SHRI GURU RAM RAI INSTITUTE OF TECHNOLOGY AND SCIENCE

CONCURRENCY AND SYNCHRONIZATION IN C#.NET

Concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other.

The Problem "Concurrency"

When you build a multithreaded application, your program needs to ensure that shared data should be protected from the possibility of multiple thread engagement with its value. What's going to happen if multiple threads were accessing the data at the same point? The CLR can suspend any thread for a while who's going to update the value or is in the middle of updating the value and at same time a thread comes to read that value which is not completely updated, that thread is reading an uncompleted/unstable data.

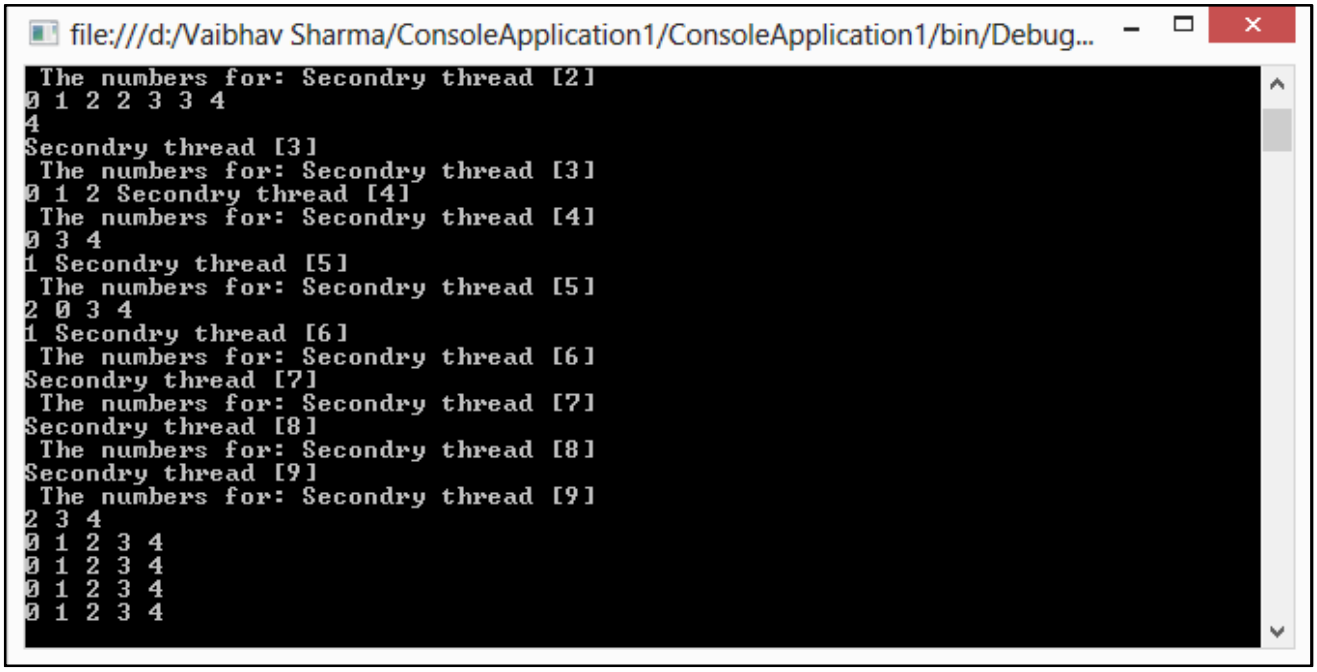
In the example below , the main thread spawning 5 secondary threads. Each secondary thread is told to make calls on the task() method on the same DisplayThread instance. Because we're not doing anything to grab the object's shared resources (here, it is the console), there is a pretty good chance that current thread can't get to the console before the task() method is able to print out the numbers. We're bound to get unpredictable results because we don't know exactly when we're going to be kicked out of the way. Note that task() method will force the current thread to pause for a randomly generate amount of time:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace Concurrency2
```

```
{
   public class DisplayThread
   {
      public void task()
      {
         Console.WriteLine(Thread.CurrentThread.Name);
         Console.WriteLine(" The numbers for: {0}",
               Thread.CurrentThread.Name);
         for (int i = 0; i < 5; i++)
         {
            Random r = new Random();
            Thread.Sleep(1000 * r.Next(2));
            Console.Write("{0} ", i);
         }
         Console.WriteLine();
      }
      // 5 Thread objects.
      // each object call the same instance of the DisplayThread object.
      public static void Main()
      { // make instance
         DisplayThread dt = new DisplayThread();
         // 5 threads that are all pointing to the same method on the same object
         Thread[] t = new Thread[10];
         for (int i = 0; i < 10; i++)
         {
            t[i] = new Thread(new ThreadStart(dt.task));
            t[i].Name = string.Format("Secondry thread [{0}]", i);
         }
         foreach (Thread ts in t) ts.Start();
         Console.ReadLine();
}}}
```

OUTPUT



Since we got inconsistent result, we need to find a way to programmatically enforce synchronized access to the shared resources. The System.Threading provides a number of synchronization types as we'll see in the following sections.

Synchronization of threads solution to the problem of concurrency

LOCK

lock -->The lock keyword marks a statement block as a critical section by obtaining the mutual-exclusion lock for a given object, executing a statement, and then releasing the lock. The following example includes a lock statement.

The lock keyword ensures that one thread does not enter a critical section of code while another thread is in the critical section. If another thread tries to enter a locked code, it will wait, block, until the object is released.

lock(expression) statement_block
Where:
expression

Specifies the object that you want to lock on. expression must be a reference type

statement _block

The statements of critical section.

In this section, we're going to use lock keyword for the synchronized access to shared resources. The lock allows us to define a scope of code that must be synchronized between threads so that incoming threads cannot interrupt the current thread. To use the lock, we need to specify a token that must be acquired by a thread to enter inside the scope which has been locked. So, when we want to lock down a private instance method, we can simply pass in an object reference to the current type:

```
private void MethodToLock()
{
        lock(this)
        {
                // The code here is thread-safe
        }
```

```
}
```

But, if we want to lock down a region of code within a public member, it is safer to declare a private object member to serve as the lock token:

```
public class DisplayThread
{
        // lock token
        private object Obj = new object();
        public void task()
        {
                // using the lock token
                lock(Obj)
                {
                        ...
                }
        }
}
```

Here is the code using the lock keyword for synchronization.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace Concurrency2
{
   public class DisplayThread
   {
```

```
private object Obj = new object();
public void task()
{
   lock(Obj)
   {
      Console.WriteLine ();
      Console.WriteLine(Thread.CurrentThread.Name);
      Console.WriteLine(" The numbers for: {0}",
         Thread.CurrentThread.Name);
   for (int i = 0; i < 5; i++)
   {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(2));
      Console.Write("{0} ", i);
   }
   Console.WriteLine();
   }
}

// 5 Thread objects.
// each object call the same instance of the DisplayThread object.
public static void Main()
{
   // make instance
   DisplayThread dt = new DisplayThread();

   // 5 threads that are all pointing to the same method on the same object
   Thread[] t = new Thread[10];
   for (int i = 0; i < 10; i++)
   {
      t[i] = new Thread(new ThreadStart(dt.task));
```

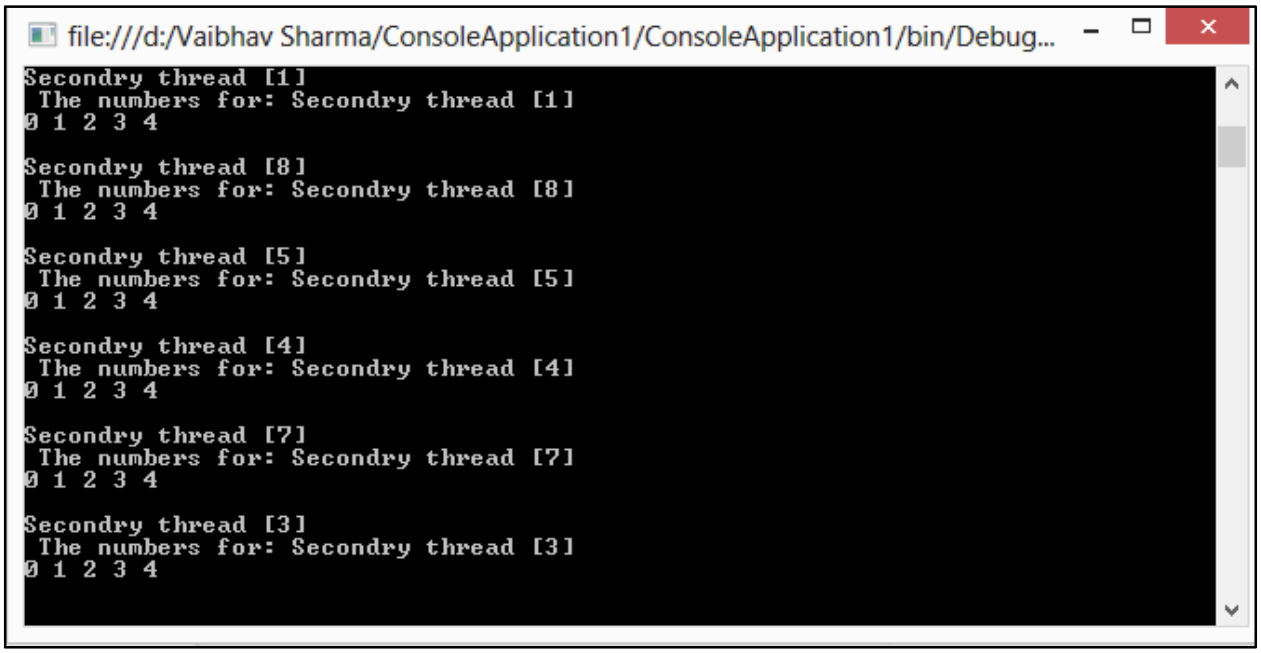
```
            t[i].Name = string.Format("Secondry thread [{0}]", i);
        }

        foreach (Thread ts in t) ts.Start();

        Console.ReadLine();
    }
  }

}
```

OUTPUT

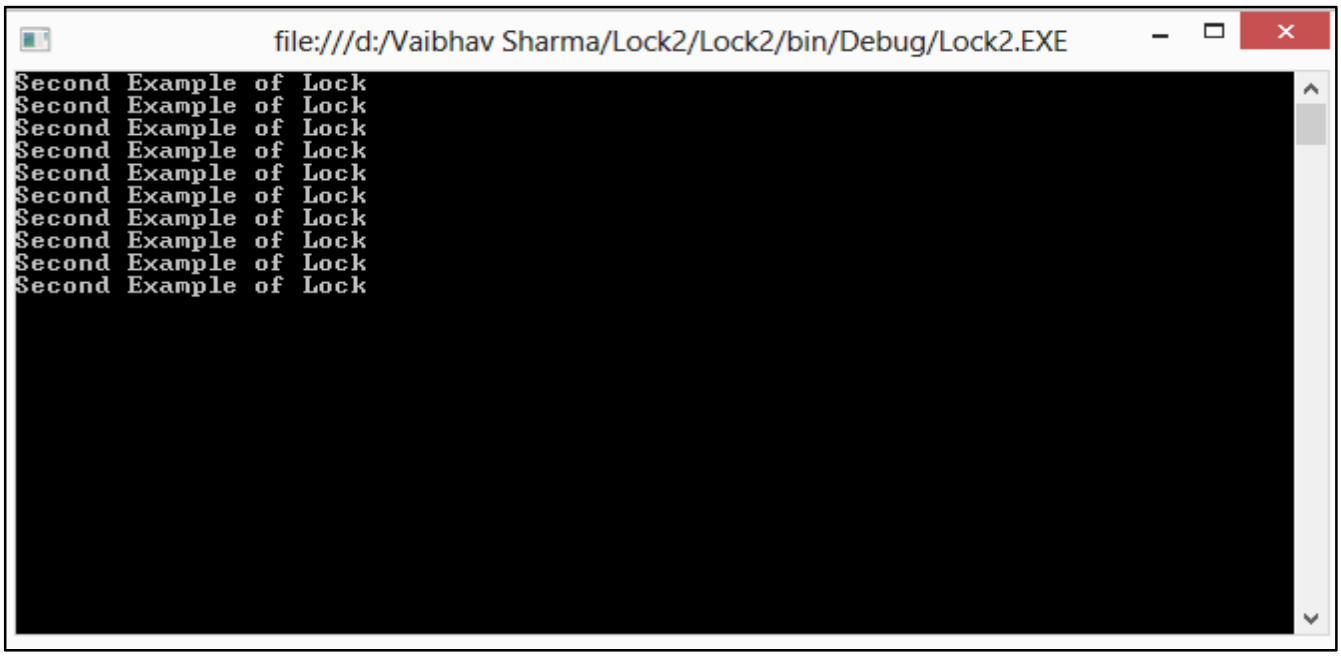


Another example of lock synchronization

```
using System;
using System.Threading;

class Program
{
   static readonly object _obj = new object();
   static void A()
   {
      // Lock on the readonly object.
      // ... Inside the lock, sleep for 100 milliseconds.
      // ... This is thread serialization.
      lock (_obj)
      {
         Thread.Sleep(100);
         Console.WriteLine("Second Example of Lock");
      }
   }
   static void Main()
   {
      // Create ten new threads.
      for (int i = 0; i < 10; i++)
      {
         Thread T = new Thread(new ThreadStart(A));
         T.Start();
      }
      Console.ReadLine();
   }
}
```

OUTPUT



MONITORS

Threading with Monitor

A monitor is a mechanism for ensuring that only one thread at a time may be running a certain piece of code (critical section). A monitor has a lock, and only one thread at a time may acquire it. To run in certain blocks of code, a thread must have acquired the monitor. A monitor is always associated with a specific object and cannot be dissociated from or replaced within that object.

Monitor has the following features:

- It is associated with an object on demand.
- It is unbound, which means it can be called directly from any context.
- An instance of the Monitor class cannot be created.

The following information is maintained for each synchronized object:

- A reference to the thread that currently holds the lock.
- A reference to a ready queue, which contains the threads that are ready to obtain the lock.
- A reference to a waiting queue, which contains the threads that are waiting for notification of a change in the state of the locked object.

Monitor Class

The Monitor Class Provides a mechanism that synchronizes access to objects. The Monitor class is a collection of static methods that provides access to the monitor associated with a particular object, which is specified through the method's first argument. the class provide following method.

- Monitor.Enter() : Acquires an exclusive lock on the specified object. This action also marks the beginning of a critical section.
- Monitor.Exit() : Releases an exclusive lock on the specified object. This action also marks the end of a critical section protected by the locked object.
- Monitor.Pules() : Notifies a thread in the waiting queue of a change in the locked object's state.
- Monitor.Wait() : Releases the lock on an object and blocks the current thread until it reacquires the lock.
- Monitor.PulesAll() : Notifies all waiting threads of a change in the object's state.
- Monitor.TryEnter() : Attempts to acquire an exclusive lock on the specified object.

Example of Synchronization-Monitor

```
using System;
using System.Threading;
class Program
{
   static readonly object _obj = new object();
   static void A()
   {
      Monitor.Enter(_obj);
      try
      {
         Thread.Sleep(100);
         Console.WriteLine("Example of Monitor");
      }
      finally
      {
         Monitor.Exit(_obj);
      }
   }
   static void Main()
   {
      // Create ten new threads.
      for (int i = 0; i < 10; i++)
      {
         Thread T = new Thread(new ThreadStart(A));
         T.Start();
      }
      Console.ReadLine();
   }
}
```

OUTPUT

SYNCHRONIZATION - MUTEX

A Mutex is similar to the lock, however, it can work across multiple processes. But it is slower than the lock.

Mutex allows us to call the WaitOne() method to lock and ReleaseMutex() to unlock. Note that a Mutex can be released only from the same thread which obtained it. The primary use for a cross-process Mutex is to ensure that only one instance of a program can run at a time. Here is the source code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace Mutex1
{
   class Program
   {
      private static Mutex m = new Mutex();
      private const int num = 1;
      private const int numth = 4;
      private static void A()
      {
         for (int i = 0; i < num; i++)
         {
            hello();
         }
      }
      private static void hello()
      {
         m.WaitOne();
         Console.WriteLine("{0} hi", Thread.CurrentThread.Name);
         Thread.Sleep(1000);
         Console.WriteLine("{0} hello", Thread.CurrentThread.Name);
         m.ReleaseMutex();
      }
      static void Main(string[] args)
      {
         for (int i = 0; i < numth; i++)
```

```
        {
            Thread t = new Thread(new ThreadStart(A));
            t.Name = string.Format("Thread{0}", i++);
            t.Start();
        }
        Console.ReadLine();

    }
  }
}
```

OUTPUT