7. Lottery Scheduling

Proportional Share scheduler (= fair-share scheduler)

- Instead of optimizing for turnaround or response time, try to guarantee that Each Job Obtain a **Certain Percentage of CPU time**
- lottery scheduling / stride scheduling / CFS (Completely Fair Scheduler)

How to share the CPU Proportionally?

Lottery scheduling

- basic idea :
  - every so often, hold a lottery to determine which process should get to run next (Probabilistically / not deterministically)
  - processes that should run more often should be given more chances to win the lottery

- tickets : represent the share of a resource that a process should receive

- The scheduler must know how many total tickets there are
- The scheduler then picks random number

- The use of randomness in lottery scheduling leads to a probabilistic correctness in meeting the desired proportion, but no guarantee. => the longer , more likely achieve the desired percentages

Use Randomness

1. Avoids strange corner-case behaviors that a more traditional algorithm may have trouble handling.
   - Ex) LRU attains worst-case performance for some cyclic-sequential workloads
2. Lightweight
3. can be quite fast

Use tickets to represent shares : to represent a proportion of ownership

Ticket Mechanisms

ticket currency

```
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)
```

A has 1000 tickets in total, B has 10 tickets in total

ticket transfer

with transfers, a process can temporarily hand off its tickets to another process

- especially useful in a client/server setting
  - to speed up the work, the client can pass the tickets to the server and try to maximize the performance of the server while the server is handling the client's request.
  - when finished, the server then transfers the tickets back to the client

ticket inflation

with inflation, a process can temporarily raise or lower the number of tickets it owns.

- Where a group of processes trust one another,
  if any one process knows it needs more CPU time, it can boost its ticket value as a way to reflect that need to the system, all without communicating with any other processes.
- OF COURSE, NOT IN a competitive scenario with processes that do not trust one another

Implementation of Lottery scheduling

- random number generator to pick the winning ticket
- a data structure to track the processes of the system (ex. list)
- total number of tickets

- to make this process most efficient
  : organize the list in sorted order, from the highest number of tickets to the lowest.
  - ensure fewest number of list iterations are taken, especially if there are a few processes that possess most of the tickets
  - the ordering does not affect the correctness of the algorithm

Fairness metric

ratio

F : the time first job completes / the time that the second job completes
a perfectly fair scheduler would achieve F = 1

When the job length isn't very long, average fairness can be quite low.
Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired fair outcome.

How to Assign tickets?

how the system behaves is strongly dependent on how tickets are allocated
=> assume : user know best. user can allocate tickets to any jobs

Randomness occasionally will not deliver the exact right proportions, especially over short time scales.
=> Stride scheduling, a deterministic fair-share scheduler

Stride Scheduling

- stride : inverse in proportion to the number of tickets it has
  - many tickets (important) => small stride,
  - less tickets (less important)-> big stride
  - to see progress of each process

- pass value : every time a process runs, increased a counter by its stride / to track its global progress

- Scheduler pick the process to run that has the lowest pass value so far => process with many tickets likely chosen
  => Exactly in proportion to their ticket values.

Stride Scheduling vs. Lottery scheduling

- Lottery scheduling achieves the proportions **probabilistically** over time
- Sitride scheduling gets them exactly right at the end of each scheduling cycle (**deterministic**)

Seems like stride scheduling is better than lottery. **Why use lottery?**
: lottery scheduling doesn't have **global state**.

- If a new job enters in the middle of stride scheduling
  - its pass value be set as 0? => monopolize the CPU
- In lottery scheduling : add a new process with whatever tickets it has, update the single global variable to track how many total tickets we have

=> lottery makes it much easier to incorporate new processes in a sensible manner.

The Linux Completely Fair Scheduler (CFS)

- implements fair-share scheduling, in a highly efficient and scalable manner
- aims to spend very little time making scheduling decisions, through both
  - its inherent design
  - its clever use of data structures well-suited to the task (red black tree)

Scheduler efficiency is surprisingly important

Virtual runtime (vruntime)

- counting-based technique
- 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

How does the scheduler know
when to stop the currently running process & run the next one?

- if CFS switches too often, fairness is increased / at the cost of performance(too much context switching)
- if CFS switches less often, performance is increased (reduced context switching) / at the cost of near-term fairness

sched_latency : how long the time slice for a process
divide the time slice & round robin

- effectively determining its time slice but in a dynamic fashion
- a typical value of sched_latency : 48 milliseconds
- CFS divides sched_latency / the number(n) of processes running on the CPU

48

But what if there are too MANY processes running?
=> set low limit : min_granularity.

- Never set the time slice less than this => ensuring not too much time is spent in scheduling overhead

CFS utilizes a **periodic timer interrupt**
= > only make decisions at fixed time intervals

- give CFS a chance to wake up and determine if the current job has reached the end of its run

CFS enables controls over **process priority**
to give some processes a higher share of the CPU
=> through **nice** level of a process

- positive nice value : lower priority
- negative : higher

When you're too nice, you just don't get as much (scheduling) attention

CFS maps the nice value of each process to a weight

```
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,
};
```

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

=> effective time slice of each process + accounting for their priority differences

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

Why we use vruntime and runtime seperately?

- runtime : time the process just ran (say, for one time slice)
- we scale runtime to reward or penalize the process based on its weight(priority)
- The updated vruntime tells the scheduler how much CPU this process has had in virtual fairness terms

=> if we only use runtime, priority is not considered

High priority processes accumulate vruntime slower, so they stay in the running longer and more often

CFS is Completely Fair Scheduler because it

- divides CPU time fairly (time slices based on weight)
- tracks fairness accurately (vruntime adjusted by weight)
- Balances responsiveness and priority (nice values respected, starvation avoided)

=> It's "fair" not in equal time, but in proportional time based on priority

CFS adjust both

- **time slice**
- **vruntime**

for each process to reflect its priority

is a bit like weighted round-robin with dynamic time slices, but built to scale and perform well under load

Using Red-Black Trees

- one major focus of CFS is efficiency
- when the scheduler has to find the next job to run, it should be fast
  - Simple data structures like lists don't scale => keeping processes in a red-black tree

Unlike simple binary tree,
which can degenerate to list-like performance under worst-case insertion patterns
balanced trees do a little extra work to maintain low depths => ensure logarithmic(and not linear) in time

in CFS, only running(or runnable) processes are kept red-black tree.

Dealing with I/O and sleeping Processes

one problem with picking the lowest vruntime to run next : jobs that have gone to sleep for a long period of time. : has small vruntime

=> That job will monopolize the CPU, effectively starving other processes.
consider if their priority levels are same and it was its choice to go to bed FOR A LONG TIME

=> HOW TO SOLVE : altering the vruntime of a job when it wakes up

- set it to the minimum value found in the tree(=among running jobs)

So it does not get too much priority, and still gets fair access

This way, CFS avoid starvation, but not without a cost
: jobs that sleep for short periods of time frequently do not ever get their fair share of the CPU
even for short time sleep => disadvantage on vruntime as reset => long-term unfairness

When picking a data structure for a system you are building, carefully consider its access patterns and its frequency of usage