Assignment 3

Code

```
/*
Problem Statement - Implement C program for CPU scheduling algorithms:
ShortestJobFirst(SJF) and Round Robin with different arrival time.
*/

#include <bits/stdc++.h>
using namespace std;
#define MAX_SIZE 100

struct process {
    string id = "##";
    float arrival_time = -1.0, burst_time;
};

class Queue {
    private:
        int item, i;
        process arr_queue[MAX_SIZE];
        int rear;
        int front;

    public:
        int current_size;
        Queue() {
            rear = 0;
            front = 0;
            current_size = 0;
        }

        bool insert(process item) {
            if (Queue::isFull()) {
                cout << "\n## Queue Reached Max!, CPU buffer overflow!\n";
                return false;
            }
            arr_queue[rear++] = item;
            current_size++;
            return true;
```

```
        }

        process pop() {
            if (Queue::isEmpty()) {
                cout << "\n## Queue is Empty!";
                process empty = { "##", -1, 0 };
                return empty;
            }
            front++;
            current_size--;
            return arr_queue[front - 1];
        }

        process frontItem() {
            if (Queue::isEmpty()) {
                cout << "\n## Queue is Empty!";
                process empty = { "##", -1, 0 };
                return empty;
            }
            return arr_queue[front];
        }

        bool isFull() {
            if (rear == MAX_SIZE)
                return true;
            return false;
        }

        bool isEmpty() {
        if (front == rear)
            return true;
        return false;
    }
};

enum algorithm {
    SJF_NON_PREEMPTIVE = 1,
    SJF_PREEMPTIVE = 2,
    ROUND_ROBIN = 3
};

void insertionSort(process given[], int size) {
    for (int step = 1; step < size; step++) {
        process key = given[step];
        int j = step - 1;

        while (key.arrival_time < given[j].arrival_time && j >= 0) {
            given[j + 1] = given[j];
            --j;
        }
```

```
        given[j + 1] = key;
    }
}

void takeInput(process given[], int no_of_processes) {
    int AT = 0, BT = 0;
    cout << "\nEnter " << no_of_processes << " processes details :";
    for (int i = 0; i < no_of_processes; i++)
    {
        cout << "\nProcess P" << i << " :\n\tAT - ";
        cin >> AT;
        cout << "\tBT - ";
        cin >> BT;
        given[i].id = to_string(i);
        given[i].arrival_time = AT;
        given[i].burst_time = BT;
    }
}

void displayProcessQueue(process given[], int size) {
    cout << "\nGiven process queue is :\n-------------------------\n";
    cout << "Process ID -   | ";
    for (size_t i = 0; i < size; i++)
    {
        cout << "  P" << given[i].id << "  |  ";
    }
    cout << "\nArrival Time - | ";
    for (size_t i = 0; i < size; i++)
    {
        cout << "  " << given[i].arrival_time << "   |  ";
    }
    cout << "\nBurst Time -   | ";
    for (size_t i = 0; i < size; i++)
    {
        cout << "  " << given[i].burst_time << "   |  ";
    }
    cout << "\n-------------------------\n\n";
}

void calculateStats(float answers[], process given[], process scheduleQueue[], int
total_time_taken) {
    float total_burst_time = 0, total_turn_around_time = 0.0, no_of_processes =
0.0;
    for (size_t i = 1; i <= total_time_taken; i++) {
        if ((scheduleQueue[i].id != "##") && (scheduleQueue[i].burst_time == 0)) {
            no_of_processes++;
            total_turn_around_time += (i + 2 -
given[stoi(scheduleQueue[i].id)].arrival_time);
            total_burst_time += given[stoi(scheduleQueue[i].id)].burst_time;
        }
```

```
    }
    answers[0] = (total_turn_around_time - total_time_taken) / no_of_processes;
    answers[1] = total_turn_around_time / no_of_processes;
}

void processScheduler(process given[], int no_of_processes, int choice) {
    switch (choice) {
        case SJF_PREEMPTIVE: {
            cout << "\nAfter scheduling with SJF Preemption :";
            cout << "\n----------------------------------------------------------
\n";
            cout << "         ...Sorting the processes by AT & BT...";
            cout << "\n----------------------------------------------------------
\n";
            cout << "\nScheduled process queue is :\n-------------------------\n";
            process scheduledQueue[MAX_SIZE];
            float statusQueue[no_of_processes];
            for (size_t i = 0; i < no_of_processes; i++) {
                statusQueue[i] = -1;
            }
            int current_time = 0, min_burst_time;
            bool allFinished;
            while (true) {
                allFinished = true;
                min_burst_time = INT_MAX;
                for (size_t process = 0; process < no_of_processes; process++) {
                    if (current_time == given[process].arrival_time &&
statusQueue[process] == -1) {
                        statusQueue[process] = given[process].burst_time;
                    }
                    if (statusQueue[process] != -1) {
                        if (statusQueue[process] != 0) {
                            allFinished = false;
                            if (statusQueue[process] <= min_burst_time) {
                                min_burst_time = statusQueue[process];
                                scheduledQueue[current_time].id =
to_string(process);
                            }
                        }
                    }
                    else allFinished = false;
                }
                if (allFinished) break;
                if (min_burst_time == INT_MAX) {
                    cout << "\n------ " << current_time << "\n| " <<
scheduledQueue[current_time].id << " |";
                    current_time++;
                    continue;
                }
                else {
```

```
                    int choosen_process = stoi(scheduledQueue[current_time].id);
                    if (statusQueue[choosen_process] != 0) {
                        if (statusQueue[choosen_process] ==
given[choosen_process].burst_time) {
                            scheduledQueue[current_time].arrival_time =
current_time;
                        }
                        statusQueue[choosen_process] -= 1;
                        scheduledQueue[current_time].burst_time =
statusQueue[choosen_process];
                        cout << "\n------ " << current_time << "\n| P" <<
scheduledQueue[current_time].id << " |";
                    }
                    current_time++;
                }
            }
            float scheduledQueueStats[2];
            calculateStats(scheduledQueueStats, given, scheduledQueue,
current_time);
            cout << "\n------ " << current_time << "\n\nAverage Waiting Time: " <<
scheduledQueueStats[0] << "\nAverage Turn Around Time: " << scheduledQueueStats[1]
<< "\n-------------------------\n";
            return;
        }

        case SJF_NON_PREEMPTIVE: {
            cout << "\n---------------------------------------------------------
\n";
            cout << "            ...Sorting the processes by AT...";
            cout << "\n---------------------------------------------------------
\n";
            cout << "\nAfter scheduling with SJF Non-Preemption :";
            insertionSort(given, no_of_processes);
            displayProcessQueue(given, no_of_processes);
            return;
        }

        case ROUND_ROBIN: {
            int time_quantum;
            cout << "\nEnter the Time Quantum: ";
            cin >> time_quantum;
            cout << "\nAfter scheduling with Round Robin :";
            cout << "\nScheduled process queue is :\n-------------------------\n";
            insertionSort(given, no_of_processes);
            int current_time = 0, no_of_processes_completed = 0;
            bool completed_processes[no_of_processes] = { false };
            process scheduledQueue[MAX_SIZE];
            Queue readyQueue = Queue();
            while (no_of_processes_completed < no_of_processes) {
                if (readyQueue.isEmpty()) {
```

```
                    if (current_time > given[no_of_processes - 1].arrival_time)
return;
                    bool isIdle = true;
                    for (int process_no = 0; process_no < no_of_processes;
process_no++) {
                        if ((given[process_no].arrival_time == current_time) &&
(given[process_no].burst_time > 0)) {
                            if (isIdle) {
                                scheduledQueue[current_time].id =
to_string(process_no);
                                scheduledQueue[current_time].arrival_time =
current_time;
                                int initial_burst_time =
given[process_no].burst_time;
                                int assigned_time = (initial_burst_time /
time_quantum) == 0 ? initial_burst_time : time_quantum;
                                scheduledQueue[current_time].burst_time =
initial_burst_time - assigned_time;
                                cout << "\n------ " << current_time << "\n| P" <<
scheduledQueue[current_time].id << " |";
                                float remaining_burst_time =
scheduledQueue[current_time].burst_time;
                                int timer = assigned_time + 1;
                                while (--timer) {
                                    current_time++;
                                    for (int process = 0; process <
no_of_processes; process++) {
                                        if ((given[process].arrival_time ==
current_time) && (given[process].burst_time > 0) && !completed_processes[process])
{
                                            if
(!readyQueue.insert(given[process])) {
                                                cout << "\nReturned";
                                                return;
                                            }
                                        }
                                    }
                                }
                                if (remaining_burst_time > 0) {
                                    readyQueue.insert({ to_string(process_no),
(float)current_time, remaining_burst_time });
                                }
                                else {
                                    completed_processes[process_no] = true;
                                    no_of_processes_completed++;
                                }
                            }
                            isIdle = false;
                        }
                    }
```

```
                    if (isIdle) {
                        cout << "\n------ " << current_time << "\n| " <<
scheduledQueue[current_time].id << " |";
                        current_time++;
                    };
                }
                else {
                    process process_at_front = readyQueue.pop();
                    if (process_at_front.burst_time > 0) {
                        scheduledQueue[current_time].id = process_at_front.id;
                        if (process_at_front.burst_time ==
given[stoi(process_at_front.id)].burst_time) {
                            scheduledQueue[current_time].arrival_time =
current_time;
                        }
                        int burst_time = process_at_front.burst_time;
                        int assigned_time = (burst_time / time_quantum) == 0 ?
burst_time : time_quantum;
                        scheduledQueue[current_time].burst_time = burst_time -
assigned_time;
                        cout << "\n------ " << current_time << "\n| P" <<
scheduledQueue[current_time].id << " |";
                        float remaining_burst_time =
scheduledQueue[current_time].burst_time;
                        int timer = assigned_time + 1;
                        while (--timer) {
                            current_time++;
                            for (int process = 0; process < no_of_processes;
process++) {
                                if ((given[process].arrival_time == current_time)
&& (given[process].burst_time > 0) && !completed_processes[process]) {
                                    if (!readyQueue.insert(given[process])) {
                                        cout << "\n Error: CPU queue is full!\n";
                                        return;
                                    }
                                }
                            }
                        }
                        if (remaining_burst_time > 0) {
                            if (!readyQueue.insert({ process_at_front.id,
(float)current_time, remaining_burst_time })) {
                                cout << "\n Error: CPU queue is full!\n";
                                return;
                            }
                        }
                        else {
                            completed_processes[stoi(process_at_front.id)] = true;
                            no_of_processes_completed++;
                        }
                        if ((current_time > given[no_of_processes -
```

```
1].arrival_time) && readyQueue.isEmpty()) {
                            float scheduledQueueStats[2];
                            calculateStats(scheduledQueueStats, given,
scheduledQueue, current_time);
                            cout << "\n------ " << current_time << "\n\nAverage
Waiting Time: " << scheduledQueueStats[0] << "\nAverage Turn Around Time: " <<
scheduledQueueStats[1] << "\n-------------------------\n";
                            return;
                        };
                    }
                    else {
                        current_time++;
                    }
                }
            }
            float scheduledQueueStats[2];
            calculateStats(scheduledQueueStats, given, scheduledQueue,
current_time);
            cout << "\n------ " << current_time << "\n\nAverage Waiting Time: " <<
scheduledQueueStats[0] << "\nAverage Turn Around Time: " << scheduledQueueStats[1]
<< "\n-------------------------\n";
            return;
        }

        default: {
            cout << "\n\nInvalid algorithm choice!\n";
            return;
        }
        }
}

int main() {
    int no_of_processes;
    cout << "\nEnter no. of processes : ";
    cin >> no_of_processes;
    process given[no_of_processes];
    takeInput(given, no_of_processes);
    displayProcessQueue(given, no_of_processes);
    cout << "\nChoose scheduling algorithm : 1. SJF Non-Preemptive\t2. SJF
Preemptive\t3. Round Robin\n\tEnter choice : ";
    int choice;
    cin >> choice;
    processScheduler(given, no_of_processes, choice);
    return 0;
}
```

Output

```
abhishek-jadhav@abhishek-jadhav-ubuntu:~/Codes/OS Assignments/33232$ ./a.out

Enter no. of processes : 5

Enter 5 processes details :
Process P0 :
        AT - 1
        BT - 2

Process P1 :
        AT - 2
        BT - 1

Process P2 :
        AT - 3
        BT - 3

Process P3 :
        AT - 4
        BT - 1

Process P4 :
        AT - 5
        BT - 2

Given process queue is :
-------------------------
Process ID -   |   P0  |    P1  |    P2  |    P3  |    P4  |
Arrival Time - |   1   |    2   |    3   |    4   |    5   |
Burst Time -   |   2   |    1   |    3   |    1   |    2   |
-------------------------


Choose scheduling algorithm : 1. SJF Non-Preemptive     2. SJF Preemptive       3.
Round Robin
        Enter choice : 1

---------------------------------------------------------
            ...Sorting the processes by AT...
---------------------------------------------------------

After scheduling with SJF Non-Preemption :
Given process queue is :
-------------------------
Process ID -   |   P0  |    P1  |    P2  |    P3  |    P4  |
Arrival Time - |   1   |    2   |    3   |    4   |    5   |
Burst Time -   |   2   |    1   |    3   |    1   |    2   |
```

```
-------------------------

abhishek-jadhav@abhishek-jadhav-ubuntu:~/Codes/OS Assignments/33232$ ./a.out

Enter no. of processes : 5

Enter 5 processes details :
Process P0 :
        AT - 1
        BT - 2

Process P1 :
        AT - 2
        BT - 1

Process P2 :
        AT - 3
        BT - 3

Process P3 :
        AT - 4
        BT - 1

Process P4 :
        AT - 5
        BT - 2

Given process queue is :
-------------------------
Process ID -   |   P0  |    P1  |    P2  |    P3  |    P4  |
Arrival Time - |   1   |    2   |    3   |    4   |    5   |
Burst Time -   |   2   |    1   |    3   |    1   |    2   |
-------------------------


Choose scheduling algorithm : 1. SJF Non-Preemptive     2. SJF Preemptive       3.
Round Robin
        Enter choice : 2

After scheduling with SJF Preemption :
----------------------------------------------------------
         ...Sorting the processes by AT & BT...
----------------------------------------------------------

Scheduled process queue is :
-------------------------

------ 0
| ## |
------ 1
```

```
| P0 |
------ 2
| P1 |
------ 3
| P0 |
------ 4
| P3 |
------ 5
| P4 |
------ 6
| P4 |
------ 7
| P2 |
------ 8
| P2 |
------ 9
| P2 |
------ 10

Average Waiting Time: 1.8
Average Turn Around Time: 3.8
-------------------------

abhishek-jadhav@abhishek-jadhav-ubuntu:~/Codes/OS Assignments/33232$ ./a.out

Enter no. of processes : 5

Enter 5 processes details :
Process P0 :
        AT - 1
        BT - 2

Process P1 :
        AT - 2
        BT - 1

Process P2 :
        AT - 3
        BT - 3

Process P3 :
        AT - 4
        BT - 1

Process P4 :
        AT - 5
        BT - 2

Given process queue is :
-------------------------
```

```
Process ID -   |   P0  |    P1  |    P2  |    P3  |    P4  |
Arrival Time - |   1   |    2   |    3   |    4   |    5   |
Burst Time -   |   2   |    1   |    3   |    1   |    2   |
-------------------------


Choose scheduling algorithm : 1. SJF Non-Preemptive     2. SJF Preemptive       3.
Round Robin
        Enter choice : 3

Enter the Time Quantum: 3

After scheduling with Round Robin :
Scheduled process queue is :
-------------------------

------ 0
| ## |
------ 1
| P0 |
------ 3
| P1 |
------ 4
| P2 |
------ 7
| P3 |
------ 8
| P4 |
------ 10

Average Waiting Time: 1.6
Average Turn Around Time: 3.6
-------------------------
```