Virtual Machines
Part 2: 18 years ago
# How They’re Different

## IBM 360
- Two execution modes
  - supervisor and problem (user)
  - all sensitive instructions are privileged instructions
- Memory is protectable: 2k-byte granularity
- All interrupt vectors and the clock are in first 512 bytes of memory
- I/O done via channel programs in memory, initiated with privileged instructions
- Dynamic address translation (virtual memory) added for Model 67

## Intel x86
- Four execution modes
  - rings 0 through 3
  - not all sensitive instructions are privileged instructions
- Memory is protectable: segment system + virtual memory
- Special register points to interrupt vector
- I/O done via memory-mapped registers
- Virtual memory is standard
A Sensitive x86 Instruction

• popf
  – pops word off stack, setting processor flags according to word’s content
    - sets all flags if in ring 0
      • including interrupt-disable flag
    - just some of them if in other rings
      • ignores interrupt-disable flag
What to Do?

• Binary rewriting
  – rewrite kernel binaries of guest OSes
    - replace sensitive instructions with hypercalls
    - do so dynamically
• Hardware virtualization
  – fix the hardware so it’s virtualizable
• Paravirtualization
  – virtual machine differs from real machine
    - provides more convenient interfaces for virtualization
    - hypervisor interface between virtual and real machines
    - guest OS source code is modified
Binary Rewriting

- Privilege-mode code run via binary translator
  - replaces sensitive instructions with hypercalls
  - translated code is cached
    - usually translated just once
  - VMWare
  - U.S. patent 6,397,242
  - more recently
    - KVM/QEMU
Fixing the Hardware

- Intel Vanderpool technology: VT-x
  - also known as VMX (virtual-machine extensions)
  - new processor mode
    - “ring -1”
      - root mode
      - other modes are non-root
  - certain events in non-root mode cause VM-exit to root mode
    - essentially a hypercall
    - data structure in root mode specifies which events cause VM-exits
  - non-VMM OSes must be written not to use root mode!

A detailed description of the VMX architecture can be found in Volume 3, Chapters 23 – 30 in the Intel 64 and IA-32 Architectures Software Developer’s Manual, available at http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. While the concepts behind VMX are straightforward, the details are not: eight chapters covering 276 pages in the software developer’s manual are required to describe it.
The status of a virtual machine (and of a real machine) is defined by the various state information, including memory, general-purpose registers, control registers, interrupt vectors, etc. For a real machine, all of this is manipulated directly via machine instructions. For a virtual machine, some of it (such as general purpose registers and much of memory) is manipulated directly, but others are manipulated via the virtual machine monitor as an intermediary. Intel’s vmx architecture (and the equivalent of AMD) allow the system to designate which state information of the virtual machine is manipulated by direct execution within the virtual machine and which must be handled via the VMM. Any attempt by the virtual machine to manipulate the latter results in a vmexit, which is a trap into the VMM, which then modifies the “virtualized” state.
The VMM, running in root mode, maintains the Virtual Machine Control Structure (VMCS), one for each VM. The guest state and host state portions contain saved registers as described in the next slides. The Control fields describe what events cause VM-Exits, what information is saved on VM-Exits, and what information is restored on VM-Enters.
On a VM-Exit, the register state of the VM is saved in the guest state area; the saved register state of the VMM is restored from the host state area.
On a VM-Entry, the VMM’s register state is saved in the host state area, the VM’s register state is restored from the guest-state area.
Examples

- **mov instruction**
  - mov $2, %rax
  - no VM-exit
  - mov $2, %CR3
  - VM-exit

- **interrupts**
  - interrupt occurs
  - VM-exit
  - **popf in ring 0**
    - affects interrupt-disable flag on guest, no effect on real machine
    - no VM-exit
  - set interrupt vector
    - VM-exit
I/O Virtualization

- Channel programs were generic
- I/O via memory-mapped registers is not
  - lots and lots and lots of device drivers
  - must VMM handle all of them?
Real-Machine OS Structure

- Process
- Process
- Process
- Process
- Process

OS

- Device drivers

Devices

Processor(s)
On a Virtual Machine ...

- Process
- OS
- Device drivers
- Virtual devices
- Virtual processor(s)

- VMM
- Devices
- Processor(s)
KVM/QEMU

• KVM
  – kernel virtual machine monitor for Linux
  – uses VMX technology (or AMD equivalent)

• QEMU
  – generic and open source machine emulator and virtualizer
  – does binary rewriting and caching as does VMware
  – emulates I/O devices as well

• KVM/QEMU
  – code executes natively until VM-exit
  – user-space QEMU code does I/O emulation
Paravirtualization

- Sensitive instructions replaced with hypervisor calls
  - traps to VMM
- Virtual machine provides higher-level device interface
  - guest machine has no device drivers
Additional Applications

• Sandboxing
  – isolate web servers
  – isolate device drivers

• Migration
  – VM not tied to particular hardware
  – easy to move from one (real) platform to another
Approaches: Before
Some portions of a process’s kernel context may be difficult or impossible to move, particularly if they are shared with other processes. An example of such difficult-to-move context is a process’s communication state, which, at the least, is tied to the home machine because of its IP address.
Virtual-Machine Migration

- Virtual machines are isolated
  - by definition!
- State is well defined
  - thus easy to identify and move
  - possible exception of virtual memory
Transferring Virtual Memory

- Eager
  - all
  - dirty
    - (clean pages come from common source)
- Lazy
  - copy on reference
- Straightforward
  - flush everything to file system on source, then access file system on target
- Weird
  - precopy
Eager–Dirty

- Freeze process on source
- Transfer all dirty pages to target
- Resume process on target
Precopy

- While process still running on source
  - transfer everything to target (eager–dirty)
- While more than x pages dirty on source
  - transfer newly dirtied pages to target
- Freeze process on source
- Transfer remaining dirty pages to target
- Resume process on target