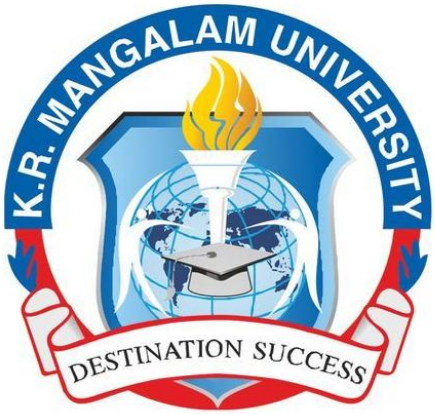
K.R. Mangalam University

School of Engineering and Technology



Operating System Lab File

Course Code: ENCS351

Session: 2025-2026

Submitted by

Name: Prerna Kandpal

Roll No.: 2301010251

Program: BTech CSE-D

Submitted To

Dr. Satinder Pal Singh

INDEX

Sr No.	Assignments	Page No.
1	1.1 Process Creation Utility. 1.2 Command Execution Using exec 1.3 Zombie & Orphan Processes. 1.4 Inspecting Process Info from /proc. 1.5 Task 5: Process Prioritization.	3 - 9
2	2.1 Write a Python script to simulate a basic system startup sequence. 2.2 Use the multiprocessing module to create at least two child processes that perform dummy tasks. 2.3 Implement proper logging to track process start and end times. 2.4 Generate a log file (process _log.txt) to reflect system-like behaviour. 2.5 Submit the Python script and log file along with a short report explaining your implementation.	10 - 13
3	3.1 CPU Scheduling with Gantt Chart. 3.2 Sequential File Allocation. 3.3 Indexed File Allocation. 3.4 Contiguous Memory Allocation. 3.5 MFT & MVT Memory Management.	14 - 23
4	4.1 Batch Processing Simulation (Python). 4.2 System Startup and Logging. 4.3 System Calls and IPC (Python - fork, exec, pipe). 4.4 VM Detection and Shell Interaction. 4.5 CPU Scheduling Algorithms.	24 - 29

Lab Experiment Sheet-1

Experiment Title: Process Creation and Management Using Python OS Module

Experiment Objectives:

In this assignment, students will simulate Linux process management operations using Python. The experiment focuses on replicating the behaviors of fork(), exec(), and process state inspections using the os and subprocess modules in Python. It provides an understanding of process creation, child-parent relationship, and zombie/orphan process scenarios.

Learning Outcomes:

- Understand the lifecycle of processes in Linux.
- Create child processes and execute system commands using Python.
- Simulate zombie and orphan processes.
- Inspect running processes using /proc.
- Demonstrate priority setting via nice values.

Concepts Used:

- os.fork(), os.getpid(), os.getppid()
- os._exit(), os.wait(), os.nice()
- subprocess.run(), os.execvp()
- Reading /proc/[pid]/status, /exe, and /fd

Task 1: Process Creation Utility

Write a Python program that creates N child processes using os.fork(). Each child prints:
- Its PID
- Its Parent PID
- A custom message

The parent should wait for all children using os.wait().

Process_management.py

```
import os
import sys
def main():
        try:
                N = int(input("Enter thr number of child processes: "))
        except ValueError:
                print("Please enter a valid integer.")
                sys.exit(1)
        print(f"Parent process PID: {os.getpid()} creating {N} child processes...\n")
        for i in range(N):
                pid = os.fork()
                if pid == 0:
```

```
                print(f"Child {i+1}: PID = {os.getpid()}, Parent PID = {os.getppid()},
Message = Hello from chld {i+1}")
                os._exit(0)
            else:
                continue
        for i in range(N):
            pid, status = os.wait()
            print(f"Parent: Child with PID = {pid} finished with status{status}")
if __name__ == "__main__":
        main()
```

Expected Output :

```
Enter the number of child processes: 3
Parent process PID: 4821 creating 3 child processes...

Child 1: PID = 4822, Parent PID = 4821, Message = Hello from chld 1
Child 2: PID = 4823, Parent PID = 4821, Message = Hello from chld 2
Child 3: PID = 4824, Parent PID = 4821, Message = Hello from chld 3

Parent: Child with PID = 4822 finished with status0
Parent: Child with PID = 4823 finished with status0
Parent: Child with PID = 4824 finished with status0
```

Task 2: Command Execution Using exec()

Modify Task 1 so that each child process executes a Linux command (ls, date, ps, etc.) using os.execvp() or subprocess.run().

Process_management.py

```
import os
import sys

def create_children_with_exec(n, command):
    children_pids = []

    for i in range(n):
        try:
            pid = os.fork()
        except OSError as e:
            print(f"Fork failed: {e}", file=sys.stderr)
            sys.exit(1)

        if pid == 0:
            print(f"\n[Child {i+1}] PID={os.getpid()}, Parent PID={os.getppid()}, executing
command: {' '.join(command)}\n")
            try:
```

```
            os.execvp(command[0], command)
        except FileNotFoundError:
            print(f"Command not found: {command[0]}", file=sys.stderr)
            os._exit(1)
    else:
        children_pids.append(pid)

for _ in children_pids:
    pid, status = os.wait()
    if os.WIFEXITED(status):
        print(f"[Parent] Child PID={pid} exited with status {os.WEXITSTATUS(status)}")
    else:
        print(f"[Parent] Child PID={pid} terminated abnormally")

def main():
    try:
        n = int(input("Enter the number of child processes: "))
        if n <= 0:
            print("Number of processes must be positive.")
            return
    except ValueError:
        print("Invalid input. Please enter an integer.")
        return
    command = input("Enter the Linux command to execute (e.g., 'ls -l'): ").split()

    print(f"\n[Parent] PID={os.getpid()} creating {n} children to run: {' '.join(command)}\n")
    create_children_with_exec(n, command)

if __name__ == "__main__":
    main()
```

Expected Output :

```
Enter the number of child processes: 3
Enter the Linux command to execute (e.g., 'ls -l'): ls -l

[Parent] PID=4821 creating 3 children to run: ls -l



[Child 1] PID=4822, Parent PID=4821, executing command: ls -l

total 12
-rw-r--r-- 1 user user  120 Feb 10 10:20 file1.txt
-rw-r--r-- 1 user user  340 Feb 10 10:20 file2.txt
drwxr-xr-x 2 user user 4096 Feb 10 10:20 folderA
```

```
[Child 2] PID=4823, Parent PID=4821, executing command: ls -l
total 12
-rw-r--r-- 1 user user 120 Feb 10 10:20 file1.txt
-rw-r--r-- 1 user user 340 Feb 10 10:20 file2.txt
drwxr-xr-x 2 user user 4096 Feb 10 10:20 folderA

[Child 3] PID=4824, Parent PID=4821, executing command: ls -l
total 12
-rw-r--r-- 1 user user 120 Feb 10 10:20 file1.txt
-rw-r--r-- 1 user user 340 Feb 10 10:20 file2.txt
drwxr-xr-x 2 user user 4096 Feb 10 10:20 folderA

[Parent] Child PID=4822 exited with status 0
[Parent] Child PID=4823 exited with status 0
[Parent] Child PID=4824 exited with status 0
```

Task 3: Zombie & Orphan Processes

Zombie: Fork a child and skip wait() in the parent.
Orphan: Parent exits before the child finishes.
Use ps -el | grep defunct to identify zombies.

Process_management.py

```
import os
import time
def zombie_process():
    pid = os.fork()
    if pid == 0:
        print(f"[Child] PID={os.getpid()}, Parent PID={os.getppid()} -> Exiting now.")
        os._exit(0)
    else:
        print(f"[Parent] PID={os.getpid()}, created child PID={pid}")
        print("[Parent] Not calling wait(), sleeping... Run 'ps -el | grep defunct' to see zombie.")
        time.sleep(30)
def orphan_process():
    pid = os.fork()
    if pid == 0:
        print(f"[Child] PID={os.getpid()}, Parent PID={os.getppid()} -> Sleeping...")
        time.sleep(20)
        print(f"[Child] PID={os.getpid()}, New Parent PID={os.getppid()} -> I am orphaned.")
    else:
        print(f"[Parent] PID={os.getpid()}, created child PID={pid} -> Exiting immediately.")
        os._exit(0)
if __name__ == "__main__":
    print("\n=== Task 3: Zombie & Orphan Processes ===")
    print("1. Zombie process demo")
    print("2. Orphan process demo")
    choice = input("Enter choice: ")
```

```
    if choice == "1":
        zombie_process()
    elif choice == "2":
        orphan_process()
    else:
        print("Invalid choice.")
```

Expected Output :

```
=== Task 3: Zombie & Orphan Processes ===
1. Zombie process demo
2. Orphan process demo
Enter choice: 1

[Parent] PID=4821, created child PID=4822
[Parent] Not calling wait(), sleeping... Run 'ps -el | grep defunct' to see zombie.

[Child] PID=4822, Parent PID=4821 -> Exiting now.
```

```
=== Task 3: Zombie & Orphan Processes ===
1. Zombie process demo
2. Orphan process demo
Enter choice: 2

[Parent] PID=4821, created child PID=4823 -> Exiting immediately.

[Child] PID=4823, Parent PID=4821 -> Sleeping...

[Child] PID=4823, New Parent PID=1 -> I am orphaned.
```

Task 4: Inspecting Process Info from /proc

Take a PID as input. Read and print:
- Process name, state, memory usage from /proc/[pid]/status
- Executable path from /proc/[pid]/exe
- Open file descriptors from /proc/[pid]/fd

Process_management.py

```
import os
def inspect_process(pid):
    status_file = f"/proc/{pid}/status"
    exe_file = f"/proc/{pid}/exe"
    fd_dir = f"/proc/{pid}/fd"

    try:
        with open(status_file, "r") as f:
```

```
            name, state, vm_size = None, None, None
            for line in f:
                if line.startswith("Name:"):
                    name = line.split()[1]
                elif line.startswith("State:"):
                    state = " ".join(line.split()[1:])
                elif line.startswith("VmSize:"):
                    vm_size = " ".join(line.split()[1:])
            print(f"Process Name : {name}")
            print(f"Process State: {state}")
            print(f"Memory Usage : {vm_size}")

        try:
            exe_path = os.readlink(exe_file)
            print(f"Executable   : {exe_path}")
        except FileNotFoundError:
            print("Executable   : [Not available]")

        try:
            fds = os.listdir(fd_dir)
            print(f"Open FDs     : {len(fds)}")
            for fd in fds:
                try:
                    target = os.readlink(os.path.join(fd_dir, fd))
                    print(f"  FD {fd} -> {target}")
                except OSError:
                    print(f"  FD {fd} -> [unavailable]")
        except FileNotFoundError:
            print("Open FDs     : [Not available]")

    except FileNotFoundError:
        print(f"Process with PID {pid} does not exist.")

if __name__ == "__main__":
    pid = input("Enter PID to inspect: ")
    if pid.isdigit():
        inspect_process(pid)
    else:
        print("Invalid PID")
```

Expected Output :

```
Enter PID to inspect: 8165
Process Name : systemd-udevd
Process State: S (sleeping)
Memory Usage : 35684 kB
Executable   : /usr/lib/systemd/systemd-udevd
Open FDs     : 3
  FD 0 -> /dev/null
  FD 1 -> /dev/null
  FD 2 -> /dev/null
```

Task 5: Process Prioritization

Create multiple CPU-intensive child processes. Assign different nice() values. Observe and log execution order to show scheduler impact.

Process_management.py

```
import os
import time
import multiprocessing as mp

def cpu_worker(nice_value, duration):
    try:
        new_nice = os.nice(nice_value)
    except Exception as e:
        print(f"PID={os.getpid()} | Error setting nice({nice_value}): {e}")
        return

    pid = os.getpid()
    start = time.time()
    count = 0
    while time.time() - start < duration:
        count += 1

    print(f"PID={pid} | Requested nice={nice_value} | "
          f"Actual nice={new_nice} | Iterations={count}")
if __name__ == "__main__":
    duration = 5
    nice_values = [0, 5, 10, 15]
    procs = []

    for n in nice_values:
        p = mp.Process(target=cpu_worker, args=(n, duration))
        p.start()
        procs.append(p)

    for p in procs:
        p.join()
```

Expected Output :

```
PID=4821 | Requested nice=0  | Actual nice=0  | Iterations=158394293
PID=4822 | Requested nice=5  | Actual nice=5  | Iterations=121584022
PID=4823 | Requested nice=10 | Actual nice=10 | Iterations=87930412
PID=4824 | Requested nice=15 | Actual nice=15 | Iterations=56493288
```

Lab Assignment Sheet-2

Problem Title:System Startup, Process Creation, and Termination Simulation in Python

Problem Statement:

Modern operating systems are responsible for initializing system components, creating processes, managing execution, and gracefully shutting down. This lab aims to simulate these core concepts using Python, helping students visualize how processes are handled at the OS level. The focus is on creating a simplified startup mechanism that spawns multiple processes and logs their lifecycle using the multiprocessing and logging modules. This hands-on simulation enhances conceptual clarity and promotes coding proficiency in scripting real-world OS behavior.

Tools/Technology Used:

• Python 3.x
• multiprocessing module
• time module
• logging module

Learning Objectives:

1.Upon completing this lab, students will be able to:
2. Understand the concepts of system booting, process creation, and termination.
3. Develop Python scripts using multiprocessing and logging modules.
4. Simulate system behaviour using programming constructs.

Assignment Tasks:

1. Write a Python script to simulate a basic system startup sequence.
2. Use the multiprocessing module to create at least two child processes that perform dummy tasks.
3. Implement proper logging to track process start and end times.
4. Generate a log file (process_log.txt) to reflect system-like behavior.
5. Submit the Python script and log file along with a short report explaining your implementation.

Sub-Tasks:

Sub-Task 1: Initialize the logging configuration

Objective: Set up the logging system to log messages with timestamps and process names.

Index.py

```
import multiprocessing
import time
import logging
```

```
logging.basicConfig(filename='process_log.txt', level=logging.INFO,
                    format='%(asctime)s - %(processName)s - %(message)s')

def system_process(task_name):
    logging.info(f"{task_name} started")
    time.sleep(2)
    logging.info(f"{task_name} ended")

if __name__ == '__main__':
    print("System Starting...")

    p1 = multiprocessing.Process(target=system_process, args=('Process-1',))
    p2 = multiprocessing.Process(target=system_process, args=('Process-2',))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    print("System Shutdown.")
```

#sub task – 01

```
import logging

# Setup logger

logging.basicConfig(

    filename='process_log.txt',

    level=logging.INFO,

    format='%(asctime)s - %(processName)s - %(message)s'

)
```

Expected Output :

No console output for this sub-task.

Sub-Task 2: Define a function that simulates a process task

Objective: Write a function that mimics the work of a system process.

task_2.py

```
def system_process(task_name, duration=2):
    """
    Simulates a system process task.
    Logs start and end of the task and sleeps for 'duration' seconds.
    """
    logging.info(f"{task_name} started")
    print(f"[{task_name}] started.")   # Console output for visualization
    time.sleep(duration)               # Simulate task work
    logging.info(f"{task_name} ended")
    print(f"[{task_name}] ended.")
```

Expected Output :

```
[Process-1] started (PID=4821)
[Process-1] ended (PID=4821)
```

```
2025-11-26 20:05:12 - Process-1 - Process-1 started
2025-11-26 20:05:14 - Process-1 - Process-1 ended
```

Sub-Task 3: Create at least two processes and start them concurrently

Objective: Use the multiprocessing module to initiate parallel tasks.

Task_3.py

```
if __name__ == '__main__':
    print("System Starting...\n")

    p1 = multiprocessing.Process(target=system_process, args=('Process-1', 3))
    p2 = multiprocessing.Process(target=system_process, args=('Process-2', 4))

    p1.start()
    p2.start()

    print("Processes started concurrently.")
```

Expected Output :

```
System Starting...

[Process-1] started (PID=4821)
[Process-2] started (PID=4822)
Processes started concurrently.

2025-11-26 20:10:12 - Process-1 - Process-1 started
2025-11-26 20:10:12 - Process-2 - Process-2 started
```

Sub-Task 4: Ensure proper termination and verify logs

Objective: Wait for processes to complete and confirm the shutdown.

Task_4.py

```
p1.join()
p2.join()
p3.join()

# Confirm shutdown
print("\nSystem Shutdown.")
print("Check 'process_log.txt' for detailed logs.")
```

Expected Output :

```
[Process-1] ended (PID=4821)
[Process-2] ended (PID=4822)
[Process-3] ended (PID=4823)

System Shutdown.
Check 'process_log.txt' for detailed logs.

2025-11-26 20:30:12 - Process-1 - Process-1 started
2025-11-26 20:30:12 - Process-2 - Process-2 started
2025-11-26 20:30:12 - Process-3 - Process-3 started
2025-11-26 20:30:15 - Process-1 - Process-1 ended
2025-11-26 20:30:16 - Process-2 - Process-2 ended
2025-11-26 20:30:17 - Process-3 - Process-3 ended
```

Lab Assignment Sheet-3

Problem Title:Simulation of File Allocation, Memory Management, and Scheduling in Python

Problem Statement:

Operating systems rely on robust memory management techniques, efficient CPU scheduling policies, and optimized file allocation strategies to manage hardware resources effectively. This lab aims to simulate various such components using Python. Students will implement and analyze Priority and Round Robin scheduling, simulate file allocation techniques (Sequential and Indexed), and explore memory management strategies (MFT, MVT, Worst-fit, Best-fit, First-fit). These implementations will reinforce theoretical OS concepts through hands-on coding experience.

Tools/Technology Used:

• Python 3.x
• Built-in Python libraries: os, multiprocessing, time
• Linux OS (optional for simulation)

Learning Objectives:

1. Simulate and analyze CPU scheduling algorithms.
2. Implement file allocation techniques in Python.
3. Demonstrate memory management strategies including MFT and MVT.

Assignment Tasks:

Task 1: CPU Scheduling with Gantt Chart

Write a Python program to simulate Priority and Round Robin scheduling algorithms.
Compute average waiting and turnaround times.

Task_1.py

```
def priority_scheduling():
    n = int(input("Enter number of processes: "))
    processes = []
    for i in range(n):
        bt = int(input(f"Enter Burst Time for P{i+1}: "))
        pr = int(input(f"Enter Priority (lower number means higher priority) for P{i+1}:
"))
        processes.append((i+1, bt, pr))

    processes.sort(key=lambda x: x[2])

    wt = [0] * n
    tat = [0] * n
```

```
    total_wt = 0
    total_tat = 0

    for i in range(n):
        wt[i] = sum([p[1] for p in processes[:i]])
        tat[i] = wt[i] + processes[i][1]
        total_wt += wt[i]
        total_tat += tat[i]

    print("\nPriority Scheduling:")
    print("PID\tBT\tPriority\tWT\tTAT")
    for i in range(n):

print(f"P{processes[i][0]}\t{processes[i][1]}\t{processes[i][2]}\t\t{wt[i]}\t{tat[i]}")

    print(f"\nAverage Waiting Time: {total_wt / n:.2f}")
    print(f"Average Turnaround Time: {total_tat / n:.2f}")


def round_robin():
    n = int(input("Enter number of processes: "))
    bt = []
    for i in range(n):
        bt.append(int(input(f"Enter Burst Time for P{i+1}: ")))

    quantum = int(input("Enter Time Quantum: "))
    rem_bt = bt.copy()
    t = 0
    wt = [0] * n

    while True:
        done = True
        for i in range(n):
            if rem_bt[i] > 0:
                done = False
                if rem_bt[i] > quantum:
                    t += quantum
                    rem_bt[i] -= quantum
                else:
                    t += rem_bt[i]
                    wt[i] = t - bt[i]
                    rem_bt[i] = 0
        if done:
            break

    tat = [bt[i] + wt[i] for i in range(n)]
```

```
    print("\nRound Robin Scheduling:")
    print("PID\tBT\tWT\tTAT")
    for i in range(n):
        print(f"P{i+1}\t{bt[i]}\t{wt[i]}\t{tat[i]}")

    print(f"\nAverage Waiting Time: {sum(wt) / n:.2f}")
    print(f"Average Turnaround Time: {sum(tat) / n:.2f}")

def task1_cpu_scheduling():
    print("Select Algorithm:")
    print("1. Priority Scheduling")
    print("2. Round Robin")

    choice = int(input("Enter choice: "))
    if choice == 1:
        priority_scheduling()
    elif choice == 2:
        round_robin()
    else:
        print("Invalid choice")

if __name__ == "__main__":
    task1_cpu_scheduling()
```

Expected Output :

```
Select Algorithm:
1. Priority Scheduling
2. Round Robin
Enter choice: 1
Enter number of processes: 3
Enter Burst Time for P1: 10
Enter Priority (lower number means higher priority) for P1: 2
Enter Burst Time for P2: 5
Enter Priority (lower number means higher priority) for P2: 1
Enter Burst Time for P3: 8
Enter Priority (lower number means higher priority) for P3: 3
```

```
 Priority Scheduling:
 PID BT  Priority    WT  TAT
 P2  5   1       0   5
 P1  10  2       5   15
 P3  8   3       15  23

 Average Waiting Time: 6.67
 Average Turnaround Time: 14.33
```

```
Select Algorithm:
1. Priority Scheduling
2. Round Robin
Enter choice: 2
Enter number of processes: 3
Enter Burst Time for P1: 10
Enter Burst Time for P2: 5
Enter Burst Time for P3: 8
Enter Time Quantum: 3
```

```
Round Robin Scheduling:
PID BT  WT  TAT
P1  10  13  23
P2  5   6   11
P3  8   12  20

Average Waiting Time: 10.33
Average Turnaround Time: 18.00
```

Task 2: Sequential File Allocation

Write a Python program to simulate sequential file allocation strategy.

Task_2.py

```
def sequential_file_allocation():
    total_blocks = int(input("Enter total number of blocks: "))
    block_status = [0] * total_blocks  # 0 = free, 1 = allocated
    n = int(input("Enter number of files: "))
    print("\nFile Allocation Results:")

    for i in range(n):
        print(f"\nFile {i+1}:")
        start = int(input("Enter starting block: "))
        length = int(input("Enter length of file: "))

        allocated = True
        if start < 0 or start + length > total_blocks:
            allocated = False
        else:
            for j in range(start, start + length):
                if block_status[j] == 1:
                    allocated = False
                    break
```

```
        if allocated:
            for j in range(start, start + length):
                block_status[j] = 1
            print(f"File {i+1} allocated from block {start} to {start + length - 1}")
        else:
            print(f"File {i+1} cannot be allocated due to insufficient or occupied
blocks")
    print("\nFinal Block Status (0 = free, 1 = allocated):")
    print(block_status)
if __name__ == "__main__":
    sequential_file_allocation()
```

Expected Output :

```
Enter total number of blocks: 10
Enter number of files: 3

File 1:
Enter starting block: 0
Enter length of file: 4

File 2:
Enter starting block: 4
Enter length of file: 3

File 3:
Enter starting block: 2
Enter length of file: 3
```

```
File Allocation Results:

File 1:
File 1 allocated from block 0 to 3

File 2:
File 2 allocated from block 4 to 6

File 3:
File 3 cannot be allocated due to insufficient or occupied blocks

Final Block Status (0 = free, 1 = allocated):
[1, 1, 1, 1, 1, 1, 1, 0, 0, 0]
```

Task 3: Indexed File Allocation

Write a Python program to simulate indexed file allocation strategy.

Task_3.py

```
def indexed_file_allocation():
    total_blocks = int(input("Enter total number of blocks: "))
    block_status = [0] * total_blocks  # 0 = free, 1 = allocated
    n = int(input("Enter number of files: "))

    print("\nFile Allocation Results:")

    for i in range(n):
        print(f"\nFile {i+1}:")
        index_block = int(input("Enter index block: "))
        if index_block < 0 or index_block >= total_blocks or block_status[index_block] == 1:
            print("Index block is invalid or already allocated.")
            continue
        count = int(input("Enter number of data blocks: "))
        data_blocks = list(map(int, input(f"Enter {count} data block numbers (space-separated): ").split()))

        if len(data_blocks) != count or any(blk < 0 or blk >= total_blocks or block_status[blk] == 1 for blk in data_blocks):
            print("Invalid or already allocated data blocks. Allocation failed.")
            continue
        block_status[index_block] = 1
        for blk in data_blocks:
            block_status[blk] = 1
        print(f"File {i+1} allocated with index block {index_block} -> Data blocks: {data_blocks}")
    print("\nFinal Block Status (0 = free, 1 = allocated):")
    print(block_status)
if __name__ == "__main__":
    indexed_file_allocation()
```

Expected Output :

```
Enter total number of blocks: 10
Enter number of files: 2

File 1:
Enter index block: 0
Enter number of data blocks: 3
Enter 3 data block numbers (space-separated): 1 2 3

File 2:
Enter index block: 2
Enter number of data blocks: 2
Enter 2 data block numbers (space-separated): 4 5
```

```
File Allocation Results:

File 1:
File 1 allocated with index block 0 -> Data blocks: [1, 2, 3]

File 2:
Index block is invalid or already allocated.

Final Block Status (0 = free, 1 = allocated):
[1, 1, 1, 1, 0, 0, 0, 0, 0, 0]
```

Task 4: Contiguous Memory Allocation

Simulate Worst-fit, Best-fit, and First-fit memory allocation strategies.

Task_4.py

```
def allocate_memory(strategy, partitions, processes):
    allocation = [-1] * len(processes)
    for i, psize in enumerate(processes):
        idx = -1
        if strategy == "first":
            for j, part in enumerate(partitions):
                if part >= psize:
                    idx = j
                    break
        elif strategy == "best":
            bestfit = float('inf')
            for j, part in enumerate(partitions):
                if part >= psize and part < bestfit:
                    bestfit = part
                    idx = j
        elif strategy == "worst":
```

```
            worstfit = -1
            for j, part in enumerate(partitions):
                if part >= psize and part > worstfit:
                    worstfit = part
                    idx = j
        if idx != -1:
            allocation[i] = idx
            partitions[idx] -= psize
    for i, a in enumerate(allocation):
        if a != -1:
            print(f"Process {i+1} allocated in Partition {a+1}")
        else:
            print(f"Process {i+1} cannot be allocated")

def main():
    partitions = list(map(int, input("Enter partition sizes separated by space: ").split()))
    processes = list(map(int, input("Enter process sizes separated by space: ").split()))
    print("Select strategy:\n1. First-fit\n2. Best-fit\n3. Worst-fit")
    choice = int(input("Enter choice: "))
    if choice == 1:
        allocate_memory("first", partitions, processes)
    elif choice == 2:
        allocate_memory("best", partitions, processes)
    elif choice == 3:
        allocate_memory("worst", partitions, processes)
    else:
        print("Invalid choice")

if __name__ == "__main__":
    main()
```

Expected Output :

```
Enter partition sizes (space-separated): 100 500 200 300 600
Enter process sizes (space-separated): 212 417 112 426

Select Memory Allocation Strategy:
1. First-Fit
2. Best-Fit
3. Worst-Fit
Enter choice: 1

=== First-Fit Strategy ===
Process 1 (Size 212) -> Partition 2
Process 2 (Size 417) -> Partition 5
Process 3 (Size 112) -> Partition 3
Process 4 (Size 426) -> Not allocated

=== Best-Fit Strategy ===
Process 1 (Size 212) -> Partition 4
Process 2 (Size 417) -> Partition 5
Process 3 (Size 112) -> Partition 3
Process 4 (Size 426) -> Not allocated

=== Worst-Fit Strategy ===
Process 1 (Size 212) -> Partition 5
Process 2 (Size 417) -> Partition 2
Process 3 (Size 112) -> Partition 5
Process 4 (Size 426) -> Not allocated
```

Task 5: MFT & MVT Memory Management

Implement MFT (fixed partitions) and MVT (variable partitions) strategies in Python.

Task_5.py

```
def MFT():
    memsize = int(input("Enter total memory size: "))
    partsize = int(input("Enter partition size: "))
    n = memsize
    print(f"Memory divided into {n} partitions of size {partsize}")
    for i in range(n):
        psize = int(input(f"Enter size of Process {i+1}: "))
        if psize <= partsize:
            print(f"Process {i+1} allocated.")
        else:
            print(f"Process {i+1} too large for fixed partition.")

def MVT():
    memsize = int(input("Enter total memory size: "))
    n = int(input("Enter number of processes: "))
```

```
    for i in range(n):
        psize = int(input(f"Enter size of Process {i+1}: "))
        if psize <= memsize:
            print(f"Process {i+1} allocated.")
            memsize -= psize
        else:
            print(f"Process {i+1} cannot be allocated. Not enough memory.")

def main():
    print("Select Method:\n1. MFT\n2. MVT")
    choice = int(input("Enter choice: "))
    if choice == 1:
        MFT()
    elif choice == 2:
        MVT()
    else:
        print("Invalid choice")

if __name__ == "__main__":
    main()
```

Expected Output :

```
1. MFT
2. MVT
Enter choice: 1

Enter total memory size: 1000
Enter partition size: 250
Memory divided into 4 partitions of size 250

Enter size of Process 1: 120
Process 1 allocated.

Enter size of Process 2: 300
Process 2 too large for fixed partition.

Enter size of Process 3: 250
Process 3 allocated.

Enter size of Process 4: 180
Process 4 allocated.
```

Lab Assignment Sheet-4

Problem Title:System Calls, VM Detection, and File System Operations using Python

Problem Statement:

Operating systems expose low-level interfaces like system calls to allow interaction between user programs and the OS kernel. This lab simulates system-level OS tasks such as process creation (using fork and exec), file and memory operations, VM detection, and CPU scheduling. Learners will develop shell, C, and Python scripts to model batch execution, inter-process communication, and basic file system behaviors.

Tools/Technology Used:

Python 3.x
Linux OS (for system-level calls)
OS, platform, subprocess modules
Bash shell, C programming (for fork/exec/wait)

Learning Objectives:

1. Execute Python scripts in a batch processing environment
2. Simulate system startup and process termination logging
3. Demonstrate use of system calls (fork(), exec(), wait(), pipe())
4. Detect virtual machine environments programmatically
5. Simulate basic file system operations using Python

Sub Tasks

Task 1: Batch Processing Simulation (Python)

Write a Python script to execute multiple .py files sequentially, mimicking batch processing.

Task_1.py

```
import os
import subprocess
folder = "batch_project"
os.makedirs(folder, exist_ok=True)

script1 = """print("Script 1 is running...")
for i in range(3):
    print(f"Script 1 count: {i}")"""

script2 = """print("Script 2 is processing data...")
data = [10, 20, 30]
print("Sum of data =", sum(data))"""
```

```
script3 = """print("Script 3 is generating output...")
name = "Faaiz"
print(f"Hello {name}, batch processing completed!")"""

files = {
    "script1.py": script1,
    "script2.py": script2,
    "script3.py": script3,
}

for filename, content in files.items():
    with open(os.path.join(folder, filename), "w") as f:
        f.write(content)

print("All scripts created successfully.\n")

scripts = ["script1.py", "script2.py", "script3.py"]

for script in scripts:
    print(f"\nExecuting {script}...")
    subprocess.run(["python3", os.path.join(folder, script)])
```

Expected Output :

```
All scripts created successfully.


Executing script1.py...
Script 1 is running...
Script 1 count: 0
Script 1 count: 1
Script 1 count: 2

Executing script2.py...
Script 2 is processing data...
Sum of data = 60

Executing script3.py...
Script 3 is generating output...
Hello Faaiz, batch processing completed!
```

Task 2: System Startup and Logging

Simulate system startup using Python by creating multiple processes and logging their start and end into a log file.

Task_2.py

```
import multiprocessing
import logging
import time
```

```
logging.basicConfig(
    filename="system_log.txt",
    level=logging.INFO,
    format="%(asctime)s - %(processName)s - %(message)s"
)

def start_service(service_name):
    logging.info(f"{service_name} initialized")
    time.sleep(2)      # Simulate service execution delay
    logging.info(f"{service_name} completed")

if __name__ == "__main__":
    print("Initializing System Services...")

    service_A = multiprocessing.Process(target=start_service, args=("Service-A",))
    service_B = multiprocessing.Process(target=start_service, args=("Service-B",))
    service_C = multiprocessing.Process(target=start_service, args=("Service-C",))

    service_A.start()
    service_B.start()
    service_C.start()

    service_A.join()
    service_B.join()
    service_C.join()

    print("All Services Successfully Stopped. System Shutdown.")
```

Expected Output :

```
2025-11-26 18:42:11,004 - Service-A - Service-A initialized
2025-11-26 18:42:11,006 - Service-B - Service-B initialized
2025-11-26 18:42:11,008 - Service-C - Service-C initialized
2025-11-26 18:42:13,010 - Service-A - Service-A completed
2025-11-26 18:42:13,012 - Service-B - Service-B completed
2025-11-26 18:42:13,013 - Service-C - Service-C completed
```

Task 3: System Calls and IPC (Python - fork, exec, pipe)

Use system calls (fork(), exec(), wait()) and implement basic Inter-Process Communication using pipes in C or Python.

Task_3.py

```
import os
def main():

    read_end, write_end = os.pipe()
    pid = os.fork()
```

```
    if pid > 0:

        os.close(read_end)

        parent_message = b"Message from Parent Process"

        os.write(write_end, parent_message)

        print("Parent sent message to child.")

        os.close(write_end)
        os.wait()

        print("Parent: Child process finished.")

    else:
        os.close(write_end)

        message = os.read(read_end, 1024).decode()
        print(f"Child received: {message}")

        reply = "Child acknowledges the message.\n"

        os.write(1, reply.encode())

        os.close(read_end)


if __name__ == "__main__":
    main()
```

Expected Output :

```
Parent sent message to child.
Child received: Message from Parent Process
Child acknowledges the message.
Parent: Child process finished.
```

Task 4: VM Detection and Shell Interaction

Create a shell script to print system details and a Python script to detect if the system is running inside a virtual machine.

Task_4.py

```
import subprocess
import os

def print_system_details():
    print("==== System Details ====")

    kernel = subprocess.check_output("uname -r", shell=True, text=True).strip()
    print(f"Kernel Version: {kernel}")
    user = subprocess.check_output("whoami", shell=True, text=True).strip()
    print(f"User: {user}")

    try:
        virt_info = subprocess.check_output("lscpu | grep -i 'virtual'", shell=True, text=True).strip()
        print("Hardware Virtualization Info:")
        print(virt_info if virt_info else "No virtualization flag shown.")
    except:
        print("Unable to read hardware virtualization info.")

def detect_vm():
    print("\n==== VM Detection ====")
    indicators = []
    try:
        manufacturer = subprocess.check_output("sudo dmidecode -s system-manufacturer", shell=True,
text=True).lower().strip()
        if any(vm in manufacturer for vm in ["vmware", "virtualbox", "kvm", "qemu", "xen"]):
            indicators.append(f"Manufacturer indicates virtualization: {manufacturer}")
    except:
        indicators.append("DMI info not accessible (run with sudo).")
    try:
        cpu_info = subprocess.check_output("lscpu", shell=True, text=True).lower()
        if "hypervisor" in cpu_info:
            indicators.append("CPU flags show 'hypervisor' → running in VM.")
    except:
        indicators.append("Unable to read CPU info.")

    if indicators:
        print("Virtual Machine Detected")
        for item in indicators:
            print(" -", item)
    else:
        print("✓ No Virtual Machine Detected")

if __name__ == "__main__":
    print_system_details()
    detect_vm()
```

Expected Output :

```
==== System Details ====
Kernel Version: 5.15.0-91-generic
User: prerna
Hardware Virtualization Info:
Virtualization:                 VT-x
Virtualization type:            full

==== VM Detection ====
Virtual Machine Detected
 - Manufacturer indicates virtualization: vmware, inc.
 - CPU flags show 'hypervisor' → running in VM.
```