CLOUD COMPUTING

Lightweight virtualization – Containers, namespaces, cgroups
&
Deployment of cloud native applications through Docker, UnionFS

Dr. Prafullata Kiran Auradkar

Department of Computer Science and Engineering

Acknowledgements:

Significant information in the slide deck presented through the Unit 2 of the course have been created by **Dr. H.L. Phalachandra** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for classroom presentation only.

Containers – Definitions

- Linux Containers or LXC is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) which can run multiple workloads, on a control host running a single linux OS instance
- LXC provides a virtual environment that has its own process and network space and does not create a full-fledged virtual machine.
- LXC uses Linux kernel cgroups and namespace isolation functionality to achieve the same.

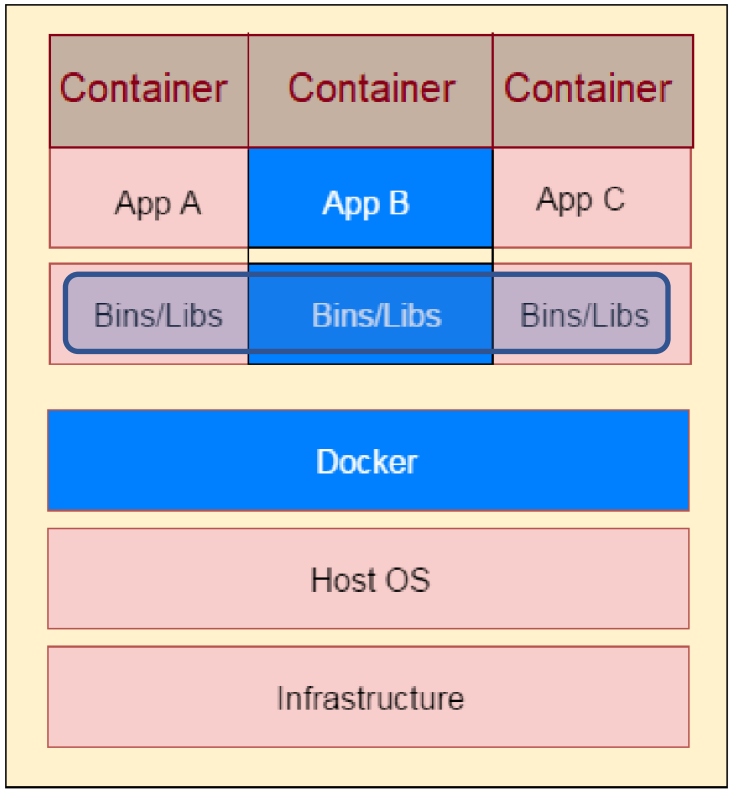
Motivation

- VMs support the objective of virtualizing the physical systems and allowing multiple tenants/applications to be isolated and share the physical resources along with Access control
- One of the challenges as observed is, this isolation achieved by or provided by VM is expensive
- Traditional OSs supporting multiple application processes, but share a disk, with all the processes capable of seeing the entire filesystem with access control built on top, and also share a network.
- Containers using a light weight mechanism, provide this virtualization extending the isolation provided by our traditional OS

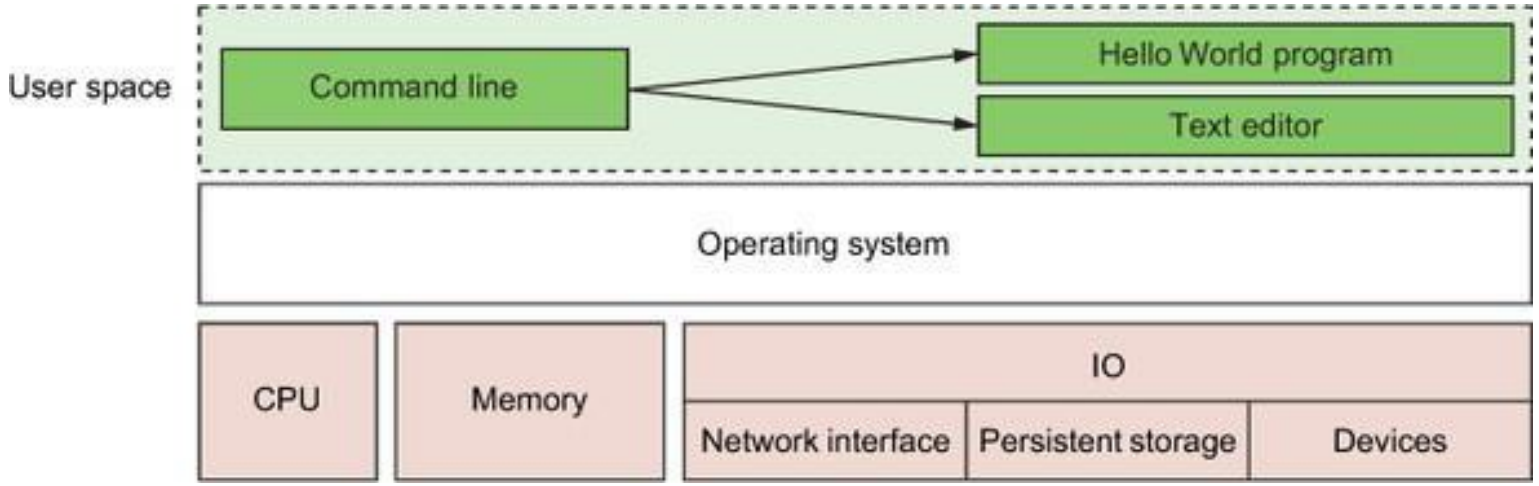
Lxc (tools) is an user space toolset for creating & managing Linux Containers – different from LXC Linux containers

Container Characteristics

- Containers sit on top of a physical server and its host OS.
- Each container shares the host OS kernel and, usually, the binaries and libraries too, but as read-only. This reduces the management overhead where a single OS needs to be maintained for bug fixes, patches, and so on.
- Containers are thus exceptionally “light”—they are only megabytes in size and take just seconds to start, versus gigabytes and minutes for a VM.
  - Container creation is similar to process creation and it has speed, agility and portability.
  - Thus containers have higher provisioning performance



Containers (Cont.)



A basic computer stack running two programs that were started from the command line

- Notice that the command-line interface, or CLI, runs in what is called user space memory, just like other programs that run on top of the operating system.
- Ideally, programs running in user space can't modify kernel space memory.
- Broadly speaking, the operating system is the interface between all user programs and the hardware that the computer is running on.

Contrasting Containers vs VMs

1. Each VM includes a separate OS image, which adds overhead in memory and storage footprint.

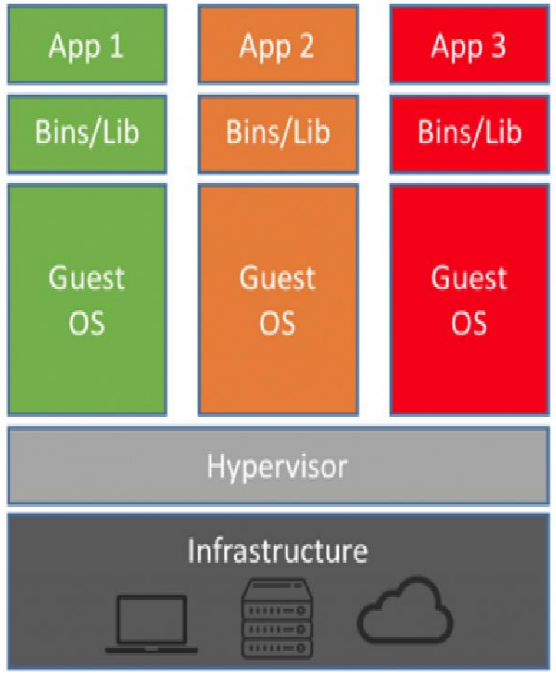
   Containers reduce management overhead as they share a common OS, only a single OS needs to be maintained for bug fixes, patches, and so on.

2. In terms of performance, VMs have to boot when provisioned making it slower, and also have I/O performance overhead

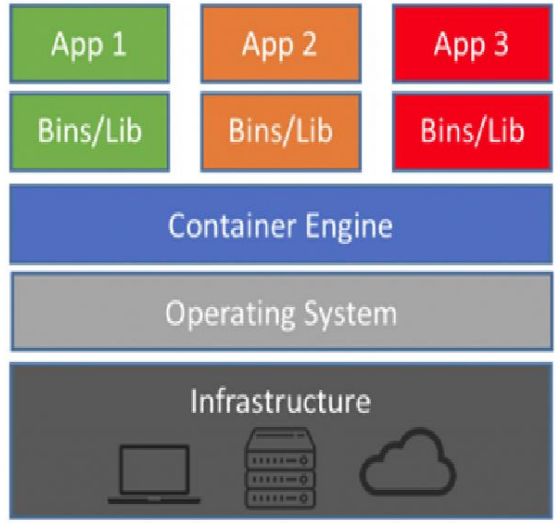
   Containers have higher performance as its creation is similar to process creation, so boots quickly and it has speed, agility and portability

3. VMs are more flexible as Hardware is virtualized to run multiple OS instances.

   Containers run on a single OS and also can support only Ubuntu containers of that type of OS where its running or containers cannot be of different OS variants



Machine Virtualization



Containers

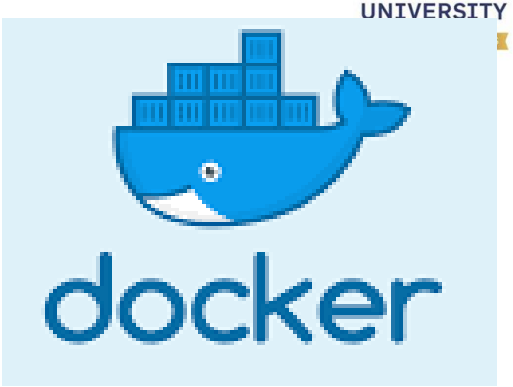
4. VMs consume more resources and come up slower than Containers which come up more quickly and consume fewer resources

CLOUD COMPUTING

VM vs Docker

Virtual Machine	Docker Container
Hardware-level process isolation	OS level process isolation
Each VM has a separate OS	Each container can share OS
Boots in minutes	Boots in seconds
VMs are of few GBs	Containers are lightweight (KBs/MBs)
Ready-made VMs are difficult to find	Pre-built docker containers are easily available
VMs can move to new host easily	Containers are destroyed and re-created rather than moving
Creating VM takes a relatively longer time	Containers can be created in seconds
More resource usage	Less resource usage

Docker



- Docker is an open platform tool which makes it easier to create, test, ship, deploy and to execute applications using containers.
- Docker containers allow us to separate the applications from the infrastructure enabling faster deployment of applications/software
- It significantly reduces the time between writing code and running it in production by providing methodologies for shipping, testing and deploying code quickly
- It can be considered as a tool that helps to package and run an application in a loosely isolated environment called a container.
- It could be looked at as a PaaS product that uses OS level virtualization to deliver S/W packages
- Docker provides the isolation and security to allow many containers to run on a single server or virtual machine
- Its typical to find between 8 -18 containers running simultaneously on a single server/VM

Docker (Cont.)

- Containers created using Docker can contain everything needed to run an application, so you do not need to rely on what is currently installed on the host system
- Docker allows these containers to be shared in a way that it would work identically.
- Docker can be used with network applications such as web servers, databases, mail servers, with terminal applications like text editors, compilers, network analysis tools, and scripts.
  - In some cases, it's even used to run GUI applications such as web browsers and productivity software.
- Docker runs on Linux software on most systems.
  - Docker is also available as a native application for both macOS and Windows
  - Docker can run native Windows applications on modern Windows server machines.

Docker Benefits

Portability, Shipping Applications

One App =

- binaries (exec, libs, etc.)
- data (assets, SQL DB, etc.)
- configs (/etc/config/files)
- logs

either in a container
or a *composition*

docker

Docker, Containers, and the Future of Application Delivery

Portability

Docker Promise: Build, Ship, Run !

- reliable deployments
- develop here, run there

docker

Docker, Containers, and the Future of Application Delivery

Build

Develop an app using Docker containers with any language and any toolchain.

Ship

Ship the "Dockerized" app and dependencies anywhere - to QA, teammates, or the cloud - without breaking anything.

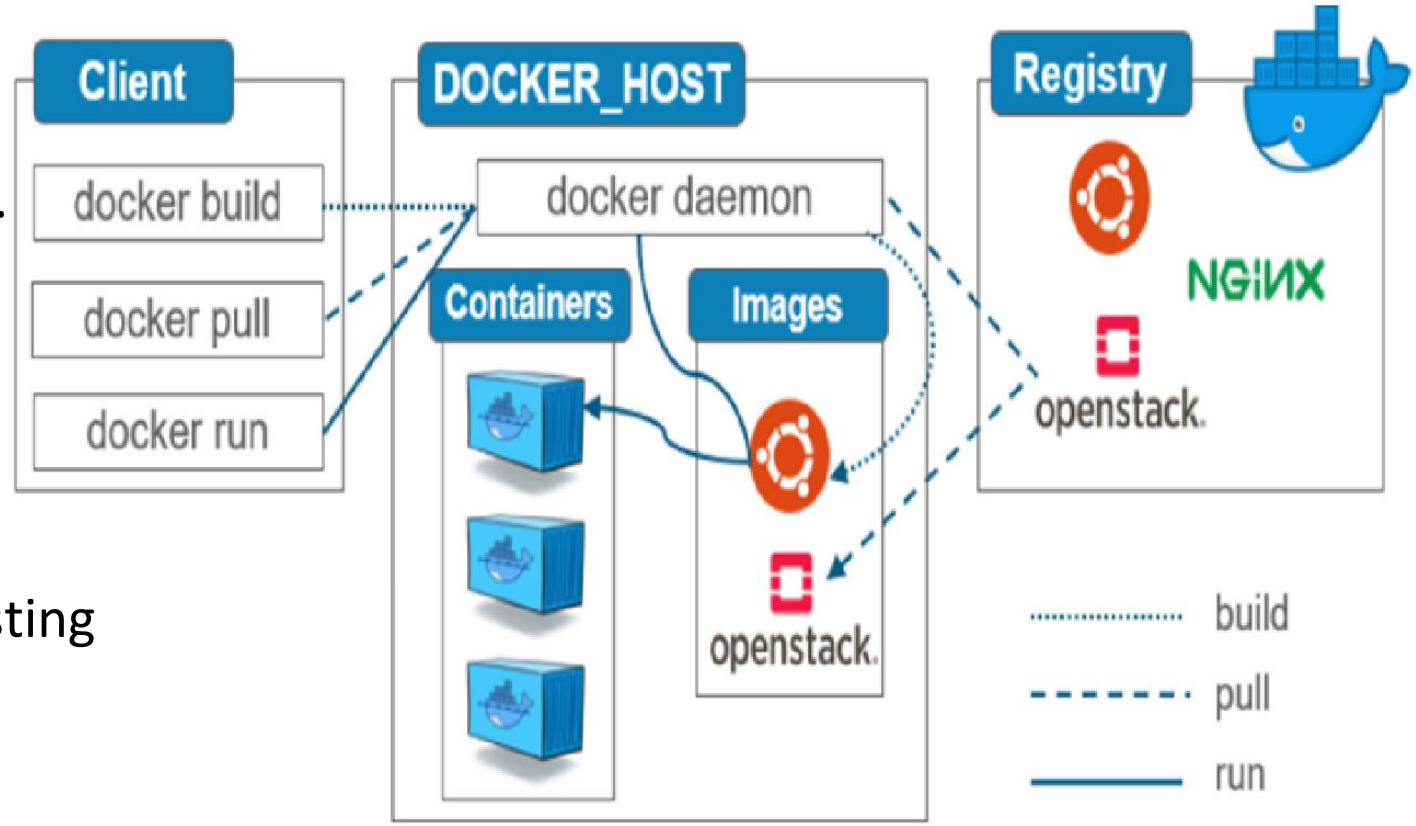
Run

Scale to 1000s of nodes, move between data centers and clouds, update with zero downtime and more.

Developers use Version Control Systems (VCS) like Git. DevOps also uses VCS for docs, scripts and Dockerfiles.
DevOps could use Dockerfile to describes how to build the image, and something like docker-compose.yml to describe how to orchestrate them.

Docker Architecture

- Docker uses a client-server architecture.
- The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.
- Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

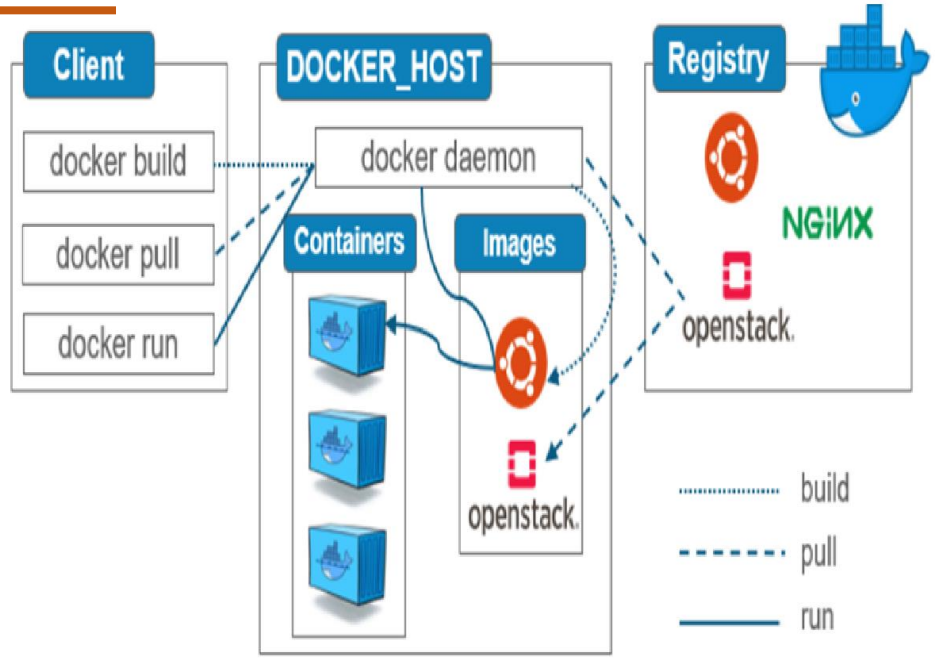


Reference:https://docs.docker.com/get-started/overview/

Docker Architecture

The Docker daemon

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
- A daemon can also communicate with other daemons to manage Docker services.



The Docker client

- The Docker client (docker) is the primary way that many Docker users interact with Docker.
- When you use commands such as **docker run**, the client sends these commands to 'dockerd' (docker daemon), which carries them out.
- The docker command uses the Docker API.
- The Docker client can communicate with more than one daemon.

Docker Architecture

Docker Host

- Docker Host has the Docker Daemon running and can host (like a private hub/registry) or connect (like to a public docker_hub/registry) to a Docker Registry which stores the Docker Images.
- The Docker Daemon running within Docker Host is responsible for the Docker Objects images and containers.

Docker Objects :

There are objects like

- Images
- Containers
- Networks
- Volumes
- Plugins and other objects which are created and used while using docker.

Docker Objects : Images

Images : This is a read-only template with instructions for creating a Docker container. This could be based on another image (available in the registry) with additional customizations. Eg. Image for a Webserver .. Original image of Ubuntu customized with installation and configuration of the webserver which is created only as a read-only image which can be deployed and an application can be run in the same.

This is done using a ***Dockerfile*** a script file which defines the syntax to indicate steps needed to create the image (and run using a run command).

Each instruction in a Dockerfile creates a ***layer*** in the image.

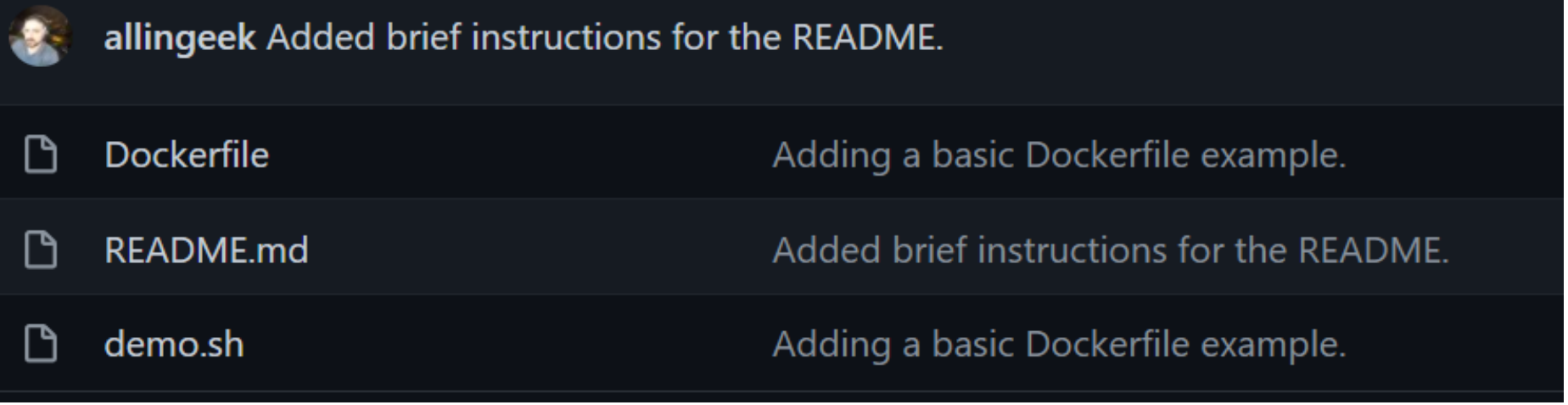
These Dockerfiles are distributed along with software that the author wants to be put into an image. In this case, you're not technically installing an image. Instead, you're following instructions to build an image.

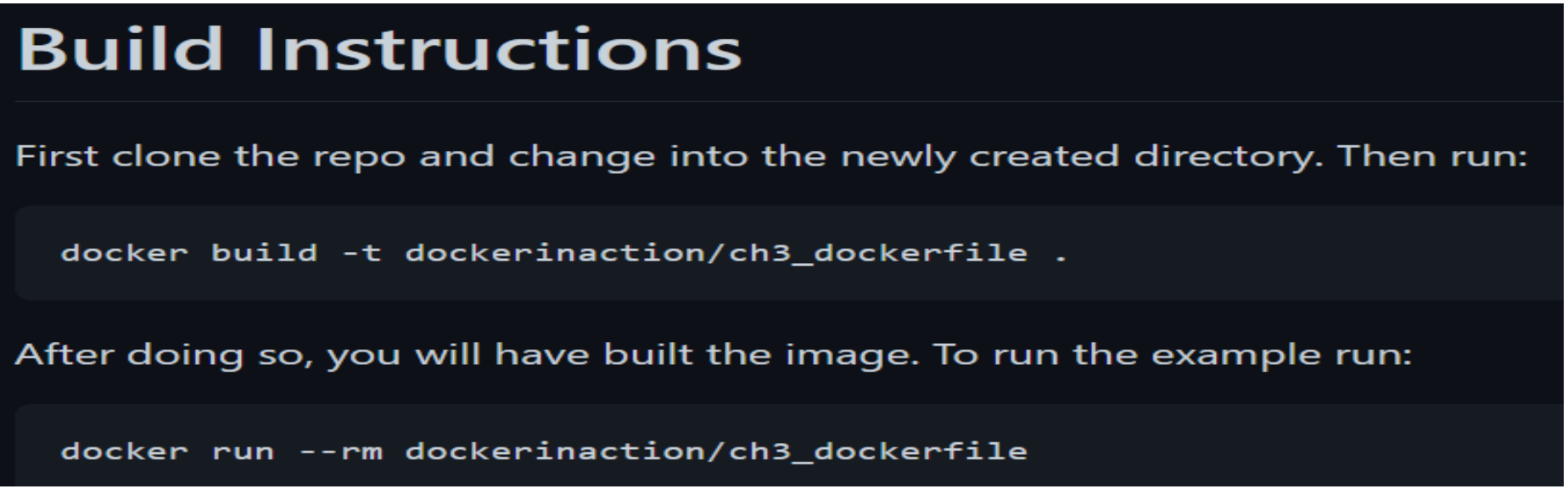
Docker Objects : Images – More on Dockerfile

- Distributing a Dockerfile is similar to distributing image files using your own distribution mechanisms. A common pattern is to distribute a Dockerfile with software from common version-control systems like Git.
  - If you have Git installed, you can try this by running an example from a public repository:
    - git clone https://github.com/dockerinaction/ch3_dockerfile.git
    - docker build -t dia_ch3/dockerfile:latest ch3_dockerfile

  In this example, you copy the project from a public source repository onto your computer and then build and install a Docker image by using the Dockerfile included with that project. The value provided to the -t option of docker build is the repository where you want to install the image.
- Building images from Dockerfile is a light way to move projects around that fits into existing workflows. Could lead to an unfavourable experience in case of a drift in dependencies between the time when the Dockerfile was authored and when an image is built on a user's computer.
- Docker Images can be removed or cleaned up with
  - docker rmi dia_ch3/dockerfile
  - rm -rf ch3_dockerfile

Docker Commands – Video Ref: https://www.youtube.com/watch?v=nXV6qihj5uw


allingeek Added brief instructions for the README.
Dockerfile
Adding a basic Dockerfile example.
README.md
Added brief instructions for the README.
demo.sh
Adding a basic Dockerfile example.


Build Instructions
First clone the repo and change into the newly created directory. Then run:
docker build -t dockerinaction/ch3_dockerfile .
After doing so, you will have built the image. To run the example run:
docker run --rm dockerinaction/ch3_dockerfile

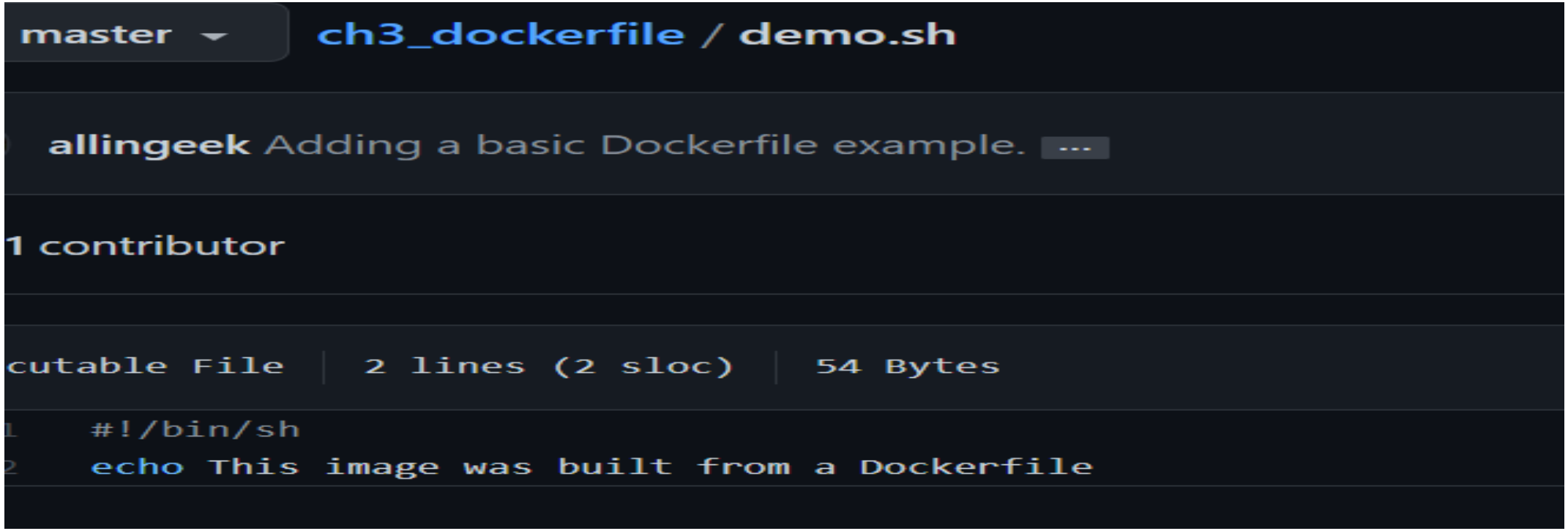
```
ubuntu@ubuntu2004:~$ git clone https://github.com/dockerinaction/ch3_dockerfile.git
Cloning into 'ch3_dockerfile'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 0), reused 7 (delta 0), pack-reused 0
Unpacking objects: 100% (7/7), 891 bytes | 111.00 KiB/s, done.
```

```
ubuntu@ubuntu2004:~$ sudo docker build -t dia_ch3/dockerfile:latest ch3_dockerfile
Sending build context to Docker daemon  62.98kB
Step 1/5 : FROM busybox:latest
latest: Pulling from library/busybox
009932687766: Pull complete
Digest: sha256:afcc7f1ac1b49db317a7196c902e61c6c3c4607d63599ee1a82d702d249a0ccb
Status: Downloaded newer image for busybox:latest
 ---> ec3f0931a6e6
Step 2/5 : MAINTAINER dia@allingeek.com
 ---> Running in 18a774adda1f
Removing intermediate container 18a774adda1f
 ---> 259b92c4282e
Step 3/5 : ADD demo.sh /demo/
 ---> b3483957bf84
Step 4/5 : WORKDIR /demo/
 ---> Running in d514ea2c6bb1
Removing intermediate container d514ea2c6bb1
 ---> 8c9f6c93f0bb
Step 5/5 : CMD ./demo.sh
 ---> Running in e104b9c20d20
Removing intermediate container e104b9c20d20
 ---> 0c8969d1290c
Successfully built 0c8969d1290c
Successfully tagged dia_ch3/dockerfile:latest
ubuntu@ubuntu2004:~$ S
```

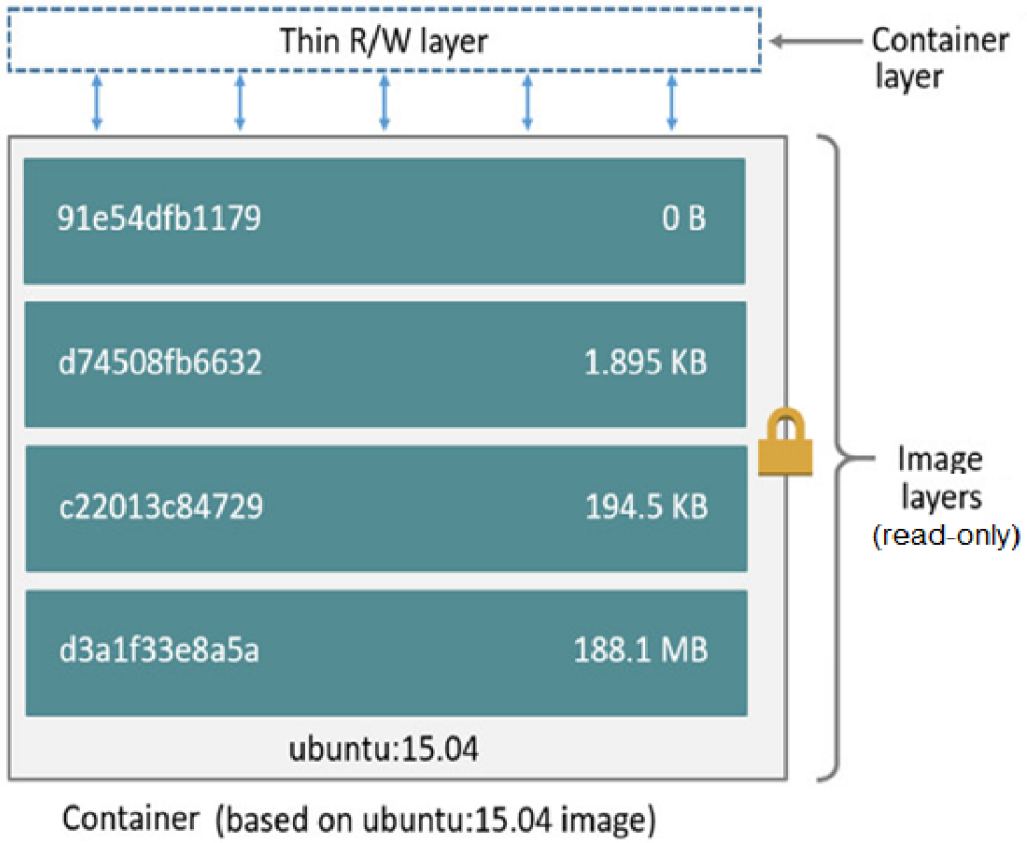
```
ubuntu@ubuntu2004:~$ sudo docker images
REPOSITORY              TAG       IMAGE ID       CREATED             SIZE
dia_ch3/dockerfile      latest    0c8969d1290c   36 minutes ago      1.24MB
busybox                 latest    ec3f0931a6e6   9 days ago          1.24MB
hello-world             latest    feb5d9fea6a5   4 months ago        13.3kB
ubuntu@ubuntu2004:~$
```

```
ubuntu@ubuntu2004:~$ sudo docker run --rm dia_ch3/dockerfile
This image was built from a Dockerfile
ubuntu@ubuntu2004:~$
```

```
ubuntu@ubuntu2004:~/ch3_dockerfile$ cat Dockerfile
FROM busybox:latest
MAINTAINER dia@allingeek.com
ADD demo.sh /demo/
WORKDIR /demo/
CMD ./demo.sh
ubuntu@ubuntu2004:~/ch3_dockerfile$
```



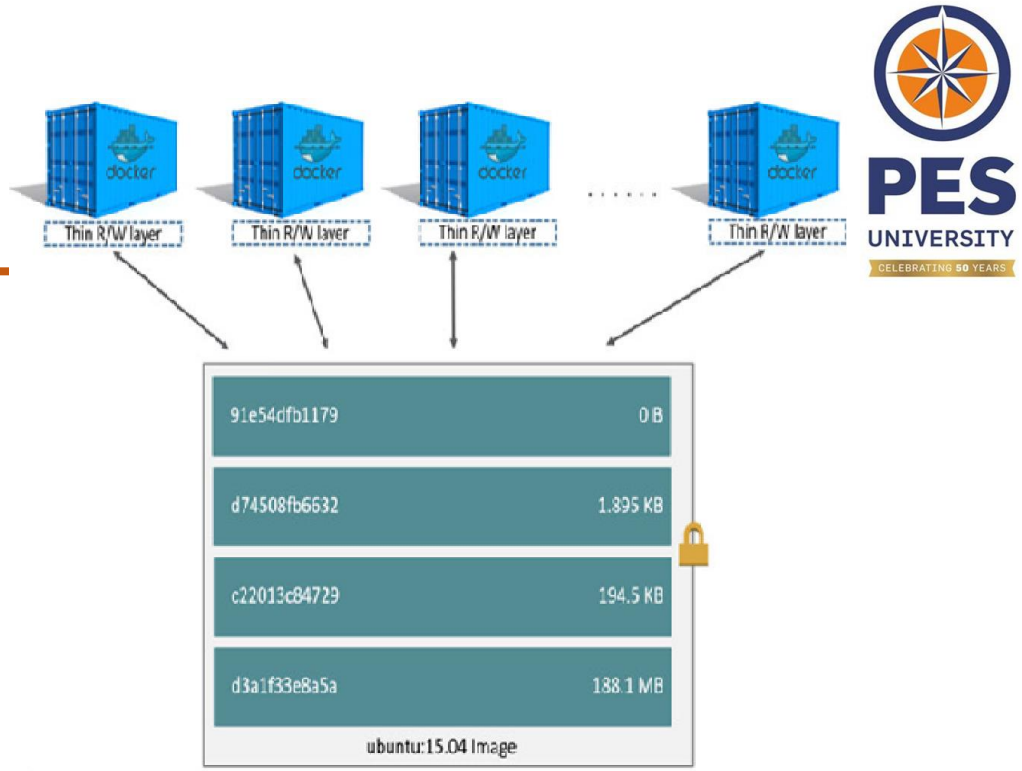
Docker Objects : Images - layers



- Specification for a Docker Image is stored in Dockerfile
  - Should be only one for a container
  - Only the definition of the image
- Image is built from Dockerfile
- Specifies the read-only file systems in which various programs are installed E.g. web server + libraries
- Each instruction in a Dockerfile creates a ***layer*** in the image.
- A layer is set of files and file metadata that is packaged and distributed as an atomic unit.
  - Internally, Docker treats each layer like an image, and layers are often called intermediate images.
  - You can even promote a layer to an image by tagging it.
  - Most layers build upon a parent layer by applying filesystem changes to the parent.
  - Whenever there is a change in the Dockerfile and the image is rebuilt, only the ***incremental changes are rebuilt*** (making it light weight, small and fast when compared to other virtualized technologies)

CLOUD COMPUTING

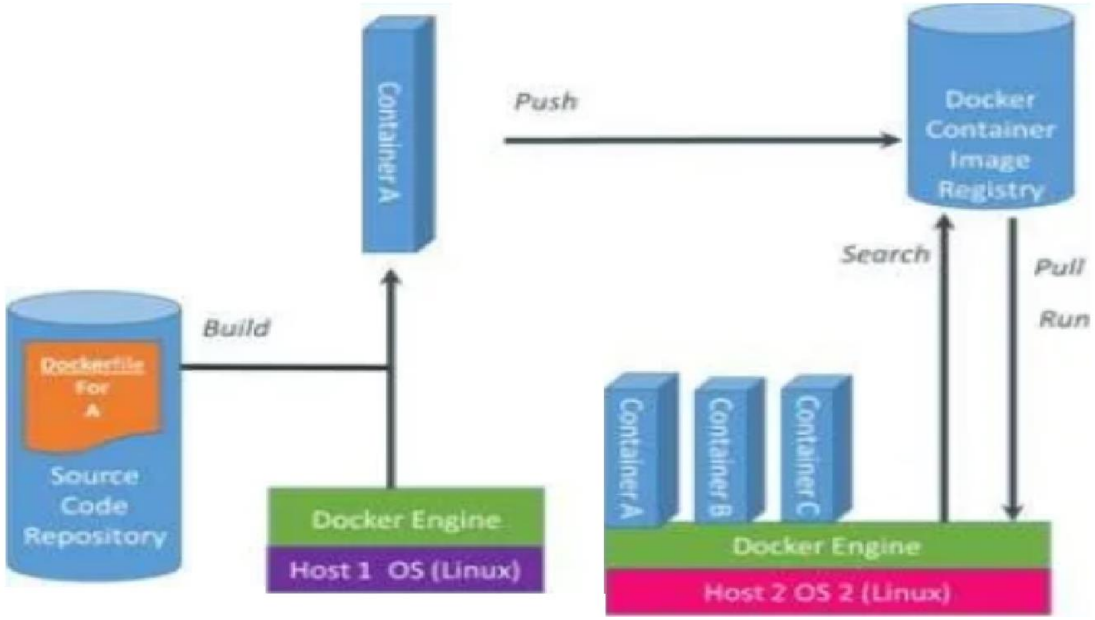
Docker Objects : Images & Docker Registries



- Image + temporary R/W file system
  - Used as temporary storage
  - Deleted when container is destroyed
- Multiple containers can use the same image & their own temporary storage

Docker registries:

- A Docker registry stores Docker images.
- ***Docker Hub*** is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default.
- You can even run your own ***private registry***.
- When you use the **docker pull or docker run** commands, the required images are pulled from your configured registry.
- When you use the docker push command, your image is pushed to your configured registry.



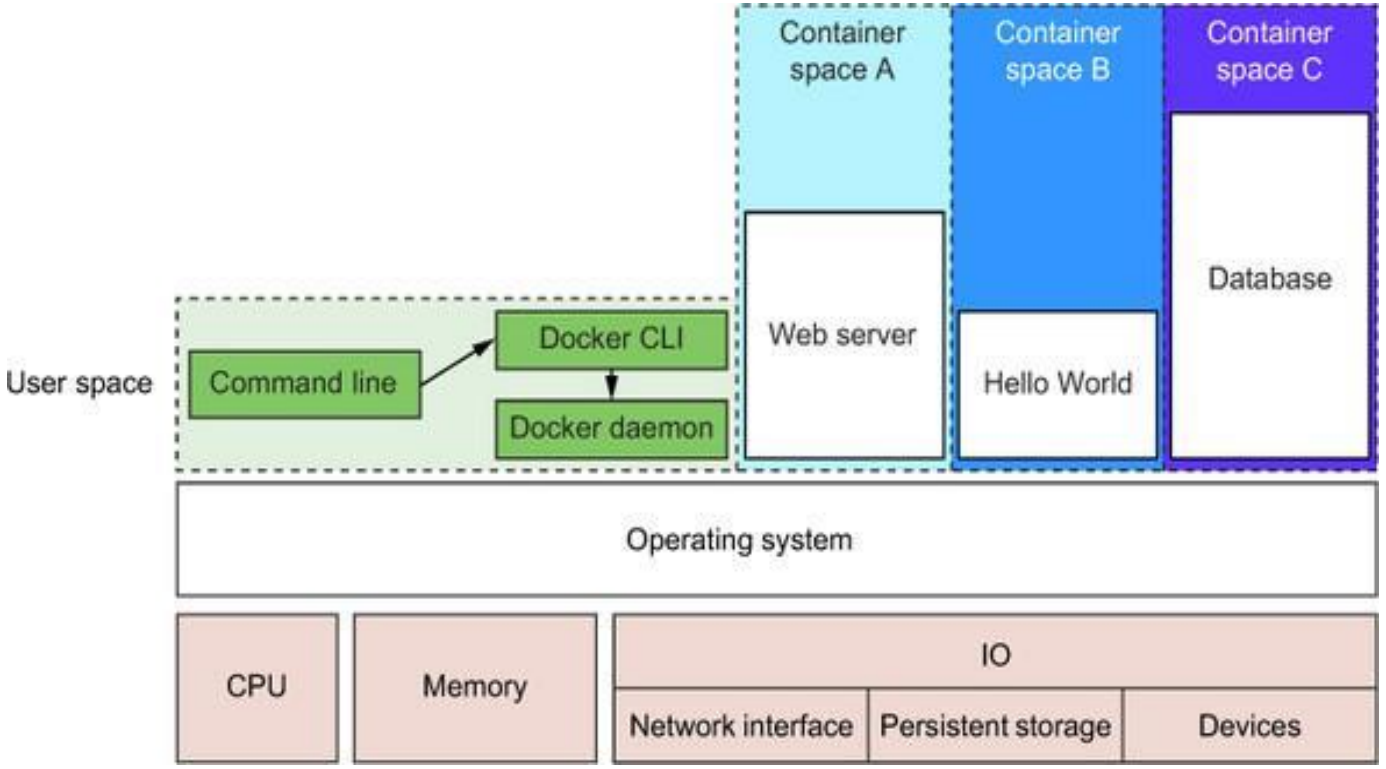
https://docs.docker.com/storage/storagedriver/

Docker Objects : Containers

- Docker Containers are the ready applications created from Docker Images. **Or** you can say they are running instances of the Images and they hold the entire package needed to run the application **Or** it could also be looked at as a runnable instance of an image **Or** an execution environment (sandbox).
- We can create, start, stop, move, or delete a container using the Docker API or CLI.
- A container can be connected to one or more networks. Storage could be attached to it, or even new image could be created based on its current state.
- A container is relatively well isolated from other containers and its host machine and this isolation can also be controlled. Processes in the container cannot access non-shared objects of other containers, and can only access a subset of the objects (like files) on the physical machine.
- Docker creates a set of name spaces when a container is created. These namespaces restrict what the objects processes in a container can see e.g. a subset of files
- A container is defined by its image and the configuration options provided to it when its created or started. When a container is removed, any changes to its state that are not stored in persistent storage will disappear

Docker running three containers on a basic Linux computer system

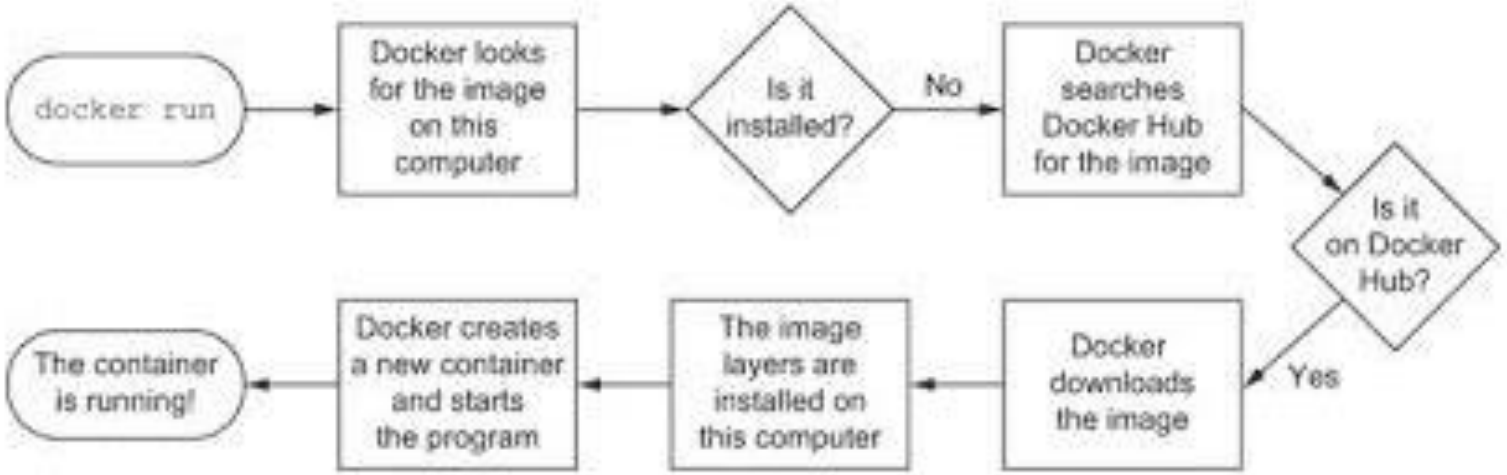
- Running Docker means running two programs in user space.
  - The first is the Docker engine that should always be running.
  - The second is the Docker CLI. This is the Docker program that users interact with to start, stop, or install software
- Each container is running as a child process of the Docker engine, wrapped with a container, and the delegate process is running in its own memory subspace of the user space.
  - Programs running inside a container can access only their own memory and resources as scoped by the container.



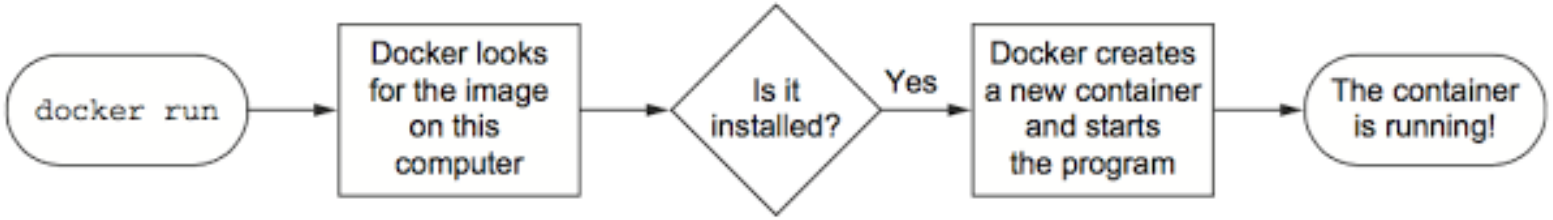
Docker can be looked at as a set of PaaS products that use OS-level virtualization to deliver software in packages called containers.

Running a program with Docker

- What happens after running **docker run**
- The image itself is a collection of files and metadata which includes the specific program to execute and other relevant configuration details

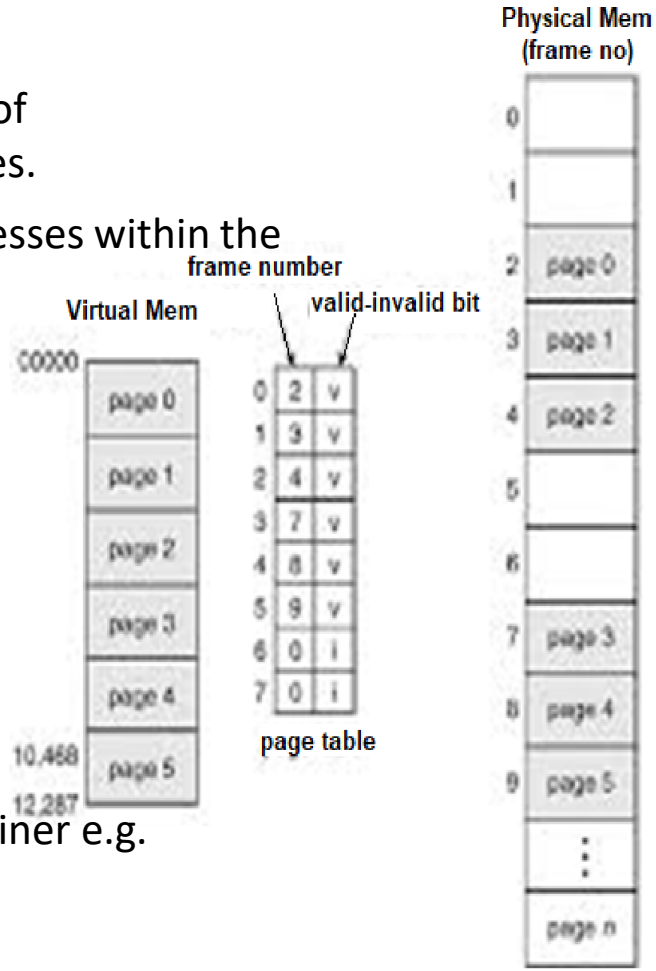


- Running docker run a second time.
- The image is already installed, so Docker can start the new container right away.



Namespaces – What's in a Name in CS?

- If you can't name an object, you can't access it. E.g. Web site – if name is hidden, can't access
- Paging : Processes can access only pages in its name spaces but cannot access physical page 1 (i.e. frame number 1 in the diagram)
- Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources and another set of processes sees a different set of resources.
- A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource
- The feature works by having the same namespace (with a name) for a group of resources and processes, but those namespaces refer to *distinct resources*
- A physical computer can have more than one namespaces (two or more)
- All the resources that a process sees can be considered a ***namespace***
  - The files seen by a process is the ***file namespace***
  - The network connections are part of ***network* namespace**
- Container Access is restricted to only subset of objects (e.g., files) on the physical machine using these namespaces. Restriction is applied on what can be seen by the object processes in a container e.g.
  - To restrict process (and container) to a subset of files use file namespace



Docker used Namespaces

- Docker builds containers at runtime using 10 major system features. Docker commands can be used to illustrate and modify features to suit the needs of the contained software and to fit the environment where the container will run.
- The specific features are as follows (a few of them are discussed in a little more detail):
  - PID namespace—Process isolation through identifiers (PID number) – not aware of what happens outside its own processes
  - UTS namespace—allows for having multiple hostnames on a single physical host - Host and domain name (as other things like IP can change)
  - MNT namespace—isolate a set of mount points such that processes in different namespaces cannot view each others files. (almost like chroot) (Filesystem access & structure)
  - IPC namespace—provides isolation to container process communication over shared memory and having an ability to invite other processes to read from the shared memory

https://www.redhat.com/sysadmin/7-linux-namespaces

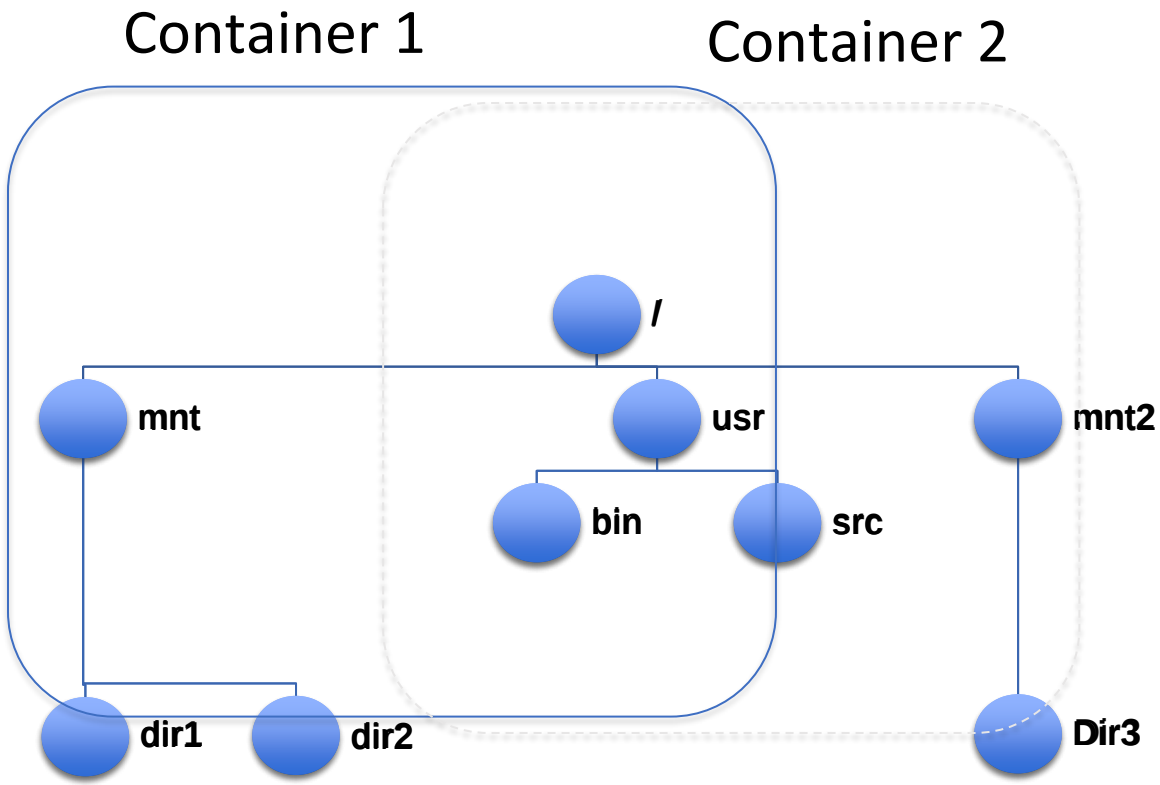
Docker used Namespaces (Cont.)

- NET namespace—allow processes inside each namespace instance to have access to a new IP address along with the full range of ports - Network access and structure
- USR namespace—provides user name-UID mapping isolating changes in names from the metadata within a container
- chroot syscall—Controls the location of the filesystem root
- cgroups—Controlling and accounting resources –rather than a hierarchical cgroup creation, there is a new cgroup with a root directory created to isolate resources and not allow navigation
- CAP drop—Operating system feature restrictions
- Security modules—Mandatory access controls

https://www.redhat.com/sysadmin/7-linux-namespaces

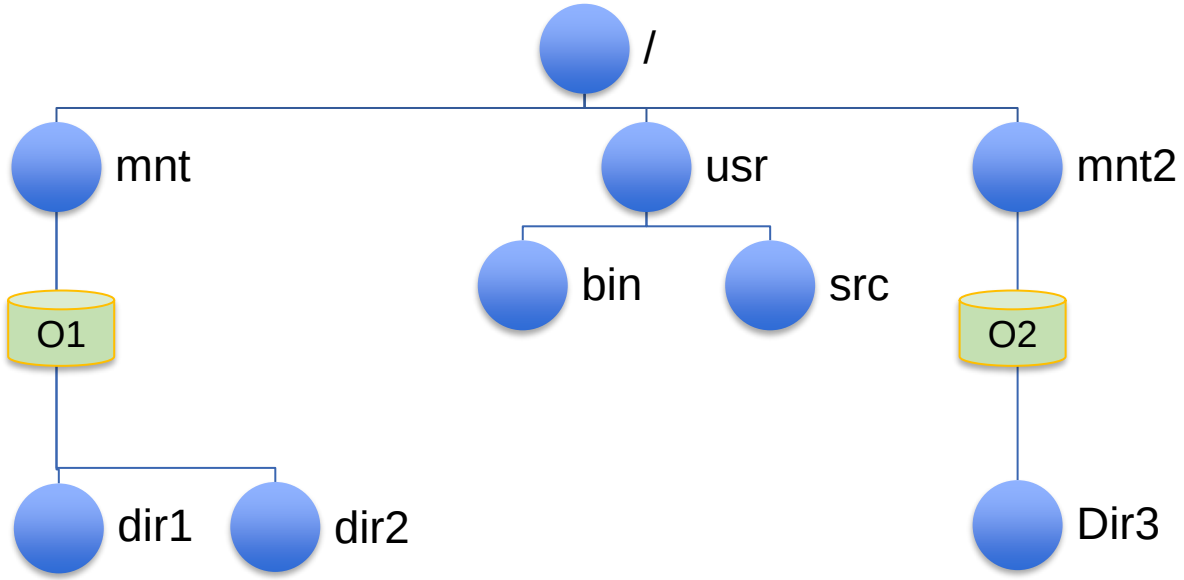
Access Control of objects : Illustration of name space (in this case MNT)

- Processes
  - in container 1 can access
    - Shared files in /usr
    - Non-shared /mnt
    - Cannot access /mnt2
  - in container 2 can access
    - Shared files in /usr
    - Non-shared /mnt2
    - Cannot access /mnt
- These are two different namespaces



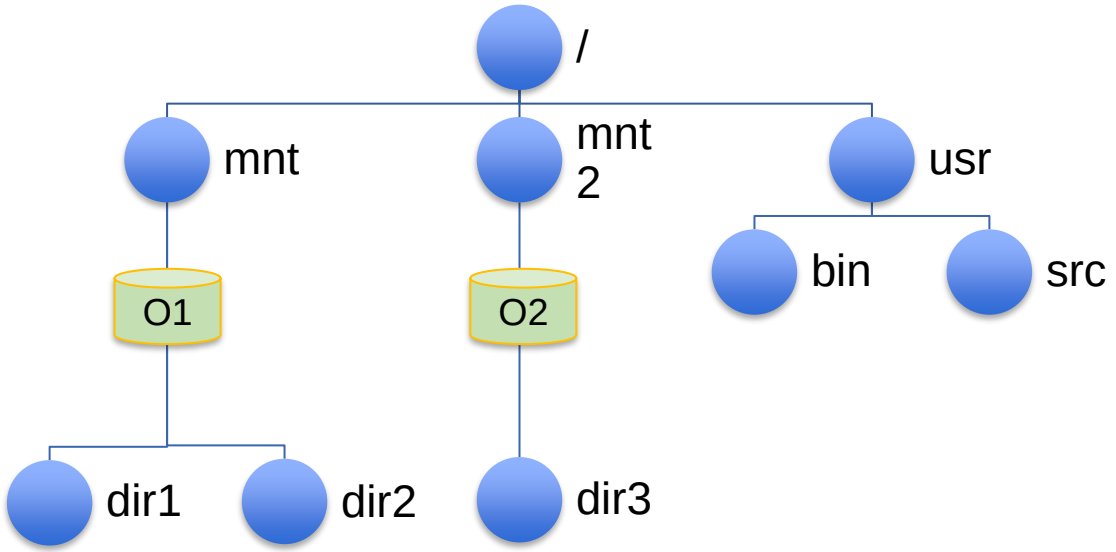
Linux Filesystems before Namespaces

- */* is the *root file system*
  - Contains *vmunix* – the OS
  - *usr, bin, src* are all subdirectories
- *O1, O2* are volumes containing different versions of an application (e.g., Oracle)
- Mounted at */mnt* and */mnt2*
- This namespace is visible to all processes
- Access to files and directories controlled by access rights



Linux Filesystems after Namespaces

- File namespace: *mount namespace*
- *O1, O2* are volumes containing different versions of an application (e.g., Oracle)
  - Mounted at */mnt* and */mnt2*
- This namespace is visible to all processes
- Access to files and directories controlled by access rights
- Which namespace is process in?
  - Look at */proc/pid/ns*
  - *ls –l* for */proc/pidx/mnt* may show as below (4026531840 is the namespace id)
    - lrwxrwxrwx. 1 mtk mtk 0 Jan 8 04:12 mnt -> mnt:[4026531840]



```
ubuntu@ubuntu2004:/proc/825/ns$ pwd
/proc/825/ns
ubuntu@ubuntu2004:/proc/825/ns$ sudo ls -l
total 0
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 net -> 'net:[4026531992]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 time -> 'time:[4026531834]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 user -> 'user:[4026531837]'
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 uts -> 'uts:[4026531838]'
ubuntu@ubuntu2004:/proc/825/ns$
```

```
ubuntu@ubuntu2004:/proc/825/ns$ sudo ls -l mnt
lrwxrwxrwx 1 postgres postgres 0 Feb 16 05:46 mnt -> 'mnt:[4026531840]'
ubuntu@ubuntu2004:/proc/825/ns$ S
```

Illustration of Namespace Operations : E.g. MNT Namespace

Creation of Namespace

- To create a namespace, must create a process with that namespace
- *pid = clone(childFunc, stackTop, CLONE_NEWNS | SIGCHLD, argv[1]);*
- Creates a new child, like *fork()*
- *NEWNS* flag indicates that child has a new mount namespace
- Child can do *mount* and *umount* to modify namespace

Programs to be joining into a namespace

Allows program to join an existing namespace

int setns

- (int fd, // namespace to join
- int nstype); // type of ns

int unshare (int flags); // which namespace

- CLONE_NEWNS specifies mount namespace
- Similar to clone(); allows caller to create a new namespace

NAME

setns - reassociate thread with a namespace

SYNOPSIS

```
#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <sched.h>

int setns(int fd, int nstype);
```

DESCRIPTION

Given a file descriptor referring to a namespace, reassociate the calling thread with that namespace.

The fd argument is a file descriptor referring to one of the namespace entries in a /proc/[pid]/ns/ directory; see **namespaces**(7) for further information on /proc/[pid]/ns/. The calling thread will be reassociated with the corresponding name-space, subject to any constraints imposed by the nstype argument.

UNSHARE(1) User Commands UNSHARE(1)

NAME

unshare - run program with some namespaces unshared from parent

SYNOPSIS

unshare [options] [program [arguments]]

DESCRIPTION

Unshares the indicated namespaces from the parent process and then executes the specified program. If program is not given, then ``${SHELL}'' is run (default: /bin/sh).

The namespaces can optionally be made persistent by bind mounting /proc/pid/ns/type files to a filesystem path and entered with **nsenter**(1) even after the program terminates (except PID namespaces where permanently running init process is required). Once a persistent namespace is no longer needed, it can be unpersisted with **umount**(8). See the **EXAMPLES** section for more details.

The namespaces to be unshared are indicated via options. Unshareable namespaces are:

mount namespace

Manual page unshare(1) line 1 (press h for help or q to quit)

```
ubuntu@ubuntu2004:~$ docker run -d --name namespaceA busybox:1.29 /bin/sh -c "sleep 30000"
Unable to find image 'busybox:1.29' locally
1.29: Pulling from library/busybox
b4a6e23922dd: Pull complete
Digest: sha256:8ccbac733d19c0dd4d70b4f0c1e12245b5fa3ad24758a11035ee505c629c0796
Status: Downloaded newer image for busybox:1.29
04904c5e958acf234a056b85942a2eaa7c41217989a927eec57c77f6dbe2aa40
ubuntu@ubuntu2004:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          POR
TS     NAMES
04904c5e958a   busybox:1.29   "/bin/sh -c 'sleep 3…"   16 seconds ago   Up 15 seconds
       namespaceA
```

```
ubuntu@ubuntu2004:~$ docker exec namespaceB ps
Error response from daemon: container 50f9267eb00ba22f481be12ae1a42000fdbc77565db3a3d1832f6
313b628c247 is not running
ubuntu@ubuntu2004:~$ docker exec namespaceA ps
PID   USER     TIME  COMMAND
    1 root      0:00 sleep 30000
    7 root      0:00 ps
ubuntu@ubuntu2004:~$
```

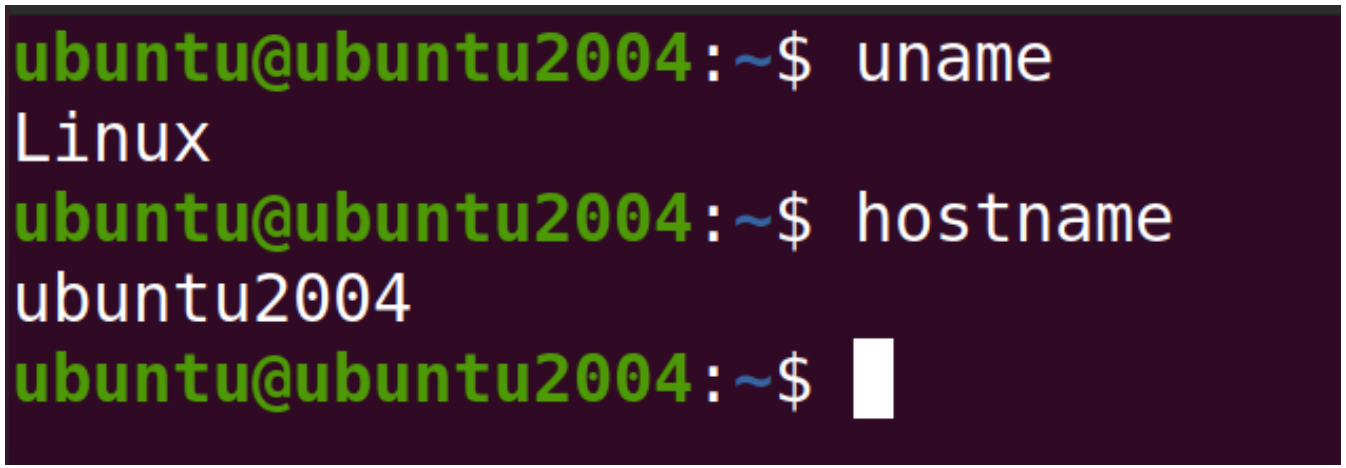
```
ubuntu@ubuntu2004:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          POR
TS     NAMES
04904c5e958a   busybox:1.29   "/bin/sh -c 'sleep 3…"   15 minutes ago   Up 15 minutes
       namespaceA
ubuntu@ubuntu2004:~$ docker exec namespaceA echo hello
hello
ubuntu@ubuntu2004:~$
```

CLOUD COMPUTING

UTS (Unix Time Sharing) Namespace

- Per namespace
  - Hostname
  - NIS domain name (Network Information Service)
- Reported by commands such as hostname
- Processes in namespace can change UTS values – only reflected in the child namespace
- Allows containers to have their own FQDN (Fully qualified Domain Name)

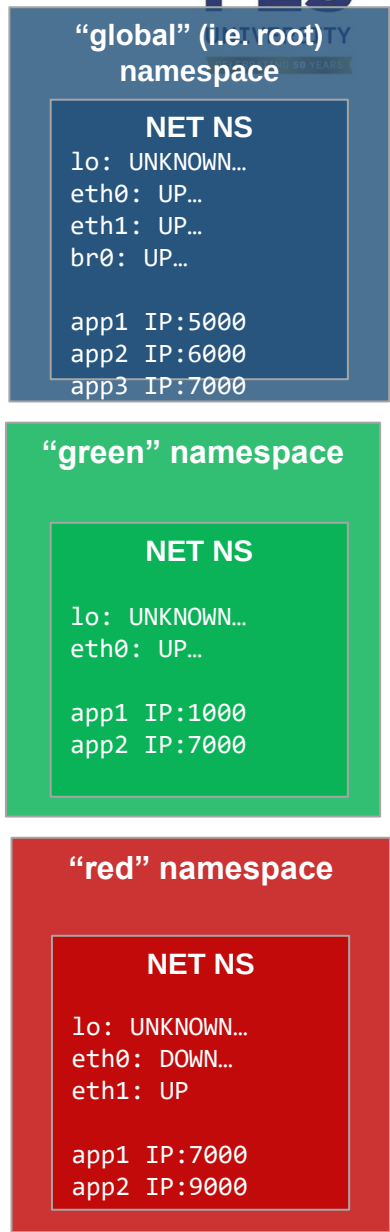
UTS namespace is about isolating hostnames.
The **UTS namespace** is used to isolate two specific elements of the system that relate to the uname system call.
The **UTS**(UNIX Time Sharing) **namespace** is named after the data structure used to store information returned by the uname system call.


ubuntu@ubuntu2004:~$ uname
Linux
ubuntu@ubuntu2004:~$ hostname
ubuntu2004
ubuntu@ubuntu2004:~$

CLOUD COMPUTING

NET namespace

- It’s a virtual network barrier encapsulating a process to isolate its network connectivity (in/out) and resources (i.e. network interfaces, route tables and rules) from linux core and other processes.
- It allow processes inside each namespace instance to have access to per namespace network objects
  - Network devices (ethernets)
  - Bridges
  - Routing tables
  - IP addresses
  - ports
- Various commands support network namespace such as ip



IP(8) Linux IP(8)

NAME

ip - show / manipulate routing, network devices, interfaces and tunnels

SYNOPSIS

ip [OPTIONS] OBJECT { COMMAND | **help** }

ip [**-force**] **-batch** filename

OBJECT := { **link** | **address** | **addrlabel** | **route** | **rule** | **neigh** | **ntable** | **tunnel** | **tuntap** | **maddress** | **mroute** | **mrule** | **monitor** | **xfrm** | **netns** | **l2tp** | **tcp_metrics** | **token** | **macsec** }

OPTIONS := { **-V**[ersion] | **-h**[uman-readable] | **-s**[tatistics] | **-d**[etails] | **-r**[esolve] | **-iec** | **-f**[amily] { **i**

-l[oops] { **maximum-add**

-t[imestamp] | **-ts**[hor

-br[ief] | **-j**[son] | **-**

EXAMPLES

ip addr
    Shows addresses assigned to all network interfaces.

ip neigh
    Shows the current neighbour table in kernel.

ip link set x up
    Bring up interface x.

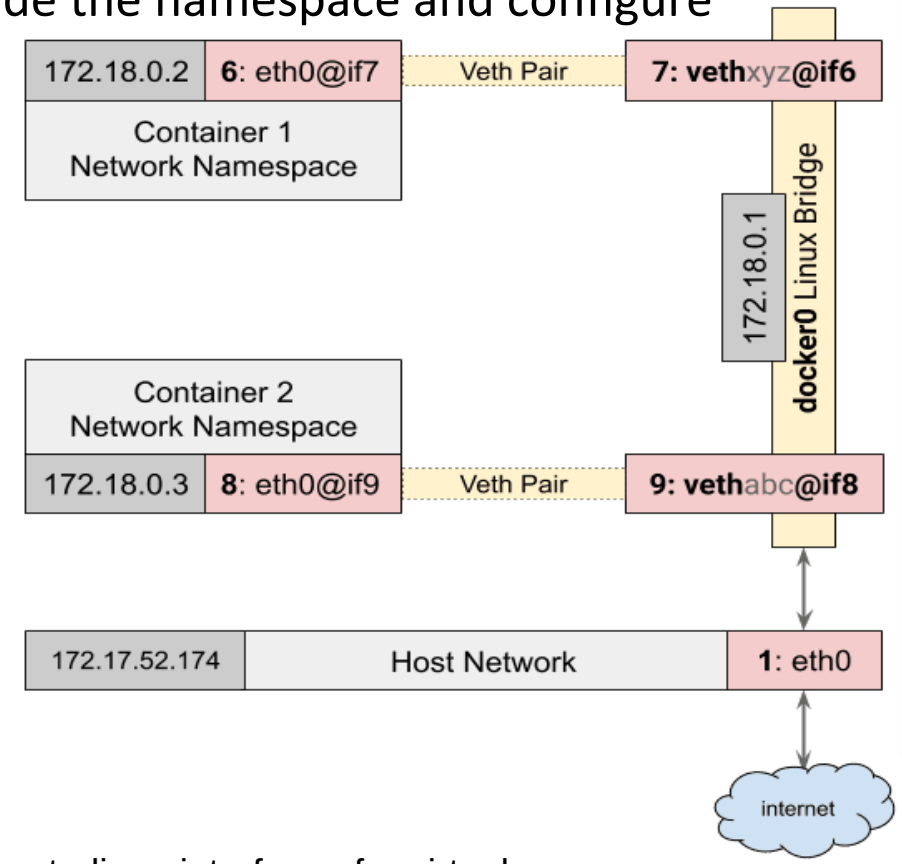
ip link set x down
    Bring down interface x.

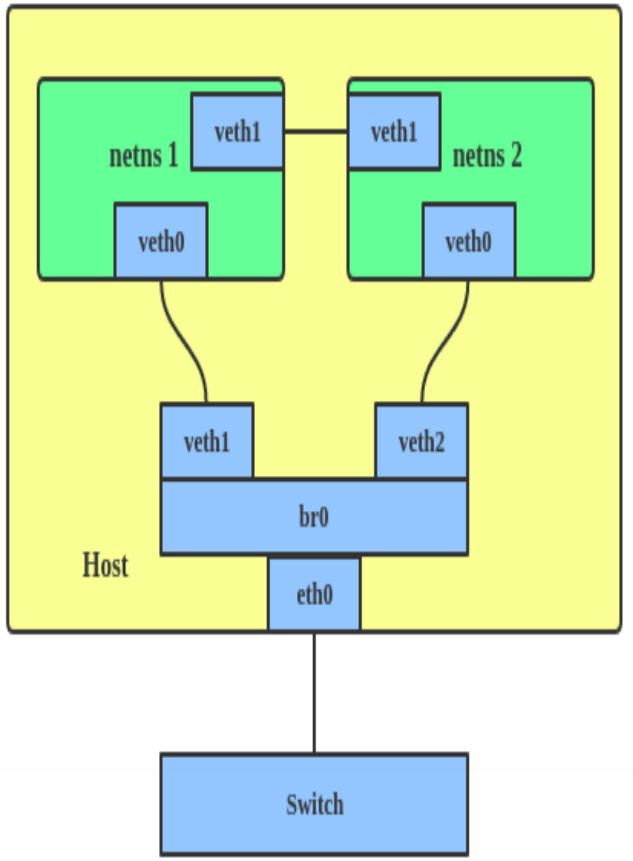
ip route
    Show table routes.

```
ubuntu@ubuntu2004:~$ ip neigh
10.0.2.2 dev enp0s3 lladdr 52:54:00:12:35:02 REACHABLE
ubuntu@ubuntu2004:~$ ip route
default via 10.0.2.2 dev enp0s3 proto dhcp metric 100
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.15 metric 100
169.254.0.0/16 dev enp0s3 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
ubuntu@ubuntu2004:~$ S
```

NET

- A VETH (virtual Ethernet) configuration as shown can be used when namespaces need to communicate to the main host namespace or between each other.
- veths – create veth pair, move one inside the namespace and configure
- Acts as a pipe between the 2 namespaces
- The VETH (virtual Ethernet) device is a local Ethernet tunnel.
- Devices are created in pairs
- Packets transmitted on one device in the pair are immediately received on the other device.
- When either device is down, the link state of the pair is down.

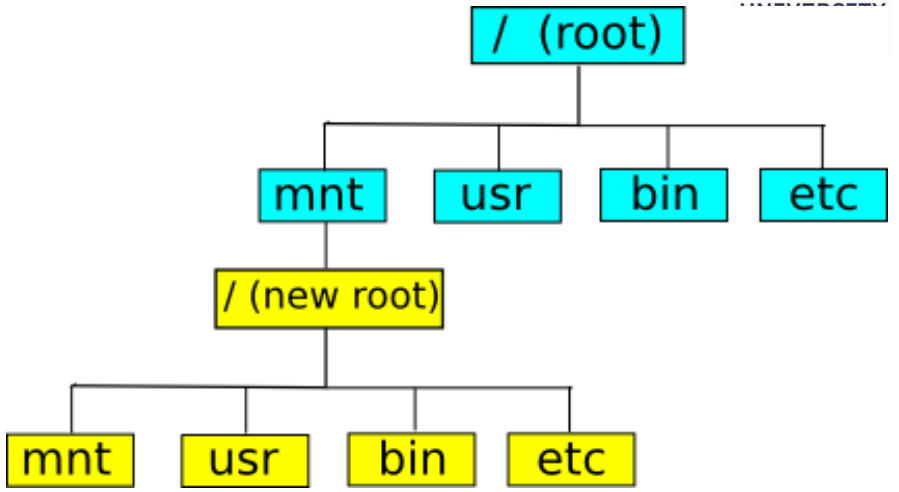




https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking

Filesystem root - chroot name space

- The OS has a file system started at root
  - /bin, and so on, contain programs and libraries
- If you want to run a program which uses a different version of /bin and so on
- This can be achieved using chroot
  - Mount new /bin on some place (say /mnt)
  - Issue chroot /mnt *command*
  - *command* will then see the root as /mnt, not the real root
- Using this technique, can give each process on system its own file system
- Changes the root directory for currently running processes as well as it children for
  - Search paths
  - Relative directories
- Using chroot can be escaped given proper capabilities, thus pivot_root is often used instead
  - chroot; points the processes file system root to new directory
  - pivot_root; detaches the new root and attaches it to process root directory
  - pivot_root info at https://man7.org/linux/man-pages/man8/pivot_root.8.html



Often used when building system images

- chroot to temp directory
- Download and install packages in chroot
- Compress chroot as a system root FS

```
CHROOT(8)                          User Commands                          CHROOT(8)

NAME
       chroot - run command or interactive shell with special root directory

SYNOPSIS
       chroot [OPTION] NEWROOT [COMMAND [ARG]...]
       chroot OPTION

DESCRIPTION
       Run COMMAND with root directory set to NEWROOT.

       --groups=G_LIST
              specify supplementary groups as g1,g2,..,gN

       --userspec=USER:GROUP
              specify user and group (ID or name) to use

       --skip-chdir
              do not change working directory to '/'
```

```
ubuntu@ubuntu2004:~$ which chroot
/usr/sbin/chroot
ubuntu@ubuntu2004:~$
```

Cgroups (or Control groups)

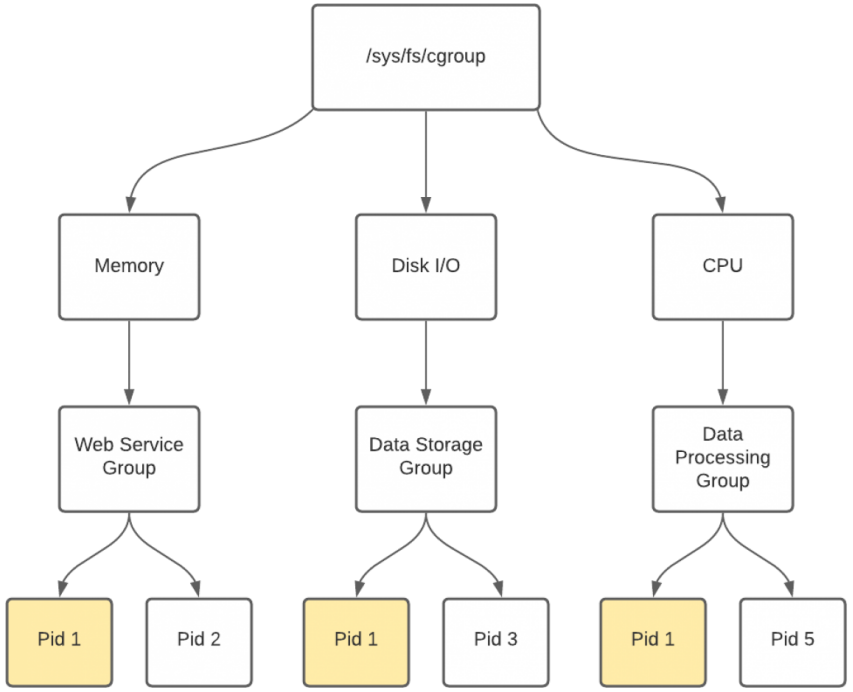
- Cgroup is a linux feature to limit, police, and account the resource usage for a set of processes. Docker uses cgroups to limit the system resources.
- Cgroups - provides mechanisms (fine grain control) to allocate, monitor and limit resources such as ***CPU time, system memory, block IO or disk bandwidth, network bandwidth, or combinations of these resources*** — among user-defined groups of tasks (processes) running on a system.
- Cgroups works on resource types. So it works by dividing resources into groups and then assigning tasks to those groups, deny access to certain resources, and even reconfigure our cgroups dynamically on a running system.
- When you install Docker binary on a linux box like ubuntu it will install cgroup related packages and create subsystem directories.
- Hardware resources can be appropriately divided up among tasks and users, increasing overall efficiency.

Linux cgroups - Functionality

- Access
  - which devices can be used per cgroup
- Resource limiting
  - memory, CPU, device accessibility, block I/O, etc.
- Prioritization
  - who gets more of the CPU, memory, etc.
- Accounting
  - resource usage per cgroup
- Control
  - freezing & check pointing
- Injection
  - packet tagging

Linux cgroups - usage

- cgroups are hierarchically structured where each of the groups are created for a resource with a number.
- Tasks are assigned to cgroups
- Each cgroups has a resource limitation
- There is a hierarchy for each resource

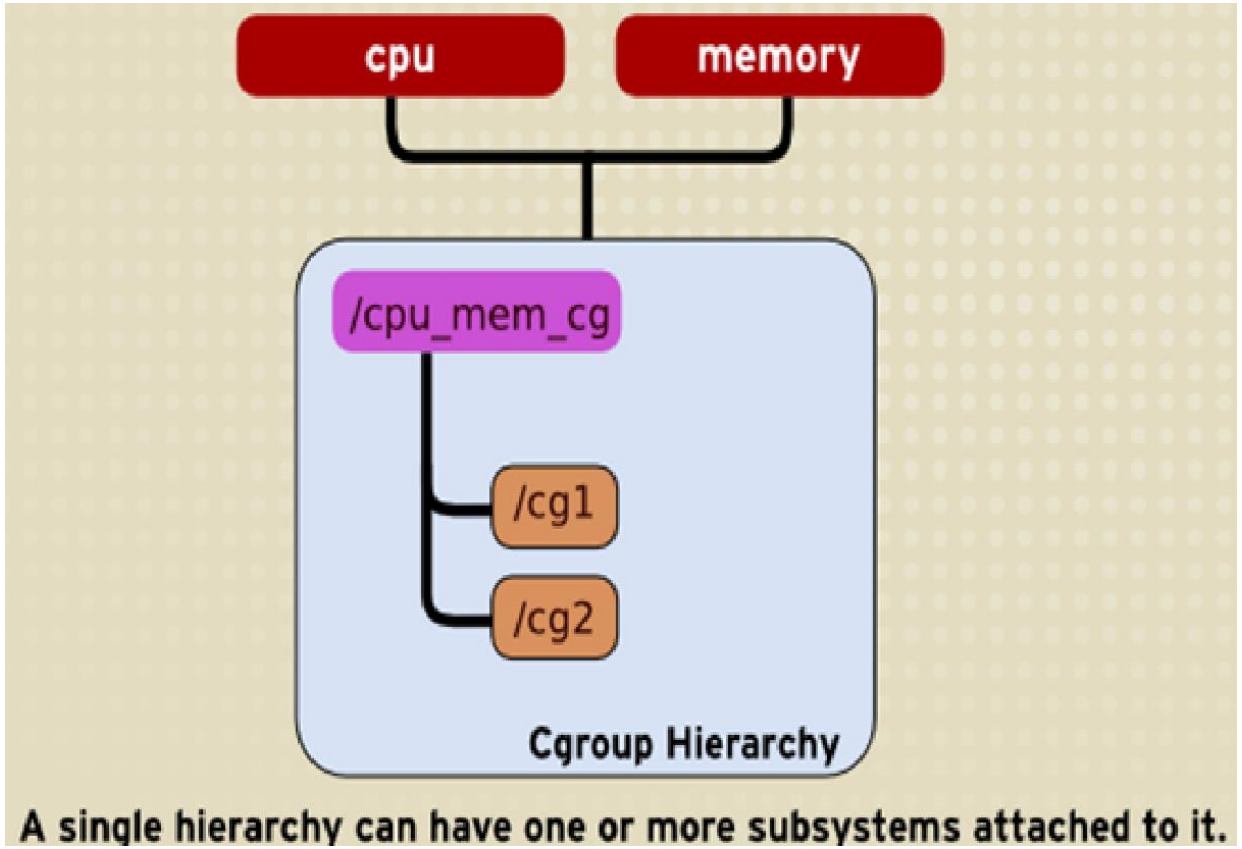


What resources can we limit?

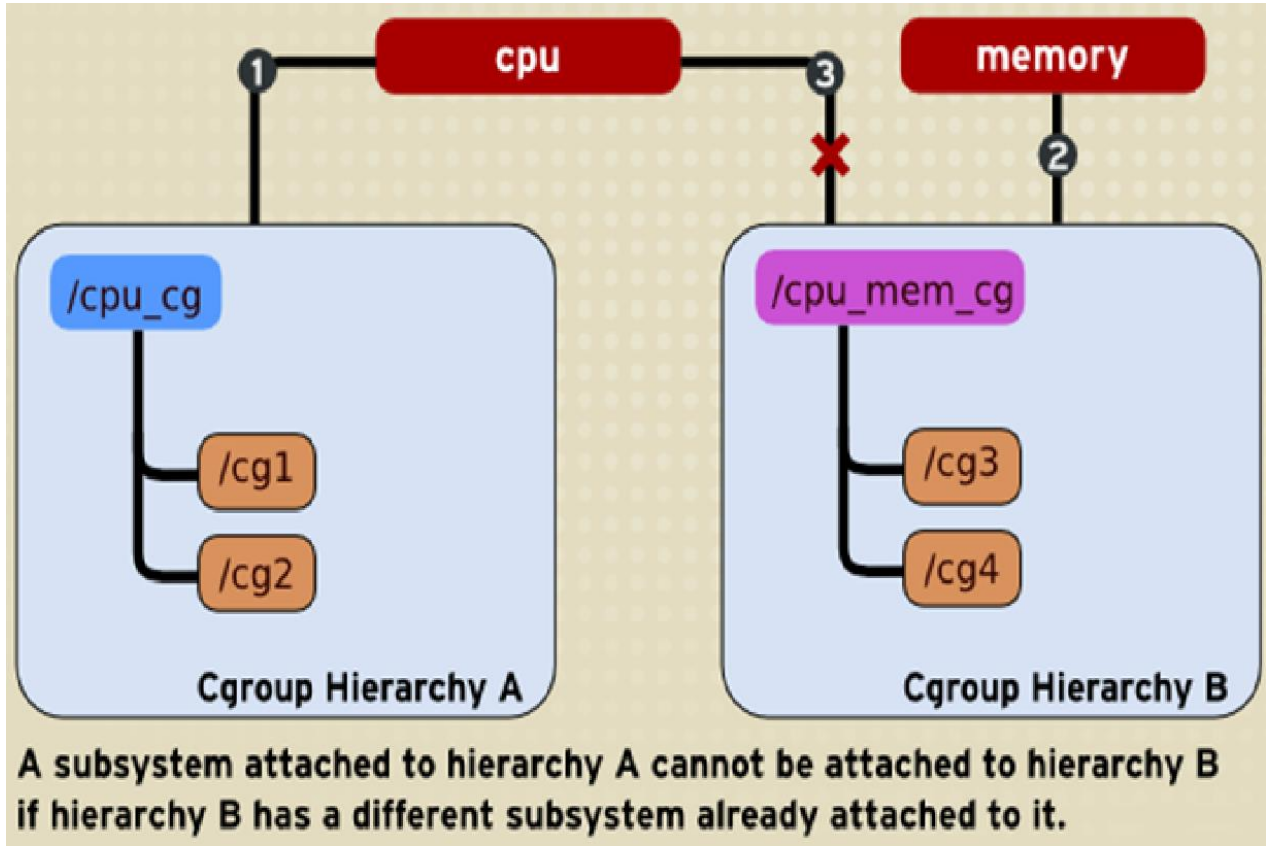
All of the following can be limited for a Cgroup

- Memory
- CPU
- Block IO
- Devices (which of the devices and allowing creation of devices ..)
- Network

cgroup Examples



A single hierarchy can have one or more subsystems attached to it.

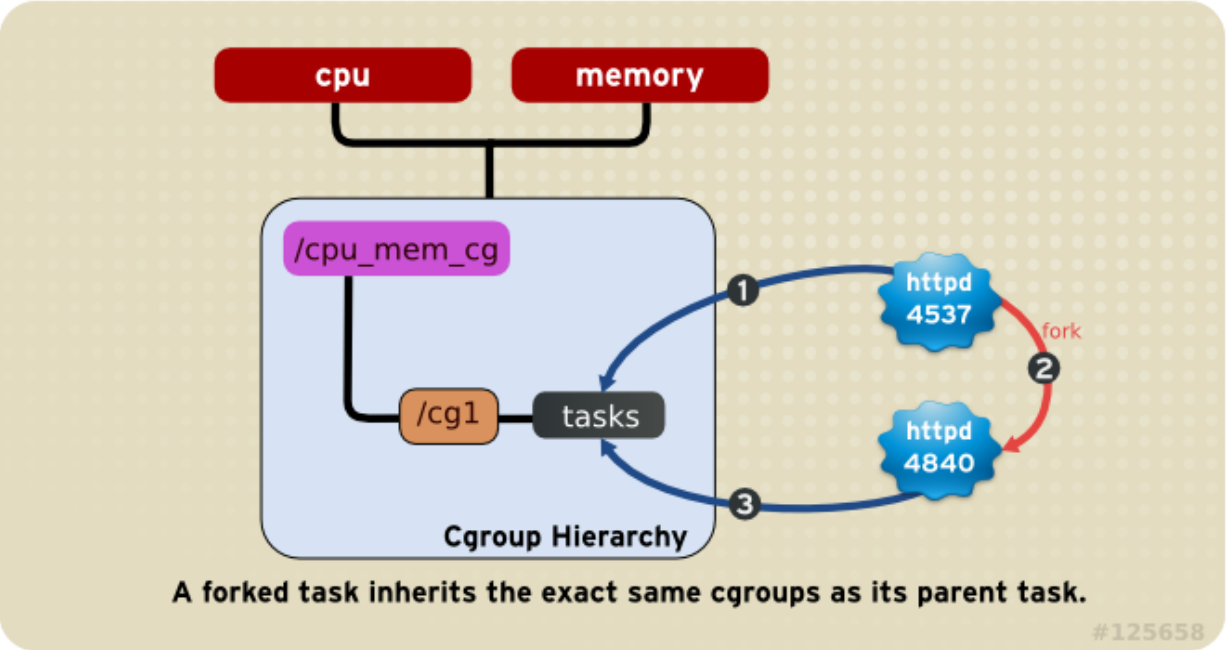


A subsystem attached to hierarchy A cannot be attached to hierarchy B if hierarchy B has a different subsystem already attached to it.

Each cgroup has a resource limitation associated with it

What resources can we limit?

- When a process creates a child process, the child process stays in the same cgroup
  - Good for servers such as NFS
  - Typical operation
    - Receive request
    - Fork child to process request
    - Child terminates when request complete
    - All children will have resource limitation of parent => the resource limitation of parent will apply to processing requests



Try the following on your docker command line:

docker run -d --name mycontainer --cpus="0.5" ubuntu sleep 1000
docker run -d --name mycontainer --memory=100m ubuntu sleep 1000
docker exec mycontainer cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us
docker exec mycontainer cat /sys/fs/cgroup/memory/memory.limit_in_bytes
docker stats mycontainer
docker rm -f mycontainer

Example With stress test:

docker run -d --name mem_test --memory=100m ubuntu sleep 1000
docker exec -it mem_test apt update && apt install -y stress
docker exec -it mem_test stress --vm 1 --vm-bytes 150m --timeout 30s
(The container **should be killed** by the **Out of Memory (OOM) killer**.)

docker logs mem_test
docker ps -a | grep mem_test
(It should show **Exited (137)**, meaning it was terminated due to OOM.)

Container Filesystem

- Programs running inside containers know nothing about image layers.
- From inside a container, the filesystem operates as though it's not running in a container or operating on an image.
- From the perspective of the container, it has exclusive copies of the files provided by the image. This is made possible with something called a *union filesystem (UFS).*
- Docker uses a variety of union filesystems and will select the best fit for your system.
- A union filesystem is part of a critical set of tools that combine to create effective filesystem isolation.
- The other tools are MNT namespaces and the chroot system call.

Union Filesystem

- Union is a type of a filesystem that can create an illusion of merging contents of several directories (created in layers) into one without modifying its original (physical) sources which can be shown in single, merged view
- Union file system operates by creating layers, making them very lightweight and fast.
- Docker Engine uses UnionFS to provide the building blocks for containers.
- Docker Engine can use multiple UnionFS variants, including AUFS, overlay2, btrfs, vfs, and DeviceMapper.

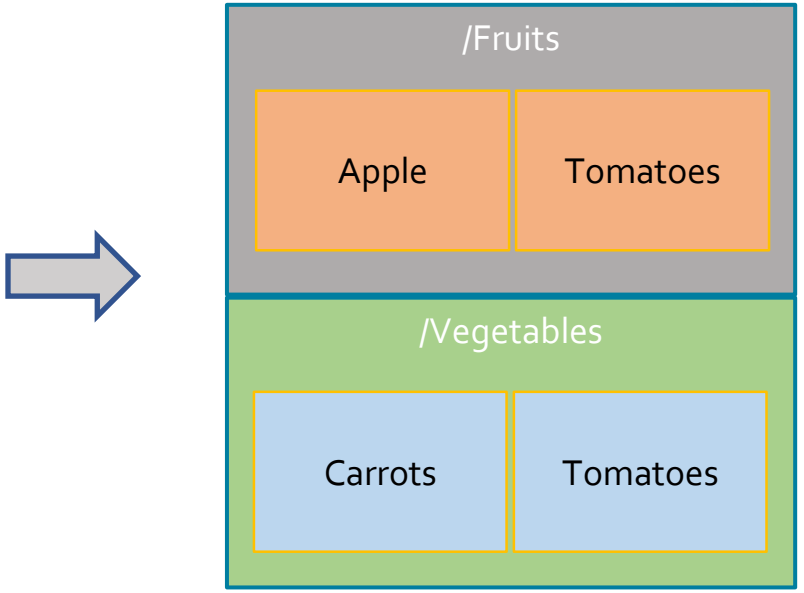
Reference:https://docs.docker.com/get-started/overview/

https://martinheinz.dev/blog/44

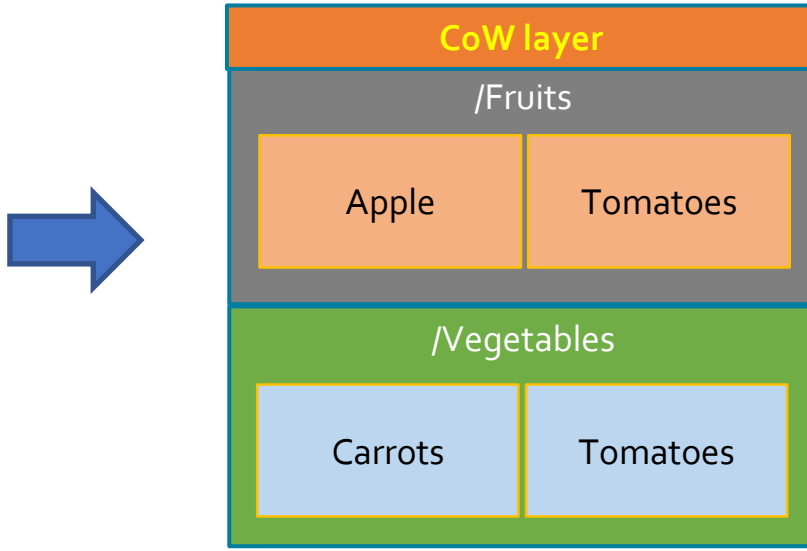
unionfs features - Layering

- unionfs permits layering of file systems
  - */Fruits* contains files *Apple, Tomato*
  - */Vegetables* contains *Carrots, Tomato*

  mount –t unionfs –o dirs=/Fruits:/Vegetables none /mnt/healthy
  - */mnt/healthy* has 3 files – *Apple, Tomato, Carrots*
  - *Tomato* comes from */Fruits* (1st in *dirs* option)
- As if /Fruits is layered on top of /Vegetables



- -o cow option on mount command enables copy on write
  - *If change is made to a file*
  - *Original file is not modified*
  - *New file is created in a hidden location*
- If /Fruits is mounted ro (read-only), then changes will be recorded in a temporary layer



http://www.linuxjournal.com/article/7714

The standard form of the **mount** command is:

mount -t type device dir

This tells the kernel to attach the filesystem found on device (which is of type type) at the directory dir. The option **-t** type is optional. The **mount** command is usually able to detect a filesystem. The root permissions are necessary to mount a filesystem by default. See section "Non-superuser mounts" below for more details. The previous contents (if any) and owner and mode of dir become invisible, and as long as this filesystem remains mounted, the pathname dir refers to the root of the filesystem on device.

If only the directory or the device is given, for example:

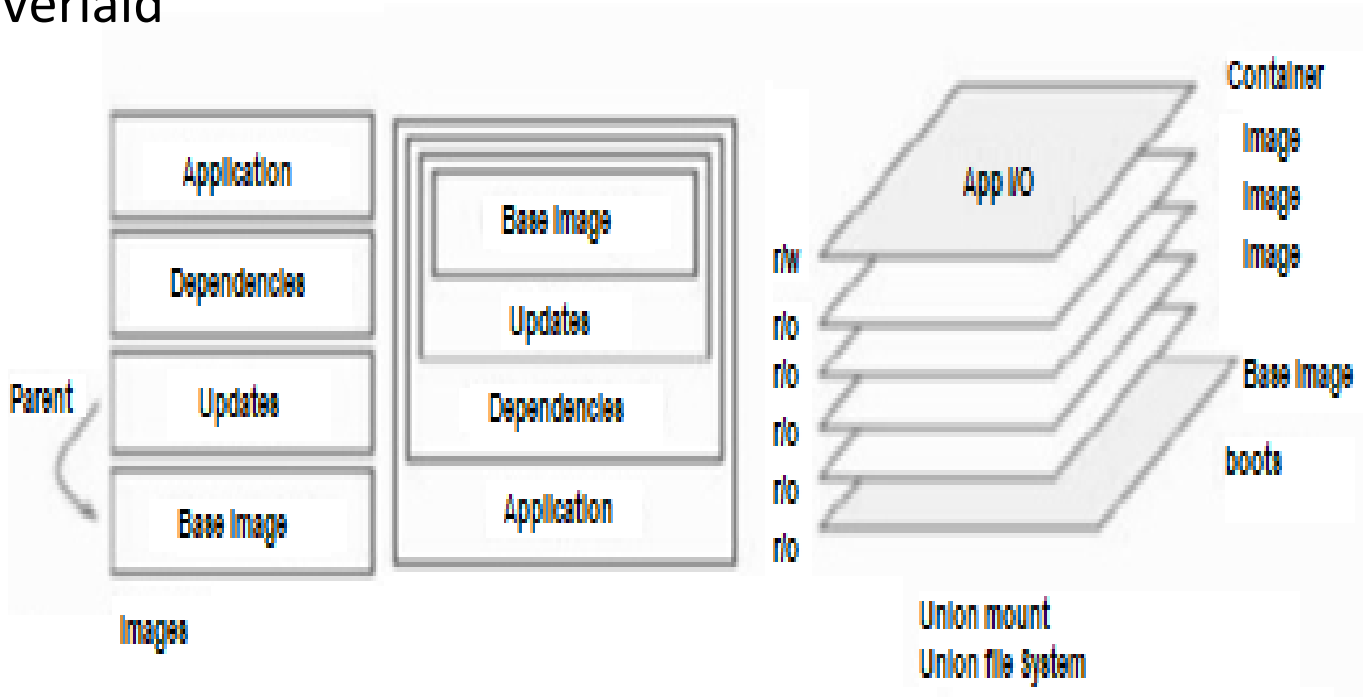
mount /dir

then **mount** looks for a mountpoint (and if not found then for a device) in the

al page mount(8) line 19 (press h for help or q to quit)

Incremental Images

- Union FS
  - Files from separate FS (branches) can be overlaid
  - Forming a single coherent FS
  - Branches may be read-only or read-write
- Docker Layers
  - Each layer is mounted on top of prior layers
  - First layer = base image (scratch, busybox, ubuntu, …)
  - A read-only layer = an image
  - The top read-write layer = container



https://www.slideshare.net/cm13/introduction-to-docker-48784005

Weaknesses of Union Filesystem

- Different filesystems have different rules about file attributes, sizes, names, and characters.
- Union filesystems are in a position where they often need to translate between the rules of different filesystems. In the best cases, they're able to provide acceptable translations. In the worst cases, features are omitted.
- Union filesystems use a pattern called *copy-on-write, and that makes implementing* memory-mapped files (the mmap system call) difficult.
- Most issues that arise with writing to the union filesystem can be addressed without changing the storage provider. These can be solved with volumes
- The union filesystem is not appropriate for working with long-lived data or sharing data between containers, or a container and the host.

Summary

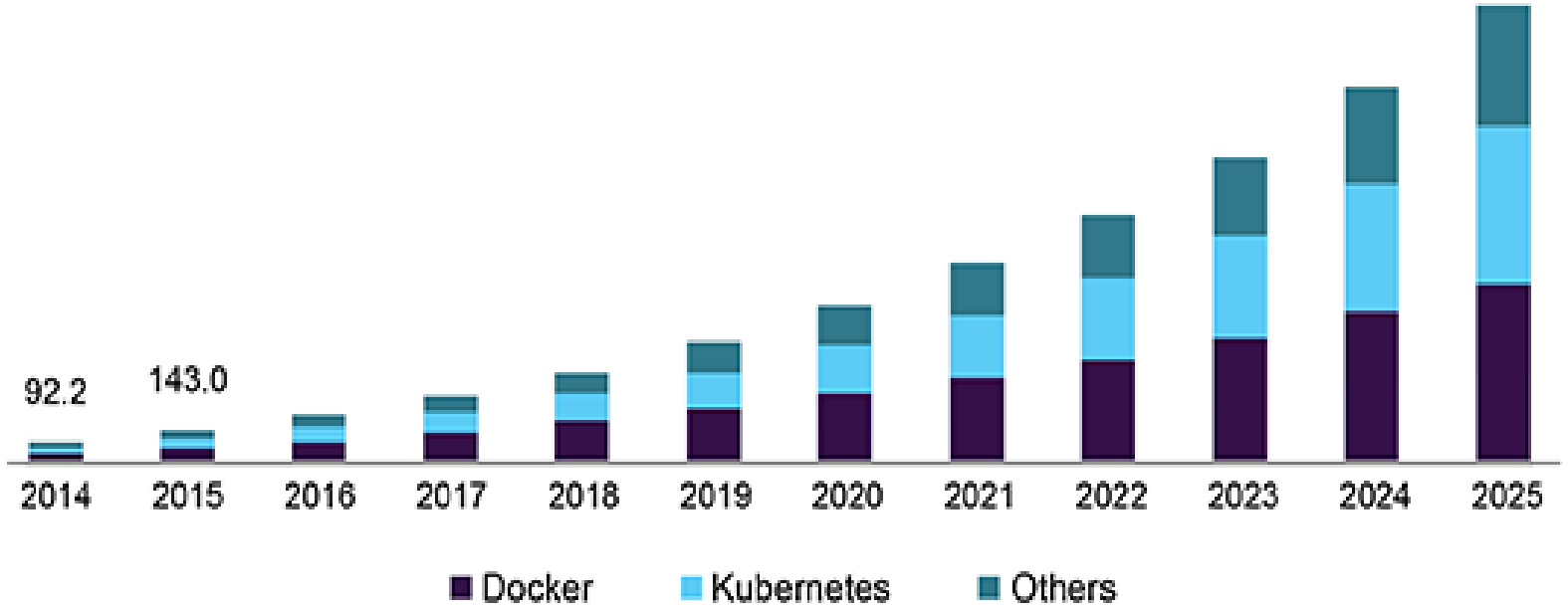
- Use namespaces for controlling resource access
  - Docker has developed their own namespace technology called **namespaces** to provide the isolated workspace called the container.
  - When you run a container, **Docker** creates a set of **namespaces** for that container.
  - These **namespaces** provide a layer of isolation.
- Use cgroups for resource sharing

CLOUD COMPUTING

Container Software

1. Docker
2. AWS Fargate
3. Google Kubernetes Engine
4. Amazon ECS
5. LXC
6. Microsoft Azure
7. Google Cloud Platform
8. Core OS

U.S. application container market size, by platform, 2014 - 2025 (USD Million)



Source: www.grandviewresearch.com

Additional References

Containers vs VMs
https://www.redhat.com/en/topics/containers/containers-vs-vms

Docker container
https://www.simplilearn.com/tutorials/docker-tutorial/what-is-docker-container
https://www.xda-developers.com/best-docker-containers-for-developers/

Understanding Linux containers
https://www.redhat.com/en/topics/containers
https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html

Additional Reading

- https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-Relationships_Between_Subsystems_Hierarchies_Control_Groups_and_Tasks.html
- https://en.wikipedia.org/wiki/Cgroups
- https://www.youtube.com/watch?v=sK5i-N34im8
  - Cgroups, namespaces and beyond: What are containers made from – Jerome Pettazzoni, Docker

  https://www.youtube.com/watch?v=nXV6qihj5uw
- https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-Relationships_Between_Subsystems_Hierarchies_Control_Groups_and_Tasks.html

THANK YOU

Prafullata Kiran Auradkar

Department of Computer Science and Engineering

prafullatak@pes.edu