Virtual Machines

Required reading: A comparison of software and hardware techniques for x86 virtualizatonKeith Adams and Ole Agesen, ASPLOS 2006

```
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 do we need?
  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?
  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 modifying e.g. %cr3 and wrecking the VMM?
```

```
idea: run each guest kernel at CPL=3
  ordinary instructions work fine
  writing %cr3 will trap to VMM
    VMM can examine guest's page table
    detect any attempt to get at non-guest physical memory
    perhaps modify page table before installing in h/w %cr3
  "trap-and-emulate"

VMM hides real machine from guests
  virtual vs real
  hardware state:
    "virtual" %cr3: set by guest
    "real" %cr3: managed by VMM
  also machine defined data strctures:
    virtual page table
    real page table (often called "shadow")
  VMM must cause guest to see only virtual CPU state
    and completely hide/protect real state

trap-and-emulate is tricky on an x86
  not all privileged instructions trap at CPL=3
  all those traps can be slow
  VMM must see PTE writes, which don't use privileged instructions

what real x86 state do we have to hide (i.e. != virtual state)?
  physical memory
  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

how shall we give guest illusion of 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
    rewrite guest's pagetable PTEs
    map PA in each PTE
  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
  VMM must copy the guest's pagetable
    so guest doesn't see VMM's modifications to PAs

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
```

```
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?

do all instructions that read/write sensitive state cause traps at CPL=3?
  push %cs will show CPL=3, not 0
  sgdt reveals real GDTR
  pushf pushes real IF
    suppose guest turned IF off
    VMM will leave real IF on, just postpone interrupts to guest
  popf ignores IF if CPL=3, no trap
    so VMM won't know if guest kernel wants interrupts
  IRET: no ring change so won't restore restore SS/ESP

how can we cope with non-trapping instructions that reveal real state?
  rewrite guest code, change them to INT 3, which traps
  keep track of original instruction, emulate in VMM
  INT 3 is one byte, so doesn't change code size/layout
  this is a simplified version of the paper's Binary Translation

how does rewriter know where instruction boundaries are?
  or whether bytes are code or data?
  can VMM look at symbol table for function entry points?

idea: scan only as executed, since execution reveals instr boundaries
  original start of kernel (making up these instructions):
  entry:
    pushl %ebp
    ...
    popf
    ...
    jnz x
    ...
    jxx y
  x:
    ...
    jxx z
  when VMM first loads guest kernel, rewrite from entry to first jump
    replace bad instrs (popf) with int3
    replace jump with int3
    then start the guest kernel
  on int3 trap to VMM
    look where the jump could go (now we know the boundaries)
    for each branch, xlate until first jump again
    replace int3 w/ original branch
    re-start
  keep track of what we've rewritten, so we don't do it again

indirect calls/jumps?
  same, but can't replace int3 with the original jump
  since we're not sure address will be the same next time
  so must take a trap every time

ret (function return)?
  == indirect jump via ptr on stack
  can't assume that ret PC on stack is from a call
  so must take a trap every time. slow!

what if guest reads or writes its own code?
```

```
  can't let guest see int3
  must re-rewrite any code the guest modifies
  can we use page protections to trap and emulate reads/writes?
    no: can't set up PTE for X but no R
  perhaps make CS != DS
    put rewritten code in CS
    put original code in DS
    write-protect original code pages
  on write trap
    emulate write
    re-rewrite if already rewritten
    tricky: must find first instruction boundary in overwritten code

do we need to rewrite guest user-level code?
  technically yes: SGDT, IF
  but probably not in practice
  user code only does INT, which traps to VMM

how to handle pagetable?
  remember VMM keeps shadow pagetable w/ different PAs in PTEs

what if guest kernel writes a PTE?
  no trap from %cr3 write...
  idea: VMM can write-protect guest's PTE pages
  trap on PTE write, emulate, also in shadow pagetable

what if guest writes %cr3 often, during context switches?
  does VMM have to scan the new page table, modify all PTEs?
  idea: lazy population of shadow page table
  start w/ empty real page table (just VMM mappings)
  so guest will generate many page faults after it load %cr3
  VMM page fault handler just copies needed PTE to shadow pagetable
    restarts guest, no guest-visible page fault

guest probably switches among same set of page tables over and over
  as it context-switches among running processes
  idea: VMM could cache multiple shadow page tables
    cache indexed by address of guest pagetable
  start with pre-populated page table on guest %cr3 write
  would make context switch much faster

how to guard guest kernel against writes by guest programs?
  both are at CPL=3
  delete kernel PTEs on IRET, re-install on INT?

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

VMware avoids many faults
  re-writing w/ VMM code, rather than int3
```

```
    often faster than non-VM kernel, e.g. cli vs setting a flag in virt state
    but then code size and fn addresses change
    how does VMware hide e.g. return EIPs?

  VMWare supports Binary Translation (see paper)
    int3 can be expensive: every fn return?
    VMWare allows translations that increase code size
    so actually executes translated code at different address
    indirect and function return pointers are different
      variables/stack hold virtual pointers, that guest expects
      translated code maps indirect pointers before call/ret
    examples of clever BT translations?
      don't trap: directly r/w VMM data structures
      for e.g. instrs that read/write EFLAGS
      via %gs segment register, which points to high address
      BT detects/rewrites guest use of %gs
      and %ds bound prevents non-%gs access to VMM memory
    adaptive PTE update handling
      can detect instructions that often write PTEs
      have them directly modify shadow PTE also
      avoid page-fault trap

  Intel/AMD hardware support for virtual machines
    has made it much easier to implement a VMM w/ reasonable performance
    h/w itself directly maintains per-guest virtual state
      CS (w/ CPL), EFLAGS, idtr, &c
    h/w knows it is in "guest mode"
      instructions directly modify virtual state
      avoids lots of traps to VMM
    h/w basically adds a new priv level
      VMM mode, CPL=0, ..., CPL=3
      guest-mode CPL=0 is not fully privileged
    no traps to VMM on system calls
      h/w handles CPL transition
    what about memory, pagetables?
      h/w supports *two* page tables
      guest page table
      VMM's page table
      guest memory refs go through double lookup
        each phys addr in guest pagetable translated through VMM's pagetable
      thus guest can directly modify its page table w/o VMM having to shadow it
        no need for VMM to write-protect guest pagetables
        no need for VMM to track %cr3 changes
      and VMM can ensure guest uses only its own memory
        only map guest's memory in VMM page table
```