CSE 451: Operating Systems
Winter 2026

Module 9
Scheduling

Gary Kimura

Scheduling

- In discussing processes and threads, we talked about context switching
  - an interrupt occurs (device completion, timer interrupt)
  - a thread causes a trap or exception
  - may need to choose a different thread/process to run
- We glossed over the choice of which process or thread is chosen to be run next
  - "some thread from the ready queue"
  - We'll limit our discussion to scheduling Kernel Threads
- This decision is called scheduling
    - scheduling is a policy
    - context switching is a mechanism

Classes of Schedulers

- Batch
  - Throughput / utilization oriented
  - Example: audit inter-bank funds transfers each night, Pixar rendering, Hadoop/MapReduce jobs
- Interactive
  - Response time oriented
  - Example: `attu.cs`
- Real time
  - Deadline driven
  - Example: embedded systems (cars, airplanes, etc.)
- Parallel
  - Speedup-driven
  - Example: “space-shared” use of a 1000-processor machine for large simulations

We’ll be talking primarily about interactive schedulers

Multiple levels of scheduling decisions

- Long term
  - Should a new “job” be “initiated,” or should it be held?
    - typical of batch systems
    - what might cause you to make a “hold” decision?
- Medium term
  - Should a running program be temporarily marked as non-runnable (e.g., swapped out)?
- Short term
  - Which thread should be given the CPU next? For how long?
  - Which I/O operation should be sent to the disk next?
  - On a multiprocessor:
    - should we attempt to coordinate the running of threads from the same address space in some way?
    - should we worry about cache state (processor affinity)?

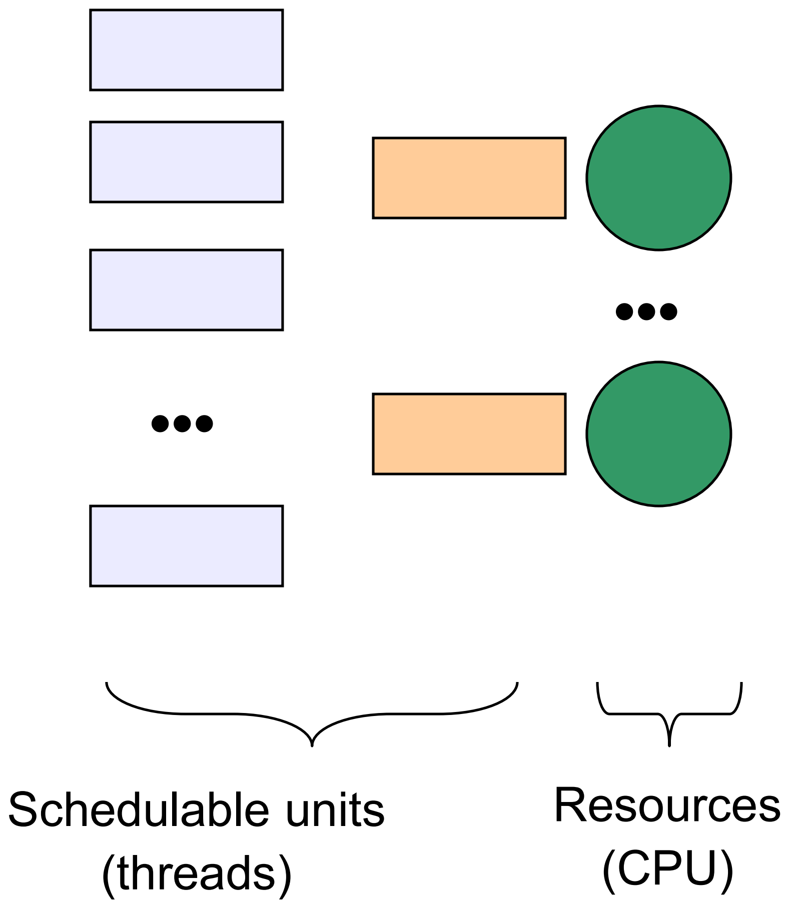
Scheduling Goals I: Performance

- Many possible metrics / performance goals (which sometimes conflict)
  - maximize CPU utilization
  - maximize throughput (`requests completed / s`)
  - minimize average response time (`average time from submission of request to completion of response`)
  - minimize average waiting time (`average time from submission of request to start of execution`)
  - minimize energy (`joules per instruction`) subject to some constraint (`e.g., frames/second`)

Scheduling Goals II: Fairness

- No single, compelling definition of “fair”
  - How to measure fairness?
    - Equal CPU consumption? (over what time scale?)
  - Fair per-user? per-process? per-thread?
  - What if one process is CPU bound and one is I/O bound?
- Sometimes the goal is to be unfair:
  - Explicitly favor some particular class of requests (priority system), but…
  - avoid starvation (be sure everyone gets at least some service)

The basic situation



Scheduling:

- Who to assign each resource to
- When to re-evaluate your decisions

When to assign?

- Pre-emptive vs. non-preemptive schedulers
  - Non-preemptive
    - once you give somebody the green light, they've got it until they relinquish it
      - an I/O operation
      - allocation of memory in a system without swapping
  - Preemptive
    - you can re-visit a decision
      - setting the timer allows you to preempt the CPU from a thread even if it doesn't relinquish it voluntarily
      - in any modern system, if you mark a program as non-runnable, its memory resources will eventually be re-allocated to others
    - Re-assignment always involves some overhead
      - Overhead doesn't contribute to the goal of any scheduler

- We'll assume "work conserving" policies
  - Never leave a resource idle when someone wants it
    - Why even mention this? When might it be useful to do something else? The disparate speed between CPU and Storage highlight this point

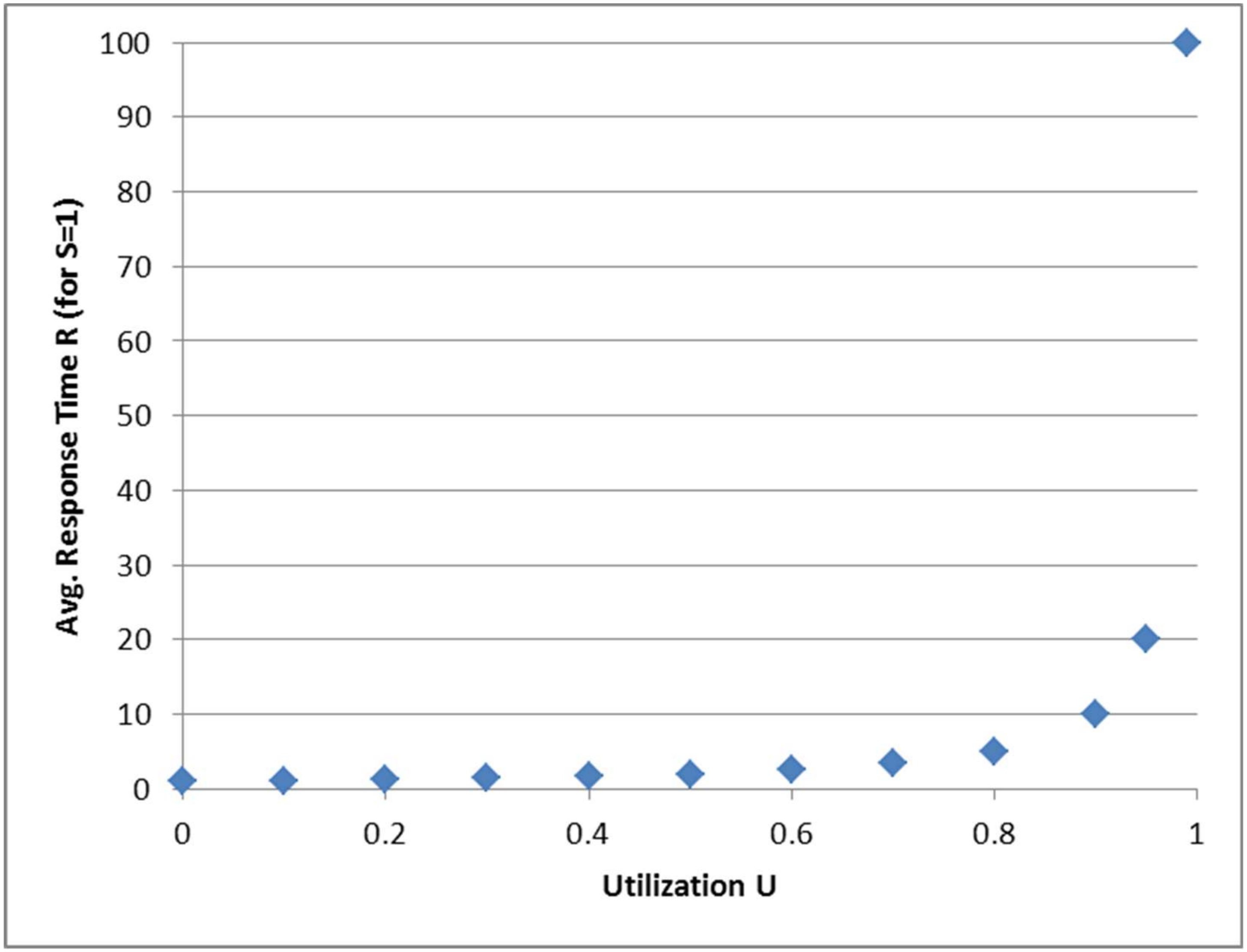
Before we look at specific policies

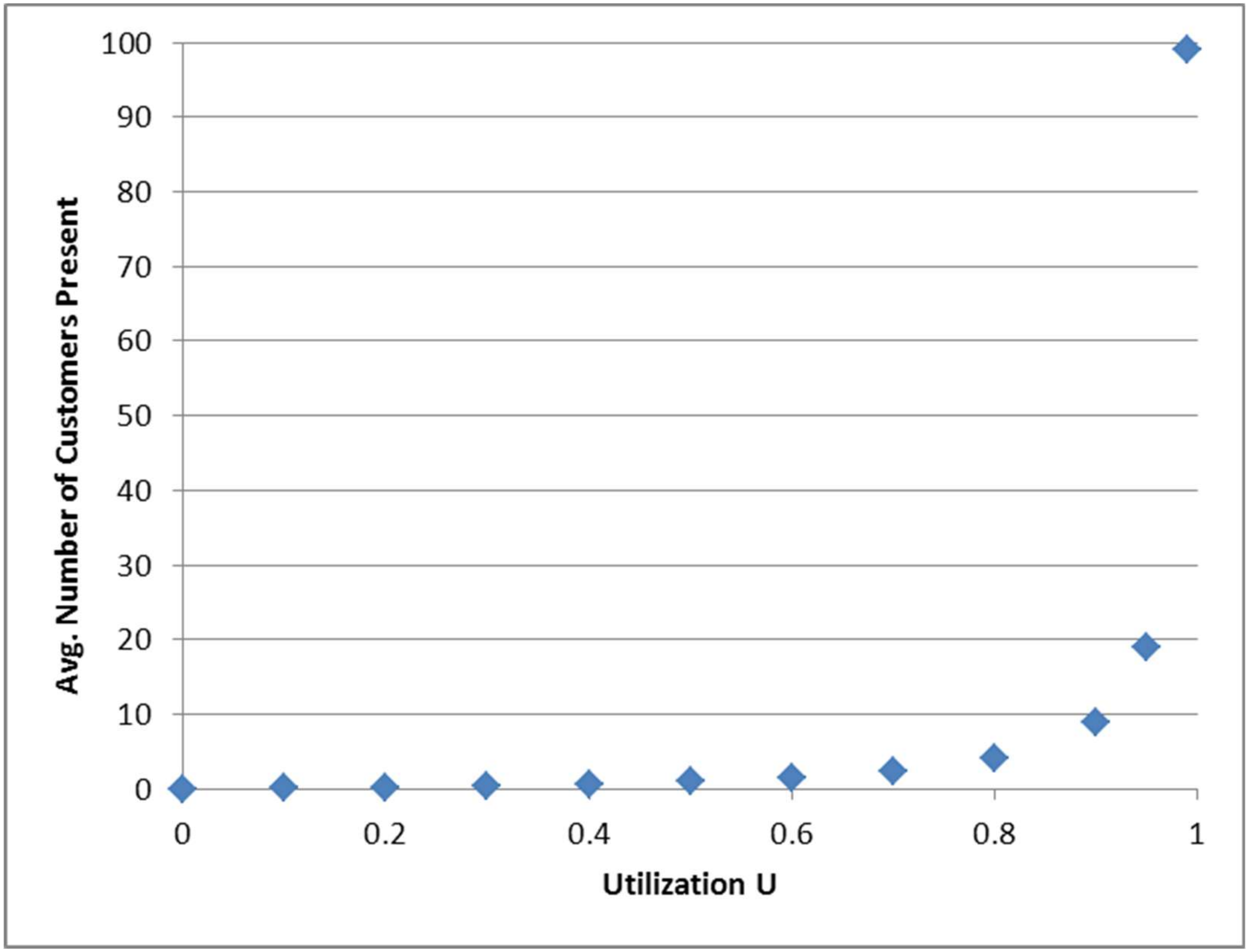
- There are some simple but useful “laws” to know about …
- The Utilization Law: U = X * S
  - Where U is utilization, X is throughput (requests per second), and S is average service requirement
    - Obviously true
    - This means that utilization is constant, independent of the schedule, so long as the workload can be processed

- Little's Law: $N = X * R$
  - Where N is average number in system, X is throughput, and R is average response time (average time in system)
    - This means that better average response time implies fewer in system, and vice versa
  - Proof:
    - Let W denote the total time-in-system accumulated by all customers during a time interval of length T
    - The average number of requests in the system $N = W / T$
    - If C customers complete during that time period, then the average contribution of each completing request $R = W / C$
    - Algebraically, $W/T = C/T * W/C$.
    - Thus, $N = X * R$

(Not quite a law – requires some assumptions)

- Response Time at a single server under FCFS scheduling: R = S / (1-U)
  - Clearly, when a customer arrives, her response time will be the service time of everyone ahead of her in line, plus her own service time: R = S * (1+A)
    - Assumes everyone has the same average service time
  - Assume that the number you see ahead of you at your instant of arrival is the long-term average number in line; so R = S * (1+N)
  - By Little's Law, N = X * R
  - So R = S * (1 + X*R) = S + S*X*R = S / (1 – X*S)
  - By the Utilization Law, U = X*S
  - So R = S / (1-U)
  - And since N = X*R, N = U / (1-U)


Avg. Response Time R (for S=1)
100
90
80
70
60
50
40
30
20
10
0
0
0.2
0.4
0.6
0.8
1
Utilization U


Avg. Number of Customers Present
100
90
80
70
60
50
40
30
20
10
0
0
0.2
0.4
0.6
0.8
1
Utilization U

- Kleinrock's Conservation Law for priority scheduling:

  $\sum_p U_p * R_p = \text{constant}$

  – Where U_p is the utilization by priority level p and R_p is the time in system of priority level p

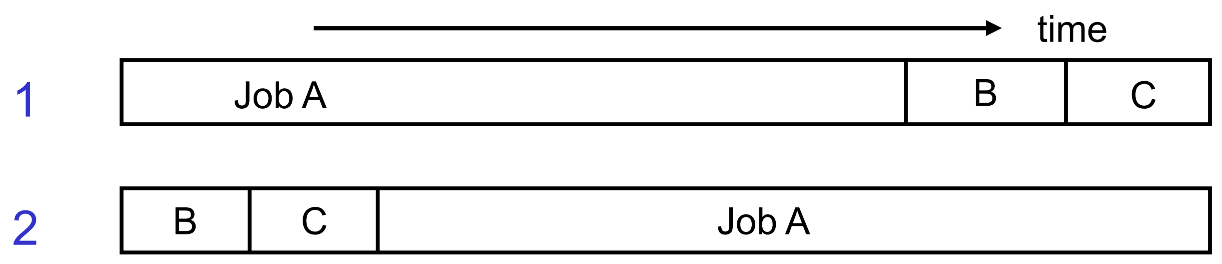
    - This means you can't improve the response time of one class of task by increasing its priority, without hurting the response time of at least one other class

Algorithm #1: FCFS/FIFO

- First-come first-served / First-in first-out (FCFS/FIFO)
  - schedule in the order that they arrive
  - “real-world” scheduling of people in (single) lines
    - supermarkets, McD’s, Starbucks …
  - jobs treated equally, no starvation
    - In what sense is this “fair”?

- Sounds perfect!
  - in the real world, when does FCFS/FIFO work well?
    - even then, what’s it’s limitation?
  - and when does it work badly?

FCFS/FIFO example



- Suppose the duration of A is 5, and the durations of B and C are each 1
  - average response time for schedule 1 (assuming A, B, and C all arrive at about time 0) is (5+6+7)/3 = 18/3 = 6
  - average response time for schedule 2 is (1+2+7)/3 = 10/3 = 3.3
  - consider also “elongation factor” – a “perceptual” measure:
    - Schedule 1: A is 5/5, B is 6/1, C is 7/1 (worst is 7, ave is 4.7)
    - Schedule 2: A is 7/5, B is 1/1, C is 2/1 (worst is 2, ave is 1.5)

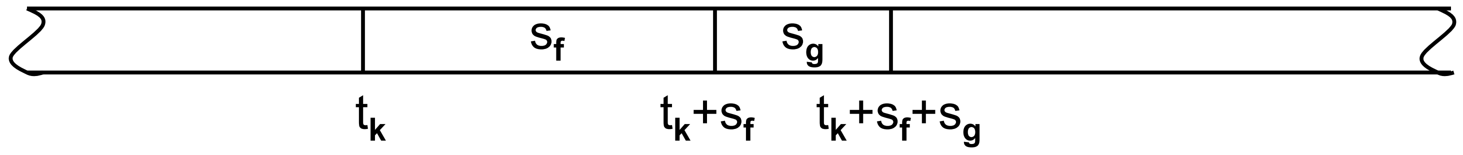
FCFS/FIFO drawbacks

- Average response time can be lousy
  - small requests wait behind big ones
- May lead to poor utilization of other resources
  - if you send me on my way, I can go keep another resource busy
  - FCFS may result in poor overlap of CPU and I/O activity
    - E.g., a CPU-intensive job prevents an I/O-intensive job from doing a small bit of computation, thus preventing it from going back and keeping the I/O subsystem busy
- Note: The more copies of the resource there are to be scheduled, the less dramatic the impact of occasional very large jobs (so long as there is a single waiting line)
  - E.g., many cores vs. one core

Algorithm #2: SPT/SJF

- Shortest processing time first / Shortest job first (SPT/SJF)
  - choose the request with the smallest service requirement
- *Provably optimal* with respect to average response time
  - Why do we care about “provably optimal”?

SPT/SJF optimality – The interchange argument



- In any schedule that is not SPT/SJF, there is some adjacent pair of requests f and g where the service time (duration) of f, s_f, exceeds that of g, s_g
- The total contribution to average response time of f and g is $2t_k+2s_f+s_g$
- If you interchange f and g, their total contribution will be $2t_k+2s_g+s_f$, which is smaller because $s_g < s_f$
- If the variability among request durations is zero, how does FCFS compare to SPT for average response time?

SPT/SJF drawbacks

- It’s non-preemptive
  - So?
- … but there’s a preemptive version – SRPT (Shortest Remaining Processing Time first) – that accommodates arrivals (rather than assuming all requests are initially available)

- Sounds perfect!
  - what about starvation?
  - can you know the processing time of a request?
  - can you guess/approximate? How?

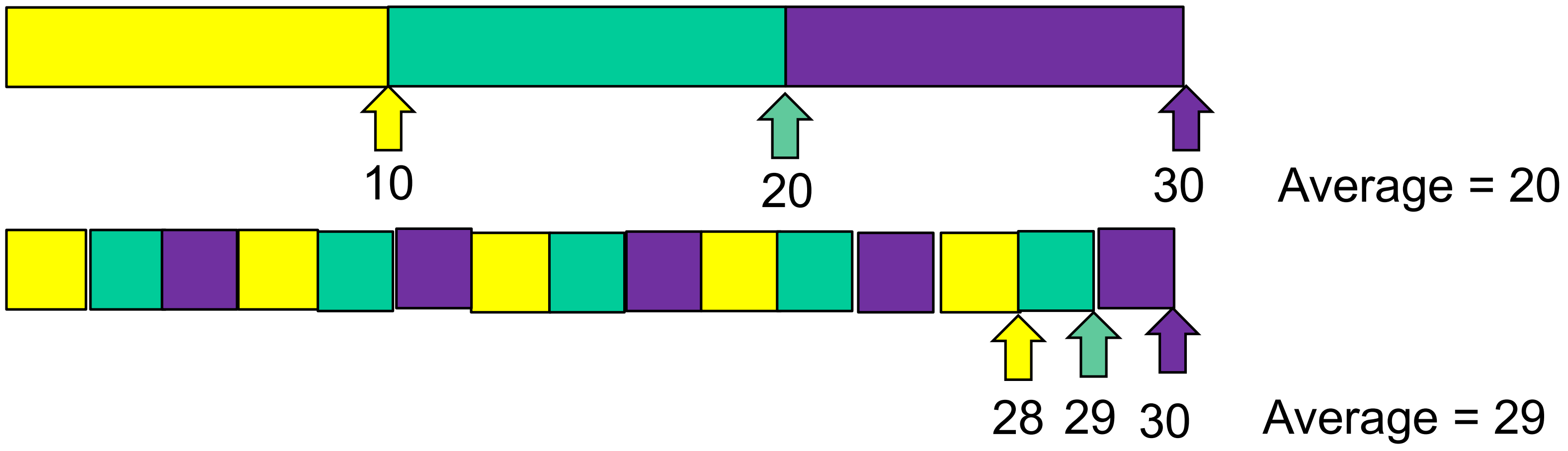
Algorithm #3: RR

- Round Robin scheduling (RR)
  - Use preemption to offset lack of information about execution times
    - I don't know which one should run first, so let's run them all!
  - ready queue is treated as a circular FIFO queue
  - each request is given a time slice, called a quantum
    - request executes for duration of quantum, or until it blocks
      - what signifies the end of a quantum?
    - time-division multiplexing (time-slicing)
  - great for timesharing
    - no starvation

- Sounds perfect!
  - how is RR an improvement over FCFS?
  - how is RR an improvement over SPT?
  - how is RR an approximation to SPT?

RR drawbacks

- What if all jobs are exactly the same length?
  - What would the schedule be (with average response time as the measure)?



- What do you set the quantum to be?
  - no value is “correct”
    - if small, then context switch often, incurring high overhead
    - if large, then response time degrades

Algorithm #4: Priority

- Assign priorities to requests
  - choose request with highest priority to run next
    - if tie, use another scheduling algorithm to break (e.g., RR)
  - Goal: non-fairness (favor one group over another)
- Abstractly modeled (and usually implemented) as multiple “priority queues”
  - put a ready request on the queue associated with its priority
- Sounds perfect!

Priority drawbacks

- How are you going to assign priorities?

- Starvation
  - if there is an endless supply of high priority jobs, no low-priority job will ever run

- Inversion (really bad starvation)
  - Assume three threads H(igh), M(edium), and L(ow) with priorities
  - Low runs and acquires a resource
  - High preempts Low and blocks on that resource
  - Medium becomes runnable and is CPU-bound
  - Low can’t finish, and High is out of luck

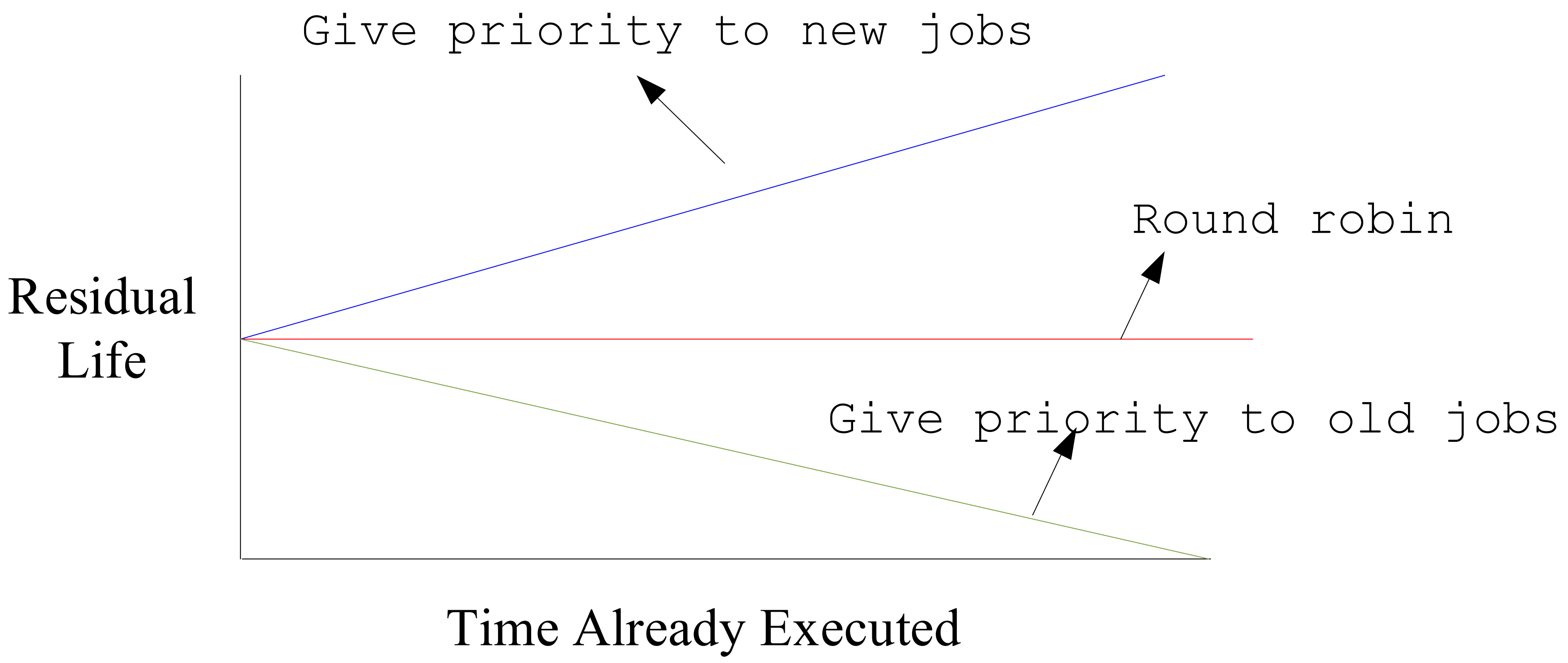
Program behavior and scheduling

- An analogy:
  - Say you're at a bank
  - There are two “identical” tellers:
    - Teller 1 has 3 people in line
    - Teller 2 has 6 people in line
  - You get into the line for Teller 1
  - Teller 2's line shrinks to 4 people

  - Why might you now switch lines, preferring 5th in line for Teller 2 over 4th in line for Teller 1?

Residual Life

- Given that a job has already executed for X seconds, how much longer will it execute, on average, before completing?



History DOES matter (or how we can estimate the future)

- It's been observed that workloads tend to have increasing residual life
  - "if you don't finish quickly, you're probably a lifer"
  - "you did it before so you're likely to do it again"
- This is exploited in practice by using a policy that discriminates against the old (not really ageism, but...)
- Which leads us to MLFQ

Quick Scheduling Recap

- More than just the CPU, but for our purpose we’ve limited our discussion to the CPU

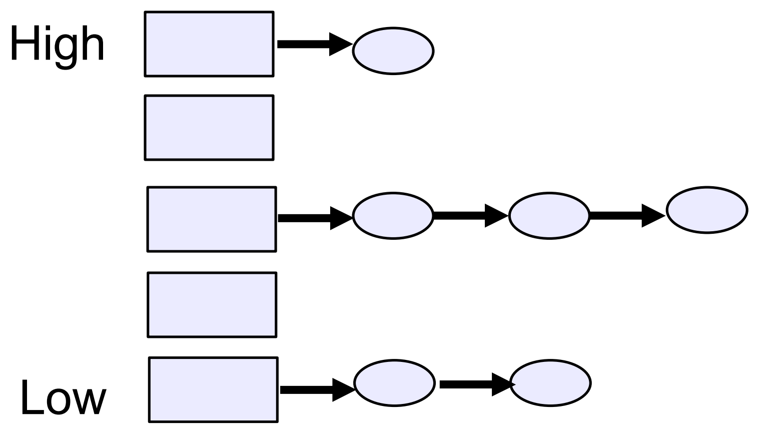
- FCFS/FIFO (First Come First Serve/First In First Out)
- SPT/SJF (Shortest Processing Time/Shortest Job First)
- RR (Round Robin)
- Priority
- Finally MLFQ (Multi-Level Feedback Queues)

- One more observation, Scheduling is more art than science.

Multi-level Feedback Queues (MLFQ)

- MLFQ:
  - there is a hierarchy of queues based on priority
  - new requests enter the highest priority queue
  - each queue is scheduled RR
  - requests move between queues based on execution history
  - lower priority queues may have longer quanta
- “Age” threads over time (feedback)
  - increase priority as a function of accumulated wait time
  - decrease priority as a function of accumulated processing time
  - many heuristics have been explored in this space. All are ugly

Illustration



Some Basic Concept

- Base Priority
- Current Priority
- Priority boost

Little Tidbits (what to do with…)

- Compute bound threads
- I/O intense threads
- How to handle processes with little or a lot of threads
- Processor affinity
- The idle thread

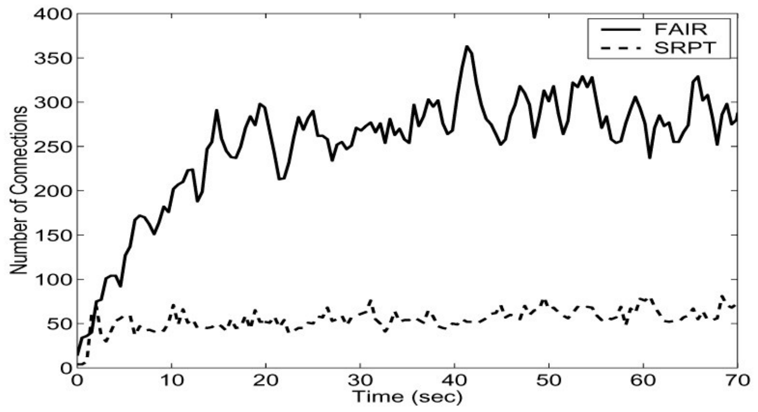
UNIX scheduling

- Canonical scheduler is pretty much MLFQ
  - 3-4 classes spanning ~170 priority levels
    - timesharing: lowest 60 priorities
    - system: middle 40 priorities
    - real-time: highest 60 priorities
  - priority scheduling across queues, RR within
    - process with highest priority always run first
    - processes with same priority scheduled RR
  - processes dynamically change priority
    - increases over time if process blocks before end of quantum
    - decreases if process uses entire quantum
- Goals:
  - reward interactive behavior over CPU hogs
    - interactive jobs typically have short bursts of CPU

Scheduling the Apache web server SRPT

- What does a web request consist of? (What's it trying to get done?)
- How are incoming web requests scheduled, in practice?
- How might you estimate the service time of an incoming request?
- Starvation under SRPT is a problem in theory – is it a problem in practice?
  - "Kleinrock's conservation law"

(Work by Bianca Schroeder and Mor Harchol-Balter at CMU)



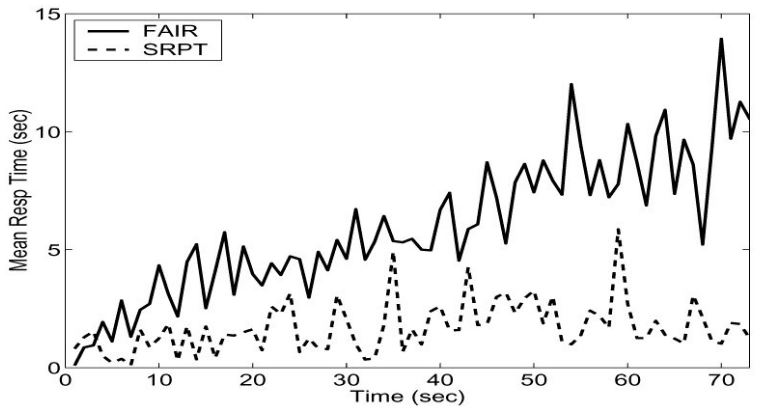
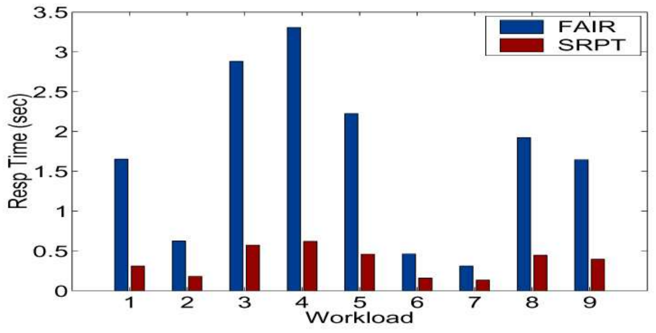
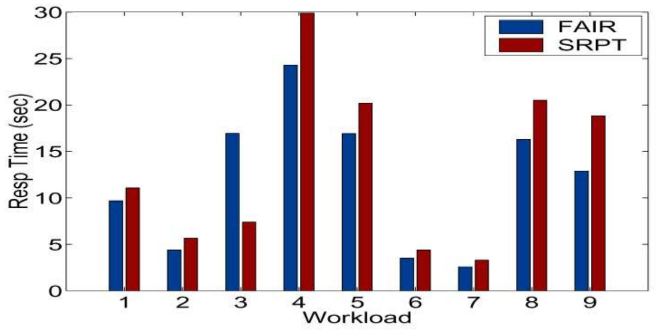


Figure 5: *Results for a persistent overload of 1.2. (Left) Buildup in connections at the server. (Right) response times.*



Mean Response Time - Baseline



Mean Response Time of biggest 1% requests - Baseline

Figure 13: *Comparison of FAIR and SRPT in the baseline case for the workloads in Table 1, under the NASA trace log. The mean response times over all requests are shown left, and the mean response times of only the biggest 1% of all requests are shown right.*

Summary

- Scheduling takes place at many levels
- It can make a huge difference in performance
  - this difference increases with the variability in service requirements
- Multiple goals, sometimes conflicting
- There are many "pure" algorithms, most with some drawbacks in practice – FCFS, SPT, RR, Priority
- Real systems use hybrids (aka ugly heuristics) that exploit observed program behavior
- Scheduling is still important, and there are still new angles to be explored – particularly in large-scale datacenters for reasons of cost and energy