CCGC 5001 - Virtualization

Module 3: Intro to Containers

HUMBER

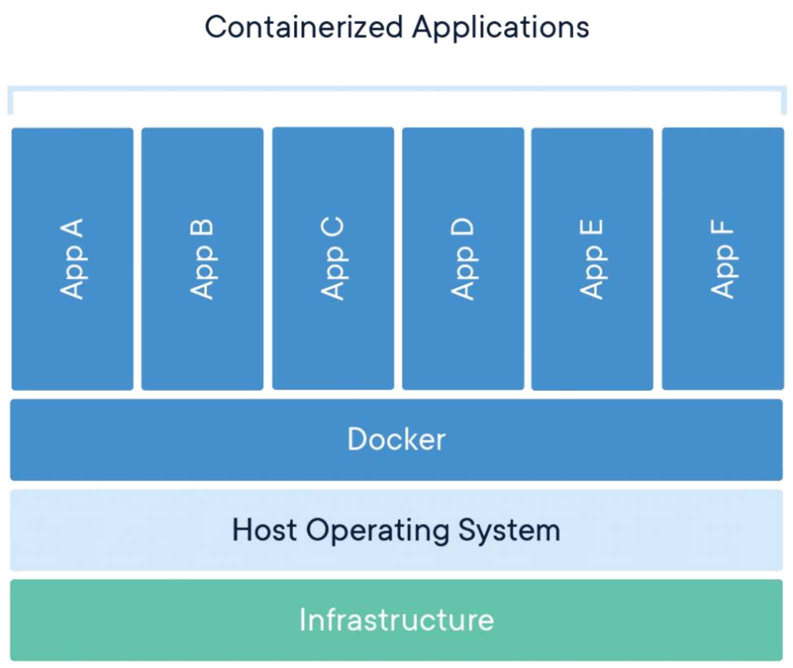
Module objectives

At the end of this module, you should be able to:

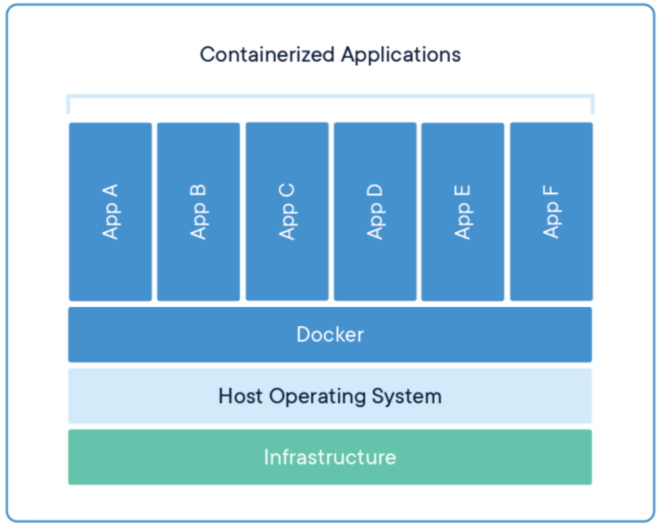
- Explain containers and list their benefits
- Run, stop, and delete a container based on an existing image
- List containers on the system
- Inspect the metadata of a running or stopped container
- Run a process such as /bin/sh in an already-running container
- Explain an anatomy of container

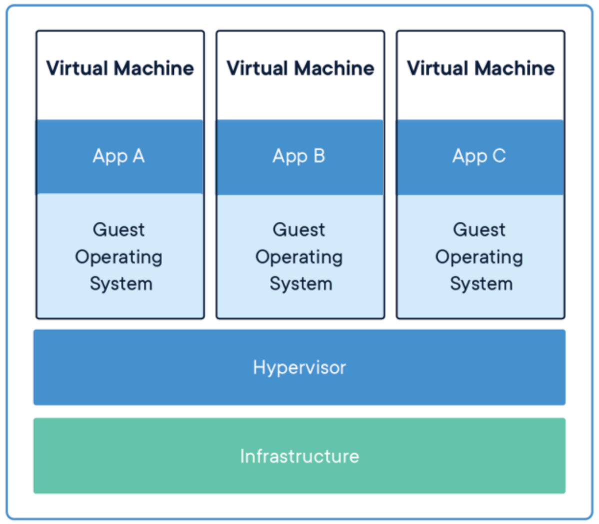
Containers

Containers are lightweight software components that bundle the application, its dependencies, and its configuration in a single image, running in isolated user environments on a traditional OS on a server or in a virtualized environment.



Containers vs. VMs





Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable, faster and efficient.

Source: https://www.docker.com/resources/what-container

The benefits of containerization

Containerization of applications bring many benefits, including the following:

- Portability between different platforms and clouds—it's truly write once, run anywhere.
- Efficiency through using far fewer resources than VMs.
- Agility that allows developers to integrate with their existing DevOps environment.
- Higher speed in the delivery of enhancements.
- Improved security by isolating applications from the host system and from each other.
- Faster app start-up and easier scaling.
- Flexibility to work on virtualized infrastructures or on bare metal servers
- Easier management since install, upgrade, and rollback processes are built into the orchestration platform.

Starting, stopping, and removing containers

Our first container:

```
$ docker container run alpine echo "Hello World"
```

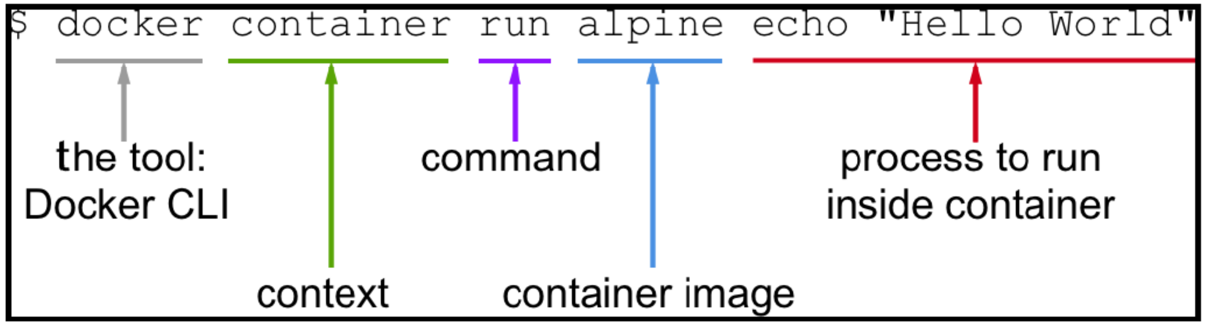
docker – The name of the Docker Command-Line Interface (CLI) tool that interact with Docker engine

container – Indicates the context we are working with

run – The command we want to execute in the given context

alpine – This is the container we want to run

echo “Hello World” – Process to run inside container


$ docker container run alpine echo "Hello World"
the tool: Docker CLI
command
process to run inside container
context
container image

Starting, stopping, and removing containers

Another container with different process

```
$ docker container run centos ping -c 5 127.0.0.1
```

You should see the following output:

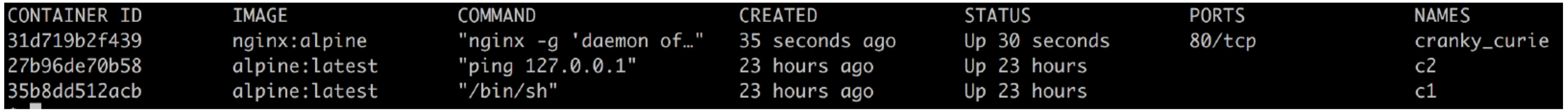
```
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
7a0437f04f83: Pull complete
Digest: sha256:5528e8b1b1719d34604c87e11dcd1c0a20bedf46e83b5632cdeac91b8c04efc1
Status: Downloaded newer image for centos:latest
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=255 time=0.020 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=255 time=0.027 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=255 time=0.028 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=255 time=0.028 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=255 time=0.026 ms

--- 127.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 107ms
rtt min/avg/max/mdev = 0.020/0.025/0.028/0.007 ms
```

Listing containers

To find out what is currently running on our host (Running state)

$ docker container ls

You should see the following output:



```
CONTAINER ID   IMAGE           COMMAND                  CREATED            STATUS           PORTS     NAMES
31d719b2f439   nginx:alpine    "nginx -g 'daemon of…"   35 seconds ago     Up 30 seconds    80/tcp    cranky_curie
27b96de70b58   alpine:latest   "ping 127.0.0.1"         23 hours ago       Up 23 hours                c2
35b8dd512acb   alpine:latest   "/bin/sh"                23 hours ago       Up 23 hours                c1
```

Container ID	This is the unique ID of the container. It is an SHA-256.
Image	This is the name of the container image from which this container is instantiated.
Command	This is the command that is used to run the main process in the container.
Created	This is the date and time when the container was created.
Status	This is the status of the container (created, restarting, running, removing, paused, exited, or dead).
Ports	This is the list of container ports that have been mapped to the host.
Names	This is the name assigned to this container (multiple names are possible).

Listing containers

Listing containers in any state, such as created, running, or exited

```
$ docker container ls -a
```

Listing the IDs of all containers, use –q parameter

```
$ docker container ls -q
```

What is the action of the following command?

```
$ docker container rm -f $(docker container ls -a -q)
```

Stopping and starting containers

Let’s run the following container

```
$ docker container run -d --name trivia fundamentalsofdocker/trivia:ed2
```

To stop the preceding container

```
$ docker container stop trivia
```

It takes about 10 seconds to stop this container. Why?

Docker sends a Linux SIGTERM signal to the main process inside the container.

If unsuccessful, Docker waits for 10 seconds and then sends SIGKILL, which will kill the process forcefully.

To start the container

```
$ docker container start trivia
```

Removing containers

The command to remove a container is as follows:

```
$ docker container rm <container ID>
```

Another command to remove a container

```
$ docker container rm <container name>
```

What if you are unable to remove a container?

Use a force removal with command-line parameter –f or --force.

Inspecting containers

Containers are runtime instances of an image and have a lot of associated data that characterizes their behavior.

To get more information about a specific container, we can use the inspect command.

```
$ docker container inspect trivia
```

The response is a big JSON object full of details.

Inspecting containers

```
[
        {
                "Id": "48630a3bf188...",
                ...
                "State": {
                        "Status": "running",
                        "Running": true,
                ...
                },
                "Image": "sha256:bbc92c8f014d605...",
                ...
                "Mounts": [],
                "Config": {
                        "Hostname": "48630a3bf188",
                        "Domainname": "",
                        ...
                },
                "NetworkSettings": {
                        "Bridge": "",
                        "SandboxID": "82aed83429263ceb6e6e...",
                        ...
                }
        }
]
```

Inspecting containers

What if I want to see just the network settings?

```
$ docker container inspect -f "{{json .NetworkSettings}}" trivia | jq
```

The –f or --filter parameter is used to define the filter.

Exec into a running container

Sometimes, we want to run another process inside an already running container.

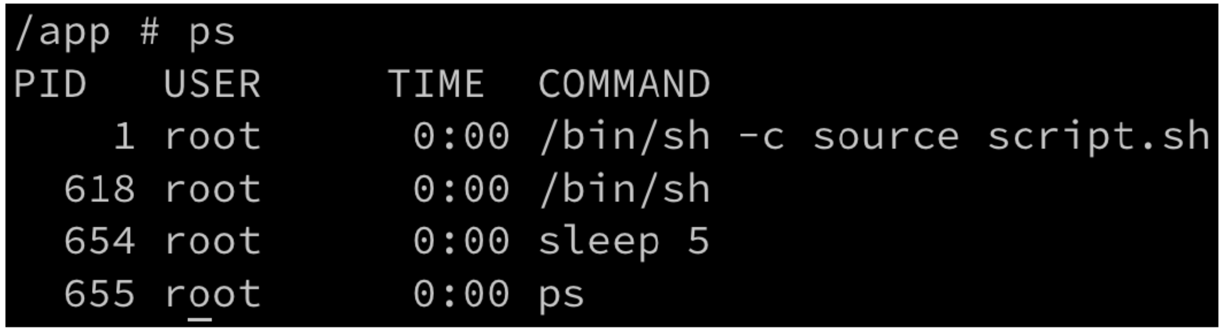
```
$ docker container exec -i -t trivia /bin/sh
```

The -i flag allows us to run the additional process interactively

The -t flag provide us a TTY (a Terminal emulator) for the command

The process we want to run is /bin/sh

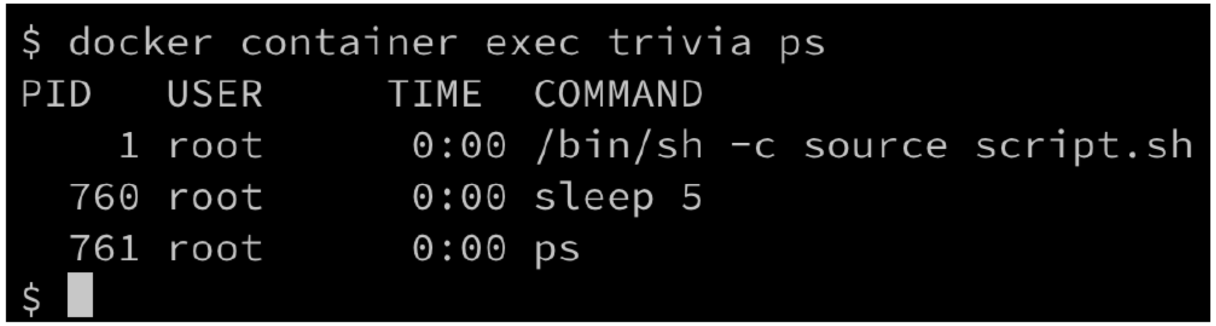
```
/app # ps
```



```
/app # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh -c source script.sh
  618 root      0:00 /bin/sh
  654 root      0:00 sleep 5
  655 root      0:00 ps
```

Exec into a running container

$ docker container exec trivia ps



```
$ docker container exec trivia ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh -c source script.sh
  760 root      0:00 sleep 5
  761 root      0:00 ps
$ 
```

Attaching to a running container

We can use the attach command to attach our Terminal's standard input and output to a running container using the ID or name of the container.

```
$ docker container attach trivia
```

Let's run another container, this time an Nginx web server

```
$ docker container run -d --name nginx -p 8080:80 nginx:alpine
```

We can access Nginx using the curl tool and running this command

```
$ curl -4 localhost:8080
```

Attaching to a running container

Let’s attach our Terminal to the nginx container

```
$ docker container attach nginx
```

How do we get logging output of Nginx?

Open another Terminal and type the following command

```
$ for n in {1..10}; do curl -4 localhost:8080; done
```

```
172.17.0.1 - - [01/Feb/2021:04:45:37 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:37 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:37 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:37 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:37 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:38 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:38 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:38 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:38 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
172.17.0.1 - - [01/Feb/2021:04:45:38 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.61.1" "-"
```

Anatomy of containers

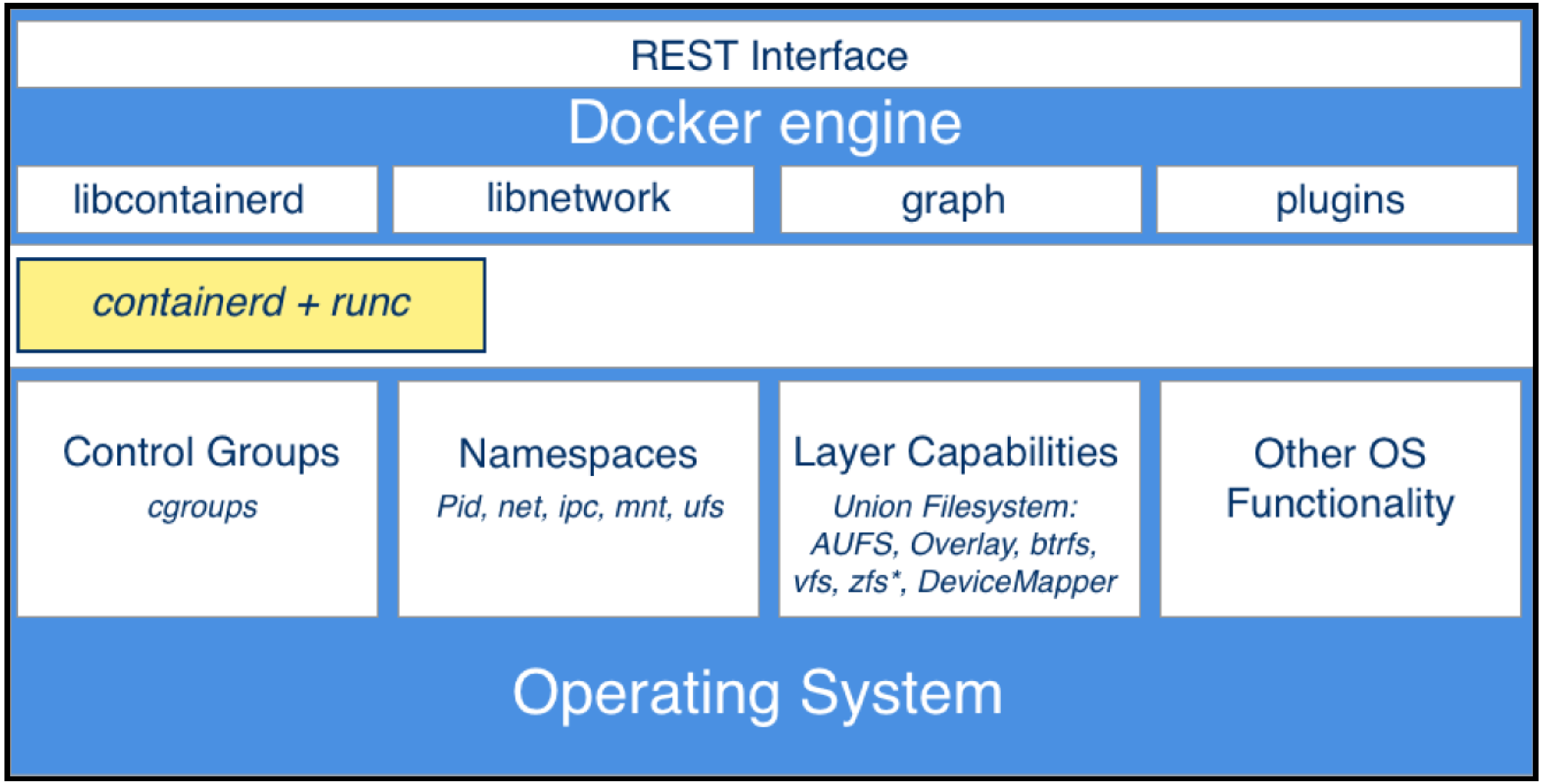
Containers are specially encapsulated and secure processes

Containers leverage features and primitives available in the Linux OS

- namespaces
- cgroups

All processes running in containers only share the same Linux kernel of the underlying host operating system

Architecture



Namespaces

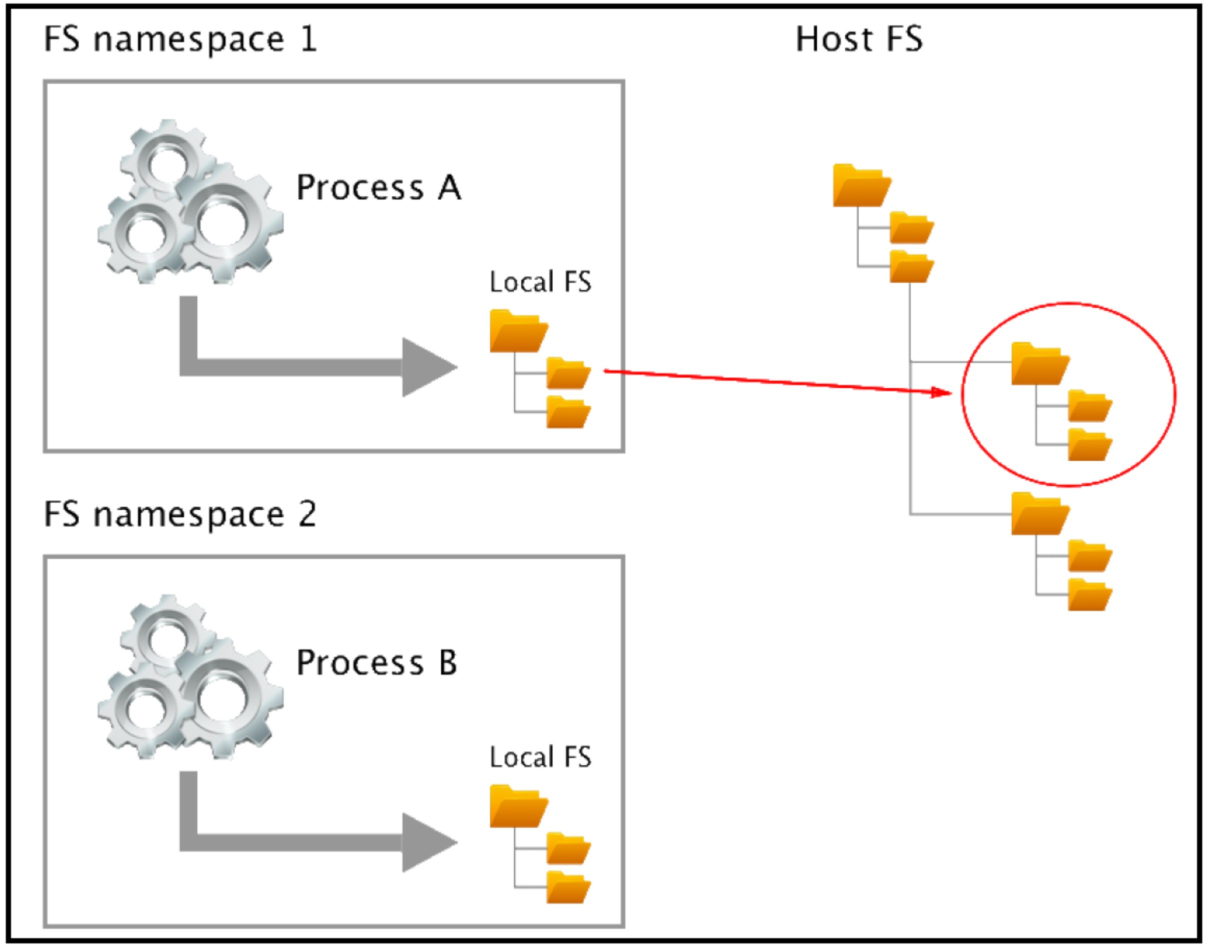
Namespace is an abstraction of global resources

- Filesystems
- Network access
- Process tree (PID namespaces)

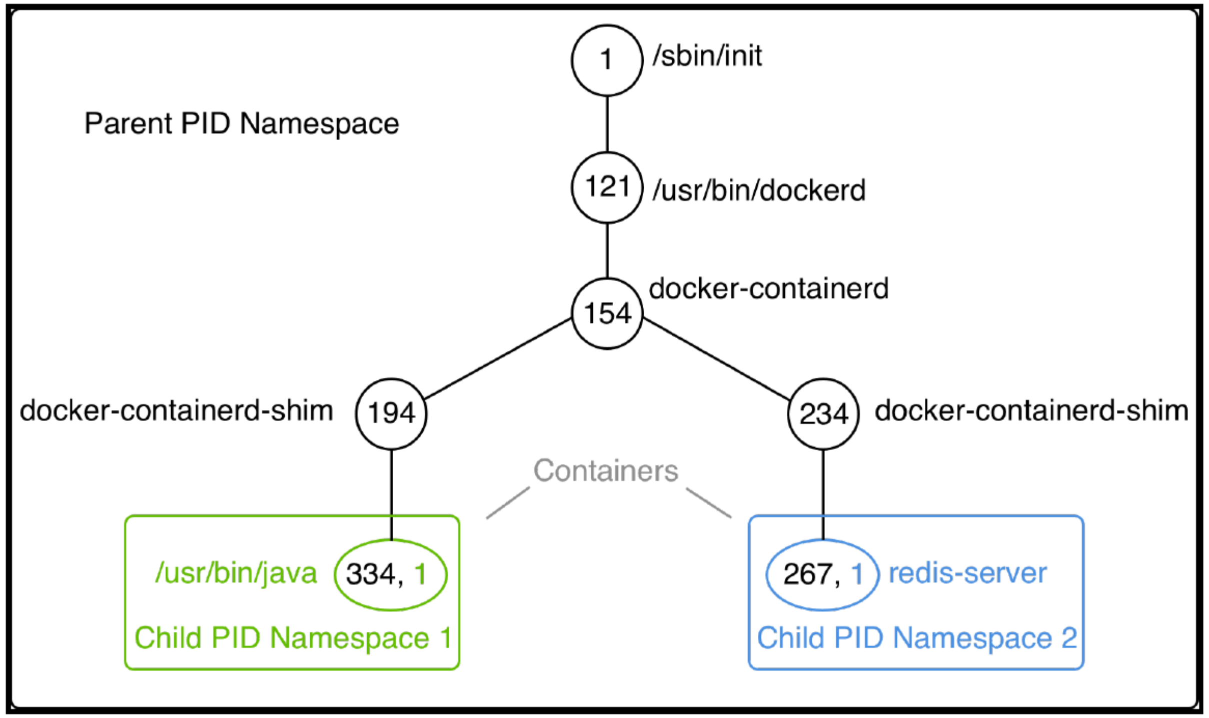
Linux system is initiated with a single instance of each namespace type

After initialization, additional namespaces can be created or joined

Filesystem namespace



PID namespace



Control groups (cgroups)

Cgroups are used to limit, manage, and isolate resource usage of collections of processes on a system

- CPU time
- System memory
- Network bandwidth

With cgroups, we can avoid noisy neighbor problem

Union filesystem (Unionfs)

Unionfs forms the backbone of container images

Allows files and directories of distinct filesystems to be overlaid to form a single coherent filesystem

runC

1

runC is a lightweight, portable container runtime

2

Provides support for Linux namespaces as well as for all security features available on Linux

3

runC is a tool for spawning and running containers as per Open Container Initiative (OCI)

Containerd

Containerd builds on top of runC and adds higher-level features

- Image transfer and storage
- Container execution
- Network and storage attachments

Manages the life cycle of containers

Most widely used container runtime

Module summary

In summary, in this module, you learned:

- How to work with containers
- How to run, stop, start, and remove a container
- How to run an arbitrary process in an already running container
- Investigated how containers work and what features of Linux OS they leverage

Thank you