Operating system level virtualization (aka containers)

DSS Lecture 12.2

Distributed software systems
CdLM Informatica - Università di Bologna

Questions

How can we make fast deployments in the cloud?

Deploying a VM can be expensive (containers are cheaper)

Some important topics:

- What is a container and how it differs from a VM
- The origins of container technologies
- Why containers can be moved from one environment to another
- What a container repository is and how to store container images in them

Agenda

- Containers
- LXC
- Docker
- Control groups

The benefits of virtualization

Hypervisors provide the following benefits:

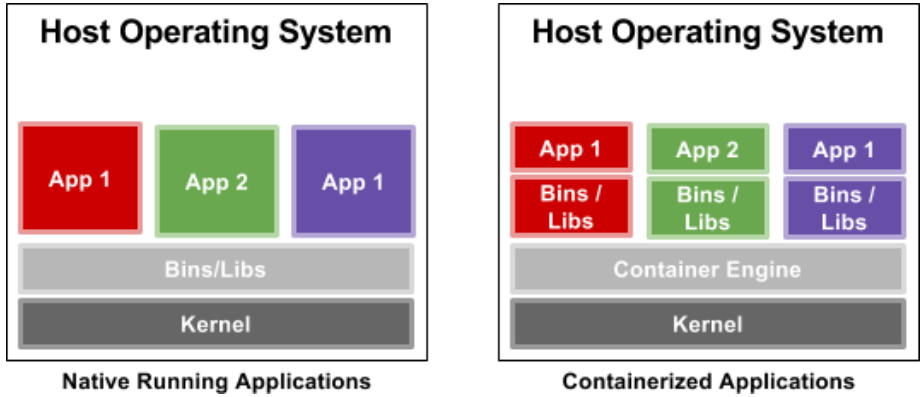
- Ability to run different operating systems on the same physical server
- Granular control over resource allocation
- Process isolation – a kernel panic on the virtual machine will not effect the host OS
- Separate the network stack and the ability to control traffic per each virtual machine
- Reduce the initial and operating costs, by simplification of data center management and better utilization of available server resources

However, running an entire operating system on a virtual machine, just to achieve a level of confinement for a single server, is not the most efficient allocation of resources

Containers

Containers, or otherwise known as operating system level virtualization, are a lightweight approach to virtualization that only provides the bare minimum that an application requires to run

- Typically a container includes
  - Application
  - Dependencies
  - Libraries
  - Binaries
  - Configuration files

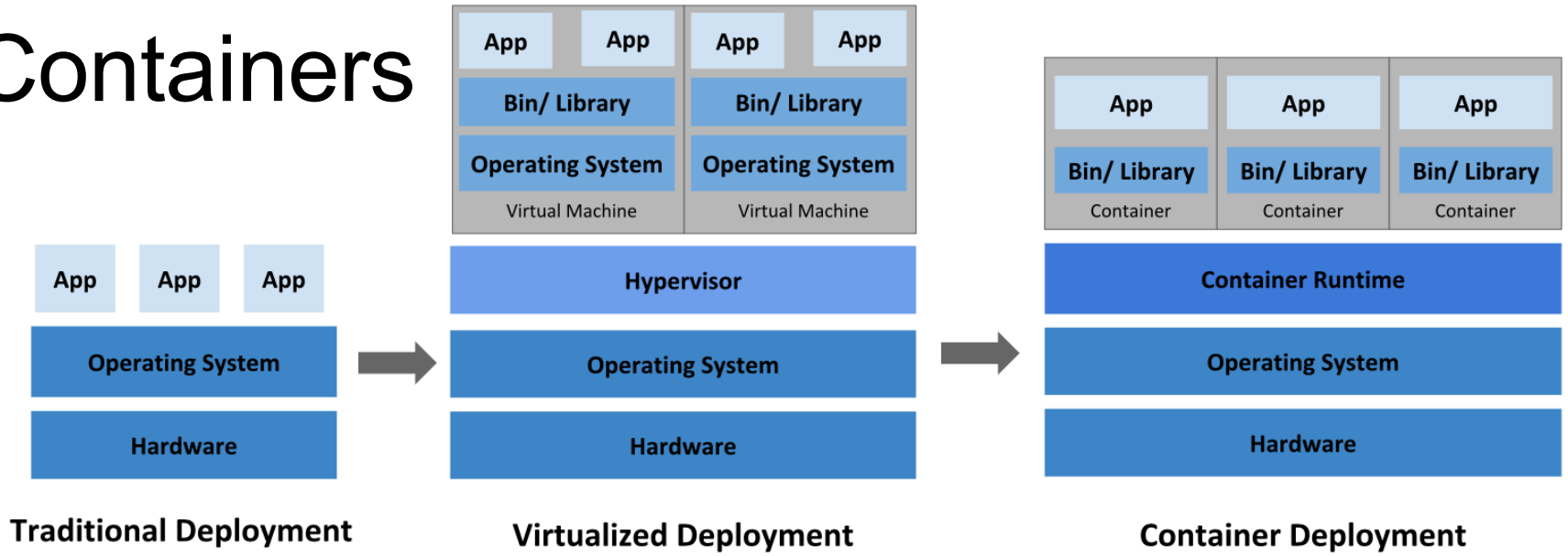


Containers

Containers can provide virtualization at either the operating system level or the application level. Some possibilities with containers are:

- Provide a complete operating system environment that is sandboxed (isolated)
- Allow packaging and isolation of applications with their entire runtime environment
- Provide a portable and lightweight operating environment
- Help to maximize resource utilization in data centers
- Aid different development, test, and production deployment workflows

Containers



A container can be defined as a single operating system image, bundling a set of isolated applications and their required resources so that they run separated from the host machine.

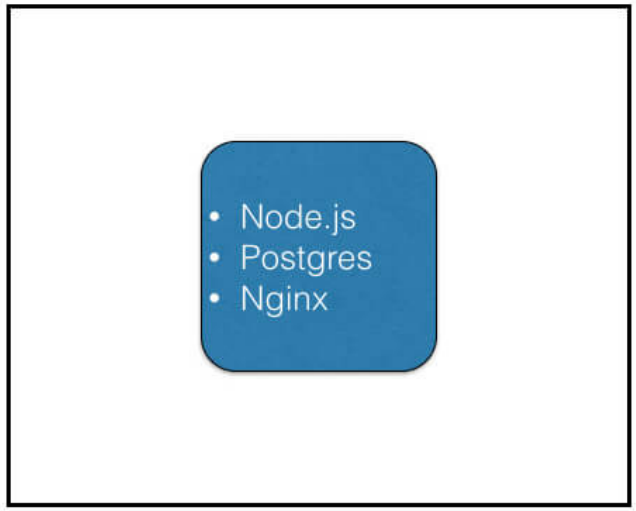
There may be multiple such containers running within the same host machine.

Two types of containers

Containers can be classified into two types:

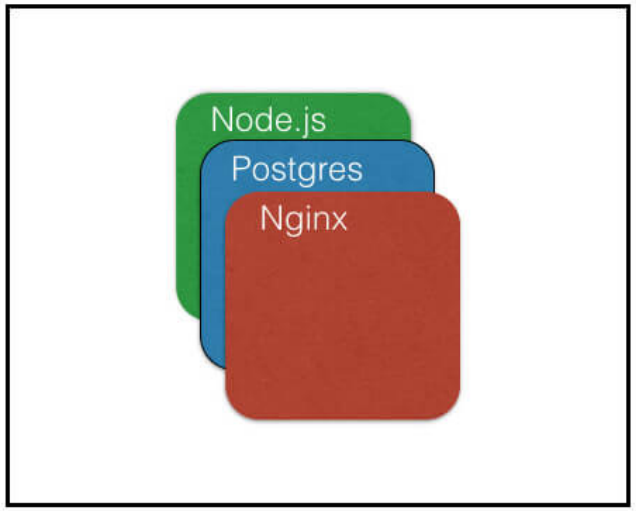
- Operating system level: An entire operating system runs in an isolated space within the host machine, sharing the same kernel as the host machine.
- Application level: An application or service, and the minimal processes required by that application, runs in an isolated space within the host machine

Two types of containers



OS containers

- Meant to used as an OS - run multiple services
- No layered filesystems by default
- Built on cgroups, namespaces, native process resource isolation
- Examples - LXC, OpenVZ, Linux VServer, BSD Jails, Solaris Zones



App containers

- Meant to run for a single service
- Layered filesystems
- Built on top of OS container technologies
- Examples - Docker, Rocket

https://blog.risingstack.com/operating-system-containers-vs-application-containers/

Containers

Containerization differs from traditional virtualization technologies and offers many advantages over traditional virtualization:

- Containers are lightweight compared to traditional virtual machines.
- Unlike containers, VM require emulation layers (either software or hardware), which consume more resources and add additional overhead.
- Containers share resources with the underlying host machine, with user space and process isolations.
- Due to the lightweight nature of containers, more containers can be run per host than virtual machines per host.
- Starting a container happens nearly instantly compared to the slower boot process of VMs.
- Containers are portable and can reliably regenerate a system environment with required software packages, irrespective of the underlying host operating system.

Example: LXC

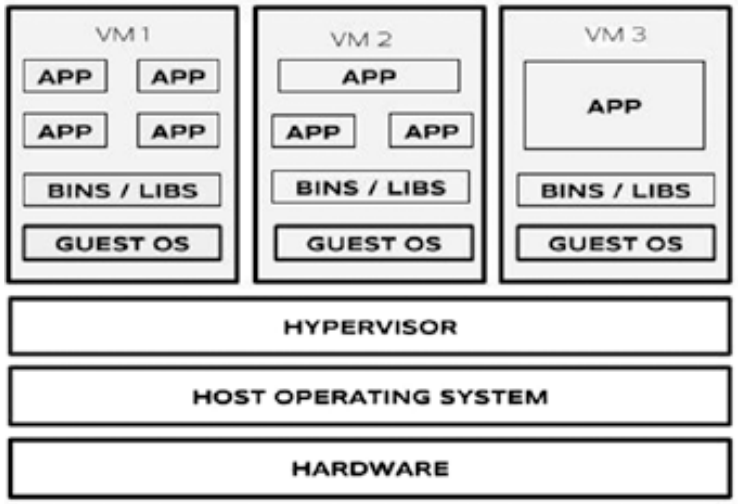
Over the last decade, various improvements to the Linux kernel were made to allow for functionality similar to hypervisors, but with less overhead – most notably the kernel namespaces and cgroups.

The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel.

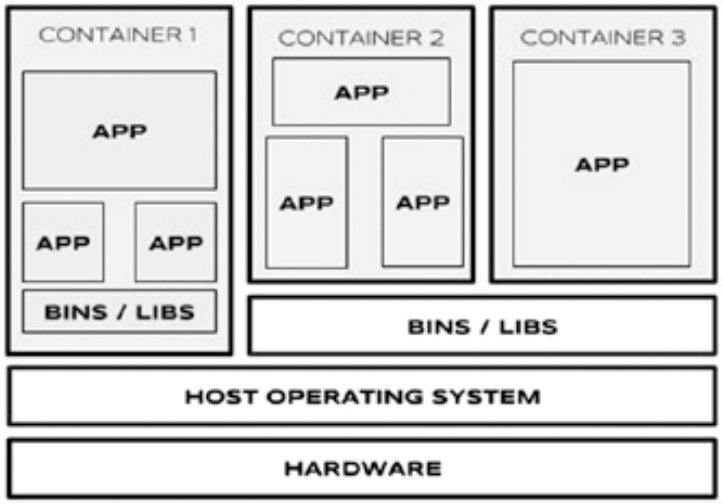
The main benefits of using LXC include:

- Lesser overheads and complexity than running a hypervisor
- Smaller footprint per container
- Start times in the millisecond range
- Native kernel support

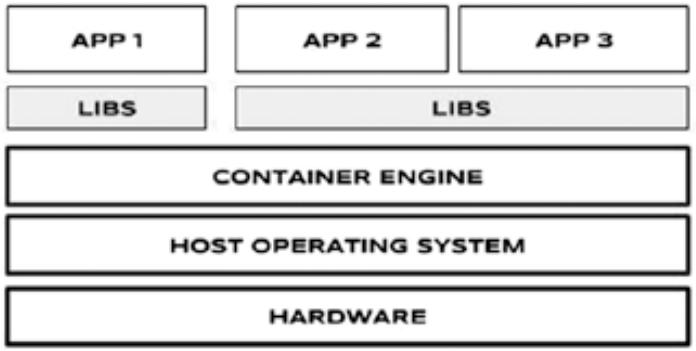
It is worth mentioning that containers are not inherently as secure as having a hypervisor between the virtual machine and the host OS



Virtual Machines on Host



Linux Containers (lxc) or Operating System Level Containers



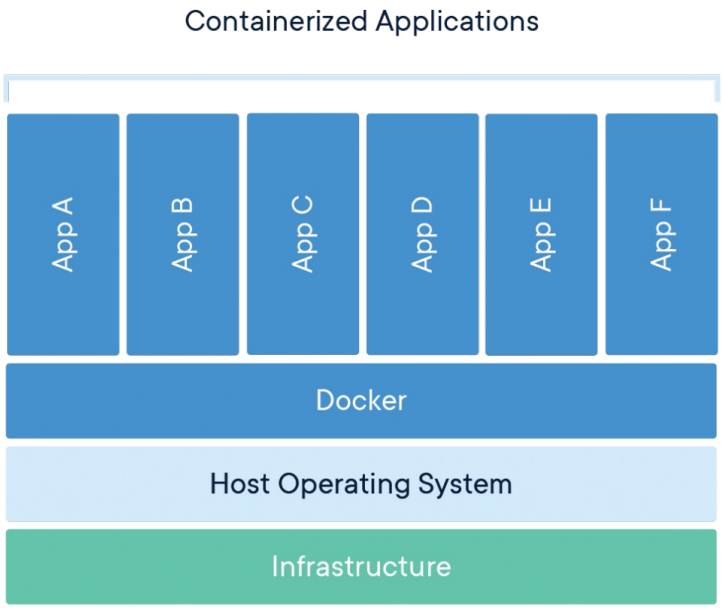
Application Level Containers

LXC use cases

- Moving or upgrading servers. ...
- Testing and rolling out changes. ...
- No downtime for upgrades or changes. ...
- Distribution and rapid provisioning. ...
- Run multiple versions of applications and web stacks. ...
- Portable cloud instances.
- Easy backups
- Test applications on multiple Linux distributions
- Easy file system acce**ss**

https://archives.flockport.com/lxc-usecases/

Example: Docker



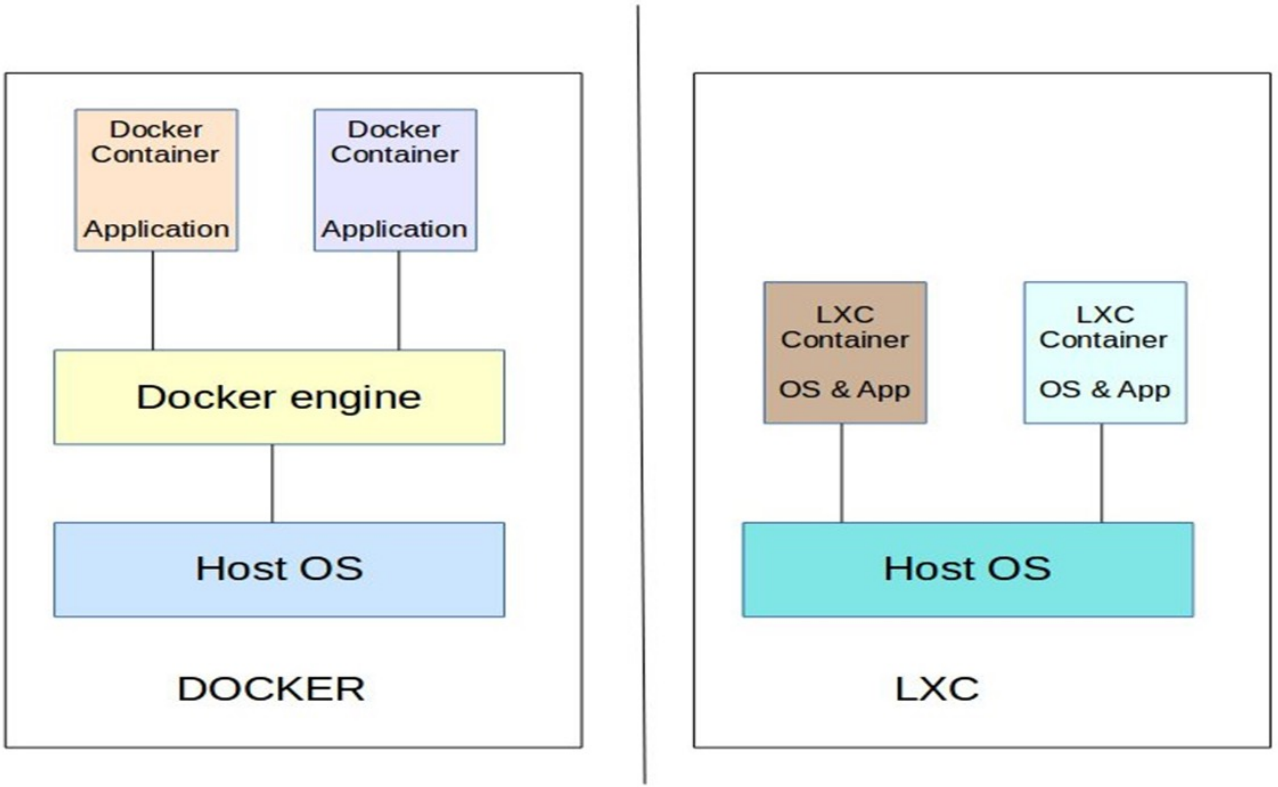
- A container is a unit of software that packages up code and all its dependencies, so the application runs quickly and reliably when ported from one computing environment to another.
- A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings

Docker use cases

- Simplifying Configuration. ...
- Code Pipeline Management. ...
- Developer Productivity. ...
- App Isolation. ...
- Server Consolidation. ...
- Debugging Capabilities. ...
- Multi-tenancy.
- Rapid deployment

https://www.airpair.com/docker/posts/8-proven-real-world-ways-to-use-docker

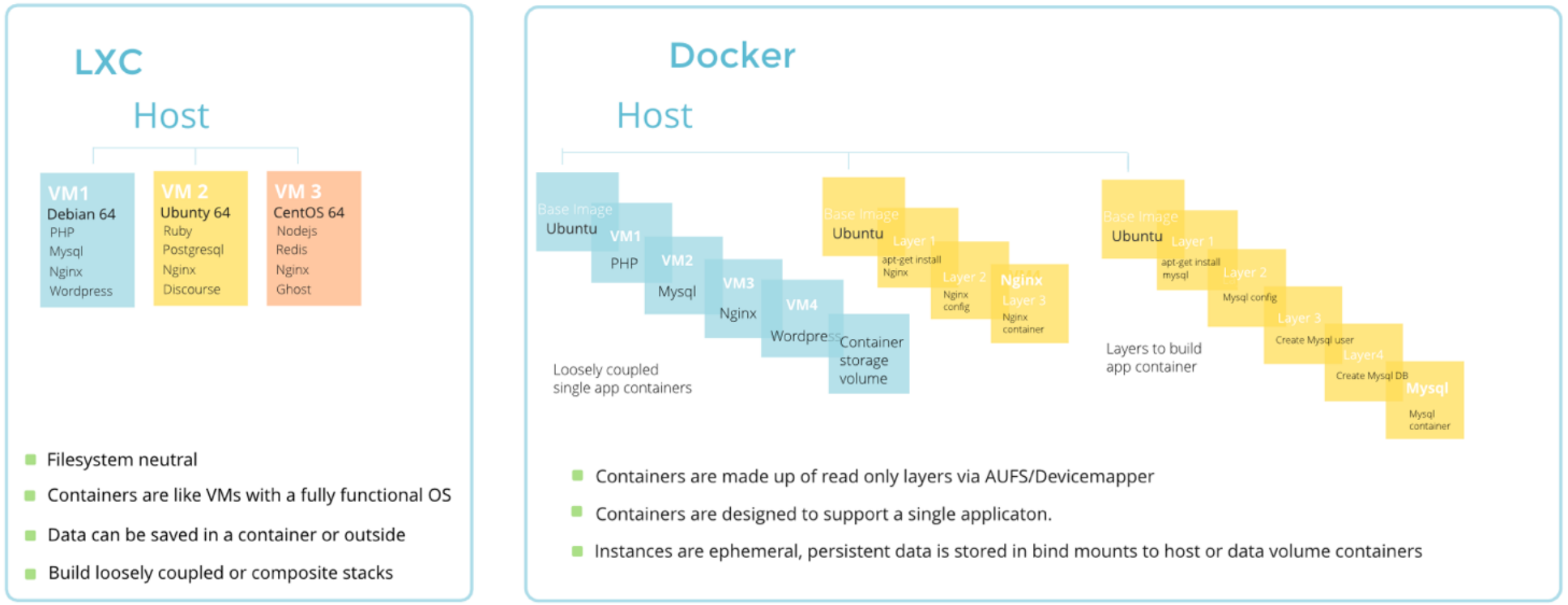
LXC vs Docker



https://bobcares.com/blog/lxc-vs-docker/

Key differences between LXC and Docker

flockport

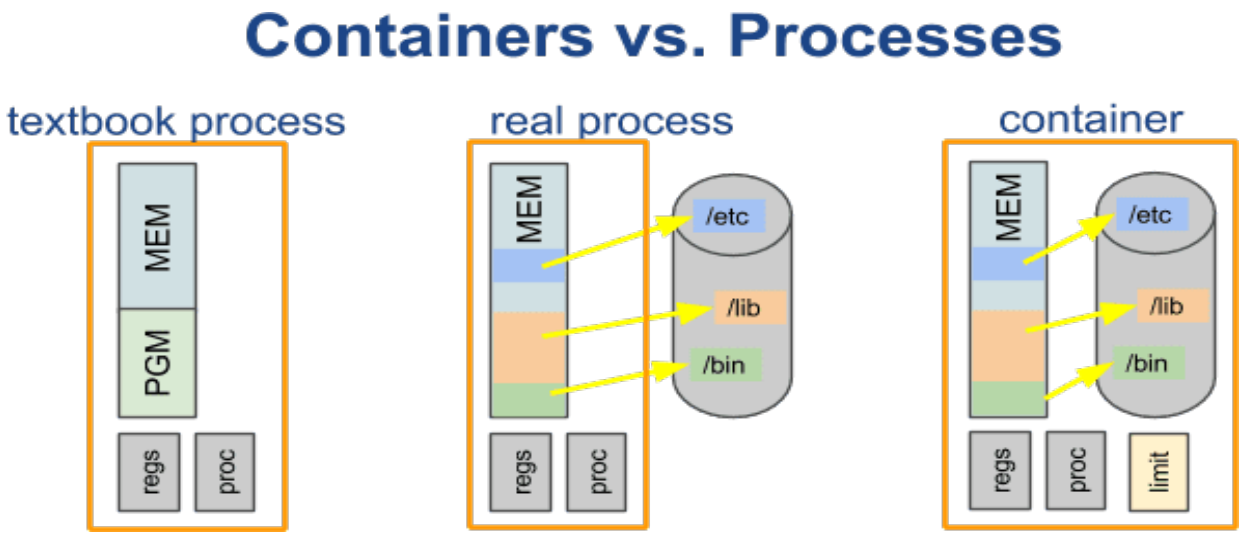


flockport.com

https://archives.flockport.com/lxc-vs-docker/

Containers vs. Processes

- Containers actually are processes with their full environment.
- Every «textbook» process has its own address space, program, CPU state, and process table entry.
  - Actually a real process has its memory mapped from the filesystem into its process address space and often consists of dozens of shared libraries



Why do we need a container?

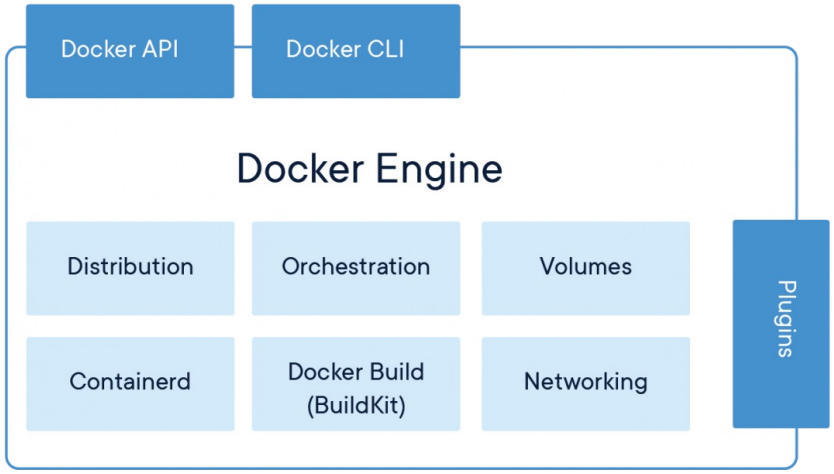
- Isolation:
  - Containers allow the deployment of one or more applications on the same physical machine, each requiring exclusive access to its respective resources.
    - For instance, multiple applications running in different containers can bind to the same physical network interface by using distinct IP addresses associated with each container.
- Security:
  - Network services can be run in a container, which limits the damage caused by a security breach or violation.
    - For instance, an intruder who successfully exploits a security hole on one of the applications running in that container is restricted to the set of actions possible within that container.

Why do we need a container?

Controlling transparency:

- Containers provide the system with a virtualized environment that can hide or limit the visibility of the physical devices or system's configuration underneath it.
- The general principle behind a container is to avoid changing the environment in which applications are running, with the exception of addressing security or isolation issues.

Example: Docker Engine



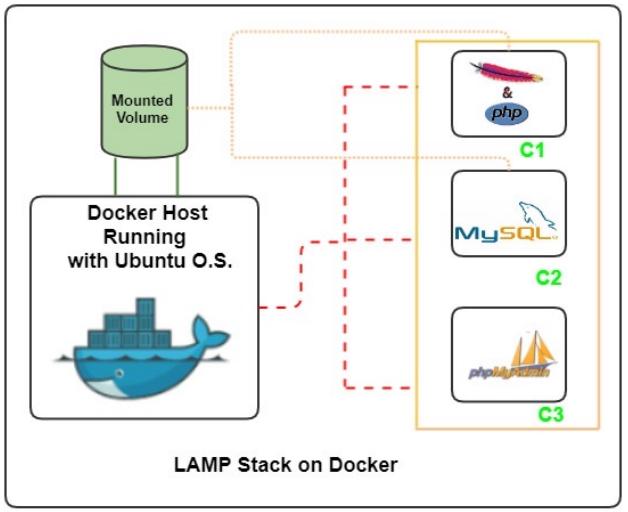
- Docker Engine is a container runtime that runs on various Linux (CentOS, Debian, Fedora, Oracle Linux, RHEL, SUSE, and Ubuntu) and Windows Server operating systems.
- Docker creates a universal packaging approach that bundles up all application dependencies inside a container which is then run on Docker Engine.
- Docker Engine enables containerized applications to run anywhere consistently on any infrastructure, solving “dependency hell” for developers and operations teams

Container engine

- Container engines use several Linux features to provide isolation while sharing the operating system.
- Linux control groups set resource utilization limits, and Linux namespaces prevent a container from seeing other containers.
- This sharing of the operating system gives us one source of performance improvement when transferring VM images.
- As long as the target machine has a container engine running on it, there is no need to transfer the operating system as part of the container image.
- Since modern operating systems are on the order of 1 GB(yte) in size, this saves a substantial amount of time.

Layers in container images

- The second source of performance improvement is the use of layers in container images.
- consider the construction of a container to run the LAMP stack, and where the image is built in layers.
- LAMP is Linux, Apache, MySQL, and PHP, a widely used stack for constructing web applications



https://github.com/amarsingh3d/Docker-LAMP

Building a container for a LAMP stack

- The process begins with creating a container image containing a Linux distribution, eg. Ubuntu. This image can be downloaded from a library using the container management system.
- Once you have created the Ubuntu container image and identified it as an image, you execute it, i.e., make a container, and you use that container to load Apache, using features of Ubuntu.
- Now you exit the container and inform the container management system that this is a second image. You execute this second image and load MySQL. Again, you exit the container and give the image another name.
- Repeating this process one more time and loading PHP, you end up with a container image holding the LAMP stack.
- Because this image was created in steps and you told the container management system to make each step an image, the final image is considered by the container management system to be made up of layers.

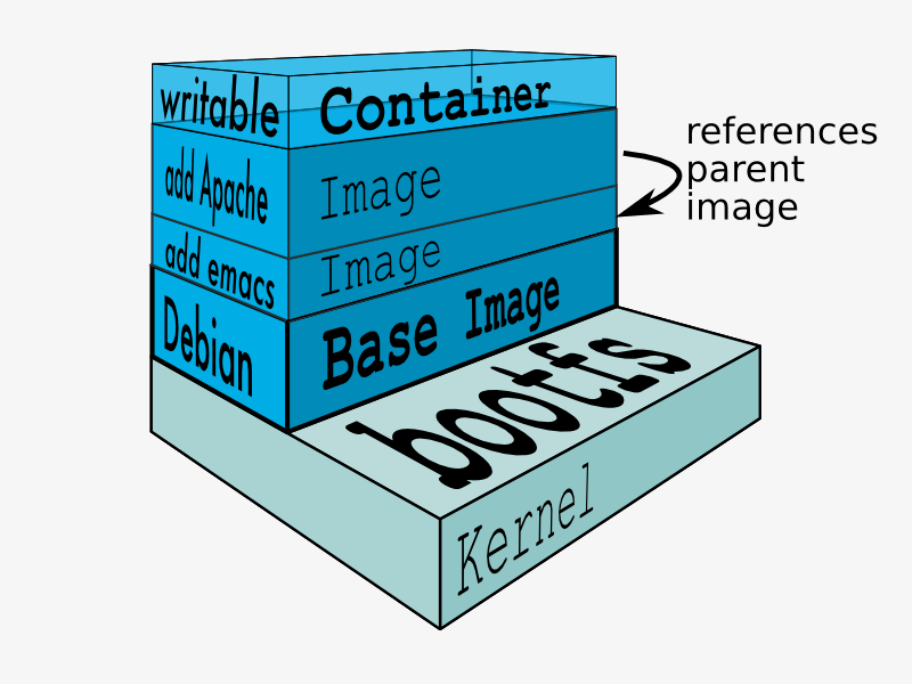
https://docs.docker.com/storage/storagedriver/

Dockerfile

- Docker can build images automatically by reading the instructions from a Dockerfile.
- A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.
- Using docker build a programmer can create an automated build that executes several command-line instructions in succession.

https://docs.docker.com/engine/reference/builder/

Layers of container



- Any RUN commands you specify in the Dockerfile creates a new layer for the container.
- In the end when you run your container, Docker combines these layers and runs your containers.
- Layering helps Docker to reduce duplication and increases the re-use.
- This is very helpful when you want to create different containers for your components. You can start with a base image that is common for all the components and then just add layers that are specific to your component.
- Layering also helps when you want to rollback your changes as you can simply switch to the old layers, and there is almost no overhead involved in doing so

Building a container - 2

- Now you move the LAMP stack container image to a different location for production use. The initial move requires moving all the elements of the stack, so the time it takes is the time to move the total stack.
- Suppose, however, you update PHP to a newer version and move this revised stack into production.
- The container management system knows that only PHP was revised and moves only PHP layer of the image.
- This saves the movement of the rest of the stack.

Building a container - 3

- Since changing a software component within an image happens much more frequently than initial image creation, placing a new version of the container into production becomes a much faster process than it would be using a VM.
- Whereas loading a VM takes on the order of minutes, loading a new version of a container takes on the order of microseconds or milliseconds.

Building a container - 4

- You can script the creation of a container image through a file. This file is specific to the tool you are using to create the container image. Such a file allows you to specify what pieces of software are to be loaded into the container and saved as an image.
- Using version control on the specification file ensures that each member of your team can create an identical container image and modify the specification file as needed.

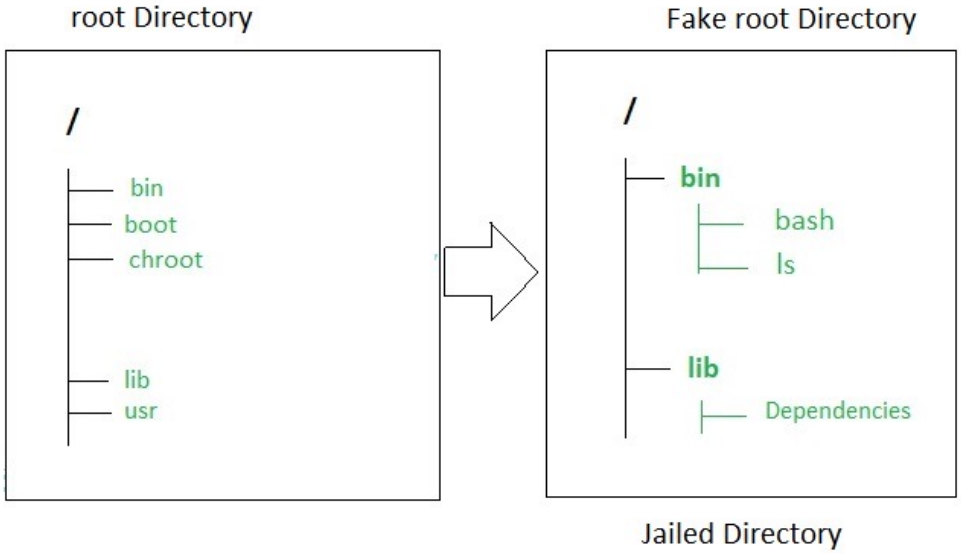
History

- **1979:**
  - Development of Unix Version 7 and **chroot** on Unix operating systems
  - Chroot is an operation that changes the apparent root directory for the current running process and its children.

- **2000:**
  - FreeBSD 4.0 was released with a new command, called jail to allow to easily and securely create a separation between the processes.
  - It also allowed isolated resource sharing between processes.

Jailed directory

- Every process has a current working directory called **root directory**.
- The `chroot` command in Linux/Unix is used to change the root directory.
- It changes the root directory for currently running processes as well as its child processes.
- A process/command that runs in such a modified environment cannot access files outside the root directory.
- This modified environment is known as "chroot jail" or **"jailed directory"**. Some root users and privileged processes are allowed to use the `chroot` command.



History

- **2004:**
  - Sun released Solaris 10, which included **Solaris Containers**, and later evolved into Solaris Zones.
  - This was the first major commercial implementation of container technology and is still used today to support commercial container implementations.

- **2005:**
  - **Open VZ** set a complete vision but it was -like implementation with no speed improvements; it was based on commands ***chroot*** and ***rlimit***.
  - The main problem with openVZ with that they changed the linux kernel itself which by time drifted more and more from the standard releases.

History

- **2006:**
  - Engineers at Google (primarily Paul Menage and Rohit Seth) started to work on a new feature under the name "process containers".
  - That was later named to control groups. It is a part of the Linux kernel since January 2008 release 2.6.24
  - This new feature (cgroups) motivated the existence of containers.

- **2008:**
  - LXC (Linux Containers) established as the *first Cgroups based framework* to support containers.
  - LXC provided a middleware component that allowed the idea of containers to exist.

History

- **2013:**
  - **Docker** was released as a framework to create, monitor and maintain containers built on LXC.

- **2014:**
  - Google announces that it has used containers as a concept since 2006 and released its open source project kubernetes that allows container clustering.
- **2015:**
  - runC container developed by Open Container Initiative (OCI).

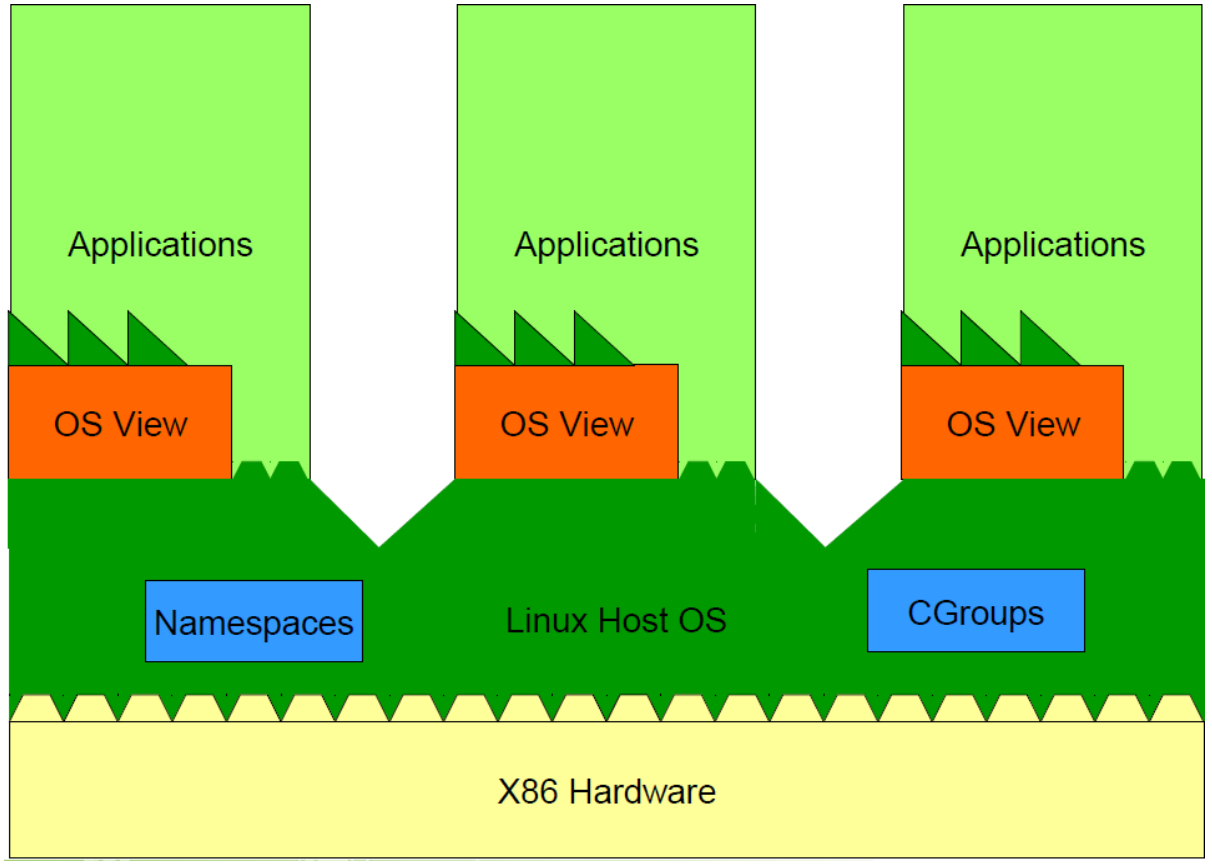
Operating-System Level Virtualization

- In between System VM and Process VM
- Not System VM:
  - Cannot choose OS
- Not Process VM:
  - Multiple processes, not isolated

- As if multiple instances of the same OS are running on the same machine
  - Example: Docker, LXC, rkt, runC, systemd-nspawn OpenVZ, Jails, Zones

Lightweight process virtualization

- A process which gives the user an illusion that he runs a Linux operating system
- You can run many such processes on a machine: they all in fact share a single Linux kernel which runs on the machine.



Lightweight process virtualization

- Opposed to hypervisor solutions (Xen) where you run another instance of the kernel
- The idea is not really a new paradigm - we have Solaris Zones and BSD jails already several years ago
- Advantages of Hypervisor-based VMs:
  - You can create VMs of other operating systems (windows, BSDs).
  - Security – Though there were cases of security vulnerabilities which were found and required installing patches to handle them (like VENOM)

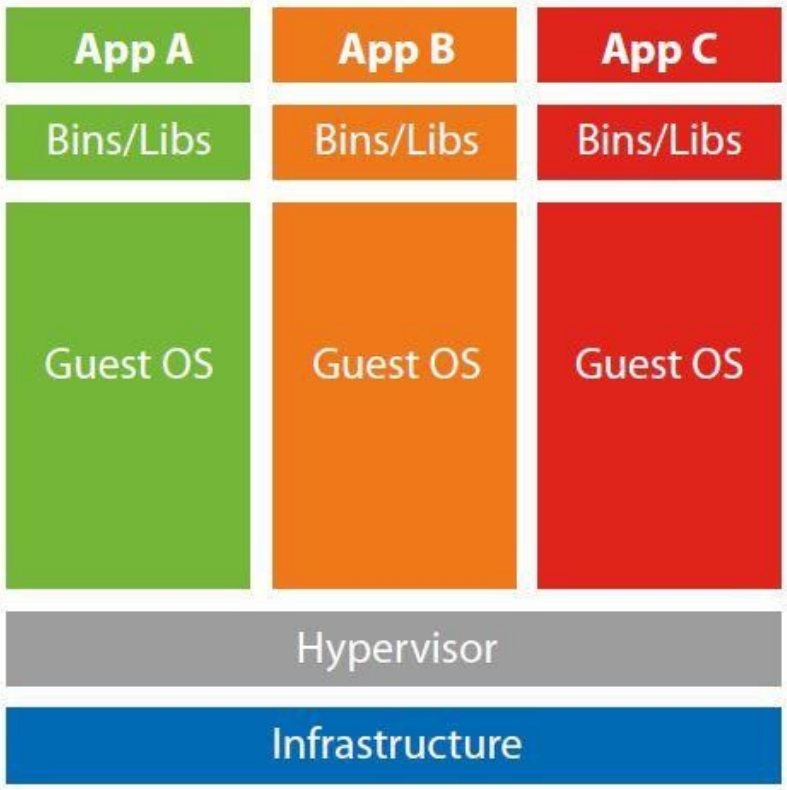
Containers versus Hypervisor-based VMs

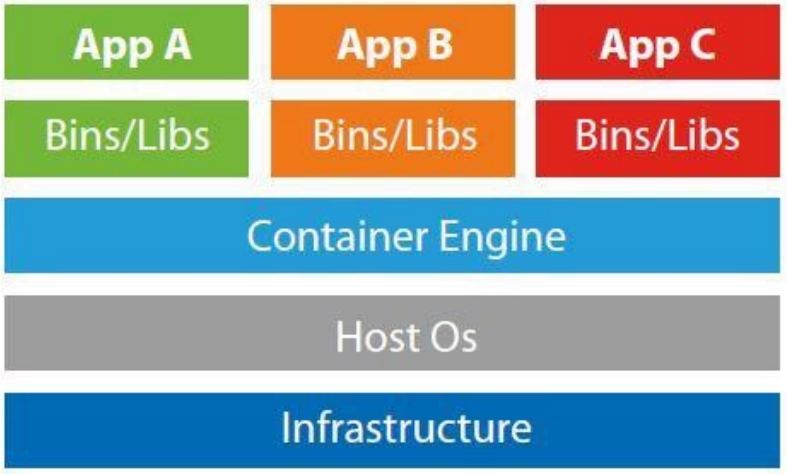
Containers – Advantages:

- **Lightweight**: occupies less resources (like memory) significantly than a hypervisor.
- **Density**: you can install many more containers on a given host than VMs.
- **Elasticity**: startup time and shutdown time is much shorter, almost instantaneous.
  - Creation of a container has the overhead of creating a Linux process, which can be of the order of milliseconds, while creating a VM based on XEN/KVM can take seconds.

The lightness of the containers in fact provides both their density and their elasticity.

VMs vs. Containers





Scaling with containers

- Because containers are executing on nodes that are hosted on VMs, scaling containers involves two different decision types.
- When scaling VMs, a monitor decides that additional VMs are required, and then allocates a new VM and loads it with the appropriate software. Scaling containers means making a two-level decision.
- First, decide that an additional container (or pod) is required for the current workload.
- Second, decide whether the new container or pod can be allocated on an existing VM or that an additional VM must be allocated.

Scaling with containers

- The need to create a new VM means that the container scaling service must be integrated with the cloud provider's VM allocation service.
- This is not, generally, a problem since the cloud providers have easy mechanisms for programmatic allocation of new VMs.
- It does mean, however, that you or your organization must choose either an orchestration tool with the knowledge that VM scaling is not included or choose an orchestration tool that is integrated with your cloud provider so that it can, in fact, manage the scaling

High level approach: lightweight VM

- I can get a shell on it
  - through `ssh` or otherwise
- It "feels" like a VM
  - own process space
  - own network interface
  - can run stuff as root
  - can install packages
  - can run services
  - can mess up routing, iptables …

Low level approach: enhanced chroot

- It's not quite like a VM:
  - uses the host kernel
  - can't boot a different OS
  - can't have its own modules

- It's just normal processes on the host machine
  - contrast with VMs which are opaque

Building Blocks of Containers

- The namespace subsystem and the cgroup subsystem are the basis of lightweight process virtualization.

- They can be used also for setting a testing environment or as a resource management/resource isolation setup and for accounting.

The cgroup subsystem: background

- The cgroup (control groups) subsystem in Linux is a Resource Management and Resource Accounting-Tracking solution, providing a generic process-grouping framework
- It handles resources such as ***processor time, number of processes per group, amount of memory per control group or combination of such resources*** for a process or set of processes.
- cpu, memory and i/o are the most important resources that cgroup deals with

History of Cgroups

- Cgroups (abbreviated from control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.
- Engineers at Google started the work on this feature in 2006 under the name "process containers".
- In late 2007, the name changed to "control groups" to avoid confusion caused by multiple meanings of the term "container" in the Linux kernel context, and the control groups functionality was merged in kernel version 2.6.24.
- There are two versions of cgroups.
- Cgroups was originally written by Paul Menage and Rohit Seth, and mainlined into the Linux kernel in 2007. Afterwards this is called cgroups version 1.
- Development and maintenance of cgroups was then taken over by Tejun Heo who redesigned and rewrote it. This rewrite is now called version 2

- Unlike v1, cgroup v2 has only a single process hierarchy and discriminates between processes, not threads

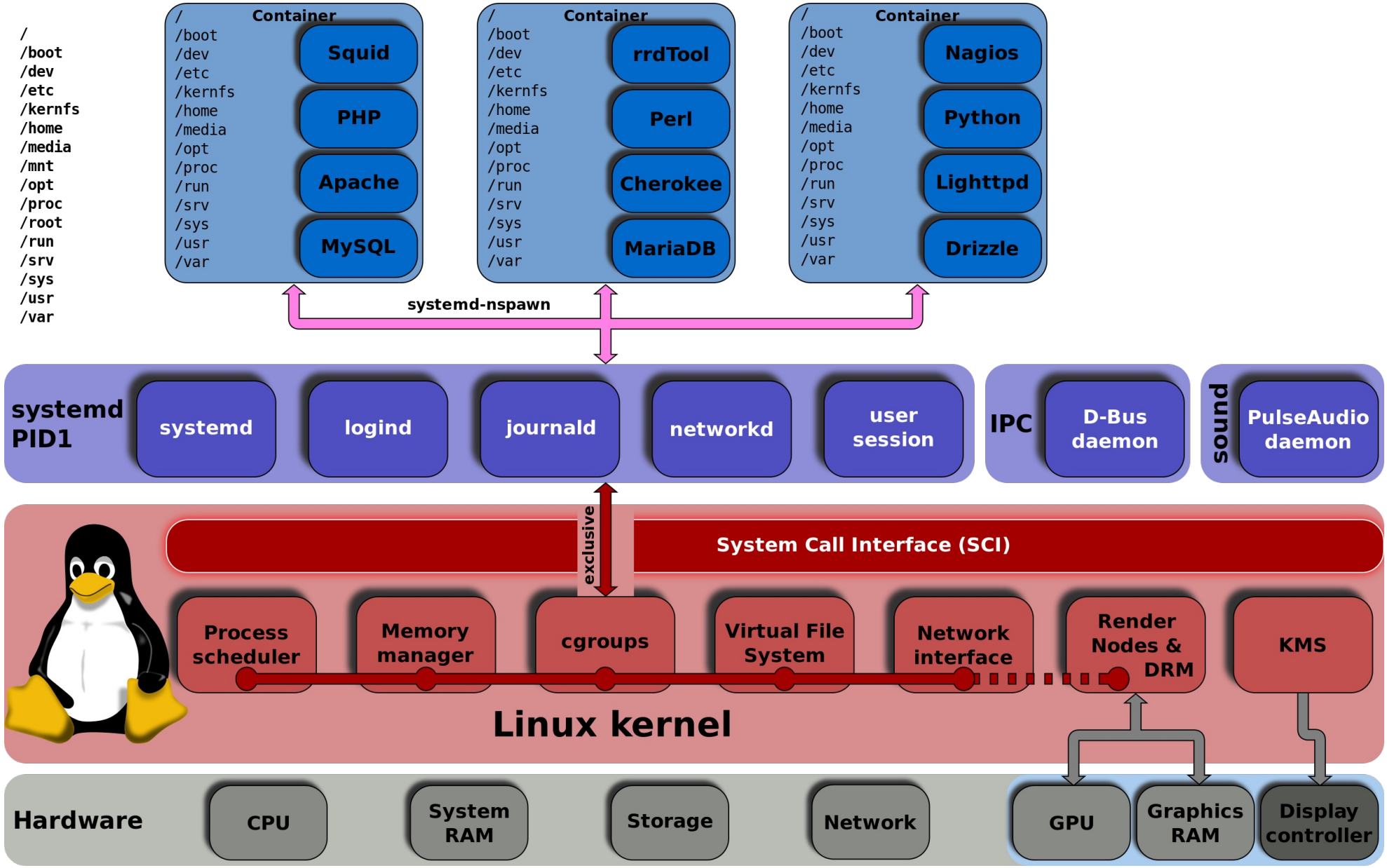
Use cases of cgroups

One of the design goals of cgroups is to provide a unified interface to many use cases, from controlling single processes (eg. by using *nice*) to full operating system-level virtualization (as provided by OpenVZ, Linux-VServer or LXC)

- Resource limiting: groups can be set to not exceed a configured memory limit, which also includes the file system cache
- Prioritization some groups may get a larger share of CPU utilization or disk I/O throughput
- Accounting measures a group's resource usage, which may be used, for example, for billing purposes
- Control freezing groups of processes, their checkpointing and restarting

Control groups

- Resource metering and limiting
  - memory
  - CPU
  - block I/O
  - network
- Prioritization
- Device node access control
- Crowd control



https://ru.wikipedia.org/wiki/Контрольная_группа_(Linux)

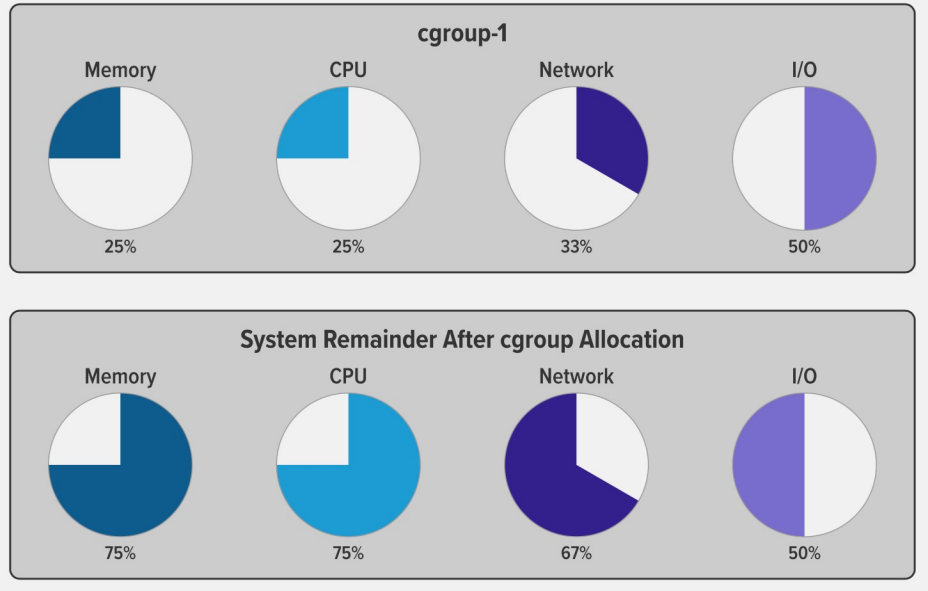
Listing cgroups

```
stylesen@harshu:~$ ls -alh /sys/fs/cgroup
total 0
drwxr-xr-x 12 root root 320 Mar 24 20:40 .
drwxr-xr-x  8 root root   0 Mar 24 20:40 ..
dr-xr-xr-x  6 root root   0 Mar 24 20:40 blkio
lrwxrwxrwx  1 root root  11 Mar 24 20:40 cpu -> cpu,cpuacct
lrwxrwxrwx  1 root root  11 Mar 24 20:40 cpuacct -> cpu,cpuacct
dr-xr-xr-x  6 root root   0 Mar 24 20:40 cpu,cpuacct
dr-xr-xr-x  3 root root   0 Mar 24 20:40 cpuset
dr-xr-xr-x  6 root root   0 Mar 24 20:40 devices
dr-xr-xr-x  4 root root   0 Mar 24 20:40 freezer
dr-xr-xr-x  7 root root   0 Mar 24 20:40 memory
lrwxrwxrwx  1 root root  16 Mar 24 20:40 net_cls -> net_cls,net_prio
dr-xr-xr-x  3 root root   0 Mar 24 20:40 net_cls,net_prio
lrwxrwxrwx  1 root root  16 Mar 24 20:40 net_prio -> net_cls,net_prio
dr-xr-xr-x  3 root root   0 Mar 24 20:40 perf_event
dr-xr-xr-x  6 root root   0 Mar 24 20:40 pids
dr-xr-xr-x  7 root root   0 Mar 24 20:40 systemd
```

Generalities

- Each subsystem has a hierarchy (tree)
  - separate hierarchies for CPU, memory, block I/O…
- Hierarchies are independent
  - the trees for e.g. memory and CPU can be different
- Each process is in a node in each hierarchy
  - think of each hierarchy as a different dimension or axis
- Each hierarchy starts with 1 node (the root)
  - Initially, all processes start at the root node*
- Each node = group of processes
  - sharing the same resources

Cgroups



- use cgroups to control how much of a given key resource (CPU, memory, network, and disk I/O) can be accessed or used by a process or set of processes.
- Cgroups are a key component of containers because there are often multiple processes running in a container that you need to control together
- The diagram shows how when you allocate a particular percentage of available system resources to a cgroup (in this case **cgroup-1**), the remaining percentage is available to other cgroups (and individual processes) on the system

https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/

Memory cgroup: accounting

- The kmemcg controller can limit the amount of memory that the kernel can utilize to manage its own internal processes.
- Keeps track of pages used by each group:
  - file (read/write/mmap from block devices)
  - active (recently accessed)
  - inactive (candidate for eviction)
- Each page is "charged" to a group
- Pages can be shared across multiple groups
  - e.g. multiple processes reading from the same files

Memory cgroup: limits

- Each group can have its own limits
  - limits are optional
  - two kinds of limits: soft and hard limits
- Soft limits are not enforced
  - they influence reclaim under memory pressure
- Hard limits will trigger a per-group **out of memory (OOM)** killer
- Limits can be set for different kinds of memory
  - physical memory
  - kernel memory

Memory cgroup: avoiding OOM killer

Do you like when the kernel terminates random processes because something unrelated ate all the memory in the system?

- Setup oom-notifier (out of memory daemon)

- Then, when the hard limit is exceeded:
  - freeze all processes in the group
  - notify user space (instead of going rampage)
  - we can kill processes, raise limits, migrate containers ...
  - when we're in the clear again, unfreeze the group

Memory cgroup: tricky details

- Each time the kernel gives a page to a process, or takes it away, it updates the counters
- This adds some overhead
- This cannot be enabled/disabled per process
  - it has to be done at boot time
- When multiple groups use the same page, only the first one is "charged"
  - but if it stops using it, the charge is moved to another group

Cpu cgroup

- Keeps track of user/system CPU time.
- Keeps track of usage per CPU
- Allows to set weights

Cpuset cgroup

- Pin groups to specific CPU(s)
- Reserve CPUs for specific apps
- Avoid processes bouncing between CPUs
- Provides extra dials and knobs per zone memory pressure, process migration costs…

Blkio cgroup

- Keeps track of I/Os for each group
  - per block device
  - read vs write
  - sync vs async
- Set throttle (limits) for each group
  - per block device
  - read vs write
- Set relative weights for each group

Net_cls and net_prio cgroup

- Automatically set traffic class or priority, for traffic generated by processes in the group

- Net_cls will assign traffic to a class
  - class then has to be matched with iptables,
  - otherwise traffic just flows normally

    - Net_prio will assign traffic to a priority
      - priorities are used by queuing disciplines

Devices cgroup

- A device cgroup associates a device access ***white list*** with each cgroup.
- Controls what the group can do on the device nodes
- Permissions include read/write/mknod

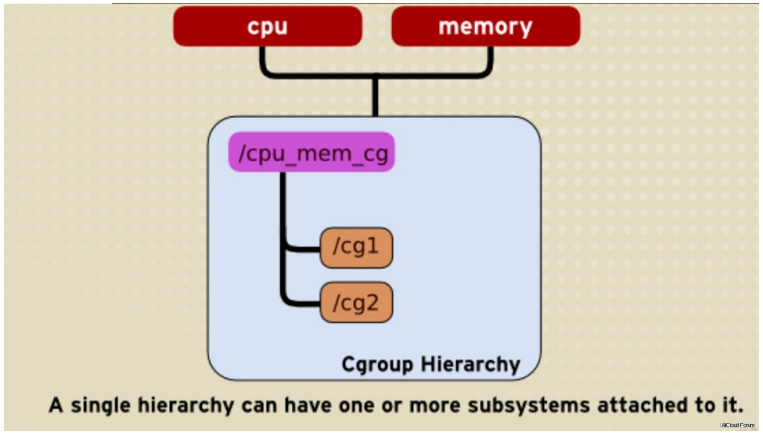
Freezer cgroup (crowd control)

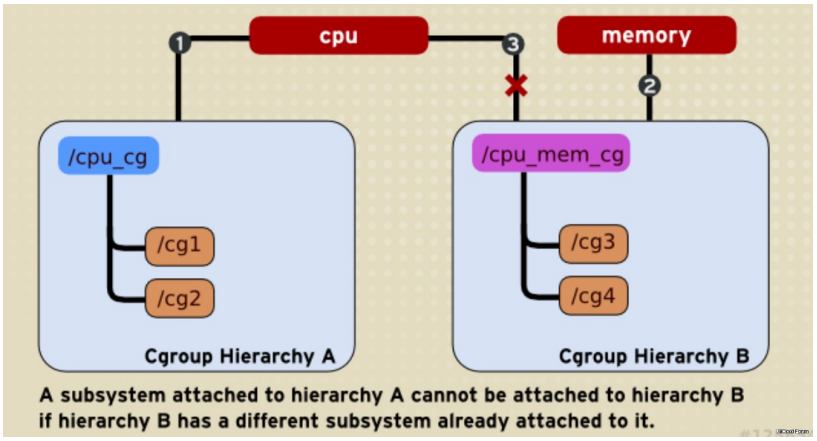
- Allows to freeze a group of processes
- Cannot be detected by the processes
- Specific usecases
  - cluster batch scheduling
  - process migration

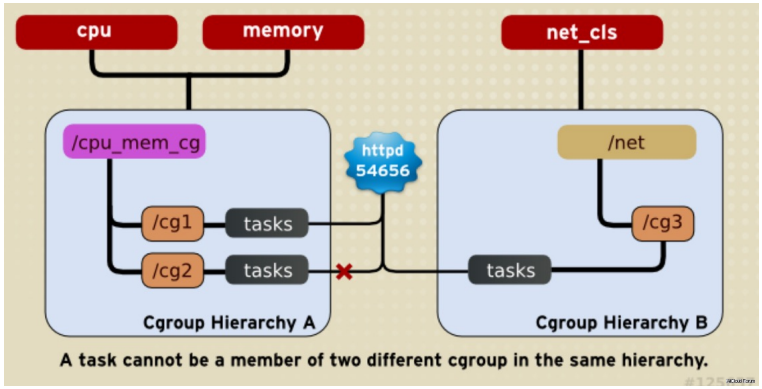
How to use Cgroups?

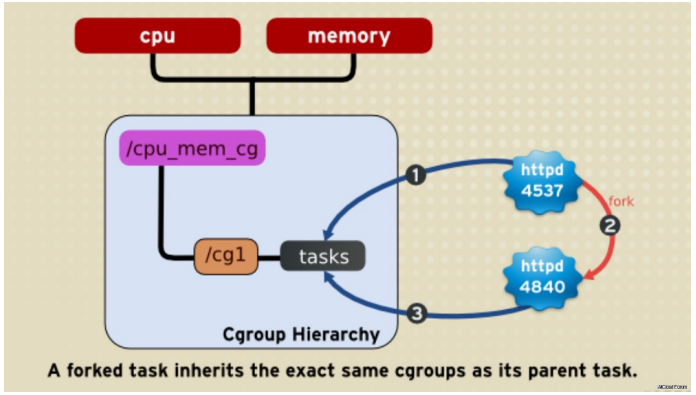
- Control groups can be used in multiple ways:
  - Directly by accessing the cgroup virtual file system manually.
  - Indirectly through other software that uses cgroups, such as Docker, Linux Containers (LXC) virtualization, libvirt, systemd, Open Grid Scheduler/Grid Engine, and Google's developmentally defunct lmctfy.
  - Four main rules

Rules for managing cgroups









https://www.alibabacloud.com/forum/read-397

Namespaces

Namespaces provide processes with their own system view

Cgroups = limit how much you can use;

namespaces = limit what you can see (you cannot use/affect what you cannot see)

- cgroups are a kernel mechanism for limiting and measuring the total resources used by a group of processes running on a system. For example, you can apply CPU, memory, network or IO quotas.
- Namespaces are a kernel mechanism for limiting the visibility that a group of processes has of the rest of a system. For example you can limit visibility to certain process trees, network interfaces, user IDs or filesystem mounts.

Linux Containers: Namespaces

- A namespace wraps a **global system resource** in an **abstraction**
  - it appear to the processes within the namespace that they have their own
  - isolated instance of the global resource.
- Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes.
- Currently, Linux provides 6 types of namespaces

Namespaces - contd

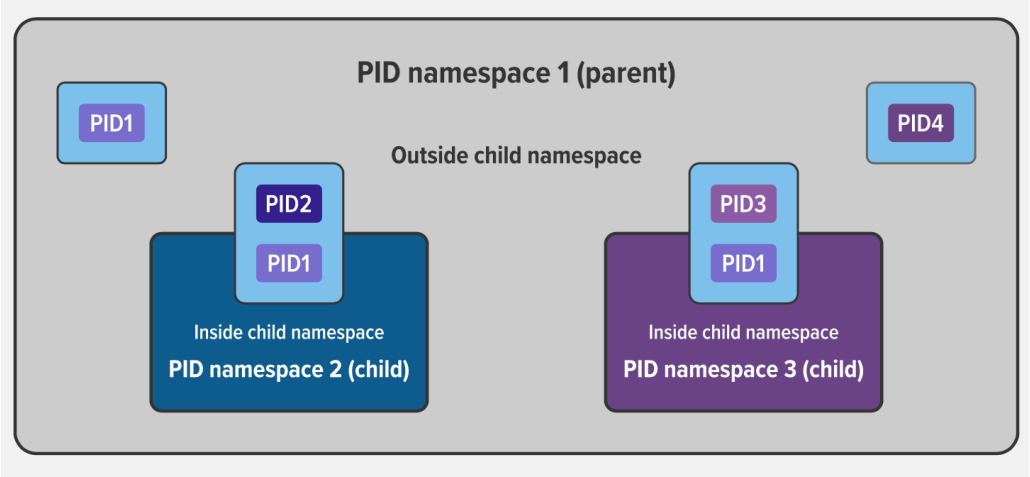
There are currently six namespaces:

- mnt (mount points, filesystems)
- pid (processes)
- net (network stack)
- ipc (System V IPC)
- uts (hostname)
- user (UIDs)
- … (plans to add more)

PIDnamespace

- Processes within a PID namespace only see processes in the same PID namespace
- Each PID namespace has its own numbering
  - starting at 1
- If PID 1 goes away, whole namespace is killed

PID - example



- In the diagram , there are three PID namespaces – a parent namespace and two child namespaces.
- Within the parent namespace, there are four processes, named PID1 through PID4. These are normal processes which can all see each other and share resources.
- The child processes with PID2 and PID3 in the parent namespace also belong to their own PID namespaces in which their PID is 1.
- From within a child namespace, the PID1 process cannot see anything outside. For example, PID1 in both child namespaces cannot see PID4 in the parent namespace.
- This provides isolation between (in this case) processes within different namespaces

https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/

IPC namespace

- IPC namespace isolates the System V inter-process communication between namespaces.

- Allows a process (or group of processes) to have its private:
  - IPC semaphores
  - IPC message queues
  - IPC shared memory

- Without risk of conflict with other instances

Copy-on-write storage

- Create a new container instantly instead of copying its whole filesystem
- Storage keeps track of what has changed
- Many options available
  - AUFS, overlay (file level)
  - device mapper thinp (block level)
  - BTRFS, ZFS (FS level)
- Considerably reduces footprint and “boot” times

What is a container

- It is like a lightweight VM
- I can get a shell on it (via ssh)
- It feels like a VM: own process space, network interface, can run stuff as root, can install packages, can run services, can mess up routing, IPtables, …

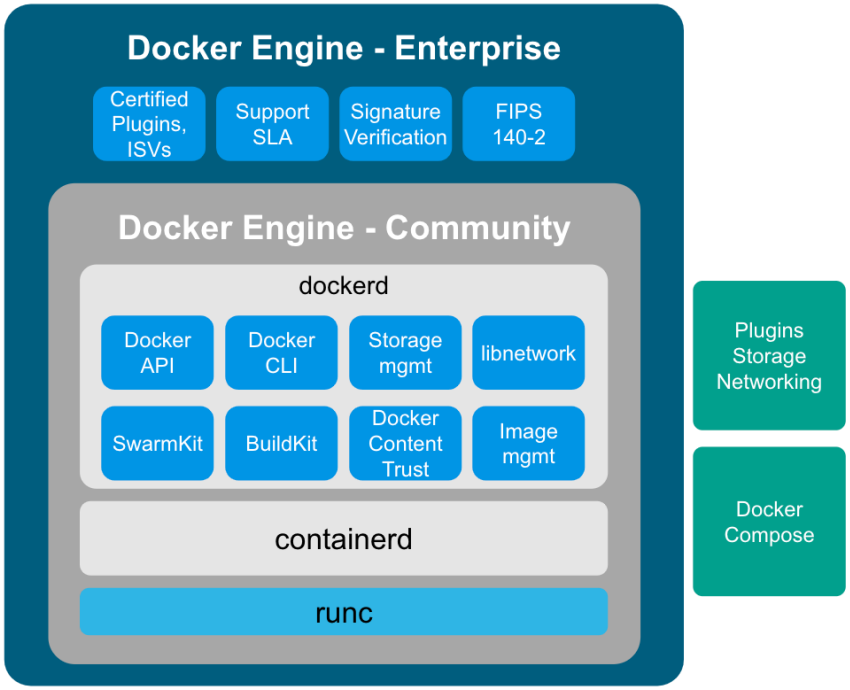
- It is different from a VM
- Uses the host kernel
- Cannot boot a different OS
- Cannot have its own modules
- Does not need init as PID 1
- Does not need syslog, cron,…
- It is a set of processes visible on the host machine (true VM are opaque)

LXC linuxcontainers.org

- LXC is a system container, that is it offers an environment as close as possible to the one you'd get from a VM, but without the overhead that comes with running a separate kernel and simulating all the hw
- Set of UserLAnd tools (Use Linux Anywhere)
- A container is a directory in /var/lib/lxc
- Small config file + root filesystem
- Early versions had no support for CopyOnWrite
- Early versions had no support to move images
- Requires significant amount of programming effort
- easy for sysadmins/ops, hard for devs

Docker Engine

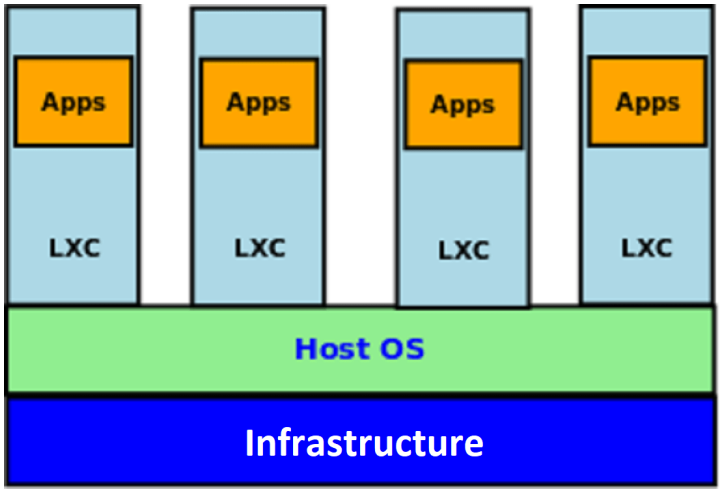
- Daemon controlled by REST(ish) API
- First versions shelled out to LXC
  - now uses its own libcontainer runtime
- Manages containers, images, builds, and much more

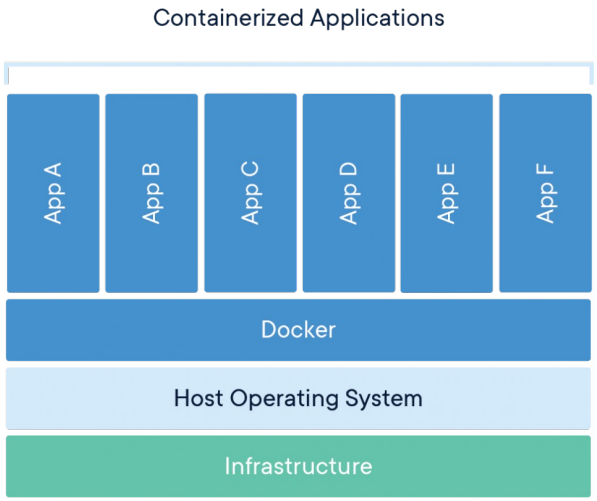


https://www.docker.com/blog/tag/docker-engine/

Which one is best?

- They all use the same kernel features
- Performance will be almost the same
- Look at:
  - features
  - design
  - ecosystem





References

- https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-i-linux-control-groups-and-process
- https://en.wikipedia.org/wiki/Operating-system-level_virtualization#IMPLEMENTATIONS
- https://en.wikipedia.org/wiki/Cgroups
- https://en.wikipedia.org/wiki/Linux_namespaces
- https://sites.google.com/site/mytechnicalcollection/cloud-computing/docker/container-vs-process
- Ivanov, *Containerization with LXC*, Packt 2017
- Kumaran, *Practical LXC and LXD*, Apress, 2017