The Evolution of Schedulers: Round-Robin to MLFQ

Chenyu Wang
Stony Brook University

Yash Singhania
Stony Brook University

1 Introduction

- Presentation Recording: recordings Passcode:.$q3qzM1
- Source Code: https://github.com/BruceWang3077/xv6-scheduler/tree/mlfq

In our project, we enhanced the xv6 operating system by introducing the Multi-Level Feedback Queue (MLFQ) scheduler, replacing the simpler round robin approach that equally distributes CPU time but fails to recognize the diverse needs of different processes. This change aims to tackle inefficiencies especially apparent in environments with a mix of interactive and computational tasks. MLFQ dynamically categorizes tasks into multiple queues based on their priority, adjusting each process's priority based on its CPU demands and interaction patterns.

Interactive tasks, often characterized by brief bursts of CPU usage followed by waiting periods for I/O operations, stand to gain from MLFQ. When these tasks resume, they are quickly elevated to a higher priority, reducing response times and enhancing the user experience. On the other hand, CPU-intensive tasks that occupy the processor for extended periods are systematically moved to lower-priority queues, preventing them from hogging resources and promoting a fairer CPU time distribution.By incorporating periodic priority boosts, MLFQ also mitigates the risk of process starvation, where tasks could linger indefinitely in low-priority states. These boosts ensure that all processes, regardless of their priority level, receive adequate CPU time. This balanced approach not only improves the system's responsiveness to interactive applications but also improves overall throughput and efficiency. Implementing MLFQ in xv6 is set to transform its scheduling capabilities, making it a valuable update for educational settings where students can witness and learn from enhanced scheduling dynamics.

The rest of the paper is organized as follows: Section 2 is the motivation of the project; Section 3 is the detailed description of the project, from MLFQ algorithm to implementation on xv6. Section 4 servers as evaluation; Section 5 and 6 are conclusion and related work.

2 Motivation

The motivation behind refining the Multi-Level Feedback Queue (MLFQ) scheduler, as detailed in the paper, stems from its sophisticated design that effectively addresses the challenges of managing diverse computing workloads without precise knowledge of job durations. MLFQ is particularly adept at optimizing both turnaround and response times by dynamically adjusting job priorities based on their execution behavior. This adaptability makes it a strong candidate for environments that handle a mix of interactive and long-running batch jobs, ensuring fair CPU time allocation and improving overall system responsiveness.

In contrast, the round-robin scheduler used in systems like xv6 is quite naive. This scheduler allocates CPU time in fixed intervals to each process in the queue, cycling through them without consideration of job length or priority. Such simplicity leads to suboptimal performance, especially in responsiveness and handling short jobs efficiently. Round-robin does not prioritize tasks based on their nature or urgency, which can result in longer wait times for interactive tasks and an inefficient handling of varied workloads.

MLFQ improves upon these aspects significantly. It enhances responsiveness to interactive jobs by prioritizing processes that frequently yield the CPU before their time slice expires. This behavior is ideal for interactive applications that require quick user feedback, ensuring that they receive prompt CPU attention. Furthermore, MLFQ's ability to learn and predict job characteristics based on historical behavior allows it to manage CPU resources more effectively, adapting to the needs of both short and long tasks dynamically. This results in a fairer and more efficient system, particularly in complex multi-tasking environments, making MLFQ a more suitable choice for modern operating systems that require robust and adaptable scheduling mechanisms.

3 Project Description

3.1 Multi-Level Feedback Queue

The Multi-Level Feedback Queue (MLFQ) scheduling algorithm is a sophisticated mechanism designed to optimize both turnaround time and responsiveness in multi-user operating systems. It dynamically adjusts the priority of processes based on their observed behavior, aiming to efficiently manage a diverse mix of interactive and batch processes. Here's a deep dive into how MLFQ works and the detailed explanations of its operational rules and corresponding graphical representations.

MLFQ uses multiple queues, each associated with a different priority level. Processes are initially placed in the highest priority queue and are moved between queues based on their behavior and history of execution. This dynamic adjustment aims to approximate the ideal scheduling by learning about the processes' characteristics as they run.

The Multi-Level Feedback Queue (MLFQ) scheduling system cleverly manages tasks across multiple queues, each assigned a different level of priority. Higher priority queues get the first dibs on CPU time, helping to quickly address the most urgent tasks. MLFQ dynamically adjusts a task's priority based on how it uses the CPU. If a task uses up its allotted time without interruption, suggesting it's a heavy-duty task, it gets moved down to a lower-priority queue. On the other hand, if a task often stops early, perhaps for fetching data or waiting for user input, it's treated as interactive and can retain or even gain priority. Additionally, the time each task has to run—known as quantum—varies between the queues. Higher priority queues have shorter quantums to swiftly handle interactive jobs, while lower ones have longer ones, ideal for the more demanding, CPU-intensive tasks.

MLFQ operates on several principles, often implemented as "rules" that define how jobs are scheduled:

Detailed Explanation of MLFQ Rules:

Rule 1: Priority Precedence This rule is foundational in maintaining order within the MLFQ system by stipulating that if one process (say A) has a higher priority than another process (say B), process A must run in preference to process B. This establishes a clear hierarchy and ensures that higher priority tasks receive the attention they need promptly, which is crucial for tasks that are more time-sensitive or critical. It helps in optimizing response times for higher priority processes, crucial for system efficiency and user experience.

Rule 2: Round-Robin Among Equals When two or more jobs share the same priority level, they are scheduled in a round-robin manner. This means each job gets a turn to use the CPU for a fixed time slice; after its time slice expires, the CPU is handed over to the next job in the queue. This rule ensures fairness, giving each job at the same priority level an equal opportunity to progress. This method prevents any job at the same priority level from monopolizing the CPU, ensuring a fair distribution of processing time among competing tasks, which is essential in a multi-user or multi-tasking environment.

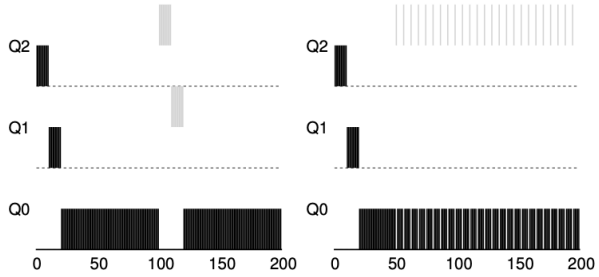

Figure 1: Interactive Job: Two Examples

Rule 3: Priority Initialization When a job first enters the system, it is placed in the highest priority queue. This rule is based on the assumption that a new job might be short and interactive, thus requiring immediate attention to keep the system responsive. Initially assigning high priority to new jobs allows the system to quickly respond to newly initiated processes, enhancing user experience by reducing apparent start-up delays.

Rule 4: Dynamic Priority Adjustment

This rule is about adapting the priority based on the job's behavior:

Rule 4a: If a job uses up its entire time allotment at a given priority level without relinquishing the CPU (e.g., for I/O operations), its priority is decreased (it moves down one queue). This adjustment is based on the rationale that the job is likely CPU-intensive and less sensitive to response time.

Rule 4b: Conversely, if a job relinquishes the CPU before using up its time allotment (indicating I/O or interaction), it retains it's priority level until the time-slice for current priority level is used up. This behavior suggests the job might be interactive, and maintaining its priority helps minimize response time.

These adaptive measures help the system dynamically adjust to the real-time needs of each job, promoting efficient CPU utilization while balancing turnaround and response times for diverse workloads.

Rule 5: Priority Boost After a certain period, all jobs are periodically boosted to the highest priority queue. This mechanism prevents any job from starving and ensures that even lower-priority, long-running jobs receive CPU time. This rule is crucial for maintaining fairness over longer periods, preventing the indefinite postponement of any job regardless of its nature, thereby ensuring that all processes make progress.

Figure 1 : Interactions of Jobs with Different Characteristics shows that this graph has two parts. The left part shows a long-running, CPU-bound job in Q0 and a short, interactive job entering at Q2, which quickly completes. The right part shows an interactive job performing I/O frequently, which

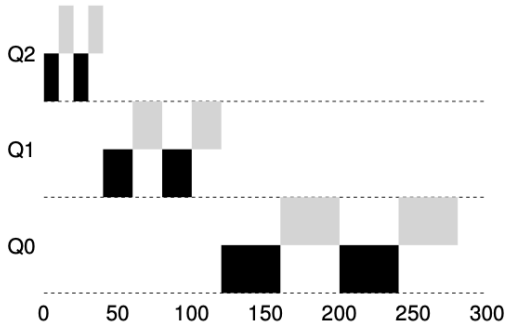

Figure 2: Lower Priority, Longer Quanta

maintains its high priority because it yields the CPU before using up its time slice. Relevance to Rules: Rule 4b: The right part of the figure perfectly illustrates Rule 4b, where a job that yields the CPU before its time allotment expires maintains its priority. Rule 2 and Rule 3: The left part shows Rule 3 in action as a new, short job starts at the highest priority and how Rule 2 handles two jobs at the same level (round-robin for equal priorities).

Figure 2: The Multi-Level Feedback Queue (MLFQ) scheduling algorithm uses multiple levels of queues and employs feedback to prioritize jobs based on their past behavior. Here's a detailed explanation of the rules that define this approach:

1. Rule 1: If one job (A) has a higher priority than another (B), A will run while B waits.

2. Rule 2: If jobs share the same priority, they run in a round-robin manner, following the time slice (quantum length) allocated to that specific queue.

3. Rule 3: When a job enters the system, it starts at the highest priority (topmost queue).

4. Rule 4: If a job exhausts its time slice in a particular queue (regardless of how often it voluntarily releases the CPU), its priority decreases, moving it down one queue.

5. Rule 5: After a specified time period (S), all jobs are moved back to the highest-priority queue.

MLFQ doesn't require prior knowledge of job behavior. Instead, it monitors job execution to assign priorities, offering the best of both worlds. It provides efficient performance for short, interactive jobs (like Shortest Job First or Shortest Time-to-Completion First), while ensuring fairness for long-running, CPU-intensive tasks. Due to its effectiveness, variations of the MLFQ scheduler are found in many operating systems, such as BSD UNIX, Solaris, and Windows NT.

3.2 Implementation

3.2.1 Implementation Choice

the following are our MLFQ implementation choices:

- **Priority Level** 4 priority levels(from 0 to 3), priority 3 is the highest and priority 0 is the lowest;
- **Timer Tick** The highest priority ready process is scheduled to run every time the xv6 timer tick(10ms) occurs;
- **Priority Time-Slice** Each priority has a fixed number of time slice:

Priority 3	8
Priority 2	16
Priority 1	32
Priority 0	unlimited

  when a process runs out of time slice associated with its current priority level, it will be downgraded one priority(except priority 0);

- **Round-Robin on the Same Priority** For multiple processes with the same priority level, the processes will be scheduled in a round-robin fashion.
- **Round-Robin Time-Slice** Each priority has it's round-robin time slice:

Priority 3	2
Priority 2	4
Priority 1	8
Priority 0	32

  if there are multiple processes with the same priority level, a process will run for the round robin time slice associated with the priority level once it's scheduled, then it will yield to the scheduler.

- **Priority Boost** To overcome the problem of starvation, a priority boost mechanism is implemented. The priority of all the processes will be reset to the Highest level(3) for every 480 timer ticks.

3.2.2 Implementation Details

There are many approaches when it comes to its implementation on xv6. In xv6, all the proc structs(64 by default) are stored in a process table (ptable) [3], and the scheduler's duty is to iterate the table to find the next runnable process and switch context to that process until either it switches back the context itself or the timer tick occurs.

The most common and intuitive way of implementation is to create data structures and logic as the scheduler algorithm suggests, which usually includes the number of queues required for the scheduler. As shown in Figure 3, each time the context is switched to scheduler, it starts round-robin loop from the highest priority queue to find a runnable process, if

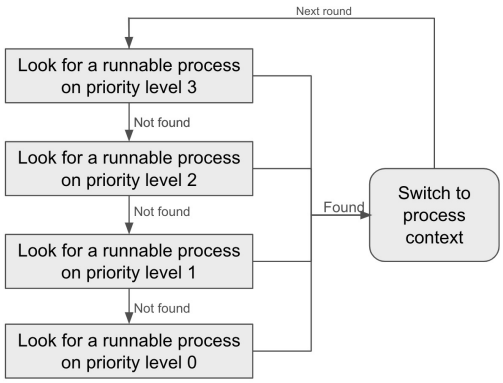

Figure 3: A Popular MLFQ Implementation

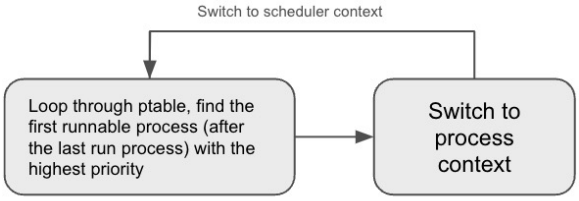

Figure 4: Our MLFQ Implementation

it's not found, then it move to the next lower priority queue until found(if no runnable process exists, just keep looping until there's one, which aligns with the original xv6 scheduler).

There are many advantages of this implementation. It's intuitive and easy to understand, it's scalable so able to be used on more advanced operating systems with more processes. However, the implementation process will be a little tricky. It requires creating several data structures in the kernel, and keeping track many states, such as the number of runnable processes on each priority level, and which priority level it should check next. Maintaining all these data queues and states correctly will be a tedious task.

This project adopts a different approach. Given that there are 64 runnable processes at most and the goal of this project is to demonstrate the performance difference brought by the scheduler design, we use an implementation which is not very intuitive, but very simple and easy to maintain.

The high level idea is to virtualize the priority queue design. No extra data structures like arrays are used to store the processes and each process has its current priority value. As illustrated in Figure 4, each time the context is switched back to the scheduler, the scheduler will iterate through the whole ptable, and choose the next process to run. It means that it will go through all the processes, and maintain a pointer pointing to the first runnable process with highest priority. Slight modification is made to the iteration so a process can keep running until it's current round-robin time-slice runs out.

This approach brought several benefits. First, no extra data

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  ...
  int tick;                 // For MLFQ
  int priority;             // For MLFQ
  int initial_tick;         // For evaluation
  int PRINT_TIME;           // For evaluation
};
```

Figure 5: Modified Code in proc.h

structures are needed to implement the multiple priority. Second, all the state information needed are the priority level and running tick store in each proc struct(shown in Figure5), thus dramatically reducing the number of states to keep track of.

4 Evaluation

Our evaluation focus on two parts: the correction of MLFQ implementation and the demonstration of MLFQ's responsiveness to interactive processes compared to xv6 original round-robin scheduler. The tests are written as user programs in xv6, which is run in QEMU emulator on a Linux machine.

We designed two user program. One is CPU intensive, which goes into a for loop with a very large number of iterations(200,000,000, in our case); the other is interactive, which goes into a for loop of 20 iterations where in each iteration it writes a single line to a file and sleeps for 5 ticks.

4.1 Correctness

In this experiment, we started 3 CPU intensive processes at the same time. We added code in the kernel it will print out the last running process and it's priority every time a timer tick occur. We run the test both on MLFQ and Round-Robin. We first take a look at round-robin, as shown in Figure 7, each of the three processes runs for 1 timer tick and swith to the next process.

From Figure 6 it can be concluded that MLFQ is running correctly. Each priority level has different length of round-robin time-slice and different priority level time-slice respectively. And after a priority boost(before 400 ticks and after 800 ticks), all the processes are moved to the highest priority again, until all the processes are finished.

4.2 Responsiveness to Interactive Processes

In this experiment, we designed a micro benchmark using the two user programs. The benchmark has 20 iterations, in

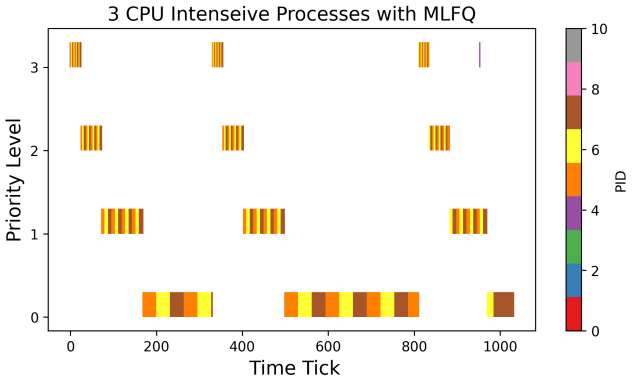

Figure 6: MLFQ Correctness

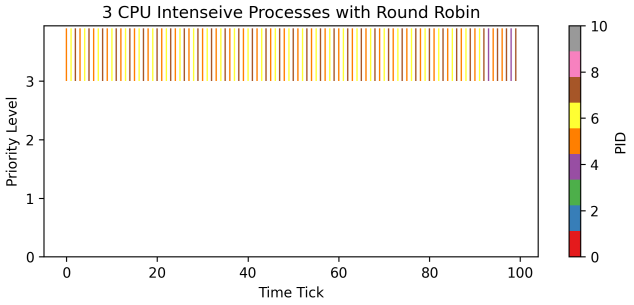

Figure 7: Round-Robin as Comparison

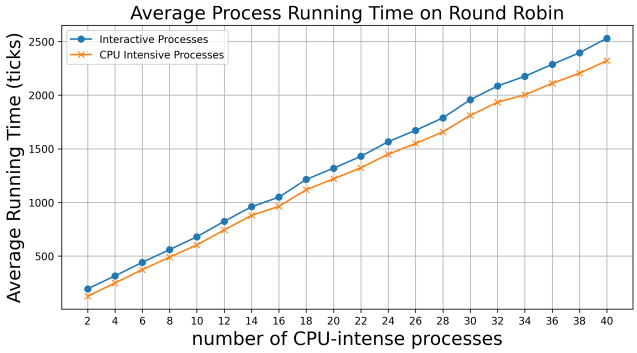

Figure 8: Mixed Benchmark on Round-Robin

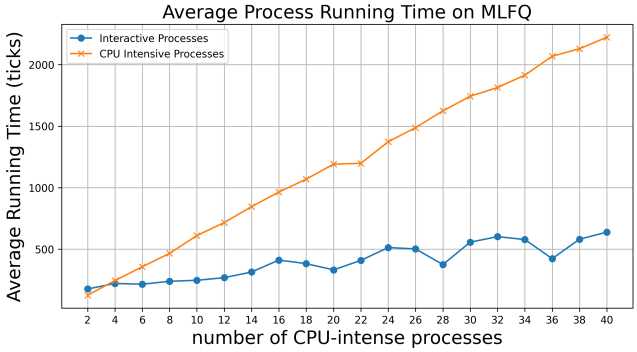

Figure 9: Mixed Benchmark on MLFQ

each iteration, 2 interactive processes are started, along with different number of CPU intensive processes, from 2 to 40. In each iteration we check the finishing time of all the processes in the unit of timer tick, and calculate the average finishing time of CPU intensive processes and interactive processes respectively.

Different from last test, simply adding printing statement to kernel code doesn't work well. Because printing itself is time costing, and interactive processes have significant less amount of time using the cpu, this approach will significantly increases the running time of CPU intensive process but the running time of interactive processes doesn't change much. In order to get the precise data, we added a new attribute *initial_tick*(Figure 5), which tracks the tick when it's initialized. Then we add a printing statement in the exit() syscall, which will print out the running time of the processes using the current tick minus the initial_tick.

We also add an attribute *PRINT_TIME*(Figure 5), which serves as a flag. Processes only with *PRINT_TIME* = 1 will print out the running time in *exit()* syscall. We implemented a new syscall *printtime()* to xv6, which will set *PRINT_TIME* to 1 for the caller process. This syscall enables us to manipulate which process will print out it's running time so only the process in the micro benchmark will print out the running time.

We run the micro benchmark on both round-robin and MLFQ. The result of round-robin is shown in Figure 8, it can be seen that with the number of CPU intensive processes increases, it takes longer for all the processes to finish. It aligns with the design of round-robin: even if an interactive process are runnable, it still has to wait until the scheduler has scheduled all the processes before it. Figure 9 shows the result on MLFQ. The finishing time of CPU-intensive processes is similar to that on round-robin because both schedulers behave fairly to CPU intensive processes. However, the finishing time of interactive processes didn't increase much with the number of CPU intensive processes increasing. This is because whenever an interactive process is ready, the scheduler will pick it over the other processes because of it's higher priority.

4.3 *sleep(time)* & Real Sleeping Time

During the implementation and evaluation, we dig deep in the kernel code to understand *sleep()* syscall since understanding it is crucial to the correctness of our implementation.

In the process, we found out that in the origial xv6 system, if a process calls *sleep(time)* syscall where *time* is the number of ticks it plans to sleep, there's possibility that it will sleep longer that *time* ticks. This is due to the round-robin design, only when the iteration reaches the process, the system will

check whether the process needs to sleep more. However, if there are too many runnable processes in the ptable, the scheduler won't reach the process until all the runnable processes are scheduled before it!

Interestingly, our implementation of choice mitigated this problem. as mentioned earlier, all the 64 processes are checked as the scheduler attempts to find the next process to run. As a result, all the sleeping processes will be checked too so they will be more likely to wake up on time. It's worth noticed that this implementation won't make all the process sleep exactly the amount of time it wants, for example, if a CPU intensive process changes its behavior to interactive and decides to sleep for some time, it's possible that this process won't be preempted because of its low priority.

We tested the problem with a simple program. The program starts 20 CPU intensive processes first then start another process *p* that just sleeps for 5 ticks and wakes up. The test shows that *p* sleeps 20 ticks on round-robin scheduler because the scheduler has to loop through all the other processes first. As a comparison, *p* sleeps for exactly 5 ticks on MLFQ because it is scheduled in time due to it's high priority.

4.4 Summary

In the evaluation section, we run two experiments on both MLFQ and round-robin scheduler. We modified kernel code and added syscall to extract the experiment data. The result shows the difference of responsiveness to interactive process due to the different design, and MLFQ will adapt better to a mixed workload where interactive processes need to be run sooner.

5 Conclusion

Multi-Level Feedback Queue is a scheduler design that set the priority of processes based on it's history. We implemented MLFQ on xv6 using a simple approach and tested its correctness and responsiveness to interactive process. The results demonstrates that MLFQ works significantly better on mixed workloads where the interactive processes in the wordloads need immediate attention. There are many other schedulers with different design and advantages, however, they're out of scope for this paper.

6 Related Work

A large number of schedulers have been presented during the past two decades. For example, multi-level feedback queue [1], BFS [4], CFS have all been used on Linux. Several studies have compared the difference of some of them [2, 4]. This project makes am attempt to implement and understand MLFQ scheduler design compared to Round Robin for educational purposes.

References

[1] Remzi H. Arpaci-Dusseau and Arpaci-Dusseau Andrea C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, LLC, 1.00 edition, 2015. http://pages.cs.wisc.edu/~remzi/OSTEP/.

[2] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The battle of the schedulers:{FreeBSD}{ULE} vs. linux {CFS}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 85–96, 2018.

[3] Russ Cox, M. Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2011. Available online: http://pdos.csail.mit.edu/6.828/2012/xv6.html (accessed on September 5, 2013).

[4] Taylor Groves, Jeff Knockel, and Eric Schulte. Bfs vs. cfs scheduler comparison. *The University of New Mexico*, 11, 2009.