developer to work with both threads and processes. In most circumstances we will prefer to use threads in our applications in preference to processes due to their performance advantages. We will not consider processes any further in this tutorial; for more details on using process in .NET see the Process class in the MSDN documentation.

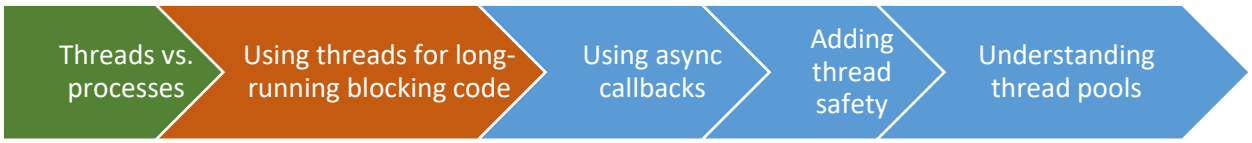
Reasons for using threads

The main reasons for using threads are as follows:

- To allow applications to process long-running tasks without stopping all other activity. For example, if an MP3 player were not multi-threaded, it would be necessary to wait until a track had finished playing before being able to press any of the control buttons[1].
- To process background tasks which never stop. Your application may have to keep monitoring the serial port for incoming data, for example
- To process tasks which happen periodically. Something as simple as repeatedly changing the colour of an icon on screen to make it look like it is flashing, benefits from a thread which can make it happen on a regular basis, regardless of what the rest of the program is doing.
- To process multiple instances of tasks, for example to allow multiple users to use a server application simultaneously.

This is not an exhaustive list, of course, and you will doubtless find others.

[1] I know, I know, you *could* write your application so that it polled the buttons periodically while doing the MP3 decompression and synthesis… but this would introduce annoying latency… and also don't be picky!



Part 1 – Dealing with slow-running tasks

To begin, you are going to create an application which performs a task requiring a lot of computation. This task takes a significant amount of time to complete.

In Visual Studio, create a new WPF Application project. Add a button and a text box called 'outputTextBox' to the main form. Add an event handler to the button's Click event, as in the example below.

```
public partial class MainWindow : Window
{
    public List<int> primeNumbers;
    public MainWindow()
    {
        InitializeComponent();
        primeNumbers = new List<int>();
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        FindPrimeNumbers(20000);
        outputTextBox.Text = primeNumbers[9999].ToString();
    }

    private void FindPrimeNumbers(int numberOfPrimesToFind)
    {
        int primeCount = 0; int currentPossiblePrime = 1;
        while (primeCount < numberOfPrimesToFind)
        {
            currentPossiblePrime++; int possibleFactor = 2; bool isPrime = true;
            while ((possibleFactor <= currentPossiblePrime / 2) && (isPrime == true))
            {
                int possibleFactor2 = currentPossiblePrime / possibleFactor;
                if (currentPossiblePrime == possibleFactor2 * possibleFactor)
                {
                    isPrime = false;
                }
                possibleFactor++;
            }
            if (isPrime)
            {
                primeCount++;
                primeNumbers.Add(currentPossiblePrime);
            }
        }
    }
}
```

This program uses a rather inefficient, brute-force method to find prime numbers, and writes the largest number it finds into the text box to show that it has finished. It's sufficiently poorly written that it takes several seconds to find the first twenty thousand primes. If your PC is super-fast, extend the time by increasing the number to find from 20000.

- When you run it, what do you observe?

For a start, there is no feedback telling you what's happening. Also, the window stops responding. You can't resize it, drag it around or even close it. That's because this is a single threaded application, and that thread is busy doing the prime number search. Until that's finished, nothing else will happen. We can improve the situation a bit by making the prime search run in a different thread.

You'll need to include a '**using System.Threading**' in your application, to indicate that you are using the threading library.

Then, modify the **buttonGo_Click** method so that it starts a new thread. In the example below, we are using a **ParameterizedThreadStart** delegate[2] because we want to specify the number of primes we wish to find; this is passed as an object argument to the **Thread.Start()** method.

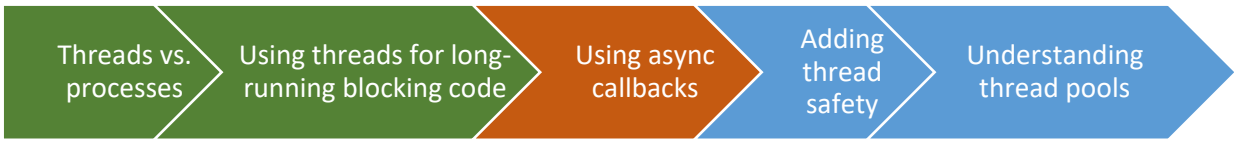
In our **FindPrimeNumbers()** method, we need to convert it back from a general **object** type to the desired **int**.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    ParameterizedThreadStart ts = new ParameterizedThreadStart(FindPrimeNumbers);
    Thread t = new Thread(ts);

    t.Start(20000);
}

private void FindPrimeNumbers(object param)
{
    int numberOfPrimesToFind = (int) param;
    // … rest of code as before …
}
```

Now, when you run the application and click the button, a new thread is created and starts executing the **FindPrimeNumbers** method. The main thread continues executing at the same time, which means that the application remains responsive – you can resize or drag the window while it is doing the calculations. There's a problem, though. The application does not tell you when it has finished. Using threads is not the same as simply calling methods – there is no return value. When the thread completes, it simply disappears. Sometimes, that's fine – but in our application, it's not.

If we want to know when a thread has finished without stopping other execution to wait for it, we have two basic options. We can set up a loop and check if the thread has finished at regular intervals, or we can arrange for the thread to signal in some way that it has completed. For that, we can use an asynchronous callback function.

[2] C# is actually clever enough to work out what sort of delegate you need here by examining the method signature; we could have simplified the code by writing **new Thread(FindPrimeNumbers)**, but for understanding it helps to spell it out rather than rely on the compiler to do everything.



2 – Asynchronous Thread Callbacks

A callback function is a method you define which the thread calls when it finishes. With a little modification to the code, you can add such a function to your application.

The code below shows a very simple method **FindPrimesFinished**, which will be called when the calculation thread completes. Notice that it must have a single parameter of type **IAsyncResult**. With that in place, it is now possible to avoid creating the thread explicitly, and simply create a **Task** which invokes the **ParameterizedThreadStart**. We can then call ContinueWith, passing a method to it. This will act as a callback – the method will fire once the task has completed but (critically) the UI stays responsive! We'll investigate tasks further in a future lab.

```
private void Button_Click(object sender, EventArgs e)
{
    ParameterizedThreadStart ts = new ParameterizedThreadStart(FindPrimeNumbers);
    Task t = Task.Run(() => ts.Invoke(20000));
    t.ContinueWith(FindPrimesFinished);}

private void FindPrimesFinished(IAsyncResult iar)
{
    System.Diagnostics.Debug.WriteLine(primeNumbers[19999]);
}
```

⚙ Make the changes to your solution.

Now when the program runs and you press the button, the program will remain responsive, and when all the primes have been found, the largest will be shown in the 'Output' window in Visual Studio.

Why don't we put its value into the text box on the form?

```
Change:
    System.Diagnostics.Debug.WriteLine(primeNumbers[19999]);

To:
    outputTextBox.Text = primeNumbers[19999].ToString();
```

➢ When you run it, what do you observe?

If you're getting an exception, don't be surprised. This catches everyone out. This happens because in Windows, the user interface and all its controls (including the text box, in our case) belong to a single thread and are **not *thread safe***. That is, you are not permitted to make changes to them from other threads (this makes a certain degree of sense, if you want to avoid having a user interface which is subject to uncontrolled change).

In order to use user interface controls from other threads, the required methods must be *invoked* on the UI thread. Here's a modified version of the code which does just that.

⚙ Make these changes now…

```
private void FindPrimesFinished(IAsyncResult iar)
{
    this.Dispatcher.Invoke(
        new Action<int>(UpdateTextBox),
        new object[] { primeNumbers[19999] });
}

private void UpdateTextBox(int number)
{
    outputTextBox.Text = number.ToString();
}
```

The Dispatcher is a queue of work to be done on a thread. When we call **this.Dispatcher** we get the Dispatcher associated with the WPF Window. Because it is associated with the Window, any work we give to this Dispatcher will be run on the UI thread.

Using the Dispatcher we can **Invoke** a method on the UI thread. We want to invoke the UpdateTextBox method but first we need to define it as an **Action** so it can be called. An action is just a generic **delegate** type (https://docs.microsoft.com/en-us/dotnet/api/system.action) and we can specify any parameters that the method requires using the syntax: Action<object, object, …>.

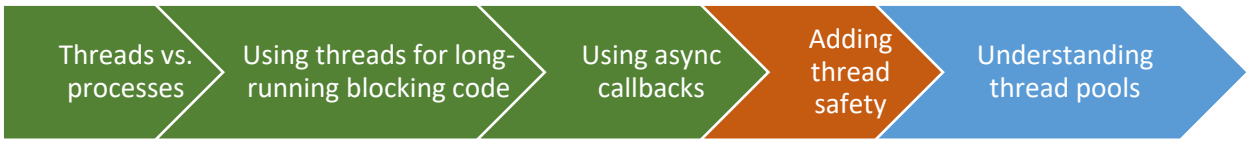
In our case UpdateTextBox takes one parameter so when we create our Action Delegate we specify only one parameter object **<int>.** Invoke uses **params** (https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params) so we also send in the int parameter as the only object in an array of object.

You may be wondering why **UpdateTextBox** just writes the 20000th prime in the text box. We don't have to wait until the end to update the text box – we now have the tools to call back from the prime search thread whenever we like. You can include the cross-thread invocation…

```
this.Dispatcher.Invoke(
        new Action<int>(UpdateTextBox),
        new object[] { currentPossiblePrime});
```

in the FindPrimeNumbers method inside `if (isPrime)`.That will give you a rolling display of the latest prime, without causing the UI to freeze.

⚙ Try it.



3 – Thread Safety

We have just experienced an exception caused due to thread safety issues. However, we are not always lucky enough to be warned that we are using threads in a way that might cause issues with our data.

In this next example we have a very simple piece of software that should generate the numbers 1-100 in an array in ascending order. However, we want this to be as quick as possible so we have decided to implement multithreading.

This is a ridiculous example, for several reasons…

- It is an extremely simple and quick task that really does not need speeding up!
- Threads have a massive startup, switching and shutdown overheads meaning that they will definitely slow down this software rather than speed it up
- You may already have determined that this task will likely fail due to the nature of the task not being a good use for threads

However, it helps prove issues around thread safety and also helps to exemplify some problems with database access that we'll look at later on.

Save your primes project to come back to later. Then, in Visual Studio, create a new solution with a Console project called ***ThreadUnsafe***.

Create a new class called ***ThreadRunner*** and give it the methods ***Run***() and ***AddValue***(***object value***):

```
class ThreadRunner
{
    public void Run() { }
    public void AddValue(object value) { }
}
```

We are creating 100 integers, and we want them to be created one after another and stored in an array. Add these member variables to the ***ThreadRunner*** class:

```
class ThreadRunner
{
    private readonly int numberToGenerate = 100;
    private int index = 0;
    private int[] orderedNumbers;

  // … rest of code as before …
}
```

Inside the ***Run*** method we will be creating our threads. We'll create 100 threads and have each one add a given number to the ***orderedNumbers*** array. The first thread will add a 1, the second thread will add a 2, etc.

We'll then start the threads and wait for them all to finish before writing out the contents of our array to the Console window.

Inside the ***Run*** method, add this code:

```
public void Run()
{
    orderedNumbers = new int[numberToGenerate];
    Thread[] threads = new Thread[numberToGenerate];

    for (int i = 0; i < numberToGenerate; i++)
    {
        ParameterizedThreadStart threadStart = new ParameterizedThreadStart(AddValue);
        Thread thread = new Thread(threadStart);
     threads[i] = thread;
    }
}
```

We are initialising our array with space for ***numberToGenerate*** (100) integers, then creating an array of threads which will help us track whether they have finished execution later.

We are then creating 100 parameterized threads (each of which will call the ***AddValue*** method) and storing them all in our ***threads*** array.

Now let's start each of our threads in turn. Add the following code at the end of the ***Run*** method, immediately after the existing for loop.

```
for (int i = 0; i < numberToGenerate; i++)
{
    threads[i].Start(i + 1);
}
```

This code simply runs through all of our 100 threads in the ***threads*** array and starts each one, passing an incrementing value to each, from 1 to 100

The next piece of code we want is to verify that the threads have finished running. There are a few ways to do this, but we'll use a basic (and not wholly efficient – but certainly good enough for our purposes) option, looping through each of the threads to see if it ***IsAlive***. If we find one that is, we go back to the beginning of our loop and try again until none are alive; then we exit the loop.

Add the following code at the end of the ***Run*** method, immediately after the existing last for loop. This is a blocking busy wait that uses CPU time. It would be better to use synchonisation mechanisms like WaitHandles, but this is simple to understand and works.

```
bool allThreadsFinished = false;
while (!allThreadsFinished)
{
    allThreadsFinished = true;
    foreach (Thread t in threads)
    {
        if (t.IsAlive)
        {
            allThreadsFinished = false;
            break;
        }
    }
}
```

Finally, for our ***Run*** method, let's write out the integers inside the ***orderedNumbers*** array to the Console window to see our lovely multithreaded output.

⚙ Add the following code at the end of the ***Run*** method:

```
foreach(int i in orderedNumbers)
{
    Console.WriteLine(i);
}
```

Now, we need to implement the ***AddValue*** method. All this code needs to do is add the value it was given as a parameter to the next position in the array. We will use ***index*** to keep track of where the next empty array position is. As each thread finishes, it will add 1 to the value of ***index*** so the next thread knows where to put its integer.

⚙ Modify the ***AddValue*** method to:

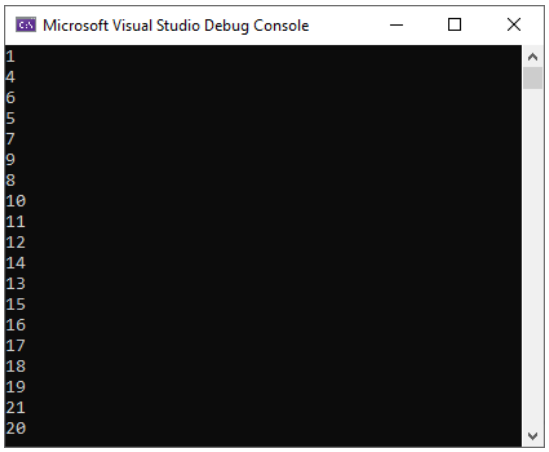
```
private void AddValue(object value)
{
    orderedNumbers[index] = (int)value;
    index++;
}
```

The last thing we need to do is create an instance of ***ThreadRunner*** from our entry point and call the ***Run*** method.

⚙ Inside the ***EntryPoint*** (in the ***Program*** class), create a new instance of ***ThreadRunner*** and call ***Run***() on this instance. You'll probably need to add a using statement for ***ThreadUnsafe***.

➢ Press F5 to run the application. What is output to the Console window?

The answer is… something like the image below; but because of the nature of threads, it will likely be different each time you run it and it will be different on different computers. This will depend on many factors including how many processors you have, how fast your processor is and how many other applications are currently running.



Try running the application a few more times to see if things change.

You may notice a few things:

- The numbers may not be ordered correctly
- There may be zeros scattered throughout the array or there may be zeros at the end of the array. This also means you have some numbers from 1-100 missing from your collection.

As we noted, this problem is very simple and, if you have a fast computer, a very slow computer or got lucky with how your threads were scheduled, it is possible that you do not see all or any of these problems – so let's make it a little more realistic.

The typical reason for using threads is where you want to perform tasks that take some time in the background, rather than allowing them to block the execution of other tasks.

Therefore, let's increase the time each thread takes to do its work – but not uniformly. Some tasks may run ever so slightly longer than others, and this is realistic in many distributed application scenarios (e.g. database access on a server is not always going to take the same amount of time).

⚙ Add a new private member variable to the ***ThreadRunner*** class, underneath the declaration of ***orderedNumbers***:

```
private Random rng = new Random();
```

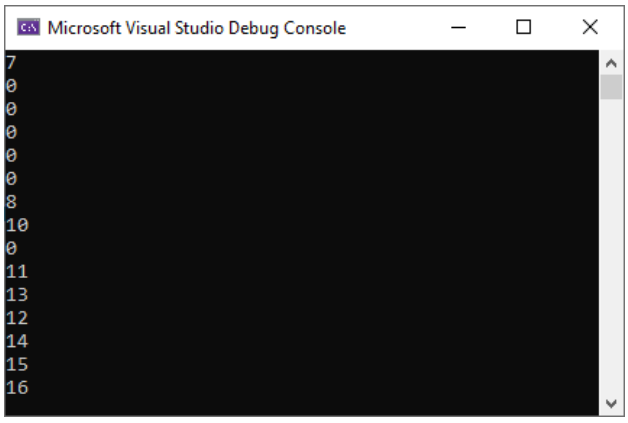
⚙ Now, immediately before the ***index++;*** line in ***AddValue***, add this line:

```
Thread.Sleep(rng.Next(6));
```

This will force the thread to wait for a random number of milliseconds, from 0-5ms before it 'finishes' its work and adds 1 to the value of ***index***. What do you think this will do to our output?

➢ Run the code to find out.

Now it is very likely you will see some serious problems. If you do not, try to increase the value in ***rng.Next()*** and run the software again.



The fact that it could have been hard to identify that there was a problem before we artificially slowed this application should tell you something important: **it can be very difficult to identify and fix errors in multithreaded applications so it's even more important to design and validate your designs before you dive into coding!**

? So why is this happening?

Inserting numbers in order is, generally, a synchronous task. We have to be *sure* that the last number was inserted before we try to inset the next. With threads, we are trying to run things at the same time, so we are likely to run into problems where things don't execute in the correct order. What makes it worse is that we are not in control of which threads get execution time and when. Therefore, we might end up with a thread trying to insert a 1 running after a thread that is trying to insert a 2.

Exacerbating this is the fact that ***index*** is being updated by all of these threads at different times too. It's quite possible that two or more threads are at the point of adding 1 to index before the next thread tries to retrieve the value of index to know where to insert its integer.

It's also possible that two or more threads access the value of ***index*** before it has been updated by another thread, accessing the same value. This will cause the last thread to finish its work to overwrite the value stored by any other thread that stored a number in the same array index location.

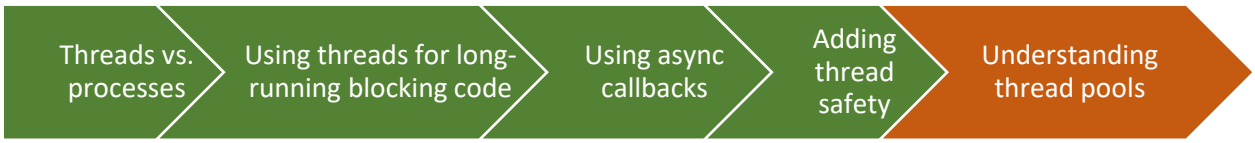
In general we need to be very careful whenever we use threads that need to:

- Pass values between the threads
- Use a common storage collection
- Be synchronised (thread A must get to a certain point in its execution before thread B performs some task)

Challenge

⚙ Can you fix this program so it adds the numbers into the correct index position in ***orderedNumbers*** based on the order that the threads were started?

There are lots of ways to do this but the best ways will not reduce the efficiency of the solution gained by multithreading it.



4 – Thread Pools

A thread is a synchronous section of code which may run concurrently with (or parallel to) other threads.

This means that you can use multiple threads to, for example…

- Keep the UI responsive whilst doing work in the background
- Split a problem into parts that can be completed at the same time, thus speeding up execution

When you create or kill a thread the operation needs to be negotiated with the Operating System

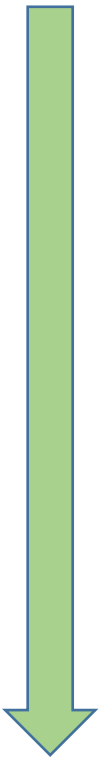
The OS does the thread scheduling and decides what the processor(s) are working on at any one time

Therefore there is lots of overhead to creating and destroying threads!

A **thread pool** is a number of threads ready to be used and then reused. Generally the number available at any one moment can scale with demand.

Number of threads in pool auto-limited per CPU. You can increase / decrease the number.

If you increase too much it can slow down execution as thread switching is still required!



	Your code requests a thread from the thread pool	
A Task object is made and returned to your code. The request is queued and control is handed back to your main code		
Thread pool manager checks queue and assigns the work to a specific thread if one is available in the pool	Your code keeps running. It can keep tabs on the thread by checking the status of the Task. . . .	
		Thread runs . Thread exits
When the thread finishes running, the thread is recycled and returned to the pool as 'available'.		

When to use the ThreadPool

Generally, the threadpool is a better place for short-running (short-lived) background work.

- Little start-up time (thread already exists ready for use).
- Work will be finished reasonably quickly, thus freeing up the thread for other work.
- The threadpool is *only* for background work – if your application is quit, they will immediately be killed.

When to make your own Thread

Creating your own threads is better if you…

- Have long-running work to do
- Doesn't tie up a threadpool thread for a long time.
- Startup overhead is minimal expense for work that will take a long time anyway.
- Want to do foreground work
- The idea here is that there is some work that should be forced to go ahead even if the user quits the application. With a foreground thread you can ensure that the work completes (e.g. a save).

5 - Further Work

Primes Search

1. Modify your primes search so that it uses more than one thread to do the searching. There are a number of ways to do this. One obvious one is to split the search space into parts, and use a different thread to do each part. Use the **Stopwatch** class to time your code, and see if there is there any advantage to using multiple threads for the purpose. SearchSpaceSplitExample inside the MultithreadExample.zip might give you some pointers on this.

2. Make the user interface more informative. Put a progress bar on it, and make it update at regular intervals (not for every prime number). Add a stop button, so that the prime search can be terminated early if desired. Work out how to make the thread(s) stop gracefully.

Thread Safety

3. You should have already sorted the insertion so that it inserts sequentially. Now change the blocking wait to use WaitHandles or another synchronisation mechanism.

   Read more about WaitHandles here:
   https://learn.microsoft.com/en-us/dotnet/api/system.threading.waithandle

Next

Review the MultithreadExample Solution. Some of this will also be useful for next week's workshop.

- SearchSpaceSplitExample shows some ideas about splitting a search space for making use of threads.
- CancellationExample investigates the use of cancellation tokens
- ThreadVsThreadpool explores the differences between threads and the use of the threadpool.

1. Use what you have learned to add threading to the server you created in the first workshop so that multiple clients can connect at the same time.

2. Add a long-running problem to the server (e.g. a prime number search) and split the search space across multiple threads using the thread pool.

3. Include cancellation so a client can cancel their search.

Outcome

By the end of this workshop, you should have created a program which searches for prime numbers using multiple threads to make it faster. Your program should have a user interface which can control the process without being slowed down by the background computation.

You should have also examined issues with thread synchronisation and thread unsafe code.

In a future workshop we'll look at using the async and await operators for running non-blocking operations on the same thread, and we'll look at Tasks in more detail too!