비율에 대해서

Scheduling: Proportional Share

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Proportional-Share Scheduler

Job 마다의 특색에 따라 달라지지.

- **Proportional-share scheduler is based around a simple concept**
  - A schedule might try to make each job to obtain a certain percentage of CPU time, instead of optimizing for turnaround or response time
  - Sometimes referred to as a fair-share scheduler or lottery scheduling
- **Underlying lottery scheduling is one very basic concept: tickets**
  - The percent of tickets of a process represents its share of the system resource

- **e.g.) Processes A and B have 75 and 25 tickets, respectively, which means A to receive 75% and B to receive 25% of the CPU**
  - Lottery scheduling achieves this probabilistically (but not deterministically) by holding a lottery every time slice ➔ A scheduler must know # of total tickets
  - Then, the scheduler picks a winning ticket (0 ~ 99 of total 100 tickets) (random number generator.)

Winning Tickets	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63	62	43	0	49	12
Resulting Scheduling	A		A	A		A	A	A	A	A	A		A		A	A	A	A	A	A
		B			B							B		B						

공평하다면!

  - 16:4 = 80%:20% (not exactly 75%:25%) → the longer these two jobs compete, the more likely they are to achieve the desired percentages

"fairness"

75	25
0 ~ 74	75 ~ 99

~~Center~~

50

100

100 ''

99

10 / 20

0.5

5

Ticket Mechanisms

100 대신 해줘

화폐

- **Lottery scheduling provides mechanisms to manipulate tickets in different and useful ways: ticket currency, transfer, inflation**
- **Currency allows a user to allocate tickets among their own jobs in whatever currency they are using**

은행체제가 발행

  – The system automatically converts it into the correct global currency
  – e.g.) Users A, B have 100 tickets each; total 200 tickets (global currency)
    global currency
    A has two jobs (A1, A2: 500 tickets each), B has one job (B1: 10 tickets)

local 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)
```

티켓을 줄 수도 있다

- **With transfer, process temporarily hand off its tickets to another**
  – Useful in client-server computing; client sends tickets to server to maximize the server's performance while the server is processing the client's job
- **With inflation, process temporarily raise/lower # of tickets it owns**

30→50 (프로세스간의 믿음이 있어야 한다)

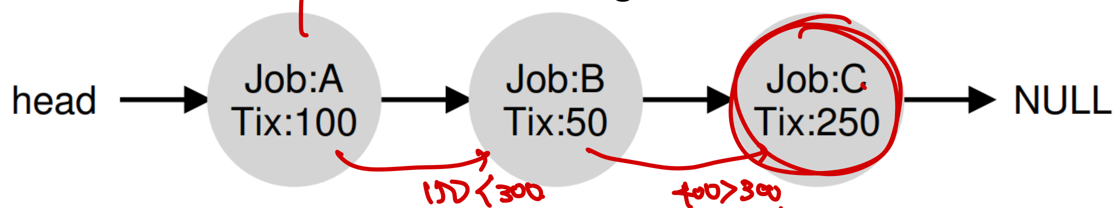
  – Inflation can be applied in environment where group of processes trust each other
  – If any one process knows it needs more CPU time, it can boost its ticket value

Implementation

desc order로 하면 search하는 비용은 줄일수 있음

250 → 100 → 50

- **Lottery scheduling is easy to implement**
  - Example of process A with 100, B with 50, C with 250 tickets (total 400 tickets)
  - Let's say we pick the number 300 from random number generator



  - The code walks the list of processes, adding each ticket value to **`counter`** until the value exceeds **`winner`**
  - Counter value changes:
    A, 100 (<300) →
    B, 150 (<300) →
    C, 400 (>300)
  - Process C is the winner

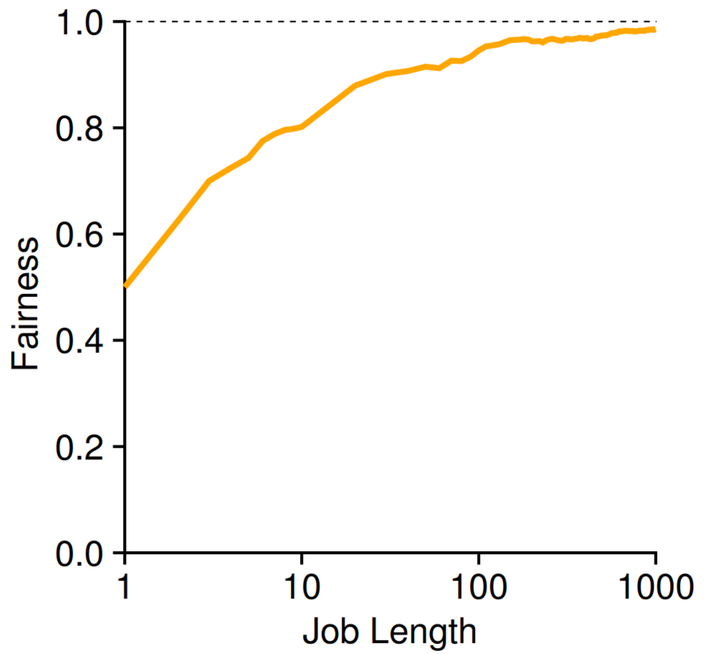
```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
//         get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets); 300

// current: use this to walk through the list of jobs
node_t *current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

- **It might generally be best to organize the list in a descending (sorted) order, from the highest number of tickets to the lowest**
  - The ordering does not affect the correctness of the algorithm

An Example

- **Two jobs competing against another, each with the same # of tickets (100) and same run time (R, which we will vary)**
  - To quantify this difference, we define a simple fairness metric, F that is the time the first job completes divided by the time that the second job completes
  - e.g.) 1st and 2nd jobs finish at 10 and 20, respectively, then F = 10/20=0.5
  - When both jobs finish at nearly the same time, F will be quite close to 1

- **The average fairness as the length of the two jobs (R) is varied from 1 to 1000 over thirty trials** → 정확히 1은 불가
  RR
  - When the job length is not 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



Stride Scheduling (1)

- **Randomness occasionally will not deliver the exact proportions especially over short time scales**
  - Stride scheduling, a deterministic fair-share scheduler, was invented
- **Each job has a stride, which is inverse to the number tickets** +ticket
  - e.g.) Jobs A, B, C with 100, 50, 250 tickets, respectively, and each stride is obtained by dividing some large number
  - For 10,000, the stride values of A, B, C are 100, 200, 40, respectively stride.
  - Every time a process runs, we will increase a counter for it (called its pass value) by its stride to track its global progress stride를 누적
  - Then, the scheduler uses the stride and pass to determine which process should run next

```
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
```

Stride Scheduling (2)

— 정확성 측면에서 ↑, 낮은 것 = 우선순위↑

- **The basic idea of the stride scheduling is simple**
  - At any given time, pick the process to run that has the lowest pass value so far
  - Stride scheduling gets them exactly right at the end of each scheduling cycle

stride 값은 가지고 있어야 함

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	…

100:50:250 tickets

2:1:5 runs

20번 D만 실행되고 들어와야한다 (독점!!)

D

stride 10

(200이 될때 까지 deterministic 하게 scheduling 할 수 있음.)

- **Given stride scheduling's precision why use lottery scheduling?**
  - Lottery scheduling has a nice property that stride one does not: no global state
  - What if new job enters in the middle of our stride schedule?
  - The new job with pass value 0 will result in the monopolization of the CPU
  - Lottery makes it much easier to incorporate new processes in sensible manner

그냥 뽑고 비교만 해주면 됨.

Linux Completely Fair Scheduling

scheduling 하는 데만...

- **In Google, scheduling uses 5% of overall datacenter CPU time**
  - Reducing the overhead as much as possible is key goal in scheduler design
- **The current Linux adopts completely fair scheduler (CFS) that implements fair-share scheduling for efficiency and scalability**
  - CFS aims to spend very little time making scheduling decisions by through both its inherent design and its clever use of data structures well-suited to the task

← CPU를 적게 쓴 애 부터! (virtual)

- **CFS uses a counting-based technique virtual runtime (vruntime)**
  - At each process runs, it accumulates vruntime proportional with physical time
  - CFS will pick the process with the lowest vruntime to next run → vruntime이 가장 작은 것은 실행시킴.
- **CFS uses sched_latency to determine how long one process should run before considering a switch**

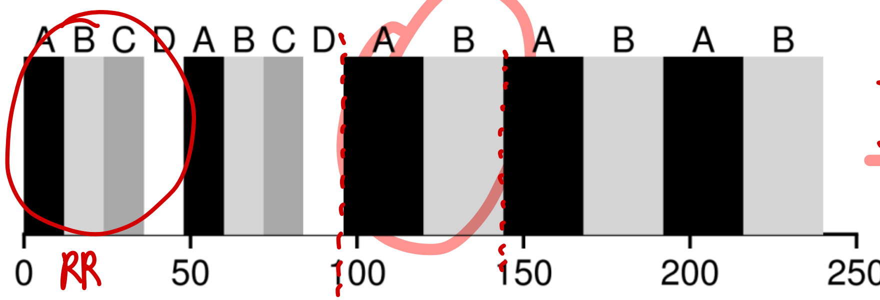
  $P_1 \xrightarrow{\text{maximum 시간}} P_2$

  - A typical sched_latency value is 48 ms (milliseconds)
  - CFS divides this value by the number of processes (n) running on the CPU to determine the time slice for a process

Basic Operation of CFS

CFS

- **Example of CFS**
  - 4 processes running (n=4), then, per-process time slice is 12 ms (48ms/4)
  - The four jobs A, B, C, D each run for two time slices and A, B complete
  - Then, the remaining two C, D each run for 24 ms (n=2) in round-robin fashion



프로세스의 개수 따라
time quantum을 바꿔야함.

1000개의 process
48/1000
↳ context-switching overhead↑

- **What if too many processes running?**
  - Result in too small of a time slice → too many context switching
  - To address this, CFS adds an additional parameter: min_granularity = 6 ms, which is the minimum time slice value
    - 프로세스가 많아도 6ms는 보장해주자.
  - e.g.) 10 processes → time slice = max(48/10, 6) = max(4.8, 6) = 6 ms

- **By doing this, in CFS, a time slice varies; non-fixed time slice**

Weighting (Niceness) (1)

우선 순위↑

- **CFS enables controls over process priority by process's nice level**
  - The nice parameter can be set anywhere from -20 to +19 with default of 0
  - Positive values imply lower priority and negative ones do higher priority
  - CFS maps the nice value of each process to a weight

하나의 방법,

```
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,
};
```

- **Priority is considered by computing time slice with the weights**

비율.

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

  - e.g.) Jobs A, B; A with nice value of -5 and B with nice value of 0 (default)

process	nice value	weight	time slice
A	-5	3121	3121/(3121+1024)*48 = 36 ms
B	0	1024	1024/(3121+1024)*48 = 12 ms

Weighting (Niceness) (2)

작은 수를 가중치!

A B
$\frac{1}{3}$ 1
x3

- **Calculating vruntime is also adapted in CFS**
  - The actual run-time that process i has accrued ($runtime_i$) is considered and scaled inversely by the weight of the process

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

2번 참조.

  - e.g.) Jobs A, B; A with nice value of -5 and B with nice value of 0 (default)

process	nice value	weight	accumulated value
A	-5	3121	1024/3121*runtime = 1/3*runtime
B	0	1024	1024/1024*runtime = 1 runtime

  - A's vruntime accumulates three times slower than B's

time slice를 작게 3배 해줘야 함.

- **When the scheduler has to find the next job to run, it should do so as quickly as possible**
  - Simple data structures like lists don't scale
  - Modern systems sometimes are comprised of 1000s processes
  - Searching through a long-list every so many milliseconds is wasteful

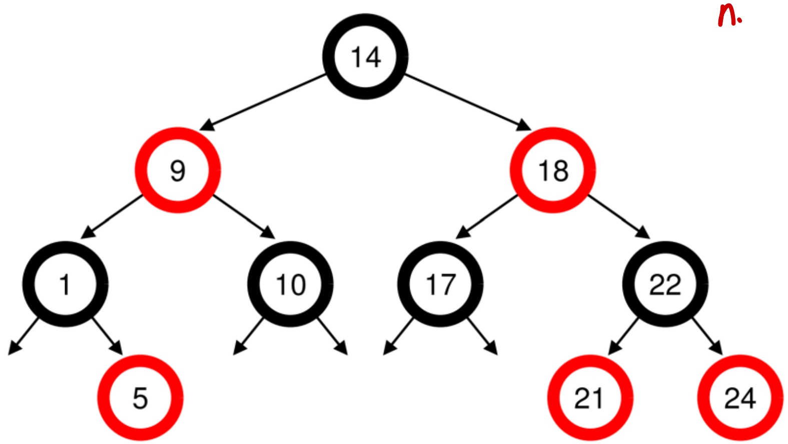
Scale

Using Red-Black Trees & Handling I/O, Sleeping

→ minimum vruntime을 O(log n)으로 찾기 위해

- **CFS keeps processes in a red-black tree, which is a balanced tree**
  - Balanced tree do a little extra work to maintain low depths, and thus ensure that operations are logarithmic in time O(log n)
  - CFS does not keep all processes in the tree; rather, only running (or runnable) processes are kept therein

n. log n. 2^{32} 4GB



CFS Red-Black Tree

Efficiently find the process with the minimum vruntime O(log n)

vruntime. 찾는걸 logN 로 할 수 있게 됨.

10초 쉬었어!

- **I/O and sleeping processes give rise some issues**
  - Two processes A, B where A is running and B is sleeping for I/O (e.g. 10 sec)
  - When B wakes up, B will monopolize the CPU for 10 sec, effectively starving A
  - CFS handles this by altering vruntime of a job when it wakes up, specifically by setting its vruntime to the minimum value found in the tree