CPU Virtualization: Advanced Scheduling

CS 571: Operating Systems (Spring 2022)

Lecture 3

Yue Cheng

Announcement

- Picking project due by 11:59pm this Friday
- We will have some time left for project discussion today

Advanced CPU Scheduling: Outline

- Scheduling algorithms
  - First In, First Out (FIFO)
  - Shortest Job First (SFJ)
  - Shortest Time-to-Completion First (STCF)
  - Round Robin (RR)
  - Priority
  - Multi-Level Feedback Queue (MLFQ)
  - Linux Completely Fair Scheduler (CFS)
  - Smarter function scheduler (SFS)

Workload Assumptions

1. Each job runs for the same amount of time

2. All jobs arrive at the same time

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

4. The run-time of each job is known

Workload Assumptions

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~All jobs only use the CPU (no I/O)~~
4. ~~The run-time of each job is known~~

Priority-Based Scheduling

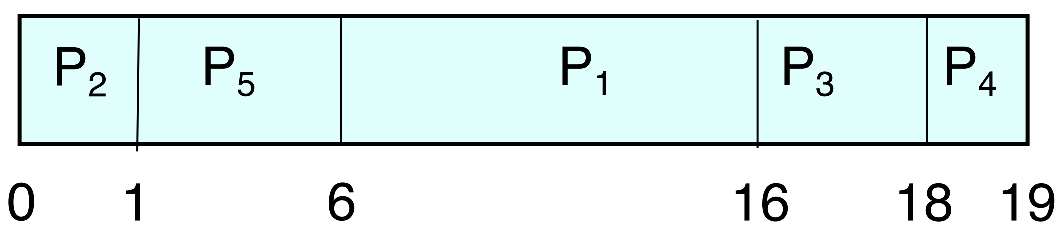
Priority-Based Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - We assume: smallest integer ≡ highest priority
  - Preemptive
  - Non-preemptive

Example for Priority-Based Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2

Priority-Based Scheduling (cont.)

- Priority Assignment
  - Internal factors: timing constraints, memory requirements, the ratio of average I/O burst to average CPU burst …
  - External factors: Importance of the process, financial considerations, hierarchy among users …

- Problem: Indefinite blocking (or starvation) – low priority processes may never execute

- One solution: Aging
  - As time progresses increase the priority of the processes that wait in the system for a long time

Multi-Level Feedback Queue (MLFQ)

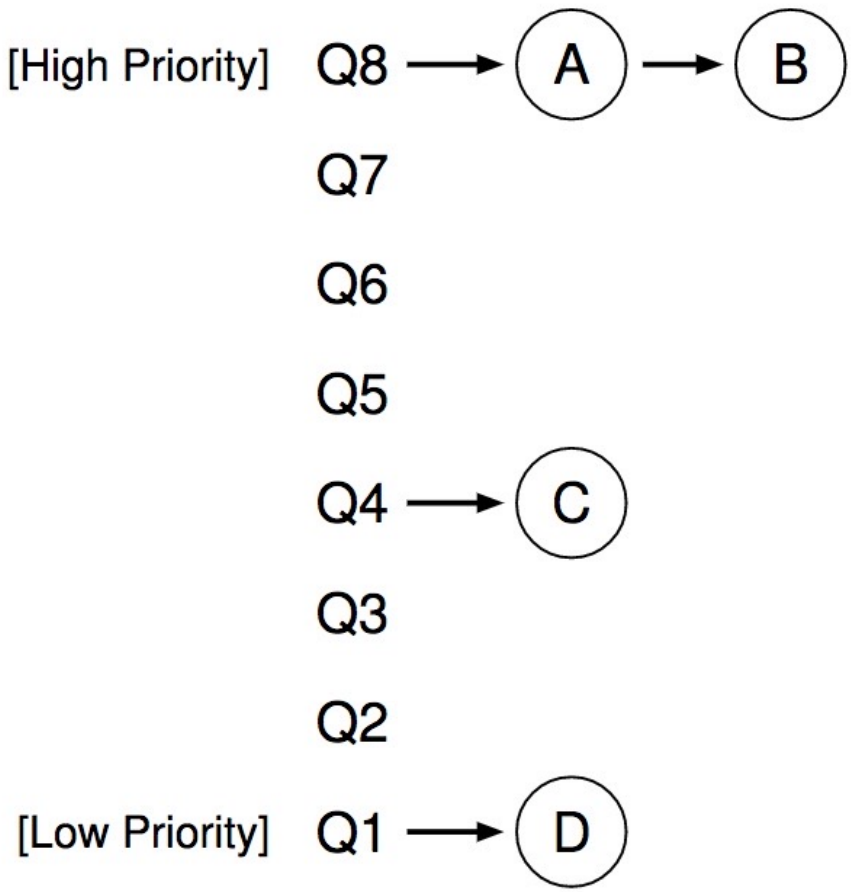
Multi-Level Feedback Queue (MLFQ)

- Goals of MLFQ
  - Optimize turnaround time
    - In reality, SJF does not work since OS does not know how long a process will run
  - Minimize response time
    - Unfortunately, RR is really bad on optimizing turnaround time

MLFQ: Basics

- MLFQ maintains a number of queues (multi-level queue)
    - Each assigned a different priority level
    - Priority decides which process should run at a given time

MLFQ Example



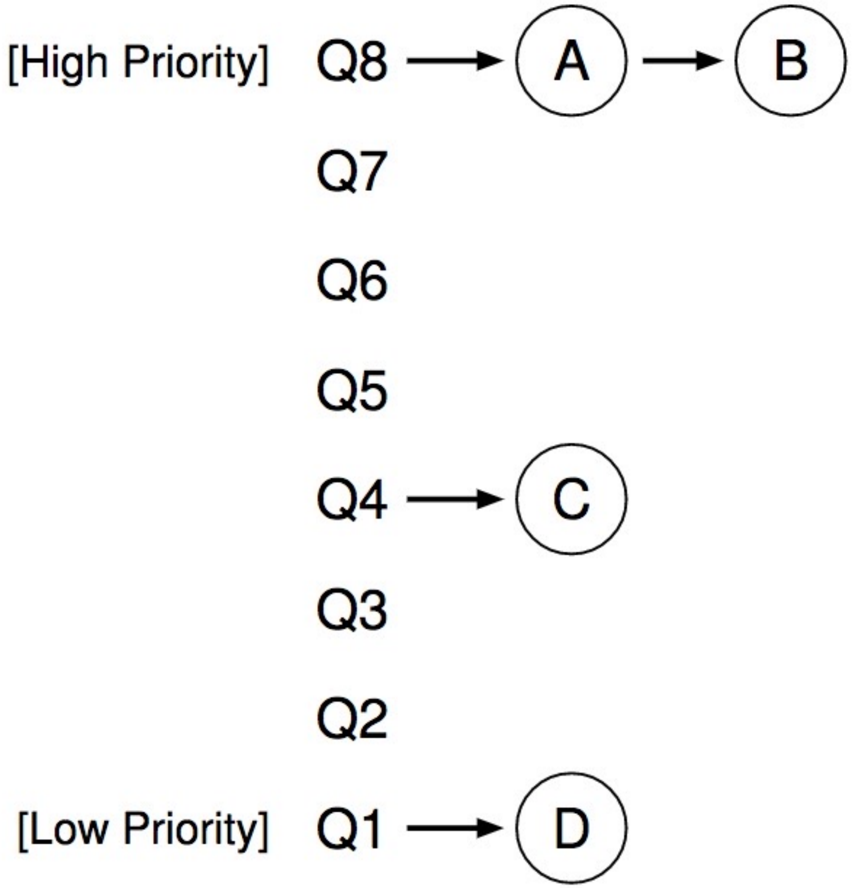
How to know process type to set priority?

1. nice
2. history

How to Check Nice Values in Linux?

- `% ps ax -o pid,ni,cmd`

MLFQ Example



How to know process type to set priority?

1. nice
2. history

In this example, A and B are given high priority to run, while C and D may starve

MLFQ: Basic Rules

- MLFQ maintains a number of queues (multi-level queue)
  - Each assigned a different priority level
  - Priority decides which process should run at a given time

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

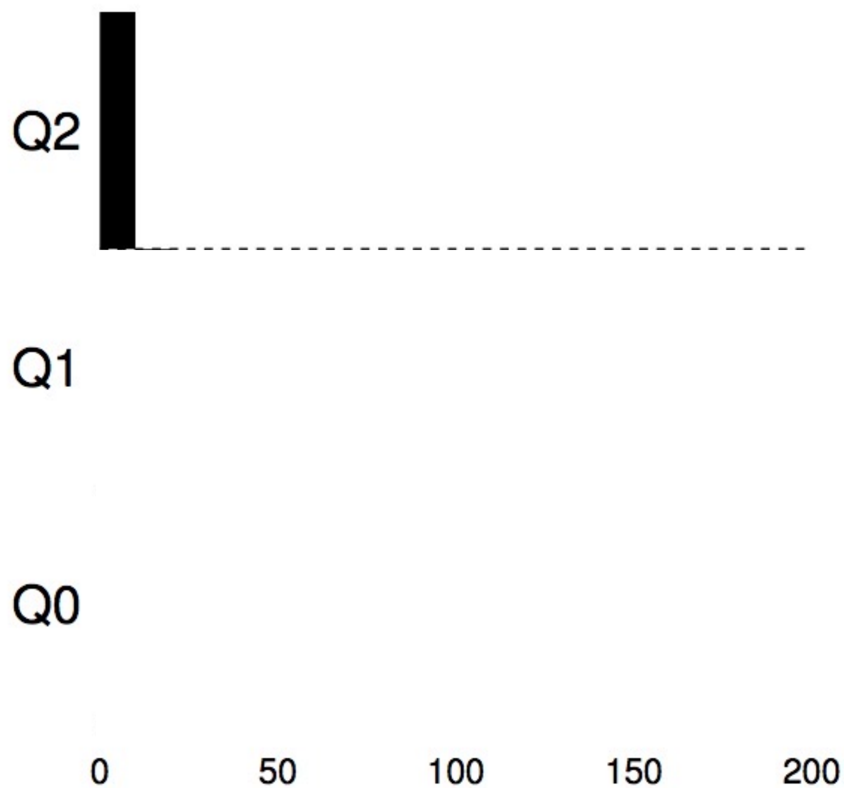
Attempt #1: Change Priority

- Workload
  - Interactive processes (many short-run CPU bursts)
  - Long-running processes (CPU-bound)
- Each time quantum = 10ms

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

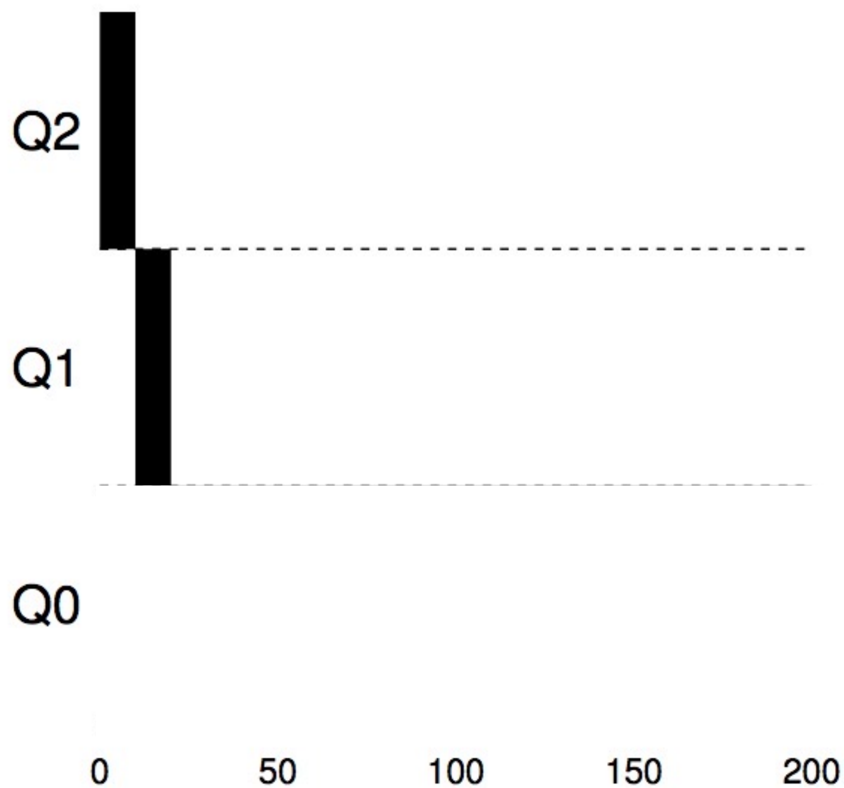
Example 1: One Single Long-Running Process

- A process enters at highest priority (time quantum = 10ms)



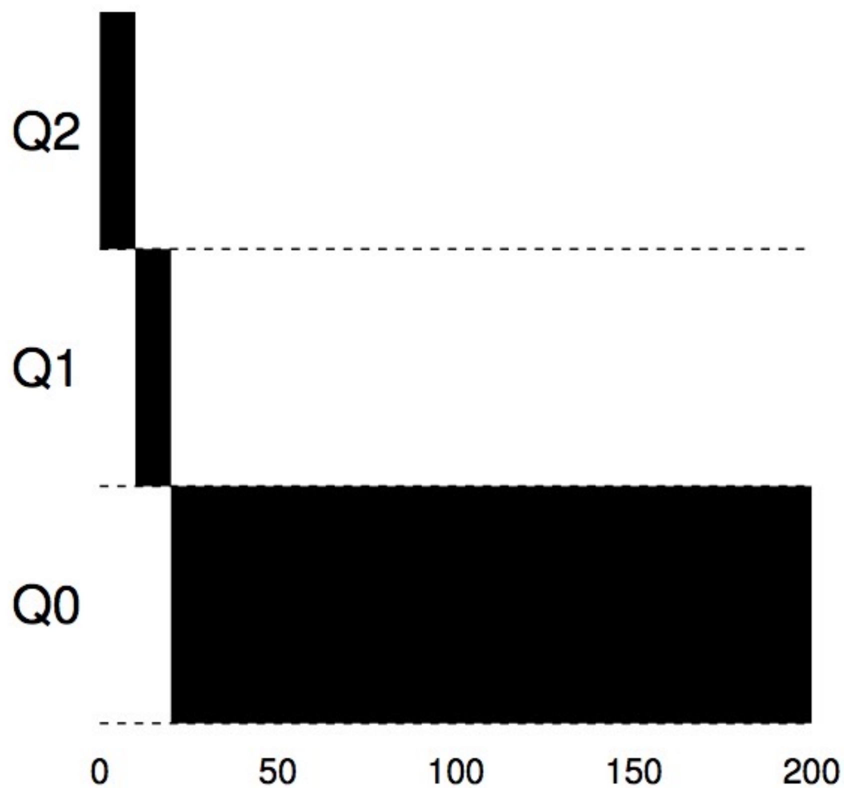
Example 1: One Single Long-Running Process

- A process enters at highest priority (time quantum = 10ms)



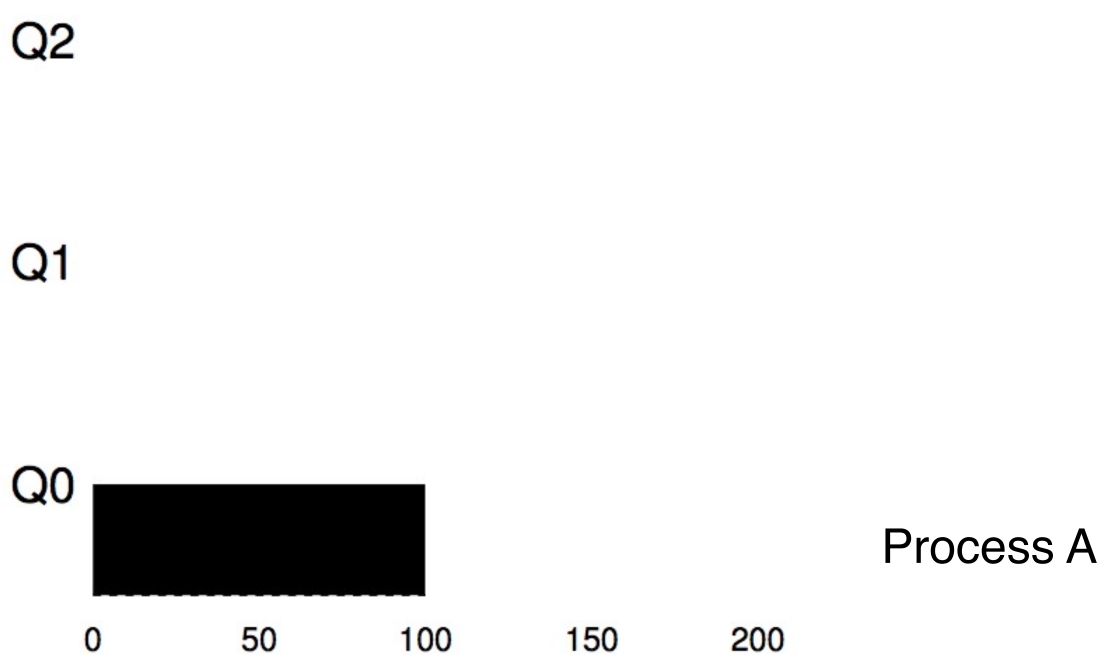
Example 1: One Single Long-Running Process

- A process enters at highest priority (time quantum = 10ms)



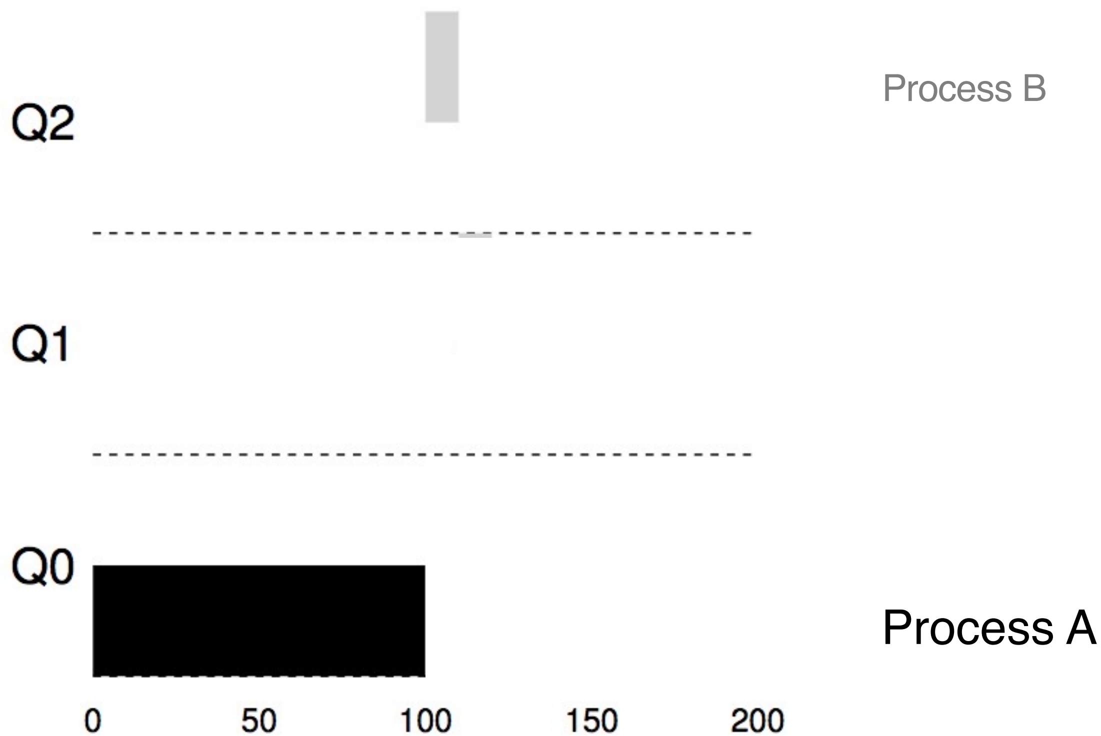
Example 2: Along Came a Short-Running Process

- Process A: long-running process (start at 0)



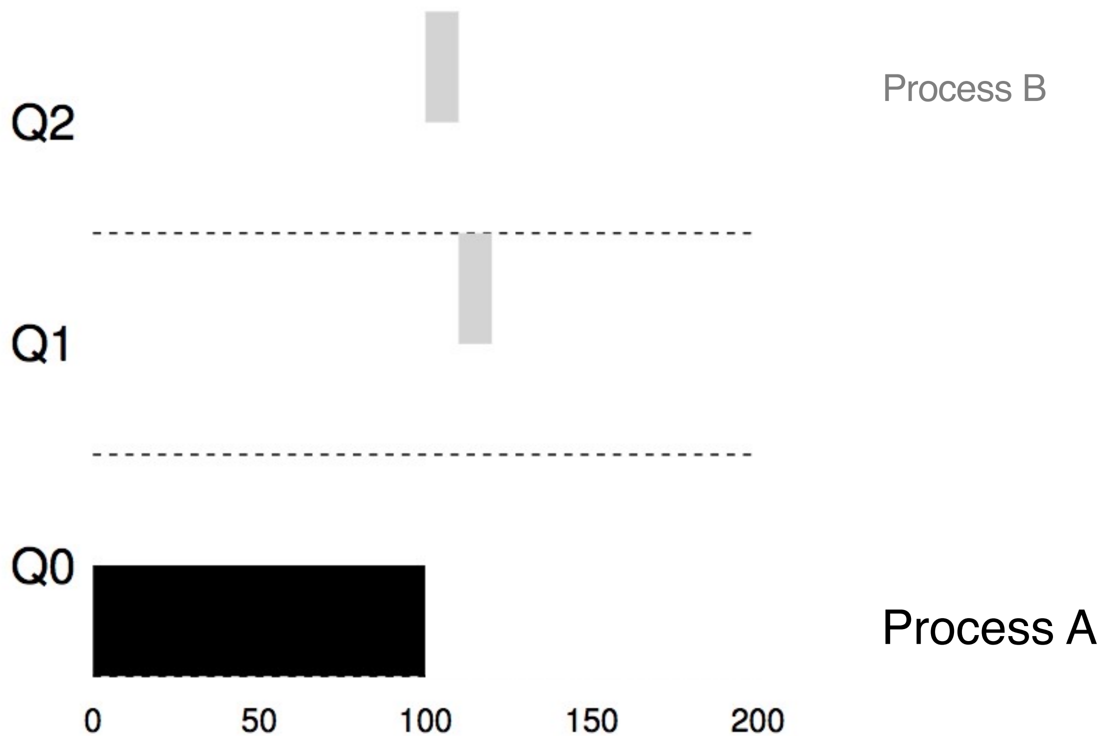
Example 2: Along Came a Short-Running Process

- Process A: long-running process (start at 0)
- Process B: short-running interactive process (start at 100)



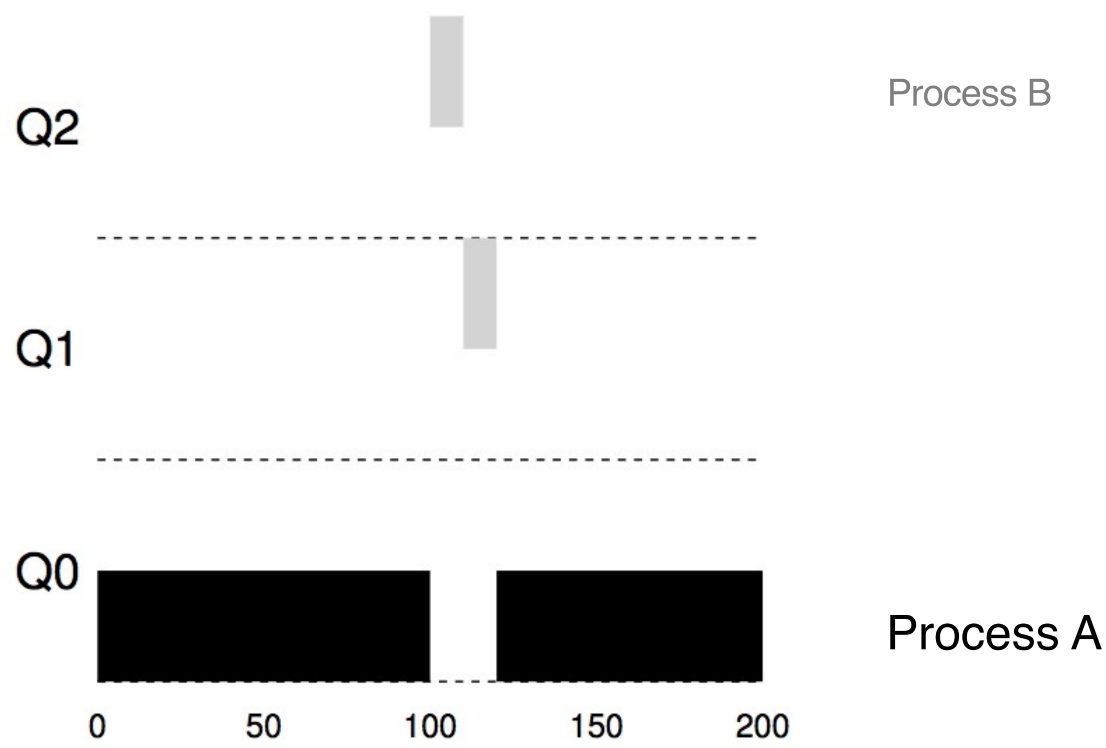
Example 2: Along Came a Short-Running Process

- Process A: long-running process (start at 0)
- Process B: short-running interactive process (start at 100)



Example 2: Along Came a Short-Running Process

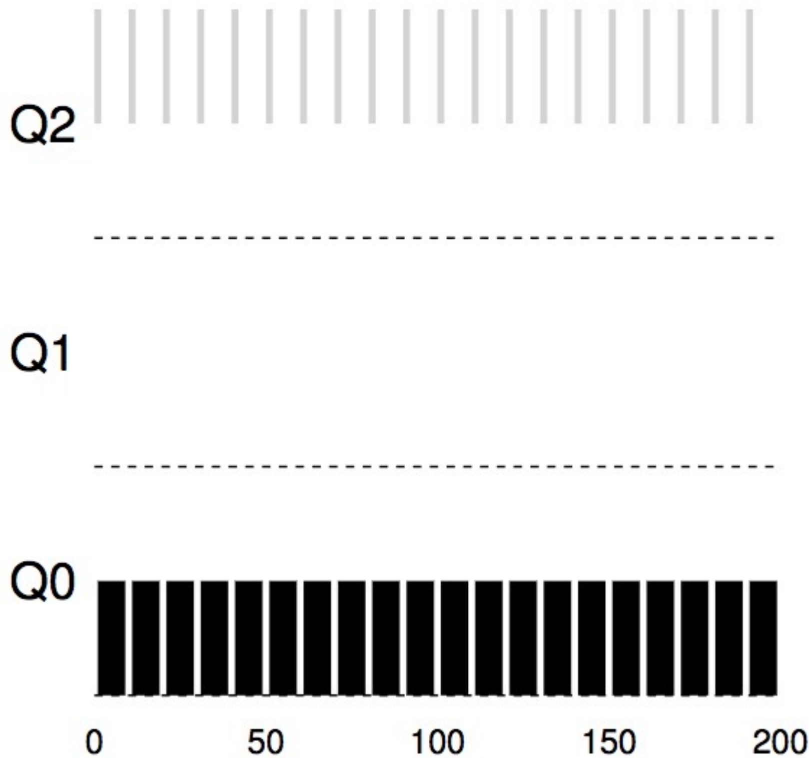
- Process A: long-running process (start at 0)
- Process B: short-running interactive process (start at 100)



Example 3: What about I/O?

- Process A: long-running process
- Process B: I/O-intensive interactive process (each CPU burst = 1ms)

Rule 4b

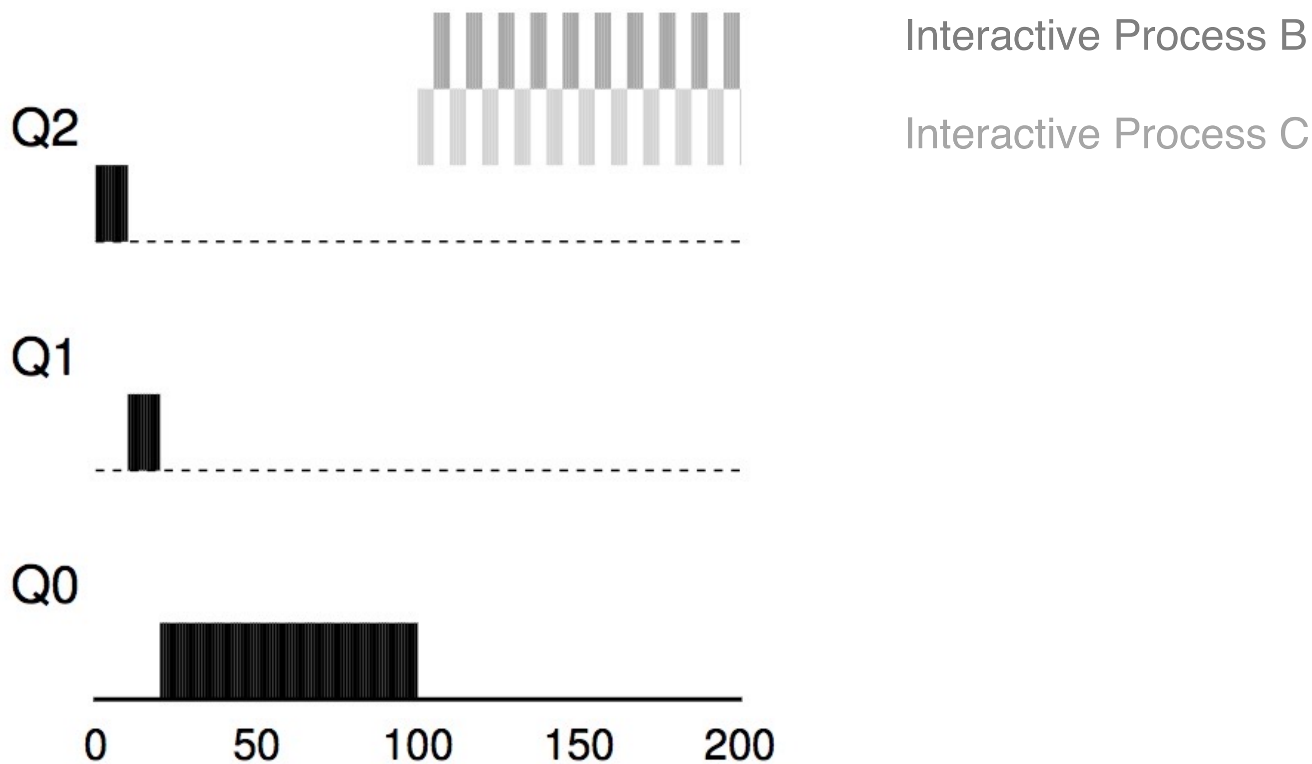


I/O-intensive Process B

CPU-intensive Process A

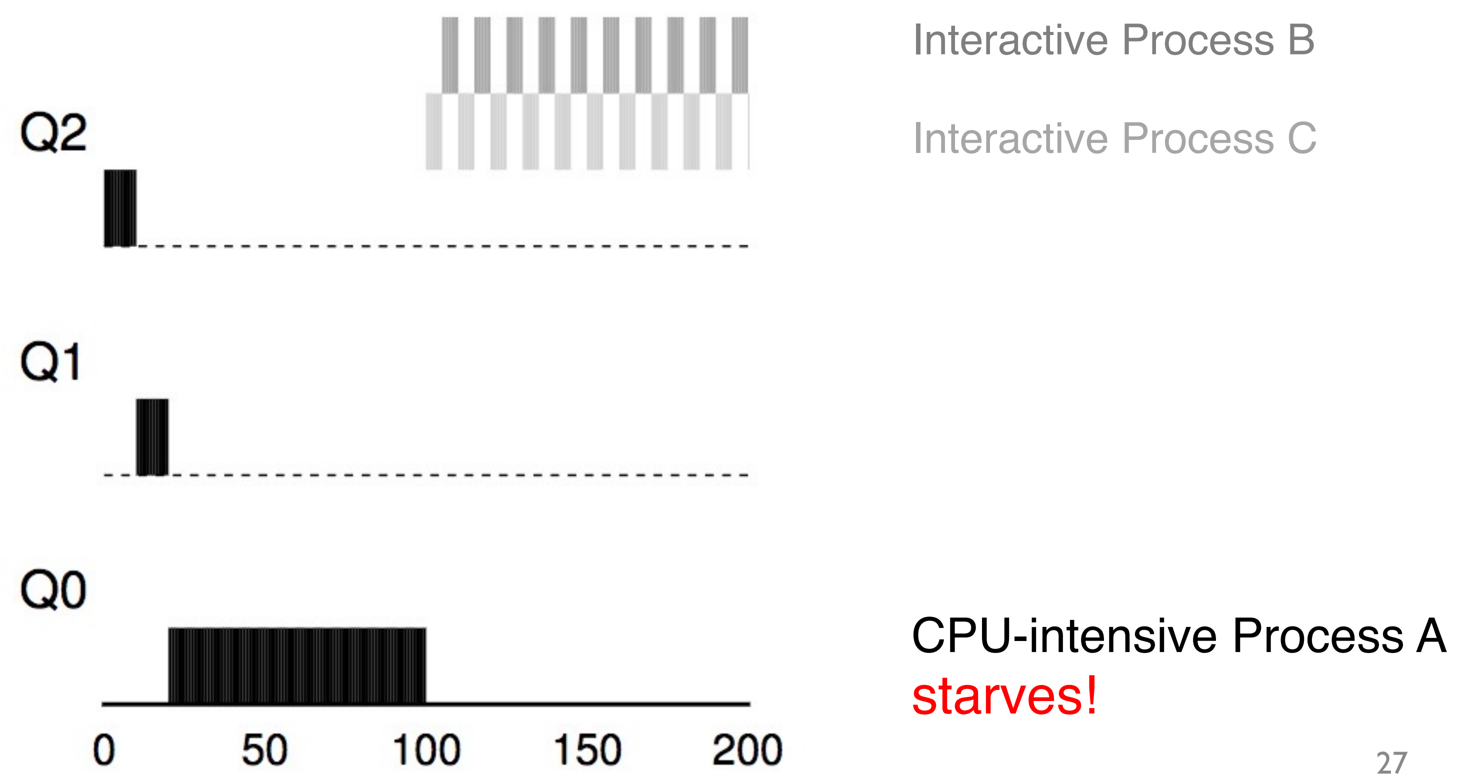
Example 4: What's the Problem?

- Process A: long-running process
- Process B + C: Interactive process



Example 4: What's the Problem?

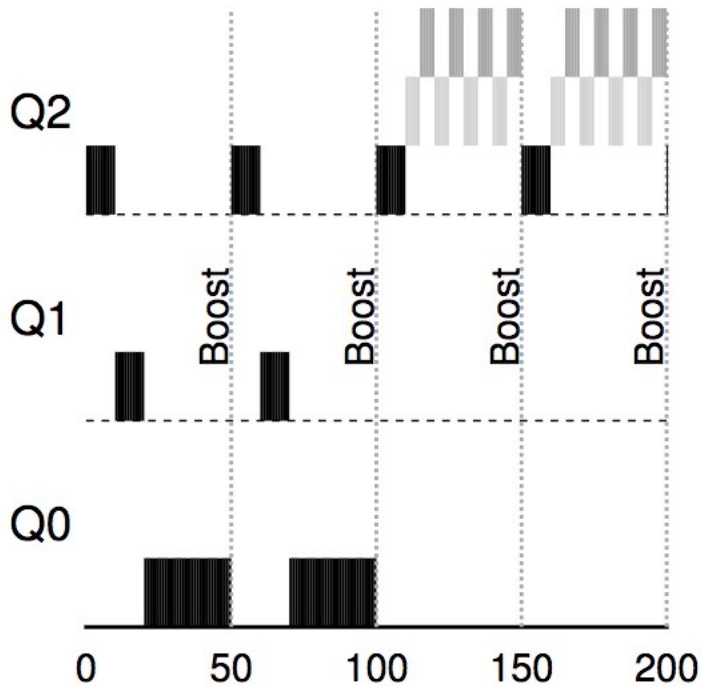
- Process A: long-running process
- Process B + C: Interactive process



Attempt #2: Priority Boost

- Simple idea: Periodically boost the priority of all processes

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



Interactive Process B

Interactive Process C

CPU-intensive Process A proceeds!

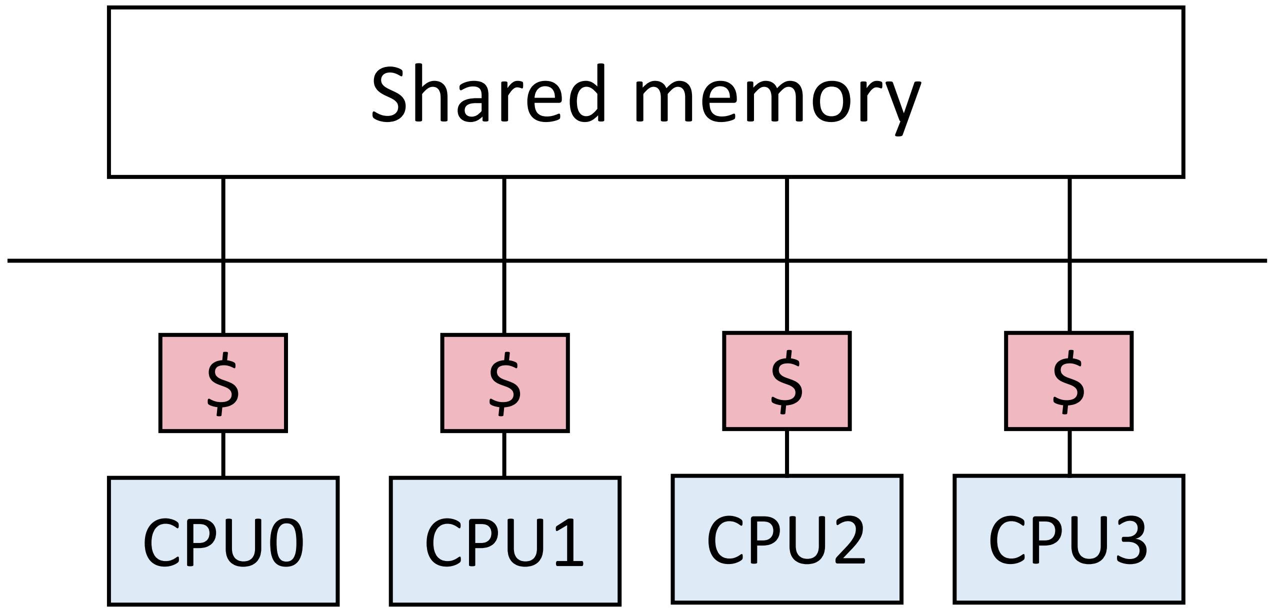
Tuning MLFQ

- MLFQ scheduler is defined by many parameters:
  - Number of queues
  - Time quantum of each queue
  - How often should priority be boosted?
  - A lot more…

- The scheduler can be configured to match the requirements of a specific system
  - Challenging and requires experience

Linux Scheduling

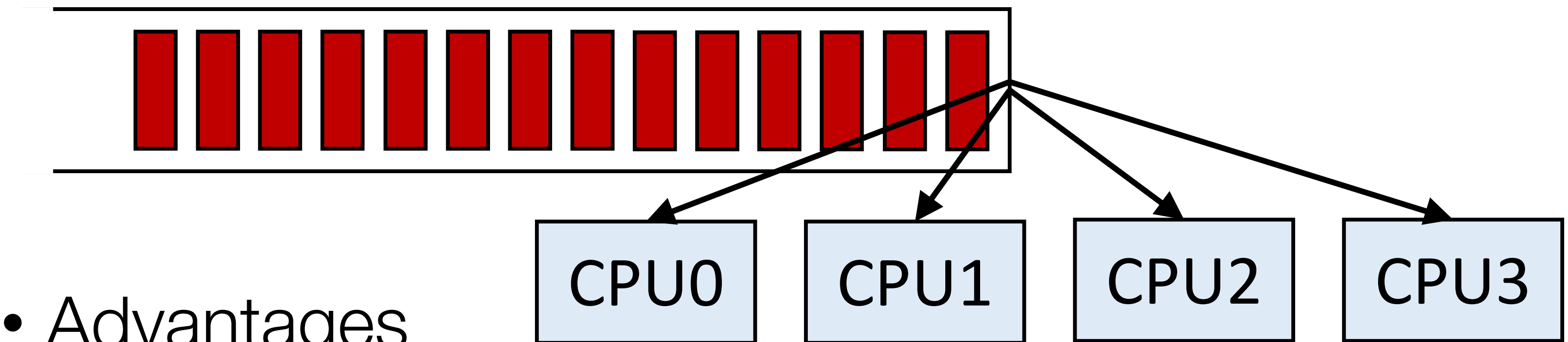
Symmetric Multiprocessing (SMP)



- Multiple CPUs
- Same access time to main memory (DRAM)
- Private CPU cache

Global Queue of Processes

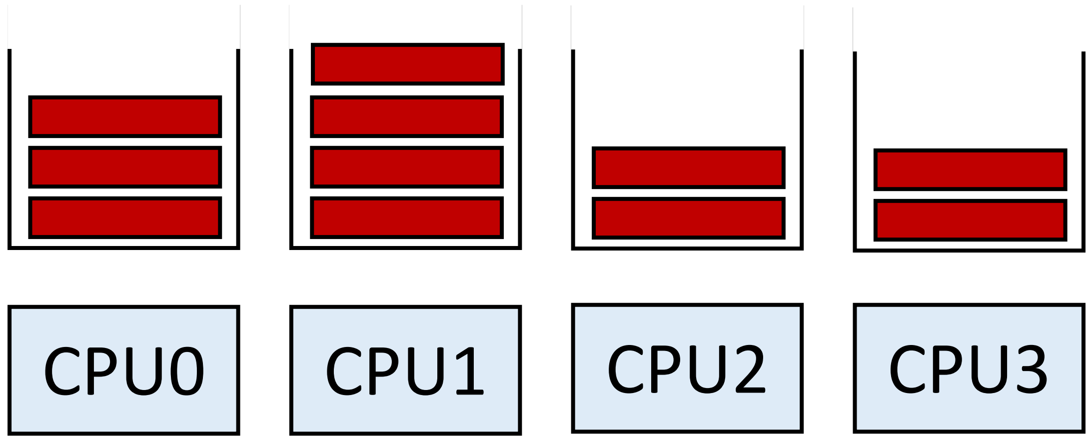
- One ready queue shared across all CPUs



- Advantages
  - Good CPU utilization
  - Fair to all processes
- Disadvantages
  - Not scalable (contention for global queue lock)
  - Poor cache locality
- Linux 2.4 uses global queue

Per-CPU queue of processes

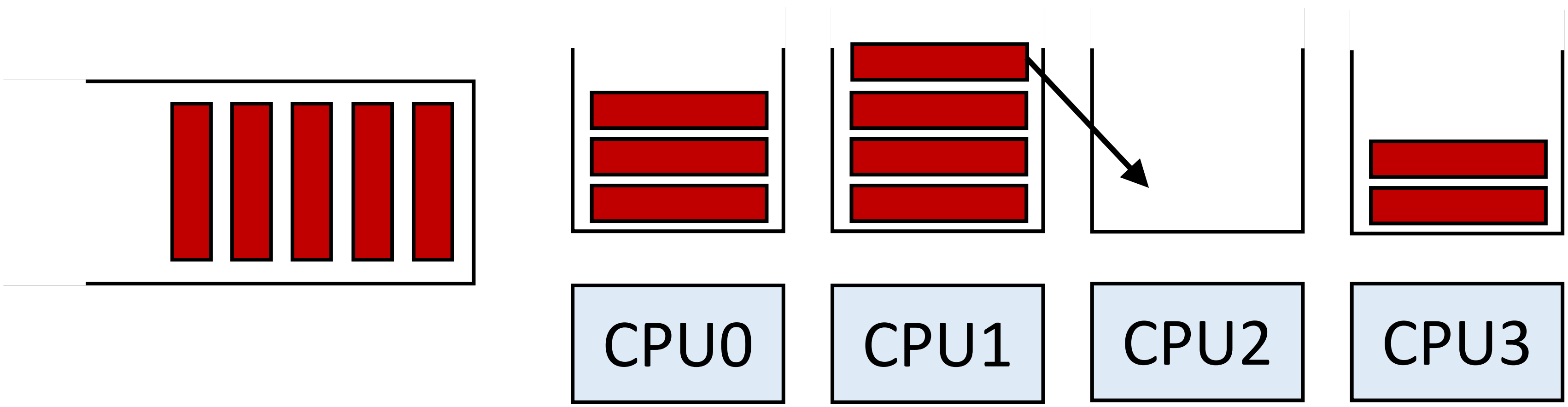
- Static partition of processes to CPUs



- Advantages
  - Easy to implement
  - Scalable (no contention on ready queue)
  - Better cache locality
- Disadvantages
  - Load imbalance (some CPUs have more processes)
    - Unfair to processes and lower CPU utilizations

Hybrid Approaches

- Use both global and per-CPU queues
- Migrate processes across per-CPU queues



- Processor affinity
  - Add process to a CPU's queue if recently run on that CPU
    - Cache state may still present

Real-Time Scheduling

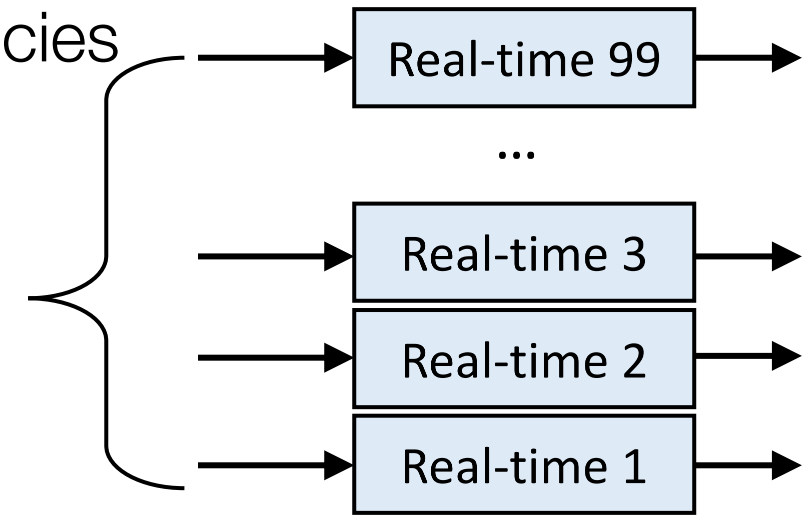
- Real-time processes have timing constraints
  - Expressed as deadlines or rate requirements
  - E.g., gaming, video/music player, autopilot

- Hard real-time systems – required to complete a critical task within a guaranteed amount of time
- Soft real-time computing – requires that critical processes receive priority over others

- Linux supports soft real-time

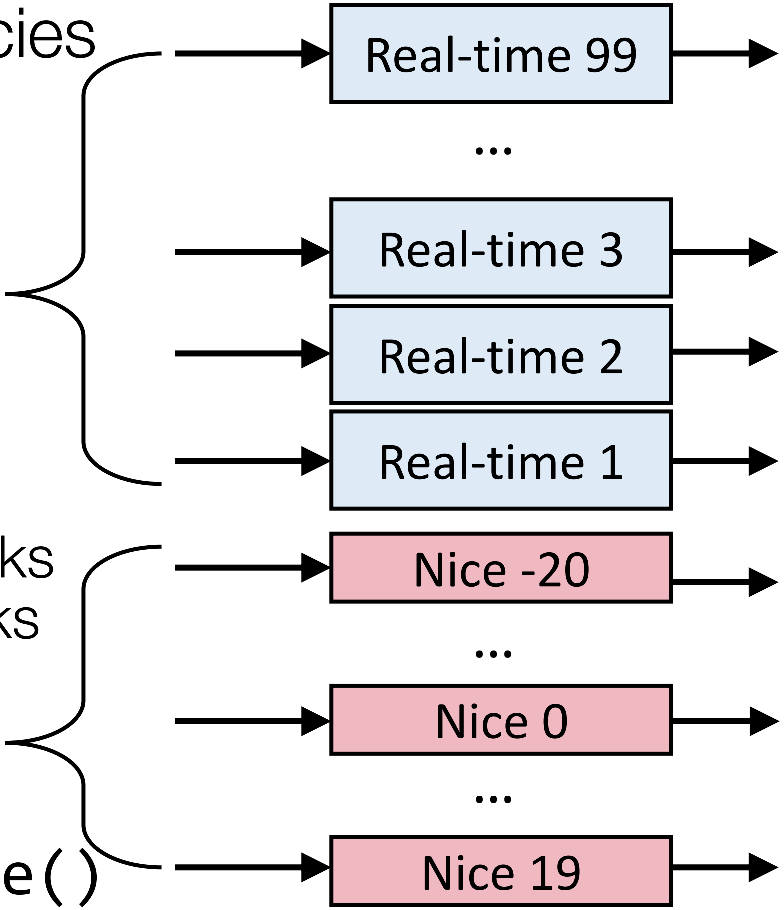
Linux: Multi-Level Queue with Priorities

- Soft real-time scheduling policies
  - `SCHED_FIFO` (FIFO)
  - `SCHED_RR` (round robin)
  - Priority over normal tasks
  - 100 static priority levels (1–99)



Linux: Multi-Level Queue with Priorities

- Soft real-time scheduling policies
  - `SCHED_FIFO` (FIFO)
  - `SCHED_RR` (round robin)
  - Priority over normal tasks
  - 100 static priority levels (1–99)
- Normal scheduling policies
  - `SCHED_NORMAL`: standard
    - `SCHED_OTHER` in POSIX
  - `SCHED_BATCH`: CPU-bound tasks
  - `SCHED_IDLE`: lower priority tasks
  - Static priority is 0
    - 40 dynamic priority levels
    - "Nice" values
- `sched_setscheduler(), nice()`
- See "`man 7 sched`" for detailed overview



Linux Scheduler History

- `O(N)` scheduler up to 2.4
  - Simple: global run queue
  - Poor performance on multiprocessor and large N
- `O(1)` scheduler in 2.5 & 2.6
  - Good performance: per-CPU run queue
  - Complex and error-pone logic to boost interactivity
  - No fairness guarantee
- Completely Fair Scheduler (CFS) in 2.6 and later
  - Currently default scheduler for `SCHED_NORMAL`
  - Processes get fair share of CPU
  - Naturally boosts interactivity

O(N) Scheduler (Linux 2.4)

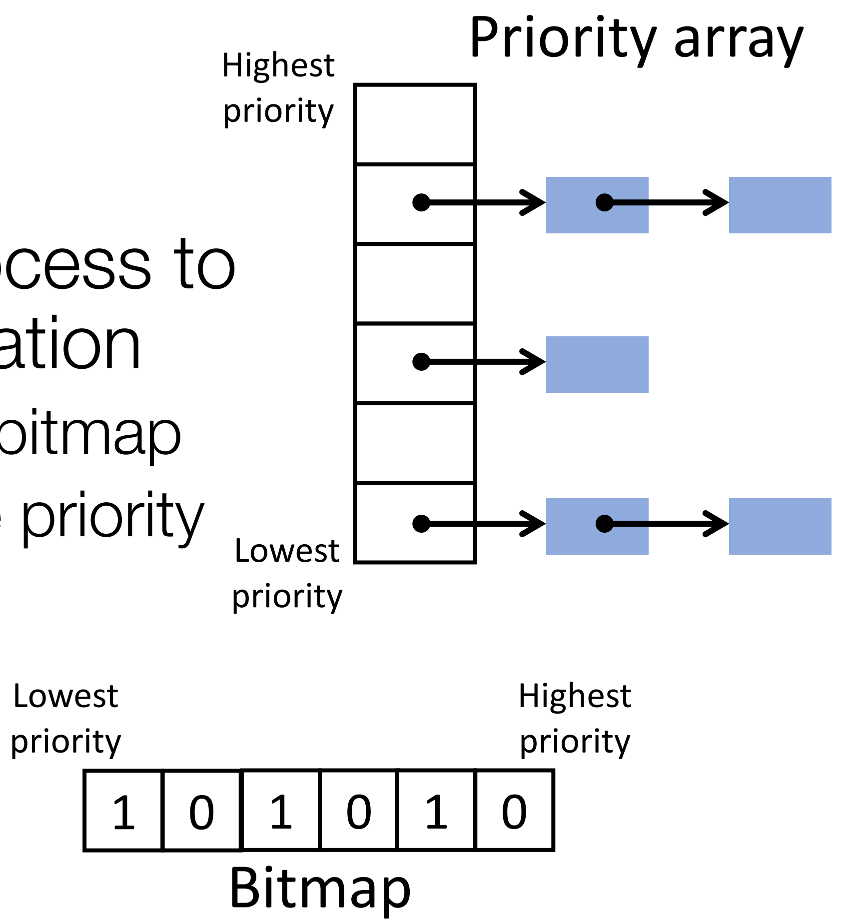
- Time is divided into epochs
- At the start of each epoch, scheduler assigns a priority to every process based on its behavior
  - Real-time processes have an absolute priority assigned to them, and are highest priority
  - Interactive processes have a dynamic priority assigned to them based on behavior in the previous epoch
  - Batch processes are given the lowest priority
- Each process' priority is used to compute a time quantum
  - Different processes can have different quantum lengths
  - Higher-priority processes generally get larger time quantums
  - When a process has completely used up its quantum, it is preempted and another process runs

O(N) Scheduler (Linux 2.4)

- When scheduler is invoked or at start of an epoch, scheduler iterates thru all processes
  - Compute a new priority for each process
- Higher-priority processes preempt lower-priority ones
- The current epoch ends when all runnable processes have consumed their entire time quantum
- Several O(N) computations in the scheduler makes it scale terribly to large numbers of processes

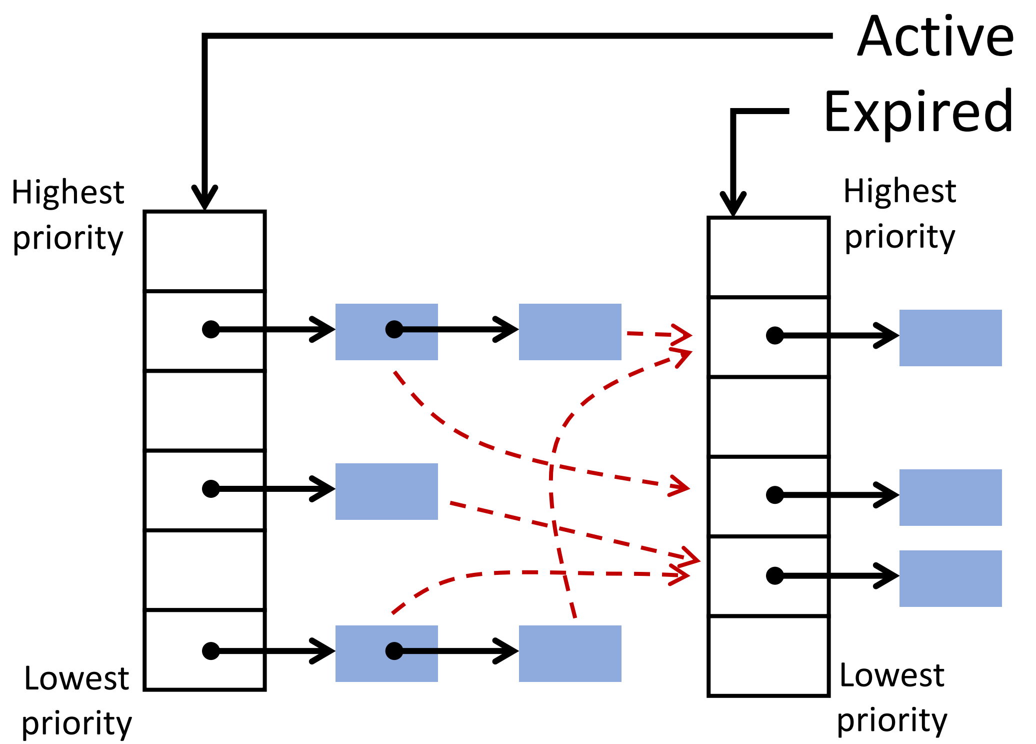
O(1) Scheduler (Linux 2.6)

- Linux `O(1)` scheduler still includes the notion of epochs, but only informally
- Priority array + bitmap
- Find the highest-priority process to run is a constant-time operation
  - Find index of lowest 1-bit in bitmap
  - Use that index to access the priority array



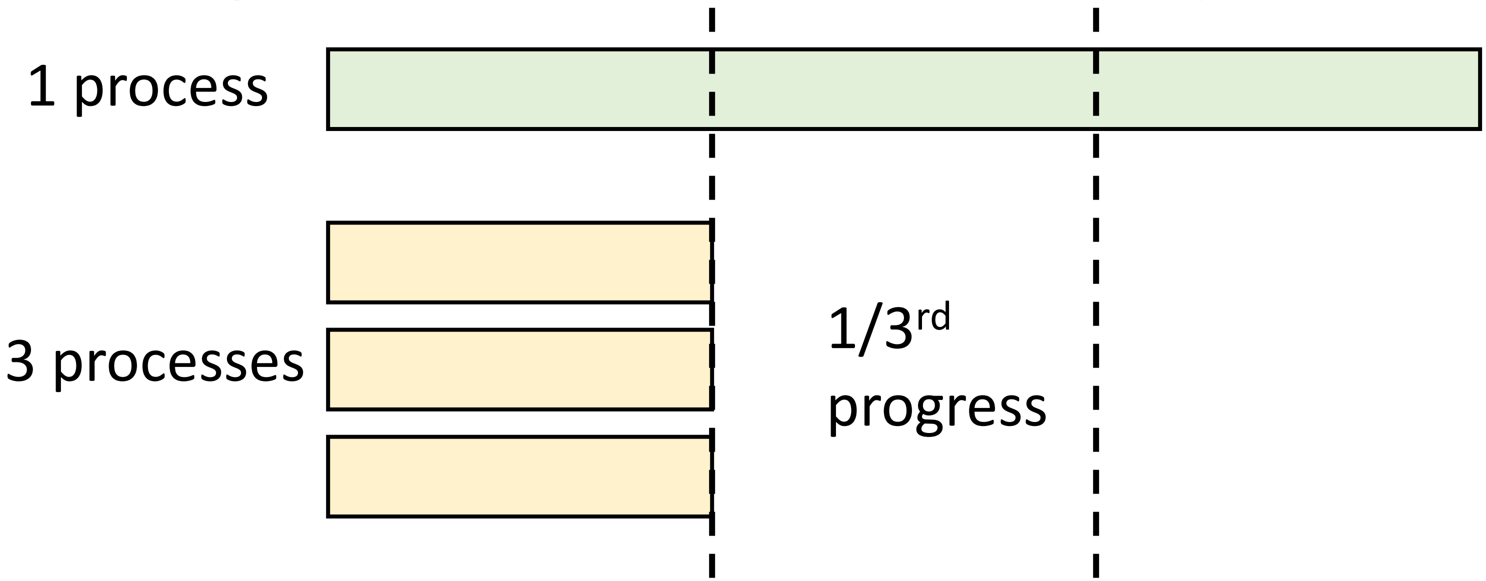
O(1) Scheduler (Linux 2.6)

- Maintains two priority arrays
  - Active array contains processes w/ remaining time
  - Expired array holds processes that have used up their quantums
- When an active process uses entire quantum, it is moved to the expired array
  - A new priority is given to that process
- When the active array is empty, the epoch is over
  - `O(1)` scheduler switches the active and expired pointers and starts over again



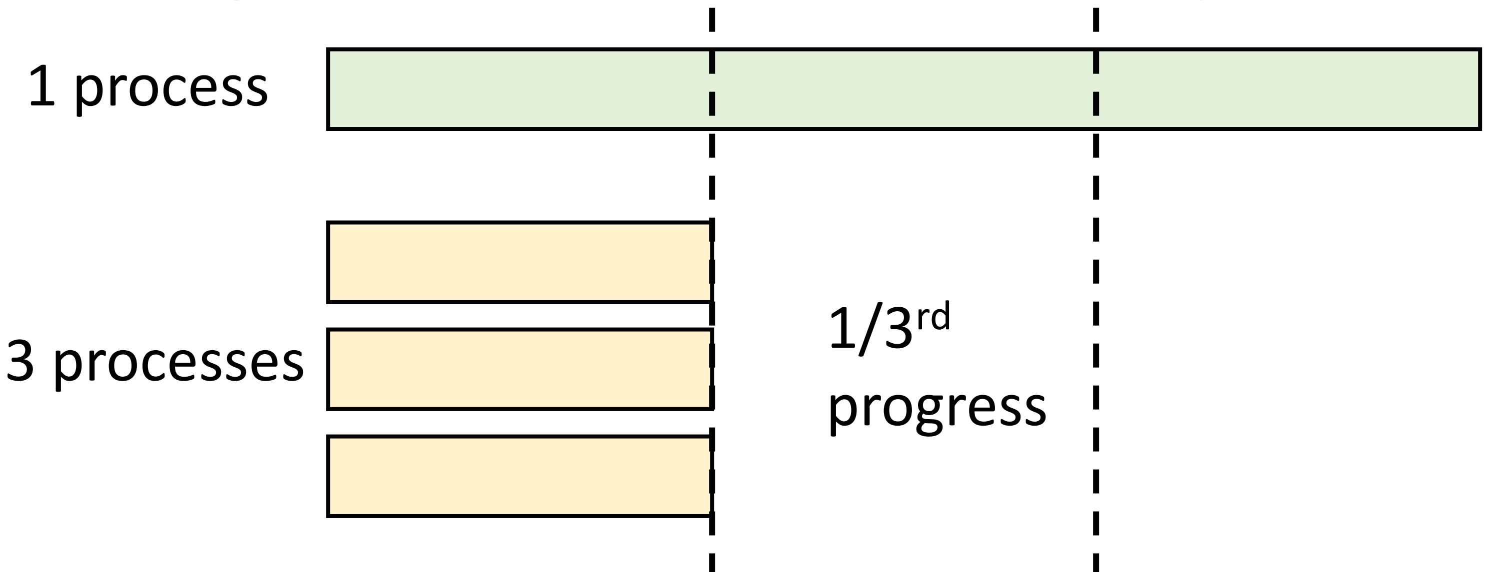
Ideal Fair Scheduling

- Infinitesimally small time slice
- N processes: each runs uniformly at 1/Nth rate



Ideal Fair Scheduling

- Infinitesimally small time slice
- N processes: each runs uniformly at $1/N^{th}$ rate



- Various approximations of the idea
  - Linux CFS
  - Lottery scheduling

Completely Fair Scheduler (Linux 2.6.23 till now)

- CFS approximates fair scheduling
  - Run each process once per schedule period `T`
    - `sysctl_sched_latency`
  - Time slice for process Pi: T * Wi/(Sum of all Wi)
    - `sched_slice()`

- Too many processes?
  - Lower bound on smallest time slice
    - `sysctl_sched_min_granularity`
  - Schedule latency T = lower bound * number of procs

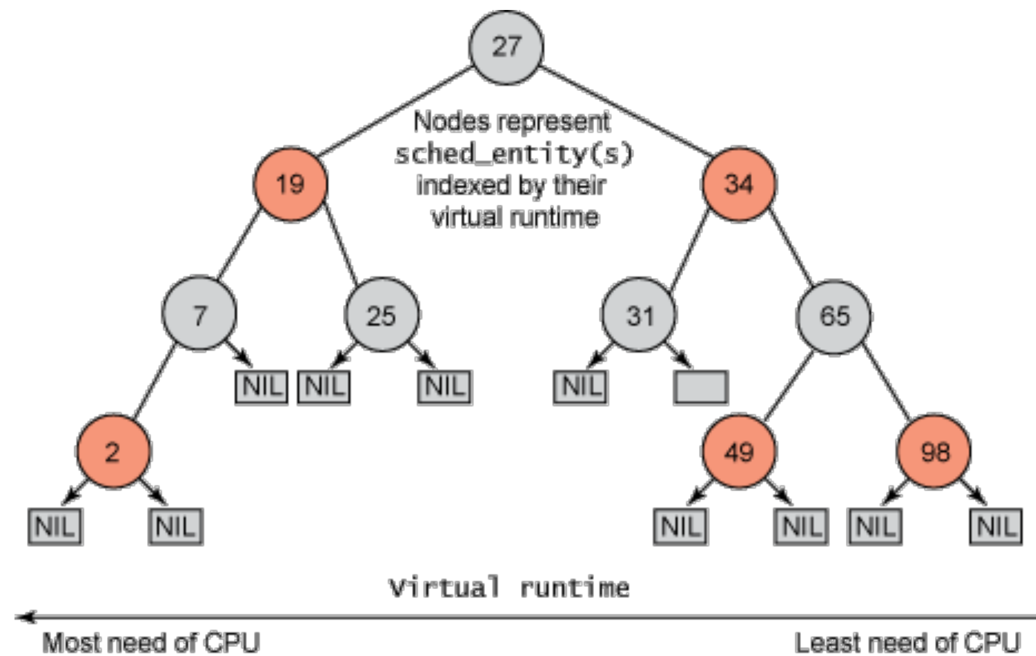
CFS: Picking the Next Process

- Pick process w/ minimum weighted vruntime so far
  - Virtual runtime:

```
task->vruntime += executed time / Wi
```

CFS: Picking the Next Process

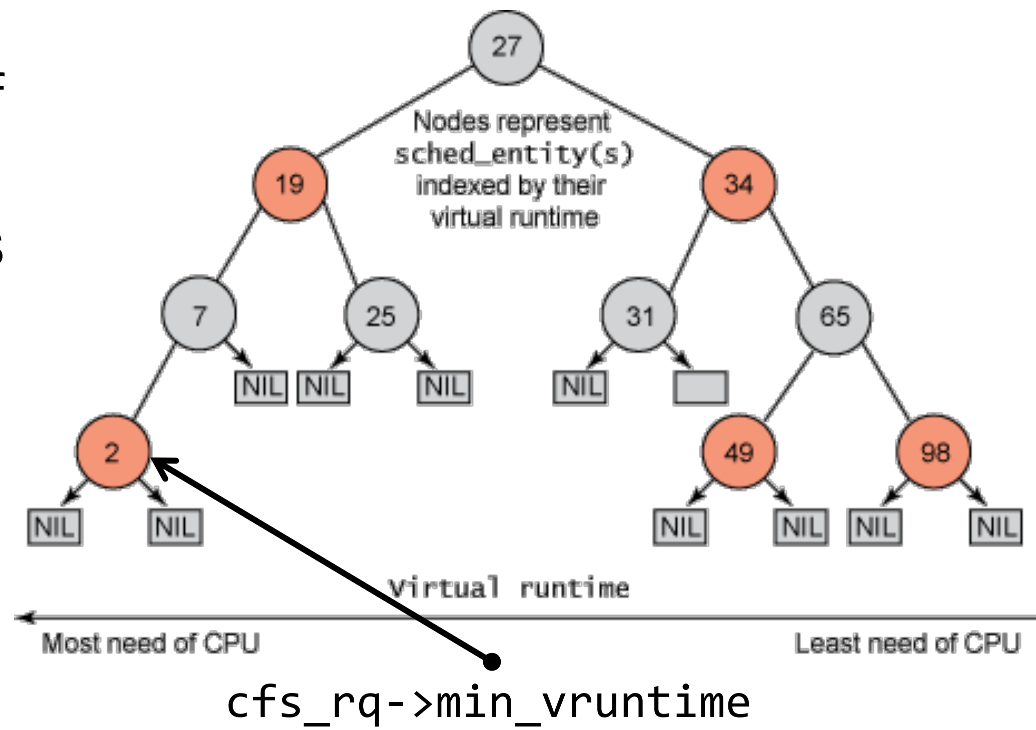
- CFS uses a red-black tree (RB tree)
  - Balanced binary search tree (BST)
  - Ordered by `vruntime` as key
  - `O(logN)` insertion, deletion, update; `O(1)`: find min



CFS: Picking the Next Process

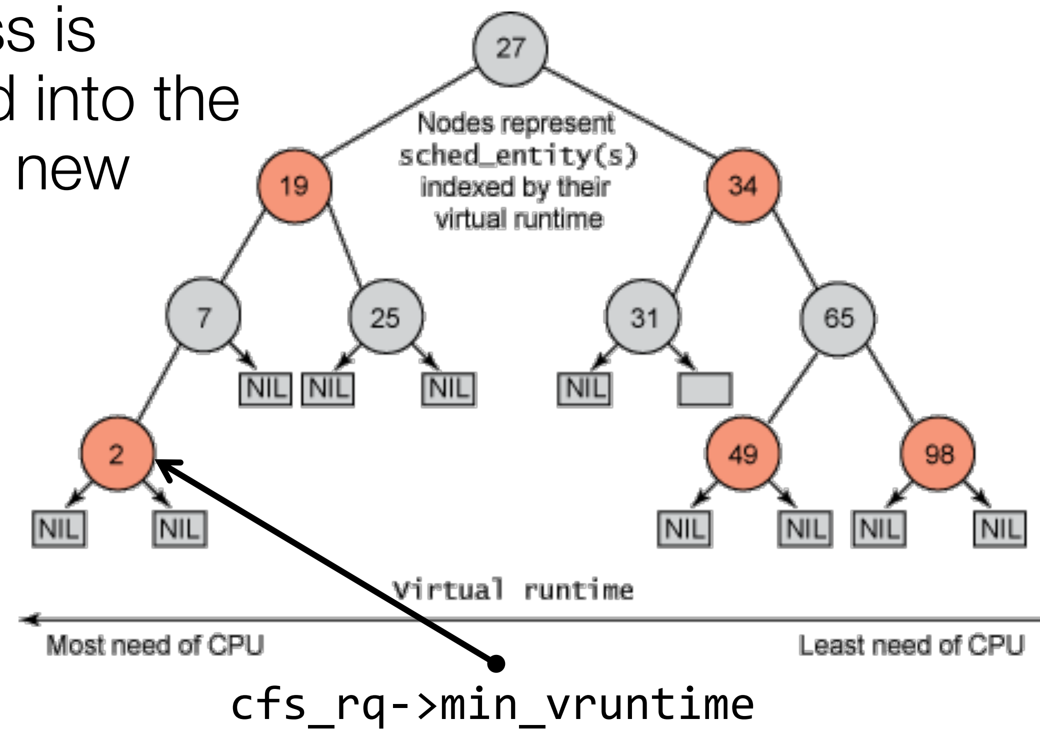
- CFS uses a red-black tree (RB tree)
  - Balanced binary search tree (BST)
  - Ordered by `vruntime` as key
  - `O(logN)` insertion, deletion, update; `O(1)`: find min

- Tasks move from left of tree to the right
- `min_vruntime` caches smallest value
- Update `vruntime` and `min_vruntime`
  - When task is added or removed
  - On every timer tick, context switch



CFS: Picking the Next Process

- Sched is invoked at context switch or at timer tick
  - Pick the left-most node w/ the lowest `vruntime`
  - If the previous process is runnable, it is inserted into the tree depending on its new `vruntime`



How CFS Handles I/O-bound Processes?

- Ideally:
  - An I/O-bound process should get higher priority and thus should get the CPU more easily (after being blocked for a while waiting for I/O)

- How CFS boosts interactivity:
  - I/O-bound processes typically have shorter CPU bursts and thus will have a low `vruntime` – higher priority