K. R. Mangalam University

School of Engineering and Technology

Operating System Lab File

Course Code: ENCS351

Submitted by

Name: Nidhi Kumari

Roll No.: 2401011484

Program: BTech CSE

Section: 6-F

Session: 2025-26

Date: 02-DEC,2025

Submitted To

Dr. Satinder Pal Singh

Index

Name – Nidhi Kumari

Roll No – 2401011482

Lab Experiment Sheet-1

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

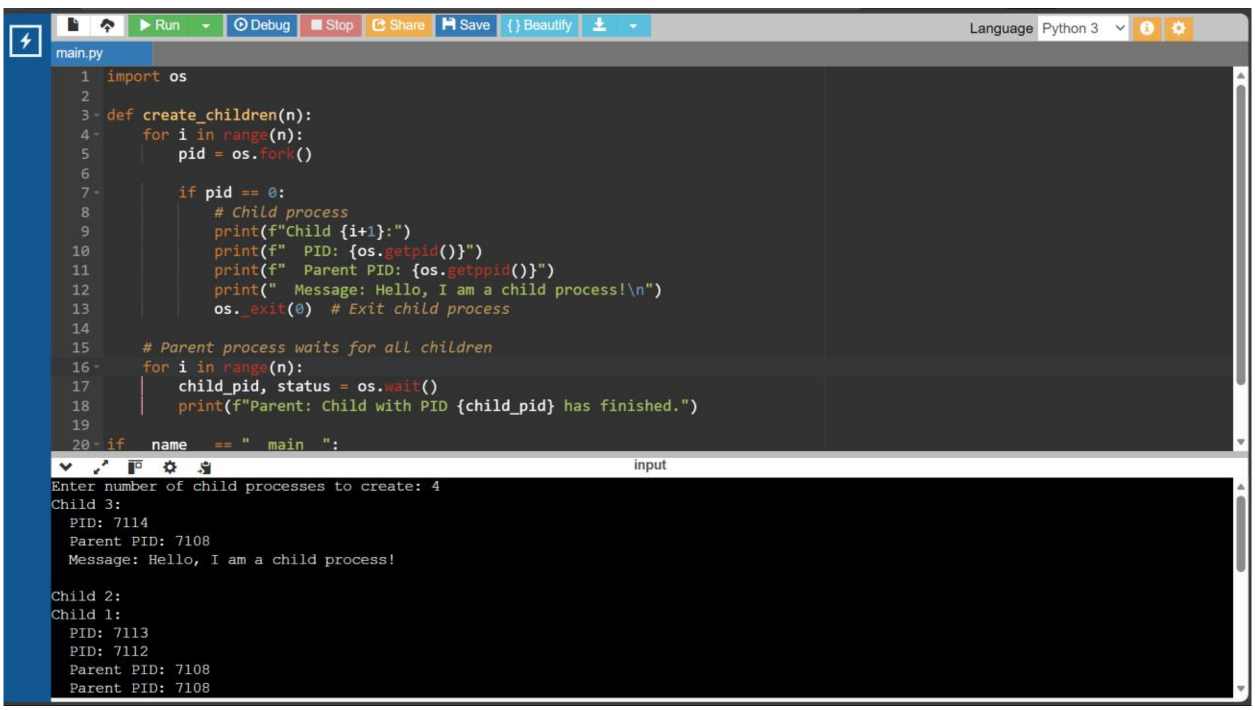
The parent should wait for all children using os.wait().

CODE (PYTHON):

```
import os


def create_children(n):
    for i in range(n):
        pid = os.fork()

        if pid == 0
            print(f"Child {i+1}:")
            print(f"  PID: {os.getpid()}")
            print(f"  Parent PID: {os.getppid()}")
            print("  Message: Hello, I am a child process!\n")
            os._exit(0)  # Exit child process
```

```
    for i in range(n):
        child_pid, status = os.wait()
        print(f"Parent: Child with PID {child_pid} has finished.")

if __name__ == "__main__":
    N = int(input("Enter number of child processes to create: "))
    create_children(N)
```

OUTPUT:



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

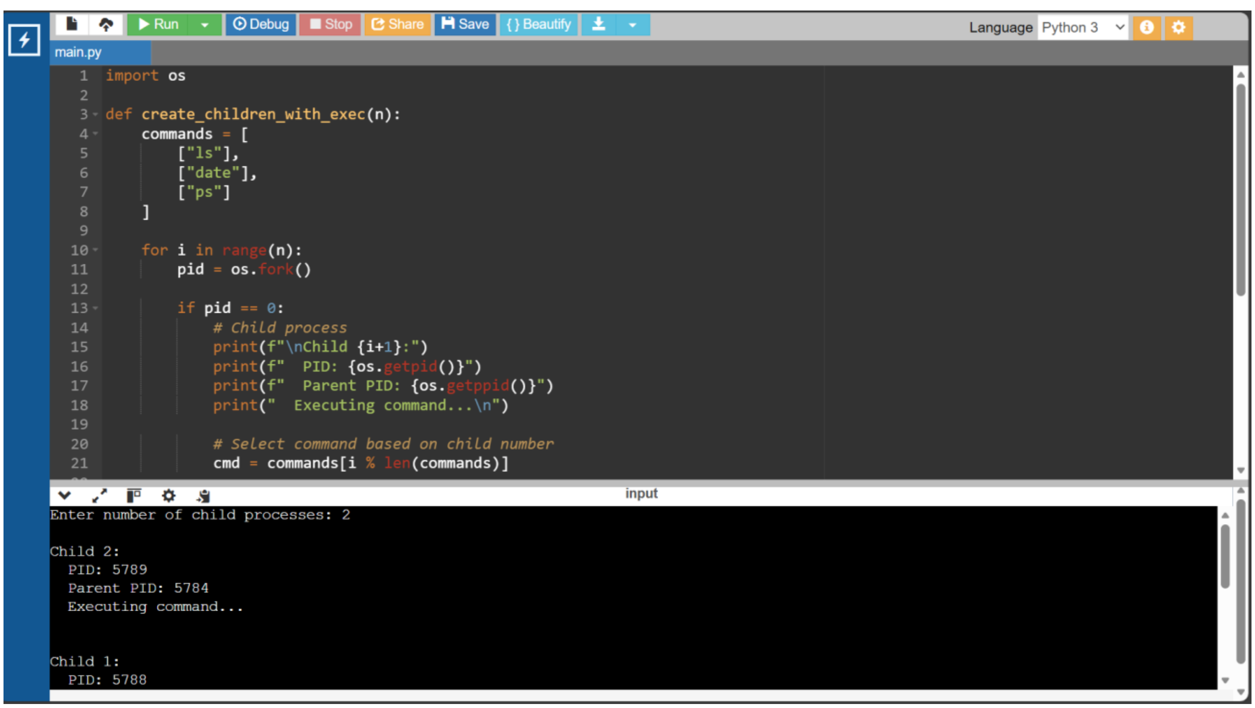
subprocess.run().

CODE(PYTHON):

```
import os


def create_children_with_exec(n):
    commands = [
        ["ls"],
        ["date"],
        ["ps"]
    ]


    for i in range(n):
        pid = os.fork()


        if pid == 0:
            # Child process
            print(f"\nChild {i+1}:")
            print(f"  PID: {os.getpid()}")
            print(f"  Parent PID: {os.getppid()}")
            print("  Executing command...\n
```

```
            cmd = commands[i % len(commands)]

            os.execvp(cmd[0], cmd)

            print("exec failed!")
            os._exit(1)
    for i in range(n):
        child_pid, status = os.wait()
        print(f"\nParent: Child with PID {child_pid} finished.")

if __name__ == "__main__":
    N = int(input("Enter number of child processes: "))
    create_children_with_exec(N)
```

OUTPUT:



Task 3: Zombie & Orphan Processes

Zombie: Fork a child and skip wait() in the parent.

Orphan: Parent exits before the child finishes.

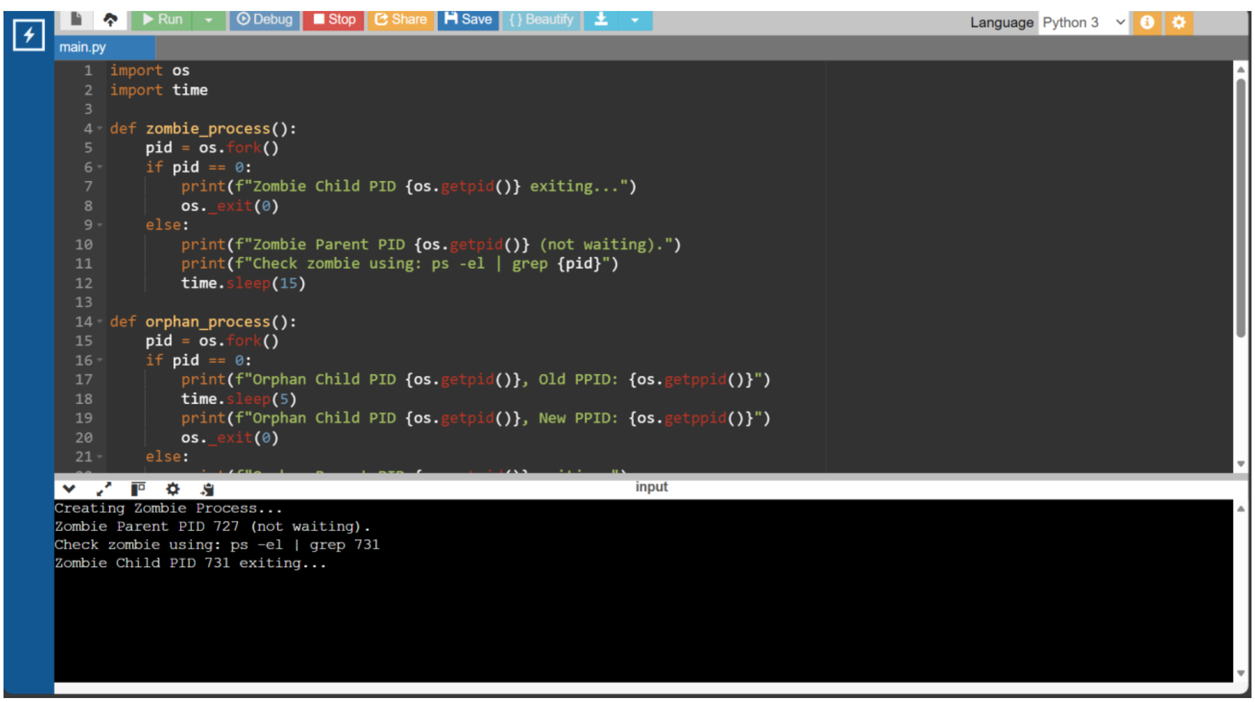
Use ps -el | grep defunct to identify zombies.

CODE(PYTHON):

```
import os

import time


def zombie_process():

    pid = os.fork()

    if pid == 0:

        print(f"Zombie Child PID {os.getpid()} exiting...")

        os._exit(0)

    else:

        print(f"Zombie Parent PID {os.getpid()} (not waiting).")

        print(f"Check zombie using: ps -el | grep {pid}")

        time.sleep(15)


def orphan_process():

    pid = os.fork()

    if pid == 0:

        print(f"Orphan Child PID {os.getpid()}, Old PPID: {os.getppid()}")

        time.sleep(5)
```

```
        print(f"Orphan Child PID {os.getpid()}, New PPID: {os.getppid()}")
        os._exit(0)
    else:
        print(f"Orphan Parent PID {os.getpid()} exiting.")
        os._exit(0)


if __name__ == "__main__":
    print("Creating Zombie Process...")
    zombie_process()
    time.sleep(2)
    print("\nCreating Orphan Process...")
    orphan_process()
```

OUTPUT:



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

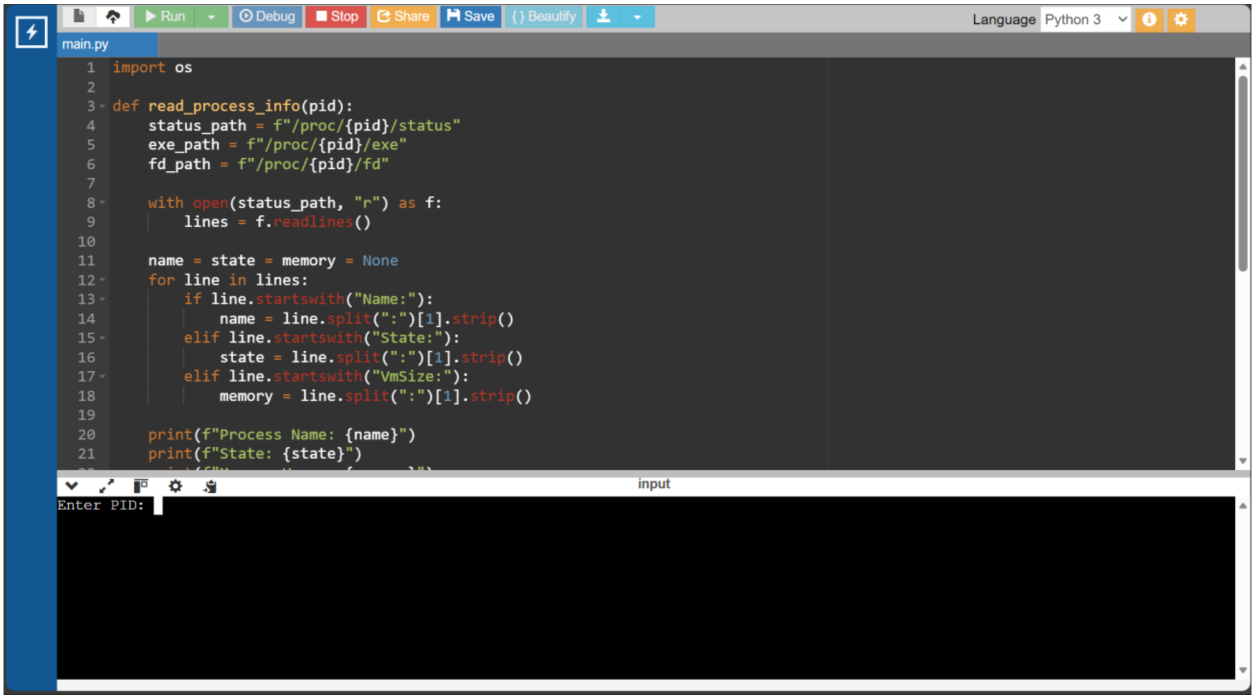
- Open file descriptors from /proc/[pid]/fd

CODE(PYTHON):

```
import os


def read_process_info(pid):
    status_path = f"/proc/{pid}/status"
    exe_path = f"/proc/{pid}/exe"
    fd_path = f"/proc/{pid}/fd"

    with open(status_path, "r") as f:
        lines = f.readlines()

    name = state = memory = None
    for line in lines:
        if line.startswith("Name:"):
            name = line.split(":")[1].strip()
        elif line.startswith("State:"):
            state = line.split(":")[1].strip()
        elif line.startswith("VmSize:"):
```

```
            memory = line.split(":")[1].strip()

    print(f"Process Name: {name}")
    print(f"State: {state}")
    print(f"Memory Usage: {memory}")

    try:
        exe = os.readlink(exe_path)
        print(f"Executable Path: {exe}")
    except:
        print("Executable Path: Not accessible")

    print("Open File Descriptors:")
    try:
        for fd in os.listdir(fd_path):
            link = os.readlink(os.path.join(fd_path, fd))
            print(f"  FD {fd} -> {link}")
    except:
        print("  Cannot access file descriptors")

if __name__ == "__main__":
    pid = input("Enter PID: ")
    read_process_info(pid)
```

OUTPUT:



Task 5: Process Prioritization

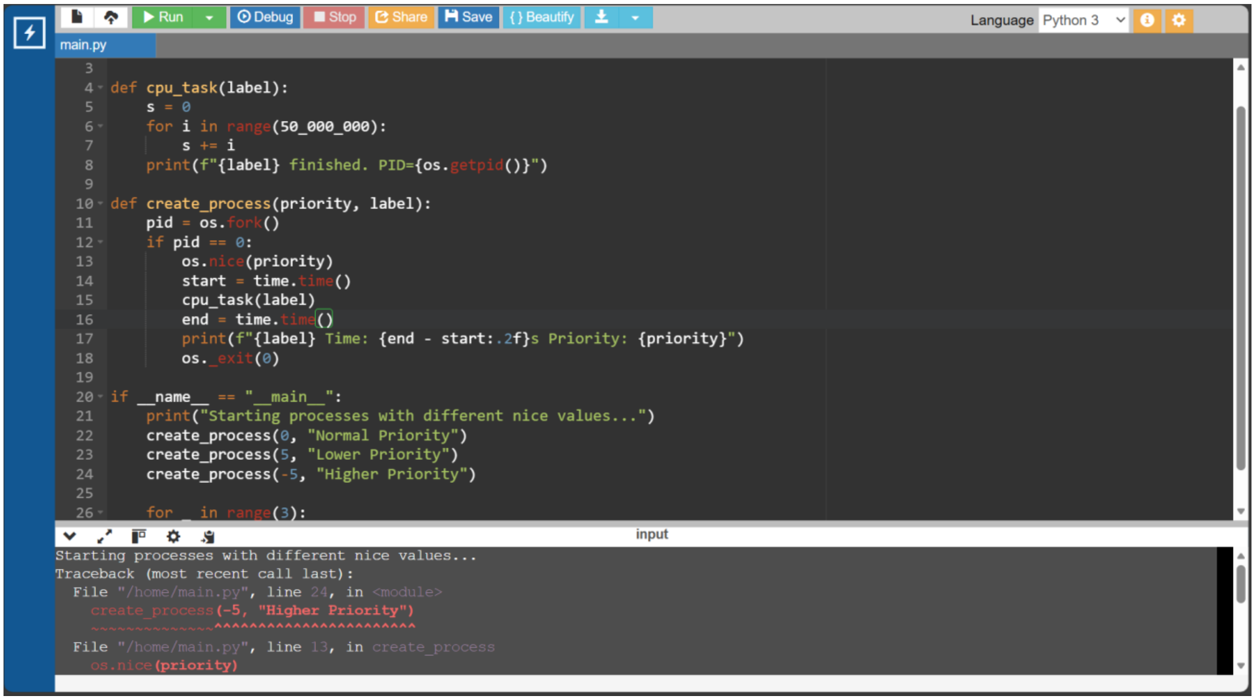
Create multiple CPU-intensive child processes. Assign different nice() values. Observe and log execution order to show scheduler impact.

CODE(PYTHON):

```
import os
import time


def cpu_task(label):
    s = 0
    for i in range(50_000_000):
        s += i
```

```
    print(f"{label} finished. PID={os.getpid()}")


def create_process(priority, label):
    pid = os.fork()
    if pid == 0:
        os.nice(priority)
        start = time.time()
        cpu_task(label)
        end = time.time()
        print(f"{label} Time: {end - start:.2f}s Priority: {priority}")
        os._exit(0)


if __name__ == "__main__":
    print("Starting processes with different nice values...")
    create_process(0, "Normal Priority")
    create_process(5, "Lower Priority")
    create_process(-5, "Higher Priority")

    for _ in range(3):
        os.wait()
```

OUTPUT:



```
def cpu_task(label):
    s = 0
    for i in range(50_000_000):
        s += i
    print(f"{label} finished. PID={os.getpid()}")

def create_process(priority, label):
    pid = os.fork()
    if pid == 0:
        os.nice(priority)
        start = time.time()
        cpu_task(label)
        end = time.time()
        print(f"{label} Time: {end - start:.2f}s Priority: {priority}")
        os._exit(0)

if __name__ == "__main__":
    print("Starting processes with different nice values...")
    create_process(0, "Normal Priority")
    create_process(5, "Lower Priority")
    create_process(-5, "Higher Priority")

    for _ in range(3):
```

```
Starting processes with different nice values...
Traceback (most recent call last):
  File "/home/main.py", line 24, in <module>
    create_process(-5, "Higher Priority")
    ~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/main.py", line 13, in create_process
    os.nice(priority)
```

Assignment 2

Nidhi Kumari

2401011484

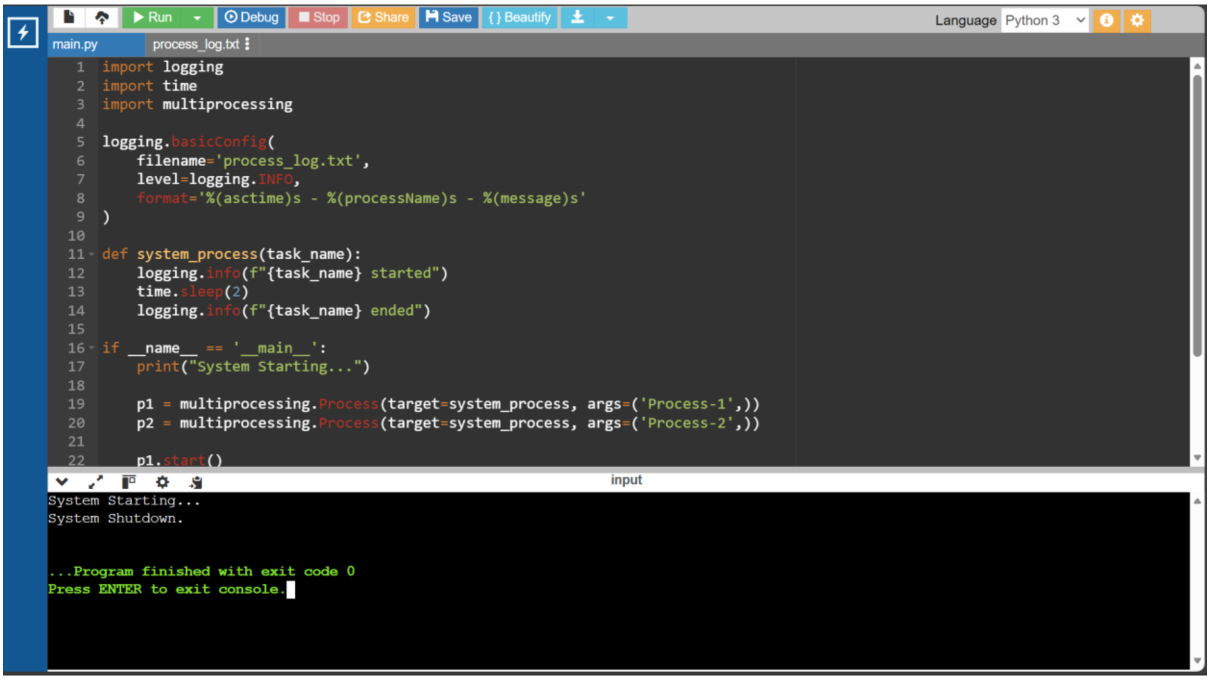
Experiment Title: System Startup, Process Creation, and Termination Simulation in Python

Implementation:

```
import multiprocessing

import time

import logging


# Setup logger

logging.basicConfig(

    filename='process_log.txt',

    level=logging.INFO,

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

)


# Dummy function to simulate a task

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

# Start processes
p1.start()
p2.start()

# Wait for processes to complete
p1.join()
p2.join()

print("System Shutdown.")
```

Output:



Nidhi Kumari

2401011484

LAB ASSIGNMENT-03

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

Task 1: CPU Scheduling with Gantt Chart

Write a Python program to simulate Priority and Round Robin scheduling

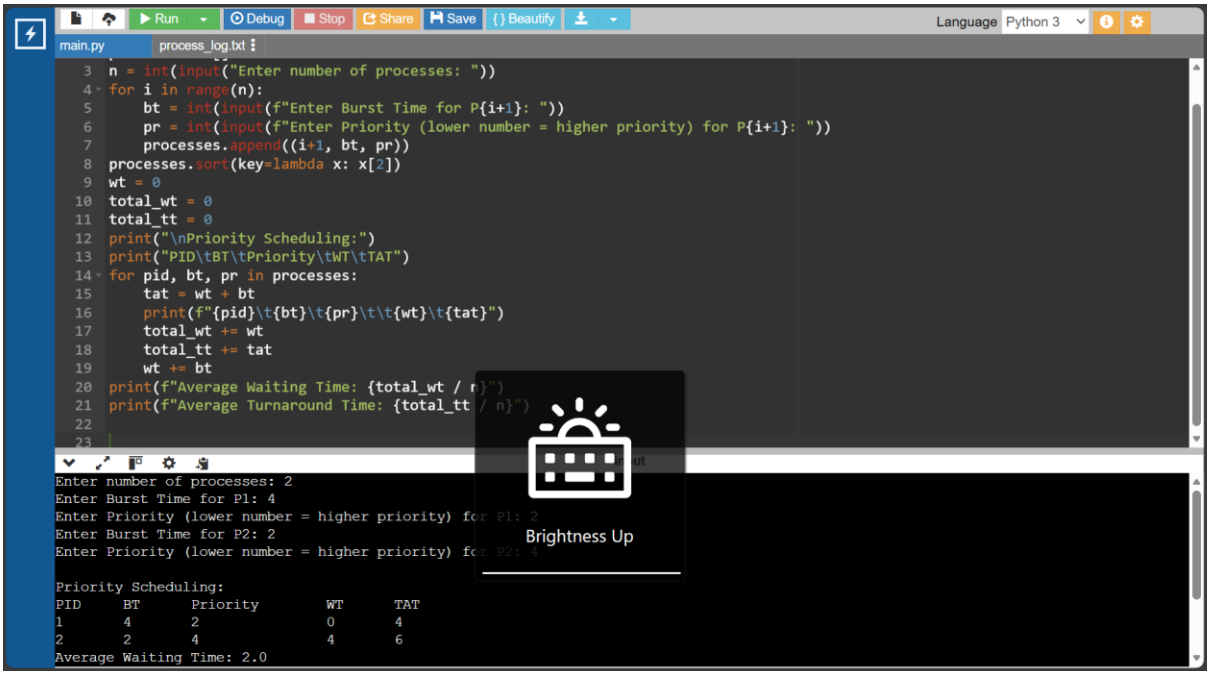
algorithms. Compute average waiting and turnaround times.

Implementation:

```
# Priority Scheduling Simulation
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:



Task 2: Sequential File Allocation

Write a Python program to simulate sequential file allocation strategy.

Implementation:

```
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):

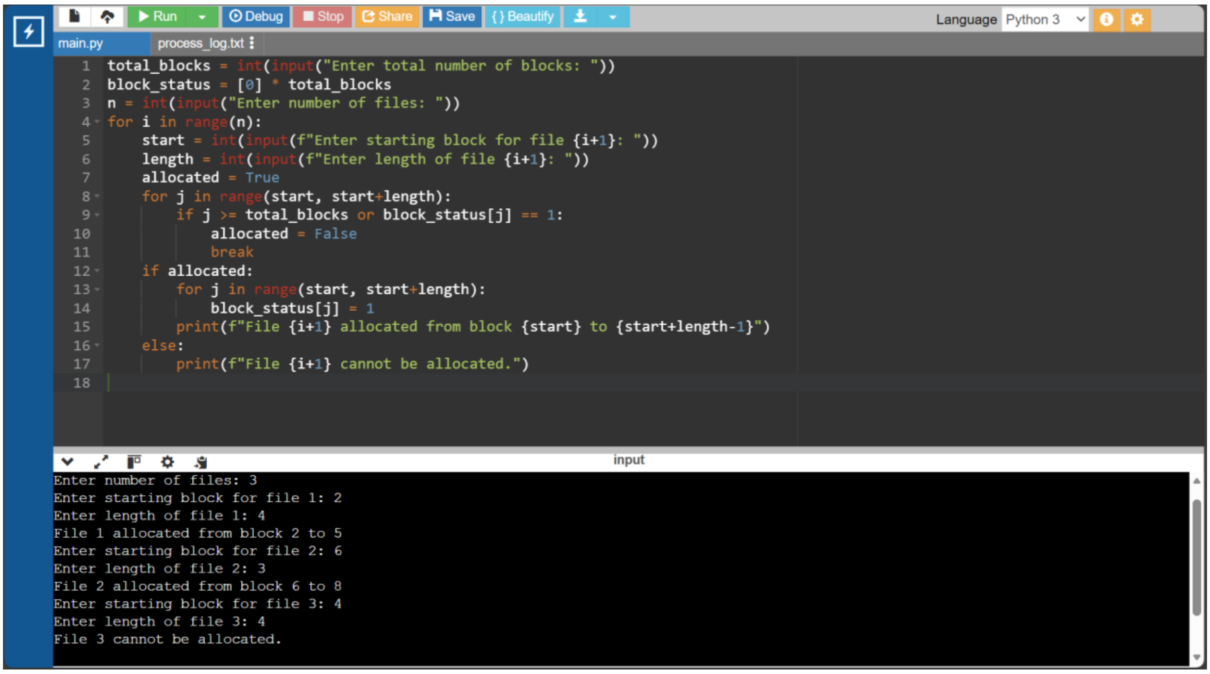
block_status[j] = 1

print(f"File {i+1} allocated from block {start} to {start+length-1}")

else:

print(f"File {i+1} cannot be allocated.")

Output:



Task 3: Indexed File Allocation

Write a Python program to simulate indexed file allocation strategy.

Implementation:

total_blocks = int(input("Enter total number of blocks: "))

block_status = [0] * total_blocks

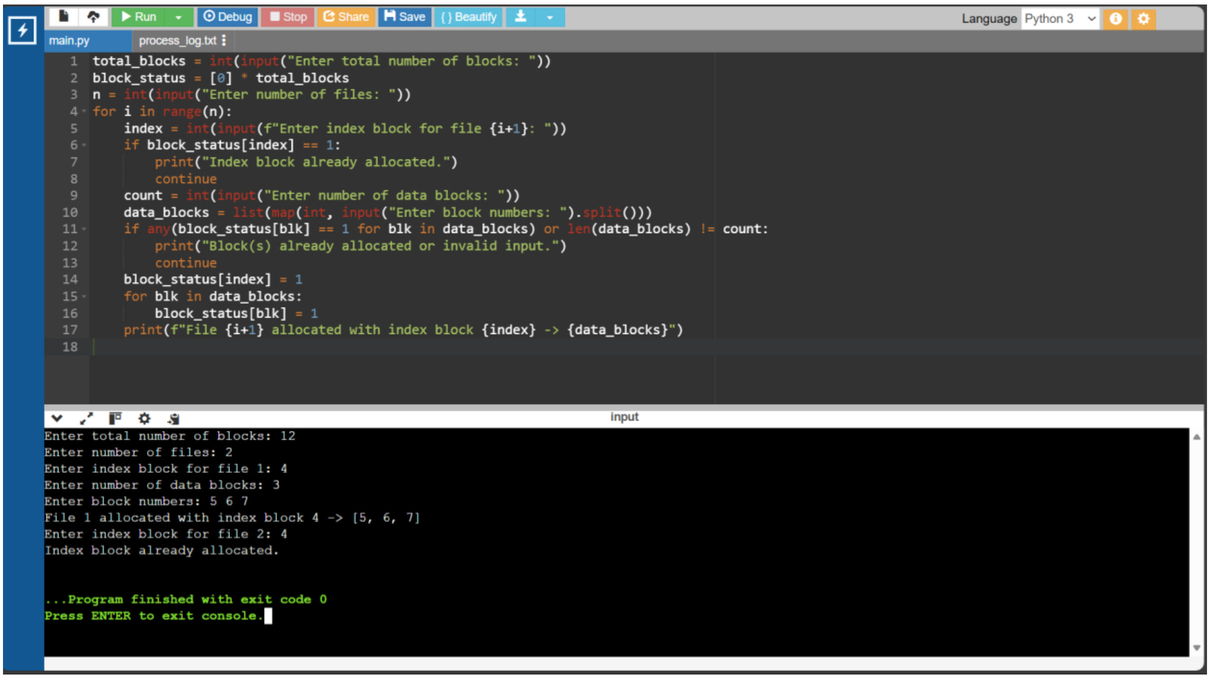
n = int(input("Enter number of files: "))

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:



Task 4: Contiguous Memory Allocation

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

Implementation:

```
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:

Task 5: MFT & MVT Memory Management

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

Implementation:

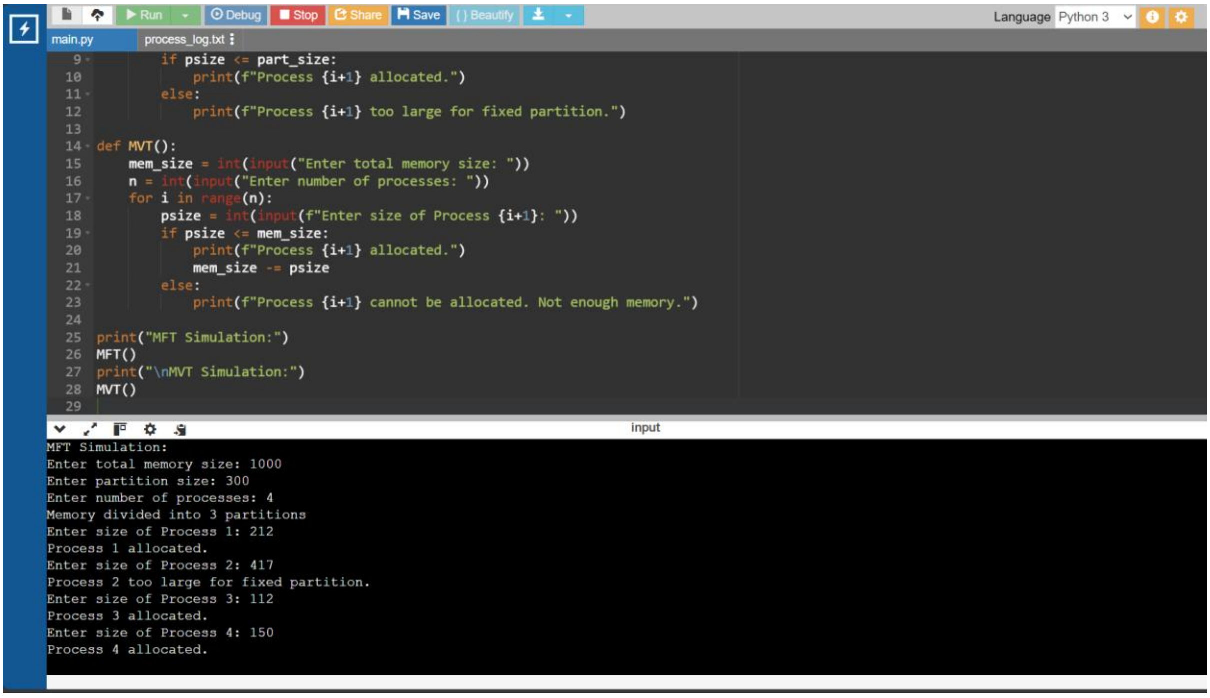
```
def MFT():
    mem_size = int(input("Enter total memory size: "))
    part_size = int(input("Enter partition size: "))
    n = int(input("Enter number of processes: "))
    partitions = mem_size // part_size
    print(f"Memory divided into {partitions} partitions")
    for i in range(n):
       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



Nidhi Kumari

2401011484

LAB ASSIGNMENT-04

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

Task 1: Batch Processing Simulation (Python)

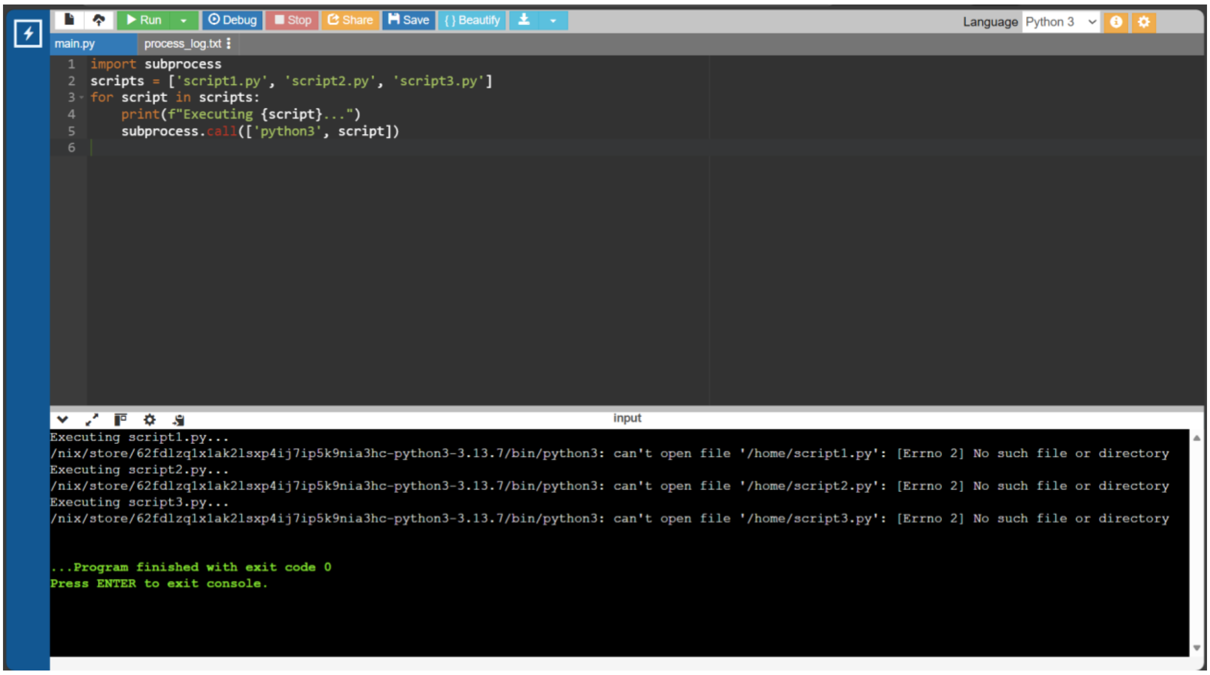
Write a Python script to execute multiple .py files sequentially, mimicking

batch processing.

Implementation:

```
import subprocess

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

for script in scripts:

    print(f"Executing {script}...")

    subprocess.call(['python3', script])
```

Output:



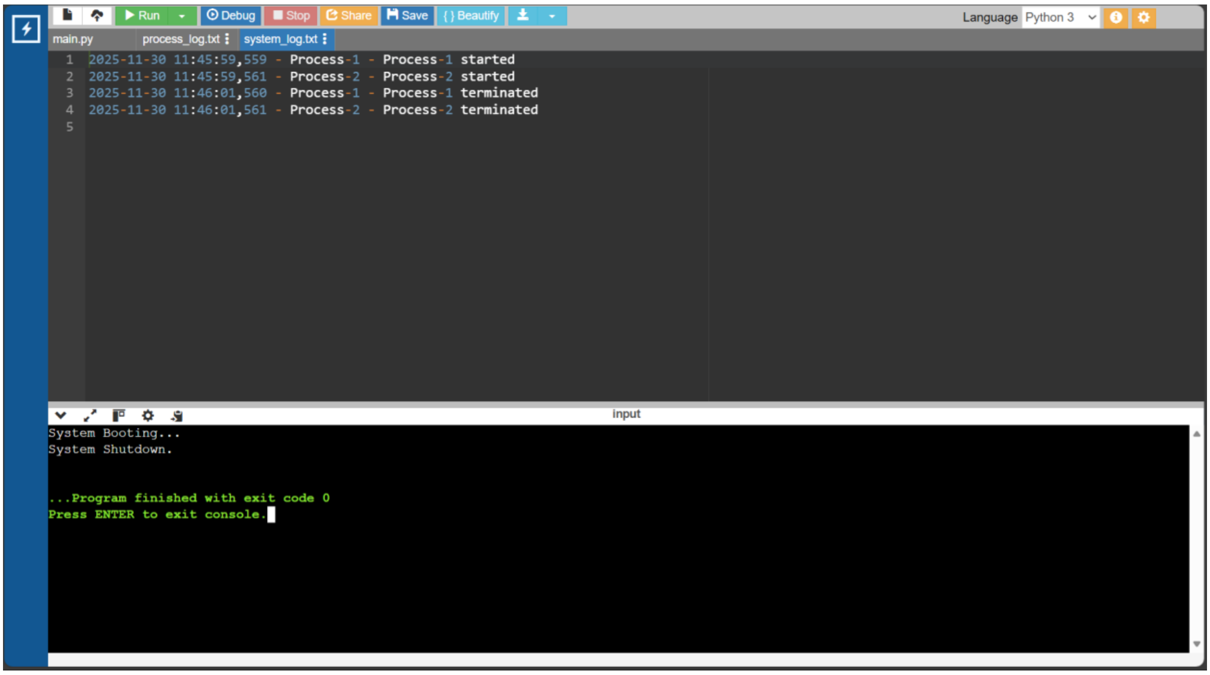
Task 2: System Startup and Logging

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

Implementation:

```
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...")

  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("System Shutdown.")
```

Output:



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

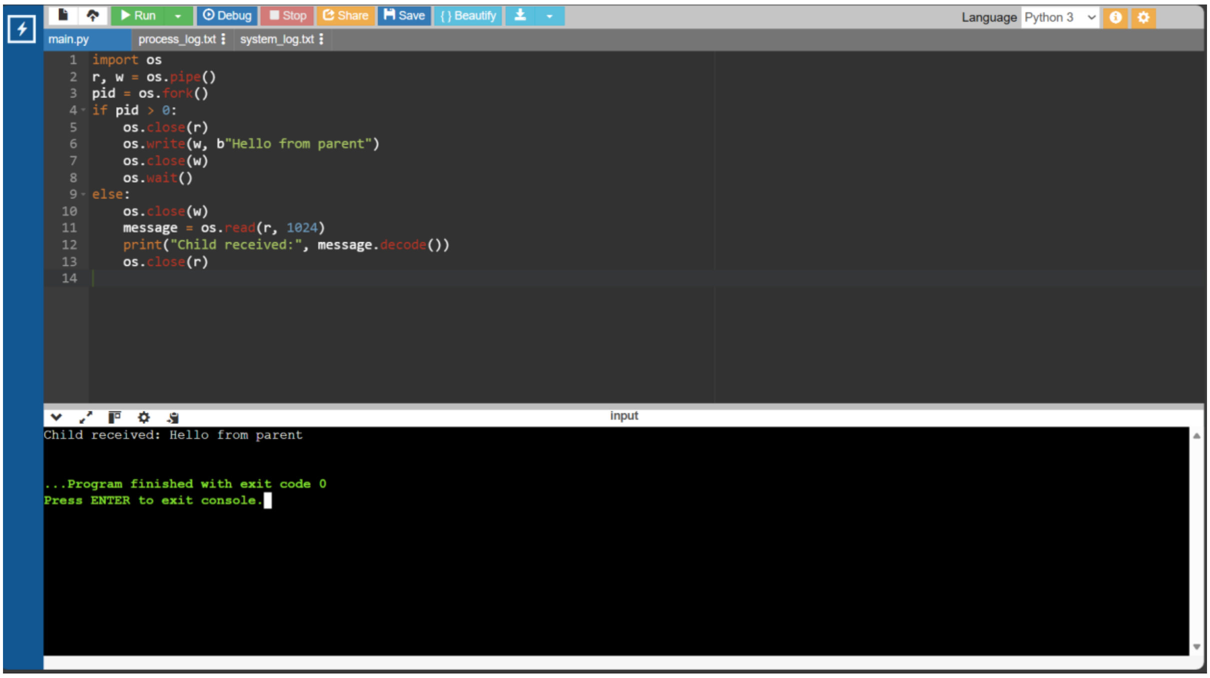
Use system calls (fork(), exec(), wait()) and implement basic Inter-Process Communication using pipes in C or Python.

Implementation:

```
import os
r, w = os.pipe()
pid = os.fork()
if pid > 0:
    os.close(r)
    os.write(w, b"Hello from parent")
    os.close(w)
    os.wait()
else:
    os.close(w)
    message = os.read(r, 1024)
    print("Child received:", message.decode())
```

os.close(r)

Output:



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.

Implementation:

#!/bin/bash

echo "Kernel Version:"

uname -r

echo "User:"

whoami

echo "Hardware Info:"

lscpu | grep 'Virtualization'

Python Script:

import os

```
import subprocess

def check_dmi():
    """Check system DMI data for known VM identifiers."""
    vm_signatures = ["virtual", "vmware", "kvm", "qemu", "hyper-v", "xen"]
    try:
        output = subprocess.check_output(["sudo", "dmidecode"],
            stderr=subprocess.DEVNULL).decode().lower()
        return any(sig in output for sig in vm_signatures)
    except:
        return False

def check_cpu_flags():
    """Check CPU flags for hypervisor bit."""
    try:
        with open("/proc/cpuinfo") as f:
            data = f.read().lower()
        return "hypervisor" in data
    except:
        return False

def check_mac_address():
    """Check if the MAC address belongs to a VM vendor."""
    vm_mac_prefixes = [
        "00:05:69", "00:0C:29", "00:1C:14",   # VMware
        "08:00:27",                   # VirtualBox
        "52:54:00",                   # QEMU / KVM
        "00:15:5D",                   # Hyper-V
    ]
    try:
        output = subprocess.check_output(["ip", "link"]).decode().lower()
```

```
        for prefix in vm_mac_prefixes:
            if prefix.lower() in output:
                return True
    except:
        pass
    return False

def detect_vm():
    print("\n--- Virtual Machine Detection ---")
    dmi = check_dmi()
    hypervisor_flag = check_cpu_flags()
    mac_vm = check_mac_address()
    if dmi or hypervisor_flag or mac_vm:
        print("This system appears to be running inside a VIRTUAL MACHINE.")
    else:
        print("This system appears to be running on BARE METAL hardware.")
    print("\nDetails:")
    print(f"DMI-based detection: {dmi}")
    print(f"CPU hypervisor flag: {hypervisor_flag}")
    print(f"MAC address virtual: {mac_vm}")

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

Output:

Task 5: CPU Scheduling Algorithms

Implement FCFS, SJF, Round Robin, and Priority Scheduling

algorithms in Python to calculate WT and TAT.

Implementation:

```
"""FCFS Scheduling"""
def fcfs(processes):
    processes.sort(key=lambda x: x['arrival'])
    time = 0
    for p in processes:
        if time < p['arrival']:
            time = p['arrival']
        p['wt'] = time - p['arrival']
        time += p['burst']
        p['tat'] = p['wt'] + p['burst']
    return processes

"""SJF Scheduling"""
def sjf(processes):
    processes = sorted(processes, key=lambda x: x['arrival'])
    completed, time = 0, 0
    n = len(processes)
    while completed < n:
        available = [p for p in processes if p['arrival'] <= time and 'done' not in p]
        if not available:
            time += 1
            continue
        p = min(available, key=lambda x: x['burst'])
        p['wt'] = time - p['arrival']
        time += p['burst']
        p['tat'] = p['wt'] + p['burst']
```

```
            p['done'] = True
            completed += 1
    return processes


"""Round Robin"""
def round_robin(processes, quantum):
    from collections import deque
    q = deque()
    time = 0
    remaining = {p['pid']: p['burst'] for p in processes}
    processes.sort(key=lambda x: x['arrival'])
    i = 0
    completed = 0
    n = len(processes)
    while completed < n:
        while i < n and processes[i]['arrival'] <= time:
            q.append(processes[i])
            i += 1
        if not q:
            time = processes[i]['arrival']
            continue
        p = q.popleft()
        exec_time = min(quantum, remaining[p['pid']])
        remaining[p['pid']] -= exec_time
        time += exec_time
        while i < n and processes[i]['arrival'] <= time:
            q.append(processes[i])
            i += 1
        if remaining[p['pid']] == 0:
            p['tat'] = time - p['arrival']
            p['wt'] = p['tat'] - p['burst']
```

```
            completed += 1
        else:
            q.append(p)
    return processes

"""Priority Scheduling"""
def priority_scheduling(processes):
    time = 0
    completed = 0
    n = len(processes)
    processes.sort(key=lambda x: x['arrival'])
    while completed < n:
        available = [p for p in processes if p['arrival'] <= time and 'done' not in p]
        if not available:
            time += 1
            continue
        p = min(available, key=lambda x: x['priority'])
        p['wt'] = time - p['arrival']
        time += p['burst']
        p['tat'] = p['wt'] + p['burst']
        p['done'] = True
        completed += 1
    return processes

processes = [
    {'pid': 1, 'arrival': 0, 'burst': 5, 'priority': 2},
    {'pid': 2, 'arrival': 1, 'burst': 3, 'priority': 1},
    {'pid': 3, 'arrival': 2, 'burst': 8, 'priority': 4},
    {'pid': 4, 'arrival': 3, 'burst': 6, 'priority': 3},
]
```

```
import copy

print("\n--- FCFS ---")

for p in fcfs(copy.deepcopy(processes)):

    print(p)


print("\n--- SJF ---")

for p in sjf(copy.deepcopy(processes)):

    print(p)


print("\n--- Round Robin (Q=2) ---")

for p in round_robin(copy.deepcopy(processes), quantum=2):

    print(p)


print("\n--- Priority Scheduling ---")

for p in priority_scheduling(copy.deepcopy(processes)):

    print(p)
```

Output: