Chapter 9: Scheduling: Proportional Share

This chapter introduces a different class of scheduling algorithms known as **proportional-share** or **fair-share** schedulers. Instead of optimizing for metrics like turnaround or response time, these schedulers aim to guarantee that each job receives a specific percentage of the CPU's time.

> **The Crux of the Problem: How to Share the CPU Proportionally?**
>
> How can we design a scheduler to share the CPU in a proportional manner? What are the key mechanisms for doing so? How effective are they?

1. Lottery Scheduling

An excellent early example of this approach is **Lottery Scheduling**. It's a simple, randomized algorithm that achieves proportional share probabilistically.

1.1. Basic Concept: Tickets Represent Your Share

- **Tickets:** Each process is assigned a number of "tickets" to represent its desired share of a resource. The percentage of total tickets a process holds corresponds to its percentage of the resource it should receive.
- **The Lottery:** At each time slice, the scheduler holds a lottery.
  1. It determines the total number of tickets across all ready processes.
  2. It picks a random "winning ticket" number from 0 to (total tickets - 1).
  3. The process holding the range that includes the winning number is chosen to run for that time slice.
- **Example:** Process A has 75 tickets (0-74) and Process B has 25 tickets (75-99). Total tickets = 100.
  - If the winning ticket is 63, Process A runs.
  - If the winning ticket is 85, Process B runs.
- **Probabilistic Nature:** Over a long period, the number of times each process wins the lottery will approximate its ticket proportion. However, over short periods, the results can be unfair due to randomness.

1.2. Ticket Mechanisms

Lottery scheduling provides several useful mechanisms for managing tickets:

- **Ticket Currency:** Allows a user to allocate tickets to their own processes in a local "currency." The OS then automatically converts this local currency into the correct global ticket value. This enables groups of processes to manage their shares independently.
- **Ticket Transfer:** A process can temporarily hand off its tickets to another. This is useful in client-server interactions, where a client can transfer its

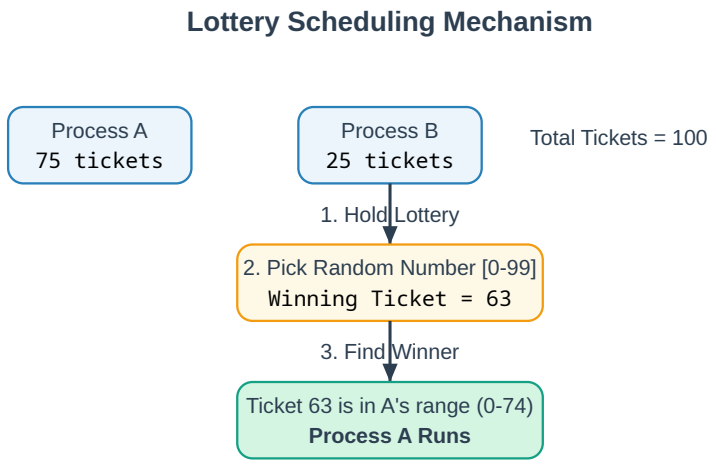

Figure 1: Lottery Scheduling Mechanism

tickets to a server to prioritize the handling of its request.

- **Ticket Inflation:** A process can temporarily increase its own ticket count. This is only useful in trusted environments where processes cooperate; otherwise, a greedy process could monopolize the CPU.

1.3. Implementation

The implementation is remarkably simple. It requires:

1. A random number generator.
2. A data structure to track processes and their tickets (e.g., a list).
3. A counter for the total number of tickets.

   => To find the winner, the scheduler picks a random number and then iterates through the process list, summing up tickets until the counter exceeds the winning number. The current process at that point is the winner. Sorting the list by ticket count can optimize this traversal.

2. Stride Scheduling

Stride Scheduling is a **deterministic** alternative to lottery scheduling, designed to achieve fair-share proportions more precisely.

- **Stride:** Each job is assigned a `stride` value, which is inversely proportional to its number of tickets. A common way to calculate it is `stride = BigNumber / tickets`.

  - A job with more tickets has a smaller stride.
- **Pass Value:** Each job also has a `pass` value, which tracks its progress. It starts at 0.
- **Policy:**
  1. The scheduler picks the process with the **lowest pass value** to run.
  2. When the process runs for a time slice, its pass value is incremented by its stride (`pass += stride`).
  3. This cycle repeats.
- **Example:** Jobs A (100 tickets), B (50), C (250). Let `BigNumber` = 10,000.
  - Strides: A=100, B=200, C=40.
  - Pass values all start at 0.
  - C runs (pass -> 40), then C runs again (pass -> 80). Then A runs (pass -> 100). Then C runs again (pass -> 120), and so on.
  - After 8 time slices, C will have run 5 times, A twice, and B once, perfectly matching their ticket proportions (5:2:1).

Stride Scheduling Trace

A: 100 tix (stride=100), B: 50 tix (stride=200), C: 250 tix (stride=40)

Pass(A)	Pass(B)	Pass(C)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Policy: Pick job with lowest pass value.
Update: `pass += stride`

Figure 2: Stride Scheduling Trace

- **Stride vs. Lottery:**
  - **Stride Scheduling** is deterministic and perfectly proportional over scheduling cycles. However, it requires maintaining global state (the pass value for each process). Adding a new job is complicated, as its initial pass value must be set carefully to avoid monopolizing the CPU.
  - **Lottery Scheduling** is probabilistic and simpler. It requires no global state per process, making it easy to add new jobs.

3. The Linux Completely Fair Scheduler (CFS)

The default scheduler in modern Linux, CFS, implements proportional sharing with a focus on efficiency and scalability.

3.1. Basic Operation: Virtual Runtime (vruntime)

- **Core Idea:** Instead of a fixed time slice, CFS aims to give each process an equal share of the CPU. It does this by tracking the **virtual runtime (`vruntime`)** for each process.
- **Policy:** CFS always picks the process with the **lowest `vruntime`** to run next.
- **`vruntime` Accumulation:** As a process runs, its `vruntime` increases. In the simplest case, it increases in proportion to physical time.

3.2. Managing the Time Slice

- **`sched_latency`:** A target value (e.g., 48ms) over which CFS aims to be perfectly fair. The time slice for each process is calculated as `sched_latency / n`, where `n` is the number of running processes.
- **`min_granularity`:** A minimum time slice value (e.g., 6ms). This prevents the time slice from becoming too small when many processes are running, which would lead to excessive context-switching overhead.

3.3. Weighting (Niceness)

- CFS allows users to influence scheduling priority using the `nice` value (-20 to +19).
- A lower `nice` value means higher priority.
- The `nice` value is mapped to a `weight`. A process with a higher weight gets a larger proportion of the CPU.
- The `vruntime` of a process now accumulates more slowly if it has a higher weight (lower `nice` value), causing the scheduler to pick it more often.
  - $vruntime_i += (\frac{weight_0}{weight_i}) \cdot runtime_i$

3.4. Efficient Implementation: Red-Black Trees

- To efficiently find the process with the minimum `vruntime`, CFS does not use a simple list. Instead, it stores all runnable processes in a **red-black tree**, a self-balancing binary search tree.
- This ensures that finding the next job to run and updating the tree are very fast, logarithmic-time operations (`O(log n)`), which is crucial on systems with thousands of processes.

3.5. Handling Sleeping Processes (I/O)

- **The Problem:** A process that sleeps for a long time (e.g., waiting for I/O) will have a very low `vruntime` when it wakes up. If not handled carefully,

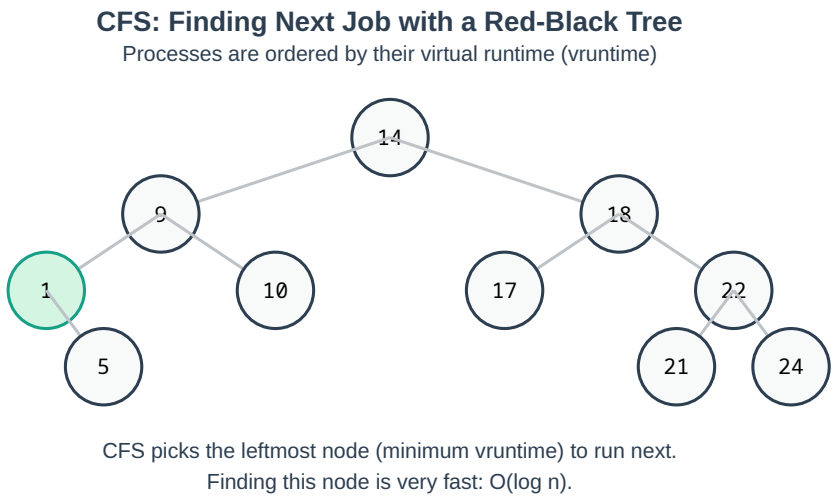

Figure 3: CFS using a Red-Black Tree

it could monopolize the CPU to "catch up," starving other processes.

- **CFS Solution:** When a sleeping process wakes up, its `vruntime` is not left at its old value. Instead, it is set to the **minimum `vruntime`** currently in the red-black tree of running jobs. This prevents starvation but can be unfair to short-lived I/O jobs.

---

4. Summary

Proportional-share schedulers offer an alternative to optimizing for turnaround or response time.

- **Lottery Scheduling** uses randomness to provide probabilistic fairness and is simple to implement.
- **Stride Scheduling** provides deterministic fairness but requires managing more state.
- **Linux CFS** is a highly efficient, scalable, and widely used fair-share scheduler that uses `vruntime` and a red-black tree to achieve its goals.

While fair-share schedulers are excellent for environments like virtualized data centers where CPU percentages need to be explicitly allocated, they can struggle with I/O-heavy workloads and leave the difficult problem of "ticket assignment" (or setting `nice` values) to the user.