Experiment No.5
Process Management: Scheduling a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS) b. Write a program to demonstrate the concept of preemptive scheduling algorithms (SJF)
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement process scheduling algorithms FCFS and SJF

Objective:

a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS)
b. Write a program to demonstrate the concept of preemptive scheduling algorithms (SJF)

Theory:

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

Jobs are executed on a first come, first serve basis. It is a non-preemptive, preemptive scheduling algorithm. Easy to understand and implement. Its implementation is based on the FIFO queue. Poor in performance as average wait time is high.

Shortest Job First (SJF)

This is also known as the shortest job first, or SJF. This is a non-preemptive, preemptive scheduling algorithm. Best approach to minimize waiting time. Easy to implement in Batch systems where required CPU time is known in advance. Impossible to implement in interactive systems where required CPU time is not known. The processor should know in advance how much time the process will take.

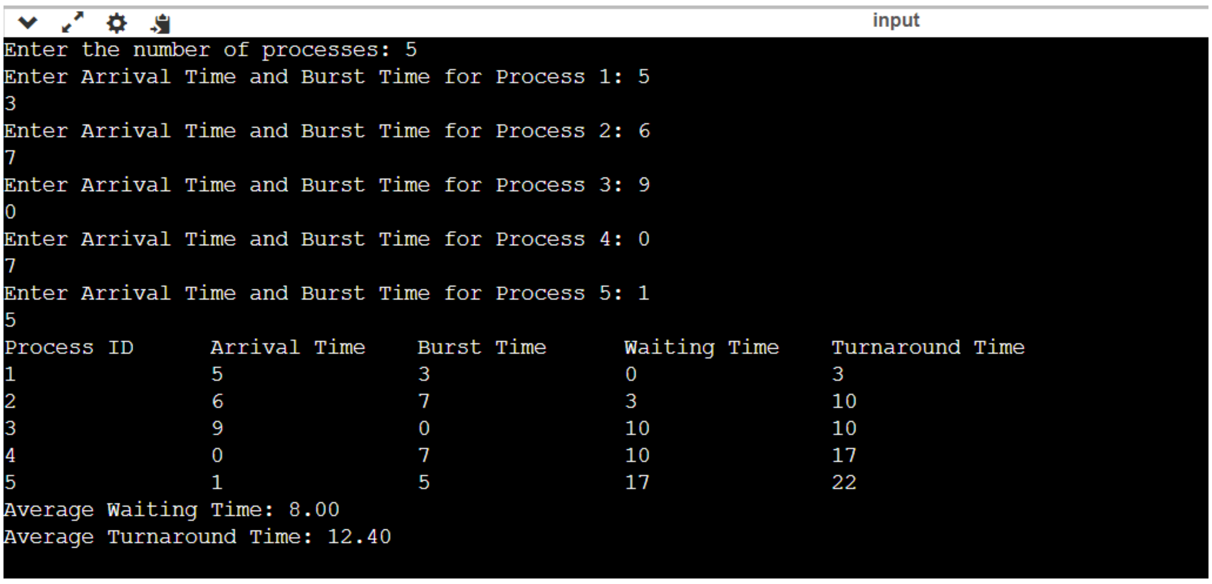
Program 1:

```
#include <stdio.h>
// Process structure
typedef struct {
   int pid; // Process ID
   int arrival_time; // Arrival time
   int burst_time; // Burst time
} Process;

// Function to calculate waiting time for each process
void calculateWaitingTime(Process processes[], int n, int waiting_time[]) {
   waiting_time[0] = 0; // Waiting time for first process is 0

   // Calculate waiting time for each process
   for ( i = 1; i < n; i++) {
         waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
```

```
    }
}

// Function to calculate turnaround time for each process
void calculateTurnaroundTime(Process processes[], int n, int waiting_time[], int turnaround_time[]) {
    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
    }
}

// Function to calculate average waiting and turnaround time
void calculateAverageTime(Process processes[], int n) {
    int waiting_time[n], turnaround_time[n];
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Calculate waiting time for each process
    calculateWaitingTime(processes, n, waiting_time);

    // Calculate turnaround time for each process
    calculateTurnaroundTime(processes, n, waiting_time, turnaround_time);

    // Print results
    printf("Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_waiting_time += waiting_time[i];
        total_turnaround_time += turnaround_time[i];
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival_time,
processes[i].burst_time, waiting_time[i], turnaround_time[i]);
    }

    // Calculate and print average waiting and turnaround time
    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
```

```
}

int main() {
    int n; // Number of processes

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

    Process processes[n]; // Array to store processes

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

    // Perform FCFS scheduling
    calculateAverageTime(processes, n);

    return 0;
}
```

Output:

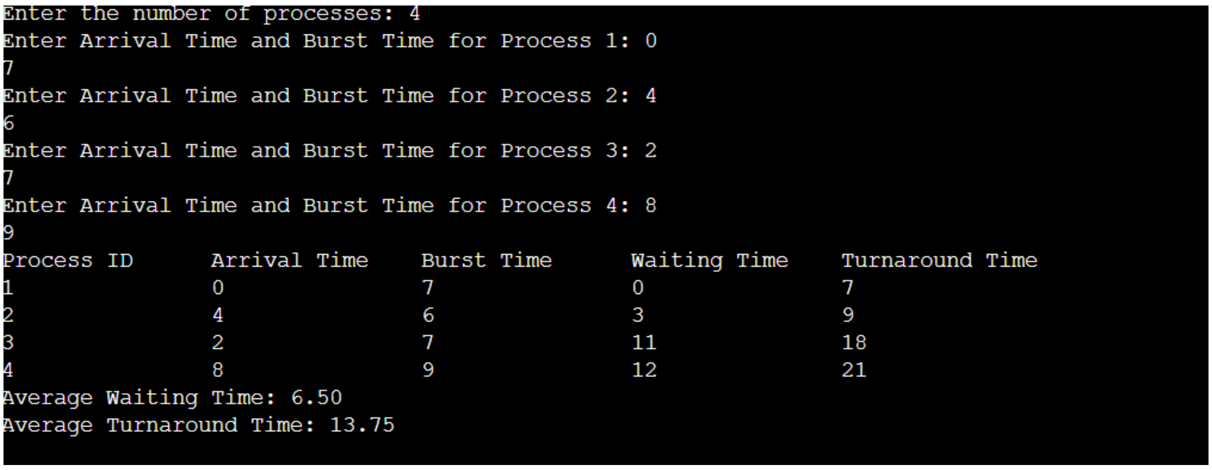


```
Enter the number of processes: 5
Enter Arrival Time and Burst Time for Process 1: 5
3
Enter Arrival Time and Burst Time for Process 2: 6
7
Enter Arrival Time and Burst Time for Process 3: 9
0
Enter Arrival Time and Burst Time for Process 4: 0
7
Enter Arrival Time and Burst Time for Process 5: 1
5
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround Time
1               5               3               0               3
2               6               7               3               10
3               9               0               10              10
4               0               7               10              17
5               1               5               17              22
Average Waiting Time: 8.00
Average Turnaround Time: 12.40
```

Program 2:

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

// Process structure
typedef struct {
    int pid; // Process ID
    int arrival_time; // Arrival time
    int burst_time; // Burst time
    bool completed; // Flag to track if the process is completed
} Process;

// Function to find process with shortest remaining burst time
int findShortestJob(Process processes[], int n, int current_time) {
    int shortest_job_index = -1;
    int shortest_burst_time = __INT_MAX__;

    for (int i = 0; i < n; i++) {
        if (!processes[i].completed && processes[i].arrival_time <= current_time &&
processes[i].burst_time < shortest_burst_time) {
            shortest_burst_time = processes[i].burst_time;
            shortest_job_index = i;
        }
    }

    return shortest_job_index;
}

// Function to calculate waiting time for each process
void calculateWaitingTime(Process processes[], int n, int waiting_time[]) {
    int current_time = 0; // Current time initialized to 0
    while (true) {
        int shortest_job_index = findShortestJob(processes, n, current_time);

        if (shortest_job_index == -1)
            break; // If all processes completed

        // Mark process as completed
        processes[shortest_job_index].completed = true;

        // Update current time
        current_time += processes[shortest_job_index].burst_time;

        // Calculate waiting time for the completed process
        waiting_time[shortest_job_index] = current_time - processes[shortest_job_index].arrival_time -
processes[shortest_job_index].burst_time;
    }
}

// Function to calculate turnaround time for each process
void calculateTurnaroundTime(Process processes[], int n, int waiting_time[], int turnaround_time[]) {
    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = processes[i].burst_time + waiting_time[i];
    }
}

// Function to calculate average waiting and turnaround time
void calculateAverageTime(Process processes[], int n) {
    int waiting_time[n], turnaround_time[n];
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Calculate waiting time for each process
    calculateWaitingTime(processes, n, waiting_time);
    // Calculate turnaround time for each process
    calculateTurnaroundTime(processes, n, waiting_time, turnaround_time);

    // Print results
    printf("Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_waiting_time += waiting_time[i];
        total_turnaround_time += turnaround_time[i];
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival_time,
processes[i].burst_time, waiting_time[i], turnaround_time[i]);
    }

    // Calculate and print average waiting and turnaround time
    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}

int main() {
    int n; // Number of processes

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

    Process processes[n]; // Array to store processes

    // Input process details
    for (int i = 0; i < n; i++) {
        printf("Enter Arrival Time and Burst Time for Process %d: ", i + 1);
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
        processes[i].pid = i + 1;
        processes[i].completed = false;
    }
    // Perform SJF scheduling
    calculateAverageTime(processes, n);


    return 0;
}
```

Output:



```
Enter the number of processes: 4
Enter Arrival Time and Burst Time for Process 1: 0
7
Enter Arrival Time and Burst Time for Process 2: 4
6
Enter Arrival Time and Burst Time for Process 3: 2
7
Enter Arrival Time and Burst Time for Process 4: 8
9
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround Time
1               0               7               0               7
2               4               6               3               9
3               2               7               11              18
4               8               9               12              21
Average Waiting Time: 6.50
Average Turnaround Time: 13.75
```

Conclusion:

What is the difference between Preemptive and Non-Preemptive algorithms?

Preemptive algorithms allow tasks to be interrupted and switched out for higher priority tasks, ensuring responsiveness but potentially causing more overhead. Non-preemptive algorithms complete tasks once started, potentially leading to longer wait times for high-priority tasks but simpler implementation. Preemptive scheduling offers better responsiveness and is common in real-time systems, whereas non-preemptive scheduling may lead to better throughput in certain scenarios but can suffer from longer response times. Preemptive algorithms use mechanisms like time slices or priority levels to determine when to switch tasks, while non-preemptive algorithms rely solely on task completion. Examples of preemptive algorithms include Round Robin and Priority Scheduling, while examples of non-preemptive algorithms include FCFS and SJF.