Completely Fair Scheduler (CFS)

Linux CPU Scheduling

Prakhar Jain

August 23, 2025

Operating Systems and Networks

Agenda

Ideal Fair Scheduling

What is Ideal Fair Scheduling?

- Imagine a perfectly divisible CPU that can run all tasks **truly simultaneously**.
- Each runnable task i receives exactly its proportional share $\frac{w_i}{\sum w}$ of CPU at every instant.
- No task ever falls behind its entitled share.

Why Ideal Fair Scheduling is Impossible

- Real CPUs are discrete: only one task runs per core at a time.
- Must approximate fairness over a time window.
- Context switches and timer ticks introduce overhead.
- CFS approximates the ideal by tracking **vruntime** and alternating tasks.

Key Idea

CFS simulates the *ideal fair scheduler* by ensuring no task lags too far behind in virtual time.

Big Picture

Why CFS?

- Goals: fairness, low latency for interactive tasks.
- Replaces O(1) scheduler since Linux 2.6.23 (2007) (why?). CFS has also been replaced now (why?).
- Core idea: **virtual runtime (vruntime)** makes CPU time comparable across tasks.
- *Fair* $\neq$ equal wall time: weight by priority/nice.
- No Heuristics
- Elegant handling of I/O and CPU bound processes.

Fairness by Virtual Time

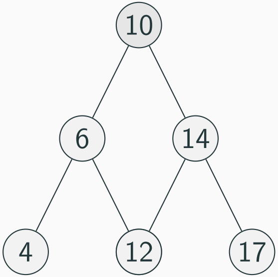
- Each runnable task has a weight w derived from `nice` (0: 1024; Δnice=+1 halves weight).
- **vruntime** increases with actual runtime scaled by $\frac{1024}{w}$.
- CFS always picks task with the **smallest vruntime** (most "unfairly treated").

$$\Delta v = \Delta t \cdot \frac{1024}{w(\texttt{nice})}$$

Runqueue Design

Data Structures

- One `rq` per CPU.
- Each `rq` maintains a **red-black tree** of runnable tasks keyed by `vruntime`.
- (timer interrupt happens) Leftmost node $\Rightarrow$ smallest vruntime $\Rightarrow$ next to run.
- Complexity: insert/remove $O(\log N)$, pick $O(1)$.



Picking the Next Task

- **enqueue**: insert task into RB-tree at `vruntime` = `max(task.v, rq.min)`.
- **dequeue**: remove current task when it blocks or exits.
- **pick_next_entity**: leftmost node of RB-tree.
- **preemption**: if a newly awakened task has smaller vruntime than current by a threshold.

Timing and Quanta

How Long Does a Task Run?

- No fixed timeslice; CFS targets **ideal fairness** within a window $T =$ `sched_latency_ns`.
- With N runnable tasks, ideal slice: $\frac{T}{N}$, but bounded by `min_granularity_ns`.
- Tickless kernels: periodic updates via hrtimers; `sched_tick()` maintains vruntime.

Key Knobs

sysctl	effect
`kernel.sched_latency_ns`	fairness window T
`kernel.sched_min_granularity_ns`	min slice
`kernel.sched_wake_up_granularity_ns`	preempt threshold

Sleep, Wake, and Interactivity

Sleep/Wake Path

- Blocking I/O: task dequeues; vruntime frozen.
- Wakeup: vruntime adjusted near current `rq.min_vruntime` to avoid *unfair head starts*.
- Interactive boost emerges naturally: sleepers do not accumulate vruntime while others do.

Preemption and Granularity

- Preempt current if $v_{new} + G < v_{curr}$, where G is wakeup granularity.
- Prevents *thrashing* between near-equal entities.
- Tunables balance latency (UI snappiness) vs throughput.

Priorities and Shares

Nice Levels and Weights

- Nice $\in [-20, 19]$ maps to weights w.
- Ratio of shares $= \frac{w_i}{\sum w}$ determines CPU fraction.
- Example: nice 0 vs nice 5: $\frac{1024}{335} \approx 3.05\times$ more CPU.

nice	weight
-5	3350
0	1024
5	335
10	110
15	36
19	15

Walkthrough

Mini Example

Three tasks A:B:C with weights 1024:1024:512.
Ideal shares: 40% : 40% : 20%.

1. Start: all $v = 0$. Pick A (leftmost). After Δt, $v_A = \Delta t$.
2. Insert back, pick B (now smallest v). After Δt, $v_B = \Delta t$.
3. Pick C; scaled by weight: $\Delta v_C = 2\,\Delta t$.
4. After several cycles, $v_A \approx v_B \approx v_C$ and observed CPU time follows shares.

Complexity and Trade-offs

Complexity

- Insert/erase: $O(\log N)$; pick leftmost: $O(1)$ (how?).
- Per-CPU state keeps cache locality.
- Overheads grow with runnable tasks per CPU (not threads per system).

Pros and Cons

Pros

- Strong fairness model.
- Good interactive latency.

Cons

- RB-tree adds $O(\log N)$ overhead.
- Tuning needed for extremes (HPC vs desktop).

Is CFS truly fair on multiprocessor systems?

References

Further Reading

- Ingo Molnár, Peter Zijlstra: CFS design discussions (LKML archives).
- *Linux Kernel Development* Robert Love.
- *Understanding the Linux Kernel* Bovet, Cesati.
- Linux kernel source: `kernel/sched/fair.c`.
- Man pages: `sched(7)`, `nice(1)`.
- Linux Implementation Details
- Overview of CFS
- The Linux Scheduler: a Decade of Wasted Cores

Questions?