CPU Scheduling (Cont.)

01/26/2023

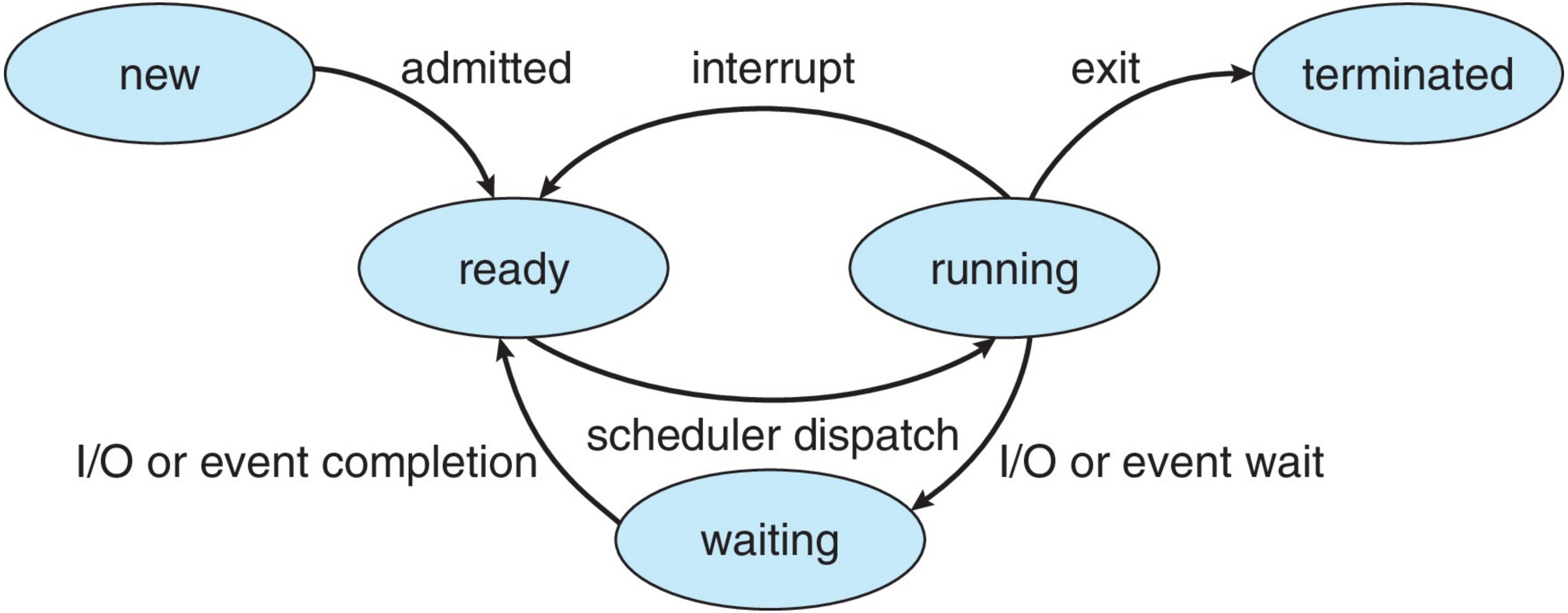
Professor Amanda Bienz

Textbook pages 214 - 217

Review

- **Want to optimize scheduler based on some metric**
- **So far, looked at two performance metrics**
- $T_{turnaround} = T_{completion} - T_{arrival}$
- $T_{response} = T_{firstrun} - T_{arrival}$

Review

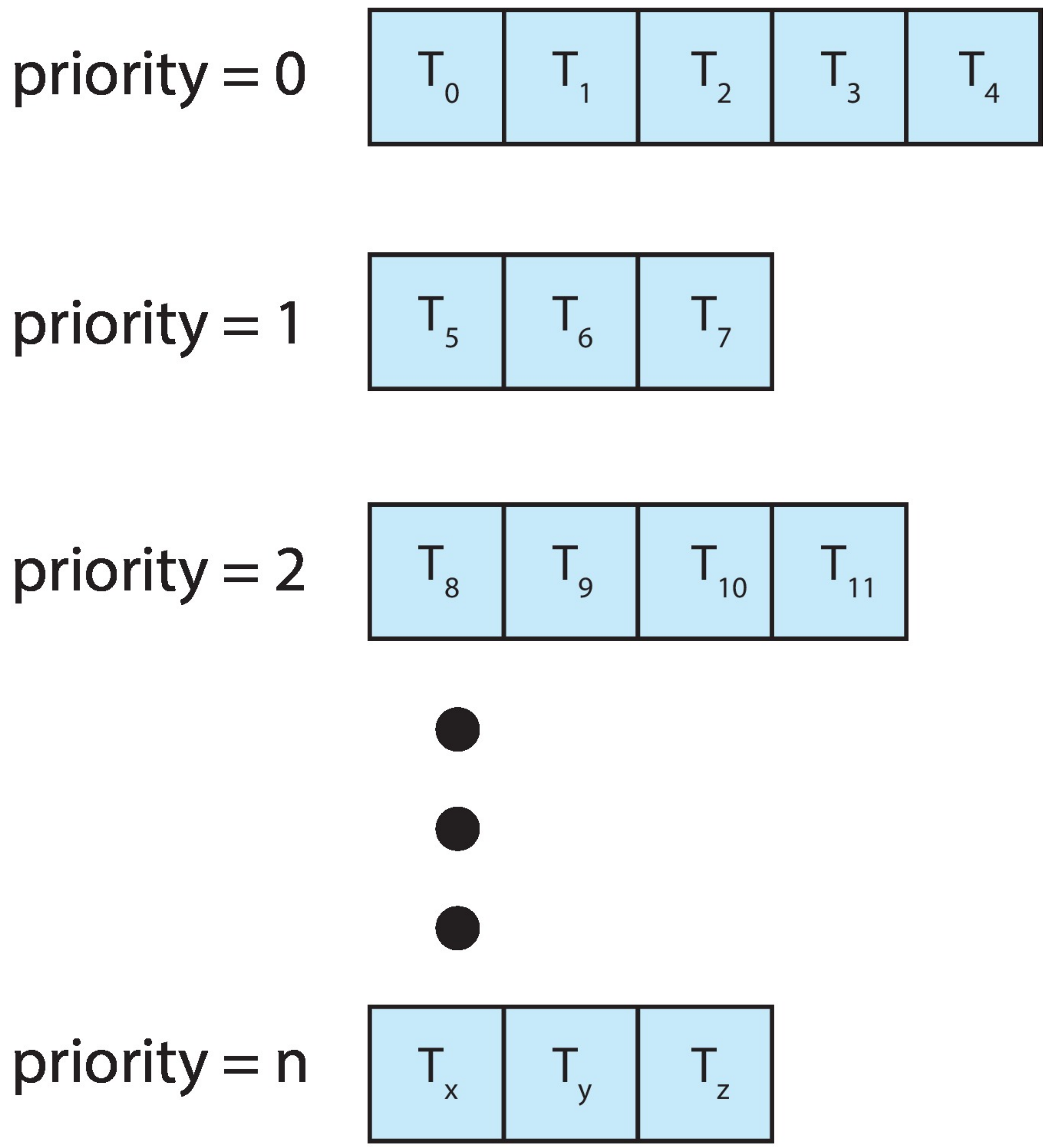

new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Review

- For systems with mixed workloads, there is generally not an easy single metric to optimize (i.e. turnaround time or response time)
- General-purpose systems rely on heuristic schedulers that try to balance the qualitative performance of the system
- Question : What's wrong with round robin?
- Aside : How hard is 'optimal' scheduling for an arbitrary performance metric?

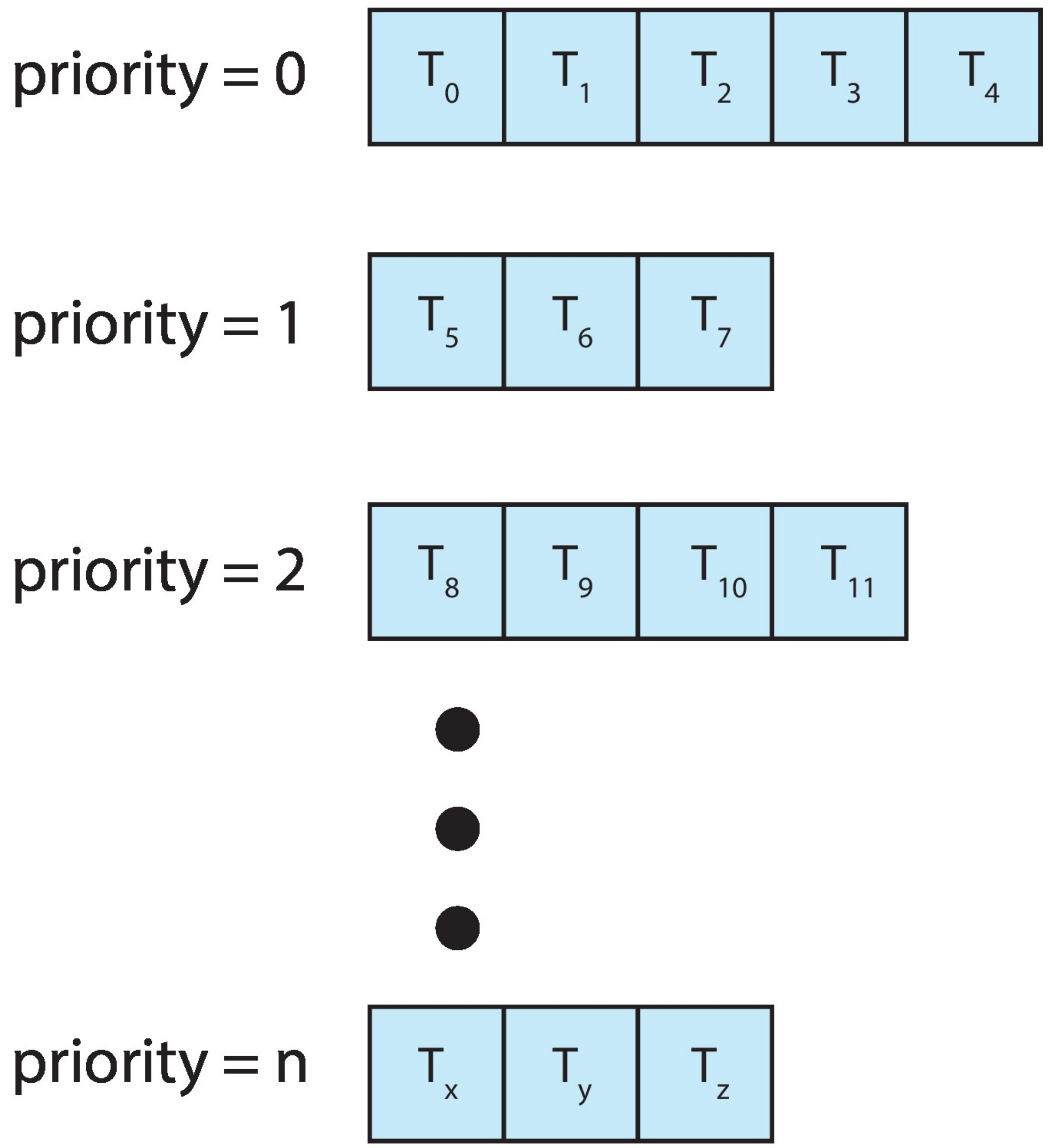
Multilevel Queue

- Using priority scheduling, have separate queue for each priority
- Schedule process in highest-priority queue



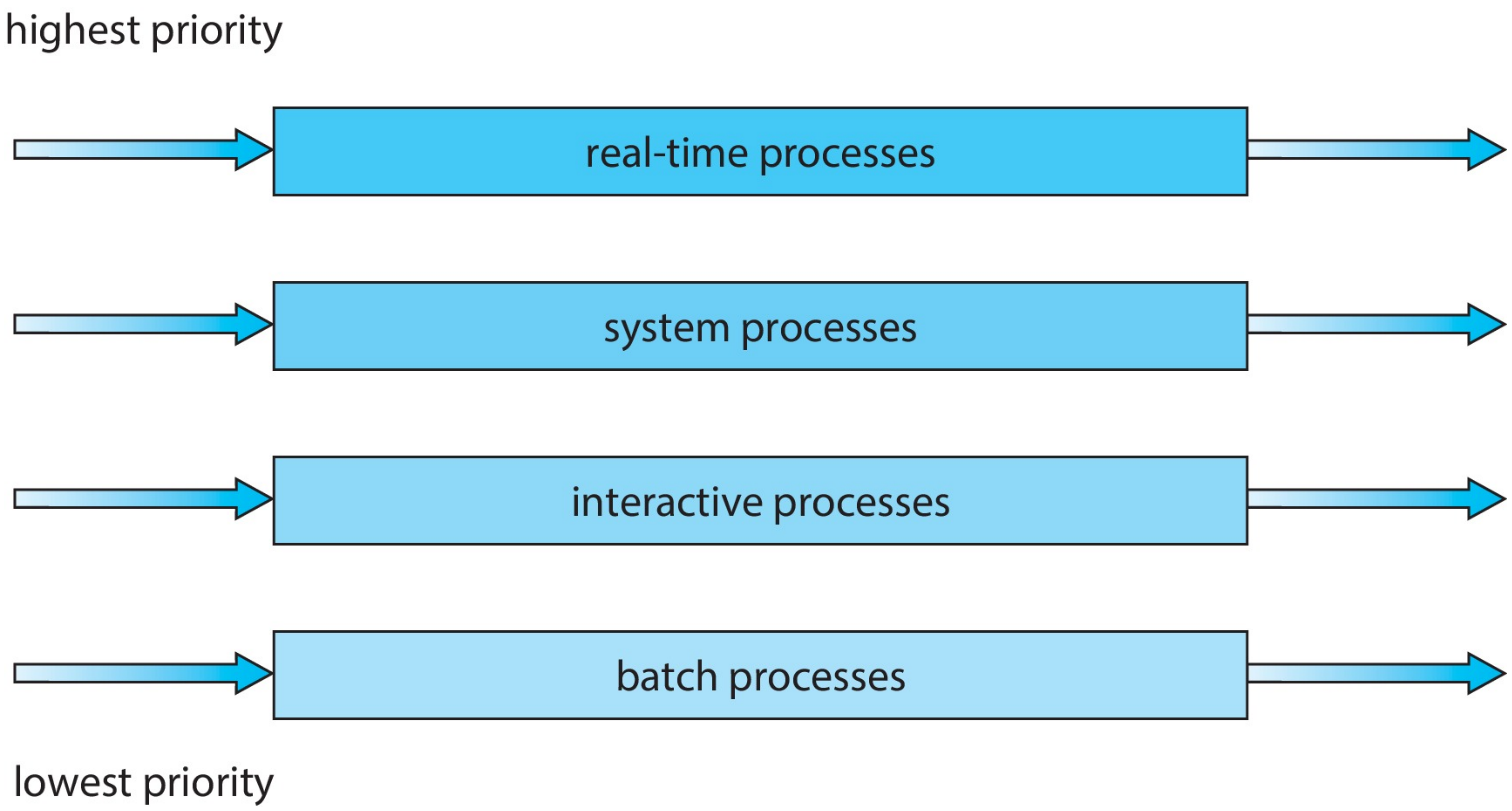
Multilevel Queue

- Similar to priority scheduling, every process with same priority could be executed with round-robin



Multilevel Queue

- Prioritization based upon process type

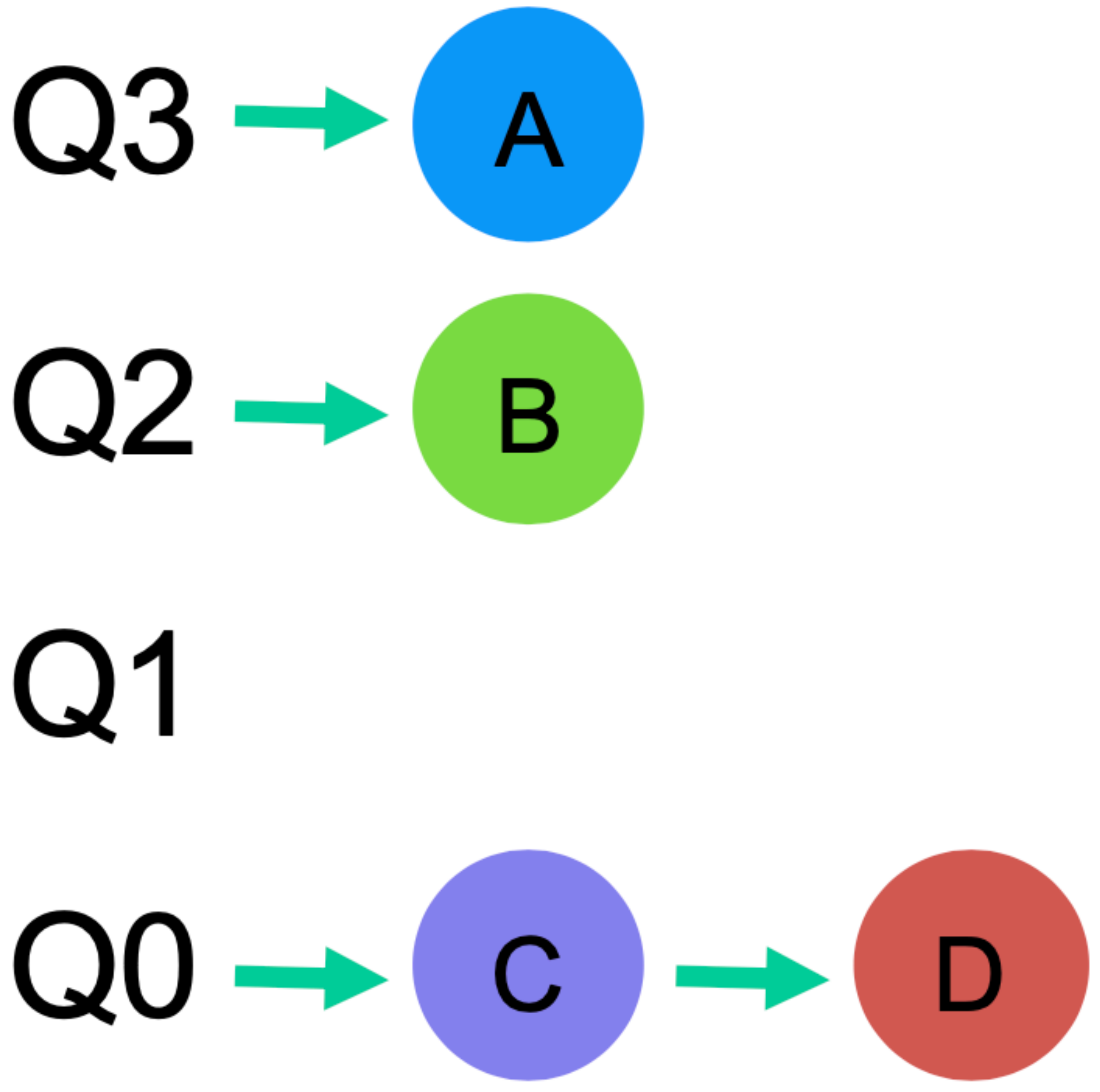


Multi-Level Feedback Queue (MLFQ)

- Goal : general purpose scheduling
- Must support two job types with distinct goals
  - “Interactive” programs care about response time
  - “Batch” programs care about turn-around time
- Approach : multiple levels of round-robin:
  - Each level has higher priority than lower levels, and preempts them

Basic Mechanism : Multiple RR Queues

- **Rule 1 : If priority(A) > priority(B), A runs**
- **Rule 2 : If priority(A) == priority(B), A & B run in RR**
- Multi-level
- Policy : how to set priority?
- Approach 1 : “nice” command
- Approach 2 : history “feedback”



MLFQ : Basic Rules

- MLFQ varies priority of job based on its observed behavior
- Example:
  - A job repeatedly relinquishes the CPU while waiting for IO
  - A job uses the CPU intensively for long periods of time

MLFQ : Basic Rules

- MLFQ varies priority of job based on its observed behavior
- Example:
  - A job repeatedly relinquishes the CPU while waiting for IO : **Keep its priority high**
  - A job uses the CPU intensively for long periods of time : **Reduce its priority**

MLFQ : How To Change Priority

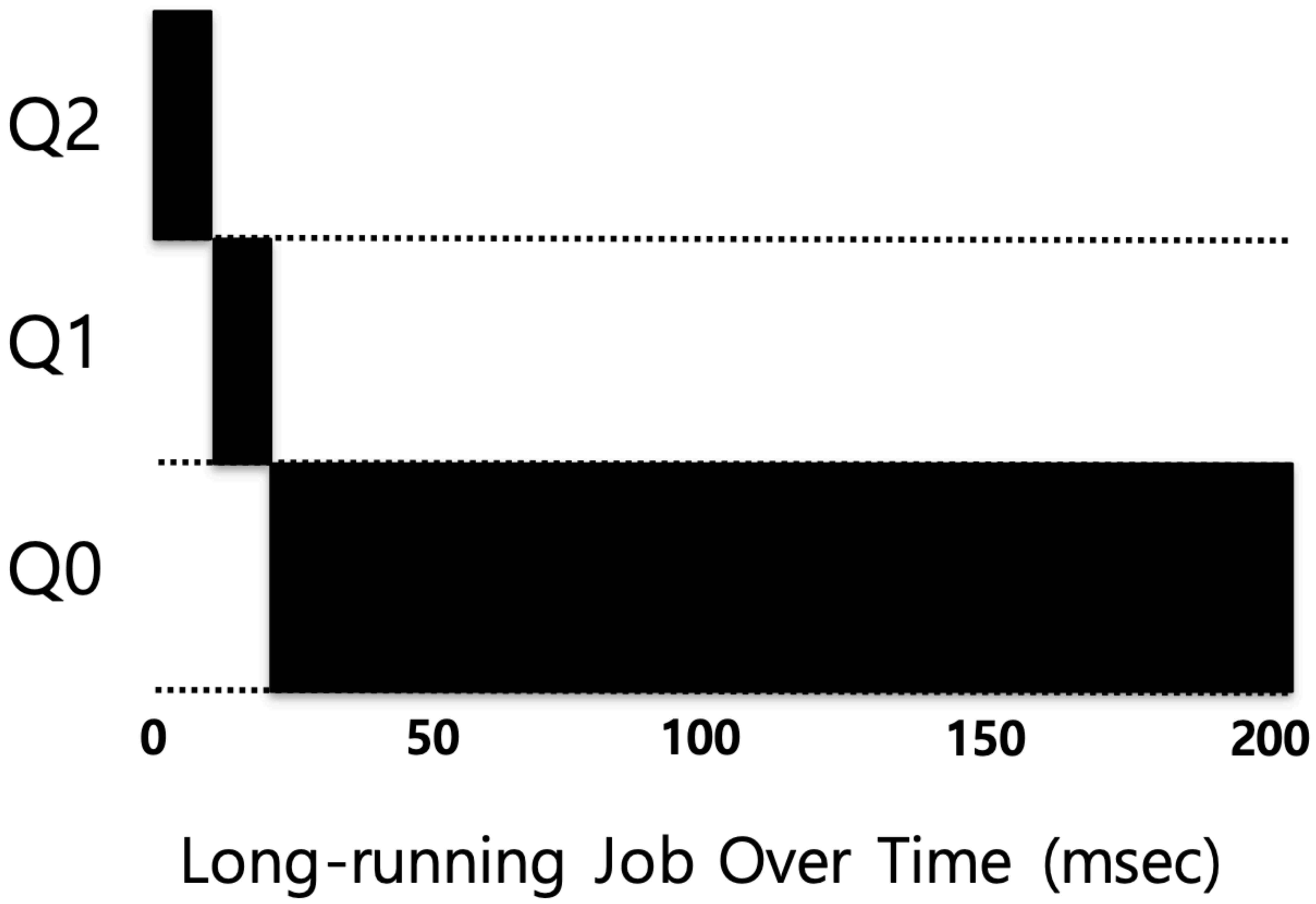
- Priority adjustment algorithm:
  - Rule 3 : When a job enters the system, it is placed at the highest priority
  - Rule 4a : If a job uses up an entire time slice while running, its priority is reduced (moves down a queue)
  - Rule 4b : If a job gives up the CPU before the time slice is up, it stays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

Example 1: A Single Long-Running Job

- A three-queue scheduler, using time slice 10ms
- Job A is compute intensive, and takes 200ms

Example 1: A Single Long-Running Job

- A three-queue scheduler, using time slice 10ms
- Job A is compute intensive, and takes 200ms

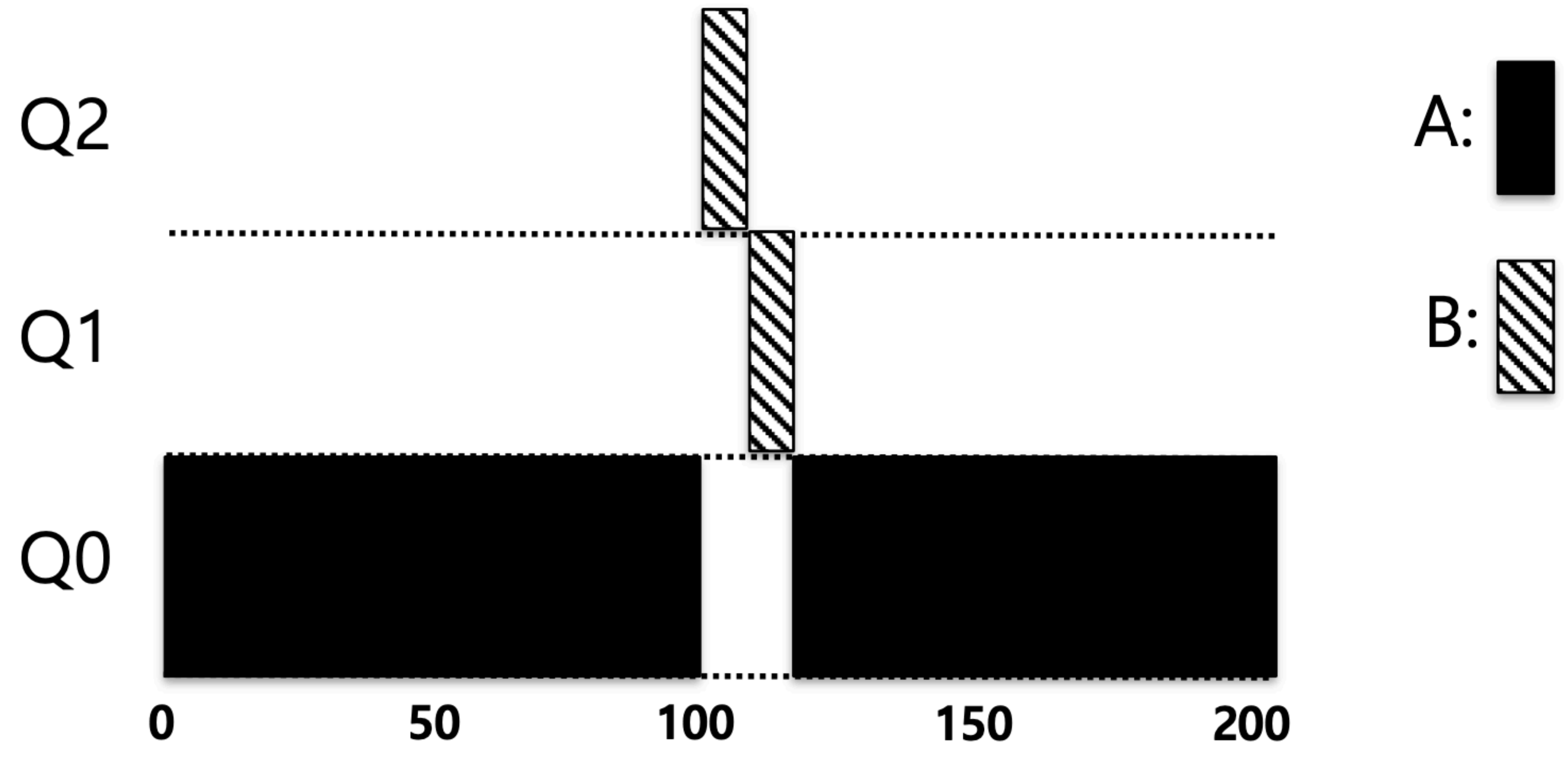


Example 2 : Along Comes a Short Job

- Job A : A long-running CPU-intensive job
- Job B : A short-running interactive job (20ms)
- A has been running for some time… B arrives at time T = 100

Example 2 : Along Comes a Short Job

- Job A : A long-running CPU-intensive job
- Job B : A short-running interactive job (20ms)
- A has been running for some time… B arrives at time T = 100



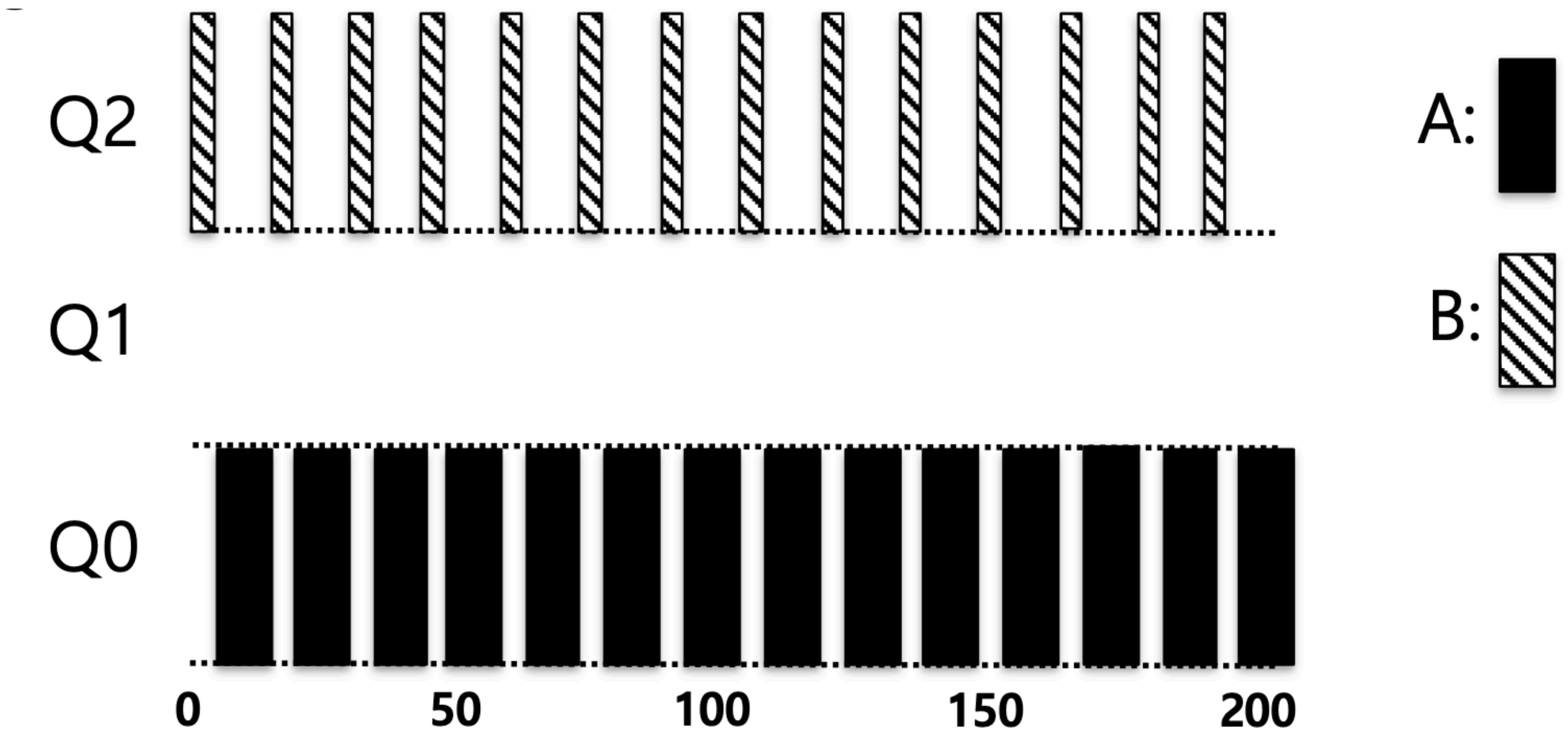
Along Came An Interactive Job (msec)

Example 3: What about I/O?

- Job A : long-running CPU-intensive job
- Job B: interactive job that needs CPU only for 1ms before performing I/O

Example 3: What about I/O?

- Job A : long-running CPU-intensive job
- Job B: interactive job that needs CPU only for 1ms before performing I/O



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

Problems with Basic MLFQ

- Can you think of a problem with this type of scheduling?

Problems with 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 a time slice, issue an I/O operation
  - The job gains higher percentage of CPU time
- A program may change its behavior over time
  - CPU bound process become I/O bound process

Solutions?

- How can we fix these problems?

Solutions?

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

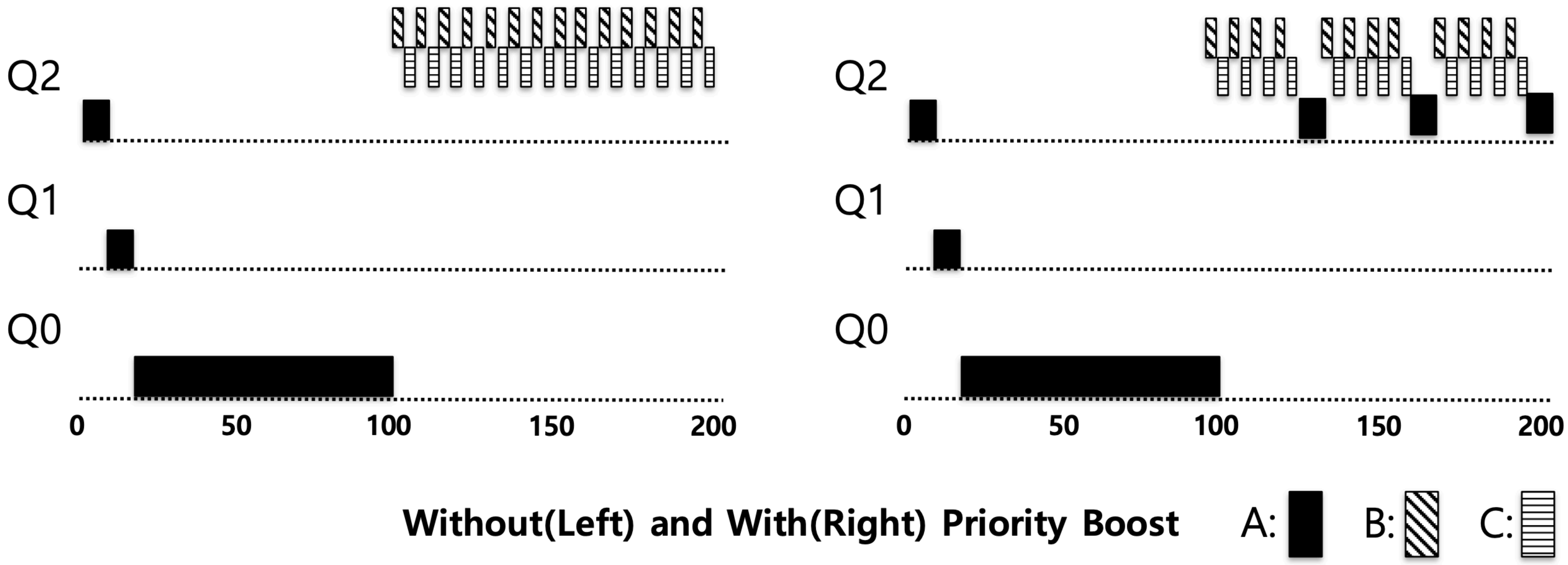
Priority Boost

- A long-running job A
- At time T=100, two short-running interactive jobs show up (B and C)
- B and C each run on CPU for 10ms, followed by 10ms of I/O
- What does this look like without a priority boost?

Priority Boost

- A long-running job A
- At time T=100, two short-running interactive jobs show up (B and C)
- B and C each run on CPU for 10ms, followed by 10ms of I/O
- Assume there is a priority boost every 50ms

Priority Boost



Without(Left) and With(Right) Priority Boost

Better Accounting

- How would you prevent gaming the scheduler?

Better Accounting

- Instead of job only moving down a queue when it has used its entire time slice at one time:
  - Updated 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 (moving down a queue)

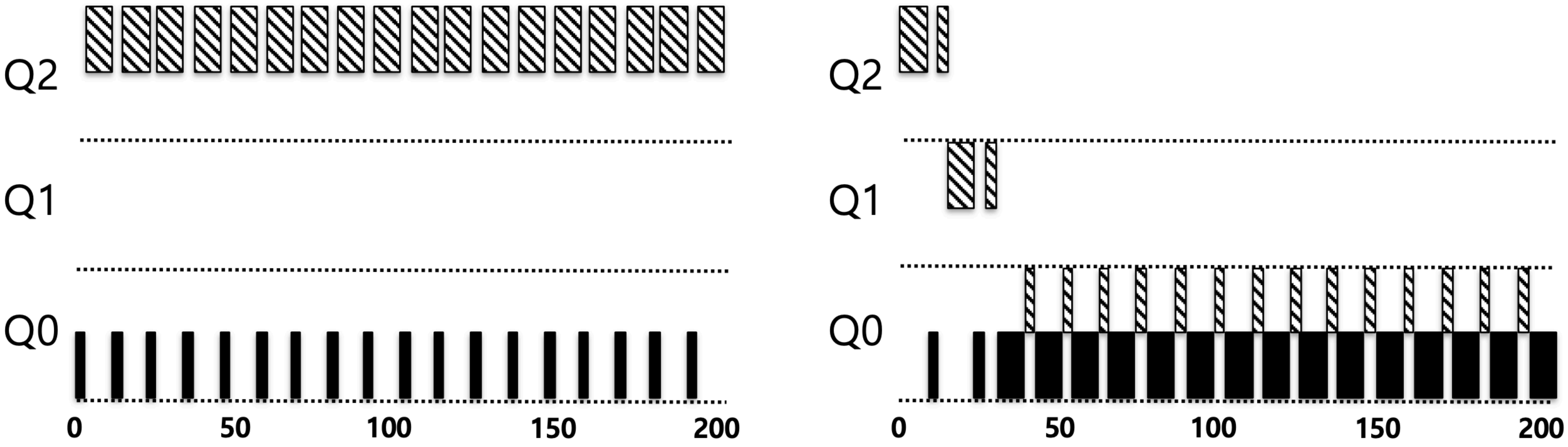
Better Accounting

- Job A : long-running CPU-intensive job
- Job B: uses CPU for first 9ms, followed by 1ms of I/O
- Assume 10ms time slices
- Priority increased if I/O before time slice finishes

Better Accounting

- Job A : long-running CPU-intensive job
- Job B: uses CPU for first 9ms, followed by 1ms of I/O
- Assume 10ms time slices
- Priority reduced as soon as utilized time-slice worth of CPU time

Better Accounting



Without(Left) and With(Right) Gaming Tolerance

Solaris MLFQ Implementation

- For the time-sharing scheduling class
  - 60 Queues
  - Slowly increasing time-slice length
    - Highest priority : 20ms
    - Lowest priority : a few hundred ms
  - Priorities boosted around every 1 second or so
  - Pages 242-244

MLFQ Summary (refined)

- Rule 1 : If priority(A) > priority(B), A runs
- Rule 2 : If priority(A) == priority(B), A&B run in RR
- Rule 3 : When a job enters the system, it is placed in highest priority queue
- Rule 4 : Once a job uses up its time allotment in a given queue, its priority is reduced
- Rule 5 : After some time period S, all jobs are moved back to topmost queue

Fairness in Scheduling

- Question : Is optimizing for response time or turnaround time fair?
- What does fairness mean in terms of scheduling?
- “Fair” scheduling : each process gets a certain percentage of CPU time
  - Think of the scheduling algorithms we’ve discussed so far (RR, SJF, MLFQ)
  - Do any of these result in fair scheduling?
- Very different metric than performance (turnaround time, response time)

Expressing Shares in Proportional Scheduling

- Let's talk about how we express a process's "share" of a resource
- How does a "share" change over time?

Basic Concept

- Tickets
  - Represent the share of a resource that a process should receive
  - The percent of tickets represents its share of the system resource in question
- Example:
  - Process A has 75 tickets : receive 75% of the CPU
  - Process B has 25 tickets : receive 25% of the CPU
- Reasons to liken to tickets:
  - Exchangeable, transferable, inflatable
- How do we turn tickets into CPU time?

Approach 1: Lottery Scheduling

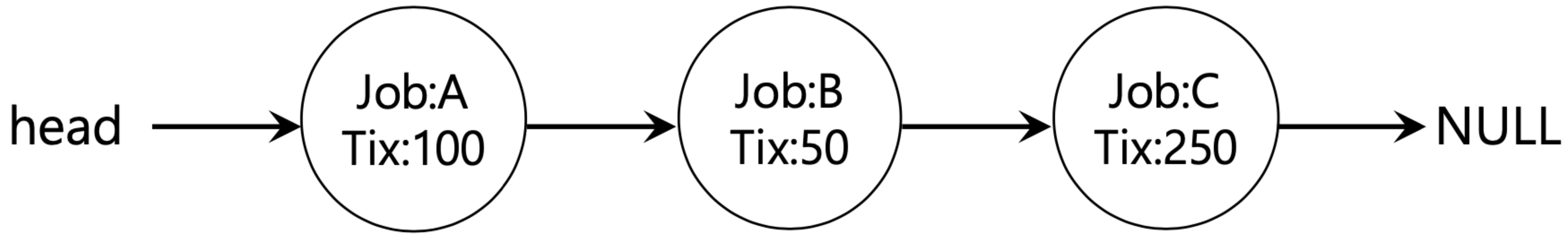
- The scheduler randomly picks a winning ticket
  - Load the state of that winning process and run it
- Example: There are 100 tickets
  - Process A has 75 tickets : 0 to 74
  - Process B has 25 tickets : 75 to 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Resulting scheduler:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

Implementation

- Example: There are three processes, A, B, and C
  - Keep the processes in a linked list



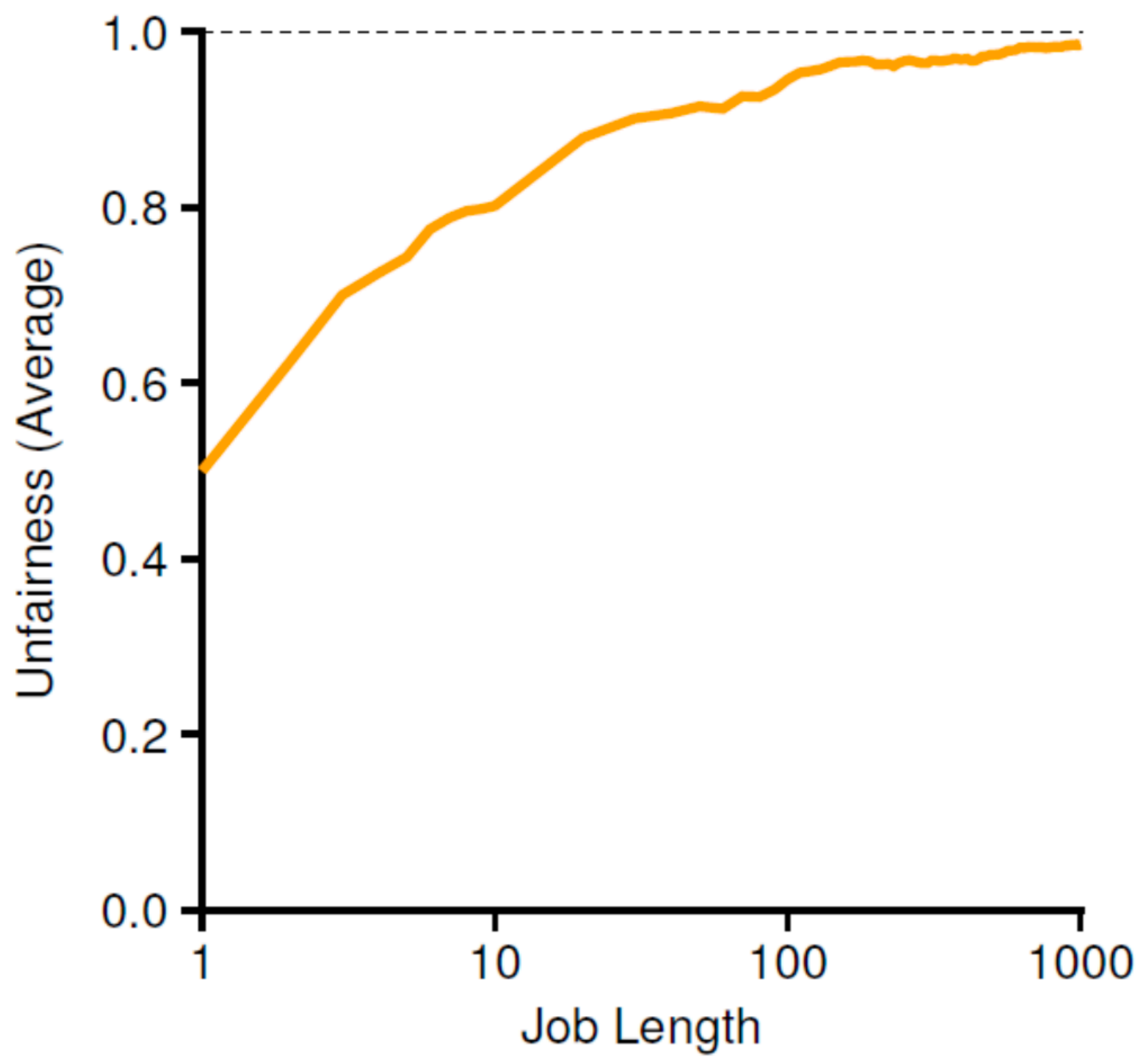

```c
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
        counter = counter + current->tickets;
        if (counter > winner)
                break; // found the winner
        current = current->next;
}
// 'current' is the winner: schedule it...
```

Implementation (Cont.)

- U : “unfairness factor” metric
  - The time the first job completes divided by the time that the second job completes
- Example :
  - There are two jobs that arrive at the same time, each has runtime 10ms
    - First job finishes at time 10ms
    - Second job finishes at time 20ms
  - U = 10/20 = 0.5
  - U will be close to 1 when both jobs finish at nearly the same time

Lottery Fairness Study

- There are two jobs. Each job has the same number of tickets (100)



When the job length is not very long,
average unfairness can be quite severe.

Attempt 2: Stride Scheduling

- **Stride of each process** : (A large number) / (the number of tickets of the process)
  - Example : A large number = 10,000
    - Process A has 100 tickets : stride of A is 100
    - Process B has 50 tickets : stride of B is 200
- Each process holds its own counter, called a **pass value**
- After a process runs, pass value incremented by stride
  - Pick the process to run that has the lowest pass value

Stride Scheduling Example

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Stride Scheduling Example

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	…

If new job enters with pass value 0,
It will monopolize the CPU!

Proportional Share Scheduling

- Not used in most user-facing operating systems - why not?
- How do you assign “shares” on your laptop?
- How does I/O fit into this?
- Some tricky implementation problems (unfairness in lottery, log(N) queue insertion in stride)
- Used mainly for coarse-grain scheduling (i.e. virtual machines in shared data centers)

Some additional schedulers

- O(1) Scheduler : https://en.wikipedia.org/wiki/O(1)_scheduler
- Completely Fair Scheduler : https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- BFS (Brain “Hug” Scheduler) : https://en.wikipedia.org/wiki/Brain_Fuck_Scheduler
- https://www.cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

In Class Questions

- Q0 is now highest priority
- Ignore bottom two bullet points of top box
- Q1 : Only moves down a priority IF it uses up entire time slice at one time. Moves down each queue (not just highest priority)
- Q2 : Moves down a priority AS SOON as it uses up one time slice, not necessarily at once
- Process A runs while process B is doing I/O