



Scheduling

◆ ◆ ◆ ◆ ◆
Don Porter
CSE 306

Last time



- ✦ We went through the high-level theory of scheduling algorithms
- ✦ Today: View into how Linux makes its scheduling decisions

Lecture goals



- ✦ Understand low-level building blocks of a scheduler
- ✦ Understand competing policy goals
- ✦ Understand the $O(1)$ scheduler
 - ✦ CFS next lecture
- ✦ Familiarity with standard Unix scheduling APIs

(Linux) Terminology Review

- ✦ `mm_struct` – represents an address space in kernel
- ✦ `task` – represents a thread in the kernel
 - ✦ A task points to 0 or 1 `mm_structs`
 - ✦ Kernel threads just “borrow” previous task’s `mm`, as they only execute in kernel address space
 - ✦ Many tasks can point to the same `mm_struct`
 - ✦ Multi-threading
- ✦ Quantum – CPU timeslice

Outline



- ✦ Policy goals (review)
- ✦ $O(1)$ Scheduler
- ✦ Scheduling interfaces

Policy goals



- ✦ Fairness – everything gets a fair share of the CPU
- ✦ Real-time deadlines
 - ✦ CPU time before a deadline more valuable than time after
- ✦ Latency vs. Throughput: Timeslice length matters!
 - ✦ GUI programs should feel responsive
 - ✦ CPU-bound jobs want long timeslices, better throughput
- ✦ User priorities
 - ✦ Virus scanning is nice, but I don't want it slowing things down

No perfect solution



- ✦ Optimizing multiple variables
- ✦ Like memory allocation, this is best-effort
 - ✦ Some workloads prefer some scheduling strategies
- ✦ Nonetheless, some solutions are generally better than others

Outline



- ✦ Policy goals
- ✦ $O(1)$ Scheduler
- ✦ Scheduling interfaces

$O(1)$ scheduler



- ✦ Goal: decide who to run next, independent of number of processes in system
 - ✦ Still maintain ability to prioritize tasks, handle partially unused quanta, etc

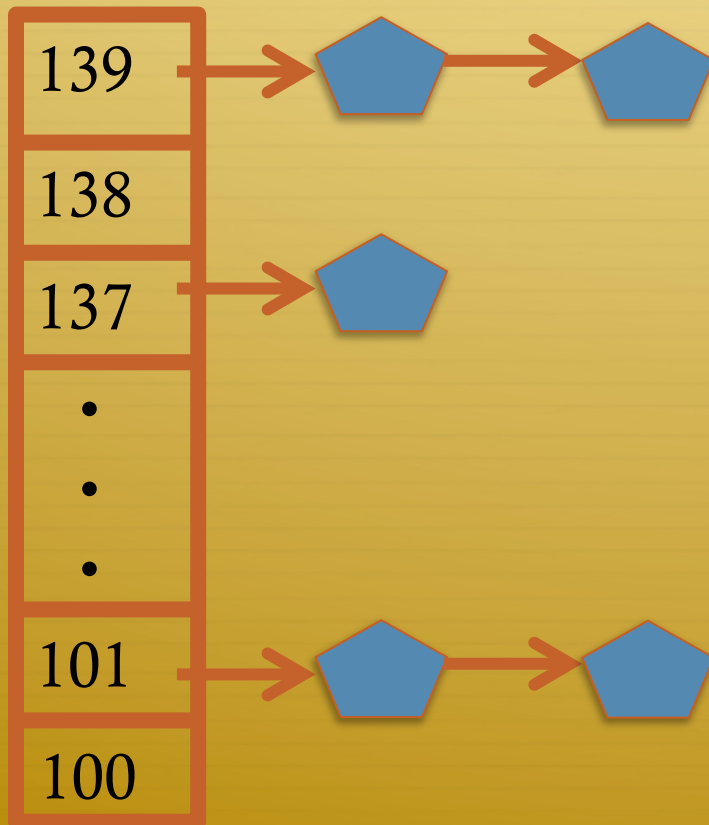
O(1) Bookkeeping



- ✦ runqueue: a list of runnable processes
 - ✦ Blocked processes are not on any runqueue
 - ✦ A runqueue belongs to a specific CPU
 - ✦ Each task is on exactly one runqueue
 - ✦ Task only scheduled on runqueue's CPU unless migrated
- ✦ $2 * 40 * \text{\#CPUs}$ runqueues
 - ✦ 40 dynamic priority levels (more later)
 - ✦ 2 sets of runqueues – one active and one expired

O(1) Data Structures

Active



Expired



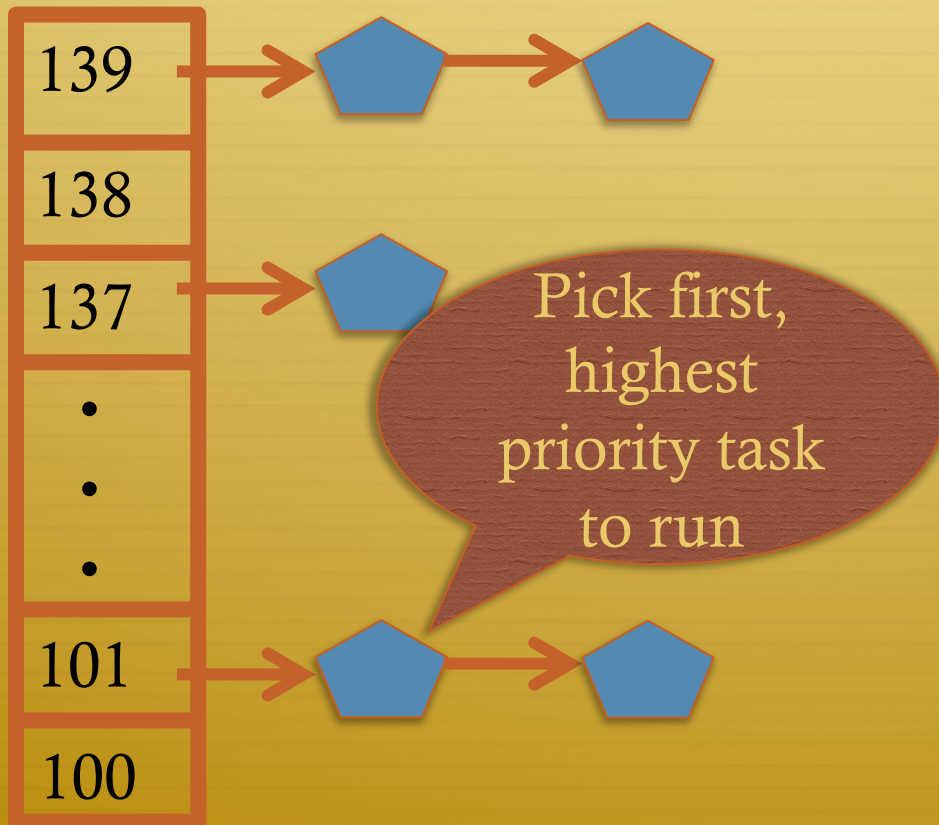
$O(1)$ Intuition



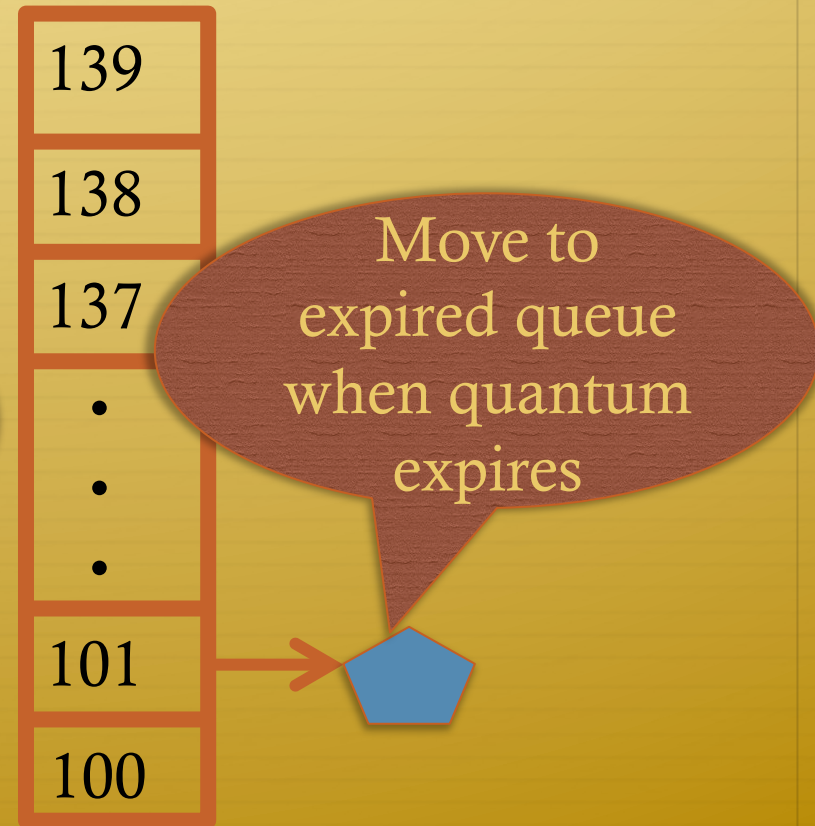
- ✦ Take the first task off the lowest-numbered runqueue on active set
 - ✦ Confusingly: a lower priority value means higher priority
- ✦ When done, put it on appropriate runqueue on expired set
- ✦ Once active is completely empty, swap which set of runqueues is active and expired
- ✦ Constant time, since fixed number of queues to check; only take first item from non-empty queue

O(1) Example

Active



Expired

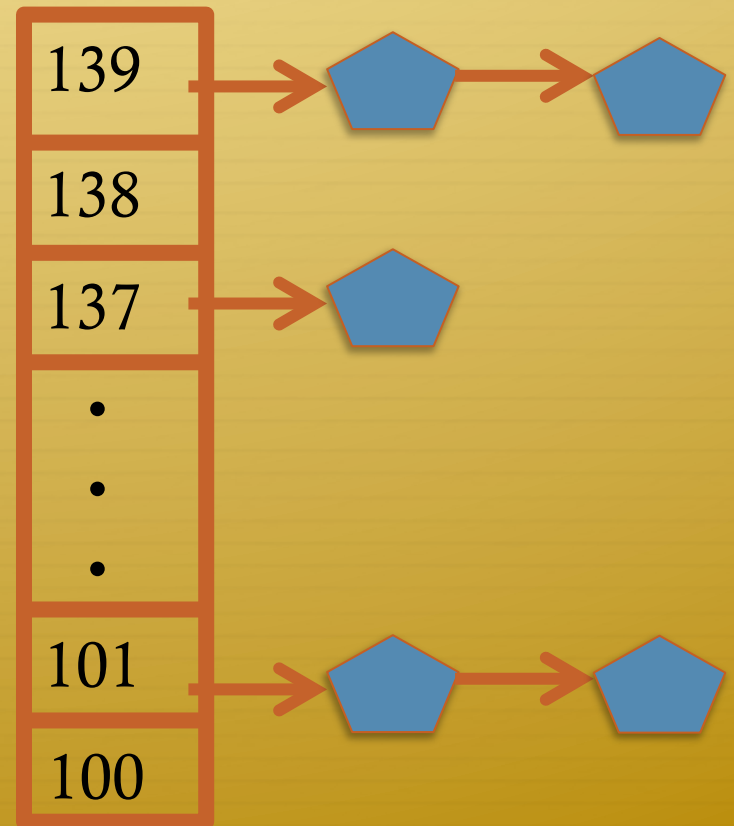


What now?

Active



Expired

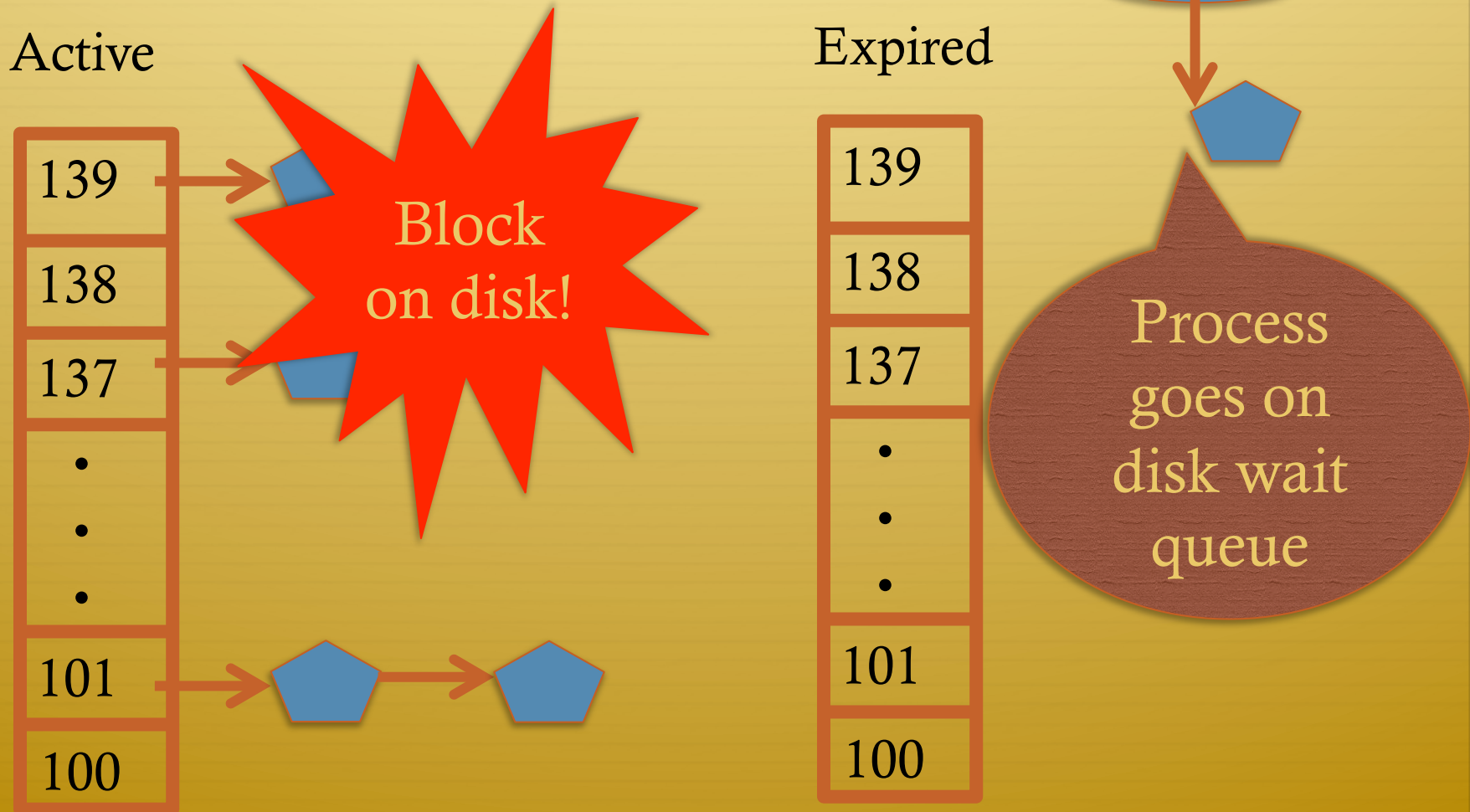


Blocked Tasks



- ✦ What if a program blocks on I/O, say for the disk?
 - ✦ It still has part of its quantum left
 - ✦ Not runnable, so don't waste time putting it on the active or expired runqueues
- ✦ We need a “wait queue” associated with each blockable event
 - ✦ Disk, lock, pipe, network socket, etc.

Blocking Example



Blocked Tasks, cont.



- ✦ A blocked task is moved to a wait queue until the expected event happens
 - ✦ **No longer on any active or expired queue!**
- ✦ Disk example:
 - ✦ After I/O completes, interrupt handler moves task back to active runqueue

Time slice tracking



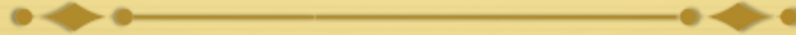
- ✦ If a process blocks and then becomes runnable, how do we know how much time it had left?
- ✦ Each task tracks ticks left in 'time_slice' field
 - ✦ On each clock tick: `current->time_slice--`
 - ✦ If time slice goes to zero, move to expired queue
 - ✦ Refill time slice
 - ✦ Schedule someone else
 - ✦ An unblocked task can use balance of time slice
 - ✦ Forking halves time slice with child

More on priorities



- ✦ 100 = highest priority
- ✦ 139 = lowest priority
- ✦ 120 = base priority
 - ✦ “nice” value: user-specified adjustment to base priority
 - ✦ Selfish (not nice) = -20 (I want to go first)
 - ✦ Really nice = +19 (I will go last)

Base time slice



$$time = \begin{cases} (140 - prio) * 20ms & prio < 120 \\ (140 - prio) * 5ms & prio \geq 120 \end{cases}$$

- ✦ “Higher” priority tasks get longer time slices
 - ✦ And run first

Goal: Responsive UIs



- ✦ Most GUI programs are I/O bound on the user
 - ✦ Unlikely to use entire time slice
- ✦ Users get annoyed when they type a key and it takes a long time to appear
- ✦ Idea: give UI programs a priority boost
 - ✦ Go to front of line, run briefly, block on I/O again
- ✦ Which ones are the UI programs?

Idea: Infer from sleep time



- ✦ By definition, I/O bound applications spend most of their time waiting on I/O
- ✦ We can monitor I/O wait time and infer which programs are GUI (and disk intensive)
- ✦ Give these applications a priority boost
- ✦ Note that this behavior can be dynamic
 - ✦ Ex: GUI configures DVD ripping, then it is CPU-bound
 - ✦ Scheduling should match program phases

Dynamic priority



$dynamic\ priority = \max (100, \min (static\ priority - bonus + 5, 139))$

- ✦ Bonus is calculated based on sleep time
- ✦ Dynamic priority determines a tasks' runqueue
- ✦ This is a heuristic to balance competing goals of CPU throughput and latency in dealing with infrequent I/O
 - ✦ May not be optimal

Dynamic Priority in $O(1)$ Scheduler

- ✦ Important: The runqueue a process goes in is determined by the **dynamic** priority, not the static priority
 - ✦ Dynamic priority is mostly determined by time spent waiting, to boost UI responsiveness
- ✦ Nice values influence **static** priority
 - ✦ No matter how “nice” you are (or aren’t), you can’t boost your dynamic priority without blocking on a wait queue!

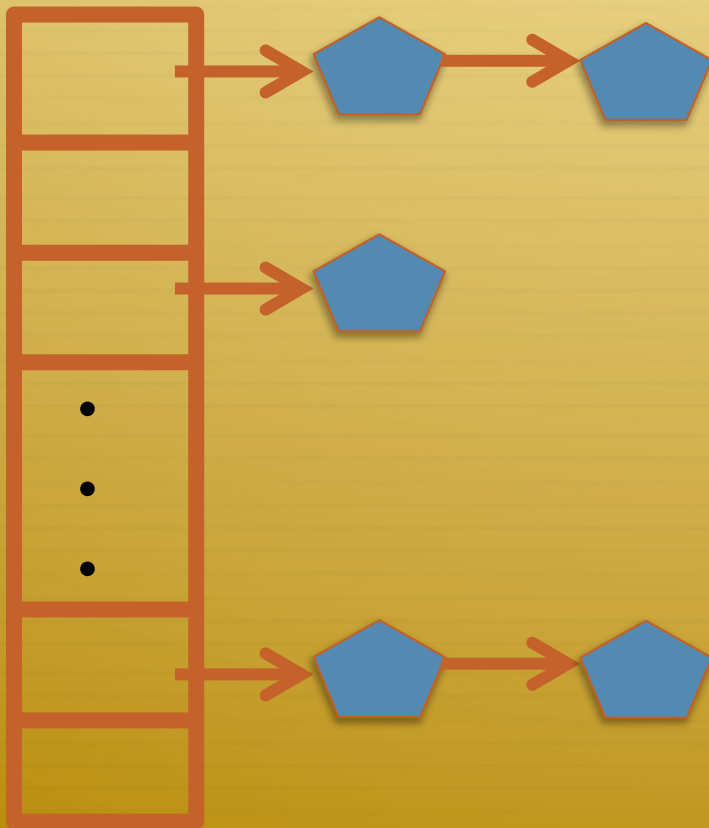
Rebalancing tasks



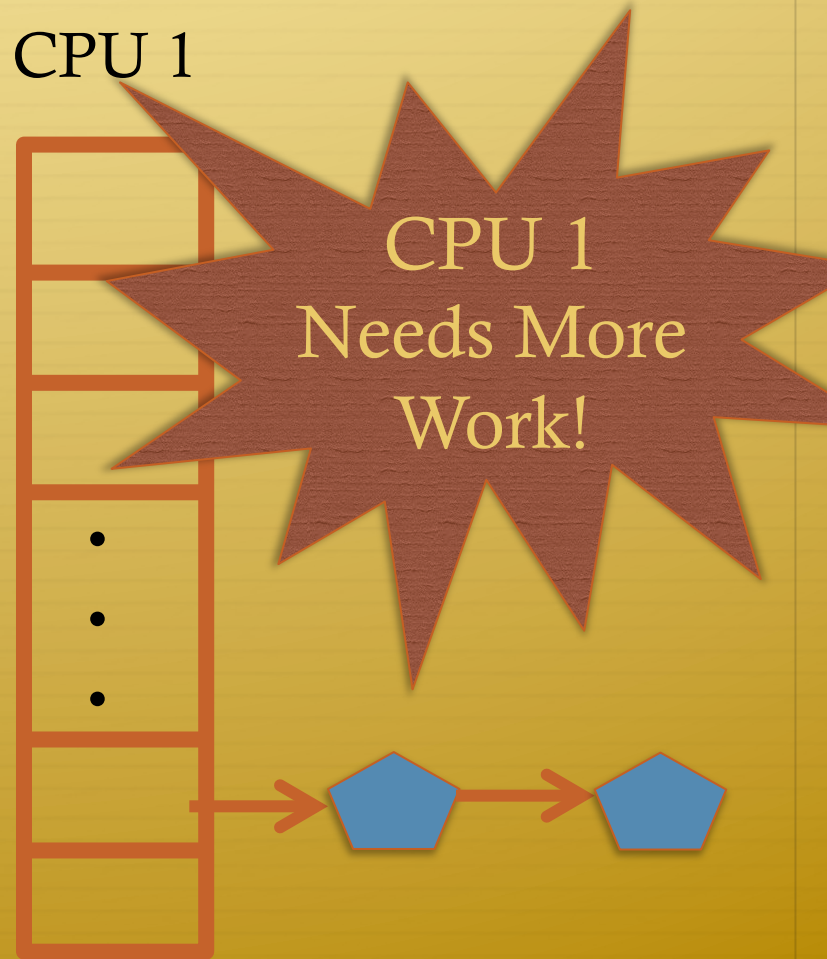
- ✦ As described, once a task ends up in one CPU's runqueue, it stays on that CPU forever

Rebalancing

CPU 0



CPU 1



Rebalancing tasks



- ✦ As described, once a task ends up in one CPU's runqueue, it stays on that CPU forever
- ✦ What if all the processes on CPU 0 exit, and all of the processes on CPU 1 fork more children?
- ✦ We need to periodically rebalance
- ✦ Balance overheads against benefits
 - ✦ Figuring out where to move tasks isn't free

Idea: Idle CPUs rebalance



- ✦ If a CPU is out of runnable tasks, it should take load from busy CPUs
 - ✦ Busy CPUs shouldn't lose time finding idle CPUs to take their work if possible
- ✦ There may not be any idle CPUs
 - ✦ Overhead to figure out whether other idle CPUs exist
 - ✦ Just have busy CPUs rebalance much less frequently

Average load



- ✦ How do we measure how busy a CPU is?
- ✦ Average number of runnable tasks over time
- ✦ Available in `/proc/loadavg`

Rebalancing strategy



- ✦ Read the loadavg of each CPU
- ✦ Find the one with the highest loadavg
- ✦ (Hand waving) Figure out how many tasks we could take
 - ✦ If worth it, lock the CPU's runqueues and take them
 - ✦ If not, try again later

Outline



- ✦ Policy goals
- ✦ $O(1)$ Scheduler
- ✦ Scheduling interfaces

Setting priorities



- ✦ `setpriority(which, who, niceval)` and `getpriority()`
 - ✦ Which: process, process group, or user id
 - ✦ PID, PGID, or UID
 - ✦ Niceval: -20 to +19 (recall earlier)
- ✦ `nice(niceval)`
 - ✦ Historical interface (backwards compatible)
 - ✦ Equivalent to:
 - ✦ `setpriority(PRIO_PROCESS, getpid(), niceval)`

Scheduler Affinity



- ✦ `sched_setaffinity` and `sched_getaffinity`
- ✦ Can specify a bitmap of CPUs on which this can be scheduled
 - ✦ Better not be 0!
- ✦ Useful for benchmarking: ensure each thread on a dedicated CPU

yield



- ✦ Moves a runnable task to the expired runqueue
 - ✦ Unless real-time (more later), then just move to the end of the active runqueue
- ✦ Several other real-time related APIs

Summary



- ✦ Understand competing scheduling goals
- ✦ Understand $O(1)$ scheduler + rebalancing
- ✦ Scheduling system calls