CDAC MUMBAI

Concepts of Operating System

Assignment 2

Part A

What will the following commands do?

- echo "Hello, World!"
  ⇨ Displays "Hello, World!" on the screen
- name="Productive"
  ⇨ Assigns the word "Productive" to the variable name
- touch file.txt
  ⇨ Creates an empty file named file.txt.
- ls -a
  ⇨ Lists all files and directories, including hidden ones.
- rm file.txt
  ⇨ Deletes file.txt
- cp file1.txt file2.txt
  ⇨ Copies file1.txt content into file2.txt.
- mv file.txt /path/to/directory/
  ⇨ Moves file.txt to a different folder (/path/to/directory/).
- chmod 755 script.sh
  ⇨ Gives the owner full access (read, write, execute) and allows others to only read and execute script.sh.
- grep "pattern" file.txt
  ⇨ Searches for the word "pattern" in file.txt and shows matching lines.
- kill PID
  ⇨ Stops the process with the given process ID (PID)
- mkdir mydir && cd mydir && touch file.txt && echo "Hello, World!" > file.txt && cat file.txt
  ⇨ Creates a folder mydir, moves into it, creates file.txt, writes "Hello, World!" inside, and then displays the content i.e. “Hello World!”
- ls -l | grep ".txt"
  ⇨ Lists all files with detailed info and filters only .txt files.
- cat file1.txt file2.txt | sort | uniq
  ⇨ Merges file1.txt and file2.txt, sorts them, and removes duplicate lines.
- ls -l | grep "^d"
  ⇨ Lists only directories from the current location.
- grep -r "pattern" /path/to/directory/
  Searches for "pattern" in all files within /path/to/directory/, including subfolders.
- cat file1.txt file2.txt | sort | uniq –d
  ⇨ Shows only the duplicate lines found in file1.txt and file2.txt.

- chmod 644 file.txt
  - ⇨ Gives the owner read and write permissions but allows group and others only to read file.txt.
- cp -r source_directory destination_directory
  - ⇨ Copies folder source_directory along with its contents to destination_directory.
- find /path/to/search -name "*.txt"
  - ⇨ Looks for all file whose name is ending with .txt inside specified directory i.e. /path/to/search.
- chmod u+x file.txt
  - ⇨ Gives execute permission to the owner for the file file.txt
- echo $PATH
  - ⇨ Displays the system's PATH variable, showing 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.
   - ⇨ **True** : ls lists files and directories in the current or specified directory.
2. **mv** is used to move files and directories.
   - ⇨ **True** : mv is used to move files and directories.
3. **cd** is used to copy files and directories.
   - ⇨ **False :** cd is used to **change directories**, not copy files. The correct command for copying is cp.
4. **pwd** stands for "print working directory" and displays the current directory.
   - ⇨ **False** : pwd stands for present working directory not for print working directory however, it displays the current directory.
5. **grep** is used to search for patterns in files.
   - ⇨ **True** – grep searches for a specific pattern in files.

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

    ⇨ **True** – chmod 755 file.txt gives full permissions to the owner (read, write, execute) and read/execute permissions to others.

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

    ⇨ **True** – mkdir -p directory1/directory2 creates nested directories, ensuring directory1 exists before creating directory2.

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

    ⇨ **True** – rm -rf file.txt forcefully removes file.txt without asking for confirmation.

Identify the Incorrect Commands:

1. **chmodx** is used to change file permissions.

    ⇨ **Incorrect :** The correct command is chmod to change file permissions**.**

2. **cpy** is used to copy files and directories.

    ⇨ **Incorrect :** The correct command is cp to copy files and directories.

3. mkfile is used to create a new file.

    ⇨ **Incorrect :** There is no mkfile command in Linux. To create a new file, use touch filename.

4. **catx** is used to concatenate files.

    ⇨ **Incorrect :** The correct command is cat to concatenate and display files.

5. **rn** is used to rename files.

    ⇨ **Incorrect :** The correct command to rename files is mv oldname newname.

Part C

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

```
mysf@mysf:~/bash_programs$ cat HW.sh
echo "Hello World!"
mysf@mysf:~/bash_programs$ bash HW.sh
Hello World!
mysf@mysf:~/bash_programs$
```

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

```
mysf@mysf:~/bash_programs$ nano var.sh
mysf@mysf:~/bash_programs$ cat var.sh
name="CDAC Mumbai"
echo $name
mysf@mysf:~/bash_programs$ bash var.sh
CDAC Mumbai
mysf@mysf:~/bash_programs$
```

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

```
mysf@mysf:~/bash_programs$ nano ip.sh
mysf@mysf:~/bash_programs$ cat ip.sh
echo Please enter a number :
read n
echo $n
mysf@mysf:~/bash_programs$ bash ip.sh
Please enter a number :
5
5
mysf@mysf:~/bash_programs$
```

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

```
mysf@mysf:~/bash_programs$ nano add.sh
mysf@mysf:~/bash_programs$ cat add.sh
a=5
b=3
echo "Sum of $a and $b is $((a+b))"
mysf@mysf:~/bash_programs$ bash add.sh
Sum of 5 and 3 is 8
mysf@mysf:~/bash_programs$
```

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

```
mysf@mysf:~/bash_programs$ nano OddEven.sh
mysf@mysf:~/bash_programs$ cat OddEven.sh
echo Enter a number  :
read n

if ((n%2==0))
then
echo Even

else

echo Odd
fi
mysf@mysf:~/bash_programs$ bash OddEven.sh
Enter a number :
3
Odd
mysf@mysf:~/bash_programs$ bash OddEven.sh
Enter a number :
4
Even
mysf@mysf:~/bash_programs$
```

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

```
mysf@mysf:~/bash_programs$ nano for.sh
mysf@mysf:~/bash_programs$ cat for.sh
for ((i=1; i<=5 ;i++))
        do
        echo $i
        done
mysf@mysf:~/bash_programs$ bash for.sh
1
2
3
4
5
mysf@mysf:~/bash_programs$
```

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

```
mysf@mysf:~/bash_programs$ nano while.sh
mysf@mysf:~/bash_programs$ cat while.sh
i=1
while (( i <= 5 ))
do
echo $i
i=$((i+1))
done
mysf@mysf:~/bash_programs$ bash while.sh
1
2
3
4
5
mysf@mysf:~/bash_programs$
```

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".

```
mysf@mysf:~/bash_programs$ nano CheckFile.sh
mysf@mysf:~/bash_programs$ cat CheckFile.sh
if [ -f "file.txt" ]; then
        echo "File exists"
else
        echo "File does not exist"
fi
mysf@mysf:~/bash_programs$ bash CheckFile.sh
File does not exist
mysf@mysf:~/bash_programs$ touch file.txt
mysf@mysf:~/bash_programs$ bash CheckFile.sh
File exists
mysf@mysf:~/bash_programs$
```

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.

```
mysf@mysf:~/bash_programs$ nano GreaterThan10.sh
mysf@mysf:~/bash_programs$ cat GreaterThan10.sh
echo Enter a number :
read n

if (( n > 10 ))
then

        echo The entered number is greater than 10

else

        echo The entered number is not greater than 10
fi
mysf@mysf:~/bash_programs$ bash GreaterThan10.sh
Enter a number :
12
The entered number is greater than 10
mysf@mysf:~/bash_programs$ bash GreaterThan10.sh
Enter a number :
8
The entered number is not greater than 10
mysf@mysf:~/bash_programs$
```

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.

```
mysf@mysf:~/bash_programs$ nano table.sh
mysf@mysf:~/bash_programs$ cat table.sh
for (( i = 1 ; i<=5 ; i++))
do

        for (( j=1 ; j <= 10 ; j++))
        do

                echo $i x $j = $((i*j))
        done
        echo ----------------------------------------------------------------------------------------

done
mysf@mysf:~/bash_programs$ bash table.sh
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10
----------------------------------------------------------------------------------------
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
----------------------------------------------------------------------------------------
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
----------------------------------------------------------------------------------------
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40
----------------------------------------------------------------------------------------
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
----------------------------------------------------------------------------------------
mysf@mysf:~/bash_programs$
```

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.

```
mysf@mysf:~/bash_programs$ nano while2.sh
mysf@mysf:~/bash_programs$ cat while2.sh
while true
do
        echo enter a number
        read n

        if ((n<0))
        then
                echo Negative number... Exiting...
                break
        fi
        echo $((n*n))
done
mysf@mysf:~/bash_programs$ bash while2.sh
enter a number
4
16
enter a number
2
4
enter a number
6
36
enter a number
-1
Negative number... Exiting...
mysf@mysf:~/bash_programs$
```

Part D

Common Interview Questions (Must know)

1. What is an operating system, and what are its primary functions?
   An Operating System is system software that acts as an interface between hardware and users, it manages system resources efficiently.
   Primary functions:
   - Process Management : Handles execution of programs (process scheduling, creation, termination).
   - Memory Management : Allocates and deallocates memory for processes.
   - File System Management : Manages data storage, retrieval, and organization.
   - Device Management : Controls input/output devices via drivers.
   - Security & Access Control : Ensures data integrity and user authentication.

2. Explain the difference between process and thread.
   Process:
   - A program in execution with its own memory and resources.
   - Independent from other processes.
   - Context switching between processes is slow.

   Thread:
   - A small execution unit within a process.
   - Shares memory with other threads of the same process.
   - Faster context switching than processes.

3. What is virtual memory, and how does it work?
   Virtual memory is a technique that allows the execution of programs larger than the physical RAM by using secondary storage as an extension of RAM.
   It works by:
   - Dividing memory into pages and storing inactive pages on disk.
   - Using paging and swapping to move data between RAM and disk.
   - Allowing multiple processes to run without being limited by physical memory size.

4. Describe the difference between multiprogramming, multitasking, and multiprocessing.
   - Multiprogramming : Multiple programs are loaded into memory and executed one by one, increasing CPU utilization.
   - Multitasking : CPU switches between multiple tasks quickly, giving an illusion of parallel execution.
   - Multiprocessing : Multiple CPUs execute different processes simultaneously, improving performance.

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, and manage files on a storage device. It ensures data is structured in a way that allows efficient access and retrieval. The key components of a file system includes:
   - Files : which store data in different formats
   - Directories: which act as containers for organizing files.
   - Metadata: such as file attributes, size.
   - Timestamps: is stored in structures called inodes or file allocation tables, depending on the file system type.
   - The superblock contains information about the overall structure of the file system. Together, these components ensure that the OS can locate, read, and modify files efficiently.

6. What is a deadlock, and how can it be prevented?
   A deadlock occurs when a group of processes become stuck in a state where each process is waiting for a resource that another process is holding, creating a cycle of dependency that prevents further execution. This situation can severely impact system performance and stability. Deadlock prevention can be achieved by avoiding circular wait conditions by enforcing a specific order in which resources are requested. Another approach is to require processes to request all necessary resources at the beginning to prevent them from holding resources while waiting for others. Deadlock detection mechanisms can also be implemented to identify and resolve deadlocks by terminating or rolling back certain processes.

7. Explain the difference between a kernel and a shell.
   The kernel is the core component of an operating system that directly interacts with hardware and manages system resources such as CPU scheduling, memory allocation, and device communication. It operates in a privileged mode, meaning it has full control over system operations. The shell, on the other hand, is a user interface that allows users to interact with the OS by entering commands. It translates user commands into instructions that the kernel can execute. There are different types of shells, such as command-line interfaces (CLI) like Bash and graphical user interfaces (GUI) like Windows Explorer, but both serve as intermediaries between the user and the system.

8. What is CPU scheduling, and why is it important?
   CPU scheduling is the process of determining which process gets CPU time when multiple tasks are competing for execution. It helps in optimizing CPU usage and ensuring fair process execution. Different scheduling algorithms like First Come First Served (FCFS), Shortest Job First (SJF) and Round Robin (RR) are used based on system requirements. Effective CPU scheduling reduces waiting time, improves system responsiveness, and enhances overall performance, making it a crucial aspect of operating

9. How does a system call work?
   A system call is a mechanism that allows a user program to request services from the operating system. Since user programs do not have direct access to hardware or sensitive system resources, they must use system calls to perform operations like file manipulation, process creation, and communication with devices. When a program makes a system call, it triggers a transition from user mode to kernel mode, where the OS processes the request. This transition is done by generating a Trap/Interrupt signal. The OS then executes the requested operation and returns the result to the user program, switching back to user mode. This mechanism ensures controlled access to system resources while maintaining security and stability.

10. What is the purpose of device drivers in an operating system?
    Device drivers are specialized software components that allow the operating system to communicate with hardware devices such as printers, keyboards, and storage drives. Since different hardware components have unique specifications and communication protocols, device drivers act as intermediaries that translate OS commands into device-specific instructions. They enable plug-and-play functionality, allowing new devices to be detected and used without manual configuration. Without device drivers, the OS would not be able to interact with hardware efficiently, making them essential for hardware compatibility and system functionality.

11. Explain the role of the page table in virtual memory management.
    The page table is a data structure used in virtual memory management to map virtual addresses to physical addresses. Since programs use virtual addresses while the actual data resides in physical memory, the operating system maintains a page table to keep track of these mappings. Each entry in the page table contains information such as the physical frame number where the corresponding page is stored. The CPU's memory management unit (MMU) consults the page table during memory access, ensuring seamless translation of virtual addresses. This enables efficient memory utilization and allows processes to run even if they are not fully loaded into RAM.

12. What is thrashing, and how can it be avoided?
    Thrashing occurs when a system spends most of its time swapping data between RAM and virtual RAM rather than executing processes. It happens when there is insufficient main memory, causing excessive page faults and slowing down performance. To prevent thrashing, operating systems use techniques like working set models to track frequently accessed pages and keep them in memory. Adjusting the degree of multiprogramming, using page replacement algorithms like Least Recently Used (LRU), and increasing RAM can also help in reducing thrashing.

13. Describe the concept of a semaphore and its use in synchronization.
    A semaphore is a synchronization mechanism used to control access to shared resources in a concurrent system. It is essentially a counter that manages the number of processes allowed to use a resource simultaneously. Semaphores can be of two types: binary semaphores, which work like locks, allowing only one process at a time, and counting semaphores, which allow multiple processes based on availability. They prevent race conditions and ensure orderly execution in multi-threaded or multi-process environments. For example, in a producer-consumer problem, semaphores help manage buffer usage efficiently by signaling when space is available or when data is ready for consumption.

14. How does an operating system handle process synchronization?
    The operating system ensures process synchronization by using mechanisms like semaphores, mutexes, and monitors to prevent race conditions and data inconsistency. When multiple processes access shared resources, synchronization techniques ensure that only one process can modify the resource at a time.

15. What is the purpose of an interrupt in operating systems?
    An interrupt is a signal sent to the CPU to indicate that an immediate action is required, allowing the system to respond quickly to events. Interrupts can be hardware-based, such as a keyboard input or network request, or software-based, like system calls. The CPU temporarily pauses its current execution, processes the necessary execution asked for and then resumes normal execution. This mechanism improves efficiency by allowing multitasking and enabling the OS to respond to user interactions and hardware events in real time.

16. Explain the concept of a file descriptor.
    A file descriptor is a unique identifier assigned by the operating system to an open file, socket, or device. It acts as a reference for performing operations such as reading, writing, or closing the file. File descriptors are typically represented as integer values and are managed by the OS in a file descriptor table. Standard file descriptors include 0 for standard input (stdin), 1 for standard output (stdout), and 2 for standard error (stderr). They allow efficient file handling in system programming and enable process communication through pipes and sockets.

17. How does a system recover from a system crash?
    System recovery after a crash involves restoring the system to a stable state and minimizing data loss. The OS uses recovery techniques such as journaling file systems, which log changes before committing them to disk, allowing rollback in case of failure. Checkpointing periodically saves system states, enabling a restart from a known safe point. The OS also performs crash dumps to analyze failure causes. Automated restart mechanisms and redundant backups further enhance system reliability and prevent permanent data corruption.

18. Describe the difference between a monolithic kernel and a microkernel.
    A monolithic kernel is a type of OS kernel where all essential services, such as memory management, process scheduling, and I/O operations, run in kernel mode. This design allows high performance but increases the risk of system crashes due to tightly coupled components. In contrast, a microkernel has a minimal core, with only basic functions like inter-process communication and memory management, while other services run in user space. This modular approach improves security and stability but may introduce performance overhead due to increased communication between kernel and user processes.

19. What is the difference between internal and external fragmentation?
    Internal fragmentation occurs when allocated memory blocks are slightly larger than the required space, leading to unused memory within an allocated block. This happens when fixed-size memory partitions are used. External fragmentation, on the other hand, happens when free memory is scattered in small, non-contiguous blocks, preventing large processes from being allocated memory even if sufficient space is available. Techniques like paging and compaction help in reducing fragmentation and improving memory utilization.

20. How does an operating system manage I/O operations?
    The OS manages I/O operations by acting as an intermediary between hardware devices and user applications. It provides device drivers that translate system calls into hardware-specific commands. The OS also uses buffering to temporarily store data for efficient processing, while spooling allows managing multiple I/O requests in a queue. Additionally, interrupt-driven and direct memory access (DMA) mechanisms optimize data transfers without burdening the CPU. Through these techniques, the OS ensures smooth, efficient, and error-free communication between the system and peripheral devices.

21. Explain the difference between preemptive and non-preemptive scheduling.
    Preemptive scheduling allows the operating system to interrupt a currently running process and allocate the CPU to another process if needed. This ensures better resource utilization and responsiveness, as higher-priority or time-sensitive tasks can take control when required. In contrast, non-preemptive scheduling ensures that once a process starts execution, it runs until completion or until it voluntarily releases the CPU. This method reduces overhead but can lead to inefficiencies if a long-running process blocks others.

22. What is round-robin scheduling, and how does it work?
    Round-robin scheduling is a CPU scheduling algorithm that assigns a fixed time slice, called a time quantum, to each process in a cyclic order. When a process gets the CPU, it executes for a maximum of this time quantum before being moved to the end of the queue, allowing the next process to execute. If the process finishes within its time slice, it exits the queue. This method ensures fairness among processes and prevents any single process from monopolizing the CPU, making it ideal for time-sharing systems.

23. Describe the priority scheduling algorithm. How is priority assigned to processes?
    Priority scheduling is a method where the CPU is allocated to processes based on their priority levels. Each process is assigned a priority, and the scheduler selects the process with the highest priority for execution. If multiple processes have the same priority, they are scheduled based on another criterion like arrival time. Priorities can be assigned dynamically based on factors like process importance, execution deadlines, or resource requirements. However, this algorithm may lead to starvation, where low-priority processes are delayed indefinitely, which can be mitigated using aging which is done by gradually increasing the priority of waiting processes.

24. What is the shortest job next (SJN) scheduling algorithm, and when is it used?
    Shortest Job (SJF), also known as Shortest Job First (SJF), is a scheduling algorithm that selects the process with the shortest execution time for the CPU. It minimizes average waiting time and improves efficiency. SJN can be either preemptive (Shortest Remaining Time First) or non-preemptive. It is commonly used in batch processing where execution time is known beforehand. However, it suffers from the "starvation" problem, where longer jobs may face indefinite delays if shorter jobs keep arriving

25. Explain the concept of multilevel queue scheduling.
    Multilevel queue scheduling divides processes into different queues based on priority, type, or resource needs. Each queue follows a specific scheduling algorithm, and scheduling occurs both within a queue and between queues. For example, system processes may reside in a high-priority queue using priority scheduling, while user processes may be in a lower queue using round-robin scheduling.

26. What is a process control block (PCB), and what information does it contain?
    A Process Control Block (PCB) is a data structure that stores essential information about a process. It contains details such as the process ID, process state, CPU registers, scheduling information, memory management details, open file lists, and security information. The OS uses the PCB to track process execution and manage context switching, ensuring that processes resume correctly after being interrupted.

27. Describe the process state diagram and the transitions between different process states.
    The process state diagram represents the lifecycle of a process in an operating system. The main states include:
    - New: The process is created but not yet scheduled for execution.
    - Ready: The process is waiting in the queue for CPU allocation.
    - Running: The process is actively executing on the CPU.
    - Waiting: The process is paused, waiting for I/O or an event.
    - Terminated: The process has finished execution and is removed from memory.

28. How does a process communicate with another process in an operating system?
    Processes communicate using Inter-Process Communication (IPC) mechanisms such as message passing and shared memory. In message passing, processes send and receive messages using communication channels like pipes, sockets, or queues. Shared memory allows multiple processes to access a common memory space, ensuring faster data exchange. Synchronization mechanisms like semaphores and mutexes prevent data inconsistencies when multiple processes modify shared data simultaneously.

29. What is process synchronization, and why is it important?
    Process synchronization ensures that multiple processes execute in a coordinated manner without interfering with each other. It is crucial in multi-threaded environments where processes share resources such as files, databases, or memory. Synchronization prevents race conditions, where multiple processes attempt to modify shared data simultaneously, leading to inconsistencies. Mechanisms like semaphores, mutexes, and monitors are used to ensure orderly execution and avoid deadlocks.

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 process has not yet retrieved its termination status. When a process finishes, the OS keeps its exit status until the parent collects it using the wait() system call. If the parent does not handle it properly, the zombie process remains, occupying system resources unnecessarily.

31. Describe the difference between internal fragmentation and external fragmentation.
    Internal fragmentation occurs when memory is allocated in fixed-size blocks, leading to wasted space inside allocated memory if a process does not fully utilize it. External fragmentation happens when free memory is scattered across different locations, making it difficult to allocate large contiguous blocks even if the total free memory is sufficient. Internal fragmentation can be reduced by using dynamic memory allocation, while external fragmentation is mitigated through memory compaction or paging techniques.

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 memory only when they are needed, rather than loading the entire process at once. This reduces memory usage and allows multiple processes to share available RAM efficiently. If a required page is not in memory, a page fault occurs, and the operating system fetches the page from disk. Demand paging enhances system performance by reducing initial load times and enabling better multitasking.

33. Explain the role of the page table in virtual memory management.
    The page table maps virtual addresses to physical memory locations. When a process accesses memory, the Memory Management Unit (MMU) consults the page table to determine the corresponding physical address. The page table helps in implementing paging, supporting virtual memory, and enabling efficient memory allocation. It can also include details like frame numbers, valid/invalid bits, and protection bits to manage memory access permissions.

34. How does a memory management unit (MMU) work?
    The MMU is a hardware component that translates virtual addresses into physical addresses during memory access. When a process requests data, the MMU checks the page table and retrieves the corresponding physical address. The MMU speeds up address translation using Translation Lookaside Buffers (TLB), which cache frequently used address mappings.

35. What is thrashing, and how can it be avoided in virtual memory systems?
    Thrashing occurs when a system spends most of its time swapping data between RAM and virtual RAM rather than executing processes. It happens when there is insufficient main memory, causing excessive page faults and slowing down performance. To prevent thrashing, operating systems use techniques like working set models to track frequently accessed pages and keep them in memory. Adjusting the degree of multiprogramming, using page replacement algorithms like Least Recently Used (LRU), and increasing RAM can also help in reducing thrashing.

36. What is a system call, and how does it facilitate communication between user programs and the operating system?
    A system call is an interface between user applications and the operating system that allows programs to request services such as file handling, process management, and memory allocation. When a user program makes a system call, the CPU switches from user mode to kernel mode, allowing the OS to execute privileged operations.

37. Describe the difference between a monolithic kernel and a microkernel.
    A monolithic kernel is a type of OS kernel where all essential services, such as memory management, process scheduling, and I/O operations, run in kernel mode. This design allows high performance but increases the risk of system crashes due to tightly coupled components. In contrast, a microkernel has a minimal core, with only basic functions like inter-process communication and memory management, while other services run in user space. This modular approach improves security and stability but may introduce performance overhead due to increased communication between kernel and user processes.

38. How does an operating system handle I/O operations?
    The OS manages I/O operations through device drivers, which act as intermediaries between hardware devices and the OS. It uses interrupts to notify the CPU when an I/O task is complete, avoiding unnecessary waiting. Buffering, caching, and spooling techniques optimize I/O performance. The OS also schedules I/O requests to ensure efficient and fair access to devices.

39. Explain the concept of a race condition and how it can be prevented.
    A race condition occurs when multiple processes or threads access and modify shared data simultaneously, leading to unpredictable behavior. It can be prevented using synchronization mechanisms such as mutexes, semaphores, and locks, which ensure that only one process modifies the shared resource at a time.

40. Describe the role of device drivers in an operating system.
    Device drivers are software components that enable the OS to communicate with hardware devices such as printers, keyboards, and storage drives. They provide a standardized interface so that applications can interact with hardware without needing to know specific details. Drivers translate OS commands into hardware-specific instructions and handle tasks like data transfer and error handling.

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 occupies an entry in the process table because its parent has not retrieved its exit status. It occurs when a child process terminates, but the parent does not call wait() to remove its entry. Zombie processes can be prevented by using proper process management, ensuring that the parent process handles child termination correctly, or by setting the child process as an orphan so that it gets adopted by the init process, which automatically reaps zombies.

42. Explain the concept of an orphan process. How does an operating system handle orphan processes?
An orphan process is a process whose parent has terminated before it completes execution. Orphan processes do not have a parent to handle their termination, but the OS automatically assigns them to the init process, which ensures they are cleaned up properly. The init process periodically checks for orphan processes and calls wait() to remove their entries from the process table, preventing resource wastage.

43. What is the relationship between a parent process and a child process in the context of process management?
A parent process is a process that creates one or more child processes using system calls like fork(). The child process is a duplicate of the parent but has a unique process ID. Both processes can run independently, but the parent often monitors the child's execution. The parent can wait for the child's termination using wait() or waitpid() to clean up system resources. If a parent terminates before its child, the child becomes an orphan process, which is then adopted by the init process.

44. How does the fork() system call work in creating a new process in Unix-like operating systems?
The fork() system call is used to create a new child process by duplicating the parent process. When fork() is called, the OS creates a new process with a unique Process ID and copies the parent's address space, including variables and program instructions. The return value of fork() helps in distinguishing the processes:
- It returns 0 in the child process.
- It returns the child's PID in the parent process.
- A negative return value indicates failure.

After the fork(), both parent and child execute the same code, but they operate independently.

45. Describe how a parent process can wait for a child process to finish execution.
A parent process can wait for its child to complete using the wait() or waitpid() system calls. When a parent calls wait(), it pauses execution until one of its child processes terminates, at which point it retrieves the child's exit status and removes it from the process table. waitpid() provides more control by allowing the parent to wait for a specific child process. These system calls help prevent zombie processes, which occur when a child exits but its termination status is not collected.

46. What is the significance of the exit status of a child process in the wait() system call?
The exit status of a child process provides information about how the process terminated. When a parent calls wait(), it receives an integer status code that indicates whether the child:
- Exited normally, exit() was called.
- Was terminated by a signal.
- Encountered an error.

By analyzing the exit status, the parent can determine if the child completed successfully or encountered an issue, allowing proper error handling and resource management.

47. How can a parent process terminate a child process in Unix-like operating systems?
    A parent process can terminate a child using the kill() system call, which sends a signal to the child. For example, kill(pid, SIGTERM) requests graceful termination, while kill(pid, SIGKILL) forcefully stops it. Alternatively, the exit() function can be used inside the child process to voluntarily terminate itself. A parent can also use setpgid() to manage multiple child processes and terminate an entire group at once.

48. Explain the difference between a process group and a session in Unix-like operating systems.
    A process group is a collection of related processes that can be controlled together, often used to manage multiple processes within a shell or job control system. Each process group has a unique group ID (PGID), and signals like `SIGKILL` can be sent to all processes in a group.
    A session is a higher-level structure that contains one or more process groups. A session typically starts when a user logs in and runs a shell, which then spawns multiple process groups. The session leader (usually a shell) can control which process group is actively receiving input.

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, such as execl(), execv(), and execvp(), replaces the current process's memory image with a new program. After a successful exec(), the calling process does not continue its previous execution; instead, it runs the new program. This is useful when a process wants to execute a different program without creating a new process using fork(). The child process in a fork()-exec() sequence often calls exec() to load and execute a new program while keeping the same PID.

50. What is the purpose of the waitpid() system call in process management? How does it differ from wait()?
    The waitpid() system call is used to wait for a specific child process to terminate, offering more control than wait(), which waits for any child. It allows a parent to:
    - Wait for a particular child's PID.
    - Check the status of a child without blocking (WNOHANG).
    - Handle multiple child processes selectively.

    This flexibility is useful when managing multiple child processes in complex applications.

51. How does process termination occur in Unix-like operating systems?
    The waitpid() system call is used to wait for a specific child process to terminate, offering more control than wait(), which waits for any child. It allows a parent to:
    - Wait for a particular child's PID.
    - Check the status of a child without blocking (WNOHANG).
    - Handle multiple child processes selectively.
      This flexibility is useful when managing multiple child processes in complex applications.

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?
    The long-term scheduler is responsible for deciding which processes are admitted into the system for execution. It controls the degree of multiprogramming by determining how many processes should be allowed in memory at a time. If too many processes are loaded, it may lead to thrashing due to excessive memory swapping. By regulating process admission, the long-term scheduler ensures that system resources are optimally utilized, balancing performance and responsiveness.

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?
    The short-term scheduler executes most frequently, selecting processes from the ready queue to allocate CPU time, making decisions every few milliseconds. The long-term scheduler runs less often, deciding which processes enter the system for execution, controlling the degree of multiprogramming. The medium-term scheduler operates occasionally, temporarily removing (swapping out) processes from memory to optimize resource usage and prevent issues like thrashing.

54. Describe a scenario where the medium-term scheduler would be invoked and explain how it helps manage system resources more efficiently.
    The medium-term scheduler is invoked when system memory is heavily used, and processes are waiting for resources, leading to reduced performance. For example, if multiple processes are waiting for I/O operations, keeping them in memory is inefficient. The scheduler swaps out these inactive processes to free RAM, allowing active processes to run smoothly. Once resources are available, the suspended processes are swapped back into memory. This prevents thrashing, optimizes resource allocation, and improves overall system efficiency.

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.

Solution :

Process	Arrival time	Burst time	Waiting time	Turn Around time	Response Time
P1	0	5	0	5	0
P2	1	3	4	7	4
P3	2	6	6	12	6
			Avg = 3.33		

Gantt Chart

	P1		P2		P3	
0		5		8		14

Thus, The average waiting time for the given processes is 3.33

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.

Process	Arrival time	Burst time	Turn Around time	Response Time
P1	0	3	3	0
P2	1	5	12	8
P3	2	1	2	3
P4	3	4	5	4
			Avg = 5.5	

Gantt Chart

	P1		P3		P4		P2	
0		3		4		8		13

Thus, The average turn around time for the given processes is 5.5

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.

Process	Arrival time	Burst time	Priority	Waiting time	Turn Around time	Response Time
P1	0	6	3	0	6	0
P2	1	4	1	5	9	6
P3	2	7	4	10	17	12
P4	3	2	2	7	9	10
				Avg=5.5		

Gantt Chart

	P1		P2		P4		P3	
0		6		10		12		19

Thus, The average waiting time for the given processes is 5.5

3. 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.

Process	Arrival time	Burst time	Waiting time	Turn Around time	Response Time
P1	0	4	6	10	0
P2	1	5	8	13	2
P3	2	2	2	4	4
P4	3	3	7	10	6
			Avg=9.25		

Gantt chart :

	P1		P2		P3		P4		P1		P2		P4		P2	
0		2		4		6		8		10		12		13		14

Thus, The average turnaround time for the given processes is 9.25

4. 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?

When a process calls fork(), it creates a new child process that is a duplicate of the parent process. However, the parent and child have separate memory spaces, meaning changes made to variables in one process do not affect the other. Steps to calculate :

1. The parent process starts with x = 5.
2. It calls fork(), which creates a child process. Now, both parent and child have their own copies of x, initially set to 5.
3. Both processes execute their respective code and increment x by 1:

⇨ The parent's x becomes 6.
⇨ The child's x also becomes 6.

Final Values of x:

- Parent process: x = 6
- Child process: x = 6