CS 202: Advanced Operating Systems

Scheduling

CPU Scheduling

- Scheduler runs when we context switching among processes/threads on the ready queue
  - What should it do? Does it matter?
- Making the decision on what thread to run is called scheduling
  - What are the goals of scheduling?
  - What are common scheduling algorithms?
  - Lottery scheduling
  - Stride Scheduling
- Scheduling activations
  - User level vs. Kernel level scheduling of threads

Scheduling

- Right from the start of multiprogramming, scheduling was identified as a big issue
  - CCTS and Multics developed much of the classical algorithms
- Scheduling is a form of resource allocation
  - CPU is the resource
  - Resource allocation needed for other resources too; sometimes similar algorithms apply
- Requires mechanisms and policy
  - Mechanisms: Context switching, Timers, process queues, process state information, …
  - Scheduling looks at the policies: i.e., when to switch and which process/thread to run next

Preemptive vs. Non-preemptive scheduling

- In *preemptive* systems where we can interrupt a running job (involuntary context switch)
  - We're interested in such schedulers…

- In *non-preemptive* systems, the scheduler waits for a running job to give up CPU (voluntary context switch)
  - Was interesting in the days of batch multiprogramming
  - Some systems continue to use cooperative scheduling

- Example algorithms:
  - RR, FCFS, Shortest Job First (how to determine shortest), Priority Scheduling

Scheduling Goals

- What are some reasonable goals for a scheduler?
- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput (# jobs/unit time)
  - Response time (Avg(T_{ready}): avg time spent on ready queue)
  - Fairness (or weighted fairness)
  - Other?

- Non-interactive applications:
  - Strive for job throughput, turnaround time (supercomputers)
- Interactive systems
  - Strive to minimize response time for interactive jobs
- Mix?

Goals II: Avoid Resource allocation pathologies

- Starvation no progress due to no access to resources
  - E.g., a high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock

- Priority inversion
  - A low priority process running before a high priority one
  - Could be a real problem, especially in real time systems
    - Mars pathfinder: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html

- Other
  - Deadlock, livelock, …

First In First Out (FIFO)

- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor
- Example: memcached
  - Facebook cache of friend lists, …

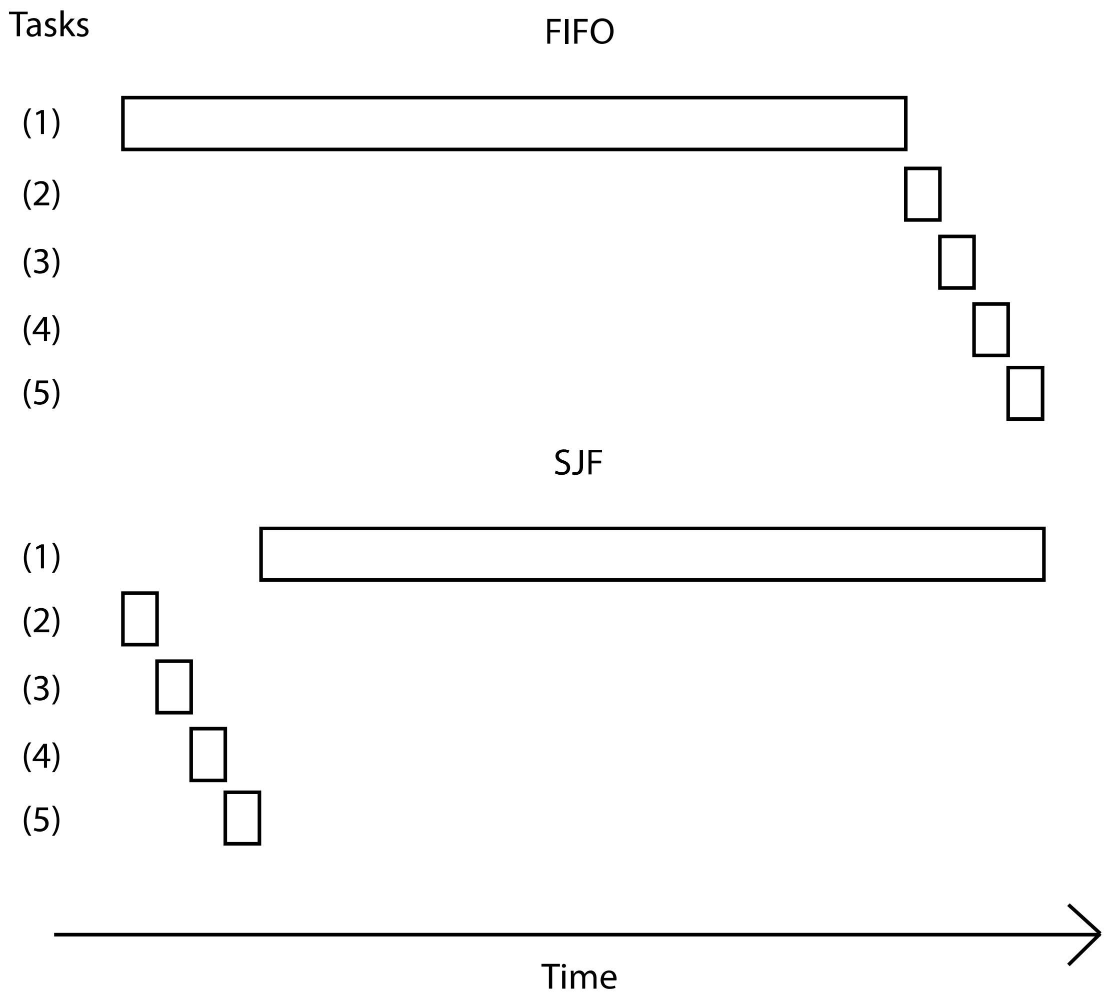
- On what workloads is FIFO particularly bad?
  - Imagine being at supermarket to buy a drink of water, but get stuck behind someone with a huge cart (or two!)
    - …and who pays in pennies!
  - Can we do better?

Shortest Job First (SJF)

- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

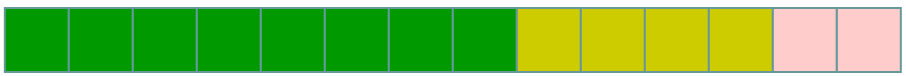
- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
  - Which completes first in SJF? Next?

FIFO vs. SJF

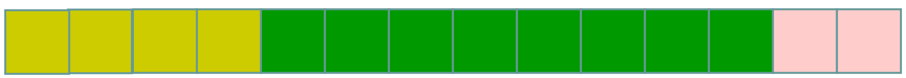


Whats the big deal? Don’t they finish at the same time?

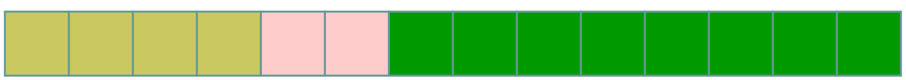
SJF Example



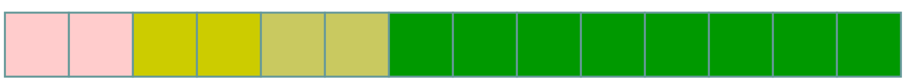
$ART = (0 + 8 + (8+4))/3 = 6.67$



$ART = (0 + 4 + (4+8))/3 = 5.33$



$ART = (0 + 4 + (4+2))/3 = 3.33$



$ART = (0 + 2 + (2+4))/3 = 2.67$

SJF

- Claim: SJF is optimal for average response time
  - Why?
- For what workloads is FIFO optimal?
  - For what is it pessimal (i.e., worst)?
- Does SJF have any downsides?

Shortest Job First (SJF)

- Problems?
  - Impossible to know size of CPU burst
    - Like choosing person in line without looking inside basket/cart
  - How can you make a reasonable guess?
  - Can potentially starve
- Flavors
  - Can be either preemptive or non-preemptive
  - Preemptive SJF is called shortest remaining time first (SRTF)

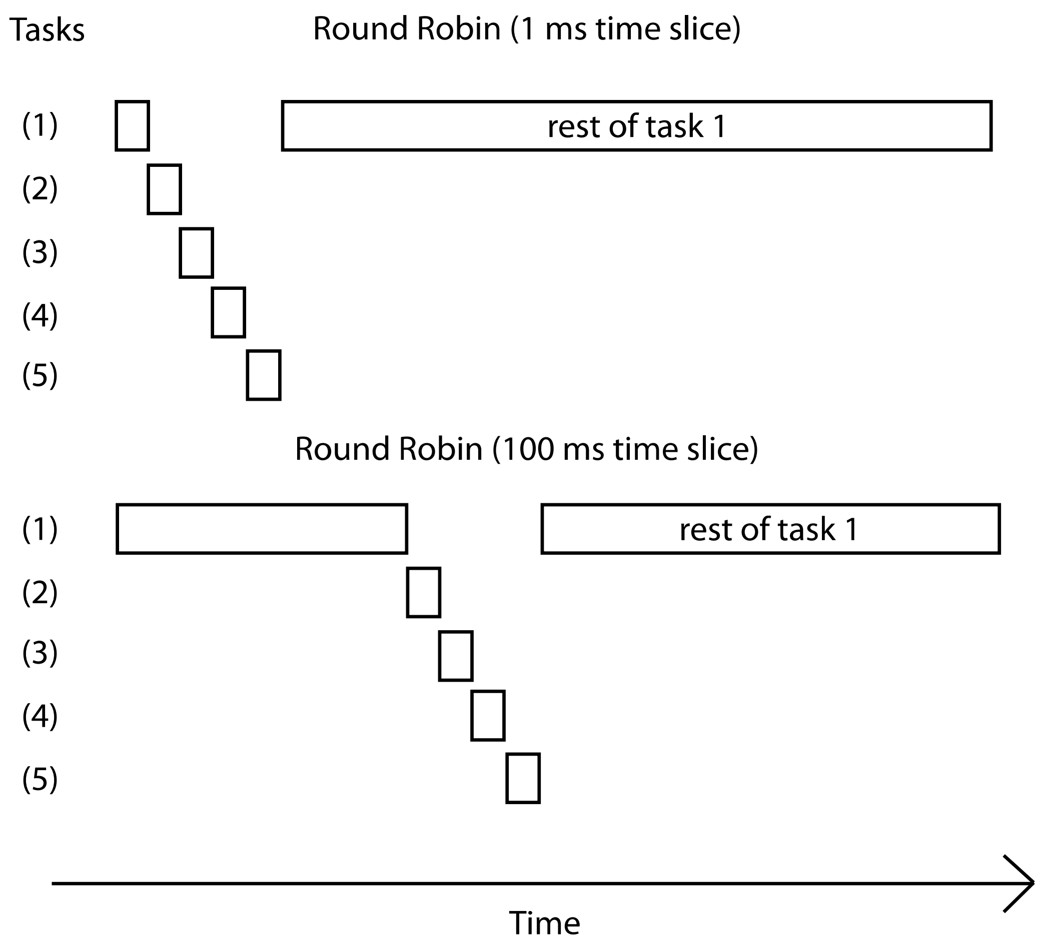
Preemptive scheduling: Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

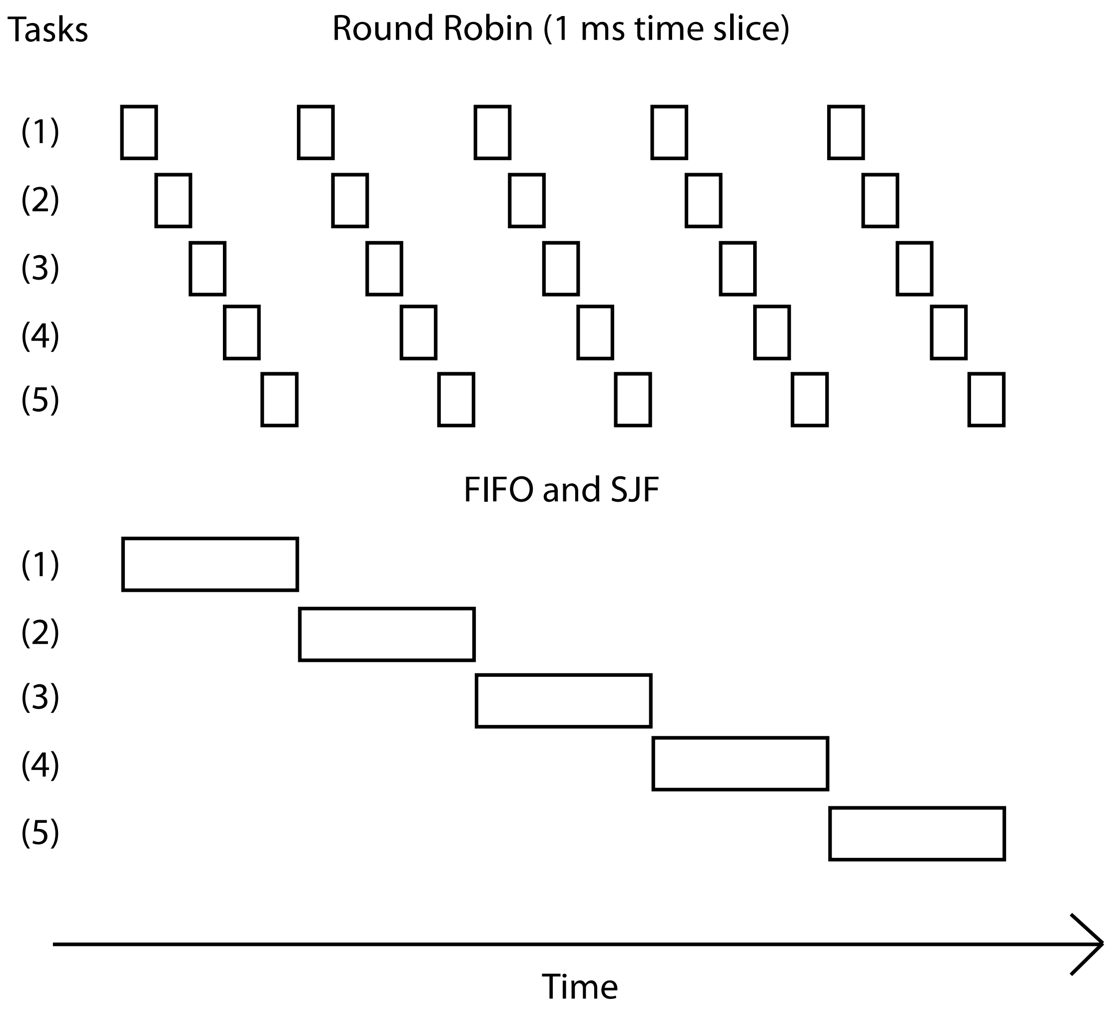
Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
rest of task 1
(2)
(3)
(4)
(5)
Round Robin (100 ms time slice)
(1)
rest of task 1
(2)
(3)
(4)
(5)
Time

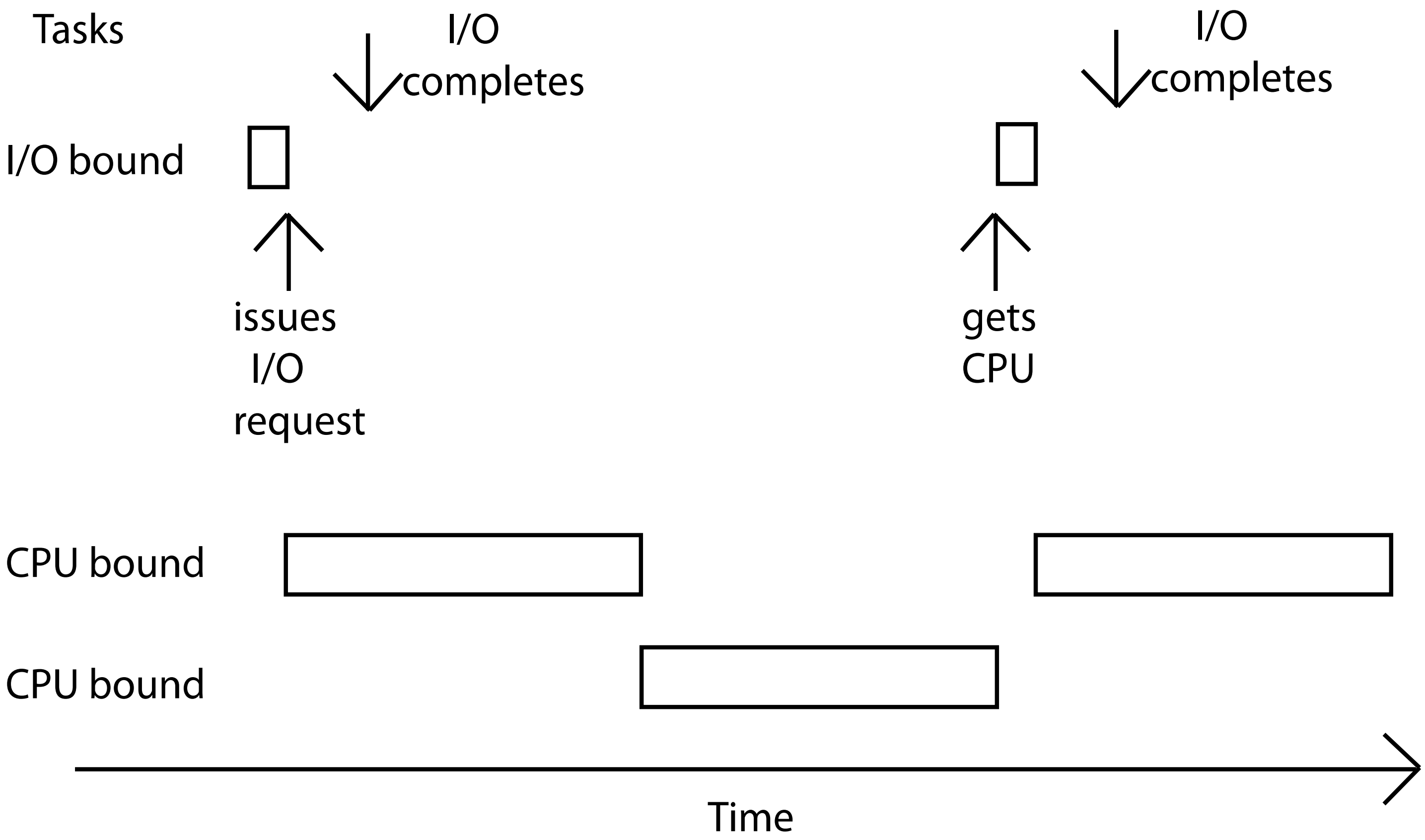
Round Robin vs. FIFO

- Many context switches can be costly
- Other than that, is Round Robin always better than FIFO?

Round Robin vs. FIFO



Mixed Workload

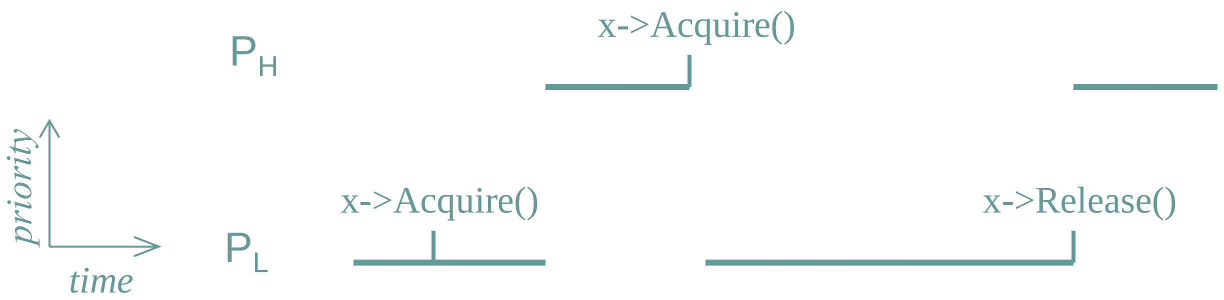


Priority Scheduling

- Priority Scheduling
  - Choose next job based on priority
    - Airline check-in for first class passengers
  - Can implement SJF, priority = 1/(expected CPU burst)
  - Also can be either preemptive or non-preemptive
- Problem?
  - Starvation – low priority jobs can wait indefinitely
- Solution
  - “Age” processes
    - Increase priority as a function of waiting time
    - Decrease priority as a function of CPU consumption

More on Priority Scheduling

- For real-time (predictable) systems, priority is often used to isolate a process from those with lower priority. *Priority inversion* is a risk unless all resources are jointly scheduled.



Priority Inheritance

- If lower priority process is being waited on by a higher priority process it inherits its priority
  - How does this help?
  - Does it prevent the previous problem?

- Priority inversion is a big problem for real-time systems
  - Mars pathfinder bug (link)

Combining Algorithms

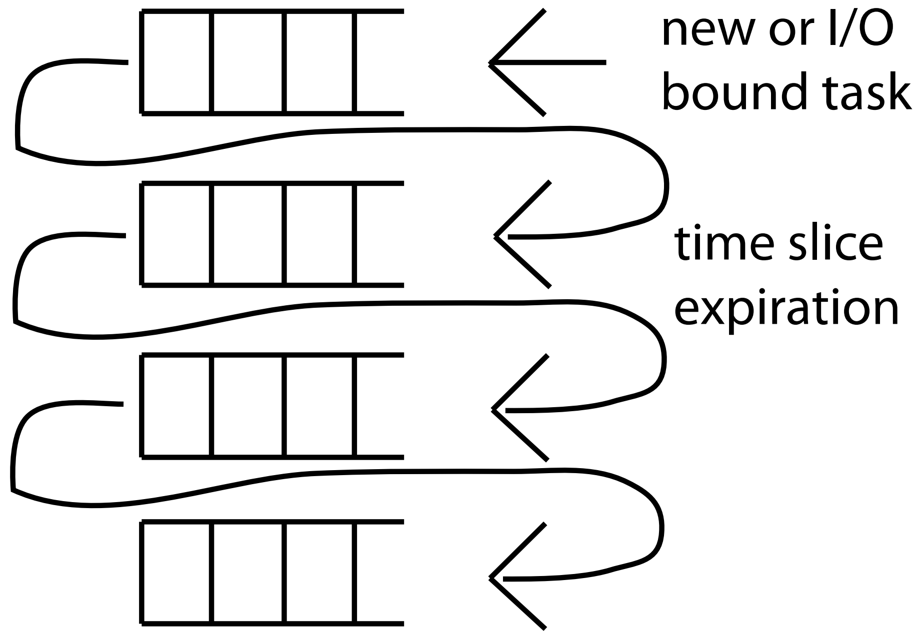
- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm for each queue
  - Move processes among queues
- Example: Multiple-level feedback queues (MLFQ)
  - Multiple queues representing different job types
    - Interactive, CPU-bound, batch, system, etc.
  - Queues have priorities, jobs on same queue scheduled RR
  - Jobs can move among queues based upon execution history
    - Feedback: Switch from interactive to CPU-bound behavior

Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Unix (and Windows and MacOS)

MFQ

Priority	Time Slice (ms)	Round Robin Queues
1	10	
2	20	
3	40	
4	80	



Unix Scheduler

- The canonical Unix scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - Timesharing: first 60 priorities
    - System: next 40 priorities
    - Real-time: next 60 priorities
    - Interrupt: next 10 (Solaris)
- Priority scheduling across queues, RR within a queue
  - The process with the highest priority always runs
  - Processes with the same priority are scheduled RR
- Processes dynamically change priority
  - Increases over time if process blocks before end of quantum
  - Decreases over time if process uses entire quantum

Linux scheduler

- Went through several iterations
- Currently CFS
  - Fair scheduler, like stride scheduling
  - Supersedes O(1) scheduler: emphasis on constant time scheduling regardless of overhead
  - CFS is O(log(N)) because of red-black tree
  - Is it really fair?
- What to do with multi-core scheduling?

Problems with Traditional schedulers

- Priority systems are ad hoc: highest priority always wins
- Try to support fair share by adjusting priorities with a feedback loop
  - Works over long term
  - highest priority still wins all the time, but now the Unix priorities are always changing
- Priority inversion: high-priority jobs can be blocked behind low-priority jobs
- Schedulers are complex and difficult to control