KR MANGALAM UNIVERSITY

SOHNA ROAD, GURUGRAM

OPERATING SYSTEM LAB FILE

COURSE CODE-ENCS351

Submitted by

Name: Aldrin Debbarma

No.: 2401011482

Program: BTech CSE

Section: 6/F

:Session:2025/26

Date: 02/12/2025

Submitted To

Dr Satinder Pal Singh Roll

INDEX

S/no	Title	Remarks
1	Lab assignment 1	
2	Lab assignemnt 2	
3	Lab assignment 3	
4	Lab assignment 4	
5		
6		
7		
8		

Lab Assignment 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: import os

```
import sys


def main():
    # Read N from command-line or set a default
    if len(sys.argv) > 1:
        try:
            N = int(sys.argv[1])
        except ValueError:
            print("Usage: python prog.py <N>")
            sys.exit(1)
    else:
        N = 3  # default number of children

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

        if pid == 0:
            # Child process
            print(f"Child #{i+1}: PID = {os.getpid()}, PPID = {os.getppid()}")
            print(f"Child #{i+1}: Hello from child process!")
            os._exit(0)  # ensure child does not continue loop

    # Parent waits for all children
    for _ in range(N):
        finished_pid, status = os.wait()
```

```
        print(f"Parent: Child with PID {finished_pid} exited with status {status}")

    print(f"Parent: PID = {os.getpid()}, all children have finished.")

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

```
$ python prog.py 3
Child #1: PID = 12345, PPID = 12340
Child #1: Hello from child process!
Child #2: PID = 12346, PPID = 12340
Child #2: Hello from child process!
Child #3: PID = 12347, PPID = 12340
Child #3: Hello from child process!
Parent: Child with PID 12345 exited with status 0
Parent: Child with PID 12347 exited with status 0
Parent: Child with PID 12346 exited with status 0
Parent: PID = 12340, all children have finished.
```

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:

```
import os

import sys

import time


def run_commands_in_children():
    """
    Creates child processes to execute specific Linux commands using os.execvp().
    """
    # A list of commands to execute. Each command is a list [program, argument1, argument2, ...]
    # This format is required for execvp.
    commands = [
        ["date"],                 # Prints current date/time
        ["ls", "-l"],             # Lists files in long format
        ["echo", "Hello form PID!"],  # Echoes a string
        ["whoami"],               # Prints current user
        ["ps"]                    # Shows process status
```

```
    ]

    N = len(commands)

    print(f"Parent Process (PID: {os.getpid()}) started.")
    print(f"Planning to spawn {N} children to run commands.\n")

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

            if pid == 0:
                # ----------------------------------------------------------
                # CHILD PROCESS
                # ----------------------------------------------------------
                my_pid = os.getpid()
                parent_pid = os.getppid()
                command_to_run = commands[i]

                print(f"--> [Child {i+1}] PID: {my_pid} (Parent: {parent_pid}) executing: {'
'.join(command_to_run)}")

                # Flush stdout to ensure the print above appears before the command output
                sys.stdout.flush()

                try:
                    # os.execvp replaces the current process image with the new command.
                    # It takes two arguments:
```

```
            # 1. The name of the executable (e.g., "ls")
            # 2. A list of arguments including the command itself (e.g., ["ls", "-l"])
            os.execvp(command_to_run[0], command_to_run)

            # If execvp is successful, this code is NEVER reached because
            # the process memory is replaced by the new program.

        except FileNotFoundError:
            print(f"[Child {i+1}] Error: Command '{command_to_run[0]}' not found.")
            sys.exit(1)
        except OSError as e:
            print(f"[Child {i+1}] Error executing command: {e}")
            sys.exit(1)

    # ----------------------------------------------------------
    # PARENT PROCESS (Loop continues)
    # ----------------------------------------------------------

  except OSError as e:
    print(f"Parent Error: Fork failed. {e}")
    sys.exit(1)

# Parent waits for all children
print(f"\nParent: Waiting for {N} children to complete...\n")

while N > 0:
  try:
```

```
            # wait() blocks until ANY child finishes
            finished_pid, status = os.wait()

            # Check if exited normally
            if os.WIFEXITED(status):
                exit_code = os.WEXITSTATUS(status)
                print(f"Parent: Child {finished_pid} finished cleanly (Exit Code: {exit_code})")
            else:
                print(f"Parent: Child {finished_pid} terminated abnormally.")

            N -= 1
        except OSError:
            # No more children to wait for
            break

    print(f"\nParent Process (PID: {os.getpid()}) exiting.")

if __name__ == "__main__":
    if not hasattr(os, 'fork'):
        print("Error: This script requires a Unix-like OS (Linux/macOS). Windows is not supported.")
        sys.exit(1)

    run_commands_in_children()
```

Output:

```
Parent Process (PID: 18139) started.
Planning to spawn 5 children to run commands.

--> [Child 1] PID: 18140 (Parent: 18139) executing: date
Parent Process (PID: 18139) started.
Planning to spawn 5 children to run commands.

--> [Child 3] PID: 18142 (Parent: 18139) executing: echo Hello form PID!
Parent Process (PID: 18139) started.
Planning to spawn 5 children to run commands.

--> [Child 4] PID: 18143 (Parent: 18139) executing: whoami
Parent Process (PID: 18139) started.
Planning to spawn 5 children to run commands.

--> [Child 2] PID: 18141 (Parent: 18139) executing: ls -l
Parent Process (PID: 18139) started.
Planning to spawn 5 children to run commands.

--> [Child 5] PID: 18144 (Parent: 18139) executing: ps
Hello form PID!
Tue Nov 25 05:04:50 PM UTC 2025
coderunner
total 4
```

```
-rw-r--r-- 1 root root 3713 Nov 25 17:04 HelloWorld.py
  PID TTY          TIME CMD
10166 ?        00:00:00 HelloWorld
18138 ?        00:00:00 timeout
18139 ?        00:00:00 python3
18144 ?        00:00:00 ps
Parent Process (PID: 18139) started.
Planning to spawn 5 children to run commands.


Parent: Waiting for 5 children to complete...

Parent: Child 18142 finished cleanly (Exit Code: 0)
Parent: Child 18140 finished cleanly (Exit Code: 0)
Parent: Child 18143 finished cleanly (Exit Code: 0)
Parent: Child 18141 finished cleanly (Exit Code: 0)
Parent: Child 18144 finished cleanly (Exit Code: 0)

Parent Process (PID: 18139) exiting.
```

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:

```
import os
import sys
import time

def create_zombie():
    pid = os.fork()

    if pid == 0:
        # ---------------- CHILD ----------------
        print(f"[Child] PID: {os.getpid()}")
        print("[Child] I am exiting immediately. I should become a zombie because my parent isn't waiting yet.")
        os._exit(0)

    else:
        # ---------------- PARENT ----------------
        print(f"[Parent] PID: {os.getpid()}")
        print(f"[Parent] Created child {pid}.")
        print("[Parent] I am going to sleep for 4 seconds (reduced from 20s to avoid timeout) WITHOUT calling wait().")
        print(f"[Parent] QUICKly run this command in another terminal to see the zombie:\n")
```

```
        print(f"    ps -p {pid} -o pid,ppid,stat,cmd")
        print("    # OR just look for defunct processes:")
        print("    ps -el | grep defunct")

        # Sleep long enough for you to check the process table,
        # but short enough to avoid the 7s execution timeout.
        time.sleep(4)

        print("\n[Parent] Waking up. Now I will wait() for the child, cleaning up the zombie entry.")
        os.wait()
        print("[Parent] Child reaped. Zombie entry should be gone.")

if __name__ == "__main__":
    if not hasattr(os, 'fork'):
        sys.exit("This script requires a Unix-like OS.")
    create_zombie()
```

```
Output:

[Parent] PID: 12463
[Parent] Created child 12464.
[Parent] I am going to sleep for 4 seconds (reduced from 20s to avoid timeout) WITHOUT calling wait().
[Parent] QUICKly run this command in another terminal to see the zombie:

    ps -p 12464 -o pid,ppid,stat,cmd
    # OR just look for defunct processes:
    ps -el | grep defunct

[Parent] Waking up. Now I will wait() for the child, cleaning up the zombie entry.
[Parent] Child reaped. Zombie entry should be gone.
```

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:

```
import os

import sys


def inspect_process(pid):
    """
    Reads and prints process details from the /proc filesystem for a given PID.
    """
    proc_path = f"/proc/{pid}"


    if not os.path.exists(proc_path):
        print(f"Error: Process with PID {pid} does not exist (or /proc is not accessible).")
        return
```

```
print(f"\n--- Inspection Results for PID: {pid} ---\n")


try:
    # 1. Read Process Name, State, and Memory from /proc/[pid]/status
    status_path = os.path.join(proc_path, "status")
    print(f"Reading {status_path}...")


    needed_fields = {"Name:", "State:", "VmRSS:", "VmSize:"}


    with open(status_path, 'r') as f:
        for line in f:
            header = line.split()[0]
            if header in needed_fields:
                # Print the line stripping leading/trailing whitespace
                print(f"  {line.strip()}")


    # 2. Read Executable Path from /proc/[pid]/exe
    # The 'exe' file in /proc is a symbolic link to the actual binary
    exe_path = os.path.join(proc_path, "exe")
    try:
        real_path = os.readlink(exe_path)
        print(f"\nExecutable Path ({exe_path}):")
        print(f"  {real_path}")
    except PermissionError:
        print(f"\nExecutable Path ({exe_path}):")
        print("  [Permission Denied]")
```

```
except OSError as e:
    print(f"\nExecutable Path ({exe_path}):")
    print(f"  [Error reading link: {e}]")

# 3. List Open File Descriptors from /proc/[pid]/fd
fd_dir = os.path.join(proc_path, "fd")
print(f"\nOpen File Descriptors ({fd_dir}):")

try:
    fds = os.listdir(fd_dir)
    if not fds:
        print("  No open file descriptors found.")
    else:
        # Limit output if there are too many
        count = 0
        for fd in sorted(fds, key=lambda x: int(x)):
            if count >= 10:
                print("  ... (more FDs hidden)")
                break

            # Resolve where the FD points to
            full_fd_path = os.path.join(fd_dir, fd)
            try:
                target = os.readlink(full_fd_path)
                print(f"  FD {fd} -> {target}")
            except OSError:
                print(f"  FD {fd} -> ???")
```

```
                count += 1

        except PermissionError:
            print("  [Permission Denied: Cannot list file descriptors]")

    except PermissionError:
        print("Error: Permission denied. You may need to run this script with 'sudo' to inspect this process.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

    print("\n--- End of Inspection ---")

if __name__ == "__main__":
    # Check OS
    if not os.path.exists("/proc"):
        print("Error: This script requires a Linux environment with the /proc filesystem.")
        sys.exit(1)

    target_pid = None

    # Try to get PID from command line arguments
    if len(sys.argv) > 1:
        try:
            target_pid = int(sys.argv[1])
        except ValueError:
            print("Error: Invalid PID provided.")
```

```
# If no argument is provided, default to current PID immediately
# This avoids blocking on input() which causes timeouts in some environments
if target_pid is None:
    print("No PID argument provided. Defaulting to current process...")
    target_pid = os.getpid()

inspect_process(target_pid)
```

```
Output:

No PID argument provided. Defaulting to current process...

--- Inspection Results for PID: 21544 ---

Reading /proc/21544/status...
  Name: python3
  State:        R (running)
  VmSize:          17940 kB
  VmRSS:            9736 kB

Executable Path (/proc/21544/exe):
  /usr/bin/python3.12

Open File Descriptors (/proc/21544/fd):
  FD 0 -> socket:[890313]
  FD 1 -> socket:[890315]
  FD 2 -> socket:[890317]
  FD 3 -> ???

--- End of Inspection ---
```

Task 5: Process Prioritization

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

```
import os

import sys

import time

import math


def heavy_computation(label):
    """
    Performs a CPU-intensive task to keep the processor busy.
    """
```

```
    start_time = time.time()
    count = 0
    # Perform a heavy calculation loop (approx 50 million iterations)
    # Adjusted to ensure it takes enough time to notice scheduler differences
    target = 30_000_000

    for i in range(target):
        count += math.sqrt(i * i + 1.5)

    end_time = time.time()
    duration = end_time - start_time
    print(f"[{label}] Finished! Duration: {duration:.4f} seconds")

def test_priorities():
    if not hasattr(os, 'nice'):
        print("Error: os.nice() is not available on this OS.")
        return

    # 1. Force Single-Core Execution
    # Modern CPUs are too fast and have too many cores to easily see scheduler
    # prioritization impacts unless we force contention on a single core.
    if hasattr(os, 'sched_setaffinity'):
        try:
            # Bind to CPU 0
            os.sched_setaffinity(0, {0})
            print("Parent: Bound to CPU 0 to force scheduler contention.\n")
        except OSError:
```

```
        print("Parent: Could not set CPU affinity (might need sudo/root). Results may vary on multi-core.")
else:
    print("Parent: os.sched_setaffinity not available. Results may vary on multi-core systems.")

# 2. Define Children with different nice adjustments
# Note: Non-root users can only INCREASE nice value (lower priority).
# We will use 0 (base), 10 (slower), and 19 (slowest).
configs = [
    ("High Priority (Nice 0) ", 0),
    ("Med Priority  (Nice 10)", 10),
    ("Low Priority  (Nice 19)", 19)
]

children_pids = []

print(f"{'Process':<25} | {'PID':<6} | {'Nice Value':<10} | {'Status'}")
print("-" * 65)

for label, nice_increment in configs:
    try:
        pid = os.fork()

        if pid == 0:
            # -------- CHILD --------
            # Adjust niceness
            try:
                current_nice = os.nice(nice_increment)
```

```
            except OSError as e:
                print(f"Error setting nice: {e}")
                os._exit(1)

            my_pid = os.getpid()
            print(f"{label:<25} | {my_pid:<6} | {current_nice:<10} | Started computation...")

            # Flush to ensure output appears before computation blocks
            sys.stdout.flush()

            # Run heavy task
            heavy_computation(label.strip())

            os._exit(0)

        else:
            # -------- PARENT --------
            children_pids.append(pid)

    except OSError as e:
        print(f"Fork failed: {e}")

# 3. Wait for all children
for _ in children_pids:
    os.wait()

print("\nParent: All children finished.")
```

```
if __name__ == "__main__":
    if sys.platform == 'win32':
        print("This script requires a Unix-like OS (Linux/macOS) for fork() and nice().")
    else:
        test_priorities()
```

Output:

```
Parent: Bound to CPU 0 to force scheduler contention.

Process                  | PID    | Nice Value | Status
-------------------------------------------------------------
Low Priority  (Nice 19)   | 14552  | 19         | Started computation...
Parent: Bound to CPU 0 to force scheduler contention.

Process                  | PID    | Nice Value | Status
-------------------------------------------------------------
Med Priority  (Nice 10)   | 14551  | 10         | Started computation...
Parent: Bound to CPU 0 to force scheduler contention.

Process                  | PID    | Nice Value | Status
-------------------------------------------------------------
High Priority (Nice 0)    | 14550  | 0          | Started computation...

Error: Command failed: timeout 7 python3 HelloWorld.py
```

Lab Assignment 2

Sub-Task 1: Initialize the logging configuration

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

```
import logging

# Setup logger

logging.basicConfig(

    filename='process_log.txt',

    level=logging.INFO,

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

)
```

Code:

```
import logging


# Setup logger configuration

# We use '/tmp/process_log.txt' to ensure we have write permissions

logging.basicConfig(

    filename='/tmp/process_log.txt',

    level=logging.INFO,

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

)


# Test the logger

logging.info("Logging configuration initialized successfully.")

print("Log file created successfully at /tmp/process_log.txt")
```

```
Output:

Log file created successfully at /tmp/process_log.txt
```

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

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

```
import time

# Dummy function to simulate a task

def system_process(task_name):

    logging.info(f"{task_name} started")

    time.sleep(2)  # Simulate task delay

    logging.info(f"{task_name} ended")
```

Code:

```
import logging

import time

import sys


# --- Sub-Task 1: Logging Configuration ---

# We use '/tmp/process_log.txt' to ensure write permissions in restricted environments.

# If you prefer console output, we can change filename to a StreamHandler.

logging.basicConfig(

    filename='/tmp/process_log.txt',

    level=logging.INFO,

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

)


# Optional: Add a handler to print logs to the console as well so you can see them immediately
```

```
console_handler = logging.StreamHandler(sys.stdout)
console_handler.setFormatter(logging.Formatter('%(asctime)s - %(processName)s - %(message)s'))
logging.getLogger().addHandler(console_handler)

# --- Sub-Task 2: Process Simulation Function ---
def system_process(task_name):
    """
    Simulates a system process by logging start/end times and sleeping.
    """
    logging.info(f"{task_name} started")
    time.sleep(2)  # Simulate task delay
    logging.info(f"{task_name} ended")

# --- Execution Block ---
if __name__ == "__main__":
    print("Starting simulation...")
    system_process("Task-A")
    print("Simulation complete. Check /tmp/process_log.txt for details.")
```

```
Output:

Starting simulation...
2025-11-25 17:41:41,177 - MainProcess - Task-A started
2025-11-25 17:41:43,178 - MainProcess - Task-A ended
Simulation complete. Check /tmp/process_log.txt for details.
```

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

Objective: Use the multiprocessing module to initiate parallel tasks.

```
import multiprocessing

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()
```

Code:

```
import logging
import time
import sys
import multiprocessing

# --- Sub-Task 1: Logging Configuration ---
# We use '/tmp/process_log.txt' to ensure write permissions in restricted environments.
```

```
# If you prefer console output, we can change filename to a StreamHandler.
logging.basicConfig(
    filename='/tmp/process_log.txt',
    level=logging.INFO,
    format='%(asctime)s - %(processName)s - %(message)s'
)

# Optional: Add a handler to print logs to the console as well so you can see them immediately
console_handler = logging.StreamHandler(sys.stdout)
console_handler.setFormatter(logging.Formatter('%(asctime)s - %(processName)s - %(message)s'))
logging.getLogger().addHandler(console_handler)

# --- Sub-Task 2: Process Simulation Function ---
def system_process(task_name):
    """
    Simulates a system process by logging start/end times and sleeping.
    """
    logging.info(f"{task_name} started")
    time.sleep(2)  # Simulate task delay
    logging.info(f"{task_name} ended")

# --- Sub-Task 3: Execution Block with Multiprocessing ---
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 Finished. Check /tmp/process_log.txt for details.")
```

```
Output:

System Starting...
2025-11-25 17:44:37,994 - Process-1 - Process-1 started
2025-11-25 17:44:37,997 - Process-2 - Process-2 started
2025-11-25 17:44:39,995 - Process-1 - Process-1 ended
2025-11-25 17:44:39,997 - Process-2 - Process-2 ended
System Finished. Check /tmp/process_log.txt for details.
```

Sub-Task 4: Ensure proper termination and verify logs

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

```
    # Wait for processes to complete
    p1.join()
    p2.join()
    print("System Shutdown.")
```

Code:

```
import logging
import time
import sys
import multiprocessing

# --- Sub-Task 1: Logging Configuration ---
# We use '/tmp/process_log.txt' to ensure write permissions in restricted environments.
# If you prefer console output, we can change filename to a StreamHandler.
logging.basicConfig(
    filename='/tmp/process_log.txt',
```

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

# Optional: Add a handler to print logs to the console as well so you can see them immediately
console_handler = logging.StreamHandler(sys.stdout)
console_handler.setFormatter(logging.Formatter('%(asctime)s - %(processName)s - %(message)s'))
logging.getLogger().addHandler(console_handler)

# --- Sub-Task 2: Process Simulation Function ---
def system_process(task_name):
    """
    Simulates a system process by logging start/end times and sleeping.
    """
    logging.info(f"{task_name} started")
    time.sleep(2)  # Simulate task delay
    logging.info(f"{task_name} ended")

# --- Sub-Task 3: Execution Block with Multiprocessing ---
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()


# --- Sub-Task 4: Ensure proper termination and verify logs ---

# Wait for processes to complete

p1.join()

p2.join()

print("System Shutdown.")
```

```
Output:

System Starting...
2025-11-25 17:46:24,409 - Process-1 - Process-1 started
2025-11-25 17:46:24,410 - Process-2 - Process-2 started
2025-11-25 17:46:26,409 - Process-1 - Process-1 ended
2025-11-25 17:46:26,410 - Process-2 - Process-2 ended
System Shutdown.
```

Program Code with Explanation:

The following Python script simulates a simple system startup. It creates a few dummy processes which perform simple tasks. Each process logs its start and end, and the logger records the process flow, simulating system behavior.

```
# Import required libraries

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.")
```

Expected Output:

• The terminal displays 'System Starting...' and 'System Shutdown.'

• A log file 'process_log.txt' is generated which records the start and end of each process.

Output:

```
✅ Using temp directory for log: /tmp/process_log.txt

============================================================
System Starting...
============================================================
[*] Starting Process-1...
[*] Starting Process-2...
[*] Waiting for processes to complete...
============================================================
System Shutdown.
============================================================

✅ Log file created successfully!
📂 Location: /tmp/process_log.txt

📄 Log Contents:
------------------------------------------------------------
2025-11-25 16:07:25,992 - Process-1 - Process-1 started
2025-11-25 16:07:25,993 - Process-2 - Process-2 started
2025-11-25 16:07:27,992 - Process-1 - Process-1 ended
2025-11-25 16:07:27,994 - Process-2 - Process-2 ended

------------------------------------------------------------
```

Lab Assignment 3

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.

Code:

```
import sys


def print_gantt_chart(execution_log):
    """
    Prints a text-based Gantt chart from an execution log.
    execution_log: list of tuples (Process ID, Duration)
    """
    print("\n" + "="*50)
    print("GANTT CHART")
    print("="*50)

    # Print the bar
    print(" ", end="")
    for pid, duration in execution_log:
        # Scale the bar slightly for visual representation if needed,
        # but here we just use fixed width or proportional to duration roughly
        print(f"|  {pid}  ", end="")
    print("|")

    # Print the timeline
    time = 0
    print(f" {time:<6}", end="")
```

```
    for pid, duration in execution_log:
        time += duration
        print(f"{time:<7}", end="")
    print("\n" + "="*50 + "\n")

def priority_scheduling():
    print("\n--- Priority Scheduling (Non-Preemptive) ---")
    try:
        n = int(input("Enter number of processes: "))
        processes = []

        # Input
        for i in range(n):
            print(f"\nProcess {i+1}:")
            bt = int(input(f"  Burst Time: "))
            pr = int(input(f"  Priority (lower # = higher priority): "))
            # Storing as [PID, BT, Priority, Original_BT]
            processes.append({'pid': f"P{i+1}", 'bt': bt, 'prio': pr})

        # Logic: Sort by Priority (Ascending)
        # If priorities are equal, usually FCFS is used (stable sort preserves order)
        processes.sort(key=lambda x: x['prio'])

        total_wt = 0
        total_tat = 0
        current_time = 0
        execution_log = [] # For Gantt Chart
```

```
    print("\nPID\tPriority\tBurst Time\tWaiting Time\tTurnaround Time")

    for p in processes:
        # Waiting Time = Current Time - Arrival Time (Assuming Arrival = 0)
        wt = current_time

        # Execute Process
        duration = p['bt']
        current_time += duration

        # Turnaround Time = Waiting Time + Burst Time
        tat = wt + duration

        total_wt += wt
        total_tat += tat

        # Add to Gantt Chart log
        execution_log.append((p['pid'], duration))

        print(f"{p['pid']}\t{p['prio']}\t\t{p['bt']}\t\t{wt}\t\t{tat}")

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

except ValueError:
```

```
        print("Invalid input! Please enter integers.")

def round_robin_scheduling():
    print("\n--- Round Robin Scheduling ---")
    try:
        n = int(input("Enter number of processes: "))
        quantum = int(input("Enter Time Quantum: "))

        processes = []
        for i in range(n):
            bt = int(input(f"Enter Burst Time for P{i+1}: "))
            # PID, Burst Time, Remaining Time, Waiting Time, Turnaround Time
            processes.append({
                'pid': f"P{i+1}",
                'bt': bt,
                'rem_bt': bt,
                'wt': 0,
                'tat': 0
            })

        current_time = 0
        completed = 0
        execution_log = [] # For Gantt Chart

        # Queue logic simulates the ready queue
        # Since arrival time is 0 for all, we can just loop through the list repeatedly
        # until all are done.
```

```
while completed < n:
    all_done = True

    for p in processes:
        if p['rem_bt'] > 0:
            all_done = False

            if p['rem_bt'] > quantum:
                # Process runs for full quantum
                current_time += quantum
                p['rem_bt'] -= quantum
                execution_log.append((p['pid'], quantum))
            else:
                # Process finishes
                time_spent = p['rem_bt']
                current_time += time_spent
                p['rem_bt'] = 0
                execution_log.append((p['pid'], time_spent))

                # Calculate Completion Metrics
                # TAT = Completion Time - Arrival Time (0)
                p['tat'] = current_time
                # WT = TAT - Original Burst Time
                p['wt'] = p['tat'] - p['bt']

                completed += 1
```

```
            if all_done:
                break

        print("\nPID\tBurst Time\tWaiting Time\tTurnaround Time")
        total_wt = 0
        total_tat = 0

        for p in processes:
            print(f"{p['pid']}\t{p['bt']}\t\t{p['wt']}\t\t{p['tat']}")
            total_wt += p['wt']
            total_tat += p['tat']

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

    except ValueError:
        print("Invalid input! Please enter integers.")

def main():
    try:
        while True:
            print("\n=== CPU Scheduling Simulator ===")
            print("1. Priority Scheduling")
            print("2. Round Robin Scheduling")
            print("3. Exit")
```

```
            choice = input("Enter your choice (1-3): ")

            if choice == '1':
                priority_scheduling()
            elif choice == '2':
                round_robin_scheduling()
            elif choice == '3':
                print("Exiting...")
                sys.exit()
            else:
                print("Invalid choice. Please try again.")
        except (EOFError, KeyboardInterrupt):
            print("\n\nInput stream closed or interrupted. Exiting simulation.")
            sys.exit()

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

Output:

```
=== CPU Scheduling Simulator ===
1. Priority Scheduling
2. Round Robin Scheduling
3. Exit
Enter your choice (1-3):
--- Round Robin Scheduling ---
Enter number of processes: Enter Time Quantum: Enter Burst Time for P1:
PID     Burst Time      Waiting Time    Turnaround Time
P1      5               0               5

================================================
GANTT CHART
================================================
 |  P1  |  P1  |  P1  |
 0     2      4      5
================================================

Average Waiting Time: 0.00
Average Turnaround Time: 5.00
```

Task 2: Sequential File Allocation

Write a Python program to simulate sequential file allocation strategy.

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.")

Code:

```
import sys

def print_gantt_chart(execution_log):
    """
    Prints a text-based Gantt chart from an execution log.
    execution_log: list of tuples (Process ID, Duration)
    """
    print("\n" + "="*50)
    print("GANTT CHART")
    print("="*50)

    # Print the bar
    print(" ", end="")
    for pid, duration in execution_log:
        # Scale the bar slightly for visual representation if needed,
        # but here we just use fixed width or proportional to duration roughly
        print(f"|  {pid}  ", end="")
    print("|")

    # Print the timeline
    time = 0
```

```
    print(f" {time:<6}", end="")
    for pid, duration in execution_log:
        time += duration
        print(f"{time:<7}", end="")
    print("\n" + "="*50 + "\n")

def priority_scheduling():
    print("\n--- Priority Scheduling (Non-Preemptive) ---")
    try:
        n = int(input("Enter number of processes: "))
        processes = []

        # Input
        for i in range(n):
            print(f"\nProcess {i+1}:")
            bt = int(input(f"  Burst Time: "))
            pr = int(input(f"  Priority (lower # = higher priority): "))
            # Storing as [PID, BT, Priority, Original_BT]
            processes.append({'pid': f"P{i+1}", 'bt': bt, 'prio': pr})

        # Logic: Sort by Priority (Ascending)
        # If priorities are equal, usually FCFS is used (stable sort preserves order)
        processes.sort(key=lambda x: x['prio'])

        total_wt = 0
        total_tat = 0
        current_time = 0
```

```
execution_log = [] # For Gantt Chart

print("\nPID\tPriority\tBurst Time\tWaiting Time\tTurnaround Time")

for p in processes:
    # Waiting Time = Current Time - Arrival Time (Assuming Arrival = 0)
    wt = current_time

    # Execute Process
    duration = p['bt']
    current_time += duration

    # Turnaround Time = Waiting Time + Burst Time
    tat = wt + duration

    total_wt += wt
    total_tat += tat

    # Add to Gantt Chart log
    execution_log.append((p['pid'], duration))

    print(f"{p['pid']}\t{p['prio']}\t\t{p['bt']}\t\t{wt}\t\t{tat}")

print_gantt_chart(execution_log)
print(f"Average Waiting Time: {total_wt / n:.2f}")
print(f"Average Turnaround Time: {total_tat / n:.2f}")
```

```
    except ValueError:
        print("Invalid input! Please enter integers.")

def round_robin_scheduling():
    print("\n--- Round Robin Scheduling ---")
    try:
        n = int(input("Enter number of processes: "))
        quantum = int(input("Enter Time Quantum: "))

        processes = []
        for i in range(n):
            bt = int(input(f"Enter Burst Time for P{i+1}: "))
            # PID, Burst Time, Remaining Time, Waiting Time, Turnaround Time
            processes.append({
                'pid': f"P{i+1}",
                'bt': bt,
                'rem_bt': bt,
                'wt': 0,
                'tat': 0
            })

        current_time = 0
        completed = 0
        execution_log = [] # For Gantt Chart

        # Queue logic simulates the ready queue
        # Since arrival time is 0 for all, we can just loop through the list repeatedly
```

```
# until all are done.


while completed < n:
    all_done = True


    for p in processes:
        if p['rem_bt'] > 0:
            all_done = False


            if p['rem_bt'] > quantum:
                # Process runs for full quantum
                current_time += quantum
                p['rem_bt'] -= quantum
                execution_log.append((p['pid'], quantum))
            else:
                # Process finishes
                time_spent = p['rem_bt']
                current_time += time_spent
                p['rem_bt'] = 0
                execution_log.append((p['pid'], time_spent))


                # Calculate Completion Metrics
                # TAT = Completion Time - Arrival Time (0)
                p['tat'] = current_time
                # WT = TAT - Original Burst Time
                p['wt'] = p['tat'] - p['bt']
```

```
                completed += 1

            if all_done:
                break

        print("\nPID\tBurst Time\tWaiting Time\tTurnaround Time")
        total_wt = 0
        total_tat = 0

        for p in processes:
            print(f"{p['pid']}\t{p['bt']}\t\t{p['wt']}\t\t{p['tat']}")
            total_wt += p['wt']
            total_tat += p['tat']

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

    except ValueError:
        print("Invalid input! Please enter integers.")

def sequential_file_allocation():
    print("\n--- Sequential File Allocation Strategy ---")
    try:
        total_blocks = int(input("Enter total number of blocks: "))
        # 0 represents unallocated, 1 represents allocated
        block_status = [0] * total_blocks
```

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

    if start < 0:
        print(f"  -> Error: Starting block cannot be negative.")
        continue

    allocated = True

    # Check if allocation is possible
    for j in range(start, start + length):
        if j >= total_blocks:
            allocated = False
            break
        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 (Block overlap or out of bounds).")

    except ValueError:
        print("Invalid input! Please enter integers.")

def main():
    try:
        while True:
            print("\n=== OS Simulation Menu ===")
            print("1. CPU: Priority Scheduling")
            print("2. CPU: Round Robin Scheduling")
            print("3. Memory: Sequential File Allocation")
            print("4. Exit")
            choice = input("Enter your choice (1-4): ")

            if choice == '1':
                priority_scheduling()
            elif choice == '2':
                round_robin_scheduling()
            elif choice == '3':
                sequential_file_allocation()
            elif choice == '4':
                print("Exiting...")
                sys.exit()
            else:
                print("Invalid choice. Please try again.")
    except (EOFError, KeyboardInterrupt):
```

```
        print("\n\nInput stream closed or interrupted. Exiting simulation.")
        sys.exit()


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

```
Output:


=== OS Simulation Menu ===
1. CPU: Priority Scheduling
2. CPU: Round Robin Scheduling
3. Memory: Sequential File Allocation
4. Exit
Enter your choice (1-4):
--- Round Robin Scheduling ---
Enter number of processes: Enter Time Quantum: Enter Burst Time for P1:
PID     Burst Time      Waiting Time    Turnaround Time
P1      1               0               1

================================================
GANTT CHART
================================================
 |  P1  |
 0     1
================================================

Average Waiting Time: 0.00
Average Turnaround Time: 1.00
```

Task 3: Indexed File Allocation

Write a Python program to simulate indexed file allocation strategy.

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}")

Code:

```
import sys


def print_gantt_chart(execution_log):
    """
    Prints a text-based Gantt chart from an execution log.
    execution_log: list of tuples (Process ID, Duration)
    """
    print("\n" + "="*50)
    print("GANTT CHART")
    print("="*50)

    # Print the bar
    print(" ", end="")
    for pid, duration in execution_log:
        # Scale the bar slightly for visual representation if needed,
        # but here we just use fixed width or proportional to duration roughly
        print(f"|  {pid}  ", end="")
    print("|")

    # Print the timeline
    time = 0
    print(f" {time:<6}", end="")
    for pid, duration in execution_log:
        time += duration
        print(f"{time:<7}", end="")
    print("\n" + "="*50 + "\n")
```

```
def priority_scheduling():
    print("\n--- Priority Scheduling (Non-Preemptive) ---")
    try:
        n = int(input("Enter number of processes: "))
        processes = []

        # Input
        for i in range(n):
            print(f"\nProcess {i+1}:")
            bt = int(input(f"  Burst Time: "))
            pr = int(input(f"  Priority (lower # = higher priority): "))
            # Storing as [PID, BT, Priority, Original_BT]
            processes.append({'pid': f"P{i+1}", 'bt': bt, 'prio': pr})

        # Logic: Sort by Priority (Ascending)
        # If priorities are equal, usually FCFS is used (stable sort preserves order)
        processes.sort(key=lambda x: x['prio'])

        total_wt = 0
        total_tat = 0
        current_time = 0
        execution_log = [] # For Gantt Chart

        print("\nPID\tPriority\tBurst Time\tWaiting Time\tTurnaround Time")

        for p in processes:
```

```
            # Waiting Time = Current Time - Arrival Time (Assuming Arrival = 0)
            wt = current_time

            # Execute Process
            duration = p['bt']
            current_time += duration

            # Turnaround Time = Waiting Time + Burst Time
            tat = wt + duration

            total_wt += wt
            total_tat += tat

            # Add to Gantt Chart log
            execution_log.append((p['pid'], duration))

            print(f"{p['pid']}\t{p['prio']}\t\t{p['bt']}\t\t{wt}\t\t{tat}")

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

    except ValueError:
        print("Invalid input! Please enter integers.")

def round_robin_scheduling():
    print("\n--- Round Robin Scheduling ---")
```

```
try:
    n = int(input("Enter number of processes: "))
    quantum = int(input("Enter Time Quantum: "))

    processes = []
    for i in range(n):
        bt = int(input(f"Enter Burst Time for P{i+1}: "))
        # PID, Burst Time, Remaining Time, Waiting Time, Turnaround Time
        processes.append({
            'pid': f"P{i+1}",
            'bt': bt,
            'rem_bt': bt,
            'wt': 0,
            'tat': 0
        })

    current_time = 0
    completed = 0
    execution_log = [] # For Gantt Chart

    # Queue logic simulates the ready queue
    # Since arrival time is 0 for all, we can just loop through the list repeatedly
    # until all are done.

    while completed < n:
        all_done = True
```

```
for p in processes:
    if p['rem_bt'] > 0:
        all_done = False

        if p['rem_bt'] > quantum:
            # Process runs for full quantum
            current_time += quantum
            p['rem_bt'] -= quantum
            execution_log.append((p['pid'], quantum))
        else:
            # Process finishes
            time_spent = p['rem_bt']
            current_time += time_spent
            p['rem_bt'] = 0
            execution_log.append((p['pid'], time_spent))

            # Calculate Completion Metrics
            # TAT = Completion Time - Arrival Time (0)
            p['tat'] = current_time
            # WT = TAT - Original Burst Time
            p['wt'] = p['tat'] - p['bt']

            completed += 1

if all_done:
    break
```

```
        print("\nPID\tBurst Time\tWaiting Time\tTurnaround Time")
        total_wt = 0
        total_tat = 0

        for p in processes:
            print(f"{p['pid']}\t{p['bt']}\t\t{p['wt']}\t\t{p['tat']}")
            total_wt += p['wt']
            total_tat += p['tat']

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

    except ValueError:
        print("Invalid input! Please enter integers.")

def sequential_file_allocation():
    print("\n--- Sequential File Allocation Strategy ---")
    try:
        total_blocks = int(input("Enter total number of blocks: "))
        # 0 represents unallocated, 1 represents allocated
        block_status = [0] * total_blocks

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

```
        length = int(input(f"  Enter length: "))

        if start < 0:
            print(f"  -> Error: Starting block cannot be negative.")
            continue

        allocated = True

        # Check if allocation is possible
        for j in range(start, start + length):
            if j >= total_blocks:
                allocated = False
                break
            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 (Block overlap or out of bounds).")

except ValueError:
    print("Invalid input! Please enter integers.")
```

```
def indexed_file_allocation():
    print("\n--- Indexed File Allocation Strategy ---")
    try:
        total_blocks = int(input("Enter total number of blocks: "))
        # 0 represents unallocated, 1 represents allocated
        block_status = [0] * total_blocks

        n = int(input("Enter number of files: "))
        for i in range(n):
            print(f"\nFile {i+1}:")

            try:
                index_block = int(input(f"  Enter index block: "))

                # Check index block validity
                if index_block < 0 or index_block >= total_blocks:
                    print(f"  -> Error: Index block {index_block} is out of bounds.")
                    continue
                if block_status[index_block] == 1:
                    print(f"  -> Error: Index block {index_block} is already allocated.")
                    continue

                count = int(input(f"  Enter number of data blocks: "))
                blocks_input = input(f"  Enter {count} block numbers separated by space: ")
                data_blocks = list(map(int, blocks_input.split()))

                if len(data_blocks) != count:
```

```
    print(f"  -> Error: Expected {count} blocks, but got {len(data_blocks)}.")
    continue

# Check data blocks validity
possible = True
for blk in data_blocks:
    if blk < 0 or blk >= total_blocks:
        print(f"  -> Error: Block {blk} is out of bounds.")
        possible = False
        break
    if block_status[blk] == 1:
        print(f"  -> Error: Block {blk} is already allocated.")
        possible = False
        break
    if blk == index_block:
         print(f"  -> Error: Block {blk} cannot be same as index block.")
         possible = False
         break

if possible:
    # Allocate Index Block
    block_status[index_block] = 1
    # Allocate Data Blocks
    for blk in data_blocks:
        block_status[blk] = 1
    print(f"  -> File {i+1} allocated. Index: {index_block} -> Blocks: {data_blocks}")
else:
```

```
                print(f"  -> File {i+1} allocation failed.")

            except ValueError:
                print("  -> Invalid input for blocks. Please enter integers.")

    except ValueError:
        print("Invalid input! Please enter integers.")

def main():
    try:
        while True:
            print("\n=== OS Simulation Menu ===")
            print("1. CPU: Priority Scheduling")
            print("2. CPU: Round Robin Scheduling")
            print("3. Memory: Sequential File Allocation")
            print("4. Memory: Indexed File Allocation")
            print("5. Exit")
            choice = input("Enter your choice (1-5): ")

            if choice == '1':
                priority_scheduling()
            elif choice == '2':
                round_robin_scheduling()
            elif choice == '3':
                sequential_file_allocation()
            elif choice == '4':
                indexed_file_allocation()
```

```
            elif choice == '5':
                print("Exiting...")
                sys.exit()
            else:
                print("Invalid choice. Please try again.")
    except (EOFError, KeyboardInterrupt):
        print("\n\nInput stream closed or interrupted. Exiting simulation.")
        sys.exit()


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

```
Output:


=== OS Simulation Menu ===
1. CPU: Priority Scheduling
2. CPU: Round Robin Scheduling
3. Memory: Sequential File Allocation
4. Memory: Indexed File Allocation
5. Exit
Enter your choice (1-5):
--- Round Robin Scheduling ---
Enter number of processes: Enter Time Quantum: Enter Burst Time for P1:
PID     Burst Time      Waiting Time    Turnaround Time
P1      5               0               5

================================================
GANTT CHART
================================================
 |  P1  |  P1  |  P1  |
 0     2      4      5
================================================

Average Waiting Time: 0.00
Average Turnaround Time: 5.00
```

Task 4: Contiguous Memory Allocation

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

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")
```

Code:

```
import sys


def print_gantt_chart(execution_log):
    """
    Prints a text-based Gantt chart from an execution log.
    execution_log: list of tuples (Process ID, Duration)
    """
    print("\n" + "="*50)
    print("GANTT CHART")
```

```
    print("="*50)

    # Print the bar
    print(" ", end="")
    for pid, duration in execution_log:
        # Scale the bar slightly for visual representation if needed,
        # but here we just use fixed width or proportional to duration roughly
        print(f"|  {pid}  ", end="")
    print("|")

    # Print the timeline
    time = 0
    print(f" {time:<6}", end="")
    for pid, duration in execution_log:
        time += duration
        print(f"{time:<7}", end="")
    print("\n" + "="*50 + "\n")

def priority_scheduling():
    print("\n--- Priority Scheduling (Non-Preemptive) ---")
    try:
        n = int(input("Enter number of processes: "))
        processes = []

        # Input
        for i in range(n):
            print(f"\nProcess {i+1}:")
```

```
    bt = int(input(f"  Burst Time: "))
    pr = int(input(f"  Priority (lower # = higher priority): "))
    # Storing as [PID, BT, Priority, Original_BT]
    processes.append({'pid': f"P{i+1}", 'bt': bt, 'prio': pr})

# Logic: Sort by Priority (Ascending)
# If priorities are equal, usually FCFS is used (stable sort preserves order)
processes.sort(key=lambda x: x['prio'])

total_wt = 0
total_tat = 0
current_time = 0
execution_log = [] # For Gantt Chart

print("\nPID\tPriority\tBurst Time\tWaiting Time\tTurnaround Time")

for p in processes:
    # Waiting Time = Current Time - Arrival Time (Assuming Arrival = 0)
    wt = current_time

    # Execute Process
    duration = p['bt']
    current_time += duration

    # Turnaround Time = Waiting Time + Burst Time
    tat = wt + duration
```

```
            total_wt += wt
            total_tat += tat

            # Add to Gantt Chart log
            execution_log.append((p['pid'], duration))

            print(f"{p['pid']}\t{p['prio']}\t\t{p['bt']}\t\t{wt}\t\t{tat}")

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

    except ValueError:
        print("Invalid input! Please enter integers.")

def round_robin_scheduling():
    print("\n--- Round Robin Scheduling ---")
    try:
        n = int(input("Enter number of processes: "))
        quantum = int(input("Enter Time Quantum: "))

        processes = []
        for i in range(n):
            bt = int(input(f"Enter Burst Time for P{i+1}: "))
            # PID, Burst Time, Remaining Time, Waiting Time, Turnaround Time
            processes.append({
                'pid': f"P{i+1}",
```

```
        'bt': bt,
        'rem_bt': bt,
        'wt': 0,
        'tat': 0
    })

current_time = 0
completed = 0
execution_log = [] # For Gantt Chart

# Queue logic simulates the ready queue
# Since arrival time is 0 for all, we can just loop through the list repeatedly
# until all are done.

while completed < n:
    all_done = True

    for p in processes:
        if p['rem_bt'] > 0:
            all_done = False

            if p['rem_bt'] > quantum:
                # Process runs for full quantum
                current_time += quantum
                p['rem_bt'] -= quantum
                execution_log.append((p['pid'], quantum))
            else:
```

```
            # Process finishes
            time_spent = p['rem_bt']
            current_time += time_spent
            p['rem_bt'] = 0
            execution_log.append((p['pid'], time_spent))

            # Calculate Completion Metrics
            # TAT = Completion Time - Arrival Time (0)
            p['tat'] = current_time
            # WT = TAT - Original Burst Time
            p['wt'] = p['tat'] - p['bt']

            completed += 1

    if all_done:
        break

print("\nPID\tBurst Time\tWaiting Time\tTurnaround Time")
total_wt = 0
total_tat = 0

for p in processes:
    print(f"{p['pid']}\t{p['bt']}\t\t{p['wt']}\t\t{p['tat']}")
    total_wt += p['wt']
    total_tat += p['tat']

print_gantt_chart(execution_log)
```

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

    except ValueError:
        print("Invalid input! Please enter integers.")

def sequential_file_allocation():
    print("\n--- Sequential File Allocation Strategy ---")
    try:
        total_blocks = int(input("Enter total number of blocks: "))
        # 0 represents unallocated, 1 represents allocated
        block_status = [0] * total_blocks

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

            if start < 0:
                print(f"  -> Error: Starting block cannot be negative.")
                continue

            allocated = True

            # Check if allocation is possible
            for j in range(start, start + length):
```

```
            if j >= total_blocks:
                allocated = False
                break
            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 (Block overlap or out of bounds).")

    except ValueError:
        print("Invalid input! Please enter integers.")

def indexed_file_allocation():
    print("\n--- Indexed File Allocation Strategy ---")
    try:
        total_blocks = int(input("Enter total number of blocks: "))
        # 0 represents unallocated, 1 represents allocated
        block_status = [0] * total_blocks

        n = int(input("Enter number of files: "))
        for i in range(n):
            print(f"\nFile {i+1}:")
```

```
try:
    index_block = int(input(f"  Enter index block: "))

    # Check index block validity
    if index_block < 0 or index_block >= total_blocks:
        print(f"  -> Error: Index block {index_block} is out of bounds.")
        continue
    if block_status[index_block] == 1:
        print(f"  -> Error: Index block {index_block} is already allocated.")
        continue

    count = int(input(f"  Enter number of data blocks: "))
    blocks_input = input(f"  Enter {count} block numbers separated by space: ")
    data_blocks = list(map(int, blocks_input.split()))

    if len(data_blocks) != count:
        print(f"  -> Error: Expected {count} blocks, but got {len(data_blocks)}.")
        continue

    # Check data blocks validity
    possible = True
    for blk in data_blocks:
        if blk < 0 or blk >= total_blocks:
            print(f"  -> Error: Block {blk} is out of bounds.")
            possible = False
            break
```

```
            if block_status[blk] == 1:
                print(f"  -> Error: Block {blk} is already allocated.")
                possible = False
                break
            if blk == index_block:
                print(f"  -> Error: Block {blk} cannot be same as index block.")
                possible = False
                break

        if possible:
            # Allocate Index Block
            block_status[index_block] = 1
            # Allocate Data Blocks
            for blk in data_blocks:
                block_status[blk] = 1
            print(f"  -> File {i+1} allocated. Index: {index_block} -> Blocks: {data_blocks}")
        else:
            print(f"  -> File {i+1} allocation failed.")

    except ValueError:
        print("  -> Invalid input for blocks. Please enter integers.")

except ValueError:
    print("Invalid input! Please enter integers.")

def contiguous_memory_allocation():
    print("\n--- Contiguous Memory Allocation ---")
```

```
print("1. First Fit")
print("2. Best Fit")
print("3. Worst Fit")

try:
    choice = input("Select Strategy (1-3): ")
    if choice not in ['1', '2', '3']:
        print("Invalid choice.")
        return

    strategy_map = {'1': 'first', '2': 'best', '3': 'worst'}
    strategy = strategy_map[choice]

    print("Enter partition sizes separated by space (e.g., 100 500 200):")
    partitions = list(map(int, input().split()))

    print("Enter process sizes separated by space (e.g., 212 417 112):")
    processes = list(map(int, input().split()))

    allocation = [-1] * len(processes)

    for i, psize in enumerate(processes):
        idx = -1

        if strategy == "first":
            # Allocate the first hole that is big enough
            for j, part in enumerate(partitions):
```

```
            if part >= psize:
                idx = j
                break

    elif strategy == "best":
        # Allocate the smallest hole that is big enough
        best_fit_size = float("inf")
        for j, part in enumerate(partitions):
            if part >= psize and part < best_fit_size:
                best_fit_size = part
                idx = j

    elif strategy == "worst":
        # Allocate the largest hole
        worst_fit_size = -1
        for j, part in enumerate(partitions):
            if part >= psize and part > worst_fit_size:
                worst_fit_size = part
                idx = j

    if idx != -1:
        allocation[i] = idx
        partitions[idx] -= psize

print(f"\nAllocation Result ({strategy.capitalize()} Fit):")
for i, a in enumerate(allocation):
    if a != -1:
```

```
            print(f"Process {i+1} ({processes[i]}) -> Partition {a+1} (Remaining space: {partitions[a]})")
        else:
            print(f"Process {i+1} ({processes[i]}) -> Not Allocated")

    except ValueError:
        print("Invalid input! Please enter integers.")

def main():
    try:
        while True:
            print("\n=== OS Simulation Menu ===")
            print("1. CPU: Priority Scheduling")
            print("2. CPU: Round Robin Scheduling")
            print("3. Memory: Sequential File Allocation")
            print("4. Memory: Indexed File Allocation")
            print("5. Memory: Contiguous Allocation (First/Best/Worst)")
            print("6. Exit")
            choice = input("Enter your choice (1-6): ")

            if choice == '1':
                priority_scheduling()
            elif choice == '2':
                round_robin_scheduling()
            elif choice == '3':
                sequential_file_allocation()
            elif choice == '4':
                indexed_file_allocation()
```

```
            elif choice == '5':
                contiguous_memory_allocation()
            elif choice == '6':
                print("Exiting...")
                sys.exit()
            else:
                print("Invalid choice. Please try again.")
    except (EOFError, KeyboardInterrupt):
        print("\n\nInput stream closed or interrupted. Exiting simulation.")
        sys.exit()

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

```
Output:


=== OS Simulation Menu ===
1. CPU: Priority Scheduling
2. CPU: Round Robin Scheduling
3. Memory: Sequential File Allocation
4. Memory: Indexed File Allocation
5. Memory: Contiguous Allocation (First/Best/Worst)
6. Exit
Enter your choice (1-6):
--- Round Robin Scheduling ---
Enter number of processes: Enter Time Quantum: Enter Burst Time for P1:
PID     Burst Time      Waiting Time    Turnaround Time
P1      5               0               5

================================================
GANTT CHART
================================================
 |  P1  |  P1  |  P1  |
 0     2      4      5
================================================

Average Waiting Time: 0.00
Average Turnaround Time: 5.00
```

Task 5: MFT & MVT Memory Management

Implement MFT (fixed partitions) and MVT (variable partitions) strategies in 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()
```

Code:

```
import sys

def print_gantt_chart(execution_log):
    """
    Prints a text-based Gantt chart from an execution log.
    execution_log: list of tuples (Process ID, Duration)
    """
    print("\n" + "="*50)
    print("GANTT CHART")
    print("="*50)

    # Print the bar
    print(" ", end="")
    for pid, duration in execution_log:
        # Scale the bar slightly for visual representation if needed,
```

```
        # but here we just use fixed width or proportional to duration roughly
        print(f"|  {pid}  ", end="")
    print("|")

    # Print the timeline
    time = 0
    print(f" {time:<6}", end="")
    for pid, duration in execution_log:
        time += duration
        print(f"{time:<7}", end="")
    print("\n" + "="*50 + "\n")

def priority_scheduling():
    print("\n--- Priority Scheduling (Non-Preemptive) ---")
    try:
        n = int(input("Enter number of processes: "))
        processes = []

        # Input
        for i in range(n):
            print(f"\nProcess {i+1}:")
            bt = int(input(f"  Burst Time: "))
            pr = int(input(f"  Priority (lower # = higher priority): "))
            # Storing as [PID, BT, Priority, Original_BT]
            processes.append({'pid': f"P{i+1}", 'bt': bt, 'prio': pr})

        # Logic: Sort by Priority (Ascending)
```

```
# If priorities are equal, usually FCFS is used (stable sort preserves order)
processes.sort(key=lambda x: x['prio'])

total_wt = 0
total_tat = 0
current_time = 0
execution_log = [] # For Gantt Chart

print("\nPID\tPriority\tBurst Time\tWaiting Time\tTurnaround Time")

for p in processes:
    # Waiting Time = Current Time - Arrival Time (Assuming Arrival = 0)
    wt = current_time

    # Execute Process
    duration = p['bt']
    current_time += duration

    # Turnaround Time = Waiting Time + Burst Time
    tat = wt + duration

    total_wt += wt
    total_tat += tat

    # Add to Gantt Chart log
    execution_log.append((p['pid'], duration))
```

```
            print(f"{p['pid']}\t{p['prio']}\t\t{p['bt']}\t\t{wt}\t\t{tat}")

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

    except ValueError:
        print("Invalid input! Please enter integers.")

def round_robin_scheduling():
    print("\n--- Round Robin Scheduling ---")
    try:
        n = int(input("Enter number of processes: "))
        quantum = int(input("Enter Time Quantum: "))

        processes = []
        for i in range(n):
            bt = int(input(f"Enter Burst Time for P{i+1}: "))
            # PID, Burst Time, Remaining Time, Waiting Time, Turnaround Time
            processes.append({
                'pid': f"P{i+1}",
                'bt': bt,
                'rem_bt': bt,
                'wt': 0,
                'tat': 0
            })
```

```
current_time = 0
completed = 0
execution_log = [] # For Gantt Chart

# Queue logic simulates the ready queue
# Since arrival time is 0 for all, we can just loop through the list repeatedly
# until all are done.

while completed < n:
    all_done = True

    for p in processes:
        if p['rem_bt'] > 0:
            all_done = False

            if p['rem_bt'] > quantum:
                # Process runs for full quantum
                current_time += quantum
                p['rem_bt'] -= quantum
                execution_log.append((p['pid'], quantum))
            else:
                # Process finishes
                time_spent = p['rem_bt']
                current_time += time_spent
                p['rem_bt'] = 0
                execution_log.append((p['pid'], time_spent))
```

```
                # Calculate Completion Metrics
                # TAT = Completion Time - Arrival Time (0)
                p['tat'] = current_time
                # WT = TAT - Original Burst Time
                p['wt'] = p['tat'] - p['bt']

                completed += 1

        if all_done:
            break

    print("\nPID\tBurst Time\tWaiting Time\tTurnaround Time")
    total_wt = 0
    total_tat = 0

    for p in processes:
        print(f"{p['pid']}\t{p['bt']}\t\t{p['wt']}\t\t{p['tat']}")
        total_wt += p['wt']
        total_tat += p['tat']

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

except ValueError:
    print("Invalid input! Please enter integers.")
```

```
def sequential_file_allocation():
    print("\n--- Sequential File Allocation Strategy ---")
    try:
        total_blocks = int(input("Enter total number of blocks: "))
        # 0 represents unallocated, 1 represents allocated
        block_status = [0] * total_blocks

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

            if start < 0:
                print(f"  -> Error: Starting block cannot be negative.")
                continue

            allocated = True

            # Check if allocation is possible
            for j in range(start, start + length):
                if j >= total_blocks:
                    allocated = False
                    break
                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 (Block overlap or out of bounds).")

    except ValueError:
        print("Invalid input! Please enter integers.")

def indexed_file_allocation():
    print("\n--- Indexed File Allocation Strategy ---")
    try:
        total_blocks = int(input("Enter total number of blocks: "))
        # 0 represents unallocated, 1 represents allocated
        block_status = [0] * total_blocks

        n = int(input("Enter number of files: "))
        for i in range(n):
            print(f"\nFile {i+1}:")

            try:
                index_block = int(input(f"  Enter index block: "))

                # Check index block validity
                if index_block < 0 or index_block >= total_blocks:
```

```
    print(f"  -> Error: Index block {index_block} is out of bounds.")
    continue
if block_status[index_block] == 1:
    print(f"  -> Error: Index block {index_block} is already allocated.")
    continue

count = int(input(f"  Enter number of data blocks: "))
blocks_input = input(f"  Enter {count} block numbers separated by space: ")
data_blocks = list(map(int, blocks_input.split()))

if len(data_blocks) != count:
    print(f"  -> Error: Expected {count} blocks, but got {len(data_blocks)}.")
    continue

# Check data blocks validity
possible = True
for blk in data_blocks:
    if blk < 0 or blk >= total_blocks:
        print(f"  -> Error: Block {blk} is out of bounds.")
        possible = False
        break
    if block_status[blk] == 1:
        print(f"  -> Error: Block {blk} is already allocated.")
        possible = False
        break
    if blk == index_block:
         print(f"  -> Error: Block {blk} cannot be same as index block.")
```

```
                possible = False
                break

        if possible:
            # Allocate Index Block
            block_status[index_block] = 1
            # Allocate Data Blocks
            for blk in data_blocks:
                block_status[blk] = 1
            print(f"  -> File {i+1} allocated. Index: {index_block} -> Blocks: {data_blocks}")
        else:
            print(f"  -> File {i+1} allocation failed.")

    except ValueError:
        print("  -> Invalid input for blocks. Please enter integers.")

except ValueError:
    print("Invalid input! Please enter integers.")

def contiguous_memory_allocation():
    print("\n--- Contiguous Memory Allocation ---")
    print("1. First Fit")
    print("2. Best Fit")
    print("3. Worst Fit")

    try:
        choice = input("Select Strategy (1-3): ")
```

```
if choice not in ['1', '2', '3']:
    print("Invalid choice.")
    return

strategy_map = {'1': 'first', '2': 'best', '3': 'worst'}
strategy = strategy_map[choice]

print("Enter partition sizes separated by space (e.g., 100 500 200):")
partitions = list(map(int, input().split()))

print("Enter process sizes separated by space (e.g., 212 417 112):")
processes = list(map(int, input().split()))

allocation = [-1] * len(processes)

for i, psize in enumerate(processes):
    idx = -1

    if strategy == "first":
        # Allocate the first hole that is big enough
        for j, part in enumerate(partitions):
            if part >= psize:
                idx = j
                break

    elif strategy == "best":
        # Allocate the smallest hole that is big enough
```

```
            best_fit_size = float("inf")
            for j, part in enumerate(partitions):
                if part >= psize and part < best_fit_size:
                    best_fit_size = part
                    idx = j

        elif strategy == "worst":
            # Allocate the largest hole
            worst_fit_size = -1
            for j, part in enumerate(partitions):
                if part >= psize and part > worst_fit_size:
                    worst_fit_size = part
                    idx = j

        if idx != -1:
            allocation[i] = idx
            partitions[idx] -= psize

    print(f"\nAllocation Result ({strategy.capitalize()} Fit):")
    for i, a in enumerate(allocation):
        if a != -1:
            print(f'Process {i+1} ({processes[i]}) -> Partition {a+1} (Remaining space: {partitions[a]})")
        else:
            print(f'Process {i+1} ({processes[i]}) -> Not Allocated")

except ValueError:
    print("Invalid input! Please enter integers.")
```

```
def mft_mvt_simulation():
    print("\n--- MFT/MVT Memory Management ---")
    print("1. MFT (Multiprogramming with Fixed Tasks)")
    print("2. MVT (Multiprogramming with Variable Tasks)")

    try:
        choice = input("Select Type (1-2): ")

        if choice == '1': # MFT
            print("\n--- MFT Simulation ---")
            mem_size = int(input("Enter total memory size: "))
            part_size = int(input("Enter partition size: "))
            n = int(input("Enter number of processes: "))

            total_partitions = mem_size // part_size
            available_partitions = total_partitions
            print(f"Total Memory: {mem_size}, Partition Size: {part_size}")
            print(f"Total Partitions Available: {total_partitions}")

            for i in range(n):
                psize = int(input(f"Enter size of Process {i+1}: "))
                if available_partitions > 0:
                    if psize <= part_size:
                        internal_frag = part_size - psize
                        print(f"  -> Process {i+1} allocated. (Internal Fragmentation: {internal_frag})")
                        available_partitions -= 1
```

```
            else:
                print(f"  -> Process {i+1} too large for partition ({psize} > {part_size}).")
            else:
                print(f"  -> Process {i+1} cannot be allocated (Memory Full).")

        print(f"Remaining Partitions: {available_partitions}")
        print(f"Total Internal Fragmentation (Theoretical worst case unused space): {(total_partitions - available_partitions) * part_size}")

    elif choice == '2': # MVT
        print("\n--- MVT Simulation ---")
        mem_size = int(input("Enter total memory size: "))
        n = int(input("Enter number of processes: "))

        current_mem = mem_size

        for i in range(n):
            psize = int(input(f"Enter size of Process {i+1}: "))
            if psize <= current_mem:
                current_mem -= psize
                print(f"  -> Process {i+1} allocated. Remaining Memory: {current_mem}")
            else:
                print(f"  -> Process {i+1} cannot be allocated. (Required: {psize}, Available: {current_mem})")

        print(f"Final External Fragmentation: {current_mem}")

    else:
        print("Invalid choice.")
```

```
    except ValueError:
        print("Invalid input! Please enter integers.")


def main():
    try:
        while True:
            print("\n=== OS Simulation Menu ===")
            print("1. CPU: Priority Scheduling")
            print("2. CPU: Round Robin Scheduling")
            print("3. Memory: Sequential File Allocation")
            print("4. Memory: Indexed File Allocation")
            print("5. Memory: Contiguous Allocation (First/Best/Worst)")
            print("6. Memory: MFT & MVT Management")
            print("7. Exit")
            choice = input("Enter your choice (1-7): ")

            if choice == '1':
                priority_scheduling()
            elif choice == '2':
                round_robin_scheduling()
            elif choice == '3':
                sequential_file_allocation()
            elif choice == '4':
                indexed_file_allocation()
            elif choice == '5':
                contiguous_memory_allocation()
```

```
            elif choice == '6':
                mft_mvt_simulation()
            elif choice == '7':
                print("Exiting...")
                sys.exit()
            else:
                print("Invalid choice. Please try again.")
    except (EOFError, KeyboardInterrupt):
        print("\n\nInput stream closed or interrupted. Exiting simulation.")
        sys.exit()

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

Output:

```
=== OS Simulation Menu ===
1. CPU: Priority Scheduling
2. CPU: Round Robin Scheduling
3. Memory: Sequential File Allocation
4. Memory: Indexed File Allocation
5. Memory: Contiguous Allocation (First/Best/Worst)
6. Memory: MFT & MVT Management
7. Exit
Enter your choice (1-7):
--- Round Robin Scheduling ---
Enter number of processes: Enter Time Quantum: Enter Burst Time for P1:
PID     Burst Time      Waiting Time    Turnaround Time
P1      5               0               5

================================================
GANTT CHART
================================================
 |  P1  |  P1  |  P1  |
 0     2      4      5
================================================

Average Waiting Time: 0.00
Average Turnaround Time: 5.00
```

Lab Assignment 4

Task 1: Batch Processing Simulation (Python)

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

import subprocess

scripts = ['script1.py', 'script2.py', 'script3.py'] for script in scripts:

print(f"Executing {script}...") subprocess.call(['python3', script])

Code:

```
import subprocess
import sys
import os

def run_batch_jobs():
    # List of python scripts to execute sequentially
    # In a real scenario, these might be data processing or ETL scripts
    scripts = ['task_a.py', 'task_b.py']

    print(f"--- Starting Batch Processing of {len(scripts)} jobs ---")

    for script in scripts:
        # Check if file exists before trying to run it
        if not os.path.exists(script):
            print(f"[Error] File not found: {script}")
            continue
```

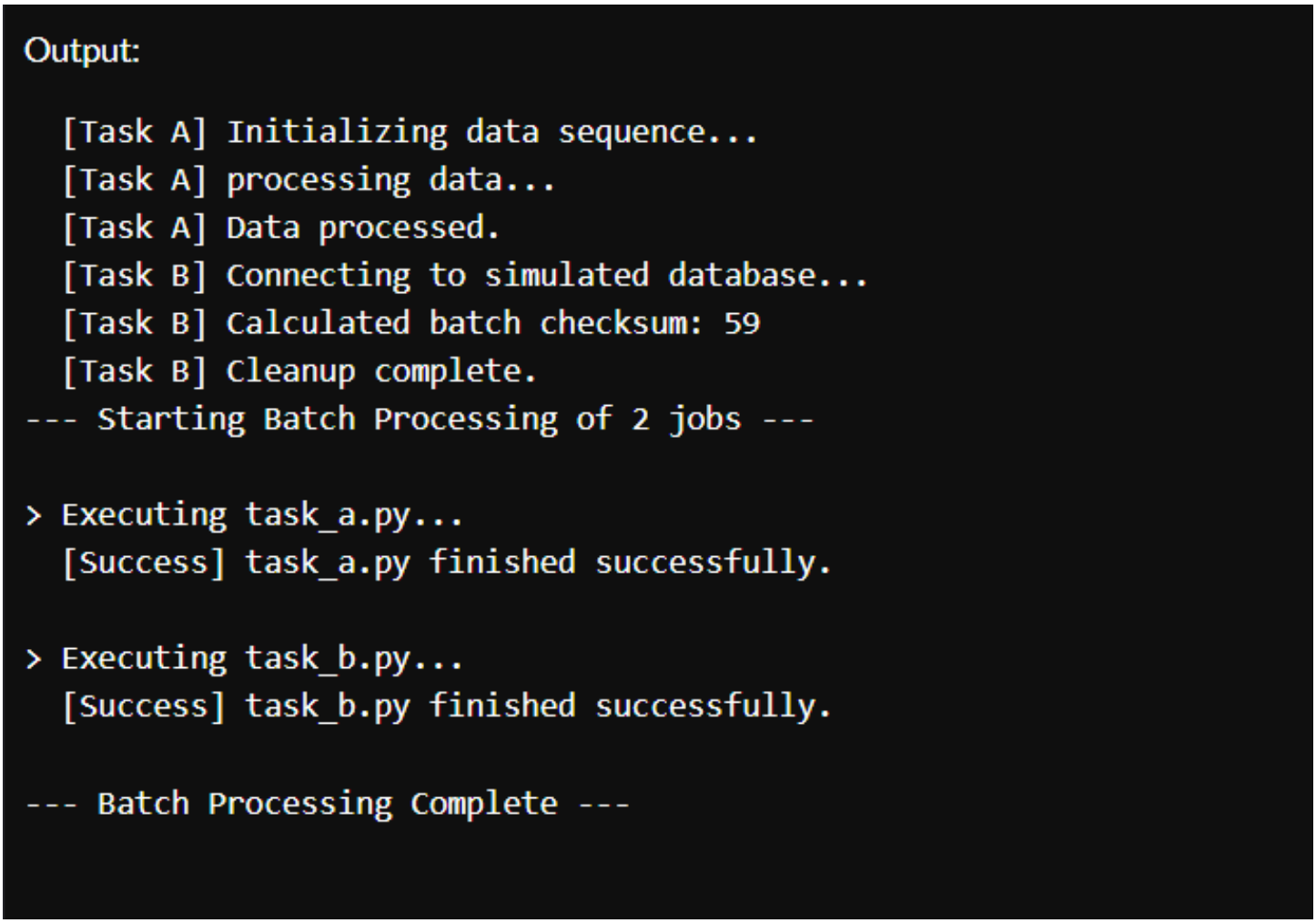
```
        print(f"\n> Executing {script}...")

        try:
            # subprocess.run is preferred over subprocess.call in modern Python
            # check=True raises an exception if the called script fails (returns non-zero exit code)
            result = subprocess.run([sys.executable, script], check=True, text=True)
            print(f"  [Success] {script} finished successfully.")

        except subprocess.CalledProcessError as e:
            print(f"  [Failed] {script} failed with error code {e.returncode}.")
        except Exception as e:
            print(f"  [Error] An unexpected error occurred: {e}")

    print("\n--- Batch Processing Complete ---")

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



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.

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.")

Code:

```
import multiprocessing
import logging
import time



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

```
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(processName)s - %(message)s'
    )
    print("Notice: File permission denied. Logging to console instead.")

def process_task(name):
    """Simulates a task running in a separate process."""
    logging.info(f"{name} started")
    time.sleep(2)
    logging.info(f"{name} terminated")

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

    # Create two separate processes
    p1 = multiprocessing.Process(target=process_task, args=("Process-1",))
    p2 = multiprocessing.Process(target=process_task, args=("Process-2",))

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

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

```
print("System Shutdown.")
```

```
Output:

Notice: File permission denied. Logging to console instead.
System Booting...
System Shutdown.

2025-11-25 18:35:00,392 - Process-2 - Process-2 started
2025-11-25 18:35:00,393 - Process-1 - Process-1 started
2025-11-25 18:35:02,393 - Process-2 - Process-2 terminated
2025-11-25 18:35:02,393 - Process-1 - Process-1 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.

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)

Code:

import os

import sys

import time

def run_ipc_demo():

"""

Demonstrates Inter-Process Communication (IPC) using os.pipe() and os.fork().

```
"""
print("--- Starting IPC Pipe Demo ---")

# Create a pipe
# r: file descriptor for reading
# w: file descriptor for writing
r, w = os.pipe()

print(f"Parent: Created pipe (Read FD: {r}, Write FD: {w})")

# Fork the process
# Returns 0 to the child process, and the child's PID to the parent process
pid = os.fork()

if pid > 0:
    # --- PARENT PROCESS ---
    print(f"Parent: Forked child process with PID {pid}")

    # Close the read end, as the parent will only write
    os.close(r)

    message = b"Hello from parent process!"
    print(f"Parent: Writing message to pipe: {message}")
    os.write(w, message)

    # Close the write end to signal EOF to the reader
    os.close(w)
```

```
        # Wait for the child to finish to avoid zombie processes
        completed_pid, status = os.wait()
        print(f"Parent: Child {completed_pid} finished.")

    else:
        # --- CHILD PROCESS ---
        # Close the write end, as the child will only read
        os.close(w)

        print("Child:  Waiting to read from pipe...")
        # Read from the pipe (buffer size 1024 bytes)
        buffer = os.read(r, 1024)

        print(f"Child:  Received message: '{buffer.decode()}'")

        # Close the read end
        os.close(r)

        # Exit cleanly
        sys.exit(0)

def run_exec_demo():
    """
    Demonstrates os.exec(), which replaces the current process with a new one.
    """
    print("\n--- Starting Exec Demo ---")
```

```
    pid = os.fork()

    if pid > 0:
        # Parent waits for child
        os.wait()
        print("Parent: Child process (which ran 'date') has finished.")
    else:
        # Child process replaces itself with the 'date' command
        print("Child:  Replacing self with 'date' command...")
        # execlp searches for 'date' in the PATH and executes it
        # Arguments: program name, arg0 (program name), [args...]
        os.execlp('date', 'date')

        # This line will never be reached if exec is successful
        print("Child:  Error - exec failed!")

if __name__ == "__main__":
    # Check if the OS supports fork (Unix/Linux/macOS only)
    if not hasattr(os, 'fork'):
        print("Error: os.fork() is not available on this operating system (Windows).")
        print("Please run this on Linux, macOS, or a Unix-based online compiler.")
        sys.exit(1)

    run_ipc_demo()
    time.sleep(1)
    run_exec_demo()
```

```
Output:

--- Starting IPC Pipe Demo ---
Parent: Created pipe (Read FD: 3, Write FD: 4)
Child:  Waiting to read from pipe...
Child:  Received message: 'Hello from parent process!'
Tue Nov 25 06:37:24 PM UTC 2025
--- Starting IPC Pipe Demo ---
Parent: Created pipe (Read FD: 3, Write FD: 4)
Parent: Forked child process with PID 29231
Parent: Writing message to pipe: b'Hello from parent process!'
Parent: Child 29231 finished.

--- Starting Exec Demo ---
Parent: Child process (which ran 'date') has 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.

#!/bin/bash

echo "Kernel Version:" uname -r

echo "User:" whoami

echo "Hardware Info:" lscpu | grep 'Virtualization'

Code:

```
import sys

import os


def detect_vm():
    """
    Detects if the script is running inside a Virtual Machine (VM)

    by checking common Linux system files and DMI information.
```

```
"""
print("--- VM Detection Utility ---")

is_vm = False
evidence = []

# 1. Check /proc/cpuinfo for the 'hypervisor' flag
if os.path.exists('/proc/cpuinfo'):
    try:
        with open('/proc/cpuinfo', 'r') as f:
            content = f.read()
            # 'hypervisor' flag is a strong indicator of virtualization
            if 'hypervisor' in content.lower():
                is_vm = True
                evidence.append("Found 'hypervisor' flag in /proc/cpuinfo")
    except Exception as e:
        print(f"Warning: Could not read /proc/cpuinfo: {e}")

# 2. Check DMI (Desktop Management Interface) product info
# Common vendors: VMware, VirtualBox, QEMU, Amazon EC2, Google, Microsoft
dmi_files = [
    '/sys/class/dmi/id/product_name',
    '/sys/class/dmi/id/sys_vendor',
    '/sys/class/dmi/id/bios_vendor'
]

vm_keywords = ['virtualbox', 'vmware', 'qemu', 'kvm', 'xen', 'amazon', 'google', 'innotek', 'openstack']
```

```
for file_path in dmi_files:
    if os.path.exists(file_path):
        try:
            with open(file_path, 'r') as f:
                content = f.read().strip().lower()
                for keyword in vm_keywords:
                    if keyword in content:
                        is_vm = True
                        evidence.append(f"Found '{content}' in {file_path}")
                        break
        except PermissionError:
            # Some systems restrict access to DMI data for non-root users
            pass
        except Exception:
            pass

# 3. Check for specific VM hardware artifacts (MAC addresses, etc - simplified here)
# (Skipped for brevity, focusing on system files)

# --- REPORT ---
if is_vm:
    print("\n[RESULT] Virtual Machine DETECTED.")
    print("Evidence:")
    for item in evidence:
        print(f"  - {item}")
else:
```

```
        print("\n[RESULT] No obvious signs of a Virtual Machine found.")
        print("(System appears to be Bare Metal or running in a container like Docker)")


if __name__ == "__main__":
    if sys.platform != 'linux':
        print("Error: This script is designed for Linux systems.")
    else:
        detect_vm()
```

```
Output:

--- VM Detection Utility ---

[RESULT] Virtual Machine DETECTED.
Evidence:
  - Found 'hypervisor' flag in /proc/cpuinfo
  - Found 'xen' in /sys/class/dmi/id/sys_vendor
  - Found 'xen' in /sys/class/dmi/id/bios_vendor
```

Task 5: CPU Scheduling Algorithms

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

Use existing Round Robin, FCFS, SJF, Priority scheduling Python codes from Lab 3)

Code:

```
from copy import deepcopy

class Process:
    def __init__(self, pid, arrival_time, burst_time, priority=0):
        self.pid = pid
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.priority = priority  # Lower number = Higher priority
        self.remaining_time = burst_time
        self.completion_time = 0
        self.waiting_time = 0
```

```
        self.turnaround_time = 0


def print_table(processes, algorithm_name):
    print(f"\n--- {algorithm_name} ---")
    print(f"{'PID':<5} {'Arr':<5} {'Burst':<6} {'Prio':<5} | {'Comp':<6} {'TAT':<5} {'WT':<5}")

    total_tat = 0
    total_wt = 0

    # Sort by PID for clean display
    display_order = sorted(processes, key=lambda x: x.pid)

    for p in display_order:
        print(f"{p.pid:<5} {p.arrival_time:<5} {p.burst_time:<6} {p.priority:<5} | {p.completion_time:<6}
{p.turnaround_time:<5} {p.waiting_time:<5}")
        total_tat += p.turnaround_time
        total_wt += p.waiting_time

    n = len(processes)
    if n > 0:
        print(f"Average TAT: {total_tat/n:.2f}")
        print(f"Average WT:  {total_wt/n:.2f}")
    print("-" * 50)

def fcfs(processes):
    """First Come First Serve"""
    # Sort by arrival time
    processes.sort(key=lambda x: x.arrival_time)
```

```
    current_time = 0
    for p in processes:
        if current_time < p.arrival_time:
            current_time = p.arrival_time

        p.completion_time = current_time + p.burst_time
        p.turnaround_time = p.completion_time - p.arrival_time
        p.waiting_time = p.turnaround_time - p.burst_time
        current_time = p.completion_time

    return processes

def sjf(processes):
    """Shortest Job First (Non-Preemptive)"""
    n = len(processes)
    completed = 0
    current_time = 0
    completed_list = []

    # Work on a copy
    remaining = list(processes)

    while completed < n:
        # Filter processes that have arrived by current_time
        available = [p for p in remaining if p.arrival_time <= current_time]
```

```
        if not available:
            # If no process has arrived, jump to next arrival
            current_time = min(remaining, key=lambda x: x.arrival_time).arrival_time
            continue

        # Select process with shortest burst time
        shortest = min(available, key=lambda x: x.burst_time)

        # Execute
        current_time += shortest.burst_time
        shortest.completion_time = current_time
        shortest.turnaround_time = shortest.completion_time - shortest.arrival_time
        shortest.waiting_time = shortest.turnaround_time - shortest.burst_time

        completed_list.append(shortest)
        remaining.remove(shortest)
        completed += 1

    return completed_list

def priority_scheduling(processes):
    """Priority Scheduling (Non-Preemptive) - Lower value is higher priority"""
    n = len(processes)
    completed = 0
    current_time = 0
    completed_list = []
    remaining = list(processes)
```

```
    while completed < n:
        available = [p for p in remaining if p.arrival_time <= current_time]

        if not available:
            current_time = min(remaining, key=lambda x: x.arrival_time).arrival_time
            continue

        # Select process with highest priority (lowest number)
        # If tie, break by arrival time
        selected = min(available, key=lambda x: (x.priority, x.arrival_time))

        current_time += selected.burst_time
        selected.completion_time = current_time
        selected.turnaround_time = selected.completion_time - selected.arrival_time
        selected.waiting_time = selected.turnaround_time - selected.burst_time

        completed_list.append(selected)
        remaining.remove(selected)
        completed += 1

    return completed_list

def round_robin(processes, quantum):
    """Round Robin (Preemptive)"""
    # Sort by arrival initially
    processes.sort(key=lambda x: x.arrival_time)
```

```
n = len(processes)
current_time = 0
completed = 0
queue = []

# We need a copy because we modify remaining_time
rr_procs = deepcopy(processes)

# Pushing initial processes to queue
# Map to track which processes are already 'seen' or in queue
in_queue_or_done = {p.pid: False for p in rr_procs}

# Helper to find process object by pid in our deepcopy list
def get_proc(pid):
    return next(p for p in rr_procs if p.pid == pid)

# Initial loading of queue
# Start time is arrival of first process
if n > 0:
    current_time = rr_procs[0].arrival_time

# Add all processes that arrive at start time
for p in rr_procs:
    if p.arrival_time <= current_time and not in_queue_or_done[p.pid]:
        queue.append(p)
        in_queue_or_done[p.pid] = True
```

```
while completed < n:
    if not queue:
        # If queue is empty but processes remain, jump time
        remaining_procs = [p for p in rr_procs if p.remaining_time > 0]
        if not remaining_procs: break
        next_arrival = min(remaining_procs, key=lambda x: x.arrival_time)
        current_time = next_arrival.arrival_time
        queue.append(next_arrival)
        in_queue_or_done[next_arrival.pid] = True

    p = queue.pop(0)

    # Execute for quantum or remaining time
    exec_time = min(quantum, p.remaining_time)
    p.remaining_time -= exec_time
    current_time += exec_time

    # Check for new arrivals while this process was executing
    # Important: Add them to queue BEFORE adding the current process back
    for new_p in rr_procs:
        if (new_p.arrival_time <= current_time and
            not in_queue_or_done[new_p.pid] and
            new_p.remaining_time > 0):
            queue.append(new_p)
            in_queue_or_done[new_p.pid] = True
```

```
        # If process still has time, add back to queue
        if p.remaining_time > 0:
            queue.append(p)
        else:
            # Process finished
            p.completion_time = current_time
            p.turnaround_time = p.completion_time - p.arrival_time
            p.waiting_time = p.turnaround_time - p.burst_time
            completed += 1

    return rr_procs

def main():
    # Sample Data: [PID, Arrival, Burst, Priority]
    # Priority: 1 is highest, 4 is lowest
    raw_data = [
        # PID, Arr, Burst, Prio
        [1, 0, 10, 3],
        [2, 2, 5, 1],
        [3, 4, 8, 2],
        [4, 5, 2, 4]
    ]

    print("--- INPUT DATA ---")
    print(f"{'PID':<5} {'Arr':<5} {'Burst':<6} {'Prio':<5}")
    for row in raw_data:
        print(f"{row[0]:<5} {row[1]:<5} {row[2]:<6} {row[3]:<5}")
```

```
    # 1. FCFS
    procs = [Process(*row) for row in raw_data]
    print_table(fcfs(procs), "FCFS")

    # 2. SJF
    procs = [Process(*row) for row in raw_data]
    print_table(sjf(procs), "SJF (Non-Preemptive)")

    # 3. Priority
    procs = [Process(*row) for row in raw_data]
    print_table(priority_scheduling(procs), "Priority (Non-Preemptive)")

    # 4. Round Robin
    procs = [Process(*row) for row in raw_data]
    print_table(round_robin(procs, quantum=2), "Round Robin (Time Quantum = 2)")

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

Output:

```
--- INPUT DATA ---
PID   Arr   Burst  Prio
1     0     10     3
2     2     5      1
3     4     8      2
4     5     2      4

--- FCFS ---
PID   Arr   Burst  Prio  | Comp   TAT   WT
1     0     10     3     | 10     10    0
2     2     5      1     | 15     13    8
3     4     8      2     | 23     19    11
4     5     2      4     | 25     20    18
Average TAT: 15.50
Average WT:  9.25
-------------------------------------------------

--- SJF (Non-Preemptive) ---
PID   Arr   Burst  Prio  | Comp   TAT   WT
1     0     10     3     | 10     10    0
2     2     5      1     | 17     15    10
3     4     8      2     | 25     21    13
4     5     2      4     | 12     7     5
Average TAT: 13.25
```

```
Average WT:  7.00
-------------------------------------------------

--- Priority (Non-Preemptive) ---
PID   Arr   Burst  Prio  | Comp   TAT   WT
1     0     10     3     | 10     10    0
2     2     5      1     | 15     13    8
3     4     8      2     | 23     19    11
4     5     2      4     | 25     20    18
Average TAT: 15.50
Average WT:  9.25
-------------------------------------------------

--- Round Robin (Time Quantum = 2) ---
PID   Arr   Burst  Prio  | Comp   TAT   WT
1     0     10     3     | 23     23    13
2     2     5      1     | 17     15    10
3     4     8      2     | 25     21    13
4     5     2      4     | 12     7     5
Average TAT: 16.50
Average WT:  10.25
-------------------------------------------------
```