Priority scheduling and round robin algorithm

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

#define MAX 10

// Structure to store process details
struct Process {
    int pid;           // Process ID
    int arrival_time;  // Arrival time
    int burst_time;    // Burst time
    int priority;      // Priority
    int remaining_time; // Remaining burst time (for preemptive)
    int start_time;    // Start time
    int end_time;      // End time
    int waiting_time;  // Waiting time
    int turnaround_time; // Turnaround time
};

// Function to calculate waiting and turnaround time
void calculateTimes(struct Process proc[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        proc[i].turnaround_time = proc[i].end_time - proc[i].arrival_time;
        proc[i].waiting_time = proc[i].turnaround_time - proc[i].burst_time;
        total_waiting_time += proc[i].waiting_time;
        total_turnaround_time += proc[i].turnaround_time;
    }
    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}

// Non-Preemptive Priority Scheduling Algorithm
void prioritySchedulingNonPreemptive(struct Process proc[], int n) {
    // Sort processes based on arrival time first (then by priority)
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            // Sort by arrival time first, then by priority if arrival times are the same
            if (proc[i].arrival_time > proc[j].arrival_time ||
                (proc[i].arrival_time == proc[j].arrival_time && proc[i].priority >
proc[j].priority)) {
                struct Process temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }
```

```
    int time = 0;
    int completed = 0;
    int isCompleted[MAX] = {0};  // Track if processes are completed

    while (completed < n) {
        int idx = -1;
        int min_priority = 9999; // Start with a large value for comparison

        // Find the process with the highest priority that has arrived and not completed
        for (int i = 0; i < n; i++) {
            if (!isCompleted[i] && proc[i].arrival_time <= time) {
                if (proc[i].priority < min_priority) {  // Lower priority number is higher priority
                    min_priority = proc[i].priority;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            // Execute the selected process
            if (proc[idx].remaining_time == proc[idx].burst_time) {
                proc[idx].start_time = time;  // Record the start time when it begins
execution
            }
            proc[idx].end_time = time + proc[idx].burst_time;  // Completion time of the
process
            time += proc[idx].burst_time;  // Move time forward by the burst time of the
selected process
            proc[idx].remaining_time = 0;  // The process is completed
            isCompleted[idx] = 1;  // Mark as completed
            completed++;
        } else {
            time++;  // If no process can run, just move time forward
        }
    }

    calculateTimes(proc, n);  // Calculate waiting time and turnaround time for each
process
}


// Preemptive Priority Scheduling Algorithm
void prioritySchedulingPreemptive(struct Process proc[], int n) {
    int time = 0, completed = 0;
    int isCompleted[MAX] = {0}; // To track completed processes

    // Initialize remaining burst time
    for (int i = 0; i < n; i++) {
        proc[i].remaining_time = proc[i].burst_time;
```

```
    }

    while (completed < n) {
        int idx = -1;
        int min_priority = 9999;

        // Find process with highest priority and shortest remaining time
        for (int i = 0; i < n; i++) {
            if (!isCompleted[i] && proc[i].arrival_time <= time && proc[i].priority <
min_priority) {
                min_priority = proc[i].priority;
                idx = i;
            }
        }

        if (idx != -1) {
            if (proc[idx].remaining_time == proc[idx].burst_time) {
                proc[idx].start_time = time;
            }

            proc[idx].remaining_time--;
            time++;

            // If the process is completed
            if (proc[idx].remaining_time == 0) {
                proc[idx].end_time = time;
                isCompleted[idx] = 1;
                completed++;
            }
        } else {
            time++;
        }
    }

    calculateTimes(proc, n);
}

// Round Robin Scheduling Algorithm
void roundRobin(struct Process proc[], int n, int quantum) {
    int time = 0, completed = 0;
    int isCompleted[MAX] = {0}; // To track completed processes

    // Initialize remaining burst time
    for (int i = 0; i < n; i++) {
        proc[i].remaining_time = proc[i].burst_time;
    }

    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0) {
```

```
            if (proc[i].remaining_time == proc[i].burst_time) {
                proc[i].start_time = time;
            }

            int time_slice = (proc[i].remaining_time < quantum) ?
proc[i].remaining_time : quantum;
            proc[i].remaining_time -= time_slice;
            time += time_slice;

            if (proc[i].remaining_time == 0) {
                proc[i].end_time = time;
                isCompleted[i] = 1;
                completed++;
            }
        }
    }
}

    calculateTimes(proc, n);
}

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

    printf("Enter number of processes: ");
    scanf("%d", &n);

    struct Process proc[MAX];

    // Input process details
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter details for Process %d:\n", proc[i].pid);
        printf("Arrival Time: ");
        scanf("%d", &proc[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &proc[i].burst_time);
        printf("Priority: ");
        scanf("%d", &proc[i].priority);
    }

    printf("\nChoose scheduling algorithm:\n");
    printf("1. Non-Preemptive Priority Scheduling\n");
    printf("2. Preemptive Priority Scheduling\n");
    printf("3. Round Robin Scheduling\n");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            prioritySchedulingNonPreemptive(proc, n);
```

```
            break;
        case 2:
            priorityScheduledPreemptive(proc, n);
            break;
        case 3:
            printf("Enter time quantum: ");
            scanf("%d", &quantum);
            roundRobin(proc, n, quantum);
            break;
        default:
            printf("Invalid choice!\n");
    }

    return 0;
}
```

Output

```
Enter number of processes: 3
Enter details for Process 1:
Arrival Time: 0
Burst Time: 4
Priority: 2
Enter details for Process 2:
Arrival Time: 0
Burst Time: 5
Priority: 2
Enter details for Process 3:
Arrival Time: 0 6
Burst Time: Priority: 4

Choose scheduling algorithm:
1. Non-Preemptive Priority Scheduling
2. Preemptive Priority Scheduling
3. Round Robin Scheduling
1
Average Waiting Time: 4.33
Average Turnaround Time: 9.33
```

```
Enter number of processes: 3
Enter details for Process 1:
Arrival Time: 0
Burst Time: 4
Priority: 2
Enter details for Process 2:
Arrival Time: 0
Burst Time: 5
Priority: 2
Enter details for Process 3:
Arrival Time: 0
Burst Time: 6
Priority: 4

Choose scheduling algorithm:
1. Non-Preemptive Priority Scheduling
2. Preemptive Priority Scheduling
3. Round Robin Scheduling
2
Average Waiting Time: 4.33
Average Turnaround Time: 9.33
```

```
Enter number of processes: 3
Enter details for Process 1:
Arrival Time: 0
Burst Time: 4
Priority: 2
Enter details for Process 2:
Arrival Time: 0
Burst Time: 5
Priority: 2
Enter details for Process 3:
Arrival Time: 0
Burst Time: 6
Priority: 4

Choose scheduling algorithm:
1. Non-Preemptive Priority Scheduling
2. Preemptive Priority Scheduling
3. Round Robin Scheduling
3
Enter time quantum: 2
Average Waiting Time: 7.00
Average Turnaround Time: 12.00
```