

Discussion Feb. 10

Wednesday, February 10, 2025

Scheduling Review

Policy question in OS – which process gets to run?

Policy goals:

- ▶ Avoid starvation
- ▶ Minimize response time for interactive jobs
- ▶ Fairness?
- ▶ Performance?

Preemption

- ▶ Why might the kernel want to preempt a process?
- ▶ Which schedulers from class are preemptable?
- ▶ Which are not?

Scheduler metrics

Turnaround time: $T_{\text{finish}} - T_{\text{arrive}}$

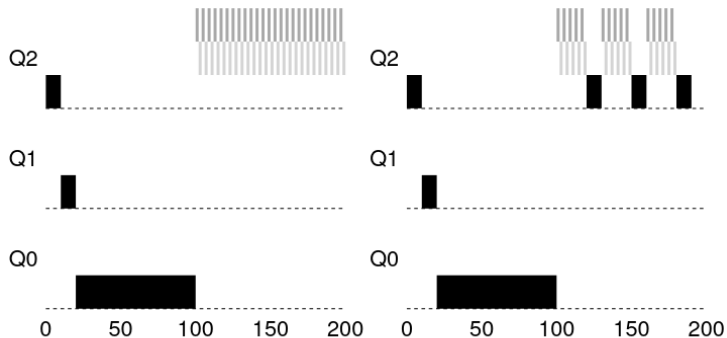
Response time: $T_{\text{scheduled}} - T_{\text{arrive}}$

Evaluating simple schedulers

Job	Arrival Time	Job Length
A	0	4
B	0	2
C	0	3

- ▶ RR scheduler: what is the avg response time for the 3 jobs?
- ▶ RR scheduler: what is the turnaround time for job B
- ▶ FIFO scheduler: what is avg turnaround time for the 3 jobs?

MLFQ Review



(OSTEP Chapter 8, Figure 4)

MLFQ Exercise

Book homework questions (see `cpu-sched-mlfq`)

<https://github.com/remzi-arpacidusseau/ostep-homework>:

1. How would you configure the scheduler parameters to behave just like a round-robin scheduler?
2. Craft a workload with two jobs and scheduler parameters so that one job takes advantage of the older Rules 4a and 4b (turned on with the `-S` flag) to game the scheduler and obtain 99% of the CPU over a particular time interval.
 - ▶ 4a: if a job uses its allotment while running, its priority is reduced
 - ▶ 4b: if a job gives up the CPU before it's allotment is up, it stays at the same priority level

Proportional Share Scheduling

Previous schedulers optimize for *turnaround* and *response* time.
What else should we think about when scheduling processes?

- ▶ Fair share of CPU

Lottery Scheduling

Algorithm

Scheduling via lottery:

- ▶ Each process is given tickets representing their proportional share.
- ▶ On each time slice, generate a random number.
- ▶ The value of the number determines the schedule.

Example (1000 total tickets):

Process	Tickets	CPU Share
P1	500	50%
P2	250	25%
P3	250	25%

Random number: 600. P2 is scheduled.

Lottery Scheduling (cont.)

Pros:

- ▶ Simple. Very little state needed in scheduler.
- ▶ Cooperating processes can voluntarily reduce their own tickets

Cons:

- ▶ Good fairness over the long term, but poor fairness over the short term.
- ▶ How many tickets should jobs be given? Who determines if a process is high or low priority.

Stride Scheduling

Proportional share without randomness.

Algorithm

Each process has a *stride* equal to $N / \text{tickets}$ for some large N .

- ▶ When a process is scheduled, increment *pass* by *stride*.
- ▶ Always schedule the process with the lowest *stride*.

Processes with more tickets have smaller stride. Therefore, they are scheduled more frequently.

Stride Scheduling (cont.)

What happens when a schedule doesn't run for awhile?

- ▶ E.g. process is blocked on I/O or sleeping.
- ▶ Process stride does not increase.
- ▶ Process will dominate CPU when it wakes up.

Solution: *global stride* and *global pass*.

- ▶ Global stride is $N / \text{all tickets}$.
- ▶ Global pass is updated by global stride on each timeslice.
- ▶ Process pass is set to global pass on wakeup.

This is the first taste of *virtual CPU time*.

Linux CFS

"Completely Fair Scheduler" in use in Linux until very recently.

- ▶ Idea: divide CPU cycles amongst processes in proportion to their weights.
- ▶ Weights: *nice* value, not tickets.

Context switch tradeoff:

- ▶ Time slices too small: good fairness, poor performance (too much context switching)
- ▶ Time slices too big: poor fairness, good performance.

Linux CFS - Deciding timeslice length

Set a scheduling interval of 48ms, or $6 * \text{number of processes}$.

1. During every interval, each process must run once. To determine each process's time slice length, divide the interval length by the number of processes proportional to their weight.
2. Linux keeps track of vruntime for each process. Processes are preempted when their vruntime exceeds their timeslice length.
3. Fairness is guaranteed within each scheduling interval.

Linux CFS - Choosing the next process

Organize processes in a red-black tree according to their vruntime. The process with smallest vruntime is always the left-most node.

- ▶ Imagine choosing the process with lowest vruntime in xv6 - linear scan through process array.
- ▶ For 100s or 1000s of processes, logarithmic lookup+insert time is important.

Advantage of choosing the minimum vruntime?

- ▶ Interactive jobs scheduled first