CS 370: OPERATING SYSTEMS
[CONTAINERS]

Shrideep Pallickara
Computer Science
Colorado State University

1

Topics covered in this lecture

- Containers
  - How they differ from virtualization
  - Key enabling concepts
    - Cgroups
    - Namespaces
    - Capabilities
  - Images

2

3

History

- Most of what containers accomplish is based on **cgroups** (control groups)
  - Created by Google Engineers Rohit Seth and Paul Menage; work started in 2006
  - Merged into the Linux kernel mainline in version 2.6.24 (January 2008)

4

What is a container?

- Ultimately, just a group of processes
- As such, a container can do anything that processes can do
  - Albeit with restrictions enforced by the kernel

Why containers?

- "build"-ing software can be difficult
  - The build process may have dependencies on specific versions of libraries and such
  - Solution: Package all dependencies in the container
- Deploying software can also be difficult
  - You may use a specific feature of Python 3.6, and if the server only has Python 3.5 the deployment breaks
  - Solution: Deploy a container

Containers have their own file system

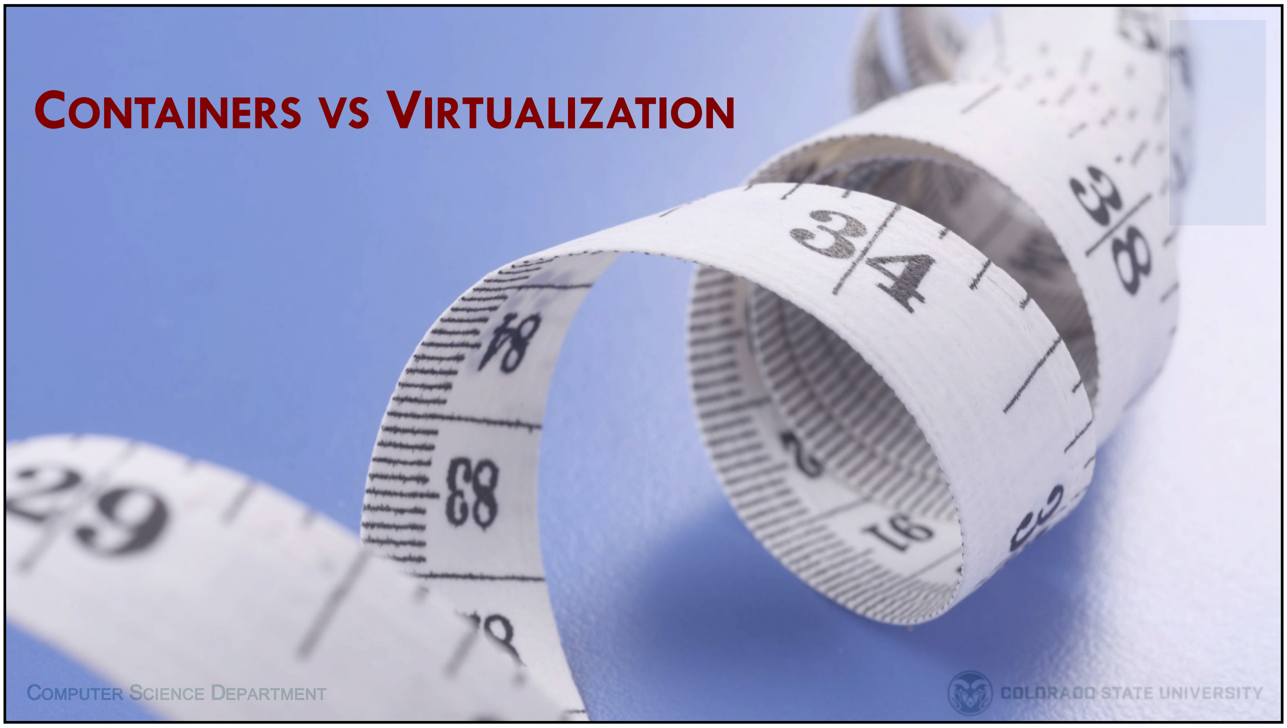
- Use this to include every dependency
- A container image is a compressed representation (usually tar) of the filesystem

7

Containers are not magic

- A running container shares the kernel of the host machine it is running on
  - A containerized application designed to run on a host with a Windows kernel will not run on a Linux host
- Caveat: This is an area that is quite fluid and changes are afoot

8



9

How are containers different from virtual machines? [1/2]

- Every VM requires its own **dedicated OS**
  - Every OS consumes CPU, RAM and storage that could otherwise be used to power more applications
  - Every OS needs patching and monitoring; in some cases, every OS requires a license
  - These overheads add up
- VMs are slower to boot
  - Migrating and moving VM workloads between hypervisors and cloud platforms can be harder than it needs to be

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT | CONTAINERS | L26-B.10

10

How are containers different from virtual machines? [2/2]

- The container is roughly analogous to the VM
- Major difference?
  - Every container *does not* require its own full-blown OS
  - In fact, all containers on a single host **share a single OS**

Container Engines

- The **container engine** (e.g. Docker Engine) is the infrastructure plumbing software that runs and orchestrates containers
- Core container runtime that runs containers

The hypervisor MO

- Once the hypervisor boots, it lays claim to all physical resources on the system such as CPU, RAM, storage, and NICs
- The hypervisor then carves these hardware resources into virtual versions that look-smell-and-feel exactly like the real thing
  - Packages them into a software construct called the VM
  - We then take those VMs and install an operating system and application on each one

What about the container engine?

- The container engine then takes OS resources such as the process tree, the filesystem, and the network stack, and carves them up into **secure, isolated** constructs called containers
- Each container looks-smells-and-feels just like a real OS
- Inside of each container we can run an application

Hypervisors versus container engines

- At a high level, we can say that hypervisors perform **hardware virtualization**
  - Carve up physical hardware resources into virtual versions
- Containers perform **OS virtualization**
  - Carve up OS resources into virtual versions

ENABLING CONSTRUCTS IN CONTAINERS

Key enabling constructs in Container

- Namespaces
- Cgroups
- Capabilities
- seccomp

NAMESPACES

Kernel namespaces are at the very heart of containers

- Lets us slice up OS so that it looks and feels like multiple isolated operating systems
- This lets us do really cool things like
  - Run multiple web servers on the same OS without having port conflicts
  - Multiple applications on the same OS without them fighting over shared config files and shared libraries
- Docker containers are an **organized collection** of namespaces

Namespaces [1/3]

- Process ID namespace
  - Use the pid namespace to provide **isolated process trees** for each container.
  - Every container gets its own process tree, i.e. every container can have its own PID 1
  - PID namespaces also mean that a container cannot see or access to the process tree of other containers, or the host it's running on
- Network namespace
  - Uses the net namespace to provide each container its own isolated network stack
  - This stack includes: interfaces, IP addresses, port ranges, and routing tables
  - For example, every container gets its own eth0 interface with its own unique IP and range of ports

Namespaces [2/3]

- Mount namespace:
  - Every container gets its own unique isolated root / filesystem
  - Every container can have its own /etc, /var, /dev etc.
  - Processes inside of a container cannot access the mount namespace of the Linux host or other containers
    - They can **only see and access their own isolated mount namespace**
- Inter-process Communication namespace
  - Uses the ipc namespace for shared memory access within a container
  - Also isolates the container from shared memory outside of the container

Namespaces [3/3]

- User namespace
  - Use user namespaces to map users inside of a container to different users on the Linux host
  - A common example is mapping the root user of a container to a non-root user on the Linux host
- UTS (UNIX Timesharing System) namespace
  - Use the uts namespace to provide each container with its own hostname

CGROUPS

23

Cgroups

- If namespaces are about isolation, control groups (cgroups) are about **setting limits**
- Containers are isolated from each other but all share a common set of OS resources — things like CPU, RAM and disk I/O
- Cgroups let us set limits on each of these so that a single container cannot use all of the CPU, RAM, or storage I/O of the host

24

Think of containers as similar to rooms in a hotel

- Yes, each room is isolated, but each room also shares a common set of resources
  - E.g. water supply, electricity supply, shared swimming pool, shared gym, shared breakfast bar etc.
- Cgroups let us set limits so that
  - No single container can use all of the water or eat everything at the breakfast bar

CAPABILITIES

Capabilities

- Arduous to run containers as non-root — non-root is so powerless it's practically useless
- What we need is a technology that lets us pick and choose which root powers our containers need in order to run?
  - **Capabilities**

Under the hood, the Linux root account is made up of a long list of capabilities

- CAP_CHOWN lets you change file ownership
- CAP_NET_BIND_SERVICE lets you bind a socket to low numbered network ports
- CAP_SETUID lets you elevate the privilege level of a process
- CAP_SYS_BOOT lets you reboot the system
- Etc, etc …
- Container engines work with capabilities so that you can run containers as root, but strip out capabilities that you don't need

seccomp

- Rarely used system calls can help an attacker
- seccomp
  - Limit the system calls a container can make to the host's kernel
- Docker blocks dozens of system calls by default
- You can customize seccomp profiles

seccomp-bpf

- Lets you run a function before every system call
- The function decides if that syscall is allowed
- Ultimately there are two ways to block scary system calls
  - Limit the containers capabilities
  - Set of seccomp-bpf permissions list
  - Best practice? Doing both

DOCKER IMAGES

Images are considered build-time constructs, whereas containers are run-time constructs

- Images are made up of **multiple layers** that get stacked on top of each other and represented as a single object
- Inside the image is a cut-down OS and all of the files and dependencies required to run an application
  - Because containers are intended to be fast and lightweight, images tend to be small.

The whole purpose of a container is to run an application or service

- The image a container is created from must contain all OS and application files required to run the app/service
- However, containers are all about being fast and lightweight
  - So images they're built from are usually small and stripped of all non-essential parts
- All containers running on a Docker host share access to the host's kernel
  - Images contain just enough OS

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT CONTAINERS L26-B.33

33

Images and Layers [1/2]

- A Docker image is just a bunch of loosely-connected read-only **layers**
- All Docker images start with a base layer, and as changes are made and new content is added, new layers are added on top
  - You might create a new image based off Ubuntu Linux 16.04.
    - This would be your image's first layer.
    - If you later add the Python package, this would be added as a second layer on top of the base layer.
    - If you then added a security patch, this would be added as a third layer at the top
- Docker takes care of stacking these layers and representing them as a single unified object

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT CONTAINERS L26-B.34

34

Images and Layers [2/2]

- As additional layers are added, the image is always the **combination** of all layers
- Docker employs a storage driver (snapshotter in newer versions) that is responsible for stacking layers and presenting them as a single unified filesystem

Multi-Architecture Images

- As Docker grew, things started getting complex — especially when new platforms and architectures, such as Windows, ARM, were added
- Have to think about whether the image we're pulling is built for the architecture we're running on
- Docker (image and registry specs) now supports **multi-architecture images**
  - A single image can have an image for Linux on x64, Linux on PowerPC, Windows x64, ARM etc.

Working with images

- The process of getting images onto a Docker host is called **pulling**
- Docker images are stored in image registries. The most common registry is Docker Hub (https://hub.docker.com)

Commands with Images

- `docker image ls` lists all of the images stored in your Docker host's local cache
  - To see the SHA256 digests of images add the --digests flag.
- `docker image inspect` gives you all of the glorious details of an image — layer data and metadata
- `docker image rm` is the command to delete images

The contents of this slide-set are based on the following references

- Nigel Poulton. *Docker Deep Dive.* ISBN: 978-1521822807 1st Edition. 2017. Chapters [1, 6, and 15]
- Julia Evans. How Containers Work! E-zine. Available from Wizard Zines. 2020.