

Enabling Technologies: Virtualization and Containerization

Virtualization

Virtualization allows running **multiple operating systems or applications** on a **single physical machine**. It creates a virtualization layer between the guest (virtual image, applications) and the host (physical hardware, storage, networking).

Characteristics of Virtualization

- **Increased security:**
 - The *virtual machine monitor (VMM)* or *hypervisor*, which is the component that allow communication between VMs and host machine, **controls and filters guest activity**, preventing harmful operations against host machine. Operations are executed only on VM which then execute them of host machine.
 - Resources from host machine can be hidden or protected.
 - *Examples:* Completely separated virtualized Java Virtual Machines (JVM) and file systems in hardware virtualization.

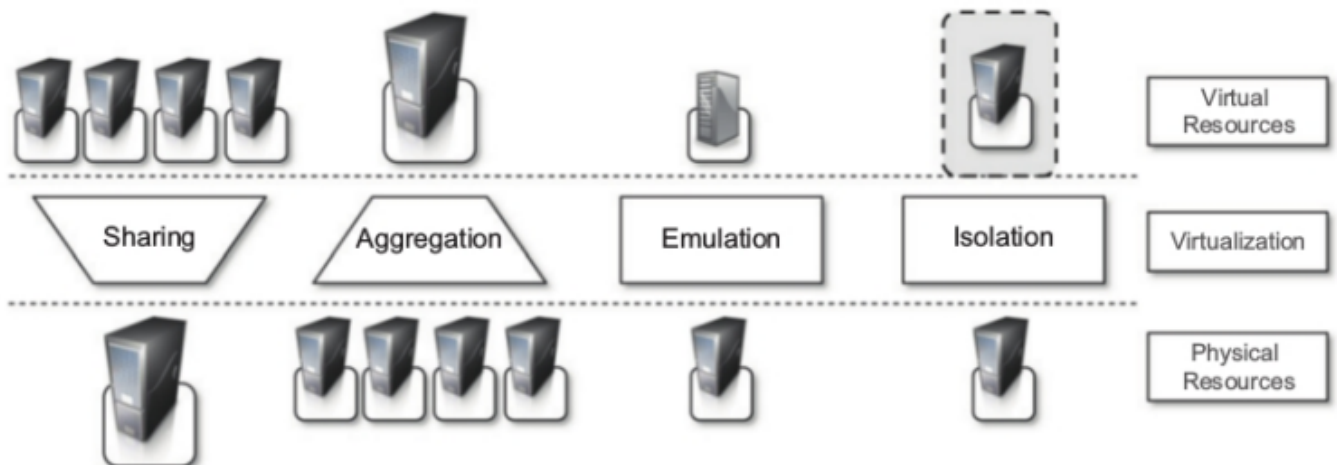


FIGURE 3.2

Functions enabled by managed execution.

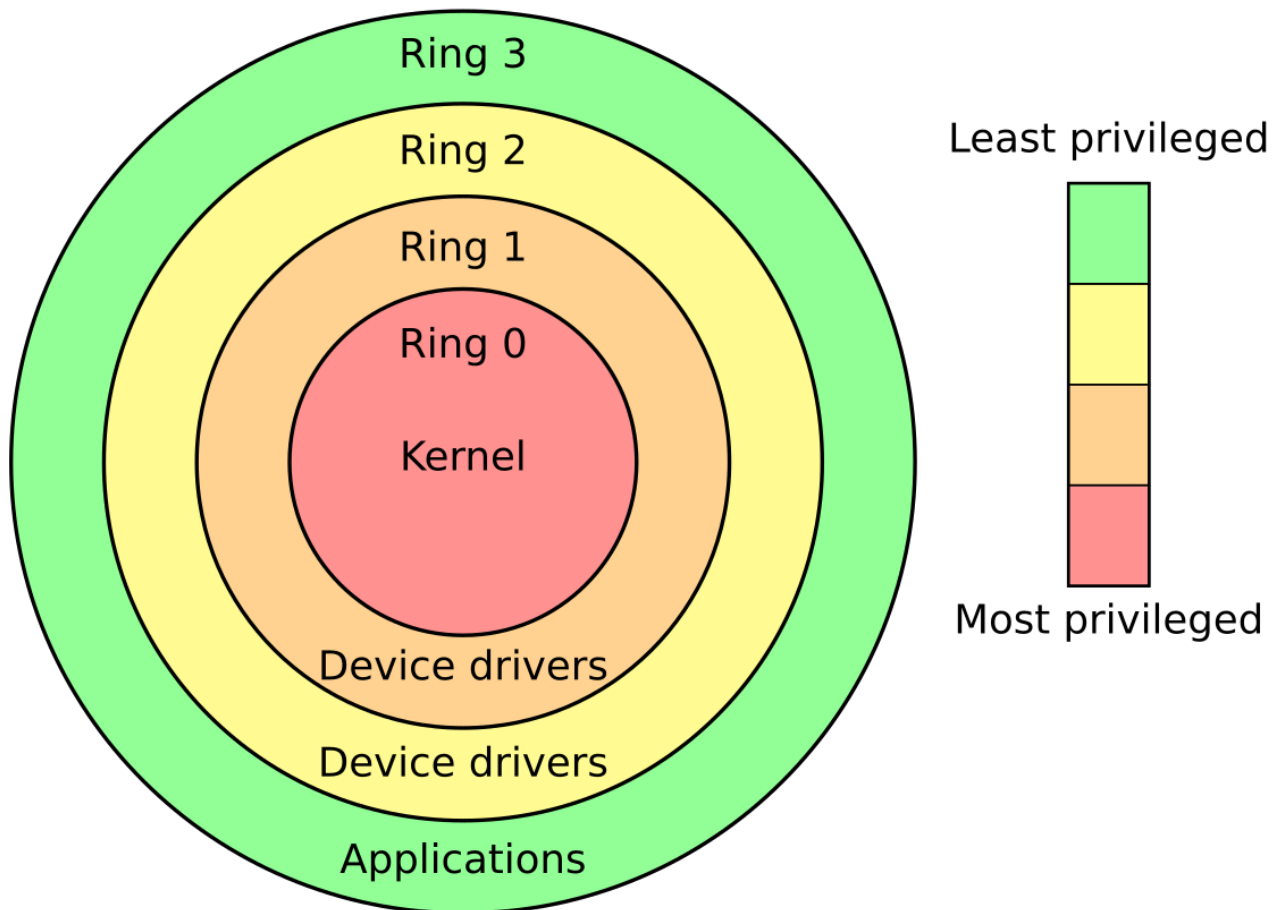
- **Managed execution:** Provides following resource capabilities:
 - **Sharing:** Creation of a separate computing environments within the same host by sharing same hardware resources.
 - **Aggregation:** Opposite of sharing, allow multiple host machine to be tied together in a single virtualized computing environment.
 - **Emulation:** Completely different environment w.r.t. to host environment can be emulated (Linux VM on Windows host machine).
 - **Isolation:** Provide to each guest a completely separate environment.

- **Portability:** Allows **transferring data and applications** between computing platforms because VMs are booted from **VM images** stored in specific disk formats (e.g., aki, ari, ami, vdi, vhd, vhdx, vmdk) that can be moved or converted.

Docker containers can run on any platform with the Docker engine.

Machine Reference Model

To understand how the different virtualization techniques work it is fundamental to recall the *layered structure of a computer system*.

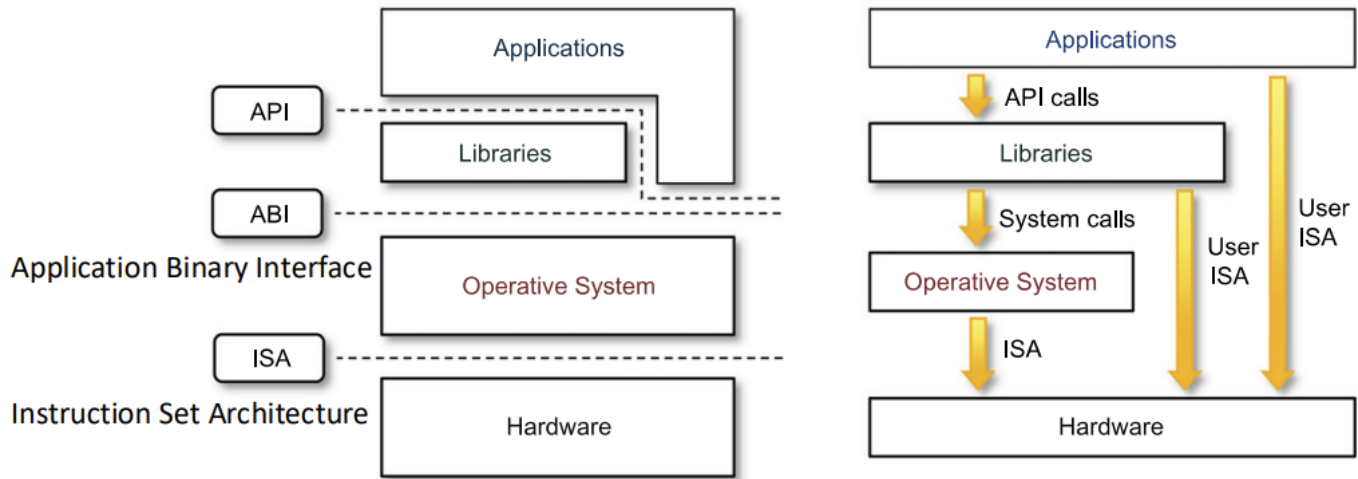


- **Ring 0: Most privileged** (*Supervisor/Kernel mode*).
- **Ring 3: Least privileged** (*User mode*).

Rings 1 and 2 are not used in modern systems.

There are **two types** of instructions:

- **Nonprivileged:** can be used **without interfering with other tasks** (e.g., arithmetic, floating-point).
- **Privileged:** instructions have restrictions and are for **sensitive operations** (e.g., I/O, changing CPU state).



- Computer systems have a **layered structure**:
 - **Hardware**
 - **Operating System**
 - **Libraries** and **Applications**.
- These layers communicate using **interfaces**:
 - **ISA - Infrastructure Set Architecture**: defines the instruction set for the hardware (CPU, Memory, ...) and is the interface between Hardware and Software. OS use ISA to interact with hardware and execute privileged instructions.
 - **ABI - Application Binary Interface**: defines low level interface between the OS and applications and libraries, which interact with OS using **System Calls**.
 - **API - Application Programming Interface**: used by applications to access libraries functionalities.

While the OS uses the **full, privileged ISA** to control hardware, user-level applications operate with a **restricted subset of the ISA** (non-privileged instructions).

They rely on the OS (via system calls) to perform operations that require privileged access to the hardware.

The thick yellow arrows emphasize how both the OS and Applications ultimately have their instructions executed by the Hardware, but through **different privilege levels and pathways**.

Hypervisor (Virtual Machine Monitor - VMM)

The **hypervisor runs in Supervisor/Kernel Mode**, so can run instruction with highest level of privilege, while a VM operating system, called *guest OS*, **runs in User Mode**.

The hypervisor controls how the VM use the hardware resources.

The **challenge** is to emulate/manage the CPU state for guest OS, requiring sensitive instructions to run in Supervisor Mode.

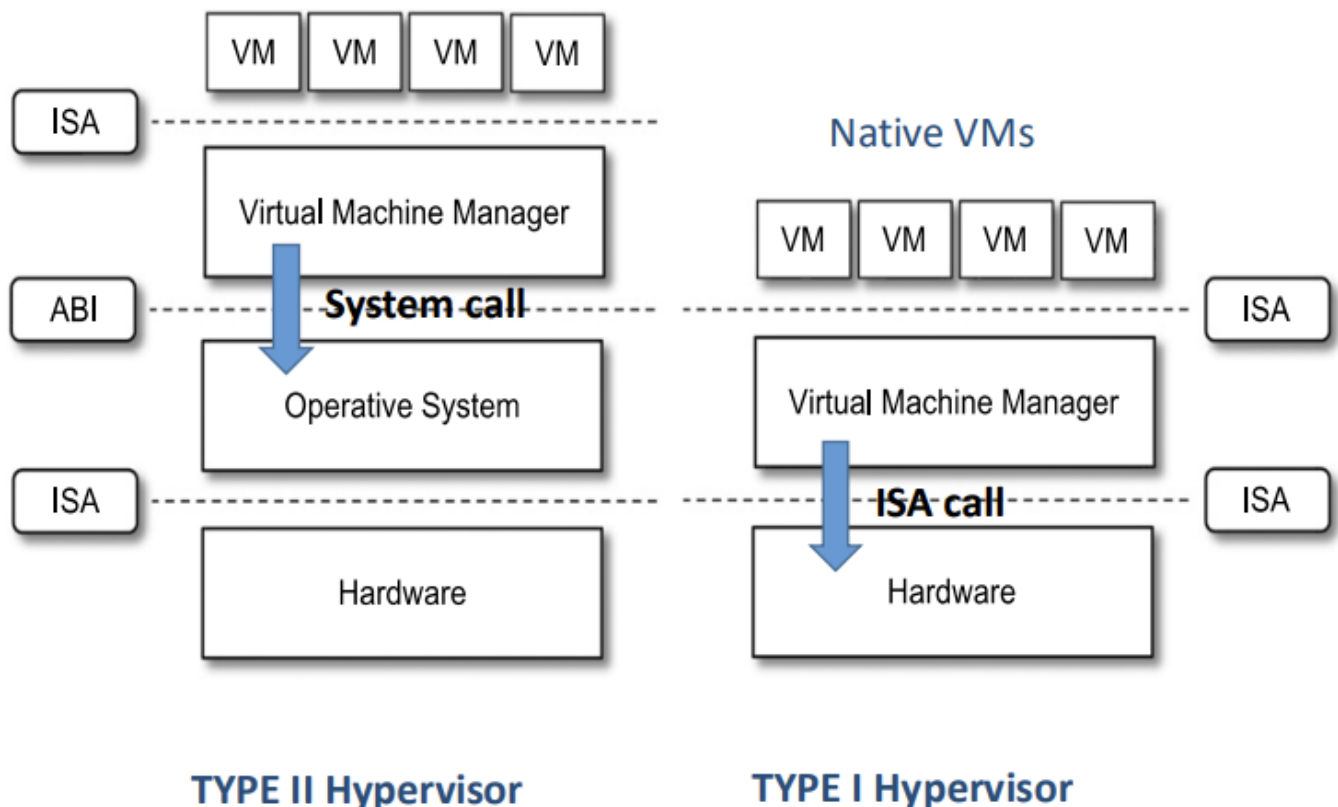
VMMs allow **several operating systems to run concurrently** on a single hardware platform; at the same time, VMMs **enforce isolation** among these systems, thus enhancing security, by ensuring that operations done by a guest OS does not affected other VM.

Original ISAs had sensitive instructions executable in user mode, preventing guest OS isolation. Intel VT and AMD-V redesigned instructions as privileged.

Set of **sufficient conditions** for a computer architecture to support virtualization and allow a VMM to operate efficiently are:

- **Equivalence:** Guest behavior should be the same as on physical hardware.
- **Resource control:** VMM has complete control of virtualized resources.
- **Efficiency:** Most instructions run without VMM intervention. (Popek and Goldberg requirements).

Hypervisor Types



- **Type II Hypervisor (Hosted VMs)**
 - A **standard OS** is installed directly on hardware, it manages hardware resources using ISA calls, so using privileged instructions.
 - Hypervisor is installed like an application on top of the OS, it interacts with the OS using **System Calls via ABI**. It does not execute privileged instruction but need to pass through OS to do it.
 - **Multiple VMs** can be managed by the Hypervisor, each VM believes it is interacting directly with hardware but this interaction is mediated by Hypervisor, which present ISA to VMs.
 - **Example:** VMware, VirtualBox.
- **Type I Hypervisor (Traditional or Bare Metal VMs)**
 - Hypervisor is directly installed on hardware so it's not dependent from OS, so it can interact directly with hardware using ISA calls, this result in **better performance**.
 - Hypervisor can be: **microkernel** (e.g., Xen, MS Hyper-V) or **monolithic** (e.g., VMware ESX).
 - Microkernel VMMs **handle CPU/memory**; device drivers are in a privileged guest OS (Domain 0).

- Monolithic VMMs **handle everything**.
 - Multiple VMs can be managed by the Hypervisor which directly allocate resources to VMs, which is generally more efficient. ISA is presented to VMs.

Trap on Privileged Instructions: When a virtualized OS tries to execute a kernel-only instruction, because it thinks that can interact directly with Hardware (given ISA presented by VMM), it causes a trap which transfer control to the hypervisor .

Both type 1 and 2 VMMs use this mechanism, the difference is that Type 1 VMM can access directly the hardware while Type 2 has to pass through the host OS.

This **prevents** virtualized OS to access hardware or compromise other VM.

The hypervisor inspects the instruction :

- If **from guest OS**, it performs it.
- If **from user program**, it emulates hardware behavior.

Examples of Hypervisors (Type 1 and Type 2):

- Virtualization without HW support: ESX Server 1.0 (Type 1), VMware Workstation 1 (Type 2).
- Paravirtualization: Xen 1.0 (Type 1), VirtualBox 5.0+ (Type 2).
- Virtualization with HW support: vSphere, Xen, Hyper-V (Type 1), VMware Fusion, KVM, Parallels (Type 2).

Full Virtualization

Full virtualization was the **initial goal of virtualization**, it aims to create an environment where a standard, unmodified guest operating system can run without being aware that it's being virtualized. The guest OS believes it has direct access to hardware.

How it Works

Instruction Stream Scanning: Hypervisor constantly monitors the instructions the guest OS tries to execute.

- **Noncritical Instructions: Executed on host hardware** for efficiency.
- **Critical Instructions:** Trapped, so VMM intercepts them and emulate hardware behavior, ensuring the guest OS functions correctly but safely.

x86 architecture architecture posed some **challenges** to Full virtualization:

- It was **not natively virtualizable**, a key issue was that sensible instructions (instructions that could alter the host machine, so break isolation) were not a subset of privileged instructions, this means that they could be executed by the guest OS without trapping to a hypervisor, making it hard to isolate VMs properly.
- **Variety of hardware peripherals** in x86 systems added to the challenge of virtualizing them all correctly.
- The x86 **architecture complexity**.

Binary translation overcomes this issue through the following mechanisms:

1. **Proactive Code Analysis:** The hypervisor scans the instruction stream of the guest operating system *before* it is executed by the physical CPU.
2. **Identification of Problematic Instructions:** During this scan, the binary translator identifies any non-virtualizable sensitive instructions, as well as other privileged or critical instructions that require mediation.
3. **On-the-Fly Code Rewriting (Translation):** When a problematic instruction is found, the binary translator **replaces it with a new sequence of instructions**. This new sequence either:
 - Safely emulates the intended behavior of the original instruction within the virtualized context.
 - Replaces the instruction with a direct call to a specific routine within the hypervisor, which then handles the operation on behalf of the guest.
4. **Control Flow Management:** The translator also modifies branch instructions at the end of basic blocks to ensure that **control returns to the hypervisor**. This **allows the hypervisor to continue the process** on subsequent blocks of code before they are executed.
5. **Transparency to the Guest OS:** The guest operating system remains unmodified and unaware that its instructions are being translated while allowing safe and correct execution of its instructions.

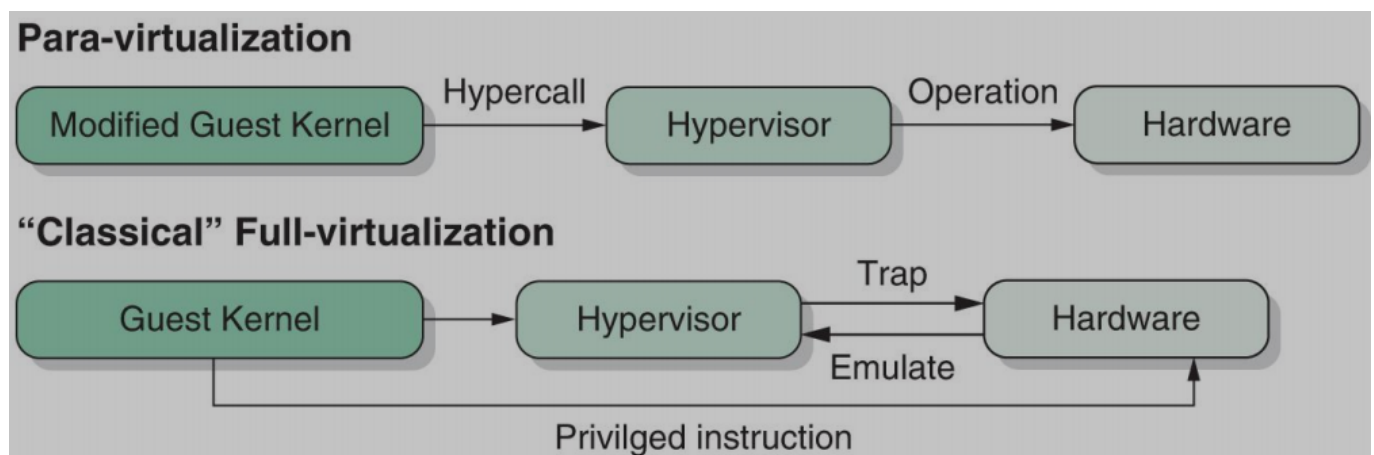
To **enhance performance**, translated blocks of code are often **cached**, so they do not need to be re-translated each time they are encountered.

In essence, binary translation provided a software-based method to enforce the principles of virtualization (isolation and control) on architectures that lacked the necessary hardware support to do so via simple trap-and-emulate for all sensitive operations. It allowed hypervisors to manage and control guest OS behavior even when the hardware wouldn't automatically signal problematic instructions.

Hardware-supported Virtualization Technology

With the advent of Intel VT-x (Virtualization Technology) and AMD-v, sensitive instructions became a subset of privileged instructions. This is a crucial architectural change that makes virtualization much cleaner. Now, instructions that need to be managed by the hypervisor (sensitive ones) reliably cause a trap to the hypervisor because they are also privileged.

Performance Note: While hardware support simplifies trapping, a lot of trap **slow down the performance** vs binary translation. This implies that even with hardware support, if every sensitive operation caused a trap and required hypervisor intervention, it could still be slower than a well-optimized binary translation system that minimizes traps by proactively rewriting code.



In paravirtualization, the guest OS is *modified* to be aware that it is running in a virtualized environment.

Instead of the guest OS attempting to execute privileged or sensitive instructions that would cause a hardware trap (even if reliably handled by hardware-assisted virtualization), the modified guest OS makes direct calls to the hypervisor. These calls are known as "**hypercalls**"

Modern Approach

Modern virtualization solutions often employ a **hybrid approach**.

They leverage **hardware virtualization** support for core CPU and memory virtualization (providing strong isolation and allowing unmodified guests for these aspects) while using **paravirtualization** for other components, particularly I/O (e.g., network and disk drivers).

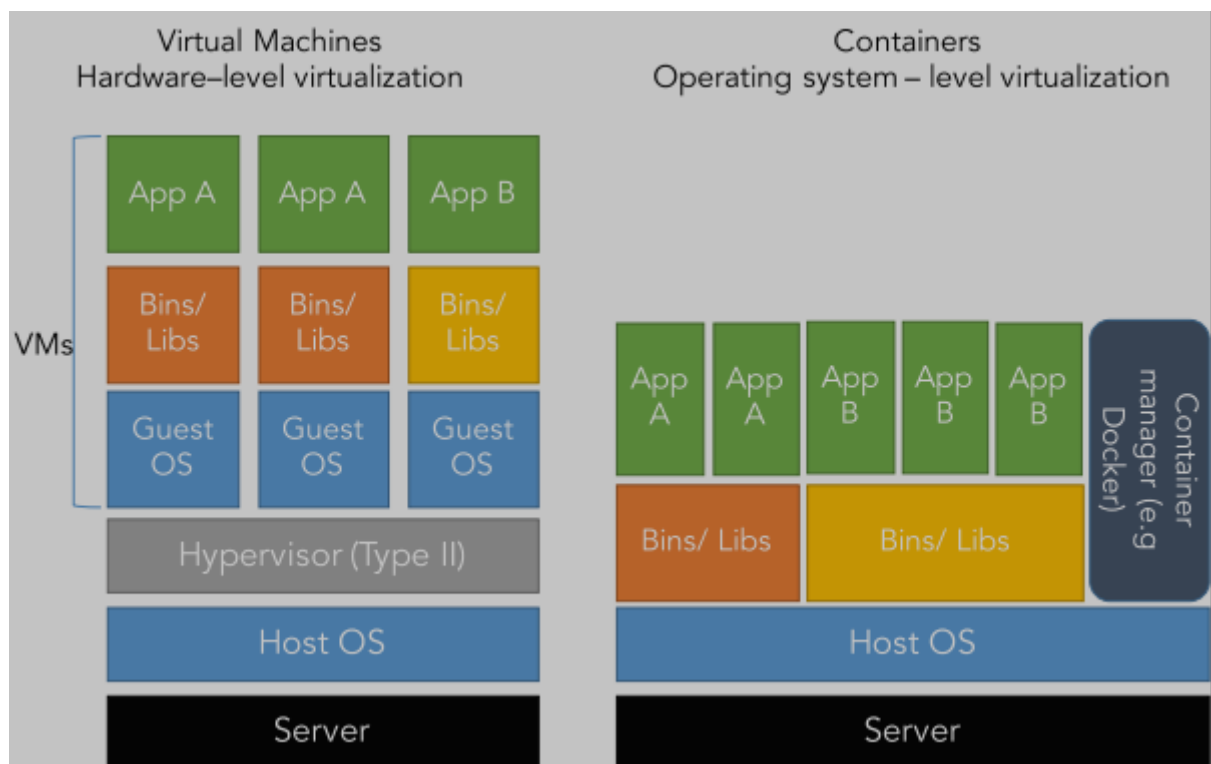
Paravirtualized drivers (PV drivers) allow the guest OS to communicate more efficiently with the hypervisor for I/O operations, avoiding the overhead of emulating hardware devices which can involve numerous traps.

This hybrid model seeks to combine the best of both worlds: the ability to run largely unmodified guests (thanks to hardware support) with the performance benefits of paravirtualization for specific, performance-critical interfaces.

Containerization

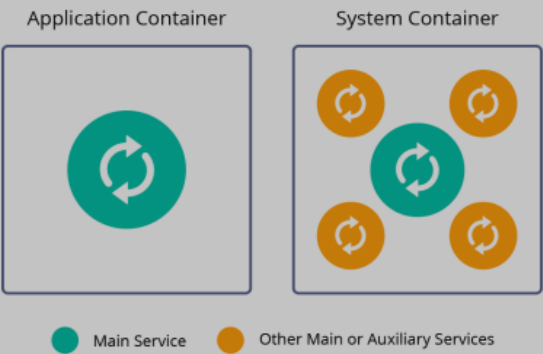
Containerization is a form of operating system-level virtualization where applications run in isolated user spaces, called **containers**, which contains only OS libraries and dependencies required to run the process, microservice or application, all sharing the host OS.

Unlike traditional virtual machines, containers **do not require hardware emulation** or a **separate guest (virtualized) OS for each application**. They are essentially **isolated groups of processes on a single host** directed by a Container Manager (e.g. *Docker*)



Application Containers vs System Containers

Application containers	System containers
Contains a single process	Contains a entire Operating System
Use a layered filesystem	File system neutral
Run microservices / application	Run a lightweight virtual machine
Used for distributing applications	Used for providing underlining infrastructure
Implementations: Docker, CRI-O	Implementations: LXC/LXD, OpenVZ/Virtuozzo, BSD jails, Linux vServer,



- **Application containers:**
 - Contain a single process.
 - Use a layered filesystem.
 - Run microservices/applications.
 - Used for distributing applications.
 - Implementations: Docker, CRI-O.
- **System containers:**
 - Contain an entire operating system.
 - Filesystem neutral.
 - Run a lightweight virtual machine.
 - Used for providing underlying infrastructure.
 - Implementations: LXC/LXD, OpenVZ/Virtuozzo, BSD jails, Linux vServer.

Fundamental Technologies for Containerization

- **chroot & pivot_root**
 - Containerization evolved from the **chroot mechanism**, introduced in UNIX, later adopted in BSD and Linux.
 - **chroot** allows changing the root directory of a process and its children to a specific directory, isolating their filesystem view.
 - Basic **chroot** involves creating a new root directory, copying necessary binaries/libraries, and running **chroot**.
 - However, a simple **chroot** is limited, not a strong standalone security feature, and root users can easily escape. It doesn't provide process or network isolation.
 - Modern container runtimes use **pivot_root(2)** instead of **chroot(2)**, which offers better isolation.

There is no network isolation, too. This missing isolation paired with the ability to leave the chroot jail leads into lots of security related concerns, because jails are sometimes used for wrong (security related) purposes.

How to solve this? **Linux namespaces**.

- **Namespace**

- Wraps a global system resource in an abstraction layer.
- Processes within a namespace perceive their own isolated instance of the resource.
- Changes in a namespace are **visible to other members** but **invisible outside**.
- Key namespaces **examples** include: **(SKIPPABLE)**
 - **Mount (mnt)**: Allows creating separate mount spaces, separated filesystem "mounted" to main filesystem.
 - **UNIX Time-sharing System (uts)**: Introduced in Linux 2.6.19 (2006). Isolates the domain name and hostname.
 - **Interprocess Communication (ipc)**: Introduced in Linux 2.6.19 (2006). Isolates IPC resources (System V IPC objects, POSIX message queues). IPC objects are destroyed with the namespace.
 - **Process ID (pid)**: Introduced in Linux 2.6.24 (2008). Gives processes independent sets of PIDs, allowing duplication across namespaces. Namespaces can be nested. The first process gets PID 1. `/proc/$PID/ns/pid` links to the namespace. Demonstrates creating isolated PID views using `unshare -fp --mount-proc` and remounting `/proc`.
 - **User ID (user)**: Isolation introduced in Linux 3.5 (2012), unprivileged creation in 3.8 (2013). Isolates user and group IDs, allowing processes to be privileged inside while unprivileged outside. `/proc/$PID/{u,g}id_map` control mappings. The `setgroups` security issue is addressed with `/proc/$PID/setgroups`. Essential for rootless containers.
 - **Cgroup**: Namespace added in Linux 4.6 (2016) to prevent leaking host information.
- System calls for namespace management:
 - `clone()`: Creates a new process in one or more new namespaces, arguments of this syscall allow to specify what namespace to create.
 - `unshare()`: Allows a process to disassociate from shared execution context, creating new namespaces for itself.
 - `setns()`: Joins an existing namespace via a file descriptor (e.g., from `/proc/$PID/ns/*`).

• Cgroups (Control Groups)

- Primary goal: Resource limiting, prioritization, accounting, and controlling.
- Organizes processes into **hierarchical groups**.
- Interface provided through a pseudo-filesystem called `cgroupfs` (`/sys/fs/cgroup`).
- Allows setting limits (e.g., memory limits via files like `memory.limit_in_bytes`, `memory.swappiness`) and assigning processes to groups (`cgroup.procs`) by writing the PID.

• UnionFS

- A collection of merged directories is called a **Union**, and each physical directory is a **Branch**.
- UnionFS allows logically **merging several directories or filesystems** (branches) **into a single virtual view** (a union).
- UnionFS **properties**:
 - Enables keeping related files in separate physical locations while presenting them together logically.
 - Acts as a **filesystem interface** to the kernel and presents itself as the kernel's VFS to the filesystems it stacks on.
 - **Assigns precedence** to branches; higher precedence branch overrides lower.

```

$ ls /Fruits
Apple Tomato
$ ls /Vegetables
Carrots Tomato
$ cat /Fruits/Tomato
I am botanically a fruit.
$ cat /Vegetables/Tomato
I am horticulturally a vegetable.
# mount -t unionfs -o dirs=/Fruits:/Vegetables none
/mnt/healthy
$ ls /mnt/healthy
Apple Carrots Tomato
$ cat /mnt/healthy/Tomato
I am botanically a fruit.

```

*** here Fruits has higher priority than Vegetables

- **Combines** directory contents and attributes, **removing duplicates**, when directories exist in multiple branches.
- If a file exists in multiple branches, the version in the higher-priority branch is used.
- Supports **mixing read-only and read-write branches**, with the **union being read-write overall**.
- Uses **copy-on-write semantics**: writes to read-only branches are copied up to a higher-priority read-write branch (a copyup operation). Demonstrates patching a CD-ROM using this.

• Composing Namespaces

- Namespaces are **composable**, enabling complex isolation scenarios like Kubernetes Pods (isolated PIDs sharing a network interface).

Docker

Docker Basics

Docker is a containerization platform that provides the ability to package and run an application in a loosely isolated environment called a **container**.

- **Containers** are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host.
- **Container Image** is a standardized package that includes all of the files, binaries, libraries, and configurations to run a container.
 - Defined by a **Dockerfile**.
 - Can be found in **Docker Hub**, which is a public **registry**.
- **Docker Registry**: Stores Docker Images, can be either Public (Docker Hub) or private.
- **Docker Repository**: Collection of **related Docker Images** within a Docker Registry.
 - *Example*: Docker Rep. may contains everything you need to run a DB service.
- There are two important principles of images:
 - Images are **immutable**. Once an image is created, it can't be modified. You can only make a new image or add changes on top of it.
 - Container images are composed of **layers**. Each layer represents a set of file system changes that add, remove, or modify files. Each instruction in a Dockerfile (**FROM**, **COPY**, **RUN**, **CMD**) creates a new **read-only** layer. *Example*: If you are building a Python app, you can start from the Python image and add additional layers to install your app's dependencies and add your code. This lets you focus on your app, rather than Python itself.

Docker Architecture

Docker engine is a Client/Server application:

- **Docker CLI (Client):** Command line interface to interact with the server using REST API.
- **Docker Daemon (Server):** Listen to Docker API requests and creates and manages Docker objects (containers, images, networks, data volumes).
- They, either:
 - Run on the **same system**
 - Docker Client can connect to **remote Docker Daemon**

```
docker run -i -t ubuntu /bin/bash
```

Example of CLI command: W.r.t. above command, Docker Daemon does what follow:

1. Locate and, eventually, download Ubuntu image from registry.
 2. Create a new container using Dockerfile from Ubuntu image.
 3. Allocate R/W filesystem as final layer of the image.
 4. Create Network Interface connected to default network.
 5. Start container and run `bin/bash` command.
- **Docker Compose:** Another Docker client that lets you work with applications consisting of a set of containers. With Docker Compose, you can define all of your containers and their configurations in a single YAML file. If you include this file in your code repository, anyone that clones your repository can get up and running with a single command.
 - **Docker Desktop:** All-in-one application that contains Daemon, Compose, Kubernetes, ...

Storage Options

By default all files created inside a container are stored on a **Container writable layer** (R/W filesystem mentioned above) that sits on top of the read-only, immutable image layers (from Dockerfile commands).

Data written to the container layer doesn't persist when the container is destroyed. This means that it can be difficult to get the data out of the container if another process needs it, reducing portability. **Solution** to this are:

Volumes

- **Persistent** storage mechanisms **managed by the Docker daemon**.
- **Stored in host filesystem** but managed by Docker (**isolated access**), in order to use it, you must **mount the volume to a container**.
- Can be mounted in **multiple containers simultaneously**.
- Easier to back up/migrate, safer to share among containers.
- Volume drivers allow storage on remote/cloud providers, encryption, etc.
- Managed via CLI or API.

Volumes are ideal for performance-critical data processing and long-term storage needs.

Bind mounts

Mount any host filesystem area (file or directory) into a container, referenced by its host path. Can be created by the container if they don't exist. Shared with host processes. Not portable. Usually higher performance than volumes.

- **Tmpfs mount:** Mounts a temporary memory area outside the container's writable layer. Persisted only in host memory. Removed when the container stops. Available only in Linux hosts. Not sharable among containers.

Activity cases for storage options:

- Backing up/migrating data: Volumes.
- Sharing data among multiple running containers: Volumes.
- Sharing configuration from host to containers: Bind mount (if host path is guaranteed).
- Data shouldn't persist: Tmpfs mount.
- Sharing source code/build artifacts: Bind mount (if host path is guaranteed).
- Storing data on remote/cloud: Volumes.
- Host filesystem structure guaranteed: Bind mount.
- Host filesystem structure NOT guaranteed: Volumes.

Networking

Containers can connect to each other or non-Docker workloads. Containers don't need to be aware they are on Docker. Types of network drivers:

- **Bridge:** Default driver. For standalone containers communicating with each other.
- **Host:** Removes network isolation, container uses host networking directly. For standalone containers.
- **Overlay:** Connects multiple Docker daemons for swarm services or standalone containers on different hosts to communicate.
- **Macvlan:** Assigns a MAC address to a container, making it appear as a physical device on your network. Useful for legacy applications expecting direct physical network connection.
- **None:** Disables networking for the container.

Activity cases for networking drivers:

- Migrating from VM setup: Macvlan.
- Network stack not isolated: Host networks.
- Containers on different hosts communicate: Overlay networks.
- Multiple containers on same host communicate: User-defined bridge networks.
- Containers look like physical hosts (unique MAC): Macvlan networks.

Outcome

- Covered Docker and its underlying technologies: Namespace, Cgroup, Union FS.
- Explained roles of Docker daemon, CLI, and API.
- Discussed Layered images.
- Reviewed Storage options: volume, bind, tmpfs.
- Outlined Networking concepts and drivers.