

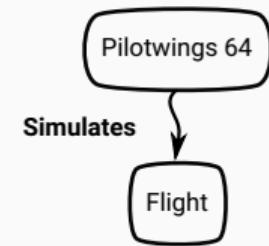
INTRODUCTION TO OS-LEVEL VIRTUALIZATION ON LINUX

Pedro Bruel

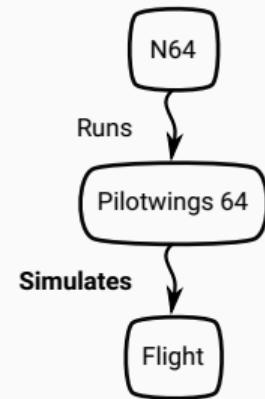
phrb@ime.usp.br

May 25th, 2020

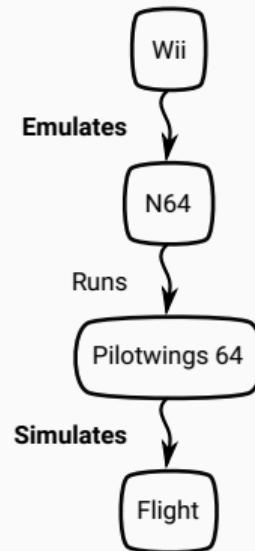
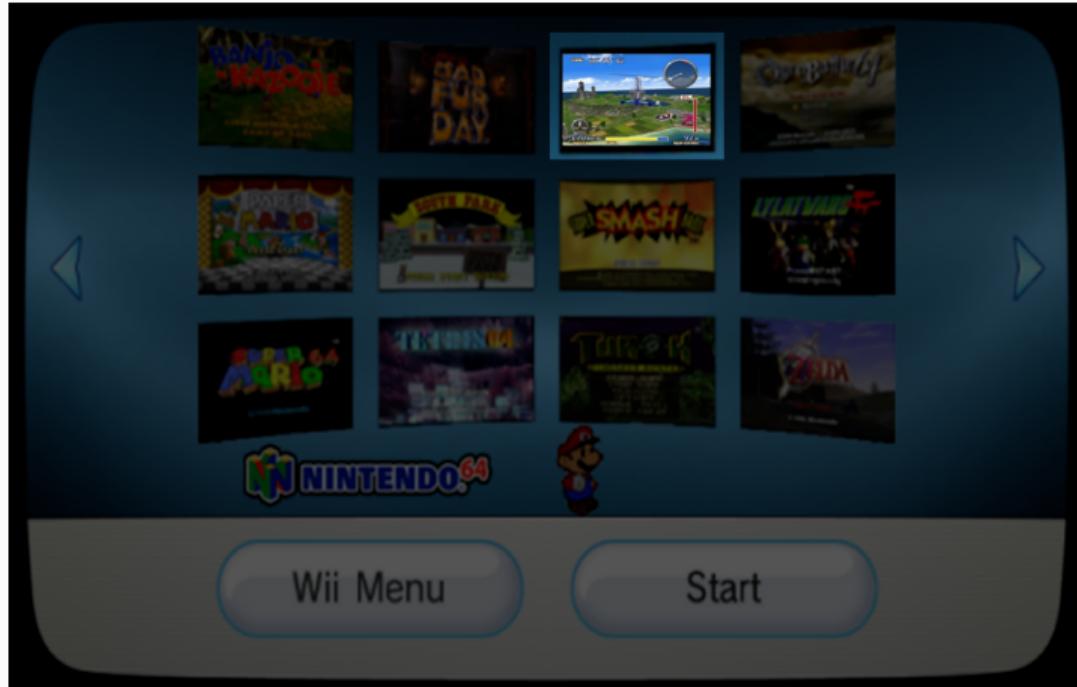
WHAT ARE SIMULATION, EMULATION, VIRTUALIZATION?



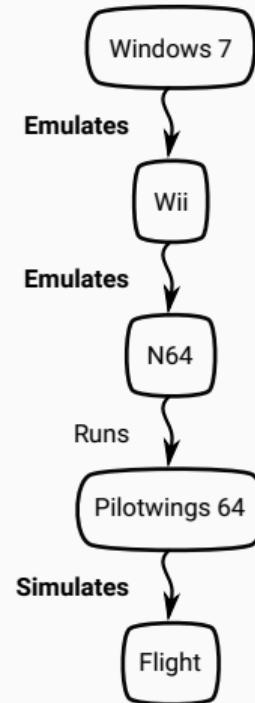
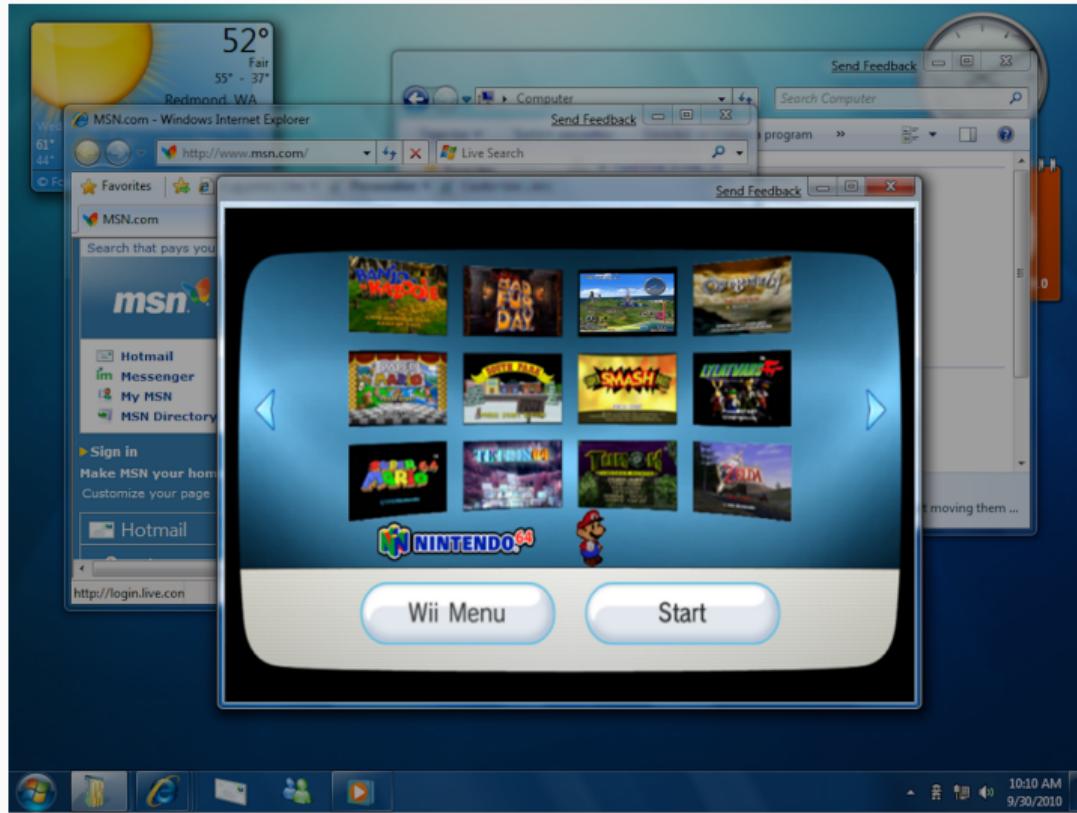
WHAT ARE SIMULATION, EMULATION, VIRTUALIZATION?



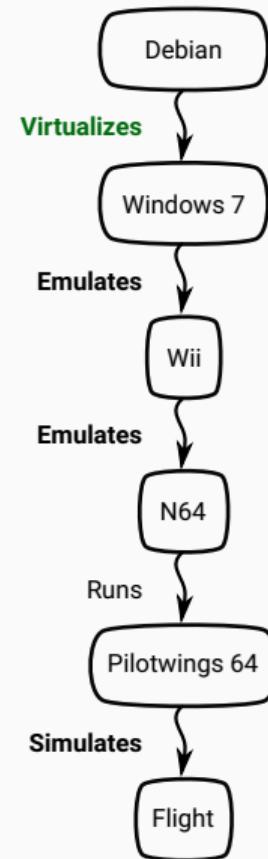
WHAT ARE SIMULATION, EMULATION, VIRTUALIZATION?



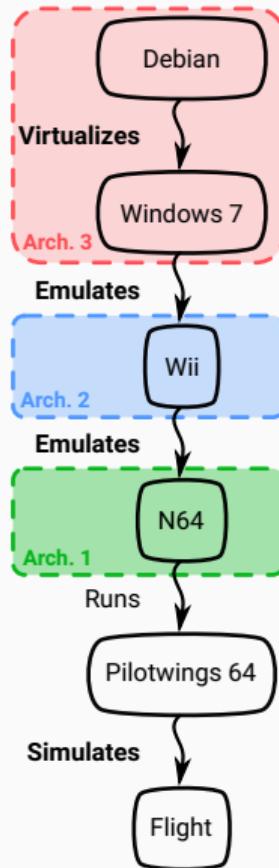
WHAT ARE SIMULATION, EMULATION, VIRTUALIZATION?



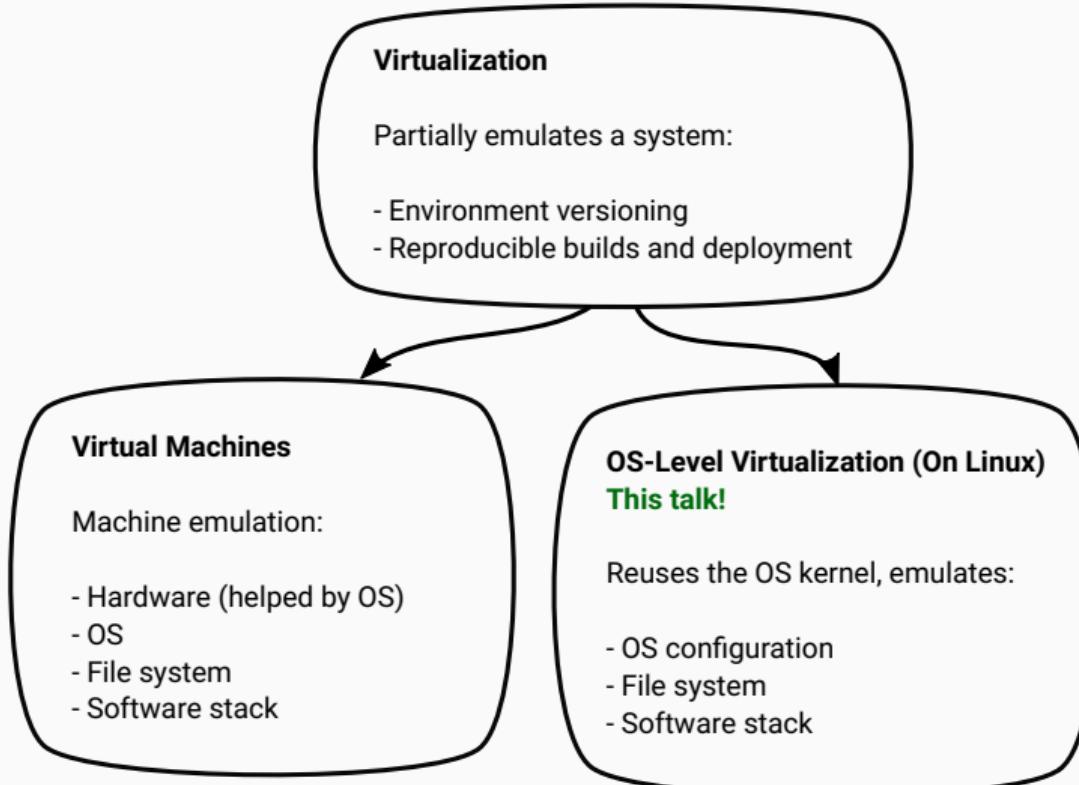
WHAT ARE SIMULATION, EMULATION, VIRTUALIZATION?



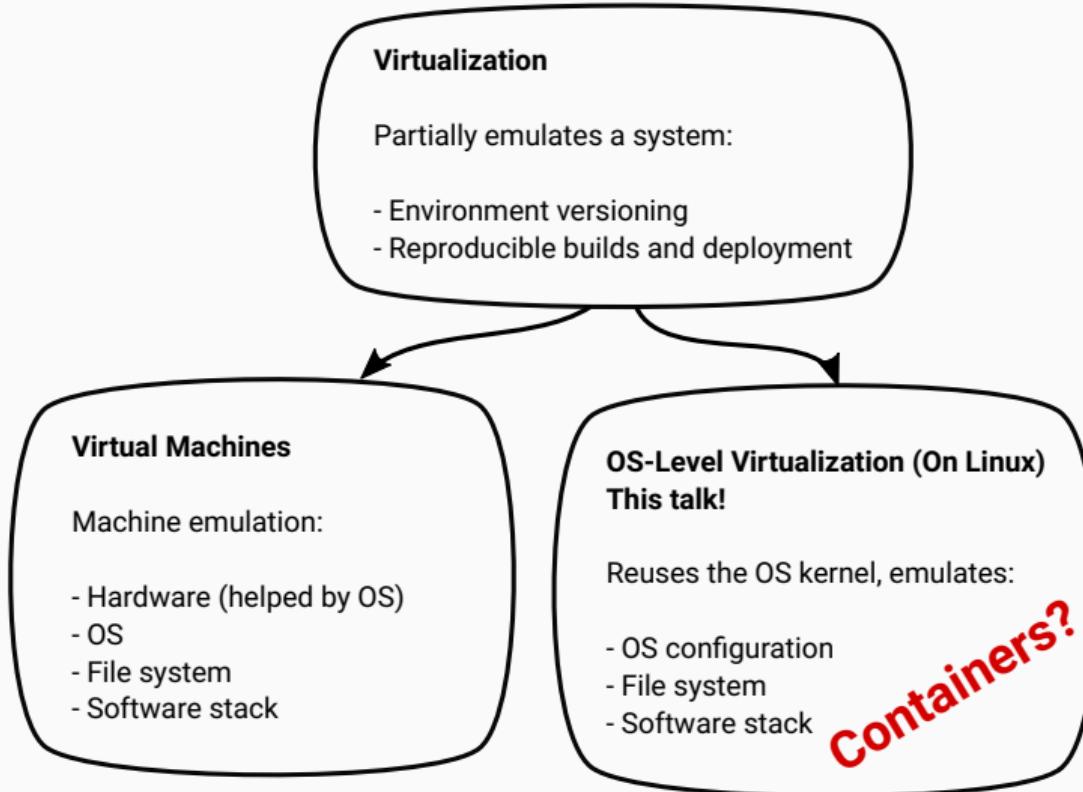
WHAT ARE SIMULATION, EMULATION, VIRTUALIZATION?



OS-LEVEL VIRTUALIZATION



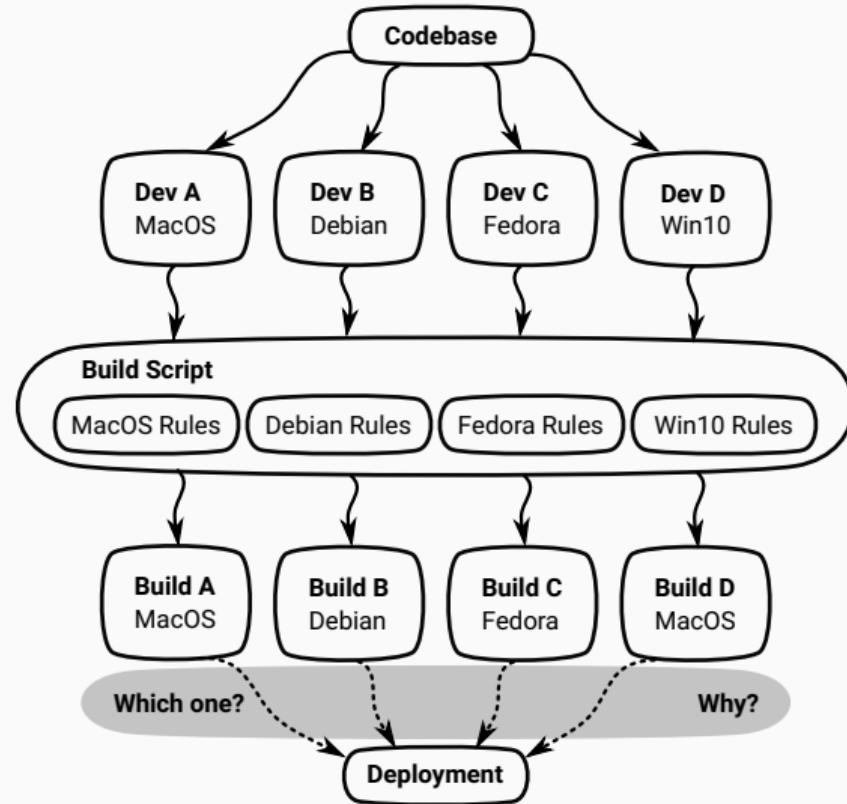
OS-LEVEL VIRTUALIZATION



OS-LEVEL VIRTUALIZATION: SCOPE OF THIS TALK

Scope of this Talk

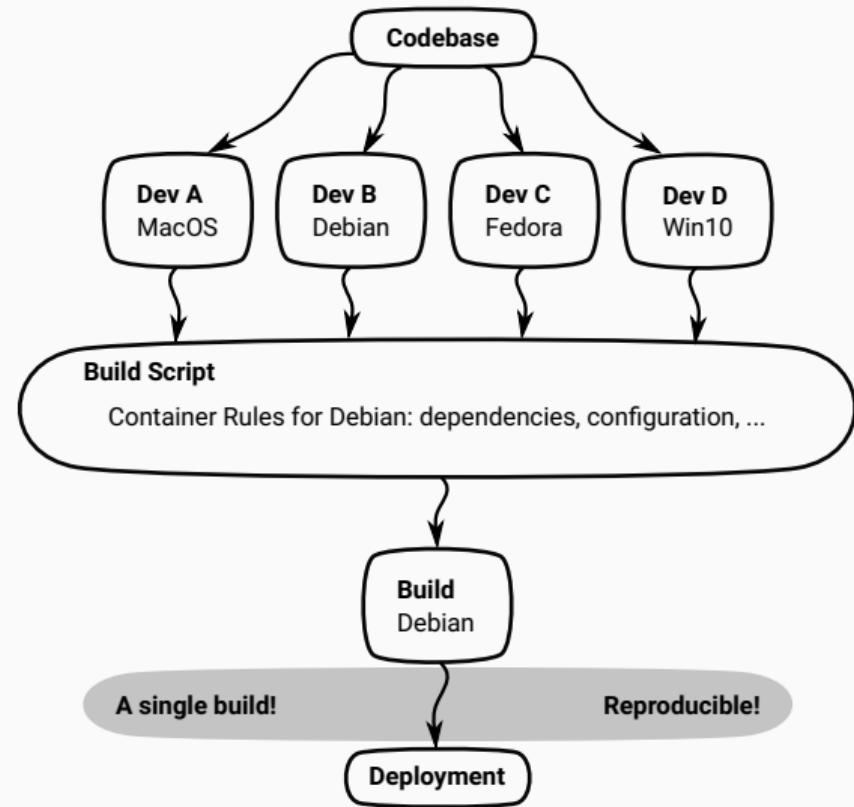
- Why should you **use containers?**
 - Environment versioning
 - Reproducible builds
 - It can be **easier**
- How do containers **work?**
- What **tools** are available?



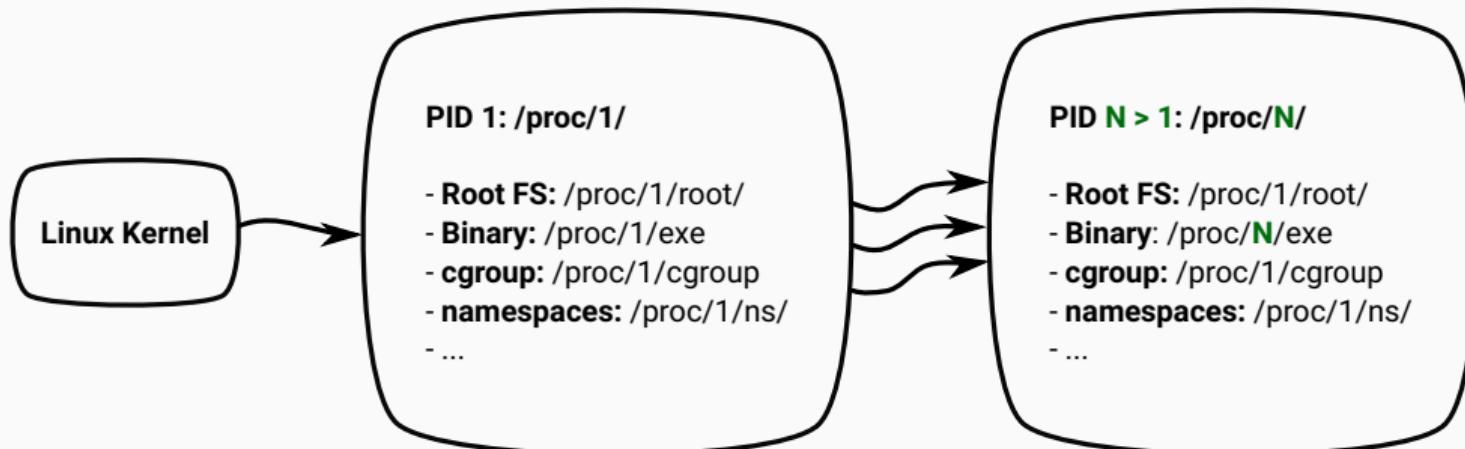
OS-LEVEL VIRTUALIZATION: SCOPE OF THIS TALK

Scope of this Talk

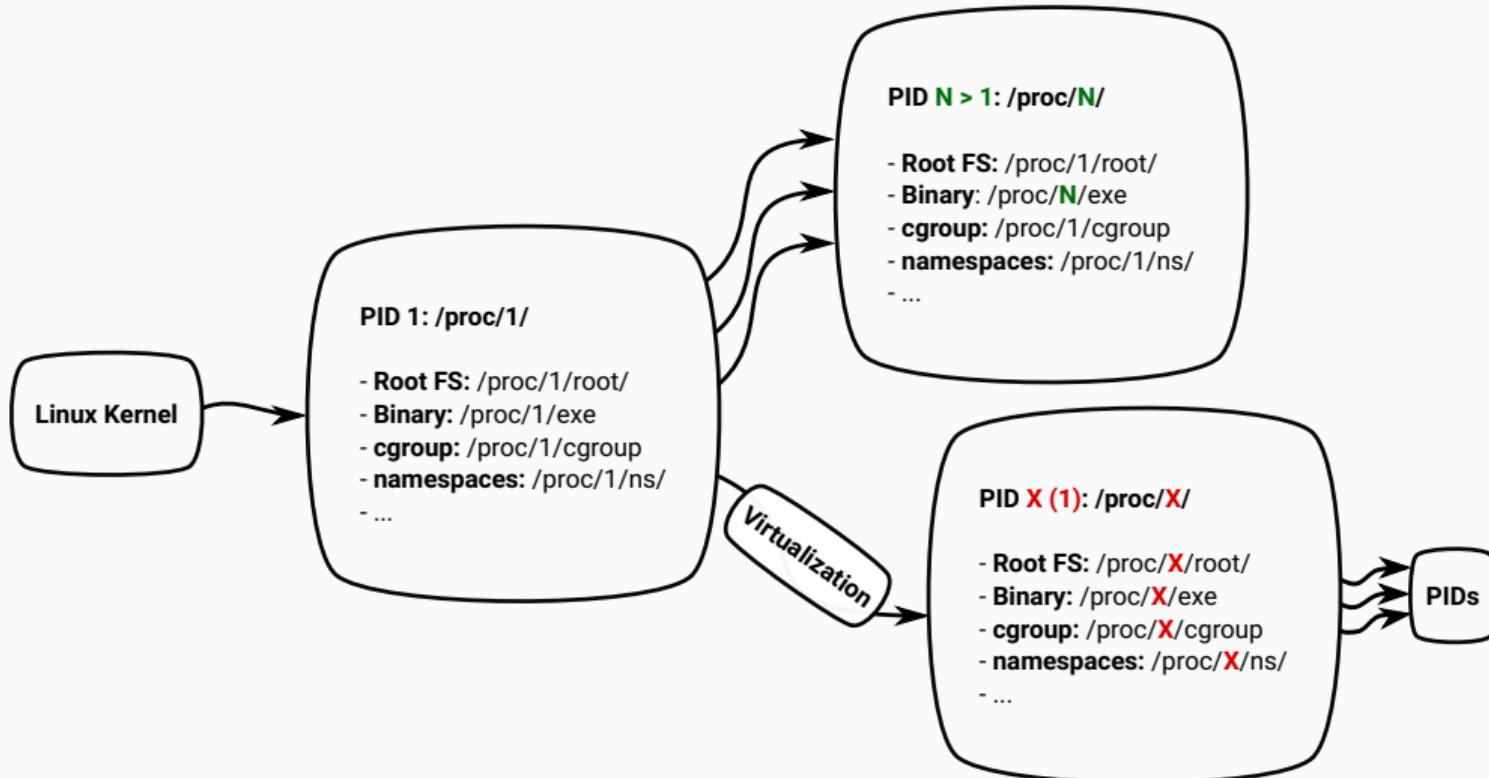
- Why should you **use containers?**
 - Environment versioning
 - Reproducible builds
 - It can be **easier**
- How do containers **work?**
- What **tools** are available?



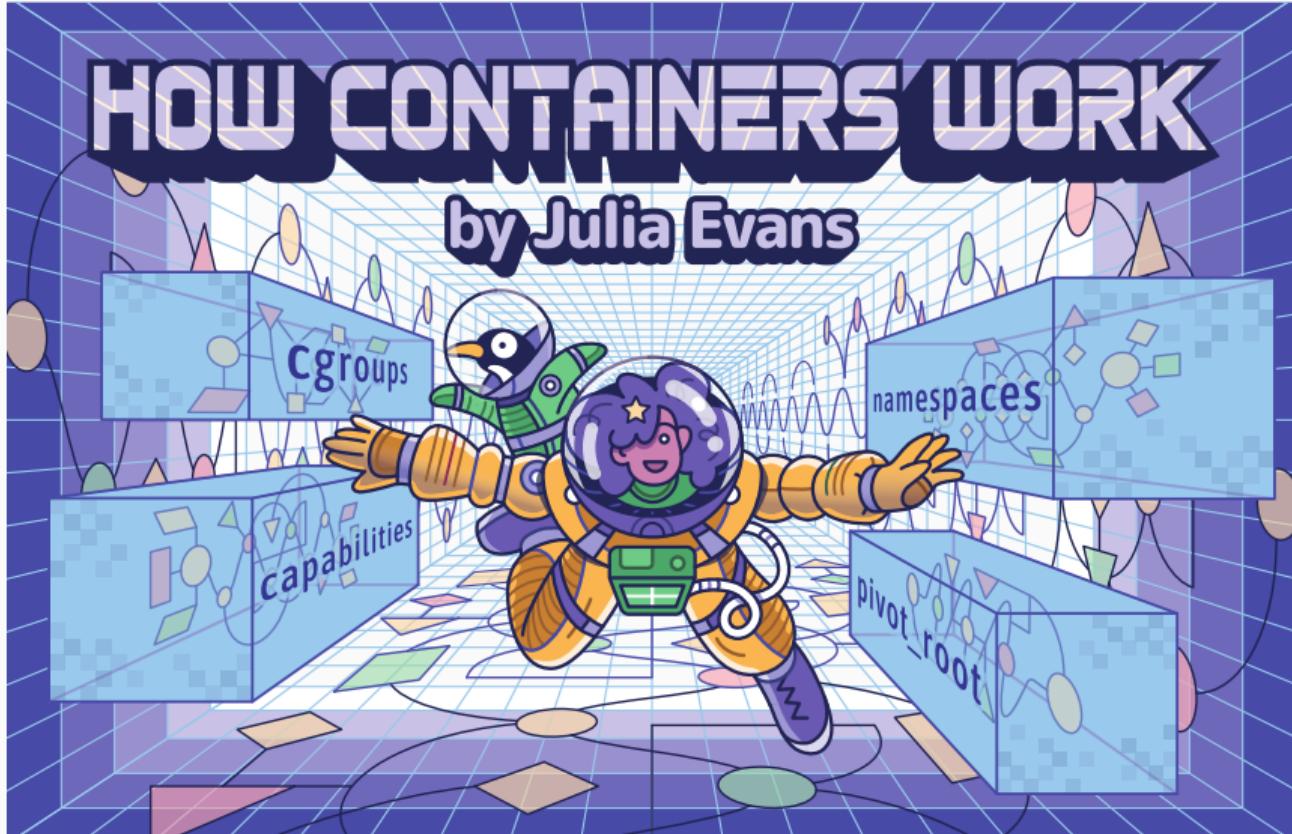
OS-LEVEL VIRTUALIZATION ON LINUX



OS-LEVEL VIRTUALIZATION ON LINUX



HOW DO CONTAINERS WORK?



HOW DO CONTAINERS WORK?

Images used **with permission**:



Pedro Bruel @pedrobruel · 2h

Hey @b0rk, could I use pages 7 and 8 from your containers zine on an undergrad class on OS-level virtualization I'm making? Also, your zines are great!



 **Julia Evans** 
@b0rk

sure!



3

9:10 PM - May 14, 2020



CONTAINERS ON LINUX ARE JUST PROCESSES

containers = processes

7

a container is a group of Linux processes



on a Mac, all your containers are actually running in a Linux virtual machine



I started 'top' in a container.
Here's what that looks like in ps:

outside the container

```
$ ps aux | grep top  
USER PID START COMMAND  
root 23540 20:55 top  
bork 23546 20:57 top
```

inside the container

```
$ ps aux | grep top  
USER PID START COMMAND  
root 25 20:55 top
```

these two are the same process!

container processes can do anything a normal process can ...



I want my container to do X Y Z W!

sure! your computer,
your rules!



... but usually they have
 restrictions

different PID
namespace
different root directory
not allowed to run some system calls
cgroup memory limit
limited capabilities

the restrictions are enforced by the Linux Kernel



NO, you can't have more memory!

on the next page we'll list all the kernel features that make this work!



CONTAINERS ON LINUX USE SOME KERNEL FEATURES

8

container kernel features

containers use these Linux Kernel features

"container" doesn't have a clear definition, but Docker containers use all of these features.

♥ pivot_root ♥

set a process's root directory to a directory with the contents of the the container image

★ cgroups ★

limit memory/CPU usage for a group of processes



only 500 MB of RAM for you!

♥ namespaces ♥

allow processes to have their own:

- | | |
|------------|----------|
| → network | → mounts |
| → PIDs | → users |
| → hostname | + more |

★ capabilities ★

security: give specific permissions

♥ seccomp-bpf ♥

security: prevent dangerous system calls

★ overlay filesystems ★

this is what makes layers work! Sharing layers saves disk space & helps containers start faster

CONTAINERS FROM SCRATCH: OBTAINING AN IMAGE

An **image** usually means:

- A **root** file system, and
- Some **metadata**



We will use the **Alpine** distribution:

- It's root FS has only **2.4MB**
- No need for metadata

Bash Script

```
#!/usr/bin/bash

IMG_DIR="alpine_img"
IMG_REPO="https://us.images.linuxcontainers.org/images"
IMG_URL="$IMG_REPO/alpine/3.11/amd64/default/20200521_13:00/rootfs.tar.xz"
[ ! -d $IMG_DIR ] && \
    mkdir -p $IMG_DIR && \
    curl $IMG_URL | tar xJ -C $IMG_DIR
```

CONTAINERS FROM SCRATCH: CREATING CGROUPS AND SETTING LIMITS

We will create a **cgroup** allowing up to:

- 50% CPU usage: 512/1024 **shares**
- 10GB of RAM

Script

```
CGROUP_ID="MAC0475-145"  
sudo cgcreate -g "cpu,cpuacct,memory:$CGROUP_ID"  
sudo cgset -r cpu.shares=512 "$CGROUP_ID"  
sudo cgset -r memory.limit_in_bytes=100000000000 "$CGROUP_ID"
```

CONTAINERS FROM SCRATCH: LAUNCHING OUR ALPINE CONTAINER

- **cgexec**: Runs using a cgroup
- **unshare**: Runs with new namespaces
- **chroot**: Changes **root** of the file system
- **mount**: Here, mounts a new **proc** directory
- **sh**: Starts a shell on the **container**
- We could install **dependencies** now

Script

```
HOSTNAME="alpine-container"
sudo cgexec -g "cpu,cpuacct,memory:$CGROUP_ID" \
    unshare -fmuipn --mount-proc \
    chroot "$IMG_DIR/" \
    /bin/sh -c "PATH=/bin && mount -t proc proc /proc && hostname $HOSTNAME && sh"
```

And some **cleanup** after:

```
sudo cgdelete cpu,cpuacct,memory:$CGROUP_ID
```

CONTAINERS FROM SCRATCH: RESOURCES

Talks

- Liz Rice, GOTO 2018
- Liz Rice, Container Camp
- Antony Shaw, Pycon

Code

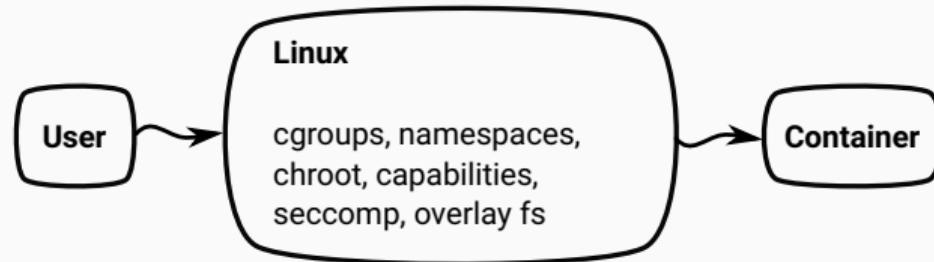
- lizrice, containers from scratch in Go
- Bocker, docker in bash
- Mocker, docker in python

Tutorials

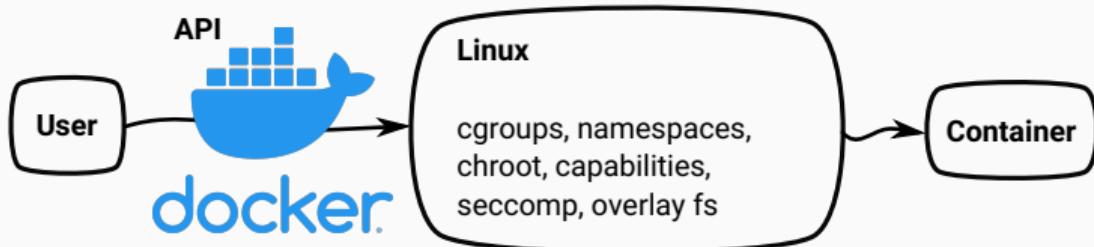
- btholt, Complete Intro to Containers



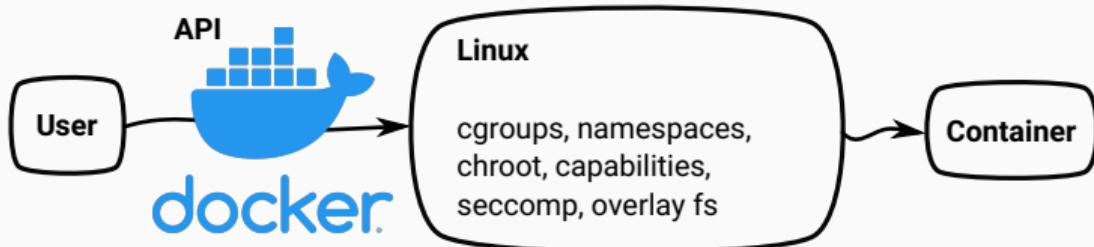
THE DOCKER API FOR CONTAINERS



THE DOCKER API FOR CONTAINERS



THE DOCKER API FOR CONTAINERS



Reproducing our Alpine Container

```
#! /bin/bash
```

```
sudo docker image pull alpine
sudo docker container run -it --memory=10g --cpu-shares=512 alpine
```

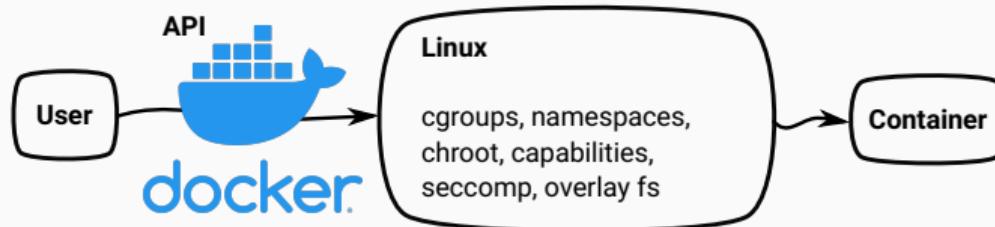
THE DOCKER API FOR CONTAINERS

Some API functions:

Docker API		Description	In Our Script
image	pull	Downloads images	mkdir, curl, tar
	ls	Lists downloaded images	
	save	Writes image to a .tar	
	build	Builds an image	
docker	run	Runs containers in images	cgcreate, cgset, cgexec, unshare, chroot, hostname, mount
	container		
	ls	Lists running containers	
	attach	Attaches to a container	
	commit	Saves container to image	

- Check the examples and the docs for more

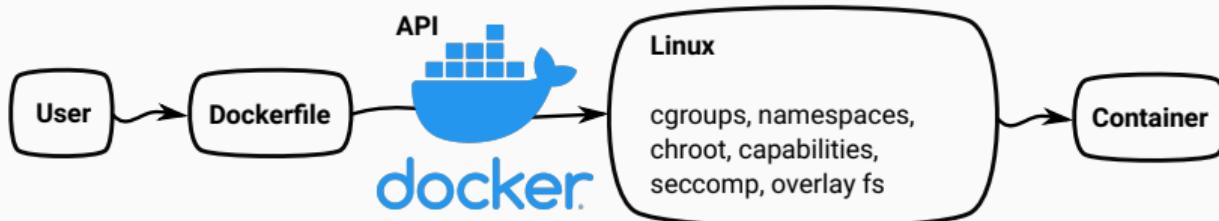
ENVIRONMENT VERSIONING WITH DOCKERFILES



Dockerfiles

- Similar to **makefiles**
- Define container **properties**:
 - Versions of images from dockerhub
 - Environment variables
 - Dependencies

ENVIRONMENT VERSIONING WITH DOCKERFILES



Dockerfiles

- Similar to **makefiles**
- Define container **properties**:
 - Versions of images from dockerhub
 - Environment variables
 - Dependencies

DOCKERFILES: A SIMPLE BULLETIN BOARD

Cloning the Repository

```
git clone https://github.com/dockersamples/node-bulletin-board
```

Dockerfile

```
FROM node:current-slim
WORKDIR /usr/src/app
COPY package.json .
RUN npm install
EXPOSE 8080
CMD [ "npm", "start" ]
COPY ..
```

DOCKERFILES: BUILDING AND RUNNING

Building the Image

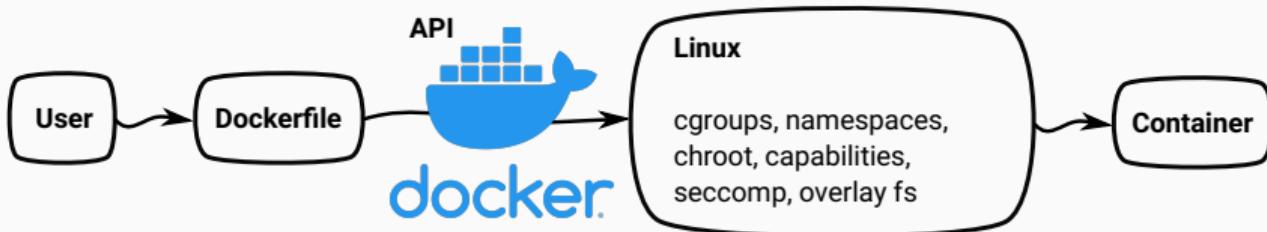
```
cd node-bulletin-board/bulletin-board-app  
sudo docker image build --tag bulletinboard:1.0 .  
sudo docker container run --publish 8000:8080 --detach --name bb bulletinboard:1.0
```

Cleaning up

```
cd node-bulletin-board/bulletin-board-app  
sudo docker container rm --force bb
```

- Check the complete tutorial

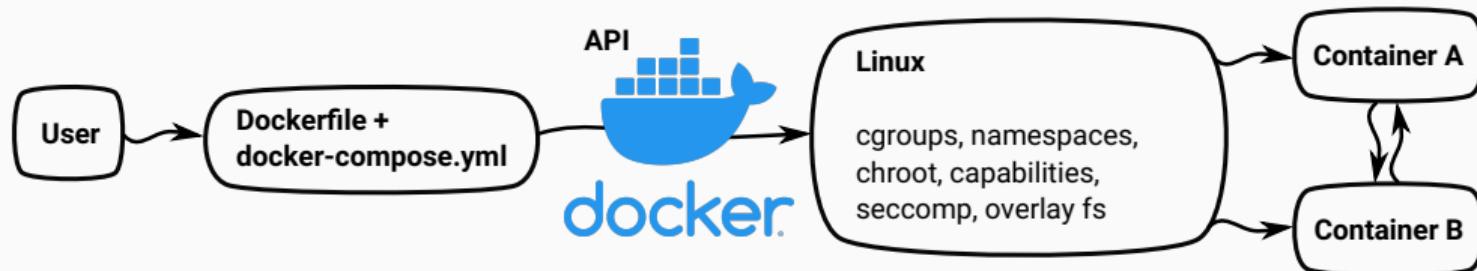
COMBINING SERVICES WITH DOCKER COMPOSE



Docker-compose

- Uses **Dockerfiles** too
- Defines **services** on **separate containers**:
 - Configure service **communication**
 - Maintain separate service **projects**

COMBINING SERVICES WITH DOCKER COMPOSE



Docker-compose

- Uses **Dockerfiles** too
- Defines **services** on **separate containers**:
 - Configure service **communication**
 - Maintain separate service **projects**

COMBINING SERVICES WITH DOCKER COMPOSE: FLASK + REDIS

```
import time, redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host = 'redis', port = 6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

COMBINING SERVICES WITH DOCKER COMPOSE: FLASK + REDIS

Flask Dockerfile

- Use an Alpine image, with Python 3.7
- Configure and install **Flask** dependencies
- Define a default **container command**

```
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP app.py
ENV FLASK_RUN_HOST 0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY ..
CMD ["flask", "run"]
```

COMBINING SERVICES WITH DOCKER COMPOSE: FLASK + REDIS

Docker Compose Configuration

- Define **service architecture**
- Use the **default** Redis Alpine image

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

- Check the complete tutorial

OS-LEVEL VIRTUALIZATION: CONCLUSION

Take-away

- You should **use containers!**
 - Environment versioning
 - Reproducible builds
 - It's **easy**
- Containers are **processes**
- Many **tools** are available:

