

# OPERATING SYSTEMS

## Process Scheduler Report



**Joel Brigida**

**Prof. Borko Furht**

**COP4610-002**

**22 October 2021**

Image source: <https://discover.hubpages.com/technology/Operating-System-basic-structure>

# TABLE OF CONTENTS

<b>INTRODUCTION.....</b>	<b>1</b>
General Flow Chart Logic of the Simulation Program.....	9
Presentation Screenshots of Final Results of Scheduling Algorithms.....	10
Discussion of Final Results of Schedulers.....	12
Bibliography .....	15
Samples of Dynamic Execution Program Output: FCFS .....	16
Samples of Dynamic Execution Program Output: SJF.....	46
Samples of Dynamic Execution Program Output: MLFQ.....	75

## Introduction

In this programming assignment, I implement three types of process scheduling algorithms using C++ language. I chose to implement the algorithms using singly linked queues of dynamically allocated memory with a front and back pointer. The Integrated Development Environment I chose to use for this process is open-source Code::Blocks version 20.03. The three algorithms implemented are: First Come First Served, or FCFS, Shortest Job First, or SJF, and Multi Level Feedback Queue, or MLFQ. The simplest but least effective is First Come First Served. This is a very easily implemented algorithm with a single Ready List queue, where each process enters the rear of the queue, waits, and is transferred to the running state in the same order it entered the list after it reaches the front of the Ready List. Although it is a simple algorithm, it has the longest process wait time of all, so it is generally avoided as a primary scheduling technique. It is useful as a low priority queue, for example with the MLFQ algorithm as will be discussed, and as a worst-case benchmark. Any process scheduler which cannot outperform FCFS is unacceptable for multitasking, since FCFS will result in the longest average waiting times for all processes. This result is demonstrated with the FCFS ( ) menu option of the project. The second algorithm is Shortest Job First, or SJF, which is implemented in real world systems using an Approximate Shortest Job First, or ASJF, prediction function. True SJF, not ASJF, results in the shortest average process waiting times, but cannot be effectively implemented because future CPU burst times are unknown and only approximated. It is, however, useful as another benchmark. If all processes are analyzed after completion from when they are new to when they are terminated, then placed in the Ready List in order of shortest CPU burst time in the front, and longest in the back, SJF can show us a best-case scenario of average process wait times, which is the exact case my process scheduler program project simulates in

the `SJF()` function. The third type of process scheduler emulated in this project is what most current operating systems use today: a Multilevel Feedback Queue, or MLFQ, scheduling algorithm. With this scheduler, there are three Ready List queues, each with different priority levels, where higher priority queues get less CPU time than lower priority queues. All processes sent to the Ready List initially or coming back from I/O queue enter the highest priority `Ready_1` queue first and get the same 5 units of CPU time in a Round Robin type of scheduling algorithm. If the process does not finish during that time, it is sent to the second `Ready_2` queue which has a lower priority than the first, but a 10-unit CPU burst time allowance with a Round Robin type of scheduling. The processes in `Ready_2` must wait until `Ready_1` is empty to get their CPU time. If the process still has not completed its CPU burst in that time, it gets sent to the third `Ready_3` queue with the lowest priority, which runs the FCFS algorithm without a CPU time limit. Processes in the third queue do not run until the first two queues are empty. If a process node leaves the `IO_wait` queue during execution of a process in the `Ready_2` list, the process leaving `IO_wait` is placed into the `Ready_1` queue, and the `Ready_2` queue process is preempted so the higher priority process can run immediately. This is the same situation if the process at the front of `Ready_3` is in the running state and a process leaves `IO_wait` to enter the `Ready_1` queue.

Since SJF is the best case for average process wait times, and FCFS is the worst case for average process wait time, I expect the MLFQ algorithm to perform in the middle of these two.

When my program is launched, it has a menu for selection. This is the `main()` function of the program. There are four options in the menu, the first three run one of the algorithms: FCFS, SJF, or MLFQ. The fourth option exits the program.

My algorithm consists of 4 files: `processes.h`, `processes.cpp`, `queues.cpp`, and `sched_driver.cpp`.

`Sched_driver.cpp` is the driver file for the program. In this file, all objects are declared and initialized, and is where the main `while` loop for program execution resides. This file also calls the function to print results at the end of each algorithm simulation. This file also has non-class functions that contain the specific object declarations, while loop, and output statements for each algorithm named `FCFS()`, `SJF()`, and `MLFQ()`.

`Processes.h` is the header file which defines the `queue` struct, the lowest level data structure, which comprises all applicable Ready Lists and I/O lists, and the `Process` and `node` structs. For FCFS and SJF, there are only two `queue` structs needed: `Ready` and `IO_wait`. For MLFQ, there are four `queue` structs needed: `Ready_1`, `Ready_2`, `Ready_3`, and `IO_wait`. Each `queue`, `Process`, and `node` defined in `processes.h` keeps track of its own required variables to produce the output at the end of each algorithm.

Each Ready List is a `queue` struct and keeps track of a `char` `algorithm` which is `'F'` for FCFS, `'S'` for SJF, or `'R'` for Round Robin depending on which algorithm the `queue` implements. `'R'` is used in MLFQ for `Ready_1` and `Ready_2` lists only, and in these cases there is an `RR_time` variable used as part of those objects which counts down during process execution. `nodes_remaining` is the total number of nodes left in each `queue` before it is empty, `total_nodes` is the total number of `nodes` processed by the `queue` which only counts up, and `CPU_total_time` is the total number of CPU time bursts carried out. `CPU_idle_time` increments when ready queues are empty, at which time the `CPU_total_time` stops incrementing. Note that `CPU_total_time + CPU_idle_time = Global_clock`. Also note that context switch is excluded from the

`Global_clock` on both I/O and Ready Lists to keep them both in sync. This caused a problem initially, where I/O was still incrementing during context switch, so the processes were all arriving back to the Ready List from I/O earlier than the model on paper says they should. Also note that `IO_total_time` also does not count during context switch, and only increments when the `IO_wait` queue is `!Empty()`. The variable `Global_clock` is stored in each `queue` struct object to keep the overall time count of the running algorithm. It is redundant for all `queue` struct objects to make sure they are all counting in sync with each other for an accurate time count. There is a `string queue_name` which is the same string as the `proc_name` in the `Process` struct and `name` in the `node` struct, which gets transferred from `Process` to `queue` to `queue` as needed from start to finish to keep track of all node transfers, which aids in debugging. Finally, the `queue` struct has pointers to the front and back: `node *front` and `node *back`.

The `Process` struct defines each of the eight processes `P1 - P8`, which begin the simulation as their own linked lists storing each of the `node` structs which are loaded manually for the first CPU bursts from the `front` of the `Process` structs before the `while` loop in `main()` to the `Ready_1` queue. At that point, the `while` loop in `main()` carries out the program until all the processes `P1 - P8` are empty, the loop condition becomes false and exits, then prints all results of the current algorithm to the screen. The destructors are then called, the program halts and asks for user input if you would like to run any of the three algorithms at the menu: `FCFS()`, `SJF()`, or `MLFQ()`. The `Process` struct has fewer variables it must record. The variable `int total_nodes` is used to record the total amount of `node` objects created in each `Process` in the beginning of the program and should not change, `nodes_remaining` is used as a countdown for debugging the number of nodes left yet to get

CPU burst time in each `Process` struct. Each process `P1 - P8` tracks its own `total_wait_time`, which accumulates node by node in each `node` struct. The Ready List(s) increment each `node` wait time individually, and when the `node` finishes its CPU burst and is either sent to I/O or sent to a lower priority Ready List, the accumulated time for that `node` is added to the `Process` before the `node` is subjected to delete. This process could likely be optimized. The `time_response` and `time_finished` is recorded in each `Process` struct by special exception of the counters. The first `wait_time = 0` for each `Process` struct triggers the storing of the `time_response` variable, and the `Empty()` condition of each `Process` struct triggers storing the `time_finished` variable. These two variables are also used to calculate the turnaround time at the end of every algorithm. The string `proc_name` is the same string shared by the struct `node` and `Process` `node` for debug purposes. They are all named `P1 - P8` to track them as they are transferred from list to list. Finally, the `Process` struct has pointers to the front and back of the list: `node *front` and `node *back`. I would also like to simplify the `Process` creation, as well. Instead of one `Process` per file input like my arrangement, there should be a single file input, which parses the string, for example:

```
P1 5 27 3 31 5 43 4 18 6 22 4 26 3 24 5
P2 4 48 5 44 7 42 12 37 9 76 4 41 9 31 7 43 8
```

Finally, there is the struct `node` that is transferred across each queue object during program execution. The `time_q` variable in each `node` is what counts down at the front of the Ready List simulating a process in the running state. The `wait_time` variable is incremented in every process behind the running process waiting in any Ready List, and upon `time_q` expiration, is transferred to the appropriate `Process` `P1 - P8`. This also has room for

optimization. When a node is transferred to another queue, it is copied from old queue to new queue, then deleted from the old queue, resulting in a temporary doubling of that particular node memory space allocation rather than an actual handoff or transfer. Each node has a pointer to node `*next`, a string name from P1 - P8 used for tracking purposes, and a char `burst_type` to verify it ends up in the correct queue during transfer: 'C' for CPU burst or 'I' for I/O burst, which are initialized when each `Process` is initially created in the `main()` with alternating CPU Burst Type Char 'C' -> 'I' -> ... -> 'C'. The `burst_type` variable in the node proved useful in the tracking of nodes across program execution. There were numerous occasions during debug where the wrong node was placed and / or deleted from the incorrect queue and had to be corrected for proper program output.

`Processes.cpp` is the implementation file for all `Process` structures P1 - P8. These are basic functions including an explicit value constructor, a destructor, as well as support functions to implement those. The two most important functions of this file are `Make_process()`, which creates linked lists out of empty `Process` objects and fills them with node objects. This is accomplished by reading an input file containing the integer numbers for each burst time for each process. There are eight files `p1.txt` - `p8.txt` in the source code folder, and each one is read out to construct the corresponding `Process`. The other important function in this implementation file is the `Print_proc_nodes()` function, used for debug to print the remaining nodes in each `Process` when needed.

`Queues.cpp` is the implementation file for all `queue` class functions. Most functions in this class take `queue` objects and `Process` objects as arguments. I had to split up all functions into separate groups to preserve proper output for each algorithm. The MLFQ functions are very different from the FCFS and SJF functions but accomplish the same goal. The

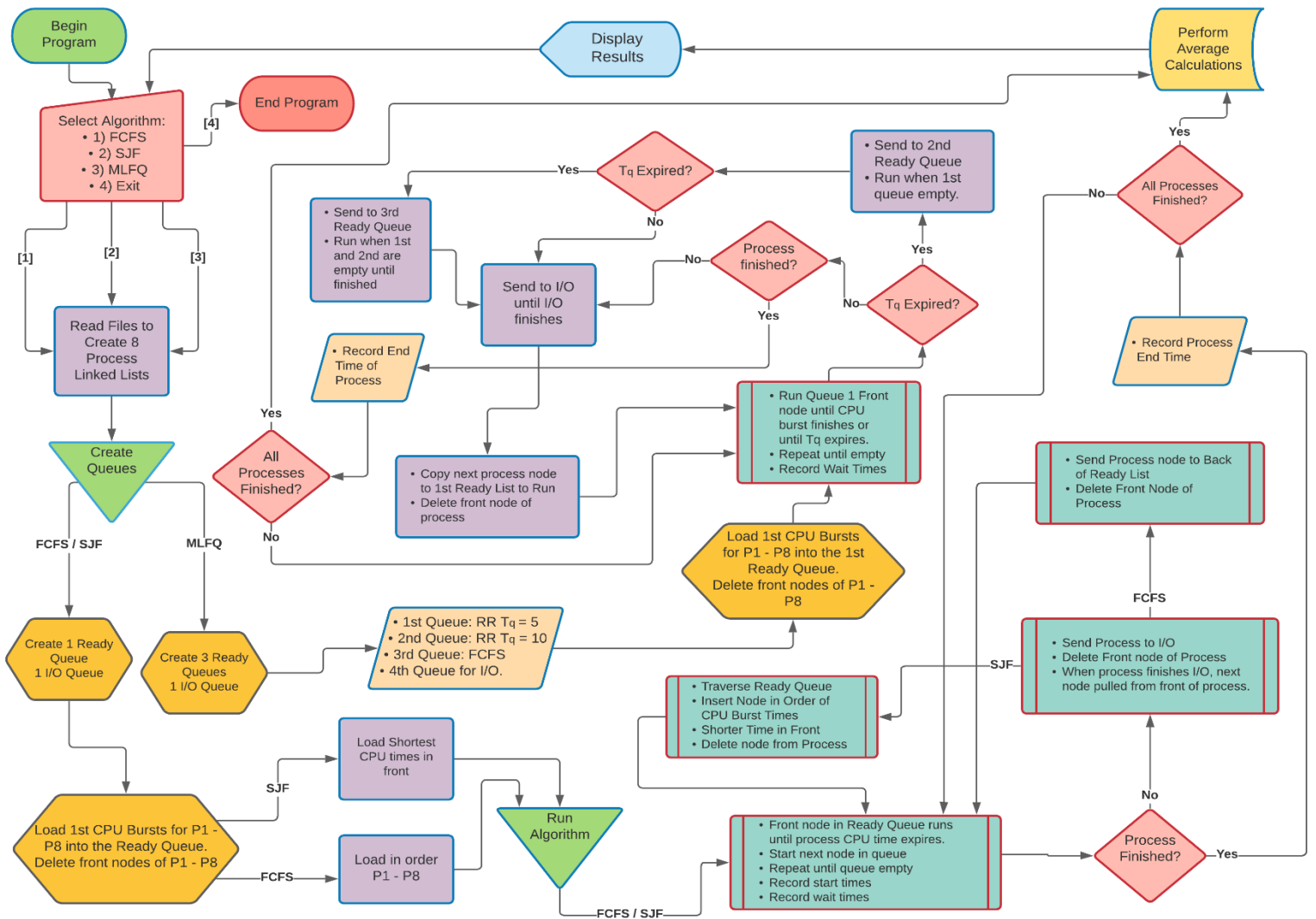


`remove()` function is common to all and used to remove a node from any position in a queue object, such as a process which ends its I/O burst in the middle of the `IO_wait` queue. `FCFS_SJF_increment_wait()` and `MLFQ_increment_wait()` are responsible for traversing the Ready Lists of the appropriate algorithms and incrementing the `wait_time` variable of each node in the list. The `FCFS_SJF_CPU_timer()` and `MLFQ_CPU_timer()` functions are called after every iteration of the `while` loop inside the `main()` function. This function is responsible for incrementing the `Global_clock`, `CPU_idle_time`, `CPU_total_time`, decrementing the `time_q` variable of each node, and in the case of MLFQ it decrements the `RR_time` of the `Ready_1` and `Ready_2` queues where applicable, and triggers the context switch between queues by calling either `FCFS_SFJ_add_to_queue()` or `MLFQ_add_to_queue()` depending on the selected algorithm. The `FCFS_SJF_IO_queue_timer()` and `MLFQ_IO_queue_timer()` functions are responsible for traversing the `IO_wait` queues for the selected algorithm and decrementing the `time_q` variable in every node as it sits in I/O. The function is also responsible for incrementing the `IO_total_time` variable, which only increments when there are node objects in the `IO_wait` queue and triggers the context switch from `IO_wait` to `Ready` or `Ready_1` depending on the selected algorithm. `Print_ready_IO()` is a debug function which is also called at every iteration inside the `while` loop in the `main()` function. It prints the contents of each Ready List and the `IO_wait` list to ensure the proper context switch is carried out. The `Results()` function is called from the `main()` function after the `while` loop condition becomes false and the loop exits. This function displays all data collected from running the selected simulation. The only function unique to only MLFQ scheduling is the `MLFQ_add_to_ready1()` which I needed to load the initial `Process` nodes into the

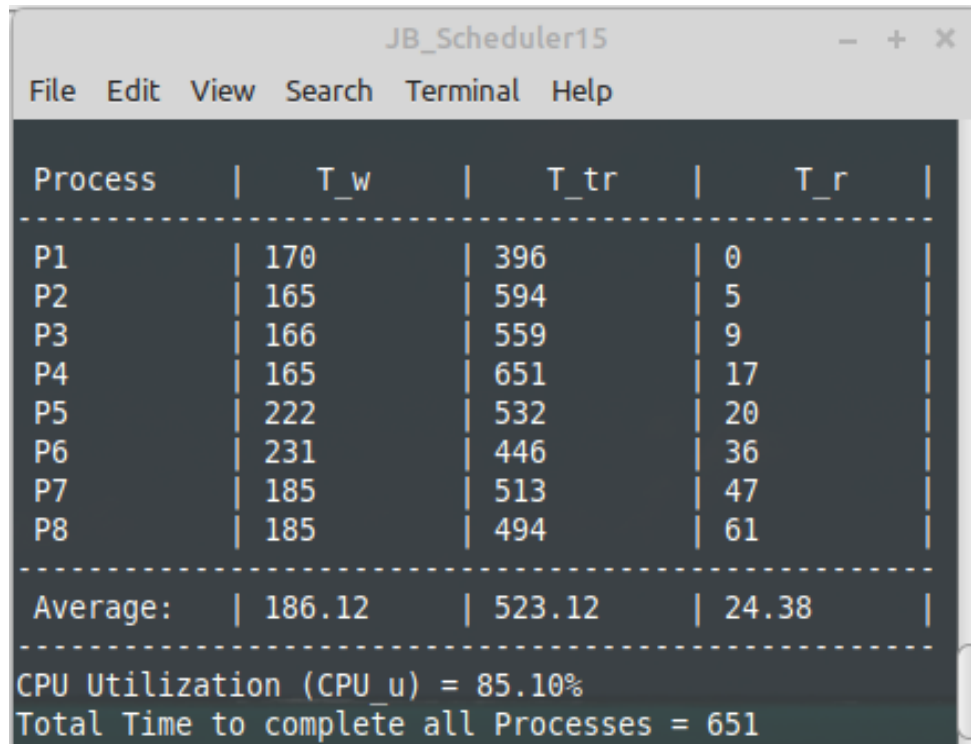
Ready\_1 list before entering the `while` loop in the `main()` function. This was necessary since that specific action needed special exceptions that could not be as easily added to the `MLFQ_add_to_queue()` function.

This code has a lot of repetition that can still be optimized. In the time frame I was given, I made all the exceptions with additional code duplication so anything could be easily reversed if there was a problem. The two most common problems I experienced are: segmentation faults, due to the use of manual pointers, and infinite loops due to the `while` loop in the `main()`. This was done on purpose to find a stable algorithm. The stable algorithm should perform the task I want, enter the `while` loop until all `Process` structures are empty, exit the loop, print results and halt back at the start menu. Having to continue to diagnose the cause of infinite loops and fix segmentation faults was a very good diagnostic experience. The trick is to keep outsmarting the program with well-placed `cout` statements to locate the line of code containing the dereferenced null pointer, which causes the segmentation fault. In my case with the infinite loops, they would always be caused by an incorrect node type getting placed in the wrong queue at the wrong time, for example, an `'I'` node in the Ready List. This was not supposed to happen, and when it did, it activates a special condition, for example, in the `FCFS_SJF_CPU_queue_timer()` function, when `if(front->burst_type == 'C')` is false, the else statement is a `cout << CPU_queue_timer: invalid condition << endl;` When this happens, the program enters infinite loop, which is actually good for diagnosing the cause. The program can be quit with `ctrl+c`, and I can scroll back through the output and find out when the clock stopped, since this condition prevents the `Global_clock` variable from incrementing.

## Scheduling Program Flowchart



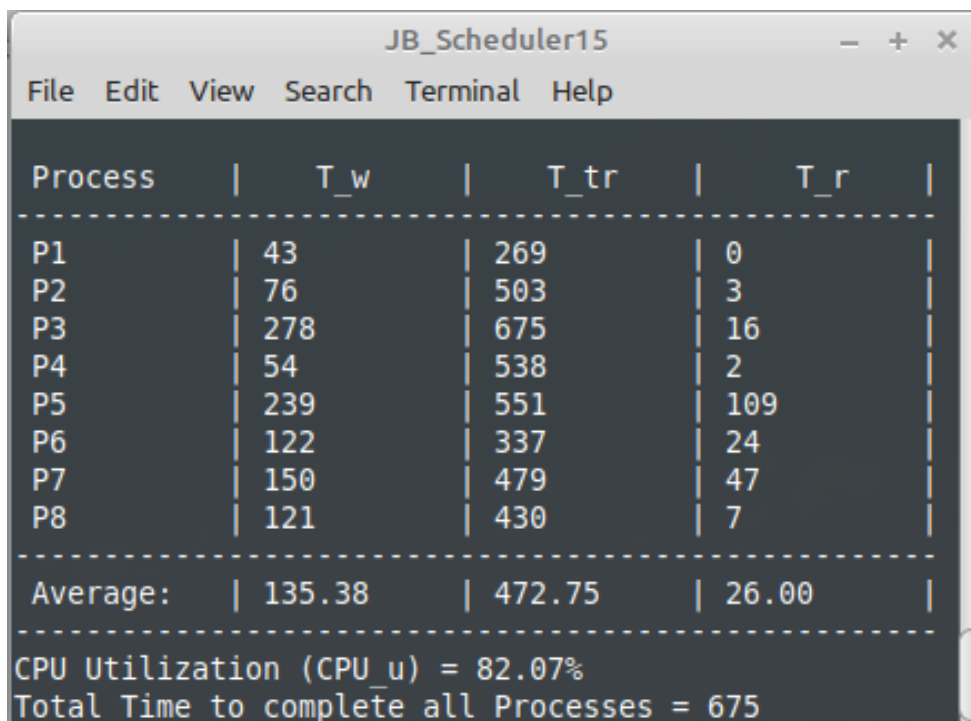
### Presentation of Final Results: FCFS Program Output



Process	T_w	T_tr	T_r
P1	170	396	0
P2	165	594	5
P3	166	559	9
P4	165	651	17
P5	222	532	20
P6	231	446	36
P7	185	513	47
P8	185	494	61
Average:	186.12	523.12	24.38

CPU Utilization (CPU\_u) = 85.10%  
Total Time to complete all Processes = 651

### Final Results: SJF Program Output



Process	T_w	T_tr	T_r
P1	43	269	0
P2	76	503	3
P3	278	675	16
P4	54	538	2
P5	239	551	109
P6	122	337	24
P7	150	479	47
P8	121	430	7
Average:	135.38	472.75	26.00

CPU Utilization (CPU\_u) = 82.07%  
Total Time to complete all Processes = 675

## Final Results: MLFQ Program Output

Process	T <sub>w</sub>	T <sub>tr</sub>	T <sub>r</sub>
P1	61	291	0
P2	126	550	5
P3	208	598	9
P4	56	545	14
P5	313	624	17
P6	215	428	22
P7	283	603	27
P8	188	500	32
Average:	181.25	517.38	15.75

CPU Utilization (CPU<sub>u</sub>) = 86.70%  
 Total Time to complete all Processes = 624

## Program Averages:

	Algorithm Average Results		
	SJF	FCFS	MLFQ
CPU Utilization: CPU <sub>u</sub>	82.07%	85.10%	86.70%
Avg Waiting Time: T <sub>w</sub>	135.38	186.12	181.25
Avg Turnaround Time: T <sub>TR</sub>	472.75	523.12	517.38
Avg Response Time: T <sub>R</sub>	26	24.38	15.75

## Discussion of Final Results

In this simulation, the average total waiting time,  $T_w$ , is highest for FCFS at 186.12, lowest for SJF at 135.38, and MLFQ is a median value at 181.25. This is a predictable outcome, since FCFS is known as the worst-case scenario for average waiting times of all processes, and SJF is the best-case. This means that MLFQ landing in the middle with a median average wait time is acceptable and plausible as a working algorithm. If the MLFQ average wait time had been below SJF or Above FCFS, that would signify incorrect programming. One issue with the SJF algorithm is that the individual process wait times are scattered, since every incoming process to the Ready list must be ordered, leaving longer CPU bursts in the back of the queue for extra time, which causes starving. This can become a problem with MLFQ as well, since processes can get stuck in a lower queue waiting for the higher priority queues to be emptied. This is less of an issue in MLFQ since the long CPU burst processes that may be waiting in a lower queue have already started their CPU time in higher queues, so they are not waiting to start like in the SJF algorithm. SJF always has the best  $T_w$ , even though it is not always the overall best algorithm.

For average turnaround time,  $T_{tr}$ , the highest value is again FCFS at 523.12, lowest for SJF at 472.75, and MLFQ is again a median value of 517.38. This is also a sensible result, since turnaround time is calculated by subtracting the process start time from the process finish time. Since FCFS is the least efficient algorithm, it is likely to score high in turnaround times since the processes spend the most time waiting for CPU time. SJF should score low due to the shortest CPU bursts always being in the front of the queue, resulting in many quickly finishing processes. SJF in this simulation was again the best in this category.

Average response time,  $T_r$ , is highest for SJF at 26.00, lowest for MLFQ at 15.75, and a median value of 24.38 for FCFS. The reason for the lowest response time of MLFQ is due to the Round Robin scheduling for the highest priority queue. Since the highest priority queue has a time quantum  $T_q$  of five time units, the algorithm is efficient and fair, since for  $n$  processes, each process gets exactly  $1/n$  of CPU time, and maximum waiting time for any process is always limited to  $(n - 1) \times T_q$ . In this situation, every process starts quickly, resulting in a low average  $T_r$ . The reason for SJF being the highest response time of 26.00 is because the Ready List must always be rearranged so the processes entering it are sorted in order of the shortest CPU burst time toward the front of the queue. This is poor performance for response time, since P5 in this case has its first CPU burst time of 16, so it is initially placed at the back of the queue. In addition to this, as processes leave the Ready List to go to I/O and then return to the Ready List, they are continually placed in front of P5 with its long CPU burst time, which starves P5 and cannot start until the `Global_clock` reaches 109. This is unacceptable because it is not fair. FCFS scores in the median range for  $T_r$  because every process in the ready queue must finish before it is transferred to the I/O queue, and the order is never shuffled like SJF. For average response time, SJF is not fair, where FCFS and MLFQ are fair, but MLFQ is far better than either SJF or FCFS due to Round Robin implementation of the `Ready_1` list.

For CPU utilization,  $CPU_u$ , the best performing algorithm is the MLFQ with 86.70%, the worst is SJF with 82.07%, and the median value is FCFS with 85.10%. It makes sense that MLFQ would score the highest in this category because it has multiple ready queues. In the case of SJF and FCFS, there is only one ready queue, so the possibility of having an empty Ready List waiting for processes returning from I/O is greater than if you had three ready queues to manage. Since  $CPU_u$  is `CPU_total_time / CPU_idle_time` in the MLFQ algorithm, as soon as

the Ready\_1 and/or the Ready\_2 queues are empty, the CPU can reduce idle time by running the processes in the Ready\_2 or Ready\_3 queues while the rest of processes are still out to I/O.

For total time to complete all processes, MLFQ scores the lowest time, finishing the fastest with 624 time units, SJF the highest time at 675, and the median value is FCFS with 651 time units. Again, the MLFQ algorithm scores the lowest total time because it has multiple ready queues which enables the CPU to keep busy when a higher priority queue is empty. Since MLFQ keeps the CPU busier, the processes get serviced in less time, so the simulation ends the fastest. SJF in this case takes the longest to complete because the longest CPU bound processes are continually waiting at the back of the ready queue to get serviced while other I/O bound short CPU burst processes always get serviced quickly at the front of the queue. This leads to an imbalance in the SJF algorithm which results in the I/O bound processes completing multiple CPU bursts before other processes can complete their first or second, such as P5 in SJF, stuck in the ready queue not starting until Global\_clock time is 109.

In conclusion, the overall best performing algorithm is the MLFQ algorithm. While it is only marginally better than FCFS in average wait time, it does not starve any large CPU burst processes like the SJF algorithm and scores the best times in multiple areas. It finishes all processes in the least total time, has the greatest CPU utilization percentage, the lowest average response time due to Round Robin scheduling in the two highest priority queues, and a median value for average turnaround time. All these factors lead to no starving of processes and a fair algorithm. This is the reason why this type of process scheduler is implemented in most current operating systems.



## Bibliography

Furht, Prof. Borijovie. "Lecture 3: CPU Scheduling." COP 4160: Operating Systems. COP 4160: Operating Systems, 2021, Boca Raton, Florida, Florida Atlantic University.