Multiprocessing with the Completely Fair Scheduler

Introducing the CFS for Linux

Avinesh Kumar (avinesh.kumar@in.ibm.com), System Software Engineer, IBM

Summary: The Linux® 2.6.23 kernel comes with a modular scheduler core and a Completely Fair Scheduler (CFS), which is implemented as a scheduling module. In this article, get acquainted with the major features of the CFS, see how it works, and look ahead to some of the expected changes for the 2.6.24 release.

Date: 08 Jan 2008
Level: Intermediate
PDF: A4 and Letter (47KB | 13 pages)Get Adobe® Reader®
Activity: 1009 views
Comments: 0 (Add comments)

Average rating (based on 50 votes)

- **Show articles and other content related to my search: linux scheduler cfs**

The scheduler in the 2.6.23 kernel paves the way for other scheduling modules to work concurrently with the core. ("Modular" in this case doesn't mean the scheduler is broken into loadable modules, rather the code itself has become modular.) For more on how a scheduler works, check out the developerWorks article "Inside the Linux scheduler" (see Resources later in this article for a link).

Major features and policies

Major features introduced in the latest scheduler include:

- The modular scheduler
- The Completely Fair Scheduler (CFS)
- CFS group scheduling

Modular scheduler

The introduction of scheduling classes makes the core scheduler quite extensible. These classes (the *scheduler modules*) encapsulate the scheduling policies. This scheduler modularizes the scheduling policies but does not modularize the scheduler itself like the Pluggable CPU scheduler framework (in that case, a default scheduler can be chosen at kernel build time, and other CPU schedulers can be used by passing an argument to the kernel at boot time).

CFS

The Completely Fair Scheduler tries to run the task with the "gravest need" for CPU time; this helps to assure that every process gets its fair share of CPU. CFS does not consider a task to be a sleeper if it sleeps for "very" short time—a short sleeper might be entitled to some bonus time, but never more than it would have had had it not slept.

CFS group scheduling

Consider an example with two users, A and B, who are running jobs on a machine. User A has just two jobs running, while user B has 48 jobs running. Group scheduling enables CFS to be fair to users A and B, rather than being fair to all 50 jobs running in the system. Both users get a 50-50 share. B would use his 50% share to run his 48 jobs and would not be able to encroach on A's 50% share.

The CFS scheduling module (implemented in kernel/sched_fair.c) is used for the following scheduling policies: `SCHED_NORMAL`, `SCHED_BATCH`, and `SCHED_IDLE`. For `SCHED_RR` and `SCHED_FIFO` policies, the real-time scheduling module is used (that module is implemented in kernel/sched_rt.c).

Some of these changes were made for these reasons:

- To enable better scheduling for servers as well as for desktops.
- To provide new features that were requested.
- To improve the heuristics. Those used in the vanilla scheduler made some attacks easy to implement. Also, if the heuristics gauged a scenario incorrectly, unwanted behaviors could result.

CFS compared to RSDL

Rotating Staircase Deadline Scheduler

The RSDL is a process scheduler that eliminated the interactivity estimation code, the algorithm from the previous Linux process scheduler that used statistics to try to predict the future with heuristics (and failed at it). RSDL is based on the concept of "fairness." Processes are treated equally and are given the same time slices. The scheduler doesn't care or even try to guess if the process is CPU- or IO-bound. The RSDL improved the user's perceived "interactivity" and was on its way to being merged into the 2.6 kernel when Ingo Molnar, the creator of the original O(1) process scheduler, created CFS, taking the basic design element of fair scheduling from RSDL. It was well received by many hackers, although RSDL (created by Con Kolivas) was also appreciated.

The credit for proving that "fair scheduling" can be achieved without conflicting with the latency goals of interactivity goes to Con Kolivas. He has proved with RSDL/SD schedulers that fairness can be achieved without all the complex heuristics used to estimate interactivity of a process.

CFS does not use a priority array, and it does away with the array-switching artifact of the vanilla scheduler. A few important differences between RSDL and CFS:

- RSDL is based on "strict fairness"; however, the CFS takes into account the longevity of sleep time in the interactive process, so short sleepers do get a bit more CPU time.
- RSDL uses priority queues like the vanilla scheduler, while CFS does not.
- RSDL and the vanilla scheduler are affected by the array-switching artifact, while CFS is not.

CFS doesn't track sleeping time and doesn't use heuristics to identify interactive tasks—it just makes sure every process gets a fair share of CPU within a set amount of time given the number of runnable processes on the CPU.

Important CFS data structures

CFS uses a time-ordered red-black tree for each CPU.

Red-black tree, defined by Wikipedia

According to Wikipedia, a *red-black tree* is a type of self-balancing binary search tree, a data structure used to implement associative arrays. For every running process, there is a node in the red-black tree. The process at the left-most position of the red-black tree is the one to be scheduled next. The red-black tree is complex, but it has a good worst-case running time for its operations and is efficient in practice: it can search, insert, and delete in *O(log n)* time, where *n* is the number of elements in the tree. The leaf nodes are not relevant and do not contain data. To save memory, sometimes a single sentinel node performs the role of all leaf nodes. All references from internal nodes to leaf nodes instead point to the sentinel node.

The tree approach works well for these reasons:

- The red-black tree is always balanced.
- Because the red-black tree is a binary tree, the time complexities of lookup operations are logarithmic. However, non-left-most lookup is hardly ever done and the left-most node pointer is always cached.
- The red-black tree is *O(log n)* in time for most operations, while the previous scheduler employed *O(1)*, using a priority array with a fixed number of priorities. *O(log n)* behavior is measurably slower, but only marginally for very large task counts. It was one of the very first things Molnar tested once he tried the tree approach.
- A red-black tree can be implemented with internal storage—that is, no external allocations are needed to maintain the data structure.

Let's look at some of the key data structures implementing the new scheduler.

Changes in struct task_struct

CFS eliminates `struct prio_array` and introduces scheduling entity and scheduling classes as defined by `struct sched_entity` and `struct sched_class`, respectively. Accordingly, `task_struct` contains information about two other structures, `sched_entity` and `sched_class`:

Listing 1. The task_struct structure

```
struct task_struct { /* Defined in 2.6.23:/usr/include/linux/sched.h */
....
-   struct prio_array *array;
+  struct sched_entity se;
+  struct sched_class *sched_class;
   ....
   ....
};
```

struct sched_entity

This structure holds sufficient information to actually accomplish the scheduling job of a task or a task-group. This is used to implement group scheduling. A scheduling entity might not be associated with a process.

Listing 2. The sched_entity structure

```
struct sched_entity { /* Defined in 2.6.23:/usr/include/linux/sched.h */
 long wait_runtime;   /* Amount of time the entity must run to become completely */
                      /* fair and balanced.*/
 s64 fair_key;
 struct load_weight   load;          /* for load-balancing */
 struct rb_node run_node;             /* To be part of Red-black tree data structure */
 unsigned int on_rq;
 ....
};
```

struct sched_class

The scheduling classes are like a chain of modules assisting the core scheduler. Each scheduler module needs to implement a set of functions as suggested by `struct sched_class`.

Listing 3. The sched_class structure

```
struct sched_class { /* Defined in 2.6.23:/usr/include/linux/sched.h */
      struct sched_class *next;
      void (*enqueue_task) (struct rq *rq, struct task_struct *p, int wakeup);
      void (*dequeue_task) (struct rq *rq, struct task_struct *p, int sleep);
      void (*yield_task) (struct rq *rq, struct task_struct *p);

      void (*check_preempt_curr) (struct rq *rq, struct task_struct *p);

      struct task_struct * (*pick_next_task) (struct rq *rq);
      void (*put_prev_task) (struct rq *rq, struct task_struct *p);

      unsigned long (*load_balance) (struct rq *this_rq, int this_cpu,
                 struct rq *busiest,
                 unsigned long max_nr_move, unsigned long max_load_move,
                 struct sched_domain *sd, enum cpu_idle_type idle,
                 int *all_pinned, int *this_best_prio);

      void (*set_curr_task) (struct rq *rq);
      void (*task_tick) (struct rq *rq, struct task_struct *p);
      void (*task_new) (struct rq *rq, struct task_struct *p);
};
```

Let's look at some of the functions in Listing 3:

- `enqueue_task`: When a task enters a runnable state, this function is called. It puts the scheduling entity (process) into the red-black tree and increments the `nr_running` variable.
- `dequeue_task`: When a task is no longer runnable, this function is called to keep the corresponding scheduling entity out of the red-black tree. It decrements the `nr_running` variable.
- `yield_task`: This function is basically just a dequeue followed by an enqueue, unless the `compat_yield sysctl` is turned on; in that case, it places the scheduling entity at the right-most end of the red-black tree.
- `check_preempt_curr`: This function checks whether the currently running task can be preempted. The CFS scheduler module does fairness testing before actually preempting the running task. This drives the wakeup preemption.
- `pick_next_task`: This function chooses the most appropriate process eligible to run next.
- `load_balance`: Each scheduler module implements a pair of functions, `load_balance_start()` and `load_balance_next()` to implement an iterator that gets called in the `load_balance` routine of the module. The core scheduler uses this method to load-balance processes managed by the scheduling module.
- `set_curr_task`: This function is called when a task changes its scheduling class or changes its task group.
- `task_tick`: This function is mostly called from time tick functions; it might lead to process switch. This drives the running preemption.
- `task_new`: The core scheduler gives the scheduling module an opportunity to manage new task startup. The CFS scheduling module uses it for group scheduling, while the scheduling module for a real-time task does not use it.

CFS-related fields in a runqueue

For each runqueue, there is a structure holding information about the associated red-black tree.

Listing 4. The cfs_rq structure

```
struct cfs_rq {/* Defined in 2.6.23:kernel/sched.c */
        struct load_weight load;
        unsigned long nr_running;

        s64 fair_clock; /* runqueue wide global clock */
        u64 exec_clock;
        s64 wait_runtime;
        u64 sleeper_bonus;
```

```
      unsigned long wait_runtime_overruns, wait_runtime_underruns;

      struct rb_root tasks_timeline; /* Points to the root of the rb-tree*/
      struct rb_node *rb_leftmost; /* Points to most eligible task to give the CPU */
      struct rb_node *rb_load_balance_curr;
#ifdef CONFIG_FAIR_GROUP_SCHED
      struct sched_entity *curr; /* Currently running entity */
      struct rq *rq;      /* cpu runqueue to which this cfs_rq is attached */
      ...
      ...
#endif
};
```

How CFS works

The CFS scheduler uses an appeasement policy that guarantees fairness. As a task gets into the runqueue, the current time is recorded, and while the process waits for the CPU, its `wait_runtime` value gets incremented by an amount depending on the number of processes currently in the runqueue. The priority values of different tasks are also considered while doing these calculations. When this task gets scheduled to the CPU, its `wait_runtime` value starts decrementing and as this value falls to such a level that other tasks become the new left-most task of the red-black tree and the current one gets preempted. This way CFS tries for the *ideal* situation where `wait_runtime` is zero!

CFS maintains the runtime of a task relative to a runqueue-wide clock called `fair_clock` (`cfs_rq->fair_clock`), which runs at a fraction of real time so that it runs at the ideal pace for a single task.

How are granularity and latency related?

The simple equation correlating granularity and latency is

gran = (lat/nr) - (lat/nr/nr)
where gran = granularity,
lat = latency, and
nr = count of running tasks.

For example, if you have four runnable tasks, then `fair_clock` will increase at one quarter of the speed of wall time. Each task then tries to catch up with this ideal time. This is a result of the quantized nature of time-shared multitasking. That is, only a single task can run at any one time; therefore, the other processes build up a debt (`wait_runtime`). So once a task gets scheduled, it will catch up its debt (and a little more because the `fair_clock` will not stop ticking during the catch up period).

Priorities are introduced by weighting tasks. Suppose we have two tasks: one is to consume twice as much CPU time as the other, yielding a 2:1 ratio. The math gets changed so that a task with weight 0.5 sees time pass by twice as fast.

We enqueue the task in the tree based on `fair_clock`.

As for *time slices*, remember, CFS does not use them, at least not in the prior way. Time slices in CFS are of variable length and are decided dynamically.

For the *load balancer*, scheduling modules implement iterators that are used to walk through all the tasks managed by that scheduling module to do load balancing.

Runtime tunables

The important `sysctls` introduced to tune the scheduler at runtime are (names ending in *ns* are in units of nanoseconds):

- `sched_latency_ns`: Targeted preemption latency for CPU-bound tasks.
- `sched_batch_wakeup_granularity_ns`: Wake-up granularity for `SCHED_BATCH`.
- `sched_wakeup_granularity_ns`: Wake-up granularity for `SCHED_OTHER`.
- `sched_compat_yield`: Applications depending heavily on `sched_yield()`'s behavior can expect varied performance because of the way CFS changes this, so turning on the `sysctls` is recommended.
- `sched_child_runs_first`: The child is scheduled next after `fork`; it's the default. If set to 0, then the parent is given the baton.
- `sched_min_granularity_ns`: Minimum preemption granularity for CPU-bound tasks.
- `sched_features`: Contains information about various debugging-related features.
- `sched_stat_granularity_ns`: Granularity for collecting scheduler statistics.

Here are some typical values of the runtime parameters on a system:

Listing 5. Typical runtime parameter values

```
[root@dodge ~]# sysctl -A|grep "sched" | grep -v "domain"
kernel.sched_min_granularity_ns = 4000000
kernel.sched_latency_ns = 40000000
kernel.sched_wakeup_granularity_ns = 2000000
kernel.sched_batch_wakeup_granularity_ns = 25000000
kernel.sched_stat_granularity_ns = 0
kernel.sched_runtime_limit_ns = 40000000
kernel.sched_child_runs_first = 1
kernel.sched_features = 29
kernel.sched_compat_yield = 0
[root@dodge ~]#
```

The new scheduler debugging interface

The new scheduler comes with a pretty good debugging interface, and it also provides runtime statistics information, implemented in kernel/sched_debug.c and kernel/sched_stats.h, respectively. To provide runtime statistics for the scheduler and debugging information, a few files have been added to the proc pseudo filesystem:

- /proc/sched_debug: Displays the current values of runtime scheduler tunables, the CFS statistics, and runqueue information on all available CPUs. Function `sched_debug_show()` gets called and defined in sched_debug.c when this proc file is read.
- /proc/schedstat: Displays runqueue-specific statistics and also domain-specific statistics for SMP systems for all connected CPUs. Function `show_schedstat()` defined in kernel/sched_stats.h handles read operations on this proc entry.
- /proc/[PID]/sched: Displays information on the related scheduling entity. Function `proc_sched_show_task()` defined in kernel/sched_debug.c gets called when this file is read.

Changes for kernel 2.6.24

What are some of the expected changes for 2.6.24 release? Well, instead of chasing a global clock (`fair_clock`), the tasks chase each other. A clock per task (scheduling entity), `vruntime`, will be introduced (`wall_time/task_weight`) and an approximated average initializes the clock of a new task.

Other important changes affect key data structures. Here are the scheduled changes in `struct sched_entity`:

Listing 6. Scheduled changes in the sched_entity structure for 2.6.24

```
struct sched_entity { /* Defined in /usr/include/linux/sched.h */
- long    wait_runtime;
- s64     fair_key;
+ u64     vruntime;
- u64     wait_start_fair;
- u64     sleep_start_fair;
      ...
      ...
}
```

These are changes in `struct cfs_rq`:

Listing 7. Scheduled changes in the cfs_rq structure for 2.6.24

```
 struct cfs_rq { /* Defined in kernel/sched.c */
-         s64 fair_clock;
-         s64 wait_runtime;
-         u64 sleeper_bonus;
-         unsigned long wait_runtime_overruns, wait_runtime_underruns;
+        u64 min_vruntime;

+        struct sched_entity *curr;

+#ifdef CONFIG_FAIR_GROUP_SCHED
       ...
+        struct task_group *tg;    /* group that "owns" this runqueue */
```

```
        ...
#endif
 };
```

A new structure has been introduced for grouping tasks:

Listing 8. Newly added task_group structure

```
struct task_group { /* Defined in kernel/sched.c */
    #ifdef CONFIG_FAIR_CGROUP_SCHED
        struct cgroup_subsys_state css;
   #endif
        /* schedulable entities of this group on each cpu */
        struct sched_entity **se;
        /* runqueue "owned" by this group on each cpu */
        struct cfs_rq **cfs_rq;
        unsigned long shares;
        /* spinlock to serialize modification to shares */
        spinlock_t lock;
        struct rcu_head rcu;
};
```

Each task tracks its runtime, and tasks are enqueued on that value. This means that the task that ran least will be the left-most one. Again, priorities are realized by weighting time. Each task strives to get scheduled exactly once during:

```
sched_period = (nr_running > sched_nr_latency) ? sysctl_sched_latency :
((nr_running * sysctl_sched_latency) / sched_nr_latency)
```

where `sched_nr_latency` = `(sysctl_sched_latency / sysctl_sched_min_granularity)`. That is, when there are more than `latency_nr` runnable tasks, the scheduling period is linearly extended. `sched_slice()`, defined in sched_fair.c, is the place for these calculations.

So, if each runnable task runs its `sched_slice()` worth of time, it has spent `sched_period` time, and each task will have run an equal amount of time proportional to its weight. Furthermore, at any point, CFS promises to run `sched_period` ahead because the task last scheduled would run again within that window.

So when a new task becomes runnable, there are strict requirements for its placement. This task cannot run before all the other tasks have run; otherwise, the promise made to them gets broken. However, because this task does get enqueued, the extra weight on the runqueue will shorten the slices of all other tasks, freeing up a slot at the end of the `sched_priod` exactly the size this new task needs. This new task is placed there.

Group scheduling enhancements in 2.6.24

In 2.6.24, you will be able to tune the scheduler to be fair to users or groups rather than just for tasks. The tasks can be grouped together to form entities, and the scheduler would be fair to these entities and then to tasks can be grouped together to form entities, and the scheduler would be fair to these entities and then to the tasks in those entities. To enable this feature, `CONFIG_FAIR_GROUP_SCHED` should be selected during the kernel build. As of now, only `SCHED_NORMAL` and `SCHED_BATCH` tasks can be grouped.

There are two mutually exclusive ways to group tasks, based on:

- User IDs.
- cgroup pseudo filesystem: This option lets an administrator create groups as needed. For details, read the file cgroups.txt in the kernel source documentation directory.

The kernel configuration parameters `CONFIG_FAIR_USER_SCHED` and `CONFIG_FAIR_CGROUP_SCHED` can help you choose your options.

Summary

The new scheduler extends scheduling capabilities by introducing scheduling classes and also simplifies debugging by improving schedule statistics. CFS is getting good reviews when tested for thread-intensive applications including 3D games.

Acknowledgments

I am thankful to Peter Zijlstra for contributing significantly to the CFS scheduler development, and for taking time out from his busy schedule to give me his comments and suggestions to improve this article. Thanks to Srivatsa Vaddagiri for his nice work on CFS group scheduling, and to RSDL creator Con Kolivas. I also gratefully acknowledge Ingo Molnar, the maintainer of the Linux Scheduler, for his interest in this article.

Resources

Learn

- "Inside the Linux scheduler" (developerWorks, June 2006) explores the attributes of the Linux 2.6 scheduler.

- Read "Towards Linux 2.6: A look into the workings of the next new kernel" (developerWorks, September 2003) for a technical and historical perspective of Linux 2.6 and the scheduler.

- "Linux and symmetric multiprocessing" (developerWorks, March 2007) introduces the topic of the 2.6 kernel and symmetric multiprocessing.

- In "Take charge of processor affinity" (developerWorks, September 2005), knowing a little bit about how the Linux 2.6 scheduler treats CPU affinity, whether soft or hard, can help you design better userspace applications.

- Ingo Molnar's git tree shows a timeline of the changes planned for 2.6.24.
- 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

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and community topics in our new developerWorks spaces.

About the author

Avinesh Kumar works as a System Software Engineer for the Andrew File System Team at the IBM Software Labs in Pune, India. He works with kernel- and user-level debugging of dumps and crashes, as well as reported bugs on the Linux, AIX, and Solaris platforms. Avinesh has an MCA from the Department of Computer Science at the University of Pune. He is a Linux enthusiast who spends his spare time exploring the Linux kernel on his Fedora Core 6 box.

Trademarks