CSCI 3753
Operating Systems

Virtual Machines

Chapters 16

Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 12/07/17

Terminology

- "Virtual machine" is a loaded term
  - E.g. Java Virtual Machine refers to a runtime environment (software) that can execute Java bytecode
- "VM" is a loaded abbreviation
  - JVM (Java Virtual Machine), Virtual Memory
- For our purposes, we will talk about Virtual Machine Monitors (VMM)
  - VMM is software that allows multiple guest OSes to run concurrently on one physical machine
    - Each guest runs on a virtual machine
  - VMM is sometimes called a *hypervisor*

Virtual Machine Monitors (VMMs)

- VMMs are everywhere
- Industry commitment
  - Software: Vmware, Xen, …
  - Hardware: Intel-VT, AMD-V
    - If Intel and AMD add it to their chip, you know it's serious …
- An old idea, actually developed by IBM in 60's

What is a VMM?

- We have seen that an OS already virtualizes
  - Syscall, processes, virtual memory, file system, sockets, …
  - Applications program to this interface
- A process already is given the illusion that it has its
  - Own memory, via virtual memory
  - Own CPU, via time slicing
  - Own I/O devices, via device-independent I/O

What is a VMM?

- A VMM extends this idea and virtualizes an entire physical machine
  - Interface supported is the hardware
    - OS defines a higher level interface
  - VMM provides the illusion that software has full control over the hardware (of course, VMM is in control)
  - VMM “applications” run in virtual machines
- Implications
  - You can boot an OS in a VM
  - Run multiple instances of an OS on same physical machine
  - Run different OSes simultaneously on the same machine

Why VMMs?

- Resource utilization
  - Machines today are powerful, want to multiplex their hardware
    - E.g. Cloud providers
  - Can migrate VMs from one machine to another without shutdown
- Software use and development
  - Can run multiple OSes simultaneously
    - E.g. No need to dual boot
  - Can do system development at the user level
- Many other cool applications
  - Debugging, emulation, security, speculation, fault tolerance
- Common theme is manipulating applications/services at the granularity of a machine

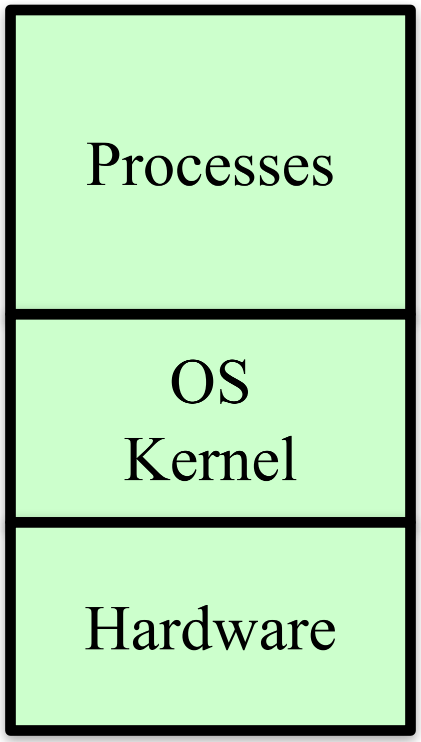
Goals of Virtualization

- Fidelity
  - OSes and applications work the same without modification
    - Except timings …
    - Although we may modify the OS a bit
- Isolation
  - VMM protects resources and VMs from each other
- Performance
  - VMM is another software layer --- overhead
  - VMware
    - CPU-intensive apps: 2-10% overhead
    - I/O intensive apps: 25-60% overhead
  - An overwhelming majority of guest instructions are executed by the hardware without VMM intervention

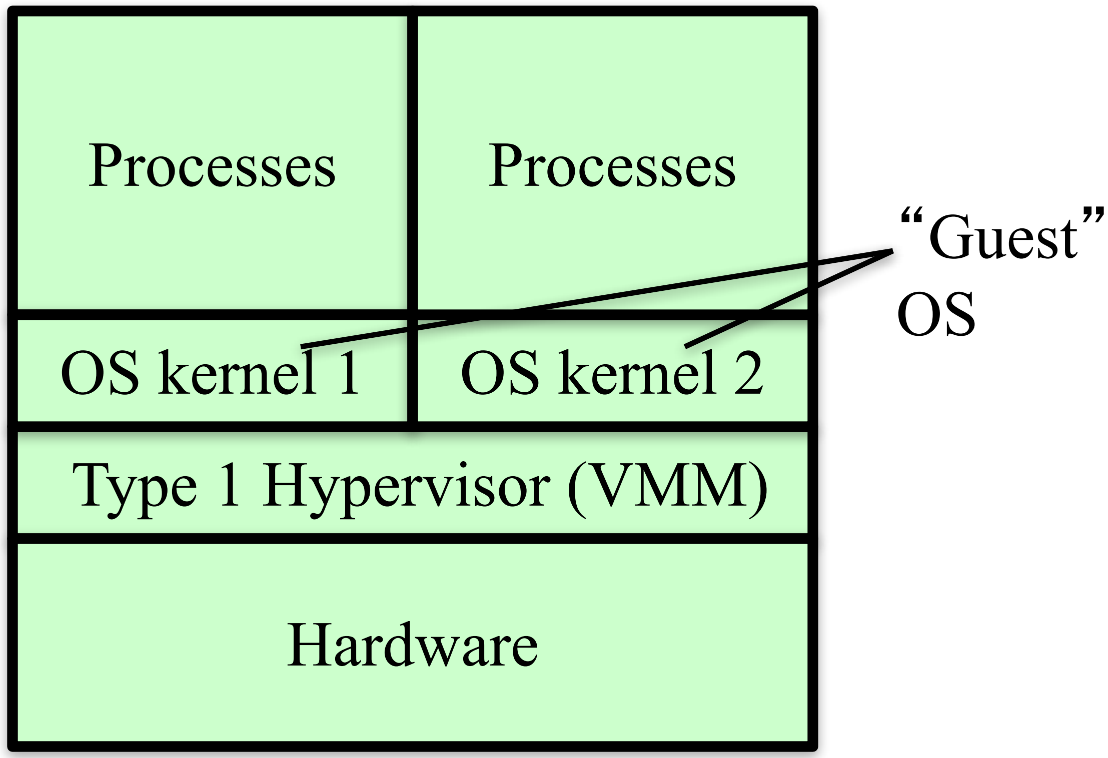
Announcements

- Final Exam
  - Monday, 12/18, 4:30-6:00 PM in class
  - Special accommodation: Monday, 12/18, 4:30 - … in ECES 112 C.
  - Material
    - Memory management, mass storage structure, file systems, security and protection, virtual machines
    - Chapters 8, 9, 10, 11, 12, 14, 15 and 16
    - Lecture sets 12, 13, 14, 15, 16, 17, 18, 19 and 20
    - One question from midterm exam
- Format
  - Similar to midterm exam
  - Closed notes, closed book, …

Type 1 Hypervisor



Traditional OS



Type 1 Hypervisor: runs directly on HW
Example: IBM's CP/CMS, Citrix XenServer, VMware ESXi, Microsoft Hyper-V

Type 1 Hypervisor

- *Type 1* (or *native*) hypervisors run directly on the host's hardware
  - A guest operating system thus runs on another level above the hypervisor
- Also known as Full virtualization
- The hypervisor provides illusion of identical hardware as the real system
  - Guest OS doesn’t have to be recompiled, and doesn’t even realize it’s executing on a VM
  - Example VMs: Microsoft Virtual Server, VMWare ESX Server
  - <u>Disadvantage</u>: Virtualized applications can run much slower, because of the many layers of fully virtualized abstraction between them and the hardware

Full Virtualization

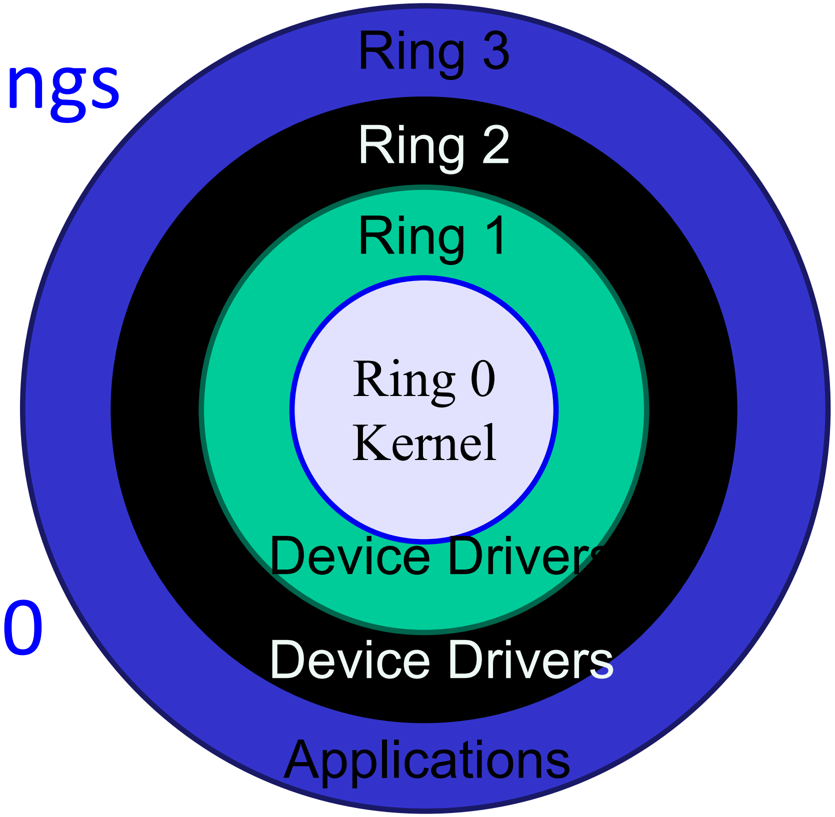
- There are two approaches in full virtualization
  - Hardware assisted virtualization
    - Provides architectural support that facilitates building a virtual machine monitor and allows guest OSs to be run in isolation
  - Software assisted virtualization
    - the virtual machine simulates enough hardware to allow an unmodified "guest" OS to be run in isolation.

Type 1 Hypervisor: How it Works

- VMM can interpret every instruction from guest OSes and execute them
- Problem: emulating/interpreting every instruction are not good options – too much software overhead
- Goal: Want to create a virtual machine that executes at close to native speeds on a CPU
- Solution: Have the guest OS execute normally, directly on the CPU, except that it is not in kernel mode.
  - Any privileged instructions invoked by the guest OS will be trapped to the hypervisor, which is in kernel mode.

Protected Mode

- Most modern CPUs support protected mode
- x86 CPUs support three rings with different privileges
  - Ring 0: OS kernel
  - Ring 1, 2: device drivers
  - Ring 3: userland
- Most OSes only use rings 0 and 3



Challenges With Virtual Hardware

Issue: x86 is not designed with virtualization in mind

1. Dealing with privileged instructions
   - OSes expect to run with high privilege (ring 0)
   - How can the VMM enable guest OSes to run in user mode (ring 3)?
2. Managing virtual memory
   - OSes expect to manage their own page tables
   - MMU supports only one level of virtualization
   - How can the VMM translate between a guest's page tables and the hosts page tables?

Privileged Instructions

- OSes rely on many privileges of ring 0
  - cri, sti, popf – Enable/disable interrupts
  - hlt – Halt the CPU until the next interrupt
  - mov cr3, 0x00FA546C – install a page table
  - Install interrupt and trap handlers
  - Etc…
- However, guest OSes run in user mode
- VMM must somehow virtualize privileged operations

Using Exceptions for Virtualization

- Ideally, when a guest executes a privileged instruction in ring 3, the CPU should generate an exception
- Example: suppose the guest executes hlt
  1. The CPU generates a protection exception
  2. The exception gets passed to the VMM
  3. The VMM can emulate the privileged instruction
     - If guest 1 runs hlt, then it wants to go to sleep
     - VMM can do guest1.yield(), then schedule guest 2

Problem: x86 Doesn’t Except Properly

Binary Translation

- x86 assembly cannot be virtualized because some privileged instructions don't generate exceptions
- Workaround: translate the unsafe assembly from the guest to safe assembly
  - Known as binary translation
  - Performed by the VMM
  - Privileged instructions are changed to function calls to code in VMM

Type 1 Hypervisor: How it Works

- The hypervisor then emulates only these privileged instructions and when done passes control back to the guest OS, also known as a “VM entry”
- This way, most ordinary (non-privileged) instructions operate at full speed, and only privileged instructions incur the overhead of a trap, also known as a “VM exit”, to the hypervisor/VMM.
- This approach to VMs is called *trap-and-emulate*

Binary Translation Example

Guest OS Assembly

```
do_atomic_operation:
    cli
    mov eax, 1
    xchg eax, [lock_addr]
    test eax, eax
    jnz spinlock
    …
    …
    mov [lock_addr], 0
    sti
    ret
```

Translated Assembly

```
do_atomic_operation:
    call [vmm_disable_interrupts]
    mov eax, 1
    xchg eax, [lock_addr]
    test eax, eax
    jnz spinlock
    …
    …
    mov [lock_addr], 0
    call [vmm_enable_interrupts]
    ret
```

Binary Translation: Pros and Cons

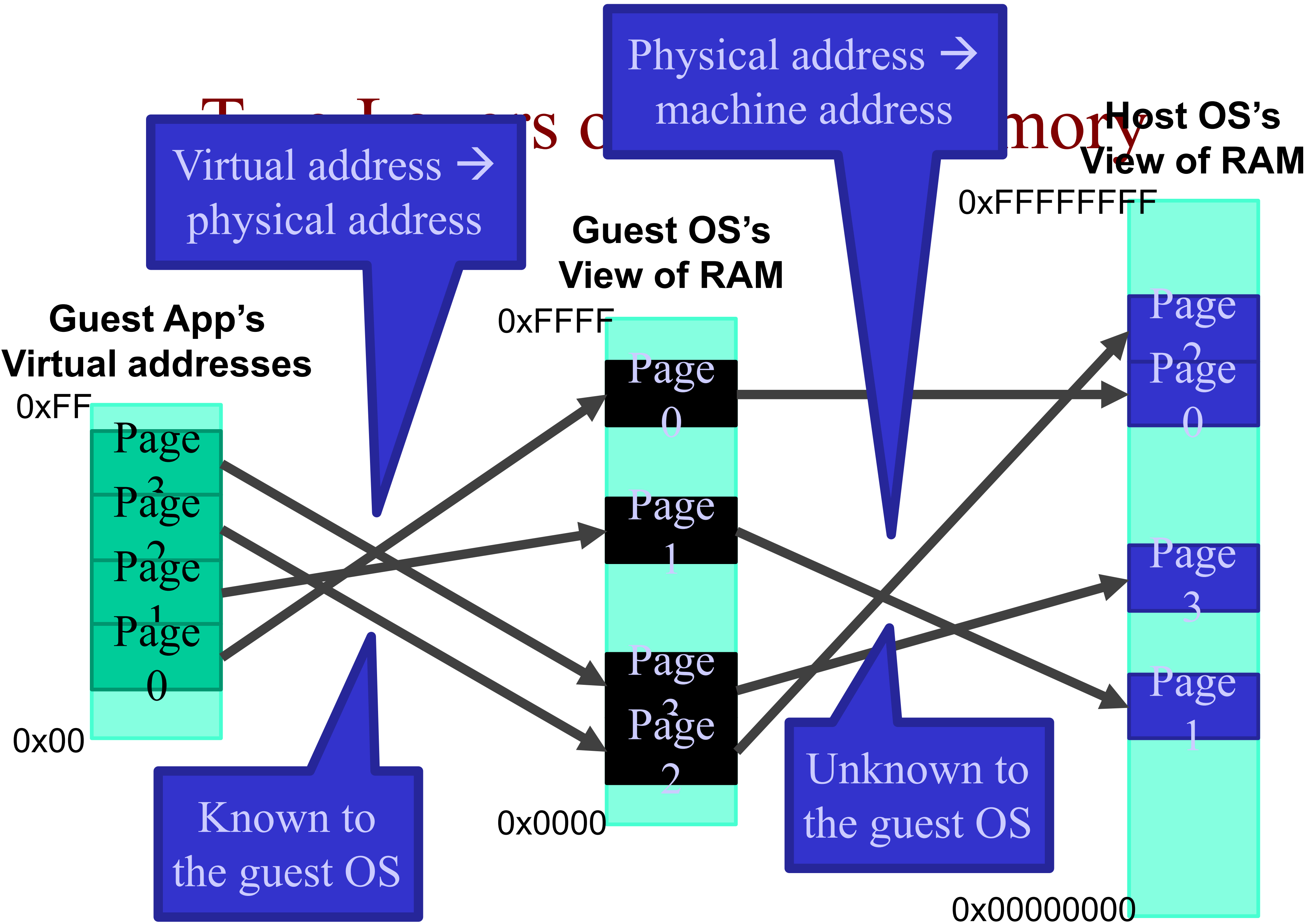
- Advantages of binary translation
  - It makes it safe to virtualize x86 assembly code
  - Translation occurs dynamically, on demand
    - No need to translate the entire guest OS
  - App code running in the guest does not need to be translated
- Disadvantages
  - Translation is slow
  - Wastes memory (duplicate copies of code in memory)
  - Translation may cause code to be expanded or shortened
    - Thus, jmp and call addresses may also need to be patched

Caching Translated Code

- Typically, VMMs maintain a cache of translated code blocks
  - LRU replacement
- Thus, frequently used code will only be translated once
  - The first execution of this code will be slow
  - Other invocations occur at native speed

Problem: How to Virtualize the MMU?

- On x86, OS expects that it can create page tables and install them in the *cr3* register
  - The OS believes that it can access physical memory
- However, virtualized guests do not have access to physical memory
- Using binary translation, the VMM can replace writes to *cr3*
  - Store the guest's root page in the virtual CPU *cr3*
  - The VMM can now walk to guest's page tables
- However, the guest's page tables cannot be installed in the physical CPU…

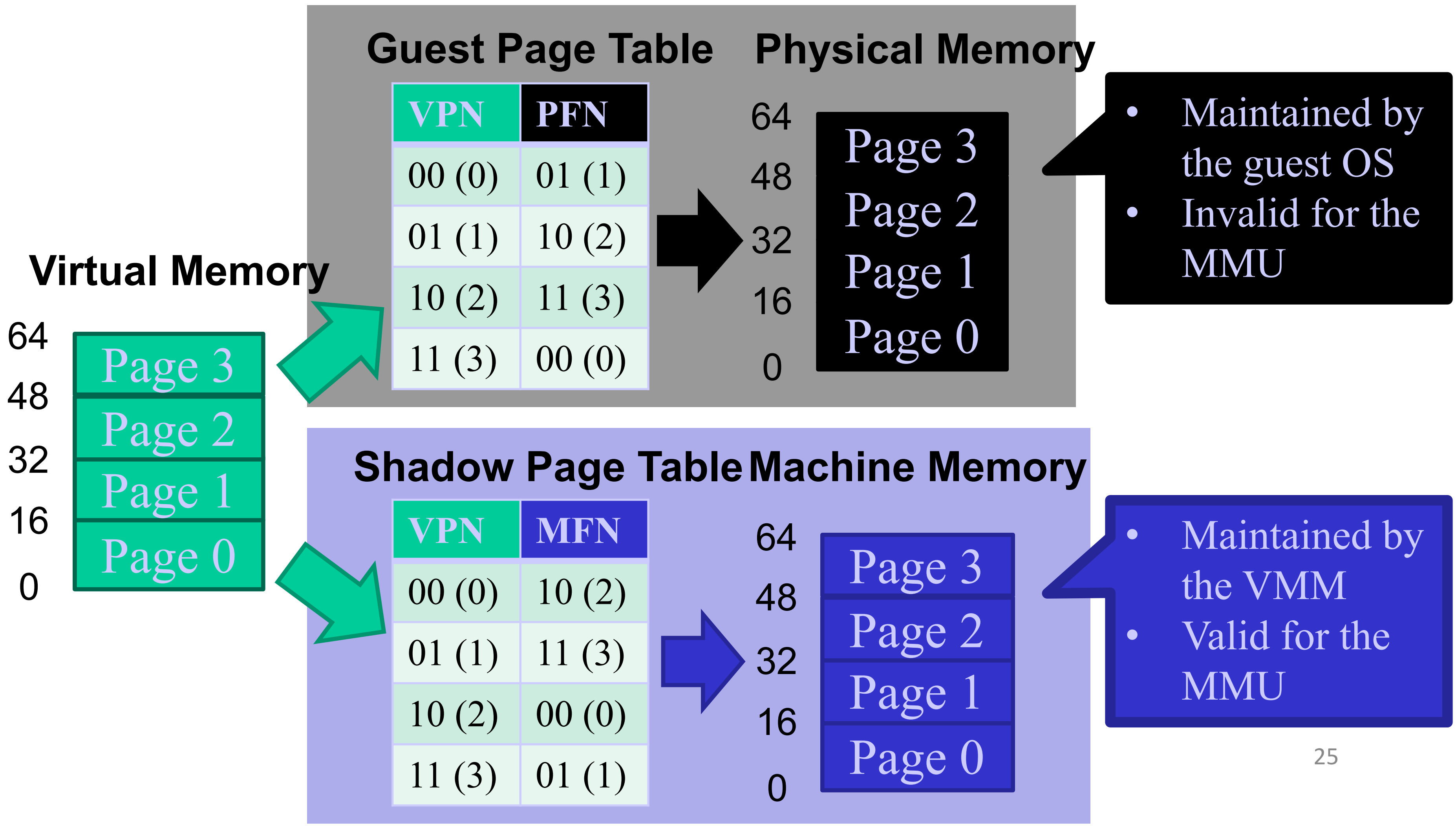

Physical address → machine address
Virtual address → physical address
Host OS's View of RAM
0xFFFFFFFF
Guest OS's View of RAM
0xFFFF
Guest App's Virtual addresses
0xFF
Page 3
Page 2
Page 1
Page 0
0x00
Page 0
Page 1
Page 3
Page 2
0x0000
Page 2
Page 0
Page 3
Page 1
0x00000000
Known to the guest OS
Unknown to the guest OS

Guest's Page Tables Are Invalid

- Guest OS page tables map virtual page numbers (VPNs) to physical frame numbers (PFNs)
- Problem: the guest is virtualized, doesn't actually know the true PFNs
  - The true location is the machine frame number (MFN)
  - MFNs are known to the VMM
- Guest page tables cannot be installed in *cr3*
  - Map VPNs to PFNs, but the PFNs are incorrect
- How can the MMU translate addresses used by the guest (VPNs) to MFNs?

Shadow Page Tables

- Solution: VMM creates shadow page tables that map VPN → MFN (as opposed to VPN→PFN)



Guest Page Table

VPN	PFN
00 (0)	01 (1)
01 (1)	10 (2)
10 (2)	11 (3)
11 (3)	00 (0)

Shadow Page Table

VPN	MFN
00 (0)	10 (2)
01 (1)	11 (3)
10 (2)	00 (0)
11 (3)	01 (1)

Building Shadow Tables

- Problem: how can the VMM maintain consistent shadow pages tables?
  - The guest OS may modify its page tables at any time
  - Modifying the tables is a simple memory write, not a privileged instruction
    - Thus, no helpful CPU exceptions :(
- Solution: mark the hardware pages containing the guest's tables as read-only
  - If the guest updates a table, an exception is generated
  - VMM catches the exception, examines the faulting write, updates the shadow table
- Special attention needs to be paid to page fault handling

Shadow page tables: Pros and Cons

- The good: shadow tables allow the MMU to directly translate guest VPNs to hardware pages
  - Thus, guest OS code and guest apps can execute directly on the CPU
- The bad:
  - Double the amount of memory used for page tables
    - i.e. the guest's tables and the shadow tables
  - Context switch from the guest to the VMM every time a page table is created or updated
    - Very high CPU overhead for memory intensive workloads

Hardware Techniques

- Modern x86 chips support hardware extensions designed to improve virtualization performance

1. Reliable exceptions during privileged instructions
   - Known as AMD-V and VT-x (Intel)
   - Released in 2006
   - Adds vmrun/vmexit instructions (like sysenter/sysret)
2. Extended page tables for guests (second level addr translation)
   - Known as RVI (AMD) and EPT (Intel)
   - Released in 2008
   - Adds another layer onto existing page table to map PFN→MFN

Pros and Cons of AMD-V and VT-x

- Greatly simplifies VMM implementation
  - No need for binary translation
  - Simplifies implementation of shadow page tables
- … however, sophisticated VMMs still use binary translation in addition to vmenter/vmexit
  - Some operations are **much** slower when using *vmexit* vs. binary translation
  - VMM observes guest code that causes frequent vmexits
  - Hot spots may be binary translated or dynamically patched to improve performance

Pros and Cons of RVI and EPT

- Huge performance advantages vs. shadow page tables
- Memory overhead
  - … but not as much as shadow page tables
- TLB misses are twice as costly
  - page tables twice as deep, hence it takes twice as long to resolve TLB misses

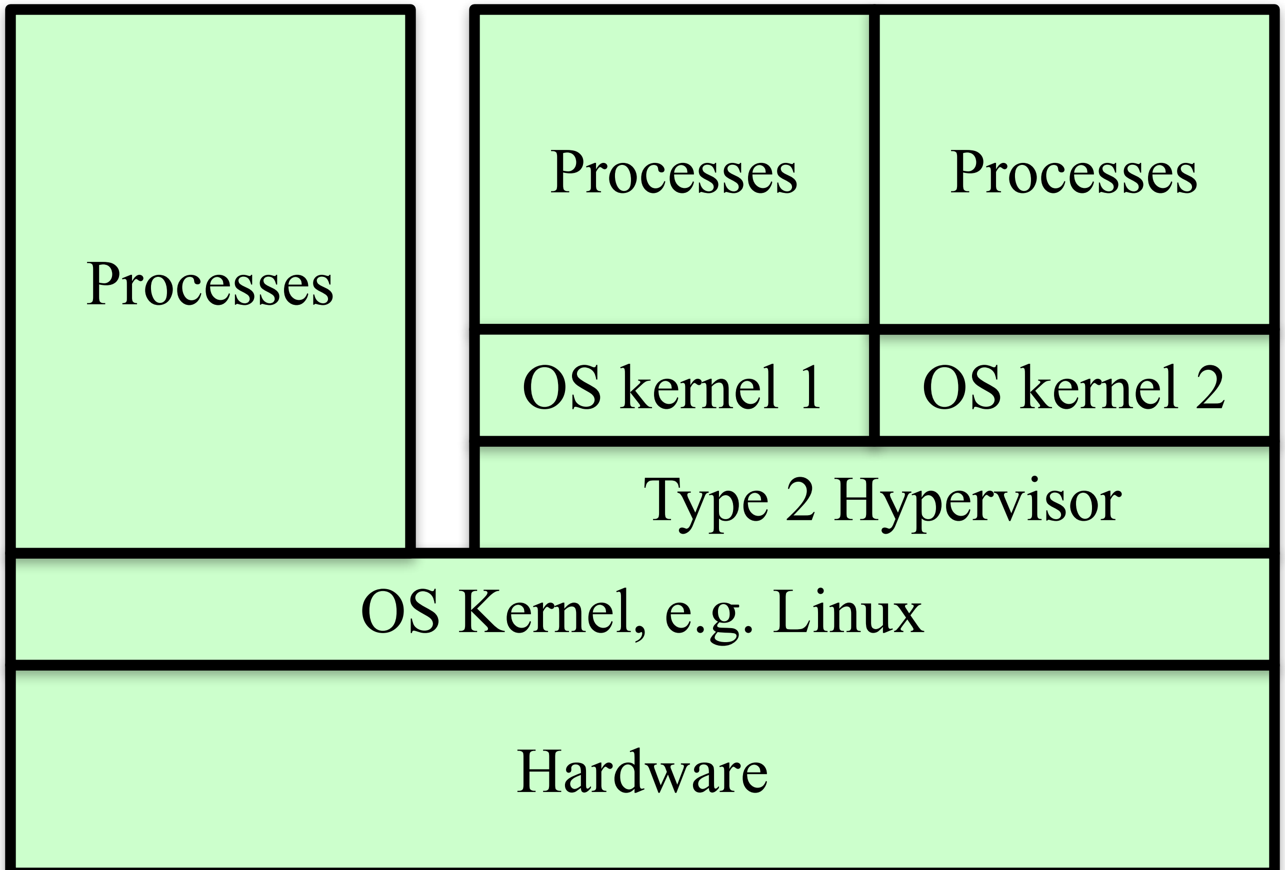
Configuring Your VMM

- Advanced VMMs like VMWare give you three options
  1. Binary translation + shadow page tables
  2. AMD-V/VT-x + shadow page tables
  3. AMD-V/VT-x + RVI/EPT
     - Fastest by far
     - But, requires very recent, expensive CPUs
- Which is best?
  - Choosing between 1 and 2 is more difficult
  - For some workloads, 2 is much slower than 1
  - Run benchmarks with your workload before deciding on 1 or 2

Type 2 Hypervisor

- Problem: (Full virtualization) it takes a lot of work to virtualize an arbitrary guest OS
  - VMM implementation is very complicated
  - Even with hardware support, performance issues remain
- What if we require that guests be modified to run in the VMM
  - How much work is it to modify guests to "cooperate" with the VMM?
  - Will VMM implementation be simpler?
  - Can we get improved performance?

Type 2 Hypervisor



Type 2 Hypervisor: runs on an OS
Example: KVM, Virtualbox

Type 2 Hypervisor

- *Type 2* (or *hosted*) hypervisors run within a conventional operating system environment
  - The hypervisor layer as a distinct 2nd software level
  - A guest operating system runs at the third level
- There are two approaches
  - Operating system-level virtualization
    - Allows multiple isolated and secure virtualized servers to run on a single physical server.
    - The OS kernel is used to implement the "guest" environment
      - Applications running in a given "guest" environment view it as a stand-alone system
  - Paravirtualization

Paravirtualization

- *Paravirtualization* addresses the performance issue:
  - Paravirtualization provides specially defined 'hooks' to allow the guest(s) and host to request and acknowledge certain tasks, which would otherwise be executed in the hypervisor
  - An OS no longer sees full virtualization of hardware, but instead must be specially recompiled to work with this particular hypervisor, exploiting special optimizations to speed performance

Paravirtualization

- Involves modifying the OS to run in the virtualized environment as a VM → needs to be recompiled
- Virtual machine does not necessarily simulate hardware, but instead (or in addition) offers a special API that can only be used by modifying the "guest" OS
  - VMware tries to run code whenever possible directly on CPU without emulation, e.g. user code
- Example VMs: Xen, Parallels, VMware Workstation

Hypercalls

- The Xen VMM exports a hypercall API
  - Methods replace privileged instructions offered by the hardware
    - E.g halt CPU, enable/disable interrupts, install page table
  - Guest OS can detect if it's running directly on hardware or on Xen
    - In the former case, typical ring 0 behavior is used
    - In the latter case, hypercalls are used
- If a guest executes a privileged instruction, crash it
  - Xen VMM makes no attempt to emulate privileged instructions
  - Simplifies Xen VMM implementation

Modifying Guests

- How much work does it take to modify a guest OS to run on Xen?
  - Linux: 3000 lines (1.36% of the kernel)
    - Including device drivers
  - Windows XP: 4620 lines (0.04% of the kernel)
    - Device drivers add another few hundred lines of code
- Modification isn’t trivial, but its certainly doable

Virtualization

- Virtualization has made a huge resurgence in the last 20 years
- Today, all OSes and most CPUs have direct support for hosting virtual machines, or becoming virtualized
- Virtualization underpins the cloud
  - E.g. Amazon EC2 rents virtual machines at low costs
  - Hugely important for innovation