



---

D u k e S y s t e m s

**CPS 510**

# **Virtual Machines and Hypervisors**


**Jeff Chase**

**Duke University**

# The magic of desktop containers



```
heap-20-base — root@5634111c2c1a: / — docker exec -it cps310 bash — 101x24
```


```
[root@5634111c2c1a:/# uname -r
4.19.76-linuxkit
[root@5634111c2c1a:/# cd /
[root@5634111c2c1a:/# ls
bin  cps310  etc  lib  lib64  media  opt  root  sbin  sys  usr
boot dev  home lib32 libx32 mnt  proc run  srv  tmp  var
[root@5634111c2c1a:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="18.04.5 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.5 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/privacy-policy"
VERSION_CODENAME=bionic
UBUNTU_CODENAME=bionic
root@5634111c2c1a:/#
```



```
chase — -bash — 80x25
```

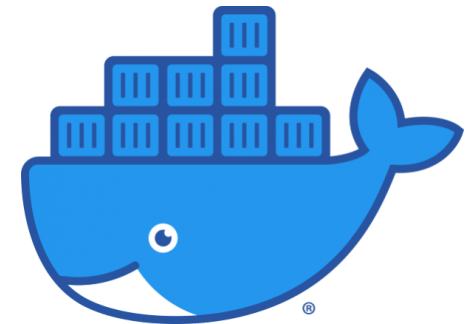
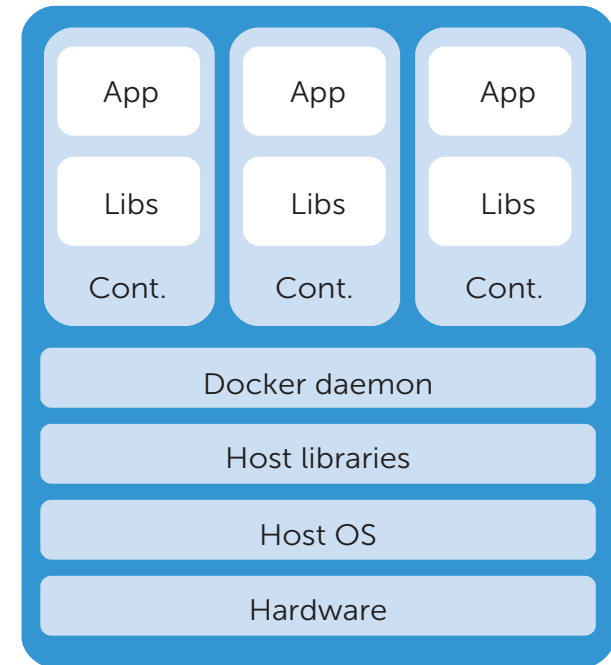
```
[chase20:~ chase$ sw_vers
ProductName:    Mac OS X
ProductVersion: 10.15.5
BuildVersion:   19F101
[chase20:~ chase$ cat /System/Library/CoreServices/SystemVersion.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>ProductBuildVersion</key>
  <string>19F101</string>
  <key>ProductCopyright</key>
  <string>1983-2020 Apple Inc.</string>
  <key>ProductName</key>
  <string>Mac OS X</string>
  <key>ProductUserVisibleVersion</key>
  <string>10.15.5</string>
  <key>ProductVersion</key>
  <string>10.15.5</string>
  <key>iOSSupportVersion</key>
  <string>13.5</string>
</dict>
</plist>
chase20:~ chase$
```



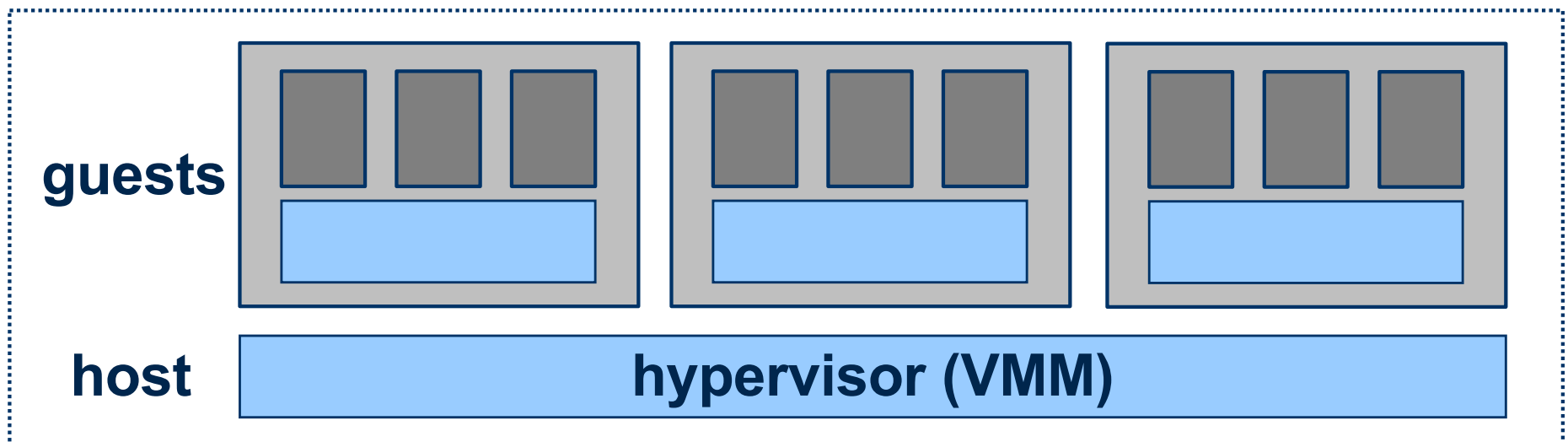
# Containers (e.g., Docker): we get it

- Containers on a machine run over a common OS kernel.
- Kernel provides foundational abstractions to build them.
  - Control groups (cgroups)
  - Namespaces
  - Filesystem layers
  - Capabilities
- But how can Linux containers run on a MacOS system?
  - **Where's the kernel?**

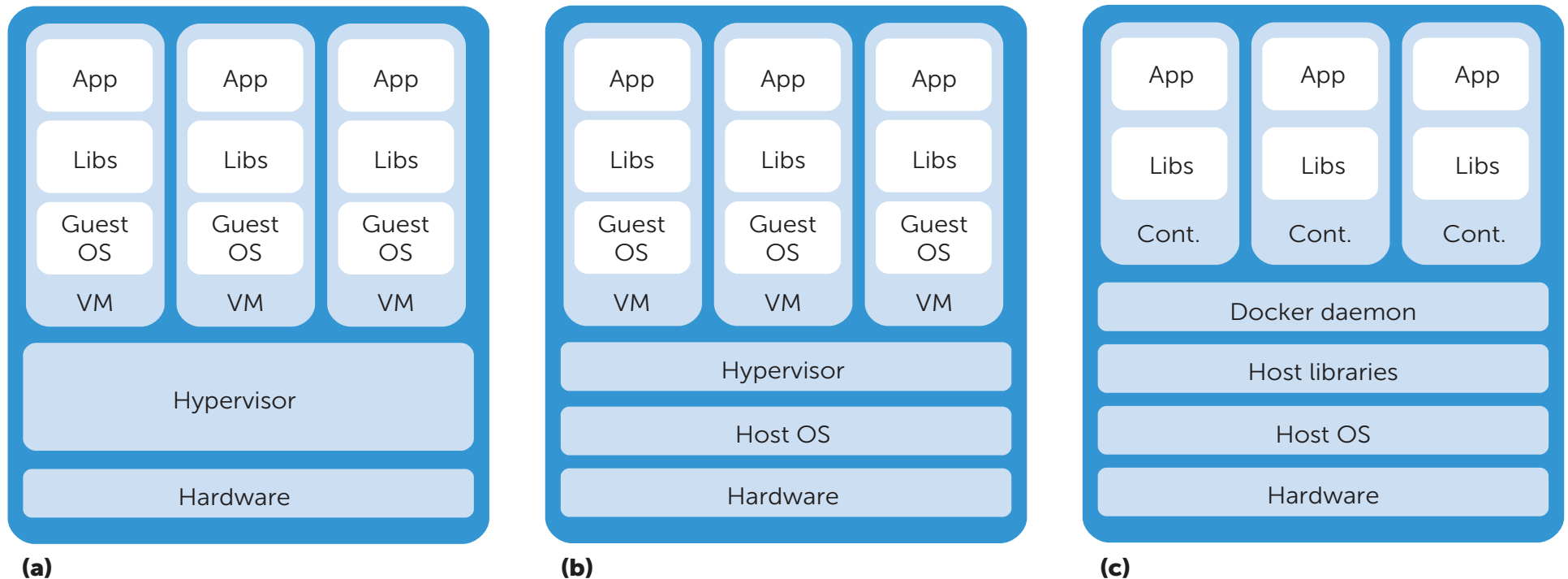


# Native virtual machines (VMs)

- Slide a **hypervisor** within/underneath the host OS kernel.
  - New OS layer: also called **virtual machine monitor (VMM)**.
- Hypervisor implements **virtual machines (VMs)**.
- Run multiple guest VM **instances** on a shared computer.
  - Each VM is a sandboxed/isolated context for any entire OS.
  - VM instance “looks the same” to guest OS as a physical machine.



# Three models for virtual servers



**FIGURE 1.** Comparing various application runtime models: (a) a type 1 hypervisor, (b) a type 2 hypervisor, and (c) a container.

“Type 1” VMM:  
bare-metal hypervisor

“Type 2” VMM:  
built into MacOS,  
Windows, Linux (KVM)

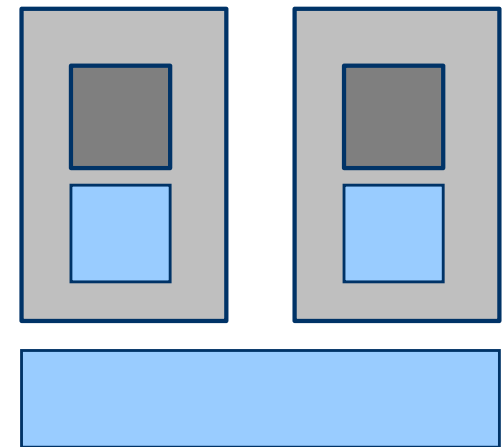
Note: containers may  
also be layered above  
VMs, within a guest OS.  
**You can have it all.**

# Understanding native VMs

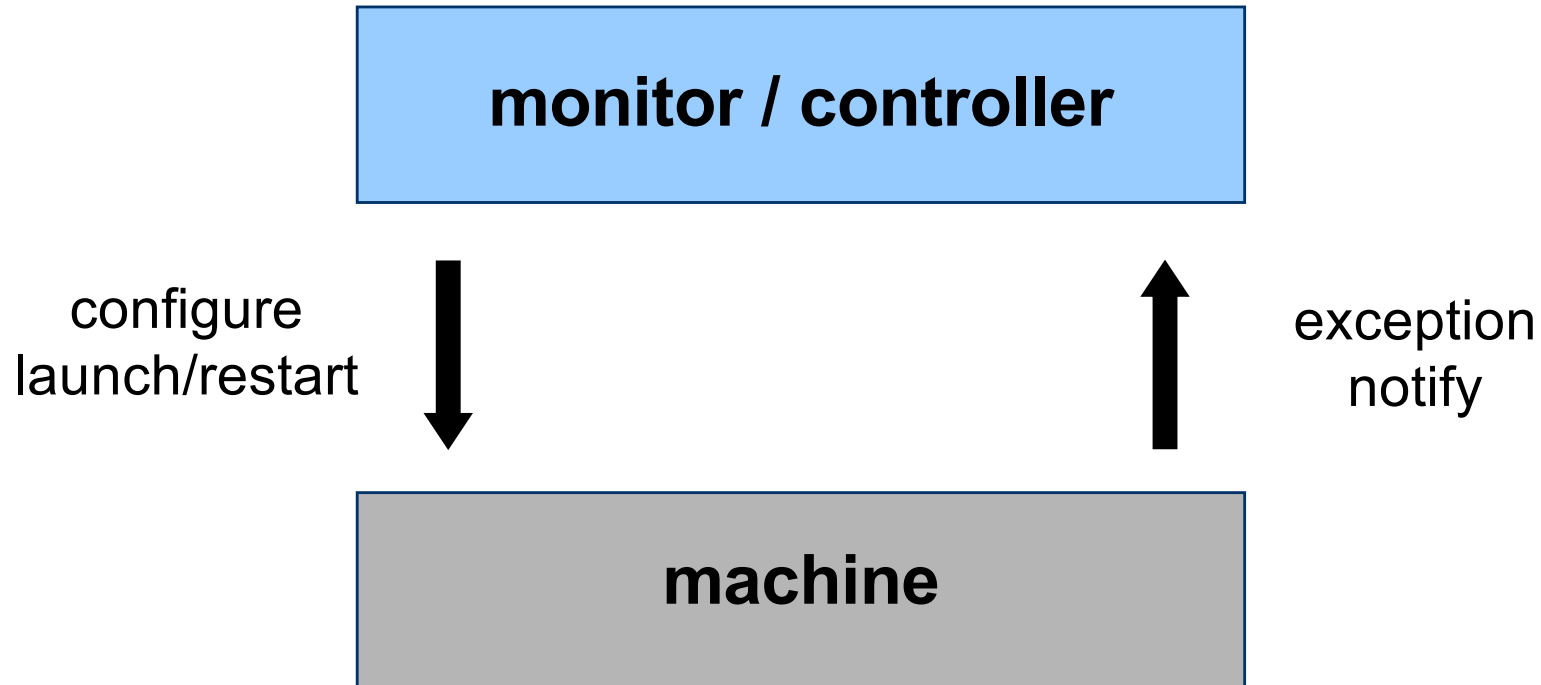
- Native VMs are distinct from e.g. Java VM (JVM).
  - JVM runs within an ordinary process.
  - Emulates/interprets ISA for an abstract machine.
    - E.g., Java bytecode is one such ISA.
  - Java compiler translates to ISA; JVM interprets it.
  - JVMs support that ISA on every real machine.
  - So compiled Java applications are “transportable”.
  - Slow! But it helps that JVMs have an embedded “just in time” (JIT) compiler to the underlying machine ISA.
- On native VMs, software is built for the native machine ISA, and runs at full speed on the hardware. **But how?**

# Virtual is real!

- The figures give an idea that a VM runs “above” the hypervisor, i.e., the hypervisor **emulates** the VM.
- But it doesn’t! A thread in a process in a VM runs right straight on the metal at full speed.
- Until there is a trap, fault, or interrupt! In a VM, some of these force a **VM exit** to host mode (hypervisor).
- Hypervisor is not “between” the VM and the CPU.
- Rather, it **intercepts** certain events and handles them in software.
- And it can inject events into a VM.



# The p-p-paradigm

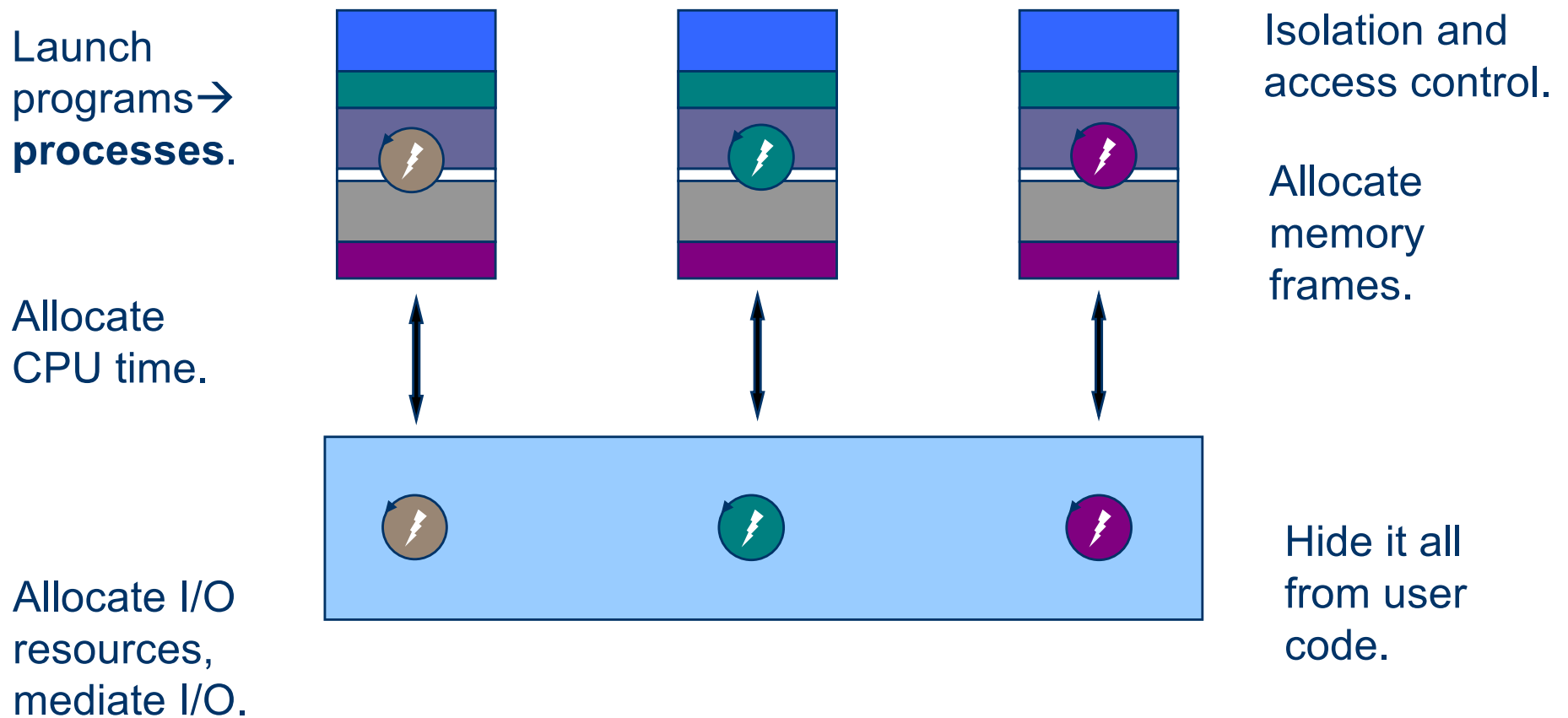


Machine runs according to its configuration. If it encounters a condition that requires controller to intervene, it suspends processing and generates an exception for the controller.

Compare to **Limited Direct Execution** in OSTEP.



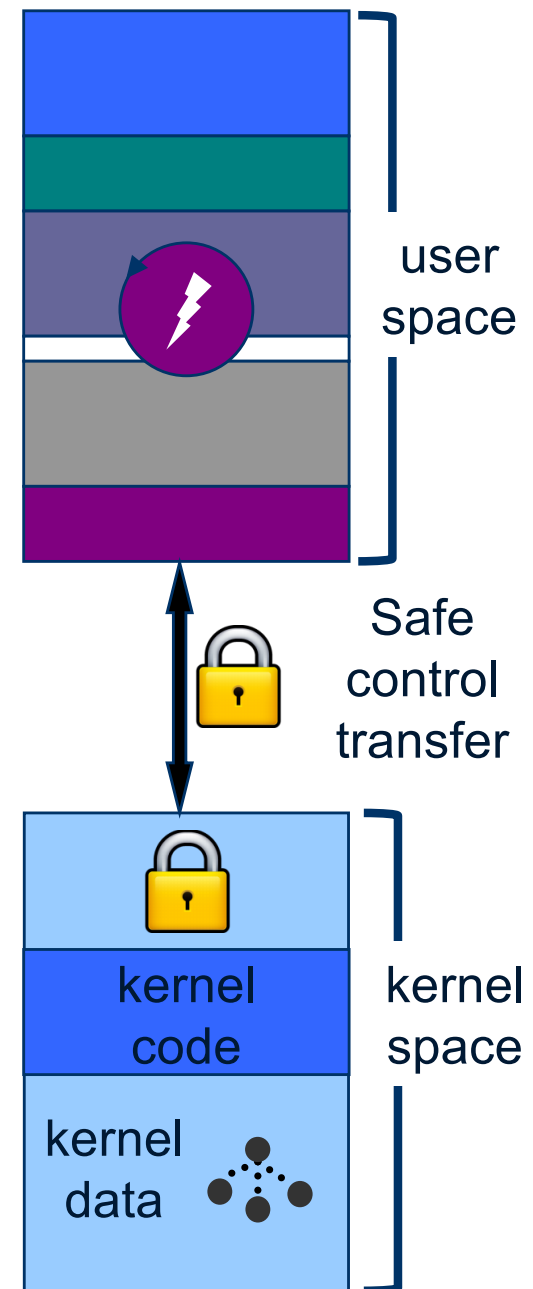
# “Virtualization” in an OS



**What are the architectural foundations that enable all this magic?**

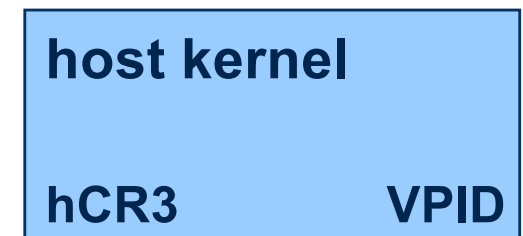
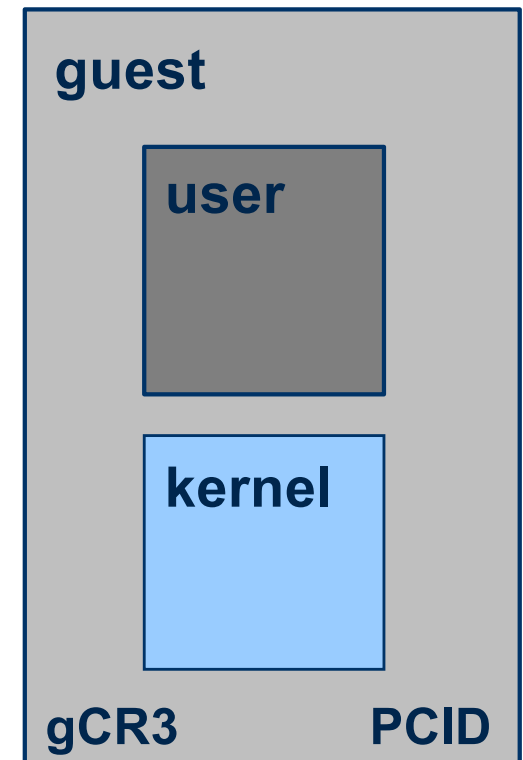
# OS: architectural foundations

- **Protected mode bit:** user vs. kernel
- **Maps** for virtual memory
  - VPN→PFN
- **Control registers** for kernel
  - CR3: current memory map base (a PFN)
  - PCID: process context ID (for TLB tags)
  - Current thread and its kernel stack SP
- **Events** transfer control to kernel handler
  - **Trap:** syscall instruction
  - **Fault**, e.g., page fault
  - **Interrupt**, e.g., I/O or clock
- **Event vectors.** Kernel configures event handling.



# VM: architectural foundations

- **Protected modes.** *Double up!* host **and** guest
- **Maps.** *Double up!* Extended Page Tables (EPT)
  - $gVPN \rightarrow gPFN$ ;  $gPFN \rightarrow hPFN$
- **Control registers.** *Double up!*
  - CR3: map base for host **and** guest
  - “Virtual processor” VPID **and** PCID
- **Event vectors.** *Double up!* Define event handlers for host kernel (hypervisor) **and** guest.
- **Events.** Configure for host vs. guest
  - **Trap:** add hypercalls for “paravirtualized” OS
  - **Fault:** configure **interceptions** per-VM
  - **Interrupt:** delivery in host vs. guest

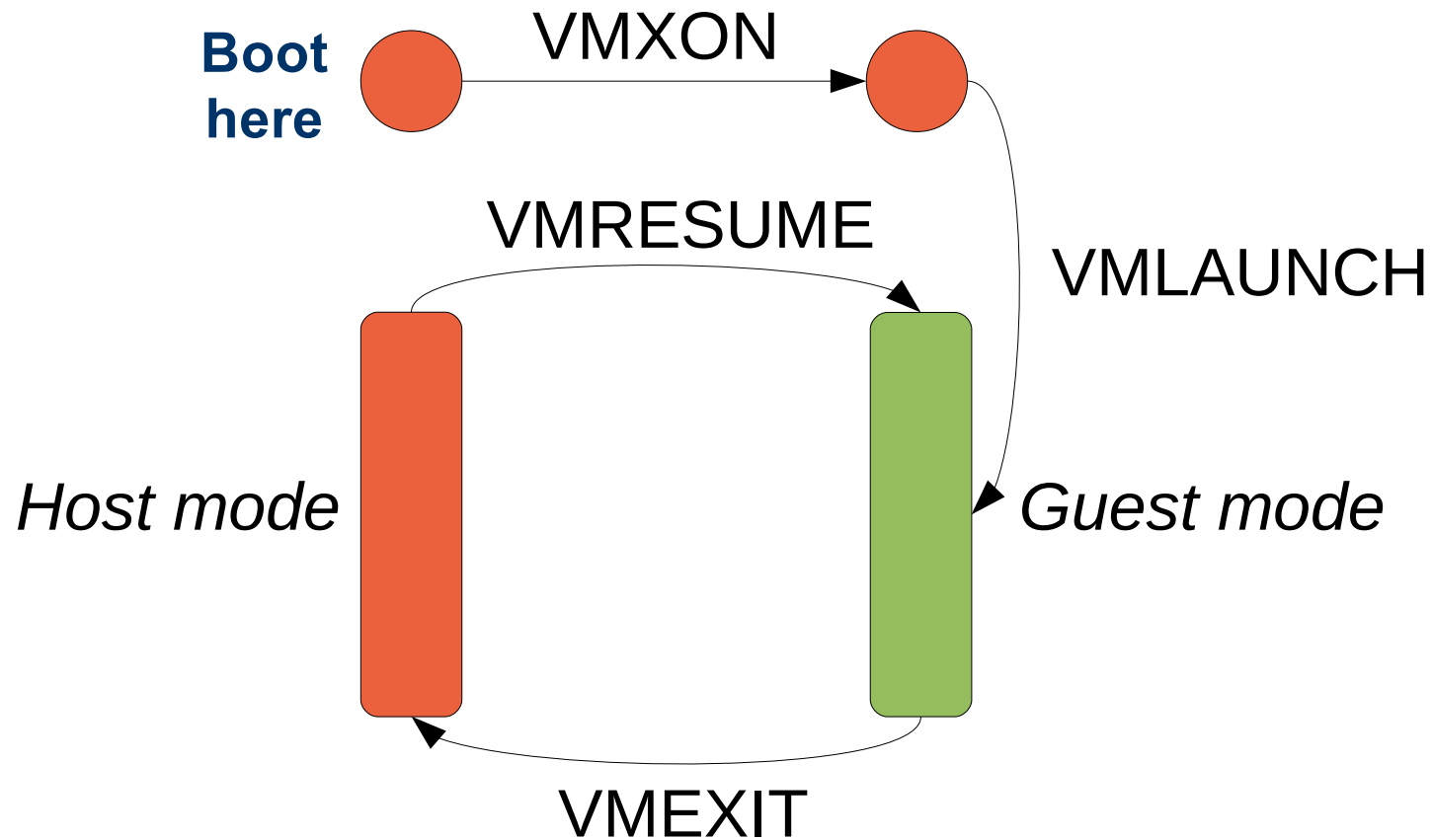


# CPU protection modes (IA64-VMX)

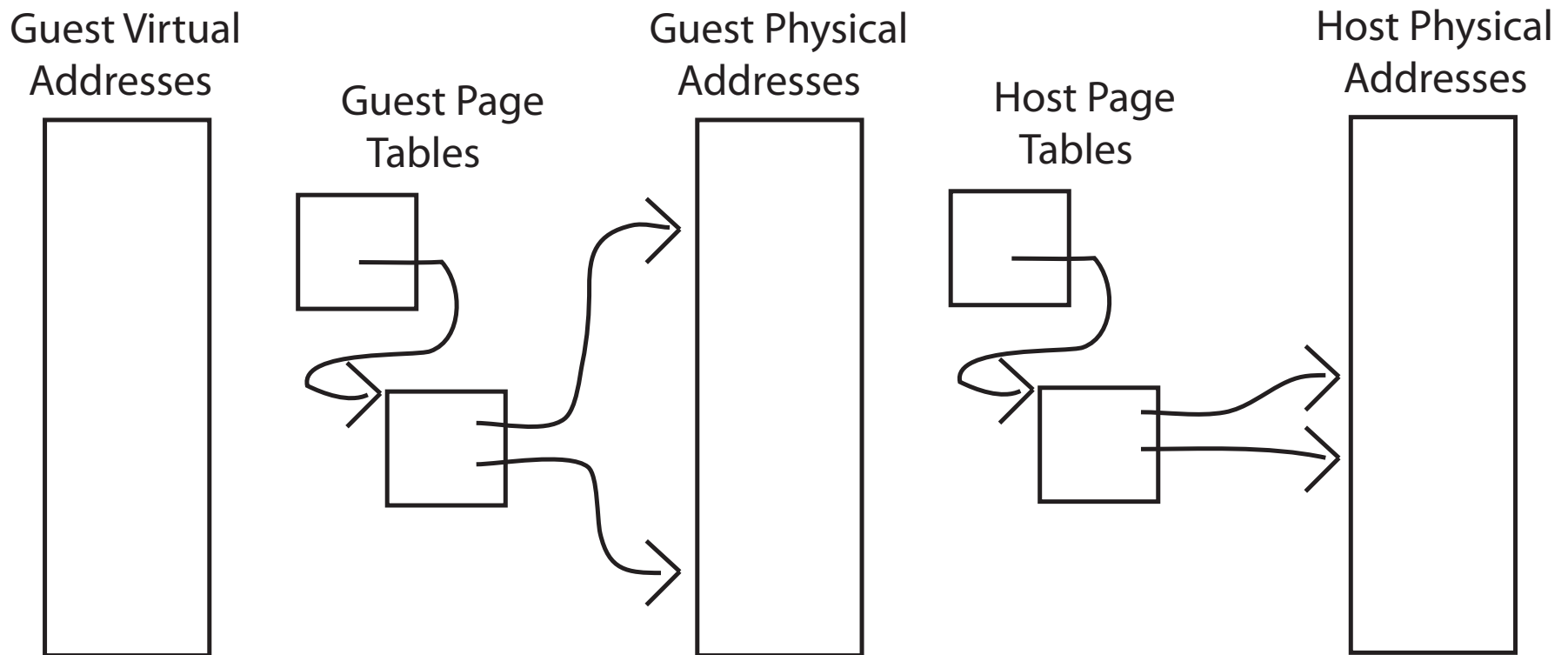
	Guest (VMX Non-Root)	Host (VMX Root)
User (Ring 3)	User process in a VM “It’s all the same.”	User process that is not within any guest VM. E.g., vanilla, or operator tools, device emulation.
Kernel (Ring 0)	Guest OS kernel: Controls a VM, might not distinguish it from a physical machine.	Hypervisor (only one)

**Double up!** Now we have **two** mode bits → four combinations.  
Many guests (each with VMCS), with many processes (each with PCID).

# VMX mode switches: simplified



# Virtual Machines and Virtual Memory

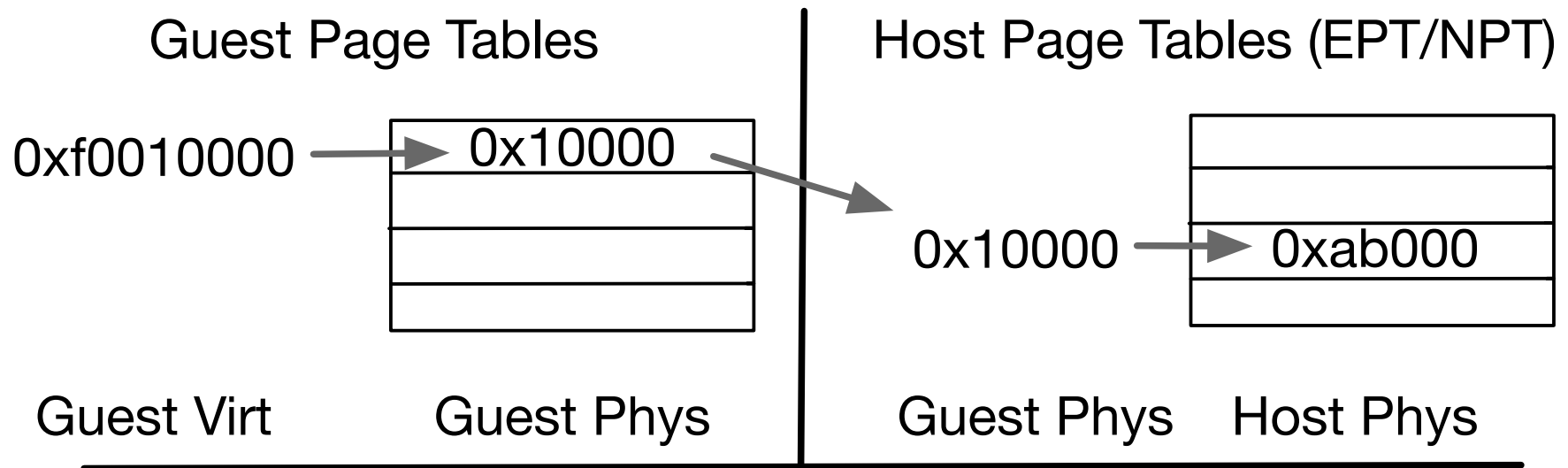


$gVA \rightarrow gPA$

$gPA \rightarrow hPA$

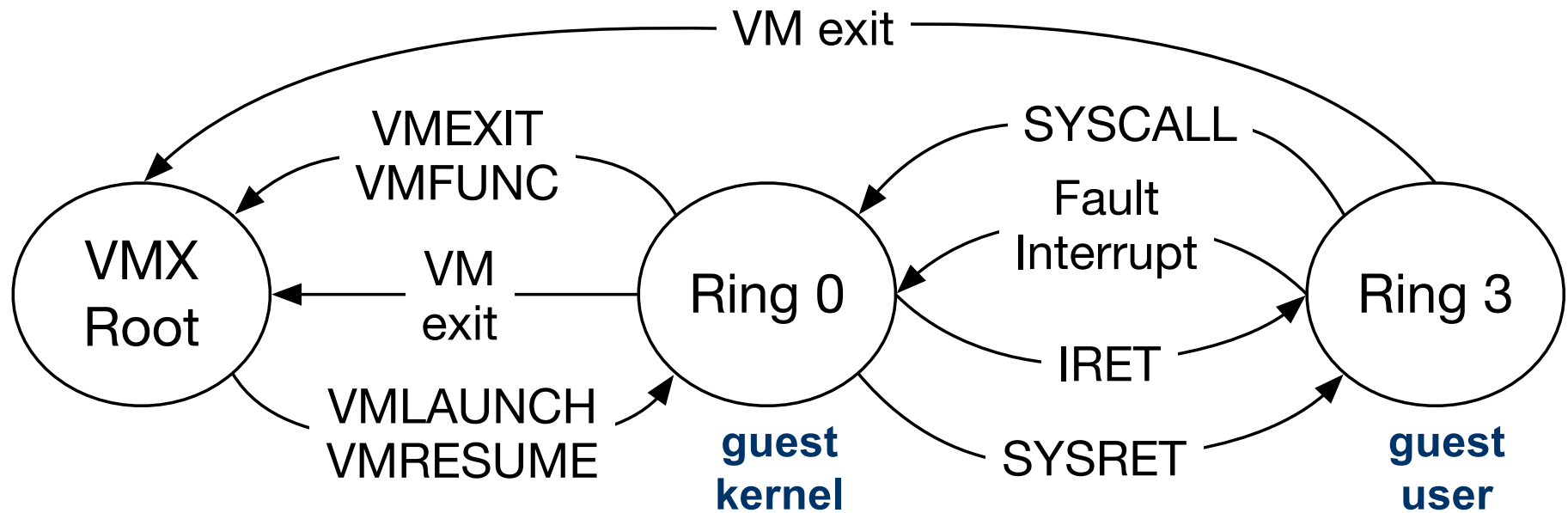
Intel VT-x: **Extended Page Tables (EPT)**

# EPT: simplified view



AMD calls the extra VM-enabling structure a **Nested Page Table (NPT)**. But it's the same thing.

# VMX mode switches



**Figure 19:** Modern privilege switching methods in the 64-bit Intel architecture.

Intel's Virtual Machine Extensions (VMX) introduce new VMX instructions and a **Virtual Machine Control Structure (VMCS)** to configure mode switches per-VM.



# What causes a VM exit?

- Page fault on gPA (VM RAM)
- VMCALL trap from guest-kernel mode (hypercall)
- VMCS-configured interceptions
  - E.g., emulated instructions
- Configured interrupts
  - E.g., clock
  - Non-virtualizable devices

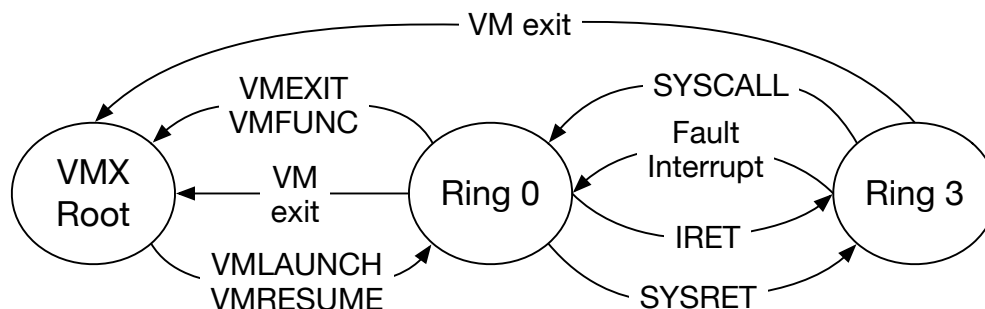
## 2.8.3 VMX Privilege Level Switching

Intel systems that take advantage of the hardware virtualization support to run multiple operating systems at the same time use a hypervisor that manages the VMs. The hypervisor creates a *Virtual Machine Control Structure* (VMCS) for each operating system instance that it wishes to run, and uses the `VMENTER` instruction to assign a logical processor to the VM.

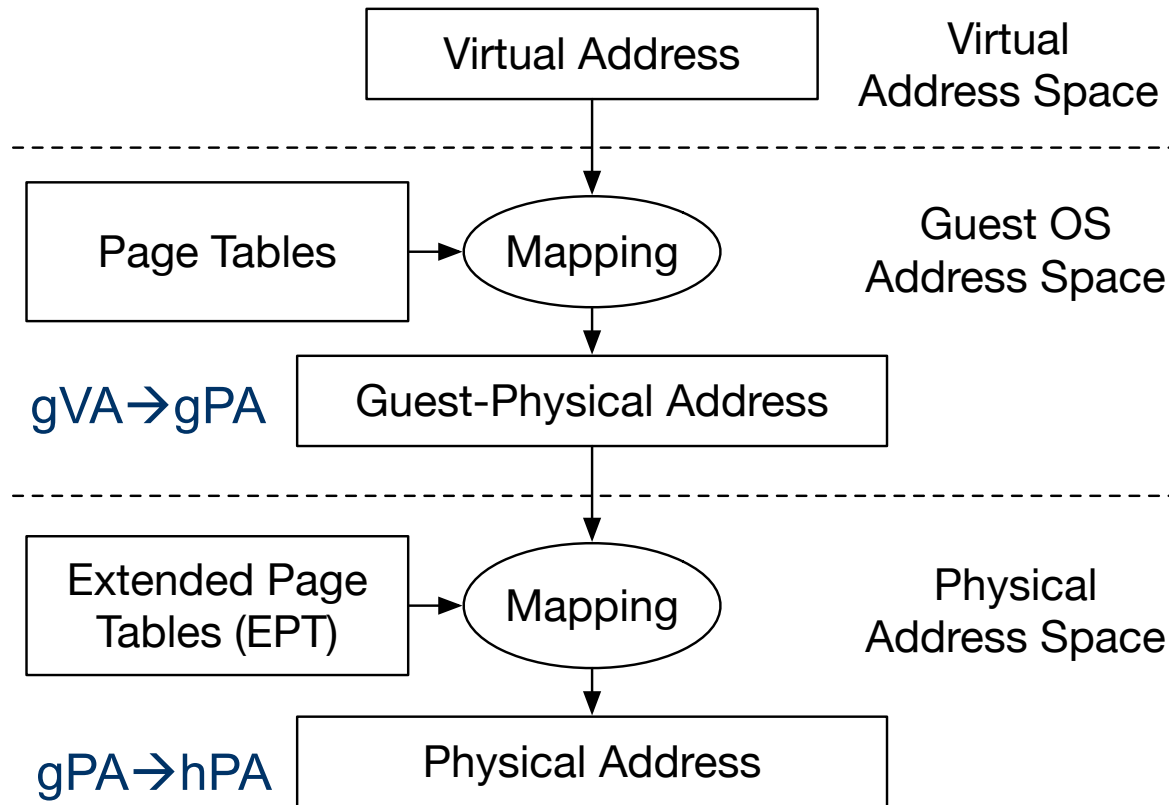
When a logical processor encounters a fault that must be handled by the hypervisor, the logical processor performs a VM exit. For example, if the address translation process encounters an EPT entry with the P flag set to 0, the CPU performs a VM exit, and the hypervisor has an opportunity to bring the page into RAM.

The VMCS shows a great application of the encapsulation principle [130], which is generally used in high-level software, to computer architecture. The Intel architecture specifies that each VMCS resides in DRAM and is 4 KB in size. However, the architecture does not specify the VMCS format, and instead requires the hypervisor to interact with the VMCS via CPU instructions such as `VMREAD` and `VMWRITE`.

This approach allows Intel to add VMX features that require VMCS format changes, without the burden of having to maintain backwards compatibility. This is no small feat, given that huge amounts of complexity in the Intel architecture were introduced due to compatibility requirements.



# Extended Page Tables: EPT (VMX)



**$gPA \rightarrow hPA$  mappings typically use 2MB superpages to save one walk level.**

## 2.5.2 Address Translation and Virtualization

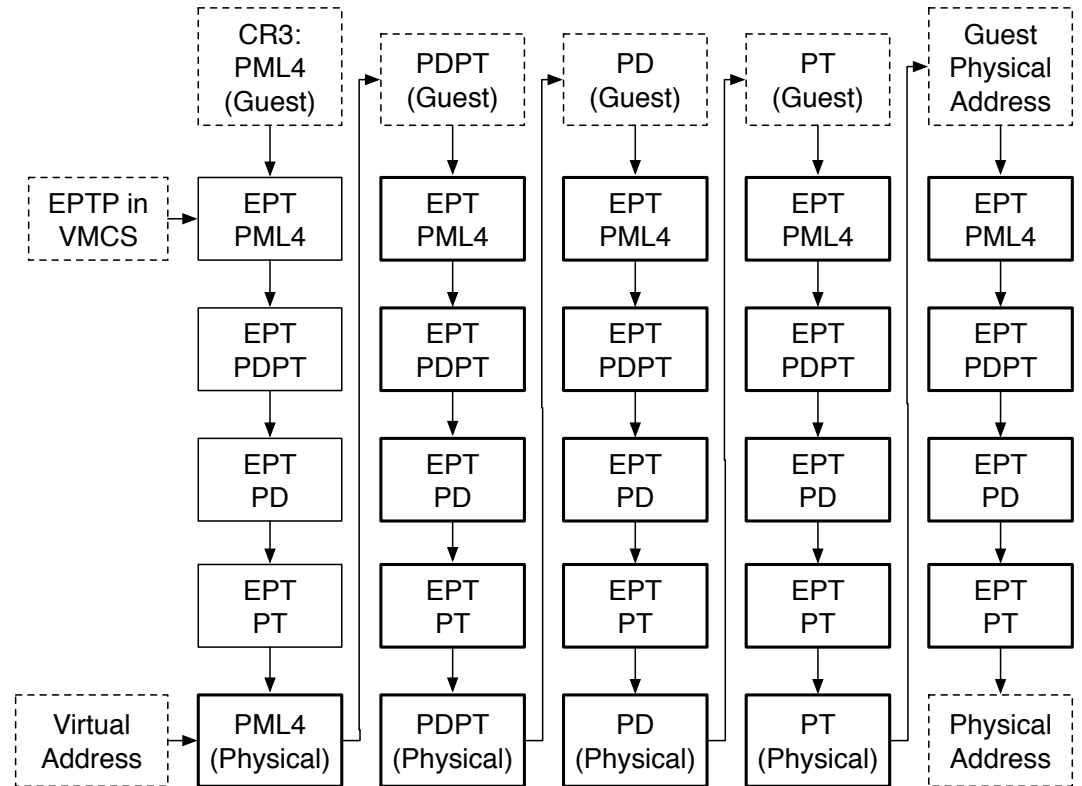
Computers that take advantage of hardware virtualization use a hypervisor to run multiple operating systems at the same time. This creates some tension, because each operating system was written under the assumption that it owns the entire computer's DRAM. The tension is solved by a second layer of address translation, illustrated in Figure 14.

When a hypervisor is active, the page tables set up by an operating system map between virtual addresses and *guest-physical addresses* in a *guest-physical address space*. The hypervisor multiplexes the computer's DRAM between the operating systems' guest-physical address spaces via the second layer of address translations, which uses *extended page tables* (EPT) to map guest-physical addresses to physical addresses.

The EPT uses the same data structure as the page tables, so the process of translating guest-physical addresses to physical addresses follows the same steps as IA-32e address translation. The main difference is that the physical address of the data structure's root node is stored in the extended page table pointer (EPTP) field in the *Virtual Machine Control Structure* (VMCS) for the guest OS. Figure 15 illustrates the address translation process in the presence of hardware virtualization.

# TLB miss → “2D page walk”

- Guest page maps are populated with gPFNs.
- Any such gPFN must translate to an hPFN to access memory.
- Thus a 4-level g-map fetch requires four rounds of gPFN→hPFN translation.
- ...through 4-level h-map.
- → at worst 20-24 map references on a TLB miss.
- Now CPUs reduce cost with new TLB-like caches, tagged by VPID and PCID.



**Figure 15:** Address translation when hardware virtualization is enabled. The kernel-managed page tables contain guest-physical addresses, so each level in the kernel’s page table requires a full walk of the hypervisor’s extended page table (EPT). A translation requires up to 20 memory accesses (the bold boxes), assuming the physical address of the kernel’s PML4 is cached.

# Type 2 example: HOSS

- In a type 2, a VM “looks like a process/VAS” to the host kernel (hypervisor).
- HOSS extends JOS → VMs.
  - Projects?
- Hypervisor talks to a user process (qemu) to emulate PC-like devices.
- Linux KVM uses this approach as well.

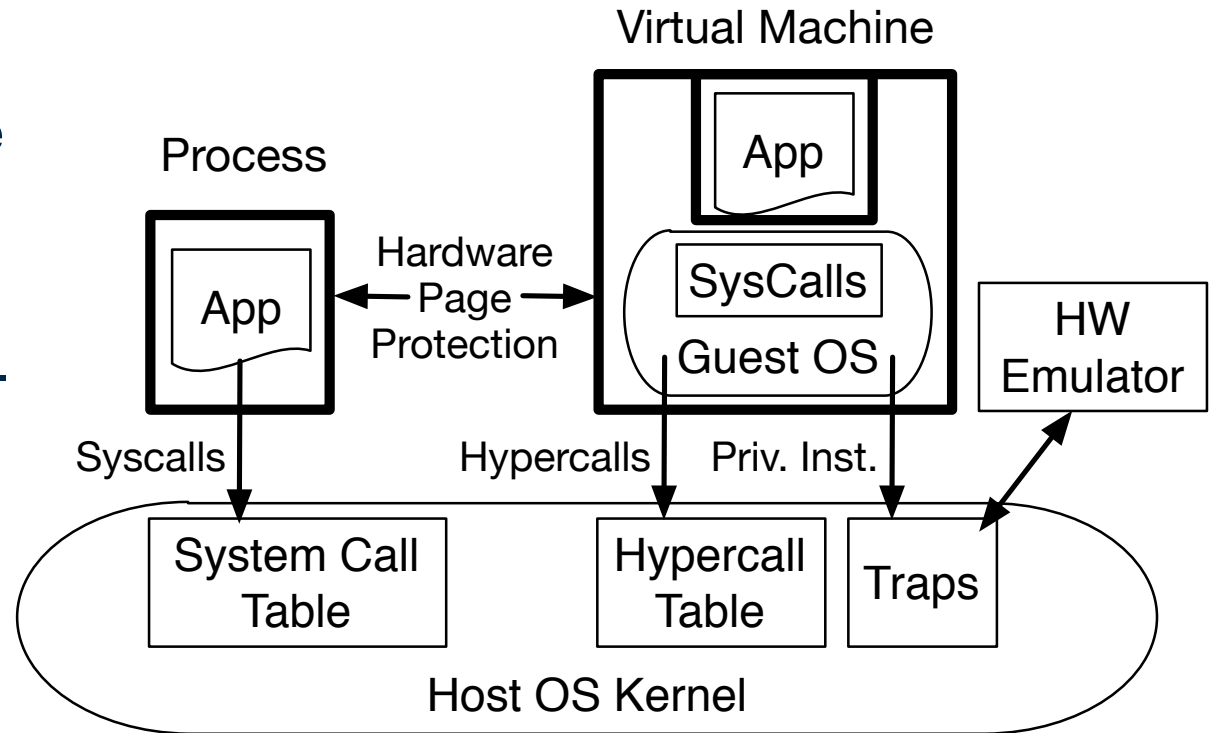


Figure 1: Comparison of an application in a process (left) to a virtual machine (right). Both execute in an isolated address space, implemented with hardware page tables. The primary difference is the ABI—system calls versus hypercalls and emulated hardware. A host OS kernel can execute a mixture of processes and VMs.

# Reflections on VMs

## Why are we doing this?

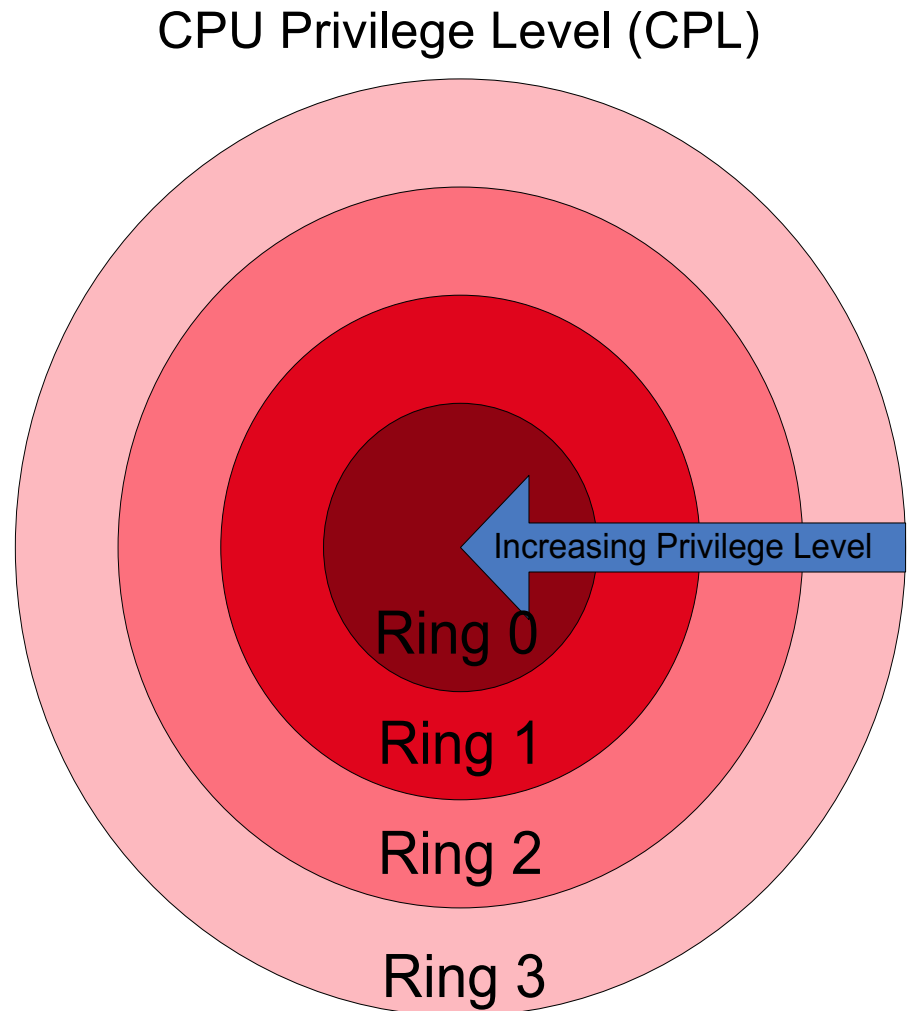
- Custom environments, fault containment/security, virtual infrastructure services with performance isolation.
- OS refactoring? Smaller TCB? Microkernels done right?
  - Focus on resources, separate control and data paths, avoid “liability inversion” (e.g., Mach pagers).
  - Run apps in guest mode and expose hardware control: Dune 2012, high-speed I/O with Arrakis 2014.
- How long before we got VMs right? Software approaches struggled without good hardware support.

### **Are Virtual Machine Monitors Microkernels Done Right?**

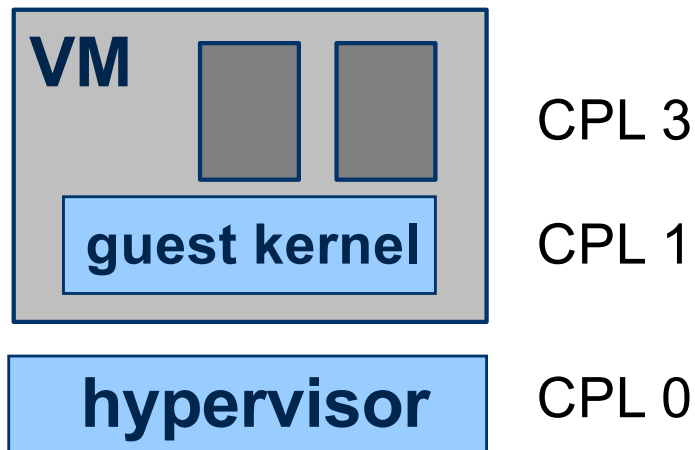
Steven Hand, Andrew Warfield, + Fraser, Kotsovinos, Magenheimer, HotOS 2005.

# IA Protection Rings (CPL)

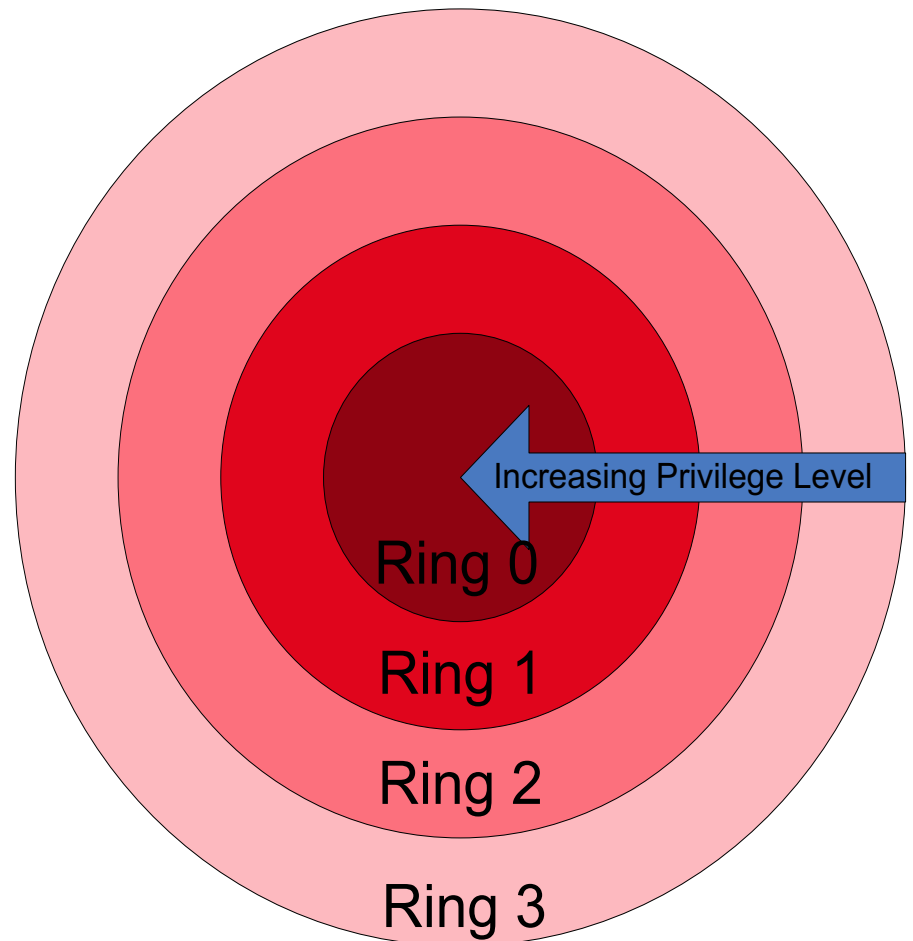
- Actually, IA has four protection levels, not two (kernel/user).
- **IA/X86 rings (CPL)**
  - **Ring 0** – “Kernel mode” (most privileged)
  - **Ring 3** – “User mode”
  - **Ring 1 & 2** – Other
- Linux only uses 0 and 3.
  - “Kernel vs. user mode”
- Pre-VT VMMs modified to lift the guest OS kernel to Ring 1: reserve Ring 0 for hypervisor.



# Why aren't (IA) rings good enough?



???



# A short list of pre-VT problems

Early IA hypervisors (VMware, Xen) had to emulate various machine behaviors and generally bend over backwards.

- IA32 page protection does not distinguish CPL 0-2.
  - Segment-grained memory protection only.
- Ring aliasing: some IA instructions expose CPL to guest!
  - Or fail silently...
- Syscalls don't work properly and require emulation.
  - IA sysenter always transitions to CPL 0. (D'oh!)
  - IA sysexit faults if the core is not in CPL 0.
- Interrupts don't work properly and require emulation.
  - Interrupt disable/enable reserved to CPL0.



# Early approaches to IA VMMs

To implement virtualization on IA in the pre-VT days, you have to modify the guest OS kernel.

- To run a proprietary OS (i.e., Windows) you had to do it on the sly.
  - VMware: software **binary translation** (code morphing—patented)
- If it's an open-source OS (e.g., Linux) you can do it out in the open.
  - Xen: **paravirtualization**
  - Basically just a change to the HAL/PAL: a VM is just another machine type to support (<2% of the OS code).
- Of course you can always do full emulation! (pre-KVM qemu)
- **All of this is washed away by VT.**
  - (Of course, hardware-based VMs were also done correctly on IBM machines a lonnnng time ago.)

# Notes on implementing VMs today

- CPUs after 2007 support protected mode(s) for hypervisors (E.g., Intel VTx).
- When the hypervisor initializes a **VM context (e.g., VMCS)**, it selects some set of event types to **intercept**, and registers handlers for them.
- Configured interceptions transfer control to a registered **hypervisor handler** routine. For example, a guest OS kernel accessing device registers may cause the physical machine to invoke the hypervisor to intervene.
- Architecture adds another level of indirection in the MMU page mappings (Intel's **Extended Page Tables**). A hypervisor uses it to specify and restrict what parts of physical memory are visible to each guest VM.
- A guest can map to or address a physical memory frame or command device DMA I/O to/from a physical frame if and only if the hypervisor permits it.
- If any guest VM tries to do anything weird, then the hypervisor regains control and can see or do anything to any part of the physical or virtual machine state before (optionally) restarting the guest VM.

# According to Dune

## 2.1 The Intel VT-x Extension

In order to improve virtualization performance and simplify VMM implementation, Intel has developed VT-x [37], a virtualization extension to the x86 ISA. AMD also provides a similar extension with a different hardware interface called SVM [3].

The simplest method of adapting hardware to support virtualization is to introduce a mechanism for trapping each instruction that accesses privileged state so that emulation can be performed by a VMM. VT-x embraces a more sophisticated approach, inspired by IBM's interpretive execution architecture [31], where as many instructions as possible, including most that access privileged state, are executed directly in hardware without any intervention from the VMM. ...The motivation for this approach is to increase performance, as traps can be a significant source of overhead.

VT-x adopts a design where the CPU is split into two operating modes: *VMX root* and *VMX non-root* mode. VMX root mode is generally used to run the VMM and does not change CPU behavior, except to enable access to new instructions for managing VT-x. VMX non-root mode, on the other hand, restricts CPU behavior and is intended for running virtualized guest OSes.

Transitions between VMX modes are managed by hardware. When the VMM executes the *VMLAUNCH* or *VMRESUME* instruction, hardware performs a *VM entry*; placing the CPU in VMX non-root mode and executing the guest. Then, when action is required from the VMM, hardware performs a *VM exit*, placing the CPU back in VMX root mode and jumping to a VMM entry point. Hardware automatically saves and restores most architectural state during both types of transitions. This is accomplished by using buffers in a memory resident data structure called the VM control structure (VMCS).

In addition to storing architectural state, the VMCS contains a myriad of configuration parameters that allow the VMM to control execution and specify which type of events should generate VM exits. This gives the VMM considerable flexibility in determining which hardware is exposed to the guest. For example, a VMM could configure the VMCS so that the *HLT* instruction causes a VM exit or it could allow the guest to halt the CPU. However, some hardware interfaces, such as the interrupt descriptor table (IDT) and privilege modes, are exposed implicitly in VMX non-root mode and never generate VM exits when accessed. Moreover, a guest can manually request a VM exit by using the *VMCALL* instruction.

Virtual memory is perhaps the most difficult hardware feature for a VMM to expose safely. A straw man solution would be to configure the VMCS so that the guest has access to the page table root register, *%CR3*. However, this would place complete trust in the guest because it would be possible for it to configure the page table to access any physical memory address, including memory that belongs to the VMM. Fortunately, VT-x includes a dedicated hardware mechanism, called the *extended page table* (EPT), that can enforce memory isolation on guests with direct access to virtual memory. It works by applying a second, underlying, layer of address translation that can only be configured by the VMM. AMD's SVM includes a similar mechanism to the EPT, referred to as a nested page table (NPT).

From **Dune: Safe User-level Access to Privileged CPU Features**, Belay et al., (Stanford), OSDI, October, 2012