Objectives

The aim of this tutorial is to provide a practical introduction to the threading, concurrency control and asynchronous facilities of .NET and how these can be used to create responsive mobile client applications.

Prerequisites

You should have read the accompanying lecture notes on concurrency and threading before attempting this tutorial.

Lab Setup

No special configuration is required to undertake these exercises.

For More Information

There are several .NET and C# books available which cover threading and concurrency. The following titles are directly relevant but there are also others that you could consult.


The MSDN documentation, Visual Studio help files and .NET framework SDK also contain a large amount of information and code samples on the various methods of creating and controlling threads.

Introduction

In this tutorial we will see how to explicitly and implicitly create and use threads in .NET applications. We will also consider the various mechanisms available for controlling concurrency, how these can be applied to ensure the logical correctness of our computations and problems that can arise from their use.

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. Such programs consist of 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. However it is possible for a program to 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)\(^1\).

- 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.

\(^1\) The CLR uses the thread scheduling mechanism of the Windows operating system on which it is running. The .NET Framework runs on Windows 32 whilst the .NET Compact Framework runs on Windows CE. Although both of these operating systems are called Windows, they are distinct operating systems and Windows CE uses an different scheduling system. In some cases this may cause differences in threading behaviour in managed code run on these two platforms.
**Processes**

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

- Each process begins executing at a pre-defined 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 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.

---

2 Although threads and processes appear to execute simultaneously, on a single CPU machine they actually execute in turn in a round-robin fashion. Each thread is given a *timeslice*, a short period of time during which it will execute on the CPU. When the timeslice expires, the thread will be removed from the CPU and the scheduler will choose another thread to execute. This operation is known as a *context switch* and takes a finite amount of time. A context switch between threads imposes a lower overhead than a context switch between processes.
Concurrenty And Asynchronous Operation

Asynchronous Operation

Many of the operations we perform in the code that we write are synchronous. Method calls are a good example of a synchronous operation. When we call a method, control passes to the called method and does not return to the caller until the method call completes. The caller is thus blocked and cannot continue until the operation it initiated (in this case, the method call) completes.

Asynchronous operations are also possible and are supported by many of the classes in the .NET Framework and .NET Compact Framework libraries, particularly those involving I/O and networking (such methods are easily recognized as their names begin with Begin or End). In an asynchronous operation, the caller initiates an operation and control immediately returns to the caller which can then carry on with other processing. Meanwhile, the operation it initiated executes in the background and will continue until it completes. Note that in this mode of operation the caller is not blocked and can carry on doing other things whilst the asynchronous operation is carried out. This is extremely useful in client applications with a user interface, such as those found in mobile applications. We want the user to be able to initiate actions through the user interface, but we also want the interface and application to remain responsive whilst those actions are performed. For example, we may have a mobile email client application which needs to retrieve the latest messages from the network. This is a slow operation and whilst it proceeds, the user may want to do other things with the application, such as compose new email messages.

Delegates and Asynchronous Operation

.NET does not offer direct support for asynchronous method calls but it does allow us to invoke a delegate asynchronously. A delegate is simply an instance of the System.Delegate class, or one of its sub-classes, and is really just a type safe function pointer.

The Delegate class has an Invoke() method, and by calling this, the method that the Delegate instance refers to is invoked. The Delegate class also has a BeginInvoke() and EndInvoke() methods which are used to asynchronously invoke the method referenced by the delegate. The BeginInvoke() method starts the referenced method execute and returns an instance of the IAsyncResult class. This object is passed to the EndInvoke() method, which simply waits until the asynchronous call completes and can be used to retrieve any results that it generated. If the asynchronous call has not completed when EndInvoke() is called, the EndInvoke() operation will block until the asynchronous operation completes.

Instead of using EndInvoke() to wait for an asynchronous operation to complete, which detracts from the potential benefits of asynchronism, we can use a callback instead. A callback is just a method which we declare that gets called when a particular operation completes. We can create a callback for an asynchronous delegate invocation by creating an
Concurrent and Asynchronous Operation

An instance which refers to the method that we want to be invoked when the asynchronous delegate invocation completes. We pass the IAsyncResult object for the asynchronous operation. We can use this in the code of our callback method to retrieve the results of the asynchronous call using EndInvoke(). This time, we know that the asynchronous call will have completed when we call EndInvoke(); if it hadn’t, our callback method wouldn’t have been called.

We could use asynchronous delegates with a callback to implement the mobile e-mail client application. Suppose there is a method in the e-mail client that performs the actions necessary to retrieve messages from the network server. When the application starts it can create a delegate to this method and invoke it asynchronously, passing in a reference to a callback method that should be invoked when the operation completes. This callback method can simply retrieve the new messages and display information about them in the Inbox folder in the user interface, although this introduces some issues concerning Windows forms and concurrency which we will discuss at the end of this tutorial.

If this all sounds rather complicated and involved, don’t worry. We can make life easier for ourselves and still use some forms of asynchronous behaviour if we choose to use Web services. When a Web reference is added to a project, Visual Studio creates a class which is a proxy to the Web service. This proxy class contains methods with the same signatures as those tagged with the [WebMethod] attribute in the Web service. In addition to this, the proxy also contains a Begin<MethodName> and an End<MethodName> method for every Web method in the Web service. These Begin and End methods are used to invoke the Web service asynchronously, in the same way that the BeginInvoke() and EndInvoke() methods of the Delegate class are used. It is also possible to declare a callback and pass this in to the Begin method. When the asynchronous Web service call completes, the callback method will be invoked and passed an IAsyncResult instance. This can be used to retrieve any values returned by the Web service call. Figure 1 shows the code that might be used in our hypothetical mobile e-mail application to retrieve messages from the server using a Web service. The MailServer class is the proxy instance created by Visual Studio. The Web service has a GetMessages() method, so the MailServer proxy class has GetMessages(), BeginGetMessages() and EndGetMessages() methods. The sample code uses the BeginGetMessages() and EndGetMessages(), together with a callback, to asynchronously retrieve new e-mail messages from the server.

One-way methods provide another way to invoke Web services asynchronously, but this is only really useful if the service does not return a

---

3 IAsyncResult is a sub-class of System.AsyncCallback and is simply a function pointer to a method which accepts a single parameter of type IAsyncResult
value (i.e. the service is being used to update data stored on the server). A one-way method is specified in the Web service by tagging it with the [OneWay] attribute. Any method tagged in this fashion will always be invoked asynchronously by the client and there is no need to use the Begin/End methods for the method to achieve asynchronous operation. Note that the method must have a void return type as it is not possible to return a value from a one-way method. One advantage of one-way calls is that nothing is ever returned from the call to the client. If the call fails for some reason no exception is passed back to the client. This is usually the desired behaviour if the client does not understand the significance of the exception or can do nothing to correct it.

```csharp
using System;

namespace AsyncMailClientSample
{
    public class MailClient
    {
        private MailServer svr;
        public static void Main()
        {
            // Create an instance of the Web service proxy
            svr = new MailServer();

            // Call BeginGetMessages to asynchronously invoke
            // the WebService
            AsyncCallback cb = new AsyncCallback(GetMsgCallback);
            svr.BeginGetMessages(cb, null);

            // Do something else whilst call proceeds
        }

        // Callback method for asynchronous call to
        // Web service
        private static void GetMsgCallback(IAsyncResult ar)
        {
            Object[] msgs = svr.EndGetMessages(ar);
            // Add messages to Inbox
        }
    }
}
```

**Figure 1**: Asynchronous invocation of a Web service

**Threading in .NET**

The .NET environment provides several ways for programmers to make use of threads. Some of the classes in the .NET Framework automatically create and use threads to implement their operations and by using these features the programmer will implicitly make use of threading. All of the asynchronous invocation features provided by .NET use implicit threading and although the programmer does not need to do anything to create or control the threads, special consideration may need to be given to how these operations are used correctly in the application. The programmer can also
explicitly create and control threads using the classes in the *System.Threading* namespace.

The *System.Threading* namespace provides the programmer with all the facilities necessary to create and manage threads within an application. The most important types in this namespace are the *Thread* type and the *ThreadStart* type. *ThreadStart* is a delegate type which is used to define the method which will be used as the body of the thread. The *Thread* type contains methods and properties which allow threads to be created, destroyed and controlled. For our purposes the most important of these are

- **Thread(ThreadStart t)**
  Constructs a new thread using the code specified by the given ThreadStart delegate as the body of the thread.

- **void Start()**
  Causes the operating system to change the state of the thread to ThreadState.Running. Once a thread is in the Running state, the operating system can schedule it for execution. Once the thread terminates, it cannot be restarted with another call to Start().

- **void Abort()**
  Raises a *ThreadAbortException*, which begins the process of terminating the thread. Not supported by .NET Compact Framework

- **static void Sleep(int millis)**
  Blocks the current thread for *at least* the specified number of milliseconds. The thread does not use any CPU cycles whilst it is blocked.

- **void Suspend()**
  Prevents the thread from running for an indefinite amount of time. The thread will start running again only after a call to Resume() is made. Not supported by .NET Compact Framework

- **void Resume()**
  Returns a suspended thread back to a running state. Not supported by .NET Compact Framework

- **int Priority**
  Gets or sets the scheduling priority of the thread, which affects the relative order in which threads are chosen for execution by the operating system scheduler.

- **string Name**
  Gets or sets a textual name for a thread. This is useful when debugging as the name of the thread appears in the debugger’s Threads window

- **static Thread CurrentThread**

---

4 The body of a thread is the sequence of statements that it will execute.

5 The Sleep() method does not guarantee that the thread will stop executing for exactly the number of milliseconds specified, merely that it will be returned to the Running state after this time. There may be another delay before the scheduler chooses to execute the thread again.
Returns the Thread object that represents the current thread of execution.

A Simple Multithreaded Application

Consider the program in Figure 2 which creates a new thread and starts it executing. The thread uses the HelloTask() method as its body and simply displays “Hello World” on the screen. When this application executes there are actually two threads; the application thread that was started with the Main() method and the new thread we have created.

A thread will terminate when it reaches the end of the sequence of statements that make up its thread body. The thread based on the HelloTask() method will terminate when it reaches the end of the HelloTask() method. This also applies to the main application thread; this will terminate when it reaches the end of the Main() method.

Using Visual Studio, create a C# console application. Enter the code from Figure 2 into this project. Build and run the project. Does it behave as you expect? You should see the phrase Hello World appearing on screen before the program terminates. It may be easier to observe this if you run the compiled application from the command line rather than from Visual Studio.

Modify the HelloTask() method by adding a call to Thread.Sleep before the statement that prints out Hello World. The Thread.Sleep call should put the thread to sleep for 10 seconds. Modify the Main() method by adding a call to print out the message Hello from Main. Build and run your application again. What do you expect to happen and in what order do you expect the statements to appear? You should find that the Hello from Main message appears on the console first and the Hello World message appears about 10 seconds later. This happens because the Main() method is running in a separate thread. When the HelloWorld thread is blocked by the Thread.Sleep call, the operating system schedules the main application thread for execution and it will print the Hello from Main message. Later on the HelloWorld thread is reactivated and prints out the HelloWorld message.

Create a new C# Console Application. Using Figure 2 as a starting point, create an application which creates three new threads. The body of each thread should consist of an infinite loop which prints out a word to the console. One thread should print out the word “Hello”, the second should print out “World” and the third should print “Again”. Compile and run your program. In what order do the words appear on screen and what does suggest about the order in which the threads execute? Run your program several times; do the threads always execute in the same order?

```csharp
using System;
using System.Threading;

namespace SimpleThreadSample1
{
    class ThreadSample
    {
```
static void Main(string[] args)
{
    ThreadSample s = new ThreadSample();

    // Create the thread
    ThreadStart tBody = new ThreadStart(s.HelloTask);
    Thread t = new Thread(tBody);

    // Start the thread
    t.Start();
}

public void HelloTask()
{
    Console.WriteLine("Hello World!\n");
}

Figure 2: Creating and starting a thread

Non-Determinism in Concurrent Programs

The program that you created in the preceding exercise was a multithreaded program with three threads of control. When you ran your programs several times you may have noticed that the words “Hello”, “Again” and “World” sometimes appeared in different orders. This happens because the scheduler multitasks between the three threads. When the timeslice for one thread expires, the scheduler must pick another thread to execute for a while. The order in which threads are picked depends upon a number of factors and is non-deterministic – that is, it cannot be predicted. Because the order in which the threads are executed is non-deterministic, the order in which the words appear on the screen is also non-deterministic and may change from one run to the next. You will also notice that words appear as a group of one word, followed by a group of another word, and so on. Words appear in groups because each thread executes several cycles of its loop in each timeslice.

This non-determinism is one of the problems that can make concurrent programming so tricky. We often want to have some control over the order in which things happen within our program but if the scheduler executes threads in an unpredictable manner how can we make sure that things happen in the correct sequence? For example, we may want our three threads to always print the words in the order “Hello” “Again” “World”.

Thread Priorities

One method of ordering the operation of threads is to prioritise the threads and make some more important than others. The more important, or higher priority threads, will run in preference to the less important, lower priority threads.

The priority of a .NET thread can be set using the Thread.Priority property and must be set to one of the constants from the
System.Threading.ThreadPriority enumeration. Five distinct thread priorities exist; Highest, AboveNormal, Normal, BelowNormal, Lowest. If there are any threads with Highest priority in the runnable state then the scheduler will execute these first. Any runnable threads with Lowest priority will only be scheduled when there are no more runnable higher priority threads. All threads are initially created with Normal priority. This is sufficient for most purposes but we may change the priority of a thread if required to impose some relative ordering on the execution of a set of threads. Note that we cannot dictate the order in which threads of the same priority will execute; all we can do is force higher priority threads to be scheduled in preference to lower priority threads.

Modify your solution to the previous exercise so that the threads do not run in an infinite loop but only cycle round each loop 100 times. Use the Thread.Priority property to set the priority of the "Hello" thread to AboveNormal and the priority of the "Again" thread to BelowNormal. The priority of the World thread will remain at its default Normal priority. Compile and run your code. What order do you expect the words to appear in now? As the thread which prints "Hello" has the highest priority it should print first. "World" should be printed next as its thread has the next highest priority and "Again" should appear last. Is this what happens? If not, can you explain why not? (HINT: The .NET console class is designed so that it can only be accessed by one thread at a time. Consider this question again when you have attempted the exercises on locking later in this tutorial)

This technique of using thread priorities to order actions is often used in real-time systems and is known as priority-based scheduling. However, it is not sufficient to control the order in which threads of the same priority are scheduled and for this we must rely on the concurrency control mechanism provided by the environment.

**Mutual Exclusion and Locking**

We have seen that a program can have multiple threads which operate simultaneously. Each of these threads can access and use objects in the program and several threads can operate on the same object at the same time. Sometimes though it is not sensible, or even desirable, to allow multiple threads to simultaneously operate on the same object or resource. For example, consider the .NET console class which supports console output. It does not make sense to allow two threads to write to the console at the same time as their output could be intermixed with unreadable results. We would like to ensure that the Console can only be accessed by one thread at a time and this is an example of the need for *mutual exclusion*.

A critical section is a block of code which, for whatever reason, must be executed under mutual exclusion (i.e. by one thread at a time). We can create a critical section and ensure mutually exclusion using locks. A lock is a programming construct which is used to guarantee mutual exclusion and is essentially an implicit variable associated with an object. Every object has a unique implicit lock variable associated with it. A critical section is created by defining a block of code; any thread which wishes to use the code in the
critical must attempt to acquire a lock before entering the critical section. If several threads attempt to acquire a lock simultaneously only one will succeed and enter the critical section. The remaining threads will be blocked and will remain blocked until the successful thread releases the lock as it leaves the critical section. At this point the blocked threads will be reactivated and will compete again to acquire the lock.

The C# lock construct is used to protect a block of code. Any thread attempting to enter this block of code must first attempt to acquire the lock on the object specified by the lock statement. Once successful, the thread can enter the associated code block, at which point it will be executing this block of code under mutual exclusion. The code block protected by the lock statement is thus a critical section. Figure 3 provides an outline example that illustrates the use of the lock statement. Typically, the lock statement uses the this pointer to specify that the calling thread must obtain the lock on the current object, although any object can be used. Sometimes it may be necessary to use nested lock statements if mutually exclusive access to several objects is required to complete an operation.

```csharp
public someMethod {
    lock (this) {
        // Critical section – any thread executing the code in
        // this block will do so under mutual exclusion
    }
}
```

Figure 3: Using the lock construct to protect a critical section

We shall use the scenario of reading and updating the balance in a bank account for the next few examples in this tutorial. Updating a bank balance is an operation that must be performed under mutual exclusion if the current balance of the account is to remain correct. Figure 4 provides a simple Account class which has an Update() method that allows the account balance to be modified.
public class Account
{
    private float _balance = -100;

    public float Balance
    {
        return _balance;
    }

    public void Update(float deltaAmount)
    {
        _balance += deltaAmount;
        Console.WriteLine("Account balance updated");
        Thread.Sleep(2000);
    }
}

Figure 4: The Account class

class AccountSample
{
    Account acc = new Account();

    public ReaderTask
    {
        string msg = "Thread" + Thread.CurrentThread.Name + ": Balance = ";
        msg += acc.Balance;
        Console.WriteLine(msg);
    }

    public WriterTask
    {
        acc.Update(10);
        Thread.Sleep(5000);
    }

    public static void Main()
    {
        AccountSample as = new AccountSample();
        Thread tReaderA = new Thread(new ThreadStart(sa.ReaderTask));
        tReaderA.Name = "A";
        Thread tReaderB = new Thread(new ThreadStart(sa.ReaderTask));
        tReaderB.Name = "B";
        Thread tWriter = new Thread(new ThreadStart(sa.WriterTask));
        tReaderA.Start();
        tReaderB.Start();
        tWriter.Start();
    }
}

Figure 5: The AccountSample class
Using Visual Studio, create a new C# console application. Add the Account class from Figure 4 to this project. Add the AccountSample class from Figure 5 to the project; this class contains two methods which are used as the bodies of three threads created in the Main() method. Two of these threads execute an infinite loop which simply displays the account balance whilst the third thread executes a loop which continually updates the account balance. Build and execute the application and observe the output. Does the account balance increment correctly? Note how the Thread.Name property is used in the DisplayBalance() method that forms the display thread bodies to customize the message displayed by each thread, allowing us to use one method to implement the bodies of several threads.

Using the sample code in Figure 3 as a template, modify the Update() method and Balance properties of the Account class so that calling threads must obtain a lock on the Account object before updating or retrieving the account balance. Build and execute your code; does the account balance increment correctly now? In this example we have two critical sections (the Balance property and the Update method) and whenever a thread is executing one of these critical sections no other thread can be executing either of the critical sections, ensuring that the Balance property and Update methods always execute under mutual exclusion.

Too much mutual exclusion can be detrimental. One of the reasons for using multiple threads is to maximize performance by allowing multiple operations to proceed at the same time. Using mutual exclusion forces operations to occur one after another, thus reducing the potential concurrency. Often all that we really need is to ensure that operations which update or write values are performed under mutual exclusion whilst operations that simply read values may be performed simultaneously. This cannot be achieved using the C# lock construct, but the ReaderWriterLock class provided by .NET can. Figure 6 shows an example of the usage of the ReaderWriterLock class to create and protect critical sections. A ReaderWriterLock uses two distinct locks; one for reading and one for writing. A thread which wants to read a value must first acquire a reader lock. Multiple Reader locks can be granted simultaneously, allowing concurrent read operations to be performed. A thread wanting to write a value must first acquire a writer lock. The writer lock will only be granted when there are no outstanding read requests; when it is granted, the calling thread is guaranteed mutually exclusive access. No reader locks will be granted until the writing thread releases the writer lock. As with the lock construct, any thread which cannot acquire the lock it requires is blocked until it can successfully acquire the lock.

Modify the Account class to introduce a ReaderWriter lock. The Balance property should be modified to acquire a Reader lock before entering the critical section whilst the Update() method should be modified to acquire a Writer lock before entering its critical section. This should ensure that updates are performed under mutual exclusion whilst balance retrievals are allowed to proceed concurrently. Build and execute your code and verify that the account balance is always modified correctly.
public class ReaderWriterSample
{
    Object obj = new Object();
    ReaderWriterLock rwl = new ReaderWriterLock();
    int timeout = 20;

    public void ReaderTask()
    {
        try
        {
            rwl.AcquireReaderLock(timeout);
            // Do a reading task that does
            // not require mutual exclusion
        }
        catch (ApplicationException)
        {
            // AcquireReaderLock timed out
        }
    }

    public void WriterTask()
    {
        try
        {
            rwl.AcquireWriterLock(timeout);
            // Do a writing task that requires
            // mutual exclusion
        }
        catch (ApplicationException)
        {
            // AcquireReaderLock timed out
        }
    }
}

Figure 6: Using a ReaderWriter lock

Monitors

A monitor is an alternative, more advanced concurrency control construct provided by .NET. Whereas locks simply provide mutual exclusion, a monitor can provide mutual exclusion and condition synchronization, thus enabling us to impose a relative ordering on the activities of multiple threads.

Consider the example of a buffer of a fixed capacity and two threads, a Producer thread which inserts items in the buffer and a Consumer thread which removes items from the buffer. The Producer must not attempt to insert items into the buffer if it is full; the Consumer must not attempt to remove items from the buffer if it is empty. We can use condition synchronization to ensure that both of these criteria are fulfilled. The condition synchronization also imposes a relative ordering on the activities of the Produced and Consumer threads; we know that can only retrieve a value from the buffer after the Producer thread has inserted a value into it.

.NET Monitors are implemented by the Monitor class. The Monitor.Enter() method is used to enter the monitor; Monitor.Exit() is used to leave the
Concurrency And Asynchronous Operation

A thread which successfully executes Monitor.Enter() is guaranteed to be operating under mutual exclusion and critical sections of code are wrapped in a pair of Monitor.Enter() and Monitor.Exit() calls. If multiple threads call Monitor.Enter() simultaneously, only one will succeed and enter the monitor, the other threads being blocked until the successful thread leaves the monitor using Monitor.Exit(). At this point the other threads will compete again to enter the monitor.

Condition synchronization relies on the use of condition variables. Essentially every object can be assumed to have an implicit condition variable. The Monitor class provides three operations Wait(), Pulse() and PulseAll() which operate on these implicit condition variables and must be supplied with an object reference when they are invoked. The three operations can only be called by a thread which has successfully entered the monitor. The Wait() operation causes the calling thread to block on the implicit condition variable associated with the specified object; when the thread blocks it releases its mutually exclusive hold on the monitor. The Pulse() operation wakes up one of the threads which have been blocked by a prior call to Wait() on the condition variable associated with the specified object. The object which is woken then automatically reacquires the mutually exclusive lock on the monitor and can carry on its operations. If there are no waiting threads then the Pulse() call has no effect. The PulseAll() operation wakes all threads blocked by a prior call to Wait(). These threads then compete with each other to reacquire the mutually exclusive lock on the monitor and the successful one can carry on its operations whilst the unsuccessful ones return to the blocked state. The code sample in Figure 7 illustrates the use of monitors.

Modify the Account class by removing the ReaderWriterLock. Modify the critical sections in the Balance property and Update() method so that they are each wrapped in calls to Monitor.Enter() and Monitor.Exit(). Build and execute your code to satisfy yourself that the monitor does indeed provide mutual exclusion in the same way as a lock does.

Now modify the Balance property so that it will only return a value when the account balance is greater than zero; if the account balance is less than or equal to zero the call to Balance should block until the account balance exceeds zero. This can be achieved by using condition synchronization. The critical section of the Balance property should be modified so that if the account balance is less than zero, a call to Wait() is executed on the Account object. The critical section of the Update() method should be modified so that a call to Pulse() on the Account object is executed if the balance after the update exceeds zero. Build and execute your code. Satisfy yourself that the balance is being updated correctly and that the display thread only display balance information when the balance is greater than zero.
class BoundedBuffer
{
    // Dummy object for signalling space available
    private object SpaceAvailable = new object();
    // Dummy object for signalling items available
    private object ItemAvailable = new object();

    private const int capacity = 5;
    private object[] buffer = new object[capacity];
    private int itemsInBuffer = 0;
    private int writePos = 0;
    private int readPos = 0;

    public void Append(object obj)
    {
        Monitor.Enter(this);
        if (itemsInBuffer == capacity)
        {
            // Buffer is full, wait for space
            Monitor.Wait(SpaceAvailable);
        }
        buffer[writePos] = obj;
        writePos = writePos + 1 % capacity;
        itemsInBuffer++;
        // Notify waiting threads; item added
        Monitor.Pulse(ItemAvailable);
        Monitor.Exit(this);
    }

    public object Take()
    {
        object val = null;
        Monitor.Enter(this);
        if (itemsInBuffer == 0)
        {
            // Buffer empty, wait for items
            Monitor.Wait(ItemAvailable);
        }
        val = buffer[readPos];
        readPos = readPos + 1 % capacity;
        itemsInBuffer--;
        // Notify waiting threads
        Monitor.Pulse(SpaceAvailable);
        Monitor.Exit(this);
        return val;
    }
}

Figure 7: Implementing a bounded buffer using monitors

Windows Forms and Multithreading

Multithreading can be extremely beneficial when creating user interfaces, particularly for resource limited mobile devices. When the user initiates an action using the user interface controls, this action may take some time to
Concurrency And Asynchronous Operation

complete perhaps because it is a long, intensive computation or because it
uses resources which introduce delays e.g. the network. Whilst this action is
carried out, the user would like the user interface to remain responsive and
react to other commands that they might issue. This is where multithreading
can help; a separate thread can be created to perform long running actions,
or actions that result in latency. The user interface executes on its own
separate thread and can continue to respond to the user whilst a previous
action is in progress. This also allows us to provide mechanisms in the user
interface for cancelling actions that are in progress. An action can easily be
cancelled by aborting the thread performing that action, although careful
consideration needs to be given to any undesirable state changes which the
action may have already taken.

When an action completes, there is often a need to update the user interface
and display some information as a result of that action. The thread
performing the action needs to be able to modify user interface controls but
this is not straightforward with Windows Forms. Windows Forms controls are not thread safe; they cannot be safely used by multiple threads and the
only thread which can safely operate on a control is the thread which created
that control. User interface controls are typically created by the user
interface thread and this which means that other threads cannot then work
directly with these controls. .NET provides a mechanism to get around this
restriction. Every Windows Form control has an Invoke() method which is
passed a delegate. The purpose of the Invoke() method is to get the method
referenced by the delegate to execute on the thread which initially created
the control. The delegate must be an instance of the MethodInvoker type
and the method it references is used to update the state of a control. The
typical use of the Invoke() method is a scenario in which the user interface
controls are created by the user interface thread; a separate thread is created
to perform an action and, when this action completes, that thread calls
Invoke() on the control it wishes to update. The delegate it passes in
references a method which updates the control based on the results of the
action that was executed and this update is then performed by the user
interface thread which owns the control.

Using Visual Studio, create a new C# Smart Device Application targeted at
the Pocket PC Emulator. The main form of the application should contain a
single menu called Options which has three items: Start, Stop and Change
Message. The Stop menu item should initially be disabled.

- Add a Label called labelMessage to your form. This label should be 240
  pixels wide and 16 pixels high and should be located at (0, 48)
- Add a private string member to your form class called msgText and
  set it to contain the text "Hello World"
- Create a method in your form class called ScrollMessage(). This
  method should contain an infinite loop which is used to modify the
  position of the lblMessage label. This is done by adding 2 to the
  current X position of the label. This can be achieved by using the
  label's Location property to retrieve a Point object describing its
  current position. The X property of the Point instance can be modified
  and assigned back to the Location property of the label. If the X
Concurrent and Asynchronous Operation

When the position becomes greater than 240 it should be set back to zero so that the message wraps around. After the label position has been modified, Thread.Sleep should be called to make the current thread pause for 50 milliseconds.

- Add a handler for the Start menu item's Click event. This should simply call the ScrollMessage() method.

Build your application and make sure it executes correctly. Your application should look something similar to Figure 8.

![Figure 8: Scrolling message application](image)

The example you have created in the preceding exercise is simple but it serves to illustrate some of the issues which must be considered when using multithreading with user interfaces. Try the following exercises.

The user would like to be able to stop the message scrolling at any point or change the text displayed by the label.

- Modify the event handler for the Start menu item, so that it creates and starts a new thread which uses the ScrollMessage() method for its thread body. Once the thread has been started the Start menu item should be disabled and the Stop menu item should be enabled.

- Add an event handler for the Stop menu item, so that it stops the thread which is modifying the position of the lblMessage label. Note that the Compact Framework does not support the Thread.Abort() method so you must find some other method of stopping the thread. **HINT:** A thread will terminate when it reaches the end of the method that defines its body. Try using a Boolean flag in the loop in ScrollMessage method to control whether the loop should continue or not.

The code you have added should enable the user to stop the label scrolling at any point. Build and execute your code. Does it execute correctly?

You should find that your application runs as expected but it is not correct as it violates the guidelines for the use of Windows Forms controls in the presence of multiple threads. The thread which updates the position of lblMessage is a different thread to that which created lblMessage and should not directly operate on the label. To overcome this issue correctly and safely we must use the Invoke() method of lblMessage, passing it a delegate to a
Concurrency And Asynchronous Operation

method which will update the position of the label for us and which will execute in the context of the application thread which initially created the label. In the .NET Framework this delegate can be a delegate to any type which has a void parameter list, a MethodInvoker delegate instance or an EventHandler delegate instance. In the .NET Compact Framework this delegate **must** be an EventHandler delegate instance and the method to which it refers must have a signature that matches that of the EventHandler type.

*Create a SetMessagePosition() method which two parameters of type object and EventArgs respectively. These parameters are not used in the method. Move the code which modifies the position of lblMessage from the ScrollMessage() method into the SetMessagePosition() method.*

*Modify the ScrollMessage() method so that it declares an instance of the EventHandler type which refers to the SetMessagePosition() method. This should be done before the loop. The code inside the loop should be modified so that it simply calls lblMessage.Invoke(), passing in the reference to the EventHandler delegate instance, then sleeps for 30 milliseconds, as before.*

The code you have added uses the lblMessage.Invoke() method to execute the SetMessagePosition() method in the context of the application thread which initially created lblMessage. This is allowable as only the owning thread is now updating the lblMessage control. Build and execute your code and verify that the message scrolls correctly when the Start menu item is clicked and stops when the Stop menu item is clicked.

Now add a Click event handler for the *Change Message* menu item. This must change the message displayed by the scrolling label as soon as this menu item is selected. Ideally the message should change to the next message in a set of predefined messages each time the *Change Message* menu item is selected. Because the Change Message event handler executes on the application thread it can simply change the Text property of lblMessage to the correct text for the new message.

An alternative approach, which is extremely useful when working with threaded user interfaces, is to make use of the powerful data binding facilities offered by .NET. Suppose we wish to change the behaviour of the application so that the message displayed by the scrolling label automatically changes every 2 seconds. In the *Start* menu item Click event handler we could create another thread which simply sleeps for 10 seconds then changes message. This thread cannot directly alter the lblMessage.Text property as this violates the threading guidelines. Instead of using the lblMessage.Invoke() method as before, we can use data binding, as shown in the following exercise.

*Add the following code to the constructor of your form*

```csharp
msgData = new MessageData();
lblMessage.DataBindings.Add("Text", msgData, "Text");
```
Now add the class shown Figure 9 to your project. Add a member variable of this type called msgData to the form class. This class simply stores a set of messages as text strings and has a Text property which returns a message string. The NextMessage() method causes subsequent calls to the Text property to return the next message in the set of stored messages. The code added to the constructor simply creates an instance of this class and binds its Text property to the Text property of the lblMessage label.

In the Click event handler of the Start menu item create a new thread which simply iterates through an infinite loop controlled by a Boolean flag. Inside the loop Thread.Sleep should be used to delay for 2 seconds before calling msgData.NextMessage() to cause the message to change to the next in the set of messages.

Modify the Click event handler for the Change Message menu item so that it calls msgData.NextMessage() to change the message.

Build and execute your code and test that it behaves as you expect.

If you examine the code of the MessageData class you will see that it defines an EventHandler member called TextChanged. This is required by the Windows Forms data binding mechanism. When a property of an object is bound to a control, the data binding mechanism looks for an EventHandler with the same name as the property plus Changed (in our case, TextChanged). If such an event handler exists then the data binding mechanism binds this event handler to the control; this causes the control to be updated using the value from the bound property whenever the event is raised. The private OnTextChanged() method is used to raise the event when the message changes. Note also that the NextMessage() method uses locking to prevent the ChangeMessage thread and the user from trying to call NextMessage() simultaneously.
Concurrent and Asynchronous Operation

public class MessageData
{
    private ArrayList _msgData = new ArrayList();
    private int _msgCount = 0;

    public event EventHandler TextChanged;

    public MessageData()
    {
        _msgData.Add("Hello World");
        _msgData.Add("Bye World");
        _msgData.Add("Buy Fish");
    }

    protected virtual void OnTextChanged()
    {
        if (TextChanged != null)
            TextChanged( this, EventArgs.Empty);
    }

    public void NextMessage()
    {
        lock(this)
        {
            _msgCount = ++_msgCount % _msgData.Count;
            OnTextChanged();
        }
    }

    public string Text
    {
        get
        {
            return (string)_msgData[_msgCount];
        }
    }
}

Figure 9: MessageData class used to implement data binding

Why Use Concurrency?

Although we have only looked at simple example programs, the use of concurrency and threading is essential in many applications, particularly in real-time and embedded systems. There are three main reasons for using concurrency:

- **Implementation of nonblocking I/O**

  In many languages, when you attempt to read information from some I/O device the program will wait until data is available before it carries on and executes subsequent statements. This is known as blocking I/O and it is often not the type of behaviour that is required within an application. If a program blocks when attempting to read from an I/O device then it cannot do anything else until data becomes available. Typical techniques for overcoming the limitations of blocking I/O are I/O multiplexing, polling and the use of signals. An alternative approach is to use a separate thread to read data from the I/O device. If the data is not available the thread

...
blocks but the rest of the program can carry on performing useful work. .NET provides asynchronous (nonblocking) I/O operations to the programmer and internally these are implemented using separate threads. If you use an asynchronous operation in a .NET program then you are implicitly making use of threads.

- **Alarms and Timers**
  
  Threads may be used to implement timer functions within a program. Programs, particularly those with time constraints, often set a timer and continue processing. When the timer expires the program is sent an asynchronous notification of this and may use this information to undertake error recovery or alternative actions. A separate thread can be set up to implement a timer by making the thread sleep for the timer duration. When the thread reawakens it can notify other threads that the timer has expired. .NET provides the Timer class which implements exactly this functionality for the developer’s convenience.

- **Independent Tasks**
  
  Sometimes a program is required to simultaneously perform independent tasks. For example, a Web server may simultaneously service independent requests from several clients, or a computer control system may need to manipulate several control parameters at the same time. Although a single-threaded program could be written to implement such systems it is often easier to consider the concurrency in the problem and use a separate thread for each independent task.

**Concurrency Problems**

The use of threading is not without its problems and it is normally harder to write and debug multithreaded programs. Consider the pseudo-code example in Figure 10. When this program is executed both Thread A and Thread B will enter an infinite loop. Thread A waits for FlagB to become false, which it never does, whilst Thread B similarly waits for FlagA to become false. The problem occurs because each thread waits on variables manipulated by the other thread. This sharing of variables creates inter-dependence between the threads which results in both threads entering a state from which neither can proceed. This situation is known as **livelock** and it is one of the problems that can occur in an application which uses concurrency. Although this is a trivial example, it is all too easy to achieve livelock in a program that uses many threads.

In livelock the threads cannot make progress but are still scheduled and perform some execution on the CPU. **Deadlock** is a similar condition in which threads cannot make progress and are blocked, thus preventing them from executing on the CPU. Deadlock often occurs when multiple locks are required before an action can be performed. If more than one critical section requires this same set of locks, but the set of locks are acquired in a different
order then deadlock can result. This is illustrated in Figure 10. Deadlock can be avoided by always acquiring and releasing locks in the same order and for only holding a lock for the absolute minimum amount of time required.

```java
FlagA = false;
FlagB = false;

Thread A
{
    while(FlagB == false) {
        // do nothing
    }
    // Do something useful
}

Thread B
{
    while(FlagA == false) {
        // do nothing
    }
    // Do something useful
}
```

**Figure 10: Livelock in a multithreaded application**

```java
object A;
object B;

Thread A
{
    lock(A)
    {
        lock(B)
        {
            // do something
        }
    }
}

Thread B
{
    lock(B)
    {
        lock(A)
        {
            // do something
        }
    }
}
```

**Figure 11: Deadlock in a multithreaded application**