Computer Science
Engineering School

Software
Engineering

Programming Technologies and Paradigms

Foundations of Concurrent and Parallel Programming

Francisco Ortín Soler

University of Oviedo

Code Examples

- All the example **code** shown in these slides is **available** for download
  - The directory is shown in underlined blue font (such as threads/daemon)
- To understand the concepts explained, the source code must be **opened**, **analyzed**, **modified**, **executed** by the students, making sure that they **understand** the code

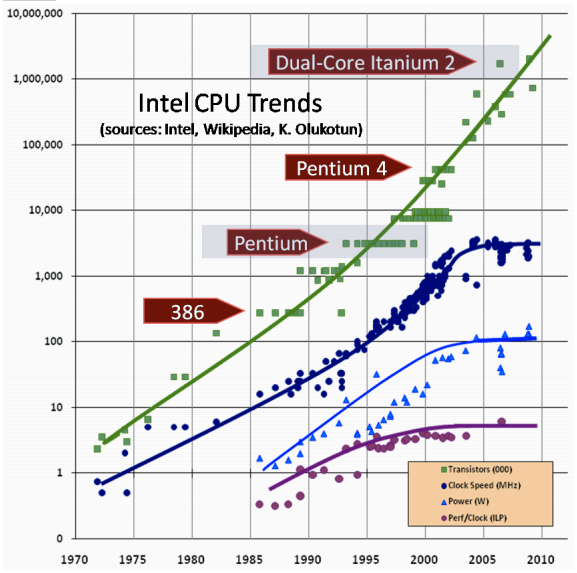
Content

- Introduction
- Concurrent and Parallel Programming
- Process and Thread
- Algorithm Parallelization
- Asynchronous Message Passing
- Explicit Thread Creation
- Race Condition
- Context Switching and Thread Pooling
- Foreground and Background Threads
- Thread and Process Synchronization
- Deadlock
- Thread-Safe Data Structures
- Parallelization by means of the Task Parallel Library
- Parallelization by means of Higher-Order Functions

Moore's Law

- Is an empirical formulation given by Gordon E. Moore in 1965
- It says that *the number of transistors on integrated circuits doubles approximately every two years*
- Therefore, in two years time, microprocessors are twice as fast as they were two years ago
  - Many digital devices follow this law
- According to Moore, this exponential increase will not be fulfilled in 2017-22 anymore

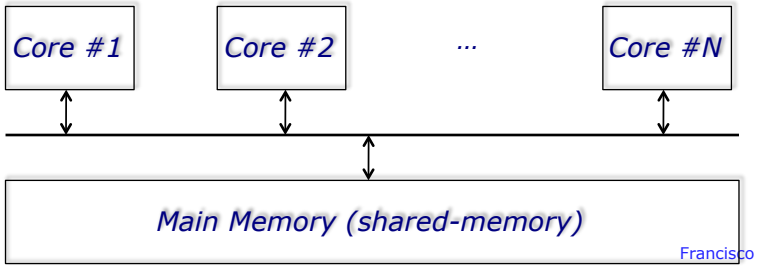
Moore's Law



- The law has been fulfilled regarding the number of transistors
- Since 2003, it has not been fulfilled for the clock speed (MHz)
- So, where are those transistors?

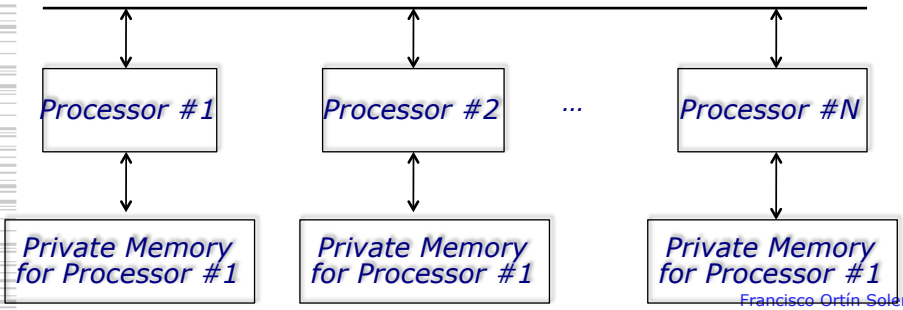
Multi-core Architectures

- The current trend in personal computers is to **increase the number** of processing **cores**, instead of increasing the clock frequency
- Multi-core microprocessors include **multiple processors** (cores) **in the same CPU**
- Parallel computation is provided not only with **multiple processes** but also with multiple **threads**
- They **share** the same **memory**



Distributed Memory

- **Distributed-memory** multiprocessor systems are composed of **many microprocessors** with their own private memory
  - Each processor could, in turn, be a multi-core processor
- Processor interaction is performed through **communication channels**
- Examples: **Clusters** (homogeneous), **Grid** (heterogeneous) and **Cloud** computing



Concurrent Programming

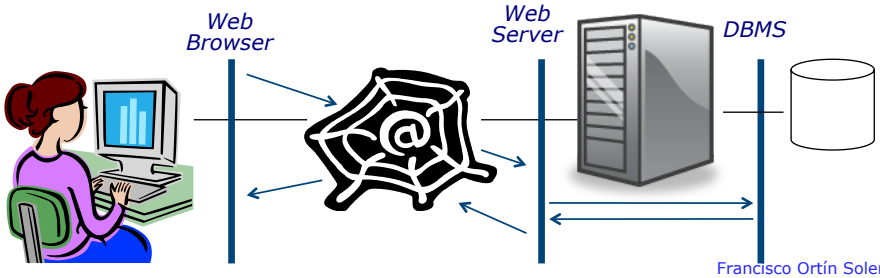
- Due to this trend, concurrent and parallel programming is becoming increasingly important
  - Traditional *sequential code,* where instructions run one after the other, does not take advantage of multiple cores
- **Concurrency** is the property of computational systems in which several **tasks** are being executed **simultaneously**, and are potentially interacting with each other
  - Tasks may be executing in multiple cores or processors, or even *time-shared* in the same single-core processor
- **Tasks** are either **threads** or **processes**

Parallel Programming

- **Parallelism** is a particular scenario of concurrency, where **tasks are executed in parallel** (simultaneously)
  - With concurrency, simultaneity can be simulated
  - In parallelism, simultaneity must be real
- **Parallelism** emphasizes the division of large problems into smaller ones (data, code, tasks…)
- **Concurrency** emphasizes iteration between tasks

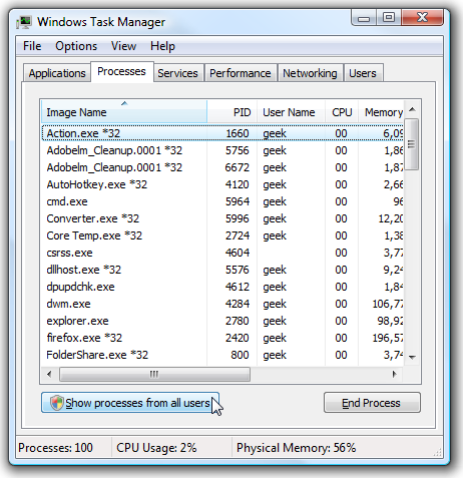
Process

- A **process** is an instance of a program that is being executed
  - Consists of instructions, execution state and runtime data values
  - Different concurrent tasks in distributed-memory systems, executing in different processors, are different processes
- Example of a system with different processes



Processes

- Running processes in Windows are displayed with the task manager (Ctrl+Alt+Del)
- Every process has a unique *Process ID* (PID) –services tag
- In .NET, processes are represented with the **`Process`** class (**`System.Diagnostics`**)



.NET Processes

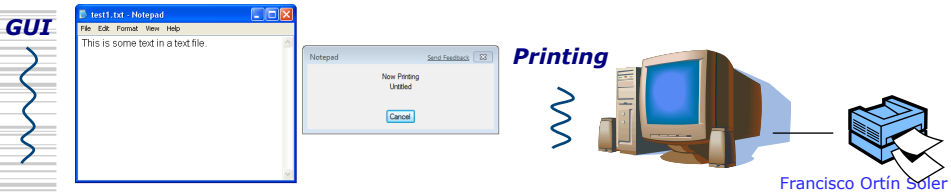
- In .NET, processes are modeled with instances of the **`Process`** class in `System.Diagnostics`

```
var processes = Process.GetProcesses();
foreach (Process process in processes) {
  Console.WriteLine("PID: {0}\tName: {1}\tVirtual
        memory: {2:N} MB", process.Id,
  process.ProcessName,
  process.VirtualMemorySize64/1024.0/1024);
```

- processes.threads/processes

Thread

- A **process** may be made up of **multiple threads**
- A **thread** is a task in a process that is executed concurrently with the rest of threads in that process, sharing the same memory
  - Each thread has its own program counter, execution stack and register values
  - In multi-core processors, threads can be parallel tasks in the same process
- Example of an application with multiple threads:



.NET Threads

- Threads are modeled with instances of the **Thread** class in System.Threading

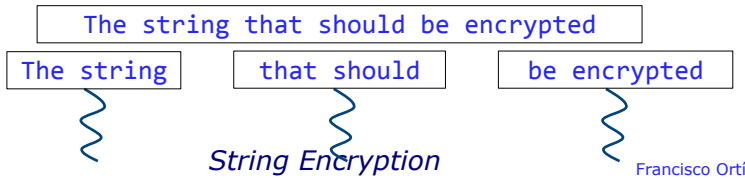
```
Console.WriteLine("Current thread. Name: {0}, id: {1},
  priority: {2}, state: {3}.",
    Thread.CurrentThread.Name,
    Thread.CurrentThread.ManagedThreadId,
    Thread.CurrentThread.Priority,
    Thread.CurrentThread.ThreadState);
```

- processes.threads/threads

Algorithm Parallelization

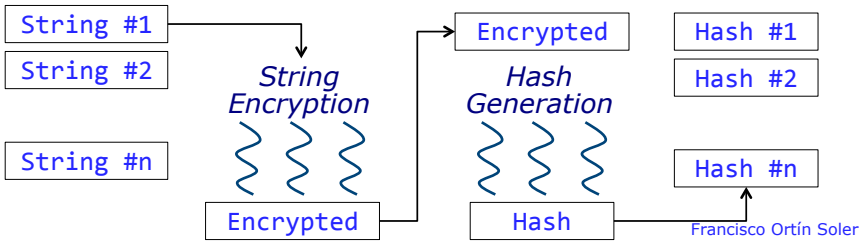
- There are two common parallelization scenarios
  1. **Task parallelism**: independent tasks that can be executed concurrently

  2. **Data parallelism**: performing the same task that computes pieces of the same data



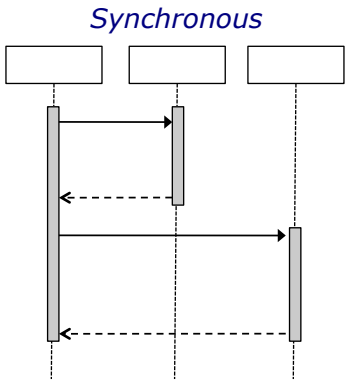
Pipeline

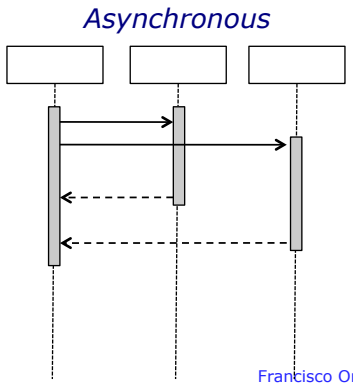
- **Pipeline** is a hybrid data / task parallelism approach
- For example, to compute hashes of encrypted strings, two parallel tasks can be executed (**task parallelism**)
  1. String encryption
  2. Hash generation
- The output of the first task is synchronized to be the input of the second one
- Pieces of the same string are computed (**data parallelism**)



Asynchronous Message Passing

- A first method to create threads is by means of an **asynchronous message passing**
  - Each message creates a new thread
- In C#, this feature is provided using **delegates**

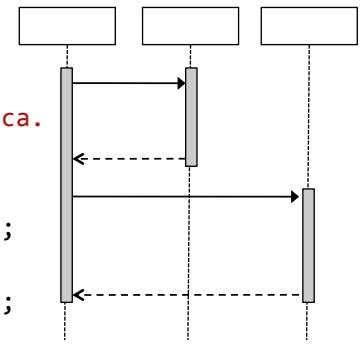




Synchronous Message Passing

- The following program downloads a Web page and counts the number of its `img` tags
- The `GetNumberOfImages` method is synchronous (sequential)

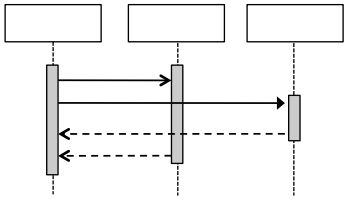
```
WebPage uniovi = new WebPage(
       "http://www.uniovi.es");
WebPage school = new WebPage(
  "http:// www.ingenieriainformatica.
   uniovi.es");
int numberOfImgsInUniovi =
        uniovi.GetNumberOfImages();
int numberOfImgsInSchool =
        school.GetNumberOfImages();
```



- delegates/sequential

Asynchronous Message Passing

1. The first `GetNumberOfImages` message is **asynchronously** passed, creating a new thread
2. The second `GetNumberOfImages` message is **synchronously** passed in the main thread (no new thread is created)
3. The number of images of the second invocation is obtained (implicit wait of the synchronous invocation)
4. Obtain the number of images of the first invocation (waiting till its termination, if it has not ended)
5. Show the results

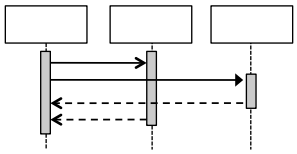


Example

Main Thread

```
WebPage uniovi = new WebPage("http://www.uniovi.es");
WebPage school = new WebPage(
                 "http://www.ingenieriainformatica.uniovi.es");
Func<int> numberOfImages = uniovi.GetNumberOfImages;
IAsyncResult asyncResult = numberOfImages.BeginInvoke(
                 null, null);
int numberImgsSchool = school.GetNumberOfImages();
int numberImgsUniovi = numberOfImages.EndInvoke(
                                     asyncResult);
Console.WriteLine("{0:N0} imgs in Uniovi, {1:N0} in the School."
                 numberImgsUniovi, numberImgsSchool);
```

Secondary Thread



- delegates/concurrent

Asynchronous Message Passing

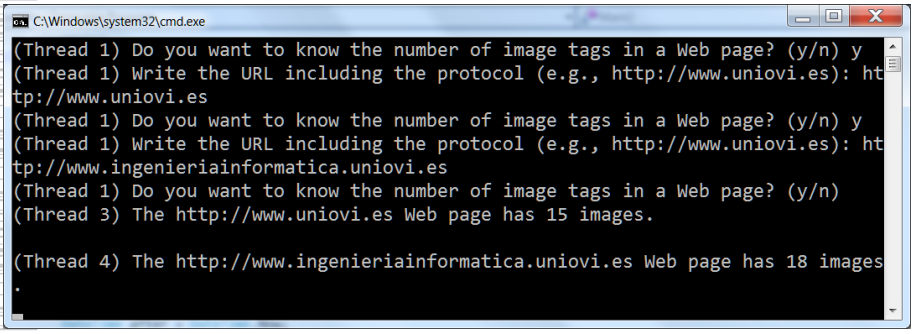
Using Callbacks

- Is it possible to know when the thread ends without waiting?
  - In the previous example, the `EndInvoke` invocation requires waiting (**is synchronous**)
- Upon asynchronous message passing (`BeginInvoke`), a **delegate** can be passed as a parameter
  - The delegate will be called when the execution of the secondary thread ends (callback)
  - Thus, no active waiting is required
  - The main thread runs concurrently
  - However, if the main thread terminates, **the secondary thread will die** (even though it has not ended)

Asynchronous Message Passing

Example (I)

- Application that ask for URLs and asynchronously counts their `img` tags
- delegates/callback

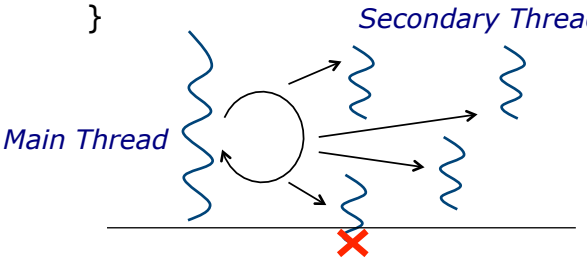


Example (II)

```
string url;
Menu menu = new Menu();
while (menu.Show(out url) != Option.Exit) {
    WebPage web = new WebPage(url);
    ((Func<int>)web.GetNumberOfImages)
        .BeginInvoke(ThreadEnds, url);
    }
```

Callback (delegate)

Parameter passed (object)



Example (III)

```
static void ThreadEnds(IAsyncResult asyncResult) {
    // * We first obtain the method that created the thread
    //   (GetNumberOfImages)
    Func<int> getNumberOfImages = (Func<int>)
            ((AsyncResult)asyncResult).AsyncDelegate;
    Console.WriteLine(
        "\n(Thread {0}) The {1} Web has {2} images.",
        Thread.CurrentThread.ManagedThreadId,
        (string)asyncResult.AsyncState,
        getNumberOfImages.EndInvoke(asyncResult)
        );
    }
```

Object parameter passed when calling BeginInvoke (the URL)

To get the returned value. No waiting is required (the thread has ended)

Explicit Thread Creation

- Using the object-oriented paradigm, the **`Thread`** class (**`System.Threading`**) encapsulates a computing thread
  - It provides another mechanism to write multithreaded applications

```
// We create the thread object
Thread thread = new Thread(delegate);
// We name it (optional)
thread.Name = "Secondary thread";
Thread.CurrentThread.Name = "Main thread";
// We set a scheduling priority (optional)
thread.Priority = ThreadPriority.BelowNormal;
// Indicate to the OS that the thread can start running it
thread.Start();
…
```

Main Thread

Secondary Thread

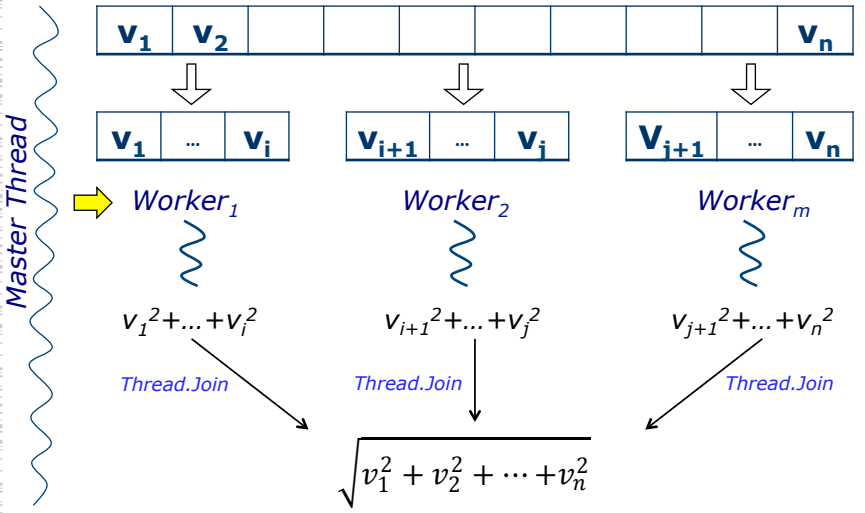
- Question: What is the main difference compared to Java?

Example

- We will analyze an example **concurrent program**, where data parallelism is used
  - Data is divided into pieces, and different threads executing the same algorithm process the data fragments concurrently
- The program computes the length, magnitude or **modulus** of a Euclidean **n-dimensional vector**
- threads/vector.modulus

$$|\overrightarrow{v}| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

Example



Example: Worker

```
internal class Worker {
  private short[] vector;
  private int fromIndex, toIndex;
  private long result;
  internal long Result { get { return this.result; } }
  internal Worker(short[] vector, int fromIndex,
                  int toIndex) {
    this.vector = vector;
    this.fromIndex = fromIndex;
    this.toIndex = toIndex;
  }
  internal void Compute() {
    this.result = 0;
    for(int i= this.fromIndex; i<=this.toIndex; i++)
      this.result += this.vector[i] * this.vector[i];
    }  }  }
```

Explicit Thread Creation

Example: Master (I)

```
public class Master {
  private short[] vector;
  private int numberOfThreads;
  public Master(short[] vector, int numberOfThreads) {
    if ( numberOfThreads < 1 ||
         numberOfThreads > vector.Length )
      throw new ArgumentException(" The number of threads
                must be lower or equal to the elements of
                the vector.");
    this.vector = vector;
    this.numberOfThreads = numberOfThreads;
  }
  …
```

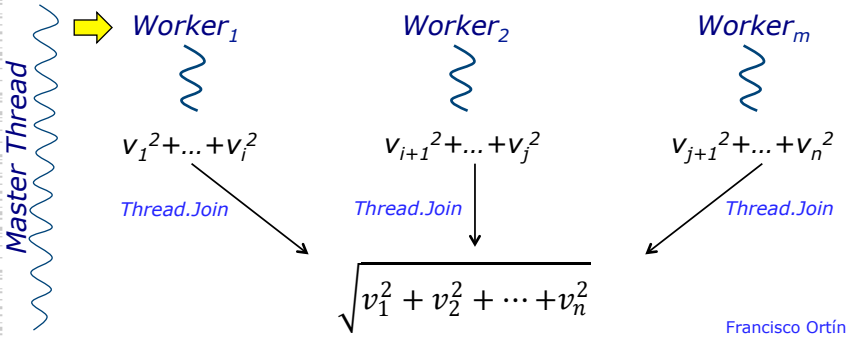
Explicit Thread Creation

Example: Master (II)

```
public double ComputeModulus() {
  Worker[] workers = new Worker[this.numberOfThreads];
  int elementsPerThread = this.vector.Length/numberOfThreads;
  for(int i=0; i < this.numberOfThreads; i++)
    workers[i] = new Worker(this.vector, i*elementsPerThread,
            (i<this.numberOfThreads-1) ? // not the last one?
            (i+1)*elementsPerThread-1: this.vector.Length-1 );
  Thread[] threads = new Thread[workers.Length];
  for(int i=0;i<workers.Length;i++) {
    threads[i] = new Thread(workers[i].Compute);
    threads[i].Start();
  }
  foreach (Thread thread in threads) thread.Join();
  long result = 0;
  foreach (Worker worker in workers)
    result += worker.Result;
  return Math.Sqrt(result);  } }
```

Thread.Join

- In the example, the `Join` message is passed to worker threads in the master thread
- When the `Join` message is passed, **the calling thread is blocked until the thread that is being called terminates**
  - Join is a very basic mechanism of thread synchronization



Race Condition

- What would have happened if `Thread.Join` had not been used?
- The master thread would have taken the worker results (potentially) **sooner** than these results would have been completely computed
  - The final value would change from one execution to another
- The situation where the result of executing multiple concurrent tasks depends on the sequence of execution is called **race condition**
  - Concurrent programs should **never** have race conditions
- Race conditions are the **origin** of multiple **errors** and **vulnerabilities** in concurrent programs and systems (including hardware)

Parameters

- We have used an **object-oriented** approach, encapsulating thread parameters as attributes
- A more **functional approach** can also be used, passing one parameter to `Start` when starting their execution

```
static void Show10Numbers(object from) {
  int? fromInt = from as int?;
  if (!fromInt.HasValue)
    throw new ArgumentException("Argument must be int");
for (int i = fromInt.Value; i < 10 + fromInt; i++) {
    Console.WriteLine(i); Thread.Sleep(1000);
  } }
static void Main() {
  Thread thread = new Thread(Show10Numbers);
  thread.Start(7);  thread.Join();
  }
```

- threads/parameters

Free Variables

- If **lambda functions** are used, the use of free variables should be carefully considered
- Each thread has its own execution stack, **from the stack frame it was created**! (the rest of the stack is shared with the thread that created the new one)
  - Therefore, previous local variables are shared

```
int local = 1;
Thread thread1 = new Thread( () => {
  Console.WriteLine("Thread 1. Local {0}.", local);
  });
local = 2;
Thread thread2 = new Thread( () => {
  Console.WriteLine("Thread 2. Local {0}.", local);
  });
thread1.Start(); thread2.Start();
```

- threads/bound.variables

Alternatives

- Parameter passing (preferable)

```
int local = 1;
Thread thread1 = new Thread( (parameter) => {
  Console.WriteLine("Parameter {0}.", parameter);
  });
local = 2;
thread1.Start(local-1);
```

- Cloning (copying) variables

```
int local = 1;
int copy = local;
Thread thread1 = new Thread( () => {
  Console.WriteLine("Copy {0}.", copy);
  });
local = 2;
thread1.Start();
```


Asynchronous Exceptions

- What happens when a running thread throws an unhandled exception?

```
static void Main() {
  try {
    new Thread(() => {
      Thread.Sleep(500);
      throw new ApplicationException("Async. Exception.");
    }).Start();
  }
  catch (Exception) {
    Console.WriteLine("The exception is handled. ");
  }
  Thread.Sleep(1000);
  Console.WriteLine("End of execution.");
}
```

The exception is not handled and the program aborts!

The try/catch should have been written in the asynchronous code

This catch is useless

- threads/asynchronous.exceptions

Runtime Performance Improvement

- The following data were obtained executing the vector modulus program in a **quad core** computer with 8GBytes RAM
- A vector of 100,000 random elements in [-10,10] was used
  - Execution time with 1 worker thread: 30 ms
  - Execution time with 4 worker threads: 10 ms
- Is execution time reduced whenever the number of threads is increased?

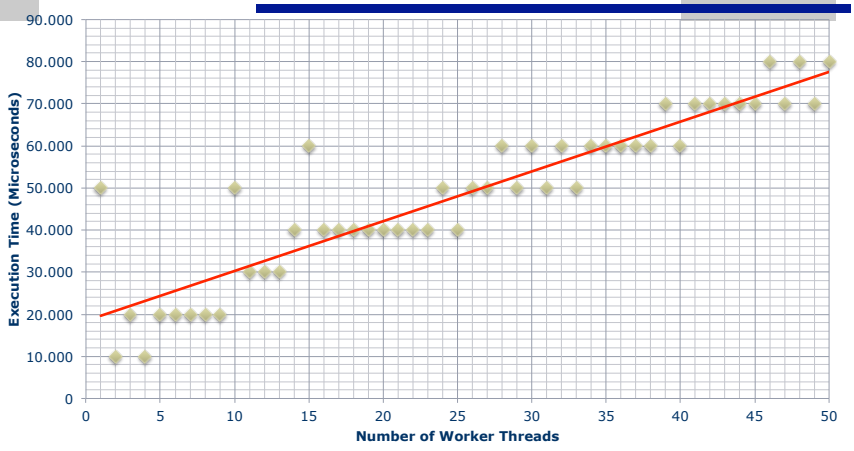
Context Switch

- A task (thread or process) **context** is the data that should be saved to allow task interruption and its future continuation
- A **context switch** is the computing process of storing and restoring the context of a task so that execution can be resumed from the same point at a later time
- This process allows the **concurrent** execution of multiple tasks in a single processor

Context Switch

- Context switching requires
  - **CPU time** to store and restore contexts of different tasks
  - **Additional memory** to store the different contexts
- Therefore, using a number of tasks much higher than the number of processors (cores) may imply **lower runtime performance** of the application
- We have executed the vector modulus application with 1-50 worker threads, and these were the results:
- threads/context.switching

Context Switch

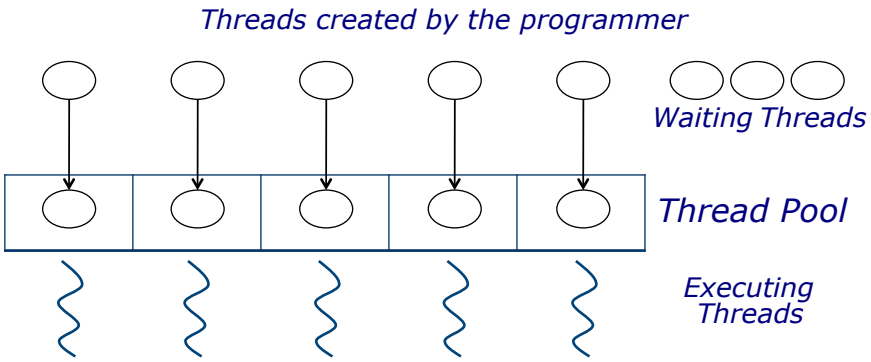


- With **2 and 4 threads** the best performance is obtained
- With **34 threads**, performance is lower than the sequential approach
- With **more than 9 threads**, there is a linear performance degradation as the number of threads grows (red line)

Thread Pooling

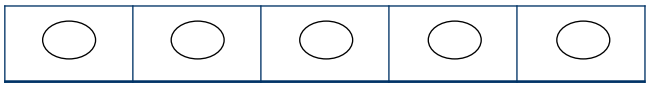
- **Context switching** may involve a computational cost in concurrent applications
- **Thread construction and destruction** also involves a performance and memory cost
- Therefore
  1. **The maximum number of threads** in an application should be related to the number of processors and memory resources
  2. Threads should be **reused**
- To achieve these objectives, the CLI provides a **thread pooling** mechanism to optimize the number of concurrent threads in a particular CPU
  - For example, the CLR 2.0 allows at most 25 concurrent threads per processor (core)
- This technique is also implemented by DBMSs (connection pooling) and Web servers

Thread Pooling



Thread Pooling

- If the process has not used any thread in a while…



Thread Pool

- The thread pool manager deletes the threads in the pool to save memory

Foreground & Background Threads

- The threads we have created with the `Thread` class so far are **foreground** threads:

  the application does not terminate until the termination of all its foreground threads
- A **background** thread (also called ***daemon***) is automatically killed when no foreground threads are being executed (e.g., monitoring services garbage collector, notification window)
  - They are commonly service providers
  - Do not confuse it with secondary or worker threads
- threads/daemon

Foreground & Background Threads

```
static void Main() {
  Thread background = new Thread(() => {
    int seconds = 0;
    while (true) {
      Thread.Sleep(1000);
      Console.WriteLine("\t\t\t\t\t\t\t\t{0} seconds.",
                        ++seconds);
    }  });
  background.IsBackground = true; // it is a daemon thread
  background.Start();
  Thread foreground = new Thread(() => {
    for (int i = 0; i < 100; i++) {
      Console.WriteLine("Iteration {0}.", i + 1);
      Thread.Sleep(100);
    }  });
  foreground.Start();
}
```


Thread Synchronization

- Sometimes, different threads have to **cooperate** one with another to achieve a common objective
- Since the order of execution is not deterministic, **threads need to be synchronized** to avoid race conditions
- A very basic synchronization mechanism we have already used is **`Thread.Join`**
  - *A thread is blocked until another one terminates*
- However, the most typical thread synchronization scenario is the concurrent access to **shared resources**
  - A **shared resource** could be a physical (printer) or logical (file) device, a data structure, an object or even a variable
- Avoiding the simultaneous use of a common resource is called **mutual exclusion**

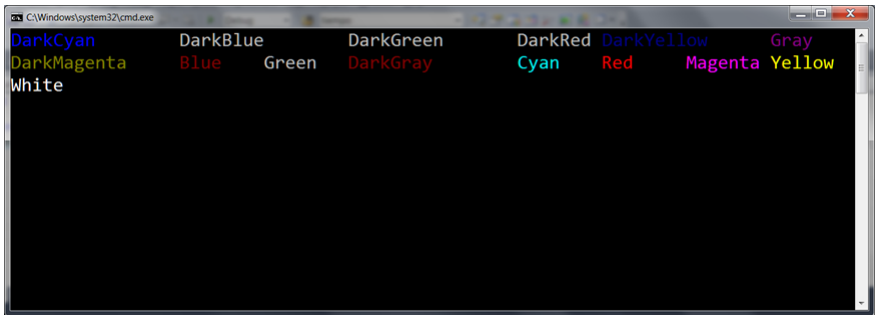
Example

- What would happen if the **Show** method is run by multiple concurrent threads?

```
public class Color {
  private ConsoleColor color;
  public Color(ConsoleColor color) {
    this.color = color;
  }
  virtual public void Show() {
    ConsoleColor previousColor = Console.ForegroundColor;
    Console.ForegroundColor = this.color;
    Console.Write("{0}\t", this.color);
    Console.ForegroundColor = previousColor;
  }
}
```


Example

- Example execution:
- synchronization/non.synchronized.colors



Shared Resource

- In the previous code, the **shared resource** is the standard output of the **console**
- The fact that mutual exclusion is not used makes the following instructions:

```
ConsoleColor previousColor = Console.ForegroundColor;
Console.ForegroundColor = this.color;
Console.Write("{0}\t", this.color);
Console.ForegroundColor = previousColor;
```

  Not to be executed **atomically**
- A **critical section** is a piece of **code** that **accesses** a **shared resource**, which must not be concurrently accessed by more than one thread of execution
  - **Thread synchronization** must be used to achieve **mutual exclusion**

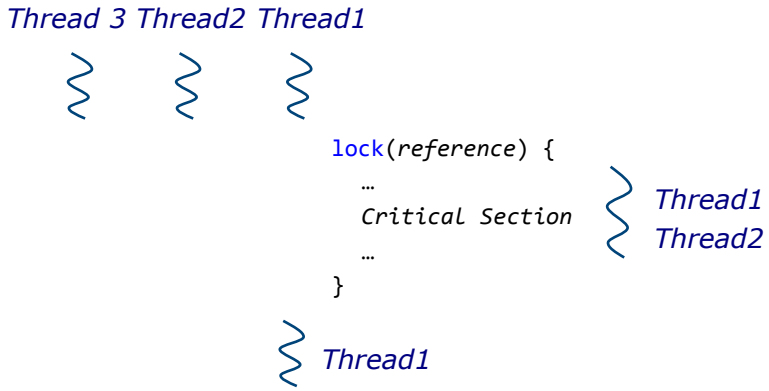
Lock

- The **`lock`** keyword is the main mechanism in C# to perform thread synchronization
- It is used to ensure that **no more than one thread** executes a critical section ⇒ **mutual exclusion**
- **`lock`** requires specifying an object (reference) as a parameter

```
lock(reference) {
  critical section
}
```

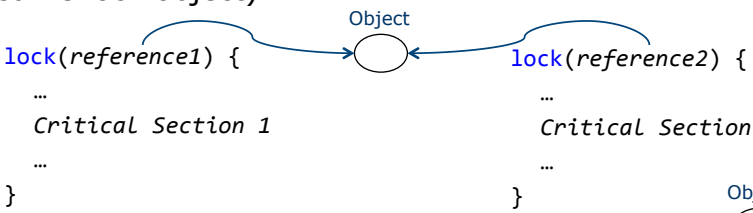
- The object models a **padlock**: 1) a thread locks the object 2) executes the critical section 3) releases the lock
- If another thread executes a lock statement over a locked object, it will **wait** until the object is released

Lock

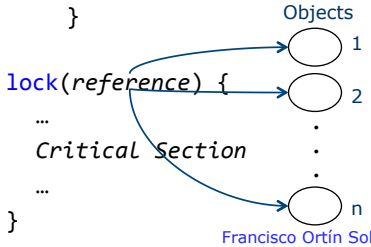


Lock

- **Notice**: Locking is performed considering the dynamic object (not the static piece of code)
- Example 1: Two pieces of mutually exclusive code (the same lock object)



- Example 2: The same piece of code could be executed concurrently by *n* different threads

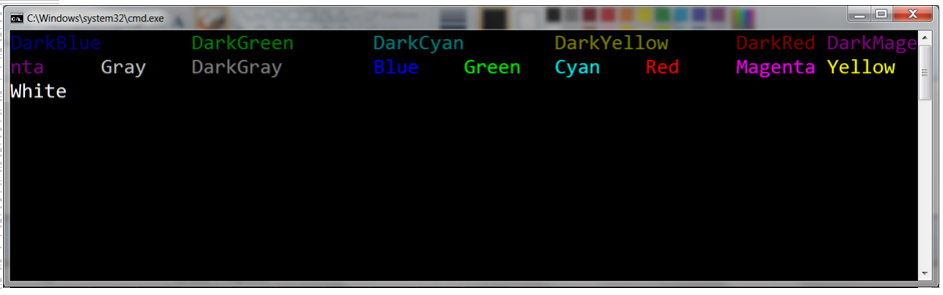


Example

- Which object can be used as a padlock to obtain mutual exclusion of the standard output?

```
public class SynchronizedColor: Color {
  public SynchronizedColor(ConsoleColor color)
    : base(color) {
  }
  override public void Show() {
    lock (Console.Out) {
      base.Show();
    }
  }
}
```


Example



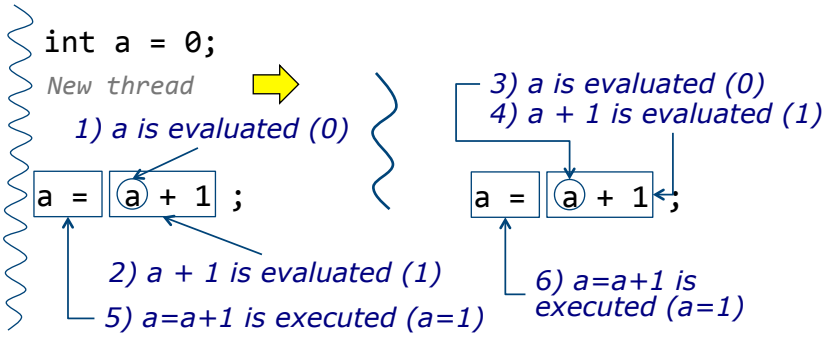
- synchronization/synchronized.colors

Assignments

- We have seen how the modification of shared resources requires synchronization
- Is it also necessary in **variable** and **field assignments**? Yes, it is also necessary
- Not all the assignments in .NET are atomic
  - 32 bit assignments are always atomic
  - 64 bit assignments (`long`, `ulong`, `double`, `decimal`) are not atomic in 32 OSs
  - `+= -= *= /= %= &= |= ^= <<= >>=` operators are not atomic
  - `++ --` operators are not atomic
- Question: Do pure functional languages have this limitation?

Assignments

- Let's analyze the concurrent execution of the `a = a + 1` expression



- After this execution, the value of `a` is 1 instead of 2!

Assignments

- Therefore, **concurrent assignments** of the same shared variable **should be synchronized**
- An alternative is using **`lock`**
  - This approach blocks running threads (mutual exclusion)
- Another approach is using the services of the **`Interlocked`** class (**`Sytem.Threading`**)
  - It provides atomic modification of variables
  - It provides better runtime performance than **`lock`**
  - The most common methods of **`Interlocked`** are **`Increment`**, **`Decrement`** and **`Exchange`**

Example

- What is shown in the console?
- synchronization/interlocked

```
static long value = 100000000;
static void Main() {
  const int numberOfThreads = 10000;
  int iterations = (int)value / numberOfThreads;
  Thread[] threads = new Thread[numberOfThreads];
  for (int i = 0; i < numberOfThreads; i++)
      threads[i] = new Thread(() => {
      for (int j = 0; j < iterations; j++)
        value = value - 1;
      });
  foreach (Thread thread in threads) thread.Start();
  foreach (Thread thread in threads) thread.Join();
  Console.WriteLine(value);
}
```

Example

- And now?
- synchronization/interlocked

```
static long value = 100000000;
static void Main() {
  const int numberOfThreads = 10000;
  int iterations = (int)value / numberOfThreads;
  Thread[] threads = new Thread[numberOfThreads];
  for (int i = 0; i < numberOfThreads; i++)
      threads[i] = new Thread(() => {
      for (int j = 0; j < iterations; j++)
        Interlocked.Decrement(ref value);
      });
  foreach (Thread thread in threads) thread.Start();
  foreach (Thread thread in threads) thread.Join();
  Console.WriteLine(value);
}
```


Mutex and Semaphore

- We have seen synchronization of threads so far
- Process synchronization can be achieved by means of **mutexes** and **semaphores**
- A **mutex** (*mut*ual *ex*clusion) is a process (and thread) synchronization mechanism
  - Is quite similar to `lock`
  - Its execution time is approximately 50 times higher than `lock`
- **Semaphores**, additionally, allow the access of ***n*** concurrent processes (or threads) to the critical section
  - They are used as a record of how many units of a particular resource are available

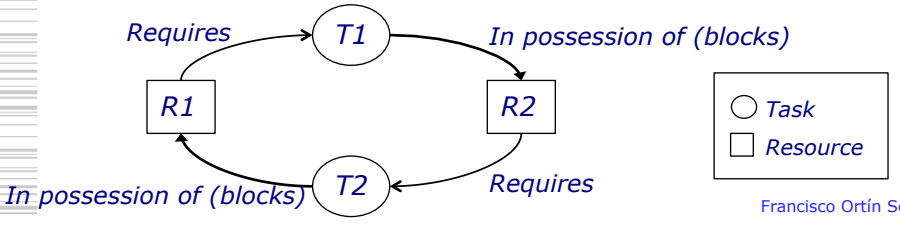
Mutex

```
static void Main() {  Only one mutex with this name is allowed in this CPU
  using (var mutex = new Mutex(false, " MutexDemo")) {
   // Waits zero milliseconds, returning whether the mutex is
   // free. Without parameters, waits until the mutex is free.
   if (!mutex.WaitOne(0)) {
      Console.WriteLine(" Another instance of the application
                          is being executed.");
      Console.WriteLine(" The program won't be executed.");
      return; // Program terminated
    }
  ExecuteProgram();
  } }
static void ExecuteProgram() {
  Console.WriteLine(" This is supposed to be the program
                execution.\nPress enter to exit");
Console.ReadLine();
}
```

- synchronization/mutex

Deadlock

- A **deadlock** is a situation in which two or more competing tasks are each waiting for the other to perform an action, and thus neither ever does
- Thereby the tasks are permanently blocked
- A common scenario is the concurrent access to a shared resource
  - All the tasks are waiting for a shared resource, possessed by another task, to be released



Example

```
public class Account {
   private decimal balance;
   public bool Withdraw(decimal amount) {
    if (this.balance < amount)
      return false;
    balance -= amount;
    return true;
  }
  public void Credit(decimal amount) { balance += amount; }
  public bool Transfer(Acount destination, decimal amount) {
    lock (this) {
      lock (destination) {
        if (this.Widthdraw(amount)) {
          destination.Credit(amount);
          return true;
        }
        else return false;
      }    }   }
```

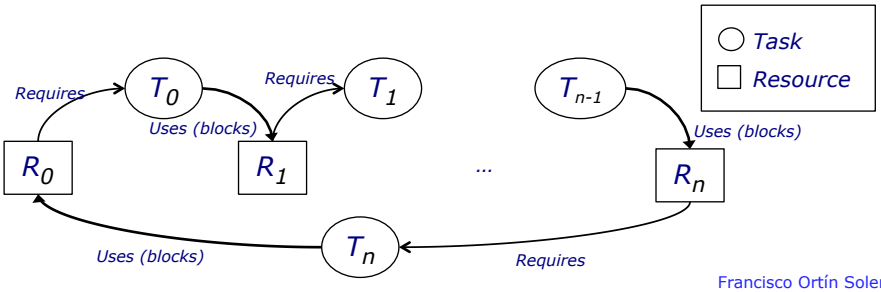
What happens when the following two statements are executed concurrently?

```
accountA.transfer(accountB, amount);
accountB.transfer(accountA, amount);
```

- deadlock/

Circular Waiting

- This condition is produced when given $T_0 \ldots T_n$ tasks, T_0 holds the R_1 resource and is waiting for the R_0 resource to be released by the T_n task… and T_n is waiting for the R_n resource held by the T_{n-1} task and holds R_0
- This condition (**circular waiting**) causes the deadlock of all the $T_0 \ldots T_n$ tasks



Avoiding Deadlock

- Deadlocks can be avoided by **preventing the circular wait condition** from occurring
  - If the wait condition is avoided, no deadlock occurs
- To do it, all the resources used by each task should be known statically (before execution)
  - **Resource assignment** should be done **denying** or postponing the request if it puts the system in an *unsafe* state (a deadlock could occur)
  - Dijkstra banker's algorithm
- However, **it is not always feasible** to know the resources used by a task **statically** (before execution)
  - A general algorithm to decide whether a deadlock may occur at program execution cannot exist ⇒ It its an **undecidable problem**

Avoiding Deadlock

```
public class Account {
   private decimal balance;
   public bool Withdraw(decimal amount) {
    if (this.balance < amount)
      return false;
    balance -= amount;
    return true;
  }
  public void Credit(decimal amount) { balance += amount; }
  public bool Transfer(Acount destination, decimal amount) {
    lock (this) {
      lock (destination) {
        if (this.Widthdraw(amount)) {
          destination.Credit(amount);
          return true;
        }
        else return false;
      }    }   }
```

- deadlock/

Avoiding Deadlock

```
public bool Widthdraw(decimal amount) {
  lock(this) {
    if (this.balance < amount)
      return false;
    balance -= amount;
    return true;
  } }
public void Credit(decimal amount) {
  lock(this) {
    balance += amount;
  } }
public bool Transfer(Account destination, decimal account) {
  if (this.Widthdraw(amount)) {
    destination.Credit(amount);
    return true;
  }
  else return false;
}
```

Could this piece of code be out of the lock statement?

- deadlock/

Seminar 6

- Seminar 6
  - Deadlock

Thread Safety

- A program, method, function or class is **thread safe** when it guarantees safe execution by multiple concurrent threads (it has no race condition)
  - This concept is also applicable to data structures
- Programming using thread safe elements does not necessarily implies thread safety
  - For instance, being `Stack` a thread-safe class, the following code is not thread safe. Why?

```
if(!stack.IsEmpty())
  stack.Pop().Message();
```

Another thread could have made the stack to be empty

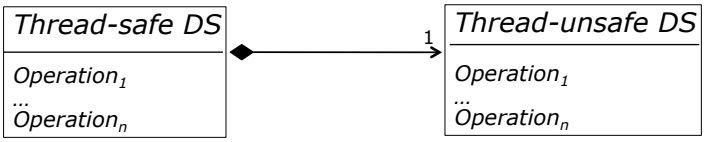
Thread-Safe Data Structures

- Most general purpose types (classes) are **not thread safe** because of **runtime performance**
- Therefore, either
  1. Special types (classes) can be used (e.g., `System.Collections.Concurrent`)
  2. Or thread safe code can be programmed using thread unsafe classes plus synchronization mechanisms

```
static void ThreadSafeAddAndShow(List list) {
  lock (list) list.Add("Item #" + list.Count);
  string[] items;
  lock (list) items = list.ToArray();
  lock (Console.Out)
    foreach (string s in items) Console.WriteLine(s);
}
```

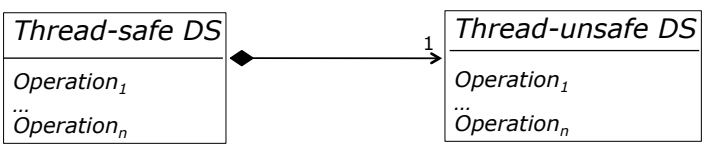
Implementing Thread-Safe DSs

- A basic implementation of thread-safe data structures is using the ***Decorator*** design pattern
  1. Taking a thread-unsafe data structure
  2. Create another class with the same interface
  3. Reuse the thread-unsafe data structure using composition (private attribute)
  4. Blocking (`lock`) the access to the thread-unsafe data structure in every method, using a private object as a padlock



Questions

- In the previous slide we mentioned, to implement a thread safe DS,
  - Blocking (`lock`) the access to the thread-unsafe data structure in every method, **using a private object** as a padlock
- Which object would be used in our example?
- Why is it better than using `this`?



Thread-Safe Data Structures

Implementing Thread-Safe DSs

- The previous technique is **straightforward**, but very **inefficient**
- In general, **read-only** concurrent operations do not require mutual exclusion of other **read-only** operations
  - Therefore, using a synchronization mechanism for all the operations may involve a runtime performance penalty
- **Write operations**, on the other hand, must be synchronized with both read and write operations
- The `lock` mechanism provided by C# does not fulfill this requirement
- Thus, the `ReaderWriterLockSlim` class has been added to .NET Framework 4 (`Enter{Read,Write}Lock` and `Exit{Read,Write}Lock`)
  - We will not go into the details

Parallelization by means of TPL

Parallelism

- Recall that parallelism is a particular scenario of concurrency, where **tasks are executed in parallel** (simultaneously)
  - **Parallelism** emphasizes the division of large problems into smaller ones (data, code, tasks…)
  - **Concurrency** emphasizes iteration between tasks

Parallelism Scenarios

- Recall, these are the most common parallelization scenarios
  - **Data parallelism**: performing the same task that computes pieces of the same data
  - **Task parallelism**: independent tasks are executed concurrently
  - **Pipeline**: A hybrid model of the two previous methods

Factors to be considered

- So far, we have considered the **creation**, **utilization** and **synchronization** of threads
- We have not considered
  - Whether increasing the number of created threads improves the global runtime performance of the application
  - How many threads should be created considering the thread pool
  - The number of existing processors (cores)

Parallelization by means of TPL

Task Parallel Library

- The **Task Parallel Library** (TPL) transparently manages these factors
  - It is part of the .NET Framework 4.0
  - `System.Threading.Tasks` namespace
- Provides the following benefits
  - **Simplifies** application parallelization
  - **Dynamically scales** the number of threads, considering the number of processors (cores)
  - Creates the appropriate number of threads, considering the **thread pool**
  - Provides services for **data parallelism**
  - Provides services for **task parallelism**
- **TPL** supports a much more **declarative** mechanism to implement parallel applications

Parallelization by means of TPL

Data Parallelism with TPL

- **Data parallelism** in TPL is mainly supported by the `ForEach` and `For` methods of the `System.Threading.Tasks.Parallel` class
  - Both receive the task to be executed as an `Action`
  - `ForEach` potentially creates one thread for each element in an `IEnumerable`
  - `For` executes a for loop in which iterations may potentially run in parallel
  - All the created threads are joined, guaranteeing their termination before the following instruction is executed

Parallelization by means of TPL

Parallel.ForEach

```
string[] fileNames=Directory.GetFiles(@"..\..\..\pics", "*.jpg");
string newDirectory = @"..\..\..\pics\rotated";
Directory.CreateDirectory(newDirectory);

Parallel.ForEach(fileNames, file => {
 string fileName = Path.GetFileName(file);
 using (Bitmap bitmap = new Bitmap(file)) {
  Console.WriteLine("Proccessing {0} with thread {1}.",
        fileName, Thread.CurrentThread.ManagedThreadId);
  bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
  bitmap.Save(Path.Combine(newDirectory, fileName));
 }
});
// Notice, TPL waits for task termination (synchronization)
```

- tpl/data.parallelism

Francisco Ortín Soler

Parallelization by means of TPL

Comparison with Sequential Impl.

- We have compared the previous TPL example with its sequential version
  - i7 2.67GHz, 8GB and Windows 7 SP1

	Sequential	TPL
Threads	1	6
Execution time (ms)	3,423	1,871
Benefit		82.95%

- Increasing the number of files, TPL does not increase the number of threads!
  - It dynamically considers whether runtime performance could be improved
- Quite similar to the sequential version
  - No need to check for thread termination

Francisco Ortín Soler

Parallelization by means of TPL

Task Parallelism with TPL

- **Task parallelism** in TPL is mainly supported by the **`Invoke`** method of the **`Parallel`** class
  - Receives a variable argument list of **`Action`**s
  - It potentially creates one thread for each **`Action`** passed as an argument
  - All the created threads are joined, guaranteeing their termination before the following instruction is executed

Parallelization by means of TPL

Parallel.Invoke

```
String text = ReadTextFile(@"..\..\..\clarin.txt");
string[] words = DivideIntoWords(text);
Parallel.Invoke(
  () => punctuationMarks = NumberOfPunctuationMarks(text),
  () => longestWords = LongestWords(words),
  () => shortestWords = ShortestWords(words),
  () => wordsAppearMoreTimes = WordsAppearMoreTimes(
                           words, out greatestOccurrence),
  () => wordsAppearFewerTimes = WordsAppearFewerTimes(
                           words, out lowestOccurrence)
);
```

- tpl/task.parallelism

Comparison with Sequential Impl.

- We have compared the previous TPL example with its sequential version
  - i7 2.67GHz, 8GB and Windows 7 SP1

	Sequential	TPL
Threads	1	5
Execution time (ms)	1,684	1,381
Benefit		21.94%

- Performance increases as the file size increases
  - These data are for a 2.33MB file
- In this case, the benefit is only 22%. Why?
- Therefore, it should be analyzed when parallelization really implies a runtime performance benefit!
  - Parallelization not always involves a significant benefit

Seminar 7

- Seminar 7
  - Parallelization with TPL

Functional Paradigm

- We have seen how multi-threaded applications should be implemented with **thread-safe** abstractions
- This condition is not fulfilled when shared **mutable** data structures are concurrently modified (write operations)
  - Computation depends on the states of abstractions (resources, input/output, global variables, mutable objects…) being modified throughout application execution
  - Synchronization is therefore required
- In the pure **functional paradigm**, state modification does not occur due to **referential transparency**
  - This paradigm tremendously facilitates the parallelization of algorithms
  - Recall, even if you are in an imperative language, **follow the functional principles to facilitate parallelization**

Functional Programming in .NET

- Due to
  - Advantages of the **functional paradigm** in **concurrent programming**
  - The **increasing demand of concurrent applications** (multi-core processors)
- The .NET platform makes extensive use of this paradigm for the development of concurrent programs (delegates and lambda functions)
  - Asynchronous message passing
  - Notification of thread termination (callbacks)
  - Explicit thread creation
  - Task Parallel Library (`For`, `Foreach`, `Invoke`...)
  - And, especially, **Parallel LINQ** (PLINQ)

Parallelization & Functional Programming

Parallel LinQ

- Parallel Linq (PLINQ) is a parallel implementation of *LINQ to Objects*
  - LINQ to Objects are queries against memory collections such as `IEnumerable` or arrays (databases and XML APIs are not supported)
- The use of the functional paradigm provides
  - **Data Parallelism**
  - In a **declarative** way
  - **Transparent** to the number of cores, dynamically choosing the number of threads
  - Being **conservative**, analyzing the overall query structure at runtime
    - If the query is likely to yield speedups by parallelization, PLINQ partitions the source sequence and run tasks concurrently
    - Otherwise, PLINQ runs the query sequentially

Parallelization & Functional Programming

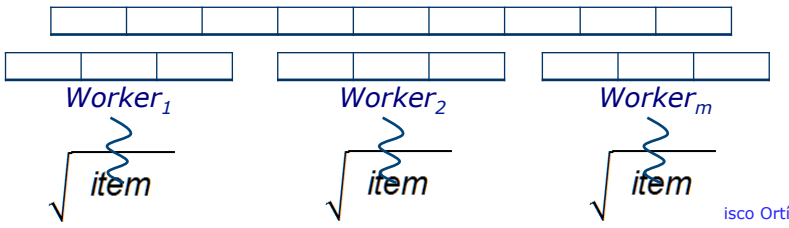
Parallel LinQ

- **Linq** compute data **sequentially**

```
vector.Select(item => Math.Sqrt(item))
```

- **PLinq**
  1. Partitions the data of the query
  2. Executes the query on each segment on separate worker threads in parallel
  3. Combines the results

```
vector.AsParallel().Select(item => Math.Sqrt(item));
```



Question

- Let's revisit the program that computes the modulus (magnitude) of a n-dimensional Euclidean vector

$$\overrightarrow{|v|} = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

- What is the Linq query that sequentially computes the vector modulus?
- What is the PLinq parallel version?

Performance Improvement

- Executing both the sequential (Linq) and parallel (Plinq) versions
  - In a quad-core computer with 8GBytes RAM
  - Using a vector of 20 million elements with random values between -10 and 10
- These were the execution times
  - Sequential: 1.775 milliseconds
  - Parallel: 1.467 milliseconds
  - Benefit: 21%
- Why do you think the performance benefit is *only* 21%? (or, when do you think a more significant benefit could be obtained?)
- Pling/

Amdahl's Law

- **Amdahl's law** (or argument) indicates the maximum expected improvement to an overall system when only part of the system is improved
  - In parallel computing, predicts the **theoretical** maximum speedup using multiple processors
- The speedup of the whole program is limited by the execution time of the sequential fraction of the program

  50% sequential, 50% parallel ⇒
  Theoretical maximum speedup is 2 (100%)

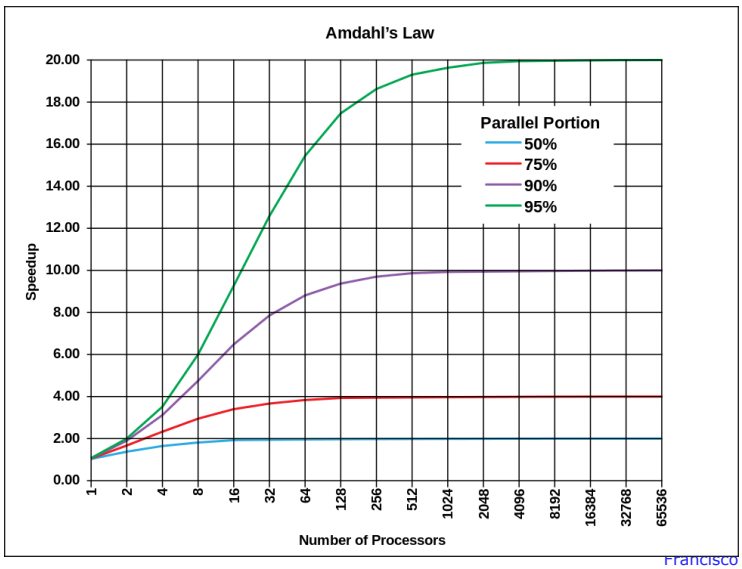
  25% sequential, 75% parallel ⇒
  Theoretical maximum speedup is 4 (300%)

  10% sequential, 90% parallel ⇒
  Theoretical maximum speedup is 10 (900%)
- The greater the execution time fraction of sequential code, the lower the speedup of the whole application (and vice versa)

Francisco Ortín Soler

Parallelization & Functional Programming

Amdahl's Law



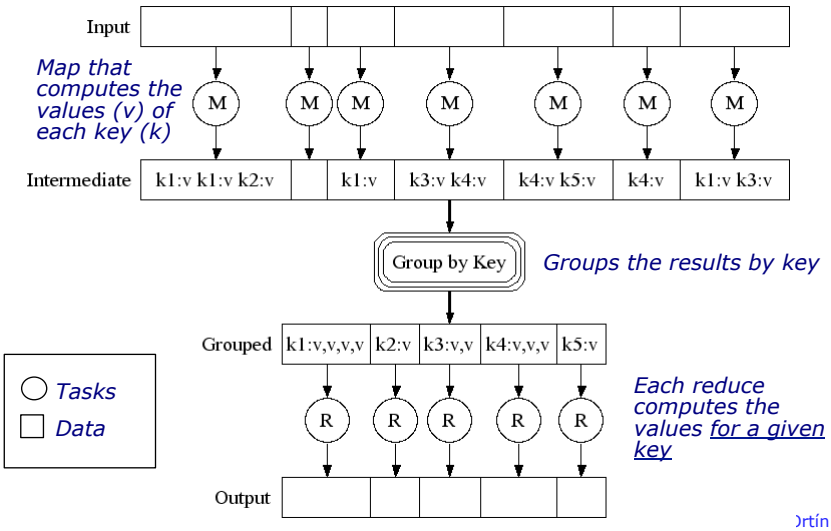
Francisco Ortín Soler

MapReduce

- **MapReduce** is a parallel (distributed) programming model for processing large data sets by means of data parallelism
- Two **higher-order** functions are used
  - `Map`: Processes all the elements in a list, generating another list
  - `Reduce` (`Fold`, `Aggregate`): Processes all the elements in a list, generating a single value
- MapReduce uses dictionaries instead of lists (the classic approach)
- Google has granted a patent on MapReduce (2004)
  - Google uses MapReduce for multiple purposes: regenerate Web page indexes, perform word counts…
- There are multiple implementations for different languages
  - Hadoop is a widespread example (Java)

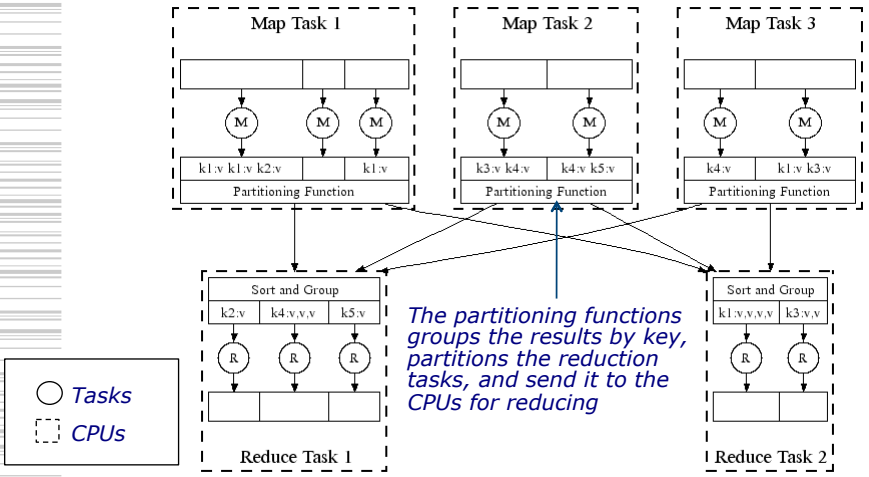
MapReduce, Concurrency

Classical example: Counting words in a document



Parallelization & Functional Programming

MapReduce, Parallelism



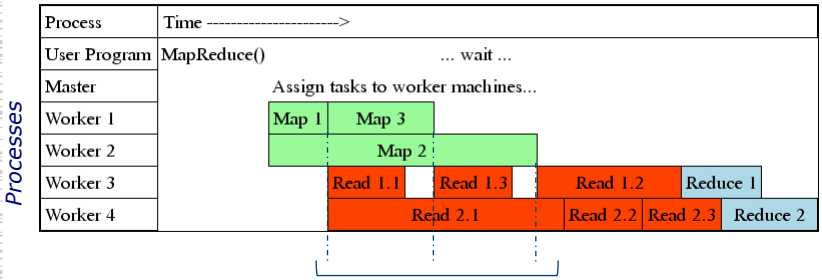
Parallelization & Functional Programming

MapReduce, Synchronization

***Pipeline** parallelism: Data is partitioned by the master, and each worker processes a different task with the data received*

***Read** tasks implement the partitioning functions*

***Reduce** tasks perform the reduction (reduce, aggregate) of the corresponding keys*



Read *worker#-2.map#*

Reads can be executed when the associated maps have terminated.

Computer Science
Engineering School

Software
Engineering

Programming Technologies and Paradigms

Foundations of Concurrent and Parallel Programming

Francisco Ortín Soler

University of Oviedo