Thread Synchronization

HOW TO SYNCHRONIZE ACCESS TO RESOURCES IN MULTITHREADED APPLICATIONS

Synchronization methods

- Atomic operations
- Synchronization events (signals)
- Locks

Why is thread synchronization required

The asynchronous nature of threads means that access to resources such as file handles, network connections, and memory must be coordinated. Otherwise, two or more threads could access the same resource at the same time, each unaware of the other's actions. The result is unpredictable data corruption.

Atomic operation

The entire operation is a unit that cannot be interrupted by another thread

Interlocked class provides several atomic methods:

Add()	Adds second arg to first by reference
Read()	Reads 64-bit integer by reference and returns it
Increment()	Increments integer by reference
Decrement()	Decrements integer by reference
Exchange()	Updates by reference and returns old value
CompareExchange()	Exchange if existing value is equal to third argument

Interlocked.Add and Interlocked.Read

```
static long count = 15;

public static void UseAdd()
{
    long toAdd = 2;
    long addVal = Interlocked.Add(ref count, toAdd);
    // addVal == 17, count = 17
    long readVal = Interlocked.Read(ref count);
    // readVal == 17, count = 17
}
```

Interlocked.Exchange

```
static long count = 15;

public static void UseExchange()
{
    long toExchange = 2;
    long exchangeVal = Interlocked.Add(ref count, toExchange);
    // exchangeVal == 15, count = 2
}
```

Interlocked.CompareExchange()

```
static long count = 15;

public static void UseExchange()
{
    long toAdd = 2;

    long oldVal = Interlocked.Read(ref count);
    long newVal = oldVal + toAdd;
    long exchangeVal = Interlocked.CompareExchange(ref count, newVal, oldVal);
    if (exchangeVal == newVal) {
        // OK, value updated successfully
    } else {
        // Other thread changed count afrer Read() and before CompareExchange()
    }
}
```

Questions: atomic operations

1. What is an atomic operation?
2. Explain the difference between Exchange() and CompareExchange() methods

Synchronization Events and Wait Handles

Synchronization events are objects that have one of two states, signaled and un-signaled, that can be used to activate and suspend threads

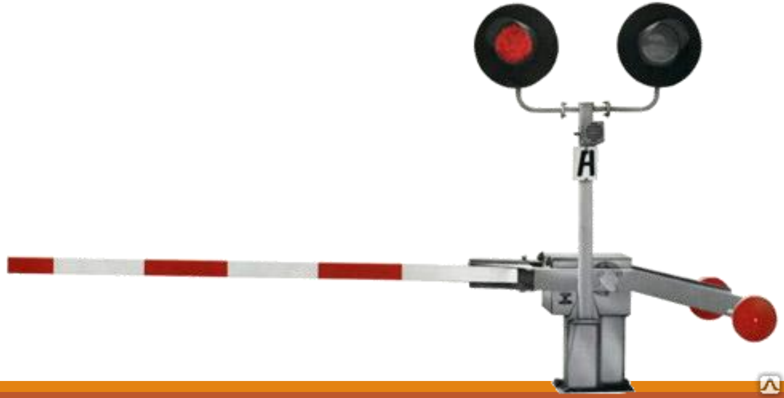
Using synchronization events one thread can tell to waiting thread, that action is resolved.

ManualResetEvent

Notifies one or more waiting threads that an event has occurred. This class cannot be inherited.

Usage:

- Set(), Reset() is used to change event signaled/un-signaled state
- WaitOne() blocks till event is signaled



AutoResetEvent

Notifies a waiting thread that an event has occurred. This class cannot be inherited.

Usage:

- Set(), Reset() - is used to change event signaled/un-signaled state
- WaitOne() - blocks till event is signaled. Resets event when released

WaitHandle

WaitHandle provides several static methods:

WaitAny(array of events)	Blocks if all events in array are un-signaled
WaitAll(array of events)	Blocks if any of events in array is un-signaled

```
class Program {
        private static AutoResetEvent _eventPulse = new AutoResetEvent(false);
        private static ManualResetEvent _eventStop = new ManualResetEvent(false);

        static void Main(string[] args) { // Start Threads
            Thread tThread1 = new Thread(ThreadLoop);
            tThread1.Start();
            Thread tThread2 = new Thread(ThreadLoop);
            tThread2.Start();
            Thread tThread3 = new Thread(ThreadLoop);
            tThread3.Start();

            for (int iIndex = 0; iIndex < 4; iIndex++) {
                Thread.Sleep(2000);
                _eventPulse.Set();   // Signal Thread to Execute
            }
            Thread.Sleep(2000);
            _eventStop.Set();        // Signal Threads to Exit
            Thread.Sleep(500);       // Wait for Threads to Exit
        }

        static void ThreadLoop() {
            Console.WriteLine("{0} Thread Start {1}", DateTime.Now.ToString("hh:MM:ss"), Thread.CurrentThread.ManagedThreadId);

            while (true) {
               if (WaitHandle.WaitAny(new WaitHandle[] { _eventStop, _eventPulse }) == 0) { // Wait For Events
                    Console.WriteLine("{0} Thread Terminating {1}", DateTime.Now.ToString("hh:MM:ss"),
Thread.CurrentThread.ManagedThreadId);
                    break;
               }

               Console.WriteLine("{0} Thread Executing {1}", DateTime.Now.ToString("hh:MM:ss"), Thread.CurrentThread.ManagedThreadId);
            }
         }
}
```

```
11:03:40 Thread Start 4
11:03:40 Thread Start 5
11:03:40 Thread Start 3
11:03:42 Thread Executing 4
11:03:44 Thread Executing 5
11:03:46 Thread Executing 3
11:03:48 Thread Executing 4
11:03:50 Thread Terminating 3
11:03:50 Thread Terminating 4
11:03:50 Thread Terminating 5
```

Questions: synchronization events

1. What is the difference between AutoResetEvent and ManualResetEvent?
2. How can user wait for one or all events?

The lock keyword

The C# lock statement can be used to ensure that a block of code runs to completion without interruption by other threads

```
public class TestThreading
{
  private System.Object lockThis = new System.Object();

  public void Process()
  {

      lock (lockThis)
      {
          // Access thread-sensitive resources.
      }
  }

}
```

The lock keyword

- The argument provided to the lock keyword must be an object based on a reference type
- Best to avoid locking on a public type, or on object instances beyond the control of your application
- Lock is internally implemented using Monitors

Monitor class

Like the lock keyword, monitors prevent blocks of code from simultaneous execution by multiple threads

```
System.Object obj = (System.Object)x;
System.Threading.Monitor.Enter(obj);
try
{
  DoSomething();
}
finally
{
  System.Threading.Monitor.Exit(obj);
}
```

Monitor

```
System.Object obj = (System.Object)x;

lock (x)
{
  DoSomething();
}
```

Equivalent using lock

Monitor class

The Monitor class provides additional functionality, which can be used in conjunction with the lock statement

The TryEnter method allows a thread that is blocked waiting for the resource to give up after a specified interval. It returns a Boolean value indicating success or failure, which can be used to detect and avoid potential deadlocks.

```
System.Object obj = (System.Object)x;
if (System.Threading.Monitor.TryEnter(obj, 1000)) {
 try
 {
   DoSomething();
 }
 finally
 {
   System.Threading.Monitor.Exit(obj);
 }
}
else {
 DoSomethingElse();
}
```

Questions: locks

1. Is thread synchronization required when app is running on single-processor single core PC? Why?
2. Is block after lock keyword atomic?
3. When should one use Monitor class and when lock keyword?
4. Can we lock on a value type? Why?

Mutex object

A mutex is similar to a monitor; it prevents the simultaneous execution of a block of code by more than one thread at a time

```
Mutex mut = new Mutex();
mut.WaitOne();
try
{
  DoSomething();
}
finally
{
  mut.ReleaseMutex();
}
```

```
Mutex mut = new Mutex();
if (mut.WaitOne(1000)) {
 try
 {
   DoSomething();
 }
 finally
 {
   mut.ReleaseMutex();
 }
}
else {
 DoSomethingElse();
}
```

Mutex object

Unlike the Monitor class, a mutex can be either local or global.

Global mutexes, also called named mutexes, are visible throughout the operating system, and can be used to synchronize threads in multiple application domains or processes.

Local mutexes derive from MarshalByRefObject, and can be used across application domain boundaries.

In addition, Mutex derives from WaitHandle, which means that it can be used with the signaling mechanisms provided by WaitHandle, such as the WaitAll, WaitAny, and SignalAndWait methods.

Semaphore class

Limits the number of threads that can access a resource or pool of resources concurrently.

Semaphore with pool size equal to 1 works like Mutex

- WaitOne() Increase count. Lock if count reaches maximum
- Release() Decrease count. Unlock one of waiting threads if any

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

public class Example {
   private static Semaphore _pool;
   private static int _padding;

   public static void Main() {
       _pool = new Semaphore(0, 3);
       for(int i = 1; i <= 5; i++) {
           Thread t = new Thread(new ParameterizedThreadStart(Worker));
           t.Start(i);
       }

       Thread.Sleep(500);
       Console.WriteLine("Main thread calls Release(3).");

       _pool.Release(3);
       Console.WriteLine("Main thread exits.");
   }

   private static void Worker(object num) {
       Console.WriteLine("Thread {0} begins " + "and waits for the semaphore.", num);
       _pool.WaitOne();
       int padding = Interlocked.Add(ref _padding, 100);
       Console.WriteLine("Thread {0} enters the semaphore.", num);

       Thread.Sleep(1000 + padding);

       Console.WriteLine("Thread {0} releases the semaphore.", num);
       Console.WriteLine("Thread {0} previous semaphore count: {1}", num, _pool.Release());
   }
}
```

```
Thread 1 begins and waits for the semaphore.
Thread 2 begins and waits for the semaphore.
Thread 3 begins and waits for the semaphore.
Thread 4 begins and waits for the semaphore.
Thread 5 begins and waits for the semaphore.
Main thread calls Release(3).
Thread 1 enters the semaphore.
Main thread exits.
Thread 2 enters the semaphore.
Thread 3 enters the semaphore.
Thread 1 releases the semaphore.
Thread 4 enters the semaphore.
Thread 1 previous semaphore count: 0
Thread 3 releases the semaphore.
Thread 3 previous semaphore count: 0
Thread 5 enters the semaphore.
Thread 2 releases the semaphore.
Thread 2 previous semaphore count: 0
Thread 4 releases the semaphore.
Thread 4 previous semaphore count: 1
Thread 5 releases the semaphore.
Thread 5 previous semaphore count: 2
```

ReadWriter locks

Defines a lock that supports single writers and multiple readers.

AcquireReaderLock()	Lock for read
ReleaseReaderLock()	Release when read finished
AcquireWriterLock()	Lock for write
ReleaseWriterLock()	Release when write finished
ReleaseLock()	Release all locks for current thread

Questions: More locks

1. When should one use Mutex or Monitor (lock)?
2. What is Semaphore class? When should it be used?
3. Imagine the situation: Three threads are reading data protected with ReadWriterLock. Fourth thread is going to update the data. Describe next steps one by one.

Thread-safe collections

Multiple threads can safely and efficiently add or remove items from these collections, without requiring additional synchronization in user code

- ConcurrentDictionary<TKey, TValue>
- ConcurrentQueue<T>
- ConcurrentStack<T>

ConcurrentDictionary.TryUpdate

```
static ConcurrentDictionary<string, int> cd = new ConcurrentDictionary<string, int>();

public void UpdateUser(string user)
{
    int oldVal = cd[user];
    int newVal = CalculateNewValue(user, oldVal);
    if (!cd.TryUpdate(user, newVal, oldVal)) {
        DoSomethingElse(user, oldVal);
    }
}
```

Non thread-safe collections

Think about

- Using thread-safe collections instead of non thread-safe
- Lock using .SyncRoot property

SyncRoot property

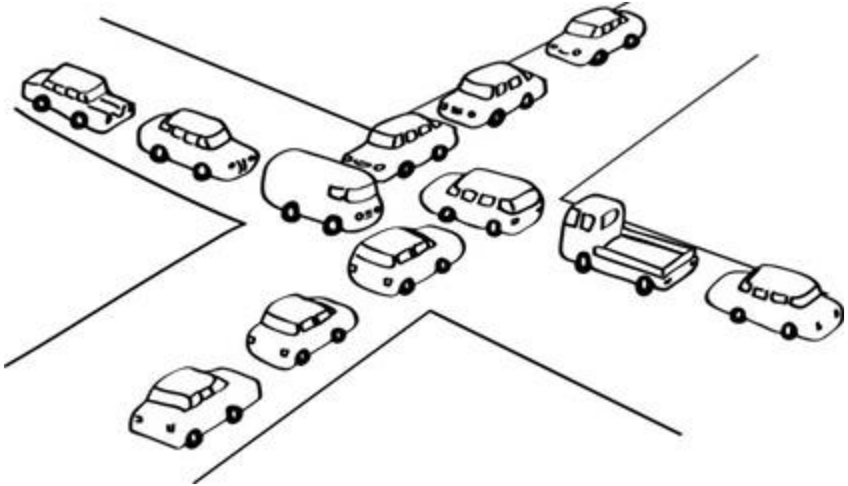
```
private static List<string> list = new List<string>();

public void IterateList()
{
    lock(list.SyncRoot)
    {
        foreach (string item in list)
        {
            DoSomethingWithItem(item);
        }
    }
}
```

Deadlocks

Multiple threads are waiting for each other and the application comes to a halt

Livelocks do not halt, but can't exit locking loop



```
static object object1 = new object();
static object object2 = new object();

public static void ObliviousFunction()
{
    lock (object1)
    {
        Thread.Sleep(1000); // Wait for the blind to lead
        lock (object2)
        {
        }
    }
}

public static void BlindFunction()
{
    lock (object2)
    {
        Thread.Sleep(1000); // Wait for oblivion
        lock (object1)
        {
        }
    }
}
```

Common deadlock prevention advices

- Don’t take the fork until you have put the spoon
- Don’t take the fork and the spoon simultaneously

Race conditions

Behaviour when the output is dependent on the sequence or timing of other uncontrollable events.

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

General recommendations

- Don't use Thread.Suspend and Thread.Resume to synchronize threads
- Don't use types as lock objects (lock (typeof(MyClass)))
- Use caution when locking on instances (lock (this))
- Do use multiple threads for tasks that require different resources, and avoid assigning multiple threads to a single resource

Recommendations for Class Libraries

- Avoid the need for synchronization, if possible
- Make static data thread safe by default
- Do not make instance data thread safe by default
- Avoid providing static methods that alter static state

Questions: Collections and deadlocks

1. List build-in .NET thread-safe collections
2. What is SyncRoot property? Where it exists and when used?
3. How to avoid deadlocks?