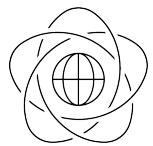
Programming 3 Advanced

Threads, Tasks

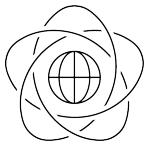
Tomasz Herman

Faculty of Mathematics and Information Science
Warsaw University of Technology

Lecture 11, 16 grudnia 2024



Outline



Threads

A thread is the smallest unit of execution within a process

Lightweight Less overhead in creating and managing them compared to processes

Shares Resources Heap memory, file handles

Independent execution Each thread has its own stack

Concurrency Can run in parallel

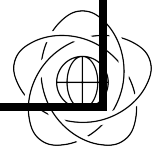
Synchronization Access to shared resources might need synchronization

Using threads

`https://learn.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading`

Creating a thread

```
1  Thread thread = new Thread(PrintA);
2  thread.Start();
3
4  PrintB();
5
6  private static void PrintA()
7  {
8      for (int i = 0; i < 100; i++)
9          Console.Write('A');
10 }
11
12 private static void PrintB()
13 {
14     for (int i = 0; i < 100; i++)
15         Console.Write('B');
16 }
```

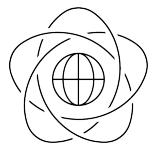


Sleeping

```
1 Thread.Sleep(TimeSpan.FromMilliseconds(10)
2 Thread.Sleep(millisecondsTimeout: 10);
```

Pausing and Interrupting threads

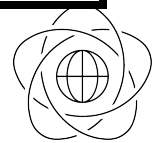
https://learn.microsoft.com/en-us/dotnet/standard/threading/pausing-and-resuming-threads



Thread safety

There is a slight chande Done is printed twice

```
1  Thread thread = new Thread(UnsafePrintOnce);
2  thread.Start();
3  UnsafePrintOnce();
4  thread.Join();
5
6  private static bool _done = false;
7  private static void UnsafePrintOnce()
8  {
9      if (_done) return;
10     _done = true;
11     Console.WriteLine("Done");
12 }
```



Thread safety

Thread-safe version

```
1  private static readonly object Locker = new();
2  private static bool _done = false;
3  private static void SafePrintOnce()
4  {
5      lock (Locker)
6      {
7          if (_done) return;
8          _done = true;
9      }
10     Console.WriteLine("Done Safely");
11 }
```

A lock statement

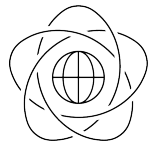
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock

Exception handling

```
1  new Thread(ThrowsException).Start();
2
3  private static void ThrowsException()
4  {
5      throw new Exception("Unhandled exception");
6  }
```

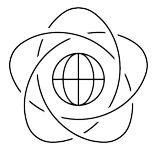
Unhandled exceptions

An unhandled exception in any thread causes the whole application to shut down.



Background threads

```
1  Thread worker = new Thread ( () => Console.ReadLine() )
2  {
3      Name = "BackgroundWorker",
4      IsBackground = true
5  };
6  worker.Start();
```

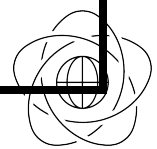


Synchronization contexts

Synchronization Context

https://stackoverflow.com/a/18098557

```
1  var _context = new MySynchronizationContext();
2  new Thread(() =>
3  {
4      Console.WriteLine(
5          $"{Thread.CurrentThread.Name} thread started");
6      _context?.Post(_ =>
7      {
8          Console.WriteLine(
9              $"Message from: {Thread.CurrentThread.Name}");
10     }, null);
11 }){Name = "Worker"}.Start();
```

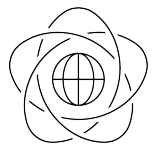


Thread Pool

```
1 Task.Run (() =>
2 {
3     Console.WriteLine(Thread.CurrentThread.Name);
4     Console.WriteLine("Hello from the thread pool");
5 });
```

Documentation

https://learn.microsoft.com/en-us/dotnet/standard/threading/the-managed-thread-pool



Creating a task

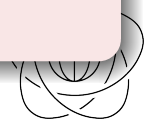
```
1  Task.Run (() =>
2  {
3      Console.WriteLine(Thread.CurrentThread.Name);
4      Console.WriteLine("Hello from the task");
5  });
```

Documentation

https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming

Thread pool

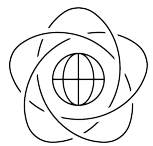
Tasks use pooled threads by default, which are background threads. This means that when the main thread ends, so do any task.



Tasks

Waiting for completion

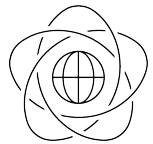
```
1  Task task = Task.Run(() =>
2  {
3      Thread.Sleep(2000);
4      Console.WriteLine(Thread.CurrentThread.Name);
5      Console.WriteLine("Hello from the thread pool");
6  });
7
8  Console.WriteLine($"Is completed: {task.IsCompleted}");
9  task.Wait(); // blocks until task finishes
10 Console.WriteLine($"Is completed: {task.IsCompleted}");
```



Tasks

Returning a value

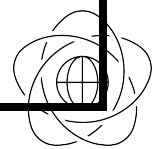
```
1  Task<int> t1 = Task.Run(() => CountPrimes(1000000, 1000000));
2  Task<int> t2 = Task.Run(() => CountPrimes(2000000, 1000000));
3  Console.WriteLine("Started two tasks counting primes");
4  // Blocks if task is not finished:
5  Console.WriteLine($"Primes in t1: {t1.Result}");
6  Console.WriteLine($"Primes in t2: {t2.Result}");
7
8  private static int CountPrimes(int from, int count)
9  {
10     // ...
11 }
```



Tasks

Handling exceptions

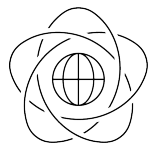
```
1  Task task = Task.Run(ThrowsException);
2
3  try
4  {
5      task.Wait();
6  }
7  catch (AggregateException e)
8  {
9      Console.WriteLine(e.InnerException);
10 }
11
12 private static void ThrowsException()
13 {
14     throw new Exception("Unhandled exception");
15 }
```



Tasks

Continuations

```
1  Task<int> t = Task.Run(() => CountPrimes(1000000, 1000000));
2  Task continuation = t.ContinueWith((Task<int> task) =>
3  {
4      Console.WriteLine($"Primes: {task.Result}");
5  });
6
7  continuation.Wait();
```



Tasks

Long running tasks

```
1  Task<int> t = Task.Factory.StartNew(() =>
2  {
3      Console.WriteLine($"{Thread.CurrentThread.Name}");
4      return CountPrimes(2, 1_000_000_000);
5  }, TaskCreationOptions.LongRunning);
6
7  t.Wait();
```