```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <pthread.h>

/*This is the definiton for each task to identify info about them*/
struct Task{
    int taskID; //The first number
    int burstTime; //the second number
    int priority; //the third number
    int arrivalTime; //used for Preemptive Priority Scheduling later
};


struct Task tasks[5]; // Array to hold tasks, 5 maximum
int numOfTasks = 0; // Number of tasks counter

/* Function declartion for scheduling algorithms */
void *fcfs(void *param);
void *sjf(void *param);
void *priority_Scheduling(void *param);

/* Function to read tasks from a file */
void readFile(const char *filename) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        perror("Error opening file");
        exit(1);
    }

    // Reading from file using while loop gathering info for each task and
storing it in array of Tasks
    while(fscanf(fp, "%d %d %d",
        &tasks[numOfTasks].taskID,
        &tasks[numOfTasks].priority,
        &tasks[numOfTasks].burstTime ) == 3) {
        tasks[numOfTasks].arrivalTime = 0; // Initializing arrival time to
0
        numOfTasks++;
```

```
    }
    fclose(fp);
}

int main() {
    readFile("input.txt"); //Reading for our input file

    //Creating the threads and attributes for scheduling algorithms
    pthread_t t1, t2, t3;
    pthread_attr_t attr;

    //Initializing thread attributes
    pthread_attr_init(&attr);

    //Creating the threads and making them wait for each other to finish
    pthread_create(&t1, &attr, fcfs, NULL);
    pthread_join(t1, NULL);

    pthread_create(&t2, &attr, sjf, NULL);
    pthread_join(t2, NULL);

    pthread_create(&t3, &attr, priority_Scheduling, NULL);
    pthread_join(t3, NULL);

    return 0;
}

void *fcfs(void *param) {
    int waitingTime[5]; //Holds waiting times for each task
    int turnaroundTime[5]; //Holds turnaround times for each task
    int totalWaiting = 0, TotalTurnaroundTime = 0; //The total overall
from the tasks

    waitingTime[0] = 0; //Waiting time for first task is 0

    //Caculating waiting time for each task afterwards
    for(int i=1; i < numOfTasks; i++) {
        waitingTime[i] = tasks[i-1].burstTime + waitingTime[i-1];
    }
```

```
    //Calculating turnaround time for each task
    for(int i=0; i < numOfTasks; i++) {
        turnaroundTime[i] = tasks[i].burstTime + waitingTime[i];
    }

    //Printing the Gantt Chart
    printf("\n--- FCFS Scheduling ---\n");
    printf("Ganatt Chart: ");

    for(int i=0; i < numOfTasks; i++) {
        printf("[T%d]", tasks[i].taskID); //Printing for each task
    }
    printf("\n"); //New Line

    //Loop through tasks again and find Total Waiting and Turnaround times
    for(int i=0; i<numOfTasks; i++) {
        printf("Task %d: Waiting = %d, Turnaround = %d\n",
tasks[i].taskID, waitingTime[i], turnaroundTime[i]);
        totalWaiting += waitingTime[i]; //Adding the wait times into total
        TotalTurnaroundTime += turnaroundTime[i]; //Adding the turnaround
times into total
    }

    //Calculating average waiting time/turnaround time and rounding
    printf("Average Waiting Time: %.2f\n",
(float)totalWaiting/numOfTasks);
    printf("Average Turnaround Time: %.2f\n",
(float)TotalTurnaroundTime/numOfTasks);
    pthread_exit(0);
}

void *sjf(void *param) {
    int waitingTime[5];
    int turnaroundTime[5];
    int totalWaiting = 0, TotalTurnaroundTime = 0;

    //Array to hold sorted tasks needed for SJF Scheduling
    struct Task sortedTasks[5];
    for(int i=0; i<numOfTasks; i++) {
```

```
        sortedTasks[i] = tasks[i]; //Copying original tasks into sorted
array
    }

    //Sorting the tasks in the sorted array based on burst time using
Bubble Sorting
    for (int i=0; i<numOfTasks-1; i++) {
        for(int j=0; j<numOfTasks-i-1; j++) {
            if(sortedTasks[j].burstTime > sortedTasks[j+1].burstTime) {
                struct Task temp = sortedTasks[j];
                sortedTasks[j] = sortedTasks[j+1];
                sortedTasks[j+1] = temp;
            }
        }
    }

    waitingTime[0] = 0; //Waiting time for first task is 0

    //Caculating the waiting time for each task after the sorting has
happened
    for(int i=1; i < numOfTasks; i++) {
        waitingTime[i] = sortedTasks[i-1].burstTime + waitingTime[i-1];
    }

    //Calculating turnaround time for each task after the sort
    for(int i=0; i < numOfTasks; i++) {
        turnaroundTime[i] = sortedTasks[i].burstTime + waitingTime[i];
    }

    //Printing the Gantt Chart
    printf("\n--- SJF Scheduling ---\n");
    printf("Gantt Chart: ");
    for(int i=0; i < numOfTasks; i++) {
        printf("[T%d]", sortedTasks[i].taskID); //Printing for each task
    }
    printf("\n");

    //Printing the waiting and turnaroundTime
    for(int i=0; i<numOfTasks; i++) {
```

```
        printf("Task %d: Waiting = %d, Turnaround = %d\n",
sortedTasks[i].taskID, waitingTime[i], turnaroundTime[i]);
        totalWaiting += waitingTime[i]; //Adding the wait times into total
        TotalTurnaroundTime += turnaroundTime[i]; //Adding the turnaround
times into total
    }

    //Calculating average waiting time/turnaround time and rounding
    printf("Average Waiting Time: %.2f\n",
(float)totalWaiting/numOfTasks);
    printf("Average Turnaround Time: %.2f\n",
(float)TotalTurnaroundTime/numOfTasks);
    printf("\n");

    pthread_exit(0);
}

void *priority_Scheduling(void *param) {

    //Storing wait times and turnaround times for each task
    int waitingTime[5] = {0};
    int turnaroundTime[5] = {0};

    //Used to track how much burst time is left for each task
    int remainingBurst[5];

    //Checks if the task is completed its run
    int isCompleted[5] = {0};

    int totalWaiting = 0, TotalTurnaroundTime = 0;
    int completedTasks = 0;
    int currentTime = 0;



    srand(time(NULL)); // Makes sure random numbers are random each time

    for(int i=0; i<numOfTasks; i++) {
        tasks[i].arrivalTime = rand() % 101; // Random arrival time
assignments between 0 and 100 ms
```

```
        remainingBurst[i] = tasks[i].burstTime; // Copying the bursts
times for computations

    }


    //Printing the generated arrival times
    printf("Generated Arrival Times: \n");
    for (int i=0; i<numOfTasks; i++) {
        printf("Task %d: Arrival Time = %d ms, Priority: %d, Burst: %d\n",
tasks[i].taskID, tasks[i].arrivalTime, tasks[i].priority,
tasks[i].burstTime);
    }

    printf("\n--- Priority Scheduling ---\n");
    printf("Gantt Chart: ");

    //Running till all tasks are completed
    while(completedTasks < numOfTasks) {
        int current = -1; //The task we are on
        int highestPriority = 200; //Used to track current highest
priority, lower number = highest

        //Finding the task with the highest priority that has arrived
        for(int i=0; i<numOfTasks; i++) {
            if(tasks[i].arrivalTime <= currentTime && !isCompleted[i]) {
                //if the priorrity is higher than our current highest
priority
                if(tasks[i].priority < highestPriority) {
                    highestPriority = tasks[i].priority; //make this our
new highest priority
                    current = i; //current task running
                }
            }
        }


        //If we have a current task to execute
        if(current != -1) {
```

```
            //Increase the waitinf time for the other tasks curerently
arrived
           for(int i=0; i<numOfTasks; i++) {
                if(i != current && tasks[i].arrivalTime <= currentTime &&
!isCompleted[i]) {
                    waitingTime[i]++;
                }
            }

            //Printing the Gantt Chart entry for the current task
           printf("[T%d]", tasks[current].taskID);


            //Executing one time unit
            remainingBurst[current]--;
            currentTime++;


            //If task is finished
            if(remainingBurst[current]==0) {
                isCompleted[current] = 1; //completed
                completedTasks++; //completed counter
                turnaroundTime[current] = currentTime -
tasks[current].arrivalTime;

                totalWaiting += waitingTime[current];
                TotalTurnaroundTime += turnaroundTime[current];
            }
        //If no current task  eady, go on
        }  else {
                currentTime++;
            }
    }

    printf("\n");

    //Printing the waiting and turnaroundTime for each task
    for(int i=0; i<numOfTasks; i++) {
        printf("Task %d: Waiting = %d, Turnaround = %d\n",
tasks[i].taskID, waitingTime[i], turnaroundTime[i]);
```

```
    }

    printf("Average Waiting Time: %.2f\n",
(float)totalWaiting/numOfTasks);
    printf("Average Turnaround Time: %.2f\n",
(float)TotalTurnaroundTime/numOfTasks);

    pthread_exit(0);
}
```