Scheduling

Don Porter

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

Undergrad review

• 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

Where can we preempt a process?

• In other words, what are the logical points at which the OS can regain control of the CPU?
• System calls
  — Before
  — During (more next time on this)
  — After
• Interrupts
  — Timer interrupt – ensures maximum time slice

(Linux) Terminology

• 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
• Low-level mechanisms
• O(1) Scheduler
• CPU topologies
• 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

Context switching
• What is it?
  – Swap out the address space and running thread
• Address space:
  – Need to change page tables
  – Update cr3 register on x86
  – Simplified by convention that kernel is at same address range in all processes
  – What would be hard about mapping kernel in different places?

Other context switching tasks
• Swap out other register state
  – Segments, debugging registers, MMX, etc.
• If descheduling a process for the last time, reclaim its memory
• Switch thread stacks

Switching threads
• Programming abstraction:
  /* Do some work */
  schedule(); /* Something else runs */
  /* Do more work */
How to switch stacks?

- Store register state on the stack in a well-defined format
- Carefully update stack registers to new stack
  - Tricky: can’t use stack-based storage for this step!

Example

```
Thread 1 (prev)

esp
ebp
regs

Thread 2 (next)

esp
ebp
regs

eax

/* eax is next->thread_info.esp */
/* push general-purpose regs*/
push ebp
mov esp, eax
pop ebp
/* pop other regs */
```

Weird code to write

- Inside schedule(), you end up with code like:
  ```c
  switch_to(me, next, &last);
  /* possibly clean up last */
  ```
- Where does last come from?
  - Output of switch_to
  - Written on my stack by previous thread (not me)!

How to code this?

- Pick a register (say ebx); before context switch, this is a pointer to last’s location on the stack
- Pick a second register (say eax) to stores the pointer to the currently running task (me)
- Make sure to push ebx after eax
- After switching stacks:
  - pop ebx /* eax still points to old task*/
  - mov (ebx), eax /* store eax at the location ebx points to */
  - pop eax /* Update eax to new task */

Outline

- Policy goals
- Low-level mechanisms
- O(1) Scheduler
- CPU topologies
- Scheduling interfaces

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?
  - Only allows round-robin scheduling
  - Can’t prioritize tasks
Even straw-ier man

- Naive approach to priorities:
  - Scan the entire list on each run
  - Or periodically reshuffle the list
- Problems:
  - Forking – where does child go?
  - What about if you only use part of your quantum?
    - E.g., blocking I/O

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 runnable 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 – one active and one 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
What now?

Active

| 139 |
| 138 |
| 137 |
| 101 |
| 100 |

Expired

| 139 |
| 138 |
| 137 |
| 101 |
| 100 |

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.

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

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

- “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

\[\text{dynamic priority} = \max (100, \min (\text{static priority} - \text{bonus} + 5, 139))\]

- 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 (directly)
  - Static priority is a starting point for dynamic priority
  - No matter how “nice” you are (or aren’t), you can’t boost your “bonus” 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 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

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

Average load

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

Why not rebalance?

- Intuition: If things run slower on another CPU
- Why might this happen?
  - NUMA (Non-Uniform Memory Access)
  - Hyper-threading
  - Multi-core cache behavior
- VS: Symmetric Multi-Processor (SMP) – performance on all CPUs is basically the same
SMP

- All CPUs similar, equally “close” to memory

NUMA

- Want to keep execution near memory; higher migration costs

Scheduling Domains

- General abstraction for CPU topology
- “Tree” of CPUs
  - Each leaf node contains a group of “close” CPUs
  - When an idle CPU rebalances, it starts at leaf node and works up to the root
  - Most rebalancing within the leaf
  - Higher threshold to rebalance across a parent

Hyper-threading

- Precursor to multi-core
  - A few more transistors than Intel knew what to do with, but not enough to build a second core on a chip yet
- Duplicate architectural state (registers, etc), but not execution resources (ALU, floating point, etc)
- OS view: 2 logical CPUs
- CPU: pipeline bubble in one “CPU” can be filled with operations from another; yielding higher utilization
Hyper-threaded scheduling

- Imagine 2 hyper-threaded CPUs
  - 4 Logical CPUs
  - But only 2 CPUs-worth of power
- Suppose I have 2 tasks
  - They will do much better on 2 different physical CPUs than sharing one physical CPU
- They will also contend for space in the cache
  - Less of a problem for threads in same program. Why?

Multi-core

- More levels of caches
- Migration among CPUs sharing a cache preferable
  - Why?
  - More likely to keep data in cache
- Scheduling domains based on shared caches
  - E.g., cores on same chip are in one domain

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

Outline

- Policy goals
- Low-level mechanisms
- O(1) Scheduler
- CPU topologies
- Scheduling interfaces
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 how context switching implemented
- Understand O(1) scheduler + rebalancing
- Understand various CPU topologies and scheduling domains
- Scheduling system calls