This is the CFS scheduler.

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 - totally in parallel.

On real hardware, we can run only a single task at once, so while that one task runs, the other tasks that are waiting for the CPU are at a disadvantage - the current task gets an unfair amount of CPU time. In CFS this fairness imbalance is expressed and tracked via the per-task p->wait_runtime (nanosec-unit) value. "wait_runtime" is the amount of time the task should now run on the CPU for it to become completely fair and balanced.

(small detail: on 'ideal' hardware, the p->wait_runtime value would always be zero - 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->wait_runtime value and it is thus very simple: it always tries to run the task with the largest p->wait_runtime value. In other words, CFS tries to run the task with the 'gravest need' for more CPU time. So 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.

In practice it works like this: the system 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 deducted from p->wait_runtime. [minus the 'fair share' it would have gotten anyway]. Once p->wait_runtime gets low 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.

The rq->fair_clock value tracks the 'CPU time a runnable task would have fairly gotten, had it been runnable during that time'. So by using rq->fair_clock values we can accurately timestamp and measure the 'expected CPU time' a task should have gotten. All runnable tasks are sorted in the rbtree by the "rq->fair_clock - p->wait_runtime" key, and CFS picks the 'leftmost' task and sticks to it. As the system progresses forwards, newly woken 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.

Some implementation details:

- the introduction of 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 about them too much.

- sched_fair.c implements the 'CFS desktop scheduler': it is a

replacement for the vanilla scheduler's SCHED_OTHER interactivity code.

I'd like to give credit to Con Kolivas for the general approach here: he has proven via RSDL/SD that 'fair scheduling' is possible and that it results in better desktop scheduling. Kudos Con!

The CFS patch uses a completely different approach and implementation from RSDL/SD. My goal was to make CFS's interactivity quality exceed that of RSDL/SD, which is a high standard to meet :-) Testing feedback is welcome to decide this one way or another. [and, in any case, all of SD's logic could be added via a kernel/sched_sd.c module as well, if Con is interested in such an approach.]

CFS's design is quite radical: it does not use runqueues, 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 vanilla scheduler and RSDL/SD are affected).

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' and has no heuristics whatsoever. There is only one central tunable (you have to switch on CONFIG_SCHED_DEBUG):

```
/proc/sys/kernel/sched_granularity_ns
```

which can be used to tune the scheduler from 'desktop' (low latencies) to 'server' (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: both types of workloads should be isolated much more agressively than under the vanilla scheduler.

(another detail: due to nanosec accounting and timeline sorting, sched_yield() support is very simple under CFS, and in fact under CFS sched_yield() behaves much better than under any other scheduler i have tested so far.)

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

- reworked/sanitized SMP load-balancing: 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.

Group scheduler extension 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_FAIR_GROUP_SCHED strives to achieve exactly that. It lets
SCHED_NORMAL/BATCH tasks be be grouped and divides CPU time fairly among such
groups. At present, there are two (mutually exclusive) mechanisms to group
tasks for CPU bandwidth control purpose:

- Based on user id (CONFIG_FAIR_USER_SCHED)
  In this option, tasks are grouped according to their user id.
- Based on "cgroup" pseudo filesystem (CONFIG_FAIR_CGROUP_SCHED)
  This options lets the administrator create arbitrary groups
  of tasks, using the "cgroup" pseudo filesystem. See
  Documentation/cgroups.txt for more information about this
  filesystem.

Only one of these options to group tasks can be chosen and not both.

Group scheduler tunables:

When CONFIG_FAIR_USER_SCHED is defined, a directory is created in sysfs for
each new user and a "cpu_share" file is added in that directory.

```
# cd /sys/kernel/uids
# cat 512/cpu_share             # Display user 512’s CPU share
1024
# echo 2048 > 512/cpu_share     # Modify user 512’s CPU share
# cat 512/cpu_share             # Display user 512’s CPU share
2048
#
```

CPU bandwidth between two users are divided in the ratio of their CPU shares.
For ex: if you would like user "root" to get twice the bandwidth of user
"guest", then set the cpu_share for both the users such that "root"’s
cpu_share is twice "guest"’s cpu_share

When CONFIG_FAIR_CGROUP_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
```