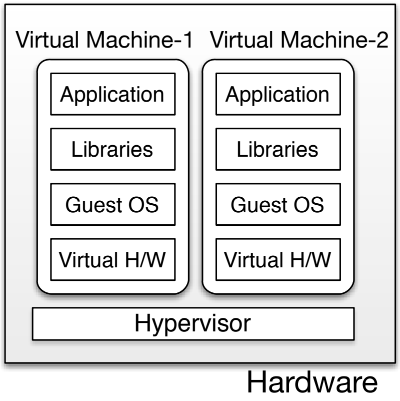
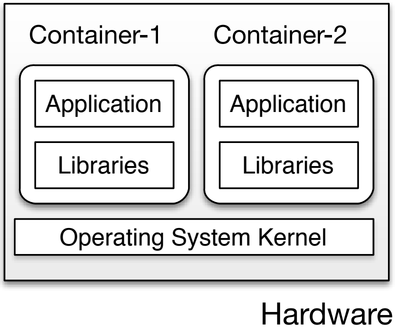
OS Virtualization and Containers

Hardware vs. OS Virtualization





OS Virtualization

- Virtualize the OS for each application
- *Not* the hardware
- Allow applications to run in isolation from each other
- If this seems similar to the process abstraction, that's because it is
- Resource sharing is a core OS primitive anyway.
- However, some services are shared:

1. File system, libraries, config files
2. Users, uids
3. Pid tree (pid's are unique)
4. Network interfaces, rules (iptables), ports
5. Time
6. Procfs, sysfs

Dual View of Containers

Containers can be viewed as either:

1. Lightweight virtual machines
2. Process groups with better isolation

Containers as OS virtualization analogue of VMs:

- Applications/processes run inside containers
  - Isolated from each other using sandboxing techniques
  - Applications should not affect or monitor apps in other containers
- Containers run on a “virtualized OS interface”
- Need no additional interposition
- No/negligible performance degradation

Essence of OS Virtualization

- Virtualize the shared OS services
- Ideally: applications not affected by others in any way
- Implemented through namespaces
- Resource isolation. Fine-grained control of CPU, memory, I/O resources consumed
- OS must try to provide isolation anyway
- OS virt is thus the natural progression:
  - Early UNIX: File system isolation with chroot
  - FreeBSD: Jails [2000] (chroot+containing the omnipotent root)
  - Solaris: Zones [2005] (refined, more isolated Jails)
  - Linux: [Virtuzzo-2000, LXC-2008, . . .]

OS Virtualization Desiderata

- Security
- Isolation
- Virtualization
  - HW devices, network IP, hostname, . . .
- Granularity
  - Containers can be arbitrarily sized
  - No dedicated CPUs required
- Transparency. No porting required. Exact same ABI/API as running on bare-metal.

OS Virtualization Challenges

- Lots of OS $\rightarrow$ App interfaces to isolate
- Security challenges abound
- Needs careful understanding of the OS
- Replace process-id with (process-id, container-id) throughout the kernel
- Update virtualization and isolation with each new OS feature
- In contrast, HW, and HW virt, are relatively stable
- What should be the limitations on the “root” user in a container?

Namespaces

- Split global kernel resource structures into separate instances.
- Pid (Each namespace has its own pid tree)
- Network : nic, iptables, routing tables
- UTS: hostname
- Mount : Private mount-points with different file-system trees
- User: User-ids
- IPC: POSIX shmem, etc

Namespace usage and implementation

- Containers are created with these new namespaces.
- unshare: run program with some namespaces unshared from parent
- unshare –map-root-user –user sh -c whoami # outputs root
- setns
- nsenter

Control groups

- CPU, memory, blkio
- cgcreate and cgexec for creating and running programs
- Control via sysfs (/sys/fs/cgroup/)
- Resource limiting, prioritization, accounting
- CPU : cpu.shares (1024 max)
- cpuset: Set which CPUs (cores) a cgroup can use
- Memory: max_usage_in_bytes for setting max allocation
- Control (freeze, checkpoint processes)

Linux Containers (LXC)

- Create “containers” using namespaces, cgroups, seccomp (security policies)
- Similar control abstraction as VMs:
- Containers have names, “FS images”, resource allocation (CPU, mem,..)
- Operations: create, start, shutdown, pause, migrate

Docker

- LXC + layered file system + image repository
- Copy-on-write file system allows images to be composed layer-wise:
  1. Base layer: Debian
  2. Layer 2: Essentials (Emacs)
  3. Layer 3: Apache web server
- Common use-case: CICD
- Continuous Integration and Deployment
- Create docker container in dev environment, “push” into production

Docker Commands

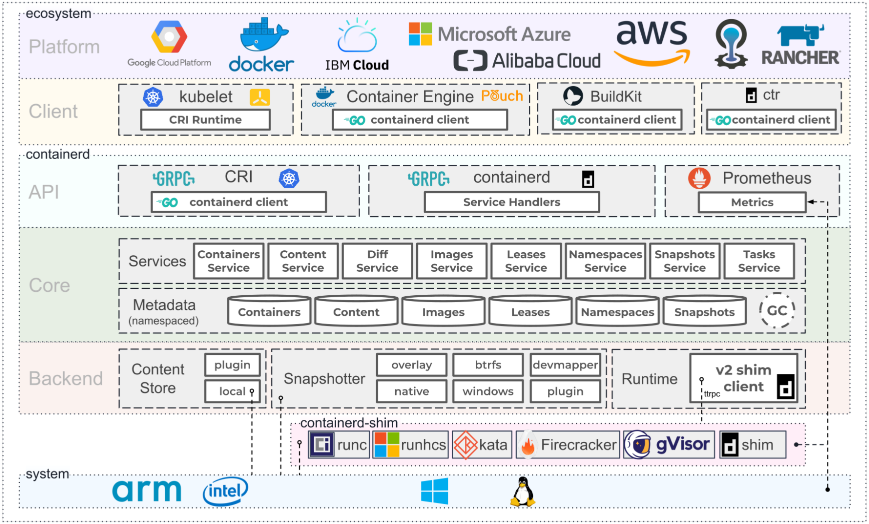
- docker inspect image-name to see image layer storage
- docker save to tar file
- commit to create a new image

Dockerfiles

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

```
docker build
```

Container Ecosystem



OS Virtualization Benefits

- “Lightweight” virtualization. No performance overhead
- Negligible resource overhead (mem, cpu)
- VMs need resources to run guest OS
- Near-instant startup
  - Guest OS boot can take tens of seconds
  - Hypervisor optimizations can reduce it to ~ 100 ms [ClearContainers]
- Dynamic resource management
- Change cpu, mem, IO resources at run-time
- Less wasted resources
- Free resources inside a VM are considered “allocated”

OS Virtualization Drawbacks

- Linux containers not secure due to large surface area
- Resource isolation may be weak
- Leads to performance interference
- Two CPU intensive containers
- One container runs fork-bomb
- Isolation is hard
- Many common, shared resources
- /procfs . Useful for system monitoring
- Exposed to container or not?

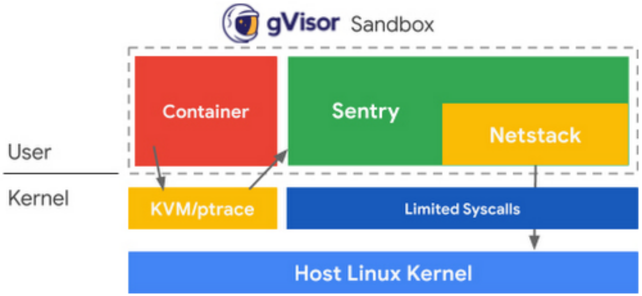
Lightweight Virtual Machines

Can we provide isolation of HW VMs and low footprint of containers?

- New trend: Stripped-down hypervisors. Reduces startup-time
- Intel ClearContainers (now Kata), Amazon Firecracker, . . .
- Target: 100ms boot time
- Most features of QEMU not needed
- Need to support only newer versions of Linux: paravirt drivers sufficient, legacy hardware (BIOS) can be removed from the VM
- Disk image transparency tackled through Plan9FS. VM can access host FS directly!
- Bypass page cache through DAX (Direct Access)
- Mmap kernel image directly from host file system and boot
- Can boot Docker images as VMs!

gVisor

- OS presents a large attack surface: generally dangerous to run in cloud environments
- Common to heavily sandbox containers via apparmor, seccomp...
- gVisor: OS system calls implemented in Go
- Intercept application syscalls and either reject, filter, proxy, or safely implement



Other Virtualization Options

Library virtualization

- Implement glibc on Windows (Cygwin, mingw)
- Implement windows APIs on Linux (Wine)

System-call virtualization

- Run applications from different OS
- Intercept syscalls made by application and reimplement them
- Linux apps on FreeBSD, SmartOS (Solaris)
- Linux apps on Windows