6.828 2014 Lecture 19: Virtual Machines ==

Read: Dune: Safe User-level Access to Privileged CPU features

Plan: virtual machines x86 virtualization dune

Virtual Machines

what's a virtual machine? simulation of a computer running as an application on a host computer accurate isolated fast

why use a VM? one computer, multiple operating systems (OSX and Windows) manage big machines (allocate CPUs/memory at o/s granularity) kernel development environment (like qemu) better fault isolation: contain break-ins

how accurate must a VM be? handle weird quirks of operating system kernels reproduce bugs exactly handle malicious software cannot let guest break out of virtual machine! usual goal: impossible for guest to distinguish VM from real computer impossible for guest to escape its VM some VMs compromise, require guest kernel modifications

VMs are an old idea 1960s: IBM used VMs to share big machines 1990s: VMWare re-popularized VMs, for x86 hardware

terminology [diagram: h/w, VMM, VMs..] VMM ("host") guest: kernel, user programs VMM might run in a host O/S, e.g. OSX or VMM might be stand-alone

VMM responsibilities divide memory among guests time-share CPU among guests simulate per-guest virtual disk, network really e.g. slice of real disk

why not simulation (e.g, Qemu)? VMM interpret each guest instruction maintain virtual machine state for each guest eflags, %cr3, &c much too slow!

idea: execute guest instructions on real CPU when possible works fine for most instructions e.g. add %eax, %ebx how to prevent guest from executing privileged instructions? could then wreck the VMM, other guests, &c

idea: run each guest kernel at CPL=3 ordinary instructions work fine privileged instructions will (usually) trap to the VMM VMM can apply the privileged operation to *virtual* state this "virtual state" is sometimes called the "shadow copy" not to the real hardware "trap-and-emulate"

Trap-and-emulate example -- CLI / STI VMM maintains virtual IF for guest VMM controls hardware IF Probably leaves interrupts enabled when guest runs Even if a guest uses CLI to disable them VMM looks at virtual IF to decide when to interrupt guest When guest executes CLI or STI: Protection violation, since guest at CPL=3 Hardware traps to VMM VMM looks at *virtual* CPL If 0, changes *virtual* IF If not 0, emulates a protection trap to guest kernel VMM must cause guest to see only virtual IF and completely hide/protect real IF

note we rely on h/w trapping to VMM if guest writes %cr3, gdtr, &c do we also need a trap if guest *read*s?

x86 virtualization

what real x86 state must VMM hide (i.e. = virtual state)? CPL (low bits of CS) since it is 3, guest expecting 0 gdt descriptors (DPL 3, not 0) gdtr (pointing to shadow gdt) idt descriptors (traps go to VMM, not guest kernel) idtr pagetable (doesn't map to expected physical addresses) %cr3 (points to shadow pagetable) IF in EFLAGS %cr0 &c

VT-x/SVM: extension for virtualizing x86 trap-and-emulate used to be hard on an x86 not all privileged instructions trap at CPL=3 popf silently ignores changes to interrupt flag pushf reveals *real* interrupt flag all those traps can be slow VMM must see PTE writes, which don't use privileged instructions success of VMs resulted Intel and AMD adding support for virtualization VT-x =

Vanderpool Technology makes it easy to implement virtual-machine monitor

VT-x: root and non-root mode VMM runs in root mode can execute privilege instructioins Guest runs in non-root mode restricts instructions New instructions to change between root/non-root mode VMLAUNCH/VMRESUME VMCALL within each mode, kernel/user mode guest can manipulate shadow CPL in kernel mode in non-root mode VM control structure (VMCS) Contains state to save or restore during transition Configuration (e.g., trap on HALT or not)

how can VMM give guest kernel illusion of dedicated physical memory? guest wants to start at PA=0, use all "installed" DRAM VMM must support many guests, they can't all really use PA=0 VMM must protect one guest's memory from other guests idea: claim DRAM size is smaller than real DRAM ensure paging is enabled maintain a "shadow" copy of guest's page table shadow maps VAs to different PAs than guest real %cr3 refers to shadow page table virtual %cr3 refers to guest's page table example: VMM allocates a guest phys mem 0x1000000 to 0x2000000 VMM gets trap if guest changes %cr3 (since guest kernel at CPL=3) VMM copies guest's pagetable to "shadow" pagetable VMM adds 0x1000000 to each PA in shadow table VMM checks that each PA is < 0x2000000 terminology guest virtual -> machine (guest phys) -> physical

how to support two layers of translation? Keep shadow page table in software scan the whole pagetable on every %cr3 load? to create the shadow page table what if guest writes %cr3 often, during context switches? idea: lazy population of shadow page table start w/ empty shadow page table (just VMM mappings) so guest will generate many page faults after it loads %cr3 VMM page fault handler just copies needed PTE to shadow pagetable restarts guest, no guest-visible page fault

what if guest kernel writes a PTE? store instruction is not privileged, no trap does VMM need to know about that write? yes, if VMM is caching multiple page tables idea: VMM can write-protect guest's PTE pages trap on PTE write, emulate, also in shadow pagetable downside: many PTE writes -> many VMM entrances -> slow

Why can't VMM just modify the guest's page-table in-place?

VTx: extended page tables Intel has hardware support: extended page table (AMD has nested page tables) Second layer of page tables Translates guest virtual to guest physical in non-root operation physical page in page directory is translated by extended page tables physical page in page tables is translated by extended page tables Can be configured only by VMM Guest page tables modifications need not be trapped

also shadow the GDT, IDT real IDT refers to VMM's trap entry points VMM can forward to guest kernel if needed VMM may also fake interrupts from virtual disk real GDT allows execution of guest kernel by CPL=3

how to handle devices? trap INB and OUTB DMA addresses are physical, VMM must translate and check rarely makes sense for guest to use real device want to share w/ other guests each guest gets a part of the disk each guest looks like a distinct Internet host each guest gets an X window VMM might mimic some standard ethernet or disk controller regardless of actual h/w on host computer or guest might run special drivers that jump to VMM

Dune --

provides process abstraction that has safe access to privileged hardware privileged hardware instructions to kernel observation: can also be useful to applications sandbox untrusted code within app (system call filtering) privilege separation (sthread) garbage collector using paging hardware processes can enter Dune mode (irreversible) like any other process: can make Unix system calls but access to privileged instructions in dune mode implementation VT-x virtualization Dune process runs in ring 0 in non-root Downloadable kernel module within Linux

Example usages deliver page fault to user space directly from hardware Unlike in JOS, program hardware to invoke application page-fault handler But safe. direct access to page table entries (e.g., set PTE bit) dsm, migration, gc direct access to privilege modes sandboxing and privilege separation

Design of Dune process (see figure 1) Linux kernel runs in ring 0, root Can manipulate real CR3, etc. Starts Dune process using VMLANCH Carefully configured VMCS Dune process runs in ring 0, non-root Can manipulate page tables entries But doesn't effect another processes or kernel PTE entries are virtual ones. Dune can runs parts of Dune process in ring 3, non-root For example, for system call filtering

Memory management Goal: Normal process memory space Expose page tables to replace pieces of kernel functionality Approach: user-controled page table entries are guest virtual Kernel performs additional translation using EPT Safe because dune process can only access guest physical addresses Challenge: Use kernel page table so that dune process has same address space as a normal process Idea: point EPT to same page table root as kernel uses Problems: - EPT different format - width of guest-physical != width of host-virtual expose full host virtual AS (48-bits) but guest-physical width is 36 bits Solution: Lazily and manually construct EPT Map only some addresses ranges in EPT address space fits in first 12G

Exposing access to hardware Separate VMCS for each process Dune exposes time stamp counter

Preserving OS interfaces systems call are hypercalls dune module vectors hypercalls through Linux's system call table

Example application: GC Collector and mutator threads Copy live data from from-space to to-space Mutator starts in to-space with registers pointing to objects Collector scans from-space concurrently Approach: use VM Set permission of unscanned areas in from-space sto "no access" Collector scans from-space and unprotects

GC Tricks with VM Fault faults to implement read/write barriers between collections Dirty bits to see what memory has been touched since last collection Free physical page without freeing virtual page Precise TLB invalidations (INVPLG)

Performance Dune overhead (see table 2) EPT overhead: TLB misses more expensive VM entries are more expensive than system calls GC benchmark (see table 6)

Edit By MaHua