【OS】Day14

【Ch9】Scheduling: Proportional Share

In this chapter, we'll examine a different type of scheduler known as a proportional-share scheduler, also sometimes referred to as a fair-share scheduler.

Proportional-share is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain pcercentage of CPU time.

Crux: How to share the CPU proportionally?

How can we design a scheduler to share the CPU in a propotional manner? What are the key mechanisms for doing so?

9.1 Basic Concept: Tickets Represent Your Share

Underlying lottery scheduling is one very basic concept: tickets, which are used to represent the share of a resource that a process should receive.

The percent of tickets that a process has represents its share of the system resource in question.

Let's look at an example. Imagine two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, we would like A to receive 75% of the CPU and B the remaining 25%.

Lottery scheduling achieves this probabilitistically by holding a lottery every so often(time slice). Holding a lottery is straightforward: the scheduler must know how many total tickets there are. The scheduler then picks a winning ticket, which is a number from 0 to 99.

Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs. The scheduler then loads the state of that winning

process and runs it.

Here is an example output of a lottery scheduler's winning tickets:

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 12
```

Here is the resulting schedule:

```
A     A  A     A  A  A  A  A  A     A     A  A  A  A  A  A
   B        B                    B     B
```

9.2 Ticket Mechanism

Lottery scheduling also provides a number of mechanisms to manipulate tickets in different and someimtes useful ways.

One way is with the concept of ticket currency. Currency allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like; the system then automatically converts said currency into the correct global value.

For example, assume users A and B have each been given 100 tickets.

- User A is running two jobs, A1 and A2, and gives them each 500 tickets(out of 1000 total) in A's currency.
- User B is running only 1 job and give it 10 tickets(out of 10 total).
- The system converts A1's and A2's allocation from 500 each in A's currency to 50 each in the global currency. Similarly, B1's 10 tickets is converted to 100 tickets.

```
User A -> 500 (A's currency) to A1 ->  50 (global currency)
       -> 500 (A's currency) to A2 ->  50 (global currency)
User B ->  10 (B's currency) to B1 -> 100 (global currency)
```

Another userful mechanism is ticket transfer. With transfers, a process can temporarily hand off its ticekts to another process. By transfering tickets to another process, the

performance of the transfered tickets can be boosted.

Finally, ticket inflation can sometimes be a useful technique. With inflation, a process can temporarily raise or lower the number of tickets it owns.

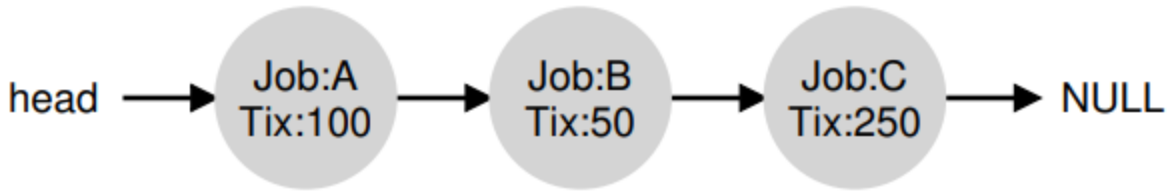
Of course, in a competitive scenario with processes that do not trust one another, this makes little sense; one greedy process could give itself a vast number of tickets and take over the machine.

Rather, inflation can be applied in an enviornment where a group of processes trust one another; in such case, if any one process knows it needs mroe CPU time, it can boost its ticket value as a way to reflect that need to the system, all without communicating with any other pcoesses.

9.3 Implementation

All we need to implement the proportional share is a good random number generator to pick the winning ticket, a data structure to track the processes of the system, and the total number of tickets.

Let's asuume we keep the processes in a list. Here is an example comprised of three processes, A, B, and C, each with some number of tickets.



To make a scheduling decision, we first have to pick a random number from the total number of tickets. If we pick 300, we simply traverse the list, with a simple counter used to help us find the winner.

The code walks the list of processes, adding each ticket value to counter until the value exceeds winner. Once that is the case, the current list element is the winner.

With our example of the winning ticket being 300, the following takes place:

1. First, counter is incremented to 100 to account for A's tickets; because 100 is less than 300, the loop continues
2. Then counter would be updated to 150(B's tickets), still less than 300 and thus again we continue
3. Finally, counter is updated to 400(greater than 300), and thus we break out of the loop with current pointing at C(the winner).

To make this process most efficient, it might generally be best to organize the list in sorted order, from the highest number of tickets to the lowest.

The ordering does not affect the correctness of the algorithm; however, it does ensure in general that the fewest number of list iterations are taken, especially if there are a few processes that possess most of the tickets.

Note: As the job lengths get larger, the fairness increases.