Professional Software Engineering

Andrea Carrara and Patrick Berggold

Hritik Singh and Mohab Hassaan – Tutors

Chair of Computational Modeling and Simulation

Lecture schedule

» Parallel Programming

» Asynchronous Programming

» PLINQ

andrea.carrara@tum.de; patrick.berggold@tum.de

PARALLEL PROGRAMMING

Parallel Programming

» Software development technique to execute program parts simultaneously

» Three ways to obtain parallel programming

- Multi-processor programming: use of multiple cores in the processor
- Mutli-threading programming: more threads in a single core
- Distributed programming: more computer simultaneously

Pros and Cons of Parallel Programming

- PROS:
  - Reduced execution time
  - Efficiency of data elaboration
  - Optimal use of processing power
  - Better scalability
- CONS:
  - Complexity: managing multiple threads, harder to debug and test
  - Synchronization: necessary with share resources
  - Overhead: can affect the overall performance of the program
  - Hardware limitations: multiple processors or cores are required
  - Not all problems are suitable for parallelization: the tradeoff between overhead and efficiency

andrea.carrara@tum.de; patrick.berggold@tum.de

Program, Process and Thread

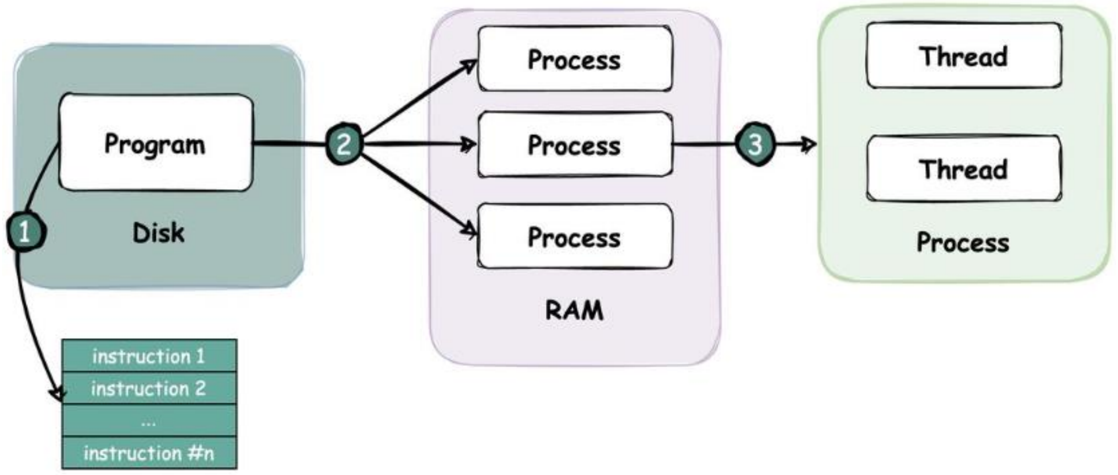
» **Program**: the set of instructions to perform a specific task. Collection of code.

» **Process**: Instance of a program executed by a computer. The operating system creates it and contains and holds the resources that the program needs to run.

» **Thread**: the smallest unit of execution in a process.

» Creation and destruction of thread are less expensive than processes.

» OS handles processes, the application handles threads



Parallel Programming in .NET – Task Parallel Library (TPL)

» **Purpose**: simplifying the process of adding parallelism and concurrency to applications

» **Features**: dynamically scales concurrency, handles partitioning and scheduling of work, provides cancellation support and state management

» TPL is the preferred way to write multithreaded and parallel code in .NET Framework 4

Introduction to Data Parallelism

» **Definition**: performing the same operation concurrently on elements in a source collection

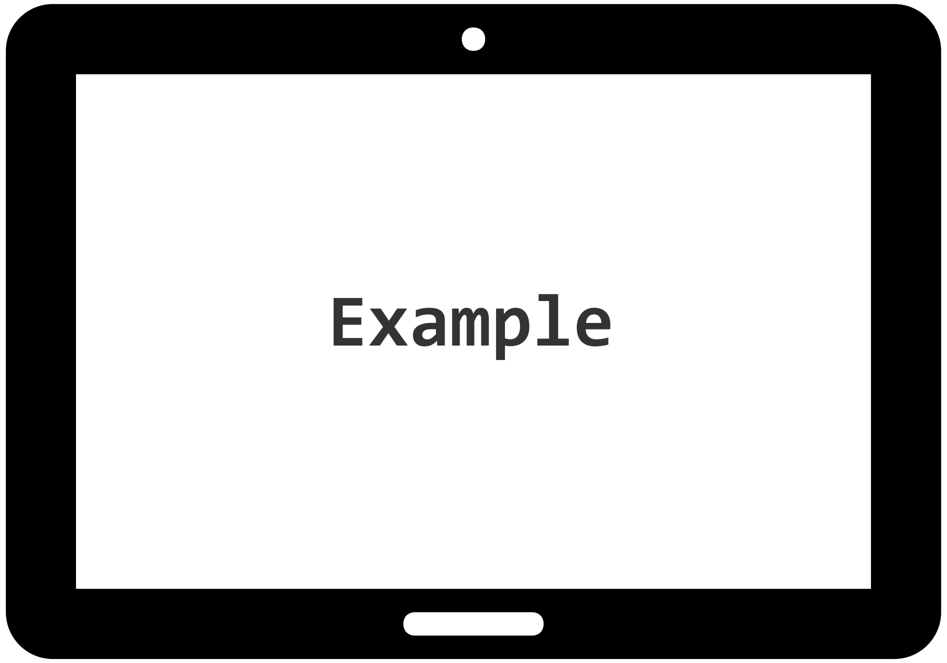
» TPL supports data parallelism through the *System.Threading.Tasks.Parallel* class

» The parallel class provides parallel implementations of **for** and **foreach** loops **(Parallel.For** and **Parallel.ForEach) .** Loop logic is written as in sequential loop

» No need to create threads or queue work items, or take locks,TPL handles low-level work

```
// Sequential version
foreach (var item in sourceCollection)
{
    Process(item);
}
// Parallel equivalent
Parallel.ForEach(sourceCollection, item => Process(item));
```

Matrix Multiplication – For and Parallel.For



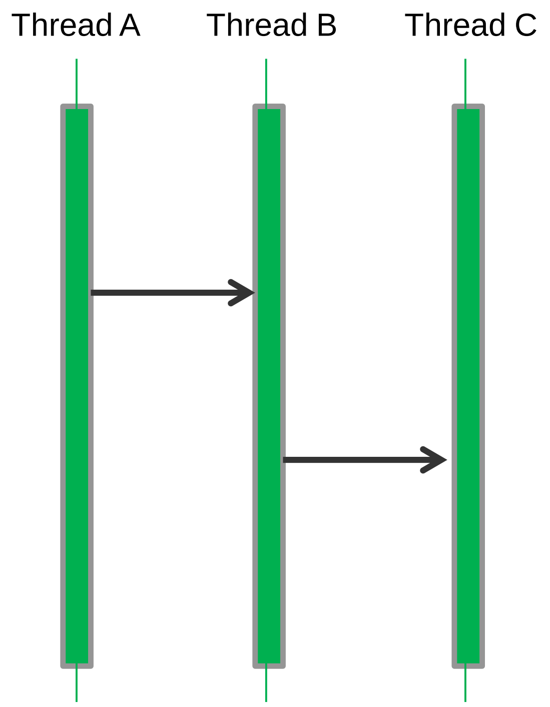
Matrix Multiplication – For and Parallel.For

- » Importance of finding the right balance in parallelization
- » Utilize processors as much as possible without over parallelizing

- » Example of parallelizing only the outer loop
- » Inner loop has small amount of work and may have negative cache effects
- » Parallelizing outer loop only maximizes concurrency benefits on most systems

Threads overview

- » Multiple execution threads in one single program
- » Required for interactive GUI applications
- » Required to make use of multicore processors
- » Access the same data
- » .NET:
  - – **System.Threading**
  - – **Task Parallel Library**



Scheduler / Thread States

» Scheduler:

  – manages threads on a single CPU using a waiting queue

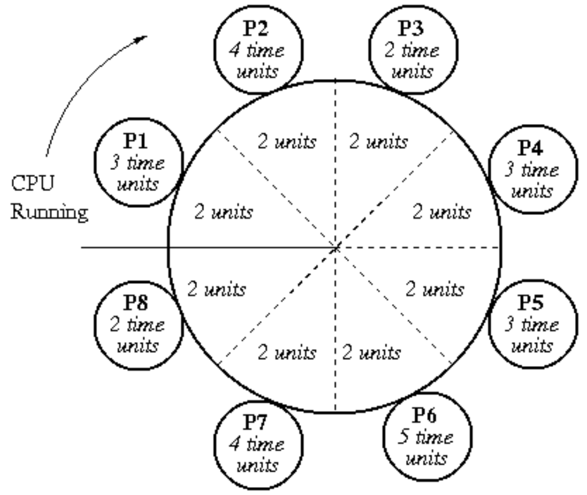
  – each thread gets a time quantum (default: 20ms)

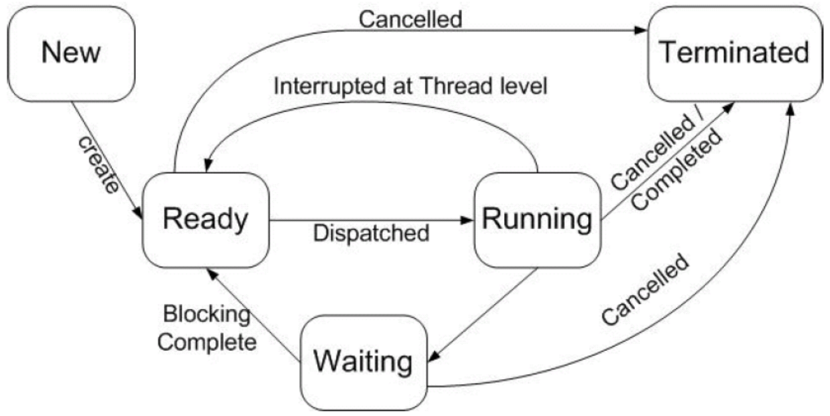
» Thread States

  – ready → in queue

  – running → is being executed

  – waiting → not in queue



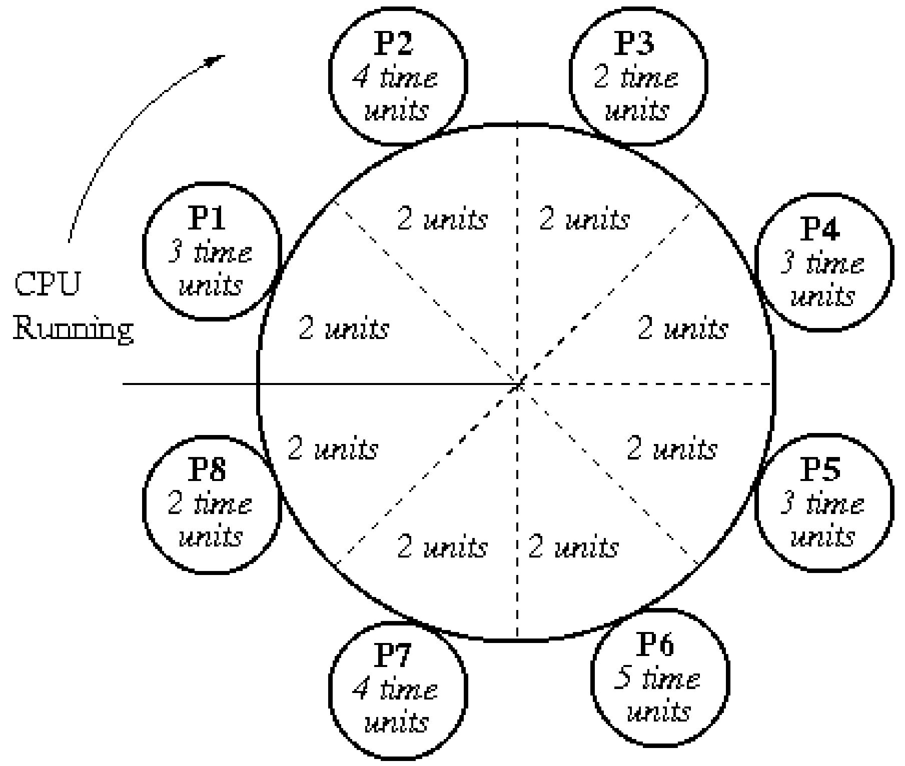


andrea.carrara@tum.de; patrick.berggold@tum.de

Scheduler / Thread States

» A thread has to leave the processor when
  – the time quantum has passed
  – the thread decides to wait
  – a thread with a higher priority enters the waiting queue

» Priority
  – each thread has a priority
  – required for foreground/background processes



Threads – Programming example

» namespace System.Threading

» class Thread

– Constructor

public Thread(ThreadStart start);

– Delegate

public delegate void ThreadStart()

– Method Start()

```
using System.Threading;

class Program
{
    static void Main(string[] args)
     {
         // create and assign the delegate
         ThreadStart del = new ThreadStart(TestMethod);

         // create the thread
         Thread thread = new Thread(del);

         // start the second thread
         thread.Start();

         for (int i = 0; i <= 100; i++)
         {
             for (int k = 1; k <= 20; k++)
                 Console.Write(".");
             Console.WriteLine("Primary Thread " + i);
         }
         Console.ReadLine();
     }

     // this method is executed in a separate thread
     public static void TestMethod()
     {
         for (int i = 0; i <= 100; i++)
         {
             for (int k = 1; k <= 20; k++)
                 Console.Write("X");
             Console.WriteLine("Secondary Thread " + i);
         }
     }
}
```

Threads – Programming example

» If you have to pass parameters to the thread method, use the delegate public delegate void ParameterizedThreadStart(object obj);

» To this delegate, general object types can be passed

```
class Program
{
    static void Main(string[] args)
    {

        //create and assign the delegate
        ParameterizedThreadStart del = new ParameterizedThreadStart(TestMethod);

        //create the thread
        Thread thread = new Thread(del);

        Object anyObject = new Object();

        // start the second thread
        thread.Start(anyObject);

        for (int i = 0; i <= 100; i++)
        {
            for (int k = 1; k <= 20; k++)
                Console.Write(".");
            Console.WriteLine("Primary Thread " + i);
        }
        Console.ReadLine();
    }

    // this method is executed in a separate thread

    public static void TestMethod(Object myObject)
    {
        for (int i = 0; i <= 100; i++)
        {
            for (int k = 1; k <= 20; k++)
                Console.Write("X");
            Console.WriteLine("Secondary Thread " + i);
        }
}}
```

Threads – More functionality

» You can always access the current thread by calling
Thread.CurrentThread

» Assign the current thread's priority by
Thread.CurrentThread.Priority = …;

» Let the thread rest by calling
Thread.Sleep(ms);

```
class Program
{
    static void Main(string[] args)
    {
        Thread.CurrentThread.Priority = ThreadPriority.Normal;
        Demo obj = new Demo();
        ThreadStart del = new ThreadStart(obj.TestMethod);

        Thread thread = new Thread(del);
        thread.Start();

        Console.WriteLine("Thread has been started");
        Thread.Sleep(2000);
}}
class Demo
{
        public void TestMethod()
        {
            try
            {
                Console.WriteLine("Secondary thread started.");

                for (int i = 0; i <= 100; i++)
                {
                    Console.WriteLine("counter = {0}", i);
                    Thread.Sleep(50);
                }
            }
            catch
            {Console.WriteLine("Something happened");}
        }
     }
```

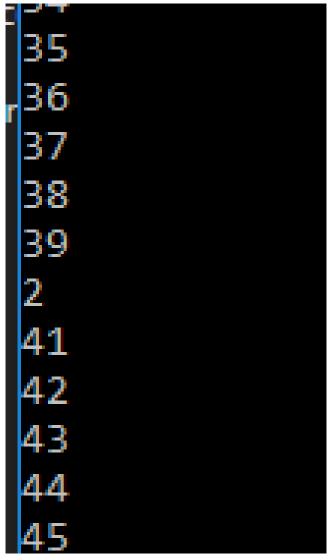
Threads – Synchronizing threads

» Thread safety:

- An object remains in a valid state, also when multiple threads are accessing it at the same time
- Multiple threads can call the same methods without causing conflicts, inconsistencies or other problems
- Such code is called thread-safe
- Code that is not thread-safe can generate different results (depending on the scheduling schema)

```
class Program
{
    static int sum = 0;
    static void Main(string[] args)
    {
        Thread t1 = new Thread(sumItUnsafe);
        Thread t2 = new Thread(sumItUnsafe);

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();

        System.Console.WriteLine(sum);
    }
    static void sumItUnsafe()
    {
        for (int i = 0; i < 50000; i++)
        {
            sum = sum + 1;
        }
}}
```

Threads – Unsynchronized threads



```
35
36
37
38
39
2
41
42
43
44
45
```

```
class Program
{
   static void Main(string[] args)
    {
        Demo obj = new Demo();
        Thread thread1, thread2;
        thread1 = new Thread(new ThreadStart(obj.Worker));
        thread2 = new Thread(new ThreadStart(obj.Worker));
        thread1.Start();
        thread2.Start();
        Console.ReadLine();
    }
}
class Demo
{
    private int value;
    public void Worker()
    {
        while (true)
        {
            value++;
            if (value > 100) break;
            Console.WriteLine(value);
        }
    }
}
//output: 1, 2, 3, 4, ... , 39, 41, 42, 43, ..., 99, 100, 40,
41
```

Threads – Synchronizing threads

» Thread-safe methods should make sure the underlying data is in a valid state when they are forced to wait

» class Monitor

» Use Monitor.Enter() and Monitor.Exit() to define critical code blocks

» Only one thread is allowed to access the critical code block at a time

```
class Demo
{
    private int value;

    public void Worker()
    {
        while (true)
        {

            // set lock
            Monitor.Enter(this);

            value++;
            if (value > 100) break;
            Console.WriteLine(value);
            Thread.Sleep(5);

            // release lock
            Monitor.Exit(this);
        }
    }
}
//output: 1, 2, 3, 4, ... , 39, 40, 41, 42, 43, ..., 99, 100
```

Synchronizing processes

- Use Mutex to synchronize processes
- Each mutex object is identified by a unique name
- Only one thread can access the critical code block
- WaitOne(): request ownership
  → returns true if ownership granted
- ReleaseMutex() releases ownership

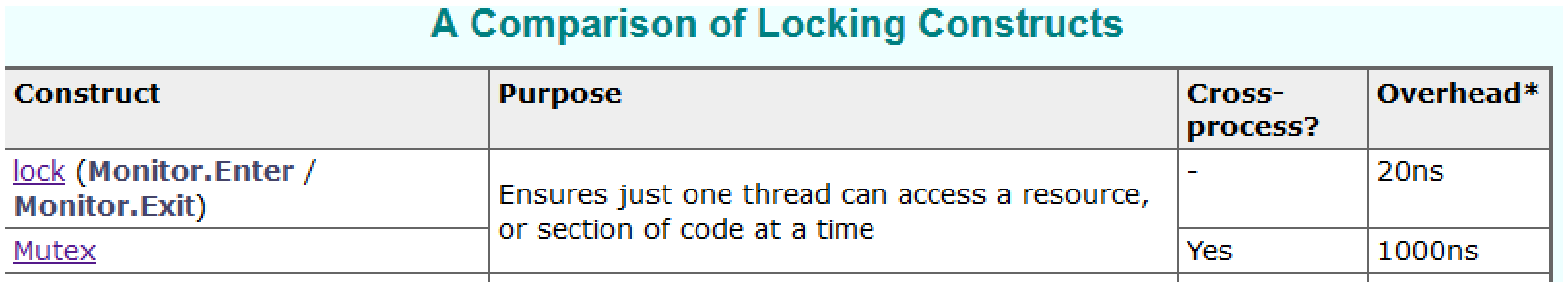
```
using System;
using System.Threading;
class Program
{
    static Mutex mutex = new Mutex();
    static int sum = 0;
    static void Main(string[] args)
    {
        Thread t1 = new Thread(sumIt);
        Thread t2 = new Thread(sumIt);

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();

        System.Console.WriteLine(sum);
    }
    static void sumIt()
    {
        for (int i = 0; i < 50000; i++)
        {
            // Wait until it is safe to enter.
            mutex.WaitOne();
            sum = sum + 1;

            // Release the Mutex.
            mutex.ReleaseMutex();
}}}
```

Synchronizing processes

» Lock == Threads, Mutex == Processes

» Lock is faster and more convenient

» Mutex can be cross applications!



A Comparison of Locking Constructs

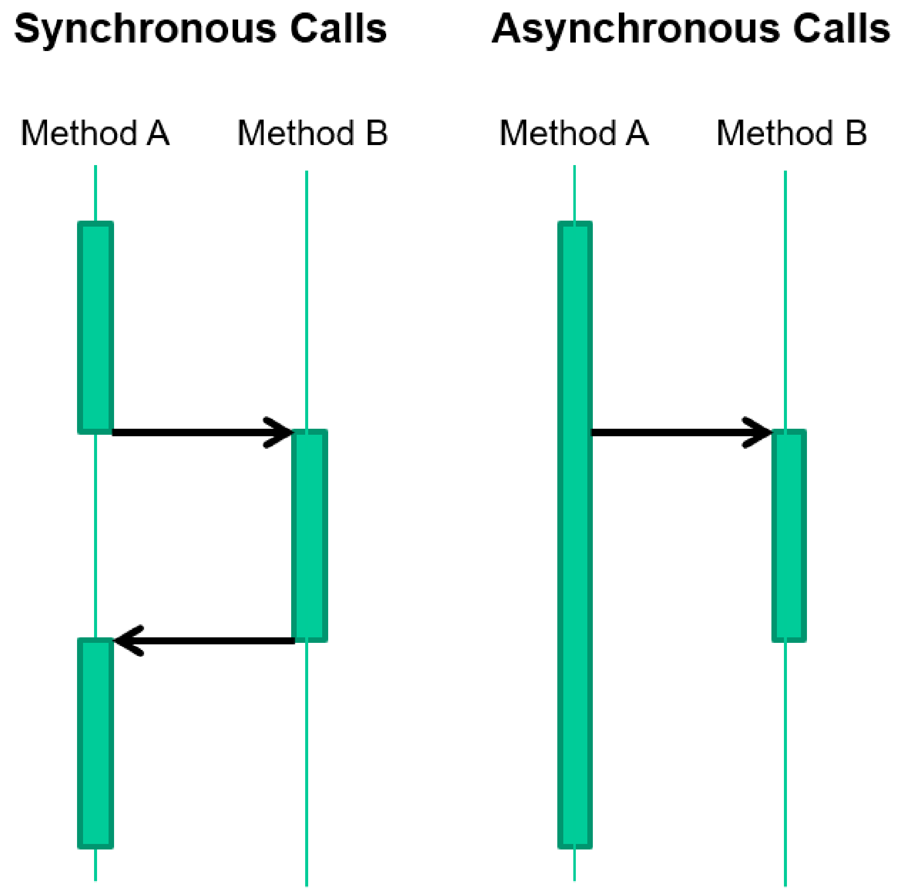
Construct	Purpose	Cross-process?	Overhead*
lock (**Monitor.Enter / Monitor.Exit**)	Ensures just one thread can access a resource, or section of code at a time	-	20ns
Mutex		Yes	1000ns

» Since 4.0 there is also a spinlock (high-contention) scenarios

» Keeps thread busy instead of switching!

Asynchronous calls

- Starting a long-running operation in a separate thread
- Continuing to execute rest of program while operation is running in background
- Allows program to remain responsive to user input or other events while long-running operation is in progress
- Often used with remote server requests or time-consuming calculations
- **asynchronous programming** : way to manage the flow of a program
- **parallel programming** : way to improve performance by executing operations concurrently



Tasks implicitly

- Can run any number of arbitrary statements concurrently
- Uses Action delegate for each item of work, can be created using lambda expressions
- Example of using Parallel.Invoke with two tasks created using lambda expressions
- First task calls DoSomeWork method, the second task calls DoSomeOtherWork method
- Both tasks run concurrently

```
Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());
```

Tasks implicitly

» Task object handles infrastructure details and provides methods and properties accessible from calling thread

» Creating a task by giving it a user delegate (named delegate, anonymous method, lambda expression)

» Example using lambda expression with call to named method

» Call to Task.Wait method to ensure task completes execution before application ends

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        Thread.CurrentThread.Name = "Main";

        // Create a task and supply a user delegate by using a 
lambda expression.
        Task taskA = new Task(() => Console.WriteLine("Hello 
from taskA."));
        // Start the task.
        taskA.Start();

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.", 
                          Thread.CurrentThread.Name);
        taskA.Wait();
    }
}
// The example displays output like the following:
//       Hello from thread 'Main'.
//       Hello from taskA.
```

```
// Define and run the task.
Task taskA = Task.Run( () => Console.WriteLine("Hello from taskA."));
```

TaskFactory.StartNew method

» Can create and start a task in one operation

» When creation and scheduling don't need to be separated

» When additional task creation options or specific scheduler are required

» When need to pass additional state into task that can be retrieved through Task.AsyncState property

Tasks implicitly

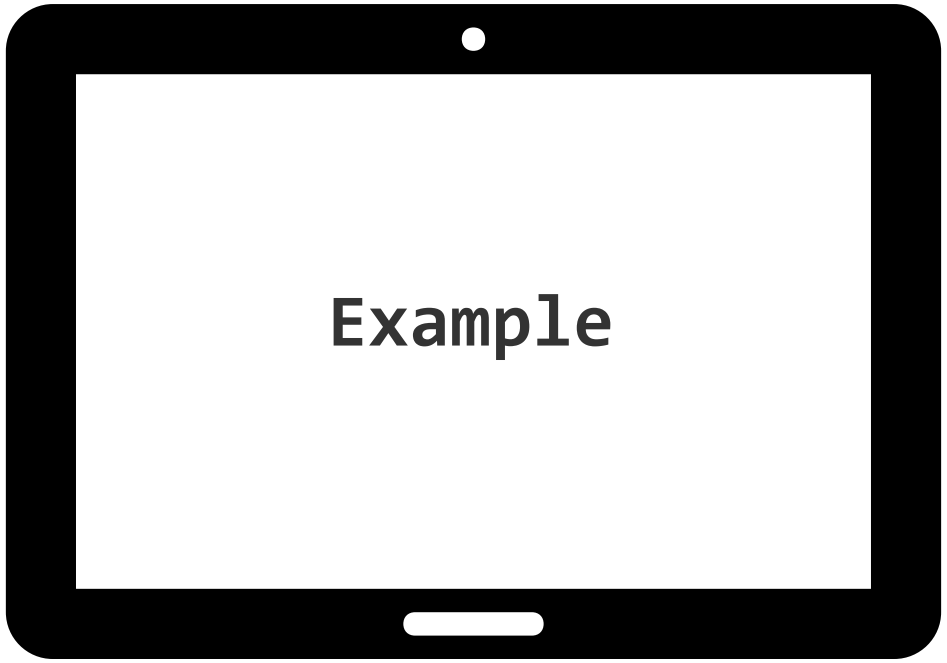
```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
    public long CreationTime;
    public int Name;
    public int ThreadNum;
}
```

1

```
public class Example
{
    public static void Main()
    {
        Task[] taskArray = new Task[10];
        for (int i = 0; i < taskArray.Length; i++)
        {
            taskArray[i] = Task.Factory.StartNew((Object obj) => {
                CustomData data = obj as CustomData;
                if (data == null)
                    return;

                data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
            },
                                                 new CustomData() { Name = i, CreationTime = DateTime.Now.Ticks });
        }
        Task.WaitAll(taskArray);
        foreach (var task in taskArray)
        {
            var data = task.AsyncState as CustomData;
            if (data != null)
                Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                                  data.Name, data.CreationTime, data.ThreadNum);
        }
    }
}
```

2

Prime numbers: Synchronous, Parallel and Asynchronous



Common functions among the programs

```
private static List<int> GetPrimeNumbers(int minimum,
int maximum)
        {
            List<int> result = new List<int>();
            for (int i = minimum; i <= maximum; i++)
            {
                if (IsPrimeNumber(i))
                {
                    result.Add(i);
                }
            }
            return result;

        }
```

```
static bool IsPrimeNumber(int number)
        {
            if (number % 2 == 0)
            {
                return number == 2;
            }
            else
            {
                var topLimit = (int)Math.Sqrt(number);

                for (int i = 3; i <= topLimit; i += 2)
                {
                    if (number % i == 0) return false;
                }
                return true;
            }
```

» using System; using System.Collections.Generic; using System.Diagnostics;

» using System.Linq; using System.Threading.Tasks;

Synchronous, Parallel and Asynchronous main

```
var sw = new Stopwatch();
sw.Start();
var primes = GetPrimeNumbers(2, 10000000);
Console.WriteLine("Total prime numbers: {0}\nProcess time: {1}", primes.Count, sw.ElapsedMilliseconds);
```

```
var sw = new Stopwatch();
sw.Start();
const int numParts = 10;
var primes = new List<int>[numParts];
Parallel.For(0, numParts, i => primes[i] = GetPrimeNumbers(i == 0 ? 2 : i * 1000000 + 1, (i + 1) * 1000000));
var result = primes.Sum(p => p.Count);
Console.WriteLine("Total prime num  ers: {0}\nProcess time: {1}", result, sw.ElapsedMilliseconds);
```

```
var sw = new Stopwatch();
sw.Start();
const int numParts = 10;
Task<List<int>>[] primes = new Task<List<int>>[numParts];
for (int i = 0; i < numParts; i++)
{
    primes[i] = GetPrimeNumbersAsync(i == 0 ? 2 : i * 1000000 + 1, (i + 1) * 1000000);
}
var results = await Task.WhenAll(primes);
Console.WriteLine("Total prime numbers: {0}\nProcess time: {1}", results.Sum(p => p.Count), sw.ElapsedMilliseconds);
```

GetPrimeNumbersAsync

```
private static async Task<List<int>>
GetPrimeNumbersAsync(int minimum, int maximum)
{
    var count = maximum - minimum + 1;
    List<int> result = new List<int>();

    return await Task.Factory.StartNew(() =>
    {
        for (int i = minimum; i <= maximum; i++)
        {
            if (IsPrimeNumber(i))
            {
                result.Add(i);
            }
        }
        return result;
    });
}
```

PLINQ

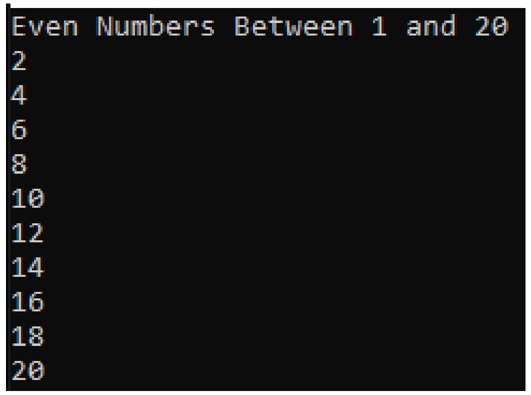
- » Parallel implementation of LINQ
- » Operates on in-memory IEnumerable or IEnumerable<T> data source
- » Deferred execution, does not begin until query is enumerated
- » Attempts to make full use of all processors on system by partitioning data source and executing query on separate worker threads in parallel
- » Parallel execution can achieve significant performance improvements
- » Parallelism can introduce complexities and not all query operations run faster in PLINQ

ParallelEnumerableOperator

ParallelEnumerable Operator	Description
AsParallel	The entry point for PLINQ. Specifies that the rest of the query should be parallelized, if it is possible.
AsSequential	Specifies that the rest of the query should be run sequentially, as a non-parallel LINQ query.
AsOrdered	Specifies that PLINQ should preserve the ordering of the source sequence for the rest of the query, or until the ordering is changed, for example by the use of an orderby (Order By in Visual Basic) clause.
AsUnordered	Specifies that PLINQ for the rest of the query is not required to preserve the ordering of the source sequence.
WithCancellation	Specifies that PLINQ should periodically monitor the state of the provided cancellation token and cancel execution if it is requested.
WithDegreeOfParallelism	Specifies the maximum number of processors that PLINQ should use to parallelize the query.
WithMergeOption	Provides a hint about how PLINQ should, if it is possible, merge parallel results back into just one sequence on the consuming thread.

LINQ Call

```
public static void Main()
    {
        var numbers = Enumerable.Range(1, 20);
        //LINQ
        var evenNumbers = numbers.Where(x => x % 2 == 0).ToList();
        Console.WriteLine("Even Numbers Between 1 and 20");
        foreach (var number in evenNumbers)
        {
            Console.WriteLine(number);
        }
        Console.ReadKey();
    }
```

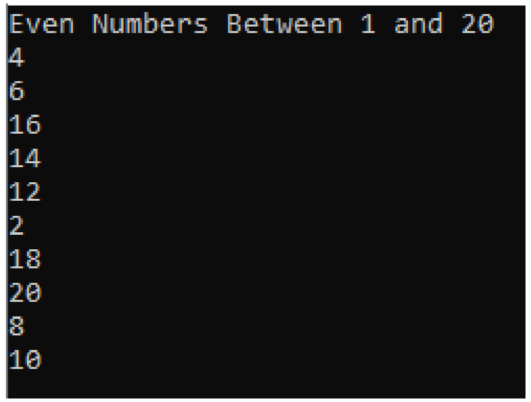


```
Even Numbers Between 1 and 20
2
4
6
8
10
12
14
16
18
20
```

PLINQ Call

```
public static void Main()
    {
        var numbers = Enumerable.Range(1, 20);
        //PLINQ call
        var evenNumbers = numbers.AsParallel().Where(x => x % 2 == 0).ToList();
        Console.WriteLine("Even Numbers Between 1 and 20");
        foreach (var number in evenNumbers)
        {
            Console.WriteLine(number);
        }

        Console.ReadKey();
    }
```

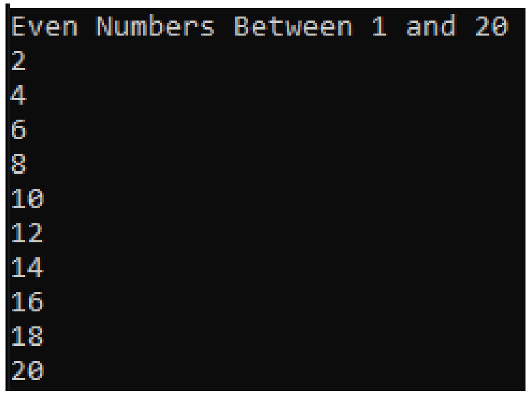
Parallel evaluation



```
Even Numbers Between 1 and 20
4
6
16
14
12
2
18
20
8
10
```

Ordered results

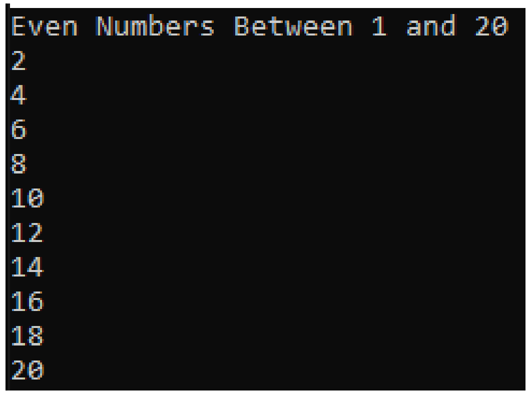
```
public static void Main()
    {

        var numbers = Enumerable.Range(1, 20);
        //PLINQ
        var evenNumbers = numbers
            .AsParallel() //Parallel Processing
            .AsOrdered() //Original Order of the numbers
            .Where(x => x % 2 == 0)
            .ToList();
        Console.WriteLine("Even Numbers Between 1 and 20");
        foreach (var number in evenNumbers)
        {
            Console.WriteLine(number);
        }

        Console.ReadKey();
    }
```



```
Even Numbers Between 1 and 20
2
4
6
8
10
12
14
16
18
20
```

Cancellation Token

```
var CTS = new CancellationTokenSource();
CTS.CancelAfter(TimeSpan.FromMilliseconds(2000));
var numbers = Enumerable.Range(1, 20);
try
{
    var evenNumbers = numbers
        .AsParallel() //Parallel Processing
        .AsOrdered() //Original Order of the numbers
        .WithDegreeOfParallelism(2) //Maximum of two threads can process the data
        .WithCancellation(CTS.Token) //Cancel the operation after 200 Milliseconds
        .Where(x => x % 2 == 0) //This logic will execute in parallel
        .ToList();
    Console.WriteLine("Even Numbers Between 1 and 20");
    foreach (var number in evenNumbers)
    {
        Console.WriteLine(number);
    }
}
catch (System.OperationCanceledException ex)
{
    Console.WriteLine("exception");
}
```



```
Even Numbers Between 1 and 20
2
4
6
8
10
12
14
16
18
20
```