

CPU Virtualization: Priority, MLFQ, and CFS

CS 571: *Operating Systems (Spring 2021)*
Lecture 3

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials created by Remzi Arpacı-Dusseau.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

CPU Scheduling: Outline

- Basic concept
- Scheduling criteria
- Scheduling algorithms
 - First In, First Out (FIFO)
 - Shortest Job First (SJF)
 - Shortest Time-to-Completion First (STCF)
 - Round Robin (RR)
 - Priority
 - Multi-Level Feedback Queue (MLFQ)
 - (Advanced) Linux scheduling
 - Completely Fair Scheduler (CFS)
 - Lottery Scheduling

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

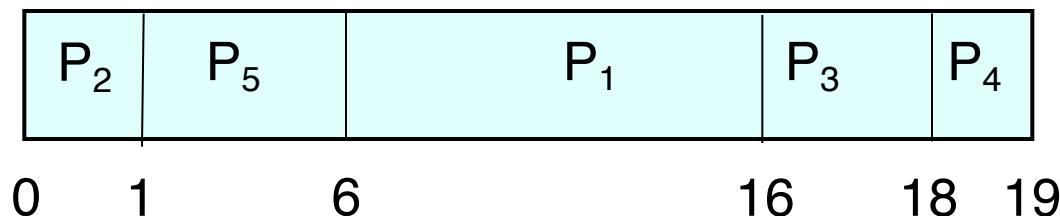
Priority-Based Scheduling

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

Example for Priority-Based Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
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)

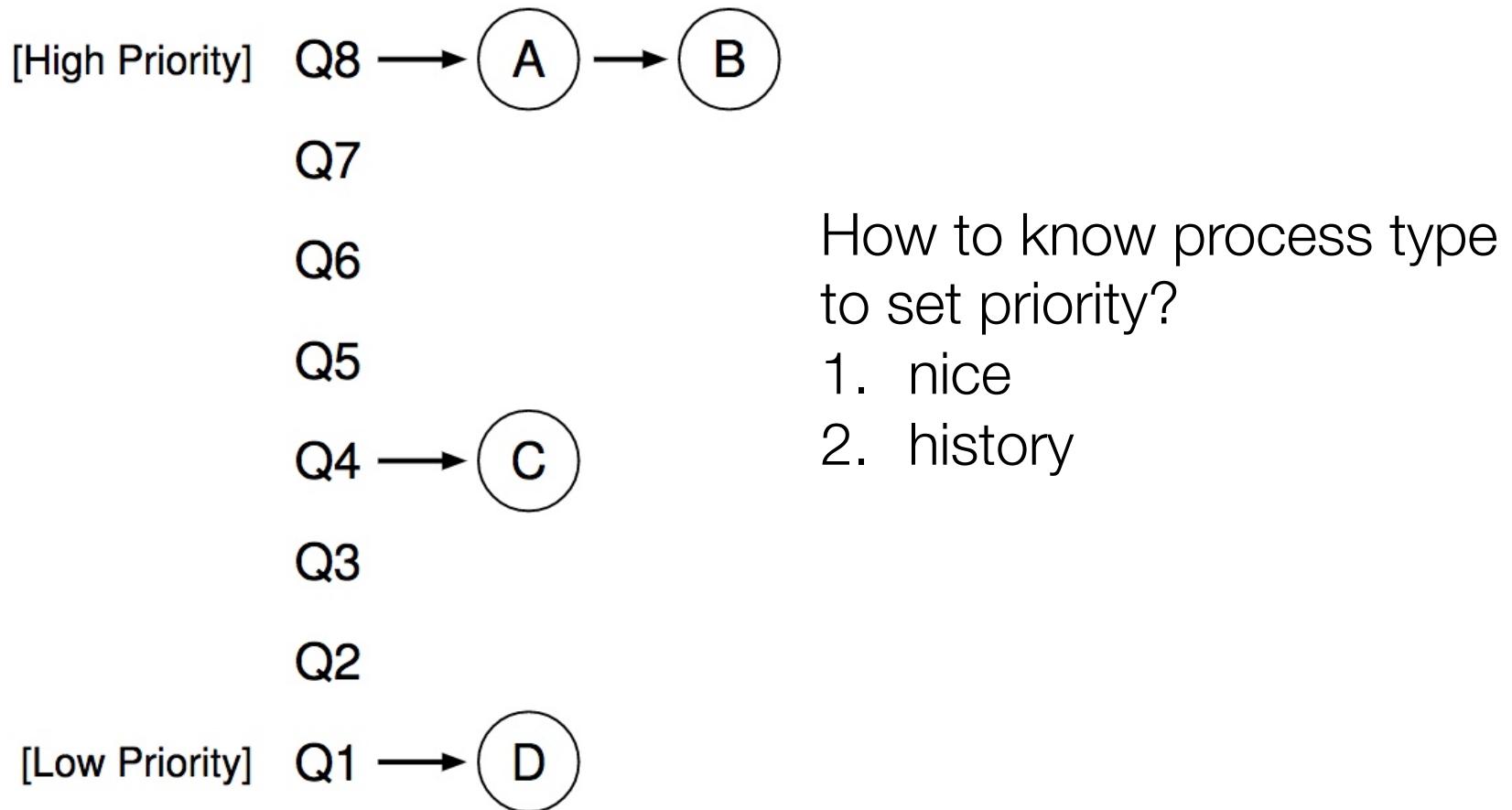
Multi-Level Feedback Queue (MLFQ)

- 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

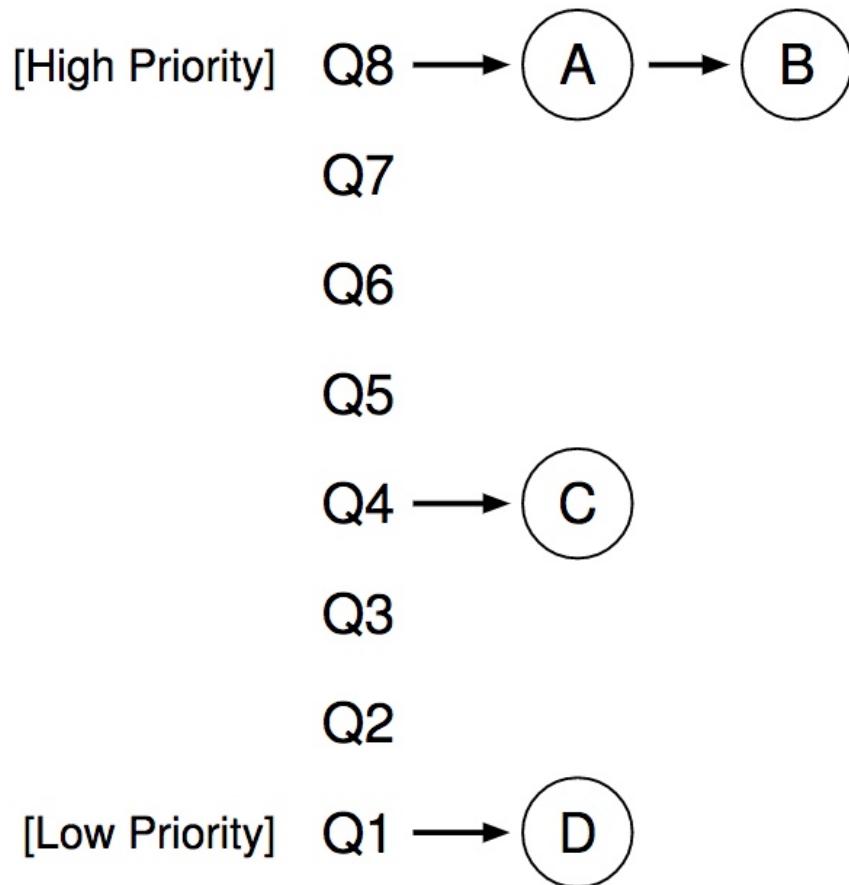
MLFQ Example



How to 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

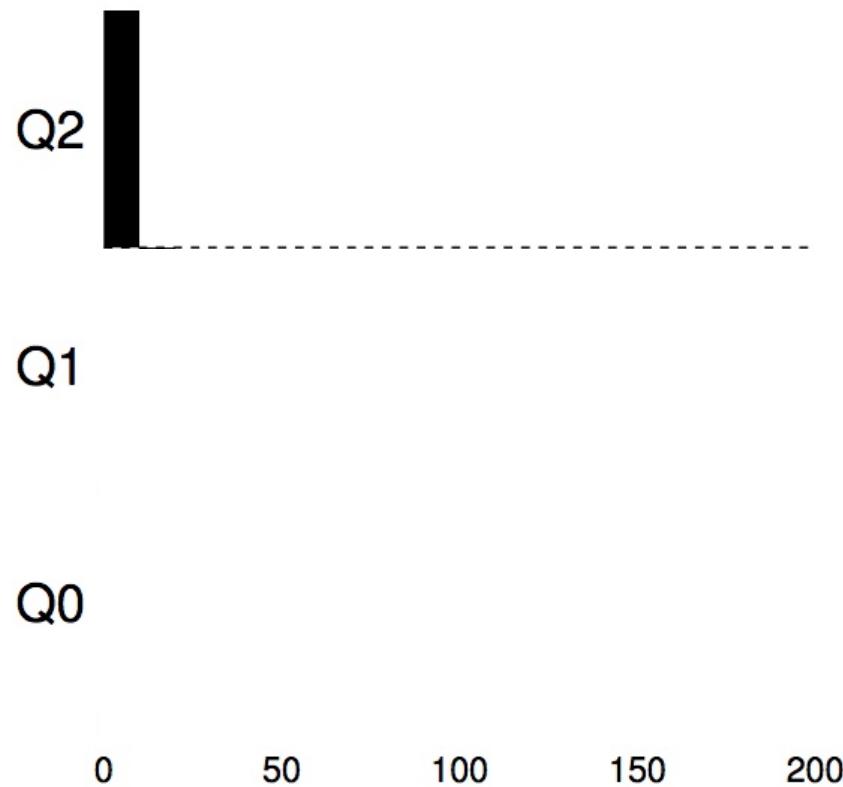
- MLFQ maintains a number of queues (multi-level queue)
 - Each assigned a different priority level
 - Priority decides which process should run at a given time
- **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
- **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, 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.

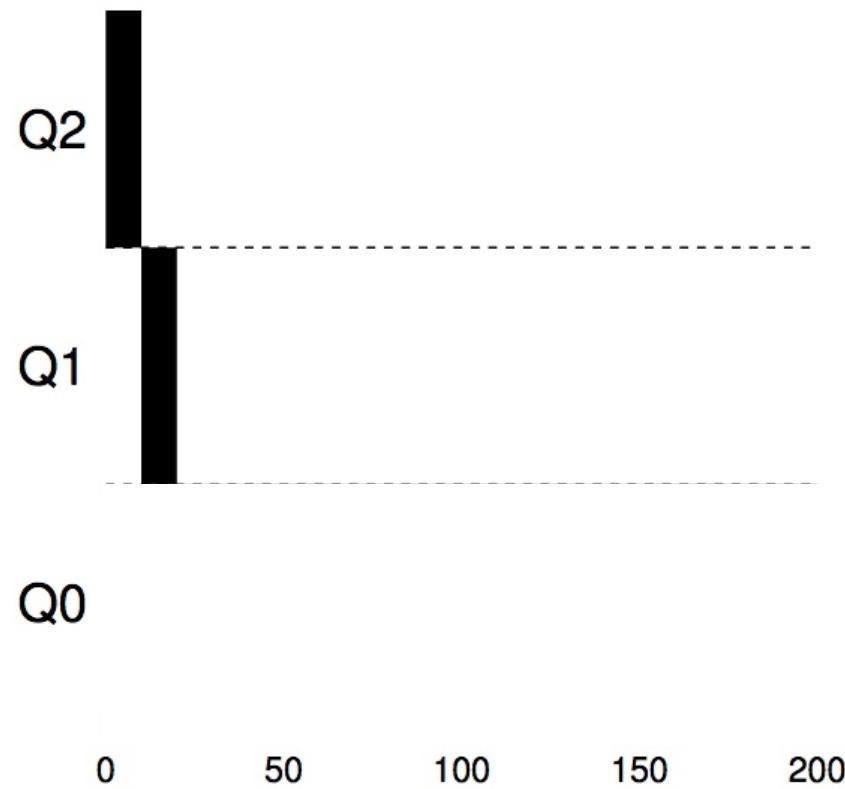
Example 1: One Single Long-Running Process

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



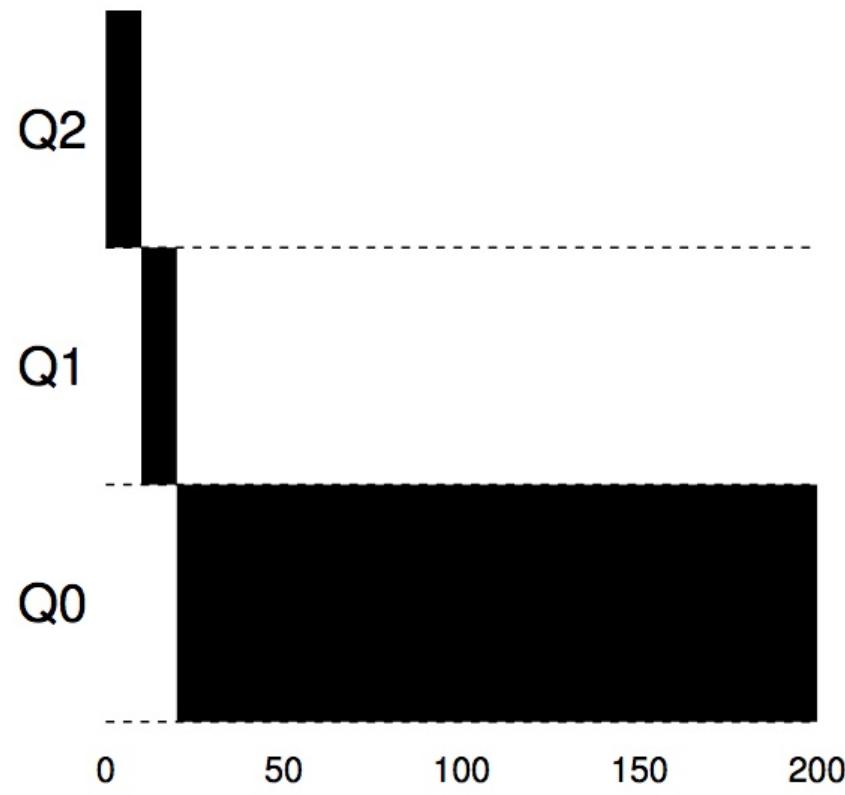
Example 1: One Single Long-Running Process

- 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)

Q2

Q1

Q0



Process A

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)



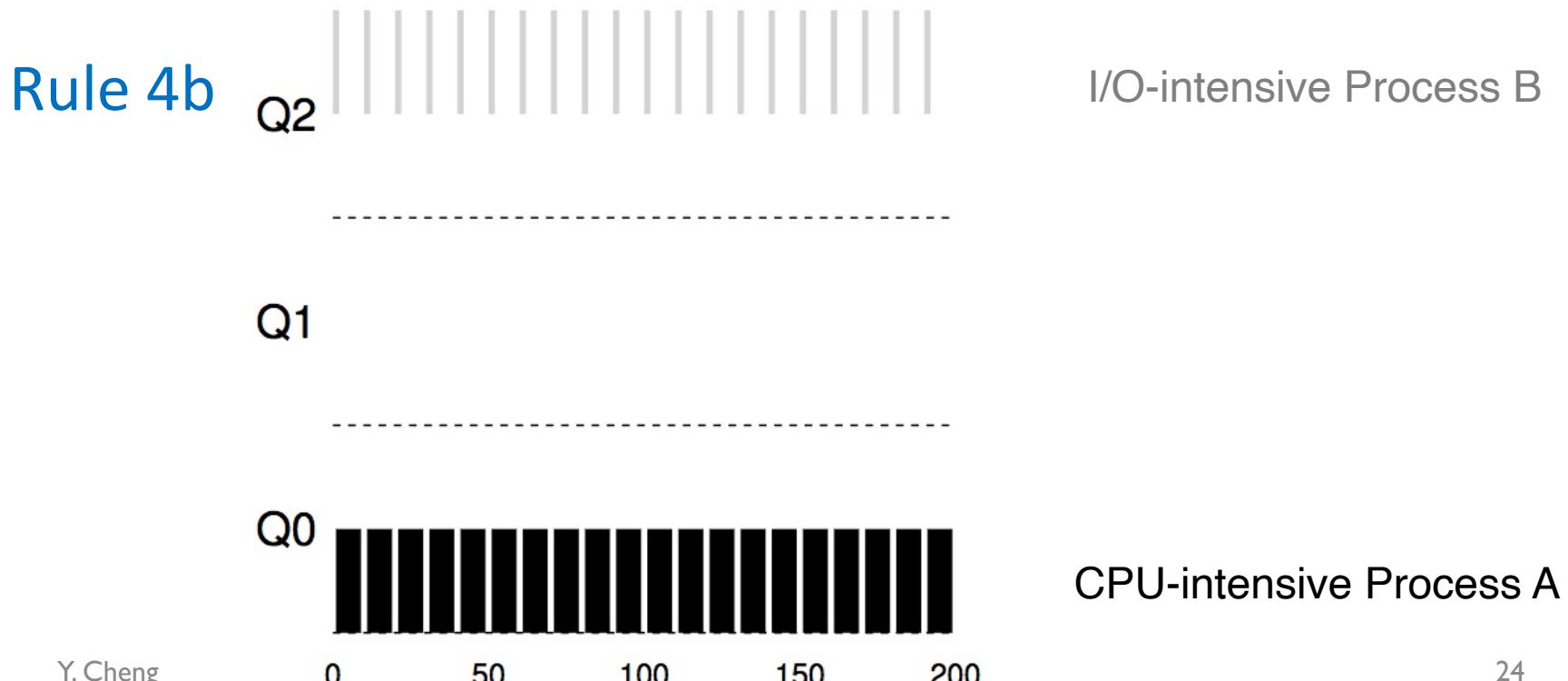
Example 2: Along Came a Short-Running Process

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



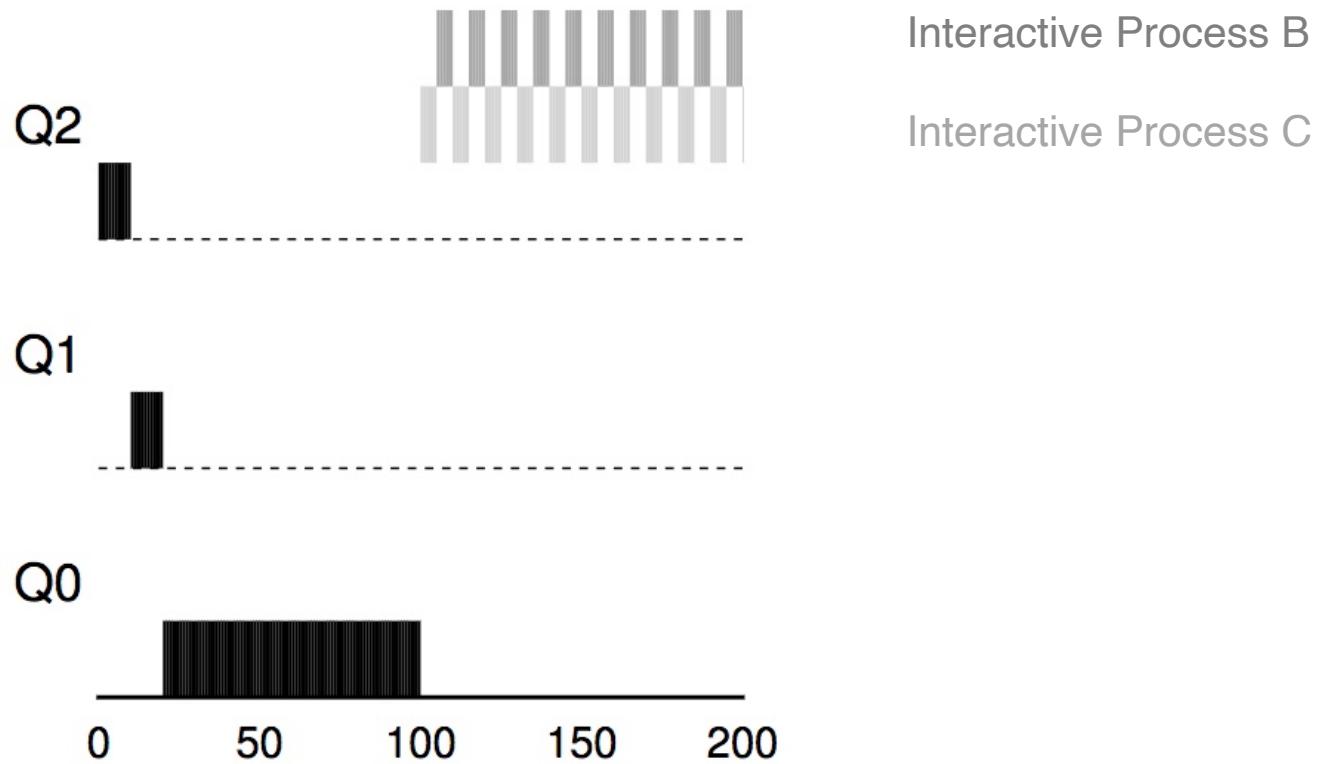
Example 3: What about I/O?

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



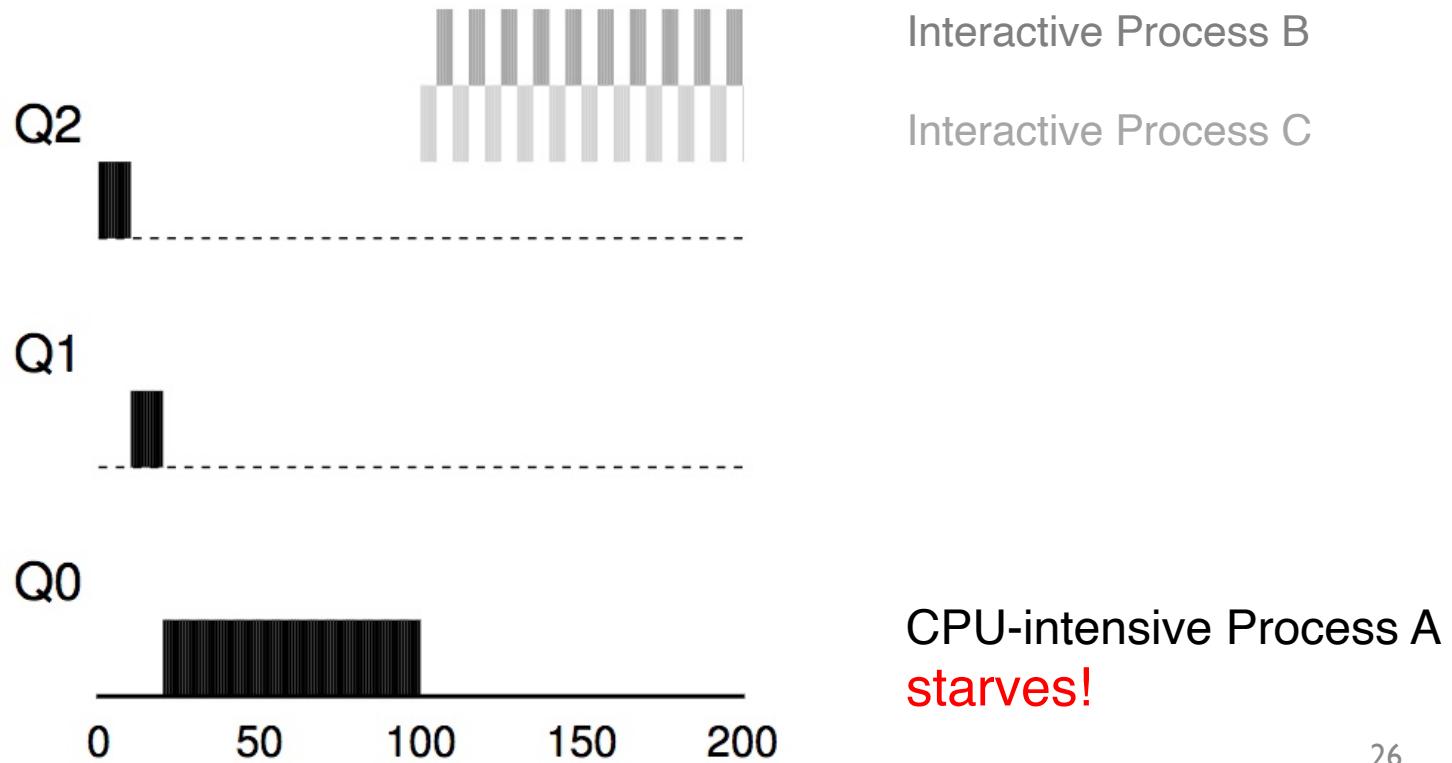
Example 4: What's the Problem?

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



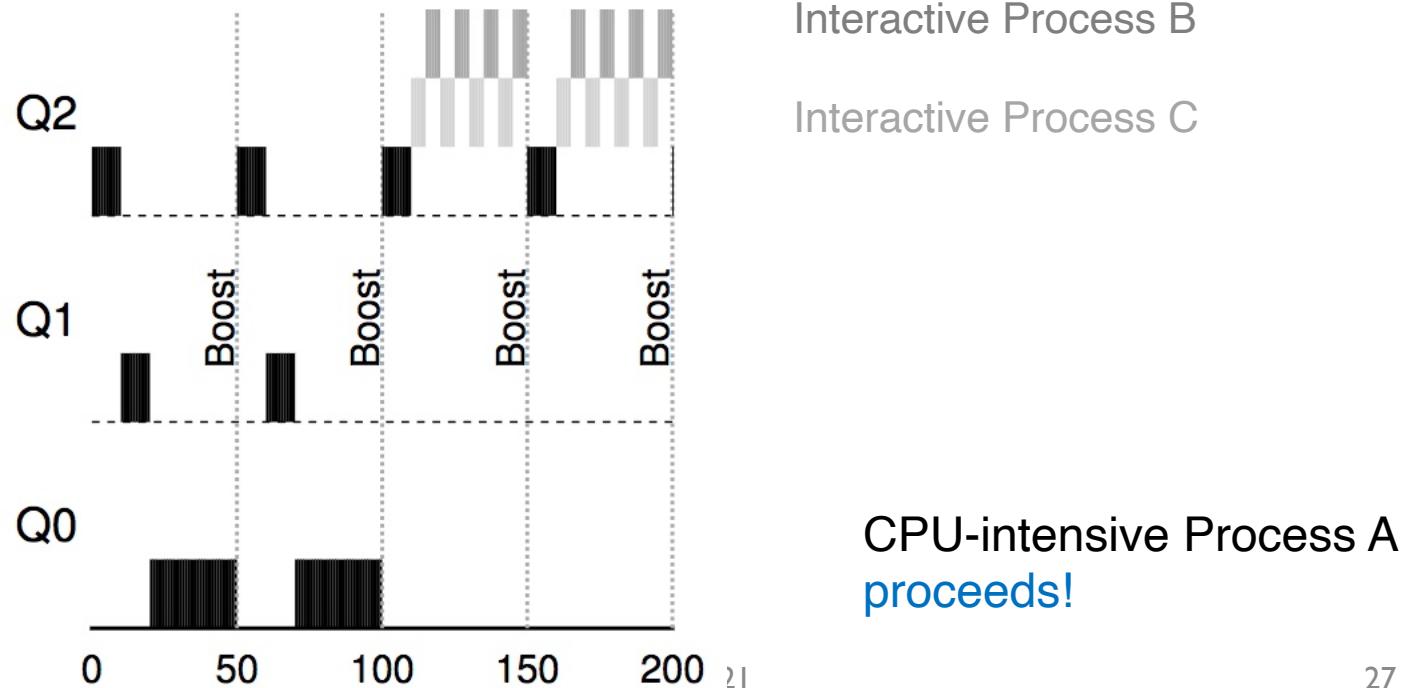
Example 4: What's the Problem?

- 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.

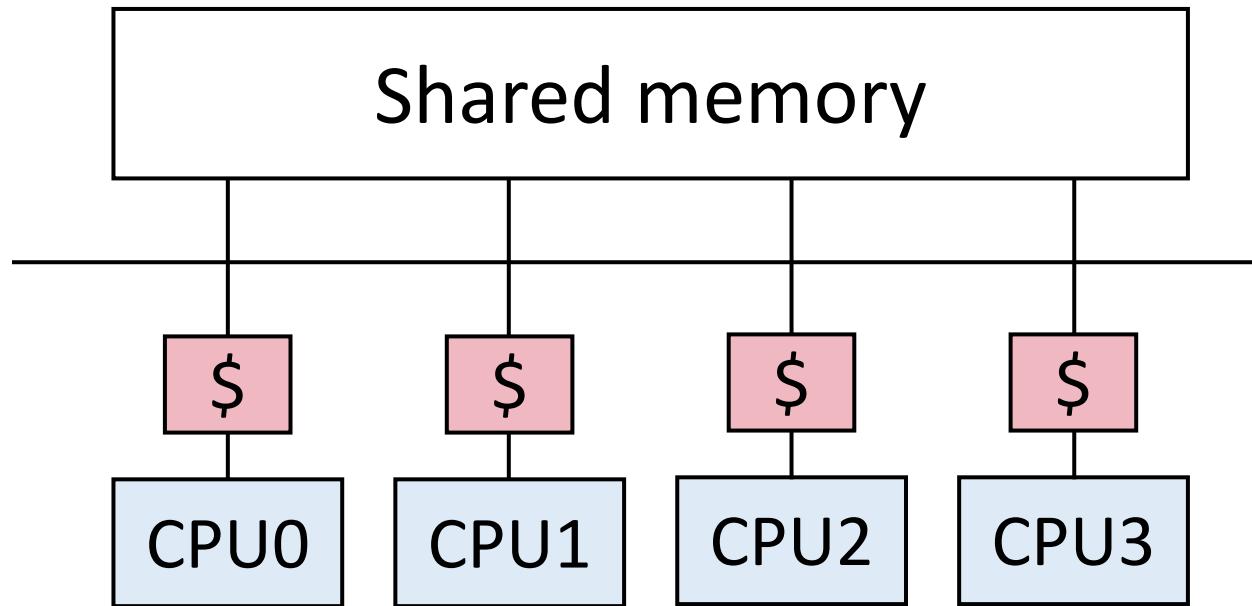


Tuning MLFQ

- 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

(Advanced) Linux Scheduling

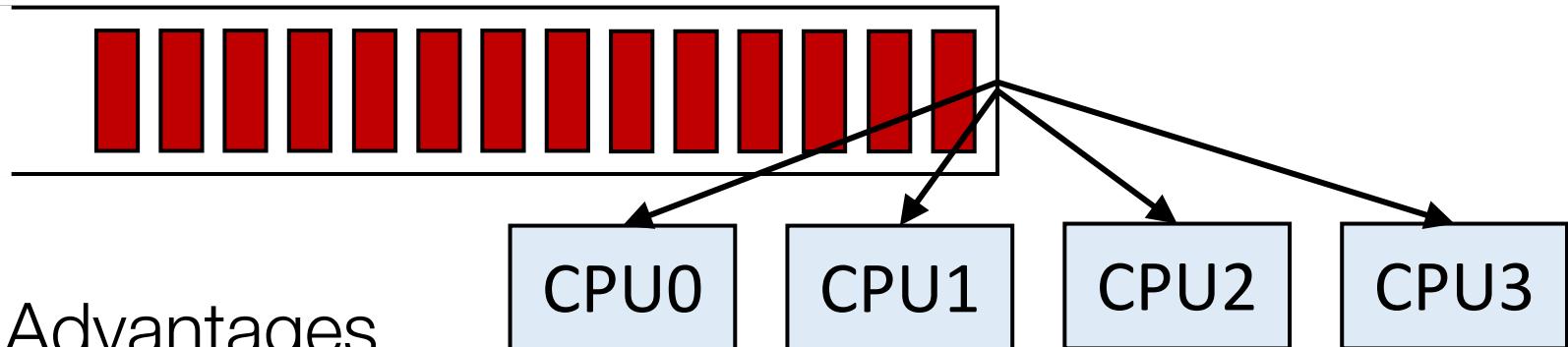
Symmetric Multiprocessing (SMP)



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

Global Queue of Processes

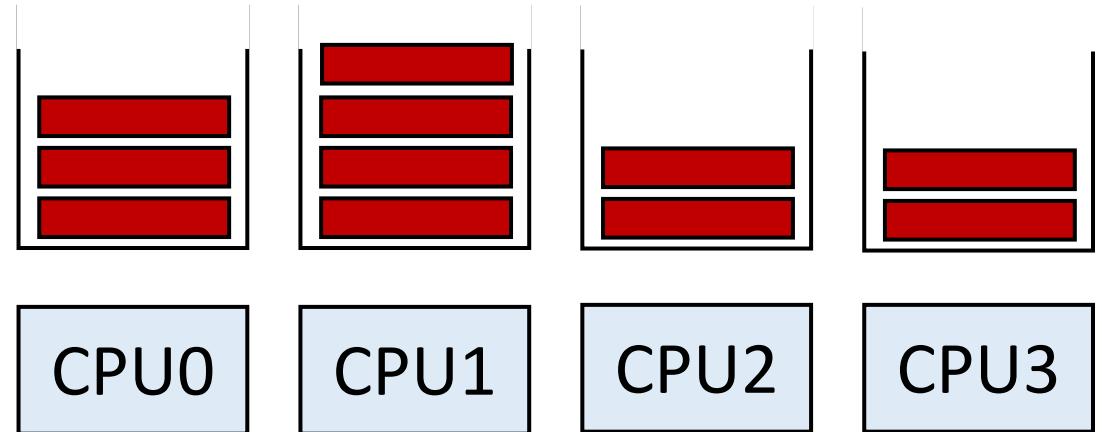
- One ready queue shared across all CPUs



- 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

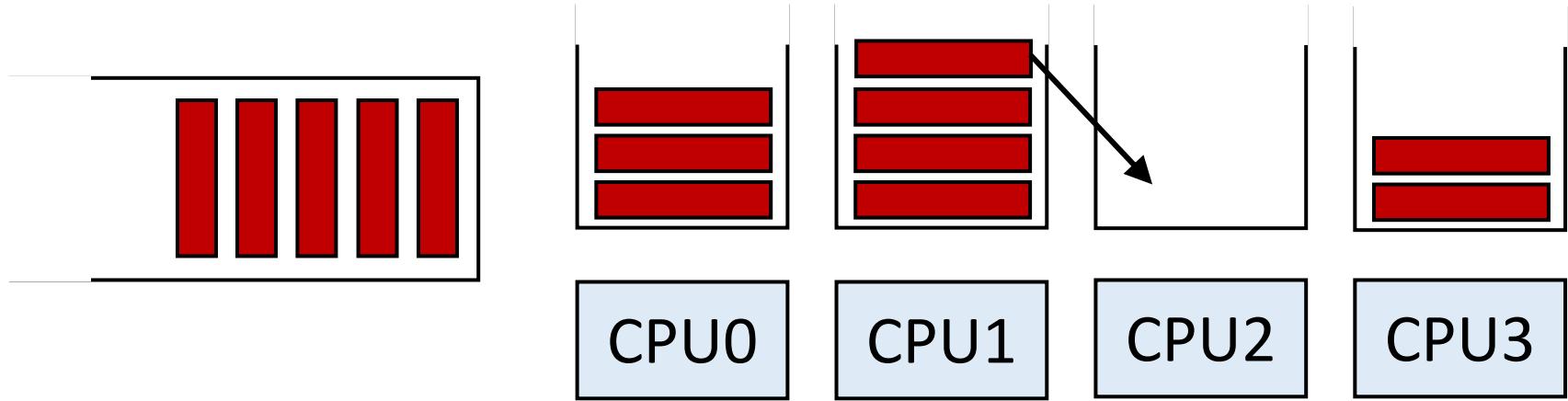
- Easy to implement
- Scalable (no contention on ready queue)
- Better cache locality

- Disadvantages

- Load imbalance (some CPUs have more processes)
 - Unfair to processes and lower CPU utilizations

Modern OSes Take 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)
-
- ```
graph LR; A[Soft real-time scheduling policies] --> B["Real-time 99"]; A --> C["Real-time 3"]; A --> D["Real-time 2"]; A --> E["Real-time 1"]
```

# 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
- 
- The diagram illustrates the Linux Multi-Level Queue with Priorities. It shows two main vertical stacks of boxes representing priority levels. The top stack, associated with 'Soft real-time scheduling policies', contains boxes labeled 'Real-time 99', '...', 'Real-time 3', 'Real-time 2', 'Real-time 1', and another '...'. The bottom stack, associated with 'Normal scheduling policies', contains boxes labeled 'Nice -20', '...', 'Nice 0', '...', and 'Nice 19'. Arrows point from each box to the right, indicating the flow of tasks. Brackets on the left side group the boxes into their respective policy categories.

# 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-prone 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)

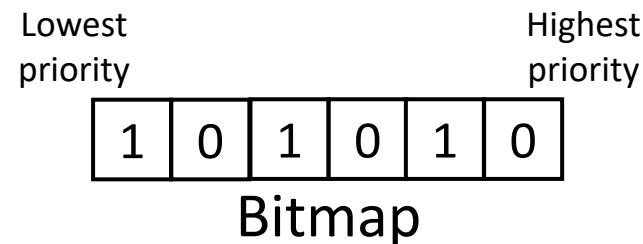
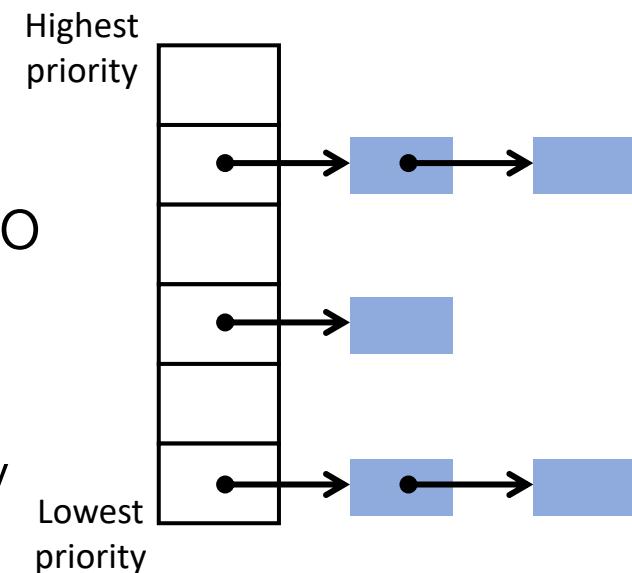
- Time is divided into **epochs**
- At the start of each epoch, scheduler assigns a priority to every process based on its behavior
  - Real-time processes have an absolute priority assigned to them, and are highest priority
  - Interactive processes have a dynamic priority assigned to them based on 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 quanta
  - 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

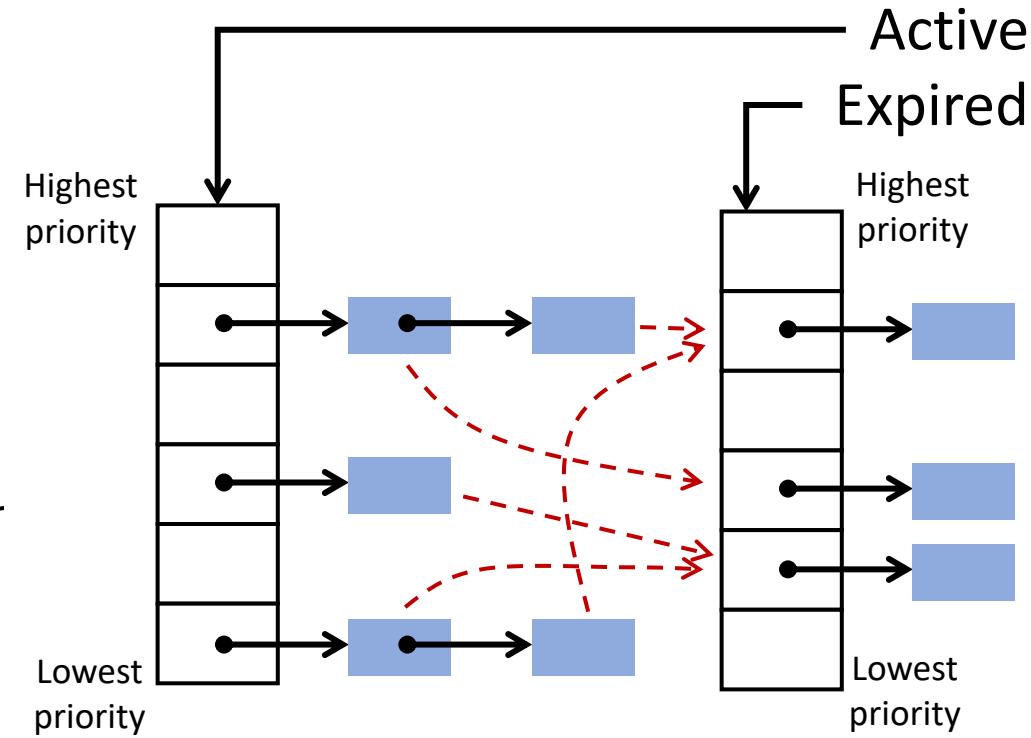
# $O(1)$ Scheduler (Linux 2.6)

- 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



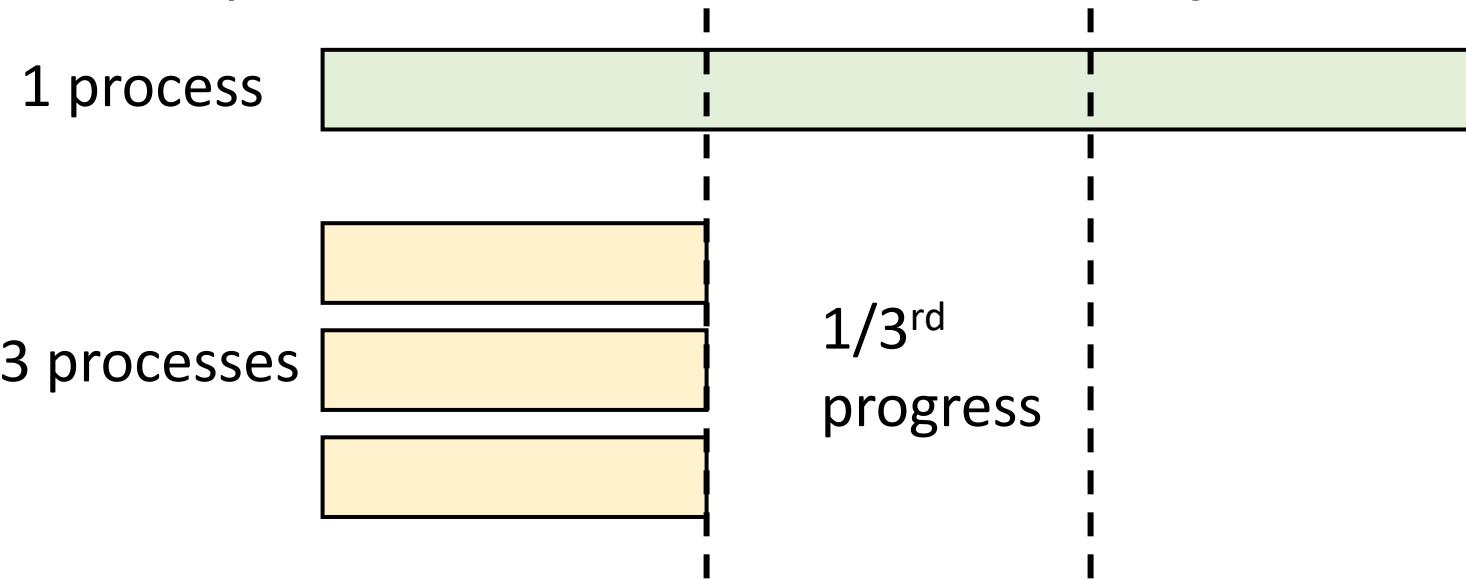
# $O(1)$ Scheduler (Linux 2.6)

- 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



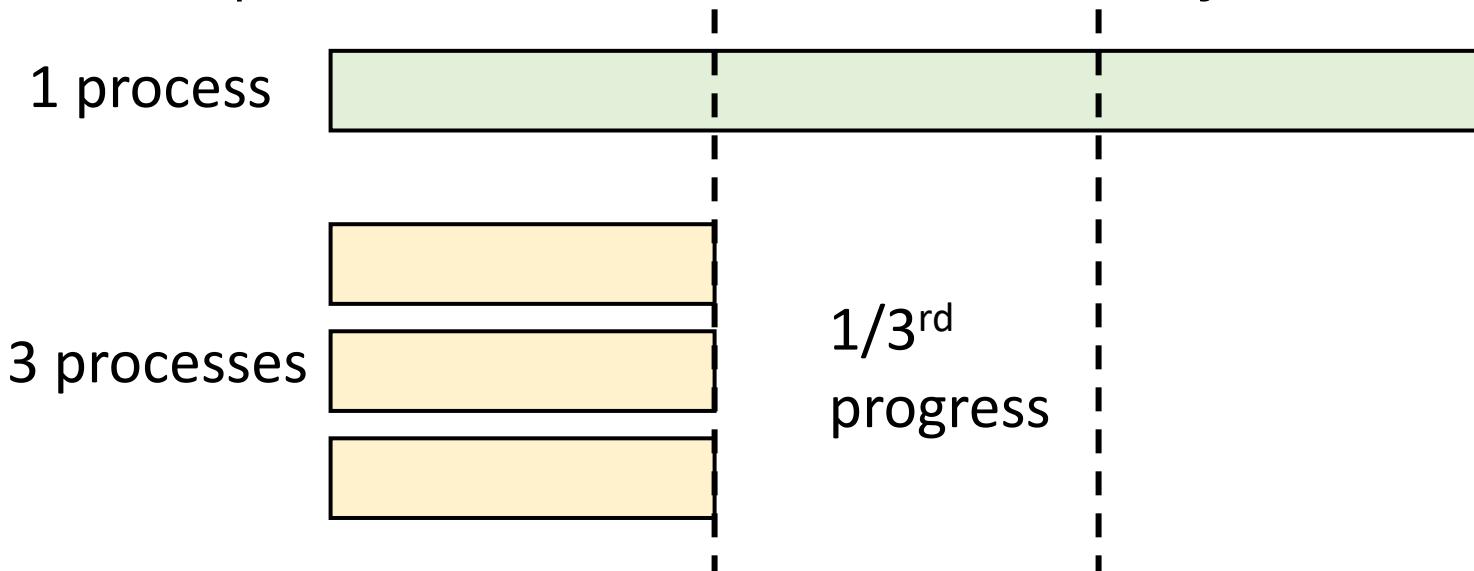
# Ideal Fair Scheduling

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



# Ideal Fair Scheduling

- Infinitesimally small time slice
- $N$  processes: each runs uniformly at  $1/N^{\text{th}}$  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 latency period  $T$ 
    - `sysctl_sched_latency`
  - Time slice for process  $P_i: T * W_i / (\text{Sum of all } W_i)$ 
    - `sched_slice()`
- Too many processes?
  - Lower bound on smallest time slice
    - `sysctl_sched_min_granularity`
  - Schedule latency = lower bound \* number of procs

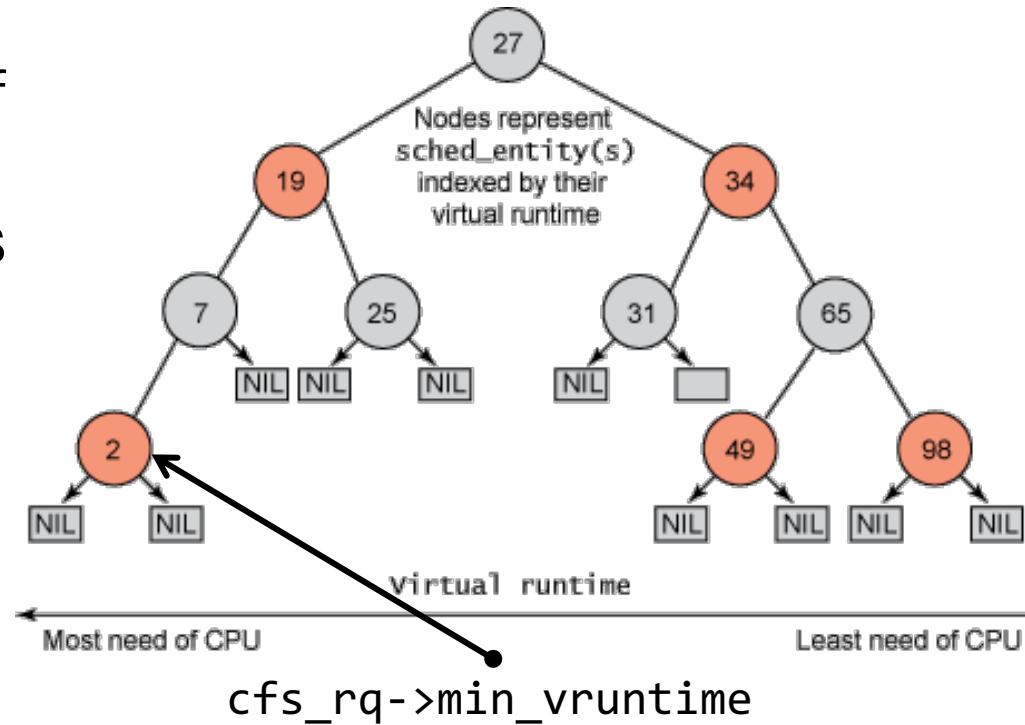
# CFS: Picking the Next Process

- Pick process w/ **minimum** weighted **vruntime** so far
  - Virtual runtime:  
 $\text{task-} \rightarrow \text{vruntime} += \text{executed time} / w_i$

# CFS: Picking the Next Process

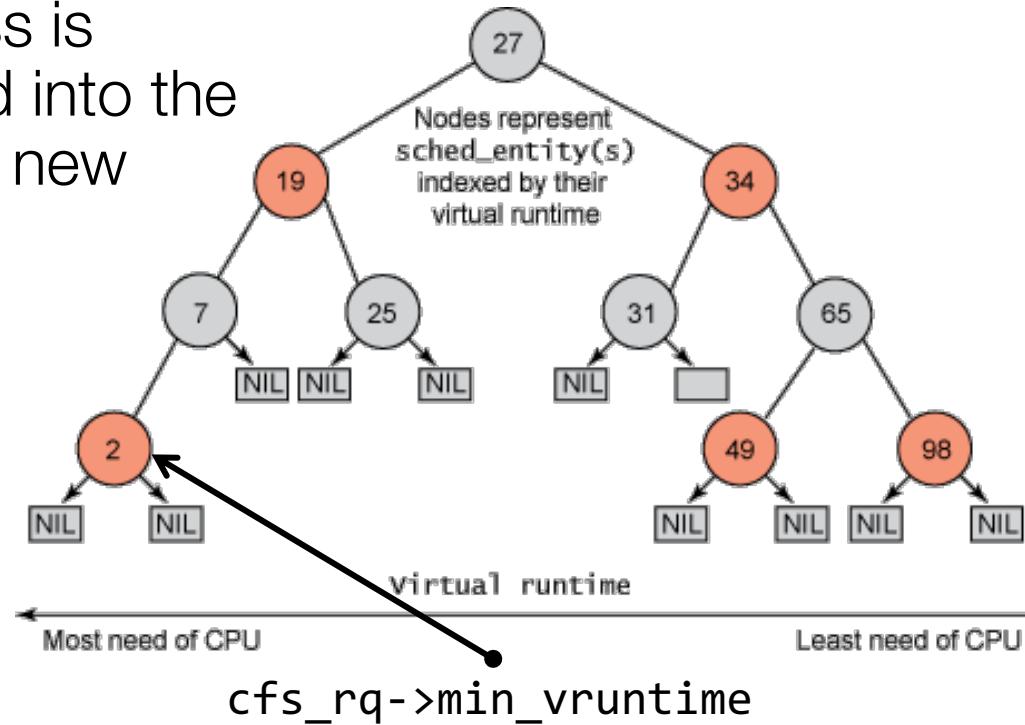
- CFS uses a red-black tree (RB tree)
    - Balanced binary search tree (BST)
    - Ordered by `vruntime` as key
    - $O(\log N)$  insertion, deletion, update;  $O(1)$ : find min

- 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

# Lottery Scheduling

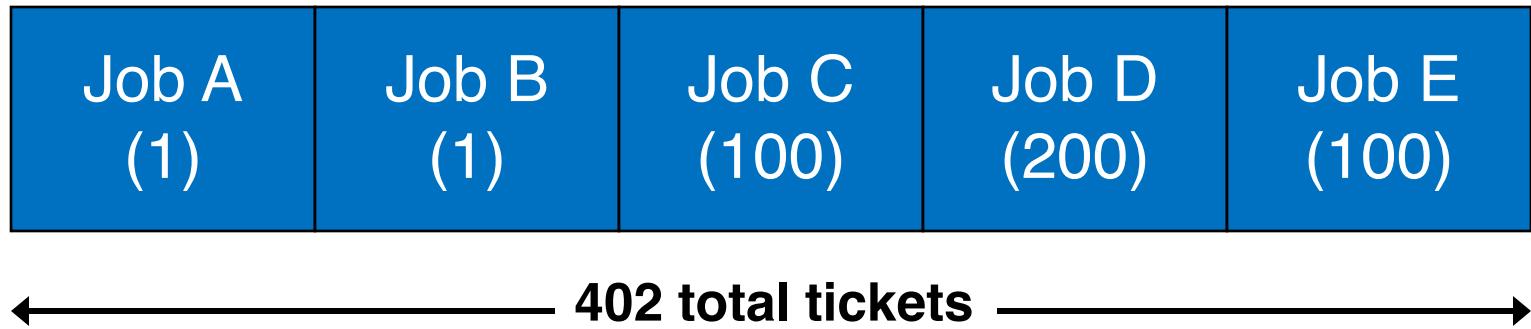
# Lottery Scheduling

- Goal: Proportional share
  - One of the fair-share schedulers
- Approach
  - Gives processes lottery tickets
  - Whoever wins runs
  - Higher priority → more tickets

# Lottery Code

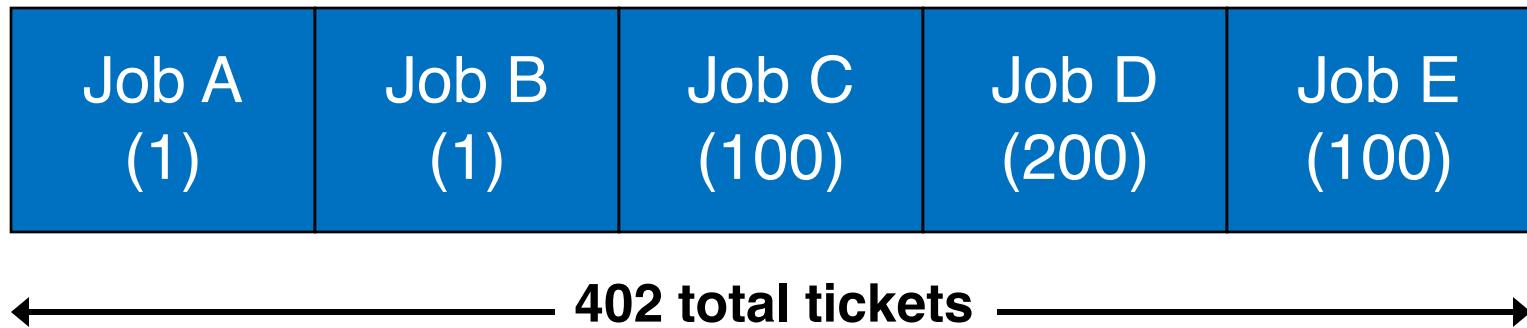
```
1 // counter: used to track if we've found the winner yet
2 int counter = 0;
3
4 // winner: use some call to a random number generator to
5 // get a value, between 0 and the total # of tickets
6 int winner = getrandom(0, totaltickets);
7
8 // current: use this to walk through the list of jobs
9 node_t *current = head;
10
11 // loop until the sum of ticket values is > the winner
12 while (current) {
13 counter = counter + current->tickets;
14 if (counter > winner)
15 break; // found the winner
16 current = current->next;
17 }
18 // 'current' is the winner: schedule it...
```

# Lottery Scheduling Example



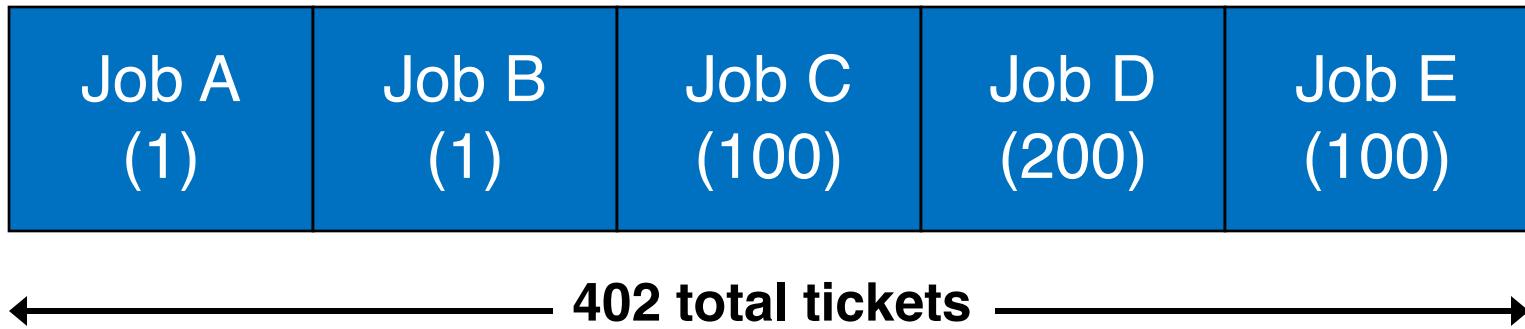
# Lottery Scheduling Example

`winner = random(402)`



# Lottery Scheduling Example

winner = 102



# Lottery Scheduling Example

winner = 102

Is 1 > 102?



|              |              |                |                |                |
|--------------|--------------|----------------|----------------|----------------|
| Job A<br>(1) | Job B<br>(1) | Job C<br>(100) | Job D<br>(200) | Job E<br>(100) |
|--------------|--------------|----------------|----------------|----------------|

← 402 total tickets →

# Lottery Scheduling Example

winner = 102

Is  $2 > 102$ ?



|              |              |                |                |                |
|--------------|--------------|----------------|----------------|----------------|
| Job A<br>(1) | Job B<br>(1) | Job C<br>(100) | Job D<br>(200) | Job E<br>(100) |
|--------------|--------------|----------------|----------------|----------------|

← 402 total tickets →

# Lottery Scheduling Example

winner = 102

Is **102 > 102?**



← **402 total tickets** →

# Lottery Scheduling Example

winner = 102

Is 302 > 102?

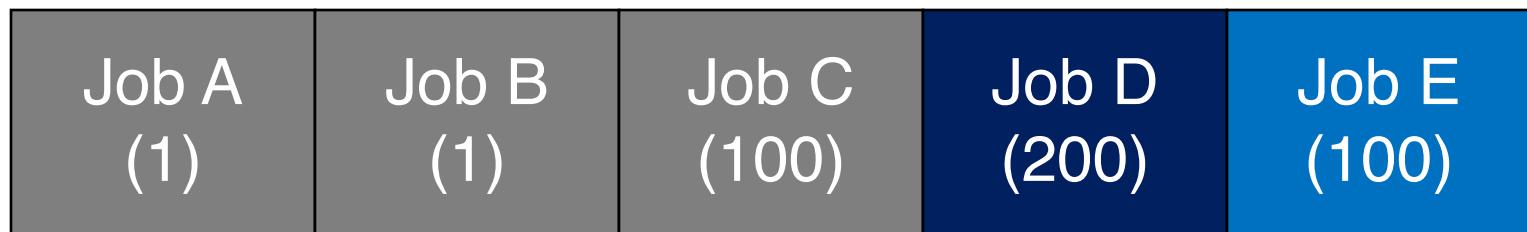


← 402 total tickets →

# Lottery Scheduling Example

winner = 102

**302 > 102**



←———— **402 total tickets** —————→

**OS picks Job D to run!**

# Other Lottery Ideas

- Ticket transfers
- Ticket currencies
- Ticket inflation
- Read more in OSTEP
- The original lottery scheduling paper:
  - [Lottery scheduling: flexible proportional-share resource management](#). Carl A. Waldspurger and William E. Weihl. USENIX OSDI'94