Laboratory 7: Threads & Concurrency

Objectives

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

We'll also look at some problems with concurrent data access and implement an optimistic control mechanism for managing concurrent update errors.

Prerequisites

Ideally, you should have read the accompanying lecture notes on concurrency and thread programming before attempting this tutorial.

Books and More Information

Threading in C#
http://www.albahari.info/threading/threading.pdf

Async in C#:
http://www.tahlildadeh.com/Files/Async%20in%20Csharp%205.pdf

Entity Framework and DB Techniques:
http://sd.blackball.lv/library/Modern_Data_Access_with_Entity_Framework_Core_(2018).pdf

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 Main() method 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 Main()) 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 2 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.

The .NET Framework and Compact Framework enable 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. 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 (on my old PC, at least) to find the first twenty thousand primes. To extend the time increase 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 Tasks

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 call **BeginInvoke** on the **ParameterizedThreadStart** object, passing it a new **AsyncCallback** delegate

```
private void buttonGo_Click(object sender, EventArgs e)
{
    ParameterizedThreadStart ts = new ParameterizedThreadStart(FindPrimeNumbers);
    ts.BeginInvoke(10000,new AsyncCallback(FindPrimesFinished),null);
}

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

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:

    Console.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:

```
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** doesn't just write the 20000th prime in the text box. The reason is that 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 – Optimistic Concurrency in Databases

We know by now that Entity Framework will handle race conditions and helps our databases to manage concurrent connections. However, this doesn't prevent us from accidentally getting into a lost update, inconsistent retrieval or dirty read scenario (see the slides – Lecture 15).

We have two options available to us: introduce pessimistic concurrency control (through the use of locking in our code) or modify our database to allow us to use optimistic concurrency control. It often makes sense to allow optimistic concurrency control – especially where the likelihood of a conflict is low.

Entity Framework is designed to work well with optimistic concurrency control but we still have to allow for it in our database tables.

We'll be basing this on the Entity Framework code we wrote in Lab 3 but there is a skeleton solution for you to download if you lost that code. We'll be using some principles discussed in Lab 3 though so make sure you've done that lab already.

Download the Lab 7 Skeleton Solution from Canvas.

Open the NuGet Package Manager and install Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Tools and Microsoft.EntityFrameworkCore.SqlServer if they aren't already installed

Open the Package Manager Console window and type: Update-Database

Look at the database and solution. The code/database links Person, Address and BankAccount so that Addresses can have many People and each person can have one BankAccount.

Open Program.cs and note the similarities from Lab3, and the addition of a BankAccount.

Press F5 to run the program. It lay take some time to load, then run.

Once execution is finished your database should now be populated with the details of 'Jane Doe'. We are going to be using Jane's table entries but we don't want to create any more copies of this data in the database so comment out the entire using block in Program.cs

What we're going to do next is access Janes BankAccount twice, simultaneously trying to update its balance – but, in order to force the worst case scenario, we're going to manipulate it manually by using two processes.

Add a new using block to the Main method:

```
using (var ctx = new Lab7Context())
{ }
```

We'll be using Linq so add: `using System.Linq;` to your using statements

Inside your new using block, insert the following code:

```
        Person prsn = ctx.People.First();
        decimal balance = prsn.BankAccount.Balance;

        decimal balancechange = 0;

        do
        {
            Console.WriteLine("Enter a balance modifier");
        }
        while (!decimal.TryParse(Console.ReadLine(), out balancechange));

        balance += balancechange;

        prsn.BankAccount.Balance = balance;
        ctx.SaveChanges();
```

This code takes the first person in your database and loads up their balance. It then asks the user to give a balance modifier, applies this modifier to the balance and finally updates the database.

Press f5 to test your code is working

You should see an exception. Look at the locals for prsn - you should see that Address and BankAccount are both **Null**. If you examine the data in the database you'll see the relationships exist there! What do you think happened?

The problem is that Entity Framework hasn't loaded in the references for the linked objects. You are effectively working with Lazy Loading turned off. We have a few ways to solve this.

1. Explicit loading

After the line: `Person prsn = ctx.People.First();` add the new line:
`ctx.Entry(prsn).Reference("BankAccount").Load();`

Now place a breakpoint on the line: `decimal balancechange = 0;`
and run the code with f5

When the breakpoint is hit, examine the balance variable – it should be 50. Now look at the prsn object. It should have BankAccount loaded but Addresses not loaded. You have explicitly loaded the BankAccount reference using the line you added.

2. Lazy Loading

Let’s turn Lazy loading on.

Open the NuGet Package Manager and install Microsoft.EntityFrameworkCore.Proxies

Inside Lab7Context.cs find the OnConfiguring method and add, at the top of the method, the line: `optionsBuilder.UseLazyLoadingProxies();`

Remove the new line you added previously:
`ctx.Entry(prsn).Reference("BankAccount").Load();`

Open Person.cs and add the `virtual` keyword before the `Address` and `BankAccount` properties. Open Address.cs and do the same before `People`.

This allows Entity Framework to override your properties at runtime, thus giving you access to objects through the Lazy Loading mechanism.

Ensure you still have a breakpoint on the line: `decimal balancechange = 0;`
and run the code with f5

When the breakpoint is hit, examine the balance variable – it should be 50. Now look at the prsn object. It should have BankAccount **and** Addresses loaded. You haven’t had to explicitly load the BankAccount reference – Entity Framework has done this for you when you got the entry.

Remove the breakpoint and press f5 again to continue past it.

When prompted, enter -20 into the console window and hit enter. The app should finish execution after a few moments.

This is good… and bad.

It’s easy, and just works out of the box… but we have loaded Address and we aren’t actually going to use it, so it was a wasteful load.

If we had lots of data associated with a Person that we didn’t actually need to load, we have now automatically loaded all of it when we loaded in the Person - which is likely to have taken quite a while and used up machine resources that could have been used for other things.

Now look at your BankAccount database table.

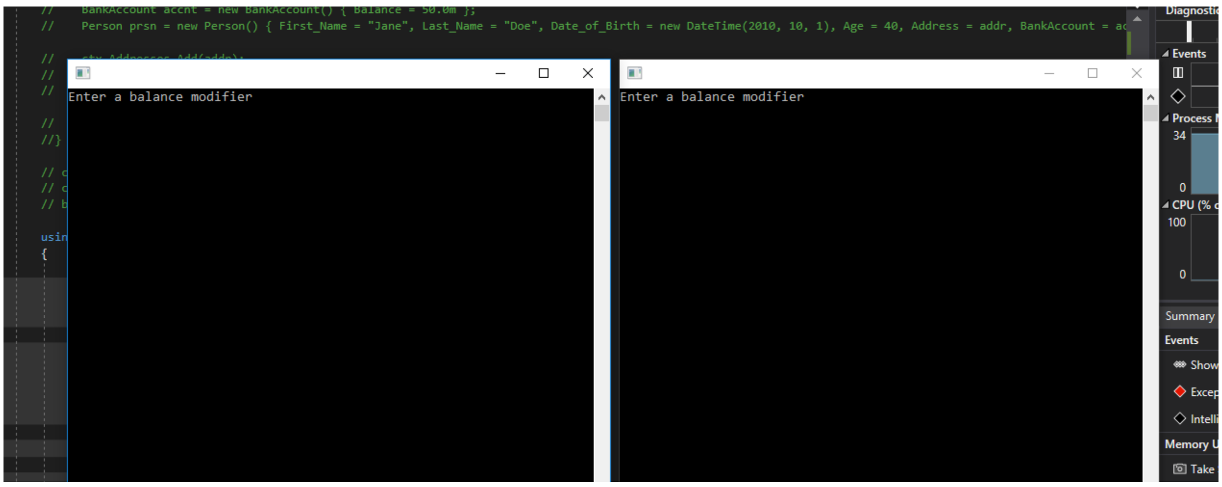
Our code should have subtracted 20 from the balance, resulting in a balance of 30

Now we’ll simulate two concurrent requests:

Right-Click on the project, and then click Debug->Start New Instance to run the code.

Once an instance is running, do the same thing again to start another instance.

You should now have two consoles open, side by side, looking like this:



```
//    BankAccount accnt = new BankAccount() { Balance = 50.0m };
//    Person prsn = new Person() { First_Name = "Jane", Last_Name = "Doe", Date_of_Birth = new DateTime(2010, 10, 1), Age = 40, Address = addr, BankAccount = ac
```

```
Enter a balance modifier
```

```
Enter a balance modifier
```

Enter 60 into one console window (and press enter)

Then enter -70 into the other console window (and press enter)

As our original balance was 30 (was 50-20), in theory this should perform the calculation:

30 + 60 - 70 = 20

Open up your BankAccount database table (SQL Server Object Explorer) and examine the Balance. It probably reads -40.00

Can you identify why?

The key thing here is that our calls were offset. If our application was multithreaded, this could actually have occurred on different concurrent threads and it may be very difficult to diagnose.

The process was:

1. Variable a = balance from database = 30
2. Variable b = balance from database = 30

3. a + 60 = 90
4. b - 70 = -40

5. Save a to database - Balance = 90
6. Save b to database - Balance = -40

We need a concurrency control mechanism and, as we've established, Entity Framework is set up to work best with Optimistic Concurrency Control.

However, in order to set it up, we have to make some changes to our code.

Open Person.cs and add the lines:

```
[Timestamp]
public byte[] RowVersion { get; set; }
```

Then do the same in Address.cs and BankAccount.cs

The Timestamp attribute classifies this parameter (column in the db) as a row version identifier. This allows Entity Framework to automatically begin using this parameter to identify if conflicts have occurred.

Because we've now changed our data model we'll need to update the database…

Add a new migration.

Check the migration code is adding three new rows, one in each table.

Update the database to the new migration

Now we'll simulate two concurrent requests again:

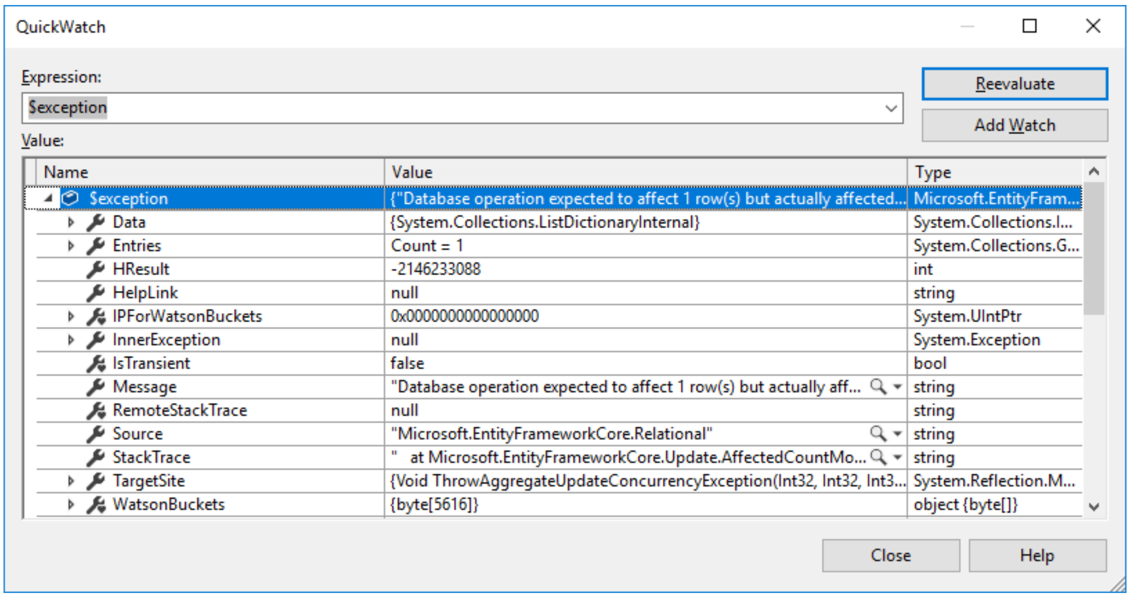
Right-Click on the project, and then click Debug->Start New Instance to run the code.

Once an instance is running, do the same thing again to start another instance.

In one console window type 90 (and press enter).

Then enter -50 into the other console window (and press enter)

You should now be staring at a nifty little exception – and (unusually) you should actually be very happy to see this exception. It should look something like this:



Expression: $exception

Value:

Name	Value	Type
$exception	{"Database operation expected to affect 1 row(s) but actually affected...	Microsoft.EntityFram...
Data	{System.Collections.ListDictionaryInternal}	System.Collections.I...
Entries	Count = 1	System.Collections.G...
HResult	-2146233088	int
HelpLink	null	string
IPForWatsonBuckets	0x0000000000000000	System.UIntPtr
InnerException	null	System.Exception
IsTransient	false	bool
Message	"Database operation expected to affect 1 row(s) but actually aff...	string
RemoteStackTrace	null	string
Source	"Microsoft.EntityFrameworkCore.Relational"	string
StackTrace	" at Microsoft.EntityFrameworkCore.Update.AffectedCountMo...	string
TargetSite	{Void ThrowAggregateUpdateConcurrencyException(Int32, Int32, Int3...	System.Reflection.M...
WatsonBuckets	{byte[5616]}	object {byte[]}

A quick read through of the exception identifies a few things that are useful to us.

1. The exception is a **ConcurrencyException**. It tells us that there has been a problem with the database operation and that the database didn’t affect any rows. Of particular importance is the line: “Data may have been modified or deleted since entities were loaded”.
2. Specifically, the exception is a DbUpdateConcurrencyException. This allows us to identify that this problem occurred explicitly when an update was attempted.

Under the hood, EntityFramework is using our RowVersion column… whenever an update is made to the row it checks the RowVersion hasn’t changed since the read was done and, if it hasn’t changed it automatically updates the byte array in the RowVersion column to a new value.

If an update is attempted but the RowVersion has changed since a read was done this exception is thrown to prevent data errors. We can catch the exception to allow recovery. In this case we’ll throw it back to the client but what you do in other applications is up to you…

Replace the entire using block in your main method with this code:

```
for (int i = 0; i < 3; i++)
{
    try
    {
        using (var ctx = new Lab7Context())
        {
            Person prsn = ctx.People.First();
            decimal balance = prsn.BankAccount.Balance;
            decimal balancechange;
            do
            {
                Console.WriteLine("Enter a balance modifier");
            }
            while (!decimal.TryParse(Console.ReadLine(), out balancechange));
            balance += balancechange;
            prsn.BankAccount.Balance = balance;
            ctx.SaveChanges();
            return;
        }
   }
   catch(DbUpdateConcurrencyException)
   {
        Console.WriteLine("Oh no! Looks like the database was modified whilst you
were making your change. Try again.");
   }
}
Console.WriteLine("Data access failed three times - perhaps try again later.");
```

This code will loop through three times, giving the user 3 attempts to manipulate the balance. If all three attempts fail (cause: update concurrency exceptions) it gives up. As soon as one succeeds it returns out of the Main method and closes the application.

Let's test if it works:

We'll simulate two concurrent requests again:

Right-Click on the project, and then click Debug->Start New Instance to run the code.

Once an instance is running, do the same thing again to start another instance.

In one console window type 20 (and press enter).

Then enter 30 into the other console window (and press enter)

You should be prompted to try again so…
Re-enter 30 into the remaining console window (and press enter)

Check your database. Your balance should be 100 (50 + 20 + 30 = 100).
If so, you are successfully using optimistic concurrency control.

4 – Further Work

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.

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.

3. Explore **Tasks** and **async**. There is a great resource here explaining the basics: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ and a good book identified on page 1 of this lab.

Self evaluation

By the end of this lab, 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 also have created an application, using Entity Framework with lazy loading and optimistic concurrency control turned on!