Lecture 18: Virtualization

CS343 – Operating Systems

Branden Ghena – Fall 2025

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

Administrivia

- Today is the last lecture!

- Exam review session on Thursday
  - Come ready to work on problems

- FilesystemLab due Thursday
  - Still okay to use slip days or late penalties on it

Administrivia

- Midterm exam 2
  - Wednesday, December 10th from 12:00-1:20 pm in the lecture hall
    - Normal 80-minute exam
  - Expect a similar format to the previous exam
    - Covers Virtual Memory through Virtualization
      - Lectures 9 through 18
      - Does NOT cover scheduling or concurrency
  - Prior exams on Canvas homepage for practice
  - May bring: **one** 8.5x11” sheet with notes on front and back

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

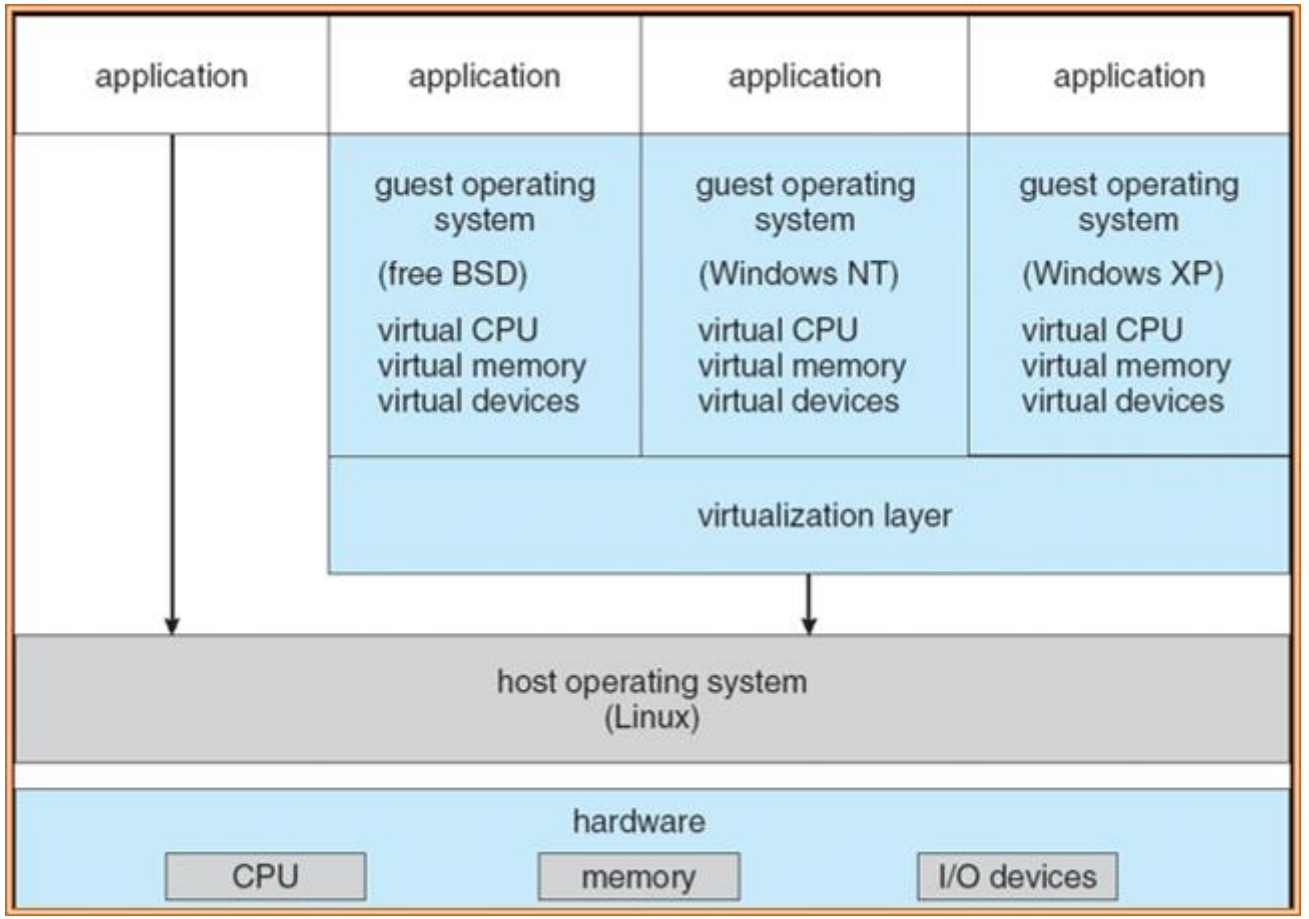
Virtualization

- 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 if you don’t need to run them simultaneously

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

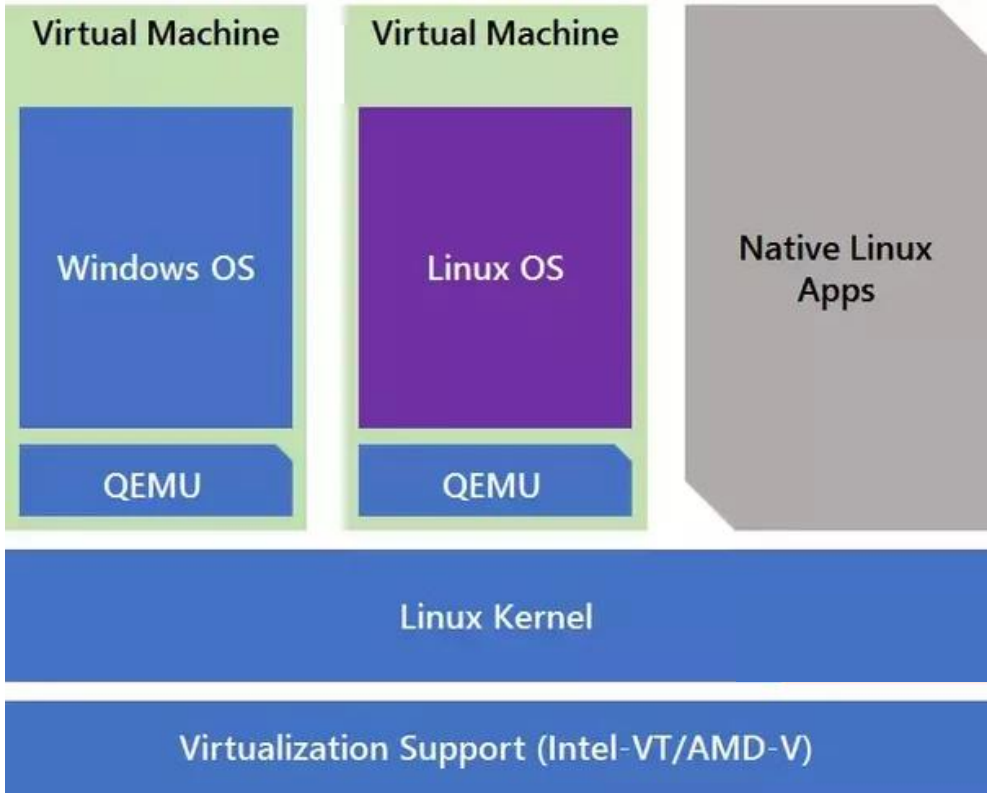
- We have been using QEMU for lab to simulate an x86-64 computer
  - 2 CPU cores
  - 2 GB of RAM
  - VirtIO GPU
  - PS/2 mouse and keyboard
  - 2 PCI IDE interfaces with hard disk and CD-ROM support
    - nautilus.iso connected to CD-ROM
  - Serial and parallel ports
    - stdio connected to serial port
    - 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
  - You could even run multiple emulators simultaneously if you wanted to

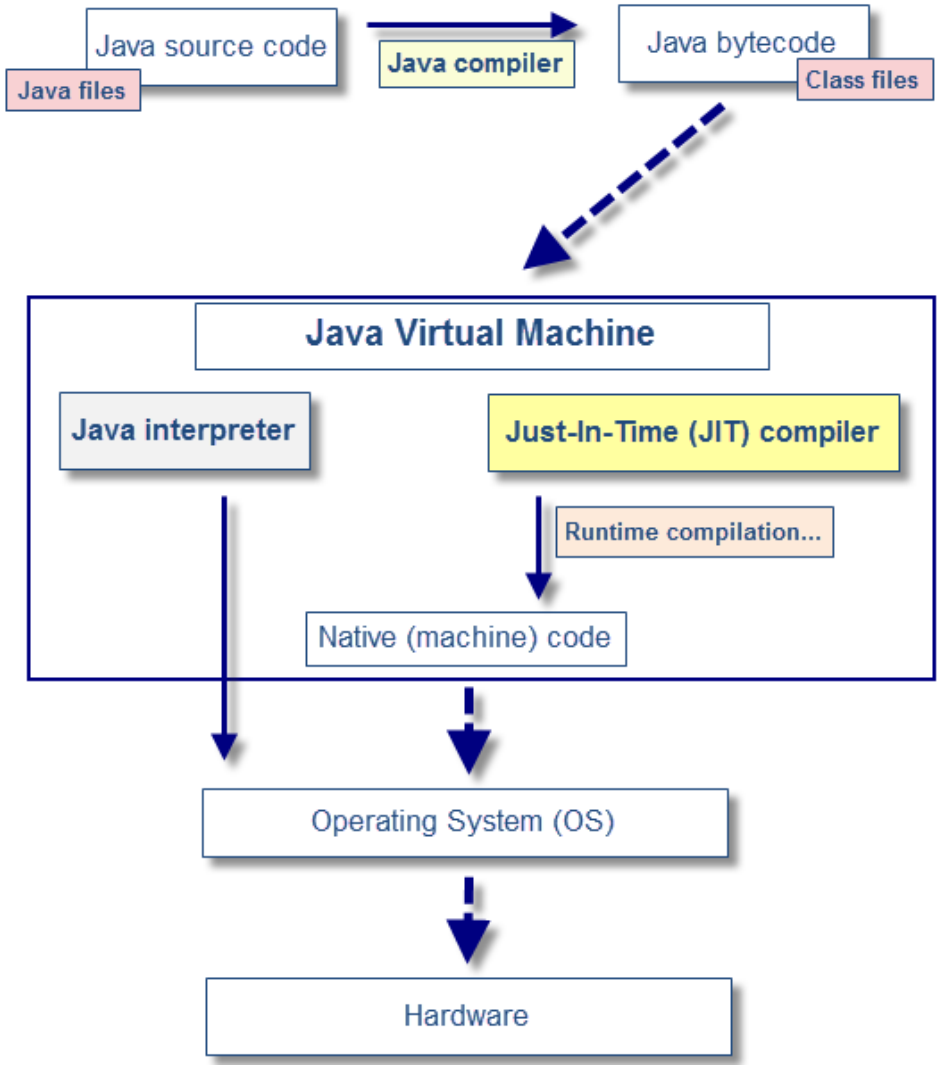
- Downsides
  - Accurate behavior is difficult to provide
  - Software runs slower than the hardware would

    (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



WebAssembly: modern version of a language emulator

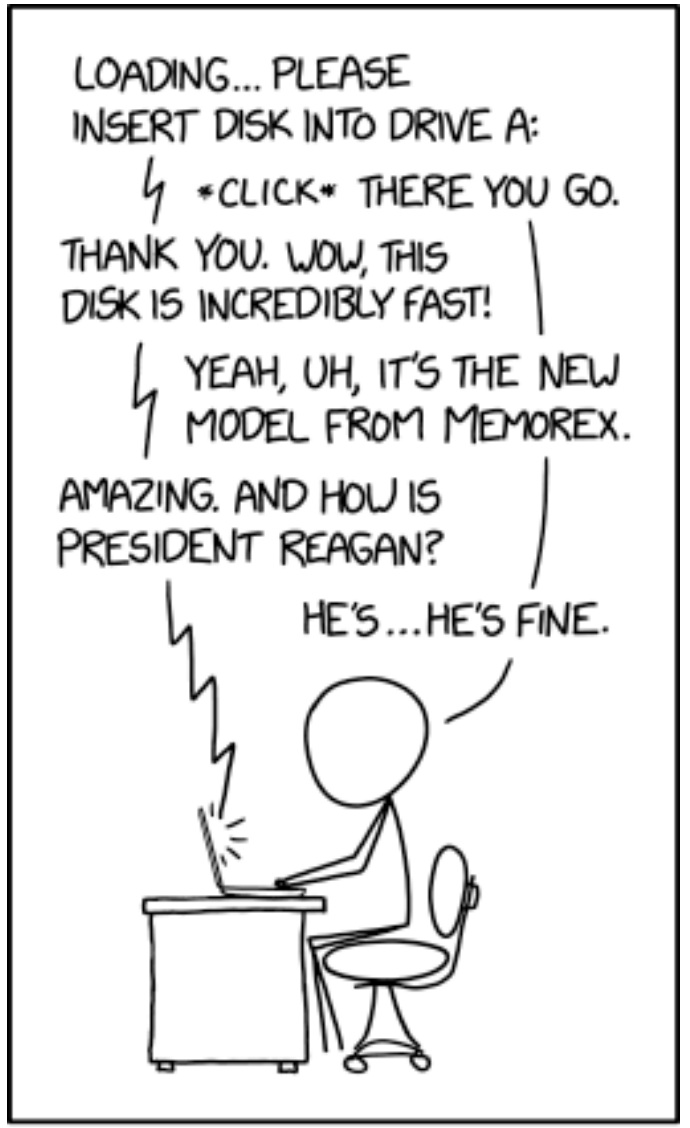
- WebAssembly (Wasm) is a compilation target
  - Like x86-64 or ARM
- Compilers generate Wasm from languages like C, C++, or Rust
- Web browser runs a virtual machine the understands and executes the Wasm instructions
  - With an API to make requests to the actual computer OS
- End result: C/C++ programs can run in Web browsers
  - While maintaining possibly-decent performance (10-100% slower)

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
  - Usually translated in advance
  - Also has just-in-time (JIT) translation
    - Used for runtime-generated x86-64 code
  - Works fine for applications that are I/O bound
- Simulates a different CPU, but leaves the remainder of the computer the same

Break + Question + Comic

- What are the best use cases for hardware emulation?



I FEEL WEIRD USING OLD SOFTWARE THAT DOESN'T KNOW IT'S BEING EMULATED.

https://xkcd.com/2221/

Break + Question + Comic

- What are the best use cases for 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 (for performance)
    - Entirely controlled environment: makes debugging easier
    - Nautilus does run significantly slower in QEMU, but that's okay for our purposes

I FEEL WEIRD USING OLD SOFTWARE THAT DOESN'T KNOW IT'S BEING EMULATED.

https://xkcd.com/2221/

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

Hypervisors use real hardware for performance

- **Big idea:** Hypervisors run the guest OS on the real hardware for performance
  - Just like we do with applications

- Normal code runs directly on hardware as usual
  - Assembly instructions, accessing memory, normal application stuff
  - Significantly faster than emulation

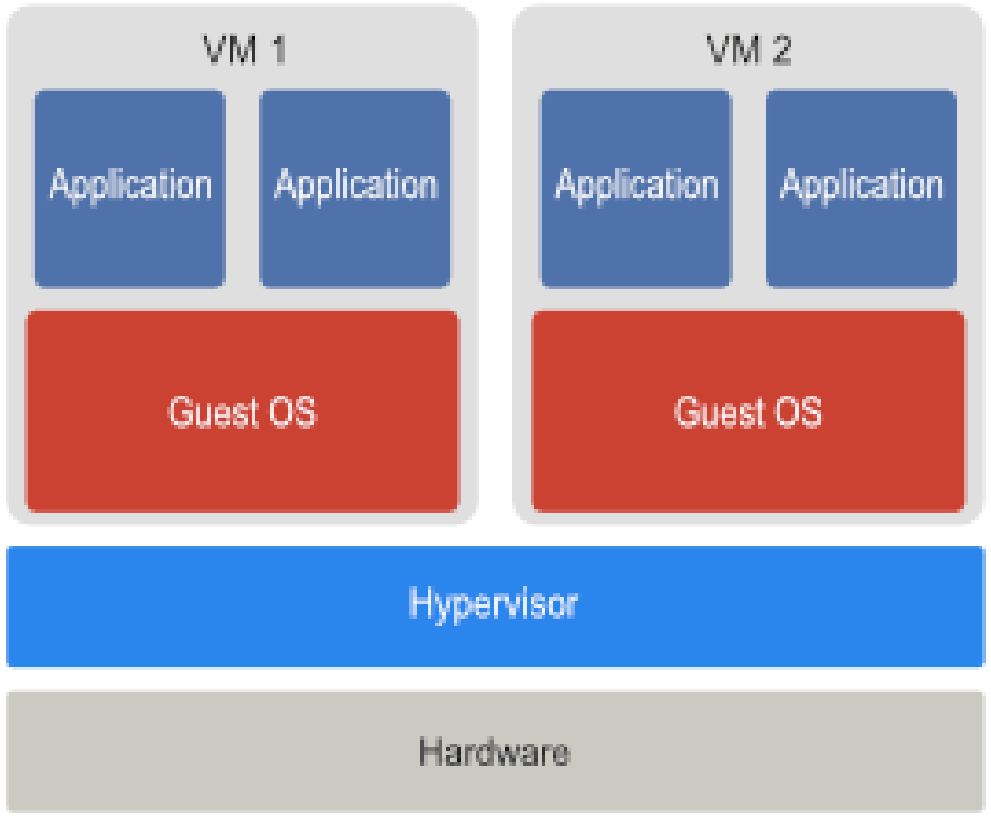
- Accesses requiring kernel permissions go through the hypervisor first, then get redirected to the guest OS
  - Context switching, virtual memory changes, device access, etc.
  - Hypervisor may intervene on some of these

Hypervisor versus Emulator

- Emulator translates all code that runs
  - Executes native code that provides the same results

- Hypervisor is only involved in "tricky" operations
  - Normal application code runs right on the hardware
  - Normal OS code *also* runs right on the hardware
  - Privileged code traps into the hypervisor to handle it

Hypervisor layering option: directly on hardware

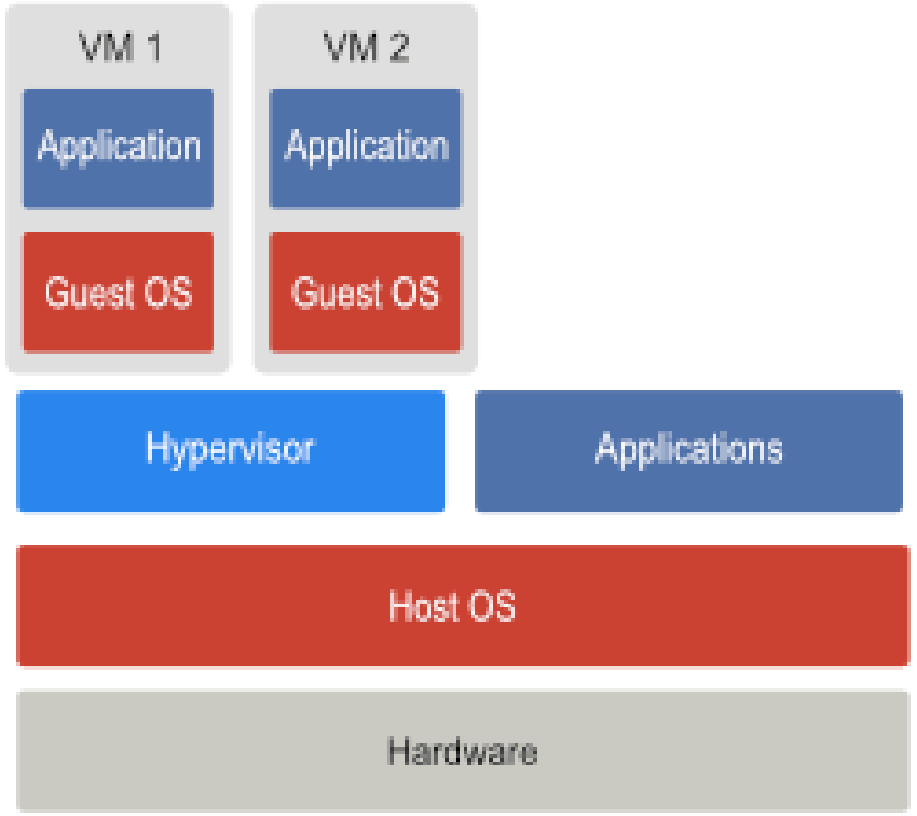


Type 1
"Bare Metal" Hypervisor

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

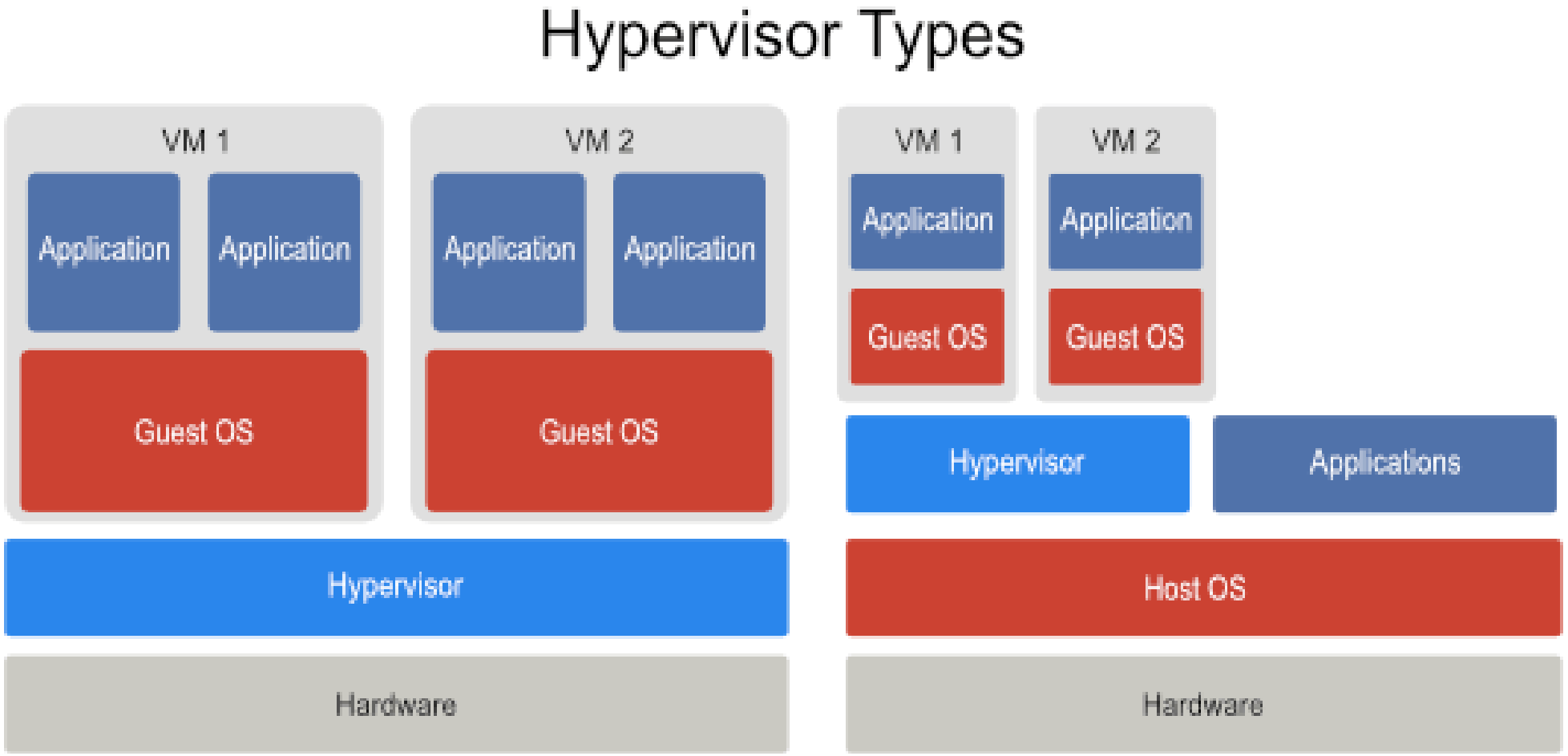
Hypervisor layering option: on top of Host OS

- 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 host kernel itself



Type 2
Hosted Hypervisor

Hypervisor layering options: comparison



"Bare Metal" Hypervisor

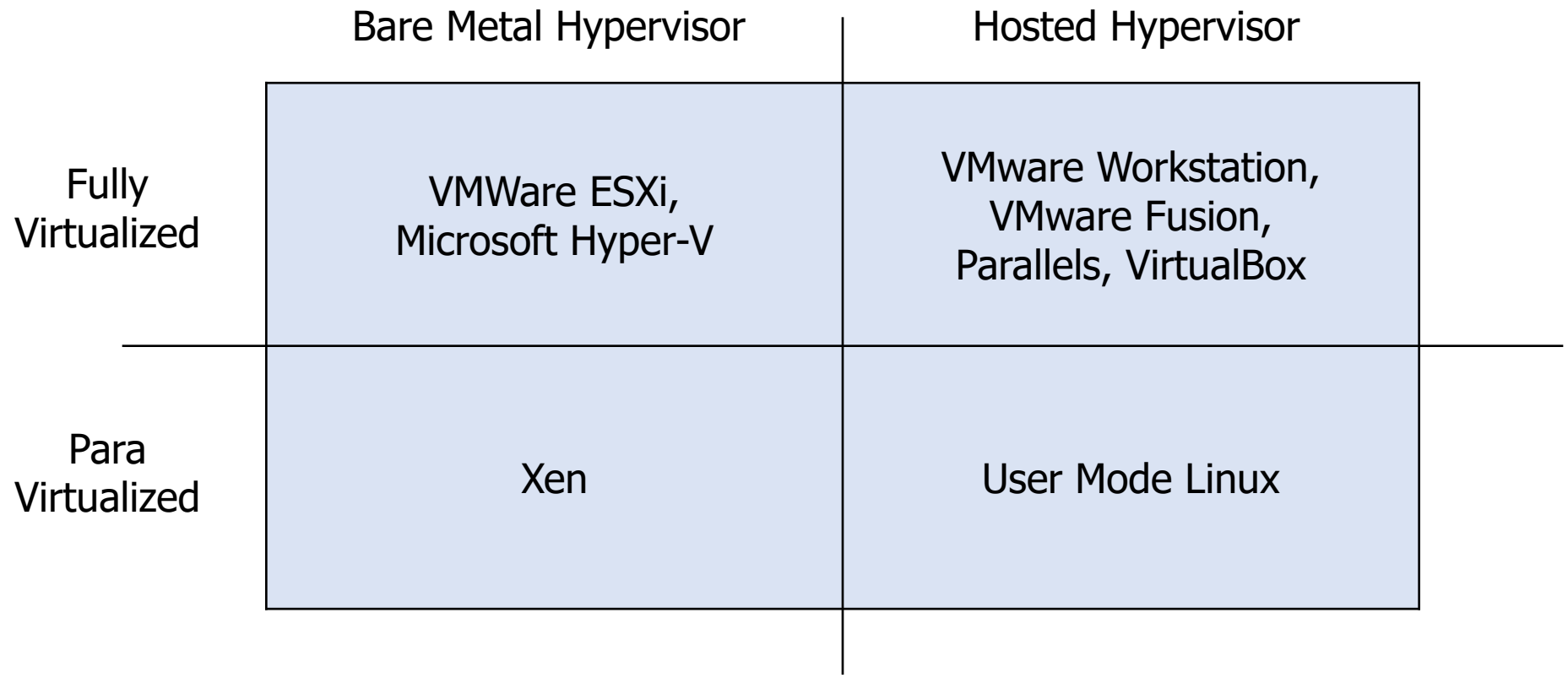
Hosted Hypervisor

Abstraction choices for hypervisor

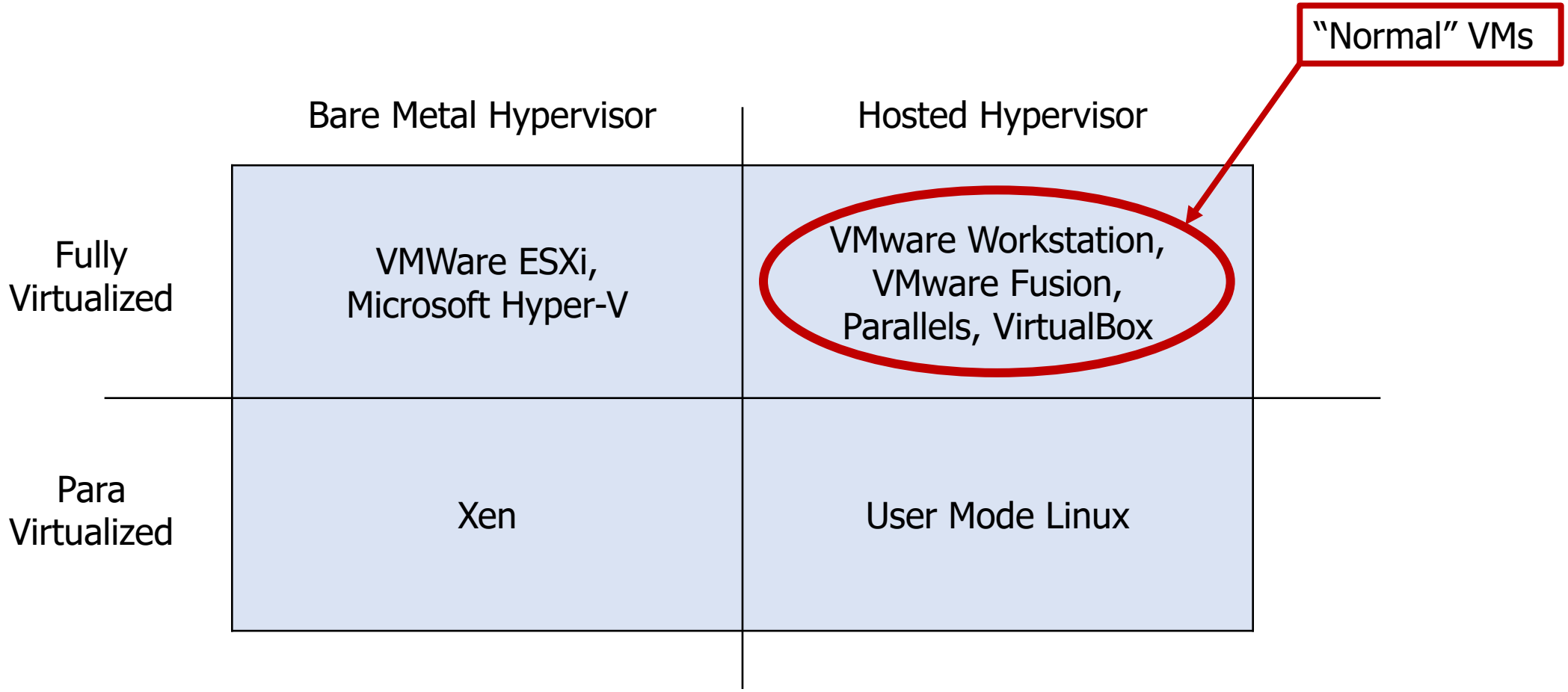
- Fully virtualizing hypervisor
  - Virtual machine looks exactly like a physical machine
    - Though not necessarily the exact same machine it's actually running on
  - Guest OS does not need to be modified in any way
  - Guest may not even be aware it's running virtually

- 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

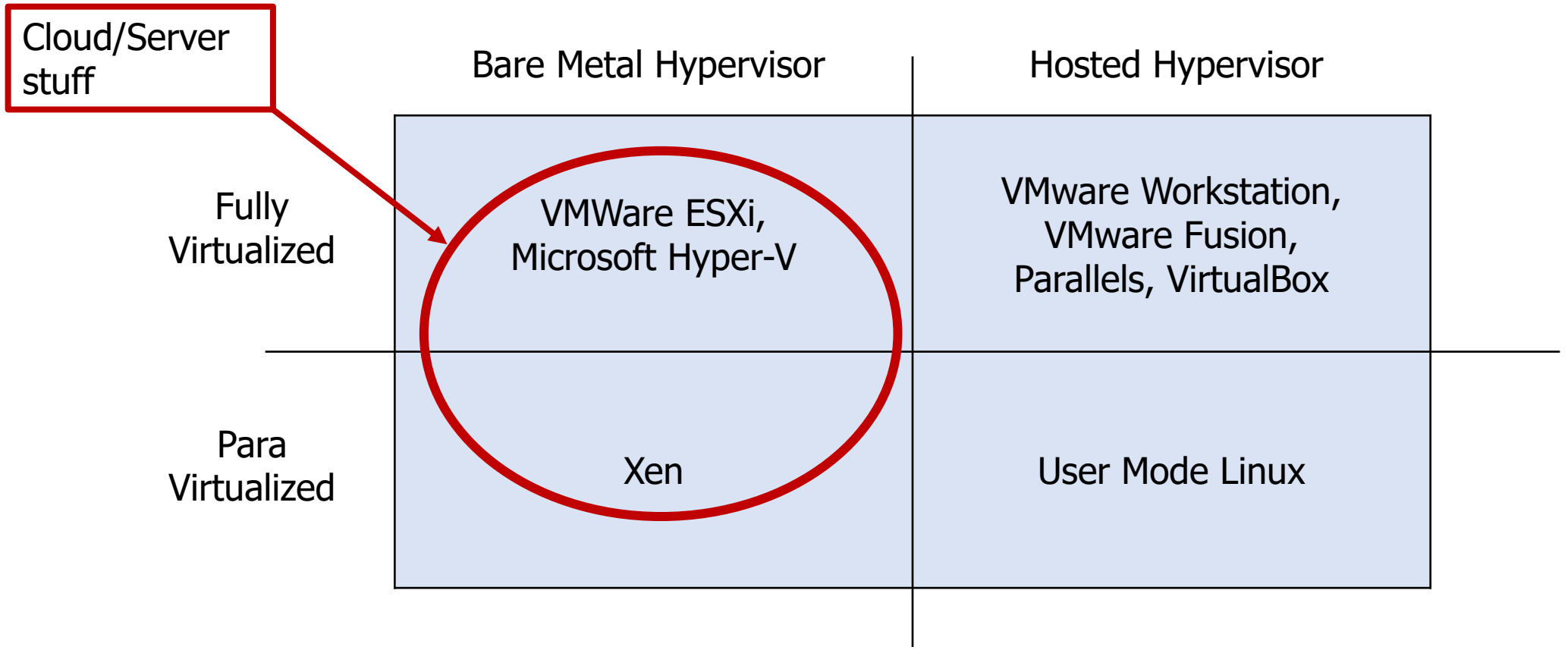
Arbitrary combinations of these are possible



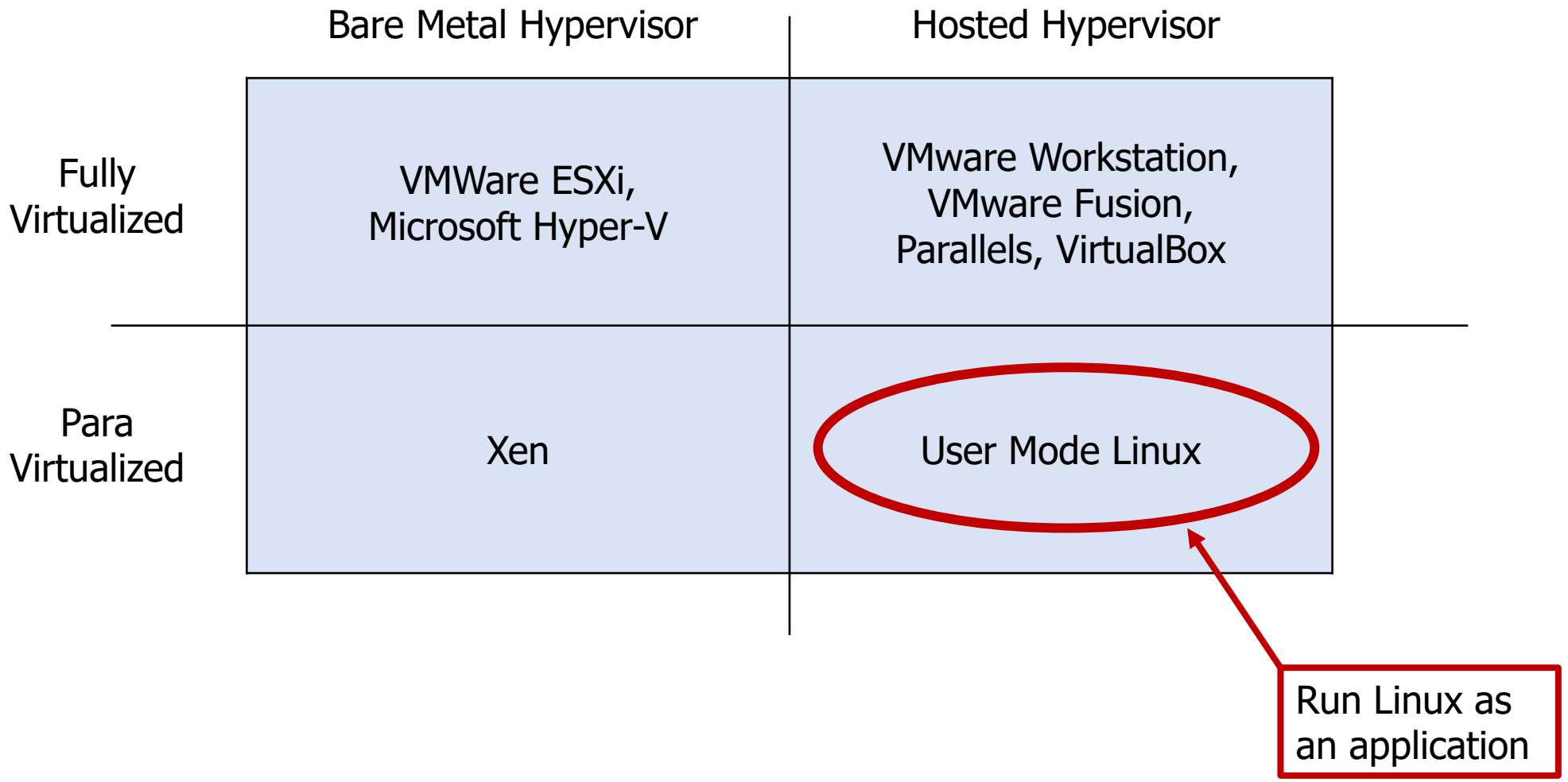
Arbitrary combinations of these are possible



Arbitrary combinations of these are possible

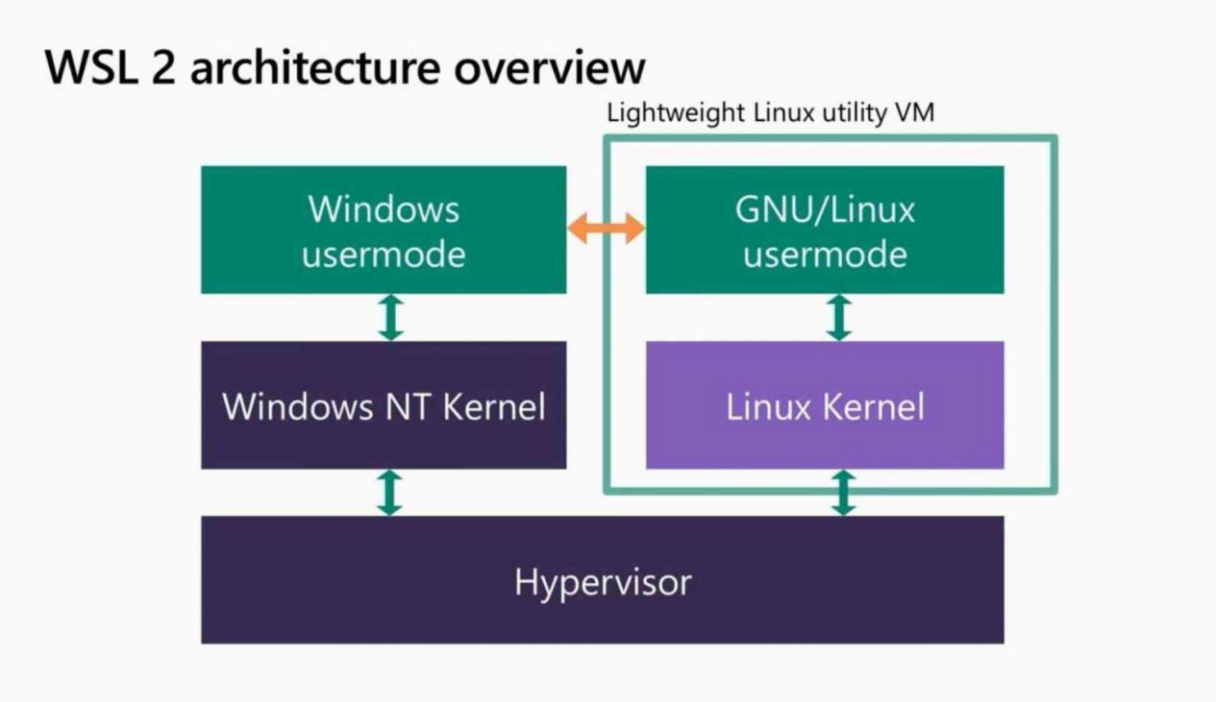


Arbitrary combinations of these are possible



Windows Subsystem for Linux (WSL 2)

- Provides Linux environment on a Windows computer

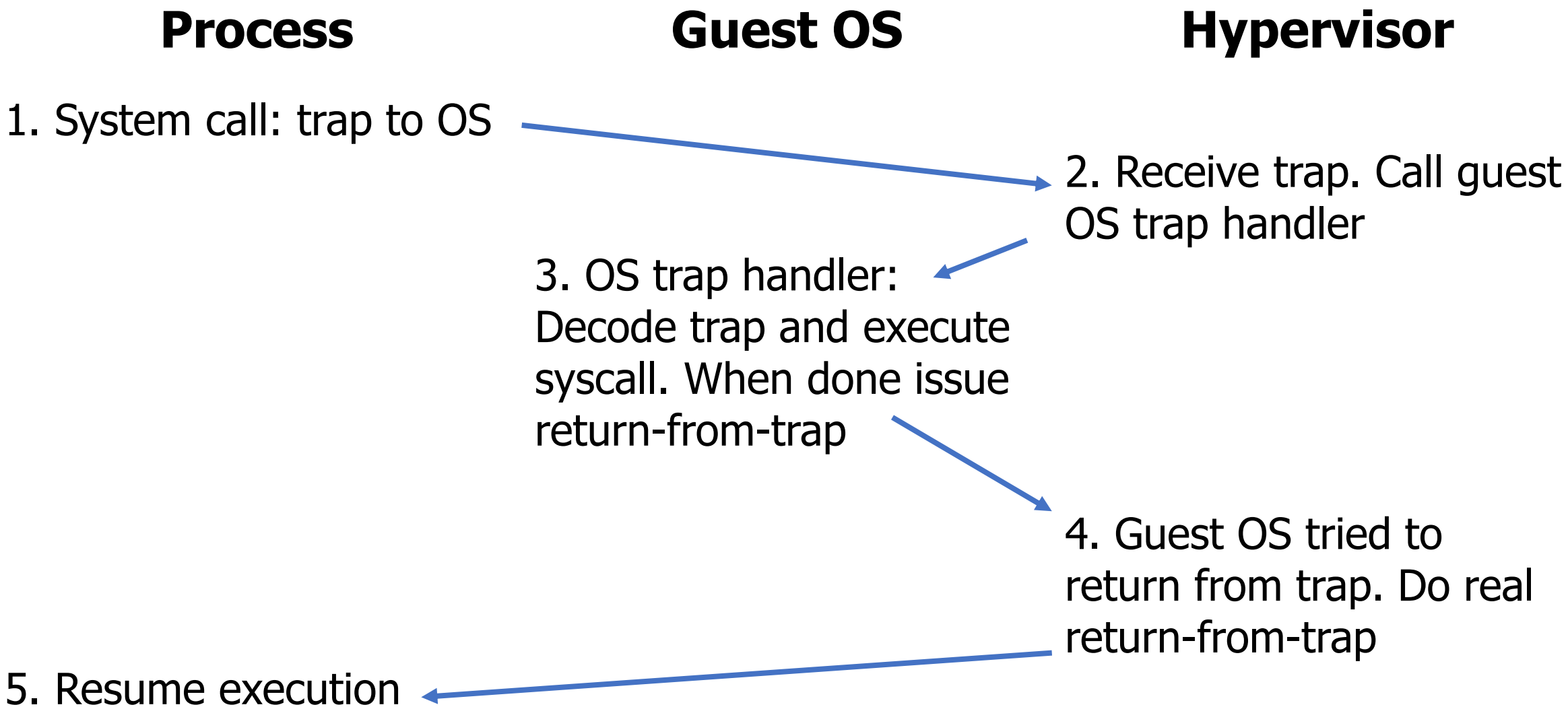


- Windows Hyper-V provides a fully-virtualized system
  - Windows and Linux both run on top of it
- Differences:
  - WSL2 also enables communication between userspaces rather than fully isolating the two
    - They can both share access to the file system, for example
  - Windows is a "privileged" guest with full control of the system

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!!

Hypervisor example: system call



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

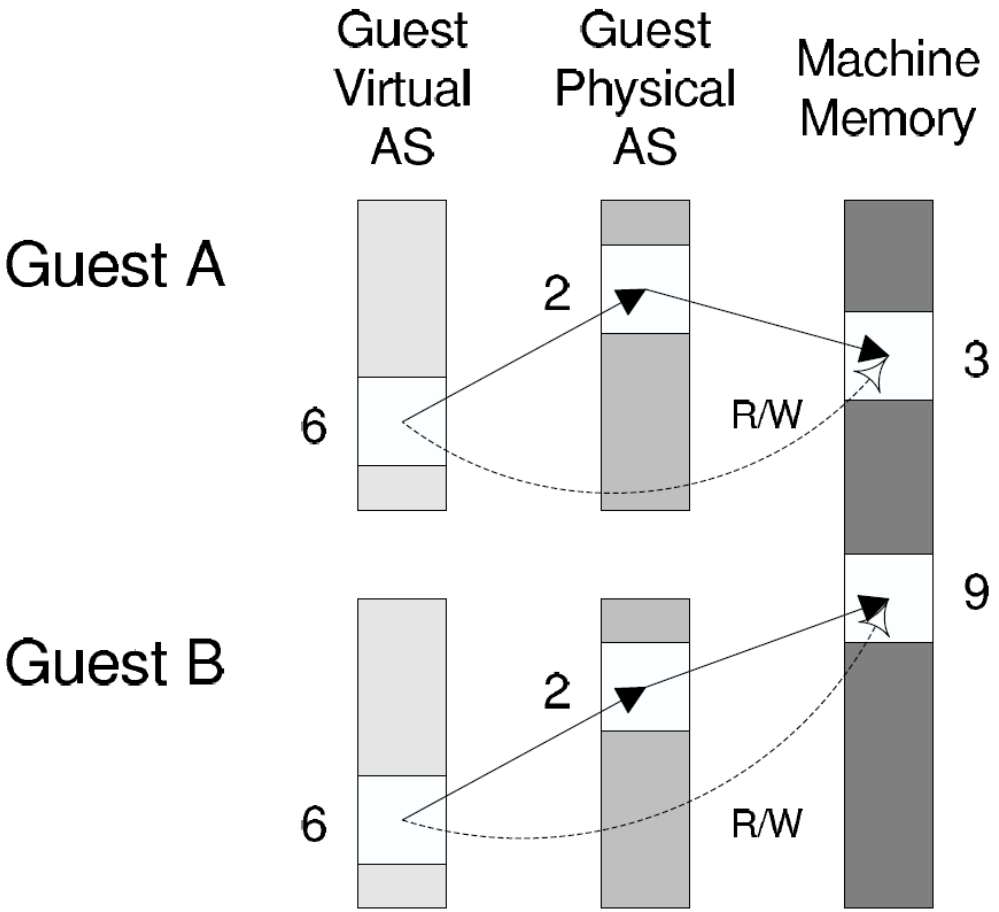
- Intel VT and AMD-V
  - Extensions to instruction set architecture to enable virtualization
  - Fix virtualization problems
  - Also speed up virtualization performance by requiring less trapping

- 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



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

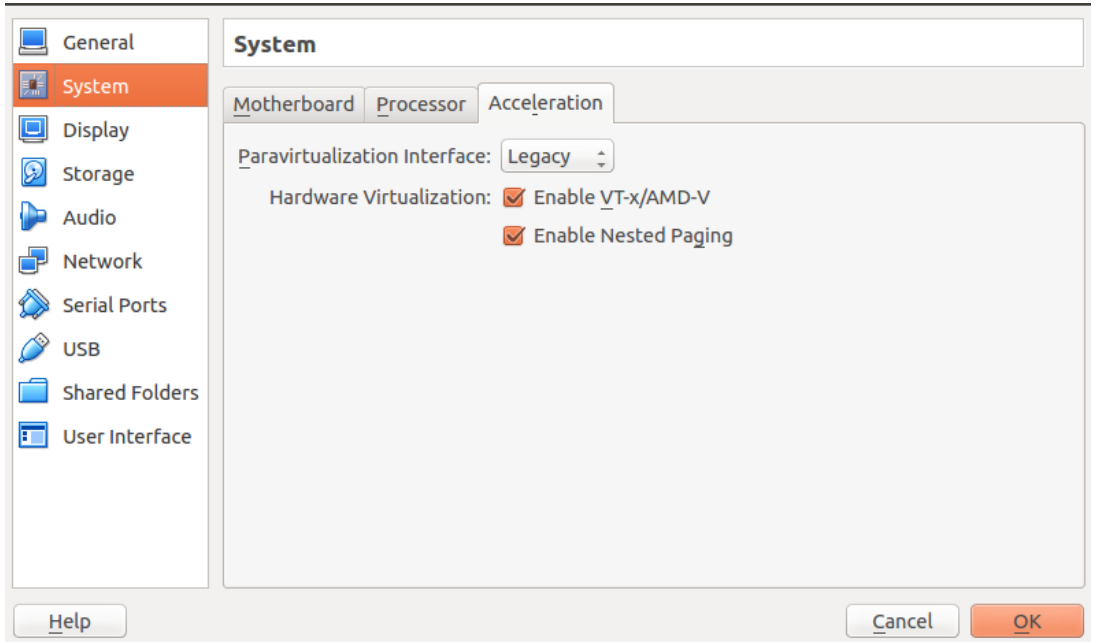
- Will VirtualBox work on the ARM Macs?
  - Will old versions of VirtualBox work?
  - What architecture will the guest OS need to be?
  - Could students run CS213 labs in it?

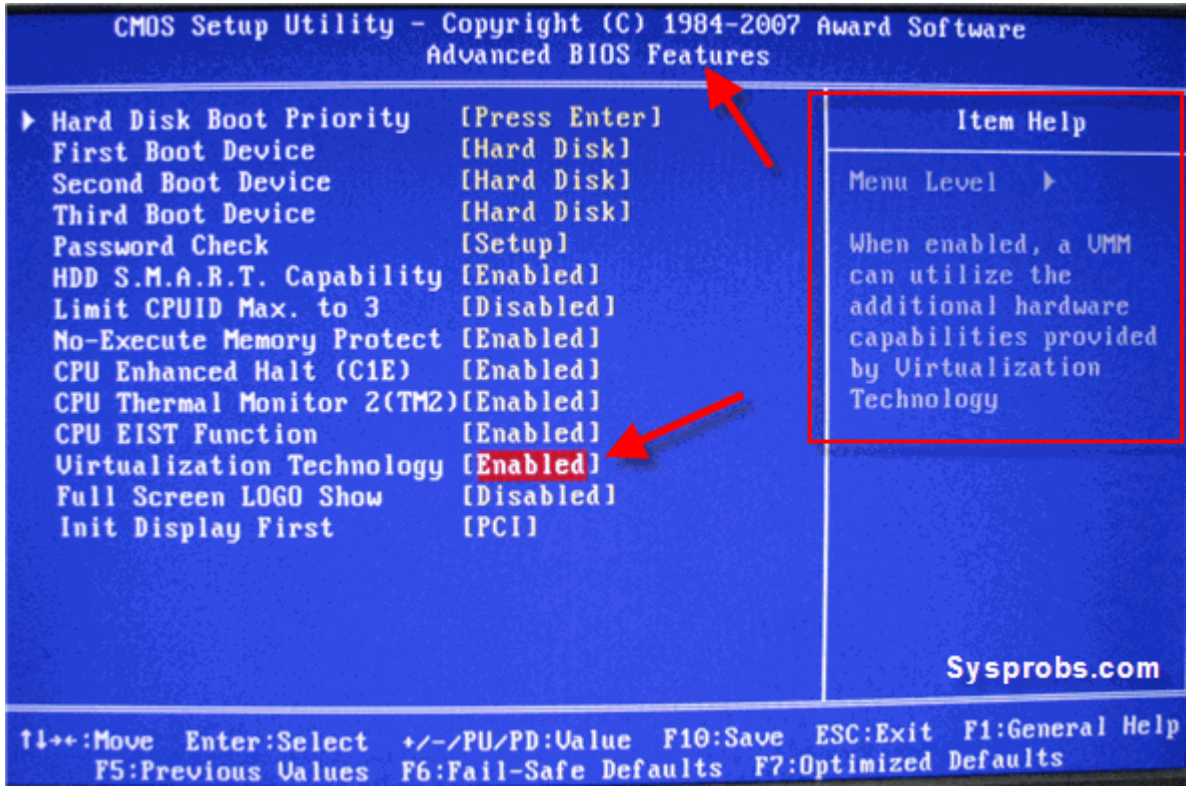
Break + Questions – VirtualBox on ARM Mac

- Will VirtualBox work on the ARM Macs?
  - Will old versions of VirtualBox work?
    - No. Originally compiled for x86-64 only. Needs a new version written for ARM. 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.
    - They would need an emulator for x86-64 assembly

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**

Not-quite-containers: Compatibility layer

- To allow software to run on another OS, provide an interface that can translate system calls
  - Essentially emulating an OS, rather than computer hardware

- Wine
  - Provides dynamic libraries that mirror those that exist in Windows
    - But new implementations make Linux system calls
  - Proton: distribution of Wine + a bunch of other tools to support games

- WSL (1.0)
  - Translates Linux system calls into Windows system calls

Cloud platform requirements

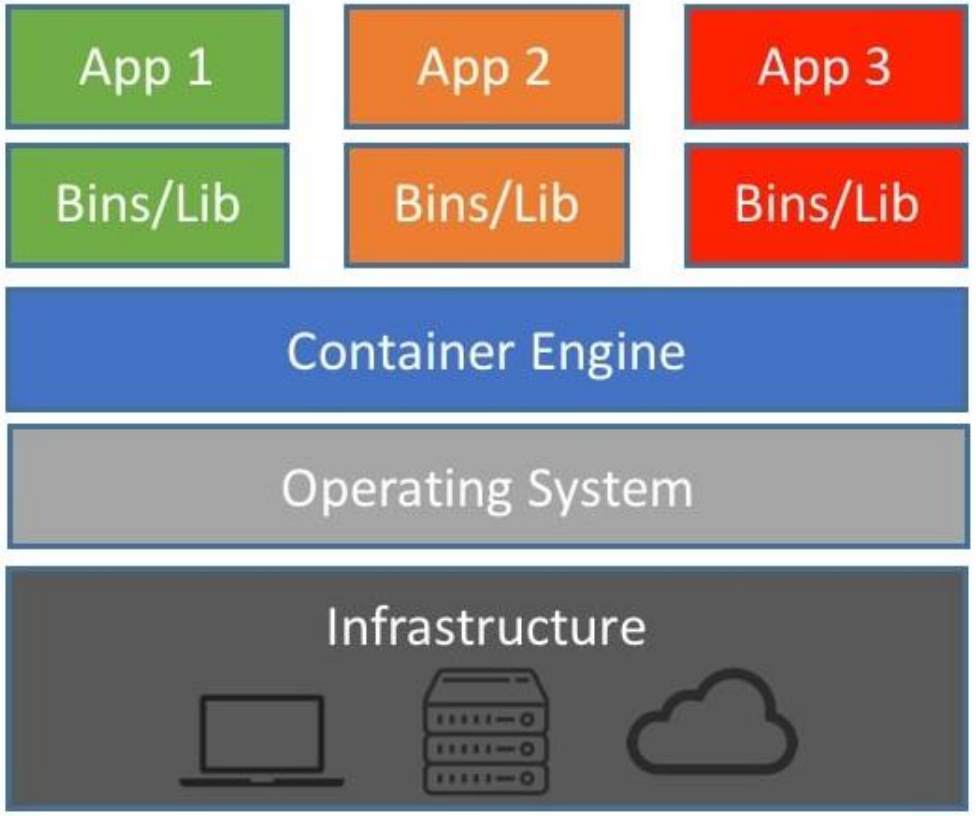
- May want to provide multiple OSes, but can do so with multiple physical machines

- 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)

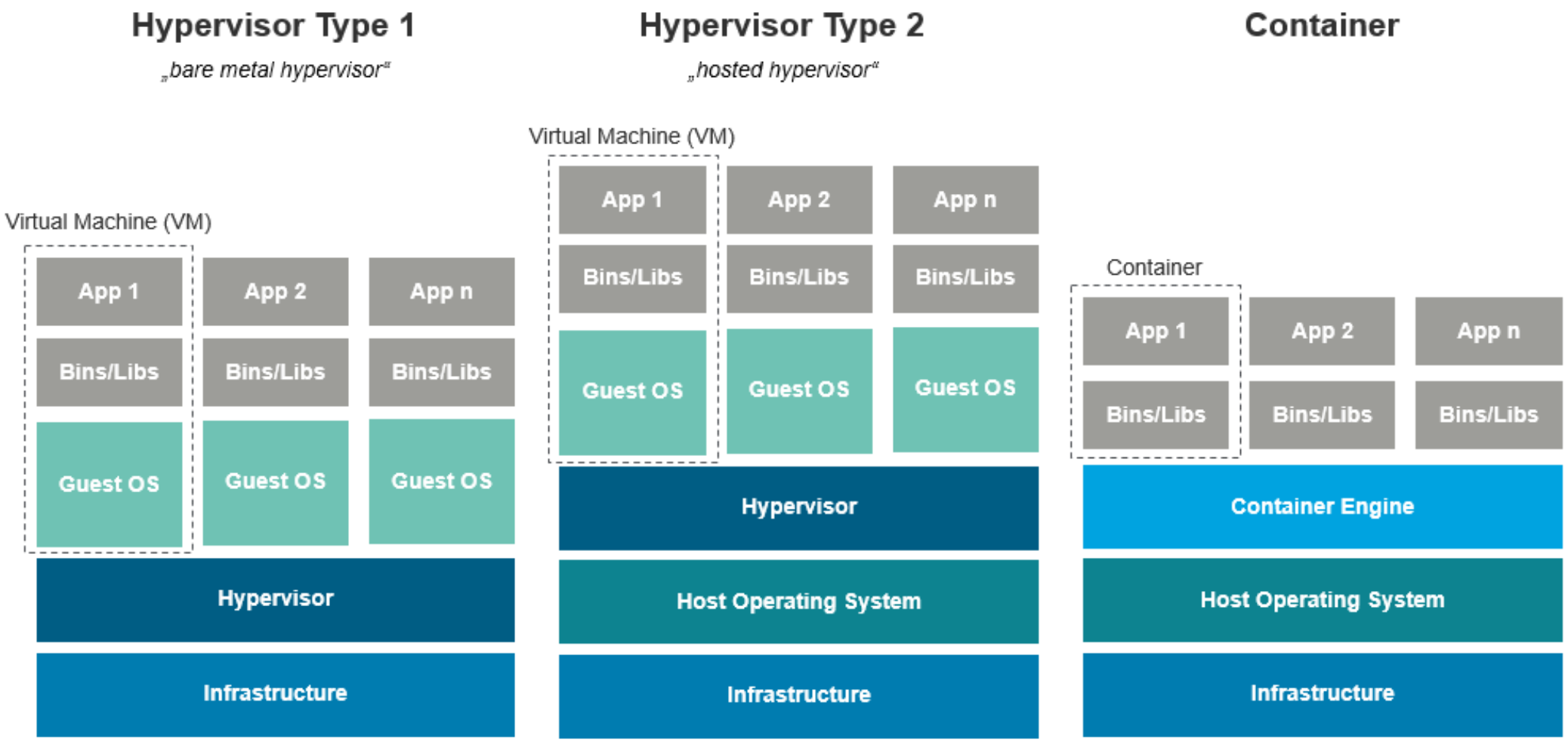
Containers

- 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

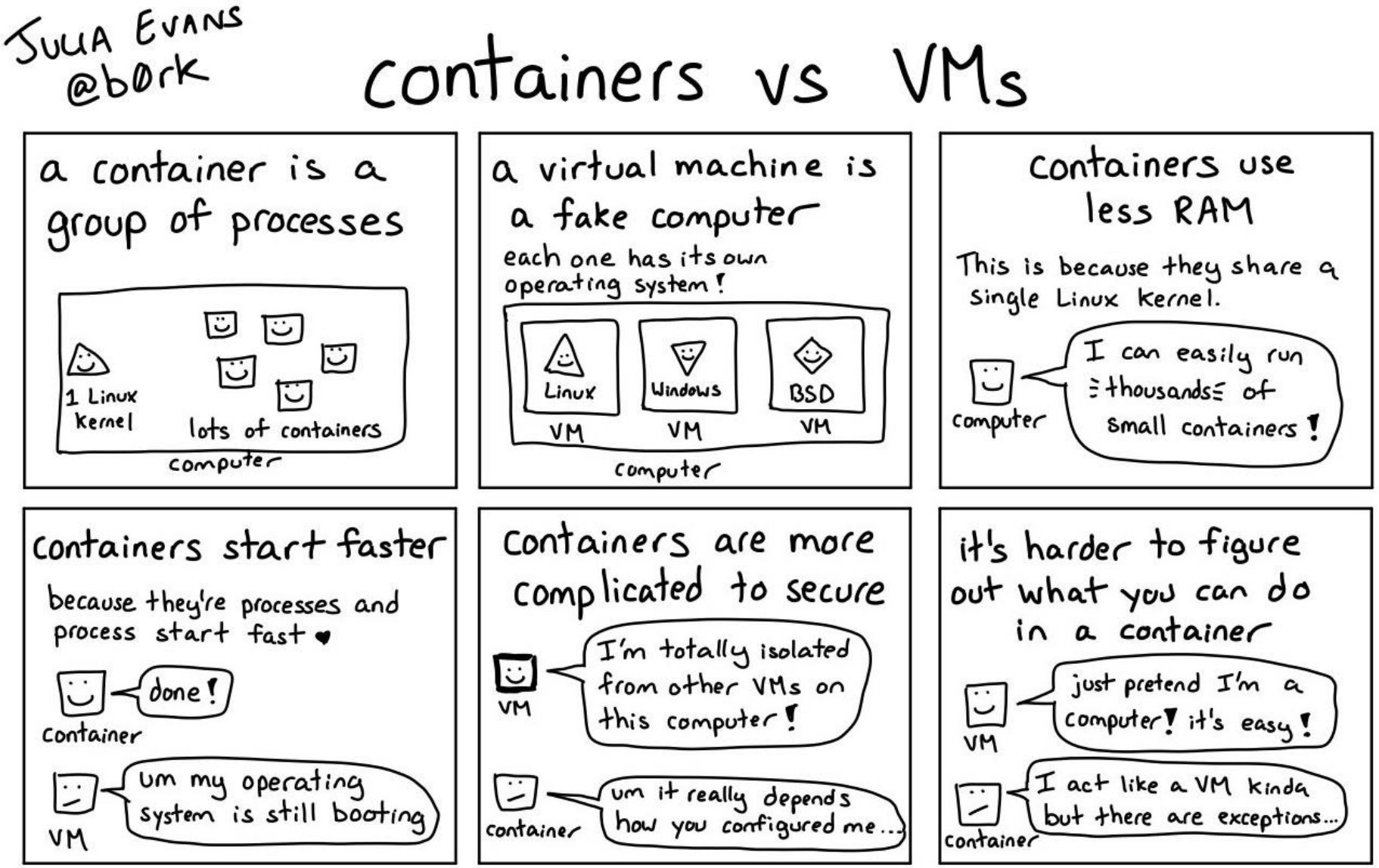


Containers

Visual comparison of Hypervisors and Containers



Bonus explanation



https://twitter.com/b0rk/status/1237744128450072578

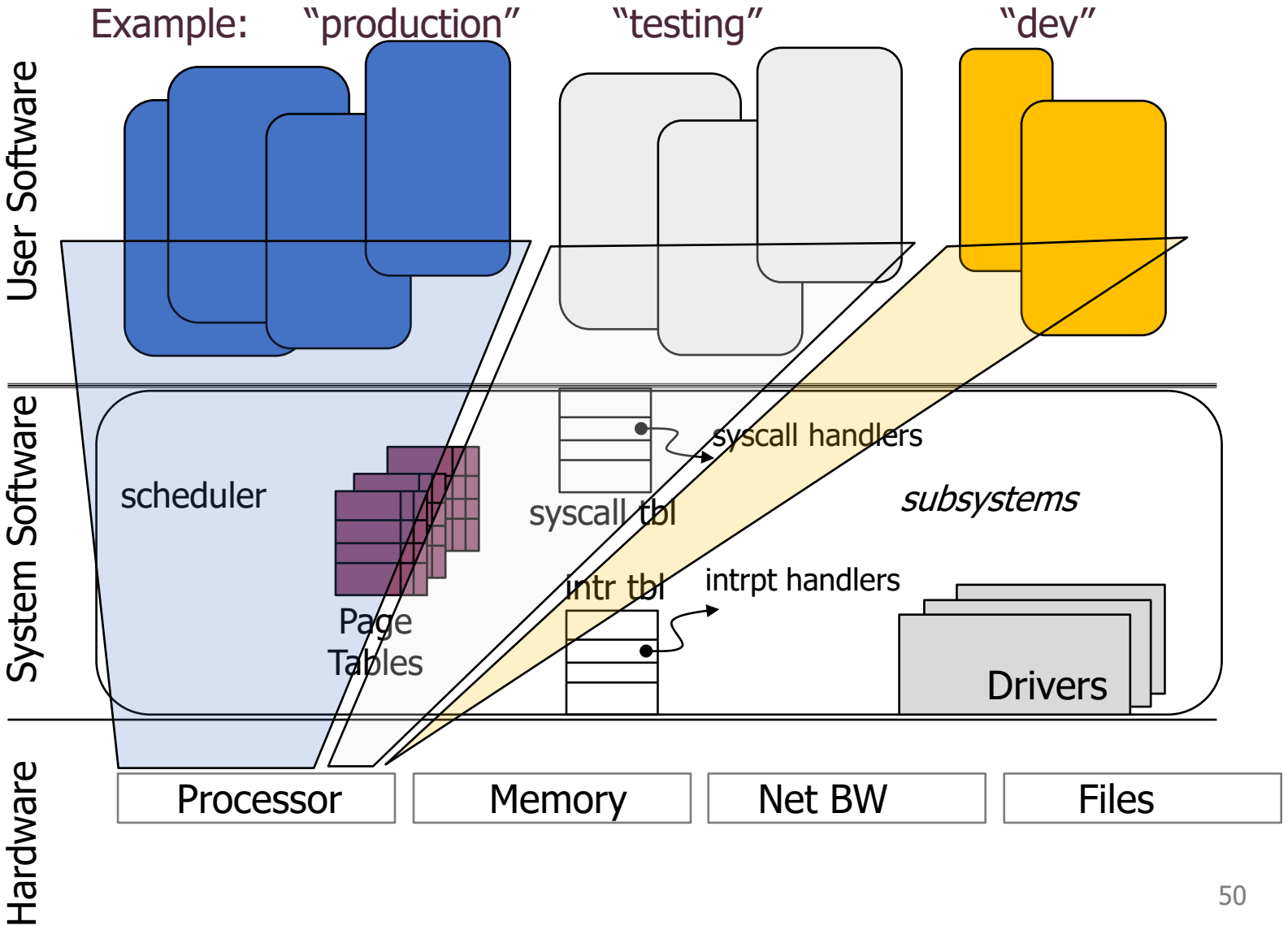
How to implement containers

- 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

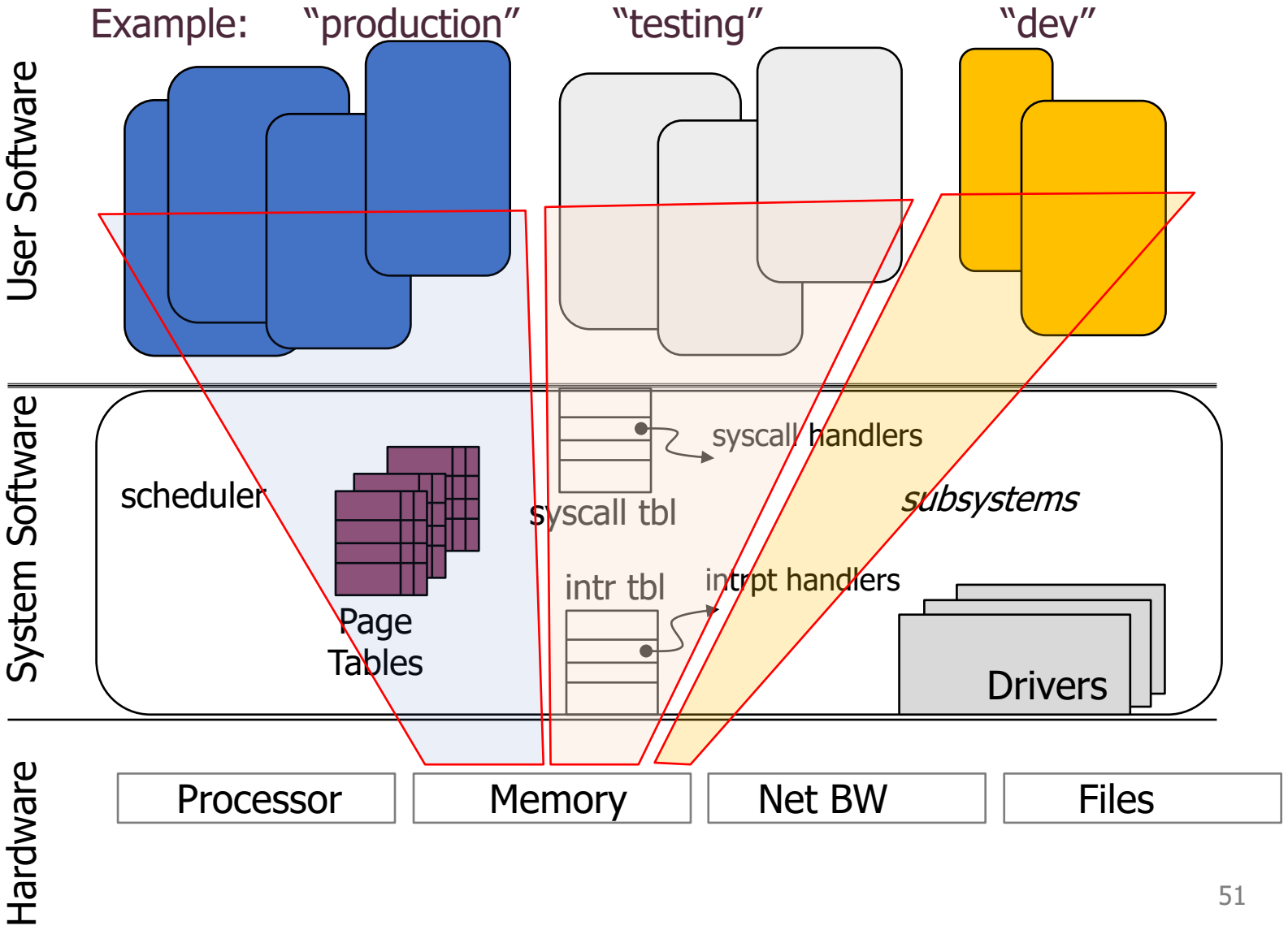
Linux cgroups (control groups)

- Collection of processes treated as a group for resource allocation
- Provider greater performance isolation between cgroups than between processes
  - Firm limits per group



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



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
- Still needs some glue to hold it all together (e.g., Docker!):
  - Specification of what resources the container should get
  - Plus what applications/libraries it contains
  - Filesystem management

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

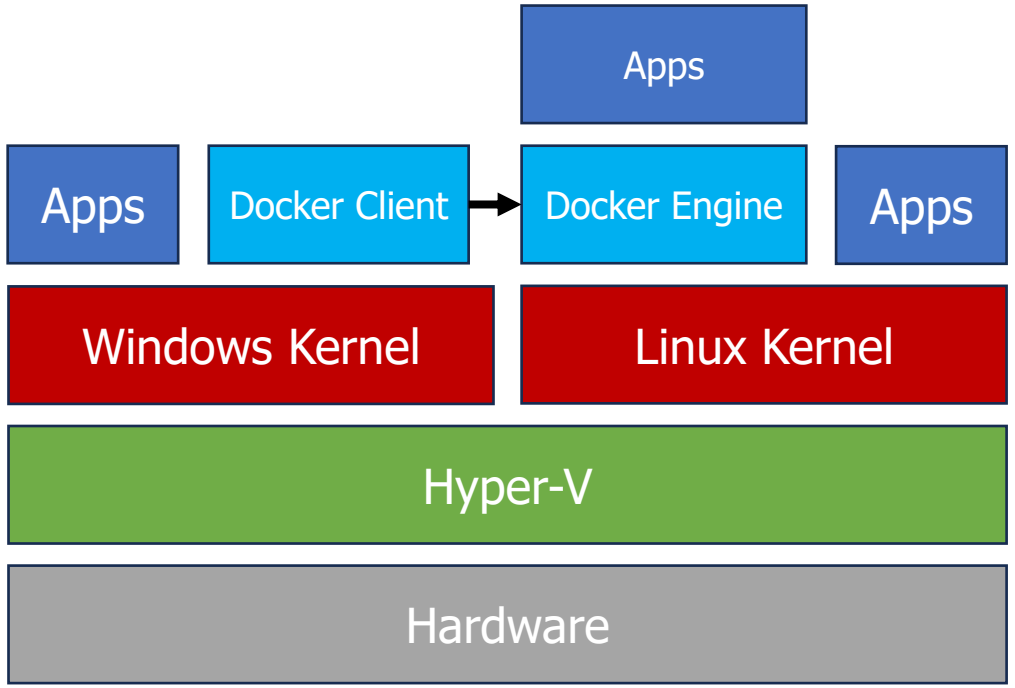
- Environments are hard to set up
  - Often the hardest part of starting software development
  - Containerized applications encapsulate requirements
  - Can be run on any system that has the same kernel it was built for

- 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 specific version of QEMU, with all the proper libraries

Docker runs on Linux

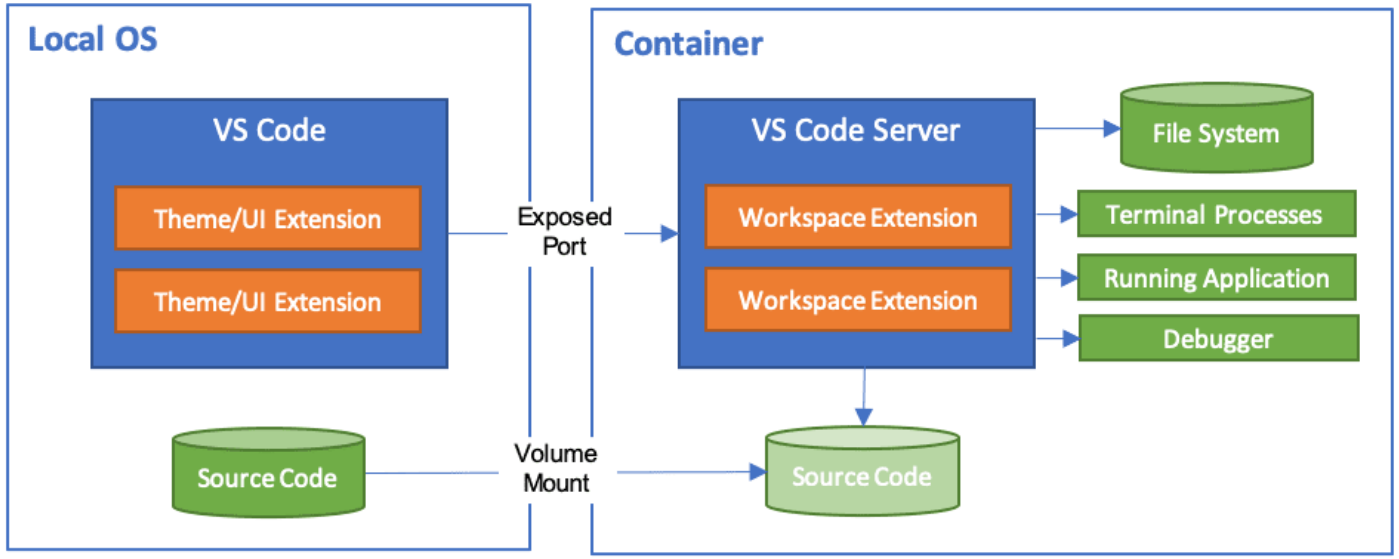
- To make it work on Windows, Docker uses a virtual machine running Linux
  - And runs your containers in there
- Typically, the virtual machine is WSL2
- Similarly Docker on MacOS starts a Linux VM



Editing code within a container environment

- Devcontainers provide a container environment for editing/testing code
  - IDEs can start them and run within them based on a configuration script

- This is a silly amount of translation on top of existing OS
  - Container -> Virtual Machine -> OS (lots of overhead)
  - BUT, most code is still running natively on the processor



An example of this I tried for Tock: https://github.com/brghena/tock-devcontainer-test/tree/main

Problem: attaching USB devices to docker containers is not well supported

Outline

- Virtualization
- Approaches
  - Emulation
  - Hypervisors
  - Containers