

Concurrency

Week 1

Introduction

How to pass this course:

- *Read the course manual*
- *Participate in class*
- *Finish whatever we do in class at home*
- *Study the topic of the last week EVERY week*
- *Read the material suggested*

Where to study?

Slides, class, book, reading proposed...

the book suggested is:

Principles of Concurrent and Distributed Programming, Second Edition

By M. Ben-Ari

Publisher: Addison-Wesley

Print ISBN-10: 0-321-31283-X

Print ISBN-13: 978-0-321-31283-9

Evaluation!

- **Final exam** —> multiple choice questions about the content and exercises.
- **Assignment** —> evaluated as **fail or pass**, delivery at the end of the course. The description and modality will be communicated soon (for now groups up to 2, further details will be provided).

NB: for main development language, we decided for **C#**

Contents

- Introduction
- Motivation:
 - Von Neumann Model
 - Overlapping Processing and Input/Output Times
- Multitasking
- Parallelism and Concurrency
- Abstraction:
 - Interleaving,
 - Atomic statements,
 - Correctness,
 - Fairness
- Challenges
 - Tracing and Testing Concurrent Processes

Introduction

Course

What is the course about?

- Question: What is your experience / impression about concurrency?



Simultaneous execution

- Question: Can you name programs / applications that run multiple activities simultaneously?

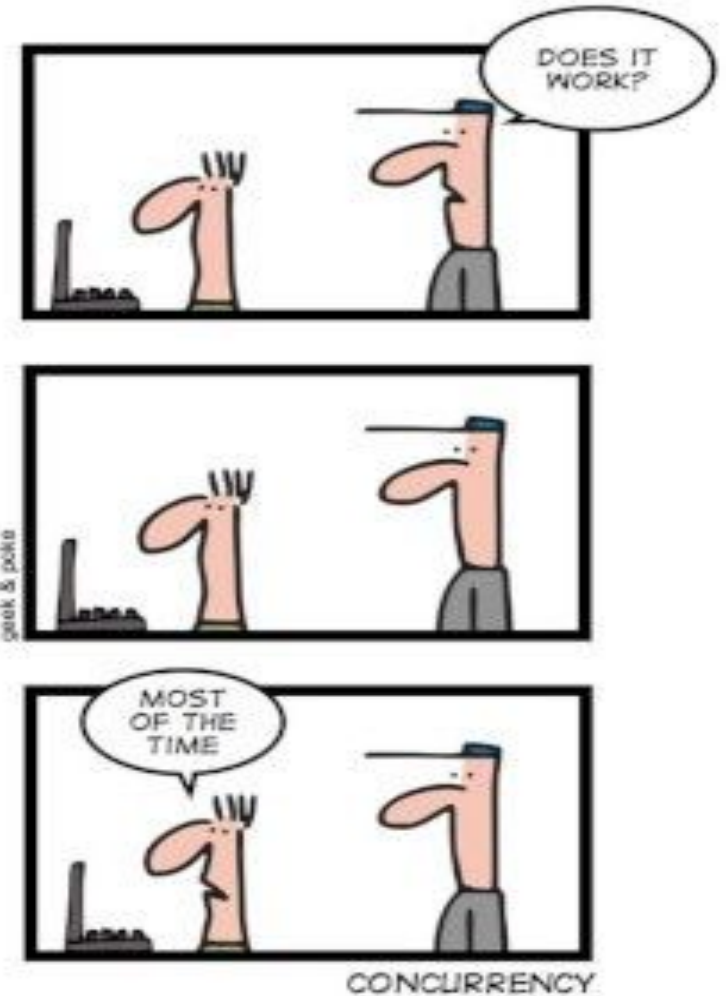


"When he needs to, Ed can really multitask!"

Why?

- Question: Why do you think this topic is important?
- Questions: What are the challenges to achieve concurrency?
- Question: What do we need to accomplish it?

SIMPLY EXPLAINED



Our course

Our course will be about:

- Fundamental concepts of concurrency / parallelism
- Analysing behaviour / execution concurrent programs
- Multiple methods to implement concepts

Program

Weekly plan:

1. Introduction
2. Processes
3. Threads & Concurrency models
4. Resources management
5. Deadlocks
6. Asynchronous vs. Synchronous
7. Review





Introduction

Concurrency / Parallelism.

Simultaneous execution

The whole topic is about running multiple tasks at the same time:

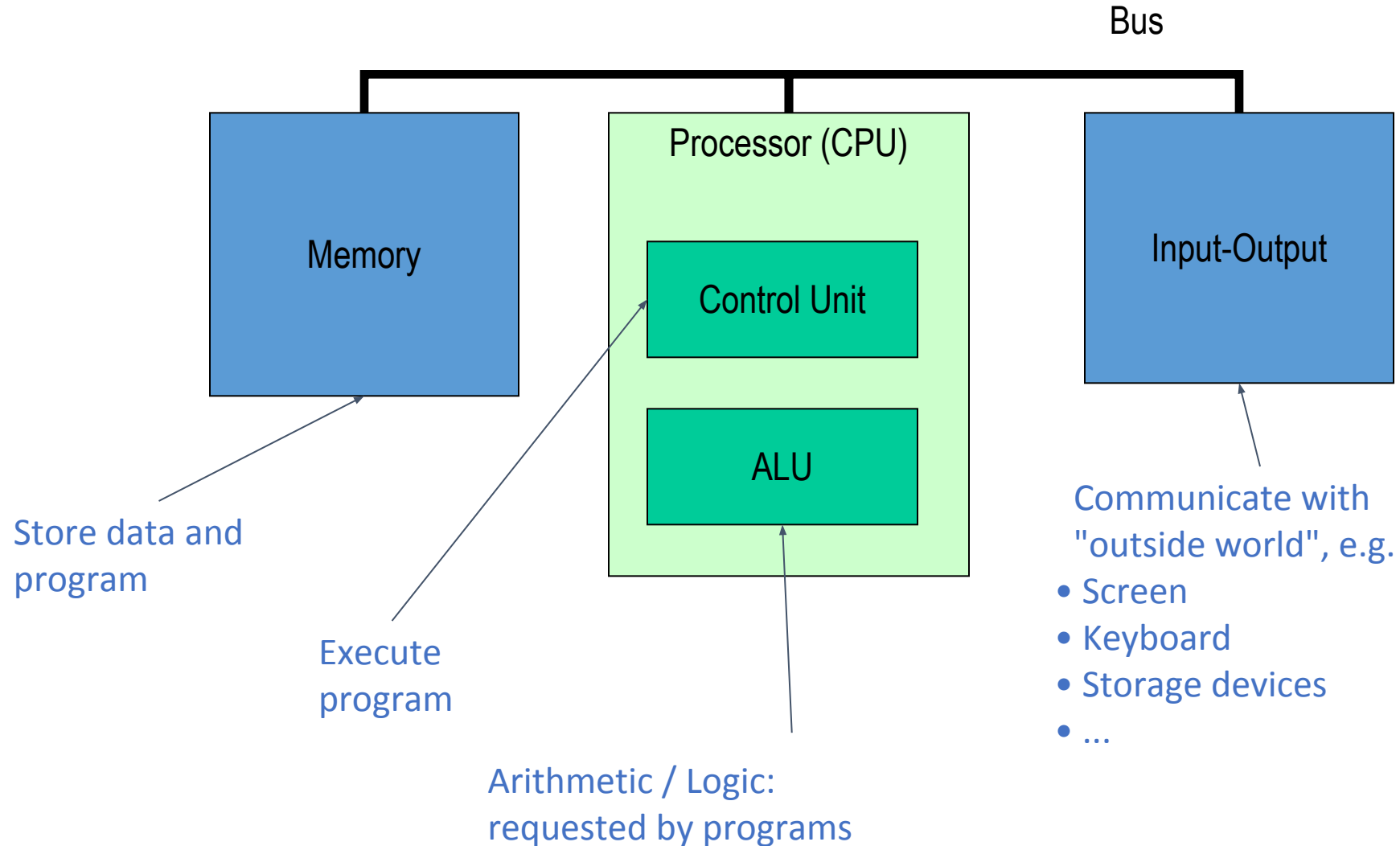
But first we need to review:

- What types of tasks we are talking about?
- How a task is executed?
- What does “at the same time” mean?

The Von Neumann Architecture (Refresh)

- John Von Neumann in 1945 proposed a model for designing and building computers, based on the following three characteristics:
 - 1) The computer consists of **four main sub-systems**:
 - Memory
 - ALU (Arithmetic/Logic Unit)
 - Control Unit
 - Input/Output System (I/O)
 - 2) Program is stored in **memory** during execution.
 - 3) Program instructions are executed **sequentially**.

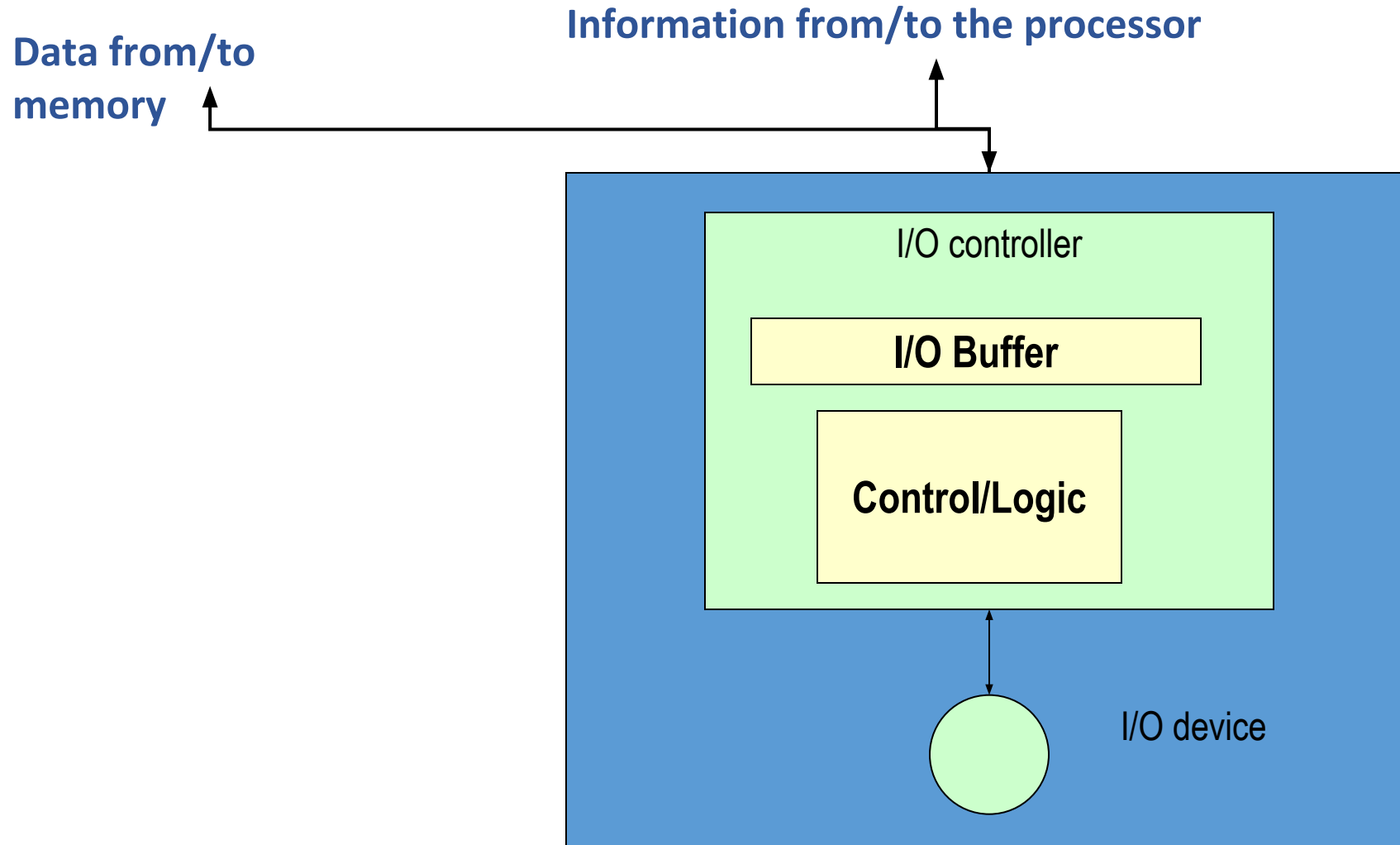
The Von Neumann Architecture



Input/Output Subsystem

- Handles devices that allow the computer system to:
 - Communicate and interact with the outside world
 - Screen, keyboard, printer, ...
 - Store information (mass-storage)
 - Hard-drives, floppies, CD, tapes, ...

Structure of the I/O Subsystem




The Control Unit

- Program is stored in memory
 - as machine language instructions, in binary
- The task of the **control unit** is to execute programs by repeatedly:
 - Fetch from memory the next instruction to be executed.
 - Decode it, that is, determine what is to be done.
 - Execute it by issuing the appropriate signals to the ALU, memory, and I/O subsystems.
 - Continues until the HALT instruction

Typical Machine Instructions

- Sample **machine language** instructions (in pseudo codes)
 - MOV : moves data from one location to another.
 - ADD : add operands
 - SUB : subtract operands
 - JMP : transfers program control flow to the indicated instruction

Example

- Assuming variable:
 - Variable A contains initial value 100, and variable B contains 150. The machine code for $C = A + B$ will be:
- Machine language
 - .DATA  define word (normally 2bytes)
 - A dw 100
 - B dw 150
 - C dw ?
 - MOV AX, A
 - ADD AX, B
 - MOV C, AX

CPU time versus I/O time

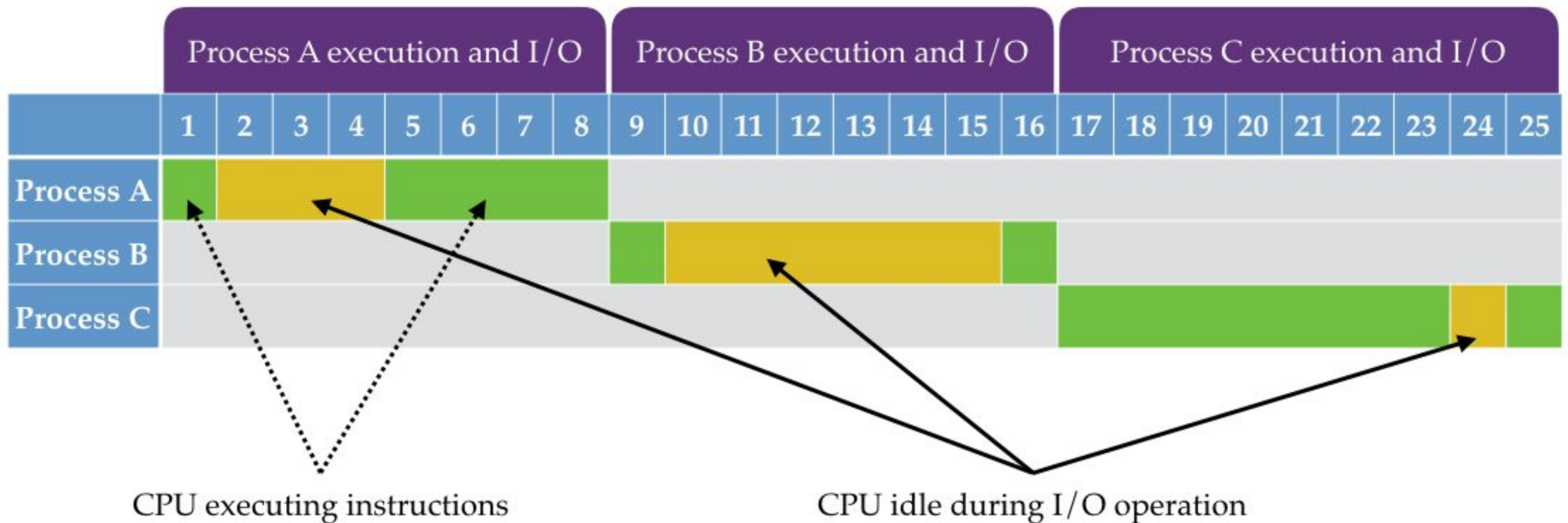
- When running a program, computers either perform calculations or access the peripheral devices.
- Therefore, the total required time to execute a program is:

Total time = CPU time + IO time

- Typical time to read a block from a hard disk is 20ms (0.2ms at SSD)
- Typical time to run an instruction 10 nsec (Nano seconds)
- A typical computer can execute up to 200 million instructions while a block is read from a hard disk.
- Conclusion: While I/O is executed, CPU will be idle.

Conclusion

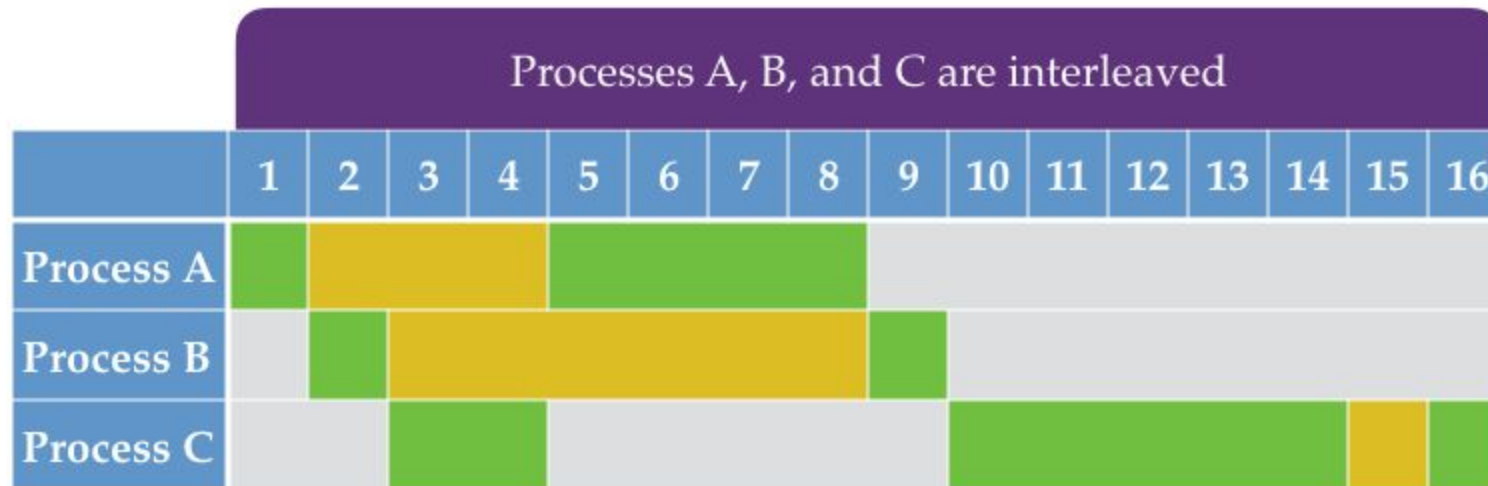
While I/O is executed, CPU will be idle.



Conclusion

While I/O is executed, CPU will be idle.

We want to be efficient, which leads us to ...



Multitasking

Multitasking

- To speed up computers we need to overlap CPU time and IO time
- *Solution:*
 - While a program is waiting for its IO operation to complete, other programs can use CPU
- This solution requires multiple programs to be loaded into the memory, hence it is named **multitasking**
- Multitasking overlaps IO time a program with CPU time of other programs

Multitasking with Multiple Processors

- If the computer has a single CPU, the programs should take turn in using it.
- When a program issues an I/O command, CPU is granted to the next waiting program

Concurrent vs Parallel

- Fast switching from a program to the next program can create the illusion that they are being executed at the **same time (concurrent)**.
 - Logically Simultaneous
- If the computer multiple CPUs, or a CPU with multiple cores, the programs can run in **parallel**.
 - Physically Simultaneous

Sharing Resources

- Sometimes different programs (or parts of the same program), may **share resources** (data for instance)
- If the data is **sequentially** accessible, the programs should take turn in accessing resource/data.
- Fast shared resource/data access can create **concurrent** access

Sharing Taking Turns



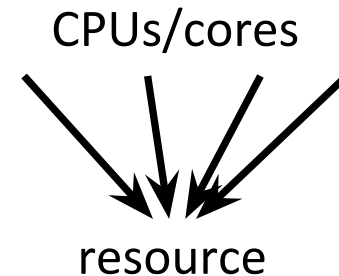
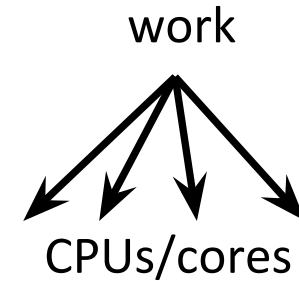
Sharing Resources

Examples:

- Several programs reading from a shared file. Is it safe?
- Several programs writing to the same file. Is it safe?
- Concurrent transactions (on bank accounts)!
- Ticket booking services.
- ...

Parallel vs. Concurrent

- **Parallel:** Using multiple processing resources (CPUs, cores) at once to solve a problem faster.
 - Example: A sorting algorithm that has several workers each sort part of the array.
- **Concurrent:** Multiple programs (or threads) accessing a shared resource at the same time.
 - Example: Many threads trying to make changes to the same data structure (a global list, map, etc.).



Concurrency

- Unlike parallelism, not always about running faster.
 - Even a single-CPU, single-core machine may want concurrency.
- Useful for:
 - *Application responsiveness*
 - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
 - *Processor utilization* (hide I/O latency)
 - If one thread (a kind of process) is waiting for I/O results, others have something else to do
 - *Failure isolation*
 - Convenient structure if we want to interleave multiple tasks and do not want an exception in one to stop the other

Abstraction

- We generally find it convenient (and in fact necessary) to limit our investigations to one or maybe two levels, and to "abstract away" the details.
- Example:
 - Your physician will listen to your heart, but he will not think about the molecules from which it is composed of.

Abstraction in Computer Science

- In computer science abstraction examples can be:
 - Using **APIs** we access resources through library function calls without knowing how they work.
 - Using high level **programming languages** we abstract away the details of the computer architecture.
 - Using **encapsulation** programmers hide the details of implementation from public specifications (consider classes in Object Oriented Programming)

Abstraction in Concurrency

- A *concurrent program* consists of a finite set of (sequential) processes.
- The processes are written using a finite set of *statements*.
- The execution of a concurrent program proceeds by executing a sequence of the statements obtained by *arbitrarily interleaving* the statements from the processes.

Abstraction in Concurrency: Example

- Assume:
 - process P has two sequential statement p1, and p2
 - process Q has two sequential statement q1, and q2
- Possible interleaving scenarios are:
 - p1->q1->p2->q2
 - p1->q1->q2->p2
 - p1->p2->q1->q2
 - q1->p1->q2->p2
 - q1->p1->p2->q2
 - q1->q2->p1->p2

p1->q2->p2->q1 is not possible. Why? Which other interleaving scenarios are not possible?

Atomic Statements

- Atomic statement model assumes that an atomic statement is **executed to completion without the possibility of interleaving** statements from another process.
- So an important property of atomic statements is that they **can't be divided** (based on a- 'not' + temnein 'to cut').

Correctness

- In sequential programs, rerunning a program with the **same input** will always give the **same result**, so it makes sense to "debug" a program.
- In a concurrent program, some scenarios may give the **correct** output while **others do not**.
- This implies that we cannot debug a concurrent program in the normal way, because each time we run the program, we will likely have a **different order of interleaving**.

Correctness

- Correctness of (non-terminating) concurrent programs is defined in terms of properties of computations, rather than a functional result.
- Two types of correctness properties are:
 1. **Safety properties** The property *must always* be true.
 - Example: *Always, a mouse cursor is displayed.*
 2. **Liveness properties** The property must *eventually* become true.
 - Example: *If you click on a mouse button, eventually the mouse cursor will change shape.*

Fairness

- Arbitrary interleaving assumes no order or speed of executing statements.
- However, it does not make sense to assume that statements from any specific process are *never* selected in the interleaving.
- A scenario of executing statements is *(weakly) fair* if a statement that is continually enabled *eventually* is selected to execute.

Traditional Challenges

- Useful set of abstractions that help to:
 - Manage concurrency
 - Manage synchronization among concurrent activities
 - Communicate information in useful way among concurrent activities
 - Do it all efficiently

Modern Challenges

- Methods and abstractions to help software engineers and application designers to:
 - Take advantage of inherent concurrency in modern application systems
 - Exploit multi-processor and multi-core architectures that are becoming ubiquitous
 - Do so with relative ease

Tracing and Testing

- Concurrent programs are hard to develop and test because of non-deterministic scheduling.
- Concurrency bugs, such as:
 - **Simultaneous access** to the same database records
 - **Atomicity** violations,
 - **Deadlocks**

are hard to detect and fix in concurrent programs.

Summary

- At the end of this course you will be able to:
 - *Describe* conceptual foundations.
 - *Describe* resource sharing.
 - *Describe* various models of process cooperation.
 - *Analyze* effective ways of structuring concurrent programs.
 - *Apply* their knowledge to design a concurrent program where different processes / threads cooperate with each other to solve a problem.
- Evaluation:
 - Final exam —> multiple choice questions about the content and exercises.
 - Assignment —> evaluated as fail or pass, delivery at the end of the course.

.net? Yes thank you!

Install **.net 2.1 LTS**:

<https://dotnet.microsoft.com/download/dotnet-core/2.1>

It is the same used in web development.

If you have it already installed, all ok!

Exercises and where to find them...

Please refer to google class for the exercises and other materials.

Whatever we do not finish in class, should be explored and finalised at home.

Keep in mind: we will also use

<https://github.com/afshinamighi/Concurrency> as source for code samples.