9: Scheduling: Proportional Share

Operating System: Three Easy Pieces

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.
  - Load the state of that *winning process* and runs it.
- Example
  - There are 100 tickets
    - Process A has 75 tickets: 0 ~ 74
    - Process B has 25 tickets: 75 ~ 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	A	B	A	B	A

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

The beauty of randomness (in scheduling)

- Deals easily with corner-case situations
  - Others should have a lot of “ifs” to prevent them
- Little state required
  - No need to track the details of each process in the past
- Really fast
  - More speed → more pseudo-randomness (or HW assistance)

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)
    - Process A has 100 tickets
    - Process B has 100 tickets

User A → *500* (A's currency) to A1 → *50* (global currency)
→ *500* (A's currency) to A2 → *50* (global currency)

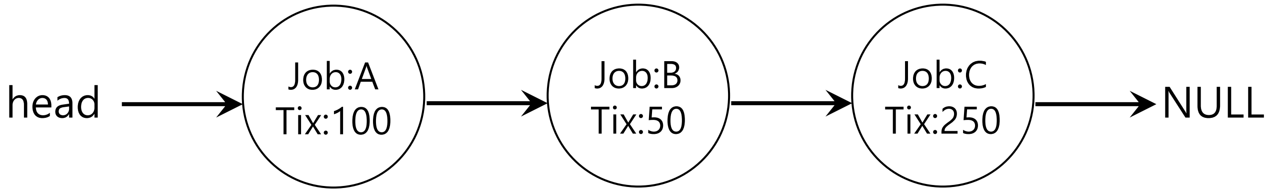
User B → *10* (B's currency) to B1 → *100* (global currency)

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 is owns.
  - If any one process needs *more CPU time*, it can boost its tickets.

Implementation

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



```
// 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);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
        counter = counter + current->tickets;
        if (counter > winner)
                break; // found the winner
        current = current->next;
}
// 'current' is the winner: schedule it...
```

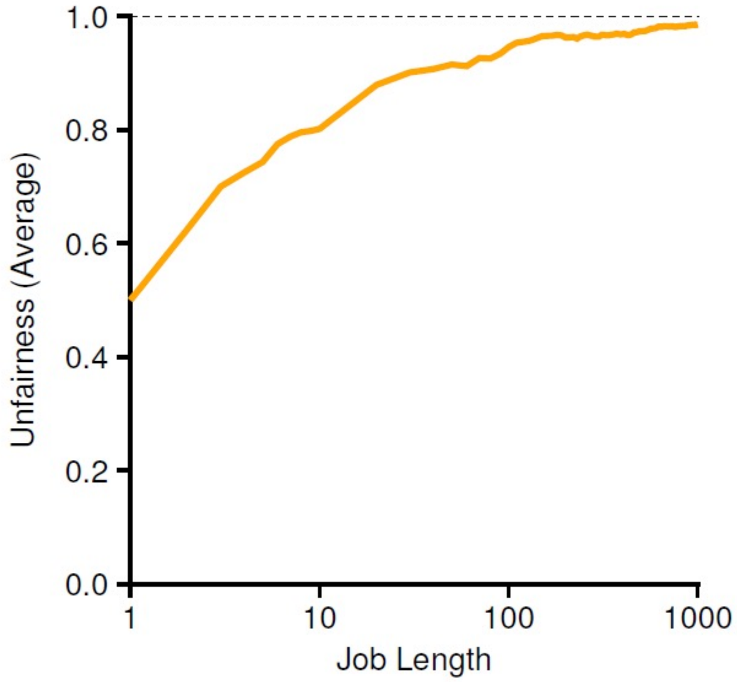
https://stackoverflow.com/questions/2509679/how-to-generate-a-random-integer-number-from-within-a-range

Implementation (Cont.)

- U: unfairness metric
  - The time the first job completes divided by the time that the second job completes.
- Example:
  - There are two jobs, each jobs has runtime 10.
    - First job finishes at time 10
    - Second job finishes at time 20
  - $\mathrm{U} = \frac{10}{20} = 0.5$
  - U will be close to 1 when both jobs finish at nearly the same time.

Lottery Fairness Study

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



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

Stride Scheduling (deterministic Fair-share scheduler)

- Stride of each process
  - Defined as (one large number) / (the number of tickets of the process)
  - Example: one large number = 10,000
    - Process A has 100 tickets → stride of A is 100
    - Process B has 50 tickets → stride of B is 200
    - Process C has 250 tickets → stride of C is 40

- A process runs, increment a counter(=pass value) for it by its stride.
  - Pick the process to run that has the lowest pass value

```
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

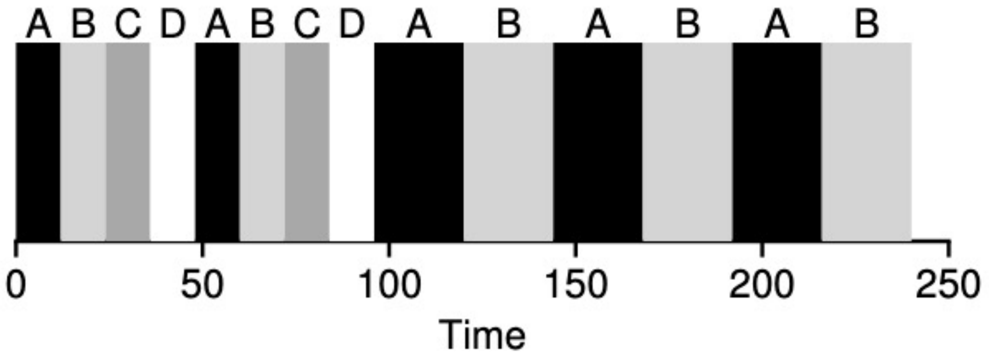
Pass(A) (stride=100)	**Pass(B)** (stride=200)	**Pass(C)** (stride=40)	**Who Runs?**
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	…

If new job enters with pass value 0,
It will monopolize the CPU!:
Stride scheduler requires global state (in contrast with lottery)

Linux Completely Fair Scheduler (CFS)

- 5% overall datacenter CPU time wasted in scheduler

- Focus on fairness and minimizing scheduling overhead
  - Keep track of the (virtual) runtime of each process
  - At scheduling time, **chooses** the process with the **lowest virtual run time**
  - Time slice is variable: according number of ready to run processes (from **`sched_latency`** (48 ms) for 1 process to **`min_granularity`** (6 ms) for any number of processes



Niceness and Weights

- Time slice and virtual runtime of the process can be affected by weights (niceness)

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

- Typically, from -20 to 19
- By default, 0

```
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,
};
```

Example

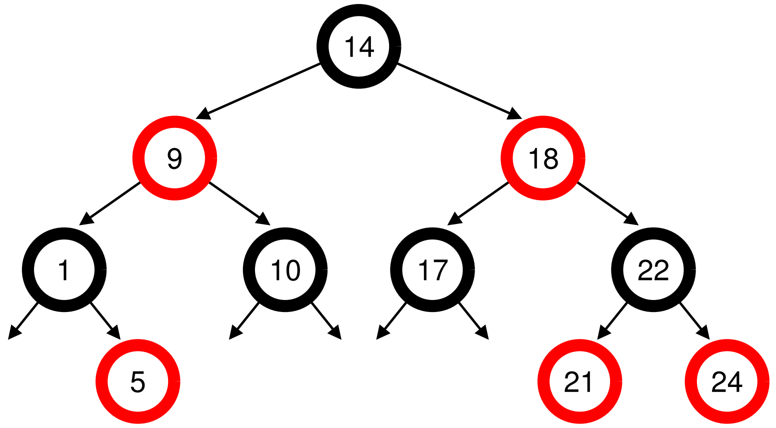
- Two process: A (niceness -5) B (niceness 0)
- `weight`$_A$=3121, `weightB`=1024
- `time_sliceA`=3/4 (36 ms), `time_sliceB`=1/4 (12 ms) of `sched_latency (48 ms)`
- Also, virtual runtime changes with weights

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

- A accumulates virtual runtime at 1/3 of B

Linux Completely Fair Scheduler (CFS)

- Minimal scheduler overhead
  - Data structures should be scalable: No lists
  - CFS uses a balanced tree (red-black tree) of the ready-to-run processes
  - O(log n) insertions and searches



- Many more details
  - I/O is handled altering the `vruntime` of the awaken process to the minimum value in the tree
  - Heuristics for multi-CPU scheduling (cache affinity, frequency, core complexity…)
  - Cooperative multi-process schedule
  - …

This lecture slide set is used in AOS course at University of Cantabria. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)