Lecture 3

Memory Virtualization
Larry Rudolph
In a traditional system there are typically two address spaces – (1) the physical address space (PAS), i.e. the DRAM of the machine, and (2) the virtual address space (VAS). The machine boot code initially executes in the PAS and then quickly sets up and switches to virtual memory; the OS and user processes run in the VAS. The OS manages the mapping from VAS to PAS through the use of the Memory Management Unit (MMU). The MMU is a part of the computer. The OS maintains a page table that maps each page of memory in the current VAS to a page of memory in the PAS. Typically the OS will maintain one page table per user level process and one for itself.
In a traditional system there are typically two address spaces – (1) the physical address space (PAS), i.e. the DRAM of the machine, and (2) the virtual address space (VAS). The machine boot code initially executes in the PAS and then quickly sets up and switches to virtual memory; the OS and user processes run in the VAS. The OS manages the mapping from VAS to PAS through the use of the Memory Management Unit (MMU). The MMU is a part of the computer. The OS maintains a page table that maps each page page of memory in the current VAS to a page of memory in the PAS. Typically the OS will maintain one page table per user level process and one for itself.
In a traditional system there are typically two address spaces – (1) the physical address space (PAS), i.e. the DRAM of the machine, and (2) the virtual address space (VAS). The machine boot code initially executes in the PAS and then quickly sets up and switches to virtual memory; the OS and user processes run in the VAS. The OS manages the mapping from VAS to PAS through the use of the Memory Management Unit (MMU). The MMU is a part of the computer. The OS maintains a page table that maps each page of memory in the current VAS to a page of memory in the PAS. Typically the OS will maintain one page table per user level process and one for itself.
In a traditional system there are typically two address spaces – (1) the physical address space (PAS), i.e. the DRAM of the machine, and (2) the virtual address space (VAS). The machine boot code initially executes in the PAS and then quickly sets up and switches to virtual memory; the OS and user processes run in the VAS. The OS manages the mapping from VAS to PAS through the use of the Memory Management Unit (MMU). The MMU is a part of the computer. The OS maintains a page table that maps each page of memory in the current VAS to a page of memory in the PAS. Typically the OS will maintain one page table for each user level process and one for itself.
Memory Management Unit (MMU)

- Virtual Address to Physical Address Translation
  - Works in fixed-sized pages (tradeoff of size: large pages require only a few mapping entries, but less control)
  - Page Protection (a page can be invalid, read only, read-execute, or read-write-execute — never execute-only)

- Translation Look-aside Buffer (TLB)
  - The TLB is a cache for Virtual to Physical mappings

- Control registers
  - Page Table location
  - Current ASID
  - Alignment checking
There are different choices in how the computer implements the memory management unit especially in regards to what is done in hardware and what in software. It should come as no surprise, that the dominant surviving architectures are the most awkward and least flexible.

MMUs initially addressed the challenge of expensive RAM and the need for protection and isolation between processes. There have been few changes although there are many interesting proposals for alternatives such as finer-grained protection and java-aware paging.
Let's follow the path when the required mapping is not in the TLB.

1. There is a miss in the TLB. The hardware will walk the current process's page table to find the mapping. The page table structure will probably be more complicated than I'm showing here.
2. One of two things can happen:
   - The required mapping is found in the page table and placed in the TLB. The instruction is restarted and all proceeds normally. Note that in this case the hardware does all the work.
   - The required mapping is not present. An page fault exception is generated by the hardware and trapped into the operating system. The OS will do what it does to figure out the correct mapping.
3. The new translation is put into the current process's page table.
4. The OS resume's execution at the faulting instruction. Now the hardware TLB refill mechanism will work.
5. The hardware put the new mapping in the TLB and life goes on.
• 1: Miss in the TLB
• 2: Entry in the page table, load into TLB or entry not in page table, page fault happens and trap to the OS
• 3: OS creates new mapping, updates page table
• 4: OS resumes execution at faulting instruction, TLB miss but success in finding it in page table
• 5: TLB has correct mapping and execution continues
In a traditional system there are typically 2 address spaces – the virtual address space (VAS) and the physical address space (PAS). The OS and user processes run in the VAS. The OS manages the mapping from VAS to PAS through the use of the Memory Management Unit (MMU) provided in the processor. The OS maintains a page table that maps each page in the current VAS to a page in the PAS. Typically the OS will maintain one page table per user level process.
In a traditional system there are typically 2 address spaces – the virtual address space (VAS) and the physical address space (PAS). The OS and user processes run in the VAS. The OS manages the mapping from VAS to PAS through the use of the Memory Management Unit (MMU) provided in the processor. The OS maintains a page table that maps each page in the current VAS to a page in the PAS. Typically the OS will maintain one page table per user level process.
In a traditional system there are typically 2 address spaces – the virtual address space (VAS) and the physical address space (PAS). The OS and user processes run in the VAS. The OS manages the mapping from VAS to PAS through the use of the Memory Management Unit (MMU) provided in the processor. The OS maintains a page table that maps each page in the current VAS to a page in the PAS. Typically the OS will maintain one page table per user level process.
Because of the vast number of instructions that access memory, including the instruction fetch itself, the hardware TLB must be used to translate virtual addresses to machine addresses. So the common case should be that the TLB holds the virtual to machine mapping. To do this we can use a shadow page table. The real hardware MMU points to the shadow page table. The shadow page table holds virtual to machine mappings. The VMM page fault handler is responsible for filling in the appropriate entries in the shadow page table based on the guest page table and PhysMap.
Shadow Page Table

• The guest OS maintains a page table
  • but this is virtualized

• The VMM maintains a real page table
  • this is not seen by the Guest OS, but it “shadows” the actions of the Guest OS

• The shadow page table is the real one
Let's follow the path when the required mapping is not in the TLB.

1. There is a miss in the TLB. The hardware will walk the shadow page table to find the mapping. The page table structure will probably be more complicated than I'm showing here.
2. One of two things can happen:
   - The required mapping is found in the page table and placed in the TLB. The instruction is restarted and all proceeds normally. Note that in this case the hardware does all the work.
   - The required mapping is not present. An page fault exception is generated by the hardware and trapped into the VMM. The VMM needs to translate the virtual address to a machine address. It starts by walking the guest's page table to determine the virtual to physical mapping. Note that the layout of the guest page table will be determined by the hardware being virtualized.
3. Once the VMM finds the guest mapping one of two things can happen:
   - The guest mapping is not present. In this case the guest expects a page fault exception. So the VMM must generate an exception on the virtual cpu state and resume executing on the first instruction of the guest exception handler. This is called a **true page fault** because the hardware page fault results in a guest visible page fault.
   - If the guest mapping is present then the VMM must translate the physical page to a machine page. This is called a **hidden page fault** because the hardware fault is a fault that would not have occurred in non-virtualized system. In order to translate the physical page to machine page the VMM must look in a data structure that maps physical pages to machine pages. This data structure is defined by the VMM, for example PMap. (A) The VMM might have perform further processing if there is no machine page backing the physical page or in other special circumstances. More on this later.
4. The virtual to machine translation is complete. The new translation is put into the shadow page table.
5. The VMM restarts the guest instruction that faulted. Now the hardware TLB refill mechanism will work.
6. The hardware put the new mapping in the TLB and life goes on.
Issues with Emulated TLBs

- Guest page table consistency
  - Rely on Guest's need to invalidate TLB
  - Guest TLB invalidations caught by monitor, emulated

- Performance
  - Guest context switches flush entire software TLB

How to keep shadow page table consistent with the guest page table.
Normally, when OS changes a page table entry it tells the TLB to flush. This should be trapped and mimicked by the VMM.
When OS changes page table due to process switch, must update the TLB as well. VMM can loop and
Guest Write to CR3

- Virtual CR3
  - Guest Page Table
  - Guest Page Table
  - Guest Page Table

- Real CR3
  - Shadow Page Table
  - Shadow Page Table
  - Shadow Page Table
Guest Write to CR3

Virtual CR3

Guest Page Table

Guest Page Table

Guest Page Table

Guest Write to CR3

Virtual CR3

Real CR3

Shadow Page Table

Shadow Page Table

Shadow Page Table
Undiscovered Guest Page Table

- Virtual CR3
  - Guest Page Table
  - Shadow Page Table
- Real CR3
  - Guest Page Table
  - Shadow Page Table
  - Guest Page Table
Undiscovered Guest Page Table

Virtual CR3

Real CR3

Guest Page Table

Guest Page Table

Guest Page Table

Guest Page Table

Undiscovered Guest Page Table

Virtual CR3

Real CR3

Shadow Page Table

Shadow Page Table

Shadow Page Table

Shadow Page Table
Issues with Shadow Page Tables

• High cost of walking guest page tables in software
• High cost of VMM entries and exits
• Page Table Consistency
  • Guest may not need to invalidate TLB on writes to off-line page tables
  • Need to trace writes to shadow page tables to invalidate entries
• Memory Bloat
  • Cache guest page tables takes memory
  • Need to determine when guest has reused page tables
Memory Tracing

• Call a monitor handler on access to traced page
  • before guest reads
  • after guest writes
  • before guest writes
• Modules can install traces and register for callbacks
  • Binary
Trace Callout Path

1. There is a miss in the TLB. The hardware will walk the shadow page table to find the mapping. The page table structure will probably be more complicated than I'm showing here.

2. One of two things can happen:
   - The required mapping is found in the page table and placed in the TLB. The instruction is restarted and all proceeds normally. Note that in this case the hardware does all the work.
   - The required mapping is not present. An page fault exception is generated by the hardware and trapped into the VMM. The VMM needs to translate the virtual address to a machine address. It starts by walking the guest's page table to determine the virtual to physical mapping. Note that the layout of the guest page table will be determined by the hardware being virtualized.

3. Once the VMM finds the guest mapping one of two things can happen:
   - The guest mapping is not present. In this case the guest expects a page fault exception. So the VMM must generate an exception on the virtual cpu state and resume executing on the first instruction of the guest exception handler. This is called a true page fault because the hardware page fault results in a guest visible page fault.
   - If the guest mapping is present then the VMM must translate the physical page to a machine page. This is called a hidden page fault because the hardware fault is a fault that would not have occurred in non-virtualized system. In order to translate the physical page to machine page the VMM must look in a data structure that maps physical pages to machine pages. This data structure is defined by the VMM, for example PMap. (A) The VMM might have perform further processing if there is no machine page backing the physical page or in other special circumstances. More on this later.

4. The virtual to machine translation is complete. The new translation is put into the shadow page table.

5. The VMM restarts the guest instruction that faulted. Now the hardware TLB refill mechanism will work.

6. The hardware put the new mapping in the TLB and life goes on.
Trace Callout Path

- 1: Miss in TLB; hardware walks the page table

- 2: Find mapping and place in TLB or page fault happens; VMM translates virtual page to machine address by using guest page table

- 3: No Guest mapping, VMM generates page fault to Guest (a True Page Fault) or VMM must setup a mapping — based on PMAP — this is Hidden Page Fault

- 4: Update shadow page table

- 5: Restart Guest
Hiding the VMM

- VMM must be in the virtual address space
  - Exception and Interrupt Handlers
  - Binary Translator: cache and virtual state
- Translation cache intermingles guest and VMM accesses
  - must distinguish these: VMM full access, Guest not as much.
- Use x86 segmentation
  - VMM in high memory
  - Guest segments do not allow access to monitor
  - Use VMM segments when executing VMM code
- How to avoid Guest seeing its instructions have changed?
Memory Allocation

- Operating Systems like to allocate all of available memory in the machine
- Multiple VMs (+ optional host) —> too many cooks
  - Not enough RAM for all VMs
  - Difficult to get Guest OS's to do the right thing (since they are blind to the bigger picture)
- Solutions: Sharing, Ballooning, Compression
Sharing Pages

- Multiple VMs with same OS have many identical pages of Guest OS code. No need to keep multiple copies.
Page Sharing: Scan Candidate PPN

VM 1  VM 2  VM 3

Machine Memory

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

hash page contents
...2bd806af

hash table

hint frame
Page Sharing: Successful Match

VM 1  VM 2  VM 3

Machine Memory

shared frame

Hash: ...06af
Refs: 2
MPN: 123b

hash table
Sharing Pages

• Use hashing

• scan pages and look up hash

• recalculate hash

• If success, then mark as COW

• How often to scan, how to scan (randomly? Read-Only guest pages?)

• Get 67% sharing, increases as number of identical VMs increase.
Performance Page Sharing

Sharing metrics for a series of experiments consisting of identical Linux VMs running SPEC95 benchmarks.

The left graph indicates the absolute amounts of memory shared and saved increase smoothly with the number of concurrent VMs.

The right graph plots these metrics as a percentage of aggregate VM memory.
# Real-World Page Sharing

<table>
<thead>
<tr>
<th>Workload</th>
<th>Guest Types</th>
<th>Total</th>
<th>Saved</th>
<th>%</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.)
Compression
Ballooning

- OS's expect fixed amount of physical memory when they boot
- Sometimes there is memory pressure, sometimes there is wasted memory
- When several VMs running, want to dynamically allocate memory among them
Ballooning

- inflate balloon (+ pressure)
- deflate balloon (− pressure)
- Guest OS manages memory implicit cooperation
- may page out to virtual disk
- may page in from virtual disk

Guest OS

balloon

CSCI-GA.3033-010 (Fall 2014) Virtualization Technology Larry Rudolph
Allocating Resources

• Fairness is for kindergardeners

• Pay for “shares” — proportional allocation of resources

• But still not great allocation:

• Add “tax” for idle pages (not exactly fair)
Shares VS Working set

- Share-based approach
  - Reclaim from VM with min shares-per-page ratio [Waldspurger '95]
  - Problem: ignores usage, unproductive hoarding [Sullivan '00]
- Desired behavior
  - VM gets full allocation when actively using memory
  - VM may lose pages when working set shrinks
Reclaiming Idle Memory

- Tax on idle memory
  - Charge more for idle page than active page
  - Idle-adjusted shares-per-page ratio

\[ \rho = \frac{S}{P \cdot (f + k \cdot (1 - f))} \]

S = # of shared pages owned by aVM
P = # of allocated pages owned by aVM
f = # of active pages within allocated pages
k = 1/(1-T) = cost within given tax rate

- Tax rate (0<=T<1)
  - Explicit administrative parameter
- Default rate = 75%

Reclaim most idle memory
Measuring Idle Memory

- ESX Server uses a **statistical sampling approach** to obtain aggregate VM working set estimates directly, without any guest involvement. Each VM is sampled independently.
  - n VM’s physical pages are selected randomly
  - For each time the guest access to a sampled page, a touched page count t is incremented.
    - The next guest access to a sampled page reestablish these mappings, increasing a touched page count t.
  - A **statistical estimate** of the fraction f of memory actively accessed by the VM is f = t/n.
  - By default, ESX Server samples 100 pages for each 30 second period.
Measuring Active Memory

- **Experiment**
  - Single Windows VM
  - Memory "toucher" app
  - Active memory estimate

- **Statistical sampling**
  - Small random subset of pages
  - Software access bits [Joy ’81]
  - Moving averages [Kim ’01]

- **Behavior**
  - Rapid response to ↑ usage
  - Gradual response to ↓ usage
  - Windows “zero page thread”
Idle Memory Tax: 0%

- **Experiment**
  - 2 VMs, 256 MB, same shares
  - VM1: Windows boot+idle
  - VM2: Linux boot+dbench
  - Solid: usage, Dotted: active

- **Change tax rate**
  - **Before:** no tax
  - VM1 idle, VM2 active
  - get same allocation
Idle Memory Tax: 75%

- Experiment
  - 2 VMs, 256 MB, same shares
  - VM1: Windows boot+idle
  - VM2: Linux boot+dbench
  - Solid: usage, Dotted: active

- Change tax rate

- After: high tax
  - Redistribute VM1 → VM2
  - VM1 reduced to min size
  - VM2 throughput improves 30%
Other Allocation Policy

- Memory performance estimates
- effect of changing allocations
- miss-rate curve construction
- Dynamic defragmentation for large pages
- I/O MMU Isolation
- SSD allocation among VMs