CSE 506: Operating Systems

Scheduling
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 application more control
    • One task can hog the CPU forever
  – Preemptive gives OS more control
    • More overheads/complexity
Where can we preempt a process?

• When can the OS can regain control?

• System calls
  – Before
  – During
  – After

• Interrupts
  – Timer interrupt
    • Ensures maximum time slice
  – Keyboard, network, disk, ...
(Linux) Terminology

• **mm_struct** – represents an address space in kernel
• **task** – represents a thread in the kernel
  – Traditionally called *process control block (PCB)*
  – A task points to 0 or 1 **mm_structs**
    • Kernel threads just “borrow” previous task’s mm
    • Possible because they only execute in high addresses
      – Shared by all processes
  – Multiple tasks can point to the same **mm_struct**
    • Multi-threading
• **Quantum** – CPU timeslice
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 don’t want slow GUI
No perfect solution

- Optimizing multiple variables
- Like memory allocation, this is best-effort
  - Some workloads prefer some scheduling strategies
- Some solutions are generally “better” than others
Context switching

• What is it?
  – Switch out the address space and running thread

• Address space:
  – Need to change page tables
    • Update CR3 register on x86
  – By convention, kernel at same address in all processes
    • What would be hard about mapping kernel in different places?
Other context switching tasks

• Switch out other register state
• Reclaim resources if needed
  – e.g., if de-scheduling a process for the last time (on exit)
    • Exercise care – page tables/mm_struct still used by kernel
• Switch thread stacks
  – Assuming each thread has its own stack
Switching threads

• Programming abstraction:

/* Do some work */
schedule(); /* Something else runs */
/* Do more work */

• Threads generally unaware of others
  – Calling schedule() can return immediately
  – Or it can return after a very long time (many threads run)
How to switch stacks?

• Store register state on stack in a well-defined format

• Carefully update stack register to new stack
  – Tricky: can’t use stack-based storage for this step!

• Assumes each process has its own kernel stack
  – The “norm” in today’s Oses
    • Just include kernel stack in the PCB
  – Not a strict requirement
    • Can use “one” stack for kernel (per CPU)
    • More headache and book-keeping
/* rax is next->thread_info.rsp */
/* push general-purpose regs*/
push rbp
mov rax, rsp
pop rbp
/* pop general-purpose 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?

- **rax**: pointer to me
- **rbx**: pointer to last’s location on my stack
- **rcx**: pointer to next

```assembly
push rax /* ptr to me on my stack */
push rbx /* ptr to local last (&last) */
mov rsp, rax(10) /* save my stack ptr */
mov rcx(10), rsp /* switch to next stack */
pop rbx /* get next’s ptr to &last */
mov rax, (rbx) /* store rax in &last */
pop rax /* Update me (rax) to new task */
```
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

• Naïve 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 task is on exactly one runqueue
    - Task only scheduled on runqueue’s CPU unless migrated

- $2 \times 40 \times \#\text{CPUs}$ runqueues
  - 40 dynamic priority levels (more on this later)
  - 2 sets of runqueues – one active and one expired
O(1) Data Structures

Active

139 -> 138 -> 137 -> . -> 101 -> 100

Expired

139 -> 138 -> 137 -> . -> 101 -> 100
O(1) Intuition

• Take first task from lowest runqueue on active set
  – Confusingly: a lower priority value means higher priority
• When done, put it on runqueue on expired set
• On empty active, swap active and expired runqueues
• Constant time
  – 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 quantum expires
What now?

Active

139
138
137

Expired

139
138
137

101
100

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
    • 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

Active

139
138
137

Block on disk!

Expired

139
138
137

Disk

Process goes on disk wait queue
Blocked Tasks, cont.

- A blocked task is moved to a wait queue
  - Moved back 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 process blocks and then becomes runnable
  – How do we know how much time it had left?

• Each task tracks time left in ‘time_slice’ field
  – On each clock period: 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

\[\text{time} =\]

\[(140 - \text{prio}) \times 20\text{ms} \quad \text{prio} < 120\]

\[(140 - \text{prio}) \times 5\text{ms} \quad \text{prio} \geq 120\]

- “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 annoyed if keypress takes 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 wait on I/O

• Monitor I/O wait time
  – 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 to encode to mp3
  – Scheduling should match program phases
Dynamic priority

- priority = max(100, min(static priority - bonus + 5, 139))
- Bonus is calculated based on sleep time
- Dynamic priority determines a task’s runqueue
- Balance throughput and latency with infrequent I/O
  - 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 and “fairness” to I/O intensive apps

• “nice” values influence static priority
  – Can’t boost dynamic priority without being in wait queue!
  – No matter how “nice” you are (or aren’t)
Average load

• How do we measure how busy a CPU is?
• Average number of runnable tasks over time
• Available in /proc/loadavg
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)`
yield()

- Moves a runnable task to the expired runqueue
  - Unless real-time, move to the end of the active runqueue
- Several other real-time related APIs
How about a “better” scheduler?

• O(1) scheduler – older Linux scheduler
  – Today: Completely Fair Scheduler (CFS) – new hotness

• Other advanced scheduling issues
  – Real-time scheduling
  – Kernel preemption
Fair Scheduling

- Idea: 50 tasks, each should get 2% of CPU time
- Do we really want this?
  - What about priorities?
  - Interactive vs. batch jobs?
  - Per-user fairness?
    - Alice has 1 task and Bob has 49; why should Bob get 98% of CPU?
If you thought O(1) was a hack...

• Real issue: O(1) scheduler is complicated
  – Heuristics for various issues makes it more complicated
  – Heuristics can end up working at cross-purposes

• Software engineering observation
  – If kernel devs. understood scheduling and workloads
    • Could make more informed design choice

• If you prefer 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

Schedule “neediest” task

List sorted by how many “ticks” the task has had
CFS Example

Once no longer the neediest, put back on the list
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 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: 13
Edge case 1

• What about a new task?
  – If task ticks start at zero, 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
- Higher-priority tasks run longer
- Low-priority tasks make some progress

Note: 10:1 ratio is a made-up example. See code for real weights. (heuristics/hacks)
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

- CFS blocked tasks removed from RB-tree
  - Just like O(1) scheduler
- Virtual clock 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 vclock value than CPU-bound jobs
Other refinements

• Per group or user scheduling
  – Controlled by real to virtual tick ratio
    • Function of number of global and user’s/group’s tasks
Recap: Ticks galore!

• Real time is measured by a timer device
  – “ticks” at a certain frequency by raising a timer interrupt

• A process’s virtual tick is some number of real ticks
  – Priorities, per-user fairness, etc... done by tuning this ratio

• Global ticks tracks max virtual ticks any process had
  – Used to calculate one’s deficit
CFS Summary

• Idea: logically a 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
Real-time scheduling

• Different model
  – Must do modest amount of work by a deadline

• Example:
  – Audio application must deliver a frame every \( n \) ms
  – Too many or too few frames unpleasant to hear
Strawman

• If I know it takes $n$ ticks to process a frame of audio
  – 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 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
  – Cached file system data gets evicted
Simple hack

- Real-time tasks get highest-priority scheduling class
  - `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
    - Other tasks may never get to run
- Assumption: RR tasks mostly blocked on I/O
  - Like GUI programs
  - Latency is the key concern
Next issue: Kernel time

• Should time spent in OS count against task?
  – Yes: Time in system call is work on behalf of that task
  – No: Time in IRQ handler may complete I/O for other 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
    • Usually assumes sequence won’t be interrupted

• General strategy: fragile code disables preemption
  – Like IRQ 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
  – Harder to do without per-thread stacks

• Result: UI programs a bit more responsive