Operating Systems Lab Assignment - 3 Report

Name: Yashraj Singh
Roll no. 2301730163
Course Code: ENCS351
Program : B.Tech CSE(AI/ML)-C

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

Objective:

The objective of this lab is to understand and demonstrate Interprocess Communication (IPC) using:

- Pipes (unidirectional communication)
- Shared Memory (fast memory-based communication)
- Synchronization techniques to avoid race conditions
- Parent–Child communication models
- Process creation and data exchange

This experiment helps simulate how operating systems allow multiple processes to exchange data efficiently.

Tools Used:

Python 3.x

Linux Environment (Ubuntu/WSL via Terminal) os module – for pipe creation and process forking multiprocessing module – for shared memory and synchronization logging module time module

Tasks Performed:

- **Task 1(CPU Scheduling with Gantt Chart):**

Description: A Python program was written to simulate:

- Priority Scheduling (lower number → higher priority)
- Round Robin Scheduling using fixed time quantum

The program accepts burst time, priority values, and process counts. It calculates:

• Waiting Time (WT)

• Turnaround Time (TAT)

• Average WT and TAT

• Scheduling order as Gantt Chart (printed in sequence) Outcome:

Correct scheduling order with computed WT and TAT for all processes.

INPUT:

os3task1.py > ...

```python
# Priority Scheduling Simulation
processes = []
n = int(input("Enter number of processes: "))
for i in range(n):
    bt = int(input(f"Enter Burst Time for P{i+1}: "))
    pr = int(input(f"Enter Priority (lower number = higher priority) for P{i+1}: "))
    processes.append((i+1, bt, pr))
processes.sort(key=lambda x: x[2])
wt = 0
total_wt = 0
total_tt = 0
print("\nPriority Scheduling:")
print("PID\tBT\tPriority\tWT\tTAT")
for pid, bt, pr in processes:
    tat = wt + bt
    print(f"{pid}\t{bt}\t{pr}\t\t{wt}\t{tat}")
    total_wt += wt
    total_tt += tat
    wt += bt
print(f"Average Waiting Time: {total_wt / n}")
print(f"Average Turnaround Time: {total_tt / n}")
```

OUTPUT:

```
Enter number of processes: 4
Enter Burst Time for P1: 10
Enter Priority (lower number = higher priority) for P1: 5
Enter Burst Time for P2: 20
Enter Priority (lower number = higher priority) for P2: 6
Enter Burst Time for P3: 30
Enter Priority (lower number = higher priority) for P3: 7
Enter Burst Time for P4: 40
Enter Priority (lower number = higher priority) for P4: 8

Priority Scheduling:
PID     BT      Priority        WT      TAT
1       10      5               0       10
2       20      6               10      30
3       30      7               30      60
4       40      8               60      100
Average Waiting Time: 25.0
Average Turnaround Time: 50.0
```

- **Task 2 (Sequential File Allocation):**

  Description: A Python script was implemented to simulate block- wise sequential file allocation.

  The user provides:

- Total blocks
- Starting block
- File length

  The system checks contiguous free space and allocates the file if possible.

  Outcome:

  Displays allocation success or failure depending on block availability.

  INPUT:

os3task2.py > ...

```python
total_blocks = int(input("Enter total number of blocks: "))
block_status = [0] * total_blocks

n = int(input("Enter number of files: "))
for i in range(n):
    start = int(input(f"Enter starting block for file {i+1}: "))
    length = int(input(f"Enter length of file {i+1}: "))
    allocated = True
    for j in range(start, start+length):
        if j >= total_blocks or block_status[j] == 1:
            allocated = False
            break
    if allocated:
        for j in range(start, start+length):
            block_status[j] = 1
        print(f"File {i+1} allocated from block {start} to {start+length-1}")
    else:
        print(f"File {i+1} cannot be allocated.")
```

OUTPUT:

```
Enter total number of blocks: 4
Enter number of files: 3
Enter starting block for file 1: 1
Enter length of file 1: 10
File 1 cannot be allocated.
Enter starting block for file 2: 2
Enter length of file 2: 20
File 2 cannot be allocated.
Enter starting block for file 3: 3
Enter length of file 3: 1
File 3 allocated from block 3 to 3
```

- **Task 3 (Indexed File Allocation):**

  Description: Simulated indexed allocation by assigning:

- Index block
- List of data blocks

  The system verifies block availability and assigns the index → data block mapping.

  Outcome:

Correct allocation table showing index block → data blocks.

INPUT:

```
os3task3.py > ...
total_blocks = int(input("Enter total number of blocks: "))
block_status = [0] * total_blocks
n = int(input("Enter number of files: "))
for i in range(n):
    index = int(input(f"Enter index block for file {i+1}: "))
    if block_status[index] == 1:
        print("Index block already allocated.")
        continue
    count = int(input("Enter number of data blocks: "))
    data_blocks = list(map(int, input("Enter block numbers: ").split()))
    if any(block_status[blk] == 1 for blk in data_blocks) or len(data_blocks) != count:
        print("Block(s) already allocated or invalid input.")
        continue
    block_status[index] = 1
    for blk in data_blocks:
        block_status[blk] = 1
    print(f"File {i+1} allocated with index block {index} -> {data_blocks}")
```

OUTPUT:

```
Enter total number of blocks: 20
Enter number of files: 2
Enter index block for file 1: 5
Enter number of data blocks: 3
Enter block numbers: 2 4 6
File 1 allocated with index block 5 -> [2, 4, 6]
Enter index block for file 2: 9
Enter number of data blocks: 2
Enter block numbers: 1 3
File 2 allocated with index block 9 -> [1, 3]
```

- **Task 4 (Contiguous Memory Allocation):**

  Description: Simulated:

- First-Fit
- Best-Fit
- Worst-Fit

For each strategy, partitions and process sizes were taken as input. Processes were allocated based on the strategy rules, and remaining partition size was updated.

Outcome:

Displayed which process was allocated to which partition for all strategies.

INPUT:

os3task4.py > ...

```python
def allocate_memory(strategy):
    partitions = list(map(int, input("Enter partition sizes: ").split()))
    processes = list(map(int, input("Enter process sizes: ").split()))
    allocation = [-1] * len(processes)

    for i, psize in enumerate(processes):
        idx = -1
        if strategy == "first":
            for j, part in enumerate(partitions):
                if part >= psize:
                    idx = j
                    break
        elif strategy == "best":
            best_fit = float("inf")
            for j, part in enumerate(partitions):
                if part >= psize and part < best_fit:
                    best_fit = part
                    idx = j
        elif strategy == "worst":
            worst_fit = -1
            for j, part in enumerate(partitions):
                if part >= psize and part > worst_fit:
                    worst_fit = part
                    idx = j
        if idx != -1:
            allocation[i] = idx
            partitions[idx] -= psize

    for i, a in enumerate(allocation):
        if a != -1:
            print(f"Process {i+1} allocated in Partition {a+1}")
        else:
            print(f"Process {i+1} cannot be allocated")

allocate_memory("first")
allocate_memory("best")
allocate_memory("worst")
```

OUTPUT:

```
Enter partition sizes: 100 500 200 300 600
Enter process sizes: 212 417 112 426
Process 1 allocated in Partition 2
Process 2 allocated in Partition 5
Process 3 allocated in Partition 2
Process 4 cannot be allocated
Enter partition sizes: 100 500
Enter process sizes: 50 400
Process 1 allocated in Partition 1
Process 2 allocated in Partition 2
Enter partition sizes: 50 100 150
Enter process sizes: 50 150 100
Process 1 allocated in Partition 3
Process 2 cannot be allocated
Process 3 allocated in Partition 2
```

- **Task 5 (MFT & MVT Memory Management):**

  Description: Two memory management schemes were simulated:

  MFT (Multiprogramming with Fixed Tasks)

- Total memory is divided into fixed-size partitions
- Only processes smaller than or equal to partition size can be allocated

  MVT (Multiprogramming with Variable Tasks)

- memory allocated dynamically
- fragmentation reduces as partitions are variable Outcome:

  Processes were allocated sequentially based on memory availability.

  System displayed whether memory was sufficient or not.

  INPUT:

os3task5.py > ...

```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()
```

OUTPUT:

```
MFT Simulation:
Enter total memory size: 100
Enter partition size: 30
Enter number of processes: 2
Memory divided into 3 partitions
Enter size of Process 1: 25
Process 1 allocated.
Enter size of Process 2: 35
Process 2 too large for fixed partition.

MVT Simulation:
Enter total memory size: 100
Enter number of processes: 2
Enter size of Process 1: 25
Process 1 allocated.
Enter size of Process 2: 35
Process 2 allocated.
```

Outputs:

- Correct simulation of Priority and Round Robin Scheduling
- Accurate Sequential & Indexed File Allocation results
- Proper memory allocation for First-fit, Best-fit, Worst-fit
- Correct MFT & MVT allocation behavior
- Screenshots verifying successful execution **Learning Outcomes:**
- Scheduling: Understood how WT, TAT, and scheduling order depend on burst time, priority, and quantum.
- File Allocation: Learned how file blocks are stored and validated by OS.
- Memory Management: Observed external & internal fragmentation across strategies.
- Fixed vs Variable Partitions: Understood differences between MFT rigid partitioning and MVT flexibility.
- Practical Understanding: Reinforced theory by simulating OS behavior.