Implementing and Evaluating Schedulers in xv6 (Multi-Level Feedback Queue, Lottery, Round-Robin)

Chavez Yi Wei Cheong Hyungbin Jun Mark Liu Jake Heller

Duke University

1 Introduction

A modern computer can run word processors, web browsers, games and many other types of programs concurrently, with almost instant response times when interacting with these programs even after rapidly switching between them. The reason computers can handle large volumes of programs so effectively is because of the scheduling functionality of modern operating systems.

Schedulers in operating systems look at a series of available programs, or a **workload** containing different **tasks**, and decide which tasks to schedule next and for how long. While this may seem simple, it is difficult when tasks have different and unknown start times and durations, with mixtures of CPU-bound and IO-bound tasks, and with different conflicting metrics to optimize for. A wide variety of schedulers have been developed to fit different needs and suited for different types of workloads. For instance, a First-In-First-Out (FIFO) scheduler is relatively easy to implement, but has poor performance and can often lead to unfair schedules. The Linux Completely Fair Scheduler (CFS) ensures fair, proportional distribution of CPU resources across tasks and is much more performant but is very complicated to implement.

A badly implemented scheduler can greatly lower the efficiency of a system, for instance by leading to starvation of tasks where some tasks never get run at all. With the importance of a good scheduler and the wide range of scheduler designs, it is thus important to evaluate the performance and fairness of different schedulers. We modify the existing round-robin (RR) scheduler in the xv6 operating system to add a multi-level feedback queue (MLFQ) and a lottery scheduler. MLFQ was chosen because it is performant, responsive and used in many modern OSes like Windows and MacOS. Lottery scheduler was chosen because it is fairer, a downside to MLFQ. To facilitate testing, we instrument xv6 to record various metrics based on the CPU cycles since boot and also design an array of different test workloads to run the different schedulers on.

The rest of the paper is structured as follows: Section 2 provides a brief background history on scheduler and scheduling design. Section 3 describes the high level design of the round-robin, MLFQ and Lottery schedulers, and describes the 6 common metrics used to evaluate schedulers (latency, responsiveness, throughput, fairness, overhead and predictability), as well as the types of test workloads we are looking to implement. Section 4 details the implementation of the two scheduler designs as well as the instrumentation added to the xv6 OS and test workloads designed to evaluate the performance of schedulers along the 6 performance criteria. We display the results of the user tests and evaluate the performance of the 3 schedulers in Section 5, and conclude in Section 6.

2 Background

The first computing systems in the early 1950s did not require schedulers, because they were large, expensive hardware-based systems that ran programs one at a time, loaded via card deck. As computing systems continued to develop, computing systems moved to batch processing and multiprogramming and developed rudimentary scheduling algorithms such as FIFO. As hardware continued to get cheaper and humans more expensive, programs become more interactive and thus more concurrency and preemptive scheduling was introduced in order to further allow for interactive timesharing. Today, different operating systems use a variety of different schedulers to fit their needs.

1. **Windows:** Cooperative Scheduling (Windows 3.1x), Priority Scheduling (Windows 95, 98), MLFQ (Windows 2000-present).
2. **Linux:** MLFQ (before 2.6.0), O(1) scheduler (2.6.0-2.6.23), CFS (2.6.23-present)
3. **MacOS:** MLFQ

3 Design

In this section, we begin by describing the design of the three schedulers we will evaluate: round-robin (RR), multi-level feedback queue (MLFQ) and lottery. We continue by describing the criteria we will use to evaluate the schedulers and the different test workloads that we will run on the schedulers.

3.1 Schedulers

3.1.1 Round-Robin

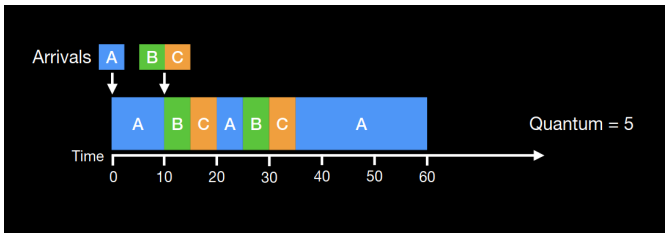

Figure 1: Sample workload and the way the round-robin scheduler schedules the tasks

RR loops through all runnable tasks, runs each task for some fixed scheduling quantum, and switches between tasks in order. For instance, referring to Figure 1, when tasks B and C are scheduled, the scheduler schedules all tasks in order (A,B,C) until completion.

We re-used the implementation of xv6's round-robin scheduler.

3.1.2 Lottery

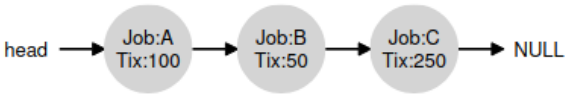

Figure 2: Sample workload for lottery scheduler and number of tickets for each task

The lottery scheduler allows all runnable tasks to have a non-zero number of lottery tickets, and generates a random number to select which task will be run next. Each task can have different numbers of lottery tickets, which enables the scheduler to prioritize some tasks over others, and some implementations allow for more customization of task priority by allowing priorities to be changed dynamically. Tasks that have more ticket having proportionally higher probability to be scheduled. For a sample workload as seen in Figure 2, the probability that B is scheduled is 1/5 times that of C and 1/2 that of B, proportionally to the number of tickets each task has.

The lottery scheduler can avoid the starvation problem, since as long as every runnable task has a non-zero number of lottery tickets, there will be a non-zero probability that the task will be selected for a scheduling event.

A downside of the lottery scheduler is that it is unpredictable and can yield widely different schedules for the same workload over different runs.

Due to time constraints, we implemented a simplified version of the lottery scheduler where each task has the same number of tickets and the number of tickets cannot be modified after task creation.

3.1.3 Multi-Level Feedback Queue

The MLFQ is a scheduling algorithm that maintains multiple FIFO queues to place runnable tasks in. Each queue has a priority level associated with it, where the scheduler will run tasks in the highest priority queues first, and then move to the lower level queues if the higher level queues are exhausted. Each priority level has a time quantum associated with it, with time quantums increasing for lower priority levels. If a task from a higher priority queue exhausts its time quantum without exiting or yielding itself, the task will be stopped and moved to a lower priority level. In particular, I/O bound and interactive tasks will either maintain their priority level or get moved into a higher level priority queue when waiting.

Overall, the MLFQ attempts to optimize both turnaround time and response time in order to make the system feel more responsive. Thus, the MLFQ favors tasks that use the CPU for short periods of time and tasks that are I/O bound or interactive.

Of course, there are some issues with this scheduler. An adversarial tasks, if the time quantum is known, can simply issue a lightweight I/O task right near the end of its time quantum, avoiding being moved down a priority level. Then, this tasks will be put back on the top level priority queue since the I/O task was lightweight, resulting in this tasks getting rescheduled and thus monopolizing the CPU over a long period of time. This can be solved by having the time quantum apply across multiple scheduling rounds, so tasks that take a long overall time will still be moved down.

Another possible issue is a task that begins without interactivity or I/O tasks, but then after a while it begins issuing lots of I/O tasks. This is unfortunate as it is likely that this tasks has already been placed in a low level priority queue, and thus will have quite long response times. There is also an issue of priority inversion, where a high priority task will attempt inter-process-communication with a low priority task, essentially blocking itself on the low priority task which will result in a long response time. Finally,

tasks at the bottom of the MLFQ will most likely be starved, since a continuous barrage of new tasks will prevent those at the bottom from ever executing.

All of the issues above can be solved by priority-boosting, or adjusting the priority level of a tasks outside of the base mechanisms of the MLFQ. For example, if a task begins executing lots of I/O's, the MLFQ can boost its priority to a higher level. For linked tasks of different priority levels, the lower priority task can temporarily be boosted to the higher priority task as to avoid inconsistent response times and blocking. Finally, if a tasks has been starved for a certain time threshold, it can be boosted up to the top level priority queue.

For the sake of time, priority-boosting and the more advanced time quantum features were not included, but we thought it was important to note some drawbacks (and possible solutions) for the base MLFQ design.

3.2 Evaluation Criteria and Metric Tests

3.2.1 Evaluation Criteria

We evaluate the schedulers for their performance and fairness along 6 criteria:

1. **Latency:** How long each task takes to complete.
2. **Responsiveness:** How long a task needs to wait before it is scheduled for the first time.
3. **Throughput:** How many tasks can be completed per unit time.
4. **Overhead:** How much extra work is done by the scheduler.
5. **Predictability:** How consistent is the performance over time
6. **Fairness:** How equal are resources given to different tasks.

3.2.2 Metric Tests

To simulate different types of workloads that a scheduler may need to handle, we designed three test workloads with different combinations of IO-bound and CPU-bound tasks to evaluate the performance of each scheduler according to the 6 criteria detailed above:

1. CPU-bound, all tasks run for same amount of time and arrive at approximately the same time
2. IO-bound, all tasks run for same amount of time and arrive at approximately the same time.
3. Mixed IO and CPU-bound, tasks start at different times and run for different amounts of time

4 Implementation

In this section, we begin by describing how we adapted the existing xv6 operating system to implement different schedulers, instrument metric tracking of CPU cycles and add user tests containing different workloads of tasks. The OS was run inside a Docker container using QEMU as a hardware emulator.

4.1 Implementing Scheduling Policies in xv6

4.1.1 Round-Robin

We used the existing xv6 implementation of the round-robin scheduler, which has a set time quantum of 10ms. The schedule continually loops through the `proc` array to find the next runnable process. Since it iterates through all existing processes, there is no possibility of a process being missed during that iteration.

4.1.2 Lottery

For lottery scheduler, when every process is created, a number of tickets is assigned for that particular process, which is currently set to 100. The number of tickets is saved as a variable in the `struct proc` as an integer when `allocproc()` is called. For our implementation, same number of tickets were assigned for every process; however, it would be also possible to assign different number of tickets for each process to prioritize some processes over others. The lottery scheduler basically goes through all runnable processes to find out the total number of lottery tickets, and then generates a positive random number smaller than the total number. For the random number, a pseudo-random number is generated using the linear-feedback shift register for 16-bit unsigned integer. The next random number is generated from the previous number, which means that changing the initial seed can ensure different results for the tests. After the random number is generated, the scheduler goes through all runnable processes one more time, checking if the process should be selected. For example, if there are 5 runnable processes with 100 lottery tickets each, there are 500 tickets total. If the random number generated is between 0 and 99, the first of the 5 runnable processes will be selected, if the number is between 100 and 199, the second process will be selected, and so on.

4.1.3 Multi-Level Feedback Queue

For our parameters, we will be using 4 priority levels, with 3 being the highest level and 0 being the lowest. The time quantums, in order of descending priority, are 1, 2, 4, ∞ timer ticks respectively.

In `struct proc` we add a variable to track the priority for each process and the number of timer ticks left

before the process reaches its time quantum. First to initialize each process's `proc` struct, in `allocproc()` we initialized each processes' priority level to 3 and their `ticks_to_go` to 1, which is the time quantum for priority 3. In `trap.c`, we decrement each process's `ticks_to_go` field at every timer interrupt.

We loop through the `proc` array that stores pointers to all of the processes in xv6. On the first pass, we only scan for processes that are `RUNNABLE` and have priority 3. If we find no matching processes, we then move onto to scanning the entire array for processes that are `RUNNABLE` and have priority 2, and so on. If at any point a process is found matching the condition, we schedule that process. When the process returns to the scheduler, we check if `ticks_to_go` has reached 0. If it has, then we lower the priority level by 1 and set the `ticks_to_go` to the next time quantum. In practice, since we cannot set a infinite time quantum for priority level 0, we just ignore this check if the process already is at priority level 0.

To maintain a round-robin ordering for each priority level, we store the index of the last process scheduled for each priority level in an array `priority_last_index`. At the beginning of each pass per priority level, we set the iterator index to the corresponding value in `priority_last_index` and iterate `NPROC` times for an whole array scan. This ensures that we will not continually schedule the same task at the start of the `proc` array, ensuring some fairness and preventing total starvation since we are not using queues in our implementation.

4.2 Instrumenting xv6 for Evaluation of Schedulers

xv6 records the number of CPU cycles since boot at a fixed, known physical memory `(CLINT\_MTIME)`, which we map to a location in virtual memory, so that the kernel can access this data while running. Metrics instrumented and recorded were printed out in a kernel syscall `print_metrics()`.

4.2.1 Latency

We measure the latency of a scheduler by measuring the number of CPU cycles it takes to finish running a process. We calculate this by subtracting the CPU cycles when a process calls `exit()` from the CPU cycles when a process was created, stored in array `createdtime[]` when `allocproc()` is called. We chose this implementation because in xv6, the `pid` of a newly created process always increments, which means that each process has a unique `pid`.

4.2.2 Responsiveness

We measure responsiveness by measuring the number of CPU cycles between process creation and first scheduling, by updating an element in the array responsetime[] when the process is scheduled for the first time. A struct variable, `int scheduled`, was created to check each time if the process has been scheduled before. In that way, we can ensure the response time is not updated twice.

4.2.3 Throughput

We created a syscall, `cycles()` that returns the CPU cycles since boot at that point. We obtain the total number of CPU cycles it takes for a fixed number of processes to run, and calculate the number of processes run per million CPU cycles.

4.2.4 Fairness

We use a variant of Jain's Fairness Index [J91] to calculate the fairness of a scheduler, where 0 is completely unfair and 1 is completely fair. Let x_i be the turnaround time for process i, and n be the total number of processes in a workload.

$$\text{Fairness} = \frac{(\sum x_i)^2}{n \sum x_i^2}$$

4.2.5 Overhead

Overhead can be measured as a ratio of the number of cycles spent in the scheduler to the number of total cycles spent. To keep track of the time spent in scheduler, every time a `swtch()` function is returned after being run with the selected process, a per-CPU variable `uint64 starttime` is updated to the current time in terms of number of cycles, so that it can be used to calculate the number of cycles in the scheduler until the next process is run. Since there are `NCPU` cores, we calculate percentage time spent in the scheduler with the equation:

$$\frac{\sum_{\text{CPUs}} \text{starttime}_i}{\text{NCPU} * \text{totaltime}}$$

Where `totaltime` is the number of cycles since boot.

4.2.6 Predictability

We measure predictability by running each workload on each scheduler 10 times, plotting boxplots and finding the interquartile range for the response times.

4.3 Implementing Test Workloads as User Programs

As mentioned before, our three workloads included a CPU-bound workload, an IO-bound workload, and another workload that was a mix of the two.

4.3.1 CPU-Bound Workload

The CPU-bound workload essentially forked 40 times, with each child process running a subtask called `looptask()`, a simple counter with no syscalls except for one print statement at the end.

4.3.2 IO-Bound Workload

The IO-bound workload similarly forked 40 times and each fork ran an IO-bound subtask called `filetask()`, namely creating and writing to a file. It was intended that the IO-bound subtasks generally take longer time than the CPU-bound subtasks, since it would more effectively show the effect of having a queue by enabling multiple CPU-bound processes to run on the CPU while the IO-bound subtask is placed on the wait queue.

4.3.3 Mixed Workload

To create a mixed workload, we forked 50 times, and for 2 of every 3 forks, we ran `looptask()`, and for 1 of 3 forks, we ran `filetask()`, since the file task takes longer to run individually. Additionally, to simulate tasks coming in at different times, the parent process would sleep every 5 forks before it started forking new tasks again.

Due to time constraints, we were not able to test priority inversion with pipe-based tasks.

5 Evaluation

We ran each of the 3 workload tests on each scheduler 10 times, and averaged each of the output metrics. In this section, we describe the key findings from each scheduler organized by each of our key metrics.

Generally, we found improvements across the board for the MLFQ scheduler. However, MLFQ performed the best with IO-heavy workloads.

5.1 Latency (Turnaround)

The biggest standout for latency is the MLFQ for IO-Bound tasks being far lower than both of the other two scheduling algorithms. However, MLFQ pays for this by having the highest turnaround time in both of the other two workloads. This is because when an IO-Bound process yields to the scheduler to wait on an I/O response, the process does not get moved down in priority and can thus be rescheduled quickly once the I/O request resolves.

Turnaround Time (millions of cycles)			
Workload Type	CPU-Bound	IO-Bound	Mixed
RR	28.62	54.34	29.49
Lottery	30.51	152.7	76.25
MLFQ	31.17	3.22	90.06

Table 1: Turnaround times of different schedulers under different workloads

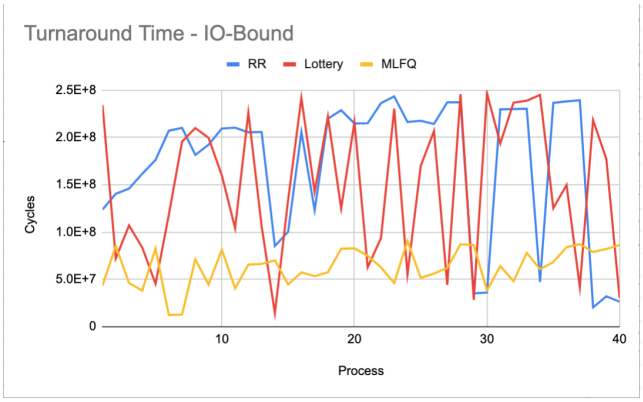

Figure 3: Turnaround time for IO-Workload

Due to the amount of CPU tasks, it seems like MLFQ struggled in both the CPU-Bound workload and the mixed workload, likely due to the CPU-Bound exercises being in different priority-levels as a result of different processes taking different amounts of CPU cycles to complete. Thus, we will have many processes reaching the bottom-most priority queues, and thus being starved out of scheduling, resulting in very long turnaround times for longer processes.

RR and Lottery performed similarly across the 3 workloads, with RR being slightly faster in CPU intensive tasks and Lottery being slightly better in IO-Bound tasks.

5.2 Responsiveness

Average Response Time (millions of cycles)			
Workload Type	CPU-Bound	IO-Bound	Mixed
RR	8.786	0.01236	3.014
Lottery	10.16	2.120	3.549
MLFQ	6.004	0.01899	0.4835

Table 2: Response times of different schedulers under different workloads

Across all three types of workloads, it is apparent

that MLFQ has the lowest average process response times. This is because, as in our implementation of MLFQ, whenever a new process is scheduled, it is automatically placed at a higher priority level, and thus once the currently running process yields in the scheduler, the new process will be scheduled before all other existing scheduled programs, especially CPU intensive tasks that have been running for a while and have thus decreased in priority.

This difference is not very apparent in the purely IO-bound task since the MLFQ will keep all of the I/O bound tasks at the highest priority level as they most likely yield to the scheduler before their time quantum is finished. This results in just one round-robin queue at the highest priority level, which makes sense comparing the time to RR.

Otherwise, lottery seems to perform slightly worse than RR on the CPU intensive tasks, mostly likely due to RR at least guaranteeing fairness for each processes' first run, while lottery may starve out some processes just due to the nondeterministic nature of the scheduler.

5.3 Throughput

Throughput (processes / million cycles)			
Workload Type	CPU-Bound	IO-Bound	Mixed
RR	0.8821	0.9212	0.5280
Lottery	0.7882	2.006	1.280
MLFQ	0.8479	2.234	1.574

Table 3: Throughput of different schedulers under different workloads

For CPU-bound processes only, the throughput did not show dramatic differences between schedulers. This is because with the absence of IO-bound processes, the CPUs will be actively running the tasks most of the time.

However, there was a dramatic improvement in the throughput of IO-bound processes, being 153% larger than the RR and and 121% larger than the Lottery scheduler. This is because an IO-bound process can stay in the wait queue in a sleep state, allowing other processees to be run by the CPU. This makes the CPU occupied by running processes more often, enhancing the throughput. For the same reason, mixed processes with MLFQ also showed an increase in throughput, being 13% larger than the RR and 72% larger than the Lottery scheduler. The improvement was not as dramatic as in the IO-bound processes, because only one third of the processes are IO-bound processes.

5.4 Overhead

Overhead (% time in scheduler)			
Workload Type	CPU-Bound	IO-Bound	Mixed
RR	15.37%	32.41%	26.74%
Lottery	11.86%	27.87%	17.29%
MLFQ	11.92%	28.41%	14.03%

Table 4: Overhead of different schedulers under different workloads

Interestingly, overhead was lowest for MLFQ, even though we expected MLFQ to have higher overhead. This was especially pronounced in the mixed task, where the CPU spend almost 3 times less time in the scheduler than for Round Robin.

MLFQ, as a result of its design, will always prioritize scheduling I/O bound tasks which are usually in the higher priority queues versus the CPU-bound tasks which are usually in the lower priority queues. Since I/O bound tasks generally spend more time blocked, finishing them faster is likely most efficient for overall CPU usage, since if we instead finished all of the CPU-heavy tasks first, the scheduler would spend a significant amount of time spinning waiting for a task to become unblocked. Thus, MLFQ overall had the lower overhead time despite having more going on in the `scheduler` function itself, as it spent less time busy spinning waiting for I/O bound tasks to finish.

However, as more and more optimizations are added onto the MLFQ like priority-boosting, the overhead should be expected to increase significantly compared to where it is now, with an additional increase in the other performance metrics. Thus, currently MLFQ just manages to keep the CPU uptime much higher than other processes, which is where the majority of the overhead wins come from.

5.5 Fairness

Fairness Index			
Workload Type	CPU-Bound	IO-Bound	Mixed
RR	0.8102	0.7493	0.7103
Lottery	0.8176	0.5729	0.5413
MLFQ	0.7452	0.5829	0.5441

Table 5: Fairness of different schedulers under different workloads

According to our fairness index, Round-Robin was the most fair while Lottery and MLFQ were similarly

low. Since MLFQ implements a priority system, it is inherently not fair, since it would prefer to starve CPU-bound processes in favor of IO-bound processes.

For Lottery, we also expect less fairness than Round Robin, because there is no guarantee that a process will be scheduled with any real-time constraints. Specifically, for a large number n, if there are n processes, and the scheduler chooses n tasks in succession with replacement, then with equal number of tickets for all processes, we expect around 38% of the processes not to be scheduled. Therefore, certain processes will be randomly starved while others will finish faster.

5.6 Predictability

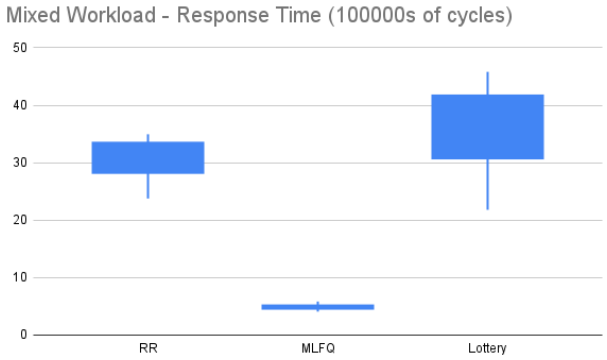

Figure 4: Box plot for the distribution of response times on a mixed workload

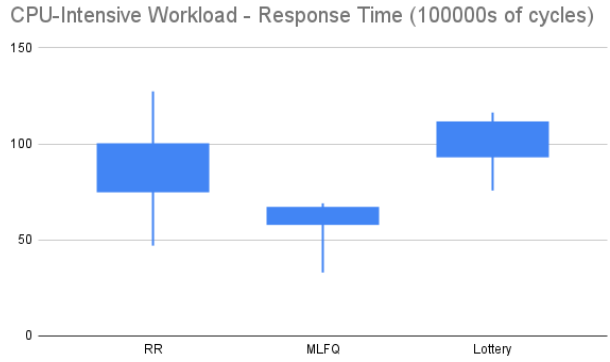

Figure 5: Box plot for the distribution of response times on a CPU-heavy workload

We find that MLFQ is the most predictable in terms of response time from our most comprehensive workload. This makes sense as it has a preferred ordering for processes while the other two scheduling algorithms do not.

We also find that the lottery scheduler a significantly worse predictability on the mixed workload than the other two schedulers. This also makes sense here, since the random nature of the lottery scheduler simply makes it less predictable especially over a smaller sample size.

6 Conclusion

In this paper, we successfully implement two schedulers (MLFQ and Lottery) on the xv6 operating system, instrument the operating system to add metrics, design workload tests simulating different combinations of IO-bound and CPU-bound processes, and evaluated the performance of the schedulers along different performance and fairness criteria.

Overall, on most performance-based metrics, MLFQ exhibit superior results in most metrics, which can mostly be attributed to the more efficient order in which MLFQ schedules tasks. It prioritizes the I/O-bound tasks that are most likely to be blocked and thus take the longest overall time before scheduling the CPU-bound tasks that are unlikely to block.

On the other hand, when it comes to fairness-based metrics, we see that the Round Robin scheduler appears to be superior, closely followed by Lottery, while MLFQ has the worst fairness. The round robin scheduler does not prioritize any task and completes all of them in turn, which in turn ensures that the variability of turnaround times between different tasks is lower. MLFQ, by prioritizing IO-bound tasks that are more likely to be blocked, will tend to favor completing these tasks first before completing the longer CPU-bound tasks, and thus more unequally distribute

We see that there is a tradeoff between fairness and performance, which occurs because performance gains often come from prioritizing certain processes over others, which inevitably leads to unequal distribution of CPU resources.

6.1 Acknowledgements

We would like to acknowledge Professor Matthew Lentz and the teaching team for their invaluable guidance in this project. We would also like to thank the creators of xv6.

References

[J91] R. Jain. The Art of Computer Systems Performance Analysis: Techniques for Experimental De- sign, Measurement, Simulation, and Modeling. *Interscience, New York*, April 1991.