RAID types

RAID (Redundant Array of Inexpensive Disks or Drives, or Redundant Array of Independent Disks).

Purposes:

performance improvement (max. write read speed),

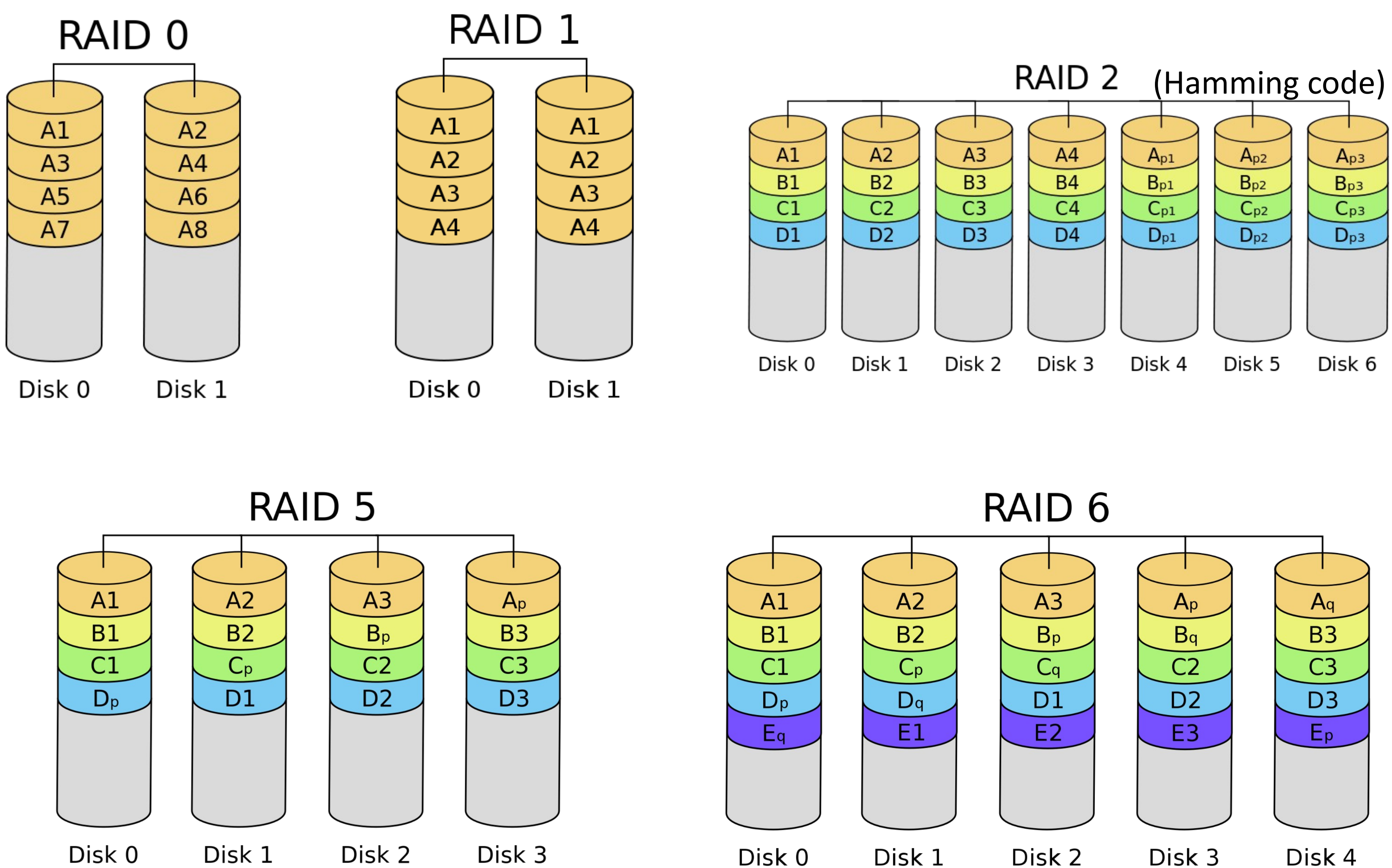
increased data security against hard drive failure.

Hardware RAID = RAID controller

or

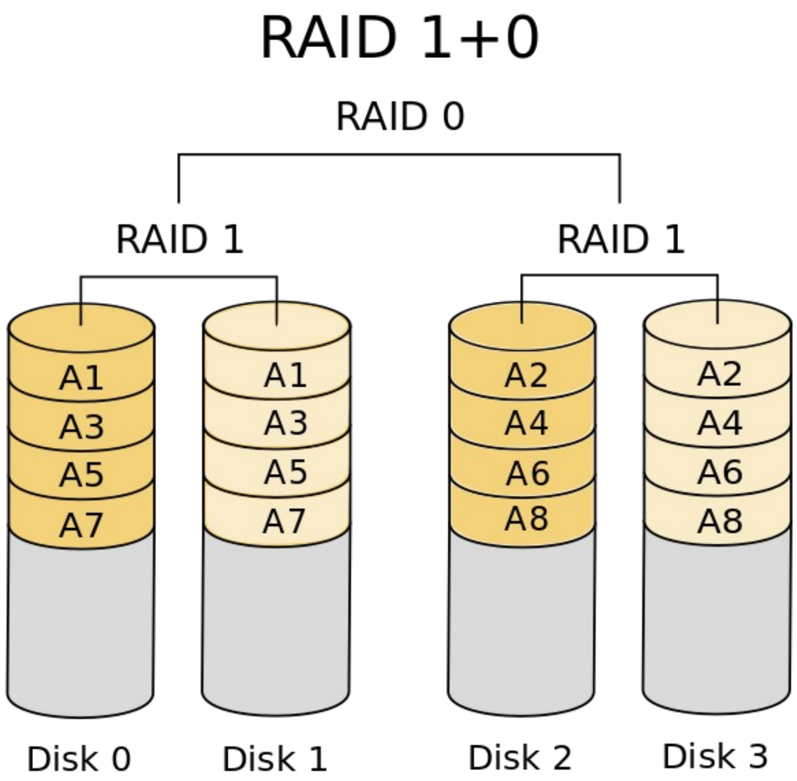
Sofware RAID

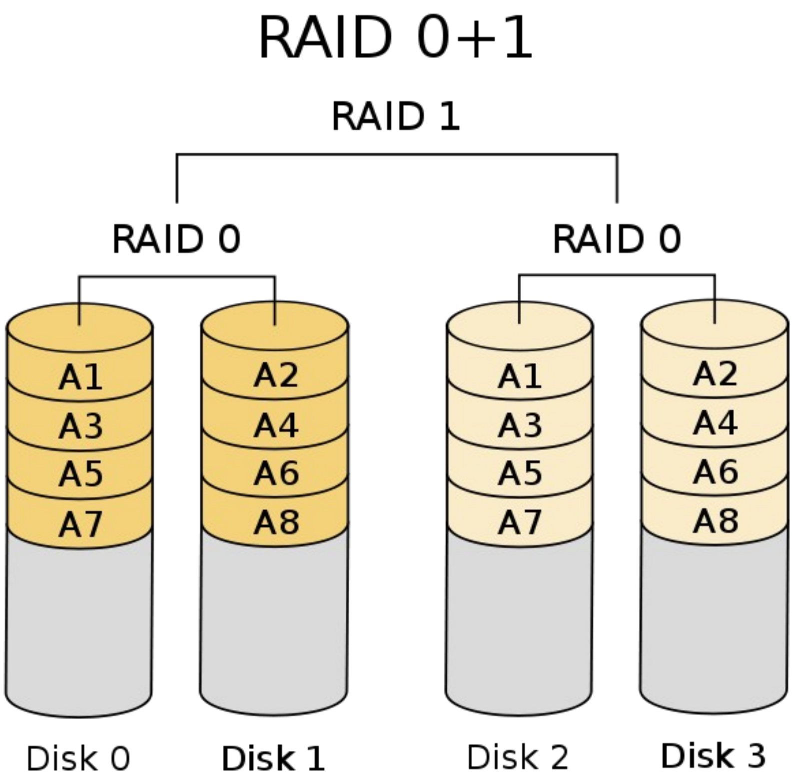
RAID types



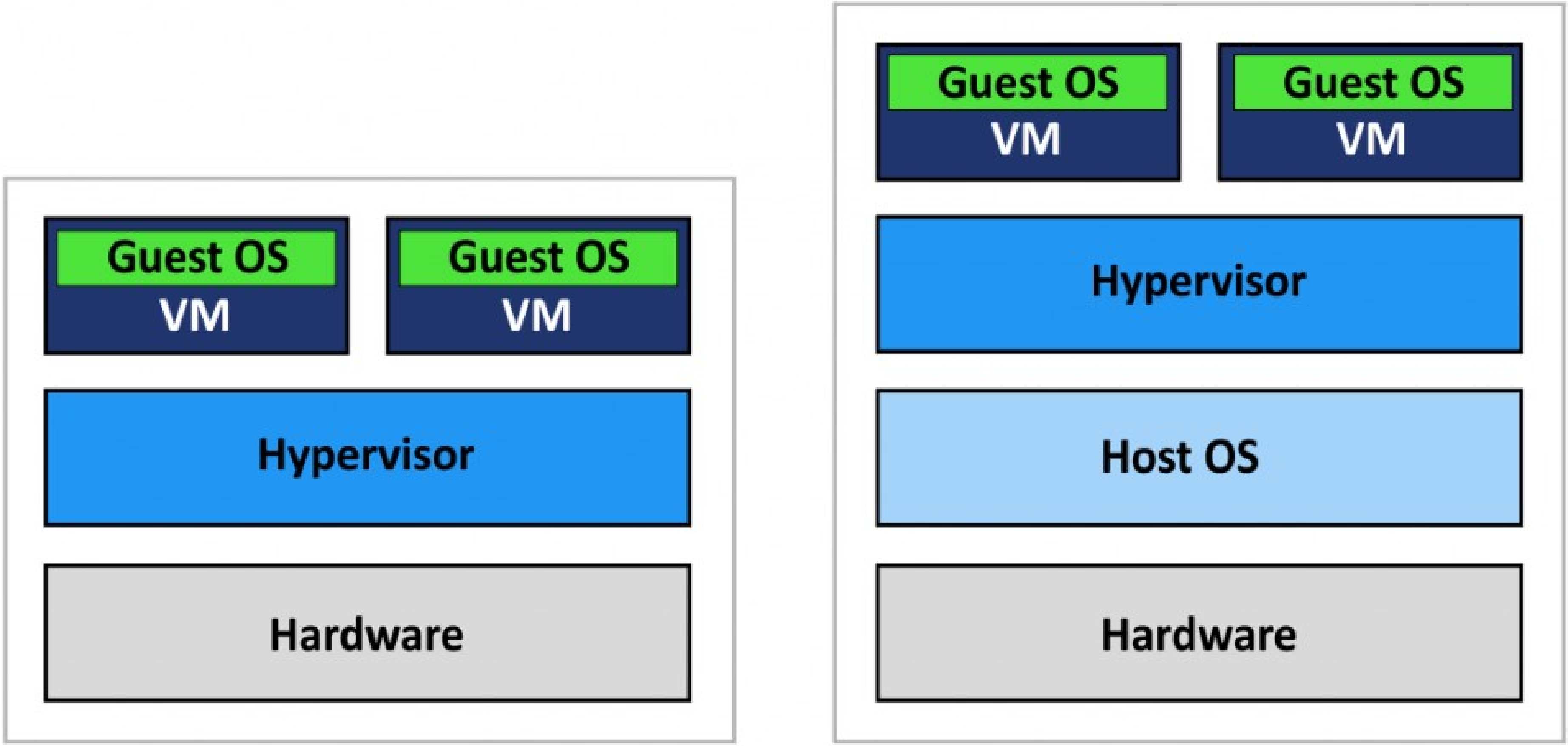
RAID 0
A1 A3 A5 A7
A2 A4 A6 A8
Disk 0
Disk 1
RAID 1
A1 A2 A3 A4
A1 A2 A3 A4
Disk 0
Disk 1
RAID 2 (Hamming code)
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3
A4 B4 C4 D4
Ap1 Bp1 Cp1 Dp1
Ap2 Bp2 Cp2 Dp2
Ap3 Bp3 Cp3 Dp3
Disk 0
Disk 1
Disk 2
Disk 3
Disk 4
Disk 5
Disk 6
RAID 5
A1 B1 C1 Dp
A2 B2 Cp D1
A3 Bp C2 D2
Ap B3 C3 D3
Disk 0
Disk 1
Disk 2
Disk 3
RAID 6
A1 B1 C1 Dp Eq
A2 B2 Cp Dq E1
A3 Bp Cq D1 E2
Ap Bq C2 D2 E3
Aq B3 C3 D3 Ep
Disk 0
Disk 1
Disk 2
Disk 3
Disk 4

Mixed RAID types principle





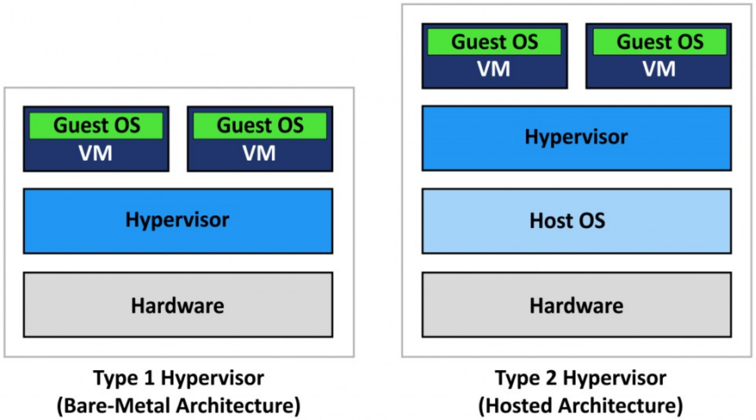
Virtual machine approach



Type 1 Hypervisor
(Bare-Metal Architecture)

Type 2 Hypervisor
(Hosted Architecture)

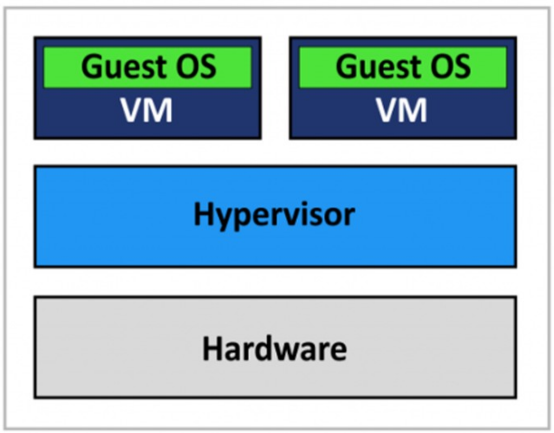
Virtual machine approach



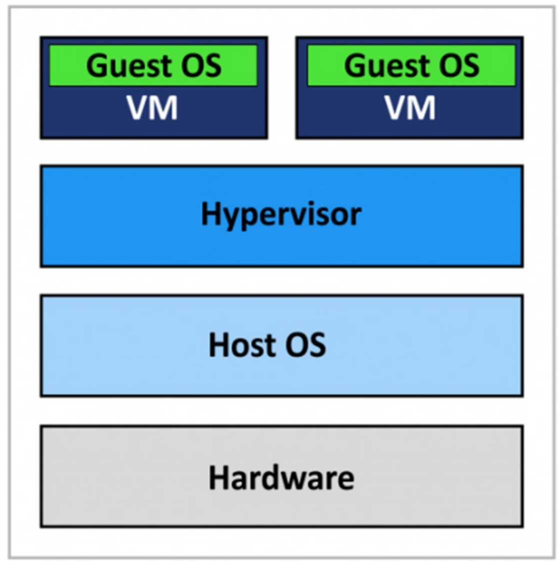
Vmware ESXi
Microsoft Hyper-V Server
Citrix XenServer
KVM

VMware Workstation
Virtualbox

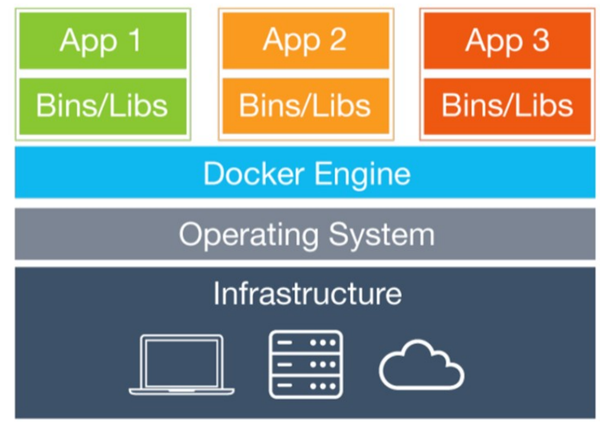
VM vs Container



Type 1 Hypervisor
(Bare-Metal Architecture)

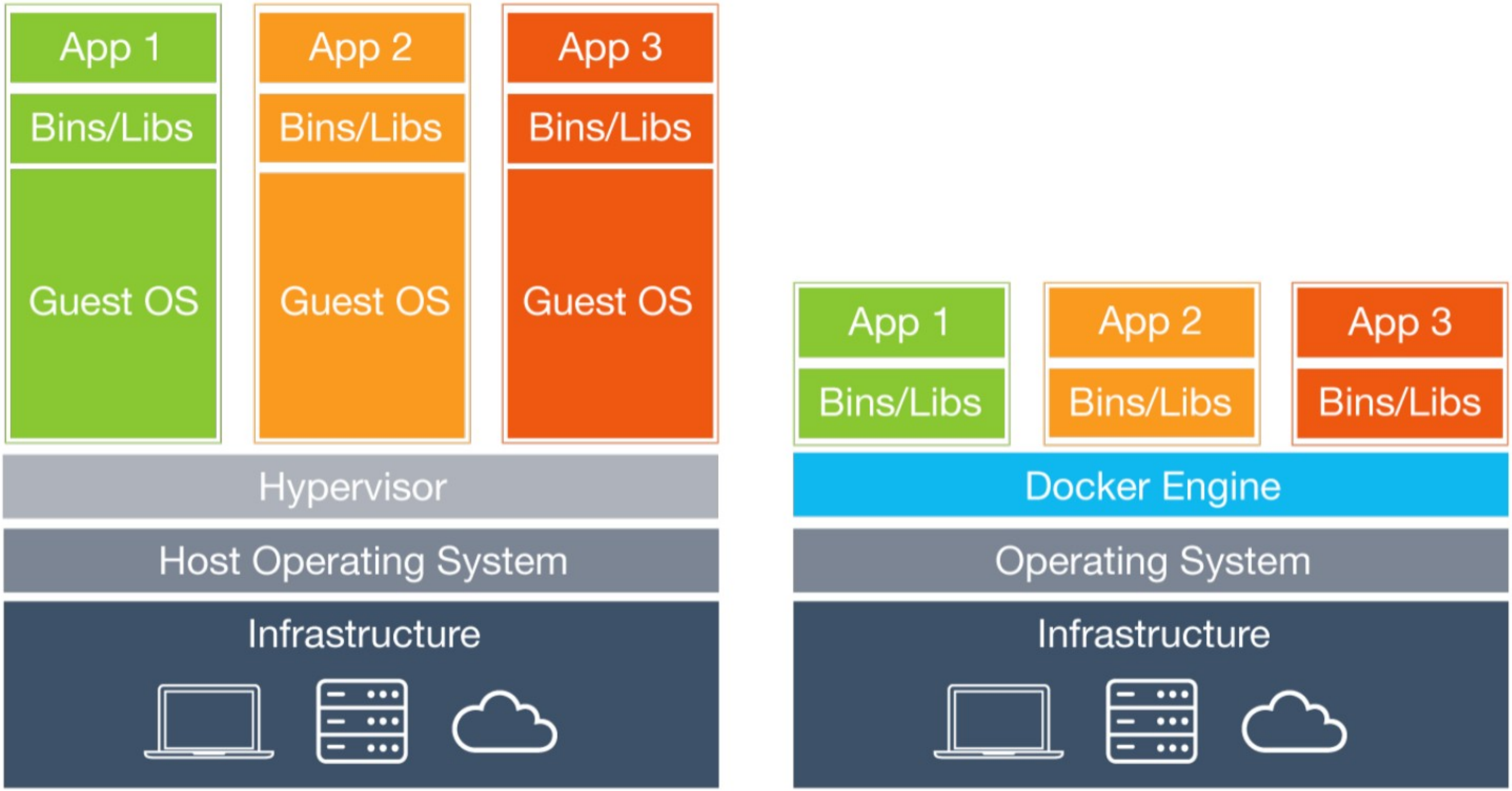


Type 2 Hypervisor
(Hosted Architecture)

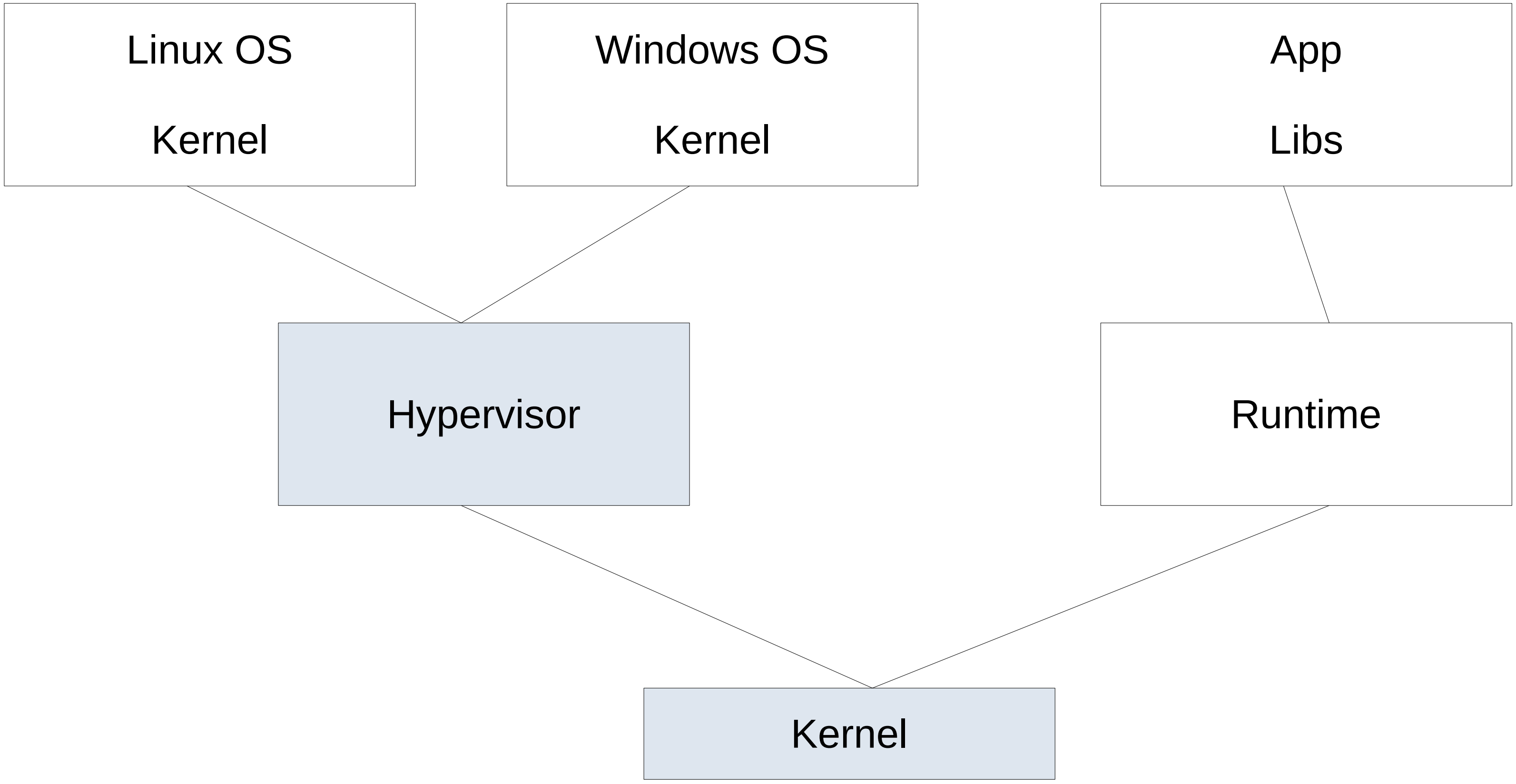


Docker

VM vs Container



VM vs Container



Containers

Linux containers are ordinary processes, which isolated by Linux kernel tools:

* Namespaces for resource isolation
* Control Groups for resource constraints
* Permissions, Capabilities, SELinux, AppArmor and other Linux security constraints

All Linux processes are started as containers.

* `cat /proc/PID/cgroup will` see the Cgroups the process is in
* `cat /proc/self/attr/current` will show SELinux labels
* `ls /proc/PID/ns` will show a list of namespaces the process is in

Container different is that it is started from a container image. So, container runtime is the operating system part that manages the cgroups, SELinux labels and namespaces and starts the container from an image

Container is

* A container is a running instance of an image
* The image contains the application code, language runtime and libraries
* External libraries such as libc are typically provided by the host operating system, but in a container is included in the images
* The container image is a read-only instance of the application that just needs to be started
* While starting a container, it adds a writable layer on the top to store any changes that are made while working with the container

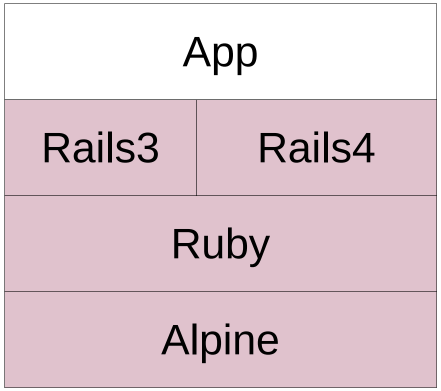
What is an Image?

- A container image is a TAR file that combines the following
  - The container root file system: a directory that looks like the standard root of the operating system, but presented as a mount namespace
  - Metadata: a JSON file that specifies how to run that root file system, including all settings required to get to a functional container (entrypoint, environment variables and more)
- Container images are layered: you can install additional contend, add a new JSON file and store the differences in a new TAR file
- Images are standardized by the OCI

- Container images are typically shared though public registries, or by sharing mechanisms to build the easily, such as Dockerfile
- The Docker container image format has become the de facto standard image format
- Open Container Initiative (OCI) has standardized the Docker container image format

Container Image Layers

- Docker images are made up of a series of filesystem layers
- Each layer adds, removes or modifies files form the preceding layer in the filesystem
- This is called an overlay filesystem, and different overlay filesystems exist, like aufs, overlay and overlay2
- By using these different image layers, and pointing to other image layers in a smart way, it’s easy to build container images that have support for multiple versions of vital components
- Apart from the different layers, container images have a container configuration file that provides instructions on how to run the container

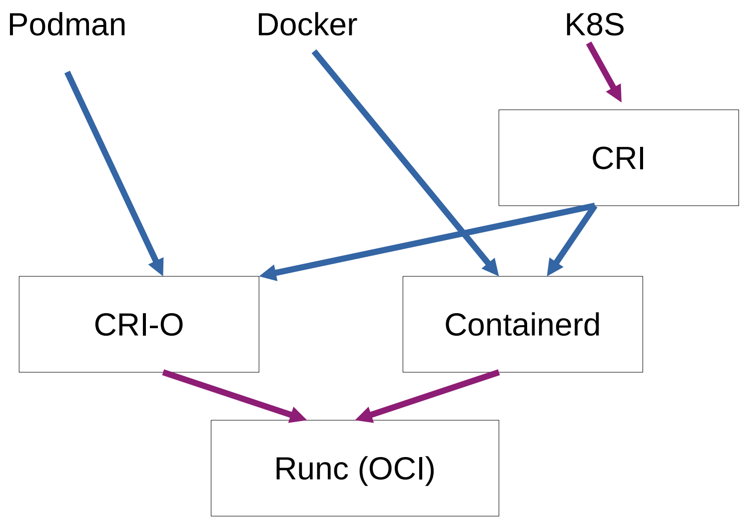


Current Linux Namespaces

- **pid**: isolation for processes
- **user**: presents isolated users only existing inside the namespace
- **mnt**: the modern alternative to the chroot jail
- **net**: provides isolated networking
- **ipc**: inter-process communication, used for sharing memory
- **uts**: UNIX time sharing, which allows processes to have their own hostname and domain name
- **cgroup**: providing isolated root directories for each cgroup
- **time**: used to virtualize the clock of a system

Container Engines. Container Runtimes

- In the early days of containers (2014-2015), containers were started and managed by a container engine
- A container engine offers multiple components
  - A core component that runs the containers
  - Optionally a daemon that controls the containers
  - A command line interface
- Common container engines include and included
  - Docker
  - LXC
  - Podman
  - System-nspawn



- While container engines developed further, container runtimes were split off as separate projects
- By splitting of container runtimes into different open source projects, it became the easier to focus on developing and standardizing them
- Containerd became the common runtime in Docker environments
- CRI-o is the common runtime in Podman environments

Storage Drives

- Storage drives write data in the writable layer of the container
- This way of data writing is not persistent and performance is not good
- The writable layer is created when running any container
- Storage drivers handle how the layers interact with one another
- For local use, the **local** driver and the **sshfs** driver are available
- For more enterprise level use, drivers are available through the Kubernetes or Swarm orchestration layer

- Different storage drivers are available for local use:
  - aufs
  - overlay
  - overlay2
  - btrfs
  - zfs
  - devicemapper
- Storagedrivers are set in /etc/docker/daemon.json (not that file may not yet exist)

```
{
    “storage-driver”: “devicemapper”

}
```

Copy on Write Strategy

- If a container changes files in a lower layer, the file is copied from the lower layer to the upper layer and modified while it is there, which is referred to as Copy on Write (CoW)
- Using the CoW strategy guarantees that only modifications are stored in the writable filesystem layer, which is very storage efficient
  - If multiple containers are started based on one image, only the differences between the container writable layers is stored
- For write-intense applications, this strategy is not so efficient, and it’s better to use external storage

Copy on Write Strategy

- Create a directory cow-test: **mkdir cowtest; cd cowtest**
- Create a file hello.sh:
  - #!/bin/sh
  - echo “hello”
- Make it executable: **chmod +x hello.sh**
- Create the file Dockerfile.base:
  - FROM ubuntu:20.04
  - COPY . /app
- Create the file Dockerfile:
  - FROM ssau/base-image:0.1
  - CMD /app/hello.sh
- Build the base image: **docker build -t ssau/base-image:0.1 -f Dockerfile.base .**
- Build the second image: **docker build -t ssau/final-image:0.1 -f Dockerfile .**
- Check image sizes to see the same size is reported for both: **docker image ls**
- Use the image ID as displayed by the **docker history** command to investigate what has happened in each of the layers:
  - **docker history <ID-of-base-image>**
  - **docker history <ID-of-final-image>**
- Notice that the difference is only in the top layer of the second image, for the rest the images are the same and the disk space is used once only

Bind Mounts

- Bind mount storage is based on Linux bind mounts
- The container mounts a directory or file from the host OS into the container
- If the host directory doesn't exist, it will be created, but only if the **-v** option is used
- The host OS still fully controls access to the file
- Docker commands cannot be used to manage the bind mount
- The **-v** option as well as the **--mount** option can be used to create the bind mount
  - **-v** is the old option, which combined multiple arguments in one field
  - **--mount** is newer and more verbose

Bind mounts work when the host computer contains the files that need to be accessible in the containers

- Configuration files
- Access to source code
- Log files

Using **--mount**

- **mkdir bind; docker run –rm -dit –name=bind1 –mount type=bind, source=”$(pwd)”/bind1,target=/app nginx:latest**

Using **-v**

  - **docker run --rm -dit --name=bind2 -v “$(pwd)”/bind2:/app nginx:latest**

Use **docker inspect <containername>** to verify

Bind Mounts - example

- **mkdir bind1**
- **echo bound > bind1/bond.txt**
- **docker run --rm -dit --name=bind1 --mount type=bind,source=”$(pwd)”/bind1,target=/app nginx:latest**
- **docker ps**
- **docker exec bind1 ls -l /app**
- We can see that file bond.txt exist
- **ls bind1/**
- We can see that file in local directory

- **echo hello > bind1/hello.txt**
- **docker exec bind1 ls -l /app**
- **docker exec bind1 touch /app/bye.txt**
- **ls -l bind1/**
- We can see the bye.txt file

- **docker run --rm -dit --name=bind2 -v “$(pwd)”/bind1:/app nginx:latest**
- **docker exec bind2 ls -l /app**
- Same but with -v

- **docker inspect bind1 | less**
- To see that mount is available in Mount section

Volumes

Volumes are the preferred way to work with persistent data as the volume survives the container lifetime

- Multiple containers can get simultaneous access to the volumes
- Data can be stored externally
- Volumes can be used to transition data from one host to another
- Volumes are supported for Linux as well as Windows containers
- Volumes live outside of the container and for that reason don't increase container size
- Volumes use drivers to specify how storage is accessed. Enterprise-grade drivers are available in Docker Swarm – not standalone

Demo: Working with Volumes

- **docker volume create myvol** creates a simple volume that uses the local file system as the storage backend
- **docker volume ls** will show the volume
- **docker volume inspect myvol** shows the properties of the volume
- **docker run -it --name voltest --rm --mount source=myvol,target=/data nginx:latest /bin/sh** will run a container and attach to the running volume
- From the container, to create some files in **/data** use **cp /etc/hosts /data; touch /data/testfile; ctrl-p ctrl-q**
- To view files in myvol **sudo -l; ls /var/lib/docker/volumes/myvol/_data/** or for Windows **dir \\wsl$\docker-desktop-data\data\docker\volumes\myvol_data**
- **docker run -it --name voltest2 --rm --mount source=myvol,target=/data nginx:latest /bin/sh**
- From the second container: **ls /data; touch /data/newfile; ctrl-p, ctrl-q**
- After that you can view file in myvol again **sudo -l; ls /var/lib/docker/volumes/myvol/_data/** or for Windows **dir \\wsl$\docker-desktop-data\data\docker\volumes\myvol_data**

Multi-container Volume Access

- To create a file in a volume, nothing special is needed and the volume can be accessed from multiple containers at the same time
- To simultaneously access files on volumes from multiple containers, a special driver is needed
- Recommended: use the **readonly** mount option to protect from file locking problems
  - **docker run -it --name voltest3 --rm --mount source=myvol,target=/data,readonly nginx:latest /bin/sh**
- Enterprise-grade drivers are available in Docker Swarm, or are provided though Kubernetes
- For non-orchestrator use, consider using the local driver NFS type

  Demo: Configuring an NFS-based Volume
- **sudo apt install nfs-serer nfs-common** for Windows see **NFS-server installation**
- **sudo mkdir /nfsdata** and **sudo touch /nfsdata/file1** and **sudo touch /nfsdata/file2**
- To add rights on folder for everyone **sudo vim /etc/exports**
  - **/nfsdata   *(rw,no_root_squash)**
- **sudo chown nobody:nogroup /nfsdata**
- **sudo systemctl restart nfs-kernel-server**
- To check that nfs-folder mountable **showmount -e localhost**
- To mount nfs into container **docker volume create --driver local --opt type=nfs --opt o=addr=127.0.0.1,rw --opt device=:/nfsdata nfsvol**
- **docker volume ls**
- To see details about volume **docker volume inspect nfsvol**
- To see volume from inside of container **docker run -it --name nfstest --rm --mount source=nfsvol,target=/data nginx:latest /bin/sh**
- **ls /nfsdata** to check that some files are in /nfsdata

Docker Essentials

https://docs.docker.com/get-started/