CPU governance and scheduling using machine learning implemented on a Linux kernel

Fall 2020 semester report

Jonathan Gao

Cornell University

December 16, 2020

Introduction

As an exploratory dive into the Linux kernel, I plan to modify the Linux scheduler to use learned decisions from a machine learning model. At this project's inception, there were many potential options for where a machine learning model may be practical. As an introduction, I will discuss several of those options, the research done in regards to those paths, and the planned route onwards.

The Scheduler

The scheduler in an operating system manages the job of "scheduling" processes. A operating system deals with numerous processes, all wanting to perform some task using CPU and memory. However, a modern computer may often only have a few cores or processing units, each only capable of running one process at once. Therefore, the operating system must decide which process gets to run on a processor for how long.

Scheduling decisions can be based on a number of parameters. Some simple scheduler implementations include the First In First Out (FIFO) scheduler, where processes get CPU runtime at a first-come-first-served basis. However, this is scheduler is clearly suboptimal, as we care about parameters such as process latency, process deadlines, and overall process throughput. Thus, we have the notion of a CPU quantum, where a process given a CPU quantum will be given a short CPU runtime, say 20ms, before it is taken off and put back onto the process queue.

CPU quanta introduced to the FIFO scheduler becomes the Round Robin scheduler, where the processes get run in FIFO order, and then get enqueued back onto the queue if it does not complete execution within its quantum. The terminology for the notion of a period of execution for a process has been referred to by several names, but has since become the primarily point of focus for modern CPU schedulers.

Scheduling a process is no trivial task, and extensive work has been done to implement an optimal scheduler. Most modern operating systems implement one of the two following schedulers, the Multi-level Feedback Queue scheduler (MLFQ) or the Completely Fair Scheduler (CFS). For example, Windows and macOS use versions of the MLFQ scheduler, whereas Linux has primarily used the CFS scheduler. I considered both of these schedulers for possible machine learning modifications, both of which will be discussed in the following sections.

MLFQ schedulers

Multi-level feedback queue schedulers use several queues rather than the single process queue discussed earlier. The "multi-level" queues correspond to different levels of time quanta on the CPU. The highest queue has the shortest quanta on the CPU, with each subsequent queue having longer quanta. With several queues, processes placed onto these queues will run for the time quantum specified by its queue. The number of queues and the range of time quanta will vary, but this structure allows for processes to be classified into groups based on their needs [1]. Using this model, we can prioritize interactive and I/O bound processes by introducting priorities, preemption, and feedback across the queues.

With this model, a process enters the MLFQ scheduler at the top queue with the shortest time quantum. If this process finishes execution within this time quantum, it will terminate and exit the scheduler. Otherwise, if the process relinquishes control of the processor within this time quantum, it will be placed back onto the queue it was dequeued from. However, if this process requires more execution time, it will be preempted at the end of its time quantum and enqueued onto the queue with the next largest time quantum. This process will continue until the process either ends up in a queue optimal for its burst time, the time quantum that it requires before it relinquishes the processor, or ends up in the last queue [2, 3].

Notice that this design prioritizes short processes that quickly relinquish the processor after execution. However, there are a few concerns with this design. Processes can potentially be "starved" from CPU time, where shorter processes constantly come in and prevent those in the lower queues from executing. This can be solved by periodically reinserting lower processes back into the top of the queues. Starvation, along with other concerns such as priority inversion, is one of the disadvantages of using a MLFQ. Much work has been done to mitigate these disadvantages, and the versions in both Windows and macOS likely have such modifications.

Potential routes for the MLFQ scheduler for machine learning

With the levels of a MLFQ scheduler being a classification problem, I had considered implementing a support vector machine classifier to predict which queue a process will end up in, avoiding the need for feedback and overhead for preemption and moving queues. Rather, processes will be enqueued onto the queue in which the classifier predicts through learned decisions on process metadata.

Considering that MLFQ schedulers require all processes to "bubble" down to their respective queues, a classifier may be able to quickly and efficiently learn which queue a process should be enqueued onto. With a perfect classifier, this can simplify the MLFQ scheduler to a simpler Multi-level queue scheduler, where processes remain on their queues until termination.

Using support vector machines, the classification of processes can be trained over time using data from a multi-level feedback queue. Once trained, scheduling a process using the SVM will still be O(1) as SVM classifiers simply use a matrix multiplication to perform classification. The classifier can also continously be trained over new data collected from the MLFQ, as the classfier should minimize feedback but will not eliminate it completely.

However, the Linux kernel does not implement a Multi-level Feedback Queue Scheduler, thus any modifications must be done on my implementation of a MLFQ scheduler. This may have been possible, as the real-time `SCHED_RR` scheduler offers 99 priority levels with a round robin scheduling policy. A MLFQ scheduler could have been built as a wrapper around this, but was ultimately decided against in preference for routes with the CFS scheduler that will be mentioned.

Several papers surrounding work with machine learning optimizations for MLFQ schedulers has shown some promising results, but seem relatively limited in scope. For example, this implementation of a SVM classifier for a MLFQ [4] uses only three queues in their own implementation, with some simulation giving questionably improved results. Another paper showed promising results for using reinforcement learning to similarly optimize how feedback in the MLFQ scheduler performs [5]. This paper notes higher CPU utilization across multiple cores as well as lower turnaround and completion time of a batch of processes. While these papers focus around real-time operating systems, I would imagine that standard interactive operating systems will benefit more from machine learning implementations due to greater variety of processes.

The CFS

The Completely Fair Scheduler aims to be completely fair, as the name suggests, for all processes regardless of the process [6]. The underlying mechanism lies in the red-black tree that all processes get inserted to. CFS does not use CPU quanta, but rather tracks the runtime of every process and aims to be divide up CPU time fairly. For example, as from this documentation [6], CFS will schedule two processes such that they both receive 50% of CPU runtime. The ordered red-black tree maintains a balanced sorted tree of process runtime, where at each scheduler tick

the process with the least runtime will be picked. With a bit of granularity to improve cache performance and reduce overhead, the CFS guarantees every process will eventually be able to run.

This scheduler implementation only offers one tunable parameter: the granularity at which a process must overtake the minimum runtime of all processes before another process is selected. Since Linux 2.6, the CFS has remained the default scheduler implementation, though with several changes and improvements since its inception.

Then, without much to change and tweak in this scheduler, what does machine learning offer for CFS? Despite no tunable parameters for the scheduler itself besides granularity, CFS has features that allow some tweaking of how processes behave relative to one another.

Niceness

CFS revamps a priority-like value for every process called niceness [7]. Process niceness comes from the idea of how "nice" a process is to other processes, or how willing they are to give up CPU time for other processes. A process has nice values ranging from -20 to +19, with all processes starting with a default nice value of 0 or otherwise inherited from its parent process. Niceness values indicate how much CPU time a process will consume relative to other processes' niceness. Processes that all have niceness values of 0 will share CPU time equally, whereas a +19 process will only have 5% CPU utilization.

Nice values for processes can be adjusted, almost like how processes are shifted around the queues in MLFQ, to give more CPU time to certain processes and less to others. Rather, we can use these to improve interactivity and improve response time based on process type. Currently, niceness is not set by any operating system feature, thus is a feature available for users to adjust their process behavior.

Additionally, there is a grouping feature introduced to the CFS to further improve interactivity [8, 9]. Control groups in Linux group processes together, where each group has set assigned resources. Instead, a process nice value only affects CPU runtime within a group. The idea of control groups was introduced to Linux to control the resources of groups of processes, where a cgroup could be defined to limit the CPU utilization, memory usage, and other resources such as network bandwidth. While cgroups do not necessarily have a notion of niceness, niceness of a cgroup is implemented through CPU quota and period values. The quota of a cgroup is

how much CPU runtime processes within the cgroup get within a period [10]. These are usually defined in microseconds. We see that we can similarly limit a cgroup of processes to consume only a percentage of CPU utilization. Cgroups do get a little more complicated, but ultimately are primarily used for containerization of processes.

How does containerization of processes improve interactivity? Consider the following example, where we have two processes of equal niceness. One of these processes has is single-threaded, while the other spawns 8 threads. Because Linux treats threads identically to processes, we see that the multithreaded process will end up taking 90% of CPU utilization. Rather, if we constrain a process and its subprocesses within a cgroup, we maintain interactivity such that threaded processes do not overwhelm the CPU.

Autogrouping

An autogrouping feature was introduced in Linux 2.6, where processes are grouped automatically. From the autogrouping documentation under the `sched` documentation [11], we see that processes spawned under a new session automatically get placed into its own kernel "task group." Despite the same notion of a task group, the autogrouping feature does not use the existing cgroup infrastructure. Unfortunately, this makes all of the different notions of task grouping very confusing. The autogrouping feature is enabled on some kernels and disabled on others, but can be checked by running `cat /proc/sys/kernel/sched_autogroup_enabled`. Task groups are created on a new session created by `setdsid()`, where all tasks spawned in this group inherit the group of the parent process. There is also another notion of process groups, where each group has its own group ID used by other syscalls for job control and credential management [12, 13]. The autogroup functionality uses the same notion of niceness, where a nice value is applied to the entire group. The nice values are found under the `/proc/$PID/autogroup` and can be changed to modify the niceness of the group.

Users have noted that the inclusion of the autogroup feature essentially nullifies the traditional niceness of processes, as now the `nice` command only gets reflected in the behavior within an autogroup [14].

Planned modifications

Using cgroups instead of processes

- instead, I want to focus on modifying the niceness values of cgroups formed by the autogrouping feature

Regression to determine niceness value for cgroup

- I was thinking that I could use ideas from MLFQ, using parameters such as execution time, estimated deadlines, etc. to determine the niceness of the process.

- process information can be found in the pseudo filesystem located under `/proc/`, along with all of the information regarding cgroups and autogrouping.

Modifying the autogroup feature

- rather than modifying the niceness of a group after its inception, I can modify the autogroup feature to decide on the niceness value.

Other notes

I can implement a syscall on a timer that "renices" all processes every so often, such that we can get dynamic nice values depending on certain factors.

The niceness of processes isn't really used, which is curious as niceness directly affects CPU utilization. Perhaps we can have a similar level system of niceness, where processes will shift between different levels of niceness.

I need to look further into sleeper fairness, which affects I/O bound processes and sleeping processes.

Data Collection

Picking workloads

- thinking about using kernel compilation time to see if we see any changes to CPU behavior and process execution

Averaging computation times

- because Linux has many background processes running, I will average computation times and results over many executions to try to constrain other variables

Quantifying performance and interactivity

- thinking about having a user task i.e. video playback run while the kernel compile runs

References

[1] "Multilevel queue," *Wikipedia*, Nov. 2018.

[2] "Multilevel feedback queue," *Wikipedia*, Sept. 2020.

[3] "Difference between Multilevel Queue (MLQ) and Multi Level Feedback Queue (MLFQ) CPU scheduling algorithms." https://www.geeksforgeeks.org/difference-between-multilevel-queue-mlq-and-multi-level-feedback-queue-mlfq-cpu-scheduling-algorithms/, July 2020.

[4] S. Satyanarayana, P. Sravan Kumar, and G. Sridevi, "Improved Process Scheduling in Real-Time Operating Systems Using Support Vector Machines," in *Proceedings of 2nd International Conference on Micro-Electronics, Electromagnetics and Telecommunications* (S. C. Satapathy, V. Bhateja, P. S. R. Chowdary, V. S. Chakravarthy, and J. Anguera, eds.), vol. 434, pp. 603–611, Singapore: Springer Singapore, 2018.

[5] D. R. Rinku and M. AshaRani, "Reinforcement learning based multi core scheduling (RLBMCS) for real time systems," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 10, p. 1805, Apr. 2020.

[6] "Documentation/scheduler/sched-design-CFS.txt - kernel/msm - Git at Google." https://android.googlesource.com/kernel/msm/+/android-msm-bullhead-3.10-marshmallow-dr/Documentation/scheduler/sched-design-CFS.txt.

[7] "Influence scheduling priority with nice and renice," Nov. 2006.

[8] T. Heo, "Control Group v2." https://www.kernel.org/doc/Documentation/cgroup-v2.txt, Oct. 2015.

[9] "Control Groups — The Linux Kernel documentation." https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html.

[10] G. Seltzer, "Understanding cgroups," Nov. 2018.

[11] "Sched(7) - Linux manual page." https://man7.org/linux/man-pages/man7/sched.7.html.

[12] "Credentials(7) - Linux manual page." https://man7.org/linux/man-pages/man7/credentials.7.html.

[13] E. W. Troan and M. K. Johnson, "The Process Model of Linux Application Development," July 2005.

[14] nburgin, "Why nice levels are a placebo and have been for a very long time, and no one seems to have noticed," Sept. 2019.