System VMs

- Support multiple guest OSes on single hardware platform; all running the same ISA
System VMs

- Support multiple guest OSes on single hardware platform; all running the same ISA
Applications (partial list)

- Simultaneous support for multiple OSes/Apps
  - Easy way to implement multiprogramming without requiring complete multiprogramming OS.

- Legacy applications

- Simultaneous support for different OSes/Apps
  - E.g. Windows and Unix

- Error containment
  - sandboxing
  - If a VM crashes, the other VMs can continue to work
    Assumes VMM is correct (smaller/simpler)

- Operating System debugging
  - Can proceed while system is being used for normal work
Applications, contd.

• **Operating System Migration**
  - Can proceed while “old” OS continues to be used

![Diagram]

New Release

- System Programmers
  - Production Users
    - new release being tested

Old Release

- System Programmers
  - Converted Production Users
    - Unconverted Production Users
    - new release installed

TIME

- System Programmers
  - Converted Production Users
    - Permanently Unconverted Production Users
    - newer release being tested
Applications, contd.

• System software development
• Support for multiple networked machines on one physical machine
  – Allows debug of network software
• Event monitoring
  – traces of execution
  – replay
• Education
System VMs

• Virtual Machine Monitor (VMM) manages real hardware resources

• All Guest systems must be given logical hardware resources

• All resources are virtualized
  – By partitioning real resources
  – By sharing real resources

• Guest state must be managed
  – By using indirection
  – By copying
State Management: Indirection

- Hold guest state in VMM memory
- Change pointer on guest switch
- Example: registers
State Management: Copying

- Hold guest state in VMM Memory
- Copy state on guest switch
System VMs: Mode of Operation

- **VMM runs in** system mode
  - VMM manages/protects processor through conventional mechanisms
- **Guest OSes run in** user mode
  - Guest OSes do not have direct control over hardware resources
    - All attempts to interact w/ hardware resources are intercepted by VMM
- VMM manages shadow copies of Guest System state (incl. control registers)
- VMM schedules and runs Guest Systems
VM Timesharing

- VMM Timeshares resources among guests
  - Similar to OS timesharing applications

- Guest OS must not be allowed to set timer interrupt
- Guest OS must not know the real timer value set by VMM
- VMM can provide guest OS with emulated virtual interval timer
Native and Hosted VMs

- **Traditional uniprocessor system**
- **Native VM system**
- **User-mode Hosted VM system**
- **Dual-mode Hosted VM system**
Virtualizing the Processor
Virtualizing the processor

Execution of the guest instructions

- Emulation
- Direct native execution
Privileged Instructions

• Definition: They are instructions that trap if executed in user mode; not in supervisor mode.

• VMM keeps track of the intended mode.
  – e.g. The guest OS is intended to work in system mode but is actually working in user mode in this virtualized environment.
Control Sensitive Instructions

All instructions that attempt to change the configuration of resources
- e.g. page table in general, timer
Behavior Sensitive instructions:

All instructions whose behavior or results depend on the configuration

- **Examples:**
  - Load physical address
  - POPF (Intel x86): pop stack into flag register
Instruction Types -- Summary

**Innocuous Instructions:** Those that are not control or behavior sensitive
VMM components

These instructions desire to change machine resources, e.g. Load Relocation Bounds Register

Privileged Instruction

These instructions do not change machine resources, but access privileged resources, e.g. IN, OUT, Write TLB

Interpreter Routine n

Interpreter Routine 2

Interpreter Routine 1

Allocator

Dispatcher

Instruction trap occurs
VMM components

• Dispatcher
  – Top level control module for VMM
  – Decides which of other components to call

• Allocator
  – Decides which system resources should be provided and to manage shared resources among VMs

• Interpreters
  – One per privileged instructions
  – Emulate the effects of privileged instructions when operating on virtual resources

• VMM runs in supervisor mode; all other software in user mode
Privileged Instruction Handling

Example:
LPSW: Load Program Status Word
Includes Mode Bit and PC (among other things)

Guest OS code in VM
(user mode)

VMM code
(privileged mode)

Privileged instruction (LPSW)
...
...
...
Next instruction (target of LPSW)

Dispatcher

LPSW Routine:
- Change mode to privileged
- Check privilege level in VM
- Emulate instruction
- Compute target
- Restore mode to user
- Jump to target
Virtual Machine “requirements”

1. **Efficiency**: All innocuous instructions are executed by the hardware directly

2. **Resource control**: The allocator must be invoked when any program attempts to affect system resources

3. **Equivalence**: Any program executes exactly as on real hardware except
   - Performance
   - Availability of system resources

- VMM must satisfy all three requirements
Virtual Machines: Main Theorem

A virtual machine monitor can be constructed if the set of sensitive instructions is a subset of the set of privileged instructions

Proof shows
Equivalence by interpreting privileged instructions and executing remaining instructions natively
Resource control by having all instructions that change resources trap to the VMM
Efficiency by executing all non-privileged instructions directly on hardware

A key aspect of the theorem is that it is easy to check
Virtual Machines: Main Theorem

A virtual machine monitor can be constructed if the set of sensitive instructions is a subset of the set of privileged instructions.

(a) Does not satisfy condition
(b) Satisfies condition — efficiently virtualizable
Recursive Virtualization

- Virtual Machine
- Virtual Machine
- Virtual Machine
- 2nd level VMM
- VMM
- Hardware

Privileged Mode
Problematic ISAs

• Some ISAs, x86 included, have sensitive but not privileged instructions → called critical instructions

• Violating one condition for efficient VMM construction → we call them hybrid virtual machines

• Solution:
  – VMM scans the guest code stream before execution
  – Discover all critical instructions
  – Replace them with traps or jump to VMM
  – This is called patching
Hybrid Virtualization: Patching

Scan Guest OS, find problem instructions, replace with jump to VMM

Original Program  →  Scanner and Patcher  →  Patched Program

Code Patch for discovered critical instruction

Control transfer, e.g. trap

VMM
How to do that?

- VMM to take control at the head of each BB.
- Scan instructions in sequence till the end of BB.
- Do the patching along the line.
- Add another trap at the end of BB to allow VMM to regain control when BB finishes.
High overhead ... Can we optimize?

• Optimization 1:
  – Trap at the end of a scanned BB can be replaced by the original jump after all possible successors have been encounters.
  – Cannot do that with indirect jumps though.

• Optimization 2:
  – VMM can scan several BB at once (if they end with direct jumps)
Optimization 3

- Trap at the beginning of a block containing the critical instruction
- Using lookup table, VMM executes specialized emulated routines (code caching).
- Different instances of the critical instruction may have several interpretations depending in the block
Optimization 3

Control transfer, e.g. trap

Code section emulated in code cache

Two critical instructions combined into a single block

Translation Table

Specialized Emulation Routines

Block 1

Block 2

Block 3

Patched Program

VMM

Code Cache
Virtualizing Memory
Virtualizing Memory: Review

- PT Pointer
- OS memory region
  - process 1 PT
    - user
    - user
    - super
  - process n PT
    - user
    - super
- OS managed Real Pages

PT: Page Table
Virtual Memory Support in System VM

• Each guest VM has its own set of virtual page tables.
• virtual address → real address → physical address
• Physical memory: the hardware memory
• Real memory: guest VM’s illusion of physical memory
• VMM maintains a swap space distinct from the swap spaces of each of the guests.
Page Table & TLB

• If page table is **architected**
  – Its structure is **defined by the ISA**
  – **OS** and hardware cooperate in maintaining and using it.
  – TLB is maintained by the hardware and not visible to OS
  – e.g. IA-32

• If **TLB** is architected
  – Its structure is defined by the ISA.
  – ISA provides instructions to manipulate it.
  – Page tables are part of the ISA’s implementation.
  – **Hardware** is unaware of the page table structure
  – TLB miss causes a trap to the OS
Virtualizing Architected Page Tables

• The OS in each guest VM maintains its own page tables.

• An optimization step:
  – VMM maintains virtual-to-physical mapping in shadow page tables, one for each guest VM.
  – These tables are the ones used by the hardware to make the translation and update the TLB.
Virtualizing Architected Page Tables

• Page table pointer register is virtualized:
  – VMM manages the real page table pointer
  – VMM has access to the virtual version of the page table pointer associated with each guest VM.
  – Access to page table pointer (for read or write) by guest VM traps to VMM.

• How to handle page faults?
**Shadow Page Tables Maintained by VMM**

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2000</td>
<td>500</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Shadow Page Table for Program 1 on VM1**

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1000</td>
<td>Not mapped</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4000</td>
<td>Not mapped</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Shadow Page Table for Program 2 on VM1**

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1000</td>
<td>3000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4000</td>
<td>Not mapped</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Shadow Page Table for Program 3 on VM2**

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Virtualizing Architected TLB

• Software (ISA) managed TLBs
• Here the TLB is virtualized and not the page table.
• VMM maintains a copy of each guest VM TLB as well as the real TLB.
• Any instruction modifying the TLB in the guest VM is intercepted by VMM to keep VMM copies up to date.
• How does VMM manage the real TLB?
Virtualizing Architected TLB: Method 1

• VMM re-write real TLB whenever a guest VM is activated
• Done after translating real addresses to physical addresses.
• Problem: fairly high overhead
Virtualizing Architectured TLB: Method 2

**ASID: Address Space IDentifier**

**Virtual TLBs**

<table>
<thead>
<tr>
<th>ASID mapping:</th>
<th>ASID</th>
<th>Virtual Page</th>
<th>Real Page</th>
</tr>
</thead>
</table>
| prog. 1 - ASID 3
| prog. 2 - ASID 7 | 3    | 1000         | 5000      |
|                 | 3    | 2000         | 1500      |
|                 | 7    | 4000         | 3000      |
|                 | ---  | ---          | ---       |

**ASID Map Table**

<table>
<thead>
<tr>
<th>Virtual ASID</th>
<th>Real ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM1:3</td>
<td>9</td>
</tr>
<tr>
<td>VM1:7</td>
<td>---</td>
</tr>
<tr>
<td>VM2:3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Real TLB**

<table>
<thead>
<tr>
<th>ASID</th>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>3000</td>
</tr>
<tr>
<td>9</td>
<td>2000</td>
<td>500</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Virtual TLB of VM1**

<table>
<thead>
<tr>
<th>ASID</th>
<th>Virtual Page</th>
<th>Real Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**Virtual TLB of VM2**

<table>
<thead>
<tr>
<th>ASID</th>
<th>Virtual Page</th>
<th>Real Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Virtualizing I/O
Virtualizing I/O

• Hardest part of virtualization
  – Many device types
  – Many devices of each type
    • Each with its own driver
  – New devices may be added during lifetime of system

• The main strategy:
  – Construct a virtual version of the device
  – Virtualize the I/O activity directed at the device
Device Types

• **Dedicated**
  – Example: Monitor, mouse, keyboard
  – Device doesn’t have to be virtualized
  – VMM still controls due to privileged mode

• **Partitioned**
  – Example: Disk
  – Make multiple, smaller virtualized versions
  – VMM must translate the parameters (and status) from the virtualized version to the physical one and back.

• **Shared**
  – Example: Network adapter
  – VMM manages virtual state information
  – Translate virtual requests to physical requests

• **Spooled**
  – Example: Printer
  – Shared but at coarse granularity
Spooled Devices

• Two level spool table
• First write to VM spool area
• When ready, VMM copies to VMM spool area
• Then invokes device
• When device finished
  – Both VM and VMM spool tables receive “complete”
Spooled Devices

Virtual Machine 1 Spool Table

<table>
<thead>
<tr>
<th>Program</th>
<th>Status</th>
<th>Location</th>
<th>Real loc.</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Printed</td>
<td>1000</td>
<td>11000</td>
<td>400</td>
</tr>
<tr>
<td>B</td>
<td>Completed</td>
<td>2000</td>
<td>12000</td>
<td>200</td>
</tr>
<tr>
<td>C</td>
<td>Running</td>
<td>3000</td>
<td>13000</td>
<td>200</td>
</tr>
<tr>
<td>D</td>
<td>Completed</td>
<td>4000</td>
<td>14000</td>
<td>500</td>
</tr>
</tbody>
</table>

Virtual Machine 2 Spool Table

<table>
<thead>
<tr>
<th>Program</th>
<th>Status</th>
<th>Location</th>
<th>Real loc.</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Running</td>
<td>1000</td>
<td>21000</td>
<td>400</td>
</tr>
<tr>
<td>Q</td>
<td>Completed</td>
<td>2000</td>
<td>22000</td>
<td>800</td>
</tr>
</tbody>
</table>

VMM Spool Table

<table>
<thead>
<tr>
<th>VM</th>
<th>Program</th>
<th>Status</th>
<th>Real loc.</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>Printed</td>
<td>30000</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>Q</td>
<td>Printing</td>
<td>31000</td>
<td>800</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>Waiting</td>
<td>31800</td>
<td>200</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>Waiting</td>
<td>30400</td>
<td>500</td>
</tr>
</tbody>
</table>

First level at VM

Second level at VMM
Non-existent Devices

• Implement virtual version only
• Example: network adapter
  – Allows VMs on same platform to communicate
Virtualizing I/O Activity

- **OS manages I/O resource**
  - Allocates space on storage devices, etc.
  - Serializes requests for shared devices
- **User software performs system calls with general I/O requests**
- **OS converts I/O calls to driver calls**
  - Driver contains device-specific software
    - Exact commands, controller registers, etc.
- **Driver generates device (and bus)-specific I/O operations**

VMM can intercept guest I/O actions at syscall interface, at device driver interface, or at the I/O operation level interface.
At system call interface

- System call traps to VMM
- VMM interprets system call to produce driver calls
- VMM contains shadow drivers
- Guest OS contains virtual I/O code and drivers
  - Must still be executed, for correct guest state updates
- Problems
  - VMM must interpret all I/O system calls for all guest OSes
  - VMM must have access to drivers for all real devices
At driver call interface

- Guest OS contains driver stubs
- Guest OS driver calls can operate on generic virtual devices
  - To simplify conversion
- VMM contains shadow drivers
  - These drivers correspond to real devices
- Generic I/O operations passed to VMM and converted to shadow driver calls
- Problem
  - VMM must have access to real drivers
  - Need generic drivers for each guest OS
  - Guest OSes must have well defined, modular driver call interface
At I/O device interface

- **Guest OSes contain real drivers**
- **Low level I/O operations trap to VMM**
- **VMM must check/translate I/O operation**
- **If legal, VMM performs I/O operation on behalf of guest**
- **VMM passes control back to guest**

**Problems**
- VMM must know some device specifics (even if it doesn’t contain full drivers)
Reasons for VM Slowdown

• VM initialization
  – Setting up virtual state
• Privileged Instruction overhead
  – Trap to VMM
  – Interpretation by VMM
  – Return from VMM to guest
• System Calls by guest in user mode
  – Requires trap/reflection back to Guest OS
• Interrupts
  – Reflect through VMM before getting to Guest OS
• Virtual Memory Management
  – Shadow page faults when page is already mapped
• Duplicated effort between VMM and Guest OS
  – Memory management done by both
Case Study: VMware
VMware: an x86 System Virtual Machine

• Applying Conventional VMs to PCs - Problems:
  – Installing the VMM on bare hardware, then booting Guests onto VMM.
  – Need to support many device types, many more drivers

• VMware solves both problems

• Uses Host OS/Guest OS model
  – Hosted VM
  – Uses Host OS for some VMM functions
    • Including I/O
VMware: Three Main components

- **Begin with already-loaded Host OS**
  - **VMDriver** (Pseudo-Driver)
    - Host OS-specific
    - Installed as a driver, but can take over the machine
    - Acts as conduit between System and User VMMs
  - **VMMonitor** (System-level VMM)
    - Slipped under installed OS via Pseudo-Driver
  - **VMApp** (User-level VMM)
    - Appears as ordinary application to installed OS
    - Can make normal I/O calls (and use installed drivers)
VMM Communication

• User VMM (VMapp) performs system call to pseudo-driver; then waits for response
• System VMM maintains control, then sends response message back to User VMM
Resource Management

- Host OS schedules processor resource
  - User-level VMM is just another application
- Host OS manages memory
  - VM memory is allocated as address space of User-level VMM
  - User level VMM “mallocs”; whole VM uses it
VMware I/O

- Guest OS contains generic drivers
- Generic drivers operate on virtual devices managed by user mode portion of VMM
- User mode portion of VMM makes normal system calls
- System calls cause Host OS to use real drivers and devices
Conclusions

• System VMs virtualize processor, memory, and I/O
• System VMs must control all resources
• There are two type of resources:
  – replicated (keyboard, ...)
  – shared (processor, memory, storage, and some I/O)
• ISAs affect the way we design system VMs:
  – sensitive instructions and privileged instructions
  – architected page tables or architected TLBs