KIET Group of Institutions, Ghaziabad

Computer

Science

And

Information

Technology

Subject: Operating System

Project Report

on

CPU Scheduling Algorithms

Submitted By:
Astha Gupta(2100290110038)
Himanshi Tyagi(2100290110064)
Ishan Sharma(2100290110065)

INDEX

ACKNOWLEDGEMENT

It gives us immense pleasure to express our deep sense of gratitude to **Mr.Vinay Kumar**, Assistant Professor of Department of Computer Science & Information Technology, whose constant encouragement and valuable guidance havebeen the source of inspiration in the completion of this work. Without his sustained interest and sound counsel, this work would have not gained the present form.We are equally thankful to all teaching faculty and ever willing supporting staff members and students of Computer Science & Information Technology , **KIET Group Of Institutions , Ghaziabad**. We are also thankful to my family for all their love and affection that helped us to fare well in my task. We thank all those who directly or indirectly helped us during this work.

Astha Gupta (2100290110038)

Himanshi Tyagi(2100290110064)

Ishan Sharma(2100290110065)

ABSTRACT

The most important aspect of job scheduling is the ability to create a multi-tasking environment. A single user can not keep either the CPU,or the VO. devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has something to execute. To have several jobs ready to run, the system must keep all of them in memory at the same time for their selection one-by- on. This work involves development of a simulator for CPU scheduling. It has been developed as a comprehensive tool which runs a simulation in real time, and generates useful data to be used for evaluation. A user-friendly and mouse- driven graphical user interface has been integrated. The system has been put through extensive experimentation. The evaluation results extremely useful for the design and development of modem operating systems. This simulator can be used for measuring performance of different scheduling algorithms and for the understanding and training of students.

INTRODUCTION

Computer systems supporting multiprogramming or multitasking execute multiple programsltasks concurrently. One of the objectives of multiprogramming is to maximize resource utilization which is achieved by sharing system resources amongst multiple user and system processes. Efficient resource sharing depends on efficient scheduling of competing processes. As processor is the most important resource, CPU scheduling becomes very important in achieving the system design goals. Many algorithms have been designed to implement CPU scheduling. Design methods include analytic modeling, deterministic modeling and simulations. Simulation being the most accurate of all is commonly employed for system's performance evaluation despite the fact that it requires complex programming for developing an efficient simulator. This work involves development of a simulator. for CPU scheduling. This simulator can be used for measuring performance of different scheduling algorithms. It simulates:

(1) First Come First Serve (FCFS) scheduling,

(2) Shortest Job First (SJF) scheduling,

(3) priority scheduling, and

(4) Round Robin (RR) scheduling.

REQUIREMENT ANALYSIS

SOFTWARE REQUIREMENT

The following software is needed in order to make the model work efficiently:

Codeblocks

HARDWARE REQUIREMENTT

The following hardware is needed in order to make the model work efficiently:

A computer

DESIGN METHODOLOGY

The system is designed to mimic the dynamic behavior of the system over time. It runs self-driven simulations and uses a synthetic workload that is artificially generated to resemble the expected conditions in the modeled system. This system can also be used to run a trace-driven simulation. Duration of CPU and I/O bursts is read from a data file. The system simulates creation, concurrent execution and termination of processes. It simulates shared utilization of processor, disk drive, printer and magnetic tape drive. It maintains system queues in memory as singly linked lists. These queues include: ready queue, disk drive queue, printer queue and magnetic tape drive queue. It generates useful data that represents the behavior of the system. The data is recorded in a data file for analysis and archival purposes.

FCFS is the simplest scheduling strategy discipline. The workload is simply processed in the order of arrival. Implementation of the FCFS scheduler is quite straightforward. It can be easily managed with a First In First Out (FIFO) queue. When a process enters the ready queue, its Process Control Block (PCB) is linked onto the tail of the ready queue. When CPU is free, it is allocated to the process at the head of the ready queue. However, it dose not take into consideration the state of the system and the resource requirements of the individual scheduling entities, because of it this scheduler may result in poor performance, lower throughput and longer average waiting times. Short jobs may suffer considerable turnaround delays and waiting times when one or more long jobs are in the system .

SJF is a scheduling discipline in which the next scheduling entity, a process, is selected on the basis of the shortest execution time of its next CPU burst. Whenever the SJF scheduler is invoked, it searches the ready queue to find the process with the shortest next execution time. SJF scheduler is invoked whenever a process completes execution or the running process surrenders control to the operating system.

The RR scheduling algorithm, which is also known as time slice scheduling, is designed especially for time sharing systems. The CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval of up to one time slice or time quantum. The ready queue which is treated as a circular queue is kept as First In First Out (FIFO) queue of processes. New processes are added at the tail of the ready queue and CPU scheduler picks the process from the head of the queue. After expiry of the time quantum the process is preempted and placed at the tail of the ready queue [1,2,3]. In real systems, however, the Context switching would take place on expiry of each time slice. RR scheduling achieves equitable sharing of system resources. Short processes may be executed within a single time quantum and thus exhibit good response time. Long processes may require several quantum and thus be forced to cycle through the ready queue for a longer time before completion.

In priority scheduling, each process in the system is assigned a priority level and the scheduler chooses the highest priority process from the ready queue. Equal priority processes are scheduled in FCFS order. In this case there is a possibility that low-priority processes be effectively locked out by the higher priority ones. In other words completion of a process within finite time of its creation cannot be guaranteed with this scheduling policy. In this scheme, however, the typical performance criteria are traded with the priority objectives. In other words the priority criteria determines the performance criteria. The order of completion of jobs is determined by the way the jobs are selected by the scheduler.

CODING AND IMPLEMENTATION

```
#include <stdio.h>

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

void fcfs_scheduling(struct Process processes[], int n) {
    int current_time = 0;
    int total_waiting_time = 0;
    printf("FCFS Scheduling:\n");

    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time > current_time) {
            current_time = processes[i].arrival_time;
        }
        int waiting_time = current_time - processes[i].arrival_time;
        total_waiting_time += waiting_time;
        printf("Process ID: %d Waiting Time: %d\n", processes[i].process_id, waiting_time);
        current_time += processes[i].burst_time;
    }

    float average_waiting_time = (float)total_waiting_time / n;
    printf("Average Waiting Time: %.2f\n\n", average_waiting_time);
```

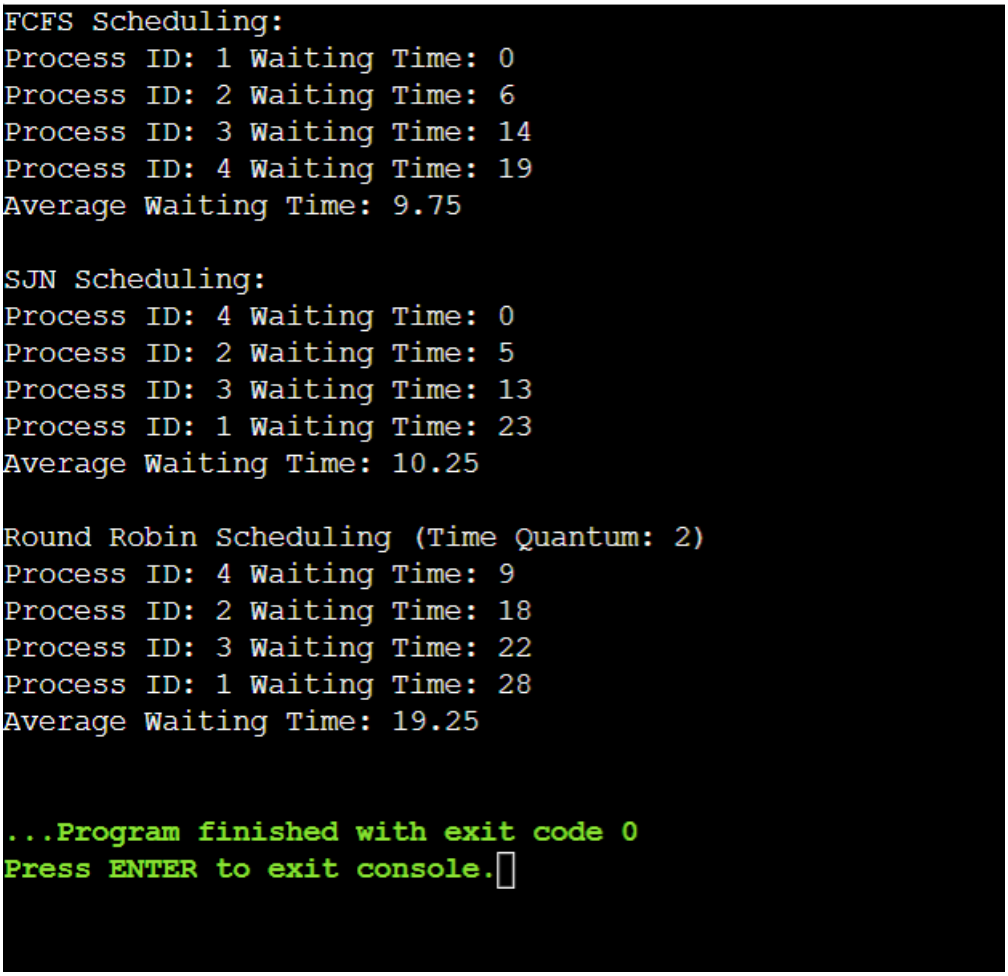
```
}

void sjn_scheduling(struct Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].burst_time > processes[j].burst_time) {
                struct Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    int current_time = 0;
    int total_waiting_time = 0;
    printf("SJN Scheduling:\n");

    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time > current_time) {
            current_time = processes[i].arrival_time;
        }
        int waiting_time = current_time - processes[i].arrival_time;
        total_waiting_time += waiting_time;
        printf("Process ID: %d Waiting Time: %d\n", processes[i].process_id, waiting_time);
        current_time += processes[i].burst_time;
    }
```

```
    float average_waiting_time = (float)total_waiting_time / n;
    printf("Average Waiting Time: %.2f\n\n", average_waiting_time);
}

void round_robin_scheduling(struct Process processes[], int n, int time_quantum) {
    int current_time = 0;
    int total_waiting_time = 0;
    printf("Round Robin Scheduling (Time Quantum: %d)\n", time_quantum);
    int remaining_burst_time[n];

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

    while (1) {
        int all_processes_completed = 1;

        for (int i = 0; i < n; i++) {
            if (remaining_burst_time[i] > 0) {
                all_processes_completed = 0;

                if (processes[i].arrival_time <= current_time) {
                    if (remaining_burst_time[i] > time_quantum) {
                        remaining_burst_time[i] -= time_quantum;
                        current_time += time_quantum;
                    }
                    else {
```

```
                int waiting_time = current_time - processes[i].arrival_time;
                total_waiting_time += waiting_time;
                current_time += remaining_burst_time[i];
                remaining_burst_time[i] = 0;
                printf("Process ID: %d Waiting Time: %d\n", processes[i].process_id, waiting_time);
            }
        }
        else {
            current_time++;
        }
    }
    if (all_processes_completed) {
        break;
    }
  }
  float average_waiting_time = (float)total_waiting_time / n;
  printf("Average Waiting Time: %.2f\n", average_waiting_time);
}
int main() {
  struct Process processes[] = {
    {1, 0, 10},
    {2, 4, 6},
    {3, 2, 8},
    {4, 5, 4}
  };
```

```
    int n = sizeof(processes) / sizeof(processes[0]);
    int time_quantum = 2;

    fcfs_scheduling(processes, n);
    sjn_scheduling(processes, n);
    round_robin_scheduling(processes, n, time_quantum);
    return 0;
}
```

Output:



RESULT/ CONCLUSION

CPU scheduling algorithms maximise utilization of the CPU and minimise resource idle time. Results have shown that the execution of FCFS scheduling produce smaller computational overheads because of its simplicity, but it gives poor performance, lower throughput and longer average waiting times. SJF is an optimal scheduling discipline in terms of minimizing the average waiting time of a given workload. However, preferred treatment of short processes in SJF scheduling tends to result in increased waiting times for long processes in comparison with FCFS scheduling. Thus, there is a possibility that long processes may get stranded in be ready queue because of continuous arrival of shorter processes in the queue. RR scheduling achieves equitable sharing of CPU. Short processes execute within a single time quantum and thus exhibit good response time. It subjects long processes to relatively longer turnaround and waiting times. In case of priority-based scheduling there is a possibility that low-priority processes be effectively locked out by the higher priority ones. In other words completion of a process within finite time of its creation cannot be guaranteed with this scheduling policy. Experimental results are extremely useful for the design and development of modern operating systems.

REFERENCE

1. https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/
2. https://youtu.be/EWkQl0n0w5M