Name : Ruturaj Sandip Sutar **Roll No : 59** **Div : B**

Batch : 2 **PRN:-12310720**

Lab 4: CPU Scheduling algorithms

1. FCFS

Code:-

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>

using namespace std;

struct Process {
    int id;             // Process ID
    int arrivalTime;    // Arrival Time
    int burstTime;      // Burst Time
    int completionTime; // Completion Time
    int turnaroundTime; // Turnaround Time
    int waitingTime;    // Waiting Time

};

bool compareArrival(Process a, Process b) {
    return a.arrivalTime < b.arrivalTime;
}
```

```
void displayGanttChart(vector<Process> &processes) {
    cout << "\nGantt Chart:\n ";

    for (size_t i = 0; i < processes.size(); i++) {
        cout << "+-------";
    }
    cout << "+\n";

    cout << "|";
    for (auto &p : processes) {
        cout << "  P" << p.id << setw(5) << "|";
    }
    cout << "\n ";

    for (size_t i = 0; i < processes.size(); i++) {
        cout << "+-------";
    }
    cout << "+\n";

    cout << "0";
    for (auto &p : processes) {
        cout << setw(8) << p.completionTime;
    }
    cout << "\n";
}
```

```
int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;
    float avgWT=0;
    float avgTAT=0;
    vector<Process> processes(n);

    // Input process details
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        cout << "Enter arrival time and burst time for process P" << processes[i].id
<< ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
    }

    // Sort processes based on arrival time
    sort(processes.begin(), processes.end(), compareArrival);

    // Calculate Completion, Turnaround, and Waiting times
    int currentTime = 0;
    for (int i = 0; i < n; i++) {
        if (currentTime < processes[i].arrivalTime) {
            currentTime = processes[i].arrivalTime; // Idle until the process arrives
        }
        processes[i].completionTime = currentTime + processes[i].burstTime;
```

```
        currentTime = processes[i].completionTime;
        processes[i].turnaroundTime = processes[i].completionTime - processes[i].arrivalTime;
        processes[i].waitingTime = processes[i].turnaroundTime - processes[i].burstTime;
    }
    for(auto &p:processes)
    {
        avgTAT+=p.turnaroundTime;
        avgWT+=p.waitingTime;
    }
    avgTAT/=n;
    avgWT/=n;
    // Display Process Information
    cout << "\nProcess\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n";
    for (auto &p : processes) {
        cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime << "\t"
             << p.completionTime << "\t\t" << p.turnaroundTime << "\t\t" << p.waitingTime << "\n";
    }

    cout<<"The average Waiting time :  "<<avgWT<<"\n";
    cout<<"The average turn around time :  "<<avgTAT<<"\n";

    displayGanttChart(processes);

    return 0;}
```

Output :-

Enter the number of processes: 3

Enter arrival time and burst time for process P1: 0 4

Enter arrival time and burst time for process P2: 2 5

Enter arrival time and burst time for process P3: 4 2

Process	Arrival	Burst	Completion	Turnaround	Waiting
P1	0	4	4	4	0
P2	2	5	9	7	2
P3	4	2	11	7	5

The average Waiting time : 2.33333

The average turn around time : 6

Gantt Chart:

```
 +-------+-------+-------+
| P1    | P2    | P3    |
 +-------+-------+-------+
0      4      9      11
```

2. SJF

Code:-

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include<limits.h>
using namespace std;

struct Process {
    int id;
    int arrivalTime;
    int burstTime;
    int completionTime;
    int turnaroundTime;
    int waitingTime;
    bool isCompleted;
};

bool compareArrival(Process a, Process b) {
    return a.arrivalTime < b.arrivalTime;
}

void displayGanttChart(vector<Process> &processes) {
    cout << "\nGantt Chart:\n ";
```

```
    for (size_t i = 0; i < processes.size(); i++) {
        cout << "+-------";
    }
    cout << "+\n";

    cout << "|";
    for (auto &p : processes) {
        cout << "  P" << p.id << setw(5) << "|";
    }
    cout << "\n ";

    for (size_t i = 0; i < processes.size(); i++) {
        cout << "+-------";
    }
    cout << "+\n";

    cout << "0";
    for (auto &p : processes) {
        cout << setw(8) << p.completionTime;
    }
    cout << "\n";
}
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 and burst time for process P" << processes[i].id
<< ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
        processes[i].isCompleted = false;
    }
    sort(processes.begin(), processes.end(), compareArrival);
    int completed = 0, currentTime = 0;
    double totalWaitingTime = 0, totalTurnaroundTime = 0;
    vector<Process> ganttChart;
    while (completed < n) {
        int idx = -1;
        int minBurstTime = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (processes[i].arrivalTime <= currentTime &&
!processes[i].isCompleted) {
                if (processes[i].burstTime < minBurstTime) {
                    minBurstTime = processes[i].burstTime;
                    idx = i;}
                if (processes[i].burstTime == minBurstTime) {
                    if (processes[i].arrivalTime < processes[idx].arrivalTime) {
                        idx = i;}}}}
        if (idx == -1) {
            currentTime++;
```

```
    } else {
        processes[idx].completionTime = currentTime +
processes[idx].burstTime;
        processes[idx].turnaroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
        processes[idx].waitingTime = processes[idx].turnaroundTime -
processes[idx].burstTime;

        totalWaitingTime += processes[idx].waitingTime;
        totalTurnaroundTime += processes[idx].turnaroundTime;

        processes[idx].isCompleted = true;
        currentTime = processes[idx].completionTime;
        completed++;

        ganttChart.push_back(processes[idx]);  }}
  cout << "\nProcess\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n";
  for (auto &p : processes) {
    cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime << "\t"
        << p.completionTime << "\t\t" << p.turnaroundTime << "\t\t" <<
p.waitingTime << "\n";
  }
  displayGanttChart(ganttChart);
  cout << fixed << setprecision(2);
  cout << "\nAverage Waiting Time: " << totalWaitingTime / n << endl;
  cout << "Average Turnaround Time: " << totalTurnaroundTime / n << endl;
  return 0;}
```

Output:-

Enter the number of processes: 3

Enter arrival time and burst time for process P1: 2 6

Enter arrival time and burst time for process P2: 0 2

Enter arrival time and burst time for process P3: 3 5

Process	Arrival	Burst	Completion	Turnaround	Waiting
P2	0	2	2	2	0
P1	2	6	8	6	0
P3	3	5	13	10	5

Gantt Chart:

```
 +-------+-------+-------+
| P2    | P1    | P3    |
 +-------+-------+-------+
0       2       8      13
```

Average Waiting Time: 1.67

Average Turnaround Time: 6.00

3. SRTF

Code:-

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

```
#include <iomanip>
#include <limits>
using namespace std;

struct Process {
    int id;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int completionTime;
    int waitingTime;
    int turnaroundTime;
};

void printGanttChart(const vector<int> &ganttChart, int totalTime) {
    cout << "\nGantt Chart:\n";
    for (int i = 0; i < totalTime; i++) {
        cout << "| P" << ganttChart[i] << " ";
    }
    cout << "|\n0";
    for (int i = 1; i <= totalTime; i++) {
        cout << "   " << i;
    }
    cout << endl;
}
```

```
void printProcessTable(const vector<Process> &processes, double avgWT,
double avgTAT) {
   cout << "\nProcess Table:\n";
   cout << setw(5) << "ID" << setw(15) << "Arrival Time" << setw(15) << "Burst
Time"
        << setw(20) << "Completion Time" << setw(15) << "Waiting Time"
        << setw(20) << "Turnaround Time" << endl;

   for (const auto &p : processes) {
      cout << setw(5) << p.id << setw(15) << p.arrivalTime << setw(15) <<
p.burstTime
           << setw(20) << p.completionTime << setw(15) << p.waitingTime
           << setw(20) << p.turnaroundTime << endl;
   }

   cout << "\nAverage Waiting Time: " << avgWT << endl;
   cout << "Average Turnaround Time: " << avgTAT << endl;
}

void srtf(vector<Process> &processes) {
   int n = processes.size();
   vector<int> ganttChart;
   int completed = 0, currentTime = 0;
   double totalWT = 0, totalTAT = 0;

   while (completed < n) {
      int idx = -1;
```

```
        int minTime = numeric_limits<int>::max();

        for (int i = 0; i < n; i++) {
            if (processes[i].arrivalTime <= currentTime &&
processes[i].remainingTime > 0 &&
                processes[i].remainingTime < minTime) {
                minTime = processes[i].remainingTime;
                idx = i;
            }
        }

        if (idx != -1) {
            ganttChart.push_back(processes[idx].id);
            processes[idx].remainingTime--;
            currentTime++;

            if (processes[idx].remainingTime == 0) {
                processes[idx].completionTime = currentTime;
                processes[idx].turnaroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
                processes[idx].waitingTime = processes[idx].turnaroundTime -
processes[idx].burstTime;

                totalWT += processes[idx].waitingTime;
                totalTAT += processes[idx].turnaroundTime;
                completed++;
            }
```

```
        } else {
            ganttChart.push_back(0); // 0 represents idle time
            currentTime++;
        }
    }

    double avgWT = totalWT / n;
    double avgTAT = totalTAT / n;
    printGanttChart(ganttChart, currentTime);
    printProcessTable(processes, avgWT, avgTAT);
}

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 and burst time for process P" << i + 1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
        processes[i].remainingTime = processes[i].burstTime;
    }

    srtf(processes);
```

return 0;}

Output:-

Enter the number of processes: 4

Enter arrival time and burst time for process P1: 3 2

Enter arrival time and burst time for process P2: 1 5

Enter arrival time and burst time for process P3: 0 6

Enter arrival time and burst time for process P4: 6 4

Gantt Chart:

| P3 | P2 | P2 | P1 | P1 | P2 | P2 | P2 | P4 | P4 | P4 | P4 | P3 | P3 | P3 | P3 | P3 |

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Process Table:

ID	Arrival Time	Burst Time	Completion Time	Waiting Time	Turnaround Time
1	3	2	5	0	2
2	1	5	8	2	7
3	0	6	17	11	17
4	6	4	12	2	6

Average Waiting Time: 3.75

Average Turnaround Time: 8

4. Priority-NonPreemptive:-

Code:-

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

struct Process {
    int id;
    int arrivalTime;
    int burstTime;
    int priority;
    int completionTime;
    int waitingTime;
    int turnaroundTime;
};

// Function to print the process table and calculate average times
void printProcessTable(const vector<Process> &processes, double avgWT, double avgTAT) {
    cout << "\nProcess Table:\n";
    cout << setw(5) << "ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time"
         << setw(10) << "Priority" << setw(20) << "Completion Time"
```

```
        << setw(15) << "Waiting Time" << setw(20) << "Turnaround Time" << endl;

    for (const auto &p : processes) {
        cout << setw(5) << p.id << setw(15) << p.arrivalTime << setw(15) << p.burstTime
             << setw(10) << p.priority << setw(20) << p.completionTime
             << setw(15) << p.waitingTime << setw(20) << p.turnaroundTime << endl;
    }

    cout << "\nAverage Waiting Time: " << avgWT << endl;
    cout << "Average Turnaround Time: " << avgTAT << endl;
}

// Priority Non-Preemptive Scheduling Function
void priorityNonPreemptive(vector<Process> &processes) {
    int n = processes.size();
    vector<int> isCompleted(n, 0);
    vector<Process> ganttChart;
    double totalWT = 0, totalTAT = 0;
    int currentTime = 0, completed = 0;

    while (completed < n) {
        int idx = -1;
        int highestPriority = -1;

        // Find the highest priority process that has arrived
```

```
    for (int i = 0; i < n; ++i) {
        if (processes[i].arrivalTime <= currentTime && !isCompleted[i]) {
            if (processes[i].priority > highestPriority) {
                highestPriority = processes[i].priority;
                idx = i;
            }
        }
    }

    if (idx != -1) {
        ganttChart.push_back(processes[idx]);
        currentTime += processes[idx].burstTime;
        processes[idx].completionTime = currentTime;
        processes[idx].turnaroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
        processes[idx].waitingTime = processes[idx].turnaroundTime -
processes[idx].burstTime;

        totalWT += processes[idx].waitingTime;
        totalTAT += processes[idx].turnaroundTime;
        isCompleted[idx] = 1;
        completed++;
    } else {
        currentTime++;
    }
  }
```

```
    double avgWT = totalWT / n;
    double avgTAT = totalTAT / n;
    printProcessTable(processes, avgWT, avgTAT);
}

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, burst time, and priority for process P" << i + 1
<< ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime >>
processes[i].priority;
    }

    priorityNonPreemptive(processes);
    return 0;
}
```

Output:-

```
Enter the number of processes: 3
Enter arrival time, burst time, and priority for process P1: 2 4 2
Enter arrival time, burst time, and priority for process P2: 0 3 3
```

Enter arrival time, burst time, and priority for process P3: 1 4 1

Process Table:

ID	Arrival Time	Burst Time	Priority	Completion Time	Waiting Time	Turnaround Time
1	2	4	2	7	1	5
2	0	3	3	3	0	3
3	1	4	1	11	6	10

Average Waiting Time: 2.33333

Average Turnaround Time: 6

5. Priority-preemptive

Code:-

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <algorithm>
using namespace std;

struct Process {
    int id;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int priority;
```

```
    int completionTime;
    int waitingTime;
    int turnaroundTime;
};

// Function to print the process table and calculate average times
void printProcessTable(const vector<Process> &processes, double avgWT,
double avgTAT) {
    cout << "\nProcess Table:\n";
    cout << setw(5) << "ID" << setw(15) << "Arrival Time" << setw(15) << "Burst
Time"
         << setw(10) << "Priority" << setw(20) << "Completion Time"
         << setw(15) << "Waiting Time" << setw(20) << "Turnaround Time" << endl;

    for (const auto &p : processes) {
        cout << setw(5) << p.id << setw(15) << p.arrivalTime << setw(15) <<
p.burstTime
             << setw(10) << p.priority << setw(20) << p.completionTime
             << setw(15) << p.waitingTime << setw(20) << p.turnaroundTime <<
endl;
    }

    cout << "\nAverage Waiting Time: " << avgWT << endl;
    cout << "Average Turnaround Time: " << avgTAT << endl;
}

// Priority Preemptive Scheduling Function
```

```
void priorityPreemptive(vector<Process> &processes) {
    int n = processes.size();
    vector<int> isCompleted(n, 0);
    double totalWT = 0, totalTAT = 0;
    int currentTime = 0, completed = 0;
    int lastExecution = -1;

    while (completed < n) {
        int idx = -1;
        int highestPriority = -1;

        // Find the highest priority process that has arrived and is not completed
        for (int i = 0; i < n; ++i) {
            if (processes[i].arrivalTime <= currentTime && !isCompleted[i]) {
                if (processes[i].priority > highestPriority) {
                    highestPriority = processes[i].priority;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            if (lastExecution != idx) {
                lastExecution = idx;
            }
```

```
            processes[idx].remainingTime--;
            currentTime++;

            // If the process is completed
            if (processes[idx].remainingTime == 0) {
                processes[idx].completionTime = currentTime;
                processes[idx].turnaroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
                processes[idx].waitingTime = processes[idx].turnaroundTime -
processes[idx].burstTime;

                totalWT += processes[idx].waitingTime;
                totalTAT += processes[idx].turnaroundTime;
                isCompleted[idx] = 1;
                completed++;
            }
        } else {
            currentTime++;
        }
    }

    double avgWT = totalWT / n;
    double avgTAT = totalTAT / n;
    printProcessTable(processes, avgWT, avgTAT);
}

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, burst time, and priority for process P" << i + 1
<< ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime >>
processes[i].priority;
        processes[i].remainingTime = processes[i].burstTime;
    }

    priorityPreemptive(processes);
    return 0;
}
```

Output:-

```
Enter the number of processes: 4
Enter arrival time, burst time, and priority for process P1: 2 5 2
Enter arrival time, burst time, and priority for process P2: 0 3 4
Enter arrival time, burst time, and priority for process P3: 2 4 1
Enter arrival time, burst time, and priority for process P4: 1 2 3

Process Table:
```

ID	Arrival Time	Burst Time	Priority	Completion Time	Waiting Time	Turnaround Time
1	2	5	2	10	3	8
2	0	3	4	3	0	3
3	2	4	1	14	8	12
4	1	2	3	5	2	4

Average Waiting Time: 3.25

Average Turnaround Time: 6.75

6. Round Robin

Code:-

```
#include <iostream>
#include <vector>
#include <queue>
#include <set>
#include <iomanip>
using namespace std;

struct Process {
    int id;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int completionTime;
    int waitingTime;
```

```
    int turnaroundTime;
};

// Function to print the Gantt Chart
void printGanttChart(const vector<pair<int, int>> &ganttChart) {
    cout << "\nGantt Chart:\n";
    for (auto &p : ganttChart) {
        cout << "| P" << p.first << " ";
    }
    cout << "|\n0";
    for (auto &p : ganttChart) {
        cout << "   " << p.second;
    }
    cout << endl;
}

// Function to print the process table and calculate average times
void printProcessTable(const vector<Process> &processes, double avgWT,
double avgTAT) {
    cout << "\nProcess Table:\n";
    cout << setw(5) << "ID" << setw(15) << "Arrival Time" << setw(15) << "Burst
Time"
         << setw(20) << "Completion Time" << setw(15) << "Waiting Time"
         << setw(20) << "Turnaround Time" << endl;

    for (auto &p : processes) {
```

```
        cout << setw(5) << p.id << setw(15) << p.arrivalTime << setw(15) <<
p.burstTime
             << setw(20) << p.completionTime << setw(15) << p.waitingTime
             << setw(20) << p.turnaroundTime << endl;
    }

    cout << "\nAverage Waiting Time: " << avgWT << endl;
    cout << "Average Turnaround Time: " << avgTAT << endl;
}

// Round Robin Scheduling Function
void roundRobin(vector<Process> &processes, int timeQuantum) {
    int n = processes.size();
    queue<int> q;
    set<int> inQueue; // To keep track of processes already in the queue
    vector<pair<int, int>> ganttChart;
    int currentTime = 0, completed = 0;
    double totalWT = 0, totalTAT = 0;

    // Add initial processes that have arrived at time 0
    for (int i = 0; i < n; ++i) {
        if (processes[i].arrivalTime <= currentTime) {
            q.push(i);
            inQueue.insert(i);
        }
    }
```

```
    while (completed < n) {
        if (q.empty()) {
            currentTime++;
            for (int i = 0; i < n; ++i) {
                if (processes[i].arrivalTime <= currentTime &&
processes[i].remainingTime > 0 && inQueue.find(i) == inQueue.end()) {
                    q.push(i);
                    inQueue.insert(i);
                }
            }
            continue;
        }

        int idx = q.front();
        q.pop();
        inQueue.erase(idx);
        ganttChart.push_back({processes[idx].id, currentTime});

        // Execute the process for time quantum or remaining time, whichever is
smaller
        int executionTime = min(timeQuantum, processes[idx].remainingTime);
        currentTime += executionTime;
        processes[idx].remainingTime -= executionTime;

        // Add to the Gantt chart
        ganttChart.push_back({processes[idx].id, currentTime});
```

```
    // Check if the process is completed
    if (processes[idx].remainingTime == 0) {
        processes[idx].completionTime = currentTime;
        processes[idx].turnaroundTime = processes[idx].completionTime -
processes[idx].arrivalTime;
        processes[idx].waitingTime = processes[idx].turnaroundTime -
processes[idx].burstTime;

        totalWT += processes[idx].waitingTime;
        totalTAT += processes[idx].turnaroundTime;
        completed++;
    }

    // Push the next arrived processes
    for (int i = 0; i < n; ++i) {
        if (processes[i].arrivalTime <= currentTime &&
processes[i].remainingTime > 0 && inQueue.find(i) == inQueue.end()) {
            q.push(i);
            inQueue.insert(i);
        }
    }

    // Reinsert the current process if it is not yet finished
    if (processes[idx].remainingTime > 0) {
        q.push(idx);
        inQueue.insert(idx);
    }
```

```
    }

    double avgWT = totalWT / n;
    double avgTAT = totalTAT / n;
    printGanttChart(ganttChart);
    printProcessTable(processes, avgWT, avgTAT);
}

int main() {
    int n, timeQuantum;
    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 and burst time for process P" << i + 1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
        processes[i].remainingTime = processes[i].burstTime;
    }

    cout << "Enter the time quantum: ";
    cin >> timeQuantum;

    roundRobin(processes, timeQuantum);
    return 0;
```

}

Output:-

Enter the number of processes: 3

Enter arrival time and burst time for process P1: 0

Enter arrival time and burst time for process P2: 1

5

Enter arrival time and burst time for process P3: 3

4

Enter the time quantum: 10

Gantt Chart:

| P1 | P1 | P2 | P2 | P3 | P3 |

0 0 2 2 7 7 11

Process Table:

ID	Arrival Time	Burst Time	Completion Time	Waiting Time	Turnaround Time
1	0	2	2	0	2
2	1	5	7	1	6
3	3	4	11	4	8

Average Waiting Time: 1.66667

Average Turnaround Time: 5.33333