Completely Fair Scheduler

Aug 01, 2009 By Chandandeep Sin... (/users/chandandeep-singh-pabla)

in

Find out how Linux's new scheduler strives to be fair to all processes and eliminate the problems with the old O(1) scheduler.

(/issue/184)

From Issue #184 August 2009 (/issue/184)

Most modern operating systems are designed to try to extract optimal performance from underlying hardware resources. This is achieved mainly by virtualization of the two main hardware resources: CPU and memory. Modern operating systems provide a multitasking environment that essentially gives each task its own virtual CPU. The task generally is unaware of the fact that it does not have exclusive use of the CPU.

Similarly, memory virtualization is achieved by giving each task its own virtual memory address space, which is then mapped onto the real memory of the system. Again, the task generally is unaware of the fact that its virtual memory addresses may not map to the same physical address in real memory.

CPU virtualization is achieved by “sharing” the CPU between multiple tasks—that is, each running task gets a small fraction of the CPU at regular intervals. The algorithm used to select one task at a time from the multiple available runnable tasks is called the scheduler, and the process of selecting the next task is called scheduling.

The scheduler is one of the most important components of any OS. Implementing a scheduling algorithm is difficult for a couple reasons. First, an acceptable algorithm has to allocate CPU time such that higher-priority tasks (for example, interactive applications like a Web browser) are given preference over low-priority tasks (for example, non-interactive batch processes like program compilation). At the same time, the scheduler must protect against low-priority process starvation. In other words, low-priority processes must be allowed to run eventually, regardless of how many high-priority processes are vying for CPU time. Schedulers also must be crafted carefully, so that processes appear to be running simultaneously without having too large an impact on system throughput.

For interactive processes like GUIs, the ideal scheduler would give each process a very small amount of time on the CPU and rapidly cycle between processes. Because users expect interactive processes to respond to input immediately, the delay between user input and process execution ideally should be imperceptible to humans—somewhere between 50 and 150ms at most.

For non-interactive processes, the situation is reversed. Switching between processes, or context switching, is a relatively expensive operation. Thus, larger slices of time on the processor and fewer context switches can improve system performance and throughput. The scheduling algorithm must strike a balance between all of these competing needs.

Like most modern operating systems, Linux is a multitasking operating system, and therefore, it has a scheduler. The Linux scheduler has evolved over time.

O(1) Scheduler

The Linux scheduler was overhauled completely with the release of kernel 2.6. This new scheduler is called the O(1) scheduler—O(...) is referred to as “big O notation”. The name was chosen because the scheduler's algorithm required constant time to make a scheduling decision, regardless of the number of tasks. The algorithm used by the O(1) scheduler relies on active and expired arrays of processes to achieve constant scheduling time. Each process is given a fixed time quantum, after which it is preempted and moved to the expired array. Once all the tasks from the active array have exhausted their time quantum and have been moved to the expired array, an array switch takes place. This switch makes the active array the new empty expired array, while the expired array becomes the active array.

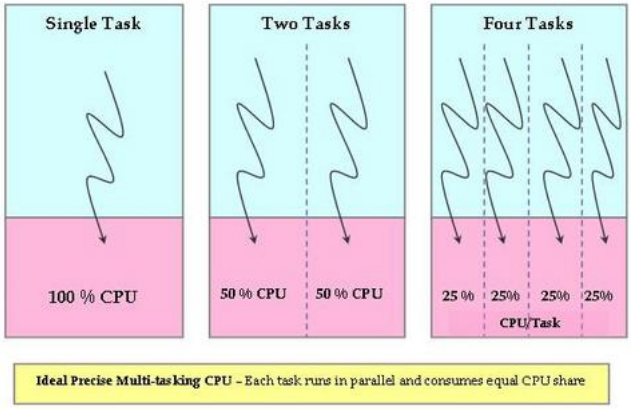
The main issue with this algorithm is the complex heuristics used to mark a task as interactive or non-interactive. The algorithm tries to identify interactive processes by analyzing average sleep time (the amount of time the process spends waiting for input). Processes that sleep for long periods of time probably are waiting for user input, so the scheduler assumes they're interactive. The scheduler gives a priority bonus to interactive tasks (for better throughput) while penalizing non-interactive tasks by lowering their priorities. All the calculations to determine the interactivity of tasks are complex and subject to potential miscalculations, causing non-interactive behavior from an interactive process.

As I explain later in this article, CFS is free from any such calculations and just tries to be “fair” to every task running in the system.

Completely Fair Scheduler

According to Ingo Molnar, the author of the CFS, its core design can be summed up in single sentence: “CFS basically models an 'ideal, precise multitasking CPU' on real hardware.”

Let's try to understand what “ideal, precise, multitasking CPU” means, as the CFS tries to emulate this CPU. An “ideal, precise, multitasking CPU” is a hardware CPU that can run multiple processes at the same time (in parallel), giving each process an equal share of processor power (not time, but power). If a single process is running, it would receive 100% of the processor's power. With two processes, each would have exactly 50% of the physical power (in parallel). Similarly, with four processes running, each would get precisely 25% of physical CPU power in parallel and so on. Therefore, this CPU would be “fair” to all the tasks running in the system (Figure 1).

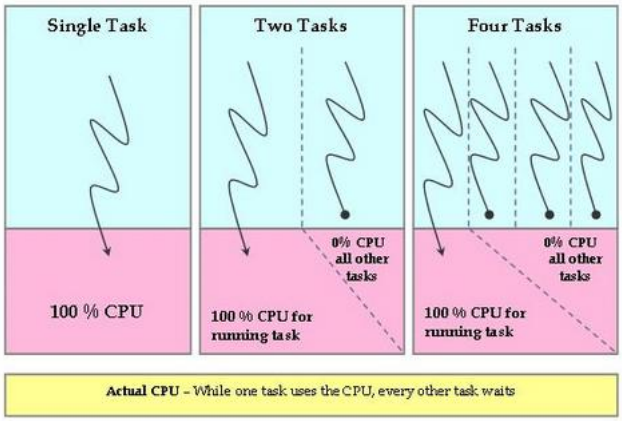


(/files

/linuxjournal.com/linuxjournal/articles/102/10267/10267f1.jpg)

Figure 1. Ideal, Precise, Multitasking CPU

Obviously, this ideal CPU is nonexistent, but the CFS tries to emulate such a processor in software. On an actual real-world processor, only one task can be allocated to a CPU at a particular time. Therefore, all other tasks wait during this period. So, while the currently running task gets 100% of the CPU power, all other tasks get 0% of the CPU power. This is obviously not fair (Figure 2).



(/files /linuxjournal.com/linuxjournal/articles/102/10267/10267f2.jpg)

Figure 2. Actual Hardware CPU

The CFS tries to eliminate this unfairness from the system. The CFS tries to keep track of the fair share of the CPU that would have been available to each process in the system. So, CFS runs a fair clock at a fraction of real CPU clock speed. The fair clock's rate of increase is calculated by dividing the wall time (in nanoseconds) by the total number of processes waiting. The resulting value is the amount of CPU time to which each process is entitled.

As a process waits for the CPU, the scheduler tracks the amount of time it would have used on the ideal processor. This wait time, represented by the per-task wait_runtime variable, is used to rank processes for scheduling and to determine the amount of time the process is allowed to execute before being preempted. The process with the longest wait time (that is, with the gravest need of CPU) is picked by the scheduler and assigned to the CPU. When this process is running, its wait time decreases, while the time of other waiting tasks increases (as they were waiting). This essentially means that after some time, there will be another task with the largest wait time (in gravest need of the CPU), and the currently running task will be preempted. Using this principle, CFS tries to be fair to all tasks and always tries to have a system with zero wait time for each process—each process has an equal share of the CPU (something an “ideal, precise, multitasking CPU” would have done).