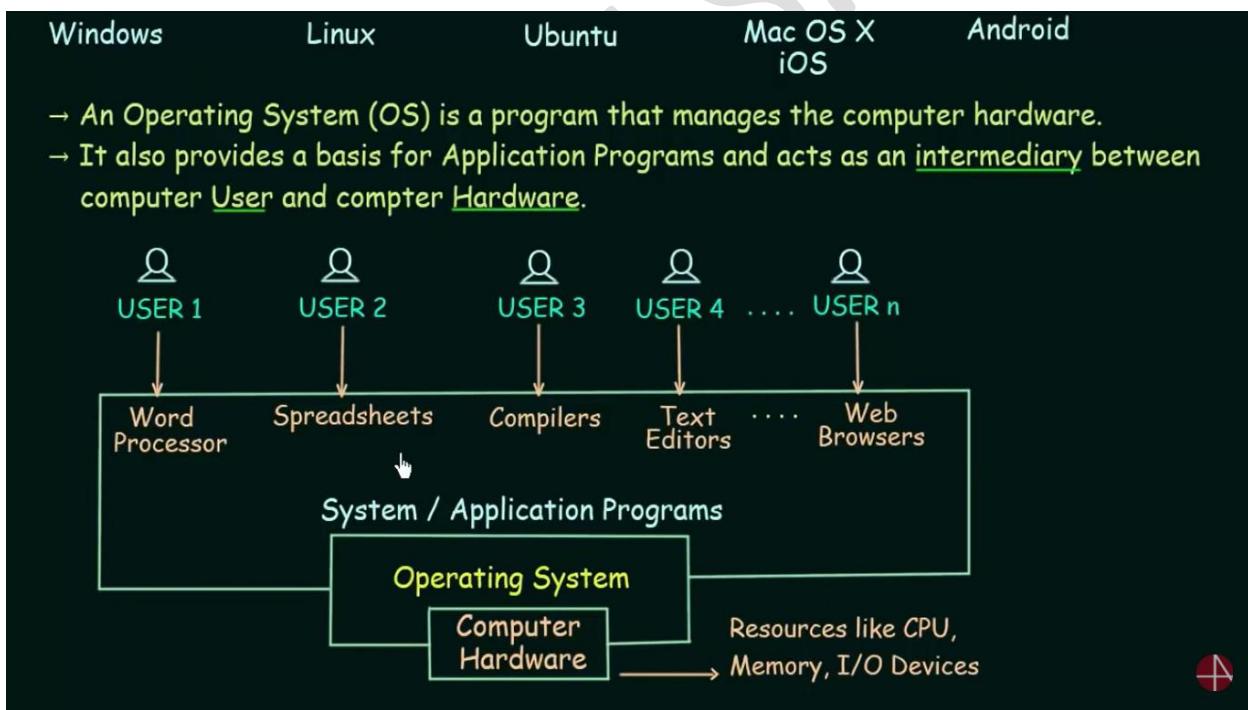
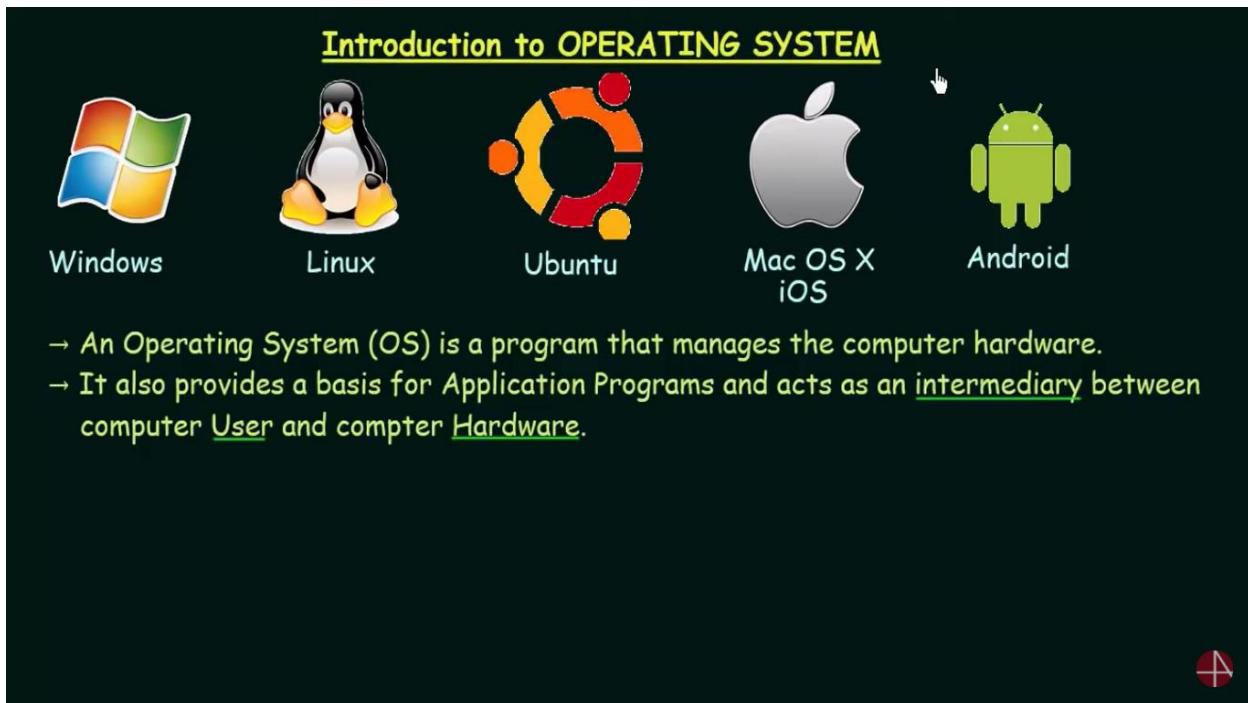


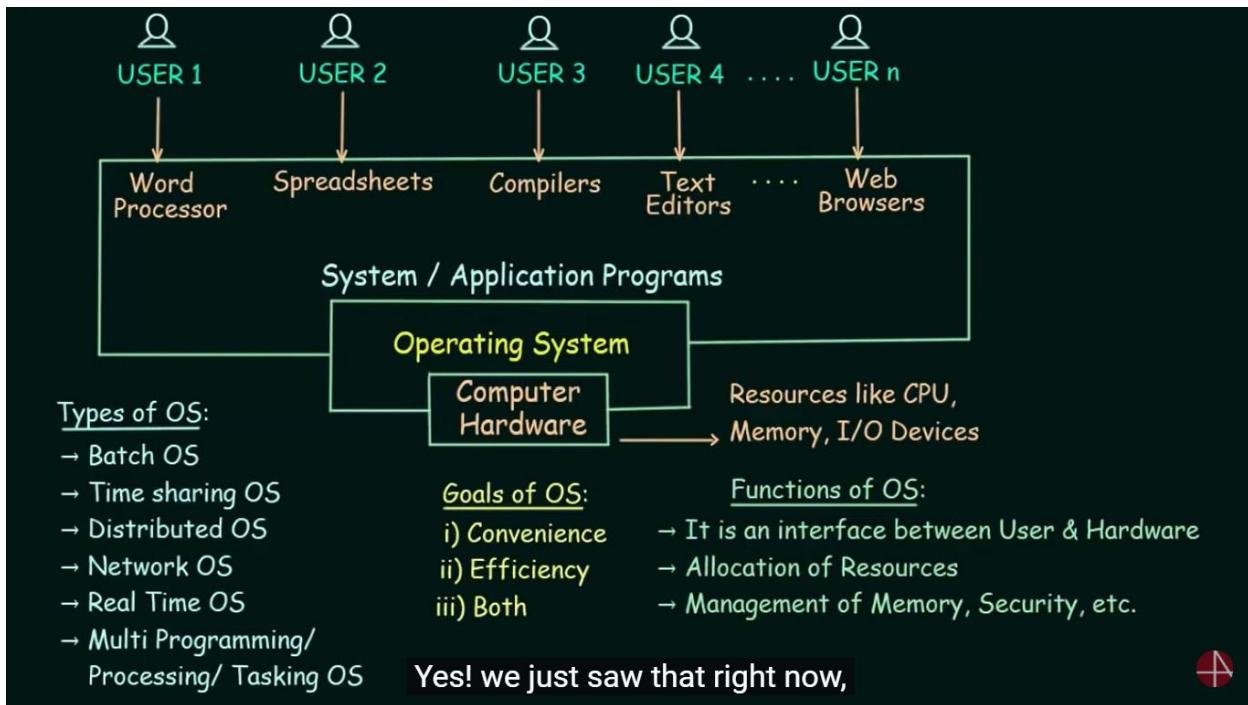
NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



NAME – RUSHIKESH SHARAD MANE

SUBJECT- OPERATING SYSTEM(OS)

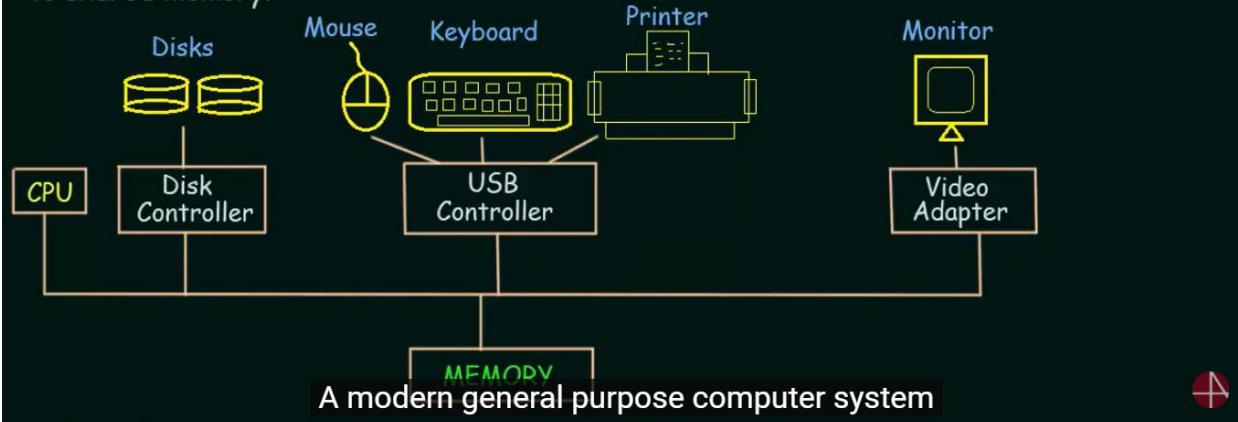
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



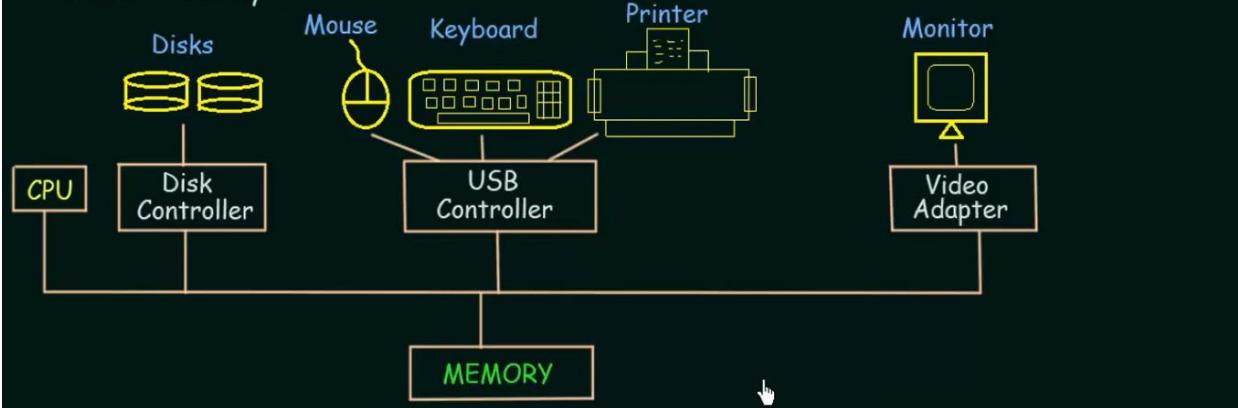
### Basics of Operating System (Computer System Operation)

Some basic knowledge of the structure of Computer System is required to understand how Operating Systems work.

- A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory.



a number of device controllers connected through a common bus that provides access to shared memory.



- Each device controller is in charge of a specific type of device
- The CPU and the device controllers can execute concurrently, competing for memory cycles
- To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access things are arranged.

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

→ To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory

Some important terms:



- 1) **Bootstrap Program:** → The initial program that runs when a computer is powered up or rebooted.
  - It is stored in the ROM.
  - It must know how to load the OS and start executing that system.
  - It must locate and load into memory the OS Kernel.
- 2) **Interrupt:** → The occurrence of an event is usually signalled by an Interrupt from Hardware or Software.
  - Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by the way of the system bus.
- 3) **System Call (Monitor call):** → Software may trigger an interrupt by executing a special operation called System Call.



When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.

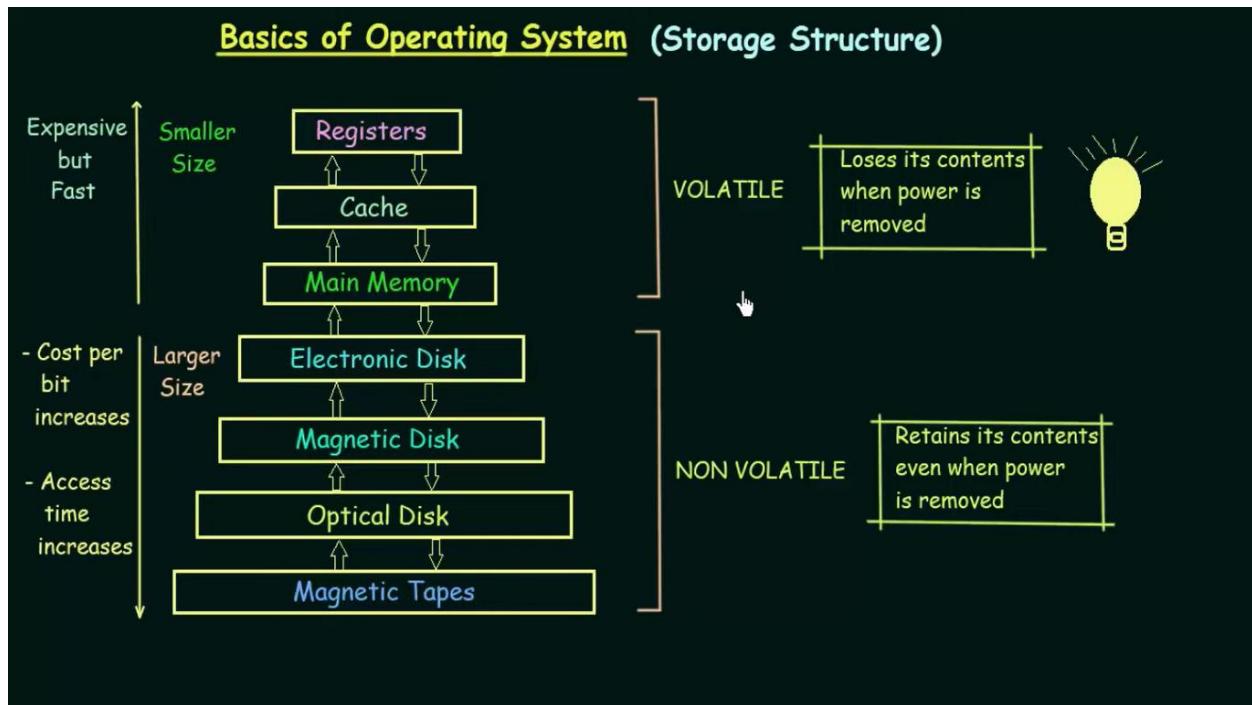
→ The fixed location usually contains the starting address where the Service Routine of the interrupt is located.

The Interrupt Service Routine executes.

On completion, the CPU resumes the interrupted computation.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

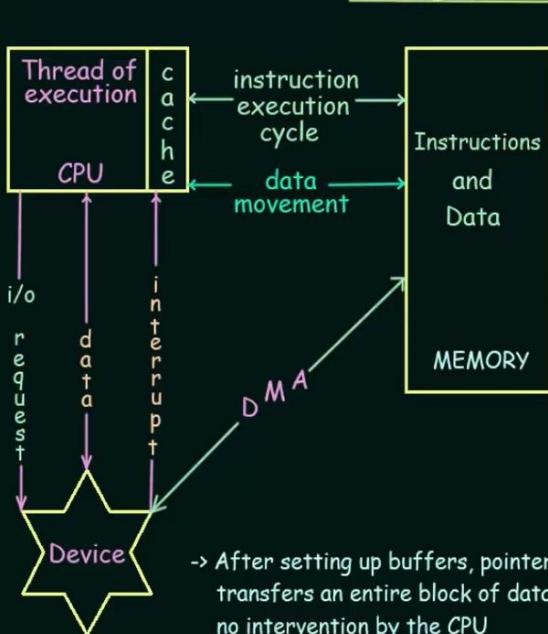


### Basics of Operating System ( I/O Structure)

- > Storage is only one of many types of I/O devices within a computer
- > A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices
- > A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus
- > Each device controller is in charge of a specific type of device
  - maintains
  - Local Buffer Storage
  - Set of Special Purpose Registers
- > Typically, operating systems have a device driver for each device controller
- > This device driver understands the device controller and presents a uniform interface to the device to the rest of the operating system



### Working of an I/O Operation:



- > To start an I/O operation, the device driver loads the appropriate registers within the device controller
- > The device controller, in turn, examines the contents of these registers to determine what action to take
- > The controller starts the transfer of data from the device to its local buffer
- > Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation
- > The device driver then returns control to the operating system

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement

To solve this problem, Direct Memory Access (DMA) is used

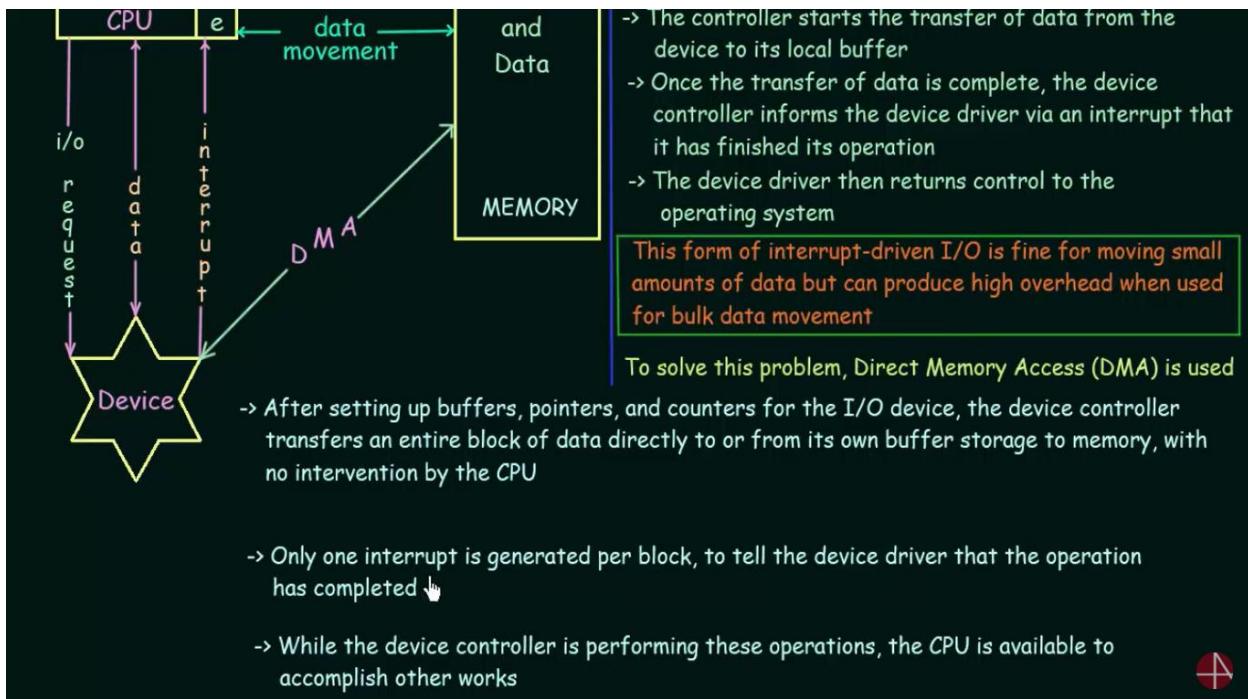
- > After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU



NAME – RUSHIKESH SHARAD MANE

SUBJECT- OPERATING SYSTEM(OS)

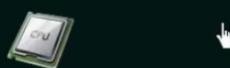
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



## Computer System Architecture

Types of Computer Systems based on number of General Purpose Processors:

### 1. Single Processor Systems



### 2. Multiprocessor Systems



### 3. Clustered Systems



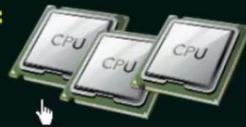
1. Single Processor Systems:



- One main CPU capable of executing a general purpose instruction set including instructions from user processes.
- Other special purpose processors are also present which perform device specific tasks



2. Multiprocessor Systems:



- Also known as parallel systems or tightly coupled systems.
- Has two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices

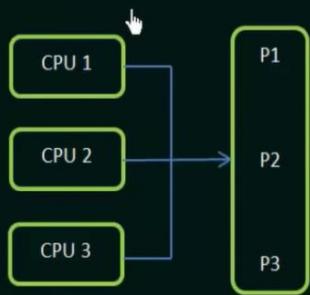
Advantages:

- 👍 Increased throughput
- 👍👍 Economy of scale
- 👍👍👍 Increased reliability

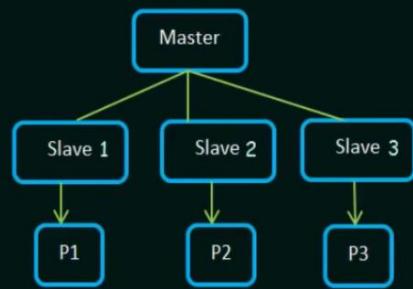


### Types of Multiprocessor Systems:

#### Symmetric Multiprocessing



#### Asymmetric Multiprocessing



### 3. Clustered Systems

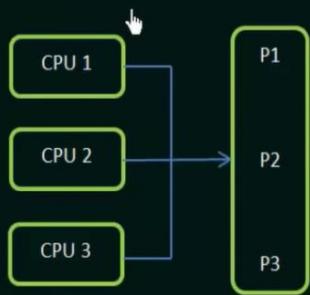


- Like multiprocessor systems, clustered systems gather together multiple CPUs to accomplish computational work.
- They are composed of two or more individual systems coupled together.
- Provides high availability
- Can be structured **asymmetrically** or **symmetrically**
  - One machine in Hot-Standby mode
  - Others run applications
  - Two or more hosts run applications
  - Monitors each other

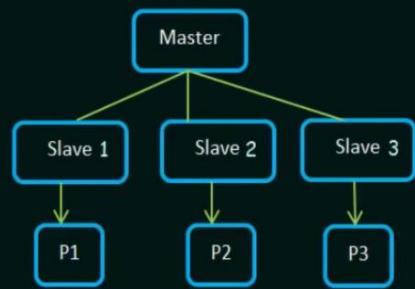


### Types of Multiprocessor Systems:

#### Symmetric Multiprocessing



#### Asymmetric Multiprocessing



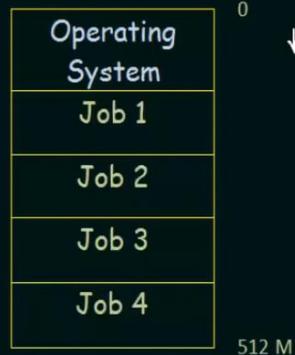
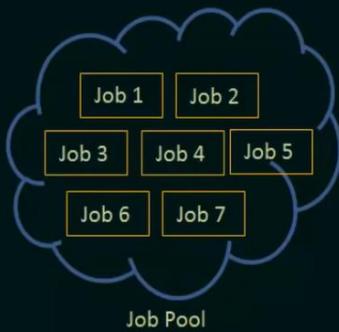
### Operating System Structure (Multiprogramming & Multitasking)

- Operating Systems vary greatly in their makeup internally
- **COMMONALITIES:**
  - (i) Multiprogramming
  - (ii) Time Sharing (Multitasking)



(i) Multiprogramming

- A single user cannot, in general, keep either the CPU or the I/O devices busy at all times
- Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.



Memory layout for a multiprogramming system

*Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the Computer system.*

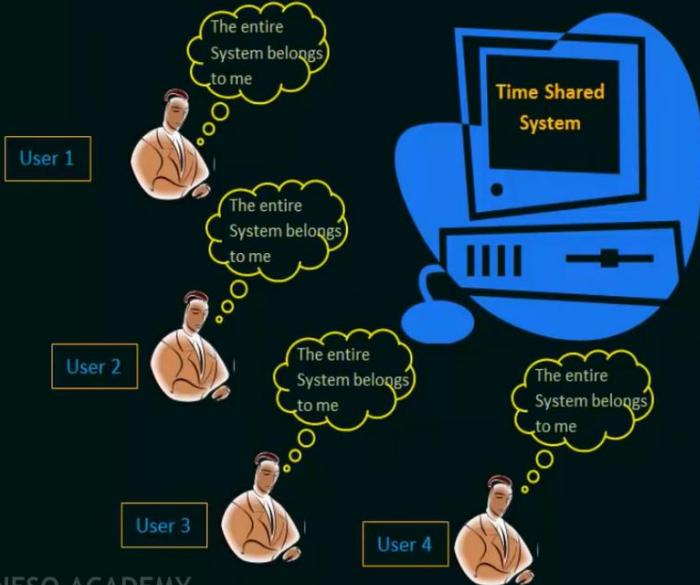


(ii) Time Sharing (Multitasking)

- CPU executes multiple jobs by switching among them
- Switches occur so frequently that the users can interact with each program while it is running
- Time sharing requires an interactive (or hands-on) computer system, which provides direct communication between the user and the system.
- A time-shared operating system allows many users to share the computer simultaneously.



- A time-shared operating system allows many users to share the computer simultaneously.



- Uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.
- Each user has at least one separate program in memory
- A program loaded into memory and executing is called a "PROCESS"



### Operating System Services

- An OS provides an environment for the execution of programs
- It provides certain services to programs and to users of those programs



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### 1) User Interface



Command Line Interface (CLI)

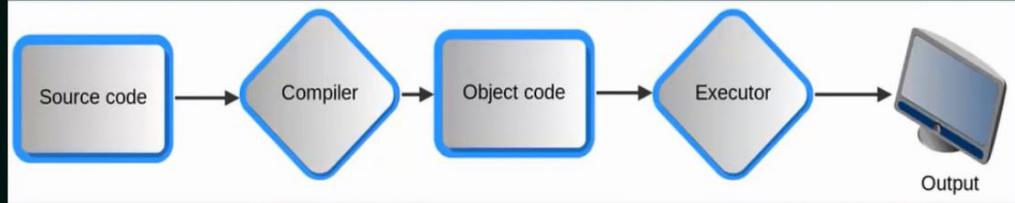


Graphical User Interface (GUI)

NESO ACADEMY



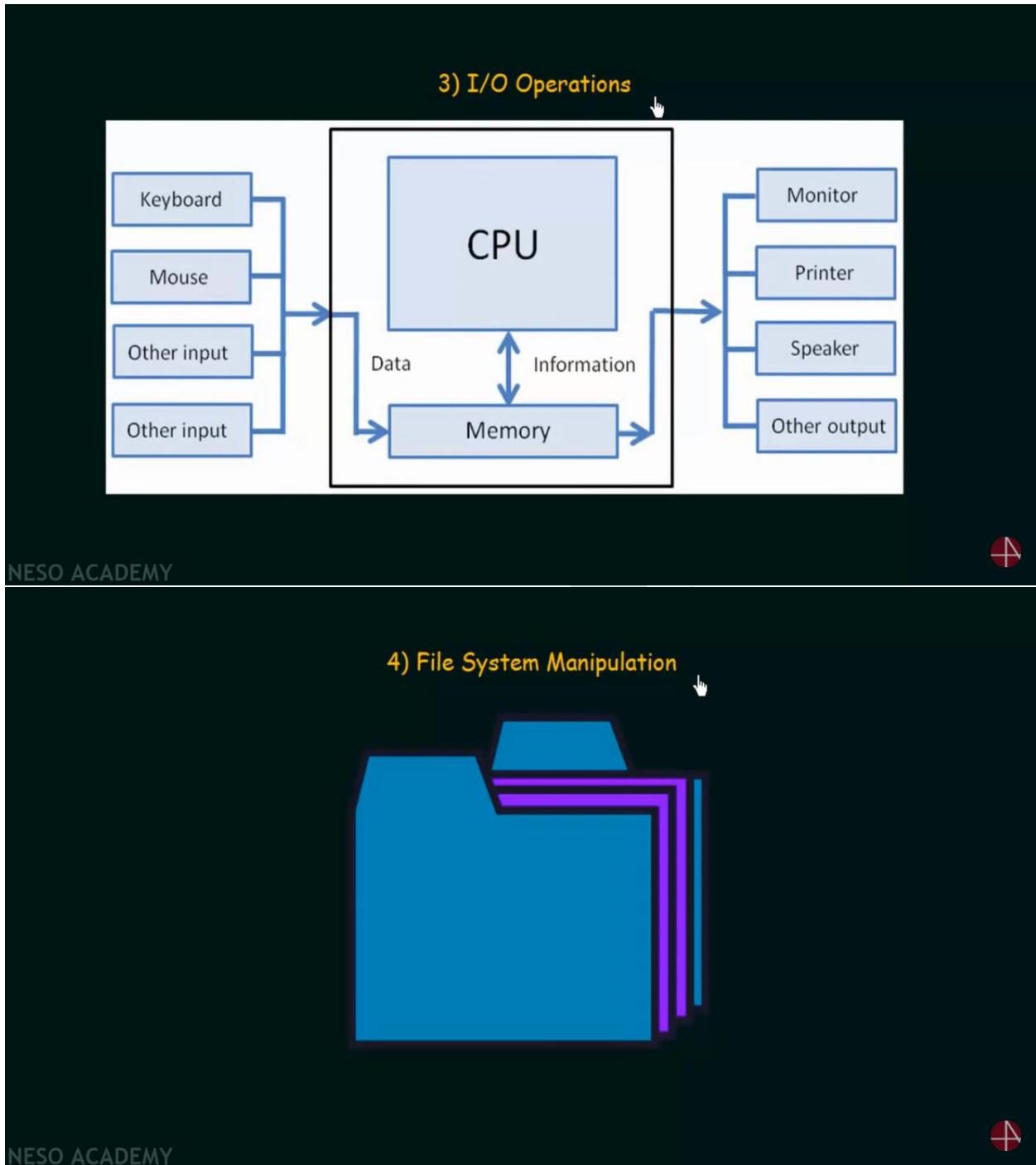
### 2) Program Execution



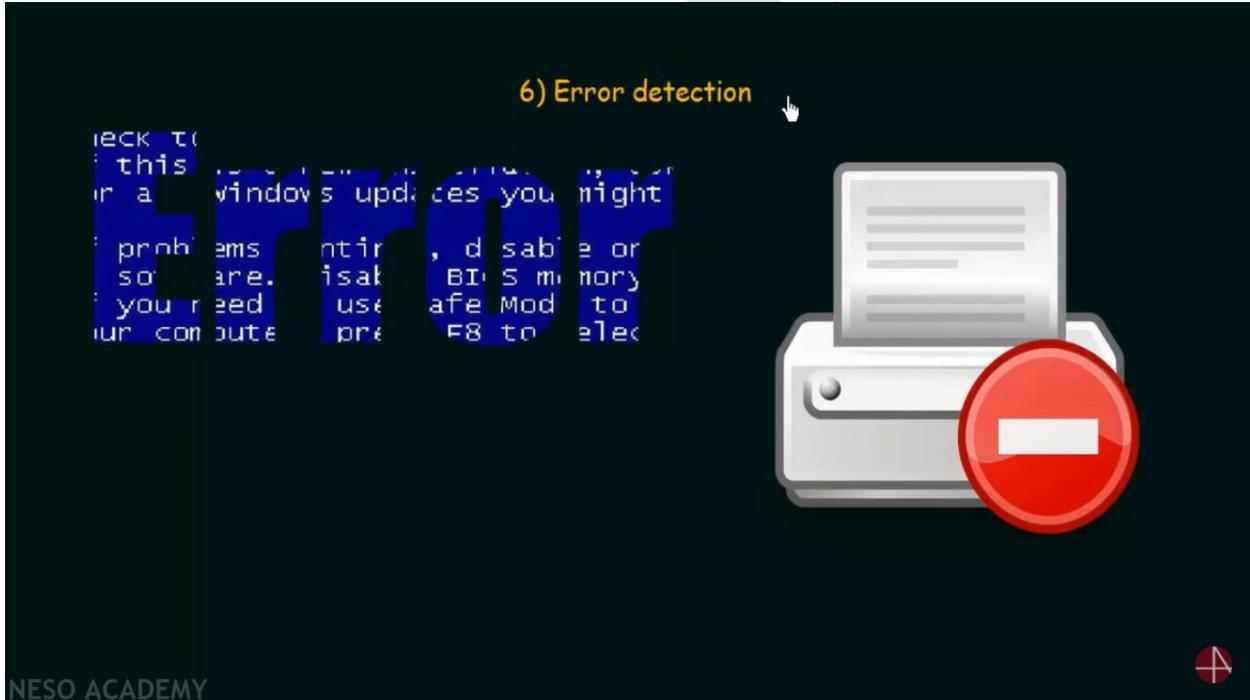
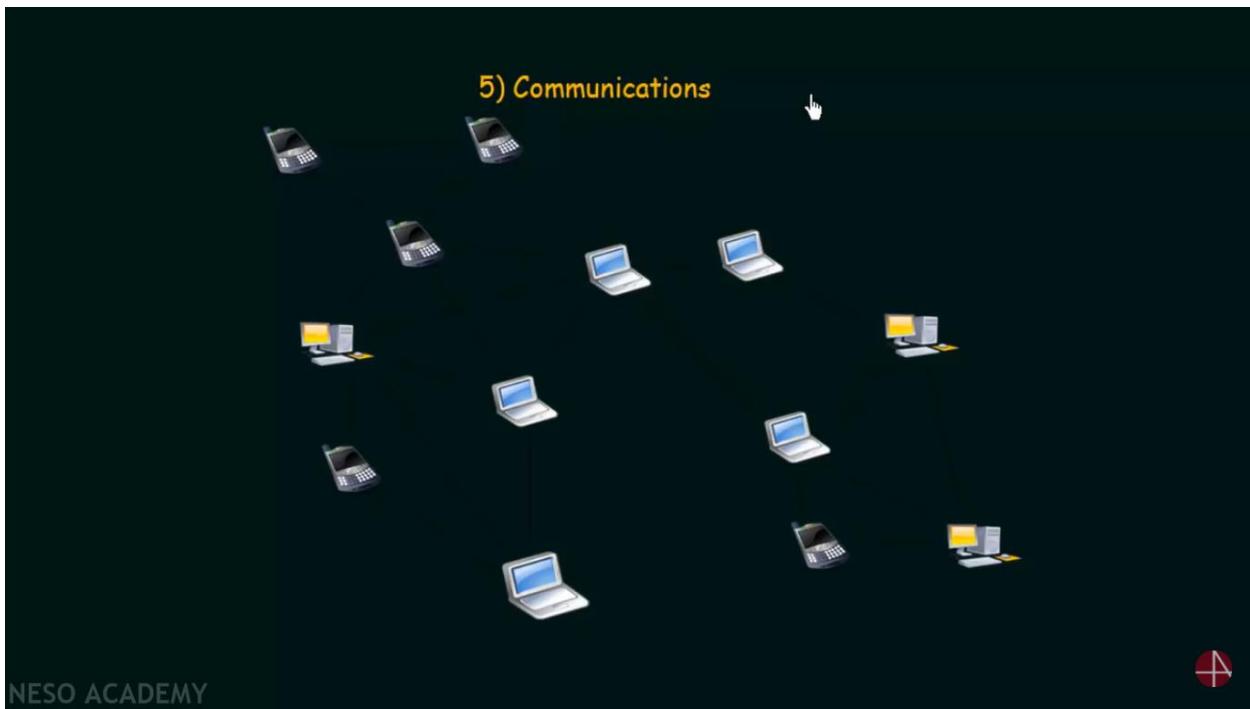
NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

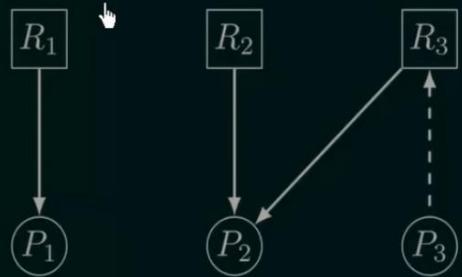


NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### 7) Resource Allocation



$R_4$

NESO ACADEMY



### 8) Accounting



NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

9) Protection and Security



NESO ACADEMY

Operating System Services

- 1. User Interface
- 2. Program Execution
- 3. I/O Operations
- 4. File System manipulation
- 5. Communications
- 6. Error Detection
- 7. Resource Allocation
- 8. Accounting
- 9. Protection and Security

NESO ACADEMY

### User Operating System Interface

There are two fundamental approaches for users to interface with the operating system:

- 1) Provide a **Command-Line Interface (CLI)** or **Command Interpreter** that allows users to directly enter commands that are to be performed by the operating system.
- 2) Allows the user to interface with the operating system via a **Graphical User Interface or GUI**.



### Command Interpreter

- Some operating systems include the command interpreter in the kernel.
- Others, such as Windows XP and UNIX, treat the command interpreter as a special program.
- On systems with multiple command interpreters to choose from, the interpreters are known as shells.
- E.g.
  - Bourne shell,
  - C shell
  - Bourne-Again shell (BASH)
  - Korn shell, etc.



### User Operating System Interface

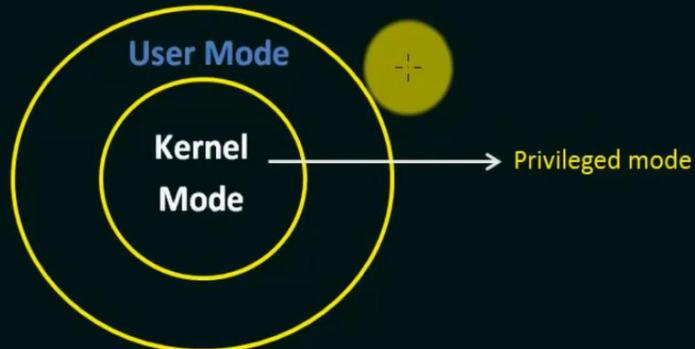
There are two fundamental approaches for users to interface with the operating system:

- 1) Provide a **Command-Line Interface (CLI)** or **Command Interpreter** that allows users to directly enter commands that are to be performed by the operating system.
- 2) Allows the user to interface with the operating system via a **Graphical User Interface or GUI**.

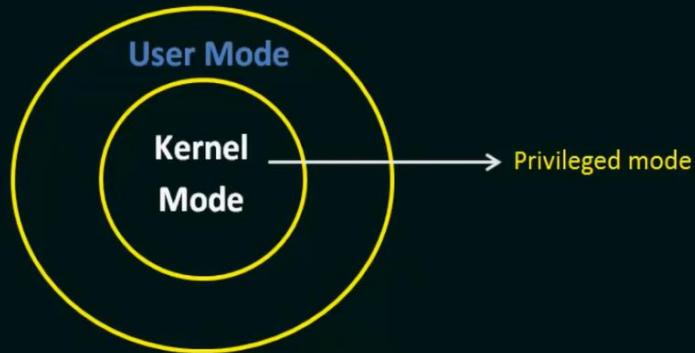


### System Calls

System calls provide an interface to the services made available by an Operating System.



System calls provide an interface to the services made available by an Operating System.

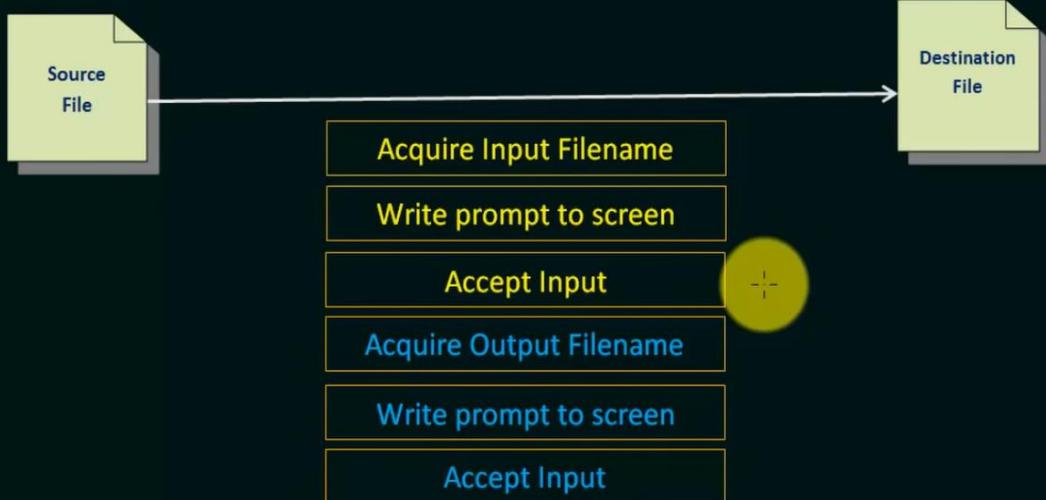


- System call is the programmatic way in which a computer program requests a service from the kernel of the operating system.
- These calls are generally available as routines written in C and C++.

NESO ACADEMY



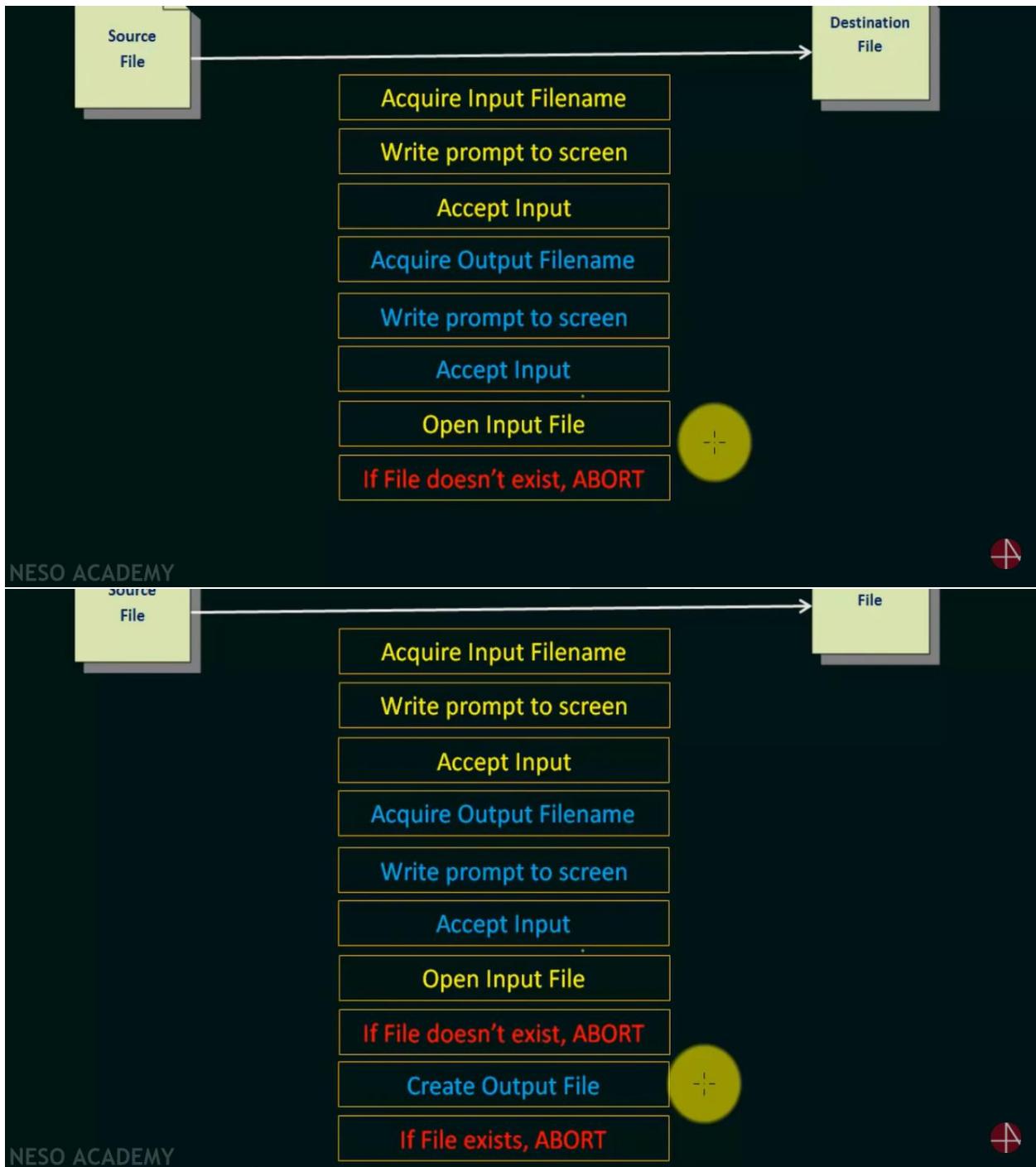
Example of a System Call sequence for writing a simple program to read data from one file and copy them to another file:



NESO ACADEMY



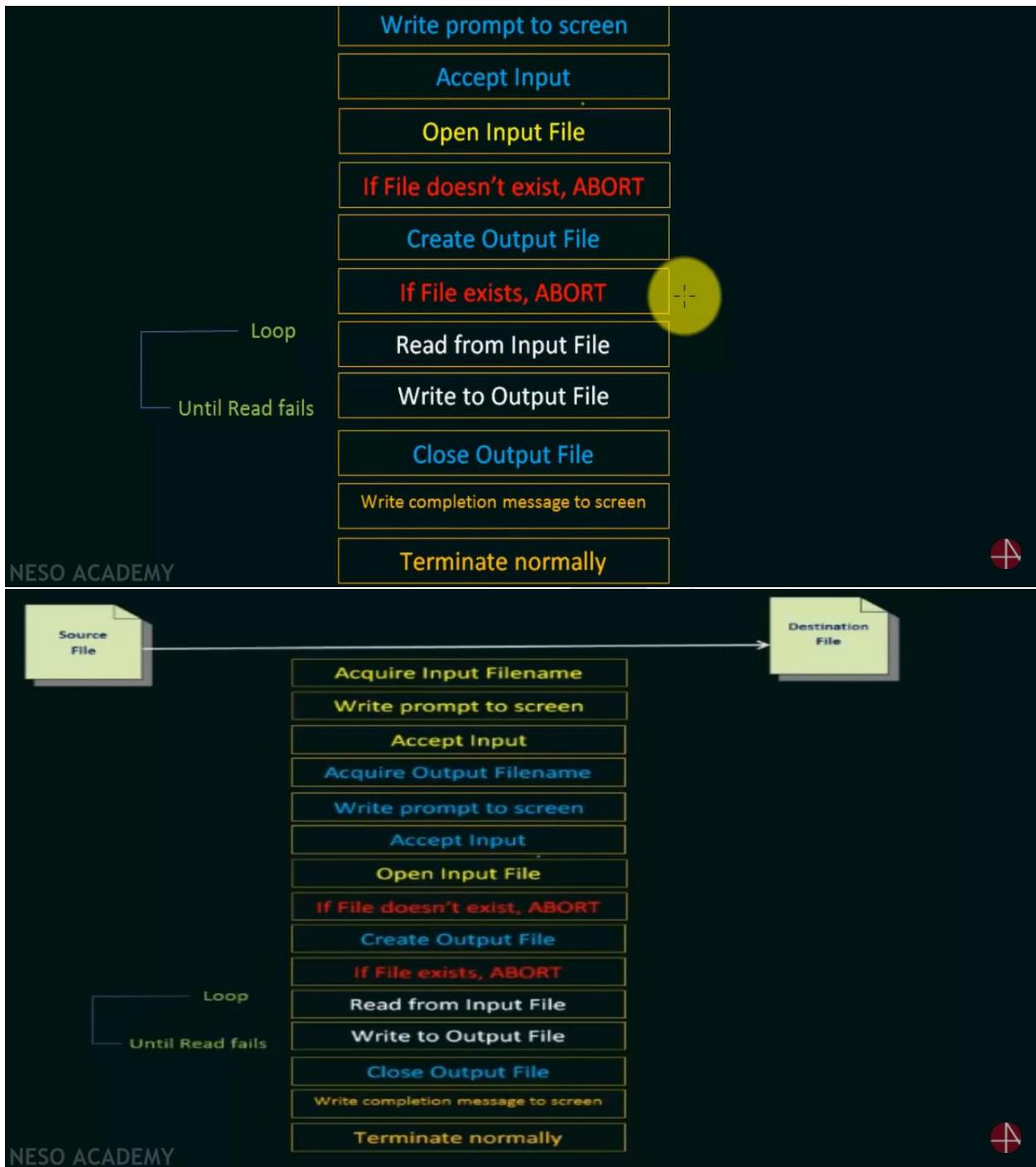
NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



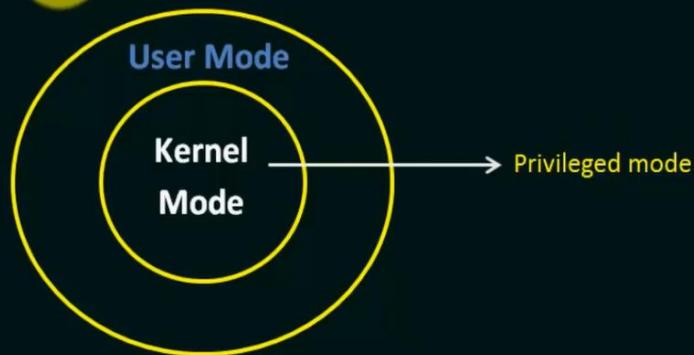
NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



System calls provide an interface to the services made available by an Operating System.



- System call is the programmatic way in which a computer program requests a service from the kernel of the operating system.
- These calls are generally available as routines written in C and C++.

NESO ACADEMY



### Types of System Calls

System calls can be grouped roughly into five major categories:

1. Process Control
2. File Manipulation
3. Device Management
4. Information Maintenance
5. Communications



NESO ACADEMY



### 1. Process Control

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory



### 2. File Manipulation

- create file, delete file
- open, close
- read, write, reposition
- get file attributes, set file attributes



### 3. Device Manipulation

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices



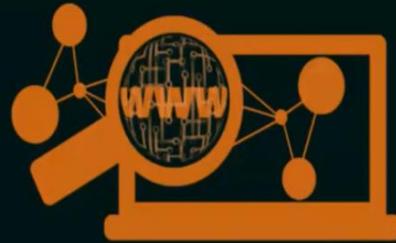
### 4. Information Maintenance

- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes



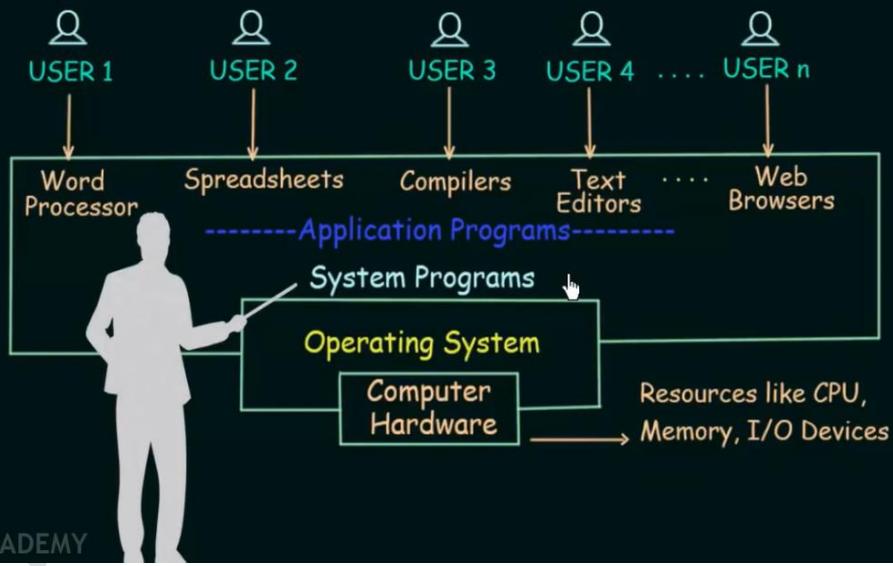
## 5. Communications

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

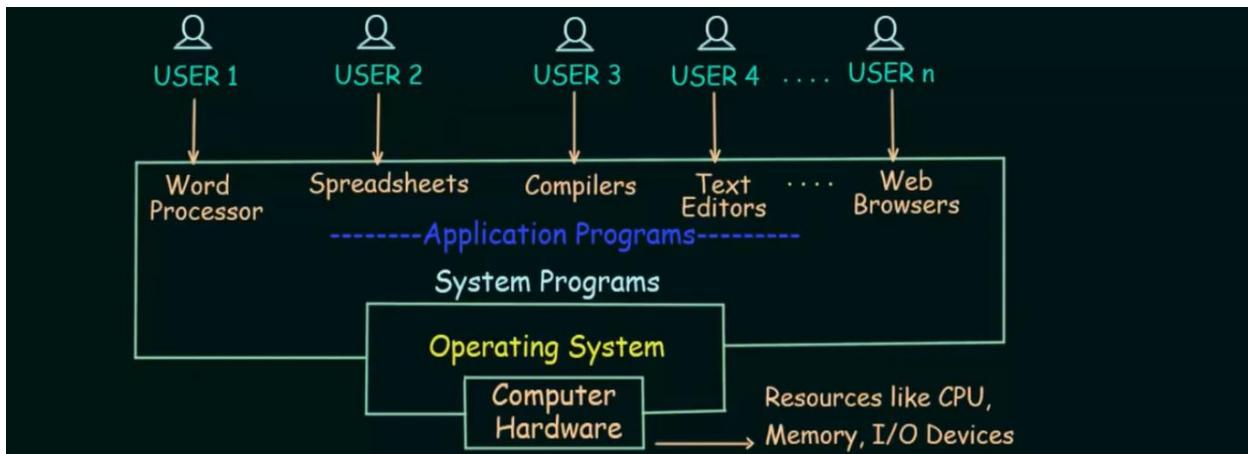


## System Programs

An important aspect of a modern system is the collection of system programs.

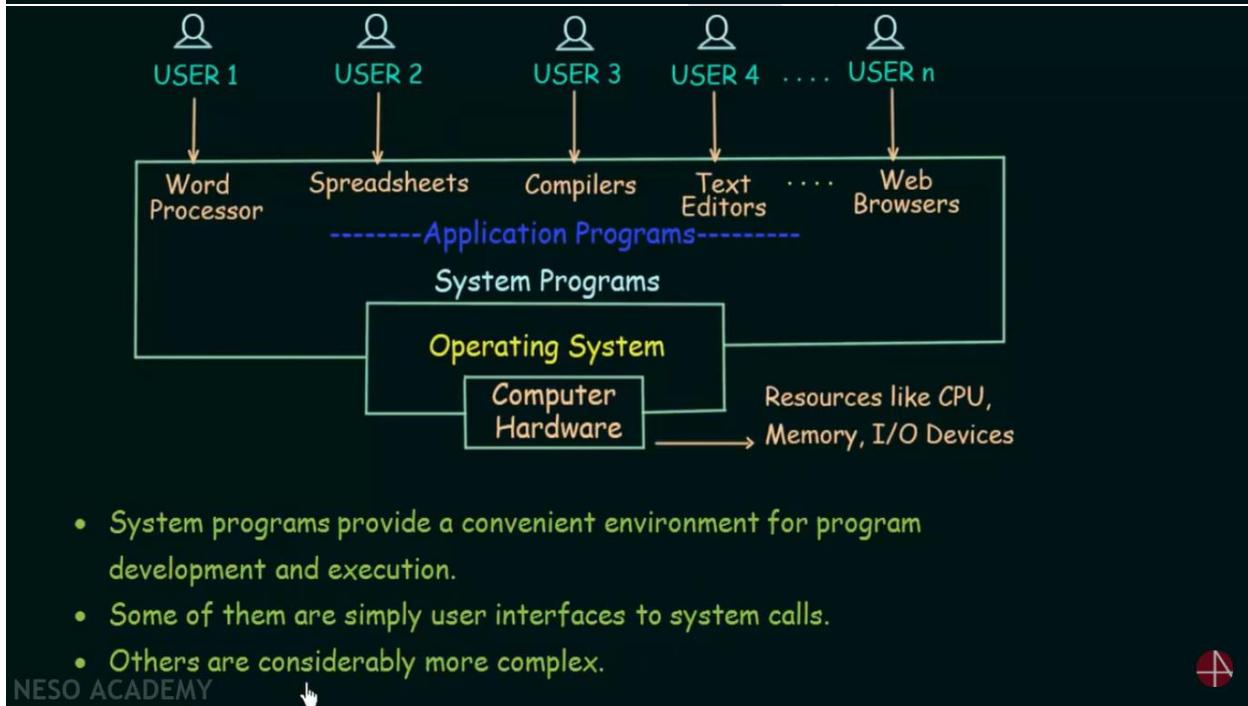


NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



- System programs provide a convenient environment for program development and execution.
- Some of them are simply user interfaces to system calls.
- Others are considerably more complex.

NESO ACADEMY



NESO ACADEMY



System Programs can be divided into the following categories:

File Management

- Create
- Delete
- Copy 
- Rename
- Print
- Dump
- List, and generally manipulate files and directories.



Status Information 

Ask the system for:

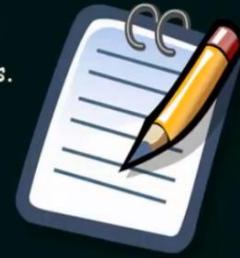
- Date, Time
- Amount of available memory or disk space
- Number of users
- Detailed performance
- Logging, and debugging information etc.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### File modification

- Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
- There may also be special commands to search contents of files or perform transformations of the text.



NESO ACADEMY



### Programming-language support

- Compilers
- Assemblers
- Debuggers and
- Interpreters

for common programming languages  
(such as C, C++, Java, Visual Basic, and PERL)  
are often provided to the user with the operating system.



NESO ACADEMY



### Program loading and execution



Once a program is assembled or compiled, it must be loaded into memory to be executed.

The system may provide:

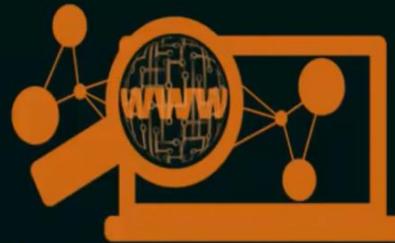
- ⦿ Absolute loaders
- ⦿ Relocatable loaders
- ⦿ Linkage editors and
- ⦿ Overlay loaders



Debugging systems for either higher-level languages or machine language are needed as well.



### Communications



These programs provide the mechanism for:

- ⦿ Creating virtual connections among processes, users, and computer systems.
- ⦿ Allowing users to send messages to one another's screens
- ⦿ To browse webpages
- ⦿ To send electronic-mail messages
- ⦿ To log in remotely or to transfer files from one machine to another.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Examples:

Web Browsers



Word Processors



Spreadsheets



Database Systems



Games



etc.

Application  
Programs



Categories of System Programs:

- File Management
- Status Information
- File Modification
- Programming-language support
- Program Loading and Execution
- Communications



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

## Operating System Design & Implementation

### Design Goals:

1<sup>st</sup> Problem: - Defining Goals and specification

- Choice of Hardware
- Type of System

Beyond this highest design level, the requirements may be much harder to specify.

### Requirements:

- User Goals ↴
- System Goals

NESO ACADEMY



### Requirements:

- User Goals
- System Goals

User Requirements:  
(User Goals)

The system should be:  
Convenient to use,  
Easy to learn & use,  
Reliable, safe & fast



NESO ACADEMY

Designer,  
Engineer Requirements:  
(System goals)

The system should be:  
Easy to design, implement,  
maintain, operate.  
It should be flexible, reliable  
error free & efficient



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Mechanisms and Policies:

Mechanisms determine how to do something.

Policies determine what will be done.



NESO ACADEMY



Mechanisms and Policies:

Mechanisms determine how to do something.

Policies determine what will be done.

**One important principle is the separation of policy from mechanism.**



Good and Flexible



Not Good

NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### Implementation:

- Once an operating system is designed, it must be implemented.
- Traditionally, operating systems have been written in assembly language.
- Now, however, they are most commonly written in higher-level languages such as C or C++

NESO ACADEMY



### Advantages of writing in high level languages:

- The code can be written faster
- It is more compact
- It is easier to understand and debug
- It is easier to port

E.g.

MS-DOS was written in Intel 8088 assembly language. Consequently, it is available on only the Intel family of CPUs.



The Linux operating system, in contrast, is written mostly in C and is available on a number of different CPUs, including Intel 80X86, Motorola 680X0, SPARC, and MIPS RX000.

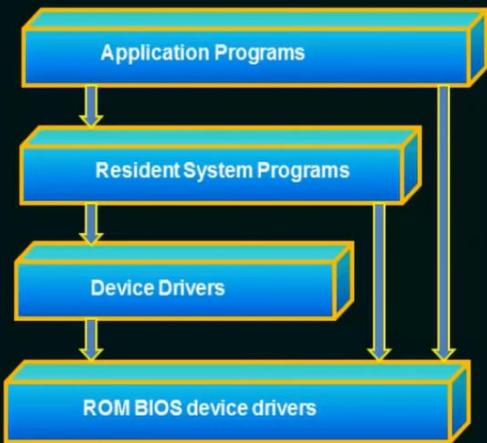
NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

## Structures of Operating System

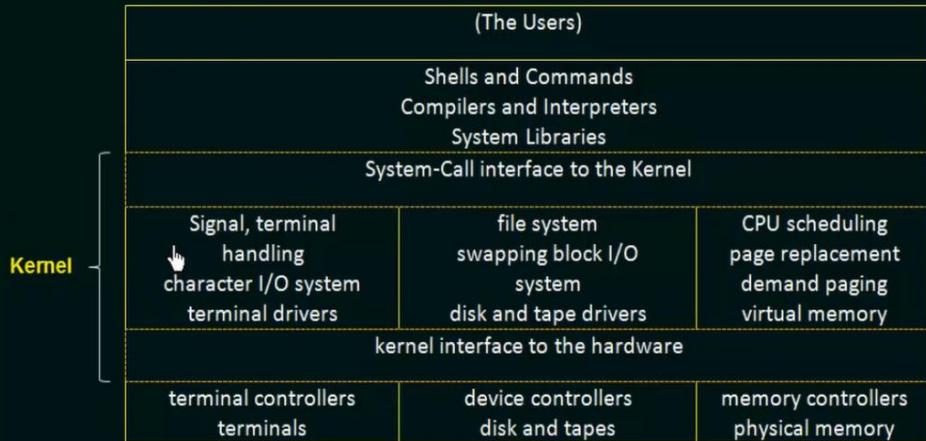
### Simple Structure



NESO ACADEMY



### Monolithic Structure

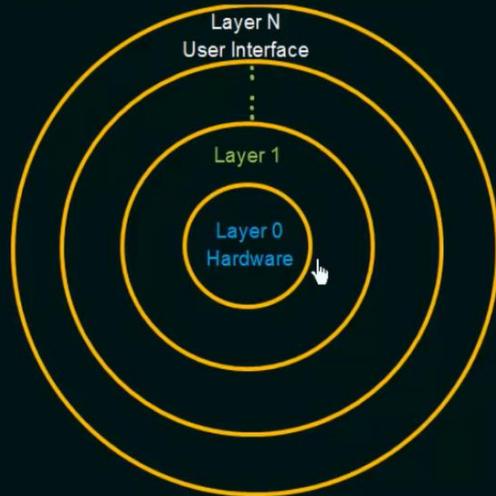


NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

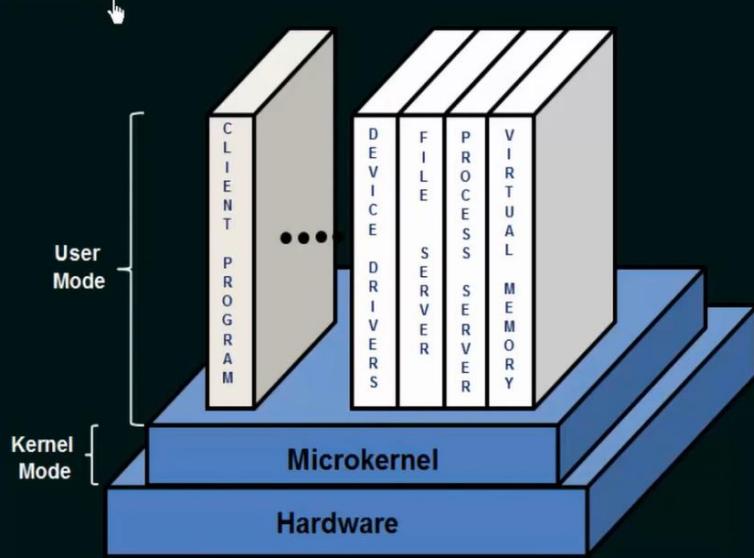
### Layered Structure



NESO ACADEMY



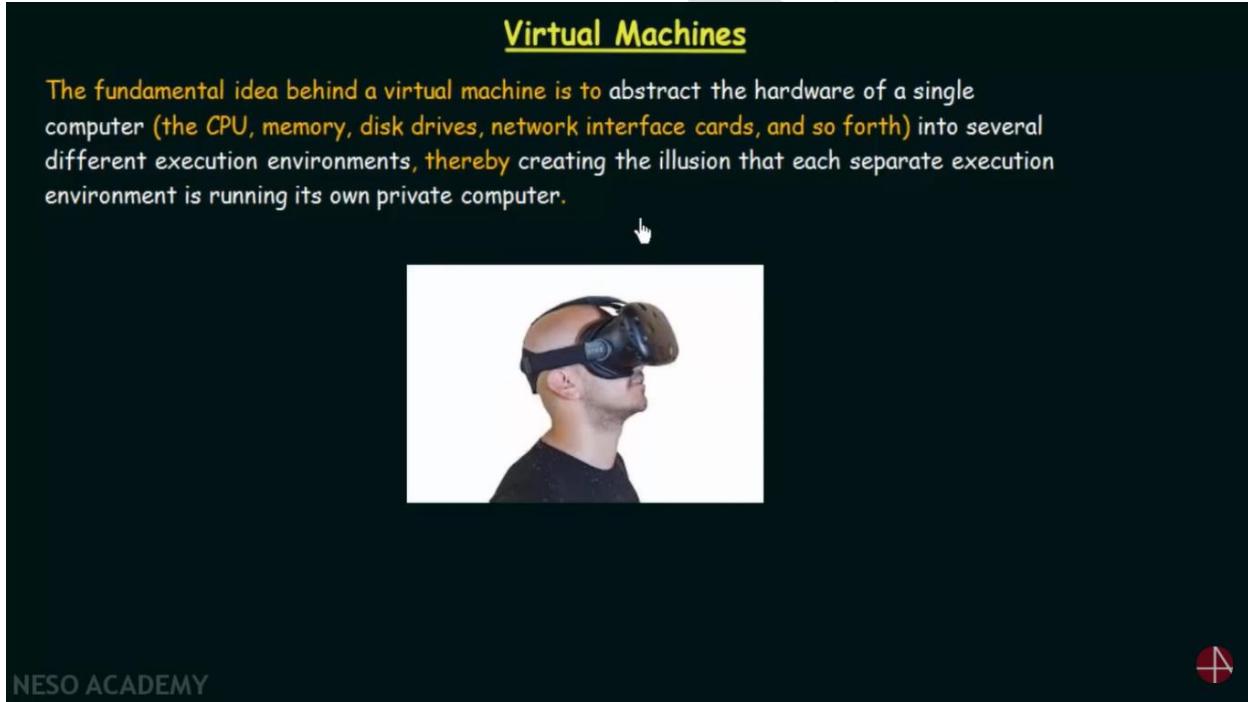
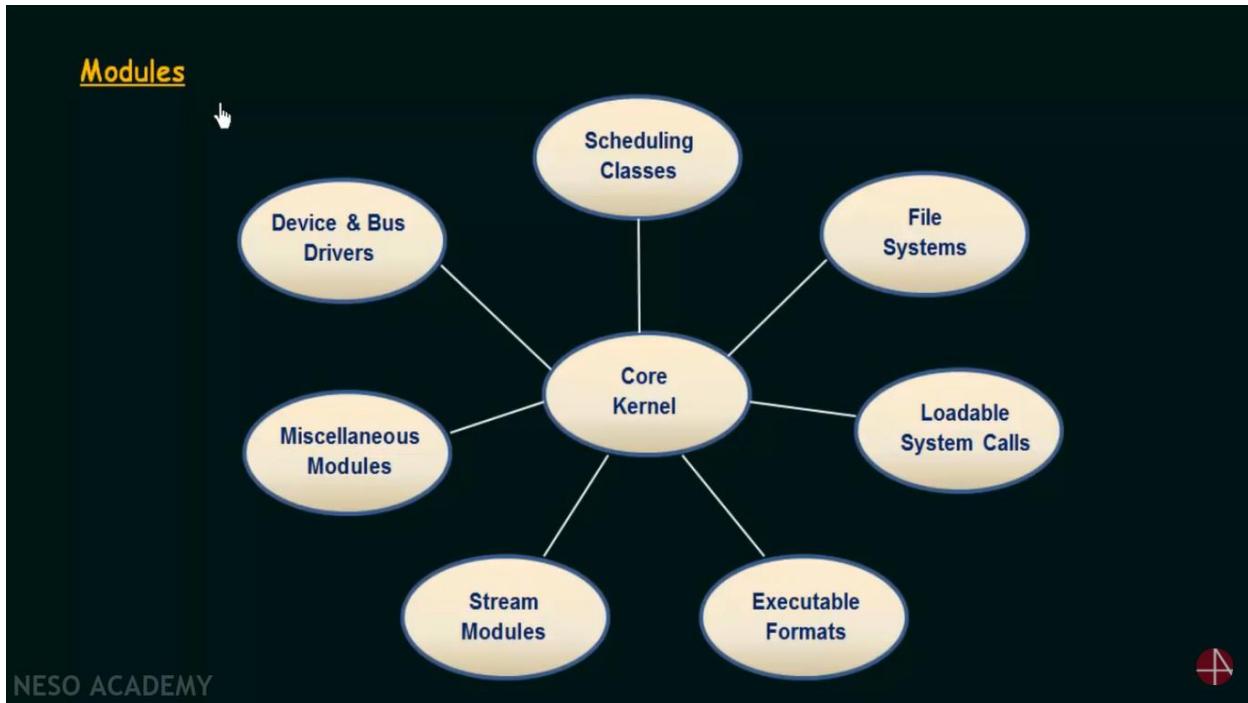
### Microkernels



NESO ACADEMY

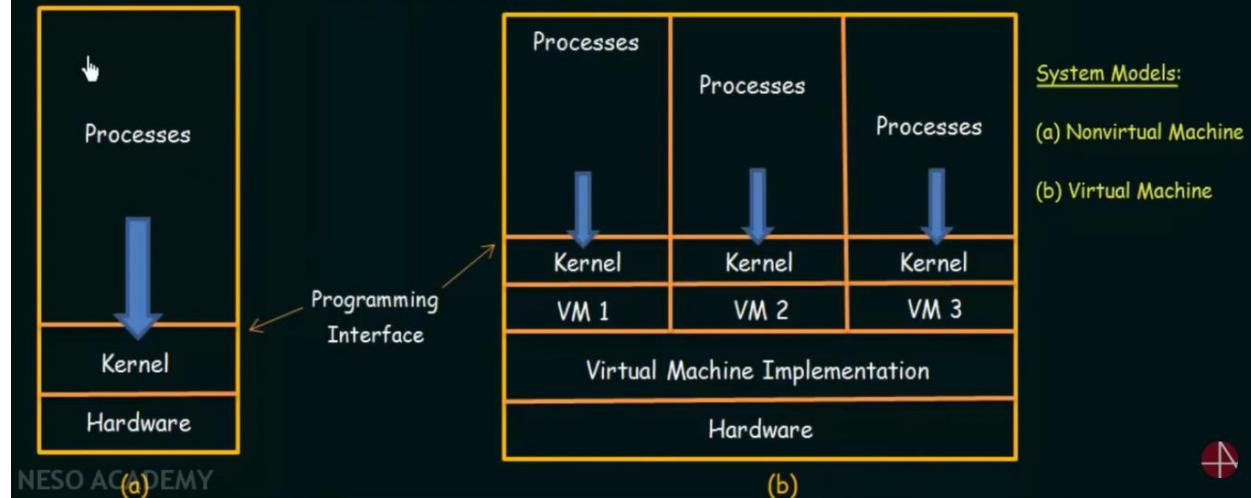


NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



## Virtual Machines

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.



## IMPLEMENTATION

Virtual Machine Software -  
 Virtual Machine itself -

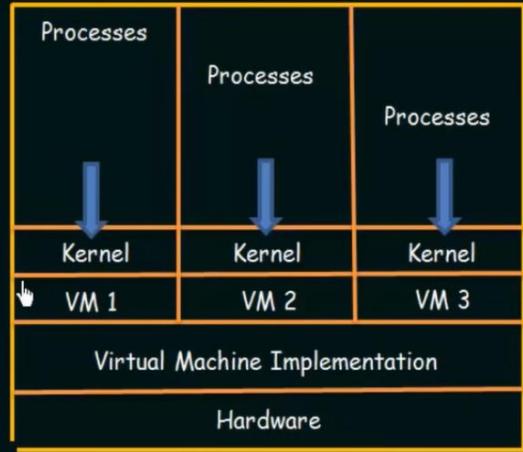
Runs in Kernel mode  
 Runs in User Mode

Just as the physical machine has two modes, however,  
 so must the virtual machine.

Consequently, we must have:

- A virtual user mode and
- A virtual kernel mode

BOTH OF WHICH RUN IN A PHYSICAL USER MODE



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### Operating System Generation & System Boot

- Design, code, and implement an operating system specifically for one machine at one site
- Operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations
- The system must then be configured or generated for each specific computer site, a process sometimes known as system generation (SYSGEN) is used for this

The following kinds of information must be determined by the SYSGEN Program:

- What CPU is to be used?
- How much memory is available?
- What devices are available?
- What operating-system options are desired?

NESO ACADEMY



### System Boot

- The procedure of starting a computer by loading the kernel is known as booting the system.
- On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel
- This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus.



### **Firmware**

NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### System Boot

- The procedure of starting a computer by loading the kernel is known as booting the system.
- On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel
- This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus.



**EPROM**  
Erasable Programmable Read Only Memory

NESO ACADEMY



### System Boot

- The procedure of starting a computer by loading the kernel is known as booting the system.
- On most computer systems, a small piece of code known as the bootstrap program or bootstrap loader locates the kernel
- This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus.

When the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution.

It is only at this point that the system is said to be

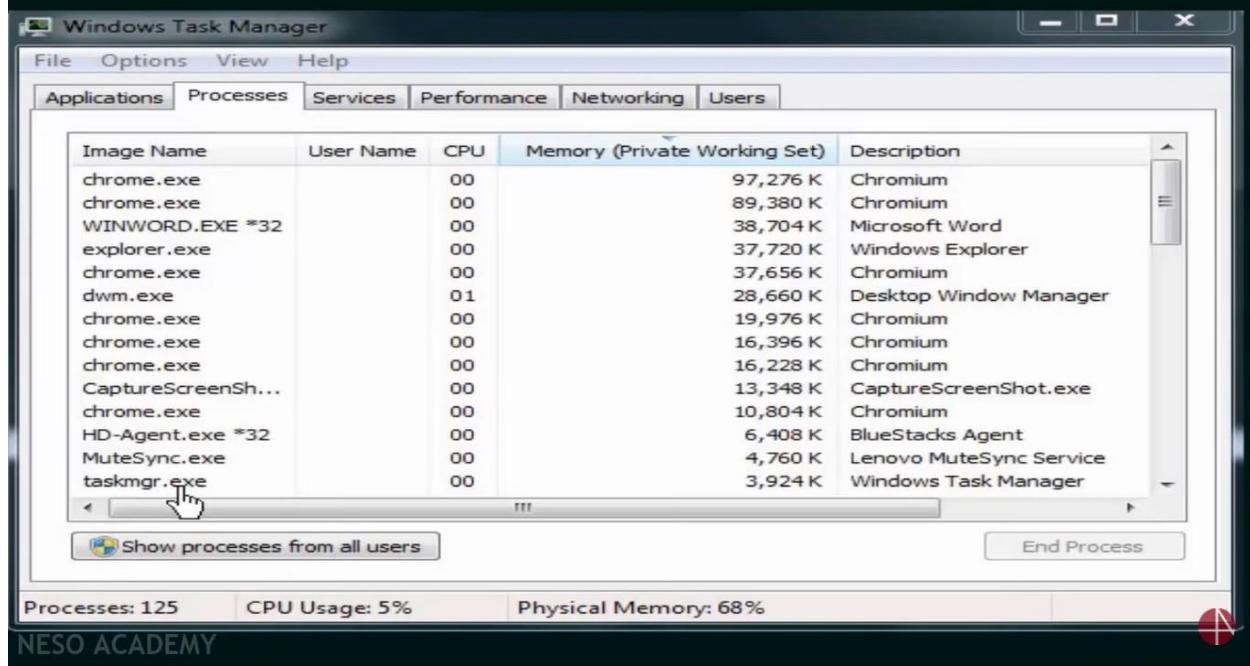
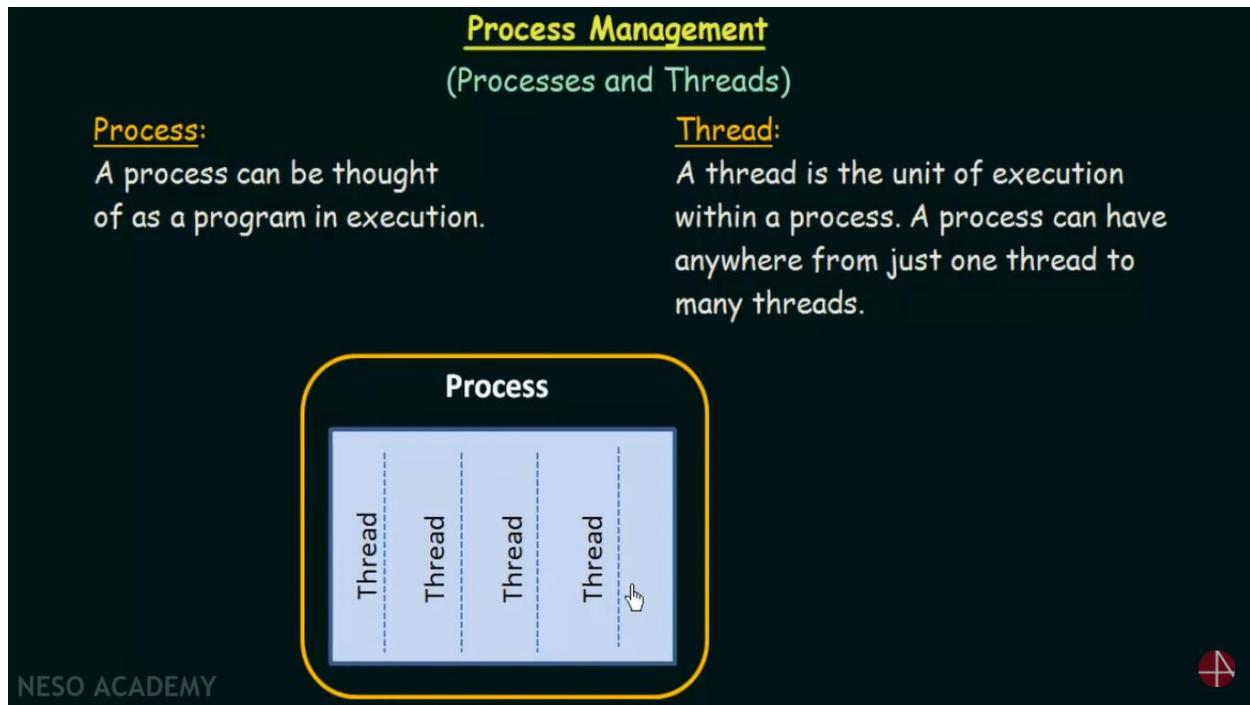


**RUNNING**

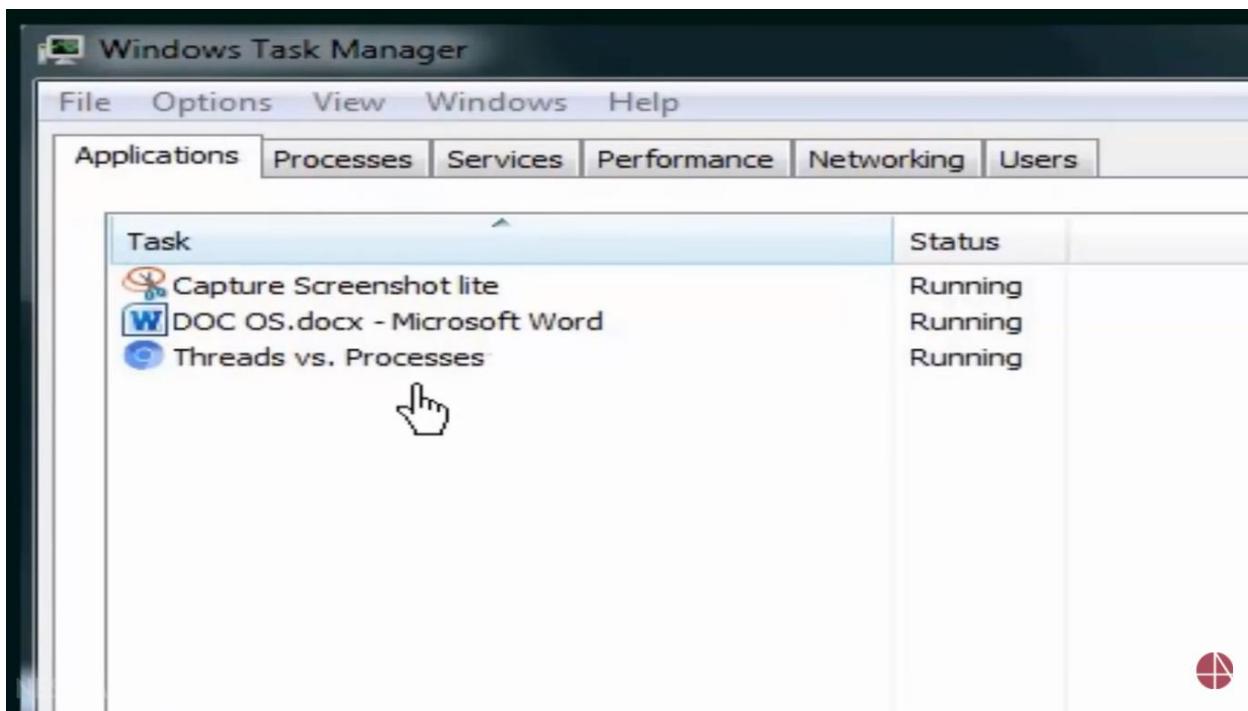
NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



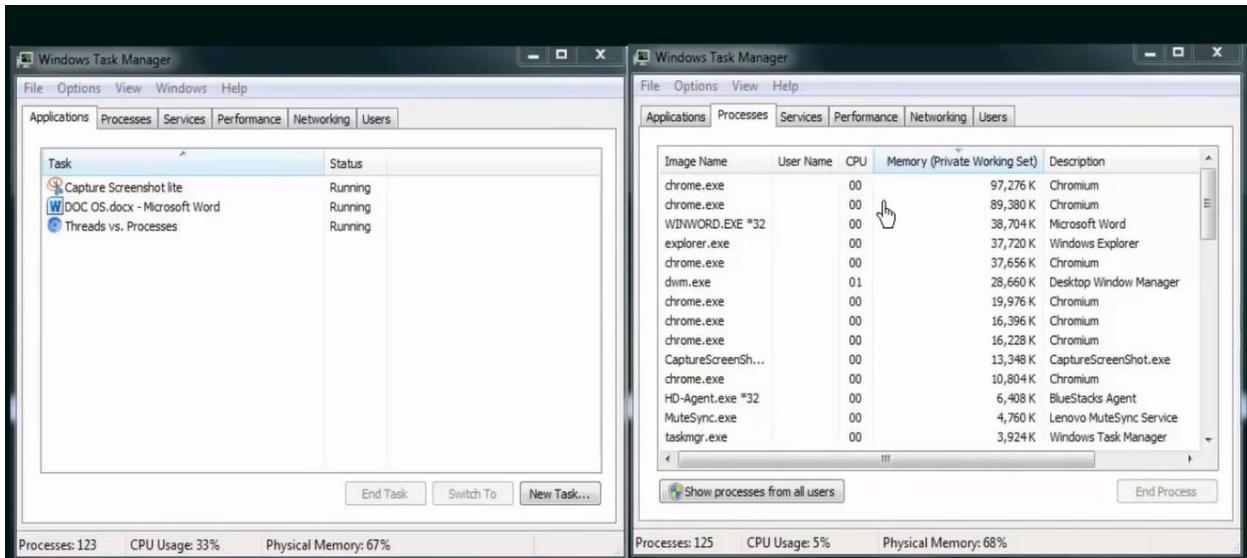
NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



The screenshot shows the Windows Task Manager window with the title bar "Windows Task Manager". The menu bar includes File, Options, View, and Help. Below the menu is a tab bar with Applications, Processes, Services, Performance, Networking, and Users. The Processes tab is selected, displaying a table with columns "Image Name", "User Name", "CPU", "Memory (Private Working Set)", and "Description". The table lists various processes and their details. At the bottom of the table are buttons for "Show processes from all users" and "End Process".

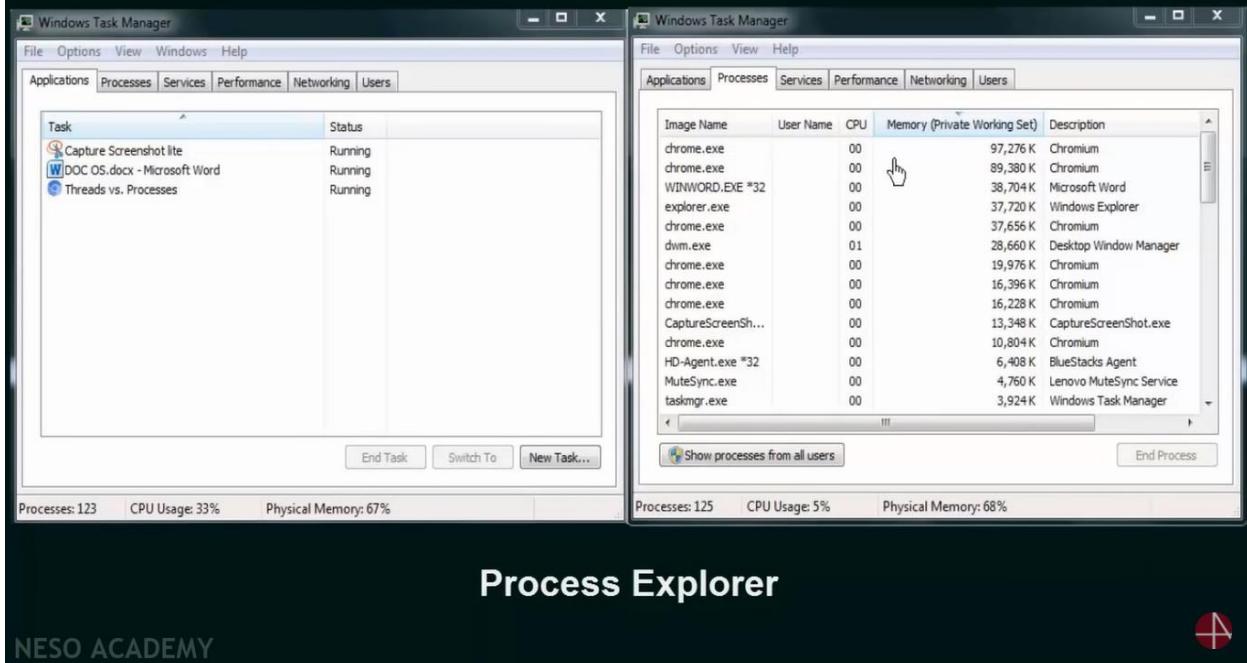
Image Name	User Name	CPU	Memory (Private Working Set)	Description
chrome.exe		00	97,276 K	Chromium
chrome.exe		00	89,380 K	Chromium
WINWORD.EXE		00	38,704 K	Microsoft Word
explorer.exe		00	37,720 K	Windows Explorer
chrome.exe		00	37,656 K	Chromium
dwm.exe		01	28,660 K	Desktop Window Manager
chrome.exe		00	19,976 K	Chromium
chrome.exe		00	16,396 K	Chromium
chrome.exe		00	16,228 K	Chromium
CaptureScreenSh...		00	13,348 K	CaptureScreenShot.exe
chrome.exe		00	10,804 K	Chromium
HD-Agent.exe *32		00	6,408 K	BlueStacks Agent
MuteSync.exe		00	4,760 K	Lenovo MuteSync Service
taskmgr.exe		00	3,924 K	Windows Task Manager

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

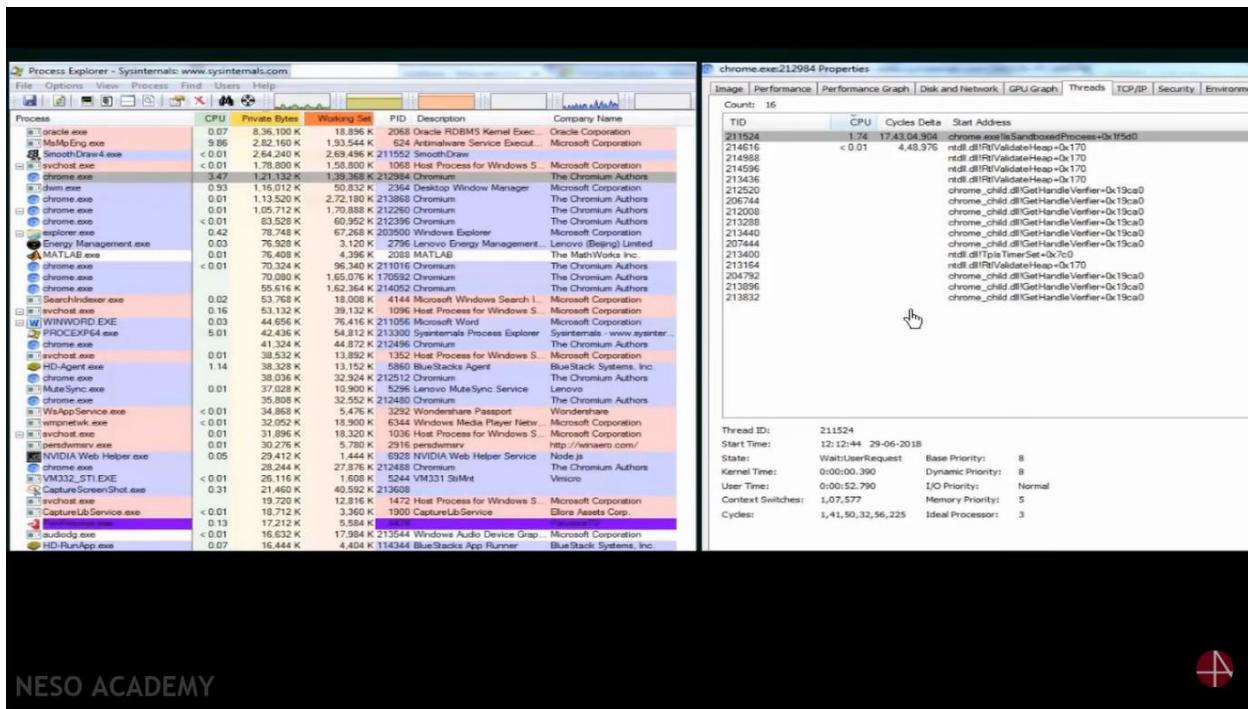


## Process Explorer

NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



NESO ACADEMY

### Process State

- ❖ As a process executes, it changes state.
  - ❖ The state of a process is defined in part by the current activity of that process.
- Each process may be in one of the following states:

**NEW**

The process is being created.

**RUNNING**

Instructions are being executed.

**WAITING**

The process is waiting for some event to occur  
 (Such as an I/O completion or reception of a signal).

**READY**

The process is waiting to be assigned to a processor.

**TERMINATED**

The process has finished execution.

NESO ACADEMY

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

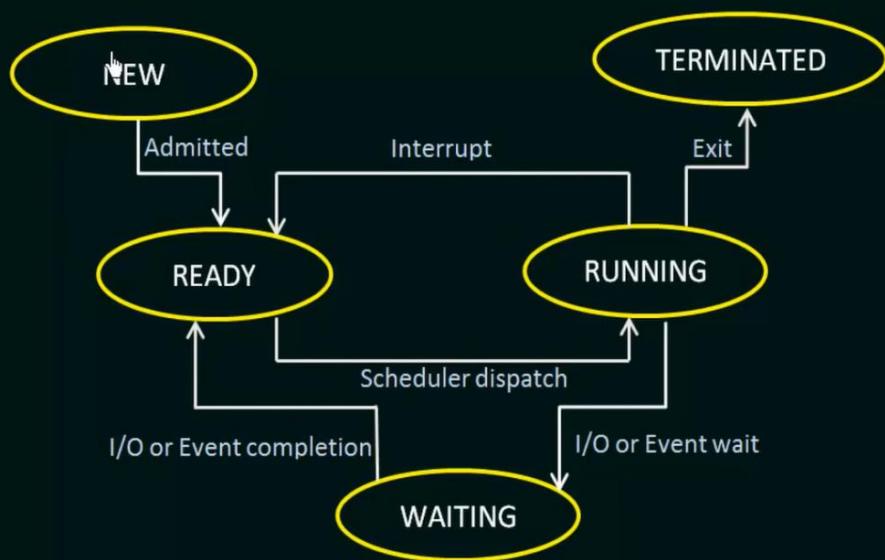


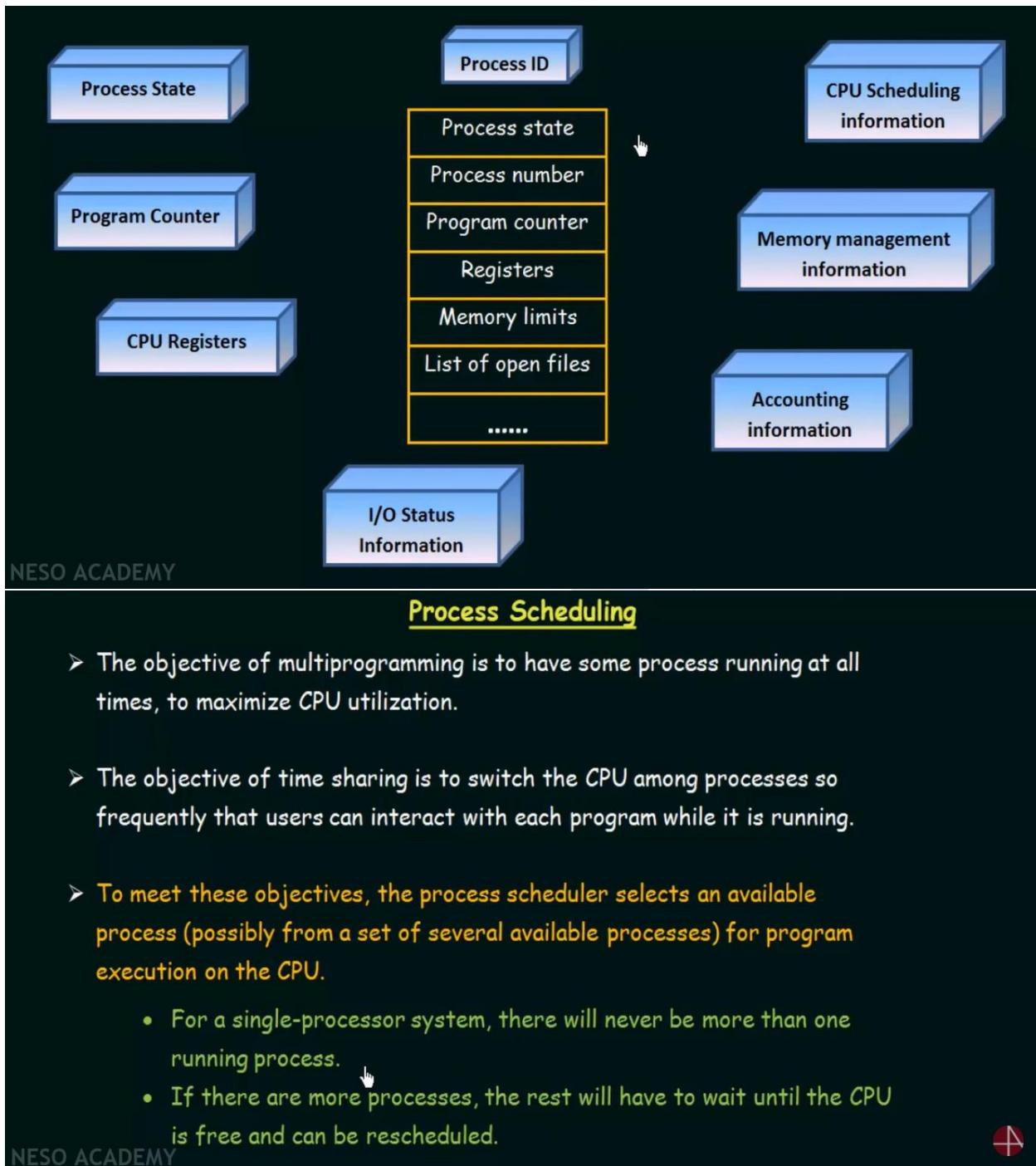
Fig: Diagram of process state



### Process Control Block

Each process is represented in the operating system by a **Process Control Block** (PCB) — also called a task control block.





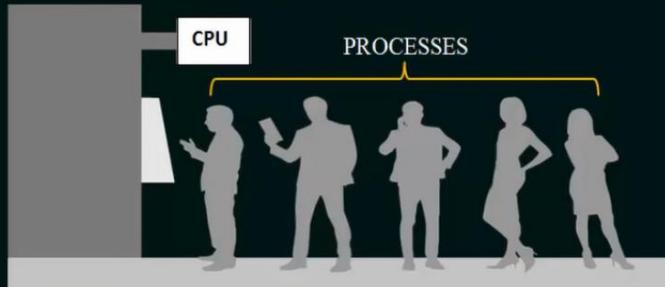
### Scheduling Queues

JOB QUEUE

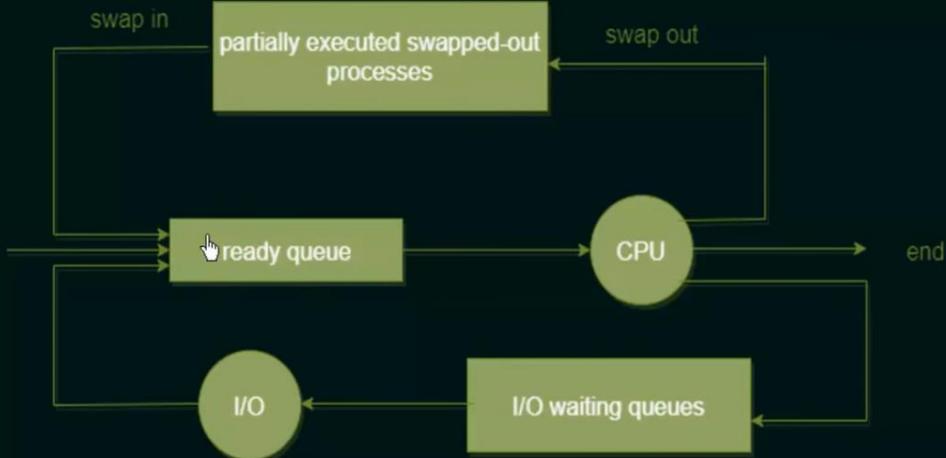
As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.

READY QUEUE

The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.



NESO ACADEMY



NESO ACADEMY

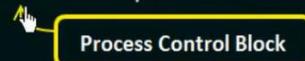


### Context Switch

- Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine.
- Such operations happen frequently on general-purpose systems.

When an interrupt occurs, the system needs to save the current **context** of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.

- The context is represented in the PCB of the process



NESO ACADEMY



Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process.



State Save



State Restore

This task is known as a **context switch**.

- Context-switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).
- Typical speeds are a few milliseconds.

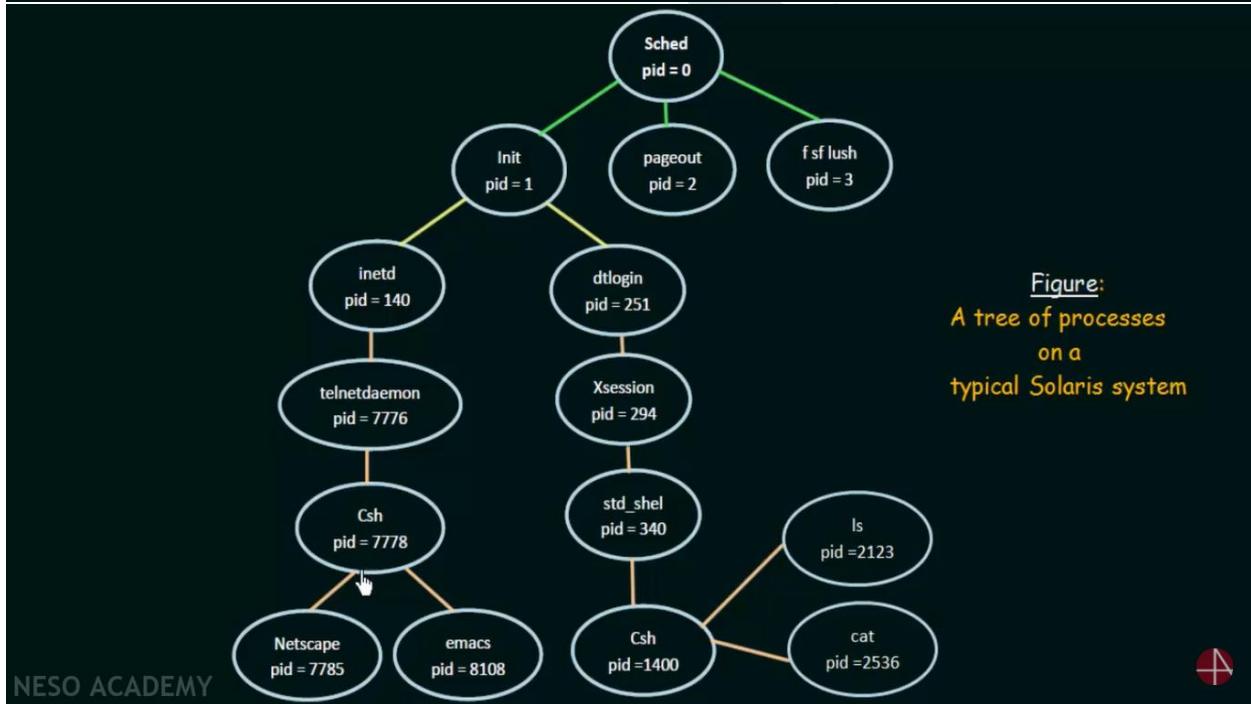
NESO ACADEMY



## Operations on Processes (Process Creation)

- A process may create several new processes, via a create-process system call, during the course of execution.
- The **creating process** is called a **parent process**, and the new processes are called the **children of that process**.
- Each of these new processes may in turn create other processes, forming a **tree of processes**.

NESO ACADEMY



When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.



### Operations on Processes (Process Termination)

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call.
- At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.



Termination can occur in other circumstances as well:

- A process can cause the termination of another process via an appropriate system call.
- Usually, such a system call can be invoked only by the parent of the process that is to be terminated. ↗
- Otherwise, users could arbitrarily kill each other's jobs.



A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (*To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.*)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. ↗



### Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or **cooperating processes**.

**Independent processes** - They cannot affect or be affected by the other processes executing in the system.

**Cooperating processes** - They can affect or be affected by the other processes executing in the system.

Any process that shares data with other processes is a cooperating process.



There are several reasons for providing an environment that allows process cooperation:

Information sharing

Computation speedup

Modularity

Convenience

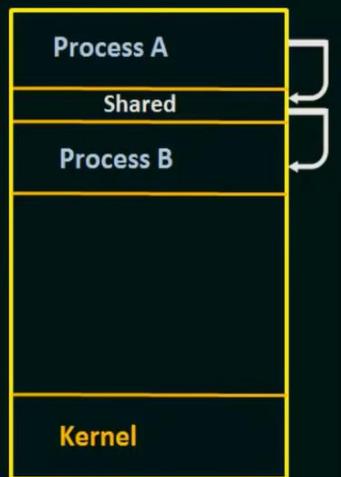


Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

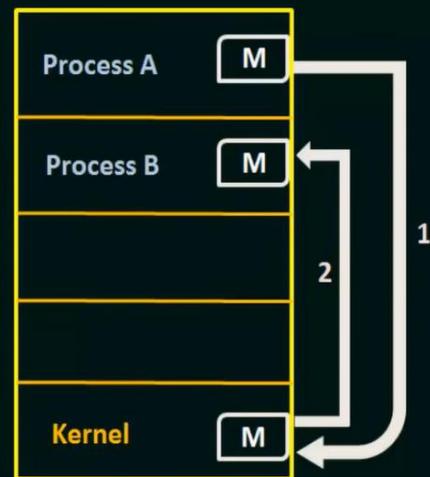
There are two fundamental models of interprocess communication:

- (1) Shared memory
- (2) Message passing

- ❖ In the shared-memory model, a region of memory that is shared by cooperating processes is established.  
Processes can then exchange information by reading and writing data to the shared region.
- ❖ In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.



(a)



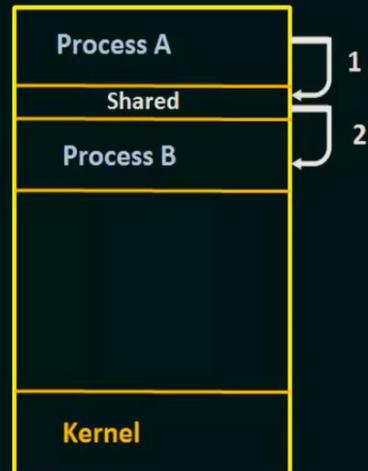
(b)

Fig: Communications models, (a) Shared memory, (b) Message Passing.



## Shared Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction. 



## Producer Consumer Problem

A producer process produces information that is consumed by a consumer process.

For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. 



- One solution to the producer-consumer problem uses **shared memory**.
- To allow producer and consumer processes to run concurrently, we must have available a **buffer of items** that can be filled by the producer and emptied by the consumer.
- This **buffer will reside in a region of memory** that is **shared** by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The **producer and consumer must be synchronized**, so that the consumer does not try to consume an item that has not yet been produced.



#### Two kinds of buffers:

Unbounded buffer

Places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

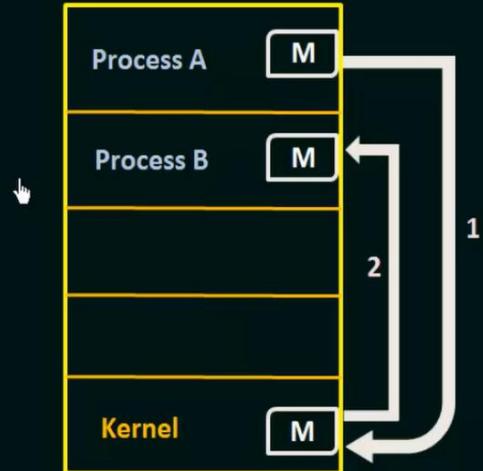
Bounded buffer

Assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.



## Message-Passing Systems (Part-1)

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.



A message-passing facility provides at least two operations:

- send (message)  
and
- receive (message)



Messages sent by a process can be of either fixed or variable size.

FIXED SIZE: The system-level implementation is straightforward.  
But makes the task of programming more difficult.

VARIABLE SIZE: Requires a more complex system-level implementation.  
But the programming task becomes simpler.



If processes P and Q want to communicate, they must send messages to and receive messages from each other.



A communication link must exist between them.

This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the send() /receive( ) operations, like:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

There are several issues related with features like:  
➤ Naming  
➤ Synchronization  
➤ Buffering



### Message-Passing Systems (Part-2)

If processes P and Q want to communicate, they must send messages to and receive messages from each other.



A communication link must exist between them.

This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the send() /receive( ) operations, like:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

There are several issues related with features like:  
➤ Naming  
➤ Synchronization  
➤ Buffering



### Naming

Processes that want to communicate must have a way to refer to each other.  
They can use either **direct** or **indirect** communication.

**Under direct communication-** Each process that wants to communicate must explicitly name the recipient or sender of the communication.

- send (P, message) - Send a message to process P.
- receive (Q, message) - Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

NESO ACADEMY

This scheme exhibits **symmetry** in addressing; that is, both the sender process and the receiver process must name the other to communicate.

**Another variant of Direct Communication-** Here, only the sender names the recipient; the recipient is not required to name the sender.

- send (P, message) - Send a message to process P.
- receive (id, message) - Receive a message from any process; *the variable id is set to the name of the process with which communication has taken place.*

This scheme employs **asymmetry** in addressing.

The **disadvantage** in both of these schemes (symmetric and asymmetric) is the **limited modularity** of the resulting process definitions.

Changing the identifier of a process may necessitate examining all other process definitions.

NESO ACADEMY



**With indirect communication:**

The messages are sent to and received from **mailboxes**, or ports.



- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each **mailbox** has a **unique identification**.
- Two processes can communicate only if the processes have a shared mailbox
  - send (A, message) — Send a message to mailbox A.
  - receive (A, message) — Receive a message from mailbox A.

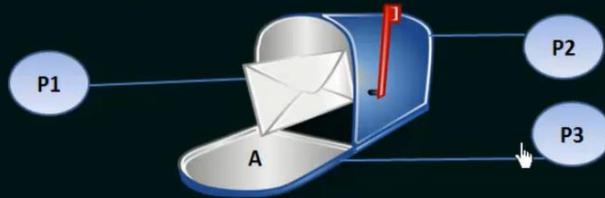


**A communication link in this scheme has the following properties:**

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox. 



Now suppose that processes P1, P2, and P3 all share mailbox A



Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1?



Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1?

The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive () operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, round robin where processes take turns receiving messages). The system may identify the receiver to the sender.

A **mailbox** may be **owned** either by a **process** or by the **operating system**.



### Synchronization

Communication between processes takes place through calls to **send()** and **receive ()** primitives. There are different design options for implementing each primitive.

Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.



Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous**.

**Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.

**Nonblocking send:** The sending process sends the message and resumes operation.

**Blocking receive:** The receiver blocks until a message is available.



**Nonblocking receive:** The receiver retrieves either a valid message or a null.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

The queue has a maximum length of  $\infty$

Whether communication cannot have any messages waiting in its exchanged by communicating ender must block until the recipient my queue.

Basically, message.

Three ways:



### Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.

Basically, such queues can be implemented in three ways:



**Zero capacity:** The queue has a maximum length of **zero**; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

**Bounded capacity:** The queue has finite **length n**; thus, **at most n messages can reside in it**. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.



**Zero capacity:** The queue has a maximum length of **zero**; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

**Bounded capacity:** The queue has finite **length n**; thus, **at most n messages can reside in it**. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

**Unbounded capacity:** The queues **length is potentially infinite**; thus, any number of messages can wait in it. The sender never blocks.



## Sockets

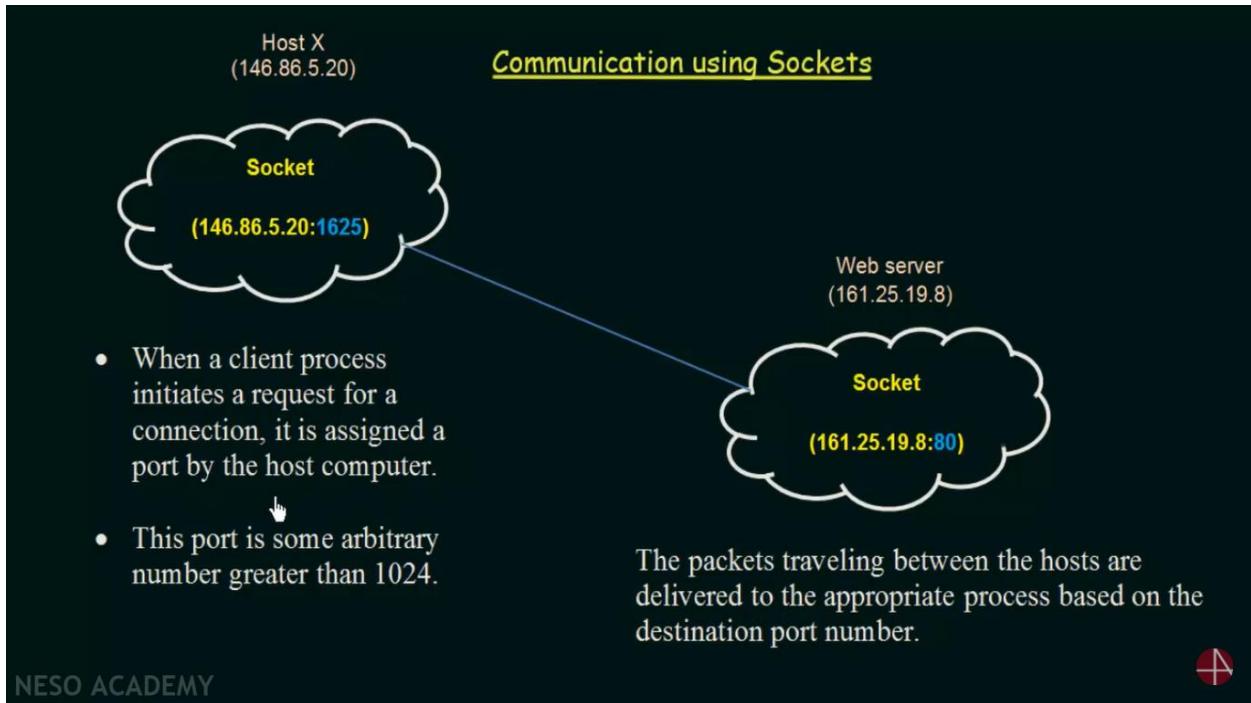
Used for communication in Client-Server Systems

- A socket is defined as an endpoint for communication.
- A pair of processes communicating over a network employ a pair of sockets—one for each process.
- A socket is identified by an IP address concatenated with a port number.
- The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. 



- A socket is identified by an IP address concatenated with a port number.
- The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.
- Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports   
(a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80).
- All ports below 1024 are considered well known; we can use them to implement standard services 

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



### Remote Procedure Calls (RPC)

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details.

- It is similar in many respects to the IPC mechanism.
- However, because we are dealing with an environment in which the processes are executing on separate systems, we must use a **message based communication scheme** to provide remote service.
- In contrast to the IPC facility, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data.
- Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function.
- The function is then executed as requested, and any output is sent back to the requester in a separate message. 



The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally

- The RPC system hides the details that allow communication to take place by providing a stub on the client side.
- Typically, a separate stub exists for each separate remote procedure.
- When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and marshals the parameters.
- Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network.
- The stub then transmits a message to the server using message passing.
- A similar stub on the server side receives this message and invokes the procedure on the server. 
- If necessary, return values are passed back to the client using the same technique.



### Issues in RPC and how they are resolved

Issues	How they are resolved
<p>Differences in data representation on the client and server machines. Eg. Representation of 32-bit integers: Some systems (known as big-endian) use the high memory address to store the most significant byte, while other systems (known as little-endian) store the least significant byte at the high memory address.</p>	<p>RPC systems define a machine-independent representation of data. One such representation is known as external data representation (XDR). On the client side, parameter marshalling involves converting the machine dependent data into XDR before they are sent to the server. On the server side, the XDR data are unmarshalled and converted to the machine-dependent representation for the server.</p>

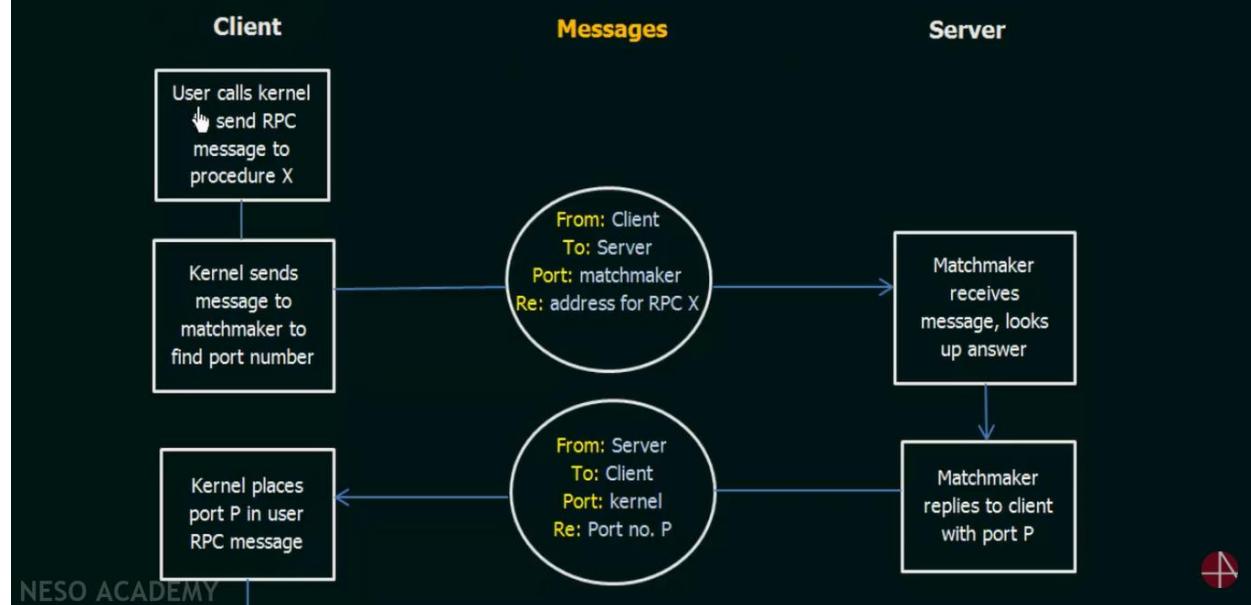


Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors.	The operating system must ensure that messages are acted on exactly once, rather than at most once. Most local procedure calls have the "exactly once" functionality, but it is more difficult to implement.
---	--

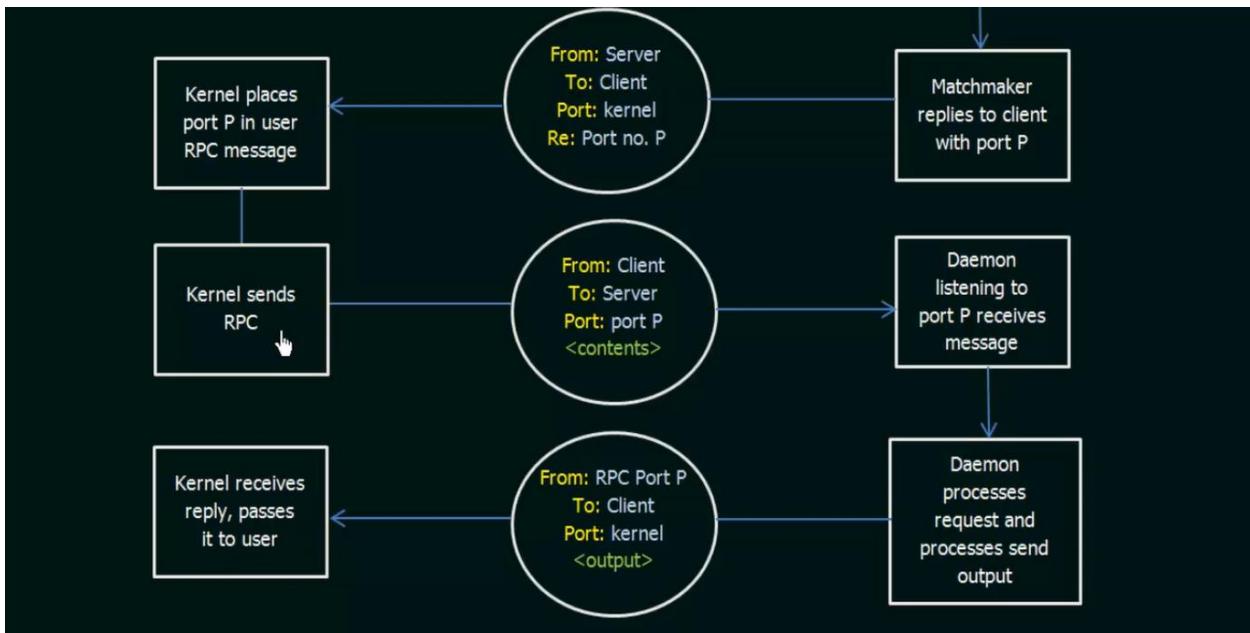


<p>With standard procedure calls, some form of binding takes place during link, load, or execution time so that a <u>procedure call's name is replaced by the memory address of the procedure call</u>. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other because they do not share memory.</p>	<p>1) The binding information may be predetermined, in the form of <b>fixed port addresses</b>. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service.</p> <p>2) Binding can be done dynamically by a <b>rendezvous mechanism</b>. Typically, an operating system provides a rendezvous (also called a <b>matchmaker</b>) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes).</p>
NESO ACADEMY	4

### Execution of a remote procedure call (RPC)



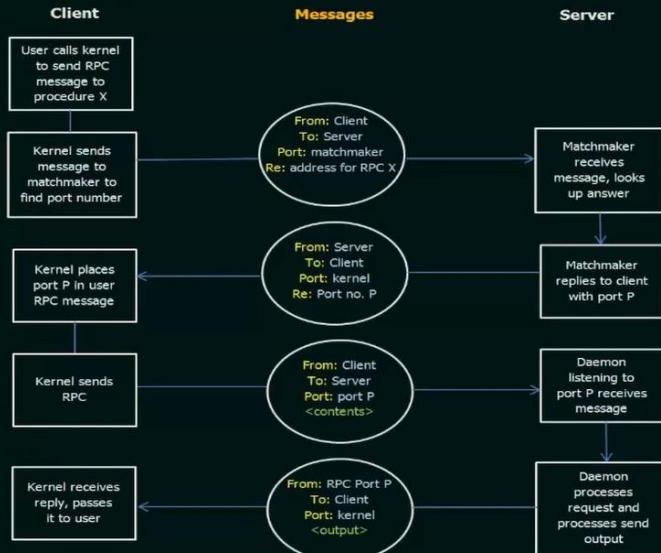
NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



NESO ACADEMY



#### Execution of a remote procedure call (RPC)



NESO ACADEMY



## Threads

A thread is a basic unit of CPU utilization.

It comprises

A thread ID

A program counter

A register set

and A stack

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional / heavyweight process has a single thread of control.

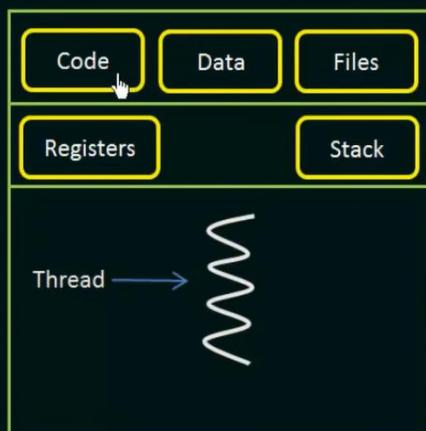
If a process has multiple threads of control, it can perform more than one task at a time.



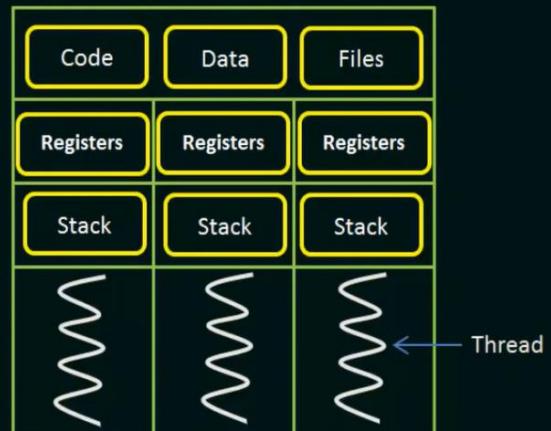
NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional / heavyweight process has a single thread of control.  
If a process has multiple threads of control, it can perform more than one task at a time.



NESO ACADEMY



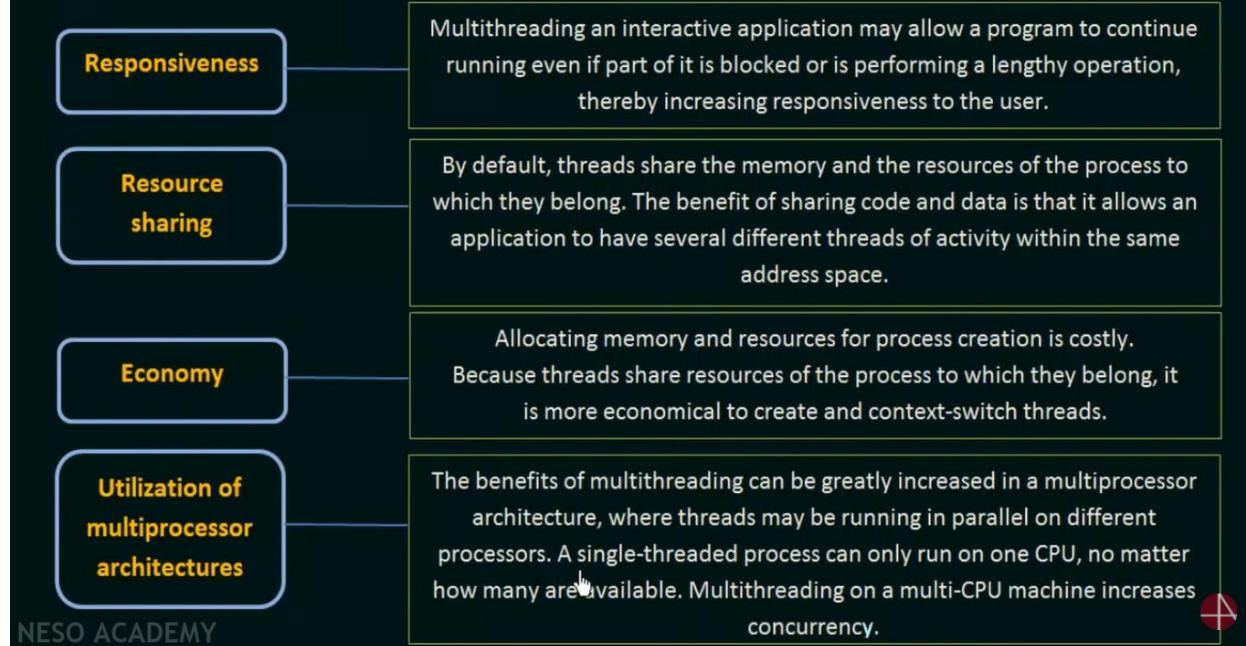
Multi-threaded process

ProcessThreadsView - C:\Users\JAISON\AppData\Local\Chromium\Application\

File	Edit	View	Options	Help
Thread ID	/	Context Switches	Last Context S...	Status
9780	9780	3	0	UserRequest
10908	10908	3	0	
13400	13400	2	0	UserRequest
13624	13624	2	0	
13788	13788	2	0	
13828	13828	413	0	WrQueue
13848	13848	9	0	UserRequest
13892	13892	3	0	WrQueue
13960	13960	4	0	UserRequest
14036	14036	2,540	0	UserRequest
14304	14304	2	0	UserRequest

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

The benefits of multithreaded programming can be broken down into four major categories:



NESO ACADEMY



### Multithreading Models and Hyperthreading

#### Types of Threads:

- 1) **User Threads** - Supported above the kernel and are managed without kernel support.
- 2) **Kernel Threads** - Supported and managed directly by the operating system.

Ultimately, there must exist a relationship between user threads and kernel threads.

There are three common ways of establishing this relationship:

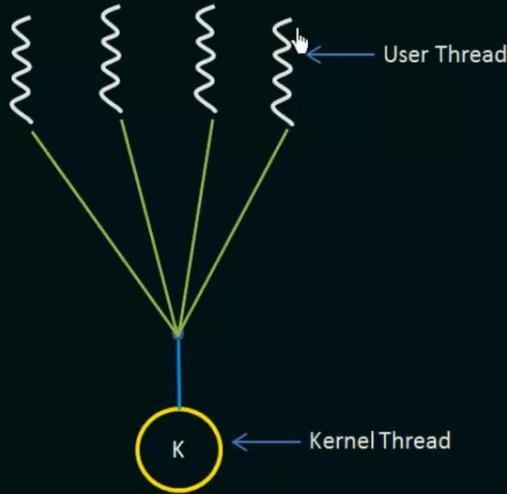


1. Many-to-One Model
2. One-to-One Model
3. Many-to-Many Model

NESO ACADEMY



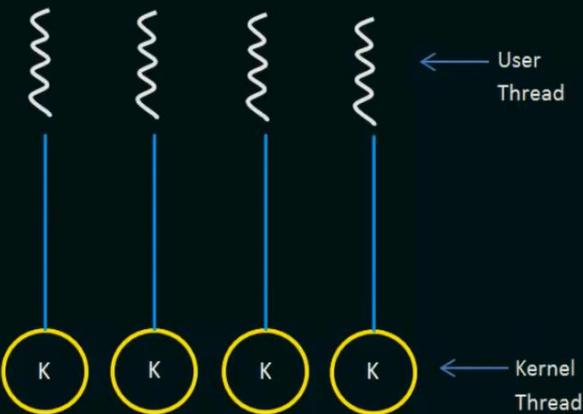
### Many-to-One Model



- Maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- The entire process will block if a thread makes a blocking system call.
- Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

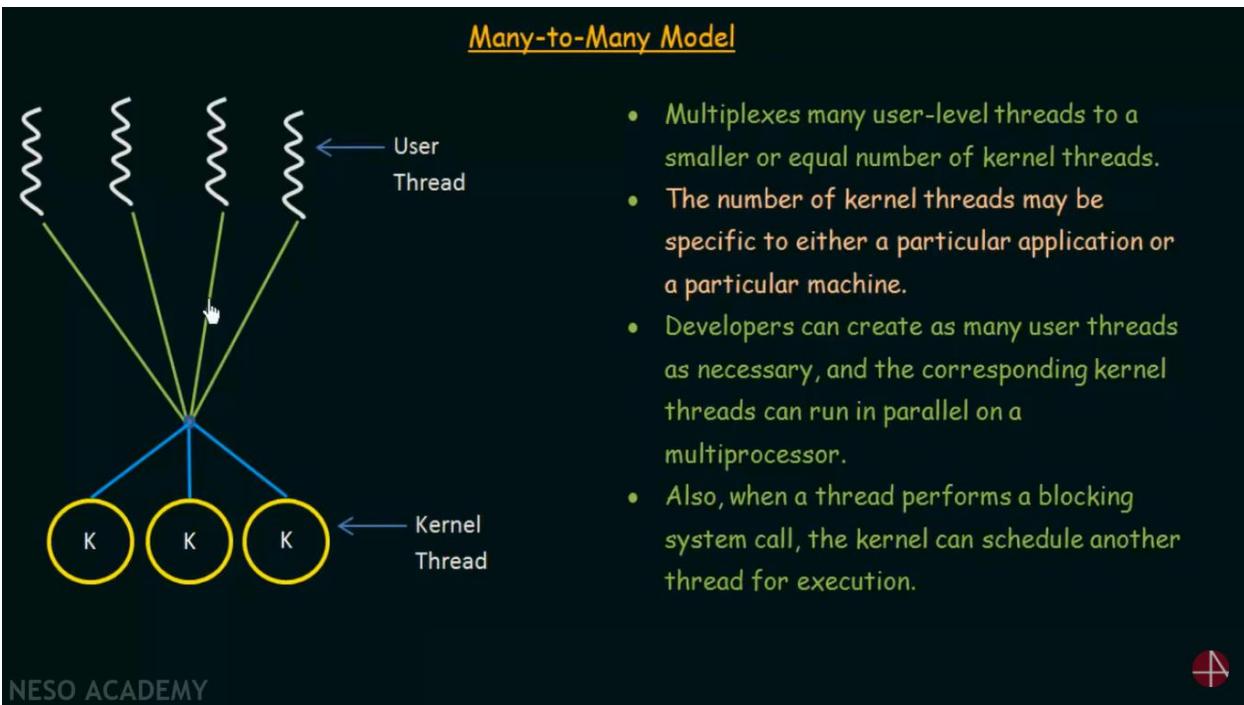


### One-to-One Model



- Maps each user thread to a kernel thread.
- Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call;
- Also allows multiple threads to run in parallel on multiprocessors.
- Creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.





NESO ACADEMY



Hyperthreading

or

Simultaneous Multithreading (SMT)

Hyperthreaded systems allow their processor cores' resources to become multiple logical processors for performance.



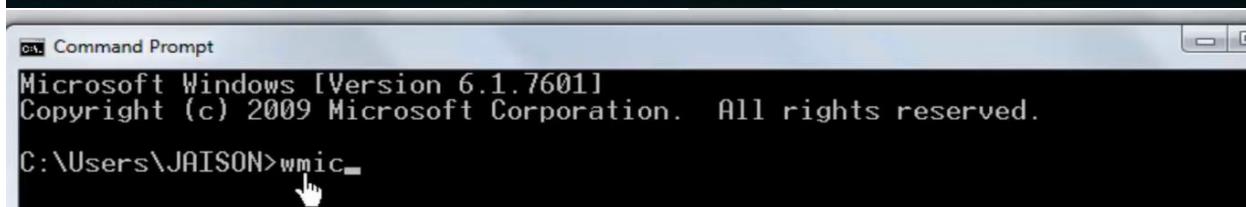
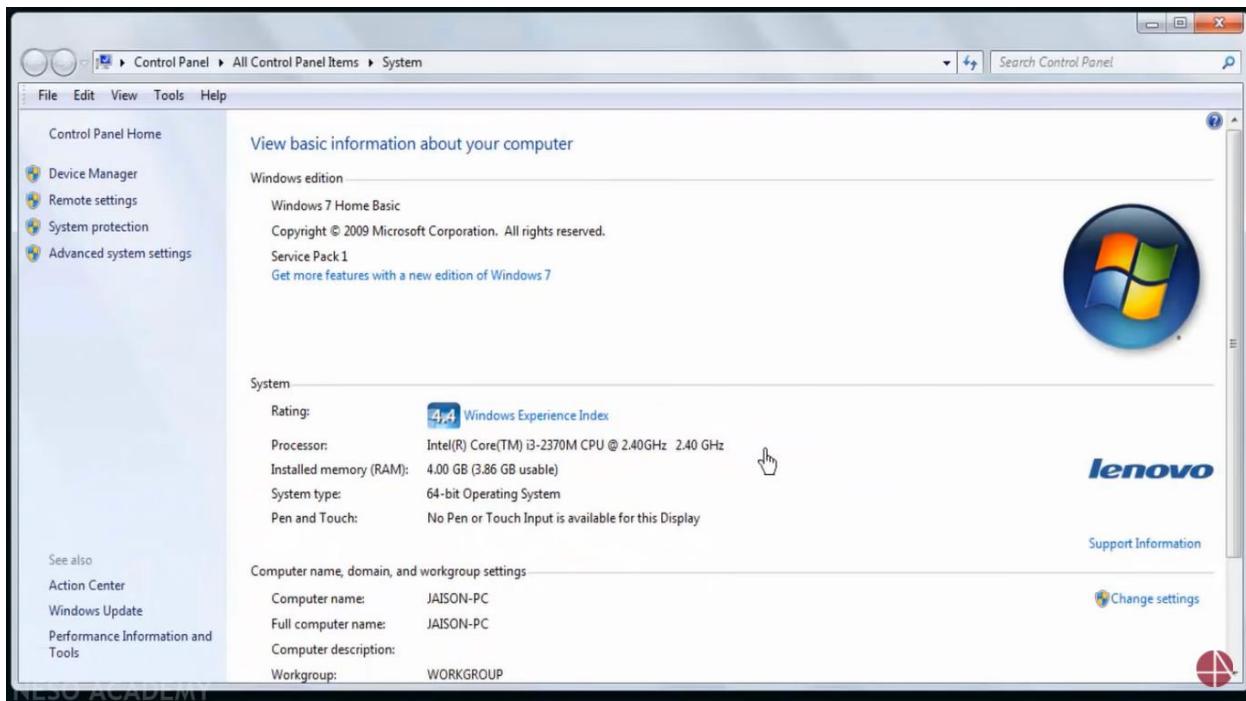
It enables the processor to execute two threads, or sets of instructions, at the same time. Since hyper-threading allows two streams to be executed in parallel, it is almost like having two separate processors working together.



NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



## WMI - Windows Management Instrumentation

It is a management infrastructure that provides access to control over a system.

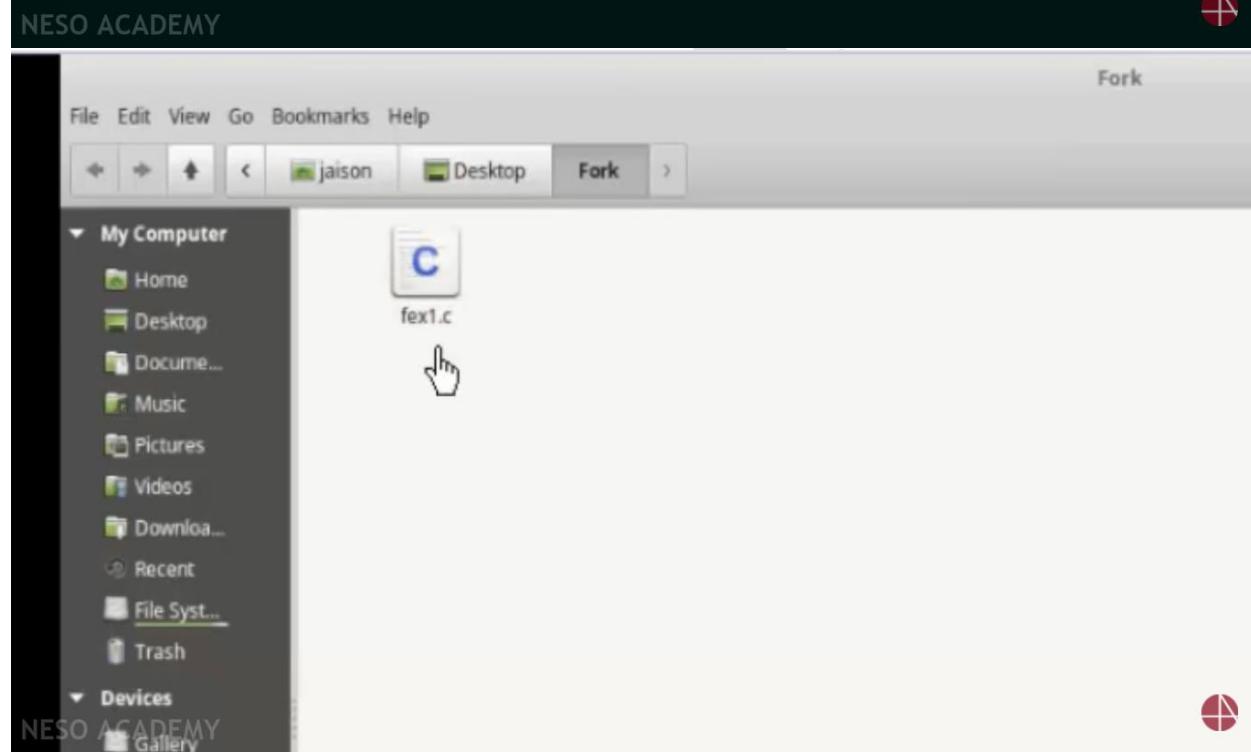


NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### The fork() and exec() System Calls

fork() : The **fork()** system call is used to create a **separate, duplicate process**.

exec() : When an **exec()** system call is invoked, the program specified in the parameter to **exec()** will **replace the entire process** – including all threads.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

The image shows a dual-monitor setup. The top monitor displays a terminal window titled 'fex1.c (~/Desktop/Fork)'. The bottom monitor also displays a terminal window with the same title. Both windows show the following C code:

```
1#include <stdio.h>
2#include <sys/types.h>
3#include <unistd.h>
4int main()
5{
6    |
7    printf("Hello Neso Academy!\n PID = %d\n", getpid());
8
9    return 0;
10}
```

The code is identical on both monitors. The bottom terminal window has a small 'Fork' window icon in its title bar. The desktop environment includes icons for 'Menu', 'File Manager', 'Terminal', 'Disk', 'Network', 'System', 'Backgrounds', and 'Fork'.

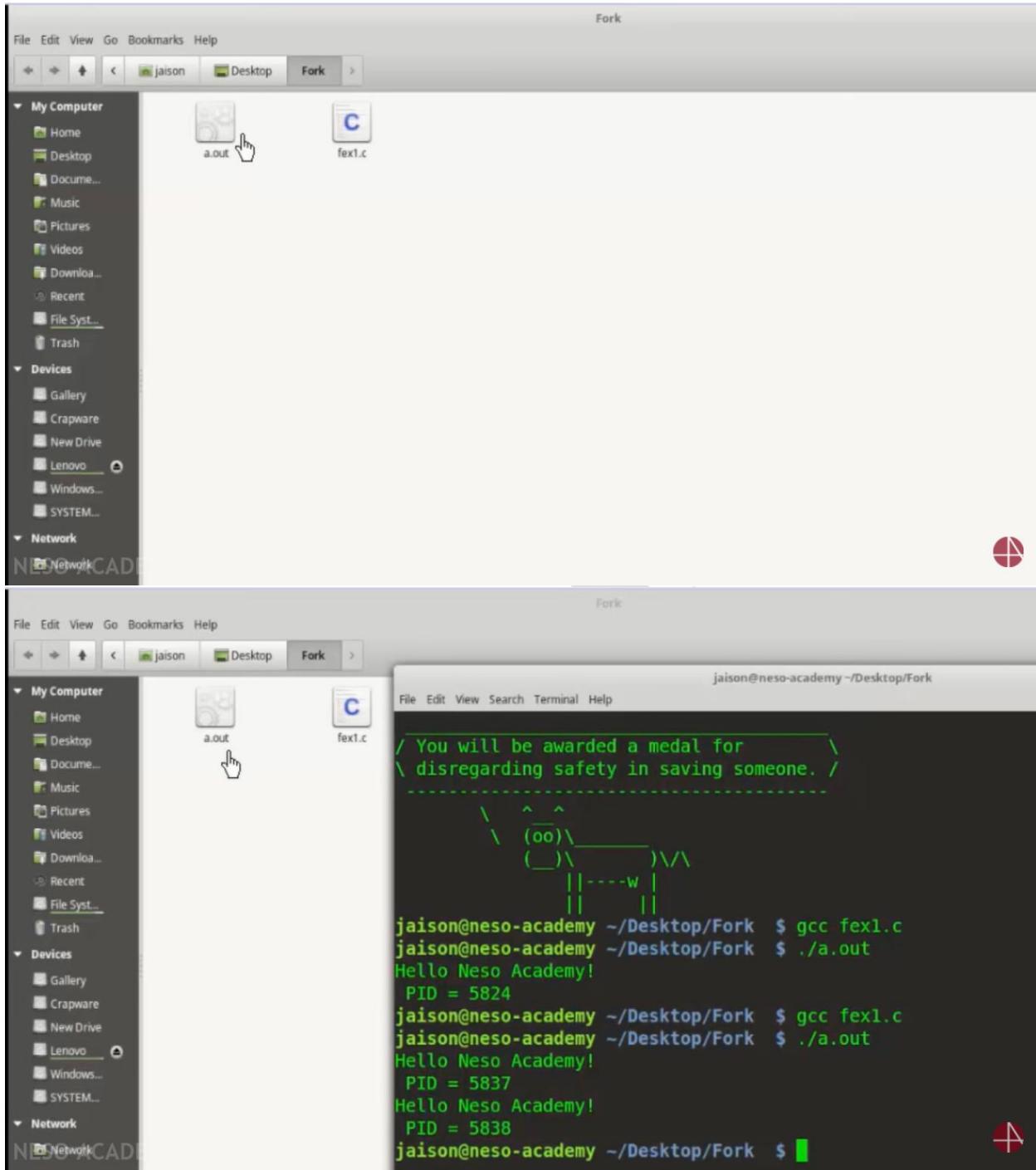
NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

The image shows a terminal window with a C program named `fex1.c`. The code includes standard headers and a main function that prints "Hello Neso Academy!" and returns 0. A note at the top of the code reads: "/ You will be awarded a medal for \ disregarding safety in saving someone. /". The terminal output shows the program running and printing its PID as 5824. The terminal prompt is `jaison@neso-academy ~/Desktop/Fork`.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    printf("Hello Neso Academy!");
    return 0;
}

/ You will be awarded a medal for \
\ disregarding safety in saving someone. /
-----^
\  ^  ^
(oo)\_____
\  (   )\getpid )\/\
 |  - - - w |
 |           |
jaison@neso-academy ~/Desktop/Fork $ gcc fex1.c
jaison@neso-academy ~/Desktop/Fork $ ./a.out
Hello Neso Academy!
PID = 5824
jaison@neso-academy ~/Desktop/Fork $ gcc fex1.c
```

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



NAME – RUSHIKESH SHARAD MANE

SUBJECT- OPERATING SYSTEM(OS)

TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

The image shows a Linux desktop environment with a terminal window and a code editor window.

**Terminal Window:**

```
jaison@neso-academy ~/Desktop/Fork
File Edit View Search Terminal Help
/ You will be awarded a medal for \
\ disregarding safety in saving someone. /
\ \ ^ ^
(oo)\_____
(_)\ \ )\/\
| |---w |
jaison@neso-academy ~/Desktop/Fork $ gcc fexl.c
jaison@neso-academy ~/Desktop/Fork $ ./a.out
Hello Neso Academy!
PID = 5824
jaison@neso-academy ~/Desktop/Fork $ gcc fexl.c
jaison@neso-academy ~/Desktop/Fork $ ./a.out
Hello Neso Academy!
PID = 5837
Hello Neso Academy!
PID = 5838
jaison@neso-academy ~/Desktop/Fork $
```

**Code Editor Window:**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    printf("Hello Neso Academy!\n PID = %d\n", getpid());
    return 0;
}
```

NAME – RUSHIKESH SHARAD MANE

SUBJECT- OPERATING SYSTEM(OS)

TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

The image shows a dual-terminal setup on a Linux desktop environment. Both terminals are titled "fex1.c (~/Desktop/Fork)".

**Terminal 1 (Top):**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello Neso Academy!\n PID = %d\n", getpid());
    return 0;
}
```

**Terminal 2 (Bottom):**

```
/ You will be awarded a medal for \
\ disregarding safety in saving someone. /
-----
 \ ^ ^
 (oo)\_____
 (   )\-----)\/\
      ||----w |
      ||     |
jaison@neso-academy ~/Desktop/Fork $ gcc fex1.c
jaison@neso-academy ~/Desktop/Fork $ ./a.out
Hello Neso Academy!
PID = 5824
jaison@neso-academy ~/Desktop/Fork $ gcc fex1.c
jaison@neso-academy ~/Desktop/Fork $ ./a.out
Hello Neso Academy!
PID = 5837
jaison@neso-academy ~/Desktop/Fork $ gcc fex1.c
jaison@neso-academy ~/Desktop/Fork $ ./a.out
Hello Neso Academy!
PID = 5838
jaison@neso-academy ~/Desktop/Fork $ gcc fex1.c
jaison@neso-academy ~/Desktop/Fork $
```

**File Explorer:**

NESO ACADEMY

**Terminal 3 (Bottom Left):**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello Neso Academy!\n PID = %d\n", getpid());
    return 0;
}
```

**Terminal 4 (Bottom Right):**

```
jaison@neso-academy ~/Desktop/Fork $ ./a.out
Hello Neso Academy!
PID = 5837
Hello Neso Academy!
PID = 5838
jaison@neso-academy ~/Desktop/Fork $ gcc fex1.c
jaison@neso-academy ~/Desktop/Fork $ ./a.out
Hello Neso Academy!
PID = 5850
jaison@neso-academy ~/Desktop/Fork $ PID=5852 pid();
Hello Neso Academy!
PID = 5851
Hello Neso Academy!
PID = 5853
jaison@neso-academy ~/Desktop/Fork $ PID=5854 pid();
Hello Neso Academy!
PID = 5855
jaison@neso-academy ~/Desktop/Fork $ PID=5856 pid();
Hello Neso Academy!
PID = 5854
```

NAME – RUSHIKESH SHARAD MANE

SUBJECT- OPERATING SYSTEM(OS)

TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

File Edit View Search Tools Documents Help

Open Save Undo Fork

fe1.c (~/Desktop/Fork)

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 int main()
5 {
6     fork();
7     fork();
8     fork();
9     printf("Hello Neso Academy!\n PID = ");
10    return 0;
11 }
```

jaison@neso-academy ~/Desktop/Fork \$ ./a.out

Hello Neso Academy!  
PID = 5837  
Hello Neso Academy!  
PID = 5838  
jaison@neso-academy ~/Desktop/Fork \$ gcc fe1.c  
jaison@neso-academy ~/Desktop/Fork \$ ./a.out

Hello Neso Academy!  
PID = 5850  
Hello Neso Academy!  
PID = 5851  
Hello Neso Academy!  
PID = 5852 → pid();  
Hello Neso Academy!  
PID = 5853  
Hello Neso Academy!  
PID = 5854  
Hello Neso Academy!  
PID = 5855  
Hello Neso Academy!  
PID = 5856  
Hello Neso Academy!  
PID = 5857  
jaison@neso-academy ~/Desktop/Fork \$ Hello Neso Academy!

NESO ACADEMY

Total number of Processes =  $2^n$

where  $n$  is the number of fork() system calls

File Edit View Search Tools Documents Help

Open Save Undo

fe1.c (~/Desktop/Fork)

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4 int main()
5 {
6     fork(); ✓
7     fork(); ✓
8     fork(); ✓
9     printf("Hello Neso Academy!\n PID = ");
10    return 0;
11 }
```

jaison@neso-academy ~/Desktop/Fork \$ ./a.out

Hello Neso Academy!  
PID = 5837  
Hello Neso Academy!  
PID = 5838  
jaison@neso-academy ~/Desktop/Fork \$ gcc fe1.c  
jaison@neso-academy ~/Desktop/Fork \$ ./a.out

Hello Neso Academy!  
PID = 5850 ←  
Hello Neso Academy!  
PID = 5851  
Hello Neso Academy!  
PID = 5852 → pid();  
Hello Neso Academy!  
PID = 5853  
Hello Neso Academy!  
PID = 5854  
Hello Neso Academy!  
PID = 5855  
Hello Neso Academy!  
PID = 5856  
Hello Neso Academy!  
PID = 5857  
jaison@neso-academy ~/Desktop/Fork \$ Hello Neso Academy!

Diagram illustrating the process tree:

```
graph LR
    P0((P0)) --> P1((P1))
    P1 --> P2((P2))
    P2 --> P3((P3))
    P2 --> P4((P4))
    P4 --> P5((P5))
    P4 --> P6((P6))
    P5 --> P7((P7))
    P6 --> P8((P8))
```

File Edit View Search Tools Documents Help

Open Save Undo Fork

fe1.c (~/Desktop/Fork)

jaison@neso-academy ~/Desktop/Fork \$ Tab Width: 4 v Ln 8, Col 12

Backgrounds Fork

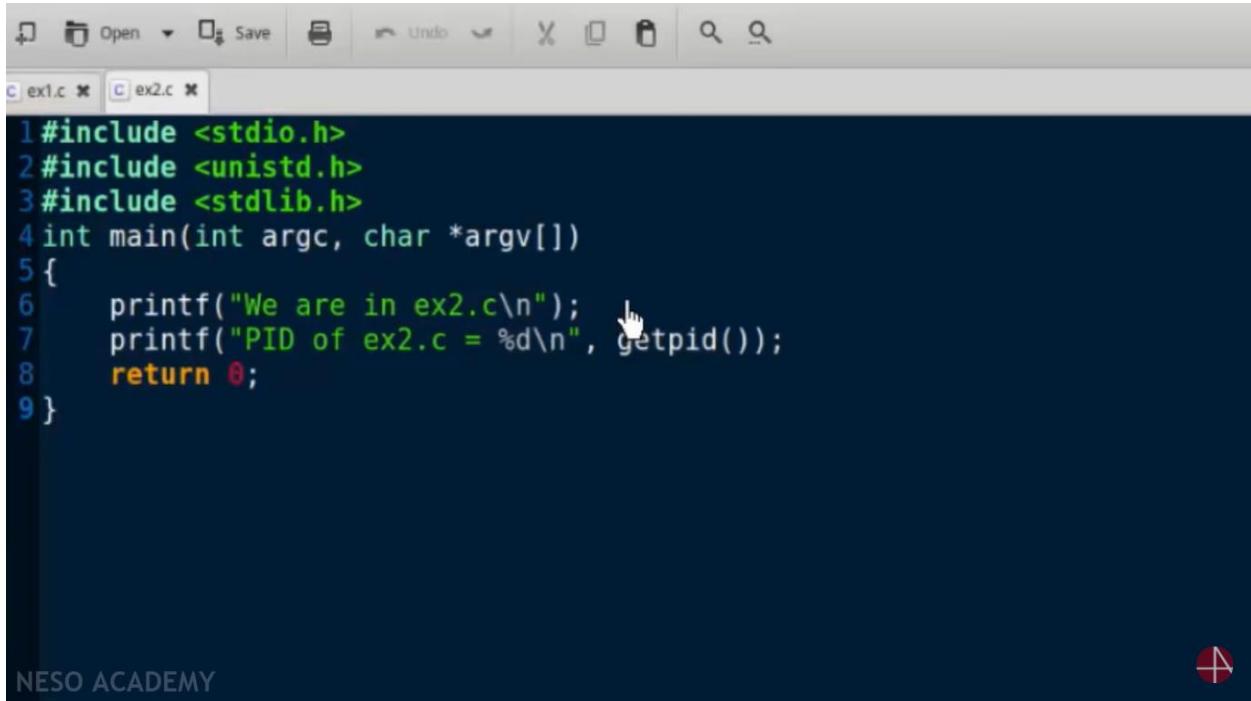
Sunday July 21, 1:30 PM

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

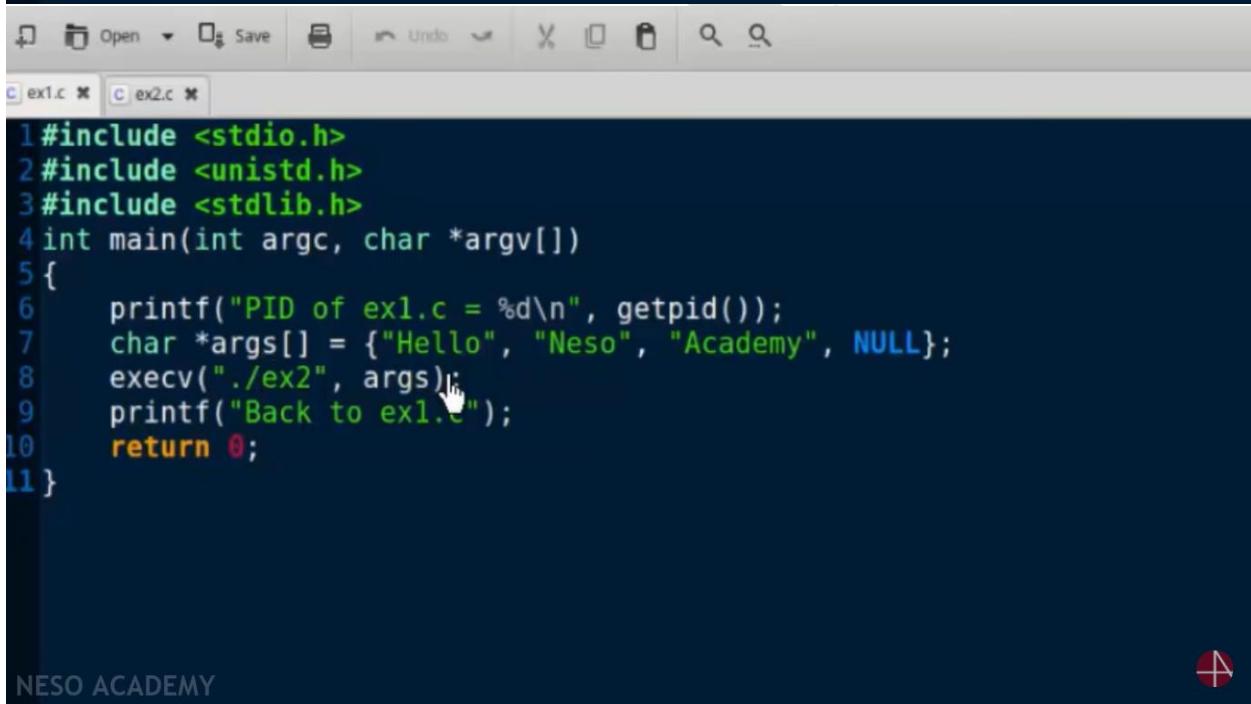
The screenshot shows a Linux desktop environment with a file manager window open. The file manager window has a sidebar with 'My Computer' and 'Devices' sections, and a main area showing two files: 'ex1.c' and 'ex2.c'. A cursor is pointing at 'ex1.c'. Below the file manager is a terminal window titled 'ex1.c' containing the following C code:

```
1#include <stdio.h>
2#include <unistd.h>
3#include <stdlib.h>
4int main(int argc, char *argv[])
5{
6    printf("PID of ex1.c = %d\n", getpid());
7    char *args[] = {"Hello", "Neso", "Academy", NULL};
8    execv("./ex2", args);
9    printf("Back to ex1.c");
10   return 0;
11}
```

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



```
1#include <stdio.h>
2#include <unistd.h>
3#include <stdlib.h>
4int main(int argc, char *argv[])
5{
6    printf("We are in ex2.c\n");
7    printf("PID of ex2.c = %d\n", getpid());
8    return 0;
9}
```



```
1#include <stdio.h>
2#include <unistd.h>
3#include <stdlib.h>
4int main(int argc, char *argv[])
5{
6    printf("PID of ex1.c = %d\n", getpid());
7    char *args[] = {"Hello", "Neso", "Academy", NULL};
8    execv("./ex2", args);
9    printf("Back to ex1.c");
10   return 0;
11}
```

NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

```
jaision@neso-academy ~/Desktop/Exec
```

```
| |
```

```
jaision@neso-academy ~/Desktop/Exec $ gcc ex1.c -o ex1
jaision@neso-academy ~/Desktop/Exec $ gcc ex2.c -o ex2
jaision@neso-academy ~/Desktop/Exec $ ./ex1
PID of ex1.c = 5962
We are in ex2.c
PID of ex2.c = 5962
jaision@neso-academy ~/Desktop/Exec $
```

```
jaision@neso-academy ~/Desktop/Exec
```

```
| |
```

```
jaision@neso-academy ~/Desktop/Exec $ gcc ex1.c -o ex1
jaision@neso-academy ~/Desktop/Exec $ gcc ex2.c -o ex2
jaision@neso-academy ~/Desktop/Exec $ ./ex1
PID of ex1.c = 5962
We are in ex2.c
PID of ex2.c = 5962
jaision@neso-academy ~/Desktop/Exec $
```

## Threading Issues (Part-1)

### The fork() and exec() System Calls

The semantics of the fork() and exec() system calls change in a multithreaded program.

#### Issue

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?

#### Solution

Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.



### But which version of fork() to use and when?

Also, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process —including all threads.



### But which version of fork() to use and when?

Also, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process —including all threads.

Which of the two versions of fork() to use depends on the application.

#### If exec() is called immediately after forking

Then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process.

In this instance, duplicating only the calling thread is appropriate.

NESO ACADEMY

#### If the separate process

#### does not call exec() after forking

Then the separate process should duplicate all threads.



## Threading Issues (Part-2)

### Thread Cancellation

Thread cancellation is the task of terminating a thread before it has completed.



If multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.



When a user presses a button on a web browser that stops a web page from loading any further, all threads loading the page are canceled.

A thread that is to be canceled  is often referred to as the **target thread**. 



Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation:** One thread immediately terminates the target thread.
2. **Deferred cancellation:** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Where the difficulty with cancellation lies:



### Where the difficulty with cancellation lies:

In situations where:

- Resources have been allocated to a canceled thread
- A thread is canceled while in the midst of updating data it is sharing with other threads.

Often, the OS will reclaim system resources from a canceled thread  
but will not reclaim all resources.

Therefore, canceling a thread asynchronously  
may not free a necessary system-wide resource.



Often, the OS will reclaim system resources from a canceled thread  
but will not reclaim all resources.

Therefore, canceling a thread asynchronously  
may not free a necessary system-wide resource.

### With **deferred** cancellation:

One thread indicates that a target thread is to be canceled.

But cancellation occurs only after the target thread has checked a flag to determine if it  
should be canceled or not.

This allows a thread to check whether it should be canceled at a point when it can be  
canceled safely.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems.

By switching the CPU among processes, the operating system can make the computer more productive.

#### Topics to be covered:

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems.
- To describe various CPU-scheduling algorithms

*Let's get the basics right* 



## *Let's get the basics right*

- In a single-processor system, only one process can run at a time.
- Any others must wait until the CPU is free and can be rescheduled.
  - The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
  - A process is executed until it must wait, typically for the completion of some I/O request.



In a simple computer system, the CPU then just sits idle.

All this waiting time is wasted; no useful work is accomplished.



NESO ACADEMY



In a simple computer system, the CPU then just sits idle.

All this waiting time is wasted; no useful work is accomplished.



- With multiprogramming, we try to use this time productively.
  - Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process and this pattern continues.

NESO ACADEMY



### CPU and I/O Burst Cycles

Process execution consists of a cycle of CPU execution and I/O wait.

Processes alternate between these two states.

Process execution begins with a

**CPU burst** That is followed by an

**I/O burst**, which is followed by another

**CPU burst**, then another

**I/O burst**,

and so on...

CPU burst is  
when the  
process is being  
executed in the  
CPU

I/O burst is  
when the CPU is  
waiting for I/O  
for further  
execution

Eventually, the final **CPU burst** ends with a system request to terminate execution. 



## Preemptive and Non-Preemptive Scheduling

### CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

### Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.



CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the **running state** to the **waiting state**
2. When a process switches from the **running state** to the **ready state** (for example, when an interrupt occurs).
3. When a process switches from the **waiting state** to the **ready state** (for example, at completion of I/O).
4. When a process terminates.

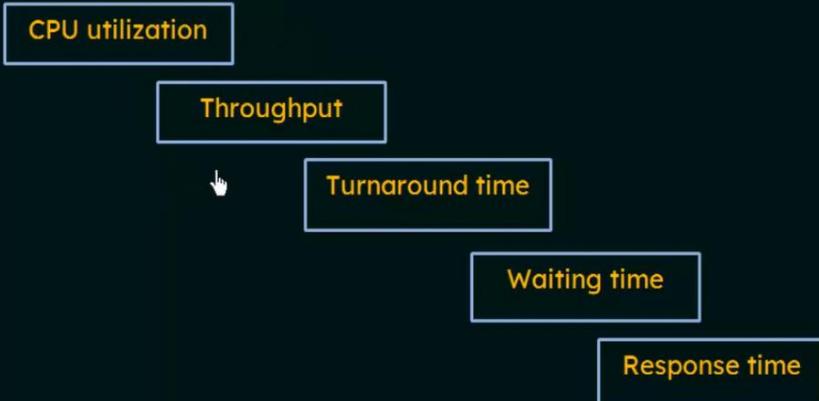
For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.

However, there is a choice for situations 2 and 3.

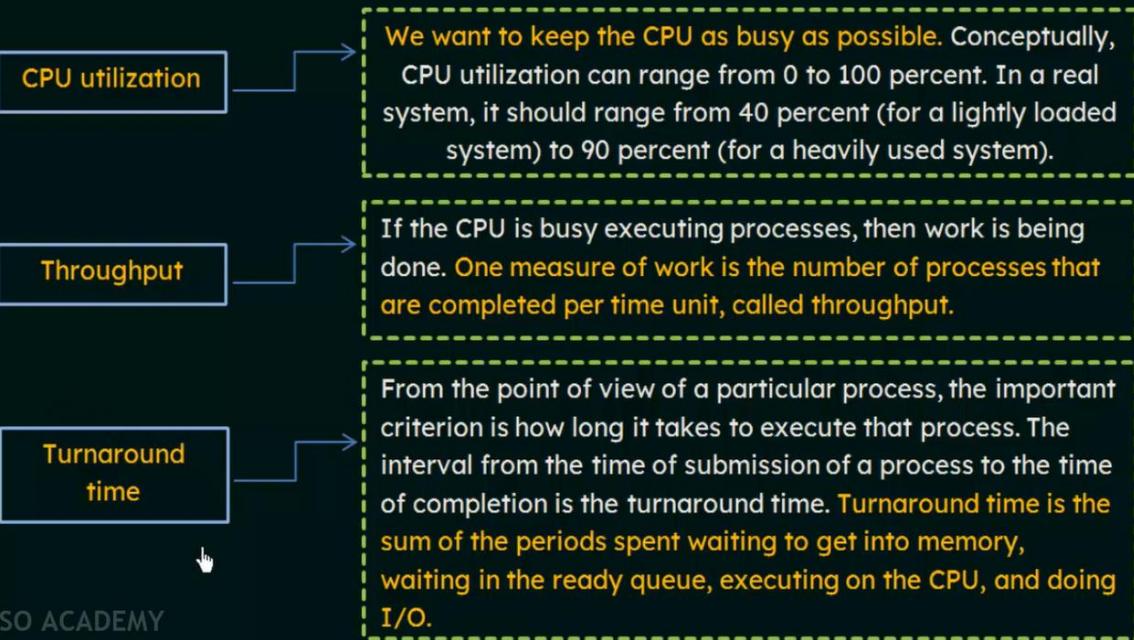
When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is preemptive.

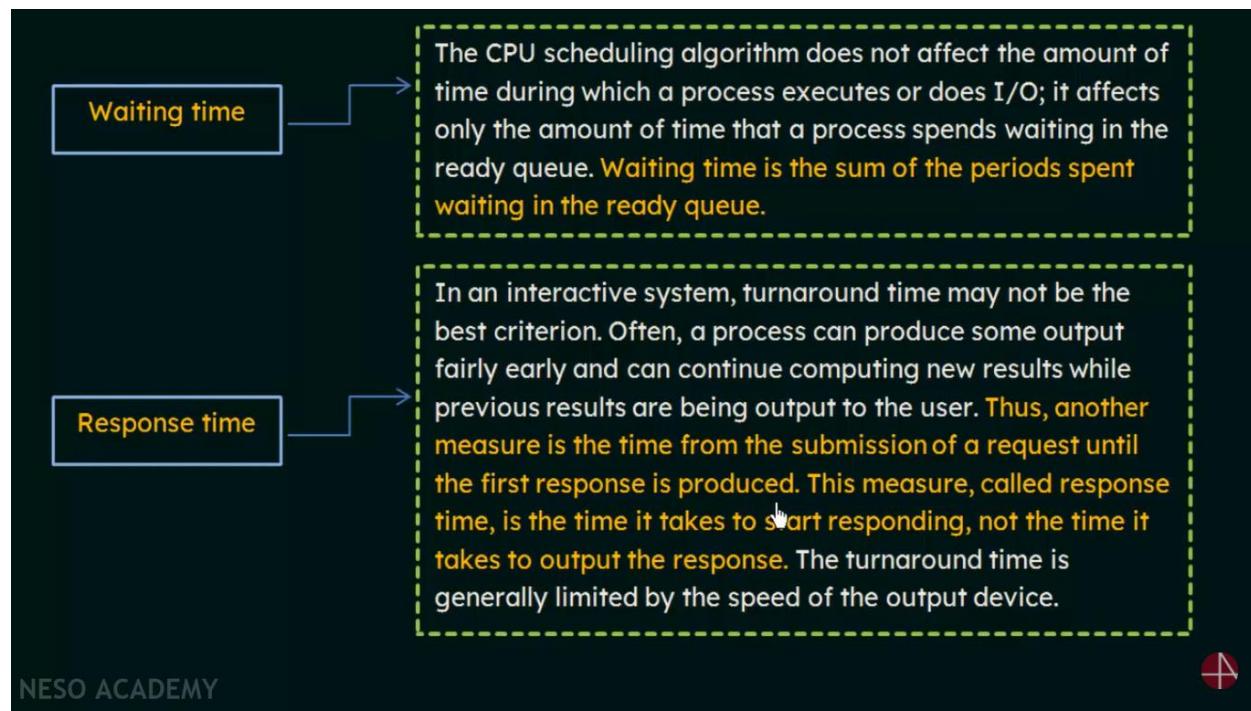


## Scheduling Criteria



## Scheduling Criteria





## Scheduling Algorithms (First-Come, First-Served Scheduling)

- By far the simplest CPU-scheduling algorithm.
- The process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a **FIFO** queue.



- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.



Consider the following set of processes that arrive at time 0

Process	Burst Time (ms)
P1	24
P2	3
P3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart:



Waiting Time for P1 = 0 ms

Waiting Time for P2 = 24 ms

Waiting Time for P3 = 27 ms

$$\text{Average Waiting Time} = (0 + 24 + 27)/3 = 17 \text{ ms}$$



If the processes arrive in the order P2, P3, P1, however the result will be shown in the following Gantt chart:



Waiting Time for P1 = 6 ms

Waiting Time for P2 = 0 ms

Waiting Time for P3 = 3 ms

$$\text{Average Waiting Time} = (6 + 0 + 3)/3 = 3 \text{ ms}$$



This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.



This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

The FCFS scheduling algorithm is nonpreemptive

- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.
- It would be disastrous to allow one process to keep the CPU for an extended period.



### First-Come, First-Served Scheduling

#### Solved Problem -1

**Convoy Effect**

If processes with higher burst time arrived before the processes with smaller burst time, then, smaller processes have to wait for a long time for longer processes to release the CPU.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Consider the set of 5 processes whose arrival time and burst time are given below:

Process ID	Arrival Time	Burst Time
P1	4	5
P2	6	4
P3	0	3
P4	6	2
P5	5	4

Calculate the **average waiting time** and **average turnaround time**,  
if FCFS Scheduling Algorithm is followed.



NESO ACADEMY



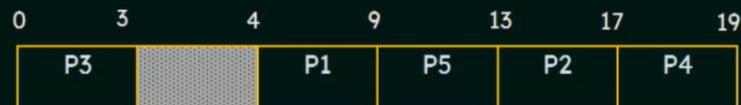
Consider the set of 5 processes whose arrival time and burst time are given below:

Process ID	Arrival Time	Burst Time
P1	4	5
P2	6	4
P3	0	3
P4	6	2
P5	5	4

Calculate the **average waiting time** and **average turnaround time**,  
if FCFS Scheduling Algorithm is followed.

Solution:

**Gantt Chart:**

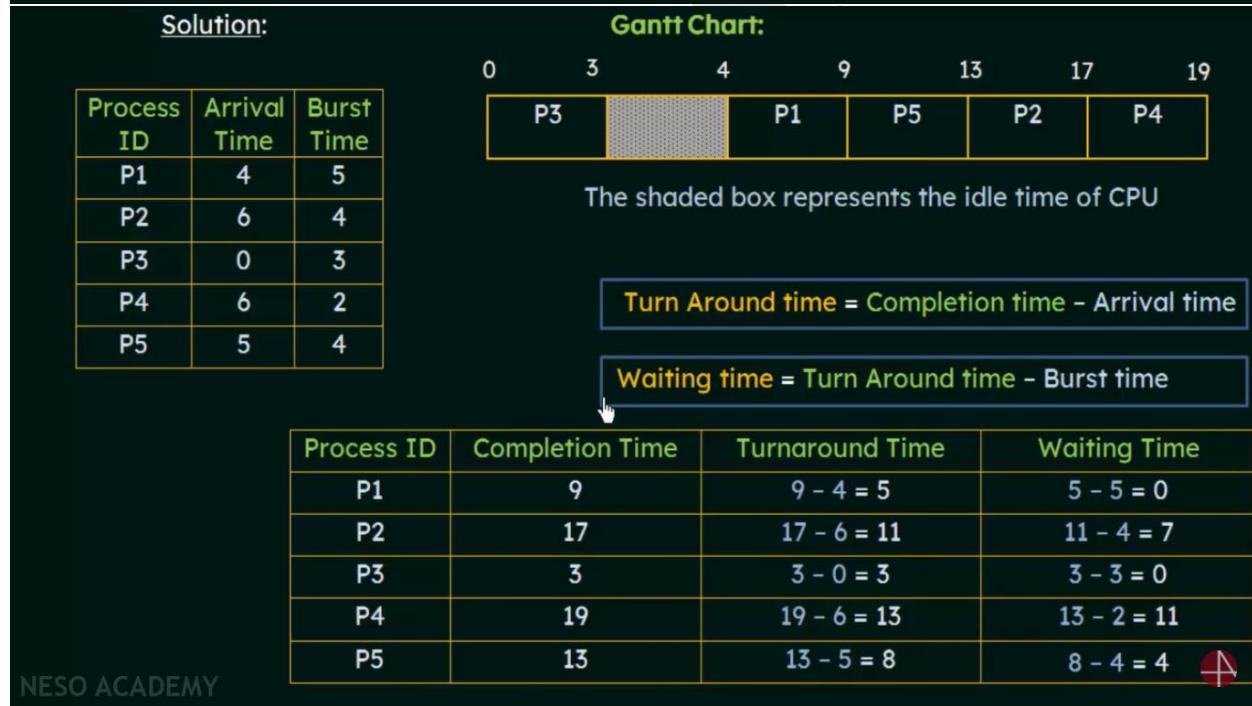
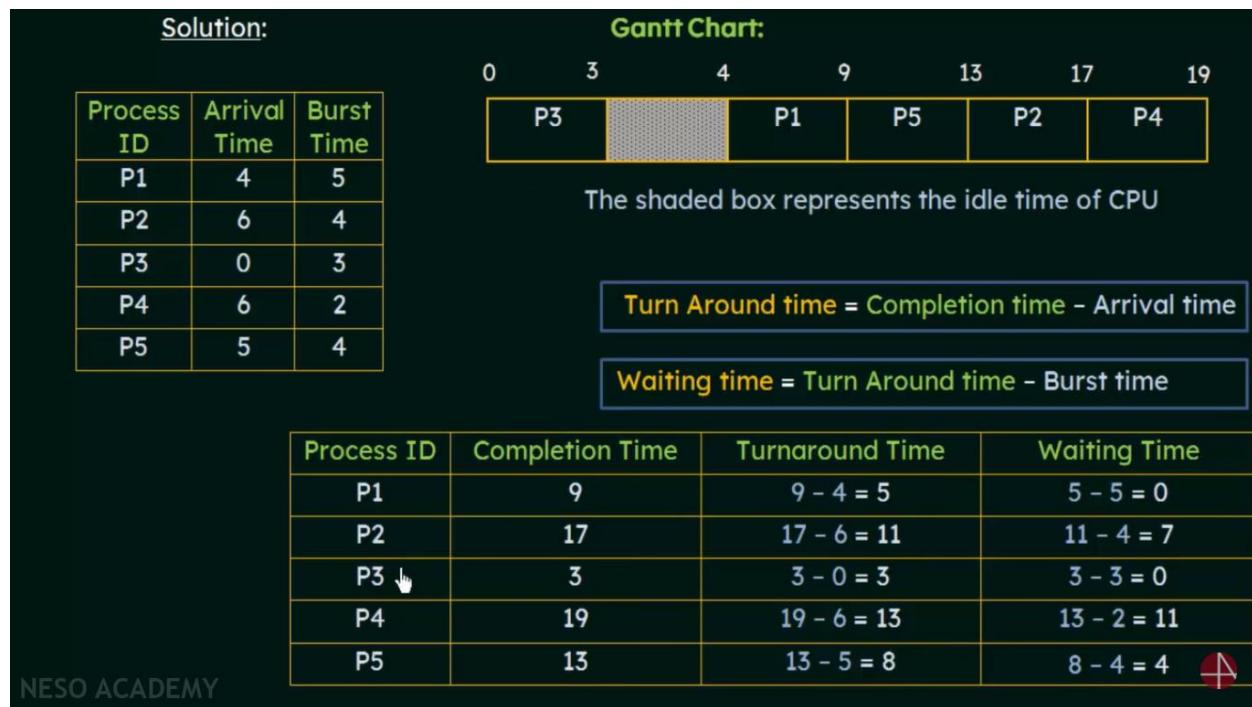


The shaded box represents the idle time of CPU



NESO ACADEMY

NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.



NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	9	9 - 4 = 5	5 - 5 = 0
P2	17	17 - 4 = 11	11 - 4 = 7
P3	3	3 - 0 = 3	3 - 3 = 0
P4	19	19 - 6 = 13	13 - 2 = 11
P5	13	13 - 5 = 8	8 - 4 = 4

Now,

$$\begin{aligned}\text{Average Turn Around time} &= (5 + 11 + 3 + 13 + 8) / 5 \\ &= 40 / 5 \\ &= 8 \text{ units}\end{aligned}$$

$$\begin{aligned}\text{Average waiting time} &= (0 + 7 + 0 + 11 + 4) / 5 \\ &\quad \downarrow \\ &= 22 / 5 \\ &= 4.4 \text{ units}\end{aligned}$$



## First-Come, First-Served Scheduling

### Solved Problem - 2

The arrival times and burst times for a set of 6 processes are given in the table below:

Process ID	Arrival Time	Burst Time
P1	0	3
P2	1	2
P3	2	1
P4	3	4
P5	4	5
P6	5	2

If FCFS Scheduling Algorithm is followed and there is 1 unit of overhead in scheduling the processes, find the efficiency of the algorithm.



NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Process ID	Arrival Time	Burst Time
P1	0	3
P2	1	2
P3	2	1
P4	3	4
P5	4	5
P6	5	2

If FCFS Scheduling Algorithm is followed and there is 1 unit of overhead in scheduling the processes, find the efficiency of the algorithm.

Solution:

**Gantt Chart:**



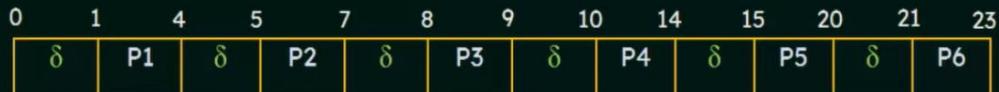
Here,  $\delta$  denotes the unit overhead in scheduling the processes

NESO ACADEMY



Solution:

**Gantt Chart:**



Here,  $\delta$  denotes the unit overhead in scheduling the processes

Now,

$$\text{Useless time or Wasted time} = 6 \times \delta = 6 \times 1 \\ = 6 \text{ units}$$

$$\text{Total time} = 23 \text{ units}$$

$$\text{Useful time} = 23 \text{ units} - 6 \text{ units} \\ = 17 \text{ units}$$

$$\text{Efficiency (η)} = \frac{\text{Useful time}}{\text{Total Time}} \\ = \frac{17 \text{ units}}{23 \text{ units}}$$

$$= 0.7391$$

$$= 73.91\%$$

NESO ACADEMY



## Scheduling Algorithms (Shortest-Job-First Scheduling)

- This algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

The SJF algorithm can be either preemptive or nonpreemptive

A more appropriate term for this scheduling method would be the

### **Shortest-Next-CPU-Burst Algorithm**

because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

NESO ACADEMY



### Example of SJF Scheduling (Non-Premptive)

Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process ID	Burst Time
P1	6
P2	8
P3	7
P4	3

Waiting Time for P1 = 3 ms

Waiting Time for P2 = 16 ms

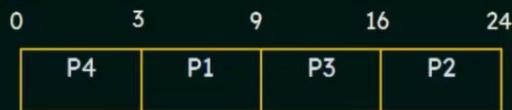
Waiting Time for P3 = 9 ms

Waiting Time for P4 = 0 ms

Average Waiting Time

$$= (3 + 16 + 9 + 0)/4 = 7 \text{ ms}$$

#### Gantt Chart:



By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

NESO ACADEMY



### Example of SJF Scheduling (Preemptive)

Consider the following four processes, with the length of the CPU burst given in milliseconds and the processes arrive at the ready queue at the times shown:

Process ID	Arrival Time	Burst Time
P1	0	7
P2	1	4
P3	2	9
P4	3	5

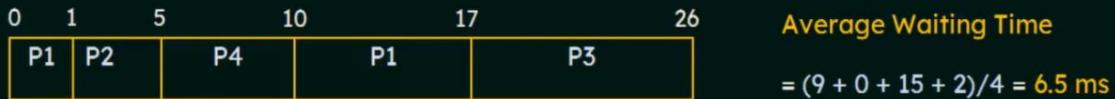
$$\text{Waiting Time for P1} = (10 - 1 - 0) = 9 \text{ ms}$$

$$\text{Waiting Time for P2} = (1 - 0 - 1) = 0 \text{ ms}$$

$$\text{Waiting Time for P3} = (17 - 0 - 2) = 15 \text{ ms}$$

$$\text{Waiting Time for P4} = (5 - 0 - 3) = 2 \text{ ms}$$

**Gantt Chart:**



Average Waiting Time

$$= (9 + 0 + 15 + 2)/4 = 6.5 \text{ ms}$$

**Waiting Time** = Total waiting Time – No. of milliseconds Process executed – Arrival Time

Preemptive SJF scheduling is sometimes called Shortest-Remaining-Time-First Scheduling.

### **Problems with SJF Scheduling:**

- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling.
- There is no way to know the length of the next CPU burst.

One approach is:

- To try to approximate SJF scheduling.
- We may not know the length of the next CPU burst, but we may be able to predict its value.
- We expect that the next CPU burst will be similar in length to the previous ones.
- Thus, by computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

## Shortest-Job-First Scheduling

## Solved Problem -1

GATE 2014

An operating system uses shortest remaining time first scheduling algorithm for pre-emptive scheduling of processes. Consider the following set of processes with their arrival times and CPU burst times (in milliseconds):

Process ID	Arrival Time	Burst Time
P1	0	12
P2	2	4
P3	3	6
P4	8	5

The average waiting time (in milliseconds) of the processes is \_\_\_\_\_.

- (A) 4.5      (B) 5.0      (C) 5.5      (D) 6.5



Process ID	Arrival Time	Burst Time
P1	0	12
P2	2	4
P3	3	6
P4	8	5

$$\text{Waiting Time for P1} = (17 - 2 - 0) = 15 \text{ ms}$$

$$\text{Waiting Time for P2} = (2 - 0 - 2) = 0 \text{ ms}$$

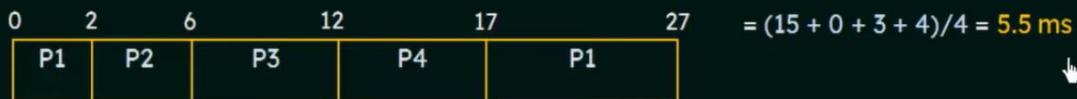
$$\text{Waiting Time for P3} = (6 - 0 - 3) = 3 \text{ ms}$$

$$\text{Waiting Time for P4} = (12 - 0 - 8) = 4 \text{ ms}$$

Solution:

## Gantt Chart:

## Average Waiting Time



**Waiting Time** = Total waiting Time - No. of milliseconds Process executed - Arrival Time



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

**Shortest-Job-First Scheduling**  
Solved Problem - 2

GATE 2016

Consider the following processes, with the arrival time and the length of the CPU burst given in milliseconds. The scheduling algorithm used is preemptive shortest remaining-time first.

Process ID	Arrival Time	Burst Time
P1	0	10
P2	3	6
P3	7	1
P4	8	3

The average turn around time of these processes is \_\_\_\_\_ milliseconds.

- (A) 8.25      (B) 10.25      (C) 6.35      (D) 4.25



Process ID	Arrival Time	Burst Time
P1	0	10 7
P2	3	6 2
P3	7	1
P4	8	3

Solution:

**Gantt Chart:**



$$\text{Turnaround Time for P1} = (20 - 0) = 20 \text{ ms}$$

$$\text{Turnaround Time for P2} = (10 - 3) = 7 \text{ ms}$$

$$\text{Turnaround Time for P3} = (8 - 7) = 1 \text{ ms}$$

$$\text{Turnaround Time for P4} = (13 - 8) = 5 \text{ ms}$$

Average Turnaround Time

$$= (20 + 7 + 1 + 5)/4$$

$$= 33/4$$

$$= 8.25 \text{ ms}$$

Turn Around time = Completion time – Arrival time



## Scheduling Algorithms (Priority Scheduling)

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst.  
The larger the CPU burst, the lower the priority, and vice versa.

Priority scheduling can be either preemptive or nonpreemptive.

A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.



Consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, P3, P4, P5, with the length of the CPU burst given in milliseconds:

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Waiting Time for P1 = 6 ms  
 Waiting Time for P2 = 0 ms  
 Waiting Time for P3 = 16 ms  
 Waiting Time for P4 = 18 ms  
 Waiting Time for P5 = 1 ms

Using **Priority Scheduling**, we would schedule these processes according to the following **Gantt Chart**:



Average Waiting Time

$$\begin{aligned} &= (6 + 0 + 16 + 18 + 1) / 5 \\ &= 41 / 5 \text{ ms} \\ &= 8.2 \text{ ms} \end{aligned}$$



### Problem with Priority Scheduling

- A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**.
- A process that is ready to run but waiting for the CPU can be considered blocked.
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

### Solution to the Problem

- A solution to the problem of indefinite blockage of low-priority processes is **aging**.
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.
- For example,  
 If priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.



NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

## Priority Scheduling

### Solved Problem -1

GATE 2017

Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds), and priority (0 is the highest priority) shown below. None of the processes have I/O burst time.

Process ID	Arrival Time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

The average waiting time (in milliseconds) of all the processes using preemptive priority scheduling algorithm is \_\_\_\_\_.

- (A) 29      (B) 30      (C) 31      (D) 32

NESO ACADEMY



Process ID	Arrival Time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

$$\text{Waiting Time for P1} = (40 - 2 - 0) = 38 \text{ ms}$$

$$\text{Waiting Time for P2} = (5 - 0 - 5) = 0 \text{ ms}$$

$$\text{Waiting Time for P3} = (49 - 0 - 12) = 37 \text{ ms}$$

$$\text{Waiting Time for P4} = (33 - 3 - 2) = 28 \text{ ms}$$

$$\text{Waiting Time for P5} = (51 - 0 - 9) = 42 \text{ ms}$$

Average Waiting Time

$$= (38 + 0 + 37 + 28 + 42) / 5$$

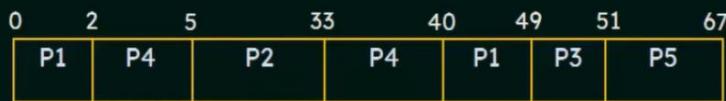
$$= 145/5 \text{ ms}$$

$$= 29 \text{ ms}$$



Solution:

Gantt Chart:



Waiting Time = Total waiting Time – No. of milliseconds Process executed – Arrival Time

NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

## Priority Scheduling

### Solved Problem - 2

Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds), and priority shown below: (Higher number represents higher priority)

Process ID	Arrival Time	Burst Time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turn around time.

NESO ACADEMY



Process ID	Arrival Time	Burst Time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	4	4 - 0 = 4	4 - 4 = 0
P2	15	15 - 1 = 14	14 - 3 = 11
P3	12	12 - 2 = 10	10 - 1 = 9
P4	9	9 - 3 = 6	6 - 5 = 1
P5	11	11 - 4 = 7	7 - 2 = 5

Solution:

Gantt Chart:



Average Turn Around time

$$= (4 + 14 + 10 + 6 + 7) / 5 \\ = 41 / 5 = 8.2 \text{ ms}$$

Turn Around time = Completion time - Arrival time

Waiting time = Turn Around time - Burst time

Average waiting time

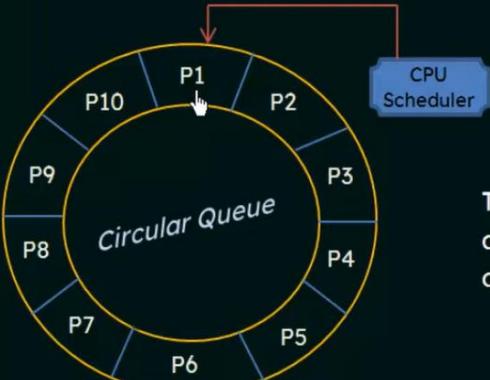
$$= (0 + 11 + 9 + 1 + 5) / 5 \\ = 26 / 5 = 5.2 \text{ ms}$$

NESO ACADEMY



## Scheduling Algorithms (Round-Robin Scheduling)

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but **preemption** is added to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined (generally from 10 to 100 milliseconds)



- The ready queue is treated as a **circular queue**.

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

**NESO ACADEMY**



### Implementation of Round Robin scheduling:

- We keep the ready queue as a **FIFO** queue of processes.
- New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.



**One of two things will then happen**

The process may have a CPU burst of less than 1 time quantum

The process itself will release the CPU voluntarily

The CPU scheduler will then proceed to the next process in the ready queue

The CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the O.S.

A context switch will be executed, and the process will be put at the tail of the ready queue

The CPU scheduler will then select the next process in the ready queue

**NESO ACADEMY**



NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

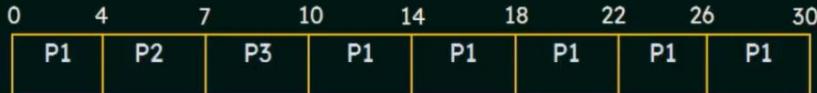
## Round-Robin Scheduling (Turnaround Time and Waiting Time)

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds and time quantum taken as 4 milliseconds for RR Scheduling:

Process ID	Burst Time
P1	24
P2	3
P3	3



Gantt Chart:

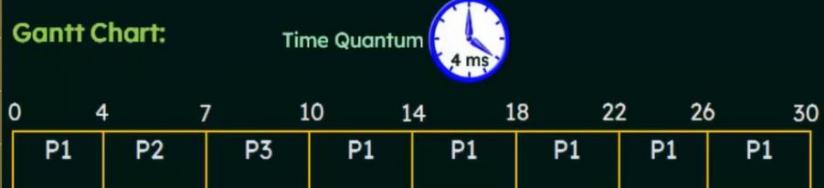


NESO ACADEMY



Process ID	Burst Time
P1	24
P2	3
P3	3

Gantt Chart:



### Method 1

$$\text{Turn Around time} = \text{Completion time} - \text{Arrival time}$$

$$\text{Waiting time} = \text{Turn Around time} - \text{Burst time}$$

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	30	$30 - 0 = 30$	$30 - 24 = 6$
P2	7	$7 - 0 = 7$	$7 - 3 = 4$
P3	10	$10 - 0 = 10$	$10 - 3 = 7$

NESO ACADEMY

### Method 2

$$\text{Waiting time} = \text{Last Start Time} - \text{Arrival Time} - (\text{Preemption} \times \text{Time Quantum})$$

Process ID	Waiting Time
P1	$26 - 0 - (5 \times 4) = 6$
P2	$4 - 0 - (0 \times 4) = 4$
P3	$7 - 0 - (0 \times 4) = 7$



NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### Method 1

$$\text{Turn Around time} = \text{Completion time} - \text{Arrival time}$$

$$\text{Waiting time} = \text{Turn Around time} - \text{Burst time}$$

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	30	$30 - 0 = 30$	$30 - 24 = 6$
P2	7	$7 - 0 = 7$	$7 - 3 = 4$
P3	10	$10 - 0 = 10$	$10 - 3 = 7$

#### Average Turn Around time

$$= (30 + 7 + 10) / 3$$

$$= 47 / 3 = 15.66 \text{ ms}$$

#### Average waiting time

$$= (6 + 4 + 7) / 3$$

$$= 17 / 3 = 5.66 \text{ ms}$$

NESO ACADEMY

### Method 2

$$\text{Waiting time} = \text{Last Start Time} - \text{Arrival Time} - (\text{Preemption} \times \text{Time Quantum})$$

Process ID	Waiting Time
P1	$26 - 0 - (5 \times 4) = 6$
P2	$4 - 0 - (0 \times 4) = 4$
P3	$7 - 0 - (0 \times 4) = 7$

#### Average waiting time

$$= (6 + 4 + 7) / 3$$

$$= 17 / 3 = 5.66 \text{ ms}$$



## Round Robin Scheduling

### Solved Problem

(Part - 1)



Consider the set of 5 processes whose arrival time and burst time are given below:

Process ID	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

If the CPU scheduling policy is Round Robin with time quantum = 2 units, calculate the average waiting time and average turn around time.

NESO ACADEMY



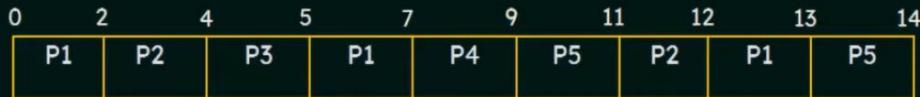
NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Process ID	Arrival Time	Burst Time
P1	0	5-3-2
P2	1	3-1
P3	2	1
P4	3	2
P5	4	3-1

Time Quantum  
2 Units



### Gantt Chart:



### Ready Queue

NESO ACADEMY



## Round Robin Scheduling

Solved Problem

(Part - 2)

Consider the set of 5 processes whose arrival time and burst time are given below:

Process ID	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

If the CPU scheduling policy is Round Robin with time quantum = 2 units, calculate the average waiting time and average turn around time.

NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
 SUBJECT- OPERATING SYSTEM(OS)  
 TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

### Gantt Chart:



Turn Around time = Completion time – Arrival time

Waiting time = Turn Around time – Burst time

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	

Process ID	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

NESO ACADEMY



Turn Around time = Completion time – Arrival time

Waiting time = Turn Around time – Burst time

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

Process ID	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

### Average Turn Around time

$$= (13 + 11 + 3 + 6 + 10) / 5$$

$$= 43 / 5 = \mathbf{8.6 \text{ units}}$$

### Average waiting time

$$= (8 + 8 + 2 + 4 + 7) / 5$$

$$= 29 / 5 = \mathbf{5.8 \text{ units}}$$

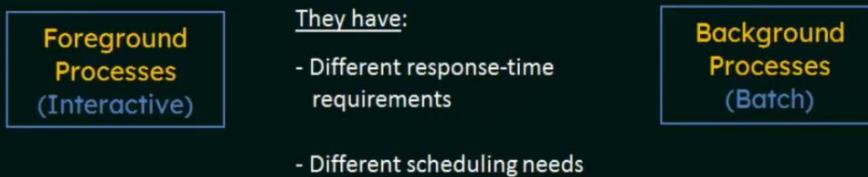
NESO ACADEMY



## Scheduling Algorithms (Multilevel Queue Scheduling)

A class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.

Example:



In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.

- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm.

Example:

Separate queues might be used for foreground and background processes.

The **foreground queue** might be scheduled by an **RR algorithm**, while the **background queue** is scheduled by an **FCFS algorithm**.

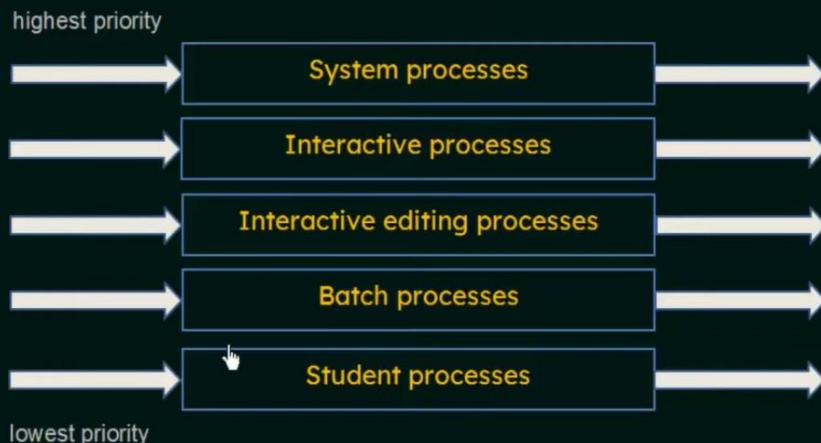


In addition, there must be scheduling among the queues, which is commonly implemented as **fixed-priority preemptive scheduling**.

For example, the foreground queue may have absolute priority over the background queue.



An example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:



## Scheduling Algorithms (Multilevel Feedback-Queue Scheduling)

The multilevel feedback-queue scheduling algorithm allows a process to move between queues.

- The idea is to separate processes according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.

This form of aging prevents starvation.





Figure: Multilevel feedback queues



In general, a multilevel feedback-queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue
- The method used to determine which queue a process will enter when that process needs service



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

## Scheduling Algorithms (Solved Problems)

Question:

1

GATE 2001

Consider a set of  $n$  tasks with known runtimes  $r_1, r_2, \dots, r_n$  to be run on a uniprocessor machine. Which of the following processor scheduling algorithms will result in the maximum throughput?

- (a) Round-Robin
- (b) Shortest-Job-First
- (c) Highest-Response-Ratio-Next
- (d) First-Come-First-Served

NESO ACADEMY



Question:

2

GATE 2002

Which of the following scheduling algorithms is non-preemptive?

- (a) Round-Robin
- (b) First In First Out
- (c) Multilevel Queue Scheduling
- (d) Multilevel Queue Scheduling with Feedback

NESO ACADEMY



Question: 3

GATE 2010

Which of the following statements are true?

- I. Shortest remaining time first scheduling may cause starvation
  - II. Preemptive scheduling may cause starvation
  - III. Round Robin is better than FCFS in terms of response time
- (a) I only
- (b) I and III only
- (c) II and III only
- (d) I, II and III

NESO ACADEMY



Question: 4

GATE 2012

Consider the 3 processes, P1, P2 and P3 shown in the table.

Process ID	Arrival Time	Time Units Required
P1	0	5
P2	1	7
P3	3	4



The completion order of the 3 processes under the policies FCFS and RR2 (round robin scheduling with CPU quantum of 2 time units) are:

- (a) FCFS: P1, P2, P3  
RR2: P1, P2, P3
- (b) FCFS: P1, P3, P2  
RR2: P1, P3, P2
- (c) FCFS: P1, P2, P3  
RR2: P1, P3, P2
- (d) FCFS: P1, P3, P2  
RR2: P1, P2, P3

NESO ACADEMY



Question:

4

GATE 2012

Consider the 3 processes, P1, P2 and P3 shown in the table.

Process ID	Arrival Time	Time Units Required
P1	0	5-3-1
P2	1	7-5-3
P3	3	4-2

P3 P1

The completion order of the 3 processes under the policies FCFS and RR2 (round robin scheduling with CPU quantum of 2 time units) are:

(a) FCFS: P1, P2, P3

(b) FCFS: P1, P3, P2

RR2: P1, P2, P3

RR2: P1, P3, P2

(c) FCFS: P1, P2, P3

(d) FCFS: P1, P3, P2

RR2: P1, P3, P2

RR2: P1, P2, P3

NESO ACADEMY



Question:

5

GATE 2013

A scheduling algorithm assigns priority proportional to the waiting time of a process.

Every process starts with priority zero (the lowest priority). The scheduler re-evaluates the process priorities every T time units and decides the next process to schedule. Which one of the following is TRUE if the processes have no I/O operations and all arrive at time zero?



(a) This algorithm is equivalent to the first-come-first-serve algorithm.

(b) This algorithm is equivalent to the round-robin algorithm.

(c) This algorithm is equivalent to the shortest-job-first algorithm.

(d) This algorithm is equivalent to the shortest-remaining-time-first algorithm.

NESO ACADEMY



## Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system.

Cooperating processes  
can either

directly share a logical  
address space  
(that is, both code and data)

or be allowed to share  
data only through files  
or messages.

Concurrent access to shared data may result in data inconsistency!

In this chapter,  
we discuss various mechanisms to ensure –  
The orderly execution of cooperating processes that share a logical address space,

So that data consistency is maintained.



## Producer Consumer Problem

A producer process produces information that is consumed by a consumer process.

For example, a compiler may produce assembly code, which is consumed by an assembler.  
The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.



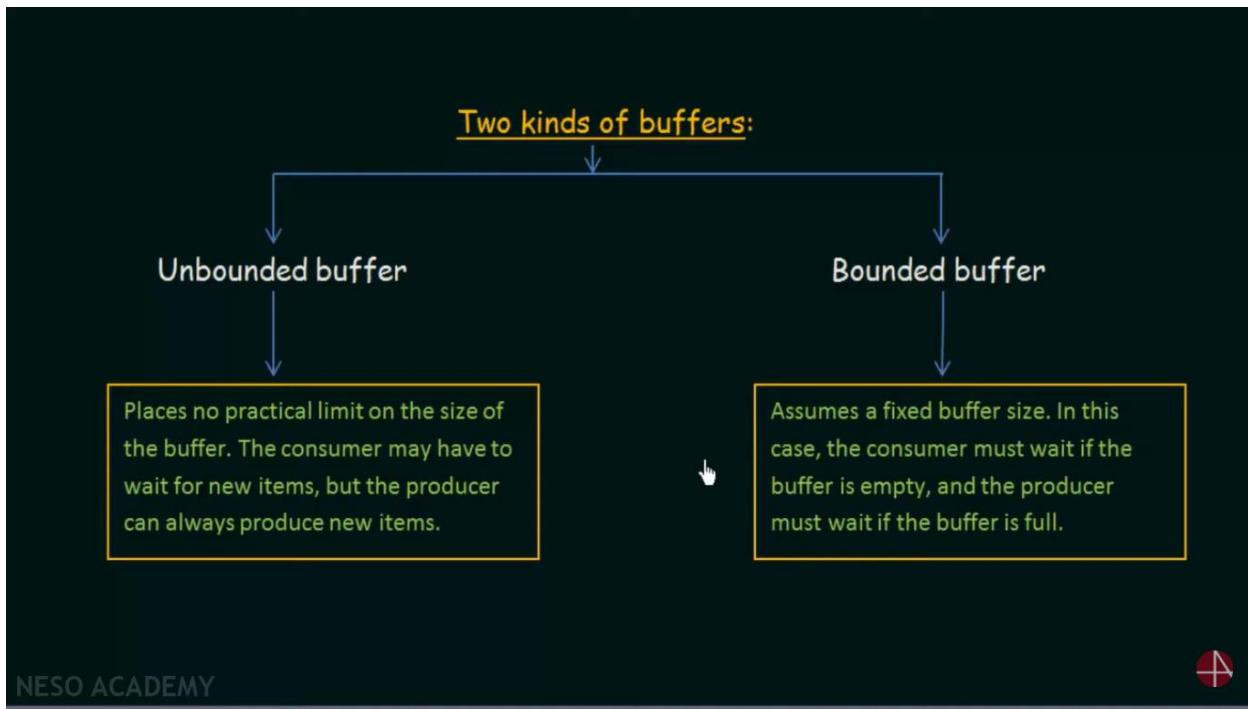
NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.  

- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.





**counter** variable = 0

counter is incremented every time we add a new item to the buffer

**counter++**

counter is decremented every time we remove one item from the buffer

**counter--**

-----Example-----

- Suppose that the value of the variable **counter** is currently 5.
- The producer and consumer processes execute the statements "**counter++**" and "**counter--**" concurrently.
- Following the execution of these two statements, the **value** of the variable counter may be **4, 5, or 6!**
- The **only correct result**, though, is **counter == 5**, which is generated correctly if the producer and consumer execute separately.



-----Example-----

- Suppose that the value of the variable counter is currently 5.
- The producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

"counter++" may be implemented in machine language (on a typical machine) as:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

"counter--" may be implemented in machine language (on a typical machine) as:

```
register2 = counter
register2 = register2 - 1
counter = register2
```



"counter++" may be implemented in machine language (on a typical machine) as:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

"counter--" may be implemented in machine language (on a typical machine) as:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

T <sub>0</sub> :	producer	execute	register <sub>1</sub> = counter	{ register <sub>1</sub> = 5 }
T <sub>1</sub> :	producer	execute	register <sub>1</sub> = register <sub>1</sub> + 1	{ register <sub>1</sub> = 6 }
T <sub>2</sub> :	consumer	execute	register <sub>2</sub> = counter	{ register <sub>2</sub> = 5 }
T <sub>3</sub> :	consumer	execute	register <sub>2</sub> = register <sub>2</sub> - 1	{ register <sub>2</sub> = 4 }
T <sub>4</sub> :	producer	execute	counter = register <sub>1</sub>	{ counter = 6 }
T <sub>5</sub> :	consumer	execute	counter = register <sub>2</sub>	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

T <sub>0</sub> :	producer	execute	register <sub>1</sub>	=	counter	{ register <sub>1</sub> = 5 }
T <sub>1</sub> :	producer	execute	register <sub>1</sub>	=	register <sub>1</sub> + 1	{ register <sub>1</sub> = 6 }
T <sub>2</sub> :	consumer	execute	register <sub>2</sub>	=	counter	{ register <sub>2</sub> = 5 }
T <sub>3</sub> :	consumer	execute	register <sub>2</sub>	=	register <sub>2</sub> - 1	{ register <sub>2</sub> = 4 }
T <sub>4</sub> :	producer	execute	counter	=	register <sub>1</sub>	{ counter = 6 }
T <sub>5</sub> :	consumer	execute	counter	=	register <sub>2</sub>	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.

Clearly, we want the resulting changes not to interfere with one another. Hence we need process synchronization.



NESO ACADEMY



### The Critical-Section Problem

Consider a system consisting of n processes { P<sub>0</sub>, P<sub>1</sub>, ..., P<sub>n</sub> }.

Each process has a segment of code, called a

critical section

in which the process may be changing common variables, updating a table, writing a file, and so on.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

That is, no two processes are executing in their critical sections at the same time.

The critical-section problem is to design a protocol that the processes can use to cooperate.

NESO ACADEMY



- Each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section.
- The critical section may be followed by an exit section.
- The remaining code is the remainder section.



Figure: General structure of a typical process.

NESO ACADEMY



A solution to the critical-section problem must satisfy the following three requirements:

**1. Mutual exclusion:**

If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.

**2. Progress:**

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

**3. Bounded waiting:**

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

NESO ACADEMY

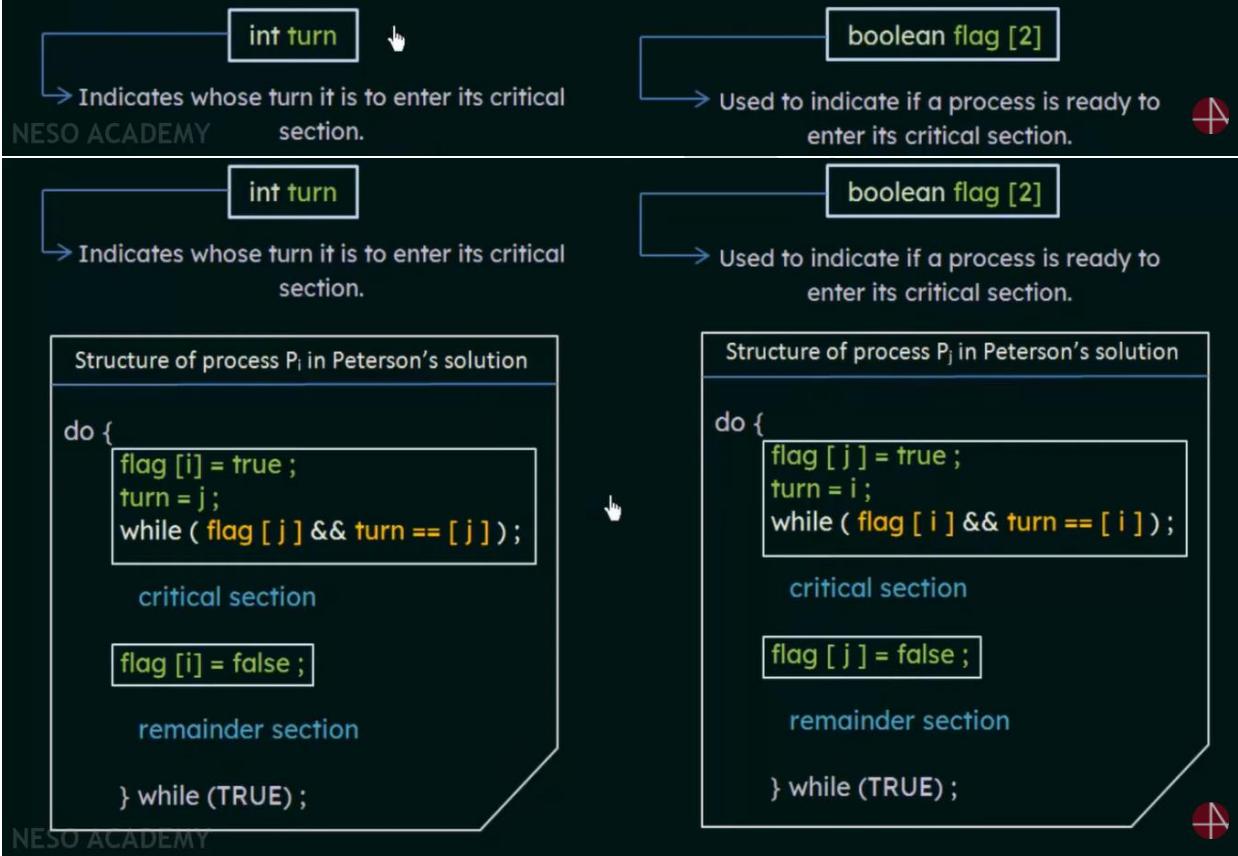


### Peterson's Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes  $P_i$  and  $P_j$

Peterson's solution requires two data items to be shared between the two processes:



A solution to the critical-section problem must satisfy the following three requirements:

**1. Mutual exclusion:**

If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.

**2. Progress:**

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely. ↴

**3. Bounded waiting:**

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.



### Test and Set Lock

- A hardware solution to the synchronization problem.
- There is a shared lock variable which can take either of the two values, 0 or 1.
- Before entering into the critical section, a process inquires about the lock.
- If it is locked, it keeps on waiting till it becomes free.
- If it is not locked, it takes the lock and executes the critical section.

```
boolean TestAndSet (boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

Atomic  
Operation

The definition of the TestAndSet () instruction



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

```
boolean TestAndSet (boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

Atomic  
Operation

The definition of the TestAndSet () instruction

```
do {  
    while (TestAndSet (&lock)) ;  
    // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```



```
do {  
    while (TestAndSet (&lock)) ;  
    // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

NESO ACADEMY



```
do {  
    while (TestAndSet (&lock)) ;  
    // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```



```
do {  
    while (TestAndSet (&lock)) ;  
    // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Satisfies mutual-exclusion.

Does not satisfy bounded-waiting.



NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

## Semaphores

- Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.



Made from 100% Bee Wax  
Contains Tea Tree Oil & Rosemary Oil to prevent ACNE also Promote Hair Growth  
gelkomb.com

NESO ACADEMY

## Semaphores

- Semaphore proposed by Edsger Dijkstra, is a technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.
- Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
- A semaphore **S** is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait ()** and **signal ()**.

**wait ()** → **P** [from the Dutch word **proberen**, which means "to test"]

**signal ()** → **V** [from the Dutch word **verhogen**, which means "to increment"]

NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Definition of wait ():

```
P (Semaphore S) {  
    while (S <= 0)  
        ; // no operation  
    S--;  
}
```

Definition of signal ():

```
V (Semaphore S) {  
    S++;  
}
```

All the modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of Semaphores:

**1. Binary Semaphore:**

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.



Definition of wait ():

```
P (Semaphore S) {  
    while (S <= 0)  
        ; // no operation  
    S--;  
}
```

Definition of signal ():

```
V (Semaphore S) {  
    S++;  
}
```

All the modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of Semaphores:

**1. Binary Semaphore:**

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.



wait () → P [from the Dutch word **proberen**, which means "to test"]

signal () → V [from the Dutch word **verhogen**, which means "to increment"]

Definition of wait ():

```
P (Semaphore S) {  
    while (S <= 0)  
        ; // no operation  
    S--;  
}
```

Definition of signal ():

```
V (Semaphore S) {  
    S++;  
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.



Definition of wait ():

```
P (Semaphore S) {  
    while (S <= 0)  
        ; // no operation  
    S--;  
}
```

Definition of signal ():

```
V (Semaphore S) {  
    S++;  
}
```

All the modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of Semaphores:

1. **Binary Semaphore:**

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.



2. **Counting Semaphore:**

Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.



### Disadvantages of Semaphores

- The main disadvantage of the semaphore definition that was discussed is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.



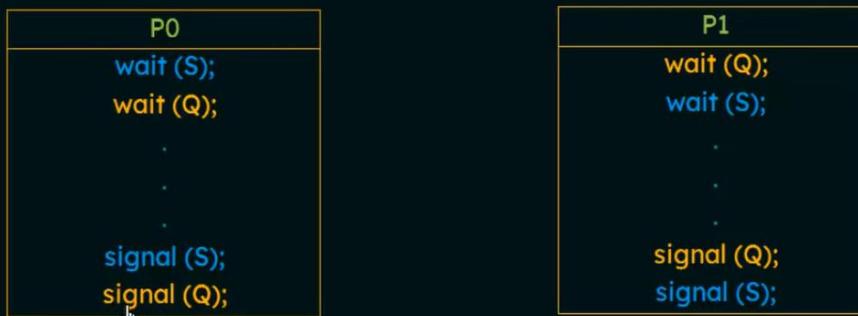
To overcome the need for busy waiting, we can modify the definition of the wait () and signal () semaphore operations.

- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.
  - However, rather than engaging in busy waiting, the process can block itself.
  - The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.



### Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked.



NESO ACADEMY



### Classic Problems of Synchronization (The Bounded-Buffer Problem)

The Bounded Buffer Problem (**Producer Consumer Problem**), is one of the classic problems of synchronization.

There is a buffer of  $n$  slots and each slot is capable of storing one unit of data.

There are two processes running, namely, Producer and Consumer, which are operating on the buffer.



NESO ACADEMY



There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, Producer and Consumer, which are operating on the buffer.



- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.



#### Solution to the Bounded Buffer Problem using Semaphores:

We will make use of three semaphores:

1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.



1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.

Producer	Consumer
<pre>do {     wait (empty); // wait until empty&gt;0         and then decrement 'empty'     wait (mutex); // acquire lock     /* add data to buffer */     signal (mutex); // release lock     signal (full); // increment 'full' } while(TRUE)</pre>	<pre>do {     wait (full); // wait until full&gt;0 and         then decrement 'full'     wait (mutex); // acquire lock     /* remove data from buffer */     signal (mutex); // release lock     signal (empty); // increment 'empty' } while(TRUE)</pre>



## Classic Problems of Synchronization (The Readers-Writers Problem)

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as Readers and to the latter as Writers.
- Obviously, if two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.

This synchronization problem is referred to as the readers-writers problem.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

#### Solution to the Readers-Writers Problem using Semaphores:

We will make use of two semaphores and an integer variable:

1. mutex, a semaphore (initialized to 1) which is used to ensure mutual exclusion when readcount is updated i.e. when any reader enters or exit from the critical section.
2. wrt, a semaphore (initialized to 1) common to both reader and writer processes.
3. readcount, an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.

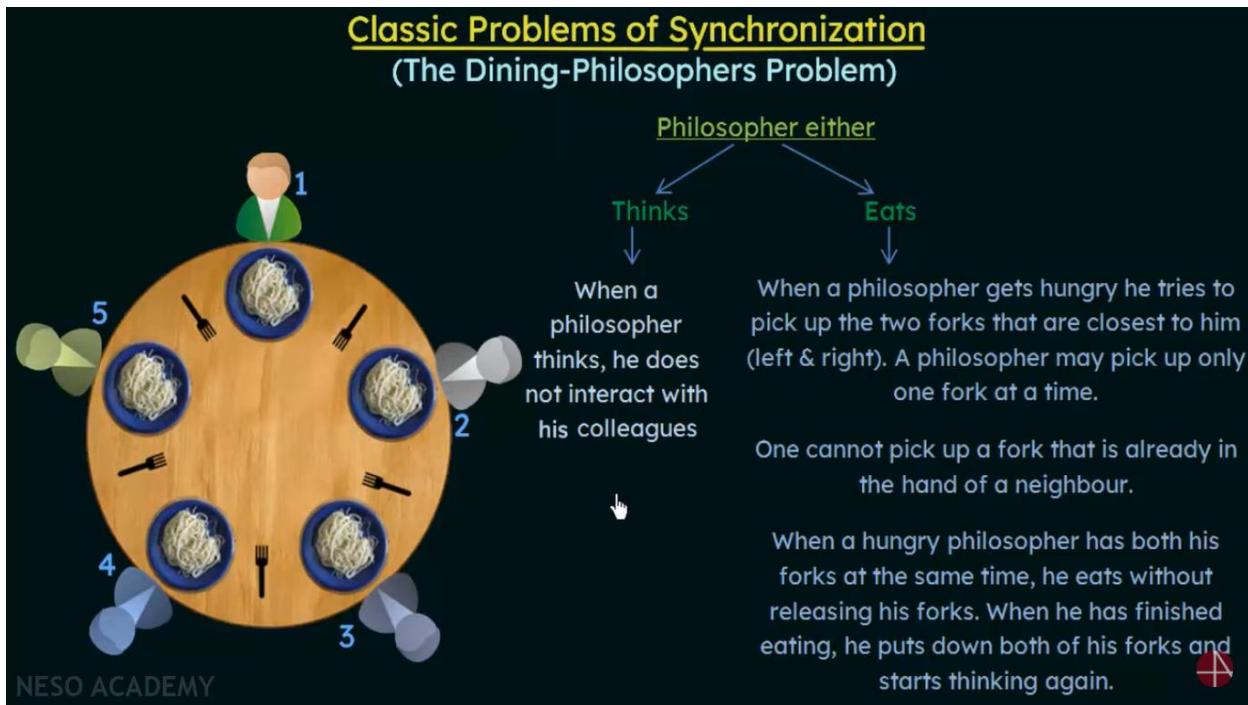
NESO ACADEMY



Writer Process	Reader Process
<pre>do {     /* writer requests for critical     section */     wait(mutex);     /* performs the write */     // leaves the critical section     signal(mutex); } while(true);</pre>	<pre>do {     wait (mutex);     readcnt++; // The number of readers has now increased by 1     if (readcnt==1)         wait (wrt); // this ensure no writer can enter if there is even one reader         signal (mutex); // other readers can enter while this current reader is         // inside the critical section     /* current reader performs reading here */     wait (mutex);     readcnt--; // a reader wants to leave     if (readcnt == 0) //no reader is left in the critical section         signal (wrt); // writers can enter         signal (mutex); // reader leaves     } while(true);</pre>

NESO ACADEMY





One simple solution is to represent each fork/chopstick with a semaphore.

A philosopher tries to grab a fork/chopstick by executing a wait () operation on that semaphore.

He releases his fork/chopsticks by executing the signal () operation on the appropriate semaphores.

Thus, the shared data are

```
semaphore chopstick[5] ;
```

where all the elements of chopstick are initialized to 1.



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

where all the elements of chopstick are initialized to 1.

The structure of philosopher i

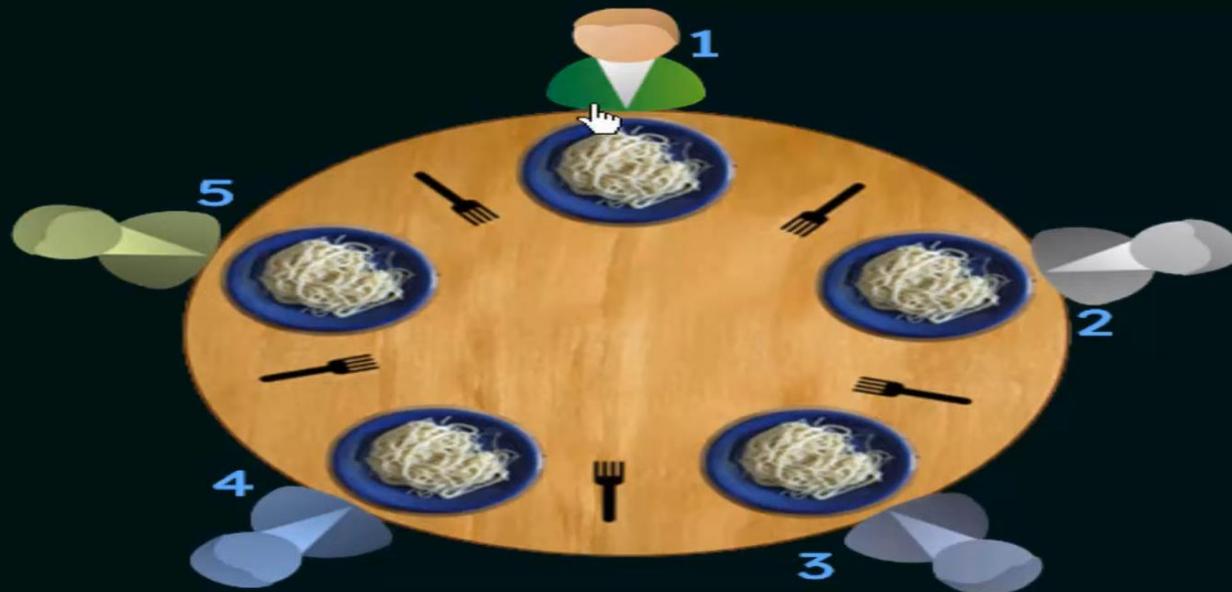
```
do {  
    wait (chopstick [i]);  
    wait(chopstick [(i + 1) % 5]);  
    ....  
    // eat  
    signal(chopstick [i]);  
    signal(chopstick [(i + 1) % 5]);  
    // think  
}while (TRUE);
```

Although this solution guarantees that no two neighbors are eating simultaneously, it could still create a deadlock.

Suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick. All the elements of chopstick will now be equal to 0.

When each philosopher tries to grab his right chopstick, he will be delayed forever.

NESO ACADEMY



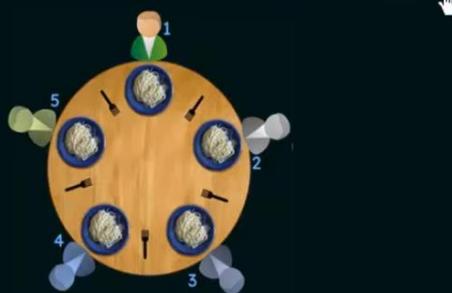
NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

#### Some possible remedies to avoid deadlocks:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available (to do this he must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.



#### Monitors

- A high level abstraction that provides a convenient and effective mechanism for process synchronization.
- A monitor type presents a set of programmer-defined operations that provide mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.



```
Syntax of a Monitor
monitor monitor_name
{
    // shared variable declarations
    procedure P1 (...) {
        ...
    }
    procedure P2 (...) {
        ...
    }
    .
    .
    procedure Pn (...) {
        ...
    }
    initialization code (...) {
        ...
    }
}
```

NESO ACADEMY

- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- Similarly, the local variables of a monitor can be accessed by only the local procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.

Condition Construct-      **condition x, y;**

The only operations that can be invoked on a condition variable are **wait ()** and **signal ()**.

The operation **x.wait()**; means that the process invoking this operation is suspended until another process invokes **x.signal()**;  
 The **x. signal ()** operation resumes exactly one suspended process.

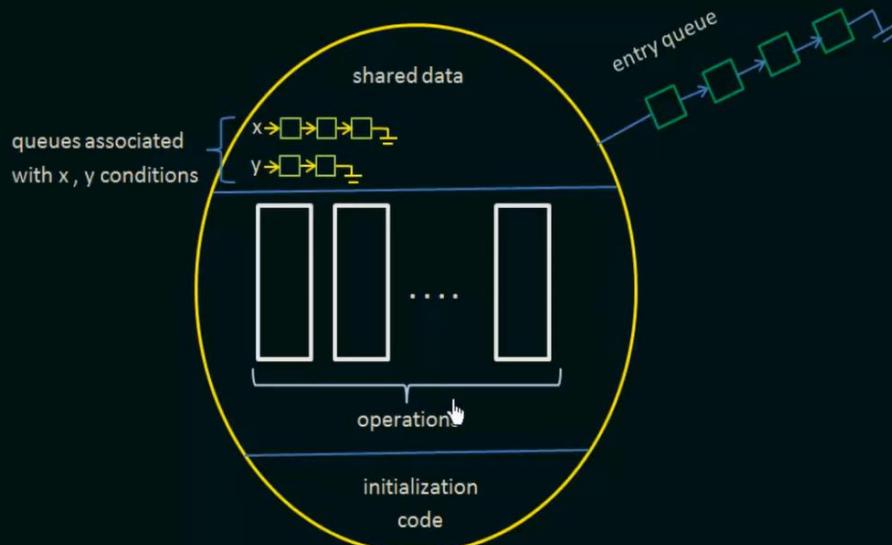


Fig: Schematic view of a monitor

NESO ACADEMY



## Dining-Philosophers Solution Using Monitors

- We now illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem.
- This solution imposes the restriction that a philosopher may pick up his chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum { thinking, hungry, eating } state [5];
```

- Philosopher *i* can set the variable state [*i*] = eating only if his two neighbors are not eating: (state [(*i*+4) % 5] != eating) and (state [(*i*+1)% 5] != eating).
- We also need to declare condition self [5]; where philosopher *i* can delay himself when he is hungry but is unable to obtain the chopsticks he needs. ↗



A monitor solution to the dining-philosopher problem

```
monitor dp {
    enum { THINKING, HUNGRY, EATING } state [5];
    condition self [5];

    void pickup (int i) {
        state [i] = HUNGRY;
        test (i);
        if (state [i] != EATING)
            self [i].wait();
    }

    void putdown(int i) {
        state [i] = THINKING;
        test ((i + 4) % 5);
        test ((i + 1) % 5);
    }

    void test (int i) {
        if ((state [(i + 4) % 5] != EATING) &&
            (state [i] == HUNGRY) &&
            (state [(i + 1) % 5] != EATING))
            state [i] = EATING;
            self [i].signal();
    }

    initialization-code () {
        for (int i = 0; i < 5; i++)
            state [i] = THINKING;
    }
}
```



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

## Process Synchronization Solved Problems (Part-1)

*Question:*

GATE 2010

Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned.

Method Used by P1	Method Used by P2
while ( $S1 == S2$ ) ; Critical Section $S1 = S2;$	while ( $S1 != S2$ ) ; Critical Section $S2 = \text{not } (S1);$

NESO ACADEMY



*Question:*

GATE 2010

Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned.

Method Used by P1	Method Used by P2
while ( $S1 == S2$ ) ; Critical Section $S1 = S2;$	while ( $S1 != S2$ ) ; Critical Section $S2 = \text{not } (S1);$

Which one of the following statements describes the properties achieved?

- (a) Mutual exclusion but not progress
- (b) Progress but not mutual exclusion
- (c) Neither mutual exclusion nor progress
- (d) Both mutual exclusion and progress

NESO ACADEMY



A solution to the critical-section problem must satisfy the following three requirements:

**1. Mutual exclusion:**

If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.

**2. Progress:**

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.



**3. Bounded waiting:**

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.



## Process Synchronization Solved Problems (Part-2)

Question:

GATE 2010

The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as  $S0=1$ ,  $S1=0$ ,  $S2=0$ .

Process P0	Process P1	Process P2
while (true) { wait (S0); print (0); release (S1); release (S2); }	wait (S1); release (S0);	wait (S2); release (S0);



Question:

GATE 2010

The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0=1, S1=0, S2=0.

Process P0	Process P1	Process P2
<pre>while (true) {     wait (S0);     print (0);     release (S1);     release (S2); }</pre>	<pre>wait (S1); release (S0);</pre>	<pre>wait (S2); release (S0);</pre>



How many times will process P0 print '0' ?

- (a) At least twice
- (b) Exactly twice
- (c) Exactly thrice
- (d) Exactly once



### Process Synchronization Solved Problems (Part-3)

Question:

GATE 2007

Two processes, P1 and P2, need to access a critical section of code. Consider the following synchronization construct used by the processes:

Process P1	Process P2
<pre>while (true) {     wants1 = true;     while (wants2 == true); <span style="color: yellow;">←</span>     /* Critical Section */     wants1=false; } /* Remainder section */</pre>	<pre>while (true) {     wants2 = true;     while (wants1==true);     /* Critical Section */     wants2 = false; } /* Remainder section */</pre>



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Process P1	Process P2
<pre>while (true) {     wants1 = true;     while (wants2 == true);     /* Critical Section */     wants1=false; } /* Remainder section */</pre>	<pre>while (true) {     wants2 = true;     while (wants1==true);     /* Critical Section */     wants2 = false; } /* Remainder section */</pre>

Here, wants1 and wants2 are shared variables, which are initialized to false. Which one of the following statements is TRUE about the above construct?

- (a) It does not ensure mutual exclusion.
- (b) It does not ensure bounded waiting.
- (c) It requires that processes enter the critical section in strict alternation.
- (d) It does not prevent deadlocks, but ensures mutual exclusion.

NESO ACADEMY



### Process Synchronization Solved Problems (Part-4)

Question:

GATE 2019

Consider three concurrent processes P1, P2 and P3 as shown below, which access a shared variable D that has been initialized to 100.

P1	P2	P3
.	.	.
D = D + 20	D = D - 50	D = D + 10
.	.	.

NESO ACADEMY



NAME – RUSHIKESH SHARAD MANE  
SUBJECT- OPERATING SYSTEM(OS)  
TOPIC-1. INTRODUCTION TO OPERATING SYSTEM.

Consider three concurrent processes P1, P2 and P3 as shown below, which access a shared variable D that has been initialized to 100.

P1	P2	P3
.	.	.
$D = D + 20$	$D = D - 50$	$D = D + 10$
.	.	.
.	.	.

The process are executed on a uniprocessor system running a time-shared operating system. If the minimum and maximum possible values of D after the three processes have completed execution are X and Y respectively, then the value of Y-X is \_\_\_\_\_.

- (a) 80
- (b) 130
- (c) 50

NESO (d) None of these



### Process Synchronization Solved Problems (Part-5)

Question:

GATE 1998

A counting semaphore was initialized to 10. Then 6 P (wait) operations and 4V (signal) operations were completed on this semaphore. The resulting value of the semaphore is

- (a) 0
- (b) 8
- (c) 10
- (d) 12



NESO ACADEMY

