

CSE 451: Operating Systems, 26au

L11: Deadlock Mitigation, CPU Scheduling

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head) Mahir Emran Ted Guan

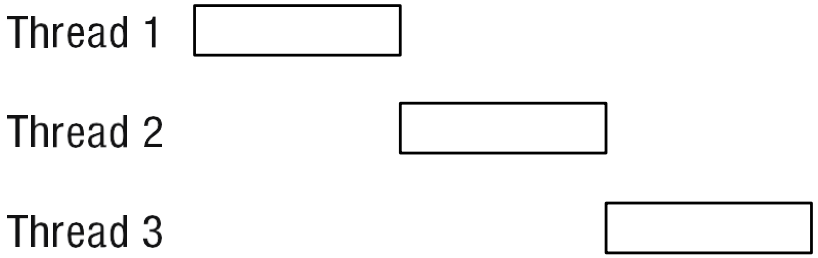
Jai Rungrachakanont Soham Raut Sagar Saby

Andrew Wood

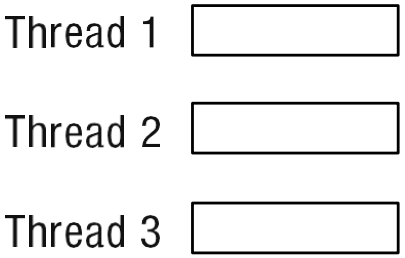
Adapted from slides by Tom Anderson

Concurrency

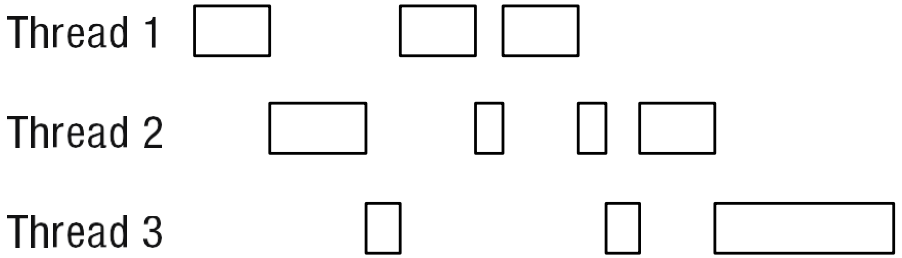
One Execution



Another Execution



Another Execution





Deadlocks, realistically

- **Low priority thread A** acquires lock
- Preempted by **high priority thread B**
- **Thread B** tries to acquire lock; sleep
- Switch back to low priority **thread A**
- Preempted by **medium priority thread C**
- **Thread C** prevents high priority **thread B** from running

Deadlocks, formalized

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa

The 🎶 Necessary Conditions for Deadlock 🎶

1. An exhaustible resource
   - Eg: a limited number of threads can use the resource
2. No preemption
   - Eg: nobody else can revoke a thread's access to the resource
3. Multiple independent requests
   - Eg: A thread holds one resource while waiting for another
4. Circular chain of requests
   - Eg: There is a set of waiting threads such that each is waiting for something held by another

A messier example still

Thread 1

1. Acquire A

3. Acquire C

5. If (maybe) Wait for B

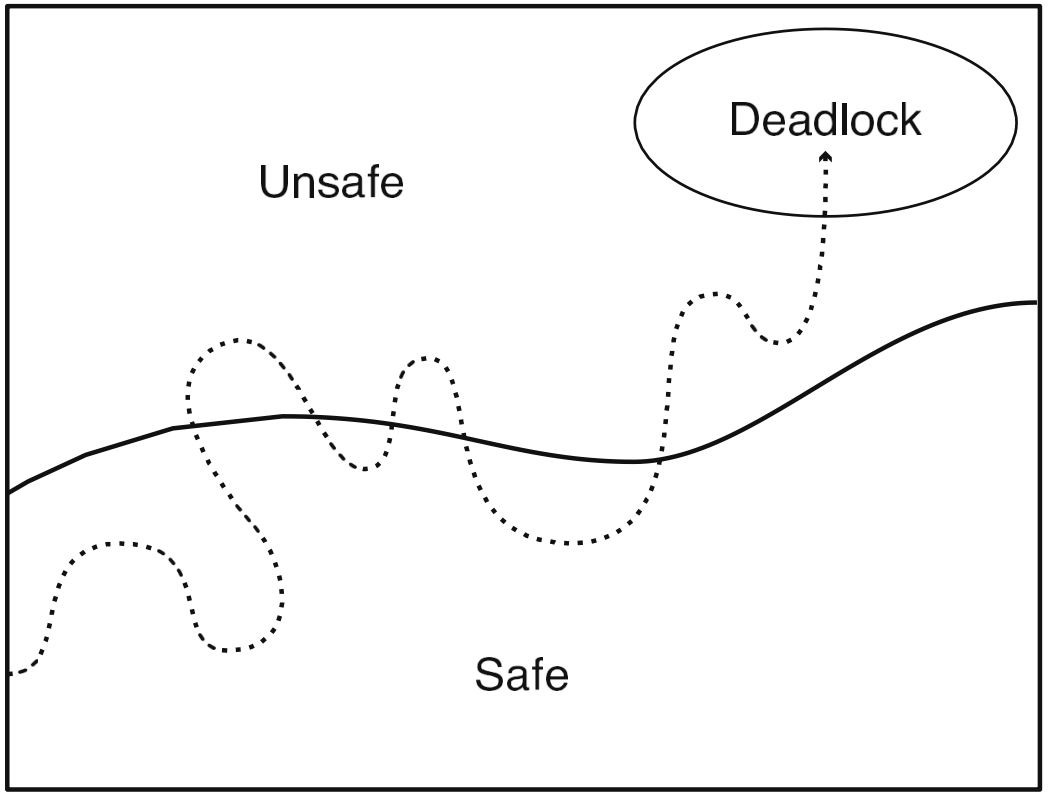
Thread 2

2. Acquire B

4. Wait for A

Deadlock Dynamics

- Safe state:
  - For any possible sequence of future resource requests, it is possible to eventually grant all requests
  - May require waiting even when resources are available!
- Unsafe state:
  - Some sequence of resource requests can result in deadlock
- Doomed state:
  - All possible computations lead to deadlock

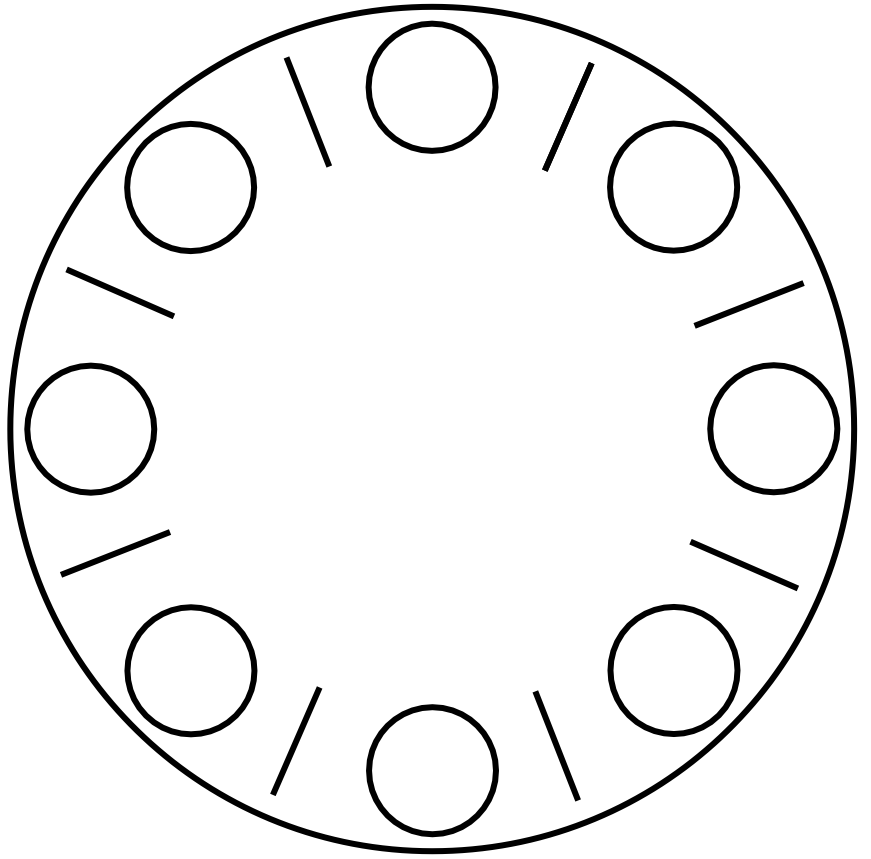


The Dining Lawyers

N lawyers. Each lawyer needs two chopsticks to eat.

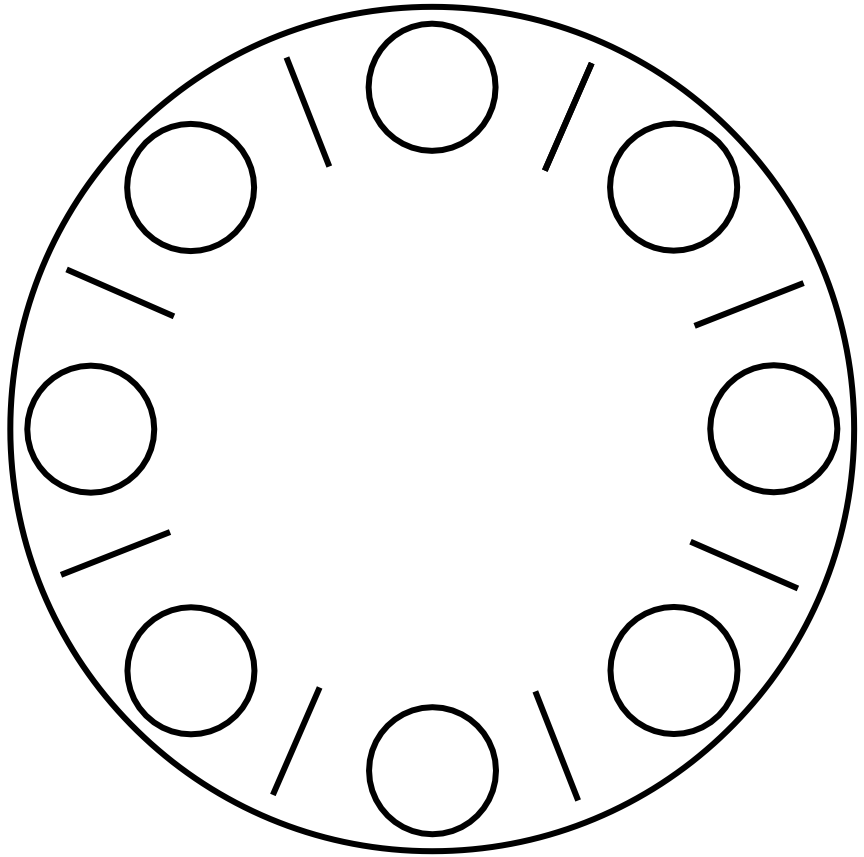
Each lawyer grabs chopstick on their right, then the one on their left.

```
lawyer(idx) {
    chopsticks[(idx + 1) % N].acquire()
    chopsticks[idx].acquire()
    eat();
    chopsticks[idx].release()
    chopsticks[(idx + 1) % N].release()
}
```



Paralysis Analysis

- Which 🎶 necessary conditions for deadlock 🎶 do the Dining Lawyers meet?
  - Limited access to resources
  - No preemption
  - Multiple independent requests (wait while holding)
  - Circular chain of requests

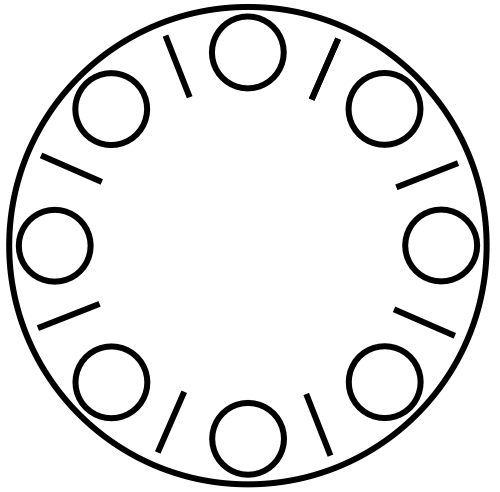


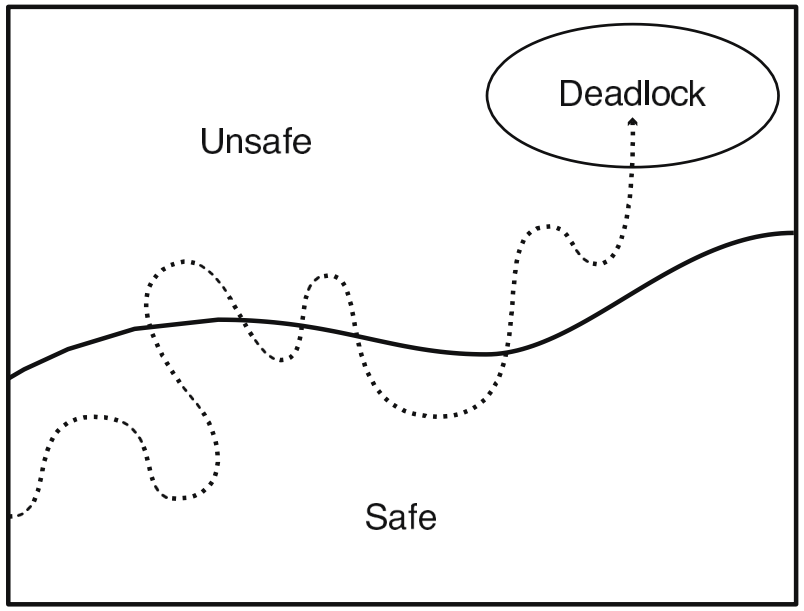
Paralysis Analysis (2)

- What are the doomed states for Dining Lawyers?

- What are the unsafe states?

- What are the safe states?





Deadlock Mitigation

Preventing Deadlock

- Exploit or limit program behavior
  - Limit program from doing anything that might lead to deadlock
- Predict the future
  - If we know what program will do, we can tell if granting a resource might lead to deadlock
- Detect and recover
  - If we can rollback a thread, we can fix a deadlock once it occurs

Exploit or Limit Behavior: resources

- Provide enough resources
  - Not always possible to increase resource count (eg, mutexes)
  - Not always obvious which resources are causing the problem

Exploit or Limit Behavior: waiting

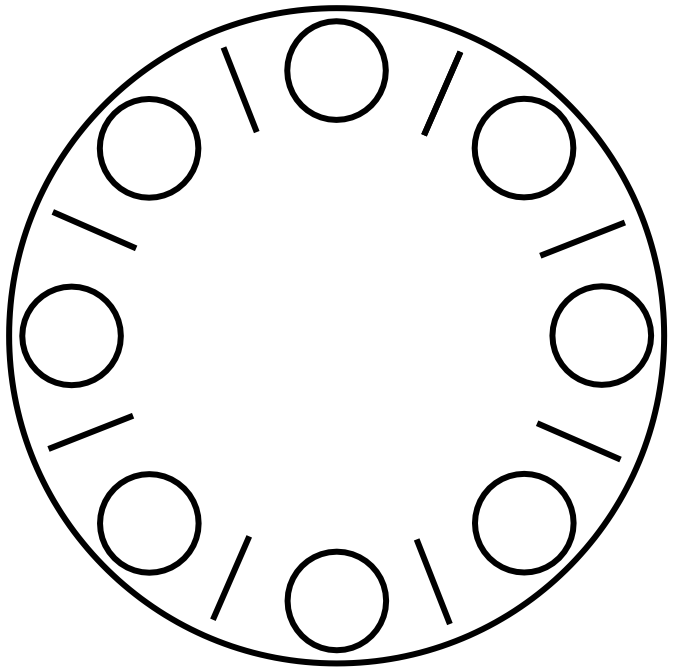
- Eliminate wait while holding
  - Release lock when calling out of module
  - Telephone circuit setup gives busy signal

Exploit or Limit Behavior: cycles

- Eliminate circular waiting
  - Lock ordering: **always acquire locks in a fixed order**
  - Example: move file from one directory to another

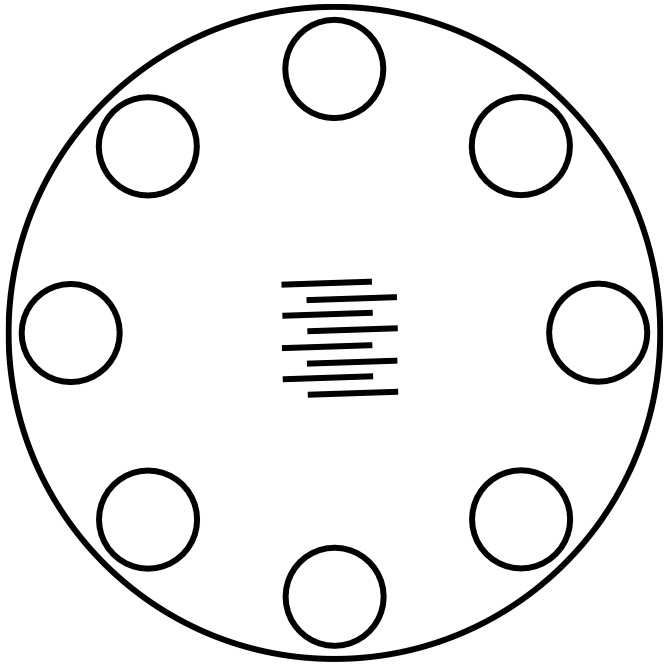
Lawyers again

- Can we use increase resource limits to eliminate deadlock?
- Can we use resource ordering to eliminate deadlock?
- How else can we be clever?



Communal Dining Lawyers

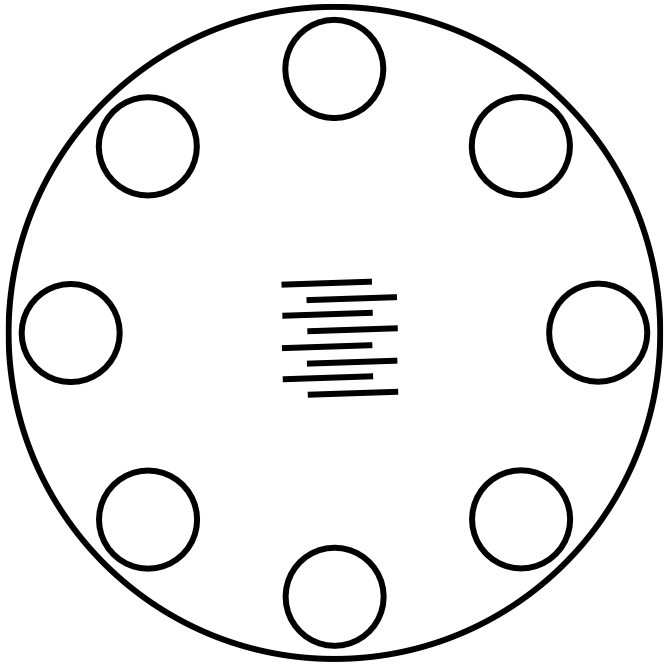
- **N** lawyers
- **N** chopsticks in **middle** of table
- each needs two chopsticks but can only take one at a time



- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Communal Mutant Dining Lawyers

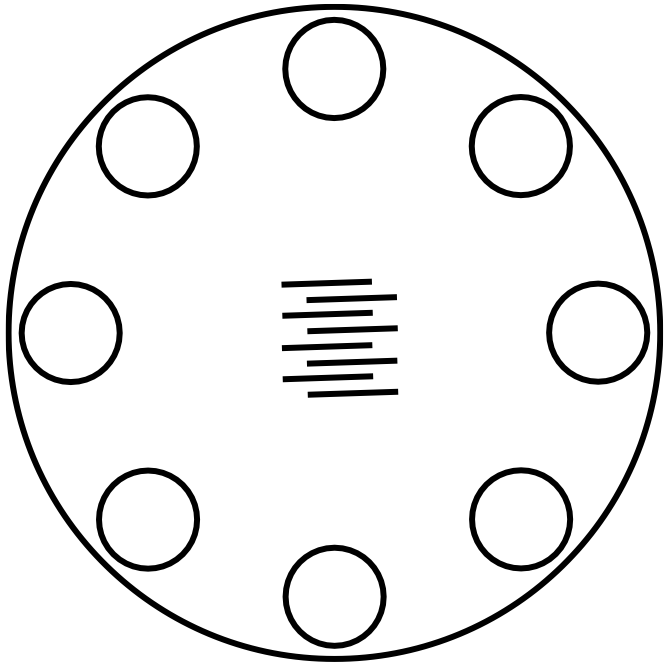
- **N** lawyers
- **N** chopsticks in **middle** of table
- each needs **k** chopsticks but can only take one at a time, for **k** > 1



- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Communal Mutant Dining Lawyers **With Butterfingers**

- **N** lawyers
- **N** chopsticks in **middle** of table
- each needs **k** chopsticks but can only take one at a time, for **k** > 1
  - Sometimes a lawyer has a slippery grip and drops a chopstick, so they need **k+1** chopsticks
- What are the safe states?
- What are the unsafe states?
- What are the doomed states?



Banker's algorithm

- "You need to spend money to make money"
  - Banks offer small business loans under the assumption the business will use it to make enough money to pay it back (and then some)

- In an OS:
  - Provide a "gate" function that asks a process to specify its worst-case resource needs before it starts to acquire them
  - Based on global knowledge of all processes, determine if allowing the process to go ahead could *possibly* lead to deadlock
  - If it could, make process wait

“Could cause deadlock” ?

- Consider how much total resource there is (T), how much is presently allocated elsewhere (A), and the maximum amount a process will need (M)
- If T – A >= M, we’re safe :)
- If T – A <= M …….well…..maybe
  - If we can find *some* order of giving each process the resources they need, we’re safe from deadlock
  - Finding this order can be difficult

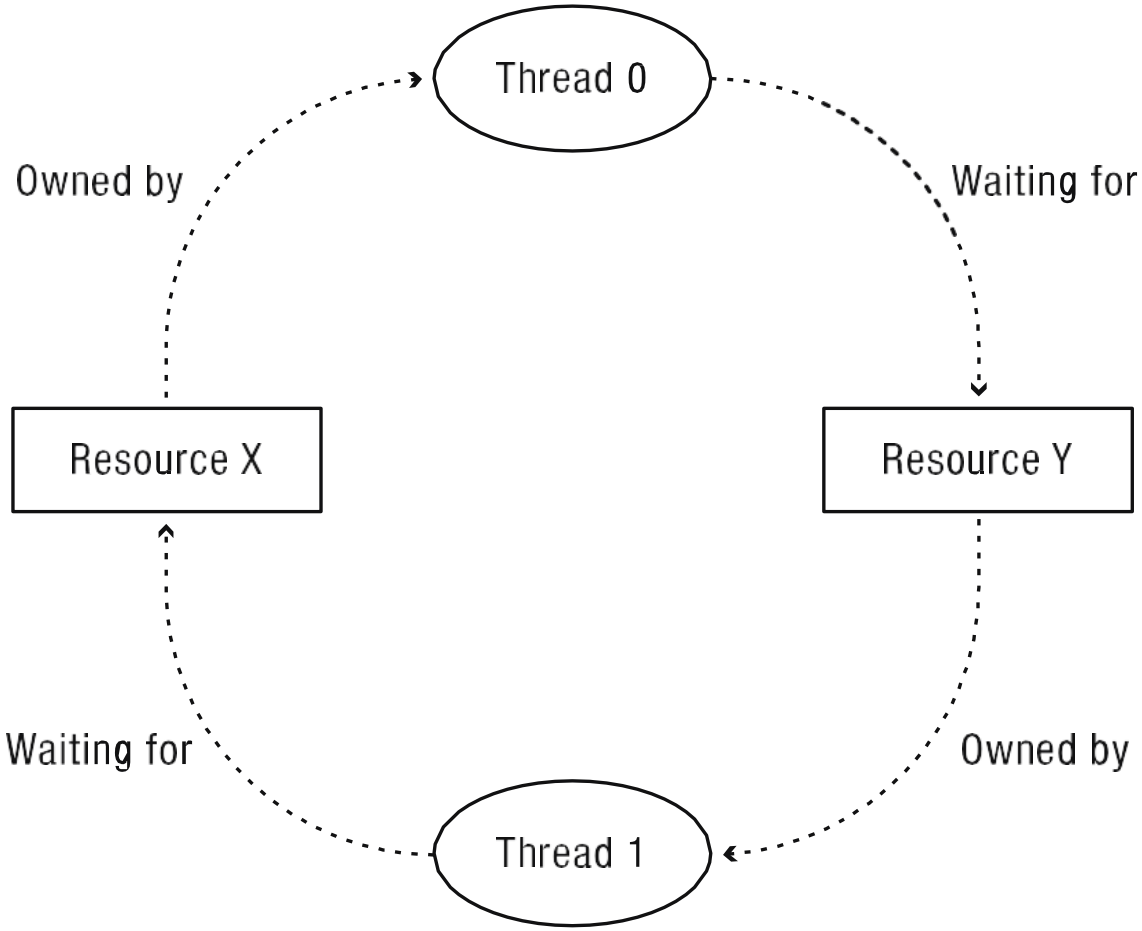
Banker’s algorithm, practically

- Practically useful when our resources are quantifiable (like chopsticks) and non-unitary (unlike a lock/mutex)
  - Computer networking algorithms
  - Heap allocators

Class concluded here on 10/23 and will resume on 10/26



Deadlocks, illustrated



Alternative to playing it safe: **“Detect and Repair”**

- Algorithm
  - Examine the “waiting-for” graph
  - Detect cycles
  - Fix cycles…somehow
- Strategy 1: Proceed without the resource
  - Requires robust exception handling code
- Strategy 2: Roll back and retry
  - Transaction: all operations are provisional until have all required resources to complete operation

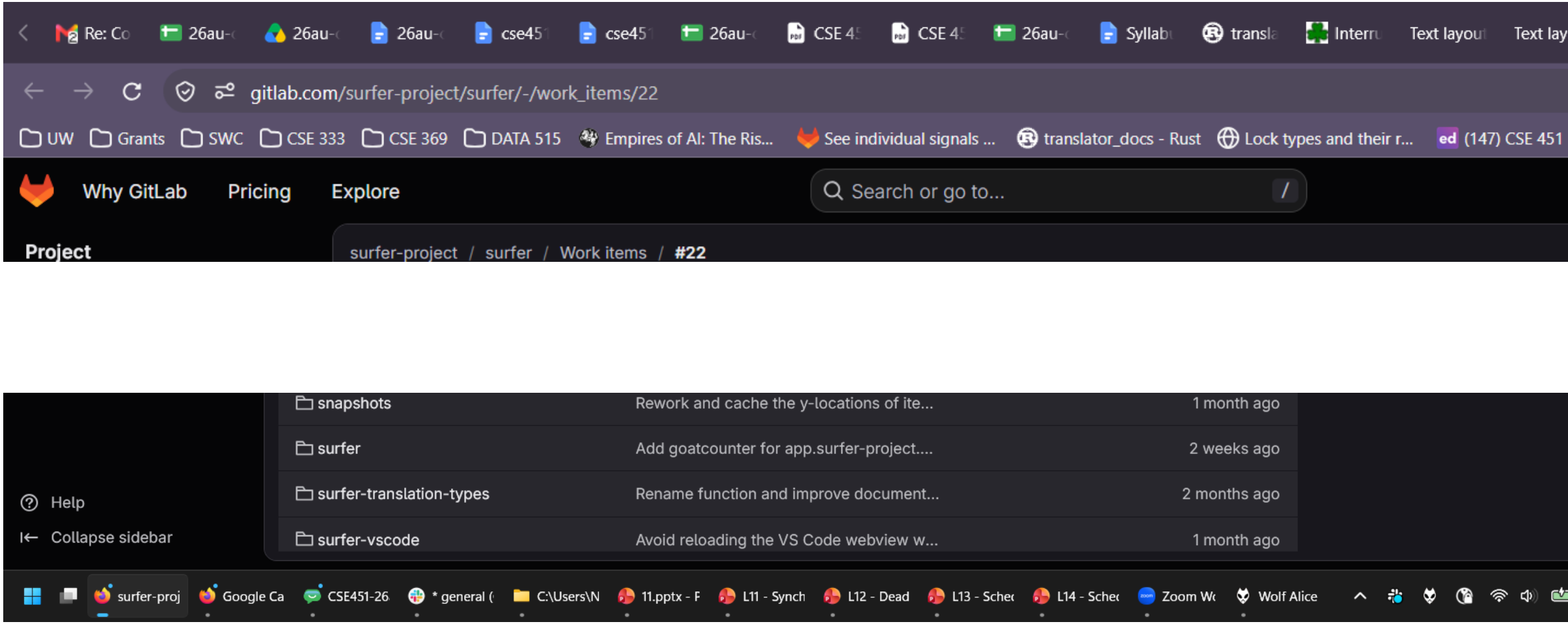
CPU Scheduling

Scheduling

- So far, we've looked at the OS scheduler primarily to understand the "**nitty-gritty**" way that it performs **context switching**. But there is *much more to it* than that.

- The most important part of the scheduler: **the policy.**
  That is, **how do we choose which process/thread runs next?**

- The answer depends on:
  - Workload
  - Uni-processor vs multi-processor


gitlab.com/surfer-project/surfer/-/work_items/22
Why GitLab
Pricing
Explore
Search or go to...
Project
surfer-project / surfer / Work items / #22
snapshots
Rework and cache the y-locations of ite...
1 month ago
surfer
Add goatcounter for app.surfer-project....
2 weeks ago
surfer-translation-types
Rename function and improve document...
2 months ago
surfer-vscode
Avoid reloading the VS Code webview w...
1 month ago
Help
Collapse sidebar

AUGUST 11, 2026

Cloudflare DDoS Threat Report H1 2026: 1 Tbps attacks soar as DNS floods and geopolitical tensions drive a new wave

Cloudforce One

8 MINUTE READ COPY URL

THIS POST IS ALSO AVAILABLE IN DEUTSCH, ESPAÑOL, FRANÇAIS, 日本語, 한국어, 繁體中文, 简体中文, AND NEDERLANDS.



Schedulers are math, schedulers are science

- We'll talk about our OS's processes in terms of **"workloads"** and **"tasks"**
  - Tasks are things like mouse clicks, web requests, shell commands, etc
  - Workloads are sets of tasks (like, a tab in a web browser)

- Our scheduler is defined by its **scheduling algorithm**
  - takes a workload as input
  - decides which tasks to do first
  - performance metric (throughput, latency) as output

- In this class, we'll only consider **pre-empting, work-conserving schedulers**
  - **Pre-empting:** we can take resources *away* from a running task
  - **Work-conserving:** a resource is used *whenever* there is a task that can make use of it (keep the resource busy)
    - (note: for *non-preemptive* schedulers, work-conserving is *not always better)*

Metrics

- We care about optimizing:
  - **Latency/response time**: how long does a task to complete?
    - **Tail latency**: what is the *worst case* response time?
  - **Throughput:** how many tasks can we do per unit of time?
  - **Overhead:** How much extra work/time is spent in the scheduler itself?
  - **Fairness:** Do all tasks get an equal share of the CPU? Should they?
  - **Predictability**: How consistent is the system's performance over time?
  - **Robustness to gaming:** like "gaming the system", not "playing Zelda"

Uni-processor Scheduling Policies

First In First Out (FIFO)

- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor

- Example: memcached
  - Facebook cache of friend lists, …

- What are some strengths of FIFO?
- What are some weaknesses of FIFO?

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

Shortest Job First (SJF)

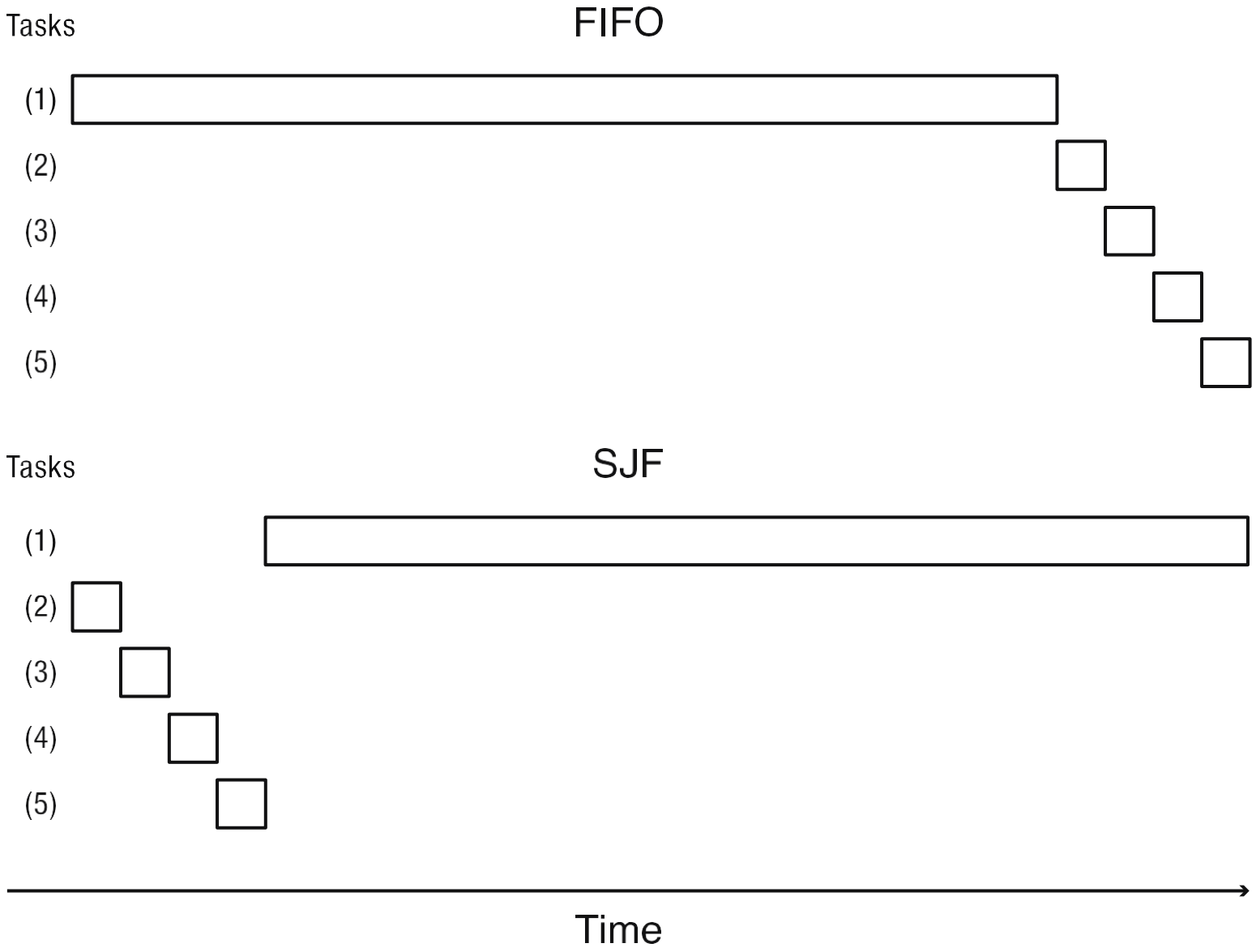
- Always run the task with the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

- Claim: SJF is optimal for average response time. Is that true?
- Does SJF have any downsides?

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



FIFO vs. SJF



Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

Fundamental Tradeoff

Average Response Time (Snappiness)



Response Time Variance (Predictability)

Tail Latency

- Designing for “the worst case” can sometimes be *really* important
  - Eg: mapreduce needs to wait for the slowest task
  - Eg: Many cloud systems provide baseline **SLA**s (service level agreements) for their customers’ applications
    - Tail behavior: 99% (or 99.9%) latency, downtime, etc…
  - Starvation of some jobs not an option

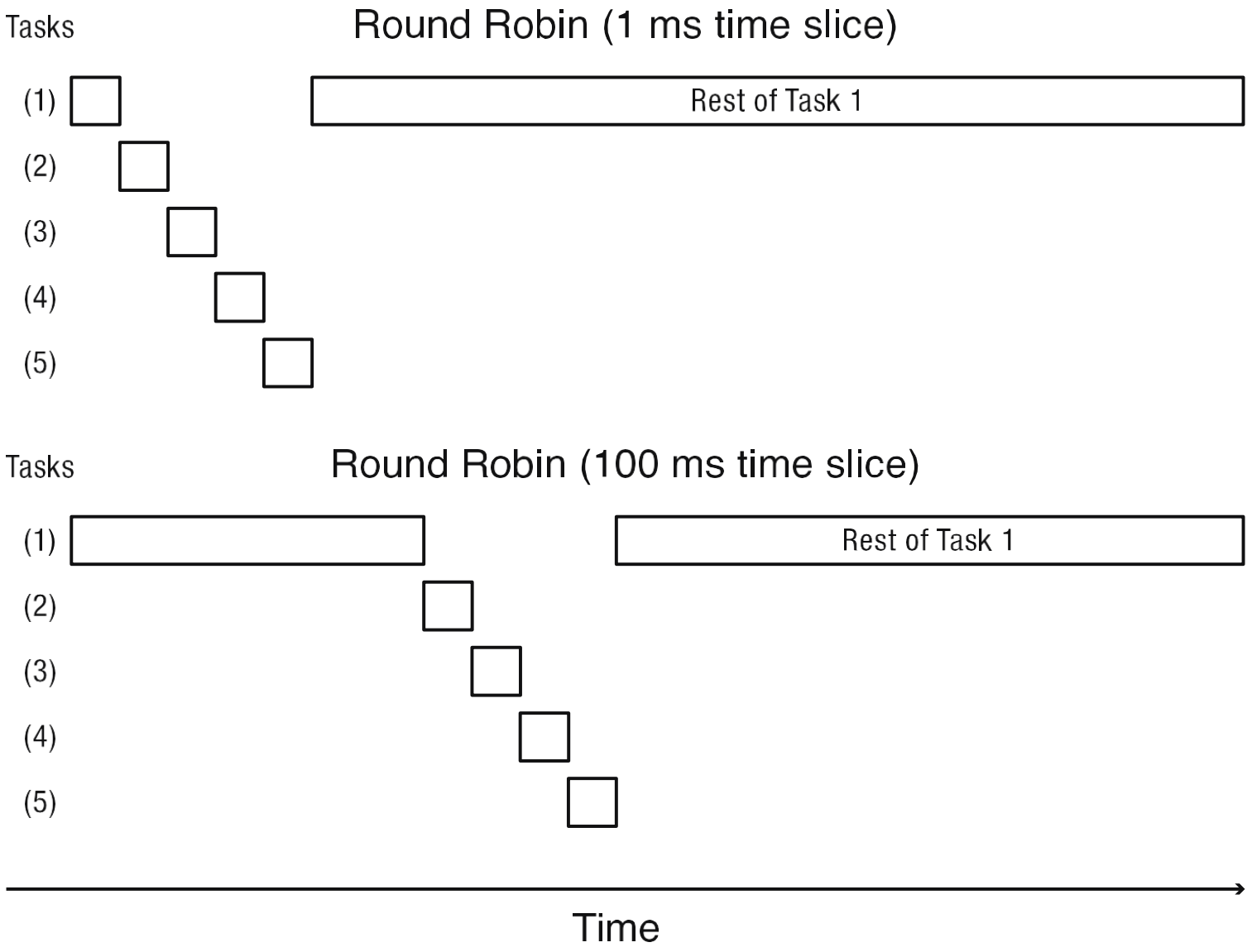
Round Robin

- Each task gets resource for a fixed period of time (“**time quantum**”)
  - If task doesn’t complete, it goes back in line
- Need to pick a time quantum
  - Infinite (max) -> FIFO scheduling
  - One instruction (min) -> Overhead, “thrashing”

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

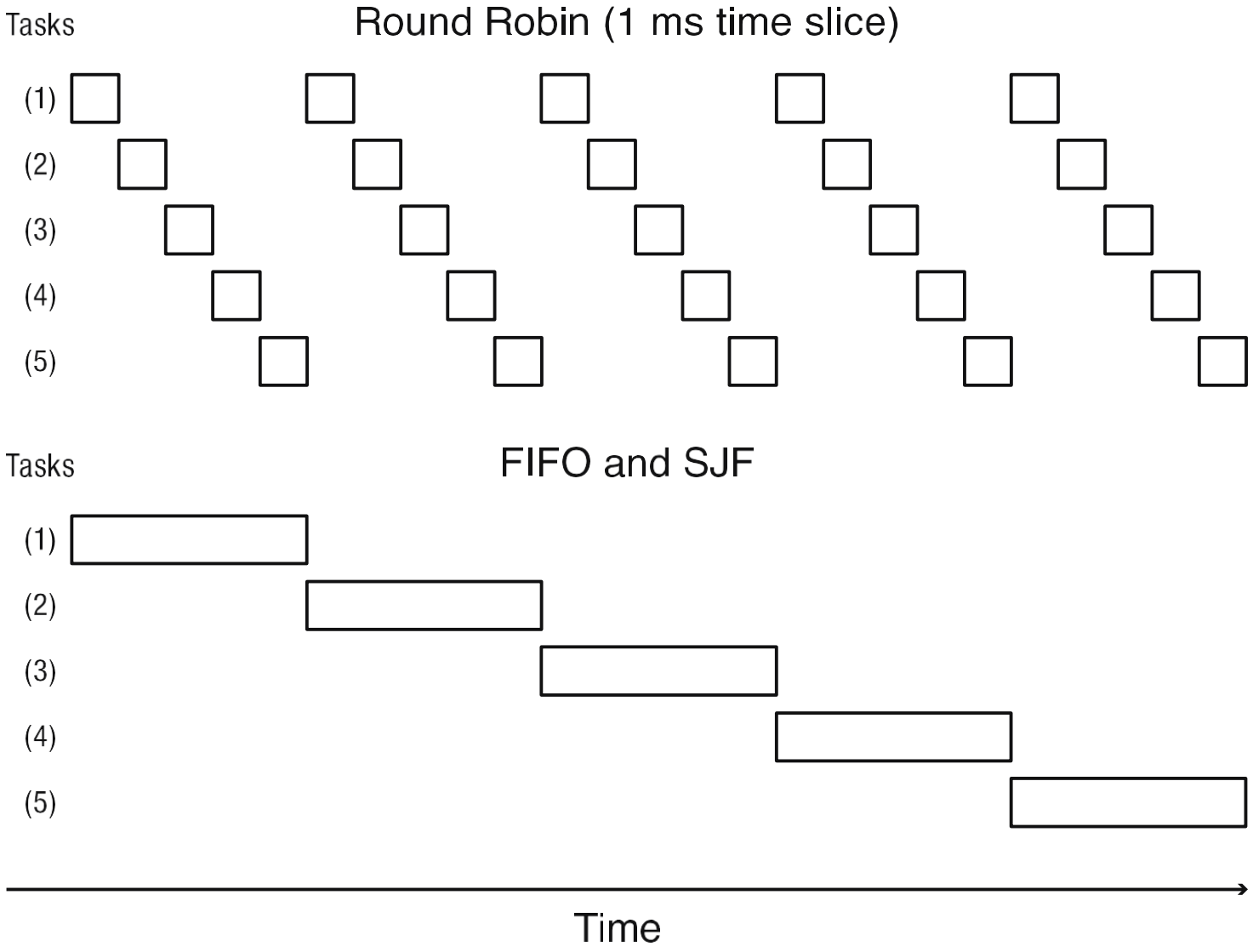


Round Robin



Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

Round Robin vs. FIFO



Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

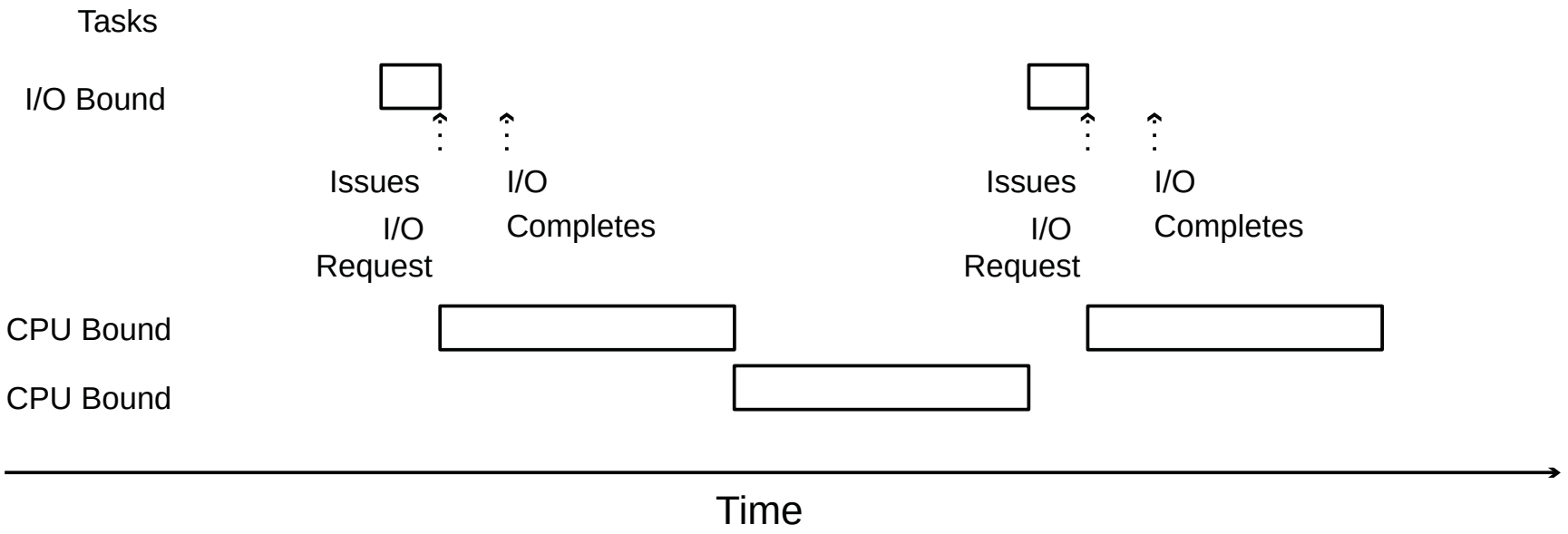
Round Robin == Fairness?

- What *is* fair?

- Equal share of the CPU?
  - What if some other tasks don't need their full share? How do we allocate the remainder?

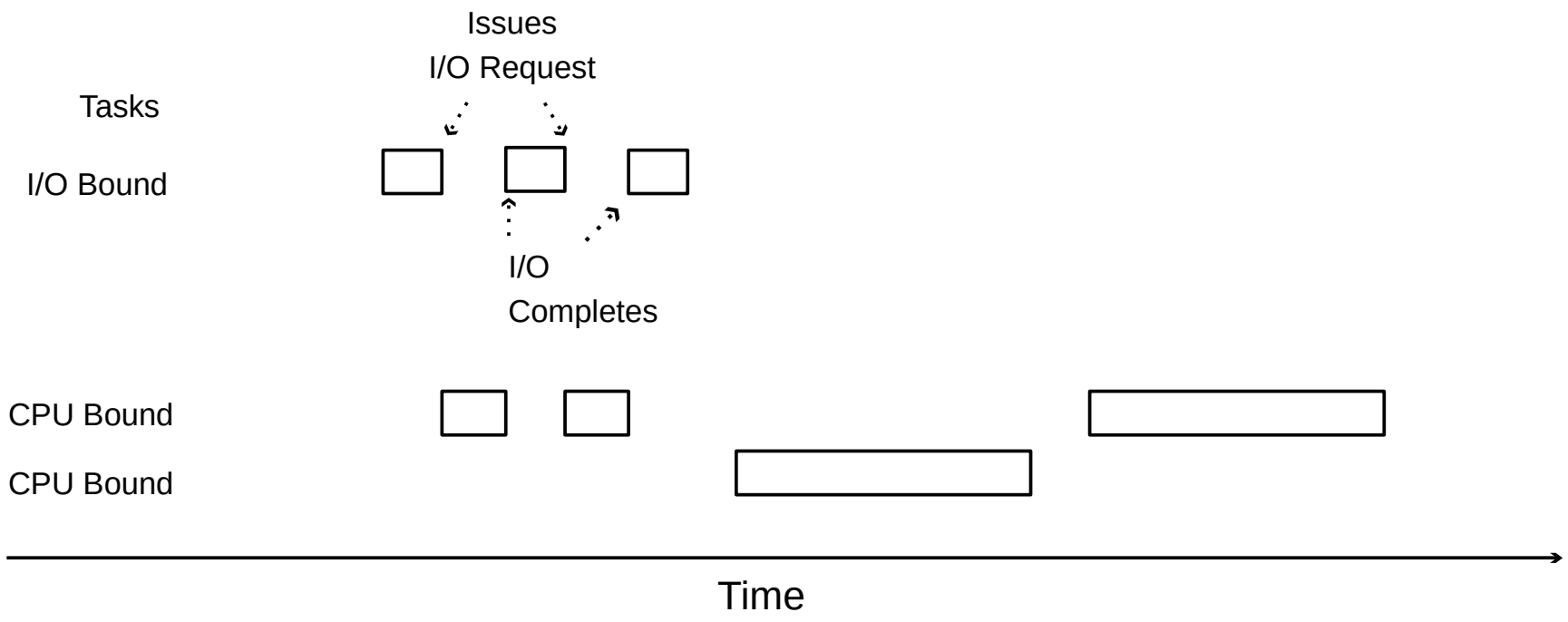
Mixing I/O-bound and Compute-bound Tasks (Round Robin)

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



Large slowdown for IO-bound tasks

Example: Prioritizing I/O



Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

We will revisit this example when discussing fair queuing

Multi-level Feedback Queue (MFQ)

- The Queen of CPU Scheduling 
  - Considerate of *all* our metrics, at the cost of not being perfect at any of them

- Idea: keep a set of round-robin queues with different priorities, and intelligently move processes between them
  - High priority queues have short time slices
  - Low priority queues have long time slices

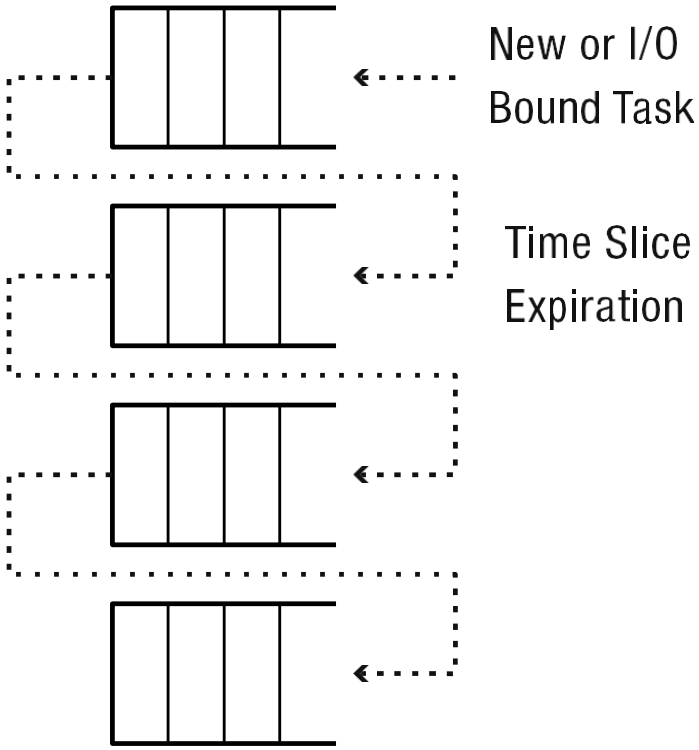
- Rules:
  - Task starts in highest priority queue
  - If time slice expires, task drops one level
  - If low priority tasks are being starved, bump them up a level

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

MFQ diagram

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

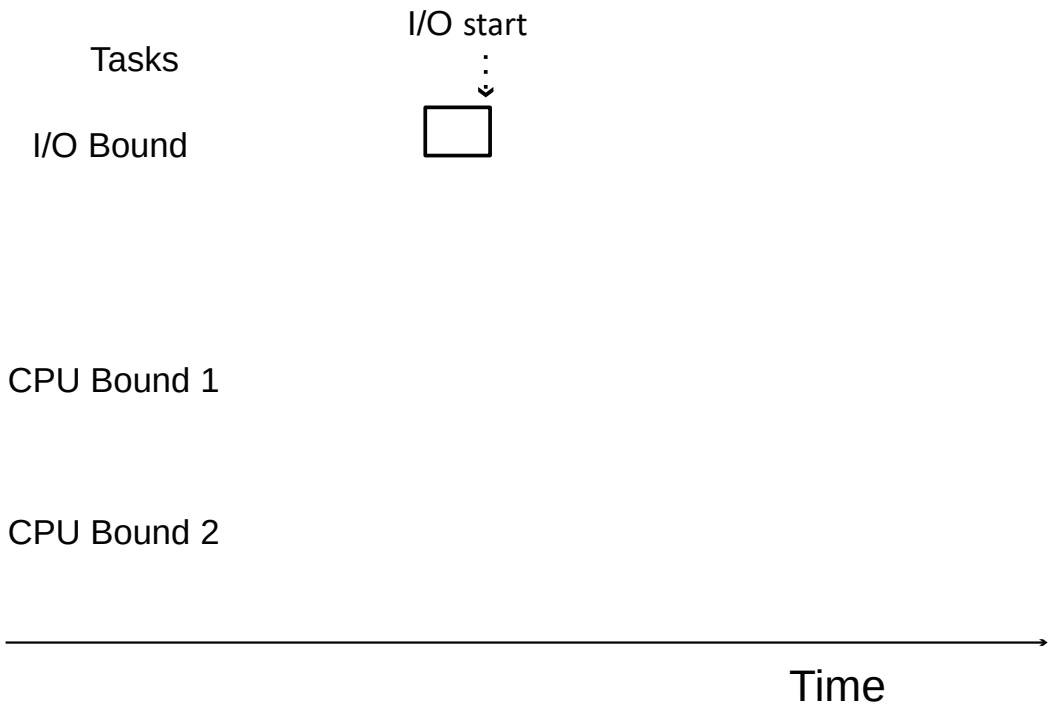
Round Robin Queues



Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

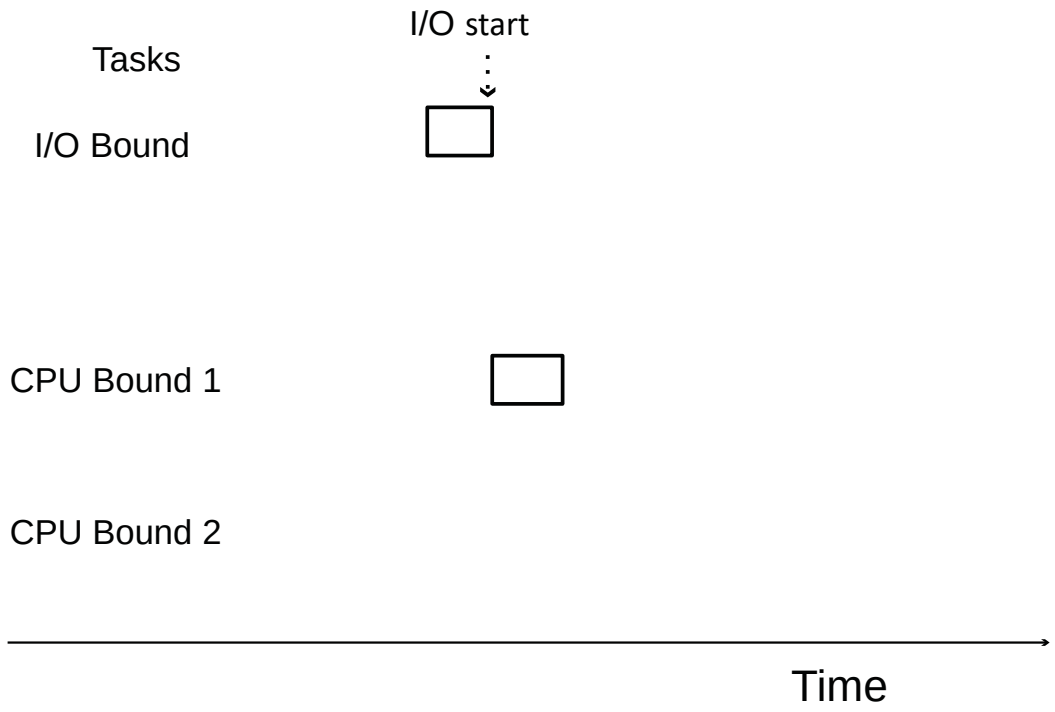
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



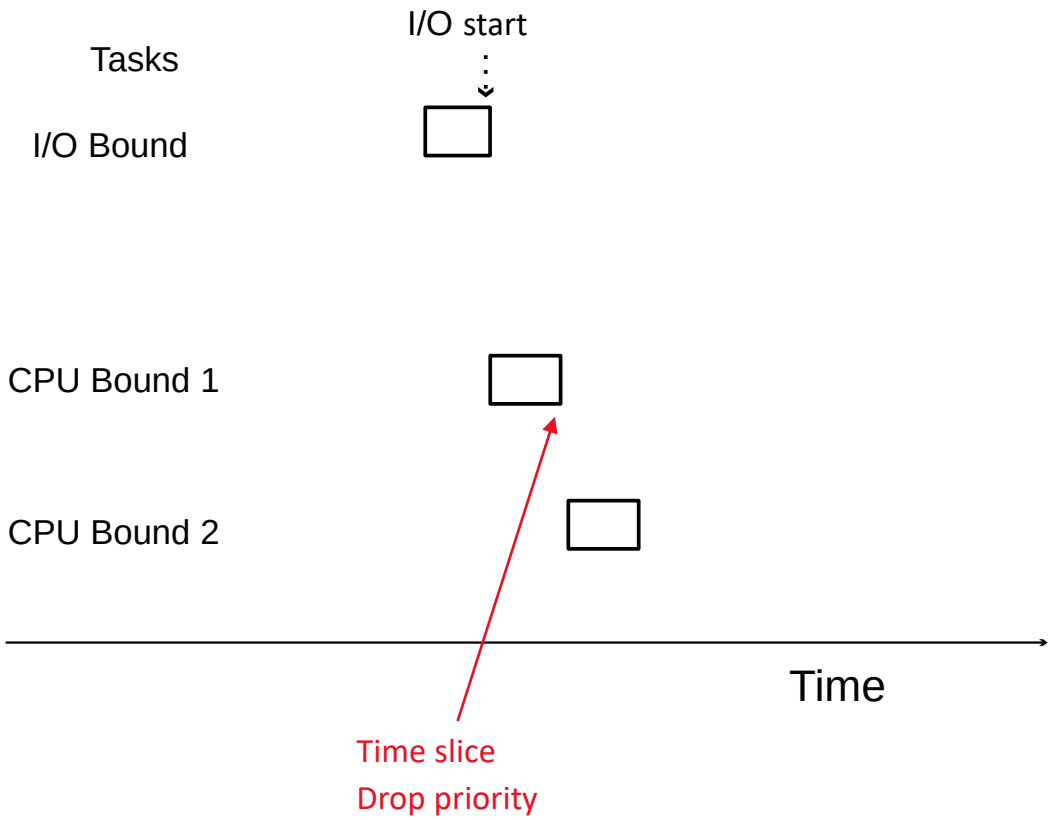
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



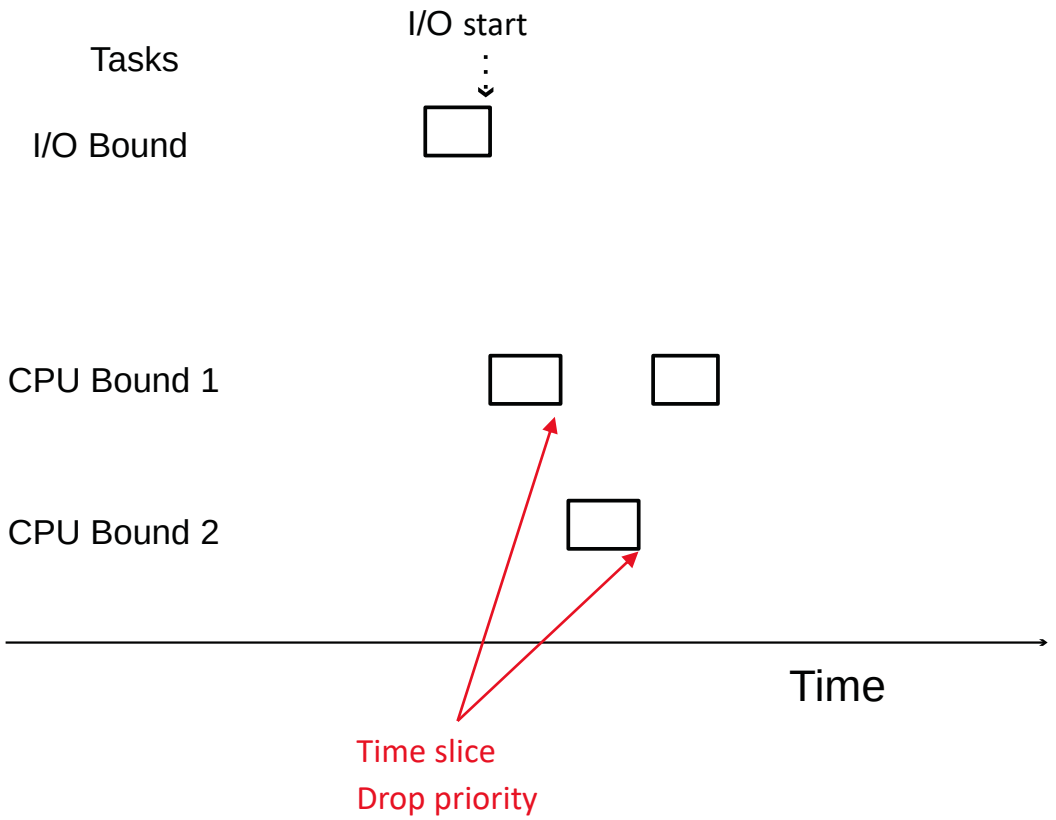
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



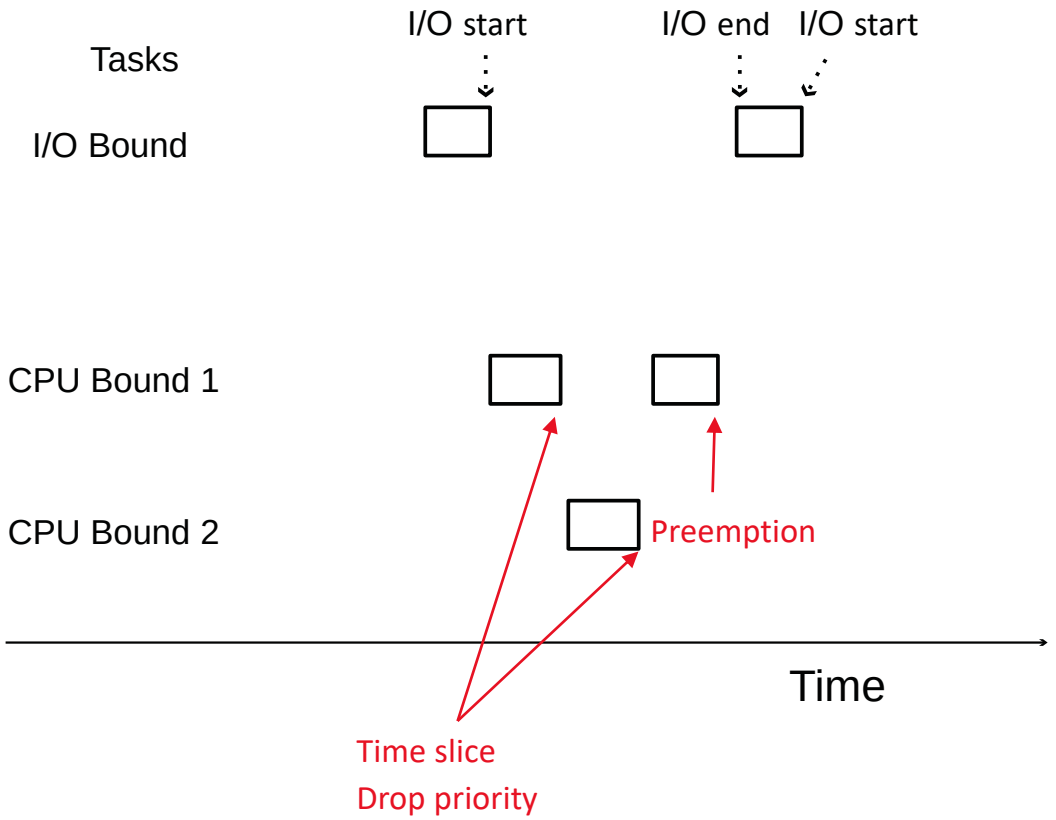
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



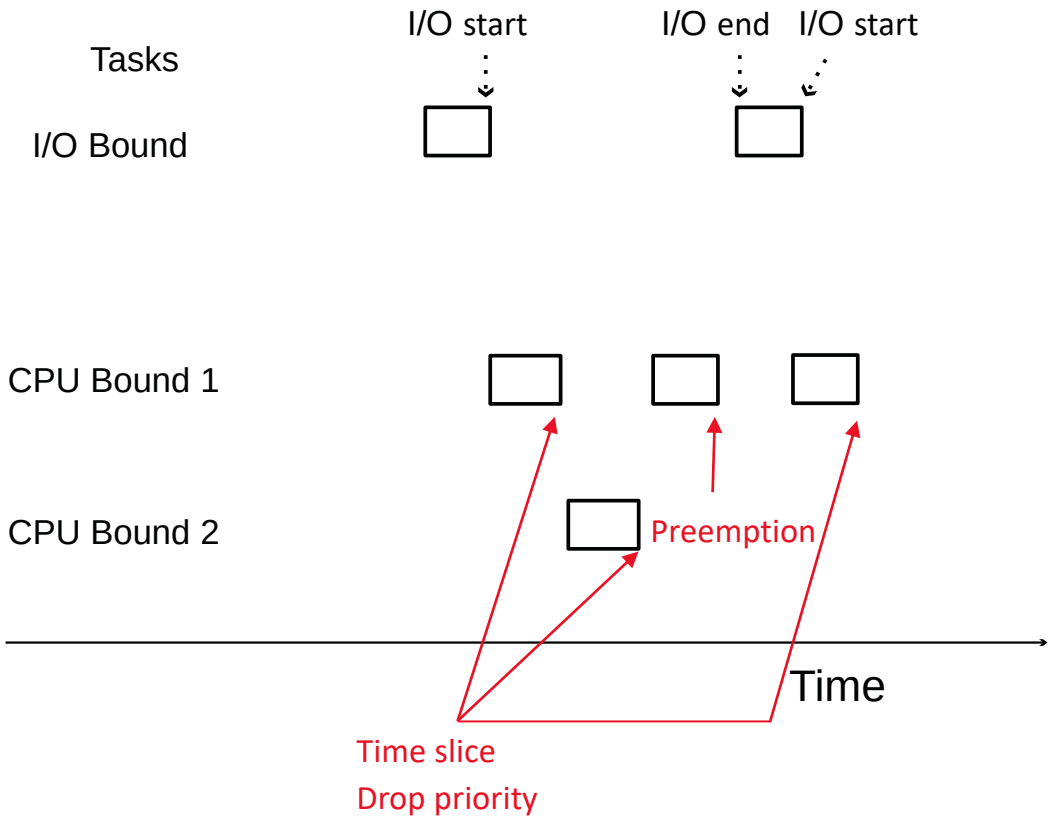
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

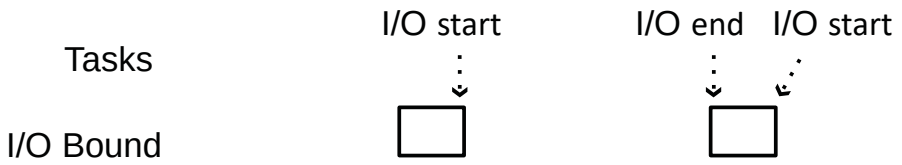
Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

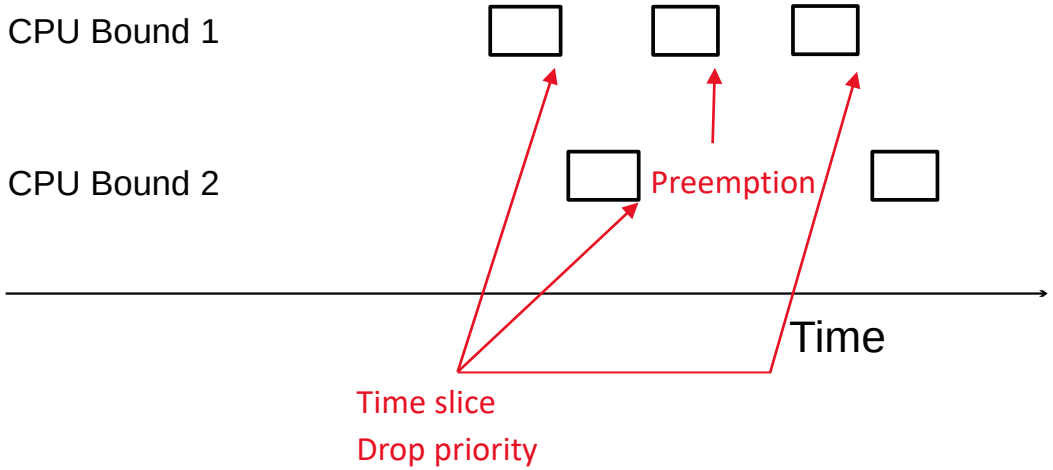


CPU Bound 1 continues here, because its time-slice was extended

Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

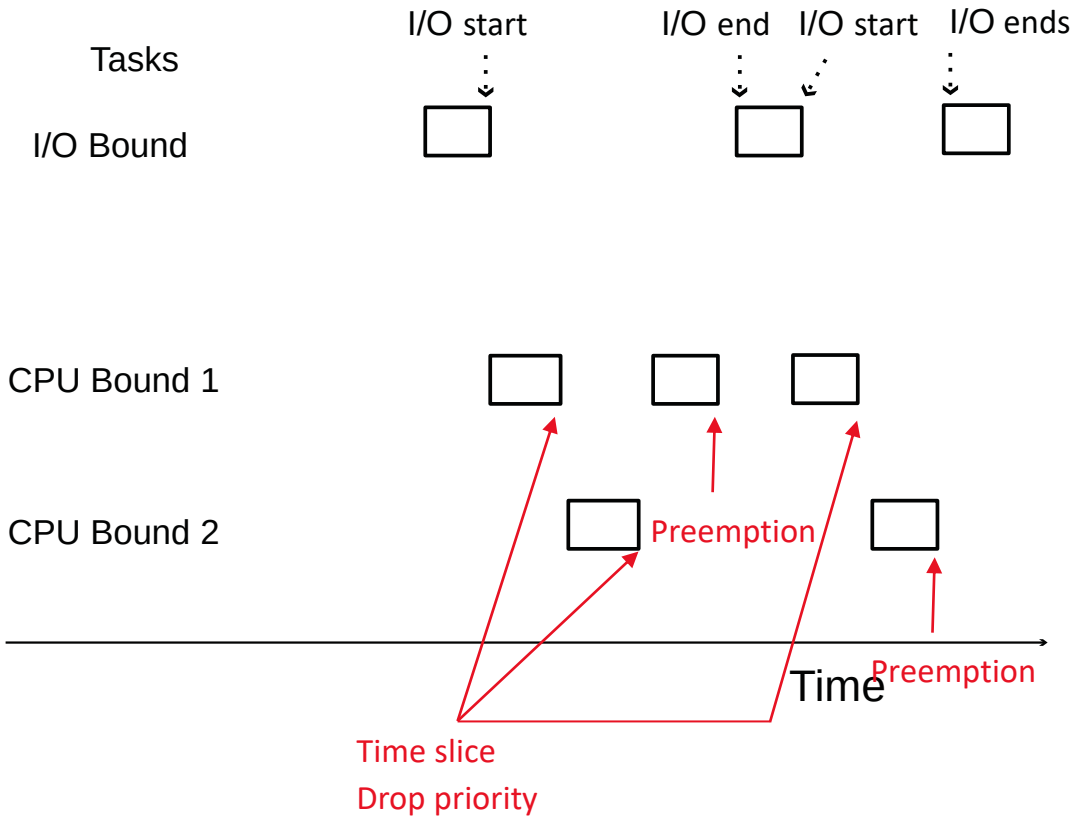
Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



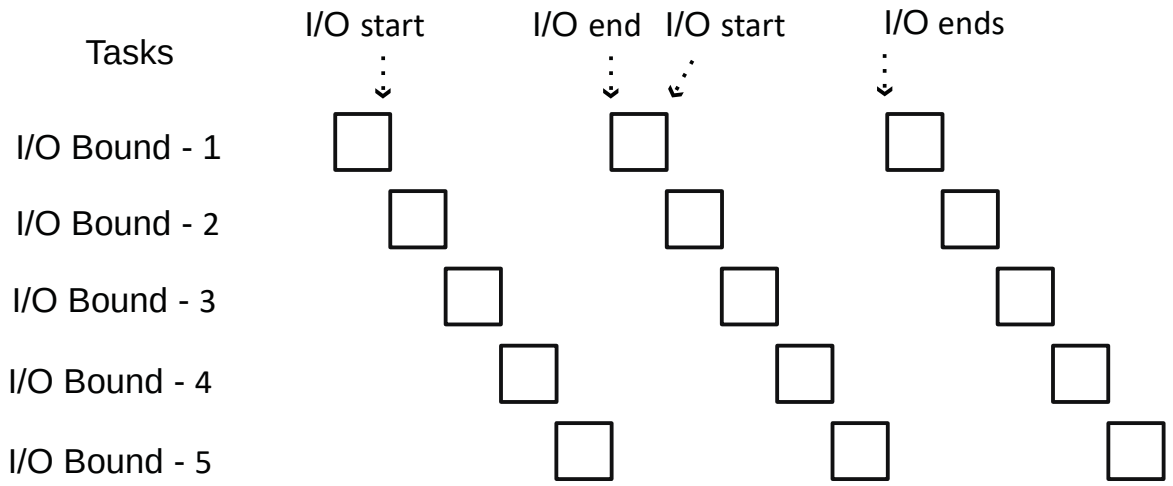


Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



MFQ ***Downside*** Example: I/O-bound Starvation



Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

CPU Bound -
At low priority

Starvation - bump to higher priority level

At the limit, the CPU-bound task should get close to 1/6th of the CPU

Fair Queueing

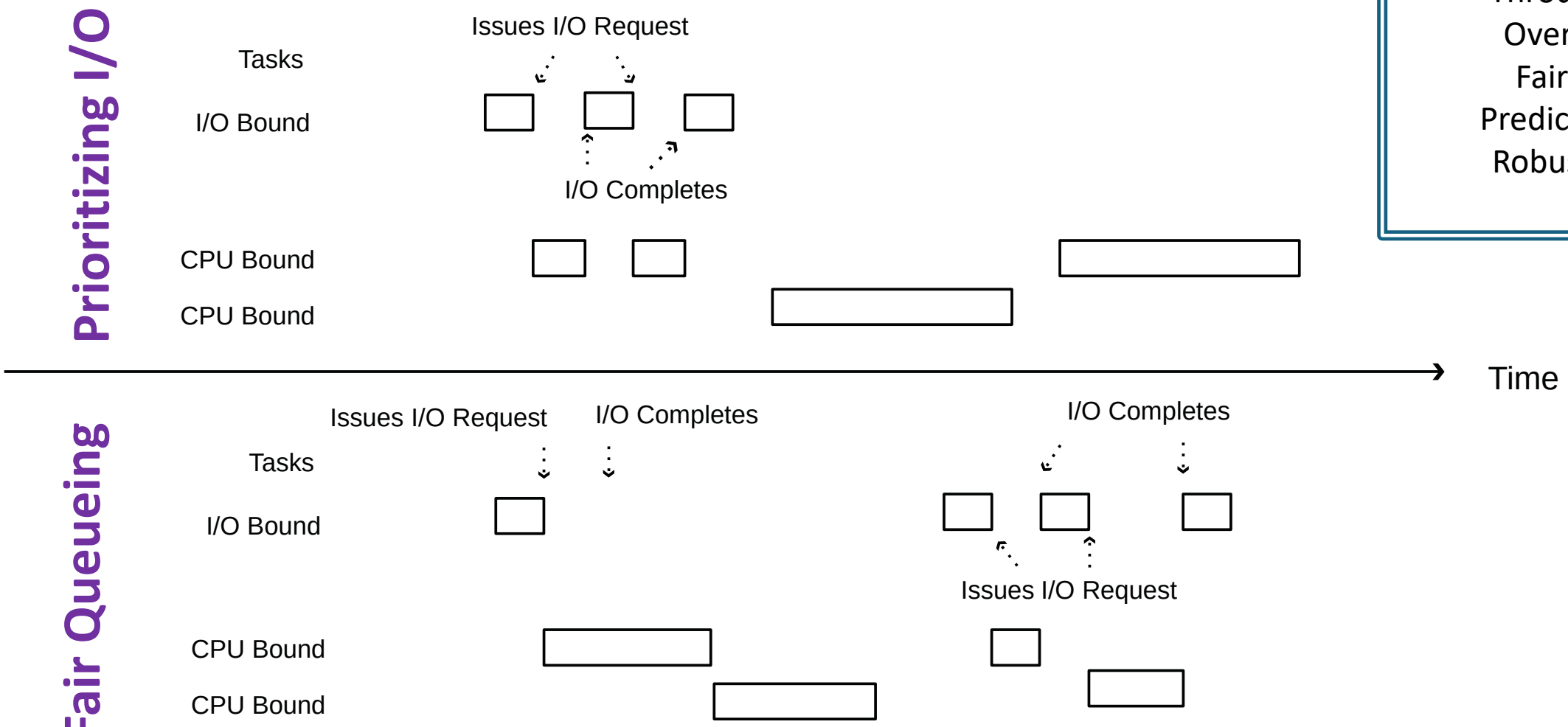
Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

- **Max-Min Fairness**
  - Maximize the minimum allocation to any task
  - Good for repetitive tasks (comes from computer networking, where workloads can be very predictable)

- Implementation:
  - Keep track of CPU usage over time (including time spent waiting for I/O)
  - Schedule the task with the smallest accumulated CPU time, +/- time quantum
  - Tasks that needs less than a “fair share” get as everything they need
  - Scheduling is recursively repeated until left-over resources are divided evenly among remaining tasks

Comparisons

Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



Uniprocessor Summary (1)

- 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.

Uniprocessor Summary (2)

- 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, especially with a long quantum
- MFQ attempts to prioritize latency sensitive tasks
  - MFQ approximates SJF => High variance for long jobs
  - Under high load, MFQ approximates round robin (everyone at highest priority)