Exercise #4

Completely Fair Scheduler (CFS)

CS 431

Pantic

Kristin Adachi

knadachi@cpp.edu

February 6, 2016

The completely fair scheduler (CFS) is a process scheduler that was merged in the 2.6.23 release of the Linux kernel in October 2007. It was developed by Ingo Monar based on the Rotating Staircase Deadline Scheduler (RSDL) by Con Kolivas. The algorithm for the CFS was based on the idea of an ideal multi-tasking processor. Its purpose is to handle the CPU resource allocation for executing processes and maintain balance or fairness in providing processor time to tasks. In general, this means that every process should be given a fair share of the processor. The overall share will grow and shrink with the total number of running processes, and the relative share is inflated or deflated based on the tasks' priorities. The CFS also aims to maximize overall CPU utilization while also maximizing interactive performance. Additionally, it introduces the concept of "virtual runtime" which is a task's actual runtime normalized to the total number of running tasks. This means that the smaller a task's virtual runtime, the higher its need for the processor. The CFS further maintains fairness by including the concept of sleeper fairness which ensures that tasks that are not currently runnable receive a comparable share of the processor when they eventually need it.

The Linux operating system was designed to handle all types of tasks including CPU-bound and I/O bound tasks. These classifications imply that some threads spend a lot of time using the CPU to do computations, and others spend a lot of time waiting for relatively slow I/O operations to complete. Because the Linux OS handles both types, it needed a scheduler that was responsive for I/O bound tasks and efficient for CPU-bound tasks. The tasks running on Linux can also be classified as real time which means that they have strict timing requirements and priority over any other task on the system. As a result, the scheduler would need to account for this classification as well.

The early 2.6 Linux scheduler, called the O(1) scheduler, was designed to solve the problems of previous schedulers. It used a queue to keep track of runnable tasks and tried to use heuristic of sleep and runtimes to determine if a task was I/O or CPU bound. Though it was much more efficient and scalable, hence the name O(1) scheduler, the large amount of code that was needed to calculate heuristics was difficult to manage and error prone. In response to this problem, the CFS uses a runqueue that is implemented as a time-ordered red-black tree rather than a queue to maintain the tasks. A red-black tree is self-balancing binary search tree where no

path in the tree will ever be more than twice as long as any other. Also, operations on the tree occur in O(logn) time (where n is the size of the tree) that allows the insertion and deletion of a node to be quick and efficient. The CFS uses the red-black tree to essentially build a timeline of future tasks execution. The tasks are ordered by their virtual runtime. This means that the tasks with the most need for the processor (lowest virtual time) are stored towards the left of the three, and the tasks with the least need for the processor (highest virtual runtimes) are stored to the right of the tree. To maintain fairness, the scheduler picks the leftmost node of the tree (the task with the highest priority) to schedule next. The task calculates its time on the CPU by adding its execution time to the virtual runtime and is then inserted back on the tree if runnable. This further emphasizes fairness because the contents of the tree migrate from the right to the left, and all tasks eventually receive execution time on the processor. The CFS makes scheduling decisions mainly when a task wakes up or is created and needs to be added to the runqueue.

In reality, the completely fair scheduler is not entirely correct because the algorithm only guarantees that the "unfair" level is less than O(n) where n is the number of tasks. Developer Mike Galbraith implemented a feature called autogrouping that significantly boosts interactive desktop performance. It was included in the Linux kernel patch for the CFS in November 2010 for the 2.6.38 kernel. He explains the basic algorithm implementation as follows:

> "Each task's signal struct contains an inherited pointer to a refcounted autogroup struct containing a task group pointer, the default for all tasks pointing to the init_task_group. When a task calls __proc_set_tty(), the process wide reference to the default group is dropped, a new task group is created, and the process is moved into the new task group. Children thereafter inherit this task group, and increase its refcount. On exit, a reference to the current task group is dropped when the last reference to each signal struct is dropped. The task group is destroyed when the last signal struct referencing it is freed. At runqueue selection time, IFF a task has no cgroup assignment, its current autogroup is used."

The autogrouping feature can be enabled from boot if it is selected. This mainly solves primary issues for multi-core and multi-cpu (SMP) systems that experience increased interactive response times while performing other tasks that use many CPU-intensive threads.

It is possible for users to change some configurations of the Linux scheduler. This is done by running certain commands in the command line. One example would be the option to select a different default scheduler. Another example regards the CFS group scheduling. If you use this feature, you would also be able to manage the process hierarchies and decide how groups are formed. This allows you to assign fairness across users and/or across processes. As mentioned earlier, another option would be enabling or disabling the autogrouping feature that was developed later to improve desktop performance.

Again, the main idea behind the completely fair scheduler is to maintain balance when providing processor time to tasks. Though it may not necessarily live up to its name, it is still one of the most efficient and scalable schedulers currently available.

Works Cited

"CFS Scheduling." *Kernel*. n.d. Web.

Galbraith, Mike. *Automated per tty task groups*. 15 Nov. 2010. Web.

Jones, Tim M. "Inside the Linux 2.6 Completely Fair Scheduler." *IMB*. 15 Dec. 2009. Web.

Seeker, Volker. *Process Scheduling in Linux*. 12 May 2013. Web.