ASSIGNMENT 5

To create c++ programs for the different scheduling algorithms.

First Come First Serve (FCFS) Scheduling

Algorithm Overview

FCFS is a non-preemptive scheduling algorithm where processes are executed in the order of their arrival. The process that arrives first is allocated the CPU first. Key metrics include:
Waiting Time (WT): Time a process waits in the ready queue.
Turnaround Time (TAT): Total time from arrival to completion (WT + Burst Time).

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

using namespace std;

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

void calculateTimes(vector<Process>& processes) {
    processes[0].waitingTime = 0;
    int currentTime = processes[0].arrivalTime + processes[0].burstTime;

    for (size_t i = 1; i < processes.size(); ++i) {
        processes[i].waitingTime = max(currentTime - processes[i].arrivalTime, 0);
        currentTime += processes[i].burstTime;
        processes[i].turnaroundTime = processes[i].waitingTime +
processes[i].burstTime;
    }
}

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

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

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

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

  calculateTimes(processes);

  cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";
  for (const auto& p : processes) {
    cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime
       << "\t" << p.waitingTime << "\t" << p.turnaroundTime << endl;
  }

  return 0;
}
```

Output:

```
Enter number of processes: 3
Enter arrival and burst time for P1: 0 5
Enter arrival and burst time for P2: 1 3
Enter arrival and burst time for P3: 2 8

Process Arrival Burst   Waiting Turnaround
P1      0       5       0       0
P2      1       3       4       7
P3      2       8       6       14


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

Shortest Job First (SJF) Scheduling (Preemptive)

Algorithm Overview

SJF prioritizes processes with the shortest burst time. The preemptive variant (Shortest Remaining Time First) allows interrupting the current process if a shorter job arrives

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

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

void sjfPreemptive(vector<Process>& processes) {
    int currentTime = 0;
    int completed = 0;
    int n = processes.size();

    while (completed != n) {
        int shortest = -1;
        int minRemaining = INT_MAX;

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

        if (shortest == -1) {
            currentTime++;
            continue;
        }

        processes[shortest].remainingTime--;
        currentTime++;

        if (processes[shortest].remainingTime == 0) {
```

```
            processes[shortest].completionTime = currentTime;
            processes[shortest].turnaroundTime =
processes[shortest].completionTime - processes[shortest].arrivalTime;
            processes[shortest].waitingTime = processes[shortest].turnaroundTime
- processes[shortest].burstTime;
            completed++;
        }
    }
}

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

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

    cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";
    for (const auto& p : processes) {
        cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime
             << "\t" << p.waitingTime << "\t" << p.turnaroundTime << endl;
    }

    return 0;
}
```

Output:

```
Enter number of processes: 4
Enter arrival and burst time for P1: 0 7
Enter arrival and burst time for P2: 2 4
Enter arrival and burst time for P3: 4 1
Enter arrival and burst time for P4: 5 4

Process Arrival Burst    Waiting Turnaround
P1      0       7        9       16
P2      2       4        1       5
P3      4       1        0       1
P4      5       4        2       6


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


Round Robin Scheduling

Algorithm Overview

Round Robin assigns a fixed time quantum to each process, cycling through the ready queue Processes are preempted after the quantum expires and requeued.

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

using namespace std;

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

void roundRobin(vector<Process>& processes, int quantum) {
   queue<int> readyQueue;
   int currentTime = 0;
   int n = processes.size();
```

```
    vector<int> startTime(n, -1);

    int index = 0;
    while (index < n || !readyQueue.empty()) {
        while (index < n && processes[index].arrivalTime <= currentTime) {
            readyQueue.push(index);
            index++;
        }

        if (readyQueue.empty()) {
            currentTime++;
            continue;
        }

        int currentIdx = readyQueue.front();
        readyQueue.pop();

        if (startTime[currentIdx] == -1) {
            startTime[currentIdx] = currentTime;
        }

        int executionTime = min(processes[currentIdx].remainingTime, quantum);
        processes[currentIdx].remainingTime -= executionTime;
        currentTime += executionTime;

        while (index < n && processes[index].arrivalTime <= currentTime) {
            readyQueue.push(index);
            index++;
        }

        if (processes[currentIdx].remainingTime > 0) {
            readyQueue.push(currentIdx);
        } else {
            processes[currentIdx].turnaroundTime = currentTime -
processes[currentIdx].arrivalTime;
            processes[currentIdx].waitingTime =
processes[currentIdx].turnaroundTime - processes[currentIdx].burstTime;
        }
    }
}

int main() {
    int n, quantum;
```

```
    cout << "Enter number of processes: ";
    cin >> n;
    cout << "Enter time quantum: ";
    cin >> quantum;

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

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

    roundRobin(processes, quantum);

    cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n";
    for (const auto& p : processes) {
        cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime
             << "\t" << p.waitingTime << "\t" << p.turnaroundTime << endl;
    }

    return 0;
}
```

Output:

```
Enter number of processes: 3
Enter time quantum: 4
Enter arrival and burst time for P1: 0 10
Enter arrival and burst time for P2: 1 5
Enter arrival and burst time for P3: 2 8

Process Arrival Burst   Waiting Turnaround
P1      0       10      13      23
P2      1       5       11      16
P3      2       8       11      19


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