

Last time...



- ✦ Scheduling overview, key trade-offs, etc.
- ✦ $O(1)$ scheduler – older Linux scheduler
 - ✦ Today: Completely Fair Scheduler (CFS) – new hotness
- ✦ Other advanced scheduling issues
 - ✦ Real-time scheduling
 - ✦ Kernel preemption
 - ✦ Priority laundering
 - ✦ Security attack trick developed at Stony Brook

Fair Scheduling



- ✦ Simple idea: 50 tasks, each should get 2% of CPU time
- ✦ Do we really want this?
 - ✦ What about priorities?
 - ✦ Interactive vs. batch jobs?
 - ✦ CPU topologies?
 - ✦ Per-user fairness?
 - ✦ Alice has one task and Bob has 49; why should Bob get 98% of CPU time?
 - ✦ Etc.?

Editorial



- ✦ Real issue: $O(1)$ scheduler bookkeeping is complicated
 - ✦ Heuristics for various issues makes it more complicated
 - ✦ Heuristics can end up working at cross-purposes
- ✦ Software engineering observation:
 - ✦ Kernel developers better understood scheduling issues and workload characteristics, could make more informed design choice
- ✦ Elegance: Structure (and complexity) of solution matches problem

CFS idea



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

But lists are inefficient



- ✦ Duh! 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: 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

Edge case 1



- ✦ 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

What happened to priorities?

- ✦ Priorities let me be deliberately unfair
 - ✦ This is a useful feature
- ✦ 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

Interactive latency



- ✦ 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 program strategy



- ✦ Just like $O(1)$ scheduler, CFS takes blocked programs out of the timeline
- ✦ 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



- ✦ Per group or user scheduling
 - ✦ Real to virtual tick ratio becomes a function of number of both global and user's/group's tasks
- ✦ Unclear how CPU topologies are addressed

CFS Summary



- ✦ Simple idea: logically a queue of runnable tasks, ordered by who has had the least CPU time
- ✦ Implemented with a tree for fast lookup, reinsertion
- ✦ Global clock counts virtual ticks
- ✦ Priorities and other features/tweaks implemented by playing games with length of a virtual tick
 - ✦ Virtual ticks vary in wall-clock length per-process

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 file system data gets evicted
 - ✦ Optional reading (interesting): Nemesis – an OS without shared caches to improve real-time scheduling

Simple hack



- ✦ Create a highest-priority scheduling class for real-time process
 - ✦ `SCHED_RR` – RR == round robin
- ✦ RR tasks fairly divide CPU time amongst themselves
 - ✦ Pray that it is enough to meet deadlines
 - ✦ If so, other tasks share the left-overs
- ✦ Assumption: like GUI programs, RR tasks will spend most of their time blocked on I/O
 - ✦ Latency is key concern

Next issue: 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

Timeslices + syscalls



- ✦ System call times vary
- ✦ Context switches 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, duh!
- ✦ 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
 - ✦ Cf: Interrupt handlers can disable interrupts if needed

Kernel Preemption



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

Priority Laundering



- ✦ Some attacks are based on race conditions for OS resources (e.g., symbolic links)
 - ✦ Generally, these are privilege-escalation attacks against administrative utilities (e.g., passwd)
- ✦ Can only be exploited if attacker controls scheduling
 - ✦ Ensure that victim is descheduled after a given system call (not explained today)
 - ✦ Ensure that attacker always gets to run after the victim

Problem rephrased



- ✦ At some arbitrary point in the future, I want to be sure task X is at the front of the scheduler queue
 - ✦ But no sooner
 - ✦ And I have some CPU-intensive work I also need to do
- ✦ Suggestions?

Dump work on your kids



✦ Strategy:

- ✦ Create a child process to do all the work
 - ✦ And a pipe
- ✦ Parent attacker spends all of its time blocked on the pipe
 - ✦ Looks I/O bound – gets priority boost!
- ✦ Just before right point in the attack, child puts a byte in the pipe
 - ✦ Parent uses short sleep intervals for fine-grained timing
- ✦ Parent stays at the front of the scheduler queue

SBU Pride



- ✦ This trick was developed as part of a larger work on exploiting race conditions at SBU
 - ✦ By Rob Johnson and SPLAT lab students
 - ✦ An optional reading, if you are interested
- ✦ Something for the old tool box...

Summary



- ✦ Understand:
 - ✦ Completely Fair Scheduler (CFS)
 - ✦ Real-time scheduling issues
 - ✦ Kernel preemption
 - ✦ Priority laundering