# CSE 451: Operating Systems
# Spring 2011

## Module 5
## Scheduling

John Zahorjan
zahorjan@cs.washington.edu
Allen Center 534

# Scheduling

- In discussing processes and threads, we talked about context switching
  - an interrupt occurs (device completion, timer interrupt)
  - a thread causes an exception (a *trap* or a *fault*)
- We glossed over the choice of which process or thread is chosen to be run next
  - "some thread from the ready queue"
- This decision is called scheduling
  - scheduling is policy
  - context switching is mechanism

# Classes of Schedulers

- Batch
  - Throughput / utilization oriented
  - Example: audit inter-bank funds transfers each night
- Interactive
  - Response time oriented
  - Example: `attu`
- 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
- Others…

We’ll be talking primarily about interactive scheduers (as does the text).

# 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)?

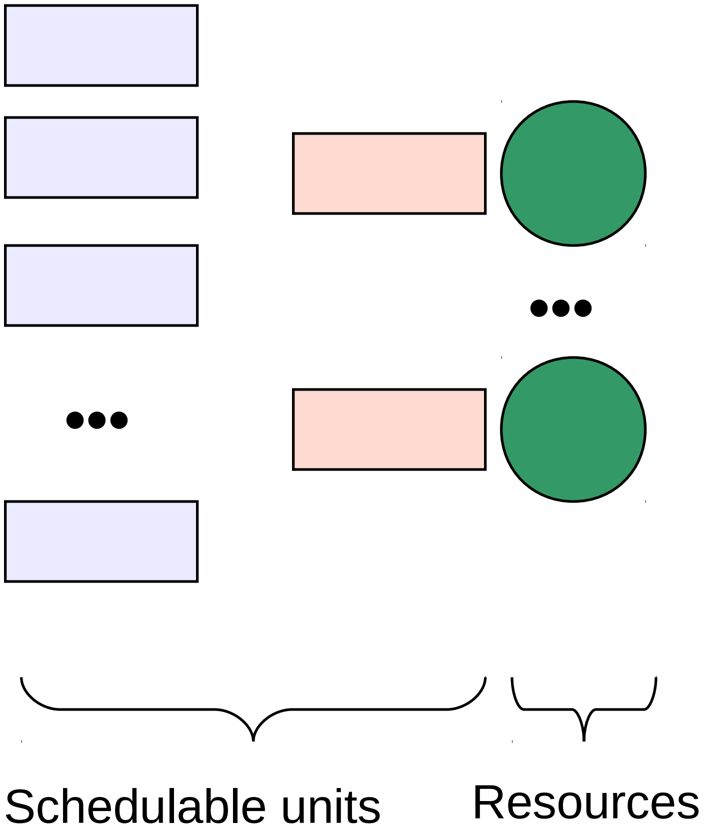
# Scheduling Goals I: Performance

- 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 IO 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

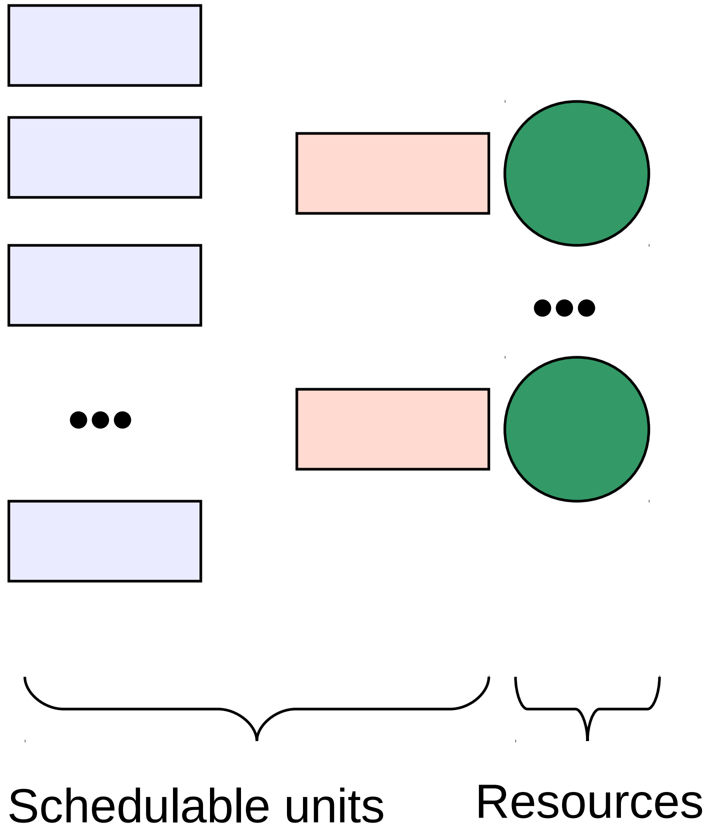
- We'll assume "work conserving" policies
  - Never leave a resource idle when someone wants it
    - Why even mention this? When might it be useful to do something else?

# 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, and S is average service time
    - This means that the 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
    - This means that good response time implies fewer in system
    - Fewer in system means less memory pressure

- Kleinrock's Conservation Law for priority scheduling
  - $\Sigma_p U_p * W_p$ = constant
  - Where $U_p$ is utlization by priority level p and $W_p$ is waiting time 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

# The basic situation




Scheduling:

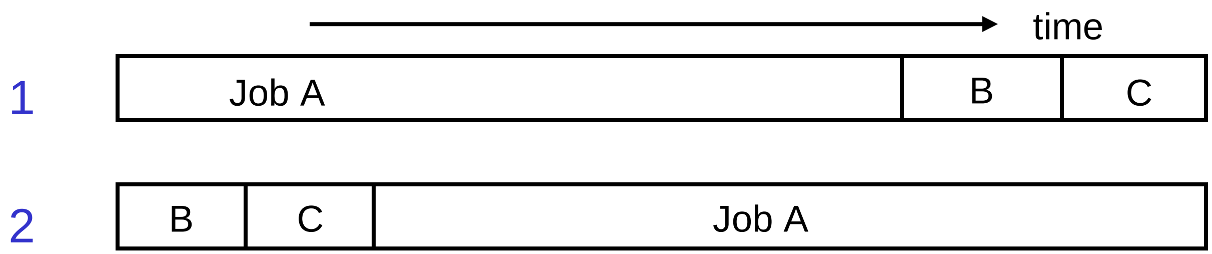
- Who to assign each resource to
- When to re-evaluate your decisions

# 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

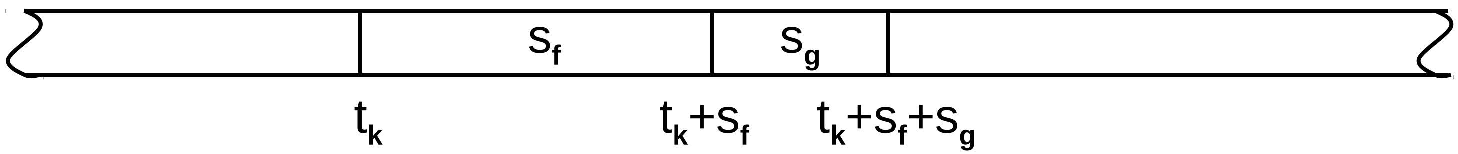
- Average response time can be lousy
  - small requests wait behind  big ones
- May lead to poor utilization of other resources
  - if you send me on my way, I can go keep another resource busy
  - FCFS may result in poor overlap of CPU and I/O activity
- Note: the more copies of the resource there are 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 pre-emption to offset lack of information about execution times
    - I don't know which one should go 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 pessimal 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

- Treats all “jobs” equally
  - if I run 100 copies of SETI@home, it degrades your service
  - how might I fix this?

# 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

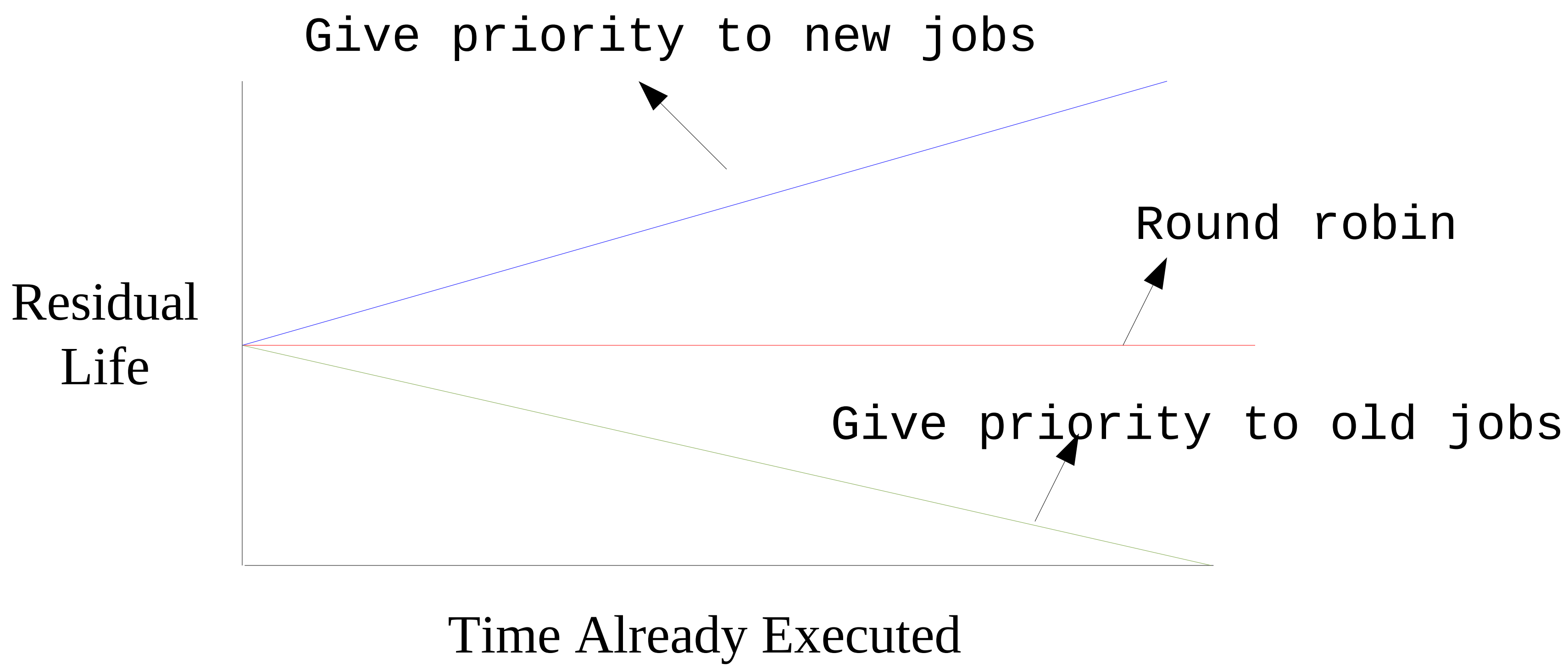
- How are you going to assign priorities?
- Starvation
  - if there is an endless supply of high priority jobs, no low-priority job will ever run
- Solution: "age" threads over time
  - increase priority as a function of accumulated wait time
  - decrease priority as a function of accumulated processing time
  - many ugly heuristics have been explored in this space

# Program Behavior and Scheduling

- An illuminating analogy:
  - Say you're at the airport waiting for a flight.
  - There are two identical ATMs:
    - ATM 1 has 3 people in line
    - ATM 2 has 6 people in line
  - You get into the line for ATM 1
  - ATM 2's line shrinks to 4 people
  - Why might you now switch lines, preferring 5$^{th}$ in line for ATM 2 over 4$^{th}$ in line for ATM 1?

# Residual Life

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



# Multi-level Feedback Queues (MLFQ)

- It has been observed that workloads tend to have increasing residual life

- This is exploited in practice using policy that discriminates against the old

- Multi-level feedback queues (MLFQ)
  - there is a hierarchy of queues
  - there is a priority ordering among the queues
  - new requests enter the highest priority queue
  - each queue is scheduled RR
  - requests move between queues based on execution history
    - The more CPU time you've accumulated (older), the lower down the list you descend
    - To prevent starvation, if you've been ignored for a while, you might move up a bit...
  - (an overhead reduction approach: queues have distinct quanta)

# 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

# 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 that exploit observed program behavior

# Bonus Material: Scheduling for Real-Time

- Just the high-level take-aways (from Chapter 19, Real-Time Systems), to be sure that they're mentioned

- In real-time systems, each task has a deadline
- The schedule (order of execution of tasks) determines whether or not each meets its deadline
- The sole performance goal is meeting the deadline

- Canonical examples:
  – Controlling flight trim on airplanes
  – Throttle control on the Prius…

(X-29, 1984; 25 msec. deadline)

# Earliest-Deadline-First (EDF)

- At any moment in time, there are a set of tasks to be scheduled, each with a deadline
- EDF optimality: If the tasks can be scheduled so that all meet their deadlines, then scheduling earliest-deadline-first will meet them as well
- Informal “proof”: an interchange argument
- A weakness: if the deadlines can't all be met, the behavior of EDF may be terrible

# Rate Monotonic Scheduling: RM

- Suppose the set of tasks is periodic:
  - each of them will be executed repeatedly (forever), and that the deadline is stated as a period (the amount of time we have to complete the next execution of the task)
  - (A task's rate is the inverse of its period)

- RM optimality: if any static assignment of priorities to tasks can guarantee that all deadlines are met, RM will meet all deadlines

- RM has more graceful behavior under overload than EDF, but…
- EDF can schedule some task sets RM cannot