Containers

Florent Glück - florent.gluck@hesge.ch

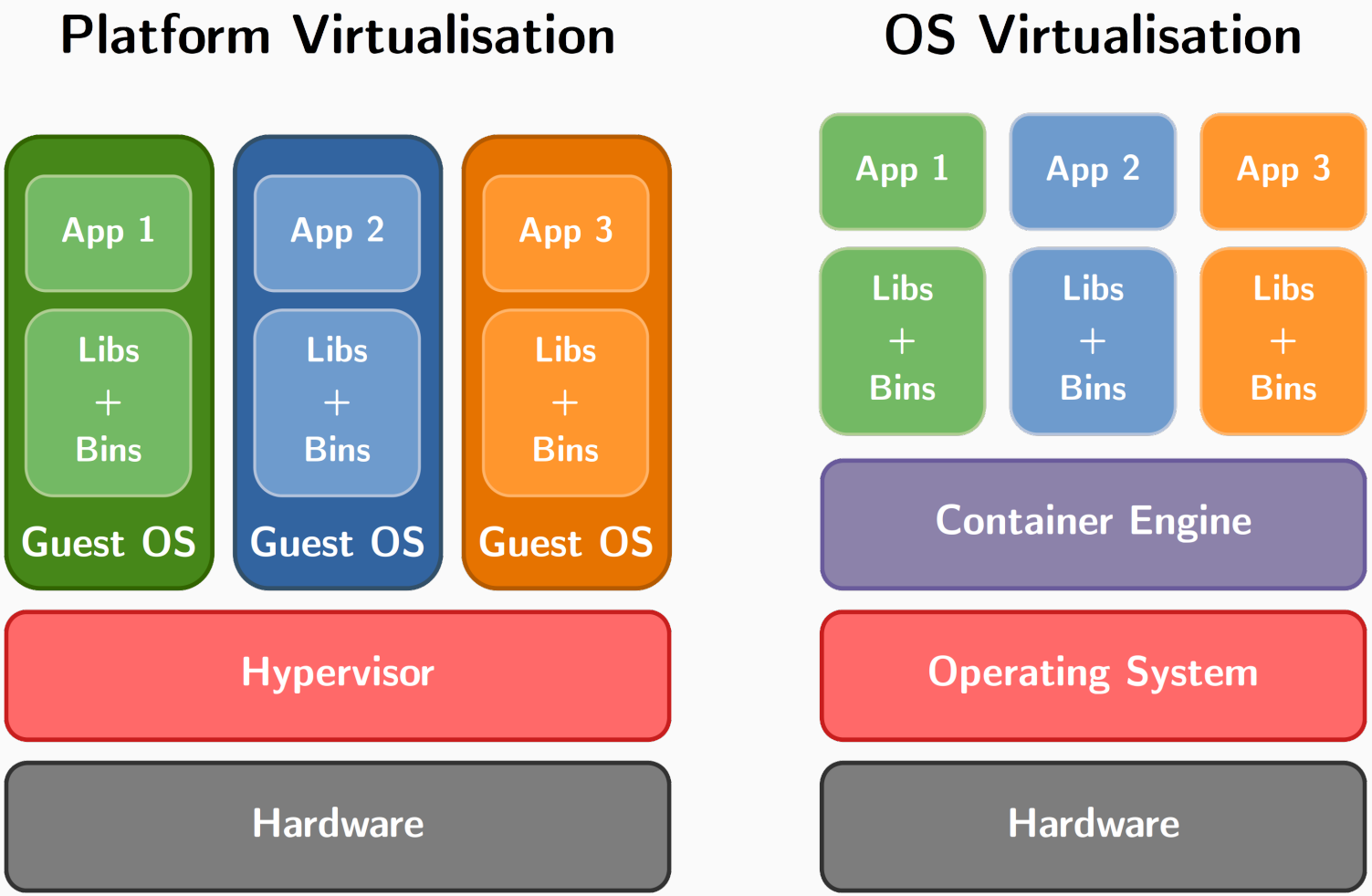
April 07, 2025

ISC - HEPIA

Operating system virtualization

- Technology that enables the kernel to create/manage multiple **isolated user-space instances** called **containers**
- Operating System (OS) virtualization is also called **containerization**
- Processes inside a container **only** see the container's contents and devices assigned to it
  - provides **isolation** (also called *sandboxing*)
- Kernel provides **resource-management** to limit the impact of a container's activities on other containers

Platform Virtualization vs OS Virtualization (1/2)



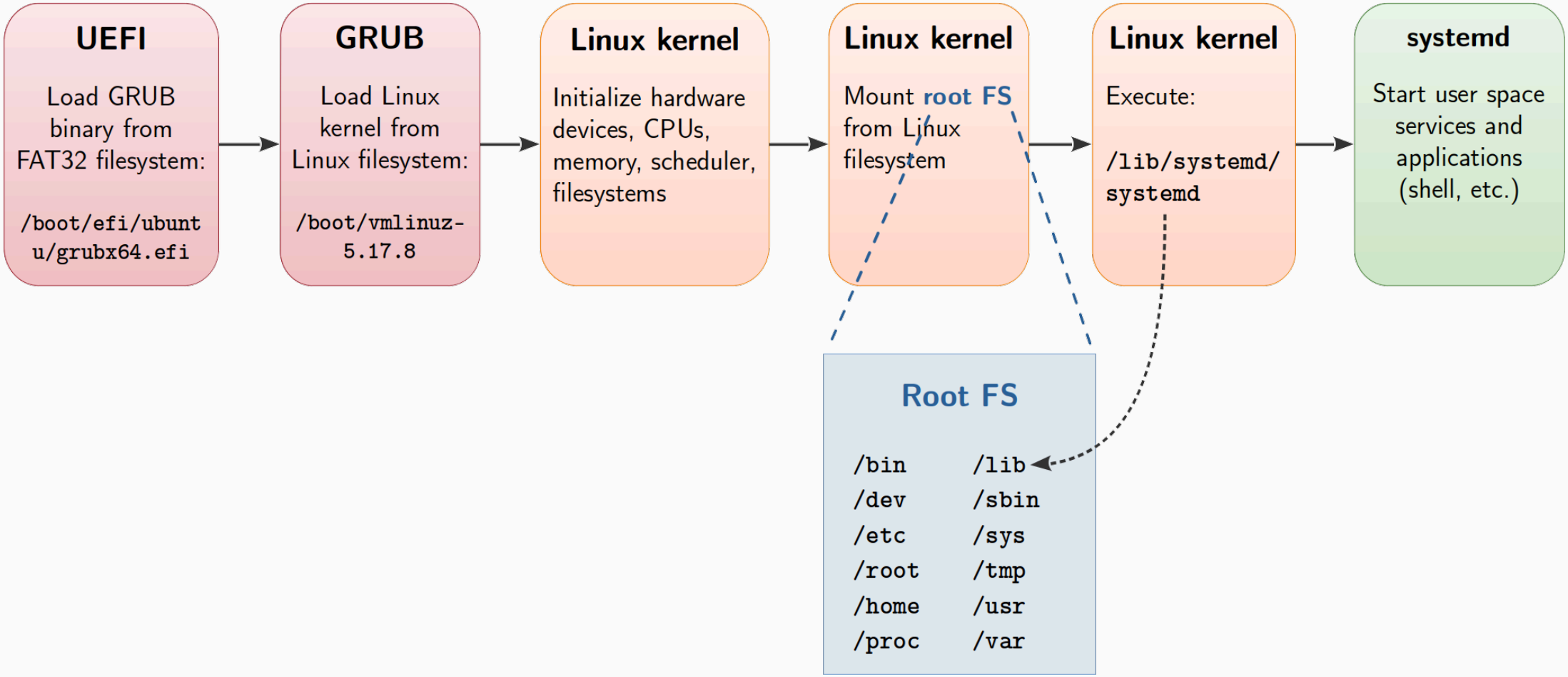
Platform virtualization vs OS virtualization (2/2)

- Virtual machines (VMs) have **different guest OS**, each with their **own kernel**
- Containers **share the same kernel**, but have different **root filesystem**, view of process tree, networking, etc.
- Compared to VMs, containers have **less overhead**, but at the cost of **less isolation (less security)**

Root filesystem

What is a **root filesystem**?

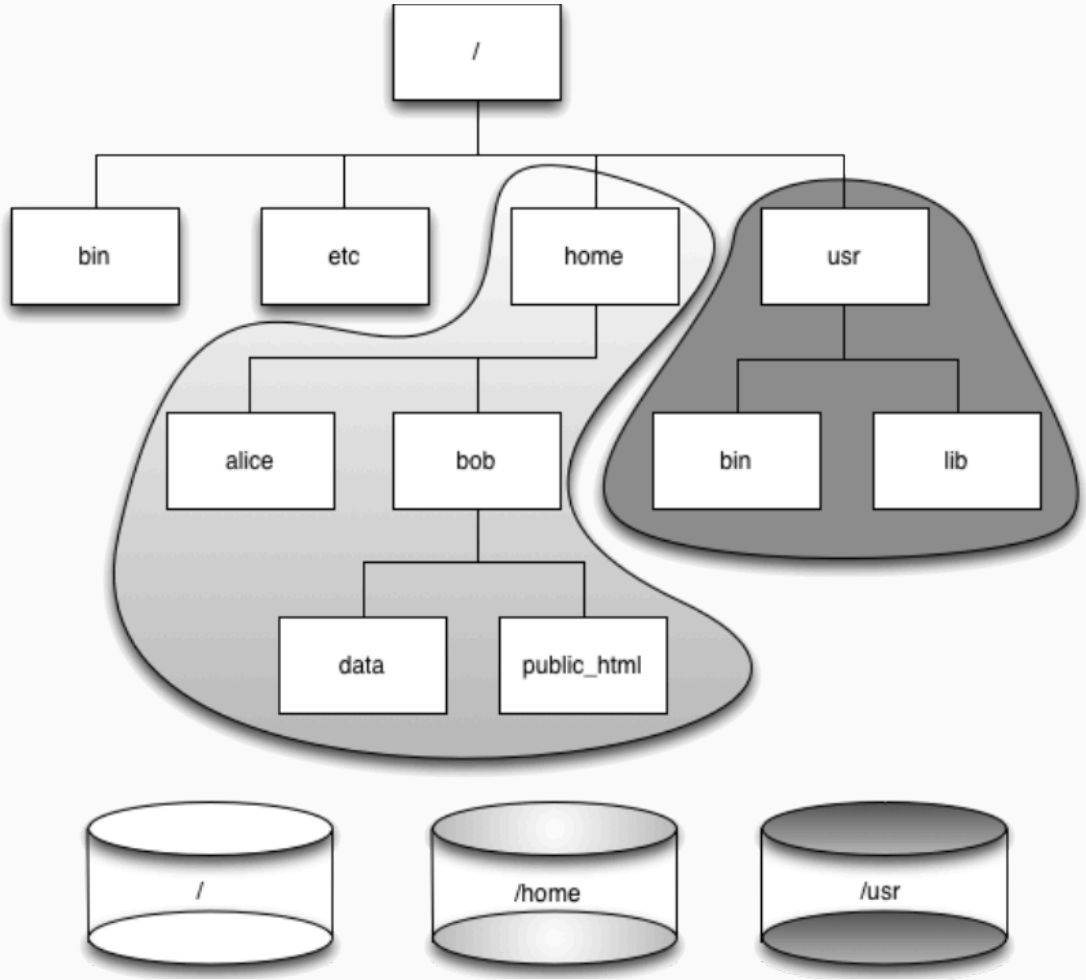
Linux boot process



Root filesystem (rootfs)

- UNIX systems have a **single global hierarchy** (tree) of files and directories
  - it can be composed of different filesystems
- A specific filesystem is mounted at the root of the hierarchy
  - it is identified by /
- This filesystem is called the **root filesystem** or simply **rootfs**

Root filesystem example



/

- 1st partition of SDD
- Read/write
- ext4

/home

- 2nd partition of SDD
- Read/write
- btrfs

/usr

- Remote network share
- Read/write
- NFS

What is a container?

- Multiple definitions depending on the type of containers framework
- **A container is a set of processes that are isolated from the host system and other containers**
  - with most frameworks, the files (rootfs) necessary to run the containers are provided as an image
- Multiple containers can run within the same host machine
- Containers sometimes called "lightweight VMs" $\longrightarrow$ however, **they are not VMs!**

History of containers

- Container technology has existed for a long time in various forms
- Significant popularity gain since the creation of Docker in 2013

Year	Technology	Operating System
1982	chroot	UNIX
1992	namespaces	Plan9
2000	BSD Jails	FreeBSD
2001	Virtuozzo containers	Linux, Windows
2004	Solaris Zones	Sun Solaris, Open Solaris
2008	LXC	Linux, also called “Linux containers”
2013	Docker	Linux, FreeBSD, Mac, Windows
2015	Singularity	Linux, containers for HPC
2018	Podman	Linux, deamonless Docker alternative

Containers look like virtual machines

From a distance: a container looks like a VM:

- I can SSH into my container
- I can have root access in it
- I can install packages in it
- I have my own network interface
- I can tweak routing table, iptables rules
- I can mount filesystems
- etc.

Containers: why?

- Lightweight, fast, disposable... virtual environments
  - “boot” in milliseconds
  - just a few MB of intrinsic disk/memory usage
  - bare metal performance is possible
- Can be used as “light” virtual machines, but with less isolation
- Can be used to build, ship, deploy, and run applications

Benefits of containerization

- **Isolation & security**
  - Provide a complete isolated and restricted OS environment
  - Allow packaging and isolation of applications with their entire runtime environment
- **Portability**
  - Container packaged with all its dependencies
- **Productivity**
  - Performance: lightweight environment
  - Consolidation: maximize resource utilization
  - Continuous integration: development, test, deployment

Containers use cases

- **Application packaging**
  - flatpak, snap, in-house
- **Infrastructure/datacenter use**
  - system virtualization $\longrightarrow$ lightweight "VMs"
  - limit applications resources' usage: memory & CPU
- **Service compartimentalization**
  - apps/services isolation $\longrightarrow$ security
  - modularity $\longrightarrow$ scalability and flexibility
- **Hosting business**
  - give a user root access without full (root) access to the "real" system

Containers philosophy: microservices

- Every component should be isolated to the finest details and containerized at that level
- Containers can be grouped together to provide a complete application
- Example: Wordpress deployment:
  - 1 nginx container
  - 1 mariadb container
  - 1 php-fpm container
- Benefits of microservices: **modularity** and **scalability**!
  - Ability to scale on demand: create more php-fpm containers when needed

Containers vs virtual machines

- Containers are lightweight compared to traditional VMs $\longrightarrow$ more containers can be run per host than traditional VMs
- Unlike containers, VMs require emulation layers $\longrightarrow$ VMs consume more resources and add overhead
- Starting a container is much faster[1] than starting a VM (no boot sequence!)
- Containers share resources with the underlying host machine and user space
- Containers provide much weaker isolation than VMs
  - containers are not as secure as VMs!

[1] when running an "equivalent" system

Which is better: containers or virtual machines?

Containers and VMs serve **different needs**:

- Containers solve deployment issues and permit dynamic scaling more easily than VMs
- Containers are easier to deploy and more lightweight
- VMs can provide a full desktop environment
- VMs can run different OS than the host and even emulate different architectures (technically, not VMs anymore)

Limitations of containers

Containers use same kernel as host $\longrightarrow$ imposes strong **limitations**:

- **Limited** to running applications compiled for the host's **kernel architecture**
  - Limitation from an hardware (CPU) point of view: can't run an `armhf` (ARM) container on top of an `amd64` (x86-64) system[1]
  - Can't run a Windows container on a Linux system
  - Limited to the host's kernel (and its features)
- **Reliability**: higher impact of a crash, especially in kernel area

[1] Although possible if using an underlying VM (QEMU) emulating the architecture

Dependency on the host's kernel

Kernel version imposes implicit limitations:

- Newer kernels are **backward compatible** with older kernels
  - new features are added, but old features remain available
  - typically: syscalls
- **The opposite is not true!**
  - an older kernel might not implement a feature present in a newer kernel
- Therefore, an application might require a specific **minimal kernel version**
  - this application **won't work** in a container based on an older kernel!

Containers: how?

Containers make extensive use of 4 key Linux kernel technologies:

Containers: how?

Containers make extensive use of 4 key Linux kernel technologies:

(1) **Capabilities**: provide security

Containers: how?

Containers make extensive use of 4 key Linux kernel technologies:

(1) **Capabilities**: provide security

(2) **Namespaces**: provide isolation

Containers: how?

Containers make extensive use of 4 key Linux kernel technologies:

(1) **Capabilities**: provide security

(2) **Namespaces**: provide isolation

(3) **Control groups** (cgroups): provide limits on resources

Containers: how?

Containers make extensive use of 4 key Linux kernel technologies:

(1) **Capabilities**: provide security

(2) **Namespaces**: provide isolation

(3) **Control groups** (cgroups): provide limits on resources

(4) **Seccomp**: provide security

Problem with traditional UNIX privilege model

- Traditional UNIX privilege model divides users into two groups:
  - normal unprivileged users
  - superuser (root, effective UID 0)
- Problem: granularity, root/non-root, is too coarse
  - no limit on possible attacks if root program is compromised!
- Solution?
  - capabilities

(1) Capabilities

Capabilities provide security through fine-grained privileges

- Capabilities **divide root's privileges** into smaller units
  - 38 capabilities as of Linux 5.4
  - root user = process with full set of capabilities
- Typical goal: replace SUID[1] programs with programs that have capabilities
- Processes and files can each have capabilities
  - process: defines what privileged operations a process can do
  - file: what capabilities a process gets when executing the file
    - stored in extended attributes (`security.capability`)

[1] Programs with the SUID bit set are executed with the same permissions as the file's owner!

(2) Namespaces

Namespaces provide isolation

- Namespaces affect processes (groups of processes)
- Linux supports multiple namespace types:
  - UTS: isolates hostnames
  - mount: isolates filesystems
  - IPC: isolates inter-process communications
  - Network: isolates networking resources
  - PID: isolates process ID
  - User: isolates user and group IDs
  - Cgroup: limit ressources

(3) Control groups (cgroups)

Cgroups provide resources limitation

- Cgroups are used to **control resources among groups of processes**
  - CPU time, memory, network bandwidth, I/O bandwidth
  - provide fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources
  - organized hierarchically (like processes) and child cgroups inherit some of their parents' attributes
- Hardware resources can be divided up among processes and users to increase overall efficiency

(4) Seccomp

Seccomp provides security

- Seccomp is used to **restrict the system calls** a process can use
- Linux kernel provides ∼400 system calls!
- Each syscall is a vector for attack against the kernel
- Most programs use only a small subset of available syscalls
  - remaining syscalls should never occur
  - if they do ⟶ potential attack!
- Seccomp allows to reduce the attack surface of the kernel
  - a key component for building application sandboxing

Container frameworks

- **LXC** provides a "lightweight VM" environment
  - provides standard OS shell interface
- **LXD** provides image management on top of LXC
- **Docker** containers are optimized to run a single application
  - configuration file specifies the base root filesystem, with dependencies needed to run a specific application
  - runs application in a containerized environment
  - easy way to package an application and all its dependencies
- **Podman** is a rootless and daemonless (use `systemd`) alternative to Docker
- Docker & Podman's goal: ship and run applications **anywhere**

Container orchestration frameworks

- Docker **Compose**: framework to manage multiple containers on a single host
- Docker **Swarm** & **Kubernetes**: frameworks to manage multiple containers on multiple hosts
- **Kubernetes**: popular container orchestration framework
  - runs over multiple physical machines
  - auto-scaling when load increases, restart services when they crash
  - fairly complex

OCI, containerd and runc

Open Container Initiative (OCI)

- A set of standards for containers, describing the image format, runtime, and distribution

containerd

- Open-source container runtime from the OCI that handles the lifecycle of containers, including image management, execution, and storage
- `containerd` uses `runc` to execute containers
- `containerd` used by Docker, Podman and Kubernetes

runc

- Low-level open-source lightweight CLI tool for spawning and running containers according to the OCI specification

Resources

- Practical LXC and LXD "Linux Containers for Virtualization and Orchestration", Senthil Kumaran S., Apress 2017
- "Is it safe to run applications in Linux Containers?" Jérôme Petazzoni, 2014
- Namespaces in operation
  https://lwn.net/Articles/531114/
- Control groups Linux kernel documentation
  https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html