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

struct Process {
   int pid, arrival_time, burst_time, priority, waiting_time, turnaround_time, completion_time, remaining_time;
};

void swap(struct Process *a, struct Process *b) {
   struct Process temp = *a;
   *a = *b;
   *b = temp;
}

void display(struct Process p[], int n, const char* algo, char execution_order[]) {
   printf("\n%s\t%s\t%s\t%s\n", "Algorithm", "Order of Execution", "CT (P1, P2, P3)", "WT (P1, P2, P3)");
   printf("%s\t%s\t(%d, %d, %d)\t(%d, %d, %d)\n", algo, execution_order, p[0].completion_time, p[1].completion_time, p[2].completion_time, p[0].waiting_time, p[1].waiting_time, p[2].waiting_time);
}

void calculateFCFS(struct Process p[], int n, char execution_order[]) {
   int time = 0;
   for (int i = 0; i < n; i++) {
      if (time < p[i].arrival_time)
         time = p[i].arrival_time;
      p[i].completion_time = time + p[i].burst_time;
      p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
```

```
        p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
        time = p[i].completion_time;
        execution_order[i] = 'P' + p[i].pid;
    }
    execution_order[n] = '\0';
}

void calculateSJF(struct Process p[], int n, char execution_order[]) {
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (p[j].burst_time < p[i].burst_time)
                swap(&p[i], &p[j]);
    calculateFCFS(p, n, execution_order);
}

void calculateSJFPreemptive(struct Process p[], int n, char execution_order[]) {
    int time = 0, completed = 0, min_index, min_remaining = INT_MAX;
    bool is_completed[n];
    for (int i = 0; i < n; i++) {
        is_completed[i] = false;
        p[i].remaining_time = p[i].burst_time;
    }
    while (completed != n) {
        min_index = -1;
        min_remaining = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (p[i].arrival_time <= time && !is_completed[i] && p[i].remaining_time < min_remaining) {
                min_index = i;
                min_remaining = p[i].remaining_time;
```

```
            }
        }
        if (min_index == -1) {
            time++;
        } else {
            execution_order[time] = 'P' + p[min_index].pid;
            p[min_index].remaining_time--;
            time++;
            if (p[min_index].remaining_time == 0) {
                p[min_index].completion_time = time;
                p[min_index].turnaround_time = p[min_index].completion_time - p[min_index].arrival_time;
                p[min_index].waiting_time = p[min_index].turnaround_time - p[min_index].burst_time;
                is_completed[min_index] = true;
                completed++;
            }
        }
    }
    execution_order[time] = '\0';
}

void calculatePriority(struct Process p[], int n, char execution_order[]) {
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (p[j].priority < p[i].priority)
                swap(&p[i], &p[j]);
    calculateFCFS(p, n, execution_order);
}

void calculateRoundRobin(struct Process p[], int n, int quantum, char execution_order[]) {
```

```
    int time = 0, completed = 0, index = 0;
    for (int i = 0; i < n; i++)
        p[i].remaining_time = p[i].burst_time;
    while (completed < n) {
        bool done = true;
        for (int i = 0; i < n; i++) {
            if (p[i].remaining_time > 0) {
                done = false;
                execution_order[index++] = 'P' + p[i].pid;
                if (p[i].remaining_time > quantum) {
                    time += quantum;
                    p[i].remaining_time -= quantum;
                } else {
                    time += p[i].remaining_time;
                    p[i].remaining_time = 0;
                    p[i].completion_time = time;
                    p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
                    p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
                    completed++;
                }
            }
        }
        if (done) break;
    }
    execution_order[index] = '\0';
}

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

```
printf("Enter number of processes: ");
scanf("%d", &n);
struct Process p[n];
printf("Enter arrival time, burst time, and priority for each process:\n");
for (int i = 0; i < n; i++) {
   p[i].pid = i + 1;
   printf("P%d (AT BT Priority): ", i + 1);
   scanf("%d %d %d", &p[i].arrival_time, &p[i].burst_time, &p[i].priority);
}
char execution_order[100];
do {
   printf("\nCPU Scheduling Algorithms Menu:");
   printf("\n1. First Come First Serve (FCFS)");
   printf("\n2. Shortest Job First (SJF) - Non-Preemptive");
   printf("\n3. Shortest Job First (SJF) - Preemptive");
   printf("\n4. Priority Scheduling (Non-Preemptive)");
   printf("\n5. Round Robin");
   printf("\n6. Exit");
   printf("\nEnter your choice: ");
   scanf("%d", &choice);
   switch (choice) {
      case 1:
         calculateFCFS(p, n, execution_order);
         display(p, n, "FCFS", execution_order);
         break;
      case 2:
         calculateSJF(p, n, execution_order);
         display(p, n, "SJF (Non-Preemptive)", execution_order);
         break;
```

```
            case 3:
                calculateSJFPreemptive(p, n, execution_order);
                display(p, n, "SJF (Preemptive)", execution_order);
                break;
            case 4:
                calculatePriority(p, n, execution_order);
                display(p, n, "Priority Scheduling", execution_order);
                break;
            case 5:
                printf("Enter time quantum for Round Robin: ");
                scanf("%d", &quantum);
                calculateRoundRobin(p, n, quantum, execution_order);
                display(p, n, "Round Robin", execution_order);
                break;
            case 6:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice! Please select again.\n");
        }
    } while (choice != 6);
    return 0;
}
```