TCSS 422: OPERATING SYSTEMS

**Scheduling:
Multi-level Feedback Queue,
Proportional Share**

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

OBJECTIVES

- Multi-level Feedback Queue
- Proportional Share Scheduler

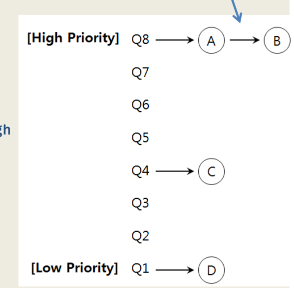
MULTI-LEVEL FEEDBACK QUEUE

- Objectives:
  - Improve turnaround time:
    Run shorter jobs first
  - Minimize response time:
    Important for interactive jobs (UI)
- Achieve without a priori knowledge of job length

MLFQ - 2

Round-Robin within a Queue

- Multiple job queues
- Adjust job priority based on observed behavior
- Interactive Jobs
  - Frequent I/O → keep priority high
  - Interactive jobs require fast response time (GUI/UI)
- Batch Jobs
  - Require long periods of CPU utilization
  - Keep priority low



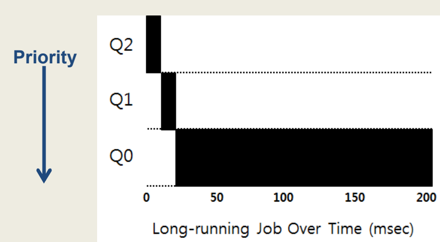
MLFQ: DETERMINING JOB PRIORITY

- New arriving jobs are placed into highest priority queue
- If a job uses its entire time slice, priority is reduced
  - Jobs appears CPU-bound ("batch" job), not interactive (GUI/UI)
- If a job relinquishes the CPU for I/O priority stays the same

MLFQ approximates SJF

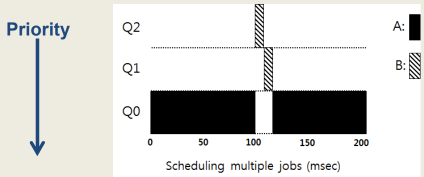
MLFQ: LONG RUNNING JOB

- Three-queue scheduler, time slice=10ms



MLFQ: BATCH AND INTERACTIVE JOBS

- A_{run_time}=200ms, B_{run_time} = 20ms
- $B_{arrival\_time}$ = 100ms

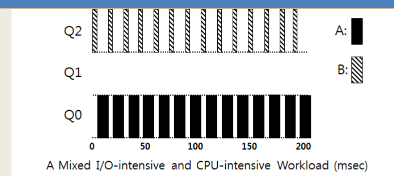


October 7, 2016 | TCSS422: Operating Systems [Fall 2016] Institute of Technology, University of Washington - Tacoma | L5.7

MLFQ: BATCH AND INTERACTIVE - 2

- Continuous interactive job with a long running batch job
  - Low response time is good for B
  - A continues to make progress

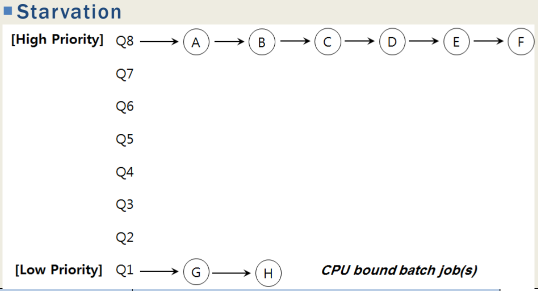
The MLFQ approach keeps interactive job(s) at the highest priority



October 7, 2016 | TCSS422: Operating Systems [Fall 2016] Institute of Technology, University of Washington - Tacoma | L5.8

MLFQ: ISSUES

- Starvation

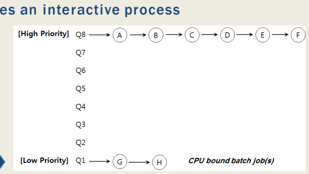


October 7, 2016 | TCSS422: Operating Systems [Fall 2016] Institute of Technology, University of Washington - Tacoma | L5.9

MLFQ: ISSUES - 2

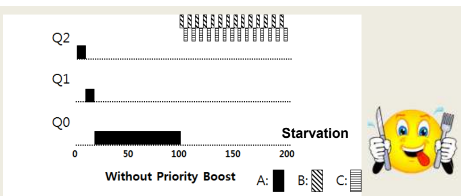
- Gaming the scheduler
  - Issue I/O operation at 99% completion of the time slice
  - Keeps job priority fixed – never lowered
- Job behavioral change
  - CPU/batch process becomes an interactive process

Priority becomes stuck



October 7, 2016 | TCSS422: Operating Systems [Fall 2016] Institute of Technology, University of Washington - Tacoma | L5.10

RESPONDING TO BEHAVIOR CHANGE

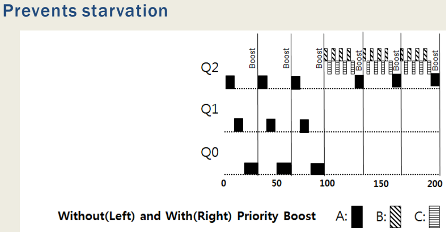


- Priority Boost
  - Reset all jobs to topmost queue after some time interval S

October 7, 2016 | TCSS422: Operating Systems [Fall 2016] Institute of Technology, University of Washington - Tacoma | L5.11

RESPONDING TO BEHAVIOR CHANGE - 2

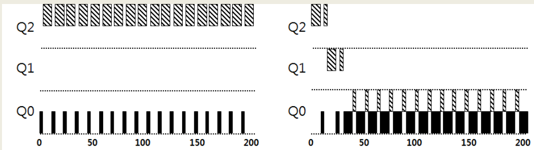
- With priority boost
  - Prevents starvation



October 7, 2016 | TCSS422: Operating Systems [Fall 2016] Institute of Technology, University of Washington - Tacoma | L5.12

PREVENTING GAMING

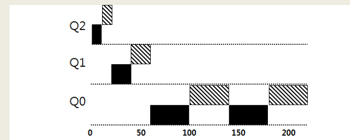
- Improved time accounting:
  - Track total job execution time in the queue
  - Each job receives a fixed time allotment
  - When allotment is exhausted, job priority is lowered



Without(Left) and With(Right) Gaming Tolerance

MLFQ: TUNING

- Consider the tradeoffs:
  - How many queues?
  - What is a good time slice?
  - How often should we “Boost” priority of jobs?
  - What about different time slices to different queues?



Example) 10ms for the highest queue, 20ms for the middle, 40ms for the lowest

PRACTICAL EXAMPLE

- Oracle Solaris MLFQ implementation
  - 60 Queues →
    w/ slowly increasing time slice (high to low priority)
  - Provides sys admins with set of editable table(s)
  - Supports adjusting time slices, boost intervals, priority changes, etc.
- Advice
  - Provide OS with hints about the process
  - Nice command → Linux

MLFQ RULE SUMMARY

- The refined set of MLFQ rules:
- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn’t).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority.
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced(i.e., it moves down on queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

PROPORTIONAL SHARE SCHEDULER

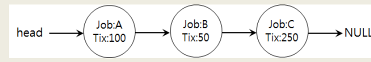
PROPORTIONAL SHARE SCHEDULER

- Also called fair-share scheduler
- Or lottery scheduler
  - Guarantee 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

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

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

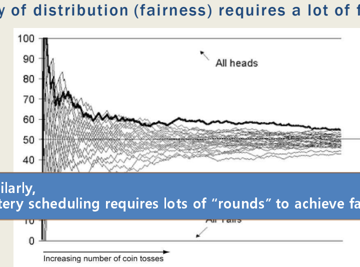
- Scheduler picks a **winning** ticket
  - Load the job with the winning ticket and run it
- Example:
  - Given 100 tickets in the pool
    - Job A has 75 tickets: 0 - 74
    - Job B has 25 tickets: 75 – 99

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

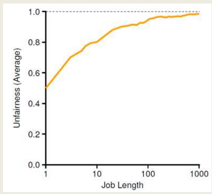
- Equality of distribution (fairness) requires a lot of flips!



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 <u>quite severe</u>.

TICKET ASSIGNMENT PROBLEM

- 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 SCHEDULING

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

STRIDE SCHEDULER

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

STRIDE SCHEDULER - 2

- Basic algorithm:
  1. Stride scheduler picks a job with the lowest pass value
  2. Scheduler increments job’s pass value by its stride and starts running
  3. Stride scheduler increments a counter
  4. When counter exceeds pass value of current job, pick a new job (go to 1)

STRIDE SCHEDULER EXAMPLE

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

<u>Priority</u>
C stride = 40
A stride = 100
B stride = 200

STRIDE SCHEDULER EXAMPLE - 2

- Randomly pick job A (all pass values=0)
- Set A's pass value to A's stride = 100
- Increment counter until > 100
- Pick a new job

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
- Time slice: Linux uses ***"Nice value"***
  - Nice value predates the CFS scheduler
  - Top shows nice values
  - Process command: `Ps ax -o pid,ni,cmd,%cpu`
- Nice Values: from -20 to 19
  - Lower is ***higher*** priority
  - Default is 0
- 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

COMPLETELY FAIR SCHEDULER - 2

- CFS uses weighted fair queueing
- Nice values become relative for determining time slices
  - Proportion of CPU time to allocate is relative to other queued tasks
- 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
- `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 - 3

- The task on a given runqueue (nice value) with the lowest `vruntime` will be scheduled text
- 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