| |
|---|
| Experiment no. 5 |
| Process Management: Scheduling<br><br>a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS)<br><br>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)**

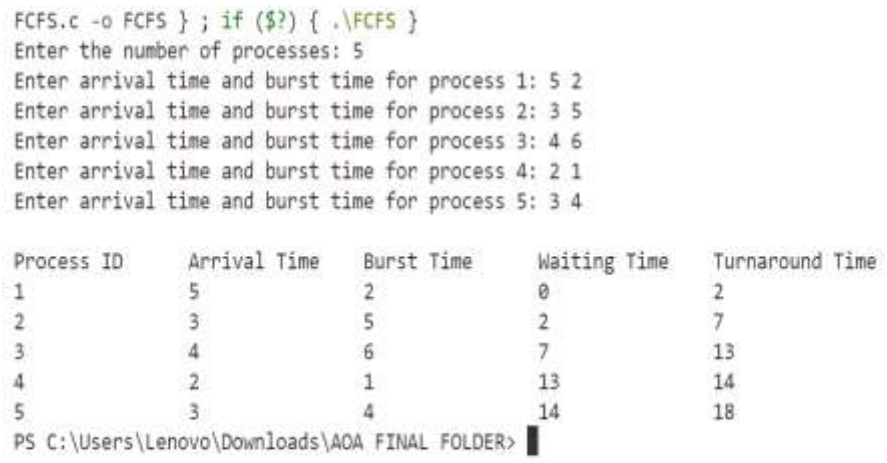
This is also known as the shortest job first, or SJF. This is a non-preemptive, preemptive scheduling algorithm. Best approach to minimize waiting time. Easy to implement in Batch systems where required CPU time is known in advance. Impossible to implement in interactive systems where required CPU time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

Code:

```
#include <stdio.h>

struct Process {
   int pid;
   int arrival_time;
   int burst_time;
};

void fcfs(struct Process p[], int n) {
   int waiting_time = 0, turnaround_time = 0;

   printf("\nProcess ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\n");

   for (int i = 0; i < n; i++) {
      if (i > 0) {
         waiting_time += p[i - 1].burst_time;
         turnaround_time += p[i - 1].burst_time;
      }
      printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].arrival_time,
p[i].burst_time, waiting_time, turnaround_time + p[i].burst_time);
   }
}
```

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

    struct Process p[n];

    for (int i = 0; i < n; i++) {
        printf("Enter arrival time and burst time for process %d: ", i + 1);
        scanf("%d %d", &p[i].arrival_time, &p[i].burst_time);
        p[i].pid = i + 1;
    }

    fcfs(p, n);

    return 0;
}
```

Output:



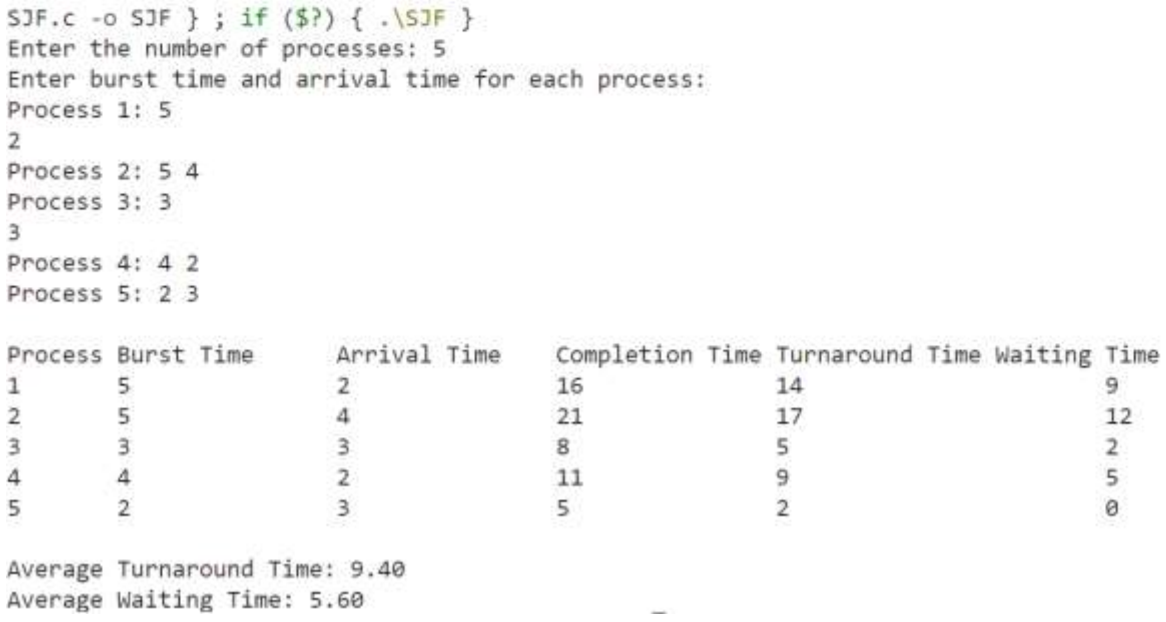
```
FCFS.c -o FCFS } ; if ($?) { .\FCFS }
Enter the number of processes: 5
Enter arrival time and burst time for process 1: 5 2
Enter arrival time and burst time for process 2: 3 5
Enter arrival time and burst time for process 3: 4 6
Enter arrival time and burst time for process 4: 2 1
Enter arrival time and burst time for process 5: 3 4

Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround Time
1               5               2               0               2
2               3               5               2               7
3               4               6               7               13
4               2               1               13              14
5               3               4               14              18
PS C:\Users\Lenovo\Downloads\AOA FINAL FOLDER>
```

SJF Code:

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Structure to represent a process
typedef struct {
    int process_id;
    int burst_time;
    int remaining_time;
    int arrival_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int is_completed;
} Process;

// Function to calculate waiting time, turnaround time, and completion time
void calculateTimes(Process processes[], int n) {
    int current_time = 0;
    int completed_processes = 0;

    while (completed_processes < n) {
        int shortest_burst_index = -1;
        int shortest_burst = __INT_MAX__;

        // Find the process with the shortest remaining burst time
        for (int i = 0; i < n; i++) {
            if (!processes[i].is_completed && processes[i].arrival_time <= current_time
&&
                processes[i].remaining_time < shortest_burst) {
                shortest_burst_index = i;
                shortest_burst = processes[i].remaining_time;
            }
        }

        if (shortest_burst_index == -1) {
            current_time++;
            continue;
        }

        // Execute the process for 1 unit of time
        processes[shortest_burst_index].remaining_time--;
        current_time++;

        // If process is completed
        if (processes[shortest_burst_index].remaining_time == 0) {
            processes[shortest_burst_index].completion_time = current_time;
            processes[shortest_burst_index].turnaround_time =
processes[shortest_burst_index].completion_time -
processes[shortest_burst_index].arrival_time;
            processes[shortest_burst_index].waiting_time =
processes[shortest_burst_index].turnaround_time -
processes[shortest_burst_index].burst_time;
            processes[shortest_burst_index].is_completed = 1;
            completed_processes++;
        }
    }
```

```
}

// Function to calculate average turnaround time and average waiting time
void calculateAverages(Process processes[], int n, float *avg_turnaround_time,
float *avg_waiting_time) {
    int total_turnaround_time = 0;
    int total_waiting_time = 0;

    for (int i = 0; i < n; i++) {
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    *avg_turnaround_time = (float)total_turnaround_time / n;
    *avg_waiting_time = (float)total_waiting_time / n;
}

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

    Process processes[MAX_PROCESSES];
    // Input the burst time and arrival time for each process
    printf("Enter burst time and arrival time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d %d", &processes[i].burst_time, &processes[i].arrival_time);
        processes[i].process_id = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].is_completed = 0;
    }

    // Calculate completion time, turnaround time, and waiting time using SJF
algorithm
    calculateTimes(processes, n);

    // Calculate average turnaround time and average waiting time
    float avg_turnaround_time, avg_waiting_time;
    calculateAverages(processes, n, &avg_turnaround_time, &avg_waiting_time);

    // Display the scheduling results
    printf("\nProcess\tBurst Time\tArrival Time\tCompletion Time\tTurnaround
Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t\t%d\n", processes[i].process_id,
processes[i].burst_time,
            processes[i].arrival_time, processes[i].completion_time,
processes[i].turnaround_time,
            processes[i].waiting_time);
    }

    printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);
    printf("Average Waiting Time: %.2f\n", avg_waiting_time);

    return 0;
}
```

Output:



```
SJF.c -o SJF } ; if ($?) { .\SJF }
Enter the number of processes: 5
Enter burst time and arrival time for each process:
Process 1: 5
2
Process 2: 5 4
Process 3: 3
3
Process 4: 4 2
Process 5: 2 3

Process Burst Time      Arrival Time    Completion Time Turnaround Time Waiting Time
1       5               2               16              14                      9
2       5               4               21              17                      12
3       3               3               8               5                       2
4       4               2               11              9                       5
5       2               3               5               2                       0

Average Turnaround Time: 9.40
Average Waiting Time: 5.60
```

**Conclusion:**

a. In conclusion, the First-Come-First-Serve (FCFS) scheduling algorithm demonstrated in the program showcases a simple yet effective approach to scheduling processes in a non-preemptive manner. By prioritizing the arrival time of processes, FCFS ensures fairness and simplicity in task execution. However, its lack of consideration for process burst times may lead to longer waiting times, especially for processes with higher execution times.

b. In summary, the Shortest Job First (SJF) scheduling algorithm exemplified in the program highlights the efficiency and optimization achieved through preemptive scheduling. By selecting the shortest burst time process for execution, SJF minimizes average waiting and turnaround times, enhancing overall system performance. Despite its effectiveness, SJF may suffer from the issue of starvation for longer processes, as shorter ones continuously preempt them. Overall, preemptive scheduling strategies like SJF offer a balance between responsiveness and efficiency in task scheduling.

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

Preemptive Scheduling:

In preemptive scheduling, the operating system has the ability to interrupt a currently running process and allocate the CPU to another process. This interruption can occur at any time, based on certain criteria such as the expiration of a time slice (quantum), the arrival of a higher-priority process, or the occurrence of an external event.

Preemptive scheduling ensures that no single process monopolizes the CPU for an extended period, which helps in providing fairness and responsiveness to all processes.

Examples of preemptive scheduling algorithms include Round Robin, Shortest Remaining Time First (SRTF), and Priority Scheduling with Aging.

**Non-Preemptive Scheduling:**

In non-preemptive scheduling, once a process starts execution, it continues to run until it voluntarily relinquishes the CPU, blocks on an I/O operation, or completes its execution. The operating system does not forcibly interrupt the process to allocate the CPU to another process.

Non-preemptive scheduling is simpler to implement and may result in less overhead because context switching only occurs when necessary (e.g., when a process completes or blocks).

However, non-preemptive scheduling can lead to potential problems such as poor responsiveness to interactive tasks and the possibility of long-running processes delaying other processes.

Examples of non-preemptive scheduling algorithms include First Come, First Served (FCFS), Shortest Job Next (SJN), and Priority Scheduling without Aging.