PROGRAM 2

Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

```
#include <stdio.h>
#include <limits.h>

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

void fcfs(struct Process processes[], int n) {
    int total_waiting_time = 0;
    int total_turnaround_time = 0;
    int currentTime = 0;

    for (int i = 0; i < n; i++) {
        // If CPU is idle until this process arrives
        if (currentTime < processes[i].arrival_time) {
            currentTime = processes[i].arrival_time;
        }

        processes[i].waiting_time = currentTime - processes[i].arrival_time;
        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;

        currentTime += processes[i].burst_time;

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

    float average_waiting_time = (float)total_waiting_time / n;
    float average_turnaround_time = (float)total_turnaround_time / n;

    printf("FCFS Scheduling:\n");
    printf("Average Waiting Time: %.2f\n", average_waiting_time);
    printf("Average Turnaround Time: %.2f\n", average_turnaround_time);
}
```

```
void sjf(struct Process processes[], int n) {
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    int is_completed[100] = {0};
    int completed = 0;
    int currentTime;

    // Start from the minimum arrival time
    int min_at = INT_MAX;
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time < min_at) {
            min_at = processes[i].arrival_time;
        }
    }
    currentTime = min_at;

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

        // Select process with minimum burst time among arrived & not completed
        for (int i = 0; i < n; i++) {
            if (!is_completed[i] && processes[i].arrival_time <= currentTime) {
                if (processes[i].burst_time < min_bt ||
                    (processes[i].burst_time == min_bt &&
                     processes[i].arrival_time < processes[idx].arrival_time) ||
                    (processes[i].burst_time == min_bt &&
                     processes[i].arrival_time == processes[idx].arrival_time &&
                     processes[i].id < processes[idx].id)) {
                    min_bt = processes[i].burst_time;
                    idx = i;
                }
            }
        }

        // If no process has arrived yet, move time forward
        if (idx == -1) {
            currentTime++;
            continue;
        }

        processes[idx].waiting_time = currentTime - processes[idx].arrival_time;
        currentTime += processes[idx].burst_time;
        processes[idx].turnaround_time = processes[idx].waiting_time +
processes[idx].burst_time;
```

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

        is_completed[idx] = 1;
        completed++;
    }

    float average_waiting_time = (float)total_waiting_time / n;
    float average_turnaround_time = (float)total_turnaround_time / n;

    printf("\nSJF Scheduling:\n");
    printf("Average Waiting Time: %.2f\n", average_waiting_time);
    printf("Average Turnaround Time: %.2f\n", average_turnaround_time);
}

void priority(struct Process processes[], int n) {
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    int is_completed[100] = {0};
    int completed = 0;
    int currentTime;

    // Start from the minimum arrival time
    int min_at = INT_MAX;
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time < min_at) {
            min_at = processes[i].arrival_time;
        }
    }
    currentTime = min_at;

    while (completed < n) {
        int idx = -1;
        int best_priority = INT_MAX; // smaller value = higher priority

        // Select highest priority (smallest number) among arrived & not completed
        for (int i = 0; i < n; i++) {
            if (!is_completed[i] && processes[i].arrival_time <= currentTime) {
                if (processes[i].priority < best_priority ||
                    (processes[i].priority == best_priority &&
                     processes[i].arrival_time < processes[idx].arrival_time) ||
                    (processes[i].priority == best_priority &&
                     processes[i].arrival_time == processes[idx].arrival_time &&
                     processes[i].id < processes[idx].id)) {
```

```
                best_priority = processes[i].priority;
                idx = i;
            }
        }
    }

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

    processes[idx].waiting_time = currentTime - processes[idx].arrival_time;
    currentTime += processes[idx].burst_time;
    processes[idx].turnaround_time = processes[idx].waiting_time +
processes[idx].burst_time;

    total_waiting_time += processes[idx].waiting_time;
    total_turnaround_time += processes[idx].turnaround_time;

    is_completed[idx] = 1;
    completed++;
  }

  float average_waiting_time = (float)total_waiting_time / n;
  float average_turnaround_time = (float)total_turnaround_time / n;

  printf("\nPriority Scheduling:\n");
  printf("Average Waiting Time: %.2f\n", average_waiting_time);
  printf("Average Turnaround Time: %.2f\n", average_turnaround_time);
}

void roundRobin(struct Process processes[], int n, int time_quantum) {
  int total_waiting_time = 0;
  int total_turnaround_time = 0;

  int rem[100];
  int in_queue[100] = {0};
  int completed = 0;
  int currentTime = 0;

  for (int i = 0; i < n; i++) {
    processes[i].remaining_time = processes[i].burst_time;
    rem[i] = processes[i].burst_time;
    processes[i].waiting_time = 0;
    processes[i].turnaround_time = 0;
  }
```

```
int queue[1000];
int front = 0, rear = 0;

// Find first arriving process
int first = 0;
int min_at = processes[0].arrival_time;
for (int i = 1; i < n; i++) {
    if (processes[i].arrival_time < min_at) {
        min_at = processes[i].arrival_time;
        first = i;
    }
}
currentTime = processes[first].arrival_time;
queue[rear++] = first;
in_queue[first] = 1;

while (completed < n) {
    // If queue is empty, jump to next arriving process
    if (front == rear) {
        int next_idx = -1;
        int next_at = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (rem[i] > 0 && processes[i].arrival_time < next_at) {
                next_at = processes[i].arrival_time;
                next_idx = i;
            }
        }
        if (next_idx == -1) break; // safety
        currentTime = processes[next_idx].arrival_time;
        queue[rear++] = next_idx;
        in_queue[next_idx] = 1;
    }

    int i = queue[front++];

    if (rem[i] > time_quantum) {
        currentTime += time_quantum;
        rem[i] -= time_quantum;
    } else {
        currentTime += rem[i];
        rem[i] = 0;
        completed++;

        processes[i].turnaround_time = currentTime - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
```

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

        // Add newly arrived processes to queue
        for (int j = 0; j < n; j++) {
            if (rem[j] > 0 && !in_queue[j] && processes[j].arrival_time <= currentTime) {
                queue[rear++] = j;
                in_queue[j] = 1;
            }
        }

        // If current process still has remaining time, push it back
        if (rem[i] > 0) {
            queue[rear++] = i;
        }
    }

    float average_waiting_time = (float)total_waiting_time / n;
    float average_turnaround_time = (float)total_turnaround_time / n;

    printf("\nRound Robin Scheduling:\n");
    printf("Average Waiting Time: %.2f\n", average_waiting_time);
    printf("Average Turnaround Time: %.2f\n", average_turnaround_time);
}

int main() {
    int n, time_quantum;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter arrival time for Process %d: ", i + 1);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority for Process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    fcfs(processes, n);
    sjf(processes, n);
    priority(processes, n);

    printf("\nEnter time quantum for Round Robin: ");
    scanf("%d", &time_quantum);
    roundRobin(processes, n, time_quantum);

    return 0;
}
```

OUTPUT:

```
INPUT
Enter the number of processes --              4
Enter Burst Time for Process 0 --             6
Enter Burst Time for Process 1 --             8
Enter Burst Time for Process 2 --             7
Enter Burst Time for Process 3 --             3
OUTPUT
PROCESS                    BURST      WAITING      TURNARO
                           TIME       TIME         UND TIME
P3                         3          0            3
P0                         6          3            9
P2                         7          9            16
P1                         8          16           24
Average Waiting Time --               7.000000
Average Turnaround Time --            13.000000
```