Context switch overhead

- Cost of saving registers (including, if appropriate, page table register)
- Cost of scheduler determining which process to run next
- Cost of restoring registers (including, if appropriate, page table register)
- Cost of flushing caches
  - L1, L2, L3, TLB

Basic Scheduling Algorithms

- FIFO (First In First Out) a.k.a. FCFS
- SJF (Shortest Job First)
- EDF (Earliest Deadline First)
  - preemptive
- Round Robin
  - preemptive
- Shortest Remaining Time First (SRTF)
  - preemptive

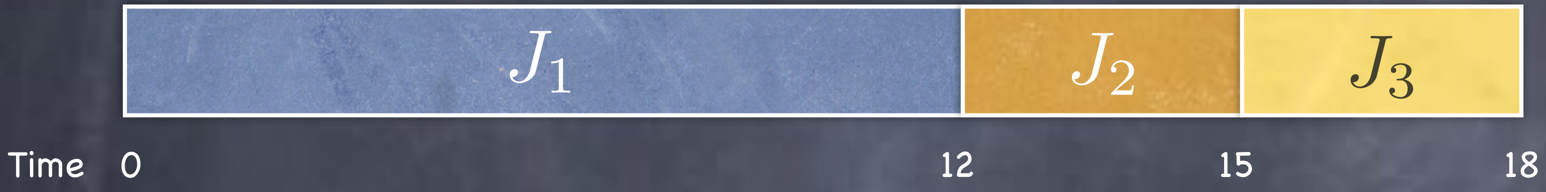
FIFO

- Jobs J_1, J_2, J_3 with compute time 12, 3, 3. Same arrival time (so can be scheduled in any order)
    - Scenario 1: Schedule order J_1, J_2, J_3

Average
Turnaround Time:
(12+15+18)/3 = 15

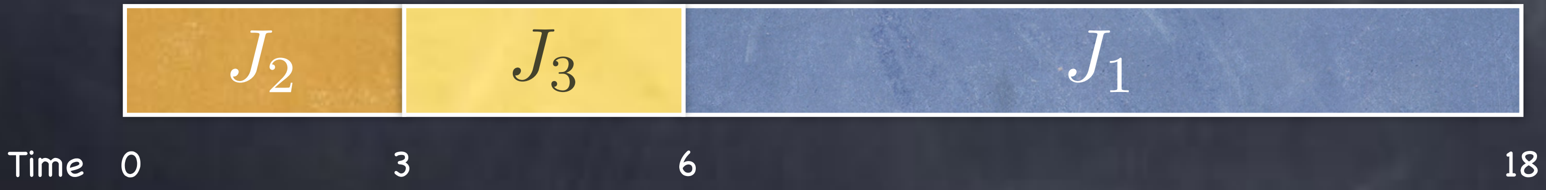
FIFO

- Jobs J_1, J_2, J_3 with compute time 12, 3, 3. Same arrival time (so can be scheduled in any order)
  - Scenario 1: Schedule order J_1, J_2, J_3



Average
Turnaround Time:
(12+15+18)/3 = 15

  - Scenario 2: Schedule order J_2, J_3, J_1



Average
Turnaround Time:
(3+6+18)/3 = 9

Average turnaround time very sensitive to schedule order!

FIFO Roundup

Simple
Low overhead
No starvation

Average turnaround time very sensitive to schedule order/arrival time

Not responsive to interactive tasks

How to minimize average turnaround time?

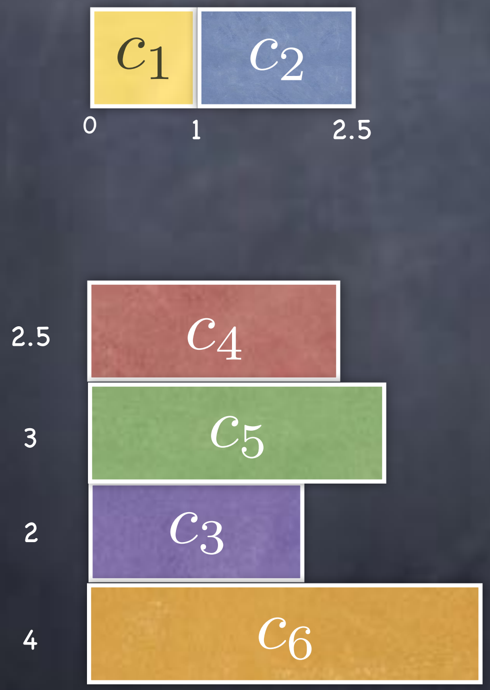
SJF: Shortest Job First

- Schedule jobs in order of estimated completion time

  (or, better, shortest length of next CPU burst!)

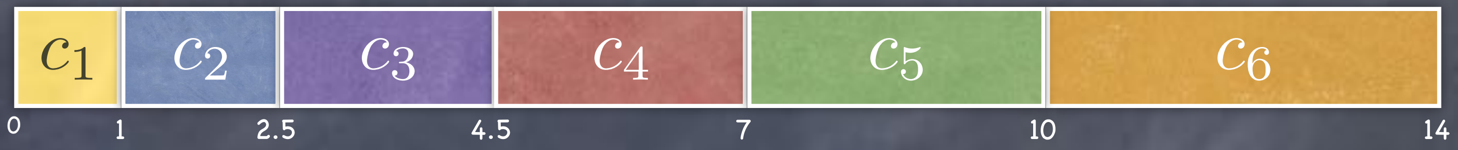
SJF: Shortest Job First

- Schedule jobs in order of estimated completion time



SJF: Shortest Job First

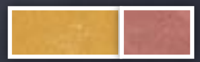
- Schedule jobs in order of estimated completion time



- Average Turnaround time (att): 39/6 = 6.5
- Would a different schedule produce a lower turnaround time?

Consider

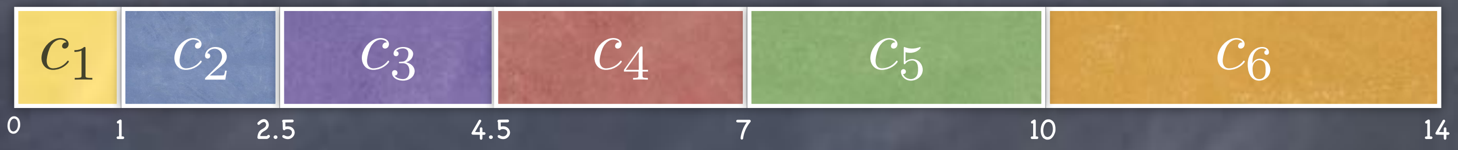

where $c_i < c_j$



att = $(c_j + (c_i + c_j))/2$

SJF: Shortest Job First

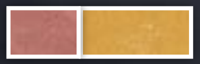
- Schedule jobs in order of estimated completion time



- Average Turnaround time (att): 39/6 = 6.5
- Would a different schedule produce a lower turnaround time?

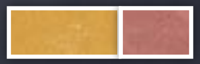
Consider

where $c_i < c_j$



att = $(c_i + (c_i + c_j))/2$

$<$



att = $(c_j + (c_i + c_j))/2$

SJF Roundup

Optimal average turnaround time

Pessimal variance in turnaround time for a given task

Need to estimate execution time

Can starve long jobs

SJF Roundup

Optimal average turnaround time

Pessimal variance in turnaround time for a given task

Need to estimate execution time

Can starve long jobs

Shortest Process Next (SJF for interactive jobs)

- Enqueue in order of estimated completion time
  - Exponential moving average (EMA): Use recent history as indicator of near future
- Let $t_n =$ duration of n^{th} CPU burst

  $\tau_n =$ estimated duration of n^{th} CPU burst

  $\tau_{n+1} =$ estimated duration of next CPU burst

$$\tau_{n+1} = \alpha\tau_n + (1-\alpha)t_n$$

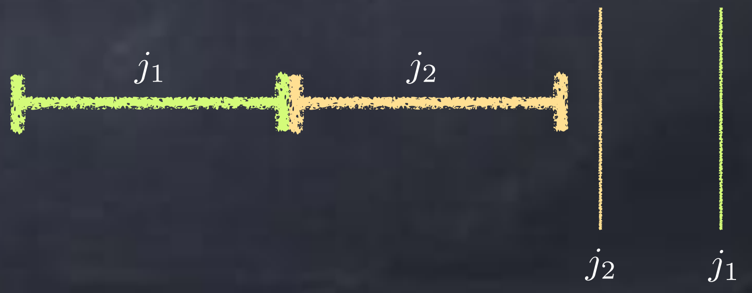
$0 \leq \alpha \leq 1$ determines weight placed on past behavior

Earliest Deadline First (EDF)

- Schedule in order of earliest deadline
- If a schedule exists that meets all deadlines, then EDF will generate that schedule!
  - does not even need to know the execution times of the jobs

Informal Proof

- Let S be a schedule of a set of jobs that meets all deadlines
- Let j_1 and j_2 be two neighboring jobs in S so that j_1.deadline > j_2.deadline
- Let S′ be S with j_1 and j_2 switched
  - S′ also meets all deadlines!
- Repeat until sorted (i.e., bubblesort)
  - Resulting schedule is EDF



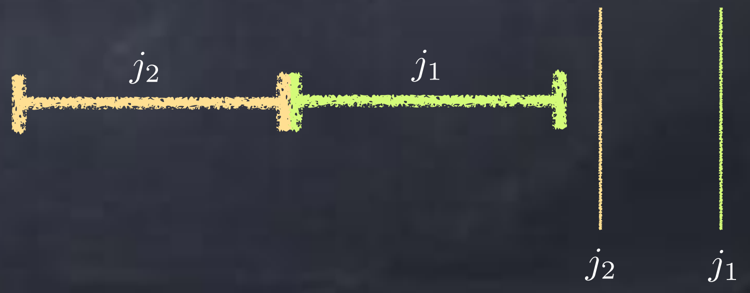
Earliest Deadline First (EDF)

- Schedule in order of earliest deadline
- If a schedule exists that meets all deadlines, then EDF will generate that schedule!
  - does not even need to know the execution times of the jobs

..but only if tasks only need the processor!

Informal Proof

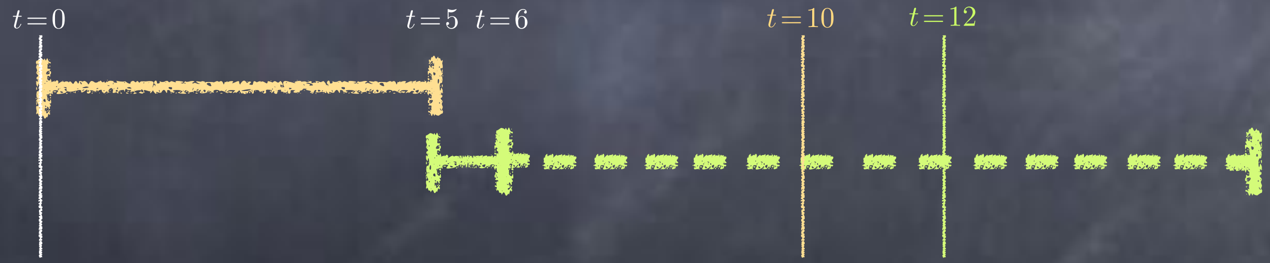
- Let S be a schedule of a set of jobs that meets all deadlines
- Let j_1 and j_2 be two neighboring jobs in S so that j_1.deadline > j_2.deadline
- Let S' be S with j_1 and j_2 switched
  - S' also meets all deadlines!
- Repeat until sorted (i.e., bubblesort)
  - Resulting schedule is EDF



When EDF fails

- Two jobs:
  - j_1: deadline at $t=12$; 1 unit of computation, 10 of I/O
  - j_2: deadline at $t=10$; 5 units of computation

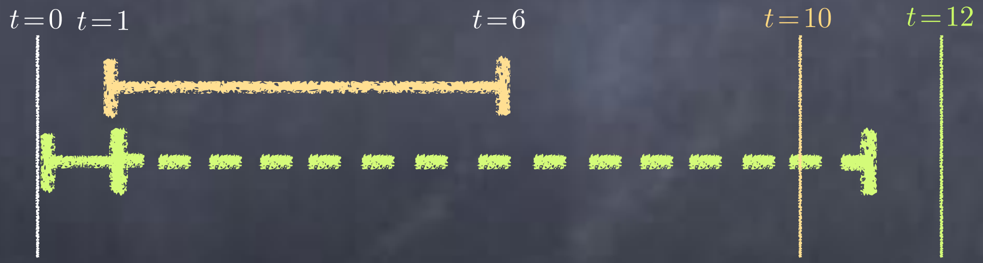
EDF:



but...

When EDF fails

- Two jobs:
    - j_1: deadline at $t=12$; 1 unit of computation, 10 of I/O
    - j_2: deadline at $t=10$; 5 units of computation



- Need to think of jobs at a finer granularity:
    - Real deadline for the computing portion of j_1 is 2!

EDF Roundup

Meets deadlines if possible (but...)

Free of starvation

Does not optimize other metrics

Cannot decide when to run jobs without deadlines

Round Robin

- Each process is allowed to run for a quantum
- Context is switched (at the latest) at the end of the quantum — preemption!
- Next job to run is the one that hasn't run for the longest amount of time
- What is a good quantum size?
  - Too long, and it morphs into FIFO
  - Too short, and much time lost context switching
  - Typical quantum: about 100X cost of context switch (~100ms vs. << 1ms)

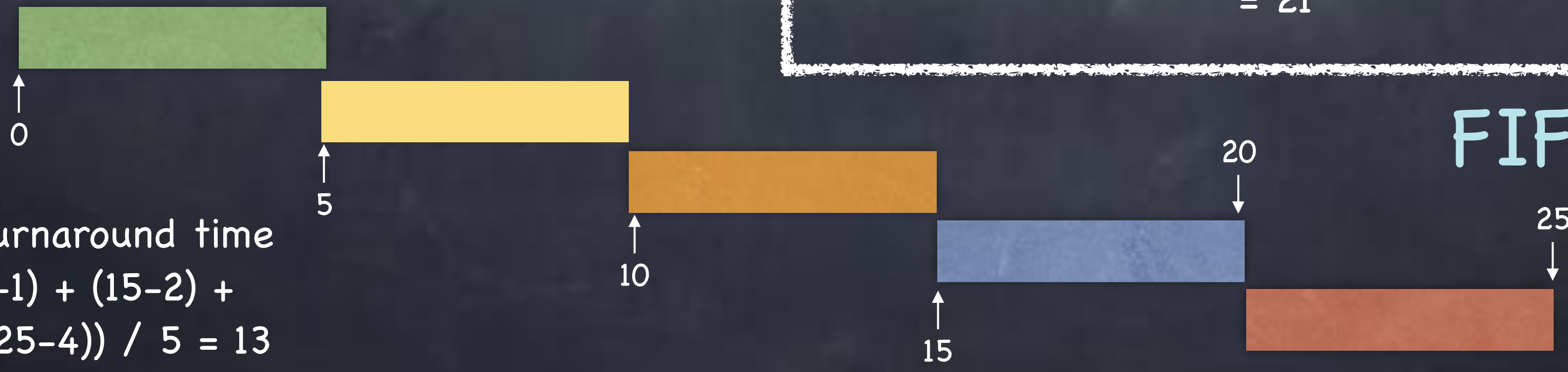
Round Robin vs FIFO

Jobs of about equal length (5 TU) start at about the same time

RR

Average Turnaround time
(21 + (22-1) + (23-2) + (24-3) + (25-4)) / 5
= 21

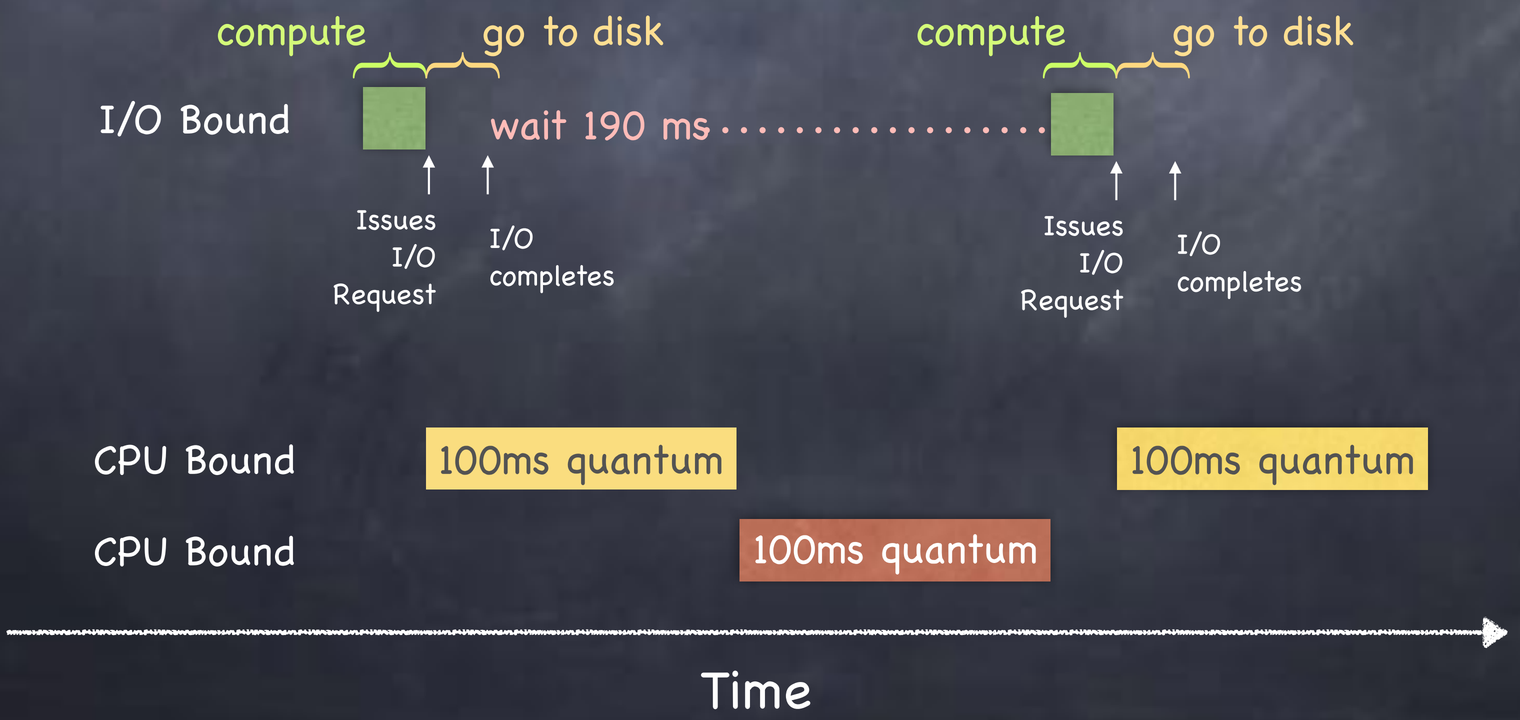
FIFO/SJF



Average Turnaround time
(5 + (10-1) + (15-2) +
(20-3) + (25-4)) / 5 = 13

At least it is fair...?

- Mix of one I/O-bound and two CPU-bound jobs
  - I/O-bound: compute; go to disk; repeat



Round Robin Roundup

No starvation

Can reduce response time

Overhead of context switching

Mix of I/O and CPU bound

Particularly bad average turnaround for simultaneous, equal length jobs

SJF

- J_1 arrives at time 0; J_2, J_3 arrive at time 10

Average Turnaround Time:
100+(110-10)+(120 -10)/3
= 103.33

SJF + Preemption

- J_1 arrives at time 0; J_2, J_3 arrive at time 10

Average Turnaround Time:
100+(110-10)+(120 -10)/3
= 103.33

- With a preemptive scheduler — SRTF

Shortest Remaining
Time First

At end of each quantum, scheduler selects job with the least remaining time to run next

- ☐ Often same job is selected, avoiding a context switch…
- ☐ …but new short jobs see improved response time

Average Turnaround Time:
(120-0)+(20-10)+(30-10)/3
= 50

SRTF Roundup

Good response time and turnaround time of I/O bound processes

Bad turnaround time and response time for CPU bound processes

Need estimate of execution for each job

Starvation

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)

Used by most versions of Linux!

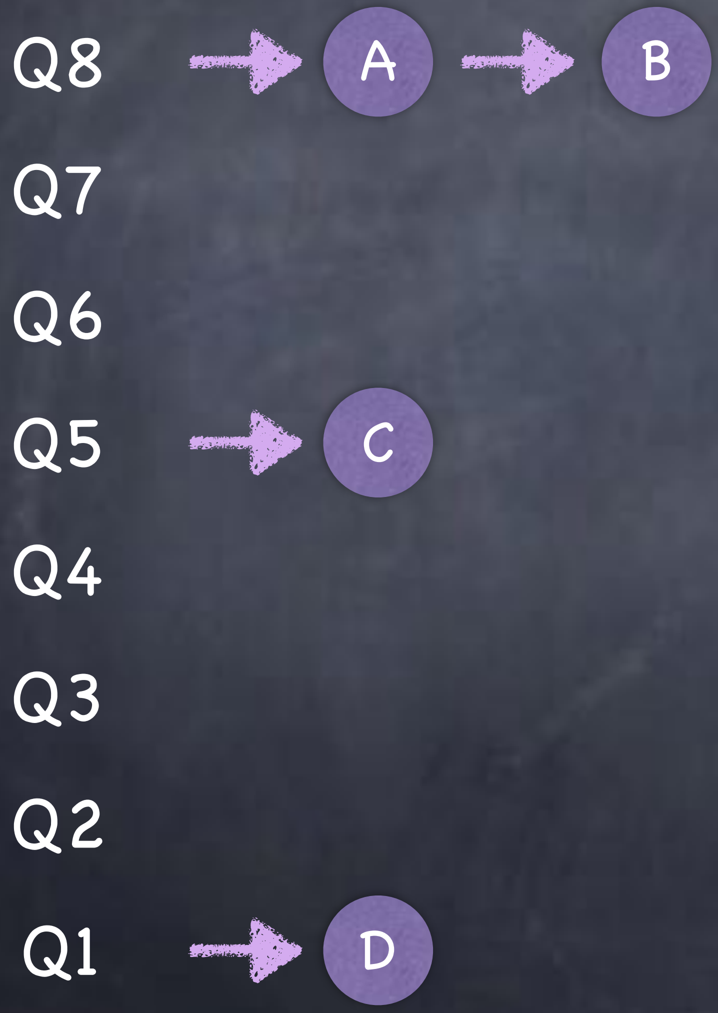
Spent Execution Time

- 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, reniserted 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
  - SET$_p$ = max (SET$_p$, min{SET of ready processes})
- To account for priority, SET grows slower for higher priority processes

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 the job changes its pattern of CPU usage

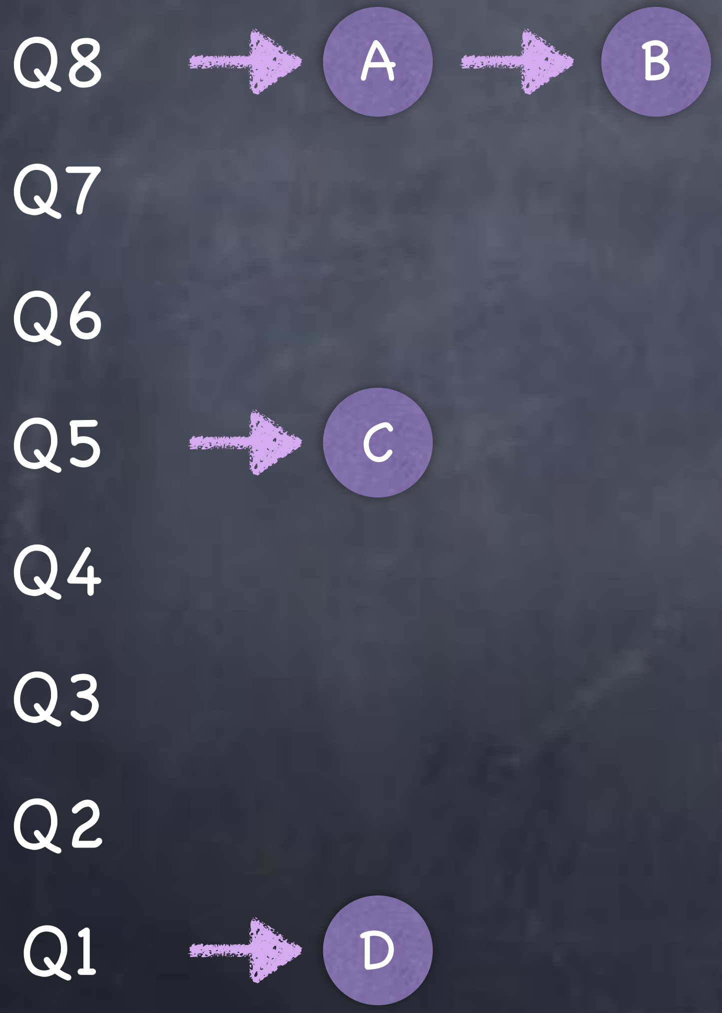
The Basic Structure



- 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 RR
- Parameter:
  - how many queues?

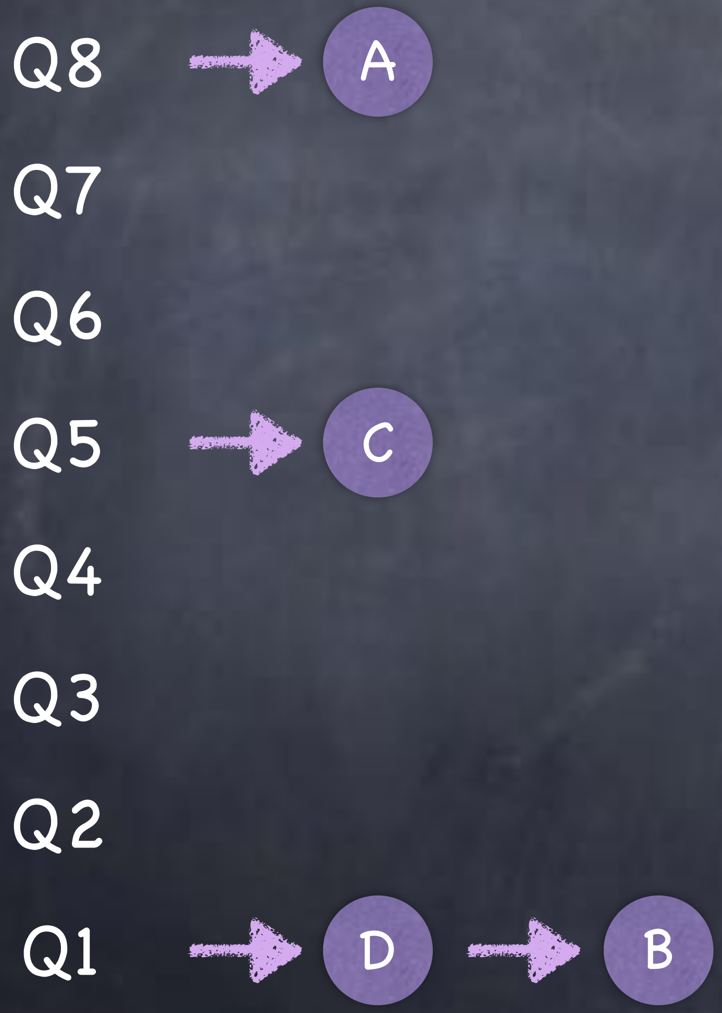
How are jobs assigned to a queue?

Moving down



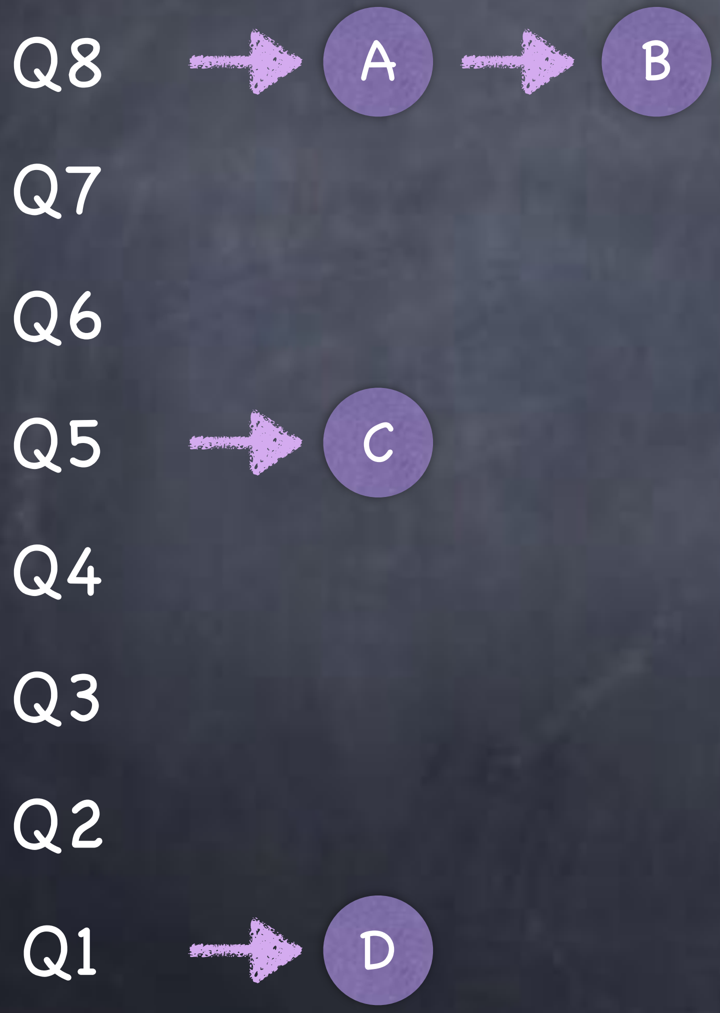
- Job starts at the top level
- If it uses full quantum before giving up CPU, moves down
- Otherwise, it stays were it is
- What about I/O?
  - Job with frequent I/O will not finish its quantum and stay at the same level
- Parameter
  - quantum size for each queue

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
- Parameter
  - time before jobs are moved up

Sneeeeakyyy…



Q8 → A → B

Q7

Q6

Q5 → C

Q4

Q3

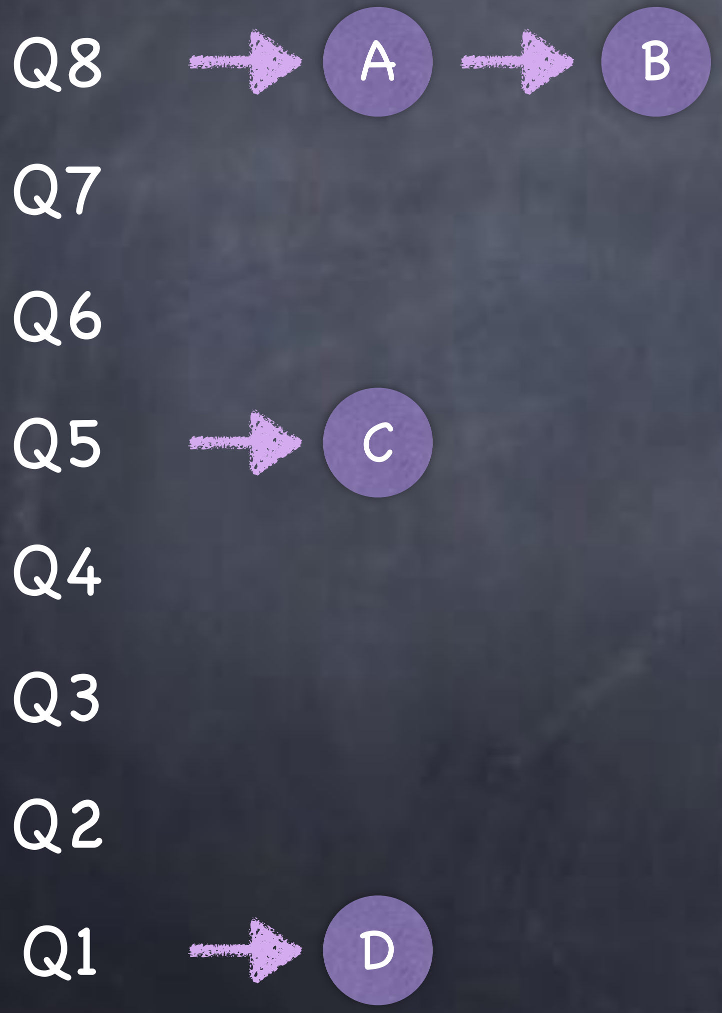
Q2

Q1 → D

- 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!
- Better accounting
    - fix a job's time budget at each level, no matter how it is used
    - more scheduler overhead

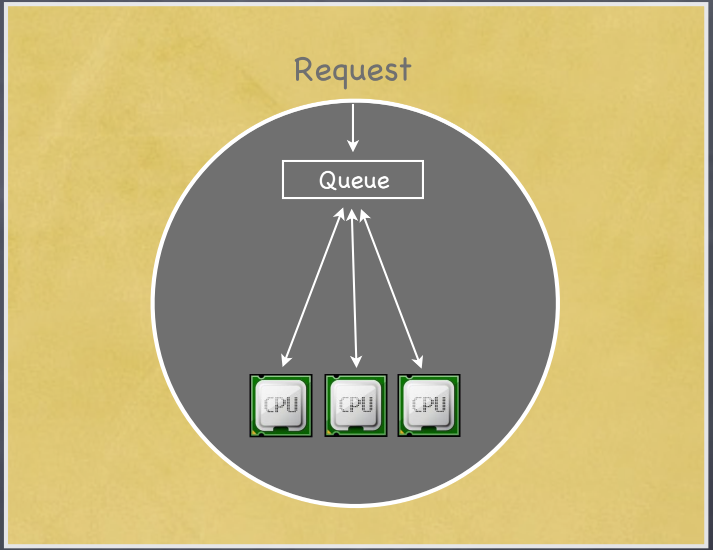
Priority Inversion



- 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

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 blocks, back on the MFQ

Single MFQ Considered Harmful

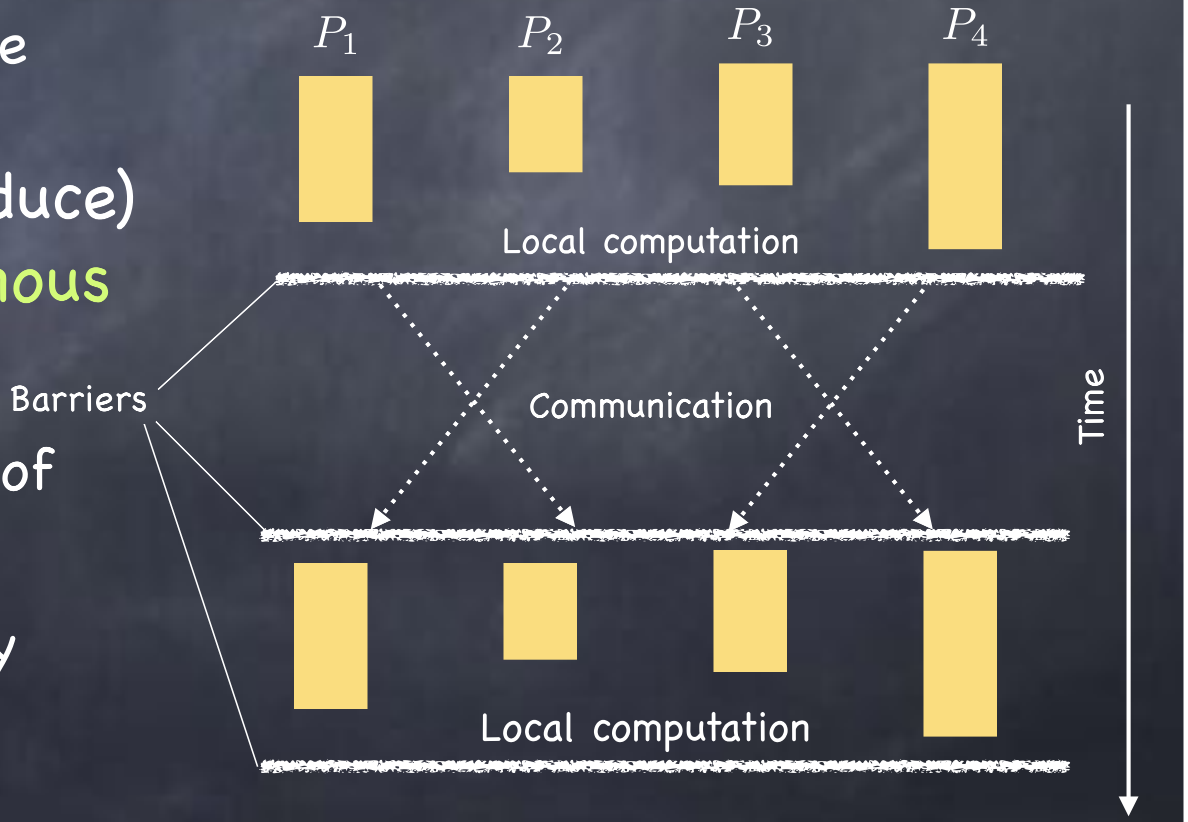
- Contention on MFQ lock
- Limited cache reuse
  - since threads hop from core to core
- Cache coherence overhead
  - core needs to fetch current MFQ state
  - on a single core, likely to be in the cache
  - on a multicore, likely to be in the cache of another processor
    - 2-3 orders of magnitude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

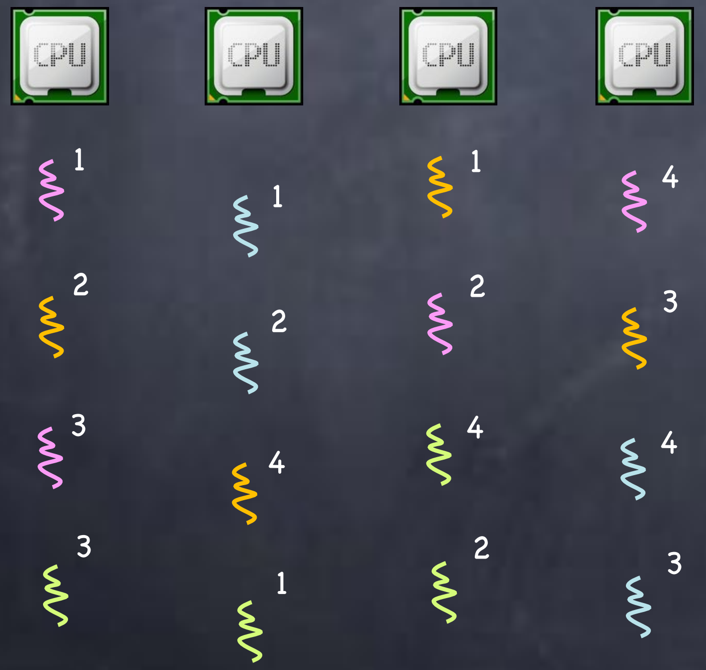


Scheduling Bulk Synchronous Applications

Oblivious Scheduling

Each process time-slices its ready list independently

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

Gang Scheduling

Schedule all tasks from the same program together

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