EXPERIMENT-1

Aim:

Write a programme to implement CPU scheduling for the **First Come First Serve**.

Theory:

Code:

```
#include <iostream>
#include <vector>

using namespace std;

struct Process{
   int id;
   int burstTime;
   double waitingTime;
   double turnaroundTime;
};

void fcfsScheduling(vector<Process> &processes){
   int n = processes.size();
   int totalTime = 0;
   double totalWaitingTime = 0;
   double totalTurnaroundTime = 0;

   cout << "Process"<< "\t" << "Burst Time" << "\t" << "Waiting Time" << "\t" << "Turnaround
Time" << endl;

   for (int i = 0; i < n; i++){
      totalTime += processes[i].burstTime;
      processes[i].turnaroundTime = totalTime;
      processes[i].waitingTime = processes[i].turnaroundTime - processes[i].burstTime;

      cout << "P" << processes[i].id << "\t\t" << processes[i].burstTime << "\t\t\t" <<
processes[i].waitingTime << "\t\t\t\t" << processes[i].turnaroundTime << endl;

      totalWaitingTime += processes[i].waitingTime;
      totalTurnaroundTime += processes[i].turnaroundTime;
   }

   double averageWaitingTime = totalWaitingTime / n;
   double averageTurnaroundTime = totalTurnaroundTime / n;

   cout << "\nAverage Waiting Time: " << averageWaitingTime << endl;
   cout << "Average Turnaround Time: " << averageTurnaroundTime << endl;
}

int main(){
   int n;
   cout << "Enter the number of processes: ";
   cin >> n;
```

```
    vector<Process> processes(n);

    for (int i = 0; i < n; i++){
        processes[i].id = i + 1;
        cout << "Enter burst time for Process P" << processes[i].id << ": ";
        cin >> processes[i].burstTime;
    }

    fcfsScheduling(processes);

    return 0;
}
```

Output:

```
Enter the number of processes: 5
Enter burst time for Process P1: 3
Enter burst time for Process P2: 5
Enter burst time for Process P3: 2
Enter burst time for Process P4: 7
Enter burst time for Process P5: 4
Process Burst Time  Waiting Time    Turnaround Time
P1      3           0               3
P2      5           3               8
P3      2           8               10
P4      7           10              17
P5      4           17              21

Average Waiting Time: 7.6
Average Turnaround Time: 11.8
```

EXPERIMENT-2

Aim:

Write a programme to implement CPU scheduling for **Shortest Job First** (SJF) and **Shortest Remaining Time First** (SRTF).

Theory:

Code:

```
#include <bits/stdc++.h>

using namespace std;

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
};

void sjf(vector<Process>& processes) {
    int n = processes.size();
    int current_time = 0;
    int completed = 0;
    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    while (completed < n) {
        int shortest_job = -1;
        int min_burst_time = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time &&
processes[i].remaining_time < min_burst_time && processes[i].remaining_time >
0) {
                shortest_job = i;
                min_burst_time = processes[i].remaining_time;
            }
        }

        if (shortest_job == -1) {
            current_time++;
        }
        else {
            processes[shortest_job].remaining_time--;
            current_time++;

            if (processes[shortest_job].remaining_time == 0) {
                completed++;
                processes[shortest_job].turnaround_time = current_time -
processes[shortest_job].arrival_time;
                processes[shortest_job].waiting_time =
processes[shortest_job].turnaround_time - processes[shortest_job].burst_time;
```

```
                total_turnaround_time +=
processes[shortest_job].turnaround_time;
                total_waiting_time += processes[shortest_job].waiting_time;
            }
        }
    }

    cout << "Process\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time"
<< endl;
    for (const Process& p : processes) {
        cout << p.id << "\t" << p.arrival_time << "\t\t" << p.burst_time <<
"\t\t" << p.turnaround_time << "\t\t" << p.waiting_time << endl;
    }

    double avg_turnaround_time = total_turnaround_time / n;
    double avg_waiting_time = total_waiting_time / n;
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
    cout << "Average Waiting Time: " << avg_waiting_time << endl;
}

// Preemptive SJF
void srtf(vector<Process>& processes) {
    int n = processes.size();
    int current_time = 0;
    int completed = 0;
    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    while (completed < n) {
        int shortest_job = -1;
        int min_remaining_time = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time &&
processes[i].remaining_time < min_remaining_time &&
processes[i].remaining_time > 0) {
                shortest_job = i;
                min_remaining_time = processes[i].remaining_time;
            }
        }

        if (shortest_job == -1) {
            current_time++;
        }
        else {
            processes[shortest_job].remaining_time--;
            current_time++;
```

```
            if (processes[shortest_job].remaining_time == 0) {
                completed++;
                processes[shortest_job].turnaround_time = current_time -
processes[shortest_job].arrival_time;
                processes[shortest_job].waiting_time =
processes[shortest_job].turnaround_time - processes[shortest_job].burst_time;
                total_turnaround_time +=
processes[shortest_job].turnaround_time;
                total_waiting_time += processes[shortest_job].waiting_time;
            }
        }
    }

    cout << "Process\tArrival Time\tBurst Time\tTurnaround Time\tWaiting Time"
<< endl;
    for (const Process& p : processes) {
        cout << p.id << "\t" << p.arrival_time << "\t\t" << p.burst_time <<
"\t\t" << p.turnaround_time << "\t\t" << p.waiting_time << endl;
    }

    double avg_turnaround_time = total_turnaround_time / n;
    double avg_waiting_time = total_waiting_time / n;
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
    cout << "Average Waiting Time: " << avg_waiting_time << endl;
}

int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    vector<Process> processes(n);

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        cout << "Enter arrival time for Process " << i + 1 << ": ";
        cin >> processes[i].arrival_time;
        cout << "Enter burst time for Process " << i + 1 << ": ";
        cin >> processes[i].burst_time;
        processes[i].remaining_time = processes[i].burst_time;
    }

    sort(processes.begin(), processes.end(), [](const Process& a, const
Process& b) {
        return a.arrival_time < b.arrival_time;
    });

    cout << "\nSJF Scheduling:\n";
```

```
    sjf(processes);

    cout << "\nSRTF Scheduling:\n";
    srtf(processes);

    return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter arrival time for Process 1: 0
Enter burst time for Process 1: 3
Enter arrival time for Process 2: 1
Enter burst time for Process 2: 2
Enter arrival time for Process 3: 2
Enter burst time for Process 3: 4
Enter arrival time for Process 4: 3
Enter burst time for Process 4: 1
SJF Scheduling:
Process Arrival Time    Burst Time  Turnaround Time Waiting Time
1       0               3           3               0
2       1               2           5               3
3       2               4           8               4
4       3               1           1               0
Average Turnaround Time: 4.25
Average Waiting Time: 1.75

SRTF Scheduling:
Process Arrival Time    Burst Time  Turnaround Time Waiting Time
1       0               3           3               0
2       1               2           5               3
3       2               4           8               4
4       3               1           1               0
Average Turnaround Time: 4.25
Average Waiting Time: 1.75
```

EXPERIMENT-3

Aim:

Write a program to perform priority scheduling.

Theory:

Code:

```
#include <bits/stdc++.h>

using namespace std;

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int priority;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

bool comparePriority(const Process& p1, const Process& p2) {
    return p1.priority < p2.priority;
}

void nonPreemptivePriorityScheduling(vector<Process>& processes) {
    int n = processes.size();

    sort(processes.begin(), processes.end(), comparePriority);

    int current_time = 0;
    for (int i = 0; i < n; i++) {
        processes[i].completion_time = current_time + processes[i].burst_time;
        processes[i].turnaround_time = processes[i].completion_time -
processes[i].arrival_time;
        processes[i].waiting_time = max(0, processes[i].turnaround_time -
processes[i].burst_time);

        current_time = processes[i].completion_time;
    }
}

void preemptivePriorityScheduling(vector<Process>& processes) {
    int n = processes.size();

    int current_time = 0;
    vector<bool> completed(n, false);

    // Creating a copy of burst time for each process
    vector<int> burst_times(n);
    for (int i = 0; i < n; i++) {
        burst_times[i] = processes[i].burst_time;
    }
```

```
    while (true) {
        int highest_priority = INT_MAX;
        int selected_process = -1;

        for (int i = 0; i < n; i++) {
            if (!completed[i] && processes[i].arrival_time <= current_time &&
processes[i].priority < highest_priority) {
                highest_priority = processes[i].priority;
                selected_process = i;
            }
        }

        if (selected_process == -1) {
            break;
        }

        processes[selected_process].completion_time = current_time + 1;
        burst_times[selected_process]--;

        if (burst_times[selected_process] == 0) {
            completed[selected_process] = true;
            processes[selected_process].turnaround_time =
processes[selected_process].completion_time -
processes[selected_process].arrival_time;
            processes[selected_process].waiting_time =
processes[selected_process].turnaround_time -
processes[selected_process].burst_time;
        }

        current_time++;
    }
}

void displayResult(const vector<Process>& processes) {
    cout << setw(10) << "Process" << setw(15) << "Arrival Time" << setw(15) <<
"Burst Time" << setw(15) << "Priority" << setw(15) << "Completion Time" <<
setw(15) << "Turnaround Time" << setw(15) << "Waiting Time" << endl;

    double total_waiting_time = 0;
    double total_turnaround_time = 0;

    for (const Process& p : processes) {
        cout << setw(10) << "P" << p.id << setw(15) << p.arrival_time <<
setw(15) << p.burst_time << setw(15) << p.priority << setw(15) <<
p.completion_time << setw(15) << p.turnaround_time << setw(15) <<
p.waiting_time << endl;

        total_waiting_time += p.waiting_time;
```

```
        total_turnaround_time += p.turnaround_time;
    }

    double avg_turnaround_time = total_turnaround_time / processes.size();
    double avg_waiting_time = total_waiting_time / processes.size();
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
    cout << "Average Waiting Time: " << avg_waiting_time << endl;
}

int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    vector<Process> processes(n);

    cout << "Enter the arrival time, burst time, and priority for each
process:" << endl;
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        cout << "Process " << i + 1 << ":" << endl;
        cout << "Arrival Time: ";
        cin >> processes[i].arrival_time;
        cout << "Burst Time: ";
        cin >> processes[i].burst_time;
        cout << "Priority: ";
        cin >> processes[i].priority;
        cout << endl;
    }

    vector<Process> processesCopy;
    processesCopy = processes;

    // Performing preemptive priority scheduling
    preemptivePriorityScheduling(processes);

    cout << "Preemptive Priority Scheduling:" << endl;
    displayResult(processes);

    // Performing non-preemptive priority scheduling
    nonPreemptivePriorityScheduling(processesCopy);

    cout << "Non-Preemptive Priority Scheduling:" << endl;
    displayResult(processesCopy);

    return 0;
}
```

Output:

```
Enter the number of processes: 5
Enter the arrival time, burst time, and priority for each process:
Process 1:
Arrival Time: 0
Burst Time: 3
Priority: 3
Process 2:
Arrival Time: 1
Burst Time: 4
Priority: 2
Process 3:
Arrival Time: 2
Burst Time: 6
Priority: 4
Process 4:
Arrival Time: 3
Burst Time: 4
Priority: 6
Process 5:
Arrival Time: 5
Burst Time: 2
Priority: 10
Preemptive Priority Scheduling:
   Process   Arrival Time     Burst Time       PriorityCompletion TimeTurnaround Time   Waiting Time
        P1              0              3              3              7              7              4
        P2              1              4              2              5              4              0
        P3              2              6              4             13             11              5
        P4              3              4              6             17             14             10
        P5              5              2             10             19             14             12
Average Turnaround Time: 10
Average Waiting Time: 6.2
Non-Preemptive Priority Scheduling:
   Process   Arrival Time     Burst Time       PriorityCompletion TimeTurnaround Time   Waiting Time
        P2              1              4              2              4              3              0
        P1              0              3              3              7              7              4
        P3              2              6              4             13             11              5
P4              3              4              6             17             14             10
        P5              5              2             10             19             14             12
Average Turnaround Time: 9.8
Average Waiting Time: 6.2
```

EXPERIMENT-4

Aim:

Write a program to implement CPU scheduling for Round Robin.

Theory:

Code:

```
#include <bits/stdc++.h>

using namespace std;

struct Process {
    int id;
    int burst_time;
    int arrival_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

void roundRobin(vector<Process>& processes, int time_quantum) {
    int n = processes.size();

    vector<int> remaining_time(n);
    for (int i = 0; i < n; i++) {
        remaining_time[i] = processes[i].burst_time;
    }

    int current_time = 0;
    int completed = 0;
    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0) {
                int execute_time = min(remaining_time[i], time_quantum);
                remaining_time[i] -= execute_time;
                current_time += execute_time;

                if (remaining_time[i] == 0) {
                    processes[i].completion_time = current_time;
                    completed++;
                }
            }
        }
    }

    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].completion_time -
processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;
    }
}

void displayResult(const vector<Process>& processes) {
```

```
    cout << "Process\tBurst Time\tArrival Time\tCompletion Time\tTurnaround
Time\tWaiting Time" << endl;
    for (const Process& p : processes) {
        cout << p.id << "\t" << p.burst_time << "\t\t" << p.arrival_time <<
"\t\t" << p.completion_time << "\t\t" << p.turnaround_time << "\t\t" <<
p.waiting_time << endl;
    }

    double total_turnaround_time = 0;
    double total_waiting_time = 0;
    for (const Process& p : processes) {
        total_turnaround_time += p.turnaround_time;
        total_waiting_time += p.waiting_time;
    }

    double avg_turnaround_time = total_turnaround_time / processes.size();
    double avg_waiting_time = total_waiting_time / processes.size();
    cout << "Average Turnaround Time: " << avg_turnaround_time << endl;
    cout << "Average Waiting Time: " << avg_waiting_time << endl;
}

int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    vector<Process> processes(n);

    cout << "Enter the arrival time and burst time for each process:" << endl;
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        cout << "Process " << i + 1 << ":" << endl;
        cout << "Arrival Time: ";
        cin >> processes[i].arrival_time;
        cout << "Burst Time: ";
        cin >> processes[i].burst_time;
        cout << endl;
    }

    int time_quantum;
    cout << "Enter the time quantum: ";
    cin >> time_quantum;

    roundRobin(processes, time_quantum);
    displayResult(processes);

    return 0;
}
```

Output:

```
Enter the number of processes: 6
Enter the arrival time and burst time for each process:
Process 1:
Arrival Time: 0
Burst Time: 7

Process 2:
Arrival Time: 1
Burst Time: 4

Process 3:
Arrival Time: 2
Burst Time: 15

Process 4:
Arrival Time: 3
Burst Time: 11

Process 5:
Arrival Time: 4
Burst Time: 20

Process 6:
Arrival Time: 4
Burst Time: 9

Enter the time quantum: 5
Process Burst Time      Arrival Time    Completion Time Turnaround Time Waiting Time
1       7               0               31              31              24
2       4               1               9               8               4
3       15              2               55              53              38
4       11              3               56              53              42
5       20              4               66              62              42
6       9               4               50              46              37
Average Turnaround Time: 42.1667
Average Waiting Time: 31.1667
```

EXPERIMENT-5.2

Aim:

Wite a program for page replacement policy using Least Recently Use (LRU) algorithm.

Theory:

Code:

```
#include <bits/stdc++.h>

using namespace std;

void printList(list<int> l) {
    for (int page : l) {
        cout << page << " ";
    }
}

int main() {
    int frameSize, numReferences;

    cout << "Enter the number of frames: ";
    cin >> frameSize;

    cout << "Enter the number of page references: ";
    cin >> numReferences;

    cout << "Enter the page reference string: ";
    vector<int> pageReferences(numReferences);
    for (int i = 0; i < numReferences; i++) {
        cin >> pageReferences[i];
    }

    list<int> lruList;
    unordered_set<int> pageSet;
    int pageFaults = 0;

    cout << "\nPage Insertion Order:" << endl;

    for (int i = 0; i < numReferences; i++) {
        int currentPage = pageReferences[i];

        if (pageSet.find(currentPage) == pageSet.end()) {
            if (lruList.size() == frameSize) {
                int leastRecentlyUsedPage = lruList.back();
                lruList.pop_back();
                pageSet.erase(leastRecentlyUsedPage);
            }

            lruList.push_front(currentPage);
            pageSet.insert(currentPage);
            pageFaults++;

            cout << "Page " << currentPage << " inserted. ";
            cout << "Current List: ";
```

```
            printList(lruList);
            cout << endl;
        } else {
            lruList.remove(currentPage);
            lruList.push_front(currentPage);

            cout << "Page " << currentPage << " already in frame. ";
            cout << "Current List: ";
            printList(lruList);
            cout << endl;
        }
    }

    cout << "\nTotal Page Faults: " << pageFaults << endl;

    return 0;
}
```

Output:

```
PS C:\Users\ankus\OneDrive\Desktop\test> ./LRU.exe
Enter the number of frames: 3
Enter the number of page references: 7
Enter the page reference string: 1 3 0 3 5 6 3

Page Insertion Order:
Page 1 inserted. Current List: 1
Page 3 inserted. Current List: 3 1
Page 0 inserted. Current List: 0 3 1
Page 3 already in frame. Current List: 3 0 1
Page 5 inserted. Current List: 5 3 0
Page 6 inserted. Current List: 6 5 3
Page 3 already in frame. Current List: 3 6 5

Total Page Faults: 5
```

EXPERIMENT-5.3

Aim:

Wite a program for page replacement policy using Optimal algorithm.

Theory:

Code:

```
#include <bits/stdc++.h>

using namespace std;

void printFrames(vector<int> frames) {
    for (int page : frames) {
        cout << page << " ";
    }
}

int main() {
    int frameSize, numReferences;

    cout << "Enter the number of frames: ";
    cin >> frameSize;

    cout << "Enter the number of page references: ";
    cin >> numReferences;

    cout << "Enter the page reference string: ";
    vector<int> pageReferences(numReferences);
    for (int i = 0; i < numReferences; i++) {
        cin >> pageReferences[i];
    }

    vector<int> frames(frameSize, -1);
    int pageFaults = 0;

    cout << "\nPage Insertion Order:" << endl;

    for (int i = 0; i < numReferences; i++) {
        int currentPage = pageReferences[i];

        if (find(frames.begin(), frames.end(), currentPage) == frames.end()) {
            int indexToReplace = -1;
            int farthestReference = -1;

            for (int j = 0; j < frameSize; j++) {
                int nextPage = pageReferences.size();
                for (int k = i + 1; k < numReferences; k++) {
                    if (frames[j] == pageReferences[k]) {
                        nextPage = k;
                        break;
                    }
                }
                if (nextPage > farthestReference) {
                    farthestReference = nextPage;
```

```
                    indexToReplace = j;
                }
            }
            frames[indexToReplace] = currentPage;
            pageFaults++;

            cout << "Page " << currentPage << " inserted. ";
            cout << "Current Frames: ";
            printFrames(frames);
            cout << endl;
        } else {
            cout << "Page " << currentPage << " already in frame. ";
            cout << "Current Frames: ";
            printFrames(frames);
            cout << endl;
        }
    }
    cout << "\nTotal Page Faults: " << pageFaults << endl;

    return 0;
}
```

Output:

```
PS C:\Users\ankus\OneDrive\Desktop\test> ./optimal.exe
Enter the number of frames: 3
Enter the number of page references: 20
Enter the page reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Page Insertion Order:
Page 7 inserted. Current Frames: 7 -1 -1
Page 0 inserted. Current Frames: 7 0 -1
Page 1 inserted. Current Frames: 7 0 1
Page 2 inserted. Current Frames: 2 0 1
Page 0 already in frame. Current Frames: 2 0 1
Page 3 inserted. Current Frames: 2 0 3
Page 0 already in frame. Current Frames: 2 0 3
Page 4 inserted. Current Frames: 2 4 3
Page 2 already in frame. Current Frames: 2 4 3
Page 3 already in frame. Current Frames: 2 4 3
Page 0 inserted. Current Frames: 2 0 3
Page 3 already in frame. Current Frames: 2 0 3
Page 2 already in frame. Current Frames: 2 0 3
Page 1 inserted. Current Frames: 2 0 1
Page 2 already in frame. Current Frames: 2 0 1
Page 0 already in frame. Current Frames: 2 0 1
Page 1 already in frame. Current Frames: 2 0 1
Page 7 inserted. Current Frames: 7 0 1
Page 0 already in frame. Current Frames: 7 0 1
Page 1 already in frame. Current Frames: 7 0 1

Total Page Faults: 9
```

EXPERIMENT-6.1

Aim:

Wite a program for memory management using First Fit algorithm.

Theory:

Code:

```
#include <iostream>
using namespace std;

void firstFit(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++) {
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i]) {
                allocation[i] = j;
                block[j] -= process[i];
                break;
            }
        }
    }

    cout << "First Fit Allocation:\n";
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            cout << "Process " << i + 1 << " allocated to Block " <<
allocation[i] + 1 << endl;
        } else {
            cout << "Process " << i + 1 << " cannot be allocated\n";
        }
    }
}

int main() {
    int m, n;

    cout << "Enter the number of memory blocks: ";
    cin >> m;
    int block[m];

    cout << "Enter the sizes of memory blocks:\n";
    for (int i = 0; i < m; i++) {
        cin >> block[i];
    }

    cout << "Enter the number of processes: ";
    cin >> n;
    int process[n];
```

```
    cout << "Enter the sizes of processes:\n";
    for (int i = 0; i < n; i++) {
        cin >> process[i];
    }

    firstFit(block, m, process, n);

    return 0;
}
```

Output:

```
Enter the number of memory blocks: 6
Enter the sizes of memory blocks:
200 400 600 500 300 250
Enter the number of processes: 4
Enter the sizes of processes:
357 210 468 491
First Fit Allocation:
Process 1 allocated to Block 2
Process 2 allocated to Block 3
Process 3 allocated to Block 4
Process 4 cannot be allocated
```

EXPERIMENT-6.2

Aim:

Wite a program for memory management using Best Fit algorithm.

Theory:

Code:

```
#include <iostream>
using namespace std;

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

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

    cout << "Best Fit Allocation:\n";
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            cout << "Process " << i + 1 << " allocated to Block " <<
allocation[i] + 1 << endl;
        } else {
            cout << "Process " << i + 1 << " cannot be allocated\n";
        }
    }
}

int main() {
    int m, n;

    cout << "Enter the number of memory blocks: ";
    cin >> m;
    int block[m];

    cout << "Enter the sizes of memory blocks:\n";
    for (int i = 0; i < m; i++) {
        cin >> block[i];
```

```
    }

    cout << "Enter the number of processes: ";
    cin >> n;
    int process[n];

    cout << "Enter the sizes of processes:\n";
    for (int i = 0; i < n; i++) {
        cin >> process[i];
    }

    bestFit(block, m, process, n);

    return 0;
}
```

Output:

```
Enter the number of memory blocks: 6
Enter the sizes of memory blocks:
200 400 600 500 300 250
Enter the number of processes: 4
Enter the sizes of processes:
357 210 468 491
Best Fit Allocation:
Process 1 allocated to Block 2
Process 2 allocated to Block 6
Process 3 allocated to Block 4
Process 4 allocated to Block 3
```

EXPERIMENT-6.3

Aim:

Wite a program for memory management using Worst Fit algorithm.

Theory:

Code:

```
#include <iostream>
using namespace std;

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

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

    cout << "Worst Fit Allocation:\n";
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            cout << "Process " << i + 1 << " allocated to Block " <<
allocation[i] + 1 << endl;
        } else {
            cout << "Process " << i + 1 << " cannot be allocated\n";
        }
    }
}

int main() {
    int m, n;

    cout << "Enter the number of memory blocks: ";
    cin >> m;
    int block[m];

    cout << "Enter the sizes of memory blocks:\n";
    for (int i = 0; i < m; i++) {
        cin >> block[i];
```

```
    }

    cout << "Enter the number of processes: ";
    cin >> n;
    int process[n];

    cout << "Enter the sizes of processes:\n";
    for (int i = 0; i < n; i++) {
        cin >> process[i];
    }

    worstFit(block, m, process, n);

    return 0;
}
```

Output:

```
Enter the number of memory blocks: 6
Enter the sizes of memory blocks:
200 400 600 500 300 250
Enter the number of processes: 4
Enter the sizes of processes:
357 210 468 491
Worst Fit Allocation:
Process 1 allocated to Block 3
Process 2 allocated to Block 4
Process 3 cannot be allocated
Process 4 cannot be allocated
```

EXPERIMENT-7

Aim:

Write a program to implement reader/writer problem using semaphore.

Code:

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
using namespace std;

class monitor{

private:
    int rcnt;
    int wcnt;
    int waitr;
    int waitw;
    pthread_cond_t canread;
    pthread_cond_t canwrite;
    pthread_mutex_t condlock;

public:
    monitor(){
        rcnt = 0;
        wcnt = 0;
        waitr = 0;
        waitw = 0;
        pthread_cond_init(&canread, NULL);
        pthread_cond_init(&canwrite, NULL);
        pthread_mutex_init(&condlock, NULL);
    }
    void beginread(int i){
        pthread_mutex_lock(&condlock);
        if (wcnt == 1 || waitw > 0){
            waitr++;
            pthread_cond_wait(&canread, &condlock);
            waitr--;
        }

        rcnt++;
        cout << "reader " << i << " is reading\n";
        pthread_mutex_unlock(&condlock);
        pthread_cond_broadcast(&canread);
    }
```

```
    void endread(int i){
        pthread_mutex_lock(&condlock);
        if (--rcnt == 0)
            pthread_cond_signal(&canwrite);
        pthread_mutex_unlock(&condlock);
    }
    void beginwrite(int i){
        pthread_mutex_lock(&condlock);
        if (wcnt == 1 || rcnt > 0){
            ++waitw;
            pthread_cond_wait(&canwrite, &condlock);
            --waitw;
        }
        wcnt = 1;
        cout << "writer " << i << " is writing\n";
        pthread_mutex_unlock(&condlock);
    }
    void endwrite(int i){
        pthread_mutex_lock(&condlock);
        wcnt = 0;
        if (waitr > 0)
            pthread_cond_signal(&canread);
        else
            pthread_cond_signal(&canwrite);
        pthread_mutex_unlock(&condlock);
    }
}
M;
void *reader(void *id){
    int c = 0;
    int i = *(int *)id;
    while (c < 5){
        usleep(1);
        M.beginread(i);
        M.endread(i);
        c++;
    }
}
void *writer(void *id){
    int c = 0;
    int i = *(int *)id;
    while (c < 5){
        usleep(1);
        M.beginwrite(i);
        M.endwrite(i);
        c++;
    }
}
```

```
int main(){
    pthread_t r[5], w[5];
    int id[5];
    for (int i = 0; i < 5; i++){
        id[i] = i;
        pthread_create(&r[i], NULL, &reader, &id[i]);
        pthread_create(&w[i], NULL, &writer, &id[i]);
    }
    for (int i = 0; i < 5; i++){
        pthread_join(r[i], NULL);
    }
    for (int i = 0; i < 5; i++){
        pthread_join(w[i], NULL);
    }
}
```

Output:

```
eader 0 is reading
eader 1 is reading
riter 0 is writing
riter 0 is writing
eader 2 is reading
eader 4 is reading
eader 3 is reading
eader 1 is reading
eader 0 is reading
riter 1 is writing
riter 2 is writing
eader 2 is reading
eader 1 is reading
eader 4 is reading
eader 0 is reading
eader 3 is reading
riter 4 is writing
riter 3 is writing
eader 2 is reading
eader 4 is reading
eader 1 is reading
eader 3 is reading
eader 0 is reading
riter 0 is writing
riter 0 is writing
riter 1 is writing
eader 2 is reading
riter 0 is writing
eader 1 is reading
eader 4 is reading
eader 3 is reading
eader 0 is reading
riter 2 is writing
eader 2 is reading
riter 4 is writing
riter 4 is writing
riter 3 is writing
eader 4 is reading
riter 4 is writing
eader 3 is reading
riter 1 is writing
riter 4 is writing
riter 2 is writing
riter 3 is writing
riter 2 is writing
riter 1 is writing
riter 2 is writing
riter 3 is writing
riter 1 is writing
```

EXPERIMENT-8

Aim:

Write a program to implement Producer-Consumer problem using semaphores.

Code:

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 10, x = 0;
void producer(){
    --mutex;
    ++full;
    --empty;
    x++;
    printf("\nProducer produces""item %d",x);
    ++mutex;
}
void consumer(){
    --mutex;
    --full;
    ++empty;
    printf("\nConsumer consumes ""item %d",x);
    x--;
    ++mutex;
}
int main(){
    int n, i;
    printf("\n1. Press 1 for Producer \n2. Press 2 for Consumer \n3.
Press 3 for Exit");
    #pragma omp critical
    for (i = 1; i > 0; i++){
        printf("\nEnter your choice:");
        scanf("%d", &n);
        switch (n){
        case 1:
            if ((mutex == 1) && (empty != 0)){
                producer();
            }
            else{
                printf("Buffer is full!");
            }
            break;
        case 2:
```

```
            if ((mutex == 1)&& (full != 0)){
                consumer();
            }
            else{
                printf("Buffer is empty!");
            }
            break;
        case 3:
            exit(0);
            break;
        }
    }
}
```

Output:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer producesitem 1
Enter your choice:2

Consumer consumes item 1
Enter your choice:3
```