CSL373: Lecture 6
CPU Scheduling

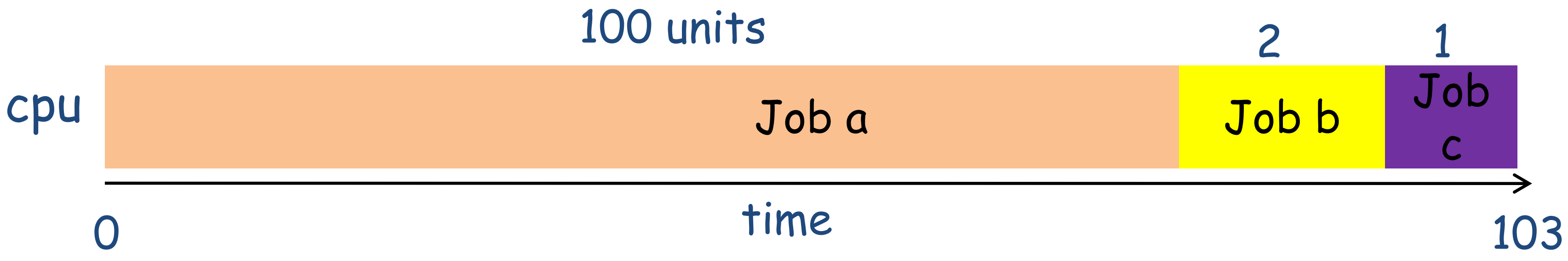
First come first served (FCFS or FIFO)

- Simplest scheduling algorithm

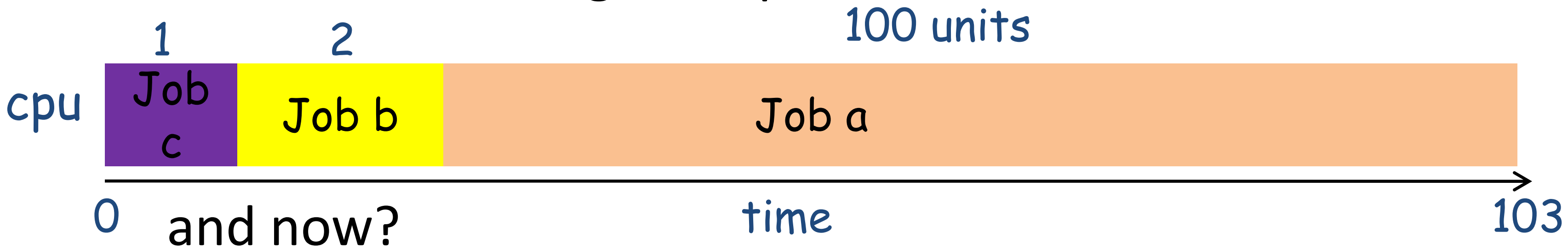
  Run jobs in order that they arrive

  Disadvantage: wait time depends on arrival order. Unfair to later jobs (worst case: long job arrives first)

  e.g.,: three jobs (A, B, C) arrive nearly simultaneously)



what's the average completion time?



and now?

FCFS and I/O utilization

- A CPU bound job will hold CPU until done, or it causes an I/O burst (rare occurrence, since the thread is CPU-bound) aka *convoy effect*
  - long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- Example: one CPU bound job, many I/O bound

  CPU bound runs (I/O devices idle)

  CPU bound blocks

  I/O bound job(s) run, quickly block on I/O

  CPU bound runs again

  I/O completes

  CPU bound still runs while I/O devices idle (continues…)

  Possible solution: run process whose I/O completed?

  Will it always work?

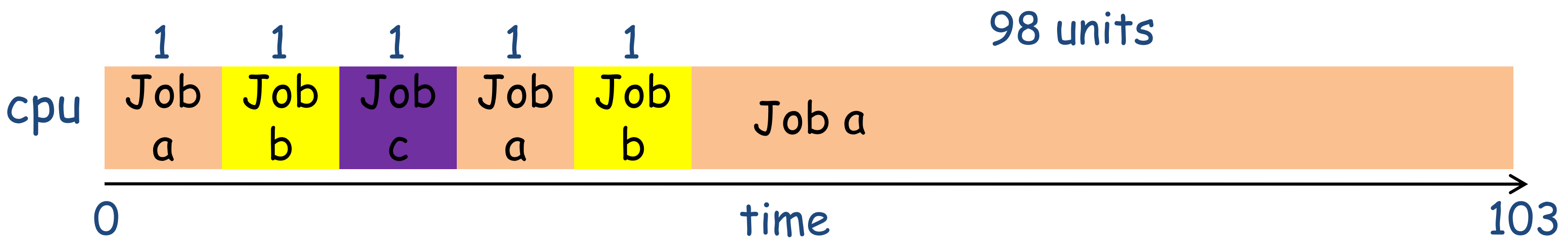
Round robin (RR)

- Solution to job monopolizing CPU? Interrupt it.

  Run job on some "time slice", when time is up, or it blocks, move it to back of a FIFO queue

  Most systems do some flavor of this

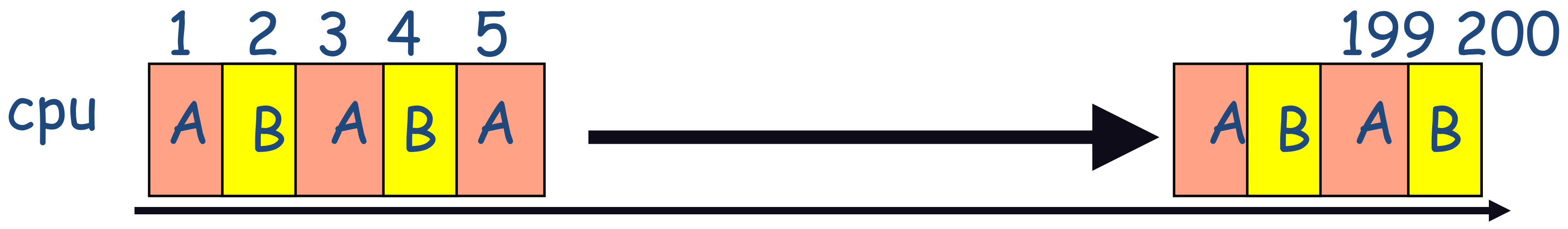
- Advantage:
  - fair allocation of CPU across jobs
  - low average waiting time when job lengths vary:



What is the avg completion time?

Round Robin's Big Disadvantage

- Varying sized jobs are good, but what about same-sized jobs? Assume 2 jobs of time=100 each:



Avg completion time?

How does this compare with FCFS for same two jobs?

RR Time slice tradeoffs

- Performance depends on length of the timeslice

  Context switching is not a free operation.

  If time slice is set too high (attempting to amortize context switch cost), you get FCFS. (i.e., processes will finish or block before their slice is up anyway)

  If it’s set too low, you’re spending all of your time context switching between threads.

  Timeslice frequently set to ≈100 milliseconds

  Context switches typically cost < 1 millisecond

  Moral: context switching is usually negligible (< 1% per timeslice in above example) unless you context switch too frequently and lose all productivity.

Priority scheduling

- Obvious: not all jobs equal

  So: rank them.

- Each process has a priority

  Run highest priority ready job in system round robin among processes of equal priority

  Priorities can be static or dynamic (Or both: Unix)

  Most systems use some variant of this

- Common use: couple priority to job characteristic

  Fight starvation? Increase priority as time spent in ready queue

  Keep I/O busy? Increase priority for jobs that often block on I/O

- Priorities can create deadlock.

  Fact: high priority always runs over low priority

  So?

Handling thread dependencies

- Priority inversion e.g., T1 at high priority, T2 at low
  - T2 acquires lock L
  - Scene 1: T1 tries to acquire L, fails, spins. T2 never gets to run
  - Scene 2: T1 tries to acquire L, fails, blocks. T3 enters system a medium priority. T2 never gets to run.
- Scheduling = deciding who should make progress
  - Obvious: a thread's importance should increase with the importance of those that depend on it.
  - Naïve priority schemes violate this
- "Priority donation"
  - Thread's priority scales with priority of dependent threads

Shortest time to completion first (STCF)

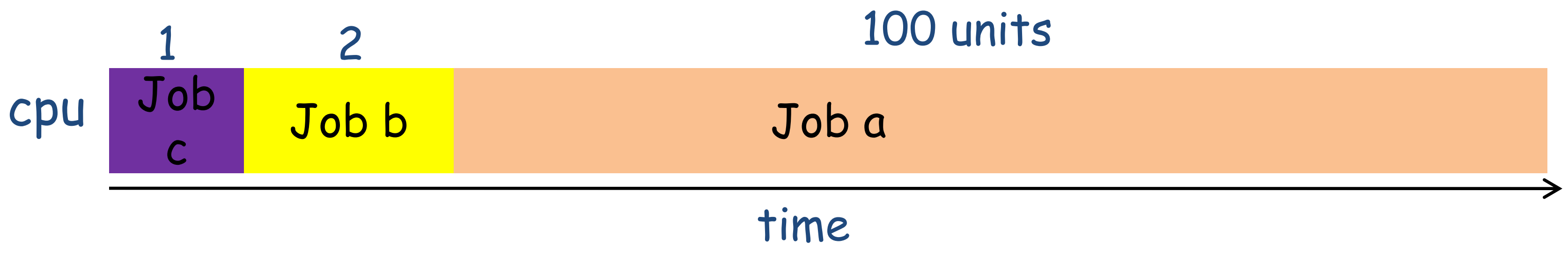
- STCF (or shortest-job-first)

  run whatever job has least amount of stuff to do.

  can be pre-emptive or non-preemptive.

- Example: same jobs (given jobs A, B, C)

  Average completion = (1 + 3 + 103)/3 ≈ 35 (vs ≈100 for FCFS)



- Provable optimal: moving shorter job before longer job improves waiting time for short job more than harms the waiting time for long job. Try the proof yourself.

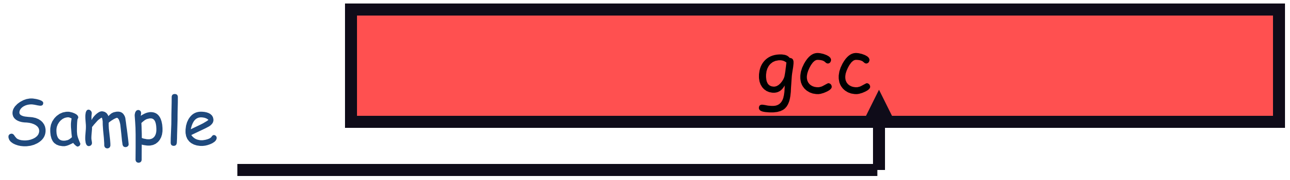
How to know job length?

- Have user tell us. If they lie, kill the job

  Not so useful in practice

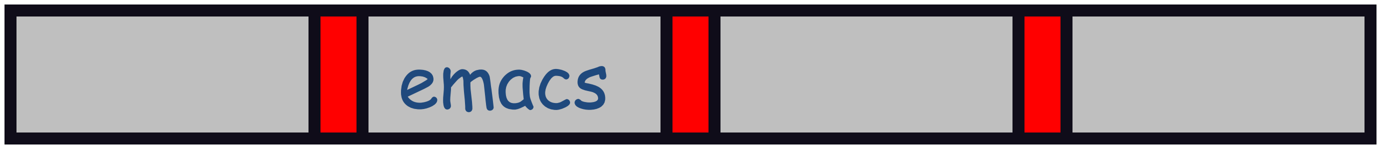
- Use the past to predict the future #1:

  Long running job will probably take a long time more



- Use the past to predict the future #2:

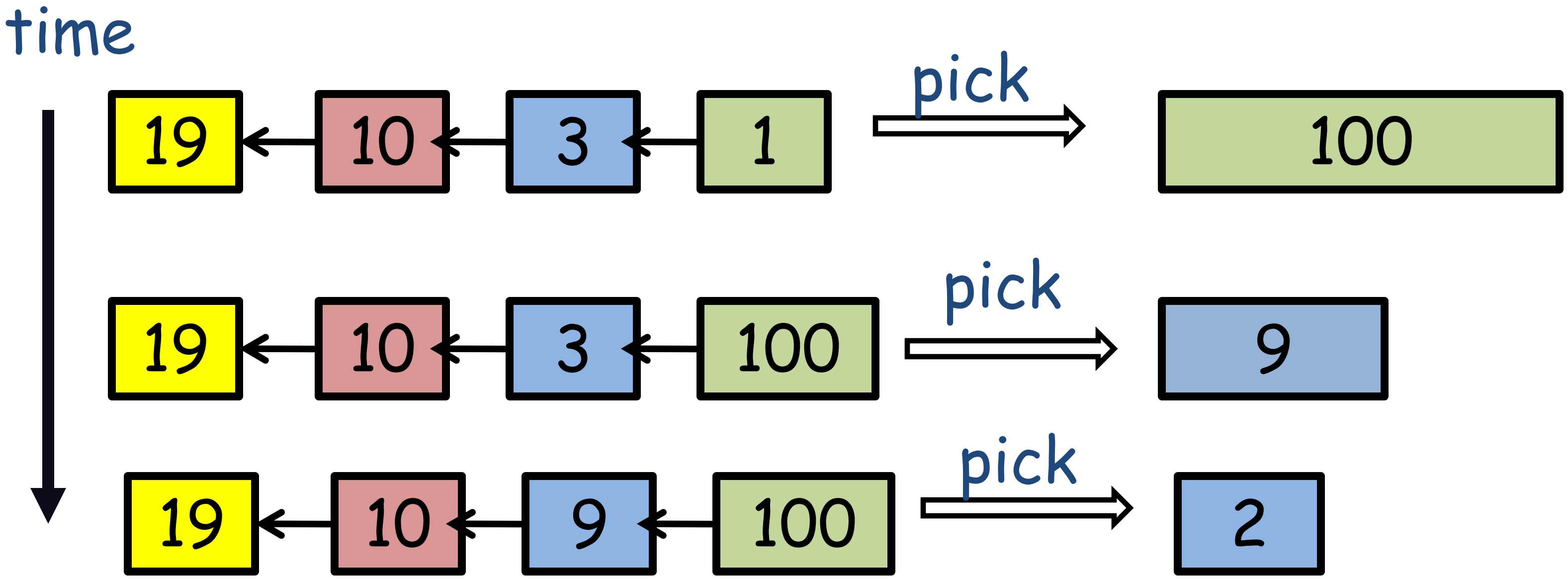
  View job as sequence of sequentially alternating CPU and I/O jobs



If previous CPU jobs in the sequence have run quickly, future ones will too (“usually”)

Approximate STCF

- ~STCF: predict length of current CPU burst using length of previous burst
  - Record length of previous burst (0 when just created)
  - At scheduling event (unblock, block, exit, …) pick smallest "past run length" off of ready queue.

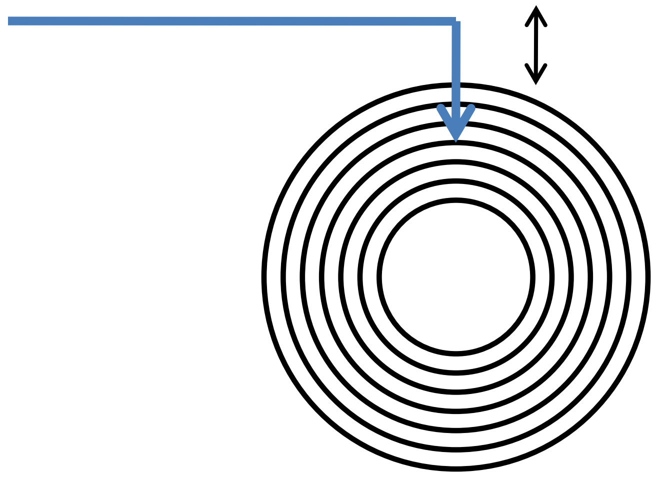


Elevator in Bharti Bldg.

- To choose direction:
  - Uses FCFS
- In each direction:
  - Follows STCF

Disk drive head

- A disk drive receives many r/w requests for different sectors simultaneously.
- Disk organized as concentric circles (called cylinders).
- The disk rotates around the center
- The disk head positions itself appropriately to read the requested sector. This positioning is also called “disk seek” and the time taken, “seek time”



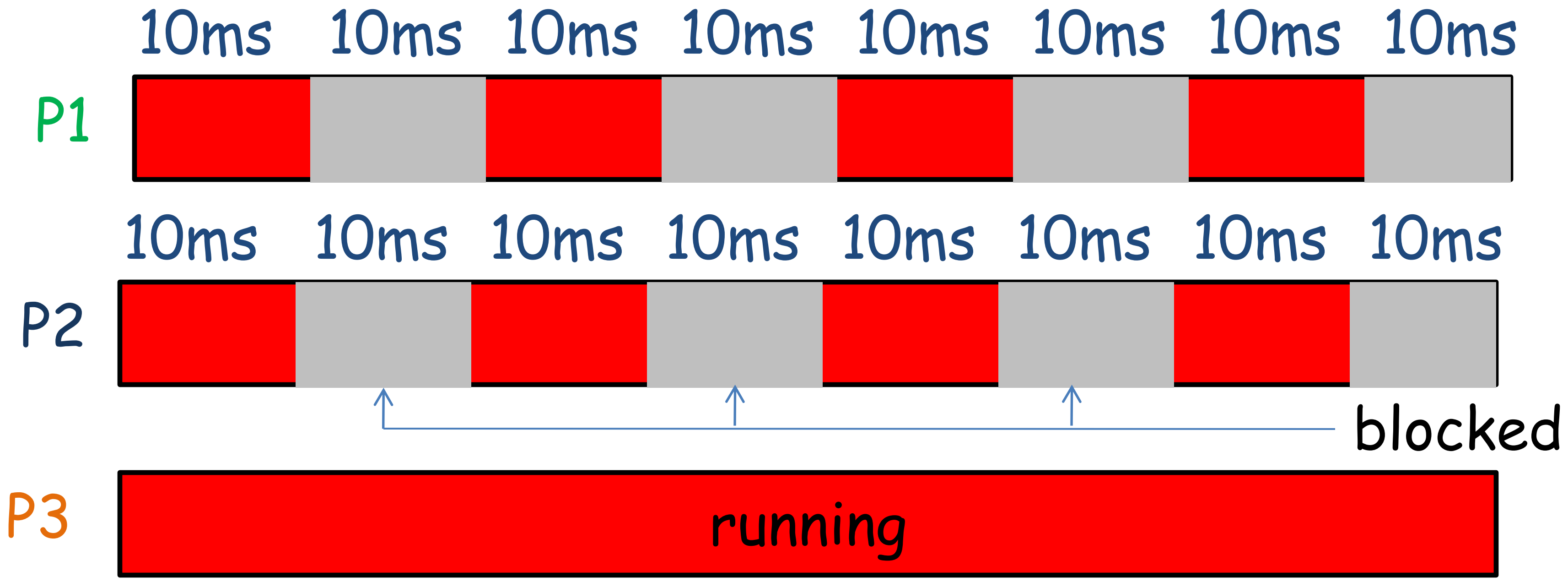
Requested sectors:
231, 245, 636, 354

Disk drive (STCF in action)

- Disk can predict length of next “job”!
  - Job = request to disk
  - Job length ≈ cost of moving disk arm to position of the requested disk block. (Farther away = more costly.)
- STCF for disks: shortest-seek-time-first (SSTF)
    - Do read/write request closest to current position
    - Preemptive: if new jobs arrive that can be serviced on the way, do these too.
    - However, do not change direction (just like an elevator). Hence, also called “elevator algorithm”
- Elevator algorithm:
  - Disk arm has direction, do closest request in that direction. Sweeps from one end to other

~STCF vs RR

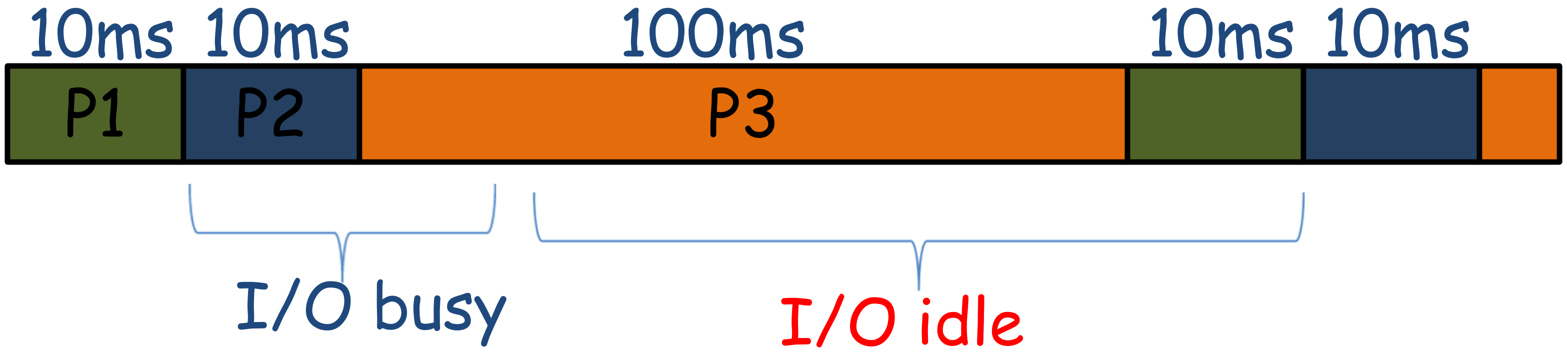
- Three processes P1, P2, P3


10ms 10ms 10ms 10ms 10ms 10ms 10ms 10ms
P1
10ms 10ms 10ms 10ms 10ms 10ms 10ms 10ms
P2
blocked
P3
running

  - 100 ms time slice.

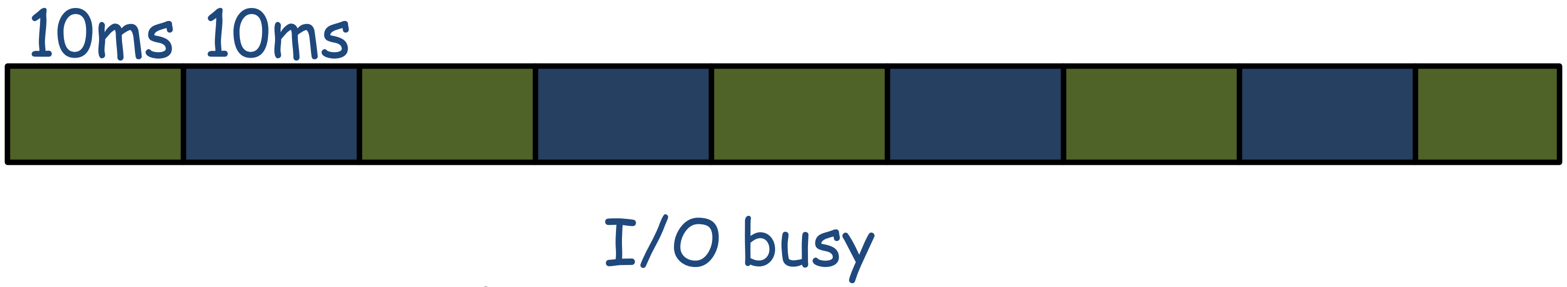
~STCF vs RR

- RR:



Problem: Long periods of idle I/O

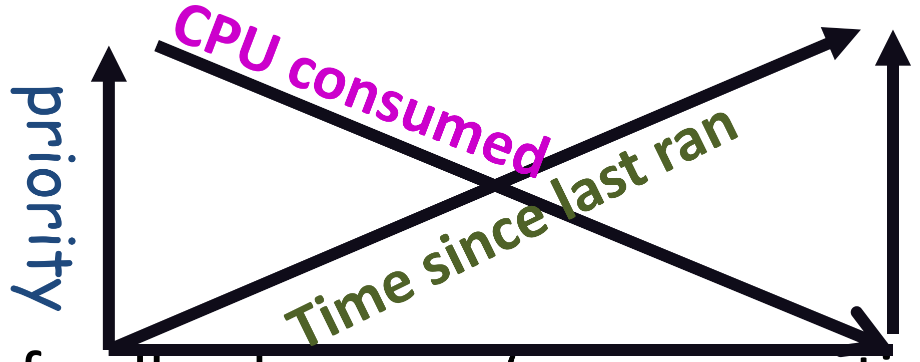
- ~STCF



Problem: Full I/O utilization, but P3 gets starved!

Generalizing: priorities + history

- ~STCF good core idea but doesn't have enough state
  - The usual STCF problem: starvation
  - Solution: compute priority as a function of both CPU time P has consumed and time since P last ran



- Multi-level feedback queue (or exponential Q)
  - Priority scheme where adjust priorities to penalize CPU intensive programs and favor I/O intensive
  - Pioneered by CTSS (MIT in 1962)
  - Implemented by you (or should have been)

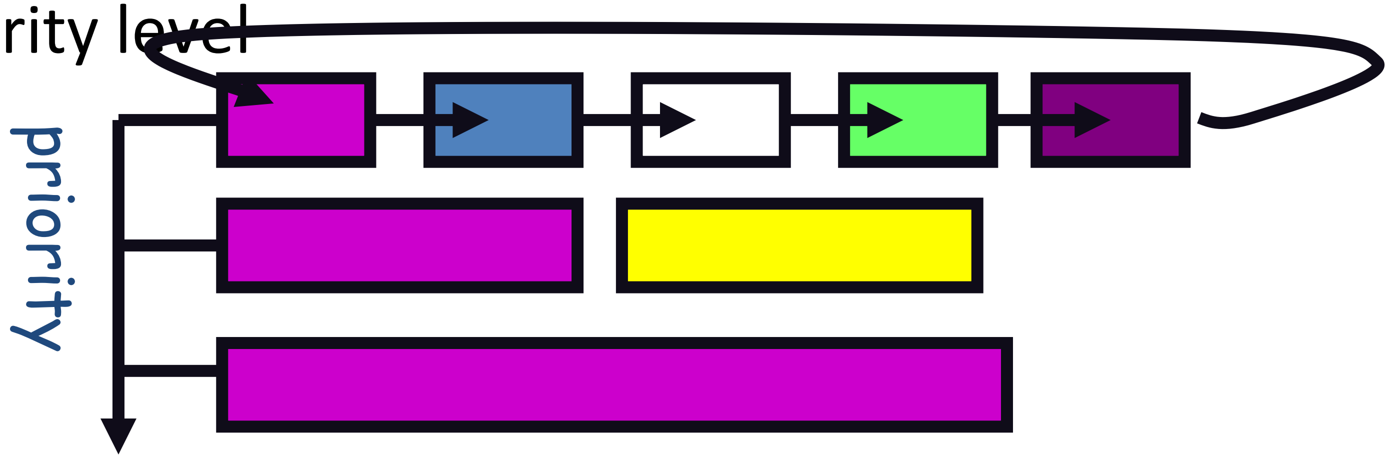
A simple multi-level feedback queue

- Attacks both efficiency and response time problems

    Efficiency: long time quanta = low switching overhead

    Response time: quickly run after becoming unblocked

- Priority queue organization: one ready queue for each priority level



process created: give high priority and short time slice

if process uses up the time slice without blocking:

priority = priority – 1;   time_slice = time_slice*2

Some problems

- Can't low priority threads starve?
    - Ad hoc: when skipped over, increase priority

- What about when past doesn't predict future?
    - e.g., CPU bound switches to I/O bound
        - Want past predictions to "age" and count less towards current view of the world.

Summary

- FIFO
  - + simple
  - - short jobs can get stuck behind long ones; poor I/O
- RR
  - + better for short jobs; fair
  - - poor when jobs are the same length; I/O utilization not optimal
- STCF
  - + optimal (avg. response time, avg. time-to-completion)
  - - hard to predict future (hence, use ~STCF)
  - - Possibility of starvation
- Multi-level feedback
  - + ~STCF
  - - unfair to long running jobs