Virtualizing Memory - 2

Yiying Zhang
Finalize your project topic and send it to the TA asap

Most groups chose topics 1, 2, 5

- I’ll use 20 min each during my office hour this Wed to go over them
- Will be recorded, but try having someone attend and ask questions

Project proposal due on 10/17!
Outline

- Software-based memory virtualization
- Hardware-assisted memory virtualization
- Memory management
  - Reclaiming
  - Sharing

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

Virtual address

- VPage #
- offset

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

TLB

Hit

Miss

Real page table

- Page-map L4 base addr (CR3)

Page-map L4 table

Page-directory pointer table

Page-directory table

Page-table table

Page table

Physical page frame number

Page offset

Main memory

© 2007 Elsevier, Inc. All rights reserved.
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

- Extra level of memory addressing

Figure B.4: VMM Memory Virtualization
Virtualizing Memory

• TLB miss flow with software-managed TLB

![Diagram: TLB Miss Flow without Virtualization]

<table>
<thead>
<tr>
<th>Process</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Load from mem</td>
<td>1. Load from mem</td>
</tr>
<tr>
<td>TLB miss: Trap</td>
<td>TLB miss: Trap</td>
</tr>
<tr>
<td></td>
<td>2. OS TLB miss handler:</td>
</tr>
<tr>
<td></td>
<td>Extract VPN from VA;</td>
</tr>
<tr>
<td></td>
<td>Do page table lookup;</td>
</tr>
<tr>
<td></td>
<td>If present and valid:</td>
</tr>
<tr>
<td></td>
<td>get PFN, update TLB;</td>
</tr>
<tr>
<td></td>
<td>Return from trap</td>
</tr>
</tbody>
</table>

3. Resume execution            3. OS TLB miss handler: Extract VPN from VA; Do page table lookup; If present and valid, get PFN, update TLB
(@PC of trapping instruction); Return from trap
Instruction is retried; Results in TLB hit

![Diagram: TLB Miss Flow with Virtualization]

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

Figure B.5: TLB Miss Flow without Virtualization

Figure B.6: TLB Miss Flow with Virtualization
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

- GPT
- PPN
- vpn
- PPN
- vpn
- vpn
- MPN
- 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 virtual 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)
```
• 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

When guest OS is running

Kernel space

User space

When application is running

No user access

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?
Outline

• Software-based memory virtualization
• Hardware-assisted memory virtualization
• Memory management
  • Reclaiming
  • Sharing
Hardware-Assisted Memory Virtualization

- Hardware support for memory virtualization
  - Intel EPT (Extended Page Table) and AMD NPT (Nested Page Table)
  - EPT: a per VM table translating PPN -> MPN, referenced by EPT base pointer
  - EPT controlled by the hypervisor, guest page table (GPT) controlled by guest OS (both exposed to hardware)
  - Hardware directly walks GPT + EPT (for each PPN access during GPT walk, needs to walk the EPT to determine MPN)
  - No VM exits due to page faults, INVLPG, or CR3 accesses
EPT Translation: Details
EPT Increases Memory Access

One memory access from the guest VM may lead up to 20 memory accesses!

sPA: machine address
nCR3: root of EPT table
nL_k: EPT table level
Pros and Cons of Shadow Paging and of EPT?
Outline

• Software-based memory virtualization

• Hardware-assisted memory virtualization

• Memory management
  • Reclaiming
  • Sharing
Reclaiming Memory

- ESX (and other hypervisors) allow overcommitment of memory
  - Total memory size of all VMs can exceed actual machine memory size
  - ESX must have some way to reclaim memory from VMs (and swap to disk)
Reclaiming Memory

• Traditional: add transparent swap layer
  • Requires “meta-level” decisions: which page from which VM to swap
  • Best data to guide decisions known only by guest OS
  • Guest and meta-level policies may clash, resulting in *double paging*

• Alternative: implicit cooperation
  • Coax guest OS into doing its own page replacement
  • Avoid meta-level policy decisions
Ballooning

- Inflate balloon (+ pressure)
- Deflate balloon (− pressure)
- Guest OS manages memory implicitly with cooperation
- May page out to virtual disk
- May page in from virtual disk
Ballooning Details

- Guest drivers
  - Inflate: guest decides which pages to page out, PPNs communicated to hypervisor via balloon driver
  - Use standard Windows/Linux/BSD kernel APIs

- Performance benchmark
  - Linux VM, memory-intensive dbench workload
  - Compare 256 MB with balloon sizes 32 - 128 MB vs. static VMs
  - Overhead 1.4% - 4.4%
Memory Sharing

- Motivation
  - Multiple VMs running same OS, apps
  - Collapse redundant copies of code, data, zeros
- Transparent page sharing
  - Map multiple PPNs to single MPN (copy-on-write)
  - Pioneered by Disco, but required guest OS hooks
- New twist: content-based sharing
  - General-purpose, no guest OS changes
  - Background activity saves memory over time
Page Sharing: Scan Candidate PPN

VM 1

VM 2

VM 3

hash page contents

...2bd806af

Machine Memory

hint frame

Hash: ...06af
VM: 3
PPN: 43f8
MPN: 123b

hash table
Page Sharing: Successful Match

VM 1  VM 2  VM 3
Machine Memory

Hash: …06af
Refs: 2
MPN: 123b

hash table
What is the benefit of keeping a "hint" entry for each scanned (but unshared) page (as compared to not maintaining anything for the page)
# Real-World Page Sharing Results

<table>
<thead>
<tr>
<th>Workload</th>
<th>Guest Types</th>
<th>Total</th>
<th>Saved</th>
<th>Saved (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corporate IT</td>
<td>10 Windows</td>
<td>2048</td>
<td>673</td>
<td>32.9</td>
</tr>
<tr>
<td>Nonprofit Org</td>
<td>9 Linux</td>
<td>1846</td>
<td>345</td>
<td>18.7</td>
</tr>
<tr>
<td>VMware</td>
<td>5 Linux</td>
<td>1658</td>
<td>120</td>
<td>7.2</td>
</tr>
</tbody>
</table>

**Corporate IT** – database, web, development servers (Oracle, Websphere, IIS, Java, etc.)

**Nonprofit Org** – web, mail, anti-virus, other servers (Apache, Majordomo, MailArmor, etc.)

**VMware** – web proxy, mail, remote access (Squid, Postfix, RAV, ssh, etc.)
Conclusion

• Software and hardware solutions for memory virtualization both have pros and cons

• More things to take care of besides the basic mechanism of memory virtualization

  • Allocation, sharing, overcommitment and reclamation