Vidyavardhini's College of Engineering and Technology

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
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 :

Non-preemptive Scheduling Algorithm (Shortest Job First):

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

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

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

```
}

void sort_by_arrival_time(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n ; j++) {
            if (processes[j].arrival_time > processes[j + 1].arrival_time) {
                swap(&processes[j], &processes[j + 1]);
            } else if (processes[j].arrival_time == processes[j + 1].arrival_time) {
                // If arrival times are equal, sort by burst time
                if (processes[j].burst_time > processes[j + 1].burst_time) {
                    swap(&processes[j], &processes[j + 1]);
                }
            }
        }
    }
}

void shortest_job_first(Process processes[], int n) {
    sort_by_arrival_time(processes, n);

    int total_waiting_time = 0;
    int completion_time[n];
    completion_time[0] = processes[0].burst_time;
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tCompletion Time\n");
```

```
    printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[0].pid,
processes[0].arrival_time, processes[0].burst_time, 0, completion_time[0]);
    for (int i = 1; i < n; i++) {
        total_waiting_time += completion_time[i - 1] - processes[i].arrival_time;
        completion_time[i] = completion_time[i - 1] + processes[i].burst_time;
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, total_waiting_time,
completion_time[i]);
    }

    double avg_waiting_time = (double) total_waiting_time / n;
    printf("Average Waiting Time: %.2lf\n", avg_waiting_time);
}

int main() {
    Process processes[] = {
        {1, 0, 5},
        {2, 1, 3},
        {3, 2, 8},
        {4, 3, 6}
    };
    int n = sizeof(processes) / sizeof(processes[0]);

    printf("Shortest Job First Scheduling Algorithm (Non-preemptive):\n");
    shortest_job_first(processes, n);
```

```
    return 0;
}
```

Output :

```
Shortest Job First Scheduling Algorithm (Non-preemptive):
Process Arrival Time    Burst Time      Waiting Time    Completion Time
1       0               5               0               5
2       1               3               4               8
3       2               8               10              16
4       3               6               23              22
Average Waiting Time: 5.75
```

Preemptive Scheduling Algorithm (Round Robin):

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

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

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

```
}

void sort_by_arrival_time(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].arrival_time > processes[j + 1].arrival_time) {
                swap(&processes[j], &processes[j + 1]);
            }
        }
    }
}

void round_robin(Process processes[], int n, int time_quantum) {
    sort_by_arrival_time(processes, n);

    int remaining_burst_time[n];
    for (int i = 0; i < n; i++) {
        remaining_burst_time[i] = processes[i].burst_time;
    }

    int t = 0;
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\n");
    while (1) {
        int done = 1;
```

```
        for (int i = 0; i < n; i++) {
            if (remaining_burst_time[i] > 0) {
                done = 0;
                if (remaining_burst_time[i] > time_quantum) {
                    t += time_quantum;
                    remaining_burst_time[i] -= time_quantum;
                } else {
                    t += remaining_burst_time[i];
                    remaining_burst_time[i] = 0;
                    printf("%d\t%d\t\t%d\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, t - processes[i].burst_time -
processes[i].arrival_time);
                }
            }
        }
        if (done == 1) break;
    }

    int total_waiting_time = 0;
    for (int i = 0; i < n; i++) {
        total_waiting_time += t - processes[i].burst_time - processes[i].arrival_time;
    }
    double avg_waiting_time = (double) total_waiting_time / n;
    printf("Average Waiting Time: %.2lf\n", avg_waiting_time);
}
```

```
int main() {
    Process processes[] = {
        {1, 0, 5},
        {2, 1, 3},
        {3, 2, 8},
        {4, 3, 6}
    };
    int n = sizeof(processes) / sizeof(processes[0]);
    int time_quantum = 2;

    printf("Round Robin Scheduling Algorithm (Preemptive):\n");
    round_robin(processes, n, time_quantum);
    return 0;
}
```

Output :

```
Round Robin Scheduling Algorithm (Preemptive):
Process Arrival Time    Burst Time      Waiting Time
2       1               3               7
1       0               5               11
4       3               6               11
3       2               8               12
Average Waiting Time: 15.00
```

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.