Batch: A1	**Roll No.: 16010123012**
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 is 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:

The First Fit memory allocation algorithm allocates the first available memory block that is large enough to accommodate the process. It scans the list of memory blocks from the beginning and assigns the first block that meets the size requirement of the process. This algorithm is simple and fast but may lead to inefficient memory utilization over time, as it does not consider the optimal block size for allocation.

Best Fit:

The Best Fit memory allocation algorithm searches for the smallest memory block that is large enough to accommodate the process. It scans the entire list of memory blocks to find the block that best fits the process size. This approach minimizes wasted memory but can be slower due to the need to search the entire list. It may also lead to fragmentation, as small leftover blocks may not be usable for future processes.

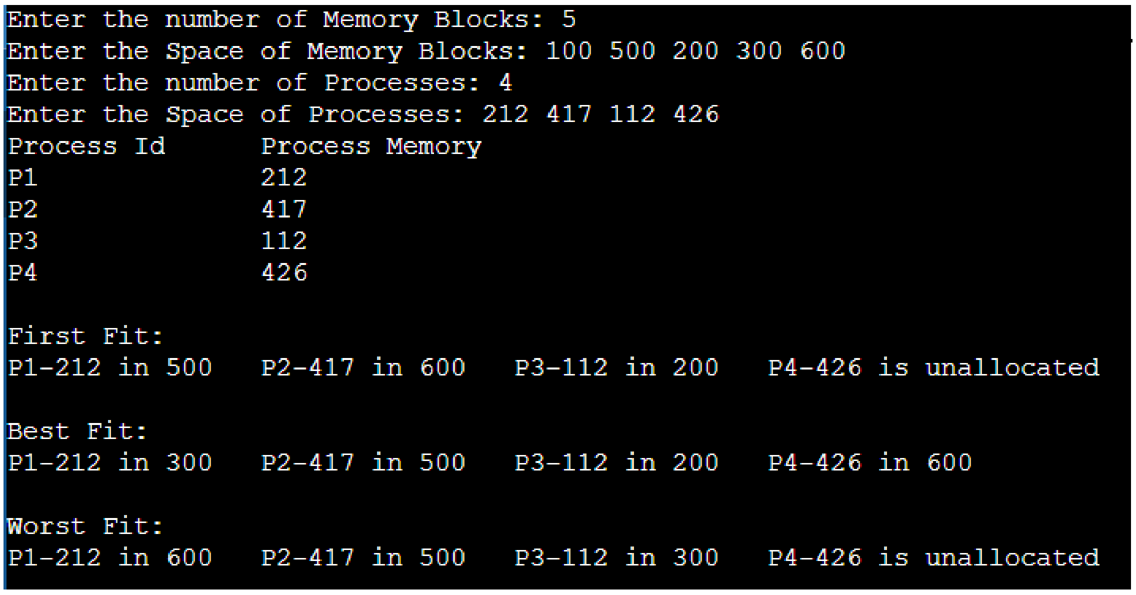
Worst Fit:

The Worst Fit memory allocation algorithm allocates the largest available memory block to the process. It scans the list of memory blocks and assigns the largest block, regardless of whether it is the best fit. This approach aims to reduce fragmentation by leaving larger leftover blocks, but it can lead to inefficient memory utilization and may not be suitable for systems with varying process sizes.

Implementation details:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
   int n;
   cout << "Enter the number of Memory Blocks: " ;
   cin >> n;
   cout << "Enter the Space of Memory Blocks: " ;
   vector < int > memoryBlock(n);
   for (int i = 0; i < n; i++) {
       cin >> memoryBlock[i];
   }
   cout << "Enter the number of Processes: " ;
   int m;
   cin >> m;
   cout << "Enter the Space of Processes: " ;
   vector < int > processMemory(n);
   for (int i = 0; i < m; i++) {
       cin >> processMemory[i];
   }
   cout << "Process Id\t" << "Process Memory\n";
   for (int i = 0; i < m; i++) {
       cout << "P" << i + 1 << "\t\t" << processMemory[i] << endl;
   }
   vector < int > sortedMemoryBlock(memoryBlock);
   sort(sortedMemoryBlock.begin(), sortedMemoryBlock.end());
   vector < int > inverseSortedMemoryBlock(sortedMemoryBlock);
   reverse(inverseSortedMemoryBlock.begin(),
inverseSortedMemoryBlock.end());
   cout << "\nFirst Fit: \n";
   for (int i = 0; i < m; i++) {
       for (int j = 0; j < n; j++) {
           if (processMemory[i] <= memoryBlock[j] && memoryBlock[j] != -
1) {
               cout << "P" << i + 1 << "-" << processMemory[i] << " in "
<< memoryBlock[j] << "\t";
               memoryBlock[j] = -1;
               break;
           }
           if (j == n-1) {
               cout << "P" << i + 1 << "-" << processMemory[i] << " is
unallocated\t";
```

```
            }
        }
    }
    cout << "\n\nBest Fit: \n";
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (processMemory[i] <= sortedMemoryBlock[j] &&
sortedMemoryBlock[j] != -1) {
                cout << "P" << i + 1 << "-" << processMemory[i] << " in "
<< sortedMemoryBlock[j] << "\t";
                sortedMemoryBlock[j] = -1;
                break;
            }
            if (j == n - 1) {
                cout << "P" << i + 1 << "-" << processMemory[i] << " is
unallocated\t";
            }
        }
    }
    cout << "\n\nWorst Fit: \n";
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (processMemory[i] <= inverseSortedMemoryBlock[j] &&
inverseSortedMemoryBlock[j] != -1) {
                cout << "P" << i + 1 << "-" << processMemory[i] << " in "
<< inverseSortedMemoryBlock[j] << "\t";
                inverseSortedMemoryBlock[j] = -1;
                break;
            }
            if (j == n - 1) {
                cout << "P" << i + 1 << "-" << processMemory[i] << " is
unallocated\t";
            }
        }
    }

    return 0;
}
```



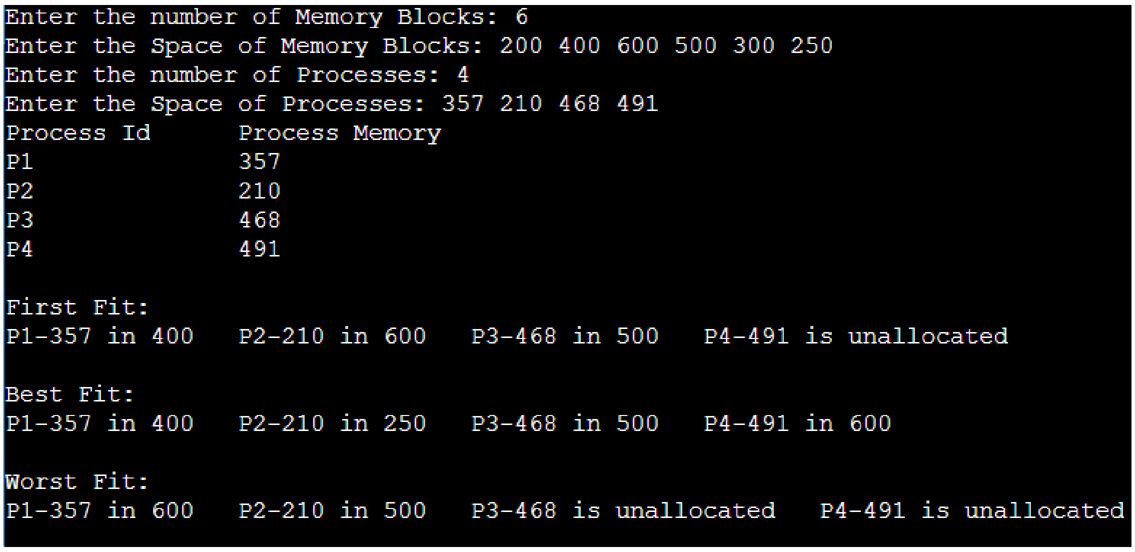
```
Enter the number of Memory Blocks: 5
Enter the Space of Memory Blocks: 100 500 200 300 600
Enter the number of Processes: 4
Enter the Space of Processes: 212 417 112 426
Process Id      Process Memory
P1              212
P2              417
P3              112
P4              426

First Fit:
P1-212 in 500   P2-417 in 600   P3-112 in 200   P4-426 is unallocated

Best Fit:
P1-212 in 300   P2-417 in 500   P3-112 in 200   P4-426 in 600

Worst Fit:
P1-212 in 600   P2-417 in 500   P3-112 in 300   P4-426 is unallocated
```

Conclusion:

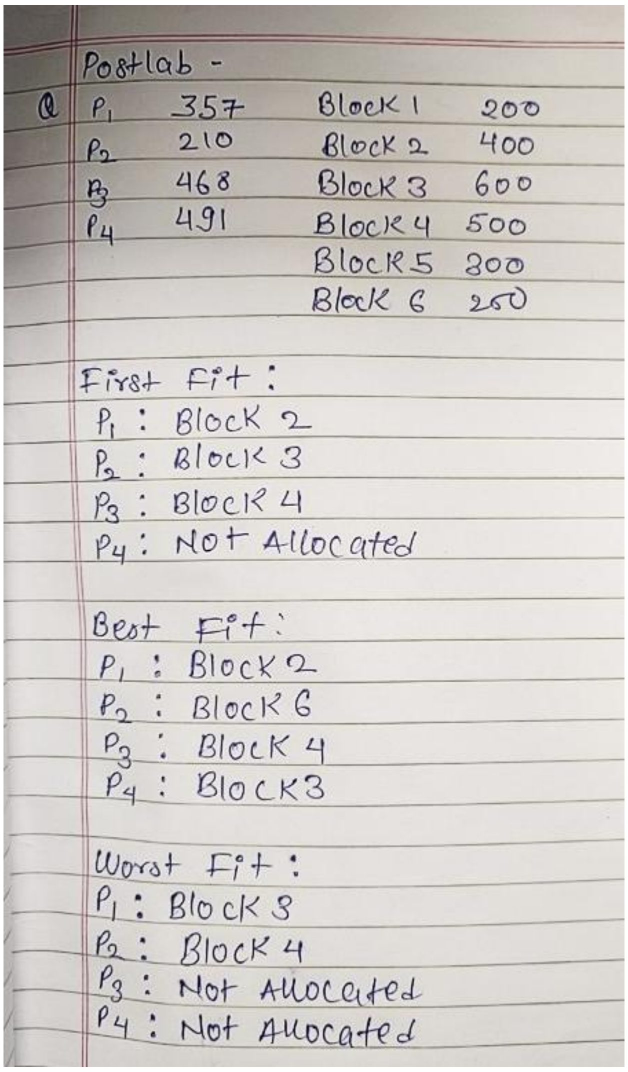
I have successfully implemented the memory allocation strategies, First Fit, Best Fit, and Worst Fit. The experiment demonstrated how different allocation methods impact memory utilization and process placement efficiency. First Fit allocates the first available suitable block, Best Fit minimizes internal fragmentation by using the smallest adequate block and Worst Fit utilizes the largest available block to leave the biggest remaining space. Understanding these strategies helps optimize storage management in operating systems, ensuring efficient resource allocation and reduced memory wastage.

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



```
Enter the number of Memory Blocks: 6
Enter the Space of Memory Blocks: 200 400 600 500 300 250
Enter the number of Processes: 4
Enter the Space of Processes: 357 210 468 491
Process Id      Process Memory
P1              357
P2              210
P3              468
P4              491

First Fit:
P1-357 in 400   P2-210 in 600   P3-468 in 500   P4-491 is unallocated

Best Fit:
P1-357 in 400   P2-210 in 250   P3-468 in 500   P4-491 in 600

Worst Fit:
P1-357 in 600   P2-210 in 500   P3-468 is unallocated   P4-491 is unallocated
```



Postlab -

Q

P_1	357	Block 1	200
P_2	210	Block 2	400
P_3	468	Block 3	600
P_4	491	Block 4	500
		Block 5	300
		Block 6	250

First Fit :
P_1 : Block 2
P_2 : Block 3
P_3 : Block 4
P_4 : Not Allocated

Best Fit :
P_1 : Block 2
P_2 : Block 6
P_3 : Block 4
P_4 : Block 3

Worst Fit :
P_1 : Block 3
P_2 : Block 4
P_3 : Not Allocated
P_4 : Not Allocated

B. Explain Buffering and its types in detail.

Buffering is a technique used in operating systems to temporarily store data in a memory area called a buffer while transferring it between processes or devices. It helps manage the speed differences between the sender and receiver, ensuring smooth data flow and improving system performance

Types of Buffering:

1. **Single Buffering:**
   A single buffer is used to hold data temporarily. The producer writes data into the buffer, and the consumer reads from it. Simple but can lead to inefficiencies if the producer and consumer operate at different speeds.

2. **Double Buffering:**
   Two buffers are used alternately. While one buffer is being filled by the producer, the other is being emptied by the consumer. Improves efficiency by allowing parallel processing.

3. **Circular Buffering:**
   A fixed-size buffer is used in a circular manner. When the buffer is full, new data overwrites the oldest data. Commonly used in streaming applications.

4. **Cache Buffering:**
   A high-speed buffer (cache) is used to store frequently accessed data. Reduces access time by keeping data closer to the processor.

5. **Spooling (Simultaneous Peripheral Operations On-Line):**
   A specialized form of buffering used for I/O operations. Data is temporarily stored in a spool (buffer) before being sent to a peripheral device (e.g., printer).

Date: 25 / 03 / 2025 **Signature of faculty in-charge**