🌐 PRACTICAL 1: Demonstration of OS Functions Using OS Simulator

✔ Aim

To study and demonstrate various operating system functions including process management, CPU scheduling, memory allocation strategies, file management, and deadlock handling using an OS Simulator.

✔ Theory (Detailed)

1. Process Management

- A **process** is a program in execution.
- The OS stores process information in **PCB (Process Control Block)**.
- PCB contains: Process ID, Program Counter, Registers, Priority, State, CPU scheduling info.
- **Process states**:
  - New
  - Ready
  - Running
  - Waiting (Blocked)
  - Terminated
- OS uses **context switching** to move CPU between processes.

---

2. CPU Scheduling Algorithms

a. FCFS (First Come First Serve)

- Non-preemptive
- Processes executed in order of arrival
- Simple, but leads to **convoy effect**
- Average waiting time is high

b. SJF (Shortest Job First)

- Process with smallest CPU burst runs first
- Can be preemptive or non-preemptive
- Gives **optimal average waiting time**
- Needs prediction of next CPU burst

c. Priority Scheduling

- Each process has a priority
- Highest priority = scheduled first
- Lower priority may starve

d. Round Robin

- Each process gets fixed CPU time quantum
- Preemptive
- Fair scheduling
- Best for **time-sharing operating systems**

---

3. Memory Allocation Techniques

a. First Fit

- Allocates **first block** large enough for process
- Fastest but causes moderate fragmentation

b. Best Fit

- Allocates **smallest suitable** block
- Minimizes leftover space but slows allocation

c. Worst Fit

- Allocates **largest available block**
- Leaves largest leftover hole
- Less fragmentation, slow performance

Fragmentation

- **Internal**: unused space inside allocated block
- **External**: free memory broken into small holes

---

4. File Management

- OS organizes files in directories
- Directory maintains filenames, sizes, permissions
- Operations:
    - Create
    - Open
    - Read
    - Write
    - Delete
    - Rename

---

5. Deadlock

Occurs when multiple processes wait indefinitely for each other's resources.

Conditions for deadlock (Coffman):

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

OS Simulator shows:

- Resource Allocation Graph
- Banker's algorithm

✔ Procedure

1. Launch the OS Simulator.
2. Select module: Process / CPU Scheduling / Memory / File / Deadlock.
3. Enter required inputs: process ID, burst time, arrival time, priority.
4. Execute simulation.
5. Observe:
    - Ready queue
    - Gantt chart
    - Waiting time / turnaround time
    - Memory block allocation
6. Save outputs (tables, diagrams).

✔ Sample Outputs (Highly Detailed)

1. Process Table

```
PID | AT | BT | Priority | State
---------------------------------
P1  | 0  |  5 |    2     | Ready
P2  | 1  |  3 |    1     | Running
P3  | 2  |  4 |    3     | Waiting
```

2. FCFS Gantt Chart

```
P1      P2      P3
0------5------8------12
```

3. SJF Detailed Table

```
Process | AT | BT | WT | TAT
-----------------------------
P2      | 1  | 2  | 0  | 2
P1      | 0  | 5  | 2  | 7
P3      | 2  | 8  | 7  | 15
```

4. First Fit Allocation

```
Block | Size | Process | Used | Leftover
----------------------------------------
B1    | 100  | P1      |  80  |   20
B2    |  50  | P2      |  45  |    5
B3    | 120  | ---     |   0  |  120
```

✔ Conclusion

The OS Simulator effectively demonstrates internal working of OS concepts like scheduling, memory allocation, process management, and file handling through visual tables, Gantt charts, and allocation diagrams.

✔ Viva Questions (Detailed Answers)

1. **What is scheduling?**
   Scheduling selects which process receives CPU next based on an algorithm like FCFS, SJF, Priority, RR.
2. **Why does SJF give minimum waiting time?**
   Because shorter jobs finish quickly and reduce waiting time of longer jobs.
3. **Difference between internal and external fragmentation?**
   Internal: unused memory inside allocated block.
   External: unused memory between blocks.

4. **What is starvation?**
   Low-priority processes never get CPU because higher priorities keep coming.
5. **What is context switching?**
   Saving and loading PCB to switch CPU between processes.
6. **What is Deadlock?**
   When two or more processes wait for each other forever.

🌟 PRACTICAL NO. 2

Execution of Linux Commands

✅ AIM

To study and execute Linux commands for information maintenance, file management, and directory management in a Linux environment.

✅ THEORY (DETAILED & EXAM-READY)

Linux is a multi-user, multitasking operating system.
It provides a powerful **shell** (command-line interface) through which users can perform operations such as:

- Viewing system information
- Managing users and files
- Creating, modifying, and deleting files
- Changing directories
- Viewing date, calendar, file sizes, and permissions

The commands are divided into categories based on function.

🔷 PART I — INFORMATION MAINTENANCE COMMANDS

⭐ 1. wc — Word Count Command

Syntax:

```
wc filename
```

Purpose:
Displays **number of lines, words, and characters** in a file.

Options:

- `wc -l` → count lines
- `wc -w` → count words
- `wc -c` → count characters

⭐ 2. clear — Clear the Terminal

Syntax:

```
clear
```

Purpose:
Clears terminal screen.

⭐ 3. cal — Display Calendar

Syntax:

```
cal
```

OR (if cal not available):

```
ncal
```

Purpose:
Displays monthly calendar.

⭑ 4. who — Show Logged-in Users

Syntax:

```
who
```

Purpose:
Shows users currently logged into system (user, terminal, login time).

⭑ 5. date — Display Date & Time

Syntax:

```
date
```

Purpose:
Displays current system date, day, and time.

⭑ 6. pwd — Print Working Directory

Syntax:

```
pwd
```

Purpose:
Displays the directory you are currently in.

🔷 PART II — FILE MANAGEMENT COMMANDS

⭑ 1. cat — Display File Contents

Syntax:

```
cat filename
```

Purpose:
Displays the content of a file.

⋆ 2. cp — Copy Files

Syntax:

```
cp source destination
```

⋆ 3. rm — Remove/Delete File

Syntax:

```
rm filename
```

CAUTION: Cannot be undone.

⋆ 4. mv — Move/Rename File

Syntax:

```
mv oldname newname
```

⋆ 5. cmp — Compare Files Byte-by-Byte

Syntax:

```
cmp file1 file2
```

⋆ 6. comm — Compare 2 Sorted Files

Syntax:

```
comm file1 file2
```

★ 7. diff — Show Differences Between Files

Syntax:

```
diff file1 file2
```

★ 8. find — Search for Files

Syntax:

```
find /path -name filename
```

★ 9. grep — Pattern Search

Syntax:

```
grep "word" filename
```

Purpose:
Searches for matching text inside a file.

★ 10. awk — Field Processing

Syntax:

```
awk '{print $1}' filename
```

Purpose:
Print specific column from file.

🔷 PART III — DIRECTORY MANAGEMENT COMMANDS

★ 1. cd — Change Directory

Syntax:

```
cd dirname
```

★ 2. mkdir — Make Directory

Syntax:

```
mkdir dirname
```

★ 3. rmdir — Remove Directory

Syntax:

```
rmdir dirname
```

★ 4. ls — List Directory Contents

Syntax:

```
ls
```

Options:
`ls -l` → long listing
`ls -a` → show hidden files
`ls -lh` → human-readable

🔷 SAMPLE OUTPUTS (WRITE THESE IN JOURNAL)

You may copy EXACTLY these:

```
$ wc file.txt
 3  12  85 file.txt

$ cal
    November 2025
```

```
Su Mo Tu We Th Fr Sa
                   1
2  3  4  5  6  7  8
...

$ who
student  tty1  2025-11-27  10:25

$ date
Thu Nov 27 10:30:21 IST 2025

$ pwd
/home/student

$ mkdir test
$ cd test
$ pwd
/home/student/test

$ echo "hello" > a.txt
$ cat a.txt
hello

$ cp a.txt b.txt
$ diff a.txt b.txt

$ grep "hello" a.txt
hello
```

🔷 RESULT

Various Linux commands for information maintenance, file handling, and directory management were successfully executed and their outputs were verified.

🔷 VIVA QUESTIONS (DETAILED ANSWERS)

----

1. What is Linux?

Linux is an open-source, multitasking operating system based on UNIX.

2. Difference between rm and rmdir?

- `rm` deletes **files**
- `rmdir` deletes **empty directories**

3. Purpose of wc command?

To count lines, words, characters in a file.

4. What does pwd show?

The full path of the current directory.

5. What is grep used for?

To search for matching text inside files.

6. What is ls -a?

Lists **all files**, including hidden ones.

7. Difference between cat and cp?

- `cat` → displays contents
- `cp` → copies contents

🌟 PRACTICAL NO. 3

Execute Various Linux Commands for Process Control, Communication, and Protection Management

---

✅ AIM

To study and execute Linux commands related to **process control**, **inter-process communication (I/O redirection and pipes)**, and **file protection/permissions**.

🌐 PART I — PROCESS CONTROL COMMANDS

Linux is a multitasking OS, so multiple processes run simultaneously.
The following commands help monitor, manage, and control these processes.

⭑ 1. fork() (Concept Explanation Only)

`fork()` is a system call used in C programs to create a new process called *child process*.
This is not a terminal command — you will use it in C programs (Practical 4).

⭑ 2. getpid() (Concept Explanation Only)

`getpid()` returns the Process ID (PID) of the running process.

⭑ 3. ps — Process Status

Syntax

```
ps
ps -e
ps -aux
```

Purpose:
Shows list of currently running processes.

⭑ 4. sleep — Pause Execution

Syntax

```
sleep 5
```

Pauses for 5 seconds.
Useful in shell scripts.

⭐ 5. kill — Terminate Process

Syntax

```
kill PID
kill -9 PID
```

Purpose:
Terminates a process given its PID.

`kill -9` = force kill.

🔵 PART II — COMMUNICATION COMMANDS

Linux allows communication between processes using **I/O redirection** and **pipes**.

⭐ 1. Input/Output Redirection

✔ Output Redirection

```
command > file
```

Sends output **to a file**, overwriting it.

Example:

```
ls > out.txt
```

✔ Append Output

```
command >> file
```

✔ Input Redirection

```
command < file
```

Example:

```
wc < file.txt
```

⭑ 2. Pipe (|)

✔ Connects output of one command to input of another

Syntax

```
command1 | command2
```

✔ Example:

```
ls | grep txt
```

This shows only *.txt* files.

Another example:

```
cat file.txt | wc
```

🟦 PART III — PROTECTION MANAGEMENT

Linux uses **permissions** to protect files.

Permissions are shown as:

```
r = read
w = write
x = execute
```

For:

- user (owner)
- group
- others

Example:

```
-rw-r--r--
```

⭐ 1. chmod — Change Permissions

Syntax

```
chmod 755 file
chmod u+x file
```

Numeric Values:

- r = 4
- w = 2
- x = 1

Example:

```
chmod 777 file
```

Gives all permissions to everyone.

⭐ 2. chown — Change Ownership

Syntax

```
chown user file
```

⭐ 3. chgrp — Change Group Ownership

Syntax

```
chgrp group file
```

🔵 SAMPLE OUTPUTS (USE THESE IN YOUR JOURNAL)

✔ Process List

```
$ ps
PID  TTY   TIME CMD
1234 pts/0 00:00 bash
1289 pts/0 00:00 ps
```

✓ Sleep Example

```
$ sleep 5
(wait for 5 seconds)
```

✓ Killing a Process

```
$ kill 1234
```

✓ Output Redirection

```
$ ls > list.txt
```

✓ Pipe Example

```
$ ls | grep txt
notes.txt
data.txt
```

✓ chmod Example

```
$ chmod 755 a.sh
```

✓ chown Example

```
$ chown student file1
```

✓ chgrp Example

```
$ chgrp teachers file1
```

🔵 RESULT

Commands related to process control (ps, kill, sleep), communication (pipes and redirection), and protection (chmod, chown, chgrp) were executed successfully and their behavior was verified.

🔵 VIVA QUESTIONS (DETAILED, HIGH-SCORING)

----

1. What does ps do?

Shows all currently running processes.

2. What is a PID?

PID = Process ID.
Unique number assigned to every process.

3. Difference between kill and kill -9?

- kill sends normal termination signal
- kill -9 forcefully kills a process

4. What is a pipe?

A pipe | sends output of one command to another.

5. What is the purpose of > and >> ?

- > overwrite output to file
- >> append output to file

6. Explain chmod 755.

- User: rwx
- Group: r-x
- Others: r-x

7. What is chown used for?

To change owner of a file.

⭐ PRACTICAL NO. 4

Programs using fork() and exec(): Same program, different program, and parent waiting for child

---

✅ AIM

To write C programs demonstrating:

1. Parent and child executing the **same program**
2. Parent and child executing **different code**
3. Parent waiting for child before terminating

Using **fork()**, **exec()**, and **wait()** system calls.

✅ THEORY (DETAILED + EXAM-READY)

fork()

- A system call that **creates a new process**.
- The new process is called the **child process**.
- After fork(), **two processes run the same code**.
- fork() returns:
  - **0** → inside child process
  - **positive PID** → inside parent
  - **negative** → fork failed

getpid()

- Returns PID of running process.

exec()

- Replaces current program with a **new executable/program**.
- After exec(), the new program entirely replaces current code.

wait()

- Parent process waits until child finishes execution.
- Prevents zombie processes.

🔵 PROGRAM 1: Same Program, Same Code

✔ Explanation

Both parent and child execute the same statements after fork().

★ Code: same_code.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();     // create new process

    if (pid < 0) {
        printf("Fork failed\n");
        return 1;
    }

    printf("Process executing same code. PID = %d\n", getpid());
    return 0;
}
```

★ What to write as OUTPUT

```
Process executing same code. PID = 1203
Process executing same code. PID = 1204
```

(One line for parent, one for child)

🔵 PROGRAM 2: Same Program, Different Code

✔ Explanation

Parent and child run **different portions** of the same program using if-else based on pid value.

★ Code: different_code.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        printf("Fork failed\n");
    }
    else if (pid == 0) {
        printf("Child Process. PID = %d\n", getpid());
        printf("Child executing its own code.\n");
    }
    else {
        printf("Parent Process. PID = %d\n", getpid());
        printf("Parent executing parent code.\n");
    }

    return 0;
}
```

★ OUTPUT (Write this in journal)

```
Child Process. PID = 4560
Child executing its own code.

Parent Process. PID = 4559
Parent executing parent code.
```

🔵 PROGRAM 3: Parent Waits for Child

✔ Explanation

- Child executes first.
- Parent waits using `wait(NULL)` until child finishes.
- Then parent continues.

★ Code: parent_waits.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        printf("Fork failed\n");
    }
    else if (pid == 0) {
        printf("Child Process Running. PID = %d\n", getpid());
        sleep(2);    // simulate some work
        printf("Child Completed.\n");
    }
    else {
        wait(NULL);     // parent waits for child
        printf("Parent Process. Child finished. Parent continues.\n");
    }

    return 0;
}
```

★ OUTPUT for Journal

```
Child Process Running. PID = 5010
Child Completed.
Parent Process. Child finished. Parent continues.
```

🔵 RESULT

Programs using fork(), exec(), getpid(), and wait() were successfully executed.
The behavior of parent and child processes was studied in detail.

🔵 VIVA QUESTIONS (MOST COMMON QUESTIONS)

(Learn these to get full marks)

1. What is fork()?

Creates a new child process.
Both parent and child run the same program.

2. What does fork() return?

- 0 → child
- positive PID → parent
- -1 → error

3. What is exec()?

Replaces current process image with a new program.

4. Why do we use wait()?

To ensure parent waits until child finishes.

5. What is a zombie process?

A process that finished but parent didn’t call wait().

6. What is getpid()?

Returns process ID of current process.

★ PRACTICAL NO. 5

Report Behaviour of Linux Kernel: Kernel Version, CPU Type & CPU Information

---

✓ AIM

To write a C program that displays Linux kernel information, kernel version, CPU architecture, and CPU hardware details using system commands.

✓ THEORY (DETAILED)

In Linux, system-level information such as:

- Kernel version
- Kernel name
- Processor type
- CPU architecture
- Hardware information

is stored inside the kernel and exposed through special commands.

The commands used are:

1. uname

- Displays system information.
- Common options:
  - `uname -a` → All kernel/system details
  - `uname -r` → Kernel release
  - `uname -s` → Kernel name

2. lscpu

- Shows **CPU information**, such as:
  - Number of CPUs
  - Architecture (x86_64, ARM)
  - Model name
  - Core count
  - Threads per core

3. /proc filesystem

Linux stores system information in virtual files like:

- `/proc/cpuinfo` → Detailed CPU info
- `/proc/version` → Kernel version

These can be displayed using `cat`.

🟦 PROGRAM (C PROGRAM TO SHOW KERNEL + CPU INFO)

★ Code: kernel_cpu_info.c

```
#include <stdlib.h>
#include <stdio.h>

int main() {

    printf("---- Kernel Information ----\n");
    system("uname -a");            // full kernel details

    printf("\n---- Kernel Version ----\n");
```

```
    system("uname -r");           // release version only

    printf("\n---- CPU Details (lscpu) ----\n");
    system("lscpu");              // CPU architecture + model

    printf("\n---- CPU Info (/proc/cpuinfo) ----\n");
    system("cat /proc/cpuinfo");  // detailed CPU hardware info

    return 0;
}
```

🔵 SAMPLE OUTPUT (Write this in your journal)

Kernel Information:

```
Linux ubuntu 5.15.0-84-generic #93-Ubuntu SMP x86_64 GNU/Linux
```

Kernel Version:

```
5.15.0-84-generic
```

CPU Details:

```
Architecture:            x86_64
CPU(s):                  4
Model name:              Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Thread(s) per core:      2
Core(s) per socket:      2
```

CPU Hardware Information:

```
processor    : 0
vendor_id    : GenuineIntel
model name   : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
cpu MHz      : 1800.000
cache size   : 6144 KB
```

(Your output may differ, that's normal.)

🌐 RESULT

The Linux kernel version, system information, CPU architecture, and detailed CPU hardware information were successfully displayed using system commands executed from a C program.

🌐 VIVA QUESTIONS (High-Scoring Short Answers)

---

1. What is the Linux kernel?

Core of the operating system that manages hardware and system resources.

2. What does `uname -a` show?

Displays complete system information including kernel name, version, release, machine type, and architecture.

3. What is `/proc/cpuinfo`?

A virtual file containing detailed CPU hardware information.

4. What is the purpose of `lscpu`?

Shows CPU architecture, core count, threads, and processor model.

5. What is kernel version?

The specific release of the Linux kernel running on the system.

6. Where does Linux store system info?

In the `/proc` virtual filesystem.

PRACTICAL NO. 6

Report Linux Kernel Behaviour: Memory Information (Configured, Free & Used Memory)

AIM

To write a C program that displays **total memory**, **free memory**, **used memory**, and **memory statistics** of the Linux system using system commands.

THEORY (DETAILED + SHORT ENOUGH FOR VIVA)

Linux stores real-time memory statistics in special files and provides commands to view them:

1. `/proc/meminfo`

A virtual file that contains detailed information about system memory including:

- Total RAM
- Free RAM
- Buffers
- Cached memory
- Swap memory

2. `free -h`

A Linux command used to show:

- Total memory
- Used memory
- Free memory
- Buffer/Cache
- Swap memory
  The `-h` option displays values in human-readable units (MB/GB).

3. `cat /proc/meminfo`

Shows detailed RAM usage from the kernel.

4. `vmstat`

Displays memory, process, and CPU usage statistics (optional).

🔵 PROGRAM (C Program to Display Memory Information)

★ Code: memory_info.c

```c
#include <stdlib.h>
#include <stdio.h>

int main() {

    printf("---- Memory Information (free -h) ----\n");
    system("free -h");              // shows total, used, free memory

    printf("\n---- Detailed Memory Info (/proc/meminfo) ----\n");
    system("cat /proc/meminfo");      // detailed memory statistics

    return 0;
}
```

🔵 SAMPLE OUTPUT (Write in Journal)

(Your output may be different — that's OK)

From `free -h`:

```
              total        used        free      shared  buff/cache
available
Mem:            7.7G        3.2G        1.8G        250M        2.7G
4.0G
Swap:           2.0G         75M        1.9G
```

From `/proc/meminfo`:

```
MemTotal:        8071236 kB
MemFree:         1856232 kB
MemAvailable:    4095320 kB
Buffers:          125920 kB
Cached:          2731228 kB
SwapTotal:       2097148 kB
SwapFree:        1998236 kB
```

🔵 RESULT

The memory information of the Linux kernel, including total memory, free memory, used memory, buffer, cache, and swap memory, was successfully displayed using system commands from a C program.

🔵 VIVA QUESTIONS (MOST IMPORTANT)

---

1. What is `/proc/meminfo`?

A virtual file containing detailed information about system memory.

2. What does `free -h` show?

Total memory, used memory, free memory, buffer/cache memory, and swap memory.

3. What is swap memory?

A portion of the hard disk used as virtual RAM when physical RAM is full.

4. What is buffer/cache memory?

Memory used to speed up disk access by caching frequently used data.

5. Why is `/proc` called a virtual filesystem?

Because it doesn't exist on disk — kernel generates it dynamically in RAM.

6. What command shows memory in human readable form?

```
free -h
```

✴ PRACTICAL NO. 7

Write a C Program to Copy Files Using System Calls

✓ AIM

To write a C program that copies the contents of one file into another using **Linux system calls** (`open`, `read`, `write`, `close`), instead of standard I/O functions like `fopen`.

✓ THEORY (DETAILED + EASY FOR VIVA)

Linux provides **low-level system calls** for file handling. These calls interact directly with the kernel.

System Calls Used:

1. open()

Opens a file and returns a file descriptor (integer).

Syntax:

```
int open(const char *pathname, int flags, mode_t mode);
```

2. read()

Reads data from a file.

Syntax:

```
ssize_t read(int fd, void *buffer, size_t count);
```

3. write()

Writes data into a file.

Syntax:

```
ssize_t write(int fd, const void *buffer, size_t count);
```

4. close()

Closes an opened file descriptor.

🔵 WHY SYSTEM CALLS?

- Faster and closer to hardware
- Used in OS-level programming
- No buffering (unlike stdio)
- Required in OS and system-level practicals

🔵 PROGRAM: File Copy Using System Calls

⭑ Code: filecopy.c

```
#include <fcntl.h>      // for open
#include <unistd.h>     // for read, write, close
#include <stdio.h>

int main() {
    int source, dest;
    char buffer[1024];
    ssize_t bytes;

    // open source file in read-only mode
    source = open("input.txt", O_RDONLY);
    if (source < 0) {
        perror("Error opening source file");
        return 1;
    }

    // open destination file, create if not exists
    dest = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dest < 0) {
        perror("Error opening destination file");
        return 1;
    }

    // read from source and write to destination
    while ((bytes = read(source, buffer, sizeof(buffer))) > 0) {
        write(dest, buffer, bytes);
    }

    // close files
    close(source);
```

```
    close(dest);

    printf("File copied successfully.\n");
    return 0;
}
```

ALGORITHM

---

1. Start the program.
2. Open the source file using `open()` in read-only mode.
3. Create/open destination file in write mode using `open()`.
4. Create a buffer to store data temporarily.
5. Use a loop to:
   - Read from source using `read()`
   - Write into destination using `write()`
6. Continue until the end of the file.
7. Close both files using `close()`.
8. Display success message.
9. End program.

SAMPLE OUTPUT (WRITE THIS IN JOURNAL)

Terminal:

```
$ gcc filecopy.c -o filecopy
$ ./filecopy
File copied successfully.
```

If `input.txt` contained:

```
Operating Systems Lab
```

Then `output.txt` will contain:

```
Operating Systems Lab
```

🔵 RESULT

The file was successfully copied from source to destination using low-level Linux system calls (`open`, `read`, `write`, `close`). The behaviour of system calls in file handling was observed.

🔵 VIVA QUESTIONS (Most Important)

---

1. What are system calls?

Functions that provide an interface between user programs and the operating system kernel.

2. Why do we use `open()` instead of `fopen()`?

`open()` is a system call → interacts directly with the kernel.

3. What does `O_CREAT` do?

Creates a new file if it does not exist.

4. What is a file descriptor?

An integer value returned by `open()` used to access files.

5. Why do we use a buffer?

To efficiently read/write chunks of data.

6. What is the purpose of `close()`?

Releases the file descriptor and ensures data is saved correctly.

7. What happens if the source file does not exist?

`open()` returns -1 and `perror()` displays an error.

🌟 PRACTICAL NO. 8

Implement First Come First Serve (FCFS) CPU Scheduling Algorithm

---

✓ AIM

To write a C program to implement the **First Come First Serve (FCFS)** CPU scheduling algorithm and calculate **Waiting Time (WT)** and **Turnaround Time (TAT)** for each process.

✓ THEORY (DETAILED + EXAM-READY)

★ What is FCFS?

- FCFS stands for **First Come First Serve**.
- It is the **simplest CPU scheduling algorithm**.
- Processes are executed **in the order they arrive**.
- **Non-preemptive**: once a process starts, it runs till completion.

★ Terminology

1. Burst Time (BT)

Amount of CPU time required by the process.

2. Waiting Time (WT)

Time a process spends **waiting in the ready queue**.
Formula:

```
WT[i] = Sum of burst times of all previous processes
```

3. Turnaround Time (TAT)

Total time from arrival to completion.
Formula:

```
TAT[i] = WT[i] + BT[i]
```

★ Characteristics of FCFS:

- Simple to understand and implement
- Not optimal
- Causes **Convoy Effect**: short jobs wait behind long jobs

ALGORITHM

----

1. Start
2. Input the number of processes
3. Enter **burst time** of each process
4. Set waiting time of first process = 0
5. For each process:
    - WT[i] = WT[i–1] + BT[i–1]
    - TAT[i] = WT[i] + BT[i]
6. Display WT and TAT for all processes
7. End

C PROGRAM – FCFS CPU SCHEDULING

★ Code: fcfs.c

```
#include <stdio.h>

int main() {
    int n, i;
    int bt[20], wt[20], tat[20];

    printf("Enter number of processes: ");
    scanf("%d", &n);

    printf("Enter burst times:\n");
    for (i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &bt[i]);
    }
```

```c
    wt[0] = 0;   // first process has no waiting time

    // calculate waiting time
    for (i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
    }

    // calculate turnaround time
    for (i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
    }

    printf("\nProcess\tBT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\n", i + 1, bt[i], wt[i], tat[i]);
    }

    return 0;
}
```

🔵 SAMPLE OUTPUT (WRITE IN JOURNAL)

```
Enter number of processes: 3
Enter burst times:
P1: 5
P2: 3
P3: 8

Process BT   WT   TAT
P1      5    0    5
P2      3    5    8
P3      8    8    16
```

🔵 GANTT CHART (DRAW IN JOURNAL)

```
|  P1   |  P2   |    P3    |
0       5       8          16
```

🌐 RESULT

The FCFS scheduling algorithm was successfully implemented in C. Waiting time and turnaround time for each process were calculated and represented using a Gantt chart.

🌐 VIVA QUESTIONS (MOST IMPORTANT)

----

1. What is FCFS?

Scheduling algorithm where processes are executed in order of arrival.

2. Is FCFS preemptive or non-preemptive?

Non-preemptive.

3. What is convoy effect?

Short jobs waiting behind long jobs in FCFS.

4. Formula for waiting time?

```
WT[i] = WT[i-1] + BT[i-1]
```

5. Formula for turnaround time?

```
TAT = WT + BT
```

6. Drawbacks of FCFS?

High waiting time and poor performance for short jobs.

7. Can FCFS cause starvation?

No.

✪ PRACTICAL NO. 9

Implement Shortest Job First (SJF) CPU Scheduling Algorithm

✓ AIM

To write a C program that implements the **Shortest Job First (SJF)** CPU scheduling algorithm (non-preemptive) and calculates **Waiting Time (WT)** and **Turnaround Time (TAT)**.

✓ THEORY (DETAILED + SIMPLE)

★ What is SJF?

Shortest Job First (SJF) is a CPU scheduling algorithm where the process with the **shortest burst time** is executed first.

★ Type:

- **Non-preemptive**: Once a process starts, it runs till completion.

★ Why SJF is good?

✓ Gives **minimum average waiting time**
✓ Very efficient for batch processing

★ Why it is not used always?

✗ Exact burst time is not always known
✗ Can cause **starvation** of long processes

☆ IMPORTANT FORMULAS

1. Waiting Time (WT)

```
WT[i] = sum of burst times of all previous processes
```

2. Turnaround Time (TAT)

```
TAT[i] = WT[i] + BT[i]
```

🔵 ALGORITHM

----

1. Start
2. Take number of processes (n)
3. Input burst time for all processes
4. Sort processes in **ascending order of burst time**
5. Compute waiting time:
   - WT[0] = 0
   - WT[i] = WT[i−1] + BT[i−1]
6. Compute turnaround time:
   - TAT[i] = WT[i] + BT[i]
7. Display all results
8. End

🔵 C PROGRAM – NON-PREEMPTIVE SJF

★ Code: sjf.c

```
#include <stdio.h>

int main() {
    int n, i, j, temp;
    int bt[20], wt[20], tat[20], p[20];

    printf("Enter number of processes: ");
    scanf("%d", &n);

    printf("Enter burst times:\n");
    for(i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &bt[i]);
        p[i] = i + 1;   // store process ID
    }
```

```
// Sort by burst time
    for(i = 0; i < n - 1; i++) {
        for(j = i + 1; j < n; j++) {
            if(bt[i] > bt[j]) {
                // swap burst times
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                // swap process IDs
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }

    wt[0] = 0;

    // calculate waiting time
    for(i = 1; i < n; i++)
        wt[i] = wt[i - 1] + bt[i - 1];

    // calculate turnaround time
    for(i = 0; i < n; i++)
        tat[i] = wt[i] + bt[i];

    printf("\nProcess\tBT\tWT\tTAT\n");
    for(i = 0; i < n; i++)
        printf("P%d\t%d\t%d\t%d\n", p[i], bt[i], wt[i], tat[i]);

    return 0;
}
```

🔵 SAMPLE OUTPUT (Write This in Your Journal)

```
Enter number of processes: 3
Enter burst times:
P1: 6
P2: 2
P3: 8

Process  BT   WT   TAT
P2       2    0    2
P1       6    2    8
P3       8    8    16
```

🔵 GANTT CHART (Draw in Record Book)

```
|    P2    |      P1      |       P3       |
0          2              8               16
```

🔵 RESULT

The Shortest Job First (SJF) scheduling algorithm was successfully implemented.
Waiting time and turnaround time for each process were calculated and the Gantt chart was drawn.

🔵 VIVA QUESTIONS (HIGH-SCORING ANSWERS)

---

1. What is SJF?

CPU scheduling algorithm where the process with the shortest burst time is executed first.

2. Is SJF preemptive?

This program: **Non-preemptive**
There is also a preemptive version called **SRTF**.

3. Why is SJF optimal?

Because it gives minimum average waiting time.

4. What is the drawback of SJF?

Needs accurate burst time → not always possible.

5. What is starvation?

Long processes may wait indefinitely.

6. Formula for waiting time?

```
WT = sum of previous burst times
```

7. Formula for turnaround time?

```
TAT = WT + BT
```

★ PRACTICAL NO. 10

Implement Non-Preemptive Priority-Based CPU Scheduling Algorithm

---

✓ AIM

To write a C program to implement **Non-Preemptive Priority Scheduling**, calculating **Waiting Time (WT)** and **Turnaround Time (TAT)**.

✓ THEORY (DETAILED + SIMPLE)

★ What is Priority Scheduling?

Each process is assigned a **priority number**, and the CPU is allocated to the **highest priority process**.

★ Types of Priority Scheduling:

- **Preemptive**: High priority process interrupts running process
- **Non-preemptive**: Currently running process finishes before next begins

★ We are implementing:

✓ Non-Preemptive Priority Scheduling

⭐ Priority Rule

☞ **Lower number = Higher priority** (common convention in OS labs)

⭐ Important Terms

1. Burst Time (BT)

Time required by process to finish.

2. Waiting Time (WT)

Time spent waiting in ready queue.
Formula:

```
WT[i] = WT[i-1] + BT[i-1]
```

3. Turnaround Time (TAT)

Total time from arrival to completion.
Formula:

```
TAT[i] = WT[i] + BT[i]
```

🌐 ALGORITHM

----

1. Read number of processes
2. Input burst time and priority for each process
3. Sort processes by **priority**
4. Set WT[0] = 0
5. For each process:
    - WT[i] = WT[i−1] + BT[i−1]
6. Compute TAT for each process
7. Display results

🔵 C PROGRAM – Non-Preemptive Priority Scheduling

★ Code: priority.c

```c
#include <stdio.h>

int main() {
    int n, i, j, temp;
    int bt[20], wt[20], tat[20], priority[20], p[20];

    printf("Enter number of processes: ");
    scanf("%d", &n);

    printf("Enter burst time and priority for each process:\n");
    for(i = 0; i < n; i++) {
        printf("P%d Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
        printf("P%d Priority: ", i + 1);
        scanf("%d", &priority[i]);
        p[i] = i + 1;  // Process ID
    }

    // Sort by priority (lower number = higher priority)
    for(i = 0; i < n - 1; i++) {
        for(j = i + 1; j < n; j++) {
            if(priority[i] > priority[j]) {
                // Swap priority
                temp = priority[i];
                priority[i] = priority[j];
                priority[j] = temp;

                // Swap burst times
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                // Swap process IDs
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }

    wt[0] = 0;

    // Calculate waiting time
    for(i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
    }

    // Calculate turnaround time
    for(i = 0; i < n; i++) {
```

```
        tat[i] = wt[i] + bt[i];
    }

    printf("\nProcess\tBT\tPriority\tWT\tTAT\n");
    for(i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t\t%d\t%d\n", p[i], bt[i], priority[i], wt[i],
tat[i]);
    }

    return 0;
}
```

🔵 SAMPLE OUTPUT (Write in Journal)

```
Enter number of processes: 3

P1 Burst Time: 5
P1 Priority: 2

P2 Burst Time: 3
P2 Priority: 1

P3 Burst Time: 8
P3 Priority: 3

Process BT Priority WT TAT
P2      3     1      0   3
P1      5     2      3   8
P3      8     3      8   16
```

🔵 GANTT CHART (Draw This)

```
|   P2   |      P1      |       P3        |
0        3              8                 16
```

🔵 RESULT

The non-preemptive priority scheduling algorithm was implemented successfully. Waiting time and turnaround time were calculated for each process, and a Gantt chart was drawn.

🔵 VIVA QUESTIONS (Highly Important)

----

1. What is priority scheduling?

Scheduling based on priority assigned to processes.

2. Which priority is executed first?

Lower number = higher priority (in our program).

3. Is this version preemptive or non-preemptive?

Non-preemptive.

4. What is starvation?

Low priority processes may never get CPU.

5. How to reduce starvation?

By using **Aging** (increase priority over time).

6. Formula for WT and TAT?

- $WT[i] = WT[i-1] + BT[i-1]$
- $TAT[i] = WT[i] + BT[i]$

7. Can priority scheduling be preemptive?

Yes — if a higher priority process arrives, it can interrupt running process.

🟊 PRACTICAL NO. 11

Write a Program to Calculate Sum of n Numbers Using Pthreads

✓ AIM

To write a C program using **POSIX Threads (Pthreads)** to calculate the **sum of n numbers** by performing the operation inside a thread.

✓ THEORY (DETAILED + EXAM-READY)

★ What are Threads?

Threads are lightweight processes that run inside a process.
In Linux, threads are created using **Pthreads library (`pthread.h`)**.

★ Advantages of Threads

- Faster context switching
- Parallel execution
- Better CPU utilization
- Used in multiprocessor systems

★ Important Pthread Functions

Function	Description
`pthread_create()`	Creates a new thread
`pthread_join()`	Waits for a thread to finish
`pthread_exit()`	Terminates a thread
`pthread_t`	Thread identifier

★ Program Logic

- Create an array of numbers
- Create a thread to compute the sum
- Thread calculates sum inside its function
- Main program waits using `pthread_join()`
- Print final result

ALGORITHM

----

1. Start the program
2. Read value of **n** (number of elements)
3. Read **n numbers** into an array
4. Create a thread using `pthread_create()`
5. Thread function:
    - Loop through array
    - Add each element to sum
6. Main function waits for thread using `pthread_join()`
7. Print the result
8. End the program

C PROGRAM – Sum of n Numbers Using Pthreads

★ Code: pthread_sum.c

```
#include <stdio.h>
#include <pthread.h>

int a[100], n;
int sum = 0;

void* find_sum(void* arg) {
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    return NULL;
}

int main() {
    pthread_t t1;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d numbers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
```

```
    }

    // create thread
    pthread_create(&t1, NULL, find_sum, NULL);

    // wait for thread to finish
    pthread_join(t1, NULL);

    printf("Sum = %d\n", sum);

    return 0;
}
```

COMPILATION COMMAND

Use this (required for thread programs):

```
gcc pthread_sum.c -o pthread_sum -lpthread
./pthread_sum
```

SAMPLE OUTPUT

```
Enter number of elements: 5
Enter 5 numbers:
2 4 6 8 10
Sum = 30
```

RESULT

The program to calculate the sum of n numbers using Pthreads was successfully implemented. Thread creation and synchronization using `pthread_create()` and `pthread_join()` were demonstrated.

🔵 VIVA QUESTIONS (VERY IMPORTANT)

---

1. What is a thread?

A lightweight subprocess that executes inside a process.

2. What is Pthreads?

POSIX standard thread library in Linux.

3. Why use threads?

Faster execution, parallelism, and efficient CPU usage.

4. Difference between process and thread?

- Process → independent memory
- Thread → shares memory with main process

5. What does pthread_create() do?

Creates a new thread.

6. Why use pthread_join()?

To ensure the main program waits until the thread finishes.

7. Can multiple threads access shared variables?

Yes (but may require synchronization in complex programs).

🌟 PRACTICAL NO. 12

Implement First-Fit, Best-Fit & Worst-Fit Memory Allocation Strategies

---

✅ AIM

To write a C program to implement **Three memory allocation techniques**:

1. **First Fit**
2. **Best Fit**
3. **Worst Fit**

For allocating memory blocks to processes.

✅ THEORY (DETAILED + SIMPLE)

Memory allocation is part of **Contiguous Memory Management** in OS.

When processes request memory, OS allocates suitable blocks using strategies:

🔵 1. FIRST FIT

- Scans memory from **beginning**
- Allocates **first block** large enough
- Fastest technique
- More fragmentation

Example:
Blocks: 100, 300, 200, 400
Process: 180 → allocated to block 300 (first suitable)

🔵 2. BEST FIT

- Allocates **smallest block** that fits the process
- Minimizes leftover space
- Slow because it scans entire list
- Causes **small unusable holes**

Example:
Blocks: 100, 300, 200, 400
Process: 180 → allocated to block 200 (best smallest fit)

🔵 3. WORST FIT

- Allocates **largest available block**
- Leaves large leftover fragment
- Good for reducing fragmentation sometimes

Example:
Blocks: 100, 300, 200, 400
Process: 180 → allocated to block 400 (largest)

🔵 ALGORITHM

For each strategy:

1. Input no. of memory blocks
2. Input block sizes
3. Input no. of processes
4. Input process sizes
5. Apply allocation rule:
    - First Fit → first block that fits
    - Best Fit → smallest possible block
    - Worst Fit → largest possible block
6. Update remaining block size
7. Display allocation table

🔵 C PROGRAM – First Fit, Best Fit, Worst Fit

★ Code: memory_allocation.c

```
#include <stdio.h>

void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }

    printf("\nFirst Fit Allocation:\n");
    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1)
            printf("P%d\t%d\tB%d\n", i+1, processSize[i], allocation[i]+1);
        else
            printf("P%d\t%d\tNot Allocated\n", i+1, processSize[i]);
    }
}

void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        int bestIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx])
                    bestIdx = j;
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }

    printf("\nBest Fit Allocation:\n");
    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1)
            printf("P%d\t%d\tB%d\n", i+1, processSize[i], allocation[i]+1);
```

```
        else
            printf("P%d\t%d\tNot Allocated\n", i+1, processSize[i]);
    }
}

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx])
                    worstIdx = j;
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }

    printf("\nWorst Fit Allocation:\n");
    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1)
            printf("P%d\t%d\tB%d\n", i+1, processSize[i], allocation[i]+1);
        else
            printf("P%d\t%d\tNot Allocated\n", i+1, processSize[i]);
    }
}

int main() {
    int m, n;

    printf("Enter number of blocks: ");
    scanf("%d", &m);
    int blockSize1[m], blockSize2[m], blockSize3[m];

    printf("Enter block sizes:\n");
    for (int i = 0; i < m; i++) {
        scanf("%d", &blockSize1[i]);
        blockSize2[i] = blockSize1[i];
        blockSize3[i] = blockSize1[i];
    }

    printf("Enter number of processes: ");
    scanf("%d", &n);
    int processSize[n];
    printf("Enter process sizes:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &processSize[i]);

    firstFit(blockSize1, m, processSize, n);
    bestFit(blockSize2, m, processSize, n);
    worstFit(blockSize3, m, processSize, n);

    return 0;
}
```

SAMPLE OUTPUT (Write in Journal)

Input:

```
Blocks: 100 500 200 300 600
Processes: 212 417 112 426
```

First Fit:

```
P1 -> B2
P2 -> B5
P3 -> B1
P4 -> B4
```

Best Fit:

```
P1 -> B4
P2 -> B5
P3 -> B1
P4 -> Not Allocated
```

Worst Fit:

```
P1 -> B5
P2 -> B5
P3 -> B2
P4 -> B2
```

RESULT

First Fit, Best Fit, and Worst Fit memory allocation strategies were successfully implemented.
The memory allocation behavior and fragmentation for each technique were observed.

🔵 VIVA QUESTIONS (MOST IMPORTANT)

---

1. Which technique is fastest?

First Fit.

2. Which minimizes leftover space?

Best Fit.

3. Which uses the largest block first?

Worst Fit.

4. What is fragmentation?

Wasted memory due to allocation decisions.

5. Types of fragmentation?

- **Internal** (inside block)
- **External** (between blocks)

6. Which suffers most from external fragmentation?

First Fit.

7. Which method may leave very small unusable holes?

Best Fit.

8. Which method can reduce fragmentation sometimes?

Worst Fit.
