Universidade Estadual de Campinas
Instituto de Computação

**MC504 Sistemas Operacionais**

# T05 CPU Virtualization: Scheduling
# Multi-Level Feedback Queue

*Referência principal*
Ch.8 of Operating Systems: Three Easy Pieces by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

*Discutido em classe em 20 de agosto de 2018*

Arthur João Catto, PhD

2º semestre de 2018

## Multi-Level Feedback Queue

- Originally designed by Fernando Corbató in 1962, refined over the years and still adopted in modern systems.
- MLFQ tries to address two apparently conflicting requirements
  - Optimizing *turnaround time* without prior knowledge of job length.
  - Minimizing *response* time in order to make the system friendlier to interactive users.

Given that we hardly know anything about a process, how can a scheduler achieve those goals?

How can a system learn as it runs the characteristics of the processes in order to make better scheduling decisions?

How can a scheduler learn from the past to make an educated guess about the future?

## A warning

Learning from the past to predict the future is a common approach in operating systems and in many other places in Computer Science.

Such approaches work when jobs have phases of behavior and are thus somewhat predictable.

One must be careful with such techniques, as they can easily be wrong and drive a system to make worse decisions than it would have made with no knowledge at all.

## MLFQ's basic rules

- MLFQ uses a number of distinct queues, each assigned a different priority level.
- At any given time, every job that is ready to run is on a single queue.
- Priorities are used to decide which job should run at a given time.

**Rule 1**: If Priority(A) > Priority(B), A runs while B doesn't.

**Rule 2**: If Priority(A) = Priority(B), A and B will be scheduled using RR.

# MLFQ Example

| | | | | | |
|---|---|---|---|---|---|
| *highest priority* | Q8 | → | A | → | B |
| | Q7 | | | | |
| | Q6 | | | | |
| | Q5 | | | | |
| | Q4 | → | C | | |
| | Q3 | | | | |
| | Q2 | | | | |
| *lowest priority* | Q1 | → | D | | |

## How MLFQ sets priorities

- If the priorities shown in the example were fixed, processes C and D would not be scheduled until A and B had finished.
- The key to MLFQ scheduling lies in how it sets *dynamic priorities*.
- MLFQ varies the priority of a job based on its observed behavior.
  - If a job repeatedly relinquishes the CPU while waiting for I/O, MLFQ will keep its priority high, as it is likely to be an interactive process.
  - If a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority.
- MLFQ tries to learn about processes as they run, and thus use the history of the job to predict its future behavior.

## Attempt #1: Adjusting priorities

- How to dynamically set the priority level of a job (and thus the queue where it belongs)?
  - Let's assume that the workload is a mix of
    - short-running interactive I/O-bound jobs, and
    - longer-running CPU-bound jobs where response time isn't important.

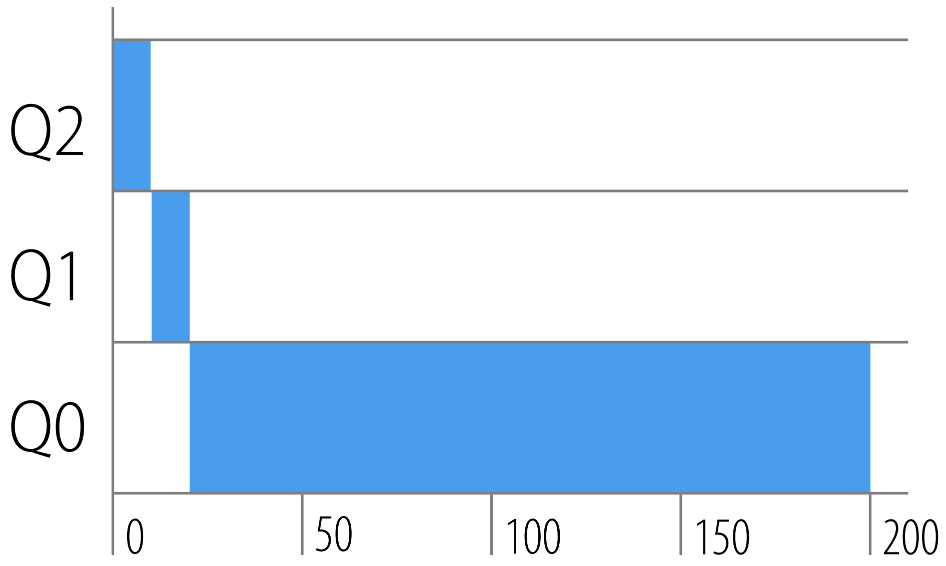
**Rule 3**: When a job enters the system, it gets highest priority and is placed in the topmost queue.

**Rule 4a**: If a job uses up an entire time slice while running, its priority is reduced and it moves down one queue.

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

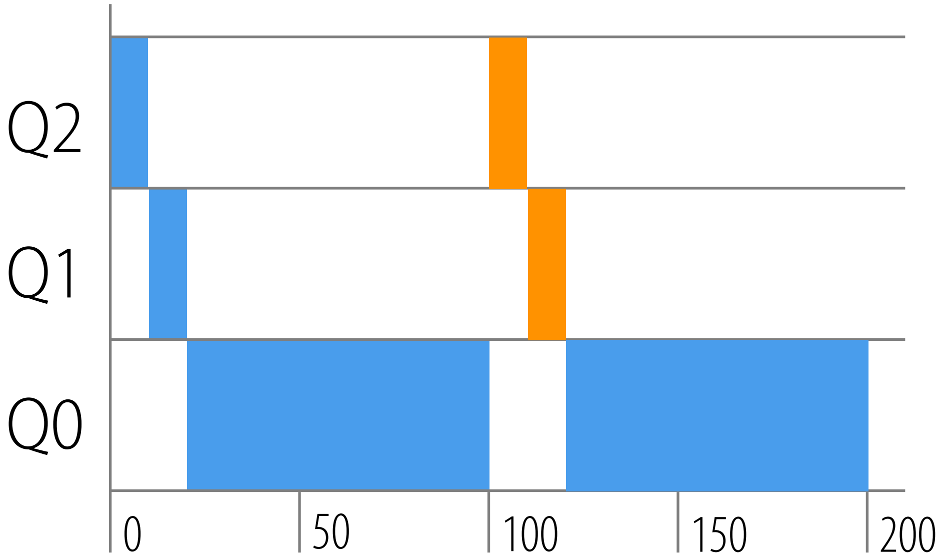
# Example 1: A single long-running job

- Assumptions
  - A three-queue MLQF scheduler with time slice 10ms
  - A long-running CPU-bound job



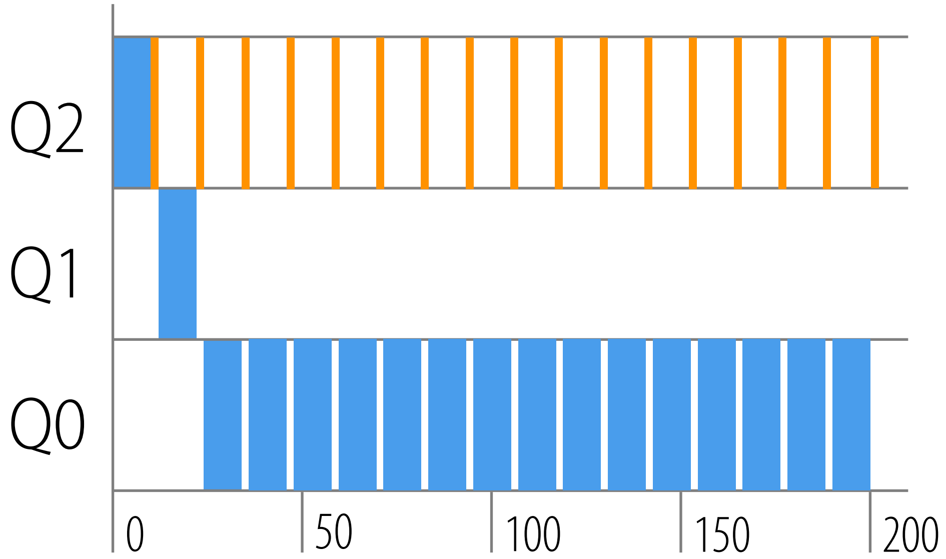
# Example 2: Along came a short-running job

- Assumptions
  - A three-queue MLQF scheduler with time slice 10ms
  - A long-running CPU-bound job
  - A short running (20ms) CPU-bound job arriving at t=100ms



## Example 3: What about I/O?

- Assumptions
  - A three-queue MLQF scheduler with time slice 10ms
  - A long-running CPU-bound job
  - A long running IO-bound job arriving just after the other one and releasing the CPU after 2ms on each burst



If our MLFQ approach performs so well on all these situations, why have we called it “a first attempt”?

Could it contain any serious flaws that we may have missed?

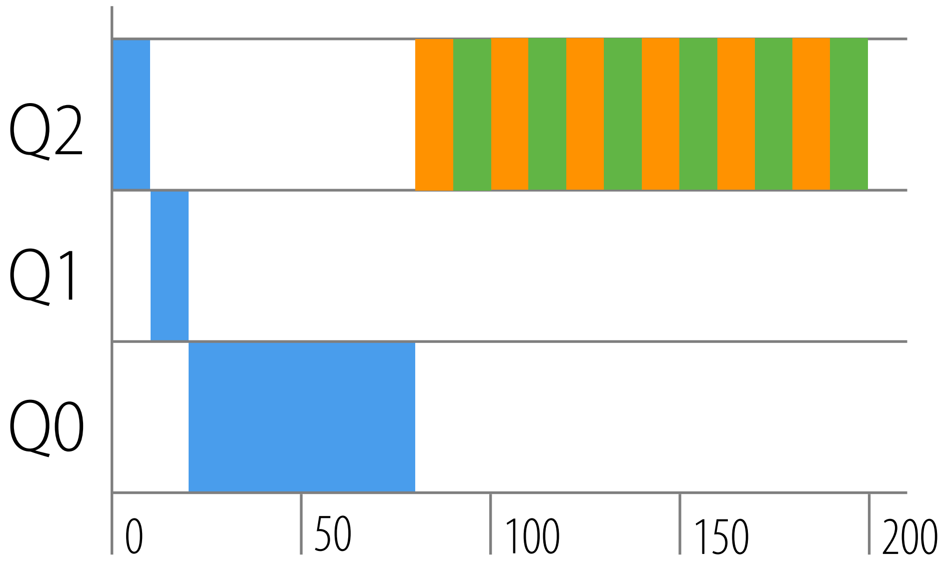
What do you think?

## Three problems with our current MLFQ

- Starvation
  - What would happen to the first job if there were “too many” concurrent IO-bound jobs?
- Gaming the scheduler
  - What would happen to a long-running CPU-bound process that issues a quick fake IO operation after consuming 99% of its time slice?
- Changing behavior
  - What would happen to an originally CPU-bound process that transitions to interactive behavior after some time?

# Example 4: Two processes starving another one

- Assumptions
  - The same system and same long-running CPU-bound job
  - Two long running IO-bound jobs arriving at *t*=80ms and doing 10ms IO operations just before the end of their time slices

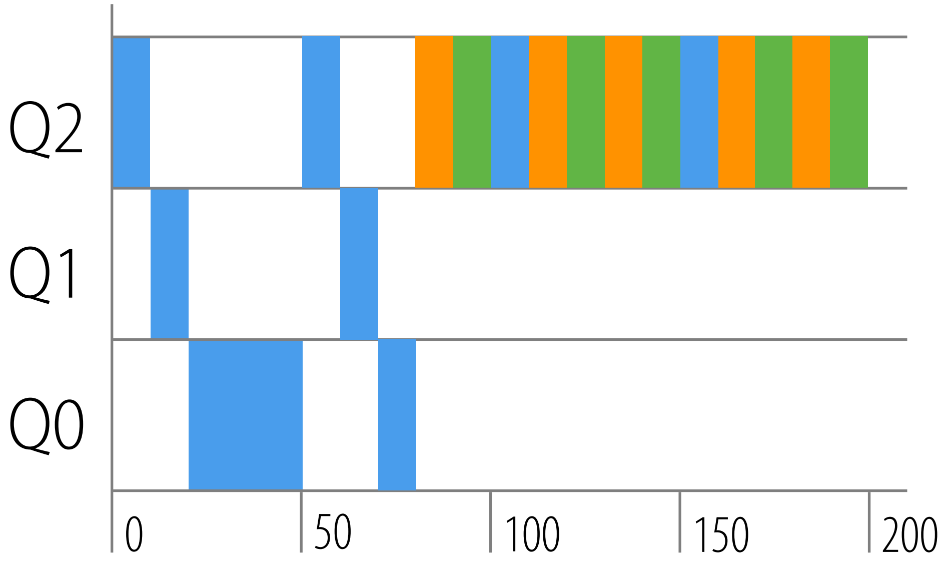


## Attempt #2: Priority boost

- To avoid starvation let's try another simple rule…

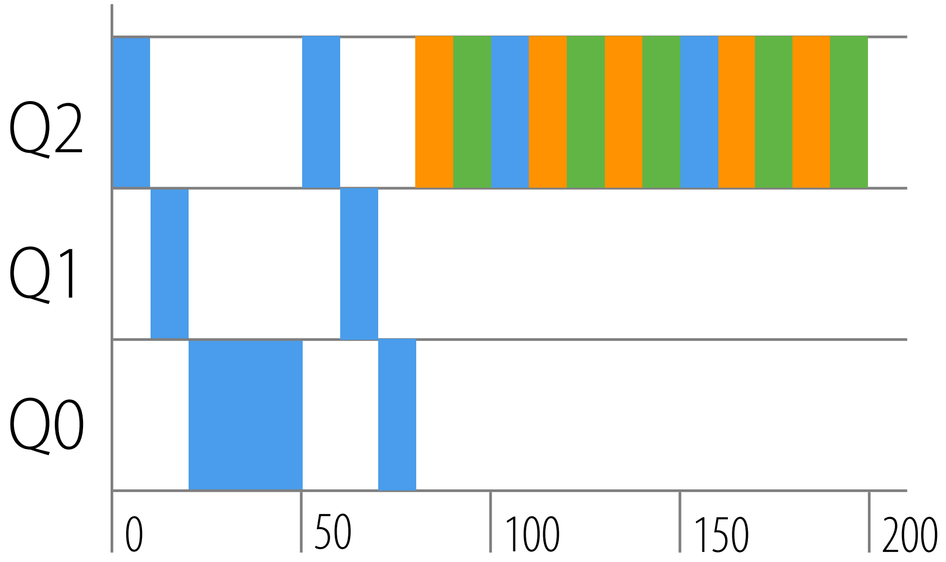
**Rule 5**: After some time period *S*, all jobs in the system are moved to the topmost queue.

- Let's assume that in our system *S*=50ms.



# Example 5: Unfair access to the CPU

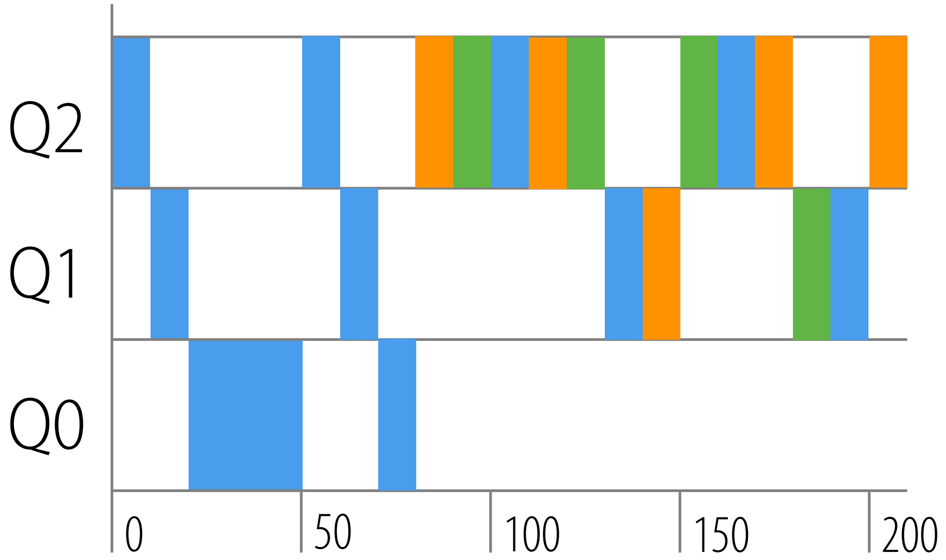
- Rule 5 avoids starvation but doesn't prevent one or more processes from gaming the scheduler and getting an unfair share of the CPU.
- Suppose that in Example 4 the IO-bound processes are gaming the CPU.
- Although the other process doesn't starve anymore, they keep getting an unfair share of CPU time by not being demoted by the scheduler.



# Attempt #3: Better accounting

- Gaming the CPU can be prevented by a slight changing Rules 4a and 4b

**Rule 4**: Once a job uses up its time slice (no matter how many times it has given up the CPU), its priority is decreased and it moves down one queue.



## Summary of MLFQ rules in our model

1. If Priority(A) > Priority(B), A runs (B doesn't).
2. If Priority(A) = Priority(B), A and B run in RR.
3. When a job enters the system, it is granted the highest priority and is placed in the topmost queue.
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 and it moves down one queue.
5. After some time period S, all the jobs in the system are moved to the topmost queue.

## Tuning MLFQ

- There are several other likely issues in MLFQ scheduling, many of them related to *parameterizing* the scheduler.
  - How many queues should there be?
  - How big should the time slice be?
  - Should the time slices be different for different queues?
  - How often should the priority be boosted to avoid starvation and account for changes in process behavior?
- MLFQ delivers excellent overall performance (similar to SJF/STCF) for short-running IO-bound jobs, and is fair and makes progress for long-running CPU-bound jobs, without demanding any *a priori* knowledge of its workload.
- This makes MLFQ the base scheduler of choice of many modern systems.