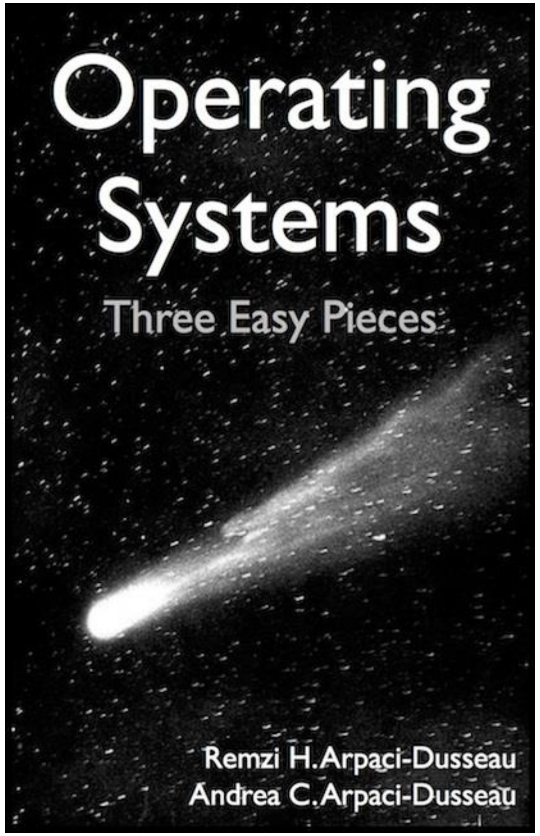
CMSC 125: Operating Systems

- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org

Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set 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 at University of Wisconsin.

9: Scheduling: Proportional Share

Operating System: Three Easy Pieces

Proportional Share Scheduler

- aka Fair-share scheduler
  - Guarantee that each process obtains *a certain percentage* of CPU time
  - Not necessarily optimizing for turnaround time or response time

Basic Concept: Hold a Lottery!

- Tickets
  - Represent the share of a resource that a process(or user) should receive
  - <u>The percent of tickets</u> 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

Approach #1:Lottery Scheduling

- Lottery can be held *every time slice*
- The scheduler picks a winning ticket (probabilistically/randomized)
  - Run the process that holds the *winning ticket*
- 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.

Aside: Why Random?

- Random <u>avoids strange corner-case behaviors</u>
- Random is <u>lightweight</u>
  - Per-process accounting is reduced
- Random is <u>fast</u>
  - Generating a random number from a generator is typically fast (hardware-supported)

Ticket Mechanisms

- **#1: Ticket Currency**
  - A user(in multiuser systems) allocates tickets among their own processes in whatever currency they would like(Local Currency)
  - The system converts the Local Currency into the correct Global Currency value
  - Example:
    - There are 200 tickets (Global Currency)
      - Process A has 100 tickets
      - Process B has 100 tickets

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

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

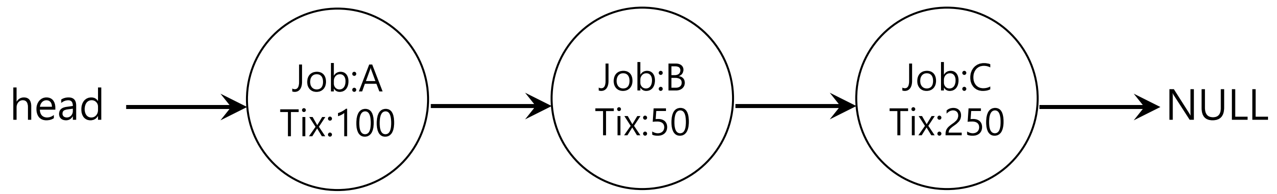
Ticket Mechanisms (Cont.)

- **#2:Ticket Transfer**
  - A process can temporarily hand off *its tickets* to another process
  - Useful in client-server applications with server doing tasks on behalf client

- **#3: 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
  - Assumes that a group of processes trust each other to prevent abuse

Implementation

- Example: There are three processes, A, B, and C. There are 400 tickets in total. Current draw returns winner=300.
  - Keep the processes in a list:




```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...
```

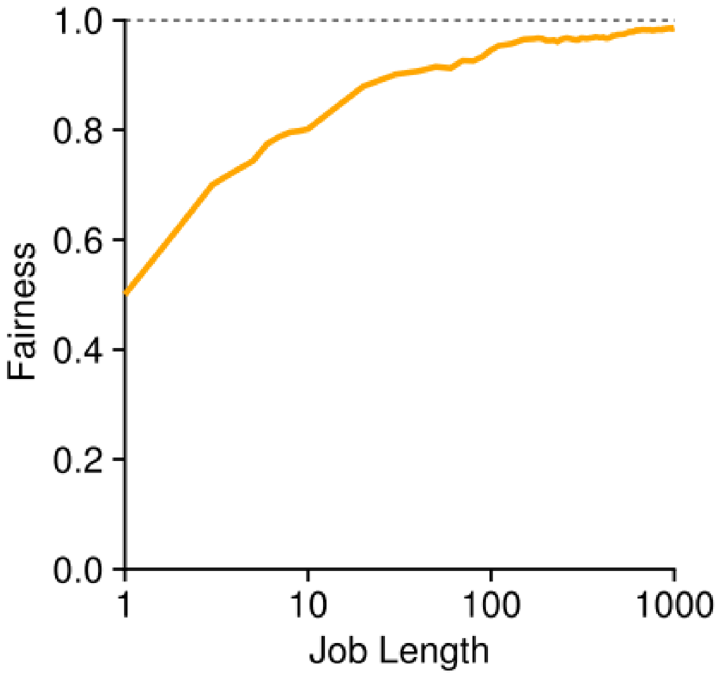
What optimizations can you think of?

Implementation (Cont.)

- How to evaluate?
- Define fairness metric `F`
  - The time the <u>first process completes</u> divided by the time that the <u>second process completes</u>
- Example:
  - There are two processes, each process has a run time `R`=10
    - First process finishes at time 10
    - Second process 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

- Simulation Scenario:
  - Two processes with run time(job length) $\mathbb{R}$ ranging from 1 to 1000. Over thirty(30) trials
  - Each process has the same number of tickets (100)



When the runtime is not very long,
average fairness can be quite severe.

How to assign tickets?

- One approach is to assume that users know best, <u>thus let users assign tickets</u>
- Still an open problem though

Approach #2: Stride Scheduling

- Randomness occasionally will not deliver good results especially for processes with short run times
- A deterministic fair-share scheduler invented by Waldspurger
- Stride of each process -
  - (Some large number) / (the number of tickets of the process)
  - Example(in the previous discussion): Some large number say 10000
    - Process A has 100 tickets → stride of A is (10000/100)=100
    - Process B has 50 tickets → stride of B is (10000/50)=200
- When a process runs, increment a `counter`(aka **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
```

Pseudocode for Stride Scheduling

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	…

Arbitrarily chosen at the start since all Pass are 0 initially

If new process enters with Pass value 0, it will monopolize the CPU!

Lottery is still better since there is no global state to be maintained per process.

Approach #3: Linux Completely Fair Scheduler (CFS)

- Goal: Fairly divide a CPU evenly among all competing processes
- Uses a counting-based technique called **virtual runtime (vruntime)**
- As a process runs, its vruntime increases (*may increase at the same rate as physical time*)
- When scheduling decision time occurs, pick the process with the *lowest vruntime*
- Uses a periodic timer interrupt – can only make decisions at fixed time intervals
  - Not affected if time slice is not a multiple of the interrupt timer interval time

Linux Completely Fair Scheduler (CFS) (Cont.)

- Control parameters:
  - **sched_latency** – determines how long one process should run before considering a switch (typical values is 48ms)
    - time slice for a process = (sched_latency / n processes)
    - Example: Given n=4, time slice = 48/4 = 12ms
  - **min_granularity** – to address too many processes running (when n is very large, time slice becomes small)
    - Minimum time slice for a process despite a large n, typically set to 6ms

Linux Completely Fair Scheduler (CFS) (Cont.)

- Weighting (**Nice**ness) – allows users/admins to control process priority (`nice` and `renice` programs)
  - Nice values range: [-20, +19], <u>(+) means lower priority</u>, <u>(-) means higher priority</u>
  - Nice values maps to weights in table

```
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,
};
```

  - Effective Time Slice

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

Linux Completely Fair Scheduler (CFS) (Cont.)

- vruntime adjustment

Actual run time accrued over time

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

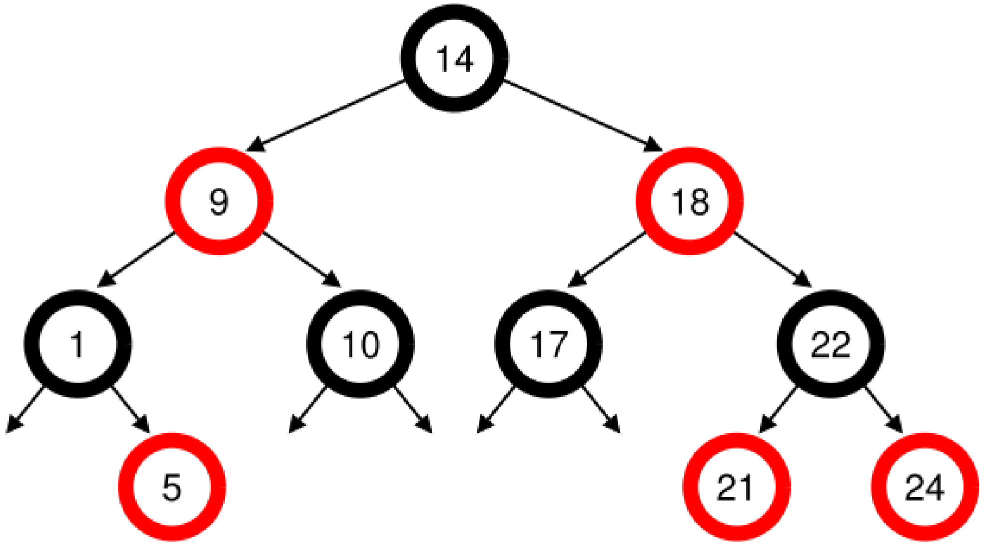
- Example:
  - Scenario: Process A with nice value of -5, Process B with nice value of 0
  - Time Slice
    - From table weight(A) = 3121, Time_Slice(A) = (3121/4145)*48 = **36ms**
    - From table weight(B) = 1024, Time_Slice(B) = (1024/4145)*48 = **12ms**
  - vruntime
    - vruntime(A) = vruntime(A) + (1024/3121)*run_time(A)
    - vruntime(B) = vruntime(A) + (1024/1024)*run_time(B)
    - Thus, vruntime(A) will accumulate at .33 the rate of vruntime(B) – **A will have smaller vruntime and will have higher priority**

Linux Completely Fair Scheduler (CFS) (Cont.)

- How to <u>efficiently</u> (as quickly as possible) find the next process to run?
- Red-Black Trees
  - Height-balanced tree – logarithmic search time compared to linear search time for lists
  - CFS places only <u>running/runnable processes</u> on the RB tree

Linux Completely Fair Scheduler (CFS) (Cont.)

- RB Tree Example
  - Given: There are 10 processes with vruntimes of 1, 5, 9, 10, 14, 18, 17, 21, 22, 24
  - If a sorted list is used, next process to run is simply the first element
    - Inserting a new process however will need the scan of all items in the sorted list, in the worst case
  - Use of RB tree(vruntimes as key) improves efficiency, most operations are logarithmic



Linux Completely Fair Scheduler (CFS) (Cont.)

- How to deal with I/O and sleeping processes (long sleep)?
  - Awaken sleeping processes may monopolize CPU since its vruntime has not been updated for a while,thus small
  - Solution: Alter the vruntime of the newly awaken process by setting it to the minimum vruntime in the RB tree
  - Processes sleeping for short periods of time will not get fair share of CPU
- Other fun stuff about CFS
  - Heuristics to improve performance
  - Handling multiple CPUs
  - Scheduling for process groups

Linux Commands

- `man sched`
- `sysctl -A | grep "sched" | grep –v "domain"`
- `sudo nice –-10 ./cpu.elf`
- ``ps -l -p `pidof cpu.elf` ``
- `cat /proc/sched_debug`
- `cat /proc/schedstat`
- ``cat /proc/`pidof cpu.elf`/sched``
- ``sudo renice -n -20 -p `pidof cpu.elf` ``
- ``sudo chrt -r -p 40 `pidof cpu.elf` ``
- ``sudo chrt -o -p 0 `pidof cpu.elf` ``