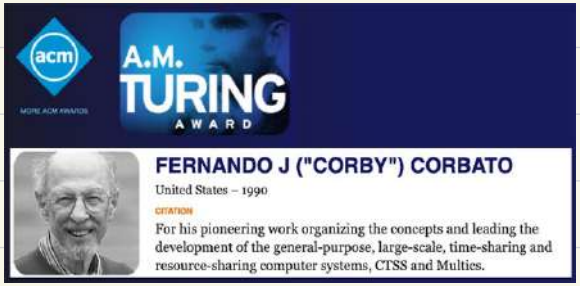
Scheduling: The Multi-level Feedback Queue (MLFQ)

Question: How to relax Assumption 5?
i.e., we don't know the run-time of each job.

	Turnaround time	response time	Knowledge of run-time
SJF/STCF →	good	bad	needed
RR →	bad	good	not necessarily.

Challenge: Given that we in general do not know anything about the job, how to design a scheduler that optimizes both turnaround time and response time?

↳ MLFQ



Historical origins of MLFQ

MLFQ scheduler was first described by Fernando J. Corbato and his colleagues in 1962, as part of their work on the Compatible Time-Sharing System (CTSS) at MIT.

CTSS was one of the earliest time-sharing systems which allowed multiple users to interact with a single computer as if each had their own private machine, in contrast to the

Batch processing systems before. CTSS fundamentally changed this model by enabling interactive computing.

MLFQ scheduler → a clever dynamic heuristic-based design, where

short and interactive jobs get priority, still allow CPU bound (long running) processes to make progress.

How do they achieve this? → adjusting priorities based on the recent CPU usage.

i.e., it learns from the past to predict the future.

Corbato's work on CTSS and later on Multics (which later influenced Unix → Linux/macOS) deeply shaped modern computing. In recognition of his contributions, Corbato was awarded the ACM Turing award in 1990, often called the "Nobel Prize of computing.

MLFQ objective: without a prior knowledge of job length
optimize turnaround time - run shorter jobs first
minimize response time.

Idea: * maintains a number of distinct queues
* each queue is assigned a different priority level

* A job that is ready to run is on a single queue

* Use RR scheduling among jobs in the same queue.

Basic rules:

Rule 1: If priority (A) > priority (B), A runs (B doesn't)

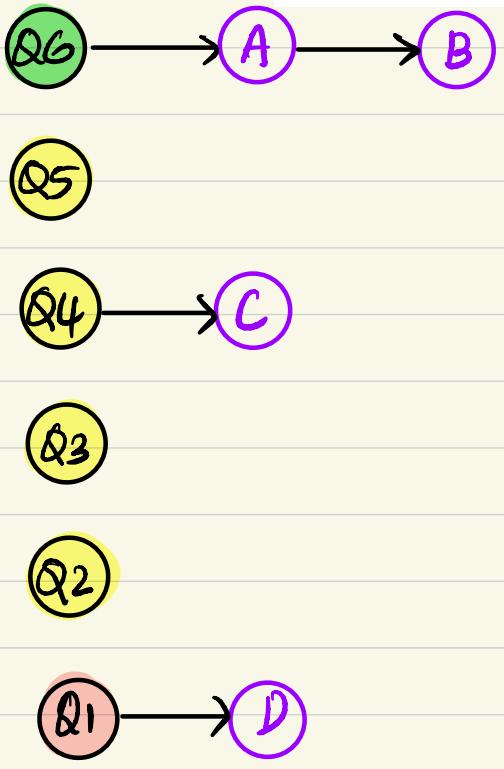
Rule 2: If priority (A) = priority (B), B runs in RR.

Suppose we have the following situation.

What happens if we maintain the same set of queues all the time?

The above rules ⇒ only A and B runs in RR, while C and D never even get to run until A and B finish.

How to rectify the above problem?
change the job priorities over time.



How to change Priority?

* MLFQ varies the priority of a job based on its observed behavior.

eg:- a job repeatedly relinquishes the CPU while waiting I/O
→ keep its priority high

a job uses the CPU intensively for long periods of time
→ reduce its priority.

How to know which jobs are CPU intensive? Define **allotment** to be the amount of time a job can spend at a given priority level before the scheduler reduces its priority. (let allotment = single time slice)

Rules for Priority adjustment:

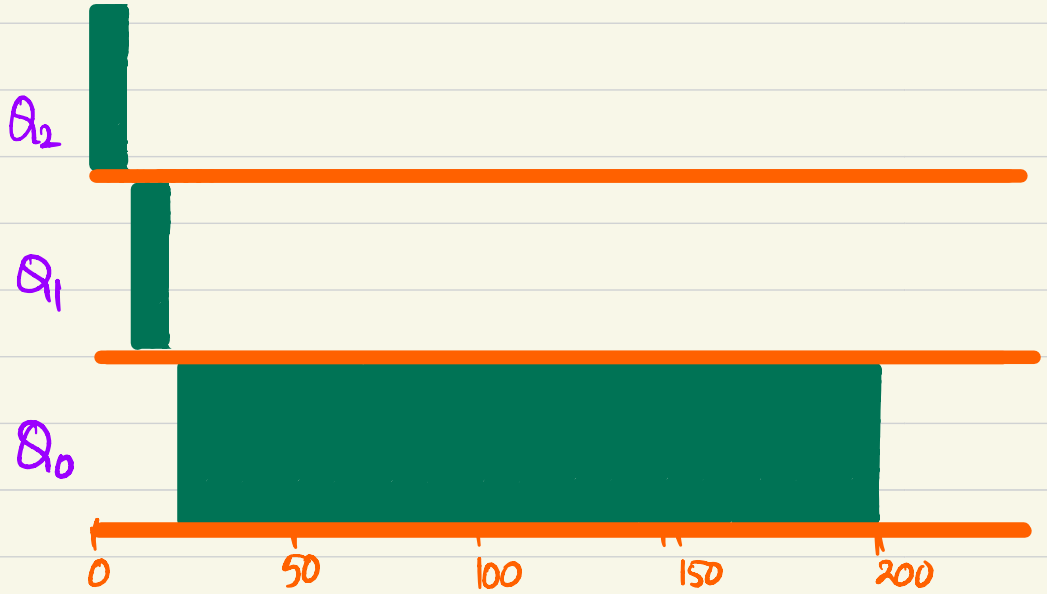
Rule 3: When a job enters the system, it is placed at the highest priority.

Rule 4a: If a job uses up its allotment, its priority is reduced to the queue below it.

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

In this manner, MLFQ approximates SJF (think, why?)

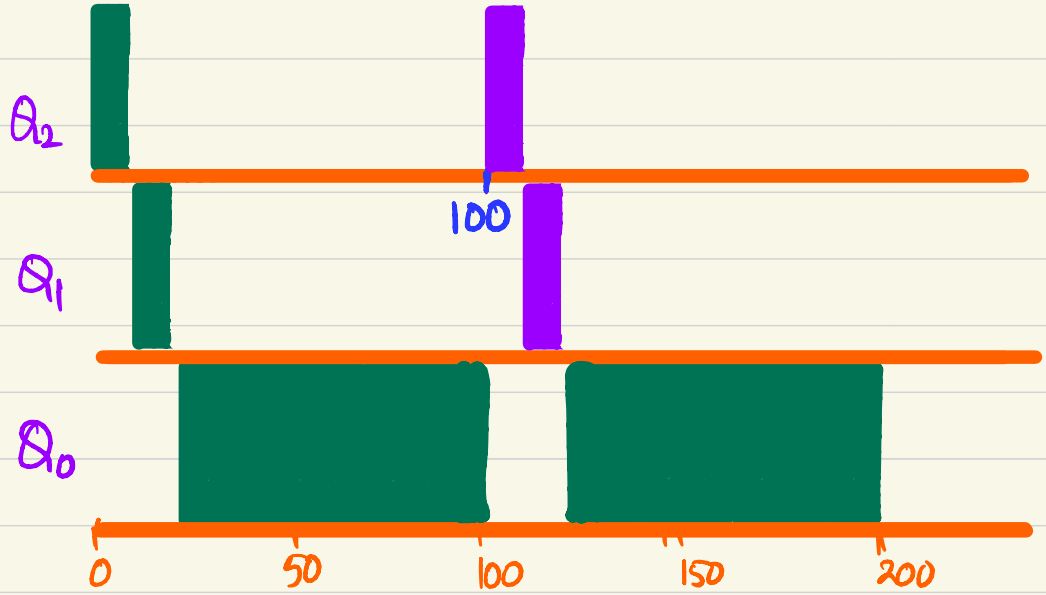
eg:- (1) A single long running job on a three-queue scheduler with time slice 10 ms.



eg:- (2) Job A → a long-running CPU intensive job

Job B → a short-running interactive job (20 ms runtime)

A has been running for sometime, and then B arrives at time T=100

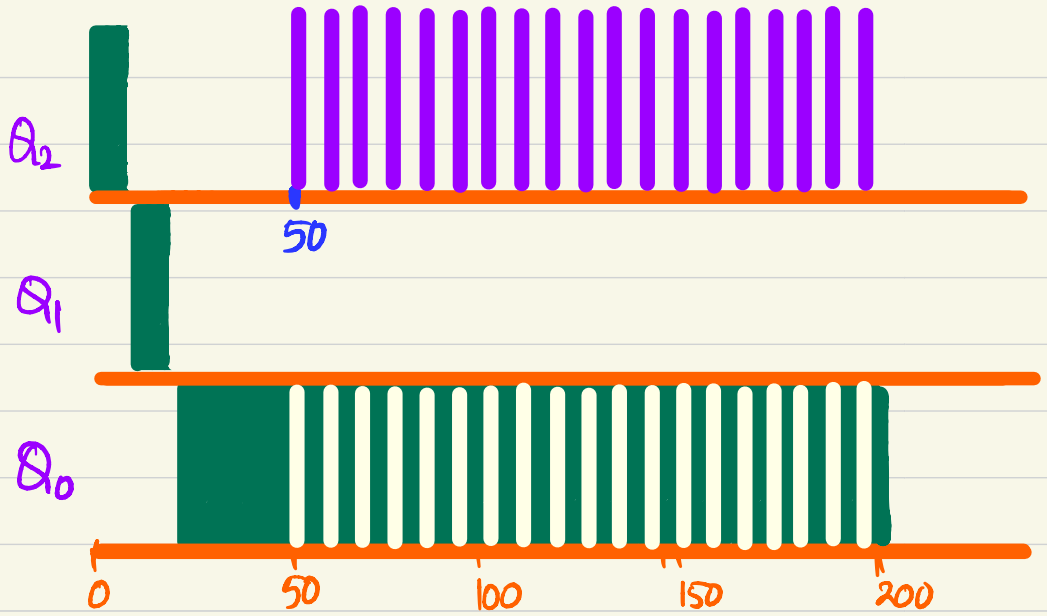


eg:-(3) with I/O

Job A → a long running job

Job B → an interactive job that needs the CPU only for 1ms before performing an I/O.

A has been running for sometime and B arrives at T=50.



Here, MLFQ keeps B at the highest priority, because B release the CPU before its allotment.

The above example illustrates that the MLFQ keeps an interactive job at the highest priority.

Problems with the basic MLFQ

* Starvation

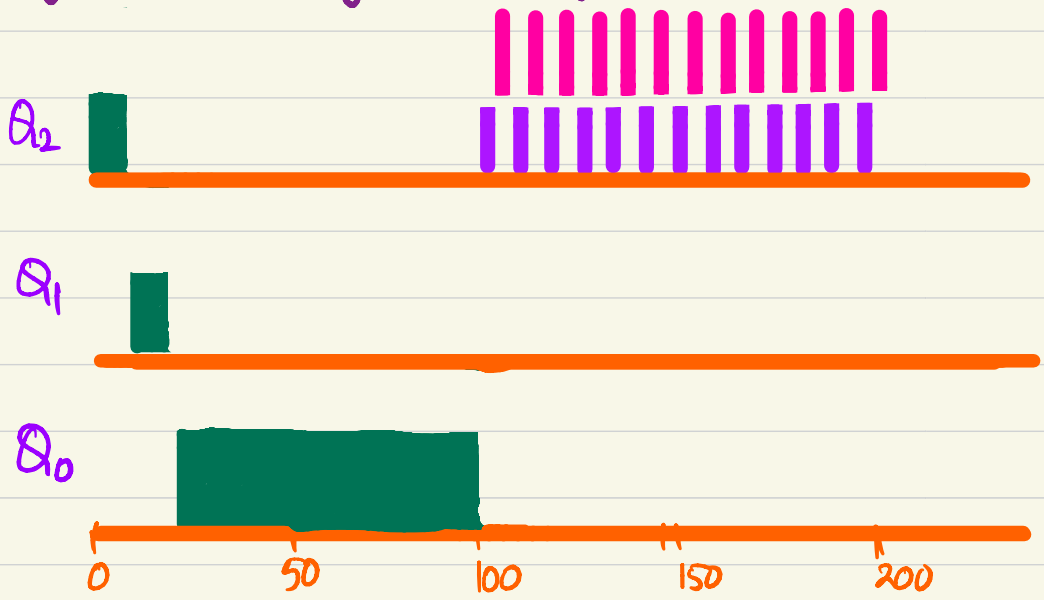
- if there are "too many" interactive jobs in the system
- long running jobs will never receive any CPU time.

* Game the scheduler
  - after running 99% of its allotment, issue an I/O operation, so that the job gain a higher percentage of CPU time.
* A program may change its behavior over time
  - CPU bound process ----> I/O bound process. eg:- video games (scheduler won't realize it)

How to prevent starving? → priority boost

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

eg:-(4) a long running job A with two short running interactive jobs B, C.



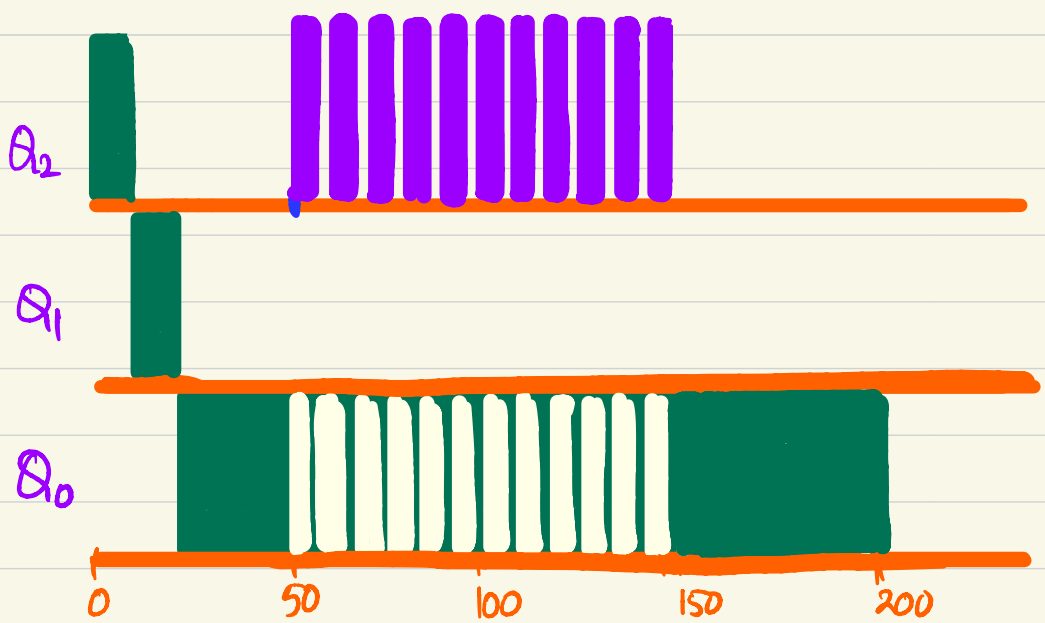
without priority boost

with priority boost

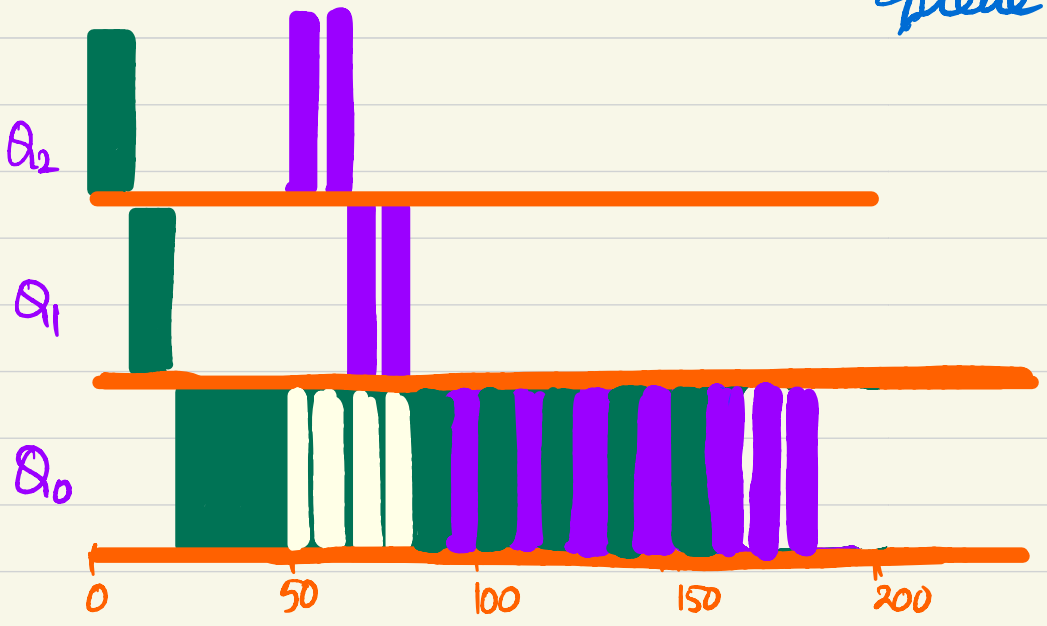
How to prevent gaming scheduler?

Rewrite Rules 4a and 4b to the following single rule.

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 the queue.



without gaming tolerance
(process staying at the top priority by issuing I/O just before allotment ends)



with game tolerance
(the same process cannot gain an unfair share of CPU here)

Tuning MLFQ and other issues

How to parameterize the scheduler?

↳ (1) How many queues should there be?

(2) How big should the time slice be per queue?

(3) What is the allotment?

(4) How often priority be boosted?

No easy answers for the above questions!

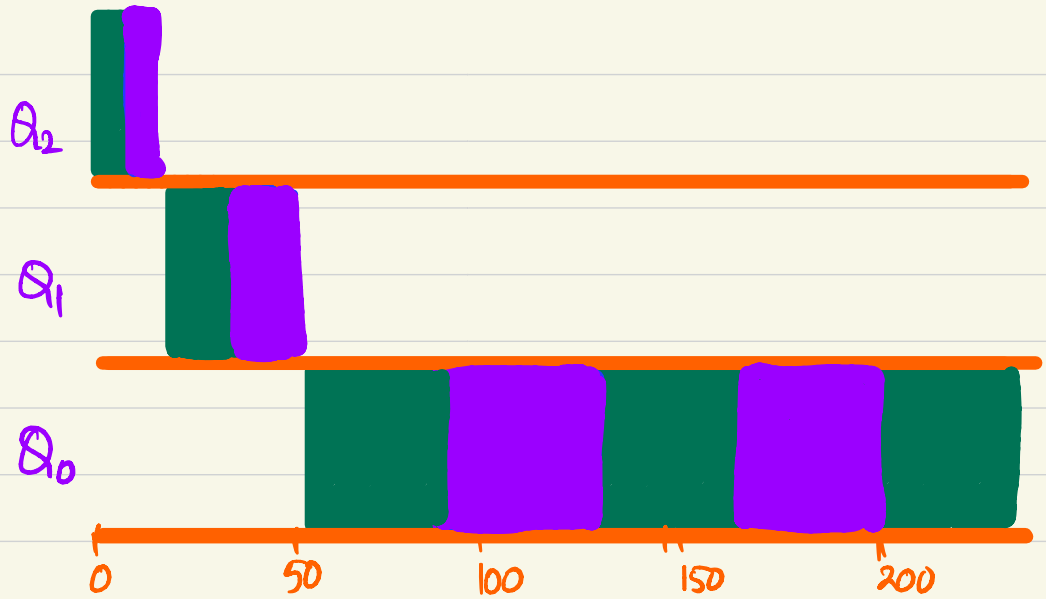
Most MLFQ variants allow for varying time-slice length across different queues.

In particular, usually high-priority queues → short time slices

eg:- 10 or fewer ms

low-priority queues → longer time slices.

eg:- 100 ms.



Eg:- 10 ms for the highest queue, 20 ms for the middle, 40 ms for the lowest.

Solaris, a unix based operating system (originally developed by Sun Microsystems in the early 1990s) use the following in its time sharing scheduling class (set of scheduling policies)

* 60 queues
* slowly increasing time-slice length
  - highest priority : 20 ms
  - lowest priority : a few 100 ms.
* priorities boosted around every 1 sec or so

Reference: Chapter 8 - OSTEP book