Docker

PTS3 / FMFI UK

Jana Kostičová

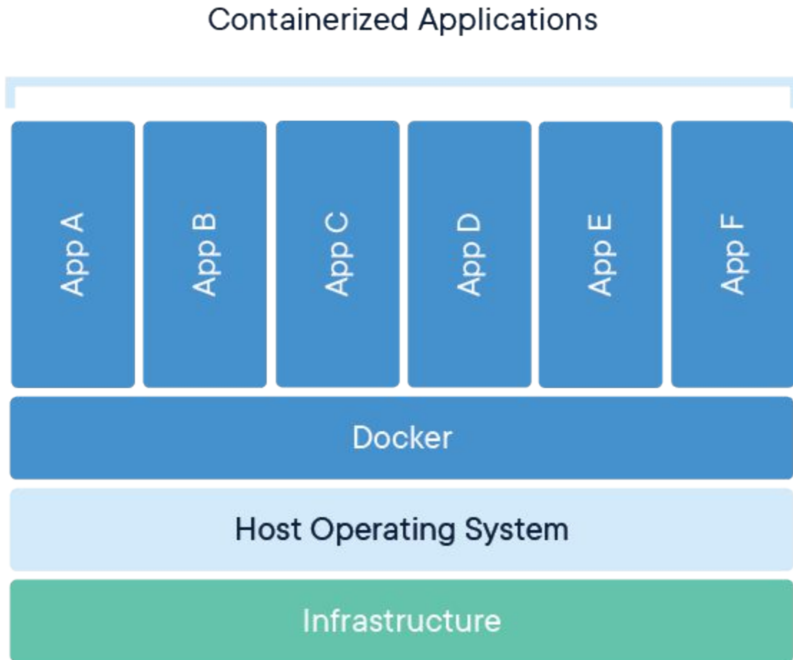
Docker

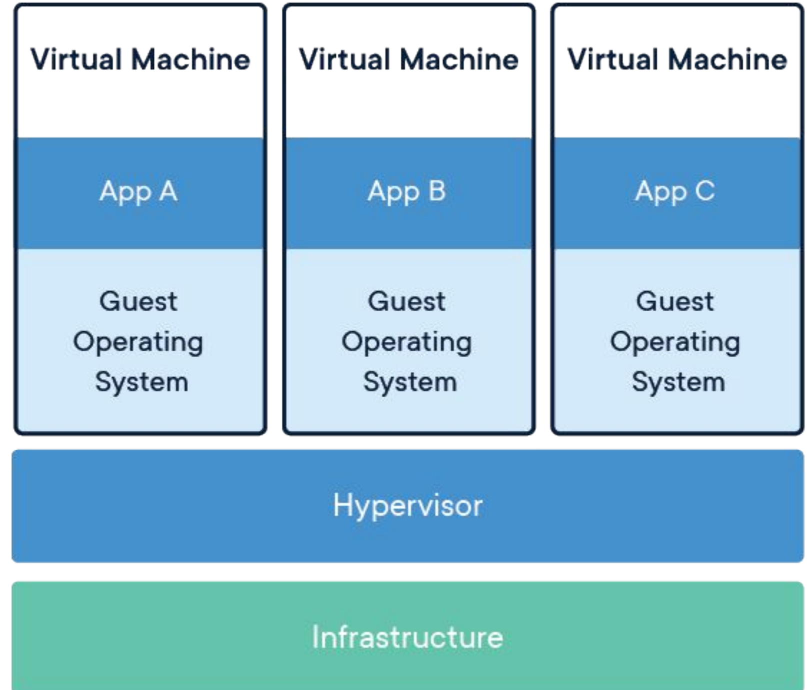
- First released in 2013
- OS-level virtualization
- Docker container packages an application and its dependencies that can run as an isolated process on the host system
  - Self-contained
  - Isolated
  - Independent
  - Portable

Docker vs Virtual Machine (VM)

- VM - HW-level virtualization, each VM has its own guest OS
- Requires a hypervisor - a software / firmware that creates and runs VMs
- 2 types of hypervisors:
  - Type 1 (bare metal): Hardware & hypervisor only (e.g., VMware ESXi, Microsoft Hyper-V, Xen).
  - Type 2 (hosted): Hardware & host OS & hypervisor (e.g., VirtualBox, VMware Workstation).

Docker vs Virtual Machine (1)





Source: Wikipedia

Docker vs Virtual Machine (2)

	Virtualization	Guest OS	Isolation & security	Resource usage	Startup time	Portability
Docker	OS-level	No (shared host OS)	Process-level (weaker)	Low	Shorter	High
Virtual Machine	HW-level (hypervisor)	Yes	Full OS-level (stronger)	Higher (OS layer)	Longer (OS setup)	Lower (include OS, more dependencies - hypervisor type, underlying HW, ..)

Combination is possible: physical server → virtual machines → containers

Use Cases

Virtual Machine

- Legacy applications
- Running applications on multiple OSs
- Security-critical applications

Docker containers

- Microservices
- Modern CI/CD pipelines
- DevOps

Docker containers & Operating systems

Linux Docker containers (standard)

- Combine existing Linux kernel features to achieve process isolation
  - Namespaces → isolate processes
  - cgroups → limit CPU / memory usage
  - Union filesystems → layer images efficiently

→ always need a host Linux kernel to run

“Multi-OS” trick:

Linux Docker containers run also on machines with another OS (typically macOS, Win), but they always use a (hidden) lightweight Linux VM on the top of the host OS!

(Win - WSL2 / Hyper-V, macOS - Lima / Colima / Hyperkit)

Windows Docker containers (less often used)

- Use Windows kernel features to achieve process isolation
- We won’t go into detail

→ always need a host Windows kernel to run

Note: No Docker containers run natively on other OSs (MacOS, unix)!

Process isolation - Namespaces

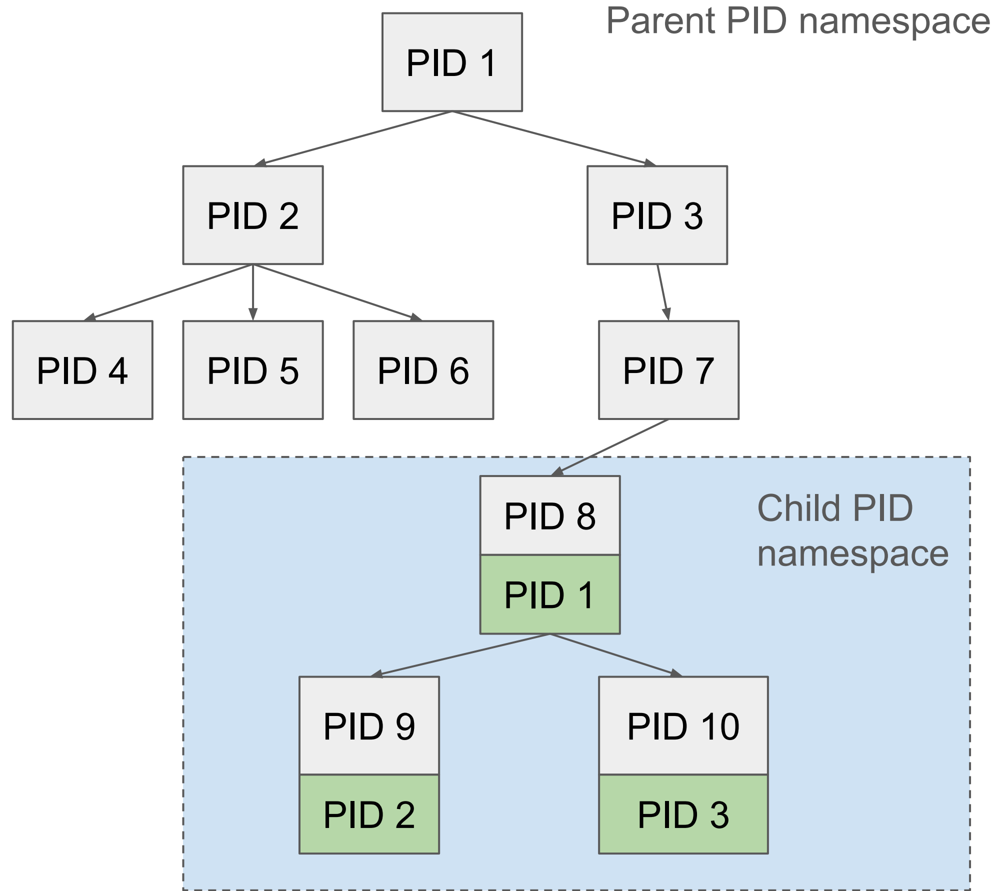
= Linux kernel feature that facilitate process isolation

A namespace wraps a global system resource in a private scope. Each container (isolated process) sees its own version of

- Processes
- Filesystem
- Network interfaces
- Users
- Etc.

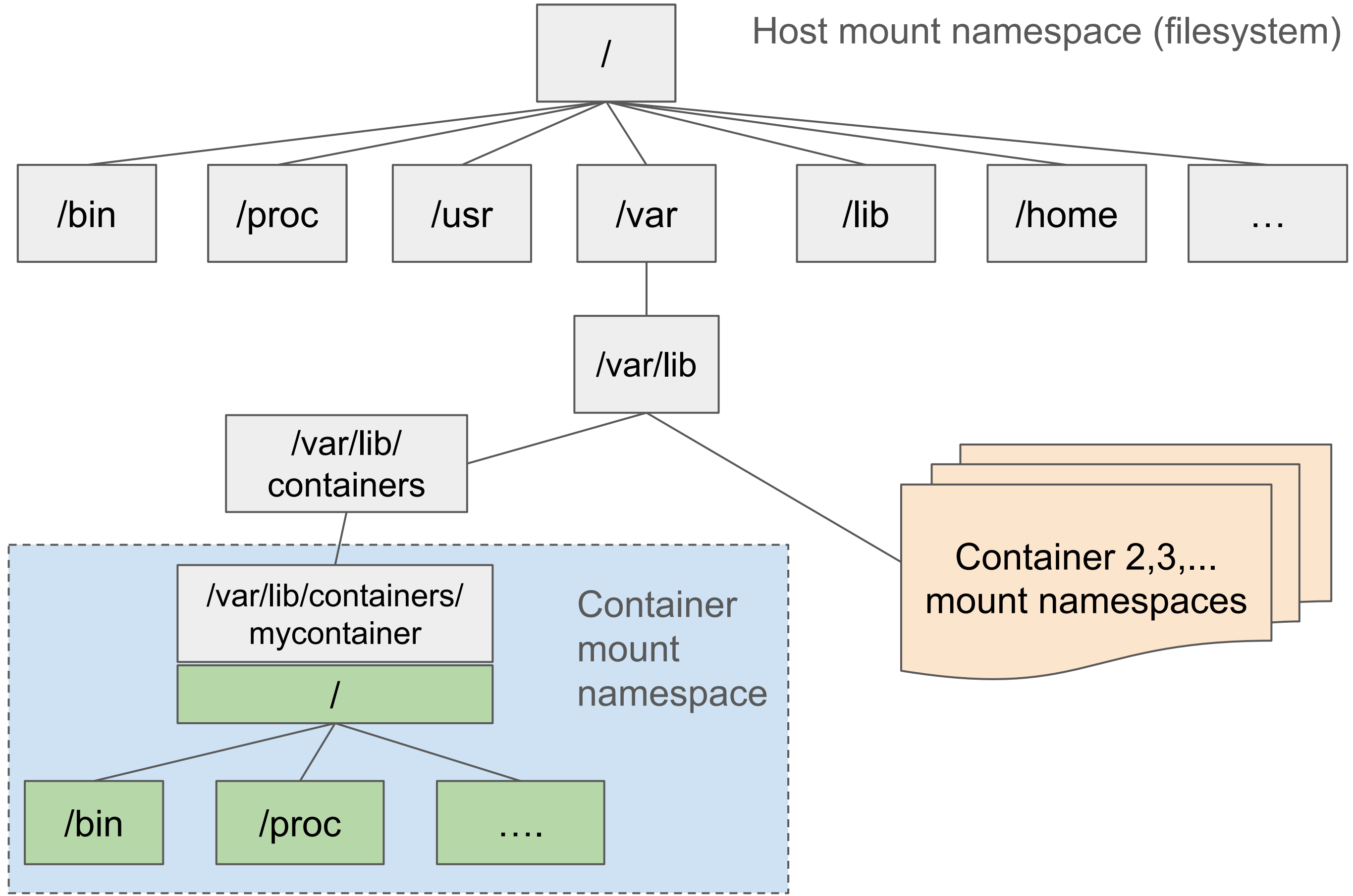
Example

Process isolation



Example

Filesystem isolation



docker run <container>

The statement run a Docker container - this is what Docker does roughly:

1. Creates new namespaces:
   - PID
   - Mount (filesystem)
   - network
   - UTS (hostname, domain)
   - IPC (inter-process communication)
   - user
2. Sets up filesystem:
   - mounts image layers
3. Configures networking:
   - Veth (virtual ethernet) pair + bridge
4. Applies cgroups:
   - resource limits
5. Starts process:
   - your app becomes PID 1 inside container

Docker concepts

We mostly use 101 course from https://www.docker.com/101-tutorial

(+ some extra docker commands / command options)

SW:

- Docker daemon
- Docker client program

Objects:

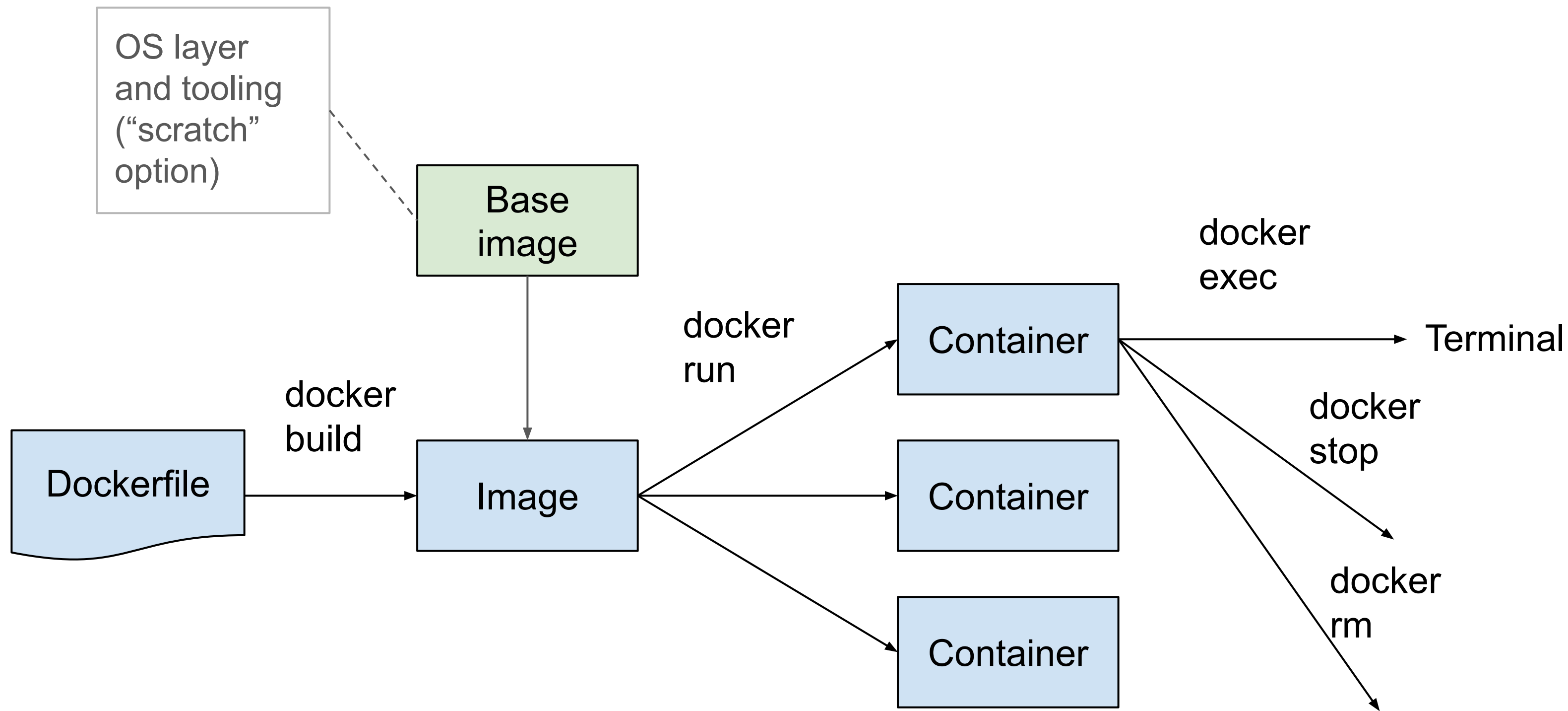
- Docker container
- Docker image
- Docker service (we will not speak about this today)

Registries:

- Docker registry (default one Docker Hub)

Tools:

- Docker Dashboard
- Docker Compose
- Docker Swarm (we will not speak about this today)


OS layer
and tooling
("scratch"
option)
Base
image
docker
exec
docker
run
Container
Terminal
docker
build
docker
stop
Dockerfile
Image
Container
docker
rm
Container

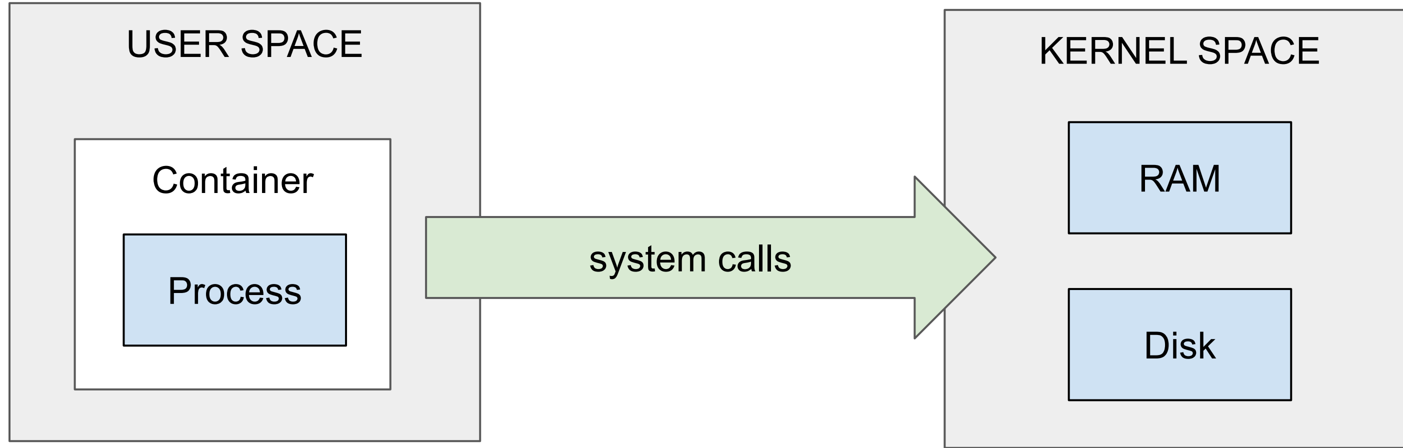
Base images (official) - examples

- **alpine** – ultra-lightweight Linux (~5MB), apk, musl libc
- **ubuntu**
- **debian**
- **node** – Node.js runtime with npm
- **python** – Python runtime with pip
- **golang**
- **eclipse-temurin** – Java (OpenJDK) runtime
- **nginx** – lightweight web server / reverse proxy
- **httpd** – Apache web server
- **postgres**
- **mysql**
- **redis** – in-memory key-value store / cache

All of them are publicly hosted on Docker Hub

“Scratch” - empty image, no need to “pull” it, just reference it as base in the Dockerfile
Used for statically compiled binaries.

User space vs kernel space



You can run any Linux distro user space on any compatible Linux kernel

- Issues are rare: e.g., host kernel is very old or missing some features

Docker container

- Uses user space from the base image (or built from scratch)
- Is isolated from the host user space by default

Image layers

Image layer = a read-only snapshot of filesystem changes, identified by a content hash (SHA256)

- Docker images are built from layers
- If two layers have the same hash, Docker considers them identical and can reuse them between images

Layer sharing

Saves space and network bandwidth; speed up builds & deploys (layer caching)

Registry:

- Layers with identical hash stored once, referenced by multiple images

Local machine:

- Docker checks layer hashes, reuses layers already downloaded

Docker containers vs architecture

- By default, each image is architecture-specific (except scratch)
- Multi-arch manifests
    - A single image name (e.g., python:3.12) points to different images for different architectures

```
docker pull python:3.12
```

- On an amd64 machine, Docker pulls python:3.12-amd64
- On an arm64 machine, Docker pulls python:3.12-arm64

References

- https://en.wikipedia.org/wiki/OS-level_virtualization
- https://en.wikipedia.org/wiki/Docker_(software)
- https://docs.docker.com/engine/reference/commandline/docker/ - CLI reference
- R. Lukoťka: Docker