A Visualization of the Completely Fair Scheduler

Thomas McKanna
CS 3800 - Operating Systems
Missouri University of Science and Technology
December 6, 2019

***Abstract*—This project seeks to provide a visual aid in learning the mechanisms by which the Completely Fair Scheduler (CFS), the scheduler used by Linux kernel, works. Unlike the majority of process scheduling algorithms, the CFS does not use a queue data structure, but instead uses a Red-Black tree. Since it is simple to mentally picture a queue, it is often easy to gain an intuitive understanding of how most scheduling algorithms work. But it is not easy to visualize a Red-Black tree and thus to gain an intuitive understanding of the CFS. To assist in gaining an understanding of the CFS, a web application has been developed which allows the user to simulate and visualize the CFS as it schedules processes.**

I. Background

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput, and processor efficiency [1]. There are many different scheduling algorithms to pick from, and each has its own benefits and drawbacks.

The Linux kernel has employed a number of scheduling algorithms over the years. Each algorithm proved to have some sort of deficiency which would give rise to the next algorithm. But with the arrival of the Linux kernel version 2.6.23 in 2007 came the Completely Fair Scheduler (CFS), which remains the scheduler for non-real-time processes to this day [2].

II. Introduction

When a software developer wants to create a thread in Linux, one of the parameters they must specify is what *scheduling class* the process should have. There are three scheduling classes to choose from:

1) `SCHED_FIFO`: for real-time threads (will use a First-In-First-Out scheduler)
2) `SCHED_RR`: for real-time threads (will use a Round Robin scheduler)
3) `SCHED_NORMAL`: for all other non-real-time threads (will use CFS)

As can be seen, the only scheduler for non-real-time processes is the CFS. Therefore, the majority of programs that the average person uses, such as web browers and text editors, are scheduled using CFS.

The creator of CFS, Ingo Molnar, summarized by saying that "CFS basically models an ideal, precise multitasking CPU on real hardware". When Molnar says an "ideal, precise multitasking CPU", he is referring to an imaginary single-core CPU which is capable or running multiple processes at the same time (in parallel), giving each process an equal share of processing power (not time, but power) [2]. For instance, if a single process is running, it would receive 100% of the processor's power. If two processes are running, each would receive 50% of the power. And with four processes running, each would get 25% of the power. By dividing up power evenly across all processes, this CPU would be "fair to all the tasks running on the system.

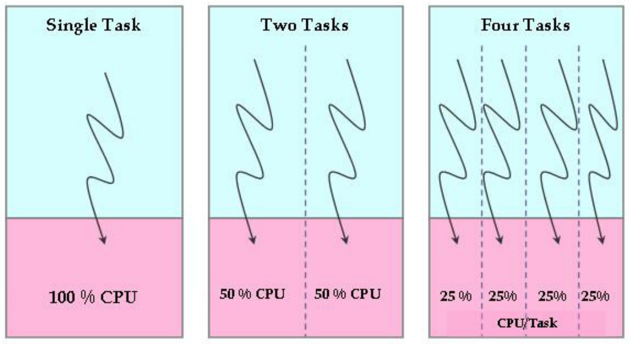


Fig. 1. An ideal precise multitasking CPU - each task runs in parallel and consumes equal CPU share

Such a processor does not actually exist, since a single-core CPU can only execute one instruction at a time; this is inherently unfair, since one process is getting 100% of the CPU while all other processes are getting 0% of the CPU. The CFS tries to emulate such a processor in software by keeping track of the fair share of the CPU that would have been available to each process in the system. The process with the longest wait time (i.e. with the gravest need of the CPU) is scheduled to be run next. More details about what sort of information is tracked and how long a process gets to execute when chosen will be provided in the algorithms section later on.

Unlike the vast majority of scheduling algorithms, the CFS employs a Red-Black tree to keep tabs on each of the available processes (most other algorithms use queues). In order to understand the mechanisms by which the CFS works, it is helpful to have a general idea of what a Red-Black tree is and what makes the data structure useful. A Red-Black tree is a ‘self-balancing’ binary search tree which conforms to five special rules. By following these rules, Red-Black trees guarantee that basic operations, such as insertion and deletion, take $O(\lg n)$ in the worst case [3]. These five rules will be stated in the algorithms section of this report.

III. Algorithms

This section outlines the implementation of the CFS and gives a basic outline of the properties of a Red-Black tree, which is the primary data structure that the CFS uses.

A. Red-Black Tree

Red-Black trees are self-balancing binary trees. By self-balancing, it is meant that after an insertion or deletion, the nodes of the tree may be rearranged so that the number of nodes on the left-hand of the tree is roughly the number of nodes on the right-hand side of the tree. There are five properties which define a Red-Black tree:

1) Every node is either red or black.
2) The root is black.
3) Every leaf node is black.
4) If a node is red, then both of its children are black.
5) For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

By following these properties, the tree is guarenteed to maintain a fairly balanced shape and to achieve $O(\lg n)$ complexity for insertion and deletion operations. As part of this project, a Red-Black tree data structure was implemented in JavaScript and is utilized in the simulation.

B. Completely Fair Scheduler

The CFS uses a Red-Black tree in which the nodes represent each of the processes. Specifically, the Linux kernel defines a `sched_entity` structure to encapsulate each process. The nodes are indexed by processor “execution time” in nanoseconds. The term “execution time” is put in quotes because the CFS allows the developer to assign a *nice* value to each process, which impacts how quickly time appears to pass for the process (this allows certain processes to receive a larger share of CPU time). Since “execution time” is dependent on the nice value, it is more accurate to say that nodes are indexed by their *virtual runtime*, which is stored in the Linux variable `vruntime` for each process.

A simplified sketch of the CFS algorithm is as follows:

1) The left-most node of the scheduling tree is chosen and sent for execution.
2) If the process completes execution, it is removed from the system and the scheduling tree.
3) If the process reaches its maximum execution time or is otherwise stopped (either voluntarily or via interrupt), it is reinserted into the scheduling tree based on its newly-updated spent execution time.
4) The new leftmost node will then be selected from the tree, starting a new iteration.

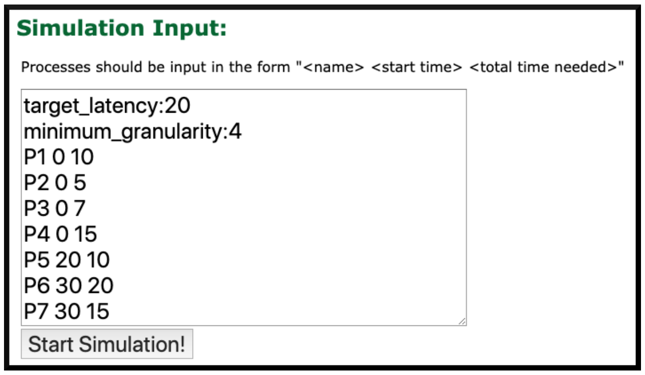
There are two primary parameters by which the CFS can be tuned and which may help clarify how the CFS actually works. The first is *target latency*, which is the minimum amount of time required for every runnable task to get at least one turn on the processor. Ideally, this value is infinitely small, but in the real world this value is usually 20ms . Once a task gets the processor, it runs for its entire weighted $\frac{1}{N}$ slice before begin preempted by another task, where N is the number of tasks that are waiting to be scheduled. For instance, if there are two tasks, each gets $20\text{ms} \times \frac{1}{2} = 10\text{ms}$ to run.

To account for context switch overhead, there is a minimum amount of time that any scheduled process must run before being preempted. This value is known as the *minimum granularity*, which is the second parameter which can be used to tune the CFS. This value is typically 4ms.

IV. Results

In order to gain a better intuition for how CFS works, a visualization-based web application was created. The web app is written with JavaScript, HTML, and CSS. The visualizer builds off of the work of David Galles, a professor at the University of San Francisco [4]. You can find the source code for the project at github.com/Thomas-McKanna/Data-Structure-Visualizations.

The web app allows the user to simulate the behavior of the CFS by inputting process information along with the target latency and minimum granularity. For each process, an identifier, start time, and end time must be provided.



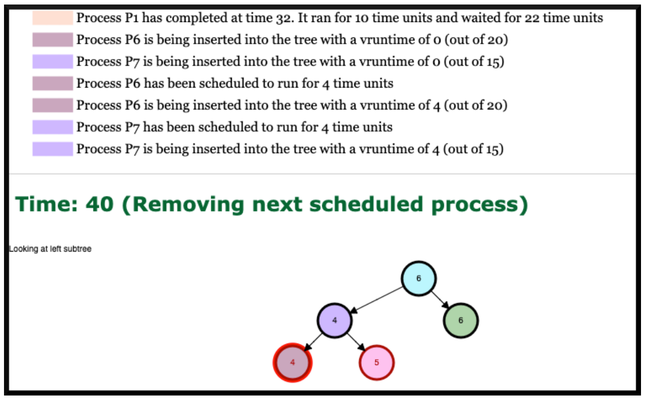
Simulation Input:

Processes should be input in the form "<name> <start time> <total time needed>"

```
target_latency:20
minimum_granularity:4
P1 0 10
P2 0 5
P3 0 7
P4 0 15
P5 20 10
P6 30 20
P7 30 15
```

Fig. 2. The user can customize the simulation parameters to their liking.

After clicking "Start Simulation", the visualization begins. As the simulation progresses, status messages are provided which indicate which process has been scheduled, how long it has been scheduled for, and how many more units of time the process will need to execute for. Below the status messages is a visualization of the tree itself. Each process is given a unique color so that the user can quickly determine which node in the tree a status message is referring to. The nodes in the tree contain a number, which is the *virtual runtime* of the process (how many time units it has been scheduled to run so far).



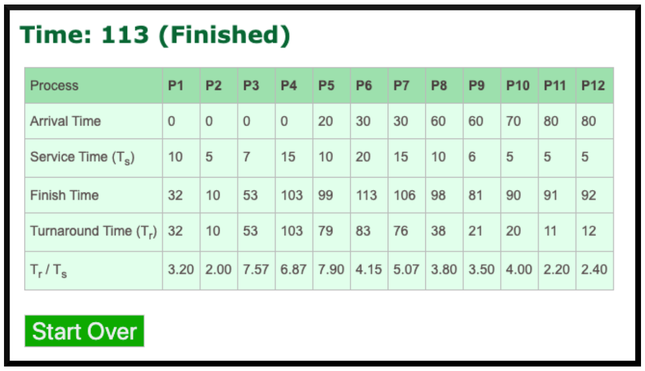
```
Process P1 has completed at time 32. It ran for 10 time units and waited for 22 time units
Process P6 is being inserted into the tree with a vruntime of 0 (out of 20)
Process P7 is being inserted into the tree with a vruntime of 0 (out of 15)
Process P6 has been scheduled to run for 4 time units
Process P6 is being inserted into the tree with a vruntime of 4 (out of 20)
Process P7 has been scheduled to run for 4 time units
Process P7 is being inserted into the tree with a vruntime of 4 (out of 15)
```

Time: 40 (Removing next scheduled process)

Looking at left subtree

Fig. 3. A screenshot of the running simulation; each insertion, deletion, and rebalancing of the tree are animated.

Once the simulation completes, a grid is presented to the user which provides statistics on the simulated processes. These statistics include arrival time, service time, finish time, turnaround time, and normalized turnaround time. The user has the option to restart the simulation with different parameters.



Time: 113 (Finished)

Process	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Arrival Time	0	0	0	0	20	30	30	60	60	70	80	80
Service Time (T_s)	10	5	7	15	10	20	15	10	6	5	5	5
Finish Time	32	10	53	103	99	113	106	98	81	90	91	92
Turnaround Time (T_r)	32	10	53	103	79	83	76	38	21	20	11	12
T_r / T_s	3.20	2.00	7.57	6.87	7.90	4.15	5.07	3.80	3.50	4.00	2.20	2.40

Fig. 4. Once the simulation completes, the user is presented with statistics on the simulation.

V. Conclusions

The mechanisms of the CFS are quite interesting if the time is taken to understand them. While at first the use of a Red-Black tree for process scheduling seems strange, the method proves to be a clever and well-performing. The CFS is fairly good for I/O bound processes since such processes are often in a blocked state and thus their virtual runtime remains low, causing them to remain on the left side of the tree. Processor bound processes are also treated quite well, because once such processes are scheduled, they are given extra execution time to make up for how long they waited. I hope that this visualization will assist students in the future who want to gain a better understanding of how the CFS operates.

References

[1] W. Stallings, *Operating Systems Internals and Design Principles*. Hoboken, New Jersey: Pearson Education, 2018.

[2] C. Pabla, "Completely fair scheduler," *Linux Journal*, August 2009.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2009.

[4] D. Galles, "Data structure visualizations." https://www.cs.usfca.edu/~galles/visualization/source.html.