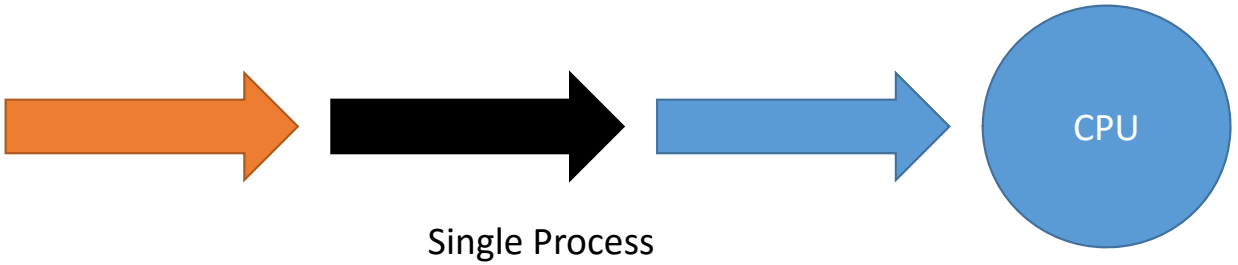
Parallel Programming

2016-2017

Single-Tasking vs Multitasking processors



Single Process

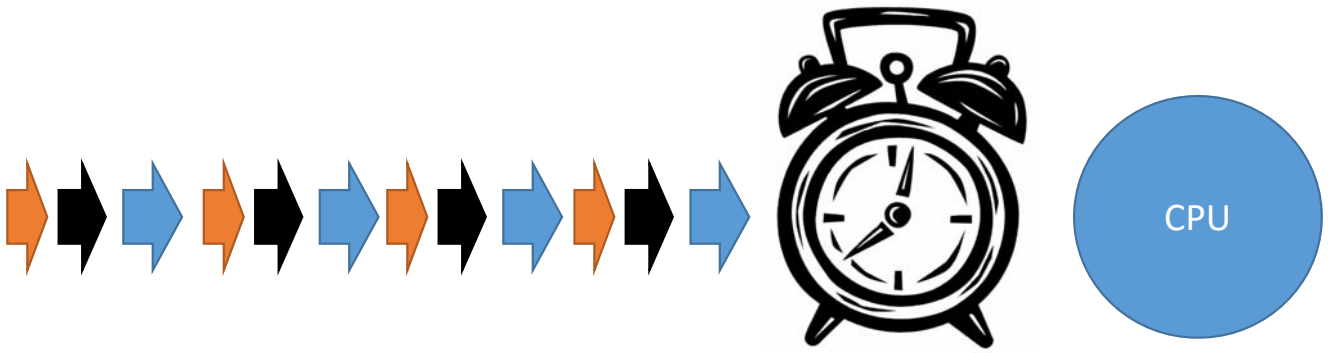
Multitasking

Single-Tasking Execution

- Initially , Computers were only used to accomplish limited tasks
- If the CPU is free, a process can occupy the CPU.
- All other processes should wait until the CPU is free again
- The end user may assume that a program is not responding
- A process might contain bug, and run in an infinite loop; the computer will hang

Multitasking

- It’s also known as Multithreading or concurrency
- The CPU is still capable of handling only one process at time
- But processes don’t run to completion , they rather run for some period , then leave the CPU for other processes and so on.



Multitasking

- Suppose we have 3 tasks (processes), for a specified Processor (call it P0) the first task(t1) needs 4 seconds to be executed, t2 needs 5 secs and t3 needs 6 secs
- What is the total time needed to execute the three tasks, given that the CPU will handle each task until it’s finished(no multitasking)
- Suppose we upgraded the CPU to enable multitasking, what would the total time be ?

Multitasking

- Thus, multitasking doesn't speed execution, theoretically ; the total time should be the same
- The main benefit is **Responsivness**
- All processes **seem** to be executing simultaneously to the user
- Practically: multitasking incur additional overhead due to the **Context Switching** , so multitasking might be a little slower to run, (depending on many factors)

Scheduling

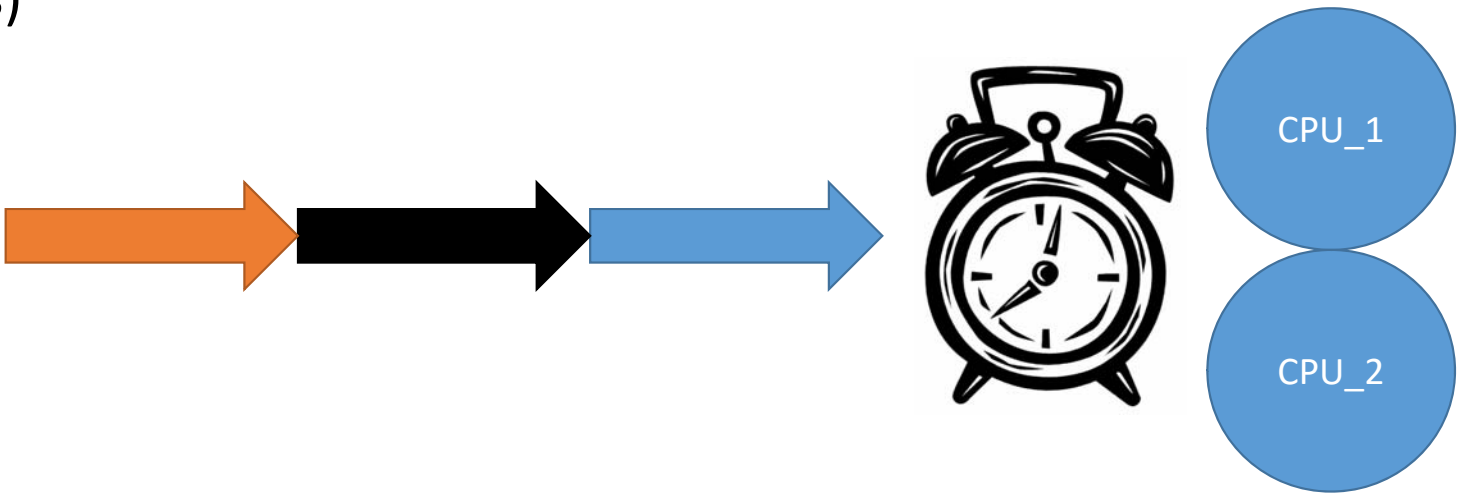
- Resource Allocation Problem:
- Determine which process should use the CPU and for how long
- The operating system is responsible for scheduling and it uses a Scheduling Algorithm to accomplish this goal
- Processes don’t have to wait until the execution queue is empty

Parallelism

- CPU evolved over years (in term of operating frequency)
- However, the increase in speed required an extensive cooling , and an unacceptable increase in the power consumption.
- Increasing the speed(operating frequency) practically stalled , and new alternatives were found
- Later, Computers were equipped with multiple CPUs to compensate for the limited frequency

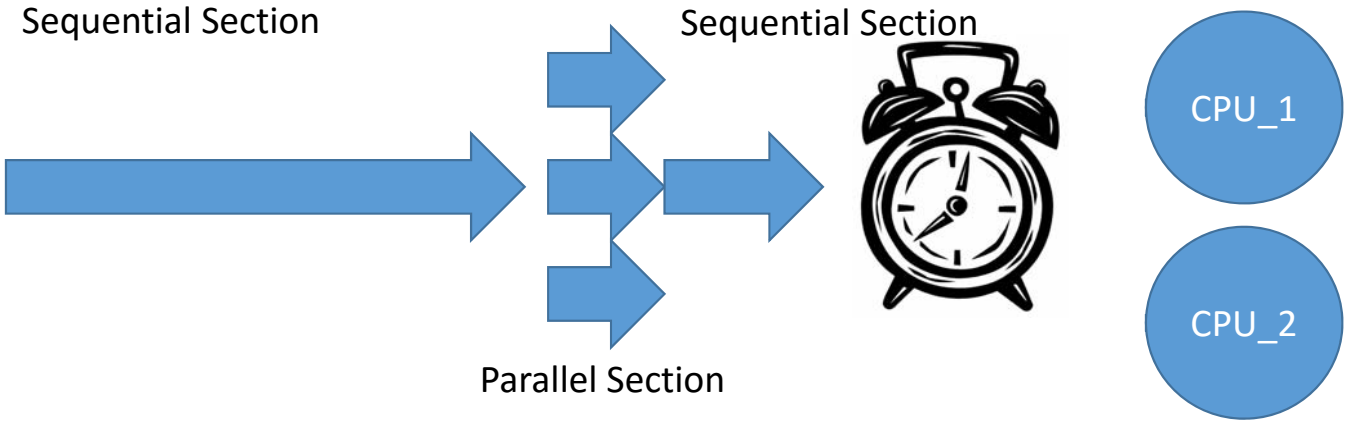
Parallelism

- Executing many processes, simultaneously
- Requires multiple CPUs
- This results in actual speedup (faster execution, less time to complete tasks)



Parallelism

- The operating system is still responsible for the scheduling mechanism
- Scheduling now , is concerned with which process to execute and using which processor
- Sequential Programs: are programs that the execution of their statements advance statement by statement(no statement shall be executed before other preceding statement)
- Parallel programs: are programs that contain segments (called threads) that can be executed in parallel



Parallelism

- Parallelism is not limited to only one Computer(multi-processor or multi-core computer)
- Parallelism is also extended to set of computers (cluster) that work collaboratively(in parallel) in order to solve a complex problem

Parallelism metrics

- Comparison between sequential and different parallel algorithms
- CPUs Utilization
- Scalability in different number of CPUs

Parallel Processing Architectures

- Shared Memory Architecture:

- All the Processes access the same memory, addressing the same variable(shared variable) should be handled with caution

It’s fast because processes can communicate through the memory

- Applicable in multi-processor computers

Parallel Processing Architectures

- Distributed Memory Architecture

- Each process has its own memory space (virtually or physically)

- No memory synchronization is needed

- Communication between processes should be handled explicitly and might need a connection link like a computer network

- Communication cost (time required for transferring messages) incur additional overhead

- Communication is handled via a Message Passing mechanism , the standard MPI(Message Passing Interface) is the most used

What to parallelize?

- Not all tasks need to be parallelized
- If the task contains only simple computations, the parallelism can make the performance worse
- Some tasks are related to the I/O subsystem (accessing Storage, external storage, network) and thus, they are bound to the speed of those subsystems not to the speed of the processor

I/O bound vs CPU bound

- Tasks are categorized to I/O bound and CPU bound
- I/O bound tasks rely heavily on the I/O subsystem and chipset bus.
- CPU bound tasks contain intensive Computations (in-memory searching , simulation, mathematical operations)

Section 1: Shared-Memory Parallelism

- This architecture is vitally important in every-day applications
- Modern applications running on PCs or Phones perform multiple tasks (accessing internet services, acquiring location, refreshing widgets,…)
- Maintaining responsiveness and performance is a determinant factor to these applications
- What we will learn in this section is related to the application architecture in Windows OS
- However, patterns and algorithmic principles are applicable in any Shared-memory environment

Section 1: Shared-Memory Parallelism

- For this section, we will consider .Net Parallel capabilities as an example, and C# as a development language
- The .Net offers a great head start for both the learning and the production modes
- Compiler Optimization simplifies the code needed and minimize the number of lines needed
- We will focus on Task Parallel Library(TPL) an architecture that is recommended by Microsoft to build parallel programs for all Windows-based devices

Section 1: Shared-Memory Parallelism

- Visual Studio is used as a Development Environment (VS2010+)
- Microsoft Developer Network(MSDN) is the primary reference for both implementation and design related information
- We will start by developing Console application (application that run in a command window) and then move to Windows Forms Application(a retired GUI development option)

Section 1: Shared-Memory Parallelism

Design Notes:

- The maximum number of threads that exploits parallelism equals the number of processors
- Shared data access should be handled with caution to avoid race conditions , synchronization should be enforced
- Shared data access should be minimized , as synchronization incurs additional overhead(that increases with the number of threads)
- The number of threads in an application, should not exceed a specific threshold.

Parallelism in .Net

- Threads are the principle and the most primitive elements in the parallel structure
- threads are defined in a simple manner (not requiring the extension of Thread class as in Java)
- Thread Class is available in Namespace “System.Threading”

Thread Class

- Represents a schedulable Thread which has:
  - Method to run (the actual execution)
  - Parameter for the run method(optional)
  - Priority
  - ….
- To pass the method to the thread, you can pass the name of the pre-declared method (a pointer to the method also called delegate)
- Alternatively , another simplified approach is used, which relies on Lambda Expressions in .Net

Thread Class

- Lambda Expression is a mechanism to pass a method as a parameter, without requiring the method to be declared previously (called anonymous method)
- ([List of parameters names]) =>{instructions}
- We want to pass a delegate(pointer) to the anonymous method
- the passed pointer should only point to the types of method that Thread Constructor expect

Thread Class

- Two constructors are available ,the first one takes a parameter-less run method , the other takes a method with one parameter of type Object
  1- void ____()
  2- void ____(Object b)
- Threads are mapped to Operating system threads. Using too many threads can hurt the performance, and manually handling threads is not Recommended

HelloWorld! Example

```
class SimpleThreadMethod
{
    1 reference
    static void ThreadMethod()//this code will be executed by firstThread
    {
        Console.WriteLine("Hello Parallel World!");
    }
    0 references
    static void Main(string[] args)//main application thread, started by the OS
    {
        Thread firstThread = new Thread(ThreadMethod);//constructor 1
        firstThread.Start();//Schedule the thread for execution
        firstThread.Join();//stops the caller thread until the "FirstThread" finishes execution
        Console.Read();
    }
}
```

HelloWorld! Example

```
class LambdaThread
{

    0 references
    static void Main(string[] args)
    {
        //constructor takes a Void ___()
        Thread firstThread = new Thread(() => { Console.WriteLine("Hello Parallel World!"); });
        firstThread.Start();
        firstThread.Join();
        Console.Read();
    }
}
```

HelloWorld! Example

```
class ParametrizedThread
{

    0 references
    static void Main(string[] args)
    {
        Thread firstThread = new Thread
            (//constructor also takes Void ____(Object b)
                (personName) =>
                {
                  Console.WriteLine("Hello Parallel World!, I am "+ personName.ToString());
                }
            );
        firstThread.Start("issa");//string is assigned to personName
        firstThread.Join();
        Console.Read();
    }
}
```