The Linux Completely Fair Scheduler (CFS)

CMPU 334 – Operating Systems
Jason Waterman
The Linux Completely Fair Scheduler (CFS)

- Implements fair-share scheduling
  - Guarantees that each job obtains *a certain percentage* of CPU time

- Efficient and scalable
  - Quickly make a scheduling decision

- Scheduling performance is important
  - Scheduling uses about 5% of datacenter CPU time at Google
CFS Operation Basics

• Fairly divides a CPU evenly among all competing (runnable) processes
  • Doesn’t use a fixed time slice

• Uses the **virtual runtime** (vruntime) of a process
  • Accumulates as the process runs
  • To schedule a process, pick the one with the lowest vruntime

• When to schedule?
  • Frequent switches increase fairness but has a higher overhead
  • Fewer switches give better performance at the cost of fairness

• Controlled by **sched_latency** parameter
  • Maximum time a process can run before considering a switch (e.g., 20 ms)
  • This is divided by the number of runnable processes to get a process time slice
  • CFS will be completely fair over this time period
CFS Example

- `sched_latency` = 48 ms
- Four processes that are runnable to start
  - Per process time slice of 12 ms (48/4)
  - `vruntime` is starts at 0 for these jobs
- Pick job with the lowest `vruntime` (A, B, C, or D in this case)
- Run job A until it has used 12 ms of `vruntime`
  - Then make a scheduling decision
  - Run the job with the lowest `vruntime`
    - (B, C, or D)
- C and D complete after 96 ms
  - Time slice is adjusted to 24 ms (48/2)
Too many processes runnable?

- Per process time slice is the `sched_latency` / runnable processes
  - A lot of runnable processes could lead to small time slices
    - Lots of context switches and more overhead
- CFS `min_granularity` parameter
  - Minimum time slice of a process (e.g., 6 ms)
  - CFS will never set the time slice of a process to less than this value
  - In this case, CFS may not be perfectly fair over the target scheduling latency
    - E.g., `sched_latency` = 48 ms with 10 runnable processes
    - Time slice 4.8 --> 6 ms
    - All jobs won’t run during the 48 ms
- Timer interrupts
  - Time slices are variable, how to set the timer?
  - Timer goes off frequently (e.g., 1 ms)
  - Gives the CFS scheduler a chance to see if the current job has reached the end of its run
Niceness Levels

• Gives the user control over process priority
  • Give some processes a higher (or lower) share of the CPU

• A **nice** level of a process
  • A measure of how nice (to other processes) your job is
  • 19 (lowest priority)
  • -20 (highest priority)

• Nice levels are mapped to a weight used to compute an effective time slice for a process
Niceness Weightings

```c
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */ 9548, 7620, 6100, 4904, 3906,
    /* -5 */ 3121, 2501, 1991, 1586, 1277,
    /* 0 */ 1024, 820, 655, 526, 423,
    /* 5 */ 335, 272, 215, 172, 137,
    /* 10 */ 110, 87, 70, 56, 45,
    /* 15 */ 36, 29, 23, 18, 15,
};
```

time_slice<sub>k</sub> = \[
\frac{\text{weight}_k}{\sum_{n=0}^{n-1} \text{weight}_i} \cdot \text{sched_latency}
\]
Niceness Weighting Example

- Two processes, A and B
  - A’s niceness level is -5 (boost in priority)
  - B’s niceness level is 0 (default)
- Calculate the time slice for A and B
  - Weight A: 3121, weight B: 1024, total weight: 4145
  - Time slice A: \(\frac{3121}{4145} = 0.753 \times \text{sched\_latency}\)
  - Time slice B: \(\frac{1024}{4145} = 0.247 \times \text{sched\_latency}\)
- Assuming a 48 ms sched\_latency:
  - Process A gets about 75% of the sched\_latency (36 ms)
  - B gets about 25% of the sched\_latency (12 ms)
- Weight table is constructed to preserve CPU proportionally ratios when the difference in nice values is constant
  - E.g., if process A had a nice value of 5 and B had a nice value of 10, they would be scheduled the same way as above
Calculating vruntime

- Higher priority processes get a longer time slice
- But we pick the process with the lowest vruntime to run next
  - To handle priority properly, vruntime must scale inversely with priority
- For our example:
  - A’s vruntime will accumulate at about a 1/3 the rate of B’s

\[ \text{vruntime}_i = \text{vruntime}_{i-1} + \frac{\text{weight}_0}{\text{weight}_i} \cdot \text{runtime}_i \]
CFS efficiency

• How quickly can the scheduler find the next job to run
  • Lists don’t scale if you have 1000s of processes to search through every millisecond

• CFS keeps processes in a red-black tree
  • A type of balanced tree
  • Does a little extra work to maintain low depths
  • \( O(\log n) \) for operations (search, insert, delete)

• CFS only keeps running and runnable processes in this structure
  • If a process is waiting on I/O, it is removed from the tree and kept track elsewhere
  • What to do when process wakes up?
    • vruntime will be behind the others and could monopolize the CPU
    • CFS sets the vruntime for the job to the minimum value found in the tree
    • Jobs that sleep for short periods often do not ever get their fair share of the CPU
Linux CFS Summary

• Linux Completely Fair Scheduler (CFS)
  • Most widely used fair-share scheduler in existence
  • A bit like a weighted round-robin with dynamic time slices
  • Built to scale and perform well under load