CONTAINER HOSTS

CONTAINER HOST BASICS

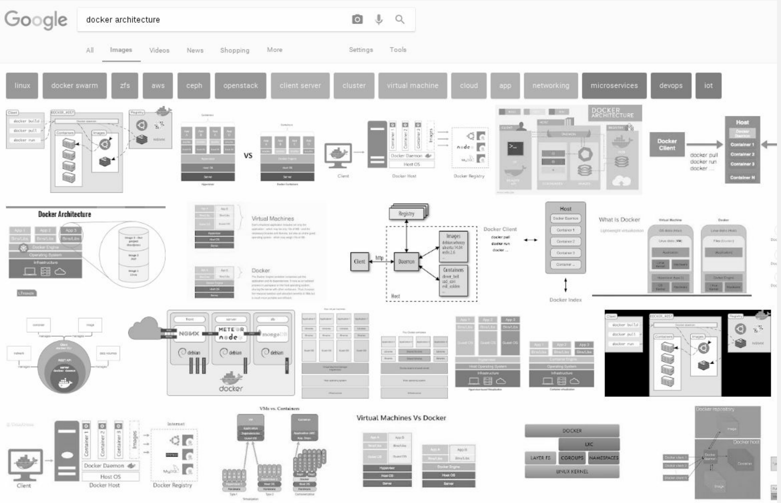
Container Engine, Runtime, and Kernel

CONTAINERS DON’T RUN ON DOCKER

The Internet is WRONG :-)

Important corrections

- Containers do not run ON docker. Containers are processes - they run on the Linux kernel. Containers are Linux processes (or Windows).
- The docker daemon is one of the many user space tools/libraries that talks to the kernel to set up containers

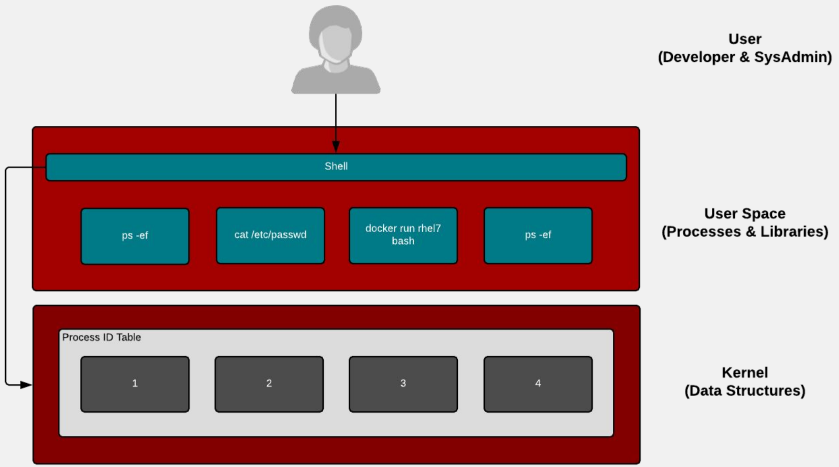


PROCESSES VS. CONTAINERS

Actually, there is no processes vs. containers in the kernel

User space and kernel work together

- There is only one process ID structure in the kernel
- There are multiple human and technical definitions for containers
- Container engines are one technical implementation which provides both a methodology and a definition for containers

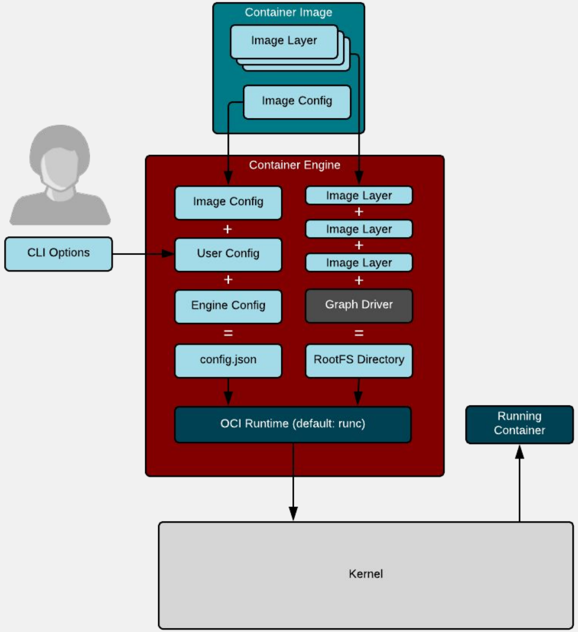


THE CONTAINER ENGINE IS BORN

This was a new concept introduced with Docker Engine and CLI

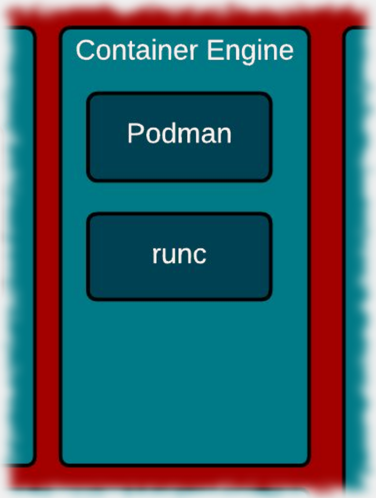
Think of the Docker Engine as a giant proof of concept - and it worked!

- Container images
- Registry Servers
- Ecosystem of pre-built images
- Container engine
- Container runtime (often confused)
- Container image builds
- API
- CLI
- A LOT of moving pieces

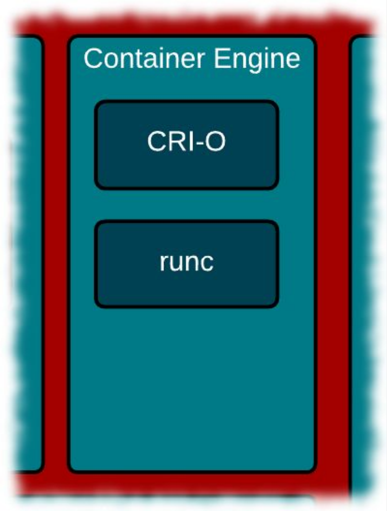


DIFFERENT ENGINES

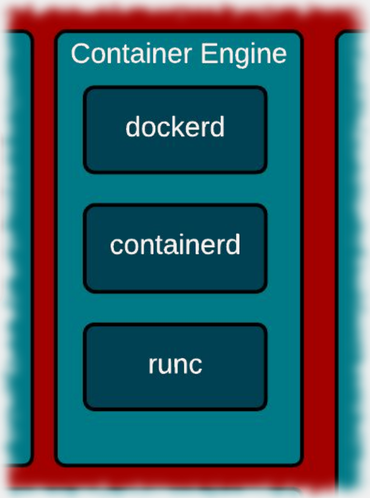
All of these container engines are OCI compliant



Podman



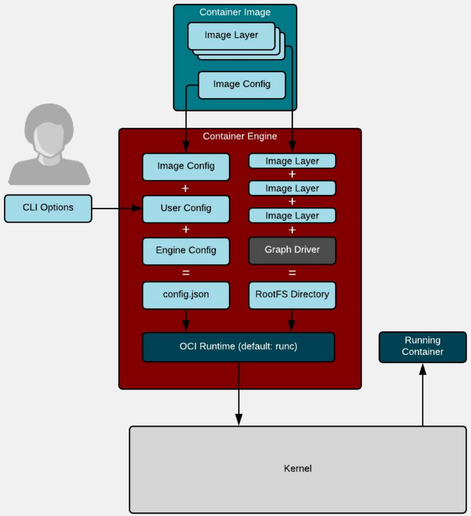
CRI-O



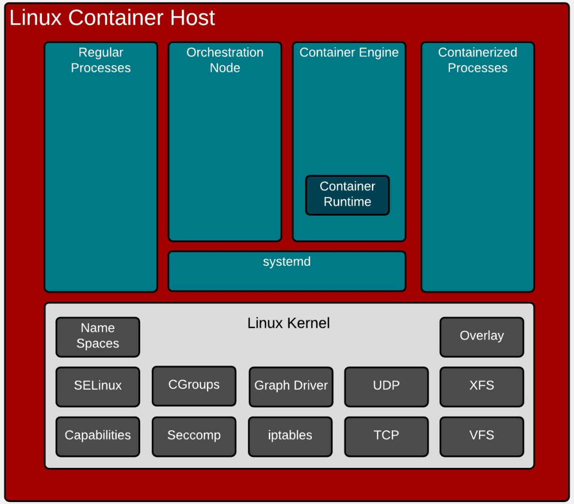
Docker

CONTAINER ENGINE VS. CONTAINER HOST

In reality the whole container host is the engine - like a Swiss watch



VS.

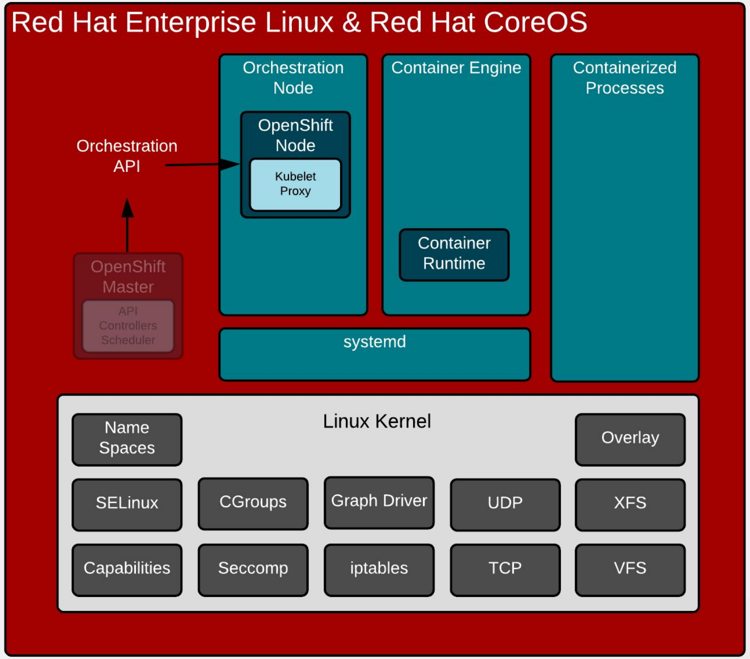


CONTAINER HOST

Released, patched, tested together

Tightly coupled communication through the kernel - all or nothing feature support:

- Operating System (kernel)
- Container Runtime (runc)
- Container Engine (Docker)
- Orchestration Node (Kubelet)
- Whole stack is responsible for running containers



Red Hat Enterprise Linux & Red Hat CoreOS

Orchestration API

OpenShift Master

API
Controllers
Scheduler

Orchestration Node

OpenShift Node

Kubelet
Proxy

Container Engine

Container Runtime

Containerized Processes

systemd

Linux Kernel

Name Spaces

Overlay

SELinux

CGroups

Graph Driver

UDP

XFS

Capabilities

Seccomp

iptables

TCP

VFS

CONTAINER ENGINE

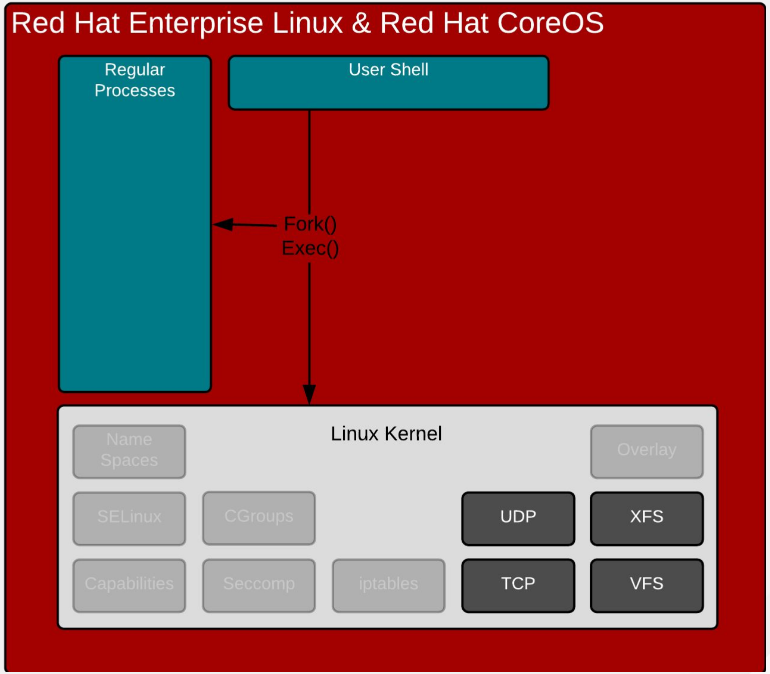
Defining a container

KERNEL

Creating regular Linux processes

Normal processes are created, destroyed, and managed with system calls:

- Fork() - Think Apache
- Exec() - Think ps
- Exit()
- Kill()
- Open()
- Close()
- System()

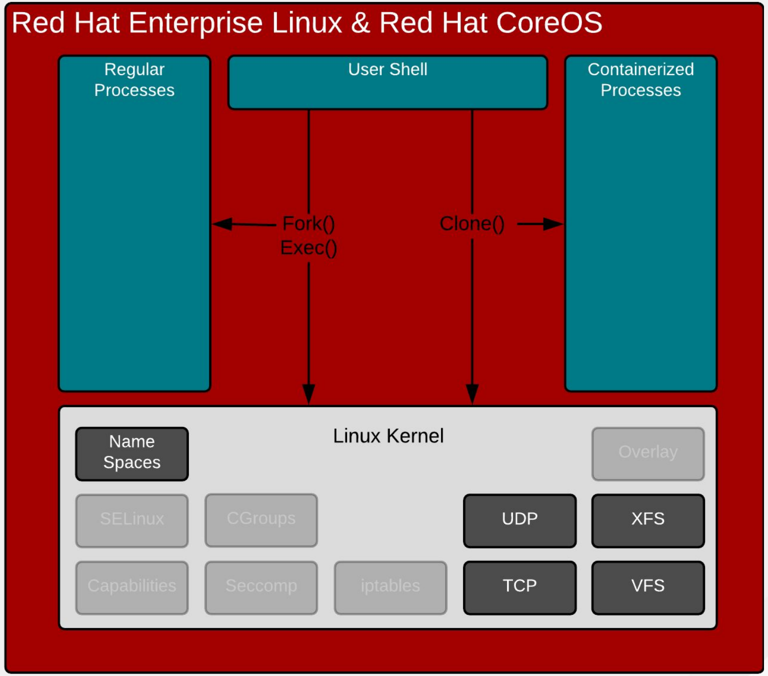


KERNEL

Creating “containerized” Linux processes

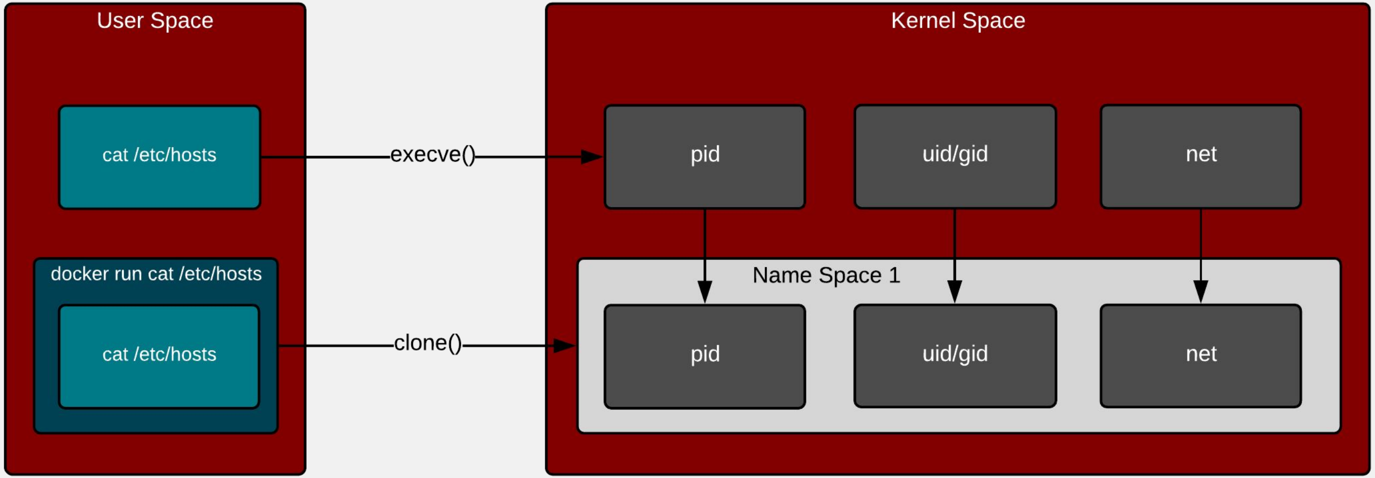
What is a container anyway?

- No kernel definition for what a container is - only processes
- Clone() - closest we have
- Creates namespaces for kernel resources
  - Mount, UTC, IPC, PID, Network, User
- Essentially, virtualized data structures



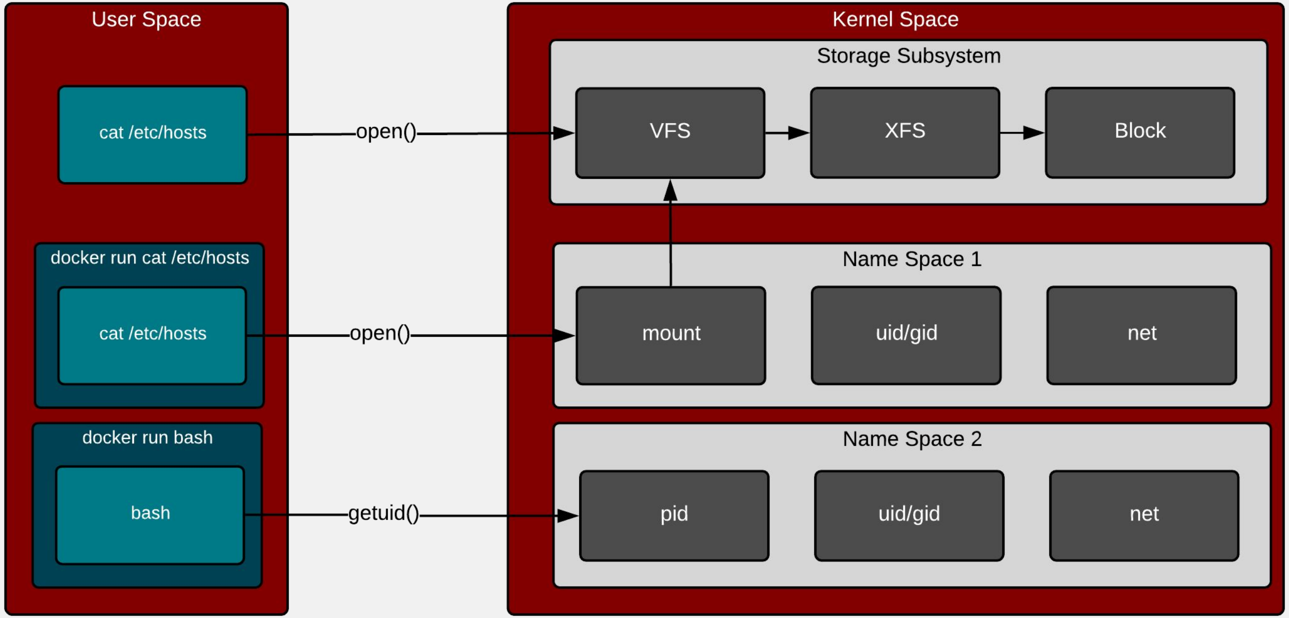
KERNEL

Namespaces are all you get with the clone() syscall



KERNEL

Even namespaced resources use the same subsystem code

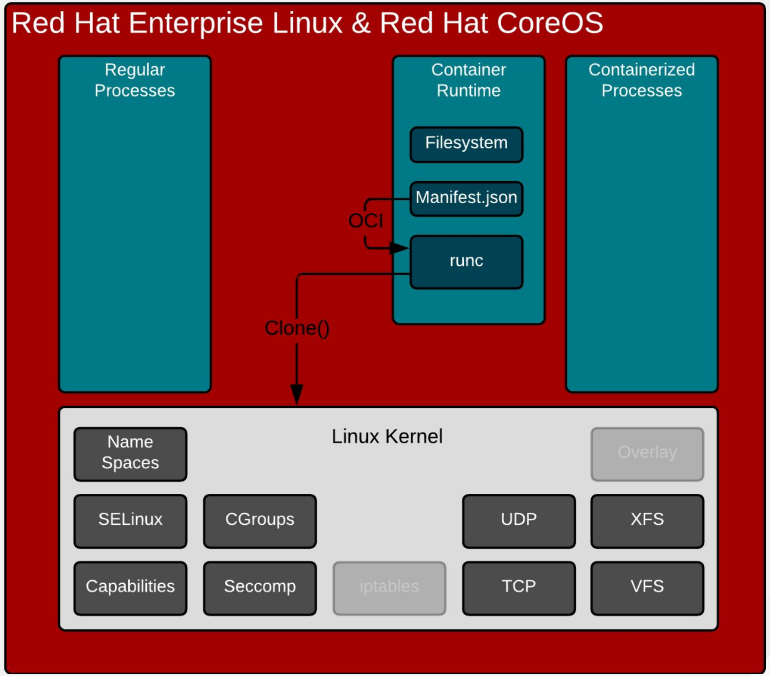


CONTAINER RUNTIME

Standarding the way user space communicates with the kernel

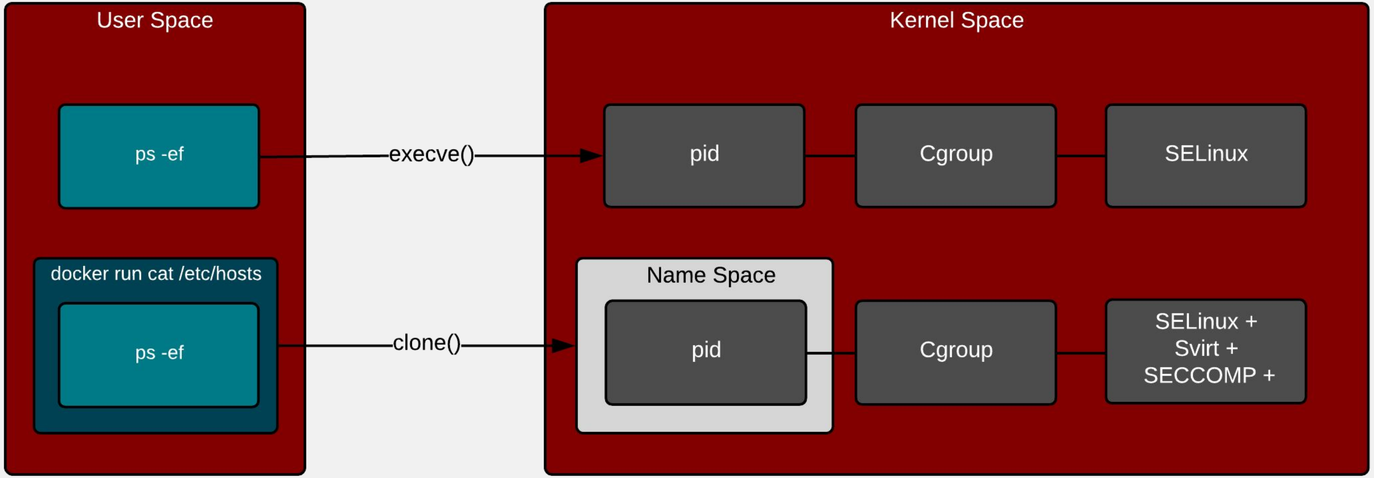
Expects some things from the user:

- OCI Manifest - json file which contains a familiar set of directives - read only, seccomp rules, privileged, volumes, etc
- Filesystem - just a plain old directory which has the extracted contents of a container image



CONTAINER RUNTIME

Adds in cgroups, SELinux, sVirt, and SECCOMP

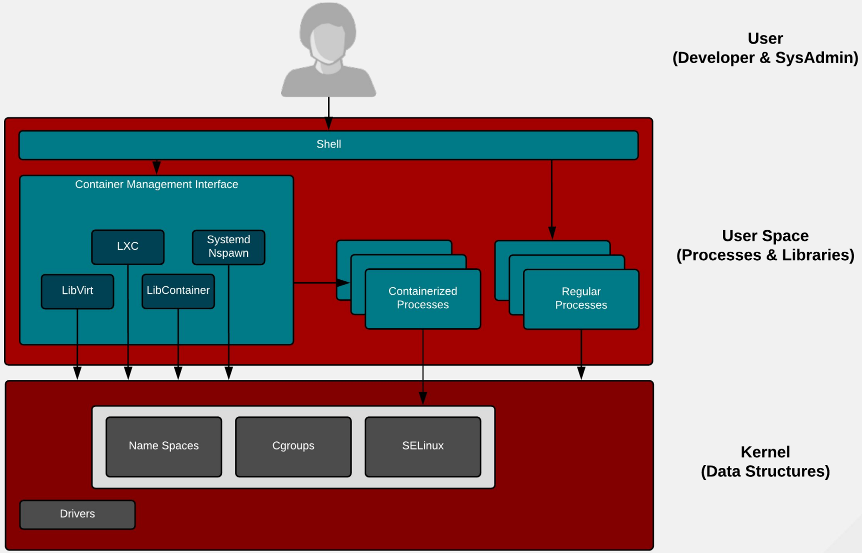


CONTAINER RUNTIME

But, there were others before runc, what's the deal?

There is a rich history of standardization attempts in Linux:

- LibVirt
- LXC
- Systemd Nspawn
- LibContainer (eventually became runc)

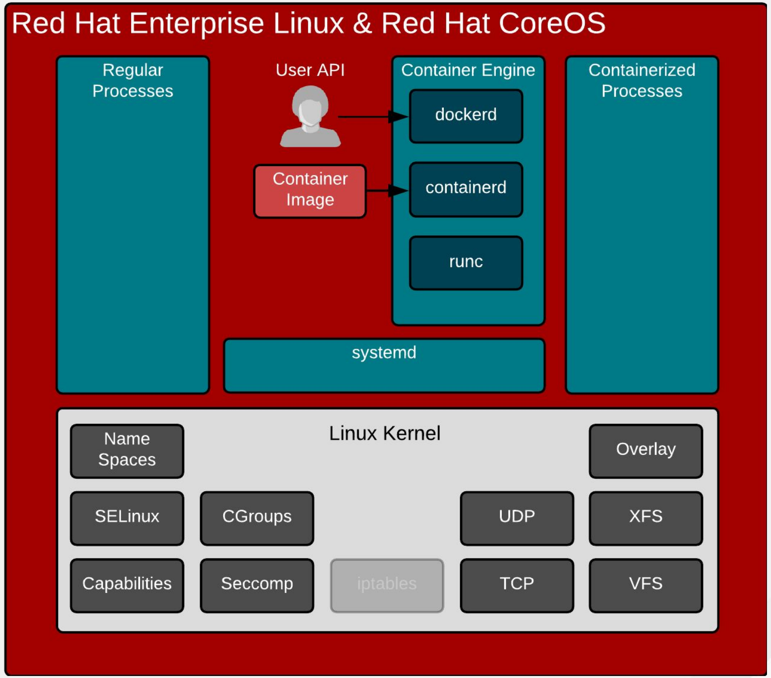


CONTAINER ENGINE

Provides an API prepares data & metadata for runc

Three major jobs:

- Provide an API for users and robots
- Pulls image, decomposes, and prepares storage
- Prepares configuration - passes to runc

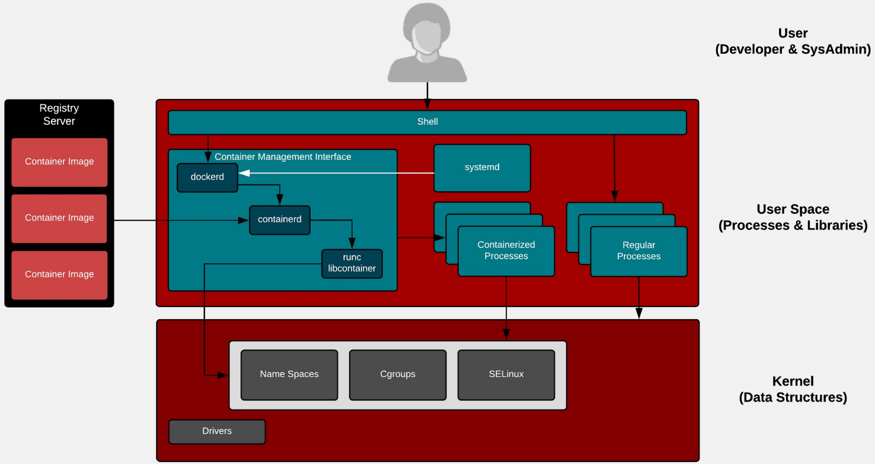


PROVIDE AN API

Regular processes, daemons, and containers all run side by side

In action:

- Number of daemons & programs working together
  - dockerd
  - containerd
  - runc

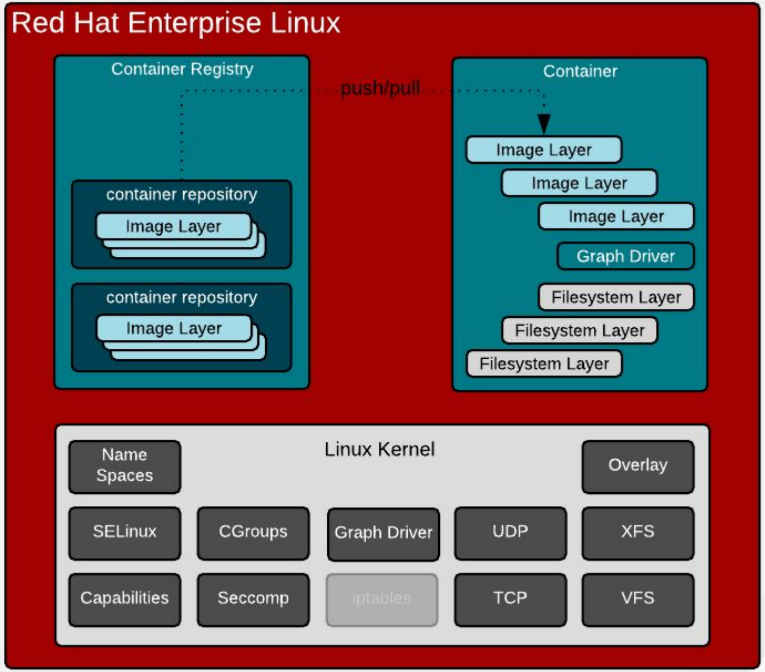


PULL IMAGES

Mapping image layers

Pulling, caching and running containers:

- Most container engines use graph drivers which rely on kernel drivers (overlay, device mapper, etc)
- There is work going on to do this in user space, but there are typically performance trade offs

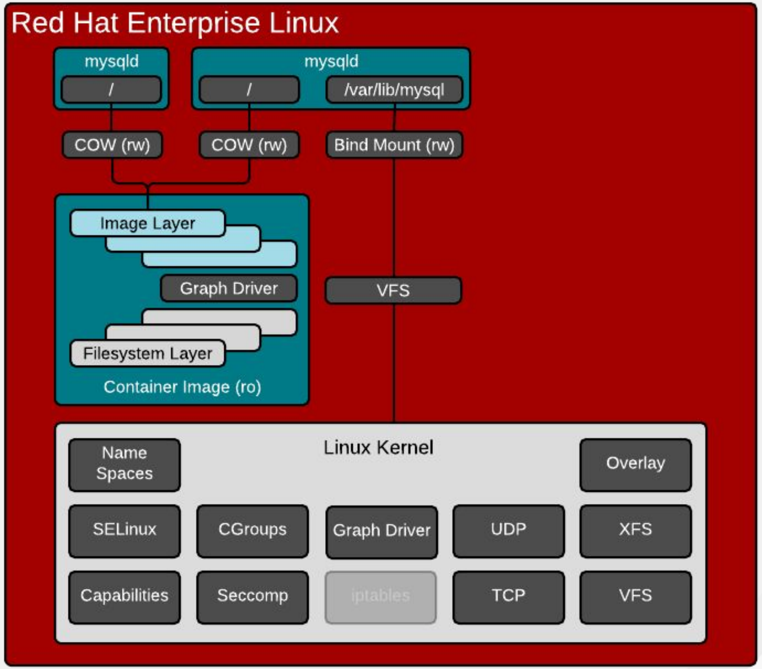


PREPARE STORAGE

Copy on write and bind mounts

Understanding implications of bind mounts:

- Copy on write layers can be slow when writing lots of small files
- Bind mounted data can reside on any VFS mount (NFS, XFS, etc)

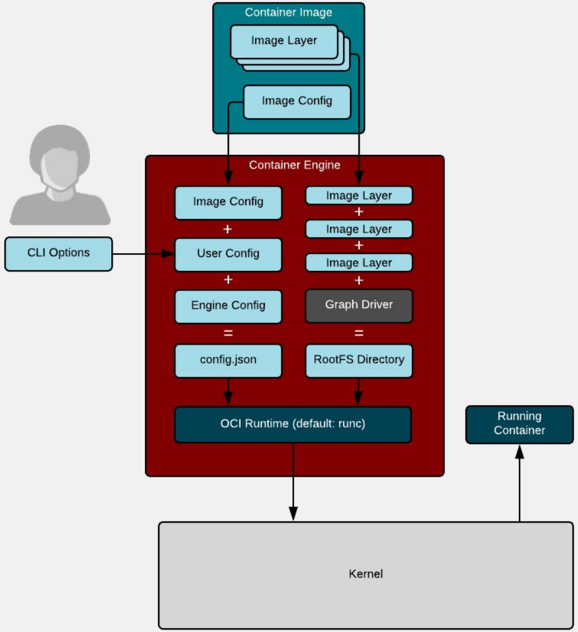


PREPARE CONFIGURATION

Combination of image, user, and engine defaults

Three major inputs:

- User inputs can override defaults in image and engine
- Image inputs can override engine defaults
- Engine provides sane defaults so that things work out of the box

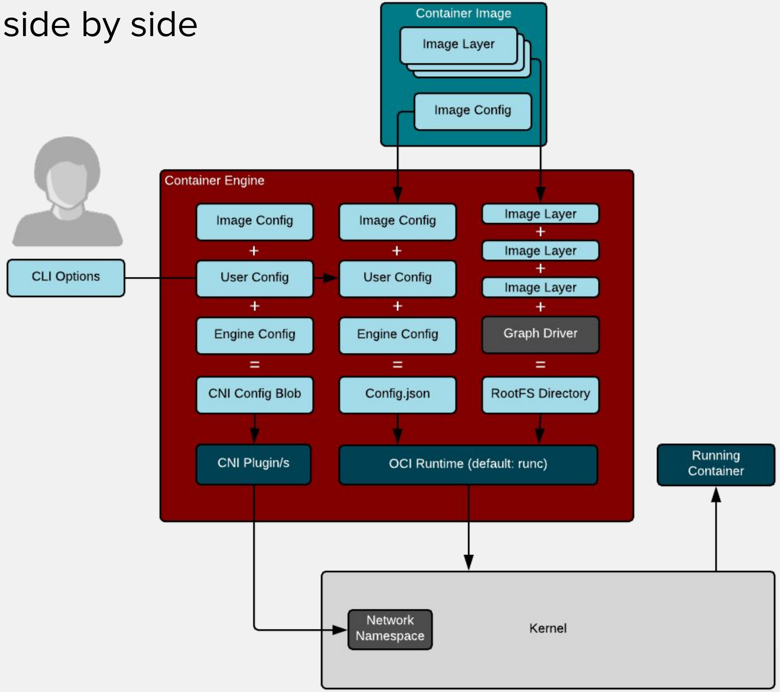


PREPARE CONFIGURATION + CNI

Regular processes, daemons, and containers all run side by side

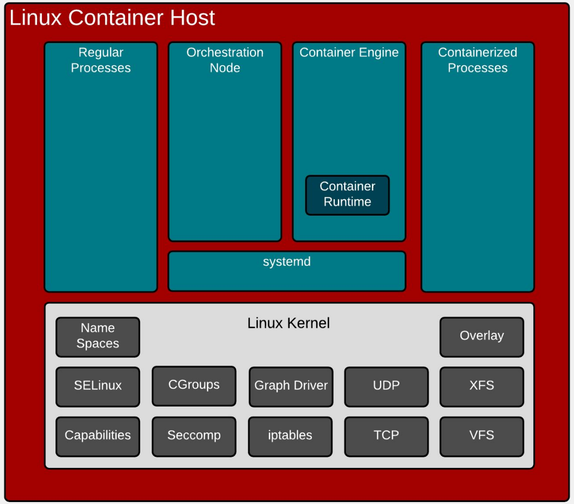
In action:

- Takes user specified options
- Pulls image, expands, and parses metadata
- Creates and prepares CNI json blob
- Hands CNI blob and environment variables to one or more plugins (bridge, portmapper, etc)



ENGINE, RUNTIME, KERNEL, AND MORE

All of these must revision together and prevent regressions together



Linux Container Host

Regular Processes

Orchestration Node

Container Engine

Container Runtime

Containerized Processes

systemd

Linux Kernel

Name Spaces

Overlay

SELinux

CGroups

Graph Driver

UDP

XFS

Capabilities

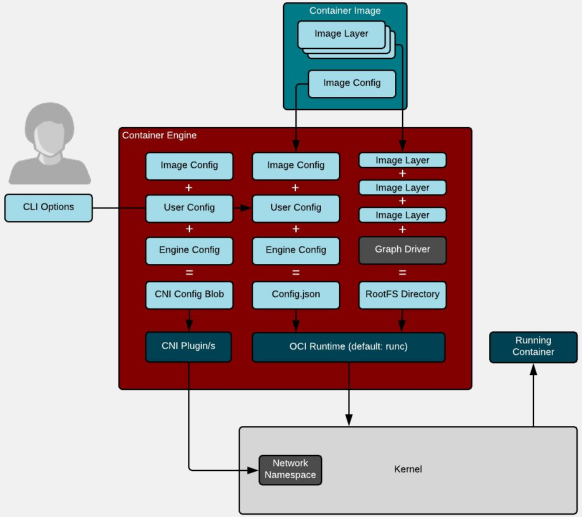
Seccomp

iptables

TCP

VFS

+



Container Image

Image Layer

Image Config

Container Engine

CLI Options

Image Config + User Config + Engine Config = CNI Config Blob

Image Config + User Config + Engine Config = Config.json

Image Layer + Image Layer + Image Layer + Graph Driver = RootFS Directory

CNI Plugin/s

OCI Runtime (default: runc)

Running Container

Network Namespace

Kernel

BONUS INFORMATION

Other related technology

Containers With Advanced Isolation

Kata Containers, gVisor, and *KubeVirt (because deep down inside you want to know)*

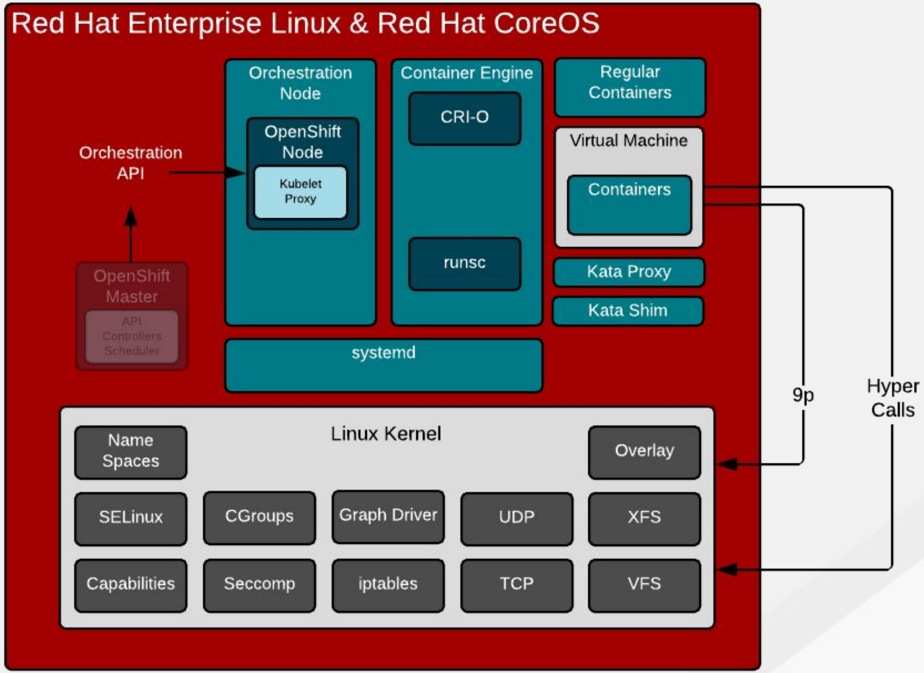
- **Kata Containers** integrate at the container runtime layer
- **gVisor** integrates at the container runtime layer
- **KubeVirt** not advanced container isolation. Add-on to Kubernetes which extends it to schedule VM workloads side by side with container workloads

Kata Containers

Containers in VMs

You still need connections to the outside world:

- Shim offers reaping of processes/VMs similar to normal containers
- Proxy allows serial access into container in VM
- P9fs is the communication channel for storage



gVisor

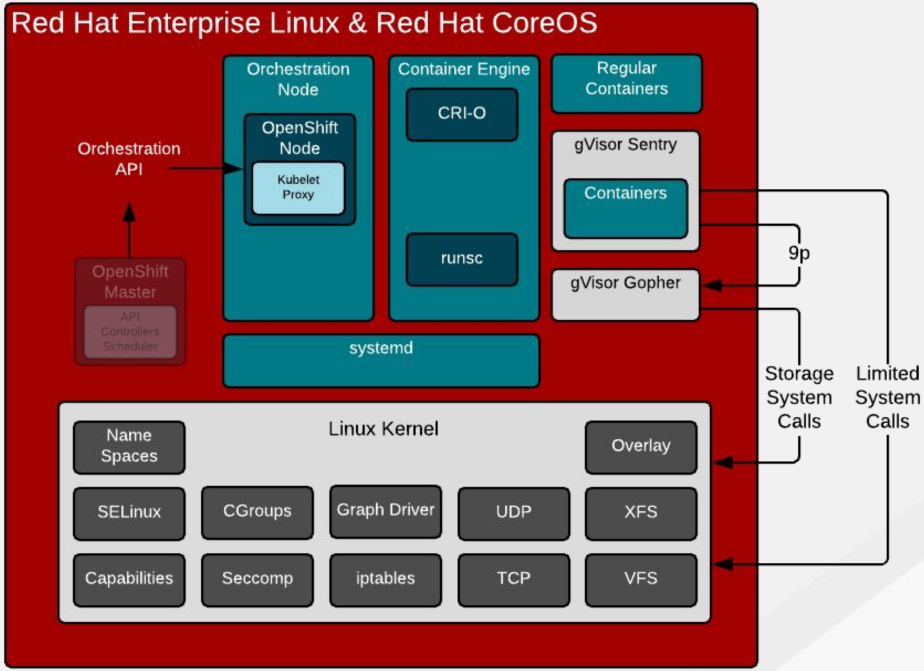
Anybody remember user mode Linux?

gVisor is:

- Written in golang
- Runs in userspace
- Reimplements syscalls
- Reimplements hardware
- Uses 9p for storage

Concerns

- Storage performance
- Limited syscall implementation



KubeVirt

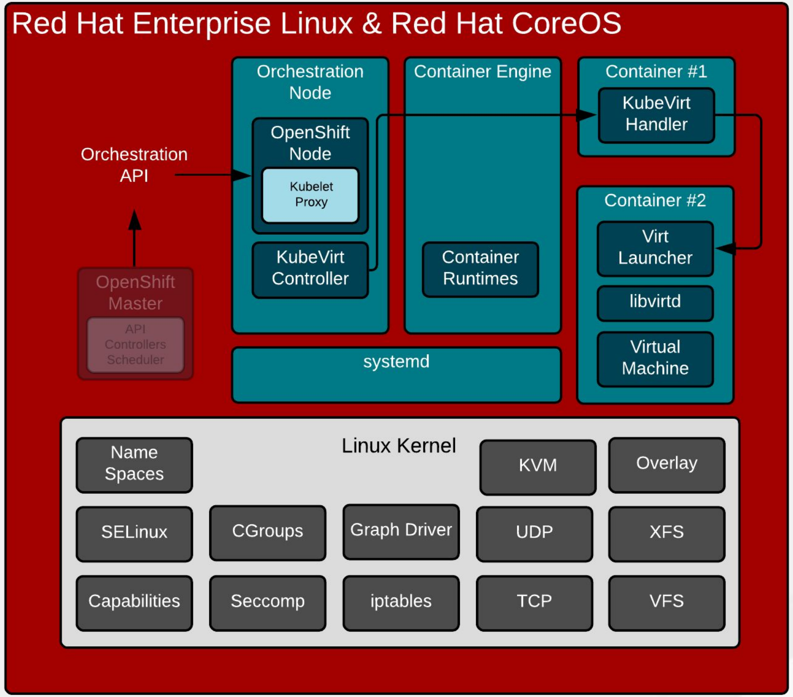
Extension of Kubernetes for running VMs

KubeVirt is:

- Custom resource in Kubernetes
- Defined/actual state VMs
- Good for VM migrations
- Uses persistent volumes for VM disk

KubeVirt is not:

- Stronger isolation for containers
- Part of the Container Engine
- A replacement Container Runtime
- Based on container images



Red Hat Enterprise Linux & Red Hat CoreOS

Orchestration API

OpenShift Master

API
Controllers
Scheduler

Orchestration Node

OpenShift Node

Kubelet
Proxy

KubeVirt Controller

Container Engine

Container Runtimes

Container #1

KubeVirt Handler

Container #2

Virt Launcher

libvirtd

Virtual Machine

systemd

Linux Kernel

Name Spaces

KVM

Overlay

SELinux

CGroups

Graph Driver

UDP

XFS

Capabilities

Seccomp

iptables

TCP

VFS