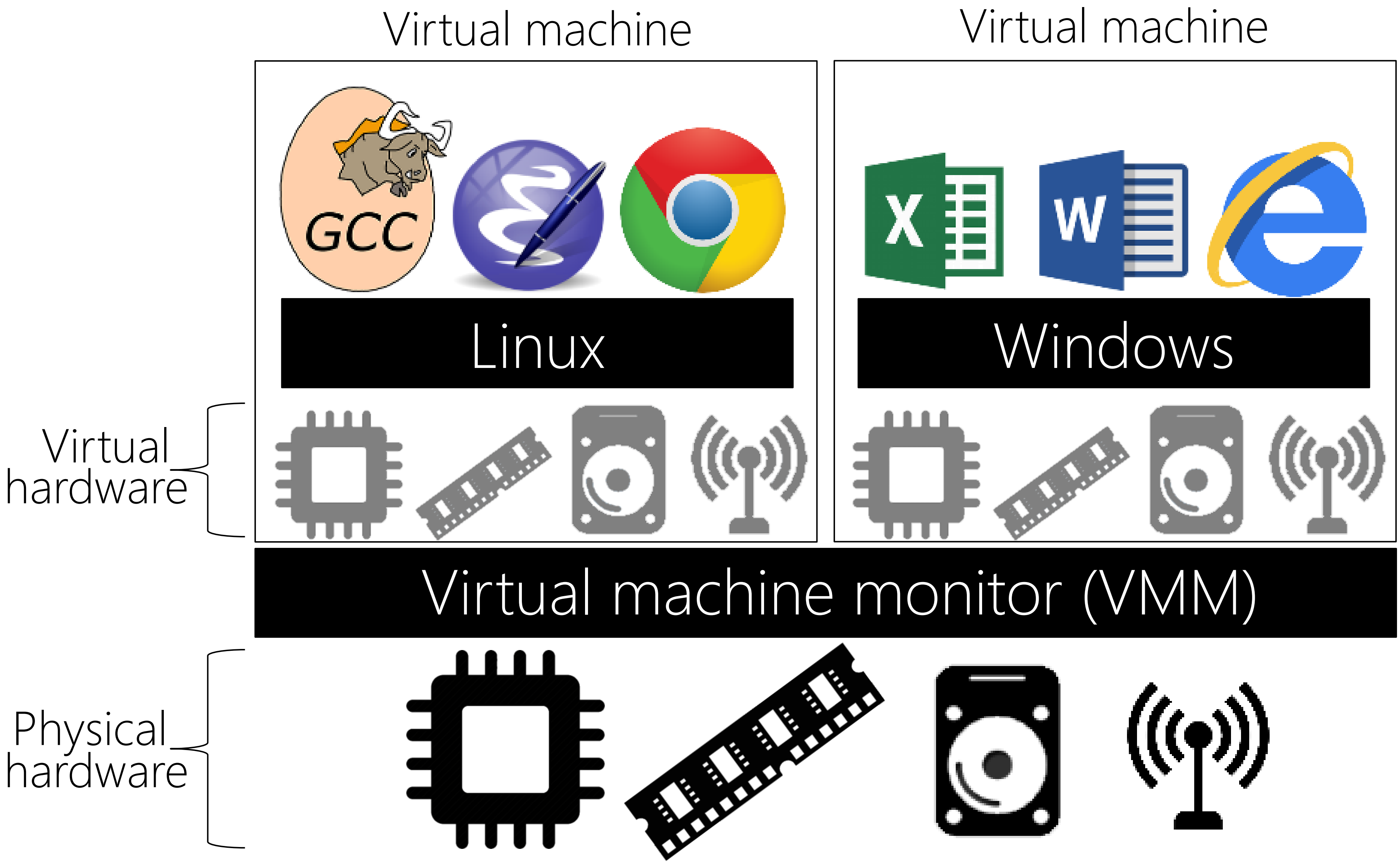
Virtualization

The Basic Idea

- Introduce a layer of abstraction that sits <u>above</u> the hardware, but <u>beneath</u> the OS (or software that directly accesses hardware)
  - Expose virtual hardware that is backed by physical hardware
  - Virtual machine monitor (VMM) implements the virtualization interface, enforces the illusion of isolated virtual machines

The Basic Idea

- Introduce a layer of abstraction that sits above the hardware, but beneath the OS (or software that directly accesses hardware)



VMM Interface vs. OS Interface

- OS provides a high level of abstraction
  - CPUs exposed via illusion of thread-private CPUs
  - Physical memory exposed via virtual memory and process abstractions
  - Devices exposed via file system abstractions and file descriptor operations (e.g., write()s on a socket)
- VMM provides a low level of abstraction
  - Software appears to be running on raw hardware, with direct access to physical memory and devices (so each VM usually includes its own OS)
- Both an OS and a VMM try to isolate different tenants (processes/VMs), and enforce fairness w.r.t. usage of physical hardware

Why Is Virtualization Useful?

- Multiplexing physical hardware in datacenters
  - A customer wants her application to run on an isolated machine . . . but her application may have low hardware utilization!
  - Bad solution: Datacenter operator grants a separate physical machine to each customer application
  - Good solution: Datacenter operator runs multiple VMs atop a single physical machine

  - Physical machine will be highly utilized even if individual VMs are lightly loaded
  - Datacenter operators can buy fewer physical machines!

Why Is Virtualization Useful?

- Multiplexing physical hardware in datacenters
  - A customer wants her application to run on an isolated machine . . . but her application may have low hardware utilization!
  - Bad solution: Datacenter operator grants a separate physical machine to each customer application
  - Good solution: Datacenter operator runs multiple VMs atop a single physical machine

- Physical machine will be highly utilized even if individual VMs are lightly loaded
- Datacenter operators can buy fewer physical machines!
- But . . . SLAs! Can’t oversubscribe physical machines *too* much.

Why Is Virtualization Useful?

- Security: Isolation between VMs is useful if VMs don't trust each other, and/or host doesn't trust guests
  - Ex: A multi-tenant datacenter like Amazon's EC2 runs code from multiple parties

M3

This family includes the M3 instance types and provides a balance of compute, memory, and network resources, and it is a good choice for many applications.

Features:

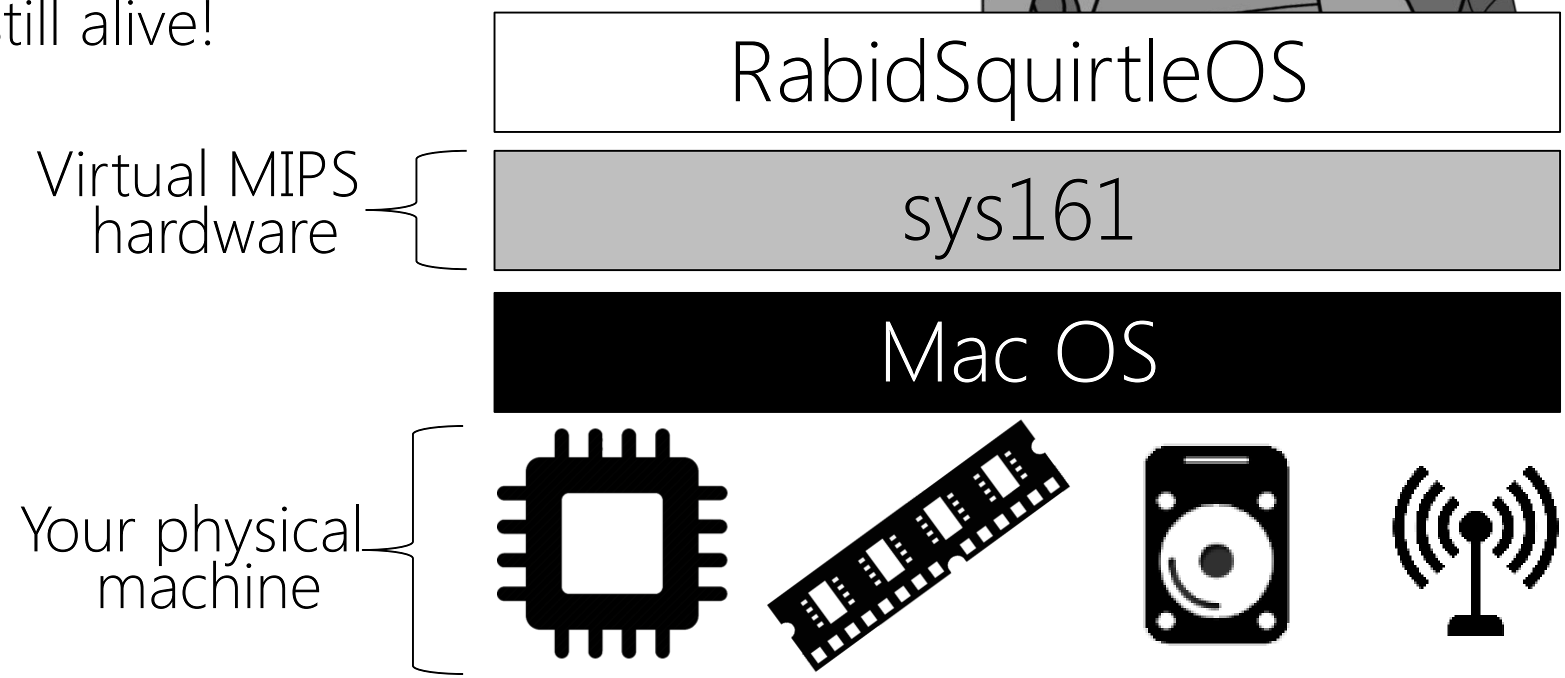
- High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors*
- SSD-based instance storage for fast I/O performance
- Balance of compute, memory, and network resources

Model	vCPU	Mem (GiB)	SSD Storage (GB)
m3.medium	1	3.75	1 x 4
m3.large	2	7.5	1 x 32
m3.xlarge	4	15	2 x 40
m3.2xlarge	8	30	2 x 80

  - Ex: On a desktop machine, user can load untrusted content in a VM (e.g., email attachment, software from unknown source)

Why Is Virtualization Useful?

- Improved productivity for developers
  - Ex: You can run Mac OS as your host, and Linux as your guest; do fun stuff on Mac OS, do dev stuff in Linux VM
  - Ex: A kernel developer loads her kernel in a VM so that, when the kernel crashes, her dev machine is still alive!



Why Is Virtualization Useful?

- Backwards compatibility: Use virtualization to run programs from an earlier age of humanity
  - Ex: Electronic archives who must run old software
  - Ex: Video game emulators for dead platforms

How Can We Implement Virtualization?

Virtualization Approach #1: Hosted Interpretation

- Run the VMM as a regular user application atop a host OS
  - VMM maintains a software-level representation of physical hardware
  - VMM steps through the instructions in the code of the VM, updating the virtual hardware as necessary

```
while(1){
    curr_instr = fetch(virtHw.PC);
    virtHw.PC += 4;
    switch(curr_instr){
        case ADD:
            int sum = virtHw.regs[curr_instr.reg0] +
                      virtHw.regs[curr_instr.reg1];
            virtHw.regs[curr_instr.reg0] = sum;
            break;
        case SUB:
            //...etc...
```

- Hosted interpretation is used by sys161 (MIPS), Bochs (x86), and several emulators for video game platforms

Virtualization Approach #1: Hosted Interpretation

- Good: Easy to handle privileged instructions
  - The guest OS will want to read and write privileged registers, manipulate the MMU, send commands to IO devices, etc.
  - The interpreter can handle privileged instructions according to a policy
    - Ex: All VM disk IO is redirected to a backing file in the host OS (similar to an OS161 SFS disk)
    - Ex: VM cannot access the network at all, or can only access a predefined set of remote IP addresses
- Good: Provides "complete" isolation (no guest instruction is directly executed on host hardware)
- Good: Can debug even low-level boot code in the guest!
- Bad: Emulating a modern processor is difficult!
- Bad: Interpretation is slow! [Ex: Two orders of magnitude for Bochs]

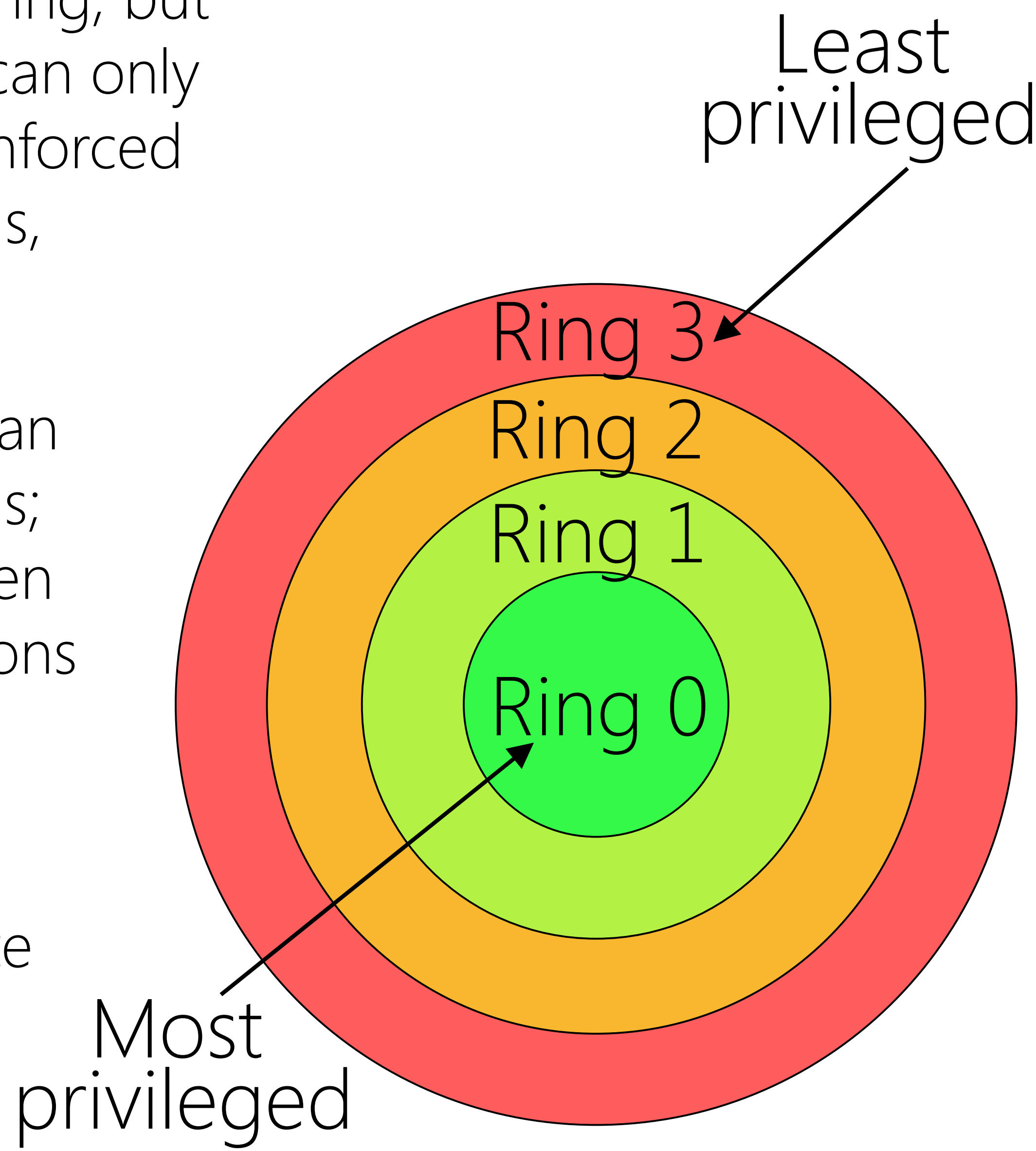
Virtualization Approach #2: Direct Execution w/Trap and Emulate

. . . but first, some x86 horrors.

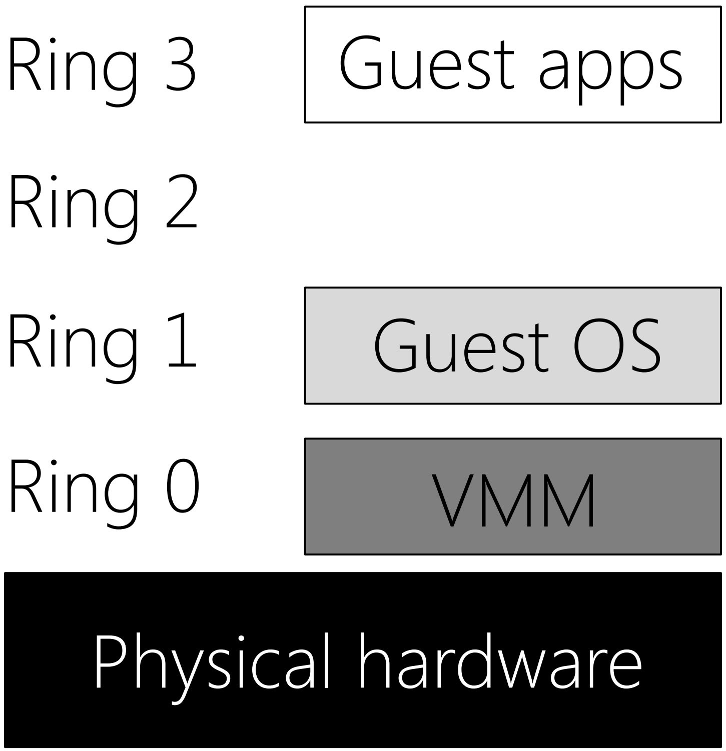
Observation 1: Code in a more privileged ring can read and write memory in a lower privilege ring, but function calls between rings can only happen through hardware-enforced mechanisms (e.g., system calls, "gates" (DON'T ASK))

Observation 2: Only Ring 0 can execute privileged instructions; Rings 1, 2, and 3 will trap when executing privileged instructions

In a normal setup, the OS executes in Ring 0, and the user-level applications execute in Ring 3.



Virtualization Approach #2: Direct Execution w/Trap and Emulate



[Assumes that guest code uses ISA of physical hardware!]

- Guest apps can't tamper with the guest OS due to ring protections
- Guest apps and guest OS can't tamper with VMM due to ring protections
- When the guest OS executes a privileged instruction, it will trap into the VMM
- When a guest app generates a system call or exception, the app will trap into the VMM
- VMM's trap handler uses a policy to decide what to do (e.g., emulate the instruction, kill the VM, etc.)

Virtualization Approach #2: Direct Execution w/Trap and Emulate

- This approach requires that a processor be "virtualizable"
  - Privileged instructions cause a trap when executed in Rings 1—3
  - Sensitive instructions access low-level machine state that should be managed by an OS or VMM
    - Ex: Instructions that modify segment/page table registers
    - Ex: IO instructions
  - Virtualizable processor: all sensitive instructions are privileged
- If a processor is virtualizable, a VMM can interpose on any sensitive instruction that the VM tries to execute
  - VMM can control how the VM interacts with the "outside world" (i.e., physical hardware)
  - VMM can fool the guest OS into thinking that guest OS runs at the highest privilege level (e.g., if guest OS invokes sensitive instruction to check the current privilege level)

Virtualization Approach #2: Direct Execution w/Trap and Emulate

- For many years, x86 chips were not virtualizable! For example, on the Pentium chip, 17 instructions were not virtualizable.
- Ex: **`push`** can push a register value onto the top of the stack
  - **`%cs`** register contains (among other things) 2 bits representing the current privilege level
  - A guest OS running in Ring 1 could **`push %cs`** and see that the privilege level isn't Ring 0!
  - To be virtualizable, **`push`** should cause a trap when invoked from Ring 1, allowing the VMM to push a fake **`%cs`** value which indicates that the guest OS is running in Ring 0

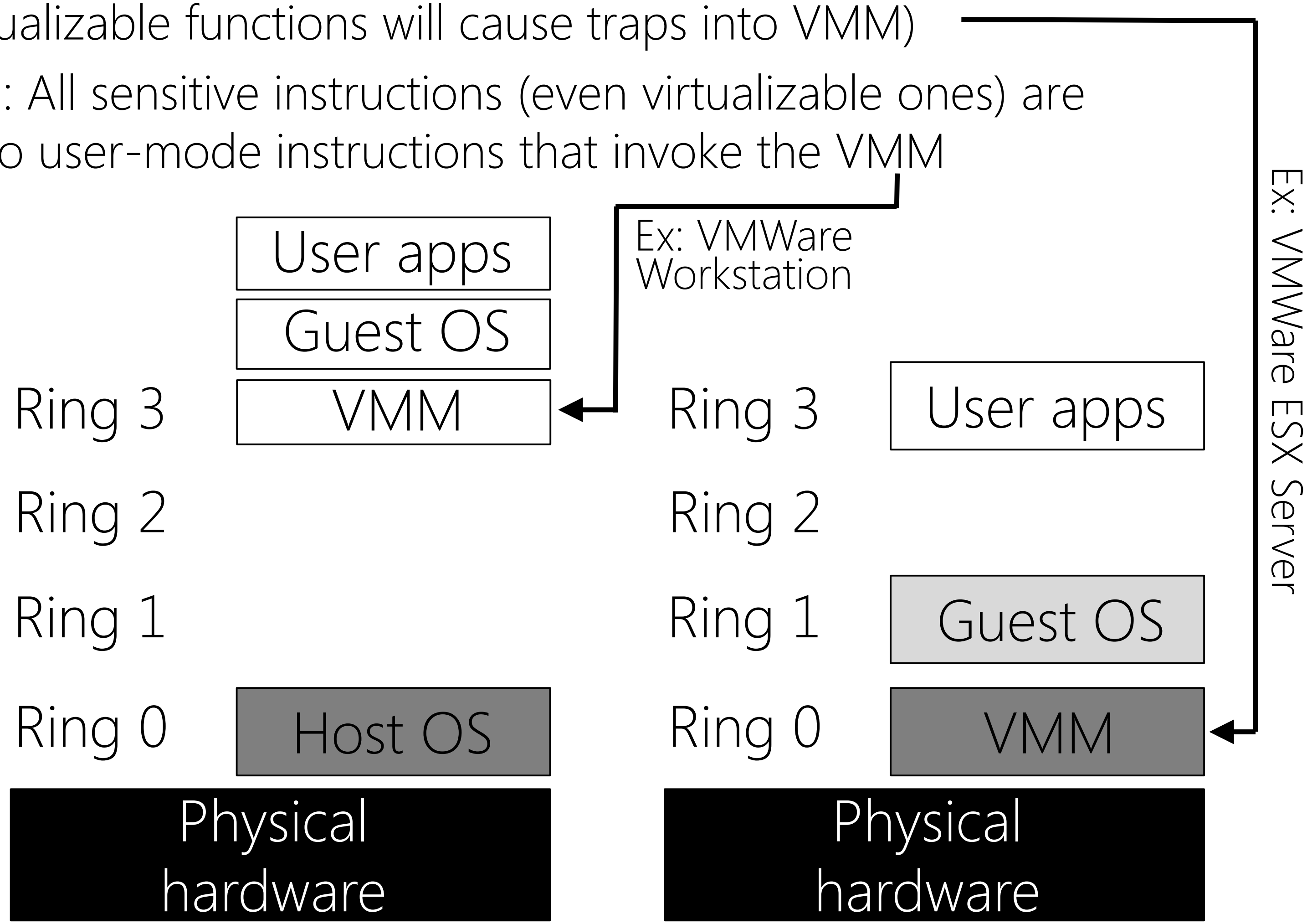
Virtualization Approach #2: Direct Execution w/Trap and Emulate

- For many years, x86 chips were not virtualizable! For example, on the Pentium chip, 17 instructions were not virtualizable.
- Ex: **`pushf/popf`** read/write the **`%eflags`** register using TOS
  - Bit 9 of **`%eflags`** enables interrupts
  - In Ring 0, **`popf`** can set bit 9, but in Ring 1, CPU silently ignores **`popf`**!
  - To be virtualizable, **`pushf/popf`** should cause traps in Ring 1 so that the VMM can detect when guest OS wants to changes its interrupt level (meaning that the VMM should change which interrupts it forwards to the guest OS)

How Can We Handle Non-virtualizable Processors?

Virtualization Approach #3: Direct Execution w/Binary Translation

- VMM dynamically rewrites nonvirtualizable instructions so that they invoke VMM
  - Bare metal VMM: VMM only needs to translate nonvirtualizable instructions (sensitive virtualizable functions will cause traps into VMM)
  - Hosted VMM: All sensitive instructions (even virtualizable ones) are translated into user-mode instructions that invoke the VMM



Virtualization Approach #3: Direct Execution w/Binary Translation

- Good: Guest code doesn't have to be modified by developers (translation is done automagically by VMM), so you can run off-the-shelf guest OSes and applications
- Good: The vast majority of instructions run at bare-metal speed
- Bad: Implementing the VMM is tricky!
  - Ex: A processor with a software-managed TLB
  - We must distinguish between:
    - Virtual memory: What the guest applications see
    - Physical memory: What the guest OS manipulates
    - Machine memory: The actual memory that the underlying machine has (and is managed by the VMM)

Direct Execution w/Binary Translation (Virtualizable Processor with Software-managed TLB)

Guest App (Ring 3)	Guest OS (Ring 1)	VMM (Ring 0)
Memory access causes a TLB miss -> trap		
		TLB handler of the VMM: Invoke the guest OS TLB handler
	TLB handler of the guest OS: Extract VPN from VA; do page table lookup; if present and valid, get PFN and update TLB	
		Trap handler of the VMM (unprivileged code trying to write TLB entry): guest OS wants to install VPN-to-PFN mapping, but VMM installs VPN-to-MFN mapping; return to guest OS TLB handler
	Guest OS executes the "return from trap" instruction	
		Trap handler of the VMM (unprivileged code trying to execute privileged instruction): Restart guest app's faulting instruction
Previously faulting memory access now succeeds		

VPN: Virtual page number
PFN: Physical frame number
MFN: Machine frame number

Direct Execution w/Binary Translation (Processor with Hardware-managed TLB)

Guest App (Ring 3) | Guest OS (Ring 1) | VMM (Ring 0)

Memory access causes a TLB miss -> trap

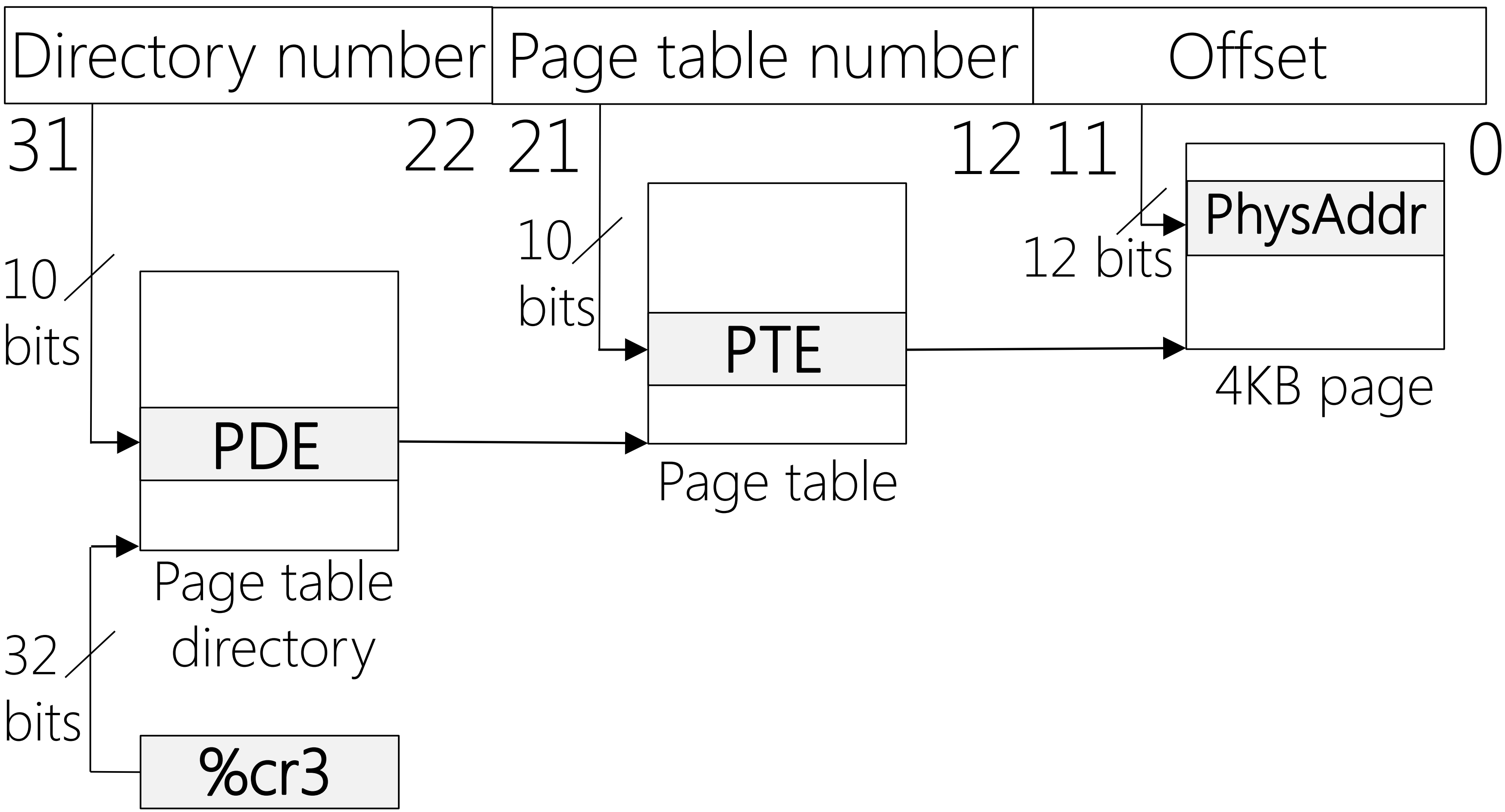
OH HAI

I'M A HARDWARE

If page is present and valid, TLB is filled and guest app automatically restarted: No opportunity for VMM mediation!

Direct Execution with Binary Translation and Hardware-controlled TLBs: Shadow Page Tables on x86

- When the guest OS in Ring 1 context switches to a new process, the guest OS sets the page table pointer %cr3



Direct Execution with Binary Translation and Hardware-controlled TLBs: Shadow Page Tables on x86

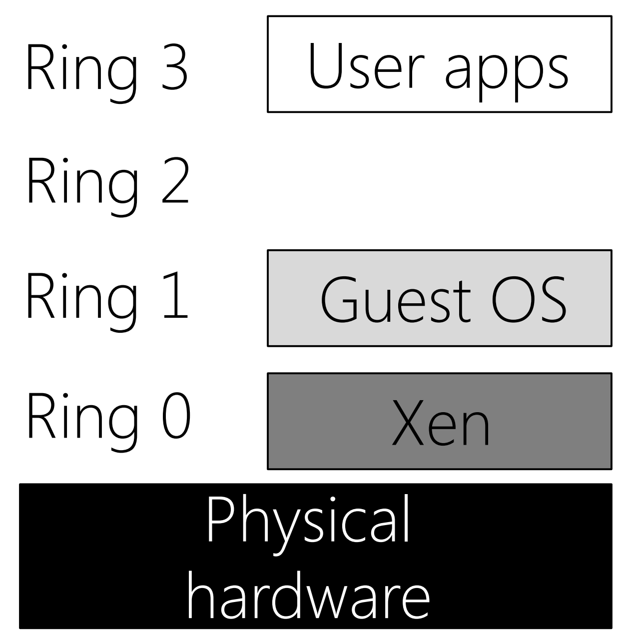
- When the guest OS in Ring 1 context switches to a new process, the guest OS sets the page table pointer %cr3
  - Assigning to %cr3 is a privileged operation!
  - So, the guest OS will trap to the VMM
  - VMM can install its own mappings for the new process
- VMM also marks the machine pages containing the guest app’s page table structures as read-only
  - The VMM knows how to interpret %cr3 and the page table format because the page table format is hardware-defined and thus well-known!
  - So, when the guest OS tries to modify a PTE, a “write attempted on read-only page” fault will invoke the VMM, who can then modify the PTE according to a VMM policy
- Overall result: VMM can always control “real” machine-level address translation

Virtualization Approach #4: Direct Execution w/ Hardware-assisted Virtualization

- Direct execution with binary translation is tricky, so . . .
- . . . let's add virtualization support to the hardware!
- Ex: Intel's VT-x
  - Adds two new modes of execution
    - VMX root mode: Equivalent to x86 without VT-x; VMM runs in this mode in Ring 0
    - VMX non-root mode: Still has rings, but sensitive operations trigger a transition to root mode, even in Ring 0
  - Adds a new hardware structure
    - Virtual machine control structure (VMCS): Configured by the VMM to determine *which* sensitive operations cause non-root code to transition to root code
    - Example of sensitive operations: Writing to %cr3; receiving an interrupt

Virtualization Approach #5: Direct Execution w/Paravirtualization

- Direct execution with binary translation is tricky, so . . .
- . . . let's rewrite the guest OS to remove sensitive-but-unprivileged instructions!
  - Define a subset of x86 that is virtualizable
  - Port the guest OS to the virtualizable subset
- Example: The Xen hypervisor
  - Guest OS is modified to inform Xen of changes to page table mappings (avoids VMM chicanery with read-only page table structures)
  - Guest OS modified to install "fast" sys call handler
    - Xen validates `int` handler at registration time, then installs it directly
    - Validated handler directly invokes guest OS in Ring 0 (in contrast to "slow" path in which system call exception invokes Xen handler in Ring 0, which then invokes guest OS handler in Ring 1)
  - Guest apps are unmodified



Virtualization Approach #5: Direct Execution w/Paravirtualization

- Good: Don't need any tricky binary translation, so paravirtualization should be faster than direct execution with binary translation
  - Paravirtualization has fewer context switches and less bookkeeping logic
- Maybe bad: Someone must port an OS to the virtualizable x86 subset . . . is this easier or harder than implementing binary translation logic?
  - Various flavors of Linux and BSD have been ported to Xen. So, porting is definitely possible for real OSes!
  - Xen can also leverage hardware-assisted virtualization! So, Xen can be used as a VMM for non-paravirtualized OSes like Windows

Virtualization Approach #6: OS-level Virtualization

- “Container” technologies are the new hotness (e.g., Docker, LXC)
  - A container is a group of Linux processes
  - Linux cgroups (“control groups”) limit the CPU, memory, network, and disk resources that the container can use; also assigns priorities
  - Linux namespaces isolate the ability of the container to see various resources
    - Ex: mnt namespace controls which part of the file system is visible to container
    - Ex: pid namespace isolates the pids that a container can manipulate
    - Ex: net namespace controls which NICs, iptables rules a container uses
- Good: Don’t need to rewrite/translate guest applications
- Good: High performance
  - Avoids context transitions between guest apps, guest OS, and VMM
  - Avoids “mark guest OS page table structures as read-only” nonsense
  - Don’t have to boot an entire OS to launch an application!
  - Don’t have to dedicate resources for an entire OS per application
- Good: Snapshots are smaller than with traditional VMs
  - Don’t need to include OS state in the snapshot!
- Bad: Guest applications are forced to use a particular host OS’s interface

Virtualization: A Summary

1. Hosted interpretation

   Easy to handle privileged instructions, can debug all guest code (even low-level code), but has bad performance and a complex VMM implementation

2. Direct execution with trap-and-emulate

   Good performance, works with unmodified guest code, but requires a virtualizable processor

3. Direct execution with binary translation

   Good performance, works with unmodified guest code and non-virtualizable processors, but implementing the VMM is tricky

4. Direct execution with hardware-assisted virtualization

   Good performance, works with unmodified guest code, is probably the future of virtualization once hardware context switches between root and non-root are optimized

5. Paravirtualization

   Good performance, but requires modification of guest OS

6. OS-level virtualization

   Good performance, works with unmodified guest code, small VM snapshots, fast VM launch, but VMs must use OS interface of the host