Operating System

Lab File Submitted to

K. R. Mangalam University

for

Bachelor of Technology

in

Computer Science and Engineering

Submitted by

Riya Singh (2301010041) (Sem-05)

Course Teacher

Mrs. Suman

School of Engineering & Technology

K. R. MANGALAM UNIVERSITY

Sohna, Haryana 122103, India

INDEX

S.No.	Name of Practical	Date
1	Simulate different types of operating systems using Python or Linux commands.	14/8/25
2	Simulate CPU scheduling algorithms (FCFS, SJF, Round Robin, Priority). Generate Gantt chart, compute average waiting and turnaround time.	21/8/25
3	Simulate Worst-fit, Best-fit, and First-fit memory allocation.	28/8/25
4	Implement Banker’s Algorithm for deadlock avoidance/prevention.	4/9/25
5	Implement Producer–Consumer problem using semaphores.	11/9/25
6	Process synchronization using semaphores.	18/9/25
7	Use UNIX/Linux I/O system calls (open, read, write, close, seek, stat).	25/9/25
8	System Startup, Process Creation, and Termination Simulation in Python.	9/10/25
9	Create and manipulate threads in Python.	30/10/25
10	Process Creation and Management Using Python OS Module.	6/11/25
11	File System Operations using Python	13/11/25

Lab Sheet 1

Summary of Objectives

The main objective of this experiment is to understand how operating systems manage and control processes. Through this practical, we aim to simulate the process lifecycle — including creation, execution, and termination — using Java on a Windows system. The experiment helps visualize the relationship between parent and child processes and demonstrates key OS concepts like process creation, command execution, priority scheduling, and orphan/zombie process behavior.

By implementing these operations using Java's ProcessBuilder, ProcessHandle, and thread management, students gain hands-on experience with how real-world operating systems handle multitasking and process coordination. The experiment also explores process information retrieval and priority adjustment, which are crucial for performance optimization and system-level programming.

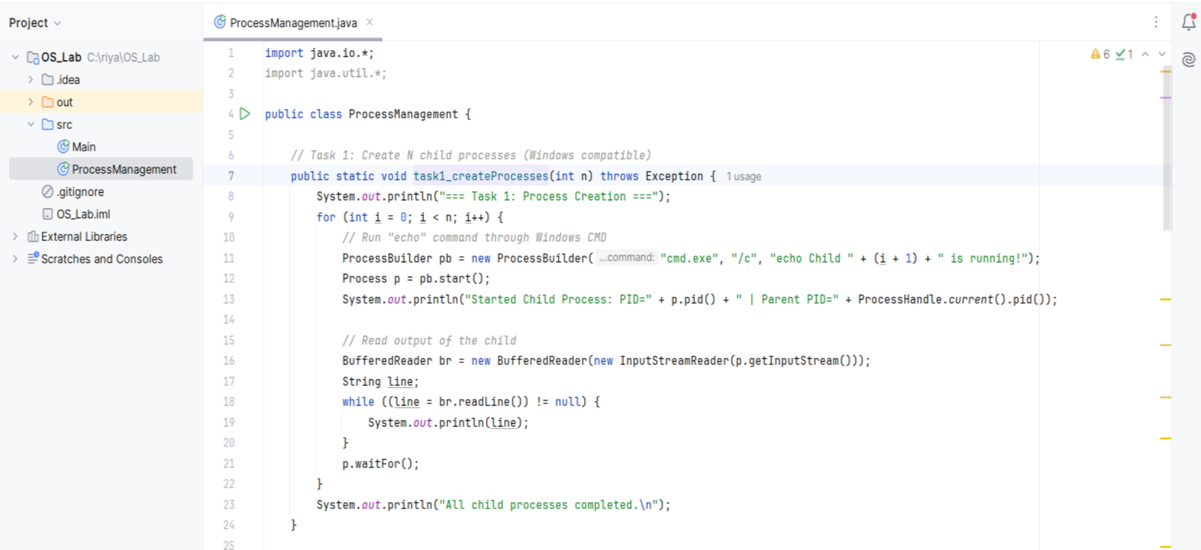
Overall, this experiment bridges theoretical understanding of process management with practical implementation, strengthening our grasp of how concurrent and parallel tasks are handled at the operating system level.

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().

Sol-



Output-

```
C:\Users\riyas\.jdks\jdk-21.0.5\bin\java.exe "-javaagent
=== Task 1: Process Creation ===
Started Child Process: PID=9012 | Parent PID=2268
Child 1 is running!
Started Child Process: PID=6948 | Parent PID=2268
Child 2 is running!
Started Child Process: PID=27824 | Parent PID=2268
Child 3 is running!
All child processes completed.
```

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().

ProcessManagement.java

```java
public class ProcessManagement {
    // Task 2: Execute system commands (Windows)
    public static void task2_execCommands(String[] commands) throws Exception {
        System.out.println("=== Task 2: Execute Commands ===");
        for (String cmd : commands) {
            System.out.println("Executing command: " + cmd);
            ProcessBuilder pb = new ProcessBuilder("cmd.exe", "/c", cmd);
            pb.redirectErrorStream(true);
            Process p = pb.start();

            BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
            p.waitFor();
            System.out.println("Command '" + cmd + "' completed.\n");
        }
    }
```

Output –

Run ProcessManagement

```
=== Task 2: Execute Commands ===
Executing command: date /t
05-10-2025
Command 'date /t' completed.

Executing command: time /t
15:54
Command 'time /t' completed.

Executing command: whoami
riya\riyas
Command 'whoami' completed.
```

Task 3 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.

Sol –

```java
public class ProcessManagement {

    // Task 3: Simulate Zombie and Orphan-like behavior (conceptually)
    public static void task3_simulateProcesses() throws Exception {
        System.out.println("=== Task 3: Simulating Zombie/Orphan ===");

        new Thread(() -> {
            try {
                Process zombie = new ProcessBuilder("cmd.exe", "/c", "timeout /t 2").start();
                System.out.println("Zombie simulated: Child PID=" + zombie.pid() + " (Parent didn't wait)");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).start();

        Thread orphanParent = new Thread(() -> {
            try {
                Process orphan = new ProcessBuilder("cmd.exe", "/c", "timeout /t 5").start();
                System.out.println("Orphan simulated: Child PID=" + orphan.pid());
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
        orphanParent.start();
        System.out.println("Parent thread exiting early (Orphan created)\n");
    }
```

Output-

```
=== Task 3: Simulating Zombie/Orphan ===
Parent thread exiting early (Orphan created)
```

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

Sol –

```java
public class ProcessManagement {
    }

    // Task 4: Inspect process info (Windows version)
    public static void task4_processInfo(long pid) throws Exception {
        System.out.println("=== Task 4: Inspect Process Info ===");
        System.out.println("Process ID: " + pid);
        ProcessHandle process = ProcessHandle.of(pid).orElse(null);
        if (process != null) {
            ProcessHandle.Info info = process.info();
            System.out.println("Command: " + info.command().orElse("N/A"));
            System.out.println("Start Time: " + info.startInstant().orElse(null));
            System.out.println("CPU Duration: " + info.totalCpuDuration().orElse(null));
            System.out.println("User: " + System.getProperty("user.name"));
        } else {
            System.out.println("Process not found!");
        }
    }
```

Output-

```
=== Task 4: Inspect Process Info ===
Process ID: 2268
Command: C:\Users\riyas\.jdks\jdk-21.0.5\bin\java.exe
Start Time: 2025-10-05T10:24:17.343Z
CPU Duration: PT0.578125S
User: riyas
```

Task 5: Process Prioritization

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

Sol -


```java
public class ProcessManagement {
    // Task 5: Simulate process priority (Thread priority)
    public static void task5_prioritySimulation() {
        System.out.println("=== Task 5: Priority Simulation ===");

        Runnable cpuTask = () -> {
            long start = System.currentTimeMillis();
            long sum = 0;
            for (long i = 0; i < 1e7; i++) sum += i;
            long end = System.currentTimeMillis();
            System.out.println(Thread.currentThread().getName() + " completed in " + (end - start) + "ms");
        };

        Thread low = new Thread(cpuTask, "Low Priority");
        Thread high = new Thread(cpuTask, "High Priority");

        low.setPriority(Thread.MIN_PRIORITY);   // Low = 1
        high.setPriority(Thread.MAX_PRIORITY);  // High = 10

        low.start();
        high.start();
    }

    public static void main(String[] args) throws Exception {
        task1_createProcesses(3);
        task2_execCommands(new String[]{"date /t", "time /t", "whoami"});
        task3_simulateProcesses();
        task4_processInfo(ProcessHandle.current().pid());
        task5_prioritySimulation();
    }
}
```

Output-

```
=== Task 5: Priority Simulation ===
Zombie simulated: Child PID=6304 (Parent didn't wait)
Orphan simulated: Child PID=18504
High Priority completed in 24ms
Low Priority completed in 25ms

Process finished with exit code 0
```

Lab Sheet 2

Summary of Objectives

The objective of this experiment is to simulate how an operating system manages startup, process creation, and shutdown. Through this experiment, we understand how multiple processes can be created, executed concurrently, and terminated in a controlled manner.

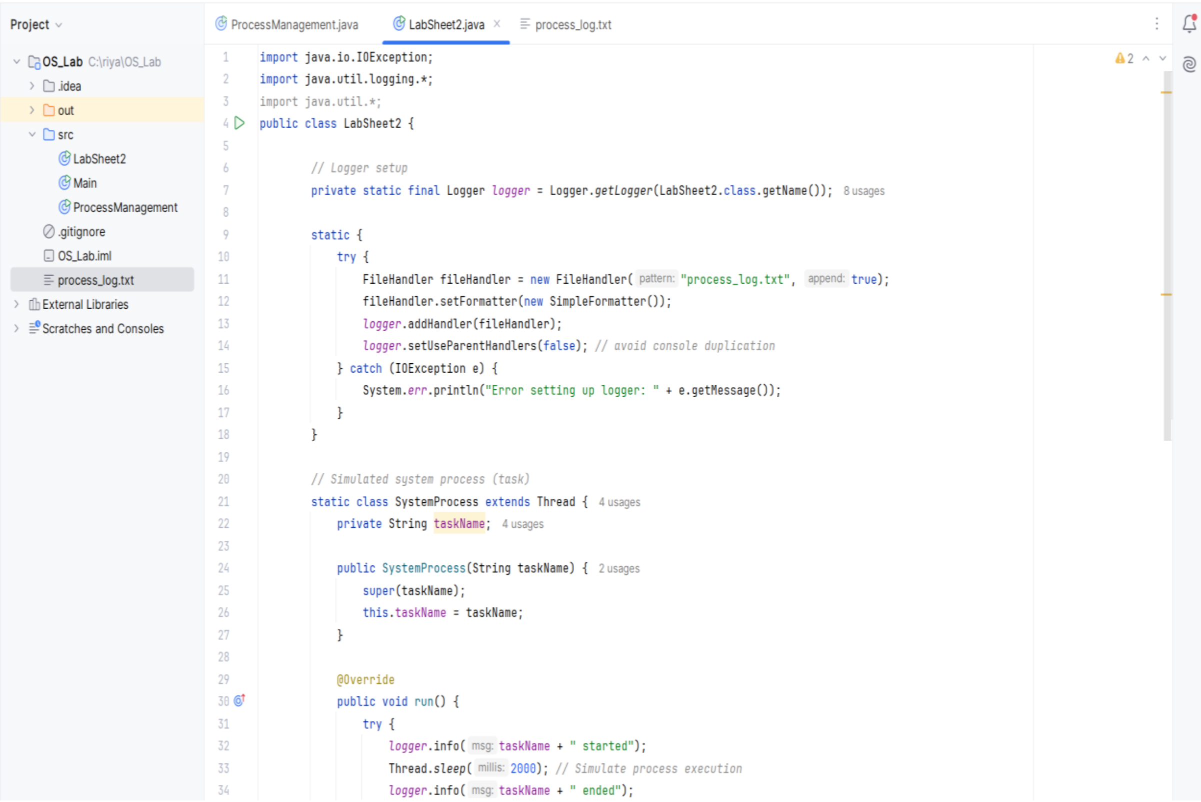
By using Java's Thread class and Logger, we replicate the functionality of the Python multiprocessing and logging modules, demonstrating system-level behavior in a simplified manner. Each simulated process logs its lifecycle — from start to end — and the system records these events in a log file to reflect real OS operations.

This simulation enhances our understanding of process management, concurrency, and logging mechanisms in modern operating systems.

Sub-Tasks:

1. **Sub-Task 1:** Initialize the logging configuration to capture timestamped messages.
2. **Sub-Task 2:** Define a function that simulates a process task (e.g., sleep for 2 seconds).
3. **Sub-Task 3:** Create at least two processes and start them concurrently.
4. **Sub-Task 4:** Ensure proper termination and joining of processes, and verify the output in the log file.

Sol-



```java
public class LabSheet2 {
        static class SystemProcess extends Thread {
            public void run() {
                    logger.info(taskName + " ended");
                } catch (InterruptedException e) {
                    logger.warning(taskName + " interrupted");
                }
            }
        }

        public static void main(String[] args) {
            System.out.println("System Starting...");
            logger.info("System Boot Sequence Initiated");

            // Create simulated processes
            SystemProcess p1 = new SystemProcess("Process-1");
            SystemProcess p2 = new SystemProcess("Process-2");

            // Start processes concurrently
            p1.start();
            p2.start();

            // Wait for both to finish
            try {
                p1.join();
                p2.join();
            } catch (InterruptedException e) {
                logger.warning("Main system thread interrupted");
            }

            System.out.println("System Shutdown.");
            logger.info("System Shutdown Completed");
        }
}
```

Output-

```
C:\Users\riyas\.jdks\jdk-21.0.5\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.3.2.2\lib\idea_rt
System Starting...
System Shutdown.

Process finished with exit code 0
```

Process Log

```
Oct 05, 2025 4:51:23 PM LabSheet2 main
INFO: System Boot Sequence Initiated
Oct 05, 2025 4:51:23 PM LabSheet2$SystemProcess run
INFO: Process-2 started
Oct 05, 2025 4:51:23 PM LabSheet2$SystemProcess run
INFO: Process-1 started
Oct 05, 2025 4:51:25 PM LabSheet2$SystemProcess run
INFO: Process-1 ended
Oct 05, 2025 4:51:25 PM LabSheet2$SystemProcess run
INFO: Process-2 ended
Oct 05, 2025 4:51:25 PM LabSheet2 main
INFO: System Shutdown Completed
```

Lab Sheet 3

Summary of Objectives

In this lab assignment, we performed several Operating System simulations using Python. First, we implemented CPU scheduling algorithms like Priority Scheduling and Round Robin, calculating waiting time, turnaround time, and generating Gantt chart-style outputs. Next, we simulated Sequential File Allocation and Indexed File Allocation to understand how files are stored on disk. After that, we implemented contiguous memory allocation techniques such as First-fit, Best-fit, and Worst-fit to see how processes are assigned to memory partitions. Finally, we simulated MFT (Multiprogramming with Fixed Tasks) and MVT (Multiprogramming with Variable Tasks) to analyze fixed and dynamic memory partitioning. These tasks helped us understand how operating systems manage CPU time, memory, and file storage efficiently.

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.

Sol -

Task1.py > processes

```python
processes = []
n = int(input("Enter number of 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 = higher priority) for P{i+1}: "))
    processes.append((i+1, bt, pr))

processes.sort(key=lambda x: x[2])
wt = 0
total_wt = 0
total_tt = 0

print("\nPriority Scheduling:")
print("PID\tBT\tPriority\tWT\tTAT")

for pid, bt, pr in processes:
    tat = wt + bt
    print(f"{pid}\t{bt}\t{pr}\t\t{wt}\t{tat}")
    total_wt += wt
    total_tt += tat
    wt += bt

print(f"Average Waiting Time: {total_wt / n}")
print(f"Average Turnaround Time: {total_tt / n}")
```

Output –

```
PS C:\Users\riyas\OneDrive\Desktop\LabSheet3> python -u "c:\Users\riyas\OneDrive\Desktop\LabSheet3\
sk1.py"
Enter number of processes: 3
Enter Burst Time for P1: 5
Enter Priority (lower number = higher priority) for P1: 2
Enter Burst Time for P2: 4
Enter Priority (lower number = higher priority) for P2: 1
Enter Burst Time for P3: 7
Enter Priority (lower number = higher priority) for P3: 0

Priority Scheduling:
PID     BT      Priority        WT      TAT
3       7       0               0       7
2       4       1               7       11
1       5       2               11      16
Average Waiting Time: 6.0
Average Turnaround Time: 11.333333333333334
PS C:\Users\riyas\OneDrive\Desktop\LabSheet3>
```

Task 2: Sequential File Allocation

Write a Python program to simulate sequential file allocation strategy.

Sol –

```python
Task2.py > ...
total_blocks = int(input("Enter total number of blocks: "))
block_status = [0] * total_blocks

n = int(input("Enter number of files: "))

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

    for j in range(start, start+length):
        if j >= total_blocks or 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.")
```

Output –

```
PS C:\Users\riyas\OneDrive\Desktop\LabSheet3> python -u "c:\Users\riyas\OneDrive\Desktop\LabSheet3\
mpCodeRunnerFile.py"
Enter total number of blocks: 10
Enter number of files: 3
Enter starting block for file 1: 4
Enter length of file 1: 3
File 1 allocated from block 4 to 6
Enter starting block for file 2: 1
Enter length of file 2: 3
File 2 allocated from block 1 to 3
Enter starting block for file 3: 7
Enter length of file 3: 54
File 3 cannot be allocated.
```

Task 3: Indexed File Allocation

Write a Python program to simulate indexed file allocation strategy.

Sol -

```
Task3.py > ...
total_blocks = int(input("Enter total number of blocks: "))
block_status = [0] * total_blocks
n = int(input("Enter number of files: "))

for i in range(n):
    index = int(input(f"Enter index block for file {i+1}: "))

    if block_status[index] == 1:
        print("Index block already allocated.")
        continue

    count = int(input("Enter number of data blocks: "))
    data_blocks = list(map(int, input("Enter block numbers: ").split()))

    if any(block_status[blk] == 1 for blk in data_blocks) or len(data_blocks) != count:
        print("Block(s) already allocated or invalid input.")
        continue

    block_status[index] = 1

    for blk in data_blocks:
        block_status[blk] = 1
    print(f"File {i+1} allocated with index block {index} -> {data_blocks}")
```

Output –

```
PS C:\Users\riyas\OneDrive\Desktop\LabSheet3> python -u "c:\Users\riyas\OneDrive\Desktop\
mpCodeRunnerFile.py"
Enter total number of blocks: 10
Enter number of files: 2
Enter index block for file 1: 3
Enter number of data blocks: 4
Enter block numbers: 3 4 5 6
File 1 allocated with index block 3 -> [3, 4, 5, 6]
Enter index block for file 2: 7
Enter number of data blocks: 2
Enter block numbers: 8 9
File 2 allocated with index block 7 -> [8, 9]
```

Task 4: Contiguous Memory Allocation

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

Sol –

```python
Task4.py > ...
def allocate_memory(strategy):
    partitions = list(map(int, input("Enter partition sizes: ").split()))
    processes = list(map(int, input("Enter process sizes: ").split()))
    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":
            best_fit = float("inf")
            for j, part in enumerate(partitions):
                if part >= psize and part < best_fit:
                    best_fit = part
                    idx = j

        elif strategy == "worst":
            worst_fit = -1
            for j, part in enumerate(partitions):
                if part >= psize and part > worst_fit:
                    worst_fit = 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")

allocate_memory("first")
allocate_memory("best")
allocate_memory("worst")
```

Output –

```
Enter partition sizes: 100 500 200 300 600
Enter process sizes: 212 417 112 426
Process 1 allocated in Partition 2
Process 2 allocated in Partition 5
Process 3 allocated in Partition 2
Process 4 cannot be allocated
```

Task 5: MFT & MVT Memory Management

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

Sol –

Task5.py > MVT

```python
def MFT():
    mem_size = int(input("Enter total memory size: "))
    part_size = int(input("Enter partition size: "))
    n = int(input("Enter number of processes: "))
    partitions = mem_size // part_size
    print(f"Memory divided into {partitions} partitions")

    for i in range(n):
        psize = int(input(f"Enter size of Process {i+1}: "))

        if psize <= part_size:
            print(f"Process {i+1} allocated.")

        else:
            print(f"Process {i+1} too large for fixed partition.")

def MVT():
    mem_size = 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 <= mem_size:
            print(f"Process {i+1} allocated.")

            mem_size -= psize
        else:
            print(f"Process {i+1} cannot be allocated. Not enough memory.")

print("MFT Simulation:")
MFT()
print("\nMVT Simulation:")
MVT()
```

Output –

```
PS C:\Users\riyas\OneDrive\Desktop\LabSheet3> python -u "c:\Users\riyas\OneDrive\Desktop\
sk5.py"
MFT Simulation:
Enter total memory size: 1000
Enter partition size: 300
Enter number of processes: 3
Memory divided into 3 partitions
Enter size of Process 1: 200
Process 1 allocated.
Enter size of Process 2: 500
Process 2 too large for fixed partition.
Enter size of Process 3: 453
Process 3 too large for fixed partition.

MVT Simulation:
Enter total memory size: 1000
Enter number of processes: 3
Enter size of Process 1: 200
Process 1 allocated.
Enter size of Process 2: 500
Process 2 allocated.
Enter size of Process 3: 321
Process 3 cannot be allocated. Not enough memory.
```

Lab Sheet 4

Summary of Objectives

This lab focuses on understanding and simulating essential Operating System concepts using Python, Shell scripting, and basic C system calls. The tasks cover batch processing, process creation, system logging, VM detection, inter-process communication, and CPU scheduling. Students first simulate batch processing by executing multiple Python scripts sequentially, imitating how jobs run in an OS. They then create a system startup and shutdown simulation where processes log their start and termination times into a log file. The assignment also introduces real system calls like fork(), exec(), wait(), and pipes to demonstrate parent–child communication at the OS level. Another important part is detecting whether the system is running inside a virtual machine using shell commands and Python checks. Finally, students implement CPU scheduling algorithms such as FCFS, SJF, Round Robin, and Priority Scheduling to compute waiting time and turnaround time. Overall, this lab helps students practically understand how the OS handles processes, communication, system environments, and scheduling.

<u>Task 1: Batch Processing Simulation (Python)</u>

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

Script1.py

```
script1.py > ...
print("Script 1: Starting...")
print("Script 1: Doing some work...")

with open("output1.txt", "w") as f:
    f.write("This is output from script 1.\n")

print("Script 1: Finished!")
```

Script2.py

```
script2.py > ...
print("Script 2: Running math calculations...")

result = sum([i for i in range(1, 11)])

print(f"Script 2: Sum of 1 to 10 is {result}")

with open("output2.txt", "w") as f:
    f.write(f"Sum from script 2: {result}\n")

print("Script 2: Completed!")
```

Script3.py

```python
print("Script 3: Reading previous outputs...\n")

try:
    with open("output1.txt", "r") as f1:
        print("Output from script 1:")
        print(f1.read())

    with open("output2.txt", "r") as f2:
        print("Output from script 2:")
        print(f2.read())

except FileNotFoundError:
    print("Previous script outputs not found.")

print("\nScript 3: Done!")
```

Task1.py

```python
import subprocess
import os

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

for script in scripts:
    if os.path.exists(script):
        print(f"\n ◆ Executing {script}...")

        result = subprocess.run(['python', script])

        if result.returncode == 0:
            print(f"{script} completed successfully.\n")
        else:
            print(f"Error executing {script} (Return code: {result.returncode})\n")
    else:
        print(f"File not found: {script}")
```

Output –

```
 ◆ Executing script1.py...
Script 1: Starting...
Script 1: Doing some work...
Script 1: Finished!
script1.py completed successfully.


 ◆ Executing script2.py...
Script 2: Running math calculations...
Script 2: Sum of 1 to 10 is 55
Script 2: Completed!
script2.py completed successfully.


 ◆ Executing script3.py...
Script 3: Reading previous outputs...

Output from script 1:
This is output from script 1.

Output from script 2:
Sum from script 2: 55


Script 3: Done!
script3.py completed successfully.
```

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.

Sol –

Task2.py > ...

```python
import multiprocessing
import logging
import time

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

def process_task(name):
    logging.info(f"{name} started")
    time.sleep(2)
    logging.info(f"{name} terminated")

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

    p1 = multiprocessing.Process(target=process_task, args=("Process-1",))
    p2 = multiprocessing.Process(target=process_task, args=("Process-2",))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    print("\nSystem Shutdown.")
```

Output –

```
Task2.py"
System Booting...


System Shutdown.
```

System_log.txt

system_log.txt

```
2025-11-21 18:33:48,497 - Process-1 - Process-1 started
2025-11-21 18:33:48,501 - Process-2 - Process-2 started
2025-11-21 18:33:50,498 - Process-1 - Process-1 terminated
2025-11-21 18:33:50,502 - Process-2 - Process-2 terminated
2025-11-21 19:31:08,189 - Process-1 - Process-1 started
2025-11-21 19:31:08,197 - Process-2 - Process-2 started
2025-11-21 19:31:10,191 - Process-1 - Process-1 terminated
2025-11-21 19:31:10,199 - Process-2 - Process-2 terminated
```

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.

Sol –

Task3.py > ...

```python
from multiprocessing import Process, Pipe

def child_process(conn):
    message = conn.recv()
    print("Child received:", message)
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()

    p = Process(target=child_process, args=(child_conn,))
    p.start()

    parent_conn.send("Hello from parent process!")
    parent_conn.close()

    p.join()
```

Output –

```
PS C:\Users\riyas\OneDrive\Desktop\LabSheet 4> python -u "c:\Users\riyas\OneDrive\Desktop\LabSheet 4\Task3.py"
Child received: Hello from parent process!
```

Task 5: CPU Scheduling Algorithms

Implement FCFS, SJF, Round Robin, and Priority Scheduling algorithms in Python to calculate WT and TAT.

Sol – FSCS

FCFS.py > ...

```python
def fcfs(processes):

    processes.sort(key=lambda x: x[1])

    total_wt = 0
    total_tat = 0
    current_time = 0

    print("\n--- FCFS Scheduling ---")
    print("PID\tAT\tBT\tWT\tTAT")

    for p in processes:
        pid, at, bt, pr = p

        if current_time < at:
            current_time = at

        wt = current_time - at
        tat = wt + bt

        current_time += bt

        total_wt += wt
        total_tat += tat

        print(f"{pid}\t{at}\t{bt}\t{wt}\t{tat}")

    print("\nAverage WT =", total_wt / len(processes))
    print("Average TAT =", total_tat / len(processes))

if __name__ == "__main__":
    n = int(input("Enter number of processes: "))

    processes = []
    for i in range(n):
        pid = int(input(f"Enter PID for Process {i+1}: "))
        at = int(input("Enter Arrival Time: "))
        bt = int(input("Enter Burst Time: "))
        pr = 0
        processes.append([pid, at, bt, pr])

    fcfs(processes)
```

Output –

```
--- FCFS Scheduling ---
PID     AT      BT      WT      TAT
2       1       2       0       2
5       2       1       1       2
1       3       4       1       5
2       1       2       0       2
5       2       1       1       2
1       3       4       1       5

Average WT = 0.6666666666666666
1       3       4       1       5

Average WT = 0.6666666666666666

Average WT = 0.6666666666666666
Average TAT = 3.0
```