IT308: Operating Systems

Scheduling: Multi-Level Feedback Queue

MLFQ: Goals

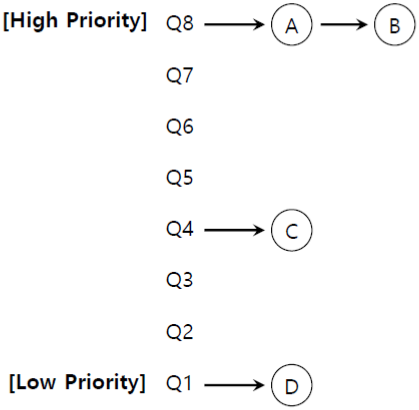
- How do we schedule jobs, without knowing their lengths, so that we
  - minimize turnaround time
  - minimize response time (for interactive jobs)

MLFQ: Goals

- How do we schedule jobs, without knowing their lengths, so that we
  - minimize turnaround time
  - minimize response time (for interactive jobs)
- Multi-Level Feedback Queue
  - Invented by Fernando J. Corbato in 1962 – won ACM Turing Award
  - Used by FreeBSD, Mac OS X, Solaris, Linux 2.6, Windows NT

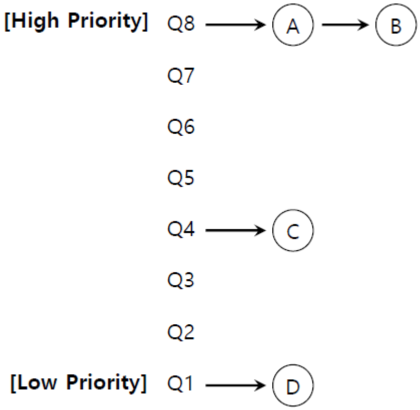
MLFQ: Basic Setup

- Consists of a number of distinct queues, each assigned a different priority level.
- Each queue has multiple ready-to-run jobs with the same priority.
- At any given time, choose to run the jobs in the queue with the highest priority.
- If there are multiple jobs at the same level, run them in Round-Robin



MLFQ: Basic Setup

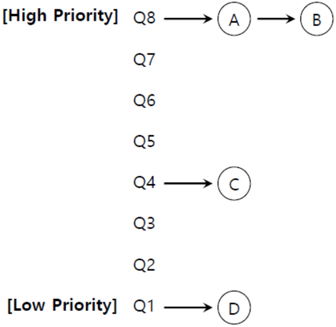
- Consists of a number of distinct queues, each assigned a different priority level.
- Each queue has multiple ready-to-run jobs with the same priority.
- At any given time, choose to run the jobs in the queue with the highest priority.
- If there are multiple jobs at the same level, run them in Round-Robin



So we see where the MLQ comes from, but not the F yet.

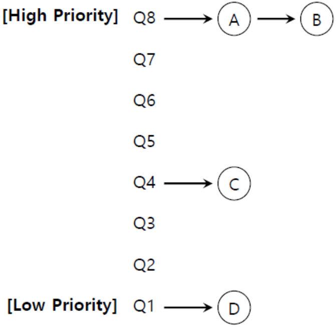
MLFQ: Rules (incomplete)

- Rule 1: if Priority(p) $>$ Priority(q), p runs (q doesn't)
- Rule 2: if Priority(p) $=$ Priority(q), run both in RR
- Does it work well? Why not?
- Rule 3: ????



MLFQ: Rules (incomplete)

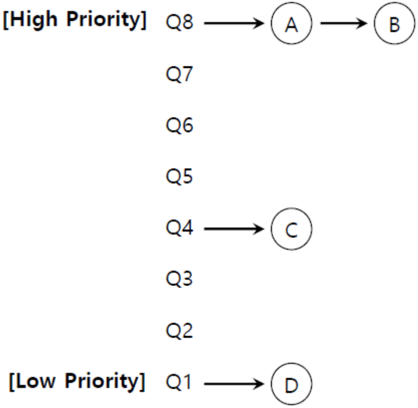
- Rule 1: if Priority(p) > Priority(q), p runs (q doesn't)
- Rule 2: if Priority(p) = Priority(q), run both in RR
- Does it work well? Why not?
- Rule 3: ????



The trick: change priorities of jobs sensibly to achieve good performance (turnaround time and response time)

First Attempt: Change Priority

- If A and B are long-running while C and D are interactive, response time is going to be crap.
- So long-running jobs should be low-priority.
- But remember we don't know how long a job is!



First Attempt: Change Priority

- Rule 3:
    - When a job enters the system it is place at the highest priority.

First Attempt: Change Priority

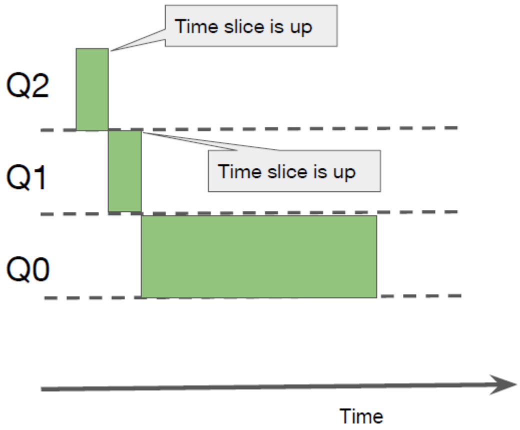
- Rule 3:
  - When a job enters the system it is place at the highest priority.
- Rule 4:
  - If a job uses an entire time slice (of some defined length), its priority is reduced (move down one queue)
  - If a job gives up the CPU before the time slice is up, it stays at the same priority level.

First Attempt: Change Priority

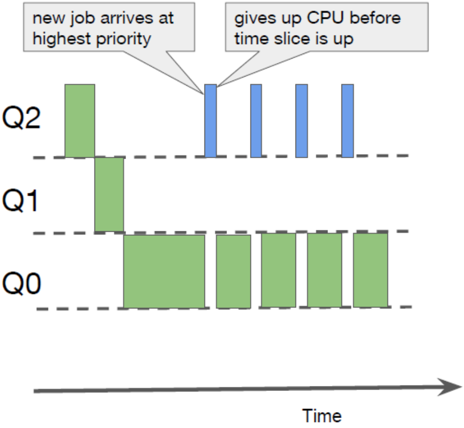
- Rule 3:
  - When a job enters the system it is place at the highest priority.
- Rule 4:
  - If a job uses an entire time slice (of some defined length), its priority is reduced (move down one queue)
  - If a job gives up the CPU before the time slice is up, it stays at the same priority level.

    (This is the F in MLFQ, i.e., feedback.)

Example: Single Long-Running Job



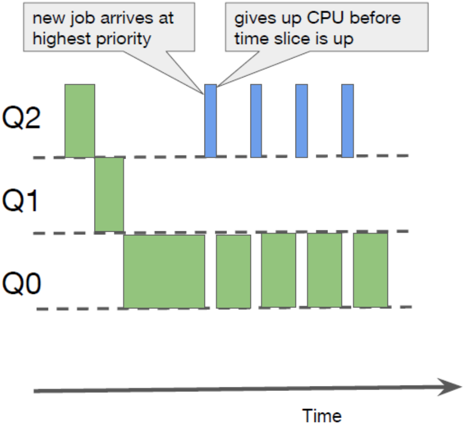
Example: then an interactive job arrives


new job arrives at highest priority
gives up CPU before time slice is up
Q2
Q1
Q0
Time

MLFQ first assumes every job is short.

- If it actually is short then good, run it quickly;
- if not, push down its priority and run it slowly; the longer the job is, the downer and slower.
  - i.e., MLFQ gradually learns that it is a long job.

Example: then an interactive job arrives

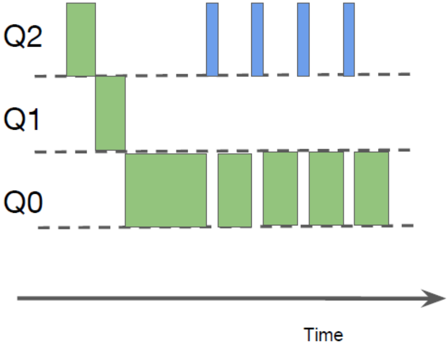

new job arrives at highest priority
gives up CPU before time slice is up
Q2
Q1
Q0
Time

MLFQ first assumes every job is short.

- If it actually is short then good, run it quickly;
- if not, push down its priority and run it slowly; the longer the job is, the downer and slower.
  - i.e., MLFQ gradually learns that it is a long job.
- It is basically approximating SJF.

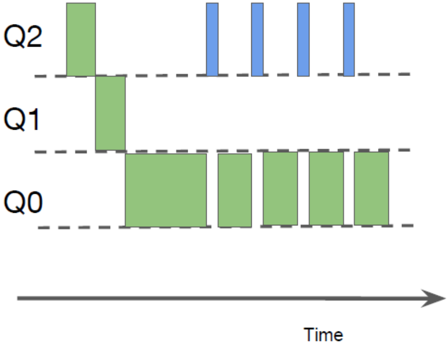
So it's all good . . . Right?

- Rule 1: if Priority(p) $>$ Priority(q), p runs (q doesn't)
- Rule 2: if Priority(p) $=$ Priority(q), run both in RR
- Rule 3: When a job enters the system it is place at the highest priority.
- Rule 4:
  - If a job uses to an entire time slice (of some defined length), its priority is reduced (move down one queue)
  - If a job gives up the CPU before the time slice is up, it stays at the same priority level.



So it's all good . . . Right?

- Rule 1: if Priority(p) > Priority(q), p runs (q doesn't)
- Rule 2: if Priority(p) = Priority(q), run both in RR
- Rule 3: When a job enters the system it is place at the highest priority.
- Rule 4:
    - If a job uses to an entire time slice (of some defined length), its priority is reduced (move down one queue)
    - If a job gives up the CPU before the time slice is up, it stays at the same priority level.



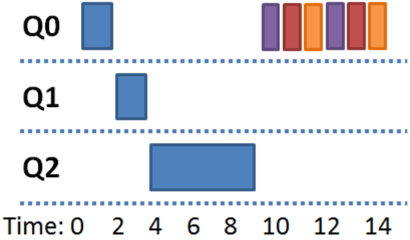
There are two problems!

Problem 1: Starvation

- If there are “too many” interactive jobs in the system, long-running jobs will never receive any CPU time.

Problem 1: Starvation

- If there are "too many" interactive jobs in the system, long-running jobs will never receive any CPU time.
- Example:

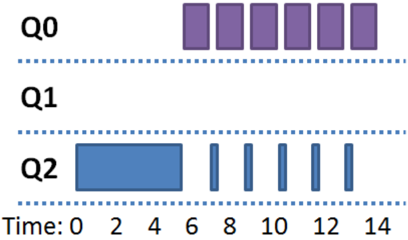


Problem 2: Gaming the scheduler

- Trick: relinquish the CPU right before the time slice ends
  - job never demoted (always high priority), monopolizes CPU time

Problem 2: Gaming the scheduler

- Trick: relinquish the CPU right before the time slice ends
  - job never demoted (always high priority), monopolizes CPU time
- Example:



Priority Boost

- Rule 5: After some time period S, move all jobs to the highest priority queue
- Solves two problems:

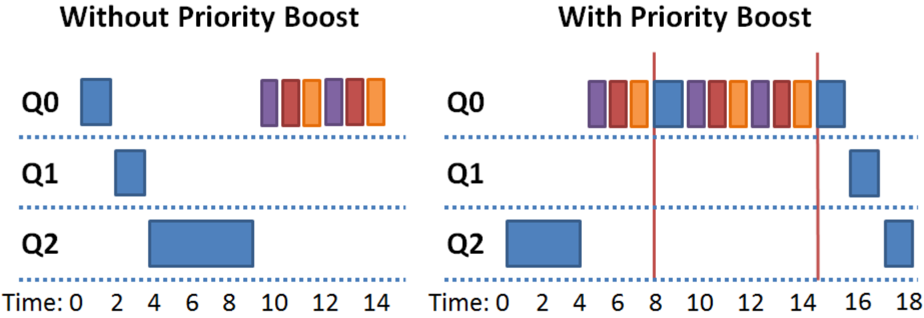
Priority Boost

- Rule 5: After some time period S, move all jobs to the highest priority queue
- Solves two problems:
  - Starvation: low priority jobs will eventually become high priority, acquire CPU time

Priority Boost

- Rule 5: After some time period S, move all jobs to the highest priority queue
- Solves two problems:
  - Starvation: low priority jobs will eventually become high priority, acquire CPU time
  - Dynamic behavior: a CPU bound job that has become interactive will now be high priority

Priority Boost Example



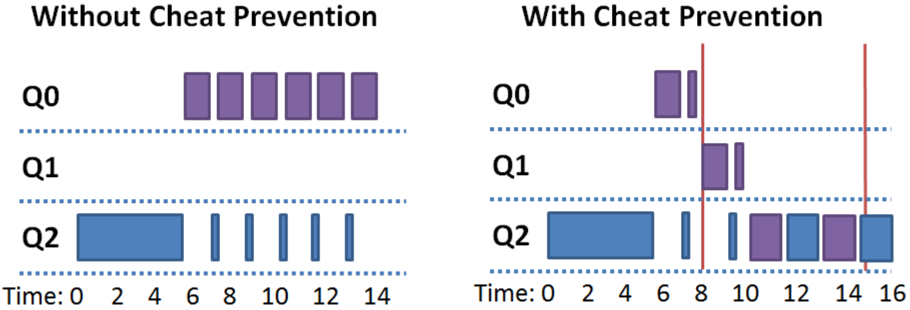
Revised Rule 4: Cheat Prevention

- Rule 4a and 4b let a job game the scheduler
  - Repeatedly yield just before the time limit expires
- Solution: better accounting
  - Rule 4: Once a job uses up its time allotment at a given priority (regardless of whether it gave up the CPU), demote its priority

Revised Rule 4: Cheat Prevention

- Rule 4a and 4b let a job game the scheduler
    - Repeatedly yield just before the time limit expires
- Solution: better accounting
    - Rule 4: Once a job uses up its time allotment at a given priority (regardless of whether it gave up the CPU), demote its priority
    - Basically, keep track of total CPU time used by each job during each time interval S
        - Instead of just looking at continuous CPU time

Preventing gaming



Tuning MLFQ

- How to parameterize MLFQ ?
  - how many queues?
  - how big should time slice be per queue?
  - how often should priority be boosted?

Tuning MLFQ

- How to parameterize MLFQ ?
  - how many queues?
  - how big should time slice be per queue?
  - how often should priority be boosted?
- No easy answers and need experience with workloads
  - varying time-slice length across different queues
  - e.g., lower priority, longer quanta