TCSS 422: OPERATING SYSTEMS

Three Easy Pieces:
April 16, 2018:
Proportional Share Scheduler

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

- Proportional Share Scheduler – Ch. 9

APRIL 16, 2018 - PROPORTIONAL SHARE SCHEDULER

PROPORTIONAL SHARE SCHEDULERS

- Preallocate a certain amount of CPU time to each process
- Each job has a scheduling weight
  - Similar to scheduling priority
- Jobs receive a share of the available CPU resources relative to this "job scheduling weight"
- Users assign or influence the assignment of job scheduling weight
- Weight does not DIRECTLY map to CPU time

LOTTERY SCHEDULER

- Also called fair-share scheduler
  - Guarantees each job receives some percentage of CPU time based on share of "tickets"
  - Each job receives an allotment of tickets
  - % of tickets corresponds to potential share of a resource
  - Can conceptually schedule any resource this way
    - CPU, disk I/O, memory

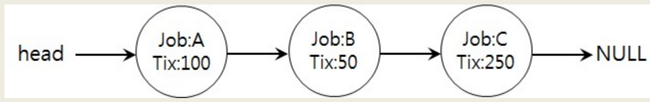
April 16, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L6a.5

LOTTERY SCHEDULER

- Simple implementation
  - Just need a random number generator
    - Picks the winning ticket
  - Maintain a data structure of jobs and tickets (list)
  - Traverse list to find the owner of the ticket
  - Consider sorting the list for speed

April 16, 2018	TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma	L6a.6

LOTTERY SCHEDULER IMPLEMENTATION




```
// 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...
```


TICKET MECHANISMS

- Ticket currency / exchange
  - User allocates tickets in any desired way
  - OS converts user currency into global currency
- Example:
  - There are 200 global tickets assigned by the OS

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 - 2

- Ticket transfer
  - Temporarily hand off tickets to another process
- Ticket inflation
  - Process can temporarily raise or lower the number of tickets it owns
  - If a process needs more CPU time, it can boost tickets.

LOTTERY SCHEDULING

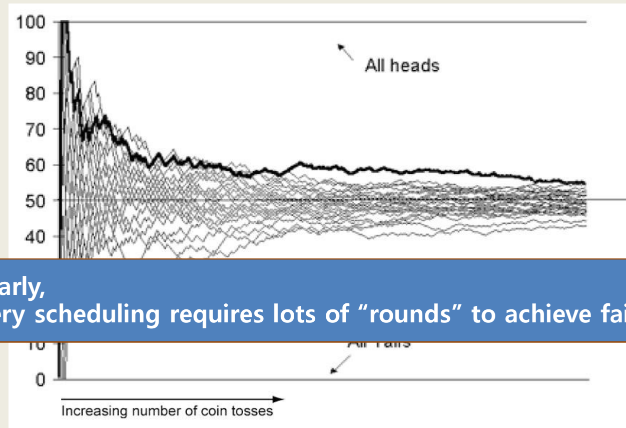
- Scheduler picks a winning ticket
  - Load the job with the winning ticket and run it
- Example:
  - Given 100 tickets in the pool
    - Job A has 75 tickets: 0 - 74
    - Job B has 25 tickets: 75 – 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Scheduled job:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

- But what do we know about probability of a coin flip?

COIN FLIPPING

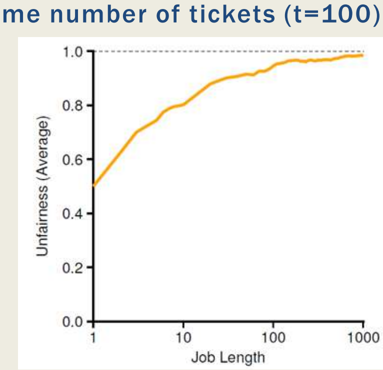
- Equality of distribution (fairness) requires a lot of flips!



Similarly,
Lottery scheduling requires lots of "rounds" to achieve fairness.

LOTTERY FAIRNESS

- With two jobs
  - Each with the same number of tickets (t=100)



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

LOTTERY SCHEDULING CHALLENGES

- What is the best approach to assign tickets to jobs?
  - Typical approach is to assume users know best
  - Users are provided with tickets, which they allocate as desired
- How should the OS automatically distribute tickets upon job arrival?
  - What do we know about incoming jobs a priori ?
  - Ticket assignment is really an open problem…

STRIDE SCHEDULER

- Addresses statistical probability issues with lottery scheduling
- Instead of guessing a random number to select a job, simply count…

STRIDE SCHEDULER - 2

- Jobs have a "stride" value
  - A stride value describes the counter pace when the job should give up the CPU
  - Stride value is <u>inverse in proportion</u> to the job's number of tickets (more tickets = smaller stride)
- Total system tickets = 10,000
  - Job A has 100 tickets → A_{stride} = 10000/100 = 100 stride
  - Job B has 50 tickets → B_{stride} = 10000/50 = 200 stride
  - Job C has 250 tickets → C_{stride} = 10000/250 = 40 stride
- Stride scheduler tracks "pass" values for each job (A, B, C)

STRIDE SCHEDULER - 3

- Basic algorithm:
  1. Stride scheduler picks job with the <u>lowest</u> pass value
  2. Scheduler increments job's pass value by its stride and starts running the job for the current time slice
  3. Stride scheduler increments a system counter
  4. After scheduling quantum, scheduler returns to #1

STRIDE SCHEDULER - 4

- Stride scheduler always runs job(s) with the lowest pass value(s)
- KEY: Jobs having low "PASS" values are scheduled more often because their pass values increase more slowly than others…

STRIDE SCHEDULER - EXAMPLE

- Stride values
  - Tickets = priority to select job
  - Stride is inverse to tickets
  - Lower stride = more chances to run (higher priority)

Priority

C stride = 40

A stride = 100

B stride = 200

STRIDE SCHEDULER EXAMPLE - 2

- Three-way tie: randomly pick job A (all pass values=0)
- Set A's pass value to A's stride = 100
- Increment sys counter by A's stride. counter → 100
- Pick a new job: two-way tie

Tickets
C = 250
A = 100
B = 50

Pass Values

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	…

← Initial job selection is random. All @ 0

← C has the most tickets and receives a lot of opportunities to run…

STRIDE SCHEDULER EXAMPLE - 3

- We set A's counter (pass value) to A's stride = 100
- Next scheduling decision between B (pass=0) and C (pass=0)
  - Randomly choose B
- C has the lowest counter for next 3 rounds

Tickets
C = 250
A = 100
B = 50

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	…

← C has the most tickets and is selected to run more often …

STRIDE SCHEDULER EXAMPLE - 4

- Job counters support determining which job to run next
- Over time jobs are scheduled to run based on their priority represented as their share of tickets…
- Tickets are analogous to job priority

Tickets
C = 250
A = 100
B = 50

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	…

LINUX: COMPLETELY FAIR SCHEDULER (CFS)

- Loosely based on the stride scheduler
- CFS models system as a Perfect Multi-Tasking System
  - In perfect system every process of the same priority receives exactly $1/n^{th}$ of the CPU time
- Scheduling classes (runqueues)
  - Each has specific priority: default, real-time
  - Scheduler picks highest priority task in highest scheduling class
  - Time quantum based on proportion of CPU time (%), not fixed time allotments
  - Quantum calculated using NICE value

COMPLETELY FAIR SCHEDULER - 2

- Time slice: Linux ***"Nice value"***
  - Nice value predates the CFS scheduler
  - Top shows nice values
  - Process command (nice & priority):
    `ps ax -o pid,ni,cmd,%cpu, pri`
- Nice Values: from -20 to 19
  - Lower is ***higher*** priority, default is 0
  - Scheduling quantum is calculated using nice value
  - Target latency:
    - Interval during which task should run at least once
    - Automatically increases as number of jobs increases

COMPLETELY FAIR SCHEDULER - 3

- Challenge:
  - How do we map a nice value to an actual CPU timeslice (ms)?
  - What is the best mapping?
    - O(1) scheduler (< 2.6.23)
      - tried to map nice value to timeslice (fixed allotment)
    - Linux completely fair scheduler
      - maps nice value based on time proportion
      - with fewer jobs in a runqueue, the time proportion is larger

COMPLETELY FAIR SCHEDULER - 4

- Nice values become relative for determining time slices
  - Proportion of CPU time to allocate is relative to other queued tasks
- Scheduler tracks virtual run time in vruntime variable
- The task on a given runqueue (nice value) with the lowest `vruntime` is scheduled next
- `struct sched_entity` contains `vruntime` parameter
  - Describes process execution time in nanoseconds
  - Perfect scheduler → achieve equal `vruntime` for all processes of same priority

COMPLETELY FAIR SCHEDULER - 5

- CFS uses weighted fair queueing
- Runqueues are stored using a linux rbtree
  - Self balancing binary search tree
  - The leftmost node will have the lowest `vruntime`
  - Walking the tree to find the left most node is only O(log N) for N nodes
  - If tree is balanced, left most node can be cached
- Key takeaway
  identifying the next job to schedule is *really* fast!

QUESTIONS