Lecture 3: Threads

Learning Goals

- Describe what a **thread** is in your own words
- Describe the role of the Operating System **kernel** in thread creation
- Create and use threads in C# using `System.Threading` Library
- Describe two methods of thread **communication** and give a code example
- Define a **race condition** and give an example
- Given code that runs in parallel, identify all **possible sequences** and outputs
- Given an application, identify where it might be useful to use separate threads
- List potential **drawbacks** or **limitations** of multiple threads

Single-Task vs Multi-Task

Single Tasking: a single program (thread) that runs sequentially

```
using System;

public static void Main() {

  for (int i=0; i<10; ++i) {
    Console.WriteLine("{0}",i);
  }

}
```

```
55
48 89 e5
48 83 ec 10
c7 45 fc 00 00 00 00
83 7d fc 09
7f 22
8b 45 fc
89 c6
bf 60 10 60 00
e8 66 fe ff ff
be 00 07 40 00
48 89 c7
e8 a9 fe ff ff
83 45 fc 01
eb d8
b8 00 00 00 00
c9
c3
```

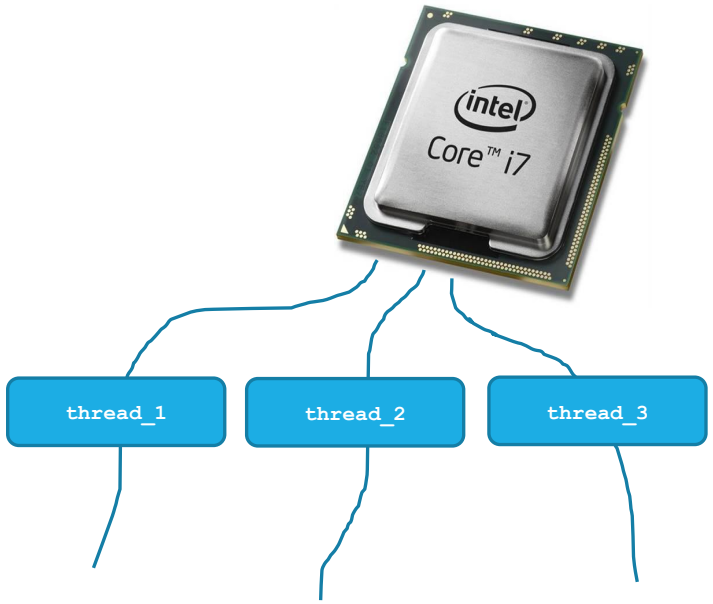
```
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
movl   $0x0,-0x4(%rbp)
cmpl   $0x9,-0x4(%rbp)
jg     40084d <main+0x37>
mov    -0x4(%rbp),%eax
mov    %eax,%esi
mov    $0x601060,%edi
callq
mov    $0x400700,%esi
mov    %rax,%rdi
callq  4006f0
addl   $0x1,-0x4(%rbp)
jmp    400825 <main+0xf>
mov    $0x0,%eax
leaveq
retq
```


```
55
48
89
e5
48
83
ec
10
fc 45 c7
```

Single-Task vs Multi-Task

Multi Tasking: sections of code can be executed in **parallel**

```
using System;
using System.Threading;

namespace concurrency
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 2; i++)
            {
                Thread thread = new Thread(MyProcess);
                // worker threads
                thread.Start();
            }
            Console.WriteLine("A process that takes 4s!");
            Thread.Sleep(4000);
            Console.WriteLine("Done!");
        }
        private static void MyProcess(object obj)
        {
            Console.WriteLine("A process that takes 4s!");
            Thread.Sleep(4000);
            Console.WriteLine("Done!");
        }
    }
}
```



What is an Operating System Kernel?

A **kernel** is the **heart** of an **operating system**, consists of a collection of applications, services and **software system calls** that implement

- Creating processes/threads
- Mutual exclusion,
- Process communication,
- Process synchronisation,
- Sharing memory,
- Scheduling and prioritization of processes,
- Handling the GUI (windows, mouse, keyboard etc.)
- Dealing with files & directories, networks etc.

A kernel provides a **set of services** and **system calls** to the **host programs** that we can access.

Creating a Thread

- `System.Threading:` namespace
- `Thread()`
  - Initializes a new instance of the Thread class
  - 4 overloads including object being passed to the thread (like methods) and the stack size of the thread
- `Join()` : wait for thread(s) to complete

Note1: If Join() is not explicitly used in the main thread, the main thread would not wait for the execution of the other threads.

Creating a Thread (Non-Static Method)

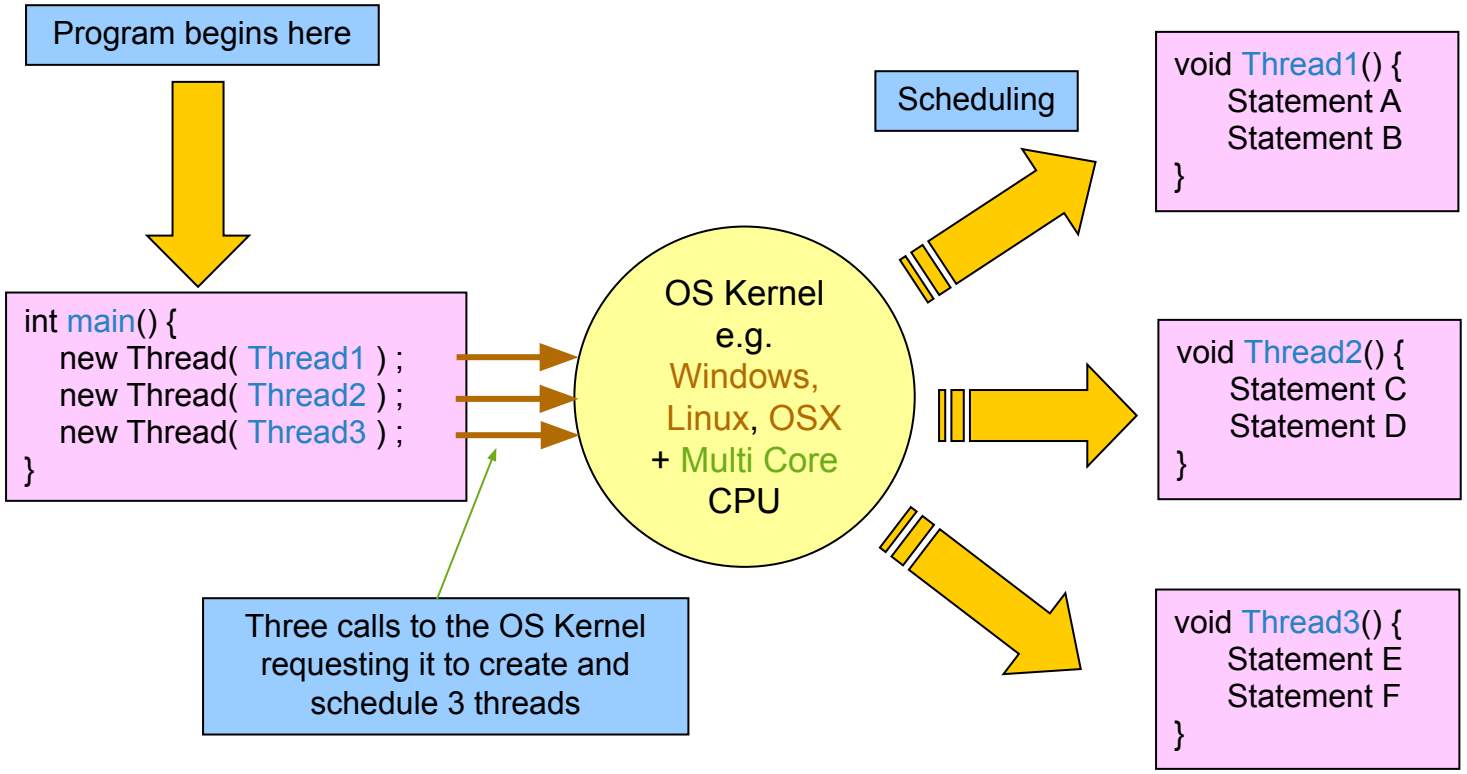
```
using System;
using System.Diagnostics;
using System.Threading;

public class NonStaticMethodMultiThreading
{
    public static void Main()
    {
        SomeClass obj = new SomeClass();
        var th = new Thread(obj.SomeWork);
        th.Start();
        Thread.Sleep(1000);

        th.Join();

    }
}
```

```
public class SomeClass
{
    public void SomeWork()
    {
        var sw = Stopwatch.StartNew();

        do
        {
            Console.WriteLine("Thread {0}: Elapsed {1:N2}(s)",
                              Thread.CurrentThread.ManagedThreadId,
                              sw.ElapsedMilliseconds / 1000.0);
            Thread.Sleep(500);
        } while (sw.ElapsedMilliseconds <= 5000);
        sw.Stop();
    }
}
```

Creating a Thread (Static Method)

```
using System;
using System.Diagnostics;
using System.Threading;

public class StaticMethodMultiThreading
{
    public static void Main()
    {
        var th = new Thread(SomeClass.SomeWork);
        th.Start();
        Thread.Sleep(1000);

        th.Join();

    }
}
```

```
public class SomeClass
{
    public static void SomeWork()
    {
        var sw = Stopwatch.StartNew();

        do
        {
            Console.WriteLine("Thread {0}: Elapsed {1:N2}(s)",
                              Thread.CurrentThread.ManagedThreadId,
                              sw.ElapsedMilliseconds / 1000.0);
            Thread.Sleep(500);
        } while (sw.ElapsedMilliseconds <= 5000);
        sw.Stop();
    }
}
```



The **main()** method calls the **OS kernel** to create 3 **threads**. The OS **schedules** them using **time slicing** if only 1 CPU or core is present, or designates them to run in **parallel** if more than 1 CPU or core is present, or some combination.

The **operating system** can then perform load balancing to distribute the work as fairly as possible.

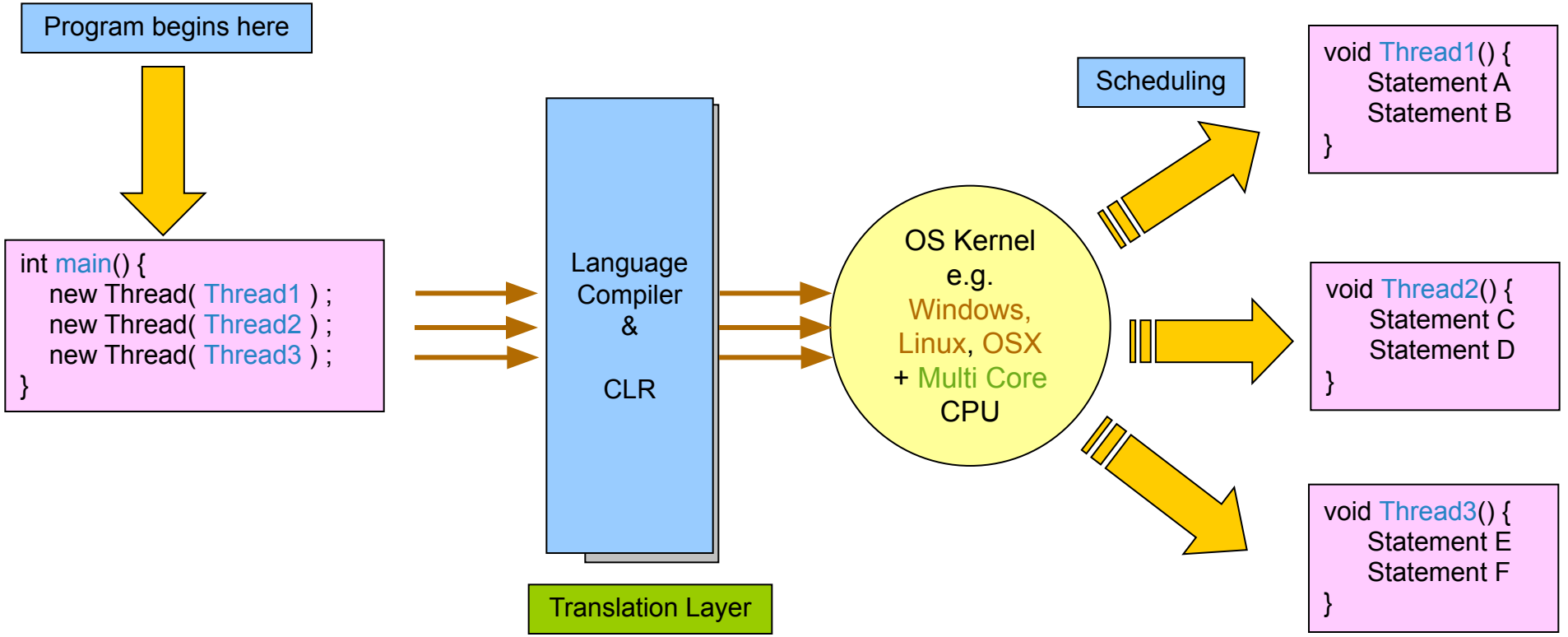
Library Abstractions

Calling OS kernel functions **explicitly** is undesirable. It **ties** our programs to a specific OS, making our code less **portable**.

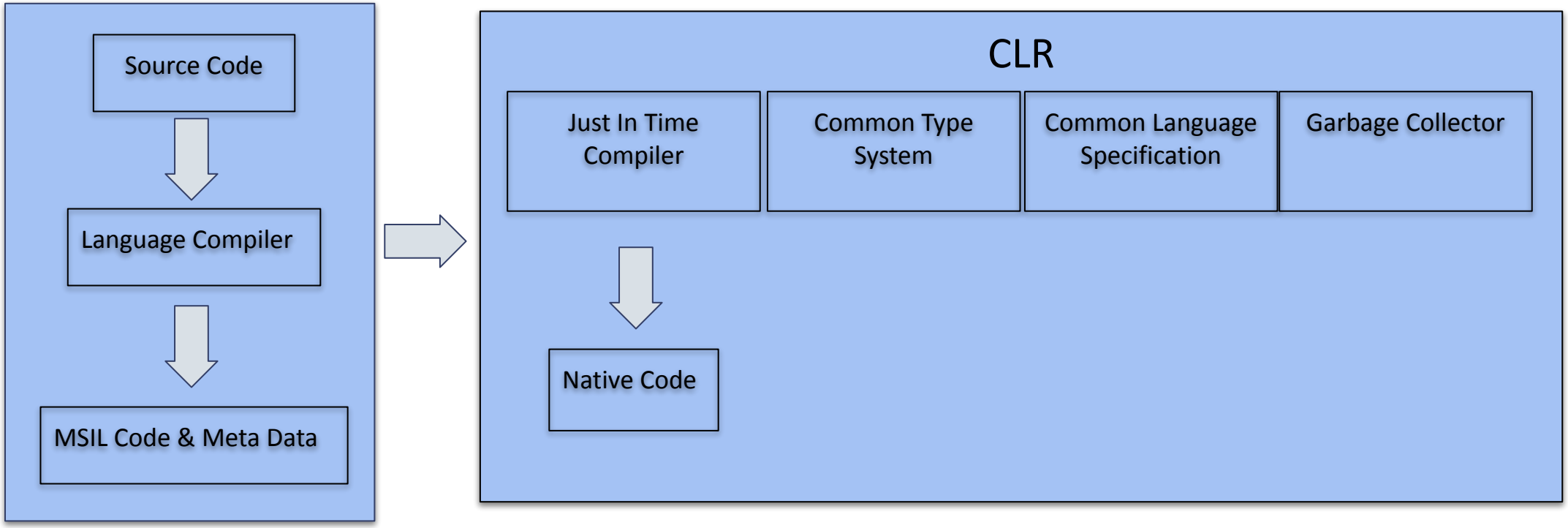
Some of the details can be **abstracted** away, wrapped in a **library** that provides a consistent **interface**.

Many multi-threading capabilities, including inter-thread communication, are part of the System.Threading namespace

Library Abstractions



What is the CLR Role in C#?



Common Language Runtime

- **Common Language Runtime**: “The common language runtime makes it easy to design components and applications whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated”

CLR Explained

- **Just In – Time Compiler (JIT):** Converts the MSIL (or CIL or IL) to native (Machine Code)
- **Garbage Collection (GC):** provides automatic energy management
- **Common Language Specification (CLS):** Common Language Specification: It provides the language Interoperability on .NET Platform
- **Common Type System (CTS):** CTS is responsible for understanding all the data type systems of .NET programming languages and converting them into CLR understandable format which will be a common format

Thread Overhead and Thread Pool

Time: the time required to set the memory stack and spinning the the thread ~ few hundred milliseconds

Memory: each thread consumes ~ 1MB of memory

Thread Pool: sharing and recycling threads (specifically for background threads) to reduce the consumption of the resources

Background Threads: similar to foreground threads but the managed execution environment is only active when the main thread is active; in other words, they die when main thread dies.

Thread Pool

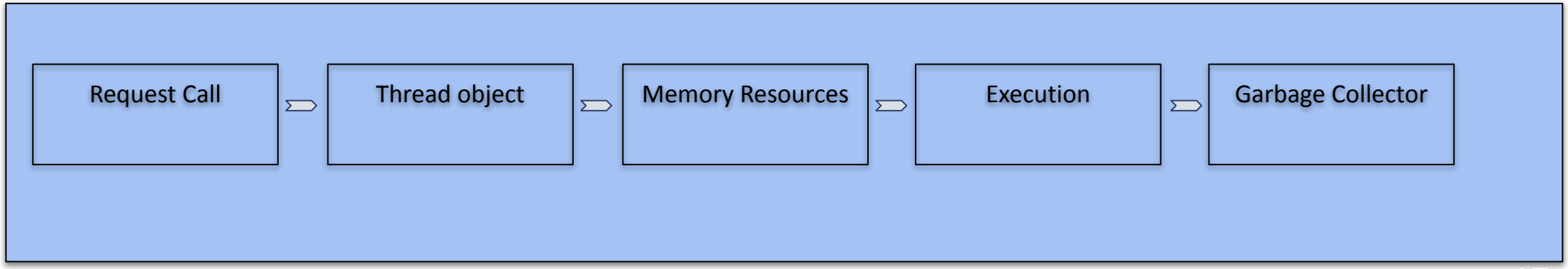
- Limits the number of parallel threads
- Queues the additional threads until the already existing threads at capacity finish executing
- How do you know if Thread Pool is being used?
  - `Thread.CurrentThread.IsThreadPoolThread`
- How to add to Thread Pool?
  - Task Parallel Library
  - Background Work
  - Asynchronous delegate
  - Call ThreadPool.QueueUserWorkItem*

Thread Pool

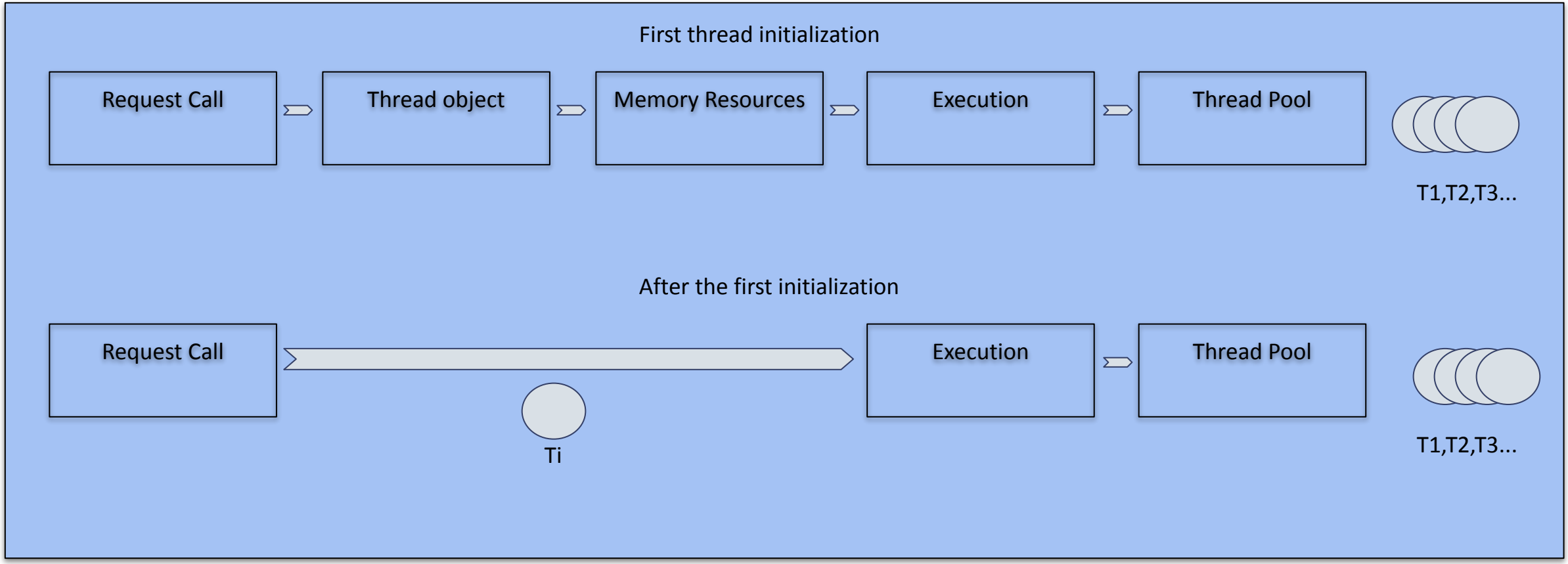
```
using System;
using System.Threading;
public class Example
{
    public static void Main()
    {
        // Queue the task.
        ThreadPool.QueueUserWorkItem(ThreadProc);
        Console.WriteLine("Main thread does some work, then sleeps.");
        Thread.Sleep(1000);
        Console.WriteLine("Main thread exits.");
    }

    // This thread procedure performs the task.
    static void ThreadProc(Object stateInfo)
    {
        // No state object was passed to QueueUserWorkItem, so stateInfo is null.
        Console.WriteLine("Hello from the thread pool.");
    }
}
```

Thread Life Cycle

- A request is made
- Thread object is created and memory is allocated
- The thread executes its task
- The garbage collector will removes the thread objects and frees up the memory



Thread Pool Life Cycle



Thread Granularity

Granularity is used to describe the degree of **parallelism** that exists inside a system.

A program (process) could be broken down into a number of parallel executing **threads**, each representing a **traceable path of sequential programming**

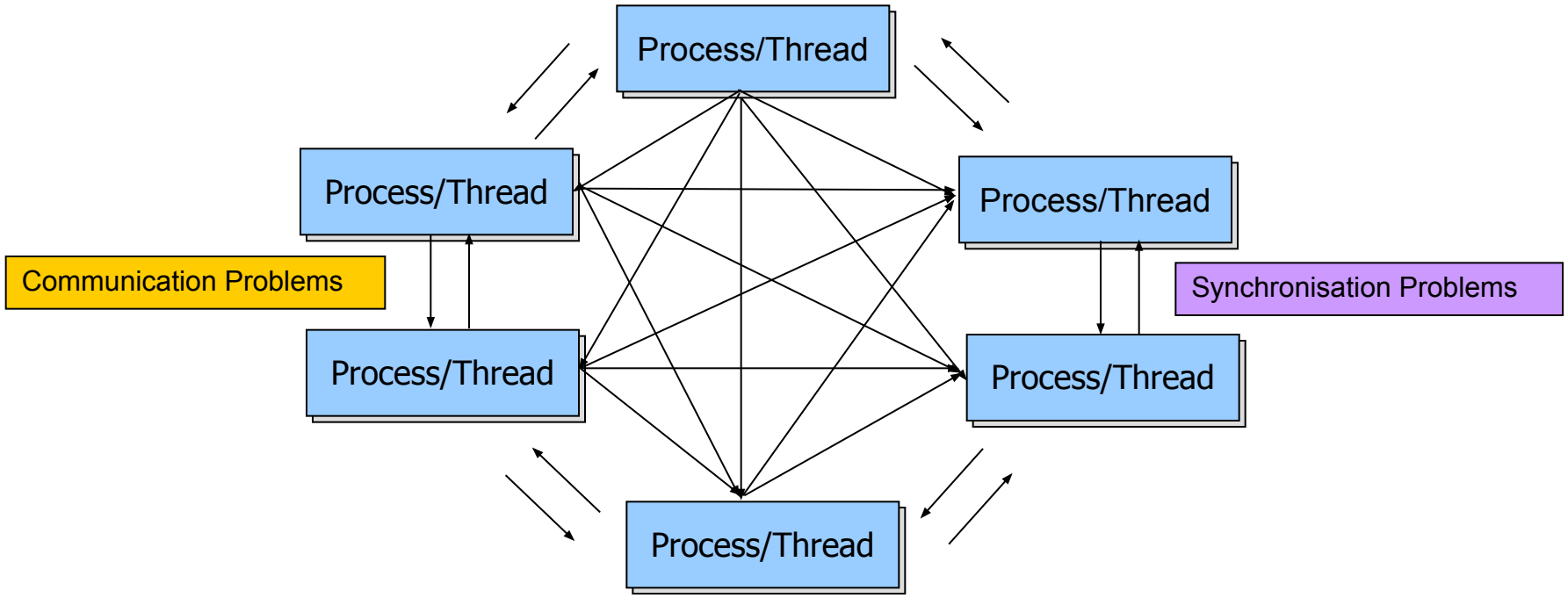
Threads allow a single application to be run on multiple cores within the hardware.

In theory we could decompose our system into **finer and finer** units until almost everything runs in parallel.

Designing too much parallelism leads to increases in **data dependencies** reducing the amount of real world parallelism taking place.

Communication and Synchronization Problems

More threads more problems. **Communication** and **synchronization** issues grow exponentially, scattering data leads to more **data dependencies**, slowing the system down, leading to **data management** challenges.



Thread Communication – Shared Memory

```
using System;
using System.Threading;

namespace SharedResources
{
    class Program
    {
        private static bool isCompleted;

        static void Main(string[] args)
        {
            Thread thread = new Thread(HelloWorld);
            //Worked Thread
            thread.Start();
            //Main Thread
            HelloWorld();
        }

        private static void HelloWorld()
        {
            if (!isCompleted)
            {
                Console.WriteLine("Hello World should print only once");
                isCompleted = true;
            }
        }
    }
}
```

Thread Communication – Shared Memory

```
using System;
using System.Threading;

namespace SharedResources
{
    class Program
    {
        static readonly object lockCompleted = new object();

        static void Main(string[] args)
        {
            Thread thread = new Thread(HelloWorld);
            //Worked Thread
            thread.Start();
            //Main Thread
            HelloWorld();
        }
```

```
        private static void HelloWorld()
        {
            lock (lockCompleted)
            {
                if (!isCompleted)
                {
                    Console.WriteLine("Hello World should print only once");
                    isCompleted = true;
                }
            }
        }
    }
}
```

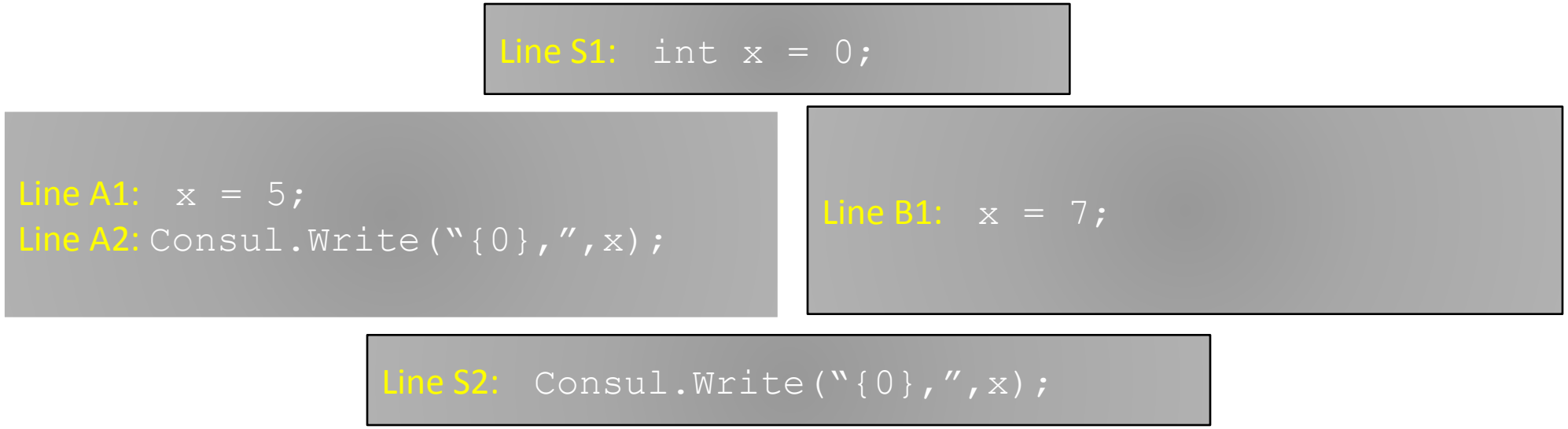
Thread Communication – Message Passing

Pass an argument into Thread's Start method:

```
static void Main() {
     Thread t = new Thread (Print);
     t.Start ("Hello from t!");
}
static void Print (object messageObj) {
     string message = (string) messageObj; // We need to cast here
     Console.WriteLine (message);
}
```

Race Conditions

A **race condition** is when the the behaviour or output of a program depends on the precise **sequence** or **timings** of events.



Is it possible to get… 0, 5,? 0, 7,? 5, 7,? 7, 7,? 5, 5,? 7, 5,?

Race Conditions

A **race condition** is when the the behaviour or output of a program depends on the precise **sequence** or **timings** of events.

Q: Will we get the same output every time?
Q: What might influence the output?

Considering these situations is important when designing concurrent systems.

- What **execution orders** ***are*** **possible** and what are the **effects** of that order?
- How do we enforce a **particular execution order**? – **synchronisation**.

Threads in Practice

- Every program has at least one main thread
- Most graphical programs have a separate thread to control the GUI

How many threads?

- Main program thread
- GUI
- Spell-check
- Backups
- Save in background

Refernces

- Threading in C# by Joseph Albahari (http://www.albahari.info/threading/threading.pdf)
- Threading in C# by Chander Dahl (https://www.linkedin.com/learning/instructors/chander-dhall?u=26890602)
- Common Language Runtime (CLR) in C# by Anshul_Aggarwal
- https://www.geeksforgeeks.org/common-language-runtime-clr-in-c-sharp/?ref=rp