

CSE 451: Operating Systems, 26au
L12: 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

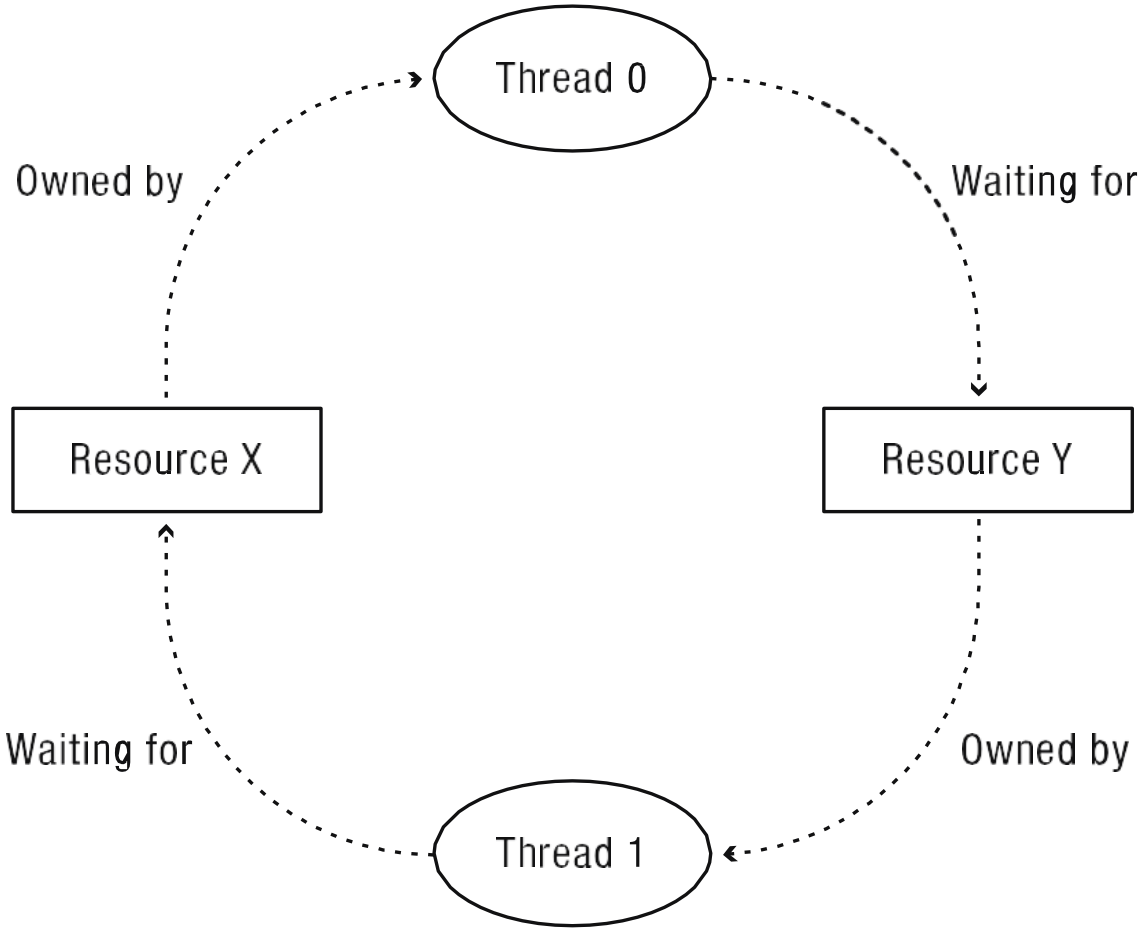
Administrivia

- Due this Wednesday @ midnite 🦇👻🎃
  - PSet 2
  - Lab 2 Part 2 design doc

- Midterm
  - In class on 11/6 (a week from Friday)
  - Covers through CPU scheduling (this slide deck)
  - Allowed one 8.5x11” of handwritten notes on both sides
  - Will release practice exam
  - Will have a review session in-class on 11/4
  - Remember: your goal isn’t a perfect score, your goal is to flex what you do know 💪💪💪

Deadlocks (wrapping up)

Deadlocks, illustrated



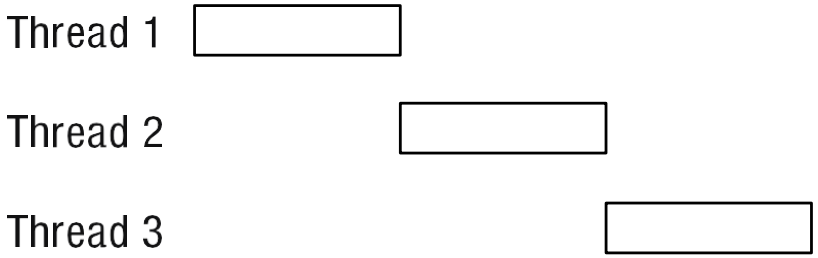
Alternative to playing it safe: **“Detect and Repair”**

- 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

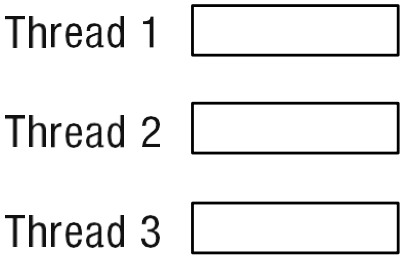
CPU Scheduling

Concurrency

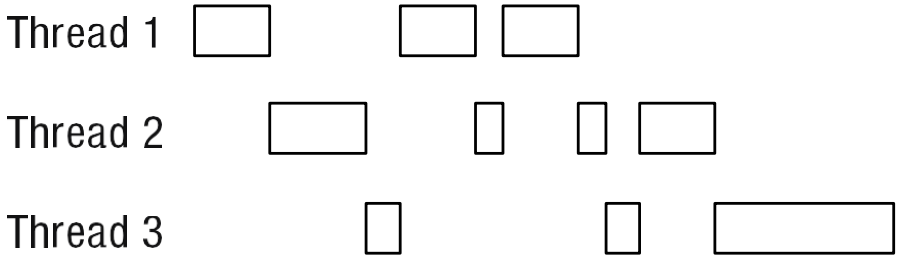
One Execution



Another Execution



Another Execution

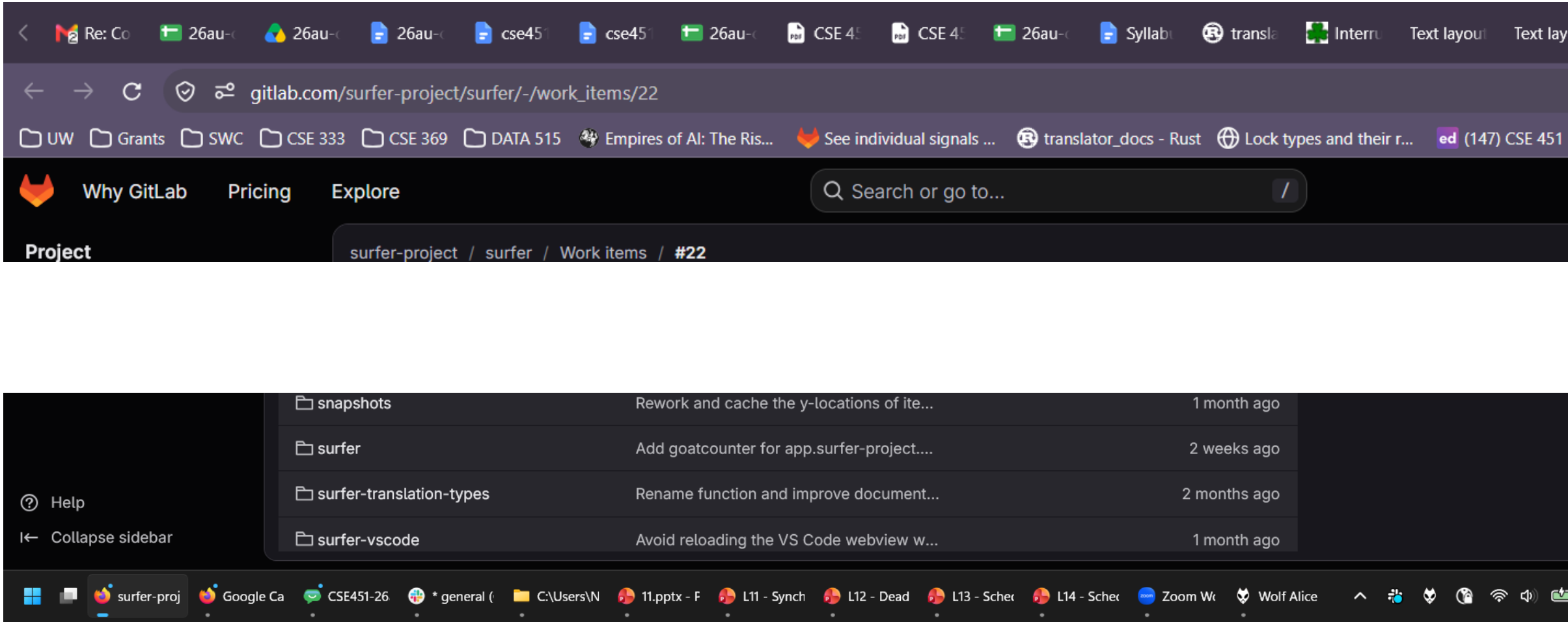


Scheduling

- 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)

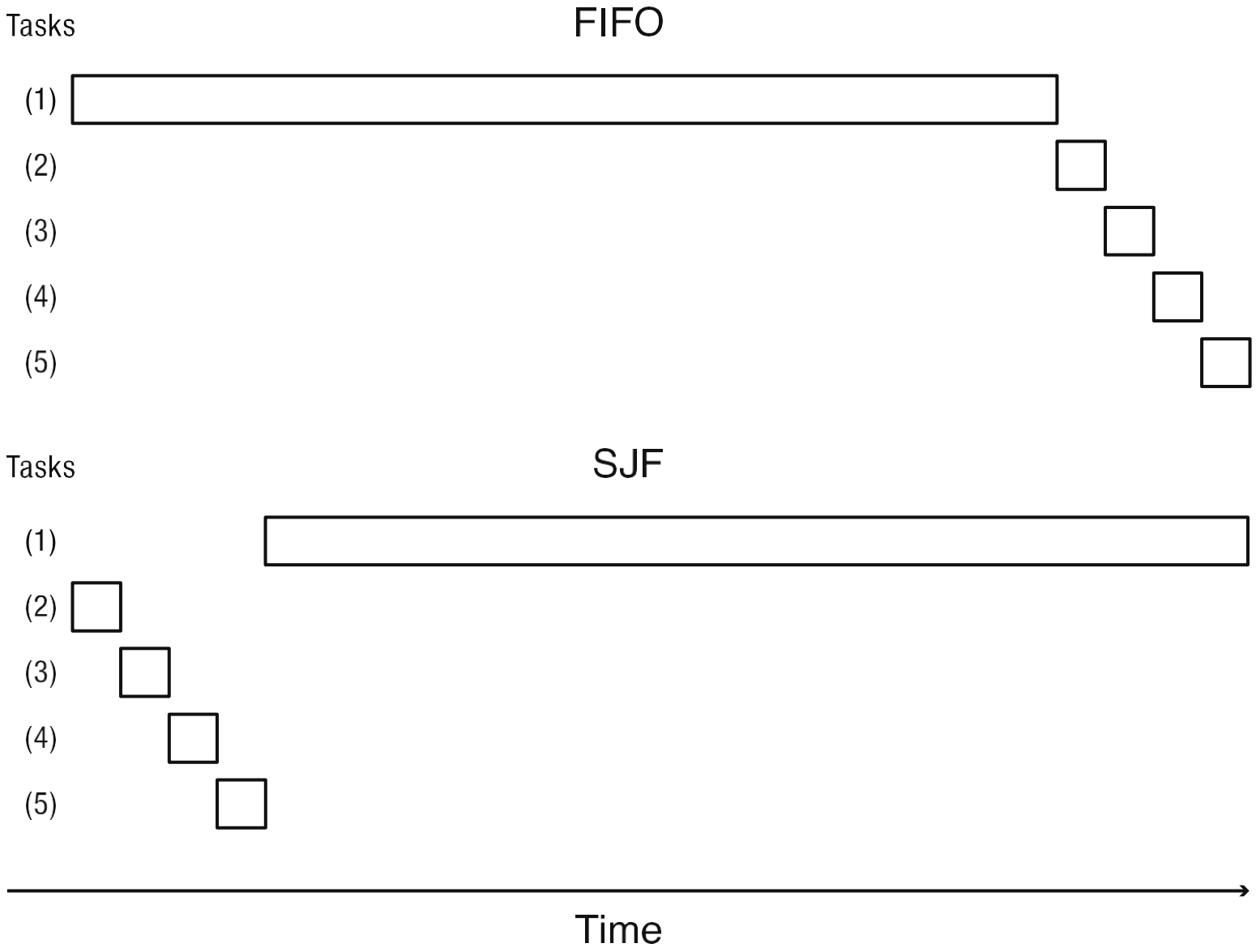
- Always run the task with the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

- 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

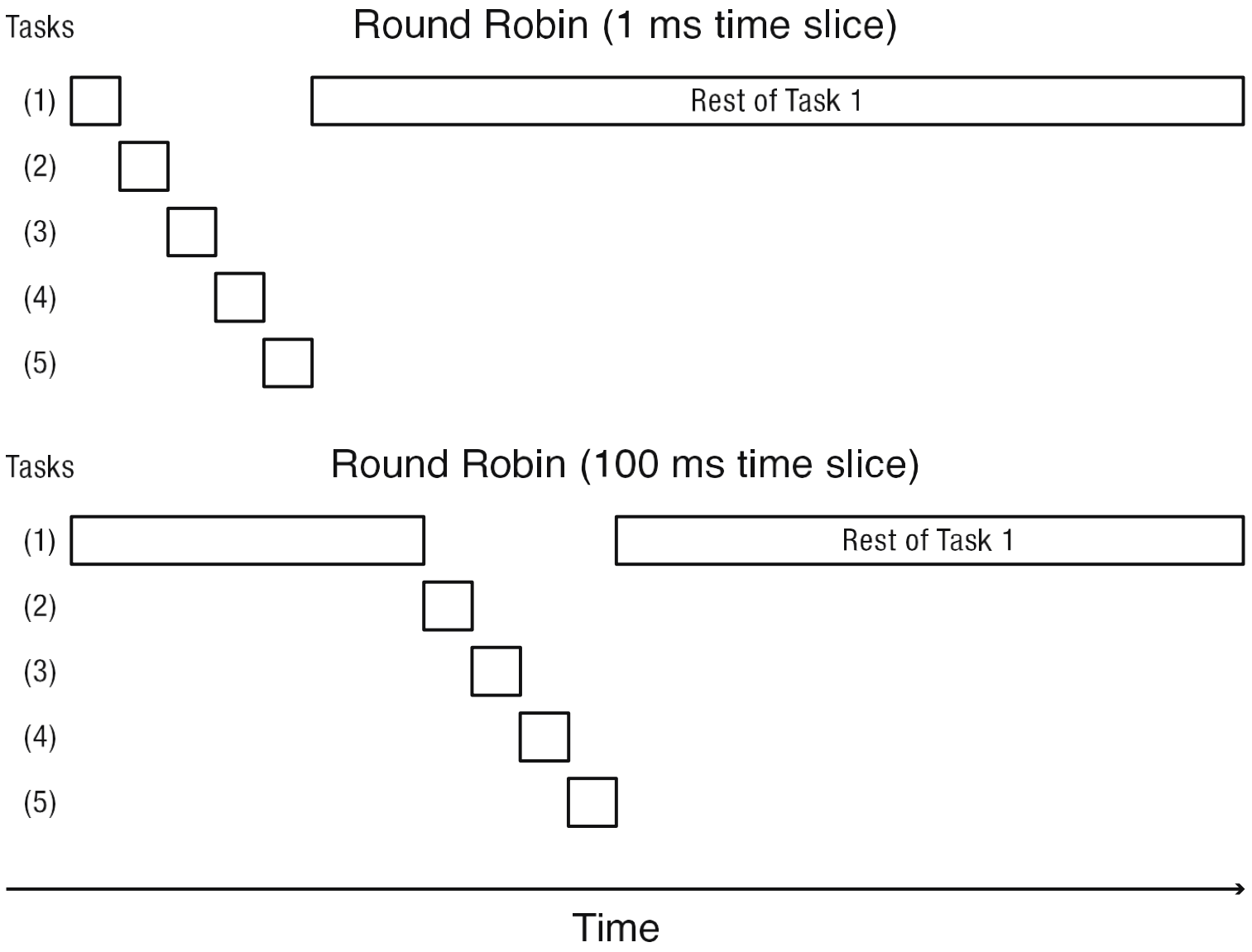
Round Robin

- 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

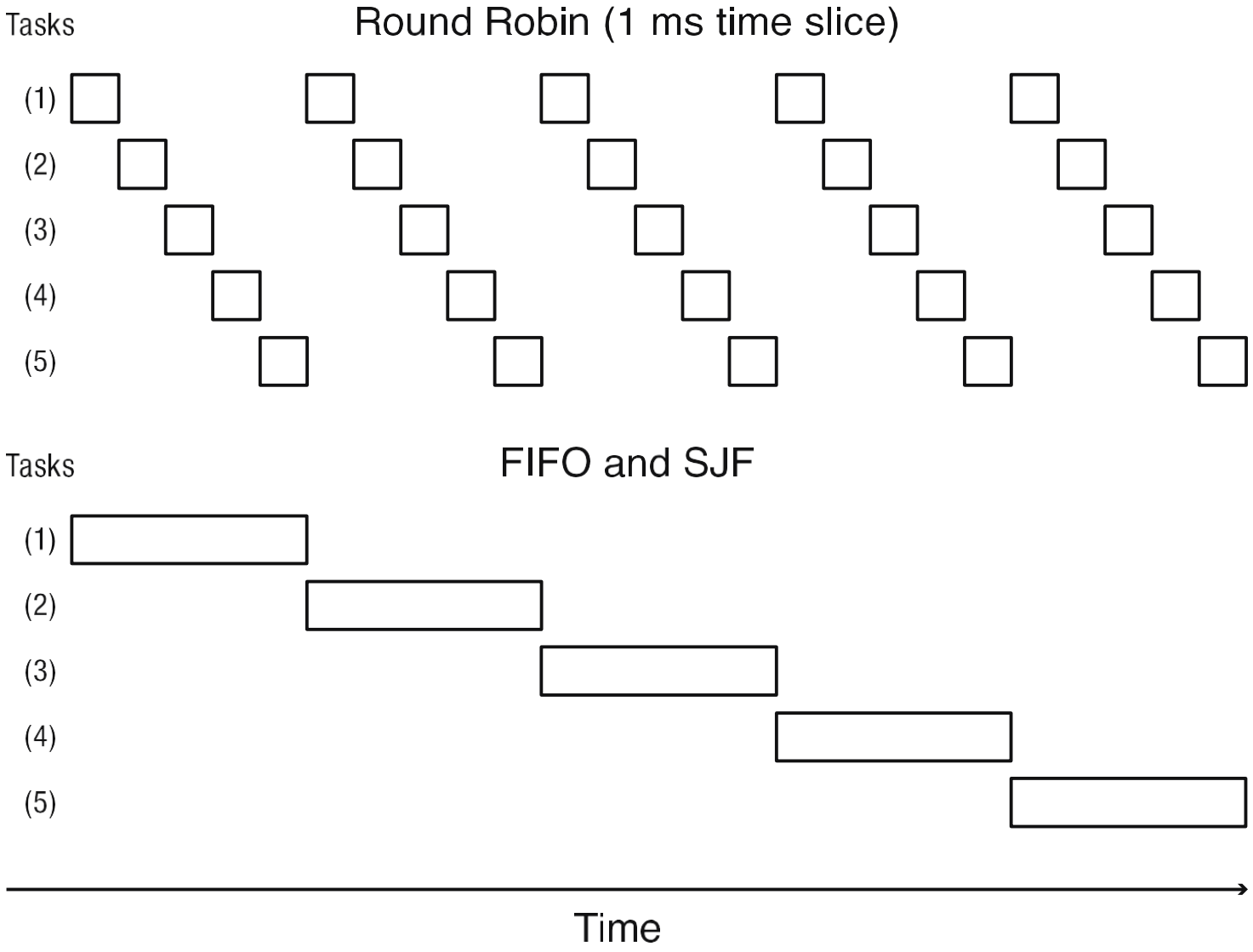


Round Robin



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

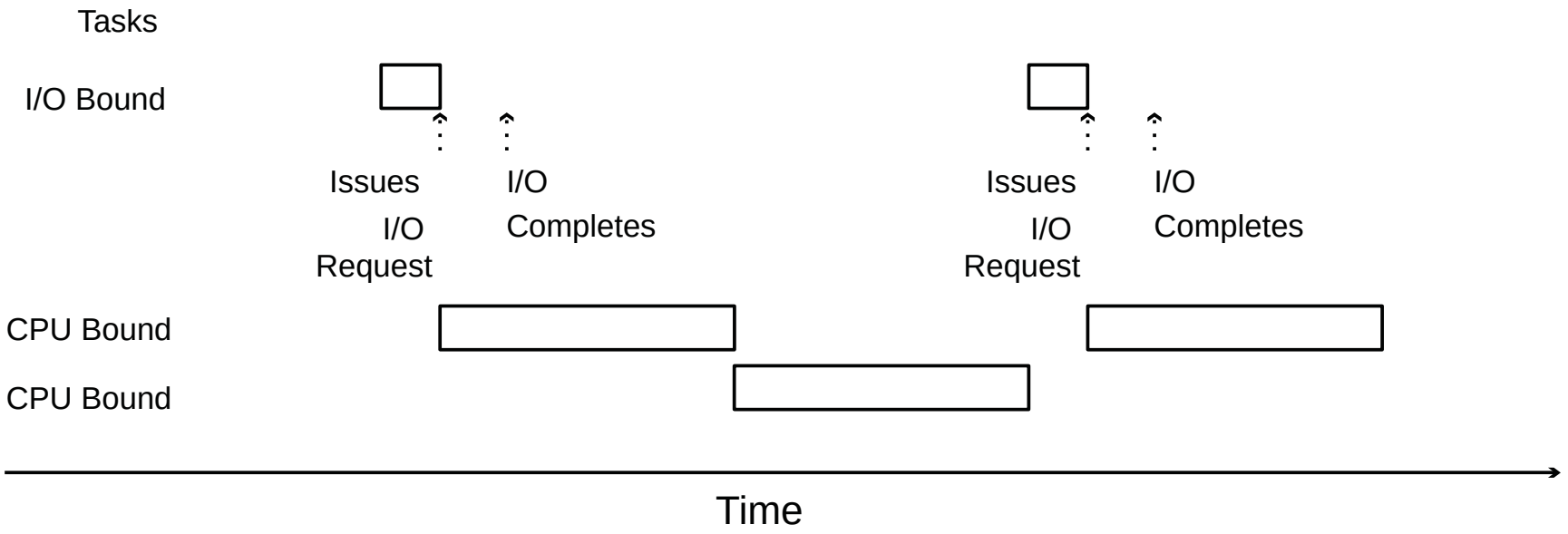
Round Robin == Fairness?

- 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?

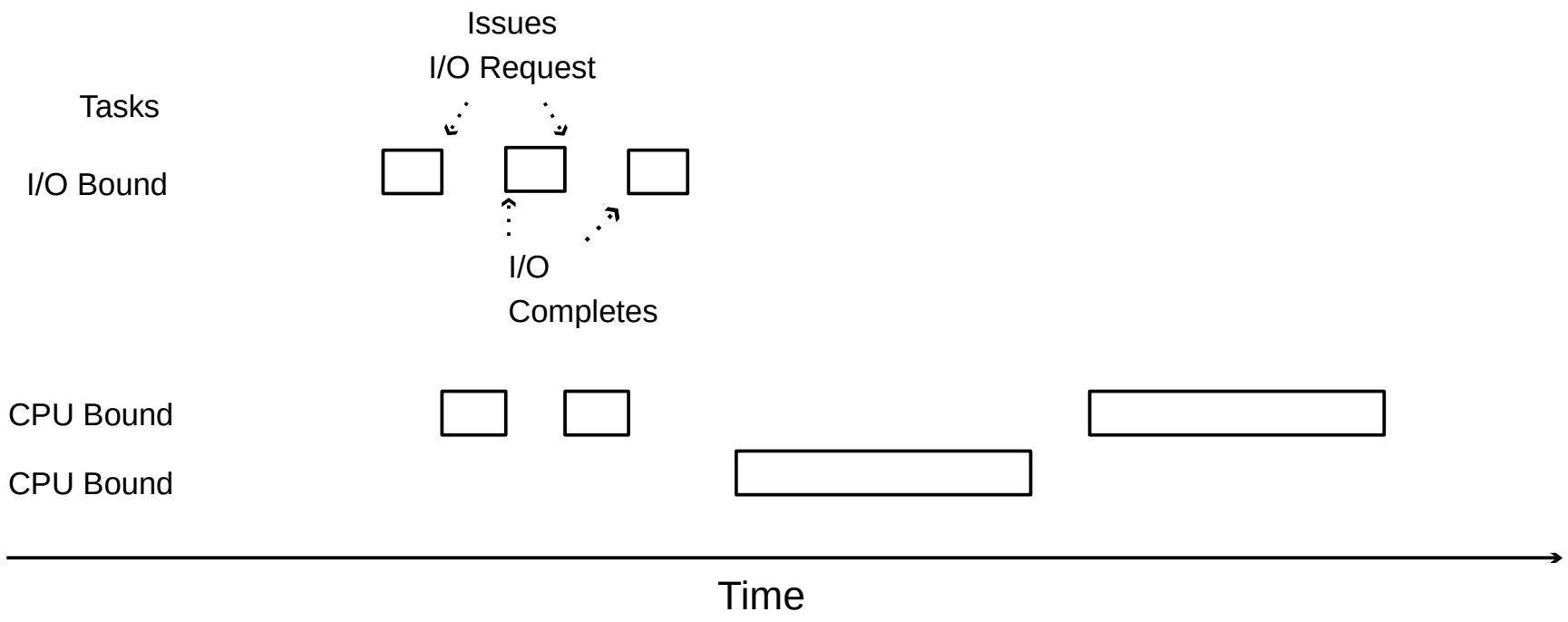
Mixing I/O-bound and Compute-bound Tasks (Round Robin)

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

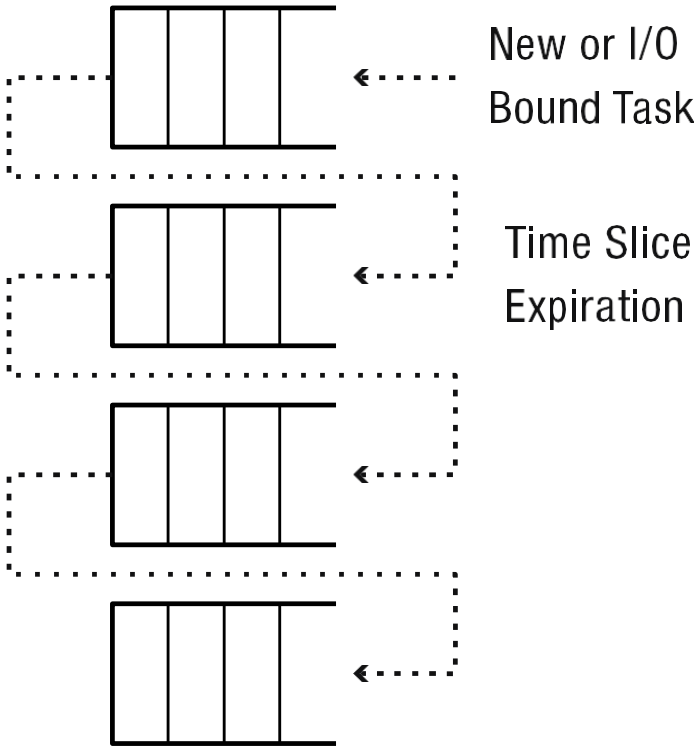
- Rules:
  - Task starts in highest priority queue
  - If time slice expires, task drops one level
  - If low priority tasks are being starved, bump them up a level

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

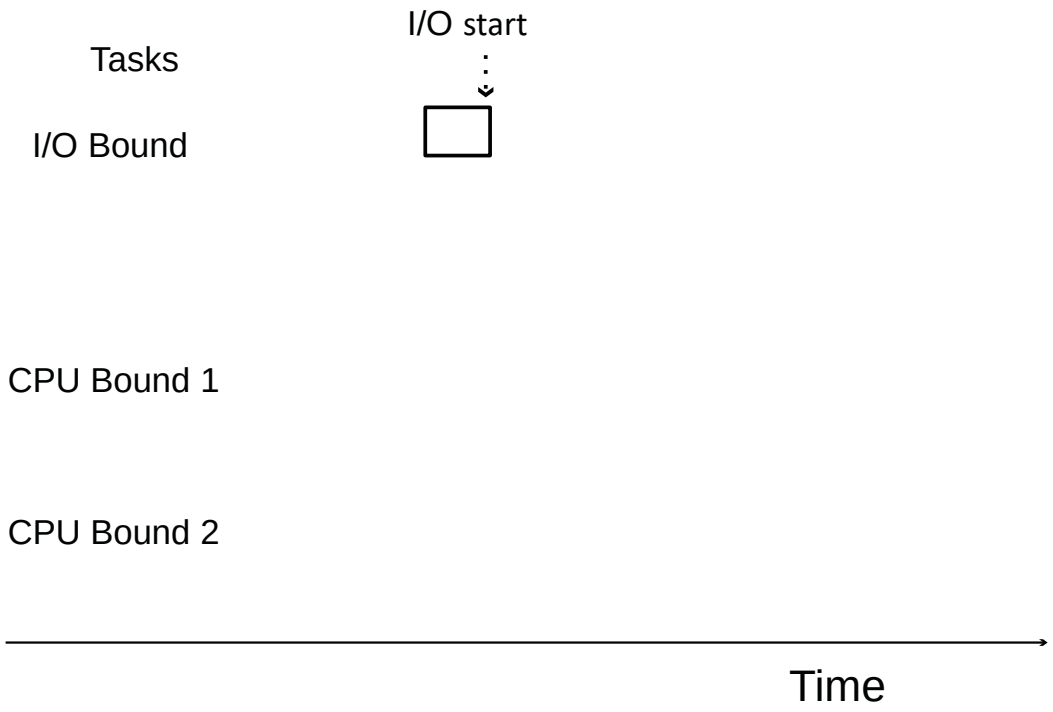
Round Robin Queues



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

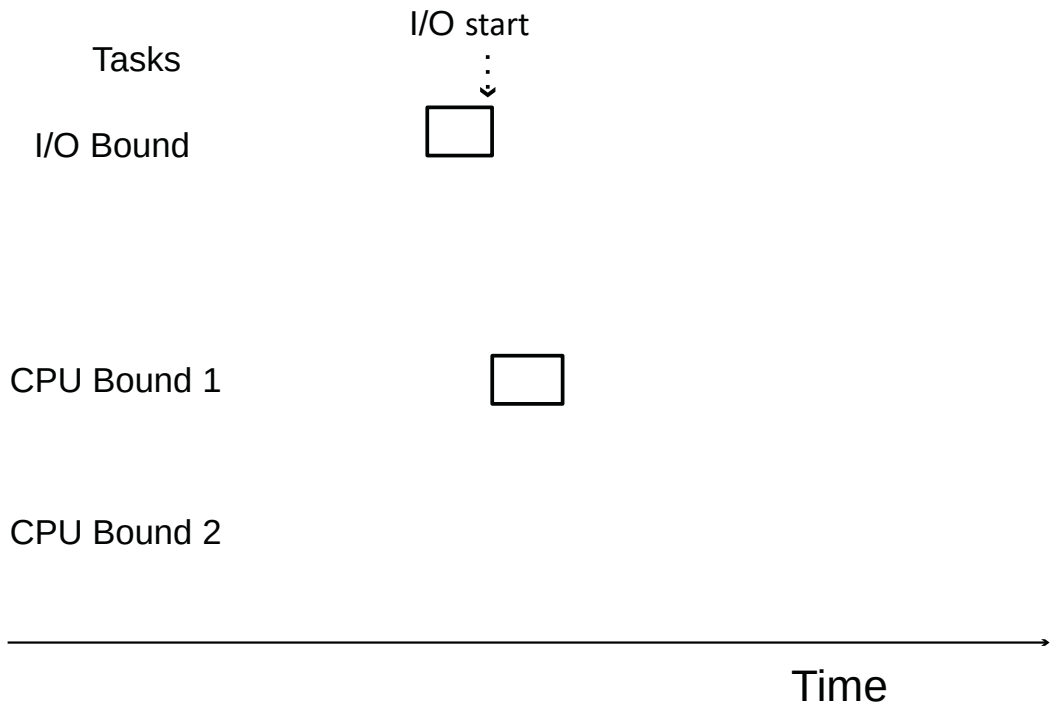
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

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



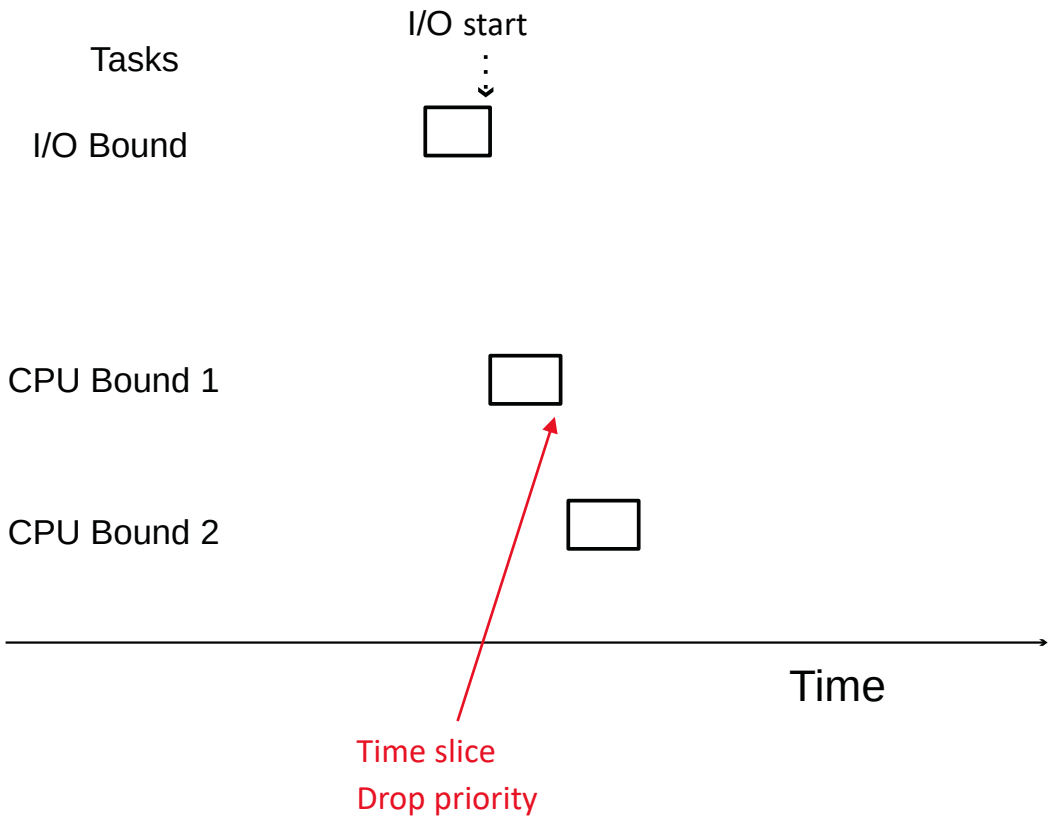
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

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



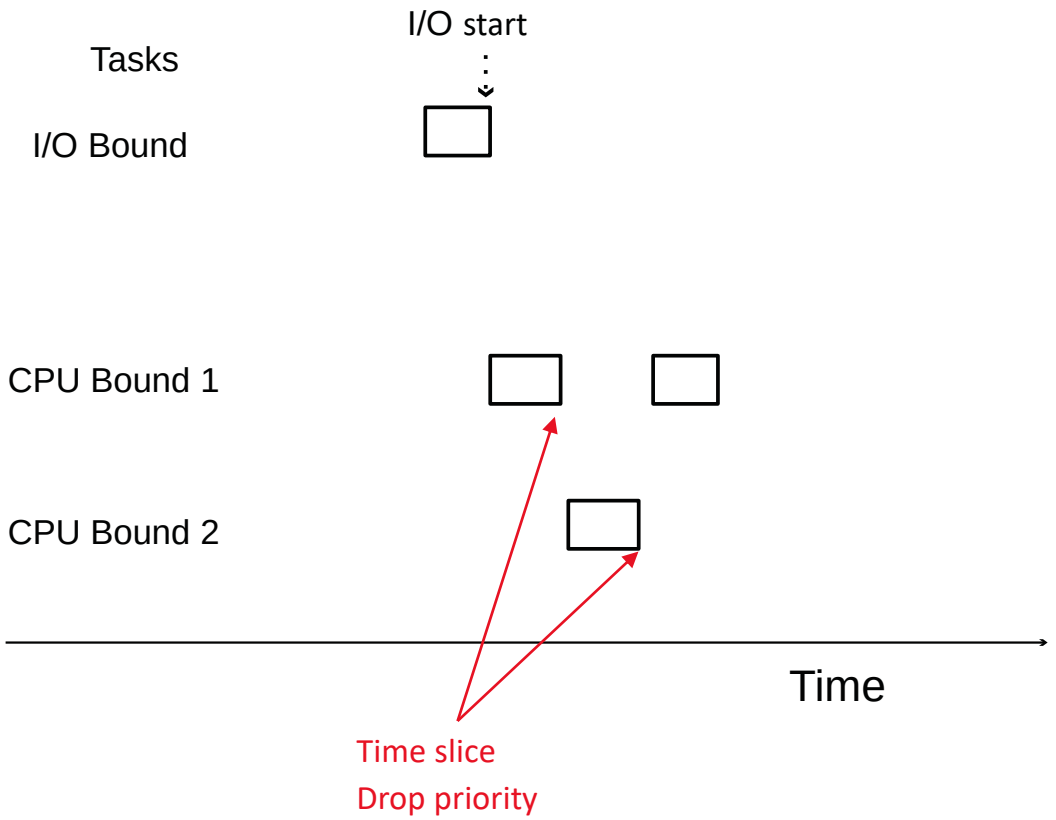
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

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



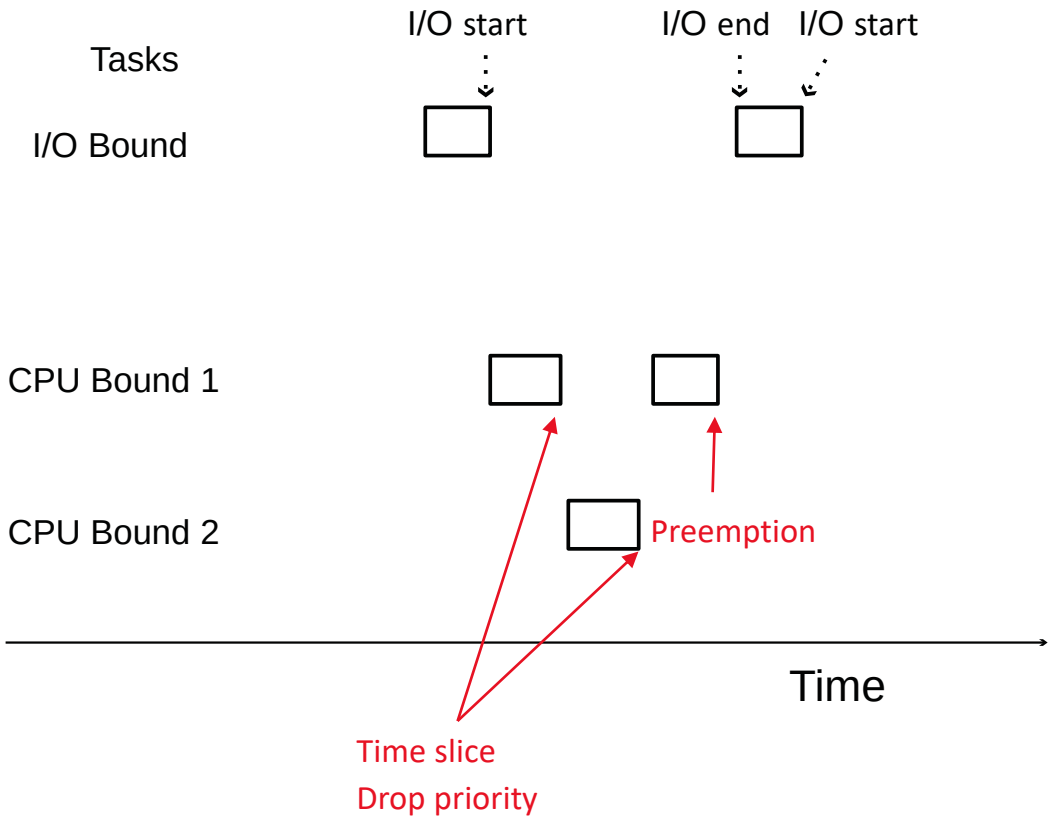
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

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



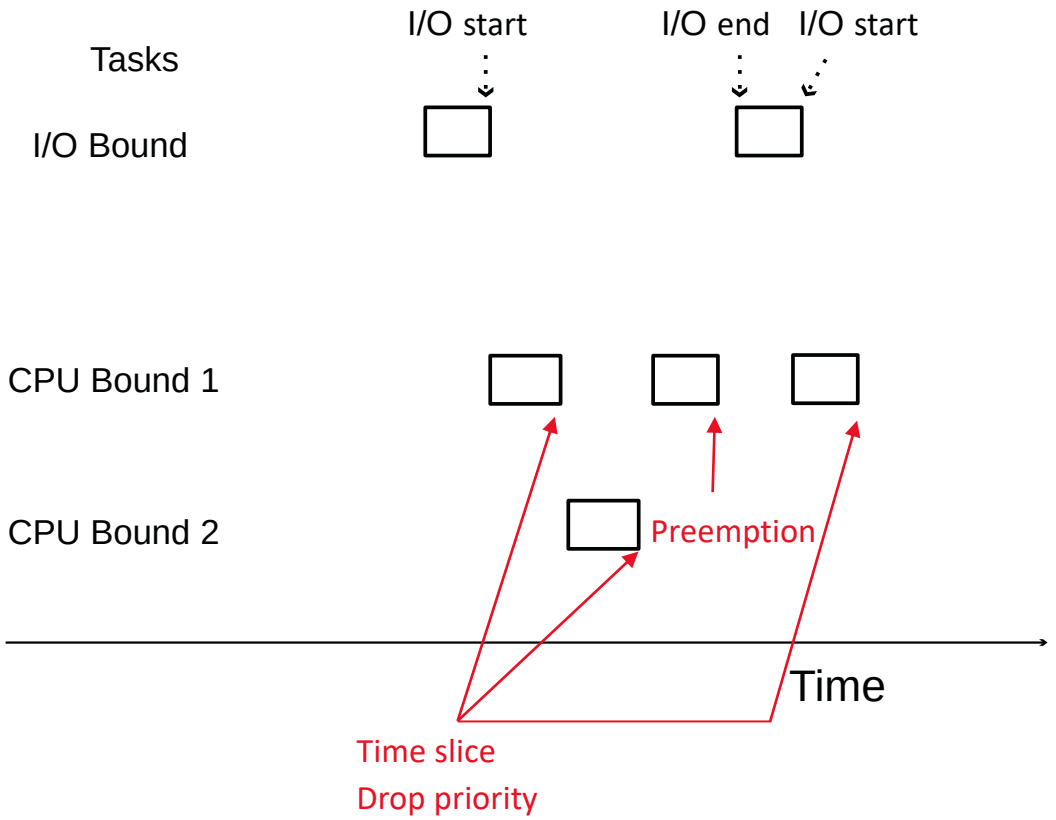
Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

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



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

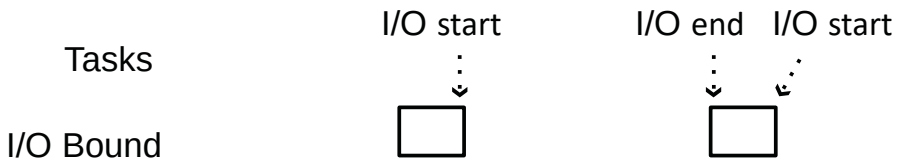
Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

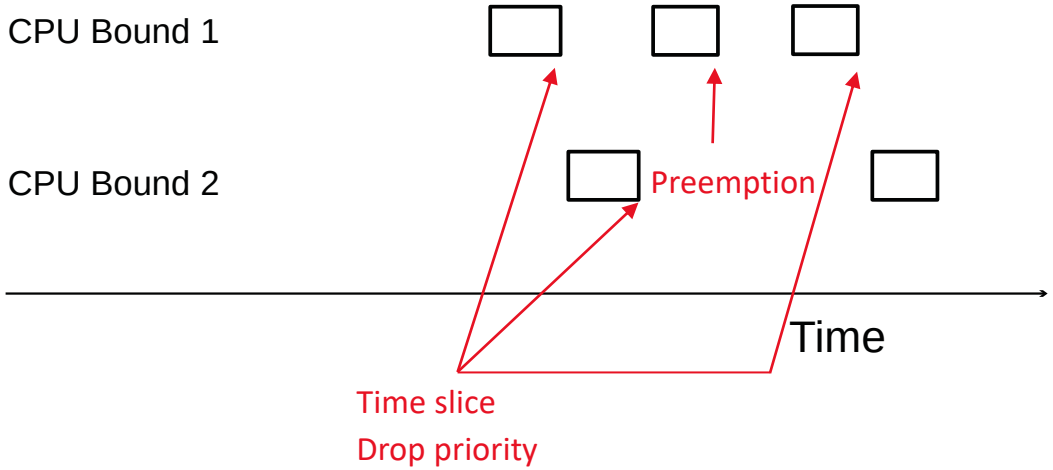


CPU Bound 1 continues here, because its time-slice was extended

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

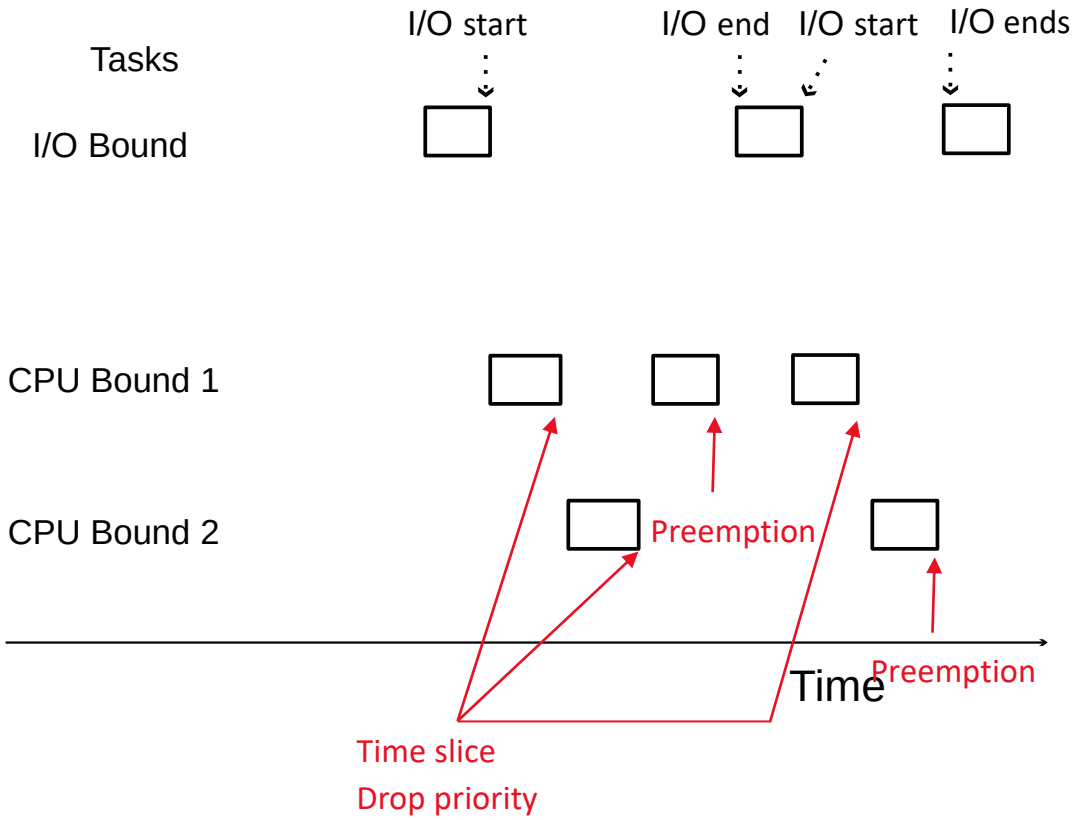
Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness



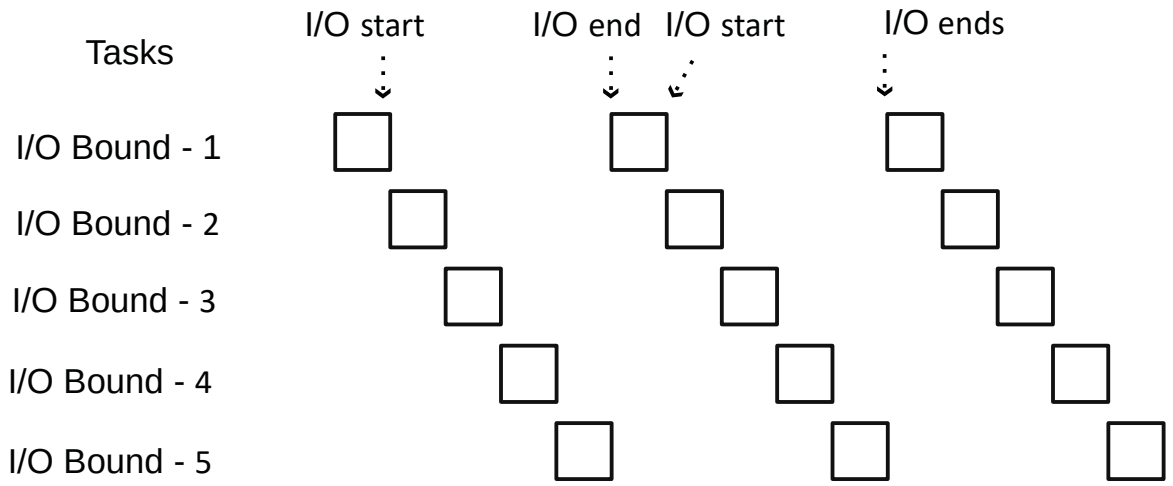


Example: Workload with I/O and Compute Tasks (Multilevel Feedback Queues)

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

Class concluded here on 10/26 and will resume on 10/28



Fair Queueing

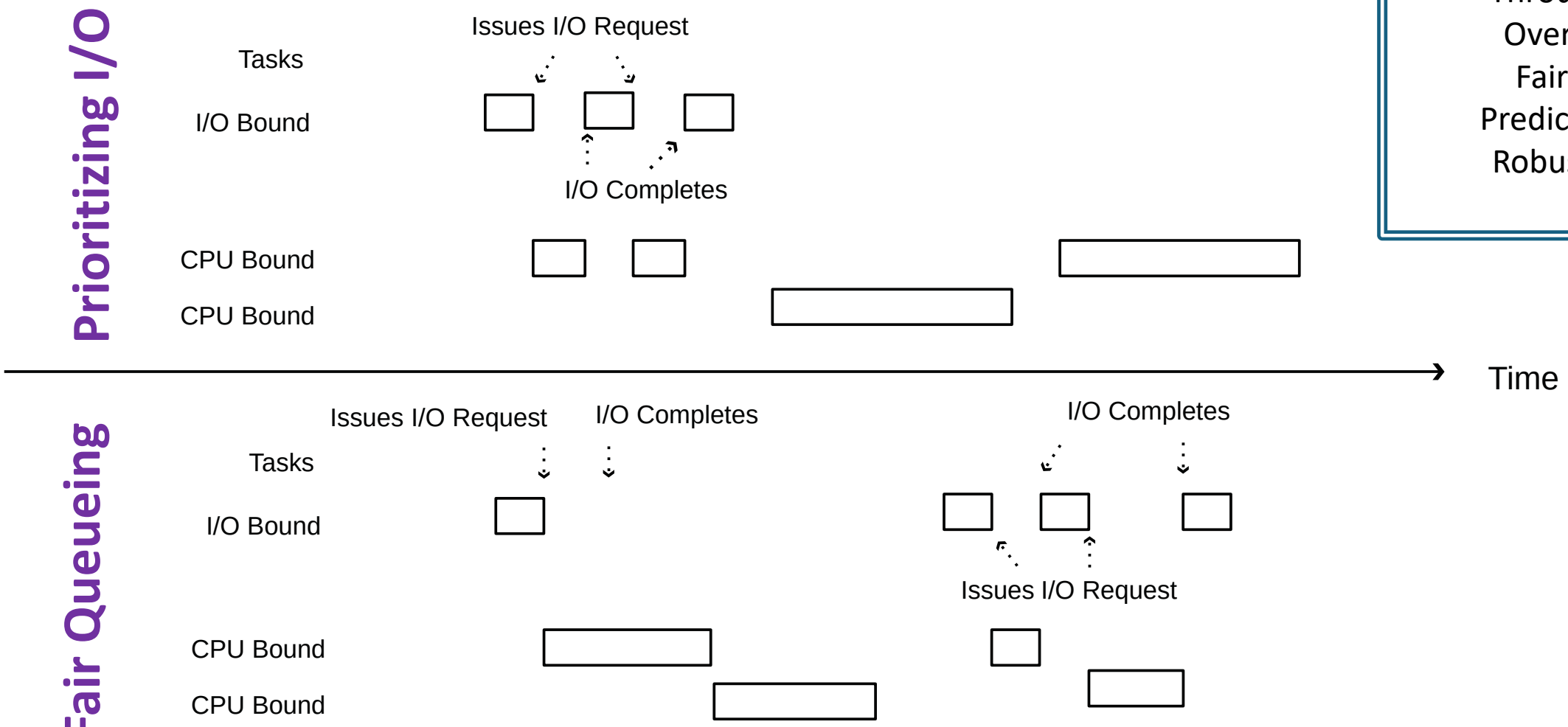
Metrics:
Avg. response time
Max response time
Throughput
Overhead
Fairness
Predictability
Robustness

- **Max-Min Fairness**
  - Maximize the minimum allocation to any task
  - Good for repetitive tasks (comes from computer networking, where workloads can be very predictable)
  - Effect: tasks that need less than a “fair share” get everything they ask for

- 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
  - 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)

Multi-processor Scheduling Policies

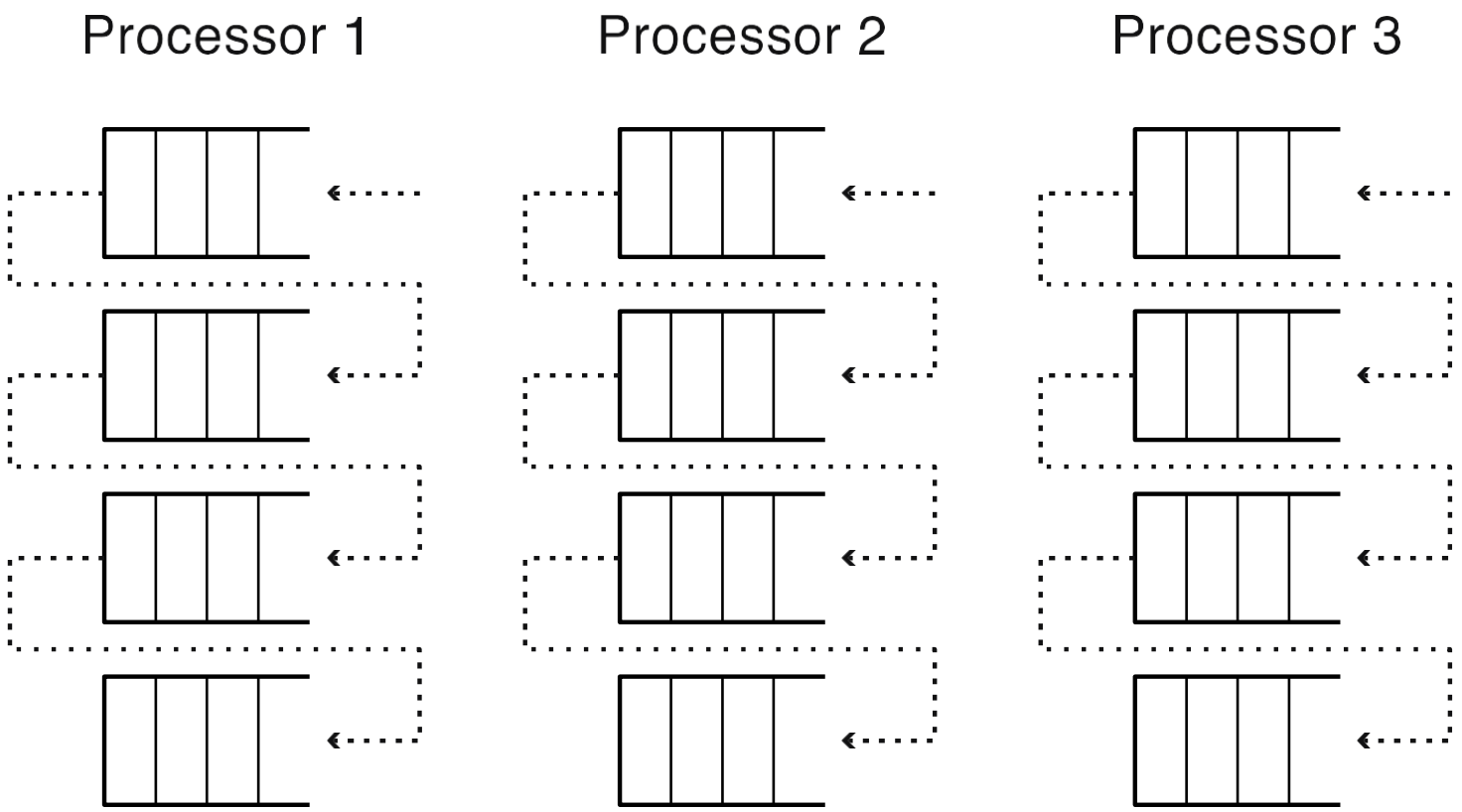
More processors, more problems

- Tasks that move to a different core will start with a cold cache and might take longer than just waiting longer for their current core
- Kernel structures need to be synchronized across cores
  - Necessitates a scheduler spinlock
  - On multi-*processor* systems, high-traffic data like the process table can ping-pong between caches

Mitigation Technique 1: Per-Processor Affinity Scheduling

- Each processor has its own ready list
  - Protected by a per-processor spinlock
- Put threads back on the ready list where it had most recently run
  - Ex: when I/O completes, or on Condition->signal
- Idle processors can steal work from other processors

Per-Processor MFQ w/ Affinity Scheduling



Tradeoff:

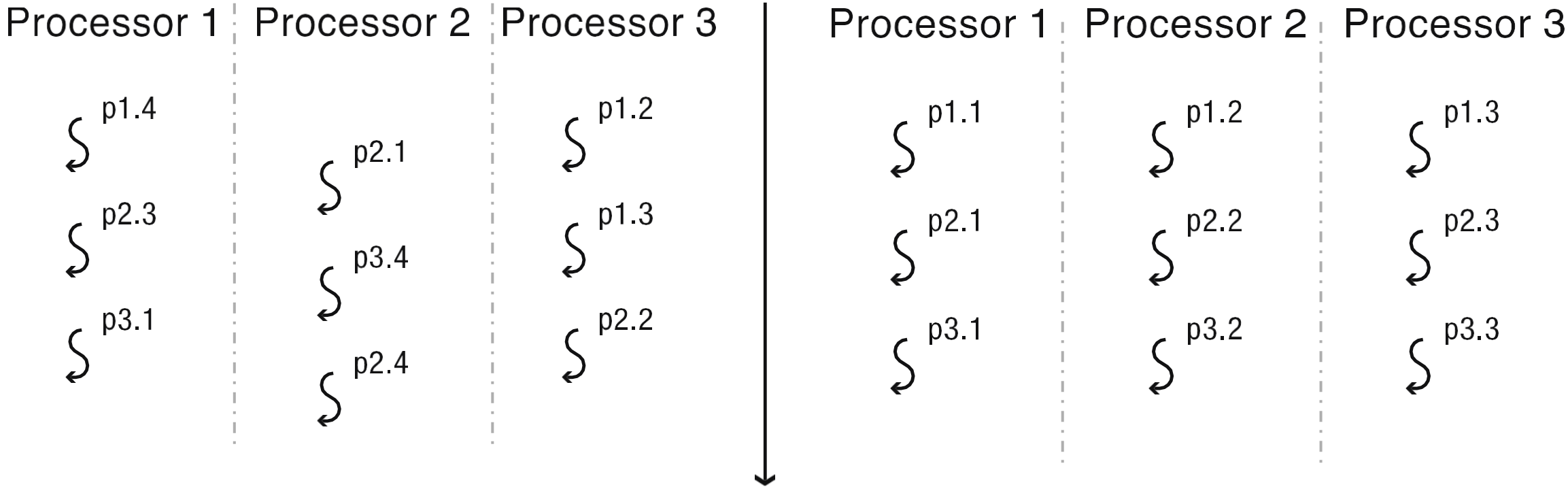
Needs rebalancing in case a lot of high-priority tasks end up on one processor

…But this requires a global view of all queues, and so halts progress on entire machine

Mitigation Technique 2: Gang Scheduling

"Oblivious scheduling": each CPU time-slices its ready list independently. Process threads are scheduled on an "as-available" basis.

"Gang scheduling": CPU time-slices are synchronized. One process is given time on all cores, and then evicted from all cores.

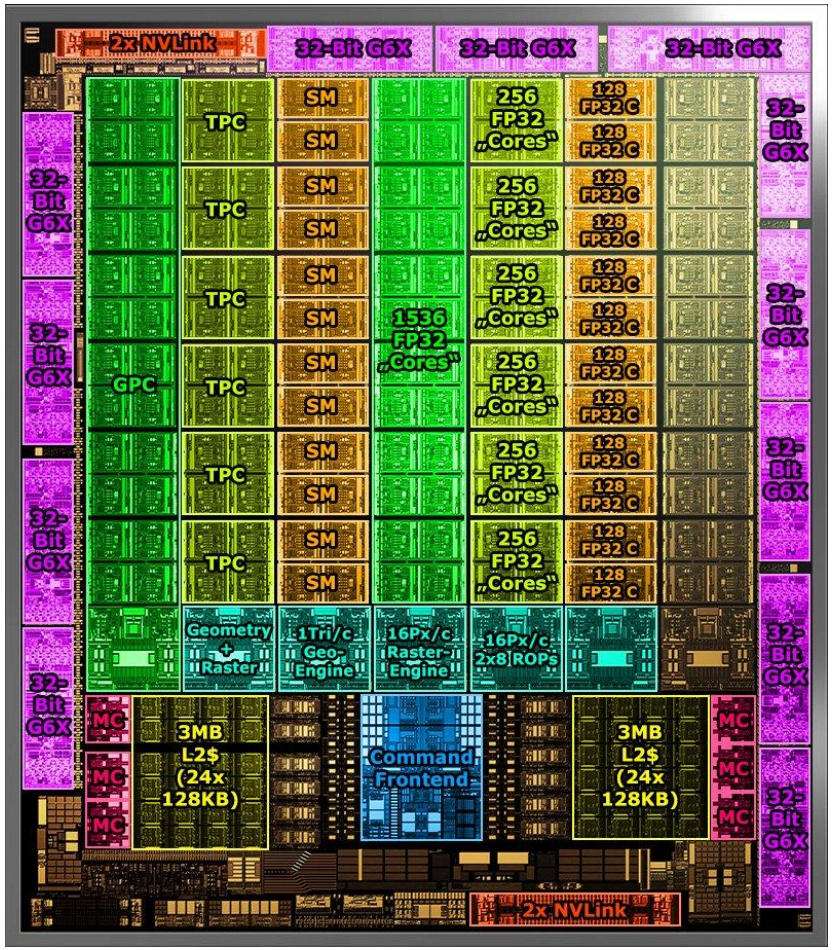


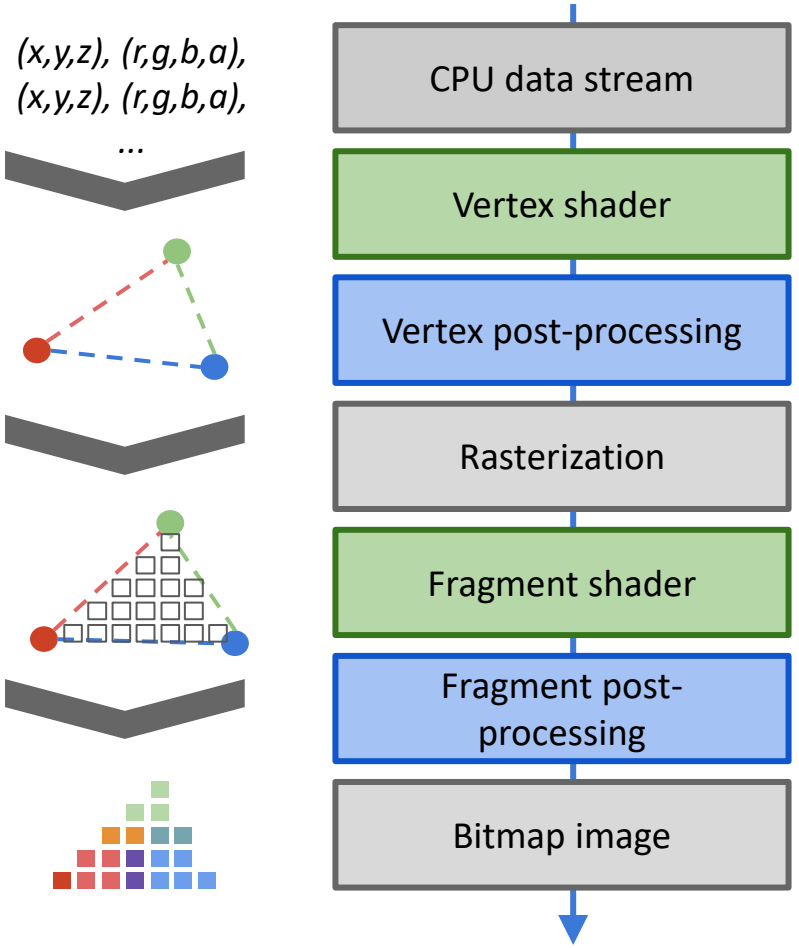
px.y = Thread y in process x

Mitigation Technique 3: Bulk Synchronous Model of Parallelism

- Idea:
  - Farm out small parcels of work to each core
  - Cores work in parallel, but without communication
  - Wait for all cores to finish their work parcels (a "**barrier**")
  - Exchange relevant data and outputs between cores
  - Repeat
- Examples:
  - Graphical rendering
  - MapReduce
  - Machine learning training and inference
  - Physical model simulations (fluid flow over a wing)

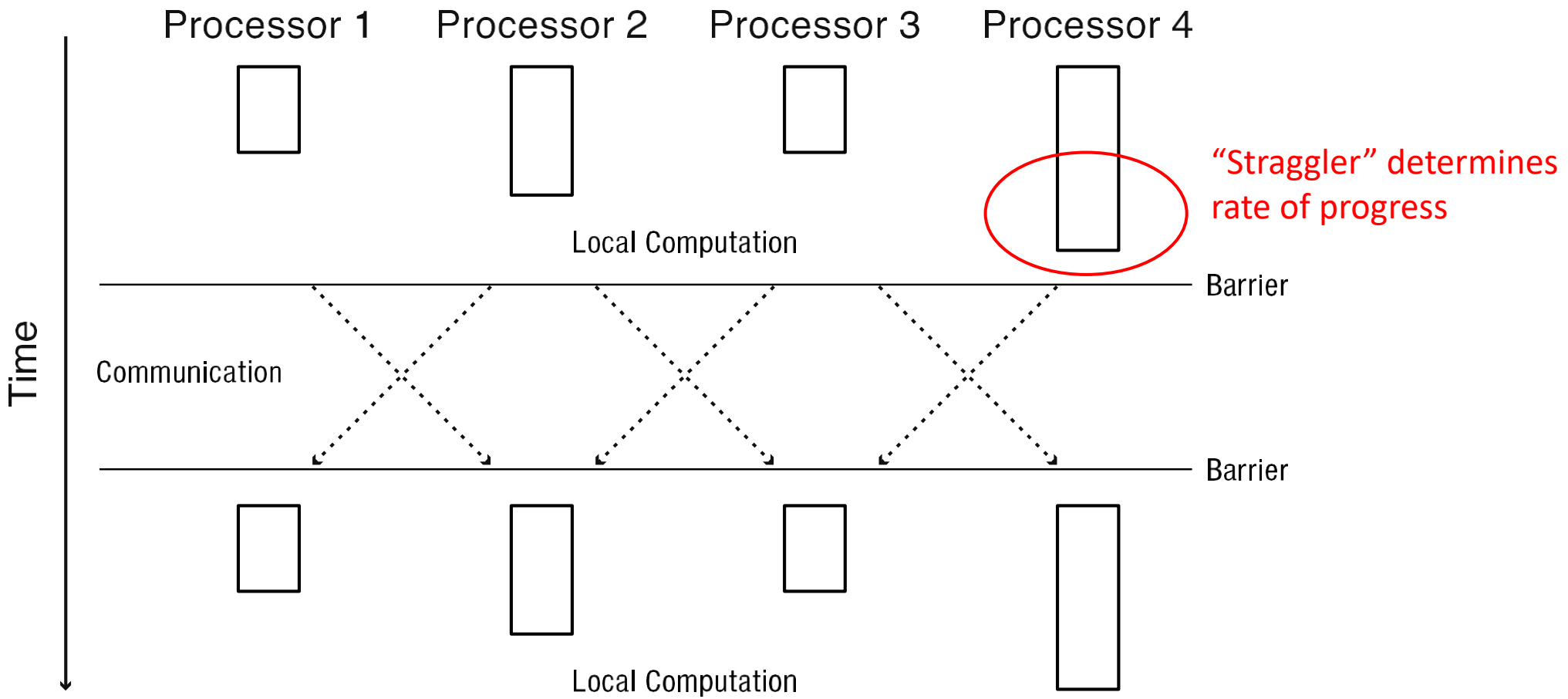
GPU as bulk synchronous processor



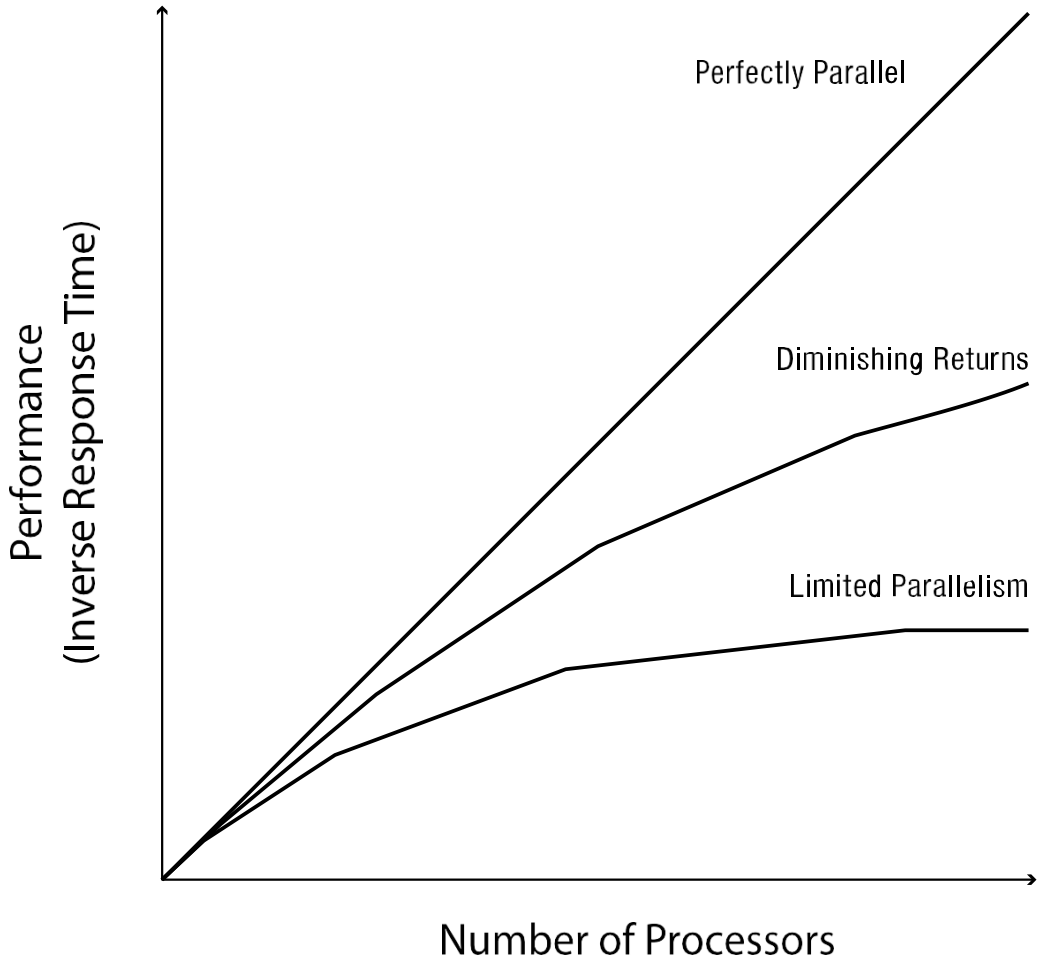


https://www.tomshardware.com/news/infrared-photographer-photos-nvidia-ga102-ampere-silicon

Bulk Synchronous, Abstractly



Parallelism isn't a magic solution



Little to no overhead to running in parallel, all cpus used

Some Overhead, some cpus underused

Can make use of additional cpus, but only up to a point (lock contention, tail latency)

This is the end of the content covered on the midterm!