# Recap

Xv6 code tour: implementation of RR.

Multilevel Scheduling

- Multilevel Qs are defined based on priority; priority-based and RR scheduling are used.
- Each process is assigned the highest priority and thus put into the highest priority Q initially.
- After running on a Q for a time slice, its priority is downgraded (if possible) and it is moved to the Q one level lower.
- After a certain time period, all processes' priorities are boosted to the highest to avoid starvation.

Advantages: no need for burst-time oracle; no starvation; good performance in turnaround time and response time.

Open question: fairness?

# L6: Fair Scheduling

(based on Chapter 9)

Slides revised from
Matthew Tancreti
Iowa State University

# Fair Scheduling

We have seen how schedulers can optimize turnaround and response time

But we have also seen issues such as starvation that result in some processes getting little run time, even while the overall system has good turnaround and response time

How to make scheduling fair?

# Lottery Scheduler

Each process is assigned numbered “tickets”

Every time slice, scheduler randomly picks a winning ticket and the process holding the ticket runs for that time slice

Over time, a process' run time is proportional to percent of tickets it holds

# Example

Process A is assigned: tickets 0 ~ 24.
Process B is assigned: tickets 25~74.
Process C is assigned: tickets 75~99.

| Winning tickets | 63 | 85 | 70 | 39 | 76 | 17 | 29 | 71 | 36 | 39 | 10 | 99 | 68 | 83 | 23 | 62 | 43 | 0 | 49 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Resulting Schedule | | | | | | A | | | | | A | | | | A | | | A | | A |
| | B | | B | B | | | B | B | B | B | | | B | | | B | B | | B | |
| | | C | | | C | | | | | | | C | | C | | | | | | |

Over 20 times, A runs for 5 times, B for 11 times, and C for 4 times. The ratio 5:11:4 is close to 25:50:25 = 5:10:5 (the ratio between their tickets).

# Example Implementation

```
1  // counter: used to track if we've found the winner yet
2  int counter = 0;
3
4  // winner: use some call to a random number generator to
5  //         get a value, between 0 and the total # of tickets
6  int winner = getrandom(0, totaltickets);
7
8  // current: use this to walk through the list of jobs
9  node_t *current = head;
10 while (current) {
11     counter = counter + current->tickets;
12     if (counter > winner)
13         break; // found the winner
14     current = current->next;
15 }
16 // 'current' is the winner: schedule it...
```

# How to Assign Tickets?

In effect, the lottery scheduler is a kind of priority scheduler

- More tickets means higher priority
- But, fairer than priority, low ticket processes can't be completely starved

For example, how can we create the effect of SJF?

- Assign number of tickets inversely proportional to each job's runtime

But there are differences

- Pro: No job can be starved, everyone gets their fair share of runtime
- Con: A short job can be unlucky and have higher response time (probabilistic/nondeterministic)

# Stride Scheduling

Lottery has good fairness over the long run, but randomness can result in suboptimal choses over a short time

**Stride scheduling** is deterministic (not random) but has same fairness

For each process, set its **stride** as a large number (e.g., 10000) divided by its number of tickets

After every time a process runs, it's **pass** counter is incremented by its stride

Scheduler always picks the job with the lowest pass counter to run next

# Example

Process A is assigned: tickets 0 ~ 24 ==> Stride = 10000/25 = 400

Process B is assigned: tickets 25~74 ==> Stride = 10000/50 = 200

Process C is assigned: tickets 75~99 ==> Stride = 10000/25 = 400

| Pass (A) | Pass (B) | Pass (C) | Who runs? |
|---|---|---|---|
| 0 | 0 | 0 | A |
| 400 | 0 | 0 | B |
| 400 | 200 | 0 | C |
| 400 | 200 | 400 | B |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# Example

Process A is assigned: tickets 0 ~ 24 ==> Stride = 10000/25 = 400

Process B is assigned: tickets 25~74 ==> Stride = 10000/50 = 200

Process C is assigned: tickets 75~99 ==> Stride = 10000/25 = 400

| Pass (A) | Pass (B) | Pass (C) | Who runs? |
|---|---|---|---|
| 0 | 0 | 0 | A |
| 400 | 0 | 0 | B |
| 400 | 200 | 0 | C |
| 400 | 200 | 400 | B |
| 400 | 400 | 400 | A |
| 800 | 400 | 400 | B |
| 800 | 600 | 400 | C |
| 800 | 600 | 800 | B |
| 800 | 800 | 800 | |

# Stride vs Lottery

| Stride | Lottery |
| --- | --- |
| Deterministic | Nondeterminsitic (random) |
| Fairness in both long run and short run | Fairness in long run |
| Stateful (need to maintain for each process its state, i.e., its pass counter) | Stateless (no state maintained for process => resilient to process joining/leaving) and thus more simple |

# Linux Completely Fair Scheduler (CFS)

Design Goals:

- Fairness in sharing CPU by runnable processes
- Efficiency – the cost for scheduling should be low
- Scalability – should work efficiently for a large number of processes

# CFS: Basic Operation

Procedure:

- For each process, the OS keeps track its runtime (i.e., how much time it has run on CPU)
- Every certain time interval, the scheduler picks the runnable process with the lowest runtime

Question: how to determine the interval?

- Can’t be too long or too short

# CFS: how to determine the interval?

Two parameters: sched_latency & min_granularity

Typically, sched_latency = 48 ms. (Ideally, the time period for scheduling "all" runnable processes once.)

Sched_latency is divided equally into n slices, where n is typically the number of running/runnable processes

- For example, if n=4, each slice = 12ms.

Slice can't be too short: slice >= min_granularity, where min_granularity is another parameter.

- For example, if n=10 and min_granularity=6ms, each slice = 6ms.

For each slice, CFS picks the runnable process with the smallest runtime to run.

# Example

Let sched_latency = 48 ms, min_granularity = 24 ms, number of processes = 3.

==> each slice = 24 ms.

| Scheduling Time | Process A Runtime | Process B Runtime | Process C Runtime | Who runs? |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | A |
| 24 | | | | |
| 48 | | | | |
| 72 | | | | |
| 96 | | | | |
| 120 | | | | |
| 144 | | | | |

# Example

Let sched_latency = 48 ms, min_granularity = 24 ms, number of processes = 3.

==> each slice = 24 ms.

| Scheduling Time | Process A Runtime | Process B Runtime | Process C Runtime | Who runs? |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | A |
| 24 | 24 | 0 | 0 | B |
| 48 | 24 | 24 | 0 | C |
| 72 | 24 | 24 | 24 | A |
| 96 | 48 | 24 | 24 | B |
| 120 | 48 | 48 | 24 | C |
| 144 | 48 | 48 | 48 | … |

# CFS: what if processes are of different priorities?

Each process is assigned a "nice" value, an integer between –20 and +19.

Nice value is mapped to weight:

```
Static const int nice_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,
                     /* 10*/     110,      87,       70,      56,      45,
                     /* 15*/       36,      29,       23,      18,      15,
};
```

Larger Nice (i.e., Nicer) <===> Smaller Weight (i.e., lower priority)

# CFS: dynamic time_slice

Instead of assigning the same-length slide for each process. For each process k:

**Time_slice k = sched_latency * weight k / (weight 0 + weight 1 + … weight n-1)**

**A process with larger weight gets longer time_slice per sched_latency!**

# CFS: dynamic runtime (virtual runtime, i.e., vruntime)

Instead of counting the physical (actual) runtime for each process, we count virtual runtime. For each process k:

**vruntime k = vruntime k + runtime * weight 0 / weight k**

**A process with larger weight has shorter vruntime counted per shed_latency, though the physical runtime is longer!**

# CFS: dynamic time_slice and vruntime

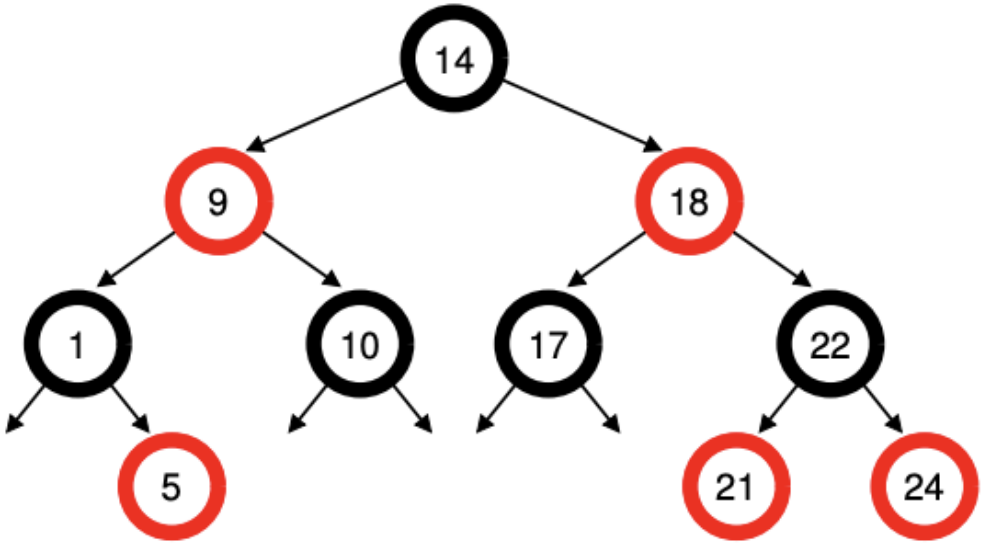
With the mechanisms of dynamic time_slice and dynamic vruntime, processes with higher priority (i.e., lower nice value, larger weight) gain more advantage – they are allocated with more CPU time!

Really fair??

# CFS: Efficient Sorting of Processes

At each scheduling time, the runnable process with the shortest vruntime should be picked.

- CFS uses red-black tree (self-balancing binary search tree) to quickly find process with lowest vruntime.



# CFS: what if a new process join?

Stateful not Stateless! - vruntime is tracked for each process.

Newer process will gain advantage?

To address this issue, the vruntime of a new process is set to the shortest one of the existing runnable processes.