CDAC MUMBAI

Concepts of Operating System

Assignment 2

# Part A

What will the following commands do?

1. echo "Hello, World!"

Ans : Prints the text **Hello, World!** to the terminal.

1. name="Productive"

Ans : Assigns the string **Productive** to the variable **name**.

1. touch file.txt

Ans : Creates an empty file named **file.txt** if it doesn't exist, or updates its last modified timestamp if it already exists.

1. ls -a

Ans : Lists all files and directories in the current directory, including hidden files (those starting with .).

1. rm file.txt

Ans : Deletes the file **file.txt**.

1. cp file1.txt file2.txt

Ans ; Copies **file1.txt** to **file2.txt** (creates a duplicate).

1. mv file.txt /path/to/directory/

Ans : Moves **file.txt** to the specified directory

1. chmod 755 script.sh

Ans : Changes the permissions of **script.sh** to:

* Owner: Read, Write, Execute
* Group: Read, Execute
* Others: Read, Execute

1. grep "pattern" file.txt

Ans : Searches for the string **pattern** in **file.txt** and prints the matching lines.

1. kill PID

Ans : Terminates the process with the specified **PID** (Process ID).

1. mkdir mydir && cd mydir && touch file.txt && echo "Hello, World!" > file.txt && cat file.txt

Ans : This sequence of commands:

* Creates a directory **mydir**
* Changes to **mydir**
* Creates an empty file **file.txt**
* Writes **Hello, World!** into **file.txt**
* Displays the content of **file.txt**

1. ls -l | grep ".txt"

Ans : Lists files in long format and filters only those with **.txt** in their names.

1. cat file1.txt file2.txt | sort | uniq

Ans : Concatenates **file1.txt** and **file2.txt**, sorts the combined output, and removes duplicate lines.

1. ls -l | grep "^d"

Ans : Lists only directories (in long format), since **^d** filters lines starting with **d** (which represents directories in ls -l).

1. grep -r "pattern" /path/to/directory/

Ans : Recursively searches for **pattern** in all files inside **/path/to/directory/** and its subdirectories.

1. cat file1.txt file2.txt | sort | uniq –d

Ans : Concatenates **file1.txt** and **file2.txt**, sorts the output, and displays only duplicate lines.

1. chmod 644 file.txt

Ans : Sets the permissions of **file.txt** to:

* Owner: Read, Write
* Group: Read
* Others: Read

1. cp -r source\_directory destination\_directory

Ans : Recursively copies **source\_directory** and its contents to **destination\_directory**.

1. find /path/to/search -name "\*.txt"

Ans : Searches for files with the **.txt** extension in the specified path and its subdirectories.

1. chmod u+x file.txt

Ans : Grants execute permission to the file owner (**u** means user).

1. echo $PATH

Ans : Displays the system's **PATH** environment variable, which contains directories where executable files are searched for.

# Part B

Identify True or False:

1. ls is used to list files and directories in a directory.

Ans : True

1. mv is used to move files and directories.

Ans : True

1. cd is used to copy files and directories.

Ans : False

1. pwd stands for "print working directory" and displays the current directory.

Ans : True

1. grep is used to search for patterns in files.

Ans : True

1. chmod 755 file.txt gives read, write, and execute permissions to the owner, and read and execute permissions to group and others.

Ans : True

1. mkdir -p directory1/directory2 creates nested directories, creating directory2 inside directory1 if directory1 does not exist.

Ans : True

1. rm -rf file.txt deletes a file forcefully without confirmation.

Ans : True

Identify the Incorrect Commands:

1. chmodx is used to change file permissions.

Ans : Correct command: **chmod** (used to change file permissions)

1. cpy is used to copy files and directories.

Ans : Correct command: **cp** (used to copy files and directories)

1. mkfile is used to create a new file.

Ans : Correct command: **touch** (used to create a new file)

1. catx is used to concatenate files.

Ans : Correct command: **cat** (used to concatenate and display files)

1. rn is used to rename files.

Ans : Correct command: **mv** used to rename or move files

# Part C

Question 1: Write a shell script that prints "Hello, World!" to the terminal.

Question 2: Declare a variable named "name" and assign the value "CDAC Mumbai" to it. Print the value of the variable.

Question 3: Write a shell script that takes a number as input from the user and prints it.

Question 4: Write a shell script that performs addition of two numbers (e.g., 5 and 3) and prints the result.

Question 5: Write a shell script that takes a number as input and prints "Even" if it is even, otherwise prints "Odd".

Question 6: Write a shell script that uses a for loop to print numbers from 1 to 5.

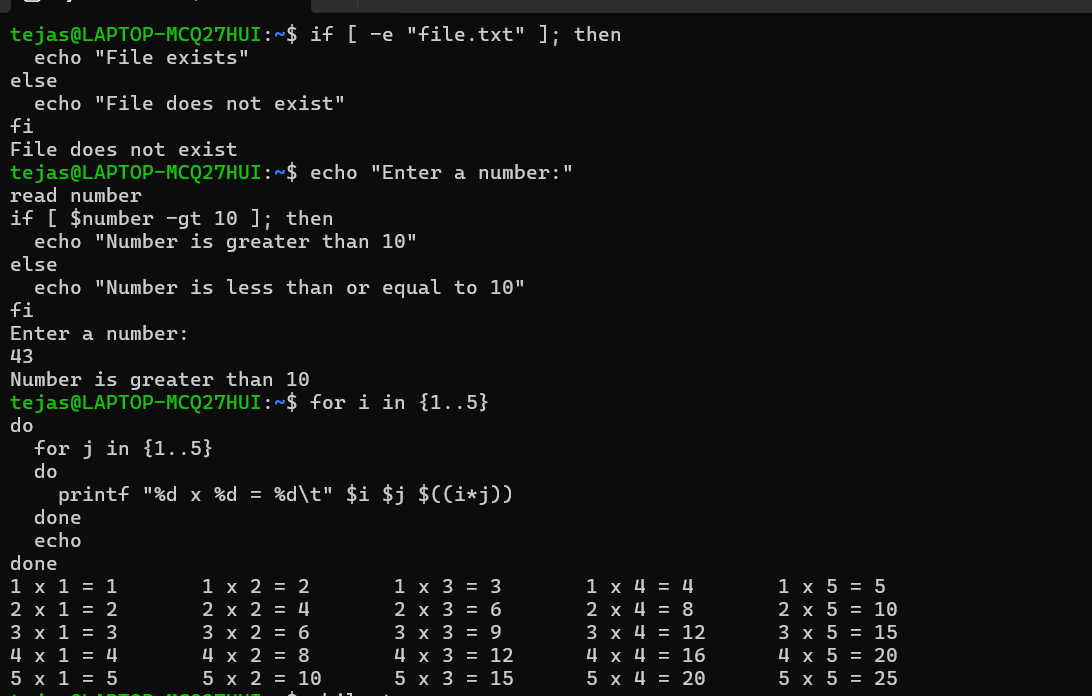
Question 7: Write a shell script that uses a while loop to print numbers from 1 to 5.

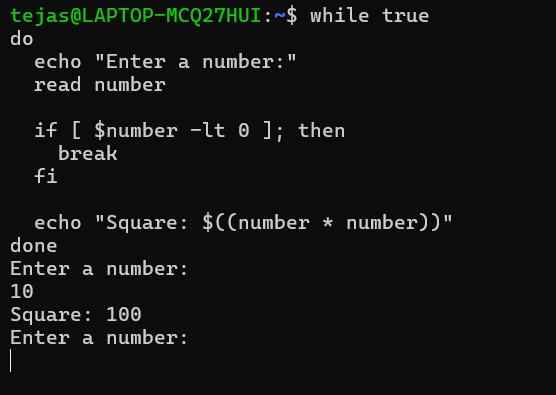
Question 8: Write a shell script that checks if a file named "file.txt" exists in the current directory. If it does, print "File exists", otherwise, print "File does not exist".

Question 9: Write a shell script that uses the if statement to check if a number is greater than 10 and prints a message accordingly.

Question 10: Write a shell script that uses nested for loops to print a multiplication table for numbers from 1 to 5. The output should be formatted nicely, with each row representing a number and each column representing the multiplication result for that number.

Question 11: Write a shell script that uses a while loop to read numbers from the user until the user enters a negative number. For each positive number entered, print its square. Use the break statement to exit the loop when a negative number is entered.





# Part D

Common Interview Questions (Must know)

1. What is an operating system, and what are its primary functions?

Ans : An **Operating System (OS)** is system software that manages computer hardware and software resources and provides services to computer programs. It acts as an interface between the user and the computer hardware.

**Primary Functions of an Operating System:**

* **Process Management:** Manages running applications and processes.
* **Memory Management:** Allocates and manages computer memory (RAM).
* **File Management:** Organizes and controls files on storage devices.
* **Device Management:** Controls peripheral devices like printers, keyboards, and monitors.
* **User Interface:** Provides interaction between users and the computer (Graphical or Command Line Interface).
* **Security Management:** Protects data and resources from unauthorized access.
* **Input/Output Management:** Manages input from devices and output to devices.

1. Explain the difference between process and thread.

Ans :

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Process** | **Thread** | | A process is an independent program in execution. | A thread is a smaller part of a process that executes independently. | | It has its own memory space. | Threads share the same memory space of the process. | | Processes are heavier and require more resources. | Threads are lighter and need fewer resources. | | Communication between processes is slower. | Communication between threads is faster because they share memory. | | If one process fails, it does not affect other processes. | If one thread fails, it can affect other threads in the same process. | |

1. What is virtual memory, and how does it work?

Ans : **Virtual Memory** is a memory management technique used by the operating system to make the computer feel like it has more RAM than it actually does.

**Working :**

* When the RAM is full, the operating system temporarily transfers some data from RAM to a special space on the **hard disk** or **SSD**.
* This special space is called the **Virtual Memory (Page File)**.
* The operating system automatically moves data back and forth between RAM and virtual memory when needed.
* This helps the computer run large programs or multiple applications at the same time without crashing.

1. Describe the difference between multiprogramming, multitasking, and multiprocessing.

Ans :

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Multiprogramming** | **Multitasking** | **Multiprocessing** |
| Definition | Running **multiple programs** in memory at the same time, but only **one program runs at a time**. | Running **multiple tasks (programs)** at the same time by quickly switching between them. | Using **two or more CPUs** to run multiple tasks at the same time. |
| Execution Speed | Slower | Faster than multiprogramming | Fastest because it uses more CPUs |
| CPU Usage | Only **one program** uses the CPU at a time | CPU switches between tasks | Multiple CPUs work together |
| Example | Printing a document while downloading a file (One at a time) | Listening to music while typing in MS Word | Servers running multiple applications at the same time |
| Cost | Low | Moderate | High (because it needs more CPUs) |

1. What is a file system, and what are its components?

Ans : A **File System** is a method used by the operating system to organize, manage, and store data on storage devices like hard disks, SSDs, or pen drives. It helps the operating system to locate and retrieve files easily. The file system controls how data is stored, accessed, and managed in the storage device.

**Components of a File System:**

1. **Files:**  
   Files are collections of data or information stored on a storage device. Each file has a name, type, and size. Examples of files include documents, images, audio, and video files.
2. **Directories(Folders):**  
   Directories are used to group multiple files together. They help to organize files in a structured way, making it easier to access and manage data.
3. **Inodes(IndexNodes):**  
   Inodes store important information about files, such as file size, file location, creation date, and permissions. Every file has a unique inode number.
4. **Superblock:**  
   The superblock contains key information about the file system, such as its size, block size, and the number of files it can store. It acts as the main record of the file system structure.
5. **DiskBlocks:**  
   Disk blocks are small units of storage where the actual file data is saved. Large files are divided into several blocks, and each block is stored separately.
6. **FileAllocationTable(FAT):**  
   The file allocation table is used to keep track of which disk blocks are used by files and which are free. It helps the system find and organize file data efficiently.
7. What is a deadlock, and how can it be prevented?

Ans : A **Deadlock** is a situation in an operating system where two or more processes are unable to proceed because each process is waiting for a resource that another process is holding. This causes the processes to remain in an infinite waiting state, and the system cannot perform further tasks. Deadlocks usually occur in systems where resources are shared among multiple processes.

**Deadlock Prevention:**

Deadlock can be prevented by ensuring that at least one of the four necessary conditions does not occur. The following methods are used to prevent deadlock:

1. **Mutual Exclusion:** Some resources can be made sharable, so multiple processes can use them at the same time (like read-only files).
2. **Hold and Wait:** Processes must request all the resources they need at once before execution starts, so they do not wait for additional resources.
3. **No Preemption:** The operating system can force a process to release resources if the process is waiting for other resources.
4. **Circular Wait:** The system assigns a unique number to each resource, and processes must request resources in increasing order of their numbers, breaking the circular chain.
5. Explain the difference between a kernel and a shell.

Ans :

|  |  |
| --- | --- |
| **Kernel** | **Shell** |
| Core part of the operating system | Interface between user and operating system |
| Manages hardware resources like CPU, memory, and devices | Takes user commands and passes them to the kernel |
| Works in the background | Works in the foreground |
| Low-level program | High-level program |
| Cannot interact directly with users | Directly interacts with users |
| Examples: Linux Kernel, Windows NT Kernel | Examples: Bash, Command Prompt, PowerShell |

1. What is CPU scheduling, and why is it important?

Ans : **CPU Scheduling** is the process of deciding which process or task will use the CPU at a given time when multiple tasks are waiting to be executed. The **CPU Scheduler** selects one process from the ready queue and assigns the CPU to that process.

CPU scheduling is important because:

* It ensures **fair use of CPU** among all processes.
* It helps in **improving system performance** by reducing waiting time.
* It makes sure that no process is **left waiting for too long**.
* It increases **CPU efficiency** by keeping the CPU busy as much as possible.
* It helps in **multitasking**, allowing multiple programs to run at the same time.

1. How does a system call work?

Ans : A **System Call** is a request made by a program to the operating system to perform certain tasks that the program cannot do directly.

The process of a system call works in these steps:

1. **User Request:** The user program requests a service (like reading a file or printing data).
2. **System Call Execution:** The request is sent to the operating system using a system call.
3. **Kernel Mode:** The operating system switches to **kernel mode** (a secure mode where the OS can access hardware).
4. **Task Execution:** The operating system performs the requested task.
5. **Return Result:** The result is sent back to the user program.
6. **User Mode:** The system switches back to **user mode** after completing the task.
7. What is the purpose of device drivers in an operating system?

Ans : A **Device Driver** is a special type of software that allows the operating system to communicate with hardware devices. The operating system itself cannot directly control hardware like printers, keyboards, or hard drives, so it uses device drivers to perform this task.

**Purpose of Device Drivers:**

1. **Communication with Hardware Devices:**  
   Device drivers enable the operating system to communicate with hardware devices like printers, scanners, or USB drives.
2. **Hardware Compatibility:**  
   They ensure that different hardware devices work with the operating system, even if they are made by different manufacturers.
3. **Simplifies User Interaction:**  
   Users don't need to know how hardware works because the device driver handles all the technical details.
4. **Hardware Control:**  
   Drivers allow the operating system to control and manage hardware devices easily.
5. **Automatic Updates and Bug Fixes:**  
   Many device drivers can be updated to improve device performance or fix errors.
6. Explain the role of the page table in virtual memory management.

Ans : A **Page Table** is a data structure used by the operating system to manage **virtual memory**. It helps the operating system keep track of the mapping between **virtual addresses** (used by programs) and **physical addresses** (actual memory locations in RAM)

The **main role of the page table** is to:

1. **Translate Virtual Addresses to Physical Addresses:**  
   It converts virtual memory addresses into physical memory addresses.
2. **Memory Protection:**  
   It protects memory by ensuring that one program cannot access another program’s memory.
3. **Efficient Memory Management:**  
   Helps the operating system manage memory efficiently by dividing memory into small fixed-sized blocks called **pages**.
4. **Page Fault Handling:**  
   If a program tries to access data not currently in RAM, the page table helps the operating system find the data on the hard disk.
5. **Memory Sharing:**  
   Allows multiple programs to share the same memory space without interfering with each other.

1. What is thrashing, and how can it be avoided?

Ans : **Thrashing** is a condition in an operating system where the computer spends most of its time **swapping data between RAM and virtual memory (hard disk)** instead of executing actual tasks.

* **Increase RAM Size:**  
  By adding more physical memory, the operating system can hold more processes in RAM, reducing the need for swapping pages frequently.
* **Limit Degree of Multiprogramming:**  
  Reducing the number of active processes running at the same time helps the system allocate sufficient memory to each process, preventing frequent page faults.
* **Page Replacement Algorithms:**  
  Using efficient algorithms like **Least Recently Used (LRU)** or **Optimal Page Replacement** ensures that only the least important pages are removed from memory.
* **Working Set Model:**  
  This method tracks the set of pages each process is currently using and keeps those pages in memory to avoid unnecessary swapping.
* **Proper Allocation of Frames:**  
  Allocating enough memory frames to each process based on its requirements helps minimize page faults.
* **Priority Scheduling:**  
  Giving higher priority to important tasks and limiting low-priority processes can help maintain system performance

1. Describe the concept of a semaphore and its use in synchronization.

Ans : A **Semaphore** is a special variable or data structure used in operating systems to control access to shared resources by multiple processes. It is a **synchronization tool** that helps prevent two or more processes from accessing the same resource at the same time, which could lead to errors.

Semaphores help in **synchronizing processes** by ensuring that only one process accesses a resource at a time.

1. How does an operating system handle process synchronization?

Ans : **Process Synchronization** is a method used by the operating system to ensure that multiple processes running at the same time do not interfere with each other when accessing shared resources.

The operating system uses various techniques to handle process synchronization:

1. **Mutex (Mutual Exclusion):**  
   Allows only one process to access a shared resource at a time. If one process is using the resource, others must wait.
2. **Semaphore:**  
   A signaling mechanism that controls the number of processes accessing a resource at the same time.
3. **Critical Section:**  
   A part of the program where shared resources are accessed. The operating system ensures that only one process enters the critical section at a time.
4. **Monitors:**  
   A high-level synchronization technique that uses procedures to control access to shared resources.
5. **Locks:**  
   Used to lock resources, so only one process can access them until the lock is released.
6. What is the purpose of an interrupt in operating systems?

Ans : An **Interrupt** is a signal sent by hardware or software to the operating system, indicating that an immediate action or attention is required. It temporarily stops the current execution of a process, allowing the operating system to handle a specific task or event before continuing with the previous process.

**Purpose of Interrupt in Operating Systems:**

1. **Efficient CPU Usage:**  
   Interrupts help the CPU perform tasks efficiently by allowing it to respond to external events without constantly checking for them.
2. **Asynchronous Event Handling:**  
   They allow the operating system to handle external events like mouse clicks, keyboard inputs, or hardware failures.
3. **Multitasking Support:**  
   Interrupts enable the system to switch between multiple tasks, supporting **multitasking**.
4. **Error Detection:**  
   They alert the system about errors such as memory overflow or hardware malfunctions.
5. **Device Communication:**  
   Hardware devices like printers and USB drives use interrupts to signal when they need CPU attention.
6. **Prioritization of Tasks:**  
   Important tasks can interrupt less important ones, ensuring that critical operations are handled first.
7. Explain the concept of a file descriptor.

Ans : A **File Descriptor** is a unique integer number assigned by the operating system to identify and manage open files or input/output resources like files, sockets, or devices. It acts as a reference that allows the operating system to track which files are being accessed by a process. When a process opens a file, the operating system assigns a file descriptor to represent that file, and the process uses this number to perform operations such as reading, writing, or closing the file. File descriptors are essential for managing input and output operations in operating systems, as they help separate standard input, output, and error messages. They also play a crucial role in inter-process communication, where data is exchanged between different processes using sockets or pipes. File descriptors improve system efficiency by simplifying file management and ensuring that each open file is uniquely identified and managed by the operating system.

1. How does a system recover from a system crash?

Ans: A **system crash** happens when a computer suddenly stops working due to software errors, hardware problems, or power failures. To recover from a system crash, the operating system restarts the computer automatically. It checks for any damaged files and tries to fix them. The system may also use **backup files** or **restore points** to recover lost data. If the crash happened during a task, the system will undo incomplete actions to avoid data corruption. Error logs are created to find out what caused the crash. These steps help the system to start working again and prevent data loss.

1. Describe the difference between a monolithic kernel and a microkernel.

Ans : The **Kernel** is the core part of an operating system that manages hardware and software communication. There are two main types of kernels: **Monolithic Kernel** and **Microkernel**.

A **Monolithic Kernel** is a large and single program where all essential services like memory management, process management, and device drivers run in the same space as the kernel. It provides faster performance because all services run together, but if one service fails, the entire system may crash. Examples of monolithic kernels include **Linux** and **Unix**.

On the other hand, a **Microkernel** is smaller and only performs basic functions like communication between hardware and software. Other services like file management and device drivers run separately in user space. This design makes the system more secure and stable because a crash in one service does not affect the whole system. However, microkernels are slower due to more communication between services. Examples of microkernels include **QNX** and **Minix**.

1. What is the difference between internal and external fragmentation?

Ans : The **Kernel** is the core part of an operating system that manages hardware and software communication. There are two main types of kernels: **Monolithic Kernel** and **Microkernel**.

A **Monolithic Kernel** is a large and single program where all essential services like memory management, process management, and device drivers run in the same space as the kernel. It provides faster performance because all services run together, but if one service fails, the entire system may crash. Examples of monolithic kernels include **Linux** and **Unix**.

On the other hand, a **Microkernel** is smaller and only performs basic functions like communication between hardware and software. Other services like file management and device drivers run separately in user space. This design makes the system more secure and stable because a crash in one service does not affect the whole system. However, microkernels are slower due to more communication between services. Examples of microkernels include **QNX** and **Minix**.

1. How does an operating system manage I/O operations?

Ans : The **Operating System (OS)** manages **Input/Output (I/O) operations** to allow communication between the computer and external devices like keyboards, printers, hard drives, and monitors. It acts as a bridge between the hardware devices and user applications.

The OS uses **I/O device drivers**, which are special programs that help the operating system understand how to interact with hardware devices. It also uses **I/O controllers** to send and receive data between devices and the CPU. The OS organizes I/O requests in a queue and processes them based on priority to avoid conflicts. It provides different I/O methods such as **Programmed I/O**, **Interrupt-driven I/O**, and **Direct Memory Access (DMA)** to improve efficiency.

The operating system ensures that devices are ready before sending data and handles errors during data transfer. It also allows multiple applications to share devices without interference through **buffering, caching, and spooling techniques**. Overall, the operating system manages I/O operations to ensure smooth, fast, and error-free communication between the system and external devices.

1. Explain the difference between preemptive and non-preemptive scheduling.

Ans : **CPU Scheduling** is the process where the operating system decides which process will use the CPU at a particular time. There are two types of scheduling: **Preemptive Scheduling** and **Non-Preemptive Scheduling**.

In **Preemptive Scheduling**, the CPU can be taken away from a running process before it completes if a higher-priority process arrives. It allows better multitasking and faster response time, but it creates more overhead because the system needs to switch between processes frequently. Examples of preemptive scheduling algorithms are **Round Robin** and **Priority Scheduling**.

On the other hand, **Non-Preemptive Scheduling** allows a running process to continue using the CPU until it finishes or goes into a waiting state. The CPU is not taken away during execution, making this method easier to implement and reducing overhead. However, it can cause longer waiting times for other processes. Examples of non-preemptive scheduling algorithms are **First-Come, First-Serve (FCFS)** and **Shortest Job Next (SJN)**.

1. What is round-robin scheduling, and how does it work?

Ans : **Round-Robin Scheduling** is one of the most popular **CPU scheduling algorithms** used in operating systems. It is designed to give each process a fair chance to use the CPU by dividing the CPU time equally among all processes.

In round-robin scheduling, each process is assigned a fixed time period called a **Time Quantum** or **Time Slice**. The CPU executes each process for the given time slice. If a process completes within that time, it is removed from the queue. However, if the process is not finished, it is moved to the back of the queue, and the next process is given the CPU.

This method ensures that no process waits too long, making it ideal for **time-sharing systems**. It provides **equal CPU time** to all processes and prevents one process from dominating the CPU. Although it gives fair treatment to all processes, the performance depends on the size of the time quantum. A smaller time quantum improves response time but increases CPU overhead due to frequent switching, while a larger time quantum may cause longer waiting times.

1. Describe the priority scheduling algorithm. How is priority assigned to processes?

Ans : **Priority Scheduling Algorithm** is a type of **CPU scheduling algorithm** where each process is assigned a priority, and the CPU is given to the process with the highest priority first. It ensures that important tasks are executed before less important ones.

**Ways to Assign Priority:**

1. **User-Defined Priority:**  
   The user or system administrator assigns the priority manually while submitting the process. Important tasks like system updates may get higher priority than normal user applications.
2. **System-Defined Priority:**  
   The operating system automatically assigns priority based on certain factors such as:
   * Process Type (System processes are given higher priority than user processes)
   * Memory Usage
   * CPU Time Requirement
   * Input/Output Operations (I/O processes may get higher priority than CPU-bound processes)
   * Waiting Time (Longer waiting processes may get higher priority to avoid **starvation**)
3. What is the shortest job next (SJN) scheduling algorithm, and when is it used?

Ans : The **Shortest Job Next (SJN)** scheduling algorithm, also known as **Shortest Job First (SJF)**, is a type of **CPU scheduling algorithm** where the process with the **smallest burst time (execution time)** is selected first for execution. This means the process that needs the least amount of CPU time is given priority.

**When is SJN Used?**

* Used in **batch processing systems** where all processes are known beforehand.
* Suitable when the execution time of all processes is predictable.
* Helps improve **CPU efficiency** and reduces **waiting time**

1. Explain the concept of multilevel queue scheduling.

Ans : **Multilevel Queue Scheduling** is a CPU scheduling algorithm that divides the ready queue into multiple smaller queues based on the type or priority of processes. Each queue handles a specific category of processes such as system processes, interactive processes, or batch processes. Higher-priority queues, like system processes, are executed first, while lower-priority queues, like batch processes, are executed later. Each queue can have its own scheduling algorithm, such as Round Robin or First-Come, First-Serve (FCFS), to manage the processes within it. Once a process is assigned to a queue, it usually cannot move to another queue. This method helps the operating system handle different types of tasks efficiently and gives preference to important processes. However, one drawback is that lower-priority processes may face **starvation**, where they wait for long periods without getting executed. Despite this, multilevel queue scheduling is widely used in systems where different types of processes need to be managed at the same time.

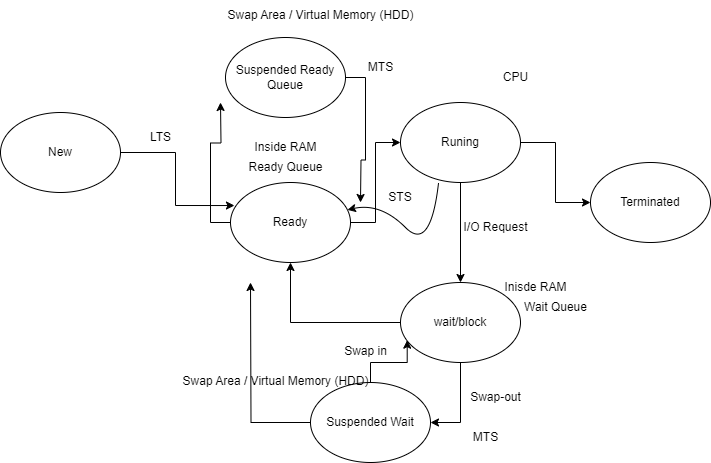
1. What is a process control block (PCB), and what information does it contain?

Ans : A **Process Control Block (PCB)** is a data structure used by the operating system to store important information about a process. It acts like an **identity card** for each process, helping the operating system manage and track processes.

**Information Contained in PCB:**

1. **Process ID:** Unique identification number for each process.
2. **Process State:** Current state of the process (Running, Ready, Waiting, or Terminated).
3. **CPU Registers:** Stores the values of CPU registers when the process is paused.
4. **Priority:** The priority level of the process.
5. **Program Counter:** Address of the next instruction to be executed.
6. **Memory Management Information:** Information about memory allocated to the process.
7. **I/O Status Information:** List of input/output devices assigned to the process.
8. **Accounting Information:** Time and resources used by the process.
9. Describe the process state diagram and the transitions between different process states.

Ans : A **Process State Diagram** represents the different stages a process goes through during its execution in an operating system. The operating system manages these states to ensure smooth execution of multiple processes.



**Process Transitions:**

* **New → Ready:** When a new process is created, it moves to the ready state.
* **Ready → Running:** The CPU scheduler selects the process and executes it.
* **Running → Waiting:** The process moves to waiting if it needs input/output resources.
* **Waiting → Ready:** When the required input/output operation is completed, the process goes back to the ready queue.
* **Running → Terminated:** The process finishes execution and is removed from memory.
* **Running → Ready:** In preemptive scheduling, a running process moves back to the ready state if its time slice is over.

1. How does a process communicate with another process in an operating system?

Ans: Processes in an operating system communicate with each other through **Inter-Process Communication (IPC)**. IPC allows processes to exchange data and information while running independently. This communication is essential in multitasking systems where different processes need to work together.

**Methods of Process Communication:**

1. **Shared Memory:**  
   In this method, two or more processes share a common memory space where they can read and write data. The processes must follow synchronization rules to avoid conflicts. This method is faster but requires careful management.
2. **Message Passing:**  
   Processes communicate by sending and receiving messages through **mailboxes** or **message queues**. This method is safer than shared memory because processes do not share direct access to memory.
3. **Pipes:**  
   A pipe is a unidirectional communication method where one process writes data, and another process reads it. It is commonly used in **Linux and Unix systems**.
4. **Sockets:**  
   Sockets allow processes to communicate over a **network** or between different systems. It is used in **client-server applications**.
5. **Signals:**  
   Signals are small messages that inform a process about certain events, like **termination requests or error notifications**.
6. What is process synchronization, and why is it important?

Ans : **Process Synchronization** is a technique used in operating systems to coordinate the execution of multiple processes that share common resources. It ensures that processes execute in a **controlled and orderly manner** without interfering with each other.

**Why is Process Synchronization Important?**

In multitasking systems, several processes may need to access shared resources like files, memory, or printers at the same time. If these processes are not synchronized, it can cause:

* **Data inconsistency** (wrong or lost data)
* **Deadlocks** (processes waiting for each other forever)
* **Race conditions** (multiple processes trying to modify shared data at the same time)

1. Explain the concept of a zombie process and how it is created.

Ans : A **Zombie Process** is a process that has completed its execution but still remains in the process table of the operating system. It is also known as a **defunct process**. Even though the process is terminated, its **Process Control Block (PCB)** remains in memory to store information like the **exit status** until the parent process reads it.

**How is a Zombie Process Created?**

1. When a child process finishes execution, it sends an **exit status** to the parent process.
2. The child process enters the **terminated state**, but its PCB remains in the process table.
3. The parent process must call the **wait()** system call to read the exit status of the child process.
4. If the parent process does not read the exit status, the child process becomes a **zombie process**.
5. Describe the difference between internal fragmentation and external fragmentation.

Ans :

| **Aspect** | **Internal Fragmentation** | **External Fragmentation** |
| --- | --- | --- |
| **Definition** | Occurs when fixed-sized memory blocks are allocated, and unused space inside the block remains. | Occurs when free memory is divided into small non-contiguous blocks, making it difficult to allocate to new processes. |
| **Cause** | Fixed-sized memory allocation (larger blocks than required by the process). | Dynamic memory allocation where memory blocks are allocated and freed continuously. |
| **Memory Waste Location** | Inside allocated memory blocks (unused space within the block). | Between allocated memory blocks (unused free memory gaps). |
| **Effect on System** | Small unused memory within blocks leads to less memory utilization. | Larger memory gaps make it difficult to allocate large processes even if the total free memory is sufficient. |
| **Solution** | Use dynamic memory allocation with exact memory size or smaller block sizes. | Memory compaction or paging techniques can reduce fragmentation. |
| **Example** | Allocating 10KB memory for a process that only requires 7KB, leaving 3KB unused. | Free memory blocks of 4KB, 6KB, and 8KB scattered across the memory, but unable to allocate a 15KB process. |

1. What is demand paging, and how does it improve memory management efficiency?

Ans : **Demand paging** is a memory management technique where pages of a program are only loaded into physical memory (RAM) when they are needed, rather than loading the entire program at once. This method is commonly used in virtual memory systems.

**How It Improves Memory Management Efficiency:**

1. **Reduced Memory Usage:** Only the necessary pages are loaded into RAM, minimizing memory consumption.
2. **Faster Program Startup:** Programs start faster since only essential pages are initially loaded.
3. **Efficient Use of RAM:** More processes can be kept in memory simultaneously, improving **multiprogramming**.
4. **On-Demand Loading:** Pages are loaded only when required, which avoids loading unused portions of the program.
5. **Paging Replacement Algorithms:** Combined with algorithms like **LRU (Least Recently Used)** or **FIFO (First In, First Out)**, unused pages can be swapped out, optimizing memory usage.
6. Explain the role of the page table in virtual memory management.

Ans : The **page table** plays a crucial role in virtual memory management by acting as a bridge between virtual addresses used by programs and physical addresses in RAM. It is a data structure maintained by the operating system to translate virtual addresses into physical addresses. When a program requests data, the virtual address is divided into a **page number** and an **offset**. The page number is used to search the page table, which provides the corresponding **frame number** in physical memory. If the page is found in RAM, the system accesses the data directly. However, if the page is not in memory, a **page fault** occurs, prompting the operating system to load the required page from secondary storage into RAM. Each page table entry typically contains information such as the frame number, protection bits, valid or invalid bits, and other control flags. This structure not only facilitates address translation but also ensures **memory protection**, **efficient memory usage**, and **page sharing** between processes. The page table helps the operating system manage memory more effectively, enabling large programs to run on limited physical memory by loading only necessary pages on demand.

1. How does a memory management unit (MMU) work?

Ans : A **Memory Management Unit (MMU)** is a hardware component in a computer system that handles the translation of virtual memory addresses into physical memory addresses. The MMU plays a crucial role in modern operating systems by ensuring efficient and secure memory management. When a program runs, it uses **virtual addresses** rather than physical addresses to access data. The MMU automatically translates these virtual addresses into physical addresses before accessing memory. This process begins when the CPU generates a virtual address during data access. The MMU then checks the **page table**, a data structure that maps virtual addresses to physical memory locations, to find the corresponding physical address. If the required data is not in physical memory, a **page fault** occurs, and the operating system retrieves the data from secondary storage. The MMU also performs **memory protection** by verifying if the process has permission to access the requested memory, preventing unauthorized access. Additionally, the MMU supports **paging** and **segmentation** techniques, which optimize memory allocation and usage. By managing address translation, access rights, and memory allocation, the MMU improves system performance, ensures data security, and enables efficient use of physical memory resources.

1. What is thrashing, and how can it be avoided in virtual memory systems?

Ans : **Thrashing** is a condition in virtual memory systems where the computer spends more time swapping pages between **RAM** and **secondary storage (like hard disk or SSD)** than executing actual processes. It occurs when there are too many page faults, causing the system to continuously load and unload pages, which significantly reduces system performance.

Thrashing happens when the system's **RAM is overloaded**, and the operating system tries to keep too many processes active at the same time without enough memory space. This constant page swapping consumes CPU time, making the system extremely slow.

To avoid thrashing, several techniques can be used. One method is **working set model**, where the operating system monitors the set of pages that a process frequently uses and ensures those pages remain in memory. Another technique is **page fault frequency (PFF)**, which adjusts the number of pages allocated to a process based on the rate of page faults. Additionally, using **priority-based scheduling** allows the system to give more memory to important processes while limiting memory allocation to less critical ones. Increasing the system's **RAM capacity** or **using efficient page replacement algorithms** like **LRU (Least Recently Used)** can also help prevent thrashing. Proper memory management and ensuring that the system does not overload its resources are essential to maintaining smooth performance and avoiding thrashing.

1. What is a system call, and how does it facilitate communication between user programs and the operating system?

Ans : A **system call** is a special function used by user programs to request services from the **operating system (OS)**. It acts as a bridge between user applications and the OS, allowing programs to access hardware resources and perform essential operations like file management, process control, and communication.

When a user program needs to perform tasks such as reading from a file, creating a new process, or accessing memory, it cannot directly communicate with the hardware for security and control reasons. Instead, the program makes a system call by sending a request to the operating system through a **system call interface**. The OS receives the request, verifies its validity, executes the requested service, and then sends the result back to the user program.

System calls ensure **controlled access** to system resources while maintaining **security and stability**. They also help the OS manage resources efficiently by preventing direct hardware access by user programs. Examples of common system calls include **open()** for opening files, **read()** for reading data, **fork()** for creating processes, and **exit()** for terminating programs. By providing a standardized way for applications to interact with the operating system, system calls simplify programming and protect the system from unauthorized access.

1. Describe the difference between a monolithic kernel and a microkernel.

Ans : The **monolithic kernel** and **microkernel** are two different types of operating system architectures, each with its own way of managing system resources and services.

A **monolithic kernel** is a large, single program where the entire operating system, including core services like **process management**, **memory management**, **file system**, and **device drivers**, runs in the **kernel space**. All system services are integrated into one unit, making the system faster because there is no need for frequent communication between components. However, this architecture can be less secure and harder to maintain since a small error in one service could crash the entire system. Examples of monolithic kernels include **Linux** and **Unix**.

On the other hand, a **microkernel** follows a more modular approach by keeping only the most essential services, such as **CPU scheduling**, **inter-process communication**, and **basic memory management**, in the kernel space. Non-essential services like **device drivers**, **file systems**, and **networking** run in **user space** as separate processes. This makes the microkernel architecture more secure, flexible, and easier to maintain. However, microkernels tend to be slower because communication between the kernel and user services requires message passing. Examples of microkernel-based systems include **MINIX** and **QNX**.

In summary, a **monolithic kernel** offers better performance but lower security, while a **microkernel** provides greater security and modularity at the cost of performance.

1. How does an operating system handle I/O operations?

Ans : An **operating system (OS)** handles **Input/Output (I/O) operations** by acting as an intermediary between user programs and hardware devices like keyboards, printers, hard drives, and network devices. The OS ensures that I/O operations are performed efficiently, securely, and without directly involving user programs in hardware management.

When a user program requests an I/O operation, such as reading a file or printing a document, the request is sent to the **I/O subsystem** of the operating system. The OS uses **system calls** to communicate between the application and the hardware. The I/O subsystem includes **device drivers**, which are special software programs that translate high-level I/O requests into low-level hardware commands. Each device driver is specific to a hardware device, ensuring that the OS can interact with different devices.

The OS uses techniques like **buffering**, where data is temporarily stored in memory before being processed, to improve performance. It also employs **spooling** for devices like printers, where multiple I/O requests are queued and processed one at a time. The OS manages I/O devices through **interrupts**, where the hardware signals the CPU when an I/O operation is complete, allowing the CPU to perform other tasks while waiting.

Additionally, the OS ensures **error handling**, **device scheduling**, and **protection** to prevent unauthorized access. By managing I/O operations, the operating system provides a smooth interface between user programs and hardware devices, improving system performance and resource utilization.

1. Explain the concept of a race condition and how it can be prevented.

Ans : A **race condition** is a situation in computing where two or more processes or threads try to access shared resources at the same time, leading to unpredictable and incorrect results. It occurs when the final outcome of a program depends on the **timing or sequence** of execution of multiple processes.

For example, if two bank account withdrawal processes are running simultaneously and both check the balance before making deductions, they might withdraw money based on the same balance without updating it correctly, resulting in incorrect account balance.

Race conditions can cause serious problems such as **data corruption**, **system crashes**, or **security vulnerabilities**. They usually occur in **multitasking or multi-threading** environments where multiple tasks share resources like memory or files.

1. Describe the role of device drivers in an operating system.

Ans : **Device drivers** play a vital role in an **operating system (OS)** by enabling communication between the operating system and hardware devices such as printers, keyboards, hard drives, and USB devices. A device driver is a special software program that acts as a translator between the hardware and the operating system, allowing the OS to control and interact with hardware components.

When a user or application performs an action involving hardware, such as printing a document or playing sound, the operating system sends the request to the appropriate device driver. The driver converts these high-level commands into **low-level instructions** that the hardware device can understand. Once the device completes the task, the driver sends feedback back to the operating system.

Device drivers help manage various hardware functions like **data transfers**, **error detection**, and **device initialization**. They also handle **interrupts**, where the hardware signals the OS when it needs attention. There are two types of device drivers: **Kernel-mode drivers**, which work directly with the operating system, and **User-mode drivers**, which manage less critical devices like printers or cameras.

Without device drivers, the operating system would not be able to interact with hardware components, making them unusable. Device drivers ensure that hardware devices work properly, improve compatibility, and allow the operating system to support a wide range of devices.

1. What is a zombie process, and how does it occur? How can a zombie process be prevented?

Ans : A **zombie process** is a process that has completed its execution but still remains in the system's process table without being fully terminated. It occurs when the **child process** finishes execution, but its **parent process** does not read its **exit status** using system calls like **wait()** or **waitpid()**. The process remains in the system to allow the parent process to collect information about its termination.

When a child process ends, it sends a termination signal to the parent process. The operating system keeps minimal information about the child process in the process table, such as its **process ID (PID)** and **exit status**, until the parent reads the termination status. If the parent process does not check this status, the child process remains as a zombie process, occupying system resources.

**How to Prevent Zombie Processes:**

1. **Using wait() or waitpid():** The parent process should use these system calls to read the child's exit status and allow the system to remove the child process from the process table.
2. **Signal Handling:** The parent process can use the **SIGCHLD** signal, which notifies the parent whenever a child process terminates, automatically cleaning up the zombie process.
3. **Automatic Reaping:** Some modern operating systems automatically clean up zombie processes if the parent process terminates.

Although zombie processes do not consume CPU or memory, having too many zombies can fill up the process table, slowing down system performance. Proper process management and system calls can prevent the creation of zombie processes and ensure efficient resource utilization.

1. Explain the concept of an orphan process. How does an operating system handle orphan processes?

Ans : An **orphan process** is a process whose **parent process has terminated** while the child process is still running. This means the child process continues executing without any parent to supervise or collect its exit status.

Orphan processes typically occur when the parent process ends unexpectedly due to errors or manual termination, while the child process remains active. Since the parent process is responsible for reading the child’s **exit status** using system calls like **wait()** or **waitpid()**, the absence of the parent leaves the child without supervision.

**How the Operating System Handles Orphan Processes:**

When an orphan process is detected, the \*\*operating system automatically reassigns the orphan process to a special process called the **init process** (process with PID 1 in Unix/Linux systems). The **init process** becomes the new parent and is responsible for waiting on the orphan process, collecting its **exit status**, and freeing system resources once the orphan process terminates.

This automatic reassignment prevents orphan processes from becoming **zombie processes** and ensures that system resources are properly managed.

**Key Points:**

* Orphan processes are not harmful because the OS handles them automatically.
* They can occur due to programming errors or unexpected parent process termination.
* Proper process management and signal handling can help avoid orphan processes.

By assigning orphan processes to the **init process**, the operating system ensures that no process is left running without supervision, maintaining system stability and resource management.

1. What is the relationship between a parent process and a child process in the context of process management?

Ans : In **process management**, the relationship between a **parent process** and a **child process** represents how processes are created and managed by the operating system. A **parent process** is a process that creates one or more **child processes** using system calls like **fork()** in Unix/Linux or **CreateProcess()** in Windows.

When a parent process creates a child process, the child process is a **copy of the parent** with its own unique **Process ID (PID)**. The child process inherits many properties from the parent, such as **environment variables**, **open files**, and **priority levels**, but it runs as a separate entity with its own memory space and execution flow.

The parent process usually waits for the child to complete using system calls like **wait()** or **waitpid()**, which allow the parent to collect the child’s **exit status** and free system resources. If the parent terminates before the child, the child process becomes an **orphan process** and is reassigned to the **init process**. If the child finishes execution but the parent does not read its exit status, the child becomes a **zombie process**.

The relationship between parent and child processes helps the operating system manage **concurrent execution**, **resource sharing**, and **process hierarchy**. This structure improves system performance, supports multitasking, and ensures better resource utilization in modern operating systems.

1. How does the fork() system call work in creating a new process in Unix-like operating systems?

Ans : The **fork()** system call is used to create a **new child process** by duplicating the **parent process** in Unix-like operating systems. When **fork()** is called, the operating system creates a copy of the parent process with a unique **Process ID (PID)**. The child process inherits the parent's **memory space**, **file descriptors**, **environment variables**, and **program code**. Both processes execute the same code from the point where **fork()** was called, but the return value differs: the parent process receives the child's **PID**, while the child process receives **0**.

1. Describe how a parent process can wait for a child process to finish execution.

Ans : A **parent process** can wait for its child process to finish using the **wait()** or **waitpid()** system calls. These functions pause the parent process until the child process terminates. The **wait()** call makes the parent wait for any child process, while **waitpid()** allows the parent to wait for a specific child process by specifying its **PID**. Once the child finishes execution, the parent collects the child's **exit status**, preventing the child from becoming a **zombie process**.

1. What is the significance of the exit status of a child process in the wait() system call?

Ans : The **exit status** of a child process provides information about how the child process terminated. It helps the parent process determine whether the child finished successfully or if an error occurred. The **wait()** system call stores the exit status in a variable, which can be analyzed using macros like **WIFEXITED()** to check if the child terminated normally or **WEXITSTATUS()** to get the actual exit code.

1. How can a parent process terminate a child process in Unix-like operating systems?

Ans : A parent process can terminate a child process using the **kill()** system call, which sends a signal to the child process based on its **PID**. Signals like **SIGTERM** request the process to terminate gracefully, while **SIGKILL** forces immediate termination without allowing the process to clean up its resources. The parent process must have the appropriate permissions to send signals to the child. The **kill()** system call is often used in scenarios where the child process becomes unresponsive or exceeds its execution time. Additionally, the parent can use signal handling mechanisms to ensure that critical resources are freed before termination.

1. Explain the difference between a process group and a session in Unix-like operating systems.

Ans : In Unix-like operating systems, processes can be grouped into **process groups** and **sessions** to facilitate job control and signal management. A **process group** is a collection of related processes that can receive signals collectively. Each process group is identified by a **Process Group ID (PGID)**, typically assigned by the parent process. This grouping is useful when managing background jobs or multitasking applications. A **session**, on the other hand, is a higher-level grouping that can contain multiple process groups. Sessions are commonly associated with user logins and controlling terminals. When a user logs into the system, a new session is created, and all processes started from that terminal belong to the same session. These structures help the operating system manage process hierarchies and signals efficiently.

1. Describe how the exec() family of functions is used to replace the current process image with a new one.

Ans : The **exec()** family of functions allows a process to replace its current program image with a new one. These functions, including **execl()**, **execv()**, and **execvp()**, load a new executable into memory and begin executing it from the entry point of the new program. Unlike the **fork()** system call, which creates a separate child process, **exec()** does not create a new process but rather replaces the existing process’s code and data. If the **exec()** call succeeds, the original program’s execution never resumes. This function is commonly used in shell programs and system utilities to execute external commands while reusing the same process.

1. What is the purpose of the waitpid() system call in process management? How does it differ from wait()?

Ans : The **waitpid()** system call serves the same purpose as **wait()**, but with additional flexibility. While **wait()** makes the parent process wait for any child process to terminate, **waitpid()** allows the parent to wait for a specific child process by specifying its **PID**. It also supports options like **WNOHANG**, which makes the call non-blocking, allowing the parent to continue execution if no child process has exited. This system call is especially useful in applications where the parent manages multiple child processes simultaneously and needs to monitor their progress individually.

1. How does process termination occur in Unix-like operating systems?

Ans : Process termination in Unix-like operating systems occurs when a process finishes execution, is killed by another process, or receives a termination signal. A process can terminate voluntarily by calling the **exit()** system call or as a result of encountering a critical error. If the parent process is still running, it can use the **wait()** or **waitpid()** system call to collect the child’s **exit status** and free system resources. If the parent does not wait, the terminated process becomes a **zombie process**. The operating system ensures that all resources allocated to the process, such as memory and file descriptors, are properly released during termination.

1. What is the role of the long-term scheduler in the process scheduling hierarchy? How does it influence the degree of multiprogramming in an operating system?

Ans : The **long-term scheduler** plays a crucial role in process scheduling by selecting which processes from the **job queue** should be loaded into memory for execution. Its primary function is to regulate the degree of **multiprogramming** — the number of processes that can reside in memory simultaneously. The long-term scheduler balances system performance by deciding the right mix of **CPU-bound** and **I/O-bound** processes. If too many processes are loaded into memory, the system may become overloaded, leading to thrashing. Conversely, if too few processes are selected, the CPU may remain idle, reducing system efficiency.

1. How does the short-term scheduler differ from the long-term and medium-term schedulers in terms of frequency of execution and the scope of its decisions?

Ans : The **short-term scheduler** (CPU scheduler) differs from the **long-term scheduler** and **medium-term scheduler** in both frequency of execution and scope of decisions. The short-term scheduler runs frequently, often every few milliseconds, to select which process from the **ready queue** will execute next. Its decisions are based on scheduling algorithms like **Round Robin** or **Priority Scheduling**. In contrast, the long-term scheduler runs much less frequently, deciding which jobs to admit into the system. The **medium-term scheduler** temporarily suspends processes (swapping) to manage memory more efficiently, helping maintain an optimal level of multiprogramming.

1. Describe a scenario where the medium-term scheduler would be invoked and explain how it helps manage system resources more efficiently.

Ans : The **medium-term scheduler** is invoked when the operating system detects memory pressure or when the system needs to optimize performance. For example, if the system is running several CPU-intensive tasks along with memory-heavy applications, the medium-term scheduler may **swap out** some inactive or low-priority processes to secondary storage, freeing memory for active processes. This process, known as **swapping**, helps prevent thrashing and improves overall system performance. Once memory becomes available, the swapped-out processes can be brought back into memory and resume execution. This dynamic management of system resources allows the operating system to maintain a balance between **CPU utilization** and **memory availability**.

Part E

1. Consider the following processes with arrival times and burst times:

| Process | Arrival Time | Burst Time |

|---------|--------------|------------|

| P1 | 0 | 5 |

| P2 | 1 | 3 |

| P3 | 2 | 6 |

Calculate the average waiting time using First-Come, First-Served (FCFS) scheduling.

Ans :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** | **Completion Time (CT)** | **Turnaround Time (TAT)** | **Waiting Time (WT)** |
| P1 | 0 | 5 | 5 | 5 | 0 |
| P2 | 1 | 3 | 8 | 7 | 4 |
| P3 | 2 | 6 | 14 | 12 | 6 |

**Average Waiting Time:**

Average Waiting Time = 0+4+6​/ 3 ​=3.33 units

1. Consider the following processes with arrival times and burst times:

| Process | Arrival Time | Burst Time |

|---------|--------------|------------|

| P1 | 0 | 3 |

| P2 | 1 | 5 |

| P3 | 2 | 1 |

| P4 | 3 | 4 |

Calculate the average turnaround time using Shortest Job First (SJF) scheduling.

Ans :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Process** | **Arrival Time** | **Burst Time** | **Completion Time (CT)** | **Turnaround Time (TAT)** | | P1 | 0 | 3 | 3 | 3 | | P3 | 2 | 1 | 4 | 2 | | P4 | 3 | 4 | 8 | 5 | | P2 | 1 | 5 | 13 | 12 | |

**Average Turnaround Time Calculation:**

Average Turnaround Time =3+2+5+12 % 4

= 5.5 units

1. Consider the following processes with arrival times, burst times, and priorities (lower number indicates higher priority):

| Process | Arrival Time | Burst Time | Priority |

|---------|--------------|------------|----------|

| P1 | 0 | 6 | 3 |

| P2 | 1 | 4 | 1 |

| P3 | 2 | 7 | 4 |

| P4 | 3 | 2 | 2 |

Calculate the average waiting time using Priority Scheduling.

Ans :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** | **Priority** | **Completion Time (CT)** | **Turnaround Time (TAT)** | **Waiting Time (WT)** |
| P1 | 0 | 6 | 3 | 13 | 13 | 7 |
| P2 | 1 | 4 | 1 | 5 | 4 | 0 |
| P3 | 2 | 7 | 4 | 20 | 18 | 11 |
| P4 | 3 | 2 | 2 | 7 | 4 | 2 |

**Average Waiting Time Calculation:**

AverageWaitingTime = 7+0+11+2% 4 = 5 units

1. Consider the following processes with arrival times and burst times, and the time quantum for Round Robin scheduling is 2 units:

| Process | Arrival Time | Burst Time |

|---------|--------------|------------|

| P1 | 0 | 4 |

| P2 | 1 | 5 |

| P3 | 2 | 2 |

| P4 | 3 | 3 |

Calculate the average turnaround time using Round Robin scheduling.

Ans :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Process** | **Arrival Time** | **Burst Time** | **Time Quantum** | **Completion Time (CT)** | **Turnaround Time (TAT)** |
| P1 | 0 | 4 | 2 | 10 | 10 |
| P2 | 1 | 5 | 2 | 15 | 14 |
| P3 | 2 | 2 | 2 | 6 | 4 |
| P4 | 3 | 3 | 2 | 13 | 10 |

Average Turnaround Time = 10+14+4+10​ % 4 = 9.5 units.

Que 5. Consider a program that uses the fork() system call to create a child process. Initially, the parent process has a variable x with a value of 5. After forking, both the parent and child processes increment the value of x by 1.

What will be the final values of x in the parent and child processes after the fork() call?

Ans : In the **fork()** system call, the operating system creates a **child process** that is a duplicate of the **parent process**. Both parent and child processes will have **separate copies** of the variable **x** in their own memory space.

**Explanation:**

* Initially, the parent process has a variable **x = 5**.
* When the **fork()** system call is executed, the child process is created as an exact copy of the parent process, including the variable **x = 5**.
* After the fork, both processes continue execution from the next instruction **independently**.
* Each process increments its own copy of **x** by **1**.

**Final Values:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Initial Value of x** | **After Increment** | **Final Value** |
| Parent Process | 5 | +1 | 6 |
| Child Process | 5 | +1 | 6 |

Submission Guidelines:

* Document each step of your solution and any challenges faced.
* Upload it on your GitHub repository

Additional Tips:

* Experiment with different options and parameters of each command to explore their functionalities.
* This assignment is tailored to align with interview expectations, CCEE standards, and industry demands.
* If you complete this then your preparation will be skyrocketed.