Priority Scheduling

- Assign a number (priority) to each job and schedule jobs in priority order
- Can implement any scheduling policy
  - Reduces to SRTF when using as priority τ_n (the estimate of the execution time)
- To avoid starvation
  - change job's priority with time (aging)
  - select jobs randomly, weighted by priority

"Completely Fair Scheduler" (CFS)

Spent Execution Time

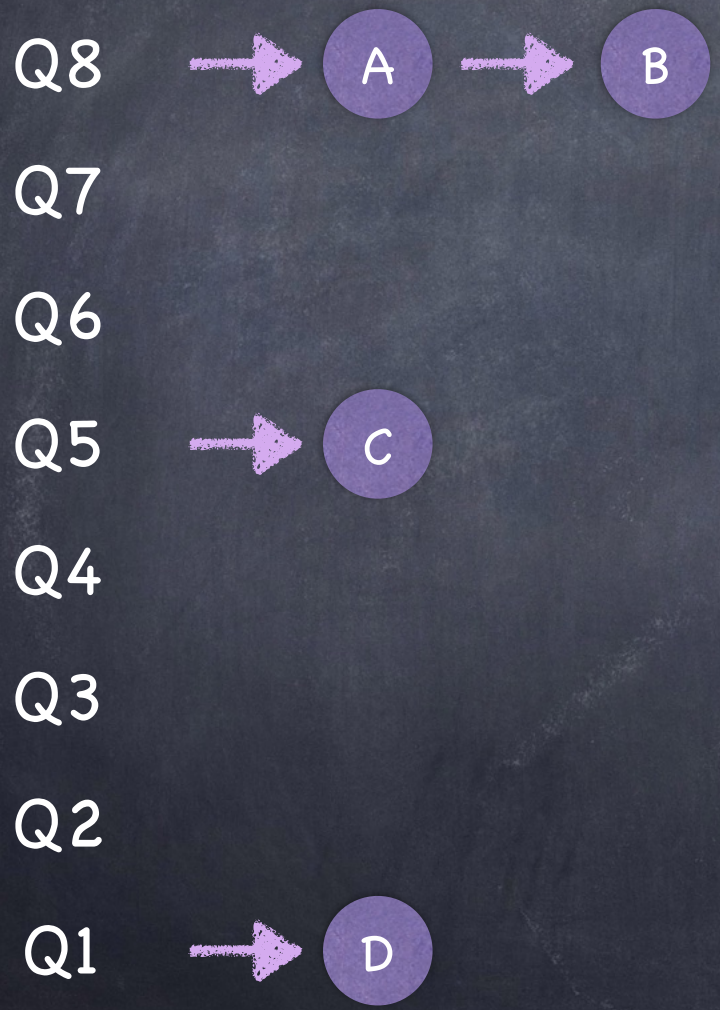
- SET: time process has been executing
- Scheduler selects process with lowest SET
- Given a quantum Δ and N processes on ready queue
  - process runs for Δ/N time (there is a minimum value)
- If it uses it up, reinserted into queue with SET += Δ/N
  - for efficiency, queue implemented as a red/black tree
- For a process p that is new or sleeps and wakes up
  - SETp = max (SETp, min{SET of ready processes})
- To account for priority, SET grows slower for higher priority processes

Used by most versions of Linux!

Multi-level Feedback Queue (MFQ)

- Scheduler learns characteristics of the jobs it is managing
  - Uses the past to predict the future
- Favors jobs that used little CPU...
  - ...but can adapt when jobs change their pattern of CPU usage

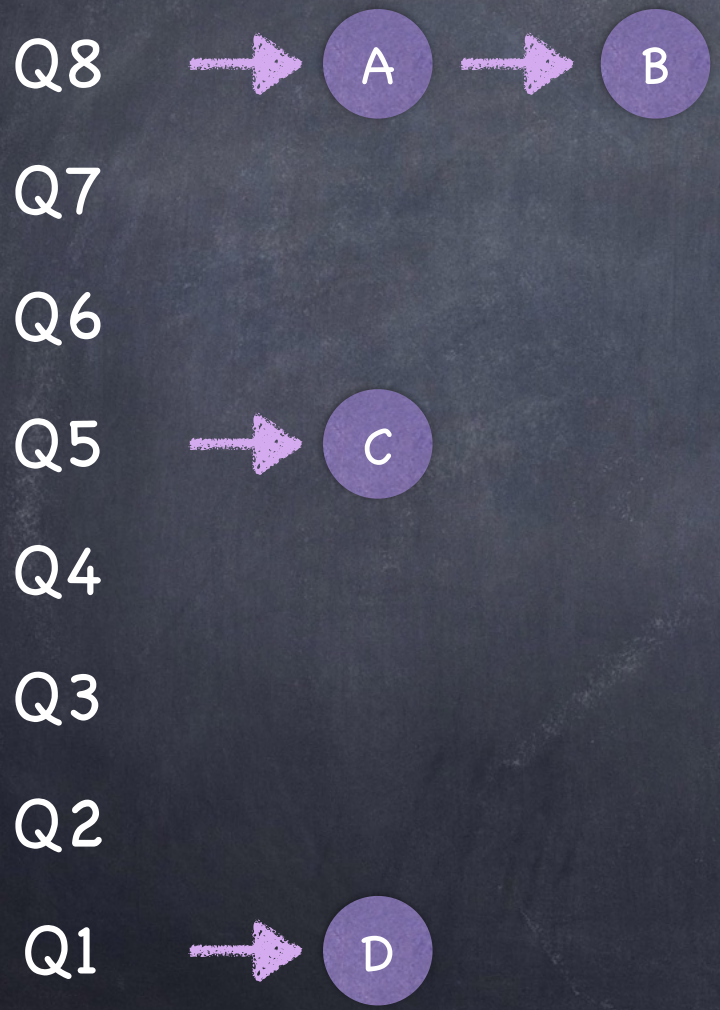
The Basic Structure



- Queues correspond to different priority levels
  - higher is better
- Scheduler runs job in queue i if no other job in higher queues
- Each queue runs Round Robin
- Parameter:
  - how many queues?

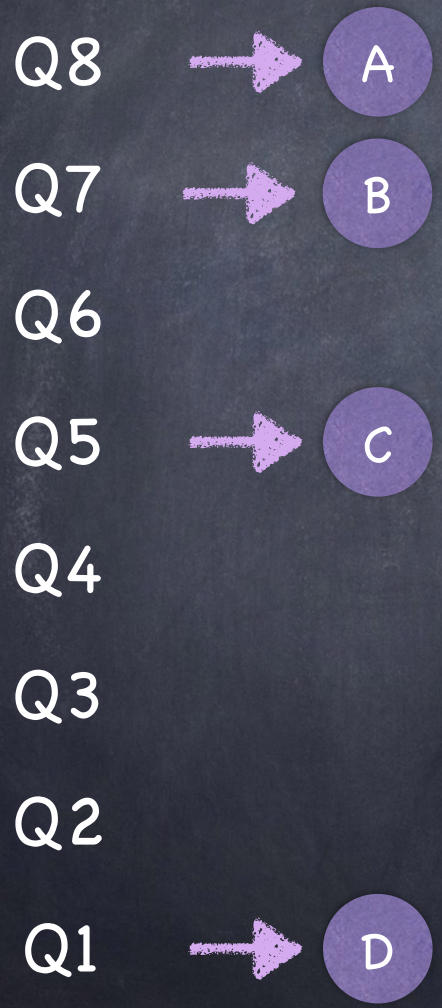
How are jobs assigned to a queue?

Moving down



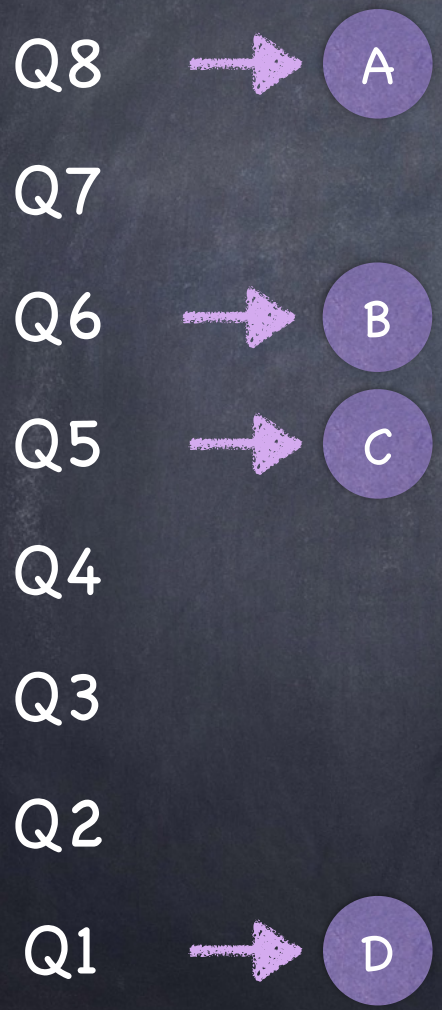
- Job starts at the top level
- If it uses full quantum before giving up CPU, moves down

Moving down



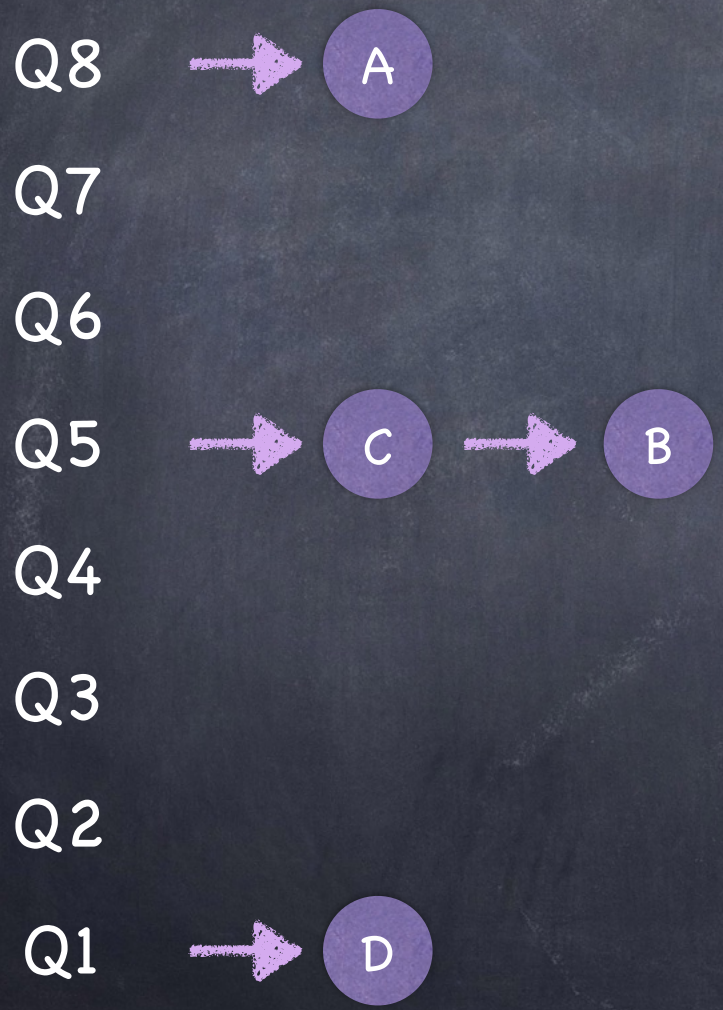
- Job starts at the top level
- If it uses full quantum before giving up CPU, moves down

Moving down



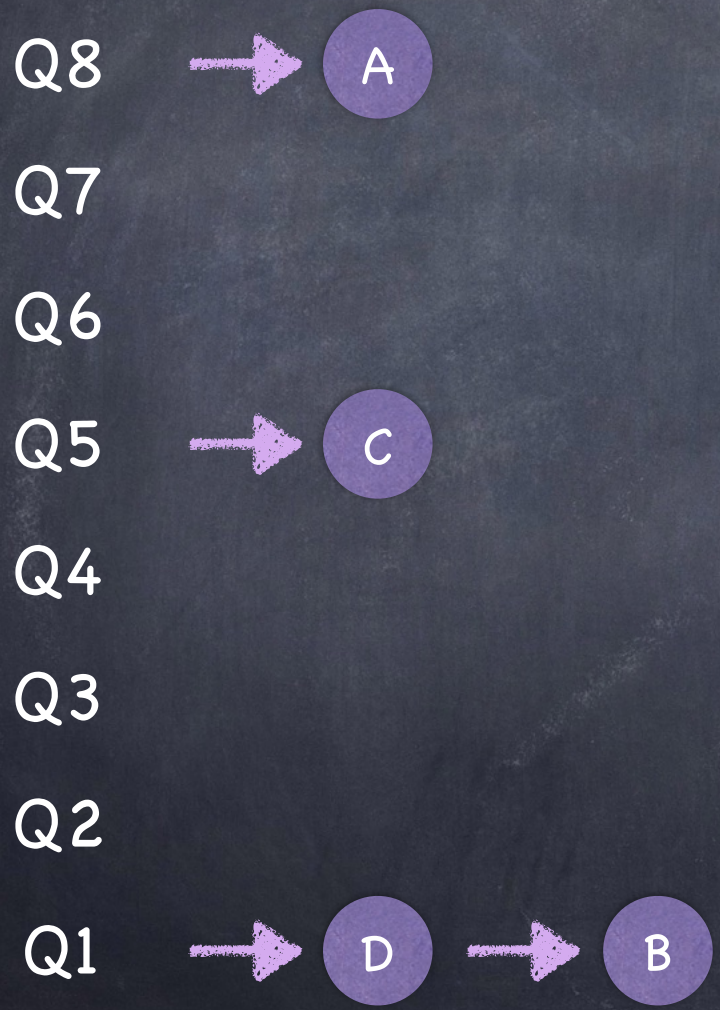
- Job starts at the top level
- If it uses full quantum before giving up CPU, moves down

Moving down



- Job starts at the top level
- If it uses full quantum before giving up CPU, moves down
- Otherwise, it stays were it is
- What about I/O?
  - Job with frequent I/O will not finish its quantum and stay at the same level
- Parameter
  - quantum size for each queue

Moving Up

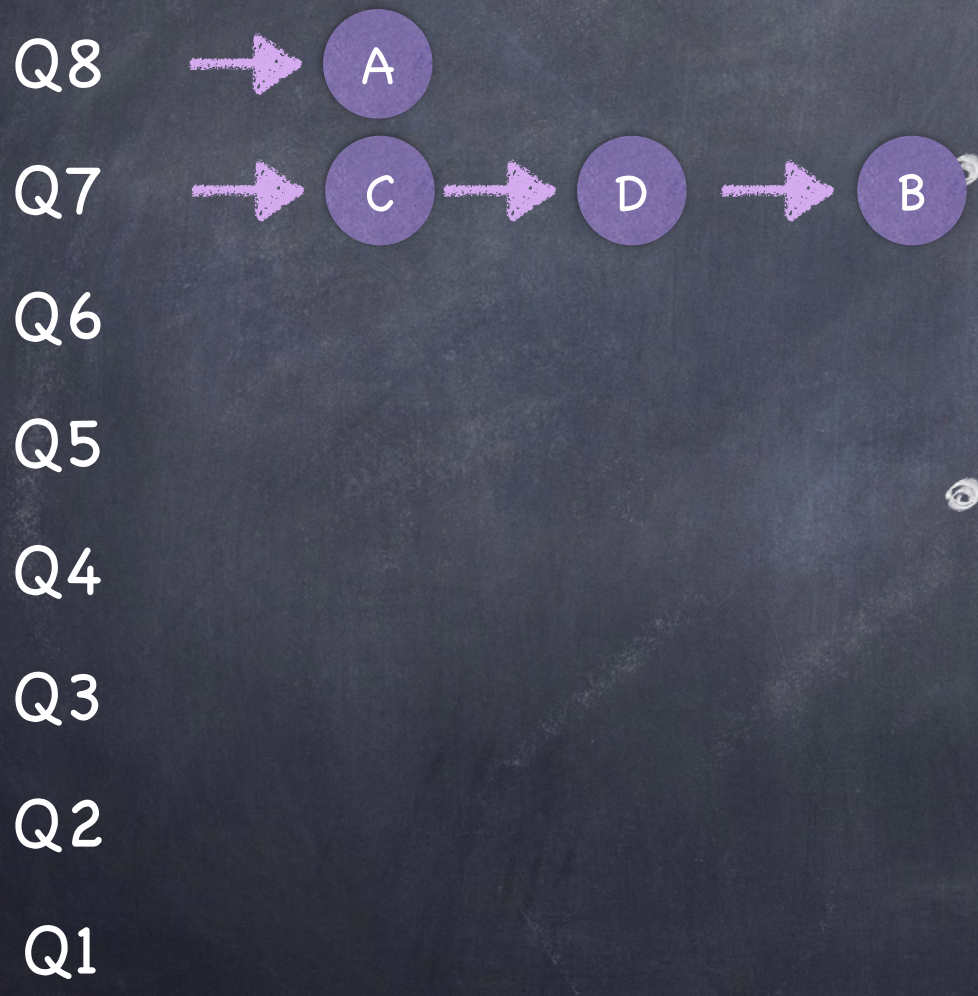


- A job's behavior can change
  - After a CPU-bound interval, process may become I/O bound
- Must allow jobs to climb up the priority ladder...
  - As simple as periodically placing all jobs in the top queue, until they percolate down again

Moving Up

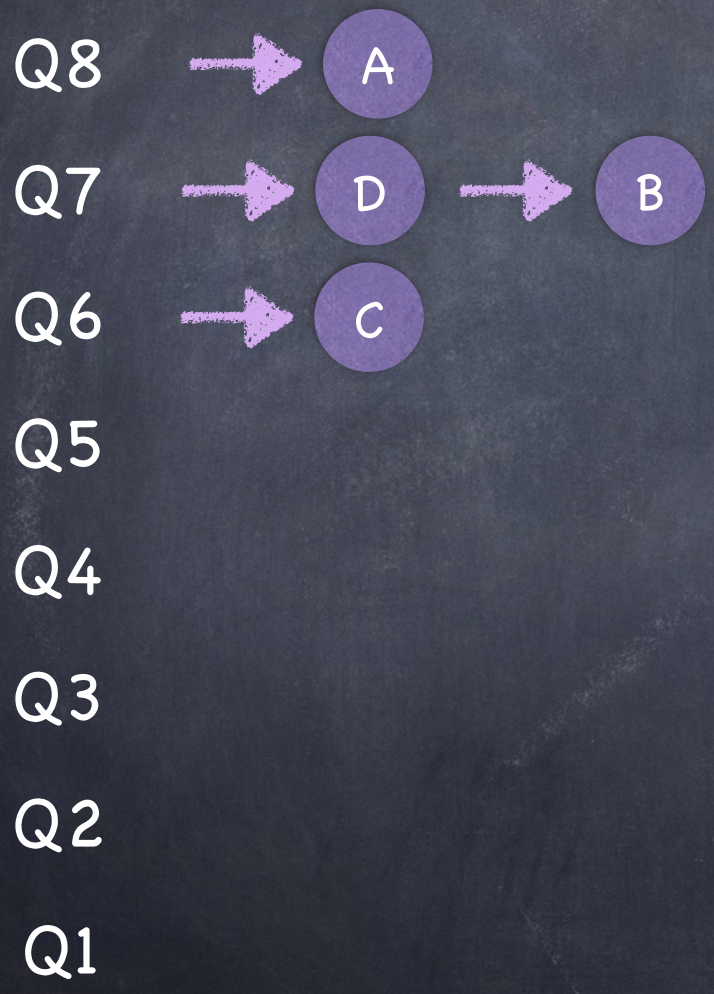
- A job's behavior can change
  - After a CPU-bound interval, process may become I/O bound
- Must allow jobs to climb up the priority ladder...
  - As simple as periodically placing all jobs in the top queue, until they percolate down again

Moving Up



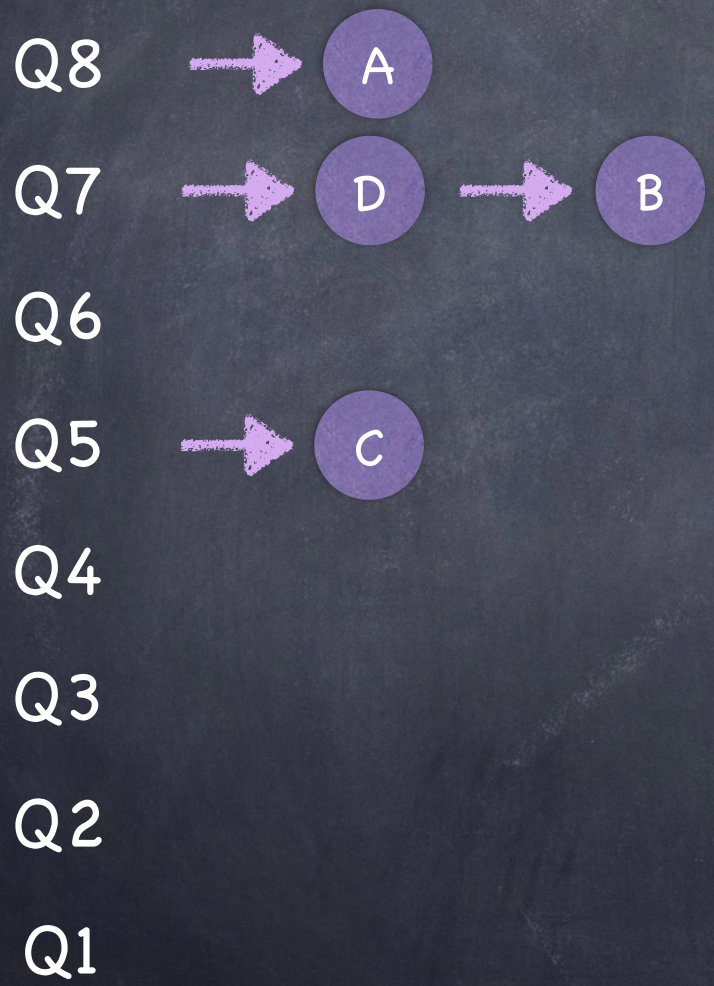
- A job's behavior can change
  - After a CPU-bound interval, process may become I/O bound
- Must allow jobs to climb up the priority ladder...
  - As simple as periodically placing all jobs in the top queue, until they percolate down again

Moving Up



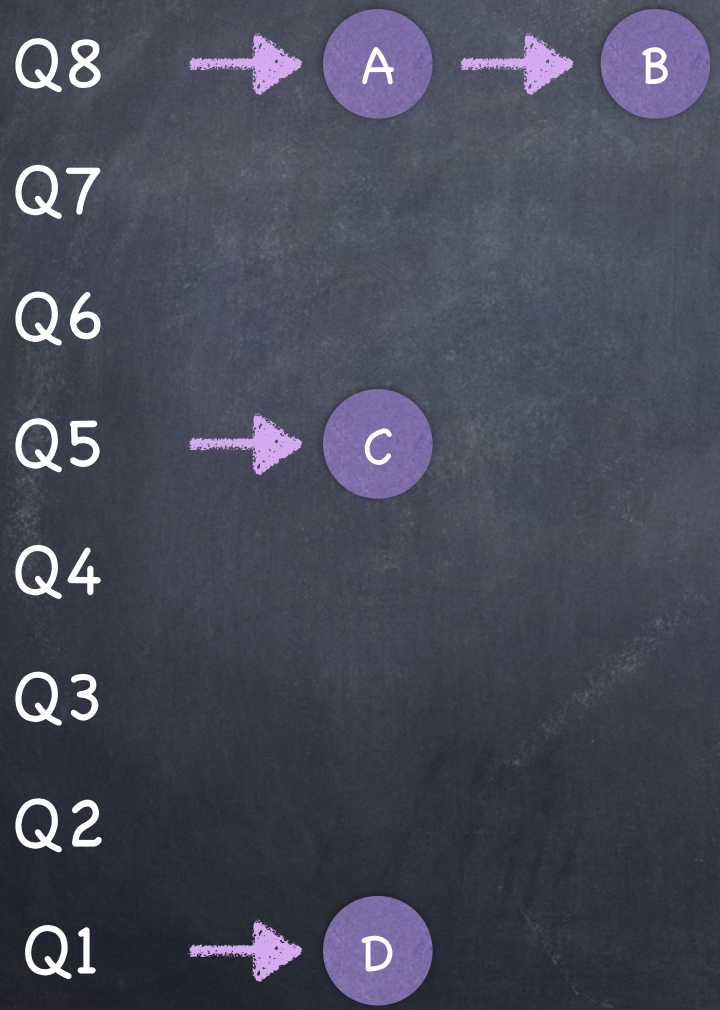
- A job's behavior can change
  - After a CPU-bound interval, process may become I/O bound
- Must allow jobs to climb up the priority ladder...
  - As simple as periodically placing all jobs in the top queue, until they percolate down again

Moving Up



- A job's behavior can change
  - After a CPU-bound interval, process may become I/O bound
- Must allow jobs to climb up the priority ladder...
  - As simple as periodically placing all jobs in the top queue, until they percolate down again
- Parameter
  - time before jobs are moved up

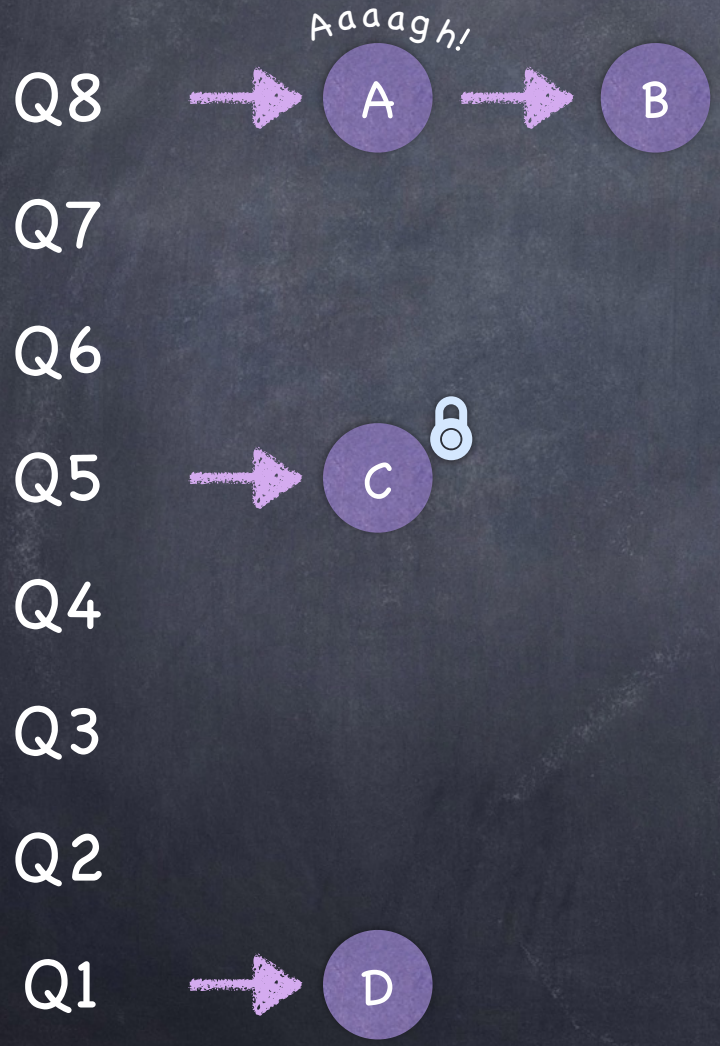
Sneeeeeakyyy...



- Say that I have a job that requires a lot of CPU
  - Start at the top queue
  - If I finish my quantum, I'll be demoted...

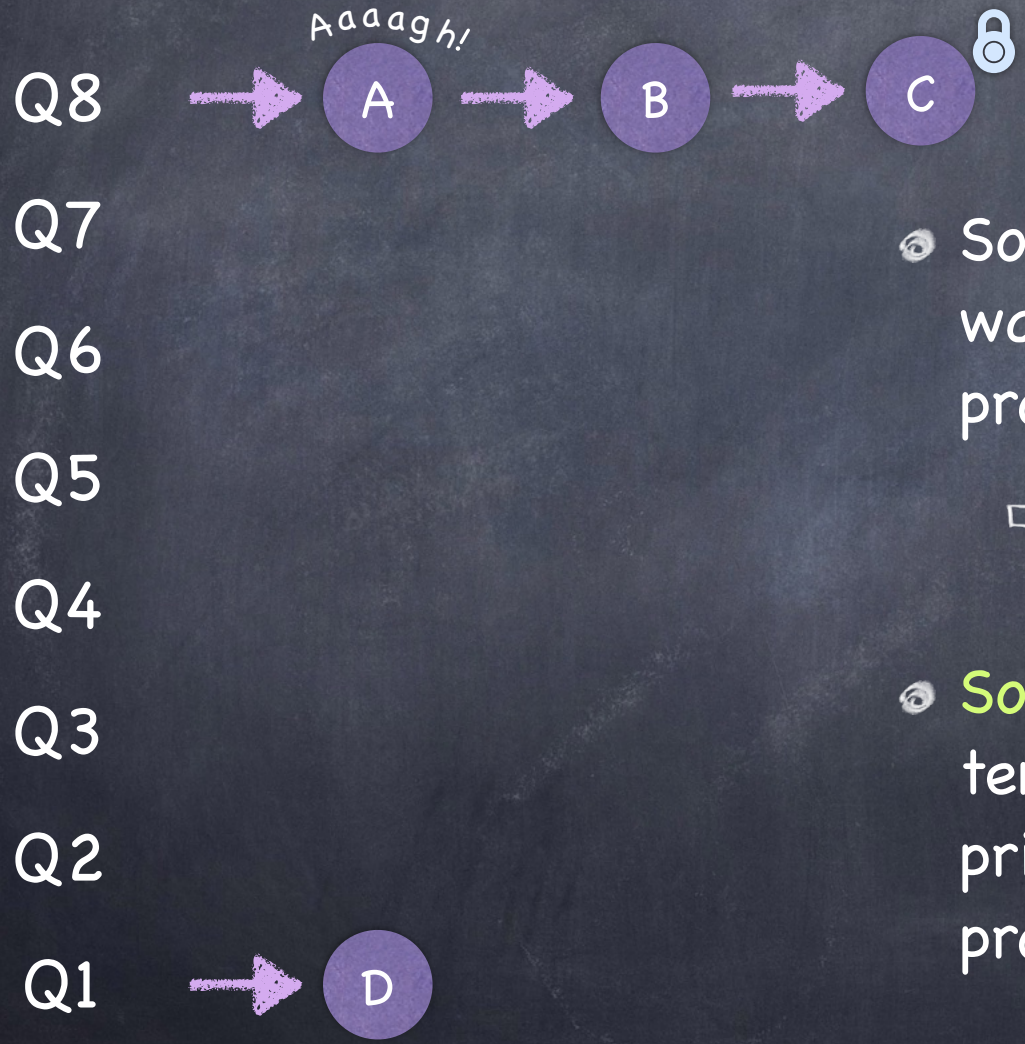
  - ...just give up the CPU before my quantum expires!
- Remedy: Better accounting
  - fix a job's time budget at each level, no matter how it is used
  - more scheduler overhead

Priority Inversion



- Some high priority process is waiting for some low priority process
  - e.g., low priority process has a lock on some resources
- Solution: Process needing lock temporarily bestows its high priority to lower priority process with lock

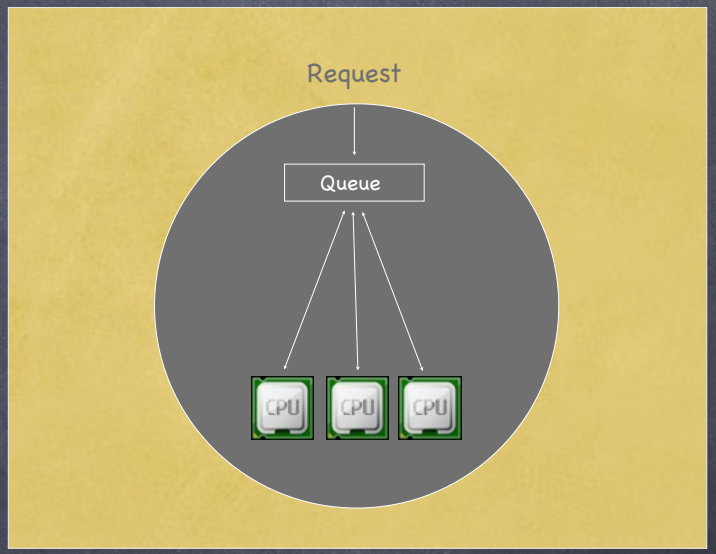
Priority Inversion



- Some high priority process is waiting for some low priority process
  - e.g., low priority process has a lock on some resources
- Solution: Process needing lock temporarily bestows its high priority to lower priority process with lock

Multi-core Scheduling: Sequential Applications

- A web server
  - A thread per user connection
  - Threads are I/O bound (access disk/network)
    - favor short jobs!



An MFQ, right?

- Idle cores take task off MFQ
- Only one core at a time gets access to MFQ
- If thread returns from I/O, back on the MFQ

Single MFQ Considered Harmful

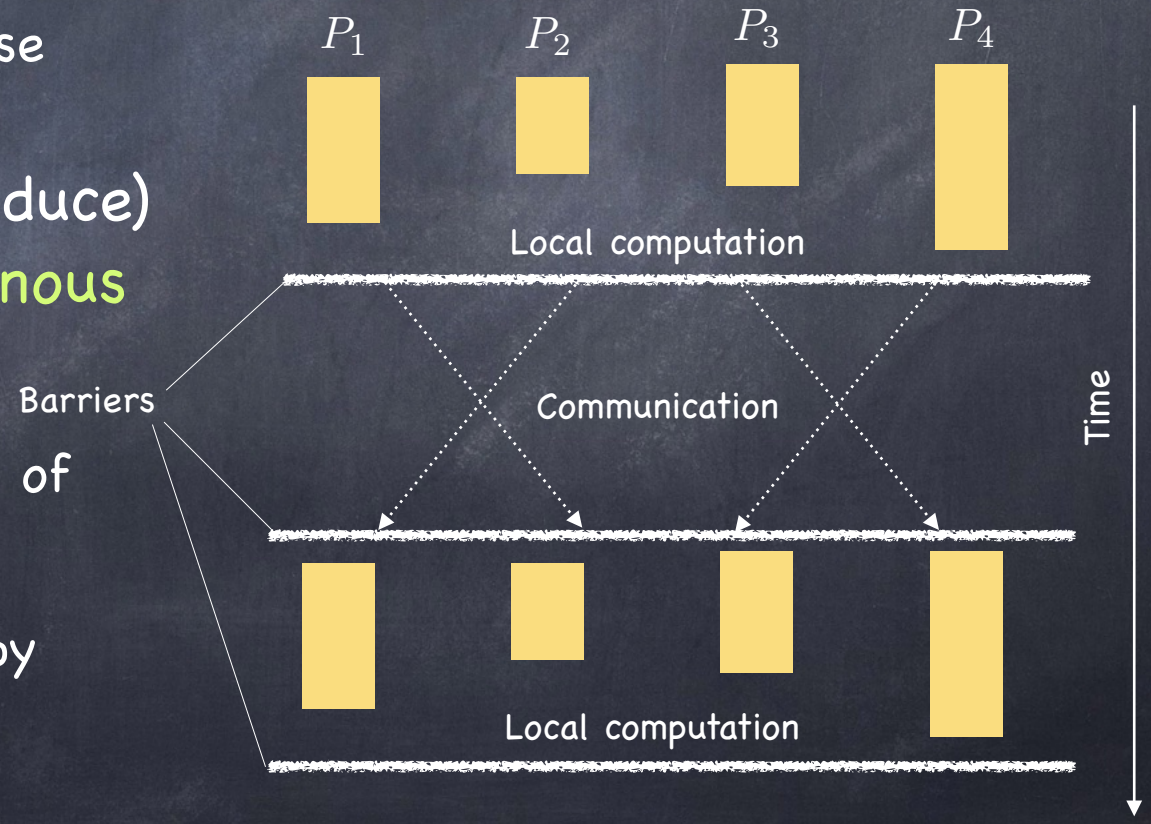
- Contention on MFQ lock
- Limited cache reuse
  - since threads hop from core to core
- Cache coherence overhead
  - core needs to fetch current MFQ state
  - on a single core, likely to be in the cache
  - on a multicore, likely to be in the cache of another processor
    - 2-3 orders of magnitude more expensive to fetch

To Each (Process), its Own (MFQ)

- Cores use affinity scheduling
  - each thread is run repeatedly on the same core
    - maximizes cache reuse
  - more complex to achieve on a single MFQ
- Idle cores can steal work from other processors
  - re-balance load at the cost of some loss of cache efficiency
  - only if it is worth the time of rewarming the cache!

Multicore Scheduling: Parallel Applications

- Application is decomposed in parallel tasks
  - granularity roughly equal to available cores
    - or poor cache reuse
  - Often (e.g., MapReduce) using bulk synchronous parallelism (BSP)
    - tasks are roughly of equal length
    - progress limited by slowest core

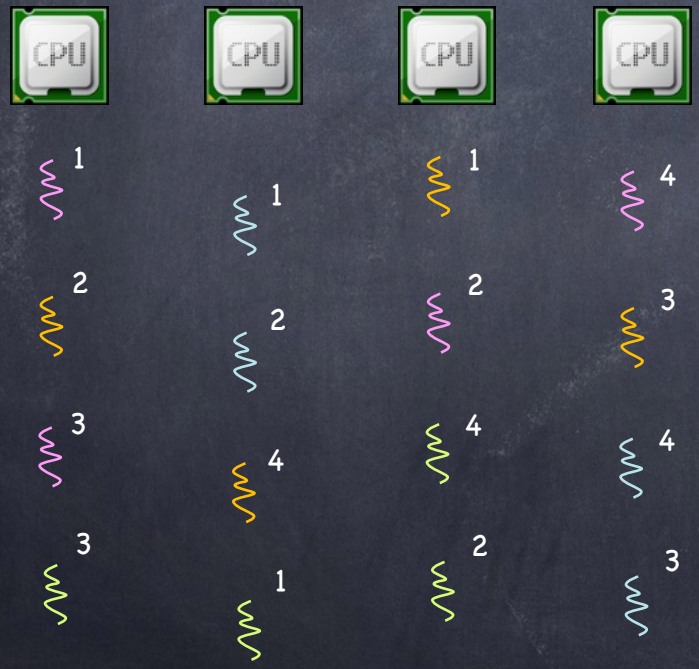


Scheduling Bulk Synchronous Applications

Oblivious Scheduling

Each core time-slices its ready list independently

Four applications, ● ● ● ●, each with four threads



Gang Scheduling

Schedule all tasks from the same application together

Four applications, ● ● ● ●, each with four threads

Length of BSP step determined by last scheduled thread!
Pink thread may be waiting on other pink threads holding lock

Concurrent Programming: Critical Sections & Locks

An OS is a concurrent program

- The "kernel contexts" of each of the processes share many data structures
  - ready queue, wait queues, file system cache, and much more
- Interrupt handlers also access those data structures!
- Need to learn how to share

Lectures Outline - I

- What is the problem?
  - no determinism, no atomicity
- What is the solution?
  - some form of lock
- How to implement locks?
  - there are multiple ways

Concurrent Programming is Hard

- Concurrent programs are non-deterministic
  - run twice with same input, get different answers
  - one time it works, another it fails
- Program statements are executed non-atomically
  - $x += 1$ compiles to something like
    - LOAD x
    - ADD 1
    - STORE x