Inside the Linux 2.6 Completely Fair Scheduler

Providing fair access to CPUs since 2.6.23

M. Tim Jones
Independent author

15 December 2009

The task scheduler is a key part of any operating system, and Linux® continues to evolve and innovate in this area. In kernel 2.6.23, the Completely Fair Scheduler (CFS) was introduced. This scheduler, instead of relying on run queues, uses a red-black tree implementation for task management. Explore the ideas behind CFS, its implementation, and advantages over the prior O(1) scheduler.

The Linux scheduler is an interesting study in competing pressures. On one side are the use models in which Linux is applied. Although Linux was originally developed as a desktop operating system experiment, you'll now find it on servers, tiny embedded devices, mainframes, and supercomputers. Not surprisingly, the scheduling loads for these domains differ. On the other side are the technological advances made in the platform, including architectures (multiprocessing, symmetric multithreading, non-uniform memory access [NUMA]) and virtualization. Also embedded here is the balance between interactivity (user responsiveness) and overall fairness. From this perspective, it's easy to see how difficult the scheduling problem can be within Linux.

A short history of Linux schedulers

Early Linux schedulers used minimal designs, obviously not focused on massive architectures with many processors or even hyperthreading. The 1.2 Linux scheduler used a circular queue for runnable task management that operated with a round-robin scheduling policy. This scheduler was efficient for adding and removing processes (with a lock to protect the structure). In short, the scheduler wasn't complex but was simple and fast.

Linux version 2.2 introduced the idea of scheduling classes, permitting scheduling policies for real-time tasks, non-preemptible tasks, and non-real-time tasks. The 2.2 scheduler also included support for symmetric multiprocessing (SMP).

The 2.4 kernel included a relatively simple scheduler that operated in O(N) time (as it iterated over every task during a scheduling event). The 2.4 scheduler divided time into epochs, and within each epoch, every task was allowed to execute up to its time slice. If a task did not use all of its time slice, then half of the remaining time slice was added to the new time slice to allow it to execute

longer in the next epoch. The scheduler would simply iterate over the tasks, applying a goodness function (metric) to determine which task to execute next. Although this approach was relatively simple, it was relatively inefficient, lacked scalability, and was weak for real-time systems. It also lacked features to exploit new hardware architectures such as multi-core processors.

The early 2.6 scheduler, called the *O(1) scheduler,* was designed to solve many of the problems with the 2.4 scheduler—namely, the scheduler was not required to iterate the entire task list to identify the next task to schedule (resulting in its name, *O(1),* which meant that it was much more efficient and much more scalable). The O(1) scheduler kept track of runnable tasks in a run queue (actually, two run queues for each priority level—one for active and one for expired tasks), which meant that to identify the task to execute next, the scheduler simply needed to dequeue the next task off the specific active per-priority run queue. The O(1) scheduler was much more scalable and incorporated interactivity metrics with numerous heuristics to determine whether tasks were I/O-bound or processor-bound. But the O(1) scheduler became unwieldy in the kernel. The large mass of code needed to calculate heuristics was fundamentally difficult to manage and, for the purist, lacked algorithmic substance.

Processes vs. threads

Linux incorporates process and thread scheduling by treating them as one in the same. A process can be viewed as a single thread, but a process can contain multiple threads that share some number of resources (code and/or data).

Given the issues facing the O(1) scheduler and other external pressures, something needed to change. That change came in the way of a kernel patch from Con Kolivas, with his Rotating Staircase Deadline Scheduler (RSDL), which included his earlier work on the staircase scheduler. The result of this work was a simply designed scheduler that incorporated fairness with bounded latency. Kolivas' scheduler impressed many (with calls to incorporate it into the current 2.6.21 mainline kernel), so it was clear that a scheduler change was on the way. Ingo Molnar, the creator of the O(1) scheduler, then developed the CFS based around some of the ideas from Kolivas' work. Let's dig into the CFS to see how it operates at a high level.

An overview of CFS

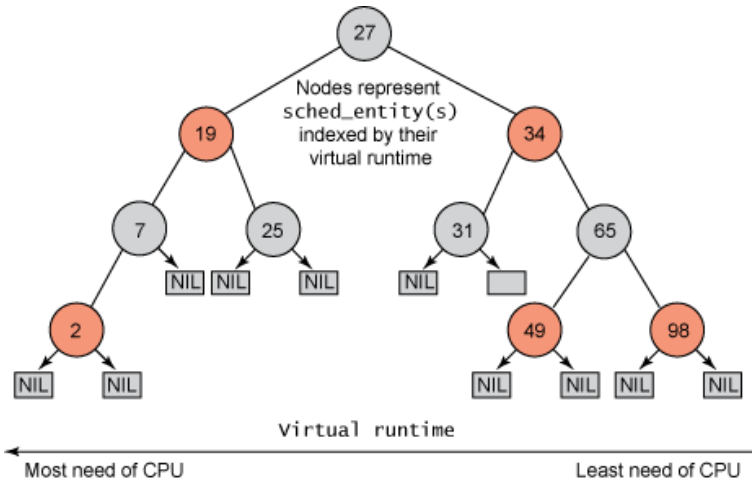
The main idea behind the CFS is to maintain balance (fairness) in providing processor time to tasks. This means processes should be given a fair amount of the processor. When the time for tasks is out of balance (meaning that one or more tasks are not given a fair amount of time relative to others), then those out-of-balance tasks should be given time to execute.

To determine the balance, the CFS maintains the amount of time provided to a given task in what's called the *virtual runtime.* The smaller a task's virtual runtime—meaning the smaller amount of time a task has been permitted access to the processor—the higher its need for the processor. The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable (for example, waiting for I/O) receive a comparable share of the processor when they eventually need it.

But rather than maintain the tasks in a run queue, as has been done in prior Linux schedulers, the CFS maintains a time-ordered red-black tree (see Figure 1). A *red-black tree* is a tree with a

couple of interesting and useful properties. First, it's self-balancing, which means that no path in the tree will ever be more than twice as long as any other. Second, operations on the tree occur in O(log *n*) time (where *n* is the number of nodes in the tree). This means that you can insert or delete a task quickly and efficiently.

Figure 1. Example of a red-black tree

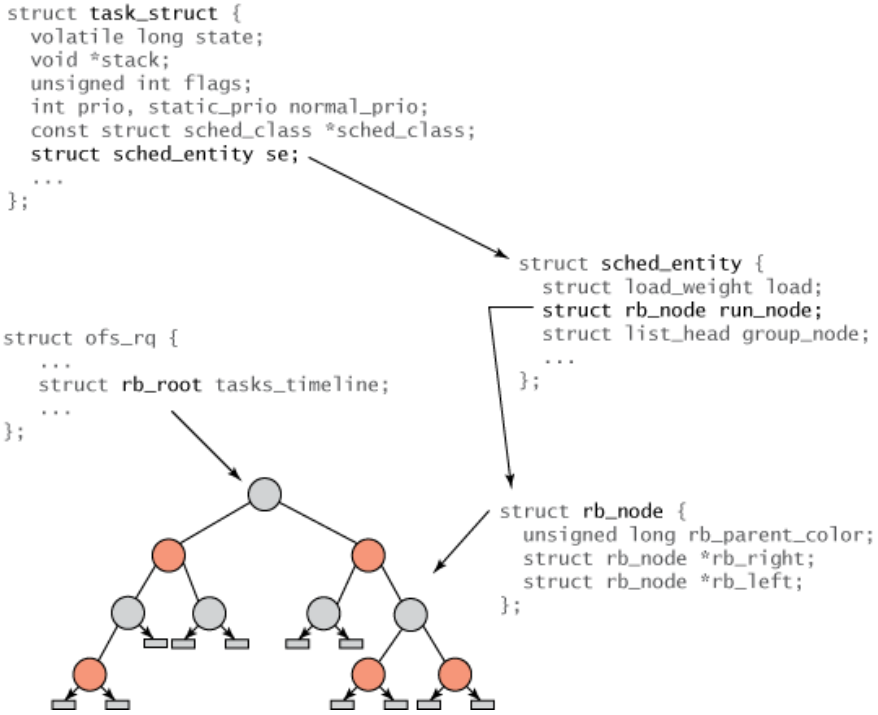


With tasks (represented by `sched_entity` objects) stored in the time-ordered red-black tree, tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the left side of the tree, and tasks with the least need of the processor (highest virtual runtimes) are stored toward the right side of the tree. The scheduler then, to be fair, picks the left-most node of the red-black tree to schedule next to maintain fairness. The task accounts for its time with the CPU by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable. In this way, tasks on the left side of the tree are given time to execute, and the contents of the tree migrate from the right to the left to maintain fairness. Therefore, each runnable task chases the other to maintain a balance of execution across the set of runnable tasks.

CFS internals

All tasks within Linux are represented by a task structure called `task_struct`. This structure (along with others associated with it) fully describes the task and includes the task's current state, its stack, process flags, priority (both static and dynamic), and much more. You can find this and many of the related structures in ./linux/include/linux/sched.h. But because not all tasks are runnable, you won't find any CFS-related fields in `task_struct`. Instead, a new structure called `sched_entity` was created to track scheduling information (see Figure 2).

Figure 2. Structure hierarchy for tasks and the red-black tree



The relationships of the various structures are shown in Figure 2. The root of the tree is referenced via the `rb_root` element from the `cfs_rq` structure (in ./kernel/sched.c). Leaves in a red-black tree contain no information, but internal nodes represent one or more tasks that are runnable. Each node in the red-black tree is represented by an `rb_node`, which contains nothing more than the child references and the color of the parent. The `rb_node` is contained within the `sched_entity` structure, which includes the `rb_node` reference, load weight, and a variety of statistics data. Most importantly, the `sched_entity` contains the `vruntime` (64-bit field), which indicates the amount of time the task has run and serves as the index for the red-black tree. Finally, the `task_struct` sits at the top, which fully describes the task and includes the `sched_entity` structure.

The scheduling function is quite simple when it comes to the CFS portion. In ./kernel/sched.c, you'll find the generic `schedule()` function, which preempts the currently running task (unless it preempts itself with `yield()`). Note that CFS has no real notion of time slices for preemption, because the preemption time is variable. The currently running task (now preempted) is returned to the red-black tree through a call to `put_prev_task` (via the scheduling class). When the `schedule` function comes to identifying the next task to schedule, it calls the `pick_next_task` function. This function is also generic (within ./kernel/sched.c), but it calls the CFS scheduler through the scheduler class. The `pick_next_task` function in CFS can be found in ./kernel/sched_fair.c (called `pick_next_task_fair()`). This function simply picks the left-most task from the red-black tree and returns the associated `sched_entity`. With this reference, a simple call to `task_of()` identifies the `task_struct` reference returned. The generic scheduler finally provides the processor to this task.

Priorities and CFS

CFS doesn't use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute. Lower-priority tasks have higher factors of decay, where higher-priority tasks

have lower factors of delay. This means that the time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task. That's an elegant solution to avoid maintaining run queues per priority.

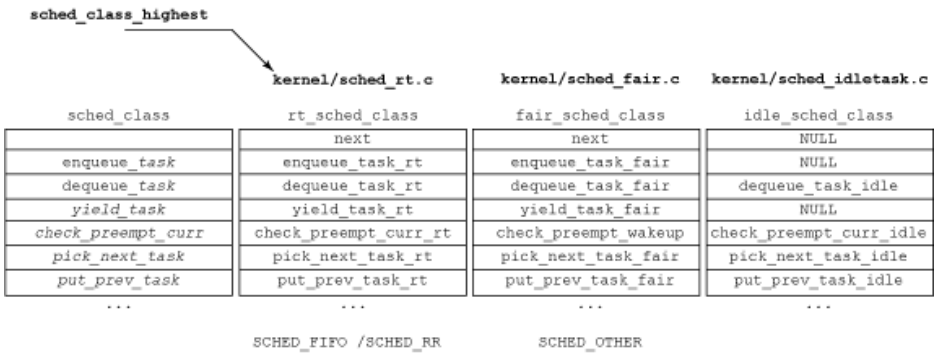
CFS group scheduling

Another interesting aspect of CFS is the concept of *group scheduling* (introduced with the 2.6.24 kernel). Group scheduling is another way to bring fairness to scheduling, particularly in the face of tasks that spawn many other tasks. Consider a server that spawns many tasks to parallelize incoming connections (a typical architecture for HTTP servers). Instead of all tasks being treated fairly uniformly, CFS introduces groups to account for this behavior. The server process that spawns tasks share their virtual runtimes across the group (in a hierarchy), while the single task maintains its own independent virtual runtime. In this way, the single task receives roughly the same scheduling time as the group. You'll find a /proc interface to manage the process hierarchies, giving you full control over how groups are formed. Using this configuration, you can assign fairness across users, across processes, or a variation of each.

Scheduling classes and domains

Also introduced with CFS is the idea of scheduling classes (recall from Figure 2). Each task belongs to a scheduling class, which determines how a task will be scheduled. A scheduling class defines a common set of functions (via `sched_class`) that define the behavior of the scheduler. For example, each scheduler provides a way to add a task to be scheduled, pull the next task to be run, yield to the scheduler, and so on. Each scheduler class is linked with one another in a singly linked list, allowing the classes to be iterated (for example, for the purposes of enablement of disablement on a given processor). The general structure is shown in Figure 3. Note here that enqueue and dequeue task functions simply add or remove a task from the particular scheduling structures. The function `pick_next_task` chooses the next task to execute (depending upon the particular policy of the scheduling class).

Figure 3. Graphical view of scheduling classes



But recall that the scheduling classes are part of the task structure itself (see Figure 2). This simplifies operations on tasks, regardless of their scheduling class. For example, the following function preempts the currently running task with a new task (where `curr` defines the currently running task, `rq` represents the red-black tree for CFS, and `p` is the next task to schedule) from ./kernel/sched.c:

```
static inline void check_preempt( struct rq *rq, struct task_struct *p )
{
  rq->curr->sched_class->check_preempt_curr( rq, p );
}
```

If this task were using the fair scheduling class, `check_preempt_curr()` would resolve to `check_preempt_wakeup()`. You can see these relationships in ./kernel/sched_rt.c, ./kernel/sched_fair.c and ./kernel/sched_idle.c.

Scheduling classes are yet another interesting aspect of the scheduling changes, but the functionality grows with the addition of scheduling domains. These domains allow you to group one or more processors hierarchically for purposes load balancing and segregation. One or more processors can share scheduling policies (and load balance between them) or implement independent scheduling policies to intentionally segregate tasks.

Other schedulers

Work on scheduling continues, and you'll find schedulers under development that push the boundaries of performance and scaling. Con Kolivas was not deterred by his Linux experience and has developed another scheduler for Linux with a provocative acronym: BFS. The scheduler was reported to have better performance on NUMA systems as well as mobile devices and was introduced into a derivative of the Android operating system.

Going further

If there's one constant with Linux, it's that change is inevitable. Today, the CFS is the 2.6 Linux scheduler; but tomorrow, it could be another new scheduler or a suite of schedulers that can be statically or dynamically invoked. There's also a certain amount of mystery in the process behind the CFS, RSDL, and kernel inclusion, but thanks to both Kolivas' and Molnar's work, we have a new level of fairness in 2.6 task scheduling.

Resources

Learn

- A red-black tree (or symmetric binary B-tree) is a self-balancing binary tree invented by Rudolf Bayer. It's a very useful tree representation that has good worst-case time for operations such as insert, search, and delete. You can find red-black trees used in a variety of applications, including the construction of associative arrays.
- Task scheduling is an important aspect to operating system design, from desktop operating system schedulers to real-time schedulers and embedded operating system schedulers. These notes from Martin C. Rinard's operating systems lecture provide a great condensed summary of processor scheduling.
- The big O notation is useful in describing the limiting behavior of a function. This Wikipedia entry includes great information as well as a useful list of function classes.
- Con Kolivas has been working on new experimental Linux schedulers for some time. You can learn more about his schedulers, including the Staircase Process Scheduler and the Rotating Staircase Deadline Scheduler, which ultimately proved that fair share scheduling was possible.
- What would life be without drama? For a behind-the-scenes look at the development of the CFS, check out this collection of Linux kernel mailing list posts, which includes Con Kolivas (who introduced the RSDL patch, implementing the core ideas of the CFS), and Ingo Molnar (the scheduler code gatekeeper who later introduced his own version of the scheduler after initially rejecting the ideas behind it). As always, the truth lies somewhere in the middle, but this interesting article from Kerneltrap uncovers another aspect of open source development.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- An alternative, BFS scheduler from Con Kolivas is available as a patch for Linux desktop systems as well as small mobile devices. Kolivas chose this acronym (whose name shall not be spoken), because he wanted to raise attention to the fact that having a scheduler support a massive number of tasks is great but shouldn't penalize a desktop scheduler built on more modest hardware.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the My developerWorks community; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of *Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.