CS 326: Operating Systems

Multi-Level Feedback Queues

Lecture 11

Today's Schedule

- Building an Ideal Scheduler
- Priority-Based Scheduling
  - Multi-Level Queues
- Multi-Level Feedback Queues
- Scheduling Domains
- Completely Fair Scheduling

Today's Schedule

- **Building an Ideal Scheduler**
- Priority-Based Scheduling
  - Multi-Level Queues
- Multi-Level Feedback Queues
- Scheduling Domains
- Completely Fair Scheduling

CPU Scheduling

- We've discussed several CPU scheduling algorithms
- Each algorithm has its own set of strengths and weaknesses
  - Like anything in computer science, scheduling is all about managing trade-offs
- By designing our scheduler carefully, we can provide good performance for a variety of workloads

The Ideal Scheduler

- In a perfect world, we'd have a scheduler with:
  - Quick turnaround times
  - Quick response times
  - Free beer
- Turnaround: amount of time from process arrival to process completion
- Response: how quickly a process starts running after it arrives

Multi-Level Feedback Queue

- The MLFQ (and variants thereof) is one of the most commonly-used scheduling algorithms
  - macOS, Windows, Solaris, FreeBSD
    - *With modifications
  - Linux: 1991 – 2001
- Builds on the idea of **priority scheduling**: different tasks will run with different priorities

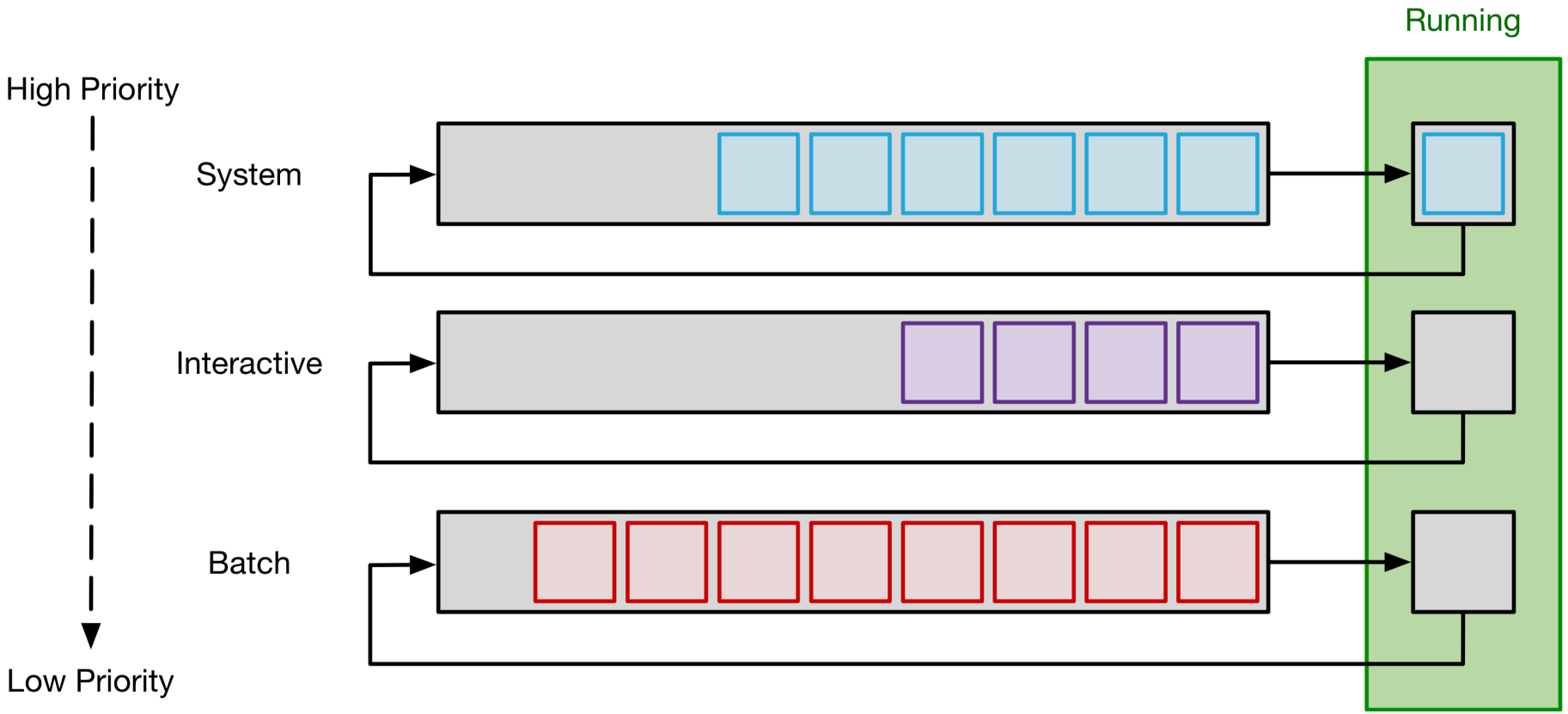
Today's Schedule

- Building an Ideal Scheduler
- **Priority-Based Scheduling**
  - Multi-Level Queues
- Multi-Level Feedback Queues
- Scheduling Domains
- Completely Fair Scheduling

Priority Scheduling

- Why use priority scheduling as a base for more complicated algorithms?
- Not all tasks have the same resource requirements:
  - Your browser: needs CPU frequently
  - Time Sync Daemon: needs CPU rarely
- By prioritizing tasks effectively into multiple **queues**, we can make sure they get the resources they need
  - Everybody's happy!

Multi-Level Queue



MLQ: Algorithms

- Many implementations of this multi-queue approach use different scheduling algorithms **per queue**
- System priority? Use FIFO
  - Must be empty before moving to the interactive queue
- Interactive? Use shortest job first (SJF)
- Batch queue? Use round robin (RR)

Dynamic Priority

- The only issue with assigning priorities in a fixed manner is that priorities often change
  - Plus, if we ask applications what priority to run at they'll all want to be #1
- Maybe you minimize your web browser… Should it still be a high priority?
- **Bursty** workloads: perhaps a task sits idle 70% of the time but needs lots of resources 30% of the time
- We need to allow priorities to change dynamically

Implementing Dynamic Priorities

- Rather than asking programs for their priority, we'll assign them automatically instead
- Let's have a thought experiment: How can we assign priorities fairly and efficiently?
- …

Implementing Dynamic Priorities

- Start the process in a queue based on its resource requirements and move it based on usage patterns
  - Reactive scheduling
- Processes that use more CPU will be moved to lower priorities over time
  - Process **aging**
- Stop using so much CPU? Priority will gradually be boosted

Reactive Algorithms

- A **reactive** algorithm doesn't really sound that great
  - It just takes what it has seen in the past into account!
  - A far cry from "intelligence"
- However, **many** patterns in computer science (and in the real world!) repeat themselves
- You can… adequately… predict the future by looking at the past

Today's Schedule

- Building an Ideal Scheduler
- Priority-Based Scheduling
  - Multi-Level Queues
- **Multi-Level Feedback Queues**
- Scheduling Domains
- Completely Fair Scheduling

Multi-Level Feedback Queue

- MLFQ is designed to be a good scheduler for systems that have a variety of process types
  - **Multimode systems**
- Goals of the algorithm:
  1. Favor short jobs
  2. Favor I/O bound jobs
  3. Split processes into queues based on usage needs

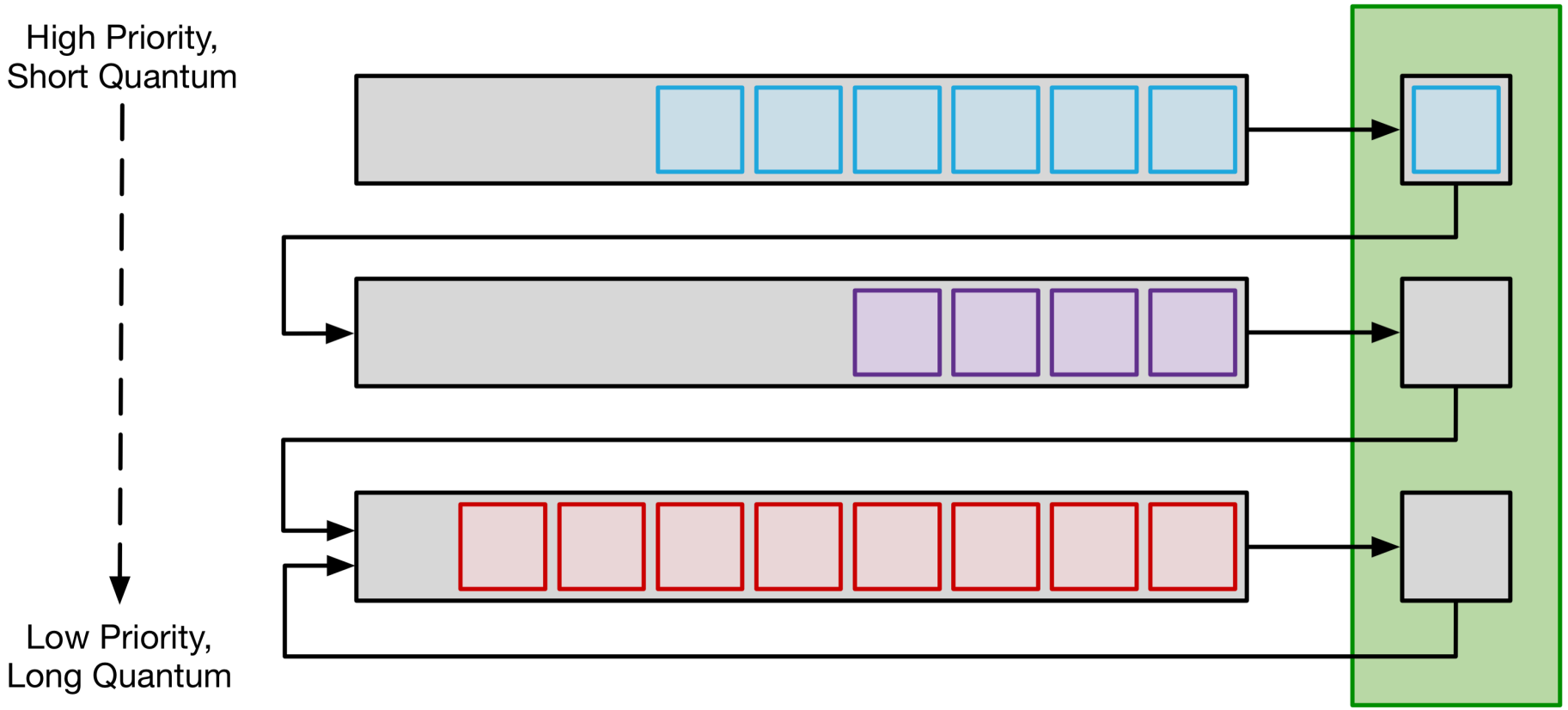
Algorithm [1/2]

- Processes enter at the highest priority queue
- When their turn comes, the process runs and does one of the following:
  1. Finishes execution
  2. Relinquishes control of the CPU
  3. Gets preempted by the OS
- When (2) happens, the process returns to the end of its current queue
- When (3) happens, the process is moved to the next lower priority queue

Algorithm [2/2]

- Once the process reaches the last queue, or *base level queue*, it stays there
  - Usually scheduled w/ Round Robin algorithm
- Long-running background tasks can happily stay in the base level queue
- If the process blocks for I/O, then it will be promoted to one queue higher
  - Allows processes to “escape” to a higher priority

MLFQ



*If a process does not use its entire quantum, it remains in its current queue.

Adding Priorities

- We have different priorities based on queues, but we can also assign processes a priority level
  - Called **nice** level on Unix systems
- Using these priorities, we can iterate over the queue and:
  - Run processes with a higher priority first
  - If all priorities are the same, we fall back to round robin

nice

- Nice levels range from -20 (highest priority) to 19 (lowest priority)
  - Why? Arbitrary
  - Why does positive = low priority?
    - Essentially ”adding” CPU usage to a process
- The default nice level is 0
- As a regular user, you can only decrease your process priority

nice your jobs

- Let’s be nice to the other users on our machine:
  - `nice -n 19 COMMAND`
  - `renice +19 PID`
- Easy way to see the effects: run `stress` and have different nice levels for each child process
- Doesn’t seem that important these days. In the past, multi-user systems were much more common
- Still somewhat true today: we can ssh into any of the machines in the department and use them remotely

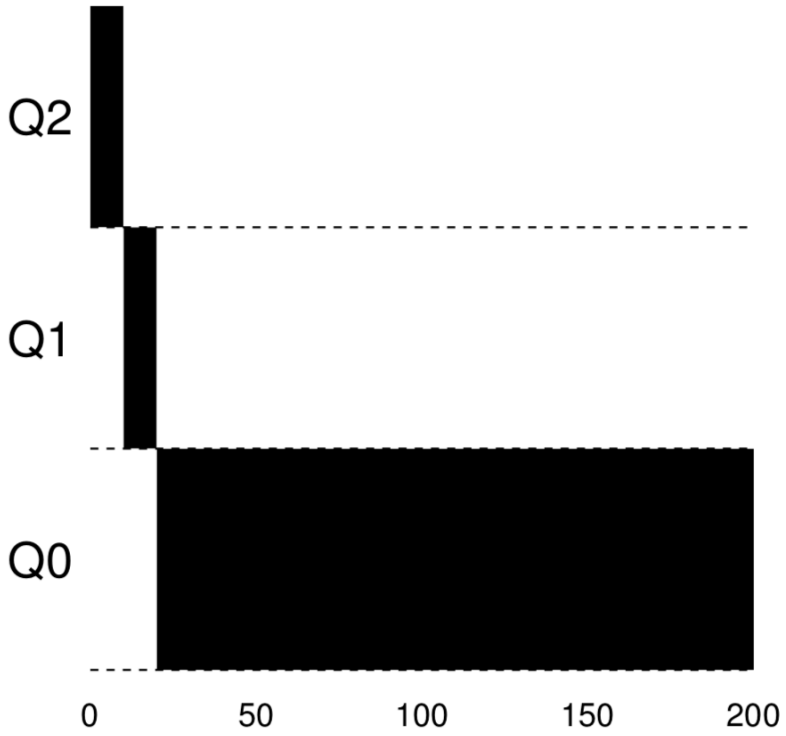
Tweaking Priorities

- The **nice** command can also be useful in tweaking your system to get better performance
- Perhaps you have a web server running to host your band's latest song…
  - …but don't want lots of web users downloading it to interrupt your minecraft session
- **nice** the web server process (apache, nginx, etc)

MLFQ CPU Usage Profiles

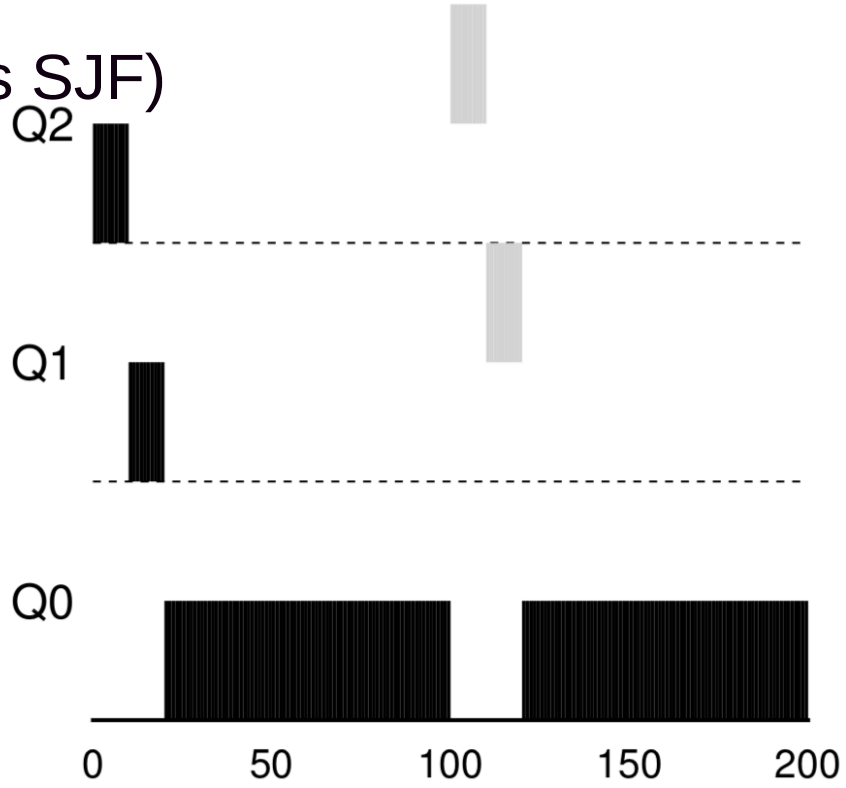
- Let’s look at some process usage profiles under MLFQ
- We have different job types:
  - Long running
  - Short
  - I/O-bound

A Long-Running Job

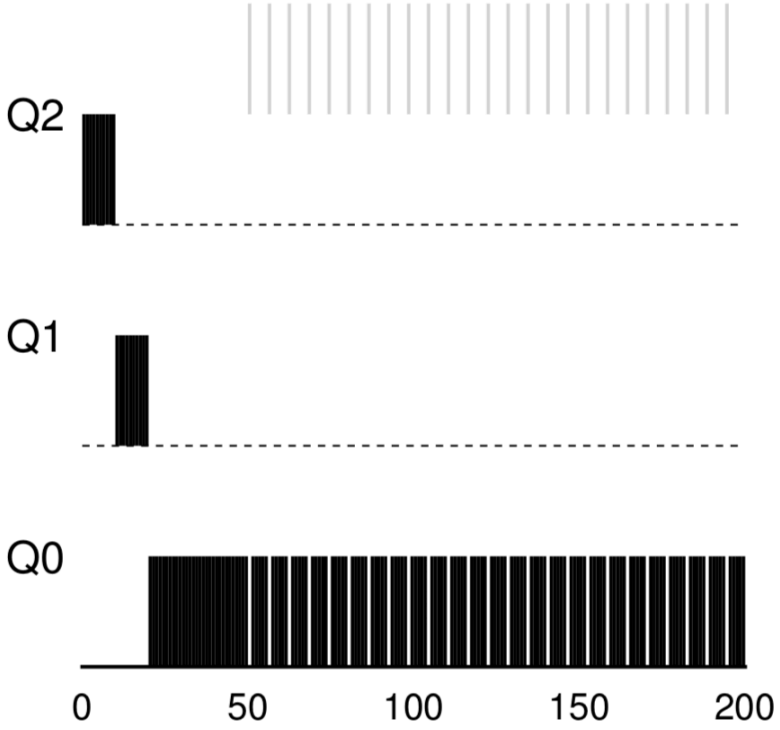


Long-Running + Interactive Job

(Approximates SJF)



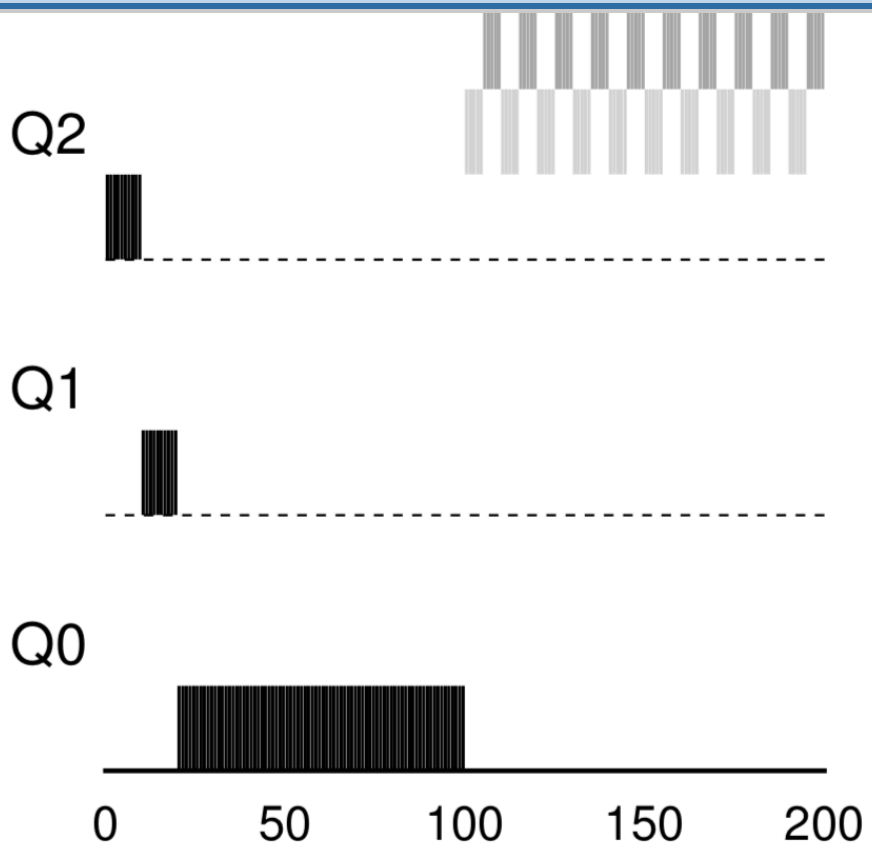
Heavy I/O



Starvation

- One major weakness of MLFQ is **starvation**
- If there are too many processes in general, long-running processes may never get to run
- This can happen from a busy interactive system
- Or, you can simulate this situation with a fork bomb
  - Don’t run this in your terminal:
    - `:(){ :|: & };:`
  - Cripples machine by starting processes that start more processes, and so on… how to do in C?

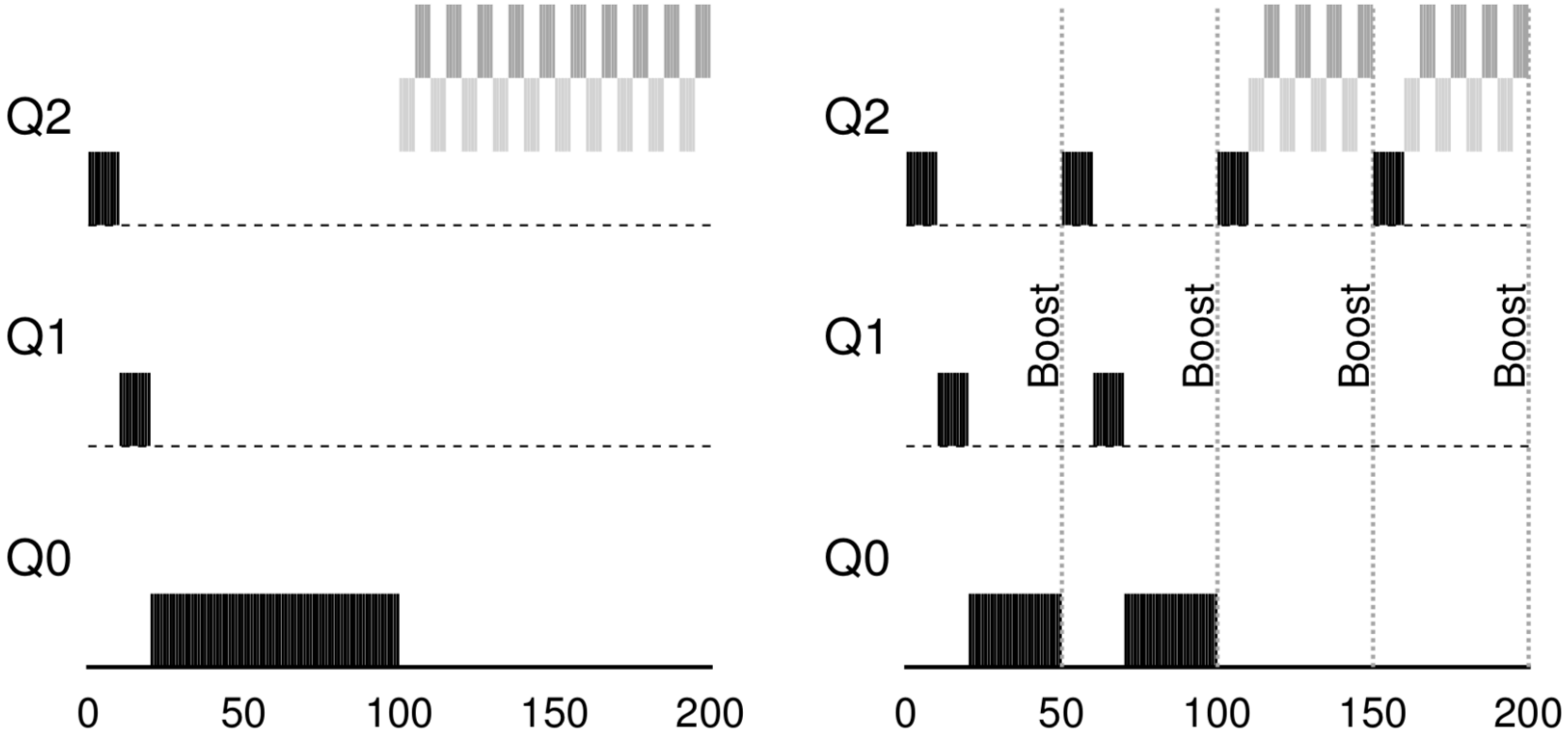
Starvation



Avoiding Starvation

- To deal with starving processes, the scheduler implements a periodic **priority boost**
- Every so often, the priority of all running processes is increased
- This prevents starving processes and also helps cope with processes that have varying usage patterns
  - If a background process has become more interactive, it gets a chance to run in the interactive queue

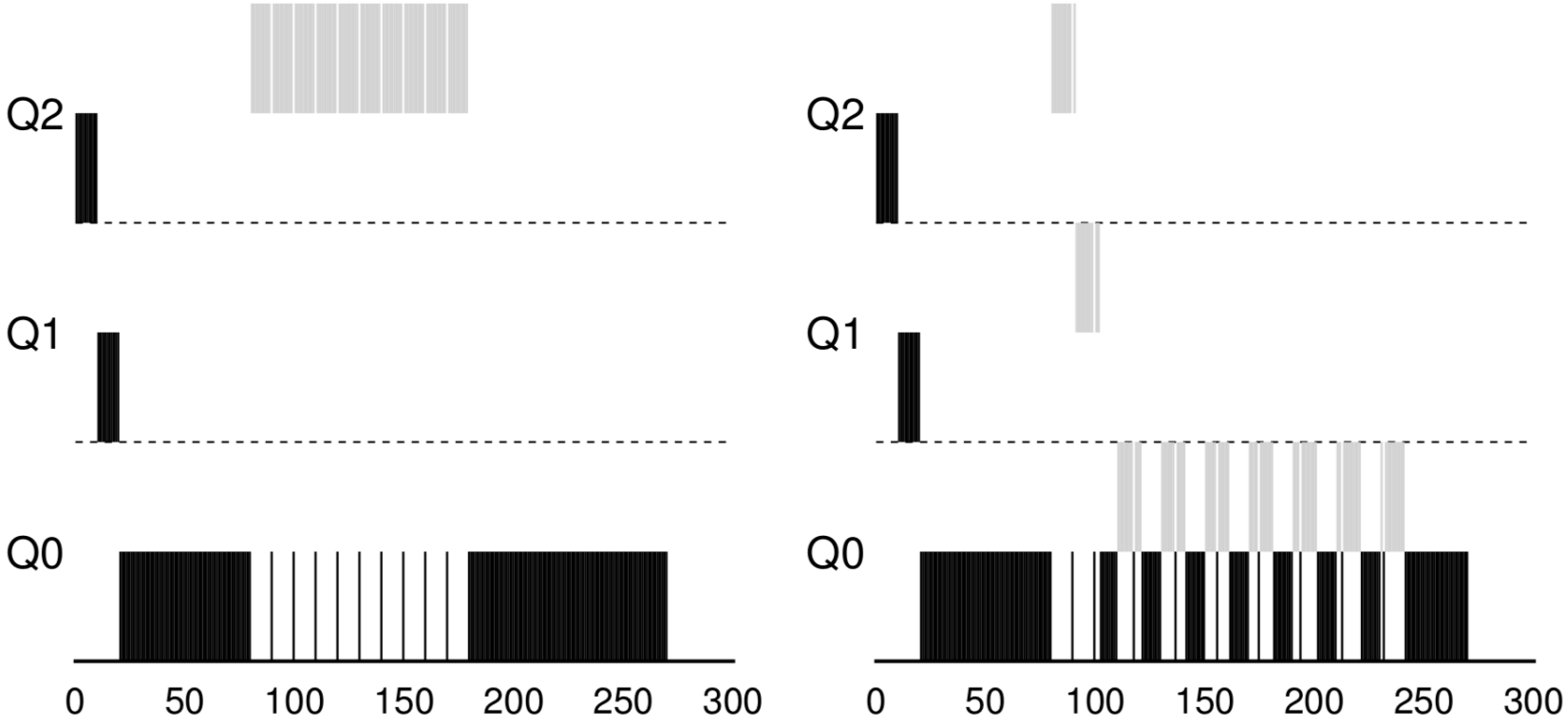
Before and After Boosting



Tricking MLFQ

- You can also “game” MLFQ by stopping your process **just before** its time quantum is up
  - Great, you get to stay in your current queue!
- To deal with this, we can record the **total** amount of time the CPU spends in each queue
  - Spending too much time (configurable threshold) causes your process to be moved to the next queue

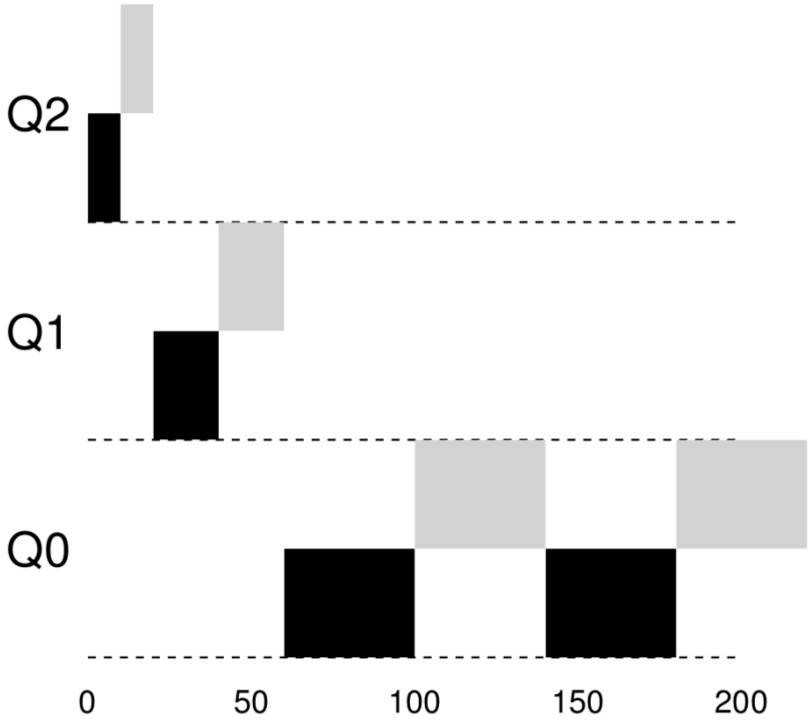
Dealing with Trickery



Process Accounting

- To help cope with these edge cases, the scheduler records total CPU time across all queues
- The more time spent on the CPU, the lower the queue your process will be placed in
- However, the lower the priority queue, the longer the scheduling quantum

CPU Time Accounting



MLFQ Goals

- MLFQ is tasked with improving both response time and turnaround time
- Response time: processes will always be considered "short running" when submitted
  - They start running quickly
- Turnaround: we avoid starvation and periodically **boost** priorities to cope with changing usage patterns
- We can have our cake and eat it too! (kinda…)

Today's Schedule

- Building an Ideal Scheduler
- Priority-Based Scheduling
  - Multi-Level Queues
- Multi-Level Feedback Queues
- **Scheduling Domains**
- Completely Fair Scheduling

Scheduling Domains [1/2]

- Unfortunately, we can't always assume there is just one CPU
- These days, we have to cope with:
  - Cache levels
  - Hyperthreading
  - Multi-core
  - Multi-processor
  - NUMA Architectures

Scheduling Domains [2/2]

- To deal with this, Linux supports **scheduling domains**
- Scheduling domains allow the various “execution units” (CPUs, hardware threads, cores, etc) to be placed in a hierarchy
  - Some are less desirable than others: I’d prefer a real core over a hyperthread
  - Natural hierarchy: one core, two logical threads

Migration

- Process migration takes these scheduling domains into account
- Example: cores on a single CPU may have separate L1 cache and unified (shared) L2 cache
  - Favor: **pinning** to a single core
  - Next best: scheduling on a different core, same CPU
  - Worst: migrating to a different CPU

Today's Schedule

- Building an Ideal Scheduler
- Priority-Based Scheduling
  - Multi-Level Queues
- Multi-Level Feedback Queues
- Scheduling Domains
- **Completely Fair Scheduling**

Completely Fair Scheduling

- As I mentioned, the Linux kernel does not use MLFQ
- Instead, completely fair scheduling (CFS) attempts to, well… be completely fair.
- In an **ideal** world, we could actually assign part of the CPU power to processes:
  - A gets 50%, B gets 25%, and C gets 25%, and they all run at the exact same time
  - Awesome! Not physically possible though

CFS

- Executes processes in a round robin fashion but **without** a fixed interrupt/time quantum
- Each process gets a **time slice**
  - 1ms or more
  - Chosen based on a **maximum execution time** (how long the process would run on an “ideal” processor)
- CPU usage is tracked per process
- Processes are inserted into a red-black tree and sorted by usage

CFS: Dividing up Time

- Let's assume a maximum execution time of 10ms
- We run 5 processes
- Each process gets 10 / 5 = 2 ms of CPU time
- To deal with **nice** values, we weight the processes
  - Pretend that they've already run for some time when they really haven't
- And that, my friends, is how scheduling works on modern OS

References

Great scheduling resource:

https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html

CFS:

https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

Scheduling figures from this lecture:Operating Systems: Three Easy Pieces

http://pages.cs.wisc.edu/~remzi/OSTEP/