【OS】Day15

【Ch9】Scheduling: Proportional Share(2)

9.6 Stride Scheduling

Each job in the system has a stride, which is inverse in proportion to the number of tickets it has.

In our example, jobs A, B, and C has 100, 50, and 250 tickets respectively, we can compute the stride of each by dividing some large number by the number of tickets each process has been assigned.

For example, if we divide 10,000 by each of those ticket values, we obtain the following stride of each process100, 200, and 40; every time a process runs, we will increment a counter for it(called its pass value) by its stride to track its global progress.

The scheduler then uses the stride and pass to determine which process should run next.

The basic idea is simple: at any given time, pick the process to run that has the lowest pass value so far; when you run a process, increment its pass counter by its stride.

A pseudocode implementation:

```
curr = remove_min(queue); //pick client with min pass
schedule(curr); //run for quantum
curr->pass += curr->stride; //update pass using stride
insert(queue, curr); //return curr to queue
```

Execution of A, B, and C:

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	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	...

Figure 9.3: **Stride Scheduling: A Trace**

Lottery Scheduling vs. Stride Scheduling

Lottery scheduling achieves the proportions probabilistically over time; stride scheduling gets them exactly right at the end of each scheduling cycle.

We might be wondering: *why use lottery scheduling at all given the precision of stride scheduling?*

Imagine a new job enters in the middle of our stride scheduling example above, what should its pass value be? Should it be set to 0?

If so, it will monopolize the CPU.

With lottery scheduling, there is no global state per process; we simply ass a new process with whatever tickets it has, update the single global variable to track how many total tickets we have, and go from there.

9.7 The Linux Completely Fair Scheduler(CFS)

The current Linux approach achieves similar goals in an alternate manner.

The scheduler entitled the Completely Fair Scheduler implements fair-share scheduling, but does so in a highly efficient and scalable manner.

Basic Operation

The goal of CFS is simple: to fairly divide a CPU evenly among all competing processes. It does so through a simple counting-based technique known as virtual runtime(`vruntime`).

As each process runs, it accumulates `vruntime`. In the most basic case, each process's `vruntime` increases at the same rate, in proportion with physical(real) time. When a scheduling decision occurs, CFS will pick the process with the lowest `vruntime` to run next.

This raise a question: *how does the scheduler know when to stop the currently running process, and run the next one?*

The tension here is clear: if CFS switches too often, fairness is increased but at the cost of performance(too much context switching); if CFS switches less often, performance is increased but at the cost of near-term fairness.

CFS manages this tension through various control parameters. The first is `sched_latency`. CFS uses this value to determine how long one process should run before considering a switch(effectively determining its time slice but in a dynamic fashion).

A typical `sched_latency` value is 48 milliseconds; CFS divides this value by the number of processes running on the CPU to determine the time slice for a process, and thus ensures that over this period of time, CFS will be completely fair.

For example, if there are n = 4 processes running, CFS divides the value of s`ched_latency` by n to arrive at a per-process time slice of 12 ms. CFS then schedules the first job and runs it until it has used 12ms of (virtual) runtime, and then checks to see if there is a job with lower `vruntime` to run instead.

But what if there are “too many” processes running? Wouldn’t that lead to too small of a time slice, and thus too many context switches?

To address this issue, CFS adds another parameter, `min_granularity`, which is usually set to a value like 6 ms. CFS will never set the time slice of a process to less than this value, ensuring that not too much time is spent in scheduling overhead.

Weighting(Niceness)

CFS also enables controls over process priority, enabling users or administrators to give some processes a higher share of the CPU.

It does this not with tickets, but through a classic UNIX mechanism known as the nice level of a process. The nice parameter can be set anywhere from -20 to +19 for a process, with a default of 0.

Positive nice values imply lower priority and negative values imply higher priority.

CFS maps the nice value of each process to a weight, as shown here:

```
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,
};
```

The weights allows us to compute the effective time slice of each process, but now accounting for their priority differences.

$$\text{time\_slice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \cdot \text{sched\_latency}$$

For example, assume there are two jobs, A and B. A is given a higher prioity by assigning ti a nice value of -5; B has the default priority(0).

This means that $weight_A$ is 3121, whereas $weight_B$ is 1024. If we compute the time slice of each job, we can get A's time slice is about 3/4 of `sched_latency`(36 ms), and B's about 1/4(12ms).

The way CFS calculates `vruntime` must also be adapted.

The following is our new formula:

$$\text{vruntime}_i = \text{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \cdot \text{runtime}_i$$

The `vruntime` scales inversely by the weight of the process.