SWE Tracks' renovation

In view of SWEBOK V3.0

Introduction to Operating Systems

Prepared by:

Ahmed Loutfy

Course Duration and Evaluation

- Duration: 6 hours
  - 2 Lectures (6 hours)

- Evaluation Criteria:
  - A comprehensive exam after finishing all the main conceptual courses

Course Outlines

Introduction to Computer System and Operating System

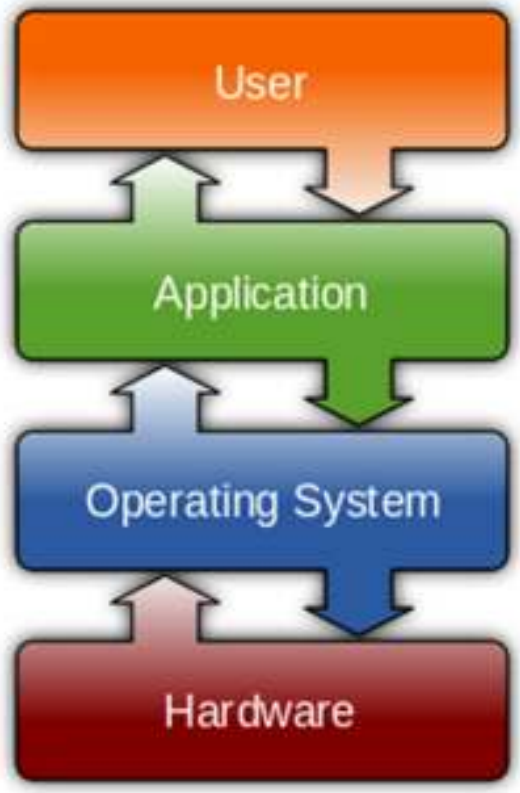
Session Content

- Computer System Components
- Von Neumann Architecture
- Computer Operating System
- Why We Need an OS?
  - Complex and Multiprocessor Systems
  - Multiuser Systems
- Operating System Components
- Why Is It Important to Know About the OS?
- Responsibilities of an OS

Course Outlines

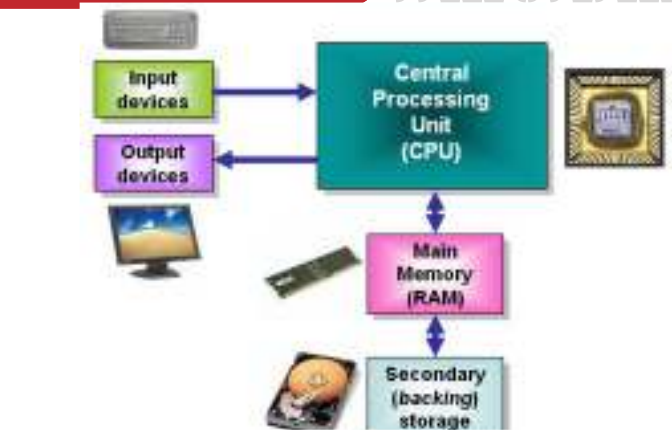
1.1 Computer System Components

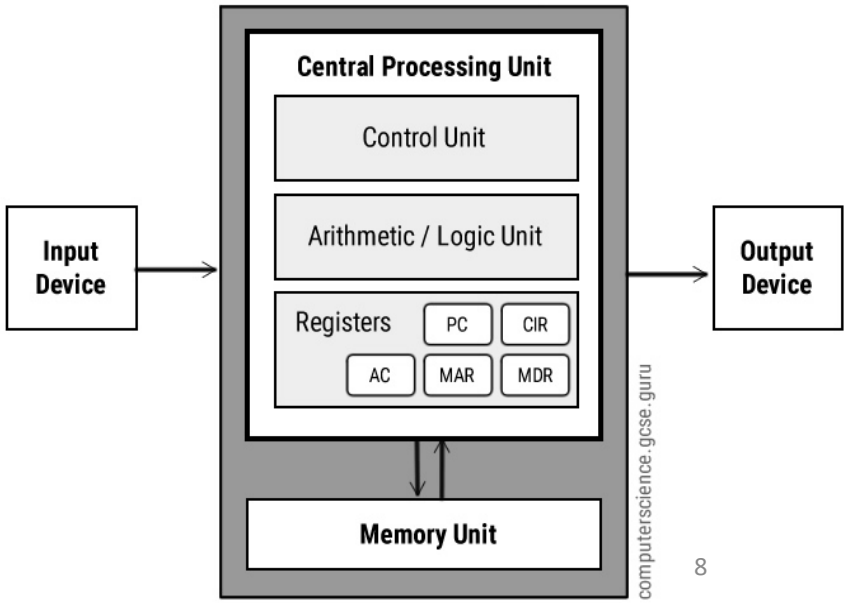
- ***Hardware:*** provides basic computing resources (CPU, memory, I/O devices)
- ***Operating system:*** controls and coordinates the use of the hardware among the various application programs for the various users
- ***Applications programs:*** define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs)
- ***Users:*** people, machines, other computers



1.2 Von Neumann Architecture

- Von Neumann architecture was first published by John von Neumann in 1945.
- His computer architecture design consists of a Control Unit (CU), Arithmetic and Logic Unit (ALU), Memory Unit, Registers and Inputs/Outputs.
- Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.





1.2.1 Central Processing Unit (CPU)

- The Central Processing Unit (CPU) is the electronic circuit responsible for executing the instructions of a computer program.
- It is sometimes referred to as the microprocessor or processor.
- The CPU contains the ALU, CU and a variety of registers.
- **1.2.1.1 Registers**
  - Registers are high speed storage areas in the CPU. All data must be stored in a register before it can be processed.

MAR	**Memory Address Register**	**Holds the memory location of data that needs to be accessed**
MDR	**Memory Data Register**	**Holds data that is being transferred to or from memory**
AC	**Accumulator**	**Where intermediate arithmetic and logic results are stored**
PC	**Program Counter**	**Contains the address of the next instruction to be executed**
CIR	**Current Instruction Register**	**Contains the current instruction during processing**

1.2.1 Central Processing Unit (CPU)

- **1.2.1.2 Arithmetic and Logic Unit (ALU)**
  - The ALU allows arithmetic (add, subtract etc) and logic (AND, OR, NOT etc) operations to be carried out.
- **1.2.1.3 Control Unit (CU)**
  - The control unit controls the operation of the computer's ALU, memory and input/output devices, telling them how to respond to the program instructions it has just read and interpreted from the memory unit.
  - The control unit also provides the timing and control signals required by other computer components.

1.2.3 Buses

- Buses are the means by which data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory.
- A standard CPU system bus is comprised of a control bus, data bus and address bus.

Address Bus	**Carries the addresses of data (but not the data) between the processor and memory**
Data Bus	**Carries data between the processor, the memory unit and the input/output devices**
Control Bus	**Carries control signals/commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer**

1.2.4 Memory Unit

- The memory unit consists of RAM (Random Access Memory) and ROM (Read Only Memory), sometimes referred to as primary or main memory .
- Unlike a hard drive (secondary memory), this memory is fast and also directly accessible by the CPU.
- RAM is split into partitions (bytes). Each partition consists of an address and its contents (both in binary form).
- The address will uniquely identify every location (byte) in the memory.
- Loading data from permanent memory (secondary storage or hard drive), into the faster and directly accessible temporary memory (RAM), allows the CPU to operate much quicker.

1.2.5 Input Devices

- Which are peripherals used to provide data and control signals to a computer.
- Input devices allow us to enter raw data for processing. For Example:
  - Keyboard (default input device)
  - Microphone
  - Scanner (2D and 3D)
  - Mouse
  - Trackball
  - Touchpad
  - Barcode and QR Code readers
  - Digital Camera

1.2.6 Output Devices

- Which are pieces of computer hardware used to communicate the results of data processing performed by a computer.
- The objective of output devices is to turn computer information into a human friendly/readable form. For Example:
  - Screen (Monitor or Console) (default output device) – LED or LCD
  - Data Projectors
  - Speaker and Headphones
  - Printer (2D and 3D) – inkjet or laser
  - Plotter (wide format printer)
  - Cutter (2D or 3D)

1.2.7 Input/Output Devices

- Some devices work as Input and Output devices like:
  - Touch Screen
  - Network
  - Most of I/O Ports (Both of serial and parallel)
  - New types of ports, like USB and Type-C ports
- All of the secondary storages used as I/O devices for permeant storage, like:
  - Hard disk
  - Floppy disk
  - Flash memory
  - CD and DVD

1.3 Computer Operating System

- When a computer turns on, the processor will execute the instructions that are presented to it; generally, the first code that runs is for the boot flow, called ***bootstrap***, which is a framework stored in **ROM** contains some instructions called basic input output instructions (**BIOS**).
- For a computer that is used for general purposes and after it has booted up, there may be a variety of applications that need to be run on it simultaneously.
- Additionally, there could be a wide range of devices that could be connected to the computer (not part of the main system, for instance).
- All these need to be abstracted and handled efficiently and seamlessly. The user expects the system to "just work." The ***operating system*** facilitates all of this and more.

1.3.1 What is an Operating System?

- The general definition of any system is: *some components integrated together for perform a desired task*.
- So, the computer operating system consists of components integrated together for operating the computer system (HW &SW).
- The definition of the operating system is:

An operating system, commonly referred to as the OS, is a program that controls the execution of other programs running on the system. It acts as a facilitator and intermediate layer between the different software components and the computer hardware

- When any operating system is built, it focuses on three main objectives:
  - *Efficiency* of the OS in terms of responsiveness, fluidity, and so on
  - *Ease of usability* to the user in terms of making it convenient
  - *Ability to* abstract and *extend* to new *devices and software*

1.3.1 What is an Operating System?

- Most OSs typically have at least two main pieces:
  - There is a core part that handles the complex, low-level functionalities and is typically referred to as the ***kernel*** *and* must be running at all times – resident in the main memory.
  - There are generally some *libraries*, *applications*, and *tools* that are shipped with the OS. For example, there could be browsers, custom features, frameworks, and OS-native applications that are bundled together.
- list of operating systems that are commonly prevalent:
  - Microsoft Windows
  - GNU/Linux-based OS
  - macOS (used for Apple's computers and client models)
  - iOS (used for Apple's smartphone/tablet models)
  - Android
- All of these operating systems have different generations, versions, and upgrades.

1.3.2 OS Categories

- The OSs can be categorized based on the different methods in use. The two most common methodologies are by the ***usage type*** and the ***design/supported features*** of the OS.S

1.3.2.1 OS Categories – Usage Types

- Based on this, there are five main categories:
  1. ***Batch***: For usages where a sequence of steps needs to be executed repeatedly without any human intervention. These classes are called batch OSs. (Old Mainframe and DOS)
  2. ***Time Sharing***: For systems where many users (or many applications) access common hardware, there could be a need to timeshare the limited resources. The OSs in such cases are categorized as time-sharing OSs. (Windows, Mac, and Linux)
  3. ***Parallel – Distributed (over tightly coupled systems)***: For hardware that is distributed physically and a single OS needs to coordinate their access, we call these systems distributed OSs. (AIX for IBM RS/6000 and Solaris for workstations)
  4. ***Network (loosly coupled)***: Another usage model, similar to the distributed scenario, is when the systems are connected over a network protocol, like IP (Internet Protocol), and therefore referred to as network OSs. (Windows server 2008, Novell Netware)
  5. ***Real Time***: In some cases, we need fine-grained time precision in execution and responsiveness. We call these systems real-time OSs.

1.3.2.2 OS Categories – Designed and Supported Features

- Based on this, there are three main categories:
  1. ***Monolithic***: In this case, the entire OS is running in a high-privilege kernel space and acts as the supervisor for all other programs to run. Common monolithic OSs include many of the UNIX flavors.
  2. ***Modular***: In some OSs, a few parts of the OS are implemented as so-called plug-and-play modules that can be updated independent of the OS kernel. Many modern OSs follow this methodology, such as Microsoft Windows, Linux flavors, and macOS.
  3. ***Micro-service based***: More modern OSs are emerging and leverage the concept of micro-services where many of the previously monolithic OS features may be broken down into smaller parts that run in either the kernel or user mode. The micro-service approach helps in assigning the right responsibility of the components and easier error tracking and maintenance. Some versions of Red Hat OS support micro-services natively.

1.4 Why We Need an OS?

- As a global view we need the OS for perform the following tasks:
  - **Run** and **facilitate** different applications running on the system.
  - **Manage conflicts** among different applications.
  - In practice, there are many **common features** that may be needed by your programs including, for example, security, which would have services like encryption, authentication, and authorization
- The purpose of the operating system is to ensure that it abstracts the HW and facilitates the seamless execution of our applications using the system.

- Now, we will take a more detailed look at the ***different complexities*** on such systems and how the OS handles them.

1.4.1 Complex and Multiprocessor Systems

- Many modern computing architectures support microprocessors with multiple CPU cores.
- When all cores provide the same or identical capabilities, they are called as ***homogeneous platforms***.
- There could also be systems that provide different capabilities on different CPU cores. These are called ***heterogeneous platforms***.
- There are also additional execution engines such as Graphics Processing Units ***(GPUs)***, which accelerate graphics and 3D processing and display
- An operating system supporting such a platform will need to ensure ***efficient scheduling*** of the different programs on the different execution engines (cores) available on the system.
- Similarly, there could be differences in the hardware devices on the platform and their capabilities such as the type of display used, peripherals connected, storage/memory used, sensors available, and so on.
- Hence, the OS would also be required to abstract the differences in the hardware configurations to the applications.

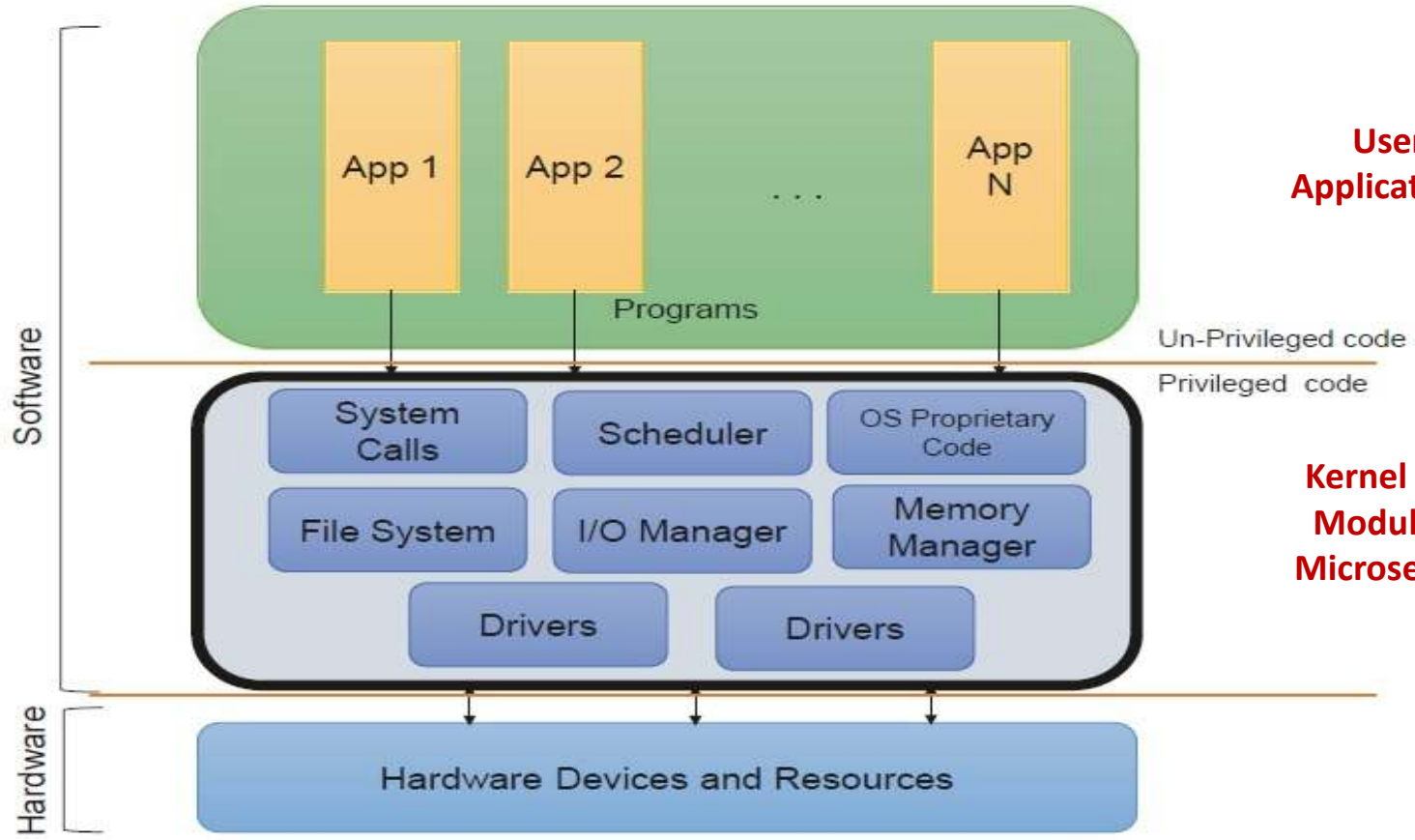
1.4.2 Multitasking and Multifunction Software

- In general, there could be ***many applications*** that may need to be running on the system ***at the same time***.
- These could include applications that the ***user initiated***, so-called "***foreground***" applications, and applications that the ***OS has initiated*** in the "***background***" for the effective functionality of the system.
- It is the OS that ensures the streamlined execution of these applications.

1.4.3 Multiuser Systems

- Often, there could be more than one user of a system such as an ***administrator*** and ***multiple*** *other users* with ***different*** levels of ***access permission*** who may want to utilize the system.
- It is important to ***streamline execution*** for each of these users so that they do not find any perceived *delay of their* ***requests***.
- At the same time, there need to be **controls** in place to manage **privacy and security** between users. The OS facilitates and manages these capabilities as well.
- It is the role of the operating system to handle these complexities in a **consistent**, **safe**, and **performant** manner. Most general-purpose OSs in use today, such as Windows, Linux, macOS, and so on, provide and handle most of the preceding complexities.

1.5 Operating System Components



User Applications

Kernel or OS Modules or Microservices

1.5 Operating System Components

- As we can see in the above slide, it supports multiple different hardware, supports co-existence of multiple applications, and abstracts the complexities. The OS exposes different levels of abstractions for applications and drivers to work together.
- Typically, there are **APIs** (application programming interfaces) that are exposed to access system resources. These APIs are then **used by programs** to **request** for communicating to the **hardware**.
- While the communication happens, there could be **requests from multiple programs and users at the same time**. The OS streamlines these requests using efficient scheduling algorithms and through management of I/Os and handling conflicts.

1.6 Why Is It Important to Know About the OS?

- Software ***developers must have a good understanding of the environment, the OS, that their code is running in***, or they won't be able to achieve the things they want with their program.
- As you, a software developer, go through the stages of development, it is important for you to keep in mind the OS interfaces and functionality as this will impact the software being developed.
- For Example:
  - the choice of **language** and needed runtime features may be OS dependent.
  - the choice of inter-process communication **(IPC)** protocols used for messaging between applications will depend on the OS offerings
- During development and **debug**, there could be usages where the developer may need to understand and interact with the OS. For example, debugging a **slowly performing or nonresponsive** application may require some understanding of how the OS performs input/output operations.

1.6 Why Is It Important to Know About the OS?

- some questions that may come up during the debug:
  - Are you accessing the file system too often and writing repeatedly to the disk?
  - Is there a garbage collector in place by the software framework/SDK?
  - Is the application holding physical memory information for too long?
  - Is the application frequently creating and swapping pages in memory? What it the average commit size and page swap rate?
  - Is there any other system event such as power event, upgrades, or virus scanning that could have affected performance?
  - Is there an impact on the application based on the scheduling policy, application priority, and utilization levels?
- If the application needs to interface with a custom device, it will most likely need to interface some low-level functionality provided by the OS using the OS-provided API for communication.
- As a software developer, it may be required to understand these APIs and leverage the OS capabilities. There could also be a need to follow certain standard protocols provided by the OS for authenticating a given user of your application to grant permissions and access.

1.7 Responsibilities of an OS

- The OS needs to be able to abstract the complexities of the underlying hardware, support multiple users, and facilitate execution of multiple applications at the same time. The following table describe the Requirements and Solutions

Requirements	Solution
Applications require time on the **CPU** to execute their instructions.	The OS shall implement and abstract this using suitable **scheduling** algorithms.
Applications require access to system **memory** for variable storage and to perform calculations based on values in memory.	The OS shall implement **memory management** and provide APIs for applications to utilize this memory.
Each software may need to access different **devices** on the platform.	The OS may provide APIs for device and I/O management and interfaces through which these devices can be communicated.

1.7 Responsibilities of an OS

Requirements	Solution
There may be a need for the user or applications to save and read back contents from the **storage**.	Most OSs have a **directory and file system** that handles the storage and retrieval of contents on the disk.
It is important to perform all of the core operations listed in the preceding **securely** and efficiently.	Most OSs have a **security subsystem** that meets specific security requirements, virtualizations, and controls and balances.
Ease of access and usability of the system.	The OS may also have an additional GUI (graphical user interface) in place to make it easy to use, access, and work with the system.

1.7 Responsibilities of an OS

- To summarize, the OS performs different functions and handles multiple responsibilities for software to co-exist, streamlining access to resources, and enabling users to perform actions. They are broadly classified into the following functional areas:
  - Scheduling
  - Memory management
  - I/O and resource management
  - Access and protection
  - File systems
  - User interface/shell

Processes and Scheduling

Content

- Introduction to Scheduling
- Process Concept
- Process Contents
- Process States
- Process Control Block (PCB)
- Context Switching
- Scheduling

2.1 Introduction to Scheduling

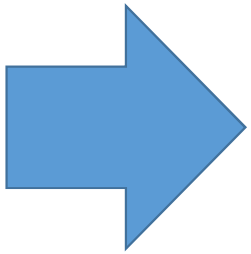
- One of the primary functionalities of the OS would be to provide the ability to run multiple, concurrent applications on the system and efficiently manage their access to system resources.
- As many programs try to run in parallel, there may be competing and conflicting requests to access hardware resources such as CPU, memory, and other devices.
- The operating system streamlines these requests and orchestrates the execution at runtime by scheduling the execution and subsequent requests to avoid conflicts.
- Before we go into the details of scheduling responsibilities and algorithms, it is important to know some background about the basic concepts of ***program execution***, ***specifically processes*** and ***threads***.

2.2 Program and Process Concept

- When a software developer builds a solution, the set of capabilities it provides is usually static and embedded in the form of processed code that is built for the OS. This is typically referred to as the program.
- When the program gets triggered to run, the OS assigns a process ID and other metrics for tracking.
- At the highest level, an executing program is tracked as a process in the OS.
- Note that in the context of different operating systems, jobs and processes may be used interchangeably. However, ***process refer to a program in execution***.

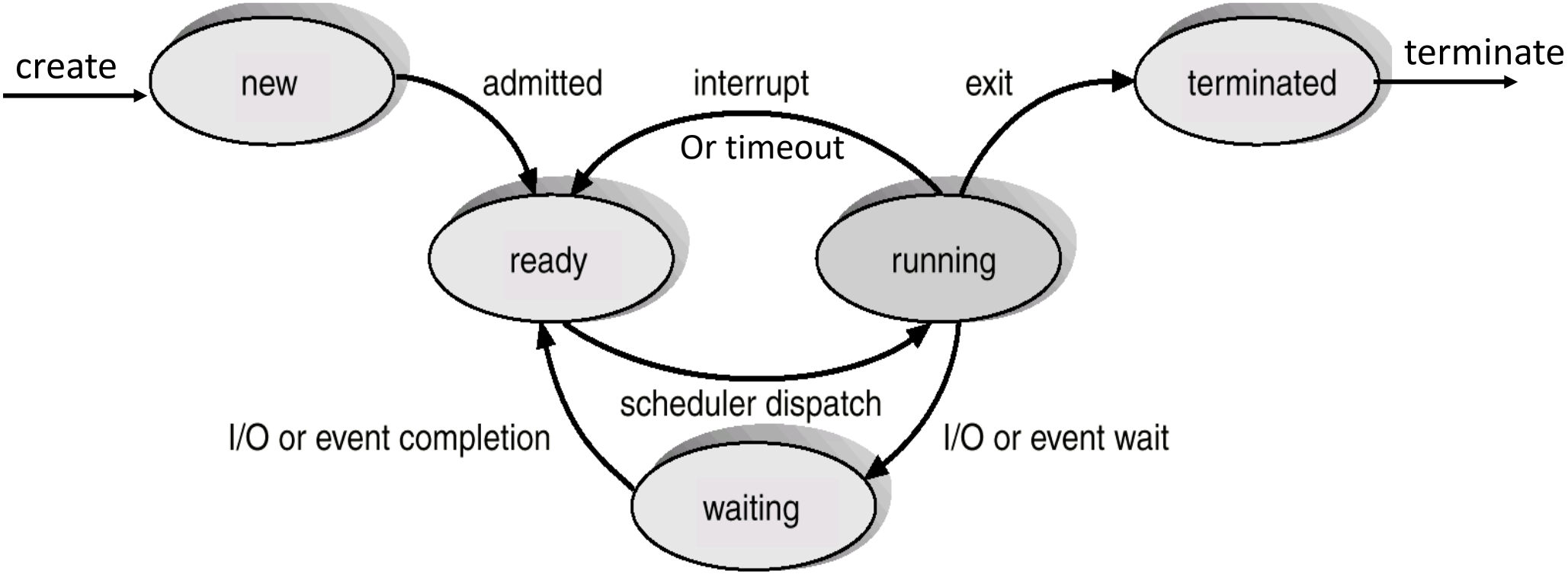
2.3 Process Contents

- Text section
  - Program instructions
- Program counter
  - Next instruction address
- Stack Section
  - Local variables
  - Return addresses
  - Method parameters
- Heap section
  - Global and Static Variables
  - Dynamic Allocation (at run-time)



- Text section
  - Program instructions
- Data Section
  - Global and Static Variables
- Stack Section
  - Local variables
  - Return addresses
  - Method parameters
  - **PC**: Next instruction address
- Heap section
  - Dynamic Allocation (at run-time)

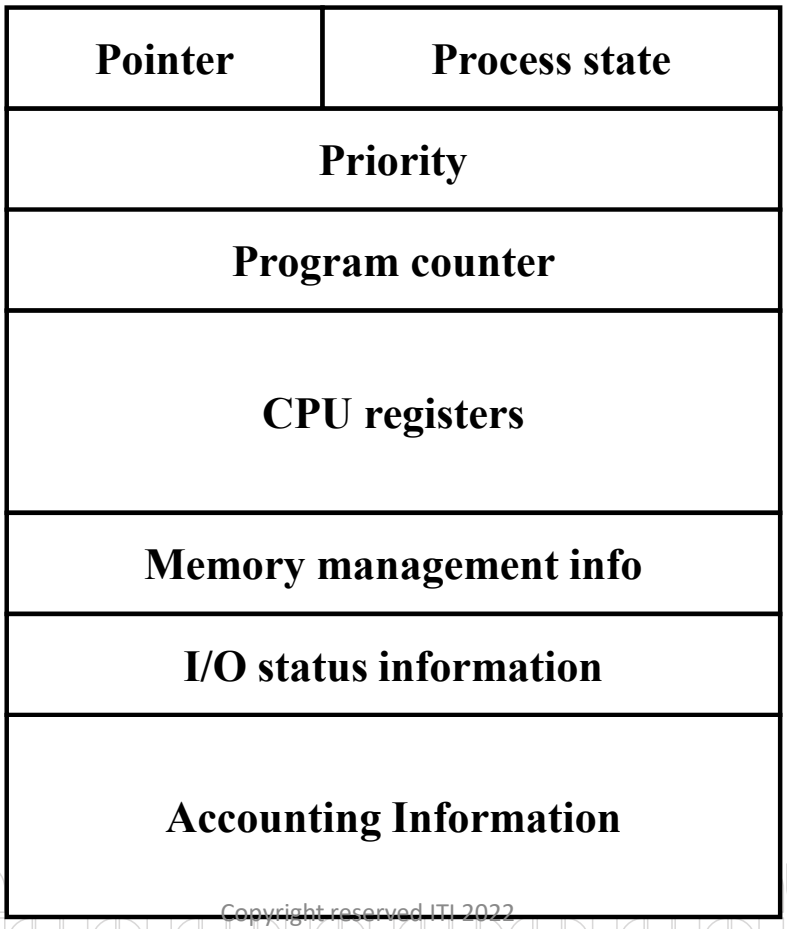
2.4 Process States



2.4 Process States

- When a program gets triggered for execution, typically say using a double click of the EXE (or using a CreateProcess() API in Windows), a new process is created.
- process typically supports multiple states of readiness in its lifecycle:
  - **New**: The process is being created.
  - **Running**: Instructions are being executed.
  - **Waiting**: The process is waiting for some event to occur.
  - **Ready**: The process is waiting to be assigned to a processor.
  - **Terminated or Exit**: The process has finished execution.
- There could be more than one CPU core on the system and hence the OS could schedule on any of the available cores.
- In order to avoid switching of context between CPU cores every time, the OS tries to limit such frequent transitions.
- The OS monitors and manages the transition of these states seamlessly and maintains the states of all such processes running on the system.

2.5 Process Control Block (PCB)



2.5 Process Control Block (PCB)

- The **process ID** is a unique identifier for the instance of the process that is to be created or currently running.
- The **process state** determines the current state of the process, described in the preceding section.
- The **pointer** could refer to the hierarchy of processes (e.g., if there was a parent process that triggered this process).
- The **priority** refers to the priority level (e.g., high, medium, low, critical, real time, etc.) that the OS may need to use to determine the scheduling.
- **Affinity and CPU register** details include if there is a need to run a process on a specific core. It may also hold other register and memory details that are needed to execute the process.

2.5 Process Control Block (PCB)

- The **program counter** usually refers to the next instruction that needs to be run.
- The **I/O status information,** like which devices assigned, limits, and so on that is used to monitor each process is also included in the structure.
- The **accounting information** such as paging requirements from memory, timers, how many time unit remaining to finish, … etc

- There could be some modifications to how the PCB looks on different OSs. However, most of the preceding are commonly represented in the PCB.

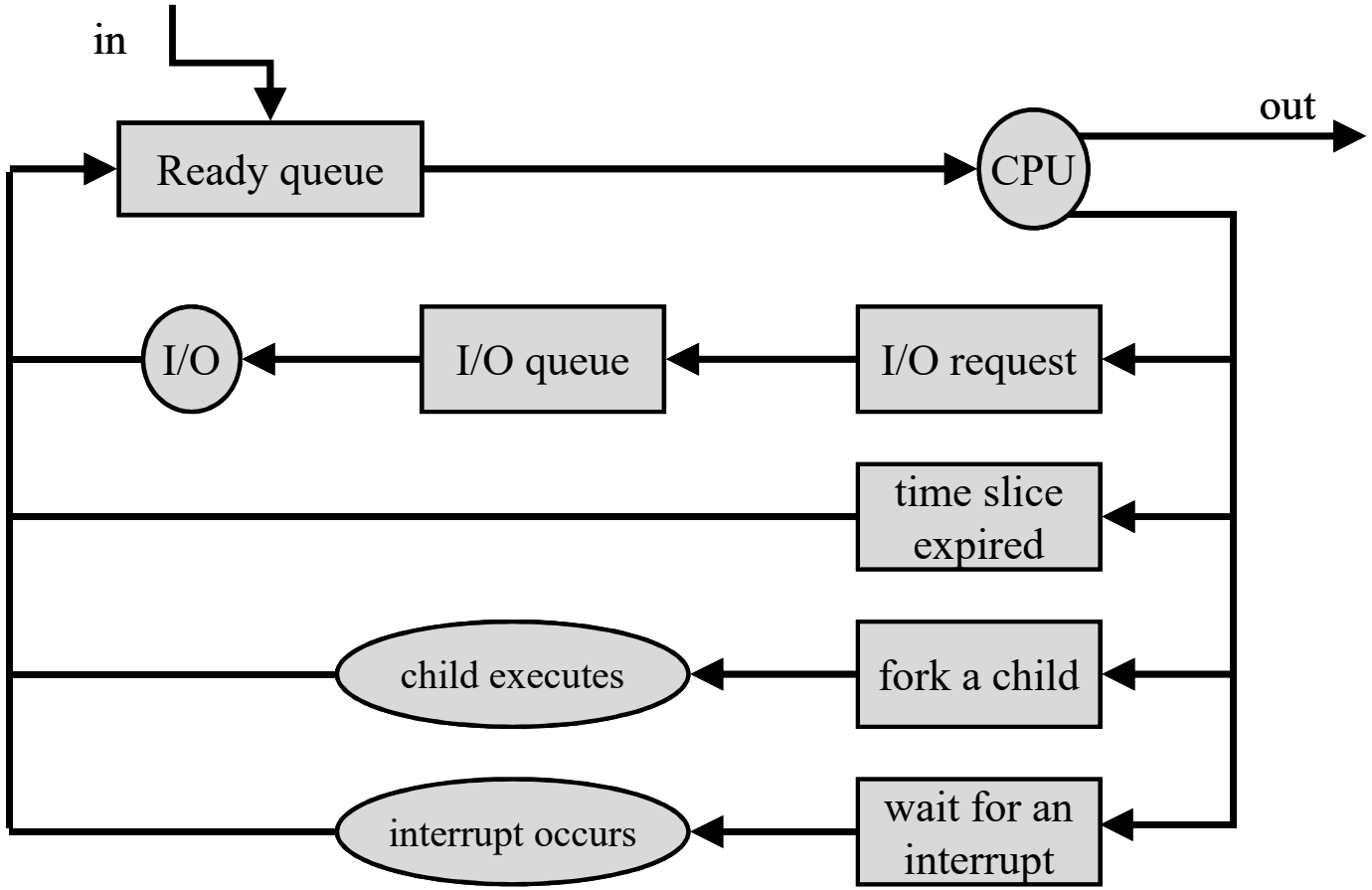
2.6 Context Switching

- The operating system may need to swap the currently executing process with another process to allow other applications to run, it does so with the help of context switching.
- When a process is executing on the CPU, the process context is determined by the program counter (instruction currently run), the processor status, register states, and various other metrics.
- When the OS needs to swap a currently executing process with another process, it must do the following steps:
  1. Pause the currently executing process and save the context.
  2. Switch to the new process.
  3. When starting a new process, the OS must set the context appropriately for that process.
- This ensures that the process executes exactly from where it was swapped.

2.7 Scheduling

- The most frequent process states are the Ready, Waiting, and Running states. The operating system will receive requests to run multiple processes at the same time and may need to streamline the execution. It uses process scheduling queues to perform this:
  1. **Ready Queue**: When a new process is created, it transitions from New to the Ready state. It enters this queue indicating that it is ready to be scheduled.
  2. **Waiting Queue**: When a process gets blocked by a dependent I/O or device or needs to be suspended temporarily, it moves to the Blocked state since it is waiting for a resource. At this point, the OS pushes such process to the Waiting queue.
  3. **Job queue** that maintains all the processes in the system at any point in time. This is usually needed for bookkeeping purposes.

2.7.1 Scheduling Queues



2.7.2 Scheduling Criteria

- Some of the typical metrics that the OS may use to determine scheduling priorities are listed in the following:
  - **CPU Utilization and Execution Runtime**: The total amount of time the process is making use of the CPU excluding NOP (no-operation) idle cycles.
  - **Volume/Execution Throughput**: Some OSs may need to support certain execution rates for a given duration.
  - **Responsiveness**: The time taken for completion of a process and the average time spent in different queues.
  - **Resource Waiting Time**: The average time taken on external I/Os on the system.

- ***Note*** *Most OSs try to ensure there is fairness and liveness in scheduling. There are various scheduling algorithms like First Come, First Serve (FCFS), Shortest Job First (SJF), Shortest Remaining Time First (SRT F), Round-Robin, Static/Dynamic Priority, and so on that the OS uses for scheduling of processes.*

2.8 Thread Concepts

- A thread is nothing more than a lightweight process.
- When a process gets executed, it could create one or more threads internally that can be executed on the processor. These threads have their own PCB; program counter, context, and register information, similar to how the process is managed.
- Threads help in performing parallelism within the same process.
- Examples:
  - Chatting program: the sending and receiving operations are independent, using threads
  - Strategic games: there are many actions happened at the same time independently, using threads
- *The OS may employ different types of threads, depending on whether they are run from an application. For instance, an application may leverage* ***user-mode*** *threads, and a kernel driver may leverage* ***kernel-mode*** *threads. The OS also handles switching from user-mode threads to kernel-mode threads as required by a process.*

Memory Management

Content

- Need of Memory Management
- Address Binding
  - Logical Address Vs. Physical Address
- Inter-process Communication
  - Shared Memory Method
  - Message Passing Method

Course Outlines

3.1 Need of Memory Management

- In systems with multiple programs running in parallel, there could be many processes in memory at the same time, and each process may have specific memory needs.
- Processes may need memory for various reasons:
  - **First**, the executable itself may need to be loaded into memory for execution. This is usually the *instructions or the code* that needs to be run.
  - The **second** item would be the data part of the executable. These could be *hardcoded* strings, text, and variables that are referenced by the process.
  - The **third** type of memory requirement could arise from runtime requests for memory. These could be needed from the *stack/heap* for the program to perform its execution.
- The OS and the kernel components may also need to be loaded in memory. Additionally, there may be a specific portion of memory needed for specific devices (Ex: printer spooling).

3.2 Address Binding

- The program each time executed it is loaded in a different memory location (according to the available spaces at time loading).
- And so, when the process in waiting state for I/O operation it may be swapped out from main memory to virtual memory (part from secondary storage), and when the waiting state changed to ready, the process must swapped in to main memory which – almost cases- another location in the main memory.
- That means the addresses of the variables which used in the program, may be changed many times at the runtime!!!
- To solve this problem, the common solution is to *map* the program's *compiled* addresses to the actual address in *physical* memory.

3.2.1 Logical Address Vs. Physical Address

- A program will have variables, instructions, and references that are included as part of the source code. The references to these are usually referred to as the symbolic addresses. When the same program gets compiled, the compiler translates these addresses into relative addresses (Logical Address).
- This is important for the OS to then load the program in memory with a given base address and then use the relative address from that base to refer to different parts of the program.
- In general, there is not enough physical memory to host all programs at the same time. This leads to the concept of virtual memory that can be mapped to physical memory.
- The ***memory management unit*** is responsible for translating virtual addresses or logical addresses to physical addresses.

3.3 Inter-process Communication

- It is often desirable to have processes communicate with each other to coordinate work, for instance. In such cases, the OS provides one or more mechanisms to enable such process-to-process communication.
- These mechanisms are broadly classified as inter-process communication (IPC). The two common ways are explained in the following, which involve:
  - Shared memory and
  - Message passing.

3.3.1 Shared Memory Method

- When two or more processes need to communicate with each other, they may create a shared memory area that is accessible by both processes.
- Then, one of the processes may act as the *producer of data*, while the other could act as the *consumer of data*.
- The ***memory acts as the communication buffer*** between these two processes.
- This is a very common mechanism to communicate between processes.
- Note: this method need a way of management when the two processes need to save in the shared memory at the same time, it is called ***Synchronization***

3.3.2 Message Passing Method

- The other method is called message passing where the two processes have a predefined communication link that could be a ***file system***, ***socket***, ***named pipe***, and so on and a protocol-based messaging mechanism that they use to communicate.
- Typically, the first step would be to establish the communication channel itself.
- For example, in the case of a ***TCP/IP communication***, one of the processes could act as the ***server*** waiting on a specific port. The other process could register as a ***client*** and connect to that port. The next step could involve sharing of messages between the client and server using predefined protocols leveraging Send and Receive commands. The processes must agree on the communication parameters and flow for this to be successful.
- ***Note: Some OSs have system calls (APIs) for sending and receiving the data among processes.***

I/O Management

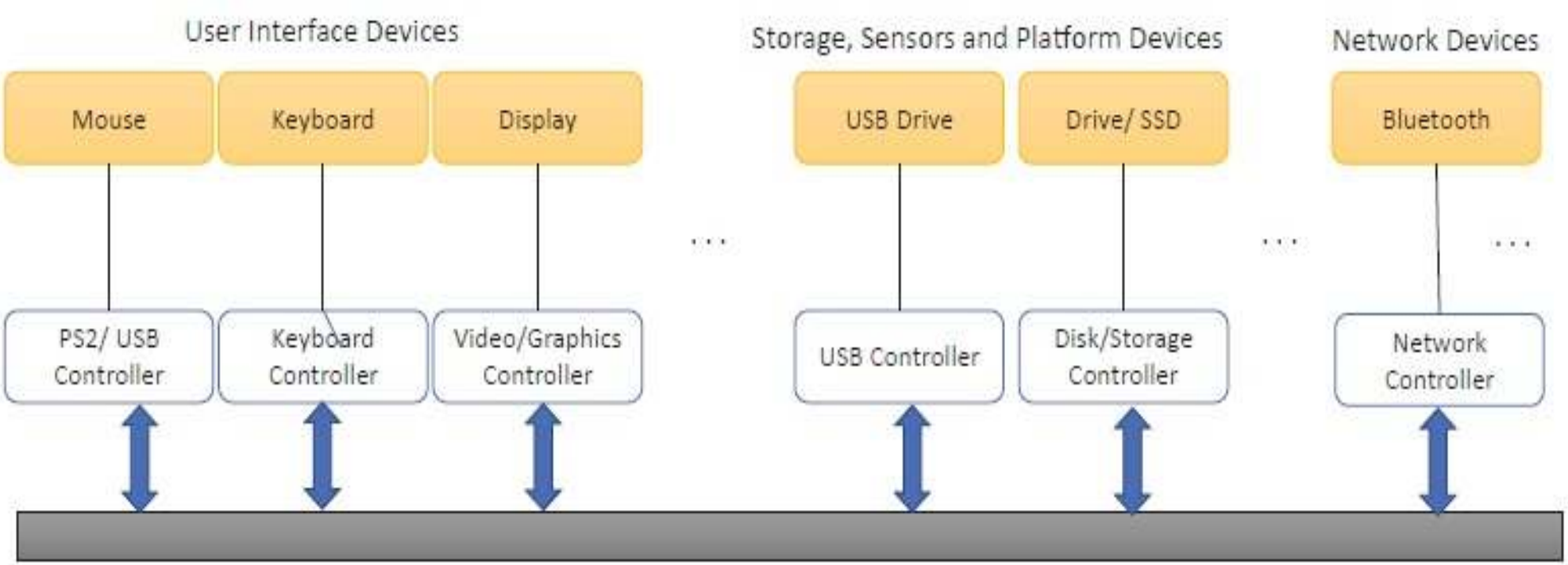
Content

- Need of I/O Management
- I/O Subsystem
- I/O Devices Categories:
  - Block Devices
  - Character Devices
- I/O Protocols Categories:
  - Special Instructions I/O
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)
- Interrupt Handling Mechanisms
- Synchronous Vs. Asynchronous I/O
- Synchronization and Critical Sections
  - Mutex
  - Semaphore
- Deadlocks

4.1 Need for I/O Management

- As part of the system, there could be multiple devices that are connected and perform different input-output functions.
- These I/O devices could be used for human interaction such as display panel, touch panels, keyboard, mouse, and track pads, to name a few.
- Another I/O devices could be to connect the system to storage devices, sensors, and so on. There could also be I/O devices for networking needs that implement certain parts of the networking stack. These could be Wi-Fi, Ethernet, and Bluetooth devices and so on.
- They vary from one to another in the form of protocols they use to communicate such as the data format, speed at which they operate, error reporting mechanisms, and more.
- The OS presents a unified I/O system that abstracts the complexity from applications. The OS handles this by establishing protocols and interfaces with each I/O controller. However, *the I/O subsystem usually forms the complex part of the operating system due to the dynamics and the wide variety of I/Os involved*.

4.1 Need for I/O Management



4.2 I/O Subsystem

- Input/output devices that are connected to the computer are called ***peripheral*** devices.
- It is communicated with the system through the busses: ***Data bus***: to transfer data, ***Address bus***: used to specify address locations, and ***Control bus***: to control a device.
- There ***could be different buses or device protocols*** that an operating system may support. The most common protocols include:
  - Peripheral Component Interconnect Express (PCIe) protocol,
  - Inter-Integrated Circuit (I2C), and
  - Advanced Configuration and Power Interface (ACPI)
- ***A device*** can be connected over ***one or more*** of these ***interfaces***.
- Consider the need to send a request to read the temperature of a specific device that is connected via ACPI. In this case, the operating system sends a request to the ACPI subsystem, targeting the device that handles the request and returns the data. This is then passed back to the application.

4.2 I/O Subsystem

- Typically, there is a software component in kernel mode called as the "***device driver***" that handles all interfaces with a device.
- It helps with communicating between the device and the OS and abstracts the device specifics.
- Similarly, there could be a driver at the bus level usually referred to as ***the bus driver***. (Ex: USB Bus driver)
- Most OSs include an inbox driver that implements the bus driver.
- There is usually a driver for each controller and each device.

4.3 I/O Devices Categories

- The I/O devices can be broadly divided into two categories called ***block*** and ***character*** devices.
- Usually, most devices would have a command and data location and a protocol that the device firmware and the driver understand.
- The ***driver*** would fill the required data and issue a command.
- The device ***firmware*** would respond back to the command and return a code that is utilized by the driver.
- The protocol, size, and format could differ from one device to another.

4.3.1 Block Devices

- These are devices with which the I/O device controller communicates by sending blocks of data.
- A block is referred to as a group of bytes that are referred together for Read/Write purposes.
- Example: ***flash memory***, ***digital camera***
- The device driver would access by specifying the size of Read/Writes which is varying from device to another.

4.3.2 Character Devices

- Another class of devices are character devices, the subtle difference is that the communication happens by sending and receiving single characters, which is usually a byte or an octet.
- Many serial port devices like ***keyboards***, some ***sensor devices***, and microcontrollers follow this mechanism.

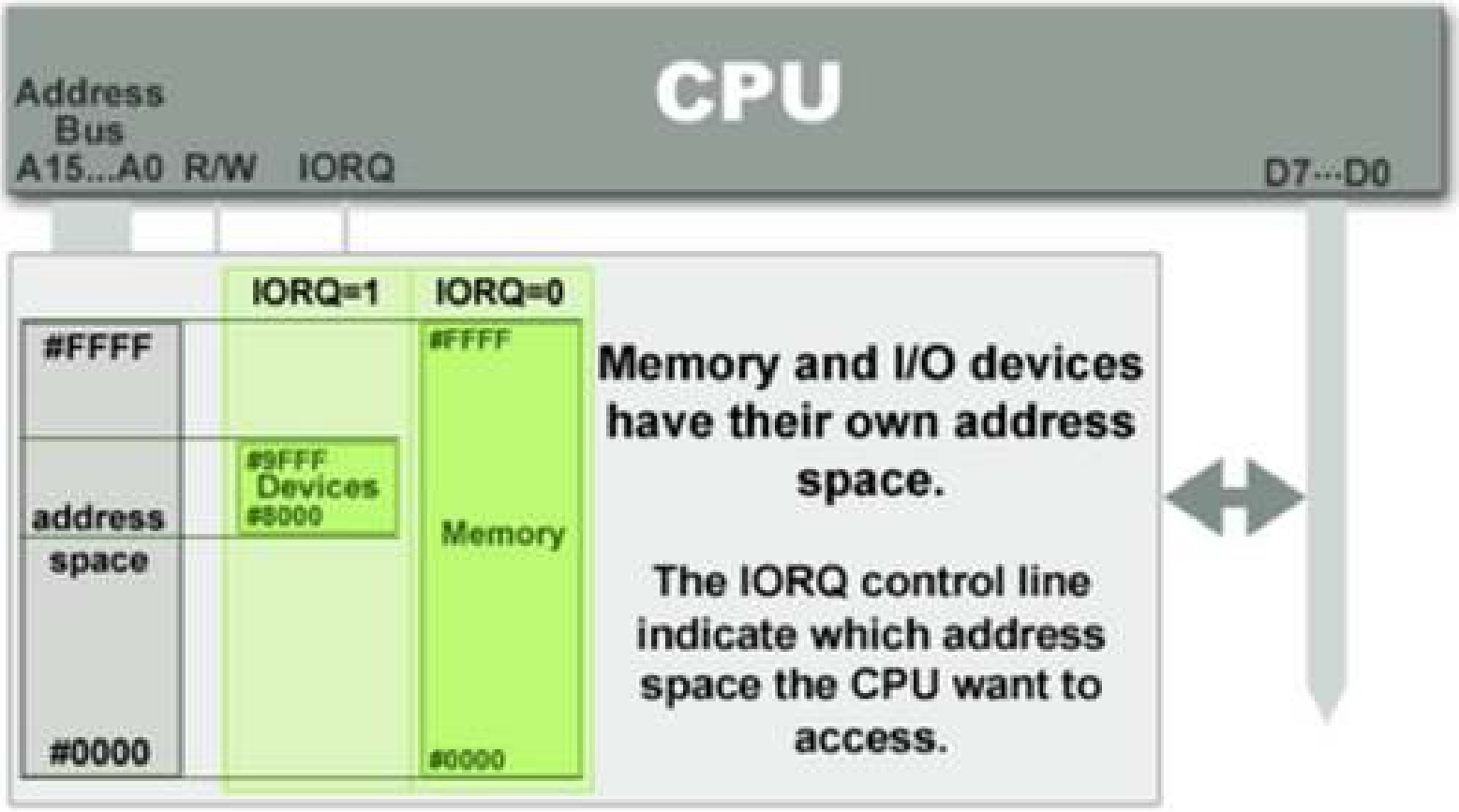
4.4 I/O Protocols Categories

- The protocols used by the different devices (block devices or character devices) could vary from one to another. There are three main categories of I/O protocols that are used:
  - Special Instruction I/O
  - Memory-Mapped I/O
  - Direct Memory Access (DMA)

4.4.1 Special Instruction I/O

- Special Instruction I/O or Port-mapped IO (PMIO)
- There could be specific CPU instructions that are custom developed for communicating with and controlling the I/O devices.
- Each device has a unique I/O address
- For example, there could be a CPU-specific protocol to communicate with the embedded controller.
- This may be needed for faster and efficient communication.
- However, such type of I/Os are special and smaller in number.

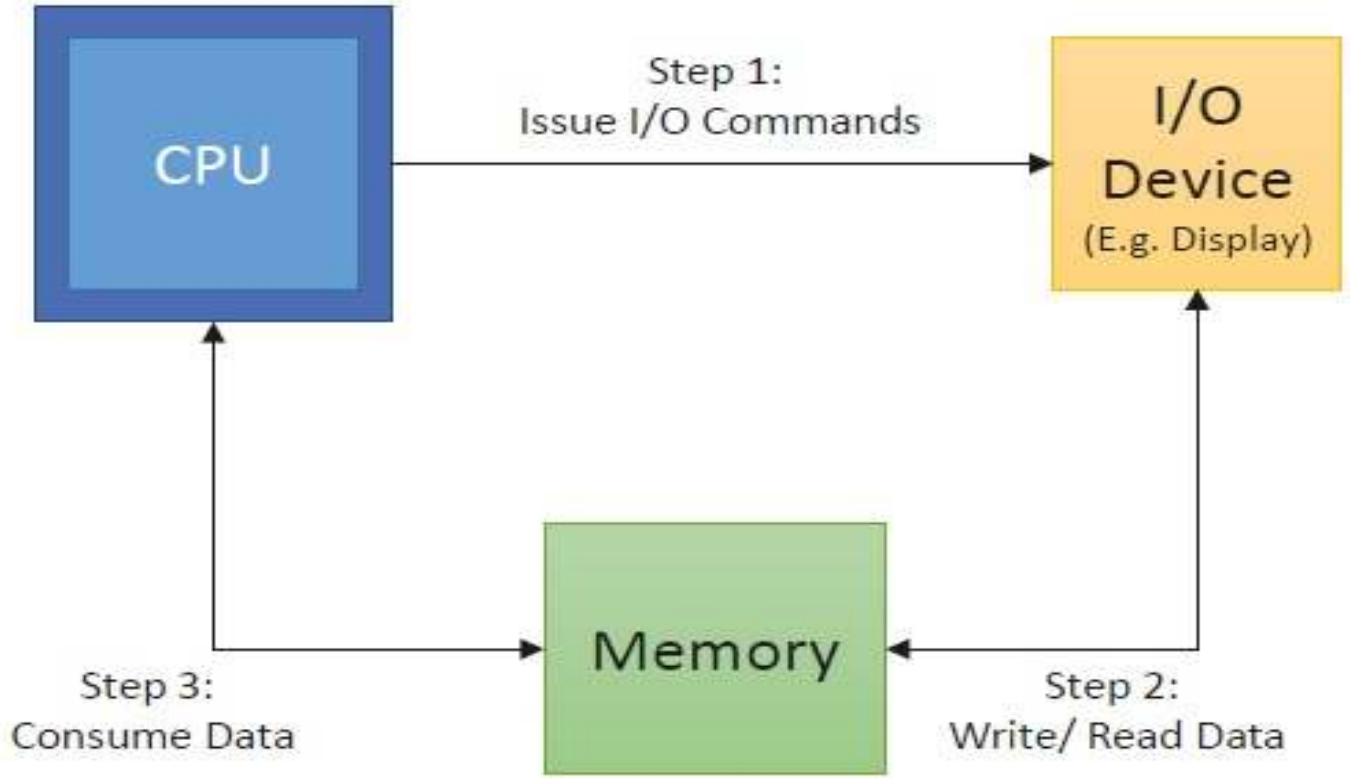
4.4.1 Special Instruction I/O



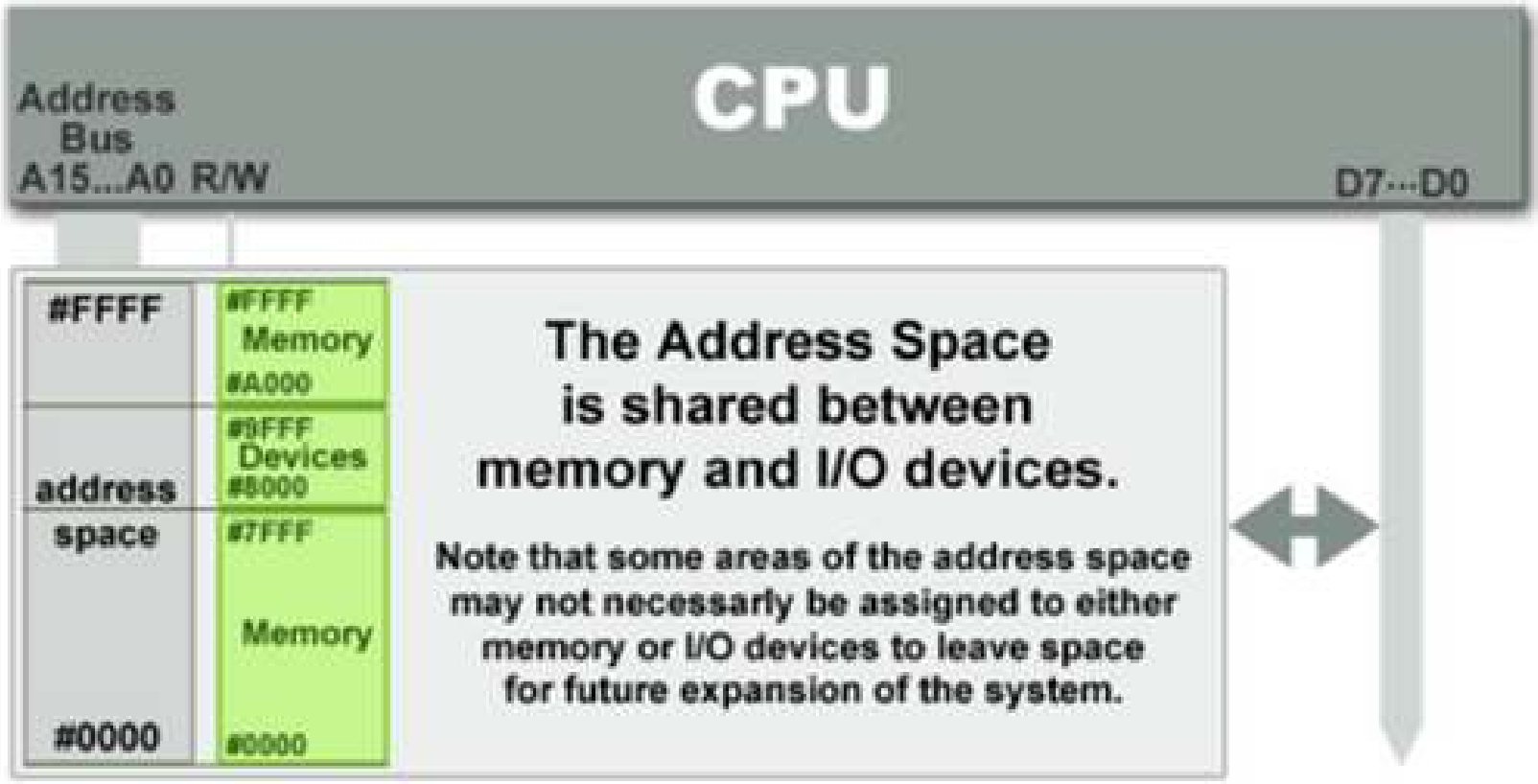
4.4.2 Memory-Mapped I/O

- The most common form of I/O protocol is memory-mapped I/O (MMIO).
- The device and OS agree on a common address range carved out by the OS, and the I/O device makes reads and writes from/to this space to communicate to the OS.
- OS components such as drivers will communicate using this interface to talk to the device.
- MMIO is also an effective mechanism for data transfer that can be implemented without using up precious CPU cycles.
- Hence, it is used to enable high-speed communication for network and graphics devices that require high data transfer rates due to the volume of data being passed.

4.4.2 Memory-Mapped I/O



4.4.2 Memory-Mapped I/O



4.4.3 Direct Memory Access (DMA)

- There could be devices that run at a slower speed than supported by the CPU or the bus it is connected on. In this case, the device can leverage DMA.
- Here, the OS *grants authority to another controller*, usually referred to as the *direct memory access controller*, to *interrupt the CPU after a specific data transfer is complete*.
- The devices running at a smaller rate can communicate back to the DMA controller after completing its operation.
- Note: Most OSs also handle additional specific device classes, blocking and nonblocking I/Os, and other I/O controls.
    - *As a programmer, you could be interacting with devices that may perform caching (an intermediate layer that acts as a buffer to report data faster) and have different error reporting mechanisms, protocols, and so on.*

4.5 Interrupt Handling Mechanisms

- Polling or programmed I/O:
  - In this mechanism, each period of time examine interrupt information and calls a specific routine (driver)

- Interrupt-Driven I/O:
  - In This mechanism, there is an ***interrupt vector*** which contains the addresses for all ***Interrupt Service Routines*** (ISR) –***drivers***- for all I/O devices, according the ***Interrupt Request Number*** (IRQ) –which ***unique*** for each device- the OS acquire the address of the address of the correct service routine
  - Most of OSs using this mechanism

4.5 Interrupt Handling Mechanisms

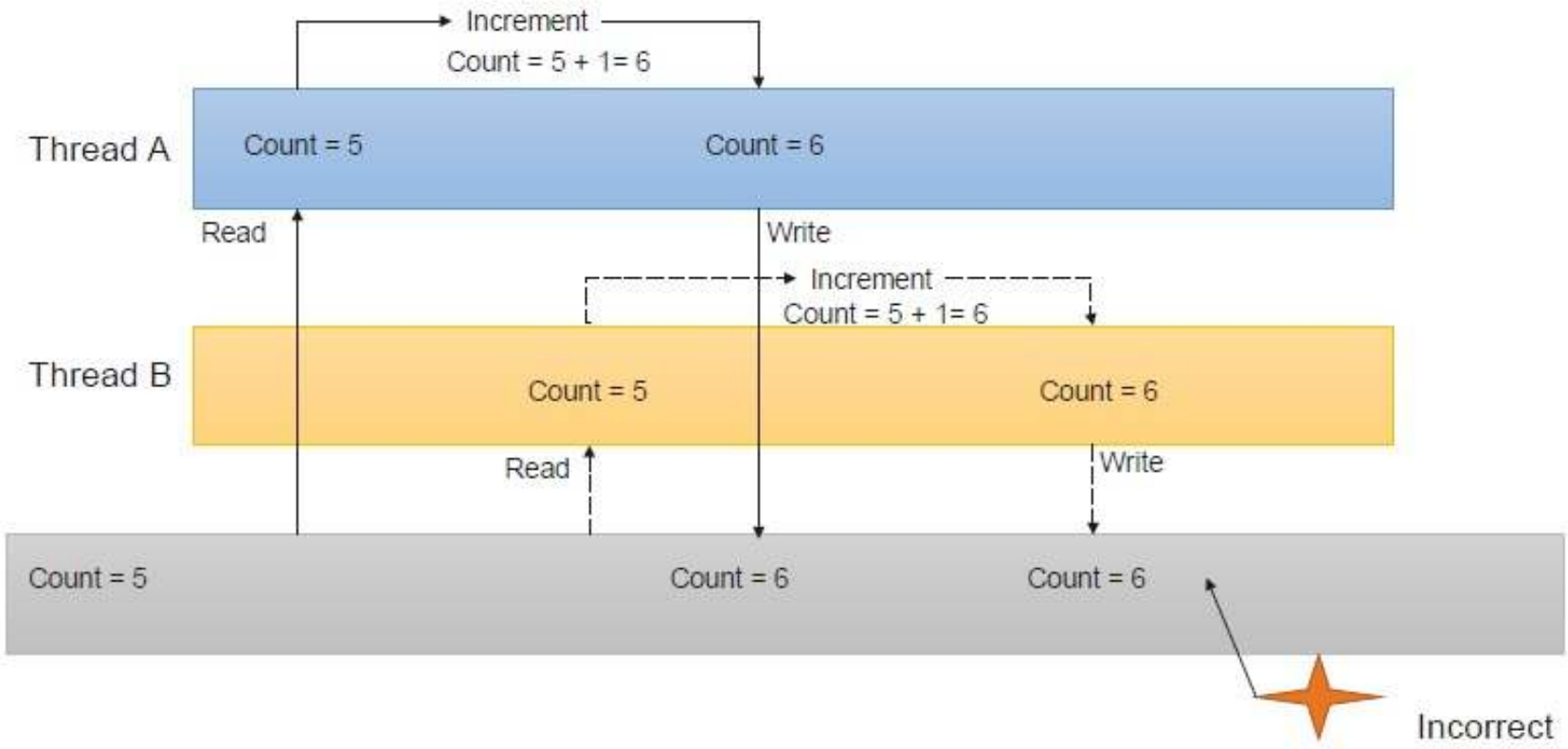
- Synchronous I/O: Example: Printing Operation
  - Process request I/O operation
  - I/O operation is started
  - I/O Operation is complete
  - Control is returned to the user process
- Asynchronous I/O: Example: Using Networking sending and receiving using threads
  - Process Request I/O operation
  - I/O operation is started
  - Control is returned immediately to the user process
  - I/O continues while system operations occur

4.7 Synchronization and Critical Sections

- In multi-threaded applications, if one thread tries to change the value of shared data at the same time as another thread tries to read the value, there could be a race condition across threads.
- In this case, the result can be unpredictable.
- The access to such shared variables via shared memory, files, ports, and other I/O resources needs to be synchronized to protect it from being corrupted.

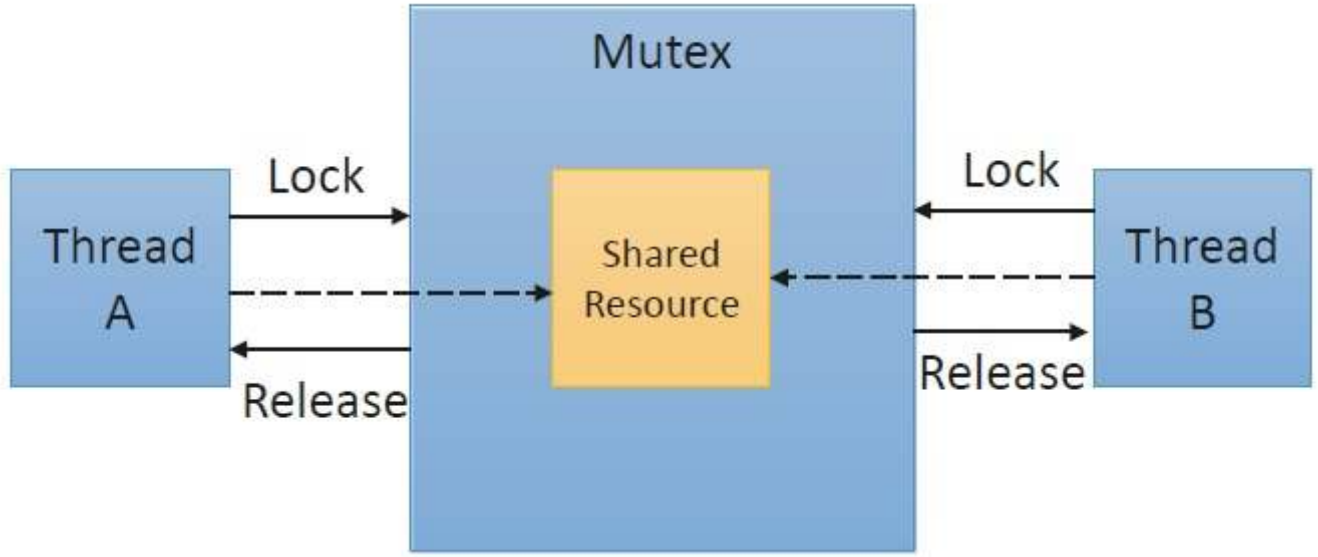
- order to support this, the operating system provides mutexes and semaphores to coordinate access to these shared resources.

4.7 Synchronization and Critical Sections



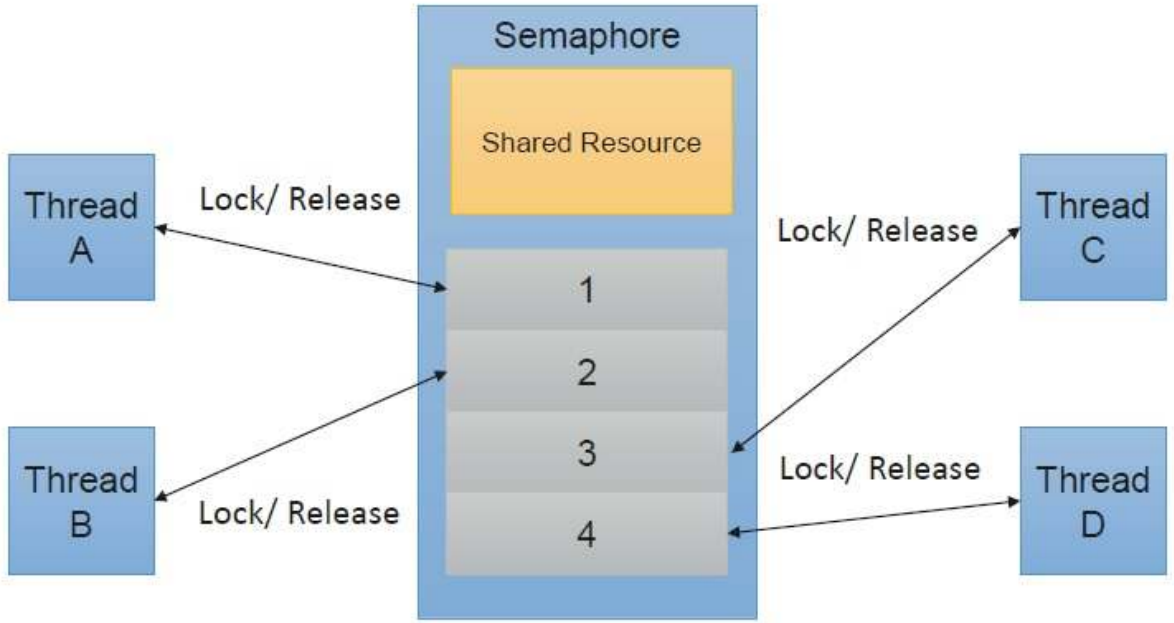
4.7.1 Mutex

- A mutex is used for implementing ***mutual exclusion***: either of the participating processes or threads can have the key (mutex) and proceed with their work.
- The other one would have to wait until the one holding the mutex finishes.



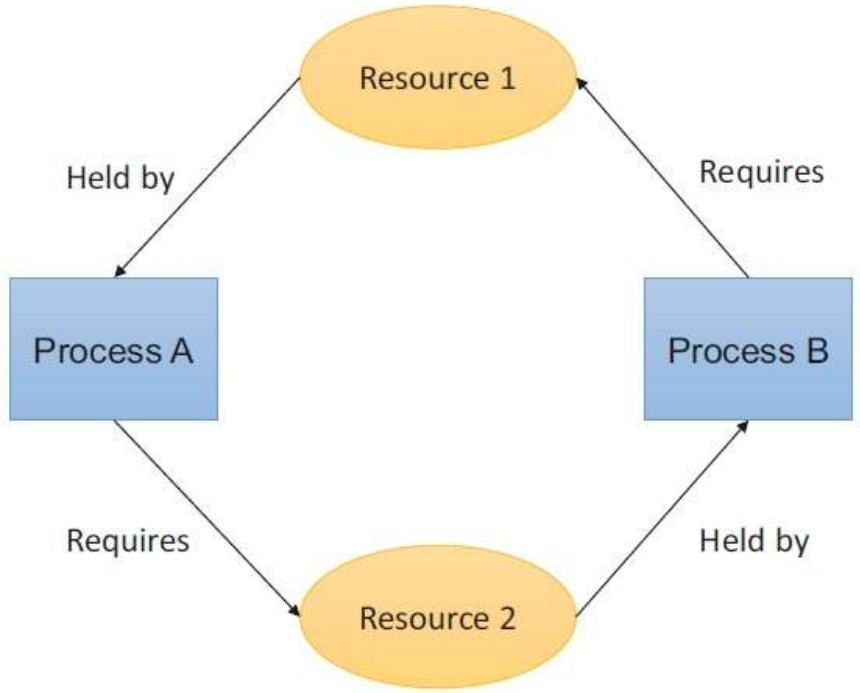
4.7.2 Semaphore

- A semaphore is a generalized mutex. A binary semaphore can assume a value of 0/1 and can be used to perform locks to certain critical sections.



4.8 Deadlocks

- When a set of processes become blocked because each process is holding a resource and waiting for another resource acquired by some other process. This is called as a ***deadlock***.
- Process A holds Resource 1 and requires Resource 2. However, Process B already is holding Resource 2, but requires Resource 1. Unless either of them releases their resource, neither of the processes may be able to move forward with the execution.



4.8 Deadlocks

- A deadlock can arise if the following four conditions hold:
  - **Mutual Exclusion**: There is at least one resource on the system that is not shareable. This means that only one process can access this at any point in time. In the preceding example, Resources 1 and 2 can be accessed by only one process at any time.
  - **Hold and Wait**: A process is holding at least one resource and is waiting for other resources to proceed with its action. In the preceding example, both Processes A and B are holding at least one resource.
  - **No Preemption**: A resource cannot be forcefully taken from a process unless released automatically.
  - **Circular Wait**: A set of processes are waiting for each other in circular form.

File Systems

Content

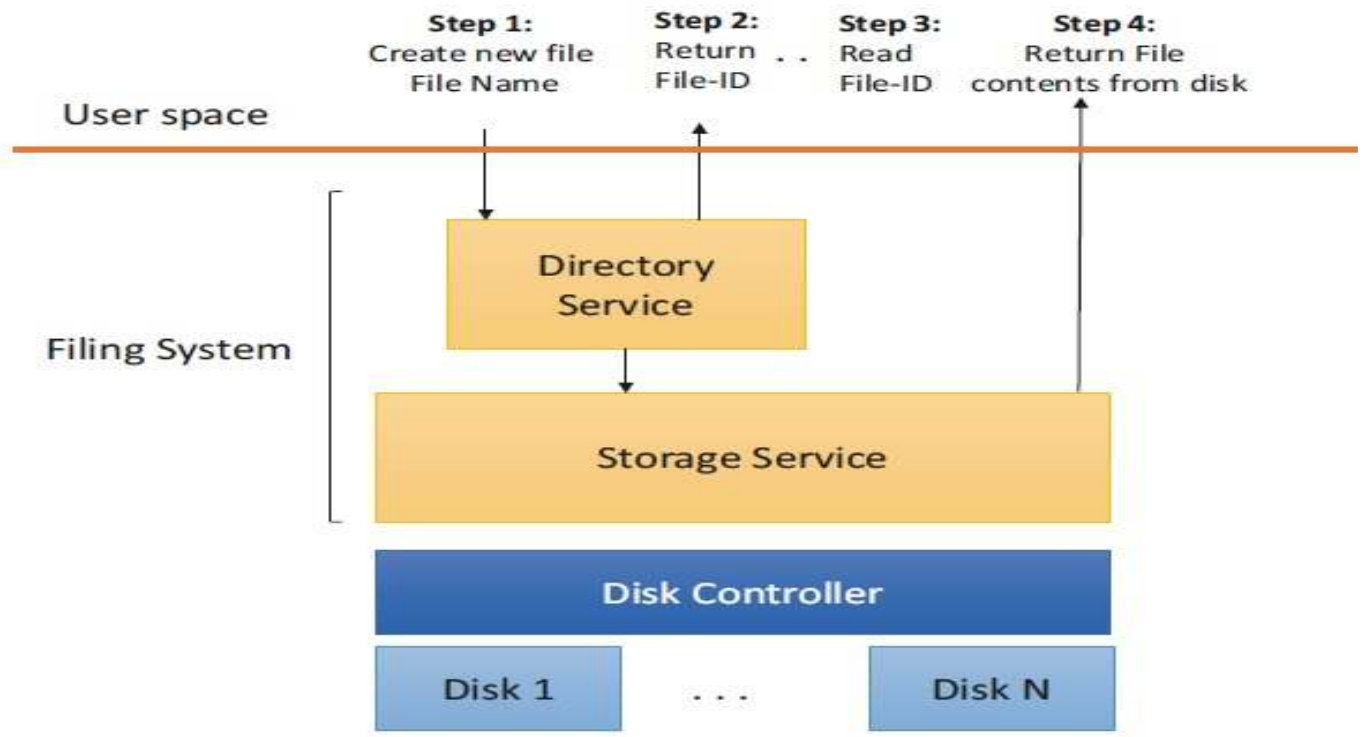
- Need for File Systems
- File Concept
- Directory Name Space
- Access Control
- Concurrency

Course Outlines

5.1 Need for File Systems

- Applications often need to read and write files to achieve their goals. We leverage the OS to create, read, and write such files on the system. We depend on the OS to maintain and manage files on the system.
- OS file systems have two main components to facilitate file management:
  - ***Directory Service***: There is a need to uniquely manage files in a structured manner, manage access, and provide Read-Write-Edit controls on the file system. This is taken care by a layer called as the directory service.
  - ***Storage Service***: There is a need to communicate to the underlying hardware such as the disk. This is managed by a storage service that abstracts different types of storage devices on the system.

5.1 Need for File Systems



5.2 File Concept

- From the perspective of the user, a file is a collection of related data that is stored together and can be accessed using a unique file ID usually referred as the file name.
- These files can be represented internally by different methods. For example, there could be **.bin** files in Windows, which only represent a sequence of bytes.
- There could be other structured contents with headers and specific sections in the file. For example, an **EXE** is also a file format in Windows with specific headers, a body, and controls in place.
- There are also many application-specific files, with their own formats. It is up to the programmer to define and identify if they require a custom file format for their application or if they can leverage a standard or common file format such as the JavaScript Object Notation (**JSON**) or the Extensible Markup Language (**XML**).

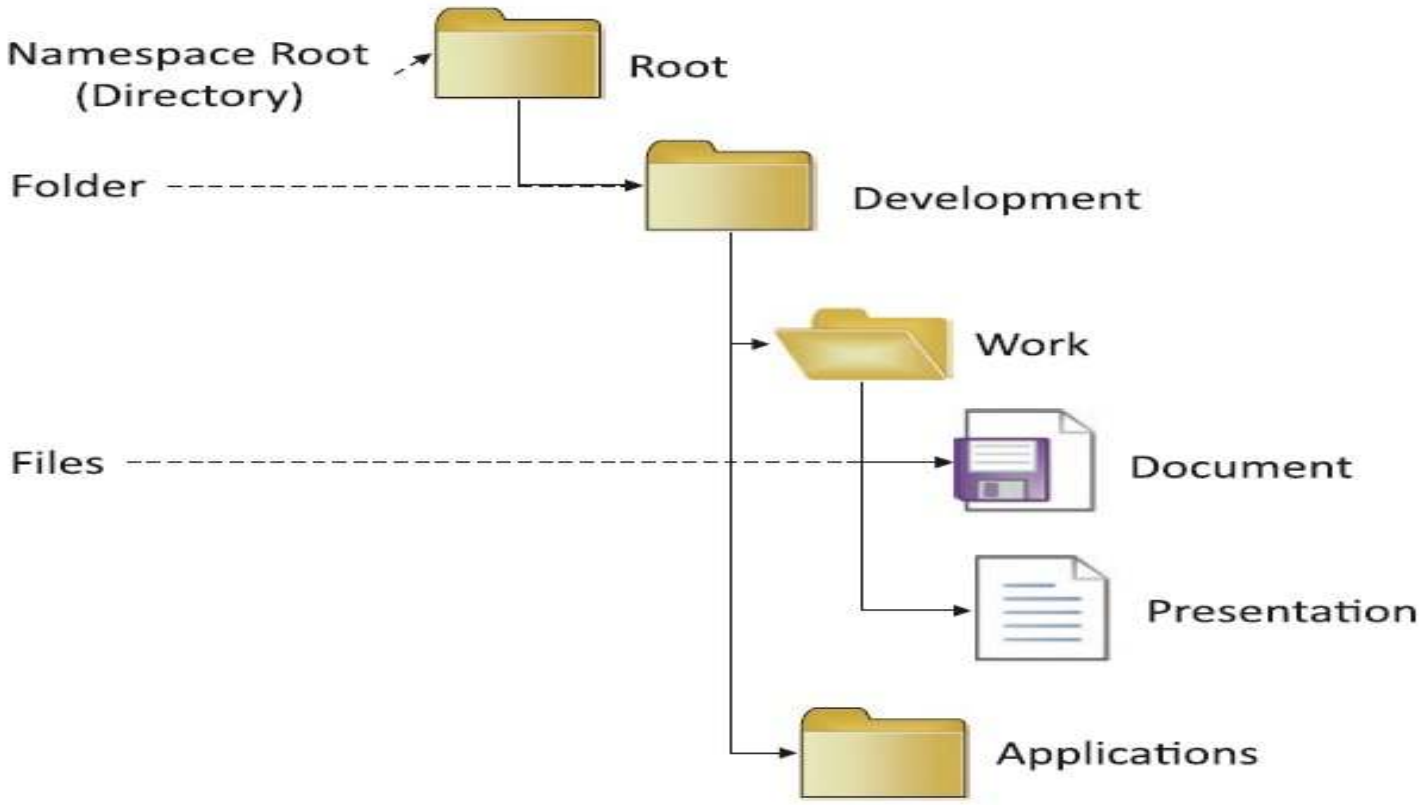
5.2 File Concept

- As a programmer, it may be important to know the attributes of the file before accessing it.
- The common attributes of any file include the ***location*** of the file, file ***extension***, ***size***, ***access controls***, and some ***history of operations*** done on the file, to name a few.
- Some of these are part of the so-called ***file control block***, which a user has access to via the OS.
- Most OSs expose APIs using which the programmer can access the details in the file control block.
- For the user, these are exposed on the ***graphical user interface via built-in tools*** shipped with the OS.

5.3 Directory Name Space

- The operating system defines a logical ordering of different files on the system based on the usage and underlying storage services. One of the criteria most OSs adopt is to structure their directory service to locate files efficiently.
- Most OSs ***organize*** their files in a ***hierarchical*** form with files organized inside ***folders***.
- Each folder in this case is a directory. This structure is called as the ***directory namespace***.
- The directory service and namespace have additional capabilities such as searches by size, type, access levels, and so on.
- The directory namespaces can be ***multileveled*** and adaptive in modern OSs as we can see in the following folder structure with folders created inside another folder

5.3 Directory Name Space



5.4 Access Control

- There are different access levels that can be applied at file and directory levels.
- The OS provides different access control ***IDs*** and ***permissions*** to different users on the system.
- Also, each file may also have ***different levels of permissions*** to Read, Write, Modify, and so on.
- For example, there may be specific files that we may want anyone to be able to access and Read but not Write and Modify.
- The file system provides and manages the controls to all files when accessed at runtime.
- These may also be ***helpful*** when ***more than one user*** is using the same system.

5.5 Concurrency and Cleanup Control

- There are many cases when the OS needs to ***ensure that a file is not moved or deleted when it is in use***.
- For example, if a user is making changes to a file, the OS needs to ensure that the same file cannot be moved or deleted by another application or process. In this case, the OS would cause the attempt to move or delete the file to fail with an appropriate error code.
- *As a programmer, it is appropriate to access a file with the required* ***access level*** *and* ***mode*** *(Read/Write). This also helps to be in line with the concurrency needs of the OS and guards against inconsistent updates.*

5.5 Concurrency and Cleanup Control

- The OS needs to be able to ***periodically clear temporarily created files*** that may no longer be required for the functioning of the system.
- This is typically done using a ***garbage collector on the system***.
- Many OSs ***mark unused files over a period of time*** and have additional settings that are exposed, which the user can set to clean up files from specified locations automatically.

Access and Protection

Content

- Access and Protection
  - User Mode and Kernel Mode (Rings)

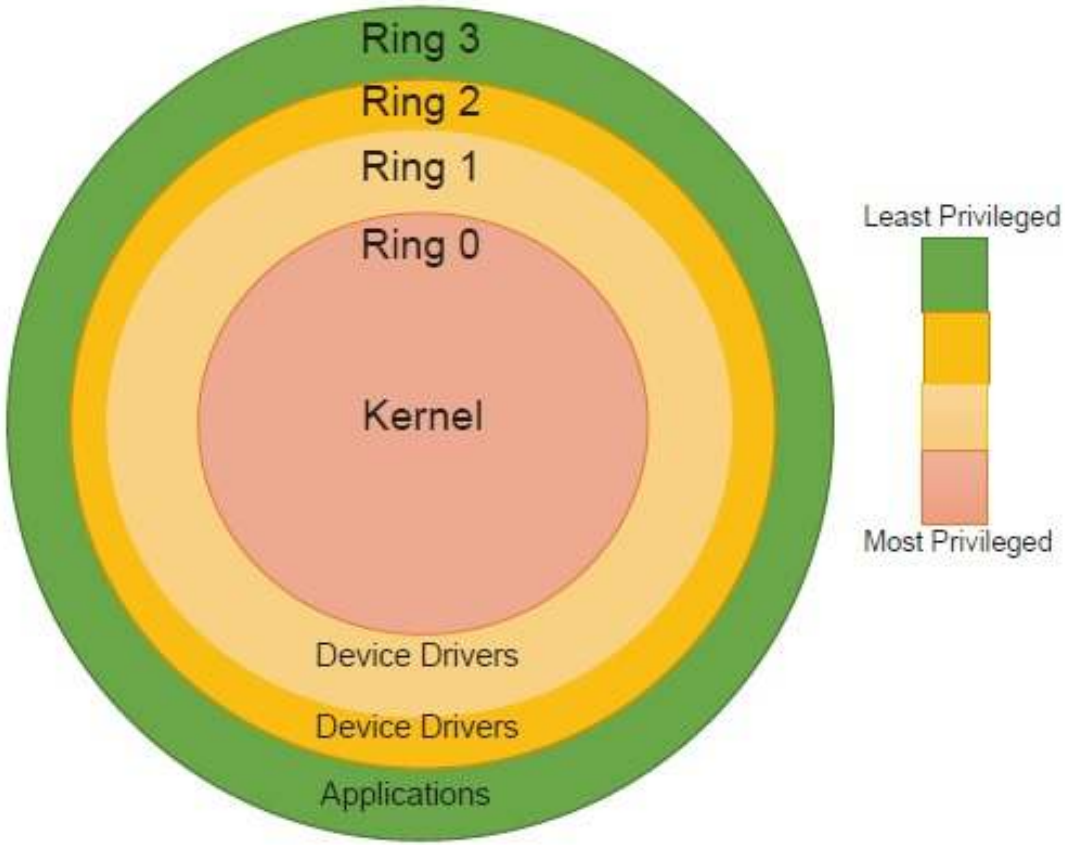
6.1 Access and Protection

- If we have a system that is used by only one user without any access, networked or otherwise, to other systems, there may still not be assurance that the contents in the system are protected.
- There is still a need to protect the ***program resources from other applications***.
- Also, there may be a need to protect ***critical devices*** on the system.
- There is always a need to connect and ***share resources and data between systems***.
- Hence, it is important to protect these resources accordingly.
- The OS provides APIs that help with access control and protection.

6.2 User Mode and Kernel Mode (Rings)

- One of the reasons the separation between user mode and kernel mode is implemented by most OSs is that it ensures ***different privilege levels are granted to programs, based on which mode they run in***.
- OS divides the program execution privileges into ***different rings***.
- Internally, programs running in ***specific rings are associated with specific access levels and privileges***.
- For example, ***applications and user-mode*** services running in *Ring 3* would ***not be able to access the hardware directly***. The ***drivers*** running on the *Ring 0* level would have the ***highest privileges and access to the hardware*** on the system.
- In practice, most OSs only leverage two rings, which are Ring 0 and Ring 3.

6.2 User Mode and Kernel Mode (Rings)



Virtualization and User Interface and Shells

Content

- Virtualization
- Protection
- User Interface and Shell

7.1 Virtualization

- Operating systems and modern hardware provide a feature called virtualization that ***virtualizes the hardware*** such that each calling environment believes it has the dedicated access it needs to function.
- Virtualization is delivered via so-called ***virtual machines*** (VMs).
- A VM has its own ***guest OS***, which may be the same as or different from the underlying ***host OS***.
- A user can launch a VM, much like running any other program, and log into the guest OS.
- The ***host OS*** provides a ***hypervisor***, which manages the access to the hardware.
- The guest OS is usually unaware of the internals and passes any resource/hardware requests to the host OS.
- The user can completely ***customize their VM and perform all their actions*** on this VM ***without affecting the host OS*** or any other VM on the system.

7.1 Virtualization

- At a high level, VMs help effectively utilize the hardware resources and are used heavily in server and cloud deployments.
- Advantages of virtualization:
  - Run operating systems where the ***physical hardware is unavailable***.
  - Easier to create ***new machines***, ***backup machines***, etc.
  - ***Software testing*** using "clean" installs of operating systems and software.
  - ***Emulate more machines*** than are physically available.
  - ***Debug problems*** (suspend and resume the problem machine).
  - ***Easy migration of virtual machines*** (shutdown needed or not).
  - Run ***legacy systems***

7.2 Protection

- There could be different ***security threats*** that may arise during the usage of a computer.
- A threat could be any ***local or remote program*** that may be attempting to ***compromise the integrity of the resources*** in the system.
- To mitigate this, modern OSs usually ***implement checks to detect and protect against such incursions***.
- The most common protection would be to ***authenticate the requester*** and apply ***authorization to any new request*** to the system.
- For example, when a request is made to a critical resource, the operating system would verify the ***user*** request (which is called as ***authentication***) and their approved ***access levels*** (which is called ***authorization***) and controls before providing access to a critical resource on the system.

7.2 Protection

- The OS may also have ***Access Control Lists (ACLs)*** that contain mapping of system resources to different permission levels. This is used internally before the OS grants permissions to any resource.
- Additionally, the OS may also provide services to encrypt and verify certificates that help with enhancing the security and protection of the system itself.

- *The programmer needs to be aware of the various access controls and protection mechanisms in place and use the right protocols and OS services to successfully access resources on the system.*

7.3 User Interface and Shell

- Although the ***user interface*** (UI) is not part of the OS kernel itself, this is typically considered to be ***an integral part of the OS***.
- Many Oss support different UIs, many of which are provided by third parties, for instance.
- There can be multiple user interfaces for the OS all being implemented either as a ***text-based*** interface or a ***graphical-based*** interface
- The graphical user interface is the rich set of graphical front-end interfaces and functionalities provided by the OS for the user to interact with the computer.
- There could be an alternate simpler interface through a ***command line shell interface*** that most OSs also provide for communication. This is a text-based interface.
- *It is common for programmers to use the shell interface instead of the GUI for quickly traversing through the file system and interacting with the OS.*

7.3 User Interface and Shell

- It is important for the software developer to be aware that the user interface and the shell interface may have an impact on their choice of programing language, handling of command line arguments, handling of the standard input-output pipes and interfacing with OS policies, and so on.
- Note that the user interface and the features can be quite varied and different from each OS to another

References

- "Essential Computer Science: A Programmer's Guide to Foundational Concepts": Paul D. Crutcher, Neeraj Kumar Singh, Peter Tiegs: Apress, 2021

Thank You

With My Best Wishes

Ahmed Loutfy