Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 15: Scheduling

Nicole Hamilton

https://web.eecs.umich.edu/~nham/

nham@umich.edu

Agenda

1. CPU scheduling.
2. Project 3 due July 27.

Agenda

1. CPU scheduling.
2. Project 3 due July 27.

CPU scheduling

If more than one thread is ready, the OS must choose which one to run.

Many possible scheduling policies; we'll explore a few.

Real schedulers often employ a complex mix of policies.

Commonly-used policies

FCFS
Round robin
STCF
Priority
Proportional share
EDF

Scheduling: Goals

What are good goals for a CPU scheduler?

1. Minimize average response time (latency).
2. Maximize throughput.
3. Fairness.
4. Ensure every thread makes progress.
5. …

But minimizing latency and maximizing throughput are often conflicting goals.

Load testing

With software systems, we actually do test them by breaking them.

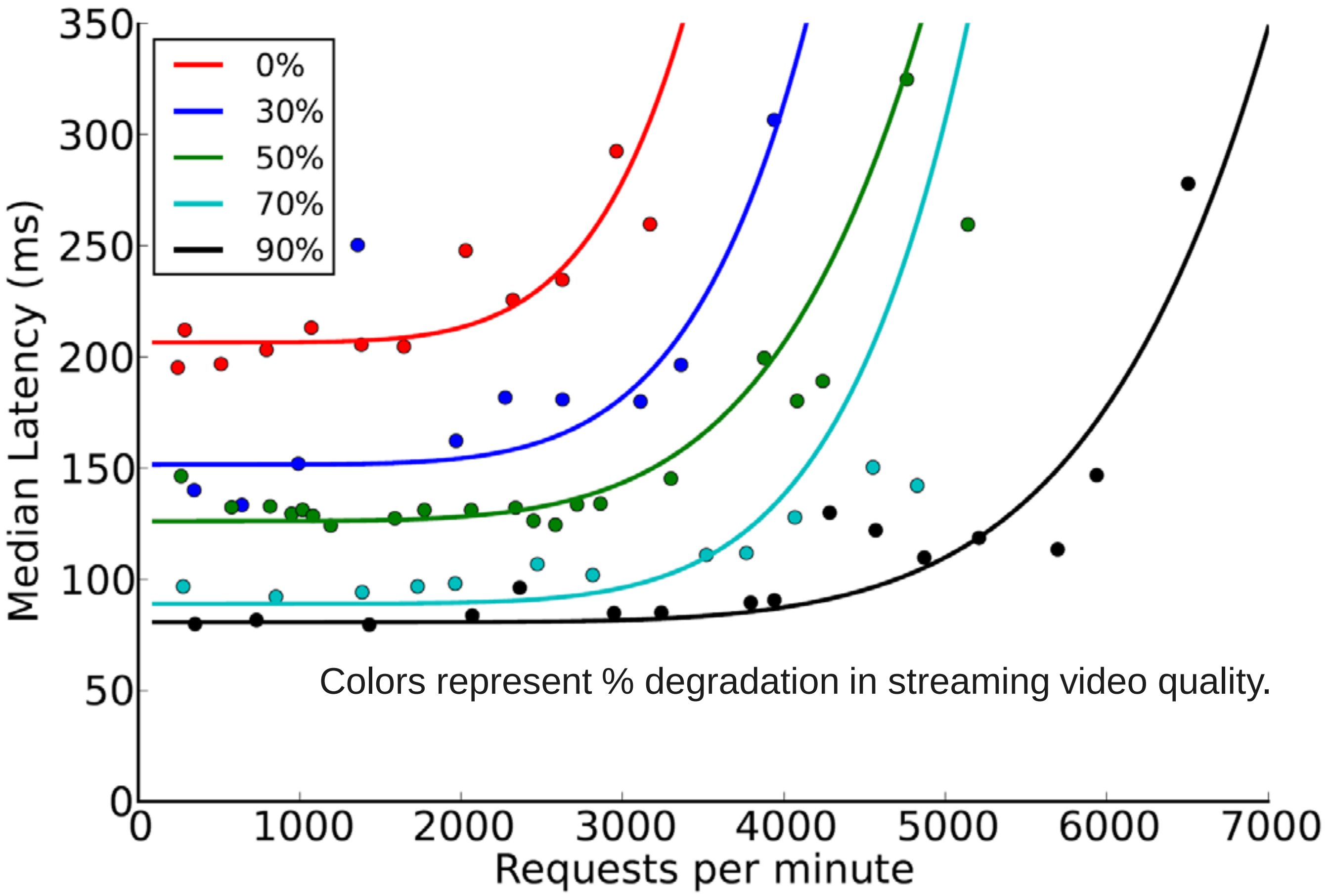
HOW DO THEY KNOW THE LOAD LIMIT ON BRIDGES, DAD?
LOAD LIMIT 10 TONS
THEY DRIVE BIGGER AND BIGGER TRUCKS OVER THE BRIDGE UNTIL IT BREAKS.
THEN THEY WEIGH THE LAST TRUCK AND REBUILD THE BRIDGE.
OH. I SHOULD'VE GUESSED.
DEAR, IF YOU DON'T KNOW THE ANSWER, JUST TELL HIM!
WATTERSON
11-26

Throughput-response curve for a Facebook video service

These are curves for a Facebook video service that was investigating reducing video quality when facing heavy load.

Latency is the response time.

Higher quality images take more time.



Colors represent % degradation in streaming video quality.

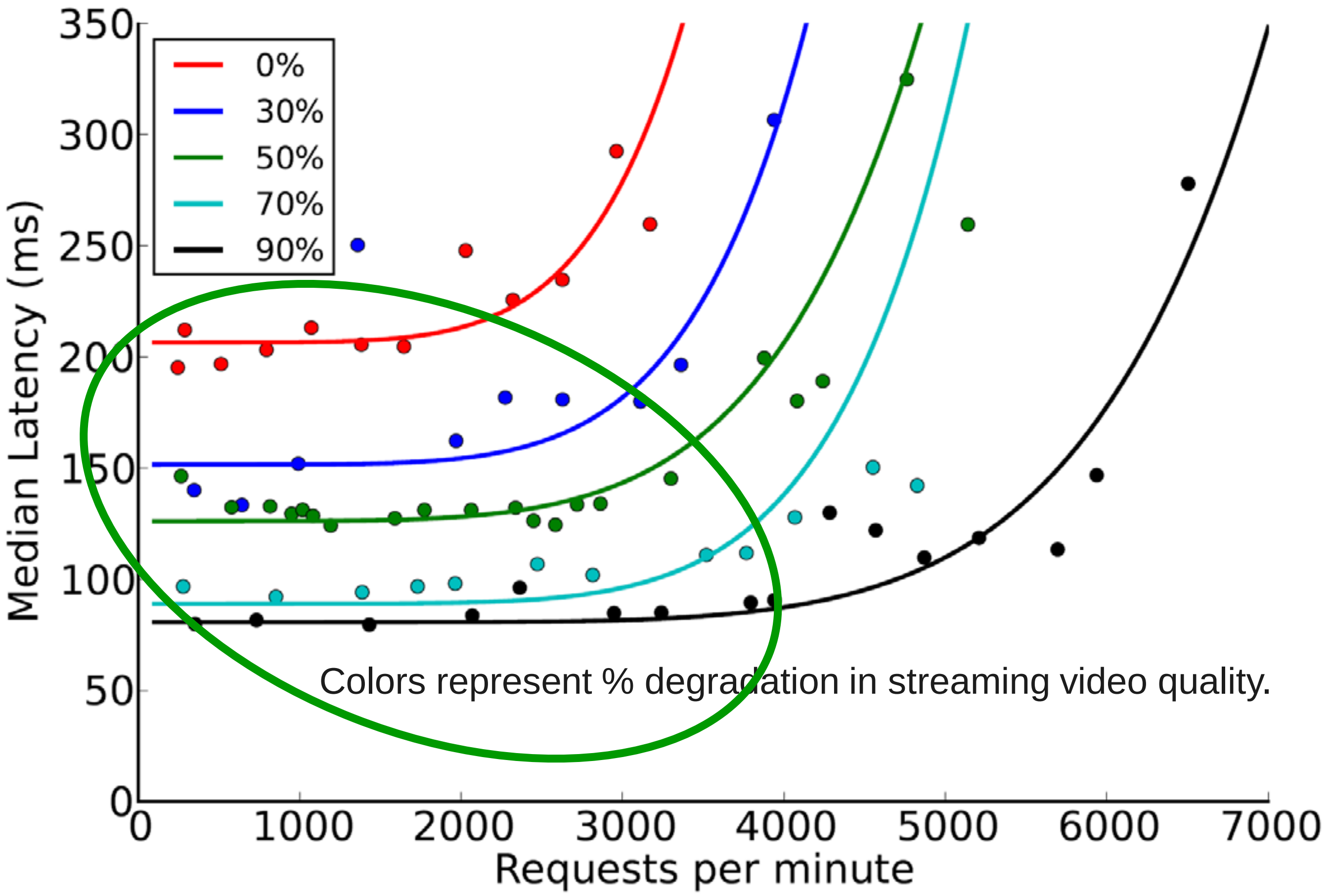
Collected from Facebook production service [Chow '16]

Throughput-response curve for a Facebook video service

At first, adding load has little effect on latency.

Requests per minute * latency < 1 minute.

The system is idle sometimes and can easily keep up.



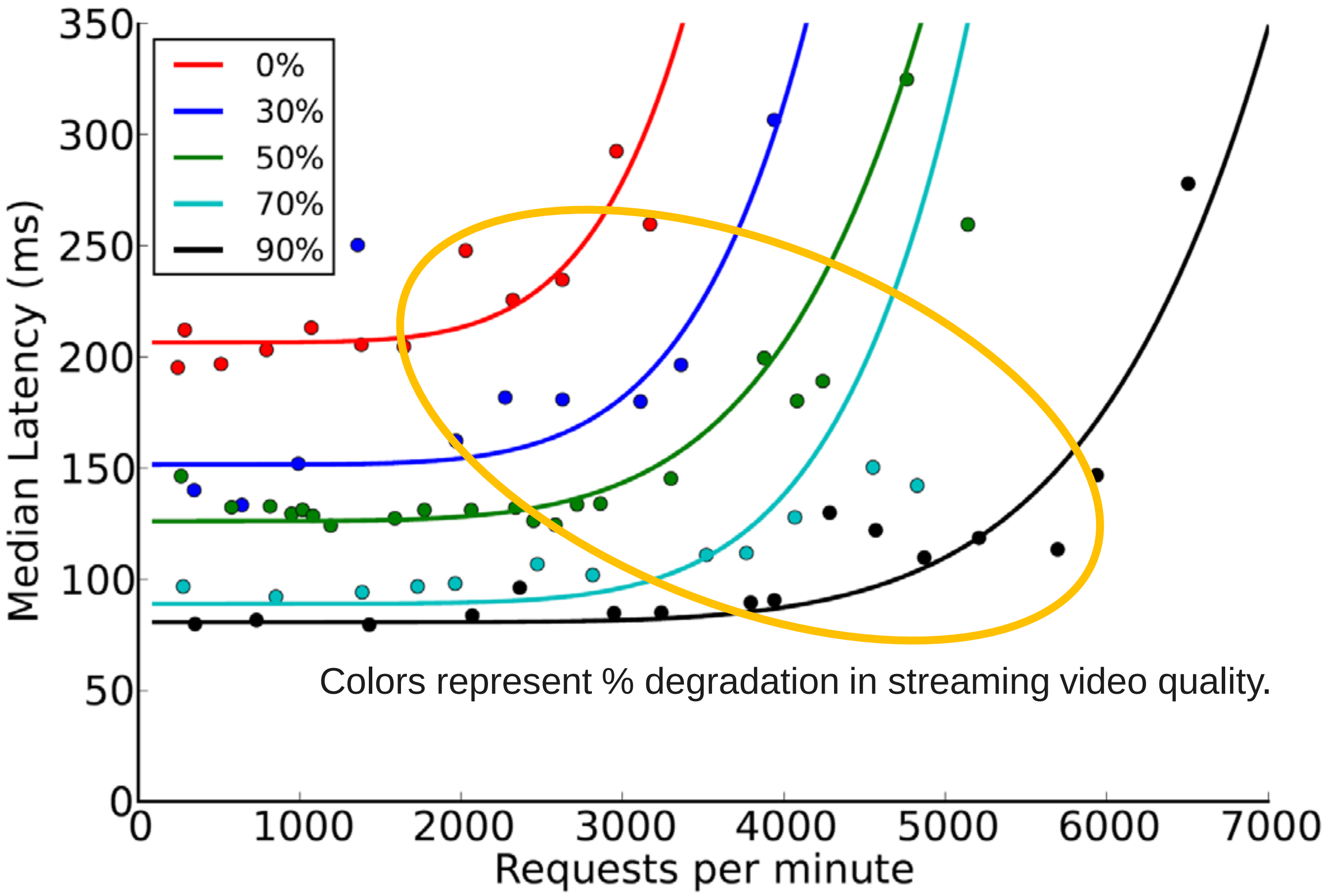
Collected from Facebook production service [Chow '16]

Throughput-response curve for a Facebook video service

But then it suddenly begins rising exponentially.

Requests per minute * latency approaching 1 minute.

The system is rarely idle and can barely keep up.



Collected from Facebook production service [Chow ‘16]

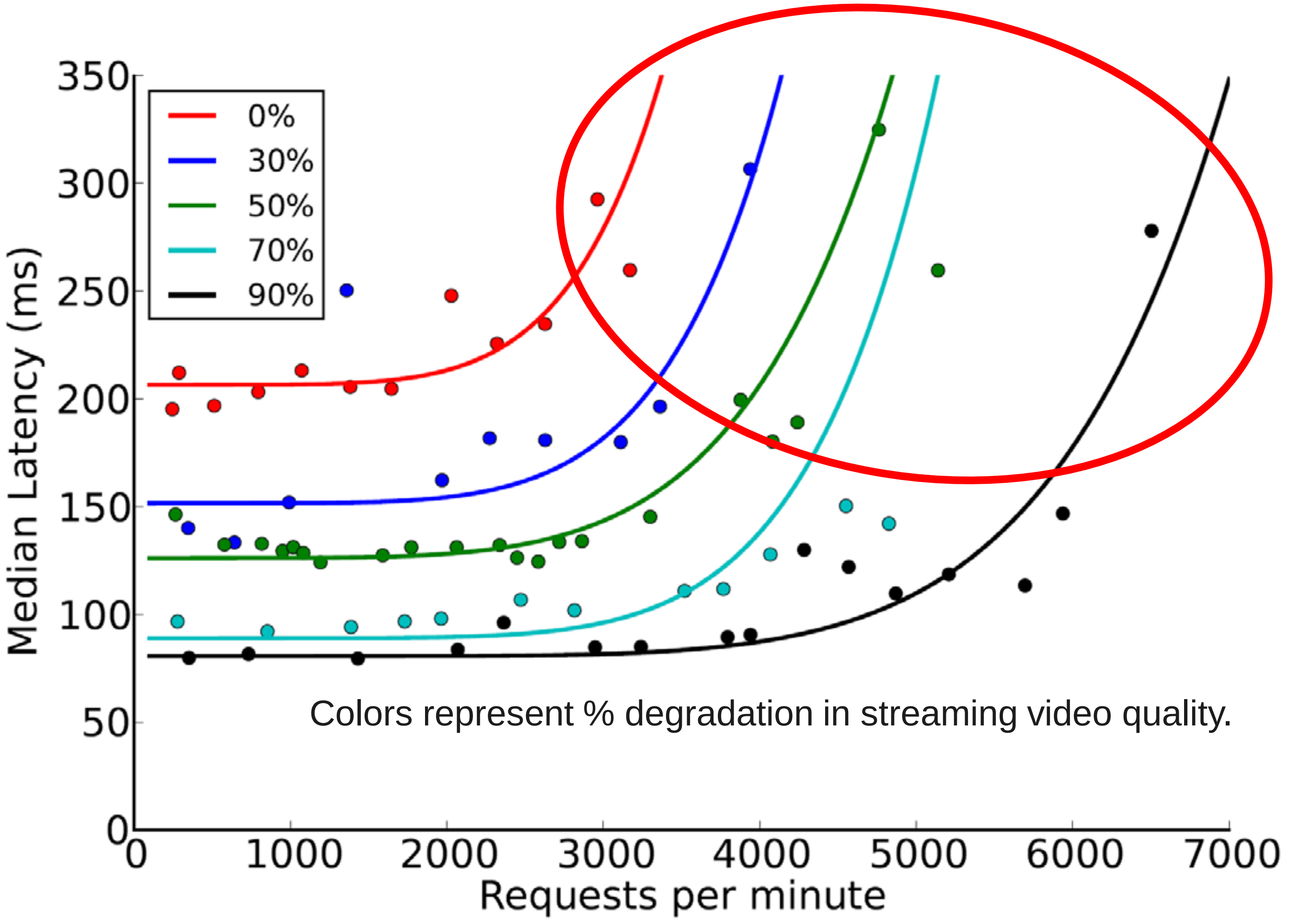
Throughput-response curve for a Facebook video service

Completely saturated. Latency has fallen off a cliff.

Requests per minute * latency > 1 minute.

With every minute, the system is falling further behind.

(This is why they will degrade under load to avoid saturation.)



Colors represent % degradation in streaming video quality.

Collected from Facebook production service [Chow ‘16]

Fairness

Share CPU among threads in equitable manner.

How to share between one big and one small job?

- Response time proportional to job size?
- Or equal time for each job?

Fairness often conflicts with response time.

Starvation = extremely unfair

Starvation can be outcome of synchronization.

- Example: Readers can starve writers.

Starvation can also be outcome of scheduling.

- Example policy: Always run highest-priority thread.
- If many high priority threads, low priority starves.
- Would like to ensure all threads eventually make progress if they can.

First-come, first-served (FCFS)

FIFO ordering among jobs.

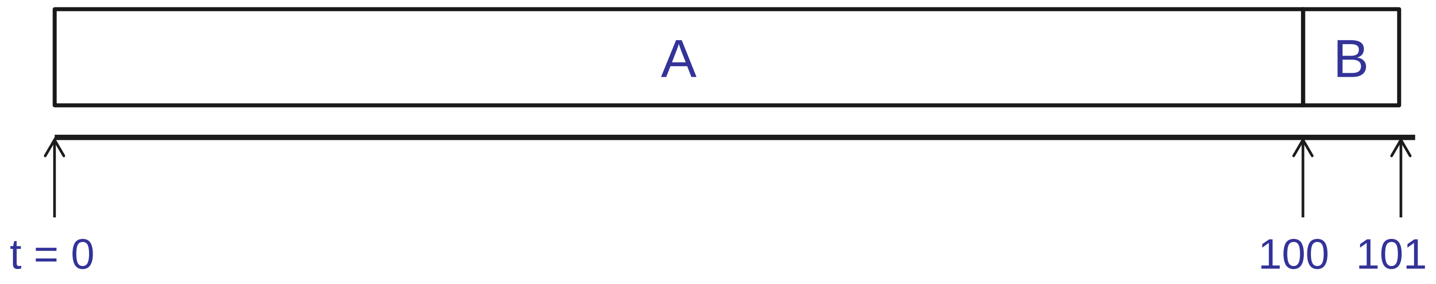
No preemption (no timer interrupts).

- Thread runs until it calls yield() or blocks.

FCFS Example

Job A: Arrives at t=0, takes 100 seconds

Job B: Arrives at t=0+, takes 1 second



A’s response time = 100

B’s response time = 101

Average response time = 100.5

FCFS Summary

Pros:

- Simple to implement

Cons:

- Short jobs can be stuck behind long ones
- Bad for interactive workloads

Round Robin

Improve average response time for short jobs

Add preemptions (via timer interrupts)

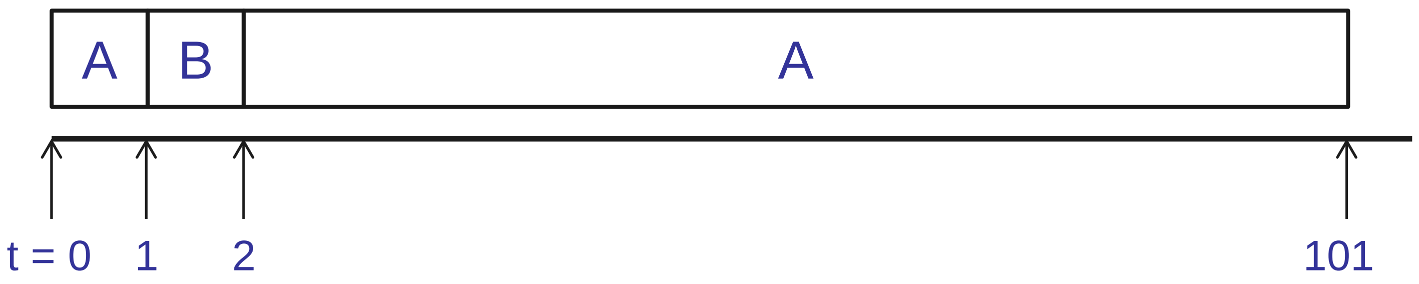
- Fixed time slice (time quantum)
- Preempt if still running when time slice is over

Round Robin Example

Job A: Arrives at t=0, takes 100 seconds

Job B: Arrives at t=0+, takes 1 second

Assume the quantum is 1 second



A’s response time = 101

B’s response time = 2

Average response time = 51.5

Choosing a time slice

What’s the problem with a big time slice?

Degenerates to FCFS and poor interactivity.

What’s the problem with a small time slice?

More context switching overhead and lower throughput.

OS typically compromises, e.g., 1 ms or 10 ms.

Round Robin Summary

Pros

- Still pretty simple.
- Good for interactive computing.

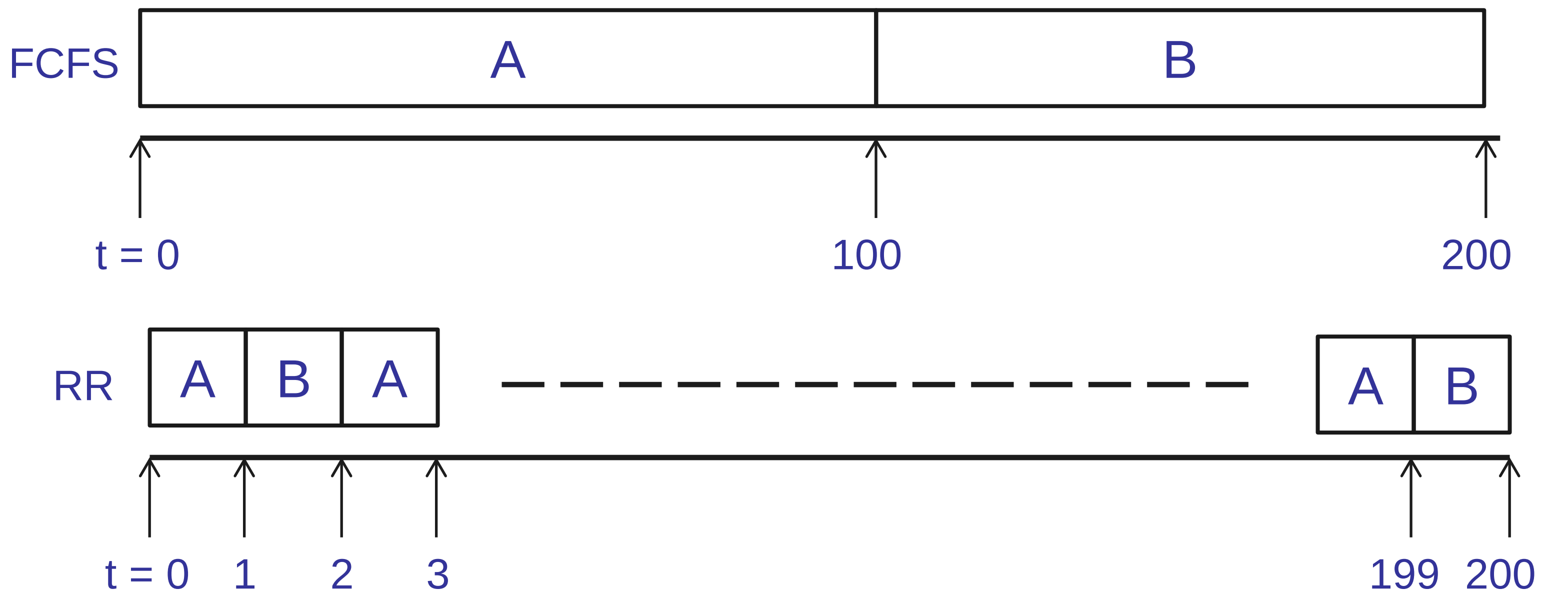
Cons

- More context-switching overhead.

Does RR always reduce avg response time vs. FCFS?

Round Robin vs. FCFS

Jobs A and B arrive at t = 0 and 0+, both take 100 secs



Average response time with FCFS = 150

Average response time with RR = 199.5

Which is more fair? RR or FCFS?

STCF

Shortest time to completion first.

Run job with least work to do.

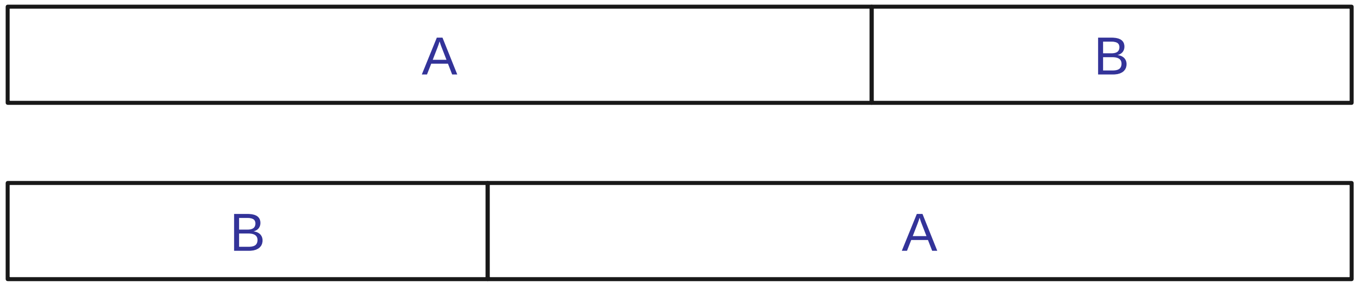
- Preempt current job if shorter job arrives.
- Job size is time to next blocking operation.

Finish short jobs first.

- Improves response time of short jobs (by a lot).
- Hurts response time of long jobs (by a little).

STCF gives optimal average response time.

Analysis of STCF



Consider 2 jobs: A longer than B

Average response time (2A + B) / 2 vs. (A + 2B) / 2

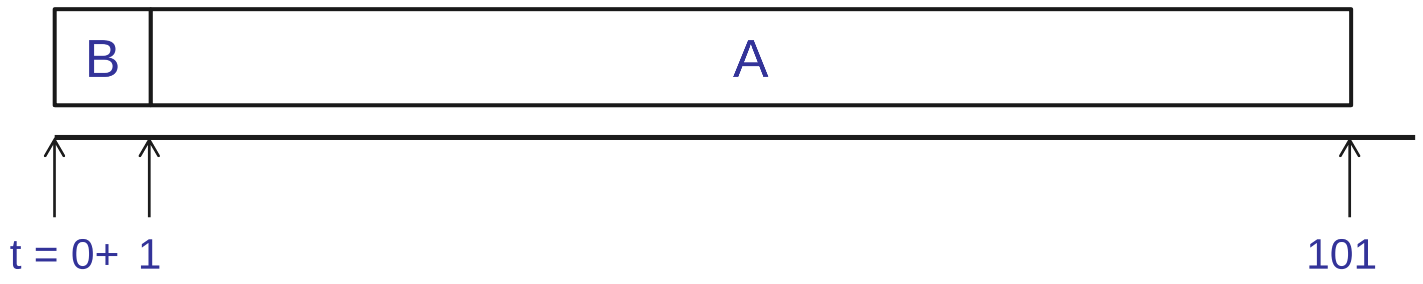
B < A, so running B first has a smaller average response time.

Keep the list sorted by estimated time to completion, pick the shortest to minimize response time.

STCF Example

Job A: Arrives at t=0, takes 100 seconds

Job B: Arrives at t=0+, takes 1 second



A’s response time = 101

B’s response time = 1

Average response time = 51 (RR was 51.5)

STCF

Pro

- Optimal average response time.

Cons

- Potential starvation for long jobs (really unfair!)
- Needs knowledge of future.

How could you estimate the time a job will run for?

Predicting job run times

Ask the job or the user?

- Strong incentive to lie ("will just take a minute")

Use past to predict future

Can assume heavy-tailed distribution

- If already run for n seconds, likely to run for n more

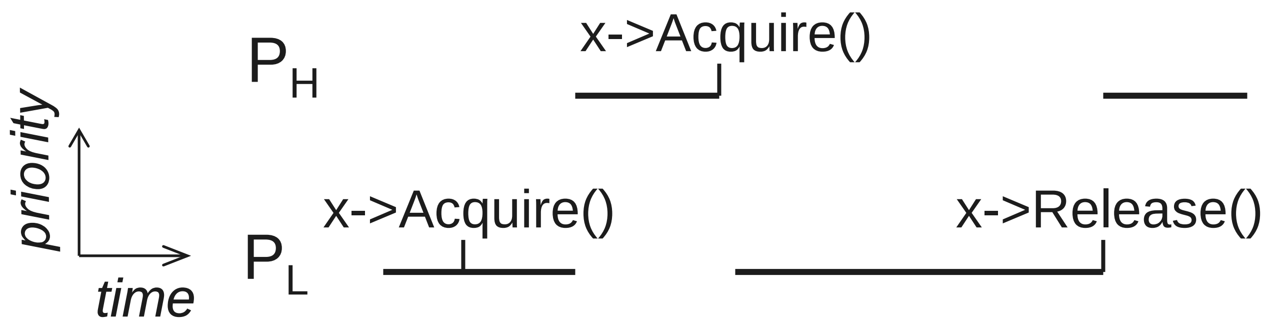
OS schedulers often identify interactive apps and boost their priority

Priority

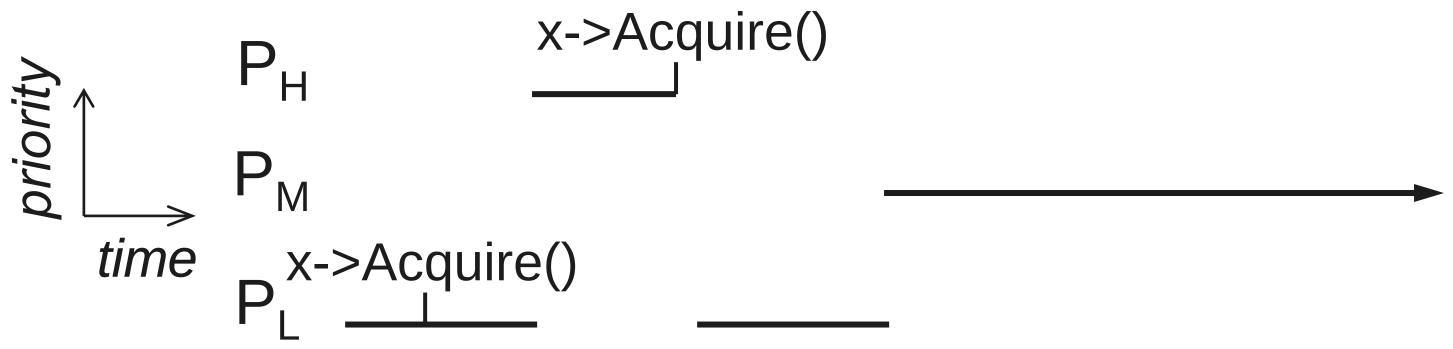
- Priority
  - Assign external priority to each job
  - Run high-priority jobs before low-priority ones
  - Use, e.g., round-robin for jobs of equal priority
  - Prone to starvation
- Methods for preventing starvation?
  - If job has not run for time t, boost priority
  - Handle priority inversion (lock held by low-priority)

Priority Inversion

Normal pattern of sharing resources.



Priority inversion. The middle priority job runs indefinitely.



Priority inversion

Strategies:

1. Analyze the dependencies. If a high priority thread is waiting on a resource held by a lower priority thread, boost the lower priority thread.

2. Windows NT: Randomly boost threads.

Hard real-time scheduling

Jobs have to complete before deadline.

- Demand / deadline known in advance.
- Examples: Vehicle control, aviation, etc.

Earliest-deadline first (EDF).

- Always run jobs whose deadline is soonest.
- Preempt if newly arriving job has earlier deadline.
- *Always succeeds if schedule is feasible.*
- But, may be very poor if schedule is infeasible.

CPU scheduling

How would you choose?

Suppose you were scheduling the checkout in a grocery store? Would you have a queue per checkout or a single queue?

Real schedulers often employ heuristics and complex tuning.

Commonly-used policies

FCFS
Round robin
STCF
Priority
Proportional share
EDF

Agenda

1. CPU scheduling.
2. Project 3 due July 27.

Project 3

Process view:

1. Every process has an address space starting from `VM_ARENA_BASEADDR` of size `VM_ARENA_SIZE`.
2. When a process starts, the entire address space is invalid.
3. Process calls `vm_map` to make pages valid.
4. Pages becomes invalid when a process ends.

Pager view:

1. One process runs at a time.
2. Sets up page table that the MMU uses for translation.
3. Handles `vm_create`, `vm_map`, and `vm_fault`.

Project 3

Swap-backed pages:

1. Global swap file shared by all processes.
2. Pager controls where pages are stored in the swap file.
3. Individual pages are private to a process.

File-mapped pages:

1. Process specifies the file and offset.
2. Can be shared across processes.

Project 3: App vs. OS

Protection

- All pages can be read from and written to.
- Using R/W bits to track reference, dirty, etc.

Sharing

- File-backed pages.
- Copy-on-write.

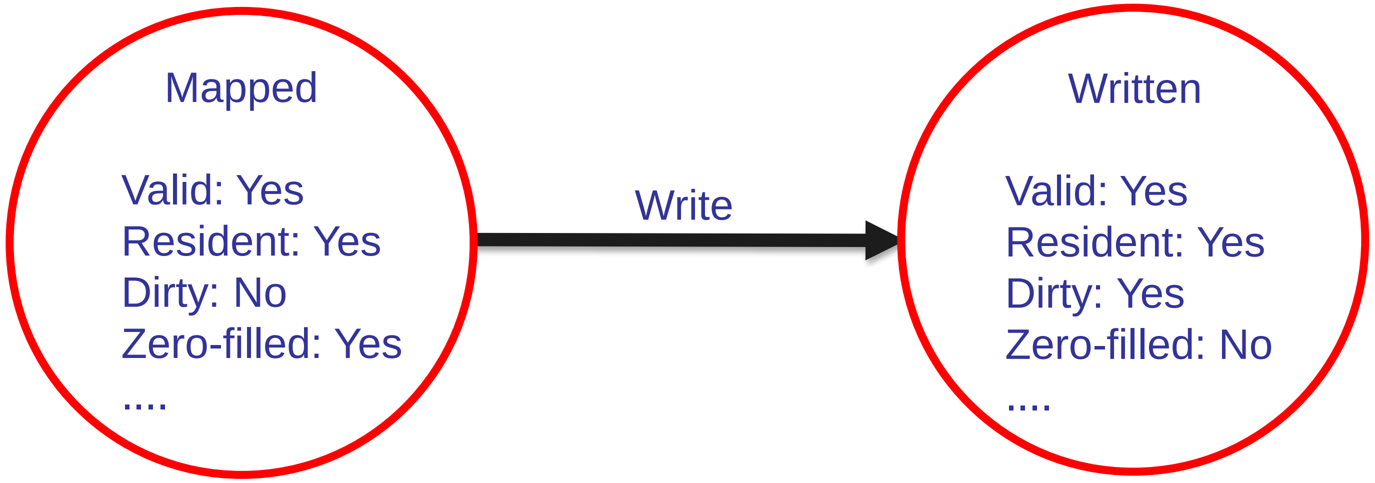
Project 3

1. Do the project incrementally.
2. Swap-backed pages only without fork.
3. Then add support for fork and file-backed pages one after the other.
4. Pro Tip: Start with state diagrams for swap-backed, file-backed pages.

Project 3: State Diagram

For each unique state, consider:

1. Transitions? Read, write, clock, copy, ...
2. Attributes? Valid, resident, dirty, ...
3. Protections? Enable read, enable write?


Mapped
Valid: Yes
Resident: Yes
Dirty: No
Zero-filled: Yes
....
Write
Written
Valid: Yes
Resident: Yes
Dirty: Yes
Zero-filled: No
....