OS LAB ASSIGNMENT – 4

Task 1 . Write a python program to implement Priority and Round Robin scheduling algorithms. Compute average waiting and turnaround times.

Code for Priority scheduling:

```
  GNU nano 8.4                                          prrr.py *
n = int(input("Enter number of processes: "))
processes = []
for i in range(n):
    bt = int(input(f"Enter Burst Time for Process {i+1}: "))
    pr = int(input(f"Enter Priority for Process {i+1} (lower = higher priority): "))
    processes.append([i+1, bt, pr])

# Sort by priority
processes.sort(key=lambda x: x[2])

waiting_time = [0] * n
turnaround_time = [0] * n

# Calculate waiting time and turnaround time
for i in range(1, n):
    waiting_time[i] = waiting_time[i-1] + processes[i-1][1]

for i in range(n):
    turnaround_time[i] = waiting_time[i] + processes[i][1]

avg_wt = sum(waiting_time) / n
avg_tat = sum(turnaround_time) / n

print("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time")
for i in range(n):
    print(f"P{processes[i][0]}\t{processes[i][1]}\t\t{processes[i][2]}\t\t{waiting_time[i]}\t\t{turnaround_time[i]}")
```

```
print(f"\nAverage Waiting Time: {avg_wt:.2f}")
print(f"Average Turnaround Time: {avg_tat:.2f}")
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ nano prrr.py

┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python prrr.py
Enter number of processes: 3
Enter Burst Time for Process 1: 10
Enter Priority for Process 1 (lower = higher priority): 2
Enter Burst Time for Process 2: 5
Enter Priority for Process 2 (lower = higher priority): 1
Enter Burst Time for Process 3: 8
Enter Priority for Process 3 (lower = higher priority): 3

Process Burst Time      Priority        Waiting Time    Turnaround Time
P2      5               1               0               5
P1      10              2               5               15
P3      8               3               15              23

Average Waiting Time: 6.67
Average Turnaround Time: 14.33
```

Code for Round Robin Scheduling:

```
# Round Robin Scheduling
n = int(input("Enter number of processes: "))
processes = []
for i in range(n):
    bt = int(input(f"Enter Burst Time for Process {i+1}: "))
    processes.append([i+1, bt])

quantum = int(input("Enter Time Quantum: "))

remaining_bt = [p[1] for p in processes]
waiting_time = [0] * n
time = 0

# Run until all processes are complete
while True:
    done = True
    for i in range(n):
        if remaining_bt[i] > 0:
            done = False
            if remaining_bt[i] > quantum:
                time += quantum
                remaining_bt[i] -= quantum
            else:
                time += remaining_bt[i]
                waiting_time[i] = time - processes[i][1]
                remaining_bt[i] = 0
```

```
    if done:
        break

# Turnaround Time = Waiting + Burst
turnaround_time = [waiting_time[i] + processes[i][1] for i in range(n)]

# Compute averages
avg_wt = sum(waiting_time) / n
avg_tat = sum(turnaround_time) / n

# Print results
print("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time")
for i in range(n):
    print(f"P{processes[i][0]}\t{processes[i][1]}\t\t{waiting_time[i]}\t\t{turnaround_time[i]}")

print(f"\nAverage Waiting Time: {avg_wt:.2f}")
print(f"Average Turnaround Time: {avg_tat:.2f}")
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ nano roundrobin.py

┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python roundrobin.py
Enter number of processes: 3
Enter Burst Time for Process 1: 10
Enter Burst Time for Process 2: 5
Enter Burst Time for Process 3: 8
Enter Time Quantum: 5

Process Burst Time      Waiting Time    Turnaround Time
P1      10              10              20
P2      5               5               10
P3      8               15              23

Average Waiting Time: 10.00
Average Turnaround Time: 17.67
```

Task 2. Simulate worst-fit, best-fit and first-fit memory allocation strategies.

Code:

```
# Memory Allocation Simulation: First-Fit, Best-Fit, Worst-Fit

def first_fit(blocks, processes):
    allocation = [-1] * len(processes)
    temp_blocks = blocks.copy()

    for i in range(len(processes)):
        for j in range(len(temp_blocks)):
            if temp_blocks[j] >= processes[i]:
                allocation[i] = j
                temp_blocks[j] -= processes[i]
                break
    return allocation


def best_fit(blocks, processes):
    allocation = [-1] * len(processes)
    temp_blocks = blocks.copy()

    for i in range(len(processes)):
        best_idx = -1
        for j in range(len(temp_blocks)):
            if temp_blocks[j] >= processes[i]:
                if best_idx == -1 or temp_blocks[j] < temp_blocks[best_idx]:
                    best_idx = j
        if best_idx != -1:
            allocation[i] = best_idx
            temp_blocks[best_idx] -= processes[i]
    return allocation


def worst_fit(blocks, processes):
    allocation = [-1] * len(processes)
    temp_blocks = blocks.copy()

    for i in range(len(processes)):
        worst_idx = -1
        for j in range(len(temp_blocks)):
            if temp_blocks[j] >= processes[i]:
                if worst_idx == -1 or temp_blocks[j] > temp_blocks[worst_idx]:
                    worst_idx = j
        if worst_idx != -1:
            allocation[i] = worst_idx
            temp_blocks[worst_idx] -= processes[i]
    return allocation


# Input example
blocks = [100, 500, 200, 300, 600]
processes = [212, 417, 112, 426]
print("Memory Blocks:", blocks)
print("Processes:", processes)
```

```python
# Apply algorithms
ff_allocation = first_fit(blocks, processes)
bf_allocation = best_fit(blocks, processes)
wf_allocation = worst_fit(blocks, processes)

# Print results
def print_allocation(name, allocation):
    print(f"\n{name} Allocation:")
    print("Process No.\tProcess Size\tBlock No.")
    for i in range(len(processes)):
        if allocation[i] != -1:
            print(f"{i+1}\t\t{processes[i]}\t\t{allocation[i]+1}")
        else:
            print(f"{i+1}\t\t{processes[i]}\t\tNot Allocated")

print_allocation("First Fit", ff_allocation)
print_allocation("Best Fit", bf_allocation)
print_allocation("Worst Fit", wf_allocation)
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python memoryallocation.py
Memory Blocks: [100, 500, 200, 300, 600]
Processes: [212, 417, 112, 426]

First Fit Allocation:
Process No.     Process Size    Block No.
1               212             2
2               417             5
3               112             2
4               426             Not Allocated

Best Fit Allocation:
Process No.     Process Size    Block No.
1               212             4
2               417             2
3               112             3
4               426             5

Worst Fit Allocation:
Process No.     Process Size    Block No.
1               212             5
2               417             2
3               112             5
4               426             Not Allocated
```

Task 3. Implement MFT and MVT Memory Management

Code for MFT (Multiprogramming with Fixed Tasks):

```
# MFT (Multiprogramming with Fixed Tasks)

memory_size = int(input("Enter total memory size (KB): "))
partition_size = int(input("Enter partition size (KB): "))
num_partitions = memory_size // partition_size
remaining_memory = memory_size % partition_size

print(f"\nNumber of partitions: {num_partitions}")
print(f"Remaining memory (unused): {remaining_memory} KB\n")

processes = []
n = int(input("Enter number of processes: "))

for i in range(n):
    p = int(input(f"Enter memory required for process {i+1} (KB): "))
    processes.append(p)

allocated = [False]*n
internal_frag = 0
external_frag = 0

for i in range(n):
    if i < num_partitions:
        if processes[i] > partition_size:
            print(f"Process {i+1} of size {processes[i]} KB cannot be allocated.")
        else:
```

```
        else:
            allocated[i] = True
            internal_frag += partition_size - processes[i]
            print(f"Process {i+1} allocated in Partition {i+1}.")
    else:
        print(f"No free partition for Process {i+1}.")

print(f"\nTotal Internal Fragmentation = {internal_frag} KB")
print(f"Total External Fragmentation = {remaining_memory} KB")
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python mft.py
Enter total memory size (KB): 1000
Enter partition size (KB): 200

Number of partitions: 5
Remaining memory (unused): 0 KB

Enter number of processes: 6
Enter memory required for process 1 (KB): 130
Enter memory required for process 2 (KB): 220
Enter memory required for process 3 (KB): 140
Enter memory required for process 4 (KB): 110
Enter memory required for process 5 (KB): 190
Enter memory required for process 6 (KB): 200
Process 1 allocated in Partition 1.
Process 2 of size 220 KB cannot be allocated.
Process 3 allocated in Partition 3.
Process 4 allocated in Partition 4.
Process 5 allocated in Partition 5.
No free partition for Process 6.

Total Internal Fragmentation = 230 KB
Total External Fragmentation = 0 KB
```

Code for MVT (Multiprogramming with Variable Tasks):

```python
# MVT (Multiprogramming with Variable Tasks)

memory_size = int(input("Enter total memory size (KB): "))
allocated_memory = 0
n = int(input("Enter number of processes: "))

for i in range(n):
    process_size = int(input(f"Enter memory required for process {i+1} (KB): "))
    if allocated_memory + process_size <= memory_size:
        allocated_memory += process_size
        print(f"Process {i+1} of size {process_size} KB allocated.")
    else:
        print(f"Not enough memory for Process {i+1}.")
        break

print(f"\nTotal memory allocated: {allocated_memory} KB")
print(f"Total memory remaining: {memory_size - allocated_memory} KB")
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ nano mvt.py

┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python mvt.py
Enter total memory size (KB): 10000
Enter number of processes: 4
Enter memory required for process 1 (KB): 200
Process 1 of size 200 KB allocated.
Enter memory required for process 2 (KB): 300
Process 2 of size 300 KB allocated.
Enter memory required for process 3 (KB): 400
Process 3 of size 400 KB allocated.
Enter memory required for process 4 (KB): 250
Process 4 of size 250 KB allocated.

Total memory allocated: 1150 KB
Total memory remaining: 8850 KB
```

Task 4 : Batch Processing Simulation (Python)

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

Code for Batch Processing:

```python
import os
import subprocess

scripts = [
    r"/mnt/c/Users/Kashish Pundir/OneDrive/Desktop/Python/Typecasting.py",
    r"/mnt/c/Users/Kashish Pundir/OneDrive/Desktop/Python/String.py"
]

print("=== Batch Processing Simulation Started ===\n")

for script in scripts:
    if os.path.exists(script):
        print(f"Executing {script}...")
        result = subprocess.run(["python", script], capture_output=True, text=True)

        print(f"--- Output of {script} ---")
        print(result.stdout)
        if result.stderr:
            print(f"--- Errors in {script} ---")
            print(result.stderr)
        print("-" * 40)
    else:
        print(f"⚠️ File not found: {script}")
    print()
print("=== Batch Processing Simulation Completed ===")
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python batchsimulation.py
=== Batch Processing Simulation Started ===

Executing /mnt/c/Users/Kashish Pundir/OneDrive/Desktop/Python/Typecasting.py...
--- Output of /mnt/c/Users/Kashish Pundir/OneDrive/Desktop/Python/Typecasting.py ---
3.0

----------------------------------------

Executing /mnt/c/Users/Kashish Pundir/OneDrive/Desktop/Python/String.py...
--- Output of /mnt/c/Users/Kashish Pundir/OneDrive/Desktop/Python/String.py ---
hello
Kashish
6
13
['hello', 'Kashish']

----------------------------------------

=== Batch Processing Simulation Completed ===
```

Task 5: System Startup and Logging

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

Code:

```python
import multiprocessing
import time
import logging
from datetime import datetime

# --- Configure logging ---
logging.basicConfig(
    filename="system_startup.log",
    level=logging.INFO,
    format="%(asctime)s - %(processName)s - %(message)s"
)

# --- Define dummy startup process ---
def system_service(service_name, duration):
    logging.info(f"{service_name} started.")
    time.sleep(duration)  # Simulate time taken for startup
    logging.info(f"{service_name} finished startup.")

# --- Main simulation ---
if __name__ == "__main__":
    print("=== System Startup Simulation Initiated ===\n")

    # Define system services (name, duration)
    services = [
        ("Network Manager", 2),
        ("Database Service", 3),
        ("Authentication Service", 1.5),
        ("File System Monitor", 2.5),
        ("Security Daemon", 1)
    ]

    # Create process list
    processes = []
    for name, duration in services:
        p = multiprocessing.Process(target=system_service, args=(name, duration), name=name)
        processes.append(p)
        p.start()  # Start the process

    # Wait for all to finish
    for p in processes:
        p.join()

    print("=== All System Services Started Successfully ===")
    print("Logs saved in 'system_startup.log'")
```

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python Startupsimulation.py
=== System Startup Simulation Initiated ===

=== All System Services Started Successfully ===
Logs saved in 'system_startup.log'
```

Task 6: 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.

Code:

```python
import os
def main():
    print("=== System Calls and IPC Simulation ===")

    # Create a pipe (two file descriptors)
    read_fd, write_fd = os.pipe()

    # Fork a new process
    pid = os.fork()

    if pid > 0:
        # 🟢 Parent process
        os.close(read_fd)  # Close unused read end
        message = "Hello from Parent Process!"
        print(f"[Parent] Sending message to child: {message}")

        # Encode and write message into pipe
        os.write(write_fd, message.encode())
        os.close(write_fd)  # Close write end after sending

        # Wait for child process to finish
        os.wait()
        print("[Parent] Child process has finished execution.")
```

```python
    else:
        # 🔵 Child process
        os.close(write_fd)  # Close unused write end

        # Read message from parent through pipe
        read_message = os.read(read_fd, 1024).decode()
        print(f"[Child] Received message from parent: {read_message}")

        # Simulate exec() - replacing the process with a new one
        print("[Child] Executing new program using exec()...")
        os.execlp("echo", "echo", "Child process executed new program via exec()!")

        # (The following line won't execute since exec() replaces the current process)
        os.close(read_fd)

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

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python systemcalls.py
=== System Calls and IPC Simulation ===
[Parent] Sending message to child: Hello from Parent Process!
[Child] Received message from parent: Hello from Parent Process!
[Child] Executing new program using exec()...
Child process executed new program via exec()!
[Parent] Child process has finished execution.
```

Task 7: 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.

Code for Shell Interaction:

```bash
#!/bin/bash
# ====================================
# System Details Script (system_info.sh)
# ====================================

echo "=== System Information ==="
echo "Hostname: $(hostname)"
echo "OS: $(lsb_release -d | cut -f2)"
echo "Kernel: $(uname -r)"
echo "Architecture: $(uname -m)"
echo "CPU Info: $(lscpu | grep 'Model name' | cut -d ':' -f2)"
echo "Total Memory: $(free -h | grep Mem | awk '{print $2}')"
echo "Disk Usage: $(df -h / | awk 'NR==2 {print $5}')"
echo "Uptime: $(uptime -p)"
echo "====================================="
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ chmod +x system_info.sh
./system_info.sh
=== System Information ===
Hostname: DESKTOP-TOVUP69
OS: Kali GNU/Linux Rolling
Kernel: 6.6.87.2-microsoft-standard-WSL2
Architecture: x86_64
CPU Info:                          Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz
Total Memory: 7.7Gi
Disk Usage: 1%
Uptime: up 41 minutes
=====================================
```

Code For VM Detection:

```python
import os
import platform
import subprocess

def detect_vm():
    print("=== Virtual Machine Detection ===")

    try:
        manufacturer = subprocess.getoutput("sudo dmidecode -s system-manufacturer")
        product_name = subprocess.getoutput("sudo dmidecode -s system-product-name")

        print(f"Manufacturer: {manufacturer}")
        print(f"Product Name: {product_name}")

        vm_indicators = ["virtualbox", "vmware", "qemu", "kvm", "hyper-v", "xen"]

        if any(vm in manufacturer.lower() for vm in vm_indicators) or \
           any(vm in product_name.lower() for vm in vm_indicators):
            print("The system appears to be running inside a Virtual Machine.")
        else:
            print("The system appears to be running on physical hardware.")
    except Exception as e:
        print("Error detecting VM:", e)

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

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python vm-detection.py
=== Virtual Machine Detection ===
[sudo] password for Kashish:
Manufacturer:
Product Name:
The system appears to be running on physical hardware.
```

Task 8: CPU Scheduling Algorithms

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

Code for FCFS:

```
# FCFS (First Come, First Server): Processes are executed in the order they arrive.

# Code:

def fcfs(processes, arrival_times, burst_times):
    n = len(processes)
    start_times = [0] * n
    completion_times = [0] * n
    waiting_times = [0] * n
    turnaround_times = [0] * n

    current_time = 0
    for i in range(n):
        if current_time < arrival_times[i]:
            current_time = arrival_times[i]
        start_times[i] = current_time
        completion_times[i] = current_time + burst_times[i]
        turnaround_times[i] = completion_times[i] - arrival_times[i]
        waiting_times[i] = turnaround_times[i] - burst_times[i]
        current_time += burst_times[i]

    print("Process\tArrival\tBurst\tStart\tCompletion\tWaiting\tTurnaround")
    for i in range(n):
        print(f"{processes[i]}\t{arrival_times[i]}\t{burst_times[i]}\t{start_times[i]}\t{completion_times[i]}\t\t{wait

processes = ['P1', 'P2', 'P3']
arrival_times = [0, 2, 4]
burst_times = [5, 3, 1]
fcfs(processes, arrival_times, burst_times)
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ nano fcfs.py

┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python fcfs.py
Process Arrival Burst   Start   Completion      Waiting Turnaround
P1      0       5       0       5               0       5
P2      2       3       5       8               3       6
P3      4       1       8       9               4       5
```

Code for SJF:

```
# SJF: Processes with the shortest burst time are executed first.

# CODE:

def sjf(processes, arrival_times, burst_times):
    n = len(processes)
    completed = [False] * n
    current_time = 0
    completed_count = 0

    start_times = [0] * n
    completion_times = [0] * n
    waiting_times = [0] * n
    turnaround_times = [0] * n

    while completed_count < n:
        # Select process with minimum burst time among those that have arrived and not completed
        idx = -1
        min_burst = float('inf')
        for i in range(n):
            if arrival_times[i] <= current_time and not completed[i]:
                if burst_times[i] < min_burst:
                    min_burst = burst_times[i]
                    idx = i

        if idx == -1:
            current_time += 1
        else:
            start_times[idx] = current_time
```

```
        else:
            start_times[idx] = current_time
            completion_times[idx] = current_time + burst_times[idx]
            turnaround_times[idx] = completion_times[idx] - arrival_times[idx]
            waiting_times[idx] = turnaround_times[idx] - burst_times[idx]
            current_time += burst_times[idx]
            completed[idx] = True
            completed_count += 1

    print("Process\tArrival\tBurst\tStart\tCompletion\tWaiting\tTurnaround")
    for i in range(n):
        print(f"{processes[i]}\t{arrival_times[i]}\t{burst_times[i]}\t{start_times[i]}\t{com

# Example usage:
processes = ['P1', 'P2', 'P3']
arrival_times = [0, 1, 2]
burst_times = [6, 8, 7]
sjf(processes, arrival_times, burst_times)
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ nano SJF.py

┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python SJF.py
Process Arrival Burst   Start   Completion      Waiting Turnaround
P1      0       6       0       6               0       6
P2      1       8       13      21              12      20
P3      2       7       6       13              4       11
```

Code for Round Robin:

```
# Round Robin: Each process gets a fixed time quantum. Processes are executed in a cyclic order.

# CODE:

from collections import deque

def round_robin(processes, arrival_times, burst_times, time_quantum):
    n = len(processes)
    remaining_bt = burst_times[:]
    current_time = 0
    waiting_times = [0] * n
    completion_times = [0] * n
    turnaround_times = [0] * n
    start_times = [-1] * n

    queue = deque()
    visited = [False] * n

    # Add processes that arrive at time 0
    for i in range(n):
        if arrival_times[i] <= current_time and not visited[i]:
            queue.append(i)
            visited[i] = True

    while queue:
        i = queue.popleft()

        if start_times[i] == -1:
            start_times[i] = max(current_time, arrival_times[i])
            current_time = start_times[i]

        exec_time = min(time_quantum, remaining_bt[i])
        remaining_bt[i] -= exec_time
        current_time += exec_time

        # Add newly arrived processes to queue
        for j in range(n):
            if arrival_times[j] <= current_time and not visited[j]:
                queue.append(j)
                visited[j] = True

        if remaining_bt[i] > 0:
            queue.append(i)
        else:
            completion_times[i] = current_time
            turnaround_times[i] = completion_times[i] - arrival_times[i]
            waiting_times[i] = turnaround_times[i] - burst_times[i]

    print("Process\tArrival\tBurst\tStart\tCompletion\tWaiting\tTurnaround")
    for i in range(n):
        print(f"{processes[i]}\t{arrival_times[i]}\t{burst_times[i]}\t{start_times[i]}\t{compl

processes = ['P1', 'P2', 'P3']
arrival_times = [0, 1, 2]
burst_times = [10, 5, 8]
time_quantum = 3
round_robin(processes, arrival_times, burst_times, time_quantum)
```

Output:

```
┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ nano rr.py

┌──(Kashish㉿DESKTOP-TOVUP69)-[~]
└─$ python rr.py
Process Arrival Burst   Start   Completion      Waiting Turnaround
P1      0       10      0       23              13      23
P2      1       5       3       14              8       13
P3      2       8       6       22              12      20
```