Lecture 10: System VMs - II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Disclaimer: Many slides of this lecture are based on the slides of authors of the textbook from Elsevier. All copyrights reserved.
Problematic ISAs

• Some ISAs, x86 included, have sensitive but not primitive 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 encountered.
  – 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.
• 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

Translation Table

Specialized Emulation Routines

Block 1

Block 2

Block 3

Translation Table

Patched Program

VMM

Two critical instructions combined into a single block
Virtualizing Memory: Review

PT Pointer

OS memory region

process 1 PT

user
user
super

process n PT

user
super

OS managed
Real Pages
Virtual Memory Support in System VM

• Each guest VM has its own set of virtual page tables.
• virtual address $\rightarrow$ real address $\rightarrow$ 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>

Program 1 on VM1 is currently active

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
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 Architected TLB: Method 2

### Virtual TLBs

<p>| ASID mapping: prog. 1 - ASID 3 | Virtual TLB of VM1 |</p>
<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>
<tr>
<td>3</td>
<td>1000</td>
<td>5000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>1500</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>7</td>
<td>4000</td>
<td>3000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

<p>| ASID mapping: prog. 2 - ASID 7 |</p>
<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>
<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>5000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

### 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>---</td>
<td>---</td>
<td>---</td>
</tr>
<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>5000</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**ASID**: Address Space IDentifier
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**
  – Monitor, mouse, keyboard
  – Device doesn’t have to be virtualized
  – VMM still controls due to privileged mode

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

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

• **Spooled**
  – 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>
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**
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
  - I/O initiated by guest OS may not always pass through call interface
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 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
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
Case Study: Xen
Xen

- Multiplexes resources at the granularity of an entire OS
  - As opposed to process-level multiplexing
  - Price: higher overhead
- Target: 100 virtual OSes per machine
Xen: Approach and Overview

• Conventional approach
  – Full virtualization
    • Cannot access the hardware
    • Problematic for certain privileged instructions (e.g., traps)
    • No real-time guarantees
Xen: Approach and Overview

- Xen: paravirtualization
  - Provides some exposures to the underlying HW
    - Better performance
    - Need modifications to the OS
    - No modifications to applications
Memory Management

• Depending on the hardware supports
  – Software managed TLB
    • Associate address space IDs with TLB tags
    • Allow coexistence of OSes
    • Avoid TLB flushing across OS boundaries
Memory Management

• X86 does not have software managed TLB
  – Xen exists at the top 64MB of every address space
  – Avoid TLB flushing when a guest OS enter/exists Xen
  – Writes are validated by Xen
CPU

• X86 supports 4 levels of privileges
  – 0 for OS, and 3 for applications
  – Xen downgrades the privilege of OSes
  – System-call and page-fault handlers registered to Xen
  – “fast handlers” for most exceptions, Xen isn’t involved
Device I/O

• Xen exposes a set of simple device abstractions
Time and Timers

• Xen provides each guest OS with
  – Real time (since machine boot)
  – Virtual time (time spent for execution)
  – Wall-clock time

• Each guest OS can program a pair of alarm timers
  – Real time
  – Virtual time
Virtual Address Translation

- No shadow pages (VMWare)
- Xen provides constrained but direct MMU updates
- All guest OSes have read-only accesses to page tables
- Updates are batched into a single hypercall
Conclusions

• System VMs virtual processor, memory, and I/O

• ISAs affect the way we design system VMs:
  – sensitive instructions and privileged instructions
  – architected page tables or architected TLBs