Virtualizing CPU - 2
Virtualizing Memory - 1

Yiying Zhang
Logistics

• Form project group by the end of today (email TA names, student IDs, email addrs, tentative project topic)

• Project proposal due 10/17!

• First quiz tentatively on 10/17
  • Topics: virtualization overview, virtualizing CPU/memory/IO
  • T or F, multiple choices, short-answer questions
  • About 10-15 minutes

Acknowledgment: slides adapted from Columbia E6998 Lecture 2 by Scott Devine
Virtualization Approach 3:
Direct Execution with Binary Translation

- VMM dynamically rewrites instructions
- So that non-virtualizable instructions can trap to VMM
- VMware’s main selling point (at least in early years)
Basic Blocks

Guest Code

<table>
<thead>
<tr>
<th>vPC</th>
<th>mov ebx, eax</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cli</td>
</tr>
<tr>
<td></td>
<td>and ebx, ~0xffff</td>
</tr>
<tr>
<td></td>
<td>mov cr3, ebx</td>
</tr>
<tr>
<td></td>
<td>sti</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

- Straight-line code
- Basic Block
- Control flow
## Binary Translation

<table>
<thead>
<tr>
<th>Guest Code</th>
<th>Translation Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ebx, eax</td>
<td>mov ebx, eax</td>
</tr>
<tr>
<td>cli</td>
<td>call HANDLE_CLI</td>
</tr>
<tr>
<td>and ebx, ~0xfff</td>
<td>and ebx, ~0xfff</td>
</tr>
<tr>
<td>mov cr3, ebx</td>
<td>mov [CO_ARG], ebx</td>
</tr>
<tr>
<td>sti</td>
<td>call HANDLE_CR3</td>
</tr>
<tr>
<td>ret</td>
<td>call HANDLE_STI</td>
</tr>
<tr>
<td></td>
<td>jmp HANDLE_RET</td>
</tr>
</tbody>
</table>

**vPC**
Q: Is the translated code for `cli` faster or slower than original `cli`?
Controlling Control Flow

<table>
<thead>
<tr>
<th>Guest Code</th>
<th>Translation Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
<td></td>
</tr>
<tr>
<td>jeq</td>
<td></td>
</tr>
<tr>
<td>add ebx, 18</td>
<td></td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
<td></td>
</tr>
<tr>
<td>mov [ecx], eax</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>test eax, 1</td>
<td></td>
</tr>
<tr>
<td>jeq</td>
<td></td>
</tr>
<tr>
<td>call END_BB</td>
<td></td>
</tr>
<tr>
<td>call END_BB</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{vEPC} \]
Controlling Control Flow

Guest Code

- test eax, 1
- jeq
- add ebx, 18
- mov ecx, [ebx]
- mov [ecx], eax
- ret

Translation Cache

- test eax, 1
- jeq
- call END_BB
- call END_BB
- add ebx, 18
- mov ecx, [ebx]
- mov [ecx], eax
- call HANDLE_RET

eax == 0
Controlling Control Flow

eax == 0
Controlling Control Flow

guest code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Translation Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
<td>test eax, 1</td>
</tr>
<tr>
<td>jeq</td>
<td>jeq</td>
</tr>
<tr>
<td>add ebx, 18</td>
<td>jmp</td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
<td>call END_BB</td>
</tr>
<tr>
<td>mov [ecx], eax</td>
<td>add ebx, 18</td>
</tr>
<tr>
<td>ret</td>
<td>mov ecx, [ebx]</td>
</tr>
<tr>
<td></td>
<td>mov [ecx], eax</td>
</tr>
<tr>
<td></td>
<td>call HANDLE_RET</td>
</tr>
<tr>
<td></td>
<td>mov [ecx], eax</td>
</tr>
<tr>
<td></td>
<td>call HANDLE_RET</td>
</tr>
</tbody>
</table>

eax == 1
Controlling Control Flow

Guest Code

```
<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
</tr>
<tr>
<td>jeq</td>
</tr>
<tr>
<td>add ebx, 18</td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
</tr>
<tr>
<td>mov [ecx], eax</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>
```

Translation Cache

```
<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
</tr>
<tr>
<td>jeq</td>
</tr>
<tr>
<td>jmp</td>
</tr>
<tr>
<td>jmp</td>
</tr>
<tr>
<td>add ebx, 18</td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
</tr>
<tr>
<td>mov [ecx], eax</td>
</tr>
<tr>
<td>call HANDLE_RET</td>
</tr>
<tr>
<td>mov [ecx], eax</td>
</tr>
<tr>
<td>call HANDLE_RET</td>
</tr>
</tbody>
</table>
```

**eax == 1**
Adaptive Binary Translation

• Binary translation can outperform classical virtualization by avoiding traps
  
  • *rdtsc* on Pentium 4: trap-and-emulate 2030 cycles, callout-and-emulate 1254 cycles, in-TC emulation 216 cycles

• What about sensitive instructions that are not privileged?
  
  • “Innocent until proven guilty”

  • Start in the innocent state and detect instructions that trap frequently
    
    • Retranslate non-IDENT to avoid the trap

    • Patch the original IDENT translation with a forwarding jump to the new translation
Virtualization Approach 4: Direct Execution with Hardware-Assisted Virtualization

- Adds a new mode so that sensitive operations could all be properly handled
- Other hardware support to make virtualization easier/faster
Hardware-Assisted CPU Virtualization (Intel VT-x)

- Two new modes of execution (orthogonal to protection rings)
  - VMX root mode: same as x86 without VT-x
  - VMX non-root mode: runs VM, sensitive instructions cause transition to root mode, even in Ring 0

- New hardware structure: VMCS (virtual machine control structure)
  - One VMCS for one virtual processor
  - Configured by VMM to determine which sensitive instructions cause VM exit
  - Specifies guest OS state
Comparison of Pre VT-x and Post VT-x

Hardware w/o VT-x

Guest Applications
- Ring 3
- Guest OS
- Ring 1
- Hypervisor
- Ring 0

Hardware w/ VT-x

Guest Applications
- VMX non-root
  - Ring 3
- Guest OS
  - VMX non-root
  - Ring 0
- Hypervisor
  - VMX root Ring 0
- Host Applications
  - VMX root Ring 3
VMX Mode Transition with Intel VT-x

- VM exit/entry (to/from root mode)
  - Registers and address space swapped in one atomic operation
  - Guest- and host-states saved and loaded to VMCS during transitions
- Whenever possible, sensitive instructions only affect states within the VMCS instead of always trapping (VM exit)
- VM exit
  - vmcall instruction
  - EPT page faults (more next lecture)
  - Interrupts
  - Some sensitive instructions (configured in VMCS)
- VM entry
  - vmlaunch instruction: enter with a new VMCS
  - vmresume instruction: enter for the last VMCS
  - Typical vm exit/enter takes ~200 cycles on modern CPU

Image source: https://www.anandtech.com/show/2480/9
Example: Guest syscall with Hardware Virtualization

- VMM fills VMCS exception table for guest OS (including a syscall handlers)
  - and sets bit in VMCS to not exit on syscall exception
- VMM executes VM entry
- Guest application invokes a syscall
- does not trap (no VMM involvement), but go to the VMCS exception table, which jumps to the guest OS’s syscall handler
Software Binary Translation vs. Hardware-Assisted Virtualization

- Software binary translation wins in:
  - Trap elimination
  - Emulation speed
  - Callout avoidance

- Hardware-assisted virtualization wins in:
  - Code density
  - Precise exceptions
  - Syscalls

![Figure 4. Virtualization nanobenchmarks.](image)

syscall, call/ret, divzero
=> native = hardware > software
in, cr8wr
=> software > native > hardware
pgfault, ptemod
=> native > software > hardware
Virtualization Approach 5: Direct Execution with Paravirtualization

- Full virtualization (no guest OS modification)
  - Tricky and has performance overhead

- Para-virtualization: modified guest OS
  - Change (rewrite) guest OS to remove sensitive but unprivileged instructions and to use other tricks to make virtualization faster
    - Guest OS works with hypervisor (i.e., knows that it is a VM) and has some exposure to hardware
    - e.g., guest OS informs hypervisor of page table changes
    - e.g., guest OS directly calls hypervisor on system calls (hypercalls)
  - Guest applications are still unmodified
  - Pros and Cons?
Other Virtualization Approaches

- **Container**: Essentially just a group of processes with some additional features (isolated namespace, isolated resources, etc.) (e.g., Docker)

- **Unikernel**: LibraryOS designed for a single application, running on hypervisor (as a VM) or host OS (as a process)

- **Sandboxing**: Limit what the applications (and libOS) can do (e.g., gVisor)

- **Language-based**: Running applications written in a high-level language on language runtimes (e.g., JVM)
Virtualization Approaches Summary

- Hosted interpretation
  - Interpret each instruction, super slow (e.g., Virtual PC on Mac)

- Direct execution with trap-and-emulate
  - Requires a virtualizable processor and only works for the same architecture

- Direct execution with binary translation
  - Works with non-virtualizable processor, but implementing VMM is tricky

- Direct execution with hardware-assisted virtualization
  - Needs new generation of hardware (which is the norm now), mode switching is still not optimized

- Direct execution with paravirtualization
  - Good performance and works with non-virtualizable processors, but require guest OS changes

- OS-level virtualization, library-level, language (app)-level, unikernels, etc.
  - More lightweight and faster to start, but less secure
Conclusion

• Virtualizing CPU is a non-trivial task, esp. for non-virtualizable architectures like x86

• Software binary translation is a neat (but very tricky) way to virtualize x86 and still meet Popek and Goldberg’s virtualization principles

• Hardware vendors keep adding more virtualization support, which makes life a lot easier

• Software and hardware techniques both have pros and cons
Virtualizing Memory

• Software-based memory virtualization

• Hardware-assisted memory virtualization

• Memory management
  • Reclaiming
  • Sharing
  • Allocation

Acknowledgment: some slides from Carl Waldspurger’s OSDI’02 presentation
Review: Regular Virtual Memory System
Virtualizing Memory

- Extra level of memory addressing

![Diagram of VMM Memory Virtualization]

Figure B.4: VMM Memory Virtualization
Review: Regular TLB

Virtual address

VPage #  offset

VPage#  PPage#  ...
VPage#  PPage#  ...
...
VPage#  PPage#  ...

TLB

Hit

Miss

Physical address

PPage #  offset

Real page table
Review: Software-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    - OS performs page fault handling
    - Restart the faulting instruction

- On a TLB miss, HW raises exception, traps to the OS
  - OS parses page table and loads PTE into TLB
    - Needs to replace if TLB is full
  - Same as in a hit…
Review: Hardware-controlled TLB

- On a TLB hit, MMU checks the valid bit
  - If valid, perform address translation
  - If invalid (e.g. page not in memory), MMU generates a page fault
    - OS performs fault handling
    - Restart the faulting instruction

- On a TLB miss
  - MMU parses page table and loads PTE into TLB
    - Needs to replace if TLB is full
  - Same as hit …
Virtualizing Memory

- TLB miss flow with software-managed TLB

**Figure B.5: TLB Miss Flow without Virtualization**

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Load from memory: TLB miss: Trap</td>
<td></td>
</tr>
<tr>
<td>2. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid: get PFN, update TLB; Return from trap</td>
<td></td>
</tr>
<tr>
<td>3. Resume execution (@PC of trapping instruction); Instruction is retried; Results in TLB hit</td>
<td></td>
</tr>
</tbody>
</table>

**Figure B.6: TLB Miss Flow with Virtualization**

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
<th>Virtual Machine Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Load from mem TLB miss: Trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. VMM TLB miss handler: Call into OS TLB handler (reducing privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid, get PFN, update TLB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Trap handler: Unprivileged code trying to update the TLB; OS is trying to install VPN-to-PFN mapping; Update TLB instead with VPN-to-MFN (privileged); Jump back to OS (reducing privilege)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Return from trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Trap handler: Unprivileged code trying to return from a trap; Return from trap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Resume execution (@PC of instruction); Instruction is retried; Results in TLB hit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Difficulty in Virtualizing Hardware-Managed TLB

- Hardware-managed TLB
  - Hardware does page table walk on each TLB miss
  - and fills TLB with the found PTE
- Hypervisor doesn’t have chance to intercept on TLB misses
- Solution-1: shadow paging
- Solution-2: direct paging (para-virtualization) (later this quarter if have time)
- Solution-3: new hardware
Shadow Paging

- MPN
- Guest VM
- GPT
- VPN
- PPN
- pmap
- SPT
- MPN
- CR3
Set Up Shadow Page Table

1. VMM intercepts guest OS setting the virtual CR3 (a sensitive operation)
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest physical address is translated into host physical address (machine address)
4. Finally, VMM sets the real CR3 to point to the shadow page table
Set Up Shadow Page Table

```python
set_cr3 (guest_page_table):
    for VPN in 0 to 220
        if guest_page_table[VPN] & PTE_P /* PTE_P: valid bit */
            PPN = guest_page_table[VPN]
            MPN = pmap[PPN]
            shadow_page_table[VPN] = MPN | PTE_P
        else
            shadow_page_table = 0
    CR3 = PHYSICAL_ADDR(shadow_page_table)
```
Question

- Assume that:
  - There are 10 VMs running on a machine
  - Each VM contains 10 applications

- **Q:** How many shadow page tables in total?
  - Shadow page tables are per application
  - Guest page tables are per application
  - pmaps are per VM
What if Guest OS Modifies Its Page Table?

• Should not allow it to happen directly
  • Since CR3 is now pointing to the shadow page table
  • Need to synchronize the shadow page table with guest page table

• VMM needs to intercept when guest OS modifies page table, and updates the shadow page table accordingly
  1. Mark the guest table pages as read-only (by setting the corresponding PTEs’ permission bits in the shadow page table)
  2. If guest OS tries to modify its page tables, it triggers page fault
  3. VMM handles the page fault by updating shadow page table
Dealing with Page Faults

- When page fault occurs, traps to VMM

- If present bit is 0 in the guest page table entry, guest OS needs to handle the fault (VMM forwards the fault to guest OS)
  - Guest OS load page from virtual disk to guest physical memory and sets present bit to 1
  - Guest OS returns from page fault, which traps into VMM again
  - VMM sees that present is 1 in guest PTE and creates entry in shadow page table
  - VMM returns from the original page fault

- If present is 1: guest OS thinks page is present (but VMM may have swapped it out), VMM handles transparently
  - VMM locates the corresponding physical page, loads it in memory if needed
  - VMM creates entry in shadow page table
  - VMM returns from the original page fault
What if a Guest App Access its Kernel Memory?

- How do we selectively allow / deny access to kernel-only pages?

- One solution: split a shadow page table into two tables
  - Two shadow page tables, one for user, one for kernel
  - When guest OS switches to guest applications, VMM will switch the shadow page table as well, vice versa
Two Memory Views of Guest VM

- **Kernel space**
  - When guest OS is running
  - No user access

- **User space**
  - When application is running
  - User access
  - No user access
The Same Question

• Assume that:
  • There are 10 VMs running on a machine
  • Each VM contains 10 applications

• Q: Now, how many shadow page tables in total?
Pros and Cons of Shadow Paging?