

# Operating Systems Lab Assignment - 4 Report

**Name:** Harsh Sen

**Roll no.** 2301730194

**Course Code:** ENCS351

**Program:** B.Tech CSE(AI/ML)

## **Experiment: System Calls, VM Detection, and File System Operations using Python**

### **Objective:**

- Executing multiple Python scripts using batch processing
- Simulating system startup and shutdown processes with logging
- Demonstrating system calls such as fork(), exec(), wait(), and pipe()
- Detecting virtual machine environments programmatically
- Using Python to simulate basic file system operations and CPU scheduling

This lab reinforces operating system internals through hands-on scripting and process-level simulations.

### **Tools Used:**

Python 3.x

Linux Environment (Ubuntu/WSL via Terminal)

Bash Shell

Python modules: os, platform, subprocess, multiprocessing, logging, time

C Programming (for fork/exec/wait)

## Tasks Performed:

- **Task 1(Batch Processing Simulation (Python)):**

**Description:** A Python script was written to execute multiple .py files sequentially, simulating batch execution systems used in early operating systems.

**Code Behavior:**

- Stores script names in a list
- Uses subprocess.call() to execute each script sequentially
- Prints execution status for clarity

**Outcome:**

Successfully executed multiple Python files in sequence, simulating batch processing.

**INPUT:**

```
os4task1.py > ...
1  import subprocess
2  scripts = ['script1.py', 'script2.py', 'script3.py']
3  for script in scripts:
4      print(f"Executing {script}...")
5      subprocess.call(['python3', script])
```

**OUTPUT:**

```
Executing script1.py...
Executing script2.py...
Executing script3.py...
```

- **Task 2 (System Startup and Logging):**

**Description:** Simulated OS boot sequence using multiple child processes. Logging was implemented using Python's logging module.

### Key Actions:

- Created multiple processes using multiprocessing.Process
- Recorded start and termination messages with timestamps
- Stored logs in system\_log.txt

### Outcome:

Accurate logging of process lifecycle events, mimicking real OS startup logs.

### INPUT:

```
os4task2.py > ...
1  import multiprocessing
2  import logging
3  import time
4  logging.basicConfig(filename='system_log.txt', level=logging.INFO,
5                      format='%(asctime)s - %(processName)s - %(message)s')
6  def process_task(name):
7      logging.info(f"{name} started")
8      time.sleep(2)
9      logging.info(f"{name} terminated")
10 if __name__ == '__main__':
11     print("System Booting...")
12     p1 = multiprocessing.Process(target=process_task, args=("Process-1",))
13     p2 = multiprocessing.Process(target=process_task, args=("Process-2",))
14     p1.start()
15     p2.start()
16     p1.join()
17     p2.join()
18     print("System Shutdown.")
```

### OUTPUT:

```
System Booting...
System Shutdown.
```

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

**Description:** Using Python's os module, simulated core system calls:

**fork()**

Creates a child process.

**exec()**

Replaces process memory with a new program.

**wait()**

Parent waits for child to finish.

**pipe()**

Used to send data between parent and child.

**Flow:**

- Parent writes "Hello from parent" into pipe
- Child reads message and prints it
- Demonstrates unidirectional IPC

**Outcome:**

Successful demonstration of system call usage and IPC using pipes.

**INPUT:**

```
os4task3.py > ...
1  import os
2  r, w = os.pipe()
3  pid = os.fork()
4  if pid > 0:
5      os.close(r)
6      os.write(w, b"Hello from parent")
7      os.close(w)
8      os.wait()
9  else:
10     os.close(w)
11     message = os.read(r, 1024)
12     print("Child received:", message.decode())
13     os.close(r)
```

## OUTPUT:

```
Child received: Hello from parent
```

- **Task 4 (VM Detection and Shell Interaction):**

**Description:** Created a Bash script to print kernel version, username, and hardware information.

A Python script was also implemented to detect virtualization through:

- Checking CPU flags
- Reading /proc/cpuinfo
- Searching for virtualization indicators

**Outcome:**

Correctly identified whether the system was running in a virtualized environment (like VMware, VirtualBox, WSL).

## INPUT:

```
os4task4.py > ...
1  import subprocess
2  import uuid
3  def check_cpu_flags():
4      """Check if 'hypervisor' flag is present in /proc/cpuinfo."""
5      try:
6          output = subprocess.check_output("cat /proc/cpuinfo", shell=True).decode()
7          if "hypervisor" in output:
8              return True
9      except:
10         pass
11     return False
12 def check_dmi_data():
13     """Check system manufacturer and product names for VM indicators."""
14     indicators = ["VirtualBox", "VMware", "KVM", "QEMU", "Hyper-V", "Xen", "Bochs"]
15     try:
16         product = subprocess.check_output("cat /sys/class/dmi/id/product_name", shell=True).decode().strip()
17         vendor = subprocess.check_output("cat /sys/class/dmi/id/sys_vendor", shell=True).decode().strip()
18         for item in indicators:
19             if item.lower() in product.lower() or item.lower() in vendor.lower():
20                 return True
21     except:
22         pass
23     return False
24 def check_mac_address():
25     """Check if MAC address belongs to known virtual NIC ranges."""
26     vm_mac_prefixes = {
27         "00:05:69", "00:0C:29", "00:1C:14",
28         "08:00:27",
29         "52:54:00",
30     }
31     mac = ':'.join(['{:02x}'.format((uuid.getnode() >> ele) & 0xff)
32                     for ele in range(40, -1, -8)])
33     mac_prefix = mac.upper()[0:8]
34     return mac_prefix in vm_mac_prefixes
35 def is_virtual_machine():
36     if check_cpu_flags():
37         return True
38     if check_dmi_data():
39         return True
40     if check_mac_address():
41         return True
```

```

42     return False
43 if __name__ == "__main__":
44     if is_virtual_machine():
45         print("⚠ This system appears to be running inside a Virtual Machine.")
46     else:
47         print("✅ This system appears to be running on Physical Hardware.")

```

```

echo "Kernel Version:"
uname -r
echo "User:"
whoami
echo "Hardware Info:"
lscpu | grep 'Virtualization'
Kernel Version:
5.15.167.4-microsoft-standard-WSL2
User:
gunjan_linux_2025
Hardware Info:
Virtualization:                AMD-V
Virtualization type:           full

```

## OUTPUT:

```

⚠ This system appears to be running inside a Virtual Machine.

```

```

echo "Kernel Version:"
uname -r
echo "User:"
whoami
echo "Hardware Info:"
lscpu | grep 'Virtualization'
Kernel Version:
5.15.167.4-microsoft-standard-WSL2

```

- **Task 5 (CPU Scheduling Algorithms):**

**Description:** Using existing codes from Lab 3, implemented:

- First-Come First-Served (FCFS)
- Shortest Job First (SJF)
- Round Robin
- Priority Scheduling

**Outcome:**

Accurate scheduling results, illustrating differences between scheduling algorithms.

**INPUT:**

```
os4task5.py > ...
1  # Priority Scheduling Simulation
2  processes = []
3  n = int(input("Enter number of processes: "))
4  for i in range(n):
5      bt = int(input(f"Enter Burst Time for P{i+1}: "))
6      pr = int(input(f"Enter Priority (lower number = higher priority) for P{i+1}: "))
7      processes.append((i+1, bt, pr))
8  processes.sort(key=lambda x: x[2])
9  wt = 0
10 total_wt = 0
11 total_tt = 0
12 print("\nPriority Scheduling:")
13 print("PID\tBT\tPriority\tWT\tTAT")
14 for pid, bt, pr in processes:
15     tat = wt + bt
16     print(f"{pid}\t{bt}\t{pr}\t\t{wt}\t{tat}")
17     total_wt += wt
18     total_tt += tat
19     wt += bt
20 print(f"Average Waiting Time: {total_wt / n}")
21 print(f"Average Turnaround Time: {total_tt / n}")
```

## OUTPUT:

```
Enter number of processes: 4
Enter Burst Time for P1: 10
Enter Priority (lower number = higher priority) for P1: 2
Enter Burst Time for P2: 5
Enter Priority (lower number = higher priority) for P2: 0
Enter Burst Time for P3: 8
Enter Priority (lower number = higher priority) for P3: 1
Enter Burst Time for P4: 6
Enter Priority (lower number = higher priority) for P4: 3

Priority Scheduling:
PID    BT    Priority    WT    TAT
2      5      0          0      5
3      8      1          5     13
1     10      2         13     23
4      6      3         23     29
Average Waiting Time: 10.25
Average Turnaround Time: 17.5
```

## Outputs:

- Executed batch processing simulation successfully
- Generated system\_log.txt with accurate timestamps
- Verified inter-process communication via pipes
- Successfully detected VM environment characteristics
- Produced correct scheduling outputs for all CPU scheduling algorithms

## Learning Outcomes:

- System Calls: Gained practical experience with low-level OS functions like fork, exec, wait, and pipe.
- Process Management: Understood how OS handles startup, shutdown, and logging.
- Batch Processing: Reinforced understanding of early OS execution models.
- VM Detection: Learned how virtualization leaves identifiable system patterns.

- Scheduling Algorithms: Improved clarity on CPU scheduling effectiveness and performance differences.