Scheduling: Introduction

CMPU 334 – Operating Systems
Jason Waterman

How to develop a scheduling policy

- How should we develop a basic framework for thinking about scheduling policies?
- What are the key assumptions?
- What metrics are important?
- What basic approaches have been used in the past?

Workload

Initial workload assumptions (we'll relax these assumptions later):

1. Each job runs for the **same amount of time**
2. All jobs **arrive** at the same time
3. Once started, each job runs to completion
4. All jobs only use the **CPU** (i.e., they don't perform no I/O)
5. The **run-time** of each job is known

Scheduling Metrics

- Performance metric: Turnaround time
  - The time at which **the job completes** minus the time at which **the job arrived** in the system

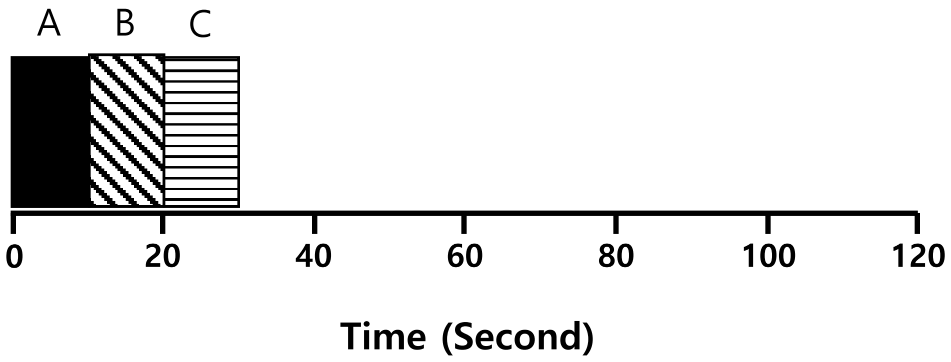
$$T_{turnaround} = T_{completion} - T_{arrival}$$

- Another metric is fairness (e.g., Jain's Fairness Index)
  - Maximum when all jobs receive the same share of CPU allocation

- Performance and fairness are often at odds in scheduling

First In, First Out (FIFO)

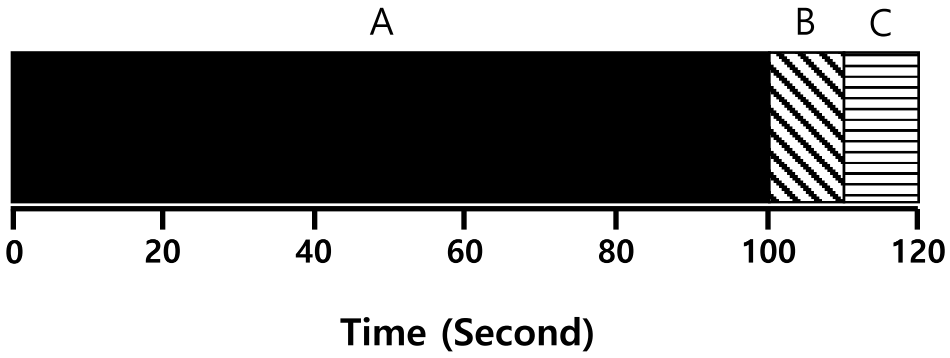
- First Come, First Served (FCFS)
  - Very simple and easy to implement
- Example:
  - A arrived just before B which arrived just before C
  - Each job runs for 10 seconds



$$Average\ turnaround\ time = \frac{10 + 20 + 30}{3} = 20\ sec$$

Why FIFO is not that great? – Convoy effect

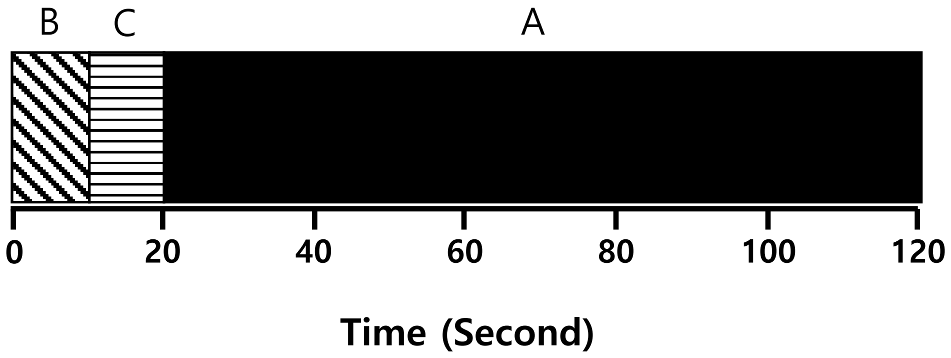
- Let’s relax assumption #1 (all jobs run for the same time)
  - Each job **no longer** runs for the same amount of time
- Example:
  - A arrived just before B which arrived just before C
  - A runs for 100 seconds, B and C run for 10 each



$$\boldsymbol{Average\ turnaround\ time = \frac{100 + 110 + 120}{3} = 110\ sec}$$

Shortest Job First (SJF)

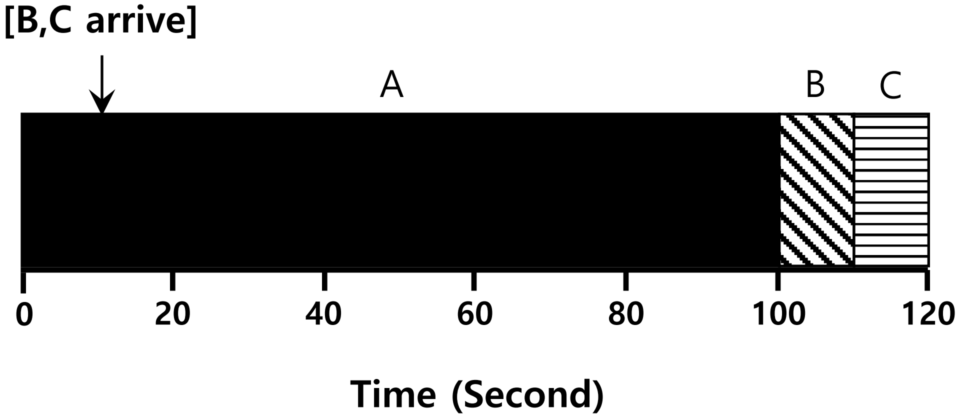
- Run the shortest job first, then the next shortest, and so on
  - Non-preemptive scheduler (no interrupting a running job)
- Example:
  - A arrived just before B which arrived just before C
  - A runs for 100 seconds, B and C run for 10 each



$$\boldsymbol{Average\ turnaround\ time = \frac{10+20+120}{3} = 50\ sec}$$

SJF with Late Arrivals from B and C

- Let's relax assumption #2 (all jobs arrive at the same time)
  - Jobs can now arrive at any time
- Example:
  - A arrives at t=0 and needs to run for 100 seconds
  - B and C arrive at t=10 and each need to run for 10 seconds



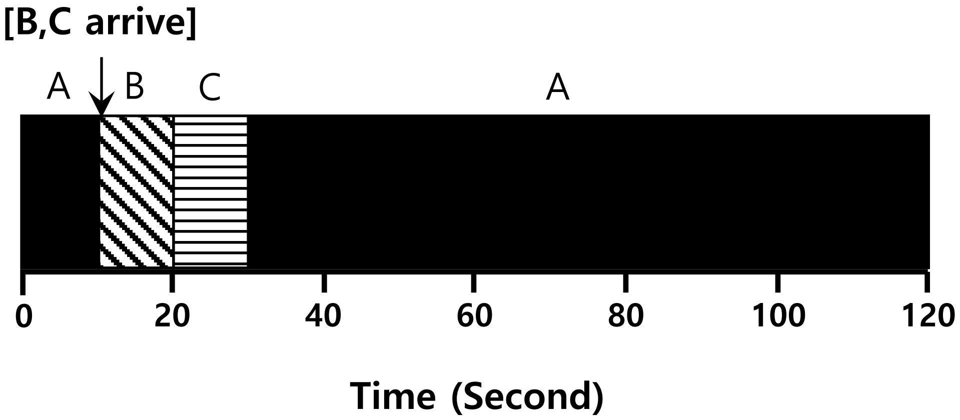
$$Average\ turnaround\ time = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33\ sec$$

Shortest Time-to-Completion First (STCF)

- Let's relax assumption #3 (once started, each job runs to completion)
- Add preemption to SJF
  - Also knows as Preemptive Shortest Job First (PSJF)
- When a new job enters the system:
  - Determine the time to complete the remaining jobs and new job
  - Schedule the job which has the least remaining time left
- Provably optimal with regard to minimizing turnaround time

Shortest Time-to-Completion First (STCF)

- Example:
  - A arrives at t=0 and needs to run for 100 seconds
  - B and C arrive at t=10 and each need to run for 10 seconds



$$Average\ turnaround\ time = \frac{(120-0)+(20-10)+(30-10)}{3} = 50\ sec$$

New scheduling metric: Response time

- The time from **when the job arrives** to the **first time it is scheduled**

$$T_{response} = T_{firstrun} - T_{arrival}$$

  - STCF and related disciplines are not particularly good for response time

How can we build a scheduler that is
sensitive to response time?

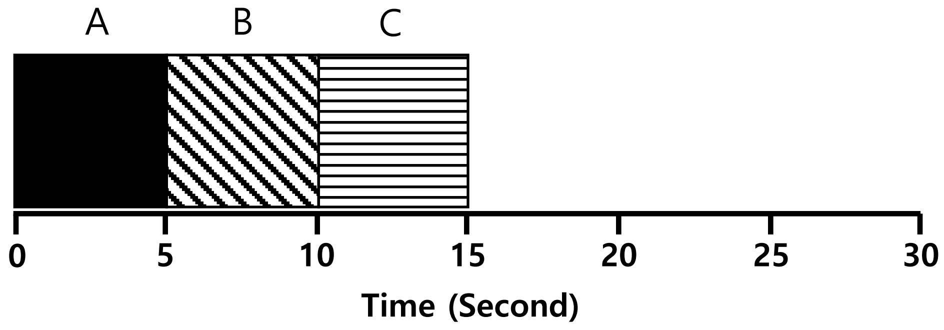
Round Robin (RR) Scheduling

- Time slicing Scheduling
  - Run a job for a time slice and then switch to the next job in the **run queue** until the jobs are finished
    - Time slice is sometimes called a scheduling quantum
  - It repeatedly does so until the jobs are finished
  - The length of a time slice must be *a multiple of* the timer-interrupt period

RR is fair, but performs poorly on metrics such as turnaround time

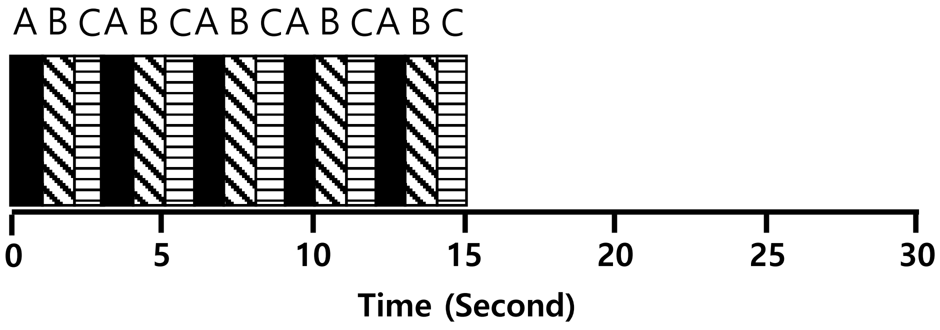
RR Scheduling Example

- A, B and C arrive at the same time
- They each wish to run for 5 seconds



SJF (Bad for Response Time)

$$T_{average\ response} = \frac{0+5+10}{3} = 5sec$$



RR with a time-slice of 1sec (Good for Response Time)

$$T_{average\ response} = \frac{0+1+2}{3} = 1sec$$

The length of the time slice is critical

- The shorter time slice
  - Better response time
  - The cost of context switching will dominate overall performance

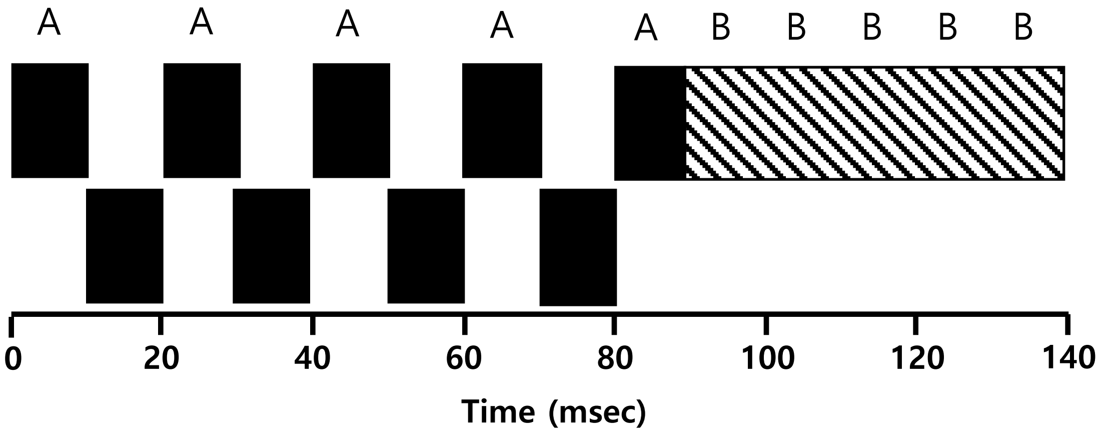
- The longer time slice
  - Amortize the cost of switching
  - Worse response time

Deciding on the length of the time slice presents a trade-off to a system designer

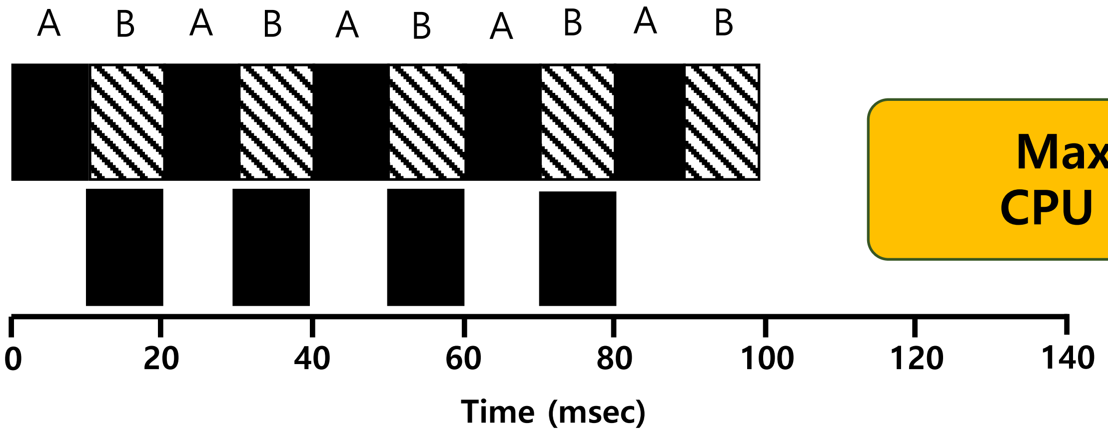
Incorporating I/O

- Let's relax assumption #4 (all jobs only use the **CPU)**
  - Jobs can now perform I/O
- Example:
  - Jobs **A** and **B** need 50ms of CPU time each
  - **A** runs for 10ms and then issues an I/O request
    - I/Os each take 10ms
  - **B** simply uses the CPU for 50ms and performs no I/O
  - The scheduler runs **A** first, then **B** after

Incorporating I/O (Cont.)



Poor Use of Resources



Overlap Allows Better Use of Resources

Maximize the CPU utilization

Incorporating I/O (Cont.)

- When a job initiates an I/O request
  - The job is blocked waiting for I/O completion
  - The scheduler should schedule another job on the CPU

- When the I/O completes
  - An interrupt is raised
  - The OS moves the process from blocked back to the ready state

What’s Next?

- Remove our final assumption: the scheduler knows the length of each job
- The OS usually knows very little about the length of each job

- How can we behave like SJF/STCF without such knowledge?
- How can we be fair and also have good response time?

Multi-Level Feedback Queue (MLFQ)

- A scheduler that learns from the past to predict the future
- Objective:
  - Optimize **turnaround time** → Run shorter jobs first
  - Minimize **response time** without *a priori knowledge of job length*

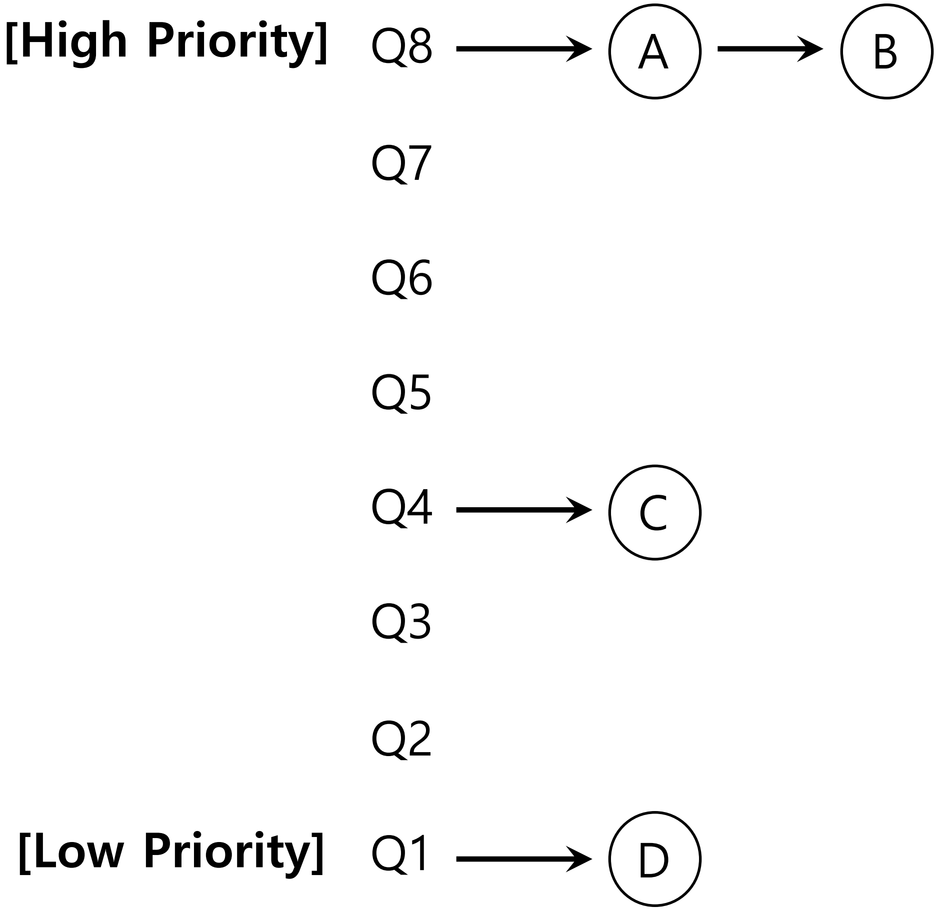
MLFQ: Basic Rules

- MLFQ has a number of distinct **queues**
  - Each queues is assigned a different priority level

- A job that is ready to run is on a single queue
  - I.e., a job can be in only one queue at any given time
  - A job **on the highest priority queue** is chosen to run
  - Use round-robin scheduling among jobs in the same queue

Rule 1: If Priority(A) > Priority(B), A runs (B doesn't)
Rule 2: If Priority(A) = Priority(B), A & B run in RR

MLFQ Example



[High Priority] Q8 → A → B

Q7

Q6

Q5

Q4 → C

Q3

Q2

[Low Priority] Q1 → D

MLFQ: Basic Rules (Cont.)

- MLFQ varies the priority of a job based on its **observed behavior**
- Example:
  - A job repeatedly relinquishes the CPU while waiting for I/O
    - Keep its priority high
    - When it runs, it doesn’t run for very long
  - A job uses the CPU intensively for long periods of time
    - Reduce its priority

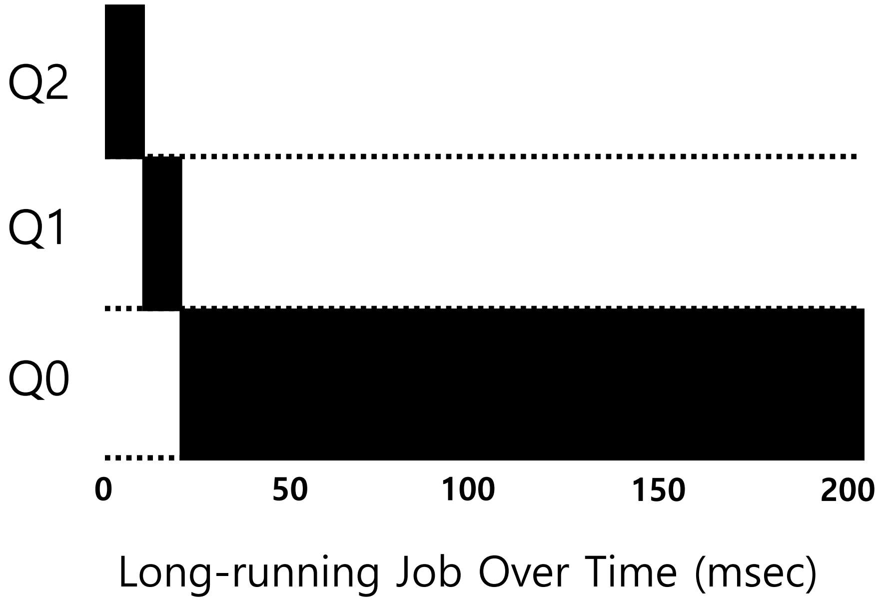
MLFQ: How to Change Priority

- MLFQ priority adjustment algorithm:
  - **Rule 3**: When a job enters the system, it is placed in the highest priority queue
  - **Rule 4a**: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down a queue level)
  - **Rule 4b**: If a job gives up the CPU before the time slice is up, it stays at the same priority level

In this manner, MLFQ approximates SJF

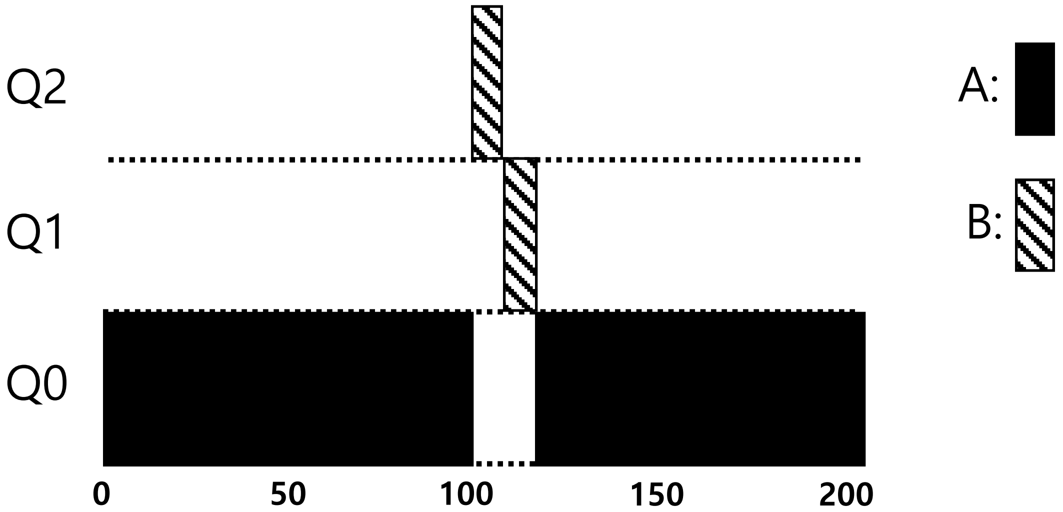
Example 1: A Single Long-Running Job

- A three-queue scheduler with time slice 10ms



Example 2: Along Comes a Short Job

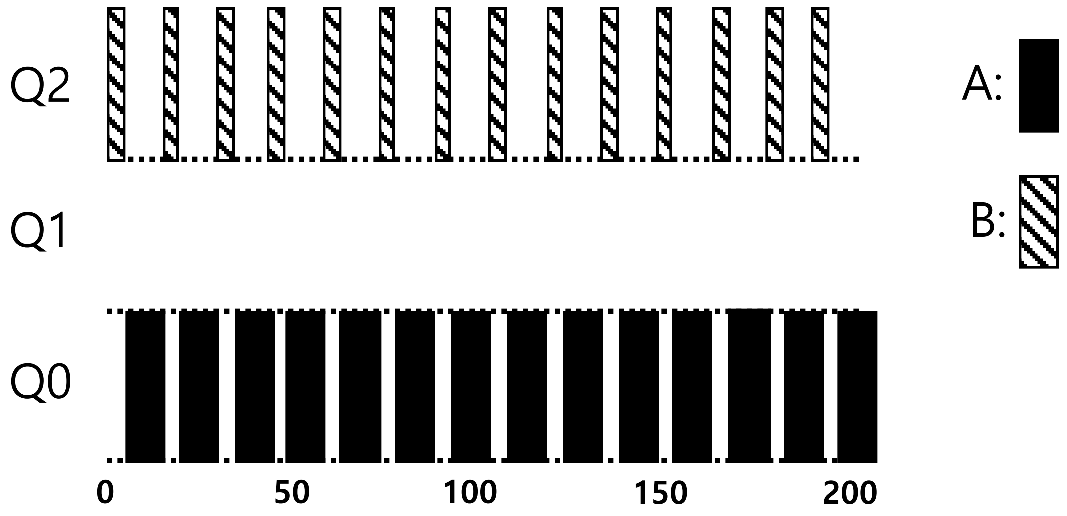
- Assumption:
  - **Job A**: A long-running CPU-intensive job
  - **Job B**: A short-running interactive job (20ms runtime)
  - A has been running for some time, and then B arrives at time T=100.



Along Comes An Interactive Job (msec)

Example 3: What About I/O?

- Assumption:
  - **Job A**: A long-running CPU-intensive job
  - **Job B**: An interactive job that need the CPU only for 1ms before performing an I/O



A Mixed I/O-intensive and CPU-intensive Workload (msec)

The MLFQ approach keeps an interactive job at the highest priority

Problems with the Basic MLFQ

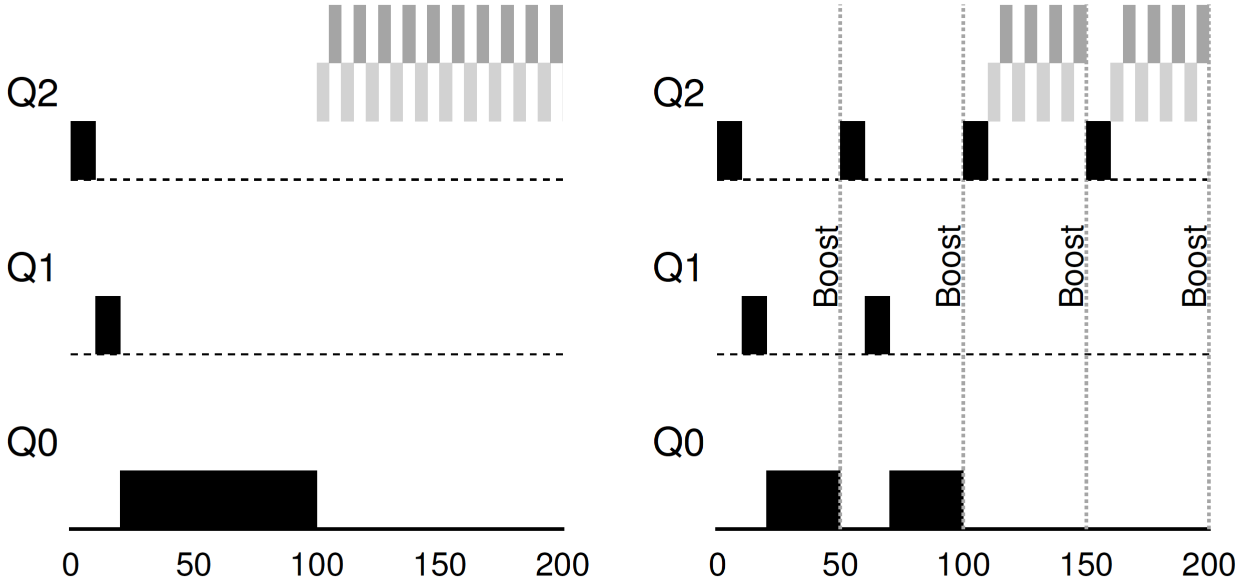
- Starvation
  - If there are “too many” interactive jobs in the system
  - Lon-running jobs will never receive any CPU time

- Game the scheduler
  - After running 99% of a time slice, issue an I/O operation (e.g., read(0))
  - The job gains a higher percentage of CPU time

- A program may change its behavior over time
  - CPU bound process → I/O bound process

The Priority Boost

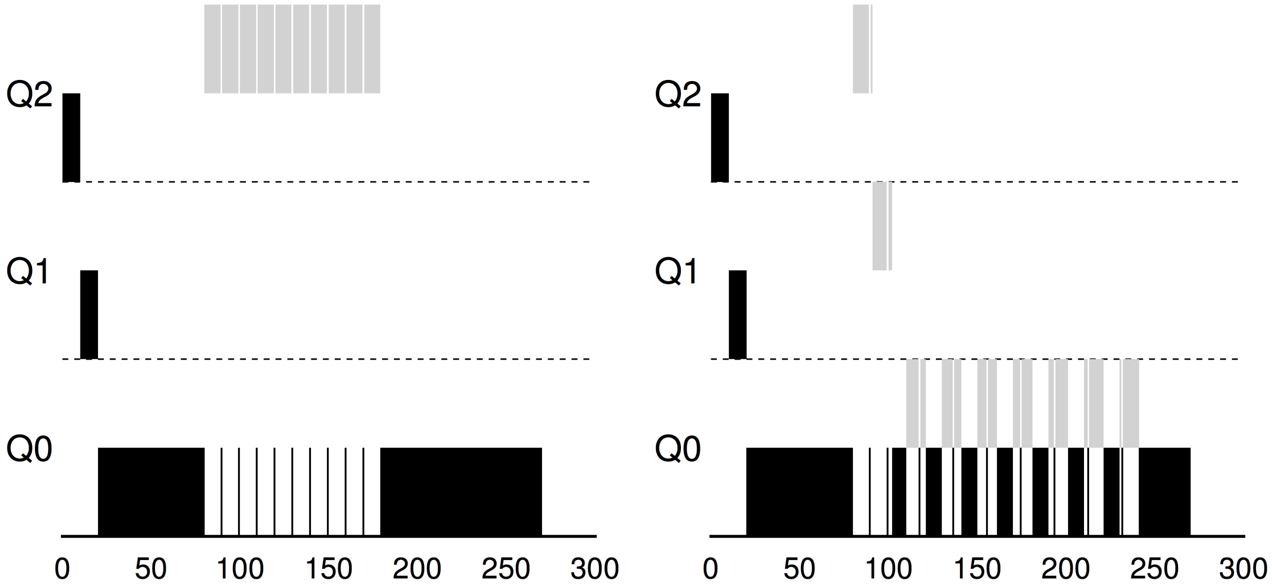
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue
  - Example:
    - A long-running job(A) with two short-running interactive job(B, C)



Without(Left) and With(Right) Priority Boost

Better Accounting

- How to prevent gaming of our scheduler?
- Solution:
  - **Rule 4** (Rewrite Rules 4a and 4b): 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)

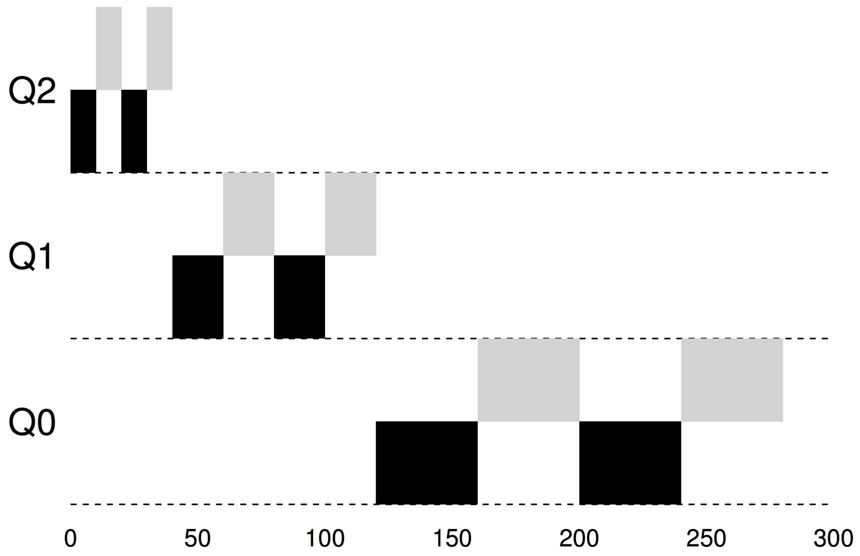


Without(Left) and With(Right) Gaming Tolerance

Tuning MLFQ And Other Issues

Lower Priority, Longer Quanta

- The high-priority queues → Short time slices
  - E.g., 10 or fewer milliseconds
- The Low-priority queue → Longer time slices
  - E.g., 100 milliseconds



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

The Solaris MLFQ implementation

- For the Time-Sharing scheduling class (TS)
  - 60 Queues
  - Slowly increasing time-slice length
    - The highest priority: 20 msec
    - The lowest priority: A few hundred milliseconds
  - Priorities boosted around every 1 second or so

MLFQ: 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