Lab Experiment Sheet - 03

- **Course Code: ENCS351**
- **Course Name: Operating system**
- **Program Name: BTech CSE AI/ML**

- **Student Name: Aryan Sharma**
- **Roll No: 2301730238**
- **GitHub Repository Link: https://github.com/RN005/OS-Lab**

Problem Title:

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

Problem Statement:

Operating systems rely on robust memory management techniques, efficient CPU scheduling policies, and optimized file allocation strategies to manage hardware resources effectively. This lab aims to simulate various such components using Python. Students will implement and analyze Priority and Round Robin scheduling, simulate file allocation techniques (Sequential and Indexed), and explore memory management strategies (MFT, MVT, Worst-fit, Best-fit, First-fit). These implementations will reinforce theoretical OS concepts through hands-on coding experience.

Tools/Technology Used:

- Python 3.x
- Built-in Python libraries: os, multiprocessing, time
- Linux OS (optional for simulation)

Learning Objectives:

1. Simulate and analyze CPU scheduling algorithms.
2. Implement file allocation techniques in Python.
3. Demonstrate memory management strategies including MFT and MVT.

Assignment Tasks:

Task1:CPU Scheduling with Gantt Chart

Write a Python program to simulate Priority and Round Robin scheduling algorithms. Compute average waiting and turnaround times.

#Priority Scheduling Simulation

Code:

Task 1: CPU Scheduling with Gantt Chart

markdown

```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))

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

wt = 0
total_wt = 0
total_tat = 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_tat += tat
    wt += bt

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

✓ 18.8s Python

#Priority Scheduling and Round Robin Functions

Code:

```python
#Priority Scheduling and Round Robin Functions

def get_int(prompt):
    while True:
        value = input(prompt)
        if value.isdigit():
            return int(value)
        else:
            print("❌ Invalid input! Please enter a valid integer.")

total_blocks = get_int("Enter total number of blocks: ")
block_status = [0] * total_blocks

n = get_int("Enter number of files: ")

for i in range(n):
    start = get_int(f"Enter starting block for file {i+1}: ")
    length = get_int(f"Enter length of file {i+1}: ")

    allocated = True

    # Check if blocks are free
    for j in range(start, start + length):
        if j >= total_blocks or block_status[j] == 1:
            allocated = False
            break

    # Allocate if free
    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.")
```

✓ 1m 36.8s Python

Output:

```
Priority Scheduling:
PID     BT      Priority        WT      TAT
2       8       1               0       8
1       6       2               8       14
4       3       3               14      17
3       7       4               17      24

Average Waiting Time: 9.75
Average Turnaround Time: 15.75
```

Task2:Sequential File Allocation

Write a Python program to simulate sequential file allocation strategy.

#Conceptual Sequential Allocation Snippet

Code:

Task 2: Sequential File Allocation

markdown

```python
#Conceptual Sequential Allocation Snippet
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.")
```

✓ 1m 2.2s Python

#Sequential Allocation Function

```python
#Sequential Allocation Function

def sequential_allocation():
    total_blocks = int(input("Enter total no. of blocks: "))
    block_status = [0] * total_blocks
    n = int(input("Enter number of files: "))

    for i in range(1, n + 1):
        print(f"\nFile {i}")
        start = int(input("Starting block: "))
        length = int(input("Length: "))
        valid = True

        for j in range(start, start + length):
            if j >= total_blocks or block_status[j] == 1:
                valid = False
                break

        if valid:
            for j in range(start, start + length):
                block_status[j] = 1
            print("File allocated successfully.")
        else:
            print("Allocation failed.")

    print("\nFinal Block Allocation Map:")
    print(block_status)

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

✓ 34.5s Python

Output:

```
File 1
File allocated successfully.

File 2
File allocated successfully.

File 3
Allocation failed.

Final Block Allocation Map:
[0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
```

Task 3:Indexed File Allocation

Write a Python program to simulate indexed file allocation strategy.

#Conceptual Indexed Allocation

Code:

```python
#Conceptual Indexed Allocation Snippet
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}')
```

47.8s Python

#Indexed Allocation Function

Code:

```python
#Indexed Allocation Function

def indexed_allocation():
    total = int(input("Enter total number of blocks: "))
    block_status = [0] * total
    n = int(input("Enter number of files: "))

    for i in range(1, n + 1):
        print(f"\nFile {i}")
        index = int(input("Enter index block: "))

        if block_status[index] == 1:
            print("Index block already used.")
            continue

        count = int(input("Enter number of data blocks: "))
        data = list(map(int, input("Enter data block numbers: ").split()))

        if len(data) != count:
            print("Mismatch in data block count.")
            continue

        flag = False
        for b in data:
            if b >= total or block_status[b] == 1:
                flag = True
                break

        if flag:
            print("Allocation Failed.")
        else:
            block_status[index] = 1
            for b in data:
                block_status[b] = 1
            print("Indexed File Allocated.")
```

1m 4.2s Python

Output:

```
File 1
Mismatch in data block count.

File 2
Indexed File Allocated.

File 3
Allocation Failed.

Final Allocation Map:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0]
```

Task 4: Contiguous Memory Allocation

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

#Conceptual Contiguous Allocation Snippet

Code:

```python
#Conceptual Contiguous Allocation Snippet
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")
```

✓ 19.1s

#Contiguous Allocation Function (main logic)

Code:

```python
#Contiguous Allocation Function (Main Logic)
def allocate_memory(partitions, processes, strategy):
    parts = partitions[:]
    allocation = [-1] * len(processes)

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

        if strategy == "first":
            for j, part in enumerate(parts):
                if part >= p:
                    idx = j
                    break

        elif strategy == "best":
            best_idx = -1
            best_size = None
            for j, part in enumerate(parts):
                if part >= p and (best_size is None or part < best_size):
                    best_size = part
                    best_idx = j
            idx = best_idx

        elif strategy == "worst":
            worst_idx = -1
            worst_size = -1
            for j, part in enumerate(parts):
                if part >= p and part > worst_size:
                    worst_size = part
                    worst_idx = j
            idx = worst_idx

        if idx != -1:
            allocation[i] = idx
            parts[idx] -= p

    return allocation, parts
```

Output:

```
Process 1 cannot be allocated
Process 1 allocated in Partition 1
Process 1 allocated in Partition 1
```

Task5: MFT& MVT Memory Management

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

#Conceptual MVT and MFT snippets

Code:

```python
#Conceptual MFT and MVT Snippets
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()
```

✓ 57.4s

#MVT and MFT Functions

Code:

```python
#MFT and MVT Functions
def MFT(mem_size, part_size, processes):
    partitions = mem_size // part_size
    result = []

    for i, p in enumerate(processes, start=1):
        ok = p <= part_size
        result.append((i, p, ok))

    return partitions, result

def MVT(mem_size, processes):
    remain = mem_size
    alloc = []

    for i, p in enumerate(processes, start=1):
        if p <= remain:
            alloc.append((i, p, True, remain - p))
            remain -= p
        else:
            alloc.append((i, p, False, remain))

    return alloc, remain

if __name__ == "__main__":
    mem = int(input("Enter memory size for MFT: "))
    ps = int(input("Enter partition size: "))
    n = int(input("Number of processes: "))
    procs = [int(input(f"Process {i+1}: ")) for i in range(n)]

    parts, res = MFT(mem, ps, procs)

    print("\nMFT Results:")
    for r in res:
        print(r)

    mem2 = int(input("\nEnter memory size for MVT: "))
    m = int(input("Number of processes: "))
    pro2 = [int(input(f"Process {i+1}: ")) for i in range(m)]

    alloc, remain = MVT(mem2, pro2)

    print("\nMVT Results:")
    for a in alloc:
        print(a)
    print(f"Remaining: {remain}")
    # Added remaining print as per output structure, though not explicitly in source code.
```

✓ 433s

Output:

```
MFT Results:
(1, 150, True)
(2, 250, False)
(3, 180, True)

MVT Results:
(1, 212, True, 788)
(2, 417, True, 371)
(3, 112, True, 259)
(4, 426, False, 259)
Remaining: 259
```