Context Switching & CPU Scheduling

Nima Honarmand

Administrivia

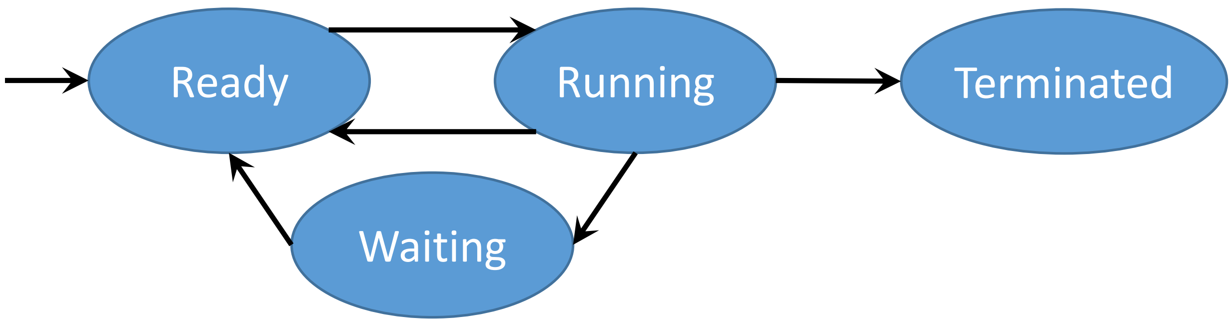
- **Midterm: next Tuesday, 10/17, in class**
- Will include everything discussed until then
- Will cover:
  - Class lectures, slides and discussions
  - All required readings (as listed on the course schedule page)
  - All blackboard discussions
  - Labs 1 and 2 and relevant xv6 code

Thread as CPU Abstraction

- Thread: OS abstraction of a CPU as exposed to programs
- Each process needs at least one thread
  - Can’t run a program without a CPU, right?
- Multi-threaded programs can have multiple threads which share the same process address space (i.e., page table and segments)
  - Analogy: multiple physical CPUs share the same physical memory

Thread States

- **Running:** the thread is scheduled and running on a CPU (either in user or kernel mode)
- **Ready (Runnable):** the thread is not currently running because it does not have a CPU to run on; otherwise, it is ready to execute
- **Waiting (Blocked):** the thread cannot be run (even if there are idle CPUs) because it is waiting for the completion of an I/O operation (e.g., disk access)
- **Terminated:** the thread has exited; waiting for its state to be cleaned up



Thread State Transitions

- **Ready → Running:** a ready thread is selected by the CPU scheduler and is switched in
- **Running → Waiting:** a running thread performing a blocking operation (e.g., requests disk read) and cannot run until the request is complete
- **Running → Ready:** a running thread is descheduled to give the CPU to another thread (not because it made a blocking request); it is ready to re-run as soon as CPU becomes available again
- **Waiting → Ready:** thread's blocking request is complete and it is ready to run again
- **Running → Terminated:** running thread calls an exit function (or terminates otherwise) and sticks around for some final book-keeping but does not need to run anymore

Run and Wait Queues

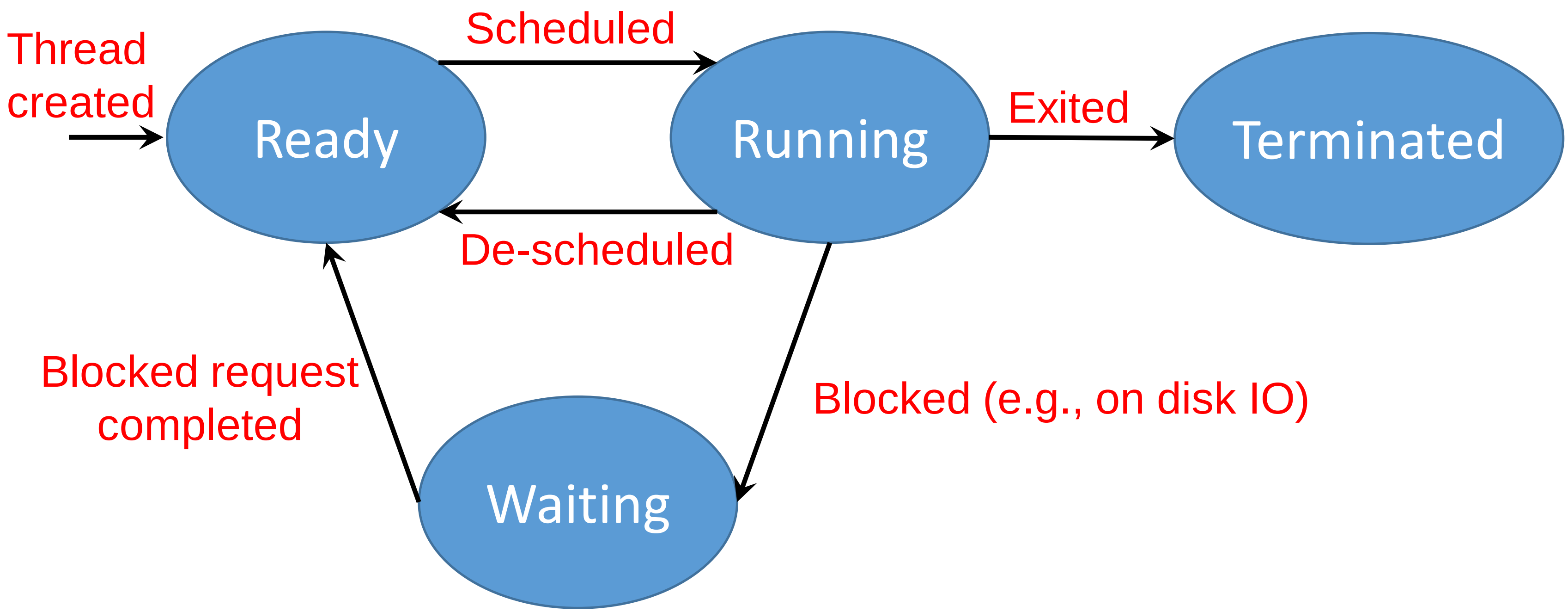
- Kernel keeps Ready threads in one or more **Ready (Run) Queue** data structures
  - CPU scheduler checks the run queue to pick the next thread
- Kernel puts a thread on a wait queue when it *blocks*, and transfers it to a run queue when it is ready to run again
  - Usually, there are separate wait queues for different causes of blocking (disk access, network, locks, etc.)

→ Each thread is either running, or ready in some run queue, or sleeping in some wait queue

- CPU Scheduler only looks among Ready threads for the next thread to run

Stony Brook University

Thread State Transitions



- How to transition? (Mechanism)
- When to transition? (Policy)

Mechanism: Context Switching

Thread's Are Like Icebergs

- You might think of a thread as a user-mode-only concept
  - Time to correct that conception!
- In general, a thread has both user-mode and kernel-mode lives
  - Like an iceberg that is partly above pater and partly below.

Thread's Are Like Icebergs (cont'd)

- When CPU is in user-mode, it is executing the *current* thread in user-mode
  - Code that thread executes comes from program instructions
- When CPU transitions to supervisor mode and starts running kernel code (because of a syscall, exception or interrupt) it is <u>still in the context of the current thread</u>
  - Code that thread executes comes from kernel instructions

Decouple notion of thread from user-mode code!

Thread's Life in Kernel & User Modes

Execution

Program Code

```
int x = getpid();
printf("my pid is %d\n", x);
```

User-mode execution (code from program ELF)

(thread is using user-mode stack)
…
Call `getpid()` library function
…
`int 0x80` (Linux system call)

Kernel-mode execution (code from kernel binary)

(use kernel-mode stack)
Save all registers on the kernel-mode stack
call `sys_getpid()`
Restore registers from kernel-mode stack
`iret` (to return to user-mode)

User-mode execution (code from program ELF)

(use user-mode stack)
return from `getpid()` library call
Call `printf()` library call
…
`int 0x80` (Linux system call)

Kernel-mode execution (code from kernel binary)

(use kernel-mode stack)
Save all registers on the kernel-mode stack
…
`iret` (to return to user-mode)

User-mode execution (code from program ELF)

(use user-mode stack)
return from `printf()` library call
…

User-mode execution
(code from program ELF)

Kernel-mode execution
(code from kernel binary)

Context Switching

- Context Switch: saving the **context** of the current thread, restore that of the next one, and start executing the next thread
- When can OS run the code to do a context switch?
  - When execution is in kernel
    - Because of a system call (e.g., `read`), exception (e.g., page fault) or an interrupt (e.g., timer interrupt)
  - …and only when execution is in kernel
    - When in user-mode, kernel code is not running, is it?

Thread Context

- Now that thread can have both user-mode and kernel-mode lives…
- It would also have separate user-mode and kernel-mode contexts
  - User-mode context: register values when running in user mode + user-mode stack
  - Kernel-mode context: register values when running in kernel mode + kernel-mode stack

Saving and Restoring Thread Context

- Again: context switching only happens when kernel code is running
- We have already saved current thread's user-mode context when switching to the kernel
  - So no need to worry about that
- We just need to save current thread's kernel mode context before switching
  - Where? Can save it on the kernel-mode stack of current thread

Context Switch Timeline

	Operating System	Hardware	Program
In A's Context			Thread A in user mode
		timer interrupt save user regs(A) to k-stack(A) witch to kernel mode jump to trap handler	
	Handle the trap Call switch() routine - save kernel regs(A) to k-stack(A)		
	- switch to k-stack(B)		
In B's Context	- restore kernel regs(B) from k-stack(B) return-from-trap (into B)		
		restore user regs(B) from k-stack(B) switch to user mode jump to B's IP	
			Thread B in user mode

xv6 Code Review

- `swtch()` function

When to Call `swtch()`?

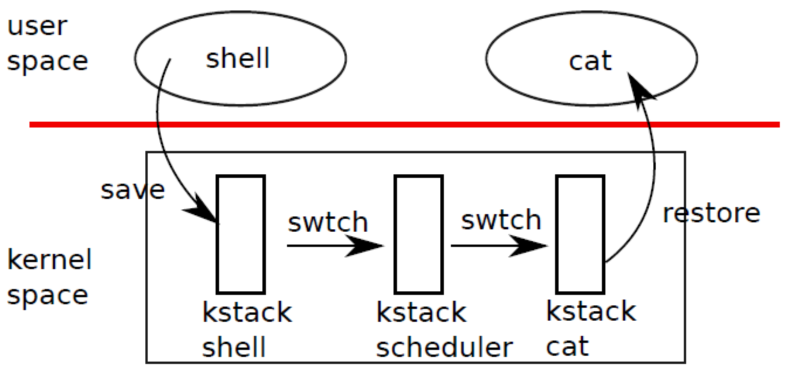
- Can only happen when in kernel mode

1) **Cooperative multi-tasking**: only when current thread voluntarily relinquishes the CPU
   - I.e., when it makes system calls like yield(), sleep(), exit() or when it performs a blocking system call (such as disk read)

2) **Preemptive multi-tasking**: take the CPU away by force, even if the thread has made no system calls
   - Use timer interrupts to force a transition to kernel
   - Once in the kernel, we can call `swtch()` if we want to

Role of CPU Scheduler

- `swtch()` just switches between two threads; it doesn’t decide which thread should be next
- Who makes that decision?
  - Answer: CPU scheduler
  - CPU Scheduler is the piece of logic that decides who should run next and for how long
- xv6 code review
  - In xv6, scheduler runs on its own thread (which runs totally in kernel mode)
  - In Linux, it runs in the context of current thread



Policy: Scheduling Discipline

Vocabulary

- Workload: set of jobs
  - Each job described by (*arrival_time*, *run_time*)
- Job: view as current CPU burst of a thread until it blocks again
  - Thread alternates between CPU and blocking operations (I/O, sleep, etc.)
- Scheduler: logic that decides which ready job to run
- Metric: measurement of scheduling quality

Workload Assumptions and Policy Goals

- (Simplistic) workload assumptions
  1) Each job runs for the same amount of time
  2) All jobs arrive at the same time
  3) Run-time of each job is known
- Metric: **Turnaround Time**
  - Job Turnaround Time: *completion_time* − *arrival_time*
- Goal: minimize average job turnaround time

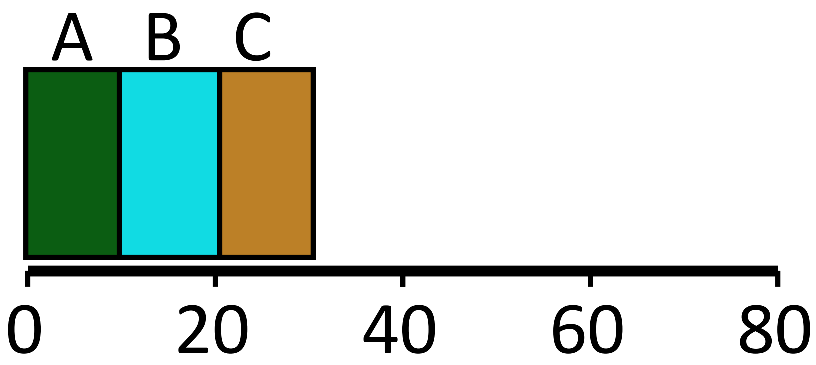
Simple Scheduler: FIFO

JOB	arrival_time (s)	run_time
A	~0	10
B	~0	10
C	~0	10

- FIFO: First In, First Out
  - also called FCFS (first come, first served)
  - run jobs in *arrival_time* order until completion
- What is the average turnaround time?

FIFO (Identical Jobs)

JOB	arrival_time (s)	run_time
A	~0	10
B	~0	10
C	~0	10



Avg. turnaround
= (10 + 20 + 30) /3
= 20

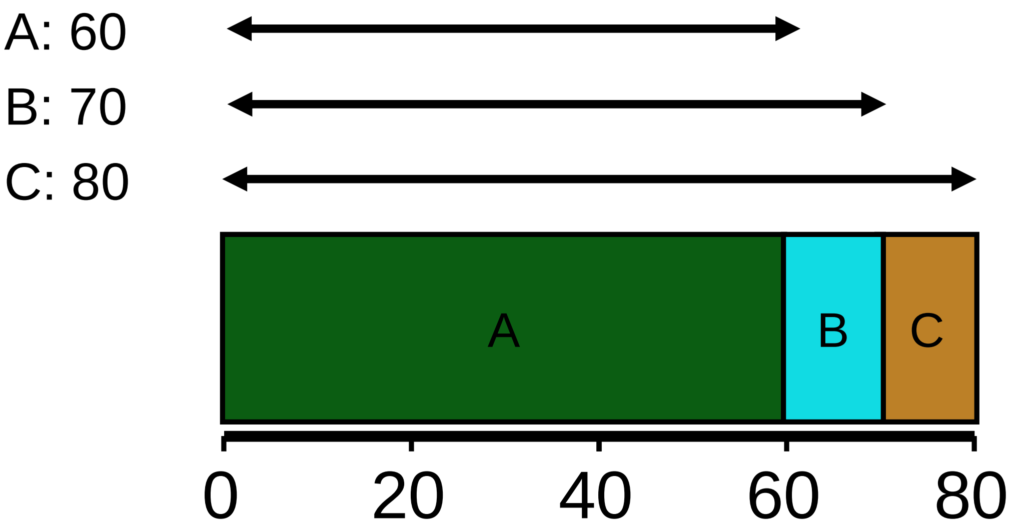
More Realistic Workload Assumptions

- Workload Assumptions
  1) ~~Each job runs for the same amount of time~~
  2) All jobs arrive at the same time
  3) Run-time of each job is known

- Any problematic workload for FIFO with new assumptions?
  - Hint: something resulting in non-optimal (i.e., high) turnaround time

FIFO: Big First Job

JOB	arrival_time (s)	run_time
A	~0	**60**
B	~0	10
C	~0	10



Avg. turnaround
= (60 + 70 + 80) /3
= **70**

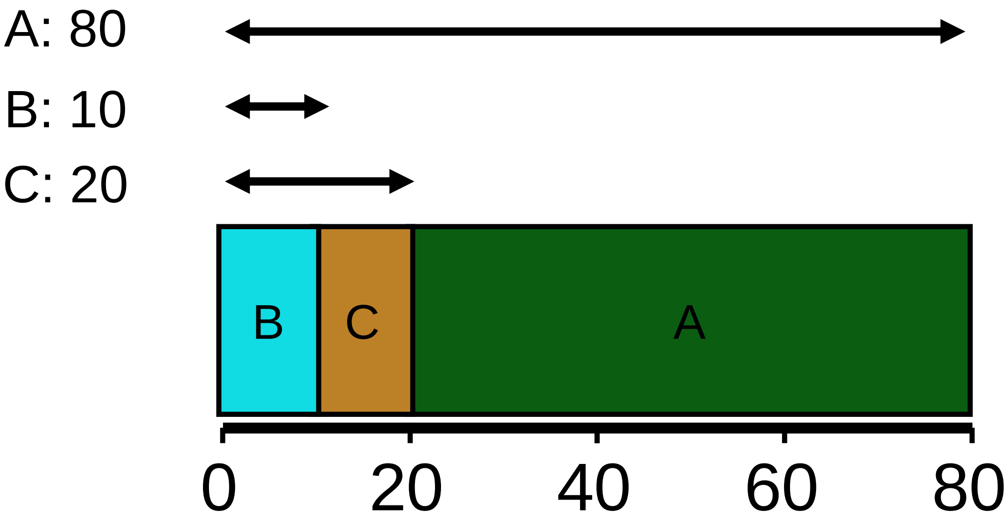
Convoy Effect

Passing the Tractor

- Problem with Previous Scheduler:
  - FIFO: Turnaround time can suffer when short jobs must wait for long jobs
- New scheduler:
  - SJF (Shortest Job First)
  - Choose job with smallest *run_time* to run first

SJF Turnaround Time

JOB	arrival_time (s)	run_time
A	~0	**60**
B	~0	10
C	~0	10



Avg. turnaround
= (10 + 20 + 80) /3
= **36.7**

SJF Turnaround Time

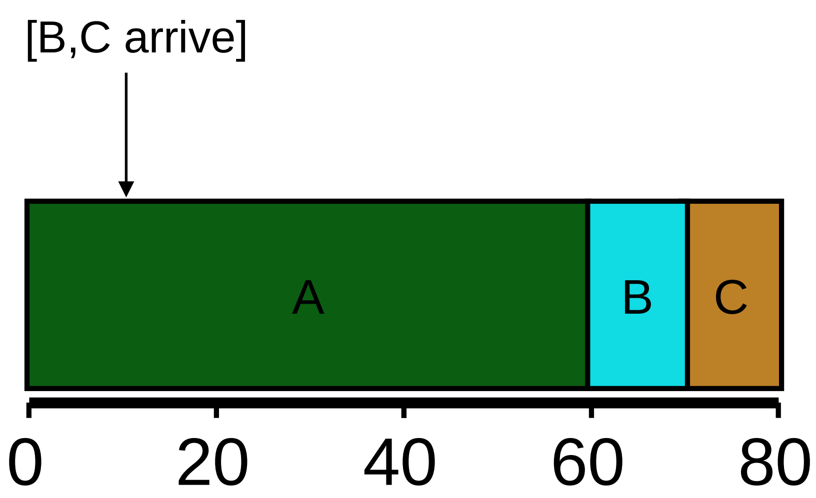
- SJF is provably optimal to minimize avg. turnaround time
  - Under current workload assumptions
  - Without preemption
- Intuition: moving shorter job before longer job improves turnaround time of short job more than it harms turnaround time of long job

More Realistic Workload Assumptions

- Workload Assumptions
  1) ~~Each job runs for the same amount of time~~
  2) ~~All jobs arrive at the same time~~
  3) Run-time of each job is known
- Any problematic workload for SJF with new assumptions?

SJF: Different Arrival Times

JOB	arrival_time (s)	run_time
A	~0	60
B	~10	10
C	~10	10



Avg. turnaround
= (60 + (70-10) + (80-10)) /3
= **63.3**

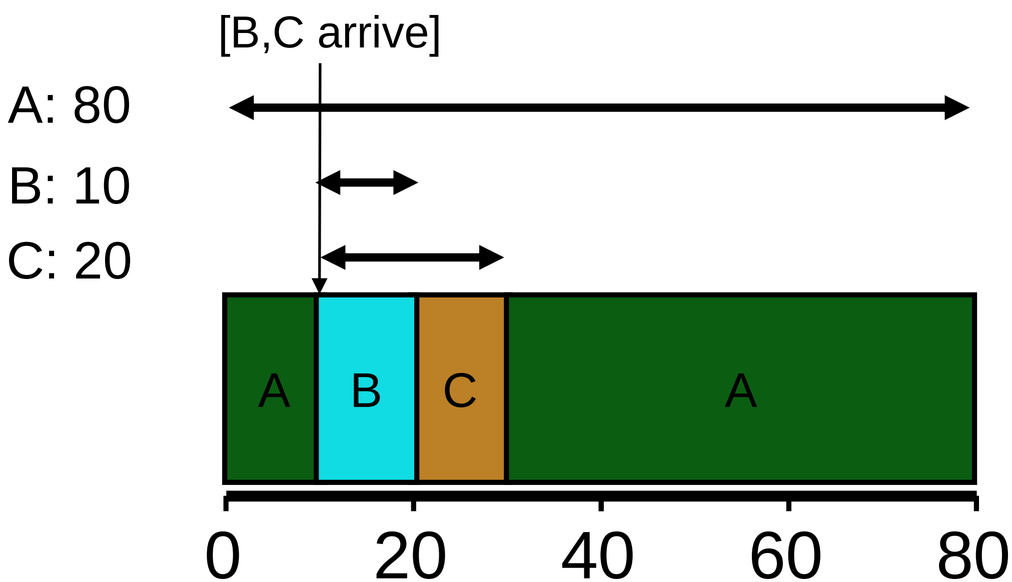
Can we do better than this?

Preemptive Scheduling

- Previous schedulers:
  - FIFO and SJF are cooperative schedulers
  - Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)
- New scheduler:
  - Preemptive: potentially schedule different job at any point by taking CPU away from running job
- STCF (Shortest Time-to-Completion First)
  - Always run job that will complete the quickest

Preemptive: STCF

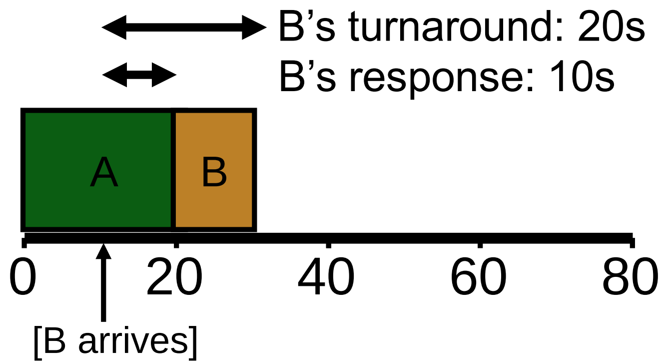
JOB	arrival_time (s)	run_time
A	~0	60
B	~10	10
C	~10	10



Avg. turnaround
= (80 + (20-10) + (30-10)) /3
= **36.6**
vs.
SJF's time of **63.3**

How about Other Metrics?

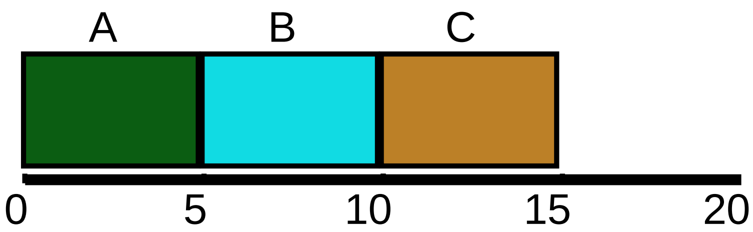
- Is turnaround time the only metric we care about?
- What about responsiveness?
  - Do you like to stare at your monitor for 10 seconds after pressing a key waiting for something to happen?
- New metric: **Response Time**
  - Job Response Time: *first_start_time – arrival_time*
  - I.e., the time that it takes for a new job to start running



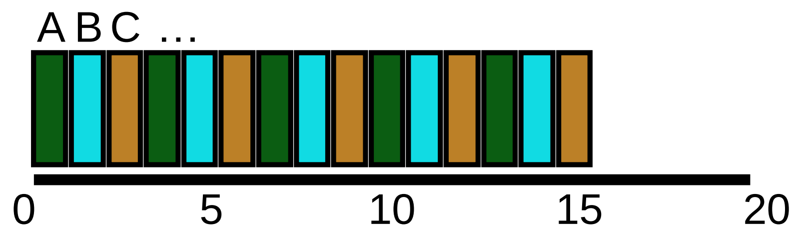
Round-Robin (RR) Scheduler

- Previous schedulers:
  - FIFO, SJF, and STCF can have poor response time
- New scheduler: RR (Round Robin)
  - Alternate ready threads every fixed-length time-slice
  - Preempt current thread at the end of its time-slice and schedule the next one in a fixed order

FIFO vs. RR



Avg. response time
= (0 + 5 + 10) / 3 = **5**



Avg. response time
= (0 + 1 + 2) / 3 = **1**

- In what way is RR worse?
  - Avg. turnaround time with equal job lengths is horrible
- *c'est la vie*
  - Impossible to optimize all metrics simultaneously
  - Try to strike a balance that works well most of the time

More Realistic Workload Assumptions

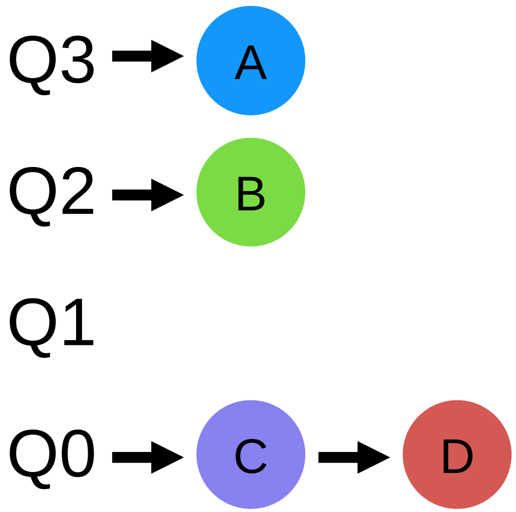
- Workload Assumptions
  1) ~~Each job runs for the same amount of time~~
  2) ~~All jobs arrive at the same time~~
  3) ~~Run-time of each job is known~~
- In practice, the OS cannot know how long a job is going to need the CPU before it completes
  - Not just the OS; Even programmer is unlikely to know it
- Need a smarter scheduler that does not rely on knowing job run-times

MLFQ: Multi-Level Feedback Queue

- Goal: general-purpose scheduling
- Must support two job types with distinct goals
  - Interactive programs care about response time
    - Example: text editor, shell, etc.
  - Batch programs care about turnaround time
    - Example: video encoder
- Approach: multiple levels of round-robin
  - Each level has higher priority than lower levels and preempts them

Priorities

- Rule 1: If priority(A) > priority(B), A runs
- Rule 2: If priority(A) == priority(B), A & B run in RR



- Multi-level
- How to know how to set priority?
  - Answer: use history "feedback"

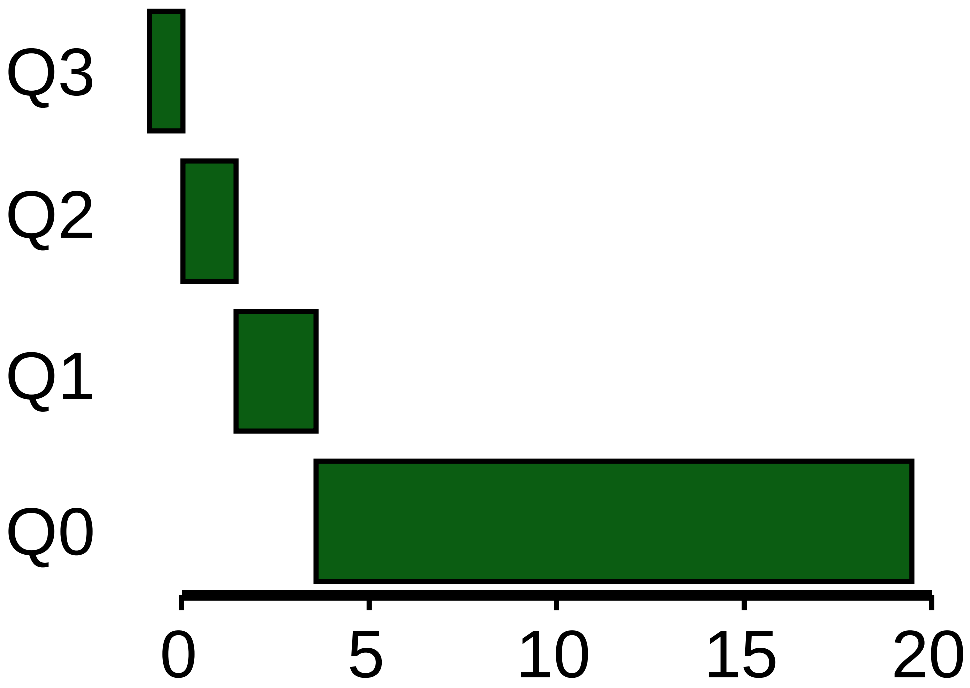
History

- Use past behavior to predict future behavior
  - Common technique in computer systems
- Threads alternate between CPU work and blocking operations (e.g., I/O)
  - Guess how next CPU burst (job) will behave based on past CPU bursts (jobs) of this thread

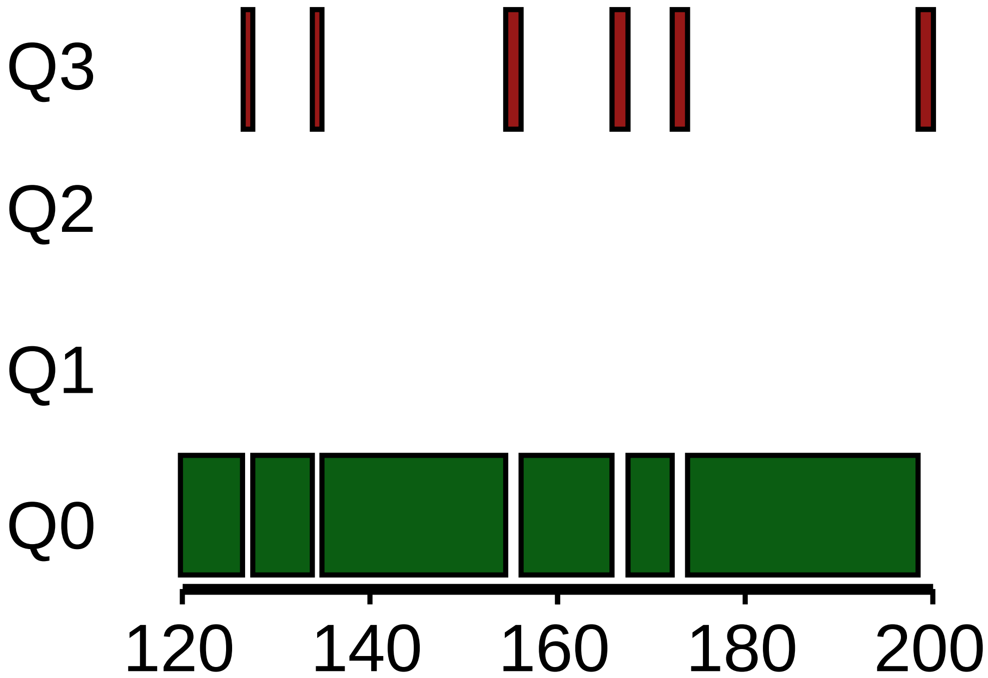
More MLFQ Rules

- Rule 1: If priority(A) > Priority(B), A runs
- Rule 2: If priority(A) == Priority(B), A & B run in RR
- Rule 3: Threads start at top priority
- Rule 4: If job uses whole time-slice, demote thread to lower priority
  - Longer time slices at lower priorities to accommodate CPU-bound applications

Example: One Long Job



An Interactive Process Joins



- Interactive process seldom uses entire time slice, so not typically demoted

Problems with MLFQ

1) Starvation
   - Too many interactive (high-priority) threads can monopolize the CPU and starve lower-priority threads
2) It is unforgiving: once demoted to lower priority, thread stays there
   - But programs may change behavior over time
     - I/O bound at some point and CPU-bound later
3) Devious programmers can game the system
   - Relinquish the CPU right before the time-slice ends
     - Never demoted; always high priority

Solutions

- Prevent starvation: periodically boost all priorities (i.e., move all threads to highest-priority queue)
  - Also takes care of unforgiving-ness
  - New Problem: how to set the boosting period?
- Prevent gaming: fix the total amount of time each thread stays at a priority level
  - I.e., do not forget about previous time-slices
  - Demote when exceed threshold
  - New Problem: how to set the threshold?
  - New Problem: has to keep more per-thread state

New Metric: Fairness

- So far, we've considered two metrics
  - Turnaround time
  - Response time
- We've seen it's impossible to minimize both simultaneously
  - We settled for a compromise: reduce response time for interactive apps and lower turnaround time for batch jobs
- But there always many jobs in the systems. What if we want them to be treated "fairly"?

Fairness

- Definition: each jobs' turnaround time should be proportional to its length (i.e., the CPU time it needs)
- Turnaround time

  = job length + time in ready queue

  = time in "Running" state + time in "Ready" state
- Therefore, fairness means amount of time a job spends in "Ready" state should be proportional to its length

Fairness (cont'd)

- Is FIFO fair?
  - No
- Is SJF fair? How about STCF?
  - No, No
- How about RR?
  - Yes, but too naïve.
  - Does not support priorities, low response time for interactive jobs, etc.
- How about MLFQ?
  - No, but boosting prevents starvation which means some attention to fairness
- There are a class of scheduling disciplines that make fairness their main goal, while paying attention to other goals such as responsiveness and priorities
  - Lottery scheduling, stride scheduling and Linux's Completely Fair Scheduler (CFS)
- Read more about them in OSTEP, chapter 9.

Linux O(1) Scheduler

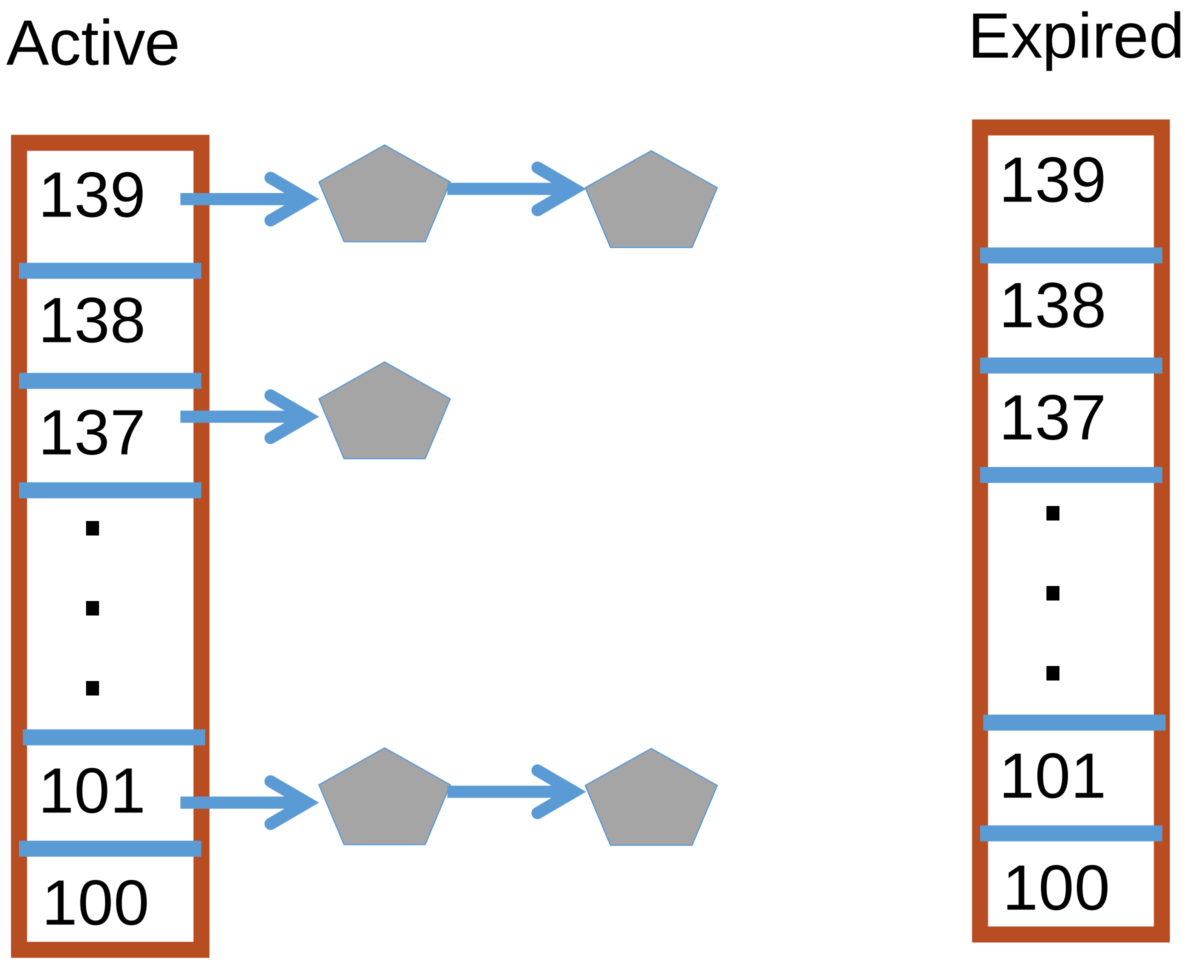
Linux O(1) Scheduler

- Think of it as a variation of MLFQ
- Goals
  - Provide good response time for short interactive jobs
  - Provide good turnaround time for long CPU-bound jobs
  - Provide a mechanism for static priority assignment
  - Be simple to implement and efficient to run
  - Etc.

O(1) Bookkeeping

- ***task***: Linux kernel lingo for thread
- ***runqueue***: a list of runnable tasks
  - Blocked threads are not on any runqueue
    - They are on some wait queue elsewhere
  - Each runqueue belongs to a specific CPU
  - Each task is on exactly one runqueue
    - Task only scheduled on runqueue's CPU unless migrated
- 2 × 40 × #CPUs runqueues
  - 40 dynamic priority levels (more later)
  - 2 sets of runqueues: <u>active</u> and <u>expired</u>

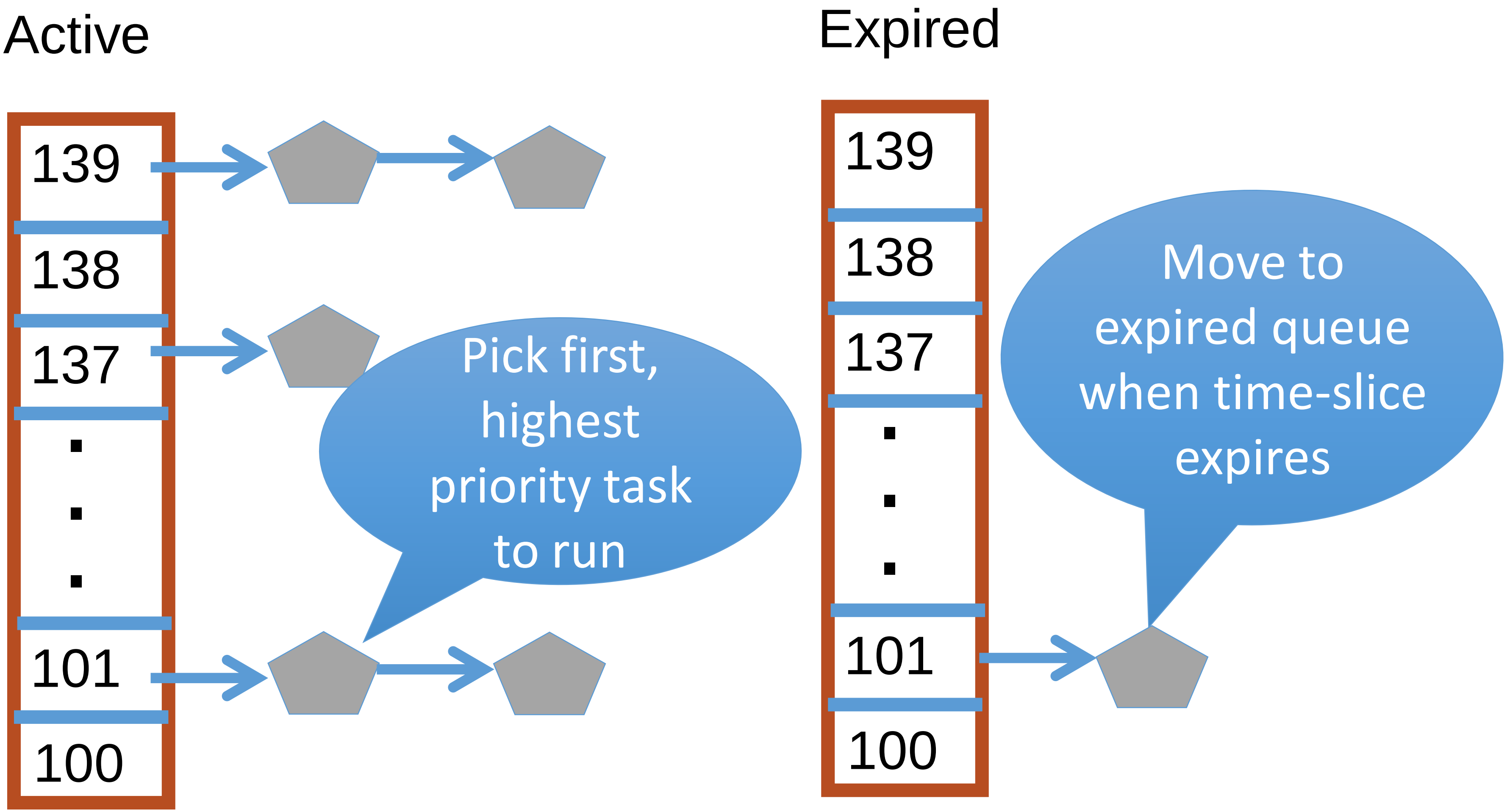
O(1) Data Structures



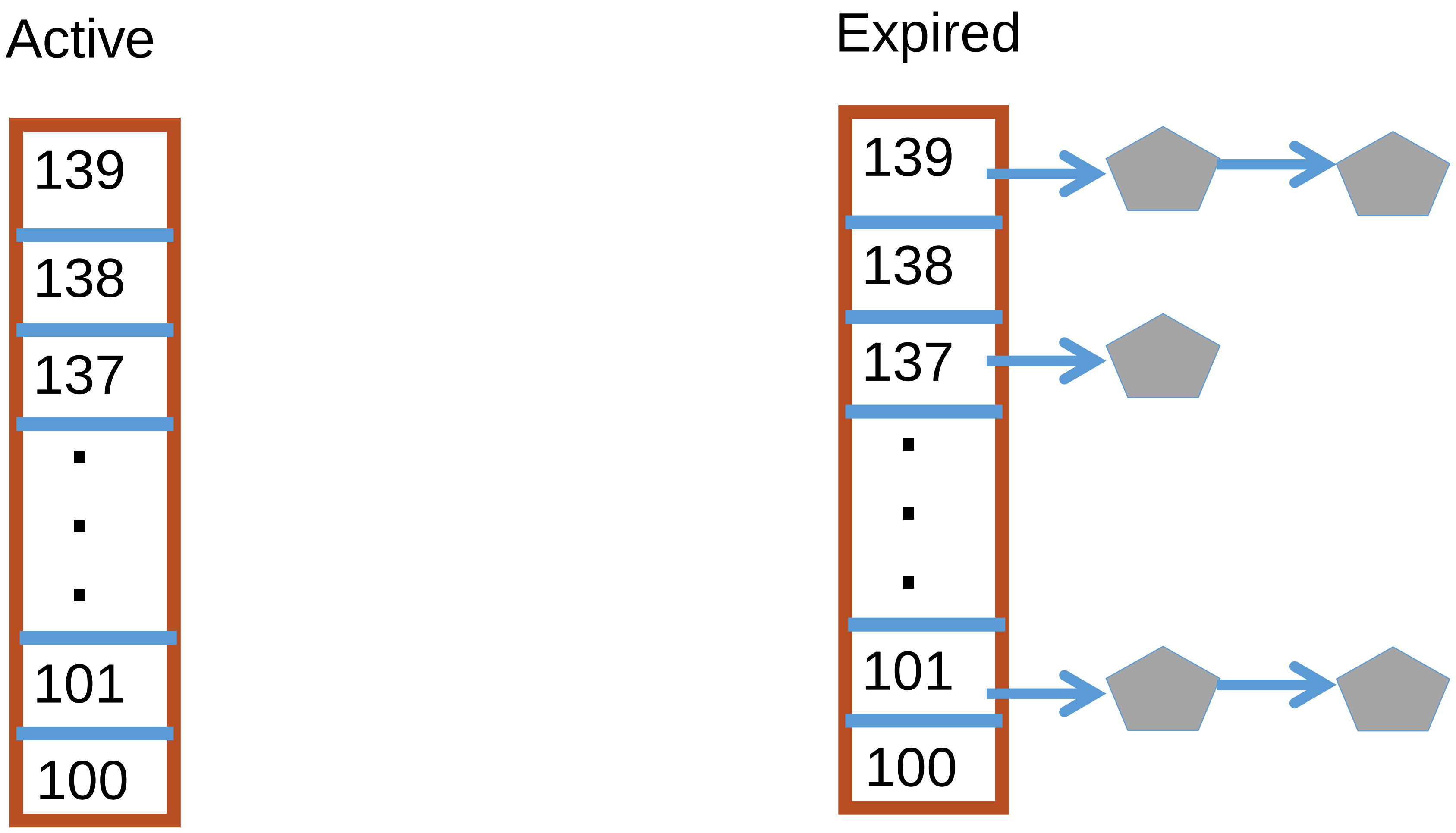
O(1) Intuition

- Take first task from highest-priority runqueue on active set
- When done, put it on runqueue on expired set
- When active set empty, swap active and expired runqueues
- Constant time: O(1)
  - Fixed number of queues to check
  - Only take first item from non-empty queue

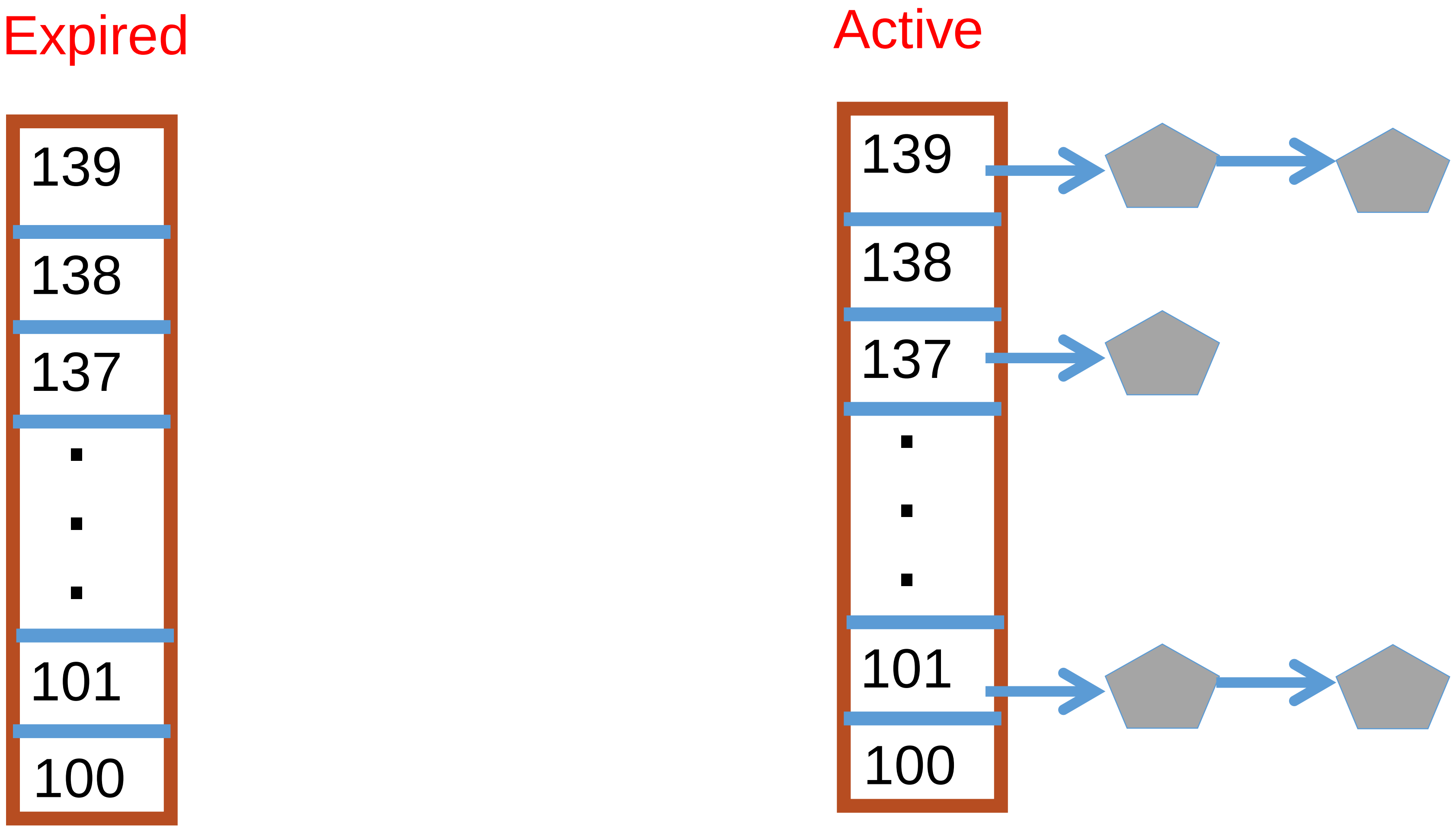
O(1) Example


Active
139
138
137
101
100
Pick first, highest priority task to run
Expired
139
138
137
101
100
Move to expired queue when time-slice expires

What Now?



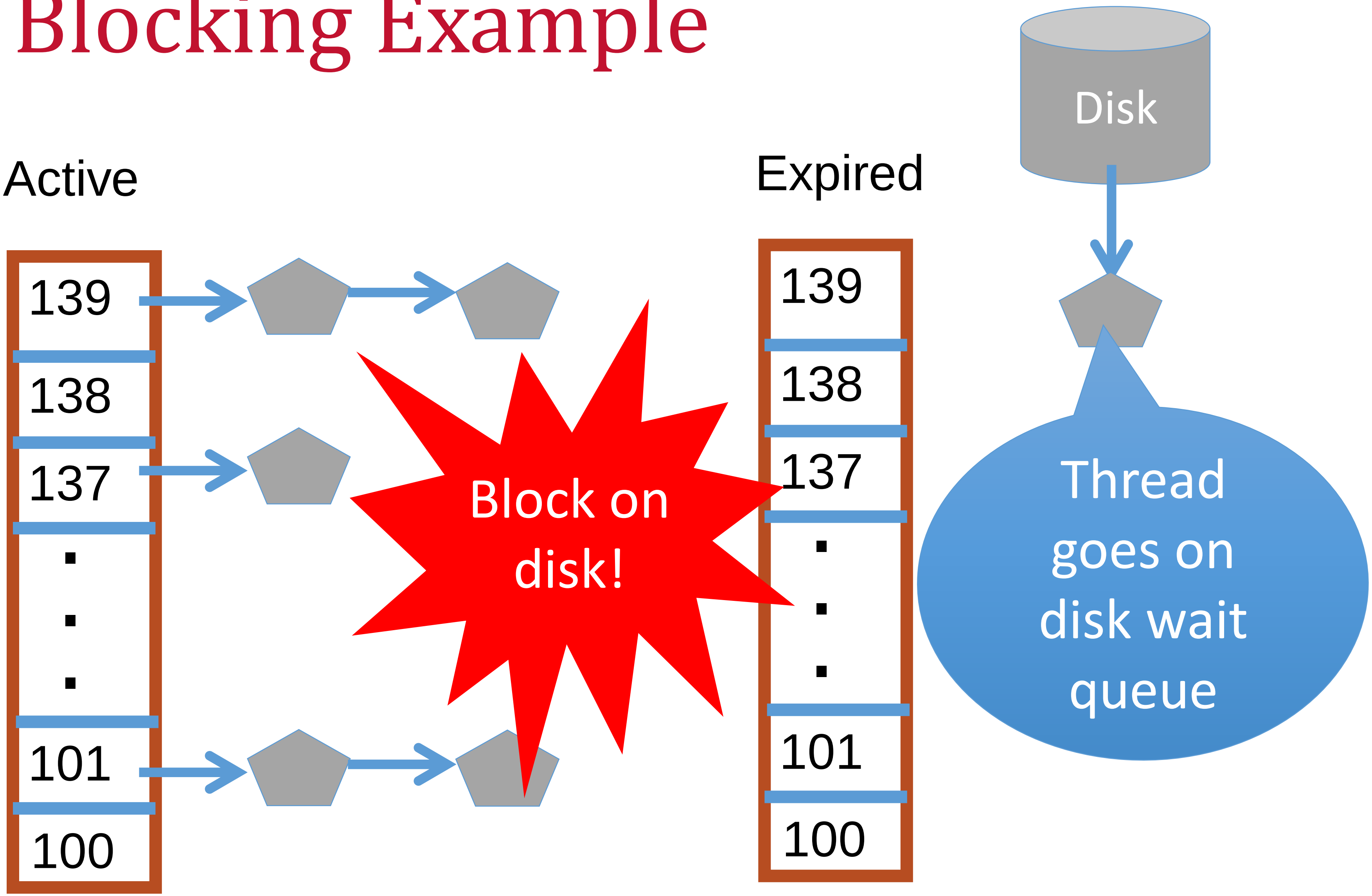
What Now?



Blocked Tasks

- What if a thread blocks, say on I/O?
  - It still has part of its quantum left
  - Not runnable
    - Don’t put on the active or expired runqueues
- Need a “wait queue” for each blocking event
  - Disk, lock, pipe, network socket, etc…

Blocking Example



Blocked Tasks (cont.)

- A blocked task is moved to a wait queue
  - Moved back to active queue when expected event happens
  - No longer on any active or expired queue!
- Disk example:
  - I/O finishes, IRQ handler puts task on active runqueue

Time Slice Tracking

- A task 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
  - When unblocked, put on active queue

More on Priorities

- 100 = highest priority
- 139 = lowest priority
- 120 = base priority
  - "nice" value: user-specified adjustment to base priority
  - Set using `nice()` system call
  - Selfish (not nice) = -20 (I want to go first)
  - Really nice = +19 (I will go last)

Base Time Slice

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

- “Higher” priority tasks get longer time slices (unlike MLFQ)
  - In addition to running first

How to Make Interactive Jobs Responsive?

- By definition, interactive applications wait on I/O a lot
  - Wait for next keyboard or mouse input, do a bit of work, wait for the next input, and so on
- Monitor I/O wait time
  - Infer which programs are UI (and disk intensive)
- Give these threads a dynamic priority boost
- Note that this behavior can be dynamic
  - Example: DVD Ripper
    - UI configures DVD ripping
    - Then it is CPU bound to encode to mp3

  → Scheduling should match program phases

Dynamic Priority

- Dynamic priority
  = *max*(100, *min*(*static_priority* − ***bonus*** + 5, 139))
- ***Bonus*** is calculated based on wait time
- <u>Dynamic priority </u>determines a task's runqueue
- Tries to balance throughput for CPU-bound programs and latency for IO-bound ones
  - May not be optimal
- Call it what you prefer
  - Carefully-studied battle-tested heuristic
  - Horrible hack that seems to work

Dynamic Priority in O(1) Scheduler

- runqueue determined by the dynamic priority
  - Not the static priority
  - Dynamic priority mostly based on time spent waiting
    - To boost UI responsiveness
- “Nice” values influence static priority
  - Can’t boost dynamic priority without being in wait queue!
  - No matter how “nice” you are or aren't

Linux's Completely Fair Scheduler (CFS)

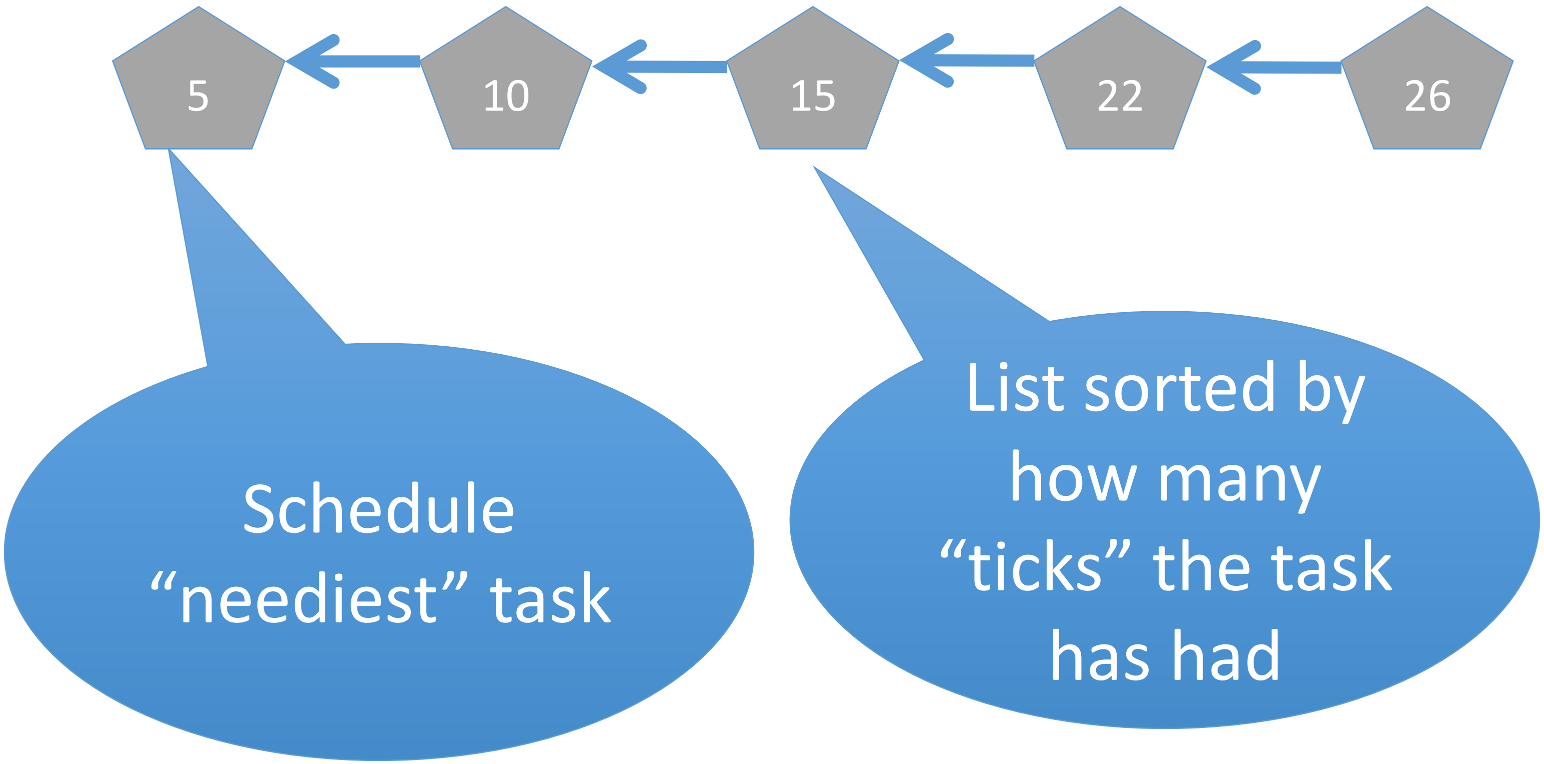
Fair Scheduling

- Idea: 50 tasks of equal length, each should get 2% of CPU time
- Is this all we want?
  - What about priorities?
  - Responsive interactive jobs?
  - Per-user fairness?
    - Alice has 1 task and Bob has 49; why should Bob get 98% of CPU?
- ***Completely Fair Scheduler (CFS)***
  - Default Linux scheduler since 2.6.23

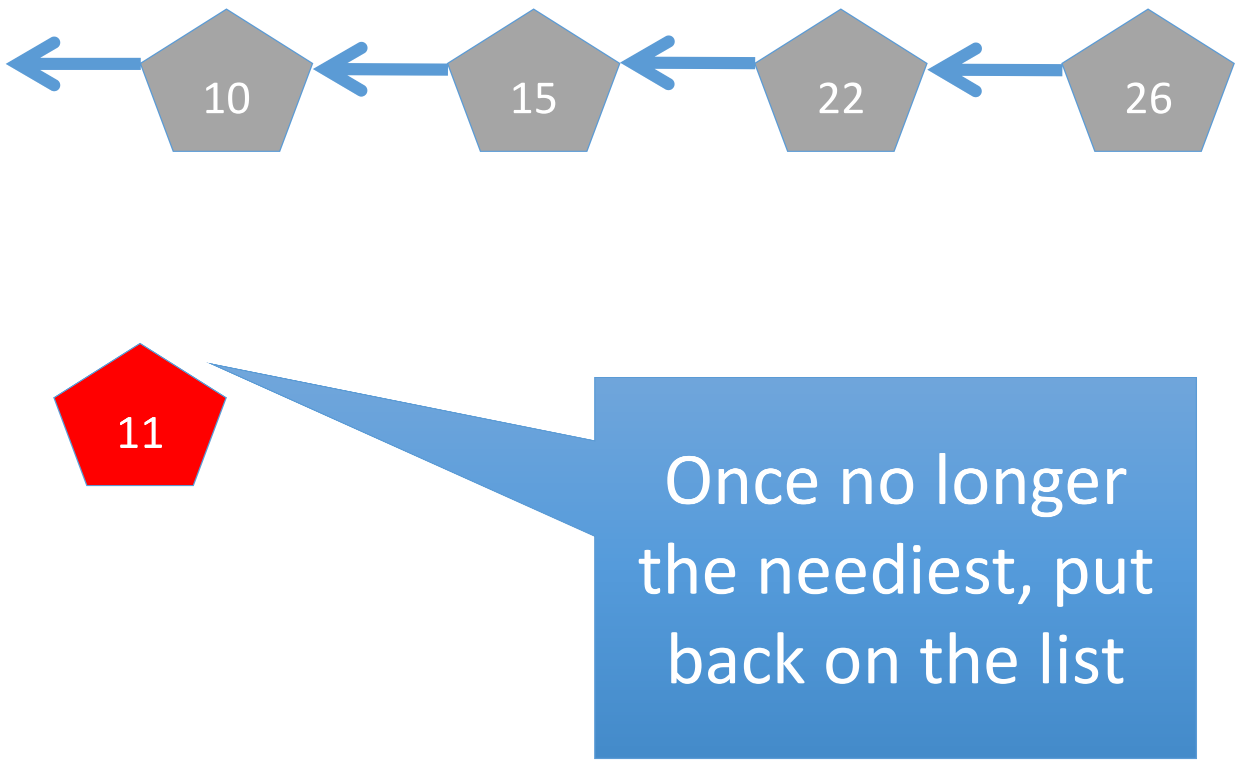
CFS idea

- Back to a simple list of tasks (conceptually)
- Ordered by how much time they have had
    - Least time to most time
- Always pick the “neediest” task to run
    - Until it is no longer neediest
    - Then re-insert old task in the timeline
    - Schedule the new neediest

CFS Example


5
10
15
22
26
Schedule "neediest" task
List sorted by how many "ticks" the task has had

CFS Example



But Lists Are Inefficient

- That's why we really use a tree
  - Red-black tree: 9/10 Linux developers recommend it
- log(n) time for:
  - Picking next task (i.e., search for left-most task)
  - Putting the task back when it is done (i.e., insertion)
  - Remember: n is total number of tasks on system

Details

- ***Global Virtual Clock (global vclock)***: ticks at a fraction of real time
  - fraction = number of total tasks

  → Indicates "Fair" share of each task

- Each task counts how many clock ticks it has had

- Example: 4 tasks
  - Global vclock ticks once every 4 real ticks
  - Each task scheduled for one real tick
    - Advances local clock by one real tick

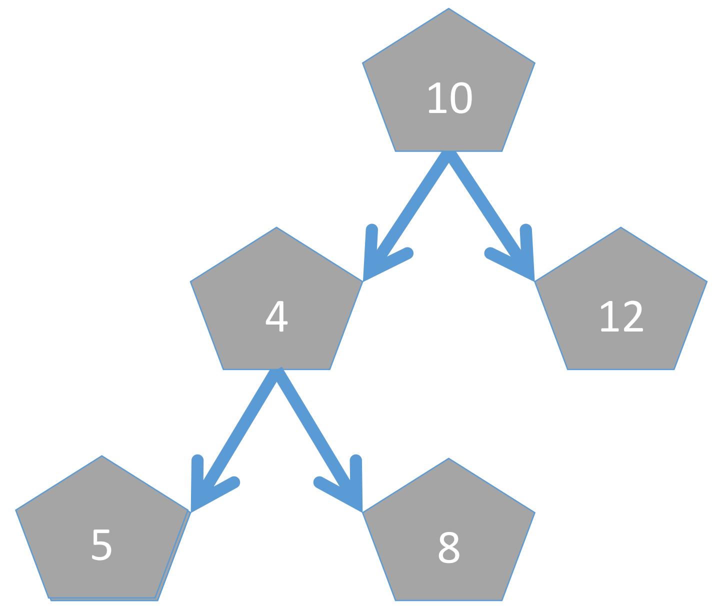
More Details

- Task’s ticks make key in RB-tree
  - Lowest tick count gets serviced first
- No more runqueues
  - Just a single tree-structured timeline

CFS Example (more realistic)

- Tasks sorted by ticks executed
- One global tick per n ticks
  - n == number of tasks (5)
- 4 ticks for first task
- Reinsert into list
- 1 tick to new first task
- Increment global clock

Global Ticks: 8



Why a Global Virtual Clock?

- What to do when a new task arrives?
  - If task ticks start at zero, unfair to run for a long time
- Strategies:
  - Could initialize to current Global Ticks
  - Could get half of parent's deficit

What about Priorities?

- Priorities let me be deliberately unfair
  - This is a useful feature
- In CFS, priorities weigh the length of a task's "local tick"
  - Local Virtual Clock
- Example:
  - For a high-priority task
    - A task-local tick may last for 10 actual clock ticks
  - For a low-priority task
    - A task-local tick may only last for 1 actual clock tick
- Higher-priority tasks run longer
- Low-priority tasks make some progress

10:1 ratio is made-up.
See code for real weights.

What about Interactive Apps?

- Recall: UI programs are I/O bound
  - We want them to be responsive to user input
  - Need to be scheduled as soon as input is available
  - Will only run for a short time

CFS and Interactive Apps

- Blocked tasks removed from RB-tree
  - Just like O(1) scheduler
- Global vclock keeps ticking while tasks are blocked
  - Increasingly large deficit between task and global vclock
- When a GUI task is runnable, goes to the front
  - Dramatically lower local-clock value than CPU-bound jobs

Other Refinements

- Per task group or user scheduling
  - Controlled by real to virtual tick ratio
    - Function of number of global and user's/group's tasks

Recap: Different Types of Ticks

- Real time is measured by a timer device
  - “ticks” at a certain frequency by raising a timer interrupt every so often
- A thread’s local virtual tick is some number of real ticks
  - Priorities, per-user fairness, etc... done by tuning this ratio
- Global Ticks tracks the fair share of each process
  - Used to calculate one’s deficit

CFS Summary

- Idea: logically a single queue of runnable tasks
  - Ordered by who has had the least CPU time
- Implemented with a tree for fast lookup
- Global clock counts virtual ticks
  - One tick per “task_count” real ticks
- Features/tweaks (e.g., prio) are hacks
  - Implemented by playing games with length of a virtual tick
  - Virtual ticks vary in wall-clock length per-process

Other Issues

Real-time Scheduling

- Different model
  - Must do modest amount of work by a deadline
- Example: audio application must deliver one frame every *n* ms
  - Too many or too few frames unpleasant to hear
- Strawman solution
  - If I know it takes *n* ticks to process a frame of audio, schedule my application n ticks before the deadline
- Problem? hard to accurately estimate *n*
  - Variable execution time depending on inputs
  - Interrupts
  - Cache misses
  - TLB misses
  - Disk accesses

Hard Problem

- Gets even harder w/ multiple applications + deadlines
- May not be able to meet all deadlines
- Shared data structures worsen variability
  - Block on locks held by other tasks

Linux Hack

- Have different scheduling classes (disciplines):
  - ***SCHED_IDLE***, ***SCHED_BATCH***, ***SCHED_OTHER***, ***SCHED_RR***, ***SCHED_FIFO***
- "Normal" tasks are in *SCHED_OTHER*
- "Real-time" tasks get highest-priority scheduling class
  - *SCHED_RR* and *SCHED_FIFO* (RR: round robin)
  - RR is preemptive, FIFO is cooperative
- RR tasks fairly divide CPU time amongst themselves
  - Pray that it is enough to meet deadlines
  - Other tasks share the left-overs (if any) and may starve
- Assumption: RR tasks mostly blocked on I/O (like GUI programs)
  - Latency is the key concern
- New real-time scheduling class since Linux 3.14: ***SCHED_DEADLINE***
  - Highest priority class in system; Uses "Earliest Deadline First" scheduling
  - Details in http://man7.org/linux/man-pages/man7/sched.7.html

Linux Scheduling-Related API

- Includes many functions to set scheduling classes, priorities, processor affinities, yielding, etc.

- See http://man7.org/linux/man-pages/man7/sched.7.html for a detailed discussion

Next Issue: Average Load

- How do we measure how "busy" a CPU is?
  - Useful, e.g., when an idle CPU wants to "steal" threads from another CPU
    - Should steal from the busiest CPU
- Average number of <u>runnable</u> tasks over time
- Available in /proc/loadavg

Next Issue: Kernel Time

- Context switches generally at user/kernel boundary
  - Or on blocking I/O operations
- System call times vary
- Problems: if a time slice expires inside of a system call:
  1) Task gets rest of system call "for free"
     - Steals from next task
  2) Potentially delays interactive/real-time tasks until finished

Idea: Kernel Preemption

- Why not preempt system calls just like user code?
- Well, because it is harder, duh!
- Why?
  - May hold a lock that other tasks need to make progress
  - May be in a sequence of HW config operations
    - Usually assumes sequence won't be interrupted
- General strategy: allow fragile code to disable preemption
  - Like interrupt handlers disabling interrupts if needed

Kernel Preemption

- Implementation: actually not too bad
  - Essentially, it is transparently disabled with any locks held
  - A few other places disabled by hand
- Result: UI programs a bit more responsive