```c
/*
Richard Tubbs PID 6322262
Aaron Anthony PID 6363137
Rosmery Martin PID 6182096
Description:
This program implements two scheduling algorithms:
First-Come First-Served (FCFS): executes processes in the order of their arrival.
Shortest Job First (SJF): schedules processes based on arrival time and burst
time
It reads process details from a file, including process number, arrival time, and
burst time. The selected algorithm is executed based on the command-line argument
provided.

For each process:
We aprocess_numberunce the algorithm requested, and proceed to calculate and
displayed the completion time, turnaround time, and waiting time.

As per assignment requirements:
We conclude by returning the calculated Average turnaround time and Average
waiting time

To run the program:
Provide the filename and the scheduling algorithm as command-line arguments.
The file should contain process details in the specified format.

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure to represent a process
struct Process
{
    int process_number;   // Process number
    int arrival_time;     // Arrival time of the process
    int burst_time;       // Burst time of the process
    int idle_time;        // Idle time
    int completion_time;  // Completion time of the process
    int turn_around_time; // Turnaround time of the process
    int waiting_time;     // Waiting time of the process
};

// Function declarations
void fcfs(struct Process process[], int n);
void sjf(struct Process process[], int n);
```

```c
// Main function
int main(int argc, char *argv[])
{
    struct Process process[50]; // Array to store processes
    int n;                      // Number of processes

    // Check if the correct number of command-line arguments is provided
    if (argc != 3)
    {
        printf("Usage: %s <filename> <algorithm>\n", argv[0]);
        return 1;
    }

    FILE *file = fopen(argv[1], "r"); // Open the file

    // Check if the file was successfully opened
    if (file == NULL)
    {
        printf("Failed to open the file.\n");
        return 1;
    }

    fscanf(file, "%d", &n); // Read the number of processes from the file

    // Read process details from the file
    for (int i = 0; i < n; i++)
    {
        process[i].process_number = i + 1;
        fscanf(file, "%d %d", &process[i].burst_time, &process[i].arrival_time);
    }

    // Determine the scheduling algorithm based on the command-line argument
    if (strcmp(argv[2], "FCFS") == 0)
    {
        printf("FCFS Scheduling Algorithm\n");
        fcfs(process, n);
    }
    else if (strcmp(argv[2], "SJF") == 0)
    {
        printf("SJF Scheduling Algorithm\n");
        sjf(process, n);
    }
    else
    {
```

```
        printf("Invalid algorithm.\n");
    }

    return 0;
}

// Function to implement Shortest Job First (SJF) scheduling algorithm
void sjf(struct Process process[], int n)
{
    int j, min = 0;
    float averageTurnAroundTime = 0, averageWaitingTime = 0;
    struct Process temp_process;
    int process_order[n]; // Array to store the order of execution of processes

    // Sort the processes based on their arrival times in ascending order
    for (int i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - i - 1; j++)
        {
            if (process[j].arrival_time > process[j + 1].arrival_time)
            {
                temp_process = process[j];
                process[j] = process[j + 1];
                process[j + 1] = temp_process;
            }
        }
    }

    // Find the process with the shortest burst time among the processes that
have arrived
    for (j = 1; j < n && process[j].arrival_time == process[0].arrival_time; j++)
    {
        if (process[j].burst_time < process[min].burst_time)
        {
            min = j;
        }
    }

    // Swap the process with the shortest burst time to the front
    temp_process = process[0];
    process[0] = process[min];
    process[min] = temp_process;
    process[0].idle_time = process[0].arrival_time;
    process[0].completion_time = process[0].idle_time + process[0].burst_time;
```

```
    // Schedule the remaining processes
    for (int i = 1; i < n; i++)
    {
        // Find the process with the shortest burst time among the remaining
processes that have arrived
        for (j = i + 1, min = i; j < n && process[j].arrival_time <= process[i -
1].completion_time; j++)
        {
            if (process[j].burst_time < process[min].burst_time)
            {
                min = j;
            }
        }

        // Swap the process with the shortest burst time to its correct position
        temp_process = process[i];
        process[i] = process[min];
        process[min] = temp_process;

        // Calculate the idle time and completion time of the process
        if (process[i].arrival_time <= process[i - 1].completion_time)
            process[i].idle_time = process[i - 1].completion_time;
        else
            process[i].idle_time = process[i].arrival_time;
        process[i].completion_time = process[i].idle_time +
process[i].burst_time;
    }

    // Print the process details and calculate average turnaround time and
waiting time
    printf("\nProcess\t\tArrivalTime\tBurstTime\tCompletionTime\tTurnAroundTime\t
WaitTime\n");
    for (int i = 0; i < n; i++)
    {
        process[i].turn_around_time = process[i].completion_time -
process[i].arrival_time;
        averageTurnAroundTime += process[i].turn_around_time;
        process[i].waiting_time = process[i].turn_around_time -
process[i].burst_time;
        averageWaitingTime += process[i].waiting_time;
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", process[i].process_number,
process[i].arrival_time, process[i].burst_time, process[i].completion_time,
process[i].turn_around_time, process[i].waiting_time);
        process_order[i] = process[i].process_number;
    }
```

```
    // Print the order of execution of processes
    printf("\nProcess Order: ");
    for (int i = 0; i < n; i++)
    {
        printf("P%d", process_order[i]);
        if (i != n - 1)
            printf("->");
    }

    // Print average turnaround time and waiting time
    averageTurnAroundTime /= n, averageWaitingTime /= n;
    printf("\nAverage Turn Around Time (%.2f / %d) = %.2f\n",
averageTurnAroundTime * n, n, averageTurnAroundTime);
    printf("Average Waiting Time (%.2f / %d) = %.2f\n", averageWaitingTime * n,
n, averageWaitingTime);
}

// Function to implement First-Come, First-Served (FCFS) scheduling algorithm
void fcfs(struct Process process[], int n)
{
    struct Process temp_process;
    int completion_time;
    float averageTurnAroundTime = 0, averageWaitingTime = 0;
    int process_order[n]; // Array to store the order of execution of processes

    // Sort the processes based on their arrival times in ascending order
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (process[j].arrival_time > process[j + 1].arrival_time)
            {
                temp_process = process[j];
                process[j] = process[j + 1];
                process[j + 1] = temp_process;
            }
        }
    }

    // Print the process details and calculate completion time, turnaround time,
and waiting time
    printf("\nProcess\t\tArrivalTime\tBurstTime\tCompletionTime\tTurnAroundTime\t
WaitTime\n");
    completion_time = process[0].arrival_time;
```

```
    for (int i = 0; i < n; i++)
    {
        completion_time += process[i].burst_time;
        process[i].completion_time = completion_time;
        process[i].turn_around_time = process[i].completion_time -
process[i].arrival_time;
        averageTurnAroundTime += process[i].turn_around_time;
        process[i].waiting_time = process[i].turn_around_time -
process[i].burst_time;
        averageWaitingTime += process[i].waiting_time;

        printf("P%-2d\t\t%-2d\t\t%-2d\t\t%-2d\t\t%-2d\t\t%-2d\n",
process[i].process_number, process[i].arrival_time, process[i].burst_time,
process[i].completion_time, process[i].turn_around_time,
process[i].waiting_time);
        process_order[i] = process[i].process_number;
    }

    // Print the order of execution of processes
    printf("\nProcess Order: ");
    for (int i = 0; i < n; i++)
    {
        printf("P%d", process_order[i]);
        if (i != n - 1)
            printf("->");
    }

    // Print average turnaround time and waiting time
    averageTurnAroundTime /= n, averageWaitingTime /= n;
    printf("\nAverage Turn Around Time (%.2f / %d) = %.2f\n",
averageTurnAroundTime * n, n, averageTurnAroundTime);
    printf("Average Waiting Time (%.2f / %d) = %.2f\n", averageWaitingTime * n,
n, averageWaitingTime);
}
```