Operating Systems
Lecture 7

CPU Scheduling
[Cont.]

Announcements

- **HW1 due next week**
- **Quiz time!**
  - Quizzes can happen in any lecture from now on.
  - Policy:
    - I understand that things come up..
    - If you are unable to attend a lecture
      - Inform us **at least 1 hour in advance**
      - Email to: cs4410-staff@cornell.edu
      - We will ignore that quiz for you

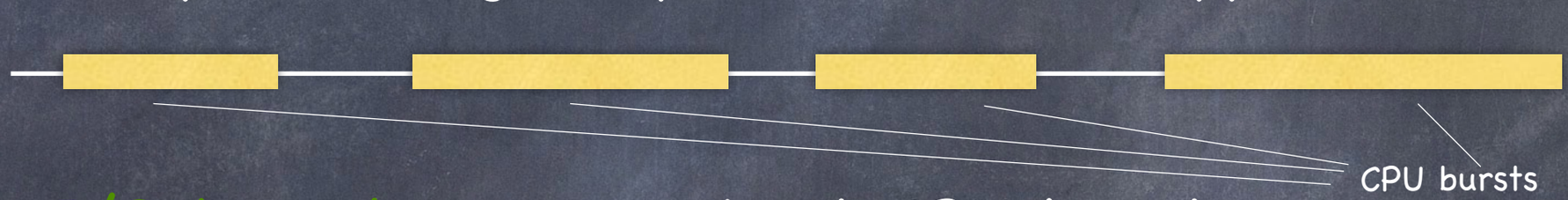
Goal for today's lecture

- **Continue understanding CPU scheduling problem**
  - Various algorithms
  - And tradeoffs
- **Designing an ideal CPU scheduler**
  - Ourselves ….
  - In today's class ….

Recap:
CPU scheduling so far

Recall: Why scheduling is challenging

- Processes are not created equal!
  - CPU-bound process: long CPU bursts
    - mp3 encoding, compilation, scientific applications



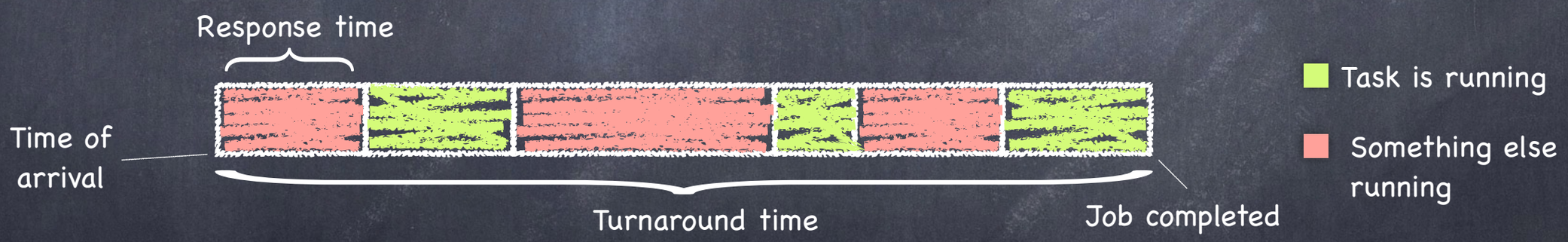
  - I/O-bound process: short CPU bursts
    - index a file system, browse small web pages

- Problem?
  - don't know jobs type before running it
  - jobs behavior can change over time

Recall: Metrics

- Response time
  - How long between job's arrival and first time job runs?
- Total waiting time
  - How much time on ready queue but not running?
    - sum of "red" intervals below

- Execution time: sum of "green" intervals



- Turnaround time: "red" + "green"
  - Time between a job's arrival and its completion
- Throughput: jobs completed/unit of time

Recall: Other Metrics

- Fairness: Who get the resources?
  - Equitable division of resources
- Starvation: How bad can it get?
  - Lack of progress by some job
- Overhead: How much useless work?
  - Time wasted switching between jobs
- Predictability: How consistent?
  - Low variance in response time for repeated requests

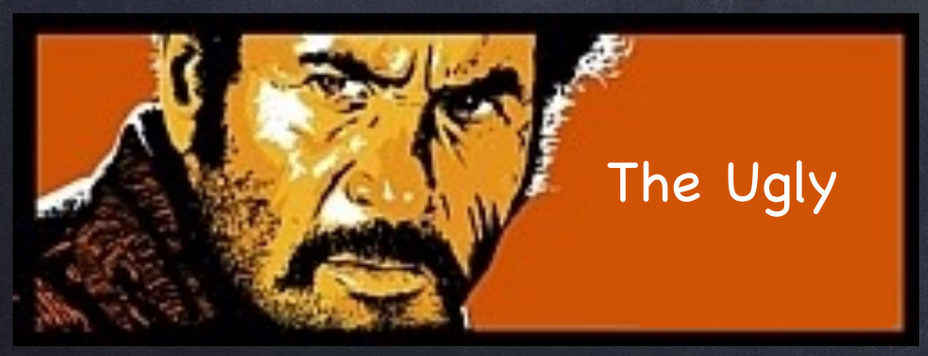
Recall: Basic Scheduling Algorithms

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

Recall: FIFO Roundup

Simple
Low overhead
No starvation

Average turnaround time
very sensitive to arrival time

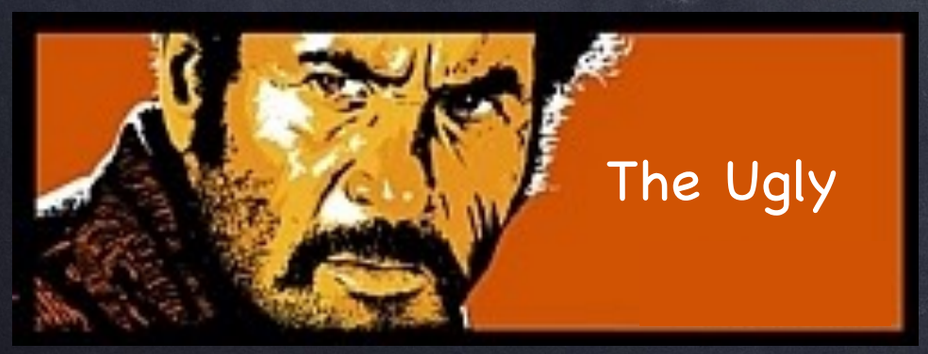


Potentially poor performance for
interactive tasks

Recall: SJF Roundup

Optimal average turnaround time
(Assuming ….?)

Need to estimate execution time



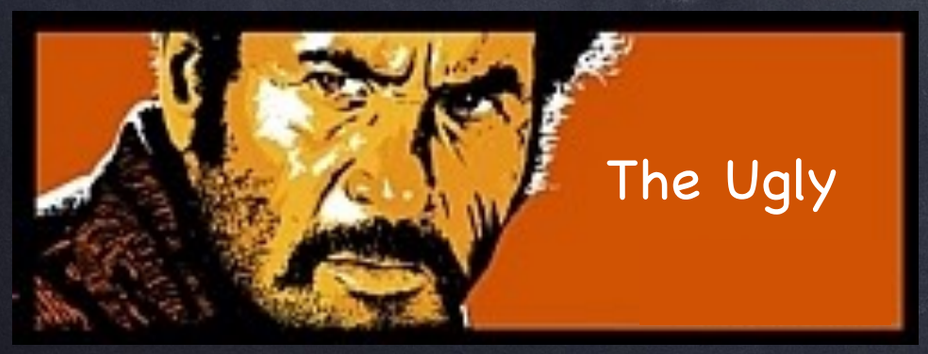
Can starve long jobs

Recall: EDF Roundup

Meets deadlines if possible (but…)

Free of starvation

Does not optimize other metrics



Cannot decide when to run jobs without deadlines

Recall: Round Robin Roundup

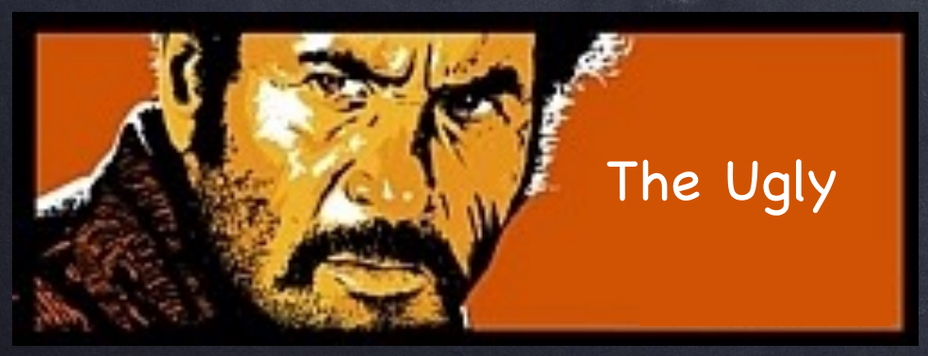
No starvation

Can reduce response time

Fair resource allocation

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

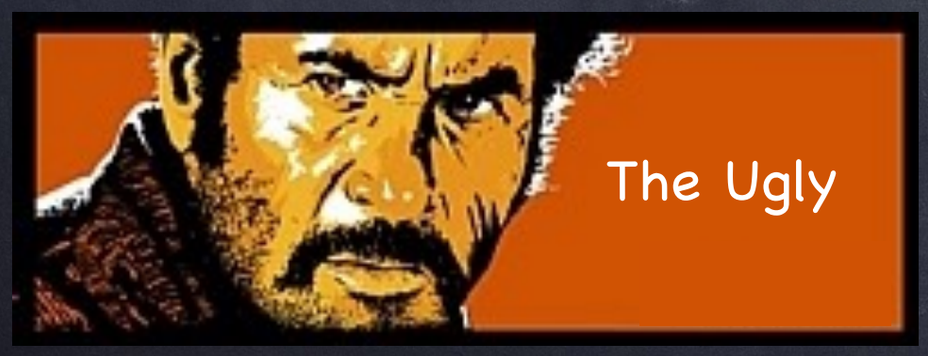
Average Turnaround Time:
(120-0)+(20-10)+(30-10)/3
= 50

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

What about priorities?

After all, all policies so far are prioritizing jobs in some form

Priority Scheduling

- Assign a number (priority) to each job
- Schedule jobs in priority order
- Breaking ties arbitrarily

Priority Scheduling

- What if job priority = job's arrival time?
  - Emulates FCFS
- What if each job has the same priority?
  - Emulates RR (by breaking ties based on jobIDs)
- What if job priority = estimated execution time?
  - Emulates SJF
  - SRTF by setting job priority = estimated remaining exec. time
- What if job priority = job's deadline?
  - Emulates EDF

Priority Scheduling

- Suppose each job has a different priority
- Can we have starvation?
  - Yes!
- Can the highest priority job ever starve?
  - Priority inversion!

Priority Inversion

Priority = 1 (highest) J_1

Priority = 2 J_2

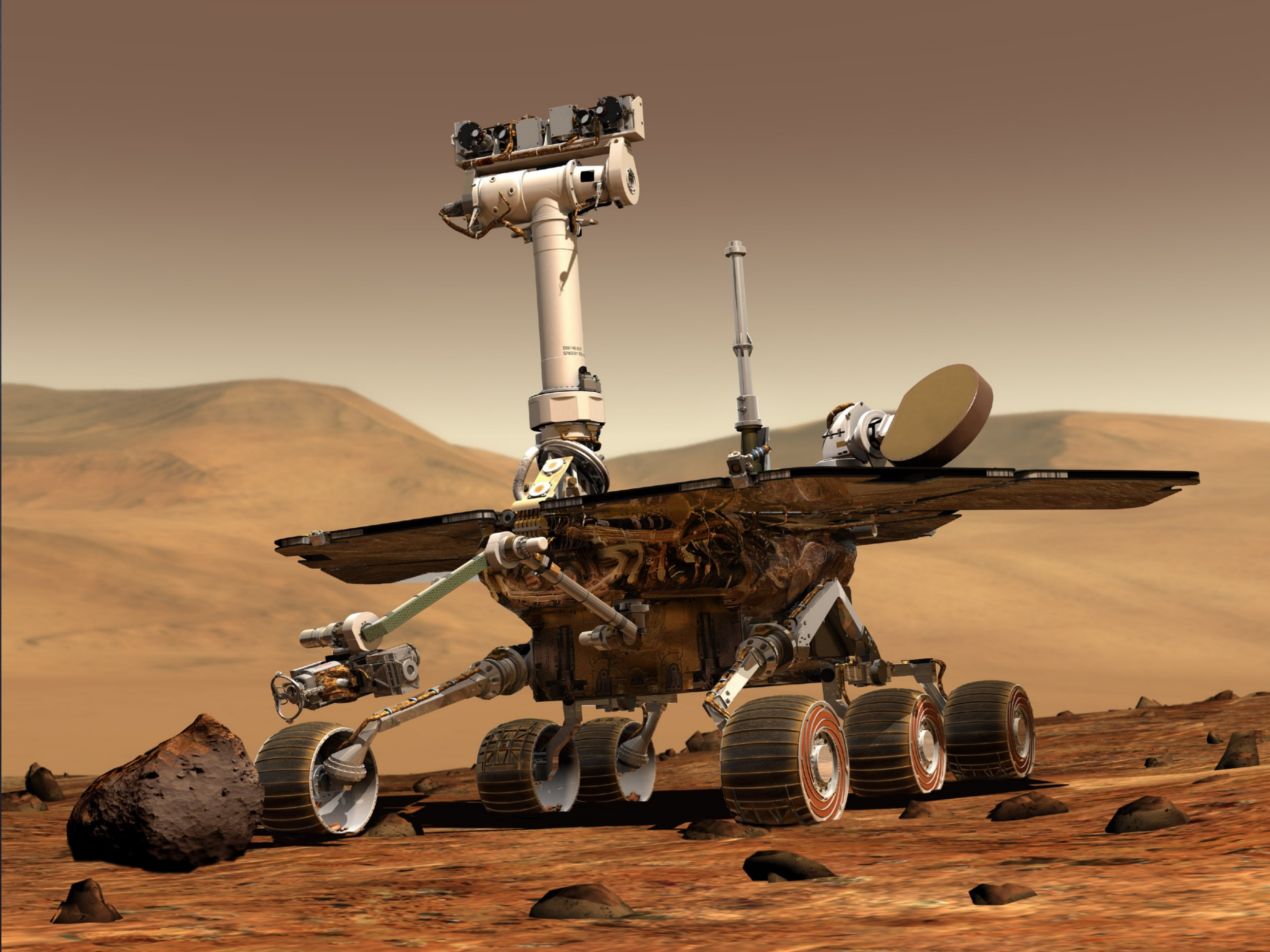
Priority = 3 (lowest) J_3

File shared between J1 and J3

- Suppose J3 is scheduled at some point
  - Acquires a "lock" on the shared file
  - Lock = nobody else can access it until J3 releases the lock
- When J1 scheduled next: It gets blocked
- Which job does the scheduler choose next?
  - J2! (Priority inversion)

Priority Inversion

- High priority tasks blocked by low priority tasks
  - Low priority task must run for high priority to progress
  - Medium priority task can starve a high priority one
- Solution?
- Priority donation/inheritance
  - High priority task "grants" its high priority to low priority task



Priority Inversion in the real world

- July 4th, 1997 —Pathfinder lands on Mars
  - First US Mars landing since Vikings in 1976; first rover
- A few days into mission....
  - Multiple system resets occur
  - System would reboot randomly, losing valuable time & progress
- Problem? Priority inversion!
  - Low priority task (for data collection) grabs a "lock"
  - A high priority task (for data distribution) blocked
  - Had to be reset to safe state
    - automated resetting based on some conditionals
    - Repeat!

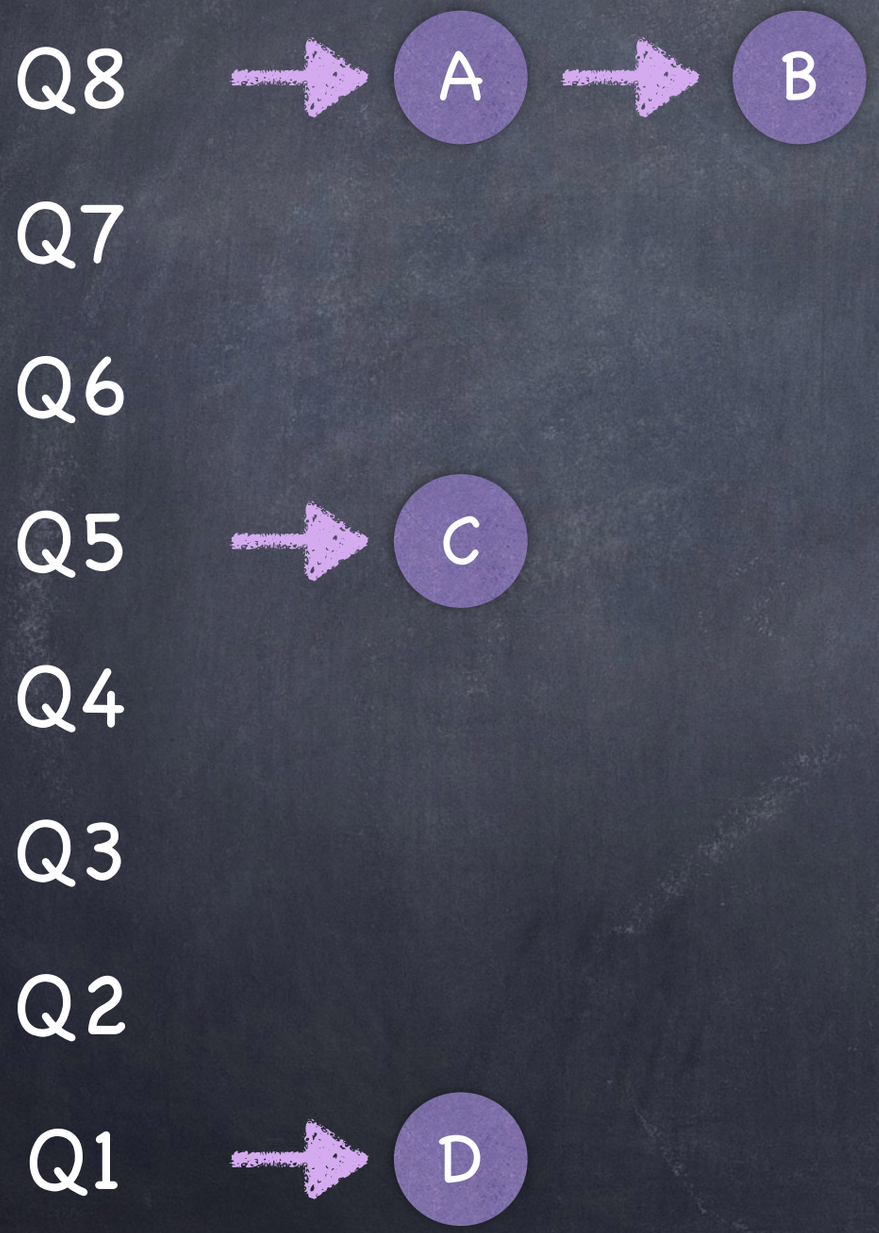
The core problem!

- All our policies so far:
  - preferential job scheduling
    - FCFS, SJF, EDF, SRTF....
  - Or, round robin (poor turnaround time)
- What we really wanted:
  - Efficiently handle **mix** of CPU-bound, I/O bound, interactive jobs, ..
  - What does "efficiently" really mean?
    - Give I/O bound jobs enough CPU to issue their next file op, and wait
    - Give interactive jobs enough CPU to respond to an input, and wait
    - Let CPU-bound jobs grind away without too much disturbance

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

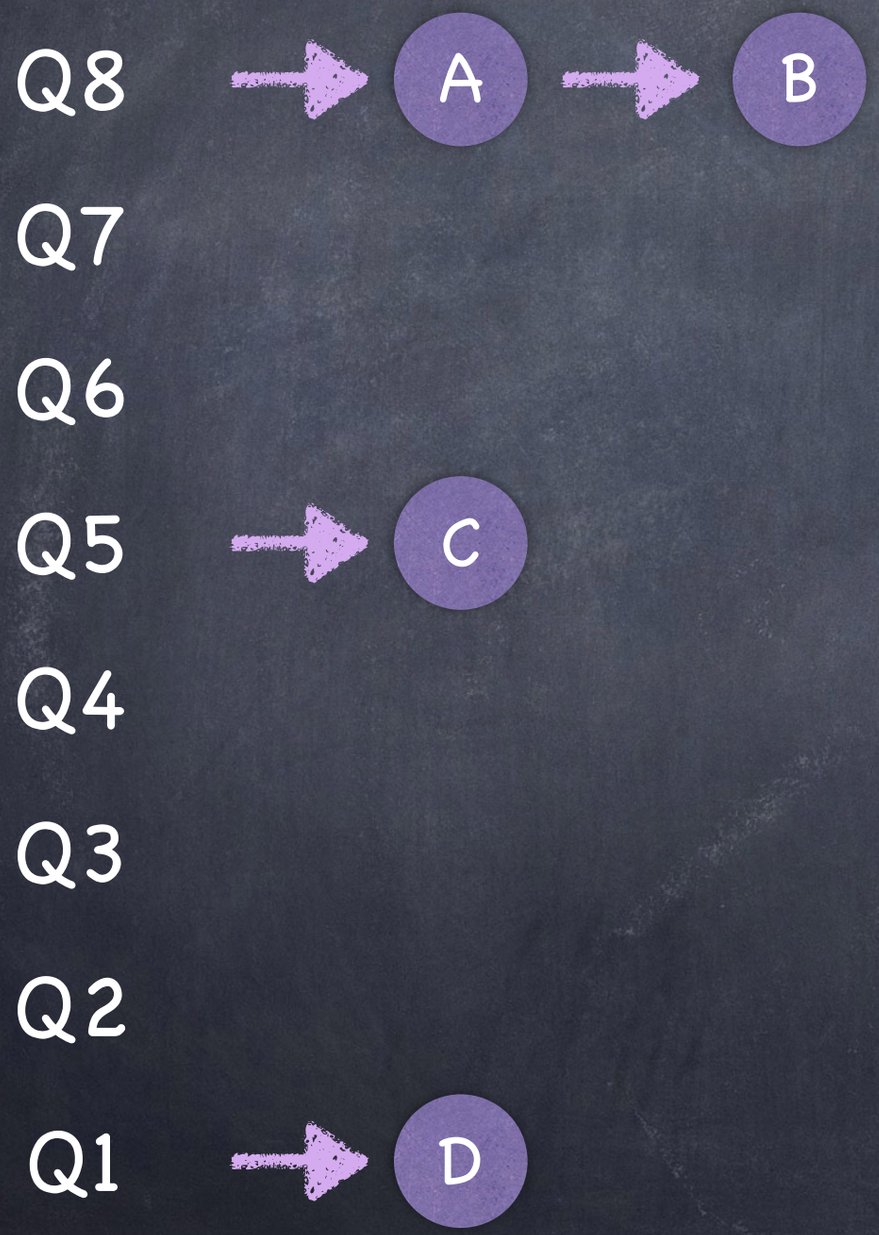
The Basic Structure



- 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?

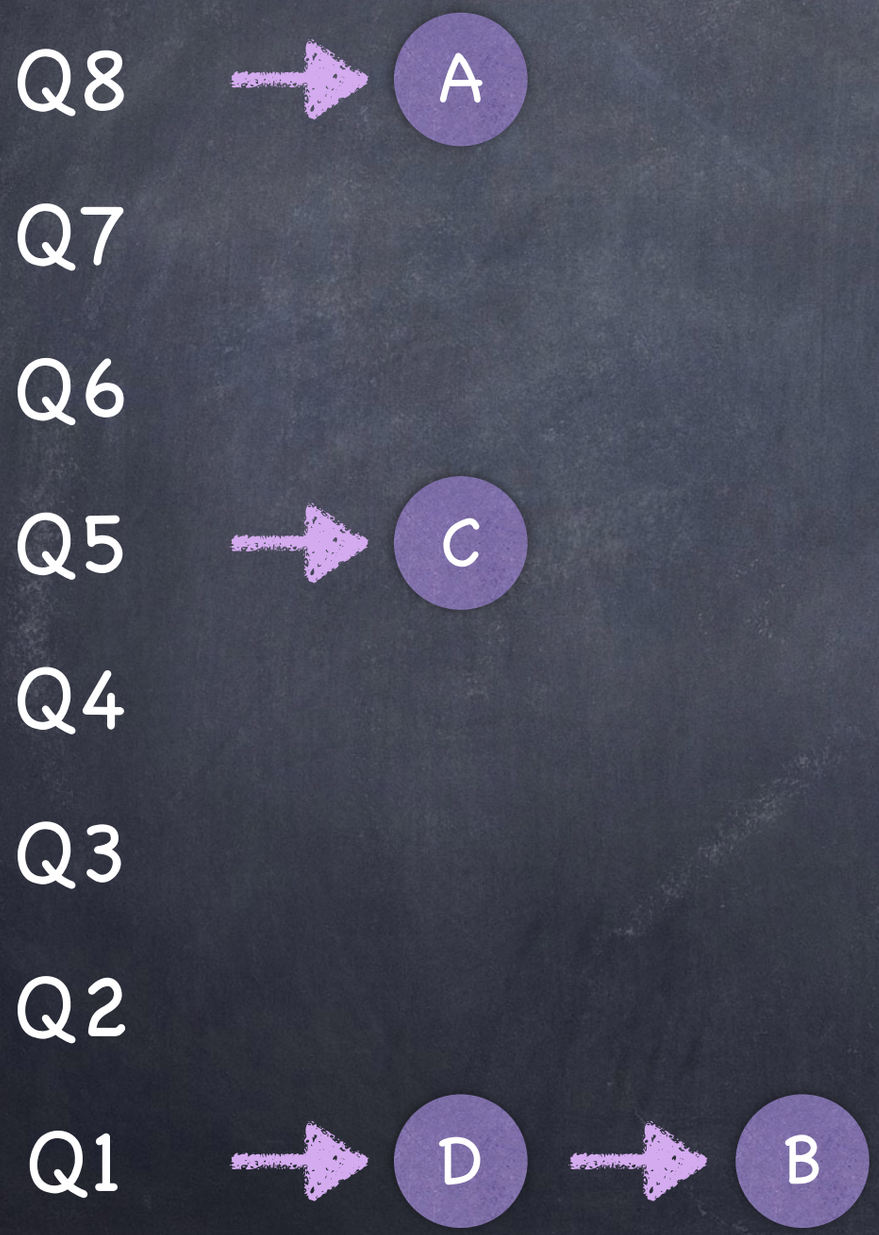
How are jobs assigned to a queue?

Moving down



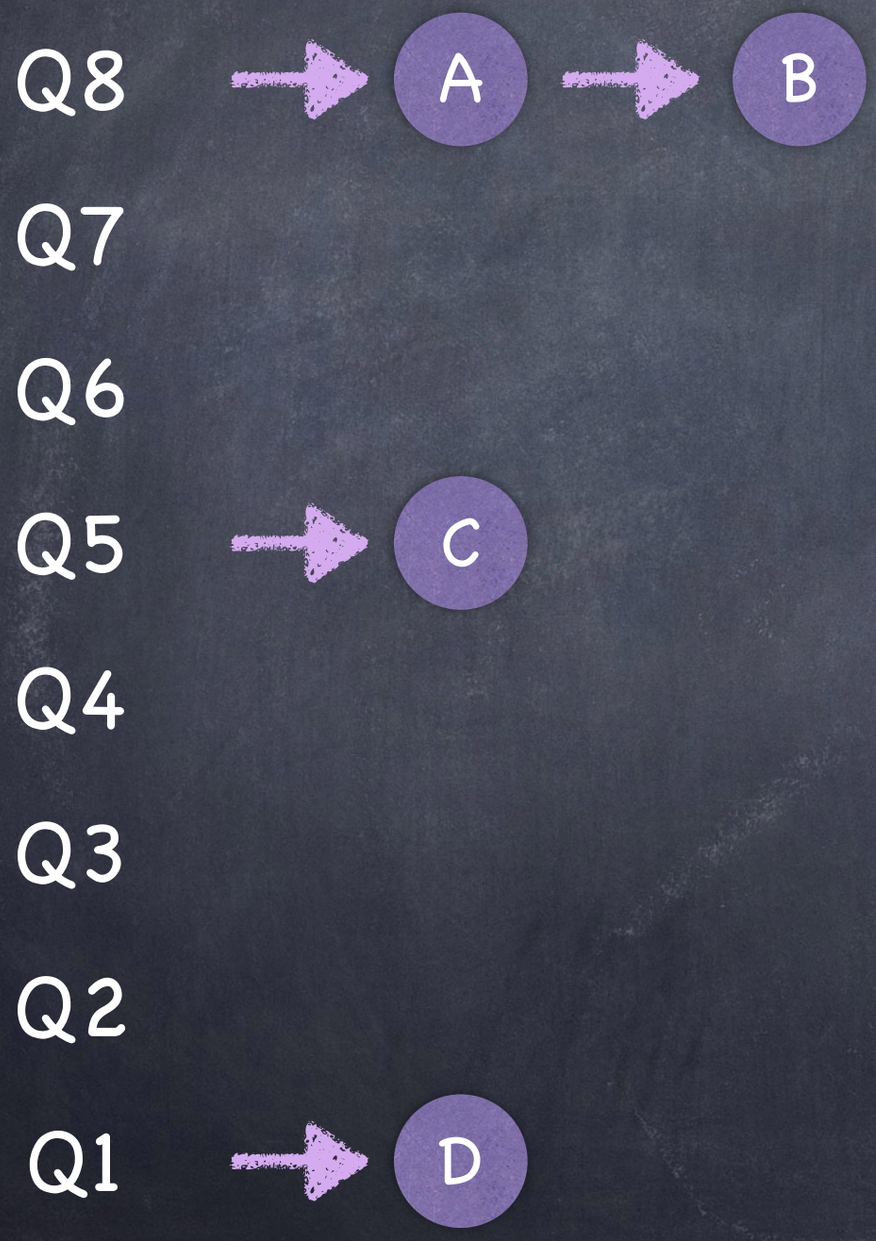
- Job starts at the top level
- If it uses full quantum before giving up CPU, moves down
- Otherwise, it stays where it is
- What about I/O?
  - Job with frequent I/O will not finish its quantum and stay at the same level
- Parameter
  - quantum size for each queue

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

Sneeeeeakyyyy...



- 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

The core problem!

- What we really wanted:
  - Efficiently handle **mix** of CPU-bound, I/O bound, interactive jobs, ..
  - What does "efficiently" really mean?
    - Give I/O bound jobs enough CPU to issue their next file op, and wait
    - Give interactive jobs enough CPU to respond to an input, and wait
    - Let CPU-bound jobs grind away without too much disturbance
- +Mechanism to avoid cheating

Let us design our own CPU scheduler

An ideal scheduler?

- Each thread gets an equal share of CPU
  - While ensuring that time-sensitive jobs are not blocked
- What is the "mechanism" we should use?
  - Priorities? Nah. We already saw issues.
  - Number of quantum used? Close, but can be cheated.
  - Why not directly track CPU time per thread?
- Scheduling decision
  - Among all "ready" threads
    - Choose the thread with minimum CPU time so far

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - Choose the thread with minimum CPU time so far
- Why may this work?
- I/O bound jobs: issue next file op, and wait
  - Blocked/sleeping threads don't advance their CPU time
  - When ready, get boosted!
- Interactive jobs: respond to an input, and wait
  - Blocked/sleeping threads don't advance their CPU time
  - When ready, get boosted!
- CPU-bound jobs: grind away all the remaining CPU cycles
  - While getting a fair allocation of CPU cycles
  - Cannot cheat!—kernel maintains CPU time for each job

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - Choose the thread with minimum CPU time so far

- But what if too many I/O bound and/or interactive jobs?
  - Starvation of CPU-bound jobs, or even priority inversion
  - How to avoid this?

- Idea 2: Introduce "target latency"
  - Period of time over which every thread should get some CPU cycles
  - Define quantum = target-latency/n
  - Every target-latency period,
    - Each thread gets at least a quantum worth of CPU time

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - If a thread has not been scheduled for target-latency time
      - Schedule it for a quantum worth of CPU time
        - Where quantum = target-latency/n
    - Else, choose the thread with minimum CPU time so far

- Problem?
  - Target latency = 20 ms, 200 threads
    - Each thread gets 0.1ms of CPU time
    - Large context switching overheads

- Idea 3: introduce a "minimum granularity"
  - Minimum time a thread must run, when scheduled

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - If a thread has not been scheduled for target-latency time
      - Schedule it for X worth of CPU time
      - Where X = maximum (quantum, min. granularity)
    - Else, choose the thread with minimum CPU time so far

- Problem?
  - Target latency = 20ms, minimum granularity = 1ms, 200 threads
    - Each thread gets 1ms worth of CPU time
    - But....
      - Some thread may have to wait for 20ms.
      - Back to being problematic for I/O and interactive jobs.

An ideal scheduler?

- We have been using priorities the wrong way all along
- We should use priorities to reflect "share" rather than preference
  - **nice** jobs: willing to give up for important jobs
  - **nice** values range from -20 to 19
  - If you are nice(r)—higher nice value—you will let important tasks run
- **Idea 3: Assign CPU cycles to threads using priorities as "weights"**
  - Each nice value is assigned a weight
    - Weight = $1024/(1.25)^{nice}$
  - Share of thread i
    - (its weight/(sum of all thread weights)) * target-latency

An ideal scheduler?

- Scheduling decision:
  - Among all "ready" threads
    - If a thread has not been scheduled for target-latency time
      - Schedule it for X worth of CPU time
      - Where X = maximum (thread's share, min. granularity)
      - Where thread's share depends on
        - thread's nice value & other threads' nice value
    - Else, choose the thread with minimum CPU time so far

- Would this work for all mix of jobs?
  - Let us see!

An ideal scheduler? Example 1

- Among all "ready" threads
  - If a thread has not been scheduled for target-latency time
    - Schedule it for X worth of CPU time
    - Where X = maximum (thread's share, min. granularity)
    - Where thread's share depends on
      - thread's nice value & other threads' nice value
    - Else, choose the thread with minimum CPU time so far

- Target latency = 20ms, Minimum granularity = 1ms
- Two CPU-bound jobs (nice = 20)
  - Each thread's share = (1/2)*20 = 10ms!
  - Each thread runs for 10ms, before the other gets CPU!

An ideal scheduler? Example 2

- Among all "ready" threads
  - If a thread has not been scheduled for target-latency time
    - Schedule it for X worth of CPU time
    - Where X = maximum (thread's share, min. granularity)
    - Where thread's share depends on
      - thread's nice value & other threads' nice value
    - Else, choose the thread with minimum CPU time so far

- Target latency = 20ms, Minimum granularity = 1ms
- A CPU-bound jobs (nice value = 20), and an I/O job (nice value = -19)
  - Thread shares will be: tiny (cpu-bound job), large (I/O job)
  - CPU-bound job can block I/O job for at most target-latency
  - I/O job will not block CPU-bound job—will go to sleep/block

An ideal scheduler?

- Among all "ready" threads
  - If a thread has not been scheduled for target-latency time
    - Schedule it for X worth of CPU time
    - Where X = maximum (thread's share, min. granularity)
    - Where thread's share depends on
      - thread's nice value & other threads' nice value
  - Else, choose the thread with minimum CPU time so far

- Very close to today's Linux CFS scheduler!
  - The only difference is Linux does scheduling on "virtual runtimes"
    - Rather than real CPU times (implementation issue)
  - Nicer job => lower weight => virtual runtime increases more quickly
  - Less Nicer job => higher weight => virtual runtime increases less quickly

We have a CPU scheduler!

- Designed by you!
  - In half a lecture …
  - Pretty close to ideal ….
  - Actually used by millions today …
- You now know CPU scheduling
- Network/Disk/… scheduling very similar

So, what does the OS really schedule?

- Most textbooks use the "old model"
  - One thread per process
  - Thus, misconception: OS schedules processes
- Usually, OS really schedules **threads**
- Switching overheads are low
  - Threads: save/restore registers
  - Processes: +the entire PCB (+active address space)