Analytical Evaluation of Linux CFS Scheduler under Extreme Workload

Sungju Huh[1)], Jonghun Yoo[3)] and Seongsoo Hong[1), 2), 3)]
[1)] *Department of Intelligent Convergence Systems,*
Graduate School of Convergence Science and Technology
[2)] *Advanced Institutes of Convergence Science and Technology*
[3)] *School of Electrical Engineering and Computer Science*
Seoul National University, Republic of Korea
{sjhuh, jhyoo, sshong}@redwood.snu.ac.kr

Abstract

The CFS scheduler is extensively used in numerous Linux installations but its behavior has not been well analyzed. It is even observed that CFS demonstrates unacceptable task starvation when it is used under extreme workload. We analytically and critically examine CFS to identify the source of the starvation problem. Based on our analysis, we suggest priority-based preemption to extend CFS.

Keywords: Linux CFS scheduler, task starvation, fair share scheduling, priority-based scheduling

1. Introduction

The CFS (complete fair scheduling) scheduler [1] has been the primary task scheduler of the mainline Linux kernel since its 2.6.23 release. The goal of the scheduler is to distribute among tasks CPU time proportionally to task weights. As it demonstrated the improved interactivity and fairness of tasks under usual workload, it has been employed in a wide range of computing systems from mobile devices to large cloud servers.

Unfortunately, the CFS scheduler becomes problematic when a system scales up to an extreme degree. In [2], it is reported that tasks scheduled by CFS were starved seriously in a system servicing a trillion requests a day. Despite the extensive use of CFS in numerous Linux installations, there are few analytical and critical evaluations of CFS. Thus, existing research results on CFS cannot successfully explain the ill behavior of CFS under extreme workload [3, 4].

In this paper, we formally analyze the CFS scheduler to show that task response time grows linearly as the number of tasks increases. Our analysis result explains why the response time of a task may grow unacceptably when thousands of tasks are concurrently running in a server, as observed in large data centers [2]. Based on our analysis, we suggest that the CFS be extended to incorporate priority-based preemption.

2. Understanding CFS

CFS attempts to achieve fairness using the notion of task weights. To help programmers specify task weights in a way consistent with older Linux, CFS uses nice values. In conventional Linux, nice values used to denote task priorities. In CFS, a nice value denotes a specific weight value. Nice values range over [-20, 19] and a smaller nice value corresponds to a larger weight.

CFS is a symmetric multiprocessor (SMP) scheduling algorithm. It maintains a run-queue of tasks in each core. It assigns each task in a run-queue virtual run-time which is later used in sorting tasks in the run-queue. The virtual run-time of a task is the task's cumulative execution time inversely scaled by its weight. Specifically, let w_0 be the weight of nice value 0 and w_i be the weight of task τ_i. Let $PR(\tau_i, t)$ be the cumulative physical run-time of task τ_i at time t. In CFS, the virtual run-time of task τ_i at time t is defined as below.

$$VR(\tau_i, t) = \frac{w_0}{w_i} \times PR(\tau_i, t)$$

CFS assigns each task a time slice which is defined as a time interval for which the task is allowed to run without being preempted. In CFS, the length of a task's time slice is proportional to its weight. The time slice of task τ_i is computed by

$$time\ slice_i = \frac{w_i}{\sum_{j \in \varphi} w_j} \times P$$

where φ is the set of runnable tasks, w_i the weight of τ_i and P the constant for given workload. P is defined as

$$P = \begin{cases} sysctl\_sched\_latency & if\ n > nr\_latency \\ min\_granularity \times n & otherwise \end{cases},$$

where n is the number of tasks. *sysctl_sched_latency*, *nr_latency* and *min_granularity* are system-wide constants and their values are 6, 8 and 0.75, respectively, in current Linux implementations.

When a task runs out of its time slice, the NEED_RESCHED flag is set. In every scheduling tick, CFS updates the virtual run-time of the currently running task and checks the NEED_RESCHED flag. If it is set, CFS schedules a task with the smallest virtual run-time in the run-queue.

3. Analysis of CFS

To analyze CFS under extreme workload, we first introduce task starvation time as a metric. It is defined as a time interval for which a task must wait in the run-queue. A scheduling algorithm is said to achieve good interactivity if the task starvation time is bounded below by a small constant. Long starvation time devastates the responsiveness of interactive tasks.

Figure 1 depicts an execution scenario where task τ_i exhibits its maximum starvation time under CFS. At time t_2, τ_i receives a user input which requires δ CPU time. Since only a small amount of time, ε is left in the current time slice, the response is sent back to the user in time t_3, which is $t_1+P+\delta-\varepsilon$.

In CFS, once a task runs out of its time slice, it can be rescheduled only after all other tasks in the run-queue completely consume their time slices. Thus, maximum starvation time T_{max} is computed by

$$T_{max} = \frac{\sum_{j \in \varphi} w_j - w_{min}}{\sum_{j \in \varphi} w_j} \times P$$

where w_{min} is the minimum weight of runnable tasks.

As defined in Section 2, P is a linear function of the number of tasks. Therefore, T_{max} is bounded by O(n). Clearly, CFS demonstrates poor interactivity in huge data center servers where the number of tasks is extremely large.

4. Implication of the Analysis

CFS was designed to overcome several problems of the classical priority-based O(1) scheduler of older Linux.

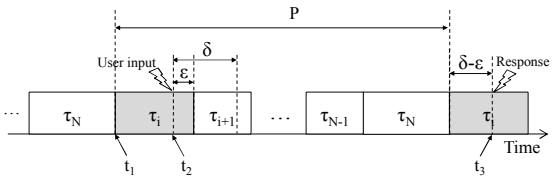

Figure 1. Execution scenario with maximum starvation time.

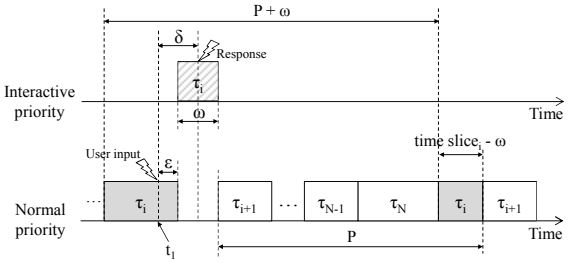

Figure 2. Improved execution scenario.

Unfortunately, our analysis shows that CFS introduces a different type of problem. Two key factors behind this problem are (1) the lack of priorities and (2) non-preemptive execution during a give time slice. Thus, we suggest that priority-based preemption be introduced to CFS. Particularly to deal with an interactive task, we define an interactive priority which is higher than those of other fair scheduling tasks. When a task receives a user input, its priority is temporarily raised to the interactive priority so that it can preempt other tasks and continue to execute even though it has used up its time slice.

Figure 2 illustrates an execution scenario of the suggested approach. At time t_1, task τ_i receives a user input which takes δ time to be serviced. Its priority is boosted to the interactive priority. Task τ_i now preempts other tasks and executes for ω more time where ω is a heuristically obtained value to service the user input. The user can receive the response for the input at desirable time $t_1+\delta$. The maximum starvation time is bounded by constant ω. To guarantee fairness, next time slice of τ_i is compensated to $time\ slice_i - \omega$.

5. Conclusion

We have formally analyzed the behavior of CFS under extreme workload to precisely characterize its task starvation problem. Based on our analysis, we have suggested that the CFS be extended to incorporate priority-based preemption.

References

[1] I. Molnar. The Completely Fair Scheduler, http://people.redhat.com/mingo/cfs-scheduler/.
[2] http://www.mattheaton.com/?p=222.
[3] L. A. Torrey, J. Coleman, and B. Miller, "A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler" Software: Practice and Experience, vol. 37(4), pp. 347~364, 2007.
[4] S. Wang, Y. Chen, W. Jiang, P. Li, T. Dai, and Y. Cui, "Fairness and Interactivity of Three CPU Schedulers in Linux" Proceeding of RTCSA, pp. 172~177, August, 2009.