MLFQ Scheduling

Operating Systems
Baochun Li
University of Toronto

Assumptions revised

Each job runs for the same amount of time (relaxed)

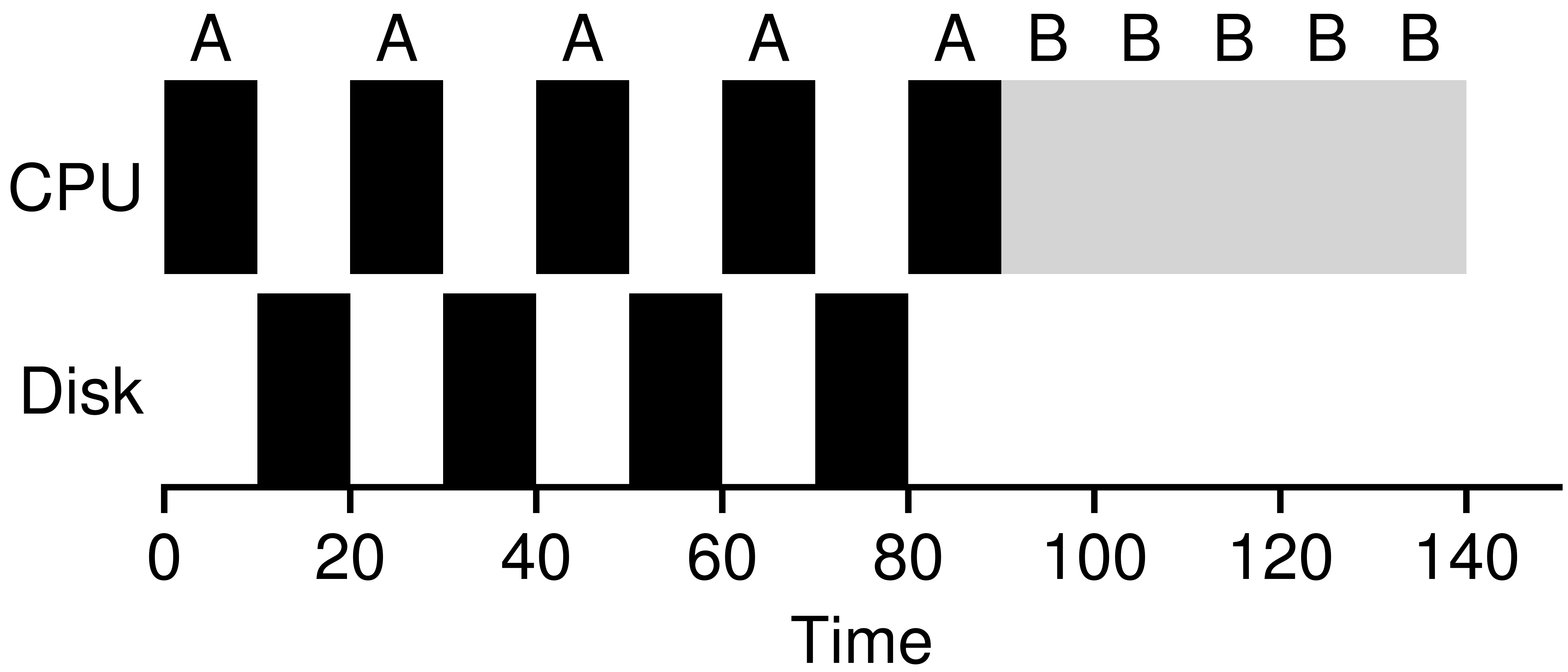
All jobs arrive at the same time (relaxed)

Once started, each job runs to completion (relaxed)

All jobs only use the CPU (no I/O)

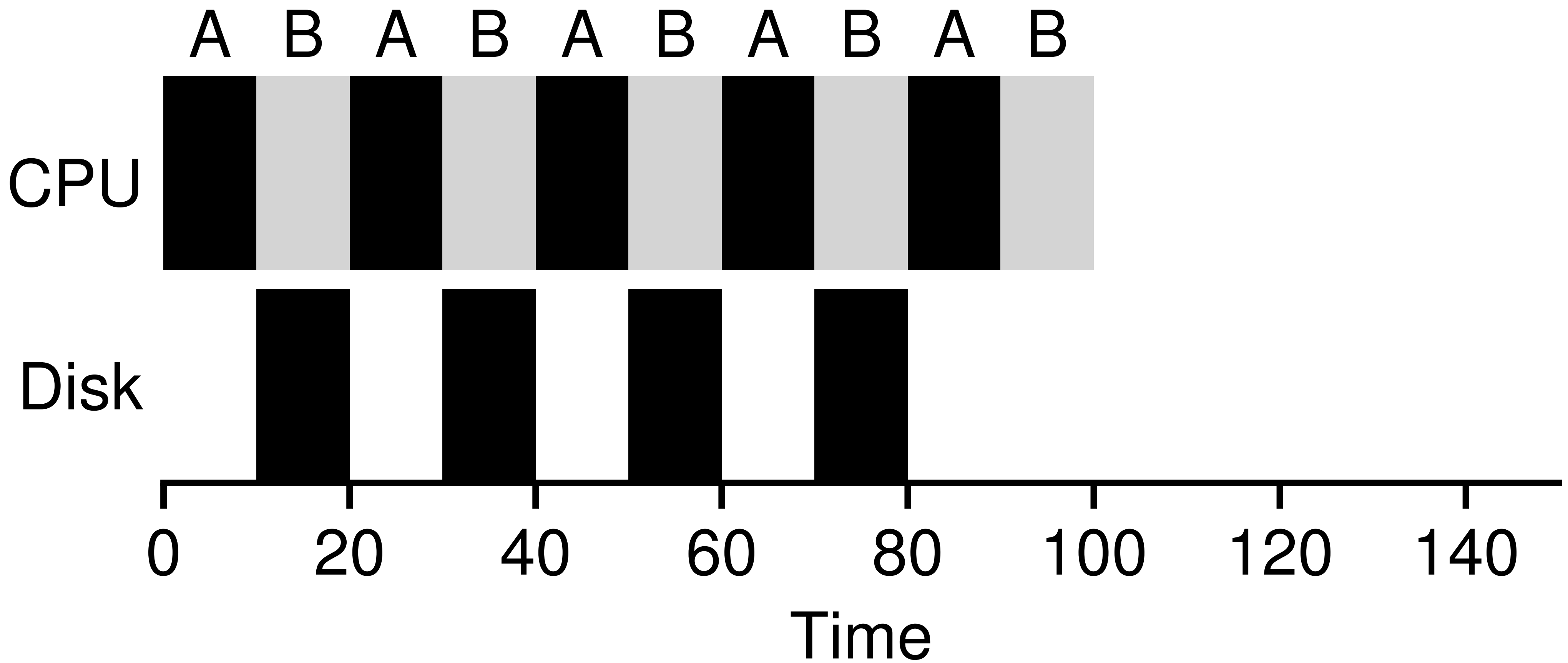
The run-time of each job is known

Relaxing the assumption to allow I/O



Scheduling result with STCF

Solution: treat each sub-job as an independent one



Scheduling result with STCF

Assumptions revised

Each job runs for the same amount of time (relaxed)

All jobs arrive at the same time (relaxed)

Once started, each job runs to completion (relaxed)

All jobs only use the CPU (no I/O) **(relaxed)**

The run-time of each job is known

How can we design a scheduler that minimizes response time for interactive jobs while also minimizing turnaround time without knowledge of job run-time?

MLFQ: Design objective

Relax the assumption that the length of each job is predictable

Achieve good response times for interactive jobs (I/O bound jobs), *and* good turnaround times for CPU-bound jobs

Multi-Level Feedback Queue: a general class of scheduling policies

- First proposed by Corbato *et al.* in 1962 — who later won the ACM Turing Award

Static Priority Scheduling

Our starting point

Preemptive Static Priority Scheduling

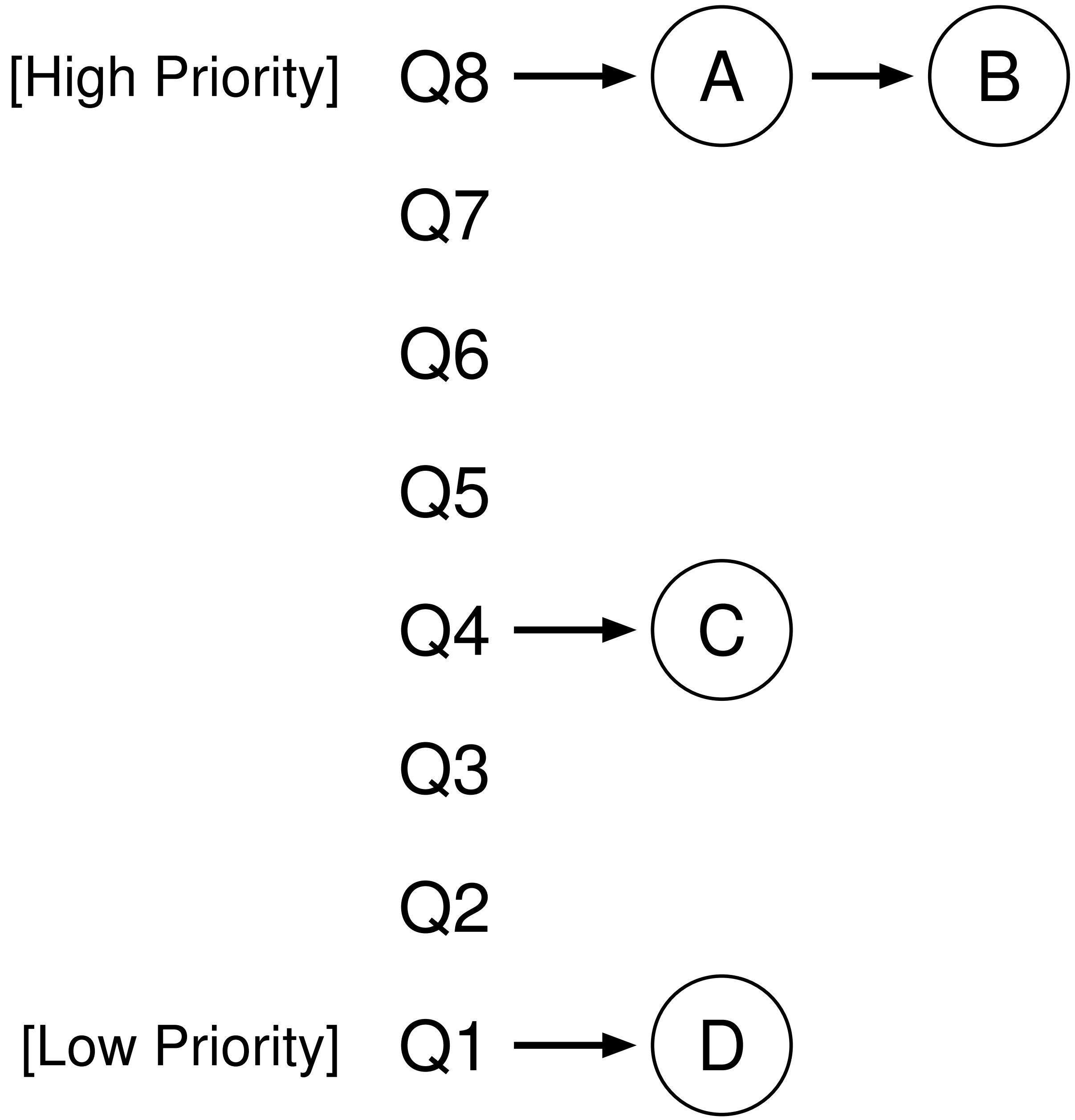
Assign a static priority to each job when it is started

Always chooses to run the highest priority job

> **Rule 1:** If Priority(A) > Priority(B), A runs.

Equal-priority job are scheduled in round-robin as they arrive

> **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.



Setting priorities: I/O bound (interactive) jobs need higher priorities, and the priorities for CPU bound jobs should be lower.

Here's a potential problem with static priority scheduling

The Priority Inversion Problem

Let us assume that spin loops around the TSL instruction are used to wait for mutual exclusion locks

The Priority Inversion Problem —

- A high-priority thread **H** becomes ready to run when a low-priority thread **L** is in the critical section
- With preemptive static priority scheduling, **H** always runs first if it is ready
- **H** begins busy waiting
- **L** is never scheduled when **H** begins running, and never leaves the critical section
- **H** waits forever — **deadlock occurs!**

One possible intuitive solution

A thread should block, not spin, when waiting to enter the critical section — problem solved?

The Priority Inversion Problem Remains

High priority thread blocks, waiting for the low priority thread to exit the critical section

A medium priority thread runs

The low priority thread never gets to run and to exit the critical section

The medium priority thread now takes priority over the high priority thread!

Real-world example: the Mars Pathfinder

Access to a shared "information bus" — mutual exclusion locks

shared by a high-priority bus management thread, and a low-priority meteorological data thread

Very rarely, a medium-priority communications thread is scheduled when the low-priority thread uses the shared bus

Since the high-priority is blocked waiting to access the shared bus

Neither the high-priority nor the low-priority threads get to execute again

Watchdog timer goes off when priority inversion occurs — panic — total system reset

The solution

Solution: priority inheritance — low-priority thread inherits the priority of the high-priority thread when it has the lock

- A research paper in 1990, titled "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," in IEEE Transactions on Computers

The operating system used was VxWorks, a real-time OS

- It has a global variable to enable priority inheritance
- The variable was set remotely while the Pathfinder was on Mars!
- And the problem was solved successfully
- QNX, VxWorks' competitor, was acquired by Blackberry and was later used as the foundation for Blackberry 10

Now let’s get back to

Multi-Level Feedback Queue

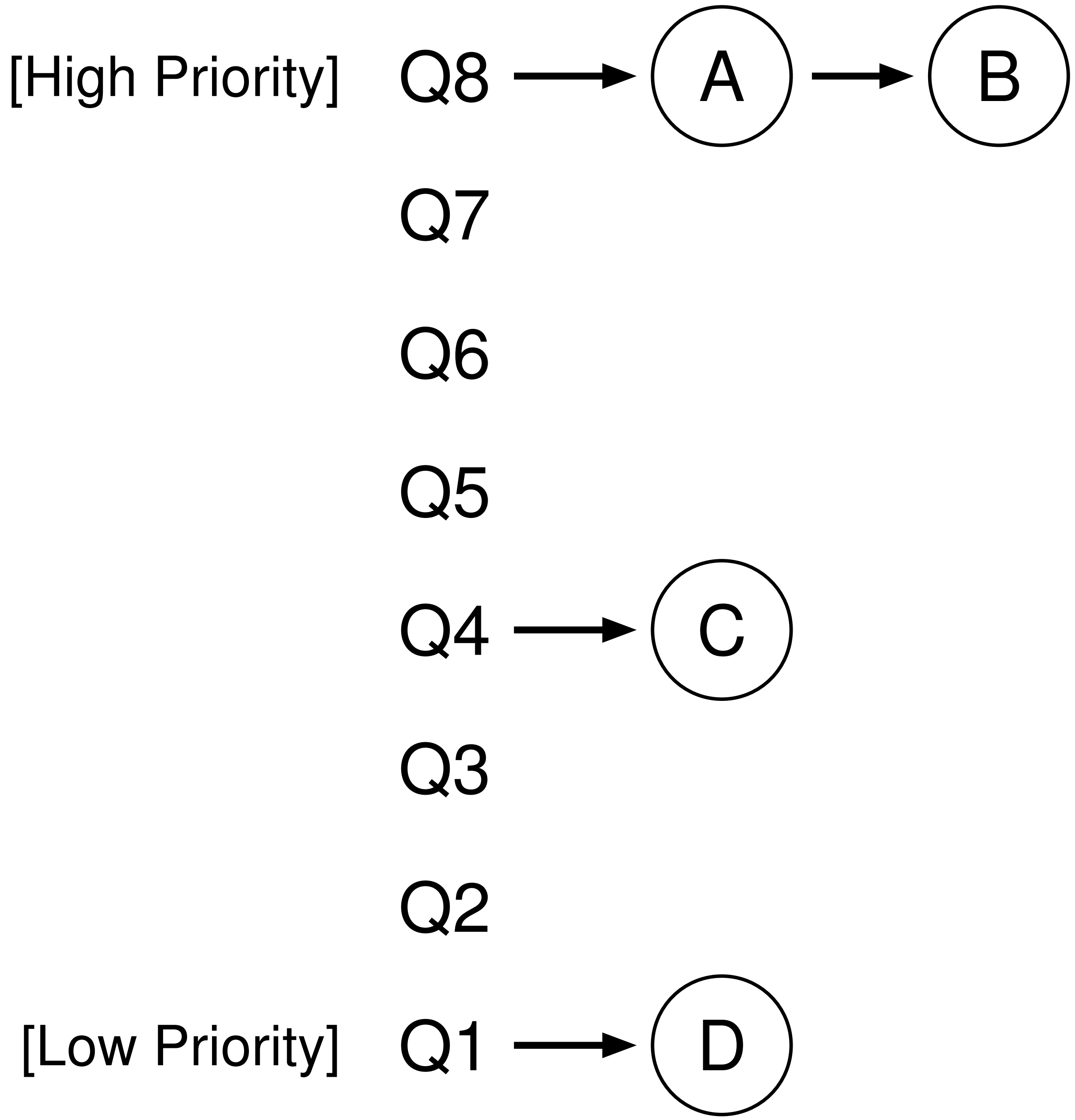
Priorities cannot be static!

Adjusting priorities

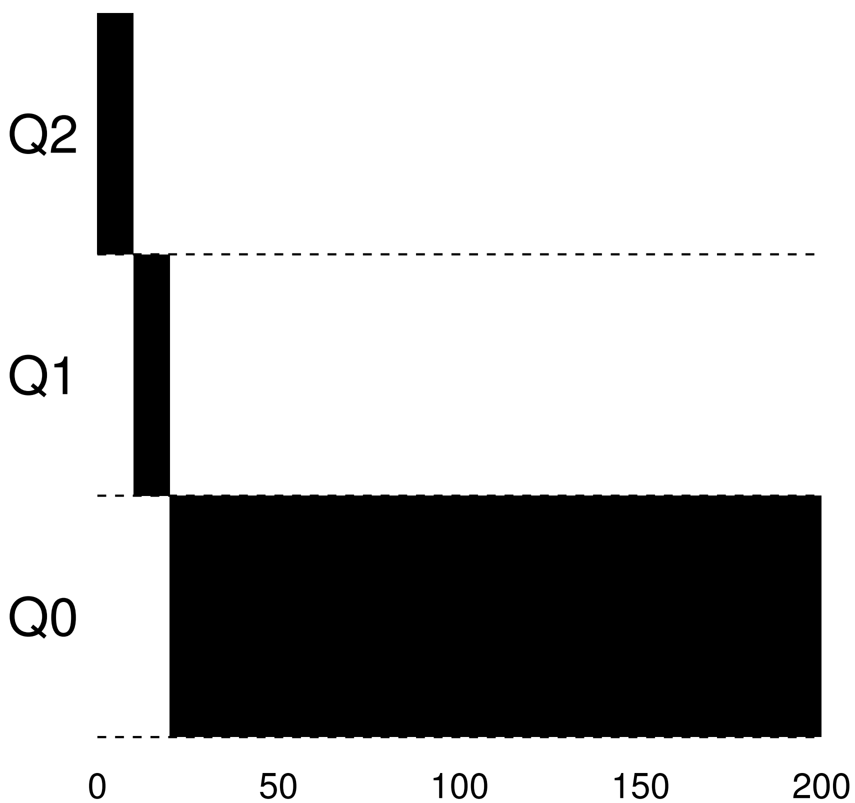
Rule 3: When a job enters, it has the highest priority.

Rule 4a: If a job uses up an entire time slice, reduce its priority by one.

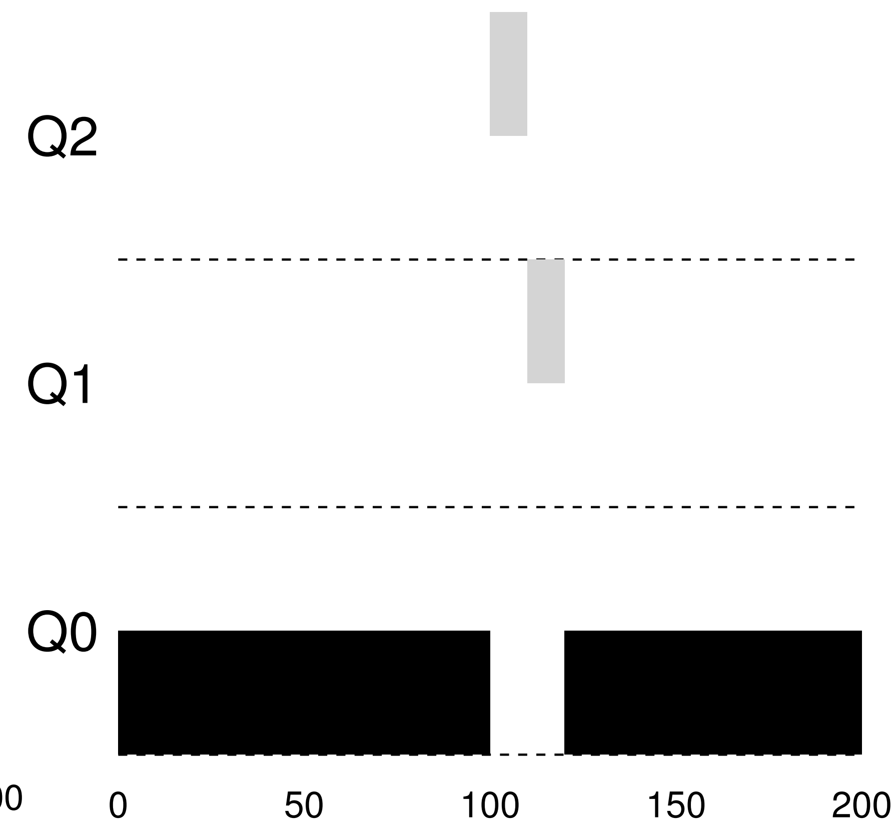
Rule 4b: If a job gives up the CPU before the time slice is up, keep it at the same level.



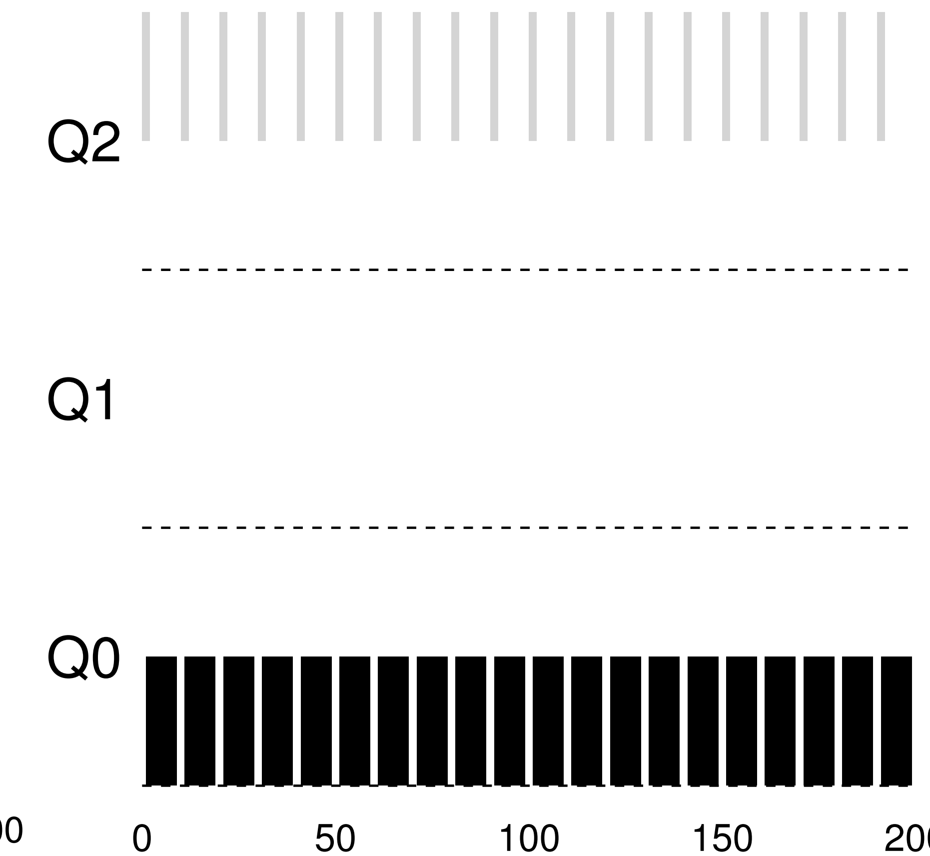
Interactive vs. CPU-bound jobs



A long-running job



A short job arrives



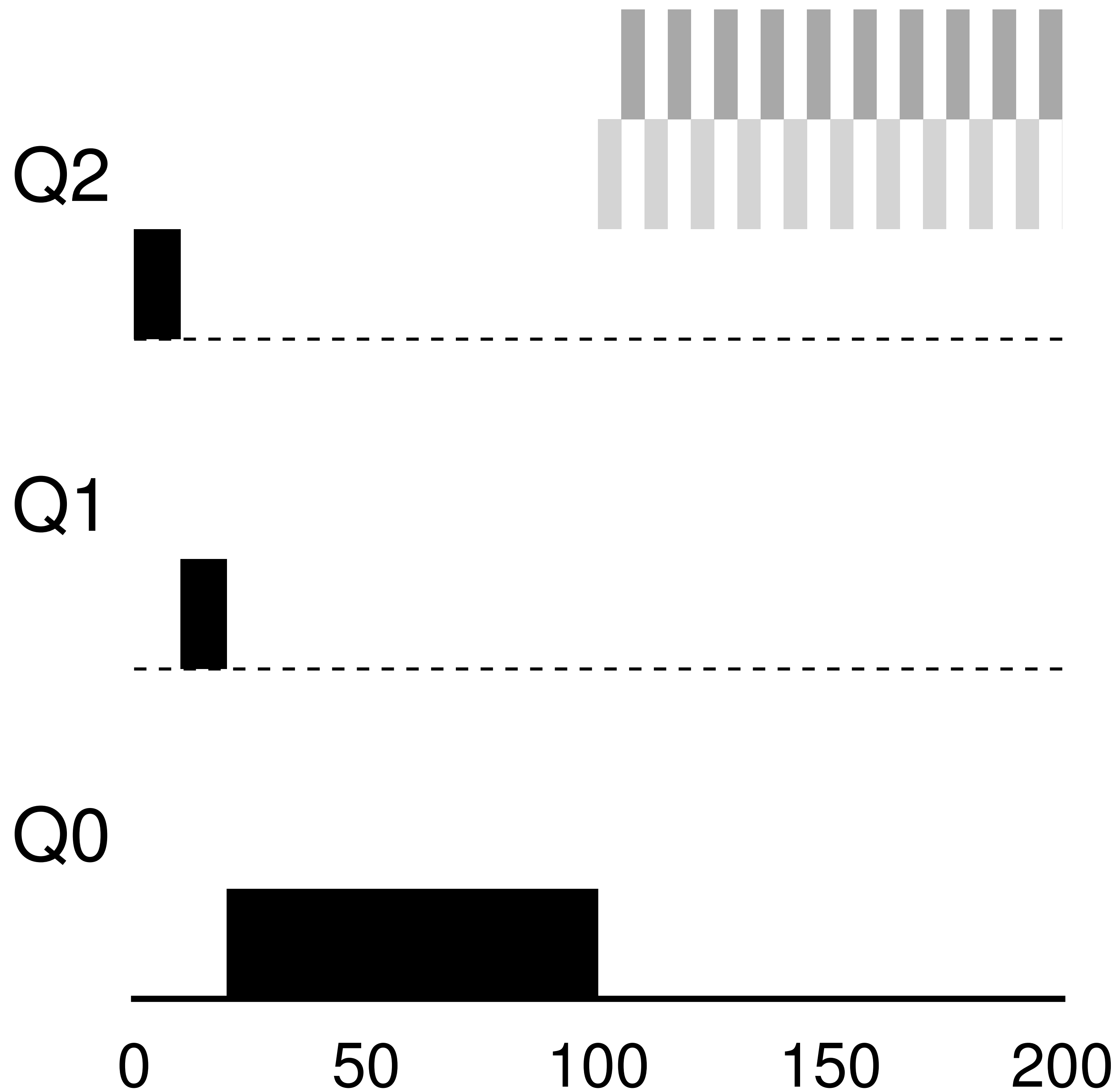
A mix of both interactive & CPU bound job

Assumption: when a job arrives, it is assumed to be interactive (I/O-bound).

MLFQ approximates preemptive SJF.

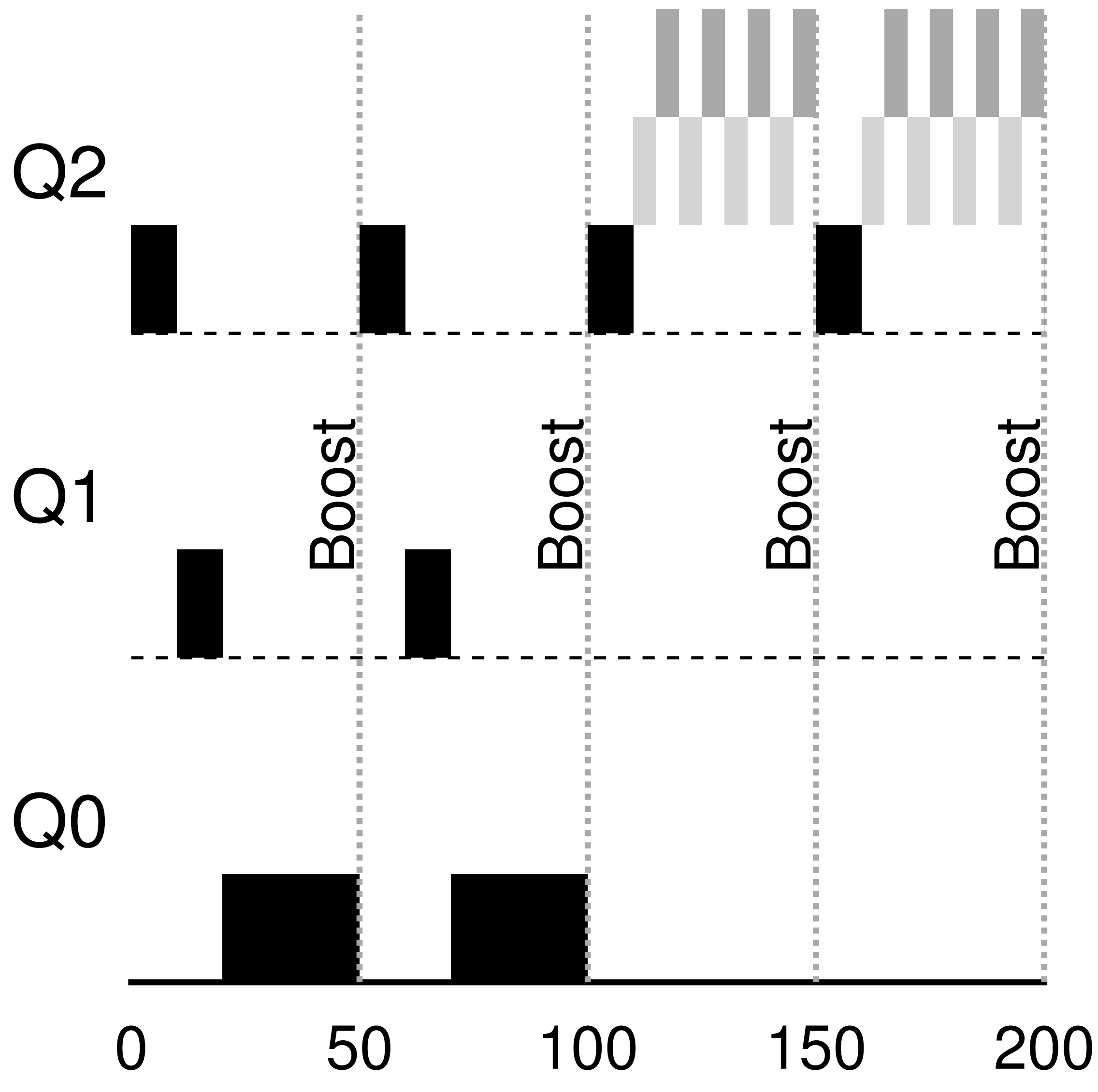
What problems does it have?

Starvation: CPU-bound jobs may never run

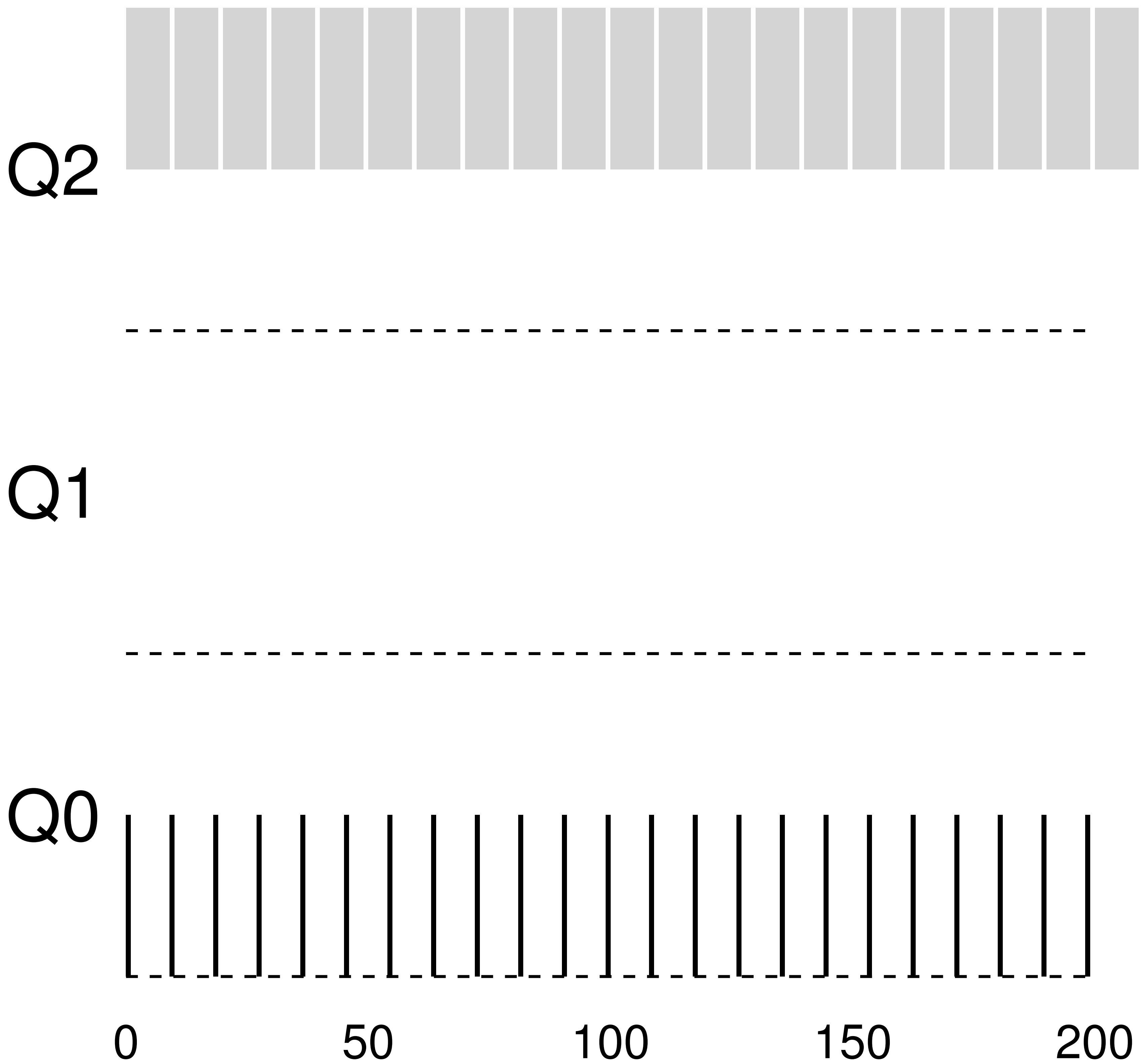


One possible fix: priority boost

Rule 5: After some time period, move all the jobs in the system to the topmost queue.

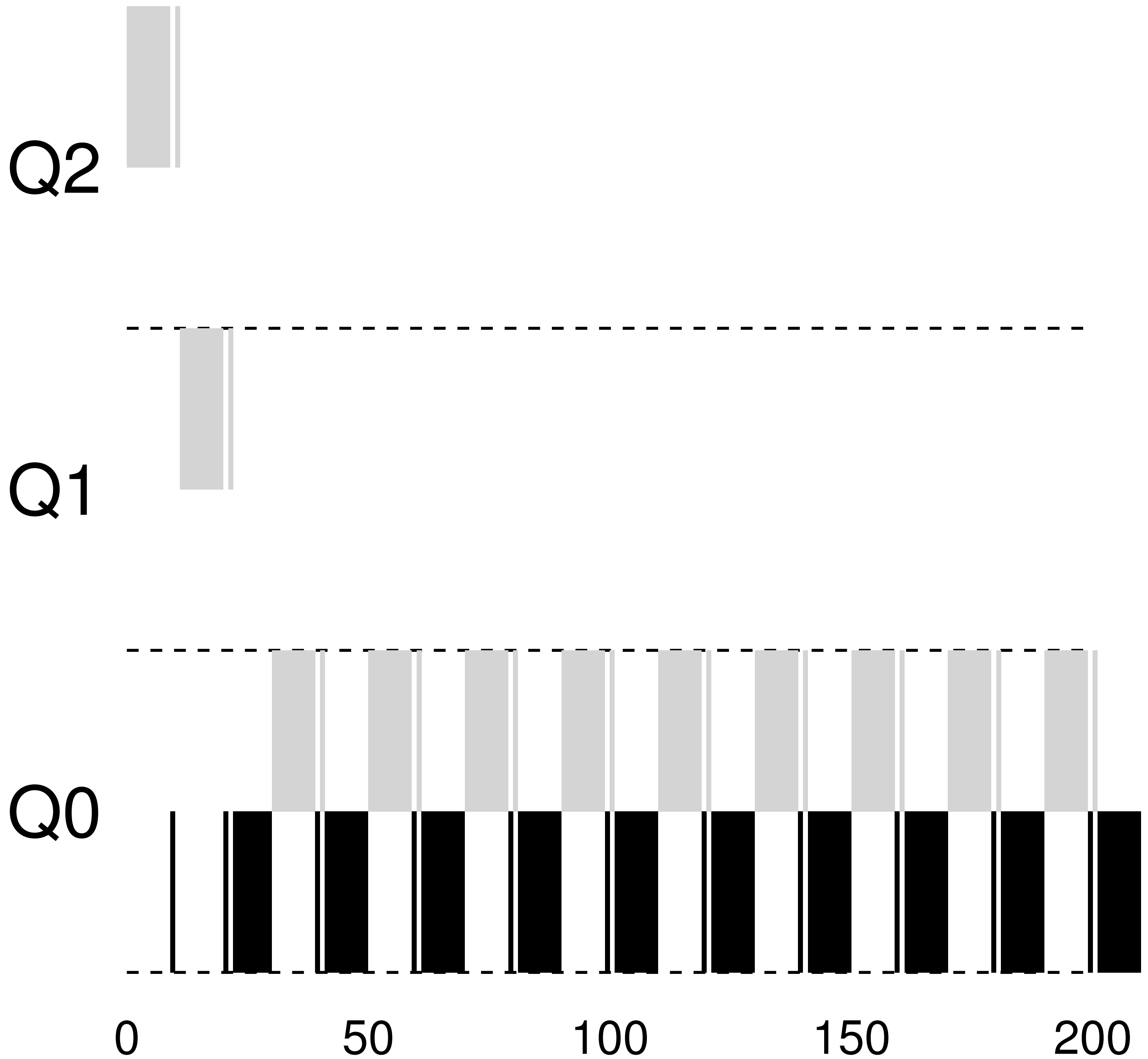


A user may game the scheduler



One possible fix: better accounting

Rule 4: Once a job used up its time allotment at a given level, reduce its priority by one.



Tuning the parameters

How many priorities?

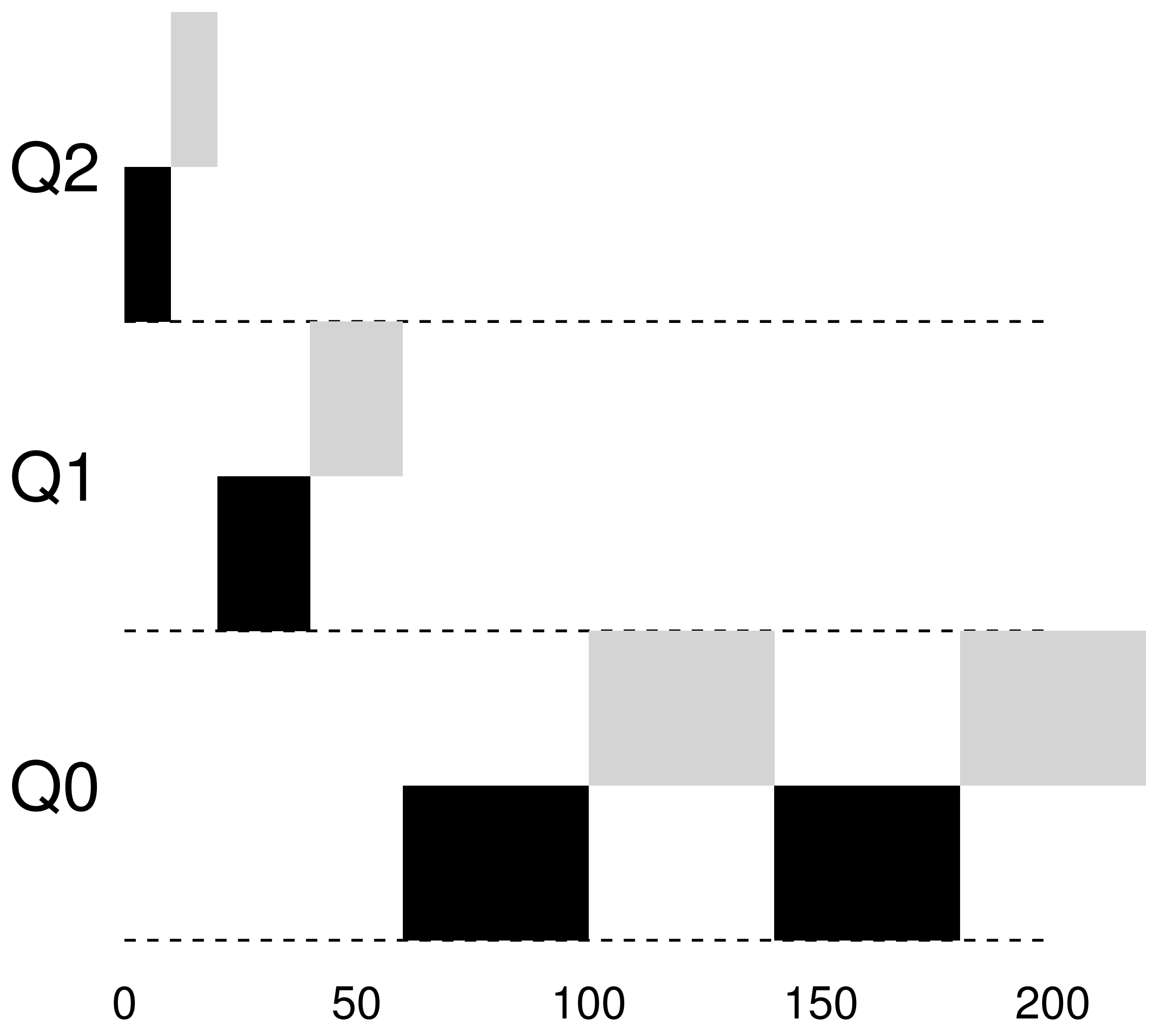
Default is 60 in Solaris

Time slice lengths?

From 20 ms (highest priority) to a few hundred ms (lowest priority) in Solaris

Frequency of priority boosts?

About 1 second in Solaris



The O(1) Scheduler in Linux

A case study

O(1) Scheduler: Linux 2.6.8.1 (2004) — 2.6.23

One "runqueue" per processor

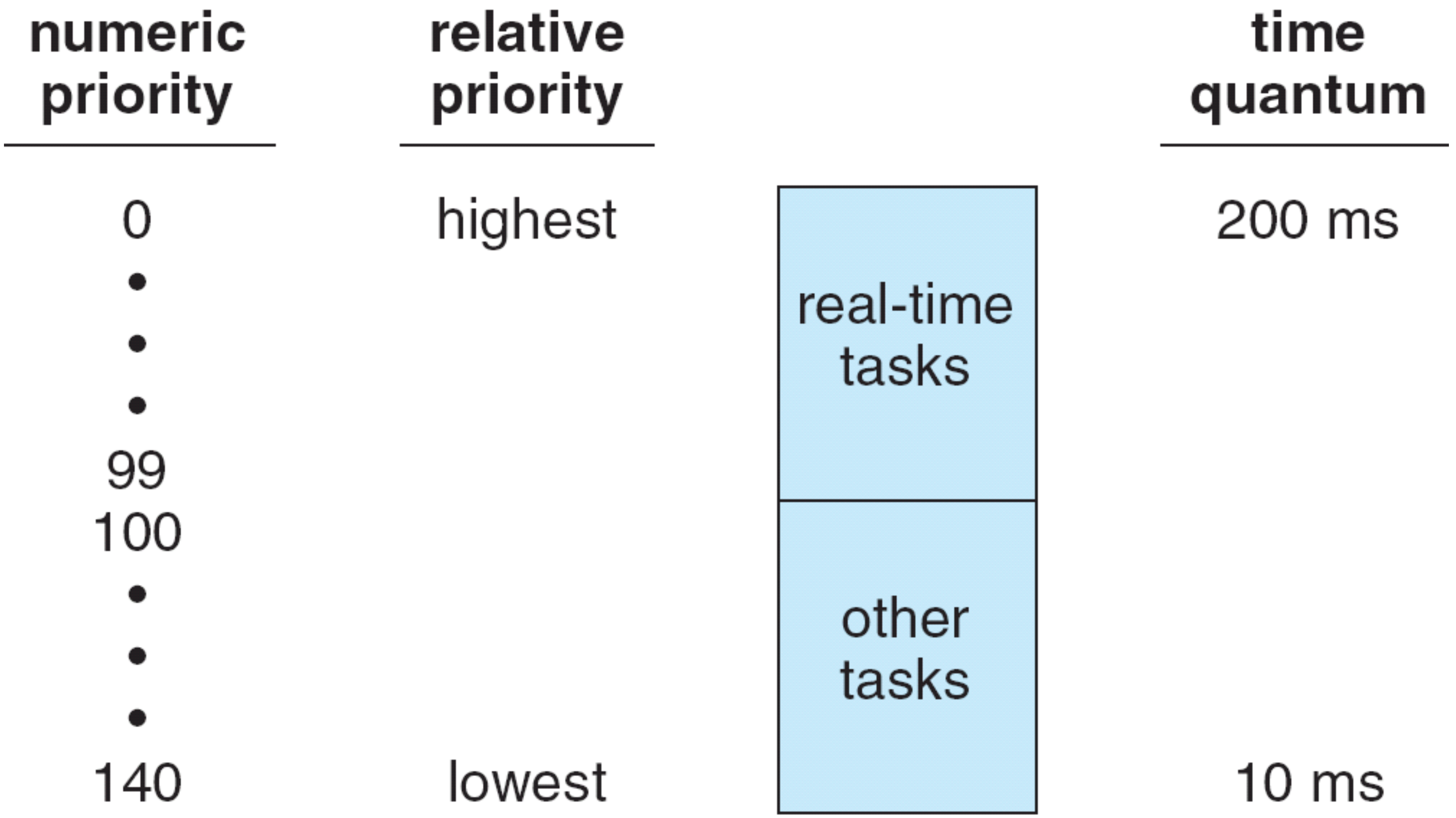
Two priority arrays: active and expired

- 140 priorities in total, 200 ms (highest priority) to 10 ms time slice
- each element in the array is a linked list of threads

Scheduler selects a task from the highest-priority active array

- O(1) operation
- threads in a certain priority: round-robin fashion
- when a thread's time slice expires, it is moved to the expired array, with an adjustment to its priority level

Priorities and time slices in the O(1) Scheduler



The runqueue data structure

When the time slice expires, an active thread is added to the expired array, possibly with an adjusted priority level

O(1) Scheduler: Linux 2.6.8.1 (2004) — 2.6.23

Different priority levels has different time slices

- 200 ms at priority 0, 10 ms at priority 139

Static priority set by "nice": initial priority

- nice values range from -20 to 19 (mapped to 100 to 139 in the runqueue)
- higher value -> lower priority
- default priority: 0 (mapped to 120)
- can be changed via the nice() system call

O(1) Scheduler: Linux 2.6.8.1 (2004) — 2.6.23

The active and expired priority arrays will be swapped when there are no threads in the active array

When a thread is moved from the active to the expired array, its priority will be adjusted based on a dynamic adjustment scheme —

`sleep_avg`: added after sleep, deducted after running

```
bonus = CURRENT_BONUS(p) - MAX_BONUS / 2;
prio = p->static_prio - bonus;
```

`CURRENT_BONUS(p)` is defined as follows:

```
(p->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG
```

What we've covered so far

Three Easy Pieces, Chapter 8 (Scheduling: MLFQ)

The "Linux CPU Scheduler" document, Sections 5.1.1 — 5.4.4