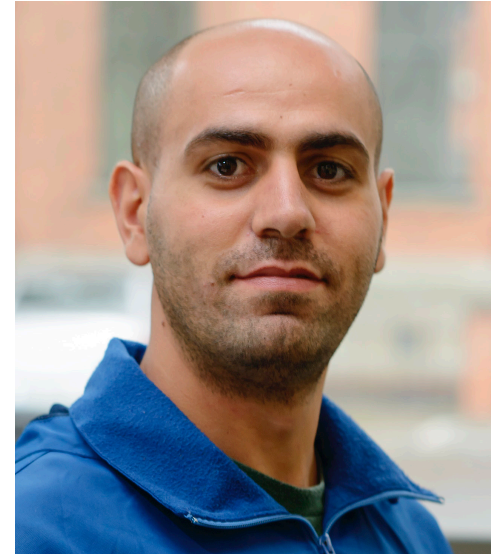
Ollscoil na Gaillimhe
University of Galway

Dr Takfarinas Saber
takfarinas.saber@nuigalway.ie

CT213 Computing Systems & Organisation

Week 1: Introduction

Who Am I



- Takfarinas Saber
  - Call me: **Tak**, Takfarinas, or Dr SABER
  - Email: takfarinas.saber@universityofgalway.ie
  - Office: IT425 (4th floor, IT Building)

- BSc, MSc and PhD Computer Science
- Research Areas:
  - Resource optimisation in Cloud
  - Engineering and optimisation of efficient software applications:
    - Distributed over multiple hardware and geographical locations
    - with various real-time user interactions
    - programs processing large quantities of data

Overview

- **Course schedule:**
  - Monday 2 – 3 pm, **AC215**
  - Monday 3 – 4 pm, **McMunn Theatre**

- **Course material:** ***http://www.nuigalway.blackboard.com***
  - I will publish slides of the lecture every week with enough information on the slides to make them as clear as possible.
    - But more info in class

- **Textbooks:**
  - Digital Design and Computer Architecture (Second Edition), David Harris & Sarah Harris, ISBN: 978-0-12-394424-5
  - Computer Systems Organization & Architecture, John D. Carpinelli, ISBN: 0-201-61253-4
  - Computer Architecture: A Quantitive Approch, John L Hennessy and David A. Patterson, ISBN: 1-55860-329-8

Labs

- **Runs from**: Week 3 (Week starting 19/09/2022)
- **Focus**: Playing with Operating System Command Line, Using Raspberry Pi
- **Required**: Own laptop as advised by University of Galway (There will be no PCs in the lab, just docking stations)

- **Two Slots:**
    - Wednesdays 4pm to 6pm, IT101 Lab
    - Thursdays 4pm to 6pm, IT101 Lab
    - ➢ You need to choose **one slot** (the class has to be split into two groups of the same size)
- https://nuigalwayie-my.sharepoint.com/:x:/g/personal/0128261s_nuigalway_ie/EZknOSEhPqNJpSiGH3VI87oBh9W0IIUxni8mLEIOnlBdgA?e=fqfhbh

Assessment

- 70% Final Exam
- 30% Continuous Assessment:
  - Assignment 1: 15%
  - Assignment 2: 15%

- You **must** attend in-class assignments (to be announced in advance)

Contact Details

- For lecture/lab related questions
  - I have also create a **Forum** on Blackboard where you can ask questions directly to me.
  - Any questions related to lectures or labs must be asked in the Forum, if you think that all the class would like to know the answer.
    - You can ask questions anonymously if you wish

- For one-to-one interaction with me:
  - The best way to contact me is by email: takfarinas.saber@universityofgalway.ie
    - I will endeavour to respond within 48 hours.
    - If you would like to send me an email, use you University of Galway email account and state in the email: name, Module ID CT213, and Student ID.
  - I will also put in place a time period to discuss directly with you when necessary.

Learning Outcomes

Upon successful completion of this module, you will be able to:

- Explain the ***main concepts*** *behind* any **Operating System** and implement some of them
- ***Write shell scripts*** (system programming) to interact with Linux system to ***solve problems***
- Examine how **processes, memory**, **file**, and **device systems** are **managed** in a computer
- Investigate an **ARM processor** and **Raspberry Pi** Device

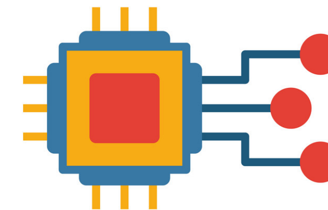
Syllabus

- Programming Models
- System software and Operating Systems
- Process Management and Process Synchronisation
  - ARM Processors
- Memory Management
- Device Management
- File Management
- Raspberry Pi

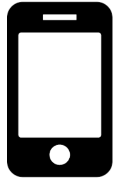
Overview of Computer Systems

Traditional Classes of Computer Systems

- **Personal Computer (PC):** A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.
- **Server:** A computer used for running larger programs for multiple users, often simultaneously, and typically accessed only via a network.
- **Supercomputer:** A class of computers with the highest performance and cost; they are configured as servers and typically cost tens to hundreds of millions of dollars.
- **Embedded computer**: A computer inside another device used for running one predetermined application or collection of software.

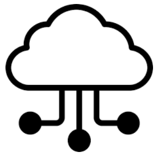


Post-PC Era



Personal mobile devices:

- Small wireless devices to connect to the Internet.
- They rely on batteries for power, and software is installed by downloading apps.
- Conventional examples are smart phones and tablets.

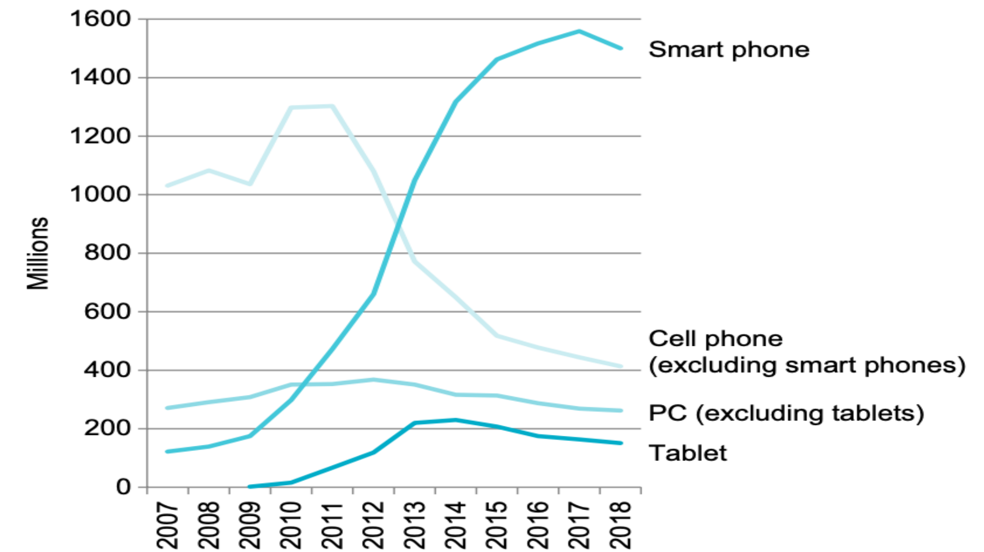


Cloud Computing:

- Refers to large collections of servers that provide services over the Internet;
- Some providers rent dynamically varying numbers of servers as a utility.

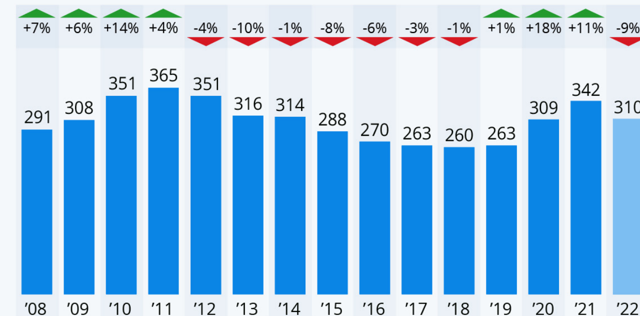
Software as a Service:

- Delivers software and data as a service over the Internet, usually via a thin program such as a browser.
- Examples include web search and email.



PC Demand Set to Slump After Pandemic Boost

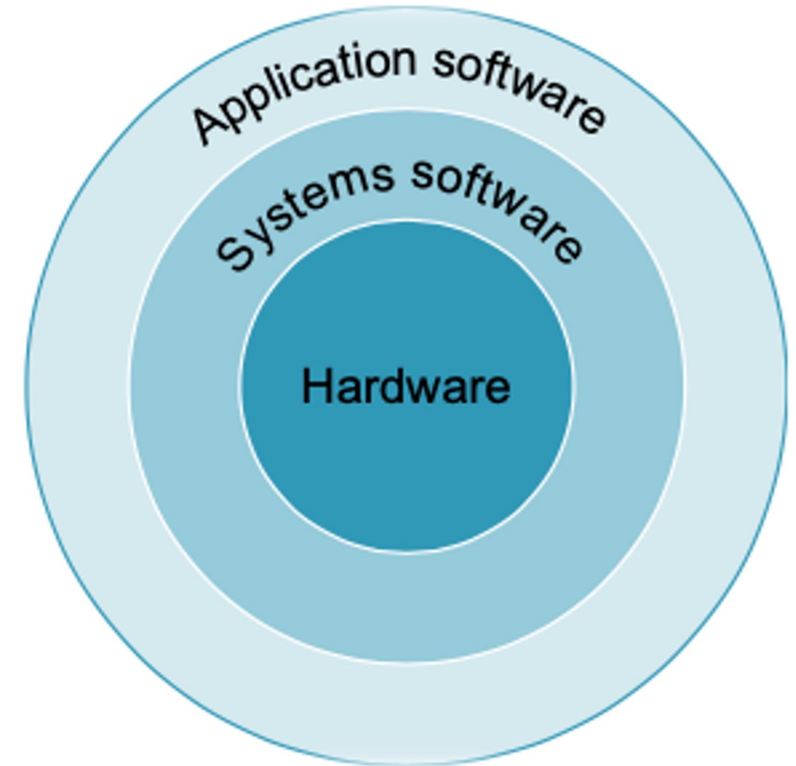
Estimated worldwide PC shipments (in million units)*



* incl. desktop PCs, notebooks and ultramobile premiums (e.g. Microsoft Surface), but NOT iPads or other tablets. Figures from 2020 onward include Chromebooks.
Source: Gartner

Computer Systems

- Application Software that provides services that are commonly useful.
- Operating system interfaces between a user's program and the hardware and provides a variety of services and supervisory functions.
- Hardware performs the tasks.
  - Which tasks?



Seven Great Ideas in Computer Organisation

1. Use Abstraction to Simplify Design

A major productivity technique for hardware and software is to use **abstractions** to characterize the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels.

2. Make the Common Case Fast

Making the **common case fast** will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is usually easier to enhance.

Seven Great Ideas in Computer Organisation

3. Performance via **Parallelism**

   Since the dawn of computing, computer architects have offered designs that get more performance by computing operations in parallel.

4. Performance via **Pipelining**

   A particular pattern of parallelism is so prevalent in computer architecture that it merits its own name: pipelining.

5. Performance via **Prediction**

   In some cases, it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate.

Seven Great Ideas in Computer Organisation

6. Hierarchy of Memories

Architects have found that they can address conflicting demands with a **hierarchy of memories:** the fastest, smallest, and the most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom.

7. Dependability via Redundancy

Since any physical device can fail, we make systems **dependable** by including redundant components that can take over when a failure occurs *and* to help detect failures.

Why Study Computing Systems & Organisation

- To get a job

Outline

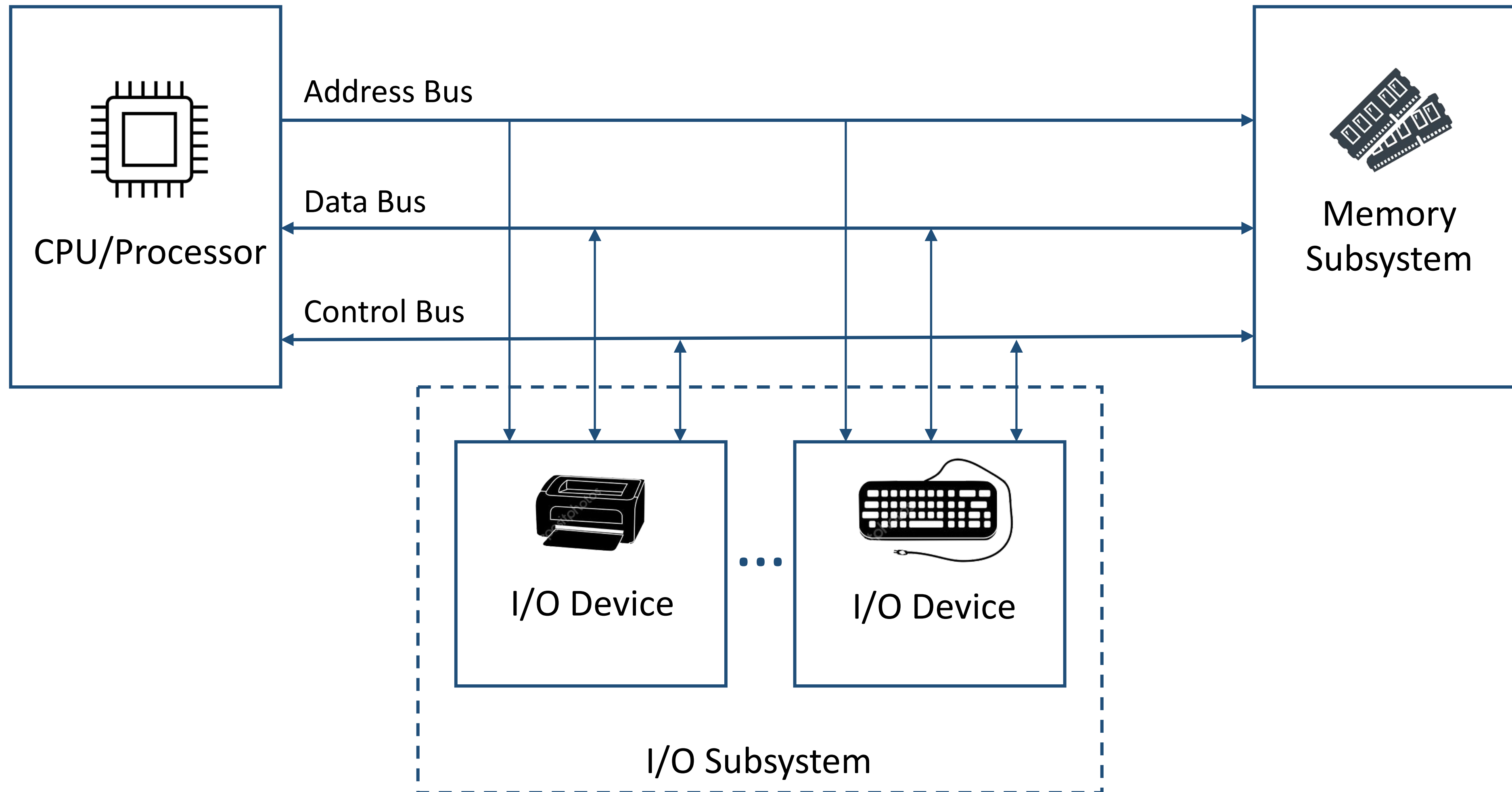
- Hardware Organisation
- Programs
- Operating System

- *Take home message:*
  - *A good understanding of how computing systems are organised is critical of IT professionals*

Hardware Organisation

How Does a Computer Look Like?

Basic Computer Organisation



Apple iPhone XS Max smart phone

At the left is the capacitive multitouch screen and LCD display.

Next to it is the battery.

To the far right is the metal frame that attaches the LCD to the back of the iPhone.

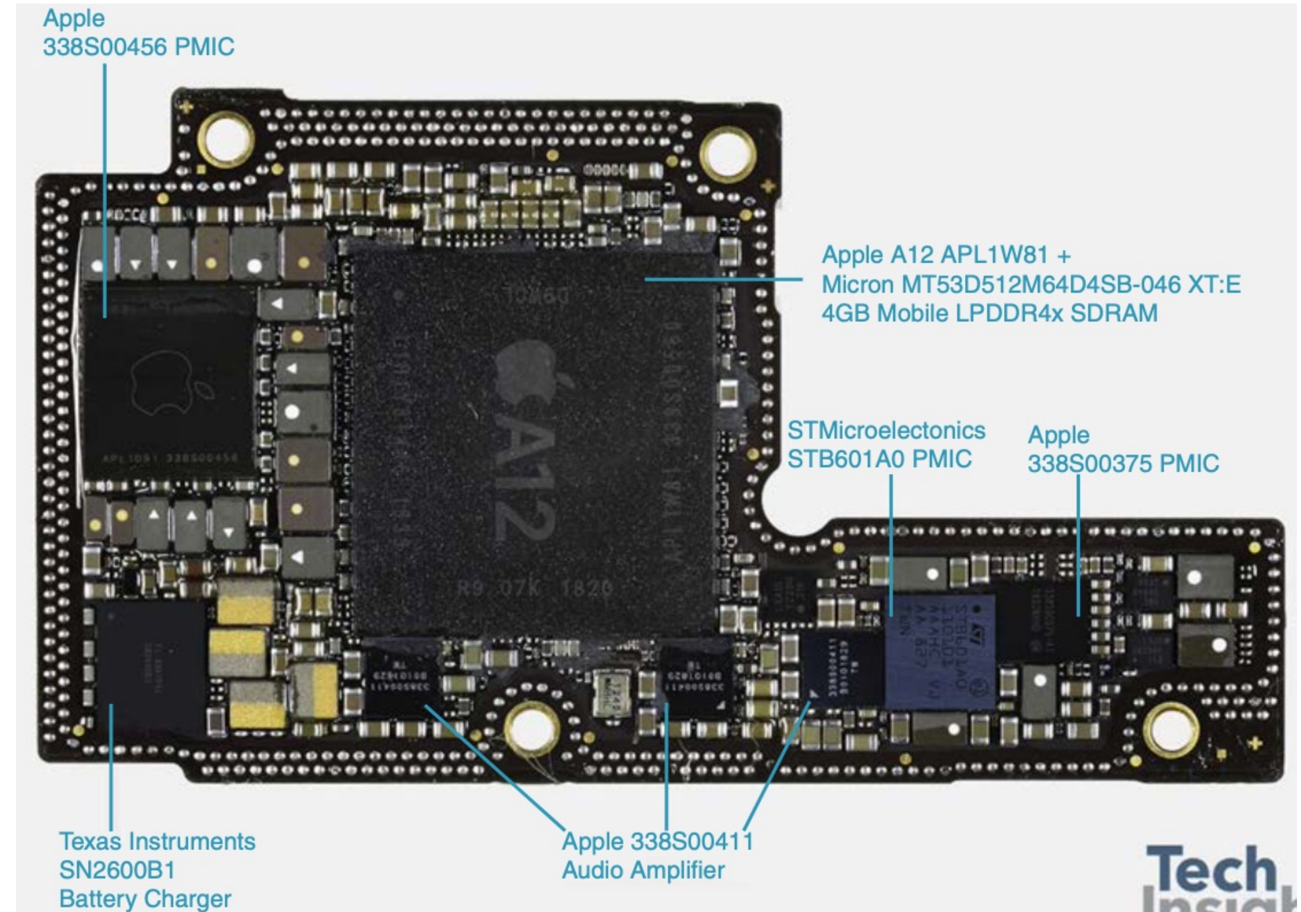
The small components in the center are what we think of as the computer; Where?

The logic board of Apple iPhone XS Max

Integrated circuit: also called a **chip**. A device combining dozens to millions of transistors.

Central Processor Unit (CPU): also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

The other chips on the board include the **Power Management Integrated Controller (PMIC)** and **Audio Amplifier** chips.

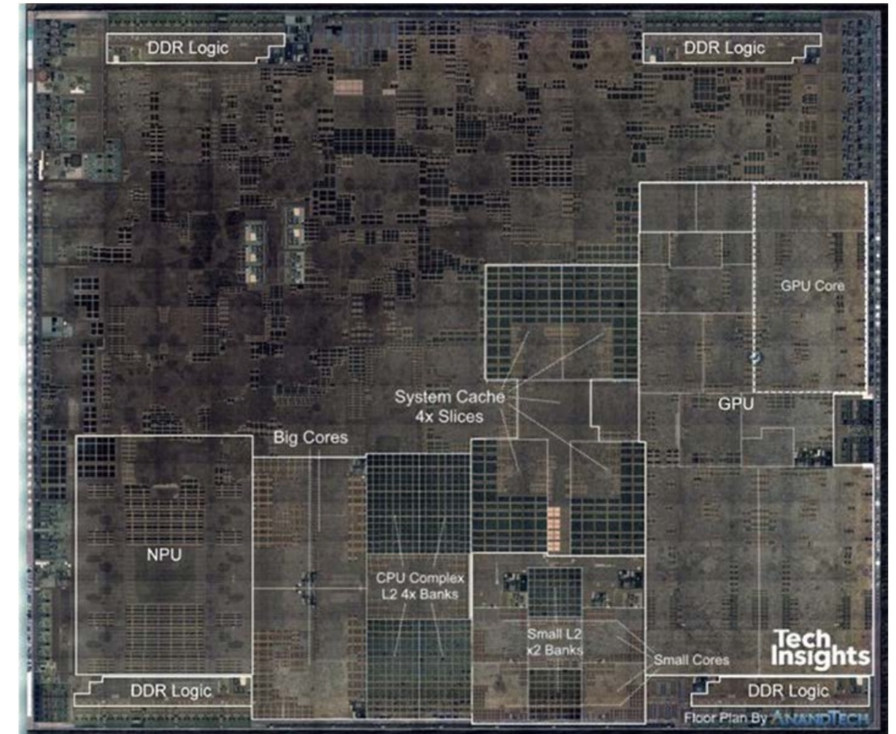


Central Processing Unit (CPU/Processor)

Responsible for executing programs

Processes programs in four steps:

1. **Fetch**: retrieve an instruction from program memory
2. **Decode**: break down the instruction into parts that have significance to specific sections of the CPU
3. **Execute**: Various portions of the CPU are connected to perform the desired operation
4. **Write Back**: Simply “writes back” the results of the execute step if necessary

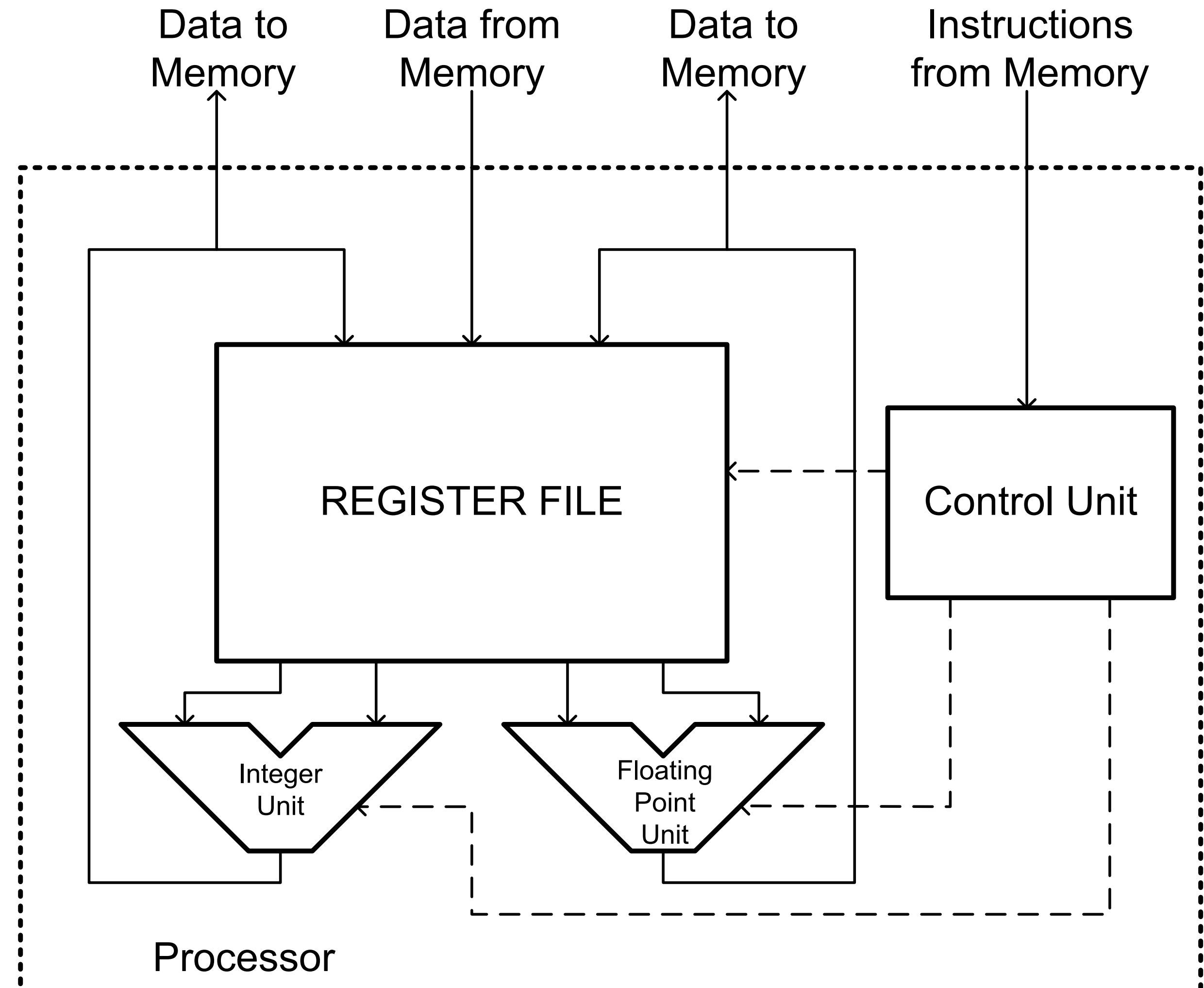


The processor integrated circuit inside the iPhone A12 package

CPU Organisation

Processors are made of:

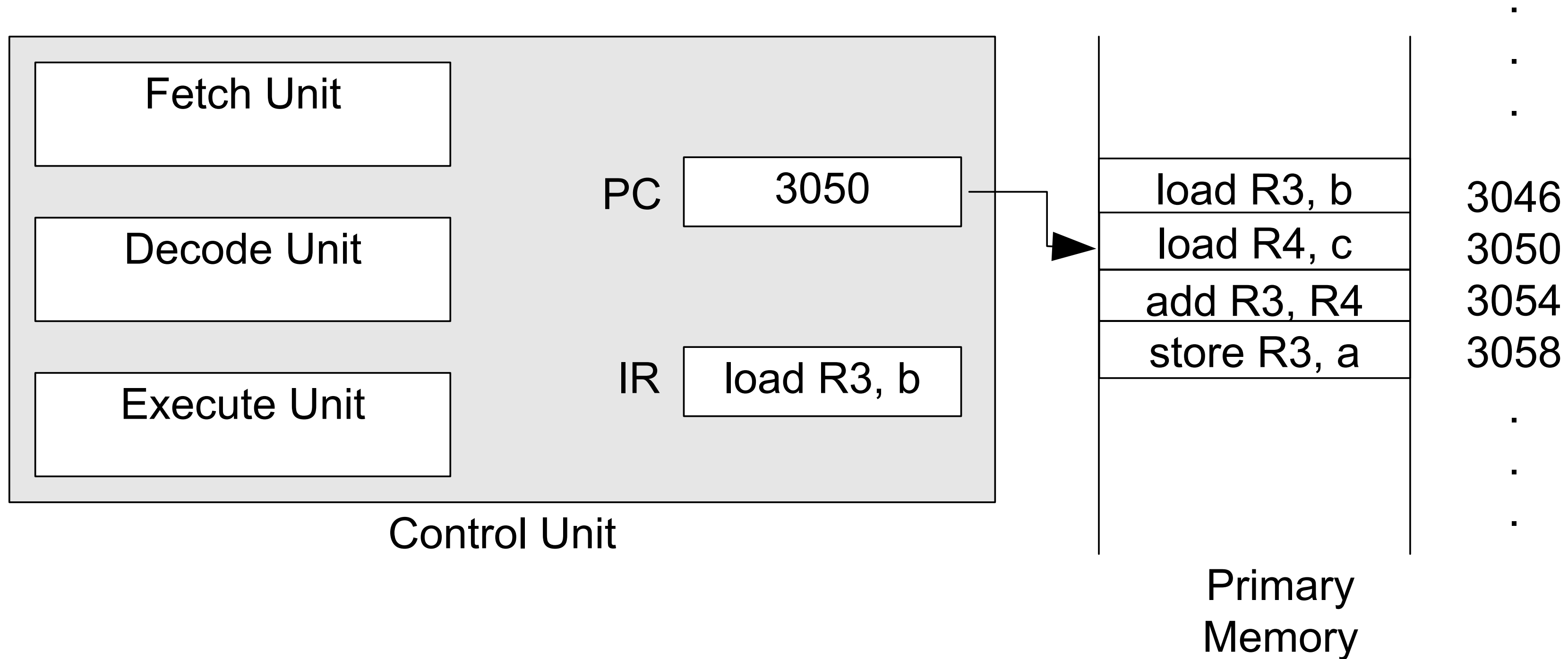
- Control Unit
- Execution Unit(s)
- Register file



Control Unit

The Control Unit controls the execution of the instructions stored in the main memory

- It retrieves and executes them



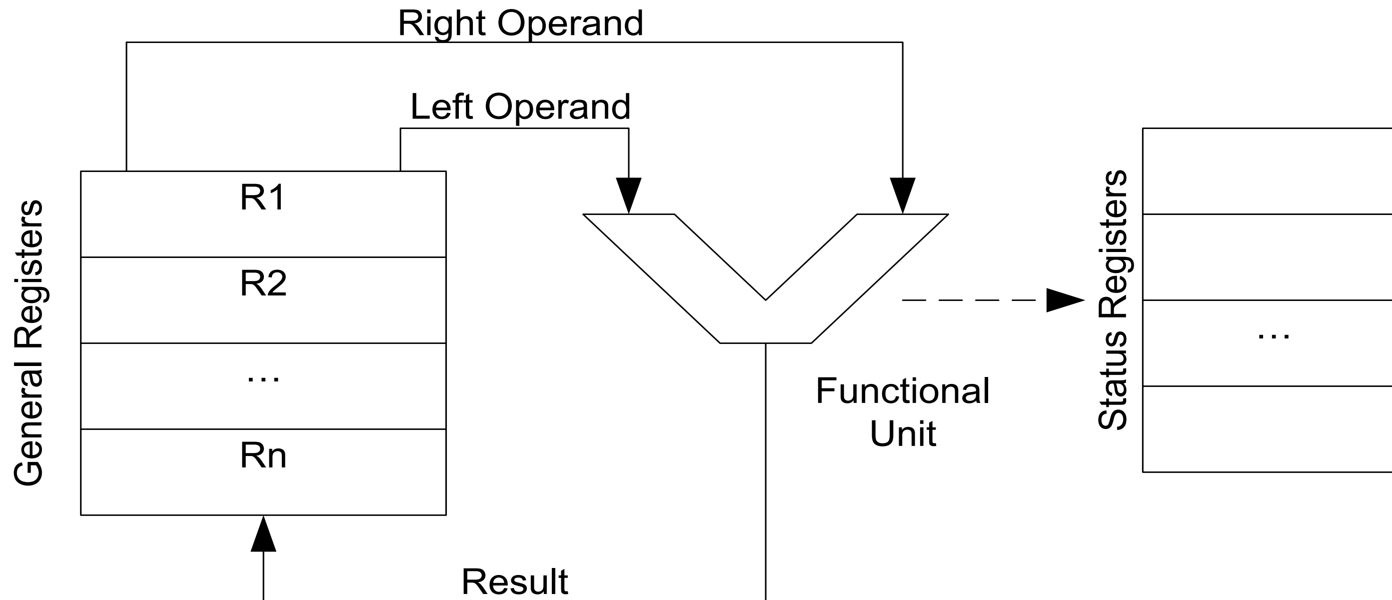
Special Registers:

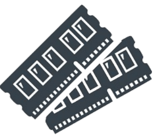
- Program counter (PC): keeps the address of the next instruction
- Instruction Register (IR): keeps the instruction being executed

Execution Unit Example

Code for a = b + c :

- LD R3, b //Load (copy) value b from memory to R3
- LD R4, c // Load (copy) value c from memory to R4
- add R3, R4 //sum placed in R3
- ST R3, a //store the result into memory as a

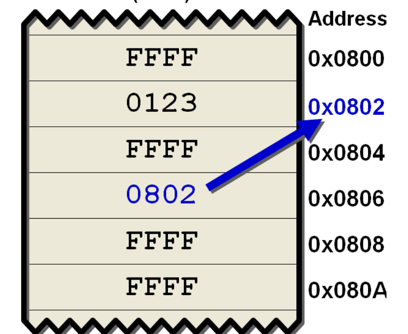




Memory Subsystem

- Memory is divided into a set of storage locations which can hold *data*.
  - Locations are numbered
  - Locations (i.e., *address*) are used to tell the memory which location the processor wants to access.

- There are two hierarchies of memory:
  1. **Nonvolatile / ROM (Read Only Memory):** Read only
     - Used to store the BIOS and/or a *bootstrap* or *boot loader* program
  2. **Volatile / RAM (Random Access Memory):** Read/write
     - Also called Primary Memory
     - Used to hold the programs, operating system and data required by the computer



Primary Memory

- Primary Memory is directly connected to the central processing unit of the computer.
  - ➢ It must be present for the CPU to function correctly.

- 3 types of primary storage:

Processors Register: Contains information that CPU needs to carry out the current instruction.

Cache memory: Special type of internal memory used by many CPUs to increase their performance or "throughput.

Main memory: Contains the programs that are currently being run and the data the programs are operating on.

Memory Subsystem (Cont’d)

- The width of **address** limits the amount of memory that a computer can access
  - Most current computers use a 64 bit address, which means that the maximum number of locations is 2^{64}, about (16 billion gigabytes).

- Memory subsystem supports **two operations**:
  - Load (or read) – address of the data location to be read
  - Store (or write) – address of the location and the data to be written

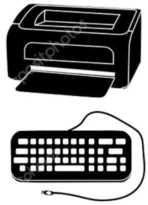
- Memory subsystem allows for more than 1 byte to be read or written at a time
  - Read and write operations operate at the width of system’s data bus, usually 32 bit (4 bytes), or 64 bits (8 bytes).
  - The address contains the address of the lowest byte to be addressed
  - e.g., with a 4 byte read operation from address 0x1000 would return bytes stored at addresses: 0x1000, 0x1001, 0x1002 and 0x1003

Memory Alignment and Word of Data

- When the computer's word size is 4 bytes the data to be read should be at a memory address which is some multiple of 4.

- When this is not the case, e.g. the data starts at address 14 instead of 16, then the computer has to read two or more 4 byte chunks and do some calculation before the requested data has been read, or it may generate an alignment fault.

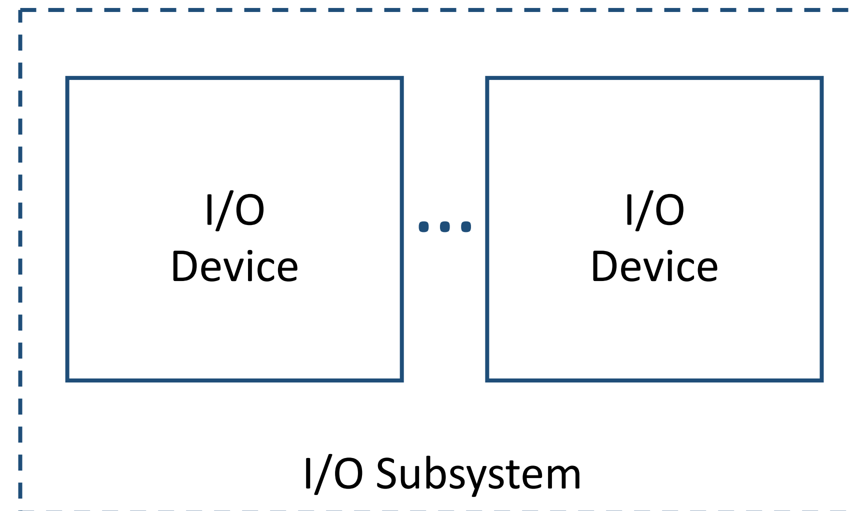
- Even though the previous data structure end is at address 13, the next data structure should start at address 16. Two padding bytes are inserted between the two data structures at addresses 14 and 15 to align the next data structure at address 16.



Input/Output

Input Devices: Anything that feeds data into the computer

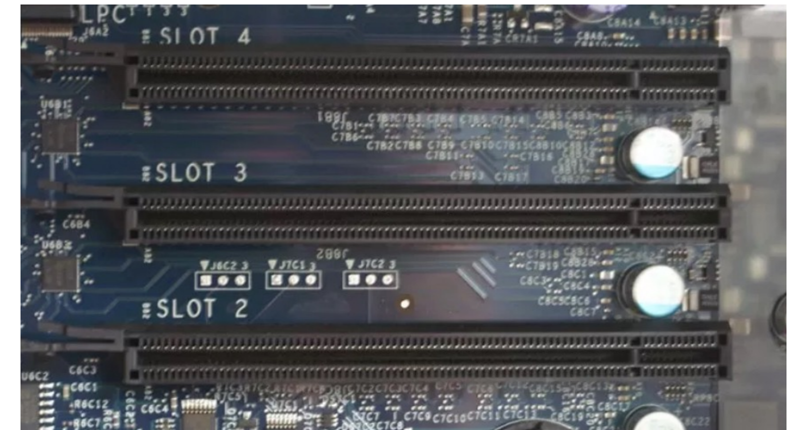
Output Devices: Display / transmit information back to the user



The I/O Subsystem

- Contains devices that the computer uses to communicate with the outside world and to store data

- I/O devices are usually communicating with the processor using an I/O bus
  - PCs are using PCI Express (Peripheral Component Interconnect Express) bus for their I/O bus
  - OS needs a ***device driver*** to access a given I/O device
    - Program that allows the OS to control the I/O device



PCI Express (PCIe 5.0)

I/O Read/Write Operations

- The I/O read and write operations are similar to the memory read and write operations.

- A processor may use:
  - **memory mapped I/O** – when the address of the I/O device is in the direct memory space, and the sequence to read/write data in the device are the same with the memory read/write sequence
  - **isolated I/O** – the process is similar, but the processor has a second set of control signals to make the distinction between a memory access and an I/O access

- **IO/M** signal **is a status signal.**
  - When this signal is low (IO/M = 0) it denotes the memory related operations.
  - When this signal is high (IO/M = 1) it denotes an I/O operation.

Programs

Programs

- Sequences of instructions that tell computer what to do
- To the computer, a program is made out of a sequence of numbers that represent individual operations.
  - Those operations are known as ***machine instructions*** or just ***instructions***
  - A set of instructions that a processor can execute is known as ***instruction set***

Program Development Tools

From a High-Level Language to the Language of Hardware

High-level programming language: A portable language such as C, C++, Java, or Visual Basic that is composed of words and algebraic notation that can be translated by a compiler into assembly language.

Compiler: A program that translates high-level language statements into assembly language statements.

Assembler: A program that translates a symbolic version of instructions into the binary version.

Assembly language: A symbolic representation of machine instructions.

Machine language: A binary representation of machine instructions.

Instruction: A command that computer hardware understands and obeys.

High-level language program (in C)

```
swap(size_t v[], size_t k)
{
   size_t temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

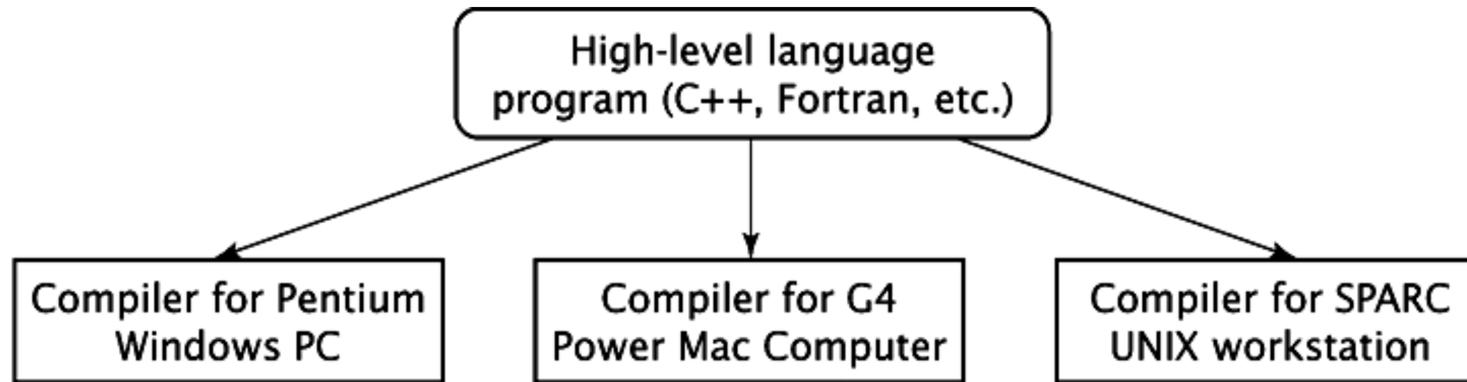
Assembly language program (for RISC-V)

```
swap:
     slli x6, x11, 3
     add  x6, x10, x6
     lw   x5, 0(x6)
     lw   x7, 4(x6)
     sw   x7, 0(x6)
     sw   x5, 4(x6)
     jalr x0, 0(x1)
```

Assembler

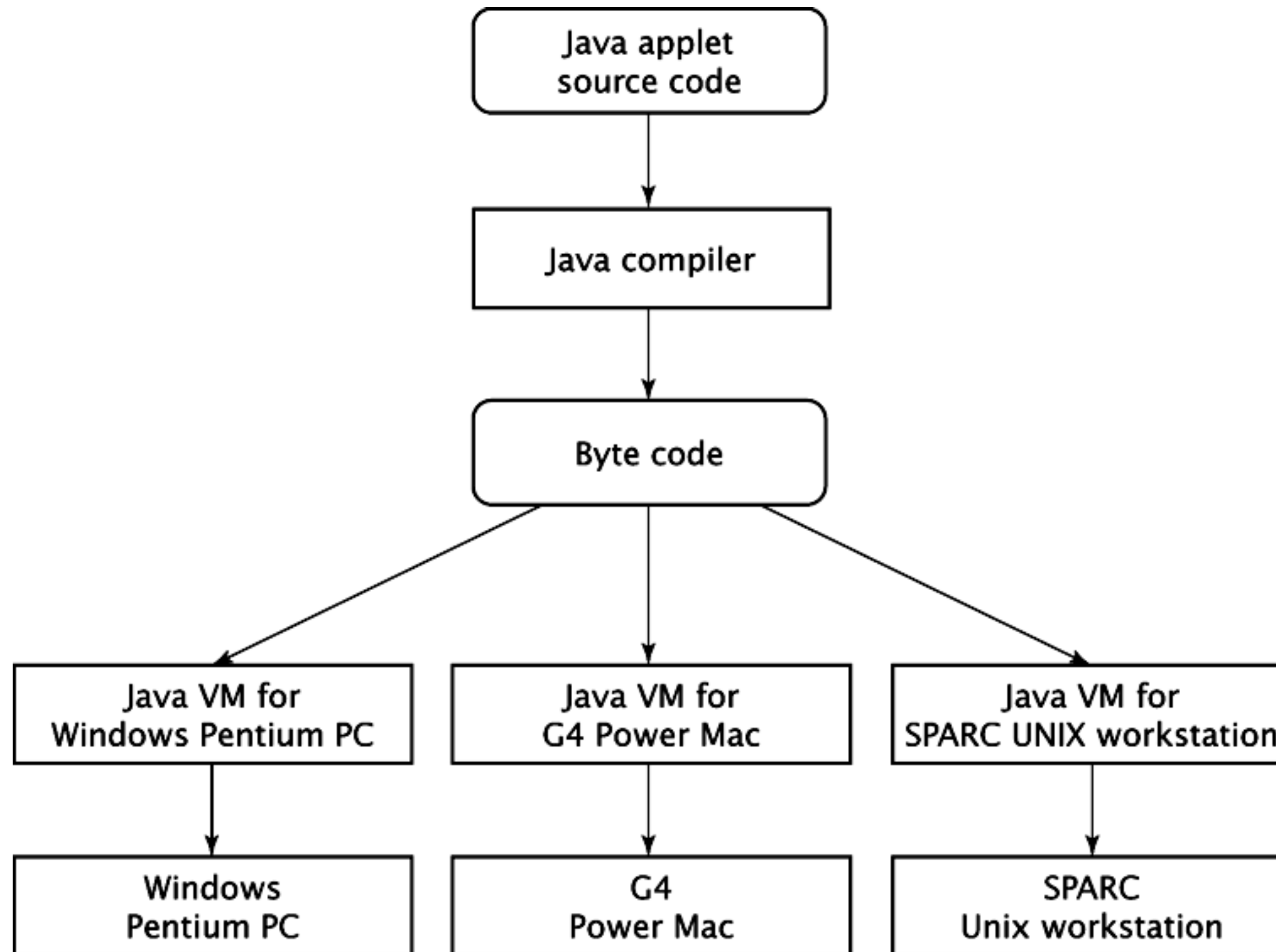
Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```



Compiling Programs

Java – Different way of processing



Operating Systems

Operating System

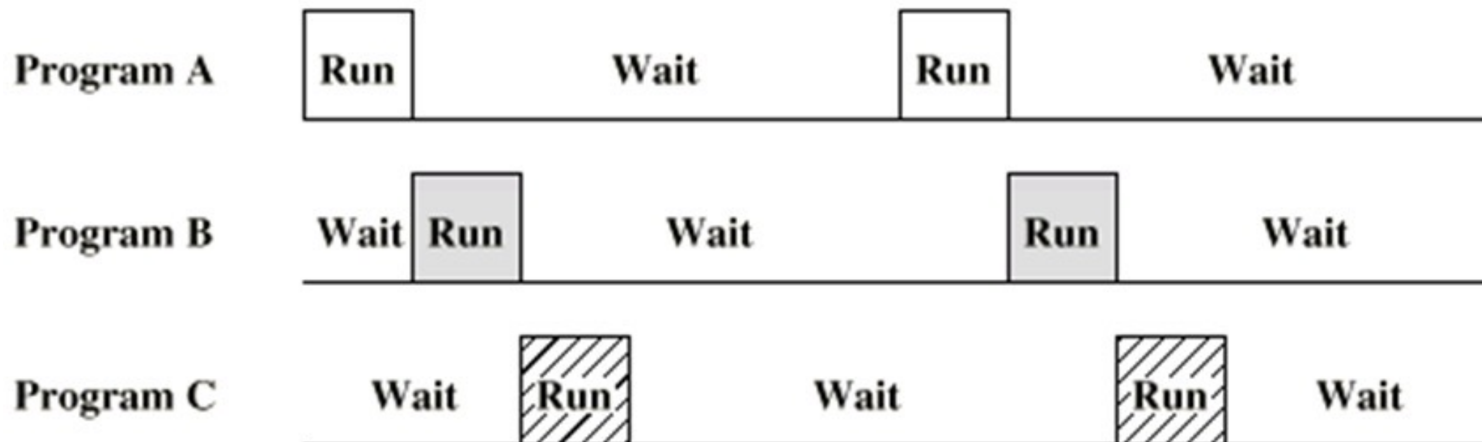
- Responsible for managing the physical resources of complex systems (PCs, workstations, mainframe computers)
- Responsible for loading and executing programs and interfacing with the users
- Usually, no operating system for small embedded systems
  - Computers designed for one specific task
- A possible definition: It is a **program** that runs on the computer, that **knows about all the hardware** and usually runs in ***privileged*** (or supervisor) mode, **having access** to physical resources that user programs can't control and has the **ability to start and stop user programs**

Dr Takfarinas Saber <takfarinas.saber@universityofgalway.ie>

Multiprogramming

- Technique that allows the system to present the illusion that multiple programs are running on the computer simultaneously

- Many multiprogrammed computers are ***multiuser***
  - Allow multiple persons to be logged on at a time



Multiprogramming (Cont’d.)

Multiprogramming is achieved by switching rapidly between programs

- FCFS – First Come, First Served (also called FIFO)
    - processes are moved to the CPU in the order in which they arrive

SJN – Shortest Job Next

looks at all processes in the ready state and dispatches the one with the smallest service time

Round Robin

distributes the processing time equitably among all ready processes

Advantages:

- simple
- easy to implement
- starvation-free

Disadvantages:

- set time too short => too much process switching => too slow.
- set time too long => unresponsive system, time wasting

Context Switch

- When a program timeslice ends, the OS stops it, removes it and gives another program control over processor
  - This is a ***context switch***

- To do a context switch the OS:
  - Copies the content of current program register file into memory
  - Restores the contents of the next program’s register file into the processor
  - and starts executing the next program.

- From the program point of view, no program can tell that a context switch has been performed

Protection

- Three rules:
  1. The result of any program running on a multiprogram computer **must be the same** as if the program was the only program running on the computer

  2. Programs **must not be able to access** other program's data and must be confident that their data will not be modified by other programs.
     - For **security** and ***privacy***

  3. Programs **must not interfere** with other program's use of I/O devices

How to Achieve Protection

Protection is achieved by the operating system having full control over the resources of the system (processor, memory and I/O devices) through:

- **Privileged Mode:** the operating system is the only one that can control the physical resources it executes in privileged mode
  - User programs execute in ***user mode***
- **Virtual Memory:** each program operates as if it were the only program on the computer, occupying a full set of the address space in its virtual space.
  - The OS is ***translating*** memory addresses that the program references into physical addresses used by the memory system.

References

- “Computer Systems Organization & Architecture”, John D. Carpinelli, ISBN: 0-201-61253-4

Ollscoil na Gaillimhe
University of Galway

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing Systems & Organisation

Programming Models

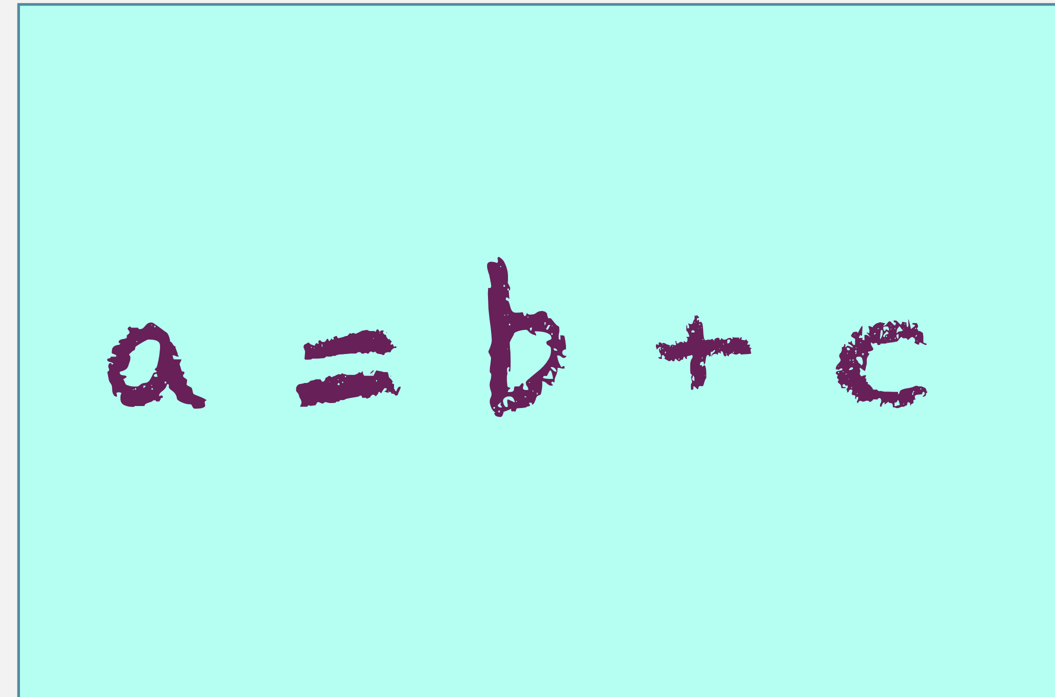
Outline

- Instruction types
- Stack
- Stack architectures
- GPR architectures
- Stack used to implement procedure calls

Programming Models

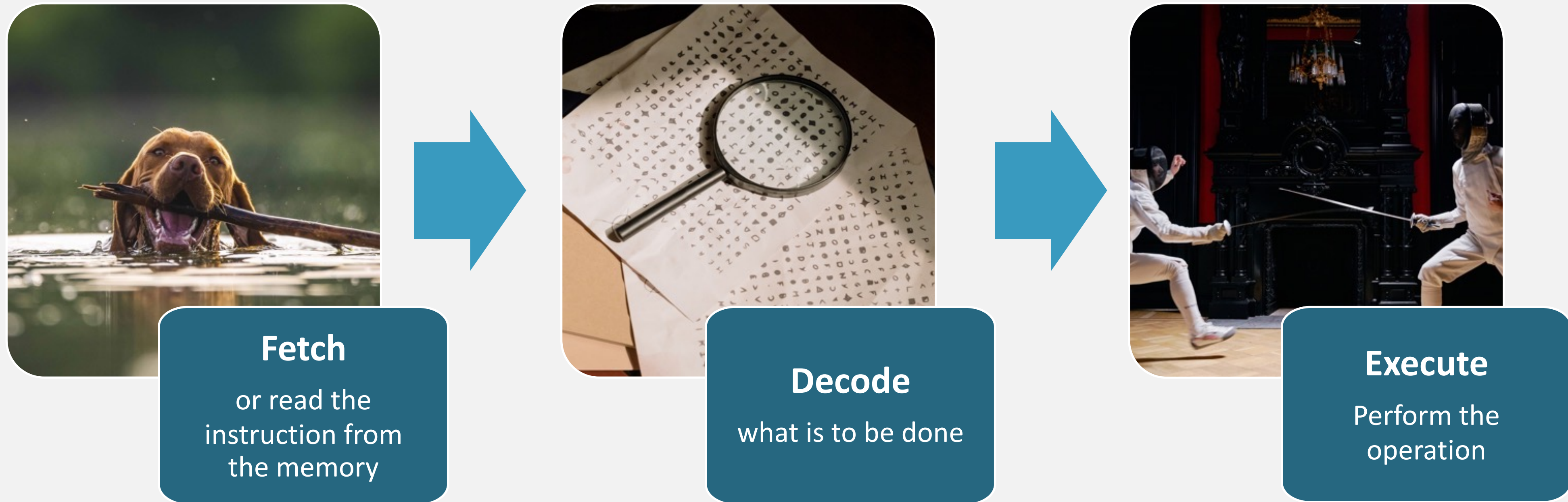
A processor programming model defines how instructions access their operands and how instructions are described in the processor's assembly language

Processors with different programming models can offer similar sets of operations but may require very different approaches to programming



The Processor - Instruction Cycles

- The instruction cycle is the **procedure of processing an instruction** by the microprocessor:



- Each of the functions fetch -> decode -> execute consist of a sequence of one or more operations inside the CPU (and interaction with the subsystems)

Types of Instructions

- **Data Transfer** Instructions
  - Operations that **move data** from one place to another
  - These instructions **don’t modify** the data, they just copy it to the destination

- **Data Operation** Instructions
  - Instructions do modify their data values
  - They typically perform some operation (e.g., +/-/*) using one or two data values (operands) and store the result

- **Program Control** Instructions
  - **Jump** or **branch** instructions used to **go in another part** of the program; Jumps can be **absolute** or **conditional** (e.g., if then else)
  - Instructions that can generate **interrupts** (software interrupts)

Data Transfer Instructions (1)

Load data from memory into the microprocessor

These instructions copy data from memory into microprocessor registers (i.e., LD)

Store data from the microprocessor into the memory

Similar to load data, except that the data is copied in the opposite direction (i.e., ST)

Data is saved from internal microprocessor registers into the memory

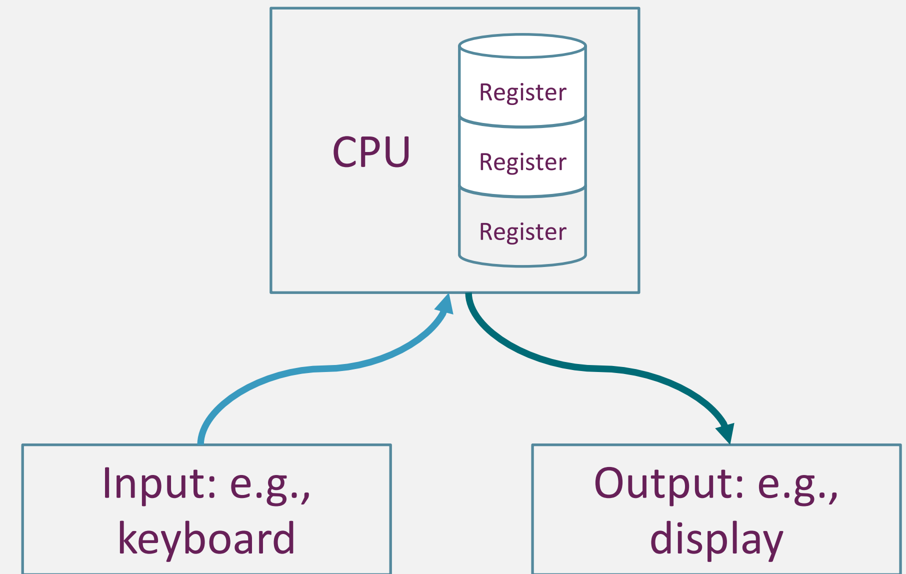
Move data within the microprocessor

These instructions move data from one microprocessor register to another (i.e., MOV)

Data Transfer Instructions (2)

- **Input data** to the microprocessor
  - A microprocessor may need to input data from the outside world, these are the instructions that input data from the input device into the microprocessor
  - An example: microprocessor needs to know which key was pressed (i.e., IORD)

- **Output data** from the microprocessor
  - The microprocessor copies data from one of its internal registers to an output device
  - In example: microprocessor may want to show on a display the content of an internal register (the key that has been pressed) (i.e., IOWR)



Data Operation Instructions

- **Arithmetic instructions**
  - add, subtract, multiply or divide
    - ADD, SUB, MUL, DIV, etc.
  - Instructions that increment or decrement one from a value
    - INC, DEC
  - Floating point instructions that operate on floating point values
    - FADD, FSUB, FMUL, FDIV

- **Logic Instructions**
  - AND, OR, XOR, NOT, etc.

- **Shift Instructions**
  - SR, SL, RR, RL, etc.

Program Control Instructions (1)

- **Jump and branch instructions (**Conditional or unconditional**):**
  - JZ: Jump if the zero flag is set
  - JNZ: Jump if the zero flag is NOT set
  - JMP: Unconditional jump; flags are ignored
  - Etc.

- **Comparison instructions:**
  - TEST: logical BITWISE AND

- **Calls and returns** a/from a routine **(**Conditional or unconditional**):**
  - CALL: call a subroutine at a certain line
  - RET: return from a subroutine
  - IRET: interrupt and return

Program Control Instructions (2)

- **Software interrupts:**
  - Generated by devices outside of a microprocessor (not part of the instruction set)
  - called **hardware interrupts**
  - INT
- **Exceptions and traps:** triggered when valid instructions perform invalid operations,
  - E.g., dividing by zero
- **Halt instructions:** causes the processor to stop executions,
  - E.g., at the end of a program
  - HALT

https://www.tutorialspoint.com/assembly_programming/index.htm

Stack Architectures

Stack Based Architectures

- The Stack
- Implementing Stacks
- Instructions in a stack-based architecture
- Stack based architecture instruction set
- Programs in stack-based architecture

The Stack (1)

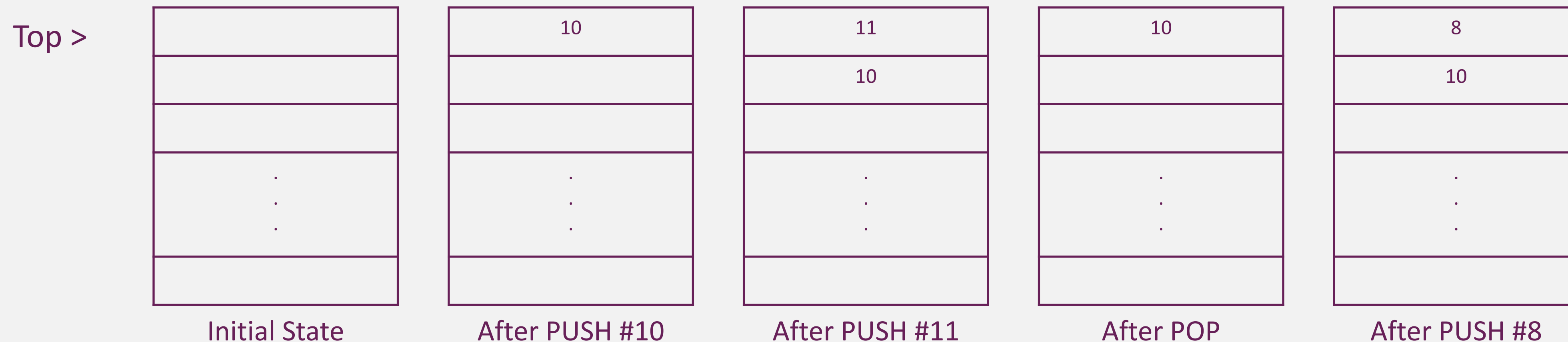
- **Last In First Out** (LIFO) data structure
- Consists of **locations**, each of which can hold a *word of data*
    - It can be used explicitly to **save/restore** data
- Supports two operations
    - **PUSH** – takes one argument and places the value of the argument in the top of the stack
    - **POP** – removes one element from the stack, saving it into a predefined register of the processor
- Used implicitly by procedure call instructions (if available in the instruction set)

The Stack (2)

When new data is added to the stack, it is placed at the top of the stack, and all the contents of the stack are pushed down one location

Consider the code:

```
PUSH #10
PUSH #11
POP
PUSH #8
```



Implementing Stacks

Two ways to implement a Stack:

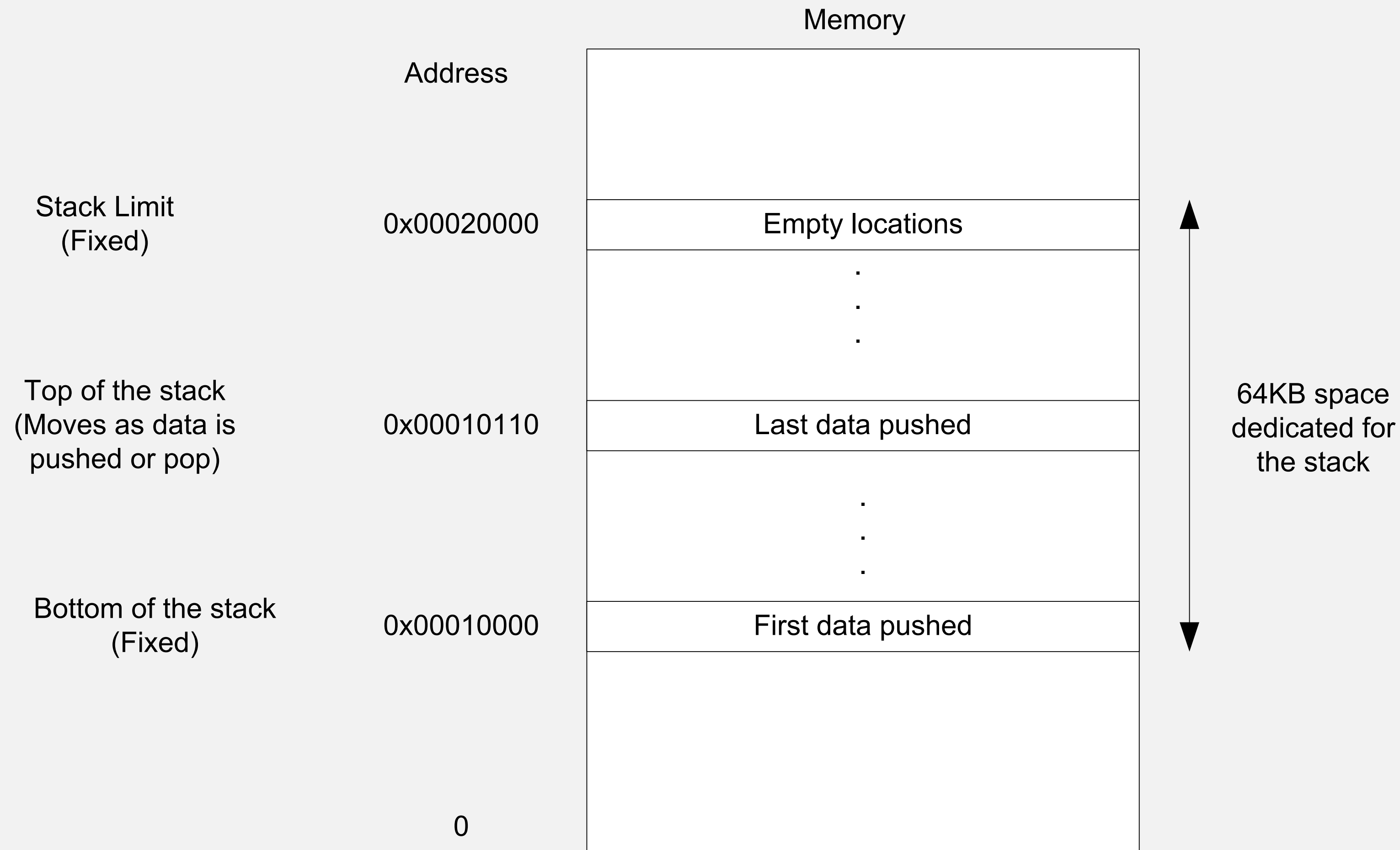
1. Dedicated hardware stack
   - It has a hardware limitation (limited number of locations)
   - Very fast
2. Memory implemented stack
   - Limited by the physical memory of the system
   - Slow compared with hardware stack, since extra memory addressing has to take place for each stack operation

Stack overflows can occur in both implementations

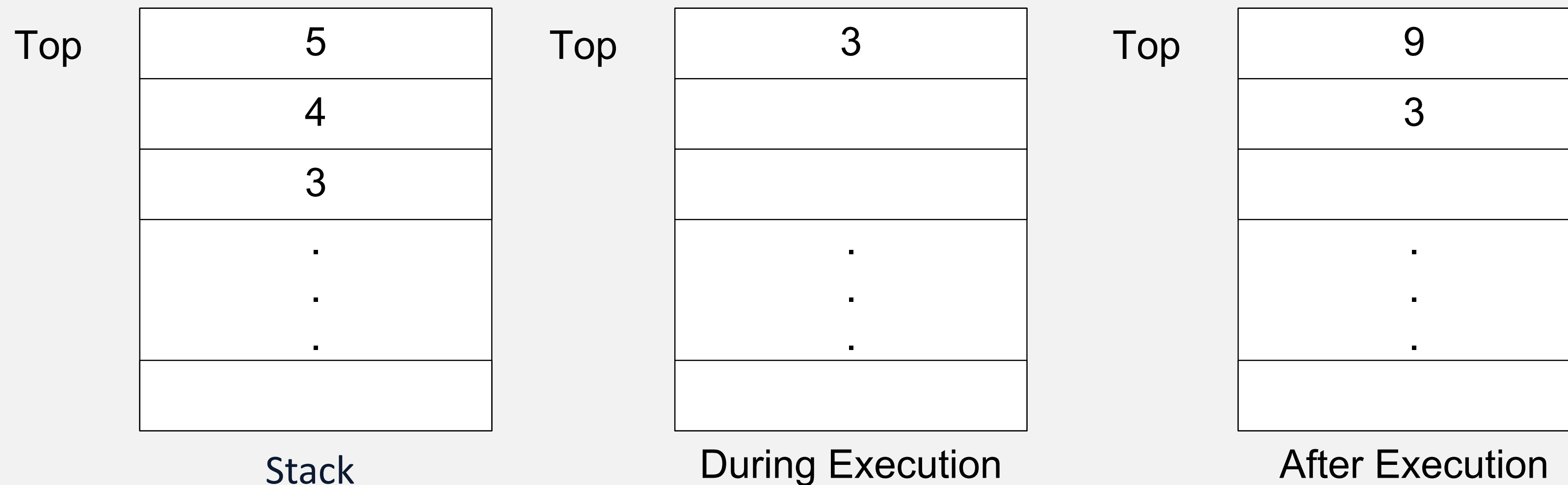
- When the amount of data in the stack exceeds the amount of space allocated to the stack (or the hardware limit of the stack)

Stack Implemented in Memory

- Every push operation will increment the top of the stack pointer (with the word size of the machine)
- Every pop operation will decrement the top of the stack pointer



Instructions in a Stack Based Architecture



- Get their operands from the stack and write their results to the stack
- Advantage - Program code takes little memory (no need to specify the address of the operands in memory or registers)
  Push is one exception, because it needs to specify the operand (either as constant or address)

Programs in Stack Based Architecture (1)

- Writing programs for stack-based architectures is not easy
  - Stack-based processors are better suited for postfix notation rather than infix notation

- **Infix** notation is the traditional way of representing math expressions, with operation placed between operands
  - E.g., a + b

- **Postfix** notation – the operation is placed after the operands
  - E.g., a b +

- Once the expression has been converted into postfix notation, implementing it in programs is easy

- Exercise: Create a stack-based program that computes: $A*(B-C)+(D+E)$

Simple Stack Based Instruction Set

Operation: $A*(B-C)+(D+E)$

	# stack contents (leftmost = top = most recent)
push A	# A
push B	# B A
push C	# C B A
subtract	# B-C A
multiply	# A*(B-C)
push D	# D A*(B-C)
push E	# E D A*(B-C)
add	# D+E A*(B-C)
add	# A*(B-C)+(D+E)

Dr Takfarinas Saber
<takfarinas.saber@universityofgalway.i

General Purpose Register Architecture

General Purpose Register Architecture

- Instructions in a GPR architecture
- A GPR instruction set
- Programs in GPR architecture

General Purpose Register Architecture (1)

- The instructions read their operands and write their results to **random access register file**.
- The general purpose register file allows the **access of any register in any order** by specifying the number (register ID) of the register
- The **main difference** between a general purpose register and the stack is that reading repeatedly a register will produce the same result and will not modify the state of the register file.
    - Popping an item from a LIFO structure (stack) will modify the contents of the stack

General Purpose Register Architecture (2)

Register File

Register 0	data
Register 1	data
Register 2	data
	. . .
Register n	data

- Many GPR architectures assign special values to some registers in the register file to make programming easier
  - I.e., sometimes, register 0 is hardwired with value 0 to generate this most common constant

Instructions in GPR Architecture (1)

- GPR instructions need to **specify:**
  - **the register** that hold their **input operands**
  - and the register that will hold the **result**

- The most common format is the **three operands instruction format**
  - E.g., **ADD r1, r2, r3** instructs the processor to read the contents of r2 and r3, add them together and write the result in r1

- Instructions having two or one input are also present in GPR architecture

Instructions in GPR Architecture (2)

- A significant difference between GPR architecture and stack-based architecture:
  - Programs can choose **which values should be stored in the register file** at any given time, allowing them to cache most accessed data
- In stack based architectures, once the data has been used, it is gone.
- GPR architectures have **better performance** from this point of view, at the expense of needing **more storage space** for the program
  - larger instructions need to encode the addresses of the operands

Simple GPR Instruction Set

ST (ra), rb	(ra) <- rb
LD ra, (rb)	ra <- (rb)
ADD ra, rb, rc	ra <- rb + rc
SUB ra, rb, rc	ra <- rb - rc
AND ra, rb, rc	ra <- rb & rc
OR ra, rb, rc	ra <- rb | rc
MOV ra, rb	ra <- rb

(ra): The memory location whose address is contained in ra

Programs in a GPR Architecture (1)

- Programming a GPR architecture processor is **less structured** than programming a stack based architecture one.

- There are **fewer restrictions on the order** in which the operations can be executed

- On stack based architectures, instructions should execute in the order that would leave the operands for the next instructions on the **top of the stack**

- On GPR, any order that places the operands for the next instruction **in the register file** before that instruction executes is valid.

- Operations that access different registers can be **reordered** without making the program invalid

Programs in GPR Architecture (2)

- Create a GPR based program that computes:
    - 2 + (7&3)

- GPR programming uses infix notation:

```
MOV R1,  #7
MOV R2,  #3
AND R3, R1, R2
MOV R4, #2
ADD R4, R3, R4
```

- The result will be placed in R4

Comparing Stack based and GPR Architectures

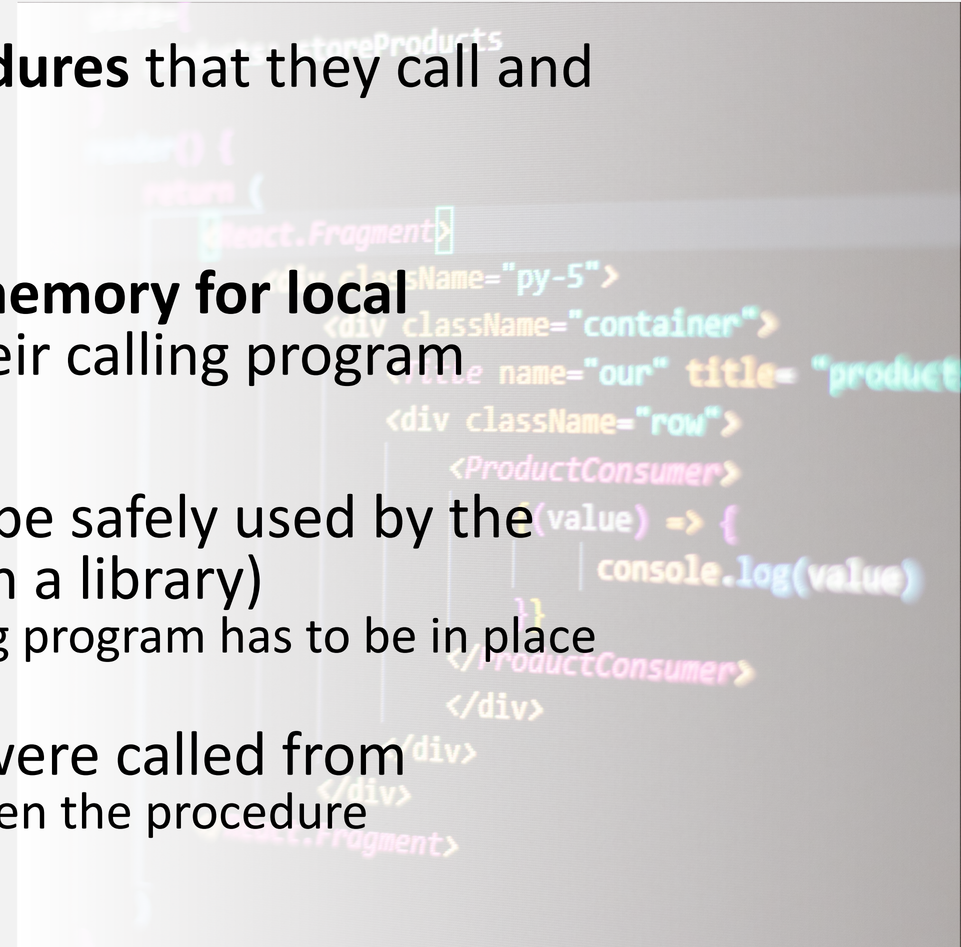
- **Stack-based architectures**
  - Instructions take **fewer bits** to encode
  - **Reduced amount of memory** taken up by programs
  - Manages the **use of register automatically** (no need for programmer intervention)
  - Instruction set does not change if size of register file has changed

- **GPR architectures**
  - With evolution of technology, the amount of space taken up by a program is less important
  - Compilers for GPR architectures achieve **better performance** with a given number of general purpose registers than those on stack-based architectures with same number of registers
    - The compiler can **choose which values to keep** (cache) in register file at any time

- Stack based processor are still attractive for certain embedded systems. GPR architectures are used by modern computers (workstations, PCs, etc.)

Stacks for Procedure Calls

Using Stacks to Implement Procedure Calls (1)



- Programs need a way to **pass inputs to the procedures** that they call and to receive outputs back from them

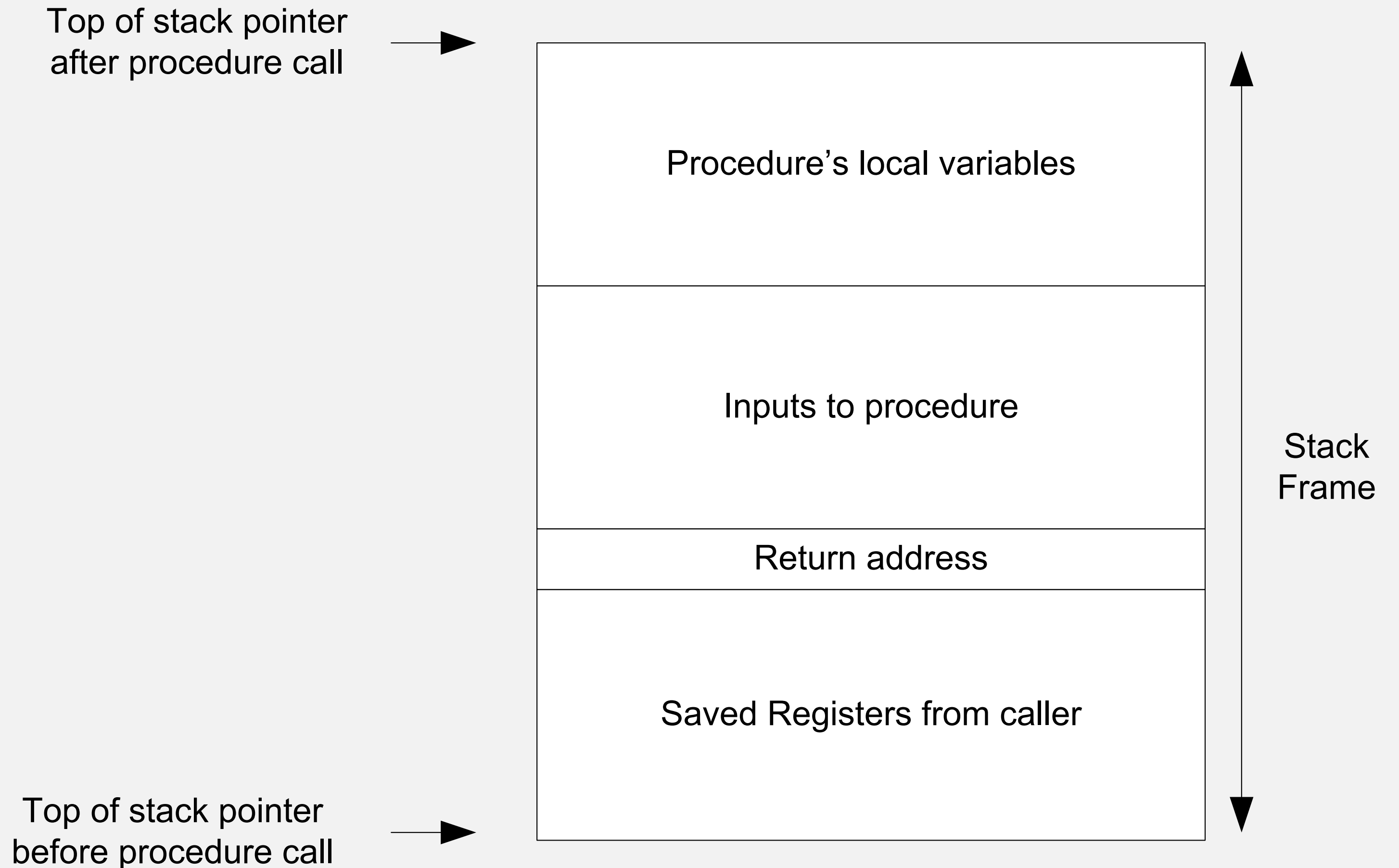
- Procedures need to be able to **allocate space in memory for local variables**, without overriding any data used by their calling program

- It is impossible to determine which registers may be safely used by the procedure (especially if the procedure is located in a library)
  - So, a mechanism to **save/restore registers** of the calling program has to be in place

- Procedures need a way to figure out where they were called from
  - So, the execution can **return to the calling program** when the procedure completes (they need to restore the program counter)

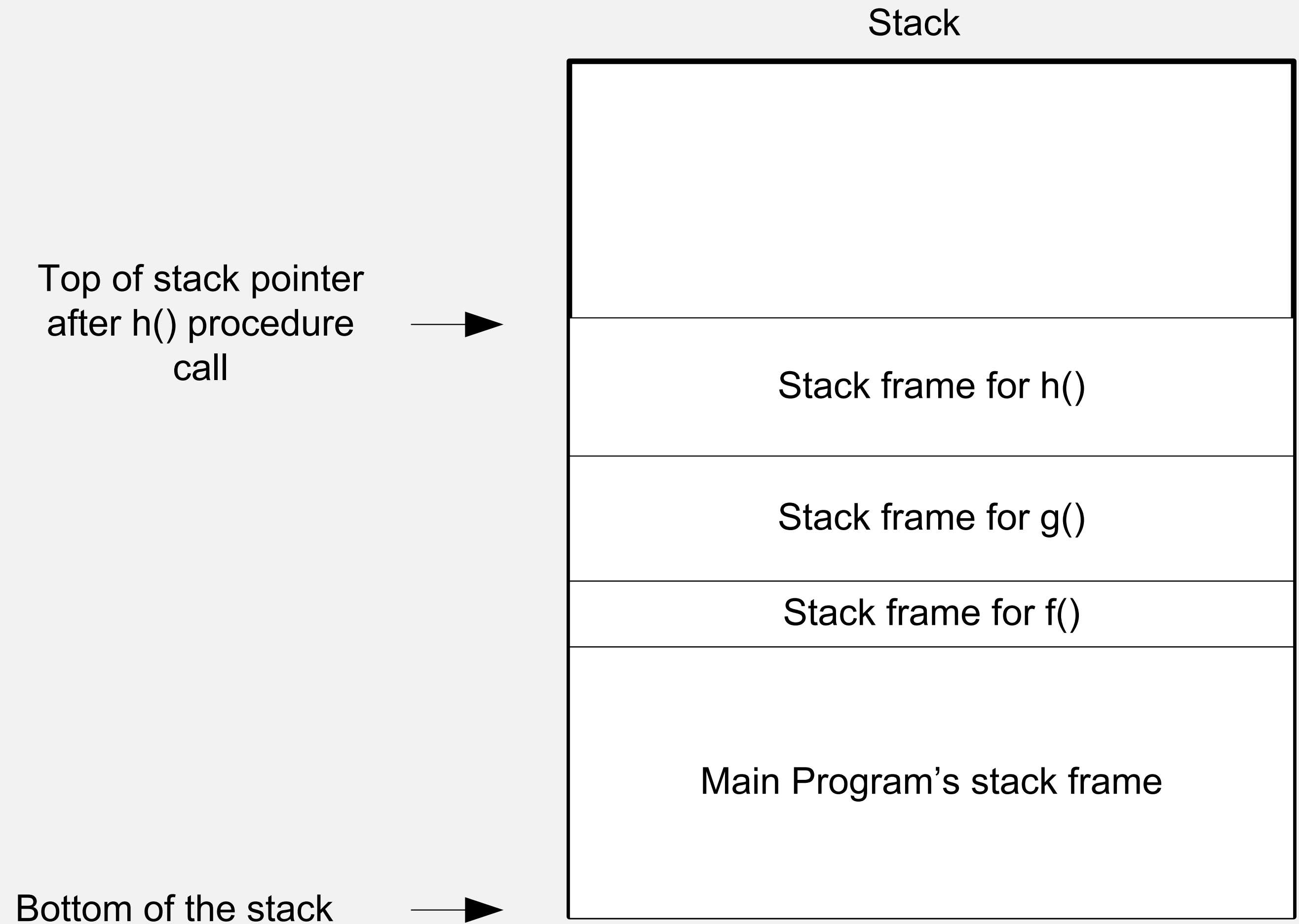
Using Stacks to Implement Procedure Calls (2)

- When a procedure is called, a **block of memory** in the stack is allocated. This is called a stack frame
- The top of the stack pointer is incremented by the **number of locations** in the stack frame
- When a procedure finishes, it jumps to the **return address** contained in the stack and execution of the calling program resumes.



Using Stacks to Implement Procedure Calls (3)

- Nested procedure calls:
  - main program calls function f(),
  - function f() calls function g(),
  - function g() calls function h()



References

- “Computer Systems Organization & Architecture”, John D. Carpinelli, ISBN: 0-201-61253-4
- “Computer Architecture”, Nicholas Charter, ISBN – 0-07-136207

- **Images taken from Pexels**:
- Photo of dog by Jozef Fehér
- Photo of magnifying glass and fencers by cottonbro
- Photo of mirror by sum+it
- Photo of hardware by Valentine Tanasovich
- Photo of tree by Johannes Plenio
- Photo of stop sign by Mwabonje
- Photo of stack by Monstera
- Photo of drawers by Stephan Streuders
- Photo of code by Antonio Batinić

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing Systems & Organisation

Lecture 3: System Software & Operating Systems

Contents

- System Software & OS
- OS Organisation
- OS Design and Implementation
- Implementation considerations
  - Processor modes
  - Kernel
  - Requesting services from OS

Photo by Vitaly Vlasov from Pexels

System Software & OS

Application Software

- A computer program designed to **perform a group of coordinated functions** for the benefit of the
- **Application software** is design solve a specific problem
- Examples of an application include word processor, a spreadsheet, an accounting application, a web browser, photo editor etc.

Photo by Pixabay from Pexels

System Software

- Programs dedicated to **managing the computer**
- System software is a software that **provides a platform** to other software.

- **System software** provides a general programming environment
- There are two main types of system software
    1. Operating System
    2. Utility Software

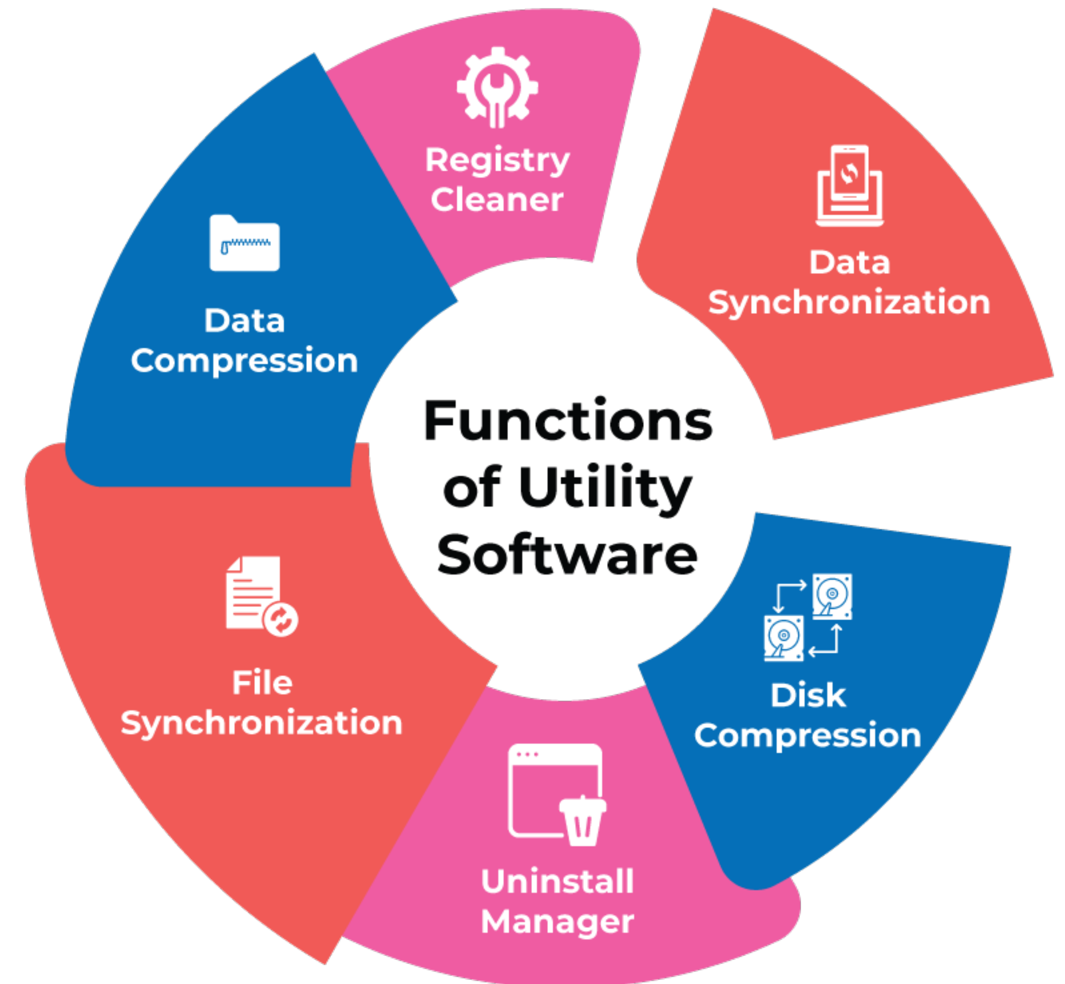
Operating System (OS)

- Provides functions used by the **application software**
- Provides the mechanisms for application software to **share** the hardware in an orderly fashion to:
  - **increase the overall performance** by allowing different application software to use different parts of the computer at the same time
  - **decrease the time to execute** a collection of programs and **increase overall system performance**
- Interacts directly with the hardware to provide an interface to other system software and with application software whenever it wants to use system's resources
  - It is application-domain independent
  - Provides resource abstraction
  - Provides resources sharing (through strict resource management policies)

Utility Software

- Utility software is system software designed to help **analyse, configure, optimize** or **maintain** a computer.

- Examples of this include:
  - data compression
  - disk cleaners
  - disk defragmentation
  - registry cleaners
  - system monitors



Resource Abstraction

- It is done by providing an **abstract model of the operation** of the hardware components
  - Different hardware components that a program may access are referred to as **resources**.
  - Any particular resource, such as a hard-disk has a generic interface that defines how the programmer can make the resource perform a desired operation.
- Abstraction generalises the hardware behaviour but restricting the flexibility
  - With abstraction, certain operations become **easy to perform**, other may become **impossible** (such as specific hardware control)
- An abstraction can be made to be much simpler than the actual resource interface
  - Similar resources can be abstracted to a common abstract resource interface
  - E.g., system software may abstract hard-disks and CD-ROMs into a single abstract disk interface

Resource Sharing

- Abstract and physical resources may be shared among a set of concurrently executing programs:

- **Space Multiplex Sharing**

  Resource can be divided in two or more ***distinct units*** of the resource that can be used independently.

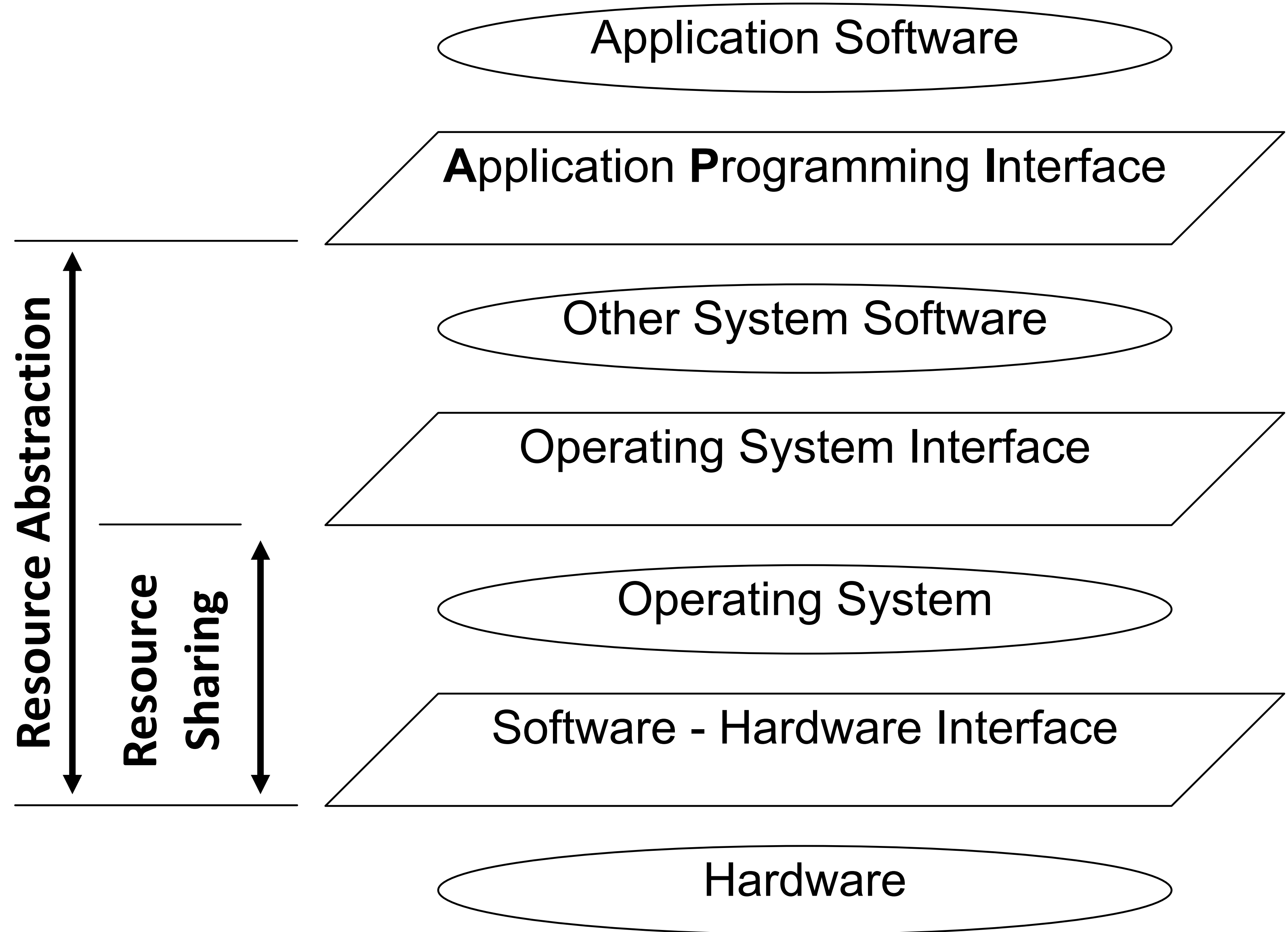
  E.g.: Memory, HDD

- **Time Multiplex Sharing**

  A process is allocated exclusive control of the entire resource for a short period of time (***not spatially divisible***)

  E.g.: Processor Resource

System Software and the OS



OS Organisation

Process and resource manager

- It uses the abstractions provided by the other managers
- Handles resource allocation

Memory manager

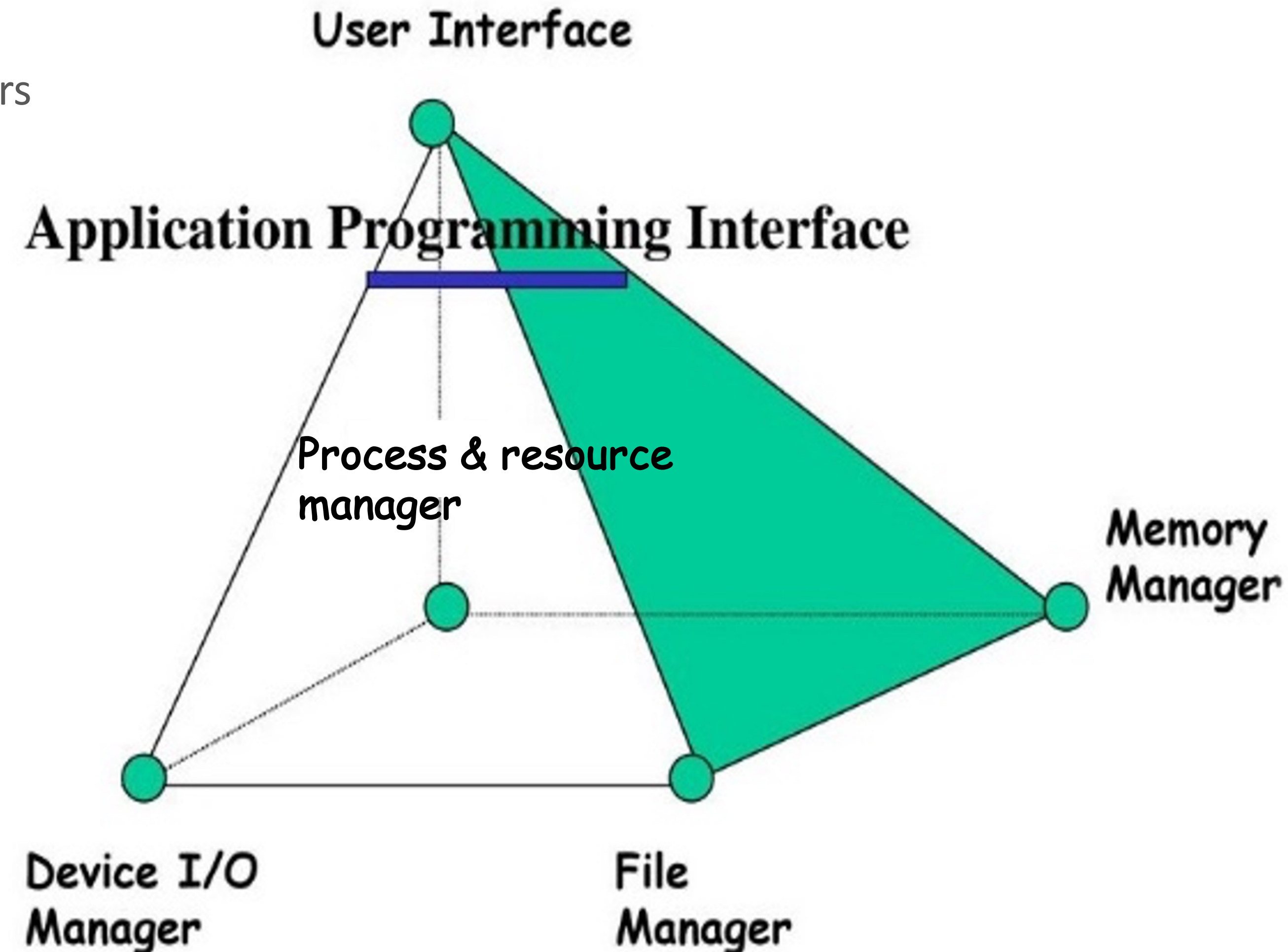
- It is classically a separate part of the operating system
- Beside other functions, it is in charge with the implementation of the virtual memory

File manager

- abstracts device I/O operations into a relatively simple operation

Device manager

- handles the details of reading and writing the physical devices
- implemented within device driver



OS Design – Functional Requirements

Processes:
- Creation, termination, control, exception handling
- Protection
- Synchronisation and communication
- Resources allocation/de-allocation

File System Management:
- Space allocation/de-allocation
- Protection, sharing, security
- Physical resource abstraction

Memory management:
- Allocation/de-allocation
- Protection and sharing

I/O devices:
- Allocation/de-allocation
- Protection and sharing
- Physical resource abstraction

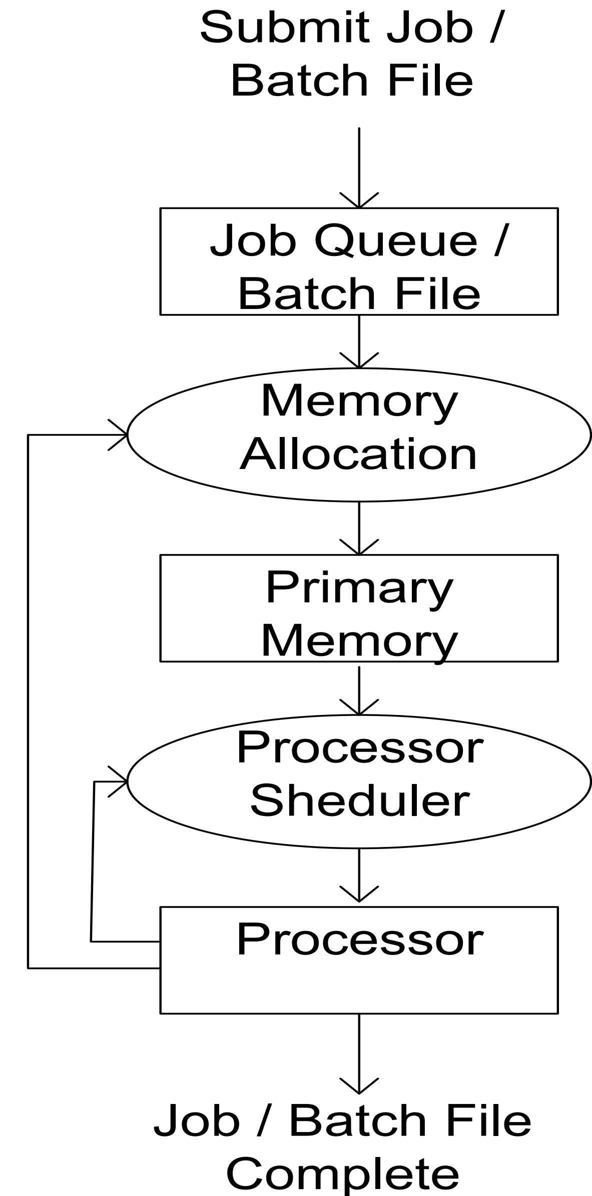
Operating Systems Evolution

Operating Systems Evolution

- **Computers with no operating system**
  - Programming in machine language
  - Lack of I/O devices
- **Rudimentary OS**
  - Programming done in assembly
  - Some basic I/O devices
  - Some I/O control modules, assembler, debugger, loader, linker
- **Batch processing systems** – service a collection of jobs, called a batch, from a queue
  - Job – predefined sequence of commands, programs and data combined into a single unit
  - Job Control Language and monitor batch (interpreter for JCL)
  - The user doesn't interact with programs while they operate

Batch Systems

- Processor scheduling : FIFO
- Memory management:
  - Memory is divided in two parts: system memory and program memory (for programs)
- I/O management – no special problems, since a job has exclusive access to the I/O devices
- File management – present

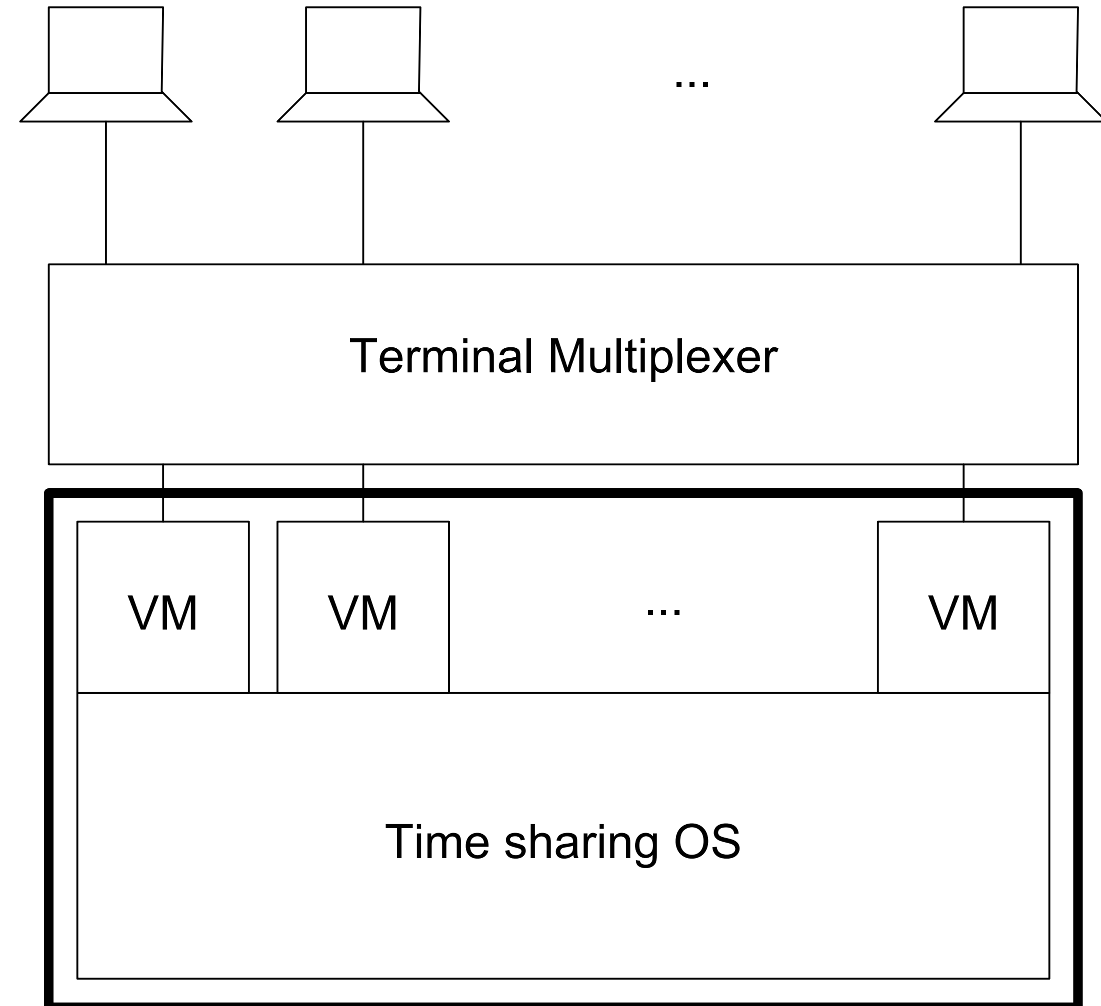


Operating Systems Evolution

- Operating systems using **multiprogramming:** the technique of loading multiple programs into space multiplexed memory while time-multiplexing the processor
  - Timesharing Systems
  - Real-time Operating Systems
  - Distributed Operating Systems
- Multiprogramming systems **common features**
  - Multitasking: multiple processes sharing machine resources
  - Hardware support for memory protection and I/O devices
  - Multi-user and multi-access support (through time sharing mechanisms)
  - Optional support for real time operations (based on efficient usage of multitasking support)
  - Interactive user interface

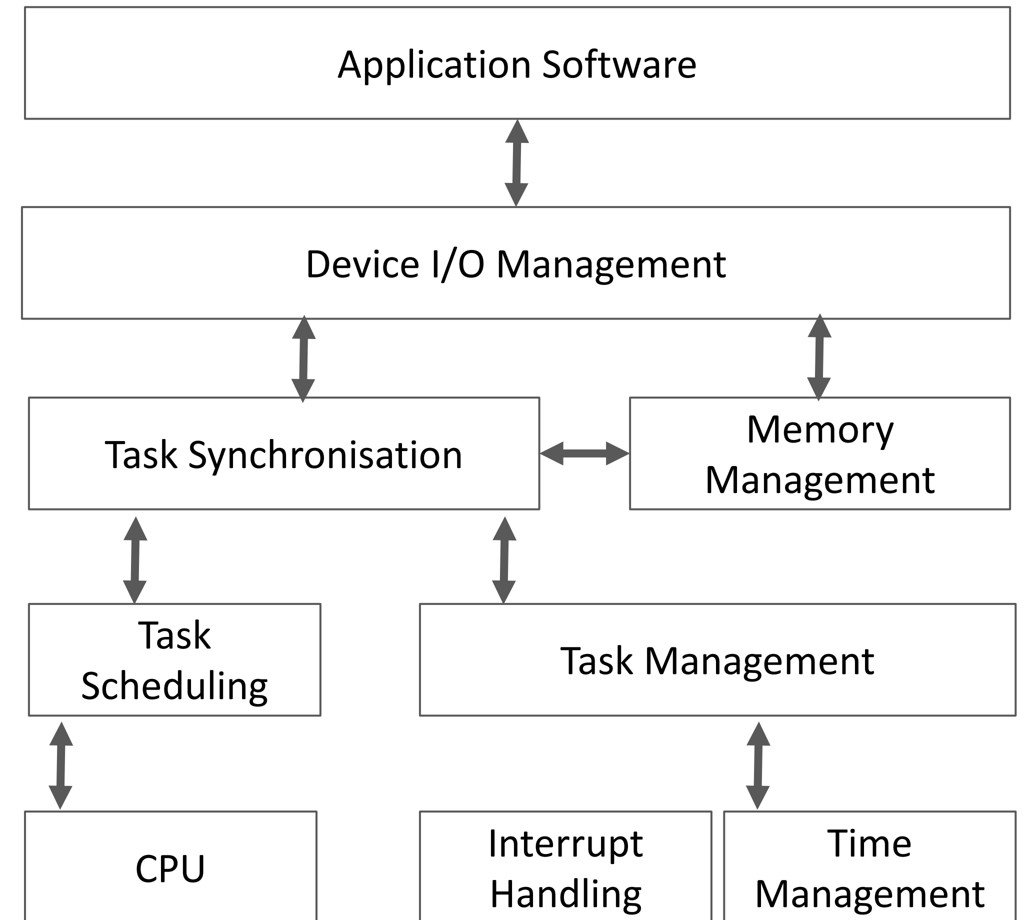
Time Sharing Systems

- Support for **multiprogramming** and **multi-user**
- Processor scheduling
  - Time slice (round robin)
- Memory management:
  - Protection and inter-process communication support
- I/O management
  - Support for protection and sharing between users
- File management
  - Protection support and sharing support between users



Real Time Operating Systems

- Used whenever a large number of **critical external events** have to be treated in a short or **limited interval of time**
- Support for multiprogramming/multi-tasking
- Main goal
  - **Minimisation of the response time** to service the external events



Real Time Operating Systems

Processor scheduling:

- Priority based preemptive

Memory management:

- Concurrent processes are loaded into the memory
- Support for protection and inter-process communication

I/O management:

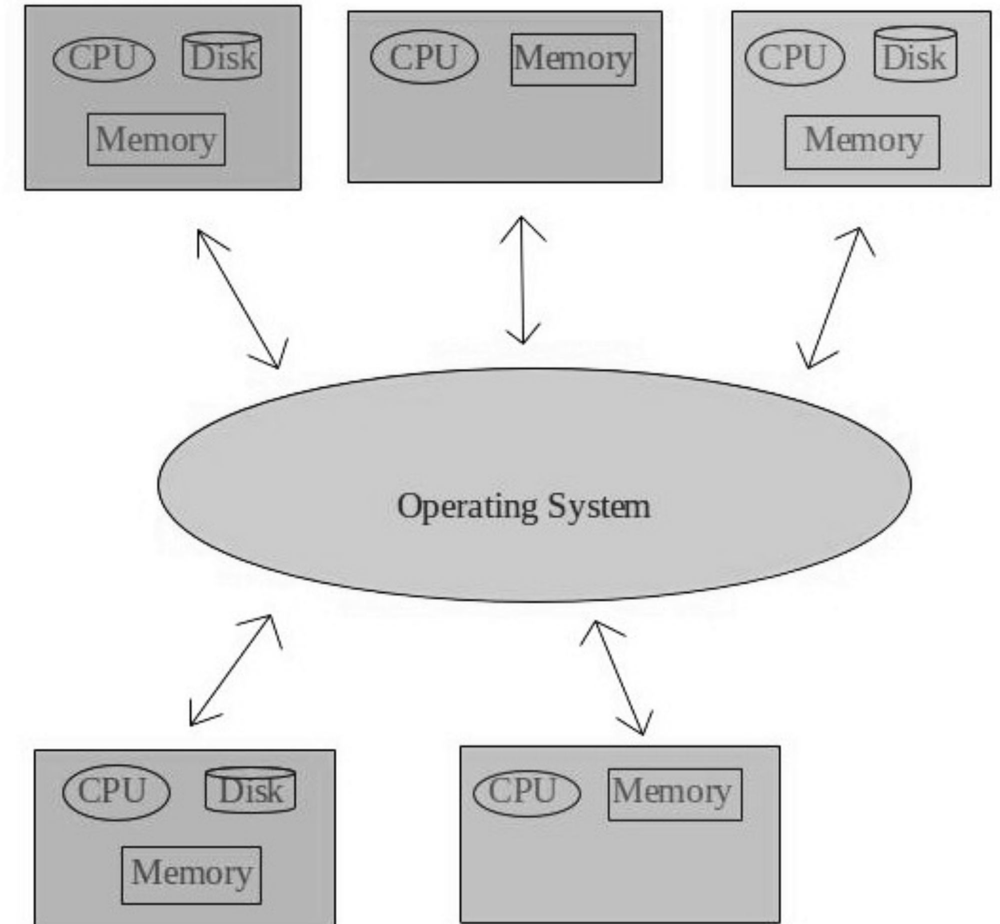
- Critical in time
- Processes dealing with I/O are directly connected to the interrupt vectors (for handling the interrupt requests)

File management:

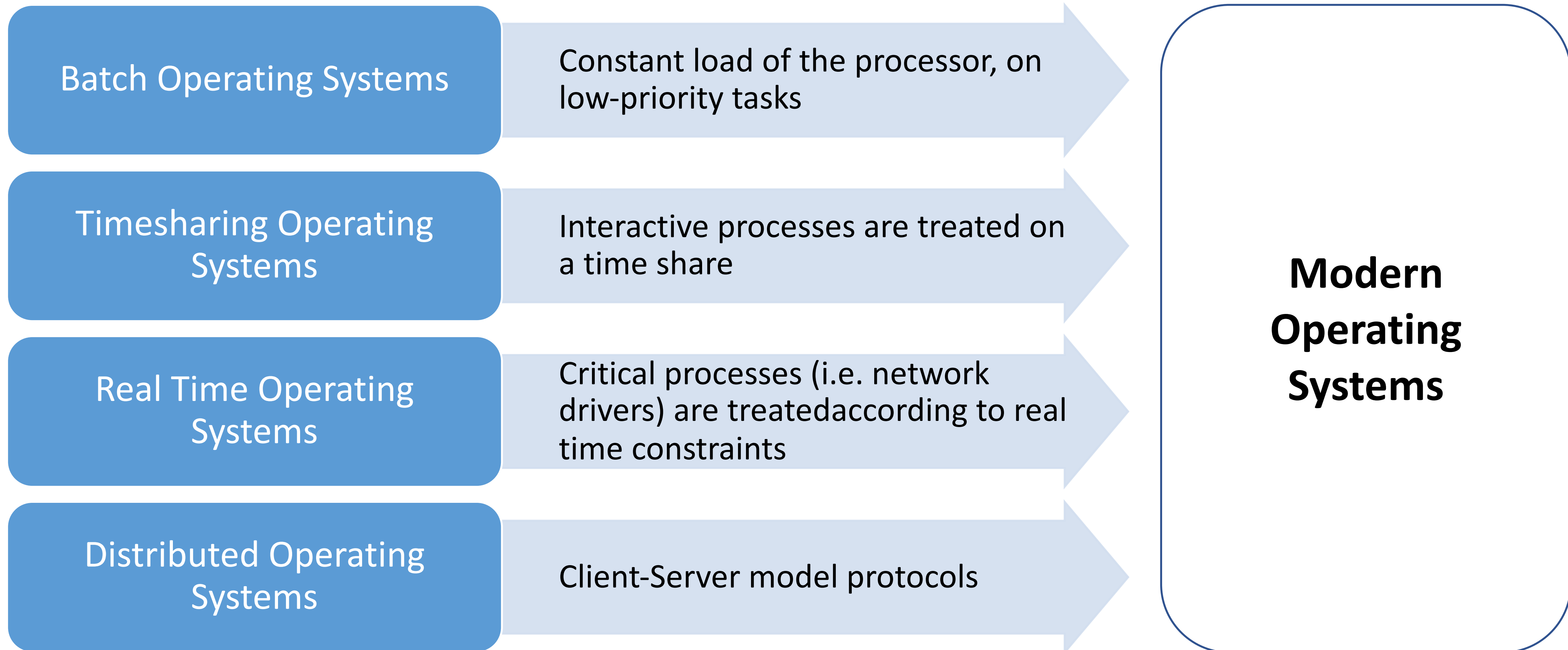
- It may be missing
- If it exists, it should comply with requirements for timesharing systems and it should satisfy the requirements for real time systems

Distributed Operating Systems

- Multiprogramming induces a **strong centralisation** tendency
- Distributed OS aims for **decentralisation**
- Based on **computer network technologies**, with different communication and synchronization protocols
- **Client-server** application architecture
- **Security** and **protection** are the primary concerns



Modern Operating Systems



OS Implementation Considerations

OS Implementations

- **Monolithic Operating System**
  - Try to achieve the functional requirements by executing all the code in the same address space to increase the performance of the system
  - Too complex to manage

- **Hierarchical Operating System**
  - Run most of their services in user space, aiming to improve maintainability and modularity of the codebase
  - Suitable for Object Oriented Programming, the levels are very well defined

Implementation Considerations

- **Multi-programming**: the illusion that multiple programs are running simultaneously
- **Protection**: access to shared system resources
- **Processor modes:** different privilege levels
  - restrictions on operations that can be run
- **Kernels:** complete control over everything in the system (i.e., supervisor)

Multiprogramming (1)

- Technique that allows the system to present the illusion that multiple programs are running on the computer simultaneously

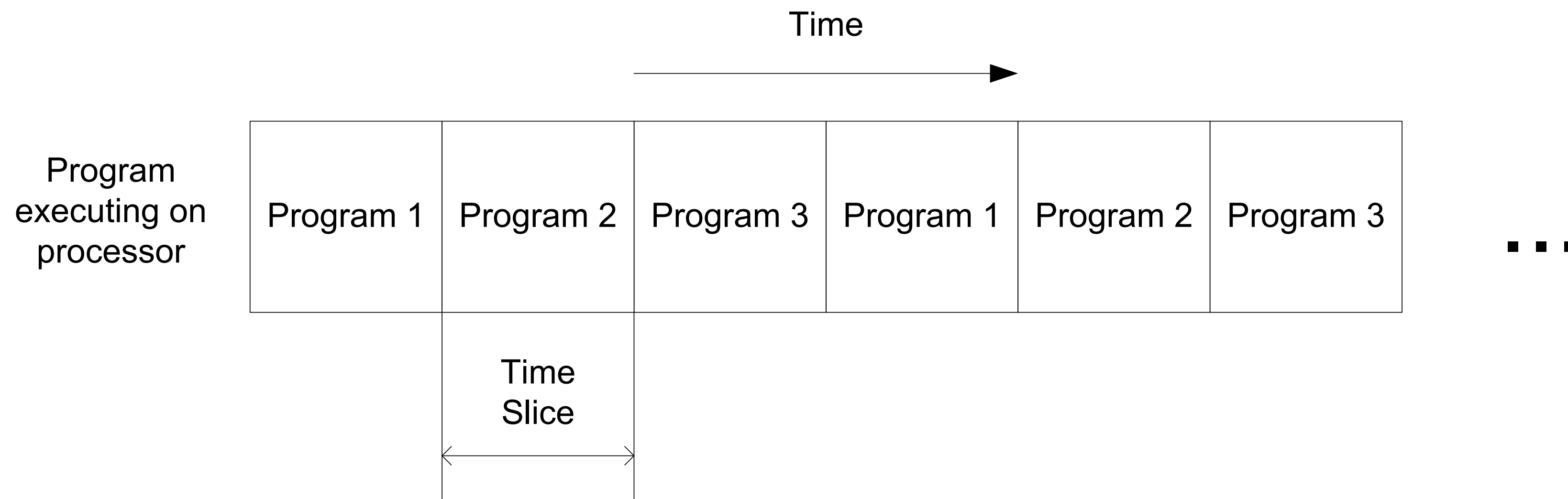
- ***Protection*** between programs is very important
  - Many multiprogrammed computers are ***multiuser***
  - Allow multiple persons to be logged on at a time

- Beside protection, data ***privacy*** is also important

- Multiprogramming is achieved by **switching rapidly between programs**.
  - Each program is allowed to execute for a fixed amount of time – ***timeslice***

Multiprogramming (2)

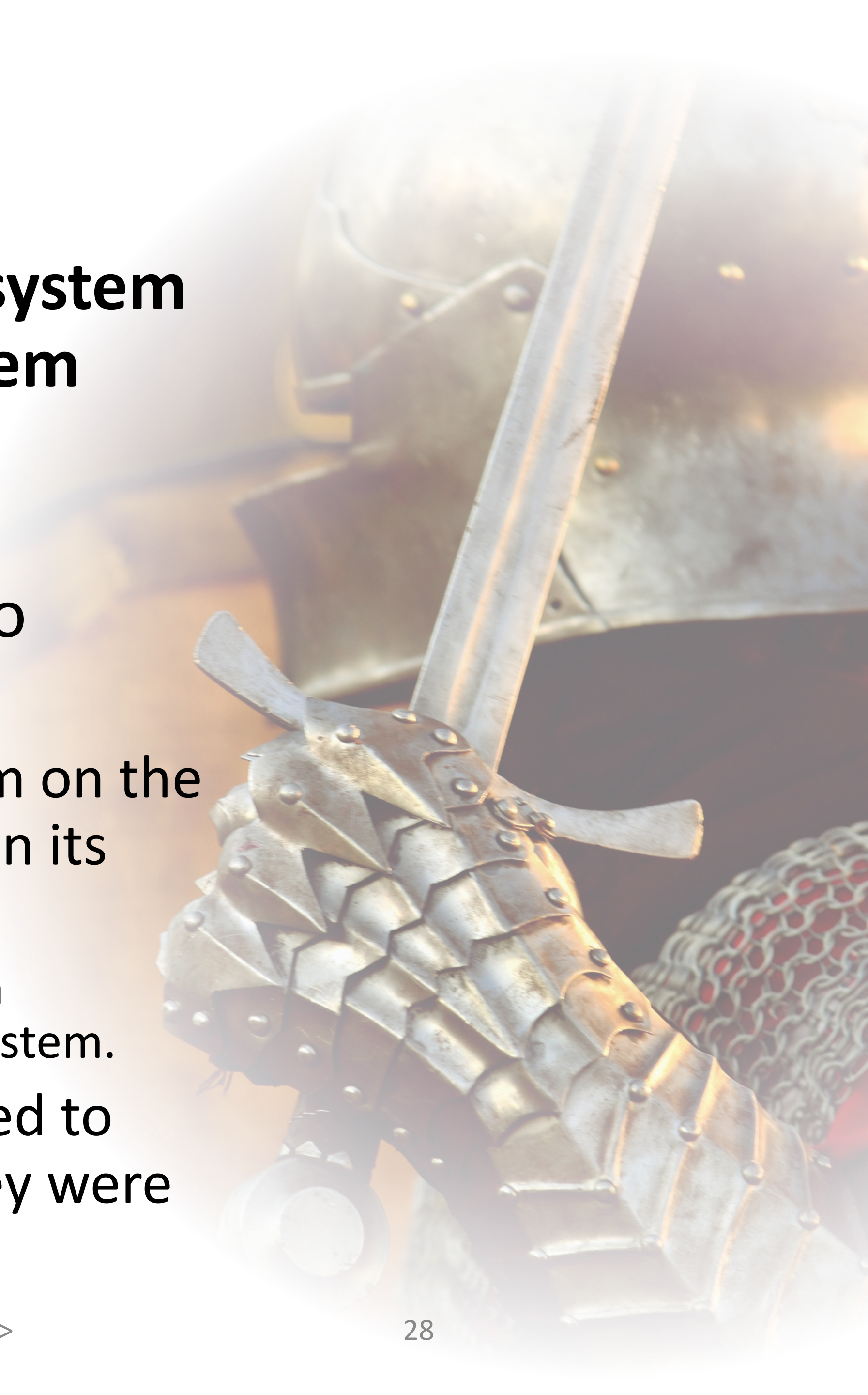
- When a program timeslice ends, the OS stops it, removes it and gives another program control over the processor – this is a ***context switch***
  - To do a context switch the OS copies the content of current program register file into memory, restores the contents of the next program's register file into the processor and starts executing the next program.
  - From the program point of view, they can't tell that a context switch has been performed



Protection (1)

- The result of any program running on a multiprogrammed computer must be the same as if the program was the only program running on the computer
- Programs must not be able to access other program's data and must be confident that their data will not be modified by other programs.
- Programs must not interfere with other program's use of I/O devices

Protection (2)



- Protection is achieved by having the **operating system have full control over the resources of the system** (processor, memory and I/O devices)

- ***Virtual memory*** is one of the techniques used to achieve protection between programs
  - Each program operates as if it were the only program on the computer, occupying a full set of the address space in its virtual space.
    - The OS is ***translating*** memory addresses that the program references into physical addresses used by the memory system.
  - As long as two program's addresses are not translated to same address space, programs can be written as they were the only ones running on the machine

Privileged Mode

- To ensure that the OS is the only one that can control the physical resources it executes in **privileged mode**
- OS is also responsible for **low level UI**
  - Keys are pressed, the OS is responsible to determine which program should receive the input
  - When a program wants to display some output, the user program executes some system call that displays the data
- User programs execute in **user mode**
  - When user mode programs want to execute something that requires privileged rights, it sends a request to the OS, known as **system call**, that asks the OS to do the operation for them

Processor Modes (1)

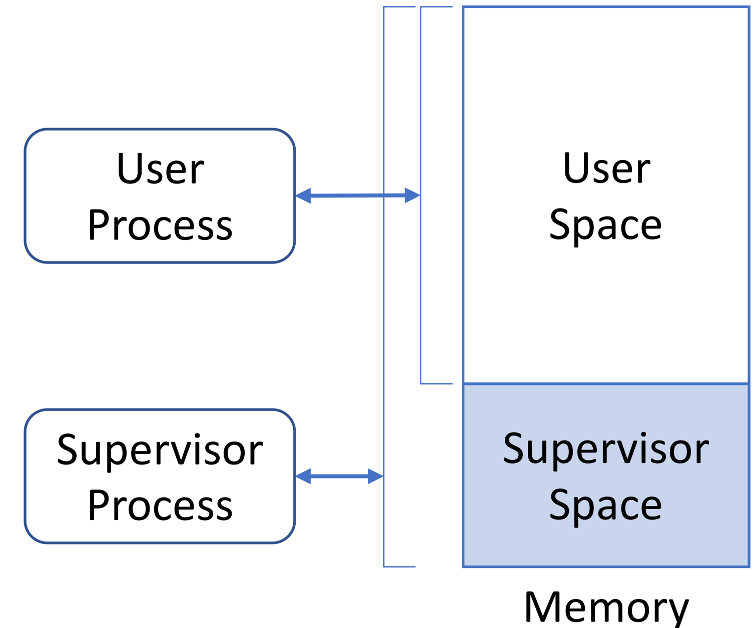
- Processor Modes are operating modes for the CPU that **place restrictions** on the operations that can be performed by the currently running process

- Hardware supported CPU modes help the operating system to **enforce rules** that would prevent viruses, spyware, and/or similar malware to run
  - Only very specific and limited “kernel” code would run unrestricted.
  - Any other software (including portions of the operating system) would run restricted and would have to ask the “kernel” for permission to modify anything that could compromise the system.

- Multiple mode levels could be designed.

Processor Modes (2)

- Mode bit to define **execution capability** of program on a processor
- *Supervisor* mode
  - The processor can execute any instruction
  - Instructions that can be executed only in supervisor mode are called *supervisor, privileged* or *protected* instructions (e.g., I/O instructions)
  - Execution process has access on both memory spaces
- *User* mode
  - The processor can execute a subset of the instruction set
  - Executing process has access only to the user space
- Some microprocessors do not make a difference between protected and user mode
- The mode bit may be logically extended to define areas of memory to be used when the processor is in supervisor mode versus when it is in user mode



Kernel

- The part of the operating system that executes in supervisor mode is called **kernel** or **nucleus**

- Operates as **trusted** software
  - Implements protection mechanisms that could not be changed through the actions of un-trusted software executing in user mode
  - Provides the lowest level abstraction layer for resources (memory, processors and IO devices)

- Fundamental design decision: should a given function of the OS be incorporated in the kernel or not?
  - Protection issues
  - Performance issues

Methods for Requesting System Services

- Through command line interface
  - By calling a specific command
  - Using a command interpreter known as a shell

- From user processes requesting services from OS:
  - By calling a system function
  - By sending a message to a system process

```
takfarinas — -bash — 80×24
(base) Takfarinass-MacBook-Pro:~ takfarinas$ ls
Applications                    Music
Cisco Packet Tracer 8.0.0       ObeoDesignerWorkspace
Desktop                         Pictures
Documents                       Public
Downloads                       Zotero
Dropbox                         eclipse
Google Drive                    eclipse-workspace
Lecture 1_.pptx                 git
Library                         iCloud Drive (Archive)
Movies                          nltk_data
(base) Takfarinass-MacBook-Pro:~ takfarinas$
```

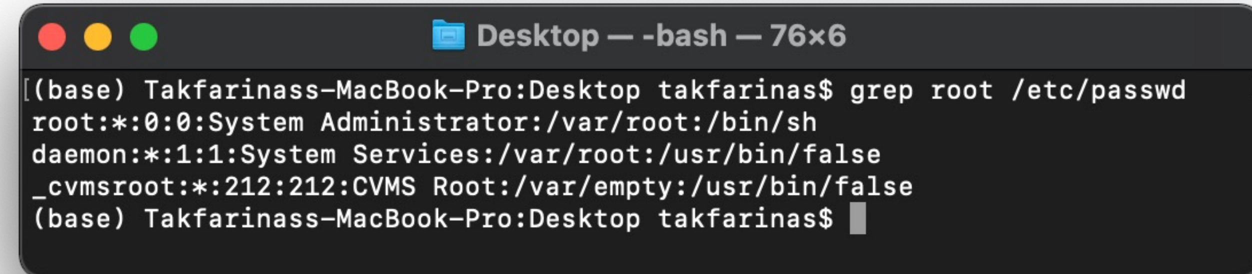
Command execution mechanism

- A key pressed by the user generates a hardware interrupt
- A specialised module of the OS reads the keyed character and then stores it in a special command line buffer
  - There are special characters (i.e., to edit the command line, that are not stored in the command line buffer)
- End of line detected: control taken by the command interpreter (shell):
  - Analysis of the command (with error or success)
  - If success, then the command interpreter decides if it is about an internal or external command (for another module)
    - If internal command: tries the execution that can end successfully or with error
    - If external: looks for the corresponding executable file and executes it with the detected parameters from previous phase

Command execution example

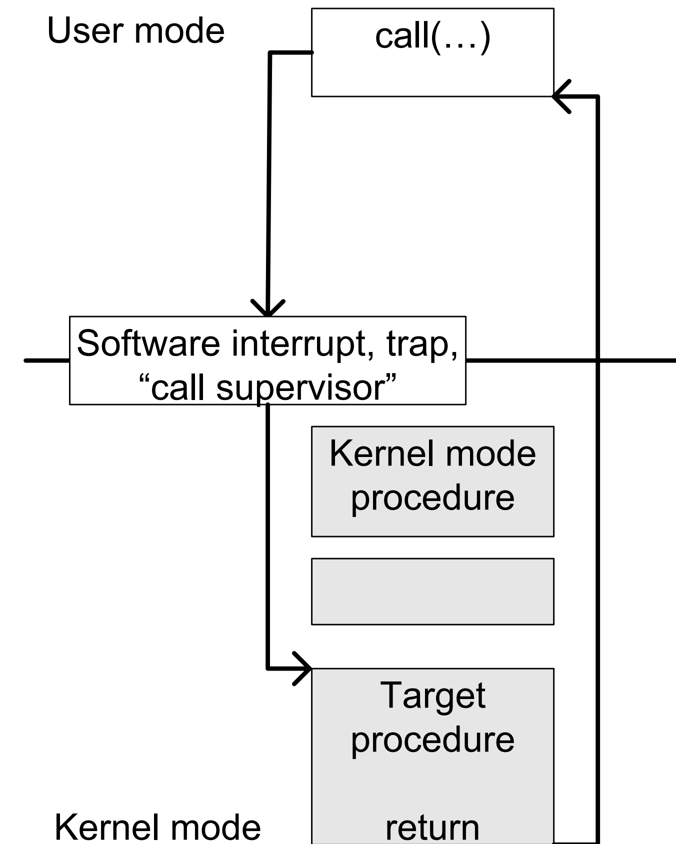
- Semantics of ***grep*** establish that the first string parameter (first) represents the search target, while the second parameter represents a file name (where to search)

```
>$ grep mouse mouse.txt
```



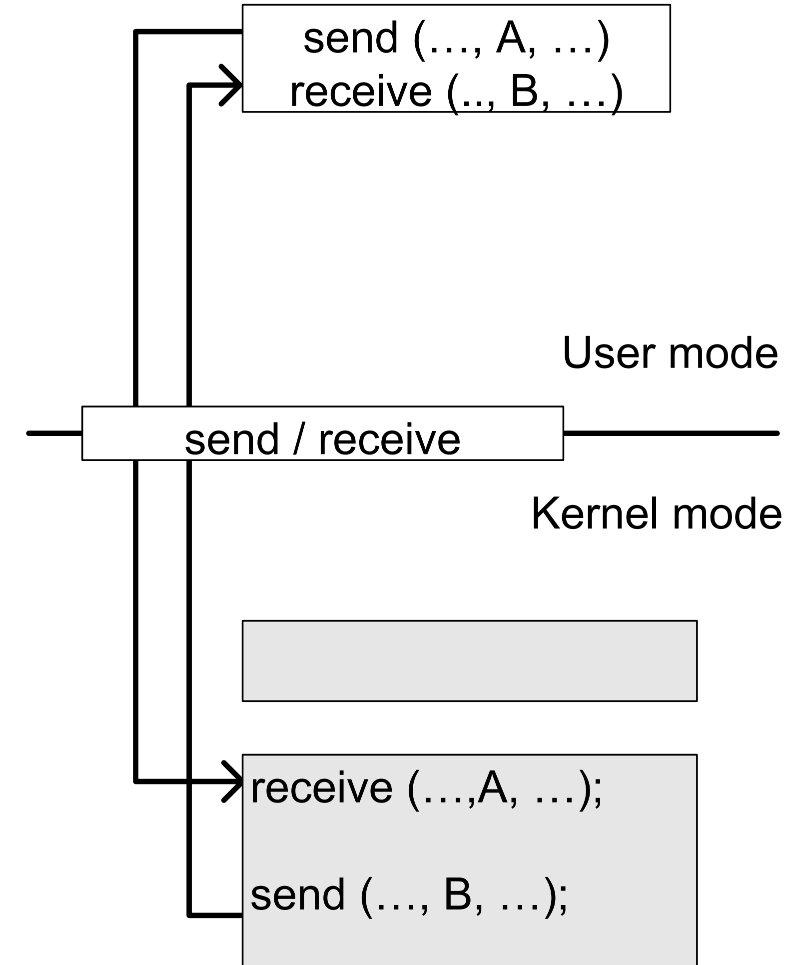
System Call

- The parameters of the call are passed according to the **specific OS convention** and hardware architecture
- Switch to **protected (supervisor) mode** using a specific mechanism
  - E.g., software interrupt, trap, special instruction of type "call supervisor"
  - mechanism that is different from a normal call
- A **special module** takes over, that will analyse the parameters and the access rights
  - This module can reject the system call
- If accepted: the **corresponding routine** from the OS is executed and the **result** is returned to the user
  - upon return, the user mode is restored



Messages

- User process **constructs a message** that describes a desired service (A)
- Uses the **send** function to pass the message to a trusted operating system process
  - The send function checks the message
  - switches the processor to **protected mode**
  - and then delivers the message to the process that implements the target function
- Meanwhile, the user waits for result with a **message receive** operation.
- When the **kernel finishes processing the request**, it sends a message (B) back to the user process



References

- “Operating Systems – A modern perspective”, Garry Nutt, ISBN 0-8053-1295-1

- Funny video: https://youtu.be/aJFwVOW0Nww

Ollscoil na Gaillimhe
University of Galway

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing Systems & Organisation

Process Management

Content

1. Process Manager Process – User perspective
2. Process – Operating System perspective
3. Threads
4. Operating System services for process management

Program and Process

- **Program**: static entity made up of source program language statements that define process behavior when executed on a set of data

- **Process**: dynamic entity that executes a program on a particular set of data using resources allocated by the system
    - two or more processes could execute the same program, each using its own data and resources

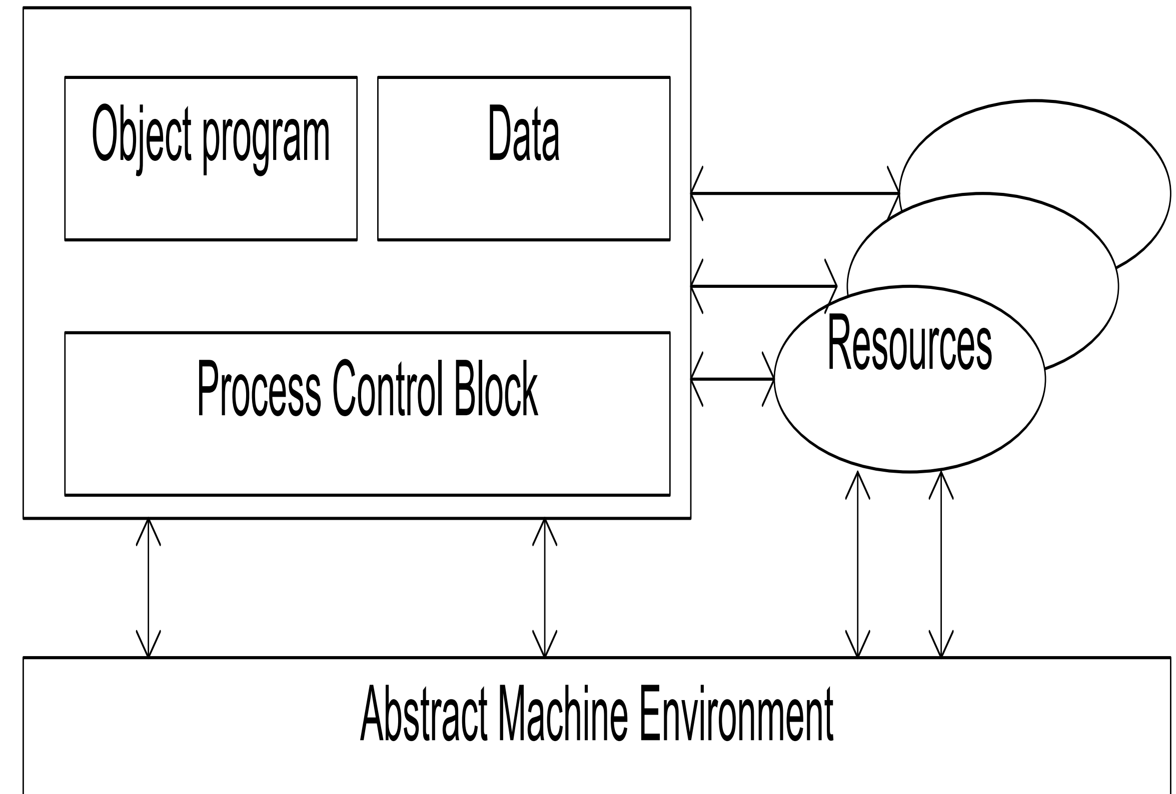
What is a Process?

- A process is a program in execution

- It is composed of:
  - Program
  - Data
  - Process Control Block (PCB): contains the state of the process in execution
    - What it is?
    - How much of its processing has been completed?
    - Etc.

Execution of a Process

- In order to execute, a process needs an **Abstract Machine Environment** to manage its use of resources
- Process Control Block (PCB) is required to map the environment state on the physical machine state
- The OS keeps a *process descriptor* for each process



Program Execution

- Each execution of the program generates a process that is executed

- Inter-process relationships:
  - **Competition** – processes are trying to get access to different resources of the system, therefore a protection between processes is necessary
  - **Cooperation** – sometime the processes need to communicate between themselves and exchange information – synchronization is needed

Process Manager Functions

- Implements:
  1. CPU sharing (called *scheduling*)
     - allocate resources to processes in conformance with certain policies
  2. Process synchronization and inter-process communication
     - deadlock strategies and protection mechanisms

Process Manager

Program

Process

The Abstract Computing Environment

File Manager

Device Manger

Memory Manager

Devices

Memory

Protection

Process Descriptor

Deadlock

Synchronizaton

Scheduler

Resource Manager

CPU

Resources

Process Manager

Process – User Perspective

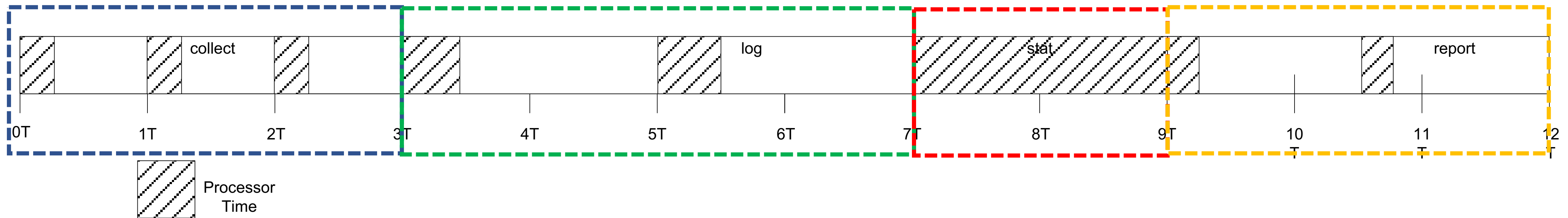
Example:

Consider an application that monitors an industrial process, to record its operation.

- The application contains 4 program modules:
  - **Data acquisition (*collect*):** Reads 3 values from a converter
    - Collecting each data is 1 T interval: ¼ T processor time, ¾ T is wait time to finish read op from converter
  - **Data storage (*log*):** Writes on the disk the 3 values read by *collect*:
    - It needs two operations (writing two values at a time, including "newline:) which take 2 T intervals each: 2/4 T processor time and 6/4 T wait time to finish the write operation
  - **Statistical processing (*stat*):** Statistical processing of the three values collected by *collect*, needs 2T
  - **Print results (*report*):** Prints two values resulted from statistical processing (*stat*)
    - Each print operation requires ¼ T processor time and 5/4 T wait time to finish print operation

Sequential Implementation

```
main(){
  while (TRUE){
      collect();
      log();
      stat();
      report();
       }
   }
```



- The time required for a cycle is 12T:
  - 4.25T is required for processing time (processor time)
  - 7.75T is wait time between various I/O operations

Multitasking Implementation

- The following processes will be executed in a quasi-parallel fashion, with the following priorities:
  - Log
  - Collect
  - Report
  - Stat

- For correct functionality, the processes need to synchronize to each other; this will be done with directives wait/signal:
  - **Wait** – wait for a signal from a specific process
  - **Signal** – send a signal to a specific process

```
void log(){
   while(TRUE){
       wait(collect);
       log_disk();
       signal (collect);
   }
}
```

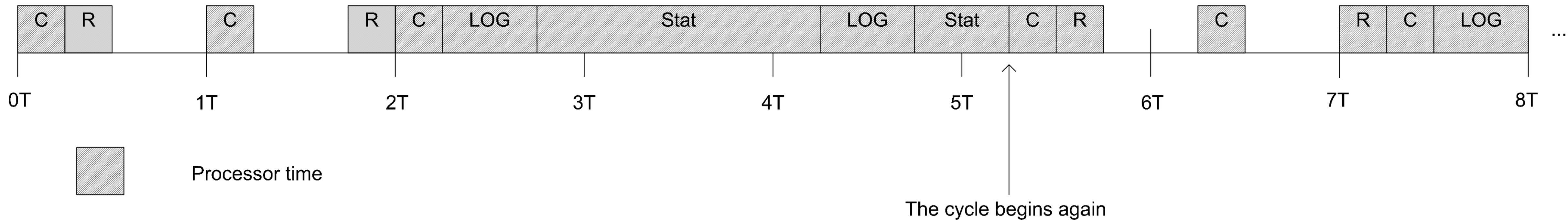
```
void report(){
   while (TRUE){
       wait (stat);
       report_pr ();
       signal (stat);
   }
}
```

```
void collect(){
   while(TRUE){
      wait (log);
      wait (stat);
      collect_ad ();
      signal (log);
      signal (stat);
   }
}
```

```
void stat(){
   while (TRUE){
       wait (collect);
       wait (report);
       stat_ad ();
       signal (collect);
       signal (report)
   }
}
```

```
main(){
  init_proc(&log(), …);
  init_proc(&collect(), …);
  init_proc(&report(), …);
  init_proc(&stat(),…);

  signal (collect); signal (collect);
  signal (stat);
  start_schedule();
}
```



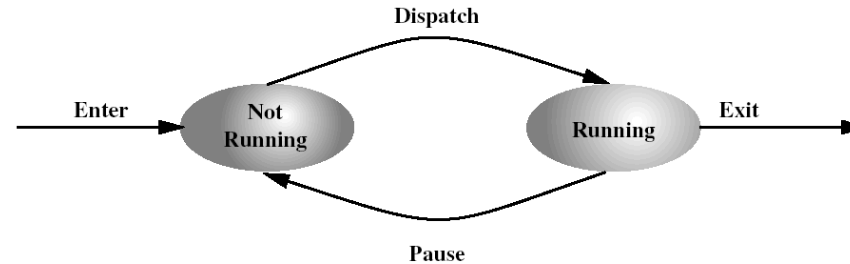
- 5.25 T for the execution of a complete cycle
- Only 1T lost in waiting time between I/O operations

Process – OS Perspective

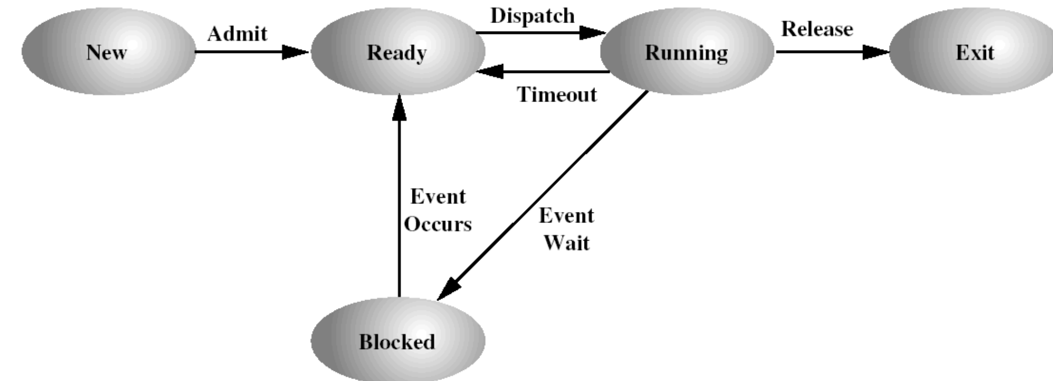
- The processor's principal function: execute machine instructions residing in main memory
  - Those instructions are provided in the form of programs
  - A processor may interleave the execution of a number of programs over time
- **Program View**
  - Its execution involves a sequence of instructions within that program
  - The behavior of individual process can be characterised by a sequence of instructions
    - ***trace*** of the process
- **Processor View**
  - Executes instructions from main memory, as dictated by changing values in the program counter (PC) register
  - The behaviour of the processor can be characterised by showing how the traces of various processes are **interleaved**

State Process Models

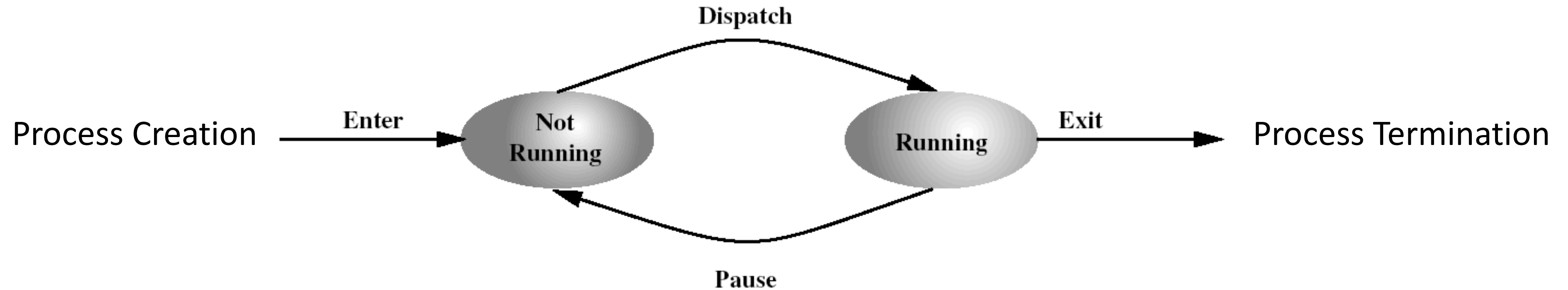
- Two State Process Model



- Five State Process Model



Two State Model



- The process can be in one of two states:
  - running or not running
- When the OS creates a new process, it enters it into the ***Not Running*** state; after that, the process exists, is known to the OS and waits for the opportunity to run
- From time to time, the currently running process will be interrupted and the dispatcher process will select a new process to run
  - The new process will be moved to **Running** state and the former one to ***Not Running*** state

Process Creation

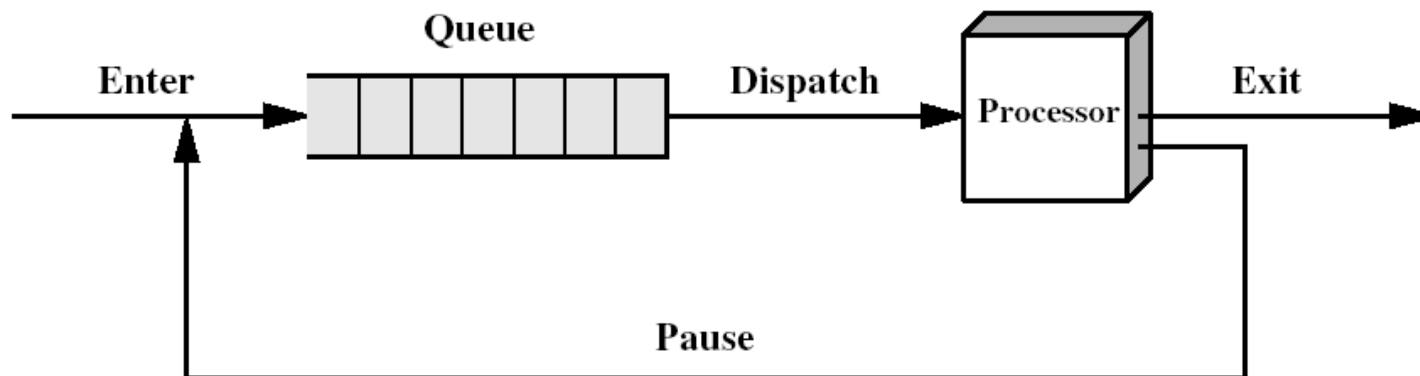
- Creation of new process:
  - The OS builds the data structures that are used to manage the process
  - The OS allocates space in main memory to the process
- Reasons for process creation:
  - New batch job
  - Interactive logon
  - Created by OS to provide a service
    - i.e., process to control printing
  - Spawned by existing process
    - i.e., to exploit parallelism

Process Termination

- Reasons for process termination
  - Process finished its execution (natural completion)
  - Total time limit exceeded
  - Errors (memory unavailable, arithmetic error, protection error, invalid instruction, privileged instruction, I/O failure, etc.)
  - Parent request
    - A parent has typically the right to request a child termination
  - Parent termination
    - When the parent terminates, the OS may automatically terminate all of its children

Queuing Discipline

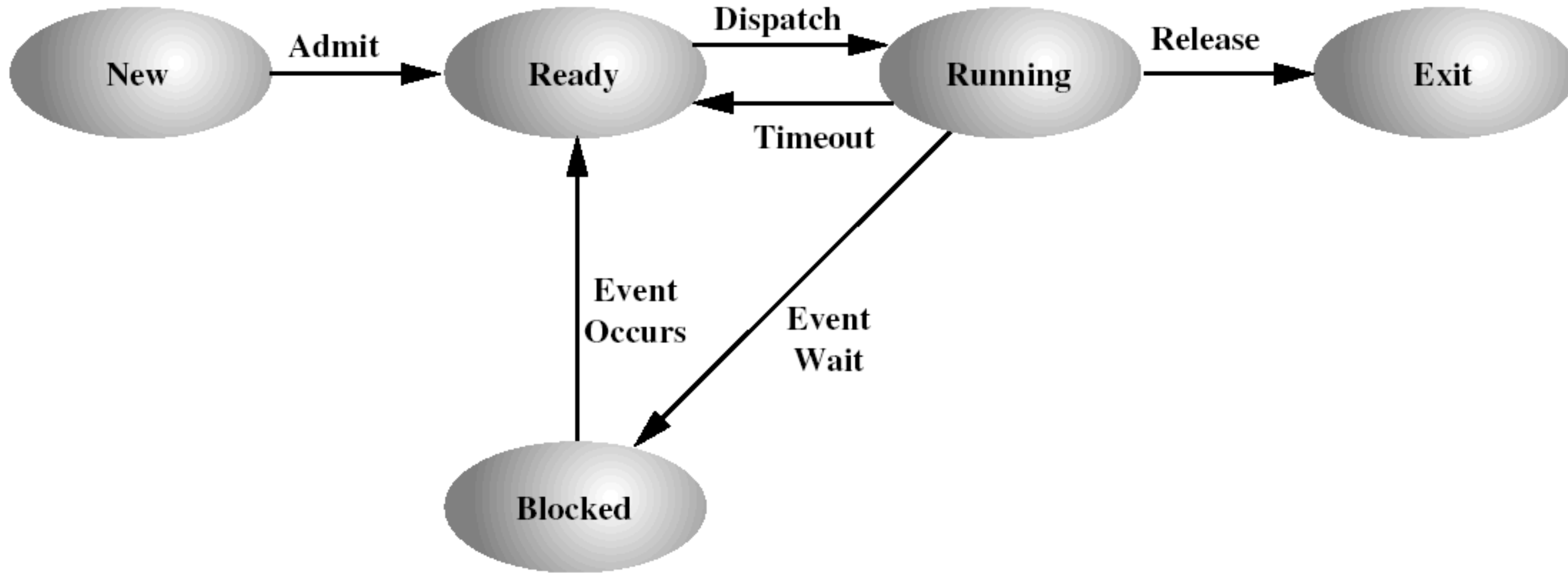
- Each process needs to be represented
  - Info relating to each process, including current state and location in memory
  - Waiting processes should be kept in some sort of queue
    - List of pointers to processes blocks
    - Linked list of data blocks; each block representing a process
- Dispatcher behavior:
  - An interrupted process is transferred in waiting queue
    - If process is completed or aborted, it is discarded
  - The dispatcher selects a process from the queue to execute



Five State Model

- ***Running*** – The process is currently being executed
  - For single processor systems, one single process can be in this state at a time
- ***Ready*** – a process that is prepared to execute when given the turn
- ***Blocked*** – a process that can't execute until some event occurs
  - Such as the completion of an I/O operation
- ***New*** – a process that has been created, but not yet accepted in the pool of executable processes by OS
  - Typically, a new process has not yet been loaded into main memory
- ***Exit*** – a process that has been released from the pool of executable processes by the OS
  - Completed or due to some errors

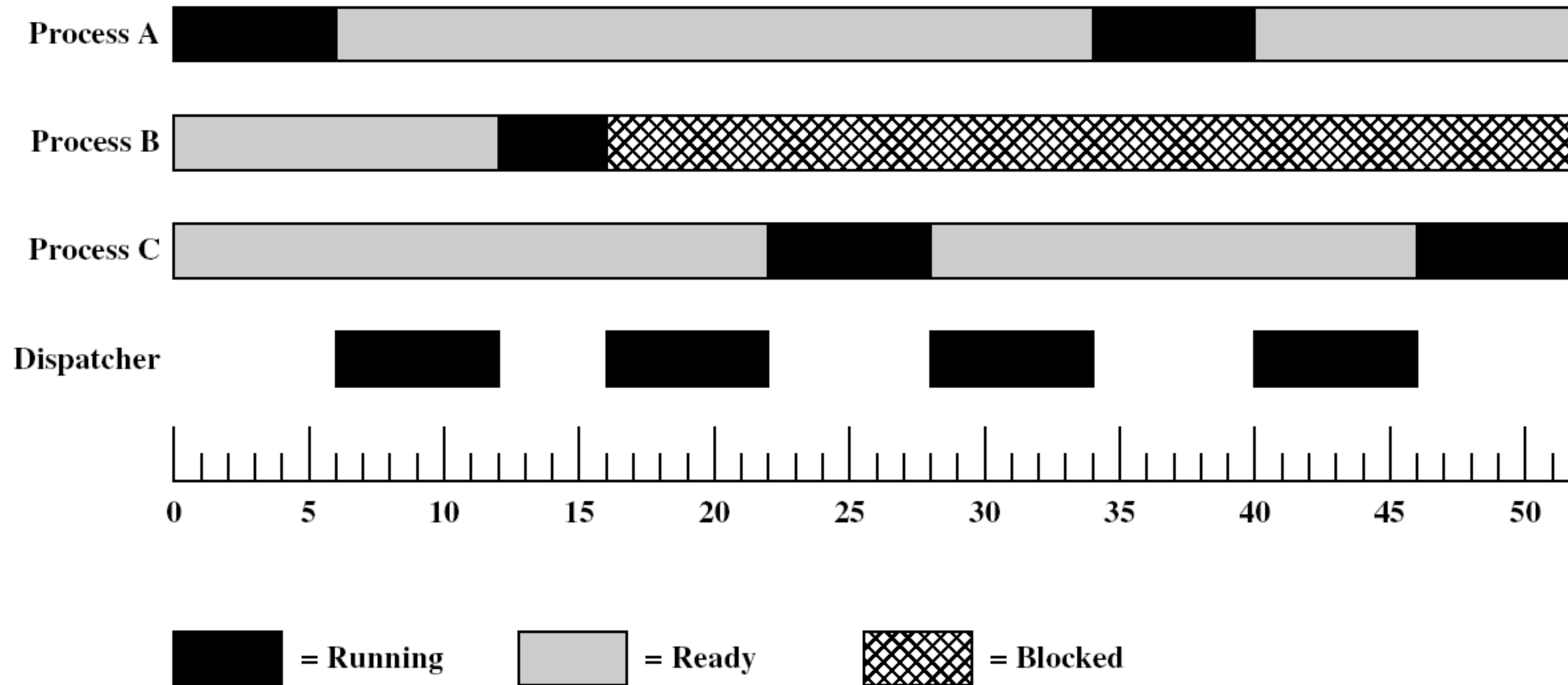
Five State Model Process Transition Diagram



Process – OS Perspective

- Consider three processes: A, B and C that are loaded in memory
- In addition, there is a small dispatcher program that switches the processor from one process to another (using Round Robin with 6 instructions)
- No use of virtual memory
- Process B invokes an I/O operation in its fourth instruction

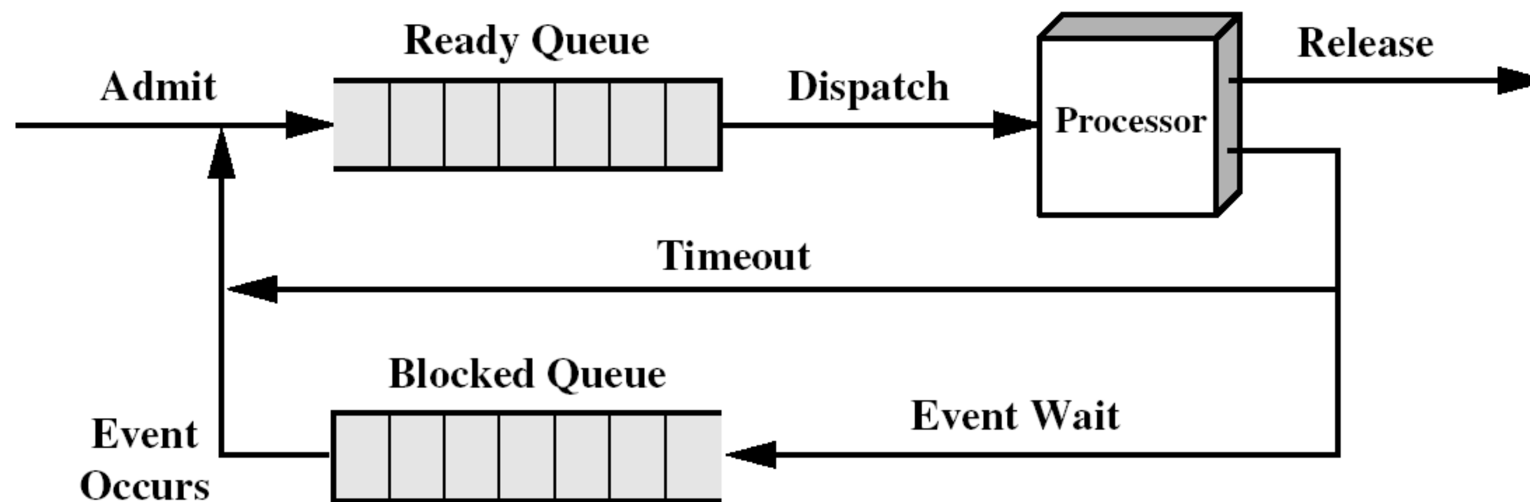
Example



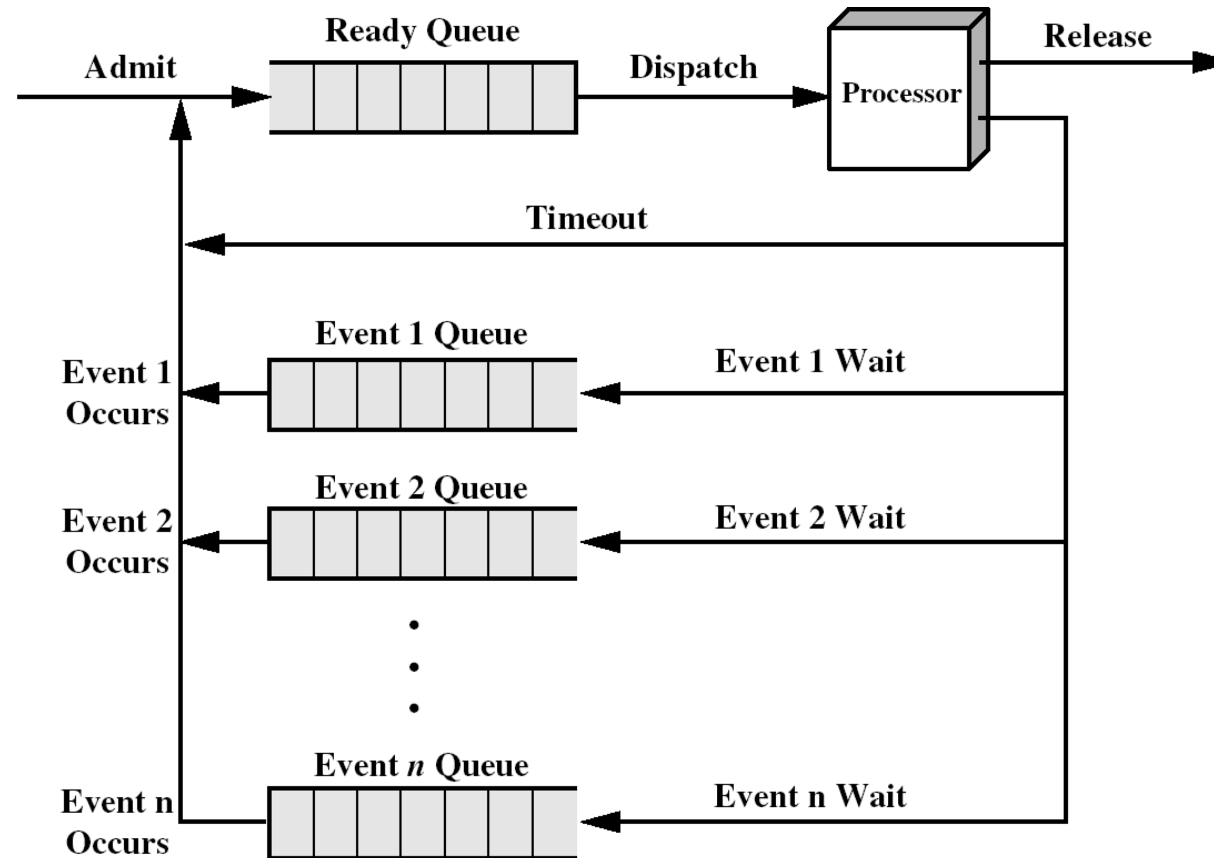
- Movement of each process described earlier (A, B and C) among the states

Queuing Discipline (1)

- There are two queues now: ready queue and blocked queue
  - When the process is admitted in the system, it is placed in ready queue
  - When a process is removed from the processor, it is either placed in ready queue or in blocked queue (depending on circumstances)
  - When an event occurs, all the processes waiting on that event are moved from blocked queue onto ready queue.

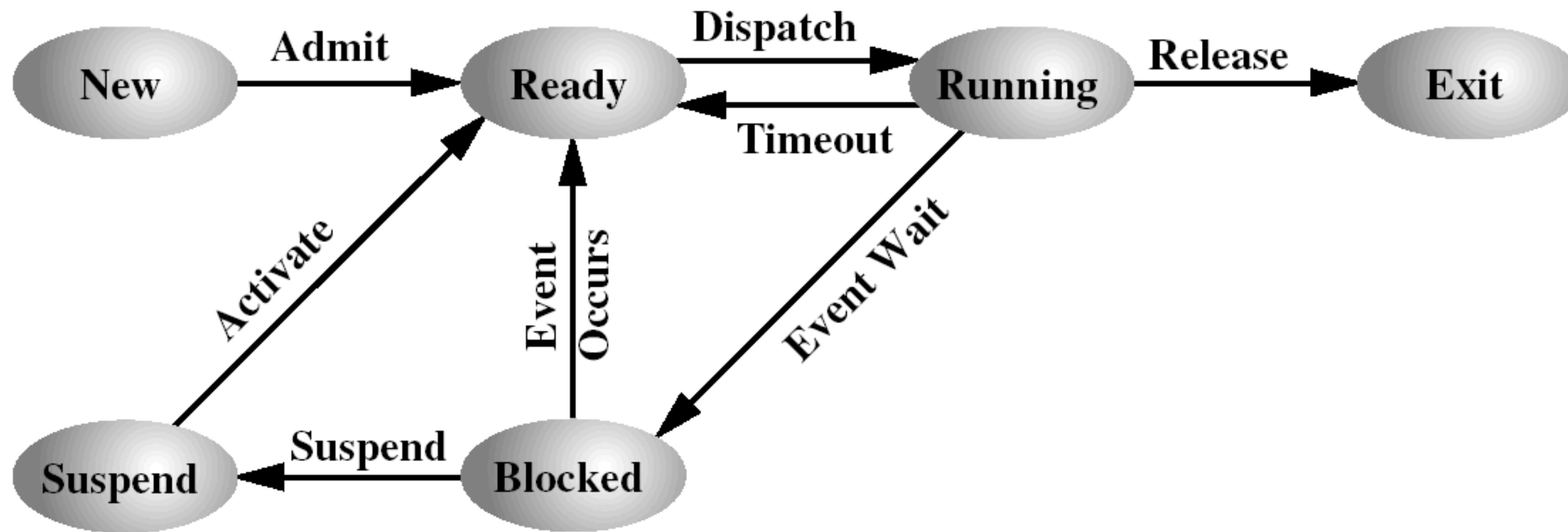


Queuing Discipline (2)



- Multiple blocked queues; one per each event
  - When event occurs, the entire list of processes is moved in ready queue

Suspended Processes



- Processor is faster than I/O so all processes could be waiting for I/O
- Swap these processes to disk to free up more memory
- Blocked state becomes suspend state when swapped to disk

Process Management Services

- *create (&process_id, attributes)*
  - Creates a new process with implicit or specified attributes
- *delete (process_id)*
  - Sometime known as destroy, terminate or exit
  - Finishes the process specified by process_id
  - Whenever the process is terminated, all the files are closed,
  - all the allocated resources are released
- *abort (process_id)*
  - Same as the delete but for abnormal termination
  - Usually generates a :post mortem dump" which contains the state of the process before the abnormal termination
- *suspend (process_id)*
  - Determines the specified process to go in suspended state

Process Management Services

- *resume (process_id)*
  - Determines the specified process to go from the suspended state in ready state

- *delay (process_id, time)*
  - Same with sleep
  - Suspends the specified process for the specified period of time
  - After the delay time elapses, the process is brought to ready state

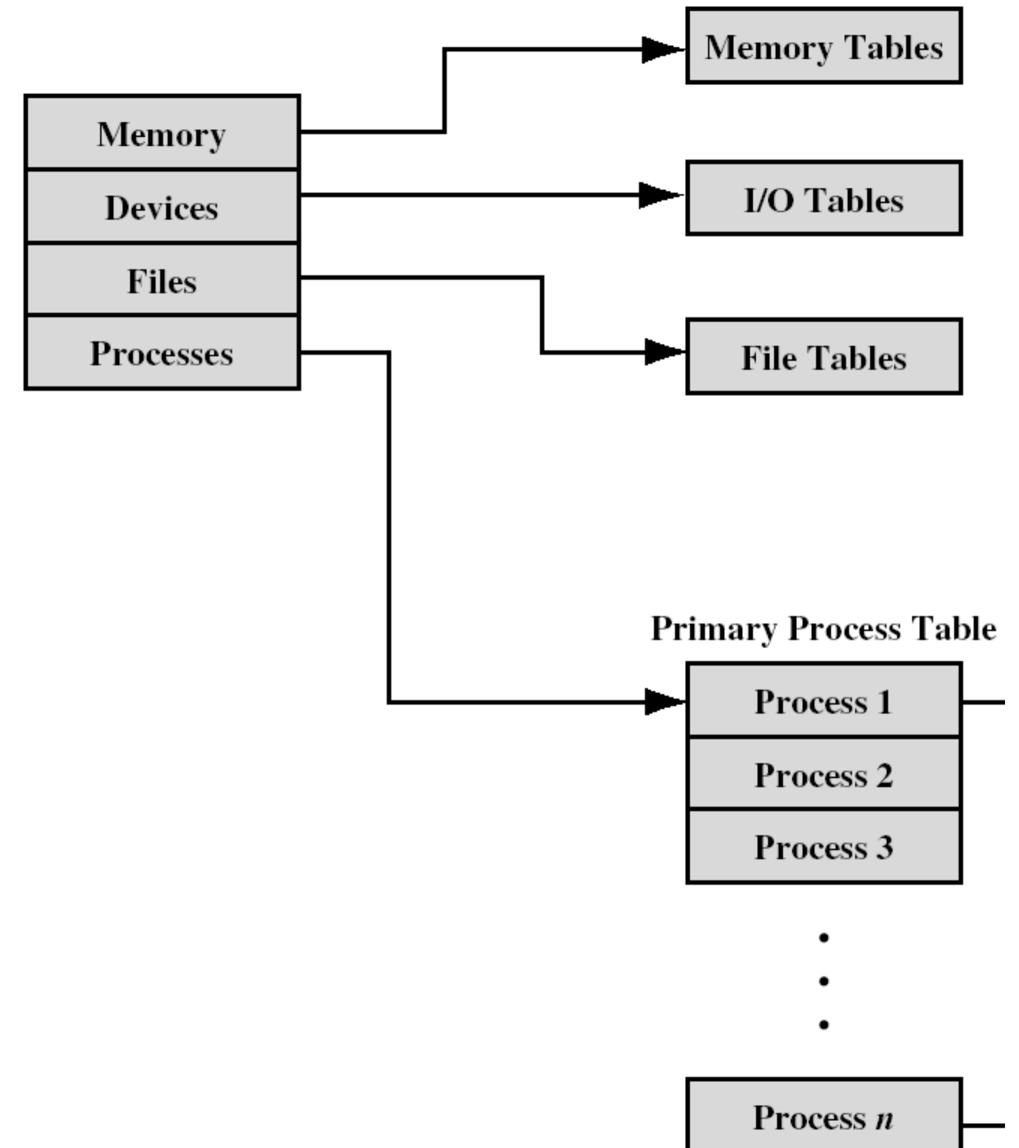
- *get_attributes (process_id, &buffer_attributes)*
  - Used to find out the attributes for the given process

- *set_attributes (process_id, buffer_attributes)*
  - Used to set the attributes of the specified process

Process Description

Process Description

- What information does the operating system need to control processes and manage resources for them?
- Operating System Control Structures
  - Memory Tables
  - I/O Tables
  - File Tables
  - Primary Process Table
  - Process Image
    - User Program, user data, stack and attributes of the process



Memory Tables

- Used to keep track of both main (real) and secondary (virtual) memory.
  - Some of main memory is reserved for use by the operating system; the remainder is available to the processes.
- Contain:
  - The allocation of main memory to processes
  - The allocation of secondary memory to processes
  - Any *protection* attributes of blocks of main or virtual memory (such as which processes can access certain shared memory regions)
  - Any information needed to manage virtual memory

I/O Tables

- Are used by the operating system to manage the I/O devices
  - At any given time, an I/O device may be available or assigned to a particular process.
  - If an I/O is in progress, the OS needs to know the status of the I/O operation and the location in main memory being used as the source or destination of the I/O transfer.

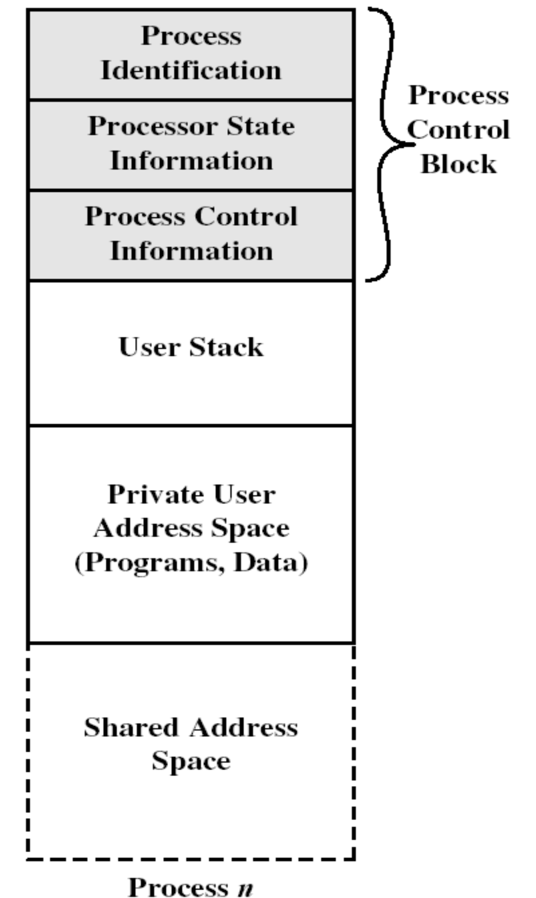
File Tables

- These tables provide information about:
  - the existence of files
  - their location on secondary memory
  - their current status
  - other attributes
- Much of this information is maintained and managed by the File Manager, in which case the process manager has little or no knowledge of files.

Process Tables

- Primary process table is used to keep one entry per each process in the operating system.
  - Each entry contains at least one pointer to a process image.

- The Process Image contains:
  - **Stack**
    - Each process has one or more stacks associated with it.
    - A stack is used to store parameters and calling addresses for procedure and system calls
  - **User Data**
    - Program data that can be modified, etc.
  - **Process Control Block**
    - Data needed by the operating system to control the process (attributes and information about process)



Process Control Block

Contains:

1. **Process Identification**: data always include a unique identifier for the process

2. **Processor State Information**: define the status of a process when it is suspended

3. **Process Control Information**: used by the OS to manage the process

Process Identification

- Identifiers
  - Numeric identifiers that may be stored with the Process Control Block include:
    - Identifier of this process
    - Identifier of the process that created this process (parent process)
    - User identifier

Processor State Information

- **User-Visible Registers**
  - A user-visible register is one that may be referenced by means of the machine language that the processor executes.
- **Control and Status Registers**
  - These are a variety of processor registers that are employed to control the operation of the processor. These include:
    - *Program Counter:* Contains the address of the next instruction to be fetched
    - *Condition codes:* Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow bits)
    - *Status information:* Includes interrupt enabled/disabled flags, execution mode
- **Stack Pointers**
  - Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.
    - The stack pointer points to the top of the stack.

Process Control Information (1)

- **Scheduling and State Information**
- This is information that is needed by the operating system to perform its scheduling function. Typical items of information:
  - *Process state:* defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
  - *Priority:* One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)
  - *Scheduling-related information:* This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
  - *Event:* Identity of event the process is awaiting before it can be resumed

Process Control Information (2)

- **Data Structuring**
  - A process may be linked to another process in a queue or other structure. E.g.,:
    - all processes in a waiting state for a particular priority level may be linked in a queue.
    - a process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.
- **Inter-process Communication**
  - Various flags, signals, and messages may be associated with communication between two independent processes.
- **Process Privileges**
  - Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed.
  - In addition, privileges may apply to the use of system utilities and services.

Process Control Information (3)

- **Memory Management**
  - This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

- **Resource Ownership and Utilization**
  - Resources controlled by the process may be indicated, such as opened files.
  - A history of utilisation of the processor or other resources may also be included
    - this information may be needed by the scheduler.

Threads and Processes

Threads and Processes

- A process is defined sometimes as a *heavyweight process*
- A thread is defined as a *lightweight process*

- Separate two ideas:
  - ***Process***: Ownership of memory, files, other resources

    -> execution of applications
  - ***Thread***: Unit of execution we use to dispatch

    -> share the same address space hence can read from and write to the same data structures

- **Multithreading**
  - Allow multiple threads per process

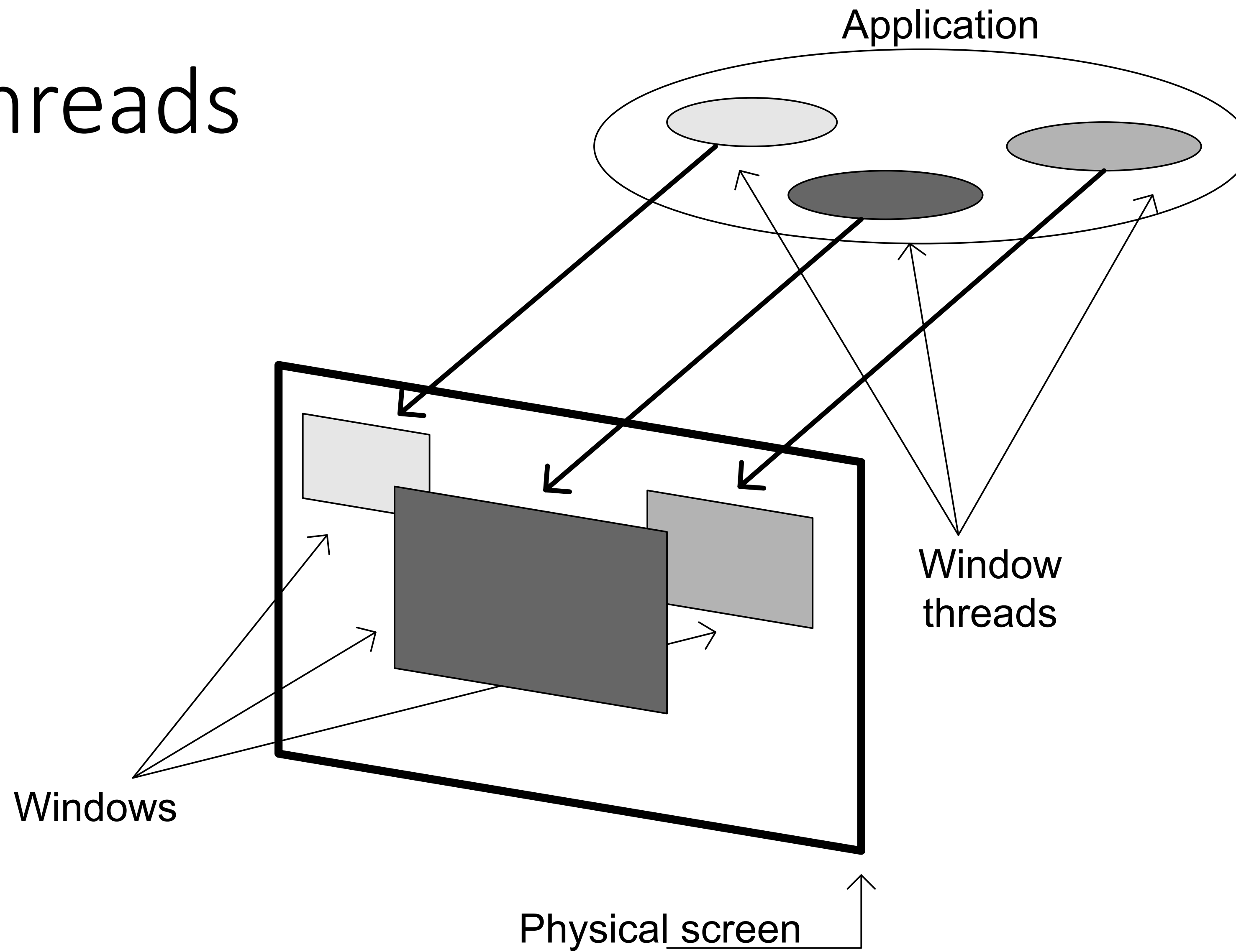
Threads (1)

- It is a unit of computation associated with a particular heavyweight process, using many of the associated process’s resources
  - has a minimum of internal state and a minimum of allocated resources
- A group of threads are sharing the same resources:
  - E.g., files, memory space, etc.
- The process is the execution environment for a family of threads
  - a process with one thread is a *classic* process
- A thread belongs only to one process

Threads (2)

- Individual execution state
- Each thread has a control block, with a state (Running/Blocked/etc.), saved registers, instruction pointer
- Separate stack and hardware state (PC, registers, PSW, etc.) per thread
- Shares memory and files with other threads that are in that process
- Faster to create a thread than a process
- Because a family of threads belonging to the same process have common resources, the thread switch is very efficient
- Thread switch for threads from different processes is as complex as classic process switch

Using Threads



References

- “Operating Systems”, William Stallings, ISBN 0-13-032986-x
- “Operating Systems – A Modern Perspective”, Garry Nutt, ISBN 0-8053-1295-1

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing System & Organisation

Lecture 5: CPU Management - Scheduling

Content

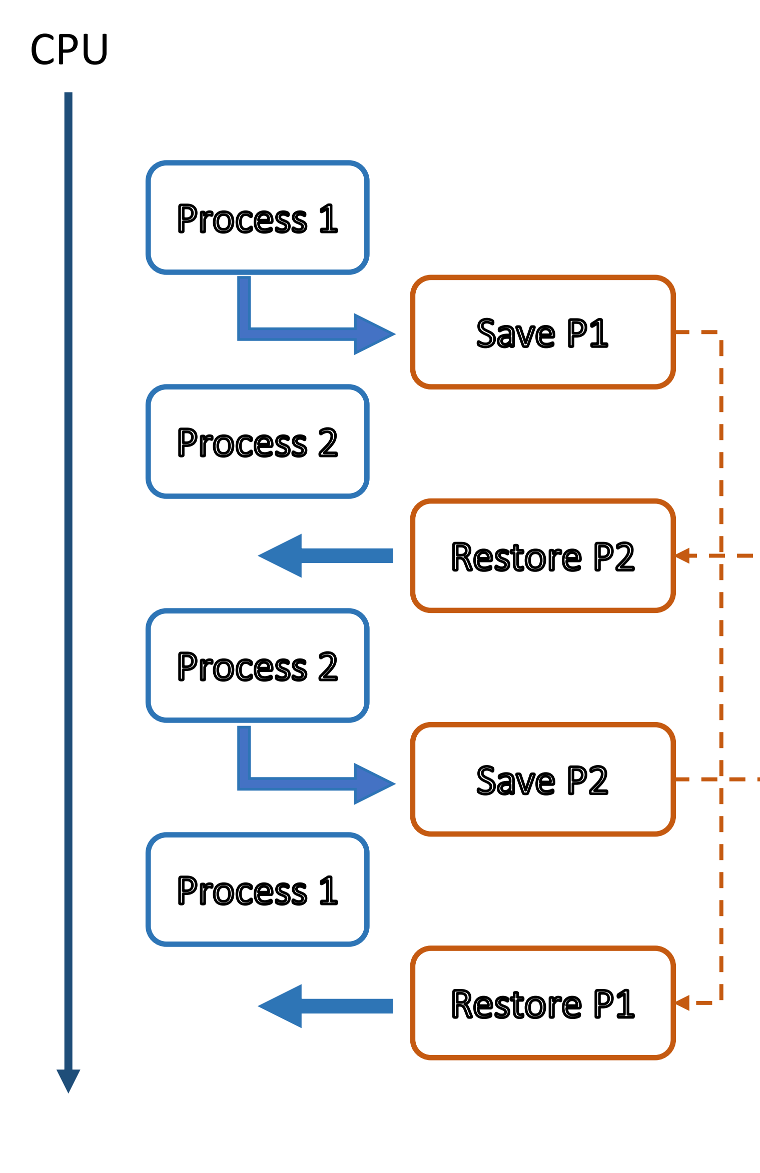
- Process scheduler organisation
- Scheduler types:
  - Non-preemptive
  - Preemptive
- Scheduling algorithms
  - FCFS (First Come First Served)
  - SRTN (Shortest Remaining Time Next)
  - SJF (Shortest Job First)
  - Time slice (Round Robin)
  - Priority based preemptive scheduling
  - MLQ (Multiple Level Queue)
  - MLQF (Multiple Level Queue with Feedback)

Photo by Andrey Grushnikov from Pexels

Scheduling

- Scheduling allows one process to use the CPU while the execution of another process is on hold (i.e., in waiting state) due to unavailability of any resource like I/O etc
  - Aims to make the system efficient, fast and fair.

- Scheduling is part of the process manager



Scheduling

- Scheduling is the mechanism that handles
  - **the removal of the running processes from the CPU**
  - **and the selection of another process**
- It is responsible for ***multiplexing*** processes on the CPU.

  -> when it is time for the ***running*** process to be removed from the CPU (in a *ready* or *suspended* state), a different process is selected from the set of processes in the ready state
- The selection of another process is based on a particular strategy.
  - The **scheduling algorithm** will determine the order in which the OS will execute the processes.

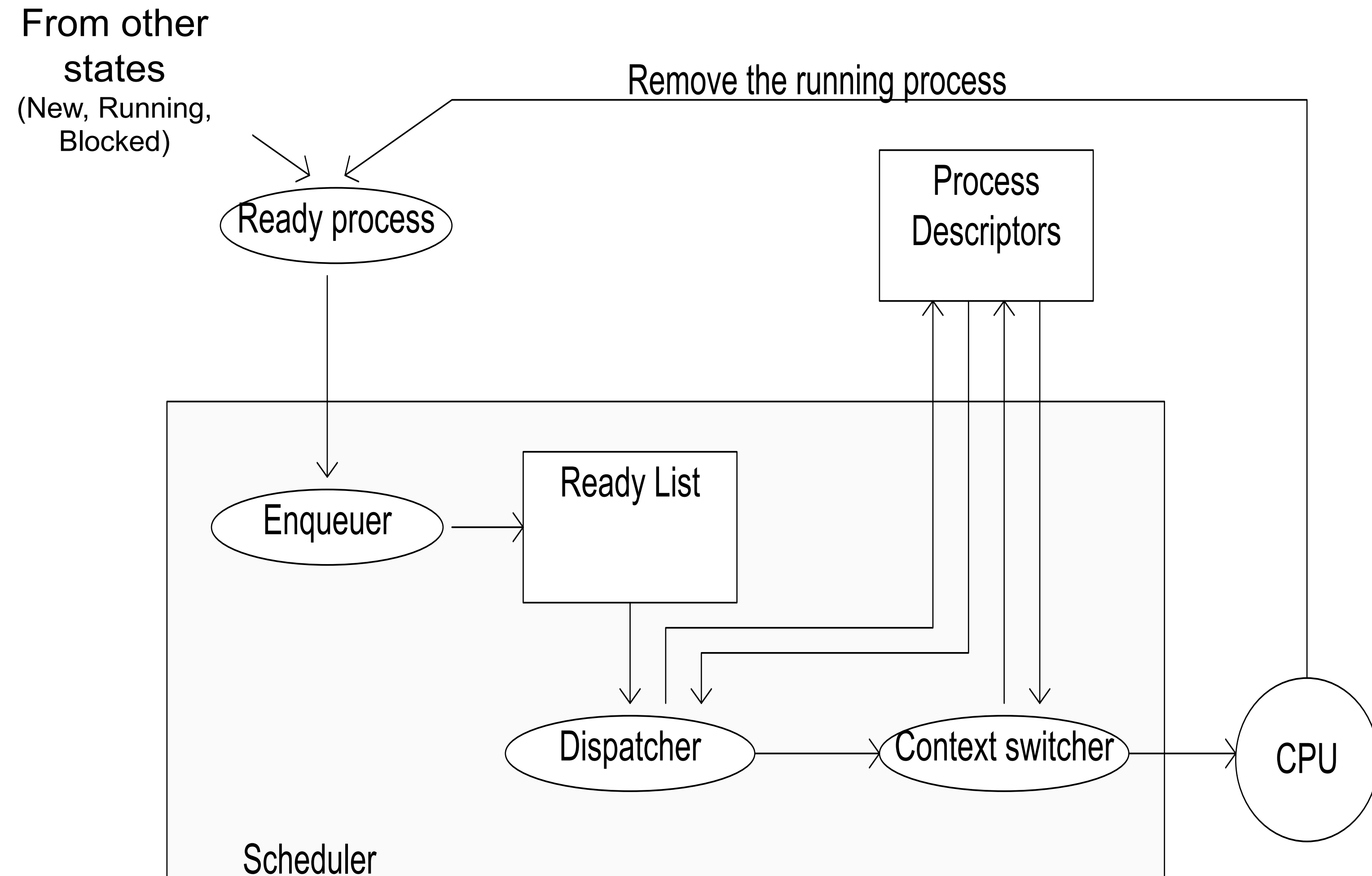
Scheduler Organisation

When a process is changed in the ready state, the ***enqueuer*** places a pointer to the process descriptor into a ready list

Context switcher saves the content of all processor registers of the process being removed into the process' descriptor, whenever the scheduler switches the CPU from executing a process to executing another

- Voluntary context switch
- Involuntary context switch

The ***dispatcher*** is invoked after the current process has been removed from the CPU; the dispatcher chooses one of the processes enqueued in the ready list and then allocates CPU to that process by performing another context switch from itself to the selected process



Scheduler Types

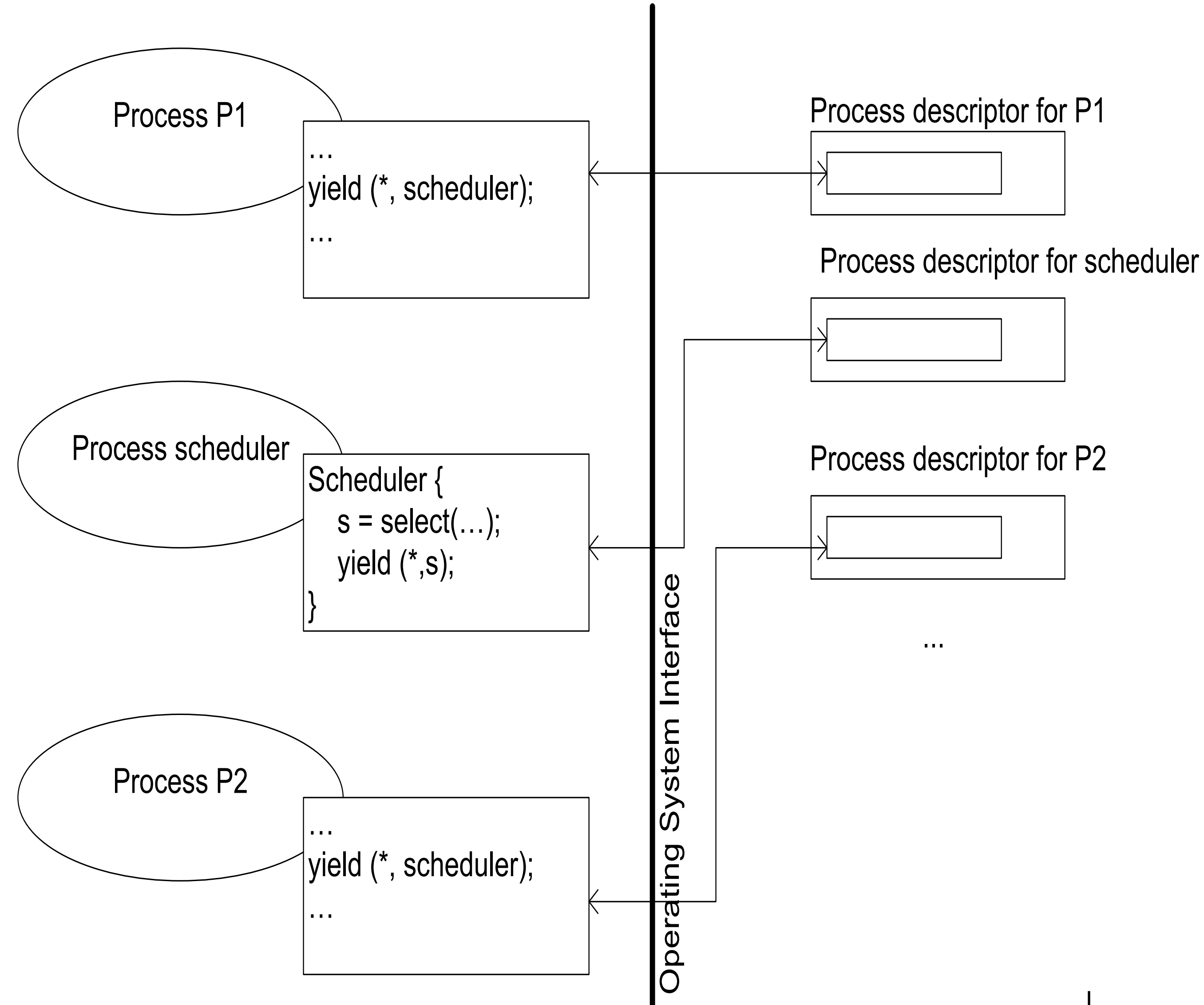
- *Cooperative* scheduler (**voluntary** CPU sharing)
  - Each process will ***periodically invoke*** the process scheduler, voluntarily sharing the CPU
  - Each process should call a function that will implement the process scheduling.
    - ***yield*** **(**$P_{current}$, P_{next}**)** (sometimes implemented as an instruction in hardware), where $P_{current}$ is an identifier of the current process and the P_{next} is an identifier of the next process)

- *Preemptive* scheduler (**involuntary** CPU sharing)
  - The interrupt system ***enforces periodic involuntary interruption*** of any process's execution; it can force a process to involuntarily execute a yield type function (or instruction)
  - This is done by incorporating an ***interval timer*** device that produces an interrupt whenever the time expires

Cooperative Scheduler

- Possible problems:
  - If the processes do not voluntarily cooperate with the others, one process could keep the CPU forever

- Cooperative multitasking allows much ***simpler implementation*** of applications
  - because their ***execution is never unexpectedly interrupted*** by the process scheduler



Preemptive Scheduler

- A programmable interval timer will cause an ***interrupt*** to run every K clock ticks of an interval time
  - thus causing the hardware to execute the logical equivalent of a yield instruction to invoke the interrupt handler
- The interrupt handler for the timer interrupt will call the scheduler to reschedule the processor ***without*** any action on the part of the running process
- The scheduler decides which process is run next
- The scheduler is guaranteed to be invoked once every K clock ticks
  - Even if a given process will execute an infinite loop, it will ***not*** block the execution of the other processes

```
IntervalTimer{
  InterruptCount = InterrptCount -1;
  if (InterruptCount <=0){
    InterruptRequest = TRUE
    InterruptCount = K;
  }
}

SetInterval(<programableValue>{
  K = programmableValue;
  InterruptCount = K;
}
```

Performance Elements

- Having a set of processes P={pi, 0<=i<n}

  - **Service time**, ***τ(pi)*** – the amount of time a process needs to be in active/running state before it completes

  - **Wait time**, ***W(pi)*** – the time the process waits in the ready state before its first transition in the active state

  - **Turn around time, T_{TRnd}*(pi)*** – the amount of time between the moment a process enters the ready state and the moment the process exits the running state for the last time

- Those elements are used to measure the ***performance*** of each scheduling algorithm

Selection Strategies

- **Non-preemptive strategies**
  - Allow any process to run to completion once it has been allocated the control of the CPU
  - A process that gets the control of the CPU, releases the CPU whenever it ends or when it voluntarily gives up the control of the CPU
- **Preemptive strategies**
  - The highest priority process among all *ready* processes is allocated the CPU
  - All lower priority processes are made to yield to the highest priority process whenever it requests the CPU
    - The scheduler is called every time a process enters the ready queue as well as when an interval timer expires
  - It allows for equitable resource sharing among processes at the expense of overloading the system

Scheduling Algorithms

- FCFS (First Come First Served)
- SJF (Shortest Job First)
- SRTN (Shortest Remaining Time Next)
- Time slice (Round Robin)
- Priority based preemptive scheduling
- MLQ (Multiple Level Queue)
- MLQF (Multiple Level Queue with Feedback)

First Come First Served

- ***Non-preemptive*** algorithm
- This scheduling strategy assigns ***priority*** to processes in the order in which they request the processor
  - The priority of a process is computed by the enqueuer by ***time stamping*** all incoming processes and then having the dispatcher select the process that has the ***oldest time stamp***
  - Possible implementation: using a FIFO data structure (where each entry points to a process descriptor)
    - the enqueuer adds processes to the tail of the queue and the dispatcher removes processes from the head of the queue
- Easy to implement
- It is not widely used because of processes unpredictable
  - *turn around time*
  - *waiting time*

FCFS Example

Pi	$\tau(Pi)$
0	350
1	125
2	475
3	250
4	75

- **Average turn around time:**
  - T_{TRnd} = (350 +475 +950 + 1200 + 1275)/5 = 850

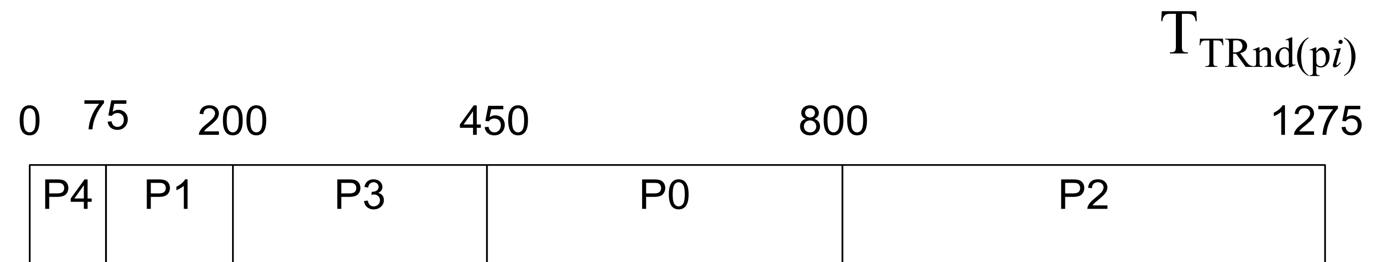
- **Average wait time:**
  - W = (0 + 350 +475 + 950 + 1200)/5 = 595

Shortest Job First

- ***Non-preemptive***
- It is an optimal algorithm from the point of view of average turn around time
  - It minimises the average turn around time
- Preferential service of short jobs
- It requires the ***knowledge of the service time*** for each process
- In the extreme case, where the system has little idle time, the processes with large service time will never be served
- In the case where it is not possible to know the service time for each process, this is estimated using predictors.

SJF Example

Pi	$\tau(Pi)$
0	350
1	125
2	475
3	250
4	75



- ***Average turn around time:***
  - T_{TRnd} = (800 + 200 +1275 + 450 + 75)/5 = 560

- ***Average wait time:***
  - W = (450 + 75 +800 + 200 + 0)/5 = 305

Shortest Remaining Time Next (SRTN)

- Similar to SJF
  - But ***preemptive***

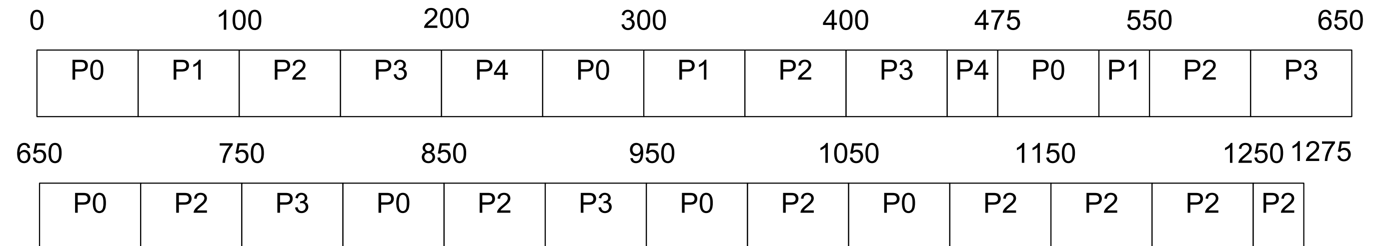
- a ***long job which is mostly complete*** might have a very short time remaining, and would therefore be prioritised

Time Slice (Round Robin)

- **Preemptive algorithm**
- Each process gets a time slice of CPU time, distributing the ***processing time equitably*** among all processes requesting the processor
- Whenever the time slice expires, the control of the CPU is given to the next process in the ready list
  - the process being switched is placed back into the ready process list
- It implies the existence of a ***specialized timer*** that measures the processor time for each process
  - every time a process becomes active, the timer is initialized
- It is ***not well suited for long jobs***, since the scheduler will be called multiple times until the job is done
- It is very sensitive to the size of the time slice
  - Too big – large delays in response time for interactive processes
  - Too small – too much time spent running the scheduler
  - Very big – turns into FCFS
- The time slice size is determined by analyzing the number of the instructions that the processor can execute in the given time slice.

Time Slice (Round Robin) Example

Pi	$\tau(Pi)$
0	350
1	125
2	475
3	250
4	75

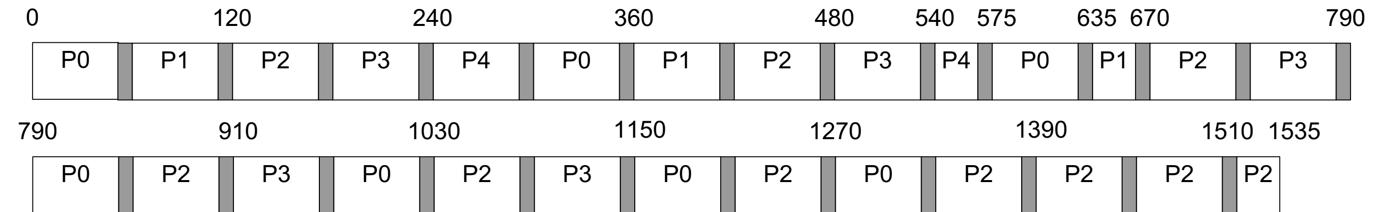


Time slice size is 50, negligible amount of time for context switching

- ***Average turn around time***:
  - T_{TRnd} = (1100 + 550 + 1275 + 950 + 475)/5 = 870
- ***Average wait time***:
  - W = (750+425+800+700+400)/5 = 615
- The wait time shows the benefit of RR algorithm in the terms of how quickly a process receives service

RR scheduling with overhead example

i	τ(p*i*)
0	350
1	125
2	475
3	250
4	75



Time slice size is 50, 10 units of time for context switching

- ***Average turn around time:***
  - T_{TRnd} = (1320 + 660 + 1535 + 1140 + 565)/5 = 1044
- ***Average wait time:***
  - W = (620 + 535 + 1060 + 890 + 490)/5 = 719

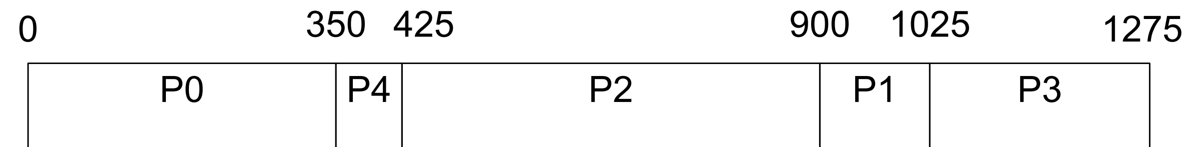
Priority based scheduling (Event Driven)

- Both ***preemptive*** and ***non-preemptive*** variants
- Each process has an ***externally assigned priority***
- Every time an event occurs that generates a process switch, the ***process with the highest priority*** is chosen from the ready process list
- There is the possibility that processes with ***low priority will never gain CPU time***
- There are variants with static and dynamic priorities; the dynamic priority computation solves the problem with processes that may never gain CPU time (the longer the process waits, the higher its priority becomes)
- It is used for real time systems.

Priority based schedule example

Pi	τ(Pi)	Priority
0	350	5
1	125	2
2	475	3
3	250	1
4	75	4

Highest priority corresponds to **highest** value



- ***Average turn around time:***
  - T_{TRnd} = (350 + 425 + 900 + 1025 + 1275)/5 = 795

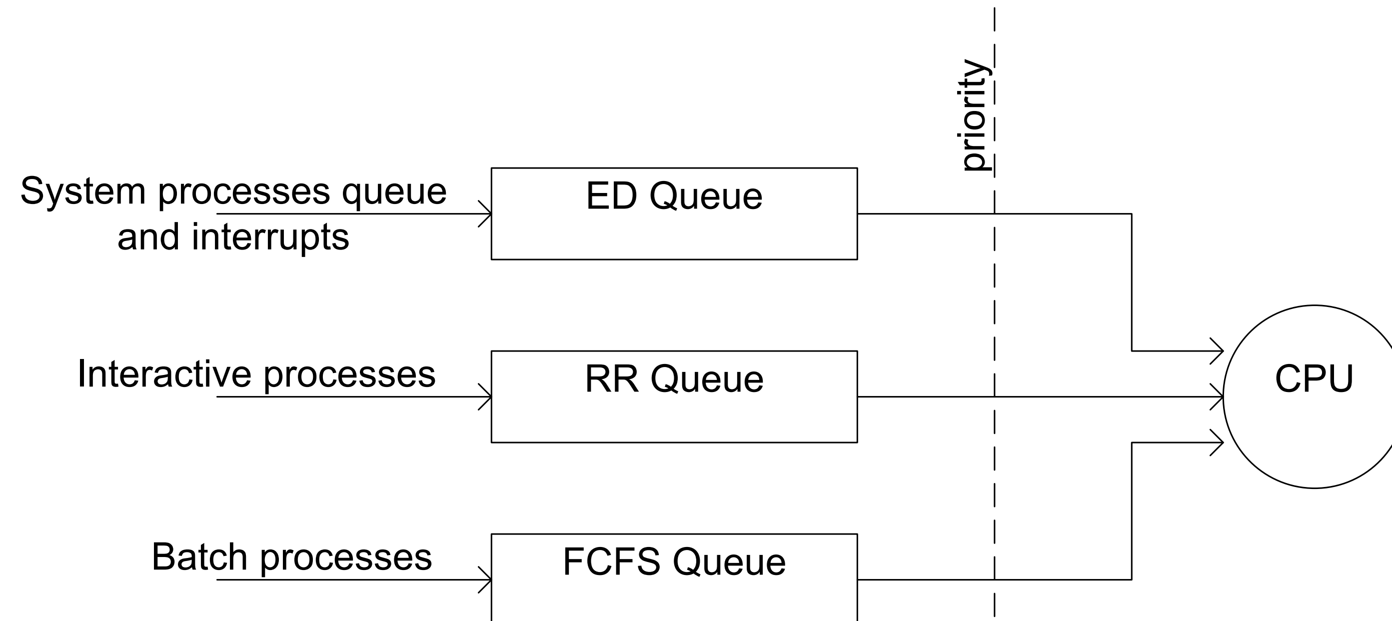
- ***Average wait time:***
  - W = (0 + 350 + 425 + 900 + 1025)/5 = 540

Multiple Level Queue scheduling

- Complex systems have requirements for real time, interactive users and batch jobs
  - Therefore, a ***combined scheduling mechanism*** should be used
- The processes are divided in ***classes***
- Each class has a process queue, and it has assigned a specific scheduling algorithm
- Each process queue is treated according to a queue scheduling algorithm:
  - Each queue has assigned a priority
  - As long as there are processes in a higher priority queue, those will be serviced

MLQ Example

- **2 queues**
  - Foreground processes (highest priority)
  - Background processes (lowest priority)
- **3 queues**
  - OS processes and interrupts (highest priority, serviced ED)
  - Interactive processes (medium priority, serviced RR)
  - Batch jobs (lowest priority, serviced FCFS)



Multiple Level Queue with feedback

- Same with MLQ, but the processes could ***migrate from class to class*** in a dynamic fashion
- Different ***strategies*** to modify the priority:
  - Increase the priority for a given process (the user needs larger share of the CPU to sustain acceptable service)
  - Decrease the priority for a given process (the user process is trying to get more CPU share, which may impact on the other users)
  - If a process is giving up the CPU before its time slice expires, then the process is assigned to a higher priority queue
- During the evolution to completion, a process may go through a ***number of different classes***
- ***Any of the previous algorithms may be used for treating a specific process class.***

Exercise

- Draw a Gantt Chart that illustrate the execution of these processes using the following scheduling algorithm:
  - ➢ FCFS (First Come First Served)
  - ➢ SJF (Shortest Job First) – nonpreemptive
  - ➢ SRTN (Shortest Remaining Time Next)
  - ➢ Time slice (Round Robin, assume a time slice of 1 second)
  - ➢ Priority based preemptive scheduling
- Calculate the average waiting time using each scheduling algorithm.

Larger Number = Higher Priority

Process	Length (s)	Arrival time (s)	Priority
	5:00	0:00	1
P2	2:00	2:00	2
P3	1:00	3:00	3

References

- “Operating Systems – A modern perspective”, Garry Nutt, ISBN 0-8053-1295-1

- Process Scheduling: https://www.youtube.com/watch?v=THqcAa1bbFU

Ollscoil na Gaillimhe
University of Galway

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing System & Organisation

Lecture 6: Process Synchronisation

Concurrent Programming

- Concurrent programs: *interleaving sets of sequential atomic instructions*.
    - i.e., interacting sequential processes run at same time, on same/different processor(s)
    - processes *interleaved*, i.e. at any time each processor runs one of instructions of the sequential processes

Correctness

If all the math is done in registers, then the results depend on interleaving (indeterminate computation).

- This dependency on unforeseen circumstances is known as a *Race Condition.*

Generalisation: a program is correct when its preconditions hold then its post conditions will hold.

```
Program1: load reg, N
Program2: load reg, N
Program1: add reg, #1
Program2: add reg, #1
Program1: store reg, N
Program2: store reg, N
```

A concurrent program *must be* correct under all possible *interleavings*.

Lets Look at this in Practice: Race Conditions

- A **race condition** occurs when a program output is dependent on the sequence or timing of code execution
  - if multiple processes of execution enter a **critical section** at about the same time; both attempt to update the shared data structure
  - ➢leads to surprising results (undesirable)
  - ❖You must work to avoid this with concurrent code

- **Critical section** = parts of the program where a shared resource is accessed
  - It needs to be protected in ways that avoid the concurrent access

Example Bank Transaction

```
Int withdraw(account, amount){
      int balance = account.balance;
      balance = balance - amount ;
      account.balance = balance;
      return balance;
}
```

Example Bank Transaction

Two processes:

- Process 1: withdraw 10 from account
- Process 2: withdraw 20 from account

```
//account.balance = 100
```

Process 1

```
Int withdraw(account, amount = 10){
        int balance = account.balance; //100
        balance = balance - amount ; //90
```

Process 2

```
Int withdraw(account, amount = 20){
        int balance = account.balance; //80
        balance = balance - amount ; //80
        account.balance = balance; //80
```

Process 1

```
        account.balance = balance; //90
        return balance; //90
}
```

Process 2

```
        return balance; //80
}
```

```
//account.balance = 90!
```

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Race Condition Consequences

We can get different results every time we run the code

- ➢ result is **indeterminate**

Deterministic computations have the same result each time

- We want deterministic concurrent code
- ➢We can use synchronisation mechanisms

Handling Race Conditions

- We need a mechanism to control access to shared resources in concurrent code
  - ➢ Synchronisation is necessary for any shared data structure

<u>Idea:</u>

- Focus on critical sections of code
  - i.e., bits that access shared resources
- We want critical sections to run with mutual exclusion
  - ➢ only one process can execute that code at the same time

Example: Bank Transactions

What code should be within the critical section?

```
1 int withdraw(account, amount){
2     int balance = account.balance;
3     balance = balance - amount ;
4     account.balance = balance;
5     return balance;
6}
```

Critical section (lines 2–4)

Q: Why is this not critical? (line 5)

Critical Section Properties

- **Mutual exclusion**: only 1 process can access at a time
- **Guarantee of progress**: processes outside the critical section cannot stop another from entering it
- **Bounded waiting**: a process waiting to enter a critical section will eventually enter
  - Processes in the critical section will eventually leave
- **Performance**: the overhead of entering/exiting should be small
  - Especially compared to amount of work done in there – why?
- **Fair**: don’t make some processes wait much longer than others

Synchronisation Solutions

Ways to protect critical sections

- Option 1: Atomicity
  - Atomic operations cannot be interrupted, in order to avoid illogical outcomes
- Option 2: Conditional synchronisation (ordering)
  - Making sure that one process runs before another

Atomicity

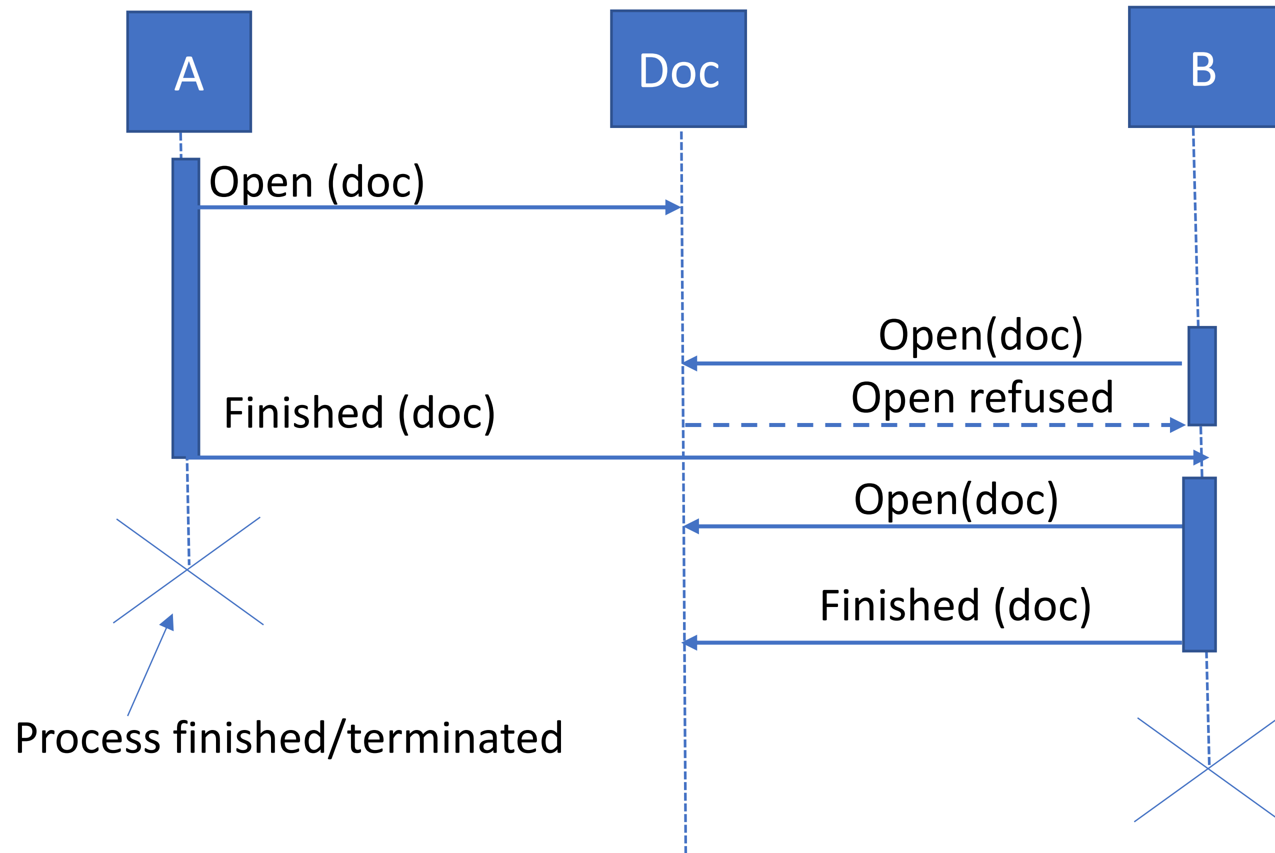
- Basic atomicity is provided by the hardware
  - E.g., *References and assignments (i.e., read & write operations) are atomic in all CPUs*

- However higher-level constructs (i.e., any sequence of two or more CPU instructions) are not atomic in general
- Some languages (e.g., Java) have mechanisms to specify multiple instructions as atomic

Conditional Synchronisation

- Strategy: Person A writes a rough draft and then Person B edits it.
  - A and B cannot write at the same time (as they are working on different versions of the paper)
  - Must ensure that Person B cannot start until Person A is finished

What Might Conditional Synchronisation Look Like?



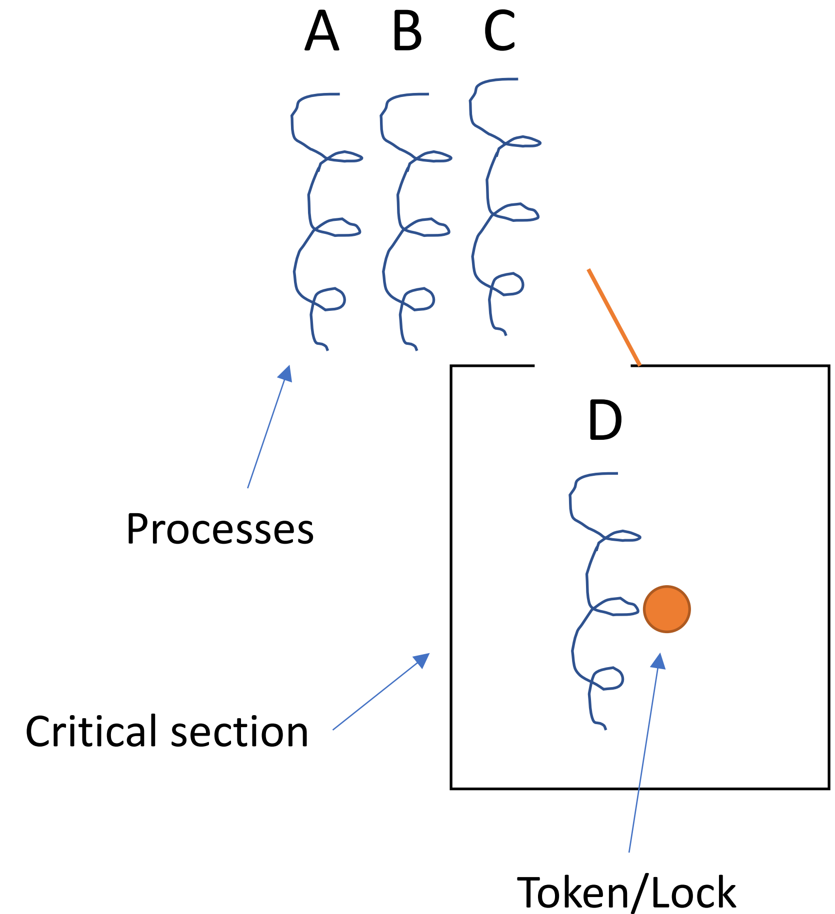
Code Constructs to Support Defining Critical Sections

- Locks
  - Very primitive, just provide mutual exclusion, minimal semantics, useful as a building block for other methods
- Semaphores
  - Basic, easy to understand
- Monitors
  - Higher level abstraction, requires language support, implicit operations

Mutual Exclusion solutions: Locks

Locks: Basic idea

- Lock = a token you need to enter a critical section of code
- If a process wants to execute a critical section…it must have the lock:
  - Need to ask for lock
  - Need to release lock
- No restrictions on executing other code



Lock States and Operation

- Locks have 2 states:
  - Held: some process is in the critical section
  - Not held: no process is in the critical section
- Locks have 2 operations:
  - Acquire:
    - mark lock as held or wait until released
    - If not held => execute immediately
  - Release:
    - mark lock as not held

If many processes call acquire, only 1 process can get the lock

Using Lock

- Locks are declared like variables:
  `Lock myLock;`
- A program can use multiple locks – why?
  `Lock myDataLock, myIoLock;`
- To use a lock:
  - Surround critical section as follows:
    - Call **acquire**() at start of critical section
    - Call **release**() at end of critical section
- Remember our general pattern for mutex

```
while (true)
    // Non_Critical_Section

    myLock.acquire();

    // Critical_Section

    myLock.release();

    // Non_Critical_Section
end while
```

Surround critical section of code

Ollscoil na Gaillimhe University of Galway

Lock Benefits

- Only 1 process can execute the critical section code at a time
- When a process is done (and calls release) another process can enter the critical section

➢ Achieves requirements of **mutual exclusion** and **progress** for concurrent systems

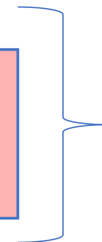
Lock Limitations

- Acquiring a lock *only* blocks processes trying to acquire the *same* lock
  - i.e., processes can acquire other locks
- **Must use the same lock for all critical sections accessing the same data** (or resource)
  - E.g., withdraw() and deposit() for a bank account
- Q: What does this mean for code complexity?
  - E.g., Add a new process that accesses same data

Lock in Use Example: Bank Transactions

See our old code:

```
int withdraw(account, amount){
        acquire(myBalanceLock);
        int balance = account.balance;
        balance = balance – amount ;
        account.balance = balance};
        release(myBalanceLock);
        return balance;
}
```



The local variable, does not need to be protected

E.g., Bank Transaction with Locks

```
//account.balance = 100
```

P1	`Int withdraw(account, amount = 10){`
	`acquire(myBalanceLock);`
	`int balance = account.balance; //100`

P2	`Int withdraw(account, amount = 20){` `acquire(myBalanceLock);      // Process STALLED`

P1	`balance = balance - amount ; //90`
	`account.balance = balance; //90`
	`release(myBalanceLock);     // NOW P2 can start`

P2	`int balance = account.balance; //90`
	`balance = balance - amount ; //70`
	`account.balance = balance; //70`
	`release(myBalanceLock);`
	`return balance; //70`
	`}`

P1	`return balance; //90` `}`

```
//account.balance = 70
```

Impacts

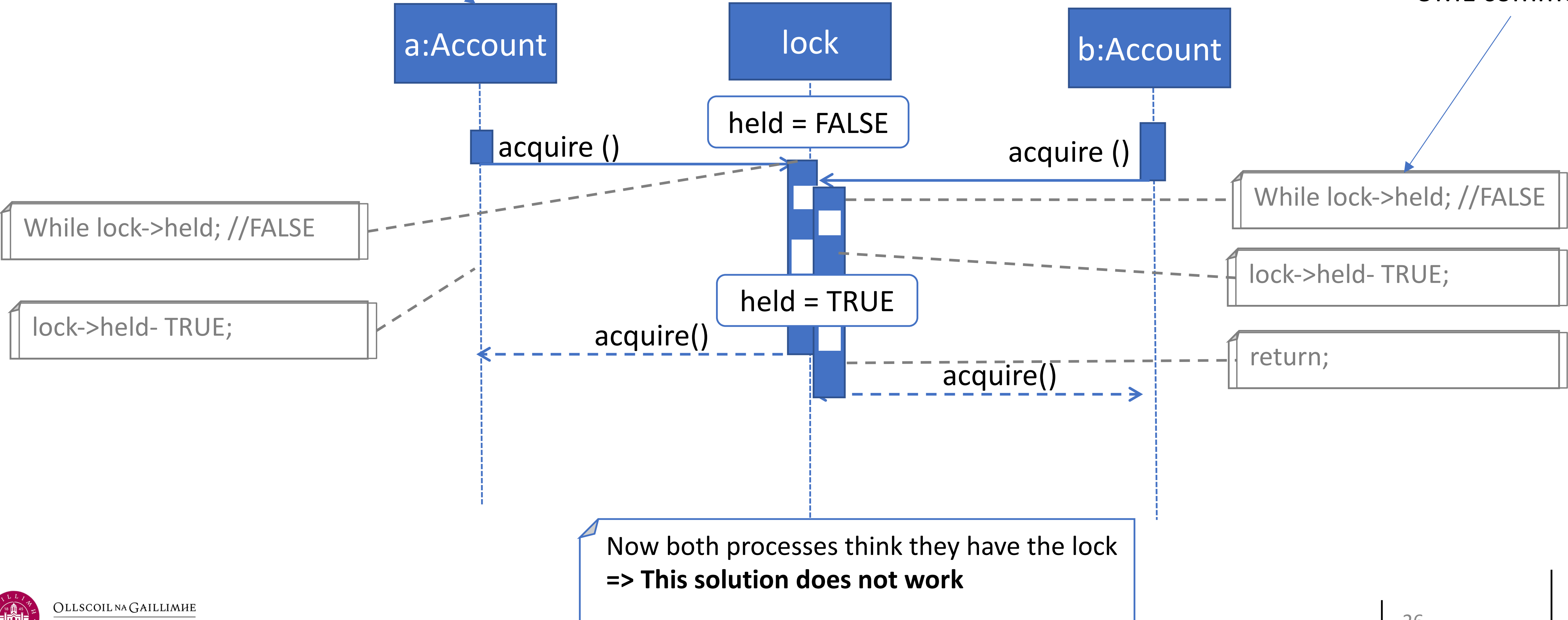
- We can run the processes in any order:
  - We will have the correct final balance

➢We no longer have a race condition

Software Implementation of Locks (v1)

```
Struct lock {
     bool held; //initially FALSE
}
void acquire(lock) {
     while(lock->held)
          ; //just wait
     lock->held = TRUE;
}
void release(lock) {
     lock->held = FALSE;
}
```

How does it run?



OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Solve via Hardware Support

```
//c code for test and set behaviour
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
}
```

Processor has a special instruction called “test and set”

- Allows atomic read **and** update

Hardware-based Spinlock

```
struct lock {
     bool held; //initially FALSE
}
void acquire(lock) {
     while(test_and_set(&lock->held))
          ; //just wait
     return;
     }
void release(lock) {
     lock->held = FALSE;
}
```

Q: Why is this called a spin lock?

Drawbacks of Spinlocks

- Spinlocks are a form of busy waiting
  => burn CPU time
- Once acquired they are held until explicitly released
  - What about other processes?
- Inefficient if lock is held for long periods
  - OS overhead of context switching
  - If Process Scheduler makes processes sleep while lock is held
    - ➢ All other processes use their CPU time to spin while the process with the lock makes no progress

Do Locks give us sufficient safety?

1. **Check Safety properties:** these must *always* be true
   - *Mutual exclusion:* Two processes must not interleave certain sequences of instructions
   - *Absence of deadlock:* Deadlock is when a non-terminating system cannot respond to any signal

2. **Check Liveness properties:** These must *eventually* be true
   - *Absence of starvation:* Information sent is delivered
   - *Fairness:* That any contention must be resolved

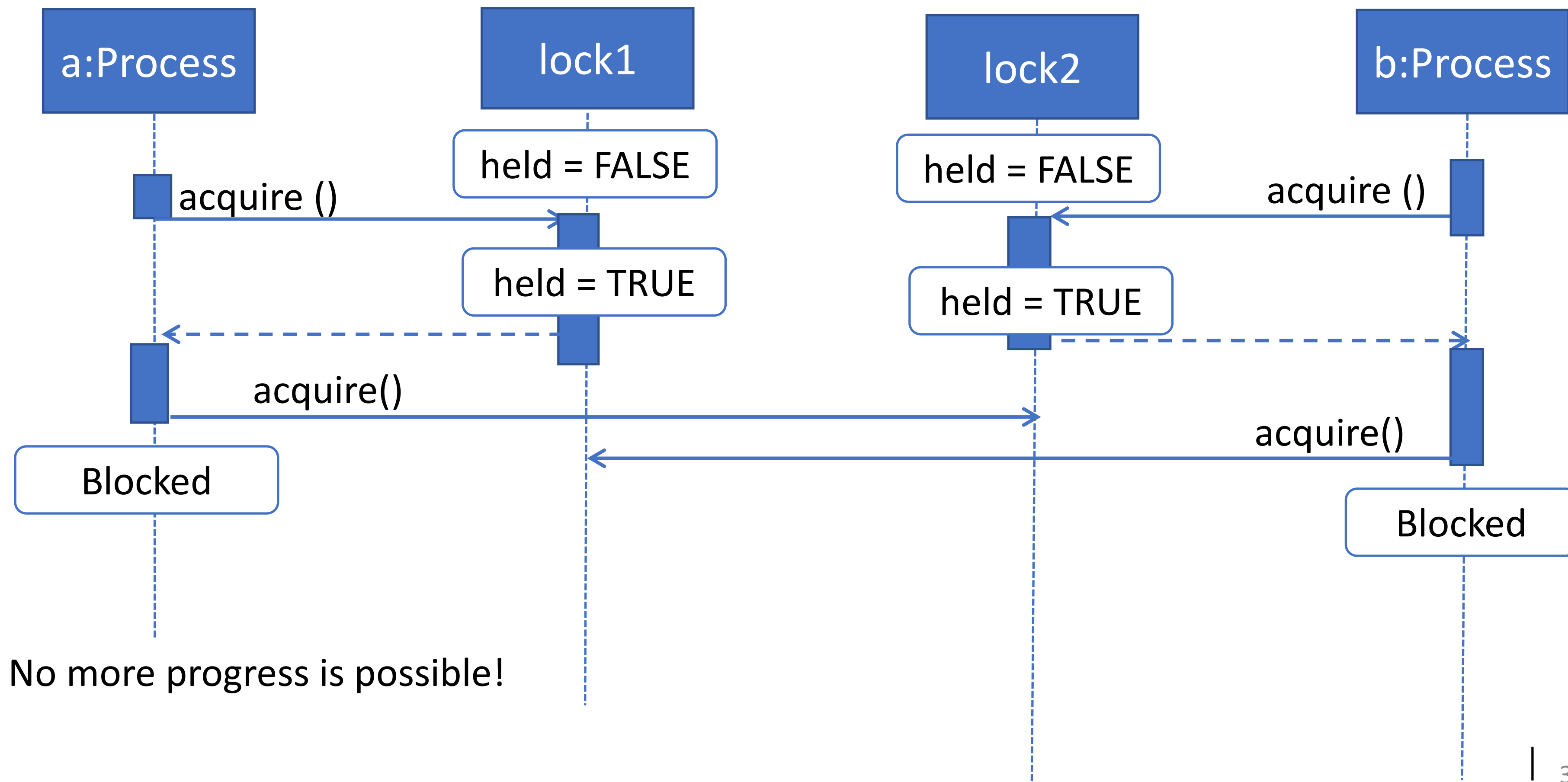
- If you can demonstrate **any** cases in which these properties do not hold

➢ then, **the system is not correct**

Q: What do you think?

Lock Deadlock Scenario

- 2+ processes, 2 shared resources, 2 locks



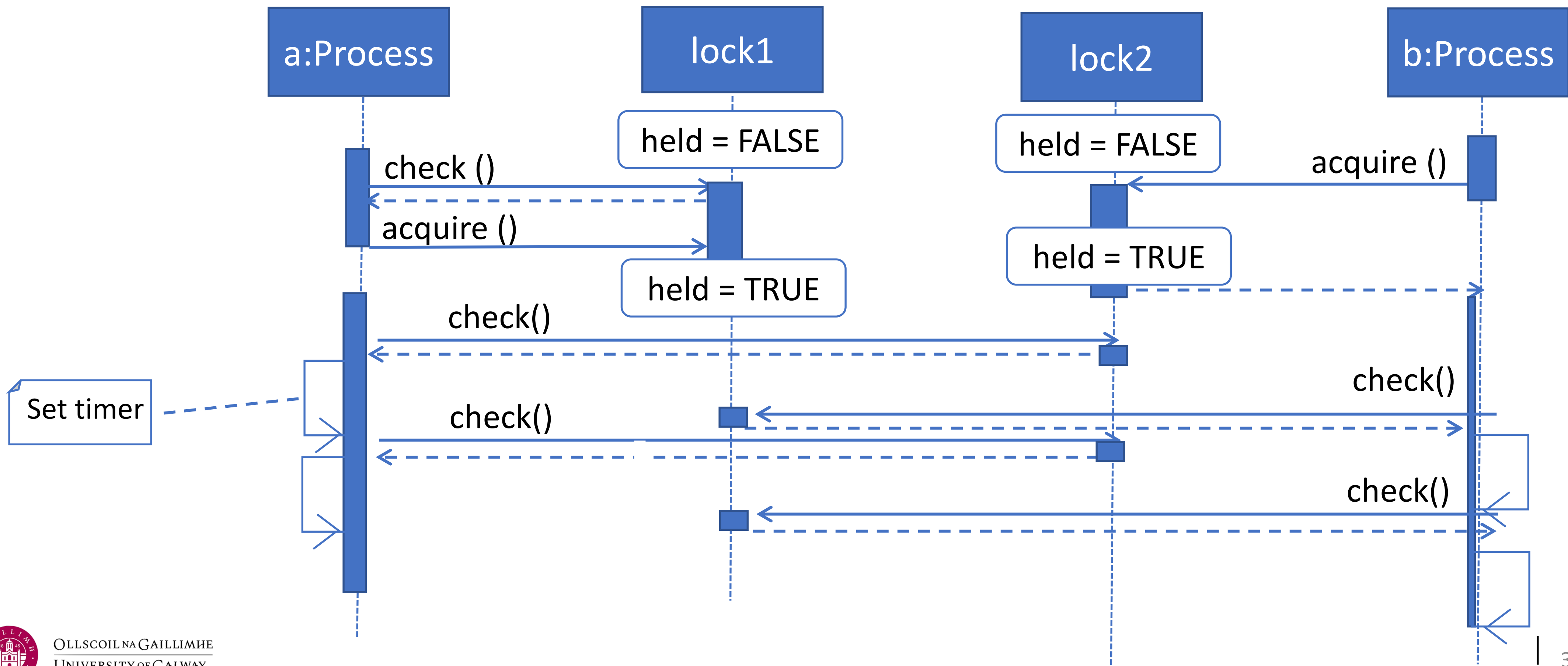
Protocols to avoid deadlock

- Add a timer to lock.request() method
  - ➢Cancel job and attempt it another time
- Add a new lock.check() method to see if a lock is already held before requesting it
  - ➢you can do something else and come back and check again
- Avoid hold and wait protocol
  - ➢never hold onto 1 resource when you need 2

But these all lead to problems too!

Livelock by trying to avoid deadlock

- 2 processes, 2 resources, locks with checking



Starvation

- More general case of livelock
- 1 or more processes do not get to run as another process is locking the resource
- Example:
  - 2 processes
    - **Process A** runs for 99ms, releases lock for 1ms
    - **Process B** runs for 1ms, releases lock for 90ms
  - ➢A sends many more requests for resource
  - ➢B hardly ever gets allocated the resource

Locks/Critical Sections and Reliability

- What if a process is interrupted, is suspended, or crashes inside its critical section?
- In the middle of the critical section, the system may be in an inconsistent state
- Not only that: the process is holding a lock and if it dies no other process waiting on that lock can proceed!

- **Developers must ensure critical regions are very short and always terminate.**

Beyond Locks

- Locks only provide mutual exclusion
  - Ensure only 1 process is in the critical section at a time
  - Good for protecting our shared resource to prevent race conditions and avoid nondeterministic execution
    - E.g., bank balance We want more!
- What about fairness, avoiding starvation, and livelock?
  - ➢We need to be able to place an ordering on the scheduling of processes

Take Home Message

- Race conditions, deadlock, livelock, fairness, and reliability are all concerns when writing concurrent code
- *Several mechanisms exist to ensure the orderly execution of cooperating processes*

Higher Level Support for Mutual Exclusion: Semaphores

Example Scenario: we want to place an order on when processes execute

- Producer- Consumer:
  - Producer: creates a resource (data)
  - Consumer: Uses a resource (data)
  - E.g. `ps | grep "gcc" | wc`
- Don't want producers and consumers to operate in lockstep (i.e., atomicity)
  - Each command must wait for the previous output
  - Implies lots of context switching (i.e., very expensive)
- **Solution:** place a fixed size buffer between producers and consumers
  - Synchronise access to buffer
  - Producer waits of buffer full; consumer waits if buffer empty

Semaphores

- Semaphore = higher level synchronisation primitive
  - Invented by Dijkstra in 1965 as part of THE OS project
- Semaphores are a kind of generalized lock
  - Main synchronisation primitive used in original UNIX
- Implement with a **counter** that is manipulated atomically via 2 operations **signal** and **wait**

`wait(semaphore)`: A.K.A., *down()* or *P()*
  decrement counter
  if counter is zero then block until semaphore is signalled

`signal(semaphore)`: A.K.A., *up()* or *V()*
  increment counter
  wake up one waiter, if any

`sem_init(semaphore, counter)`:
  set initial counter value

Semaphore Pseudocode

`wait()` and `signal()` are critical sections!

- Hence, they must be executed atomically with respect to each other

- Each semaphore has an associated queue of processes
  - When `wait()` is called by a process
    - If semaphore is available => process continues
    - If semaphore is unavailable => process blocks, waits on queue
  - `signal()` opens the semaphore
    - If processes are waiting on a queue => one process is unblocked
    - If no processes are on the queue => the signal is remembered for the next time wait() is called

Note: Blocking processes are not spinning, they release the CPU to do other work

```
struct semaphore {
        int value;
        queue L; // list of processes
}
wait (S) {
        if (s.value > 0)
                s.value = s.value -1;
        else {
                add this process to s.L;
                block;
        }
}
signal (S) {
        if (S.L != EMPTY){
                remove a process P from S.L;
                wakeup(P);
        } else
                s.value = s.value + 1;
}
```

Semaphore Initialisation

- If semaphore initialised to 1
  - First call to wait goes through
    - Semaphore value goes from 1 to 0
  - Second call to wait() blocks
    - Semaphore value stays at zero, process goes on queue
  - If first process calls signal()
    - Semaphore value stays at 0
    - Wakes up second process

$\Rightarrow$Acts like a mutex lock

$\Rightarrow$Can use semaphores to implement locks

This is called a **binary semaphore**

What happens if we initialise to 2?

Initial value of semaphore = number of processes that can be active at once:

- Sem_init(sem, 2)
  - value=2, L =[]

Consider multiple processes:

- Process1: wait(sem)
  - `value=1,L=[],  P1 executes`
- Process2: wait(sem)
  - `value=0, L[], P2 executes`
- Process3: wait(sem)
  - `value=0, L[P3], P3 blocks`

```
struct semaphore {
        int value;
        queue L; // list of processes
}
wait (S) {
        if (s.value > 0)
                s.value = s.value -1;
        else {
                add this process to s.L;
                block;
        }
}
signal (S) {
        if (S.L != EMPTY){
                remove a process P from
S.L;
                wakeup(P);
        } else
                s.value = s.value + 1;
}
```

Uses of Semaphores

- Allocating a number of resources
  - Shared buffers: each time you want to access a buffer, call wait() => you are queued if there is no buffer available
- Counter is initialised to N = number of resources
- Called a **counting semaphore**
- Useful for conditional synchronisation
  - i.e., one process is waiting for another process to finish a piece of work before it continues

Semaphores for Mutual Exclusion

With semaphores:

- guaranteeing mutual exclusion for N processes is trivial

```
semaphore mutex = 1;

void Process(int i) {
    while (1) {
            // Non Critical Section Bit
            wait(mutex) // grab the mutual exclusion semaphore
            // Do the Critical Section Bit
            signal(mutex) //grab the mutual exclusion semaphore
    }
}

int main ( ) {
    cobegin {
            Process(1);   Process(2);
    }
}
```

Bounded Buffer Problem

- Producer-consumer problem
  - Buffer in memory
    - Finite size of N entries
  - A producer process inserts an entry into it
  - A consumer process removes an entry from it
- Processes are concurrent
  - ➢ We must use a synchronisation mechanism to control access to shared variables describing buffer state

Producer-Consumer Single Buffer

- Simplest case
  - Single producer process, single consumer process
  - Single shared buffer between the Producer and the Consumer
- Requirements
  - Consumer must wait for Producer to fill buffer
  - Producer must wait for Consumer to empty buffer (if filled)

Semaphores can be Hard to Use

- Complex patterns of resource usage
  - Cannot capture relationships with semaphores alone
  - Need extra state variables to record information

$\Rightarrow$ Produce buggy code that is hard to write

- If one coder forgets to do `V()/signal()` after critical section, the whole system can deadlock

Monitors

- Need a higher level construct:
  - Groups the responsibility for correctness
  - Supports controlled access to shared data

- *Monitors*: an extension of the monolithic monitor used in OS to allocate memory.
  - A programming language construct that supports controlled access to shared data
  - Synchronisation code added by compiler, enforced at runtime (Less work for programmer!)
- Monitors keep track of who is allowed to access the shared data and when they can do it
- Monitors Encapsulate
  - Shared data structures
  - Procedures that operate on shared data
  - Synchronisation between concurrent processes that invoke these procedures

Detection and Protection of Deadlock

Requirements for Deadlock

All 4 conditions must hold for deadlock to occur:

1. **Mutex**: at least one held resource must be non-shareable
2. **No pre-emption:** resources cannot be pre-empted (no way to break priority or take a resource away once allocated
   - Locks have this property
3. **Hold and wait**: there exists a process holding a resource and waiting for another resource
4. **Circular wait**: there exists a set of processes P_1, P_2,…,P_N such that P_1 is waiting for P_2, P_2 is waiting for P_3,… and P_N is waiting for P_1

Make code more efficient, hence, we want them

Need to avoid circular wait

If only 3 conditions hold then:

- you can get starvation
- but not deadlock

Sample Deadlock

- Acquire locks in different orders
- Example:

<u>Process 1</u>	<u>Process 2</u>
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

Sample Deadlock – Check for Deadlock

- Example:

Process 1	Process 2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

1. Do we have mutex?
2. Do we have hold and wait?
3. Do we have no pre-emption?
4. Do we have a circular wait?

Deadlocks without Locks

- Deadlocks can occur for any resource or any time a process waits, e.g.
  - Messages: waiting to receive a message before sending a message
    - i.e., hold and wait
  - Allocation: waiting to allocate resources before freeing another resource
    - i.e., hold and wait

Testing for Real World Deadlock

- How do cars do it?
  - We have rules to avoid it/recover from it
  - E.g.,
    - Never block an intersection
    - Must backup if you find yourself doing so (a form of pre-emption)
- Why does this work?
  - Breaks a “hold and wait”
  - Shows that refusing to hold a resource while waiting for something else is a key element of avoiding deadlock

Dealing With Deadlocks: Ignore

- Strategy 1: Ignore the fact that deadlocks may occur
  - Write code, put nothing special in
  - Sometimes you have to re-boot the system
  - May work for some unimportant or simple applications where deadlock does not occur often
- Quite a common approach!

Dealing with Deadlock: Reactive

- Periodically check for evidence of deadlock
  - E.g., add timeouts to acquiring a lock, if you timeout then it implies deadlock has occurred and you must do something
- Recovery actions:
  - Blue screen of death and reboot computer
  - Pick a process to terminate, e.g., a low priority one
    - Only works with some types of applications
    - May corrupt data so process needs to do clean-up when terminated

Dealing with Deadlock: Proactive

- Prevent 1 of the 4 necessary conditions for deadlock
- No single approach is appropriate (or possible) for all circumstances
  - Need techniques for each of the four conditions

Solution 1: No Mutual Exclusion

- Make resources shareable
- Example: read-only files
  - No need for locks
- Example: per-process variables
  - Counters per process instead of global counter
- Not possible for all bits of code/applications

Fixing our Sample Deadlock Code

Original code:

Process 1	Process 2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

Solution 1: Avoid Hold and Wait

Only request a resource when you have none

- I.e., release a resource before requesting another

Process 1	Process 2
lock(x);	lock(y);
A=A+10;	B=B+10;
unlock(x);	**unlock(y);**
lock(y);	lock(x);
B=B+20;	A=A+20;
unlock(y);	**unlock(x);**
lock(x);	**lock(y);**
A=A+30;	B=B+30;
unlock (x);	unlock(y);

Original code:

Process 1	Process 2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

Never hold x when want y:

- Works in many cases
- But you cannot maintain a relationship between x and y

Solution 2: Avoid Hold and Wait

Acquire all resources at once

- E.g., use a single lock to protect all data
- Having fewer locks is called lock coarsening

Process 1	Process 2
lock(z);	**lock(z);**
A=A+10;	B=B+10;
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock (z);	**unlock(z);**

Original code:

Process 1	Process 2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

Problem: low concurrency

- All processes accessing A or B cannot run at the same time
- Even if they don't access both variables!

Prevention: Adding Pre-emption

- Locks cannot be pre-empted but other pre-emptive methods are possible
- Strategy: pre-empt resources
- Example:
  - If process A is waiting for a resource held by process B, then take the resource from B and give it to A
- Problems:
  - Only works for some resources
    - E.g., CPU and memory (using virtual memory)
  - Not possible if a resource cannot be saved and restored
    - Otherwise, taking away a lock causes issues
  - Also, there is an overhead cost for “pre-empt” and “restore”

Prevention: Eliminate Circular Waits

Strategy: Impose an ordering on resources

- Processes must acquire the highest ranked resource first

Process 1	Process 2
lock(x);	**lock(x);**
lock(y);	lock(y);
A=A+10;	B=B+10;
B=B+20;	A=A+20;
A = A+B;	A=A+B;
unlock(y);	**unlock(x);**
A=A+30;	B=B+30;
unlock (x);	unlock(y);

Original code:

Process 1	Process 2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x)	unlock(y);

Locks are always acquired in the same order

- We have eliminated the circular dependency
- Means you will need to lock a resource for a longer period

Preventing Circular Wait: Lock Hierarchy

Strategy: Define an ordering of <u>all</u> locks in your program

- Always acquire locks in that order

Problem: Sometimes you do not know the order that the events will be used

- Recall our code for transferring money from 1 account to another

```
transfer(acc1, acc2, amount){
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
}
```

How do we know the global order?

- ➢ Need extra code to find this out and then acquire them In the right order
- ➢ It could get worse

Lock Hierarchy Problems

Solution 1.1:

- Order based on hash code of variable

Problem?

- What about same account with the same hash code?

```
transfer(acc1, acc2, amount){
    acc1Hash = hashCode(acc1);
    acc2Hash = hashCode(acc2);
    if (acc1Hash < acc2Hash) {
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
    }else{
        acquire(acc2.a_lock);
        acquire(acc1.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc2.a_lock);
        release(acc1.a_lock);
    }
```

Lock Hierarchy Problems

Solution 1.2:

- Order based on hash code of the locked variable
- Deal with ties

```
lock tieLock; // a global lock

transfer(acc1, acc2, amount){
    acc1Hash = hashCode(acc1);
    acc2Hash = hashCode(acc2);
    if (acc1Hash < acc2Hash) {
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
    }else if (acc1Hash > acc2Hash) {
        acquire(acc2.a_lock);
        acquire(acc1.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc2.a_lock);
        release(acc1.a_lock);
    } else {
        acquire(tieLock);
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
        release(tieLock);
    }
}
```

Extra Resources:

Mike Swift Concurrency videos:

- https://www.youtube.com/channel/UCBRYU9uye8e-ZuWQMPBAoYA/videos

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing System & Organisation

Lecture 7: Memory Management

Content

1. Memory management
2. Address space of a process
3. Segmentation
4. Paging

Memory Management

Memory Management

- In multiprogramming systems, the user part of memory is subdivided to accommodate multiple processes
- The task of subdivision is carried out by the operating system and is known as ***memory management***
- Memory needs to be allocated efficiently to pack as many processes into memory as possible

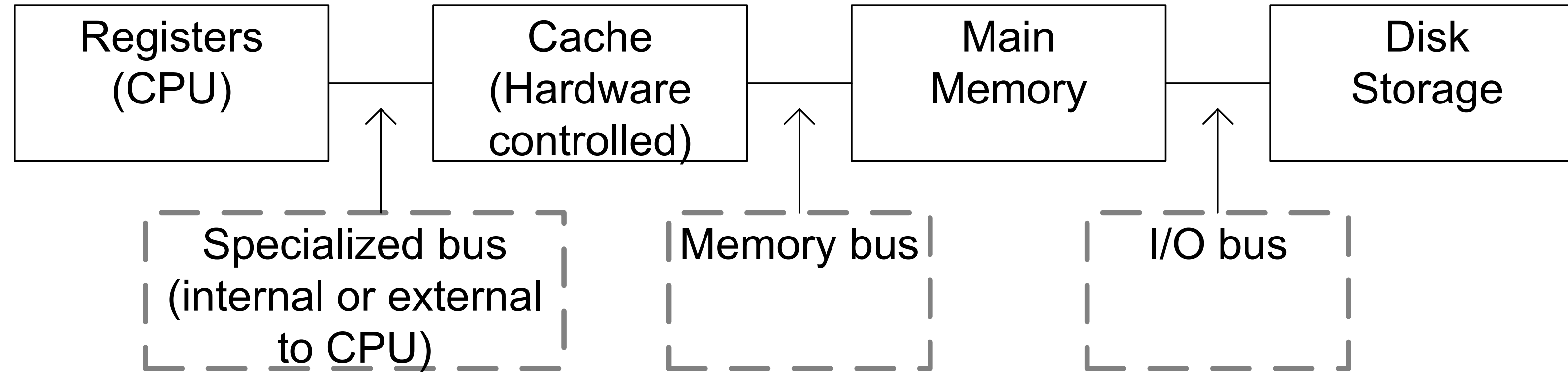
Memory Management Requirements

- **Relocation**
  - Loading dynamically the program into an arbitrary memory space, whose address limits are known only at execution time
- **Protection**
  - Each process should be protected against unwanted interference from other processes
- **Sharing**
  - Any protection mechanism should be flexible enough to allow several processes to access the same portion in the main memory

Memory Organisation

- **Logical organisation**
  - Most programs are organised in modules
    - Some modules are un-modifiable (read only and/or execute only)
    - Others contain data that can be modified
  - The operating system must take care of the possibility of sharing modules across processes
- **Physical organisation**
  - Memory is organised as at least a two-level hierarchy.
  - The OS should hide this fact and should perform the data movement between the main memory and secondary memory without the programmer’s concern

Memory Hierarchy Review



- It is a tradeoff between size, speed and cost
- **Register**
  - Fastest memory element; but small storage; very expensive
- **Cache**
  - Fast and small compared to main memory; acts as a buffer between the CPU and main memory: it contains the most recent used memory locations (*address* and *contents* are recorded here)
- **Main memory** is the RAM of the system
- **Disk storage** - HDD

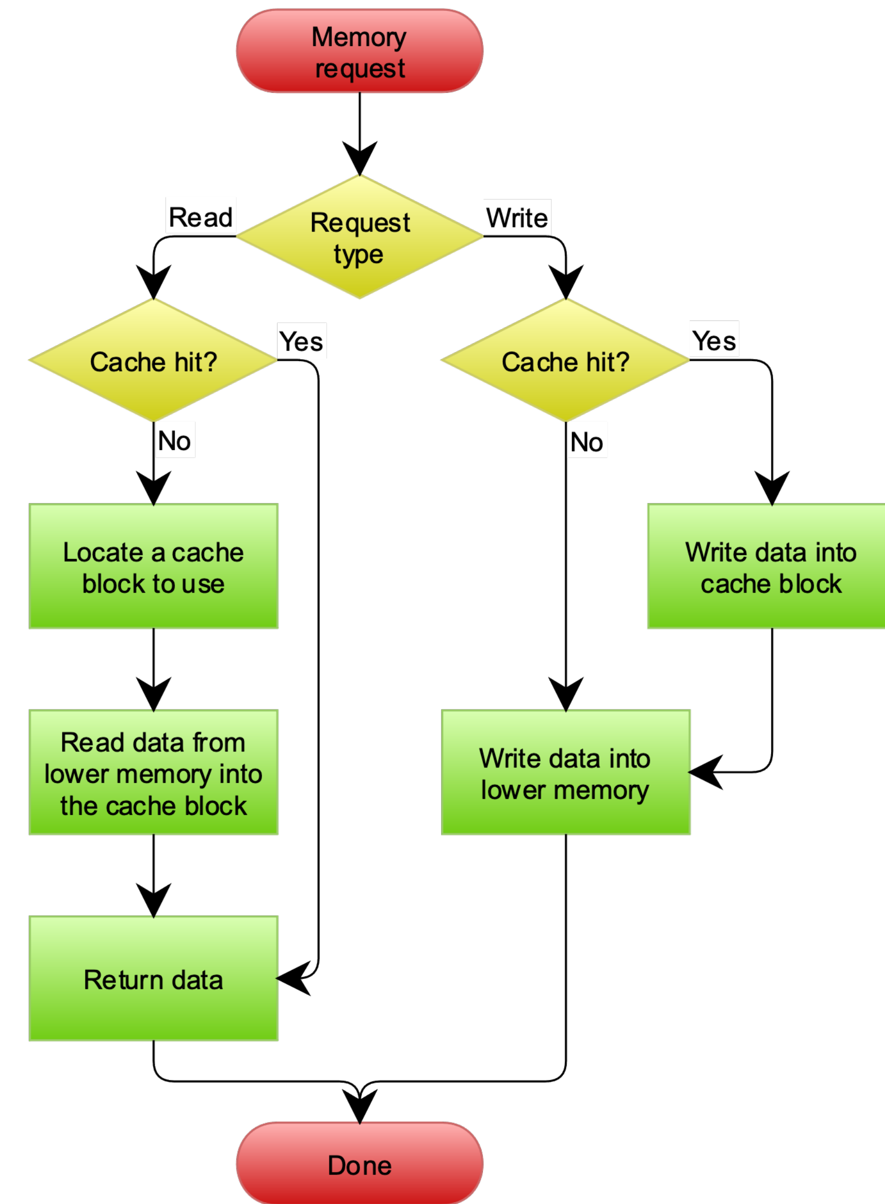
Caching

- Reading from cache is faster than recomputing a result or reading from a slower data store
  - thus, the more requests that can be served from the cache, the faster the system performs.
- When reading data from a lower memory, also store a copy in the cache
  - Future requests for that data can be served faster
- A ***cache hit*** occurs when the requested data can be found in a cache, while a ***cache miss*** occurs when it cannot.

Cache review

- Typical computer applications access data with a high degree of locality of reference:
  - ***Temporal locality***: data is requested that has been recently requested already
  - ***Spatial locality***: data is requested that is stored physically close to data that has already been requested

- When a system writes data to cache, it must at some point write that data to the main memory as well following the ***Write policies:***
  - ***Write-through***: write is done synchronously both to the cache and to main memory
  - ***Write-back***: initially, writing is done only to the cache. The write to main memory is postponed until the modified content is about to be replaced by another cache block.



https://commons.wikimedia.org/w/index.php?curid=27652294

Process Address Space

Process Address Space

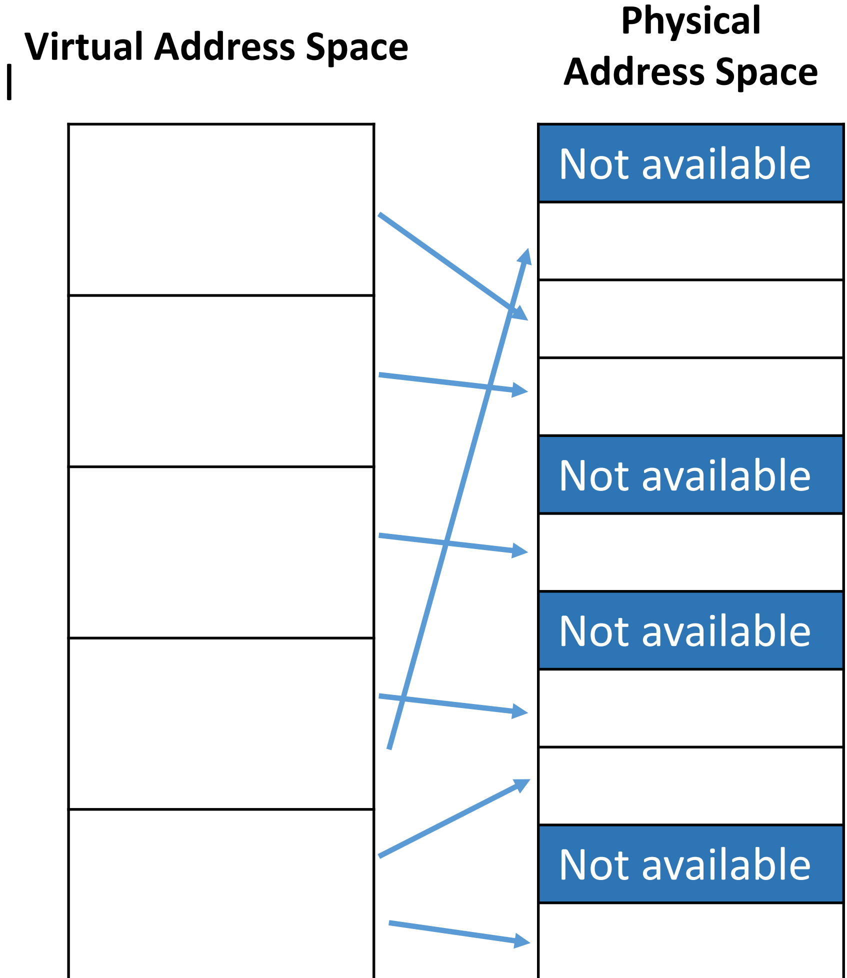
- When accessing memory, a process is said to operate within an ***address space*** (data items are accessible within the range of addresses available to the process)

- The number of bits allocated to specify the address is an ***architectural decision***
  - Many early computers had:
    - 16 bits for address (thus allowing for a space of 64KB of direct addressing -> 2^{16})
    - Then, 32 bits, which allows for 4GB of direct addressing memory space
  - Now most computers had 64 bits for addresses
    - We say that such a system gives a ***virtual address space*** of 16 ExaBytes (16 billion gigabytes)
    - Although, the amount of physical memory in such a system is likely to be less than this)

Address Binding

- An address used in an instruction can point ***anywhere*** in the virtual address space of the process
  - It still must be ***bound*** to a physical memory address
- Programs are made of modules.
- Compilers or assemblers ***do not know where the module will be loaded*** in the physical memory
  - ➢ Virtual addresses must be translated to physical addresses

➢ Address translation can be **dynamic** or **static**.



Static Address Binding

- OS is responsible for managing the memory, so it will give the loader ***a base address where to load the module***
  - The loader **converts** each virtual addresses in the module to absolute physical addresses by adding the the base address
  - This is called ***static binding***

- Simple/Easy to Implement
- But,
  - Once loaded, the code or data of the program cannot be moved into another part of memory without change in the static binding
  - All the processes executing in such a system would share the same physical address space
    - no protection from one another if addressing errors occur
    - even the OS code is exposed to addressing errors

Dynamic Address Binding

- Dynamic address binding:
  - Keeps loaded addresses ***relative*** to the start of a process
- Advantages of dynamic address binding:
  - A given program can ***run anywhere*** in the physical memory and ***can be moved around*** by the operating system
  - All of the addresses that it is using are relative to its own virtual address space, so it is ***unaware of the physical locations*** at which it happens to have been placed
  - It is possible to protect processes from each other and protect the operating system from application processes by a mechanism we employ for isolating the addresses seen by the processes
- Disadvantage:
  - A mechanism is needed to bind the virtual addresses within the loaded instructions to physical addresses when the instructions are executed

Hardware Assisted Relocation and Protection

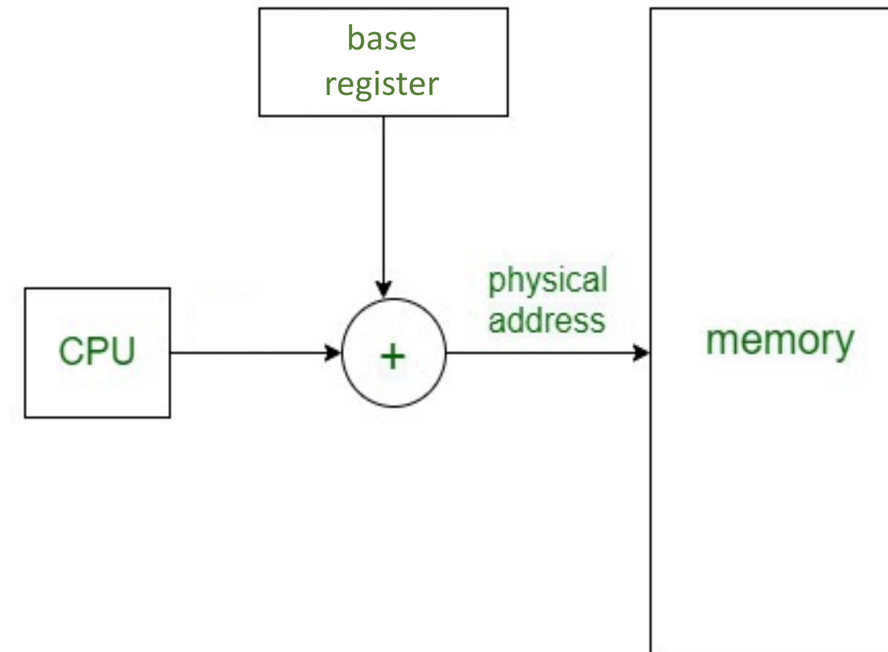
- Dynamic binding must be ***implemented in hardware***, since it introduces translation as part of every memory access

- If the basic requirement for modules is to be held **contiguously** in physical memory and contain addresses relative to their first location:
  - The first location is called the ***base*** of the process
- Suppose that an instruction is fetched and decoded and contains an address reference
  - This address reference is relative to the base, so the value of the base must be added to it (***base + address reference)*** in order to obtain the correct physical address to be sent to the memory controller

Hardware Relocation and Protection

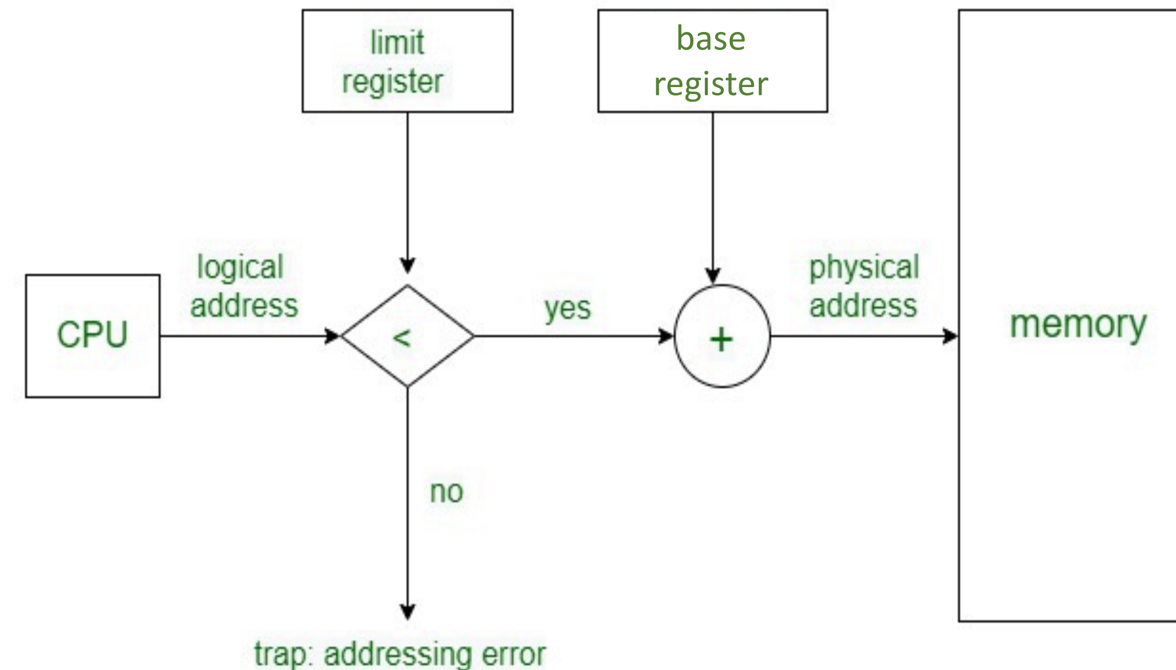
- The simplest form of dynamic binding hardware is a ***base register*** and a memory management unit (MMU*)* to perform the translation
  - The operating system must load the base register as part of setting up the state of a process before passing control to it

- Problem: This approach does not provide any protection between processes:
  - We cannot be sure that a process does not use an address that is not in its space.



Hardware Relocation and Protection

- Solution: combine the relocation and protection functions in one unit
  - By adding a second register (the ***limit register***) that delimits the upper bound of the program in the physical memory



Segmentation

Segmented Virtual Memory

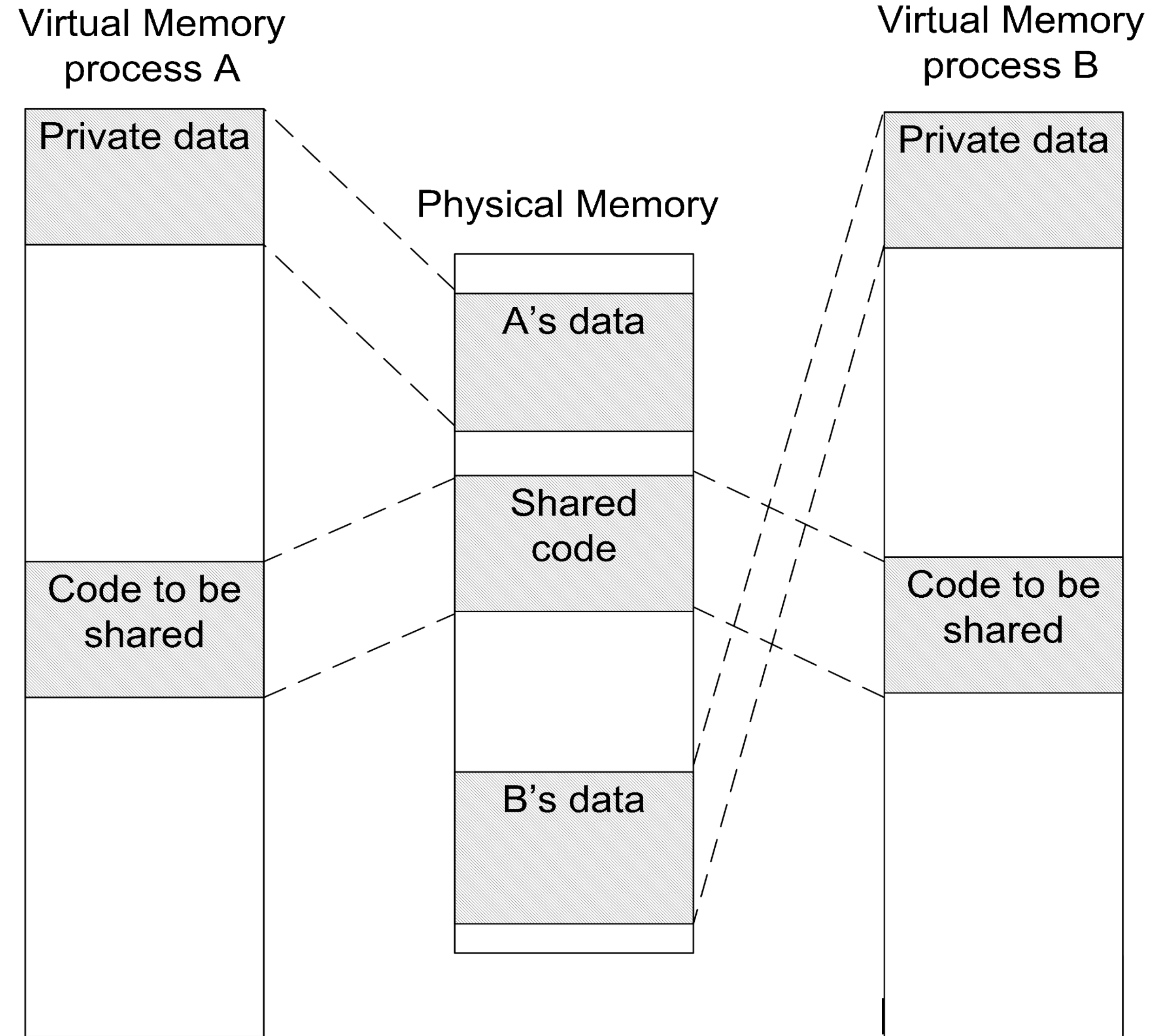
- In practice, it is **not very useful** for a program to occupy a single **contiguous** range of physical addresses

- Such as scheme would prevent two processes from sharing the code
  - i.e., using this scheme, it is difficult to arrange two executions of same program (two processes) to access different data while still being able to share same code

- This can be achieved if the system has two base registers and two limit registers, thus allowing two separate memory ranges or ***segments*** per process

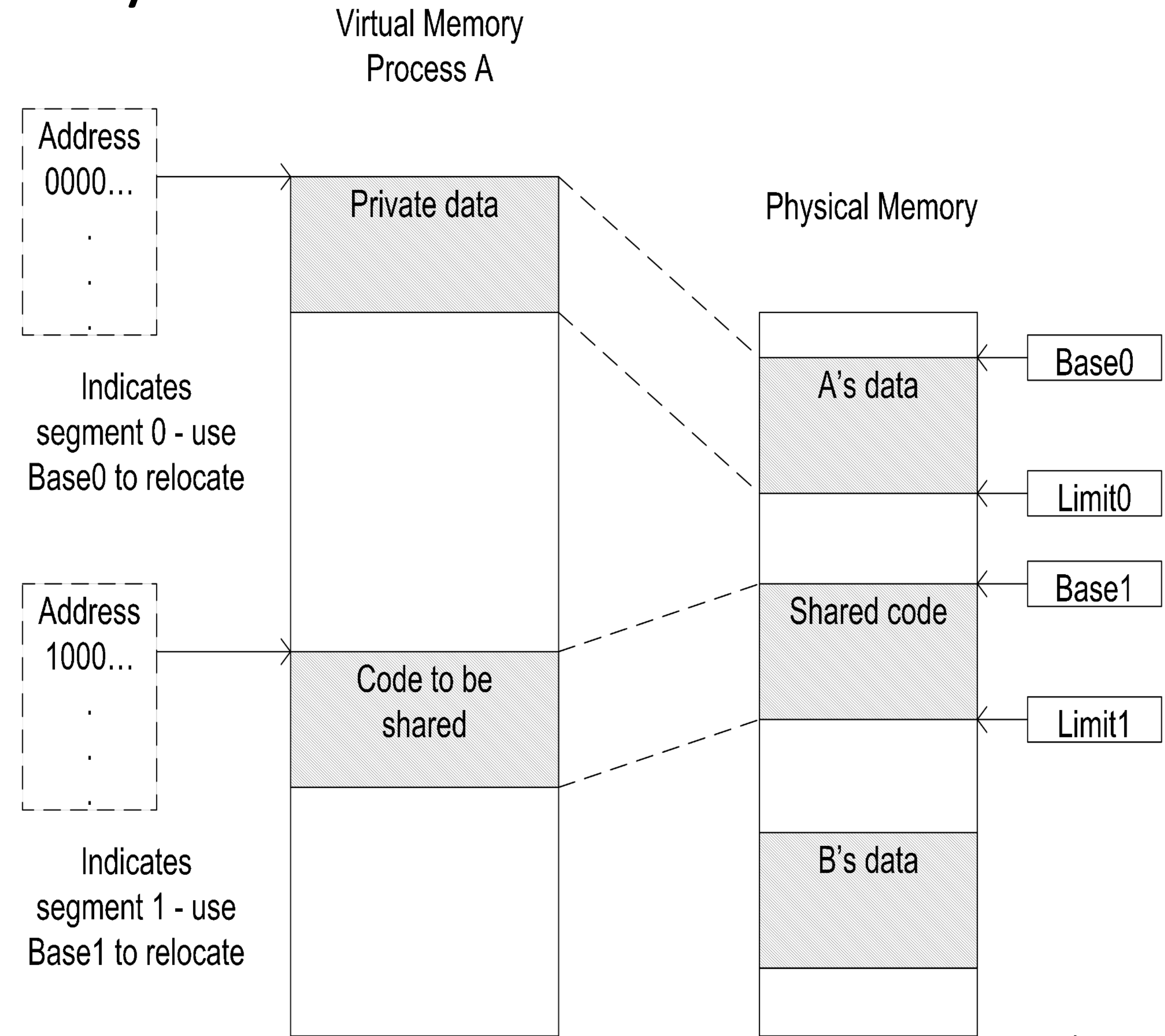
Segmented Virtual Memory

Two processes sharing a code segment but having private data segments



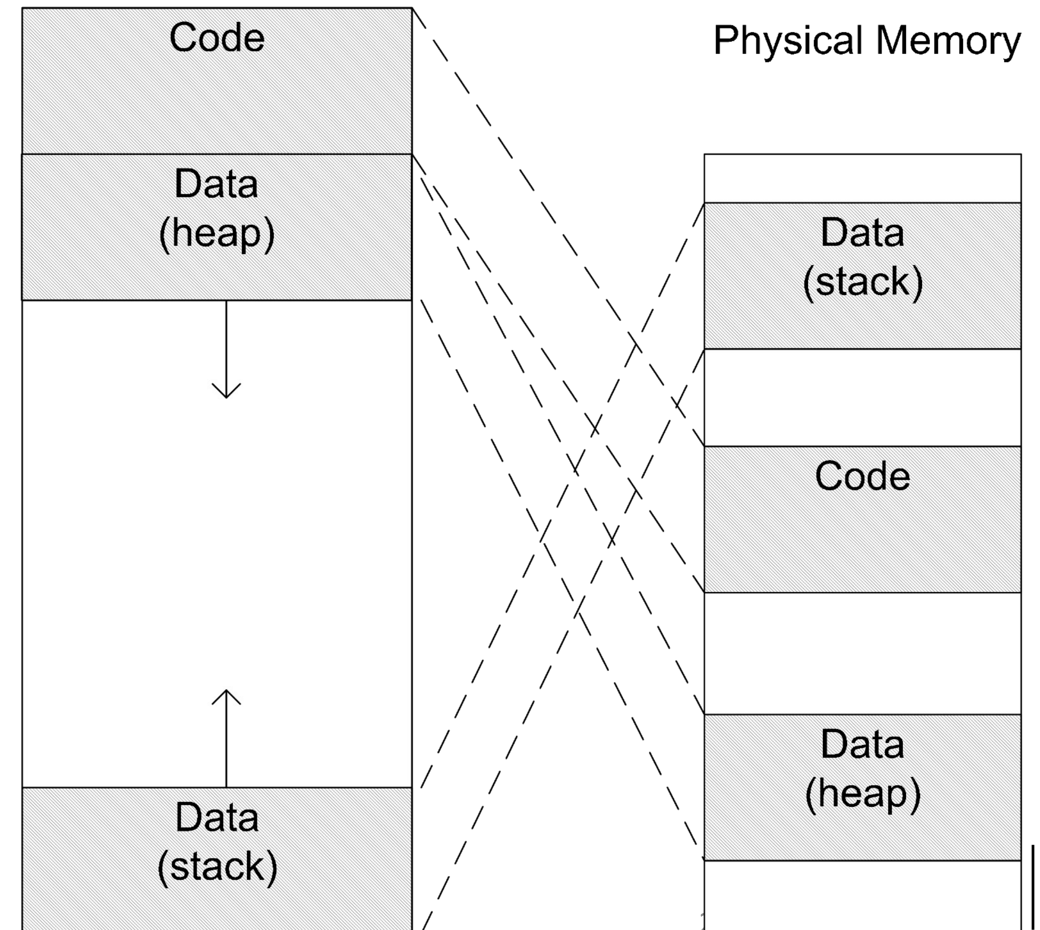
Segmented Virtual Memory

Most significant bit of the virtual address is taken as a ***segment identifier***, with 0 for data segment and 1 for code segment



Segmented Virtual Memory

- Within a single program, it is usual to have separate areas for ***code, stack*** and ***heap***;
- Language systems have conventions on how the virtual address space is arranged
  - Code segment will not grow in size
  - Heap (may be growing)
  - Stack at the top of virtual memory, growing in opposite direction than Heap
- In order to realize the relocation (and protection), three segments would be preferable



Segmented Virtual Addresses

- The segment is the unit of protection and sharing
  - the more we have, the more flexible
- 2 ways to organise segmented address:

1. Virtual address space is split into a ***segment number*** and a ***byte number*** within a segment
   - The number of bits used for segment addressing is usually fixed by the CPU designer

Maximum number of segments is 2^x | Maximum segment size is 2^y

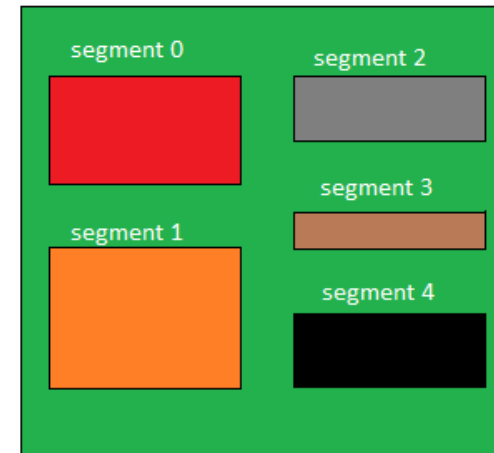
Segment number X bits	Byte offset in segment Y bits

Virtual Address : address field of an instruction

2. The segment number is supplied separated from the offset portion of the address.
   - This is done in X86 processors

Segmented Address Translation

- For dynamic address translation in the operating system
  - Hardware must keep a ***segment table*** for each process in which the location of each segment is recorded
- A process can have many segments, only those currently being used for instruction fetch and operand access need to be in main memory
  - other segments could be held on backing store until they are needed.
- If an address is presented for a segment that is not present in main memory, then the address translation hardware generates an ***addressing exception***.
  - This is handled by the operating system, causing the segment to be fetched into main memory and the mechanism restarted

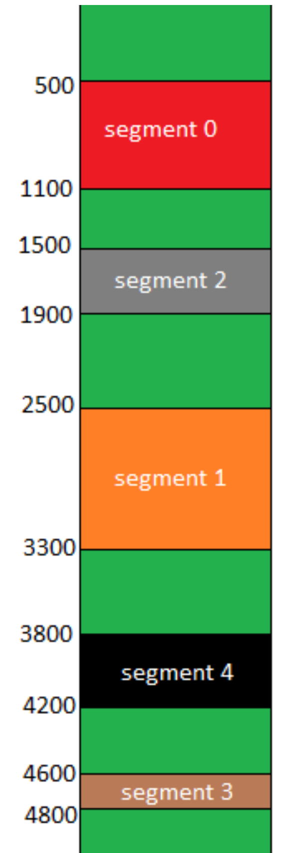


Logical Address Space

Segment Number

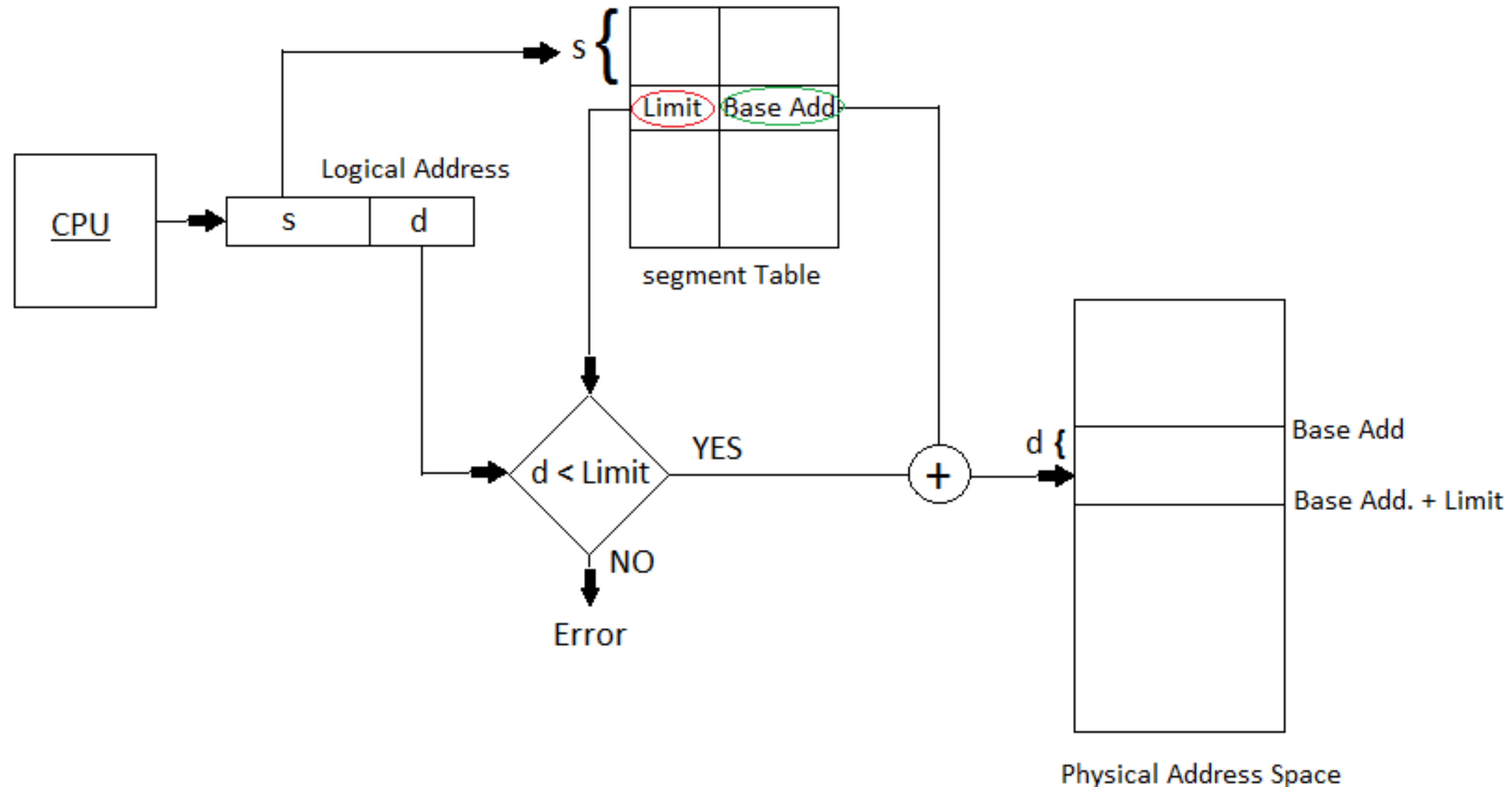
Segment Number	base address	Limit
0	500	600
1	2500	800
2	1500	400
3	4600	200
4	3800	400

Segment Table



Physical Address Space

Address Translation in Segmentation System



s = number of bits to represent the segment
d = number of bits to represent the size of the segment
limit = length of the segment
base add = initial physical address in memory

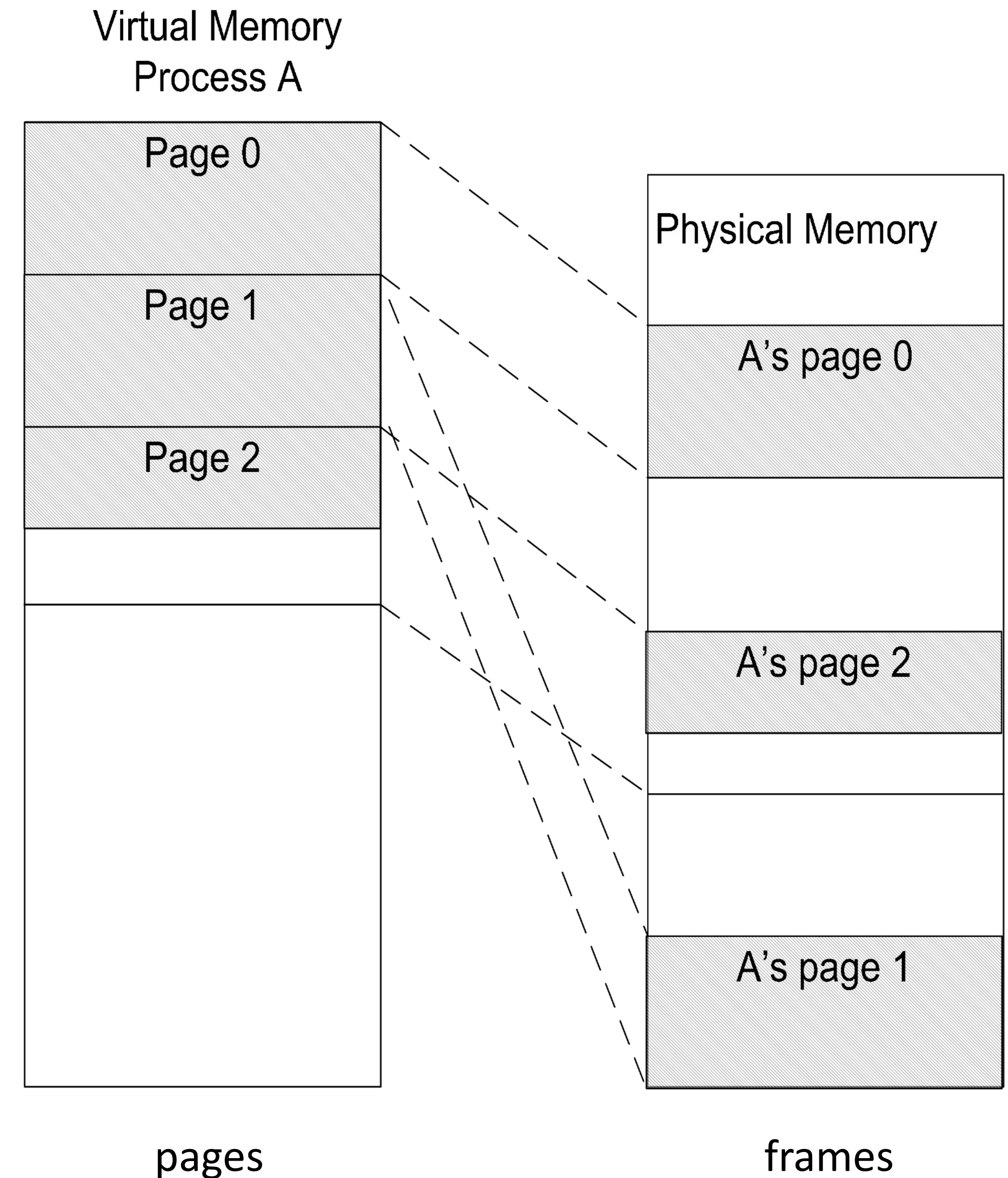
Segmentation Summary

- A process is divided into ***a number of segments*** that do not need to be equal in size
- When a process is brought into the main memory, all of its segments are usually brought into the main memory and ***a process segment table*** is setup.
- **Advantages**:
  - The virtual address space of a process is divided into logically distinct units which correspond to constituent parts of a process
  - Segments are the natural units of access control
    - Rrocesses may have different access rights for different segments and sharing code/data with other processes
- **Disadvantages**:
  - Inconvenient for operating system to manage storage allocation for variable-sized segments
  - After the system has been running for a while, the free memory available can be fragmented
  - ***External fragmentation:*** sometimes, even though the total free memory might be far greater than the size of some segment that must be loaded, there is no single area large enough to load it

Paging

Paged Virtual Memory

- The need to keep each loaded segment contiguous in the physical memory poses a significant disadvantage:
  - It leads to fragmentation
  - It complicates the physical storage allocation problem
- Solution: ***paging***, where blocks of a fixed size are used for memory allocation (so that if there is any free space, it is of the right size)
- Memory is divided into page ***frames***, and the user program is divided into ***pages*** of the same size



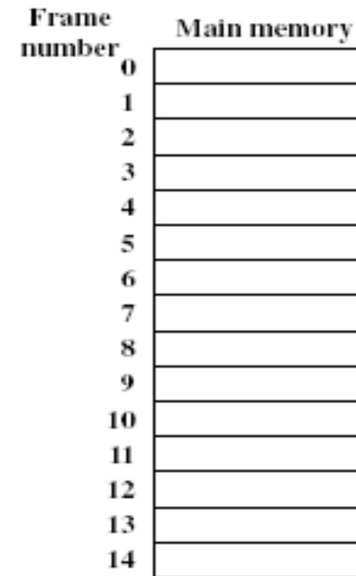
Paged Virtual Memory

- Typical page size is small (1 to 4KB)
  - In paged systems, a process would require many pages
- The limited size of physical memory can cause problems. Therefore,
  - a portion of the disk storage could be used as extension to the main memory (backing store)
  - and the pages of a process may be in the main memory and/or in this backing store
- The operating system ***must manage the two levels of storage and the transfer of pages*** between them
- It must keep a ***page table*** for each process to record information about the pages
  - A ***present bit*** is needed to indicate whether the page is in main memory or not
  - A ***modify bit*** indicates if the page has been altered since last loaded into main memory
  - If not modified, the page does not have to be written to the disk when swapped out

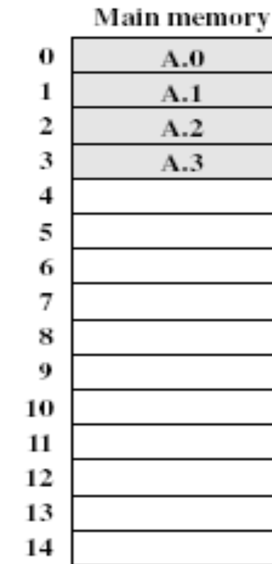
Paging Example

All the processes (A, B, C and D) are stored on disk and are about to be loaded in the memory (by the operating system)

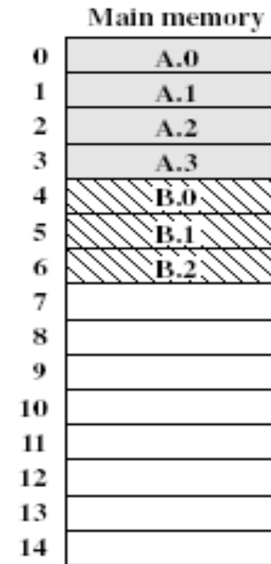
- Process A has four pages
- Process B has three pages
- Process C has four pages
- Process D has five pages



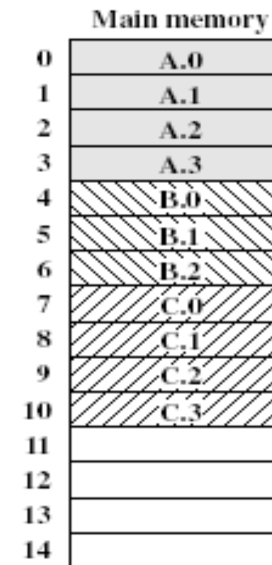
(a) Fifteen Available Frames



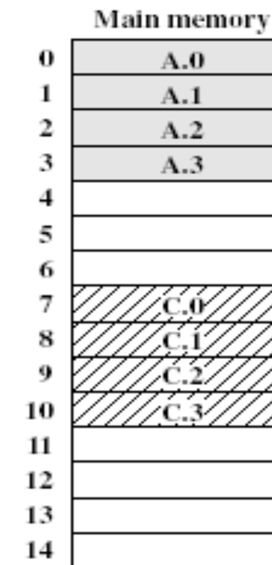
(b) Load Process A



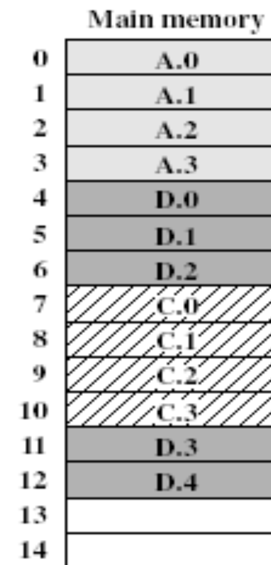
(c) Load Process B



(d) Load Process C



(e) Swap out B



(f) Load Process D

Paging Example

0	0
1	1
2	2
3	3

Process A page table

0	—
1	—
2	—

Process B page table

0	7
1	8
2	9
3	10

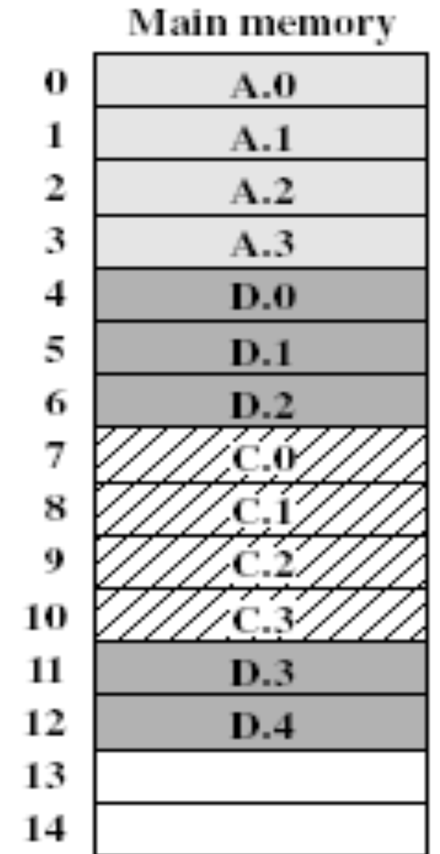
Process C page table

0	4
1	5
2	6
3	11
4	12

Process D page table

13
14

Free frame list

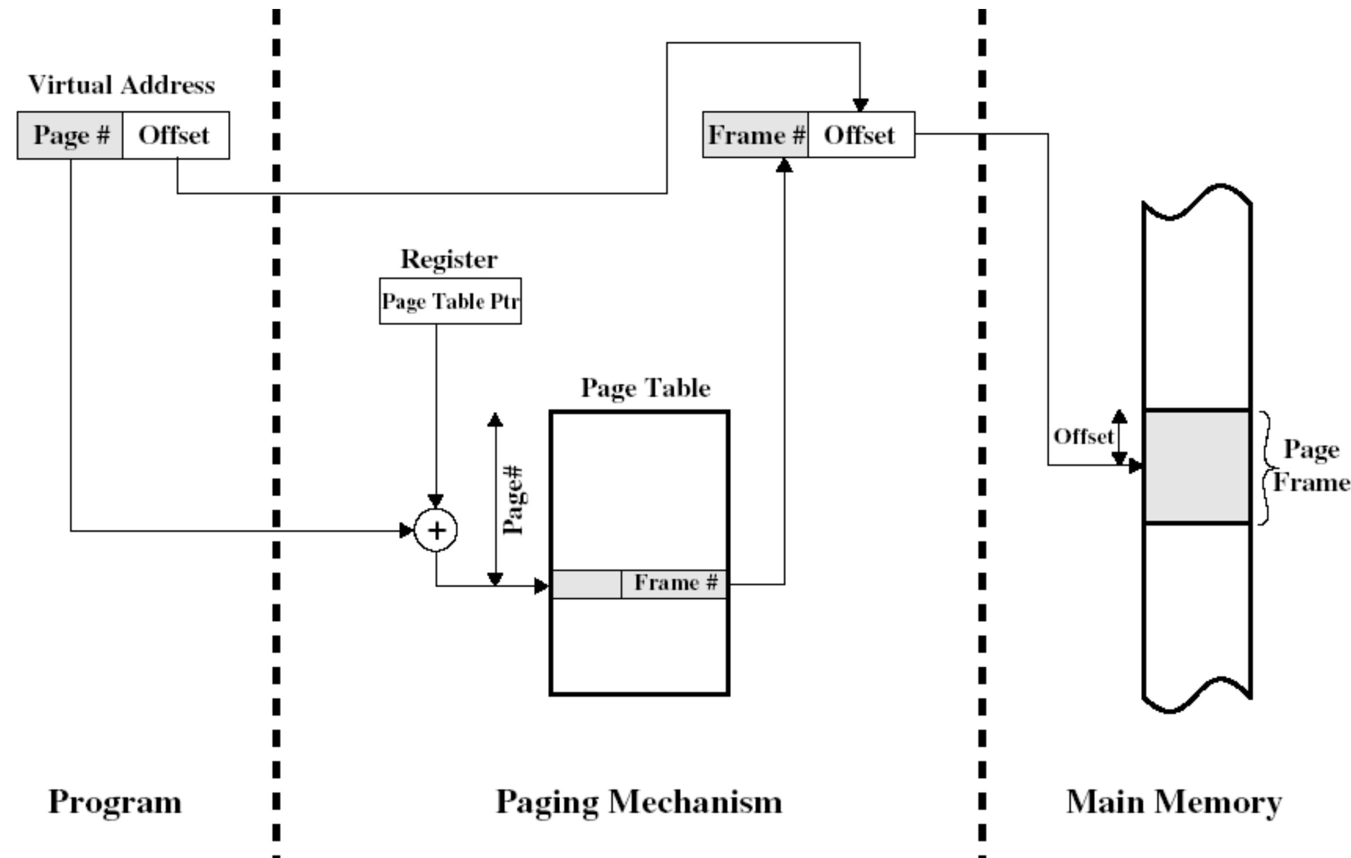


(f) Load Process D

- Various page tables at the time
  - Each Page Table Entry (PTE) contains the number of the frame in main memory (if any) that holds that page
  - In addition, typically, the operating system maintains a list of all frames in main memory that are currently unoccupied and available for pages

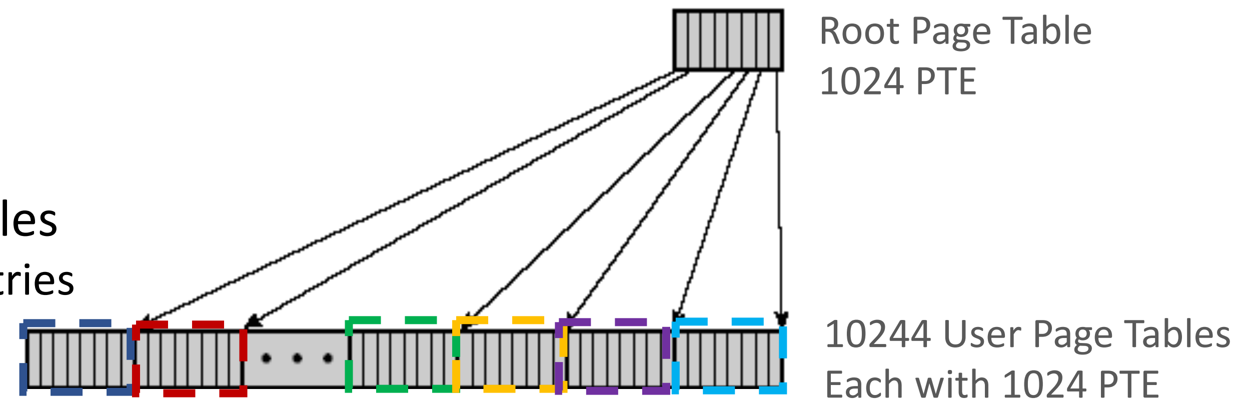
Paged Virtual Memory Address Translation

- Translation of a virtual address (***page*** + offset) into a physical address (***frame*** + offset)
  - using a page table
- Page table is stored in the main memory
  - Each process maintains a pointer in one of its registers, to the page table
- The page number is used to index that table and lookup the corresponding frame number
- Combining the frame number with the offset from the virtual address gives the real physical address



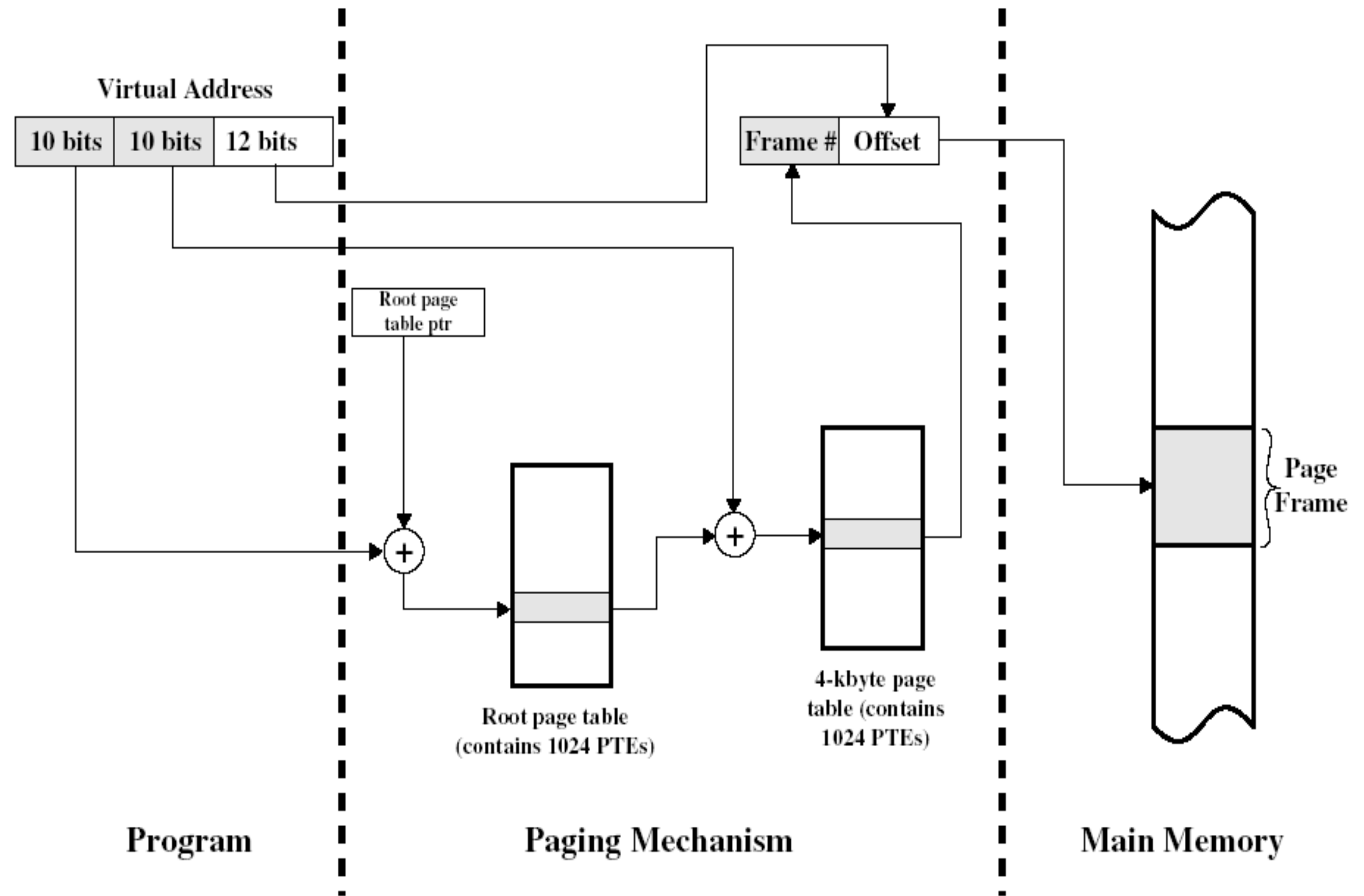
Paged Virtual Memory Address Translation

- Processes could occupy ***huge amounts of virtual memory***
  - E.g., in a 32bit addressing system with pages of size 4KB:
    - 12 bits for offset
    - 20 bits for number of pages
  - This means 2^{20} entries could be in each page table
    - If each entry occupies 4Bytes (32bit address)
    - Then *each page table would take 4MB*
      - ➢ Unacceptably high!
- Solution: a ***two-level scheme*** to organise large page tables
  - Root Page Table with 2^{10} (1024 entries, 4 Bytes each) entries occupying 4KB of main memory
    - Root page always remains in the main memory
  - User Page Tables can reside in either the main memory or in disk



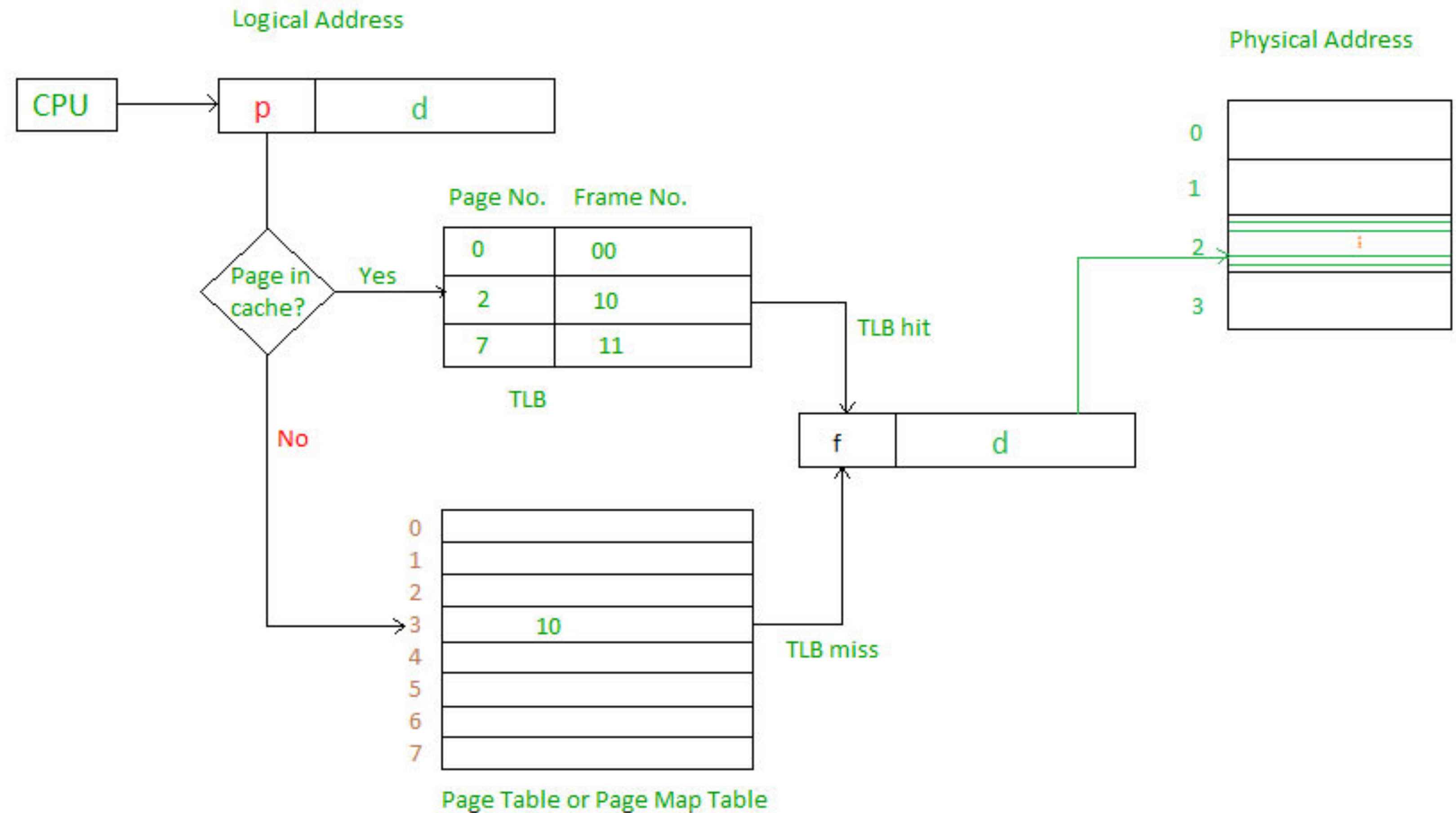
Paged Virtual Memory Address Translation

- The first 10 bits of a virtual address are used to find a PTE to the user page table
- The next 10 bits of the virtual memory address are used find the PTE for the page that is referenced by the virtual address
- Every virtual memory reference causes two physical memory accesses:
  - one to fetch the appropriate User Page Table entry
  - the other to fetch the desired page
- To overcome this, most of the virtual memory schemas make use of a ***special high-speed cache*** for page entries



Translation Lookaside Buffer (TLB)

- **A kind of cache memory:** it contains the page entries that have been most recently used
- TLB is searched for each address reference
- TLB is nearly always present in any processor that utilizes paged or segmented virtual memory
  - Including in most desktops, laptops, and servers.



Translation Lookaside Buffer (TLB)

- The virtual page number is extracted from the virtual address and a lookup is initiated
  - If multiple processes, then special care needs to be taken, so the ***page from one process would not be confused with another's***

- If a match is found (***TLB hit***), then an access check is made, based on the information stored in the flags
  - The physical page base, taken form TLB is appended to the offset from the virtual address to form the complete physical address
  - The flags field will indicate the access rights and other information (i.e. if a write is being attempted to a page that is read only etc)

- If an address reference is made to a page that ***is in the main memory but not in the TLB***, then address translation fails (***TLB miss***) and a new entry in the TLB needs to be created for that page

- If an address reference is made to a page that ***is not in the main memory***, the address translation will fail again. No match will be found in the address table and the addressing hardware will raise an exception, called ***page fault***
  - The operating system will handle this exception

Paging Summary

- **Advantages** – by using fixed size pages in virtual address space and fixed size pages in physical address space, it ***addresses some of the problems with segmentation***:
  - ***External fragmentatio***n is no longer a problem (all frames in physical memory are same size)
  - Transfers to/from disks can be performed at granularity of individual pages

- **Disadvantages**
  - The page size is a choice made by CPU or OS designer
    - It may ***not fit the size of program data structures*** and lead to internal fragmentation in which storage allocation request must be rounded to an integral number of pages
  - There may be no correspondence between ***page protection settings*** and ***application data structures***
    - If two processes are to share data structures, they may do so at the level of sharing entire pages
  - Requiring page tables per process , it is likely that the OS require **more storage** for its internal data structures

References

- “Operating Systems”, William Stallings, ISBN 0-13-032986-X
- “Operating Systems”, Jean Bacon and Tim Harris, ISBN 0-321-11789-1

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing System & Organisation

Lecture 8: Device Management

Content

- Device management
- Device Communication Approaches
- Buffering

Device Management

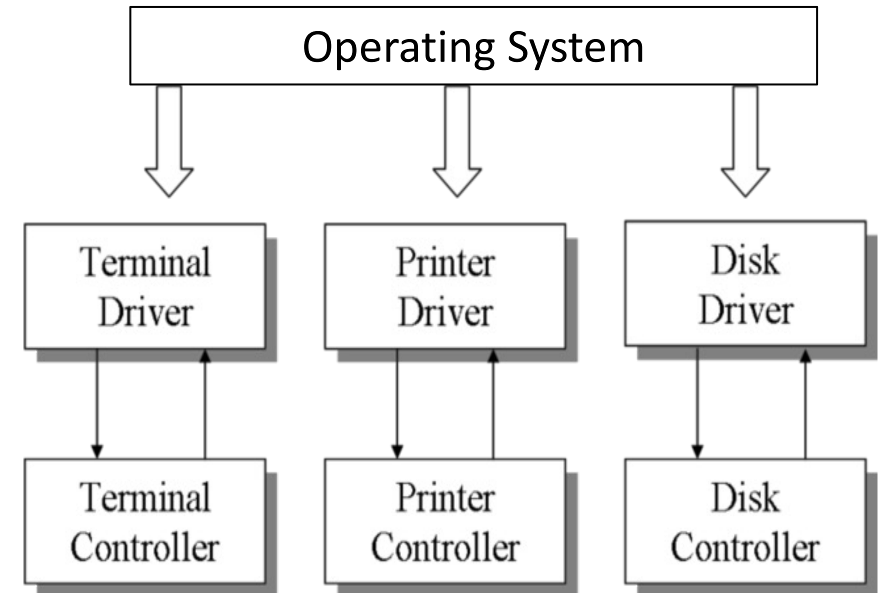
- Device management is the process of managing the **implementation**, **operation** and **maintenance** of physical and/or virtual devices.

- It is a broad term that includes **various administrative tools and processes** for the maintenance and upkeep of a computing, network, mobile and/or virtual device.

- The status of any computing device (internal or external), may be either **free** or **busy**.
  - If a device requested by a process is free at a specific instant of time, the operating system allocates it to the process.

Device Management

- The Operating System manages the devices with the help of:
  - **Device controllers:** hardware components that contain some buffer registers to store the data temporarily.
    - E.g., disk controller, printer controller and a terminal controller
  - **Device drivers:** software programs that are used by an operating system to control the functioning of various devices in a uniform manner.



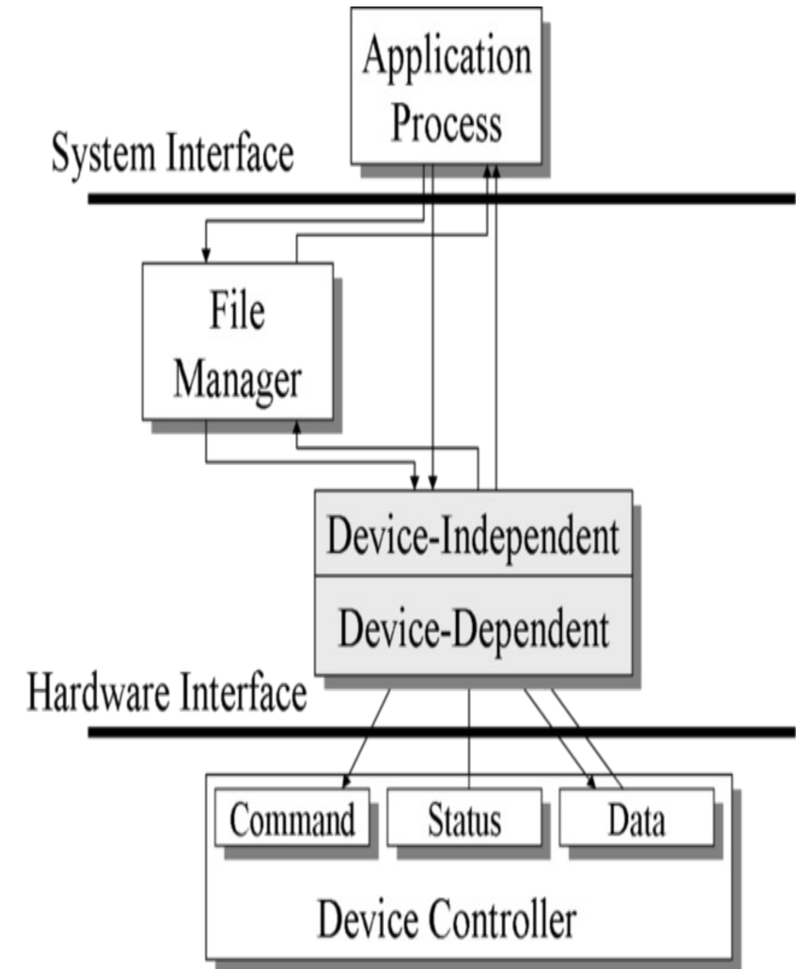
Device Management

- The device controller used in a device management operation includes three different registers: **command**, **status**, and **data**.

- The other major responsibility of the device management function is to implement **Application Programming Interfaces (APIs)**.

- Each device controller is **specific to a particular device**
  - ➢ the device driver implementation will be device specific

- Why?
  - To provide correct commands to the controller
  - To interpret the Controller Status Register (CSR) correctly
  - To transfer data to and from device controller data registers as required for correct device operation



Device Communication Approaches

- A computer must have a way of detecting the arrival of **any type of input**

- There are various ways to enable I/O devices to communicate with the processor:
  - Polling
  - Interrupts
  - Direct I/O
  - Memory Mapped I/O

Polling

- Implementation
  - ***Periodically checking status*** of the device to see if it is time for the next I/O operation
  - I/O device simply puts the information in a Status register, and the processor must come and get the information.

- Efficiency
  - ***Simplest*** way for an I/O device to communicate with the processor.
  - ***Inefficient method***: most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program.
    - ➢ Much of the processor's time is wasted on unnecessary polls.

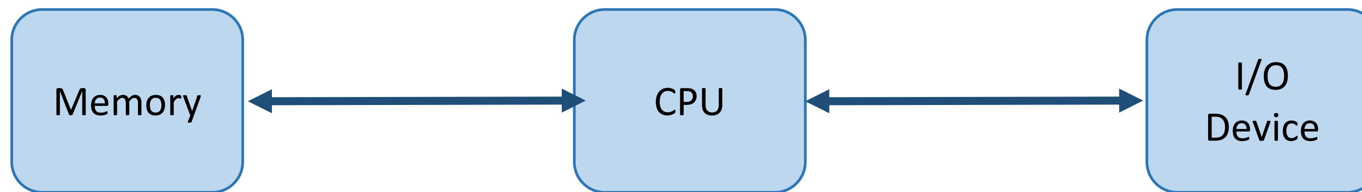
Interrupts

- Implementation
  - A device controller puts out an ***interrupt signal*** when it needs CPU's attention
  - When CPU receives an interrupt, it ***saves its current state*** and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events).
  - When the interrupting device has been dealt with, the ***CPU continues with its original task*** as if it had never been interrupted.

- Efficiency
  - Interrupts allow the processor to deal with events that can happen at any time.
  - Interrupts remove the need for the CPU to constantly check the Controller Status register.

Direct I/O

- Implementation
  - Uses software which explicitly transfers data to/from the **controller's data registers**
    - Separate I/O and memory address spaces.
    - The control indicates whether address information is for memory or I/O.



- Efficiency
  - Reduced CPU utilisation (no caches or buffers)

Memory Mapped I/O

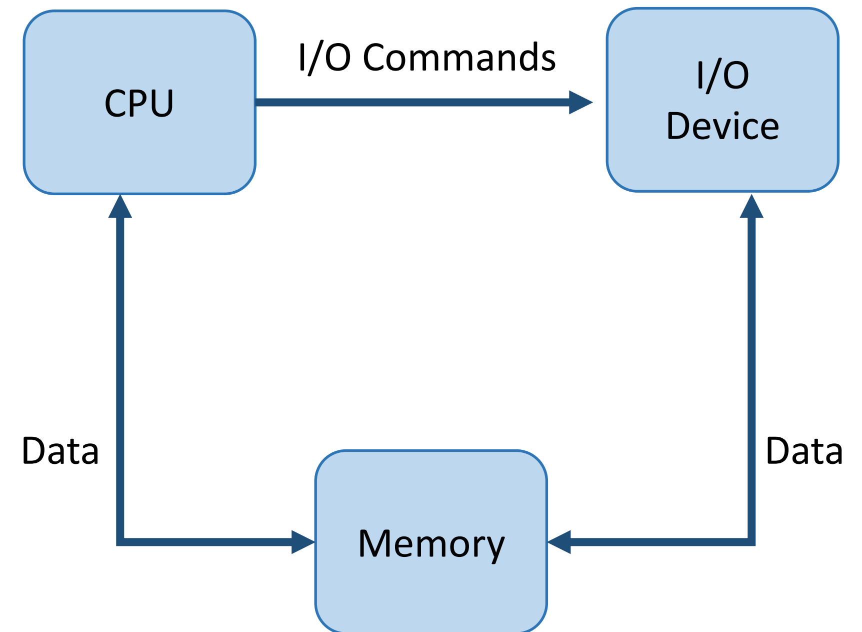
Implementation

- Direct connection between I/O device and certain main memory locations so that I/O device can **transfer block of data** to/from memory without going through CPU.

- OS allocates buffer in memory to the I/O device to send data to the CPU.
  - I/O device operates **asynchronously** with CPU
  - Interrupts CPU when finished.

Efficiency

- Memory mapped IO is ideal for most high-speed I/O devices like disks, communication interfaces.



Design Objectives

- **Efficiency**
  - Most I/O devices are extremely slow compared with the processor and main memory
    - Buffering is one way to deal with this issue

- **Generality**
  - It is desirable to handle all devices in a uniform and consistent manner
    - In the way user processes see the devices
    - In the way the Operating System manages the I/O devices and operations

Buffering

Buffering is the technique by which the device manager can keep slower I/O devices busy during times when a process is not requiring I/O operations.

- **Input buffering:** having the input device read information into the primary memory before the process requests it.
- **Output buffering:** saving information in memory and then writing it to the device while the process continues execution

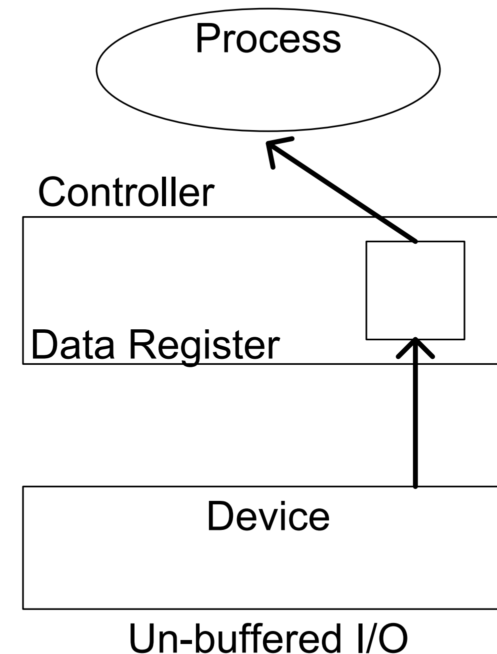
Hardware Level Buffering

Consider a simple character device controller that reads a single byte from a router for each input operation.

Normal operation:

1. Read occurs
2. The driver passes a read command to the controller
3. The controller instructs the device to put the next character into one-byte data controller's register
4. The process calling for the byte **waits** for the operation to complete

- then retrieves the character from the data register

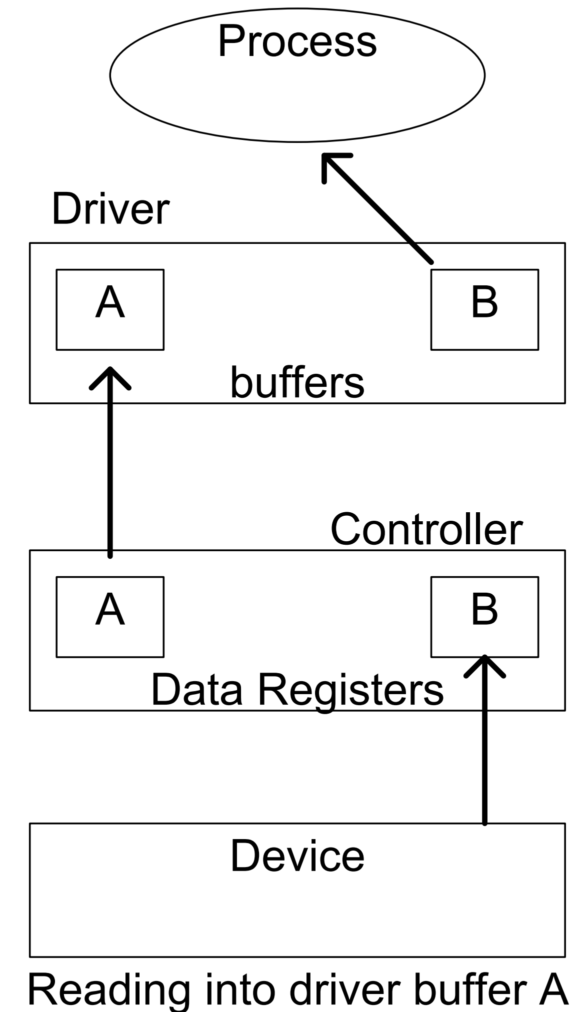


Buffered operation:

- The next character to be read by the process has already been placed into the data register, even though the process has not yet called for the read operation
- Adding a hardware buffer to the controller **decreases** the amount of time the process has to **wait**

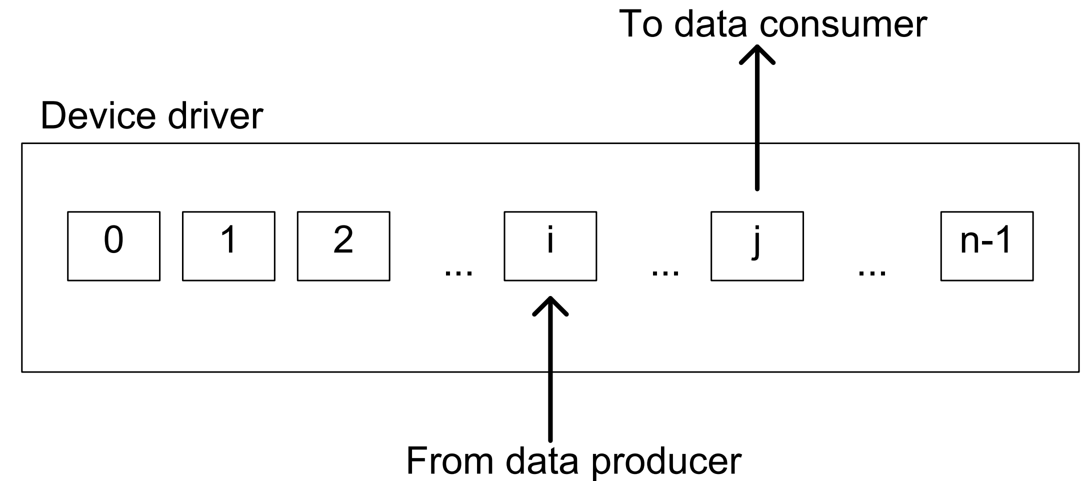
Driver Level Buffering

- This is generally called **double buffering**
  - One buffer is for the driver to store the data while waiting for the higher layers to read it
  - The other buffer is to store data from the lower-level module



Using Multiple Buffers

- The number of buffers is extended from two to n
- The data producer is writing into buffer i while the data consumer is reading form buffer j
  - In this configuration:
    - If i<j: buffers [j+1, n-1] and [0, i-1] are full
    - If j<i: buffers [j+1, i-1] are full



- This is known as ***circular buffering***

References

- “Operating Systems”, William Stallings, ISBN 0-13-032986-X
- “Operating Systems – A modern perspective”, Garry Nutt, ISBN 0-8053-1295-1

Ollscoil na Gaillimhe
University of Galway

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

CT213 Computing System & Organisation

Lecture 9: Module Review + Exam Structure

Topics Covered

- Computer Systems
- Programming Models
- System software and Operating Systems
- Process Management
- CPU Manager
- Process Synchronisation
- Memory Management
- Device Management
- File Management

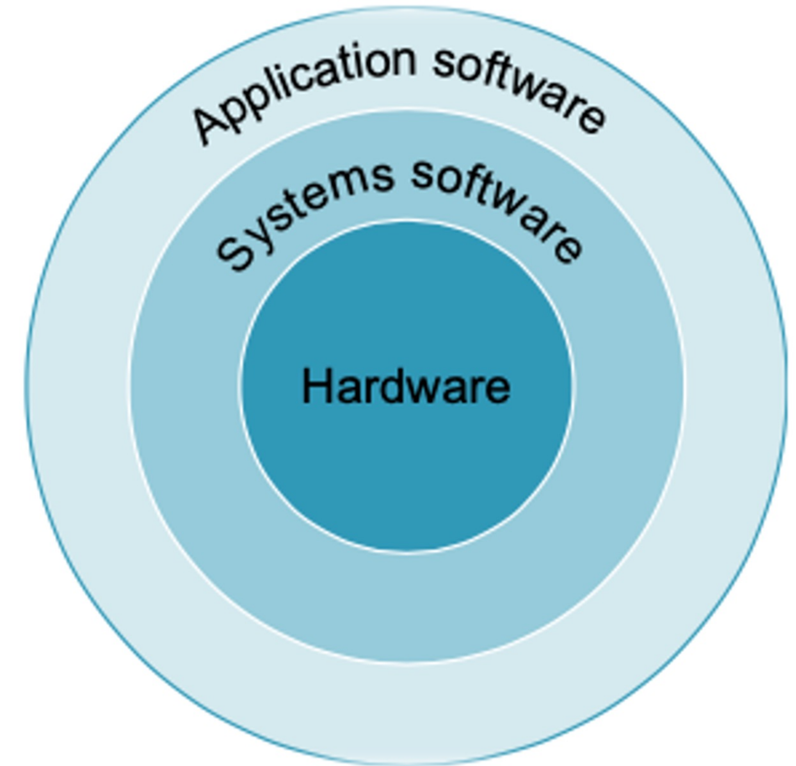
Computer Systems

Computer Systems

- Application Software that provides services that are commonly useful.

- Operating system interfaces between a user's program and the hardware and provides a variety of services and supervisory functions.

- Hardware performs the tasks.



Dr Takfarinas Saber <takfarinas.saber@universityofgalway.ie>

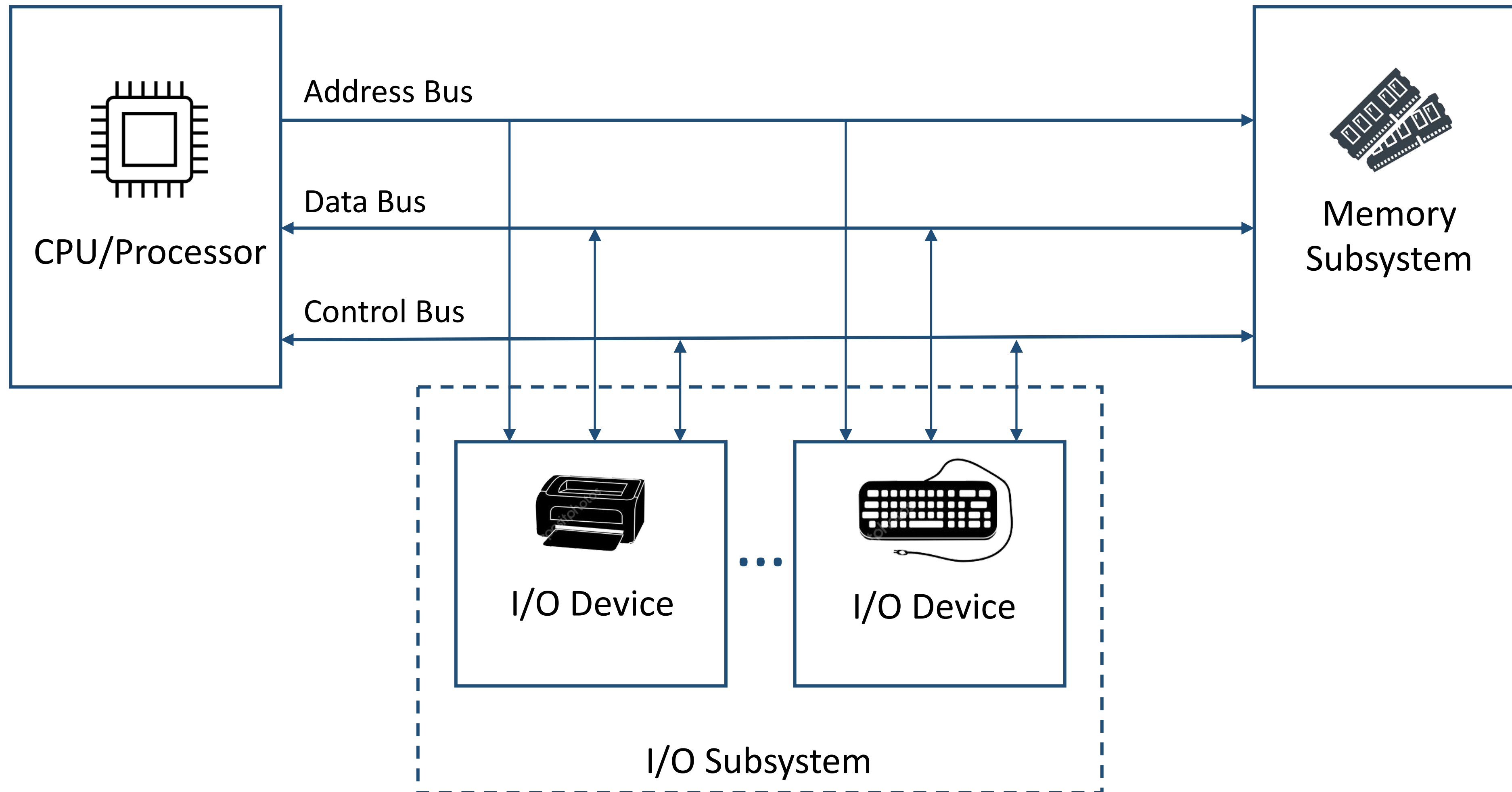
Seven Great Ideas in Computer Organisation

Computer Systems & CPU

- Review of **Computer Organisation**
  - Generic organization
  - Components: buses, CPU, memory, I/O subsystem
- **Programs**
  - Development tools
  - High level programming languages
  - Assembly programming language
  - Machine language
- **Operating System**
  - Multiprogramming
  - Protection
  - Privileged mode

How Does a Computer Look Like?

Basic Computer Organisation



Programming Models

What's a programming language??

– Programming languages are a notation for describing computations, both to people and to machines.

– Must be translated before execution

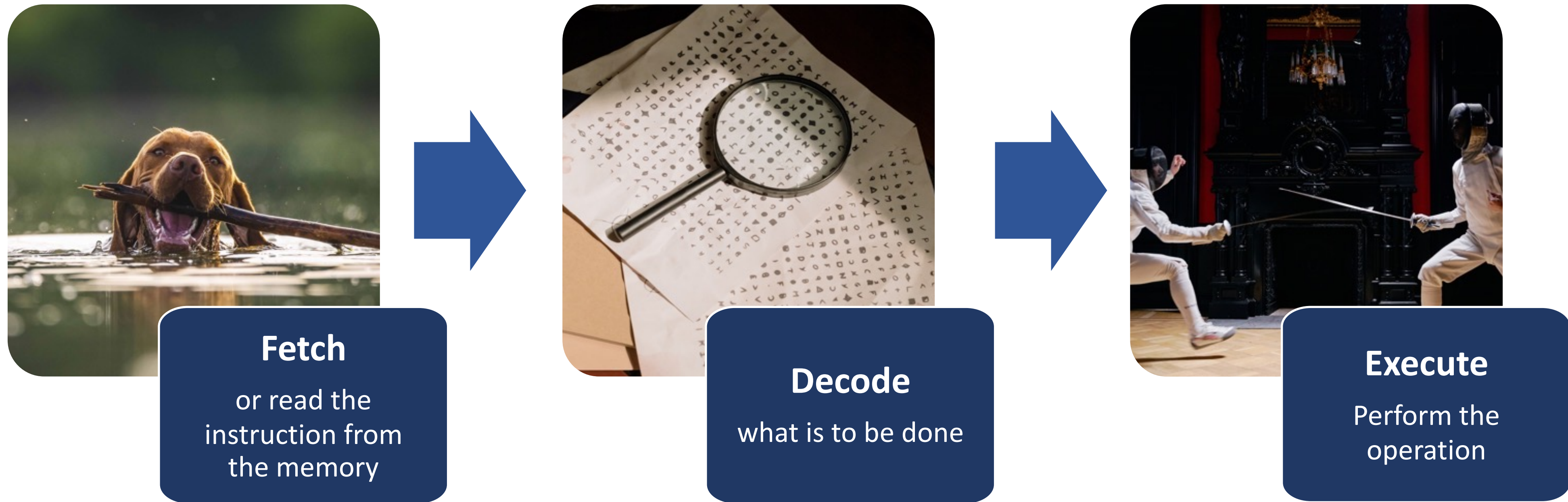
– A compiler is a program that handles the translation from a source program into a target program

newPos = pos + time * 60 →

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

The Processor - Instruction Cycles

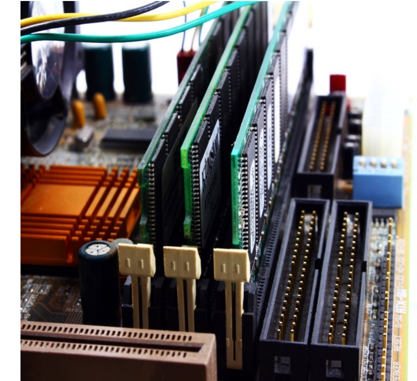
- The instruction cycle is the **procedure of processing an instruction** by the microprocessor:



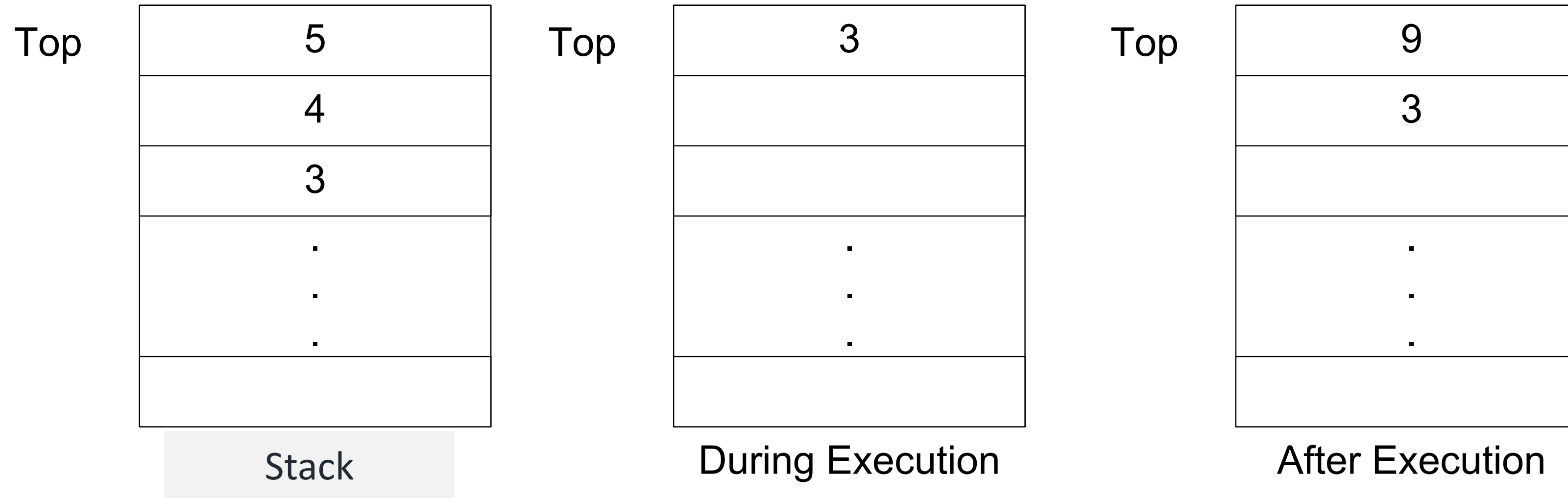
- Each of the functions fetch -> decode -> execute consist of a sequence of one or more operations inside the CPU (and interaction with the subsystems)

Types of Instructions

- **Data Transfer** Instructions
  - Operations that **move data** from one place to another
  - These instructions **don't modify** the data, they just copy it to the destination
- **Data Operation** Instructions
  - Instructions do modify their data values
  - They typically perform some operation (e.g., +/-/*) using one or two data values (operands) and store the result
- **Program Control** Instructions
  - **Jump** or **branch** instructions used to **go in another part** of the program; Jumps can be **absolute** or **conditional** (e.g., if then else)
  - Instructions that can generate **interrupts** (software interrupts)



Instructions in a Stack Based Architecture



- Get their operands from the stack and write their results to the stack
- Advantage - Program code takes little memory (no need to specify the address of the operands in memory or registers)
  Push is one exception, because it needs to specify the operand (either as constant or address)

General Purpose Register Architecture Architecture

- GPR instructions need to **specify:**
  - **the register** that hold their **input operands**
  - and the register that will hold the **result**

	Register File
Register 0	data
Register 1	data
Register 2	data
	. . .
Register n	data

- The most common format is the **three operands instruction format**
  - E.g., **ADD r1, r2, r3** instructs the processor to read the contents of r2 and r3, add them together and write the result in r1

- Instructions having two or one input are also present in GPR architecture

Comparing Stack based and GPR Architectures

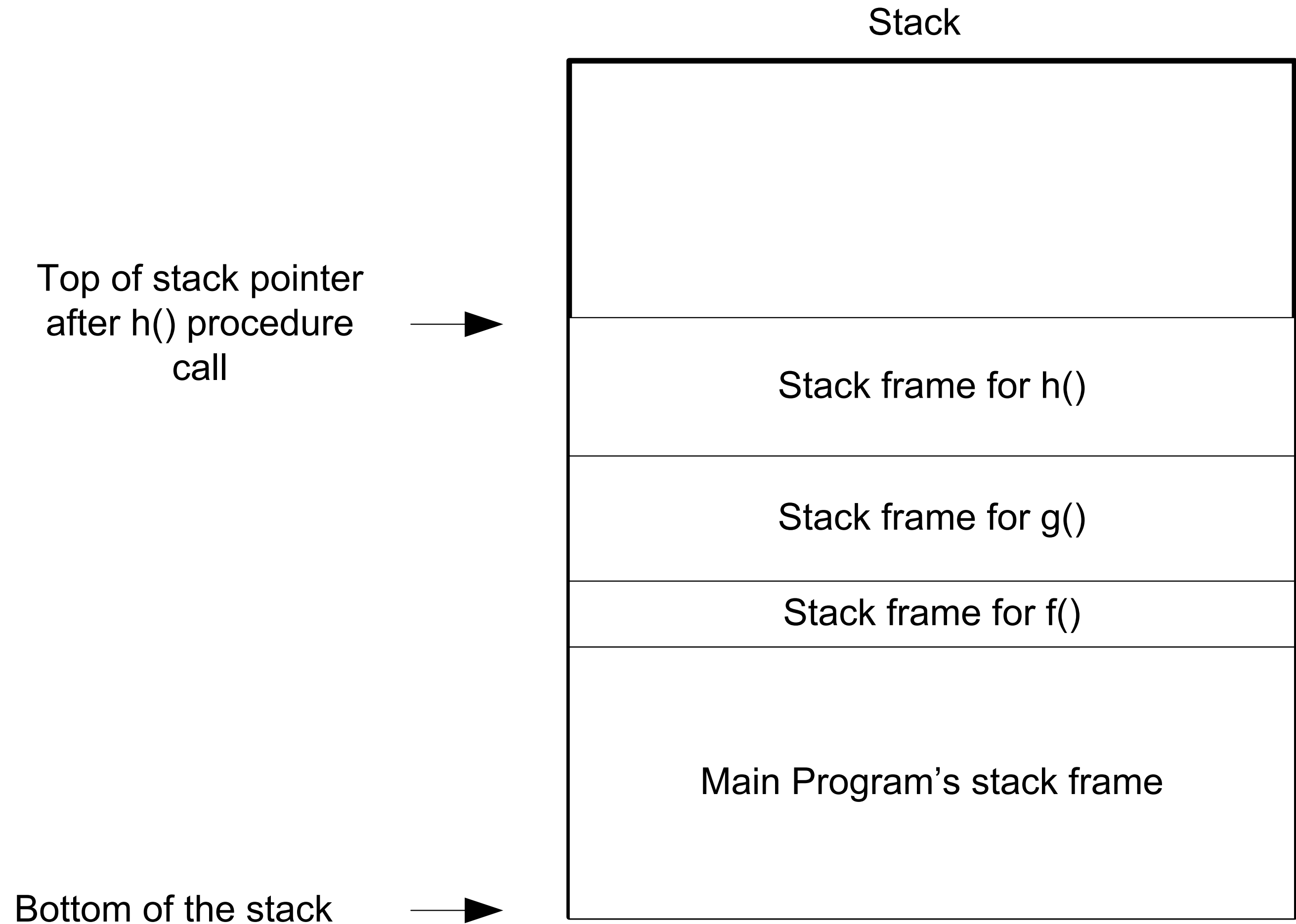
- **Stack-based architectures**
  - Instructions take **fewer bits** to encode
  - **Reduced amount of memory** taken up by programs
  - Manages the **use of register automatically** (no need for programmer intervention)
  - Instruction set does not change if size of register file has changed

- **General Purpose Register Architecture architectures**
  - With evolution of technology, the amount of space taken up by a program is less important
  - Compilers for GPR architectures achieve **better performance** with a given number of general purpose registers than those on stack-based architectures with same number of registers
    - The compiler can **choose which values to keep** (cache) in register file at any time

- Stack based processor are still attractive for certain embedded systems. GPR architectures are used by modern computers (workstations, PCs, etc.)

Using Stacks to Implement Procedure Calls

- Nested procedure calls:
  - main program calls function f(),
  - function f() calls function g(),
  - function g() calls function h()

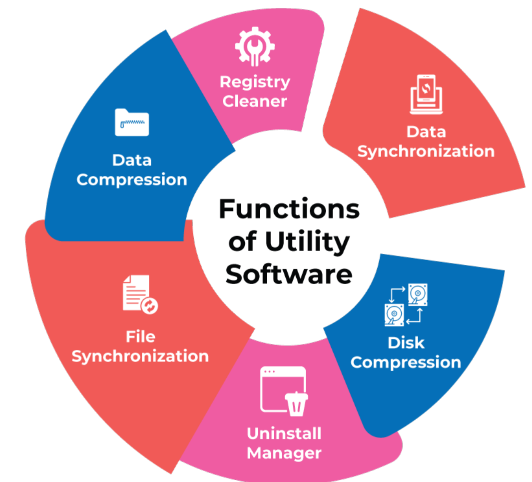


System Software
& Operating Systems

System Software

- Programs dedicated to **managing the computer**
- System software is a software that **provides a platform** to other software.

- **System software** provides a general programming environment
- There are two main types of system software
    1. Operating System
    2. Utility Software



Operating System (OS)

- Provides functions used by the **application software**
- Provides the mechanisms for application software to **share** the hardware in an orderly fashion to:
  - **Increase the overall performance**
  - **Ensure security**
- Interacts directly with the hardware to provide an interface to other software to use system's resources
  - It is application-domain independent
  - Provides resource **abstraction**
  - Provides resources **sharing**

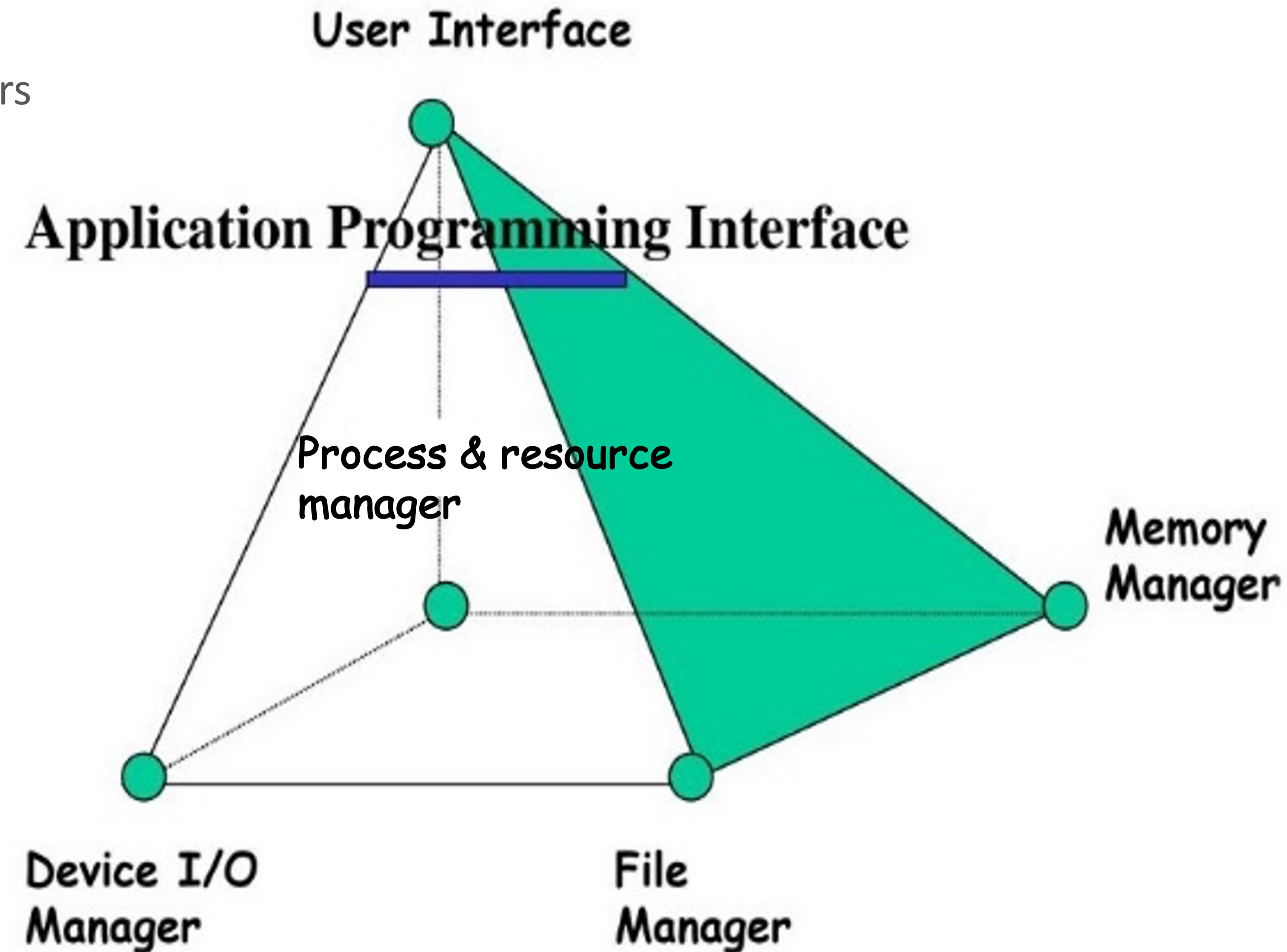
OS Organisation

Process and resource manager
- It uses the abstractions provided by the other managers
- Handles resource allocation

Memory manager
- It is classically a separate part of the operating system
- Beside other functions, it is in charge with the implementation of the virtual memory

File manager
- abstracts device I/O operations into a relatively simple operation

Device manager
- handles the details of reading and writing the physical devices
- implemented within device driver



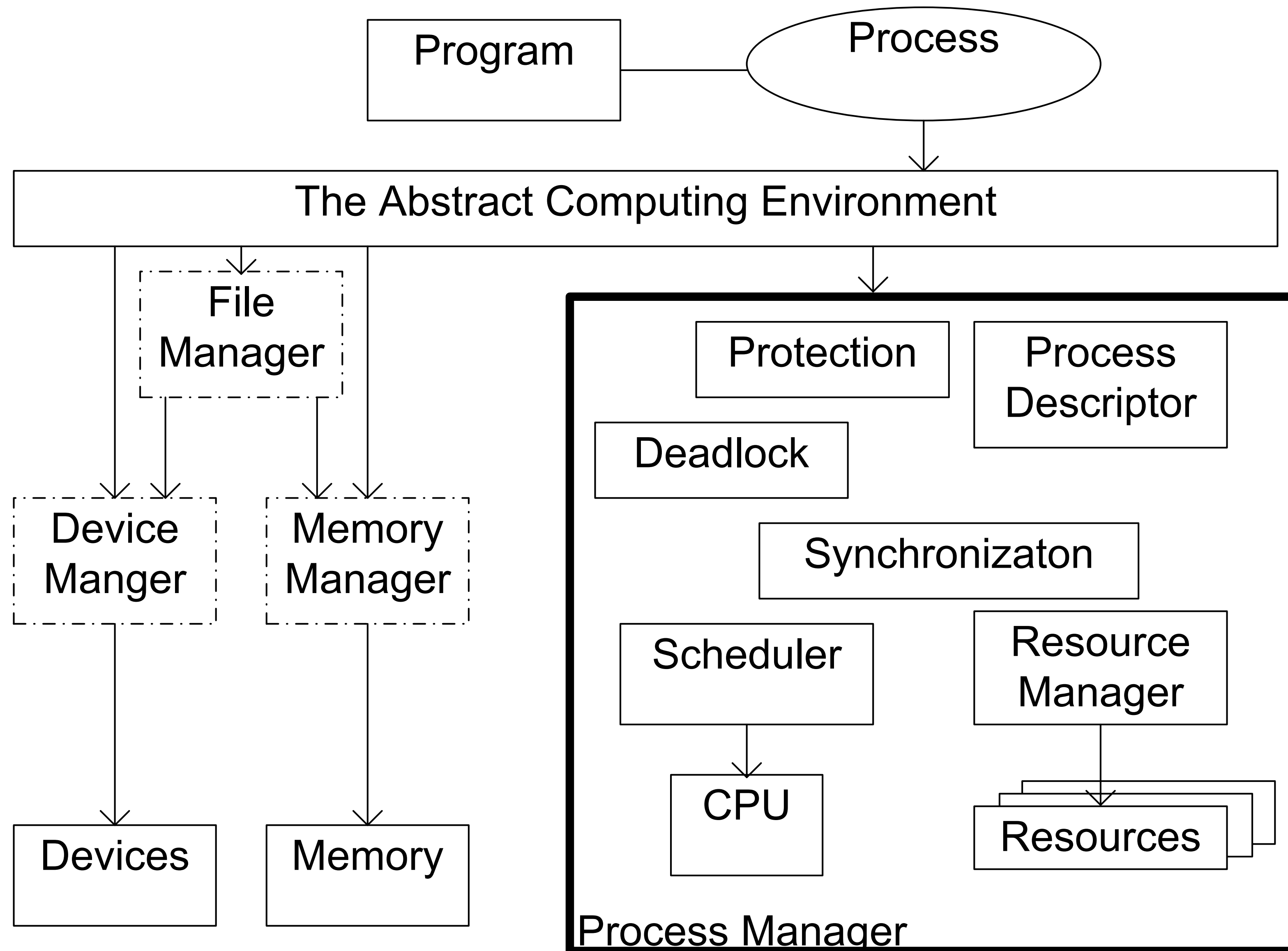
Process Management

What is a Process?

- A process is a program in execution

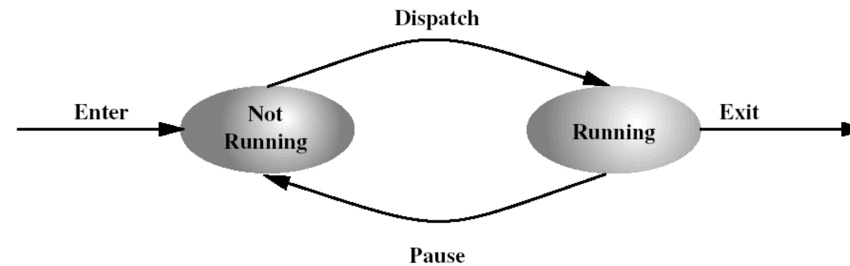
- It is composed of:
  - Program
  - Data
  - Process Control Block (PCB): contains the state of the process in execution
    - What it is?
    - How much of its processing has been completed?
    - Etc.

Process Manager

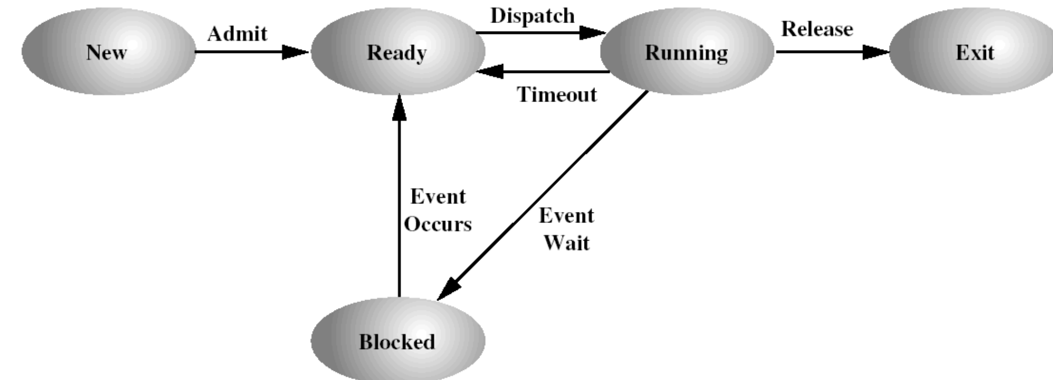


State Process Models

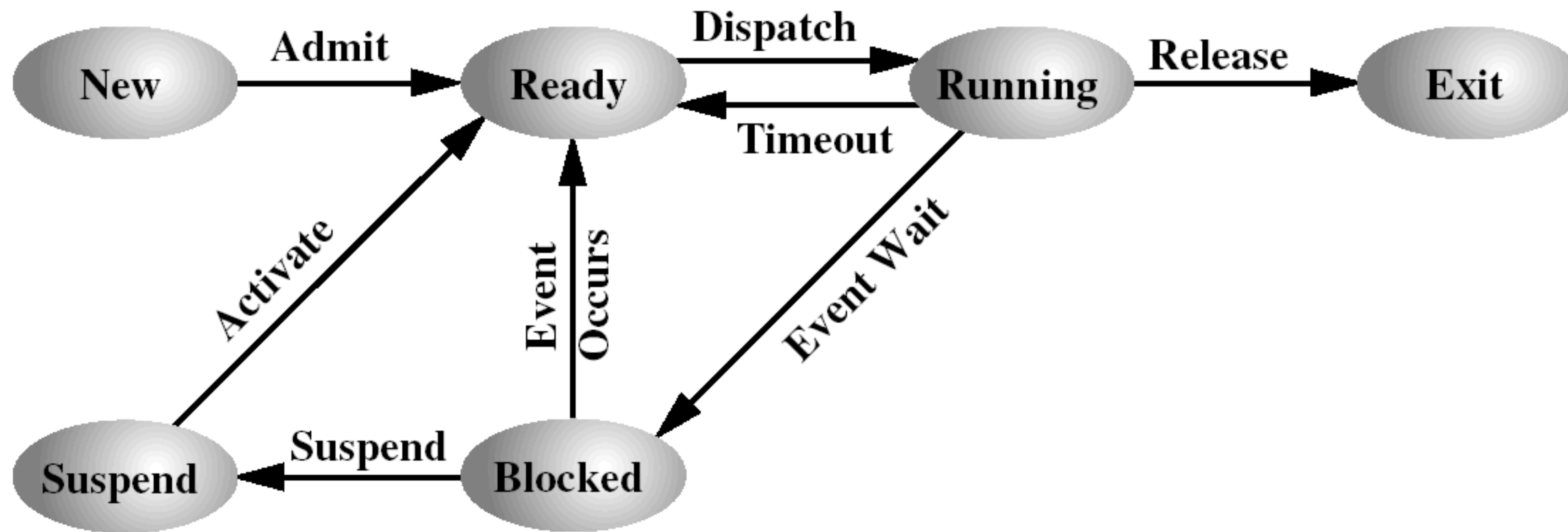
- Two State Process Model



- Five State Process Model



Suspended Processes



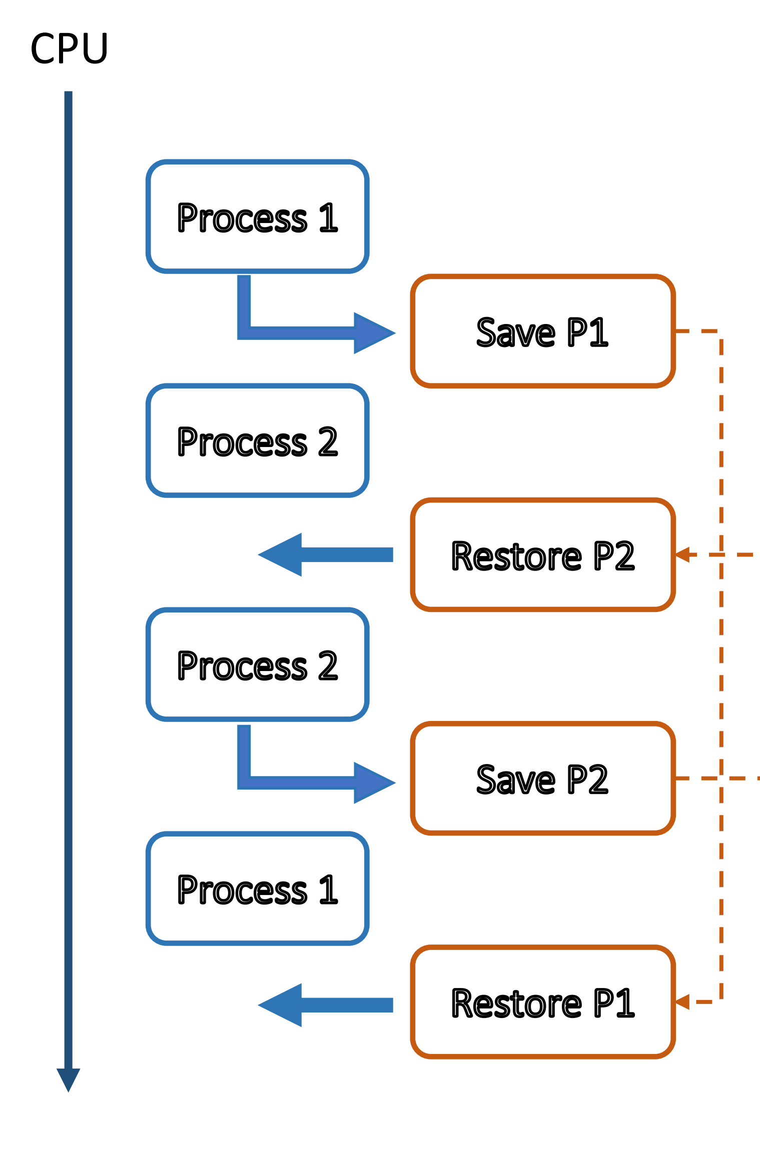
- Processor is faster than I/O so all processes could be waiting for I/O
- Swap these processes to disk to free up more memory
- Blocked state becomes suspend state when swapped to disk

CPU Management - Scheduling

Scheduling

- Scheduling allows one process to use the CPU while the execution of another process is on hold (i.e., in waiting state) due to unavailability of any resource like I/O etc
    - Aims to make the system efficient, fast and fair.

- Scheduling is part of the process manager



Scheduler Types

- *Cooperative* scheduler (**voluntary** CPU sharing)
  - Each process will ***periodically invoke*** the process scheduler, voluntarily sharing the CPU
  - Each process should call a function that will implement the process scheduling.
    - ***yield*** **(**$P_{current}$, P_{next}**)** (sometimes implemented as an instruction in hardware), where $P_{current}$ is an identifier of the current process and the P_{next} is an identifier of the next process)

- *Preemptive* scheduler (**involuntary** CPU sharing)
  - The interrupt system ***enforces periodic involuntary interruption*** of any process's execution; it can force a process to involuntarily execute a yield type function (or instruction)
  - This is done by incorporating an ***interval timer*** device that produces an interrupt whenever the time expires

Scheduling Algorithms

- FCFS (First Come First Served)
- SJF (Shortest Job First)
- SRTN (Shortest Remaining Time Next)
- Time slice (Round Robin)
- Priority based preemptive scheduling
- MLQ (Multiple Level Queue)
- MLQF (Multiple Level Queue with Feedback)

Process Synchronisation

Concurrent Programming

- Concurrent programs: *interleaving sets of sequential atomic instructions*.
  - i.e., interacting sequential processes run at same time, on same/different processor(s)
  - processes *interleaved*, i.e. at any time each processor runs one of instructions of the sequential processes

Lets Look at this in Practice: Race Conditions

- A **race condition** occurs when a program output is dependent on the sequence or timing of code execution
  - if multiple processes of execution enter a **critical section** at about the same time; both attempt to update the shared data structure
  - ➢ leads to surprising results (undesirable)
  - ❖ You must work to avoid this with concurrent code

- **Critical section** = parts of the program where a shared resource is accessed
  - It needs to be protected in ways that avoid the concurrent access

Synchronisation Solutions

Ways to protect critical sections

- Option 1: Atomicity
  - Atomic operations cannot be interrupted, in order to avoid illogical outcomes
- Option 2: Conditional synchronisation (ordering)
  - Making sure that one process runs before another

Using Lock

- Locks are declared like variables:
  `Lock myLock;`
- A program can use multiple locks – why?
  `Lock myDataLock, myIoLock;`
- To use a lock:
  - Surround critical section as follows:
    - Call **acquire**() at start of critical section
    - Call **release**() at end of critical section
- Remember our general pattern for mutex

```
while (true)
    // Non_Critical_Section

    myLock.acquire();

    // Critical_Section

    myLock.release();

    // Non_Critical_Section
end while
```

Surround critical section of code

Do Locks give us sufficient safety?

1. **Check Safety properties:** these must *always* be true
   - *Mutual exclusion:* Two processes must not interleave certain sequences of instructions
   - *Absence of deadlock:* Deadlock is when a non-terminating system cannot respond to any signal

2. **Check Liveness properties:** These must *eventually* be true
   - *Absence of starvation:* Information sent is delivered
   - *Fairness:* That any contention must be resolved

- If you can demonstrate **any** cases in which these properties do not hold
- ➢ then, **the system is not correct**

Q: What do you think?

Requirements for Deadlock

All 4 conditions must hold for deadlock to occur:

1. **Mutex**: at least one held resource must be non-shareable
2. **No pre-emption:** resources cannot be pre-empted (no way to break priority or take a resource away once allocated
   - Locks have this property
3. **Hold and wait**: there exists a process holding a resource and waiting for another resource
4. **Circular wait**: there exists a set of processes P_1, P_2,…,P_N such that P_1 is waiting for P_2, P_2 is waiting for P_3,… and P_N is waiting for P_1

Make code more efficient, hence, we want them

Need to avoid circular wait

If only 3 conditions hold then:

- you can get starvation
- but not deadlock

Semaphores

- Semaphore = higher level synchronisation primitive
  - Invented by Dijkstra in 1965 as part of THE OS project
- Semaphores are a kind of generalized lock
  - Main synchronisation primitive used in original UNIX
- Implement with a **counter** that is manipulated atomically via 2 operations **signal** and **wait**

`wait(semaphore)`: A.K.A., *down()* or *P()*

decrement counter

if counter is zero then block until semaphore is signalled

`signal(semaphore)`: A.K.A., *up()* or *V()*

increment counter

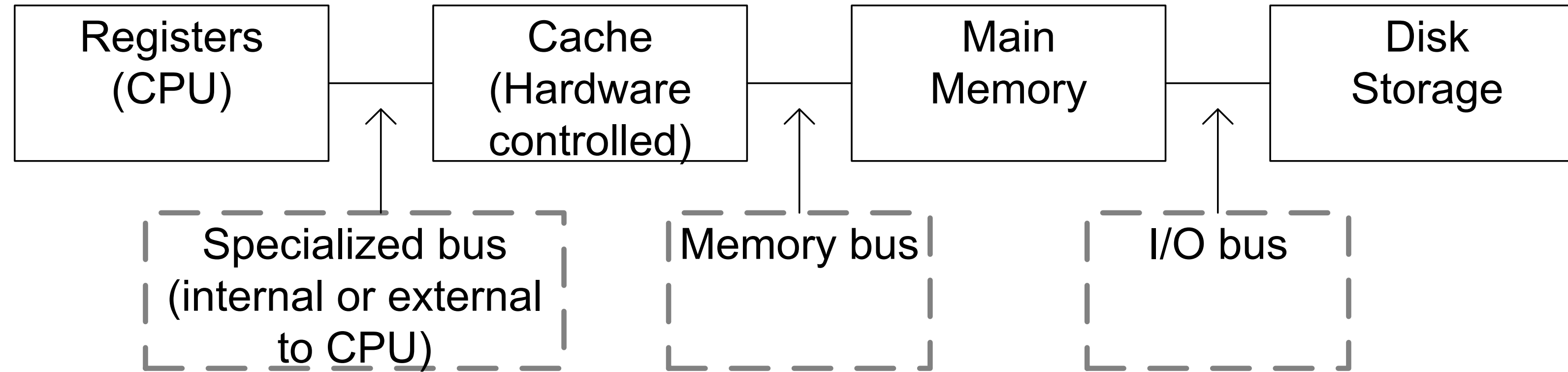
wake up one waiter, if any

`sem_init(semaphore, counter)`:

set initial counter value

Memory Management

Memory Hierarchy Review

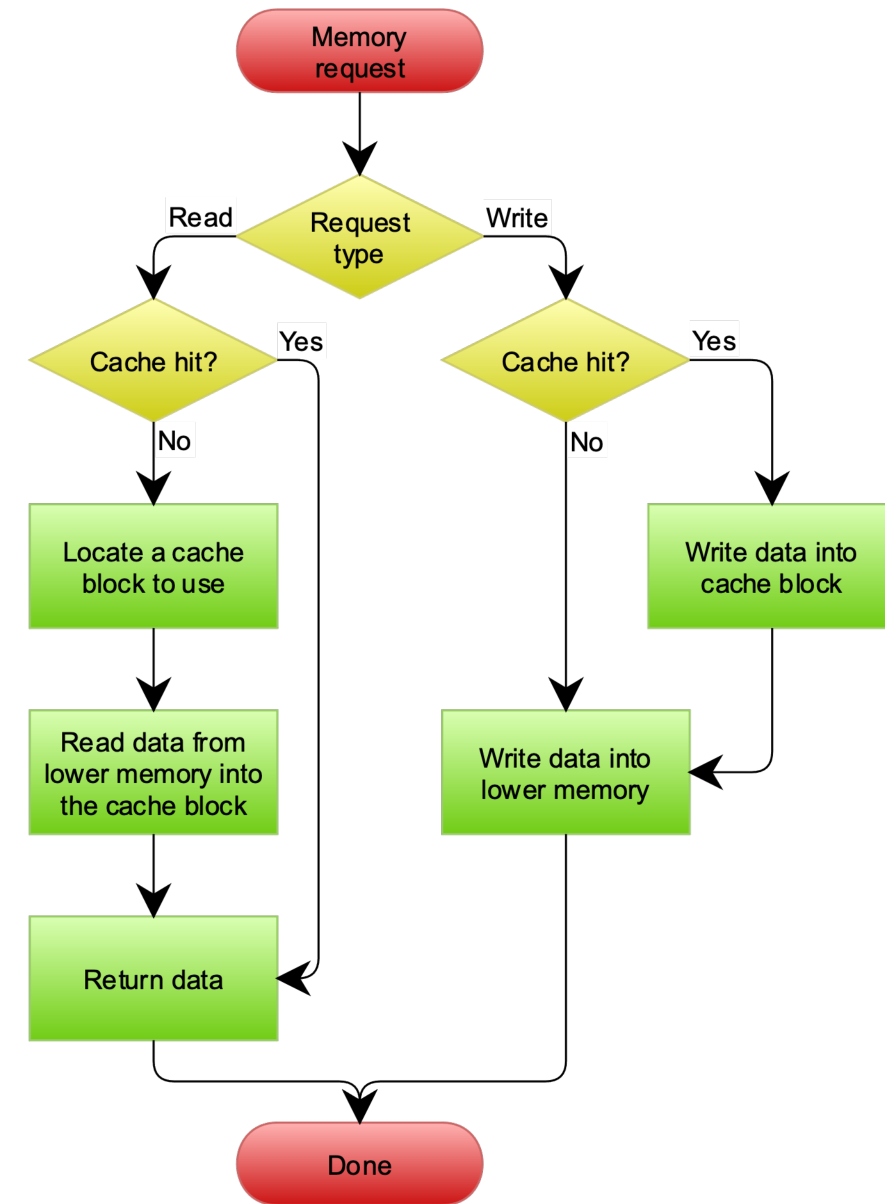


- It is a tradeoff between size, speed and cost
- **Register**
  - Fastest memory element; but small storage; very expensive
- **Cache**
  - Fast and small compared to main memory; acts as a buffer between the CPU and main memory: it contains the most recent used memory locations (*address* and *contents* are recorded here)
- **Main memory** is the RAM of the system
- **Disk storage** - HDD

Cache review

- Typical computer applications access data with a high degree of locality of reference:
  - ***Temporal locality***: data is requested that has been recently requested already
  - ***Spatial locality***: data is requested that is stored physically close to data that has already been requested

- When a system writes data to cache, it must at some point write that data to the main memory as well following the ***Write policies:***
  - ***Write-through***: write is done synchronously both to the cache and to main memory
  - ***Write-back***: initially, writing is done only to the cache. The write to main memory is postponed until the modified content is about to be replaced by another cache block.



https://commons.wikimedia.org/w/index.php?curid=27652294

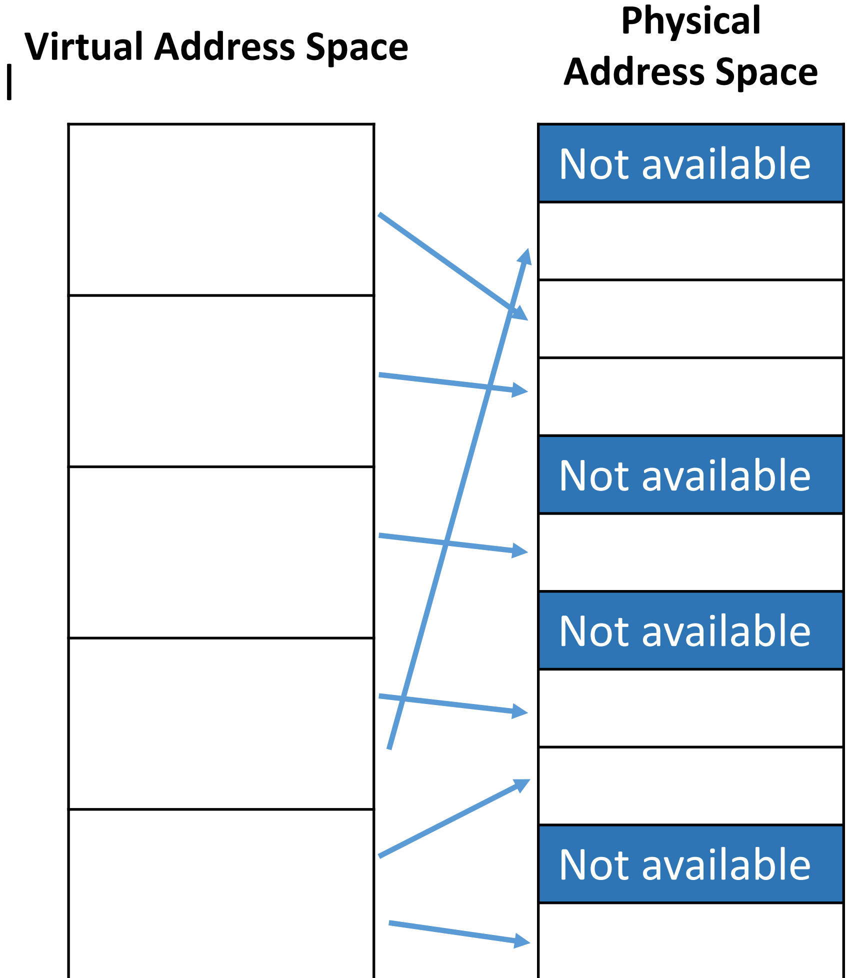
Memory Organisation

- **Logical organisation**
  - Most programs are organised in modules
    - Some modules are un-modifiable (read only and/or execute only)
    - Others contain data that can be modified
  - The operating system must take care of the possibility of sharing modules across processes
- **Physical organisation**
  - Memory is organised as at least a two-level hierarchy.
  - The OS should hide this fact and should perform the data movement between the main memory and secondary memory without the programmer's concern

Address Binding

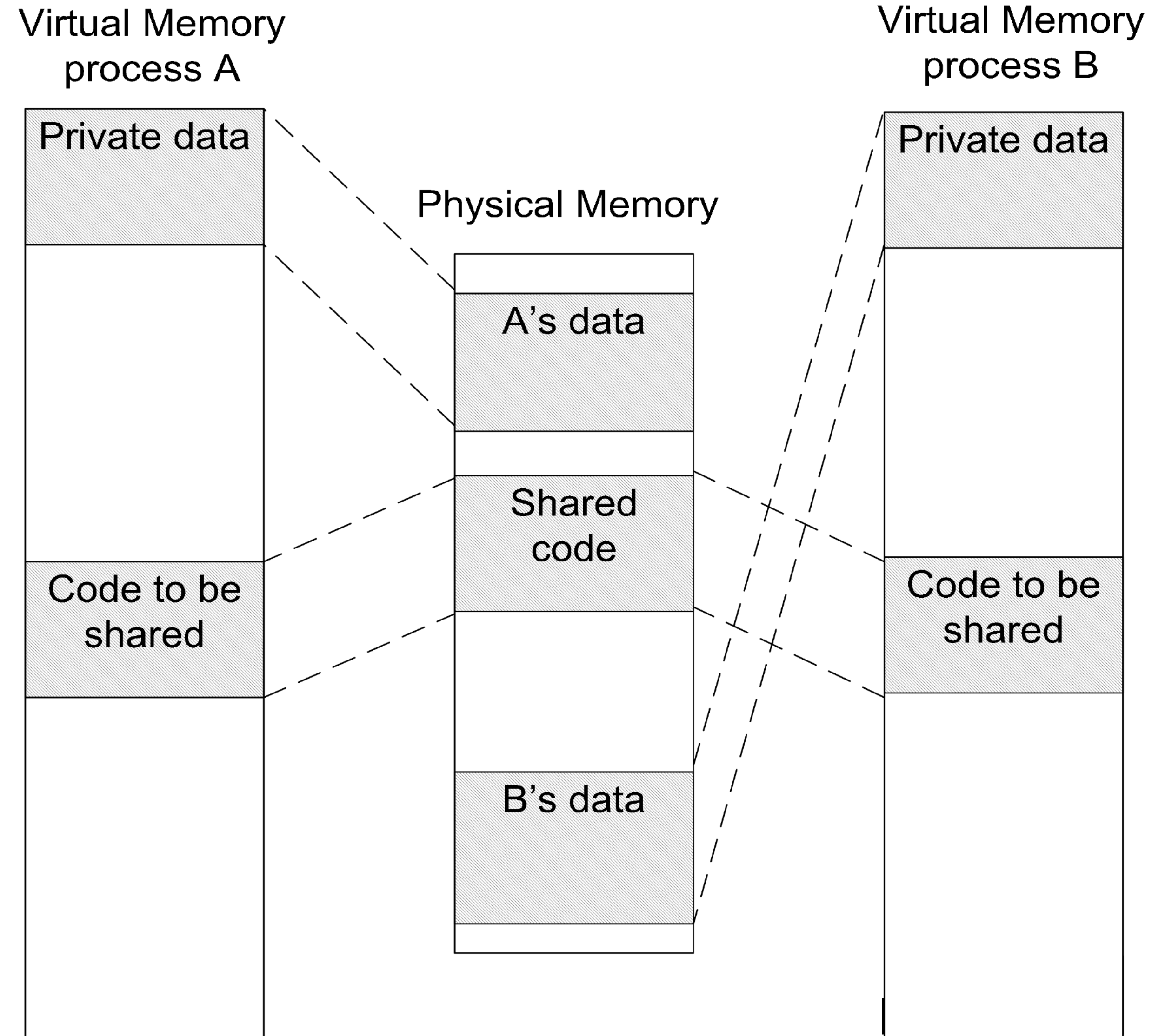
- An address used in an instruction can point ***anywhere*** in the virtual address space of the process
  - It still must be ***bound*** to a physical memory address
- Programs are made of modules.
- Compilers or assemblers ***do not know where the module will be loaded*** in the physical memory
  - ➢ Virtual addresses must be translated to physical addresses

➢ Address translation can be **dynamic** or **static**.



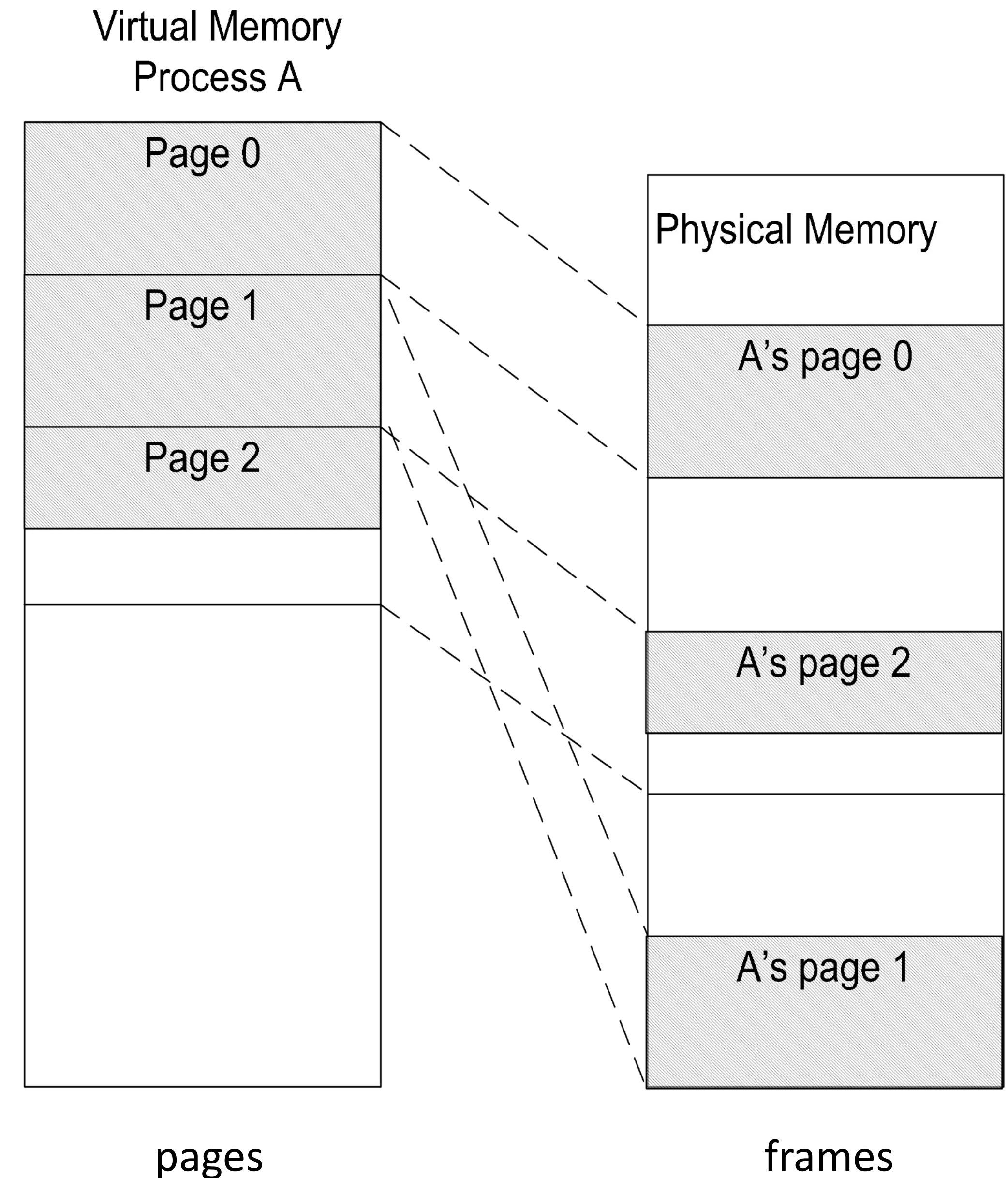
Segmented Virtual Memory

Two processes sharing a code segment but having private data segments



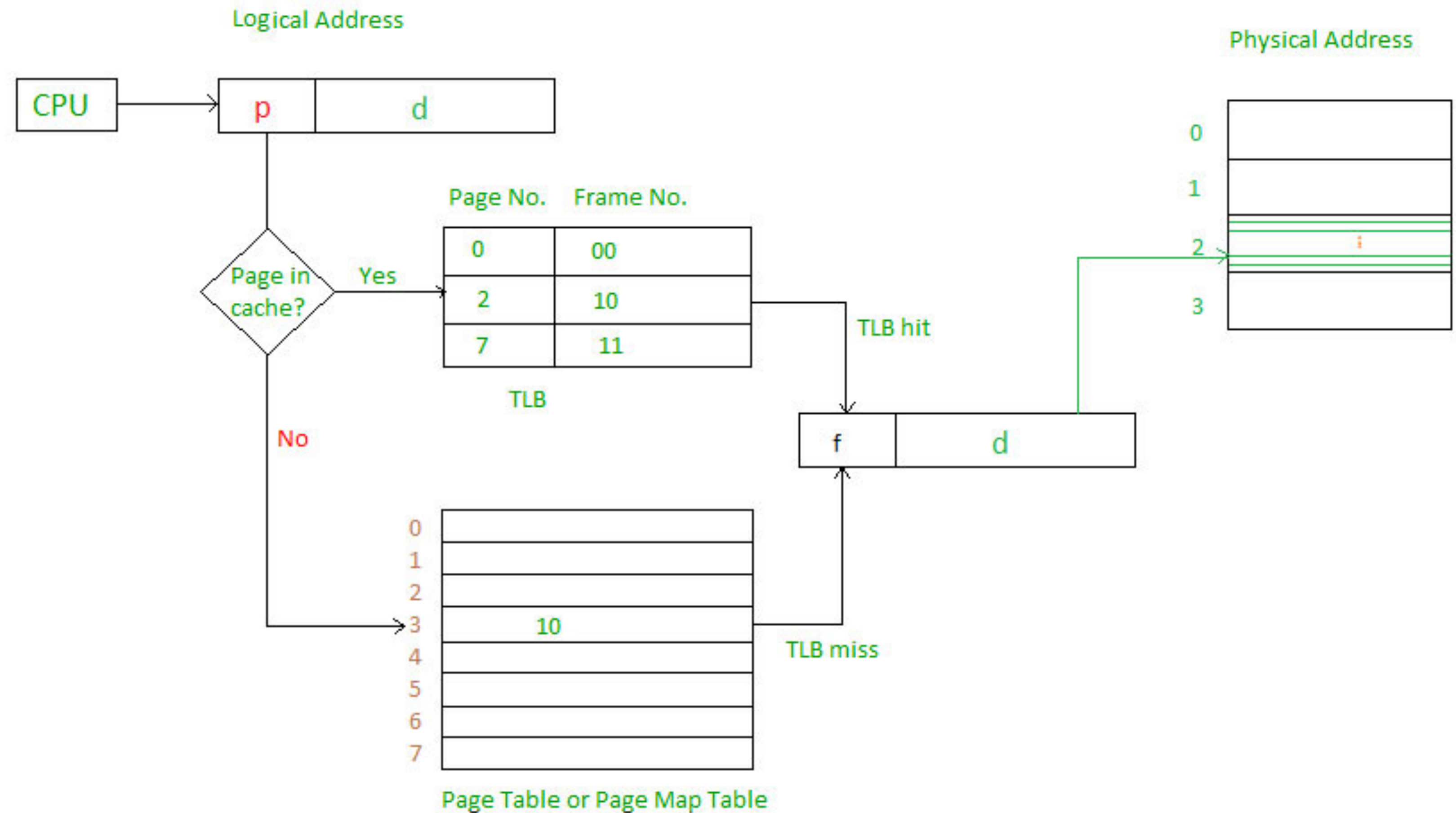
Paged Virtual Memory

- The need to keep each loaded segment contiguous in the physical memory poses a significant disadvantage:
  - It leads to fragmentation
  - It complicates the physical storage allocation problem
- Solution: ***paging***, where blocks of a fixed size are used for memory allocation (so that if there is any free space, it is of the right size)
- Memory is divided into page ***frames***, and the user program is divided into ***pages*** of the same size



Translation Lookaside Buffer (TLB)

- **A kind of cache memory:** it contains the page entries that have been most recently used
- TLB is searched for each address reference
- TLB is nearly always present in any processor that utilizes paged or segmented virtual memory
    - Including in most desktops, laptops, and servers.

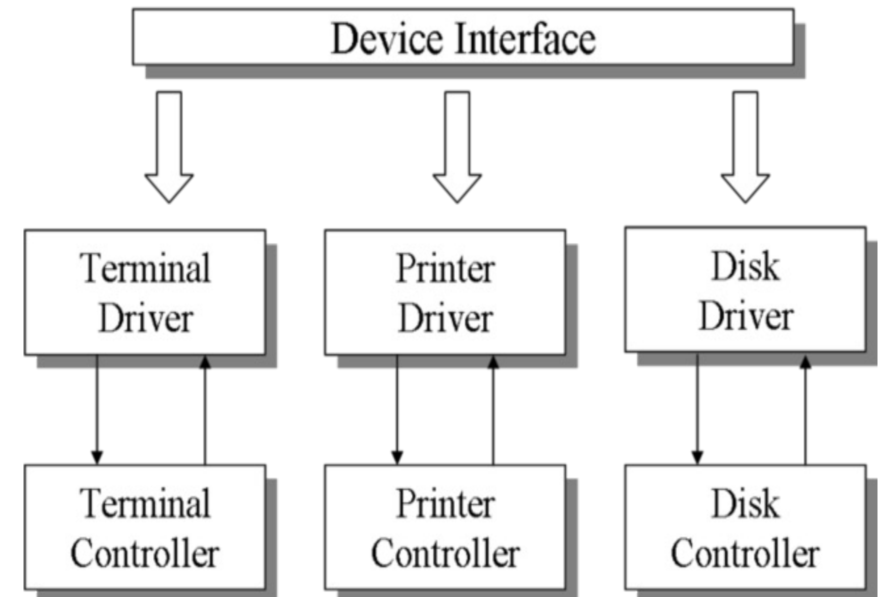


OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Device Management

Device Management

- An operating system manages the devices with the help of:
  - **Device controllers:** hardware components that contain some buffer registers to store the data temporarily.
    - E.g., disk controller, printer controller and a terminal controller
  - **Device drivers:** software programs that are used by an operating system to control the functioning of various devices in a uniform manner.

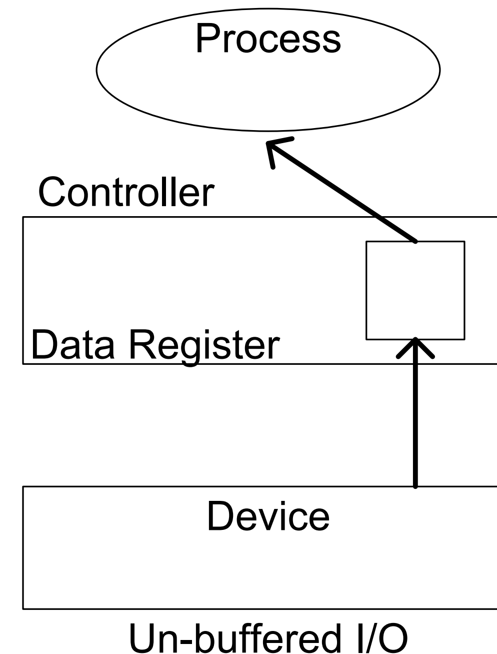


Device Communication Approaches

- A computer must have a way of detecting the arrival of **any type of input.**
- There are various ways to enable I/O devices to communicate with the processor:
  - Polling
  - Interrupts
  - Direct I/O
  - Memory Mapped I/O

Hardware Level Buffering

Consider a simple character device controller that reads a single byte from a router for each input operation.



Normal operation:

1. Read occurs
2. The driver passes a read command to the controller
3. The controller instructs the device to put the next character into one-byte data controller's register
4. The process calling for the byte **waits** for the operation to complete

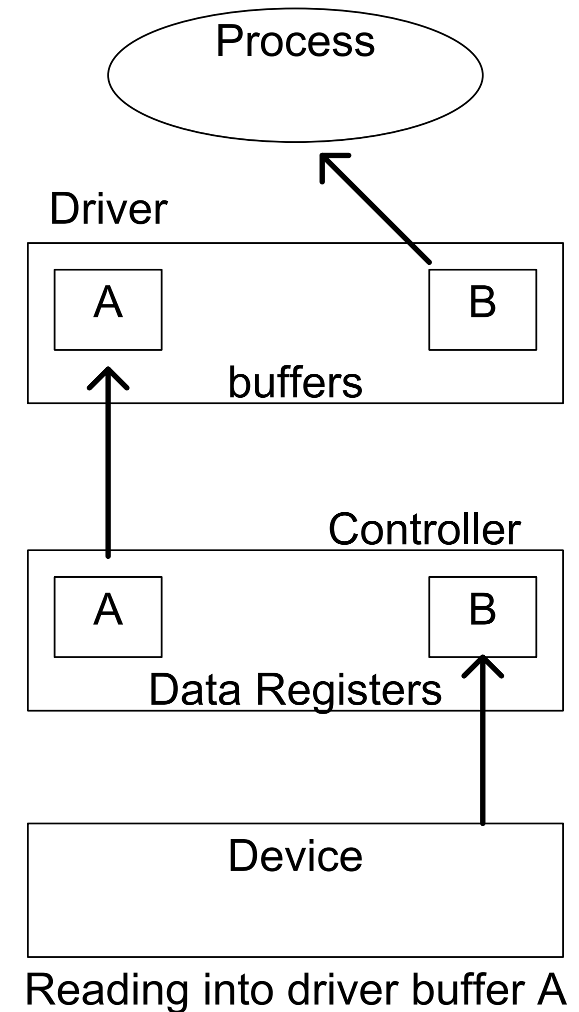
- then retrieves the character from the data register

Buffered operation:

- The next character to be read by the process has already been placed into the data register, even though the process has not yet called for the read operation
- Adding a hardware buffer to the controller **decreases** the amount of time the process has to **wait**

Driver Level Buffering

- This is generally called **double buffering**
  - One buffer is for the driver to store the data while waiting for the higher layers to read it
  - The other buffer is to store data from the lower-level module



Exam Details

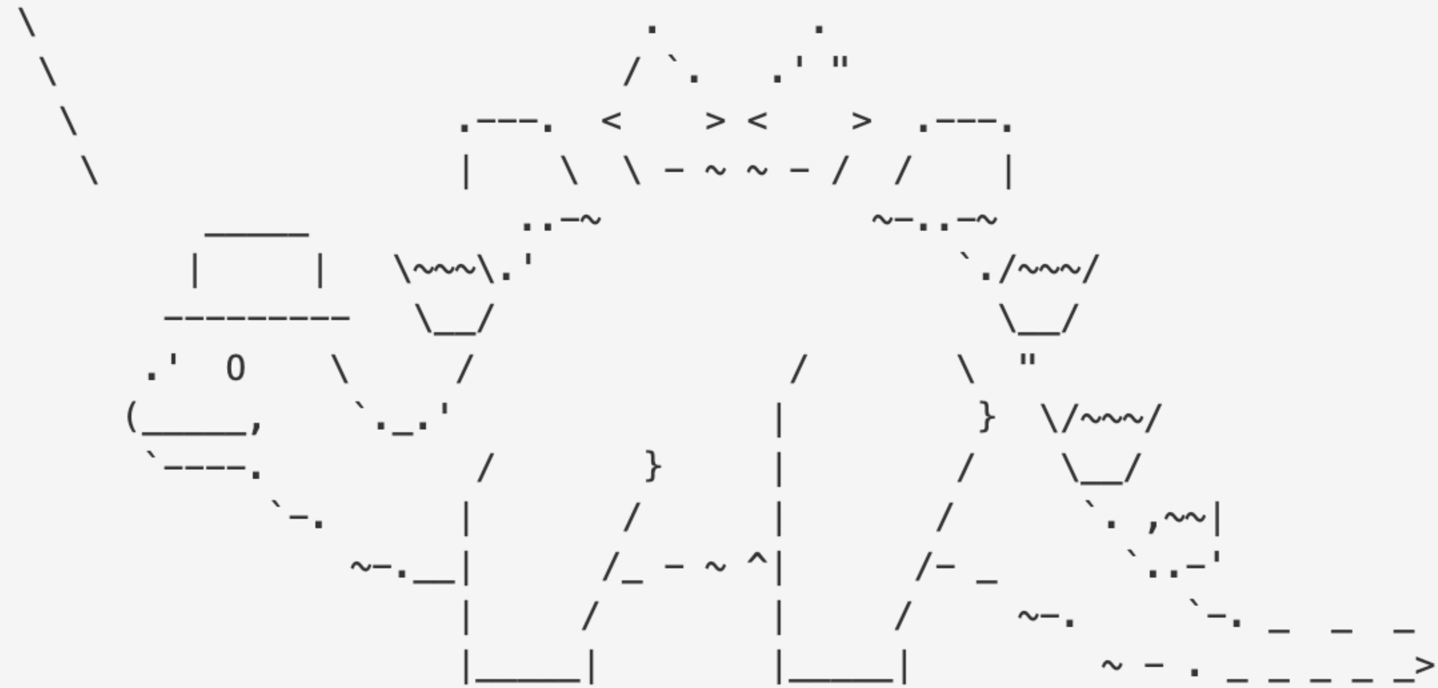
Exam Details

- 09/12/2022 at 13:00
- 2 hours
- Answer all questions (4 out of 4 questions).
- All questions carry equal marks (25pts each)

- Previous exam papers **are** relevant!
  - They will give you an idea of how the questions are asked.
  - However, some content is different! (this is a new content)

Questions

- Question 1:
  - Computer Systems
  - Programming Models
- Question 2:
  - Process Management
- Question 3:
  - CPU Management (Scheduling and Synchronisation)
- Question 4:
  - Memory Management
  - Device Management


< Best of luck for CT213 Exam >