**CDAC MUMBAI**

**Concepts of Operating System**

**Assignment 2**

**Part A**

**What will the following commands do?**

• **echo "Hello, World!"** :: The command echo "Hello, World!" prints "Hello, World!" to the terminal.

• **name="Productive" ::**

The command name="Productive" assigns the value "Productive" to the variable name.

• **touch file.txt ::** The command touch file.txt creates an empty file named file.txt or updates its timestamp if it already exists.

**• ls -a** :: The command ls -a lists all files and directories, including hidden files (those starting with .)

• **rm file.txt ::** The command rm file.txt deletes the file file.txt permanently

• **cp file1.txt file2.txt ::** The command cp file1.txt file2.txt copies the contents of file1.txt into file2.txt. If file2.txt doesn’t exist, it creates a new one.

• **mv file.txt /path/to/directory/ ::** The command mv file.txt /path/to/directory/ moves file.txt to the specified directory. If the directory path is incorrect, it will show an error.

• **chmod 755 script.sh ::**

The command chmod 755 script.sh sets permissions for script.sh:

7 (rwx) → Owner can read, write, and execute.

5 (r-x) → Group and others can read and execute but not write.

• **grep "pattern" file.txt ::** The command grep "pattern" file.txt searches for the word "pattern" inside file.txt and displays matching lines.

• **kill PID ::** The command kill PID terminates the process with the specified Process ID (PID)

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

1. mkdir mydir → Creates a directory named mydir.

2. cd mydir → Moves into the mydir directory.

3. touch file.txt → Creates an empty file named file.txt.

4. echo "Hello, World!" > file.txt → Writes "Hello, World!" into file.txt.

5. cat file.txt → Displays the contents of file.txt.

**• ls -l | grep ".txt" ::** The command ls -l | grep ".txt" lists all .txt files in the current directory with detailed information.

• **cat file1.txt file2.txt | sort | uniq ::**

1. cat file1.txt file2.txt → Displays the contents of both files.

2. sort → Sorts the combined output alphabetically.

3. uniq → Removes duplicate lines, showing only unique ones.

Result: A sorted list of unique lines from file1.txt and file2.txt.

**• ls -l | grep "^d" ::**

The command ls -l | grep "^d" lists only directories in the current directory.

1.ls -l → Lists files and directories with details.

2.grep "^d" → Filters lines that start with "d", which indicates a directory in ls -l output.

**• grep -r "pattern" /path/to/directory/ ::**

The command grep -r "pattern" /path/to/directory/ searches recursively for "pattern" in all files inside /path/to/directory/.

1. -r → Recursively searches in subdirectories.

2. "pattern" → The text to search for.

3. /path/to/directory/ → The directory where the search is performed.

• **cat file1.txt file2.txt | sort | uniq –d ::**

The command cat file1.txt file2.txt | sort | uniq -d finds duplicate lines that appear in both file1.txt and file2.txt.

1. cat file1.txt file2.txt : Displays contents of both files.

2. sort : Sorts the combined output alphabetically.

3. uniq -d : Shows only duplicate lines (lines that appear more than once)

• **chmod 644 file.txt ::** The command chmod 644 file.txt sets the permissions for file.txt:

6 (rw-) → Owner can read and write.

4 (r--) → Group and others can only read.

Makes the file readable by everyone but only editable by the owner

**• cp -r source\_directory destination\_directory ::**

The command cp -r source\_directory destination\_directory recursively copies source\_directory and all its contents (files and subdirectories) to destination\_directory.

cp → Copies files and directories.

-r → Recursively copies all subdirectories and files.

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

The command find /path/to/search -name "\*.txt" searches for all .txt files inside /path/to/search and its subdirectories.

1. find → Searches for files and directories.

2. /path/to/search → The directory where the search starts.

3. -name "\*.txt" → Looks for files with a .txt extension.

**• chmod u+x file.txt ::**

The command chmod u+x file.txt adds execute (x) permission for the file owner (u).

1. The owner can now run file.txt as an executable.

**• echo $PATH ::**

The command echo $PATH displays the system's executable search path, which is a list of directories where the shell looks for executable files.

**Part B**

**Identify True or False:**

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

2. mv is used to move files and directories. :: **True**

3. cd is used to copy files and directories. :: **False**

4. pwd stands for "print working directory" and displays the current directory. :: **True**

5. grep is used to search for patterns in files. :: **True**

6. chmod 755 file.txt gives read, write, and execute permissions to the owner, and read and execute

permissions to group and others. :: **True**

7. mkdir -p directory1/directory2 creates nested directories, creating directory2 inside directory1

if directory1 does not exist. :: **True**

8. rm -rf file.txt deletes a file forcefully without confirmation. :: **True**

**Identify the Incorrect Commands:**

1. chmodx is used to change file permissions. :: **Incorrect** // Correct - chmod

2. cpy is used to copy files and directories. :: **Incorrect** //correct- cp

3. mkfile is used to create a new file. :: **Incorrect**

// correct - touch filename to create an empty file.

- echo "text" > filename to create a file with content.

-nano filename or vim filename to create and edit a file.

4. catx is used to concatenate files. :: **Incorrect** // Correct- cat

5. rn is used to rename files. :: **Incorrect** //correct- mv oldname newname

**Part C**

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

nano hello.sh

echo "Hello, World!"

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

**value of the variable.**

nano variable.sh

name="CDAC Mumbai"

echo "The value of name is: $name"

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

nano input\_number.sh

echo "Enter a number:"

read number

echo "You entered: $number"

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

nano addition.sh

num1=5

num2=3

sum=$((num1 + num2))

echo "The sum of $num1 and $num2 is: $sum"

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

nano even\_odd.sh

echo "Enter a number:"

read num

if ((num % 2 == 0)); then

echo "Even"

else

echo "Odd"

fi

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

nano loop.sh

for i in {1..5}

do

echo $i

done

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

nano while\_loop.sh

i=1

while [ $i -le 5 ]

do

echo $i

((i++))

done

**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".**

nano check\_file.sh

if [ -f "file.txt" ]; then

echo "File exists"

else

echo "File does not exist"

fi

**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.**

nano check\_number.sh

echo "Enter a number:"

read num

if [ $num -gt 10 ]; then

echo "The number is greater than 10."

else

echo "The number is 10 or less."

fi

**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.**

nano multiplication\_table.sh

echo "Multiplication Table (1 to 5)"

for i in {1..5}

do

for j in {1..5}

do

printf "%4d" $((i \* j))

done

echo # Move to the next line

done

**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.**

nano square\_numbers.sh

while true

do

echo "Enter a number (enter a negative number to exit):"

read num

if [ $num -lt 0 ]; then

echo "Negative number entered. Exiting..."

break

fi

square=$((num \* num))

echo "Square of $num is: $square"

done

**Part D**

**Common Interview Questions (Must know)**

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

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

Primary Functions of an Operating System:

Process Management – Handles process creation, scheduling, and termination.

Memory Management – Allocates and deallocates memory to programs.

File System Management – Manages files and directories on storage devices.

Device Management – Controls hardware devices using drivers.

Security & Access Control – Provides user authentication and data protection.

User Interface – Offers CLI (Command Line Interface) or GUI (Graphical User Interface).

Networking – Manages network connections and communication.

Error Detection & Handling

**2. Explain the difference between process and thread.**

1. Definition

A process is an independent execution unit with its own memory space.

A thread is a lightweight execution unit within a process, sharing the process’s memory.

2. Memory

Each process has its own address space (isolated memory).

Threads within the same process share memory.

3. Communication

Inter-process communication (IPC) is needed to share data between processes.

Threads can directly communicate by accessing shared memory.

4. Creation Time

Creating a new process is slower due to memory allocation and resource management.

Creating a new thread is faster as it shares the process’s resources.

5. Execution

Processes run independently and do not share resources.

Multiple threads within a process can run concurrently, sharing CPU time.

6. Example

Running a browser and a text editor as separate processes.

Multiple tabs in a browser running as threads within the browser process.

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

Virtual Memory :

Virtual memory is a memory management technique that allows a computer to compensate for physical RAM limitations by using disk space as temporary memory.

It creates an illusion of a larger memory by mapping logical addresses to physical addresses.

How Virtual Memory Works:

Paging – The OS divides memory into fixed-sized blocks called pages and swaps them between RAM and disk as needed.

Segmentation – Memory is divided into variable-sized segments based on logical divisions like functions or modules.

Swap Space – When RAM is full, inactive processes are moved to disk (swap space) and brought back when needed.

Demand Paging – Only required pages are loaded into RAM, reducing memory usage.

Page Replacement Algorithms – If RAM is full, the OS decides which pages to swap out using algorithms like LRU (Least Recently Used) and FIFO (First In, First Out).

**4. Describe the difference between multiprogramming, multitasking, and multiprocessing.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Multiprogramming** | **Multitasking** | **Multiprocessing** |
| **Definition** | Running multiple programs in memory at the same time, but only one executes at a time. | Running multiple tasks (programs/processes) simultaneously by switching between them quickly. | Using multiple processors (CPUs) to execute multiple processes simultaneously. |
| **Execution** | One program runs at a time, others wait. | CPU rapidly switches between tasks, giving the illusion of parallel execution. | Multiple processes run in parallel on different CPUs. |
| **Resource Use** | Uses a single CPU, switching between processes. | Uses a single CPU but efficiently switches between processes. | Uses multiple CPUs, each handling different processes. |
| **Example** | Running a compiler while a text editor is open. | Browsing the internet while listening to music. | High-performance computing with multi-core processors. |

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

A file system is a method used by an operating system to store, organize, retrieve, and manage data on storage devices like hard drives, SSDs, and USBs.

It defines how files and directories are structured and accessed.

Components of a File System:

1.Files – The basic unit of storage that contains data (text, programs, images, etc.).

2.Directories (Folders) – Containers that hold files and other directories to organize data.

3.Inodes (Index Nodes) – Data structures that store metadata (file size, permissions, timestamps, location on disk) for each file.

4.Superblock – Contains information about the file system, such as size, number of files, and free space.

5.Boot Block – Stores bootloader code to initialize the operating system.

6.File Allocation Table (FAT) or Index – Keeps track of file locations on the disk.

7.Journaling (Optional) – Maintains logs to prevent data corruption in case of crashes.

**6. What is a deadlock, and how can it be prevented?**

A deadlock is a situation in which two or more processes are unable to proceed because each is waiting for a resource that another process is holding.

This results in an infinite waiting state where no process can continue execution.

Conditions for Deadlock:

A deadlock can occur if all four of these conditions hold simultaneously:

1. Mutual Exclusion – Only one process can use a resource at a time.

2. Hold and Wait – A process holding at least one resource is waiting for additional resources held by other processes.

3. No Preemption – A resource cannot be forcibly taken from a process; it must be released voluntarily.

4. Circular Wait – A circular chain of processes exists where each process is waiting for a resource held by the next process in the chain.

**7. Explain the difference between a kernel and a shell.**

1. Definition

The core part of an operating system that directly interacts with hardware and manages system resources.

A command-line or graphical interface that allows users to communicate with the OS.

2. Function

Handles process management, memory management, device management, and system calls.

Translates user commands into system calls that the kernel can execute.

3. Interaction

Interacts directly with hardware and system resources.

Acts as an interface between the user and the kernel.

4. Types

Monolithic Kernel, Microkernel, Hybrid Kernel.

Command-line shells (Bash, Zsh) and GUI shells (GNOME, Windows Explorer).

5. Example

Linux Kernel, Windows NT Kernel.

Bash (Linux), Command Prompt (Windows), PowerShell.

**8. What is CPU scheduling, and why is it important?**

CPU scheduling is the process of selecting which process from the ready queue will be executed by the CPU next.

It ensures efficient CPU utilization and minimizes waiting time by managing multiple processes in a multitasking system.

Why is CPU Scheduling Important?

1. Maximizes CPU Utilization – Keeps the CPU busy by assigning processes efficiently.

2. Reduces Waiting Time – Ensures fair execution of processes to minimize delays.

3. Improves Response Time – Enhances user experience in interactive systems.

4. Increases Throughput – Maximizes the number of processes executed in a given time.

**9. How does a system call work?**

A system call is a request made by a program to the operating system (OS) to perform a privileged operation, such as file handling, process management, or hardware communication. Since user programs cannot directly access system resources, they use system calls to interact with the OS.

**10. What is the purpose of device drivers in an operating system?**

A device driver is a specialized software component that allows the operating system (OS) to communicate with hardware devices such as printers, keyboards, hard drives, and network adapters.

Purpose of Device Drivers:

1. Hardware Abstraction – Provides a standard interface between the OS and hardware, making device communication independent of specific hardware details.

2. Facilitates Communication – Translates OS instructions into commands that hardware devices understand.

3. Ensures Compatibility – Allows different operating systems to work with various hardware devices without modification.

4. Enhances Performance – Optimized drivers ensure efficient and stable device operations.

5. Handles Device-Specific Operations – Manages low-level hardware control, such as sending signals to a printer or reading disk sectors.

**11. Explain the role of the page table in virtual memory management.**

A page table is a data structure used by the operating system (OS) to map virtual addresses to physical addresses in a system that supports virtual memory.

It helps the CPU translate the virtual memory addresses used by programs into actual locations in RAM.

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

Thrashing occurs when a system spends more time swapping pages in and out of memory (paging) rather than executing actual processes.

This happens when there is insufficient RAM, causing frequent page faults and excessive disk I/O, leading to a severe drop in performance.

How to Avoid Thrashing

1. Use Working Set Model – Ensure each process has enough pages in memory for efficient execution.

2. Adjust Degree of Multiprogramming – Reduce the number of active processes when thrashing is detected.

3. Increase RAM – More physical memory reduces excessive paging.

4. Use Page Replacement Algorithms Efficiently – Implement Least Recently Used (LRU) or Optimal Page Replacement to minimize unnecessary swaps.

5. Monitor System Performance – Detect and control high paging activity using OS tools (e.g., vmstat in Linux).

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

A semaphore is a synchronization mechanism used in operating systems to manage concurrent processes and prevent race conditions. It is an integer variable that is used to control access to shared resources in a multi-process or multi-threaded environment.

Use of Semaphores in Synchronization:

1. Mutual Exclusion – Prevents multiple processes from accessing a shared resource simultaneously (e.g., file access).

2. Process Synchronization – Ensures that processes execute in the correct order (e.g., producer-consumer problem).

3. Avoids Race Conditions – Prevents data inconsistency due to simultaneous access by multiple processes.

4. Resource Allocation – Controls access to limited resources (e.g., database connections, printers).

**14. How does an operating system handle process synchronization?**

Process synchronization ensures that multiple processes or threads can execute safely without conflicts when accessing shared resources.

The operating system (OS) provides various mechanisms to prevent race conditions, maintain data consistency, and ensure correct execution order.

**15. What is the purpose of an interrupt in operating systems?**

An interrupt is a signal sent to the CPU to indicate that an event requires immediate attention. It allows the operating system to respond to hardware or software events efficiently and quickly, without continuously checking for them (polling).

**16. Explain the concept of a file descriptor.**

A file descriptor (FD) is a unique integer identifier assigned by the operating system to represent an open file, socket, or other I/O resource. It acts as a reference for processes to perform read, write, and close operations on files and devices.

**17. How does a system recover from a system crash?**

A system crash occurs when the operating system or a critical process fails unexpectedly, leading to system instability or shutdown. Recovery mechanisms help restore the system to a stable state.

**18. Describe the difference between a monolithic kernel and a microkernel.**

The kernel is the core of an operating system that manages hardware and system processes. The two main types of kernels are Monolithic Kernel and Microkernel.

1. Structure

Monolithic Kernel : A single large program handling all OS services.

Microkernel : Minimal core kernel; most services run in user space.

2. Performance

Faster (direct communication between OS components).

Slower (inter-process communication (IPC) needed).

3. Stability

Less stable; a crash in one component can crash the entire system.

More stable; failures in one service don’t affect the entire OS.

4. Security

Less secure (all components share the same memory space).

More secure (services run in isolated user space).

5. Ease of Development

Harder to modify (changes require recompiling the whole kernel).

Easier to modify (individual services can be updated independently).

6. Examples

Linux, Windows (older versions), Unix

Minix, QNX, Windows NT, macOS X

**19. What is the difference between internal and external fragmentation?**

Fragmentation occurs when memory is allocated inefficiently, leading to wasted space. It is classified into Internal and External fragmentation.

1. Definition

Internal Fragmentation : Wasted space inside allocated memory blocks.

External fragmentation : Free memory is available but fragmented into small, unusable chunks.

2. Cause

Fixed-sized memory allocation (e.g., block size > required size).

Dynamic memory allocation where free blocks are scattered.

3. Example

Allocating 8KB for a 6KB process wastes 2KB.

A 10MB process cannot be allocated if only 10MB exists in fragments (not contiguous).

4. Solution

Use dynamic partitioning or paging.

Use compaction, paging, or segmentation to consolidate free space.

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

The Operating System (OS) acts as an interface between user programs and hardware devices to manage Input/Output (I/O) operations efficiently.

I/O System Components

The OS uses the following components to manage I/O:

I/O Devices – Includes keyboards, monitors, hard drives, network cards, etc.

Device Drivers – Software that communicates between the OS and hardware.

I/O Controller – Hardware that processes I/O requests.

System Calls – Allow user programs to request I/O services from the OS.

**21. Explain the difference between preemptive and non-preemptive scheduling.**

CPU scheduling determines which process gets CPU time. It is classified into Preemptive and Non-Preemptive scheduling.

1.Definition

Preemptive Scheduling : A process can be interrupted and moved back to the ready queue before it completes execution.

Non-Preemptive Scheduling : A process runs until completion or voluntarily releases the CPU.

2. Control

The OS controls process execution and can stop a running process.

The process itself decides when to release the CPU.

3. Response Time

Faster response time for high-priority tasks.

Slower response time as processes run to completion.

4. Efficiency

More efficient for time-sharing systems.

Simpler but less efficient for real-time or batch processing.

5. Examples

Round Robin (RR), Shortest Remaining Time First (SRTF), Priority Scheduling.

First-Come-First-Serve (FCFS), Shortest Job First (SJF), Priority (Non-Preemptive).

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

Round-Robin (RR) Scheduling is a preemptive CPU scheduling algorithm used in time-sharing systems. It ensures fair execution by assigning a fixed time slice (quantum) to each process in a circular order.

How It Works

1. Processes are placed in a queue in the order they arrive.

2. Each process gets a fixed time slice (quantum) to execute.

3. If a process completes within its time slice, it exits.

4. If not, the process is preempted and moved to the end of the queue.

5. The next process in the queue gets CPU time, and the cycle repeats.

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

Priority Scheduling is a CPU scheduling algorithm where each process is assigned a priority. The CPU selects the process with the highest priority for execution first.

How It Works

1. Each process is assigned a priority (lower number = higher priority, or vice versa).

2. The CPU schedules the process with the highest priority first.

3. If two processes have the same priority, they follow First-Come-First-Serve (FCFS).

4. Can be either Preemptive or Non-Preemptive:

Preemptive: A higher-priority process interrupts a lower-priority process.

Non-Preemptive: The CPU waits until the current process finishes before switching.

Priority Assignment

Static Priority: Assigned at process creation (e.g., system-defined priority for tasks like I/O vs. CPU jobs).

Dynamic Priority: Adjusted during execution (e.g., increasing priority for aging processes to prevent starvation).

**24. What is the shortest job next (SJN) scheduling algorithm, and when is it used?**

Shortest Job Next (SJN), also known as Shortest Job First (SJF), is a non-preemptive CPU scheduling algorithm where the process with the shortest burst time is executed first.

When is SJN Used?

Batch Processing Systems (where execution time is known in advance).

CPU-bound processes (to minimize turnaround time).

When burst times are predictable.

**25. Explain the concept of multilevel queue scheduling.**

Multilevel Queue Scheduling is a CPU scheduling algorithm that divides processes into different priority-based queues, where each queue has its own scheduling policy.

It is used when processes can be classified into distinct categories (e.g., system processes, interactive processes, batch jobs).

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

A Process Control Block (PCB) is a data structure maintained by the operating system (OS) for each process. It stores important information about a process, allowing the OS to manage and control process execution efficiently.

Information Stored in a PCB

Category Details

Process ID (PID) A unique identifier for the process.

Process State Current state (New, Ready, Running, Waiting, Terminated).

Program Counter (PC) Stores the address of the next instruction to be executed.

CPU Registers Includes general-purpose registers, stack pointers, etc.

Scheduling Information Process priority, scheduling algorithm data, etc.

Memory Management Info Page tables, base and limit registers, etc.

I/O Status Information Open files, I/O devices assigned to the process.

Accounting Information CPU time used, process start time, etc.

Parent/Child Process Info Links to parent and child processes.

**27. Describe the process state diagram and the transitions between different process states.**

A process goes through different states from creation to termination. The Process State Diagram represents these states and transitions.

Process States

1. New : Process is created but not yet ready to run.

2. Ready : Process is waiting in the queue for CPU allocation.

3. Running : Process is currently executing on the CPU.

4. Waiting : Process is waiting for an event (e.g., I/O completion).

5. Terminated : Process has completed execution or was forcefully stopped.

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

Inter-Process Communication (IPC) allows processes to exchange data and synchronize execution. IPC is essential in multi-processing systems where processes need to collaborate.

**29. What is process synchronization, and why is it important?**

Process synchronization ensures that multiple processes can execute concurrently without interfering with each other, especially when they share resources (e.g., memory, files, or variables).

Why is Process Synchronization Important?

1. Prevents Race Conditions – Ensures processes do not overwrite each other's data.

2. Avoids Deadlocks – Prevents situations where processes are waiting indefinitely.

3. Ensures Data Consistency – Protects shared resources from corruption.

4. Improves System Efficiency – Allows better CPU and resource utilization.

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

A zombie process is a process that has completed execution but still has an entry in the process table because its parent has not read its exit status.

How is a Zombie Process Created?

A child process executes and terminates (using exit() function).

The parent process does not call wait() to retrieve the child’s exit status.

The child process remains in the process table as a "zombie" (defunct process)

**31. Describe the difference between internal fragmentation and external fragmentation.**

Fragmentation occurs when memory is allocated inefficiently, leading to wasted space. It is classified into Internal and External fragmentation.

1. Definition

Internal Fragmentation : Wasted space inside allocated memory blocks.

External fragmentation : Free memory is available but fragmented into small, unusable chunks.

2. Cause

Fixed-sized memory allocation (e.g., block size > required size).

Dynamic memory allocation where free blocks are scattered.

3. Example

Allocating 8KB for a 6KB process wastes 2KB.

A 10MB process cannot be allocated if only 10MB exists in fragments (not contiguous).

4. Solution

Use dynamic partitioning or paging.

Use compaction, paging, or segmentation to consolidate free space.

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

Demand paging is a memory management technique where pages are loaded into RAM only when needed, rather than preloading the entire process into memory. This improves efficiency by reducing unnecessary memory usage.

**33. Explain the role of the page table in virtual memory management.**

The page table is a key data structure used by the operating system to manage virtual memory. It acts as a mapping mechanism between virtual addresses (used by processes) and physical addresses (used by RAM).

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

The Memory Management Unit (MMU) is a hardware component in the CPU responsible for translating virtual addresses (used by programs) into physical addresses (used by RAM). It ensures efficient memory allocation, protection, and security in modern operating systems.

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

Thrashing occurs when a computer spends more time swapping pages between RAM and disk than executing actual processes. This leads to severe performance degradation, as the CPU is constantly waiting for memory pages to load.

How to Avoid Thrashing?

Working Set Model → Allocate enough memory pages per process based on its working set (actively used pages).

Page Fault Frequency (PFF) Control → Monitor page fault rates and adjust memory allocation accordingly.

Increase RAM → More physical memory reduces reliance on swap space.

Use Better Page Replacement Algorithms → Algorithms like LRU (Least Recently Used) help retain frequently used pages.

Reduce Multiprogramming Level → Limit the number of active processes to avoid excessive memory contention.

Disk Optimization (Faster SSDs) → Using SSDs instead of HDDs improves swap performance.

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

A system call is a mechanism that allows user programs to request services from the operating system (OS) kernel. It acts as a bridge between user space and kernel space, enabling programs to perform privileged operations like file handling, process control, memory management, and networking.

**37. Describe the difference between a monolithic kernel and a microkernel.**

The kernel is the core of an operating system that manages hardware and system processes. The two main types of kernels are Monolithic Kernel and Microkernel.

1. Structure

Monolithic Kernel : A single large program handling all OS services.

Microkernel : Minimal core kernel; most services run in user space.

2. Performance

Faster (direct communication between OS components).

Slower (inter-process communication (IPC) needed).

3. Stability

Less stable; a crash in one component can crash the entire system.

More stable; failures in one service don’t affect the entire OS.

4. Security

Less secure (all components share the same memory space).

More secure (services run in isolated user space).

5. Ease of Development

Harder to modify (changes require recompiling the whole kernel).

Easier to modify (individual services can be updated independently).

6. Examples

Linux, Windows (older versions), Unix

Minix, QNX, Windows NT, macOS X

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

The Operating System (OS) acts as an interface between user programs and hardware devices to manage Input/Output (I/O) operations efficiently.

I/O System Components

The OS uses the following components to manage I/O:

I/O Devices – Includes keyboards, monitors, hard drives, network cards, etc.

Device Drivers – Software that communicates between the OS and hardware.

I/O Controller – Hardware that processes I/O requests.

System Calls – Allow user programs to request I/O services from the OS.

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

A race condition occurs when multiple processes or threads access a shared resource concurrently, and the final result depends on the order of execution. This can lead to unexpected behavior, data corruption, or security vulnerabilities.

How to Prevent Race Conditions?

1. Use Mutual Exclusion (Locks)

2. Use Semaphores

3. Implement Atomic Operations

4. Use Condition Variables

5. Apply Proper Synchronization Mechanisms

**40. Describe the role of device drivers in an operating system.**

A device driver is a specialized software component that allows the operating system (OS) to communicate with hardware devices like printers, keyboards, graphics cards, and storage devices.

Roles of Device Drivers

1. Acts as an Interface Between OS and Hardware

2. Facilitates Hardware Communication

3. Enables Plug-and-Play Functionality

4. Handles Interrupts and I/O Operations

5. Provides Hardware Abstraction

6. Ensures Compatibility and Updates

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

A zombie process is a process that has completed execution but still has an entry in the process table because its parent process has not read its exit status. It is also known as a defunct process.

How Does a Zombie Process Occur?

1. A child process is created using fork().

2.The child process finishes execution and moves to the terminated state.

3.The parent process is supposed to read the child’s exit status using wait() or waitpid().

4.If the parent fails to do so, the child process remains in the process table as a zombie

**42. Explain the concept of an orphan process. How does an operating system handle orphan**

**processes?**

An orphan process is a child process whose parent process has terminated, leaving it running in the system without supervision.

How Does the Operating System Handle Orphan Processes?

1. Reassigning Orphan Processes to init (PID 1)

2. Using a Daemon Process

3. Proper Process Management

**43. What is the relationship between a parent process and a child process in the context of process**

**management?**

What is a Parent Process?

A parent process is a process that creates another process (child process) using the fork() system call in UNIX-based operating systems.

What is a Child Process?

A child process is a process that is created by a parent process. It inherits some attributes from the parent, such as environment variables, file descriptors, and priority.

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

The fork() system call is used in Unix-like operating systems to create a new child process by duplicating the calling (parent) process.

How fork() Works?

When a process calls fork(), it:

Creates a new child process that runs independently.

Duplicates the parent process, including variables, open files, and execution state.

Returns two different values:

- To the parent: Returns the PID (Process ID) of the child.

- To the child: Returns 0

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

A parent process can wait for its child process to finish execution using the wait() or waitpid() system calls in Unix-like operating systems.

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

When a child process terminates, it returns an exit status to the parent process, which can be retrieved using the wait() or waitpid() system calls. The exit status provides important information about how the child process ended.

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

A parent process can terminate a child process using the kill() system call by sending a termination signal.

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

Process Group

A process group is a collection of processes that are logically related and share a process group ID (PGID). It helps in managing multiple processes as a unit.

All processes in a process group share the same PGID.

The first process in a group is usually the leader.

Signals (e.g., SIGTERM, SIGKILL) can be sent to an entire process group.

Useful for handling foreground and background jobs in a shell.

Session

A session is a higher-level concept that consists of one or more process groups and is identified by a session ID (SID).

A session is created using the setsid() system call.

A session has one controlling terminal.

A process in a session is either a foreground or background process.

Sessions help manage interactive programs and login sessions.

**49. Describe how the exec() family of functions is used to replace the current process image with a**

**new one.**

The exec() family of functions is used to replace the current process image with a new executable program, without changing the process ID (PID). This means that the new program takes over the calling process, and the original program does not continue executing after a successful exec() call.

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

waitpid() System Call in Process Management

The waitpid() system call in Unix/Linux is used by a parent process to wait for a specific child process to terminate and collect its exit status. It provides more control compared to wait().

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

Process Termination in Unix-Like Operating Systems

Process termination in Unix-like operating systems occurs when a process completes its execution or is forcefully stopped. A process can terminate in different ways, either voluntarily or involuntarily.

Ways a Process Can Terminate

1. Normal Termination (Voluntary)

2. Abnormal Termination

3. Forced Termination

4. Process Termination by Parent (wait() and waitpid())

**52. 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?**

Role of the Long-Term Scheduler in Process Scheduling Hierarchy

The long-term scheduler (also called the job scheduler) is responsible for selecting processes from the job queue and loading them into memory for execution. It plays a crucial role in controlling the degree of multiprogramming, which determines how many processes are in the system at a given time.

Influence on Degree of Multiprogramming

The degree of multiprogramming refers to the number of processes in memory at the same time. The long-term scheduler directly influences this by:

-Increasing Multiprogramming when CPU is underutilized (adding more jobs).

-Decreasing Multiprogramming when CPU is overloaded (limiting job entry).

**53. 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?**

Short-Term Scheduler (CPU Scheduler)

- Runs very frequently (milliseconds).

- Decides which process gets CPU time next.

- Executes whenever a process needs CPU (e.g., after I/O completion or context switch).

Long-Term Scheduler (Job Scheduler)

- Runs infrequently (seconds to minutes).

- Decides which processes enter the system for execution.

- Controls the degree of multiprogramming.

Medium-Term Scheduler (Swapper)

- Runs occasionally (depends on system load).

- Suspends and resumes processes to free memory.

- Handles process swapping (removing/restoring processes from RAM).

**54. Describe a scenario where the medium-term scheduler would be invoked and explain how it helps**

**manage system resources more efficiently.**

**| 0**

**| P2**

**| P3**

**| 1**

**| 2**

**| 5**

**| 3**

Consider a system with limited RAM where multiple processes are competing for memory. Suppose the system is running P1, P2, and P3, but P1 is waiting for an I/O operation to complete. This waiting process holds memory but is not using CPU time, leading to inefficient resource utilization.

How the Medium-Term Scheduler Helps

The medium-term scheduler suspends P1 (swaps it out to disk), freeing up RAM for other processes. Once P1's I/O operation completes, it is swapped back into RAM and resumes execution.

Example of Process Execution Timeline:

Time Process in Execution

0s P2 starts running

1s P3 is ready

2s P1 starts an I/O operation (Blocked)

3s Medium-Term Scheduler Suspends P1 (Swapped to Disk)

5s P2 and P3 continue execution with freed memory

Later P1’s I/O completes → Scheduler swaps P1 back

**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.**

Step 1: Calculate Completion Time (CT)

FCFS executes processes in the order of arrival.

P1 starts at 0 → Finishes at 0 + 5 = 5

P2 starts at 5 → Finishes at 5 + 3 = 8

P3 starts at 8 → Finishes at 8 + 6 = 14

Step 2: Calculate Turnaround Time (TAT)

TAT=CT−ArrivalTime

p1 : 5 - 0 = 5

p2 : 8-1 = 7

p3 : 14-2 = 12

Step 3: Calculate Waiting Time (WT)

WT=TAT−BurstTime

p1 : 5 - 5 = 0

p2 : 7-3 = 4

p3 : 12-6 = 6

Step 4: Calculate Average Waiting Time (AWT)

AWT = ∑WT / Number of Process

AWT = 0+4+6/3

= 10/3

= 3.33ms

**2. 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.**

Step 1: Execution Order (SJF - Non-Preemptive)

SJF selects the process with the shortest Burst Time first.

If multiple processes are available, the one with the shortest Burst Time is chosen first.

Process Execution Order:

P1 arrives at 0 → Starts execution.

P3 arrives at 2, but P1 is still executing.

P1 finishes at 3, and now P3 (Shortest BT = 1) executes.

P3 finishes at 4, and now P4 (BT = 4) executes.

P4 finishes at 8, and now P2 (BT = 5) executes.

P2 finishes at 13.

Average Turnaround Time (ATAT) = 5.5 ms

**3. 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.**

Average Waiting Time (AWT) = 5.5 ms

**4. 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.**

Average Turnaround Time (ATAT) = 8.75 ms

**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?**

Parent Process: x = 5

fork() is called → A new child process is created with its own copy of x = 5

Both Parent and Child increment x by 1

Parent Process: x = 6

Child Process: x = 6 (modifies its own copy of x)

Final Values of x

In Parent Process: x = 6

In Child Process: x = 6

Both processes have independent copies of x

**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.**