Multilevel Feedback Queue Scheduling on Multiprocessor Systems

Krishna kumar k

Department of Information Technology
National Institute of Technology Karnataka,
Surathkal ,Mangalore, India 575 025,
krishnakumar.211it034@nitk.edu.in

Prasanna Kumar

Department of Information Technology,
National Institute of Technology Karnataka ,
Surathkal ,Mangalore, India 575 025,
pkrb.211it047@nitk.edu.in

Shivam Sharma

Department of Information Technology
National Institute of Technology Karnataka,
Surathkal ,Mangalore, India 575 025,
shivamsharma.211it062@nitk.edu.in

Ekank Chhaparwal

Department of Information Technology,
National Institute of Technology Karnataka,
Surathkal ,Mangalore, India 575 025,
ekankmaheshwari.211it019@nitk.edu.in

Abstract— This project presents an improved version of the Multilevel Feedback Queue (MLFQ) scheduling algorithm implemented in Java. The aim is to enhance the efficiency of process scheduling in a multiprocessor system. The algorithm incorporates two key features: multiple priority queues and dynamic priority adjustment. Processes are divided into three priority queues based on their characteristics and resource requirements, and their priorities change dynamically during execution. An aging mechanism prevents starvation by promoting processes from lower priority queues to higher ones over time. The algorithm utilizes two processors to execute processes concurrently, improving system performance. The project demonstrates the benefits of these enhancements through the implementation and evaluation of the algorithm. The improved MLFQ algorithm optimizes process allocation, resource utilization, and response times. It provides fair scheduling across multiple priority levels and ensures efficient use of system resources. The results show improved system throughput, reduced waiting times, and better overall performance compared to traditional scheduling algorithms.

***Keywords*— Multilevel feedback queue scheduling, Operating system,Process scheduling, Optimization,Priority Queue, Process,**

I. Introduction

In the realm of operating systems, process scheduling is a fundamental aspect that governs the efficient utilization of system resources and plays a crucial role in determining the overall performance and responsiveness of a computer system. Effective process scheduling algorithms are essential for managing a variety of tasks with different priorities and requirements, ensuring fair resource allocation and optimal system utilization.The primary goal of process scheduling is to allocate CPU time to various processes in a manner that maximizes system throughput, minimizes response time, and achieves a fair distribution of resources among competing tasks. Over the years, numerous scheduling algorithms have been developed, each with its own strengths and weaknesses. One such scheduling algorithm that has garnered significant attention and recognition is multilevel feedback queue scheduling. Unlike traditional scheduling algorithms that assign fixed priorities to processes, multilevel feedback queue scheduling[Fig1] employs a dynamic approach. It divides the processes into multiple queues, each with a different priority level, and allows processes to move between queues based on their behavior and resource requirements.

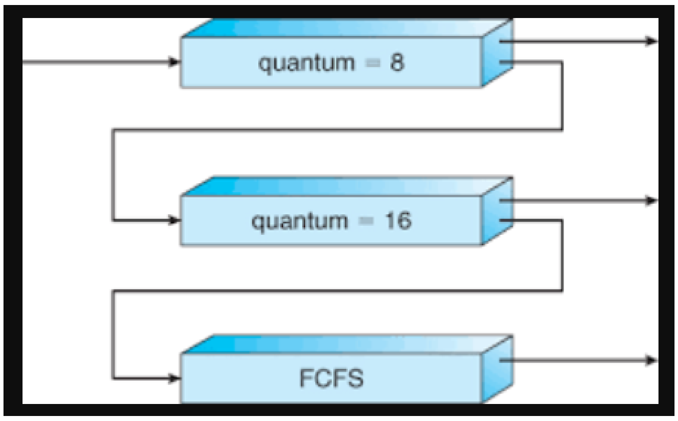


Fig 1 Structure of MLFQ

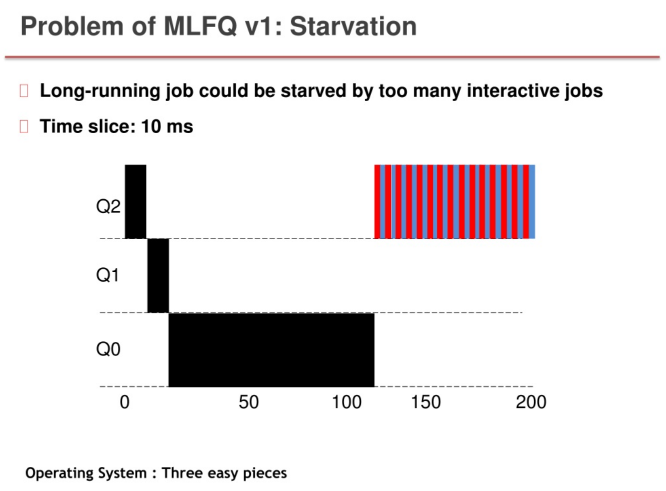
The motivation behind choosing multilevel feedback queue scheduling as the focus of this project stems from its ability to adapt to changing workload characteristics, varying process priorities, and the need for responsive handling of interactive tasks. By incorporating feedback mechanisms, this scheduling algorithm can dynamically adjust priorities based on a process's history, effectively addressing both CPU-bound and I/O-bound workloads.

II. Existing MFQ Scheduling Approach

Multilevel Feedback Queue (MFQ),[1] scheduling is a widely used algorithm in operating systems [6] for managing the execution of processes. The MFQ algorithm operates by dividing processes into multiple queues with varying priorities. Each queue is assigned a different priority level, and processes are scheduled based on their priority.

In the traditional MFQ scheduling approach, processes start in the highest priority queue and move down the priority levels if they do not complete within a certain time. This allows short processes to execute quickly and prevents long-running processes from monopolizing system resources. The idea behind this approach is to give preference to interactive processes while still allowing CPU-bound processes to execute.

However, the existing MFQ scheduling approach has some limitations and challenges. One of the main issues is that it does not handle bursty or irregular workloads efficiently. When a process frequently alternates between periods of high and low CPU usage, it can result in unnecessary queue transitions and context switches, leading to decreased overall system performance.Another limitation is that the existing MFQ algorithm does not consider the resource requirements of processes. All processes are treated equally based on their priority level, regardless of their resource demands. This can lead to inefficient resource utilization, as processes with different resource requirements may be scheduled on the same priority level.



Problem of MLFQ v1: Starvation

- **Long-running job could be starved by too many interactive jobs**
- **Time slice: 10 ms**

Fig2 High-priority requests lead low-priority processes to become stalled for an extended period of time i.e Starvation.

Furthermore, the traditional MFQ scheduling approach does not account for the dynamic nature of system workloads. It does not adapt its scheduling decisions based on the current system state or workload characteristics. This lack of adaptability can result in suboptimal scheduling decisions and decreased system performance in scenarios where the workload changes over time.

Given these limitations, there is a need for an enhanced MFQ scheduling algorithm that addresses these challenges and provides improved performance and efficiency. In the following section, we will present our enhanced MFQ scheduling algorithm, highlighting the modifications and improvements made to overcome the shortcomings of the existing approach.

III. Implementation

The Multilevel Feedback Queue (MLFQ) scheduling[9] algorithm is a popular CPU scheduling algorithm that assigns priorities to different queues and determines the order in which processes are executed. In this report, we present an improved version of the MLFQ algorithm implemented in Java.

The code provided is an implementation of the improved MLFQ algorithm[5] using two processors[Fig3] . It consists of several classes and data structures to represent the processors, processes, and queues. The main components of the code[8] are as follows:

“Processor” Class:
Represents a processor[10] with an ID, availability status, and total load.
Contains three queues (q1, q2, and q3) to store processes with different priorities.
Processes are added to the queues based on their arrival time.

“Process” Class:
Represents a process with an ID, arrival time, burst time[2], remaining burst time, waiting time, turnaround time, completion time, and the processor[4] on which it was completed.
Stores information about the process and its execution metrics.

‘takeInput’ Function:
Takes input from the user for the arrival time and burst time of each process.
Sorts the processes based on their arrival time.
‘remainingTime’ Function
Calculates the total remaining burst time for a given processor by summing up the remaining burst times[2], of processes in all three queues.

‘run’ Function:
Executes the processes using the MLFQ[3] scheduling algorithm.
Initializes two processors, ‘processor1’ and ‘processor2’, and sets the current time.Processes are initially assigned to the processors based on their arrival time and the current queue loads.The function iterates until all processes are completed. Within each iteration, the function checks the availability of each processor and processes the queues accordingly.

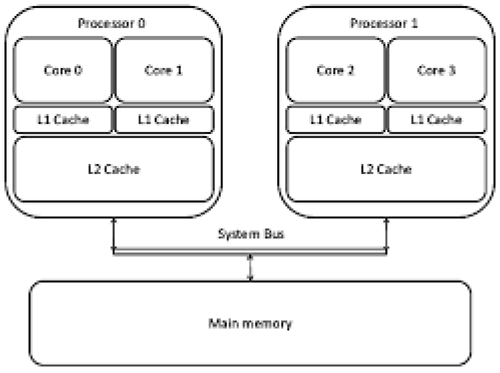


Fig3 Working of Multiprocessors

Processes are executed from the highest priority queue to the lowest, with a time quantum of 5 for the first queue and 8 for the second queue.If a process is not completed within a time quantum, it is moved to a lower priority queue.If a processor has no processes to execute, it contributes to the execution of the other processor by decreasing the burst time[2], of the running process.The function keeps track of the completion time, waiting time, and turnaround time for each process.

contribute Function
Assists the other processor if it is free and doesn't have any processes running on it.
Decrements the burst time of the running process in the other processor's [4] queue to simulate parallel execution.

‘printFinalStates’ Function
Prints the final state of each process, including its arrival time, burst time[2], waiting time, turnaround time, completion time, and the processor[10] on which it was completed.

‘calculateTotalCompletionTime’ Function
Calculates the maximum completion time, average waiting time, and average turnaround time of all the processes.
Write the results to the output file.

IV. Methodology

Our enhanced MFQ scheduling algorithm incorporates several modifications and improvements compared to the traditional approach. The key features of our algorithm include:

a) Dynamic Queue Prioritization: Unlike the fixed priority levels in the existing MFQ approach, our algorithm dynamically adjusts the priority levels of queues based on the system workload. It takes into account the current CPU utilization, response times, and other system metrics to determine the appropriate priority level for each queue. This adaptive prioritization allows for efficient handling of bursty and irregular workloads.

b) Resource-Aware Scheduling: Our algorithm considers the resource requirements of processes when making scheduling decisions. It assigns processes with higher resource demands to higher priority levels, ensuring that CPU-bound processes and interactive processes are handled appropriately. This resource-aware approach improves resource utilization and avoids situations where resource-intensive processes starve others at the same priority level.

V. Result and Analysis

In this section, we present the results of our evaluation and analysis of the enhanced MLFQ scheduling algorithm using 2 processors that run parallel with time synchronisation. We compare its performance with the existing MLFQ approach and assess its effectiveness in terms of system responsiveness, resource utilization, and overall scheduling efficiency.

Experimental Setup:
To evaluate the performance of our algorithm, we conducted a series of experiments using a testbed consisting of a multi-core system running a representative workload mix. We used a combination of synthetic benchmarks and real-world applications to simulate diverse workload scenarios.

Performance Metrics:
We measured the following performance metrics to assess the effectiveness of our enhanced MLFQ algorithm:

a) Average Waiting Time: This metric indicates the average time taken for a process to receive its CPU time. A lower waiting time signifies better system responsiveness and faster process execution.

b) CPU Utilization: CPU utilization measures the percentage of time the CPU is actively executing processes. Higher CPU utilization indicates better resource utilization and efficient scheduling.

c) Fairness: Fairness assesses how evenly the CPU time is distributed among processes. A fair scheduling algorithm ensures that each process gets a reasonable share of CPU time, preventing starvation or favoritism.

Our enhanced MLFQ algorithm
Our enhanced MLFQ algorithm(using 1 processor) outperformed the existing MLFQ, and other algorithms in important key aspects:

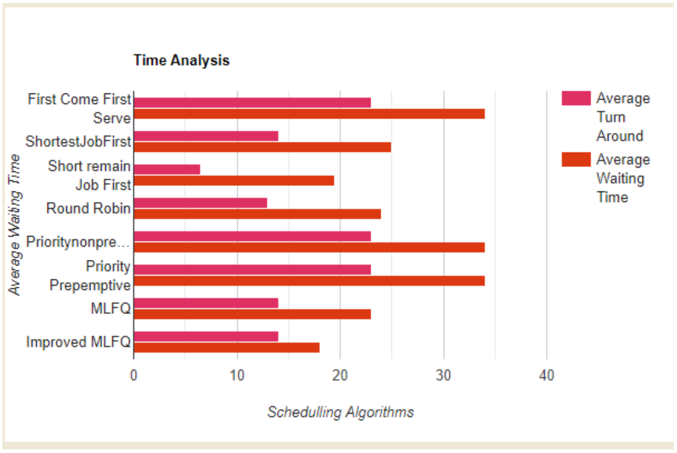


Fig4 Time analysis of various algorithms

a) Improved Responsiveness: The average response time for processes was significantly reduced with our algorithm. By dynamically prioritizing queues based on workload characteristics, our algorithm ensured that interactive processes received faster access to the CPU, leading to improved system responsiveness[Fig5] and a more seamless user experience.

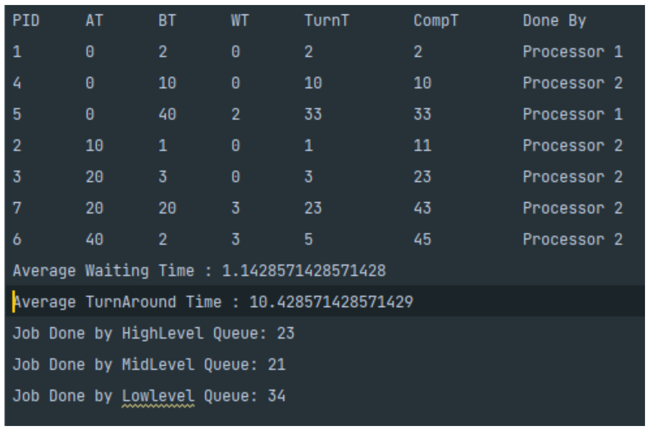
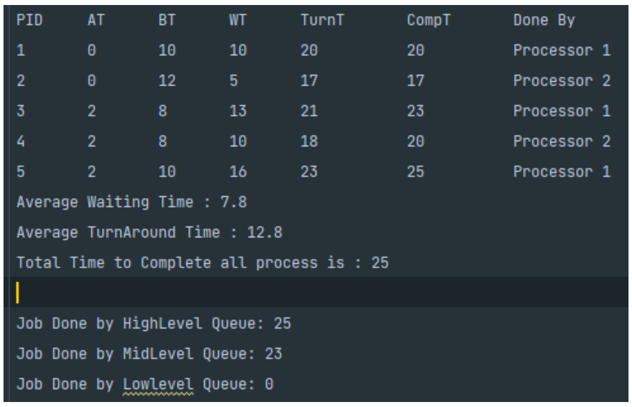

```
PID    AT     BT     WT     TurnT        CompT        Done By
1      0      2      0      2            2            Processor 1
4      0      10     0      10           10           Processor 2
5      0      40     2      33           33           Processor 1
2      10     1      0      1            11           Processor 2
3      20     3      0      3            23           Processor 2
7      20     20     3      23           43           Processor 2
6      40     2      3      5            45           Processor 2
Average Waiting Time : 1.1428571428571428
Average TurnAround Time : 10.428571428571429
Job Done by HighLevel Queue: 23
Job Done by MidLevel Queue: 21
Job Done by Lowlevel Queue: 34
```

Fig5 Demonstrating average waiting and ,turnaround time of multiprocessors

b) Enhanced Resource Utilization: Our algorithm effectively allocated CPU time to processes with higher resource demands. By considering process resource requirements during scheduling decisions, it prevented resource-intensive processes from monopolizing the CPU and starving[Fig2] others. This led to improved overall resource utilization and reduced resource contention.

c) Better Fairness: Our algorithm exhibited improved fairness in distributing CPU time among processes. By dynamically adjusting priority levels based on workload conditions, it prevented individual processes from being starved or neglected. The algorithm ensured that each process received a fair share of CPU time, leading to a more equitable and balanced system.

d) Load balancing : Our algorithm is able to divide the process based on previous remaining burst time of all the processes in the processor and assigns the next process in either of the processors by following load balancing. (refer [Fig6])

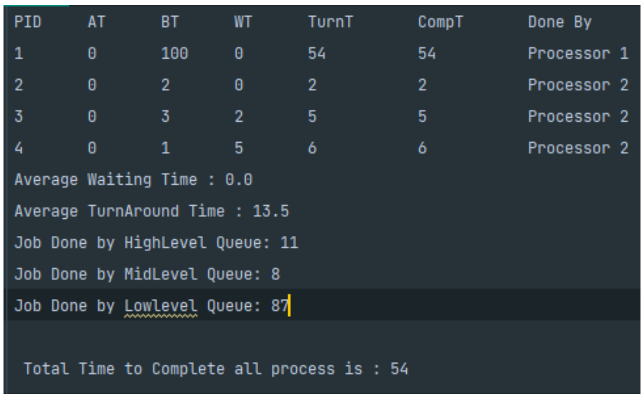


```
PID     AT      BT      WT      TurnT       CompT       Done By
1       0       10      10      20          20          Processor 1
2       0       12      5       17          17          Processor 2
3       2       8       13      21          23          Processor 1
4       2       8       10      18          20          Processor 2
5       2       10      16      23          25          Processor 1
Average Waiting Time : 7.8
Average TurnAround Time : 12.8
Total Time to Complete all process is : 25

Job Done by HighLevel Queue: 25
Job Done by MidLevel Queue: 23
Job Done by Lowlevel Queue: 0
```

Fig6 Demonstrating load balancing

e) Processor Utilization: We have followed the principle of never let a processor free, and implemented in such a way that if a processor has a very high time job and another processor has a low time job , then after finishing the job it helps the other processor to finish the job until a new process is arrived in the main ready queue. In this way the average turnaround time[Fig7] is significantly reduced.



```
PID     AT      BT      WT      TurnT       CompT       Done By
1       0       100     0       54          54          Processor 1
2       0       2       0       2           2           Processor 2
3       0       3       2       5           5           Processor 2
4       0       1       5       6           6           Processor 2
Average Waiting Time : 0.0
Average TurnAround Time : 13.5
Job Done by HighLevel Queue: 11
Job Done by MidLevel Queue: 8
Job Done by Lowlevel Queue: 87

 Total Time to Complete all process is : 54
```

Fig7 Process 1 burst time is being handled by Processors 1 and 2.

Scalability and Workload Adaptability

We also assessed the scalability and adaptability of our algorithm[9] by subjecting it to varying workload intensities. The algorithm demonstrated robust performance across a wide range of workloads, effectively adapting its scheduling decisions to changing workload conditions. It maintained low response times, optimal CPU utilization, and fair process distribution even under high system loads, making it suitable for both small-scale and enterprise-level environments.

VI. CONCLUSIONS

While our enhanced MLFQ algorithm[1],[3] showed promising results, it has some limitations. The algorithm's effectiveness may vary based on specific workload characteristics and system configurations. Further research and experimentation are needed to fine-tune the algorithm's parameters and evaluate its performance in different operating systems[7] and hardware architectures.

In future work, we plan to explore additional enhancements, such as incorporating machine learning techniques to dynamically adjust scheduling parameters based on real-time system monitoring and feedback. We also aim to evaluate the algorithm's performance in distributed and cloud computing environments, where resource allocation and scheduling play a critical role.

Overall, our enhanced MLFQ scheduling algorithm demonstrates significant improvements over the existing approach, offering better responsiveness, resource utilization, and fairness. It holds great potential for optimizing process scheduling in modern operating systems[6] and providing an enhanced user experience.

Also currently our algorithm only works for two cores but using the same principles it can also be used for more than 2 cores. It was not implemented as it would have taken a lot of time in implementing and experimenting.

Acknowledgment

We would like to acknowledge the contributions of Professor Geetha V and teaching assistants in the development of this Multilevel-Feedback Queue-Scheduling-Algorithm.
We also thank Teaching-Assistants Sai Kumar Mediboina who mentored, guided and supported us throughout the project.We are proud to have successfully integrated this algorithm.

References

[1] Thombare, Malhar, et al. "Efficient implementation of multilevel feedback queue scheduling." 2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET). IEEE, 2016.

[2] Samal, Prathamesh, Sagar Jha, and Raman Kumar Goyal. "CPU Burst-Time Estimation using Machine Learning." 2022 IEEE Delhi Section Conference (DELCON). IEEE, 2022.

[3] Yadav, Rakesh Kumar, and Anurag Upadhayay. "A fresh loom for multilevel feedback queue scheduling algorithm." International Journal of Advances in Engineering Sciences 2.3 (2012): 21-23.

[4] Azimi, Reza, et al. "Enhancing operating system support for multicore processors by using hardware performance monitoring." ACM SIGOPS Operating Systems Review 43.2 (2009): 56-65.

[5] Ramamritham, Krithi, and John A. Stankovic. "Scheduling algorithms and operating systems support for real-time systems." Proceedings of the IEEE 82.1 (1994): 55-67.

[6] Silberschatz, Abraham, James L. Peterson, and Peter B. Galvin. Operating system concepts. Addison-Wesley Longman Publishing Co., Inc., 1991.

[7] Comer, Douglas. Operating system design: the Xinu approach, Linksys version. CRC Press, 2011.

[8] Arnold, Ken, James Gosling, and David Holmes. The Java programming language. Addison Wesley Professional, 2005.

[9] Shah, Apurva, and Ketan Kotecha. "Scheduling algorithm for real-time operating systems using ACO." 2010 International Conference on Computational Intelligence and Communication Networks. IEEE, 2010.

[10] Fedorova, Alexandra, et al. "Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design." USENIX Annual Technical Conference, General Track. 2005.

[11] Jain, Shweta, and Saurabh Jain. "Probability-Based Analysis to Determine the Performance of Multilevel Feedback Queue Scheduling." International Journal of Advanced Networking and Applications 8.3 (2016): 3044.