



# C# THREADS

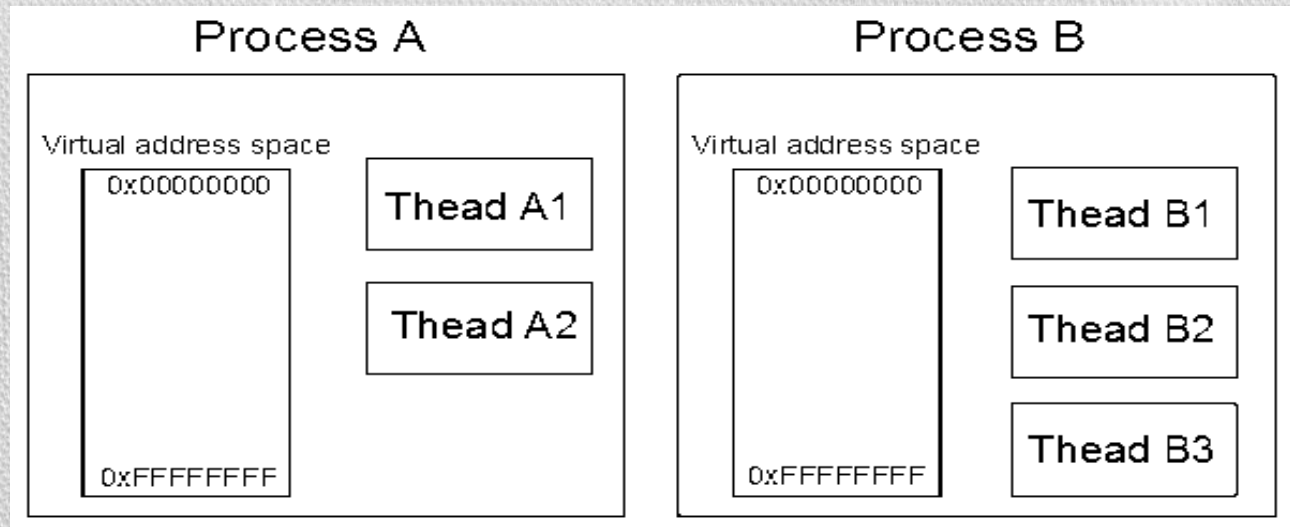
Samyukta Mudugal

CSCI 5448 – Spring 2011

# Outline

- Introduction to threads.
- Advantages and Disadvantages of using threads
- C# support for threading
- Creating threads in C#
  - Using Thread Class
  - Using Thread Pool
- Synchronization of threads
  - Blocking
  - Simple Locking
  - Mutex
  - Semaphores
- Deadlocks
- Aborting Threads

# Thread



- A thread is an independent execution path with its own resources allocated by the CPU.
- Threads run in parallel within a process just as processes run in parallel on a computer.
- A process can have multiple threads and they share memory among them.
- Each thread had its own context of execution.
  - The meaning of context differs from one OS to the other.
  - It could be any of these – registers, memory maps, tables, lists etc.

# Magic of Multithreading

- Achieve true parallelism in a multiprocessor environment.
  - The threads can execute in parallel on different processors.
- Increasing the responsiveness of a user interface.
  - By running a parallel “worker” thread, the main UI thread is free to continue processing keyboard and mouse events.
- Achieve better use of an otherwise blocked CPU.
  - While one thread waits for a response from a hardware or another computer, the other threads can continue execution.
- Distinguish tasks of varying priority.
  - A high-priority thread manages time-critical tasks, and a low-priority thread performs other tasks.

# Cost of Multithreading

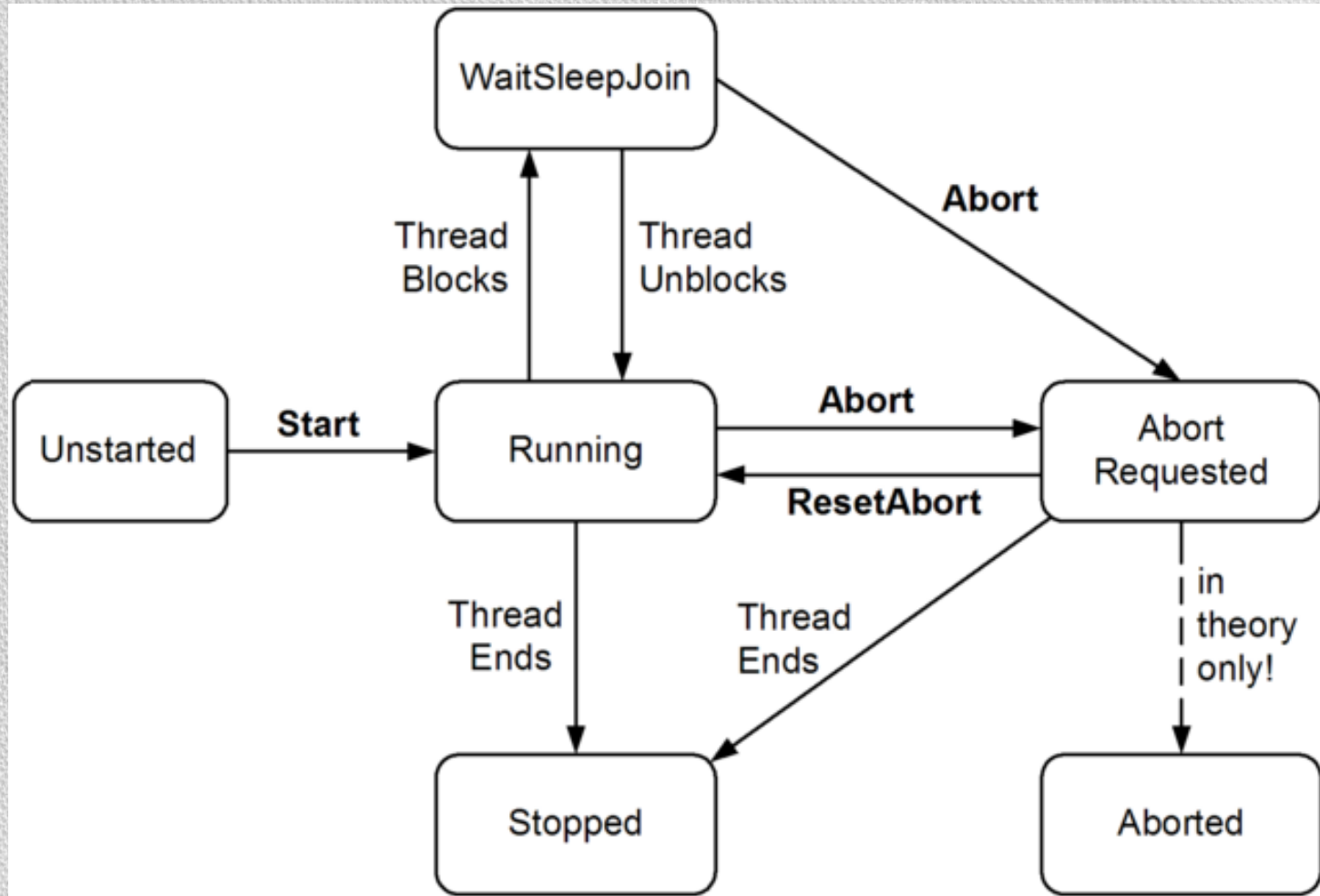
- System could get complex!
- CPU costs – scheduling, managing, switching, creation/tear down.
  - Instead of making your system faster it could in turn make it slower!
- Controlling the shared access of resources.
  - Using the synchronization objects appropriately around the shared resources.
- IF synchronization is not done properly it could lead to problems such as deadlocks, race conditions.
  - These issues are extremely hard to debug.
- Testing multithreaded applications is harder.
  - Defects are often timing-related and more difficult to reproduce.

# Threading in C#

- Import the following namespaces to support threading -  
using System;  
using System.Threading;
- C# client program starts in a single thread created automatically by the CLR and operating system (the “main” thread), and is made multithreaded by creating additional threads.
- C# provides a ‘thread’ class that can be used to create a thread, control a thread, set its priority, get its status, etc.
- C# also provides a pool of threads to which tasks can be assigned.

# Thread State Machine

- A thread can be in one of the major states shown below



# Creation

- C# supports 2 methods to create threads
  - Using the Thread class.
  - Using the Thread pool.
- Thread Class: A new thread object is created and a delegate is passed to the thread's constructor.
  - A thread ends when the delegate passed to the Thread's constructor finishes executing.
- Thread Pool: C# provides a pool of threads. Tasks can be assigned to the threads in the pool. This gets rid of the overhead of creation, deletion and managing the threads.



# Using the Thread Class

```
namespace Thread1
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;
            // Creating the thread
            Console.WriteLine("Creating the thread using the THREAD Class. \n");
            Thread foo = new Thread(testFunc);
            // Starting the thread
            Console.WriteLine("Starting the execution of the foo thread. \n");
            foo.Start();
            Console.WriteLine("Waiting for the thread to be alive. \n");
            while (!foo.IsAlive) ;

            // Main Thread execution
            for (i = 0; i < 50; i++)
            {
                Console.Write("Y");
            }
            while (true) ;
        }

        static void testFunc()
        {
            int i;
            for (i = 0; i < 50; i++)
            {
                Console.Write("X");
            }
        }
    }
}
```



# Using the Thread Pool

- Initializing the arrays and Manual Reset Events.
- Manual Reset Event is an object that allows one thread to signal another thread when something happens. In the case of this code, we use these events to signal the main thread that a work item has been completed.

```
namespace ThreadPoolTest
{
    class Program
    {
        private const int NumThreads = 5;

        private static int[] inputArray;
        private static double[] resultArray;
        private static ManualResetEvent[] resetEvents;

        private static void Main(string[] args)
        {
            inputArray = new int[NumThreads];
            resultArray = new double[NumThreads];
            resetEvents = new ManualResetEvent[NumThreads];
        }
    }
}
```

# Thread Pool contd..

- Assigning the tasks to the threads in the pool.
- Any method that you want to queue up for the thread pool to run needs to take one argument, an object, and return void.
- The argument will end up being whatever you passed in as the second argument to the *QueueUserWorkItem* call.

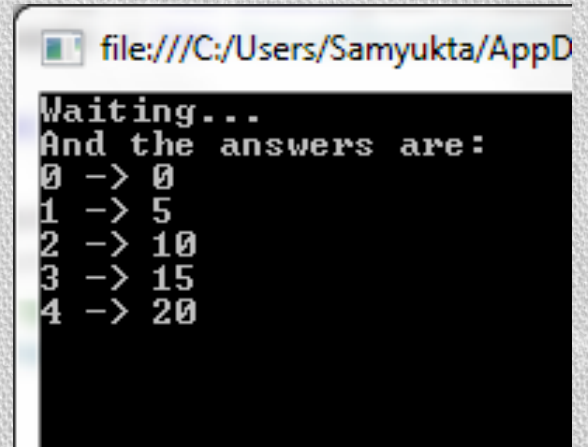
```
for (int s = 0; s < NumThreads; s++)
{
    inputArray[s] = s;
    resetEvents[s] = new ManualResetEvent(false);
    // Queue the tasks onto the threads in the pool
    ThreadPool.QueueUserWorkItem(new WaitCallback(DoWork), (object)s);
}

// Wait for all the threads to finish
Console.WriteLine("Waiting...");
WaitHandle.WaitAll(resetEvents);

// Print the answers
Console.WriteLine("And the answers are: ");
for (int i = 0; i < NumThreads; i++)
    Console.WriteLine(inputArray[i] + " -> " + resultArray[i]);
while (true) ;
}

private static void DoWork(object o)
{
    int index = (int)o;
    // Fills the resultant array initial*5
    resultArray[index] = index*5;

    resetEvents[index].Set();
}
3/30/2011
```



```
file:///C:/Users/Samyukta/AppD
Waiting...
And the answers are:
0 -> 0
1 -> 5
2 -> 10
3 -> 15
4 -> 20
```

# Synchronization

- Synchronization refers to coordinating the actions of threads for a predictable outcome.
- Synchronization is particularly important when threads access the same data.
- 4 different kinds of synchronization constructs:
  - Simple Blocking methods
  - Locking Constructs
  - Signaling Constructs
  - Non Blocking Synchronization Constructs

# Blocking

- Sometimes a thread might have to pause until a certain condition is met. It could either spin or block.
- A blocked thread immediately *yields* its processor time slice, and from then on consumes no processor time until its blocking condition is satisfied.
- Spinning is wasteful on processor time because time and resources are allocated accordingly even though the thread is just waiting.
- Some methods used for this are: Sleep, Join and Task.Wait.
- The ***ThreadState*** property is used to check if the thread is blocked.

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```

# Simple Locking

- 3 main exclusive locking constructs are **locks**, **mutexes** and **semaphores**.
- Only one thread can lock the synchronizing object (myLock) at a time, and any contending threads are blocked until the lock is released.
- If more than one thread contends the lock, they are queued on a “ready queue” and granted the lock on a first-come, first-served basis.

```
class threadSafe_Locker
{
    static readonly object myLock = new object();
    static int val;

    static void DemoLock()
    {
        lock(myLock)
        {
            val++;
        }
    }
}
```

# Mutex

- Mutexes can span across multiple processes in a system.
- To lock the mutex: WaitOne method
- To unlock the mutex: ReleaseMutex method

```
class OneAtATimePlease
{
    static void Main()
    {
        // Naming a Mutex makes it available computer-wide. Use a name that's
        // unique to your company and application (e.g., include your URL).

        using (var mutex = new Mutex (false, "oreilly.com OneAtATimeDemo"))
        {
            // Wait a few seconds if contended, in case another instance
            // of the program is still in the process of shutting down.

            if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))
            {
                Console.WriteLine ("Another app instance is running. Bye!");
                return;
            }
            RunProgram();
        }
    }

    static void RunProgram()
    {
        Console.WriteLine ("Running. Press Enter to exit");
        Console.ReadLine();
    }
}
```

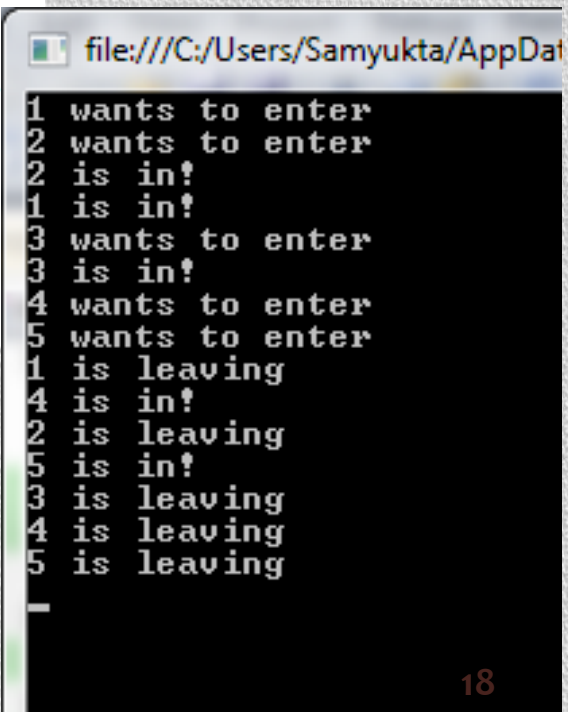


# Semaphores

- Semaphores limit the number of threads that can access a resource concurrently. The maximum number of threads is called the capacity of the semaphore.
- They work by keeping a counter.
- Each time a thread obtains the semaphore the counter is incremented and once the semaphore is released the counter is decremented.
- A semaphore with a capacity of one is same as a mutex or a lock.
- Any thread can call lock or release on a semaphore but with a mutex, only the thread that obtained it can release it.
- Semaphores can be useful in preventing too many threads executing the same piece of code at once.

# Semaphores contd..

```
class Program
{
    static SemaphoreSlim mySemaphore = new SemaphoreSlim(3);
    static void Main(string[] args)
    {
        for (int i = 1; i <= 5; i++) new Thread(SemFunc).Start(i);
        while (true) ;
    }
    static void SemFunc(object id)
    {
        Console.WriteLine(id + " wants to enter");
        mySemaphore.Wait();
        Console.WriteLine(id + " is in!");           // Only three threads
        Thread.Sleep(1000 * (int)id);              // can be here at
        Console.WriteLine(id + " is leaving");       // a time.
        mySemaphore.Release();
    }
}
```



```
file:///C:/Users/Samyukta/AppDat
1 wants to enter
2 wants to enter
2 is in!
1 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
3 is leaving
4 is leaving
5 is leaving
```

# Deadlocks

- A deadlock happens when two threads each wait for a resource held by the other, so neither can proceed.
- Example of deadlock:
  - Thread 1 acquires lock A.
  - Thread 2 acquires lock B.
  - Thread 1 attempts to acquire lock B, but it is already held by Thread 2 and thus Thread 1 blocks until B is released.
  - Thread 2 attempts to acquire lock A, but it is held by Thread 1 and thus Thread 2 blocks until A is released
- At this point, both threads are blocked and will never wake up!

# Deadlocks contd..

- Example code to demonstrate deadlock
- Deadlocks can be avoided using various methods such as:
  - Lock Leveling
  - Threads acquiring the resource in same order.
  - Using a time out during acquiring a lock.
  - Be wary of locking around calling methods in objects that may have references back to your own object.

```
object lockA = new object();
object lockB = new object();

// Thread 1
void t1() {
    lock (lockA) {
        lock (lockB) {
            /* ... */
        }
    }
}

// Thread 2
void t2() {
    lock (lockB) {
        lock (lockA) {
            /* ... */
        }
    }
}
```

# Aborting Threads

- A thread can be forcibly ended by calling the ‘abort’ method.

```
class Abort
{
    static void Main()
    {
        Thread t = new Thread (delegate() { while (true); } );

        Console.WriteLine (t.ThreadState);    // Unstarted

        t.Start();
        Thread.Sleep (1000);
        Console.WriteLine (t.ThreadState);    // Running

        t.Abort();
        Console.WriteLine (t.ThreadState);    // AbortRequested

        t.Join();
        Console.WriteLine (t.ThreadState);    // Stopped
    }
}
```

- The thread upon being aborted immediately enters the AbortRequested state. If it then terminates as expected, it goes into the Stopped state. The caller can wait for this to happen by calling Join.

# References

- MSDN C# threads library and tutorial -  
[http://msdn.microsoft.com/en-us/library/aa645740\(v=VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa645740(v=VS.71).aspx)
- Threading in C# -Joseph Albahari  
<http://www.albahari.com/threading/>
- Wikipedia Entry on threads  
[http://en.wikipedia.org/wiki/Thread\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Thread_(computer_science))



# Questions?