CPU controller with flat runqueue

CFS, how does it work?

Rik van Riel, Facebook

CPU controller with flat runqueue

- Introduction to CFS
  - CPU time, vruntime, weight & load
  - Time slicing & preemption
- Current cgroup CPU controller
- CPU controller redesign
- Conclusions

Introduction to CFS

- Runtime, task priority & vruntime scaling
- Scheduling & min_vruntime
- Wakeup & placement
- Preemption
- Load & weight

Runtime, priority & vruntime

- CFS tracks “vruntime” per CFS sched_entity
  - Every task has a sched_entity
  - Every group has a sched_entity (w/ hierarchical rqs)
- vruntime is runtime, scaled by entity priority
  - Higher priority → slower vruntime
  - Lower priority → faster vruntime

```
se→vruntime += (NICE_0_LOAD / se→load.weight) * delta_exec;
```

Scheduling & min_vruntime

- Entities on a cfs_rq sorted by vruntime
- CFS runs the task with the smallest vruntime
  - Some exceptions, don’t worry about those now
  - Running causes vruntime to increase
  - Eventually another task is picked
- min_vruntime: the smallest vruntime of longer-running tasks on a CPU
  - Advances with the vruntime of running CFS tasks
    - Idle? Realtime? min_vruntime stays the same
  - Lower priority tasks make min_vruntime advance faster
  - More tasks make min_vruntime advance slower, as tasks “leapfrog”
  - cfs_rq→min_vruntime never decreases

Wakeup & placement

- At wakeup time, a task already has an se→vruntime
  - Smaller than cfs_rq→min_vruntime, if something else ran
  - cfs_rq→min_vruntime never goes backwards
- Task that was asleep may have vruntime advantage
  - Limit vruntime advantage to (half) a timeslice max
  - If multiple tasks wake up at once, min_vruntime may not advance for several time slices
  - Problem limited, because when many tasks are running each gets a shorter time slice

Preemption

- “Should the woken up task interrupt the currently running task?”
  - Take the vruntime difference (vdiff) between the tasks
  - Scale wakeup_granularity with the priority of the woken up task
  - If vdiff > wakeup_granularity, preempt current task
- Some problems:
  - vruntime affected by all recent tasks, not just current
  - Preemption affected by priority of woken up task, not current task
    - Low priority task preempts nobody (fine)
    - High priority task preempts everybody, even higher priority tasks (oops)

Load & weight

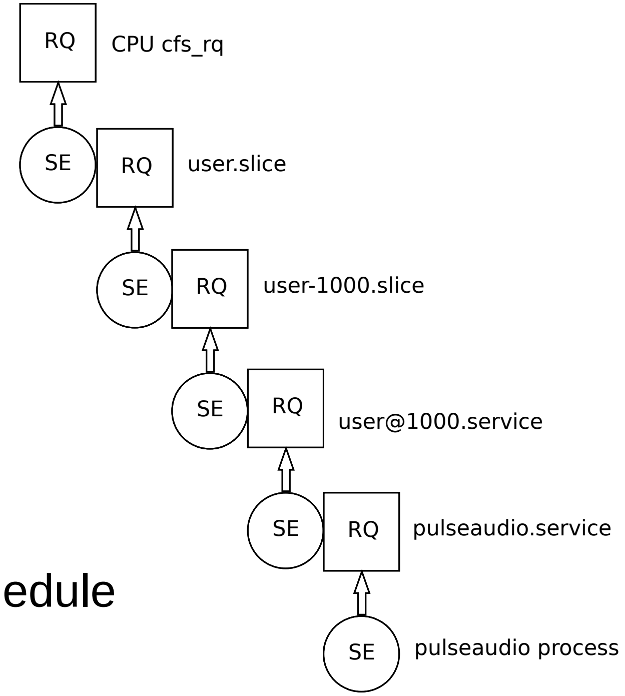
- se→load.weight priority of a sched_entity
  - For tasks, determined by nice level
- se→avg.load_avg == weight * duty cycle
  - Weight 1000, busy 1% of the time → load_avg = 10
  - Weight 100, busy 100% of the time → load_avg = 100
- cfs_rq→load.weight == sum of entity weights
- cfs_rq→runnable_load_avg ~= sum of entity weights
- cfs_rq→avg.load_avg ~= sum of se->load_avg

Current CPU controller

- Hierarchical runqueues
- Hierarchical load_avg calculation
- Task hierarchical load (task_h_load)

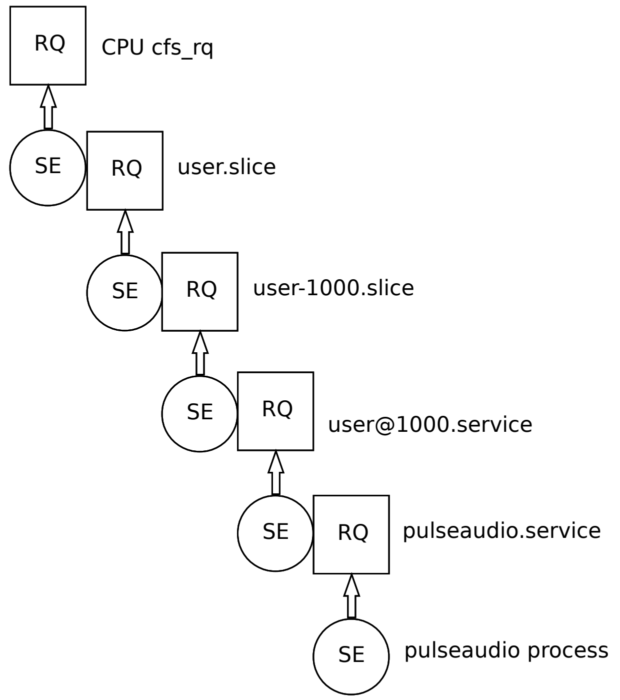
Current cgroup CPU controller

- Task has sched_entity (se)
- Group has se & cfs_rq
- Task se on group cfs_rq
- Group se on parent cfs_rq, etc...
- Build up entire hierarchy on wakeup
  - for_each_sched_entity() loops
  - Put each se on parent's cfs_rq, recalculate priorities
- Tear it back down when task sleeps
- Do vruntime accounting at each level, at every reschedule
- load_avg calculated periodically



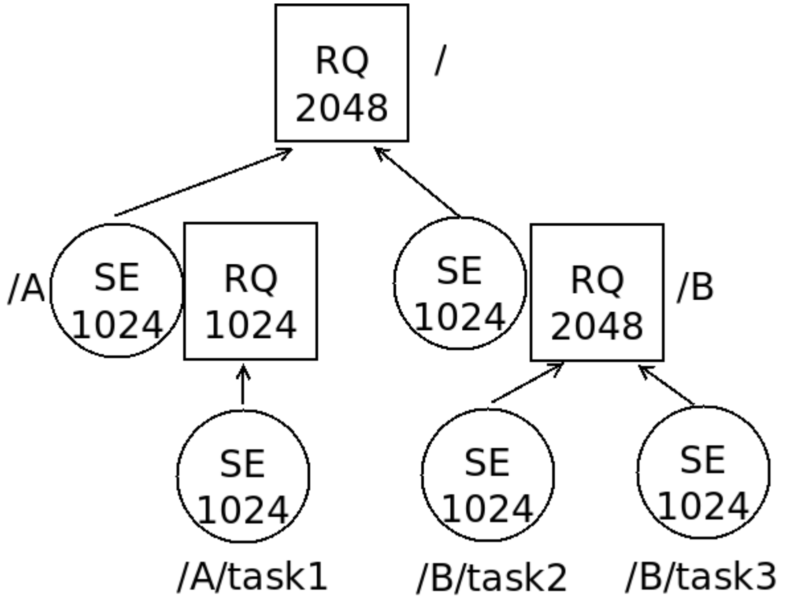
CPU controller load_avg

- Load average is propagated up
- At each level:
  - se→avg.load_avg ~= weight * %busy
  - cfs_rq→avg.load_avg
    - Sum of se→avg.load_avg for SEs on rq
    - Eg. 3 tasks on rq, or 2 tasks + 2 groups
- Cannot just add things up directly
  - 2 tasks each 50% busy, but simultaneously
  - Group only 50% busy as well!
  - Need to measure duty cycle at each level
    - “Is there something on this group’s runqueue?”
    - Periodic walk up from the bottom



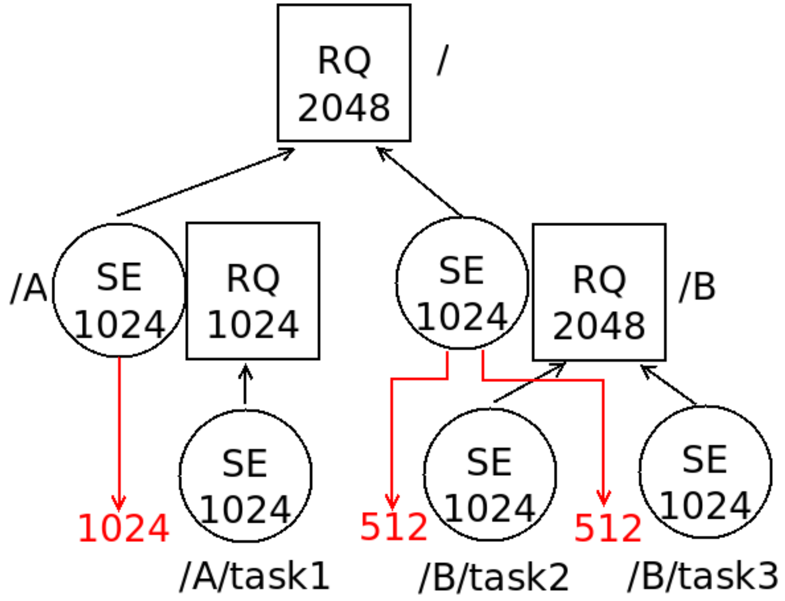
Task hierarchical load

- With hierarchical runqueues
  - A tree of groups and tasks
  - Groups A & B, of equal weight
  - Tasks 1, 2 & 3, “of equal weight”
  - But tasks 2 & 3 divide the group weight!



Task hierarchical load

- Assume 100% duty cycle
  - Load == weight (simplest case)
- Single SE inherits parent’s weight
- Multiple SEs split their parent’s weight
- task_h_load ~= %busy * hierarchical weight (!)
- task_h_weight functionally equivalent to nice level priorities!

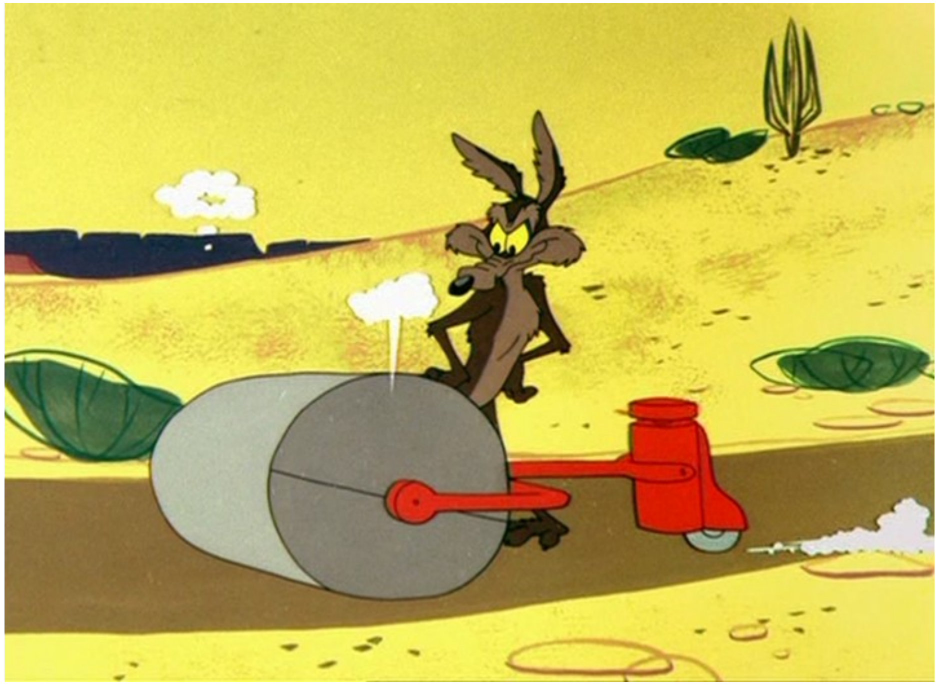


CPU controller redesign

- Flatten the runqueues!
- Basic design
- Pitfalls
- Performance

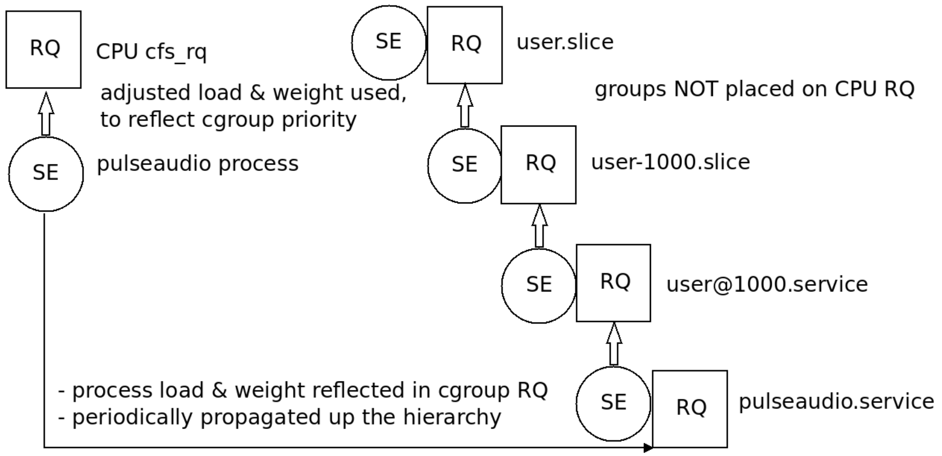
Flatten the runqueues!

- A plan emerges!
  - CFS can already split CPU time fairly between tasks of different priorities
  - We already calculate hierarchical load
  - Hierarchical priority code already rate limited, and it still works
  - We have almost everything we need to get rid of the hierarchical runqueues
- What could possibly go wrong?



New CPU controller

- Basic design
  - All tasks in root cfs_rq
  - Groups not placed on root cfs_rq
  - Rate limit hierarchy walks as much as possible
  - Use hierarchical load & weight for task priority
  - Scale vruntime with hierarchical task weight
  - Slight variation on vruntime formula


RQ
CPU cfs_rq
adjusted load & weight used, to reflect cgroup priority
SE
pulseaudio process
SE
RQ
user.slice
groups NOT placed on CPU RQ
SE
RQ
user-1000.slice
SE
RQ
user@1000.service
SE
RQ
pulseaudio.service
- process load & weight reflected in cgroup RQ
- periodically propagated up the hierarchy

se→vruntime += (NICE_0_LOAD / task_se_h_weight(se)) * delta_exec;

Pitfalls

- enqueue_task_fair needs careful ordering of operations, to avoid zero h_weight tasks
- Root cfs_rq→load.weight now identical to runnable_load_avg?
  - Can we get rid of one of them?
- Using group h_load to figure out task_se_h_weight
  - Load is a floating average, it will be 0 after a long sleep
  - Zero h_weight results in infinite vruntime, and also confuses the load balancer
  - Wrong hack gets around that, need better ideas
- Fixed point math + deep hierarchy → zero priority tasks
  - Could get zero length timeslice, infinite vruntime multiplier, even without bugs
  - Need full load resolution?

Pitfalls

- wakeup_preempt_entity only takes priority of woken task into account
  - High priority task preempts higher priority task
  - Have ugly hack in place to “fix” that
- Walks the hierarchy on every enqueue_task_fair to avoid zero priority.
  - Most overhead is here! How to reduce?
- Need complete redesign of CFS bandwidth control.
  - Have a design worked out…
- Most CPU use difference in benchmarks … is in USER space!
  - “What the scheduler causes others to do” is a larger effect than “what the scheduler does”, at least for many workloads
  - Makes it harder to evaluate scheduler changes.

Performance results

- Early results
  - Code tuned against my workloads, likely to change
  - Workload running “3 levels deep” (systemd hierarchy)
- Memcache style workload (~10k context switches/second/cpu)
  - Old code: CPU controller ~2-4% overhead
  - New code: CPU controller ~0.7% overhead
  - Most overhead in enqueue_task_fair hierarchy walking!
- Web serving workload
  - Two sets of servers receiving similar queries
  - ~1% lower CPU use with the new CPU controller code
- Worst case messaging workload
  - ~10% with the old code (still have to try new implementation)

Diffstat

- How much complexity does this new implementation add?

```
include/linux/sched.h |    8
kernel/sched/core.c   |    3
kernel/sched/debug.c  |   15
kernel/sched/fair.c   |  821 +++++++++++++++++++++---------------------------
kernel/sched/pelt.c   |   68 +---
kernel/sched/pelt.h   |    2
kernel/sched/sched.h  |    9
7 files changed, 367 insertions(+), 559 deletions(-)
```

Conclusions

- CPU controller has (unnecessary) overhead
- Reducing that overhead also simplifies CFS a little
- A number of unsolved questions remain
- Questions?