Scheduling, Part 2

Don Porter

Logical Diagram

Today’s Lecture
Switching to CPU scheduling

Hardware

CPU Scheduler

Device Drivers

Memory Management

Interrupts

Disk

Net

Consistency

 Today’s Lecture
Switching to CPU scheduling

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

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
CFS Example

List sorted by how many "ticks" the task has had

Schedule "neediest" task

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

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

Details

- Global virtual clock: ticks at a fraction of real time
  - Runqueue->fair_clock
  - Fraction is number of total tasks
- Each task counts how many clock ticks it has had
- Example: 4 tasks, equal number of virtual ticks
  - Global vclock ticks once every 4 real ticks
  - Each task scheduled for one real tick; advances local clock by one tick

CFS Example (more realistic)
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 weight:
  – Example:
    – For a high-priority task, 10 actual clock ticks
    – For a low-priority task, a virtual, task-local tick may 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 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
• 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

Recap: Ticks galore!
• Real time is measured by a timer device, which “ticks” at a certain frequency by raising a timer interrupt
• A process’s virtual tick is some number of real ticks
  – We implement priorities, per-user fairness, etc. by tuning this ratio
• The global tick counter tracks maximum possible virtual ticks
  – Used to calculate one’s deficit
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 nth 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 operations 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 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

Summary

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