# THREAD Concurrency

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

The computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors.

Here is an example, the main (primary) t**hread spawning ten secondary worker threads. Each worker thread is told to make calls on the threadJob()** method on the same **PrintingThread** 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 **threadJob()** 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 **threadJob()** method will force the current thread to pause for a randomly generate amount of time:

using System;

using System.Threading;

namespace MultiThreadIII

{

public class PrintingThread

{

public void threadJob()

{

Console.WriteLine(Thread.CurrentThread.Name);

Console.WriteLine("Here are the numbers for: {0}",Thread.CurrentThread.Name);

for (int i = 0; i < 10; i++)

{

Random r = new Random();

Thread.Sleep(1000 \* r.Next(2));

Console.Write("{0} ", i);

}

Console.WriteLine();

}

// 10 Thread objects.

// each object call the same instance of the PrintingThread object.

public static void Main()

{

// make instance

PrintingThread pt = new PrintingThread();

// 10 threads that are all pointing to the same method on the same object

Thread[] ts = new Thread[10];

for (int i = 0; i < 10; i++)

{

ts[i] = new Thread(new ThreadStart(pt.threadJob));

ts[i].Name = string.Format("Worker thread [{0}]", i);

}

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

Console.ReadLine();

}

}

}

As expected, we get seemingly random results from the run:

Worker thread [0]

Worker thread [1]

Here are the numbers for: Worker thread [1]

0 1 2 3 4 Worker thread [2]

Here are the numbers for: Worker thread [2]

0 1 Here are the numbers for: Worker thread [0]

2 0 3 1 4 2 5 5 3 6 4 Worker thread [3]

Here are the numbers for: Worker thread [3]

7 6 0 8 7 1 8 2 9 Worker thread [4]

Here are the numbers for: Worker thread [4]

5 9

3

6 0 7 1 8 Worker thread [5]

Here are the numbers for: Worker thread [5]

2 9

4 3 5 0 Worker thread [6]

Here are the numbers for: Worker thread [6]

6 1 4 0 5 1 Worker thread [7]

Here are the numbers for: Worker thread [7]

6 2 7 2 7 0 3 8 3 1 9

Worker thread [8]

Here are the numbers for: Worker thread [8]

8 4 4 0 2 5 5 1 6 Worker thread [9]

Here are the numbers for: Worker thread [9]

9

3 6 7 8 2 0 4 7 1 3 9

5 4 8 6 9

5 6 7 7 2 8 8 9

9

3 4 5 6 7 8 9

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 - lock Keyword

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 PrintingThread

{

// lock token

private object threadLock = new object();

public void threadJob()

{

// using the lock token

lock(threadLock)

{

...

}

}

}

Here is the code using the **lock** keyword for synchronization.

using System;

using System.Threading;

namespace MultiThreadIII

{

public class PrintingThread

{

private object threadLock = new object();

public void threadJob()

{

lock (threadLock)

{

Console.WriteLine();

Console.WriteLine(Thread.CurrentThread.Name);

Console.WriteLine("Here are the numbers for: {0}",

Thread.CurrentThread.Name);

for (int i = 0; i < 10; i++)

{

Random r = new Random();

Thread.Sleep(1000 \* r.Next(2));

Console.Write("{0} ", i);

}

Console.WriteLine();

}

}

// 10 Thread objects.

// each object call the same instance of the PrintingThread object.

public static void Main()

{

// make instance

PrintingThread pt = new PrintingThread();

// 10 threads that are all pointing to the same method on the same object

Thread[] ts = new Thread[10];

for (int i = 0; i < 10; i++)

{

ts[i] = new Thread(new ThreadStart(pt.threadJob));

ts[i].Name = string.Format("Worker thread [{0}]", i);

}

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

Console.ReadLine();

}

}

}

Now we get consistent results by allowing the current thread to complete its task. Once a thread enters into the locked scope, the lock token is inaccessible by other threads until the current thread relinquishes the lock token:

Worker thread [0]

Here are the numbers for: Worker thread [0]

0 1 2 3 4 5 6 7 8 9

Worker thread [1]

Here are the numbers for: Worker thread [1]

0 1 2 3 4 5 6 7 8 9

Worker thread [2]

Here are the numbers for: Worker thread [2]

0 1 2 3 4 5 6 7 8 9

Worker thread [5]

Here are the numbers for: Worker thread [5]

0 1 2 3 4 5 6 7 8 9

Worker thread [7]

Here are the numbers for: Worker thread [7]

0 1 2 3 4 5 6 7 8 9

Worker thread [9]

Here are the numbers for: Worker thread [9]

0 1 2 3 4 5 6 7 8 9

Worker thread [3]

Here are the numbers for: Worker thread [3]

0 1 2 3 4 5 6 7 8 9

Worker thread [4]

Here are the numbers for: Worker thread [4]

0 1 2 3 4 5 6 7 8 9

Worker thread [6]

Here are the numbers for: Worker thread [6]

0 1 2 3 4 5 6 7 8 9

Worker thread [8]

Here are the numbers for: Worker thread [8]

0 1 2 3 4 5 6 7 8 9

# 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.Threading;

namespace MultiThreadIII

{

public class MutextTest

{

static Mutex mutex = new Mutex(false, "bogotobogo.com");

public static void Main()

{

if (!mutex.WaitOne(TimeSpan.FromSeconds(5), false))

{

Console.WriteLine("Busy running another app!");

return;

}

try

{

Console.WriteLine("Now running. Hit any key to exit.");

Console.ReadLine();

}

finally

{

mutex.ReleaseMutex();

}

}

}

}

Output is:

Now running. Hit any key to exit.

# Synchronization – Semaphore

A **Semaphore** with a capacity of **one** is like a **Mutex** or **lock**. However, the **Semaphore** has no **owner**. In other words, it's not aware of thread. Any thread can call **Release** on a **Semaphore** while with a **Mutex** or **lock**, only the thread that obtained the lock can release it.

We can use **Semaphores** to limit concurrency by preventing too many threads from executing a particular piece of code at once. In the following example, 7 threads are trying to enter a section of a code which allows only 5 threads at once:

using System;

using System.Threading;

namespace MultiThreadIII

{

public class SemaphoreTest

{

// 4 available from capacity of 5

static Semaphore semaphore = new Semaphore(4, 5);

public static void Main()

{

for (int i = 1; i <= 7; i++) new Thread(Enter).Start(i);

}

static void Enter(object id)

{

Console.WriteLine("{0} is trying to enter", id);

semaphore.WaitOne();

Console.WriteLine("{0} is in!", id);

Thread.Sleep(100 \* (int)id);

Console.WriteLine("{0} is leaving", id);

semaphore.Release();

}

}

}

Output from the code run:

1 is trying to enter

2 is trying to enter

2 is in!

1 is in!

3 is trying to enter

3 is in!

4 is trying to enter

4 is in!

5 is trying to enter

6 is trying to enter

7 is trying to enter

1 is leaving

7 is in!

2 is leaving

6 is in!

3 is leaving

5 is in!

4 is leaving

7 is leaving

6 is leaving

5 is leaving

# Synchronization – Monitor

The **lock** is just a shorthand notation for working with the **System.Threading.Monitor** class. Actually, the lock scope after the compilation looks something like this:

using System;

using System.Threading;

namespace MultiThreadIII

{

public class PrintingThread

{

private object threadLock = new object();

public void threadJob()

{

Monitor.Enter(threadLock);

try

{

Console.WriteLine();

Console.WriteLine(Thread.CurrentThread.Name);

Console.WriteLine("Here are the numbers for: {0}",

Thread.CurrentThread.Name);

for (int i = 0; i < 10; i++)

{

Random r = new Random();

Thread.Sleep(1000 \* r.Next(2));

Console.Write("{0} ", i);

}

Console.WriteLine();

}

finally

{

Monitor.Exit(threadLock);

}

}

// 10 Thread objects.

// each object call the same instance of the PrintingThread object.

public static void Main()

{

// make instance

PrintingThread pt = new PrintingThread();

// 10 threads that are all pointing to the same method on the same object

Thread[] ts = new Thread[10];

for (int i = 0; i < 10; i++)

{

ts[i] = new Thread(new ThreadStart(pt.threadJob));

ts[i].Name = string.Format("Worker thread [{0}]", i);

}

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

Console.ReadLine();

}

}

}

We'll get the same result as in the previous example. Note that the Monitor.Enter() method is the recipient of the thread token we specified as the argument to the lock.

Monitor.Enter(threadLock);

All code within a locked scope is wrapped within a **try** block. The **finally** block ensures that the thread token is released regardless of any possibility of runtime exception.

Monitor.Exit(threadLock);

But why we want to use System.Threading.Monitor though it requires more coding than just using the lock keyword approach?  
It's all about control. If we use the Monitor, we're able to tell the active thread to wait by using **Wait**() method and let the waiting threads know when the current thread is completed via **Pulse**() and **PulseAll**() methods.

# Synchronization - Semaphore vs. Monitor

In order to avoid data corruption and other problems, applications must control how threads access to shared resources. It is referred to as thread synchronization. The fundamental thread synchronization constructs are monitors and semaphores. Which one should we use? It depends on what the system or language supports.

* A monitor is a set of routines that are protected by a mutual exclusion lock. A thread cannot execute any of the routines in the monitor until it acquires the lock, which means that only one thread at a time can execute within the monitor. All other threads must wait for the currently executing thread to release the lock. A thread can suspend itself in the monitor and wait for an event to occur, in which case another thread is given the chance to enter the monitor. At some point the suspended thread is notified that the event has occurred, allowing it to awake and reacquire the lock as soon as possible.
* A semaphore is a simpler construct, just a lock that protects a shared resource. Before using a shared resource, the application must acquire the lock. Any other thread that tries to use the resource is blocked until the owning thread releases the lock, at which point one of the waiting threads acquires the lock and is unblocked. This is the most basic kind of semaphore, a mutual exclusion, or mutex, semaphore. There are other semaphore types, such as counting semaphores (which let a maximum of n threads access a resource at any given time) and event semaphores (which notify one or all waiting threads that an event has occurred), but they all work in much the same way.  
  Monitors and semaphores are equivalent, but monitors are simpler to use because they handle all details of lock acquisition and release. When using semaphores, an application must be very careful to release any locks a thread has acquired when it terminates. Otherwise, no other thread that needs the shared resource can proceed. In addition, every routine that accesses the shared resource must explicitly acquire a lock before using the resource, something that is easily forgotten when coding. Monitors always and automatically acquire the necessary locks.

# Synchronization - System.Threading.Interlocked

It's hard to realize that assignment and simple arithmetic operations are **not atomic** until we look at the underlying CIL code. So, the **System.Threading** provides a type that allow us to operate on a data atomically with less overhead than with the **Monitor** type. Here are the members of the **System.Threading.Interlocked** type.

|  |  |
| --- | --- |
| CompareExchange() | Safely tests two values for equality. If they are equal, changes one of the values with a third. |
| Decrement() | Safely decrements a value by 1. |
| Exchange() | Safely swaps two values. |
| **Increment()** | **Safely increments a value by 1.** |

Here is an example of converting a code of incrementing an integer to the one which does it atomically by using Interlocked.Increment() method:

public void IncrementIt()

{

lock(lockToken)

{

int myNewInt1 = myInt1++;

int myNewInt2 = myInt1--;

}

}

We can convert this to:

public void IncrementIt()

{

int myNewInt1 = Interlocked.Increment(ref myInt1);

int myNewInt2 = Interlocked.Decrement(ref myInt2);

}

Here are other examples of assignment and changing values:

public void SafeAssignment()

{

Interlocked.Exchange(ref i, 2011);

}

public void CompareAndExchange()

{

// if current value is 2010, change it to 2011

Interlocked.CompareExchange(ref i, 2011, 2010);

}

# [Synchronization] Attribute

Another way of synchronization is to use [Synchronization] Attribute.   
This is a member of the **System.Runtime.Remoting.Contexts** namespace. This class-level attribute effectively locks down **all** instance member code of the object to achieve the thread safety. It will place the object within a synchronized context when the CLR allocates objects which have [Synchronization] Attribute. Note that objects should not be removed from a contextual boundary should derive from **ContextBoundObject**. Here is the code and this will give the same results we get from the previous example:

using System;

using System.Threading;

using System.Runtime.Remoting.Contexts;

namespace MultiThreadIII

{

[Synchronization]

public class PrintingThread : ContextBoundObject

{

private object threadLock = new object();

public void threadJob()

{

Console.WriteLine();

Console.WriteLine(Thread.CurrentThread.Name);

Console.WriteLine("Here are the numbers for: {0}",

Thread.CurrentThread.Name);

for (int i = 0; i < 10; i++)

{

Random r = new Random();

Thread.Sleep(1000 \* r.Next(2));

Console.Write("{0} ", i);

}

Console.WriteLine();

}

// 10 Thread objects.

// each object call the same instance of the PrintingThread object.

public static void Main()

{

// make instance

PrintingThread pt = new PrintingThread();

// 10 threads that are all pointing to the same method on the same object

Thread[] ts = new Thread[10];

for (int i = 0; i < 10; i++)

{

ts[i] = new Thread(new ThreadStart(pt.threadJob));

ts[i].Name = string.Format("Worker thread [{0}]", i);

}

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

Console.ReadLine();

}

}

}

The downside of this approach is that even if a given method is not using thread-sensitive data, the CLR will **still** lock invocations to the method, and this could degrade the overall functionality of the type. So, we should be careful when use this approach.

# Timer Callbacks

Not a few applications need to call a specific method with certain time intervals.   
In this case, we can use the **System.Threading.Timer** type with a related delegate **TimerCallback**.

As an example, we'll write a code that will print the current time every 3 second until told not to:

using System;

using System.Threading;

using System.Runtime.Remoting.Contexts;

namespace MultiThreadIII

{

public class TimerCallbaskTest

{

static void DisplayTime(object state)

{

Console.WriteLine("Time is: {0}",

DateTime.Now.ToLongTimeString());

}

public static void Main()

{

// create the delegate for Timer

TimerCallback timerCallback = new TimerCallback(DisplayTime);

// set timer

Timer time = new Timer(

timerCallback, // TimerCallback delegate type

null, // Any info to pass into the called method

0, // Waiting time

3000); // Interval

Console.WriteLine("To terminate, hit any key..");

Console.ReadLine();

}

}

}

Output is:

To terminate, hit any key..

Time is: 7:31:08 PM

Time is: 7:31:11 PM

Time is: 7:31:14 PM

Time is: 7:31:17 PM

Time is: 7:31:20 PM

Time is: 7:31:23 PM

Time is: 7:31:26 PM

When we invoke a method asynchronously using delegate via **BeginInvoke**() method, the CLR does not create a new thread. For efficiency, a delegate's **BeginInvoke**() method uses a pool of worker threads which is maintained by runtime. To allow us to interact with this pool of waiting threads, the **System.Threading** provides the **ThreadPool** class types.

If we want to queue a method call for processing by a worker thread in the pool,   
we need to use the **ThreadPool.QueueUserWorkItem()** method. This method has been overloaded to allow us to specify an optional **System.Object** for custom state data in addition to an instance of the **WaitCallback** delegate:

public sealed class ThreadPool

{

...

public static bool QueueUserWorkItem(WaitCallback callBack);

public static bool QueueUserWorkItem(WaitCallback callBack, object state)

}

Here is the code:

using System;

using System.Threading;

namespace MultiThreadIII

{

public class ThreadPoolTest

{

static int count = 0;

public void threadJob()

{

Console.WriteLine("Doing threadJob {0}", count++);

}

}

class Program

{

public static void Main()

{

// make instance

ThreadPoolTest tp = new ThreadPoolTest();

WaitCallback wi = new WaitCallback(DoThreadJob);

// queue the method 10 times

for (int i = 0; i < 10; i++)

{

ThreadPool.QueueUserWorkItem(wi,tp);

}

Console.WriteLine("Queued all");

Console.ReadLine();

}

static void DoThreadJob(object state)

{

ThreadPoolTest tp = (ThreadPoolTest)state;

tp.threadJob();

}

}

}

Output is:

Queued all

Doing threadJob 1

Doing threadJob 0

Doing threadJob 2

Doing threadJob 4

Doing threadJob 5

Doing threadJob 3

Doing threadJob 7

Doing threadJob 6

Doing threadJob 9

Doing threadJob 8

Here is the summary:

* The thread pool manages thread efficiently by minimizing the number of threads that must be created, started, and stopped.
* By using the thread pool, we can focus on our business problem rather than the application's threading infrastructure.
* If we require foreground threads or must set the thread priority, we may want to use manual thread management because pooled threads are always background threads with default priority, ThreadPriority.Normal.
* If we need a thread with a fixed identity in order to abort it, suspend it, or discover it by name, the manual thread management should be preferred.