Advanced Programming C#

Lecture 12

dr inż. Małgorzata Janik

malgorzata.janik@pw.edu.pl


WYDZIAŁ FIZYKI
POLITECHNIKA WARSZAWSKA
POLITECHNIKA WARSZAWSKA
WARSAW UNIVERSITY OF TECHNOLOGY

Project

Project part III

- Final Date: 25.01.2020 (next week!)
- Presentations (max 5 min per project!):
  - 1 poster/slide that advertises your project
  - presentation of the program
    - real-time presentation of the application
  - to be shown to the whole group
- Code will be evaluated
  - Final code should be added to Moodle
  - Project **must** be shared via git repository as well
  - Each person from the team can get separate mark
  - If application does not meet basic requirements from the specification, the project is not graded

Multithreading

Threading in C#

- C# supports parallel execution of code through multithreading.
- A thread is an independent execution path, able to run simultaneously with other threads.
- A C# client program (Console, WPF, or Windows Forms) starts in a single thread created automatically by the CLR and operating system (the “main” thread), and is made multithreaded by creating additional threads.

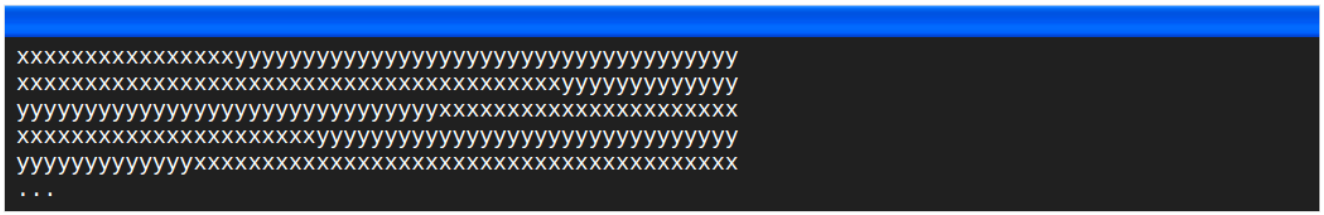
http://www.albahari.com/threading/

Threading: basics

```
class ThreadTest
{
 static void Main()
 {
  Thread t = new Thread (WriteY);     // Kick off a new thread
  t.Start();                          // running WriteY()

  // Simultaneously, do something on the main thread.
  for (int i = 0; i < 1000; i++) Console.Write ("x");
 }

 static void WriteY()
 {
  for (int i = 0; i < 1000; i++) Console.Write ("y");
 }
}
```



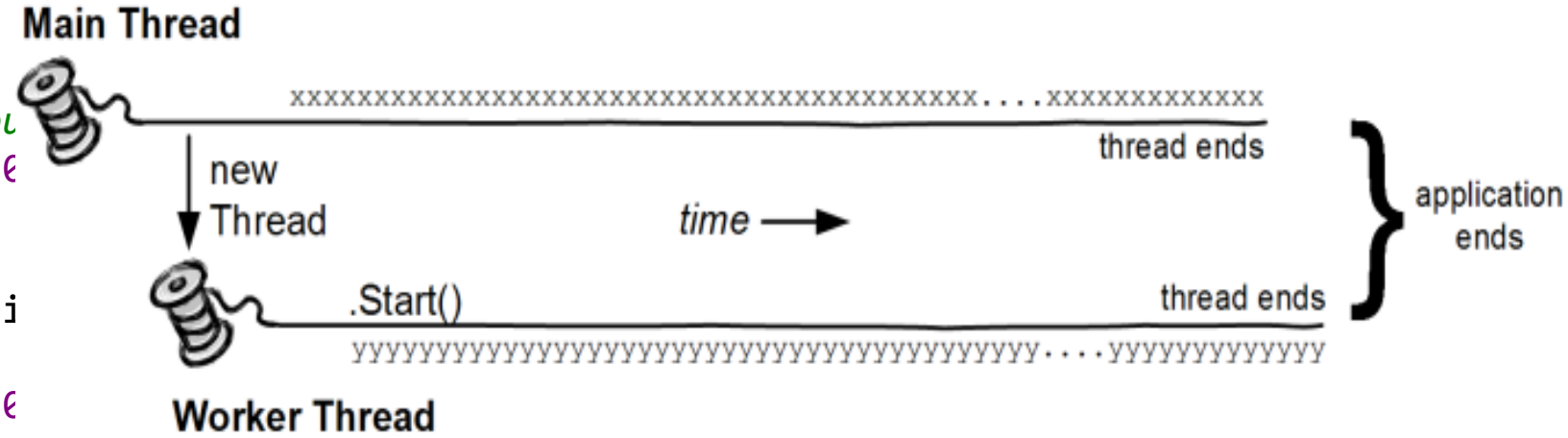
http://www.albahari.com/threading/

Threading: basics

```
class ThreadTest
{
 static void Main()
 {
  Thread t = new Thread (WriteY);     // Kick off a new thread
  t.Start();

  // Simultaneou
  for (int i = 0
 }

 static void Wri
 {
  for (int i = 0
 }
}
```




```
xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

http://www.albahari.com/threading/

How does it work?

- Multithreading is managed internally by a thread scheduler, a function the CLR typically delegates to the operating system.
- A thread scheduler ensures all active threads are allocated appropriate execution time, and that threads that are waiting or blocked do not consume CPU time.
  - On a single-processor computer, a thread scheduler performs **time-slicing** — rapidly switching execution between each of the active threads.
  - On a multi-processor computer, multithreading is implemented with a mixture of time-slicing and genuine concurrency, where different threads run code simultaneously on different CPUs.

http://www.albahari.com/threading/

Threads vs Processes

- A thread is analogous to the operating system process in which your application runs.
  - Just as processes run in parallel on a computer, threads run in parallel within a single process.
- Processes are fully isolated from each other; threads have just a limited degree of isolation.

http://www.albahari.com/threading/

Creating and starting threads

- Threads are **created using the Thread class's constructor**, passing in a ThreadStart delegate which indicates where execution should begin.
- Calling **Start** on the thread then sets it running. The thread continues until its method returns, at which point the thread ends.
- Creating threads:

```
static void Main() {
 Thread t = new Thread (new ThreadStart (Go));
 Thread t2 = new Thread (Go); //The same = just shorter syntax
 t.Start();   // Run Go() on the new thread.
 t2.Start();
}

static void Go()
  {
    Console.WriteLine ("hello!");
  }
```

http://www.albahari.com/threading/

Task 1: Crating threads

- Create new Console Application.
- Write simple function
  - public static void Increment()

    which prints 10 numbers: 0 1 2 3 4 5 6 7 8 9 in a for loop
- In Main function:
  - Create and start new threads t1 and t2 using both methods (as in the example from previous slide) that run Increment function.
  - Run Increment() from the main thread.
  - In the end of the program print „Main thread at the end” and wait for the key from the user

Naming threads

- Each thread has a Name property that you can set for the benefit of debugging. This is particularly useful in Visual Studio, since the thread's name is displayed in the Threads Window and Debug Location toolbar. You can set a thread's name just once; attempts to change it later will throw an exception.
- The static Thread.CurrentThread property gives you the currently executing thread. In the following example, we set the main thread's name:

```
class ThreadNaming
{
 static void Main()
 {
  Thread.CurrentThread.Name = "main";
  Thread worker = new Thread (Go);
  worker.Name = "worker";
  worker.Start();
  Go();
 }
 static void Go()
 {
  Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);
 }
}
```

http://www.albahari.com/threading/

Task 2: Naming

- Name all the threads (including the main thread) so that they are easily identifiable.
- In Increment method add information which thread prints the number
  - Do the same for the next task (Decrement)

Passing data to a thread

- The easiest way to pass arguments to a thread's target method is to execute a lambda expression that calls the method with the desired arguments:

```
Thread t = new Thread ( () => Print ("Hello from t!") );
t.Start();
```

- You can do the same thing almost as easily in C# 2.0 with anonymous methods:

```
Thread t = new Thread (delegate()
{
    Print ("Hello from t!");
});

 t.Start();
```

http://www.albahari.com/threading/

Task 3: Threads with parameters

- Write simple Decrement function
  - public static void Decrement(int **max**)

    which prints numbers from **max** to 0 in a for loop
- In Main function
  - Create and start threads t3 and t4 which print numbers from 5 to 0 (t3) and 10 to 0 (t4). Threads should be created using both mentioned methods.

Join and Sleep

- You can wait for another thread to end by calling its **Join** method. For example:

```
static void Main()
{
 Thread t = new Thread (Go);
 t.Start();
 t.Join();
 Console.WriteLine ("Thread t has ended!");
}

static void Go()
{
 for (int i = 0; i < 1000; i++) Console.Write ("y");
}
```

This prints “y” 1,000 times, followed by “Thread t has ended!” immediately afterward. You can include a timeout when calling Join, either in milliseconds or as a TimeSpan. It then returns true if the thread ended or false if it timed out.

- **Thread.Sleep** pauses the current thread for a specified period:

```
Thread.Sleep (TimeSpan.FromHours (1));  // sleep for 1 hour
Thread.Sleep (500);                     // sleep for 500 milliseconds
```

- While waiting on a Sleep or Join, a thread is blocked and so does not consume CPU resources.

Task 4: Join & Sleep

- In the Increment method sleep for 10 ms after printing each number.
- Using Join method order main thread to wait for all the others (t1,t2,t3,t4) at the end of the program to finish.

Abort

- By default threads die after they fulfilled their function (when they encounter the end of function)
- All blocking methods (such as Sleep, Join, EndInvoke, and Wait) block forever if the unblocking condition is never met and no timeout is specified. Occasionally, it can be useful to release a blocked thread prematurely; for instance, when ending an application. Two methods accomplish this:
  - Thread.Interrupt (almost never needed)
  - **Thread.Abort**
- A blocked thread can be forcibly released via its Abort method. **ThreadAbortException** is thrown. Thread should catch it in the try catch block.
- Program does not *kill* threads; it rather nicely asks (using an exception) for a suicide.

Exceptions and threads

- Catching exceptions in threads should always be handled inside the function the thread is starting (certainly not outside)

```
public static void Main()
{
   new Thread (Go).Start();
}

static void Go()
{
 try
 {
  //normal thread execution
 }
 catch (Exception ex)
 {
  // Typically log the exception, and/or signal another thread
  // that we've come unstuck
  // ...
 }
}
```

http://www.albahari.com/threading/

Task 5: Abort

- In the Increment and Decrement function add exception handling.
  - In case of finding signal from Abort method thread should print text „Thread [name] Aborted!” and finish.
  - In case of other exceptions, program should print the name of the exception.
- Abort t1 and t4 Threads in the Main function.

Locks

- Exclusive locking is used to ensure that only one thread can enter particular sections of code at a time.
- Other threads wait untill the lock becomes available.

```
class ThreadSafe
{
 static bool done;
 static readonly object locker = new object();

 static void Main()
 {
  new Thread (Go).Start();
  Go();
 }

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

http://www.albahari.com/threading/

Task 6: Locks

- Add to the program global variable
  - **static int GLOBAL_VALUE = 0;**
- Increase GLOBAL_VALUE by 1 in Increment function and decrease by 1 in Decrement function (see next point)
- Change Increment and Decrement functions to increase a chance of problems:
  - Create **int tmp = GLOBAL_VALUE;** variable in the beginning; increment/decrement it
  - Rewrite **GLOBAL_VALUE = tmp;** in the end of the for loop
- Print the GLOBAL_VALUE value at the end of the program. Observe variations from number 5 when running program.
- Add locks to deal with the created problem.

References

- **Only one, but highly recommended:**
  - **http://www.albahari.com/threading**

THE END

dr inż. Małgorzata Janik
malgorzata.janik@pw.edu.pl