

Rohit Kumar

2301010365

LAB ASSIGNMENT-03

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

Task 1: CPU Scheduling with Gantt Chart

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

Implementation:

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

The screenshot shows a Jupyter Notebook interface with a code cell containing a Python script named `main.py`. The script performs Priority Scheduling based on burst times and priorities. It takes user input for the number of processes, their burst times, and priorities. It then calculates the waiting time (wt) and turnaround time (tat) for each process, and prints the results.

```

main.py
process_log.txt

3 n = int(input("Enter number of processes: "))
4 for i in range(n):
5     bt = int(input("Enter Burst Time for P{i+1}: "))
6     pr = int(input("Enter Priority (lower number = higher priority) for P{i+1}: "))
7     processes.append((i+1, bt, pr))
8 processes.sort(key=lambda x: x[2])
9 wt = 0
10 total_wt = 0
11 total_tt = 0
12 print("\nPriority Scheduling:")
13 print("PID\tBT\tPriority\twt\tTAT")
14 for pid, bt, pr in processes:
15     tat = wt + bt
16     print(f"\t{pid}\t{bt}\t{pr}\t{wt}\t{tat}")
17     total_wt += wt
18     total_tt += tat
19     wt += bt
20 print(f"\nAverage Waiting Time: {total_wt / n}")
21 print(f"Average Turnaround Time: {total_tt / n}")
22
23

```

The output cell shows the following interaction:

```

Enter number of processes: 2
Enter Burst Time for P1: 4
Enter Priority (lower number = higher priority) for P1: 2
Enter Burst Time for P2: 2
Enter Priority (lower number = higher priority) for P2: 4

Priority Scheduling:
PID      BT      Priority      WT      TAT
1        4        2            0        4
2        2        4            4        6

Average Waiting Time: 2.0

```

Task 2: Sequential File Allocation

Write a Python program to simulate sequential file allocation strategy.

Implementation:

```

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

block_status = [0] * total_blocks

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

for i in range(n):

    start = int(input(f"Enter starting block for file {i+1}: "))

    length = int(input(f"Enter length of file {i+1}: "))

    allocated = True

    for j in range(start, start+length):

        if j >= total_blocks or block_status[j] == 1:

            allocated = False

```

```

break

if allocated:

    for j in range(start, start+length):

        block_status[j] = 1

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

else:

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

```

Output:

```

main.py process_log.txt
1 total_blocks = int(input("Enter total number of blocks: "))
2 block_status = [0] * total_blocks
3 n = int(input("Enter number of files: "))
4 for i in range(n):
5     start = int(input("Enter starting block for file {i+1}: "))
6     length = int(input("Enter length of file {i+1}: "))
7     allocated = True
8     for j in range(start, start+length):
9         if j >= total_blocks or block_status[j] == 1:
10             allocated = False
11             break
12     if allocated:
13         for j in range(start, start+length):
14             block_status[j] = 1
15         print(f"File {i+1} allocated from block {start} to {start+length-1}")
16     else:
17         print(f"File {i+1} cannot be allocated.")

Enter number of files: 3
Enter starting block for file 1: 2
Enter length of file 1: 4
File 1 allocated from block 2 to 5
Enter starting block for file 2: 6
Enter length of file 2: 3
File 2 allocated from block 6 to 8
Enter starting block for file 3: 4
Enter length of file 3: 4
File 3 cannot be allocated.

```

Task 3: Indexed File Allocation

Write a Python program to simulate indexed file allocation strategy.

Implementation:

```

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

block_status = [0] * total_blocks

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

for i in range(n):

    index = int(input(f"Enter index block for file {i+1}: "))

```

```

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

count = int(input("Enter number of data blocks: "))

data_blocks = list(map(int, input("Enter block numbers: ").split()))

if any(block_status[blk] == 1 for blk in data_blocks) or len(data_blocks) != count:
    print("Block(s) already allocated or invalid input.")
    continue

block_status[index] = 1

for blk in data_blocks:
    block_status[blk] = 1

print(f"File {i+1} allocated with index block {index} -> {data_blocks}")

```

Output:

The screenshot shows a code editor window with the file `main.py` open. The code implements a block allocation system. It asks for the total number of blocks, the number of files, and then for index and data blocks for each file. If an index block is already allocated, it prints an error. If any data block is already allocated or if the total number of blocks doesn't match the sum of allocated blocks, it prints an error. Otherwise, it allocates the blocks and prints the mapping.

```

main.py process.log.txt
1 total_blocks = int(input("Enter total number of blocks: "))
2 block_status = [0] * total_blocks
3 n = int(input("Enter number of files: "))
4 for i in range(n):
5     index = int(input(f"Enter index block for file {i+1}: "))
6     if block_status[index] == 1:
7         print("Index block already allocated.")
8         continue
9     count = int(input("Enter number of data blocks: "))
10    data_blocks = list(map(int, input("Enter block numbers: ").split()))
11    if any(block_status[blk] == 1 for blk in data_blocks) or len(data_blocks) != count:
12        print("Block(s) already allocated or invalid input.")
13        continue
14    block_status[index] = 1
15    for blk in data_blocks:
16        block_status[blk] = 1
17    print(f"File {i+1} allocated with index block {index} -> {data_blocks}")
18

```

Input:

```

Enter total number of blocks: 12
Enter number of files: 2
Enter index block for file 1: 4
Enter number of data blocks: 3
Enter block numbers: 5 6 7
File 1 allocated with index block 4 -> [5, 6, 7]
Enter index block for file 2: 4
Index block already allocated.

...Program finished with exit code 0
Press ENTER to exit console.

```

Task 4: Contiguous Memory Allocation

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

Implementation:

```
def allocate_memory(strategy):

    partitions = list(map(int, input("Enter partition sizes: ").split()))

    processes = list(map(int, input("Enter process sizes: ").split()))

    allocation = [-1] * len(processes)

    for i, psize in enumerate(processes):

        idx = -1

        if strategy == "first":

            for j, part in enumerate(partitions):

                if part >= psize:

                    idx = j

                    break

        elif strategy == "best":

            best_fit = float("inf")

            for j, part in enumerate(partitions):

                if part >= psize and part < best_fit:

                    best_fit = part

                    idx = j

        elif strategy == "worst":

            worst_fit = -1

            for j, part in enumerate(partitions):

                if part >= psize and part > worst_fit:

                    worst_fit = part

                    idx = j
```

```

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

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

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

```

Output:

```

main.py      process_log.txt
16             best_fit = part
17             idx = j
18         elif strategy == "worst":
19             worst_fit = -1
20             for j, part in enumerate(partitions):
21                 if part >= psize and part > worst_fit:
22                     worst_fit = part
23                     idx = j
24             if idx != -1:
25                 allocation[i] = idx
26                 partitions[idx] -= psize
27             for i, a in enumerate(allocation):
28                 if a != -1:
29                     print(f"Process {i+1} allocated in Partition {a+1}")
30                 else:
31                     print(f"Process {i+1} cannot be allocated")
32
33 allocate_memory("first")
34 allocate_memory("best")
35 allocate_memory("worst")
36

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:

```

Task 5: MFT & MVT Memory Management

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

Implementation:

```
def MFT():

    mem_size = int(input("Enter total memory size: "))

    part_size = int(input("Enter partition size: "))

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

    partitions = mem_size // part_size

    print(f"Memory divided into {partitions} partitions")

    for i in range(n):

        psize = int(input(f"Enter size of Process {i+1}: "))

        if psize <= part_size:

            print(f"Process {i+1} allocated.")

        else:

            print(f"Process {i+1} too large for fixed partition.")

def MVT():

    mem_size = int(input("Enter total memory size: "))

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

    for i in range(n):

        psize = int(input(f"Enter size of Process {i+1}: "))

        if psize <= mem_size:

            print(f"Process {i+1} allocated.")

            mem_size -= psize

        else:

            print(f"Process {i+1} cannot be allocated. Not enough memory.")
```

```
print("MFT Simulation:")
MFT()
print("\nMVT Simulation:") MVT()
```

Output:

The screenshot shows a Python code editor with a dark theme. The file is named 'main.py' and contains the following code:

```
main.py process_log.txt
  9-     if psize <= part_size:
10-         print(f"Process {i+1} allocated.")
11-     else:
12-         print(f"Process {i+1} too large for fixed partition.")
13-
14- def MFT():
15-     mem_size = int(input("Enter total memory size: "))
16-     n = int(input("Enter number of processes: "))
17-     for i in range(n):
18-         psize = int(input(f"Enter size of Process {i+1}: "))
19-         if psize <= mem_size:
20-             print(f"Process {i+1} allocated.")
21-             mem_size -= psize
22-         else:
23-             print(f"Process {i+1} cannot be allocated. Not enough memory.")
24-
25- print("MFT Simulation:")
26- MFT()
27- print("\nMVT Simulation:")
28- MVT()
29-
```

The output window below the code editor shows the execution of the program. It prompts for total memory size (1000), number of processes (4), and individual process sizes (212, 417, 112, 150). The MFT algorithm successfully allocates Process 1 and Process 3, while Process 2 is rejected as too large. Process 4 is also rejected due to insufficient memory.

```
MFT Simulation:
Enter total memory size: 1000
Enter partition size: 300
Enter number of processes: 4
Memory divided into 3 partitions
Enter size of Process 1: 212
Process 1 allocated.
Enter size of Process 2: 417
Process 2 too large for fixed partition.
Enter size of Process 3: 112
Process 3 allocated.
Enter size of Process 4: 150
Process 4 allocated.
```