CFS Scheduler

1. OVERVIEW

CFS stands for "Completely Fair Scheduler," and is the new "desktop" process scheduler implemented by Ingo Molnar and merged in Linux 2.6.23. It is the replacement for the previous vanilla scheduler's SCHED_OTHER interactivity code.

80% of CFS's design can be summed up in a single sentence: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.

"Ideal multi-tasking CPU" is a (non-existent :-)) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at 1/nr_running speed. For example: if there are 2 tasks running, then it runs each at 50% physical power --- i.e., actually in parallel.

On real hardware, we can run only a single task at once, so we have to introduce the concept of "virtual runtime." The virtual runtime of a task specifies when its next timeslice would start execution on the ideal multi-tasking CPU described above. In practice, the virtual runtime of a task is its actual runtime normalized to the total number of running tasks.

2. FEW IMPLEMENTATION DETAILS

In CFS the virtual runtime is expressed and tracked via the per-task p->se.vruntime (nanosec-unit) value. This way, it's possible to accurately timestamp and measure the "expected CPU time" a task should have gotten.

[small detail: on "ideal" hardware, at any time all tasks would have the same p->se.vruntime value --- i.e., tasks would execute simultaneously and no task would ever get "out of balance" from the "ideal" share of CPU time.]

CFS's task picking logic is based on this p->se.vruntime value and it is thus very simple: it always tries to run the task with the smallest p->se.vruntime value (i.e., the task which executed least so far). CFS always tries to split up CPU time between runnable tasks as close to "ideal multitasking hardware" as possible.

Most of the rest of CFS's design just falls out of this really simple concept, with a few add-on embellishments like nice levels, multiprocessing and various algorithm variants to recognize sleepers.

3. THE RBTREE

CFS's design is quite radical: it does not use the old data structures for the runqueues, but it uses a time-ordered rbtree to build a "timeline" of future task execution, and thus has no "array switch" artifacts (by which both the previous vanilla scheduler and RSDL/SD are affected).

CFS also maintains the rq->cfs.min_vruntime value, which is a monotonic increasing value tracking the smallest vruntime among all tasks in the runqueue. The total amount of work done by the system is tracked using min_vruntime; that value is used to place newly activated entities on the left side of the tree as much as possible.

The total number of running tasks in the runqueue is accounted through the rq->cfs.load value, which is the sum of the weights of the tasks queued on the runqueue.

CFS maintains a time-ordered rbtree, where all runnable tasks are sorted by the p->se.vruntime key (there is a subtraction using rq->cfs.min_vruntime to account for possible wraparounds). CFS picks the "leftmost" task from this tree and sticks to it.
As the system progresses forwards, the executed tasks are put into the tree more and more to the right --- slowly but surely giving a chance for every task to become the "leftmost task" and thus get on the CPU within a deterministic amount of time.

Summing up, CFS works like this: it runs a task a bit, and when the task schedules (or a scheduler tick happens) the task's CPU usage is "accounted for": the (small) time it just spent using the physical CPU is added to p->se.vruntime. Once p->se.vruntime gets high enough so that another task becomes the "leftmost task" of the time-ordered rbtree it maintains (plus a small amount of "granularity" distance relative to the leftmost task so that we do not over-schedule tasks and trash the cache), then the new leftmost task is picked and the current task is preempted.

4. SOME FEATURES OF CFS

CFS uses nanosecond granularity accounting and does not rely on any jiffies or other HZ detail. Thus the CFS scheduler has no notion of "timeslices" in the way the previous scheduler had, and has no heuristics whatsoever. There is only one central tunable (you have to switch on CONFIG_SCHED_DEBUG):

```
/proc/sys/kernel/sched_min_granularity_ns
```

which can be used to tune the scheduler from "desktop" (i.e., low latencies) to "server" (i.e., good batching) workloads. It defaults to a setting suitable for desktop workloads. SCHED_BATCH is handled by the CFS scheduler module too.

Due to its design, the CFS scheduler is not prone to any of the "attacks" that exist today against the heuristics of the stock scheduler: fiftyp.c, thud.c, chew.c, ring-test.c, massive_intr.c all work fine and do not impact interactivity and produce the expected behavior.

The CFS scheduler has a much stronger handling of nice levels and SCHED_BATCH than the previous vanilla scheduler: both types of workloads are isolated much more aggressively.

SMP load-balancing has been reworked/sanitized: the runqueue-walking assumptions are gone from the load-balancing code now, and iterators of the scheduling modules are used. The balancing code got quite a bit simpler as a

result.

5. Scheduling policies

CFS implements three scheduling policies:

- SCHED_NORMAL (traditionally called SCHED_OTHER): The scheduling policy that is used for regular tasks.

- SCHED_BATCH: Does not preempt nearly as often as regular tasks would, thereby allowing tasks to run longer and make better use of caches but at the cost of interactivity. This is well suited for batch jobs.

- SCHED_IDLE: This is even weaker than nice 19, but its not a true idle timer scheduler in order to avoid to get into priority inversion problems which would deadlock the machine.

SCHED_FIFO/_RR are implemented in sched_rt.c and are as specified by POSIX.

The command chrt from util-linux-ng 2.13.1.1 can set all of these except SCHED_IDLE.

6. SCHEDULING CLASSES

The new CFS scheduler has been designed in such a way to introduce "Scheduling Classes," an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming too much about them.

sched_fair.c implements the CFS scheduler described above.

sched_rt.c implements SCHED_FIFO and SCHED_RR semantics, in a simpler way than the previous vanilla scheduler did. It uses 100 runqueues (for all 100 RT priority levels, instead of 140 in the previous scheduler) and it needs no expired array.

Scheduling classes are implemented through the sched_class structure, which contains hooks to functions that must be called whenever an interesting event occurs.

This is the (partial) list of the hooks:

- enqueue_task(...)

  Called when a task enters a runnable state.
  It puts the scheduling entity (task) into the red-black tree and increments the nr_running variable.

- dequeue_tree(...)

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 if a task that entered the runnable state should preempt the currently running task.

- pick_next_task(...)

  This function chooses the most appropriate task eligible to run next.

- 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.

7. GROUP SCHEDULER EXTENSIONS TO CFS

Normally, the scheduler operates on individual tasks and strives to provide fair CPU time to each task. Sometimes, it may be desirable to group tasks and provide fair CPU time to each such task group. For example, it may be desirable to first provide fair CPU time to each user on the system and then to each task belonging to a user.

CONFIG_CGROUP_SCHED strives to achieve exactly that. It lets tasks to be grouped and divides CPU time fairly among such groups.

CONFIG_RT_GROUP_SCHED permits to group real-time (i.e., SCHED_FIFO and SCHED_RR) tasks.

CONFIG_FAIR_GROUP_SCHED permits to group CFS (i.e., SCHED_NORMAL and SCHED_BATCH) tasks.

These options need CONFIG_CGROUPS to be defined, and let the administrator create arbitrary groups of tasks, using the "cgroup" pseudo filesystem. See

Documentation/cgroups/cgroups.txt for more information about this filesystem.

When CONFIG_FAIR_GROUP_SCHED is defined, a "cpu.shares" file is created for each group created using the pseudo filesystem. See example steps below to create task groups and modify their CPU share using the "cgroups" pseudo filesystem.

```
# mkdir /dev/cpuctl
# mount -t cgroup -ocpu none /dev/cpuctl
# cd /dev/cpuctl

# mkdir multimedia      # create "multimedia" group of tasks
# mkdir browser         # create "browser" group of tasks

# #Configure the multimedia group to receive twice the CPU bandwidth
# #that of browser group

# echo 2048 > multimedia/cpu.shares
# echo 1024 > browser/cpu.shares

# firefox &     # Launch firefox and move it to "browser" group
# echo <firefox_pid> > browser/tasks

# #Launch gmplayer (or your favourite movie player)
# echo <movie_player_pid> > multimedia/tasks
```