Assignment 4

Course Code & Name: ENCS351 /Operating System

Name: jai

Roll no: 2301420045

Introduction

This report presents the implementation and analysis of five system-level tasks related to system calls, virtual machine detection, file system operations, and CPU scheduling using Python, Bash, and C.

The objective was to practically understand how operating systems handle process management, system calls, inter-process communication, system information extraction, and CPU scheduling.

Each task demonstrates an essential OS concept through hands-on coding and execution in a Linux environment**.**

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

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.

Task 1: Batch Processing Simulation (Python)

Objective:

To simulate batch processing by executing multiple Python scripts sequentially using the subprocess module.

Concept Used:

Batch processing refers to executing a group of jobs without user interaction. The OS schedules them one after another, similar to early batch operating systems.

Working:

- The script prints the name of each file being executed.
- Each script runs independently as a subprocess.
- Execution order is strictly sequential, simulating batch processing.

Outcome:

Successfully executed multiple Python scripts in order, validating batch job behavior.

```
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system$ cd assignment_4
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$ ls
1.py  2.py  3.py  file1.txt  system_details_task4.sh  task1.py  task2.py  task3.py  task4.py  task5.py
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$ python task1.py
executinng 1.py....
hello this is  the first script file
executinng 2.py....
this is the second script file
executinng 3.py....
this is the third script file
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$
```

Task 2: System Startup and Logging

Objective:

To mimic OS boot and shutdown processes by creating multiple processes and logging their start/termination in a log file.

Concept Used:

- Multiprocessing for parallel process creation
- Logging module for system logs
- Process start → execution → termination lifecycle

Implementation Summary:

- Two processes were created using multiprocessing.Process().
- Logging was configured to write into system_log.txt.
- Each process logs:
    - Process start
    - Execution delay (simulated using time.sleep())
    - Process termination

Outcome:

A realistic simulation of OS process boot-up logging, similar to /var/log/boot.log and systemd journals.

```
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system$ cd assignment_4
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$ ls
1.py  2.py  3.py  file1.txt  system_details_task4.sh  task1.py  task2.py  task3.py  task4.py  task5.py
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$ python task1.py
executinng 1.py....
hello this is  the first script file
executinng 2.py....
this is the second script file
executinng 3.py....
this is the third script file
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$ python task2.py
system booting
system shutdown
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$
```

```
sjai5803@Jai:/mnt/c/Users/sjai5/Desktop/assignment_2$ ls
2301420045_assignment2.py  process_log.txt
sjai5803@Jai:/mnt/c/Users/sjai5/Desktop/assignment_2$ cat process_log.txt
2025-09-22 09:30:33,491 - MainProcess - System startup initiated
2025-09-22 09:30:33,502 - Process-2 - process-2 started
2025-09-22 09:30:33,504 - Process-1 - process-1 started
2025-09-22 09:30:35,505 - Process-2 - process-2 ended
2025-09-22 09:30:35,505 - Process-1 - process-1 ended
2025-09-22 09:30:35,508 - MainProcess - System shutdown complete
sjai5803@Jai:/mnt/c/Users/sjai5/Desktop/assignment_2$
```

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.

Objective:

To demonstrate low-level OS operations such as:

- Process creation using fork()
- Execution of new programs using exec()
- Process synchronization with wait()
- Inter-Process Communication (IPC) using pipes

Concept Used:

A pipe is created between parent and child. The parent writes a message; the child reads it.

Implementation Summary:

- os.pipe() creates a read and write file descriptor.
- os.fork() splits the program into parent and child.
- Parent writes "Hello from parent".
- Child receives and prints the message.
- Parent waits using os.wait().

Outcome:

Successfully demonstrated how Unix-based OS handles:

- Process duplication
- File descriptor sharing
- IPC via pipes
- Clean termination using wait

This mirrors real mechanisms in Linux kernel for parent-child communication

```
system shutdown
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$ python task3.py
chile recieved Hello from  parent
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$
```

Task 4: VM Detection and Shell Interaction

Objective:

1. Create a shell script to display system details.
2. Write a Python script to check whether the system is running inside a Virtual Machine.

Concept Used:

- uname, whoami, and lscpu commands
- Virtualization flags inside CPU hardware info
- Using Python's platform and system command execution

Shell Script Output Components:

- Kernel Version
- Active User
- Virtualization Technology Detection

Common checks:

- Presence of virtualization strings (e.g., *KVM, VMware, VirtualBox*)
- CPU flags
- BIOS vendor information

Outcome:

Both scripts worked successfully. The shell script interacted with OS utilities, while Python VM detection correctly identified virtualization environments using hardware-level clues.

```
  GNU nano 7.2                                                  sys
#!/bin/bash
echo "Kernel Version:"
uname -r
echo "User:"
whoami
echo "Hardware Info:"
lscpu | grep 'Virtualization'
```

Task 5: CPU Scheduling Algorithms

Objective:

Implement classical CPU scheduling algorithms and compute:

- Waiting Time (WT)
- Turnaround Time (TAT)

Concept Used:

CPU scheduling is a core OS function that determines which process runs next. These algorithms affect CPU utilization, throughput, and fairness.

Outcome:

- Implementations correctly calculated WT and TAT.
- Algorithms behaved as expected:
    - FCFS favored arrival order
    - SJF minimized average waiting time
    - RR ensured fairness
    - Priority scheduling selected tasks based on priority level

This task strengthened understanding of how OS kernels manage process queues.

```
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$ python task5.py

========= CPU Scheduling Results =========

--- FCFS Scheduling ---
PID     AT      BT      WT      TAT
1       0       6       0       6
2       1       8       5       13
3       2       7       12      19
4       3       3       18      21

Average WT = 8.75
Average TAT = 14.75

--- SJF (Non-Preemptive) Scheduling ---
PID     AT      BT      WT      TAT
1       0       6       0       6
2       1       8       15      23
3       2       7       7       14
4       3       3       3       6

Average WT = 6.25
Average TAT = 12.25
```

```
--- Round Robin Scheduling (q = 3 ) ---
PID     AT      BT      WT      TAT
1       0       6       9       15
2       1       8       14      22
3       2       7       15      22
4       3       3       6       9

Average WT = 11.00
Average TAT = 17.00

--- Priority Scheduling ---
PID     AT      BT      PR      WT      TAT
1       0       6       2       0       6
2       1       8       1       5       13
3       2       7       3       12      19
4       3       3       4       18      21

Average WT = 8.75
Average TAT = 14.75
sjai5803@Jai:/mnt/c/Users/sjai5/Operating_system/assignment_4$
```

Conclusion

This lab assignment provided hands-on exposure to how operating systems manage processes, interact with hardware, perform IPC, and schedule tasks.
Through Python, C, and Shell scripting, the tasks recreated real OS-level behaviors such as:

- Process lifecycle control
- Logging mechanisms
- System calls and IPC
- Virtualization detection
- CPU scheduling strategies

Each task successfully met the learning objectives and reinforced practical understanding of OS internals, bridging theoretical concepts with real execution.