
# Operating Systems (AI224)

## Course Reader

Dr. Karim Lounis

Winter 2025

# Preface

*Could you imaging how modern computers could be without operating systems? Just think about it for a moment. If you answered yes, then you are maybe a smart person, and if you answered no, then you may also be a smart person. Technically, every single computer on earth, that is currently powered on, is running an operating system, or something that is close to it.*

*An operating system is that software component that stands as an interface between the user and the computer so that the computer can be used easily and efficiently. In other words, some people were hanging around for sometime, worked very hard and collaborated to write lots of lines of code so that you can easily and efficiently use and exploit the capabilities and the functionalities of a complex electronic machine — the computer — to do a plenty of nice and interesting stuff.*

*It is important to note that people who wrote code for operating systems, most probably, were students once a while, attended courses at universities or colleges, and learned a lot about computers, software, and operating systems. Yet you may not be the person who is planning to work as a full-time software engineer or developer to write code for operating systems, you may certainly be that person who will be developing intelligent software solutions that rely on operating systems.*

*The course of Operating Systems is considered to be among the most fundamental courses in computer science. It covers various topics related to how operating systems offer an interactive software interface that transparently does many complex things in the background while giving the user the illusion of just sitting behind an easy-to-use and easy-to-exploit automated machine.*

*Considering the importance of the course, in general, and the importance of the topic, in particular, I decided to produce this course reader document as an additional and fruitful resource for students taking the course of Operating Systems at the National School of Artificial Intelligence (ENSIA). This document summaries the course content and provides various exercises to support them in better understanding the course and training themselves to thinking and solving problems related to operating systems concepts, mechanisms, and paradigms. Each section of this document aligns with the course syllabus, offering additional insights, practical examples, and hands-on activities to enhance your understanding and allow you to build strong foundational knowledge in this field.*

*This is the second version of the document — v1.1.*

# About the Author

Dr. Karim Lounis is an Assistant Professor at the National School of Artificial Intelligence (ENSIA), Sidi AbdAllah, Algeria. He obtained his Ph.D. degree in computer science from Queen's University, Canada, in 2020. He received a first master's degree in Networks and Distributed Systems from the University of Science and Technology Houari Boumediene (USTHB), Algeria, in 2013, and a second master's degree in Security of Information Systems from the University of East-Paris Creteil (UPEC), France, in 2014. Also, he obtained his Licence degree in Networks and Telecommunication Engineering from USTHB, Algeria, in 2011. After completing his second master's degree, he worked as a Research Assistant on information security with the system security group (CISPA—Center for IT-Security, Privacy and Accountability), Saarland University, Germany, from 2014 to 2015, and then with the security and trust of software systems group (SnT—Interdisciplinary Centre for Security, Reliability and Trust), University of Luxembourg, Luxembourg, from 2015 to 2017. He was a Postdoctoral fellow and a teaching adjunct at the School of Computing (Queen's University — Canada) from 2020 to 2021. His research interests include information security, networks security, and IoT security.

# Contents

# Chapter 1

# About the Course

## 1.1 Course Syllabus

The course presents the basic concepts of multitasking systems and the foundations of operating systems. It covers different topics such as process and thread management, system interrupt handling, process synchronization, memory management, file systems, and an introduction to operating system security. Also, the course provides practical mapping (through labs) of what is offered during the course lectures. The labs are mostly focused on process management using C (e.g., use of fork, wait, exit, and pipes), Java process synchronization (semaphores and monitors), and the use of GNU/Linux operating systems and writing GNU/Linux bash scripts.

## 1.2 Prerequisites

The course requires basic understanding of how computers operates. Some notions of software, algorithmic, and programming, and computer architecture. Being familliar with programming languages, such as C and Java, is a big plus. Also, I recommend taking the courses of Information Technology Essentials, Introduction to Linux, and Algorithms & Data Structures.

## 1.3 Motivation

There are various reasons for why you should like and enjoy this course:

- If you are using a computer (e.g., WS, PC, mac, tablet, smartphone, etc), then you are using an operating system. Could you use the computer, as it is supposed to be used, without having an operating system?
- You have been using an operating system, but you may not know what it is, and you may certainly not know its functions. What's happening under the hood? what happens when you run a program?
- You have learned how to write and run interesting programs, but you do not know how they interact with an operating system. What happens when you double-click on a program? or when it crashes (and why)?
- With advencement in operating systems, you are capable of running multiple programs and applications simultaniously without issues. How is this possible? You may have one processor and limited memory.
- You may have 8 to 32GB of memory (RAM) to run programs. How is it possible to smoothly run a 60GB program (e.g., GTA5) without any issues (and even simultaniously run various large programs)?
- You are used to simultaniously download files, make a skype-call, browse the net, and send emails without encountering any kind of interference. You only have one network card, how is this happening?
- You have certainly got the chance to write a program that runs a dirty infinite loop, while other programs are running. How comes that this dirty program did not influcene and block the others from running?
- You usually run multiple programs at the same time without any issues. Have ever gone trhough a scenario where the webpage you requested appeared in your skype call, where the face of the person you were talking to appeared on the web-browser (did not get mixed up)?
- You create and use files and folders everyday, but have you ever asked yourself how does the operating system stores different files and folder in permanent storage devices, e.g., hard disk drives?

## 1.4 Learning Objectives

In addition to becoming able to answer the questions raised in the previous section (Motivation), by the end of this course, students will:

- Throughout the lectures:
  - Learn the foundations of operating systems.
  - Know what an operating system is, what are its provided services, and what are its main components.
  - Know how operating systems deal with multi-processing and multi-threading in multiprocessors systems.
  - Know how operating systems enforce synchronization and guarantee data integrity.
  - Know how operating systems manage processes, central memory, and computer resources.
  - Know how operating systems manage and store files an folders in hard disk and solide state drives.
  - Know how operating systems provide security services.
- Throughout the labs:
  - Use GNU/Linux operating system (learn main components, architecture, file system, command lines, etc).
  - Use GNU/Linux bash scripts to write programs.
  - Use Java to solve synchronization problems.
  - Use C programming in Unix-like operating systems for process management and inter-process communication (e.g., fork, exit, wait, pipes, files, sockets, and RPCs).
- Throughout the tutorial sessions:
  - Solve problems related to operating systems.
  - Design algorithms for process synchronization.
  - Be able to answer fundamental questions related to operating systems.

## 1.5 Teaching Team

The course will be managed by the following faculty members:

- **Delivering lectures (head of the course):**
  - **Dr. Karim Lounis**: Assistant Professor (MCB).
- **Delivering labs:**
  - **Dr. Zouhyer Tamrabet**: Assistant Professor (MCB).
  - **Dr. Chabane Djeddi**: Assistant Professor (MCB).
  - **Dr. Mohamed Akram Khelili**: Assistant Professor (MCB).
- **Delivering tutorials:**
  - **Dr. Noureddine Lasla**: Assosiate Professor (MCA).
  - **Dr. Okba Tibermacine**: Assosiate Professor (MCA).
  - **Dr. Ouarda Lounis**: Assistant Professor (MCB).
  - **Dr. Zahia Mabrek**: Assistant Professor (MAB).

## 1.6 Course Delivery

During an academic week, the course will run as follows:

- **Lectures.** Lectures are weekly held for 15 weeks in an amphitheater. Week 8 is reserved for a midterm exam. Lecture sessions run for 90 minutes. Lecture slides will be posted on the course website after the lecture, and sometimes before the lecture.
- **Labs.** During these sessions, you're gonna be using your computers (or an ENSIA lab computer) to write computer programs that solve the lab's problems. In the meantime, we're gonna be using C and Java programming languages, and GNU/Linux bash scripts.
- **Tutorials.** These are sessions that run for 90 minutes. During these sessions, you're gonna be solving operating system's related problems. You will be given an exercise worksheet, time to solve the exercises, and then will discuss their solutions together with your tutors.

- **Office Hours.** If you had questions and needed further clarification about the lecture slides, or the labs, then office hours will be the perfect location for you to get your questions answered and clarified. They are held every week by each member of the teaching team.

According to *Article 17 — Resolution Nº499 — April 09th, 2023*, the course lectures, tutorials, and/or labs can be delivered either in-person or remotely. This should be decided by the teaching team members.

## 1.7 Course Textbook

In this course, we do not impose students to acquire a specific textbook. However, we do recommend having access to some interesting books as additional resources for the course. Below is a list of books that quite related to the course:

- Operating System Concepts, Ninth Edition (or newer), by A. Silberschatz, P. B. Galvin, and G. Gagne, published by John Wiley.
- Modern Operating Systems, Third Edition (or newer), by A. S. Tanenbaum, published by Pearson, Prentice Hall.

Also, feel free to use any other consistent and credible resources available over the Internet.

## 1.8 Course Schedule

The schedule of lecture topics is approximate, and may be adjusted slightly during the semester. The Textbook Sections column is a rough guide (cf., Textbooks 1 and 2 in the previous section); the assignments provide detailed information about the required readings in the textbook and course reader.

## 1.9 Method of Evaluation

The course grade will be calculated based on the following inputs:

- **CE (Continuous Evaluation).** During tutorial and lab sessions, the concerned instructor will assign marks to students based on their participation and work in the session (cf., *Article 18 — Resolution Nº499 — April 09th, 2023*). This evaluation is worth 6% of the total course mark.

A student who has accumulated three (03) unjustified absences, or five (05) justified absences in the course (in particular, during lab sessions), will be considered as **expelled** from the class and will automatically be assigned a **zero** mark in the course for the runnig semester (cf., *§2 — Article 18 — Resolution Nº499 — April 09th, 2023*).

- **Quizzes.** There will be online weekly quizzes (worth 7%).
- **Lab Test**. There will be a lab test on Week 12. It is worth 7%.
- **Midsemester exam.** There will be one 1h-exam (worth 20%).
- **Final Exam.** There will be a 3-hour final exam (worth 60%).

After each exam (midsemester or final), the course main instructor must communicate to the student the solution of the exam along with the marking scheme (cf., *Article 34 — Resolution Nº499 — April 09th, 2023*). Also, the instructor will coordinate with the school administration to organize a review session where students can have access to their exam papers to review the marking cf., *Article 35 — Resolution Nº499 — April 09th, 2023*).

## 1.10 Students with Disabilities

If you have a disability that may impact your ability to carry out assigned course work, it is important that you contact me and the administration. We will review your concerns and determine, with you, appropriate and necessary accommodations. All information and documentation of disability is confidential.

## 1.11 Recommendation Letters and Student Support

I strongly advice students to get in touch with their instructor as soon as possible when facing difficulties. Due to school regulations, late actions taken by students (e.g., justifying absences) may lead to bad consequences (e.g., not being able to rewrite an exam). Use office hours or reach out by email to set up an appointment and discuss your issues. You should not let your problems accumulate till the end of the semester or academic year.

If you need a recommendation letter from me to study abroad, you must have taken at least three courses with me (and me being your main instructor (Lecturer) in the course) and have obtained at least a 14.50 grade in my course. This will allow me to fairly and concisely write a correct recommendation letter for you.

## 1.12 Academic Integrity

In this course, the principle of academic integrity relies on six core values: honesty, trust, fairness, respect, responsibility, and courage (see AcademicIntegrity.org). These values are crucial and key to the development, enrichment, and sustainability of an academic community in which all members succeed and prosper.

By taking this course, you are responsible for familiarizing yourself with the regulations concerning academic integrity and for ensuring that your assignments conform to the principles of academic integrity.

Departures from academic integrity include plagiarism, use of unauthorized materials, facilitation, forgery, and falsification, and are antithetical to the development of an academic community at the National School of Artificial Intelligence. Given the seriousness of these matters, actions that contravene the regulation on academic integrity may carry sanctions that are decided by the disciplinary committee (following an AI panel —— Academic integrity panel). These sanctions can range from a warning to the loss of grades on an assignment to the failure of a course (assigned an eliminatory grade), to a requirement to withdraw from the school.

Although collaboration between students in discussing and understanding ideas related to this course is encouraged, violation of the following regula-

tions will not be tolerated:

- R1. Always properly acknowledging all sources of information used or referred to in your own academic work (e.g., project proposal, project report, research paper, etc). You may need to obtain permission for using particular artifacts.
- R2. Never seeking to obtain an unfair advantage for yourself or another in any form of academic work or examination (e.g., during an exam, when the proctor asks you to drop your pencils or pens, you must comply for the sake of fairness).
- R3. Collaborating with others when appropriate but always producing your own work independently when required.
- R4. Never obtaining unauthorized external assistance in the creation of academic work (e.g., expert assistance during a project).
- R5. Always present accurate data and information in your academic work (i.e., declaring false information and data is not tolerated).

## 1.13 Intellectual Property

Students should be aware that this course contains the intellectual property of their instructor. Intellectual property includes items such as:

- Lecture content, spoken and written (and any audio/video recording thereof);
- Lecture handouts, presentations, and other materials prepared for the course (e.g., slides);
- Questions or solution sets from various types of assessments (e.g., assignments, quizzes, midterm exams, final exams); and
- Work protected by copyright (e.g., any work authored by the instructor).

Course materials and the intellectual property contained therein, are used to enhance a student's educational experience. However, sharing this intellectual property without the intellectual property owner's permission is a violation of intellectual property rights.

For this reason, it is necessary to ask the instructor for permission before uploading and sharing their intellectual property online (e.g., to an online repository). Permission from the instructor is necessary before sharing the intellectual property of others from completed courses with students taking the same/similar courses in subsequent terms/years.

In many cases, instructors might be happy to allow distribution of certain materials. However, doing so without expressed permission is considered a violation of intellectual property rights.

# Chapter 2

# Operating Systems Overview

An operating system is the fundamental software that manages a computer's hardware and software resources. It serves as an intermediary between the user and the complex intricacies of the system, providing a user-friendly interface and ensuring efficient utilization of resources. In this section, we delve into operating systems. You will learn what an operating system is, its main functions, and types.

## 2.1 Operating System

Consider the following definition:

**Operating System**

An operating system, or OS, is a set of programs (called **modules**) that cooperate for the good management and use of **computer resources**. These modules can be classified into two classes, primary and secondary. The primary modules constitute the **kernel** of an OS.

The definition illustrates that an operating system is actually a set of programs (modules). These programs cooperate and work together for the common aim of efficiently managing the resources of a computer, providing the user with an interface to easily **exploit** (make use) those resources.

The operating system's modules can be primary or secondary:

- **Primary modules (Kernel):**
  - **Process management.** This module is responsible for creating, executing, and terminating processes. It includes a **process scheduler**, that decides which process will run next and when it should be stoped, based on a **scheduling algorithm**, e.g., FCFS, SJF, STRF, RR, and PR.
  - **Memory managment module.** This module decides where to load a process in the memory (RAM) and how to keep track of each portion of the memory. It allocates and frees space in the central memory — RAM.
  - **File system module.** This module keeps track of files, their identity, type, how they are stored in permanent storage devices (e.g., HDD and SSD), and where. Also, how the hard drive is logically structured (file systems: NTFS, FAT32, ext4, etc).
  - **Interrupt handling module.** This module handles interrupts: hardware (e.g., network traffic, keystrokes, mouse, etc), exception (e.g., division by 0, overflow, etc), and system calls (e.g., read and write on I/O devices)
- **Secondary modules:**
  - **Drivers.** These are system programs that interface between the operating system and certain hardware devices, e.g., advanced keyboard, network cards, printer, graphical card, etc.
  - **Networking module.** It implements a list of functions and services, called **communication protocols** (e.g., TCP/IP protocol suite), to allows inter/inner-computer communication.
  - **Security managment module.** Provides security services, such as encryption, authentication, data integrity, and availability.
  - **The Shell.** Is the program that implements the interface of interaction, either as command-line interface (CLI) and/or a graphical user interface (GUI), between the user and the kernel. In the GUI-Shell, users can run programs by clicking and executing them, whereas in CLI-Shell, programs are run by executing commands.

## 2.2 Operating System Services

Operating systems act as intermediaries, managing computer hardware and software resources to provide a user-friendly environment. They offer essential services to users as well as to itself:

**Services provided by the operating system to the users:**

**User Interface.** Could be a Batch Interface[1] [1945-1968], a CLI (Command Line Interface)[1969-] or GUI (Graphical User Interface) [1970-]. This is the interface through which the user interacts with the operating system. This interaction occurs either through the use of a keyboard to type and run commands, or using the mouse on a graphical user interface to select and run programs. Earlier systems, those which used punchcards, adopted the batch interface, where users submit batch jobs (punchcards programs) to be executed without any real-time interaction with the system during the programs execution. This latter system is no longer used. Modern operating systems used in personal computers or servers come with both CLI and GUI. Yet, certain systems, e.g., networking devices, come only with a CLI interface.

**Program Execution.** Programs need to be loaded into the main memory and get executed till completion. The operating system ensures that. It manages programs execution by fairely allocating the required resources among the programs while keeping track of their different states and progress.

**I/O Operations.** Running programs may require I/O operations (e.g., read from a file, read from a microphone, read from a webcam, ..., write into a file, write into a printer, write into a network card, etc), the operating system provides special functions to program to use I/O devices.

**File System Manipulation.** Programs may need to manipulate files. The operating system provides special functions for such manipulations. It allows programs to create, name, copy, store, read from, write to, delete files.

**Communication.** Programs may need to communicate with each other, locally (e.g., pipes and shared memory) or remotely. The OS provides special functions to guarantee a safe, secure, and reliable communication.

**Error Detection.** The operating system needs to detect and correct errors

[1]Batch OS — consists of having an operator rearranging and submiting group of similar jobs (a.k.a., batch) for a sequential execution — punchcards in & punchcards out.

when they occur during programs execution (e.g., device failure). For each type of error, the operating system takes the appropriate action.

**Services provided by the operating system to the users and system itself:**

**Resource Allocation.** The operating system ensures that resources are shared in an optimal manner among multiple users and jobs. For example, the central memory is a limited critical resource that is used by various programs to store the code and variables. The operating system provides mechanisms to request memory space when needed, and provide program with the necessary memory requirements for their smooth execution. Similarly, the processor is managed among several programs to allow give all of them the opportunity to run their codes in a fairly reliable manner.

**Accounting.** The operating system keeps track of which user uses how much and what kind of resources for accounting and statistics.

**Protection and Security.** The operating system provides certain security services to its users to preserve their security and privacy. For example, through its security management module, the operating system provides authentication using passwords, fingerprints, or face recognition, to control access to the computer resources. It also provides encryption mechanisms to secure access to confidential data on storage devices. Furthermore, the module allows the user to check and verify that files were not tampered-with by an unauthorized entity using mechanisms based on cryptographic hash functions. Last but not the least, some operating systems provide fault-tolerance and recovery to guarantee system availability, e.g., MINIX.

## 2.3 Types of Operating Systems

The diverse computational needs across various domains have led to the development of a multitude of operating systems. From the resource-constrained environments of embedded systems to the demanding workloads of high-performance computing (HPCs), different operating systems have evolved to cater to specific requirements, offering varying levels of functionality, performance, reliability, and security. In ths following, we discuss some of them.

**Multiprogramming Systems.** These are systems that can load multiple user programs onto the central memory at a time (in addition to the

OS). Modern operating systems are multi-programming systems. In a multi-programming system, user program can either run sequentially (one after the other — in the first come first served mode), in parallel (if multiple processing unit are available — multi-processor and multi-core systems), or alternatively (in the round-robin mode, a.k.a., fake parallelism).

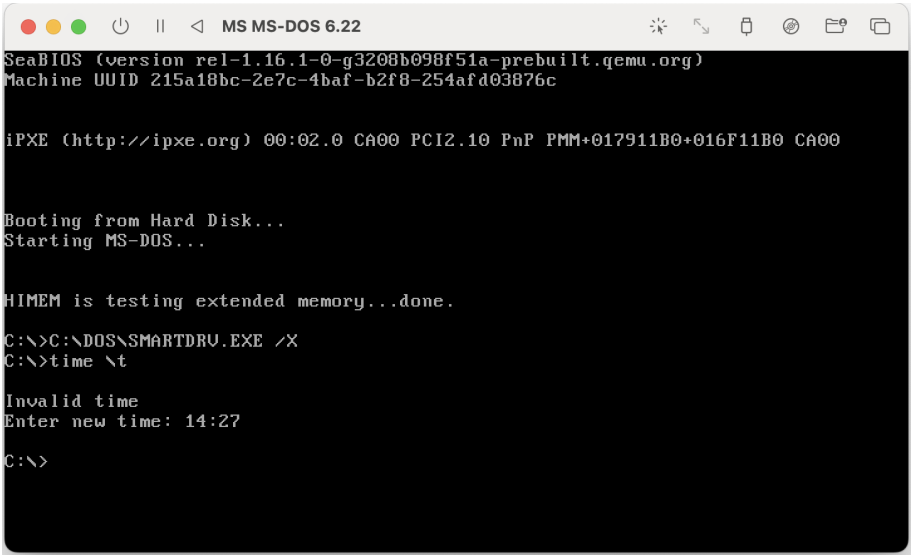



Figure 2.1: Example of a single-tasking operating systems: MS-DOS 6.22.

**Single-tasking Systems.** These are systems that can only load one single user program (in addition to the operating system code) onto the central memory at a time, e.g., Microsoft's MS-DOS (viz., Figure 2.3). Here, the operating system run a kind of infinite loop (i.e., while the user has not asked to shutdown the system) and waiting for the user to request the execution of a program. When the program is started, the computer jumps from the execution of the operating system code, into the execution of the user program's code, and then going back to the operating system code.
**Time-sharing Systems.** A.k.a., multitasking systems. These systems run multiple user programs either in parallel (or **real-parallelism**), or in **fake-parallism**, i.e., giving the user the **illusion** of executing everything simultaniously, but in reality, the **time** is **shared** among various programs from different users. In fact, these systems have two main features: (1) They are multiprogramming systems, i.e., load multiple user programs onto the cen-

tral memory (in addition to the operating system's code), and (2) they are muti-user systems, i.e., allow multiple users to share a single computer simultaniously to run their programs in a **time-sharing mode**. The time-sharing mode consits of dividing the CPU time into small intervals, called time **slices** or **quanta**, and allocating these slices to different users or processes. This creates the illusion that each user has exclusive access to the computer, even though they are all sharing the same resources. Modern operating systems like GNU/Linux distributions, Apple's macOS X, and Microsoft Windows (since Windows 95 & NT), are time-sharing operating systems.

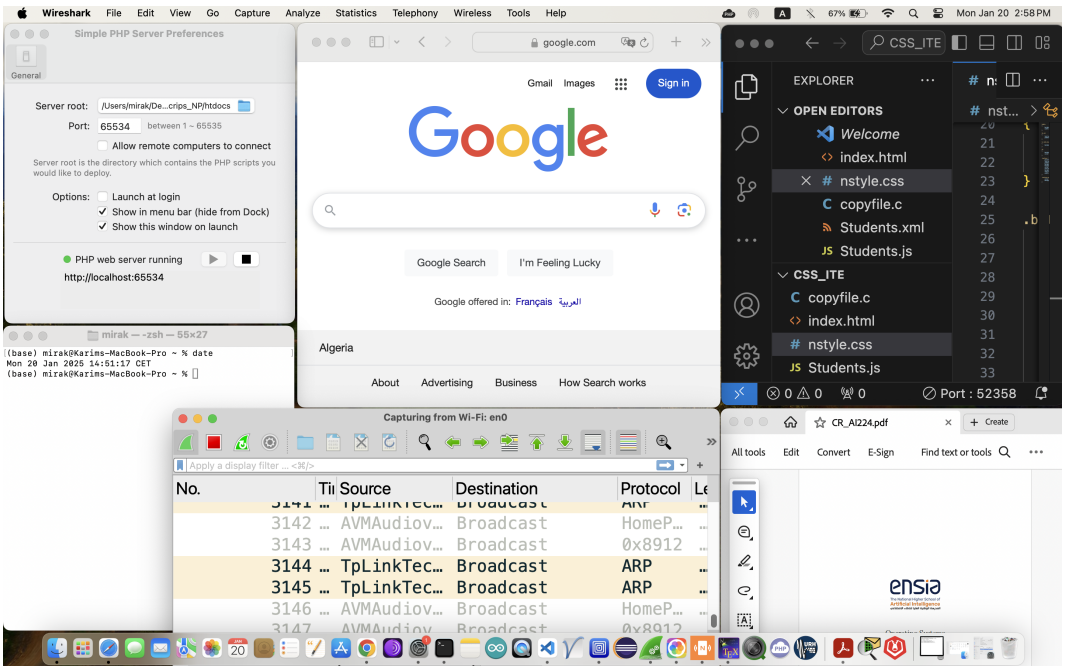



Figure 2.2: Example of a multi-tasking operating systems: Apple macOS 14.6.1 (Sonoma) running various application software (e.g., simple php server, Google Chrome, VS Code, CMD shell, Wireshark, and PDF reader).

**Embedded OSs.** An embedded device is a specialized computer system (perceive it as a black-box) designed to perform a dedicated function within a larger mechanical or electrical system (e.g., an ATM, a washing machine, a self-driving vehicle, a robot, a drone, a router, a military jet-fighter, etc). An embedded operating system is simply the system software for an em-

bedded system. The software either comes in the form of a **firmware**, i.e., a small low-level program with no interface (no shell module — no CMD and no GUI), or as a restricted form of ordinary operating systems that we see everyday (i.e., sometimes with a CMD shell, and sometimes with both CMD and GUI). Such systems include TinyOS, Linux Embedded, LiteOS, Microsoft Windows compact edition, and Windows Embedded. Most embedded operating systems used nowadays are Linux-based operating systems.

**Real-Time OSs.** A real-time operating system is **generally** (not always) an embedded operating system designed for critical systems, e.g., industrial facilities (e.g., nuclear power plants, chemical manufacturing plants, water treatments facilities, etc), military systems (e.g., radars, missiles, fighter-jets, etc), and mission-critical systems (e.g., aircrafts, spacecrafts, trains, etc). There are many real-time operating systems out there: FreeRTOS (Amazon), VxWorks (Wide River), QNX Neutino (Quantum Software Systems), ThreadX (Microsoft), and Zephyr (Linux Foundation).

**Distributed OSs.** These operating systems designed to manage a collection of independent and distributed computers (also known as nodes) and make them appear to the user as a single, powerfull, unified computer system. In such system, the computers that compose the entire system are distributed world-wide and connected to each other using a network (in a LAN, MAN, or WAN). Each computer runs a part of the distributed operating system (generally, a basic kernel, a communication module, and a distributed service) and cooperate with the other computers to unify the entire structure. The computers are organized following one of these architectures: (1) The client-server architecture, where some parts of the operating system reside on a central server, while other parts run on client nodes. (2) Peer-to-Peer architecture, where each node runs similar parts of the operating systems and can act as a server and client at the same time. (3) Hybrid, where the two previous architectures are combined.

**Network OSs.** These operating systems are designed for networks administration and management (e.g., Windows NT, Windows 2000-2022 server, or Red Hat Enterprise Linux). In a company for example, with 10K employees or more, some dedicated computers (powerfull computers) are used as servers on which a network operating system is installed (e.g., Microsoft Windwos 2025 Server). The employees' computers are then set-up (during installation) to be domain user computers, instead of workgroup user computers (i.e., independent entiry in the network). In a domain user computer, the computer is not tight to a specific user account (it is seen an a computer that belongs to the company and any employee can use it, through their remotely defined user account). The network administrator creates the user accounts on the server, and then the users (employees) can log into their respective accounts from whatever domain user computer and use it. The server manages the user accounts, the security policy (who can do what, when, and where), and other resources (e.g., printers, files, database, website, etc).

**Mobile OSs.** These are operating systems designed to work on mobile devices — smartphones, PDAs, tablets, smartwatches, etc (e.g., Windows Mobile, Android, iOS, Symbian OS, BlackBerry OS, etc). These operating systems share some common feature with ordinary operating systems that we use on personal computer, but do have some key distinctions, such as, managing smaller screens, power-saving focus, limited RAM focus, simplifed file system, limited multi-tasking, etc.

## 2.4 Booting an Operating System

To launch the operating system, the **kernel** (of the desired operating system — in case multiple OS are installed) needs to be located (in the hard disk or solide state drive) and loaded onto the central memory, then executed. This is known as **booting the system**.

When the computer is powred on, the BIOS (Basic Input/Output System) — a firmware stored in the ROM — performs the POST (Power-On-Self-Test) process, where it checks the hardware components (e.g., CPU, memory, storage devices, pheripherals, etc) to ensure they are functioning correctly. An error code may be displayed (or a beeping sound is played — on old motherboards). Then, it checks a pre-configured boot order (usually, stored in the CMOS settings) to determine which device to boot from (e.g., Floppy disk drive, HDD, SSD, USB drive, SD card, CD-ROM, DVD-ROM, etc).

Once the bootable device is selected, the BIOS read the first sector (Sector 0) — MBR (Master Boot Record), or GPT (GUID Partition Table) if UEFI[2] is used, to loead and execute the code of the **bootstrap loader**. The bootstrap loader (or boot code) locates and loads the next stage bootloader (a.k.a., **Stage 2 bootloader**, or operating system loader). This will help in the case multiple operating systems were installed on different disk partitions (the storage device is usually divided into paritions — parts).

The bootloader locates the operating system loader (e.g., Windows boot manager or Linux GRUB) in the first partition. The operating system bootloader then (after reading the other paritions meta-information) displays a menu to request the user to select which operating systems to load (if only one operating system is installed, then no menu is displayed). The user (or the bootloader, if the user stays idle) selects the operating system to load, i.e., the kernel to load. When the kernel is loaded, it starts executing the operating system.

---

**Exercise.** Answer the following questions:

1. What is the kernel of an operating system?
2. Which module of the operating system is responsible for managing the execution of programs and which one is responsbile for managing Internet connections?
3. Which module of the operating system provides the user with a GUI or CLI to interact with the operating system to run programs and software?
4. Provide three examples of hardware resources and logical resources.
5. Which module of the operating system is responsible for encrypting folders and files and which one is responsible for organizing the storage of files in HDD?
6. Give one example of an interrupt.

---

The solution of the exercise will be discussed during the lecture.

[2]UEFI (Unified Extensible Firmware Interface) is a successor to BIOS (Basic Input/Output System), aiming to address its technical shortcomings.

## 2.5 Unix and GNU/Linux Operating System

The story of Unix and GNU/Linux started in the '60s when punched cards were still used. A cooperative project, called MULTICS (Multiplexed Information and Computing Service), started in 1964, led by MIT, General Electric, and AT&T Bell Labs, aimed at the development of a time-sharing operating system, Multics, to operate on GE-645 mainframes. The project failed in 1969.

Figure 2.3: Some employees working on punched-cards computers (on the left), a GE-645 mainframe (in the center), and Denis Ritchie and Ken Thomson (from AT&A Bell Labs) working on a PDP-11/20 (on the right).

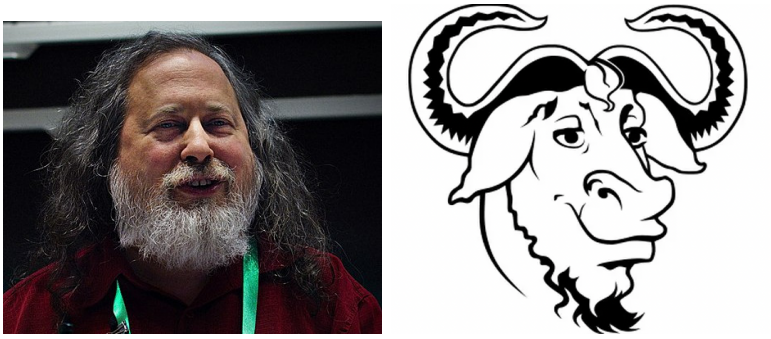
Ken Thomson, who worked on the project decided to continue working on the project to, at least, create something out of the project aches. He formed a team in **1970** and built Unics, which later became Unix. The first version of Unics (Unix v1.0) was fully written in assembly and was run on DEC PDP-11/20 minicomputers. In 1972, Denis Retchie, a programmer from the team, developed the C programming language. In the meantime, the team was progressing and upgrading Unix to v4.0. They decided to rewrite Unix in C, creating Unix v5.0 in **1972**. However, at that time, AT&T was forbidden from entering the computer market after some legal issues. So, they could not commercialize the product.

As a consequence, AT&T decided to license the source code of Unix to third commercial and academic parties (they were legally allowed to do so), e.g., UC Berkeley, Microsoft, IBM, DEC, HP and Sun Microsystems. In **1977**, the University of California Berkeley was licensed the source code and started developing the system further creating **BSD**. Also, in **1980**, Microsoft created Xenix based on Unix v7.0.

In **1984**, AT&T and Bell Labs got separated and AT&T started selling a commercial Unix version called **System V**. Then, from System V and BSD, many versions of Unix were created, E.g., HP-UX (1984) and IBM AIX (1986) were both based on System V. Whereas, the ULTRIX (1984) from DEC (for PDP-11 and VAX) and Sun Microsystems Solaris (1981) were based on BSD.

The UC Berkeley started replacing AT&T files with their own files to become separate from AT&T and System V. It managed to publish its source code and create FreeBSD in **1983**. E.g., Of course, there was a lawsuit with AT&T and got dropped.

In **1983**, Richard Stallman founded the GNU foundation (which stands for GNU is Not Unix) with the aim of developing an operating system as a free software replacement for Unix (1984).



Richard Stallman (Left) and the GNU project logo (right).

In the early '90s, the GNU system was almost complete but missing an important part, which was the kernel (GNU Hurd was legging behind). Few year earlier (1987), somewhere in The Netherlands, Andrew Tanenbaum wrote MINIX, a Unix-like operating system, particularly developed as an educational tool to teach operating systems and get rid of AT&T licence issues. At that time, in Finland, Linus Torvald, a student, who used MINIX and knew about Unix wanted to overcome the deficiencies of MINIX and wrote his own operating system (with a kernel called **Linux**) which he wanted to make public. In **1991**, he released Linux under GPL (GNU General Public Licence). The combination of Linux kernel and the incomplete GNU system made a completely free operating system, called GNU/Linux operating system.

The GNU General Public License (GNU GPL) is a series of widely used free software licenses that guarantee end users the four freedoms to run,

study, share, and modify the software. These GPL series are all copyleft licenses, which means that any derivative work must be distributed under the same or equivalent license terms.

The Linux kernel is not the Unix kernel. It does not use Unix source code but uses Unix philosophy and architecture. Many operating systems nowadays use the Linux kernel. E.g., the Android operating system uses the Linux kernel. Also, 60% of servers outside use Linux. It also runs on embedded systems, including routers, smart home devices, video game consoles, televisions (e.g., Samsung and LG Smart TVs), automobiles (e.g., Tesla, Audi, Mercedes-Benz, Hyundai, and Toyota), and spacecraft (e.g., Falcon 9 rocket). There are thousands of GNU/Linux distributions: Red Hat, Fedora, Mandriva, Arch, Ubuntu (considered as a spyware by Richard Stallman), Debian, Backtrack, Kali, Parrot, Linux Mint, Yellow dog, and Kubuntu, these are just distributions (different desktops, ...) but the same Linux Kernel.

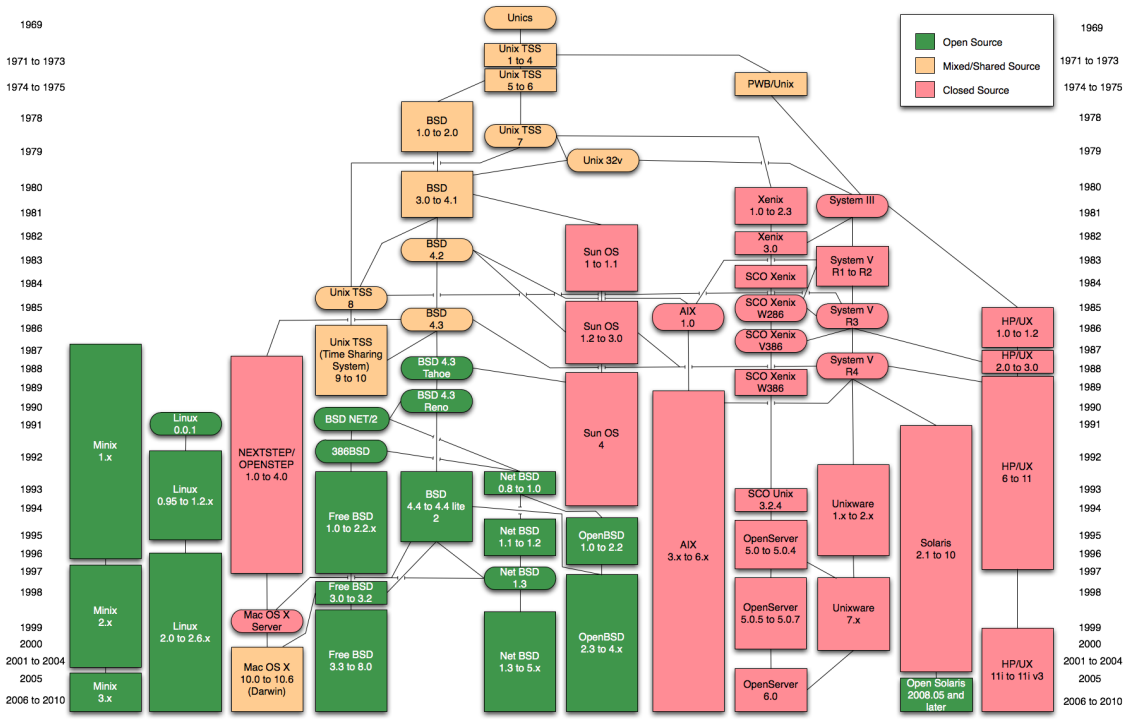



Figure 2.4: Unix-based Operating Systems

# Chapter 3

# Computer Architecture Review

This chapter will mainly serve as a review. Although most of the content was discussed in the information technology essentials course (AI101), we do cover additional topics such as, instruction execution cycle. The chapter will allow student to review the most fundamental concepts in computer architecture to better understand what is coming in the next chapters.

## 3.1 Computer

Before defining what a computer is, I would like you to take few seconds to look at the following images and think of what do they have in common.

After looking at those images, we could realize that, iIn fact, all these are computers ... computers are not just laptops and desktops. If we ask Google's `Gemeni` system to provide us with a definition for computers, it says "A computer is an electronic device capable of processing information and performing tasks based on instructions provided by software. Essentially, it's a machine that manipulates data according to a set of rules". It also says: "Computers range from small embedded systems found in everyday appliances to powerful supercomputers used for scientific research. Despite their diversity, they all share the fundamental ability to process information rapidly and accurately" (Aug 2024). The second part of the provided definition clearly shows that computers may take different forms and may be used in various applications. These different types of computers do share some common components and features, which make them literally "computers" with respect to the first part of the definition.

Laptops

Washing machines

Cars

Smartwatches


Automatic Teller Machine

Magnetic Resonance Imaging

Tablets (iPads)

Playstations

Smart cards

Now, let us consider the following generic and abstract definition:

**Computer**

A computer is an electronic machine that is designed to automatically compute (i.e., solve) problems at high speed.

Let us take a moment and analyze the above definition. The definition states that computers are eletronic machines — so they are not purely mechanical machines. This is due to the rapid development in the field of electronics. Efficient and miniaturized electronic components, particularly transistors and integrated circuits, have seen their integration in computers. There exists an observation made by the engineer Gordon Moor, which predicts that the number of transistors in an integrated circuit doubles about

every two years. This trend was a driving force in the semiconductor industry for decades, leading to exponential growth in computing power. However, in recent years, physical limitations and economic factors have slowed down this pace. While we still see improvements in processor speed and efficiency, the rate of increase is not as dramatic as it once was.

The definition also states that computers were designed to "automatically" compute. That refers to reducing human intervention. Computers are intended to be standalone systems, i.e., systems that operate independently.

The next important feature in the definition is "compute problems". The word "compute" here means "solve". The definition hence becomes "computers are electronic machines designed to automatically solve problems at a very high velocity". So, computers are actually machines designed to **solve** different types of problems at a very high speed and with less or no human intervention. Computers actually solve problems by performing a combinations of logical and arithemetic operations structured as a set of instructions, called **programs**. The following are some daily problems that we encounter, and that different computers can try to solve for us:

- Given a list of 1000 natural numbers, you can instruct a computer, e.g., a laptop or your smartphone, to calculate the sum of those thousand numbers in less than a microsecond.
- Given some geographic coordinates of latitude 36.68913533573665 and longitude 2.8677613464487894 (e.g., ENSIA location) and another coordinate (36.71184153180695, 3.202466267757517 — Algiers International Airport), you can instruct a self-driving car to take a person from ENSIA school to the international airport in around 40 minutes.
- Given 7Kg of dirty clothes, you can instruct a modern washing machine to clean-wash the clothes and turn them dry in a matter of 90 minutes.
- You can instruct a refrigirator to maintain a temperature of $6c^o$ within the fridge compartment and a temperature of -$15c^o$ within the feezer compartment.
- You can instruct an ATM to check your bank account sold and to withdraw a certain amount of money by inserting you bank card into the card reader and proving your card (account) ownership.

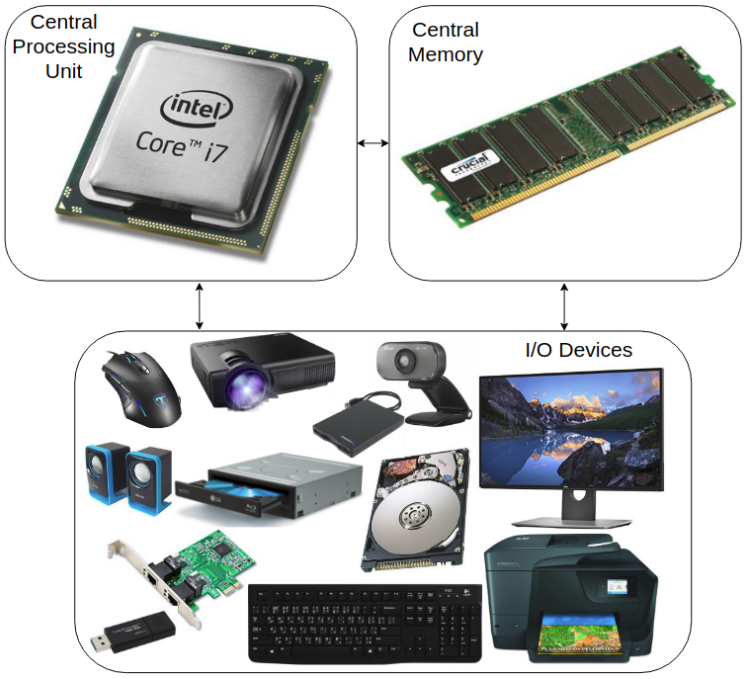



Figure 3.1: The main three components of a computer

- A radiologist can instruct an MRI system to scan a specific part of a patient body to diagnosis possible factures in the their brain.

Despite their diverse functions, these computers (and many others) share a common computational foundation. Each type of computer incorporates a processing unit, a.k.a., **processor**, for calculation and decision-making, **memory** for data storage, and **input and output interfaces** to interact with its surroundings. We explore these three components in the next section.

## 3.2 Computer Components

In the previous section, we talked about different types of computers and mentioned that despite their diverse functions, computers share a common computational foundation. Also, computers have considerably evolved in terms of structure (or architecture), size, and performance. However, from

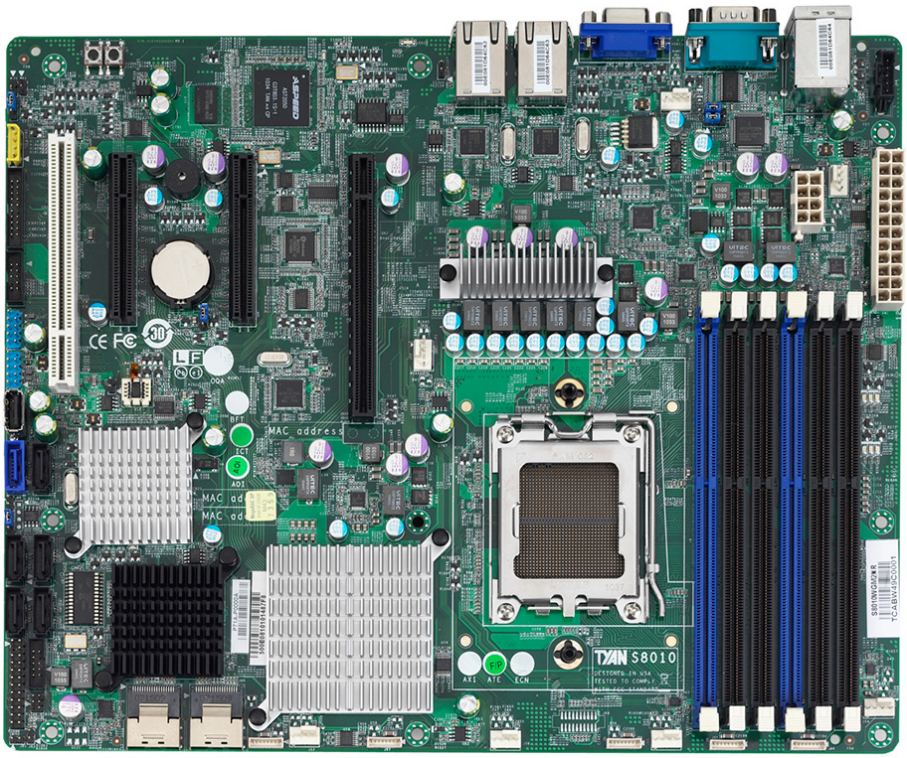



Figure 3.2: A mono-processor motherboard

a superfecial viewpoint, the various computers have three common "main" components (viz., Figure 3.1):

- **CPU.** The Central Processing Unit, a.k.a., Processor or Microprocessor, is responsible for performing arithmetic and logical operations. It is the brain of the computer.

- **Central Memory.** Is the component responsible for storing programs to be executed by the computer.

- **Peripherals.** A.k.a., I/O devices, are components used to input or output information to and from a computer.

These three components are interconnected via communication lines (**system bus**) in a cricuit board, known as **the motherboard** (viz., Figure 3.2).

## 3.3 Von Neumann Architectural Model

The Von Neumann architecture is a fundamental model for computer systems. It comprises four primary components: the Central Processing Unit (CPU), the Central Memory, the Input/Output (I/O) unit, and the System Bus. The CPU serves as the brain, executing instructions and performing calculations. The Central Memory stores both data and instructions, accessible by the CPU. The I/O unit facilitates communication between the computer and external devices. Finally, the System Bus acts as the communication channel connecting these components. It consits of a control bus, to issue read and write commands to the central memory, an address bus to indicate the address of memory locations, and the data bus to transfer instructions and data between the processor and the central memory. The bus is also used by the I/O device controllers to transfer data between the central memory and the device controllers. Figure 3.3 illustrates the Von Neumann architectural model.

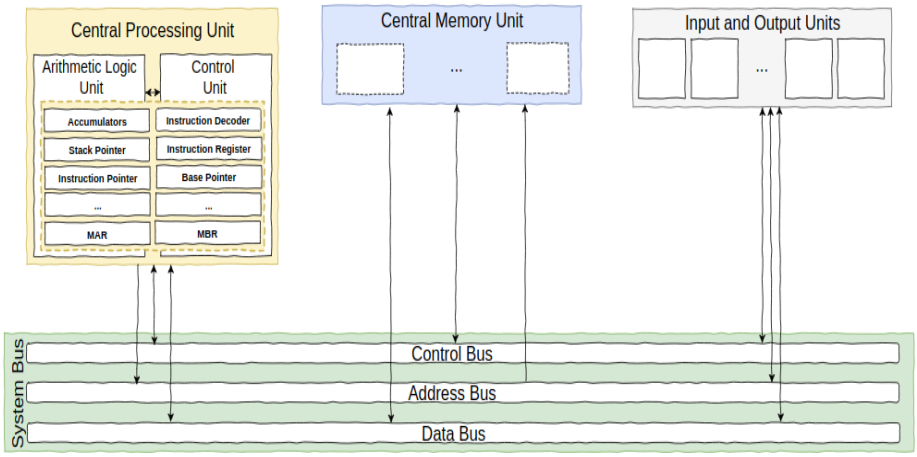



Figure 3.3: The Von Neumann Architectural Model

## 3.4 Central Processing Unit

In this section, we go over central processing units, their structures, types, and instruction sets.


Figure 3.4: Processors from different brands

**Processor**

A processor is a complex electronic circuit designed to execute machine instructions at a very high speed.

The advent of microtechnology, which enabled the miniaturization of electronic components, led to the development of smaller processors, called **microprocessors**. We often use the term **microprocessor** instead of processor. Figure 3.4 illustrates different processors from different time and brands.

### 3.4.1 CPU Characteristics

Processors are featured by five characteristics: **frequency** (in MHz or GHz), **internal memory** (cache and registers), **word size** (8-64 bit), **number of cores**, and **number of thread per core**.

- **Frequency.** The frequency of a processor represents the number of operations the processor is able to perform per second. The processor contains a special circuit, called **clock**. This electronic circuit generates a periodic box-type signal (i.e., ⊓⊔⊓⊔⊓⊔) at a given frequency. If at every one second, the signal containes 3 billion oscillations (i.e., clock cycles ⊓⊔), we say that the processor has a clock speed (or frequency) of 3.0 Giga Hertz, or 3GHz. Typically, in one clock cycle (⊓⊔), a processor performs one (01) single operation. However, modern processors can perform more than one operation per clock cycle.

- **Internal Memory.** The processor contains two types of internal memory, **cache** and **registers**. The cache memory is mainly used to store instructions (program's instructions) and data that has recently been used, or information that is under the assumption of pretty soon reuse. This cache memory is also used to store another type of information, called **addresses**[1], to speed up access to the central memory during programs execution. The **registers** [2] is another type of rapid memory that is used by the processor to temporary keep instructions and data that is being used during a given program execution. Most of these registers get updated from the execution of one instruction to another.

- **Word-Size.** The word-size of a processor represents the size of its internal registers. It also represents the amount of bits the processor can handle or process at a time. An 8-bit processor has a certain number of 8-bit registers. Also, when the processor performs computation, it only handle unit of 8 bits at a time. Modern computers (e.g., on your laptops) carry 64-bit processors.

- **Number of Cores.** Modern processors integrate more than one processing unit (more than one brain) within a single electronic chip (socket). They are called, **multicore** processors, e.g., Intel Pentium D (2005) dual-core, Core i3, i5, and i9, AMD RYZEN 3, Apple Silicon M1/2, etc. Each core has it own processing unit and internal memory.

- **Thread per Core.** Modern programming techniques allows a programmer to design their algorithms and codes in such a way so that the execution of the code can happen through different and independent parallel execution flows, called **threads**. For example, suppose we had to computer the expression `(a·b) + (c·d)`. The programmer can create two separate execution flows, one computing the term (a·b) and another computing (c·d). Each execution flow will run on a separate core, or on the same core at the same time. In fact, modern processors can execute multiple threads per single core at a time, which tries to maximize the processor usability. That is called **hyperthreading** [3].

[1] An address is an information that tells us where in the central memory a specific instruction or data is stored

[2] Registers are aligned memory cells made of flipflops and latches circuits. One latche can only hold one (01) single bit of information. A 16-bit register stores 16 bits.

[3] Hyperthreading is a technology (introduced by Intel in 2002) where processors can

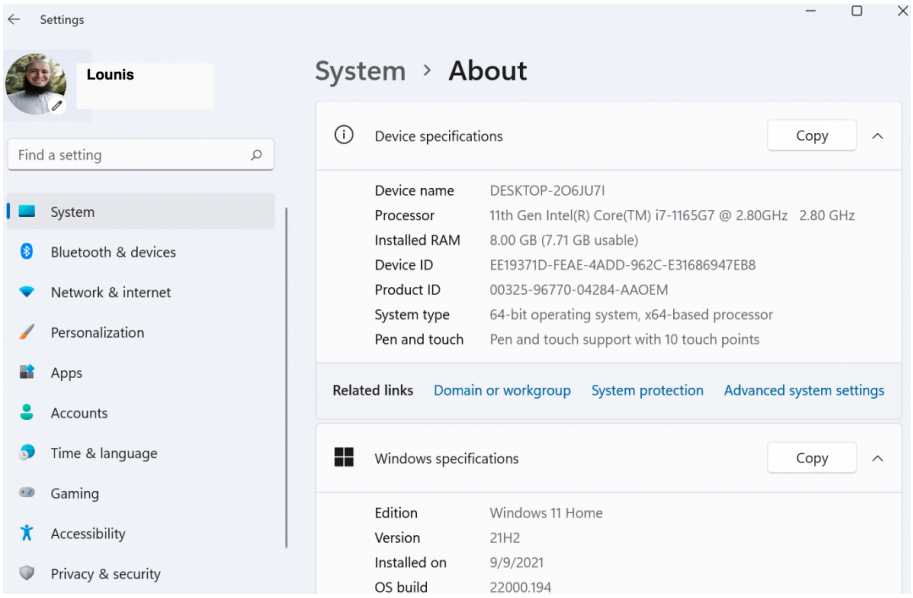



Figure 3.5: Features of a processor running Windows 11 Home

A processor is connected to the motherboard through the **CPU socket**. The CPU requires a fan to keep its temperature cool, otherwise, the CPU burns, or fries. CPUs are very expensive. Users should be very careful. It is not a good thing to have them burned.

There exists another feature that characterizes processors. The **IPC**, or **Instruction Per clock Cycle**. It measures the average number of instructions that the CPU can execute per clock cycle.

On Windows PCs you can see the computer's processor features by browsing the computer's properties (viz., Figure 3.5). On GNU/Linux machines, you can use the `lscpu` command to check the specifications (viz., Figures 3.6). The GNU/Linux output provides more useful detail. The first line (Architecture) says that the computer uses a 64-bit processor that follows the Intel's `x86` architecture. The second line tells us that the processor supports both operational mode, 32 bit and 64 bit. This means, if you try to run a program that was developed on a 32-bit computer, it will still run on this one.

execute multiple threads (execution flows) per core.

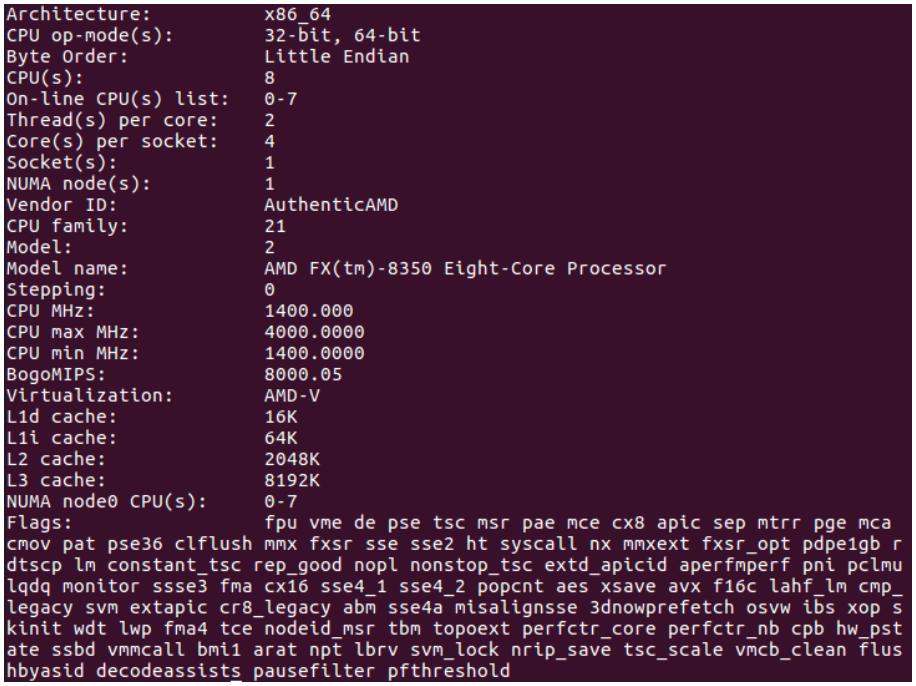



Figure 3.6: Features of an eight-core processor running GNU/Linux Ubuntu

The third line (Little Endian) refers to the order in which bytes of a multi-byte data value are stored in computer memory. In a little-endian system, the least significant byte (LSB) of a multi-byte value is stored at the smallest memory address, followed by the next least significant byte, and so on, with the most significant byte (MSB) at the highest address. The fourth line (CPU) indicates the number of **logical processors** in this physical processor. This is obtained by multiplying the number of physical cores (i.e., Line 7) by the number of threads per core (Line 6). This processor, AMD FX(tm)-8350 (Line 13), comes in one single chip, one CPU socket (Line 8). The maximum operational frequency of this processor is 4000MHz (Line 16), i.e., 4GHz. This value can sometimes be increased so that the processor runs faster. This is often called **overclocking**. Gamers generally do that to have better performances and response time, but it reduces the processor's lifetime as a consequence. Fome Line 20 to 23, the sizes of the different caches are shown. For example, L3 can hold up to 8192 kilobytes.

### 3.4.2 CPU Structure

Structurally, a CPU consists of:

- **Arithmetic Logical Unit (ALU).** This unit performs airhtmetic (`+`, `-`, `*`, ÷) and logical (`and`, `or`, `not`) operations.
- **Control Unit (CU).** This unit fetches (retrieves from central memory) instructions and decode them (i.e., understand their meaning).
- **Cache memory.** The internal processor's cache is organized in three levels L1, L2, and L3. The first two levels are generally separated to hold data and instructions. For example, in L1 cache, the latter will be organized as L1i and L1d, one for instructions, and the other for data.
- **Registers.** These are small and fast storage locations (of length word-size) used for instructions execution. Some of them are general-purpose, whereas other are special-purpose. In the following paragraphs, we explore some of them:

  **General-purpose Registers:**

| | |
|---|---|
| AX | Accumulator register: Used for arithmetic and logical operations |
| BX | Base register: Used for addressing memory locations |
| CX | Counting registers: used in loops |
| DX | Data register: holds operands and results of operations |

These registers are denoted eax, ebx, ecx, edx on 32-bit CISC processors, as rax, rbx, ... on 64-bit CISC processors, as r0, r1, r2, ... on 32-bit RISC processors, and x0, x1, ... on 64-bit RISC processors.

  **Indexing Registers:**

| | |
|---|---|
| BP | Base Pointer: Points stack frames and helps access local variables |
| SP | Stack Pointer: Used to point the top of the stack |
| IP | Instruction Pointer: Points the address of the next instruction (PC) |
| LR | Link register: Holds the return address (during function calls) |

These registers are denoted ebp, esp, eip, elr on 32-bit CISC processors, as rbp, rsp, ... on 64-bit CISC processors, as r13, r14, r15, ... on 32-bit RISC processors, and bp, sp, pc, lr, ... on 64-bit RISC processors.

**Status and Memory Registers:**

| | |
|---|---|
| FL | Flags[4]: Holds various flags (carry flag, zero flag, sing flag, interrupt flag, parity flag, overflow flag, ... trap flag) |
| SR | Segment registers point base address of segments |
| MAR | Memory address register: Holds address to read from/write to RAM. |
| MBR | Memory buffer register: Holds data to be written, or just read from RAM. |
| CRI | Holds the current instruction to be decoded. |
| ISRSP | Used to manage nested interrupts (points interrupt stack in kernel space) |

- **Communicaiton buses.** The buses allow the CPU to communicate with external computer components. The system bus includes, the address bus, the data bus, and the control bus.

Multicore CPUs have an ALU and CU for each core. Also, on certain architectures, each core has its dedicated cache (L1, L2, and L3), whereas on other architectures, the cores may share the third level cache.

### 3.4.3 Processor Instruction Set

The Processor Instruction Set (PIS) is the set of machine instructions that a given processor is able to decode and execute (supported instructions). There are two main classes: CISC and RISC.

- **Complex Instruction Set Computer.** Is a computer architecture in which single instructions can execute several low-level operations. Examples of such processors include, but not limited to: PDP-11 and VAX architectures, Motorola 6800, 6809 and 68000-families, the Intel 8080, iAPX432 and x86-family, the Zilog Z80, Z8 and Z8000-families, and the Intel 8051-family. CISC processors have the following features:
  - Requires less instructions for a given code. This is due to the complex nature of instructions in CISC. One single instruction encodes multiple sub-instructions or operations.
  - Microprogramming is easy to implement. This includes writing firmware, such as the BIOS.
  - CISC processors are larger as they contain more transistors. More transistors means more heat dissipation.

  - The execution of one instruction may take multiple clock cycles, decreasing efficiency.

- **Reduced Instruction Set Computer.** Is a computer architecture in which each single instruction can execute one single low-level operation. Examples of such processors include, but not limited to: ARC processor, the DEC Alpha, the AMD Am29000, the ARM architecture, the Atmel AVR, Blackfin, Intel i860, Intel i960, LoongArch, Motorola 88000, the MIPS architecture, the PA-RISC, the Power ISA, the RISC-V, the SuperH, and the SPARC. RISC processors have the following features:

  - Requires more instructions for a given code. One instruction encodes one single operation.
  - Greater performance due to simplified instruction set.
  - Used in supercomputer, such as the Fugaku.
  - Less expensive, as they use smaller chips.
  - Less transistors, so less heat dissipation compared to CISC, hence their application on small devices, e.g., smartphones and tablets.

Nowadays, we rather talk about x86 processors Vs ARM processors instead of CISC Vs RISC processor. This is because most processors used these days are either Intel-based (i.e., x86) or ARM-based.

### CISC Instruction Set

In this subsection, we briefly discuss some assembly instructions for CISC x86 computer achitecture. These instructions, once complied into machine code (binary), they are correctly interpreted by most CISC processors:

| | |
|---|---|
| `MOV Ax, 0x1111` | Moves the value 0x1111 to register Ax |
| `MOV Al, 0x11` | Moves the value 0x11 to the lower part of Ax |
| `INC Ax` | Increments the current value stored in Ax |
| `DEC Ax` | Decrements the current value stored in Ax |
| `DIV Bx` | Divides the content of Ax by Bx and store it in Ax |
| `MUL Cx` | Multiplies the content of Ax with Cx and store it in Ax |
| `PUSH Ax` | Pushes the content of Ax onto the stack (SP=SP-2) |
| `POP Ax` | Pops the top of the stack (pointed by SP) into Ax |
| `JMP ABCD` | Jumps to label ABCD |
| `CMP Ax, 1` | Compares the content of Ax with value 1 and sets the Flags |
| `JE ABCD` | Jumps to ABCD if Ax was equal to 1 (previous operation) |
| `LOOP ABCD` | Jumps to ABCD and decrements the Cx while Cx is non-zero |
| `HLT` | Terminates the program |

You are not required to learn all the CISC assembly by heart, but I just want you to see how some basic CISC instructions look like. Also, if you are given a small **assembly program** [5] you will be able to tell what the program does.

**Example.** Consider the following x86-assembly program where the final result is stored in the `ax` register. Use the above instructions description and find out what the below code does (you can try it online too).

```
Start:
MOV AX, 0x0001
MOV CX, 0x0001
T:
 MUL CX
 INC CX
 CMP CX, 0x0006
 JBE T     //Jump if CX is below or equal to 6
HLT
```

[5] An assembly code is the code of a program that is written using the assembly language — either CISC assembly or RISC assembly. The program needs one further compilation step to become binary code (machine code).

If you walk through this program, you will see that you are iterating over `MUL CX`, `INC CX`, and **CMP CX, 0x0006**, six times. In each iteration, you multiply the content of the `AX` register by the content of `CX`, which keep incrementing after each iteration. That is, you are computing `AX`=$1 \times 2 \times 3 \times 4 \times 5 \times 6$. At the end of the execution of this code, the `CX` register will contain the value `0x0007`, breaking them the `JBE T` condition (jump if below or equal) and going to `HLT`. The accumulator register `AX` will contain the value `0x02D0`. This latter value corresponds to the decimal value of 720, which is the factorial of 6. Hence, this assembly program computes the factorial of 6.

---

**Exercise.** What is the value that will be stored in the memory at address `0x111F` after the below assembly code is executed? Remember, `DIV CX` is equivalent to `AX = AX` $\div$ `CX`.

```
start:
MOV AX, 0x0002
MOV BX, 0x0030
ADD AX, BX
MOV CX, 0x000A
DIV CX
INC AX
MOV [0x111F], AX
HLT
```

a) `0x0005`

b) `0x0002`

c) `0x0007`

d) `0x00E0`

e) `0x0008`

---

The solution of the exercise will be discussed during the lecture.

### RISC Instruction Set

In this subsection, we briefly discuss some assembly instructions for RISC computer achitecture. These instructions, once complied into machine code (binary), they are correctly interpreted by most RISC processors:

- `LDR R0, =0x1b25` //Load register R0 with the value `0x1b25`.
- `MOV R0, ♯10` //Move the decimal value `10` to register.
- `ldr r1, [r0]` //Load register R1 with content from the memory. The address of the memory location to load from is contained in register R0.

• `ADD R1, R2, ♯10` //Add the `10` to register R2 and store result in R1.

• `STR R2, [R0]` //Copy content of R2 and store it in memory at address contained in register R0.

• `JMP T` //Jump to label "T". This can be used in loops.

• `CMP R0 R1` //Compare the content of R0 and R1 . Actually, R1 is substracted from R0, and based on the result, some flag in the state (flag) register are set, e.g., the zero flag, the sign flag, etc.

• `BEQ T` //Branch to label "T" if equal, i.e., if the zero flag is set (e.g., if R0=R1 in the previous `CMP` instruction) .

• `SUBS R0, R0, ♯10` //Substract and set the flags. The value `10` is substracted from the content of R0. The result is stored into register R0. Based on the result, the flags are set at the flag register.

In what follows, most of the examples given in this course reader are expressed using CISC x86 assembly language (i.e., use the x86 instructions set) on a 16-bit computer.

### 3.4.4 Machine Code

Machine code is a sequence of bytes, e.g., 01010101 01010111...01010100, that can be interpreted and executed by a dedicated CPU. In fact, each instruction in the assembly language, e.g., `MOV AX, BX`, is transformed into a corresponding machine code by the **assembler program** (e.g., Netwide Assembler a.k.a., nasm) to be executed by the processor.

Each instruction in machine code is generally composed of an **Operation code** (`opcode`) and **operand**. For example:

- In `MOV AX, [0xF565]`, the keyword `MOV` is the opcode whereas the two parameters `AX` and `[0xF565]` represent the operand. Here, we are moving the content in the memory at address `0xf565` into the `AX` register.
- In `JMP T`, the opcode is the keyword `JMP` whereas the letter "T" is a label. Here, the instruction aim to perform a jump from the current instruction into the instruction where the label is defined in the code.

Some instructions however, just consists of an **operation code**. For example:

- In `NOP`, the keyword `NOP` is the opcode for no operation. It asks the CPU to do nothing and just move to the next instruction in the program.
- In `HLT`, the keyword `HLT` is an operation code to stop execution. It generally indicates the end of the program that is being executed.

It is important to note that instructions may have different sizes, e.g., in Intel, 1 byte to 14 bytes. This means that one instruction may be stored in multiple memory locations if one memory location cannot hold the entire instruction code.

### 3.4.5 Instruction Execution Cycle

Generally, an instruction is composed of three to four units: (1) **Fetch**, where a program's instruction is retrieved from the central memory and brought to the processor. At that moment, the address of the instructions stored in the PC (Program Counter) register — a.k.a., instruction pointer (IP). The address of the instruction is placed on the MAR (Memory Address Register) then a read-request is issued to the central memory controller to retrieve the content stored the indicated address (i.e., [PC]). After a memory access time, the content (which is the retrieved instruction) is placed into the MBR (Memory Buffer Register). The instruction is placed into the current instruction register (CIR). (2) **Decode**, where the instruction is decoded within the processor (by the control unit — CU), and transformed into a sequence of elementary operations. If the instruction requires operands from memory, the CPU gets them in the MBR after issuing a fetch operand operation. The operand is stored in one of the general purpose registers and the PC is updated (i.e., generally, incremented, to point to then next instruction of the program). (3) **Execute**, where the instruction is executed. The ALU executes the instruction and the state register is updated (i.e., FLAGS register). (4) **Write back**, where the result is written back to the central memory if required by the instruction.

These four instruction units, i.e., fetch, decode, execute, and write back, are known as **machine cycle**.

As we have seen before, instructions could be:

- Data transfer: from and to memory or between registers, e.g., `MOV AX, [0x52F3]` or `MOV AX, BX`.

- Arithmetic operation: Addition, subtraction, division, and product, e.g., `DIV AX, CX`, `SUB AX, CX` , or `ADD AX, BX`
- Logical operation: AND, OR, NOT, and comparison, e.g., `CMP AX, 0x0001` or `XOR Ax, Ax`.
- Sequence control: Branch and tests, e.g., `JE X` or `JMP X`.

**Example.** Consider the following CISC assembly code. The code may have been generated from a high-level code that computes the addition of two integer variables, `A` and `B`, and stores the result in another variable `C`. The value of the variables A, B, and C are stored the memory locations at `[0xA522]`,`[0xA523]`, and `[0xA524]`, respectively.

| Code | RAM | |
|---|---|---|
| ... | Address | Content |
| ... | ... | ... |
| ... | [0xA522] | 0x002C |
| ... | [0xA523] | 0x0100 |
| MOV AX, [0xA522] | [0xA524] | 0xFB88 |
| MOV BX, [0xA523] | ... | ... |
| ADD AX, BX | [0x5100] | MOV AX, [0xA522] |
| MOV [0xA524], AX | [0x5101] | MOV BX, [0xA523] |
| ... | [0x5102] | ADD AX, BX |
| ... | [0x5103] | MOV [0xA524], AX |
| ... | ... | ... |

Initially, the program counter register (`PC`) contains the address `[0x5100]`, which is the address of the first instruction, i.e., `MOV AX, [0xA522]`. A **fetch** operation takes place by placing the address `[0x5100]` in the memory address register (`MAR`) and by issuing a read request to the central memory (RAM). After an access time, the requested content is placed into the memory buffer register (`MBR`) and sent back to the CPU. As the content is an instruction, the latter is placed in the current instruction register (`CIR`) to start decoding it. The control unit starts the `decode` operating to understand the meaning of the instruction. The instruction (`MOV AX, [0xA522]`) requires an operand to be retrieved from the memory and placed into the accumulator register (`AX`). At this point, the CPU updates the `PC` register by incrementing it so that it points to the next instruction (`0x5101`). The address of the operand

(0xA522) is placed into the MAR and a read request is issued. The content (i.e., 0x002C) is brought back to the CPU through the MBR and placed into the AX register as requested by the instruction.

Now, the CPU moves to the second instruction. A **fetch** operation takes place by placing the address [0x5101] in the memory address register (MAR) and by issuing a read request to the central memory (RAM). After an access time, the requested content is placed into the memory buffer register (MBR) and sent back to the CPU. As the content is an instruction, the latter is placed in the current instruction register (CIR) to start decoding it. The control unit starts the decode operating to understand the meaning of the instruction. The instruction (MOV BX, [0xA523]) requires an operand to be retrieved from the memory and placed into the base register (BX). At this point, the CPU updates the PC register by incrementing it so that it points to the next instruction (0x5102). The address of the operand (0xA523) is placed into the MAR and a read request is issued. The content (i.e., 0x0100) is brought back to the CPU through the MBR and placed into the BX register as requested by the instruction.

Next, the third instruction. A **fetch** operation takes place by placing the address [0x5102] in the memory address register (MAR) and by issuing a read request to the central memory (RAM). After an access time, the requested content is placed into the memory buffer register (MBR) and sent back to the CPU. As the content is an instruction, the latter is placed in the current instruction register (CIR) to start decoding it. The control unit starts the decode operating to understand the meaning of the instruction. The instruction (ADD AX, BX) instructs the CPU to take the content stored in the register AX and the one stored in the register BX, sum them together, and store the result in the AX register. At this point, the CPU updates the PC register by incrementing it so that it points to the next instruction (0x5103). The ADD instruction will involve the ALU of the CPU to perform the addition. The flag register will also be updated accordingly — if required (e.g., carry flag, zero flag, etc). After the addition (execute operation), the value 0x0100 + 0x002C = 0x012C is stored in AX and the CPU moves to the next and last instruction (i.e., MOV [0xA524], AX).

Similar to the previous instruction, the last instruction is brought to the current instruction register (i.e., **fetch** operation). The last instruction tells (**decode**) the CPU that it has to transfer the content of AX register to the central memory at address [0xA524], which is the address of the variable C in our high-level code. The PC register content is incremented, the content

of the register `AX` is placed into the `MBR` register, and a write operation is issued (**write back**). After an access time, the value `0x012C` is placed in the memory location of address `[0xA524]`.

### 3.4.6 Instruction per Clock Cycle

The instruction per clock cycle (IPC) refers to the average number of instructions that a CPU can simultaniously perform per single clock cycle (CC). Eariler processors (called, **sequential** CPUs) performed one machine cycle (i.e., fetch, decode, execute, or write back) per clock cycle (⊓⊔). I.e., around three or more clock cycles per simple instruction (certain complex instructions may take more than three clock cycles). Table 3.1 illustrates the execution of two instructions in a sequential processor. Each instruction takes four (04) machine cycles, the execution requires eight (08) clock cycles.

| **Instruction** | **1** | | | | **2** | | | |
|---|---|---|---|---|---|---|---|---|
| Fetch | | | | | | | | |
| Decode | | | | | | | | |
| Execute | | | | | | | | |
| Write back | | | | | | | | |
| Clock (⊓⊔) | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |

Table 3.1: Instructions execution in sequential processors

The next processors, knowns as **pipelined** CPUs, performed multiple machine cycles per clock cycle. E.g., executing I1, decoding I2, and fetching for I3. Table 3.2 illustrates the execution of four instructions in a piplined processor, where one clock cycle can handle up to four machine cycles.

If we take a close look at the 4th cycle in Table 3.2 , we can see that the processor is handling four (04) different machine cycles, regardless of which machine cycle belongs to which instruction. As one complete instruction consists of four (04) machine cycles (i.e., fetch, decode, execute, or write back), we can claim that pipelined processors can execute one instruction per clock cycle.

| **Instruction** | **1** | **2** | **3** | **4** | | | | |
|---|---|---|---|---|---|---|---|---|
| Fetch | | | | | | | | |
| Decode | | | | | | | | |
| Execute | | | | | | | | |
| Write back | | | | | | | | |
| Clock (⊓⊔) | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |

Table 3.2: Instructions execution in piplined processors

A core principle of RISC architecture is to execute most instructions in a single clock cycle. This was a key design goal to improve performance compared to complex instruction set computers (CISC). However, certain instructions (complex instructions, e.g., multiplication and division) may take several clock cycles to execute (e.g., `call` takes 18 clock cycle on an MCS-51 processor). Reading a 16-bit data from the central memory using an 8-bit data bus may take several cycles too.

Now, modern processors, the ones we use today (2024), known as **superscalar** CPUs, can execute multiple instructions per clock cycle. They use ILP (Instruction-Level Parallelism). Some of the processors, e.g., Apple's Silicon M1, uses **out-of-order execution**, which is a technique to maximize the utilization of the processor by executing instructions in an order different from the original program sequence.

### 3.4.7 Single and Multiprocessors Systems

Based on the number of processors are system may have, systems are classified into two categories: single-processor and multiprocessors systems.

#### Single-Processor Systems

In a single-processor system, there is only one **general-purpose** CPU (Central Process Unit) capable of executing general-purpose instruction sets. Such systems may certainly have other **special-purpose** processors such as device-specific processor (or controllers), or a general-purpose processor on mainframes. Generally, a system with $n$ special-purpose processors and one general-purpose processor is a considered to be a single-processor system.

Among the most important special-purpose processors that we can find in todays systems, the **Graphical Processing Unit**, a.k.a., **GPU** (viz.,

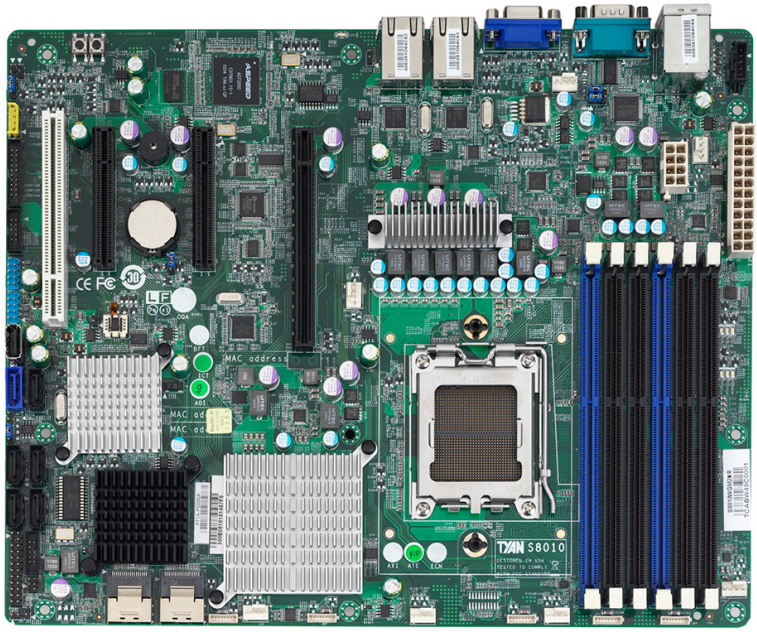



Figure 3.7: Motherboard of a Single-Processor System


Figure 3.8: Graphical Processing Unit — GPU (Left) and Google's Tensor Processing Unit — TPU (Right)

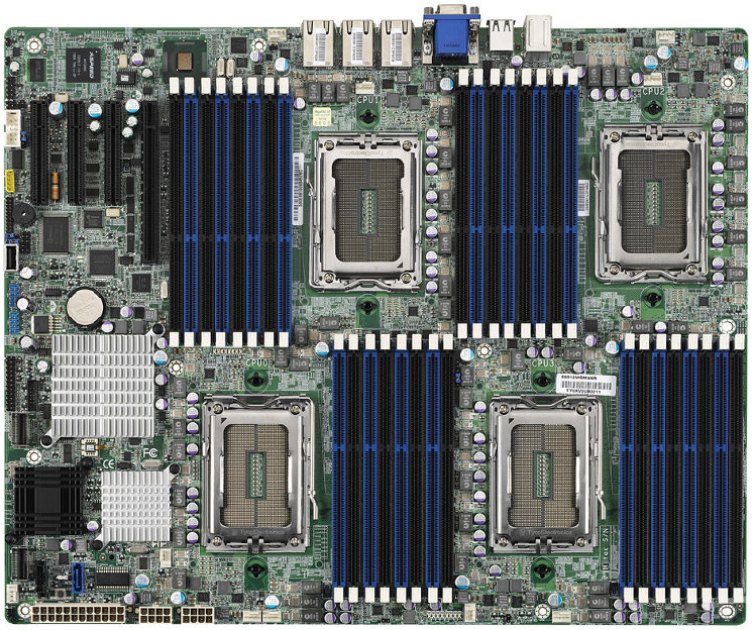


Figure 3.9: Motherboard of a Multi-Processor System

Figure 3.8-Left). It Is a special-purpose processor used for processing visual and graphical-related tasks. This is critical for users like, gamers, and graphical-content creators. As they are optimized for parallel processing, they are used a lot for scientific simulations, cryptanalysis, and AI applications (model training).

In 2015, Google developed **Tensor Processing Unit**, or **TPU**. This special-purpose processor that is specifically designed and optimized for machine leanring workloads (viz., Figure 3.8-Right).

Finally, another class of processors is the **Quantum CPUs**. Quantum CPUs represent the next frontier in computing. Unlike traditional CPUs that use bits (0s and 1s), quantum computers utilize **qubits**, which can exist in multiple states simultaneously. This quantum property allows for exponentially faster calculations for certain types of problems, particularly those related to optimization, cryptography, and materials science.

### Multiprocessor Systems

In a multiprocessor system, there are two or more CPUs (viz., Figure 3.9) capable of executing a general-purpose instruction set, and sharing the computer bus, poissibly the clock, the memory, and peripheral devices. This type of systems initially appeared in servers (powerful computers) then migrated to desktops, laptops, and recently smartphones and tablets. Nowadays, the are cheaper than multiple single-core systems and they provide better performance and require less cost.

Modern CPU design include multiple computing cores on a single chip, they are called **multicore**, e.g., Intel Pentium D (2005) dual-core, Core i3, i5, and i9, AMD RYZEN 3, Apple Silicon M1/2, etc.

Multiprocessors also come in the form of a network of computers. When computers are connected together using a local network, they form a **clustered system** (Figure 3.10). Each of the computers in the cluster can be either be a single-processor system or a multicore system.

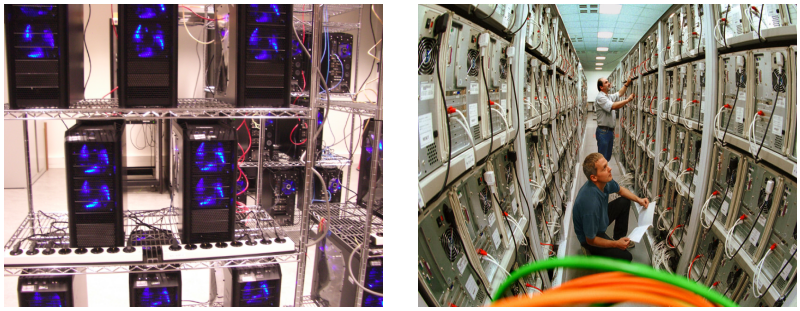


Figure 3.10: Cluster Systems

Clusters emerged as a viable computing architecture when advances in technology made it economically feasible to connect multiple microprocessors via high-speed networks. Coupled with the development of software to coordinate these interconnected systems, clusters became capable of delivering enhanced performance, fault tolerance, and scalability. Today, clusters are widely used for tasks such as shared storage, high availability services, and demanding high-performance computing applications.

## 3.5 Central Memory

The central memory or **RAM** (Random Access Memory), is a **volatile** type of memory circuit that is used to store information (e.g., text, programs, videos, images, etc). Technically, it stores bits (or bytes).

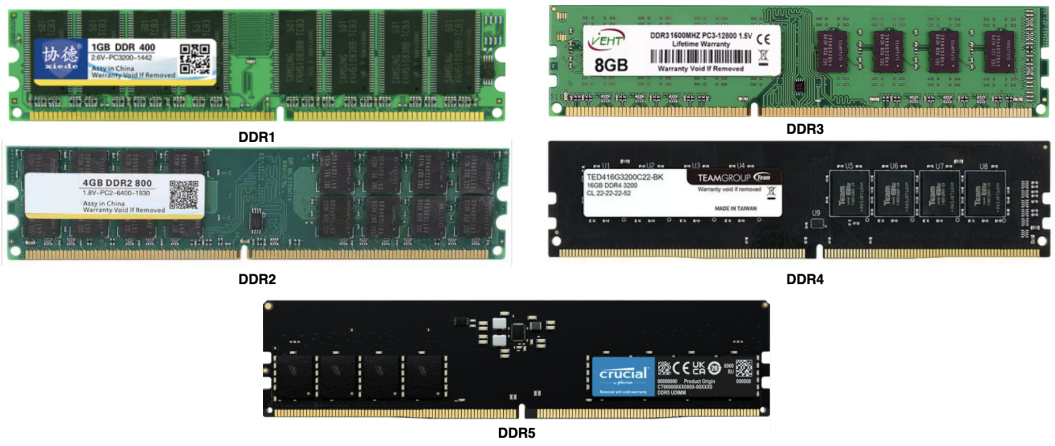



Figure 3.11: Different technologies for RAM

RAMs are mainly featured by their: **frequency** (the number of operations it can perform per second), **storage capacity**, **data rate** (the amount of bits that can flow in and out of the RAM per second), and **latency** (the response time). They are connected to the motherboard using the **DIMM** (Dual In-line Memory Module) slots.

RAMs have undergone considerable development. There are various technologies: SDR SDRAM, DDR SDRAM, RDRAM, DDR2, DDR3, DDR4, and DDR5, LPDDR, and GDDR. They differ by their frequency, storage capacity, data rate, and latency.

### 3.5.1 Memory Classification

Depending on their nature, a memory can either be a RAM or a ROM:

**RAM (Random Access Memory)**: This type of memory requires constant power to keep its content. If power is out, data is lost. We can find two categories:

- **Static RAM**: or SRAM, uses latches components (flip-flops) to store data. It has smaller storage capacity but runs faster. It is mainly used to implement CPU cache.
- **Dynamic RAM**: or DRAM, uses capacitors to store data (charged capacitor encode bit 1, discharged capacitor encode bit 0). As it uses capacitors, DRAMs require frequent refreshment to keep the data. They have larger storage-capacity, but run slower than SRAMs. They are typically used to implement the central memory.

**ROM (Read Only Memory)**: This type of memory does not require power to permanently keep its content. If power is out, data remain in the memory. We can find two categories:

- **Programmable**: This type of ROM can be erased and reprogrammed multiple times, e.g., EPROM and EEPROM. They are used to implement memory-sticks and pendrives.
- **Non-Programmable**: These cannot be programmed twice, e.g., ROM and PROM (only once, after manufacturing). They were used to store firmware, e.g., BIOS, and data (e.g., CD-ROM).

### 3.5.2 Memory Interface

The interface of a memory is composed of three main components:

- **Address Size.** The size of one address is expressed in `M` bits $(b_0, \ldots, b_{M-1})$. For example, the hexadecimal value `0x34f5` could represent a 16-bit address in a 16-bit computer system.
- **Word Size.** The size of one memory location is `W` bits $(d_0, \ldots, d_W)$. Generally, if one memory location stores 8-bits (1 byte), we refer to the memory as a **byte-addressable** memory (i.e., `W=8` bits). If the memory stores more or less than that, it is a **word-addressable** memory.
- **Control Signal.** Used to issue read or write operation signals.

**Example.** A memory that has `M=4` and `W=8` is a `16 Byte`-size memory.

There are 16 possible addresses ($2^M = 2^4 = 16$):

| | |
|---|---|
| `0000,0001,0010,0011,` | `0x0,0x1,0x2,0x3` |
| `0100,0101,0110,0111,` | `0x4,0x5,0x6,0x7` |
| `1000,1001,1010,1011,` | `0x8,0x9,0xA,0xB` |
| `1100,1101,1110,1111.` | `0xC,0xD,0xE,0xF` |

To understand the concept of central memory, you need to perceive it as a vertical chest of drawers. Each drawer represents on memory location that stores `W` bits (the word-size). Also, each memory locaiton (drawer) is uniquely identified by an ID, known as the address. Addresses (drawer IDs) have the same sizes, i.e., address size on `M` bits. For example, we had a byte-addressable memory, i.e., `W=8` bits, we can store one single byte (8 bits) per single drawer. Furthermore, if the size of one address is encoded on `M=16` bits, this would mean that we need 16 bits to encode an address. Hence, we can encode $2^{16}$ possible addresses. We can theoritically claim that the memory has $2^{16}$ memory locaiton (drawer). The total size of the memory would be $2^{16} \times 8 = 64$ Kbyte. In the next paragraph, we explain how we did obtain this result.

When the value of `M` is greater than 10, multiples of the bytes (octets) are used:

- For `W=8 and M=10`, a memory has a size of `1 Kilobyte`.

$$2^{10} \times 8 = 1024 \times 8 = \texttt{K} \cdot \texttt{Byte} = 1 \text{ KB}$$

- For `W=8 and M=20`, a memory has a size of `1 Megabyte`.

$$2^{20} \times 8 = \texttt{M} \cdot \texttt{Byte} = 1 \text{ MB}$$

- For `W=8 and M=30`, a memory has a size of `1 Gigabyte`.

$$2^{30} \times 8 = \texttt{G} \cdot \texttt{Byte} = 1 \text{ GB}$$

- For `W=8 and M=40`, a memory has a size of `1 Terabyte`.

$$2^{40} \times 8 = \texttt{T} \cdot \texttt{Byte} = 1 \text{ TB}$$

The memory of our previous example had a size of $2^{16} \times 8$ . The value 8 represent 8 bits. Since we want a result expressed in terms of bytes, we would simple convert the 8 bits into 1 byte. Hence, the size become $2^{16}$ bytes. Using the above multiples, we obtain $2^{16} = 2^{10} \times 2^6 = 2^6$ Kilobyte = 64 Kbyte.

---

**Exercise.** Consider a processor (CPU) with a word-size of 16 bits connected to a word-addressable memory (RAM). How much memory space can be addressed in this system?

a) 16 kilo bytes.
b) 32 kilo bytes.
c) 64 kilo bytes.
d) 128 kilo bytes.
e) 256 kilo bytes.
f) 512 kilo bytes.

---

The solution of the exercise will be discussed during the lecture.

### 3.5.3 Permanent and Backup Storage Memories

There are different types of storage media for permanent storage, backup, and data archiving:

- **Magnetic tape**: Tapes are sequential access medium. They appeared in 1951 and are still being strictly used for archive and backup purposes. Tapes were used for the very first time to record computer data in the UNIVAC I. The tapes were metallic and 1200 feet long, very heavy, the data was coded into magnetic substances. They became more practical after switching from metal tape to plastic tape. Nowadays magnetic tapes can hold up to `580 TB` (2020).

Figure 3.12: Magnetic tapes

Magnetic tapes utilize magnetization to encode data in a binary format. The tape is coated with a thin layer of magnetic material, typically iron oxide. A write head, equipped with an electromagnet, is used to magnetize the magnetic coating on the tape. The direction of the magnetic field determines the bit value. That is, if the direction of the magnetic field is north pole, then it encodes a bit 1, otherwise (south pole), it is a 0.

- **Hard Disk Drive**: Appeared in 1956 and still being used. They are faster than tap cartridge but support less storage capacities. Nowadays HDD can hold up to `22 TB` (2022). Most computers have HDD as secondary storage. Hard disk drives also use magnetic technology to encode bits over disk surfaces.

Figure 3.13: Hard Disk Drives

- **Optical disk**: Optical discs store data by creating tiny pits and lands on a reflective surface. To encode 0s and 1s on the disc surface, a specific laser (infrared, red, or blue-violet) is applied on the surface to

burn specific spots and create pits (encoding 0). To encode one, the spot is left as is. The sharper was the laser (i.e., had higher frequency, e.g., blue-violet laser), the thinner was the pits and lands, and the larger was the disc storage capacity (Blu-ray holds up to 25GB of data).


Figure 3.14: Different Oprical Disc

To read data, a same-type laser reads the data by detecting whether the laser light is reflected (land, representing a binary 1) or not (pit, representing a binary 0) as the disc spins. While offering high storage capacity, optical discs are susceptible to scratches and degradation, making them slower to access data compared to other storage technologies and less suitable for frequent data modification.

- **Solid state storage**: A.k.a, flash memory. They primarily use NAND flash memory.to implement their circuitry. They are used in thumb drives, secure digital cards, memory sticks, etc. Unlike traditional Hard Disk Drives (HDDs), SSDs have no moving parts, making them faster, more reliable, and quieter. They are being adopted by modern laptops.


Figure 3.15: Solid State Drives

- **Floppy disk**: Used in 1980s and early 1990s, it has low data rate and low storage capacity (less than `3MB`). They are no longer being used.


Figure 3.16: Floppy Disk

## 3.6 I/O Devices

Input and output devices, a.k.a., **peripherals**, are computer electronic components used to input data into the computer and output data out of it. Some devices support both mode, input and output. E.g., touchscreens.

Figure 3.17: Input and Outout Devices

Input and output devices are connected to the motherboard through either **expansion cards** or **device controller cards** using various ports, including USB, HDMI, VGA, RJ11, etc. Expansion cards and device controller cards are connected to the motherboard using **slots** and **connectors**. Also, peripherals are plugged into the motherboard through various port, e.g., VGA, HDMI, RJ-45, PS/2, USB, DB25, etc.

## 3.7 Brief History of Computers

The evolution of computers is a testament to human ingenuity. Originating from mechanical devices like Babbage's Difference Engine (1822, by Charles Babbage — UK), designed for specific calculations tabulate polynomial functions (Figure 3.18-Left), computing progressed to electromechanical machines utilizing relays and gears. A pivotal leap occurred with the advent of electronic computers. The ABC (by John V. Atanasoff and his graduate student, Clifford Berry, in 1942 — USA) was the first electronic computer (Figure 3.18-Right). It was designed to solve problems with up to 29 different

variables (one operation per 15 seconds). Of course, before ABC, some other computers were designed, but were electro-mechanical machines. E.g., The Torpedo Data Computer (1938) — in the USA, and the Z-series: Z2 (1940), Z3 (1941), and Z4 (1950) — in Germany.

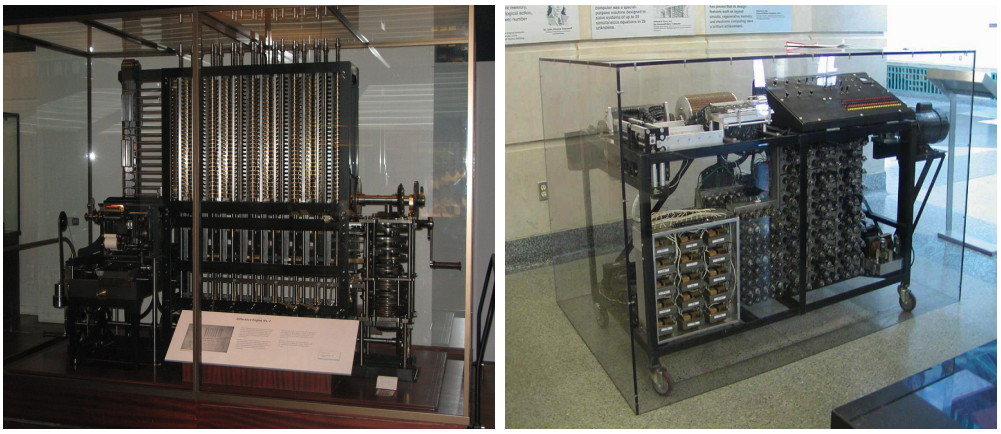


Figure 3.18: The Babbage Differential Engine (Left) and the Atanasoff–Berry Computer (Right)

In 1945, the ENIAC (Electronic Numerical Integrator And Computer, by the University of Pennsylvania, in 1945 — USA), which introduced programmability and marked the beginning of the modern computing era. It was claimed to be the first **turing-complete** (general-purpose) computer.

Building upon these foundations, today's computers have evolved into immensely powerful and versatile machines capable of handling vast amounts of data and complex computations, reshaping industries and societies worldwide. Computers like ENIAC used IBM's **punchcards** (or **punched cards** — viz., Figure 3.20) to input and output data. While the concept of punched cards predates electronic computers, their widespread adoption as a primary input and output method for these machines began in the 1940s and 1950s.

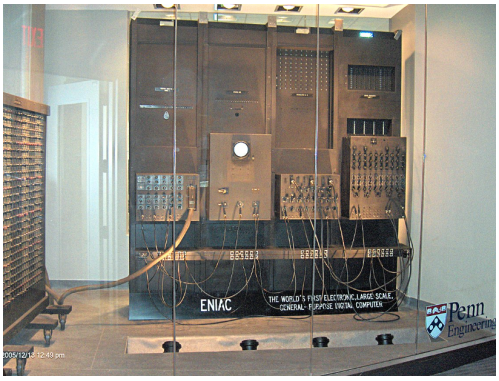



Figure 3.19: Electronic Numerical Integrator And Computer (ENIAC)

Punchcards are pieces of stiff papers that contain digital information represented by the presence or absence of holes in predefined positions (viz., Figure 3.20). These holes were made by a keypunch machine. The pattern of holes represented data or instructions that could be read by a computer or data processing machine. The largest punchcard computer program (1950s) was the SAGE air defense system with 62,500 punchards (4.5 megabyte).


Figure 3.20: A punchard encoding a Hello World message

Computers in the '40s and early '50s ran one program at a time. The manual process of punching cards and feeding them to the machine worked Okay, as the computers were slow. Yet, when computers became faster, that manual process was taking longer. Also, the accidental errors that programmers were facing, such as card jamming and key punching errors, were waste of time. It considerably affected productivity.

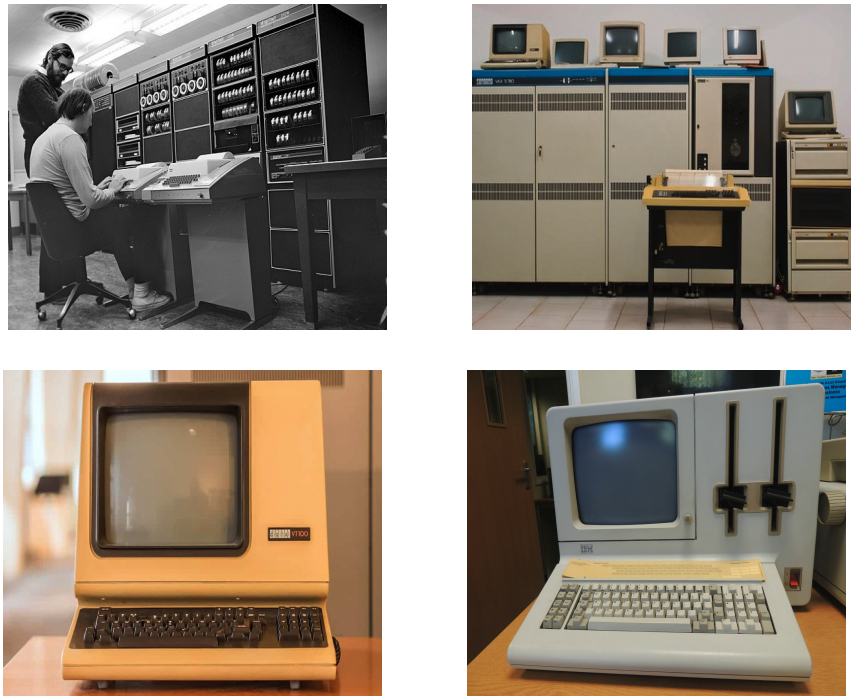


Figure 3.21: From left to right: PDP-11, VAX-11 by Digital Equipment Corporation (DEC), DEC's '70s computer, and IBM's '70s computer.

In the era of one-off computers, programmers only had to write code for that single machine. With the proliferation and diversification of computers, their configurations were not always identical. For examle, a program developed for a given machine `A` would fail to correctly run on another machine `B` from different brand. This created a huge pain for programmers.

However, when faster computers came in, the need to run multiple programs simultaneously on the same CPU has been raised. There was a need for an independent **program** to schedule the execution of concurrent programs as well to manage shared resources among various programs. Such program is called: **Operating System**. The first instance of such a program was the **GM-NAA I/O**, produced in **1956** by General Motors' Research division for its **IBM 704** mainframe computer.

The 1980s marked a pivotal era in computing, transitioning from large, room-sized mainframes to more accessible personal computers (PCs). This

decade witnessed the birth of iconic machines that laid the foundation for the digital age we know today. Furthermore. the introduction of RGB (Red-Green-Blue) color was a crucial step in enhancing the visual appeal and usability of computers.

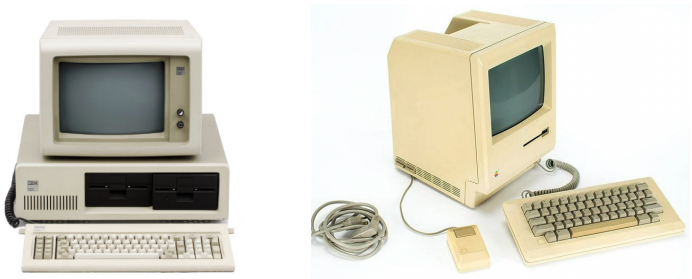


Figure 3.22: IBM 1981 PC (Left) and Macintosh 1984 computer (Right).

The 1990s also marked a significant evolution in computing, characterized by the rapid expansion of personal computers into homes and businesses.

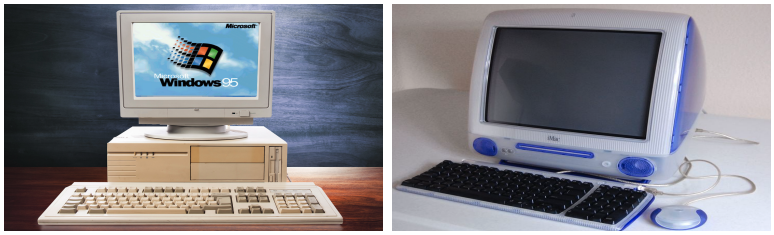



Figure 3.23: '90s IBM's PC (Left) and '90s Apple's iMac (Right).

Today's computers are a far cry from their earlier counterparts. They've evolved into incredibly powerful, portable, and interconnected devices.

Figure 3.24: Nowadays PCs: Gamine Desktop (Left) and a Laptop (Right).

## 3.8 Type of Computers

Computers come in different forms and computing resources for various types of applications. In the follwoing paragraphs, we discuss the most important ones:

- **Supercomputer.** Are computers with high performance. The performance is commonly measured in floating-point operations per second (FLOPS) instead of million instructions per second (MIPS). These computers are used for data-intensive and computation-heavy scientific and engineering purposes such as oil and gas exploration, molecular modeling, physical simulations, aerodynamics, nuclear fusion research, and cryptoanalysis. Figure 3.25 illustrates two supercomputers.

Figure 3.25: Control Data Corporation's CDC 6600 supercomputer 1964 (Right) and IBM's Blue Gene/P supercomputer 2007 (Left).

- **Mainframes.** Are powerful and large computers but not as large as supercomputers (Figure 3.26). These computers are used primarily by large organizations for critical applications like bulk data processing for tasks such as censuses, industry and consumer statistics, enterprise resource planning, and large-scale transaction processing.

Figure 3.26: IBM's 1401 mainframe '60s (Right) and IBM's Z15 mainframe 2019 (Left).

- **Server.** Is a computer that runs a service. It does not necessarily have to be a powerful computer. It could be your PC or a Raspberry Pi.

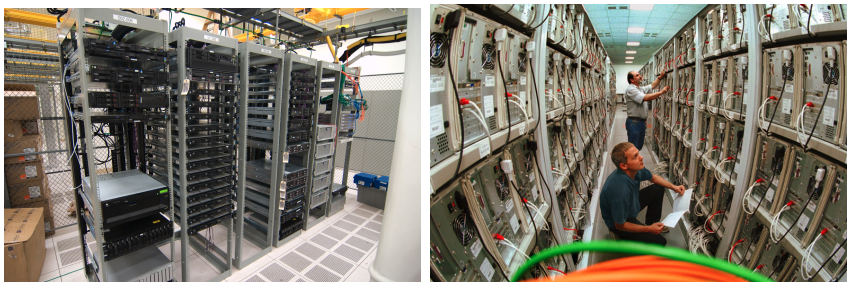


Figure 3.27: Cluster of servers (Right) and server racks (Left).

- **Laptop.** You know this one! Just a portable small-sized personal computer. E.g., HP's ProBooks, Apple's MacBooks, etc.

- **Desktop computer.** Sometimes — workstation. Is actually any computer that can fit on top of a table.

- **Things.** Any object that can perform computation. From a lightbulb, thermostate, to a self-driving vehicle.

## 3.9 Computers, Single-board computers, and Microcontrollers

Computers, single-board computers, and microcontrollers are computing devices with varying capabilities and applications. A computer is a general-purpose machine with extensive resources, while a single-board computer is a smaller, self-contained system often used for development or specific tasks (e.g., Raspberry Pi). Microcontrollers, the smallest of the three, are highly specialized chips designed for embedded systems with limited resources, focusing on control and automation. They integrate the processor and the central memory in one chip, called microcontroller.


Figure 3.28: Computers (laptop and tiny computer — on the left), Single-board computers (Raspberry Pi — top right corner), and Microcontrollers (Arduino board — bottom right corner).

# Chapter 4

# Process Management

In this chapter, we will explore the core processes that drive the operation of an operating system. We will delve into concepts such as processes, Process Context Blocks (PCBs), interrupts, context-switching, concurrent processes, threads, and Direct Memory Access (DMA). By understanding these foundational elements, we will lay the groundwork for exploring the more advanced features and techniques employed by modern operating systems.

## 4.1 Processes and Programs

When a program is executing (it is running) on a computer, we call it a **process**. A process can be preceived as the **active** version of a program. Every process is uniquely identified by a **PID (Process ID)**. On Windows PCs, you can use the command **taskmgr** to display the running processes (Windows task manager). As you can see in Figure 4.1, the first column (`image name`) shows the name of the process, the second one its PID (e.g., notepad.exe has PID 3096), the third one indicates the user who initiated the process (OlsenG), the fourth one show the CPU utilization in percent, the fifth one indicates the memory utilization (1420 Kbyte for notepad.exe), and the last column provides some addition description as the name may not reveal that much information.

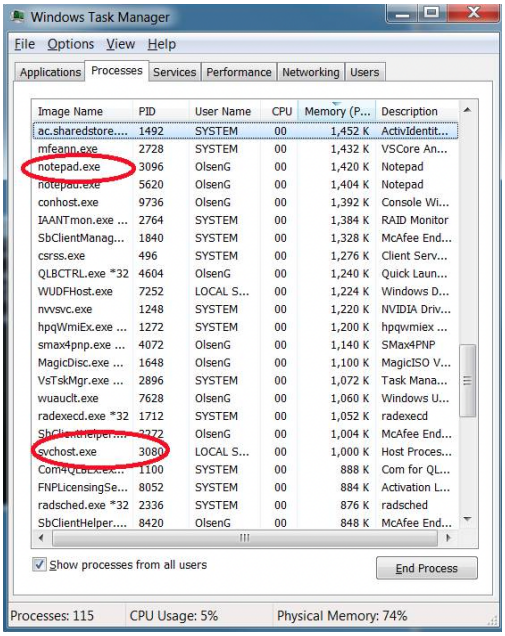



Figure 4.1: Windows task manager

**Process**

A process is an instance of a computer program that is being executed.

A process consists of:

- **An address space.** This contains the program code and the its data (i.e., global variables, heap, and stack).
- **The CPU state.** The values of CPU registers (e.g., PC, AX, BX, CX, EX, BP, FLAGS, ...,SP, etc).
- **A set of resources.** This includes open files, network connections, ...
- **A lifetime.** This refers to the different states in which the process has been: created-executed-[interrupted-resumed]-terminated.

Overall: A process is all what you need to run or resume a program.

There are two types of processes: **system processes** that execute system codes, and **user processes** that execute user codes (but treated in the same

way). System processes are executed in kernel mode (master mode — system mode) whereas user processes are executed in user mode (slave mode).

You can terminate any process by selecting it and hitting the "End Process" button (blocked processes, free space, ...). Similarly, on GNU/Linux, you can use the command **ps** or **top** to display the running processes — viz., Figure 4.2. You can terminate a process by identifying their PID and running the command `kill -9 PID` on the terminal.

Some **processes** can be divided into multiple independent execution flows, called **threads** (or **lightweight process**). On GNU/Linux, you can type `ps -aux | grep "Google Chrome"` to display threads related to the Google Chrome browser:

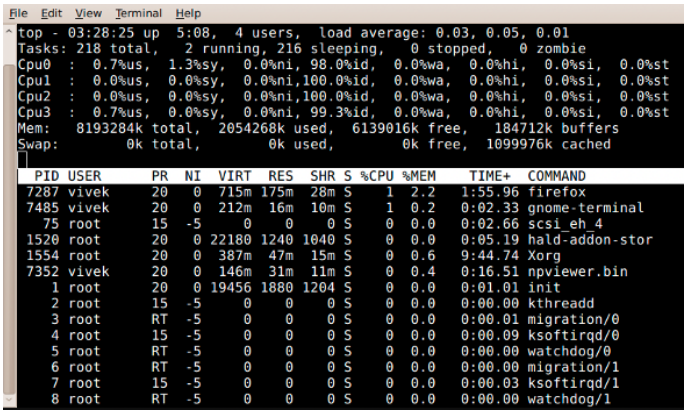



Figure 4.2: GNU/Linux task manager (top command)

```
1372 ??         0:02.39 /System/Library/Frameworks/ApplicationServices.framework/Frameworks/SpeechSynthesis.framework/Resources/com.apple.speech.speechsynthesisd
1390 ??         0:08.80 /System/Library/PrivateFrameworks/CoreLSKD.framework/Versions/A/lskdd
1467 ??         0:03.38 /System/Library/CoreServices/sharedfilelistd
1572 ??         0:24.34 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
1573 ??        29:37.86 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
1574 ??        53:21.01 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
1575 ??        15:30.98 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
1586 ??       128:41.49 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
1595 ??       156:14.74 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
1787 ??         9:35.72 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
1794 ??         0:37.19 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
1820 ??         0:06.54 /Applications/Google Chrome.app/Contents/Frameworks/Google Chrome Framework.framework/Versions/116.0.5845.187/Helpers/Google Chrome Helper (Renderer).app/Contents/MacOS/Google
2519 ??         0:02.55 /System/Library/PrivateFrameworks/CoreFP.framework/Versions/A/fairplayd
2581 ??         0:00.70 /System/Library/PrivateFrameworks/IMDPersistence.framework/IMAutomaticHistoryDeletionAgent.app/Contents/MacOS/IMAutomaticHistoryDeletionAgent
2969 ??         1:12.56 /System/Library/CoreServices/PowerChime.app/Contents/MacOS/PowerChime
2976 ??         3:36.54 /usr/sbin/distnoted agent
3058 ??         0:00.44 /System/Library/PrivateFrameworks/CommerceKit.framework/Versions/A/Resources/storedownloadd
3096 ??       512:57.69 /Applications/Adobe Acrobat Reader.app/Contents/MacOS/AdobeReader
3098 ??         1:39.33 /Applications/Adobe Acrobat Reader.app/Contents/Helpers/AcroCEF/RdrCEF.app/Contents/MacOS/RdrCEF */Applications/Adobe Acrobat Reader.app/Contents/Helpers/AcroCEF/RdrCEF.app* --
3106 ??        58:24.75 /Applications/Adobe Acrobat Reader.app/Contents/Frameworks/RdrCEF Helper (GPU).app/Contents/MacOS/RdrCEF Helper (GPU) --type=gpu-process --log-severity=disable --user-agent-pro
3111 ??         8:18.74 /Applications/Adobe Acrobat Reader.app/Contents/Frameworks/AdobeCrashReporter.framework/Versions/A/Adobe Crash Handler.app/Contents/MacOS/Adobe Crash Handler 3096 AdobeReader 2
3113 ??         0:05.15 /Applications/Adobe Acrobat Reader.app/Contents/Frameworks/RdrCEF Helper.app/Contents/MacOS/RdrCEF Helper --type=utility --utility-sub-type=storage.mojom.StorageService --lang=
3116 ??         0:11.15 /Applications/Adobe Acrobat Reader.app/Contents/Frameworks/RdrCEF Helper.app/Contents/MacOS/RdrCEF Helper --type=utility --utility-sub-type=network.mojom.NetworkService --lang=
3118 ??         0:11.95 /Applications/Adobe Acrobat Reader.app/Contents/Frameworks/RdrCEF Helper (Renderer).app/Contents/MacOS/RdrCEF Helper (Renderer) --type=renderer --log-severity=disable --user-ag
3119 ??         0:50.04 /Applications/Adobe Acrobat Reader.app/Contents/Frameworks/RdrCEF Helper (Renderer).app/Contents/MacOS/RdrCEF Helper (Renderer) --type=renderer --log-severity=disable --user-ag
3138 ??         0:00.13 /System/Library/PrivateFrameworks/SystemStatusServer.framework/Support/systemstatusd
```

Figure 4.3: Threads from Google Chrome browser (main process PID 1572) and from Adobe Acrobat Reader (main process PID 3096).

Furthermore, various terms are used (interchangably) to describe processes. A process is called process, job, or task (batch mode). However, there is a slight different:

- **Process**. As defined ealier.
- **Task**. A task could be a process or a thread performing some computation (e.g., printing, downloading, updating, etc). When multiple tasks are executed, that is multitasking — multithreading or multiprocessing. In Java programming environment, when a thread is running, it is called a task.
- **Job**. A complete unit of work. It could be a set of tasks.

## 4.2 Process States

During its lifetime, a process transits from one state to another as illustrated in Figure 4.4:

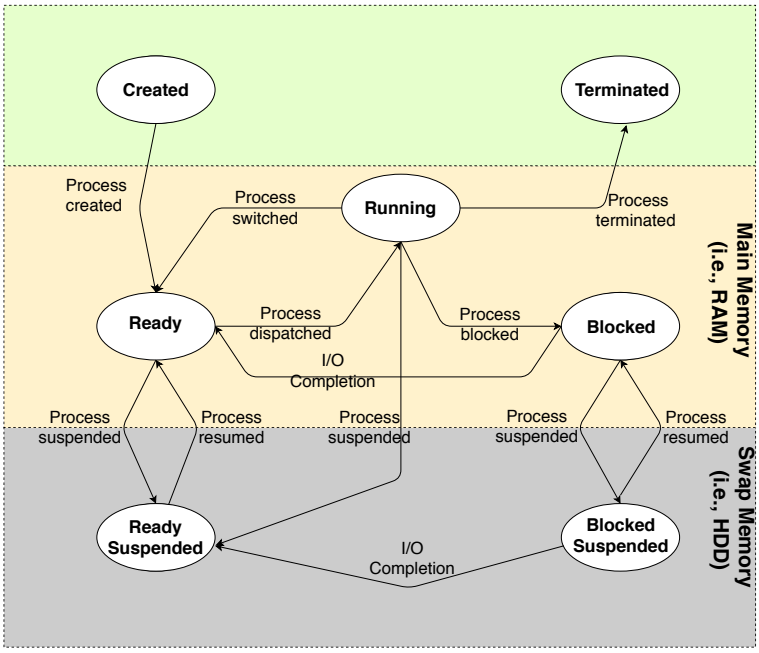



Figure 4.4: Process States During Process Lifetime

The different states are defined as follows:

- **New.** This state indicates that the process has just been created. for example, in C language the primitive `fork()` is used to fork (create) a new process.
- **Ready.** This state indicates that the process is having all needed resources and is waiting (in the ready queue) for the processor (CPU) to be allocated.
- **Running.** Here the process is being executed on the CPU.
- **Blocked.** In this state, the process is waiting (in the blocked process queue) for an event to happen. This could be due to an Input/Output operation requested by the process to be performed by the operating system. It can also be that case where the process is waitin for a wake-up signal from another process.
- **Suspended.** If a process is suspended, this means that it has been swapped out of RAM for some reason. For example, a process can be suspended to free some space for other process, intentionally suspended by a superuser ([8am-6pm]) to save CPU cycles, swapped out of RAM after being longly waiting for an event to happen, or swapped out by the parent process.
- **Terminated.** Here, the process has finished its task and/or has been killed (e.g., in Unix-like operating systems the command kill 2730 is used to terminate the process with PID 2730).

## 4.3 Process Control Block

Every process is represented by a data structure called PCB (Process Control Block) that stores information about the process last state. The PCB contains enough information to block, suspend, resume execution, and keep track of processes.

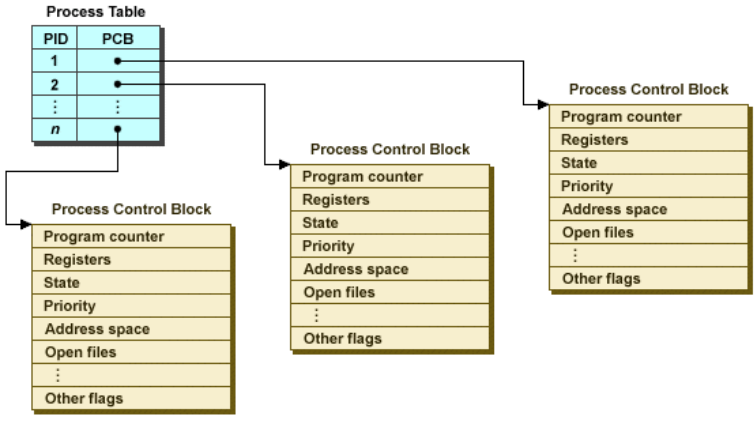



Figure 4.5: Process Control Block

## 4.4 Process Creation and Termination

On Unix-like operating system, processes go through the following phases:

**Creation.** A process can create other processes, called children processes.

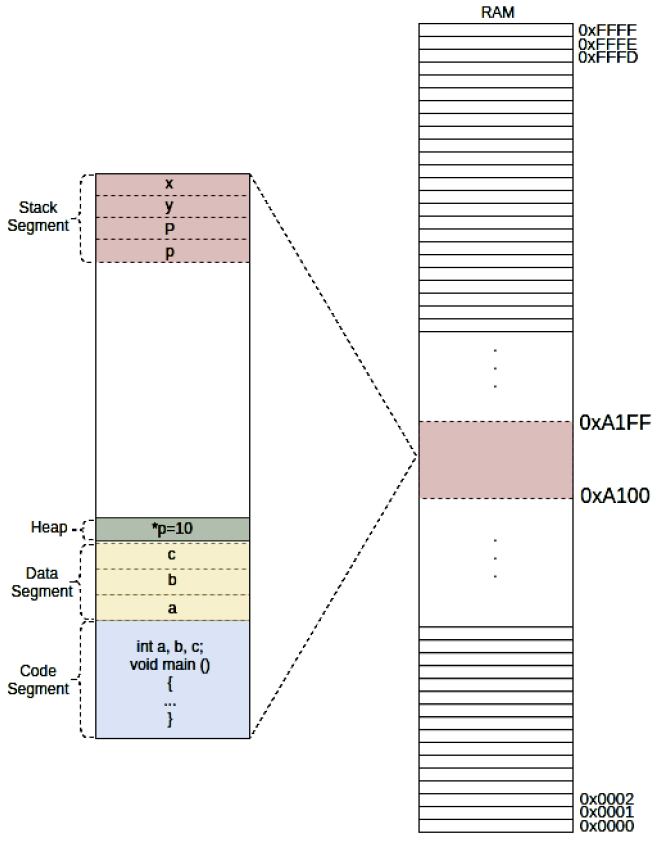
- A process executes the `fork()` system call. Here, the OS duplicates the parent PCB and makes appropriate changes. This includes, assigning the corresponding PID, occupied memory addresses, PC value, etc.
- The parent process can continue its execution or wait (call `wait()`).
- A process can pass data to another process (e.g., using pipes).
- The address space can either be a duplication of the parent space (same data and code), or a newly loaded program (call `execlp()`).
- Resources are inherited (e.g., open files, variables, privileges, ...).

**Termination.** Occurs as follows:

- A process executes the `exit()` system call.
- Running `exit()` returns an integer to the parent.
- A parent process can terminate a child using `kill()` system call.

To better understand how processes are created, consider a program that has the following code. When a program is created, the operating system allocates some memory space to accomodate the memory requirements of the program. The allocated memory space is generally divided into four (04) different partitions (or segments): code, data, stack, and heap. In the code segment will host the code of the program, i.e., the program instructions. The data segment will hold the global variables. The stack segment will hold the local variables and will be used to manage function calls too (i.e., return from a function). The last partition, the heap, will be used for dynamic allocation of memory.

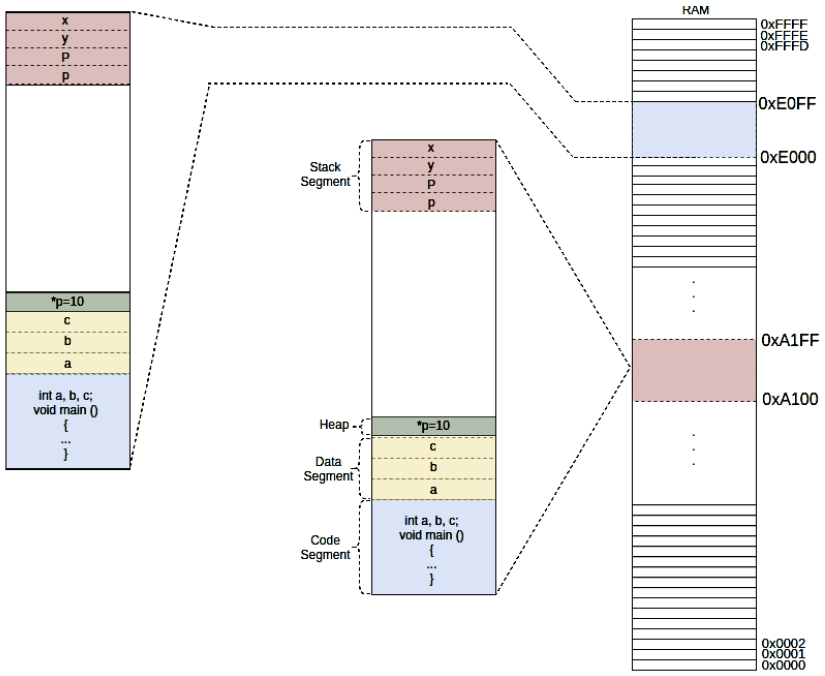
```
int a, b, c;

void main()
{
  int x, y;
  pid_t P;
  int *p;
  p = (int*) malloc(sizeof(int));
  *p = 10;
  P = fork()
}
```




Here, the program declares three global variables of type integer, `a, b`, and `c`. These variables will be stored within the data segment of the program (viz., partition ■ in the graphical illustration below). Then, within the `main` function, the code declares two integers (`x` and `y`), one variable `P` of type `pid_t`, and one pointer `p` to an integer. All these variables are local variables

to the function `main`. They will be stored in the stack partition (viz., partition ■), specifically, in the `main` function frame within the stack. The code will eventually be stored within the code segment (viz., partition ■). When the program runs the 4th instruction of the `main` function, it will request the operating system (so, this is a system call) to dynamically get some memory space to store the value of the pointer, here an integer. That will be stored within the heap segment of the program (viz., partition ■).

When the program execute the 5th instruction of the `main` function, it will request the operating system (syscall, again) to create a new process out of the calling one. This will make the operating system execute the `fork()` syscall, which duplicates the PCB of the calling process and allocates similar memory space to the new born process.




The child process will inherit all the variables as well as their values. However, if the child decides to modify the values of the variables, it will only affect the child variables and not the parent ones. This is totally logical as now the new variables point to new memory partitions (viz., partition ■ in RAM). The parent process variables (the original variables) are located within a different partition (viz., partition ■ in the RAM).

**Example.** Given the following parent process code snippet, how many processes would be created in total? Also, how many times the message "Welcome to ENSIA" gets printed?

```
...
for(unsigned int i=1; i++; i<3)
{
    fork();
    printf("Welcome to ENSIA");
}
...
```

This code consists of a `for` loop that will run for two iterations, when `i=1` and when `i=2`. During the first iteration, the parent process, let us call it $\texttt{P}_0$, will execute the `fork()` statement, which will create a new process $\texttt{P}_1$. Also, the second iteration will result in the creation of a second child $\texttt{P}_2$. Both children ($\texttt{P}_1$ and $\texttt{P}_2$) along with the parent process will display the message "`Welcome to ENSIA`". As the parent makes two iterations, it will display the message twice.

Now for the child processes, when the first child, $\texttt{P}_1$, was created, the program counter register was already pointing to the next instruction in the code, which is the `printf()` statement. So, when the child is created, it will start executing from that printing statement. The first child will print the message for `i=1` and for `i=2`, so two times. The second child ($\texttt{P}_2$), however, starts with `i=2`. So, it will just print the message and terminate. This makes a total of 05 printed message, thus far.

Going back to the first child, $\texttt{P}_1$. When `i=1`, just after it got created, it displayed the message and then moved to the second iteration (i.e., `i=2`). At this second iteration, $\texttt{P}_1$ creates a child process, $\texttt{P}_{11}$, displays the message, and then termiantes. Now, the remaining process, $\texttt{P}_{11}$, displays the message and terminates. This gives us a total of four (04) processes (including the parent process) and six (06) displayed messages.

As a general rule, when a process calls in `fork()` within a loop that is supposed to iterate for `n` times. The total number of processes, including the parent, would be $2^{\texttt{n}}$. In our example, the loop iterated twice, so the total number of processes was $2^2 = 4$.

## 4.5 Concurrent Processes

Concurrent processes are essential for modern computing systems as they allow multiple tasks to be executed simultaneously, improving system responsiveness and efficiency. By dividing complex tasks into smaller, independent processes, concurrent execution can significantly reduce the overall time required to complete these tasks.

This is particularly beneficial for applications that involve I/O operations, such as reading from a disk or network, as the CPU can continue processing other tasks while waiting for I/O operations to complete. Additionally, concurrent processes can enhance resource utilization by allowing multiple processes to share system resources more effectively.

Processes are said to be concurrent when: (1) multiple programs are loaded into the central memory, (2) their execution is happening in parallel, i.e., running on different physical CPUs or CPU cores, or in fake-parallel, i.e., running on the same CPU, but in a time-sharing fashion. Hence, their time durations or executions are either **overlapping** (real parallelism) or **interleaving** (fake parallelism).

We can use PPG (Process Precedence Graph) to graphically visualize execution order of concurrent processes. A PPG is a directed graph where the nodes (circles) represent processes and the edges represent the precedence constraints. Hence, a node `P` that is connected to another node `P'` with an edge going from `P` to `P'` (`P` →`P'`), indicates that the process `P` should execute and terminate first before `P'` can start executing.

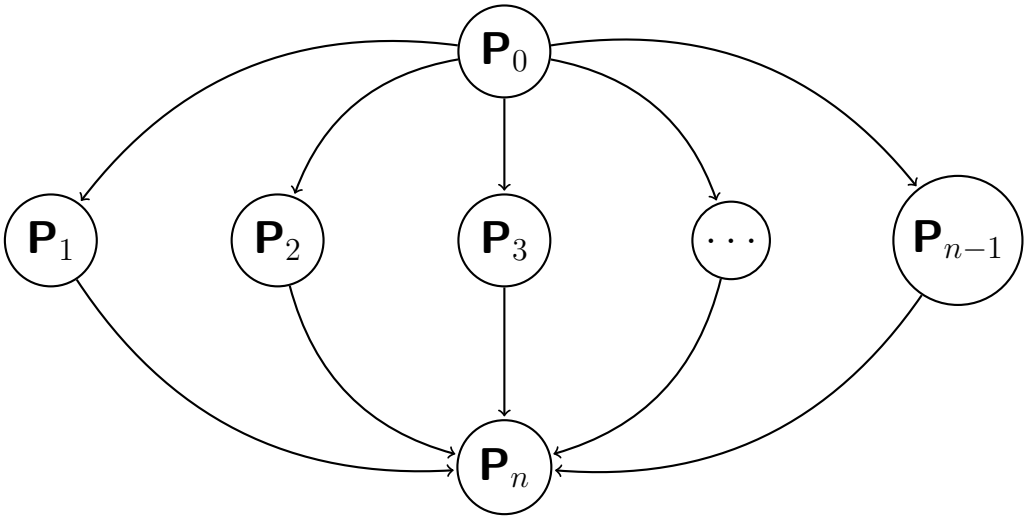
In a PPG, we organize nodes in levels. The top node(s) represent the first processes that should execute without any constraint. The next level of processes represent a group of processes that can execute after the termination of the ones on the first level. Thus, executions can either be sequential (serial — cascaded levels) or in parallel (processes within the same level).

Furthermore, The PPG can be represented using a textual construct called, `ParBegin-ParEnd` (for parallel begin and end), or `CoBegin-CoEnd` (for concurrent begin and end). In this structure, sequential executions are framed within a `Begin-End` construct, whereas the parallel executions are framed within a `ParBegin-ParEnd` construct.

**Example.** Consider the following code:

```
Begin
    P0
    ParBegin
        P1
        P2
        ...
        Pn−1
    ParEnd
    Pn
End
```

This code consits of three sequential blocks. The first block consists of one single process $\texttt{P}_0$. The second block consists of a parallel execution of $n-1$ processes ($\texttt{P}_1$, $\texttt{P}_2$, ..., $\texttt{P}_{n-1}$). The last block is a single process $\texttt{P}_n$. The code can be translated into the following PPG:




Processes can run in parallel (or fake-concurrency) and cooperate with each other for several reasons:

- **Information Sharing.** Processes can collaborate on the same file (or database) to share information. The simplest example you could think of is one process computing a mathematical expression and writing the result on a file, and another process that reads the result from the file and uses it to perform further calculations.

- **Computational Speedup.** Parallel execution of different parts of a program speeds up the total execution time and decreses the response time. Program designers should consider designing their program in way it can run using different execution flows (i.e., multithreading).

- **Modularity.** This is one of the most important concept in software engineering. Designing software in a modular fashion allows better performance, reliability, maintainability, and security. Modern operating systems are built in a modular fashion. This modularity requires concurrent execution of processes and their collaboration.

- **Convenience.** An individual users may run different processes.

Processes can communicate via two fundamental mechanisms: (1) Shared memory, this could be a variable, data structure, or a file. (2) Message passing, which requires a physical/logical communication link (local: pipes, remote: RPC and sockets).

## 4.6 Interrupts

Interrupts are essential mechanisms in operating systems that allow hardware devices to signal the CPU of events requiring immediate attention. When an interrupt occurs, the CPU pauses its current task, saves its state, and jumps to a pre-determined interrupt handler routine to address the event. This enables the operating system to respond promptly to external stimuli, such as keystrokes, mouse movements, or network data, ensuring a responsive and interactive user experience. Interrupts also play a crucial role in process managment where it allows user processes to ask the operating system, through a supervisor call (or system call) to access the hardware and perform I/O operations, e.g., display a message on the screen, read content from a file, write content to a file, send a signal to another process, etc.

First of all, let us consider the following definition for interrupts:

**Interrupt**

Interrupt is an event that alters the sequence of instructions executed by the CPU. A dedicated program called ISR (Interrupt Service Routine) or interrupt handler is executed as a consequence.

Each interrupt has it own **interrupt handler**, a.k.a., ISR (Interrupt Service Routine) that services the cause of the interrupt., An ISR is a low-level program (written in assembly language) that is executed in kernel mode to service the cause of the interrupt. They are located by an IVT (Interrupt Vector Table)[1] stored in the memory. E.g., in x86 architectures, it is generally stored at address 0x0000-0x03FF. The number of interrupt types is limited by the architecture.

### 4.6.1 Type of Interrupts

There exist two types of interrupts, software and hardware:

**Software-based Interrupts.** When the system is interrupted following the execution of an instruction in a program code, the interrupt is said to be software-based. There are two types:

1. **Exceptions.** An exception happens when the current instruction perform an illegal action such as: division by 0, arithmetic overflow, buffer overflow, page fault, access protected memory space, or attempt to execute a privileged instruction (process is terminated).

2. **System calls.** A.k.a., traps. User program are executed in user mode. In this mode, programs do not have direct access to peripheral (i.e., during an I/O operations). They have to issue a system call, or supervisor call, which causes an interrupt. At the assembly level, the system call will be compiled into the `INT` instruction.

**Hardware-based Interrupts**: These asynchronous interrupts are trigged by a hardware unit such as the timer/clock, keyboard, mouse, power button, ..., device controllers or DMA, to get the attention from the CPU. They can be **maskable** (MI) or **non-maskable** (NMI).

[1] Also check IDT (Interrupt Descriptor Table) used on modern x86 architectures.

- **Maskable.** These are interrupts that the CPU can choose to ignore based on its current state and priority. They are typically used for non-critical events that can be handled later without causing significant issues. E.g., Keyboard input, network packets arriving, etc.
- **Non-Maskable.** These are interrupts that the CPU cannot ignore. They demand immediate attention, often for critical events that could jeopardize system stability or integrity. E.g., hardware errors, system watchdog timer expiring, etc..

In addition to the above classifications, other types of interrupts exist:

- **Inter-Processor Interrupt.** Used between processors to communicate in a multiprocessor system, e.g., one processor core requests another processor core to stop when the system is shutting down by the first processor (`$sutdown now`).
- **Spurious Interrupt.** A.k.a., phantom or ghost interrupt, is an unwanted hardware interrupt, e.g., when an interrupt that has been signaled to the processor and it is no longer required (or interrupt source disappeared), or buggy hardware..
- **Raster interrupt.** This is a type of interrupts used in old monitors. It was employed to synchronize the display with the CPU's processing speed. In these monitors, the electron beam that traced the screen's pixels moved horizontally across the screen, line by line. A raster interrupt was generated at the end of each line, signaling the CPU that it could update the video memory with the next line's data.

We call an **interrupt storm** the event where the operating system receives a large number of interrupts (from hardware) that consumes the majority of the processor's time spurious signals, interrupt handler execution took too long, ..., faulty drivers. The system becomes non-responsive.

### 4.6.2 System Calls

System calls Constitutes an interface between the user (programs) and the services that are made available by an operating system. They constitute a programmatic way in which a computer program requests a service from

the operating system (e.g., reading from a file, writing to a file, creating a new process, allocating memory, free memory, sending packets through the network interface, sending a signal to another process, etc). Each operating system has its own system calls (e.g., GNU/Linux has 300, FreeBSD has 500, and Windows 7 has 700).

System calls point to specific programs called routines that are written in C, C++, or assembly language. The operating system provides an API (Application Programming Interface) as an interface between user programs and system calls. For example, to write a message on the monitor, the user program, uses a function in a high-level programming language (`printf( )` in C or `cout` in C++). That function will be mapped through a library to the `wirte()` system call to send data to the standard output (monitor).

System calls can be grouped into six categories:

1. **Process Control.** These are system calls for managing processes. They include calls to end, abort, load, execute, create, and terminate processes. They also include calls to get/set process attributes, wait for a time, wait for an event, signal an event, allocate and free memory. E.g., `exec()` is used to execute an arbitrary process, `fork()` to create a new process, `wait()` to wait for a process, `exit()` to terminate a process, `getuid()` to get the user ID of the running process, `getgid()` to get the group ID of the process, `getpid()` to get the PID of the process, `getppid()` to get the parent PID of the process, etc.

2. **File Management.** These are system calls for managing files. They includes calls to create, delete, open, and close files, read, write on files, and get/set files attributes. E.g., `create(), open(), close(), read(), write(), chmod(), chown`, and `umask` are common system call for file management.

3. **Device Manipulation.** Operating systems provide a list of system calls to acquire, release, read, and write on devices (e.g., network card). For example, `request(), release(), read(), write(), open()`, and `close()` are device manipulation system calls.

4. **Information Maintenance.** There are system calls to get/set time or data, get/set system data, get/set resource attributes. E.g., `time(), date()`, and `alarm()` are used for information maintenance.

5. **Communication.** These are system calls for inter-process communicaiton. They include calls to create, delete a network connection, send and receive messages, and transfer status information. They include syscalls such as `pipe(), mmap(), shm_open()`, and `gethostid()`.

6. **Protection.** For security and access control, the operating system provides a ser of syscalls to set ownership and access rights on resources. E.g., `chmod(), umask()`, `chown()`.

### 4.6.3 Hardware Interrupt and Polling

You may have asked yourself how are hardware events (keyboard presses, incoming network packets, mouse movement, ...) escalated to running programs? Well, there exist two ways:

**Polling.** When OS periodically queries each device to see if new information is available. This method is simple but has high latency and results in wasting many CPU cycles.

**Interrupts.** Here a device sends a signal to the **interrupt controller**[2] to request attention, CPU preempts running process to handle device request.

The period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt is called the interrupt latency. After each (user mode) instruction is executed, the CPU checks whether the interrupt controller has any interrupt pending. If an interrupt is there, the current process is "interrupted", and the appropriate handler is executed.

Hence, after each instruction execution cycle, the processor check a flag "interrupt flag" in the FLAGS register to determine whether there is (or there are) any interrupt(s) pending. If the flag is set to 1, then at least one hardware interrupt is pending, and has to be serviced. To that end, the currently running process is **interrupted** and the corresponding interrupt service routine is invoked (viz., Figure 4.6).

[2]PIC & APIC, generally, integrated circuits within the southbridge.

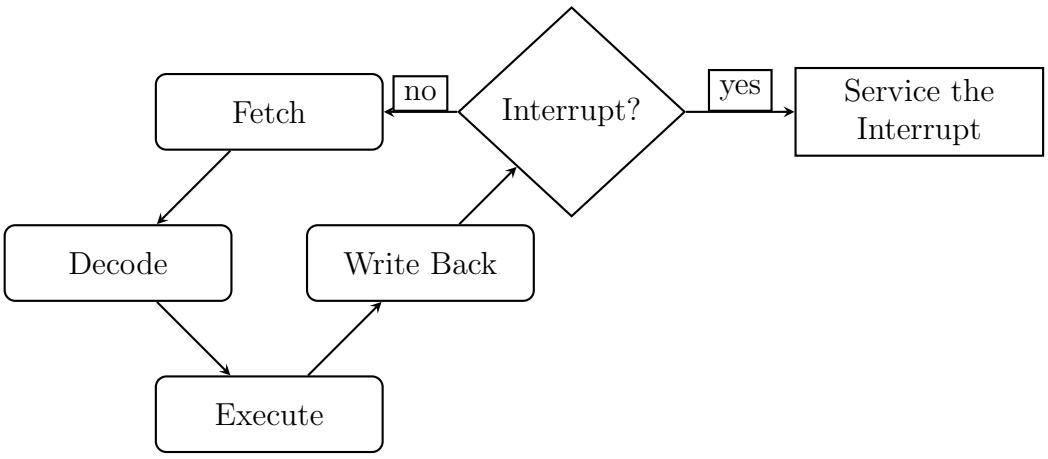



Figure 4.6: Instruction Execution and Interrupts Cycle

## 4.7 Context-Switching

During a process turnaround time (lifetime), the process may obtain the CPU for a while to execute some of its code, lose it for a while, then get it back to continue its execution till it terminates. Each time the process is interrupted to give the CPU to another process to execute, and then resumed to continue executing. During this computing phenomenon, a certain number of information, called **context**, must be saved (state save) to the process PCB when the process is interrupted and then loaded again (state restore) from the PCB when the process is about to resume.

Depending on the implementation of the operating system, the context may contain various types of information. In general, the context of a process should contain the following information:

- The content of all CPU registers including the PC and the flags register.
- The memory-management information, i.e., the address space occupied by the process.
- The acquired resources, e.g., files, semaphores, network connections, and I/O devices (printer, scanner, etc).

Basically, context-switching is the mechanism that allows the kernel (the operating system) in one atomic operation to: (1) Save the context of current

process into its PCB (so that the system does not lose track of the process), and (2) load the context of the newly scheduled process from its PCB so that it can start running (or resume if it was previously interrupted).

As you may have figured it out, context-switching is an important mechanism for time-sharing systems. It allows the following:

1. It increases the CPU throughput. I.e., the number of processes executed by time unit.

2. It increases the CPU response time. As you context-switch, you allow processes to execute and respond quickly.

3. It prevents certain processes from hogging on the CPU infinitely. For example, this will prevent a `while(true);`—like program to occupy the CPU forever and deny other processes from executing.

4. It allows high-priority processes to be executed. Long low-priority processes will not hog on the CPU and prevent high-priority processes which just arrvied to the system from executing.

5. It reduces process turn-around-time. As processes get the chance to execute, they will spend less time waiting in the system.

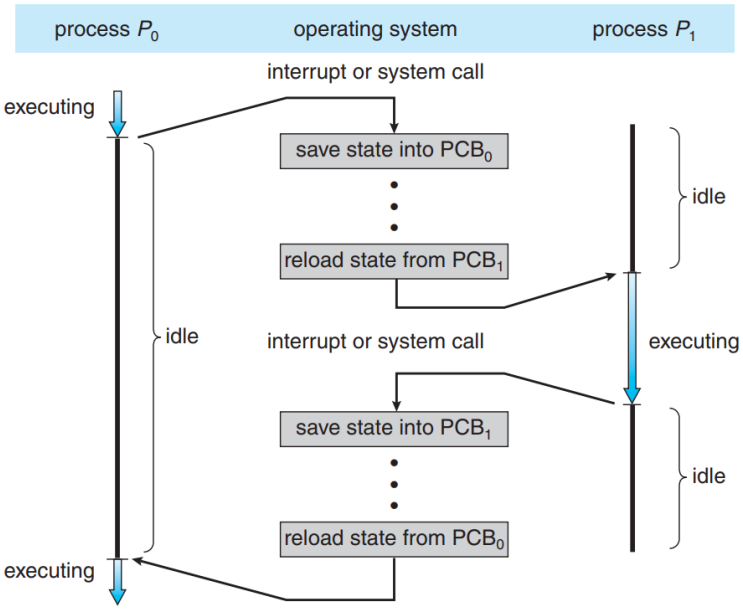
However, context-switching has also a cost. Actually, it depends on the size of the context. The bigger is the context, the longer will the context-switch operation last. The system must not be interrupted for a long time.

The PCB serves as the repository for any information that may vary from process to process (Src: Textbook — Figure 3.9 — Page 109):

Last but not least, interrupt handler should not be considerd as processes, or special processes. They do not have PIDs and they are not handled as processes.

### 4.7.1 Interrupts with Context-Switching

Looking at the various existing computer architectures, you would find out that the steps for context-switching may vary from one architectur to another — old architectures to modern architectures. I.e, there is no standard steps that all computers apply. In this course (and course reader), we present the commonly used steps in the literature (for the sake of education).




Assuming that process a P1 (e.g., notepad) is executing and process P2 (e.g., Abobe reader) is the next process in the **ready queue** (this is a queue of PCBs of processes that are ready to be executed — they are just waiting for the CPU to be allocated to them). Then, if an interrupt occurs::

1. CPU pushes the current PC onto the stack (of the interrupted process, here P1) and updates the PC-register to contain the address (obtained from the Interrupt Controller) of the 1st instruction in the corresponding interrupt handler (interrupt service routine).

2. The interrupt handler (a.k.a., ISR) starts execution:

   - It pushes all remaining registers onto the stack then performs its task (e.g., display error message, update clock, move mouse, etc).
   - It invokes an operating system module, called the scheduler, to cause context-switching if required. If that is the case (i.e., a context-switching is required), the scheduler:
     - Copies all registers values from the stack of the process (P1) into PCB1 (process control block of process P1).
     - Moves PCB1 to end of ready/blocked queue, and PCB2 into the head. PCB1 is moved to the ready queue if the scheduler

determined that it was the time to interrupt P1 and deallocate the CPU and select another (P2) to be allocated the CPU. If PCB1 is moved to the blocked queue, then that should be a consequence of system call made by process P1 (e.g., P1 requested an I/O operation). That results in an interrupt and the corresponding routine is invoked.

  - Copies registers values from PCB2 into CPU-registers, except the PC values which get copied to the top of the new stack (P2's stack) so that P2 can execute.

- The interrupt routine executes `RTI` (ReTurn from Interrupt) to indicate its termination. This causes the CPU to automatically pop the PC value from the new stack to CPU's PC register.

3. The CPU then, starts/resumes the execution of process P2.

### 4.7.2 Interrupts without Context-Switching

Some of the interrupts are short and quick. They do not cause a context-switching. That means, the interrupted process continues executing after the interrupt service routine is done servicing the interrupt.

For example, assuming that process P1 is executing and a keyboard/-mouse interrupt occurs:

1. CPU (hardware) pushes the current PC onto the stack and updates the PC-register to contain the address of the 1st instruction in the corresponding interrupt handler.

2. The interrupt handler starts execution by pushing all other registers onto the stack.

3. Performs its specific task that modifies the CPU registers content.

4. Pops back the registers values from the stack into the CPU.

5. The handler executes RTI (ReTurn from Interrupt) which causes the CPU (hardware) to pop PC value from stack to CPU's PC register.

6. The CPU then, resumes executing process P1.

### 4.7.3 Interrupts in General

Generally, right after an interrupt (hardware or software):

1. The CPU pushes the PC value into the stack of the interrupted process, and which is pointed by SP (stack pointer) register.
2. The CPU loads the PC register with a new value (address of the handler's first instruction from the IVT table).
3. The handler saves the remaining registers onto the stack.
4. The handler executes its specific task.
5. The handler (through the scheduler) determines whether a context-switching is needed or not: If yes, a context switching is performed. The previously pushed registers are saved into the PCB (of the interrupted process) and the CPU is loaded with a new content, except the PC, from a new PCB. Else, if no context-switching is required, goto Step 6.
6. The handler Executes RTI.
7. The CPU pops back the PC value from the stack pointed by SP.

Furthermore, it is important to note that interrupts may interrupt other interrupts. In fact, during the execution of an interrupt handler, other interrupts can occur. Interrupts are executed in a **nested order**, i.e., if an interrupt of a higher priority occurs while another interrupt handler of a lower priority is executing, the current (low-priority) interrupt handler is interrupted. When an interrupt handler is interrupted, its PC value is pushed by the CPU onto the stack dedicated for interrupts (located in kernel space) — pointed by the interrupt Stack Pointer (ISP). The new handler saves the other register into kernel interrupt stack. Once the new handler terminates, it pops the registers of the old handler and executes RTI, which informs the CPU to pop the PC value from interrupt stack so that the old handler can resume execution.

## 4.8 Threads

Threads constitute a fundamental concept in software development, in general, and operating systems, in particular. They offer improved responsive-

ness, performance, and resource utilization by allowing processes to execute multiple tasks concurrently.

### 4.8.1 Lightweight Processes

Threads are considered **lightweight processes**. The feature of lightweight result from the fact that creating threads does not need to allocate additional memory space as we generally would do with ordinary processes. Actually, when creating threads: The process that creates threads shares its code, data, heap, resources (e.g., open files), user/group ID, ... etc, with the created threads. Hence, we do not need to allocate additional memory space to store the code and data of the new thread. Also, threads only require a dedicated stack and will have their proper values when they use the CPU registers. Morevoer, thread context-switching is very fast compared to process context-switching. However, context-switching a thread is not always faster than context-switching a process. If we are context-switching two threads that were created out of the same process, then the operating will be faster. Nevertheless, if you are switchig two threads that were created out of two different processes, then the switching operationg will take the same time as switching two different processes.

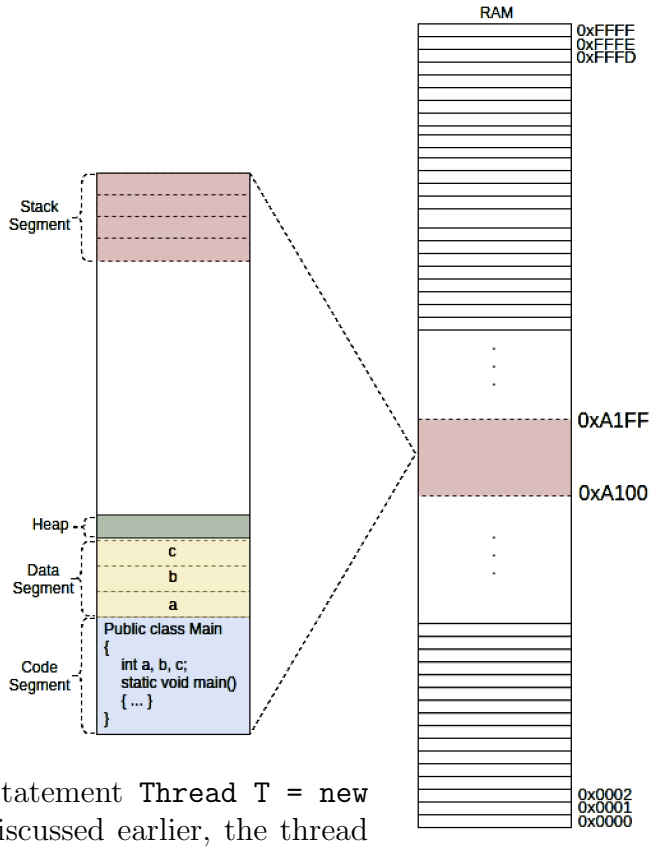
Similar to processes, threads are identified by a TID (Thread IDentifer). Also, each thread is associated to a data structure called TCB (Thread Control Block) which contains the thread's Identifier, thread's state, priority, thread's registers values, and a pointer to the original process PCB. On a GNU/Linux, the command `$top -H -p 4234` would display the threads that are related to the process whose PID is 4234.

To better understand the different between the creation of a process and the creation of a thread, let us consider an example.

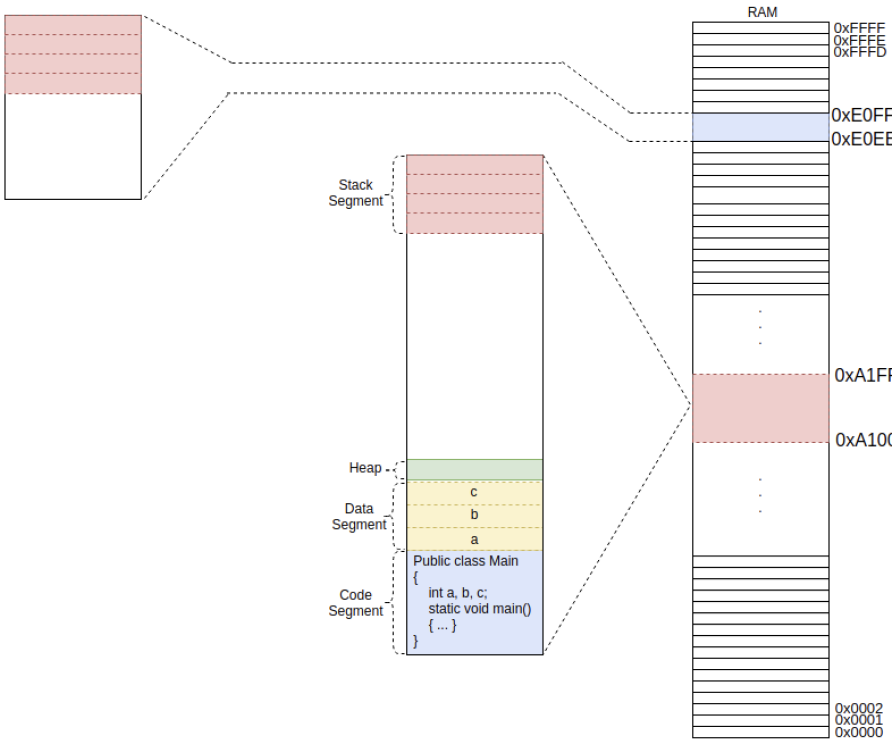
Consider a program that has the following code (cf., Section 4.4). We have seen in Section 4.4, that when a program is created, the operating system allocates some memory space to accomodate the memory requirements of the program. The allocated memory space is generally divided into four (04) different partitions (or segments): code, data, stack, and heap. In the code segment will host the code of the program, i.e., the program instructions. The data segment will hold the global variables. The stack segment will hold the local variables and will be used to manage function calls too (i.e., return from a function). The last partition, the heap, will be used for dynamic allocation of memory (e.g., when `malloc()` is used).

We have also seen (in Section 4.4) that when the above program executes the `fork()` syscall, the operating system duplicates the PCB of the calling process and allocates similar memory space to the new born process as illustrated. The child process will inherit all the variables as well as their values. However, if the child decides to modify the values of the variables, it will only affect the child variables and not the parent ones. This is totally logical as now the new variables point to new memory partitions (viz., partition ■ in RAM). The parent process variables (the original variables) are located within a different partition (viz., partition ■ in the RAM).
Now, let us consider the following Java program:

```
public class Main
{
  int a, b, c;
  static void main
  {
    Thread T = new Thread(new Runnable()
    {
      @override
      public void run
      {...}
    });
    T.start();
  }
}
```




Within the `main` function, the instruction statement `Thread T = new Thread()` allows the creation of a thread. As discussed earlier, the thread will be allocated a dedicated stack and will inherit all remaining partitions. The thread will have complete access to the gloab variables of the main process. Its code is the same as the main process's code, but will start executing within the `run()` function when the parent executes `T.start()`. The creation of the thread results in the following illustration:




The main process remain as is, whereas the created thread will have access to the main process code, data, and heap, but will use its own stack partition.

To grasp the notion of lightweight processes, you could imagine how much memory is used when you create 10 child processes from a 10MB program versus when you create 10 thread out of a 10MB program.

It is important to note that older process model used to be composed of only one thread (single execution flow). We talked about **single-threaded processes**. Now, modern processes are **multi-threaded processes**, i.e., the process can perform more than one task at a time (multiple execution flows) — viz., Figure 4.7. However, this poses new challenges for system designers and application programmers. In fact, they have to consider designing their software and system in a way that they can make use of the entire multi-processing resources. I mean, if you know that your software will run on multiprocessor system, then it will be very unpleasant and unprofessional to build your software as a single-threaded piece of code. You would rather employ the modular approach and design your code to be multithreaded.

Most modern OS's kernel are now multithreaded, e.g., GNU/Linux use a

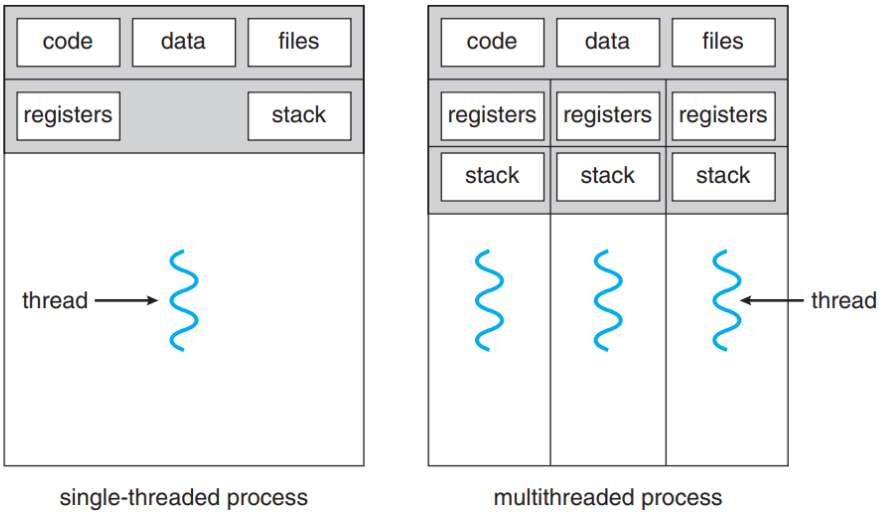



Figure 4.7: Single-threaded versus Multi-threaded Processes.

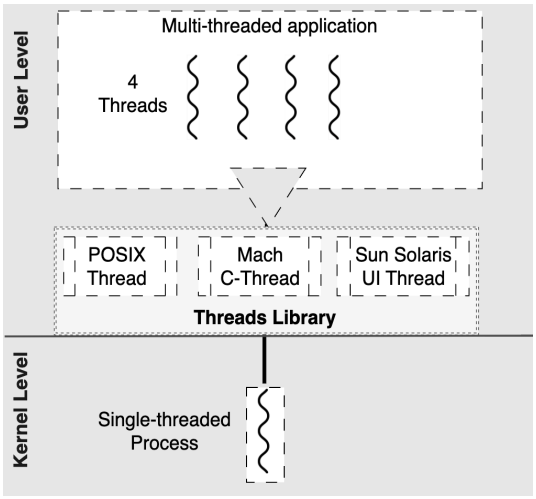
kernel thread for managing the amount of free memory in the system.

### 4.8.2 Advantages of Threads

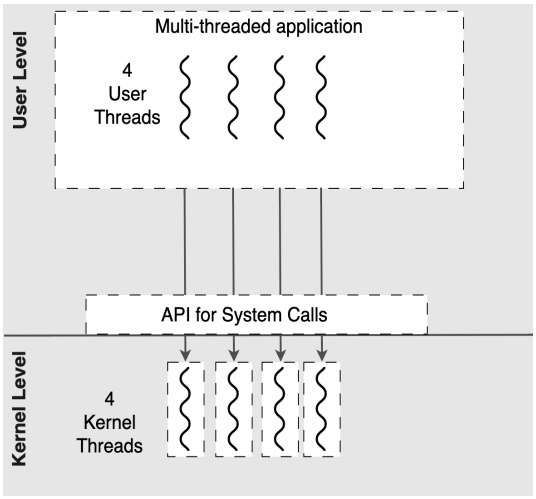
In general, threads have the follwoing advantages:

- Responsiveness: If a thread is blocked, the other one can continue running.
- Resource sharing: No need for interprocess communication. Threads from a same process would share code and data.
- Economy: We don't need to allocate a separate memory space.
- Scalability: Utilization of multiprocessors/multi-cores, whereas a single-threaded process uses only one core.

Executing a multi-threaded application on a single-core CPU system is meaningless (fake concurrency $\neq$ (real concurrency = parallelism)). Modern CPUs can run more than one thread per CPU core. Those are called hyperthreaded processors, e.g., Intel i5-2410M, has 4 cores, runs 2 threads per core.




User-level Threads




Kernel-level threads

### 4.8.3 Types of Threads

As part of being lightweight processes, threads are often implemented at two levels, user and kernel:

**User-level Threads.** In this type of threads, the OS is not aware of the presence of multiple threads in one process (Many-to-One threading model). A threading library is used to perform threads context-switching. The main process memory will be used for threads stack allocation. Also, system calls from one thread would block the main process (not efficient). The OS schedules the process as a single-threaded process entity. It applies iInternal thread scheduling (it is lighter, flexible, and portable).

**Kernel-level Threads.** In this type of threads, the OS is aware of the concept of threads (i.e., kernel is multi-threaded) and offers system calls to create and manages threads (One-to-One threading model). Here, system calls are used to create physical threads. Each user thread is mapped to one kernel thread.. Also, system calls from one thread do not block the other running threads. With kernel-level thread we can use multiple CPU cores at a time. The following libraries and framework support kernek-level threads: Pthreads API, Win32 API, and Grand Central Dispatch framework.

## 4.9 Direct Memory Access (DMA)

Direct memory access (DMA) is the process of transferring data without the involvement of the processor itself. It is often used for transferring data from the memory to I/O devices or from I/O devices to the memory. This is performed through the use of a **special-purpose processor** known as **DMA controller**, which notifies the CPU that it is ready for data transfer. When the CPU is notified by a DMA, it relinquishes control of its external memory bus and grants the control of the bus to the DMA controller. The DMA controller then transfers the specified amount of data and signals the processor upon completion of the transfer (i.e., DMA I/O completion).

To better understand the importance of having DMA in a computer, imagine you would like to download and save this course reader document on your hard-disk drive for future reference. By doing so, do you think it is necessary to involve the CPU for storing this pdf file in your hard-disk drive, or not? You would say that the data (pdf file) is received by your network card and then transfered to the desired location on your hard-disk drive (e.g., Donwloads folder). Invoking the CPU to perform this transfer would certainly be an overkill. CPU's cycles would be wasted.

Now you could imagine, when multiple devices (or peripherals) are all attempting to access some memory locations. This is the reason for the requirement for a DMA controller (viz., Figure 4.8).

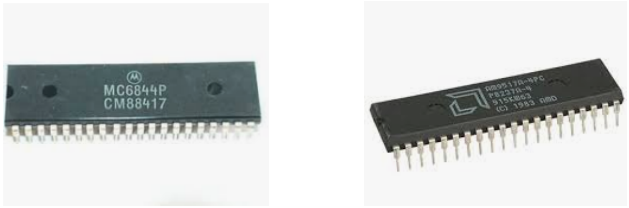



Figure 4.8: Motorola MC6844P DMA controller (left) and Intel 8237A-4 DMA controller (right)).

### 4.9.1 Device Controller and Device Driver

Each I/O device has a dedicated device controller, which is a hardware component that interfaces between the device itself and the computer (CPU and memory). We have seen that in a computer system, I/O devices are

connected to the motherboard via a port (e.g., USB, RJ-45, RJ-11, RS-232, DisplayPort, DB25, etc), whereas the device controller is connected via a slot (connecting without soldering), e.g., eSATA, IDE, FDD, etc. Sometimes, one device controller handles multiple I/O devices.

Figure 4.9: Device Controller of a Hard Disk Drive (HDD)

A device controller receives data from a connected device and stores the data temporarily in some special purpose registers, known as **buffer**. E.g., Your Ethernet network card would buffer incoming network traffic, i.e., packets received from the Internet or local network.

Furthermore, each device has a dedicated **driver**, which is a type of system software that interfaces between the device controller and the operating system. The driver contains interrupt handlers that service the interrupts that are generated by that device controller. Hence, when the driver for an I/O device is installed, interrupt handlers that service the interrupts of that device will eventually get registrated.

### 4.9.2 DMA and Interrupt Mechanism

Consider a problem where a large amount of data needs to be transfered from central memory (i.e., RAM) to an I/O device or the reverse. One possible solution, but not efficient, is the use of polling. We have seen that in polling, the CPU periodically sends polls to I/O devices asking for any event. I/O device controllers respond with a positive acknowledgment if there are any data to be transfered. If that is the case, the I/O device controller starts sending the data to CPU, which then sends them to memory.

The second solution is the use of interrupts. When there is data to be transfered, I/O device controllers request the CPU to perform an I/O oper-

ation with RAM to store the data. The CPU relases control ove the system bus and grant access to the device controllers. The CPU will have to wait for the device to read/write data. Of course, there is a speed mismatch and lot of CPU cycles are wasted. Just imagine the speed of the CPU versus the speed of a mechanical hard-disk drive. This is unefficient as the CPU is not meant to wait but to constantly execute instructions.

A better option would be to use DMA. DMA technology allows I/O devices to directly access the memory without involving, or disturbing, the CPU (in particular, I/O devices that do large transfers, e.g., HDD, Graphical card, Network card, Sound card, etc).

### 4.9.3 DMA Operational Modes

As there is only one system bus (address, control, and data bus), technically, the DMA controller can operate in one of the following modes:

1. **Burst mode.** The CPU grants DMA controller access over the system bus. DMA occupies the system bus for transferring a complete block of data (256 to 512 bytes) before releasing the bus back to the CPU.

    - CPU can only access its local cache (L1, L2, and L3).
    - CPU may remain Idle waiting for the bus to be released.

2. **Cycle-stealing mode.** The CPU grants DMA controller access over the system bus. The bus is relinquished for a few clock cycles so that the CPU can perform other tasks (use cache) while the transfer of few bytes takes place. (sometime — Single-cycle mode).

    - CPU can only access its local cache (L1, L2, and L3).
    - CPU may not remain Idle for longer time.
    - DMA controller keeps asking for the bus for each transfer.

### 4.9.4 DMA and Cache Coherency

Another important aspect in DMA technology is **cache coherency.** This occurs when a DMA operation involves accessing a memory location in RAM which values have been updated by the CPU but are still stored on the CPU's

cache (i.e., not yet written to memory). The DMA controller would then transfer data that is not up-to-date. To solve this problem there are two possible solutions:

- **Fully cache coherent system.** Here the DMA controller can communicate with the cache to invalidate values in case of a Read operation. If it is a Write operation, the cache controller flushes the cache (i.e., update memory location) ensuring that the memory location contains the most up to date value (some overhead but guarantees the coherency).
- **Not Fully cache coherent system.** Cache coherency is handeled by the operating system. The operating system will be required to decide if the cache should be flushed prior to a DMA operation, or invalidated afterwards.

**Example.** Consider a DMA transfer from a device to the memory. First of all, device driver is instructed (after the execution of a service routine, following a system call) to transfer data from device to memory, E.g., transfer 256 bytes from disk at $@_d$ to RAM at $@_m$.

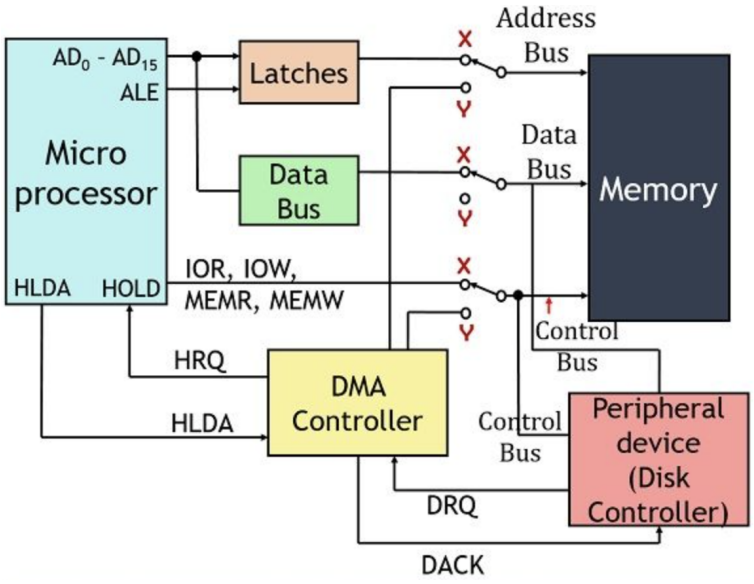



Figure 4.10: Functionning of a DMA Controller

The device driver (through ISR) instructs the device controller to transfer 256 bytes to central memory. The device controller places a request for

transfer to the DMA controller (i.e., DRQ — DMA Request). DMA controller requests the system bus (i.e., HRQ — Hold Request). The CPU then delegates the buses to the DMA controller (i.e., HLDA — Hold request Acknowledgement). The DMA controller seizes the system bus (CPU is momentarily prevented from accessing the bus — hence the RAM). At this time, the CPU can or cannot execute instructions. The DMA controller sets up the memory controller and informs the device controller to start the data transfer (DACK — DMA Acknowldegement). Upon reception, the device controller starts sending each byte of data to the memory. For each transferred byte, the DMA controller byte counter is decreased and its internal address register increased. Once done (counter=0), the DMA interrupts the CPU to signal a DMA transfer completion so that the CPU resumes. This process is illustrated in Figure 4.10.

# Chapter 5

# Process Synchronization

Imagine that you were able to create 05 copies of your EDAHABIA card. Then, assuming that you have 10000 DA in your account, what could happen if you and four of your trusted friends walk into the post office and use those generated copies of the card and try to withdraw the amount of 10000 DA at extactly the same time? Two possibilities, you will only be able to withdraw 10000 DA, and the four other attempts will fail, or you will manage to withdraw 50000 DA. The first possibility is consistent and legal, whereas the second one is incoherent and illegal. You would not like to implement a system that contains such a flaw and that can be abused by some people to steal money. Synchronization is the solution to such problem.

Process synchronization is a critical aspect of multi-threaded software, in general, and operating systems, in particular. It ensures that multiple processes or threads can coordinate and share resources consistently and effectively. It prevents race conditions (e.g., withdrawing 50000DA from an account that contains 10000 DA), deadlocks (e.g., you try to withdraw and deposit money at the same time, and both operations fail), and other concurrency-related issues. Synchronization mechanisms, such as semaphores, mutexes, and monitors, help regulate access to shared resources, ensuring that processes execute in a predictable and orderly manner. By effectively managing process synchronization, operating systems can maintain system stability, prevent data corruption, and enhance overall system performance.

In this chapter, we delve into process synchronization. You will learn how operating systems use different types of synchronization mechanisms to allow consistent cooperation between concurrent processes.

## 5.1 Synchronization

We say that a system $\mathtt{S}_1$ is synchronized with another system $\mathtt{S}_2$, if the speed at which system $\mathtt{S}_1$ operates **depends** on the speed at which $\mathtt{S}_2$ operates. If both systems are mutually synchronized, their operational speeds depend on each others speed. In general, a system that operates based on the speed of another system is refered to as a **synchronous** system. For example, the processor (CPU) contains an electronic component (a crystal oscillator) that periodically generates a precise electrical signal, called pulse (a clock cycle). Within a clock cycle, the processor performs one or more operations (depending on the processor's IPC — instruction per clock cycle). The speed of the processor depends on the speed at which the clock generates its pulses, i.e., the frequency of the clock. In the world of software and process management, a set of processes are said to be **synchronized** if the speed at which they progress their executions dependes on the execution speed and **progress** of each other. If a system evolves independently, then it is **asynchronous**.

Another real life example is the traffic light system. Traffic light system is a synchronization mechanism that allows to enforce order at road and street intersections. When people respect traffic lights, the risk of accident will be considerably decreased. Also, faireness between traffic users is established.

If there is no traffic light system, people will have to follow the traffic rules and regulations (right-side priority, stop sign, yielding at a roundabout, etc). Otherwise, chaos will take place, as illustrated in the above picture. Such scenarios is called, a **deadlock**. People are waiting for each other, hopping that somebody will make a move. That move will never happen.

## 5.2 Race Condition and Data Corruption

To understand the importance of process synchronization in computer systems, consider a variable, `X`, of whatever type, that is shared between two threads, Thread 1 and 2. There respective codes are give as follows:

| **Thread 1** | **Thread 2** |
|---|---|
| (A) X:=1; | (D) X:=2; |
| (B) ... | (E) ... |
| (C) print(X); | (F) X:=3; |

What could be the value of `X` that is printed by Thread 1 when both threads are executed in real or fake-concurency? If you try to mentally run the code, you will realize that the value of `X` printed by Thread 1 can either be 1, 2, or 3. But which value is most likely? We could say that it will depend on the amount of code in (B) and (E). We could also say it will depend on which thread started the execution first. It could be relative to the used hardware if each thread is running on one CPU core. In reality, the correct answer and value depends on the logic of the program. That is, what does the program (or the two threads) try to do?

To understand what the `logic` of a program mean, let us consider another example. Let `i` be a shared variable between two Java threads, Thread 1 and Thread 2, where `i` is initially set to 5:

| **Thread 1** | **Thread 2** |
|---|---|
| `i = i + 1;` | `i = i - 1;` |

What is the final value of `i` when both threads are executed in **fake-concurency**?

```
Thread Thread 1 = new Thread();
Thread Thread 2 = new Thread();
Thread 1.start();
Thread 2.start();
```

You would again argue that the value of `i` could be: 4, 5, or 6. But which value is most likely? You could say, that would depend on which thread started first, which thread had the heighest priority, or on which hardware each thread was executed. Again, the correct value depends on the `logic` of the program. Well, what is this `logic`?

If the value of `i` was initially set to 5, then, we run the threads in parellel (fake-concurrency), the value of the variable `i` should be the same as its initial value, i.e., 5. This is because, logically, one thread will increment the variable (so, it makes `i` go from 5 to 6) and the other one will decrement it (so, it makes `i` go from 6 to 5). Also, if thy executed in the reverse order, i.e., decrement then increment, the value of the variable will go from 5 to 4, and then from 4 to 5. So, the consistent value is 5. The other two values, i.e., 4 and 6 are not correct. The software should not output such thing.

Now, certainly, you are maybe asking yourselves how do we get the two incorrect values, 4 and 6? Well, here it is, you will use your assembly and process management skills to clarify that. Let us take the previous example, in particular, the two statements: `i=i+1` and `i=i-1`, and let us write the two statements in x86 assembly. Let us pose `[0xB4FF55EF]` as the address of the variable `i`, and let `AX` ibe the accumulator register:

For the statement `i = i + 1` it would be :

```
I0 MOV AX, [0xB4FF55EF]
I1 INC AX
I2 MOV [0xB4FF55EF], AX
```

Also, for the statement `i = i - 1` it would be :

```
I3 MOV AX, [0xB4FF55EF]
I4 DEC AX
I5 MOV [0xB4FF55EF], AX
```

When the two threads are concurrently executing, their low-level instructions are interleaved in some order (due to fake-concurrency). In fact, we have seen in the previous chapter, that an interrupt may take place at anytime, and a context-switch may occur. In this example, we will suppose that a `timer` interrupt[1] happens every two instructions. So, if we assume

[1]In time-sharing systems, processes are scheduled (allocated the CPU) for a short-time slot, called quantum. When a quantum expires, a timer interrupt is raised, and the current process is interrupted to give the CPU to the next waiting process (in the ready queue).

that Thread 1 started executing first, then if `i` was initially set to `n`, the instructions, $I_0$, $I_1$, $I_2$, $I_3$, $I_4$, and $I_5$, could be executed as follows:

```
I0 MOV AX, [0xB4FF55EF]          results in AX=n
I1 INC AX                        results in AX=n+1
Timer interrupt:  context-switching T1 by T2
I3 MOV AX, [0xB4FF55EF]          results in AX=n
I4 DEC AX                        results in AX=n-1
Timer interrupt:  context-switching T2 by T1
I2 MOV [0xB4FF55EF], AX          results in i=n+1
T1 terminates
I5 MOV [0xB4FF55EF], AX          results in i=n-1
T2 terminates
```

The final state is `i=n-1` (in the previous example, `i=4`). This is an incorrect state, since the correct value for the variable `i` should be `n`. Furthermore, if we did not interrupt Thread 2 after executing $I_4$ and allowed it to execute $I_5$, we would have obtained the following execution order:

```
I0 MOV AX, [0xB4FF55EF]          results in AX=n
I1 INC AX                        results in AX=n+1
Timer interrupt:  context-switching T1 by T2
I3 MOV AX, [0xB4FF55EF]          results in AX=n
I4 DEC AX                        results in AX=n-1
I5 MOV [0xB4FF55EF], AX          results in i=n-1
T2 terminates
I2 MOV [0xB4FF55EF], AX          results in i=n+1
T1 terminates
```

The final state is `i=n+1` (in the previous example, `i=6`). This is again an incorrect state, since the correct value for the variable `i` should be `n`. This paradigm is called: **shared data corruption**, or **race-condition**. The problem of shared data corruption occurs when:

- Threads can execute in (fake) concurrency, i.e., any process/thread can be interrupted at any point in its instruction flow (e.g.,quantum expires, or I/O operation), and the CPU is assigned to another thread.

- Threads can execute in (real concurrency) parallel (e.g., two execution flows of two different processes/threads are simultaneously executing on separate CPUs or processing cores).

- These concurrent or parallel executing threads share some data.
- The outcome of the execution depends on the order in which those concurrent threads modified the shared variable.

Therefore, the concurrent or parallel execution of multiple threads may result in the corruption of shared data by among several threads. A multiprocessing system may reach an incorrect state if it allows the manipulation of shared data concurrently by multiple processes. We need a way to ensure that only one thread at a time can manipulate a shared variable among multiple concurrent threads. We need **synchronization mechanisms**.

## 5.3 Boolean Variables for Synchronization

From a programmatic viewpoint, one could propose to use a boolean variable to control access to a shared variable among a set of threads that are concurrently running in the system. This method may work sometimes, but may fail too. And when it fails, it is chaos.

To clarify that, let `i` be a shared variable between two threads, Thread 1 and Thread 2, where `i` is initially set to 5. And Let `free` be a boolean variable initialized to `true`:

| **Thread 1** | **Thread 2** |
|---|---|
| `while(!free);` | `while(!free);` |
| `free=false;` | `free=false;` |
| `i = i + 1;` | `i = i - 1;` |
| `free=true;` | `free=true;` |

If the threads, Thread 1 and 2, execute one after the other, i.e., Thread 1 then Thread 2, or Thread 2 then Thread 1, there will be no issues, and the final value of `i` will be 5. Nevertheless, if they execute in an interleaving fashion, problems occur. For example, among the possible execution scenarios, both threads read the value of `free` from the central memory to

one of the registers (e.g., `AX=true`). Before, the condition is tested by both threads, both have read the value `true`. This could happen if an interrupt happend just after one thread read the value from the memory, and the other thread got the chance to do the same. Now, both threads will test the value of `free` against the condition and get `while(!true)`, which bypasses the while-spinning loop. Both threads will pass to the next instruction statement, i.e., `i = i + 1` for Thread 1 and `i = i - 1` for Thread 2. Boom!!! race condition, everything is screwed up! Now, the variable `i` is corrupted.

What about if we used, two boolean variables instead of one. Let us try that. Let `i` be a shared variable between two threads, Thread 1 and Thread 2, where `i` is initially set to 5. And let `free1` and `free2` be two booleans initialized to `true`:

| **Thread 1** | **Thread 2** |
|---|---|
| `free1=false;` | `free2=false;` |
| `while(!free2);` | `while(!free1);` |
| `i = i + 1;` | `i = i - 1;` |
| `free1=true;` | `free2=true;` |

If the threads, Thread 1 and 2, execute one after the other, i.e., Thread 1 then Thread 2, or Thread 2 then Thread 1, there will be no issues, and the final value of `i` will be 5. Nevertheless, if they execute in an interleaving fashion, then a very interesting problem occurs. Let us explain that: Assume that both threads have read the value of the booleans. Then, both will test the variable against the condition. Boom!!! both threads are stuck in the while-spinning loop forever. This is called a **deadlock**. Deadlock is when processes start looking at each other forever without progressing.

So process synchronization is a multiprocessing or multi-threading paradigm that aims to manage and control the execution and the access to shared resources between multiple processes or threads. This is very Important when the order of execution matters, i.e., when there is `logic` to enforce. Also, it allows transparent process communication and preserve data integrity (data coherency). In real life, several synchronization mechanisms are used. They include: mutex, semaphores, monitors, and message passing.

## 5.4 Critical Section

In multi-threading and multi-processing systems, a critical section is a part of a thread code in which the program requests to use shared resources on which the access is mutually exclusive. So, basically, it is the part of the code that, if not properly managed, may cause data corruption. Hence, any thread that uses a shared resource — which access is mutually exclusive — is generally organized into three to five parts:

```
Thread()
{
       Remainder section

         Entry section

         Critical section

         Exit section

       Remainder section
}
```

In the `Entry Section`, the process places a request to the operating system to grant it access to the critical section, whereas in the `Exit Section`, the process informs the operating system of leaving the critical section to wake up any waiting processes. The **mutual-exclusive access** to a resource (e.g., a variable) ensures that only one process or thread at a time may be granted write-mode access to that resource while any other process or thread requiring it be denied reading or writing-mode access. The `Remainder sections` are section that contain other code that does not access any shared variable which access is mutually-exclusive.

**Example.** Let Thread 1 and Thread 2 be two **concurrent** Threads. Identify the critical sections for each thread.

| **Thread 1** | **Thread 2** |
|---|---|
| `u = v + 50;` | `b = p - 51;` |
| `i = i + 15;` | `i = i - 15;` |
| `w = v % 20;` | `z = p * 90;` |
| `v = i / 10;` | `p = z % 10;` |

After analyzing the above codes, we can determine that only variable `i` is shared by both threads. Thread 1 is accessing the variable `i` in the 2nd and 4th instruction statement, whereas Thread 2 is accessing it in its 2nd instruction statement. Thread 1 is accessing the variable in read and write mode in its 2nd instruction and in read-only mode in its 4th instruction. We say that Thread 1 is a `writer` (because it accesses the variables in write mode, at least once). Thread 2 accesses the variable, in its 2nd instruction, as a writer too. Hence, we will consider those three instruction statements as critical sections: For Thread 1, is it `i = i + 15` and `v = i / 10`, and for Thread 2, it is `i = i - 15`. The other instruction statements represent the remainder sections. Last but not least, to correct the code, we should complete it by inserting the entry and exit sections. We obtain the following:

| **Thread 1** | **Thread 2** |
|---|---|
| `u = v + 50;` | `b = p - 51;` |
| `Lock(i);` | `Lock(i);` |
| `i = i + 15;` | `i = i - 15;` |
| `Unlock(i);` | `Unlock(i);` |
| `w = v % 20;` | `z = p * 90;` |
| `Lock(i);` | `p = z % 10;` |
| `v = i / 10;` | |
| `Unlock(i);` | |

So, before using the shared variable, a Thread request exclusive access to the variable from the operating system by issuing a system call, here, we called it `Lock()/Unlock()`, to acquire exlusive access to the variable.

The **critical section problem** is all about designing a protocol (a solution) that processes or threads can apply to cooperate while accessing shared resources without any issues. Such protocol must satisfy the following three requirements:

**Mutual Exclusion.** At most, one process or thread is executing its critical section at a time (i.e., manipulating the shared exclusive-accessed resource).

**Progress.** A process or a thread that is outside its critical section (i.e., remainder section) should not block a process or thread from getting into their critical section.

**Bounded Waiting**. A process or thread must not wait indefinitely in the queue (or indefinitely spinning around) to access its critical section.

Also, the protocol should not make any assumptions about the speed or number of CPUs in the system, and any process may be interrupted during its critical section. This is fine as the operating system will guarantee that no other process will access and mess up with the resources that are already locked by the interrupted process.

## 5.5 Synchronization Mechanisms

Synchronization mechanisms are essential tools in concurrent programming, ensuring that multiple threads or processes can coordinate and share resources effectively. These mechanisms help prevent race conditions, deadlocks, and other concurrency-related issues.

### 5.5.1 Mutex

Our first synchronization mechanism is mutex locks. Mutex locks, for **Mut**ual **ex**clusion, is the simplest software-based mechanism (solution) for the critical section problem. The mechanism consists of a **boolean variable** `M` (i.e., `M` ∈{`true, false`}) that can be exclusively accessed and updated using two **primitives** (system calls). These two primitives are: the `acquire()` (entry section code) and the `release()` (exit section code) — A.k.a., `lock()` | `unlock()`. Mutex locks can be seen as: an open door to a room, that if closed and locked, no access to the room would be possible while the door is closed and locked. Access would be possible if the door is unlocked and opend again. Hence, with mutex locks, to access its critical section, a process

must first acquire the lock (i.e., execute `acquire(M)`), and then must release the lock (i.e., execute `release(M)`) when it is done executing the critical section (i.e., leaving the critical section).

If the lock is available (`M=true` — we also say, free, released, or unlocked), then a process can acquire the lock and change its value (to `M=false` — i.e., held, acquired, unavailable, or locked) and then enter the critical section. Otherwise, if the lock is not available (`M=false`), then a process cannot acquire the lock and cannot enter the critical section. Instead, the process will actively keep on waiting for the lock to become available (`M=true`).

Below is a classical implementation of a mutex lock primitives:

```
acquire(M)
{
    while(!M);
    M = false;
}
```

```
release(M)
{
    M = true;
}
```

```
Thread( )
{
  acquire(M);
  critical section;
  release(M);
}
```

As you can see, the use of mutex locks creates **busy-waiting** (i.e., continuous looping or spinning). We often called the mutex lock, a **spinlock**. If a process finds the lock unavailable, it spins around (i.e., it keeps trying until the lock is released by another process). Also, while spinning, no context-switching is required. This engenders an active waiting that may waste CPU cycles. Hence, mutex is useful when the critical section is short (i.e., the spinning time would be short and the overhead would be good since no context-switching occurs). Of course, at some point in time, a spinning process may get interrupted after its alloted quantum expires, giving the chance to the lock holder (process holding the lock) to release it. Recall, a good solution for the critical section must guarantee progress (i.e., a thread that is executing its remainder section should not block other threads from accessing the critical section), bounded-waiting (i.e., sooner or later, the lock will be released), and mutual exclusion (i.e., only one process should be running their critical section at a time).

Now, let go back to the implementation of mutex. If you take a look at the above code, you would certainly ask a question: Do we guarantee mutual exclusive access to the critical section with such software implementation? —— Initially, `M=true`? If you carefully read the code, you will realize that

mutual exclusive access to the critical section may still be violated, and we are not solving any problem here!!!

Just consider the following execution scenario:

- Thread 1 runs the statement `while(!M);` and leaves the loop.
- Context-switching occurs, switching Thread 1 with Thread 2.
- Thread 2 runs the statement `while(!M);` and leaves the loop.
- Thread 2 updates the mutex lock `M=false`.
- Context-switching occurs, switching Thread 2 with Thread 1.
- Thread 1 updates the mutex lock `M=false`.
- Both Thread 1 and 2 are now into the critical section.

So, what is going on here? Actually, the mutex lock would only make sense if the `acquire()` and `release()` primitives were **atomic** primitives. That is, when a process execute one of the primitive, the computer must make sure that no other process is executing the primitives at the same time. The execution of two premitive **must** be sequential what so ever.

### 5.5.2 Test_and_Set() and Intel's XCHG

In the previous subsection, we have seen that mutex can only make sense if the primitves were to be executed atomically. To understand the mechanism of atomicity, we often abtract and represent the implementation of the two primitives (`acquire()` and `release()`) using the **Test and Set (TAS) Instruction**. The aim is to implement a CPU instruction that could write to a given memory location and return its old value in an uninterrupted way (i.e., atomicaly — without being interrupted). If this is possible, then we can implement mutex and the latter will work for sure. The atomic hardware instruction can algorithmically be expressed as:

```
int test_and_set(int * L)
{
  int prev = *L;
  *L = 0;
  return prev;
}
```

```
remainder section
while (test_and_set(&lock) == 0);
critical section
lock = 1; //make it available
remainder section
```

Here in the function, the previous content of the lock `L` gets loaded into one of the registers (e.g., AX) and then the memory content is set to 0 ( `L=0`). If two CPUs execute `test_and_set()` at the same time, the hardware will ensure that the two function calls happen **atomically** and **sequentially**.

On CISC computers, in general, and Intel processors, in particular, the **XCHG** instruction has been integrated as part of the processors instruction set. The instruction allow a process to write to a memory location and return its old value in an uninterrupted way (E.g., `MOV AX, 0` then `XCHG AX, [0x0E3045]`). This instruction can algorithmically be expressed as:

```
int XCHG(int * L, int v)
{
  int prev = *L;
  *L = v;
  return prev;
}
```

```
remainder section
while (XCHG(&lock, 0) == 0);
critical section
lock = 1; //make it available
remainder section
```

Here in the function, the previous content of the lock `L` gets loaded into one of the registers (e.g., AX) and then the memory content is set to 0 (`L=0`). The execution of `XCHG()` on a CPU core is atomic and indivisible.

Consider the following 8086 assembly code to be executed in a single-processor system where `[0xF457]` is the address of a mutex lock

```
          MOV AL, 0
Label:  XCHG [0XF457], AL
          TEST AL, AL     //Test performs bitwise AND
          JZ Label
          ...  critical section ...
          MOV [0xF457], 1
```

It works fine because processor cannot switch from one process to another in the middle of XCHG instruction (takes two indivisible CPU cyles).

Nevertheless, this may fail in a multiprocessor or multicore system. In fact, the instruction does not lock access to the system bus. It only ensure that the `test_and_set` operation happen in two indivisible CPU cycle within a given processor (CPU). We need another option to work in multi-processor systems. Actually, Intel introduced the `LOCK` instruction. It allow one CPU to take full control of the system bus for two indivisible CPU cycle. No other processor will be able to access the system bus (and hence the central memory) until the holding-CPU finishes executing the `XCHG` instruction. The previous code becomes:

```
          MOV AL, 0
Label:    LOCK XCHG [0XF457], AL
          TEST AL, AL     //Test performs bitwise AND
          JZ Label
          ...  critical section ...
          MOV [0xF457], 1
```

---

**Exercise.** Could you think about some scenarios where the use of mutex may lead to a deadlock?

---

### 5.5.3 Semaphores

Semaphores are essential synchronization mechanisms in operating systems, providing a way to control access to shared resources and prevent race conditions. Semaphore are generalized software mechanisms for the critical section problem and process synchronization (ordering). They were developed by the computer scientist Edsger Dijkstra in 1962. They were used for the very first time in the "THE OS" operating system that was also developed by the same author in the early 1960s at the Eindhoven University of Technology (NL).

A semaphore consists of an **integer variable** `S` and two **atomic** primitives, `P()` and `V()`, for its modification. The primitive `P()` (for Proberen — to test in Dutch) allows to test the value of the semaphore and decrement it if greater or equal to 1, otherwise wait. The primitive `V()` (for Verhogen— to increment in Dutch) allows to increment the value of the semaphore. The primitive `P()` can also be denoted as `acquire()`, `wait()`, or `down()` . Similarly, the primitive `V()` can also be denoted as `release()`, `signal()`, or `up()`. The primitive are classically implemented as follows:

```
Semaphore S=1;

      P(S)                    V(S)
      {                       {
          while(S≤0);             S=S+1;
          S=S-1;
      }                       }
```

The primitive `P()` is used to acquire the semaphore `M`, whereas the primitive `V()` is used to release the semaphore `M`. Note the use of busy-waiting — the spinning instruction statement, in the code of the `P()`.

We distinguish two main types of semaphores:

- **Binary semaphores** are used when the semaphore value ranges only between 0 and 1 (i.e., Mutex lock is a particular case of semaphores).
- **Counting semaphores** are used when the semaphore value ranges over an unrestricted domain. For example, if consider a petrol station with six (06) petrol pumps, and a flow of vehicles is getting into the station to pump gas/diesel, we would use a counting semaphore, initialized to 06, to make sure that at most 06 vehicles are using the pumps at the same time, irrespective of which pump is being used. We would then write the following code for a given vehicle_i:

**Vehicle_i**

**Begin**

```
P(pump);

    Pump_gas() || Pump_diesel();

V(pump);
```

**End**

Also, we distinguish two styles of use:

- Mutual exclusion style, in which case `M` is initialized to $n \mid n \geq 1$.
- Waiting style, in which case `M` is initialized to 0.

**Example (Waiting-style).** Let Thread 1 and Thread 2 be two Java threads such that T1 should be executed after the termination of T2:

| **Thread 1** | **Thread 2** |
|---|---|
| `Code 1;` | `Code 2;` |

Such that:

Thread Thread_1 = new Thread();

Thread Thread_2 = new Thread();

Thread_1.start();

Thread_2.start();

The code is run on a muti-core (mutiprocessor) computer system. Let us use a semaphore to enforce the process execution order. Here, we need to use semaphores in their **waiting-style**. We obtain the following code:

| **Thread 1** | **Thread 2** |
|---|---|
| `P(S);` | `Code 2;` |
| `Code 1;` | `V(S);` |

We use a semaphore `S` initialized to 0: `Semaphore S = 0;` Like this, Thread 1 will be stuck (spinning around) when trying to acquire the semaphore `S`, whereas Thread 2 would run till completion, and then would release `S`. Next, Thread 1 would be able to acquire `S` and run till completion.

**Example (Mutual exclusion style).** Let `i` be a shared variable between two Java threads 1 and 2, where `i` is initially set to 5. We want to synchronize these two threads so that the final value of `i` remains 5, when the two threads are executed concurrently for a certain equal number of times:

| **Thread 1** | **Thread 2** |
|---|---|
| `i = i + 1;` | `i = i - 1;` |

Thread Thread_1 = new Thread();

Thread Thread_2 = new Thread();

Thread_1.start();

Thread_2.start();

We use a semaphore `S` in mutual exclusion style. That is, we initialize it to 1, and make each thread try to acquire the semaphore before accessing the variable `i`, and then releasing the semaphore one done with the variable.

We obtain the following code:

```
Semaphore S=1;
```

| **Thread 1** | **Thread 2** |
|---|---|
| `P(S);` | `P(S);` |
| `i = i + 1;` | `i = i - 1;` |
| `V(S);` | `V(S);` |

Furthermore, there exist two different implementations for semaphores: The first one **with busy waiting** (spinning), which is the implementation we have seen thus far, and the second **without busy waiting** (through the use of a waiting-queue). In the second implementation, a process can block itself with a system primitive `block()` when trying to acquire an unavailable semaphore and wake up another process (among the blocked processes) with a primitive `wake()` when releasing the semaphore. Hence, the semaphore definition becomes:

1. An integer `S` that holds the number of instances of a given resource available for processes to use simultaneously.

2. A semaphore queue `Q` where blocked processes wait for the availability of a resource (critical section).

The semaphore can only be modified by the two atomic primitives: `P()`, which tells the currently running process or thread to claim the control of a resource or make it wait if that resource is not available, and `V()`, which allows a process or thread to inform other processes or threads that it is done with that resource and to wake up a blocked process or thread in the queue if there are any.

The two primitives `P()` and `V()` become:

```
P(S)                              V(S)
{                                  {
  S = S - 1;                        S = S + 1;
  if(s<0)                           if(S≤0)
  {                                  {
     Block & place P in Q;              Wake up P from Q;
  }                                   }
}                                  }
```

The primitive `P()` decrements the value of `S` then: if `S<0` the "current" process is blocked and placed in the queue `Q`. Else it carries on. Also, the primitive `V()` increments the value of `S` then: if `S`$\leq$ `0` the "blocked" process in the head of the queue is dequeued and resumed.

Regarding the binary semaphore, the primitive `P(S)` nullifies the value of `S` and places all other processes to the waiting queue `Q` (semaphore queue). Also, the primitve `V(S)` makes the critical section available by setting the value of `S` to `1` or releasing a "blocked" process from the queue `Q`:

```
P(S)                              V(S)
{                                  {
  if(S==1) S = 0;                   if(Q is ∅) S = 1;
  Else                              Else
     Block & place P in Q;           Wake up P from Q;
}                                  }
```

Each process, once dequeued and resumed will **straightforwardly** start executing its critical section.

### 5.5.4 Peterson's Algorithm

Peterson's algorithm is a solution for the critical section problem involving two processes that alternate their critical section execution. It was proposed by Gary L. Peterson in 1981. Recall the critical section problem is all about designing a protocol that processes can apply to cooperate. Such protocol must satisfy the following three requirements: (1) **Mutual Exclusion**, where at most, one process is executing its critical section at a time. (2) **Progress**, where a process that is outisde its critical section should not prevent another process from accessing the critical section, and (3) **Bounded Waiting**, where a process must not wait indefinitely in the queue to access its critical section. The Peterson's algorthm worked fine on old systems. However, there is no guarantee that it will work correctly on modern architectures, in particular, those architectures that uses out-of-order execution.

In Peterson's algorithm, a process $\texttt{P}_i$ has the following structure:

```
do {
      Interested[i]=true;
      turn = j;
      while (Interested[j] && turn==j);
        critical section
      Interested[i]=false;
      ...
    } while(true);
```

The variable `Interested[]` indicates that a process is ready to enter the critical section, whereas the variable `turn` indicates whose turn it is to enter the critical section.

To understand how Peterson's algorithm works, let us consider two processes: $\texttt{P}_i$ and $\texttt{P}_j$. Let us try to run them in parallel. Initially, both processes will make the variable `Interested[]` equal to `true`. I.e., $\texttt{P}_i$ makes `Interested[i]` equal to `true` and $\texttt{P}_j$ makes `Interested[j]` equal to `true`. Then, let us assume that $\texttt{P}_i$ makes `turn=j` and enters the while condition. The condition is `true && true`, which stucks process $\texttt{P}_i$ to spin around the while condition. Next, $\texttt{P}_j$ makes `turn=i` and enters the while condition too. The condition is `true && true`, which stucks process $\texttt{P}_j$ to spin around the

while condition. However, by the time $P_j$ turned the variable `turn` to `i`, the first process $P_i$ get deblocked (condition became `true && false`), giving them access to the critical section.

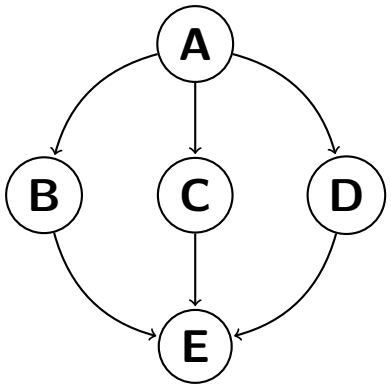
The limitation of Peterson's algorithm is that processes should be executed alternatively. Also, although the Peterson's algorithms fulfills the critical section problem requirements, it suffers from busy waiting.

### 5.5.5 Interrupt Disabling

To solve the problem of the critical section, certain old operating systems, running on single-processor systems, used to disable all interrupts when a shared variable is being manipulated (i.e., entering critical region), and then re-enable them just before leaving the critical region. Early versions of UNIX did that. However, this approach is unwise. Also, disabling interrupts would only affect the CPU that executed the disable instruction.

## 5.6 PPG for Synchronization

Recall PPGs (Process Precedence Graphs) are directed graph that are used to graphically express the order on which processes or threads are executed with respect to other processes or threads.




In a PPG, `(A)`$\rightarrow$`(B)` means that `A` must finish before `B` can start.

The above PPG could be expressed using the folloing `CoBegin-CoEnd` code:

**Begin**

A;

**CoBegin**;

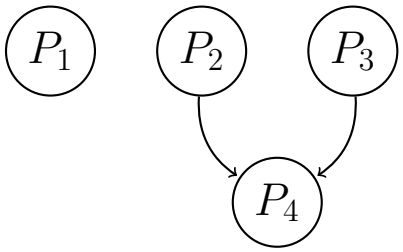
B, C, D

**CoEnd**;

E;

**End**

**Example**: Let us write `CoBegin/CoEnd` code for the following PPG :




The precedence constraints are: Process $P_1$, $P_2$, and $P_3$ should execute in parallel. However, process $P_4$ should execute after $P_2$ and $P_3$ terminate, and can execute before, after, or in parallel with $P_1$. Hence, the following code will make it:

**CoBegin**

**Begin**

**CoBegin** $P_2$, $P_3$ **CoEnd**

$P_4$;

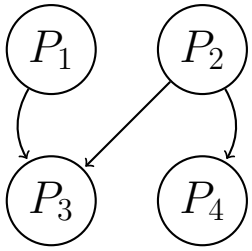
**End**

$P_1$;

**CoEnd**

Hence, we can use `CoBegin/CoEnd` construct to structure the execution of codes that have to be executed in parallel or sequential.

However, can we convert any PPG into a code that uses the construct `CoBegin/CoEnd`? Let us try with this example:




You may propose the following codes:

**Begin**
  **CoBegin** $P_1$, $P_2$; **CoEnd**
  **CoBegin** $P_3$, $P_4$; **CoEnd**
**End**

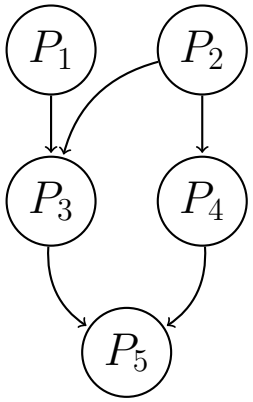
**CoBegin**
  **Begin** $P_2$, $P_4$; **End**
  **Begin** $P_1$, $P_3$; **End**
**CoEnd**

Both codes are wrong. The first one froces $P_1$ to terminate before $P_4$, where in reality, there is not precedence constraint between the two processes. The two processes may need to cooperate at some point. Also, in the second code, we are breaking the precedence constraint that $P_3$ should start after $P_2$ terminate. We are running them in parallel, which is wrong.

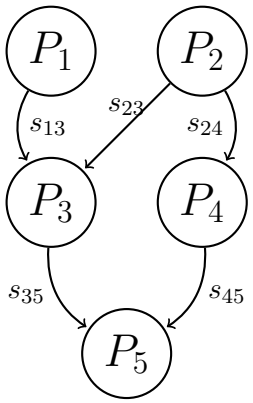
The truth is that, **CoBeing/CoEnd** construct cannot represent any arbitrary **precedence constraints**. Additional constraints are added in some situations, which may not be useful.

## 5.7 Semaphores with PPGs

We can use PPGs to manage semaphore assignment where solving critical section problems. For example, if we had a critical section problem and have managed to represent the processes that are involved in the problem using the following PPG, we then easily translate the precedence graph into a code using semaphores:




We can start with a brute-froce approach where we create as many semaphores as arrows:

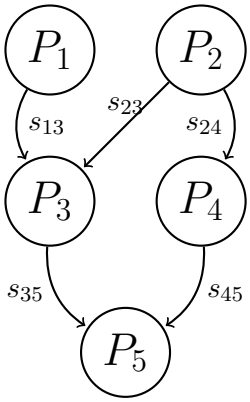


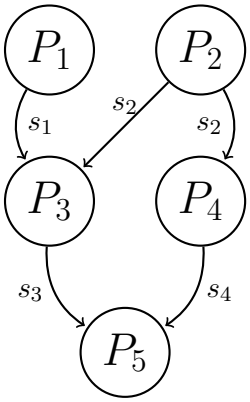

`s`$_{13}$`, s`$_{23}$`, s`$_{24}$`, s`$_{35}$`, s`$_{45}$`: Semaphore := 0;` (5 semaphores)

We can come up with the following codes:

| Process 1 | Process 2 | Process 3 | Process 4 | Process 5 |
|---|---|---|---|---|
| code$_1$; | code$_2$; | `P(`$s_{13}$`);` | `P(`$s_{24}$`);` | `P(`$s_{45}$`);` |
| `V(`$s_{13}$`);` | `V(`$s_{23}$`);` | `P(`$s_{23}$`);` | code$_4$; | `P(`$s_{35}$`);` |
| | `V(`$s_{24}$`);` | code$_3$; | `V(`$s_{45}$`);` | code$_5$; |
| | | `V(`$s_{35}$`);` | | |

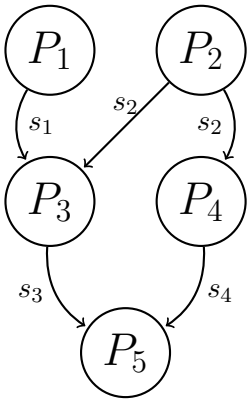
We can try to optimize and reduce the number of semaphores by renaming `s`$_{xy}$ with `s`$_x$:

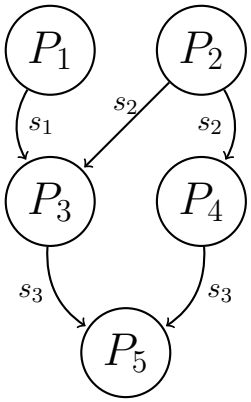



`s1, s2, s3, s4: Semaphore := 0;` (4 semaphores)

We can actually go further and eliminate the fourth semaphore:




`s1, s2, s3: Semaphore := 0;` (3 semaphores)

Our code becomes:

| Process 1 | Process 2 | Process 3 | Process 4 | Process 5 |
|---|---|---|---|---|
| code$_1$; | code$_2$; | `P(`$s_1$`);` | `P(`$s_2$`);` | `P(`$s_3$`);` |
| `V(`$s_1$`);` | `V(`$s_2$`);` | `P(`$s_2$`);` | code$_4$; | `P(`$s_3$`);` |
| | `V(`$s_2$`);` | code$_3$; | `V(`$s_3$`);` | code$_5$; |
| | | `V(`$s_3$`);` | | |

`s1, s2, s3: Semaphore := 0;` (3 semaphores)

Now, can we optimize the code further? Well, there is a way to reduce the number of repeated acquire primitive calls. For example, if within a

given code, of a given process, we are making $n$ acquire calls on the same semaphore, we can then make it one single call and initialize the semaphore to $-n+1$. We can apply this technique to our previous code on semaphore $s_3$ and obtain the following code:

| Process 1 | Process 2 | Process 3 | Process 4 | Process 5 |
|---|---|---|---|---|
| $\text{code}_1$; | $\text{code}_2$; | P($s_1$); | P($s_2$); | P($s_3$); |
| V($s_1$); | V($s_2$); | P($s_2$); | $\text{code}_4$; | $\text{code}_5$; |
| | V($s_2$); | $\text{code}_3$; | V($s_3$); | |
| | | V($s_3$); | | |

Semaphore $s_1 = 0$, $s_2 = 0$, $s_3 = -1$;

## 5.8 Classical Synchronization Problems

Classical synchronization problems, such as the producer-consumer problem, the dining philosophers problem, and the readers-writers problem, are fundamental to understanding and addressing concurrency-related issues in operating systems and multi-threaded applications. These problems highlight common scenarios where multiple processes or threads need to coordinate access to shared resources, and they serve as benchmarks for evaluating the effectiveness of synchronization mechanisms. By studying and solving these classical problems, developers can gain valuable insights into the challenges and best practices for designing and implementing concurrent systems.

### 5.8.1 Readers-Writers Problem

This problem consists of: (1) A set of reader threads $R = \{r_1, \ldots, r_n\}$ and (2) a set of writer threads $W = \{w_1, \ldots, w_m\}$. Both readers and writers share a data structure $D$ (e.g., file, database, variable, ...) such that:

- Multiple readers $R' \subseteq R$ can read at the same time.
- Only one writer $w_i \in W$ can write at a time.
- If $\exists r \in R$ that is reading, new writers have to wait (no preemption).
- If $\exists w \in W$ that is writing, new writers and readers have to wait.

- If $\exists r \in R$ that is reading, new readers can start reading.

When it comes to one reader $R = \{r\}$ and one write $W = \{w\}$, the solution is straightforward. We use a binary semaphore `m` in mutual exlusive style (i.e., initialized to 1):

```
Semaphore m = 1;

r()                                w()
{                                  {
   while(1)                            while(1)
   {                                   {
      acquire(m);                        acquire(m);
      read();                             write();
      release(m);                         release(m);
      do_something_else();                do_something_else();
   }                                   }
}                                  }
```

However, when it comes to multiple readers $R = \{r_1, \ldots, r_n\}$ and multiple writers $W = \{w_1, \ldots, w_m\}$, things get interesting. There exist three solutions: Readers have priority, Writers have priority, and starvation free solution. We discuss then in the next subsections.

#### Readers have priority

In readers have priority solution, we consider the following facts: when a new reader arrives while a writer is already waiting for previous readers to terminate, then the new reader **can** start reading (i.e., it joins) with the other readers without waiting. This solution is not a starvation-free solution because if there is a steady stream of readers, writers may not get the chance to write. We discuss the algorithm in the class.

#### Writers have priority

In this solution, when a new reader arrives and a writer is already waiting for previous readers to terminate, then the new reader **cannot** start reading along with the other readers but has to wait. And when there is waiting readers and writers during a writing, writers have the priority to be waken up over the readers at the end of the writing.Similar to the previous solution, this solution is not a starvation-free solution. In fact, if there is a steady stream of writers, readers may not get the chance to read. We discuss the algorithm in the class. Also, the solution will be the subject for Lab 8.

#### Starvation-free

In the starvation-free solution, readers yield for writers and writers yield for readers. Basically, when a reader shows up, it cannot start if a writer is waiting. Also, when writers are waiting, one writer can start after the last current reader finishes, i.e., a busy system: ..., one writer, batch of readers, one writer, ...

### 5.8.2 Dinning Philosophers Problem

This problem was originally formulated in 1965 by Edsger Dijkstra as a student exam exercise, presented in terms of computers competing for access to tape drive peripherals.

A set of five philosophers sitting around a table $P = \{p_0, p_1, p_2, p_3, p_4\}$ and five forks $F = \{f_0, f_1, f_2, f_3, f_4\}$. Each philosopher $P_i$ has one fork on his left side and one fork on his right side s.t:

- Each philosopher has a plate of spaghetti in front of him.
- A philosopher spends his time alternating between thinking & eating.
- Another version: Philosophers eating rice using chopsticks.
- A philosopher $P_i$ needs both forks ($f_i$ & $f_{i+1}$) to be able to eat.
- If a philosopher grabs one fork, it does not put it down till getting the second fork and eating (**Hold and wait property**).
- A philosopher is not allowed to take a fork from another philosopher's hands (**No preemption property**).
- A philosopher cannot take both forks at the same time.
- Parallelism increases with the number of philosophers.

In operating system,, this classical problem models a group of concurrent threads that share multiple resources with mutual exclusive access.

In the dinning philisopher, the code of each philosopher will have the following form:

```
Philosopher(Integer i)
{
   while(1)
   {
      Think();
      Eat();
   }
}
```

There exist multiple solutions for this problem, we discuss two of them.

**Solution 1.** Pick up a fork on the right side and then pick up another fork on the left side, then start eating the food.

```
Philosopher(Integer i)
   {
      while(1)
      {
         Think();
         acquire(Fork[i]);
         acquire(Fork[(i+1) mod 5]);
         Eat();
         release(Fork[i]);
         release(Fork[(i+1) mod 5]);
      }
   }
```

In this solution, no two adjacent philosophers will eat at the same time. If all philosophers start picking their first fork (`Fork[i]`), there will be a deadlock. In fact, Philosopher 0 will pick up `Fork[0]`, Philosopher 1 will pick up `Fork[1]`, Philosopher 2 will pick up `Fork[2]`, Philosopher 3 will pick up `Fork[3]`, and Philosopher 4 (the last one) will pick up `Fork[4]`. Then, when it comes to picking up the other fork, every single process will find the fork unavailable. Philosopher 0 will find `Fork[1]` unavailable, Philosopher 1 will find `Fork[2]`, Philosopher 2 will find `Fork[3]` unavailable unavailable, Philosopher 3 will find `Fork[4]` unavailable, and Philosopher 4 will find `Fork[0]` unavailable. Now, we are in a situation called **circular waiting**.

**Solution 2.** In this solition, a mix of left handed and right handed philosophers. This approach would break the circular waiting situation that we have encountered in the previous solution.

```
Semaphore Fork [5]; /* ∀i ∈ {0,1,2,3,4} Fork[i]=1 */
Philosopher(integer i)
{
   while(1)
   {
      Think();
      acquire(Fork[Min(i, (i+1) mod 5)]);
      acquire(Fork[Max(i, (i+1) mod 5)]);
      Eat();
      release(Fork[Min(i, (i+1) mod 5)]);
      release(Fork[Max(i, (i+1) mod 5)]);
   }
}
```

Actually, we are setting up a global ordering as per the request of resources. We are asking each philosopher to first pick up the fork with the small index (i.e., `i`) and then the one with the big index (i.e., `i+1`). This solution is deadlock-free and starvation-free.

If you look in the literature, you would find other interesting solutions. For example, there exists a solution where you force the system with a global lock (mutex) so that only one philosopher can eat at a given time. This of course has a clear performance bug. Another solution would use an arbitrator (waiter), where two forks are requested at a time, i.e., picking both forks in a critical section. This solution has reduced parallelismn becasue if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available. Another interesting solution consists of allowing philosophers to put down a fork if the other one is not available (but may starve due to simultaneous checking).

Last but not the least, you may think about obliging philosophers to wait for a random time before trying to grab the forks. This may work fine on certain applications (e.g., IEEE 802.3's CSMA/CD and IEEE 802.11's CSMA/CA) but not on other "critical" applications (e.g., Nuclear power plant system).

### 5.8.3 Producer-Consumer Problem

This problem consists of: (1) A set of producer threads $P = \{p_1, \ldots, p_n\}$ & (2) a set of consumer threads $C = \{c_1, \ldots, c_m\}$. Each producer thread $p_i$ produces an item and places it into a buffer $B$, and each consumer $c_i$ consumes one item from that buffer such that:

- A producer $p_i \in P$ produces an item if the buffer is not full.
- A consumer $c_i \in C$ consumes an item if the buffer is not empty.
- Producers and consumers must have exclusive access on the buffer.
- The buffer $B$ is bounded, i.e., has a limited size $n \in \mathcal{N}$.
- A.k.a., bounded buffer problem.

This principle is used in Unix pipes. For example, if you run the command: $ netstat -natp | grep chrome, then the first portion will run the command `netstat -natp`, which display active network connections, and then the output will be handed out to the second command, `grep chrome`, as an input to consume it and produce another output, which is displaying active network connections that contain the keyword "chrome".

For one producer and one consumer, we can come up with the following synchronization-free solution:

```
Producer()
{
   while(true)
   {
      item = Produce();
      Place(item, B);

   }
}
```

```
Consumer()
{
   while(true)
    {
            Take(B, item);
            Consume(item);

    }
}
```

Here, the producer produces an item and places it inside a shared buffer `B`. Then, the consumer accesses buffer to take the item and consume it. The access to the buffer must be mutually exclusive. To that end, we are going to use a binary semaphore `b=1`. Also, we are going to use two counting

semaphores, `m=N` and `e=0`. The counting semaphore `m` will indicate the number of empty spots available in the buffer so that the producer produces items, whereas the counting semaphore `e` will indicate the number of filled spots in the buffer so that the consumer consumes items. We can then mock the follwoing code:

```
Semaphore m = N; Semaphore e = 0; Semaphore b = 1;
```

```
Producer()
{
   while(true)
   {
      acquire(m);
      item = Produce();
      acquire(b);
      Place(item, B);
      release(b);
      release(e);
   }
}
```

```
Consumer()
{
   while(true)
    {
     acquire(e);
     acquire(b);
      Take(B, item);
      release(b);
       Consume(item);
       release(m);
    }
}
```

Finally, there exists several classical synchronization problems:

- Readers and writers problem.
- Dining Philosophers problem.
- Producers and consumers problem.
- Sleeping barber problem (by Edsger Dijkstra, 1965).
- Cigarette smokers problem (by Suhas Patil, 1971).

We covered three of them, the rest are for you to explore.

## 5.9 Using Semaphores for Synchronization

The following steps describe an informal procedure of writing codes with semaphores:

1. Understand the problem to solve: "Understanding a question is half an answer"-Socrates 470-399 BJC.
2. Determine which processes or threads are needed. E.g., Readers and writers.
3. Write down informally, conditions under which a process must wait for another, e.g., A reader waits if a writer is writing.
4. Figure out what counters and semaphores are needed.
5. Code and debug.

## 5.10 Deadlock

Deadlocks are a critical issue in operating systems, as they can lead to system crashes, resource wastage, and unpredictable behavior. Deadlocks occur when two or more processes are waiting for each other to release resources, creating a circular dependency that prevents any process from proceeding. By identifying potential deadlock scenarios, developers can implement strategies to avoid them, such as deadlock detection algorithms or careful resource allocation. Also, if a deadlock occurs, it's crucial to detect it as quickly as possible to minimize its impact. Deadlock detection algorithms can help identify and resolve deadlocks.

### 5.10.1 Deadlock Overview

By definition, a deadlock iIs a system state in which processes or threads are waiting for some events that will never occur (availability of some resources). A set of processes $P = \{p_1, \ldots, p_n\}$ is deadlocked if: $\forall p_i \in P : p_i$ is waiting for an event that can only be caused by another process $p_{j \neq i} \in P$.

Regarding the use of computer resourses by processes, we note that: A process/thread acquires some resources through a protocol — a.k.a., resource allocation sequence, where the process requests a resources (in general, through a syscall), obtains the resources, uses the resources, and then releases the resources (again, in general, through syscall). A resource could be physical (e.g., CPU, memory, and I/O devices) or logical (e.g., data structure, socket, variable, semaphore, etc). Also, a process cannot request a number of resources that exceeds the total number of available resources in the system.

Figure 5.1: Traffic Deadlock Situation in a City Intersection

Hence, the event that each process is waiting for is just the fact that a resource, logical or physical (**nonpreemptable**, i.e., cannot be taken away while being used), is to be released by another process.

**Example.** Consider a database where two processes P1 and P2 try to access two records for modification. Process P1 from User 1, locks record r1 then tries to lock record r2. Process P2 from User 2 however, locks record r2 then tries to lock record r1 as illustrated below:

```
Semaphore r1_lock = 1;
Semaphore r2_lock = 1;

        Process P1                Process P2

        P(r1_lock);               P(r2_lock);
        P(r2_lock);               P(r1_lock);
        Update(r1 and r2);        Update(r1 and r2);
        V(r2_lock);               V(r1_lock);
        V(r1_lock);               V(r2_lock);
```

We can clearly see that when P1 and P2 execute, they will both get stuck in their second instruction statement when both try to acquire the second lock (`r2_lock` for P1 and `r1_lock` for P2). Now, P1 is waiting for P2 to release `r2_lock`, and P2 is waiting for P1 to release `r1_lock`. This is a deadlock. You can conclude that a minor difference in coding style may lead an entire information system to not work properly.

The code should have been written as:

```
Process P1                  Process P2

P(r1_lock);                 P(r1_lock);
P(r2_lock);                 P(r2_lock);
Update(r1 and r2);          Update(r1 and r2);
V(r1_lock);                 V(r1_lock);
V(r2_lock);                 V(r2_lock);
```

Deadlock is possible (i.e., may occur) due to the presence of some aspects in a given program or application's code, called the Necessary Conditions:

1. **Mutual Exclusion.** A resource (critical section) can only be allocated to one process at a time. Because we want to make sure that only one thread access a shared resource at a time, we use synchronization mechanisms to enforce mutual exclusion, but fail to control deadlocks.

2. **No Preemption.** A resource (critical section) cannot be taken away from any process during its critical section execution. Otherwise, the ressource will be corrupted. Imagine that two threads are simultaniously using a printer, and their printing is happenning on the same paper.

3. **Hold and Wait.** Once a resource (critical section) is taken, a process does not release it till it is done with it. For example, in the dining philosophers problem, in Solution 1, we could not take a fork from a philosopher which is already holding on it. That resulted in a deadlock when all philosophers tried to acquire their second fork.

4. **Circular Wait.** When each process $P_i$ is waiting for a resource held by process $P_{(i+1)[n]}$, a circular waiting scenario may occur.

To deal with deadlocks, we need to focus on the last two conditions. The mutual exclusion and the non-preemption are necessary and important — we cannot eliminate them. In general, there exists three main approaches to deal with deadlocks:

1. **Deadlock Ignorance.** It is the user who has to detect deadlocks and take actions (e.g., terminate processes or take back resources). This is used by most operating systems, e.g., GNU/Linux and Windows.

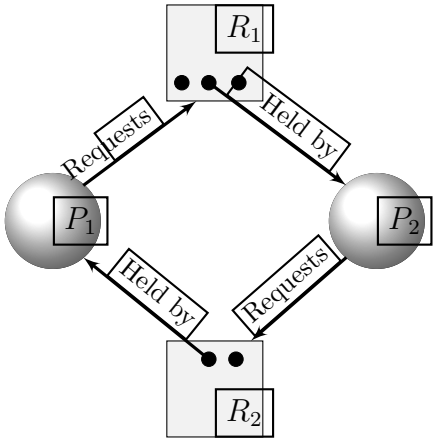
2. **Deadlock Avoidance and Prevention**: Ensure that the system will **never** enter into a deadlocked state:

   - **Deadlock Prevention.** This uses static rules for requesting resources. E.g., a process must request all needed resources at a time (in the dinning philosophers, a philosopher would grab both forks at the same time).
   - **Deadlock Avoidance.** This uses dynamic rules. For each request, the operating system determines whether a deadlock may arise or not, before granting a given resource to a given process (e.g., RAGs, Banker's Algorithm).

3. **Deadlock Detection and Recovery.** Mostly used in DBMSs for transactions execution. Rollback if a deadlock is detected (i.e., two algorithms r needed, one to detect and one to recover).

### 5.10.2 Resource Allocation Graphs (RAGs)

Resource allocation graphs, not to be confused with PPGs, are directed graph where the vertices $V = P \cup R$ represent the set of processes $P = \{P_1, \ldots, P_n\}$ and system resources $R = \{R_1, \ldots, R_m\}$, whereas the edges $E$ express which process $P_i \in P$ has requested or is holding a given resource $R_j \in R$.

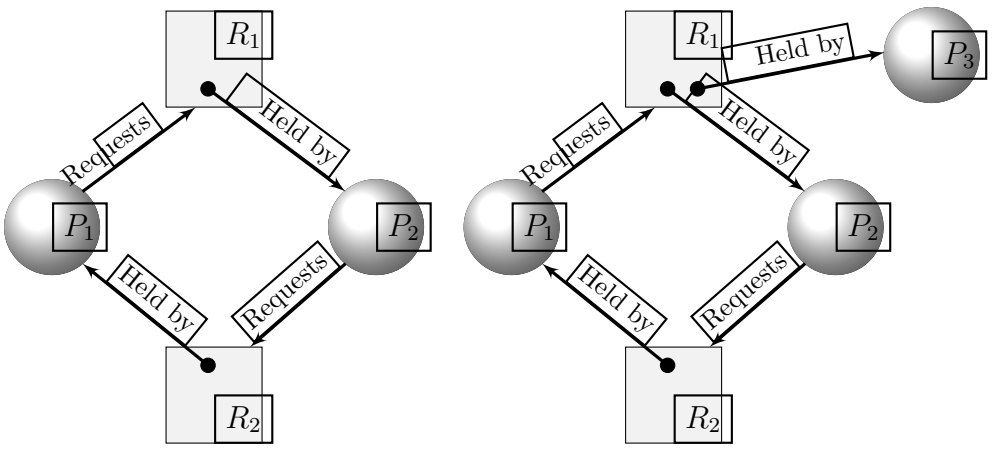



- An edge from process $P_i \in P$ to a resource $R_j \in R$, signifies that process $P_i \in P$ has requested an instance of resource type $R_j \in R$ and its currently waiting for it ($P_i \rightarrow R_j$ is called request edge).

- An edge from a resource $R_i \in R$ to a process $P_j \in P$, signifies that process $P_j \in P$ has been allocated an instance of resource type $R_j \in R$ ($R_i \rightarrow P_i$ is called assignment edge).
- Each process $P_i \in P$ is depicted by a circle, whereas resources $R_i \in R$ are represented by rectangles.

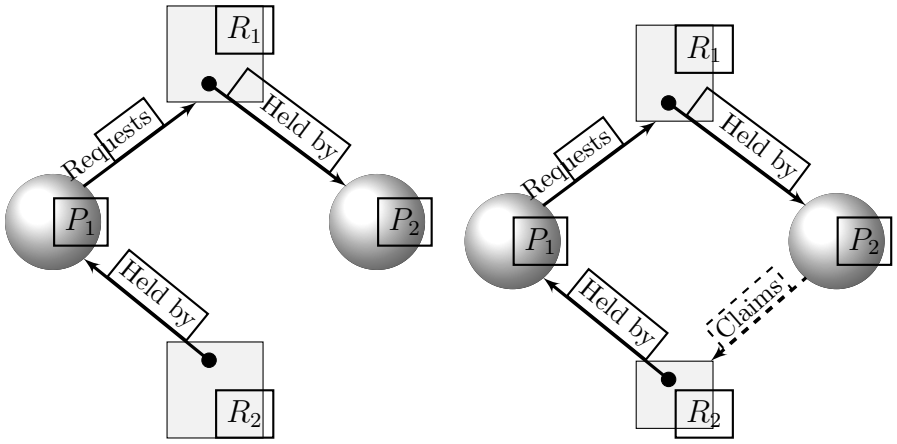
We can use a RAGs (Resource Allocation Graphs) to identify whether a particular resource allocation sequence `S` is deadlock-free or not. If a given RAG does not contain any closed path (cycle), then the sequence is **deadlock-free**. Otherwise, deadlock may or may not happen. In fact, we could have multiple instances of a given resource, or could have processes that may release a given resource soon ($\exists$ a way out).

Consider the following two resource allocaiton graphs. On the left-side, the RAG contains a cycle and has one single instance of a resource per resource. Hence, there is a deadlock. Process P1 is waiting for the release of resource R1, while P2 is holding on it and waiting for resource R2, which is held by process P1. On the right-side graph, there exits a cycle but no deadlock. This is because one of the processes, here P3, will release an instance of resource R1, which is needed by process P1. There will be not deadlock as P1 will have its required resources to run till completion.



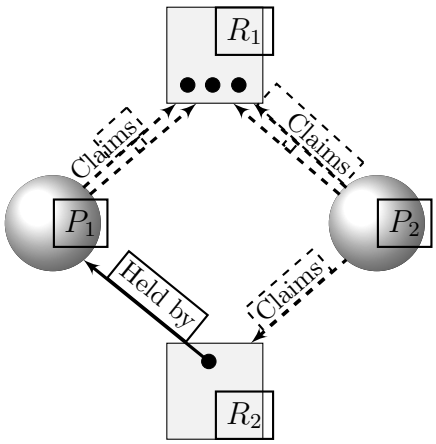

It is important to note that if there is **no deadlock now**, it does not guarantee anything about **no deadlock in the future**. If we consider the following RAG (the one below, on the left-side), we have process P1 is

requesting resource R1 held by process P2 while holding on resource R2. If process P1 terminates execution before process P2 requests (we never know) an instance of resource R2, then there will be no problem. Nevertheless, if P2 issues a request for an instance of resource R2, then a cycle will be formed and a deadlock occurs.

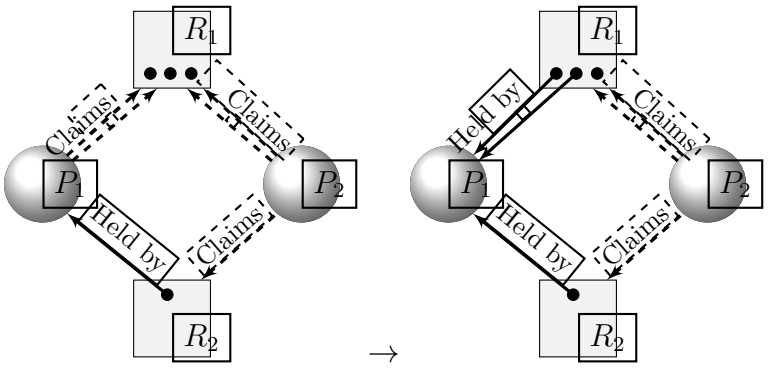



Then, you may ask yourselves a question and say: how do I know that a process may request a resource in the near future before other processes terminate execution and release some resources. We need to extend the resource allocation graph with a third type of edge, called **claim edge**. It is represented using dashed lines (viz., RAG on the right-side) and it indicates that a process may issue a request for a given resource. If the request is issued, the claim edge becomes a request edge. With such edge, we can now have two possible scenarios, one safe and the other unsafe (may lead to deadlock). In fact, resource allocation graphs with claim edges means that there is a potential for deadlock if the operating system blindly grants some requests to some processes.

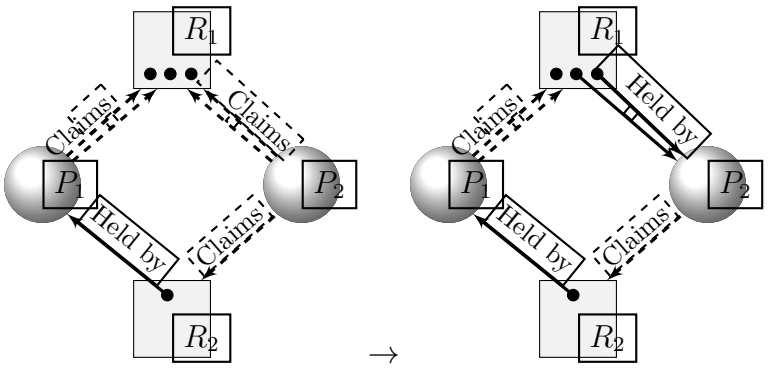
**Example.** Consider the following resource allocation graph:




If the operating system grants P1 two instances of resource type R1, then: no deadlock.




But if the operating system grants P2 two instances of resource type R1, then: deadlock may occur (Unsafe state).



## 5.11 Deadlock Handling Techniques

### 5.11.1 Deadlock Prevention

In deadlock prevention systems, the operating system ensures that at least one of the four necessary conditions cannot hold by enforcing **static rules**. We have mentioned earlier that we should not touch the mutual exclusion and the no-preemption conditions, we need those two. We focus on the two remaining ones, the hold and wait, and the circular waiting. Deadlock prevention uses static rules and deals with the previous two conditions as follows:

1. **Hold and Wait.** Here, the system should not allow a process to hold resources and wait for other resources. To that end, systems with deadlock prevention requires from the process to get all the resources at a time or get nothing. E.g., if a process need to copy a file from a DVD-ROM device to the HDD drive, then print the document using a printer, then the process will have to acquire the three resources first, and then start the task of copying and printing. This solution may engender starvation as processes will have to wait for resources that are not being used while they are waiting. Also, if one of the resources gets faulty, the system would face a problem of discontinuity.

2. **Circular Wait.** The system imposes a global ordering as per requesting resources, e.g., assuming resources are indexed, then we could use an increasing order of enumeration. This is what we did with the dinning philosophers when we asked then to pick up the fork with the small index and then the one with the big index. However, this might not be ideal for all processes.

Applying static rules is not always a good option as sometimes there is an iInability to handle unforeseen needs. For example, If a process has unforeseen resource needs during execution, static rules might not be able to adapt.

### 5.11.2 Deadlock Avoidance

Systems that use deadlock avoidance apply dynamic rules. For each received request, the operating system determines whether a deadlock may arise or not before granting a given resource to a given process.

A safe state is a system state in which the system can allocate resources to processes in some order that avoids ending up in a deadlock. In a system with $n$-processes and $m$-resources where processes request different numbers and types of resources there will be different execution sequences for the running processes, e.g., $< P_1, P_3, P_2, \ldots, P_k >$. Some of those sequences leave the system in a safe sate, whereas others end up into an unsafe state.

Deadlock avoidance systems tries to find whether a **safe sequence** exists. If a safe sequence exists, then given the current state of the system, there is a scenario to execute the running processes without ending up in a deadlock. A **safe sequence.** Is a sequence $S$ of processes: $\forall P_i \in S$, the resources requested by $P_i \in S$ can be allocated from the currently available resources plus the resources held by all $P_{j<i} \in S$ (i.e., process $P_i \in S$ may wait).

**Example.** Consider a system that has 12 instances of a given resources type e.g., printer, and consider three processes $P_0$, $P_1$, and $P_2$ such that:

- Process $P_0$ needs 10 printers to complete its execution.
- Process $P_1$ needs 4 printers to complete its execution.
- Process $P_2$ needs 9 printers to complete its execution.

If at a given time $t_0$ process $P_0$ is holding 5 printers, and $P_1$ and $P_2$ are holding 2 printers each (Here, the system is supposed to be in a safe state.

The sequence $< P_1, P_0, P_2 >$ is a **safe sequences**, whereas the sequences $< P_0, P_1, P_2 >, < P_0, P_2, P_1 >, < P_2, P_1, P_0 >, < P_2, P_0, P_1 >$, and $< P_1, P_2, P_0 >$ are not. They drive the system into a deadlock (unsafe state). At $t_1$, if $P_2$ is allocated one additional printer, the system goes from a safe state to an unsafe state.

The idea of deadlock avoidance consists of **determining** whether a given process that is requesting a given resource **must wait** or **must be immediately allocated** the requested resource.The principle is that if a resource is allocated, then the system should remain in a safe state. A resource is granted **only if** the allocation leaves the system in a safe sate. Hence, if a process requests a resource that is **currently available**, it may still have to wait.

Having a set of information (e.g., available resources, claimed resources, held resources, etc), the operating system can operate the **Banker's Algorithm** (by Edsger Dijkstra) to determine whether a set of processes can be run till completion without deadlock (safe state) or not (unsafe state).

The **Banker's algorithm** is an algorithm that allows the avoidance of deadlocks by determining whether a safe sequence $S$ exists or not. The algorithm uses four data structures:

1. **Available.** A vector of size $m$ (resource types) to express the currently available number of instances for a given resource $R_j$.

   $$Available[j] = k : \text{there are } k \text{ instances of resource type } R_j$$

2. **Max.** A matrix of size $n \times m$ to express the maximum number of resources to be requested by a given process.

   $$Max[i, j] = k : \text{at most } k \text{ instances of resource type } R_j \text{ will be requested by process } P_i.$$

3. **Allocation.** A matrix of size $n \times m$ to express the currently held instances of each resource type by a given process.

   $$Allocation[i, j] = k : \text{Currently there are } k \text{ instances of resource type } R_j \text{ held by process } P_i.$$

4. **Need.** A matrix of size $n \times m$ to express the remaining number of instances of each resource type to be requested by a given process.

   $$Need[i, j] = k : \text{There are } k \text{ instances of resource type } R_j \text{ to be requested by process } P_i \text{ at anytime in the future.}$$

$$\textbf{Need[i, j] = Max[i, j] - Allocation[i, j]}$$

If some requests were issued by a process $P_i$, and let $\texttt{Request}_i = < \sharp R_0, \ldots, \sharp R_m >$ be the request vector for process $P_i$, the request algorithm will operates as follows:

1. If $\texttt{Request}_i \leq \texttt{Need}_i$, goto 2, else Error. in fact, a process $P_i$ that asks for more than what it has initially declared (i.e., $\text{Max}_i$) will be immediately terminated by the system.

2. If $\texttt{Request}_i \leq \texttt{Available}$, goto 3, else $P_i$ must wait.

3. The system pretends having (hypothetically) allocated the requested resources to $P_i$ by updating the data structures:

$$\texttt{Available} = \texttt{Available} - \texttt{Request}_i$$

$$\texttt{Allocation}_i = \texttt{Allocation}_i + \texttt{Request}_i$$

$$\texttt{Need}_i = \texttt{Need}_i - \texttt{Request}_i$$

So assuming that a given resource request, e.g., $\texttt{Request}_i$ has been fulfilled, the system is brought to a new state. We run the Banker's algorithm:

1. Let `Work` and `Finish` be two vectors of length `m` and `n`, respectively, such that `Work=Available` and $\forall$`i: Finish[i]=false`.

2. Find an index `i`:

   - `Finish[i] == false`
   - $\texttt{Need}_i \leq$ `Work`

     `Else goto 4.`

3. $\texttt{Work} = \texttt{Work} + \texttt{Allocation}_i$ ;

   `Finish[i]=true`;

   `goto 2`;

4. `If` $\forall$`i:Finish[i]==true` then the system is in safe state.

### 5.11.3 Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In such environments, two algorithms are used. The first one is the deadlock-detection algorithm, which examines the state of the system to determine whether a deadlock has occurred or not. The frequency of invoking this algorithm depends on the likelihood of a deadlock and how many processes will be affected by a deadlock. The system may apply static rules, such as, when the CPU utilization drops below 40% then examin the system, or execute examin the system each

1 hour. A much tougher approach would be to examin the system after each resource request (Overhead in computational time). The second algorithm is the deadlock-recovery algorithm, which tries to drive back the system into a safe state.

If a system contains resources that have only one instance, then we can use an algorithm that exploits a variant of the RAG, a.k.a., **wait-for-graph**, or WFG. Similar to RAG, WFG is a directed graph in which the nodes represent processes and the edges represent a process waiting for another process. In a WFG, the meaning of an edge $P_i \rightarrow Pj$ is that process $Pi$ is waiting for a resource that is held by process $P_j$. The WFG is obtained from the RAG by removing the resource nodes and fusing the appropriate edges. In a WGF, if there exists a **cycle** then the system is in a deadlock state.

Let $\mathcal{R}$ be a resource-allocation graph and let $\mathcal{W}$ be its corresponding wait-for-graph then:

$$\forall(P_i, P_j, R_r) : \ (P_i \rightarrow R_r) \in \mathcal{R} \ \wedge \ (R_r \rightarrow P_j) \in \mathcal{R} \implies (P_i \rightarrow Pj) \in \mathcal{W}$$

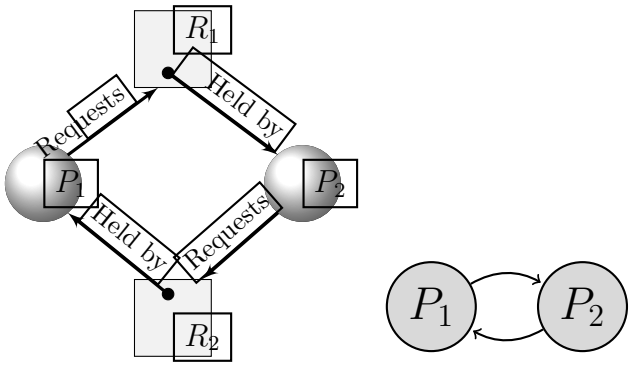



Figure 5.2: Converting a RAG to WFG

However, if a system contains resources that have multiple instances, then the system can apply a modified version of the Banker's algorithm to detect deadlocks:

1. Let `Work` and `Finish` be vectors of length `m` and `n` respectively such that `Work=Available` and $\forall$`i, if Allocation`$_i \neq 0$`: Finish[i]=false. Else, Finish[i]=true.`

2. Find an index `i`:

- `Finish[i] == false`
- $\texttt{Request}_i \leq \texttt{Work}$

  `Else goto 4.`

3. $\texttt{Work} = \texttt{Work} + \texttt{Allocation}_i$ ;

   `Finish[i]=true;`

   `goto 2;`

4. `If` $\exists$`i:Finish[i]==false` then the system is deadlocked.

When a system determines that a deadlock is taking place. A deadlock recovery routine is executed:

- **Process termination.** Abort processes involved in a deadlock, all at a time or one by one till deadlock is gone. Which process to terminate would depend on the process priority, number of resources held by the process, how much is left to terminate the process, dependency with respect to other processes, etc. Basically, the processes which termination will not incur an expensive cost should be aborted.
- **Resource preemption.** Taking away resources from certain process and allocating them to others.

# Exercises

In the following, we provide a list of exercises related to different topics covered in this course reader. Solutions can be discussed during office hours.

**Exercise 1.** Provide definitions and explanation for the following questions, related to process synchionization.

1. What is a critical section in a given program?

2. Explain how old uniprocessor operating systems (e.g., earlier version of UNIX) handled the critical section problem.

3. Explain the two styles of using semaphores.

4. Can a process be interrupted (context switched) during the execution of its critical section? Explain your answer.

5. Why disabling interrupts system during critical section execution is not a good idea in multiprocessor systems?

6. What are the three requirements for the critical section problem.

7. What are the two main advantages of implementing the primitives `acquire()` and `release()` using queues instead of using a busy waiting statement in `acquire()` and an incrementation statement in `release()`.

8. Semaphores are mutual-exclusive accessed shared variables used to set up synchronization among cooperative processes. Explain why semaphores cannot be used when cooperative processes are executing on different computers?

9. Monitors is a high-level programming concept used to set up process synchronization. Monitors use semaphores in their low-level implementation but provide programmers with programming facilities to make synchronization coding less error-prone. Explain how monitors work?

10. Pure monitors only provide mutual exclusion. Usually, for process synchronization, we also need to set up process ordering i.e., execute process P1 before P2. Which data structure do we need to do that? Explain how that data structure works?

11. In monitors, we can use the `notify()` primitive to wake up a sleeping process from the waiting queue. This primitive appears to have two semantics. Why is that possible? Explain the difference between the two semantics.

12. Explain under which circumstances, it is not correct to use the `notify()` primitive to wake up a sleeping process from the waiting queue, but the `notifyAll()` primitive should be used instead.

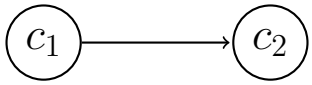
**Exercise 2.** Let x and y be two shared variables stored in the main memory. Assuming that the two variables are initialized to 10, what can be the possible outcomes for x and y if the following two processes are executed concurrently?

Process A : `x = x + y` Process B : `y = x * x`

**Exercise 3.** Draw a precedence graph (a.k.a., dependency graph) that reflects the parallelism in the following code. Recall a precedence graph is a *directed graph* in which the nodes (vertices) represent processes and the edges represent the causal relation between two processes. If an edge exists from process node $P_i$ to $P_j$ then the process $P_j$ must start executing its code $c_j$ only when process $P_i$ finishes executing its code $c_i$. Also we use the notation `ParBegin` and `ParEnd` to indicate a block in which program codes (or instructions) must run in parallel. Another alternative notation that can be found in the literature is `CoBegin` and `CoEnd` for concurrent begin/end.

```
Begin
  c0
  ParBegin
    c1
    Begin
      c2; c3; c4
    End
    c5
    c6
    Begin
      c7; c8; c9
    End
  ParEnd
  c10
End
```

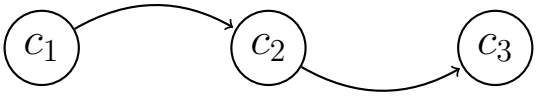
**Exercise 4.** Complete the following program code to reflect the process precedence graph shown below (where $c_i$ refers to the Code $i$, and $i \in \{1, 2\}$). This code basically consists of creating two processes in parallel: the first process executes program $P_1$ and the second process executes program $P_2$. You should use one semaphore for that.




| **P1:** | **P2:** |
|---|---|
| Begin | Begin |
| Code 1; | Code 2; |
| End | End |

ParBegin **P1**, **P2** ParEnd

**Exercise 5.** Complete the following program code to reflect the process precedence graph shown below (where $c_i$ refers to the Code $i$, and $i \in \{1, 2, 3\}$). This code basically consists of creating two processes in parallel: the first process executes program $P_1$ and the second process executes program $P_2$. You should semaphores for that.




| **P1:** | **P2:** |
|---|---|
| Begin | Begin |
| Code 1; | Code 2; |
| Code 3; | ; |
| End | End |

ParBegin **P1**, **P2** ParEnd

**Exercise 6.** Let Process $P_1$ and $P_2$ be two processes that execute concurrently using a shared semaphore $s$ initialized to 10. Each process increments, in an infinite loop, an integer $i$ (process $P_1$) or $j$ (process $P_2$) both initialized to 0. After executing for a while, the values of $i$ and $j$ will change considerably. In this case, what will be the relation between the two variables?

**Begin**
Semaphore M = 10;
Integer i = 0;
Integer j = 0;
**ParBegin** $P_1$, $P_2$ **ParEnd**
**End**

| **P1** | **P2** |
|---|---|
| Begin | Begin |
| while(true) { P(M); i++; } | for(int x=10; x>=0; x- -) { j++; V(M); } |
| End | End |

**Exercise 7.** Consider the following code:

**Begin**
Semaphore M=0;
Semaphore N =0;
**ParBegin** P1, P2 **ParEnd**
**End**

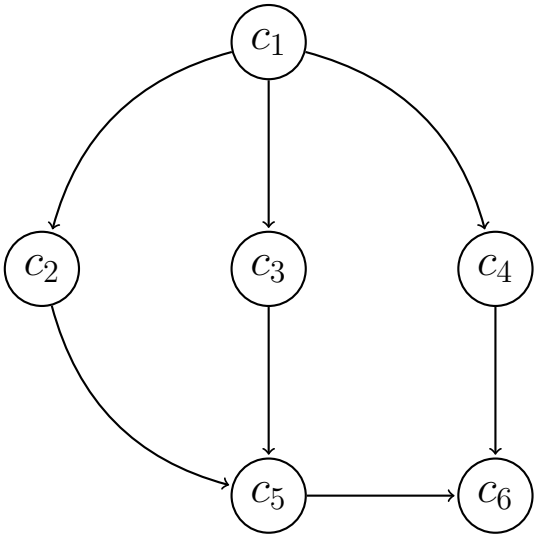
| P1: | P2: |
|---|---|
| **Begin** | **Begin** |
| P(N); | P(M); |
| Code 1; | Code 2; |
| V(M) | V(N); |
| **End** | **End** |

(a) Give the precedence graph for the above code and check whether a deadlock may occur or not. If there is a deadlock, fix it.

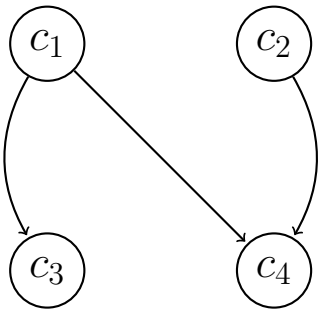
(b) Modify the code of the previous question so that the two programs $P_1$ and $P_2$ execute alternatively (i.e., $\ldots, P_1, P_2, P_1, P_2, P_1, \ldots$ with $P_1$ being the first to start execution.

(c) Modify the previous program (using two semaphores) so that the execution order becomes $\ldots, P_1, P_2, P_2, P_1, P_2, P_2, P_1, \ldots$) with $P_2$ being the first to start execution.

**Exercise 8.** Write the program code that reflects the following precedence graph using six processes $P_1, \ldots, P_6$ each executing its dedicated code $c_1, \ldots, c_6$. You are required to use three semaphores.




**Exercise 9.** Consider the following notation: let $\mathrm{Bc}_1$ (for Begin $\mathrm{c}_1$) expresses the starting of the code $\mathrm{c}_1$ and $\mathrm{Ec}_1$ (for End $\mathrm{c}_1$) the ending of the code $\mathrm{c}_1$, and let the precedence graph shown below reflects the execution order of program codes $\mathrm{c}_1$, $\mathrm{c}_2$, $\mathrm{c}_3$, and $\mathrm{c}_4$. Then, do the execution sequences shown below respect the precedence graph constraints or not?



a) $Bc_1Ec_1Bc_2Ec_2Ec_3Bc_3Ec_4Bc_4$
b) $Bc_1Bc_2Ec_2Ec_1Bc_3Bc_4Ec_3Ec_4$
c) $Bc_1Bc_2Ec_1Ec_2Bc_3Bc_4Ec_3Ec_4$
d) $Bc_1Bc_2Ec_2Ec_1Bc_3Ec_3Bc_4Ec_4$
e) $Bc_1Bc_2Ec_2Bc_3Ec_1Bc_4Ec_4Ec_3$
f) $Bc_1Bc_2Ec_2Bc_3Ec_3Ec_1Bc_4Ec_4$
g) $Bc_1Bc_2Ec_1Bc_3Ec_2Ec_3Bc_4Ec_4$
h) $Bc_2Ec_2Bc_1Ec_1Bc_4Ec_4Bc_3Ec_3$

**Exercise 10.** Consider the Peterson's algorithm for the critical section problem with two processes.

**P0**
```
Begin
  While(true)
  {
    (1) Flag[0]=ture;
    (2) turn=1;
    (3) while(Flag[1] && turn==1);
    (4)... critical section ...
    (5) Flag[0] = false;
  }
End
```

**P1**
```
Begin
  While(true)
  {
    (1) Flag[1]=ture;
    (2) turn=0;
    (3) while(Flag[0] && turn==0);
    (4)... critical section ...
    (5) Flag[1] = false;
  }
End
```

a) If the two statements (1) and (2) are swapped in both processes, state whether the altered code still satisfies the three requirements of the critical section problem. Explain which requirement has been violated.

b) If we change the logical operator in the condition of the while statement (Line 3) from `and` to `or` (i.e., while(flag[1-i] && turn==1-i) becomes while(flag[1-i] || turn==1-i)), state whether the altered code still satisfies the three requirements of the critical section problem. Explain which requirement has been violated.

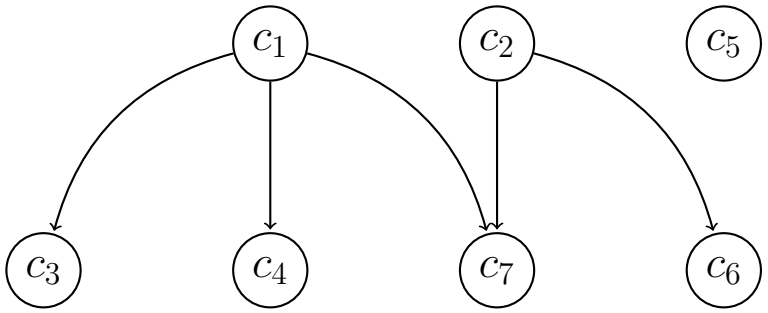
**Exercise 11.** Hardware solutions to the critical section problem use read-modify-write instructions: these instructions are indivisible because the CPU gets to hold onto the memory bus for two cycles, to both read and then update the shared memory location. As an example, here is code for implementing mutual exclusion using a Swap () machine-code instruction; that instruction has the opcode EXCH in the Pentium instruction set. Each process $P_i$ executes the same code to access its critical section, where `key` is a boolean local variable and `lock` is a shared memory location among processes $P_i$ that contains a boolean value, initially set to `false`.

**Process $P_i$:**

```
do {
        key = true;
        while (key == true) Swap(&lock, &key);
        ... Critical section of P_i ...
        lock = false
    } while(true);
```

Describe what happens if three processes try to enter their critical section at about the same time. How does the given code ensure that only one process gets entry?

**Exercise 12.** Write CoBegin/CoEnd code for the following precedence graph. Make your program express as much parallelism as possible within the limitations of CoBegin/CoEnd, while being sure to enforce all the constraints that are in the precedence graph. (The best solutions introduce two or three extra precedence edges. Try to find one of them.)




**Exercise 13.** Consider the following producer-consumer problem: A customer enters a fastfood to grab $K \geq 1$ sandwiches and leave. If no sandwich is available (`N==0`), the customer waits in a queue. Customers can enter the shop at anytime and stand in the queue if no sandwich is available. A cooker, prepares $S \geq 1$ sandwiches at a time then randomly choose a customer to grab some sandwiches. The selected customer may not be the right customer, i.e., if the number of available sandwiches is less than what the customer needs, the customer needs to go back into the queue. Use monitors to write synchronization code for this problem.

In following, we provide a list of exercises related to memory management. It covers paging, multi-level paging, segmentation, and segmentation with paging. Students can try to solve exercises to evaluate their understanding.

**Exercise 1.** Provide definitions and explanation for the following questions, related to memory management.

1. Briefly explain the difference between absolute code and relocatable code.

2. Briefly describe the dynamic loading and dynamically linked libraries.

3. What are the advantages of dynamically linked libraries over statically linked libraries?

4. Explain why and how CPUs use an internal cash memory.

5. Explain how a program is transformed from a source code (e.g., main.c) to a binary executable file (e.g., main.exe). Describe in which phase libraries and header files are linked and included.

6. Explain the difference between internal and external fragmentation.

7. Can we have internal fragmentation in a memory system that uses segmentation? Justify your answer.

8. Why there is no external fragmentation in a memory system that uses paging?

9. Briefly discuss what is the concept of virtual memory.

10. Briefly explain what is the different between a segment fault and a page fault? Explain what happens in each event.

11. Briefly discuss how modern operating systems that use virtual memory allow the execution of programs that have a size larger than the available RAM (Read Only Memory).

12. Briefly describe what thrashing is. How can the operating system detect when thrashing occurs? What should the operating system do when thrashing occurs?

**Exercise 2.** Assuming a byte-addressed memory (i.e., every byte of memory has its own address), give the size of the memory for the following address sizes:

For `M=11` the size is ...

For `M=25` the size is ...

For `M=36` the size is ...

For `M=44` the size is ...

For `M=1` the size is ...

For `M=27` the size is ...

**Exercise 3.** Consider a computer in which the virtual memory consists of 8 pages of 1024 bytes each, mapped onto physical memory of 32 page frames. Then, assuming that the memory is byte-addressed, how many bits are there in a virtual address? How many bits are there in the physical address?

**Exercise 4.** Consider a 64-bit computer that uses single-level paging for memory management. It uses 64KB as page size and 8 bytes per page entry in the page table.

(a) Assuming a program that uses virtual addresses from 0...4GB, what is the size of the page table (a.k.a., flat page table)?

(b) Now consider the same computer from Question 2.4, what would be the size of the page table for a 4GB process, if the computer operates a 4-level paging scheme with page number split equally among the 4 levels (i.e., each level uses the same number of bits to code a page number)?

**Exercise 5.** Consider a 32-bit computer that has a byte-addressed memory of 1MB. It uses single-paging scheme with 4KB page size. Having the page table below, translate the following virtual addresses into physical addresses.

| Page ♯ | Base address | Validity |
|---|---|---|
| 1 | 0x6C000 | V |
| 2 | 0xB1000 | V |
| 3 | 0xC4000 | V |
| 4 | 0xA8000 | V |
| 5 | 0xE6000 | V |
| 6 | 0xD1000 | V |
| ... | ... | V |

- Virtual address `0x00004532` has physical address:

- Virtual address `0x00006EF2` has physical address:

- Virtual address `0x00001F33` has physical address:

- Virtual address `0x000031E2` has physical address:

**Exercise 6.** A computer uses segmentation with paging. A virtual address consists of six hexadecimal digits, SSPPEE, where SS is a segment number and PPEE is the offset within the segment. The segment is paged, with PP as the page number and EE as the page offset. The page size is 256 bytes. The physical address is on 16 bits. The figure below shows selected parts of the current memory contents. All numbers are hexadecimal. Segments and page tables are indexed starting at 0. Two-digit frame numbers are stored in the segment tables, in the page tables, and in STBR (Segment Table Base Register). These frame numbers are converted to main memory addresses by appending 0x00. For example, the entry 0F in a segment table means that the segment's page table is located at address 0x0F00. A * means the page is on disk.

`STBR=0x06` List of free page frames: `0x1A, 0x08, 0x21, 0x16, 0x0E`.

Segment tables:

| 0x0200 | 0x0F |
|---|---|
| | 0x13 |
| | 0x09 |
| | * |
| | 4E |

| 0x0600 | 0x09 |
|---|---|
| | 0x0A |
| | 0x4C |
| | 0x3D |
| | 0x5D |

| 0x3000 | 0x0F |
|---|---|
| | 0x09 |
| | 0x13 |
| | 0x4C |
| | * |

Page tables:

| 0x3D00 | 0x5C |
|---|---|
| | 0x1E |
| | 0x5A |
| | 59 |
| | * |

| 0x4C00 | * |
|---|---|
| | 0x3B |
| | * |
| | 0X20 |
| | * |

| 0x0A00 | * |
|---|---|
| | 0x57 |
| | * |
| | * |
| | * |

| 0x0900 | 04 |
|---|---|

| 0x0F00 | * |
|---|---|
| | 0x25 |

| 0x1300 | 0x18 |
|---|---|

| 0x4E00 | 11 |
|---|---|
| | * |

(a) The following sequence of virtual memory addresses are generated by the current process. What are the corresponding main memory addresses? In answering this question, assume that new pages are brought into main memory using the given list of free page frames (0x1A, 0x08, 0x21, 0x16, 0x0E). Write changes to the segment or page tables into the above figure.

| Virtual Address | Physical Address | Page Fault? |
|---|---|---|
| 0x0000b2 | ..... | Yes/No |
| 0x020216 | ..... | Yes/No |
| 0x02015e | ..... | Yes/No |
| 0x0202a7 | ..... | Yes/No |

(b) Following the four accesses performed in part (a), the system switches execution to another process. The value `0x02` is loaded into the STBR. What segments, if any, are shared between this new process and the previous process?

(c) Describe what happens when the new process (STBR=`0x02`) references address `0x030142`. Keep using list of free page frames from part (14.1). Write changes to the segment or page tables into the above figure. Can you determine what main-memory address results after address translation? If so, state the address. If not, state what additional information is needed.

**Exercise 7.** Using segmentation with paging, this table shows three methods of dividing up virtual addresses. For each case, calculate maximum size for a segment, a segment table, and a segment's page table. You may state your

answers in terms of bytes, kilobytes, megabytes, gigabytes, or using a format like $2^x$ byte. Also, explain your computations.

| **Bits in virtual address** | **Page size (byte)** | **Size of entires in segment table & page table** | **Max ♯ of segments segments** | **Max size of a segment** | **Max size of a segment table** | **Max size of a segment's page table** |
|---|---|---|---|---|---|---|
| 16 bits | 128 | 8 byte | 64 | | | |
| 32 bits | 1024 | 8 byte | 4096 | | | |
| 32 bits | 8192 | 8 byte | 65536 | | | |

**Exercise 8.** Find the maximum allowable page-fault rate so that effective virtual memory access time is $\leq$ 200 nanoseconds on a computer system with the following characteristics. The time to access main memory is 100 nanoseconds; this is sometimes referred to as the raw memory access time. Paging is used, with the page table stored in main memory. The address translation cache (TLB, Translation Lookaside Buffer) has a 95% hit rate. TLB access time is negligible compared to the 100 nanosecond memory access time, so:

- When address translation encounters a TLB hit, the virtual memory access time is 100 nanoseconds. (Virtual memory access time is the same as raw memory access time, because we assume that address translation time is negligible.)

- When there is a TLB miss with no page fault, the virtual memory access time is 200 nanoseconds. (Address translation must access memory to read an entry in the page table, thus adding 100 nanoseconds of overhead.) The time to handle a page fault is 8 milliseconds when a clean page is replaced, and 16 milliseconds when a dirty page is replaced. The replaced page is dirty 65% of the time; this means that 35% of the time the replaced page is clean. (A dirty page is one that has been written to since it was loaded into main memory. Replacing a dirty page takes longer because the operating system has to write the contents of the dirty page back to disk before it can reuse that page frame.)

**Exercise 9.** This program swaps the elements in the first half of array A with the elements in the second half of the array:

```
var A: array[1..1000] of integer;
temp1, temp2, i:  integer;
// Code to initialize A has been omitted.
for (i := 1 to 500)
{
     temp1 := A[i]; // This code swaps A[i] and A[500+i].
     temp2 := A[500+i];
     A[i] := temp2;
     A[500+i] := temp1;
}
```

Assume that 100 integers fit into a page, so array A occupies 10 pages. Initially all of A is on disk. How many page faults occur during program execution in cases (a) and (b)? Use Least Recently Used (LRU) page replacement. [In this problem we only consider page faults that result from accessing A: assume that there are no page faults from fetching instructions.]

(a) Five page frames of size 100 integers are allocated to array A.

(b) One page frame of size 100 integers is allocated to array A.

**Exercise 10.** Consider a 32-bit computer that uses paging as a memory management scheme. Each page table entry occupies 4 bytes.

(a) What is the smallest page size that allows the page table to fit into one page?

(b) A page size of 1KB is chosen. This means that the page table has to be stored in multiple levels. Describe how the 32 bit virtual address is divided into fields: how many fields are there, and how many bits are in each field?

**Exercise 11.** The following sequence of virtual memory references is generated when a program is executed. Each memory reference is a 12 bit number written as three hexadecimal digits.

0x019, 0x01A, 0x1E4, 0x170, 0x073, 0x30E, 0x185, 0x24B, 0x24C, 0x430, 0x458, 0x364

(a) What is the reference string, assuming a page size of 256 Bytes?

(b) Find the page fault rate for the reference string in part (a): assume that 2 frames of main memory are available to the program and the FIFO page replacement algorithm is used. Note that the page fault rate is calculated as "number of page faults" divided by "number of virtual memory references used to form the reference string".

(c) Repeat (b) using the LRU (Least Recently Used) page replacement algorithm.

(d) Repeat (b) using the optimal page replacement algorithm.

**Exercise 11.** Assume you have a reference string for a process with m frames, and initially all m frames are empty. The reference string has length p, with n distinct page numbers occurring in it. Give an upper bound and a lower bound on the number of page faults. Your bounds should hold for any page-replacement algorithm. Briefly justify your answers.

**Exercise 12.** Consider a main memory with 220 nanoseconds raw access time, and a TLB (Translation Look-aside Buffer) cache with 120 nanoseconds access time, respectively. Knowing that the TLB hit ratio is 98%, calculate the effective access time.

---

In the following, we provide some exercises related to operating system's security.

**Exercise 1.** The following code snippet is used in a gaming machine where the user, after paying 50 Dinars, is asked to input a value that is less than 10 to win some money. Find a vulnerability in this code that if exploited would allow a user to win more than 1050 Dinars. What is this vulnerability, and how can you fix it?

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main()
{
    int user_Input_Value;
    int system_Winning_Factor = rand() % 100 + 1;

    printf("Input a value that is less than 10 to win\n");
    scanf("%d", &user_Input_Value);


    if(user_Input_Value*100<=1000)
    {
        printf("Congratulation! You won %d Dinars\n", (user_Input_Value*system_Winning_Factor)+50);
    }
    else
    {
        printf("Too bad! You lost 50 Dinars\n");
    }
}
```

**Exercise 2.** The following C code is used to implement an authentication mechanism to allow authorized and trusted users to access some resources. The administrator has set up a password "ensia24" and shared the password with authorized users. Find the vulnerability that allows an unauthorized user to bypass the authentication mechanism without knowing the password and get successfully logged in. The malicious user does not know the password, but know how the program was designed (i.e., the structure). Fix the vulnerability.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void main()
{
    char password[8] = {'e','n','s','i', 'a', '2', '4', '\0'};
    char pass[8];
    char username[8];
    unsigned int miss = 0;

    printf("Enter username: ");
    gets(username);
    X:  printf("Enter password: ");
        gets(pass);
        if(strncmp(password, pass, 8) == 0) //compare two strings
        {
            printf("Successful login as %s \n", username);
            execlp("/bin/sh", "/bin/./sh", (char *) NULL);
        }
        else
        {
            printf("Login failed! \n");
            miss++;
            if(miss==3);
            else goto X;
        }
}
```

**Exercise 3.** The following code got updated to provide a better security. Find another vulnerability that would allow a malicious user to crack this authentication mechanism and possibly get a successful login. The malicious may exploit the vulnerability and use tools like `John`.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void main()
{
    char username[8];
    unsigned int miss = 0;
    char pass[8];
    char password[8] = {'e','n','s','i', 'a', '2', '4', '\0'};

    printf("Enter username: ");
    gets(username);
    X:  printf("Enter password: ");
        gets(pass);
        if(strncmp(password, pass, 8) == 0) //compare two strings
        {
            printf("Successful login as %s \n", username);
            execlp("/bin/sh", "/bin/./sh", (char *) NULL);
        }
        else
        {
            printf("Login failed! \n");
            miss++;
            if(miss==3);
            else goto X;
        }
  printf("Exit ... Goodbye %d \n", miss);
}
```