

238P: Operating Systems

Lecture 14: Process scheduling

This lecture is heavily based on
the material developed by Don Porter

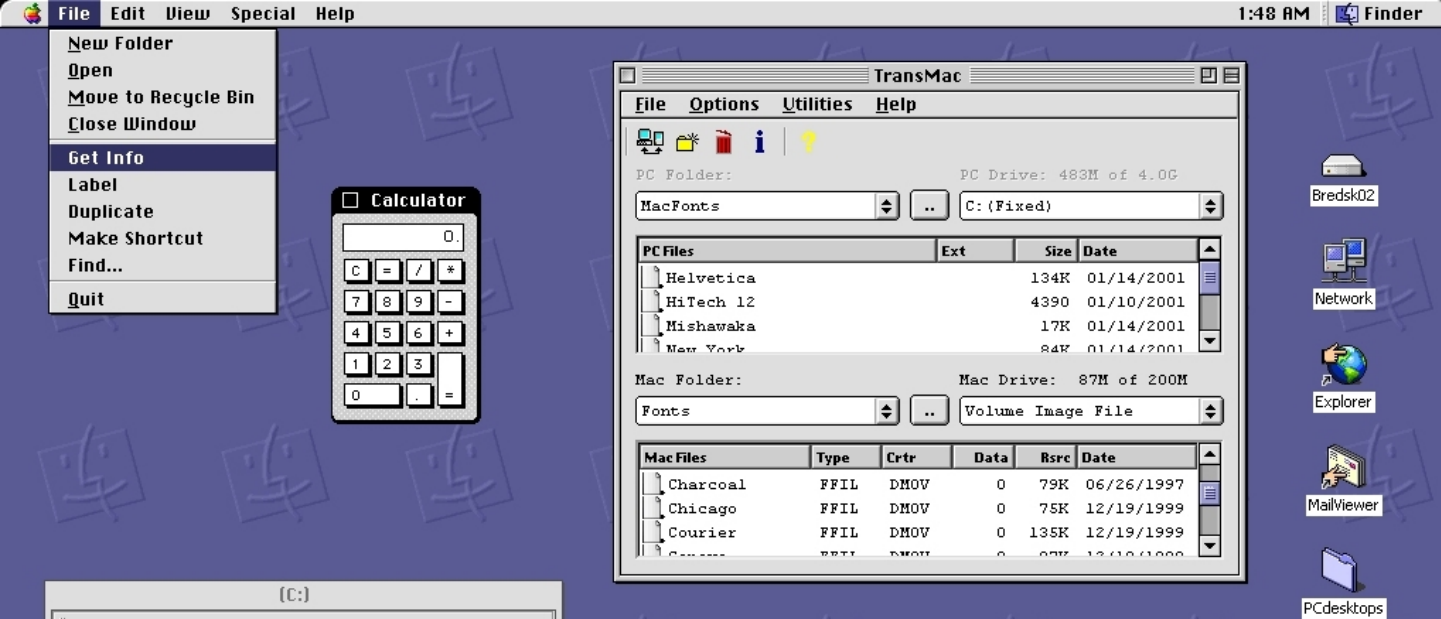
Anton Burtsev
November, 2017

Cooperative vs preemptive

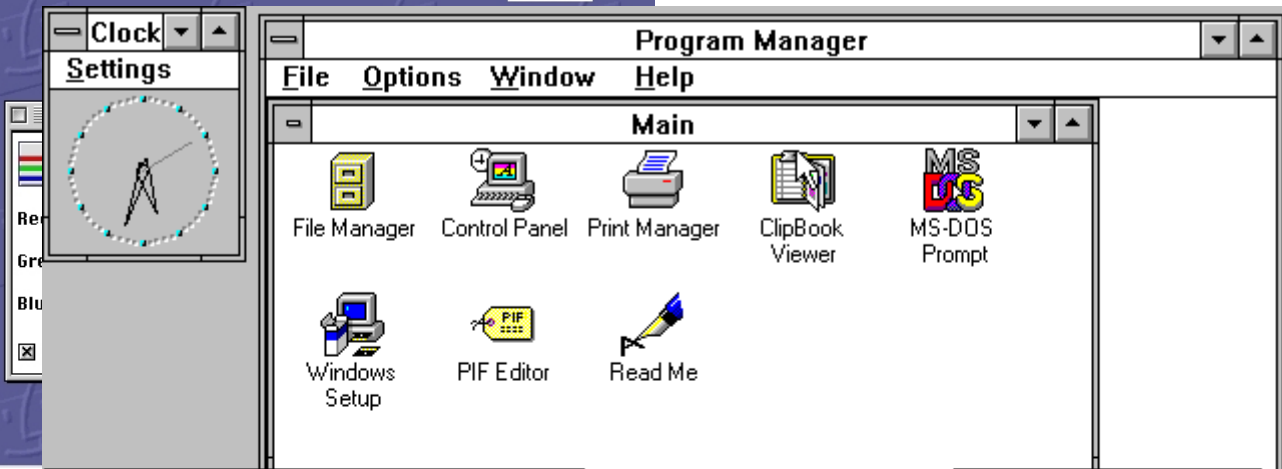
- What is cooperative multitasking?
- What is preemptive multitasking?
- Pros/cons?

Cooperative vs preemptive

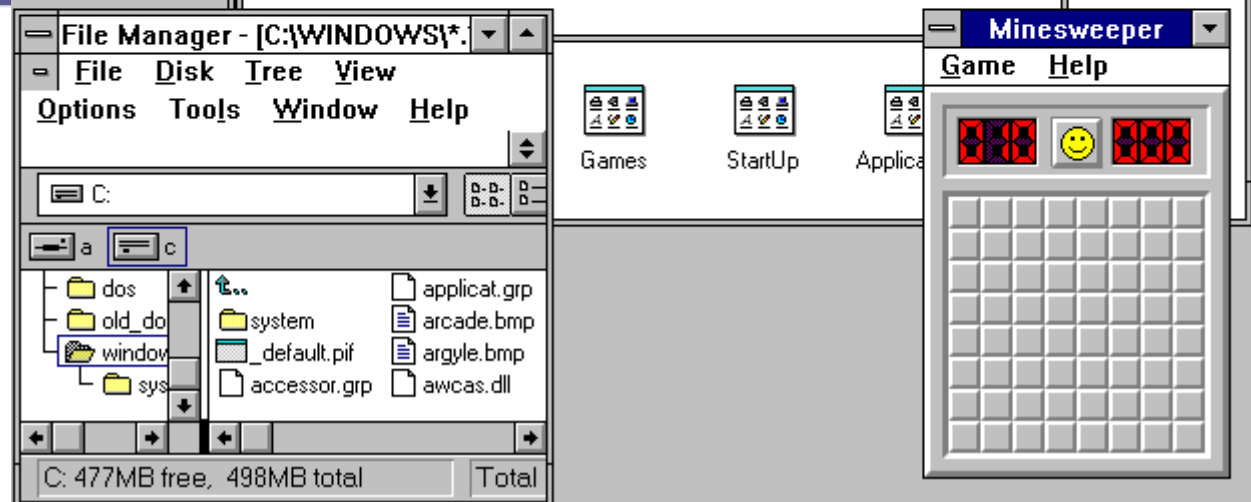
- What is cooperative multitasking?
 - Processes voluntarily yield CPU when they are done
- What is preemptive multitasking?
 - OS only lets tasks run for a limited time, then forcibly context switches the CPU
- Pros/cons?
 - Cooperative gives more control; so much that one task can hog the CPU forever
 - Preemptive gives OS more control, more overheads/complexity



- MacOS 9



- Windows 3.1



At what point process can get preempted?

At what point process can get preempted?

- When entered the kernel
 - Inside one of the system calls
- Timer interrupt
 - Ensures maximum time slice

Policy vs mechanism

- Remember we know the mechanism
 - Context switching
 - Switch stacks
- This lecture is about policy
 - Pick the next process to run

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

Strawman scheduler (xv6)

- Organize all processes as a simple list
- In `schedule()`:
 - Pick first one on list to run next
 - Put suspended task at the end of the list

Xv6 scheduler

```
2458 scheduler(void)
2459 {
2462     for(;;){
2468         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2469             if(p->state != RUNNABLE)
2470                 continue;
2475             proc = p;
2476             switchvm(p);
2477             p->state = RUNNING;
2478             swtch(&cpu->scheduler, proc->context);
2479             switchkvm();
2483             proc = 0;
2484         }
2487     }
2488 }
```

Strawman scheduler (xv6)

- Organize all processes as a simple list
- In `schedule()`:
 - Pick first one on list to run next
 - Put suspended task at the end of the list
- Problem?

Strawman scheduler

- Organize all processes as a simple list
- In `schedule()`:
 - Pick first one on list to run next
 - Put suspended task at the end of the list
- Problem?
 - Allows only round-robin scheduling
 - Can't prioritize tasks

Priority based scheduling

- Higher-priority processes run first
- Processes within the same priority are round-robin

O(1) scheduler (Linux 2.6 – 2.6.22)

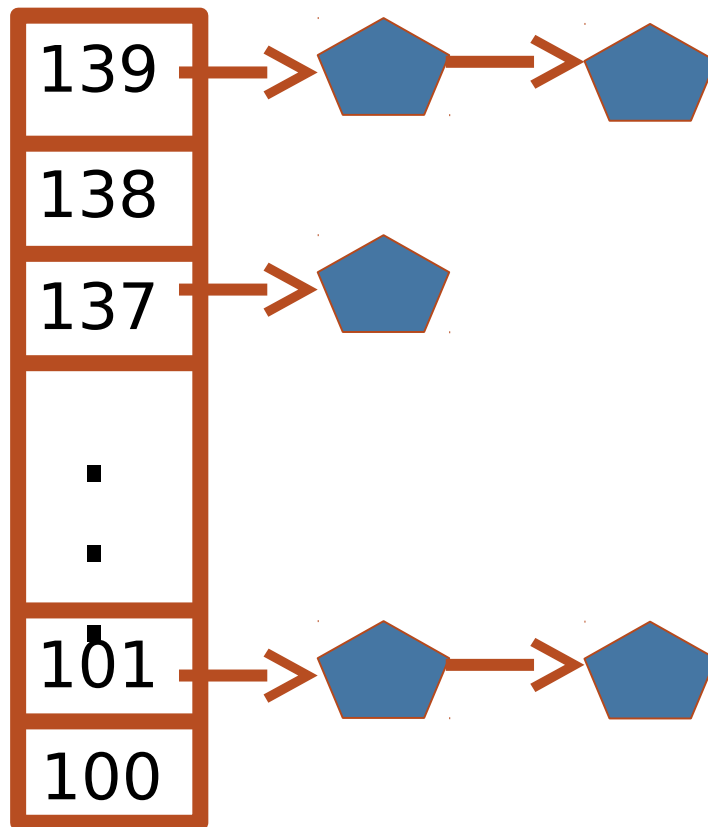
- Priority based scheduling
- Goal: decide who to run next, independent of number of processes in system
 - Still maintain ability to prioritize tasks, handle partially unused quanta, etc

O(1) data structures

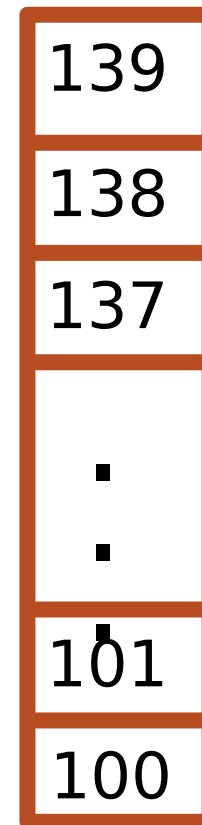
- runqueue: a list of runnable processes
 - Blocked processes are not on any runqueue
 - A runqueue belongs to a specific CPU
 - Each task is on exactly one runqueue
 - Task only scheduled on runqueue's CPU unless migrated
- $2 * 40 * \text{\#CPUs}$ runqueues
 - 40 dynamic priority levels (more later)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

$O(1)$ data structures (contd)

Active



Expired

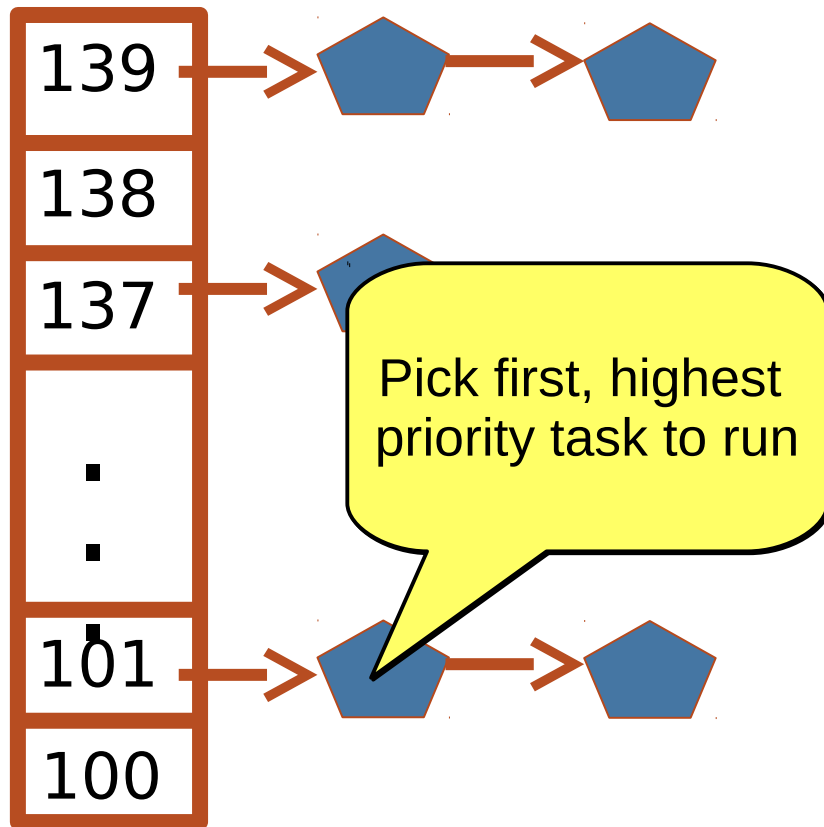


$O(1)$ intuition

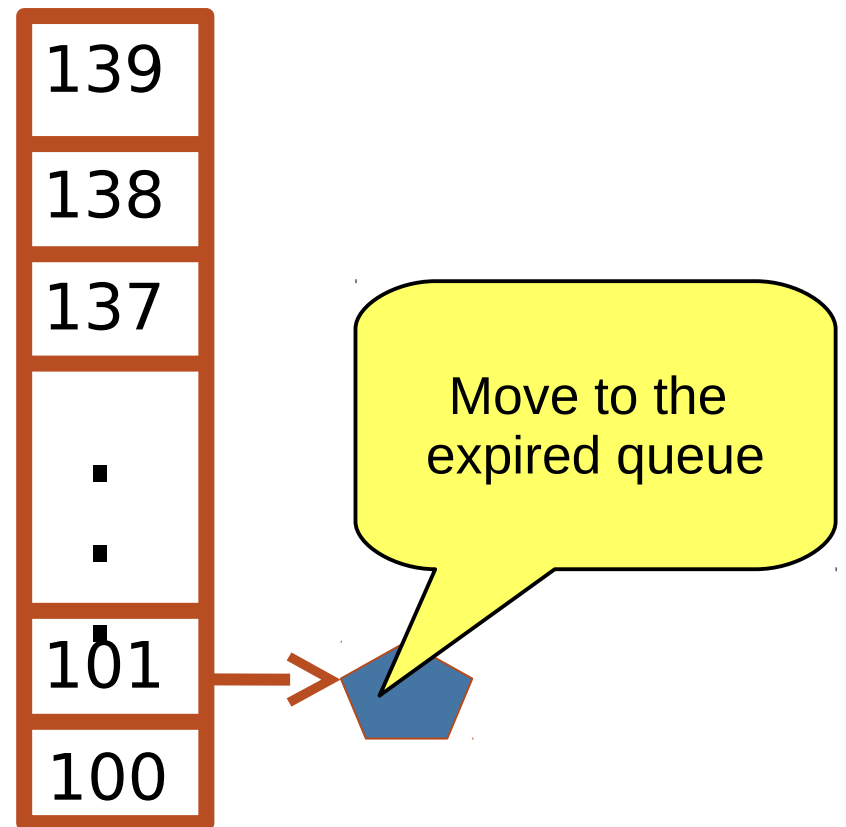
- Take the first task off the lowest-numbered runqueue on active set
 - Confusingly: a lower priority value means higher priority
- When done, put it on appropriate runqueue on expired set
- Once active is completely empty, swap which set of runqueues is active and expired
- Constant time, since fixed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

O(1) example

Active

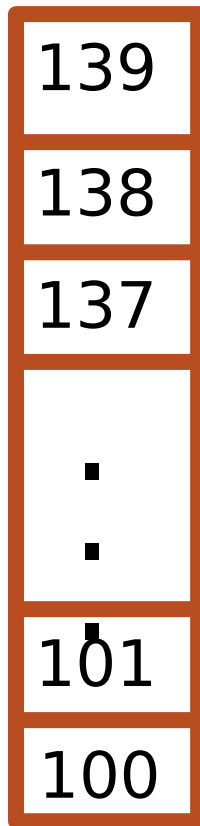


Expired



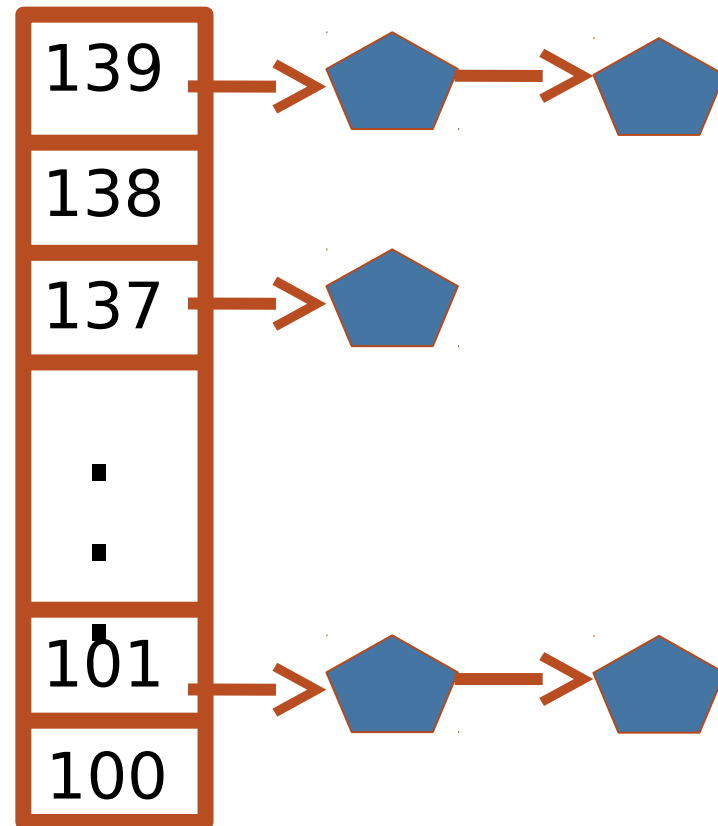
What now?

Active



Flip active and expired queues

Expired

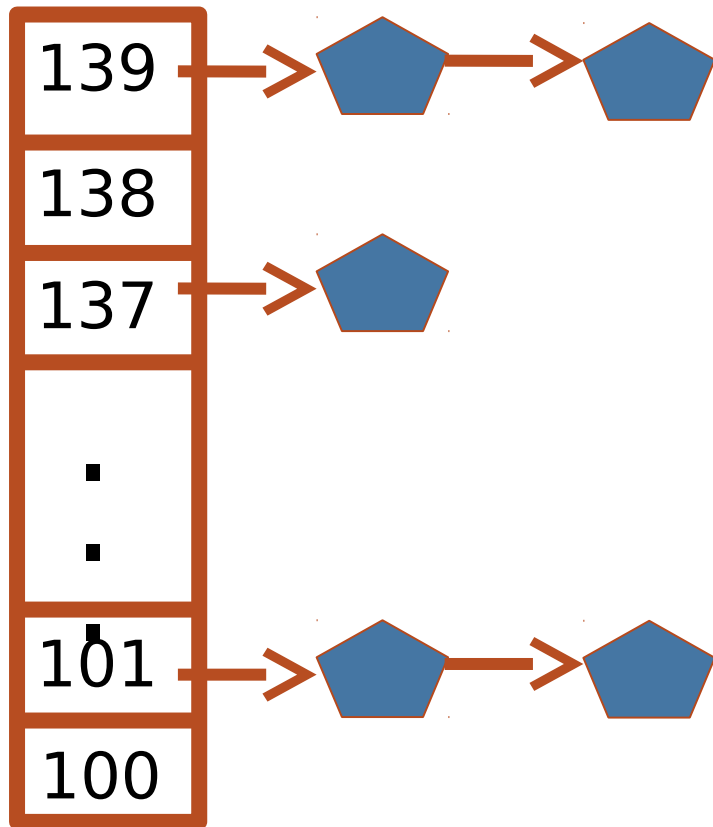


Blocked tasks

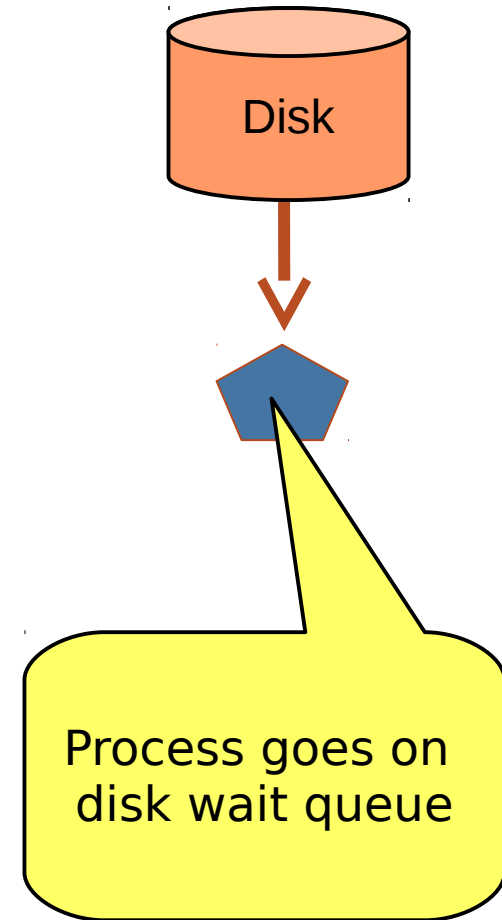
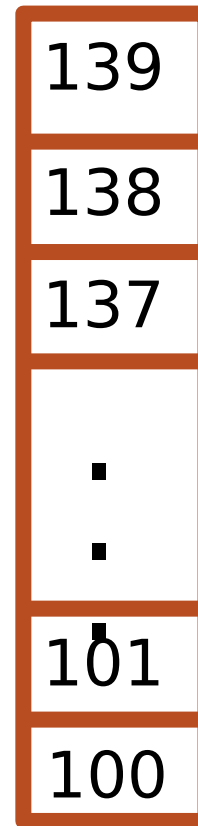
- What if a program blocks on I/O, say for the disk?
 - It still has part of its quantum left
 - Not runnable, so don't waste time putting it on the active or expired runqueues
- We need a “wait queue” associated with each blockable event
 - Disk, lock, pipe, network socket, etc.

Blocking example

Active



Expired



Blocked tasks (contd)

- 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

- 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
 - Priorities 0 – 99 are for real-time processes
- 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

- Timeslice:

 If priority < 120

 Time = (140 – prio) * 20 ms

 else

 Time = (140 – prio) * 5 ms

- “Higher” priority tasks get more time

- And run first

Responsive UI

- Most GUI programs are I/O bound
 - Wait 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 (I/O bound),
 - Then starts ripping (re-encoding into mpeg) and becomes CPU-bound
 - Scheduling should match program phases

Dynamic priority

dynamic priority =

$$\max (100, \min ((\textit{static priority} - \textit{bonus} + 5), 139))$$

- 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 $O(1)$

- 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!

Completely Fair Scheduler
Linux 2.6.23 - now

Fairness

- Each task makes proportional progress on the CPU
 - No starvation

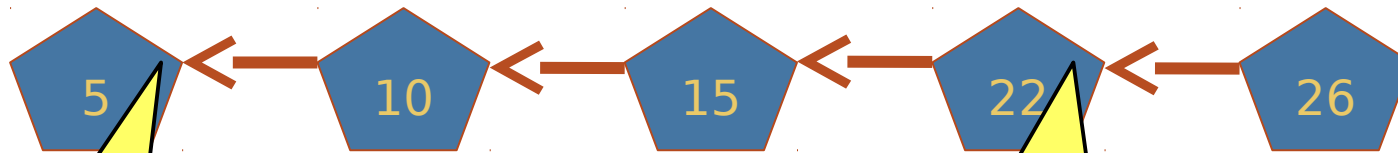
Problems with $O(1)$

- Heuristics became hard
 - Hard to maintain and make sense of

CFS idea

- Back to a simple list of tasks (conceptually)
 - Ordered by how much time they ran
 - 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

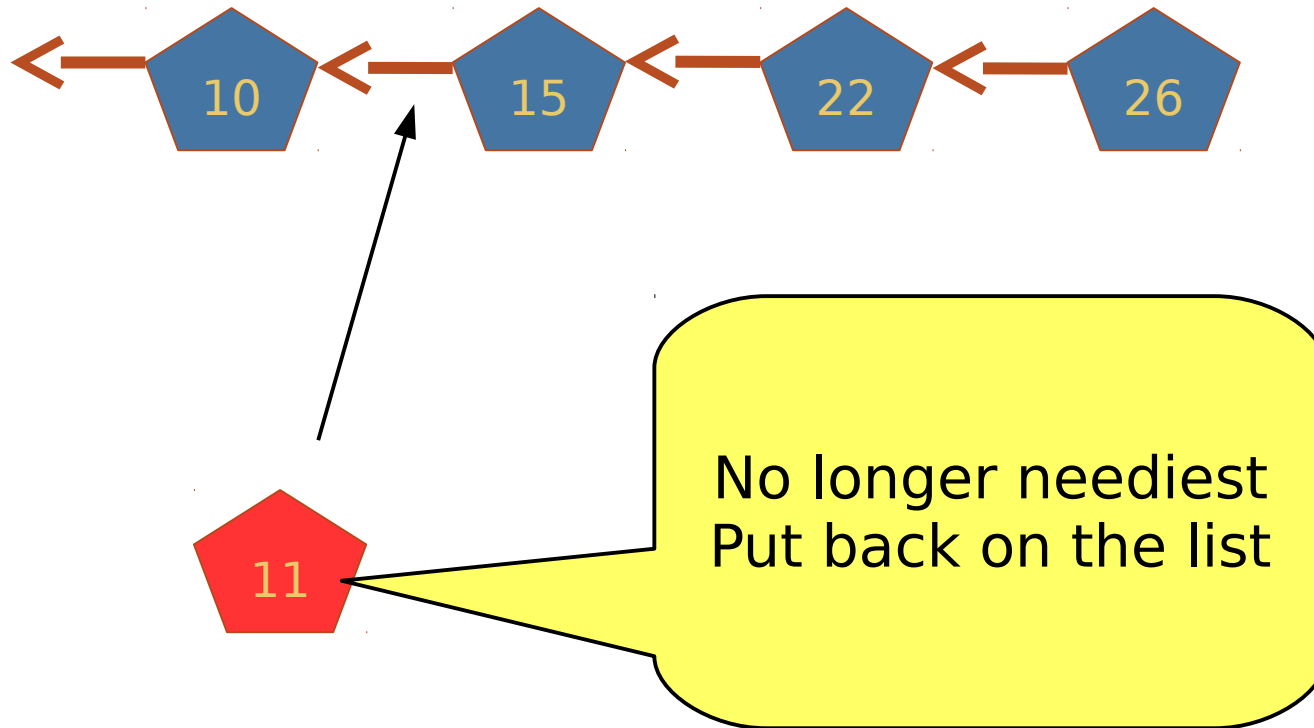
CFS example



Schedule the
neediest task

List sorted by how
many cycles the
task has had

CFS example



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

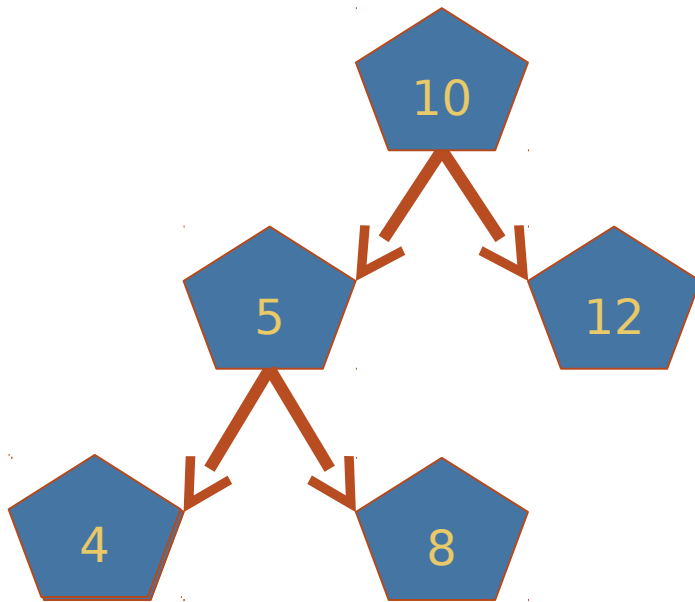
CPU time accounting

- Global virtual clock: ticks at a fraction of real time
 - Fraction is number of total tasks
- 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 tick

More details

- Task's ticks make key in RB-tree
 - Fewest tick count get serviced first
- No more runqueues
 - Just a single tree-structured timeline

CFS example



- Tasks sorted by ticks executed
- Global ticks = 12
- One global tick per n ticks
 - $n ==$ number of tasks (5)
- 4 ticks for first task
 - Reinsert into the list
- 1 tick to new first task
 - Increment global clock

New tasks

- What about a new task?
 - If task ticks start at zero, doesn't it get to unfairly run for a long time?
- Strategies:
 - Could initialize to current time (start at right)
 - Could get half of parent's deficit

Priorities

- In CFS, priorities weigh the length of a task's "tick"
- Example:
 - For a high-priority task, a virtual, task-local tick may last for 10 actual clock ticks
 - For a low-priority task, a virtual, task-local tick may only last for 1 actual clock tick
- Result: Higher-priority tasks run longer, low-priority tasks make some progress

Interactivity

- Recall: GUI 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

GUI programs

- Just like $O(1)$ scheduler, CFS takes blocked programs out of the RB-tree of runnable processes
- Virtual clock continues ticking while tasks are blocked
 - Increasingly large deficit between task and global vclock
- When a GUI task is runnable, generally goes to the front
 - Dramatically lower vclock value than CPU-bound jobs
 - Reminder: “front” is left side of tree

Other refinements

- User A has 1 job, user B has 99
 - B will get 99% of CPU time
 - We want A and B split CPU in half
- Per group or user scheduling
 - Real to virtual tick ratio becomes a function of number of both global and user's/group's tasks

Group scheduling

- Per group or user scheduling
 - Real to virtual tick ratio becomes a function of number of both global and user's/group's tasks

Real-time scheduling

Real-time scheduling

- Different model: need to do a modest amount of work by a deadline
- Example:
 - Audio application needs to deliver a frame every n th of a second
 - Too many or too few frames unpleasant to hear

Strawman

- If I know it takes n ticks to process a frame of audio, just schedule my application n ticks before the deadline
- Problems?
- Hard to accurately estimate n
 - Interrupts
 - Cache misses
 - Disk accesses
 - Variable execution time depending on inputs

Hard problem

- Gets even worse with multiple applications + deadlines
- May not be able to meet all deadlines
- Interactions through shared data structures worsen variability
 - Block on locks held by other tasks
 - Cached CPU, TLB, and file system data gets evicted

Real-time scheduling in Linux

- Linux has soft-real time scheduling
 - No hard real-time guarantees
- All real-time tasks are higher priority than any conventional process
 - Priorities 0 – 99
- Assumption: like GUI programs, RR tasks will spend most of their time blocked on I/O
 - Latency is key concern

Real-time policies

- First-in, first-out: `SCHED_FIFO`
 - Static priority
 - Process is only preempted for a higher priority process
 - No time quanta; it runs until its done, blocked or yields voluntarily
- Round robin: `SCHED_RR`
 - Same as above but with a time quanta (800ms)

Accounting kernel time

- Should time spent in the OS count against an application's time slice?
 - Yes: Time in a system call is work on behalf of that task
 - No: Time in an interrupt handler may be completing I/O for another task

Latency of system calls

- System call times vary
- Context switches are generally at system call boundary
 - Can also context switch on blocking I/O operations
- If a time slice expires inside of a system call:
 - Task gets rest of system call “for free”
 - Steals from next task
 - Potentially delays interactive/real time task until finished

Idea: kernel preemption

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

Conclusion

- $O(1)$
 - Two sets of runques
 - Each process has priority
- CFS
 - Queue of runnable tasks
 - Red/black tree for fast lookup and insertion
- Real-time
 - Run in front of $O(1)$ or CFS scheduler
 - No good solution so far

Thank you!