CFS (Completely Fair Scheduler) in the Linux kernel
EITF60
Abstract

How an operating system deals with scheduling is essential to how well it utilizes the hardware at its disposal, such as multi-core CPUs. This report attempts to summarize the details of how the scheduling of tasks works in the Linux kernel. The concept of a fair scheduling algorithm is covered through the Completely Fair Scheduler used in Linux.
Introduction

Linux is an operating system found essentially everywhere today – from servers to smartphones and cars. Linux consists of the following components (Linux.com, What is Linux?):

- Bootloader
- Kernel
- Deamons
- Shell
- Graphical Server
- Desktop Environment
- Applications

In this report I will focus on the kernel and more specifically the scheduling algorithm used and how it handles CPU’s with multiple cores. The kernel operates at the core of an operating system. It is the interface between applications running in user mode and hardware. As such it is responsible for the management of CPU resources which includes the scheduling of tasks. The Linux kernel specifically is an open source project containing as of version 2.14.2 more than 20 million lines of code.

![Figure 1 (the role of the kernel) (Bild från Josh. 2015)](image)

Since Linux is used in so many environments it must provide a scheduler that maximizes the bandwidth and CPU utilization as well as one that ensures a high level of interactive performance. The interactivity being especially important in desktop distributions.

Discussion

The responsibility of a scheduler in an operating system is to distribute CPU resources to tasks. This can be accomplished in many ways. Static priorities can be assigned to tasks. Tasks can then be scheduled accordingly, meaning that at any moment a task with priority $P_1$ will have to wait for all the tasks with a priority $P_x > P_1$. An obvious issue arises with this approach – what if there is an unknown number of active tasks with a priority $P_x > P_1$? Starvation will arise which means the starved task will be unable to utilize any resources for an unknown amount of time.
This is where the concept of a fair scheduler comes in. During any scheduling period in an optimal fair scheduler any task regardless of its priority will be able to utilize CPU resources during a time slice that is proportional to its priority.

**Completely Fair Scheduler**

In CFS each core of the CPU has its own run queue. Each task has a so called *nice value* and *weight* assigned to it. The nice value represents how “kind” the specific task is to other tasks. In other words, a task with a high nice value has a lower priority and is thus less likely to take more of the CPUs bandwidth than a task with a low nice value. There is, not surprisingly, a direct correlation then between a task’s nice value and its weight – the weight is in fact mapped from its nice value accordingly: `prio_to_weight[]`.

The portion of the CPU bandwidth assigned to a task is directly proportional to its weight - \[ \text{portion} = \frac{L_t}{L_{cfs,rq}} \] where \( L_t \) is the load weight of the task and \( L_{cfs,rq} \) is the sum of the load weights of all the tasks in the run queue. The time slice that any task then receives is equal to the multiplication of the portion and the period within which the scheduler wants all the tasks to be executed on the CPU – \[ \text{time slice} = \frac{L_t}{L_{cfs,rq}} \times \text{period} \]. The period in this context is not to be confused with the period that is the inverse of the CPUs clock speed – in fact, the period here stretches and reduces in length in proportion to the tasks that needs to be executed within it.

The way that CFS determines the order in which the tasks are to be scheduled for execution is essentially using mainly two concepts – a task’s virtual runtime (\( \text{vruntime} \)) and a red-black tree.

The virtual runtime is a concept introduced to achieve fairness. It takes the nice value as well as the physical execution time of that task into account. The equation is as follows; \[ \text{vruntime} = \frac{\text{execution time}}{L_t} \times NICE_\_LOAD \], where \( NICE_\_LOAD \) is the default nice value. A task with a high load and thus high priority will then more frequently gain access to the CPU than its lower priority counterpart since \( \text{vruntime} \) is inversely proportional to the tasks weight. On the other hand, it can never completely starve a task with lower priority since \( \text{vruntime} \) is also proportional to the tasks time spent executing. (Wang, Chen, Jiang, Li, Dai och Cui 2009; Wong, Tan, Kumari och Lam 2008)

A red-black tree is a self-balancing binary search tree with 4 essential properties; (Morris, 1998)

1. Every node is either red or black
2. Every leaf (NULL) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

With the following properties the tree is ensured to always be balanced and have a height = \( 2\log_2(n + 1) \). This in turn means that all operations on the tree will have a Big O notation of \( O(\log n) \) which makes it efficient for use in a scheduler. The tree is sorted by \( \text{vruntime} \) so the leftmost node in this tree will always contain the task with the lowest \( \text{vruntime} \) and by extension the largest need for CPU bandwidth. After a task has finished executing its assigned time slice it is reinserted with the
updated \textit{vruntime} into the red-black tree. The scheduler always picks the leftmost node in the tree to execute next.

The red-black tree in combination with \textit{vruntime} thus ensures that higher priority tasks will gain access to the CPU resources more frequently without completely starving the lower priority tasks from being able to be executed.

\textbf{Load balancing}

The Linux kernel employs SMP architecture which means that all the cores and/or processors are connected to the same main memory as well as treated equally by the kernel. The kernel deals with multicore processors by assigning each core a run queue. The obvious issue that then remains is to balance these queues so that no queue is overloaded while another is idle. For this purpose, the kernel performs load balancing. Load balance is achieved in the Linux kernel by determining, at every tick, whether the load of each cores’ run queue is fairly distributed. It does this by calculating the average load of each core. If the load is not fairly distributed among the cores the migration will take place. Tasks in the overloaded queue will be moved to a core with a lower load. The goal here is to utilize
each core equally and thus achieving a higher overall utilization of the CPU. (Geunsik, Changwoo och Younglk 2012)

Figure 3 (load-balancing operation in Linux) (Bild från Geunsik, Changwoo och Younglk 2012, 2)
References


