CS 423
Operating System Design

https://cs423-uiuc.github.io

Tianyin Xu
tyxu@illinois.edu

* Thanks Adam Bates for the slides.

Scheduling

- A forever topic in Computer Systems and Life
  - Uniprocessor:100 threads in the ready queue – which one to run next?
  - Multiprocessor: 400 threads in the ready queues of four cores – which one to run next on which core?
  - Cluster: 1000 MapReduce jobs – which one to run on which machine and on which core?
  - Datacenters: 10000 user request – which one to run on which datacenter on which cluster on which machine?

More complexity

- Jobs/requests are not created equal.
  - Some are more important than the others
- Jobs/requests could have deadlines
  - Finishing late means nothing but wasting resources.
- Jobs/requests have constraints
  - Affinity is important – same node and same PCIe switch for GPUs
- Workloads could be very different.

Scheduling

- Always an active research topic
  - Everyone wants run more jobs with less resources
- In this class, we are going to focus on the simplest setup – a uniprocessor

What Are Scheduling Goals?

- What are the goals of a scheduler?
- Scheduling Goals:
  - Generate illusion of concurrency
  - Maximize resource utilization (e.g., mix CPU and I/O bound processes appropriately)
  - Meet needs of both I/O-bound and CPU-bound processes
    - Give I/O-bound processes better interactive response
    - Do not starve CPU-bound processes
  - Support Real-Time (RT) applications

Definitions

- Task/Job
  - Something that needs CPU time: a thread associated with a process or with the kernel…
  - … a user request, e.g., mouse click, web request, shell command, …
- Latency/response time
  - How long does a task take to complete?
- Throughput
  - How many tasks can be done per unit of time?

Definitions

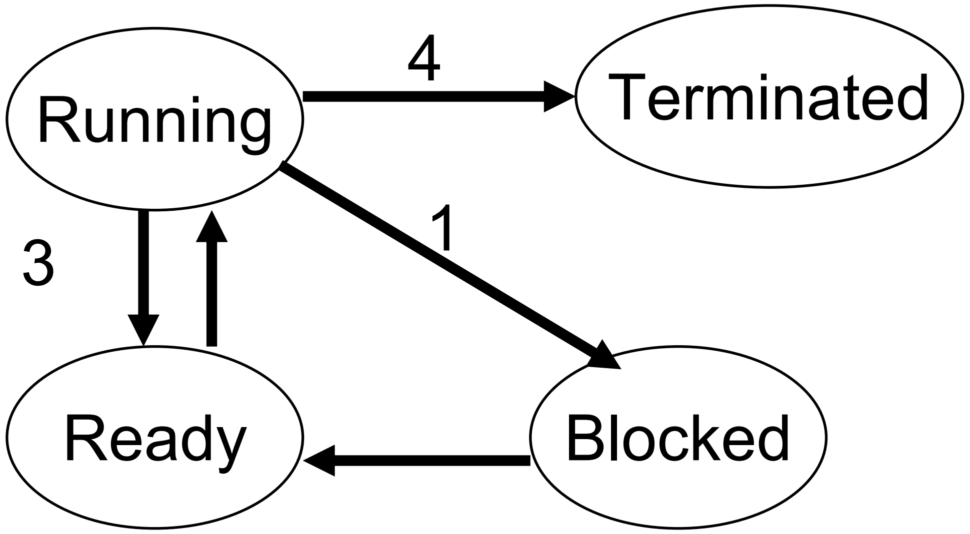
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Predictability
  - How consistent is the performance over time?
- Starvation
  - A task ‘never’ receives the resources it needs to complete
  - Not very fair : - (

Definitions

- Workload
  - Set of tasks for system to perform
- Work-conserving
  - Resource is used whenever there is a task to run
  - For non-preemptive schedulers, work-conserving is not always better

Definitions

- Non-preemptive scheduling:
  - The running process keeps the CPU until it voluntarily gives up the CPU
    - process exits
    - switches to blocked state
    - 1 and 4 only (no 3)



- Preemptive scheduling:
  - The running process can be interrupted and must release the CPU (can be forced to give up CPU)

Definitions

- Scheduling algorithm
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output
  - Only preemptive, work-conserving schedulers to be considered

First In First Out (FIFO)

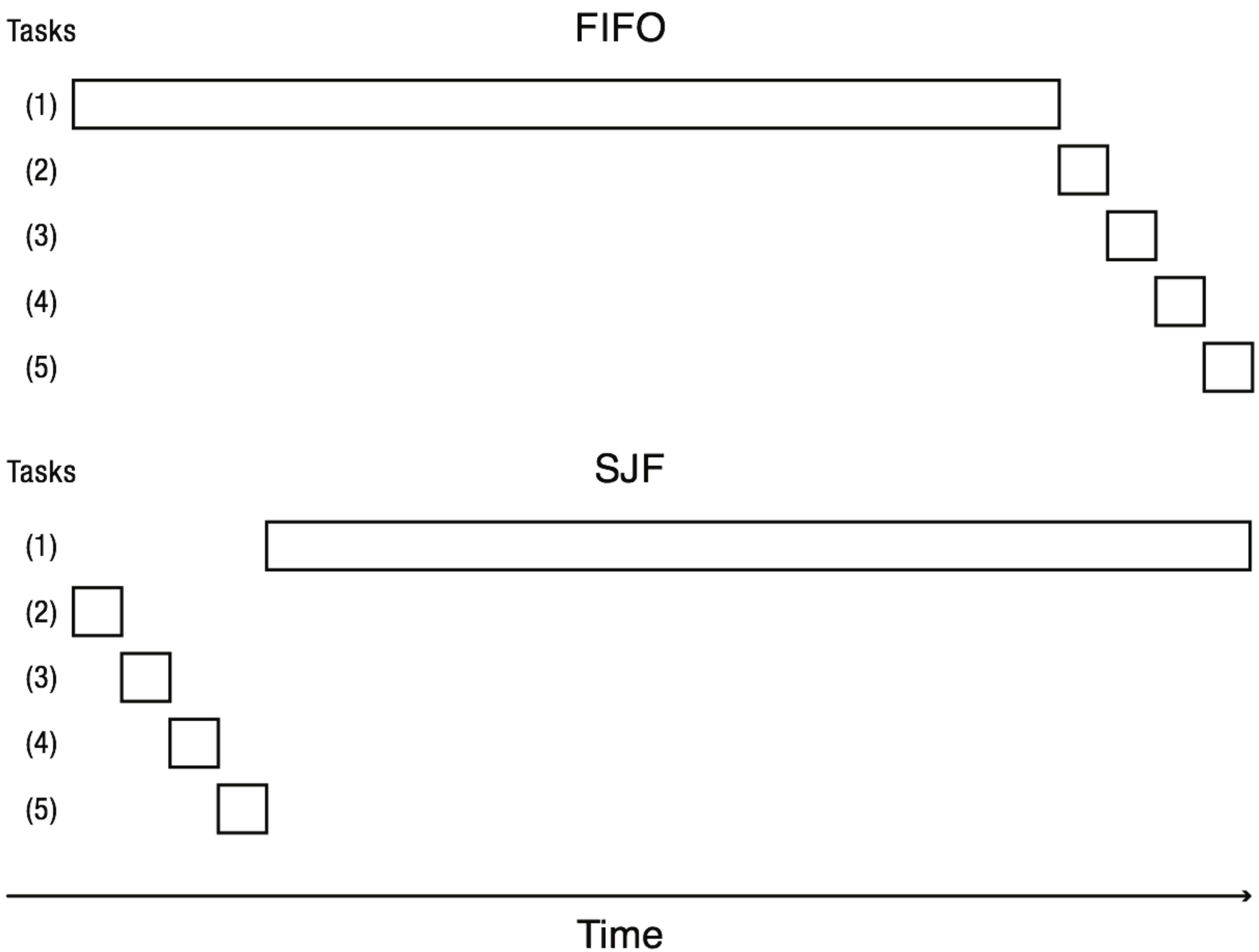
- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor
- On what workloads would FIFO be particularly bad?

Shortest Job First (SJF)

- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
  - Which completes first in SJF? Next?

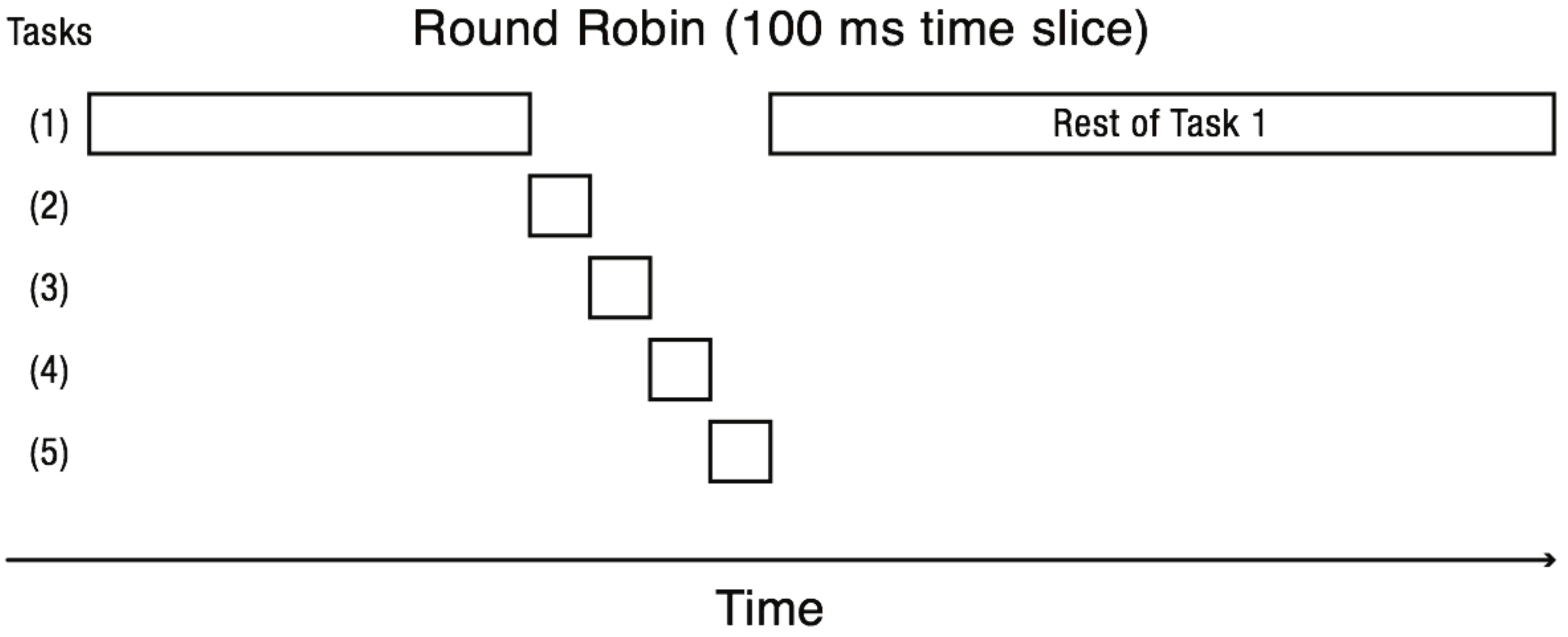
FIFO vs. SJF



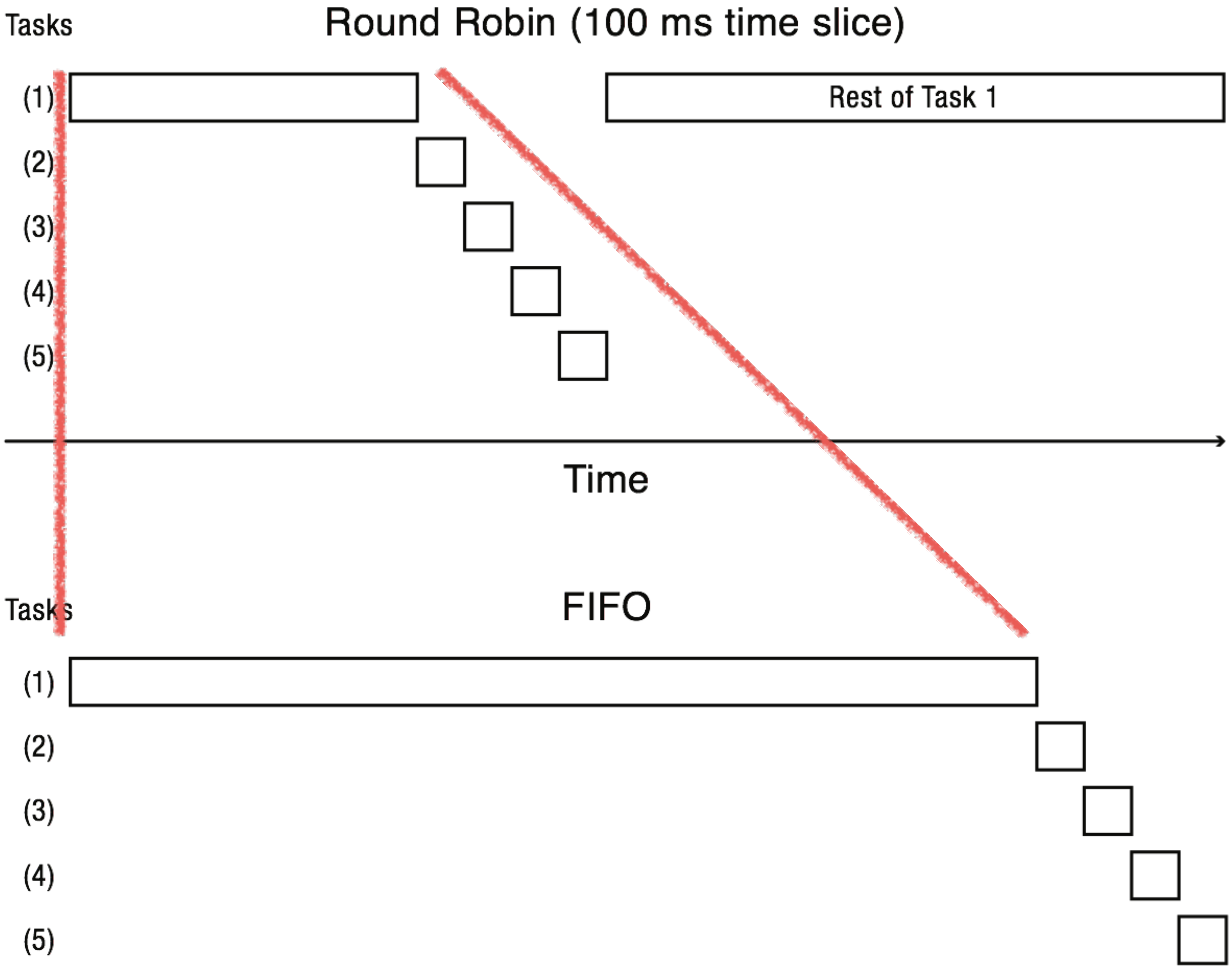
Round Robin (RR)

- Each task gets resource for a fixed period of time (time quantum)
    - If task doesn't complete, it goes back in line
- Characteristics of scheduler change depending on the time quantum size
  - What if time quantum is too short?
    - One instruction?
  - What if time quantum is too long?
    - Infinite?

Round Robin

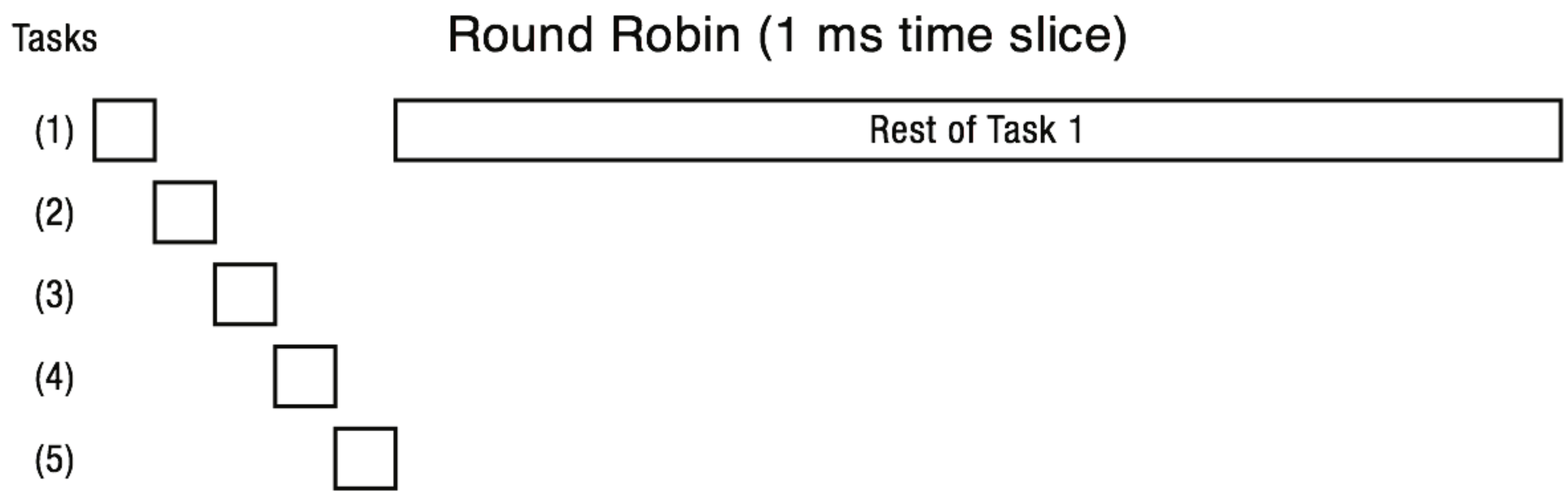


Round Robin

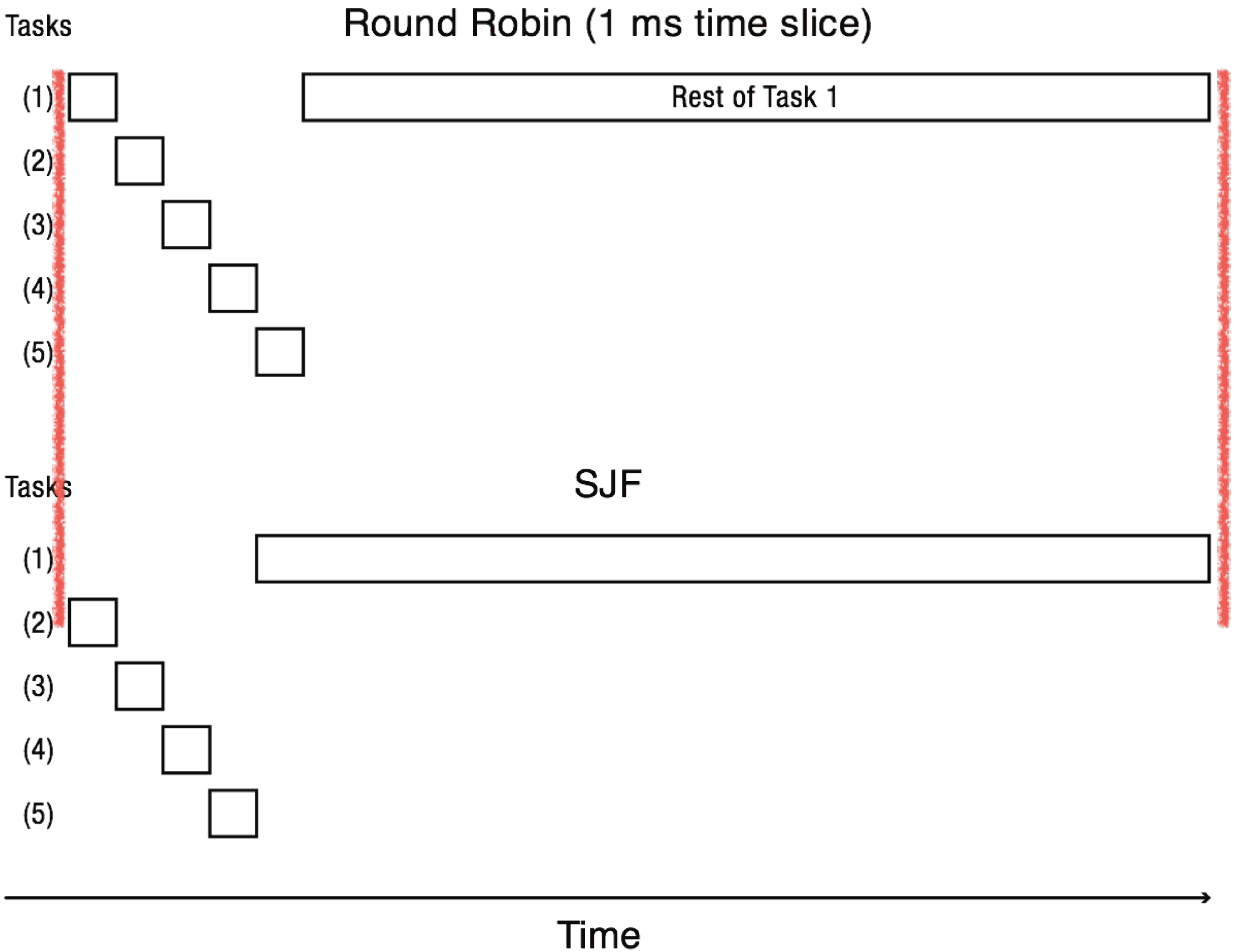


Tasks
Round Robin (100 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Time
Tasks
FIFO
(1)
(2)
(3)
(4)
(5)

Round Robin



Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Tasks
SJF
(1)
(2)
(3)
(4)
(5)
Time

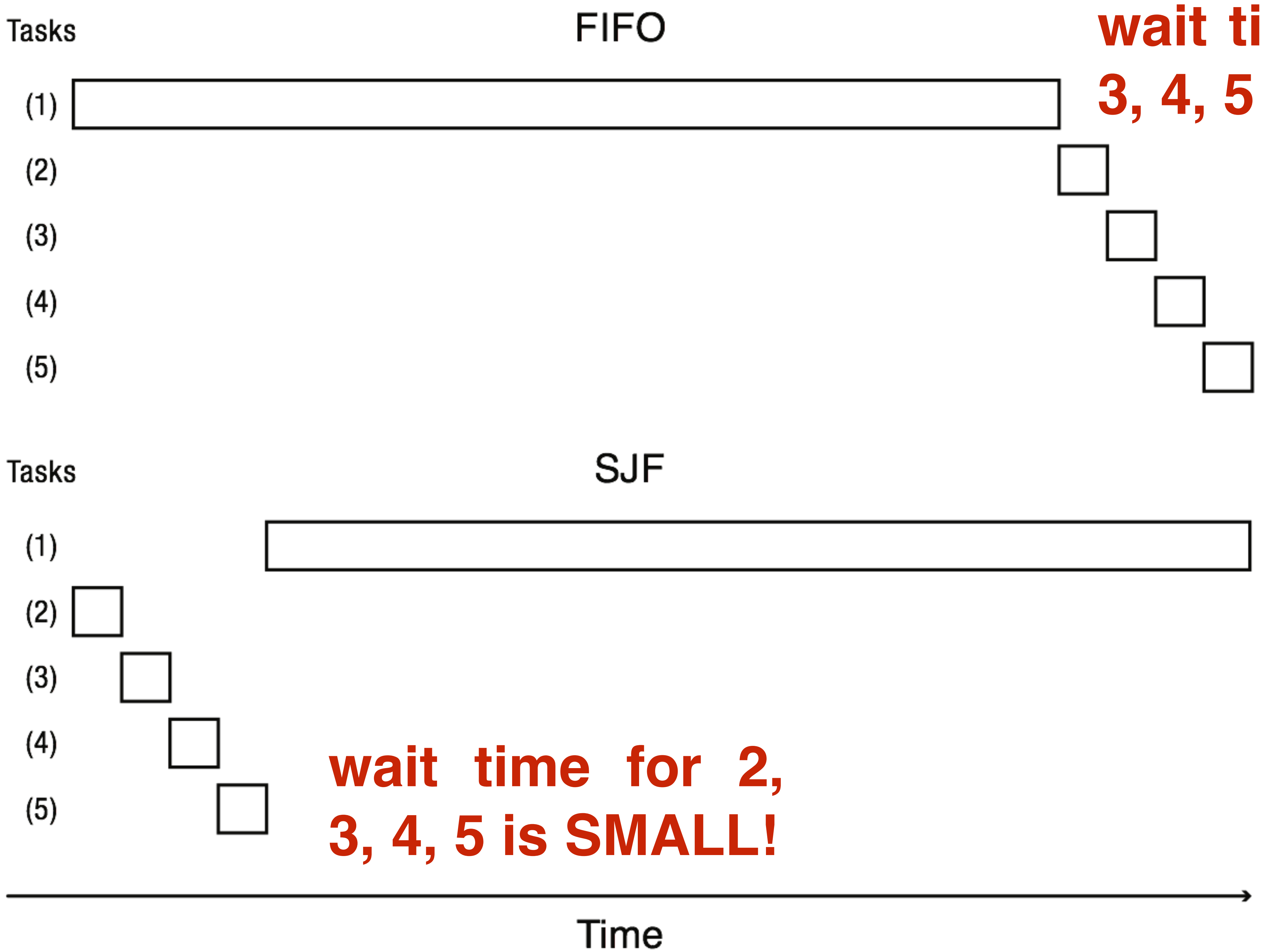
Scheduling

- Basic scheduling algorithms
  - FIFO (FCFS)
  - Shortest job first
  - Round Robin

Scheduling

- Basic scheduling algorithms
  - FIFO (FCFS)
  - Shortest job first
  - Round Robin
- What is an optimal algorithm in the sense of maximizing the number of jobs finished (i.e., minimizing average response time)?

FIFO vs. SJF



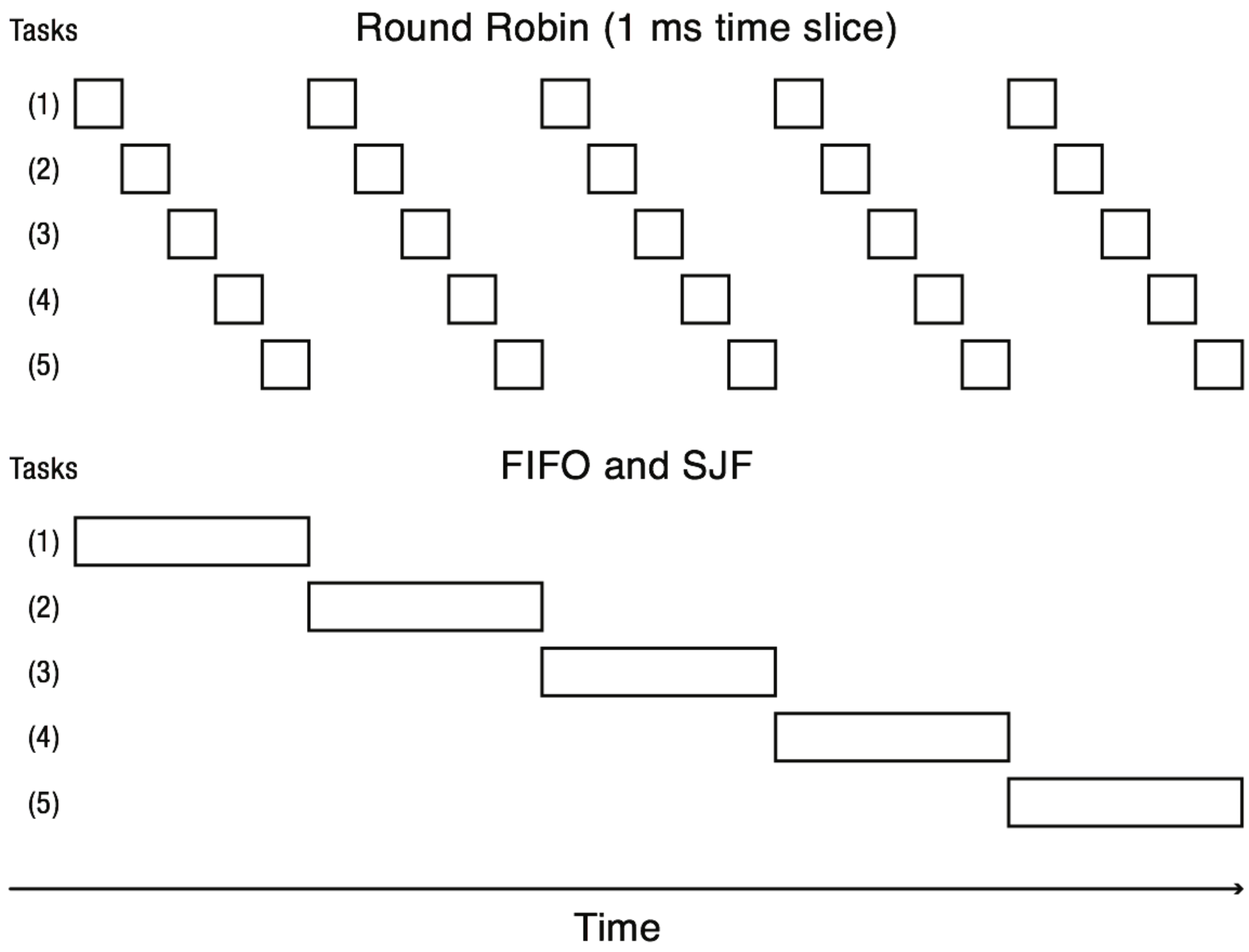
wait time for 2, 3, 4, 5 is BIG!

wait time for 2, 3, 4, 5 is SMALL!

Scheduling

- Basic scheduling algorithms
  - FIFO (FCFS)
  - Shortest job first
  - Round Robin
- Assuming zero-cost to time slicing, is Round Robin always better than FIFO?

RR v. FIFO (fixed size tasks)



Starvation, Sample Bias

- Suppose you want to compare two scheduling algorithms
  - Create some infinite sequence of arriving tasks
  - Start measuring
  - Stop at some point
  - Compute average response time as the average for completed tasks between start and stop
- Is this valid or invalid?

Sample Bias Solutions

- Measure for long enough that # of completed tasks >> # of uncompleted tasks
  - For both systems!
- Start and stop system in idle periods
  - Idle period: no work to do
  - If algorithms are work-conserving, both will complete the same tasks

Round Robin = Fairness?

Is Round Robin the fairest possible algorithm?

What is fair?

- FIFO?
- Equal share of the CPU?
- What if some tasks don’t need their full share?
- Minimize worst case divergence?
- Time task would take if no one else was running
- Time task takes under scheduling algorithm

Fairness needs to be defined.

- 4 kids share a cake.
  - Each gets 25% of the cake.
  - Quite fair!

- There is one little kids and the kid can only eat 10% of the cake.
  - We either force her to eat the 25% -- to be fair
  - Or we give 15% remaining to the other 3 kids.
    - Min-max fairness

Max-Min Fairness

- The ***least*** **demanding one** will **get its fair share *first***
- After this, the ***next least*** **demanding one** will **get its fair share *first***
- And so on...

Max-Min Fairness

- Kid 1: 20%
- Kid 2: 26%
- Kid 3: 40%
- Kid 4: 50%

- 100% -> 25% each kid
  - 20% -> 5% left -> 1.666666% to the other three
    25%
    25%
    25%

Max-Min Fairness

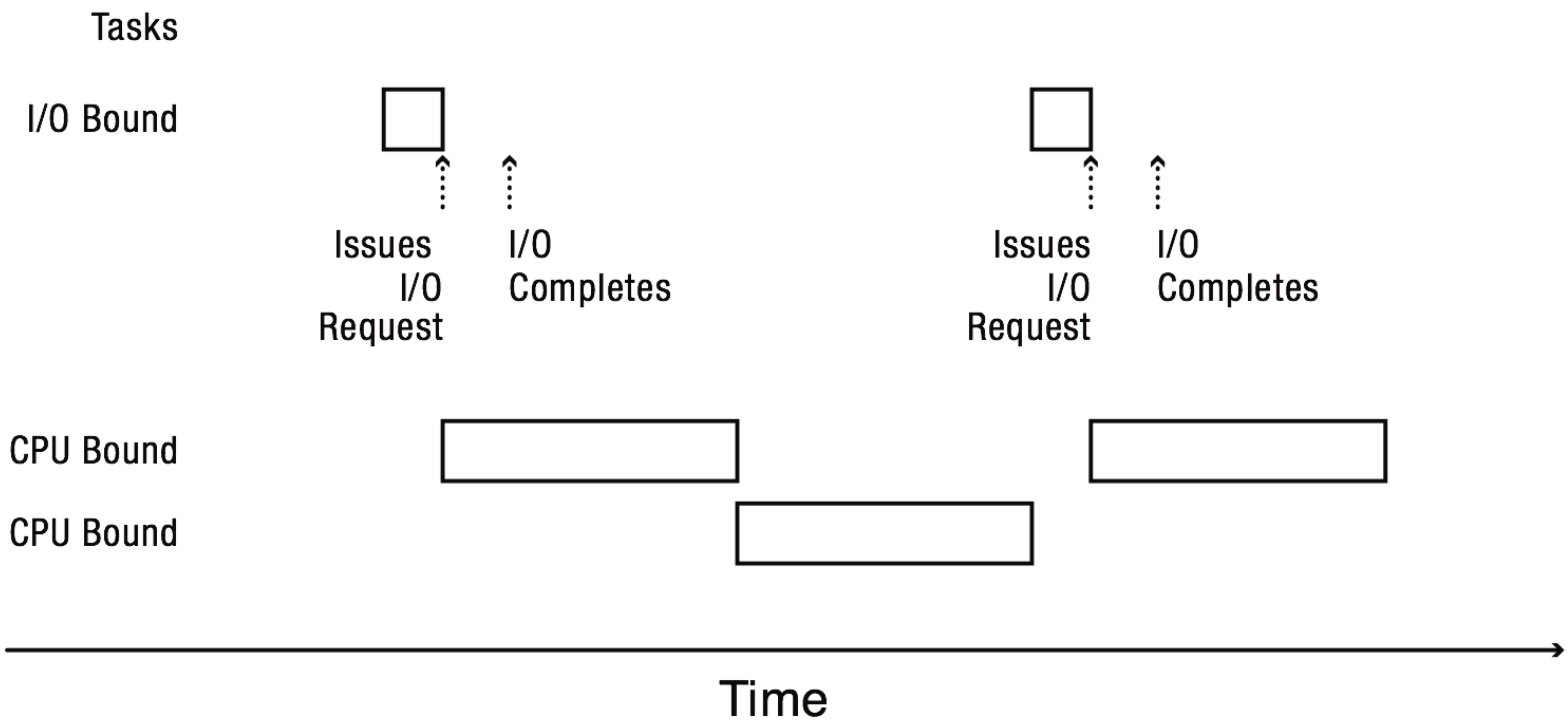
- Kid 1: 20%
- Kid 2: 26%
- Kid 3: 40%
- Kid 4: 50%

- 100% -> 25% each kid
  - ~~20%~~
    ~~26%~~
    27%
    27%

Max-Min Fairness

- How do we balance a mixture of repeating tasks?
  - Some I/O bound, need only a little CPU
  - Some compute bound, can use as much CPU as they are assigned
- One approach: maximize the minimum allocation given to a task
  - If any task needs less than an equal share, schedule the smallest of these first
  - Split the remaining time using max-min
  - If all remaining tasks need at least equal share, split evenly

Mixed Workloads??



Multi-Level Feedback Queue

- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have short time slices
  - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
- Tasks start in highest priority queue
  - If time slice expires, task drops one level

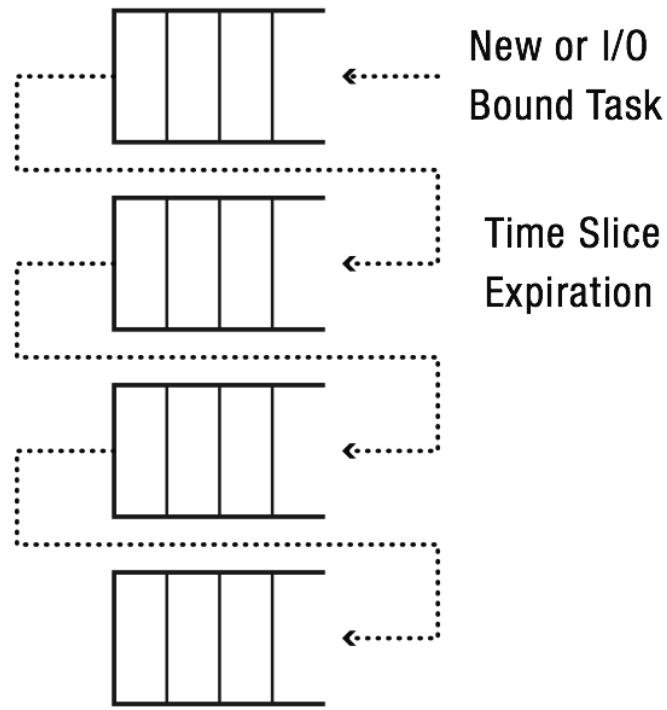
Multi-Level Feedback Queue

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)

Multi-Level Feedback Queue

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

Round Robin Queues



Why is MLFQ a good design?

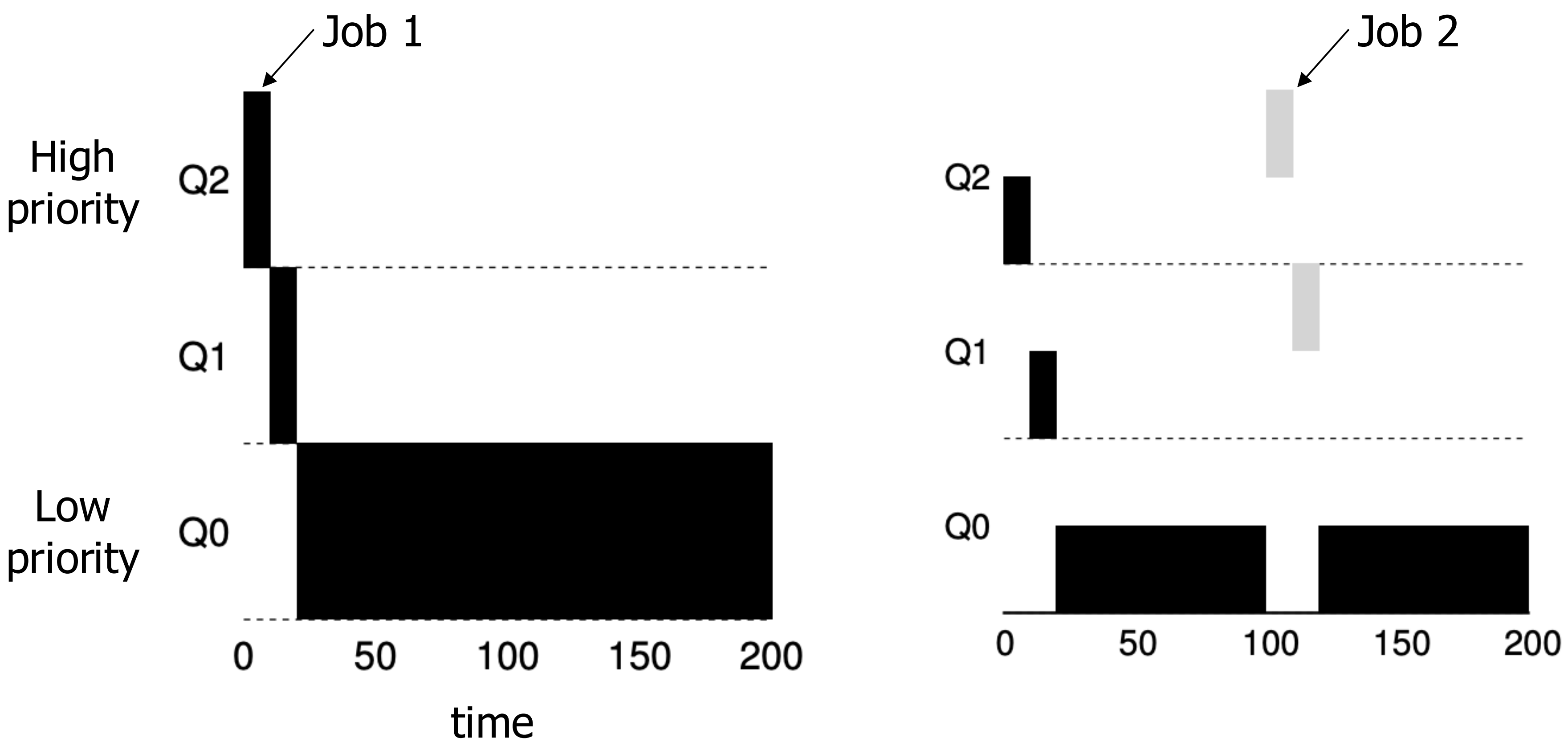
- How to design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without a priori knowledge of job length?

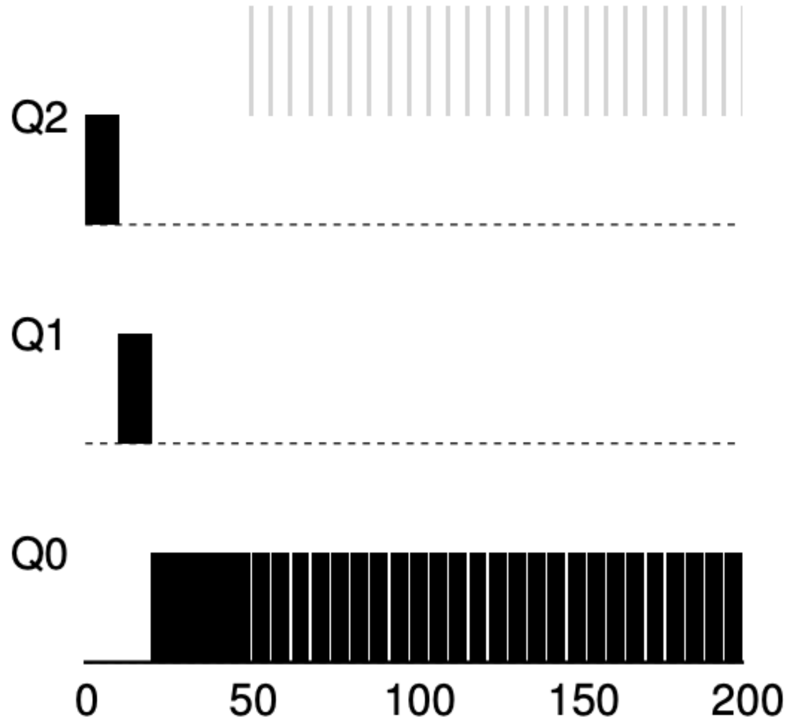
- Yes, SJF – the assumption is to know which is the "shortest.."
  - It's just very hard to know in advance.
  - Sometimes processes/threads could try to game (we will see an example).

Why is MLFQ a good design?

- The Key Idea
  - Dynamically adjusting the priority level based on observing the behavior of the processes/threads
- Basic Design
  - When a job enters the system, it is placed at the highest priority (the topmost queue).
  - If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  - If a job gives up the CPU before the time slice is up, it stays at the same priority level.

Basic Design

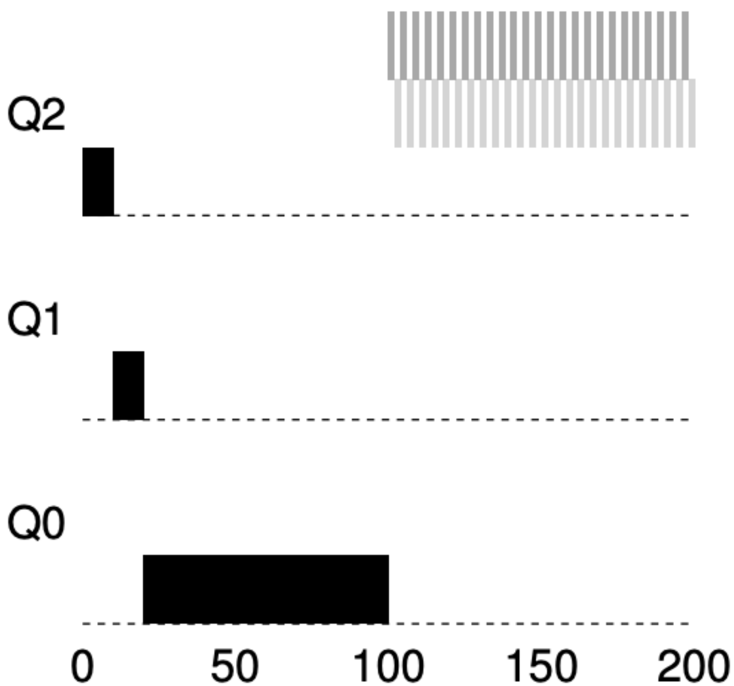


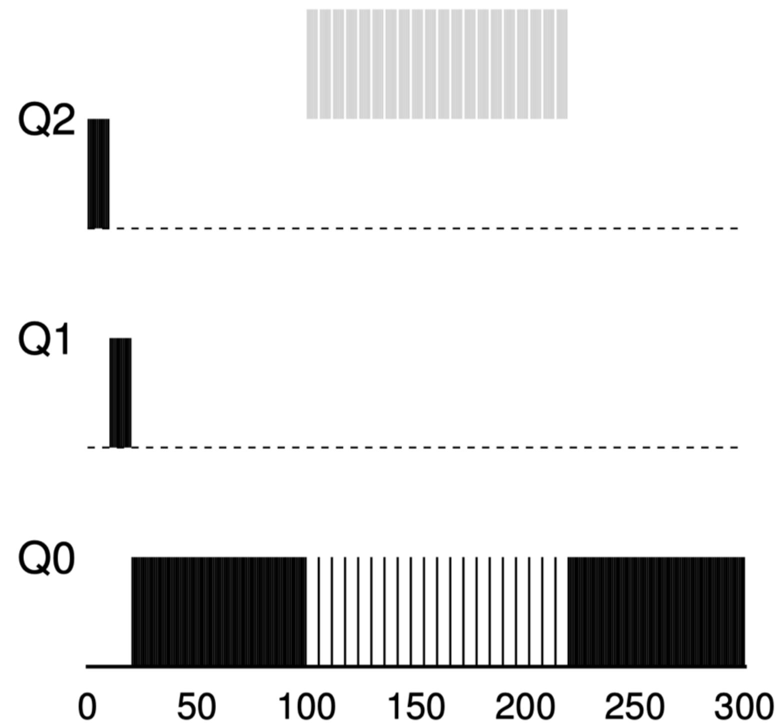


- because it doesn’t know whether a job will be a short job or a long-running job, it first assumes it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process.

Limitations?

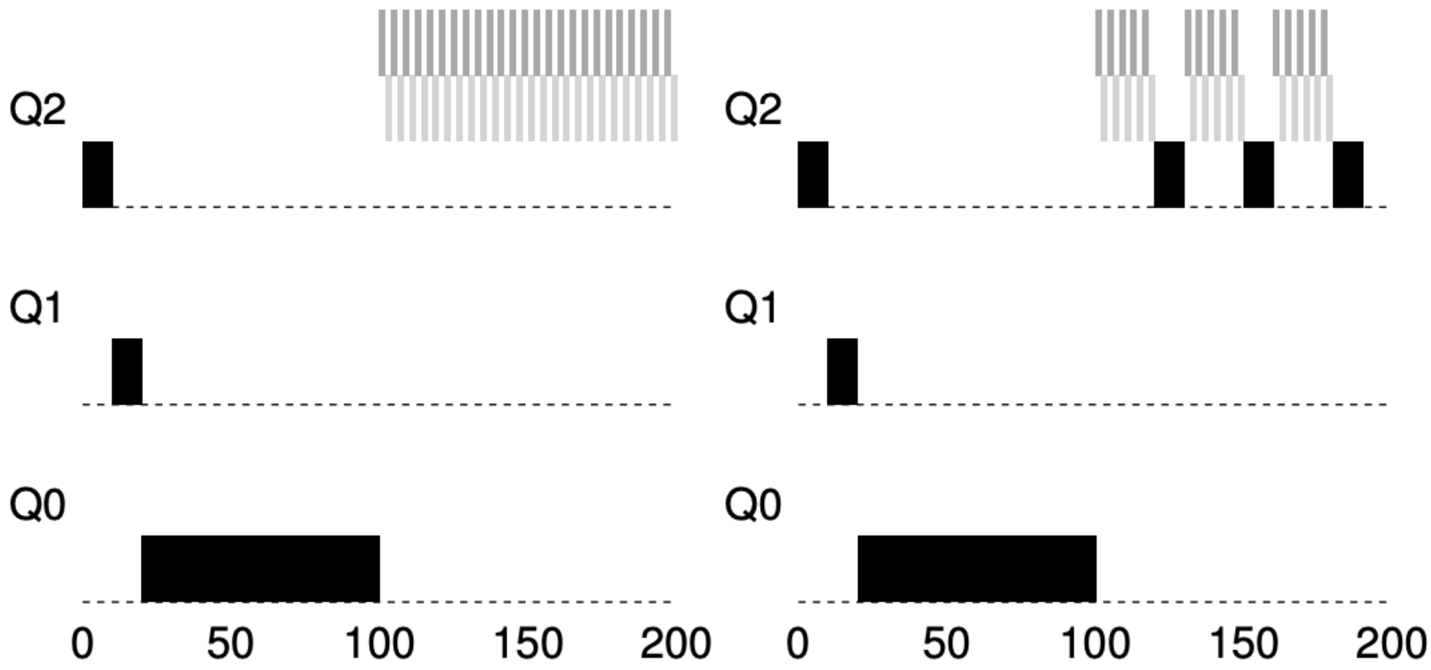
- Starvation
- A process changing its characteristics
- Gaming the scheduler





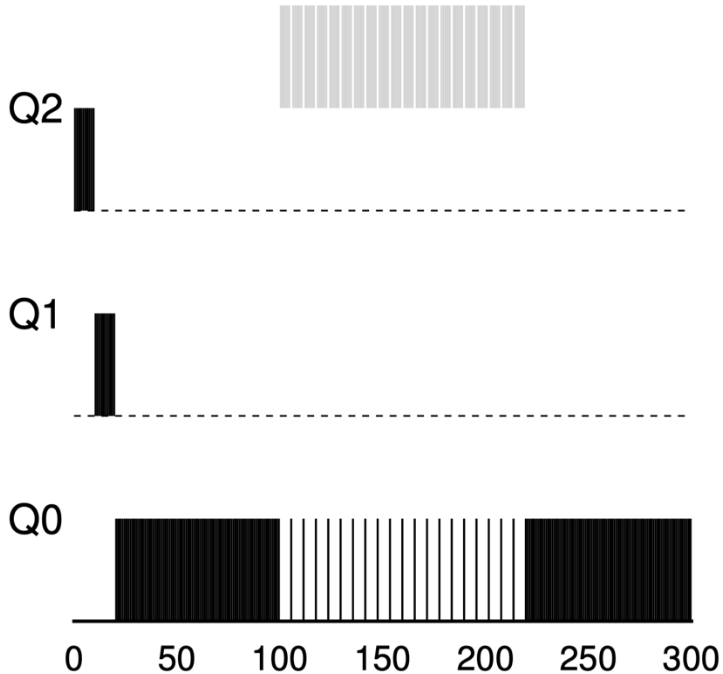
Priority Boost

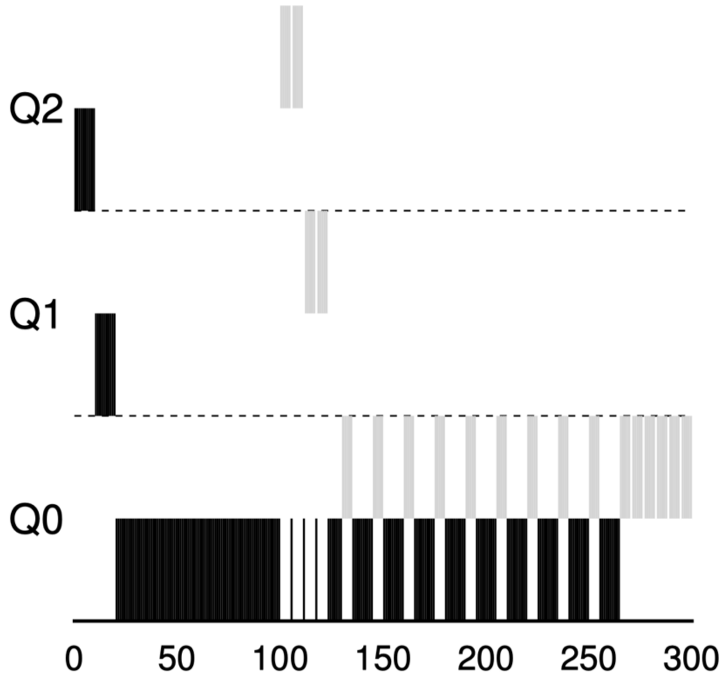
- After some time period S, move all the jobs in the system to the topmost queue



Better Accounting

- 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 one queue).





Sounds perfect?

- How many queues should there be?
- How big should the time slice be per queue?
- How often should priority be boosted in order to avoid starvation and account for changes in behavior?

Summary

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.

Summary

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.

Summary

- Max-Min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-Min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.
- Is MFQ optimally fair??