Lecture 18: Virtualization

CS343 – Operating Systems
Branden Ghena – Fall 2022

Some slides borrowed from:
Jaswinder Pal Singh (Princeton), Harsha V. Madhyastha (Michigan), and UC Berkeley CS162

Administrivia

- PagingLab due today!

- Midterm exam 2
  - Wednesday, December 7th from 12:00-2:00pm in the lecture hall

  - Expect a similar format to the previous exam
    - Covers Device I/O through Virtualization
      - Lectures 10-16 and 18
      - Does NOT cover scheduling or concurrency. Does NOT cover Embedded OS

  - Practice exam will be posted soon
  - Practice problems during lecture on Thursday

Today's Goals

- Explore notion of a "virtual machine" and how to virtualize computers.

- Understand challenges and tradeoffs for several approaches
  - Emulation
 - Hypervisors
  - Containers

Outline

- **Virtualization**

- Approaches
  - Emulation

  - Hypervisors

  - Containers

Virtualization

- Virtual (fake) versions of real resources are often provided to users
  - Memory – virtual memory
  - CPU – processes and scheduler
  - Disk – files

- OS provides these abstractions to simplify applications
  - And provide security!

Virtual Machines (VMs)

- What about virtualizing the whole computer?
  - Provide interfaces that look like a normal computer
  - But actually interact with software that manages and multiplexes access

- Run an entire OS within an OS
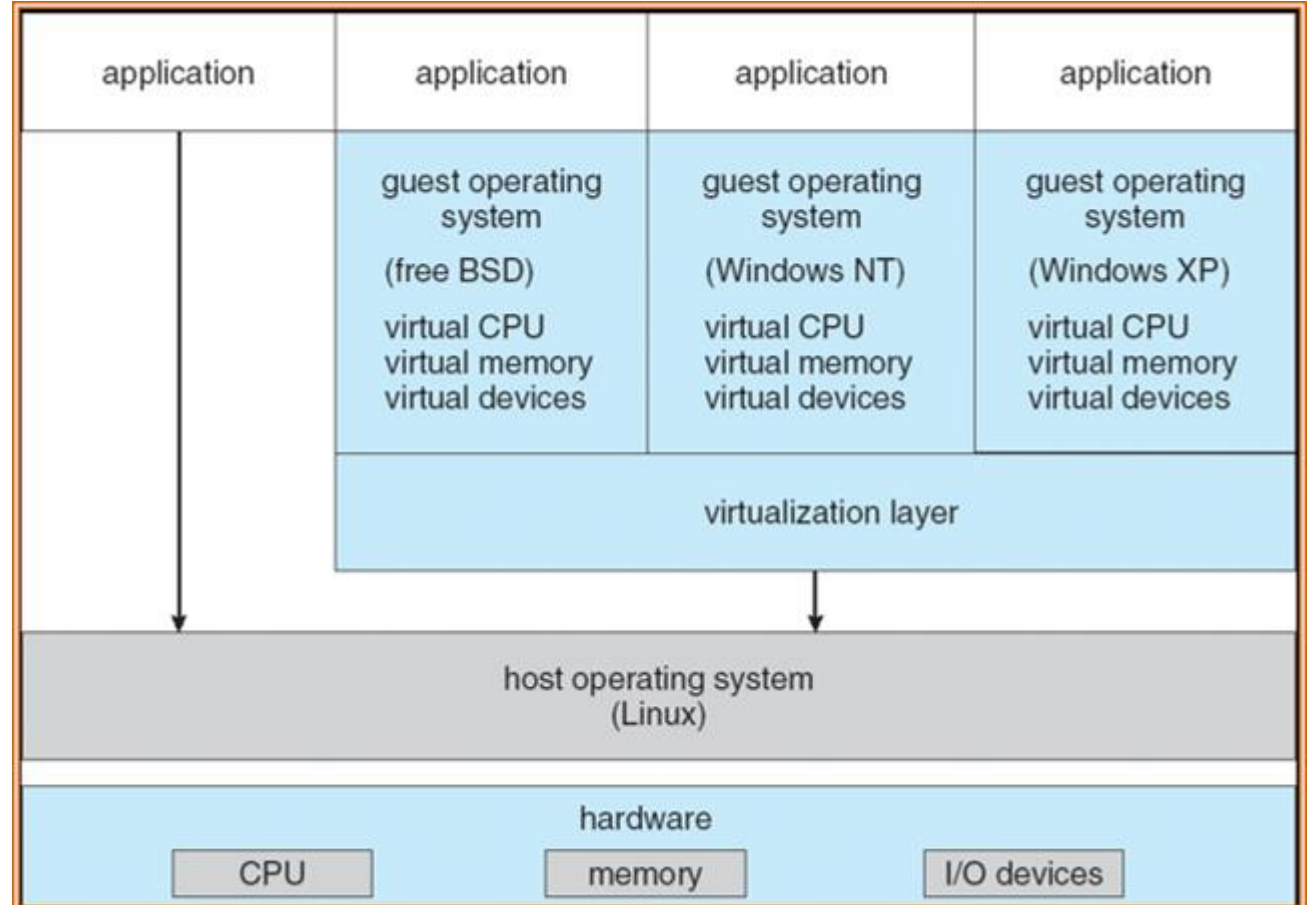
Original motivation: support more applications

- 1960s IBM mainframes had many different OSes
  - Some applications only written for certain OSes though

- Virtualization allowed multiple OSes to run on a single mainframe
  - Which let one powerful computer serve varied needs of many people

- Still applies today to some degree
  - Windows + Ubuntu VM
    - Want PowerPoint and also terminal environment (vim/make/gcc)

  - MacOS + Windows VM
    - Various Windows-only programs
    - Often dual boot rather than VM

Modern motivation: package and isolate applications

- High-performance applications aren't really stand-alone
  - Assumptions about OS
  - Assumptions about libraries and services
  - Multiple processes working together

- A virtual machine is a method to encapsulate "entire stack"
  - Even down to expectations of hardware

- Cloud computing platforms run many applications together
  - Need isolation from each other in a strongly controllable way
    - Exactly 2 GB of RAM go to this
    - Exactly two processor cores go to that

Virtualization approaches

1. Simulate everything in the computer completely
   - Emulation

2. Simulate parts of the computer, but not all of it (actually use CPU)
   - Hypervisor

3. Simulate the operating system (software environment)
   - Containers

Outline

- Virtualization

- **Approaches**
  - **Emulation**

  - Hypervisors

  - Containers

Software emulation

- User software emulates the behavior of every single instruction
  - Data structures for Processor, Memory, I/O, etc.
  - Code for Instruction Cycle:
    - Fetch next instruction
    - Decode
    - Perform operation
    - Update state

- Example: Gameboy emulator
  - Simulates every behavior as-if it were actually Gameboy hardware

Emulation example: QEMU

- We have been using QEMU for lab to simulate an x86-64 computer
  - 2 CPU cores
  - 2 GB of RAM
  - VirtIO GPU
  - PS/2 mouse and keyboard
  - 2 PCI IDE interfaces with hard disk and CD-ROM support
    - nautilus.iso connected to CD-ROM
  - Serial and parallel ports
    - stdio connected to serial port
    - file parport.out connected to parallel port
  - Other stuff
    - Floppy disk
    - PCI and ISA network adapters
    - Intel HD Audio Controller and HDA codec
    - PCI UHCI, OHCI, EHCI or XHCI USB controller and a virtual USB-1.1 hub
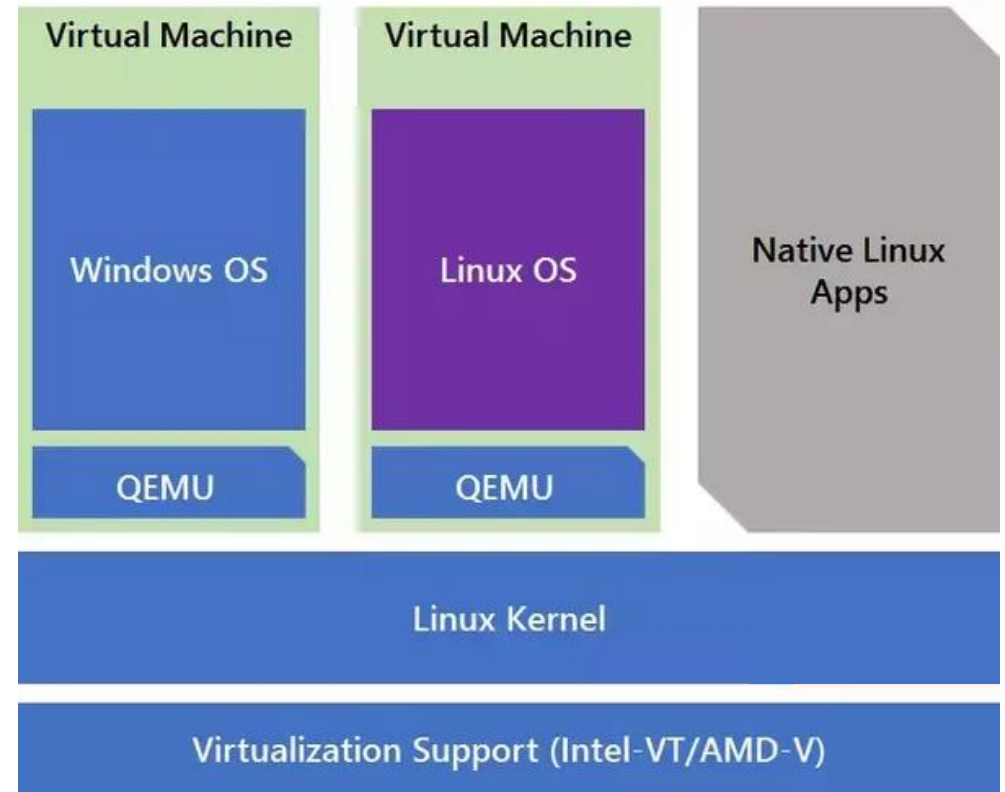
Emulation tradeoffs

- Upsides
  - Any hardware you want
 - Entirely in userspace

- Downside
  - Complicated to get accurate
  - Software runs slower than the hardware would

  (But modern hardware might run software faster than old hardware)

Simple emulators: interpreted languages

- Create a simple environment for code to execute within

- Interpret code instructions (bytecode or lines of code) and perform actions
  - Example: fakes a machine that executes Java bytecode

- Still ties in to many parts of the real machine
  - Filesystem
  - Devices
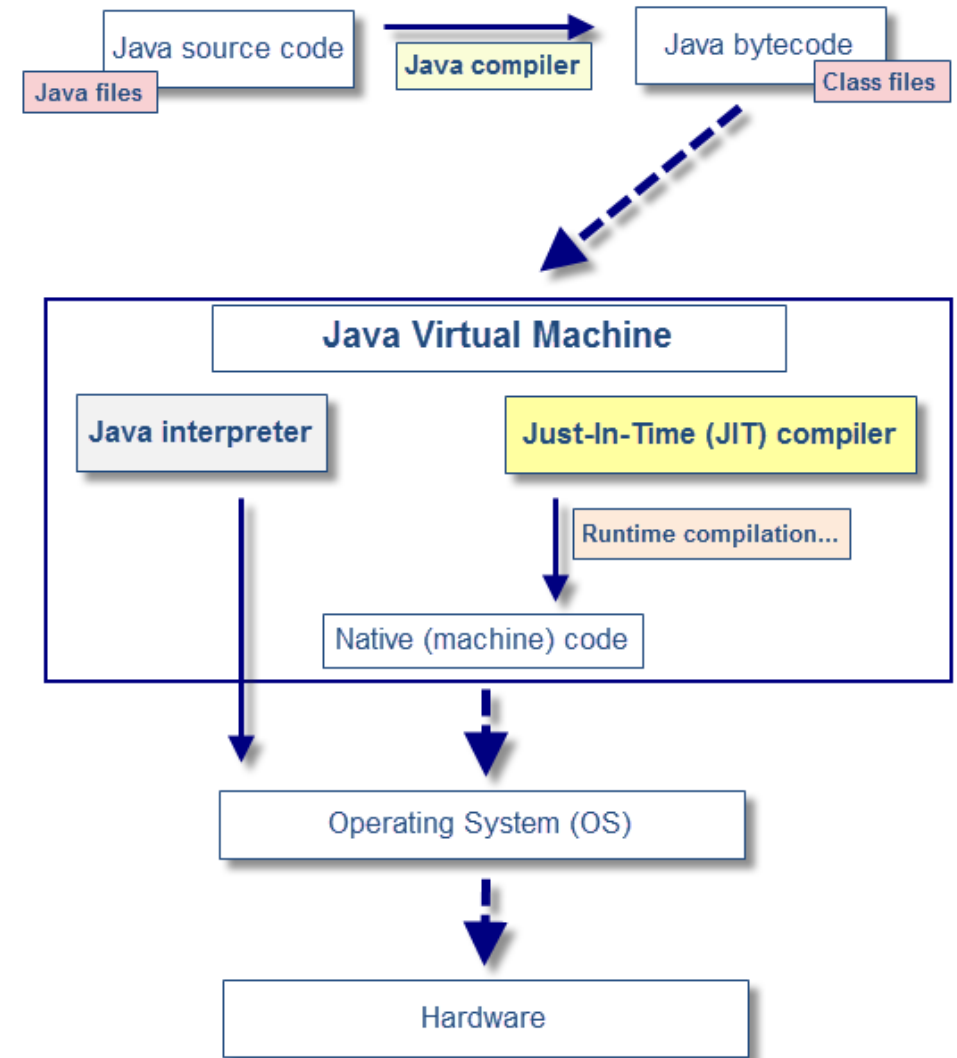
Not-quite-emulation: binary translation

- MacOS on ARM (M1)
  - Uses ARM processor with ARM instruction set
  - Old programs were compiled for x86-64 instruction set

- Solution: translate assembly instructions
  - Can be translated in advance
  - Or just-in-time (JIT)
  - Works fine for applications that are I/O bound

- Simulates a different CPU, but leaves the remainder of the computer the same

Rosetta 2

Break + Open Question

- What are the best use cases for full hardware emulation?

Break + Open Question

- What are the best use cases for full hardware emulation?

    - Non-standard, slow machines (example: old gaming consoles)
        - Where people do not have access to the original hardware
        - The original hardware was slower than modern hardware
        - And there are existing application binaries that they want to run

    - Application-level simulation/testing (example: Nautilus!)
        - Still hopefully for the above
        - Entirely controlled environment

Outline

- Virtualization

- **Approaches**
  - Emulation

  - **Hypervisors**

  - Containers

How do we speed up virtual machines?

"Efficiency … demands that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor, with no software intervention…"

—Popek and Goldberg, 1974

- Need to use some parts of the computer for real while simulating other parts

Virtual Machine Monitor (VMM)

- Also known as hypervisors
    - OS kernel is the system "supervisor" and manages the computer
    - Hypervisor manages supervisors

- Creates the illusion that the OS has full control over the hardware
    - And even gives real (limited) access to hardware whenever possible
    - But may actually be sharing full computer resources among several OSes

- Probably what you had in mind as virtual machines
    - VirtualBox, VMWare, Parallels

Hypervisor layering: directly on hardware
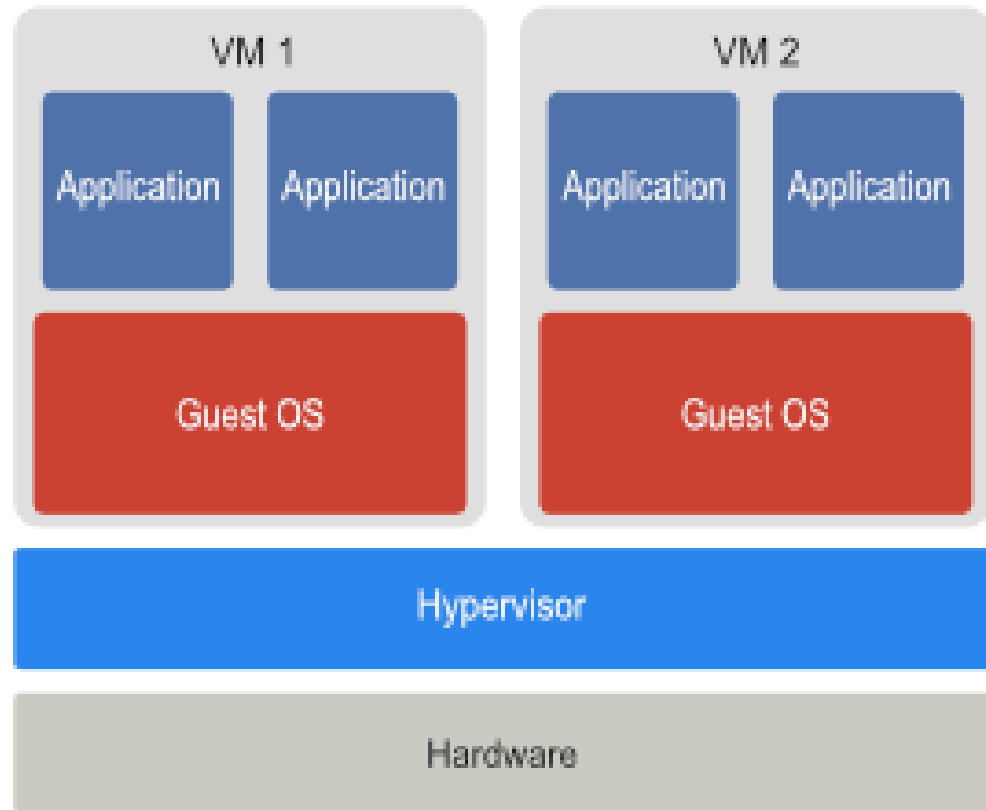
Type 1
"Bare Metal" Hypervisor

- Hypervisor manages hardware directly

- Guest OSes run on top of it
  - "Guest OS" as in it isn't actually in charge of the computer

# Hypervisor layering: on top of Host OS

- Normal operating system runs on hardware
  - Known as "Host OS"

- Hypervisor runs on top of host and coordinates with it to enable interactions with hardware
  - Some coordination may be within the kernel itself
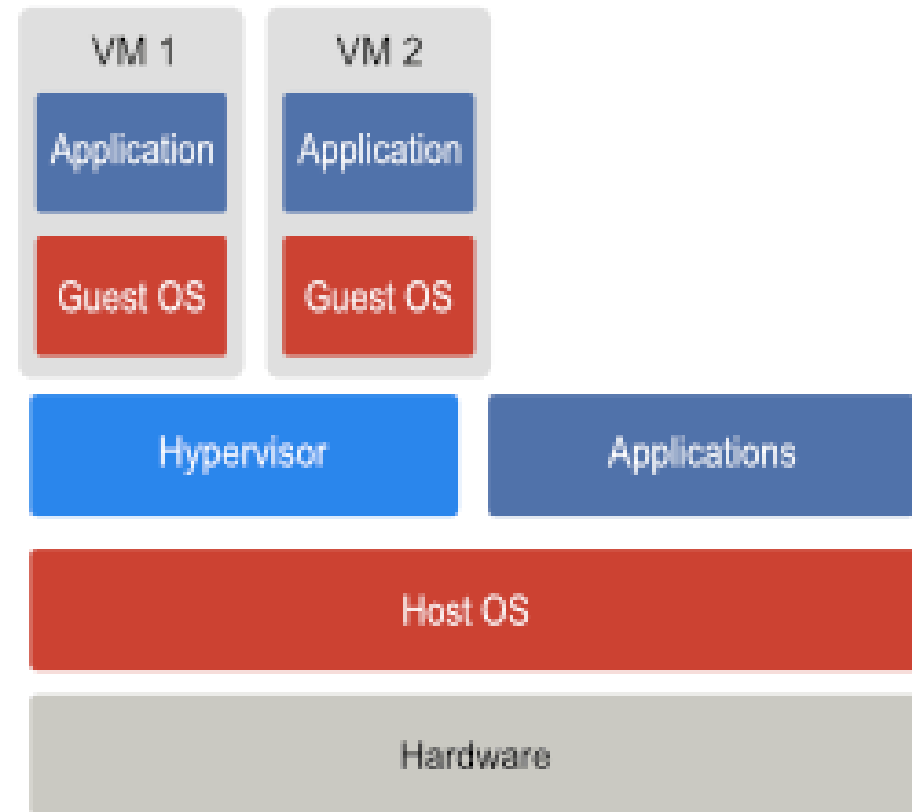
Type 2
Hosted Hypervisor

Hypervisor layering: comparison

Hypervisor Types

Type 1
"Bare Metal" Hypervisor

Type 2
Hosted Hypervisor
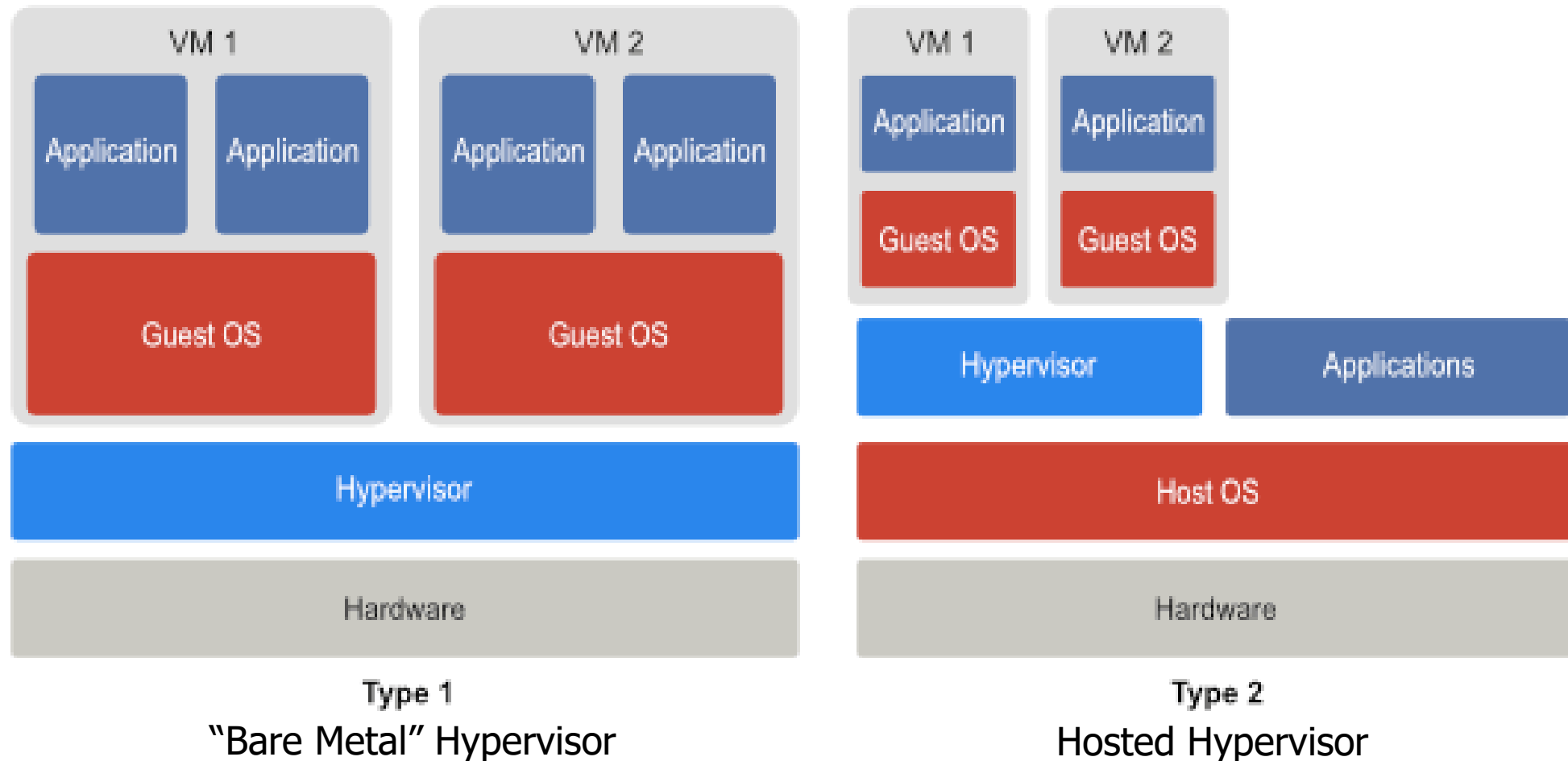
Abstraction choices for hypervisor

- Fully virtualizing hypervisor
  - Virtual machine looks exactly like a physical machine
    - Though not necessarily the same machine it's running on
  - Guest OS does not need to be modified in any way
  - Guest may not even be aware it's running virtually

- Para-virtualizing hypervisor
  - Guest OS has extensions to cooperate with hypervisor
  - Sacrifice transparency for better performance
    - Same abstraction-breaking ideal from previous lectures
  - Might include an API to interact with hypervisor
  - Guest OS likely can skip some stuff the hypervisor handles instead

Arbitrary combinations of these are possible

	Bare Metal Hypervisor	Hosted Hypervisor
Fully Virtualized	VMWare ESXi, Microsoft Hyper-V	VMware Workstation, VMware Fusion, Parallels, VirtualBox
Para Virtualized	Xen	User Mode Linux

Arbitrary combinations of these are possible

"Normal" VMs

	Bare Metal Hypervisor	Hosted Hypervisor
Fully Virtualized	VMWare ESXi, Microsoft Hyper-V	VMware Workstation, VMware Fusion, Parallels, VirtualBox
Para Virtualized	Xen	User Mode Linux

Arbitrary combinations of these are possible

Cloud/Server stuff

	Bare Metal Hypervisor	Hosted Hypervisor
Fully Virtualized	VMWare ESXi, Microsoft Hyper-V	VMware Workstation, VMware Fusion, Parallels, VirtualBox
Para Virtualized	Xen	User Mode Linux

Arbitrary combinations of these are possible

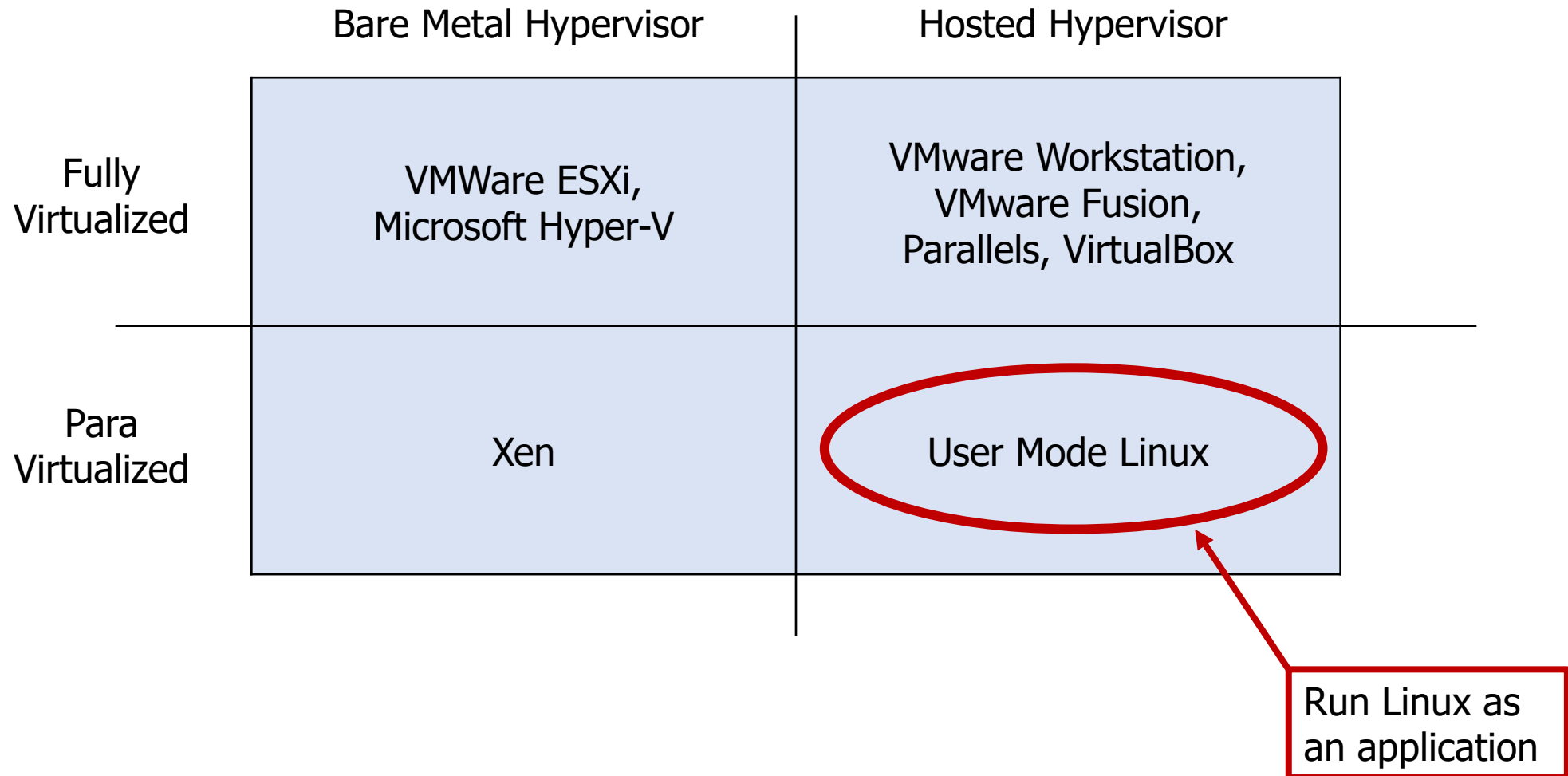|	Bare Metal Hypervisor	Hosted Hypervisor
Fully Virtualized	VMWare ESXi, Microsoft Hyper-V	VMware Workstation, VMware Fusion, Parallels, VirtualBox
Para Virtualized	Xen	User Mode Linux

Run Linux as an application
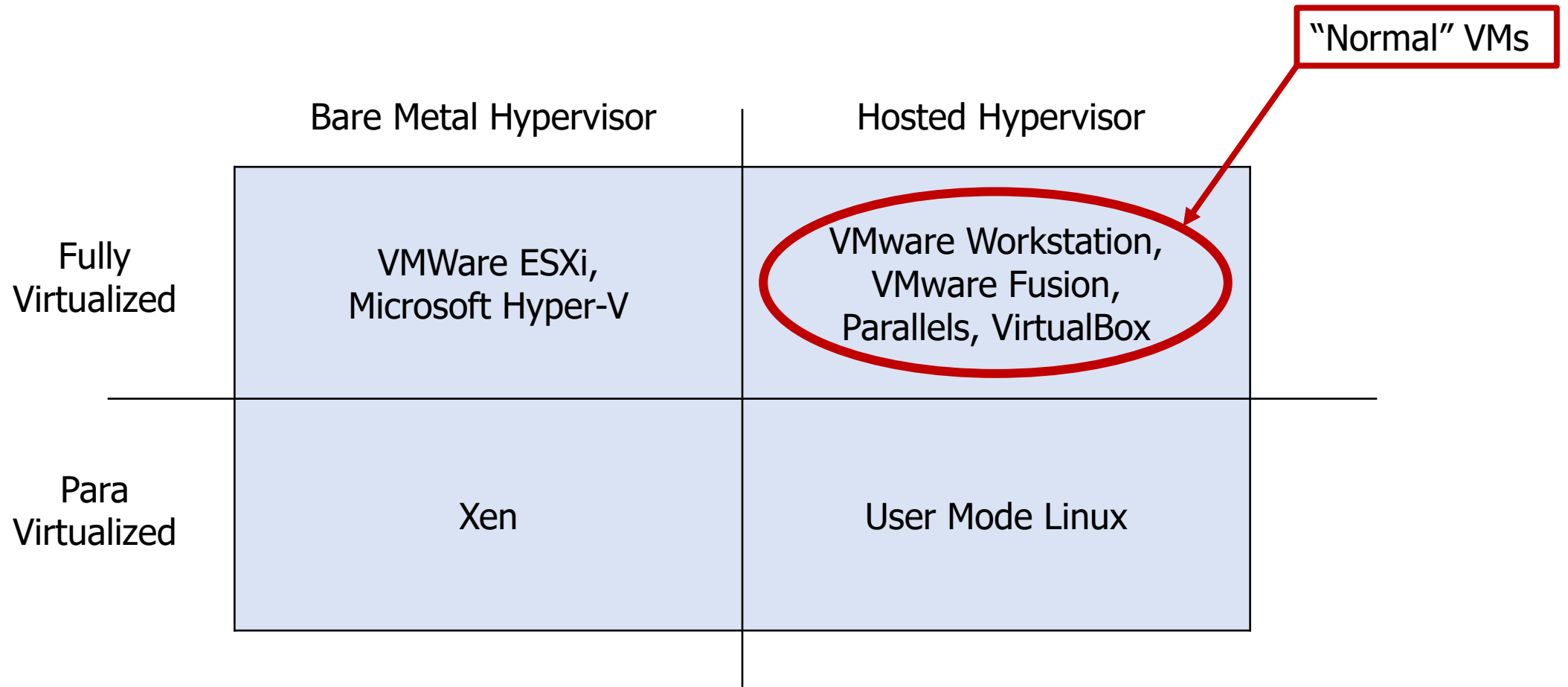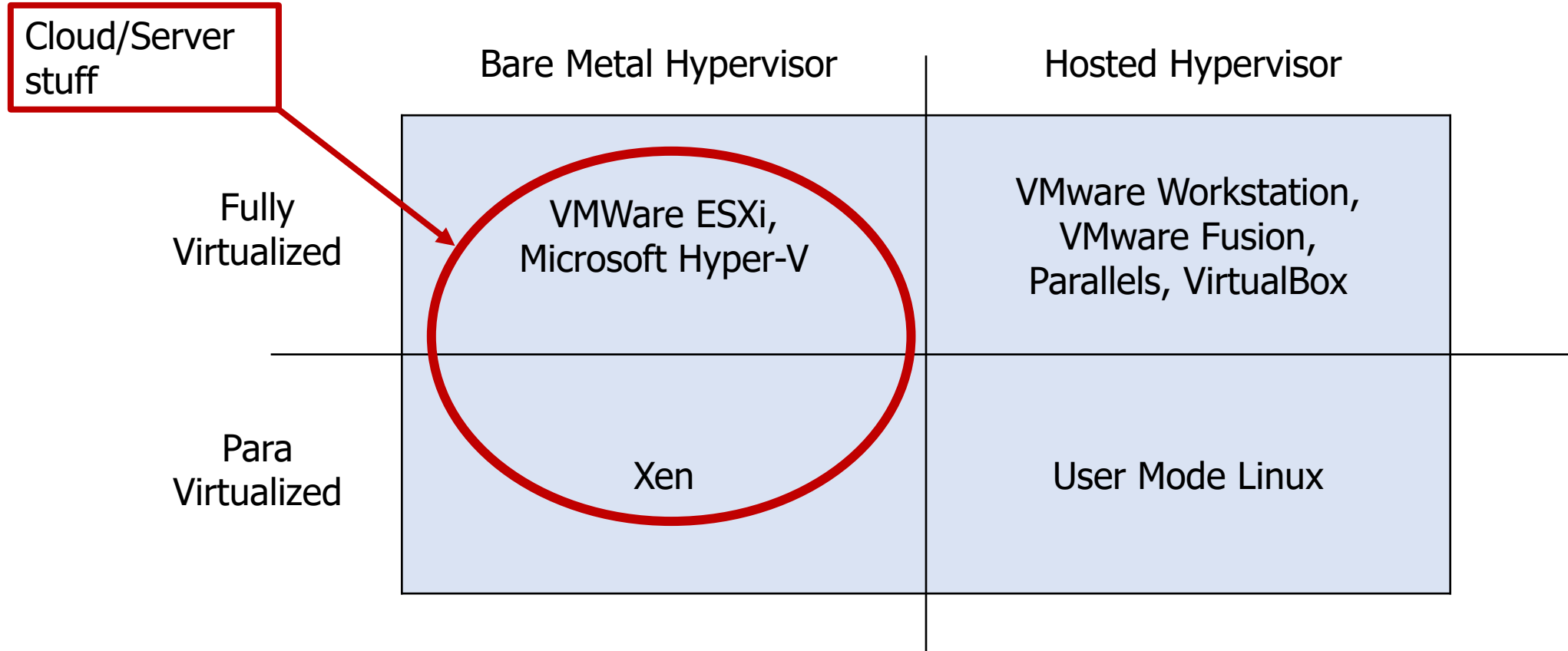
Hypervisor example: system call

Process	**Guest OS**	**Hypervisor**
1. System call: trap to OS		
		2. Receive trap. Call guest OS trap handler
	3. OS trap handler: Decode trap and execute syscall. When done issue return-from-trap	
		4. OS tried to return from trap. Do real return-from-trap
5. Resume execution		

Hypervisor challenges: privileged instructions

- The guest OS is going to run privileged instructions
  - Scheduling threads, editing page tables, modifying interrupt state

- Cannot let it have full control over the hardware
  - Otherwise it really isn't a "guest" and host might never regain control

- Solution: trap into hypervisor
  - Bare metal: Illegal instruction fault goes directly to hypervisor
  - Hosted: Illegal instruction fault in Host OS passed to hypervisor
    - Which can actually do something to handle it!!

Problem: x86 doesn't virtualize very well

- CPU architecture is virtualizable only if sensitive instructions always trap if run in user mode

- Historically, x86 does not guarantee this
  - Some instructions behave differently in user mode
  - For example: some instructions have no effect when run in user mode

- One solution: binary translation
  - Find all unacceptable instructions in the OS binary (possibly at runtime)
  - Replace with different instructions that trap to hypervisor
    - Which will perform the originally desired operations

Virtualization extensions to x86

- Intel VT and AMD-V
  - Extensions to instruction set architecture to enable virtualization
  - Fix virtualization problems
  - Also speed up virtualization performance by requiring less trapping

- VM Entry/Exit
  - Swap out Virtual Machine Control Structure (VMCS) that specifies OS state
    - Registers, Address Space, Executing Threads
  - Example optimization: Virtual Processor ID in TLB entries
    - Allows Guest OS and Host OS to share a TLB

Hypervisor challenges: Memory virtualization

- Guest OS maintains its own page tables, mapping virtual to physical memory
  - But the guest itself is running in virtual memory

- Hypervisor maintains "shadow page tables" that map Guest memory pages to actual memory pages
  - Guest modifications to page tables trap to hypervisor that modifies its own tables accordingly

- Virtual extensions can do this double-translation in hardware

Guest Virtual AS   Guest Physical AS   Machine Memory

Guest A

2

6

R/W

3

Guest B

2

6

R/W

9
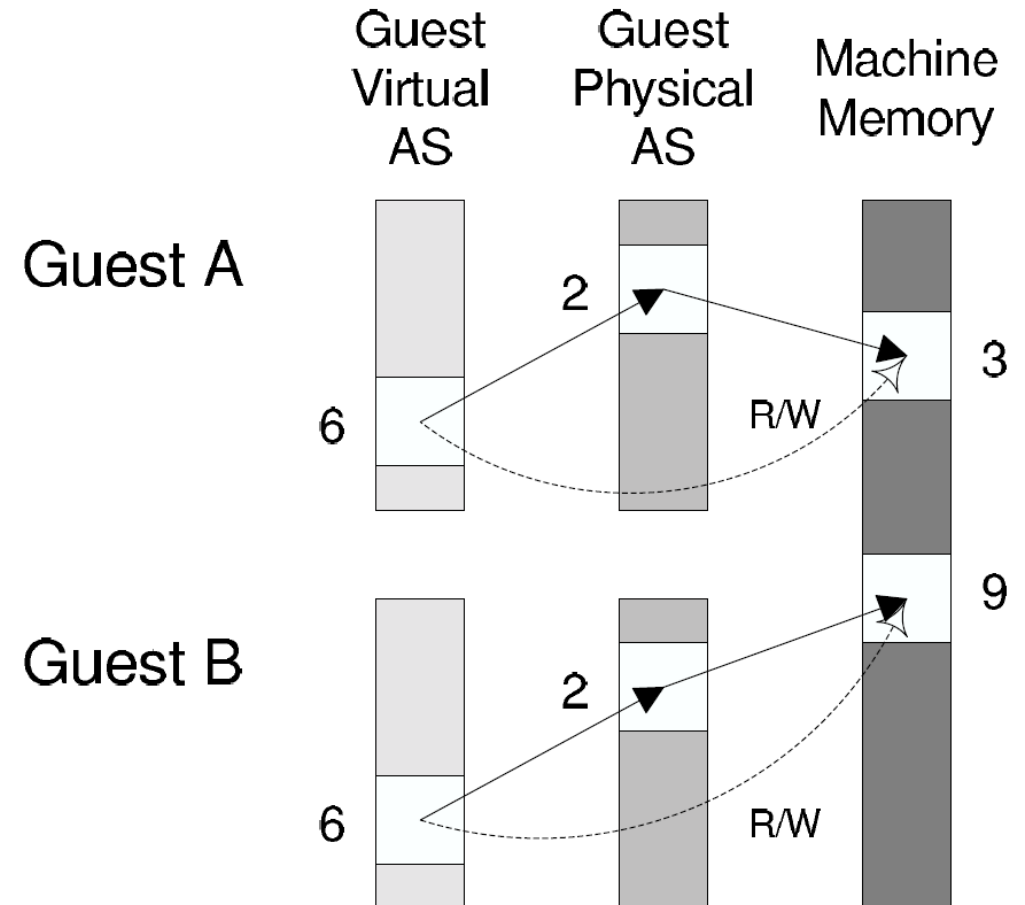
Hypervisor challenge: I/O devices

- Difficult to replicate all the different drivers that can exist in a kernel in the hypervisor

- One solution: leverage host OS drivers
    - Present virtual I/O devices to guest OS
    - Guest interacts with virtual I/O through its own device driver
    - Calls get sent to hypervisor, which makes appropriate calls to host drivers

# Break + Questions – VirtualBox on ARM Mac

- Will VirtualBox work on the new ARM Macs?
  - Will the program run as-is?

  - What architecture will the guest OS need to be?

  - Could students run CS213 labs in it?

Break + Questions – VirtualBox on ARM Mac

- Will VirtualBox work on the new ARM Macs?
  - Will the program run as-is?
    - No. Currently compiled for x86-64. Needs to be recompiled. But it probably has a bunch of hardware-specific code that needs to be rewritten too… (support exists as of October 2022!)

  - What architecture will the guest OS need to be?
    - ARM. VirtualBox is a hypervisor that runs code on the actual processor.
    - Windows and Linux do have some ARM support…

  - Could students run CS213 labs in it?
    - Not really… Any x86-64 specific stuff won't work.

Sidebar: virtualization extensions often disabled by default

- ## Most users will never have a need for them
  - ### And developers can probably figure out BIOS settings
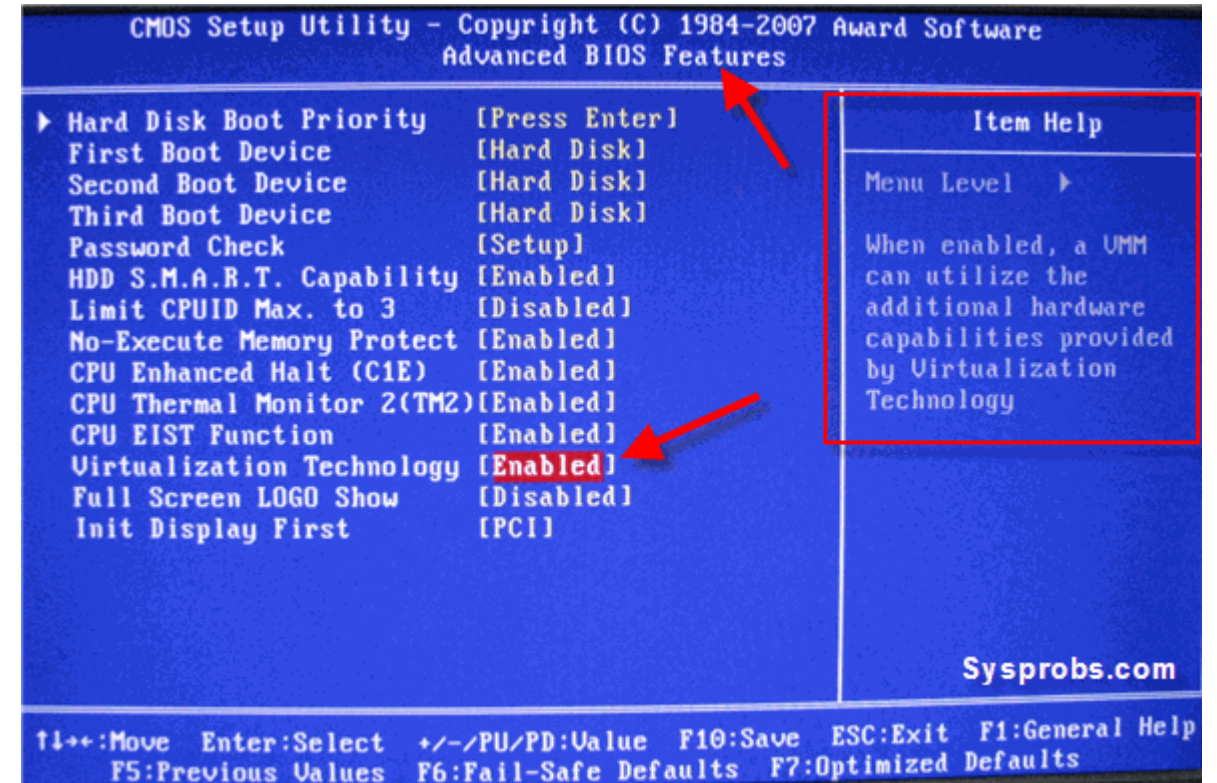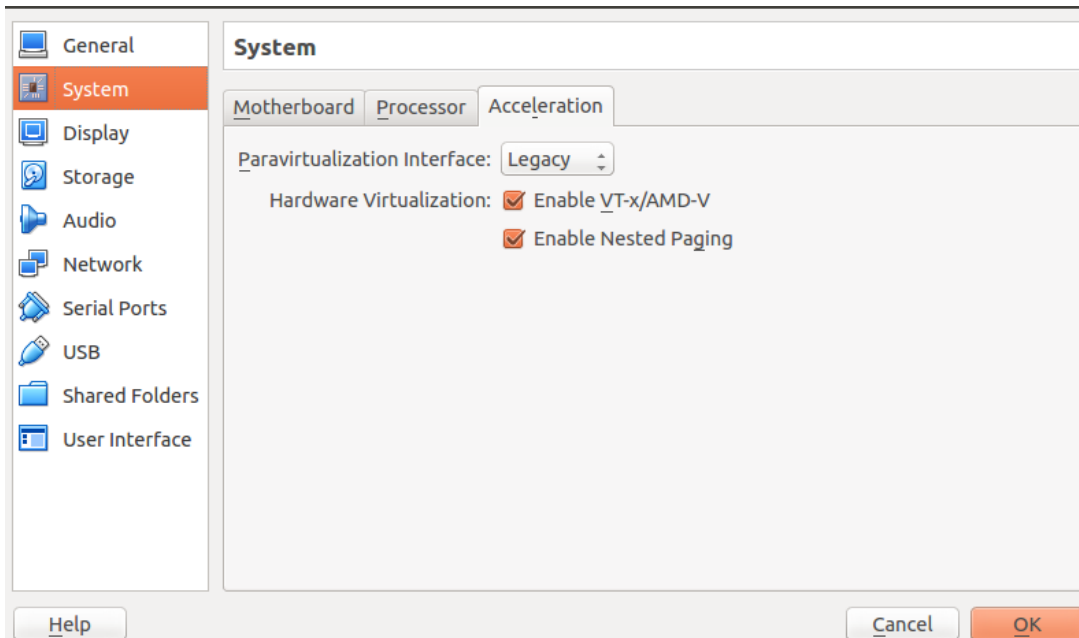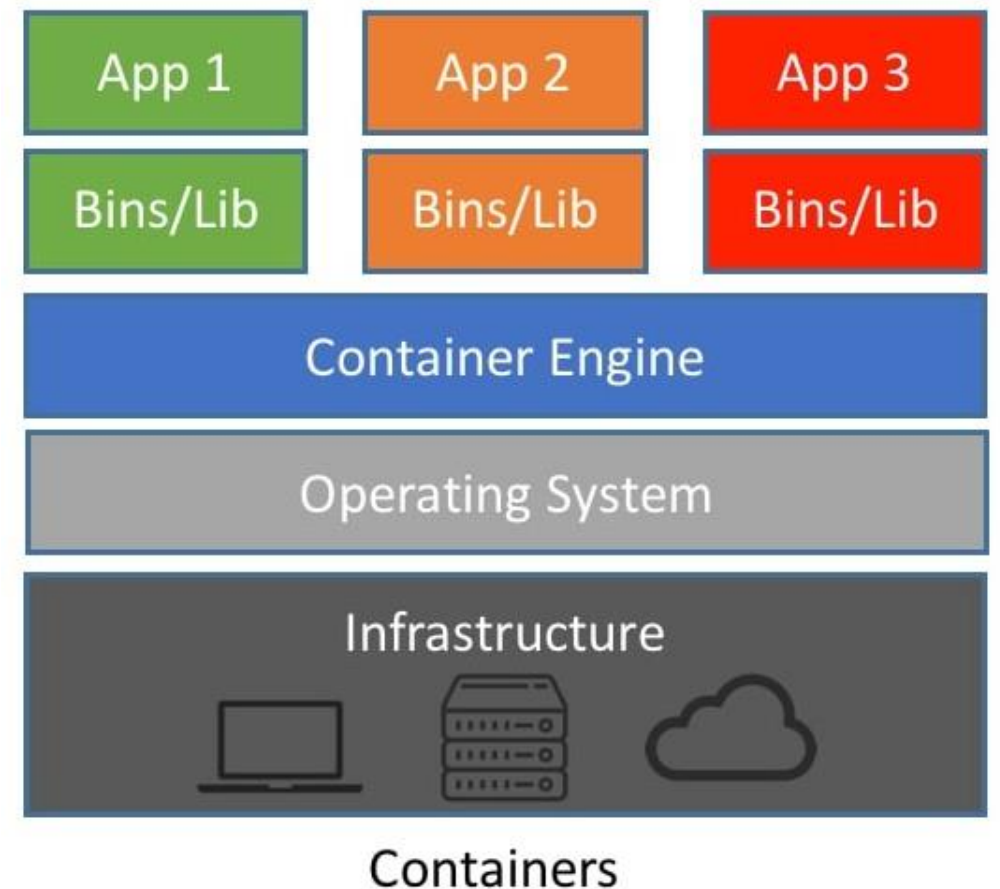
Outline

- Virtualization

- **Approaches**
  - Emulation

  - Hypervisors

  - **Containers**

Cloud platform requirements

- May want to provide multiple OSes, but can do so with multiple physical machines

- Really want encapsulation and isolation
  - Encapsulation
    - Include particular shared libraries that application needs
    - Without interfering with other applications on system
  - Isolation
    - Guarantee certain processing and memory allocations to each application
    - Limit visibility into the filesystem (without overhead of partition per app)

Containers

- Provide each application with illusion of its own dedicated OS
  - Isolated resources: processor and memory

  - Isolated namespace: PIDs, network, filesystem

  - Includes only the binaries and libraries it needs

Containers

Visual comparison of Hypervisors and Containers

Hypervisor Type 1
„bare metal hypervisor"

Virtual Machine (VM)

App 1	App 2	App n
Bins/Libs	Bins/Libs	Bins/Libs
Guest OS	Guest OS	Guest OS

Hypervisor

Infrastructure

Hypervisor Type 2
„hosted hypervisor"

Virtual Machine (VM)

App 1	App 2	App n
Bins/Libs	Bins/Libs	Bins/Libs
Guest OS	Guest OS	Guest OS

Hypervisor

Host Operating System

Infrastructure

Container

Container

| App 1 | App 2 | App n |
| Bins/Libs | Bins/Libs | Bins/Libs |

Container Engine

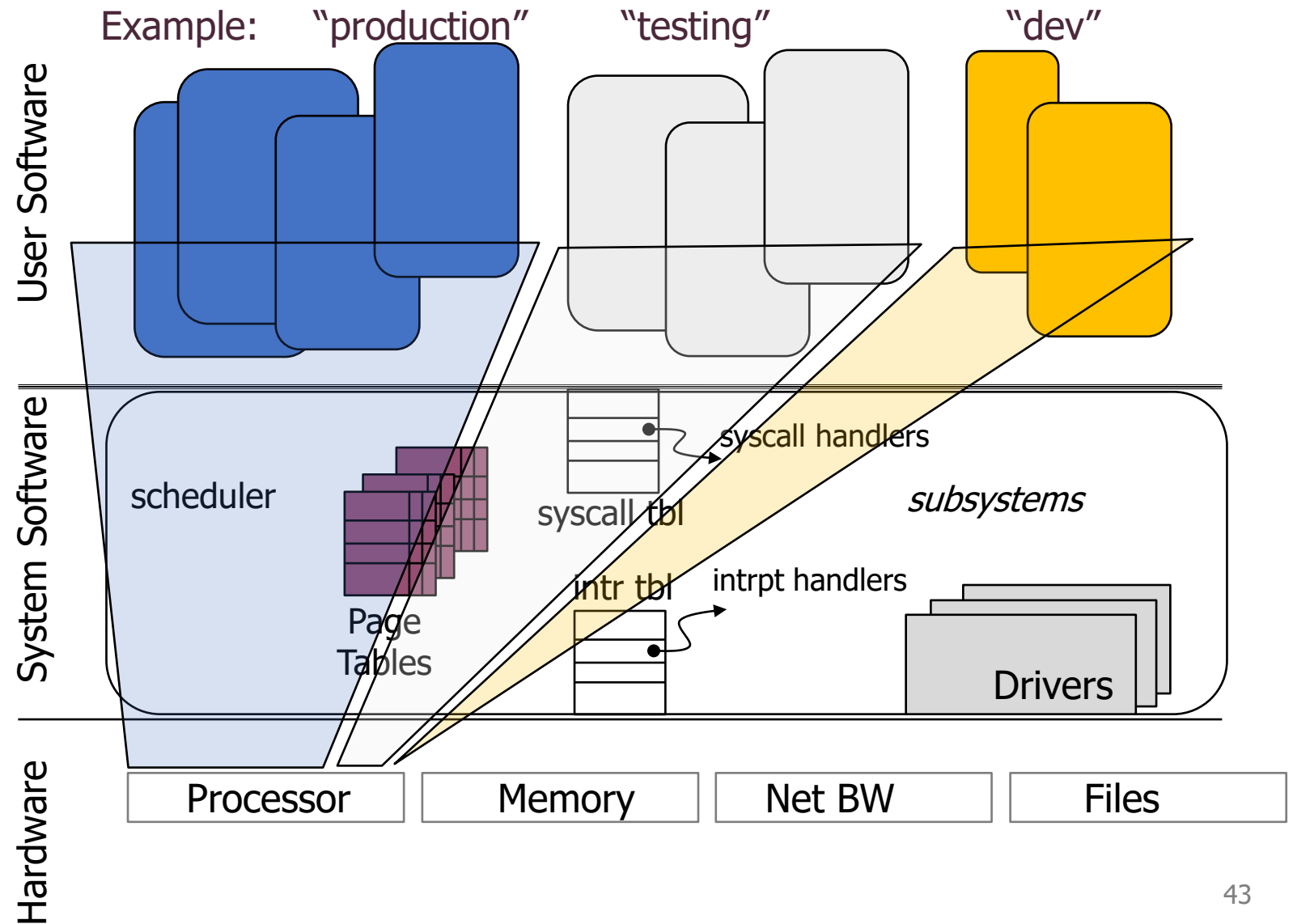Host Operating System

Infrastructure

How to implement containers

- Needs the ability to isolate one process from the effects of another
  - More strongly than a normal OS does anyways

- On Linux, there is a related idea which is the basis for containers
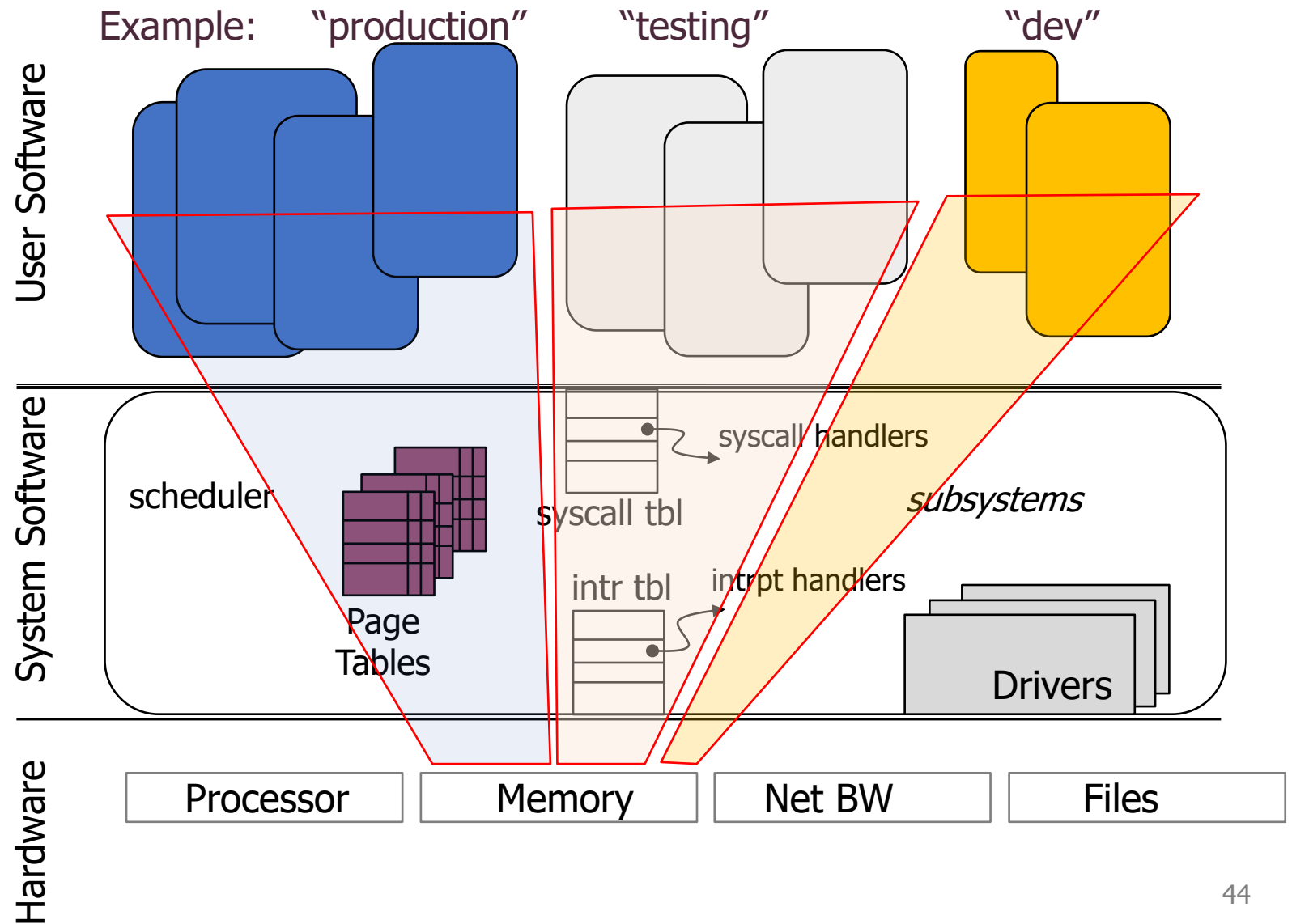 - cgroups

Linux cgroups (control groups)

- Collection of processes treated as a group for resource allocation

- Provider greater performance isolation between cgroups than between processes
  - Firm limits per group

Example: "production" "testing" "dev"

User Software

System Software

scheduler

Page Tables

syscall tbl

syscall handlers

intr tbl

intrpt handlers

subsystems

Drivers

Hardware

| Processor | Memory | Net BW | Files |

Linux cgroups (control groups)

- Collection of processes treated as a group for resource allocation

- Provider greater performance isolation between cgroups than between processes
  - Firm limits per group

Example: "production"   "testing"   "dev"

User Software

System Software

scheduler

Page Tables

syscall tbl

intr tbl

syscall handlers

intrpt handlers

subsystems

Drivers

Hardware

| Processor | Memory | Net BW | Files |

44

cgroups can be used to build containers

- Devices can be connected or denied to a cgroup
  - cgroup processes will not be able to detect device at all

- Accounting can be done on cgroup usage
  - Memory, CPU, disk I/O

Docker

- Container packaging, distribution, and execution
  - Also created open standard for container runtimes

- Images
  - Describes starting state of a Docker container
  - Like a snapshot of the system

- Union file system
  - Image describes file system as a sequence of layers
    - Each layer includes some files
  - Overall file system is the *union* of all the layers
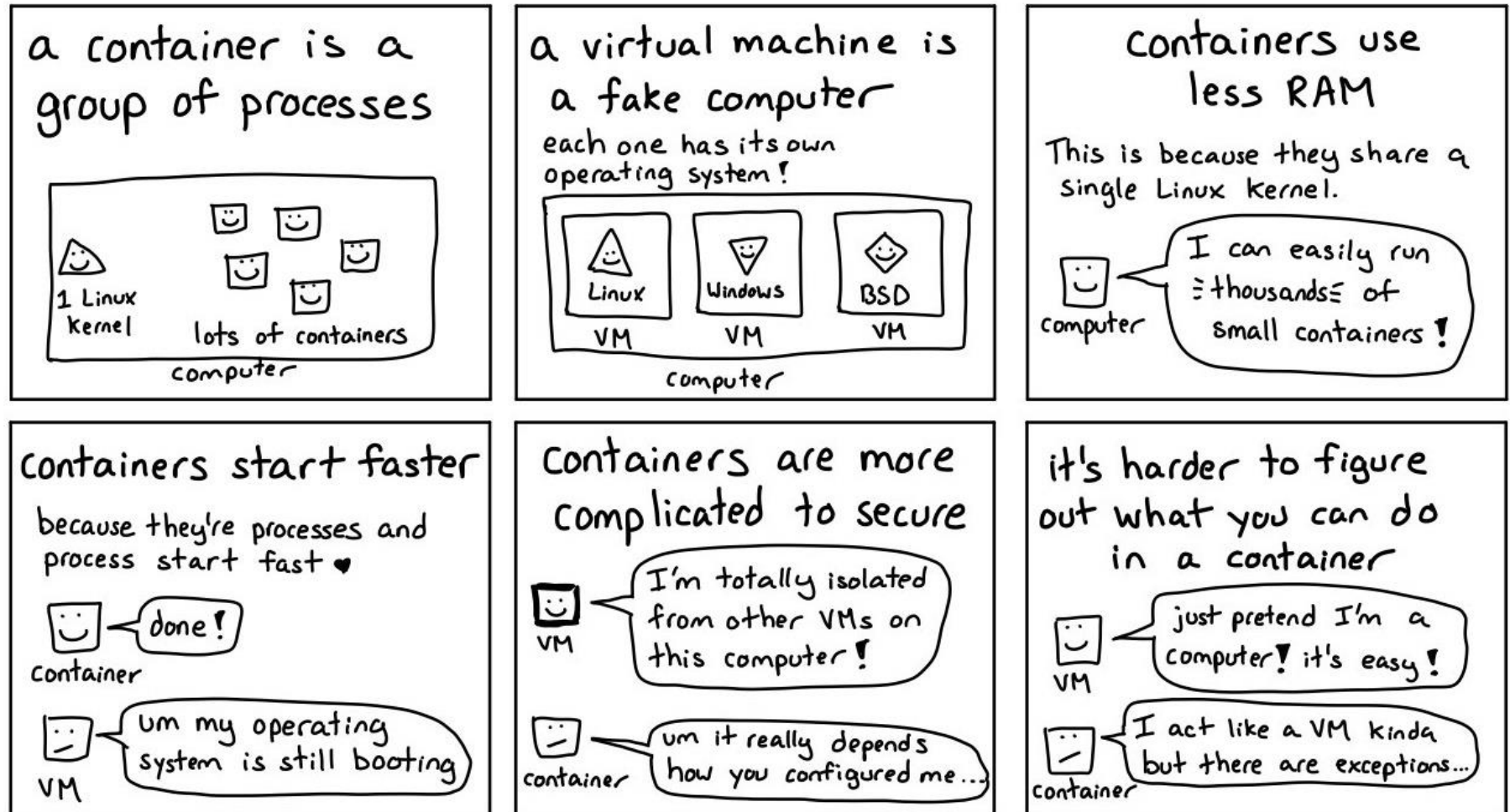 - Layers can be reused in different images

Docker use cases

- Environments are hard to set up
  - Often the hardest part of starting software development
  - Containerized applications encapsulate requirements
  - Can be run on any system that has the same kernel it was built for

- Packages an application and its requirements into a container
  - Can be used by an individual to more easily run an application
  - Can be deployed to a cloud server to run

- Example: could use Docker for QEMU + Nautilus
  - Need a very specific version of QEMU, with all the proper libraries

Bonus explanation

Outline

- Virtualization

- Approaches
    - Emulation

    - Hypervisors

    - Containers