Creating Multi-threaded Applications

Software techniques

Department of
Automation and
Applied Informatics

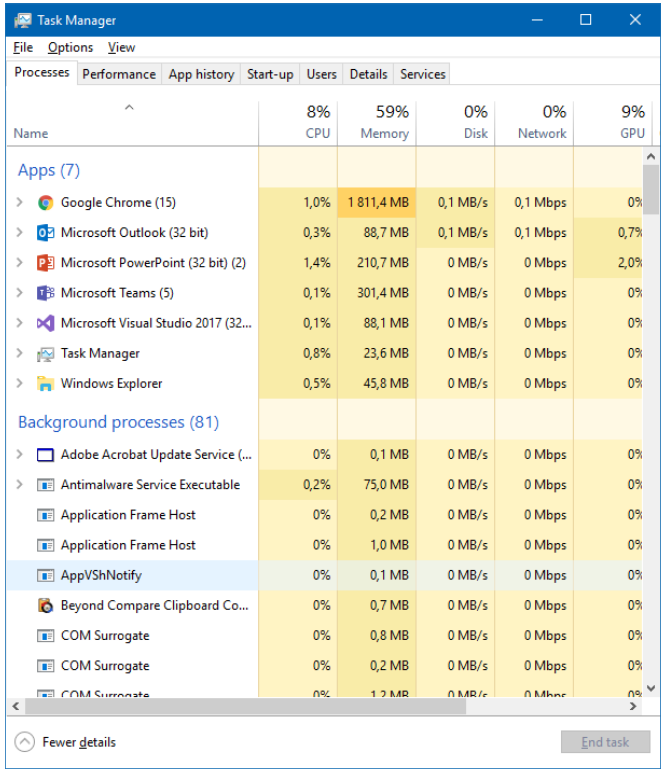
Content

- The basics
  - >Process
  - >Thread
- Handling thread
- Synchronizing threads and processes
- ...

Process

Process

- Application(program) != process
  - > The process is an instance of an application that is loaded into the memory
- What are the parts of a process?
  - > Private address space
  - > System resources
  - > At least one thread



Task Manager
File Options View
Processes Performance App history Start-up Users Details Services
Fewer details
End task

Name	8% CPU	59% Memory	0% Disk	0% Network	9% GPU
Apps (7)					
Google Chrome (15)	1,0%	1 811,4 MB	0,1 MB/s	0,1 Mbps	0%
Microsoft Outlook (32 bit)	0,3%	88,7 MB	0,1 MB/s	0,1 Mbps	0,7%
Microsoft PowerPoint (32 bit) (2)	1,4%	210,7 MB	0 MB/s	0 Mbps	2,0%
Microsoft Teams (5)	0,1%	301,4 MB	0 MB/s	0 Mbps	0%
Microsoft Visual Studio 2017 (32...	0,1%	88,1 MB	0 MB/s	0 Mbps	0%
Task Manager	0,8%	23,6 MB	0 MB/s	0 Mbps	0%
Windows Explorer	0,5%	45,8 MB	0 MB/s	0 Mbps	0%
Background processes (81)					
Adobe Acrobat Update Service (...	0%	0,1 MB	0 MB/s	0 Mbps	0%
Antimalware Service Executable	0,2%	75,0 MB	0 MB/s	0 Mbps	0%
Application Frame Host	0%	0,2 MB	0 MB/s	0 Mbps	0%
Application Frame Host	0%	1,0 MB	0 MB/s	0 Mbps	0%
AppVShNotify	0%	0,1 MB	0 MB/s	0 Mbps	0%
Beyond Compare Clipboard Co...	0%	0,7 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	0,8 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	0,2 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	1,2 MB	0 MB/s	0 Mbps	0%

Process

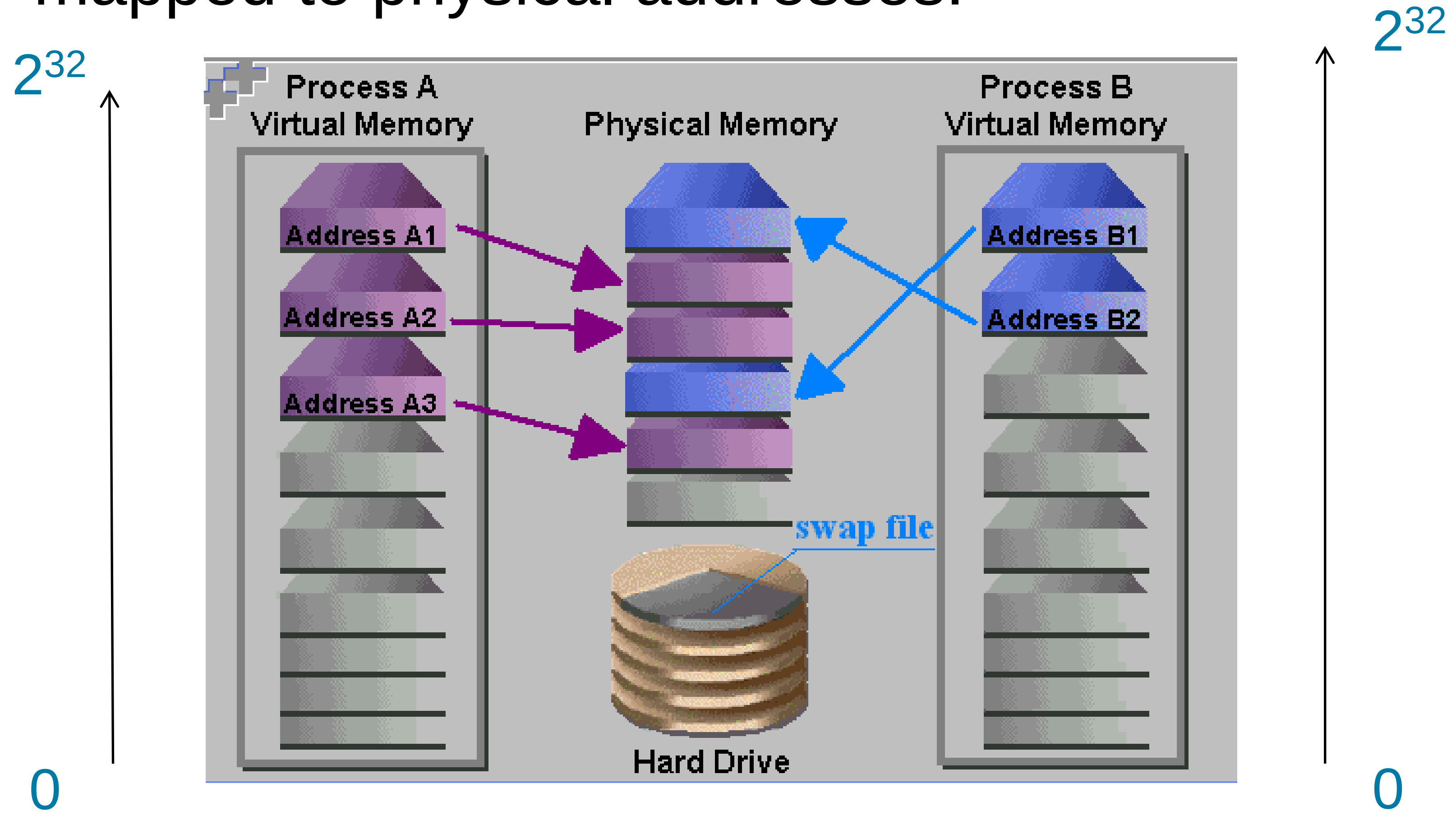
- The process is the unit of **protection**
  - > The thread is the unit of scheduling
- Processes are isolated from each other
  - > They are protected from each other
  - > The OS is protected from the user processes
  - > Ending a process has got no effect on other processes or on the OS
- What is protected? The memory!
  - > Every process has got about 2 GB virtual address space (on a 32 bit OS)
    - – The whole address space in a 32 bit OS is 2^{32} (4 GB) but the OS reserves about 2 GB for itself
  - > Virtual addresses are converted to physical addresses (RAM) by the help of the virtual memory manager

Memory management

- The virtual addresses in the processes are mapped to physical addresses.

2^{32}

2^{32}



Memory management

- On a 32-bit system each process has about 2GB of virtual memory
- It is not assigned to the process by default: **we have to ask for it**!
  - C/C++ global/static variables – Memory is allocated when the application starts running.
  - Local variables – Memory is allocated on the stack
  - malloc, new – Allocates memory dynamically on the heap
- In the background, the memory is allocated by **making API calls to the OS**. API functions e.g.:
  - VirtualAlloc, HeapAlloc, ...
- If we refer to an invalid memory address:
  - „Access violation at address ...” message

```
int* p = 0xfa2343c2;
*p = 21;
```

Creating processes

- API
  - > CreateProcess(...) – starts a new process. One of the parameters is the path for the executable file.
  - > Etc.
- .NET
  - > System.Diagnostics.Process class
    - – Start() – starts a process
    - – Kill() – terminates a process
    - – Process.GetCurrentProcess() – gets the actually executing process

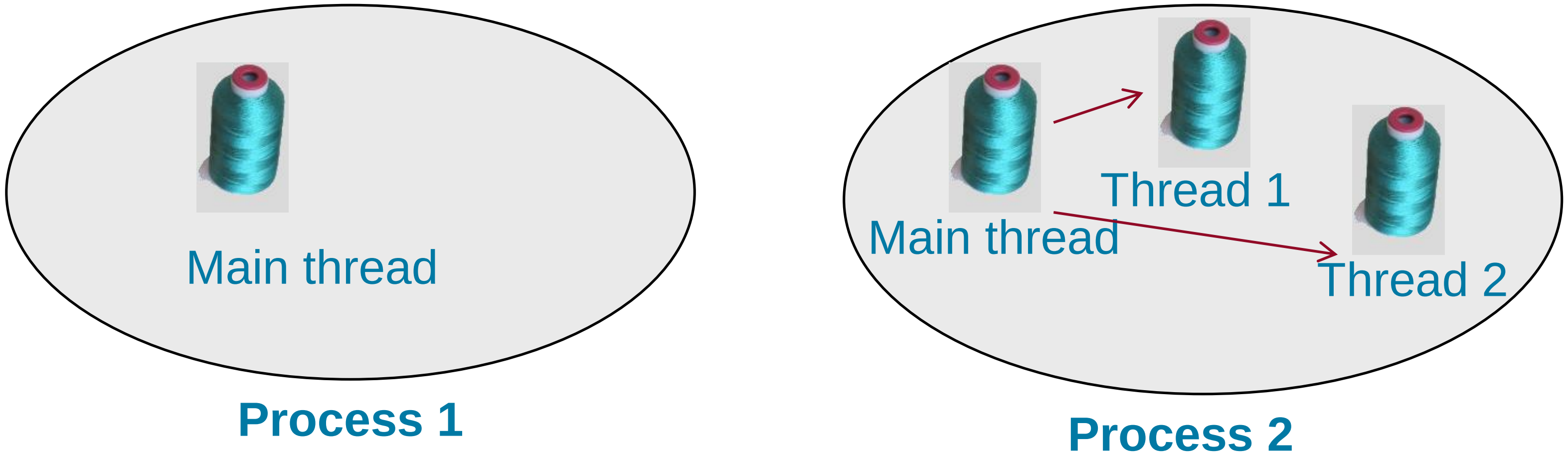
```
Process.GetCurrentProcess().Kill();
```

  - > What does this do?

Multi-threaded applications

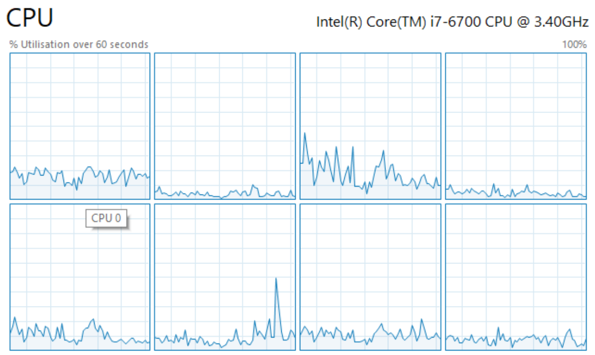
Thread

- The thread is a unit for **scheduling** and **execution**
- Our previous applications were single-threaded
  - When we start a process a thread is automatically created - **main thread**
  - We can start new threads (our application will be multi-threaded).
  - A thread in a process can be considered as a *task*



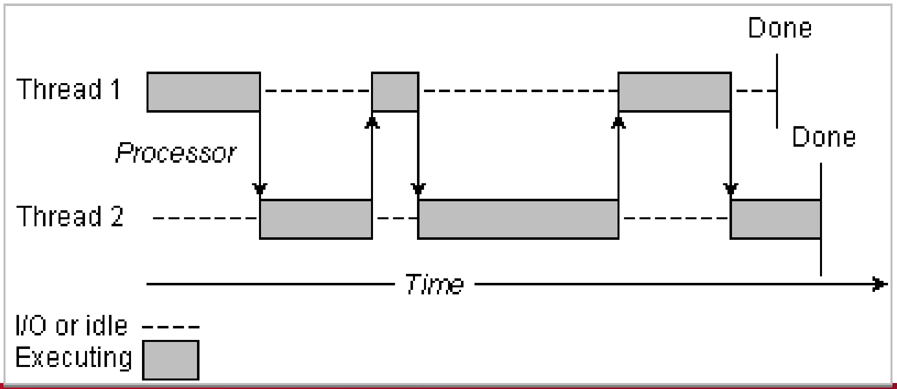
Scheduling the threads

- The virtual addresses in the processes are mapped to physical addresses.
- The OS is responsible for scheduling the threads
- Picks up an executable thread and assigns it to the processor
- One CPU can execute one thread at a certain time (Hyperthreading!)
- Types of scheduling
  - **Non preemptive**: a thread must give up using the CPU itself so that an other thread can use it. E.g. a Windows 3.1. Starving!
  - **Preemptive**: the scheduler can take away the CPU from a thread after a while, if there are higher priority threads or if the dedicated timeframe has elapsed. The threads seem to execute in parallel even if we have got just one processor.



CPU
Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
% Utilisation over 60 seconds
100%
CPU 0

- States of threads
  - Running
  - Suspended
  - Waiting for I/O operation
- Thread priority
  - Will be discussed later…

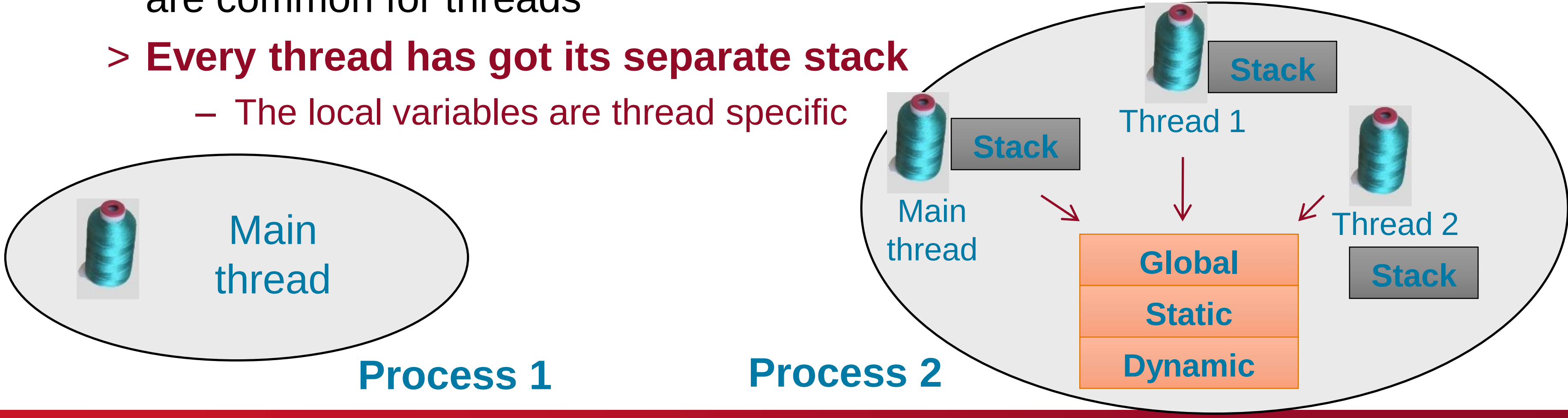


The advantage of multi-threaded applications

- **Better utilization of the CPU**
  - > A thread can execute while an other one is waiting for the completion of an IO operation (e.g.: reading data from the disk) **Demo**
  - > Utilizing all (more than one) the CPU cores
- **Long running blocking operations in GUI (e.g. Windows Forms) applications**
  - > Don't run long operations in the main thread as it will block the GUI
  - > While executing the operation in a background worker thread the main thread (and the GUI) will remain responsive. The main thread will be able to handle the user events. **Demo**
- **Time-critical tasks**
  - > E.g. process control, execute the task in a high priority thread.
- **Server applications (e.g. webserver)**
  - > Multiple clients can use them in parallel. Smaller max response time.
  - > Better CPU utilization

Thread←→ Process

- A thread is more lightweight than a process (reserves less resources),
  - especially in **Windows**, in Linux/Unix this is only mostly true
- **Processes** are isolated: communication between processes is difficult
- **Threads** inside of an application share the same memory space, that makes communication easy
  - The C++ global, static and dynamic variables are common for threads
  - **Every thread has got its separate stack**
    - The local variables are thread specific



Synchronous/Asynchronous execution

- Definitions
  - **Synchronous execution**: The caller is blocked until the execution of the task
  - **Asynchronous execution** : The caller won’t wait for the completion of the task. The caller will somehow be notified about the completion (and the result).

- How to implement asynchronous tasks?
  - I/O operations (file handling, networking, etc.): ***asynchronous I/O execution*** (it is supported by the OS). Based on the interrupt handling of the OS, no threads are needed.
  - Executing the task ***in a new thread.*** This needs more resources, needs a new thread, but it is a general solution and is not only for I/O operations.

Native threading

- Win32 API thread handling functions
  - Important
    - CreateThread – starts a new thread
  - Additional functions (no need to know)
    - **TerminateThread** – terminates a thread, NEVER USE IT
    - SuspendThread – suspends a thread
    - ResumeThread – exits a thread from the suspended state
    - Sleep – makes a thread sleep for a while
    - ...

The basics of using threads in .NET

Starting a thread

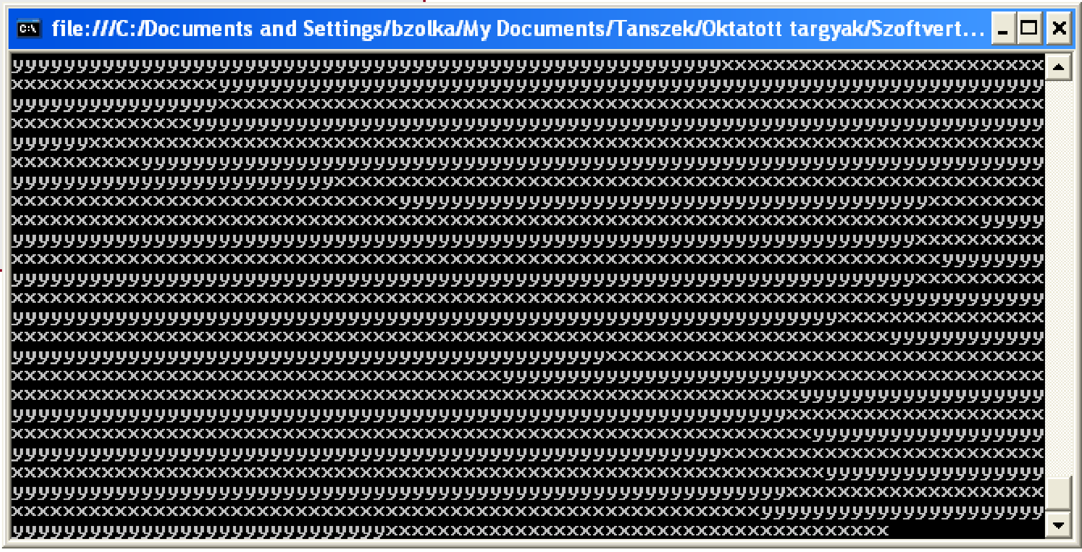
- System.Threading namespace,
- Thread class
- Start a new thread ➡

Starting a thread

Demo1

- A WriteY() is the so called **thread function**

```
class Program
{
    static void Main(string[] args)
    {
        Thread t = new Thread( new ThreadStart(WriteY) );
        t.Start();
        while (true) Console.Write("x");
    }
    static void WriteY()
    {
        while (true) Console.Write("y");
     }
}
```


file:///C:/Documents and Settings/bzolka/My Documents/Tanszek/Oktatott targyak/Szoftvert...

```
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Starting a thread

- ThreadStart delegate

```
Thread t = new Thread( new ThreadStart(WriteY) );
```

- Simpler form since .NET 2.0

```
Thread t = new Thread(WriteY);
```

  - After.NET 2.0 delegates can generally be used this way
- The Start() method will start the thread
  - Until this call the thread is not scheduled for execution
- The thread will finish the execution if the thread function exits
- A parameter can be passed to the thread function
  - ***ParameterizedThreadStart*** delegate

Parameterized thread start

```
class Program
{
    static void Main(string[] args)
    {
      Thread t = new Thread(new ParameterizedThreadStart(WriteAny));
      // This would also work
      // Thread t = new Thread( WriteAny );
      t.Start("Z"); // passing a parameter to the thread method
      while (true)
          Console.Write("x");
    }
    static void WriteAny(object param)
    {
        while (true)
            Console.Write(param);
    }
}
```

Passing parameter by a method reference of an object

- Passing a method reference (non-static) of an object

```
class ThreadClass
{
    private string text;

    public ThreadClass(string text)
    {
        this.text = text;
    }

    public void WriteAny()
    {
        while (true)
            Console.Write(text);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        ThreadClass ts =
            new ThreadClass("Z");
        Thread t =
            new Thread(ts.WriteAny);
        t.Start();
        while (true)
            Console.Write("X");
    }
}
```

Background and foreground threads

Demo

- A created thread will be a **foreground** thread by default
- A process will exit if there is no more running foreground thread, all the foreground threads have finished execution
- Starting a **background** thread

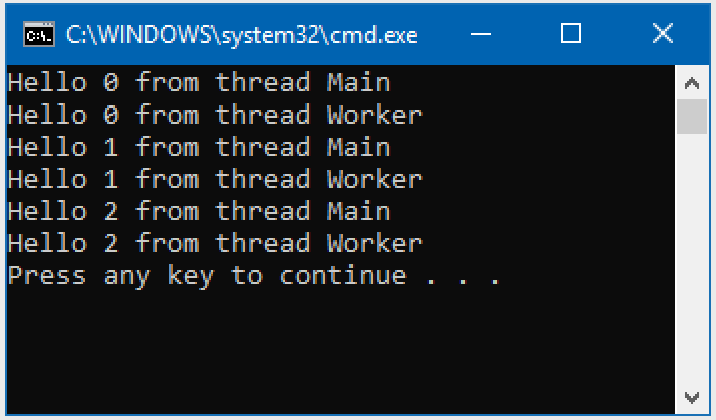
```
static void Main(string[] args)
{
    Thread t = new Thread(ThreadFunc);
    t.Start();
    t.IsBackground = true;
    Console.WriteLine(“I am the main thread and I am done...\n");
}

static void ThreadFunc()
{
    for (int i = 0; i < 5; i++)
    {
        Thread.Sleep(1000);
        Console.WriteLine(“I am a background thread with number: " + i);
    }
}
```

Some useful methods/properties

- ***Thread.GetCurrent ()*** static method– returns the currently executing thread
- ***Name*** property: a name can be assigned to a thread (helps debugging)
- ***Thread.Sleep(int millisec)*** and ***Thread.Sleep(TimeSpan ts)*** static methods– makes the calling thread sleep, **it won't consume CPU**

Example

```
class Program
{
    static void Main()
    {
        Thread.CurrentThread.Name = "Main";
        Thread worker = new Thread(SayHello);
        worker.Name = "Worker";
        worker.Start();
        SayHello();
    }
    static void SayHello()
    {
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine("Hello {0} from thread {1}",
                i, Thread.CurrentThread.Name);
            Thread.Sleep(1000);
        }
    }
}
```

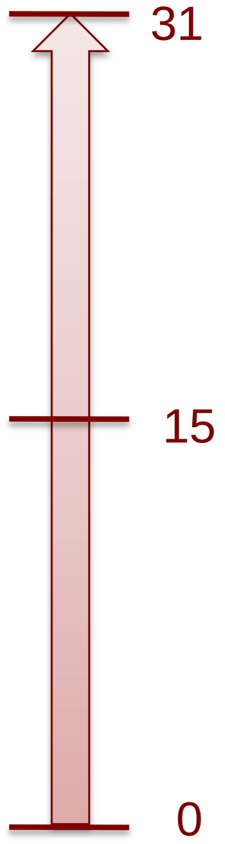


```
Hello 0 from thread Main
Hello 0 from thread Worker
Hello 1 from thread Main
Hello 1 from thread Worker
Hello 2 from thread Main
Hello 2 from thread Worker
Press any key to continue . . .
```

Demo3

Priority of threads

- The effective priority is determined based on the priority of the thread and the priority of the process
- **Process priority**
  - enum ProcessPriority { Idle, BelowNormal, **Normal**, AboveNormal, RealTime}
- **Thread priority**
  - enum ThreadPriority { Lowest, BelowNormal, **Normal**, AboveNormal, Highest }
- The default value is the Normal for both
- The exact implementation is OS-dependent
- In Windows Forms applications the process with the active window may get some boost



```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;
Thread.CurrentThread.Priority = ThreadPriority.Highest;
```

Scheduling threads – not only in .NET

- ***RealTime*** process priority
  - We ask the OS not to take away the CPU from the process
  - Dangerous, we can cut out the OS as well!
    - If we get stuck in an infinite loop the reset button may be the only solution!
    - If we don't let the OS run, many services may freeze (moving the mouse, writing to file, etc.)
  - The OS may increase the priority of a thread if a greater priority thread is waiting for its result.
- **If a „guaranteed" response time is needed** (e.g. process control, simulation, voice transfer (Skype), multimedia, etc.), it's worth separating the followings:
  - Process control: great /real-time priority thread/process
  - GUI : smaller priority thread/process
    - Refreshing the GUI won't consume so much CPU time
    - We cannot do this if the GUI interaction is very important as well (e.g. pressing a button should immediately have its effect)

„Real-time”

- Real-time operating systems
  - > **Hard real-time**
    - – Guaranteed response/execution time. Does not mean that it is fast, it may be a large value!
  - > **Soft real-time**
    - – The desired response time can be achieved with a high probability.
- Remark
  - > Environments that use garbage collection are less deterministic.
    - – Simulations should be implemented in C++
    - – Multimedia applications tend to get stuck if the GC is executing (this is not typical for .NET but is typical for the .NET Compact Framework)

Exception handling

- After .NET 2.0: if there is an unhandled exception on any of the threads → the application will exit
- The exception has to be handled inside of the thread. The starting thread won't get the exception!

```
public static void Main()
{
    // No point in wrapping this in a try-catch block
   new Thread(ThreadFunc).Start();
}

static void ThreadFunc()
{
    try { throw new Exception(“Error"); }
    catch (Exception ex)
    {
        // Log the exception, notify the user, etc.
    }
}
```

Exception handling in Windows Forms applications

- The exceptions of the event handlers can be handled in a general way
  - Application.ThreadException event
  - Handles the exceptions of (just) the event handlers
  - Forwarding the exceptions towards the ThreadException can be turned off

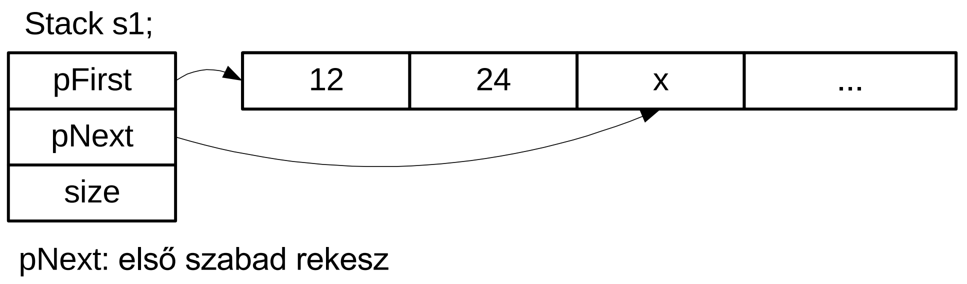
```
static class Program
{
    static void Main()
    {
        Application.ThreadException += HandleError;
        Application.Run(new Form1());
    }

    static void HandleError(object sender, ThreadExceptionEventArgs e)
    {
        // Log exception, then either exit the app or continue...
    }
}
```

Synchronising threads

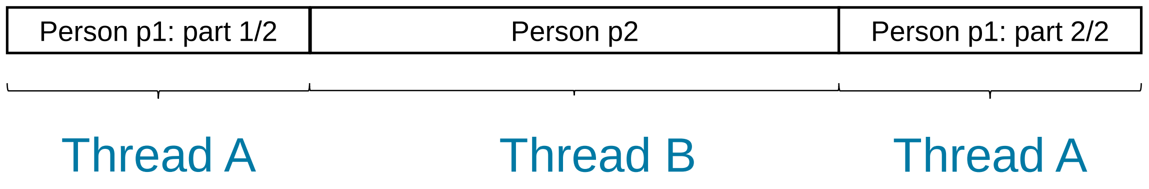
What to look out for when threads communicate?

The problem of mutual exclusion

- Environment: shared resource used from multiple threads
- Shared resource: memory (variable, object), file, etc.
- Consistency must be preserved. Even in a one-CPU case we don't know when the OS will take away the CPU from a thread!
  - Example: using a Stack class from multiple threads



  - Example: saving a Person object into file/memory from multiple threads
    - We may happen to write out just the half of the object when the CPU schedules an other thread for execution:



The problem of mutual exclusion

- How to preserve consistency? By mutual exclusion:
  ensure that the shared resource can just be accessed from one thread at the same time.
- Try to do this: introduce a ***bool flag*** variable, that is to indicate the reserved state of the resource (if it is reserved we have to wait).

```
while(flag); // Wait, if it is reserved
flag = 1; // Reserve it
   // We use the resource here
flag = 0; // If we don't need it
```

Thread A

```
while(flag);
flag = 1;
    // We use the resource here
flag = 0;
```

Thread B

- If the CPU is taken away from the thread right between the while(flag); and the flag = 1; operations→ both threads will see the resource free

Mutual exclusion

- The solution: we need an atomic test_and_set operation.
  - > The hardware architecture should support it
  - > Based on it the problem of mutual exclusion can be solved, a ***critical section*** can be defined
- **Critical section**
  - > A region of the code where we use the shared resource. This part is considered to be atomic.
  - > Both the OS and .NET supports it
- There was an other problem as well
  - > The ***while(flag);*** uses CPU intensively (either 100%)

.NET synchronization solutions I.

- Mutual exclusion is just one of the synchronization problems
- **Locking solutions**:

Name	Goal	Between processes?	Speed
lock C# operator (Monitor.Enter/Monitor. Leave)	A shared resource can just be accessed from one thread at the same time.	No	Fast
Mutex	Similar to the lock, but also works between processes. E.g.Single-instance application	Yes	Average
Semaphore	Similar to the Mutex, but it allows N parallel access.	Yes	Average
ReaderWriterLock	Efficient if there are many readers. Any number of readers can have access to the shared resource at the same time, but there can just be one writer (the writer excludes other readers as well). E.g. a cache that is rarely modified.	No	Average

Using the *lock*

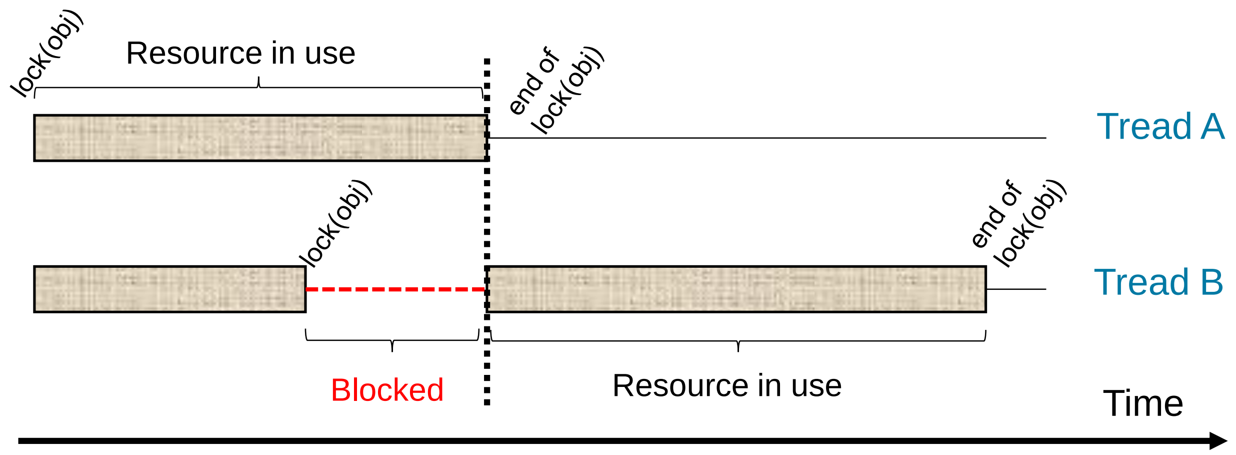
- Goal: print ‘Done’ once using ‘racing’ threads.

```
class ThreadSafeClass
{
    static bool done; // shared resource
    static object syncObject = new object();
    static void Main()
    {
        new Thread(Run).Start();
        new Thread(Run).Start();
    }

    static void Run()
    {
        lock (syncObject)
        {
            if (!done) { done = true; Console.WriteLine("Done"); }
        }
    }
}
```

Using the *lock*

```
while (flag) ;
flag = 1;
//Using the resource
flag = 0;
```

- How lock works
  - It takes any Object as a parameter
  - Checks if there is a lock on the object or not
    - If there isn't, it puts a lock on it and lets the thread continue the execution (technically this is an **atomic test_and_set**)
    - If there is, it blocks the calling thread until the object is unlocked.
  - Waiting for a lock does not consume CPU
  - Waiting threads will get into a FIFO queue. They will be let into the critical section in the order of their arrivals.
  - When leaving the lock region, the lock on the object is released



The lock and the Monitor class

- From the lock the compiler will generate the following

```
lock (syncObject)
{
    // ...
}
```

```
try
{
    Monitor.Enter(syncObject);
    // ...
}
finally
{
    Monitor.Exit(syncObject);
}
```

- Monitor.TryEnter() – similar, but a timeout can be set

Nesting locks

- The repeated locking of the same object in the same thread won’t cause blocking.
- To effectively release the lock, it must be released as many times as it was locked.

```
static object syncObject = new object();

static void Main()
{
    lock (syncObject)
    {
        Console.WriteLine(„locked");
        Nest();
        Console.WriteLine(„still locked");
    } // Release lock
}

static void Nest()
{
    lock (syncObject)
    {
        // ...
    } // Still locked
}
```

What can be used as a synchronization object?

- What can be the parameters of the lock?
  - Just reference types (classes)!
- How to protect our variables?
  - Use non-static parameters to protect non-static fields (object level locking)
  - Use static parameters to protect static fields (class level locking)

Object level:

```
class ThreadSafeClass
{
    long var = 0;
    object syncObject = new object();

    void Increment()
    {
        lock (syncObject) { var++; }
    }
}
```

Class level:

```
class ThreadSafeClass
{
    static long staticVar = 0;
    static object classSyncObject
        = new object();

    static void IncrementStatic()
    {
        lock (classSyncObject)
        { staticVar++; }
    }
}
```

Thread-safe classes

- If a class is thread-safe it can be used from multiple threads without any further synchronization
- The consistency is guaranteed by the class itself
  - There is no need for locking when we use it

Thread-safe Stack class ➔
(empty / full check is missing)

```
public class Stack<T>
{
    readonly int size;
    int current = 0;
    T[] items;
    object syncObject = new object();

    public void Push(T item)
    {
        lock (syncObject)
        {
            items[current++] = item;
        }
    }

    public T Pop()
    {
        lock (syncObject)
        {
            return items[--current];
        }
    }
}
```

Thread-safe classes

- Should we make all our classes thread-safe?
  - Threading is quite time-consuming
  - If it is likely to be used in a multi-threaded environment, make the class thread-safe, otherwise don’t.
    - Later we can still put some locks around its operations, or create a thread-safe wrapper class that has got the same operations as the original one.
- Classes in the .NET Framework
  - Usually just static variables are protected (synchronized), just the static operations/properties are thread-safe (check the documentation though)

Thread-safe operations

- What do we have to protect?
  - In C# only reading and writing 32 bit long (or shorter) variables is atomic

```
static int x, y;
static long z;

static void Test()
{
    long myLocal;
    x = 3; // Atomic
    z = 3; // Not atomic on 32 bit OS (z is 64 bits)
    myLocal = z; // Not atomic
    y += x; // Not atomic: two operations (read + write)
    x++; // Not atomic: two operations (read + write)
}
```

- How can we protect them
  - Use Lock/Mutex
  - Use non-blocking atomic operations: InterlockedXXX, very efficient

Interlocked atomic operations

- What do you need to know? Show an example (e.g. the bold part)

```
class Program
{
    static long sum;

    static void Main()
    {
        Interlocked.Increment(ref sum);
        Interlocked.Decrement(ref sum);
        Interlocked.Add(ref sum, 3); // Adding a number
        Console.WriteLine(Interlocked.Read(ref sum)); //64 bit read
        // Writing a value and reading the previous one in one atomic op.
        // This will print out 3, and the sum will be 10
        Console.WriteLine(Interlocked.Exchange(ref sum, 10));
        // If sum is 10, then print 123 in an atomic operation
        Interlocked.CompareExchange(ref sum, 123, 10);
    }
}
```

Volatile fields

```
class Foo
{
    public int x;
    public int y;
}
```

Not always 5!

Thread 1

```
foo.y = 5;
foo.x = 1;
```

Thread 2

```
if (foo.x == 1)
{
    Console.WriteLine(foo.y);
}
```

- The value of y can be cached in a register that the other thread cannot see.
- The compiler is allowed to modify the order of reading and writing non-volatile fields if the change does not effect the result concerning one thread.

Volatile fields

- Solution

```
class Foo
{
    public volatile int x;
    public volatile int y;
}
```

  - A volatile write cannot be delayed.
  - A volatile read can not be done earlier.
  - Writing into the memory is done „immediately” (the value is not cached into a register)
- Remark: in case of a lock there is no need to use volatile fields
  - When we enter a lock block all the variables will be refreshed from the memory
  - When we exit a lock block all the variables will be saved into the memory

Stopping a thread – Solution 1

- Use a bool value, and check it regularly in a while!
- Wait for the exiting thread : Thread.Join()

```
class ThreadClass
{
  static volatile bool exit = false;

  static void Main()
  {
    Thread thread = new Thread(Run);
    thread.Start();
    Thread.Sleep(2000);
    Console.WriteLine(
      "Signaling to worker thread.");
    exit = true;
    Console.WriteLine("Waiting for
      worker thread to exit.");
    thread.Join();
    Console.WriteLine("Worker thread
      definitely has exited.");
    }
}
```

```
static void Run()
{
  int counter = 0;
  while (!exit)
  {
    Thread.Sleep(300);
    Console.WriteLine(counter++.ToString());
  }

  Console.WriteLine(
    "Exiting from worker thread.");
  }
}
```

.NET synchronization - solutions II.

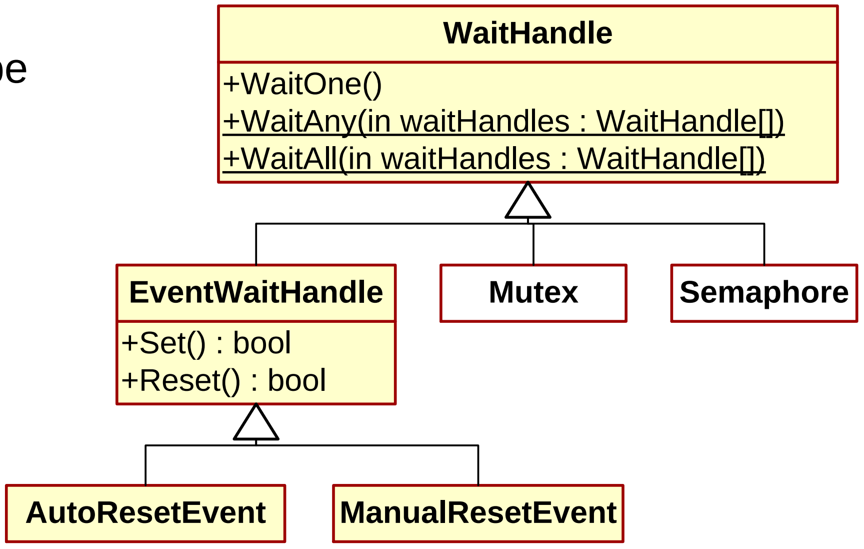
- **lock** solves the problem of mutual exclusion
- Cannot be used to make a signal for an other thread (e.g.: data is ready, exit, etc.)

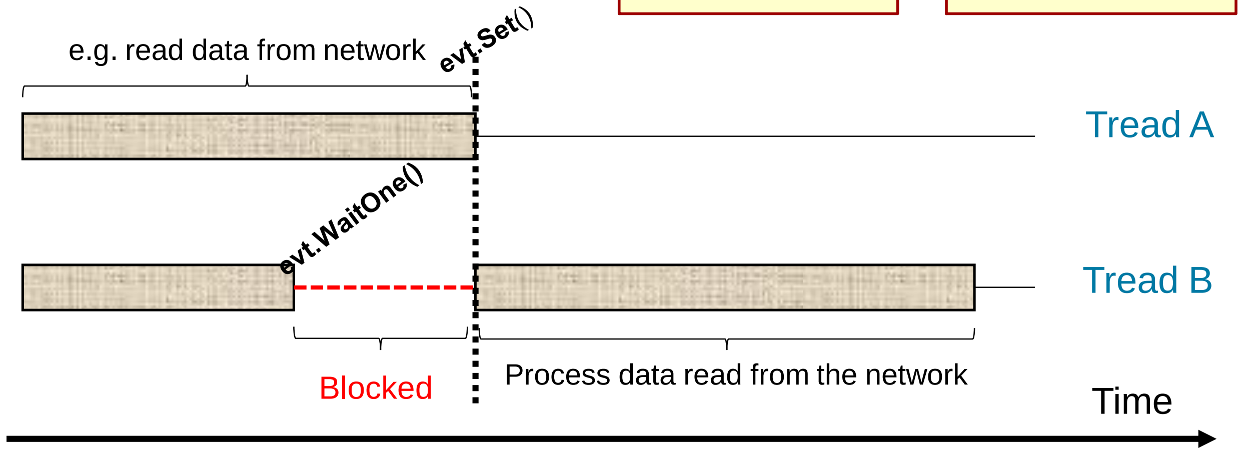
Synchronization objects that can be used to make signals

Name	Goal	Between processes?	Speed
EventWaitHandle	A thread can wait for a signal sent by an other thread in an efficient way.	Yes	Average
Monitor.Wait and Monitor.Pulse	A thread can wait for any blocking condition in an efficient way.	No	Average

WaitHandle hierarchy

- WaitHandle
  - The abstract base of classes that can be used to wait for an event
- AutoResetEvent, ManualResetEvent
  - the actual implementations we will use




e.g. read data from network
evt.Set()
Tread A
evt.WaitOne()
Tread B
Blocked
Process data read from the network
Time

AutoResetEvent

- **Just like the ski-lift…**
- **Set** operation:
  - Sets the event object signaled. This will unblock exactly one waiting thread. If there are no waiting threads it will remain signaled. Does not count the number of Set calls (more than one call is the same as one).
- **WaitOne** operation:
  - If the event object is not signaled the calling thread will be blocked (does not consume CPU)
  - If the event object is signaled:
    - The calling thread in not blocked and can continue the execution
    - In case of AutoResetEvent, the event will automatically be set back to its non-signaled state (in case of ManualResetEvent events we have to do it manually)
- **Reset** operation:
  - Resets: sets the event object non-signaled

AutoResetEvent example

- Note: WaitOne takes an optional timeout parameter

```
class SimpleEventDemo
{
    // Set a default value through the constructor
    static EventWaitHandle wh = new AutoResetEvent(false);

    static void Main()
    {
        new Thread(Run).Start();
        Thread.Sleep(1000); // Sleep a little
        wh.Set(); // Wake up
    }

    static void Run()
    {
        Console.WriteLine(“Waiting to be notified ...");
        wh.WaitOne(); // Waiting
        Console.WriteLine(“The notification has arrived.");
    }
}
```

AutoResetEvent – examples

- AutoResetEvent
  - See ThreadDemo solution...
- Producer/consumer
  - It is needed frequently in real life
    - A background thread continuously reads data from the serial port/network, converts it to messages and puts these into a queue for later processing.
    - Load balancing (if there are bursts in the arrival of requests, we need not delay messages when we can not process them immediately)
    - Scalability: there can be any number of consumer threads executing in parallel.
  - See ThreadDemo solution...

Classes that derive from WaitHandle

- **ManualResetEvent**
  - Similar to AutoResetEvent but won't set the event back to non-signaled automatically
  - Just like a „gate”: if it is signaled every thread can continue the execution, otherwise every thread is blocked.
- **Mutex**
  - Just like the lock, mutual exclusion
  - Can be uses between threads of different processes. This is much slower (100 times) than the lock.
  - Can be reserved by calling WaitOne() (reserves and continues or blocks in an atomic way), and can be released by calling ReleaseMutex().
- **Semaphore**
  - Just like the mutex, but more than one (N, initial value for the counter) access is allowed.
  - WaitOne() is used to wait for it. If the semaphore is free it won't block but will decrease the counter by one. The semaphore is free if its counter is greater than 0. One can release the semaphore by calling ReleaseSemaphore(), that will increase the counter by one.

WaitHandle

- Classes that derive from the WaitHandle implement the IDisposable interface (as they are wrapper classes around native handlers)

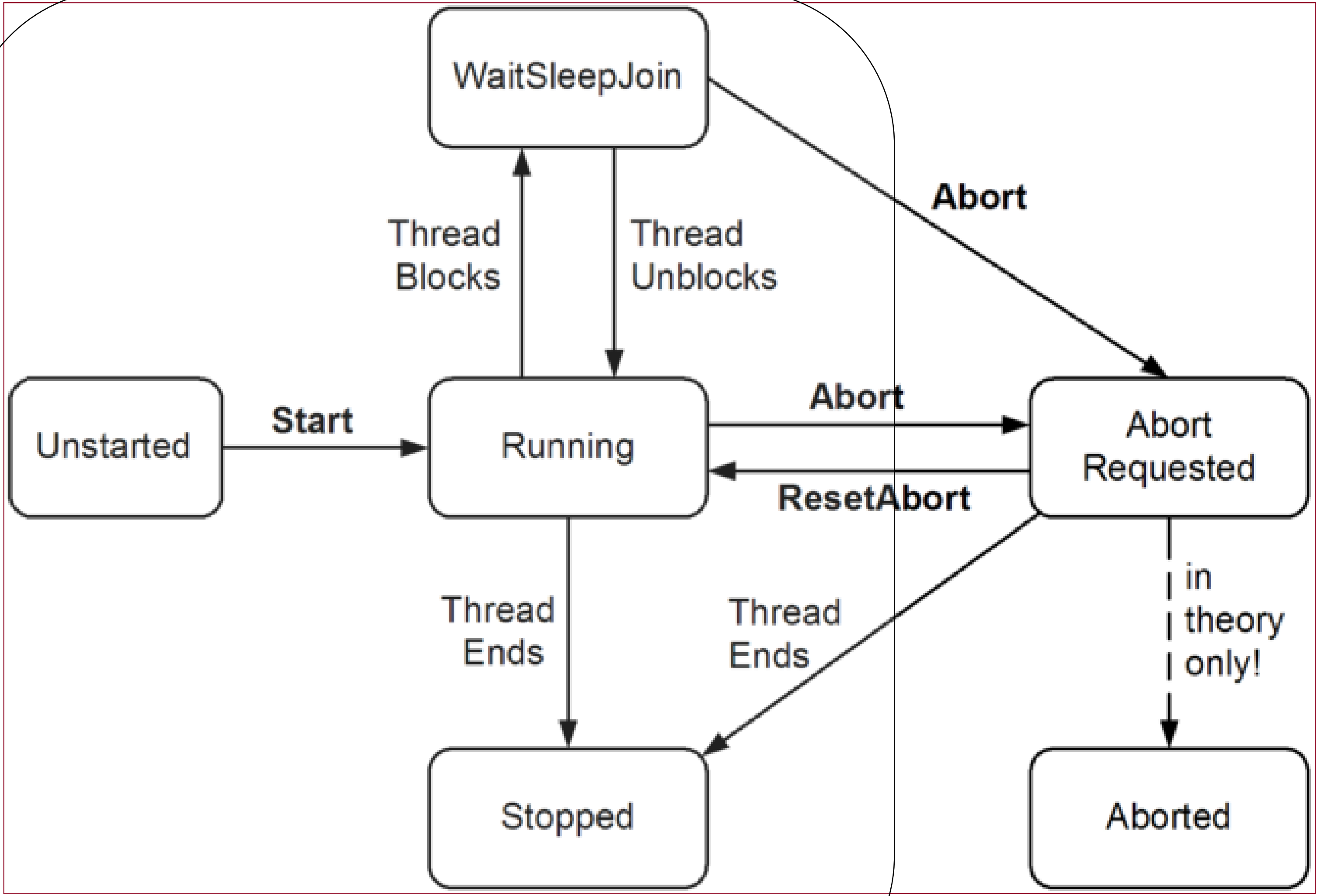
- **WaitHandle.WaitAny** static operation
  - The parameter is an array of WaitHandles.

    If any of them are signaled it won't block.

Demo
(Producer/Consumer queue)

- **WaitHandle.WaitAll** static operation
  - The parameter is an array of WaitHandles. If all of them are signaled it won't block.

States of thread



States of threads

- **WaitSleepJoin** state (important!!!)
  - Blocked state
  - After Sleep, Join, lock, WaitOne, WaitAny, WaitAll operations.
  - All operations may block. Until when? Ways to leave the blocking state:
    - The condition we were waiting for becomes true
    - The timeout has elapsed
    - The blocking is interrupted by a Thread.Interrupt call
    - The blocking is aborted by a Thread.Abort call

Stopping a thread – solution 2

- Until now we have been using a bool flag member variable. But what happens when our class is calling an other class where the thread is blocked and we cannot check the bool value?

Demo

- The solution is the **Thread.Interrupt()** call
  - > The thread will get a **ThreadInterruptedException**, but only if it was in the **WaitSleepJoin** state
    - – That is it won't exit from a while(true); loop

Thread.Interrupt()

```
Thread thread = new Thread(Run);
thread.Start();
Thread.Sleep(2000);
thread.Interrupt(); // Signaling to worker thread.
thread.Join(); // Waiting for worker thread to exit.
Console.WriteLine("Worker thread definitely has exited.");
// ...
static void Run()
{
    try
    {
        string task;
        while (true)
        {
            task = queue.GetTask();
        }
    }
    catch (ThreadInterruptedException)
    {
    }
    Console.WriteLine("Exiting from worker thread.");
}
```

```
class OneItemQueue
{
    public string GetTask()
    {
        // WaitSleepJoin állapotban
        // blokkolt.
        autoResetEvent.WaitOne();
        return task;
    }
    ...
```

Stopping a thread – solution 3

- **Thread.Abort()** operation
- Similar to Thread.Interrupt(), but
  - The thread will get a **ThreadAbortException**
  - The thread will get it in all states (not only in the WaitSleepJoin state)
    - That is it will exit from the while(true); loop
    - The finally blocks will be executed
- Don't use it as you don't know what the thread was actually doing! You can get stuck in an inconsistent state.

```
StreamWriter w = File.CreateText("myfile.txt");
// If it aborts here, bad things may happen! ☺
try
{
    w.Write("Abort-Safe");
}
finally { w.Dispose(); }
```

- Only in one case
  - If we exit from the application as well.
  - The unreleased resources will be released if we exit the application

Multi-threaded Windows Forms applications

Architecture

- Windows Forms controls (classes that derive from the Control class) can only be accessed from the thread where they were created.
  - GUI elements are usually created in the main thread
  - Don’t access the GUI (call method/property) from a worker thread.
  - Only **Control.Invoke** can be called from a different thread
  - **Control.Invoke** executes the parameter method in the thread that has created the control as well (usually this is the main thread).
- **BackgroundWorker** class
  - Hides the Control.Invoke calls
  - Easy to use

Thread pool

Thread pool

- E.g.: we are creating a server application. The goal is to serve multiple client requests in parallel.
- Solution 1 – A new thread is created for every request. Problems
  - > There will be too many threads if there are many parallel requests
    - – It is not worth using more than about 100 threads per process (overhead of changing between threads)
  - > Creating a new thread is costly
- The better solution is to use thread pools
  - > Pre-created threads
  - > We just ask for one when we want to execute a task in a new thread
  - > After the task has finished, we give the thread back to the thread pool so that it can be used to execute another task.
  - > If there are no threads available in the thread pool:
    - – We create a new thread and put it into the pool
    - – If there are too many threads in the pool – block the caller until a thread becomes free

ThreadPool

- ThreadPool is a .NET class
- It will be created for every process
- When should we use the ThreadPool?
  - Only for short operations. Don’t block the tread pool threads for a long time (or run the risk of using them up quickly).
- The Thread.Interrupt cannot really be used
  - Use a flag
  - Use ManualResetEvent

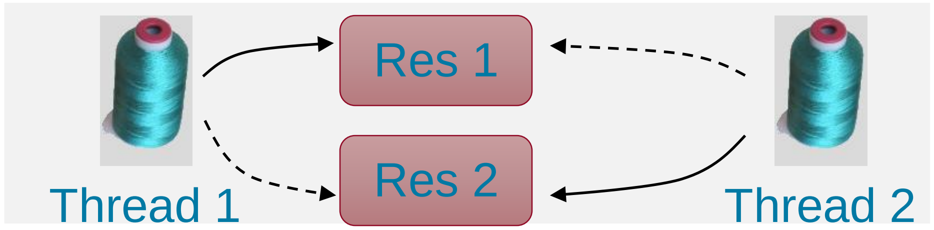
Using the thread pool

```
class Test
{
    static ManualResetEvent doneEvent = new ManualResetEvent(false);

    public static void Main()
    {
        ThreadPool.QueueUserWorkItem(Run); // delegate as parameter
        Console.WriteLine("Waiting for threads to complete...");
        doneEvent.WaitOne();
        Console.WriteLine("Worked has ended!");
        Console.ReadLine();
    }

    public static void Run(object instance)
    {
        Console.WriteLine("Started...");
        Thread.Sleep(1000);
        Console.WriteLine("Ended");
        doneEvent.Set();
    }
}
```

Deadlock

Deadlock Demo

- **Deadlock**: Two (or more) threads are waiting for each other. The cause of the problem: waiting while locking resources!



- **Can be detected**: We have to search in a directed graph. We have to implement it ourselves. Usually we don't do this. Instead we try to avoid it.
- **Avoiding it**:
  - The resources are locked in the same order in all the threads
  - It cannot always be done ☹
- **Alternative solution:** Try to get the lock just for a while (use timeouts).

OUTRO

Drawbacks of multi-threaded applications

- Makes applications more complex
  - > The synchronization between threads needs to be solved. E.g. mutual exclusion
- Very-very-very-very difficult to figure out the problems resulting from incorrect synchronization
  - – E.g. if we forget to add a lock, the application might crash/halt only once every month
  - – Hard to debug, hard to reproduce
- Too many threads may overload the system
  - > Needs scheduling, *context switch* – CPU intensive
  - > Every thread has got its own stack – needs memory
- Deadlocks
- Lesson: concurrent programming can't be semi-learned.

Some things we missed

- How to use a Mutex, Semaphore
  - > In a similar way to AutoResetEvent
- Named Mutex, Event, Semaphore
  - > Only if we need to synchronize processes
- Using Monitor.Wait, Monitor.Pulse and Monitor.PulseAll
- A more detailed look at BackgroundWorker
- Thread Local Storage
  - > Variables tied to a thread
- Higher level thread handling
  - > Task, Task<T>
- Language level support for asynchronous operations
  - > async, await keywords, asynchronous methods
- Etc.

References

- **Threading in C#**
  - http://www.albahari.com/threading/
- .NET Framework Developer's Guide - Managed Threading
  - http://msdn2.microsoft.com/en-us/library/3e8s7xdd.aspx