Virtual Machines
Part 2: 15 years ago
How They’re Different

<table>
<thead>
<tr>
<th>IBM 360</th>
<th>Intel x86</th>
</tr>
</thead>
</table>
| • Two execution modes  
  – supervisor and problem (user)  
  – all sensitive instructions are privileged instructions  
| • Four execution modes  
  – rings 0 through 3  
  – not all sensitive instructions are privileged instructions |
| • Memory is protectable: 2k-byte granularity  
| • Memory is protectable: segment system + virtual memory |
| • All interrupt vectors and the clock are in first 512 bytes of memory  
| • Special register points to interrupt vector |
| • I/O done via channel programs in memory, initiated with privileged instructions  
| • I/O done via memory-mapped registers |
| • Dynamic address translation (virtual memory) added for Model 67  
| • Virtual memory is standard |
Rings

- apps
- 2
- 1
- kernel
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
      - code in root mode specifies which events cause VM-exits
    - non-VMM OSes must be written not to use root mode!

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.
Examples

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

- interrupts
  - interrupt occurs
  - VM-exit, if requested
  - `pop` in ring 0
  - affects interrupt-disable flag on guest
  - 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 …
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
Microkernels
OS Services as User Apps

- Version control
- Application program
- Application program
- File system A
- File system B
- Line discipline
- TCP/IP

user mode

Microkernel

privileged mode
Why?

- It’s cool ...
- Assume that OS coders are incompetent, malicious, or both ...
  - OS components run as protected user-level applications
- Extensibility
  - easier to add, modify, and extend user-level components than kernel components
Implementation Issues

• How are modules linked together?
• How is data moved around efficiently?
Mach

- Developed at CMU, then Utah
- Early versions shared kernel with Unix
  - basis of NeXT OS
    - basis of Macintosh OS X
- Later versions still shared kernel with Unix
  - basis of OSF/1
- Even later versions actually functioned as working microkernel
  - basis of GNU/HURD project
    - HURD: HIRD of Unix-replacing daemons
    - HIRD: HURD of interfaces representing depth
Mach Ports (1)

- Linkage construct

Client → Port → Server

Client

Port

Server

Send rights

Receive rights
Mach Ports (2)

- Communication construct

Client -> Request Message -> Request Port -> Response Port -> Server
Mach Ports (3)

- Communication construct

Client → Request Port → Response Port → Response Message → Server

Request Message → Request Port → Client
RPC

- Ports used to implement *remote procedure calls*
  - communication across process boundaries
  - if procedures are on same machine …
    - local RPC
Example

File system A

Application program

Response Port

Request Port

Response Port

Request Port

Disk driver

user mode

privileged mode
Successful Microkernel Systems

•
•
• ...

Operating Systems In Depth
Copyright © 2016 Thomas W. Doeppner. All rights reserved.
Attempts

- **Windows NT 3.1**
  - graphics subsystem ran as user-level process
  - moved to kernel in 4.0 for performance reasons
- **Macintosh OS X**
  - based on Mach
  - all services in kernel for performance reasons
- **HURD**
  - based on Mach
  - services implemented as user processes
  - no one uses it, for performance reasons ...