Operating Systems
Lecture 11: Virtualization

Yonghao Lee

June 20, 2025

1 Introduction

Virtualization means decoupling software from hardware, and it represents one of the most significant advances in computer systems over the last 25 years.

1.1 Benefits for Organizations

For organizations, virtualization enables **server consolidation**, which reduces costs for:

- Hardware acquisition
- Cooling systems
- Electricity consumption
- Maintenance overhead

Before virtualization, the typical deployment model was **one service per server**: web services ran on dedicated web servers, mail services on dedicated mail servers, and so forth. Virtualization allows multiple services to run on a single physical server, with each service running in its own virtual environment with its own operating system.

1.2 Benefits for Users

For end users, virtualization means the ability to run multiple operating systems simultaneously on a single machine, enabling greater flexibility and resource utilization.

1.3 What is Virtualization?

At its core, virtualization works similarly to how an operating system virtualizes hardware resources for processes. The OS creates the illusion that more resources are available than physically exist, with each application believing it has exclusive access to the entire machine.

Definition 1.1: ***Virtualization*** *is the decoupling of software from physical hardware limitations through a level of indirection. It hides the constrained physical reality while providing exactly the same interfaces, but implemented differently.*

Note: *Virtualization differs from abstraction in a key way: abstraction provides a different kind of interface or object, while virtualization provides the same object but implemented through different underlying mechanisms.*

1.4 Processes as Virtual Machines

A process is a virtual machine as applications see it.

- **CPU** - but only non-privileged instructions
  - Applications get CPU access but cannot execute privileged operations directly

  - The OS mediates access to system-level functions through system calls

- **Virtual memory** (abstraction of physical RAM)
  - Each process sees its own private, contiguous address space
  - The OS maps virtual addresses to physical memory behind the scenes
  - Provides isolation between processes

- **Also new abstractions supported by the OS**
  - e.g. a file system - provides persistent storage abstraction
  - Network sockets, process management, device drivers
  - So actually an ”abstract virtual machine”
  - The OS doesn’t just virtualize hardware - it adds entirely new interfaces

1.5 Real Virtual Machines

Real virtual machines virtualize the entire physical hardware as the operating system sees it. This means providing:

- **Complete CPU virtualization** - including all instructions, both privileged and non-privileged
- **Full physical memory access** - with complete address mapping capabilities
- **Control over all peripheral devices** - direct hardware device management

This comprehensive virtualization enables running a complete, unmodified operating system on top of the virtual machine. The guest OS believes it has direct access to real hardware.

1.6 From Physical to Virtual Machine

Traditional Computing Model: Previously, we had applications running above the OS, with hardware below it, and the OS mediating between the two layers.

Virtual Machine Model: With VMs, we have multiple operating systems with a hypervisor positioned between them and the hardware. The operating systems above the hypervisor believe they have access to real hardware, yet they only communicate with virtual hardware.

The Hypervisor/VMM: The hypervisor (Virtual Machine Monitor) is an extra level of indirection that decouples hardware from the operating systems. It provides:

- **Multiplexing** - allows multiple OSs to share physical resources
- **Strong isolation** - prevents guest OSs from interfering with each other
- **Resource management** - manages physical resources in front of the OSs it controls

Therefore, it performs functions similar to an operating system, but manages OSs instead of applications.

1.7 Virtualization with Multiplexing

Virtualization with multiplexing involves taking one physical resource and dividing it among multiple users or processes. This approach combines two key techniques: **multiplexing** (resource sharing) and **abstraction** (simplified interfaces).

In virtual memory systems, this concept is clearly demonstrated. The physical RAM contains multiple processes, where each process is allocated a portion of the available frames. The operating system manages this allocation dynamically, allowing efficient sharing of the physical memory resource.

This approach achieves both multiplexing and abstraction simultaneously:

- **Multiplexing**: Physical RAM is shared among different processes
- **Abstraction**: Each process perceives it has its own private, contiguous virtual address space

The same pattern applies to other system resources, such as CPU scheduling (multiplexing processor time) and disk partitions, which make a single disk appear as multiple smaller disks, each providing independent block storage.

1.8 Virtualization with Aggregation

This approach creates a virtual device using multiple physical devices. Rather than dividing one resource, we combine several smaller hardware components to give users the impression that they are operating on a single, larger virtual hardware system.

A prime example is RAID (Redundant Array of Independent Disks), which aggregates multiple physical disks. The data is replicated internally to improve reliability, while the interface remains conventional block storage. Any file system that can be stored on a conventional disk can be stored on RAID instead, with the underlying complexity hidden from the user.

1.9 Benefits of Virtualization

1. **OS diversity**: Run multiple operating systems on the same physical machine
2. **Server Consolidation**: Reduce the number of physical servers by combining multiple services onto fewer machines
3. **Security**: Isolation of VMs provides containment; enables monitoring of system behavior for intrusion detection and malware analysis
4. **Availability**: VMs can run anywhere and benefit from live migration, allowing seamless movement between physical hosts without downtime
5. **Foundation of Cloud Computing**: Virtualization enables cloud providers to efficiently share physical infrastructure among multiple customers. Each customer receives isolated virtual resources (VMs, storage, networks) while the provider maximizes hardware utilization. This makes on-demand resource allocation, scalability, and multi-tenancy possible.

1.10 Server Consolidation in Depth

Servers often run a single major application or service, it provides strong isolation between services and instances, but inefficient and inflexible, in using of hardware and energy, for example, some are idle while others are overloaded.

Virtualization facilitates consolidation: multiple logical servers on a shared physical infrastructure, which gives many benefits like saves hardware and energy, disaster recovery and etc.

We consolidate many services into a fewer number of machines when the workload is low, and as demand for a particular services increases, we migrate some VMs to other machines to run that service.

We can build a data center with fewer total resources, since they are used as needed instead of dedicated to a single service.

1.11 VM Workload Multiplexing

VM workload multiplexing enables efficient server consolidation by combining multiple virtual machines on a single physical server. This approach takes advantage of the fact that different workloads typically peak at different times.

The total server capacity required is less than the sum of individual VM capacities ($s_1 < s_2 + s_3$), because:

- Different VMs reach peak usage at different times
- Combined workloads create a smoother, more predictable demand pattern
- Statistical multiplexing reduces the likelihood of simultaneous peak demands

1.12 Other Uses of Virtualization

1. **Support for legacy applications**: Bring up a VM emulating old systems to run outdated software that cannot run on modern hardware or operating systems
2. **Low-level software development**: If software (e.g., drivers) crashes, it might crash the VM but not the whole system. A special case is OS development, where snapshots allow quick recovery from system crashes during kernel development
3. **Security**: Provides isolation for malware analysis and secure execution of untrusted code in contained environments
4. **Encapsulation**

1.13 VM Encapsulation

VM encapsulation captures the entire virtual machine state within files, including the operating system, applications, data, memory, and device state.

Key capabilities:

- **Snapshots and clones**: Capture VM state on-the-fly and restore to any point in time

- **Rapid system provisioning**: Deploy new systems by copying VM files
- **Backup and migration**: Move VMs between hosts for better load balancing and disaster recovery

2 Types of Hypervisors

2.1 Type I Hypervisor (Native/Bare-metal)

- **Direct hardware access**: Hypervisor runs directly on physical hardware
- **No host OS**: The hypervisor serves as the base operating system
- **Better performance**: Lower overhead and more efficient resource access
- **Examples**: VMware ESXi, Microsoft Hyper-V, Xen
- **Use case**: Enterprise servers, data centers

2.2 Type II Hypervisor (Hosted)

- **Runs on host OS**: Hypervisor is software running on top of an existing operating system
- **More layers**: Hardware $\rightarrow$ Host OS $\rightarrow$ Hypervisor $\rightarrow$ Guest OS
- **Higher overhead**: Additional software layers increase performance cost
- **Examples**: VMware Workstation, VirtualBox, Parallels
- **Use case**: Desktop virtualization, development, testing

2.3 Application Encapsulation/Container (Docker)

- **Shared kernel**: All containers share the host OS kernel
- **No guest OS**: Applications run directly on shared OS without separate operating systems
- **Lightweight**: Much lower overhead compared to full VMs
- **Process-level isolation**: Containers isolate applications, not entire operating systems
- **Examples**: Docker, Kubernetes pods
- **Use case**: Microservices, application deployment, cloud-native applications

Key Differences:

- **Resource overhead**: Type I $<$ Type II $<$ Full VMs; Containers are lightest
- **Isolation level**: Full OS isolation (Type I/II) vs. Process isolation (containers)
- **Startup time**: Containers start in seconds, VMs take minutes
- **Use cases**: Containers for applications, VMs for complete system isolation

3 Virtualization Implementation

3.1 Virtualization Basics

The hypervisor functions like an OS kernel, while VMs operate like processes. The hypervisor controls everything, including scheduling VMs, allocating resources to VMs, and multiplexing I/O devices among them.

However, the guest operating systems within each VM believe they run directly on the hardware and control everything. The hypervisor must maintain this illusion.

This deception is essential for virtualization to work, now how to fake it?

3.2 How to Make the Illusion

We have:

1. Trap and Emulate
2. Binary Translation
3. Paravirtualization
4. Hardware Assistance

We will focus on Type-1 hypervisors though most of this applies to other types too.

3.3 Trap and Emulate

Only the hypervisor runs on bare metal in privileged mode (ring 0). The VMs run directly on the hardware in user mode, with both user applications and the VM's guest OS executing in ring 3.

Process flow:

- When applications make system calls, they trap into the guest OS
- The guest OS handles the system call in user mode (ring 3)
- When the guest OS attempts to execute privileged instructions, it traps to the hypervisor (General Protection Fault)
- The hypervisor executes the privileged instructions on the guest OS's behalf
- Results are returned to the guest OS, maintaining the illusion of direct hardware control

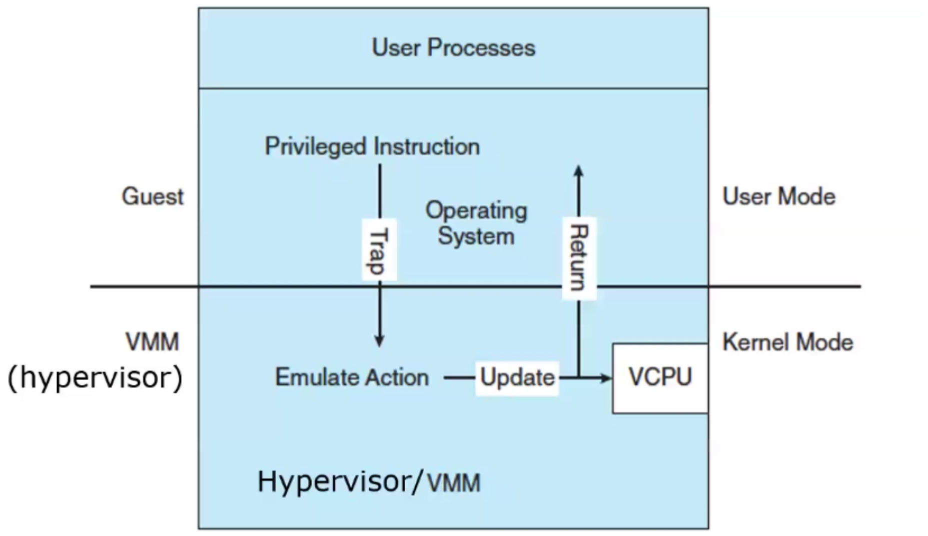


Figure 1: Trap and Emulate

Requirements and Limitations: This approach requires that all sensitive instructions be privileged instructions. This assumption was not satisfied in Intel x86 architecture during the 1980s, where some sensitive instructions executed directly without trapping. Modern architectures now satisfy this requirement, making trap-and-emulate viable. The approach works effectively when the number of traps is manageable and the architecture properly supports virtualization.

3.4 Example: Virtualizing the CLI (Clear Interrupt) Instruction

When a guest OS executes the `CLI` instruction to disable interrupts, the hypervisor must maintain the illusion while preserving system-wide interrupt handling.

Process flow:

1. **Guest OS intent**: Executes `CLI` to temporarily disable interrupts for critical sections
2. **Trap occurs**: `CLI` is privileged, so it causes a General Protection Fault, transferring control to the hypervisor
3. **Hypervisor response**:
   - Sets a **virtual interrupt flag** to "disabled" for that specific VM
   - Does **not** disable physical interrupts (which would affect other VMs and the hypervisor)
4. **Physical interrupt handling**:
   - Hardware continues generating interrupts normally
   - Hypervisor receives all physical interrupts
   - For each interrupt, hypervisor checks the target VM's virtual interrupt flag

5. **Interrupt delivery decision**:
   - If VM's virtual interrupts are disabled → hypervisor **queues** the interrupt
   - If VM's virtual interrupts are enabled → hypervisor **delivers** the interrupt immediately
6. **Re-enabling**: When guest OS executes `STI`, hypervisor clears the virtual interrupt flag and delivers all queued interrupts

Result: The guest OS receives exactly the expected behavior (no interrupts during the disabled period, all interrupts delivered after re-enabling) while the hypervisor maintains control over physical interrupt handling and isolation between VMs.

3.5 Sensitive-but-Unprivileged Instructions (Red Pills)

"You take the red pill... you stay in Wonderland, and I show you how deep the rabbit hole goes." [The Matrix]

These are unprivileged x86 (32-bit) instructions that can reveal virtualization to the guest OS:

Characteristics:

- Read data that can only be set by privileged instructions
- Behave differently in privileged vs. unprivileged mode
- Do not cause traps when executed in user mode (ring 3)

Examples:

PUSHFD instruction:

- Writes the Interrupt-Enable Flag (IF) to stack
- Problem: Reveals actual physical interrupt state vs. virtual interrupt state
- Guest OS can discover that its virtual interrupt control doesn't match reality

POPFD instruction:

- Reads the Interrupt-Enable Flag (IF) into the EFLAGS register
- Behavior depends on privilege level:
  - Ring 0: Actually modifies the interrupt flag
  - Ring 3: Silently ignores interrupt flag changes
- Guest OS can test whether it truly has kernel privileges

Impact: These instructions break the trap-and-emulate model because they execute directly without hypervisor intervention, allowing guest software to detect virtualization and potentially behave differently or refuse to run.

3.6 Binary Translation

As a solution, we use dynamic binary translation when not all sensitive instructions are privileged.

Process:

- Translates all VM code blocks into safe code before execution
- **Normal instructions**: Uses identity translation (unchanged)
- **Sensitive instructions**: Replaces with hypercalls (calls to the hypervisor, similar to system calls)
- Stores translated code in a **code cache**
- Future executions of the same code use the safe cached version

Benefits: This approach handles both privileged instructions (which trap) and sensitive-but-unprivileged instructions (which are replaced), ensuring complete virtualization support regardless of hardware limitations.

3.7 Hardware Assistance for Type 1 AKA "Ring -1"

Modern processors provide hardware support to solve the virtualization challenges of early x86 architecture.

Popek/Goldberg Theorem Compliance:

- All sensitive operations must be privileged to enable proper virtualization
- Hardware ensures all sensitive operations cause traps that can be caught
- Available in x64 architecture (2003-2004)

Technologies: Intel VT-x and AMD SVM (2006)

Hardware Virtualization Features:

- **New privilege mode** (unofficially "Ring -1"): Hypervisor runs at higher privilege than ring 0, while guest OS (guest kernel) runs in native ring 0
- **New instructions:**
  - VMRUN (AMD)
  - VMLAUNCH + VMRESUME (Intel)
- **New exception/transition:** #VMEXIT / VM-Exit for automatic traps to hypervisor
- **New data structure:** VM control block (VMCS) for hardware-managed VM state
- **Complete trap coverage:** Can trap to hypervisor for every sensitive operation

Benefits: Eliminates the need for binary translation, provides better performance, ensures complete virtualization support, and enables guest OS to run with full ring 0 privileges while maintaining hypervisor control through ring -1.

4 The Problem with Virtual Memory in Virtualization

4.1 Core Challenge

Even with hardware virtualization support, virtualizing the MMU is a challenge because the guest OS thinks it owns physical memory and the page table.

4.2 Guest OS Expectations

The guest operating system expects to:

- Modify (add/remove) page → frame mappings
- Perform context switches (load new page tables)
- Have direct control over memory management

4.3 The Problem

Critical Issue: If we don't virtualize the MMU, then the guest OS can wreak havoc in the real RAM by accessing memory it shouldn't have access to.

5 Solution 1: Shadow Tables

5.1 Basic Concept

Virtualize the MMU by pointing it to a real page table (”shadow table”) while making the guest OS think it's still managing memory via its own (guest) page table.

5.2 How Shadow Tables Work

1. **Guest Perspective:** Guest OS creates and manages what it thinks is its page table
2. **Hypervisor Role:** Maintains a shadow page table that maps guest OS pages to real memory frames managed by the hypervisor
3. **Address Translation:**

$$\text{Guest thinks:} \quad \text{Virtual Address} \rightarrow \text{Guest Physical Address} \tag{1}$$
$$\text{Shadow does:} \quad \text{Virtual Address} \rightarrow \text{Real Physical Address} \tag{2}$$

5.3 Key Properties

- **Same structure:** Shadow table has the same page table structure as the guest page table
- **Virtual-to-real mapping:** Hypervisor maintains mapping from ”virtual” frames to real frames
- **Read-only guest tables:** Pages containing the guest page table are marked read-only

5.4 CR3 Register Virtualization

The CR3 (page table root register) access must also be virtualized:

- **Guest perspective:** Guest sees the frame of its page table root (virtualized)
- **Reality:** Actual CR3 points at the shadow table

5.5 The Read-Only Mechanism

When the guest OS modifies its page table (e.g., add a page mapping):

1. Guest writes to a page in the page table (read-only) $\rightarrow$ triggers trap
2. Traps to hypervisor, which updates the shadow page table accordingly
3. Guest OS remains unaware of the shadow table's existence

5.6 Implementation Notes

- Other implementation alternatives exist beyond shadow tables
- Shadow tables represent one approach to solving the MMU virtualization challenge

6 Pros and Cons of Shadow Tables

6.1 Advantages

- **Hardware speed memory access:** As long as there are no page faults, memory access operates at hardware speed

6.2 Disadvantages

- **Horribly slow when we have page faults:** Performance degrades significantly during page fault handling
- **Must trap every guest page table update:** We must trap every time the guest page table is updated (access violation = page fault)
- **All page faults become VM exits:** Every page fault is now a VM exit event, which is very slow

Performance Impact: The major weakness of shadow tables is that any page table modification by the guest OS triggers expensive VM exits, making memory management operations significantly slower.

7 Solution 2: Second Level Address Translation (SLAT)

7.1 The Need for a Better Solution

As a result, shadow tables are not so good. We need to introduce Second Level Address Translation (SLAT) or "Nested Paging" to solve the problem.

7.2 Core Idea

The idea is to offload virtual frame to real frame mapping from the hypervisor (which is software) to the hardware.

This way:

- No shadow page table is needed
- The page table managed by guest OS works normally
- The page table translation ends in the virtual frame
- A second page table-like translation from virtual frame to physical frame is visible to hypervisor only
- This second page table is managed by the hypervisor

7.3 SLAT with Page Table Hierarchies

Important Detail: The original page table is based on frame numbers (pointing down the table hierarchy), so second level translation is needed for each of them.

7.4 Understanding SLAT Frame Translation

7.4.1 How Page Tables Work

Page tables are hierarchical structures where:

- Each page table entry contains a **frame number**
- These frame numbers **point down the table hierarchy** to the next level
- The hierarchy continues until reaching the final data page

7.4.2 What SLAT Actually Does

SLAT acts as a **frame number translator** during the page table walk:

SLAT's Job: Translate every frame number that appears in page table entries from virtual frame numbers (what guest thinks) to real physical frame numbers (actual hardware locations).

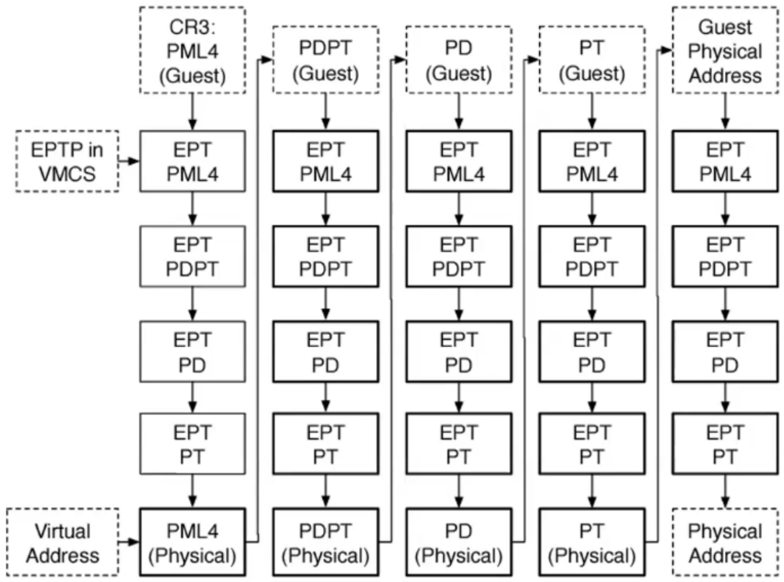


Figure 2: The Grim Reality of SLAT

The core challenge is that the Guest Operating System (running inside the VM) manages its own set of page tables and believes it is interacting directly with physical memory. However, the addresses it produces, known as Guest Physical Addresses (GPAs), are not the true hardware addresses. The hypervisor must translate these GPAs into Host Physical Addresses (HPAs) to ensure isolation and proper memory management. This two-layered translation is handled efficiently by hardware features like Intel's Extended Page Tables (EPT) or AMD's Rapid Virtualization Indexing (RVI).

The "grim reality" depicted is the worst-case scenario where every memory lookup requires a full walk through both the guest's page tables and the hypervisor's EPTs, leading to a large number of memory accesses for a single translation.

The following is a sequential walk-through of the address translation process as shown in the diagram.

1. **Starting Point (Guest Virtual Address):** The process begins when a program in the guest OS needs to access a memory location. The CPU starts with the value in the guest's `CR3` register, which contains the *guest physical address* of the top-level guest page table: the Page Map Level 4 (PML4) table.

2. **Finding the Guest PML4:** Before it can even start the guest's translation, the hardware must find the *real* location of the guest's PML4 table. It takes the *guest physical address* from the `CR3` register and translates it by performing a full, four-level walk through the hypervisor's Extended Page Tables (EPTs). This finds the *host physical address* of the guest's PML4.

3. **Guest PML4 to Guest PDPT:** With the real location of the guest PML4 known, the CPU uses the first part of the original virtual address to find an entry inside it. This entry contains the *guest physical address* of the next-level table, the Page Directory Pointer Table (PDPT).

4. **Translating the PDPT Address:** This *guest physical address* of the PDPT must now be translated. The hardware again performs a full, four-level walk through the EPTs to convert this guest physical address into the *host physical address* of the guest's PDPT.
5. **Repeating the Process:** This pattern of "look up in guest table, then translate the resulting pointer via EPT" continues down the entire hierarchy:
   - The CPU uses the real location of the guest's PDPT to find the *guest physical address* of the Page Directory (PD).
   - It then translates this address through the EPTs to find the *host physical address* of the PD.
   - The CPU uses the real location of the guest's PD to find the *guest physical address* of the Page Table (PT).
   - It translates this address through the EPTs to find the *host physical address* of the PT.
6. **Final Translation:** Finally, the CPU uses the real location of the guest's PT to find the *guest physical address* of the actual data page the application wants. For the last time, the hardware performs a full EPT walk to translate this into the final **Host Physical Address**.

Only after this entire nested sequence is complete does the CPU have the true hardware address required to access the data in RAM.

This is actually not that bad, even though we have multiple second level translation for each virtual to real translation, but this is in hardware speed and in practice, TLB is used for the rescue, and this way, there is no need for expensive VM exits.

8 Containers and Processes

- Processes can be thought of as virtual machines from the perspective of an application.
- Containers are very similar in this regard.
- Another way to view containers is as a method of packaging a set of processes that have their own isolated OS resources.
- This isolation and control is achieved through two key Linux concepts: **namespaces** and **cgroups**.

9 Understanding Containers

- Containers are considered a middle-ground technology between traditional processes and full virtual machines (VMs).
  - For example, different processes on an OS share the same file system, whereas containers do not.

- The core idea of containers is to virtualize the Operating System itself, rather than the underlying hardware.
- Each container has its own file system, memory, and other OS resources, making it feel like it is running completely by itself.
- A container typically packages everything an application needs to run:
  - The application code
  - Dependencies
  - Libraries
  - Binaries
  - Configuration files

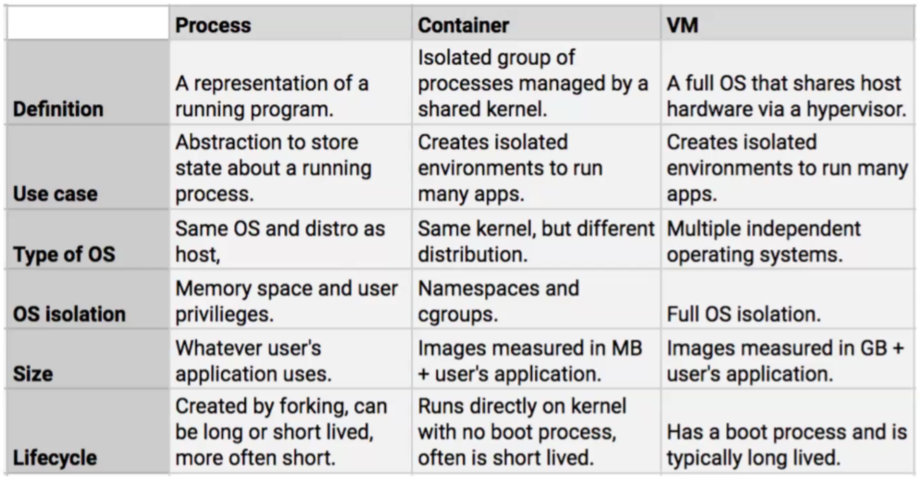
10 The Role of Namespaces and Cgroups

10.1 Namespaces: Providing Isolation

- A **namespace** is an abstraction that wraps a global system resource. This makes it appear to processes within that namespace that they have their own private instance of that resource.
- By definition, a standard process already has its own address space and restricted privileges.
- Containers extend this idea by having their own namespaces.
  - For example, a container with its own PID (Process ID) namespace will only see its own processes. The first process inside it will have PID 1, which is a completely different process from PID 1 in another container.
- There are 7 defined namespaces: PID, User, Mount, Network, IPC, UTS, and cgroups.

10.2 Cgroups: Controlling Resources

- **Cgroups** (Control Groups) are used to manage and limit the resources that a container can use.
- They answer the question: ”How much of a resource should I give a container?”.
- This includes resources like CPU time, RAM, the number of file descriptors, and I/O bandwidth.



	Process	Container	VM
Definition	A representation of a running program.	Isolated group of processes managed by a shared kernel.	A full OS that shares host hardware via a hypervisor.
Use case	Abstraction to store state about a running process.	Creates isolated environments to run many apps.	Creates isolated environments to run many apps.
Type of OS	Same OS and distro as host,	Same kernel, but different distribution.	Multiple independent operating systems.
OS isolation	Memory space and user privilieges.	Namespaces and cgroups.	Full OS isolation.
Size	Whatever user's application uses.	Images measured in MB + user's application.	Images measured in GB + user's application.
Lifecycle	Created by forking, can be long or short lived, more often short.	Runs directly on kernel with no boot process, often is short lived.	Has a boot process and is typically long lived.

Figure 3: Comparison

11 I/O Virtualization: Sharing Hardware with VMs

When multiple Virtual Machines (VMs) run on one computer, they all need to share physical hardware like network cards or USB ports. I/O Virtualization deals with how the hypervisor manages this sharing safely. The presentation shows three main ways to do this.

11.1 The Emulated Model (Slow but Compatible)

This is like having a universal translator. The VM doesn't talk to the real hardware, but to a ”fake” generic version created by the hypervisor.

- The hypervisor pretends to be a simple, common hardware device that every OS already knows how to use.
- When the VM tries to use this fake device, the hypervisor ”traps” the request.
- It then translates that request and performs the action on the real hardware for the VM.
- The hypervisor itself must have the real drivers for the physical hardware installed.

11.2 The Split / Paravirtualized Model (Faster and Aware)

In this model, the Guest OS knows it's a VM and cooperates with the hypervisor for better performance.

- This method uses paravirtualization, meaning the Guest OS is modified to work with the hypervisor.
- The Guest OS needs special drivers installed that allow it to talk directly to the hypervisor for I/O tasks.
- This is more efficient because it avoids the slow process of trapping and emulating hardware actions.

11.3 The Pass-through Model (Fastest but Dedicated)

This is the highest performance option, like giving a VM its own private hardware device.

- The Guest OS is allowed direct access to a physical I/O device.
- This is extremely fast but requires special hardware support (like Intel VT-d or AMD-Vi) to make sure the VM is isolated and secure.
- This means that a specific physical device is dedicated entirely to one VM and cannot be shared.

12 Understanding Paravirtualization

Paravirtualization is a virtualization technique where the Guest Operating System's source code is modified to make it aware that it is running inside a virtual environment.

12.1 The Core Concept

- The Guest OS is changed so that all calls to privileged instructions are replaced with direct calls to the hypervisor. These special, virtualization-aware calls are known as **hypercalls**.
- The goal of this modification is to reduce the number of performance-costly traps and remove sensitive instructions.

12.2 Advantages

- It is significantly more efficient to modify the source code to cooperate with the hypervisor than it is to emulate hardware instructions in real-time, as is done in binary translation.

12.3 Disadvantages and Requirements

- A major limitation is that you cannot run any unmodified Guest OS ”as is”.
- This technique requires having access to the source code of the Guest OS in order to make the necessary changes.
- The Guest OS must be recompiled specifically for each hypervisor it will run on.