Batch: E-2 **Roll No.: 16010123325**

Experiment No. 10

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE: Implementation of Memory Allocation Algorithms-BF,WF,FF

AIM: Implementation of Basic CPU Scheduling Algorithms – Non Preemptive [FCFS , SJF]

Expected Outcome of Experiment:

CO5 Understand Storage management with allocation

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**

2. **Achyut S. Godbole , Atul Kahate "Operating Systems" McGraw Hill Third Edition.**

3. **William Stallings, "Operating System Internal & Design Principles", Pearson.**

4. **Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.**

Pre Lab/ Prior Concepts:

Memory is central to the operation of computing systems.

Memory = a large array of words/bytes.

Each byte or word has its own address.

Memory contains the program to be executed and data, both.

Program is executed line by line with Instruction Fetch, Instruction Decode, Operand Fetch, Execute cycles.

Program counter contains the address of the memory location to be executed next.

Memory consists of a large array of words or bytes, each with its own address.

The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example,

first fetches an instruction from memory.

The instruction is then decoded and may cause operands to be fetched from memory.

After the instruction has been executed on the operands, results may be stored back in memory.

Memory unit only sees a stream of addresses + read requests, or address + data and write requests

Description of the application to be implemented:

First Fit:

- Iterate through the list of memory partitions.
- Allocate the process to the first partition that has sufficient space.
- Update the remaining size of the partition.
- If no partition is large enough, the process remains unallocated.

Best Fit :

- Find the smallest partition that can accommodate the process.
- Allocate the process to this partition.
- Update the remaining partition size.
- If no suitable partition is found, the process remains unallocated.

Worst Fit :

- Find the largest available partition.
- Allocate the process to this partition.
- Update the remaining partition size.
- If no partition is large enough, the process remains unallocated.

Implementation details:

First Fit:

```
#include <stdio.h>

void implimentFirstFit(int blockSize[], int blocks, int processSize[], int
processes)
{

    int allocate[processes];
    int occupied[blocks];


    for(int i = 0; i < processes; i++)
    {
        allocate[i] = -1;
    }

    for(int i = 0; i < blocks; i++){
        occupied[i] = 0;
    }


    for (int i = 0; i < processes; i++)
    {
        for (int j = 0; j < blocks; j++)
        {
        if (!occupied[j] && blockSize[j] >= processSize[i])
            {

                allocate[i] = j;
                occupied[j] = 1;

                break;
            }
        }
    }
```

```
    printf("\nProcess No.\tProcess Size\tBlock no.\n"); for (int
    i = 0; i < processes; i++)
    {
        printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]); if
        (allocate[i] != -1)
            printf("%d\n",allocate[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main()
{
    int blockSize[] = {30, 5, 10};
    int processSize[] = {10, 6, 9};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);

    implimentFirstFit(blockSize, m, processSize, n);
}
```

Output-

```
Process No.     Process Size    Block no.
1               10              1
2               6               3
3               9               Not Allocated
```

Best Fit-

```
#include <stdio.h>

void bestFit(int block[], int m, int process[], int n) {
    int allocation[n];

    for (int i = 0; i < n; i++)
        allocation[i] = -1;
    for (int i = 0; i < n; i++) {
```

```
        int bestIdx = -1;

        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i]) {
                if (bestIdx == -1 || block[j] < block[bestIdx])
                    bestIdx = j;
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            block[bestIdx] -= process[i];
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main() {
    int block[] = {100, 500, 200, 300, 600};
    int process[] = {212, 417, 112, 426};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);

    bestFit(block, m, process, n);

    return 0;
}
```

Output-

```
Process No.     Process Size    Block No.
1               212             4
2               417             2
3               112             3
4               426             5
```

Worst Fit-

```c
#include <stdio.h>

void worstFit(int blockSize[], int numBlocks, int processSize[], int
numProcesses) {
    int allocation[numProcesses];


    for (int i = 0; i < numProcesses; i++)
        allocation[i] = -1;


    for (int i = 0; i < numProcesses; i++) {
        int worstIdx = -1;

        for (int j = 0; j < numBlocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx]) {
                    worstIdx = j;
                }
            }
        }


        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < numProcesses; i++) {
        printf(" %d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
```

```
        else
            printf("Not Allocated\n");
    }
}

int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int numBlocks = sizeof(blockSize) / sizeof(blockSize[0]);
    int numProcesses = sizeof(processSize) / sizeof(processSize[0]);

    worstFit(blockSize, numBlocks, processSize, numProcesses);

    return 0;
}
```

Output-

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

Conclusion :

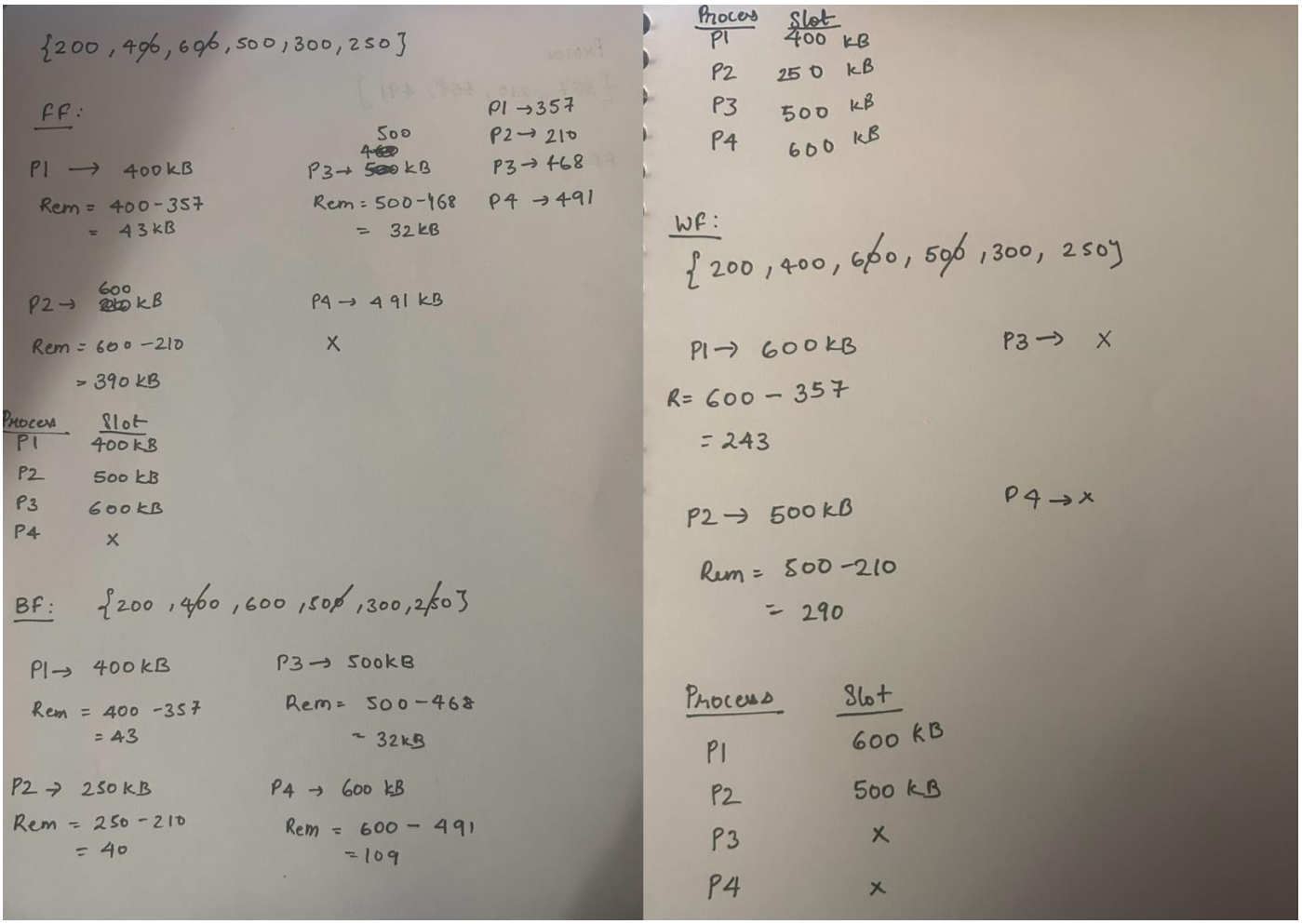
The above experiment highlights memory allocation with fixed partitions using First, Best and Worse algorithms.

Post Lab Descriptive Questions

A. Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB and 250 KB. These partitions need to be allocated to four processes of sizes 357 KB, 210 KB, 468 KB and 491 KB in that order.

Perform the allocation of processes using- First Fit Algorithm, Best Fit Algorithm,

Worst Fit Algorithm



B. Explain Buffering and its types in detail.

Buffering is a temporary storage mechanism that helps manage data transfer between devices/processes with different speeds. It improves efficiency and prevents delays.

Types of Buffering

1. Single Buffering

- Uses one buffer to store data temporarily.
- The process waits while data is being transferred.
- Simple but inefficient for high-speed devices.

2. Double Buffering

- Uses two buffers alternately—one for processing, one for loading.
- Reduces idle time and improves performance
- Faster than single buffering but needs extra memory.

3. Circular (Multi) Buffering

- Uses multiple buffers in a circular queue.
- Maximizes efficiency in real-time and high-speed applications.
- Prevents data loss and improves throughput.

Date: _____ **Signature of faculty in-charge**