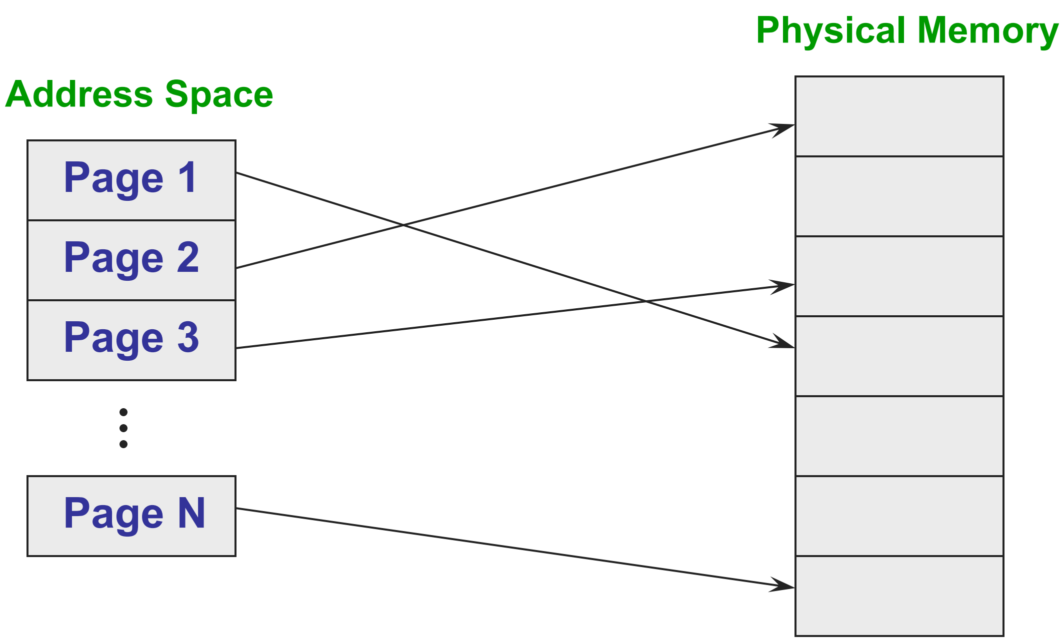
EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

Recap: Paging

- Both address spaces and physical memory broken up into fixed size pages



Recap: Paging

- Virtual address to physical address translation using page table

Virtual page #	Physical page #	Protection
0	105	RX
1	15	R
2	283	RW
3	invalid	
…	invalid	
1048575	invalid	

- Can manipulate protection bits to maintain other bits (resident, referenced, dirty) in OS

Recap: Page Replacement

- Not all virtual pages can be in physical mem.
- Steady state: Evict a page to make another page resident
  - Use reference bit to identify pages to evict
  - Use dirty bit to identify need for write-back

Recap: Process creation

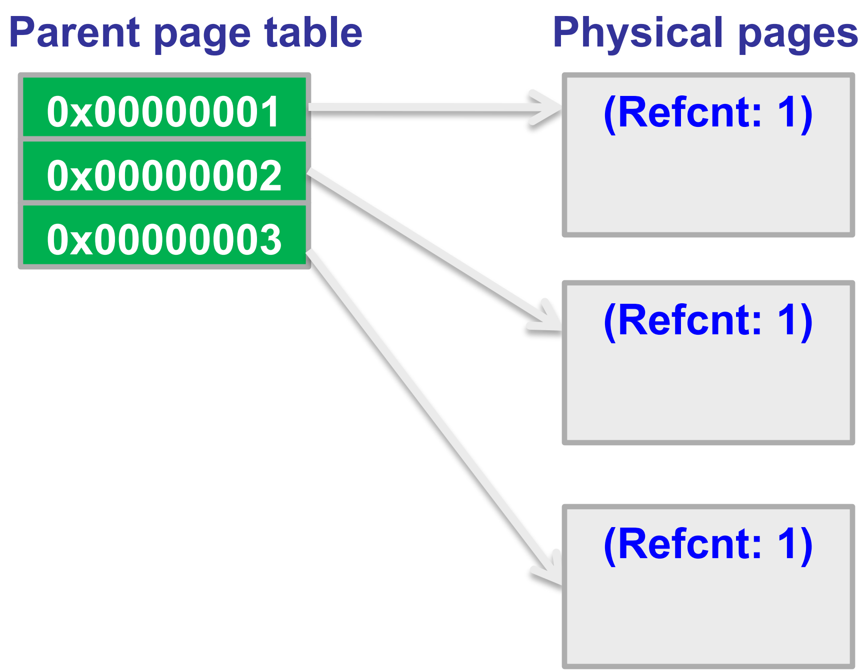
- System calls to start a process:
  1. Fork() creates a copy of current process
  2. Exec(program, args) replaces current address space with specified program

```
If (fork() == 0) {
    exec ();        /* child */
} else {
    /* parent */
}
```

Avoiding work on fork

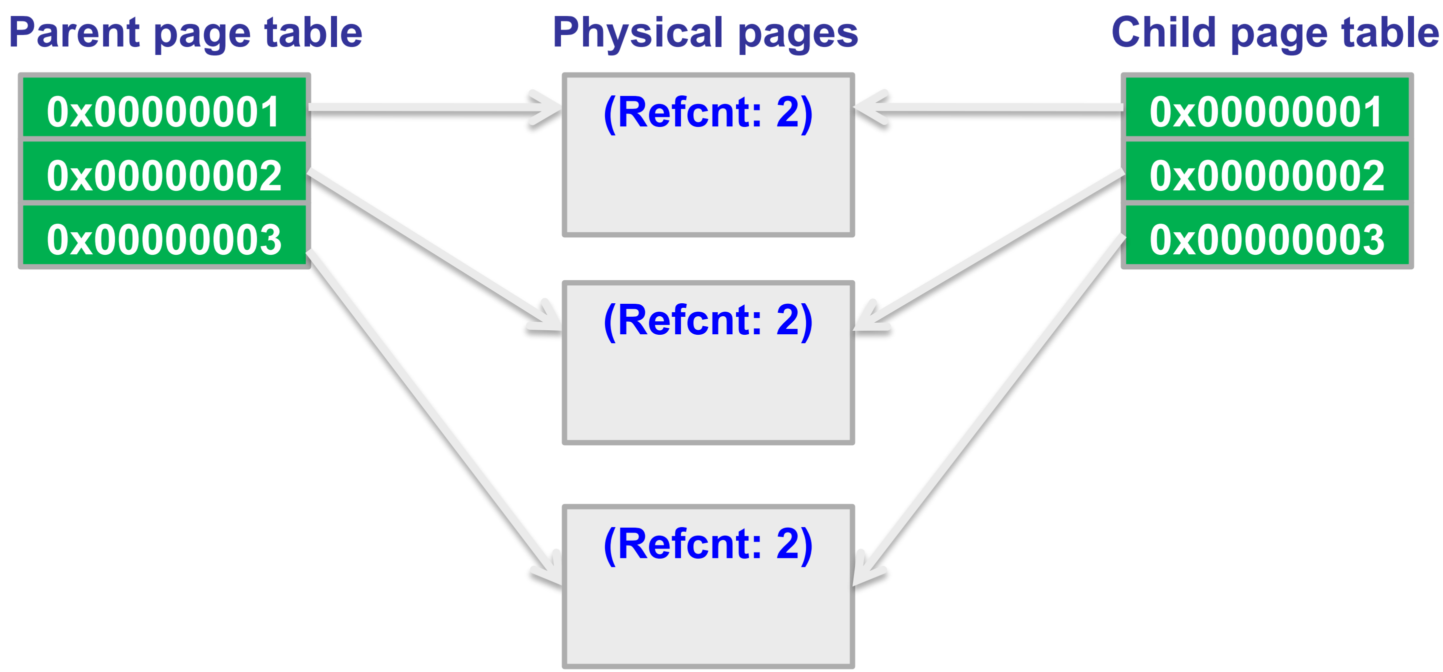
- Copying entire address space is expensive
- Instead, Unix uses copy-on-write
  - Maintain reference count for each physical page
  - On fork(), copy only the page table of parent
    - » Increment reference count by one
  - On store by parent or child to page with refcnt > 1:
    - » Make a copy of the page with refcnt of one
    - » Modify PTE of modifier to point to new page
    - » Decrement reference count of old page

Copy-on-write: Example



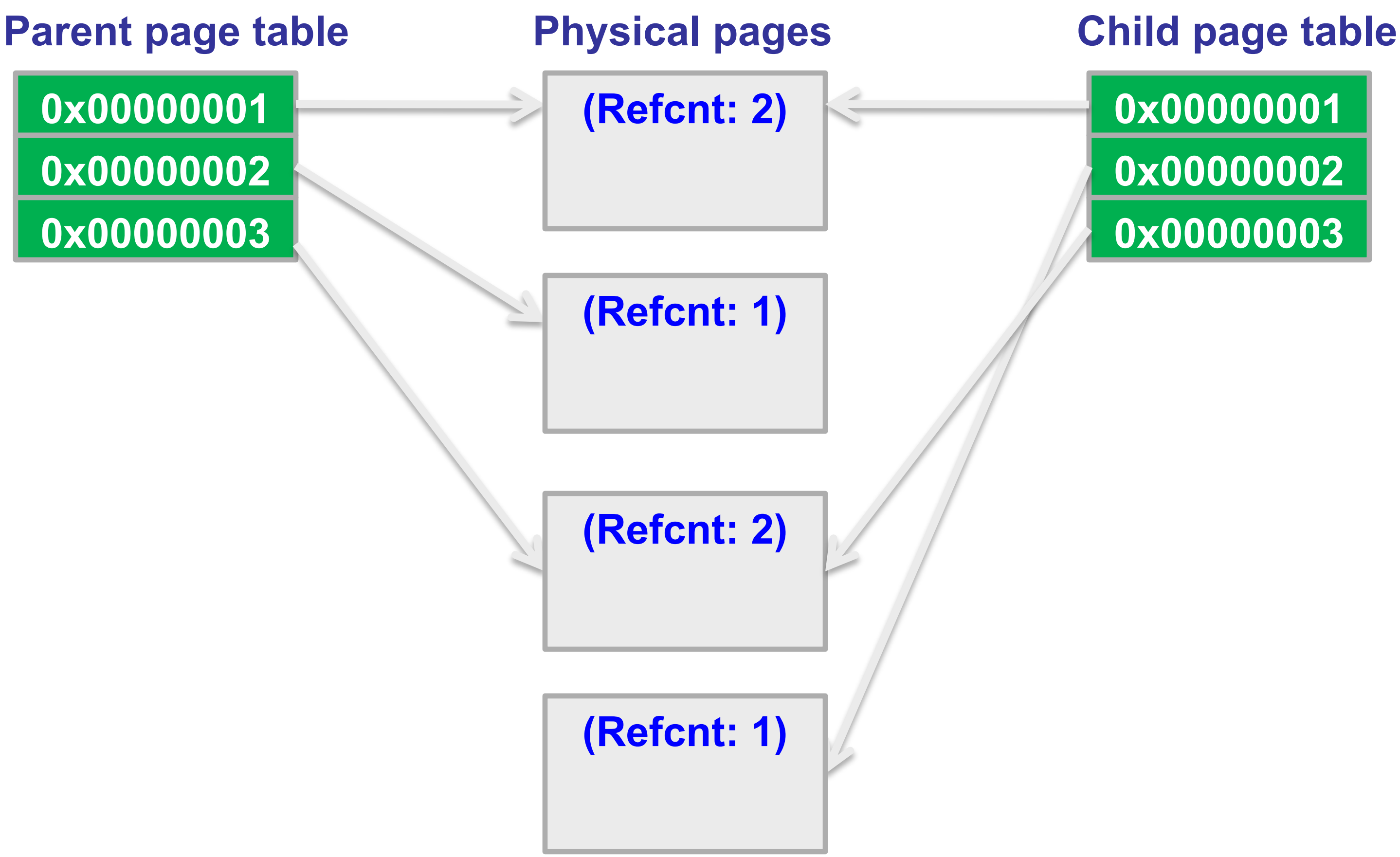
Parent about to fork()

Copy-on-write: Example



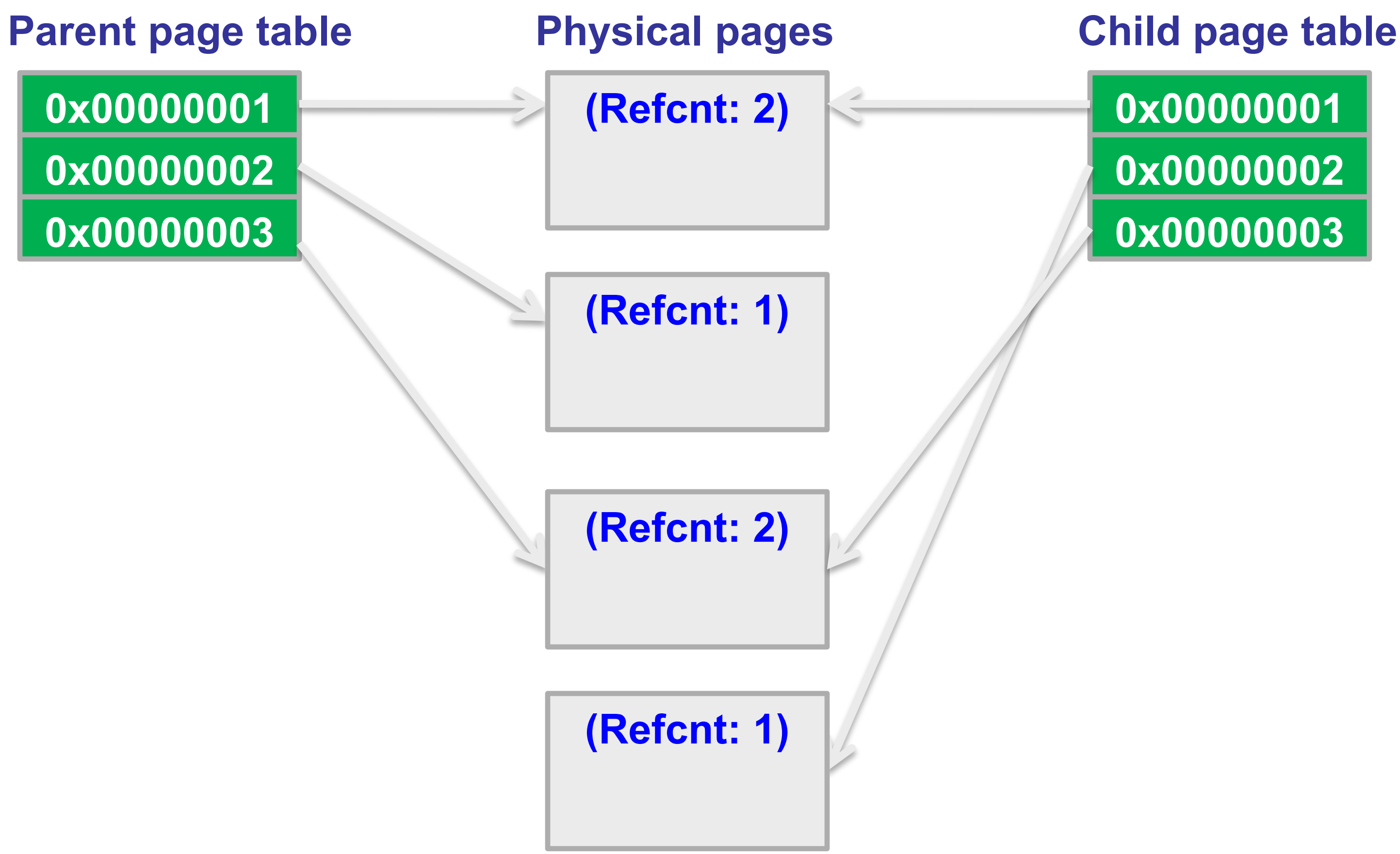
Copy-on-write of parent address space

Copy-on-write: Example



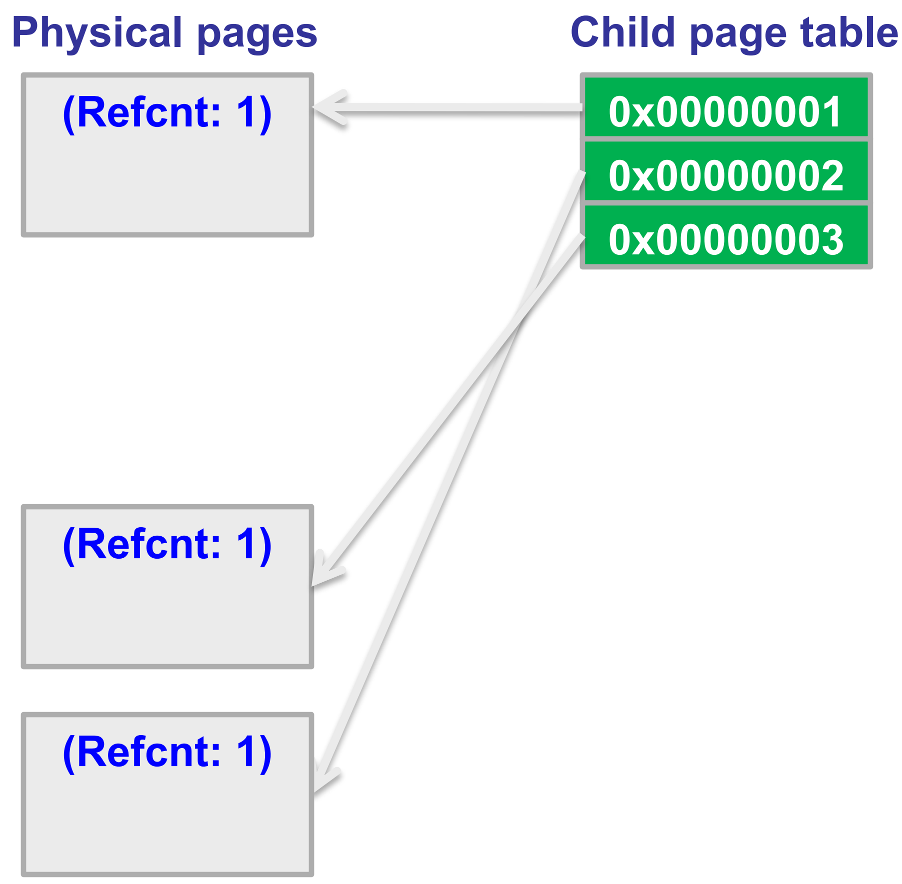
Child modifies 2nd virtual page

Copy-on-write: Example



Parent modifies 2nd virtual page

Copy-on-write: Example



Parent exits

Making exec() faster

- exec() initializes code in the address space
  - Naive solution: read file, copy into memory
  - Can we do better?
- Observation: most code never accessed
  - Load code on-demand
  - Similar to loading memory paged to disk
  - Memory-mapped files (file-backed pages in P3)

File-backed vs. swap-backed

- Swap-backed pages
  - Block on disk chosen by pager
  - A process's writes to a page visible only to that process
  - Modifications lost after process exit

- File-backed pages
  - Block on disk chosen by app
  - Any process's write to a page visible to other processes that map the same block
  - Modifications persist across process lifetimes

Processes sharing memory

- How to divide phys. memory among processes?
  - Goals: fairness versus efficiency

- Global replacement
  - Can evict pages from faulting process or any other
- Local replacement
  - Can evict pages only from faulting process
  - Must determine how many frames each process gets

- Pros and cons?

Thrashing

- What happens if many large processes all actively use their entire address space?

- Performance degrades rapidly as miss rate goes up
  - Avg access time = hit rate * hit time + miss rate * miss time
  - E.g., hit time = .0001 ms; miss time = 10 ms
    - » Average access time (100% hit rate) = .0001 ms
    - » Average access time (1% miss rate) = .100099 ms
    - » Average access time (10% miss rate) = 1.00090 ms

Solutions to Thrashing

- Buy more DRAM
  - Very common solution in cloud servers
  - Price per GB fallen by 4x since 2009

- Run fewer processes for longer time slices
  - Reduces page faults
  - But, poor interactivity due to long time slices

Working set

- Thrashing depends on portion of address space actively used by each process
  - What do we mean by "actively using"?
- Working set = all pages used in last T seconds
  - Larger working set ➔ need more memory
- Sum of all working sets should fit in memory
  - Only run subset of processes that fit in memory
- How to measure size of working set?
  - Periodic sweep of clock hand in LRU clock

Project 3

- Hope you have a state machine for swap-backed pages by now???

- Things to consider:
  - Transitions?
  - Properties that capture state of a page?
  - Protection bits?

- Don’t translate state machine into if-else cases!
- Think ahead in designing data structures

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

CPU scheduling

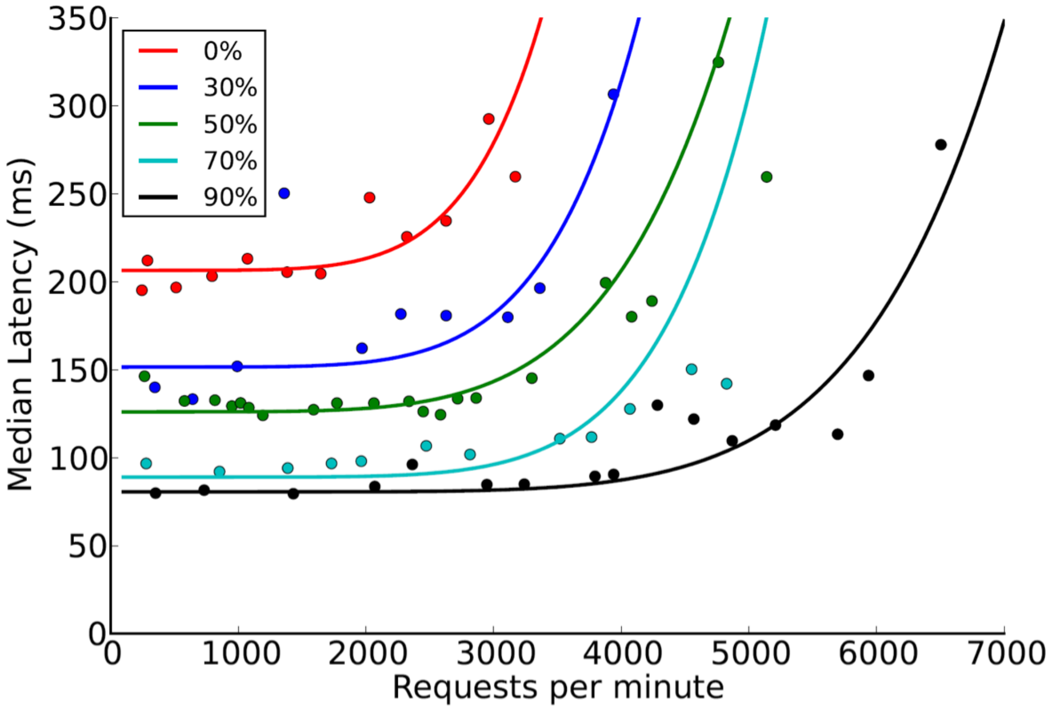
- If >1 thread is ready, choose which to run

- Many possible scheduling policies
  - Goal today is to explore fundamental ones
  - Real schedulers often a complex mix of policies

Scheduling: Goals

- What are good goals for a CPU scheduler?
  - Minimize average response time
  - Maximize throughput
  - Fairness
- "Minimize latency" at odds with "maximize tput"

Throughput-response curves



- Collected from Facebook production service [Chow '16]
  - Each colored line: throughput vs. latency at different quality
  - Left of graph – adding load → little effect on response time
  - Right of graph – adding load → exponential increase in latency

Load testing

Fairness

- Share CPU among threads in equitable manner
- How to share between 1 big and 1 small job?
  - Response time proportional to job size?
  - Or equal time for each job?
- Fairness often conflicts with response time

Starvation = extremely unfair

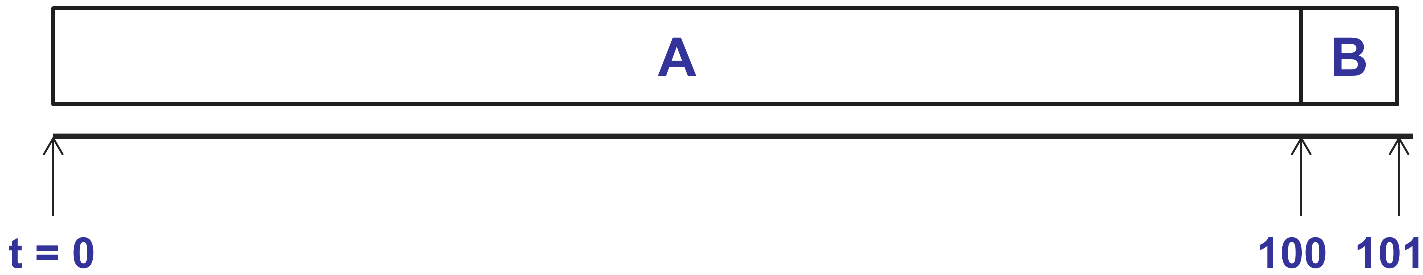
- Starvation can be outcome of synchronization
  - Example: Readers can starve writers
- Starvation can also be outcome of scheduling
  - Example: always run highest-priority thread
  - If many high priority threads, low priority starves

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

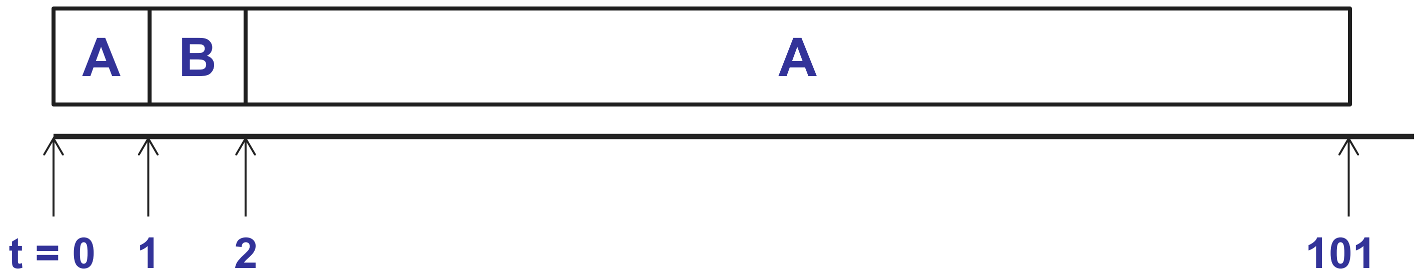
- Pros:
  - Simple to implement
- Cons:
  - Short jobs can be stuck behind long ones
  - Bad for interactive workloads

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



- A's response time = 101
- B's response time = 2
- Average response time = 51.5

Choosing a time slice

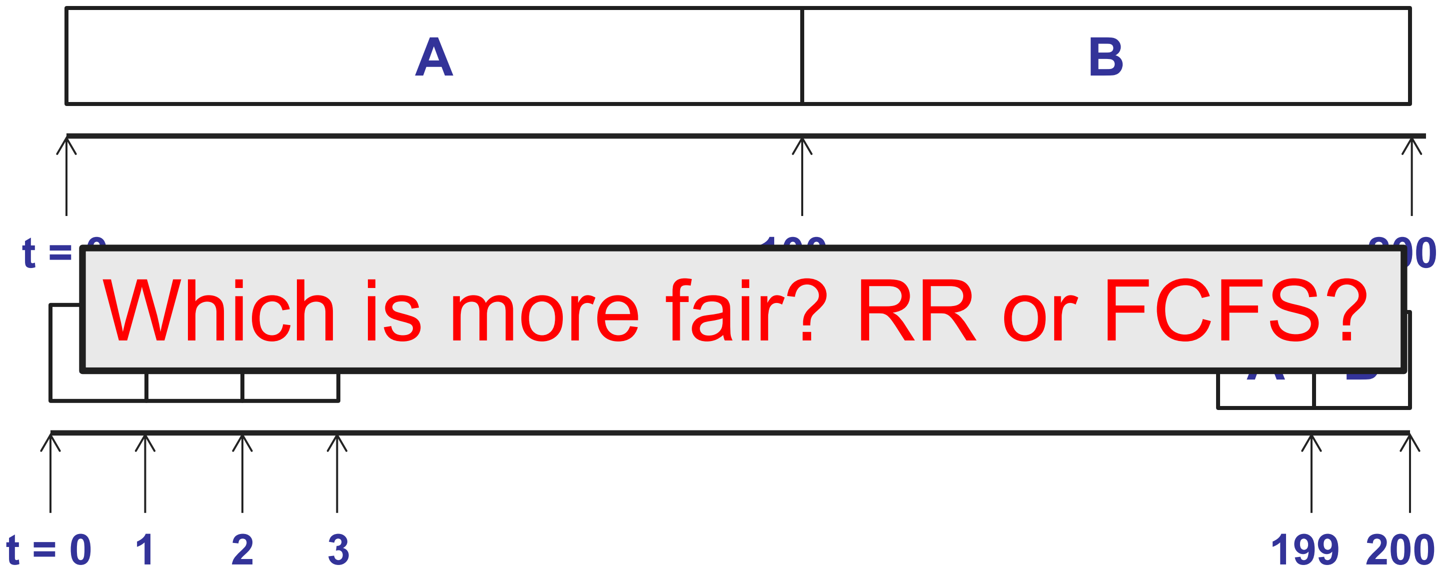
- What's the problem with a big time slice?
  - Degenerates to FCFS (poor interactivity)
- What's the problem with a small time slice?
  - More context switching overhead (low throughput)
- OS typically compromises: e.g., 1ms or 10ms

Round Robin Summary

- Pros:
  - Still pretty simple
  - Good for interactive computing
- Cons?
  - More context-switching overhead
- Comparison: Does RR always reduce average response time vs. FCFS?

Round Robin vs. FCFS

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



- Average response time with FCFS = 150
- Average response time with RR = 199.5

STCF

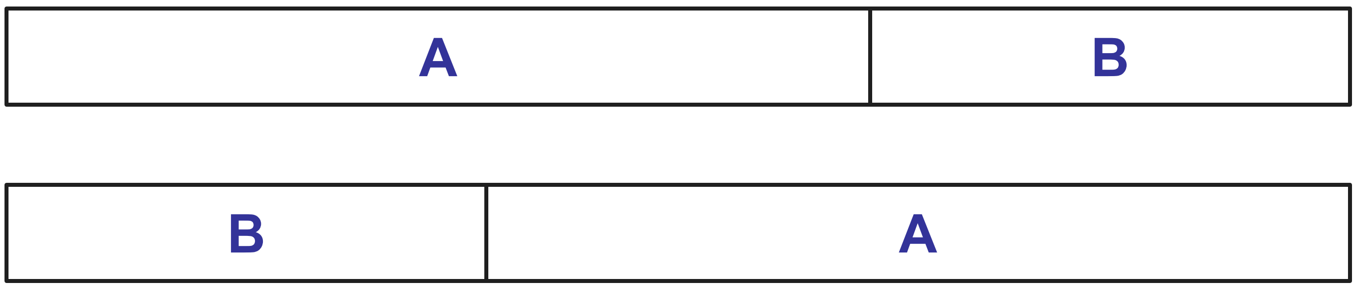
- Shortest time to completion first

- 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

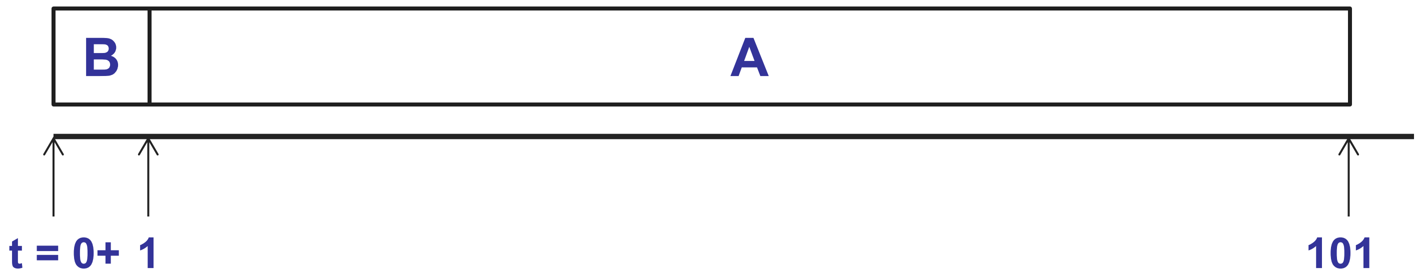
Analysis of STCF



- Consider 2 jobs: A longer than B
- Average response time (2A+B)/2 vs. (A+2B)/2
- B < A, so 2nd has smaller avg. response time
- Apply iteratively (e.g., bubble sort) to minimize

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

STCF

- Pro:
  - Optimal average response time
- Cons?
  - Potential starvation for long jobs (really unfair!)
  - Needs knowledge of future
- How to 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