Operating Systems

KAIST

9: Scheduling: Proportional Share

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.

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)

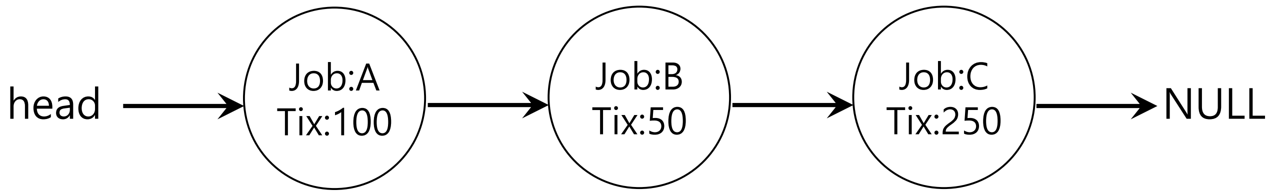
User B → *10* (B's currency) to B1 → *100* (global currency)

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

Implementation

- Example: There are there processes, A, B, and C.
  - Keep the processes in a list sorted with the ticket size: highest ticket first



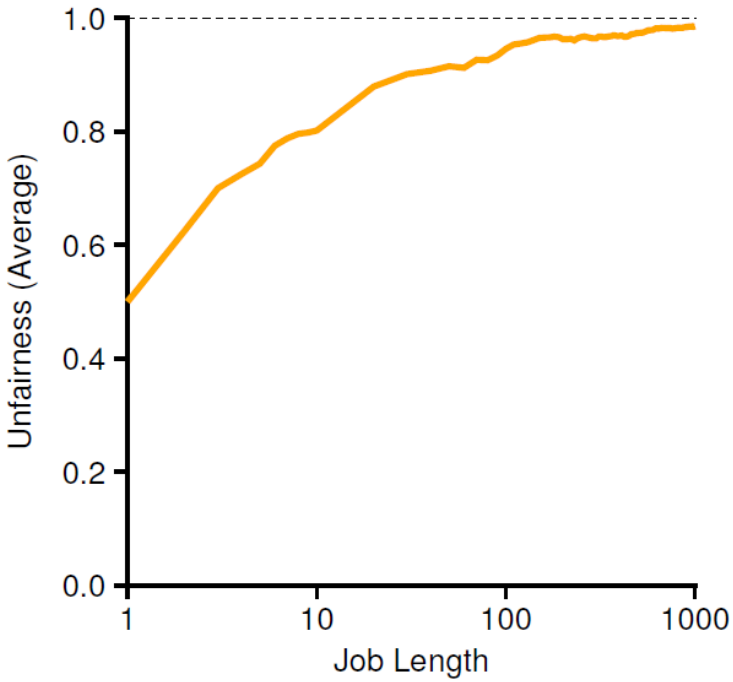

```c
// counter: used to track if we've found the winner yet
int counter = 0;

// 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...
```

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.

Deterministic Approach: Stride Scheduling

- 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

- 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	…

Stride scheduling needs to maintain the per process pass value.
If new job enters with pass value 0 it will monopolize the CPU!

Advantage of Lottery scheduling: no per-process state

The Linux Completely Fair Scheduling (CFS)

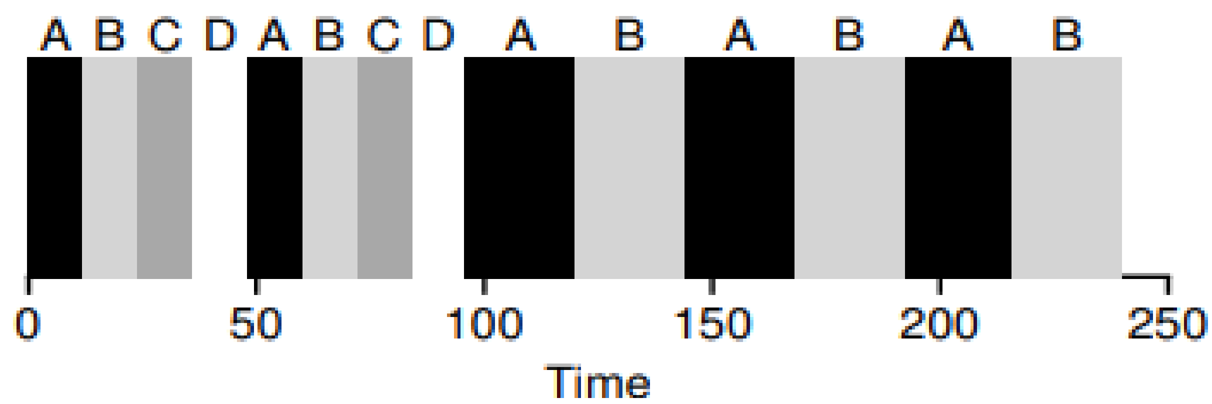
- Completely Fair Scheduling (CFS)
  - The current CPU scheduler in Linux
  - Non-fixed timeslice.
    - CFS assigns process`s timeslice a proportion of the processor.
  - Priority
    - Enables control over priority by using nice value.
  - Efficient data structure.
    - Use red-black tree for efficient search, insertion and deletion of a process.

Basic

- Virtual runtime (vruntime)
  - Denote how long the process has been executing.
  - Per-process variable
  - Increase in **proportion with physical (real) time** when it runs.
  - CFS will pick the process with the **lowest vruntime** to run next.
- sched_latency
  - A typical value is 48 (milliseconds)
  - process`s timeslice = sched_latency / (the number of process)

Example

- Simple Example
  - 4 processes (A,B,C,D) and then 2 processes(C,D) complete.



- min_granularity
  - The minimum timeslice (6ms)
  - Ensure that not too much time is spent in scheduling overhead, When there are too many processes running.

Weight

- Nice value
  - CFS enables controls over process priority.
  - Nice parameter is integer value and can be set from -20 to +19.
  - The nice value is mapped to a weight (value is not important)

```
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,
};
```

Weighting (Niceness)

- New timeslice formula

$$time\_slice_k = \frac{weight_k}{\sum_{n=0}^{n-1} weight_i} \cdot sched\_latency$$

- Simple Example
  - Assign Process `A` a nice value of -5 and process `B` a nice value of 0.

Process	nice value	weight	Time slice
A	-5	3121	36 ms
B	0	1024	12 ms

vruntime with weight

- Weighting (Niceness)
  - vruntime formula
    - Calculate the actual run time. Scales it inversely by the weight of process.

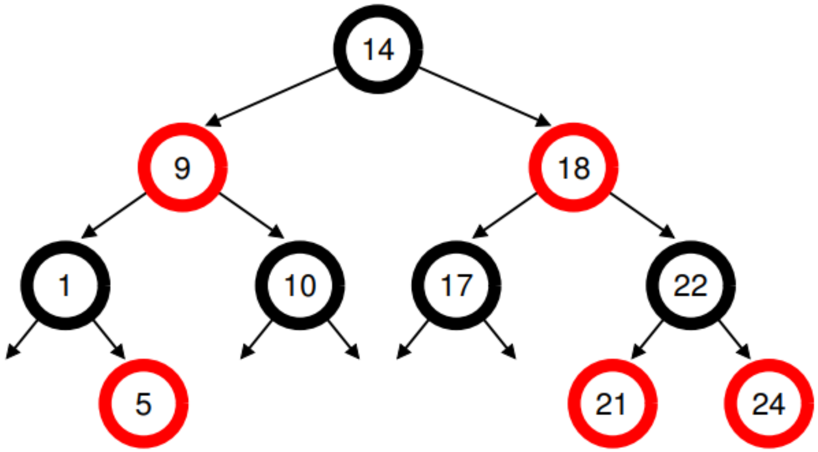
$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} \cdot runtime_i$$

  - Simple Example

Process	nice value	weight	Accumulated value
A	-5	3121	1 * runtime
B	0	1024	3 * runtime

Structure of ready queue

- Red-Black Tree
  - Balanced binary tree (can address worst-case insertion)
  - Ordering of Red-Black Tree : O(log n)
  - Efficiently find the process with minimum virtual runtime.
  - Only running (or runnable) processes are kept therein.



IO and sleeping process

- Dealing with I/O and Sleeping processes
  - Avoid the situation where some process monopolizes the CPU, if process have significantly small vruntime after sleeping.
  - Set the vruntime of process to the minimum value found in tree when it wakes up.
  - Process that sleep for short periods of time frequently do not ever get their fair share of the CPU.