Virtualizing the CPU: Scheduling, Context Switching & Multithreading

Nima Honarmand
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
(Linux) Terminology

- **mm_struct** – represents an address space in kernel
- **task_struct** – represents a thread in the kernel
  - Traditionally called *process control block (PCB)*
  - A **task_struct** points to a **mm_struct** to represent its address space
  - Many tasks can point to the same **mm_struct**
    - Multi-threading (topic of the next lecture)
- Quantum – CPU timeslice
Context Switching
Context Switching

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

• 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?

• Thread context:
  • Save and restore general purpose registers
  • Switch the stack
Other Context Switching Tasks

• Switch out other thread state
  • Other register state if used
    • Segment selectors (fs and gs)
    • Floating point registers
    • Debugging registers
    • Performance counters
  • Update TSS

• Reclaim resources if needed
  • E.g., if de-scheduling a process for the last time (on exit) reclaim its memory
Switching Threads

• Programming abstraction:

/* Do some work */

schedule();  // Choose Something else
             // to run & switch to it

/* Do more work */
schedule() in a Nutshell

```c
schedule() {
    struct task_struct *prev, *next, *last;
    ...
    prev = current; // current thread
    next = ... // next thread to switch to
    ...
    switch_to(prev, next, last);
    // clean up last if need be
    // etc.
}
```

- In `switch_to()`, `prev`’s registers are saved, stacks are switched and `next`’s registers are restored

- Where does `last` come from?
  - Output of `switch_to`
  - Written on my stack by previous thread (not me)!
What Happens in `switch_to()`?

- Lots of inline assembly code
  - Totally architecture specific — we assume x86.
- Push `prev`’s registers on the current stack
- Save `prev`’s stack pointer to its `task_struct`
- Restore `next`’s stack pointer from its `task_struct`
- Pop `next`’s registers from the new stack
- We assume each process has its own kernel stack
  - Common in modern OSes
  - **Note:** We’re discussing context switch while in the kernel so the current stack is the kernel stack

**DANGER!** Do not use the stack while doing this.
How to Code This?

• **rax**: pointer to *prev*; **rcx**: pointer to *next*
• **rbx**: pointer to last’s location on my stack
• **OFFS**: offset of stack pointer value in task_struct
• Make sure **rbx** is pushed after **rax**

**Push Regs**
- `push rax`  /* ptr to me on my stack */
- `push rbx`  /* ptr to local last (&last) */

**Switch Stacks**
- `mov rsp, OFFS(rax)`  /* save my stack ptr */
- `mov OFFS(rcx), rsp`  /* switch to next stack */

**Pop Regs**
- `pop rbx`  /* get next’s ptr to &last */
- `mov rax, (rbx)`  /* store rax in &last */
- `pop rax`  /* Update me to new task */
Scheduling Policy & Algorithms
Policy Goals

• Fairness – everyone gets a fair share of the CPU

• User priorities
  • Virus scanning is nice, but don’t want slow GUI

• Latency vs. Throughput
  • GUI programs should feel responsive (latency sensitive)
  • CPU-bound jobs want long CPU time (throughput sensitive)
  • Application’s behavior can change over time
    → Policy needs to dynamically adapt to changes in application behavior

• Real-time deadlines
  • CPU time before a deadline more valuable than time after
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
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

• Problems?
  • Only allows round-robin scheduling
  • Can’t prioritize tasks
  • What if you only use part of your quantum (e.g., blocking I/O)?
  • How to support both latency-sensitive and throughput-sensitive applications?
(Old) Linux 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 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 highest-priority runqueue on active set

• 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**
  - Task priorities: 139, 138, 137, 101, 100
  - Pick first, highest priority task to run

- **Expired**
  - Task priorities: 139, 138, 137, 101, 100
  - Move to expired queue when quantum expires
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
    • 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
•
•
101
100

Expired

139
138
137
•
•
101
100

Block on disk!

Disk

Process goes on disk wait queue
Blocked Tasks (cont.)

• A blocked task is moved to a wait queue
  • Moved back to active queue 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 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

\[
time = \begin{cases} 
(140 - \text{prio}) \times 20ms & \text{prio} < 120 \\
(140 - \text{prio}) \times 5ms & \text{prio} \geq 120 
\end{cases}
\]

• “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 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

- **Problem:** How to know 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 UI (and disk intensive)

• Give these applications a priority boost

• Note that this behavior can be dynamic
  • Example: DVD Ripper
    • UI configures DVD ripping
    • Then it is CPU bound to encode to mp3
  → Scheduling should match program phases
Dynamic Priority

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

- **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
New Linux Scheduler: Completely Fair Scheduler (CFS)
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?

• *Completely Fair Scheduler (CFS)*
  • Default Linux scheduler since 2.6.23
CFS idea

• Back to a simple list of tasks (conceptually)

• Ordered by how much time they have 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

• 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 = number of total tasks
  - Indicates “Fair” share of each task

- 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: 8
Edge Case 1

• What about a new task?
  • If task ticks start at zero, unfair to run for a long time

• Strategies:
  • Could initialize to current Global Ticks
  • 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 task-local tick may last for 10 actual clock ticks
  • For a low-priority task
    • A task-local tick may only last for 1 actual clock tick

• Higher-priority tasks run longer
• Low-priority tasks make some progress

10:1 ratio is a made-up example. See code for real weights.
Interactive Latency

• Recall: UI 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
UI Program Strategy

• Blocked tasks removed from RB-tree
  • Just like O(1) scheduler

• Global vclock 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 local-clock value than CPU-bound jobs
Other Refinements

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

• 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 the fair share of each process
  • Used to calculate one’s deficit
CFS Summary

• Idea: logically a single 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
Other Issues
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 solution
  • If I know it takes $n$ ticks to process a frame of audio, schedule my application $n$ ticks before the deadline

• Problem? hard to accurately estimate $n$
  • Variable execution time depending on inputs
  • Interrupts
  • Cache misses
  • Disk accesses
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
Linux’s Hack

• Have different scheduling classes:
  • `SCHED_IDLE`, `SCHED_BATCH`, `SCHED_OTHER`, `SCHED_RR`, `SCHED_FIFO`

• “Normal” tasks are in class `SCHED_OTHER`

• “Real-time” tasks get highest-priority scheduling class
  • `SCHED_RR` and `SCHED_FIFO` (RR: round robin)
  • RR is preemptive, FIFO is cooperative

• RR tasks fairly divide CPU time amongst themselves
  • Pray that it is enough to meet deadlines
  • Other tasks share the left-overs (if any)

• Assumption: RR tasks mostly blocked on I/O (like GUI programs)
  • Latency is the key concern

• New scheduling class in recent Linux: `SCHED_DEADLINE`
  • Highest priority class in system; Uses “Earliest Deadline First” scheduling
  • Details in `http://man7.org/linux/man-pages/man7/sched.7.html`
Linux Scheduling-Related API

• Includes many functions to set scheduling classes, priorities, processor affinities, yielding, etc.

• See http://man7.org/linux/man-pages/man7/sched.7.html for a detailed discussion
Next Issue: Average Load

• How do we measure how busy a CPU is?
• Average number of **runnable** tasks over time
• Available in `/proc/loadavg`
Next Issue: Kernel Time

• Context switches generally at user/kernel boundary
  • Or on blocking I/O operations

• System call times vary

• Problems: if a time slice expires inside of a system call:
  1) Task gets rest of system call “for free”
     • Steals from next task
  2) 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: allow fragile code to disable 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

• Result: UI programs a bit more responsive
Threading
Threading Review

• Multiple threads of execution in one address space
  • Why?
    • Exploits multiple processors
    • Separate execution stream from address spaces, I/O descriptors, etc.
    • Improve responsiveness of UI (and similar applications)

• x86 hardware:
  • One CR3 register and set of page tables
    • Shared by 2+ different contexts (each has RIP, RSP, etc.)

• Linux:
  • One \texttt{mm\_struct} shared by several \texttt{task\_structs}
Threading Libraries

• Kernel provides basic functionality
  • e.g.: create new thread

• Threading library (e.g., libpthread) provides nice API
  • Thread management (join, cleanup, etc.)
  • Synchronization (mutex, condition variables, etc.)
  • Thread-local storage

• Part of design is division of labor
  • Between kernel and library
User vs. Kernel Threading

• Kernel threading
  • Every application-level thread is kernel-visible
    • Has its own `task_struct`
  • Called 1:1 threading

• User threading
  • Multiple application-level threads ($m$)
    • multiplexed on $n$ kernel-visible threads ($m \geq n$)
  • Context switching can be done in user space
    • Just a matter of saving/restoring all registers (including RSP!)
  • Called $m:n$ threading
    • Special case: $m:1$ (no kernel support) — Cannot schedule multiple threads (of same process) across CPUs
User Threading Implementation

• User scheduler creates:
  • Analog of `task_struct` for each thread
    • Stores register state when switching
  • Stack for each thread
  • Some sort of run queue and scheduling policy
    • Can use any algorithm: simple round-robin, O(1), CFS, etc.

• Context switching similar to what we have seen already
  • Save/restore general purpose registers
  • Switch stacks
Tradeoffs of Threading Approaches

• Context switching overheads

• Finer-grained scheduling control

• Blocking I/O
Context Switching Overheads

• Takes a few hundred cycles to get in/out of kernel
  • Plus cost of saving/restoring registers
  • Plus cost of extra TLB/cache misses

• Time in the scheduler counts against your timeslice

• Forking a thread halves your time slice
  • At least in some schedulers

• 2 threads, 1 CPU
  • Run the context switch code in user-mode
    • Avoiding trap overheads, etc.
    • Get more time from the kernel
Finer-Grained Scheduling Control

• Example: Thread 1 has lock, Thread 2 waiting for lock
  • Thread 1’s quantum expired
  • Thread 2 spinning until its quantum expires
  • Can donate Thread 2’s quantum to Thread 1?
    • Both threads will make faster progress!

• Many examples (producer/consumer, barriers, etc.)

• Underlying problem:
  • Application’s data and synchronization unknown to kernel
    → Kernel makes blind decisions
Blocking I/O

• I/O requires going to the kernel (generally)

• When one user thread does I/O
  • All other user threads in same kernel thread wait

• Solvable with async I/O (`aio` in Unix) and `poll()`-based programming
  • `aio` to avoid blocking on storage access
  • `poll()` to avoid blocking on network access

• Much more complicated to program
  • Still not a perfect solution
Recap: User Threading Complexity

• Lots of libc/libpthread changes
  • Especially, if designed to be application-transparent
  • Working around “unfriendly” blocking kernel API

• Bookkeeping gets much more complicated
  • Second scheduler
  • Synchronization different

• Preemption becomes complicated
  • Should use (expensive) timer signals from OS

→ Good user-mode threading needs better kernel/user interface
Proposal: Scheduler Activations

• Required reading assignment

• Better API for user-level threading
  • Not available on Linux

• On any blocking operation, kernel *upcalls* back to user scheduler
  • Eliminates most libc changes
  • Easier notification of blocking events

• User scheduler keeps kernel notified of how many runnable tasks it has (via system call)
Threading in Practice

• User-threading has come in and out of vogue
  • Correlated with efficiency of OS thread create and switch

• Linux 2.4 – Kernel threading was slow
  • User-level thread packages were hot (e.g., LinuxThreads)
    • Code is really complicated
      • Hard to maintain
      • Hard to tune

• Linux 2.6 – Substantial effort into tuning kernel threads
  • Native POSIX Threads Library (NPTL) — GNU implementation of the POSIX threads (pthreads) API
  • Most JVMs abandoned user threads
    • Tolerable performance at low complexity
Kernel Threading and Synch. Performance

- Consider implementing `pthread_mutex_lock/unlock`
  - Simple lock/unlock functionality

- When lock is uncontended, you want operations to be completely in user-mode
  - Avoid going to kernel (fast path)

- What if the lock is contended?
  - Thread 2 has to wait until Thread 1 releases the lock
Dealing with Contention

Two options:

1) Pure user-mode implementation: Thread 2 spins (busy-wait) until lock is released by Thread 1
   • Thread 2 spins until timeslice finishes → Thread 1 is scheduled back in, releases the lock, and finishes timeslice → Thread 2 is scheduled and grabs the lock
   • Thread 2 wastes processor cycles
   • Gets worse as thread count grows

2) Use kernel’s help: Thread 2 spins for a short while and then puts itself to sleep
   • Thread 1 has to wake it up after releasing the lock
   • How?
Dealing with Contention (2)

• How to wake up a sleeping thread waiting on a lock?
  • Old solution: send it a signal (more on signals in IPC lecture)
    • Complicated to implement and very slow
  • New solution: futex

• Futex: essentially a shared wait queue in the kernel
• Idea:
  • (Fast path) use atomic instructions in user space to implement uncontended case for a lock (avoid going to kernel)
  • (Slow path) if task needs to block, ask the kernel to put you on a given futex wait queue
  • Task that releases the lock wakes up next task on the futex wait queue

• Futex improves NPTL synch. performance significantly, and simplify code compared to using signals
• See optional reading on futexes for more details