Table of Contents

Introduction – Page 2

Program Overview – Page 3

Data and Analysis – Page 5

Raw Program Output – Appendix I, CPUScheduler\_Output.txt

Source Code – Appendix II, Chetmichals\_CPUScheduler.cpp

Executable – Appendix III, CPUScheduler.exe

Introduction

In this report, we will go over the results of a CPU scheduling simulation, showing the differences in the CPU utilization, turnaround times, waiting times, and response times of three different CPU scheduling Algorithms. The Algorithms are as follows:

* **First Come First Serve**, FCFS. The process waiting on CPU execution are placed into a First In First Out Queue, simply executing the process at the top of the queue in order until the queue is empty.
* **Shortest Job First**, SJF. Traditionally the CPU would try and estimate the burst time of processes to be executed, but in this simulation the exact burst time is known. Thus, the process in waiting are treated as a list, where the element that has the lowest execution will be the one moved to execution. In the event of a tie between execution time, the element that is highest on the list will be moved to execution, since the list is naturally ordered from longest waiting to shortest waiting.
* **Multi-Level Feedback Queue**, MLFQ. Processes would be distributed between three queues of various priorities, where processes in a higher level queue will always execute before those in a lower level queue. In each individual priority queue, the processes are choices by round robin. By default, all process will be placed in the highest priority queue, queue 1, and then if taking too long to execute will be bumped off and moved into a lower level queue. Processes in priority queue 1 will have up to 6 time units to execute before being moved down into priority level 2, and processes in queue 2 will have 12 time units to execute before being bumped down to priority level 3. Process that are in priority level 3 will have unlimited time to execute. In addition, if any higher priority process become ready for execution, the lower priority process will be bumped off of execution and to the end of its respected queue.

Program Overview

The simulation makes use of a fairly simply while loop, that continues to simulate each CPU cycle until all process have been completed. Below is the original pseudocode and an overview of the planned data structure used to structure the individual algorithms of the program:

Put all Processes into ready queue

while wait queue and ready queue are not empty

{

if execution is empty pop top process off ready queue, put it into execution box // for SJF, figure out which job is first

// For MLFQ, maintain 3 queues, and check each queue for an entry to pop off.

advance one time unit

decrement waiting time in IO queue

check if anything in IO queue should move to Ready Queue.

evaluate if process in execution box should be moved to finished, IO, or ready

}

vector ReadyQueue

Vector IOQueue

Vector FinishedList

Varrable ExecutionBox

struct ProcessInfo

{

int processID;

int burstNumber;

int executionTime;

int turnaroundTime;

int responseTime;

bool responceFlag

}

The final program for the individual algorithms flow came very close to the original draft. For comparison, here is the function that drives the simulation of FCFS, seen on the next page:

void FCFS\_Modular()

{

    //Put all Processes into ready queue and inizile all the global varables

    setProcessTimes();

    InitilzeProcessInfo();

    vector <int> ioDequeueList;

    while (ReadyQueue.size() != 0 || IOQueue.size() != 0 || inExecution > 0)

    {

//if execution is empty pop top process off ready queue, put it into execution box

        MoveToExectuion\_FCFS();

CPUCycle++; //advances one time unit.

//Add a cycle to the wait time of all process in the ready queue

for (int i = 0; i < (int)ReadyQueue.size(); i++)

ProcessInfo[ReadyQueue[i]].waitTime++;

//Used to print info when a process leaves

//the exectuion queue without something replacing it

        if (inExecution < 0)

        {

            if (inExecution == - 1)PrintFrame(inExecution);

            idleTime++;

            inExecution--;

        }

//Updates the IO process times, and removes

//items for the IOqueue when they finish.

        UpdateIO(ioDequeueList);

//Evalute if the process in execution

//should be moved to finished or IO.

        CheckExecution();

//Sorts the list of finished processes to line up with the examples slightly better

sort(FinishedList.begin(),FinishedList.end());

    }

}

Data and Analysis

In this section, I will be showing the results of the CPU simulation as a series of tables, and giving a brief analysis of each.

Table 1, CPU Utilization

|  |  |  |  |
| --- | --- | --- | --- |
|  | CPU Cycles Total | Idle Cycles | CPU Utilization |
| FCFS | 1002 | 166 | 83.43% |
| SJF | 939 | 103 | 89.03% |
| MLFQ | 925 | 89 | 90.38% |

From table 1 we can see that MLFQ has the highest utilization with the least amount of idle cycles. Following closely is SJF, which is slightly less efficient, and finally is FCFS, which has the most wasted cycles of any of them. It seems the main reason FCFS has its significantly higher amount of idle cycles is due to not having as many processes on the IO queue as possible.

Table 2, Turnaround Time

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | Average |
| FCFS | 803 | 773 | 692 | 834 | 1002 | 665 | 702 | 788 | 807 | 785.1 |
| SJF | 823 | 519 | 922 | 939 | 678 | 483 | 278 | 472 | 357 | 607.9 |
| MLFQ | 894 | 576 | 799 | 925 | 637 | 723 | 365 | 787 | 381 | 676.3 |

Table 2 shows the turnaround times of all the processes. It seems that SJF has the faster turnaround, due to being able to treat processes with equal priority, while MLFQ suffers from mild starvation, where processes that have a few high bursts will not be processed until the very end of the simulation. FCFS has the highest due not being able to prioritize process at all, since longer processes will significantly delay the execution of shorter process.

Table 3, Response Time

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | Average |
| FCFS | 0 | 17 | 27 | 45 | 62 | 67 | 77 | 83 | 92 | 52.2 |
| SJF | 57 | 26 | 299 | 141 | 0 | 43 | 5 | 17 | 11 | 66.6 |
| MLFQ | 0 | 6 | 12 | 18 | 24 | 29 | 35 | 41 | 47 | 23.6 |

Table 3 shows the response times of the processes. As we can see, MLFQ giving all processes only 6 time units to execute their first burst results in the faster response time of the three. After that FCFS is second, since it doesn’t prioritize processes at all the processes will get a chance to execute their first burst before a one gets to execute their second burst. SJF results in the slowest Response Time, due to long burst processes getting starved out to short one.

Table 4, Wait Time

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | Average |
| FCFS | 347 | 392 | 282 | 297 | 391 | 368 | 479 | 429 | 511 | 388.4 |
| SJF | 367 | 138 | 512 | 402 | 67 | 186 | 55 | 113 | 61 | 211.2 |
| MLFQ | 438 | 195 | 389 | 388 | 26 | 426 | 142 | 428 | 85 | 279.7 |

Table 4 shows us the total wait times of the processes. As we can see, SJF while having a high response time, has the lowest average wait time of the 3 run algorithms. After that is MLFQ, which while it does have minimal downtime on high priority tasks, has a longer wait time compared to SJF. Last if FCFS, which due to having nothing to prioritize processes at all, has the longest wait time on average.