Virtual Machines

• CS418, Spring 2006, 17-19 April 2006
Contents

• What is a VM?
• Applications of VMs
• Classification of VM
• Implementation Issues
• Future Directions
• Virtual Machines vs Microkernels
  – Not covered
What is a machine?

• Machine is an implementation of some exported interface.

• For example:
  – Hardware
    • *Interface*: ISA+Exceptions
  – Operating System
    • *Interface*: *User-Mode* ISA + System calls
  – Programming Language
    • *Interface*: Core Language constructs
What is a Virtual Machine?

● Something that is not a machine? No

● It is a machine that provides an illusion of some other machine

● Virtual Machine Monitor (VMM) provides the same interface* as the other machine, and simulates its implementation behavior

● Interesting part: It can do useful things between the interface and the implementation that not usually observable by the users (guests).

(*) in some cases, very similar interface with a little modifications
Types of Virtual Machines

• Kind 1: Full Machine Virtualization
  – *Type I*: The VMM runs on bare hardware. All guest operating systems run within the control of the VMM.
  – *Type II*: The VMM runs as an ordinary application inside some host operating system, but emulates a machine that can run one or more OS-es within it.

• Kind 2: Paravirtualization
  – *Type III*: The VMM runs on bare hardware, but does not emulate it precisely. The guest OS-es that run on this VMM must adapt to the new interface.

• Kind 3: Language Runtimes
  – Ex: Java Virtual Machine  (Not to be covered)
Normal Setup

Hardware -- “real machine”

Operating System

APP
APP
APP
APP
Virtual machine Type I

Operating System
Simulated Machine
Virtual Machine Monitor (VMM)
Hardware -- “real machine”
Motivation

- Machines are (or were) expensive
- Can run more than one OS
- Can “clone” a machine for backup
- Planning for fail-over
- Mass deployment (e.g. Web servers)
- Upgrade and software migration
- Machine Migration and Load balancing
Motivation II

- Emulating a different machine (e.g. for OS bring-up or backwards compatibility)
- Providing a system-level debugging environment
- Administration and logging
- Simulation and Experimentation
  - Ex: simulating large networks.
- Strong Isolation between the guest Operating systems.
What Must Be Emulated?

- Instruction set (both user and supervisor)
- Exception handling interface
- Interrupt mechanism
- Some set of devices
- Basically, everything the OS would normally see.
IBM 360

- Provided Uniform Instruction set for a wide range of machines
- Not all machines implemented the whole ISA
- Low-end machines emulated some instructions in the terms of others.
  - (ex: multiplication through repeated addition)
IBM 370

- Used to implement reliable multiplexing of IBM’s mainframe computers
- Each task was run in its own Virtual machine.
- Virtualizability was an explicit goal of the architecture
- Virtualization was complete: one of the things that could be run inside the VM 370 is VM 370 itself.
Virtual Machines Type II

- The VMM runs as an ordinary application inside some host operating system, but emulates a machine that can run one or more OS-es within it.
  - Ex: VMWare, QEMU, UML
Virtual Machines Type II

- Hardware
- Operating System 1
- Virtual Machine
- OS 2
- OS 3
- APP
- APP
- APP
Motivations

• Operating system development and debugging
  – OS development without having to run on the native machine (and damage it?)
  – Can obtain a dump if the guest-OS crashes
  – Can use the host operating system to control or debug the guest operating system

• Simpler implementation, utilizes the facilities and abstractions of the host OS of emulating the machine.
AMD SimNow

- Developed by VirtuTech for AMD
- Provides full system simulation for OS bringup.
- Earliest “deployment” of AMD Pacifica, NX, and other new features
- Lets supporting software be developed early for hardware testing and faster market deployment.
What does it take to Virtualize a system?

• For Full Machine Virtualization:
  – CPU Virtualization
  – Memory Virtualization
  – Device (I/O) Virtualization
  – Interrupt / Exception Virtualization

• Considerations:
  – Transparency
  – Performance
  – Complexity
What is in the VMM?

Virtual machine Monitor

**M-State**
- OS's context Information
- Register State
- Privilege mode Information
- Virtual Memory / Page Table

M-state 2
- ..
- ..

Hardware
Virtualizing the instruction set

- Consider the privileged instruction `cli` – clear interrupt flag
Staying in Control

Hardware -- “real machine”

Virtual machine

Operating System

Virtually Privileged, Really unprivileged

APP

Unprivileged

Privileged (Supervisor mode) execution
Is this enough?


- Provide Sufficient Conditions for Virtualizability.
Definitions

● Privileged Instructions

● Sensitive Instructions
  – Control Sensitive
  – Behavior Sensitive
    • Location Sensitive
    • Mode Sensitive

● “Innocuous” Instructions

● Traps
Popek-Goldberg Theorems

• **THEOREM 1**: For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

• **THEOREM 2**: A conventional third generation computer is "recursively virtualizable" if it is: (a) virtualizable, and (b) a VMM without any timing dependencies can be constructed for it.

• **THEOREM 3**: A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions.
Case study - IA32


• Privileged Instructions
  • CLI
  • LGDT, LLDT, LIDT
  • Load/store Control Regs
  • IRET
  • HLT
  • RWMSR, WRMSR
  • RDPMC
  • LMSW

• Sensitive Instructions
  • Pushf / Popf
  • SGDT, SLDT, SIDT
  • SMSW
  • LAR, LSL, VERR, VERW
  • Push / Pop Segment Regs
Case 1: Privileged Instructions

Hardware -- “real machine”

Virtual machine

Operating System

cli

Trap

Unprivileged

Note not to interrupt this OS
The Difficult Case

- Flags on IA32

| Res | I | V | V | A | V | R | 0 | N | I | O | O | D | I | T | S | Z | 0 | A | 0 | P | 1 | C |
|     | D | I | I | I | C | M | F | T | P | L | F | F | F | F | F | F | F | F | F | F | F | F |

- Trying to set Privileged flags from user mode silently fails
- User mode can read privileged flags
Solution

• Interpret guest-OS instructions
Can do a little better

Lives in OS address space, but protected from it.

1) Note that we entered because of Pushf
2) Fixup the Guest OS stack with *Virtually correct* value of flags from its M-state
3) Continue execution of Guest OS
MMU Virtualization

- Virtual Physical Memory
- Virtual Virtual Memory
  - Transparency: VMM cannot occupy Virtual Addresses
  - Efficiency: MMU cannot be implemented in software
  - Solution: Shadow Paging
Shadow Paging

Guest Page Table

Shadow on First Access of PTE & Write-protect the containing Guest Page Table

Unshadow PTE when the write-protected containing Guest Page Table is caught being modified, and then un-write-protect that table and continue

Shadow Page Table

(i) "Accessed" bit set on shadowing
(ii) "Dirty" bit set on first write after shadowing
Memory resource Management


• Ballooning
  – Guest VM is in the best position to decide what is valuable to it

• Content based page sharing

• Hot I/O re-mapping
I/O Virtualization

• Problem: Lots of devices
  - Vendor specific interfaces

• The VMM must support all devices, but can export a uniform virtual device to all guest operating systems.

• Need to intercept Bus probes so that one OS does not take control of any device

• Isolation
  - Need to protect each VM from interfering with the other's use of the device

• Real time requirements
  - Audio, video, Network
I/O Virtualization

• Performance Issues:
  – Allow as much direct communication as possible.
  – Copying data is expensive (Network)

• Disk Virtualization
  – Every guest implements its own disk scheduling, which may interfere with one another.

• Legacy devices
  – Some PCI devices are very hard to virtualize

• Many BIOS calls Non-reentrant

• DMA
  – This circumvents virtual memory
  – Impossible to virtualize without HW support
Implementation of Type II VMs

• VMM works by mapping guest OS abstractions into HW abstractions
  – Use host OS's process abstraction for each VM
    • Use guest OS's context switching mechanism
    • Use Ptrace like facility for controlling these VMs
  – Use host Signals for virtual interrupts / exceptions
  – Use host's Per-process address space mappings for Virtual MMU
  – Use host provided files and devices for virtual devices where possible
    • A large file to implement a virtual “disk”
    • File read/write for virtual disk read/write
I/O Virtualization in Type II VMs


Figure adapted from the above paper.

Their copyright reads: Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes. This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.
Interrupt / Exception handling

This is not to be covered

Figure 3: Precise Delivery of Interrupts/Exceptions in VDebug
Paravirtualization

• Trade 100% interface compatibility for performance
  – Fully compliant with ABI (Apps run unmodified)
  – Guest OS-es must be modified to use VMM’s interface

• Guest OS-es are Virtualization aware
  – Privileged instruction are replaced by hypervisor calls

• These VMs assume Popek-Goldberg Theorem 1
  – No need for interpretation / translation

• Ex: Xen, UML
Xen and the Art of Virtualization

More Paravirtualization

- Virtual time vs Real time
  - Virtualizing time is not always possible, but the Guest-OS can be made aware of this to cope with it

- Expose real resource availability
  - Guest-OS can optimize behavior based on actual information.

- Hypervisor occupies a fixed address
  - Reduces complexity and redundant mappings

- MMU virtualization can be done without shadow page tables
More Paravirtualization

- Guest OS-es can be given protected access to certain HW devices
- The Hypervisor can export a clean set of device interfaces.

**PRE-VIRTUALIZATION**
- Attempt to partially automate the process of modifying the Guest-OS by utilizing the help of the compiler tool chain
Xen Device Channels

**Guest Requests DMA:**
1. Grant Reference for Page P2 placed on device channel
2. IDD removes GR
3. Sends pin request to Xen

4. Xen looks up GR in active grant table
5. GR validated against Guest (if necessary)
6. Pinning is acknowledged to IDD
7. IDD sends DMA request to device
Sharing Device drivers

**Device Channels:**
Safe and efficient interdomain messaging

**Unified Interfaces:**
Single driver source
Drivers may be shared

**I/O Spaces:**
Provide strong vertical isolation for driver/device instances

**Operating System A**
(e.g. Windows XP)

**Operating System B**
(e.g. Linux)

**Device Manager**

**Safe Hardware Interface**

**I/O Space A**

**I/O Space B**

**I/O Space C**

**Legacy Interfaces:**
Original OS driver
Still isolated for safety
No driver sharing or restart
Unmodified device-driver reuse

Hardware Support for Virtualization

• Problems with IA-32 (recap):
  – Non-faulting sensitive instructions
    • Ring levels stored in segment selector
    • System state stored in EFLAGS
  – Segment Descriptor caching
  – TLB flushes due to Guest/VMM transitions

• Proposed solutions
  – Intel VT
  – AMD Pacifica
Intel Virtualization Technology (VT)

Traditional Setup

- Ring 3: Apps
- Ring 2: DD?
- Ring 1: DD?
- Ring 0: Kernel

VMX Non-Root Operation For Guest OS-es

- VMXON
- VMSOFF

VMX Root Operation Typically for VMM

VM Entry

VM Exit

Ring 3

Ring 2

Ring 1

Ring 0: VMM
What is added?

- VMX root operation mode, or “Ring -1”.
- VMCS – the VM control structure.
- 10 Instructions.
- Changes to the “normal” (VMX non-root) mode operation for some instructions.
- VMM can specify what interrupts / exceptions it wants notification.
- Shadows for control register reads.
- I/O virtualization support.
VMCS structure

- Guest-state area
  - Guest register state, Activity / Interruptibility state
- Host-state area
  - Host register state
- VM execution control fields
  - Determine of there must be a VM-exit for:
    - Pin based Interrupts / NMIs
    - Some instructions like HLT, INVLPG, RDPNC, read/write Control regs, etc
  - Exception bitmap
  - I/O bitmap
VMCS structure (cont)

- VM execution control fields (cont)
  - Time-stamp counter offsets
  - Guest/Host Masks and Read Shadows for CR0 / CR4
- VM-entry / VM-exit control fields
  - MSR load options
  - Event injection
- VM-exit information fields.
The Virtualization

Innocuous instrs

Guest

Read from CR0/4

IVT

CPUID
Read CR3
VM instrs
R/W MSRs
INVD

HLT, IN/OUT, INVLPG
PAUSE, RDPDC, RDTSC
Write to CR3 ...

Current VMCS

Interrupt
Exception

VMM

HW
Language Run-times

Native APP

Java APP

Java APP

J VM (export HW independent interface)

Operating System 1

Hardware
IO Virtualization Support

- Desirable:
  - Device emulation: Legacy / new interface
  - Device assignment to VMs
  - Device assisted sharing

- DMA remapping using IO page tables
  - IO-TLB to speed up translation
  - DMA isolation and VM protection
  - Facilitates device assignment and VM relocation
AMD Pacifica

• No extra rings
• VMM and guest run in different address spaces (worlds)
• Support for fast world-switches
  – Support for tagged TLBs
• Virtual Machines controlled through VMCBs
• Interrupt virtualization and Virtual interrupts
• Intercepts
  – Instruction, Interrupt, exception, I/O
• IOMMU / GART + DEV checking
Security considerations

Intel LaGrande(*) and AMD64(#) architectures

- Secure startup(*#)
  - Trusted Platform Module (TPM)
- Protected Input / Output(*)
  - Encrypted Keyboard / mouse / USB input
  - Protected pathway to frame-buffer
- Attestation(*)
- Secure storage of Keys(*)
- Security Exception(#)
  - Redirect INITS to scrub sensitive information