Operating Systems
CS 3502 Sec 01 Spring 2025
Project 2

Matthew Pfleger
Kennesaw State University
mpfleger@students.kennesaw.edu
GitHub

April 28, 2025

1 Abstract

This project explores and compares two different CPU scheduling algorithms: Shortest Remaining Time First (SRTF) and Multi-Level Feedback Queue (MLFQ) to determine which one is best suited for the fictional company ***OwlTech Systems***. Both were implemented using C# to simulate how an operating system handles process scheduling. A custom Process class was created to track details like arrival time, burst time, and completion metrics for each simulated task. The SRTF algorithm focuses on improving responsiveness by always selecting the job with the shortest remaining time, while MLFQ is designed to adapt to different types of processes by adjusting priorities and allowing movement between multiple queues. Performance was evaluated using various test cases, including general use, large-scale simulations, and edge cases. Key metrics such as CPU utilization, average waiting time, turnaround time, and throughput were measured and visualized through a series of graphs. This project highlights the strengths and trade-offs of each algorithm and gives a practical understanding of how CPU scheduling affects overall system performance.

Contents

2 Introduction

2.1 CPU Scheduling

CPU scheduling is a process done by an operating system to determine which task or jobs get to use the CPU in a particular time frame. The CPU can only handle one process at a given time so being able to schedule multiple processes is extremely important. The main goal of CPU scheduling is to maximize CPU utilization and minimize waiting times and response times.

2.2 Project Background

As a new employee of the fictional ***OwlTech Systems*** I have been tasked with evaluating different CPU scheduling algorithms to determine which would be best for the company needs.

This project is a simulation of a system using two different CPU scheduling algorithms. The purpose of this project is to demonstrate how to implement various types of CPU scheduling algorithms, and to compare the performance between the different algorithms to determine which algorithm to recommended to my bosses at ***OwlTech Systems***.

Using the C# programming language I created a *Process* class for this project. This class will be used in simulation to demonstrate a process needing execution in an operating system. I will be referring to this class throughout this report.

```
public class Process
{
    public int Id { get; set; }
    public int ArrivalTime { get; set; }
    public int BurstTime { get; set; }
    public int RemainingTime { get; set; }
    public int CompletionTime { get; set; }
    public int WaitingTime { get; set; }
    public int TurnaroundTime { get; set; }
}
```

3 Shortest Remaining Time First (SRTF)

3.1 Description

The Shortest Remaining Time First scheduling algorithm is the preemptive form of the Shortest Job Next (SJN) algorithm. In SRTF the processes are scheduled based on the job with the lowest remaining burst time.

SRTF algorithms are great for minimizing average waiting times. Because of its preemptive nature, shorter jobs will not be waiting while longer jobs are being executed. Shorter processes thrive under SRTF scheduling causing the system to feel more responsive overall. Systems that use SRTF scheduling algorithms are ideal for time-critical systems that need to ensure time-sensitive processes execute on time.

Because the system will constantly re-prioritize shorter jobs, there is a risk that longer jobs will experience indefinite waiting and starvation if shorter jobs keep arriving. SRTF requires a high amount of overhead from the operating system, because of the necessity to constantly check for incoming jobs and their burst times and frequent context switching.

3.2 Implementation

```
public static void srtfAlgo(List<Process> processes)
        {
            int time = 0; // Current time in the scheduling simulation
            int completed = 0; // Number of processes that have
    completed execution
            int n = processes.Count; // Total number of processes

            // Continue until all processes are completed
            while (completed != n)
            {
                Process shortest = null; // Variable to store the
    process with the shortest remaining time

                // Iterate through all processes to find the shortest
    remaining time process
                foreach (var process in processes)
                {
                    // Check if the process has arrived and still has
    remaining time
                    if (process.ArrivalTime <= time && process.
    RemainingTime > 0)
                    {
                        // Update the shortest process if it's null or
     the current process has a shorter remaining time
                        if (shortest == null || process.RemainingTime
    < shortest.RemainingTime)
                        {
                            shortest = process;
                        }
                    }
                }

                // If no process is ready to execute, increment the
    time and continue
                if (shortest == null)
                {
                    time++;
                    continue;
                }

                // Execute the shortest process for one unit of time
                shortest.RemainingTime--;

                // If the process has completed execution
                if (shortest.RemainingTime == 0)
                {
                    // Calculate completion time, turnaround time, and
     waiting time
                    shortest.CompletionTime = time + 1; // Completion
    time is the current time + 1
                    shortest.TurnaroundTime = shortest.CompletionTime
    - shortest.ArrivalTime; // Turnaround time
                    shortest.WaitingTime = shortest.TurnaroundTime -
    shortest.BurstTime; // Waiting time
                    completed++; // Increment the count of completed
    processes
                }

                // Increment the current time
                time++;
            }
        }
```

In my implementation of the Shortest Remaining Time First it takes a List of

our Process class. The function will simulate several processes executing through a system using a SRTF scheduling algorithm, making sure to track each processes completion, waiting, and turnaround time.

First, the program will initialize three integer variables *time* (current time in the simulation), *completed* (the number of processes that have completed execution), and n (the total number of processes).

The main while loop is declared to allow the simulation to run until all the processes have been executed. Then, find the processes that have the shortest remaining time by iterating though all the processes in the list and determining if the process has arrived, and if it still has remaining time to execute. If both conditions are met by multiple processes, select the one with the shortest time remaining for execution.

An if loop is included in the middle of the code to handle idle time, where no job is executing. The loop checks to see if there is no process executing if so, the code will increase the time and use the *continue* keyword to return looping through the code.

Execution of a given process is simulating by decrementing the current process in execution's remaining time. Once a process's remaining time is zero, execution is complete. Once the process is complete the if loop will handle process completion. This loop will record the completion, waiting, and turnaround time of the completed process for later performance comparisons.

Once the code reaches the last instruction in the main while loop it will increment the *time* variable, marking the passage of one unit of time.

4 Multi-Level Feedback Queue (MLFQ)

4.1 Description

Multi-Level Feedback Queue Scheduling algorithms are like the Multilevel Queue Scheduling, where processes are divided into several hierarchy queues. Each queue has its own level of priority and process type. In MLFQ scheduling processes can move between levels which leads to more efficient scheduling. The priorities of each process can be adjusted dynamically, based on a multitude of factors. Each level is assigned a time slice (quantum) that determines the amount of CPU time each process will have before it is preempted for another process.

Multi-Level Feedback Queue Scheduling is more flexible than other CPU scheduling algorithms allowing processes to change priority and move between queues as needed. Because of the time quantum and preemption MLFQ algorithms will prevent starvation from occurring.

However, one major downside of MLFQ algorithms is the high overhead required to implement them. Due to constant context switching, queue swapping, and priority adjustments MLFQ's can also be highly complex algorithms.

4.2 Implementation

```
1  public static void mlfqAlgo(List<Process> processes, int[] quantum)
2  {
3      int time = 0;
4      var readyQueues = new Queue<Process>[quantum.Length];
5
6      for (int i = 0; i < quantum.Length; i++)
7      {
8          readyQueues[i] = new Queue<Process>();
9      }
10
11     var pendingProcesses = new List<Process>(processes.OrderBy(p => p.
       ArrivalTime));
```

```
12
13      while (pendingProcesses.Count > 0 || readyQueues.Any(q => q.Count
    > 0))
14      {
15          // Add newly arrived processes to the first queue
16          for (int i = 0; i < pendingProcesses.Count;)
17          {
18              if (pendingProcesses[i].ArrivalTime <= time)
19              {
20                  pendingProcesses[i].RemainingTime = pendingProcesses[i
    ].BurstTime;
21                  readyQueues[0].Enqueue(pendingProcesses[i]);
22                  pendingProcesses.RemoveAt(i);
23              }
24              else i++;
25          }
26
27          bool didExecute = false;
28
29          for (int i = 0; i < quantum.Length; i++)
30          {
31              if (readyQueues[i].Count == 0)
32                  continue;
33
34              var process = readyQueues[i].Dequeue();
35
36              // Execute the process
37              int execTime = Math.Min(quantum[i], process.RemainingTime)
    ;
38              time += execTime;
39              process.RemainingTime -= execTime;
40
41              // Check for newly arrived processes during execution
42              for (int j = 0; j < pendingProcesses.Count;)
43              {
44                  if (pendingProcesses[j].ArrivalTime <= time)
45                  {
46                      pendingProcesses[j].RemainingTime =
    pendingProcesses[j].BurstTime;
47                      readyQueues[0].Enqueue(pendingProcesses[j]);
48                      pendingProcesses.RemoveAt(j);
49                  }
50                  else j++;
51              }
52
53              if (process.RemainingTime == 0)
54              {
55                  process.CompletionTime = time;
56                  process.TurnaroundTime = process.CompletionTime -
    process.ArrivalTime;
57                  process.WaitingTime = process.TurnaroundTime - process
    .BurstTime;
58              }
59              else
60              {
61                  // Demote to lower queue or stay in last queue
62                  if (i + 1 < quantum.Length)
63                      readyQueues[i + 1].Enqueue(process);
64                  else
65                      readyQueues[i].Enqueue(process);
66              }
67
68              didExecute = true;
69              break; // Only execute one process per cycle
70          }
```

```
71
72          // If no process was executed, advance time
73          if (!didExecute)
74          {
75              time++;
76          }
77      }
78  }
```

My implementation of a Multi-Level Feedback Queue scheduling algorithm takes two inputs. The first input is a list of our Process class from earlier. The second is an array of quantum values, representing the time quanta for each queue in the MLFQ.

Three variables are initialized at the beginning of the program *time* (tracks the current time in the simulation), *readyQueues* (an array of queues where each array corresponds to a level in the MLFQ), and *pendingProcesses* (a list of processes that have yet to arrive; sorted by arrival time).

The main loop for the simulation until all *penndingProcesses* have been added ot the queues, and all *readyQueues* are empty (no processes remain to execute). At each time interval the algorithm checks for arriving processes and adds them to the first queue (*readyQueues[0]*). Each processes remaining time variable is initialized to its burst time.

The algorithm iterates through each queue from highest to lowest priority. If the queue is empty, it moves on to the next queue in order of priority. If the queue has processes, the first process is dequeued for execution. The process is executed for the minimum value of either its time quantum or its remaining time. The time variable is increased by the execution time, and the remaining time for the process is decreased.

During processes execution the algorithm checks for any newly arriving processes. If any processes arrive during this time they are added to the first queue (*readyQueues[0]*). The algorithm also includes an if loop to handle CPU idle time. If during the main loop no process was executed the algorithm increments the time variable to continue the simulation.

5 Testing

5.1 General

To begin our analysis, I test both scheduling algorithms using a straightforward, controlled scenario involving three processes. Each process is assigned a unique arrival time and burst time to simulate a basic yet representative workload. This test case is designed to verify the fundamental correctness and functionality of the algorithms under typical operating conditions. By observing their behavior in this initial setup, I can establish a baseline for comparison and ensure that the core scheduling logic operates as intended before introducing more complex or varied test scenarios.

I created a method named *PerformanceMetrics()* to output the results of each algorithm. These outputs include average waiting time, average turnaround time, CPU utilization, and throughput (processes/second). In each testing section I have included graphs to compare the performance of each algorithm.

```
1           public static void PerformanceMetrics(List<Process> processes)
2           {
3               int totalTime = processes.Max(p => p.CompletionTime);
4
5               int totalProcesses = processes.Count; // Total number of
    processes
6               // Waiting time
```

```
            double avgWaitingTime = processes.Average(p => p.WaitingTime);
            // Turnaround time
            double avgTurnaroundTime = processes.Average(p => p.TurnaroundTime);
            //Throughput/second
            double throughput = (double)totalProcesses / totalTime;
            // Cpu utilization
            double totalBurstTime = processes.Sum(p => p.BurstTime);
            double cpuUtil = (totalBurstTime / totalTime) * 100;









            // Display the performance metrics
            Console.WriteLine("Performance Metrics:");
            Console.WriteLine($"Total Processes: {totalProcesses}");
            Console.WriteLine($"Average Waiting Time: {avgWaitingTime:F2}");
            Console.WriteLine($"Average Turnaround Time: {avgTurnaroundTime:F2}");
            Console.WriteLine($"Throughput: {throughput:F2} processes/second");
            Console.WriteLine($"CPU Utilization: {cpuUtil:F2}%");
        }
    }
}
```

5.1.1 Results

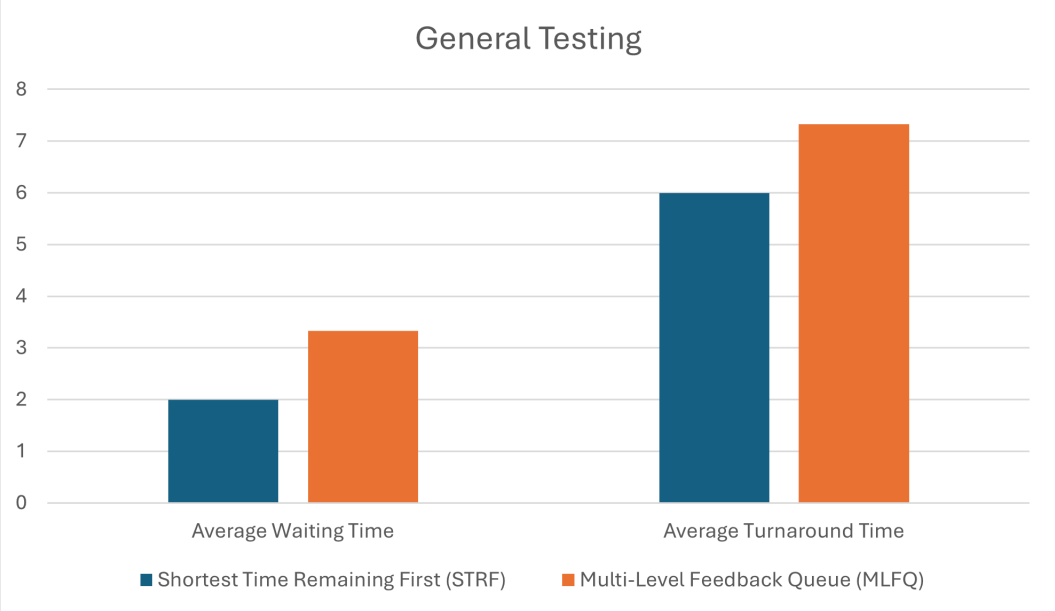

Figure 1: General Testing (Avg Wait Times / Avg Turnaround Time)

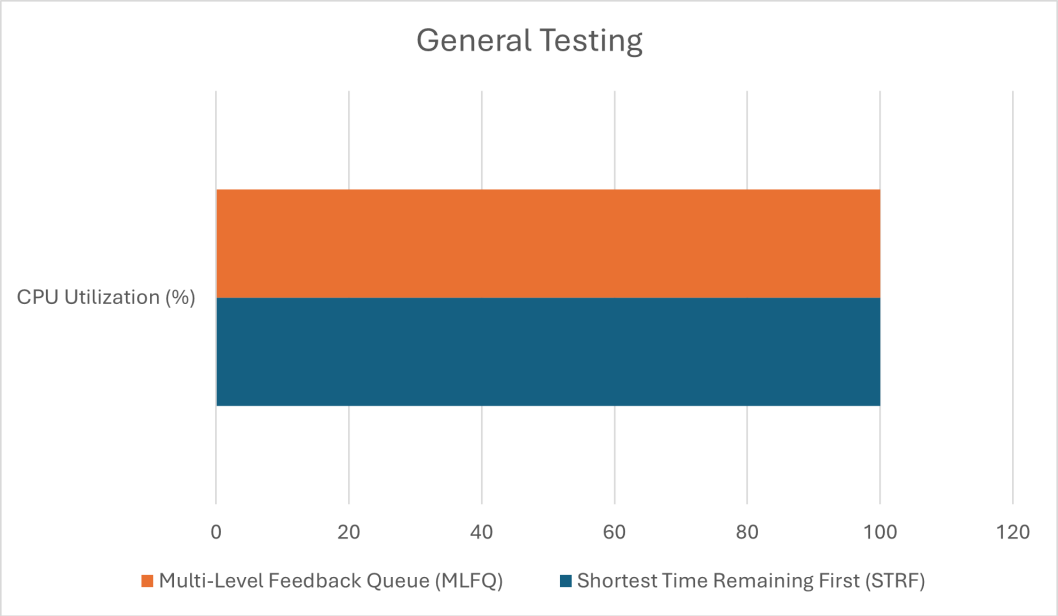

Figure 2: General Testing (CPU Utilization)

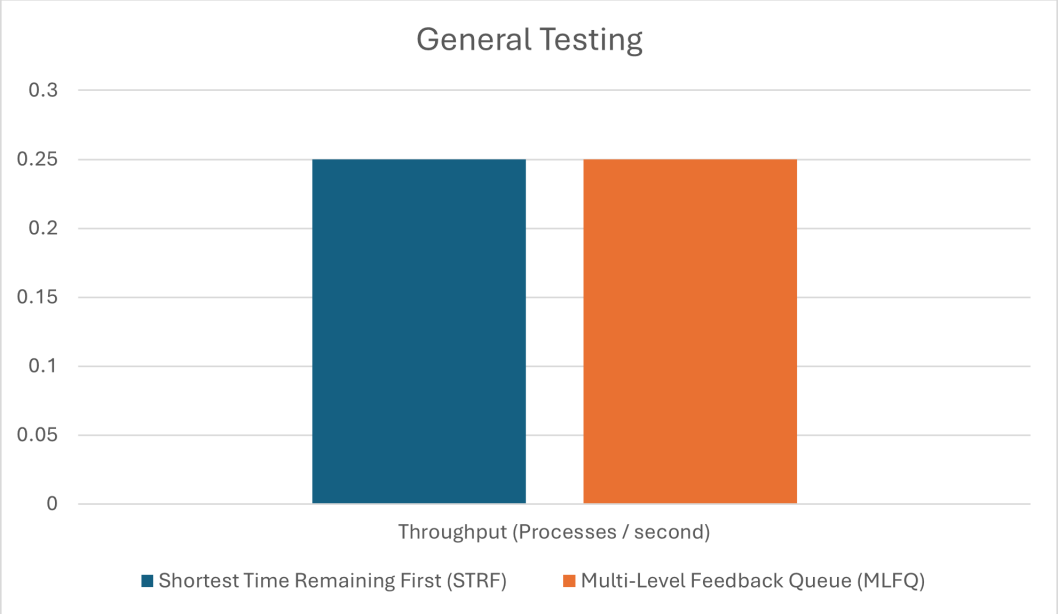

Figure 3: General Testing (Throughput)

5.2 Large Cases

To evaluate how the scheduling algorithms handle a more demanding workload, I ran a test involving 1000 individual processes. Each process was given randomly assigned arrival and burst times, creating a workload that mimics a complex and varied real-world system. This test was designed to observe how the algorithms perform under pressure—specifically looking at how they manage tasks, allocate CPU time, and maintain system responsiveness. The results offer a deeper understanding of each algorithm's strengths and limitations when scaled to a much larger number of processes.

5.2.1 Results

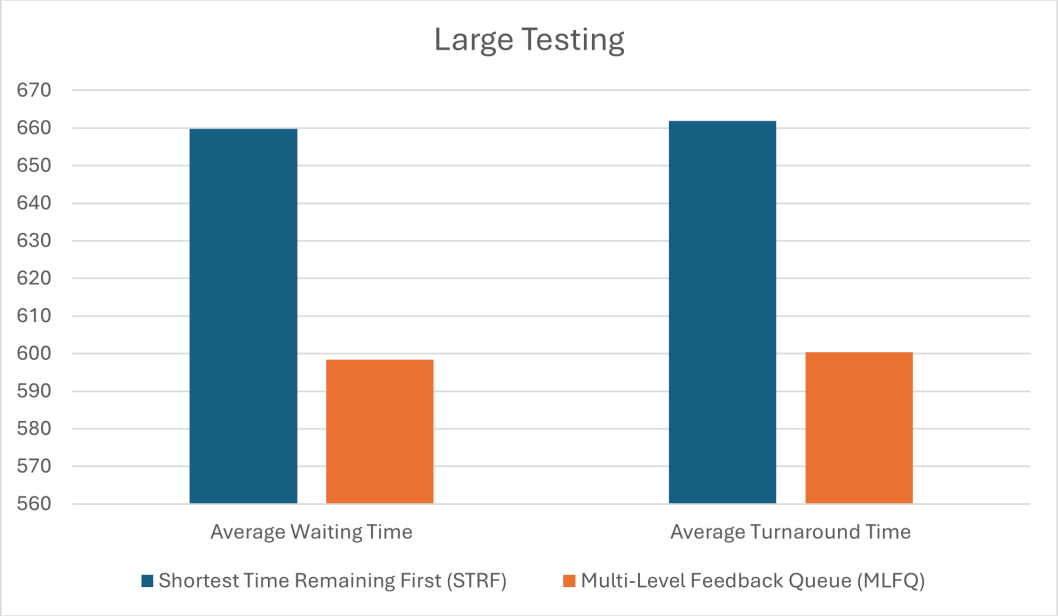

Figure 4: Large Testing (Avg Wait Times / Avg Turnaround Time)

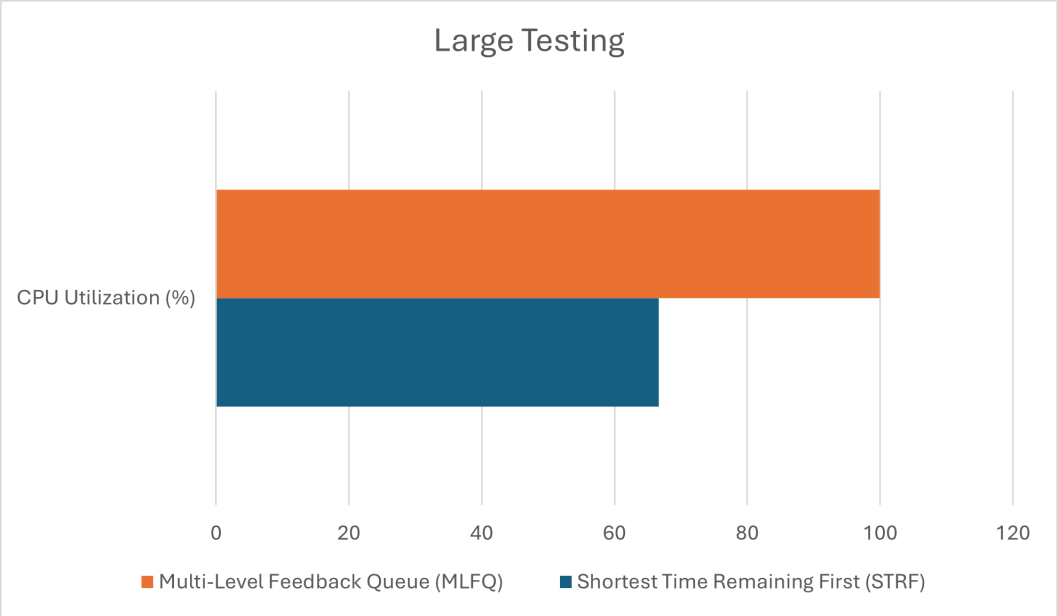

Figure 5: Large Testing (CPU Utilization)

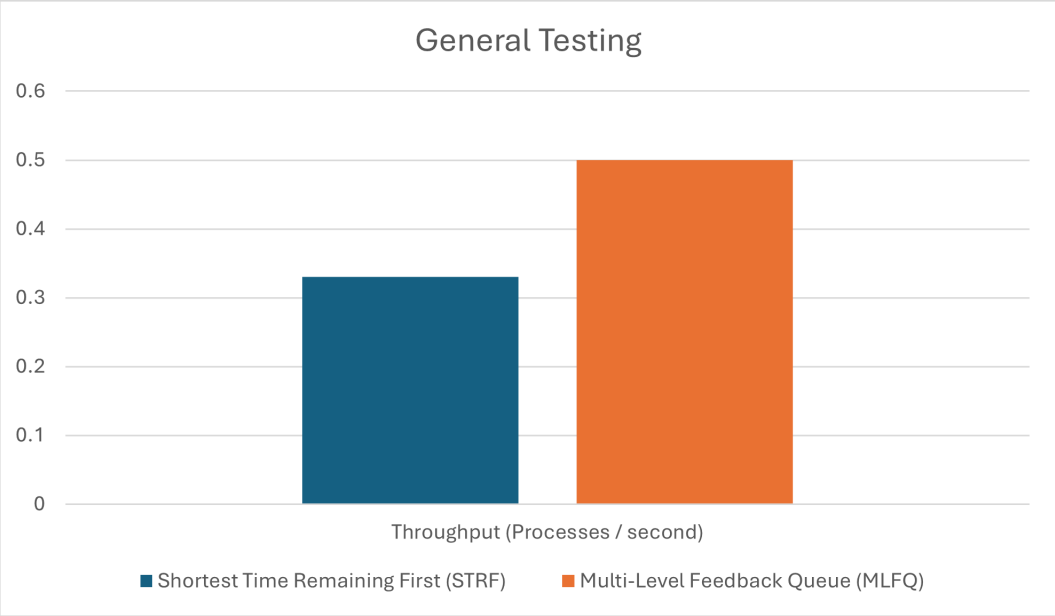

Figure 6: Large Testing (Throughput)

5.3 Edge Cases

Further testing of my algorithm designs required a look at two different edge cases. A single process load, and multiple processes arriving at the same time. Looking at these cases is critical for determining how the algorithms handle these abnormal cases.

For the single process I evaluated the behavior of both the SRTF (Shortest Remaining Time First) and MLFQ (Multi-Level Feedback Queue) algorithms when managing only a single process. This scenario represents the minimal load condition and is useful for confirming the algorithms can correctly handle and terminate processes without unnecessary overhead or mismanagement. As expected, both algorithms performed optimally, processing the single task without any delays or scheduling conflicts.

The second edge case test involves three processes that arrive at the exact same time but have varying burst times. It was designed to test the fairness and prioritization logic of each algorithm under simultaneous load conditions. For the SRTF algorithm, the test validated its ability to correctly prioritize processes based on the shortest remaining time, successfully preempting longer tasks when shorter ones were available. The MLFQ algorithm, meanwhile, was assessed for how it handled these identical arrival times within its multiple queues and time quanta settings. The test revealed how effectively the algorithm could distribute and promote/demote processes across different levels of priority based on CPU usage and time slice behavior.

5.3.1 Results

For this results section it is less effective to look at graph comparisons because the true purpose of these tests is that each algorithm can execute properly under these conditions. Therefore, I will put the output of each algorithm below.

SINGLE PROCESS:

```
SRTF Algorithm Edge Case Test: Single Process
Performance Metrics:
Total Processes: 1
Average Waiting Time: 0.00
Average Turnaround Time: 5.00
Throughput: 0.20 processes/second
CPU Utilization: 100.00%

MLFQ Algorithm Edge Case Test: Single Process
Performance Metrics:
Total Processes: 1
Average Waiting Time: 0.00
Average Turnaround Time: 5.00
Throughput: 0.20 processes/second
CPU Utilization: 100.00%
```

SAME ARRIVAL TIME:

```
SRTF Algorithm Edge Case Test: Same Arrival Time Process
Performance Metrics:
Total Processes: 3
Average Waiting Time: 3.33
Average Turnaround Time: 7.33
Throughput: 0.25 processes/second
CPU Utilization: 100.00%

MLFQ Algorithm Edge Case Test: Same Arrival Time Process
Performance Metrics:
Total Processes: 3
Average Waiting Time: 6.33
Average Turnaround Time: 10.33
Throughput: 0.25 processes/second
CPU Utilization: 100.00%
```

6 Conclusion

6.1 Recommendation

My final recommendation for ***OwlTech Systems*** is for the Multi-Level Feedback Queue (MLFQ) scheduling algorithm. The MLFQ scheduling algorithm outperforms the Shortest Remaining Time First (SRTF) algorithm in key ways that I believe make it the superior choice for ***OwlTech Systems***.

In heavy load testing the MLFQ runs circles around SRTF. In my testing of the two algorithms MLFQ significantly outperformed SRTF in CPU utilization (MLFQ: 99.95% vs SRTF: 66.67%), and in throughput (MLFQ: 0.5 processes/sec vs SRTF: 0.33 processes/sec). MLFQ also thrives in average waiting time and turnaround time under heavy load (**Waiting Time:** MLFQ: 598.40 vs SRTF: 650.83) & (**Turnaround Time:** MLFQ: 600.40 vs SRTF: 661.83). This data indicates the MLFQ scheduling algorithm will be effective in handling many concurrent processes, something that is common in real world, dynamic systems.

While in some areas such as the edge cases you see SRTF have slight advantages, the MLFQ algorithm remains a strong competitor. Also, MLFQ inherently

provides better responsiveness by allowing priority boosting and dynamic queue reassignment. This adaptability makes it a great CPU scheduling algorithm for general-purpose operating systems where process behavior can be less predictable.

7 References

References

[1] GeeksforGeeks, "Shortest remaining time first (preemptive sjf) scheduling algorithm," 07 2017.

[2] GeeksforGeeks, "C tutorial," 04 2020.

[3] GeeksForGeeks, "Multilevel feedback queue scheduling (mlfq) cpu scheduling - geeksforgeeks," 08 2019.

[4] GeeksforGeeks, "Cpu scheduling in operating systems - geeksforgeeks," 07 2019.

[5] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*. Wiley, 2018.

[6] T. Point, "Operating system scheduling algorithms - tutorialspoint," 2019.