Scheduling: Proportional Share

CMPU 334 – Operating Systems
Jason Waterman
Proportional Share Scheduler

• Fair-share scheduler
  • Guarantee that each job obtain a certain percentage of CPU time
  • Not optimized for turnaround or response time
Basic Concept

• Tickets
  • Represent the share of a resource that a process should receive
  • The percent of tickets represents its share of the system resource in question

• Example
  • There are two processes, A and B
    • Process A has 75 tickets → receive 75% of the CPU
    • Process B has 25 tickets → receive 25% of the CPU
Lottery scheduling

• The scheduler picks a winning ticket
  • Switch to the winning process and run it

• Example
  • There are 100 tickets
    • Process A has 75 tickets: 0 to 74
    • Process B has 25 tickets: 75 to 99

Scheduler’s winning tickets: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63

Resulting scheduler: A B A A B A A A A A B A B A

A runs 11/15 = 73.3%, B runs 4/15 = 26.7%

The longer these two jobs compete, the more likely they are to achieve the desired percentages
Ticket Mechanisms

• Ticket currency
  • A user allocates tickets among their own jobs in whatever currency they would like
  • The system converts the currency into the correct global value
• Example
  • There are 200 tickets (Global currency)
  • User A has 100 tickets (with two processes A1 and A2)
  • User B has 100 tickets (with one process B1)

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)
Ticket Mechanisms (Cont.)

• Ticket transfer
  • A process can temporarily hand off its tickets to another process

• Ticket inflation
  • A process can temporarily raise or lower the number of tickets it owns
  • If any one process needs more CPU time, it can boost its tickets
    • Convenient way to reflect this need without communicating with other processes
  • Assumes processes cooperate and are friendly with each other
Implementation

- Example: There are three processes, A, B, and C
  - Keep the processes in a list:

```
1 // counter: used to track if we've found the winner yet
2 int counter = 0;
3
4 // winner: use some call to a random number generator to
5 // get a value, between 0 and the total # of tickets
6 int winner = getrandom(0, totaltickets);
7
8 // current: use this to walk through the list of jobs
9 node_t *current = head;
10
11 // loop until the sum of ticket values is > the winner
12 while (current) {
13     counter = counter + current->tickets;
14     if (counter > winner)
15         break; // found the winner
16     current = current->next;
17 }
18 // 'current' is the winner: schedule it...
```
Fairness

- $F$: fairness metric
  - Assume 2 jobs, same number of tickets, same runtime
  - To be fair, we would like both jobs to finish at the same time
  - Metric is the time the first job completes divided by the time that the second job completes

- Example:
  - There are two jobs, each job has runtime 10
    - First job finishes at time 10
    - Second job finishes at time 20
  - $F = \frac{10}{20} = 0.5$
  - $F$ will be close to 1 when both jobs finish at nearly the same time
Lottery Fairness Study

- There are two jobs
  - Each job has the same number of tickets (100)

When the job length is not very long, average unfairness can be quite severe.
Lottery Discussion

• Simplicity of implementation
  • Random number generator
  • List of processes
  • Total number of tickets

• How do you assign tickets?
  • Tough problem
  • System behavior depends on how tickets are allocated

• Let the users decide how to allocate tickets?
  • It remains an open question
Stride Scheduling

• Random is easy to implement, but it may not deliver exactly the right proportions

• Stride of each process
  • (A large number) / (the number of tickets of the process)
  • Example: A large number = 10,000
    • Process A has 100 tickets → stride of A is 100
    • Process B has 50 tickets → stride of B is 200

• When a process runs, increment its counter (pass value) by its stride
  • Pick the process to run that has the lowest pass value

```plaintext
current = remove_min(queue);              // pick client with minimum pass
schedule(current);                        // use resource for quantum
current->pass += current->stride;         // compute next pass using stride
insert(queue, current);                   // put back into the queue
```

A pseudo code implementation
Stride Scheduling Example

<table>
<thead>
<tr>
<th>Pass(A) (stride=100)</th>
<th>Pass(B) (stride=200)</th>
<th>Pass(C) (stride=40)</th>
<th>Who Runs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>40</td>
<td>C</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>80</td>
<td>C</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>120</td>
<td>A</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>120</td>
<td>C</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>160</td>
<td>C</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>200</td>
<td>...</td>
</tr>
</tbody>
</table>

If new job enters with pass value 0, it will **monopolize** the CPU!
The Linux Completely Fair Scheduler (CFS)

- Implements fair-share scheduling

- Efficient and scalable
  - Quickly make a scheduling decision

- Scheduling performance is important
  - Scheduling uses about 5% of datacenter CPU time at Google
CFS Basic Operation

• Fairly divides a CPU evenly among all competing (runnable) processes
  • Doesn’t use a fixed time slice
• Uses the **virtual runtime** \((v\text{runtime})\) of a process
  • Accumulates as the process runs
  • To schedule a process, pick the one with the lowest \(v\text{runtime}\)
• When to schedule?
  • Frequent switches increase fairness but has a higher overhead
  • Fewer switches give better performance at the cost of fairness
• Controlled by **sched_latency** parameter
  • Maximum time a process can run before considering a switch (e.g., 48 ms)
  • This is divided by the number of runnable processes to get a process time slice
  • CFS will be completely fair over this time period
CFS Example

- `sched_latency` = 48 ms

- Four processes that are runnable to start
  - Per process time slice of 12 ms (48/4)
  - `vruntime` is starts at 0 for these jobs

- Pick job with the lowest `vruntime` (A, B, C, or D in this case)

- Run job A until it has used 12 ms of `vruntime`
  - Then make a scheduling decision
  - Run the job with the lowest `vruntime`
    - (B, C, or D)

- C and D complete after 96 ms
  - Time slice is adjusted to 24 ms (48/2)
Too many processes runnable?

- **Per process time slice is the `sched_latency / runnable processes`**
  - A lot of runnable processes could lead to small time slices
    - Lots of context switches and more overhead
- **CFS `min_granularity` parameter**
  - Minimum time slice of a process (e.g., 6 ms)
  - CFS will never set the time slice of a process to less than this value
  - In this case, may not be perfectly fair over the target scheduling latency
    - E.g., `sched_latency` = 48 ms with 10 runnable processes
    - time slice 4.8 --> 6 ms
    - all jobs won’t run during the 48 ms
- **Timer interrupts**
  - Time slices are variable, how to set the timer?
  - Timer goes off frequently (e.g., 1 ms)
  - Gives the CFS scheduler a chance to see if the current job has reached the end of its run
Niceness Levels

- Gives the user control over process priority
  - Give some processes a higher (or lower) share of the CPU
- Not through tickets, but with a **nice** level of a process
  - A measure of how nice (to other processes) your job is
  - 19 (lowest priority)
  - -20 (highest priority)
- Nice levels are mapped to a weight used to compute an effective time slice for a process
Niceness Weightings

```c
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,
};
```

\[
\text{time\_slice}_k = \frac{\text{weight}_k}{\sum_{n=0}^{n-1} \text{weight}_i} \cdot \text{sched\_latency}
\]
Niceness Weighting Example

• Two processes, A and B
  • A’s niceness level is -5 (boost in priority)
  • B’s niceness level is 0 (default)

• Calculate the time slice for A and B
  • Weight A: 3121, weight B: 1024, total weight: 4145
  • Time slice A: $\frac{3121}{4145} = 0.753 \times \text{sched\_latency}$
  • Time slice B: $\frac{1024}{4145} = 0.247 \times \text{sched\_latency}$

• Assuming a 48 ms sched\_latency:
  • Process A gets about 75% of the sched\_latency (36 ms)
  • B gets about 25% of the sched\_latency (12 ms)

• Weight table is constructed to preserve CPU proportionally ratios when the difference in nice values is constant
  • E.g., if process A had a nice value of 5 and B had a nice value of 10, they would be scheduled for the same time slices as above $A = \frac{335}{445} = 0.753$
Calculating vruntime

• Higher priority processes get a longer time slice
• But we pick the process with the lowest vruntime to run next
  • To handle priority properly, vruntime must scale inversely with priority
• For our example:
  • A’s vruntime will accumulate at about a 1/3 the rate of B’s

\[ vruntime_i = vruntime_i + \frac{weight_0}{weight_i} \cdot runtime_i \]
CFS efficiency

• How quickly can the scheduler find the next job to run
  • Lists don’t scale if you have 1000s of processes to search through every millisecond

• CFS keeps processes in a red-black tree
  • A type of balanced tree
  • Does a little extra work to maintain low depths
  • $O(\log n)$ for operations (search, insert, delete)

• CFS only keeps running and runnable processes in this structure
  • If a process is waiting on I/O, it is removed from the tree and kept track elsewhere
  • What to do when process wakes up?
    • vruntime will be behind the others and could monopolize the CPU
    • CFS sets the vruntime for the job to the minimum value found in the tree
    • Jobs that sleep for short periods often do not ever get their fair share of the CPU
Summary

• We looked at three proportional share schedulers
• Lottery scheduling
  • Uses randomness to achieve proportional share
  • Can be unfair with short running jobs
  • Can be implemented with no shared state between processes
  • Ticket allocation can be difficult
• Stride scheduling
  • Achieves proportional share deterministically
• Linux Completely Fair Scheduler (CFS)
  • Most widely used fair-share scheduler in existence
  • A bit like a weighted round-robin with dynamic time slices
  • Built to scale and perform well under load