Lab Report: The Priority Scheduler

User-level OS Scheduler Group 3

January 18, 2025

Milan La Rivière
m.m.la.riviere@student.vu.nl

Gideon Pol
g.h.g.pol@student.vu.nl

Bas Jansweijer
b.jansweijer@student.vu.nl

Supervisor
Hexiang Geng

Supervisor
Daniele Bonetta

ABSTRACT

Schedulers play a critical role in optimizing resource usage and ensuring efficient task management in computing systems. This report investigates user-level OS schedulers on Linux, developed using the hello-ebpf framework. We compare various custom schedulers—FIFO, Round Robin, and Priority-based schedulers—focusing on total runtime and wait time to evaluate their performance. A key hypothesis is that prioritizing IO-heavy tasks can enhance performance by reducing their wait times, enabling more efficient handling of IO operations. The experiments reveal that a Priority Scheduler with anti-starvation measures (NoStarvation) provides notable improvements for IO-intensive workloads without significantly impacting compute-heavy tasks. These findings underscore the importance of tailored scheduling strategies in optimizing task execution in diverse computing environments.

1 INTRODUCTION

This report will look into schedulers for user level os on Linux, created with the hello-ebpf framework. The schedulers will be compared on total run time, and total wait time to determine which schedulers excel in which field.

1.1 The Problem

Schedulers are important in computing and task management because they help optimize the use of resources, manage workloads, and ensure efficiency. If a scheduler is inefficient, every task that is running on that computer will be impacted. Therefore, optimizing a scheduler is always a useful improvement. The current default scheduler for Linux is called EEVDF, which stands for Earliest Eligible Virtual Deadline First. This scheduler generally provides good performance, but is not necessarily optimal in all situations. The problem that the custom-made schedulers will try to resolve regards IO operations. These operations often require tasks to block, which could result in tasks prematurely ending their allotted computing time slot. For an IO-heavy task it is thus hypothesized that a shorter wait time in combination with a smaller time slice might lead to better performance. If these IO-heavy tasks are scheduled sooner (or with a priority), they can make progress until the next IO operation, while minimally delaying the other tasks since they only require a small amount of CPU time before they block and allow other tasks to run. The proposed scheduling approach will therefore look to separate the tasks that are IO-heavy to give these tasks priority. This will hopefully provide a meaningful performance improvement for these IO-heavy tasks without hampering the execution of other tasks too much.

1.2 Report Structure

A custom scheduler called the IOPriorityScheduler has been implemented. This scheduler prioritizes tasks that use only a small part of their assigned slice time. This way, tasks that block often will have a shorter wait time. This is further explained in section 3. The first experiment with this scheduler is to find the optimal slice time parameters using coordinate descent. For a second experiment, the custom-made IOPriorityScheduler (with the best parameters) is compared to the FIFO and SampleScheduler from the original hello ebpf blog post [1].

2 BACKGROUND

The schedulers created in this report are all created using hello-ebpf.

2.1 eBPF

eBPF (Extended Berkeley Packet Filter) is a technology in the Linux kernel that allows users to run custom programs inside the kernel without modifying the kernel source code or loading kernel modules. This way core parts of the kernel can be controlled within userspace, where the loaded programs are executed safely and efficiently. This includes tasks such as networking, tracing, security enforcement, monitoring, and in this case scheduling. eBPF programs are written in a restricted subset of C, compiled to bytecode, and then verified and executed by the kernel.

2.2 hello-ebpf

Hello-ebpf is a wrapper library for eBPF implemented in Java. It exposes the basic API of eBPF along with Java helper classes and functions to promote Java's OOP nature. eBPF is exposed to the user as a C library, which does not match the hello-ebpf's Java environment. To reconcile this, hello-ebpf does not marshall eBPF function calls between Java and C, but transpiles the BPF part of the program to C for efficiency. This means only a subset of Java is possible in this part, for which the framework provides special Java-like classes and structures.

From our experience in the project, this also makes hello-ebpf quite cumbersome to use as the additional layer of abstraction introduces yet another point of failure in the build process. During both the experimentation and development phase inexplicable errors were encountered often, many of which seemed to almost randomly appear and disappear. This problem is also something the authors of hello-ebpf acknowledge in their blog post[1].

2.3 Experiment setup

All the experiments are conducted on a QEMU virtual machine running Ubuntu 24.10 with a kernel that supports custom schedulers through eBPF. The virtual machine is configured with 4GB of memory, and 4 CPU cores and itself runs on a Ubuntu laptop. To be certain that there are no false comparison and conclusions, all experiments were run on the same laptop. To benchmark the schedulers, the Renaissance benchmark suite [2] was used.

3 SCHEDULER DESIGN

As stated in subsection 1.1, the scheduler needs to separate the tasks that deal with IO operations from the tasks that do not. This would allow the scheduler to reduce the wait time of the IO tasks by giving them priority. The main assumption of the design is that the tasks that run IO operations will block more often as they have to wait on slow IO operations. Based on this assumption, we have designed two schedulers that modify the standard Round Robin scheduling approach. Before implementing the schedulers described in this report, we thus decided to also implement a Round Robin scheduler.

3.1 Initial priority scheduler

The initial version of the scheduler prioritizes tasks that have low slice time usage. Instead of a normal Round Robin scheduler which has a single FIFO queue, this scheduler contains two queues, one 'normal' queue and one 'priority' queue. Initially, all the tasks are put in the normal queue. As tasks are scheduled, we then track what percentage of the allotted slice was used by the task. When the task is enqueued, the scheduler then checks whether the slice time usage was under a specified threshold and put the task in the priority queue if this is the case. This scheduler is implemented in IOPrioSched.java and can be configured with the following flags:

- *slice_time*: the length of a single slice in nanoseconds.
- *slice_time_prio*: the length of a slice given to tasks coming from the priority queue in nanoseconds.
- *prio_slice_usage_percentage*: The percentage of the normal slice time that a task can use to still be classified as an IO-heavy task. When tasks use more than this percentage, they are put in the normal queue. In this report this percentage was always set to 5%.

3.2 Using a weighted average

The initial approach described above only tracks the slice usage of the latest slice. We hypothesized that this could incorrectly classify some of the tasks due to the random variations in slice time usage. Some tasks might therefore be able to receive priority if they encounter an early blocking operation, even if in general this task tents to use its entire slice. To prevent these misclassifications, an additional version of the scheduler was created which bases the decision of putting a task in the normal or priority queue on more than just the usage of the last slice. To prevent having to store the slice time usage of the last *n* slices, weighted average is used where the average slice time usage after each slice is calculated according to the following formula:

$$average\_usage_{t+1} = c \cdot average\_usage_t + (1 - c) \cdot last\_slice\_usage$$

By then comparing the *average_usage* instead of the usage of the last slice time this version of the scheduler ensures that only tasks that consistently encounter blocking operations early on are put in the priority queue. This additional version of the scheduler was implemented in PrioSchedWeightedAvg.java and introduces two additional configurable settings:

- *weighted_avg_mult*: the value of the constant c from the formula. This was set to 0.99 in all experiments.
- *initial_usage_percentage*: the value that the *average_usage* is initialized at ($average\_usage_0$). This value is set to 1 (i.e. 100%) for all experiments in this report.

3.3 Anti starvation measures

One fundamental issue with the use of a priority queue is the possibility for starvation. If the criteria for being put in the priority queue are too generous, some tasks which are put in the normal queue might never receive CPU time due to there always being tasks with priority. To prevent this, one last version of the weighted average scheduler was created, which prevents the same task from being put in the priority queue twice in a row. This scheduler has been implemented in the file PrioSchedWeightedAvgNoStarvation.java.

4 EXPERIMENTAL RESULTS

4.1 Finding the optimal slice time

To achieve good performance the previously discussed parameters must be set to sensible values that let the scheduler operate effectively. Due to time constraints, the choice was made to systematically investigate a subset of the available parameters. Through experimentation, the slice time and priority slice time parameters seemed to affect performance most drastically, and were thus chosen for search. To further reduce the necessary benchmarking time, we used the coordinated search strategy to search the parameter space. To this end, the priority slice time was first fixed at 10MS, investigating how the regular slice time impacts performance. Afterwards, the optimal slice time was used to find the best priority slice time. Note that for this subsection, only the Dotty, db-shootout, reactors and page rank benchmarks were used in the interest of saving time. These specific benchmarks were chosen to provide a good mix of IO-heavy and compute heavy benchmarks.

Scheduler	5M	10M	20M	40M	80M
RoundRobinSched	39.9	32.5	29.8	29.9	29.6
IOPrioSched	36.3	35.3	31.8	30.7	30.2
PrioSchedWeightedAvg	36.5	34.4	32.1	32.0	30.1
PrioSchedWeightedAvgNoStarvation	35.9	33.7	31.2	30.4	30.0

Table 1: Combined times of the benchmarks for each slice time (seconds)

From the two tables above a clear trend for all schedulers emerges. As we increase the slice time, the execution time decreases. This is likely because the amount of context switching is minimized when using longer slice times. However, the wait time (i.e. the time tasks need to wait in the queues) also increases. Because of this tradeoff, there is thus no clear 'best' slice time. In further experiments, we therefore opted to use a slice time of 20 million nanoseconds as this

Scheduler	5M	10M	20M	40M	80M
RoundRobinSched	114	125	153	217	270
IOPrioSched	123	120	156	201	0
PrioSchedWeightedAvg	127	125	168	216	0
PrioSchedWeightedAvgNoStarvation	128	134	165	223	297

Table 2: Combined wait times for each slice time (milliseconds)

was the default within hello-ebpf. Using this slice time, the slice time settings for tasks that are given priority are explored below:

Scheduler	10M	20M	40M	80M
IOPrioSched	32.6	33.7	31.7	32.5
PrioSchedWeightedAvg	32.1	31.6	31.6	31.8
PrioSchedWeightedAvgNoStarvation	32.0	31.3	31.6	30.9

Table 3: Combined times of the benchmarks for each priority slice time (seconds)

Scheduler	10M	20M	40M	80M
IOPrioSched	153	136	162	165
PrioSchedWeightedAvg	168	142	159	171
PrioSchedWeightedAvgNoStarvation	160	151	132	139

Table 4: Combined wait times for each priority slice time (milliseconds)

When it comes to changing priority slice time (tables 3 and 4), it seems the relationship with the runtime and wait time is not as clear. Especially for the NoStarvation version of the priority scheduler, the wait time seems to decrease with higher priority slice times. There is no clear explanation for the observed trends, exploring why the priority slice time has such different relations to especially the wait time could be an avenue for further exploration.

4.2 Custom-made schedulers

When looking at the comparison on the full benchmark (as shown in Figure 1, the NoStarvation version of the implemented scheduler clearly shows to be more consistent, and on average faster than the FIFO scheduler.

A possible explanation for this is that IO is more reliable in the NoStarvation policy, as it is prioritized when slice usage is low. As a result, the system spends less time waiting for IO and therefore also limits the potentially unreliable performance it inherits from it.

When looking at individual runs, it becomes apparent that the NoStarvation scheduler is especially fast for benchmarks that are IO-heavy, as was intended. Inspecting Figure 2 — db-shootout is an IO-heavy benchmark — reveals that NoStarvation is clearly faster than the other schedulers, but when looking at Figure 3 — a computer heavy benchmark — all schedulers execute at about the same speed.

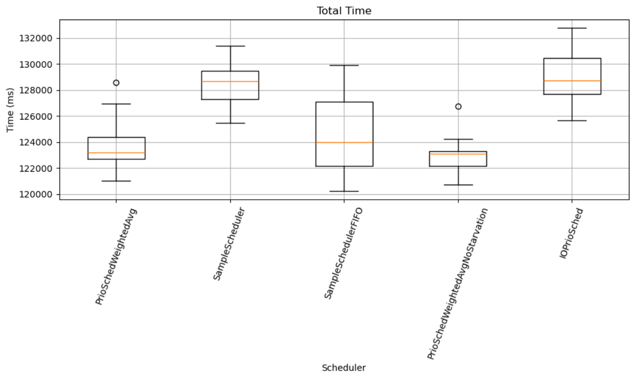

Figure 1: Comparison between all relevant schedulers on the full Renaissance benchmark (20 runs, top 25 percentile removed due to outliers)

It is especially interesting that the NoStarvation policy is significantly faster than the WeightedAvg policy. On its own, WeightedAvg does not deal effectively with the workload, with performance being roughly equal to that of the FIFO scheduler. However, NoStarvation significantly outperforms all other schedulers, showing that contention and starvation are important issues that are dealt with in this policy as intended.

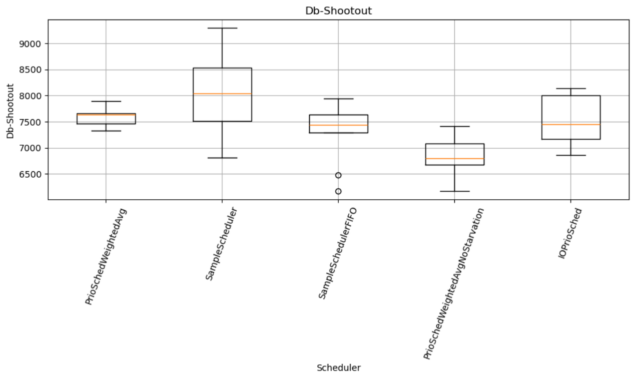

Figure 2: Comparison between all relevant schedulers on the db-shootout benchmark (20 runs, top 25 percentile removed due to outliers)

Inspecting individual run Figure 3 - a compute-intensive benchmark - shows comparable performance for all schedulers. This can be explained by the fact that compute-heavy tasks are likely to use a significant part of their slice, thereby mostly foregoing the priority queue mechanism. As a result, the priority schedulers still provide good performance for tasks not focused on IO, which is in line with the objective.

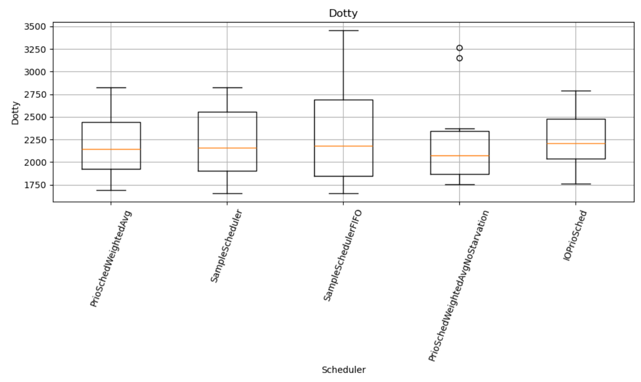

Figure 3: Comparison between all relevant schedulers on the dotty benchmark (20 runs, top 25 percentile removed due to outliers)

5 CONCLUSION

Though the results are not completely conclusive, it does seem like the NoStarvation version of the scheduler might provide a performance increase when running an IO-heavy benchmark like db-shootout. Interestingly, the vanilla and weighted average versions of the priority scheduler did not seem to provide the same performance increase, leading to the hypothesis that the starvation of tasks not (yet) classified as IO-heavy indeed has an impact on the performance. Apart from the performance improvement for IO-heavy tasks the initial goal of our scheduler also states that more compute-heavy tasks would likely not be impacted much by giving priority to the IO-heavy tasks. This seems to indeed be true, as all schedulers have similar performance on the compute-heavy dotty benchmark.

6 DISCUSSION

6.1 Priority threshold

Since we use the percentage of the normal queue slice time usage when the slice length of the priority tasks is too small, they cannot ever come above this threshold and are thus stuck in the priority queue even if they always use their entire priority slice each time. In general, this flaw in the design doesn't pose much of an issue as there is no reason to use such small priority slice time values, but it is still something to keep in mind when choosing the slice time values.

6.2 Further exploration of the parameters

As stated in section 3, some provided parameters were kept constant during all tests in this report. This was due to a lack of time. With more time, it would have been interesting to explore changing these values. For example, trying out different *prio_slice_usage_percentage* values could yield interesting results that could potentially improve the scheduler. Otherwise, changing the calculation parameters for the weighted average could also yield interesting results. Additionally, the slice-time exploration uncovered a trade off between the wait time and run time. Further exploration of the relationship of these metrics and the slice time settings might allow us to choose a more optimal trade-off.

6.3 Testing the main assumption of the scheduler design

The main assumption of proposed schedulers is that IO-heavy tasks will block more often. This assumption seems intuitive since, each time an IO operation is encountered, a task will likely have to wait (and thus block). For completeness, it would still be beneficial to test this assumption experimentally by comparing the behavior of IO-heavy and compute heavy tasks.

6.4 Remaining remarks

We make multiple observations as to why we have gone beyond the traditional requirements of the assignments.

Firstly, the hello-ebpf framework is difficult to work with. It is very brittle (as was also warned by the blogpost). At the end of the project, it would not run anymore on our laptops after not having touched it over the Christmas break, seemingly because of a kernel update. This required additional setup time and took away from the implementing and experimenting we could do. Because of this some of the experiments we wanted to do, such as comparing against our round robin implementation and a clearer reproduction of the experiment from the blog post were unable to be completed. This was because Renaissance stopped working for these schedulers, which we could not fix with the time left after setting up the environment again. The code, as well as this report, is openly available at this GitHub repository: https://github.com/BasJansw/DS-scheduler-lab. The time table can be found here: https://docs.google.com/spreadsheets/d/ 1lCkIxTOf-74NRkhexyLwy07DB-g0kFHnhszDtTkYWfg

REFERENCES

[1] Johannes Bechberger. *Hello eBPF: Writing a Linux scheduler in Java with eBPF (15) - Mostly nerdless — mostlynerdless.de.* https://mostlynerdless.de/blog/2024/09/10/hello-ebpf-writing-a-linux-scheduler-in-java-with-ebpf-15/. [Accessed 15-01-2025].

[2] Aleksandar Prokopec et al. *Renaissance Benchmark Suite.* https://renaissance.dev. A modern benchmark suite for the Java Virtual Machine, focusing on realistic, concurrent, and parallel workloads. 2019. URL: https://renaissance.dev.