# Concurrent Programming

C#

# Threads

When threads are useful?

When we want our program to react when some "heavy" task is performed in the background

Various types of processes - servers. While waiting for data on one thread, a program may be executing on another.

When we have a program that does a lot of calculations (e.g. compression of multimedia files) and we want to parallel them in some way. The effect will be felt when we physically have many cores.

This number can be checked with **Environment.ProcessorCount**

# Threads

When the threads can be harmful?

- When there are too many of them. The time for switching and allocation will be too expensive,
- When a task performed by a thread will take less time than starting a given thread.
- When we don't fully predict interactions between threads, debugging is very cumbersome.
- When we use a lot of disk, we should not be creating multiple threads, but rather one or two and scheduling read and write jobs. (someone tried to copy several files from a CD / DVD at once?)

# Threads

- When we operate on threads, we need to add to the program

```
using System.Threading;
```

- Each program has at least one thread, called the main thread
- Each thread has its own separate stack so local variables are modified independently.
- Global variables are shared by threads (synchronization often required)

# Threads

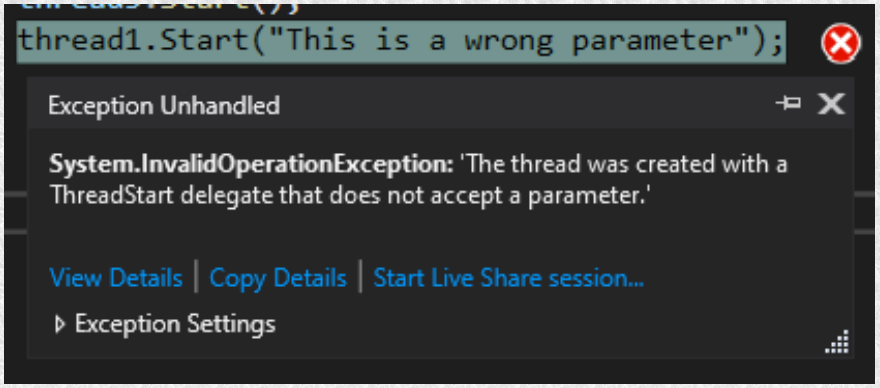
An example of the methods acting as threads.

```
static void OurThreadWithoutParameters()
{
    Console.WriteLine("Thread without parameters");
}

static void OurThread()
{
    Console.WriteLine($"Overloaded method Thread without parameeter");
}

static void OurThread(object o)
{
    string message = o as string;
    Console.WriteLine($"Overloaded method Thread with parameeter got message:
{message}");
}

private void OurThreadWithParameter(object o)
{
 if (o == null)
    {
    Console.WriteLine("I got null as a parameter");
    return;
    }
Console.WriteLine("Thread with parameter got message: " + (string)o);
}
```

# Threads

## Creating and starting threads without parameters

```
Thread thread1 = new Thread(new ThreadStart(OurThreadWithoutParameters));
Thread thread2 = new Thread(OurThreadWithoutParameters); //compilator adds
automaticly new(ThreadStart
Thread thread3 = new Thread(new ThreadStart(OurThread));
//Thread thread3 = new Thread(OurThread); //When we have overloaded methods (without
and with parameters, the compiler does not know whether to use ThreadStart or
ParameterizedThreadStart
thread1.Start();
thread2.Start();
thread3.Start();
//thread1.Start("This is a wrong parameter");//Invalid operation exception
```



# Threads

## Example of run threads with parameters

```
Thread thread4 = new Thread(new
ParameterizedThreadStart(OurThreadWithParameter));
Thread thread5 = new Thread(OurThreadWithParameter);
Thread thread6 = new Thread(new ParameterizedThreadStart(OurThread));

thread4.Start(); //Possible such a start, but there will be no parameters o
= null
thread5.Start("Message for thread 5");
thread6.Start("Message for thread 6");
```

# Threads

## An anonymous run example

```
static void OurThreadWithTypedParameter(string message)
{
    Console.WriteLine("Thread with typed parameter got message: " + message);
}

string changingMessage;
changingMessage = "Message befor thread created";
Thread thread7 = new Thread(delegate ()
{ OurThreadWithTypedParameter(changingMessage); }); //using an anonymous
method, we do not have to specify the object parameter, but we can specify a
specific type, e.g. string
changingMessage = "Message after thread chreated";
Thread thread8 = new Thread(delegate()
    {
    Console.WriteLine(changingMessage);
    });
thread7.Start();
thread8.Start();
```

# Threads

## Example of running with lambda

```
Thread thread9 = new Thread((parameter) =>
        {  //thread code
        Console.WriteLine("Lambda thread show message: " + (parameter as
string));
        });
thread9.Start("Some message for lambda created thread");

Thread thread10 = new Thread((parameter) =>
        {  //thread code
        OurThreadWithTypedParameter(parameter as string);
        });
thread10.Start("Some message for lambda created thread with method");
```

# Threads

Example of running from an object

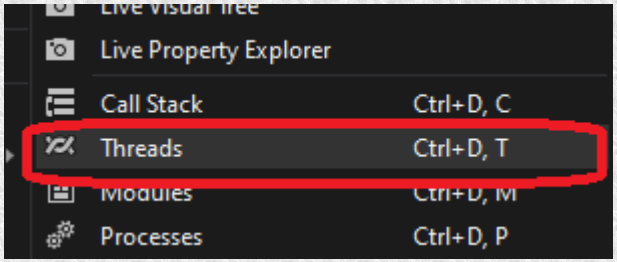
```
public class VariousThreads
        {
            public void thread1()
             {
             MessageBox.Show("I'm thread 1");
             }
            public void thread2()
             {
             MessageBox.Show("I'm thread 2");
             }
        }

VariousThreads varTh = new VariousThreads();
  Thread th1 = new Thread(varTh.thread1);
  Thread th2 = new Thread(varTh.thread2);
  th1.Start();
  th2.Start();
```

# Threads

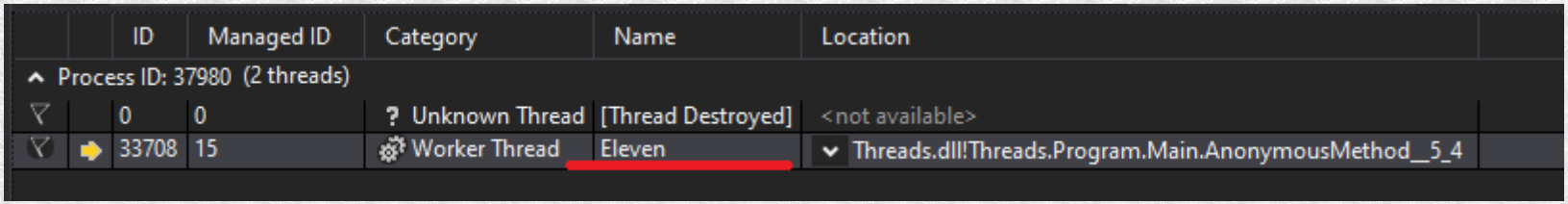
## Naming Threads - Debugging Help

```
Thread thread11 = new Thread(() =>
Console.WriteLine($"Hello. I'm {Thread.CurrentThread.Name} thread.")
            );
thread11.Name = "Eleven";
thread11.Start();
```

You can show threads debug window by selecting menu: Debug->Windows->Threads




On this window you will see all application threads with theirs names:



# Threads

## Foreground and background threads

Every thread has **IsBackground** property. Default it is **false**. When parent thread has finished child threads are still running.

When **IsBackground=true**. Child Threads end with parent together.
The finally block is skipped. This is an undesirable situation, so we should wait for the end of child threads.

Changing the work from background to foreground and vice versa does not affect priority.

# Threads

## Foreground and background threads

```
Thread parentThread = new Thread(() =>
        {
        Console.WriteLine($"Thread {Thread.CurrentThread.Name} started");
        Thread childThread1 = new Thread(() =>
            {
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} started");
            Thread.Sleep(5000);
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} finished");
            })
            { Name = "ChildThread1"}; //backgroud = false as default
            //{ Name = "ChildThread1",IsBackground = true };
         Thread childThread2 = new Thread(() =>
            {
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} started");
            Thread.Sleep(5000);
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} finished");
            })
            { Name = "ChildThread2"};//backgroud = false as default
            //{ Name = "ChildThread2",IsBackground=true };

            childThread1.Start();
            childThread2.Start();
            Console.WriteLine($"Thread {Thread.CurrentThread.Name} finished");
        })
        { Name = "Parent" };
parentThread.Start();
```

# Threads

## Priority

```
enum ThreadPriority { Lowest, BelowNormal, Normal,
  AboveNormal, Highest }
```

- Indicates how much processor time is allocated to a given thread in a thread group of one process.
- Setting it to Highest does not mean that it will be a real-time thread. You would also have to set a priority for the process.

```
Process.GetCurrentProcess().PriorityClass =
    ProcessPriorityClass.High;
```

- **Realtime** priority is even higher, then our process will run continuously, but when it enters the endless loop, we will not regain control over the system.
- If our application has a (especially complicated) graphical interface, we should not raise the priority either, because refreshing will slow down the entire system.

# Process

```
static void Main(string[] args)
{
    Process process = Process.Start("notepad.exe");
     Thread.Sleep(3000);
     process.Kill();

     var e = Process.Start(@"C:\Program Files (x86)\Microsoft\Edge\Application\
msedge.exe");

     var processes = Process.GetProcesses();
     foreach (var item in processes)
        {
         try
            {
            Console.WriteLine(item.ProcessName + item.MainModule.FileName);
            }
        catch (Exception ex)
            {
            Console.WriteLine(ex.Message);
            }
        }
    var edges = Process.GetProcessesByName("msedge");
    Thread.Sleep(3000);
    edges[0].Kill(true);
}
```

# Threads

## Exceptions

```
Try
   {
    thread.Start();
   }
catch
    {
    Console.WriteLine("Error during starting thread");
    }
```

- Such a launch will only throw us an exception at startup, we will not catch an exception "thrown" in the thread.
- Exceptions from threads can end the application and must be caught at the thread level.

# Synchronization

## Blocking

- Processes blocked due to waiting for an event, e.g. **Sleep, Join, lock, Semaphore** etc. Immediately resing CPU time, add **WaitSleepJoin** to **ThreadState** property, and don't queue until unlocked
- Unblocking may occur for 4 reasons:
  - The unlock condition has been met
  - Timeout expired
  - Has been interrupted by **Thread.Interrupt**
  - Has been interrupted by **Thread.Abort** (.net<5.0 and not .core)

# Synchronization

## Waiting `Sleep` and `SpinWait`

```
Thread.Sleep(0); // resign of the assigned time quantum
Thread.Sleep(1000); // sleep for 1000 ms
Thread.Sleep(TimeSpan.FromHours(1)); // sleep for 1 hour
Thread.Sleep(Timeout.Infinite); // sleep forever until break.
```

In general, Sleep causes the thread to give up CPU time. Such thread is not queued for the given time.

```
Thread.SpinWait (100); // do empty 100 processor cycles
```

The thread does not give up on the processor, but performs empty operations on it. It is not in the **WaitSleepJoin** state and cannot be interrupted by **Interrupt**. It can be used when we want to wait very shortly or simulate a load.
A thread behaves similarly during active wait.

# Synchronization

## Wait Join

```
Thread thread1 = new Thread(() =>
    {
    Console.WriteLine("Thread started and going to sleep for 3 s...");
    Thread.Sleep(3000);
    Console.WriteLine("Thread woke up and finished");
    });
thread1.Start();
Console.WriteLine("Main thread has started child thread and going to
wait for it...");
thread1.Join();
Console.WriteLine("Main thread lived to see");
```

We are waiting for the thread to end. The mechanism of collecting messages is not stopped, in the case of a window application, events would be queued

# Critical section

- When to block
  - Anywhere where multiple threads can access common variables
  - Wherever we want to have the indivisibility of operations, e.g. checking a condition and executing something
- what to beware of
  - We shouldn't block too much because it's hard to analyze such code and it's easy to cause DeadLock
  - Too large pieces of code executed by a single process reduction concurrency.
  - When the granularity is too small, the synchronization time is high

# Critical section

Several threads (n_threads) do the same task:

```
for (int ii = 0; ii < 1000000; ii++)
   {
        {
            counter++;
        }
   }
```

Without protection, the result will be <= n_threads*1000000

# Critical section

lock

```
private object locker = new object();

...

for (int ii = 0; ii < 1000000; ii++)
      {
      lock (locker)
         {
         counter++;
         }
      }
```

- Only one thread at a time may be in the protected area, others will be waiting in the FIFO queue
- Waiting threads are in the state of **`WaitSleepJoin`**.
- Such threads can also be terminated by interrupt or abort

# Critical section

Selection of the object that will be locker

- It must be a reference type
- Usually it is related to the facilities we operate on, e.g.

```
List <string> list = new List <string>();
void Test() {
   lock (list) {
      list.Add ("Item 1");
...
```

We should use objects that are `private` to avoid unintended interaction from the outside

- For the same reason, we should not use e.g. `lock (this){}` or `lock (typeof (Widget)) { ... }`
- Using an object to lock a snippet of code does not automatically lock that object. It is possible to change the object outside the protected area.

# Critical section

## Nested locking

```
static object x = new object();
  static void Main() {
  lock (x) {
     Console.WriteLine ("I locked");
     Nest();
     Console.WriteLine ("I unlocked");
     }
  //Completly unlocked
  }

static void Nest() {
  lock (x) {
  …//double lock
  }
  //last unlock
  }
```

The locking thread can block as much as it wants, but the blocked thread will wait for the outermost one anyway.

# Critical section

`Monitor` – explication `lock`

- `lock` it is actually a syntax shortcut for something like this:

```
Monitor.Enter(locker);
   try
          {
          counter++;
          }

finally
          {
          Monitor.Exit(locker);
          }
```

- Call `Monitor.Exit` without earlier `Monitor.Enter` couses exemption throw

- The monitor also has a `TryEnter` method where we can specify a timeout if we enter before the end of time, it will return true or false if a timout occurs

# Critical section

`Monitor` – explication `lock`

```
Thread[] tharray3 = new Thread[100];
for (int ii = 0; ii < 100; ii++)
    {
    tharray3[ii] = new Thread(ThreadWithMonitor);
    tharray3[ii].Name = "Thread " + ii.ToString();
    }
sw.Restart();
foreach (var th in tharray3)
    {
    th.Start();
    }
foreach (var th in tharray3)
    {
    th.Join();
    }
Console.WriteLine($"All Monitor threads finished in
{sw.ElapsedMilliseconds} ms counter = {counter}");
```

# Critical section

*Interlocked* – atomic operations

```
for (int ii = 0; ii < 1000000; ii++)
    {
    Interlocked.Increment(ref counter);
    }
```

- Additionally, we have at our disposal
  - `Add` Adding up to two numbers
  - `CompareExchange` comparison and possible substitution
  - `Decrement`
  - `Equals`
  - `Exchange`
  - `Read` reading the number 64b
  - `ReferenceEquals` comparison of two references

# Critical section

Mutex

```
static Mutex mutex = new Mutex(false, "tu.kielce.pl mutex");
        static void ThreadWithMutex(object o)
        {
            //it is very slow, there is 100 times less iterations
            for (int ii = 0; ii < countFor/100; ii++)
            {
                mutex.WaitOne();
                counter++;
                mutex.ReleaseMutex();
            }
        }
```

# Critical section

Semaphore

```
static Semaphore sem = new Semaphore(1, 1);
static void ThreadWithSemaphore()
{
    //it is very slow, there is 100 times less iterations
    for (int ii = 0; ii < countFor/100; ii++)
    {
        sem.WaitOne();
        counter++;
        sem.Release();
    }
}
```

# Critical section

SemaphoreSlim

- **SemaphoreSlim** is a lightweight alternative to Semaphore and can be used only for synchronization within a single process boundary
- It doesn't support named system semaphores.

```
static SemaphoreSlim semSlim = new SemaphoreSlim(1, 1);
static void ThreadWithSemaphoreSlim()
{   //Notice. It isn't so slow as Semaphore
    for (int ii = 0; ii < countFor; ii++)
    {
        semSlim.Wait();
        counter++;
        semSlim.Release();
    }
}
```

# Critical section

## SpinLock

- SpinLock - synchronization primitive that spins while it waits to acquire a lock.
- On multicore computers, when wait times are expected to be short and when contention is minimal, SpinLock can perform better than other kinds of locks
- use SpinLock only when you determine by profiling that the System.Threading.Monitor method or the Interlocked methods are significantly slowing
- Remember to use false in exit method to provides the best performance true is used on IA64 architectures to use the memory fence, which flushes the buffers.

```
static SpinLock spinLock = new SpinLock();
static void ThreadWithSpinLock()
{
    bool lockTaken = false;
    for (int ii = 0; ii < countFor; ii++)
    {
        lockTaken = false;
        spinLock.Enter(ref lockTaken);
        {
            counter++;
        }
        spinLock.Exit(false);
    }
}
```

# Critical section

- Synchronization by lock but data are chunked with bigger portions. Lock isn't called so many times.

```
static void ThreadWithLock2()
        {
            for (int ii = 0; ii < 1000; ii++)
            {
                lock (locker)
                {
                    for (int jj = 0; jj < 1000; jj++)
                        counter++;
                }
            }
        }
```

# Critical section

- To sum up

```
--------------- Unsafe thread increment --------------
All unsafe threads finished in 1221 ms counter = 24268196
--------------- lock thread increment --------------
All lock threads finished in 5857 ms counter = 100000000
--------------- Monitor thread increment --------------
All Monitor threads finished in 6363 ms counter = 100000000
--------------- Interlocked thread increment --------------
All Interlocked threads finished in 3504 ms counter = 100000000
--------------- mutex thread increment --------------
All mutex threads finished in 4822 ms counter = 1000000   t x 100
--------------- semaphore thread increment --------------
All semaphore threads finished in 4799 ms counter = 1000000  t x 100
--------------- semaphore slim thread increment --------------
All semaphore slim threads finished in 14417 ms counter = 100000000
--------------- SpinLock thread increment --------------
All SpinLock threads finished in 13190 ms counter = 100000000
--------------- lock thread increment with more granulity --------------
All lock 2 threads finished in 286 ms counter = 100000000
```

We can see that rationally dividing the computation is the best

# Context Bound Object

Only in .net framework. Automatic blocking of method calls from one class instance.

```
using System.Runtime.Remoting.Contexts;

[Synchronization]
  public class SafeClass : ContextBoundObject
  {
  }
```

The CLR (Common Language Runtime) ensures that only one thread can call the code of the same object instance at the same time. The trick is that when you create a `SafeClass` object, it creates a proxy object through which the calls to `SafeClass` methods pass.

# Context Bound Object

```
[Synchronization]
class SafeCounter : ContextBoundObject
{
    public int Counter { get; }
    public SafeCounter()
    {
        Counter = 0;
    }

    public void CounterInc()
    {
        Counter++;
    }
}

static SafeCounter sf = new SafeCounter();
static void ThreadWithoutProtection()
    {
    //it is very slow
    for (int ii = 0; ii < 10000; ii++)
            {
            sf.CounterInc();
            }
    }
```

# Context Bound Object

```
static void Main(string[] args)
    {
    Console.WriteLine("--------------- SafeCounter ContextBoundObject
thread increment -------------- ");
    Thread[] tharray = new Thread[100];
    for (int ii = 0; ii < 100; ii++)
        {
        tharray[ii] = new Thread(ThreadWithoutProtection);
        tharray[ii].Name = "Thread " + ii.ToString();
        }
Stopwatch sw = new Stopwatch();
sw.Start();
foreach (var th in tharray)
    {
    th.Start();
    }
foreach (var th in tharray)
    {
    th.Join();
    }
Console.WriteLine($"All ContextBoundObject threads finished in
{sw.ElapsedMilliseconds} ms counter = {sf.Counter}");
            Console.ReadLine();
}
```

# Context Bound Object

Automatic synchronization cannot be applied to protect static fields or classes derived from ContextBoundObject, e.g. Windows Form

You should also remember that it still doesn't solve the problem when we call something like this for the collection:

```
 SafeClass sc = new SafeClass();

...

if (sc.Count > 0)
  {
   sc.RemoveAt (0);
  }
```

# Context Bound Object

if another object is created from a secure object, it is automatically also safe in the same context, unless we decide otherwise using attributes.

```
[Synchronization (SynchronizationAttribute.REQUIRES_NEW)]
public class SomeClassB : ContextBoundObject { ...
```

**NOT_SUPPORTED -** equivalent to not using Synchronized

**SUPPORTED** - Indicates that the class to which this attribute is applied is not dependent on whether the context has synchronization.

**REQUIRED** - (default) Indicates that the class to which this attribute is applied must be created in a context that has synchronization.

**REQUIRES_NEW** - Indicates that the class to which this attribute is applied must be created in a context with a new instance of the synchronization property each time

# Interrupting thread

- `Thread.Interrupt` – interrupts the current wait and causes it to quit

  exception `ThreadInterruptedException`

```
static void InfiniteThread()
      {
      try
        {
        Thread.Sleep(Timeout.Infinite);
        }
      catch (ThreadInterruptedException ex)
        {
        Console.WriteLine("InfiniteThread cought an exception: " + ex.Message);
        }
      Console.WriteLine("InfiniteThread ended normaly");
      }
```

- Keep in mind that interrupting in this way can be dangerous unless you know exactly where you are and clean up.

# Aborting thread

- This is old method and shouldn't be used. If you want end some code in abort way you can run it in separate process and call `Kill`
- `Thread.Abort` – It is similar to Interrupt, except that it throws a `ThreadAbortException` exception and the exception is thrown again at the end of the catch block, unless `Thread.ResetAbort();` is used in the catch block;
- The operation is similar, but with `Interrupt` the thread is only interrupted while waiting, `Abort` can do it anywhere in the execution, even in non our code.

# Thread states

- **`ThreadState`** – a bit combination of the three layers.
- Startup, lock, thread interruption (`Unstarted, Running, WaitSleepJoin, Stopped, AbortRequested`)
- The foreground and background of the thread (`Background, Foreground`)
- Progress in the suspend thread (`SuspendRequested, Suspended` ) used by deprecated methods
- The final state of the thread is determined by the bit sum of the three "Layers". And yes, there may be a thread for example

`Background, Unstarted`

or

`SuspendRequested, Background, WaitSleepJoin`

# Thread states

- Two states are also never used in ThreadState enumeration: **`StopRequested`** i **`Aborted`**
- To make things even more complicated, Running has a value of 0 so comparing x will get us nothing

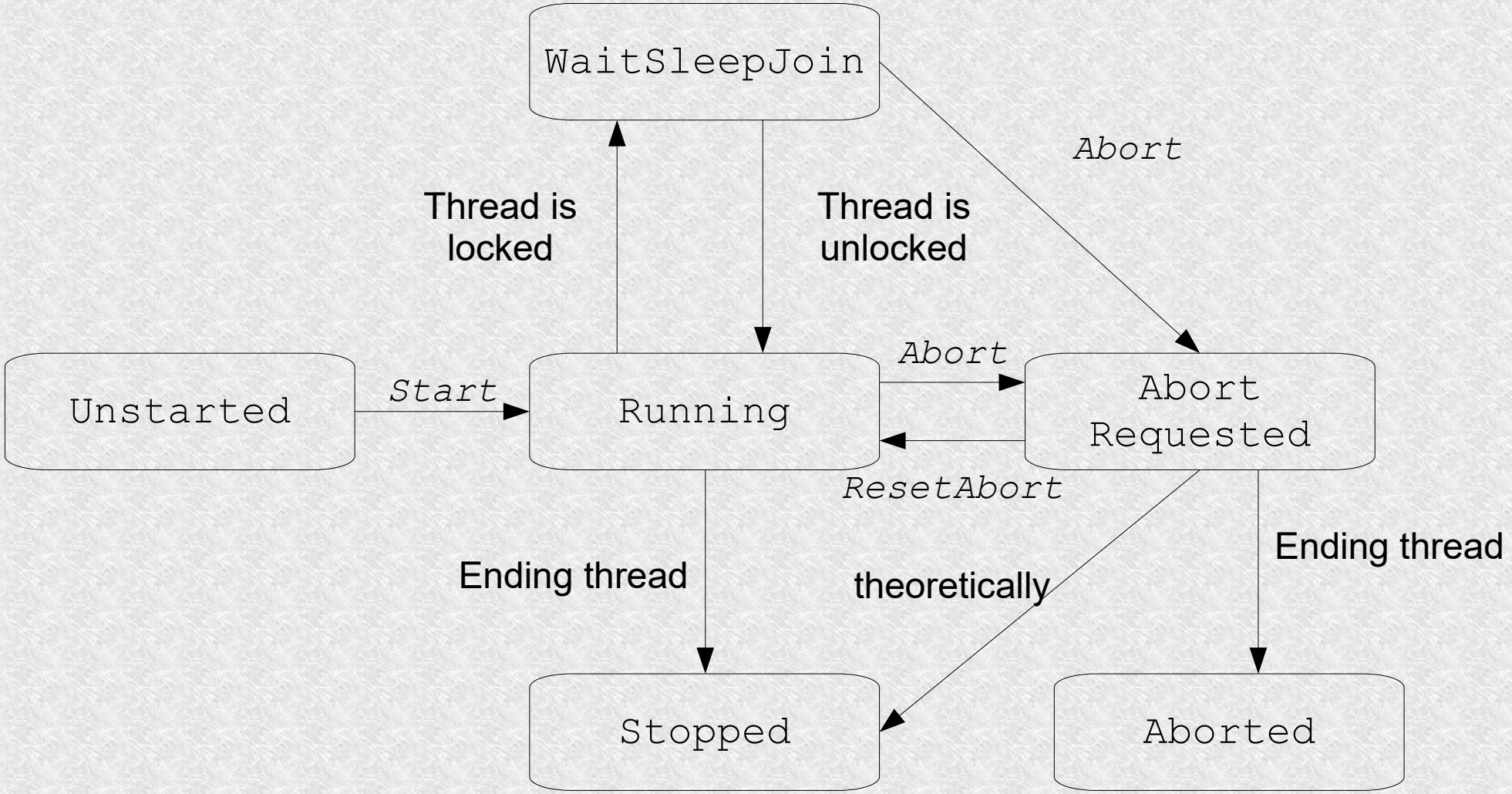
```
if ((t.ThreadState & ThreadState.Running) > 0)...
```

will get us nothing

- You can help with IsAlive, but it only returns false before start and when it ends. When a thread is locked it is also true.
- It's best to write your own method:

```
public static ThreadState SimpleThreadState (ThreadState ts)
    {
    return ts & (ThreadState.Aborted |
                 ThreadState.AbortRequested |
                 ThreadState.Stopped |
                 ThreadState.Unstarted |
                 ThreadState.WaitSleepJoin);
    }
```

# Thread states



Thank You