# Threading in C#

AARHUS UNIVERSITY

AARHUS UNIVERSITY SCHOOL OF ENGINEERING

MICHAEL LOFT
ML@ASE.AU.DK

# C# threads

Concurrency

Amdahl's Law

Multithreading architecture in WIN32

Creating and starting threads*

Suspending, resuming and stopping threads *

Background and foreground threads

Thread priorities

The .Net thread pool

Thread synchronization

* Covered before class

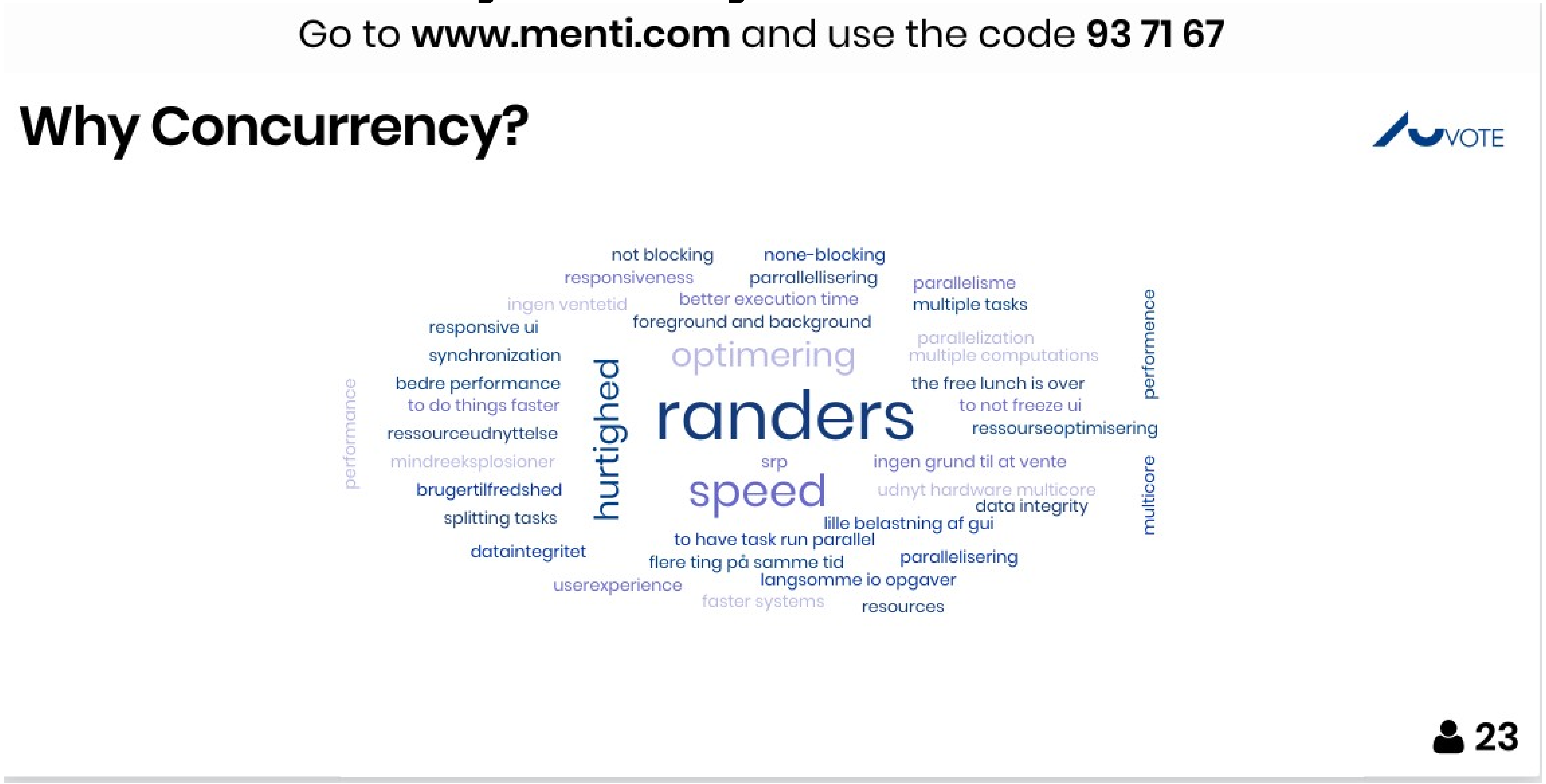
## Concurrency – What?

Things happening at the same time.

(or switching tasks so fast, that we get the illusion of things happening at the same time)

## Concurrency – Why?

## Concurrency – why



Go to **www.menti.com** and use the code **93 71 67**

**Why Concurrency?**

## Concurrency – What?

Things happening at the same time.

(or switching tasks so fast, that we get the illusion of things happening at the same time)

## Different types of concurrency

## Two types of parallelism

- Data Parallelism
  - Concurrency comes from performing the same calculation on different data elements *simultaneously*

- Functional Parallelism
  - Concurrency comes from working on independent functional tasks *simultaneously*

## The three flavors of functional Parallelism

Minimizing latency in reacting to events

Preventing slow I/O from blocking the main thread

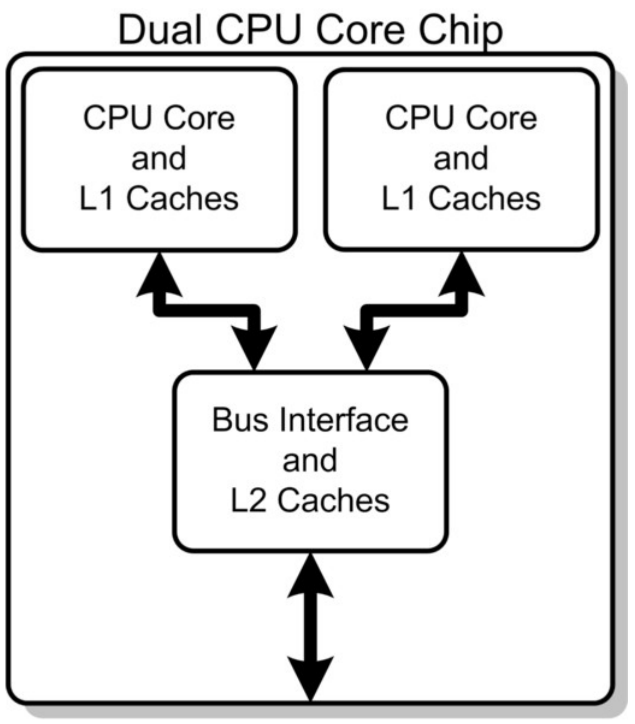
Maximizing throughput on multicore systems

# The future belongs to Multi-core CPUs

Increasing the CPU frequency is hard

Performance improvements → multi-core CPU designs.

In Herb Sutters words:
The free lunch is over!
Multicore HW requires concurrent SW!



## Ahmdal’s law for parallelization

## Limits on performance gain through parallelization

What limits performance gain through parallelization?

- Size of the part of the program that can be parallelized
- Number of CPU cores
- Context switching (not considered by Ahmdal’s law!)

Which is most impeding to performance gain?

Ahmdal’s law…

- provides a *theoretical upper bound* on overall system speed-up when part of the system is parallelized using multiple processors

# Amdahl's Law for parallelization

P = fraction of the algorithm that can be parallelized

N = number of CPU cores times.

$$Overall\,speedup = S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

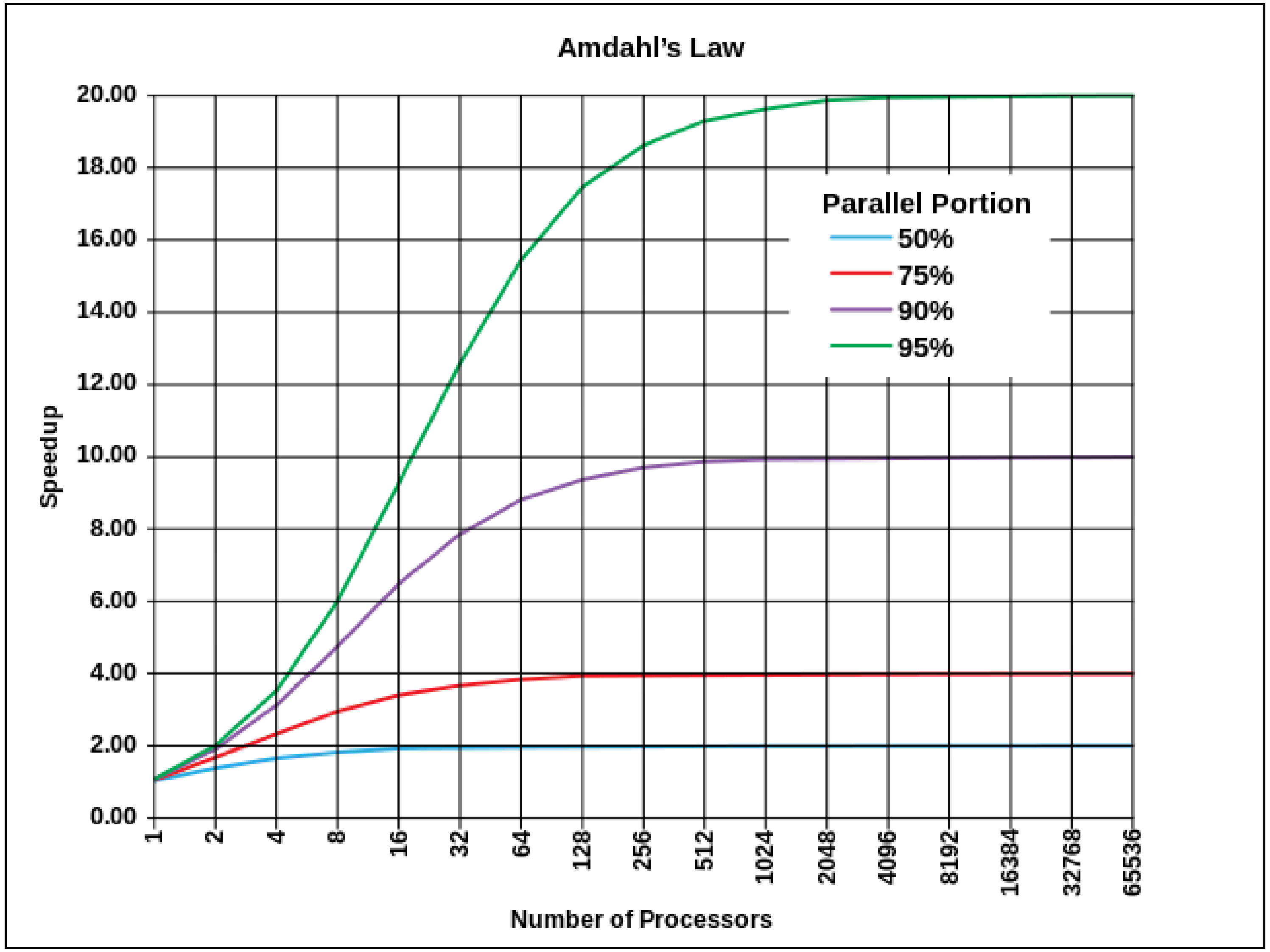
Examples

30% of an algorithm may be parallelized on a quad-core architecture (P = 0.3, N=4) → S(N) = 1.290 *(performance gain 29%)*

90% of an algorithm may be parallelized on a dual-core architecture (P=0.9, N=2) → S(N) = 1.818 *(performance gain 82%)*

As N→∞, S(N) →?

## Amdahl's Law – cores ain’t all!

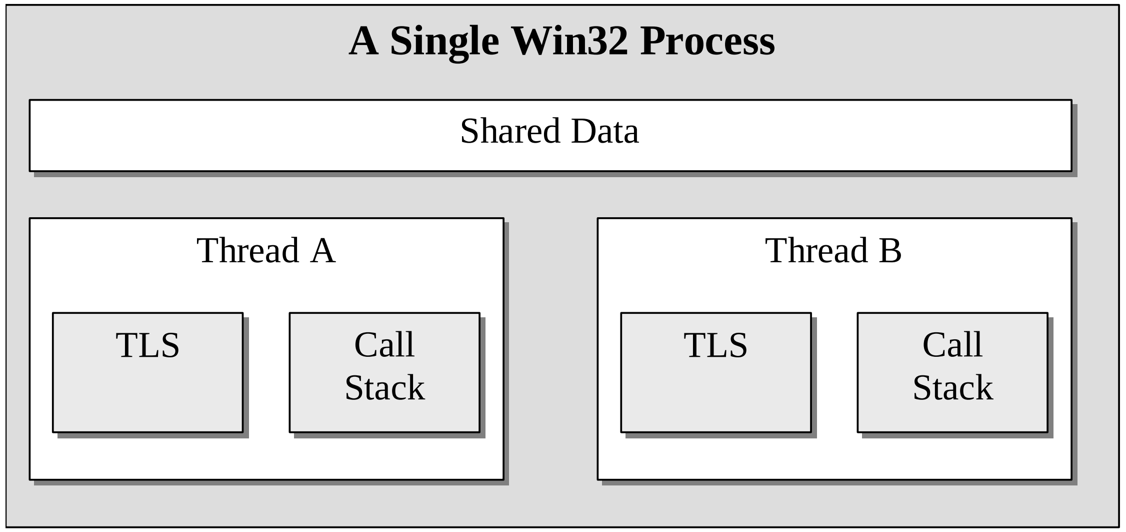


## WIN32 architecture

# Multithreading architecture in WIN32

Data in one process cannot be accessed from another process.

Threads in a process share data.



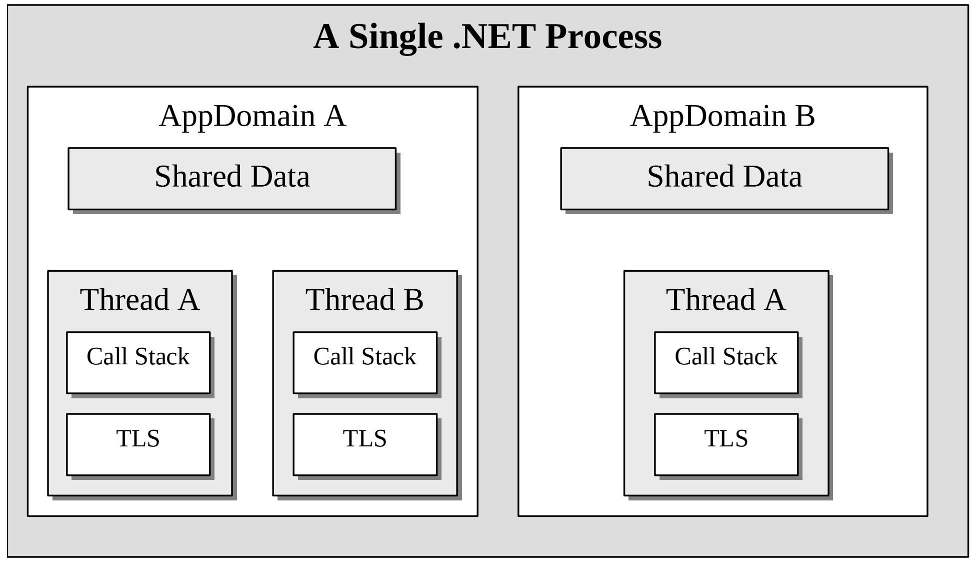
TLS: Thread Local Storage

## Multithreading architecture in .Net

*Application domains* provide isolation, unloading, and security boundaries for executing managed code within a process.

Multiple application domains can run in a single process

- No one-to-one correlation between application domains and threads.
- Several threads can belong to a single application domain
- At any given time, a thread executes in a single application domain – but may change app domains over time



## Thread Basics

## Multithreading resources

Namespace **`System.Threading`**

- Classes and interfaces that enable multithreaded programming
- **`Mutex`**, **`Monitor`**, **`AutoResetEvent`**, **`Thread`**, **`ThreadPool Timer`**, …

Namespace **`System.Threading.Tasks`**

- The **`Task`** asynchronous operation abstraction (and friends)

Namespace **`System.Collections.Concurrent`**

Thread-safe collection classes to use when multiple threads access the same collection concurrently

Namespace **`System.Windows.Threading`**

- Types to support the WPF threading system, primarily **`Dispatcher`** and friends

## Creating and starting threads

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();
      Thread myThread =
                 new Thread(work.DoLotsOfWork);

      myThread.Start();
      System.Console.ReadKey();
    }
}
```

```
public class LotsOfWork
{
    public void DoLotsOfWork()
    {
        for (int i = 0; i < 1000; i++)
        {
            // TODO: add a lot of work here..
            Console.WriteLine("iteration: {0}", i);
        }
    }
}
```

# Passing parameters - Properties

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();
      Thread myThread =
               new Thread(work.DoLotsOfWork);
      work.NumberOfIterations = 500;
      myThread.Start();
      System.Console.ReadKey();
    }
}
```

```
public class LotsOfWork
{
    public int NumberOfIterations { get; set; } = 0;

    public void DoLotsOfWork()
    {
        for (int i = 0; i < NumberOfIterations; i++)
        {
            // TODO: add a lot of work here..
            Console.WriteLine("iteration: {0}", i);
        }
    }
}
```

# Passing parameters – To the Start() method

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();

      Thread myThread = new Thread(work.DoLotsOfWork);
      myThread.Start(500);

      // --- OR ---
      Thread myThread2 = new Thread(() => work.DoLotsOfWork(300));
      myThread2.Start(); // Specify parameter in lambda expression

      System.Console.ReadKey();
    }
}
```

```
public class LotsOfWork
{
    public void DoLotsOfWork(object parameter)
    {
        int numberOfIterations = (int) parameter;
        for (int i = 0; i < numberOfIterations; i++)
        {
            // TODO: add a lot of work here..
            Console.WriteLine("iteration: {0}", i);
        }
    }
}
```

Only one parameter,
must be of type `object`
and casted in thread func

## Pausing a thread

```
public class LotsOfWork
{
    public void DoLotsOfWork(object parameter)
    {
        int numberOfIterations = (int) parameter;
        for (int i = 0; i < numberOfIterations; i++)
        {
            // TODO: add a lot of work here..

            Console.WriteLine("iteration: {0}", i);

            Thread.Sleep(50); // 50 milliseconds
        }
    }
}
```

The thread will sleep for ***at least*** 50 milliseconds.

## Yield

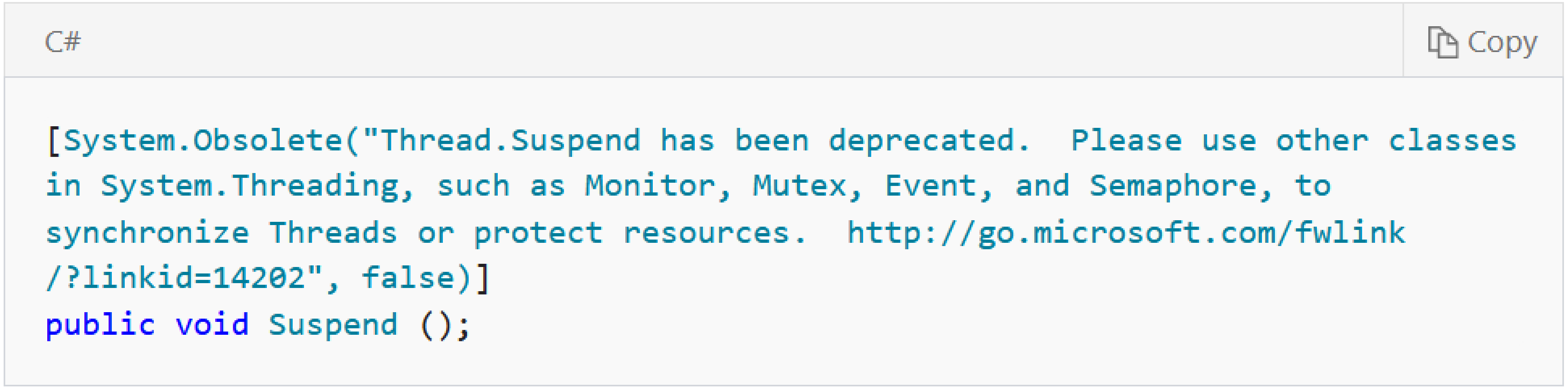
```
public class LotsOfWork
{
    public void DoLotsOfWork(object parameter)
    {
        int numberOfIterations = (int) parameter;
        for (int i = 0; i < numberOfIterations; i++)
        {
            // TODO: add a lot of work here..

            Console.WriteLine("iteration: {0}", i);

            Thread.Sleep(0); // Yield
        }
    }
}
```

Sleep(0) means that the thread **yields**.

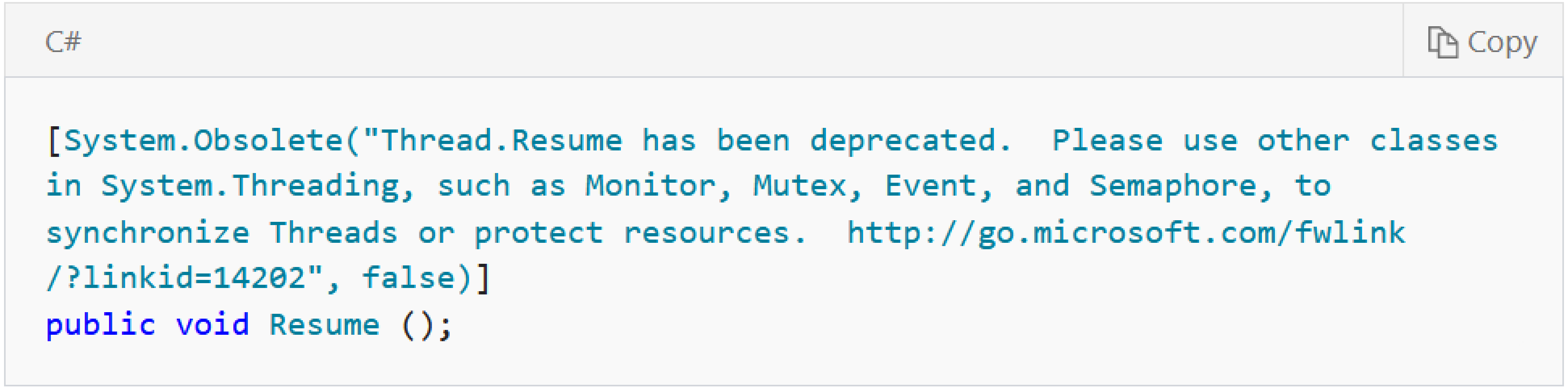
If another thread is ready to run, it will run.

If no other thread is ready to run, the thread that yielded will continue.

## Don’t use Thread.Suspend and Thread.Resume



```csharp
[System.Obsolete("Thread.Suspend has been deprecated.  Please use other classes
in System.Threading, such as Monitor, Mutex, Event, and Semaphore, to
synchronize Threads or protect resources.  http://go.microsoft.com/fwlink
/?linkid=14202", false)]
public void Suspend ();
```



```csharp
[System.Obsolete("Thread.Resume has been deprecated.  Please use other classes
in System.Threading, such as Monitor, Mutex, Event, and Semaphore, to
synchronize Threads or protect resources.  http://go.microsoft.com/fwlink
/?linkid=14202", false)]
public void Resume ();
```

https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.suspend?view=netframework-4.7
https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.resume?view=netframework-4.7

## When does the program exit?

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();

      Thread myThread =
           new Thread(work.DoLotsOfWork);
      myThread.Start();
      System.Console.WriteLine(”Goodbye from 
main");
      System.Console.ReadKey();
    }
}
```

```
public class LotsOfWork
{
    public void DoLotsOfWork()
    {
        for (int i = 0; i < 1000; i++)
        {
            // TODO: add a lot of work here..
            Console.WriteLine("iteration: {0}", i);
        }
    }
}
```

# Waiting for other threads to complete (join)

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();

      Thread myThread =
               new Thread(work.DoLotsOfWork);
      myThread.Start(50);

      myThread.Join();
      System.Console.WriteLine("Hello from main");

      System.Console.ReadKey();
    }
}
```

The Join() method blocks the caller until the thread on which join was called completes.

## Waiting for other threads to complete (join)

```
class Program

{

  static void Main(string[] args)

  {

      LotsOfWork work = new LotsOfWork();


      Thread myThread =

                new Thread(work.DoLotsOfWork);

      myThread.Start(50);


      myThread.Join(1000); // continue if not
joined after 1000 ms

      System.Console.WriteLine("Hello from main");


      System.Console.ReadKey();

    }

}
```

But you can specify a timeout.

# Joining on threads using `WaitHandle`

```
public static void Main()
{
    WaitHandle[] handles = new WaitHandle[] // The thread wait handles
    {
        new AutoResetEvent(false), new AutoResetEvent(false)
    };

    Thread workerThread0 = new Thread(DoWorkWithHandle);
    Thread workerThread1 = new Thread(DoWorkWithHandle);

    workerThread0.Start(handles[0]);
    workerThread1.Start(handles[1]);

    WaitHandle.WaitAll(handles);

    System.Console.WriteLine("Join on WaitHandle complete");
}


public static void DoWorkWithHandle(object hdl)

{

    var handle = (AutoResetEvent) hdl;

    // Do meaningful work

    handle.Set();

}
```

## Foreground and Background Threads

A thread’s foreground/background status has *no* relation to its priority or allocation of execution time

### Foreground threads

Main thread (“GUI thread”) and all threads you create explicitly are by default foreground threads

Application does not terminate until all foreground threads have terminated

### Background threads

Threads may be “pushed” to background threads:

```
myThread.IsBackground = true;
```

Applications may terminate even if background threads are still running!

# Background and foreground threads

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();
      Thread myThread =
              new Thread(work.DoLotsOfWork);
      myThread.IsBackground = true;
      myThread.Start(50);
      System.Console.ReadKey();
    }
}
```

The program will now terminate instantly, when a key is pressed, because myThread is a background thread.

# Stopping a thread - gracefully

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();
      Thread myThread =
           new Thread(work.DoLotsOfWork);
      myThread.Start();
      System.Console.ReadKey();
      work.ShallStop = true;
    }
}
```

```
public class LotsOfWork
{
  public bool ShallStop { get; set; } = false;
  public void DoLotsOfWork()
  {
     int iteration = 0;
     while (!ShallStop)
     {
         Console.WriteLine("iteration: {0}",
iteration);
         Thread.Sleep(50); // 50 milliseconds
         iteration++;
     }
     Console.WriteLine("Thread is done!");
  }
}
```

## Aborting a thread (not so graceful)

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();
      Thread myThread =
                 new Thread(work.DoLotsOfWork);
      myThread.Start();
      System.Console.ReadKey();
      if (myThead.IsAlive) myThread.Abort();
    }
}
```

```
public class LotsOfWork
{
  public bool ShallStop { get; set; } = false;
  public void DoLotsOfWork()
  {
     int iteration = 0;
     while (!ShallStop)
     {
         Console.WriteLine("iteration: {0}",
iteration);
         Thread.Sleep(50); // 50 milliseconds
         iteration++;
     }
     Console.WriteLine("Thread is done!");
  }
}
```

## Aborting a thread (not so graceful)

```
class Program
{
  static void Main(string[] args)
  {
      LotsOfWork work = new LotsOfWork();

      Thread myThread = new
Thread(work.DoLotsOfWork);
      myThread.Start();

      System.Console.ReadKey();
      if (myThead.IsAlive) myThread.Abort();
    }
}
```

Avoid Abort() if you can.

Abort() throws an exception on the thread.

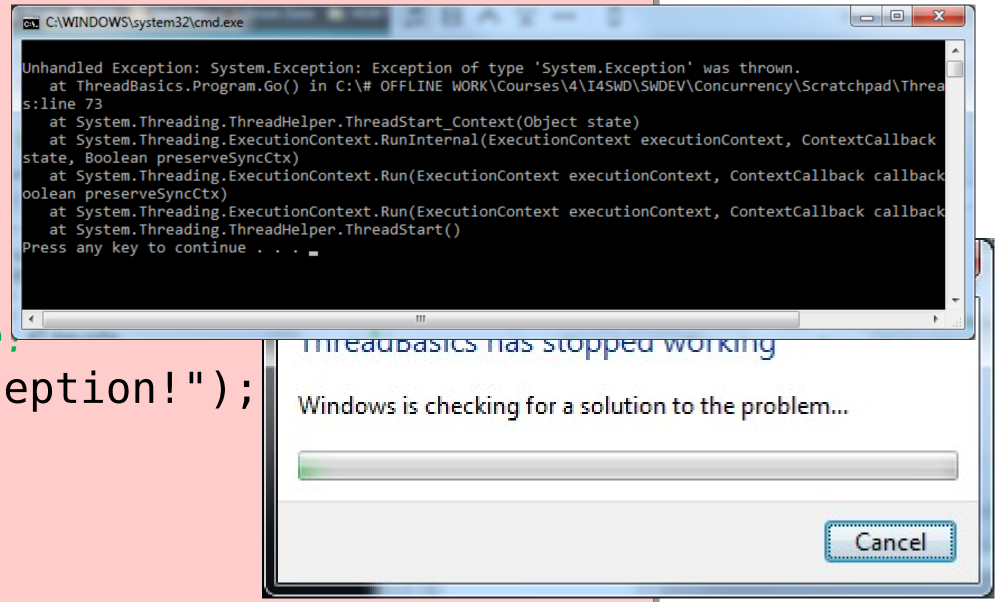
You don’t know what the thread was doing, when it was aborted.

The exception can be caught by the thread (and the thread keeps running…).

# Threads and exception handling

Exceptions thrown in child thread is not caught by creating thread, regardless of `try`/`catch`/`finally` in scope of creating thread

```
static void Main()
{
  try
  {
    new Thread (Go).Start()
  }
  catch (Exception ex)
  {
    // We'll never get here!
    Console.WriteLine ("Exception!");
  }
}

static void Go()
{
  throw new Exception(); // Will terminate
application
}
```



```
Unhandled Exception: System.Exception: Exception of type 'System.Exception' was thrown.
   at ThreadBasics.Program.Go() in C:\# OFFLINE WORK\Courses\4\I4SWD\SWDEV\Concurrency\Scratchpad\Threa
s:line 73
   at System.Threading.ThreadHelper.ThreadStart_Context(Object state)
   at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback
state, Boolean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback
oolean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback
   at System.Threading.ThreadHelper.ThreadStart()
Press any key to continue . . .
```

ThreadBasics has stopped working

Windows is checking for a solution to the problem...

# Threads and exception handling

Correct exception handling in child thread:

```
static void Main()
{
  new Thread(BetterGo).Start();
  Console.ReadKey();
}

static void BetterGo()
{
  try
  {
    throw new Exception(“OH NO!”); // NullReferenceException -
will get caught below
  }
  catch (Exception ex)
  {
    Console.WriteLine("Exception! " + ex.Message);
  }
}
```

# Thread priorities

Manipulate thread priority through `Thread.ThreadPriority`

```
// Example
myThread.Priority = ThreadPriority.Highest;
```

To do real-time work, also elevate process priority using
`System.Diagnostics.Process`

```
// Example
System.Diagnostics.Process.GetCurrentProcess().PriorityClass =
      System.Diagnostics.ProcessPriorityClass.High;
```

Understand what you’re doing – manipulating priorities may cause all sorts of (not-so-)funny problems

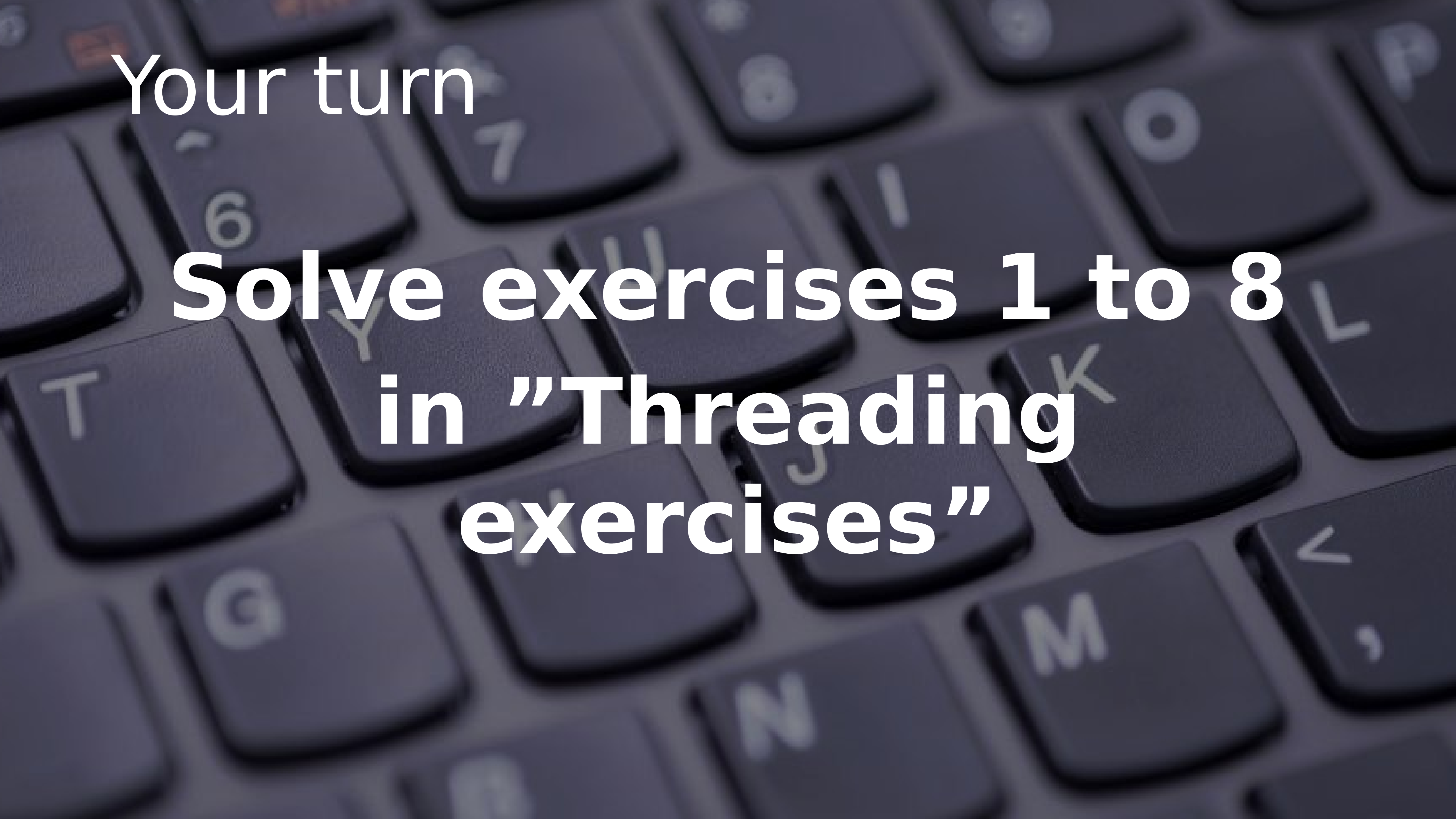
Priority inversion, livelocks, starvation, …

## Thread priorities - Danger, beware!

The .NET scheduler uses thread priorities when deciding which thread to run.

You almost never want to mess with this. So don’t!

## Your turn



**Solve exercises 1 to 8 in ”Threading exercises”**

## Thread Pooling

## Inefficient use of threads

Declaring new threads is easy in code – too easy, perhaps?

Injudicious use of threads incurs an overhead in time and memory

1 MB memory per thread (primarily stack)

200.000 cycles to create a thread, 100.000 cycles to retire it

6.000-8.000 cycles to switch between threads. Context switching → overhead

Many threads → ineffective use of cache

## Thread Pooling

*Thread Pooling* provides a solution

- A bounded number of threads are declared *a priori*
- Threads are recycled
- Work items can be queued on the thread pool -> fine-grained multithreading possible without performance penalty

The Common Language Runtime (CLR) provides each application with a Thread Pool

Some caveats

- You cannot set the name of a thread from the TP
- Threads from the TP are always background threads

## Using the Thread Pool

Using the Thread Pool:

- Calling `ThreadPool.QueueUserWorkItem()`
- Using an asynchronous delegate
- Using the Task Parallel Library (TPL)

# Using ThreadPool.QueueUserWorkItem()

```
static void Main(string[] args)
{
    for (var i = 0; i < 1000; i++)
    {
        var id = i;

    // Execute a new instance of DoWork when a ThreadPool
    // thread becomes available
        ThreadPool.QueueUserWorkItem(DoWork, id);
    }
    Thread.Sleep(60000);
}

static void DoWork(object data)
{
    Console.WriteLine("hello from DoWork({0})", (int)data);
    Thread.Sleep(1000);
}
```

# Using the Thread Pool – asynchronous delegate

```
static void Main(string[] args) {
    Func<string, int> strlenDelegate = CalcStringLength;
    IAsyncResult cookie = strlenDelegate.BeginInvoke("A very long string", new AsyncCallback(AddComplete), null);
    while (!cookie.AsyncWaitHandle.WaitOne(100,true)) {
        Console.WriteLine("Doing some work in Main()!");
    }
    Console.Read();
}
static int CalcStringLength(string text) {
    // Very complicated string length computation goes here!
    return text.Length;
}
//Target of AsyncCallback delegate should match the following pattern
public static void AddComplete(IAsyncResult iftAr) {
    Console.WriteLine("AddComplete() running on thread {0}", Thread.CurrentThread.ManagedThreadId);
    Console.WriteLine("Operation completed.");

    //Getting result
    AsyncResult ar = (AsyncResult)iftAr;
    Func<string, int> fsi = (Func<string, int>)ar.AsyncDelegate;
    int result = fsi.EndInvoke(iftAr);

    Console.WriteLine("String length: {0}", result);
}
```

# Using Task Parallel Library (TPL)

In .Net 4.0, TPL and the Task abstraction was introduced

- Tasks are fast, convenient and flexible to use
- A task should be considered a small, isolated unit of work

```
public static void Main()
{
    Task myTask = Task.Run((Action) TaskFunc);
    myTask.Wait();
    Console.WriteLine("TaskFunc complete");
}


static void TaskFunc()
{
    Console.WriteLine("Hello from TaskFunc");
    Thread.Sleep(250);
}
```

# Async / Await

In .Net 4.5, async and await keywords where introduced

```
async Task<int> AccessTheWebAsync() {
    HttpClient client = new HttpClient();

    Task<string> getStringTask =

client.GetStringAsync("http://msdn.microsoft.com");

    DoIndependentWork();

    string urlContents = await getStringTask;

    return urlContents.Length;
}
```

## Basic Synchronization Mechanisms

## Basic synchronization

Simple blocking: `Thread.Sleep()`, `Thread.Join()`, `Task.Wait()`

Locking constructs:

- Exclusive: `lock()`, `Monitor.Enter()` and `Monitor.Exit()`, `Mutex`
- Non-exclusive : `Semaphore`

Signalling constructs: `Monitor.Wait()`, `xxxEvent`

# Exclusive locking using Monitor

- The `Monitor` prevents collisions if used correctly, i.e. by all users of the shared resource

```
Monitor.Enter(o);
try{
  //critical section
  ...
}
finally{
  Monitor.Exit(o);
}
```

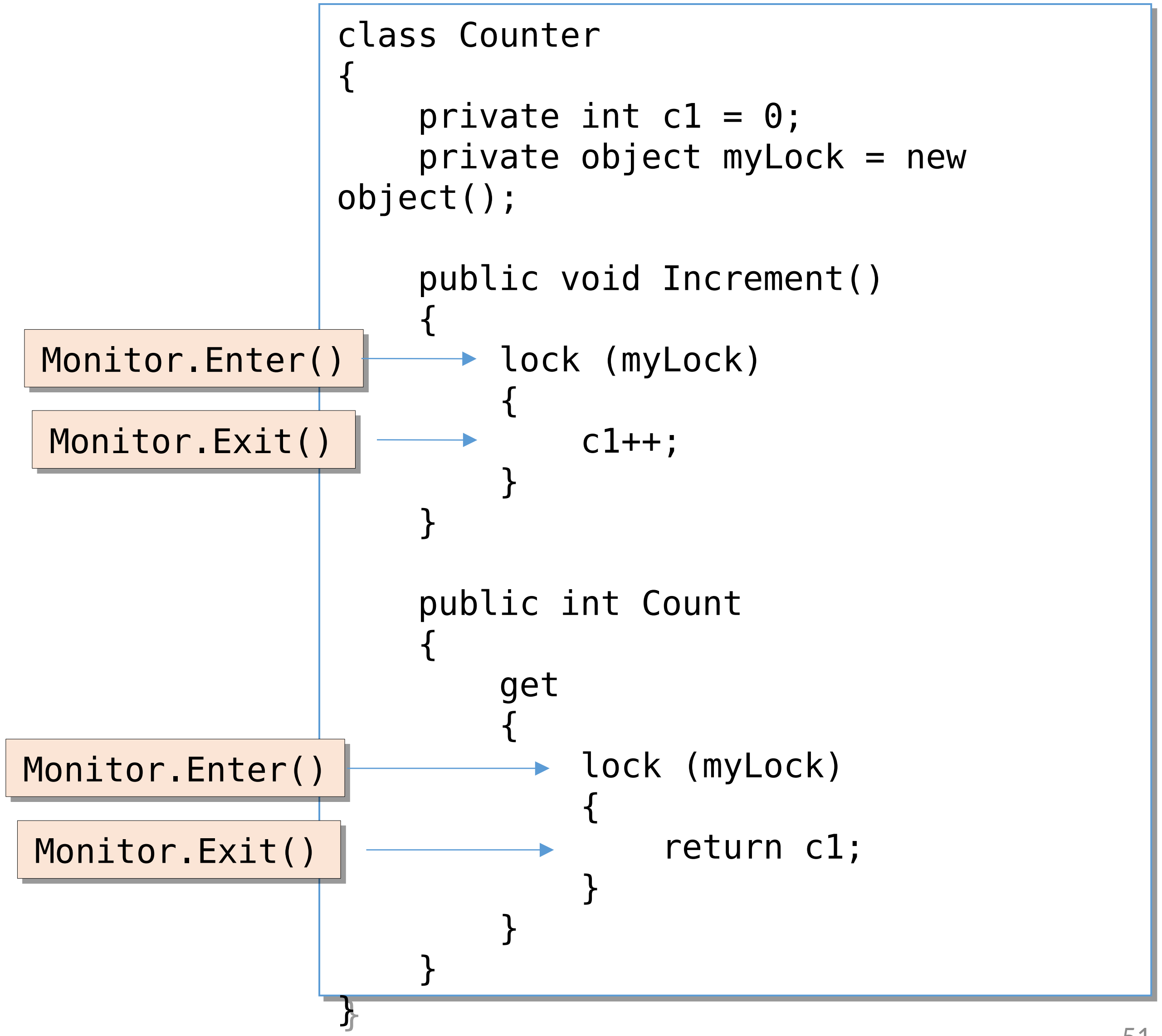
- `Monitor.Enter()` only blocks other threads – the calling thread may re-enter the monitor
- Monitors can only lock reference types – value types are *boxed* (this constitutes a synchronization error!)
- Monitors can only lock within same application domain

# Exclusive locking using lock()

The C# `lock()`-statement is shorthand for using monitors

Best practice is to create a lock object.

You can lock on any object, including `this`, but then others can lock on the same object as well and prevent concurrency.



```
class Counter
{
    private int c1 = 0;
    private object myLock = new
object();

    public void Increment()
    {
        lock (myLock)
        {
            c1++;
        }
    }

    public int Count
    {
        get
        {
            lock (myLock)
            {
                return c1;
            }
        }
    }
}
```

## Exclusive locking using Mutex

A `Mutex` can lock across application domains/processes

`Mutex` locks on itself (not an arbitrary `object`)

`Mutex.WaitOne()` acquires the mutex
Blocks until mutex is available or optional timeout elapses

`Mutex.ReleaseMutex()` releases the mutex
Only owning thread may release the mutex – otherwise an `AplicationException` is thrown

Named mutexes can span processes

## Exclusive locking using Mutex

```
class SynchDemo
{
  private static int _c1 = 0;
  private static Mutex _mutex = new Mutex();

  public static void Main(string[] args)
  {
    var tasks = new Task[] { Task.Run((Action) IncC1), Task.Run((Action) 
IncC1) };
    Task.WaitAll(tasks);
  }

  private static void IncC1()
  {
    if (!_mutex.WaitOne(TimeSpan.FromSeconds(3)))
    {
        Console.WriteLine("Another task is holding C1 - bye!");
        return;
    }
    Console.WriteLine("Press a key to increment C1");
    Console.ReadKey(true);
    ++_c1;
    _mutex.ReleaseMutex();
  }
}
```

# Non-exclusive locking using Semaphore

- Windows semaphores are counting semaphores
  - Unnamed semaphore → local to hosting application domain
  - Named semaphore → accessible system-wide

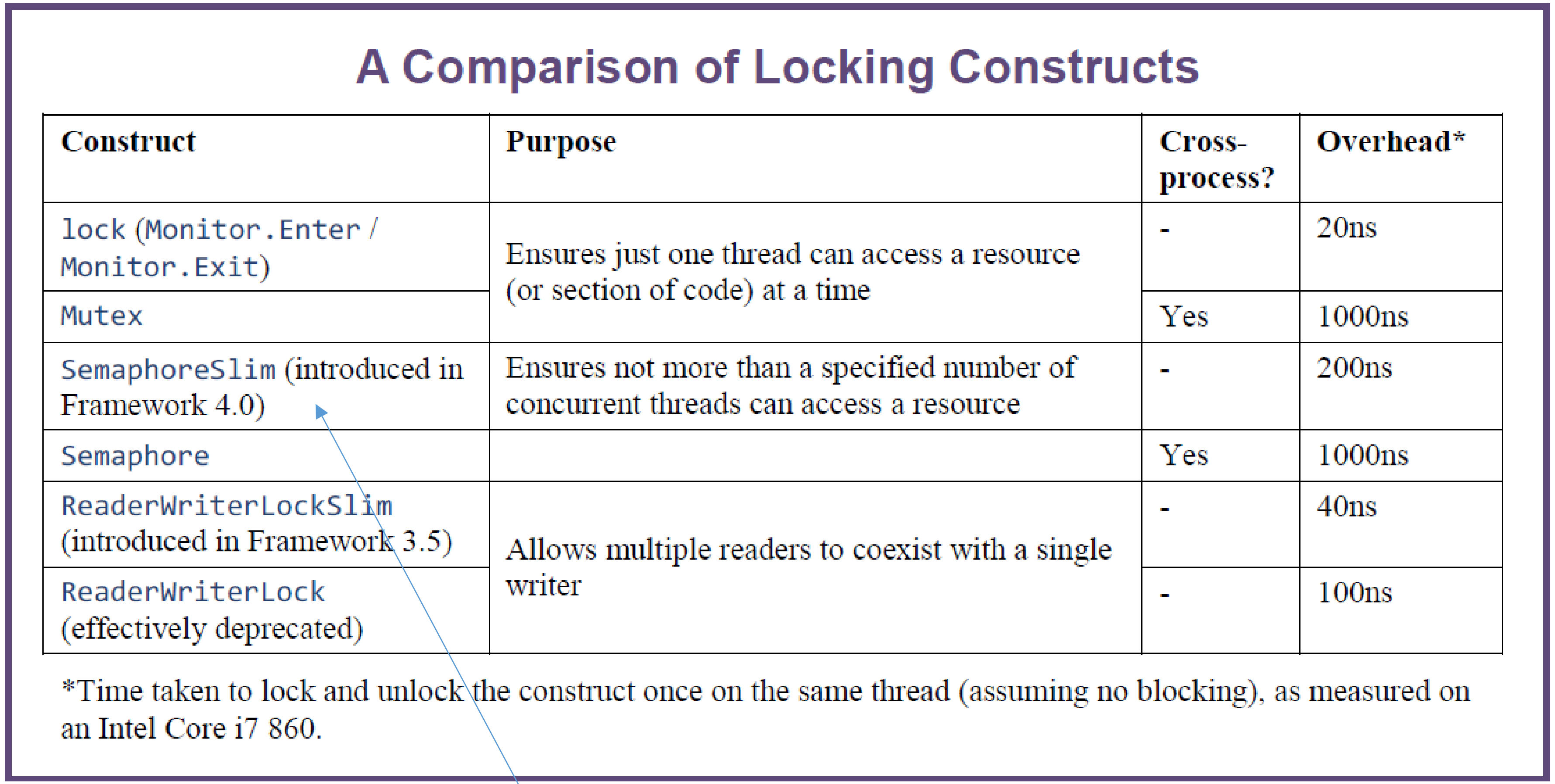
- Enter the semaphore by `Semaphore.WaitOne()`
  - Semaphore > 0 → Entry granted, semaphore decremented
  - Semaphore = 0 → Caller blocked

- Release semaphore by `Semaphore.Release()`
  - No threads waiting → Semaphore incremented
  - Threads waiting → Some thread released (no order)

- Semaphores are thread-agnostic (vs. mutexes)

- Windows semaphores have a max count (!)

# Comparison of Locking Constructs



## A Comparison of Locking Constructs

| Construct | Purpose | Cross-process? | Overhead* |
|---|---|---|---|
| `lock` (`Monitor.Enter` / `Monitor.Exit`) | Ensures just one thread can access a resource (or section of code) at a time | - | 20ns |
| `Mutex` | | Yes | 1000ns |
| `SemaphoreSlim` (introduced in Framework 4.0) | Ensures not more than a specified number of concurrent threads can access a resource | - | 200ns |
| `Semaphore` | | Yes | 1000ns |
| `ReaderWriterLockSlim` (introduced in Framework 3.5) | Allows multiple readers to coexist with a single writer | - | 40ns |
| `ReaderWriterLock` (effectively deprecated) | | - | 100ns |

*Time taken to lock and unlock the construct once on the same thread (assuming no blocking), as measured on an Intel Core i7 860.

Lightweight alternative to Semaphore for use within same process (no naming)

# Signaling with Event Wait Handles

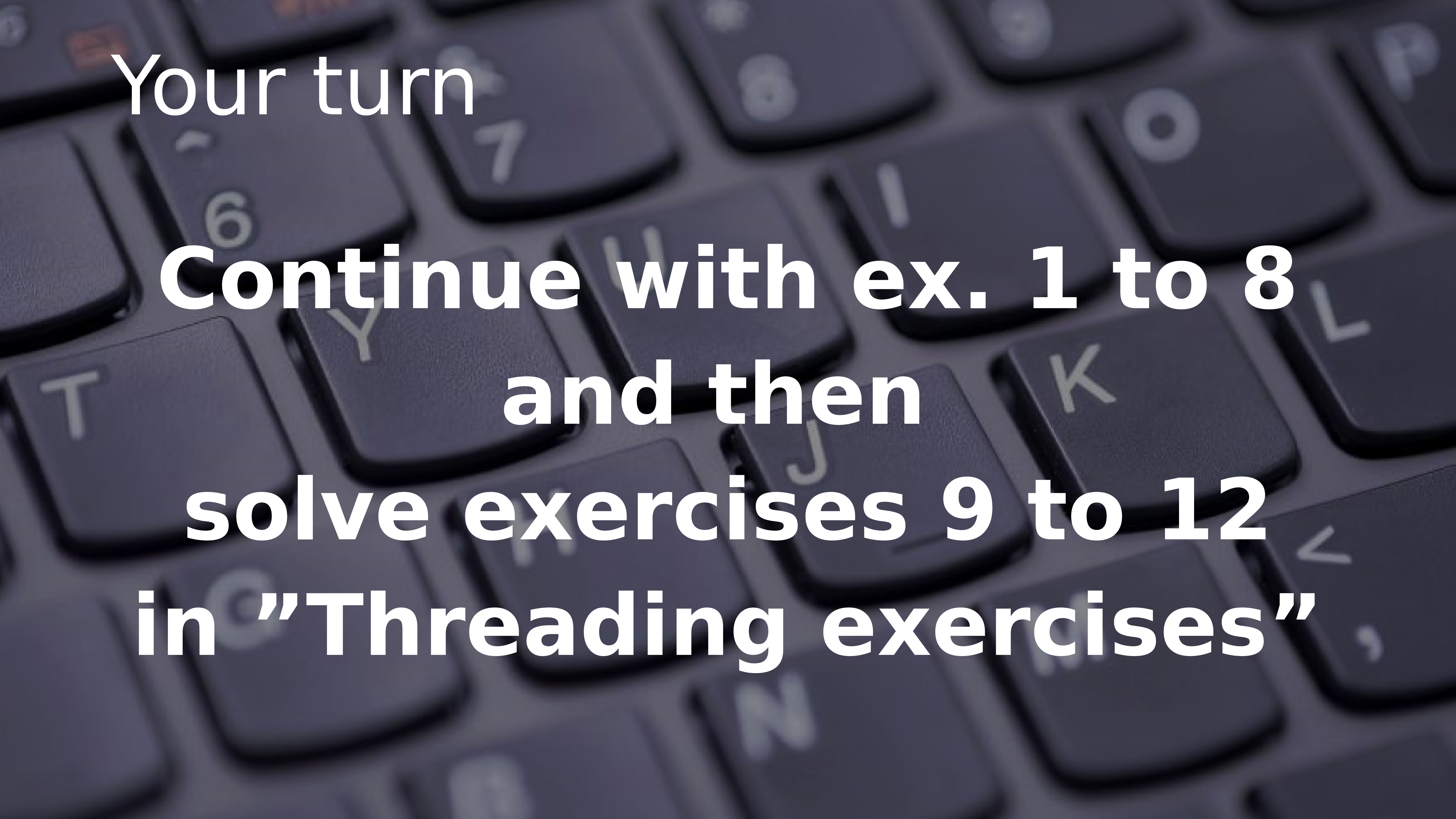
Signaling: When one thread waits until it is signaled by another thread.

Event wait handles are simplest form

Not related to C# `event`

Three flavors:

| | |
|---|---|
| `AutoResetEvent` | `Set()` unblocks waiter once → Think “turnstile” |
| `ManualResetEvent` | `Set()` unblocks waiter until next `Reset()` → Think “gate” |
| `CountdownEvent` | Predetermined number of `Set()` → unblock (from .Net 4.0) |



## Your turn

**Continue with ex. 1 to 8**
**and then**
**solve exercises 9 to 12**
**in ”Threading exercises”**

# References and image sources

Images:

Down the rabbit hole: https://i.pinimg.com/originals/3b/a4/1c/3ba41c16b44edd7d9c5c9faeec965fad.gif

Computer keyboard: http://stockmedia.cc/computing_technology/slides/DSD_8790.jpg

Nuclear explosion: http://www.greenpeace.org/international/en/multimedia/photos/mushroom-cloud/

AARHUS UNIVERSITY