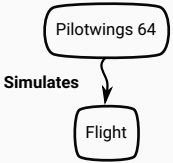
# Introduction to OS-Level Virtualization on Linux

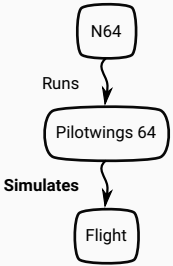
Pedro Bruel
*phrb@ime.usp.br*

May 25th, 2020

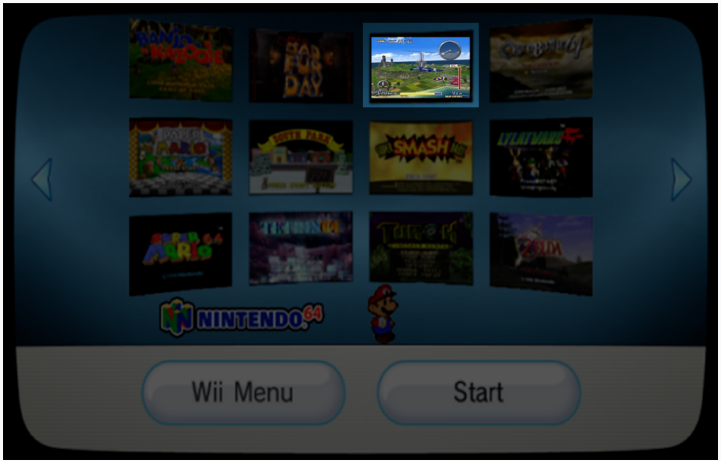
# What are Simulation, Emulation, Virtualization?

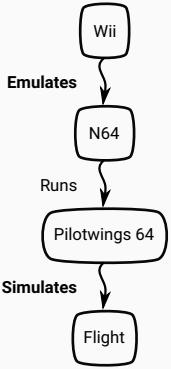



# What are Simulation, Emulation, Virtualization?

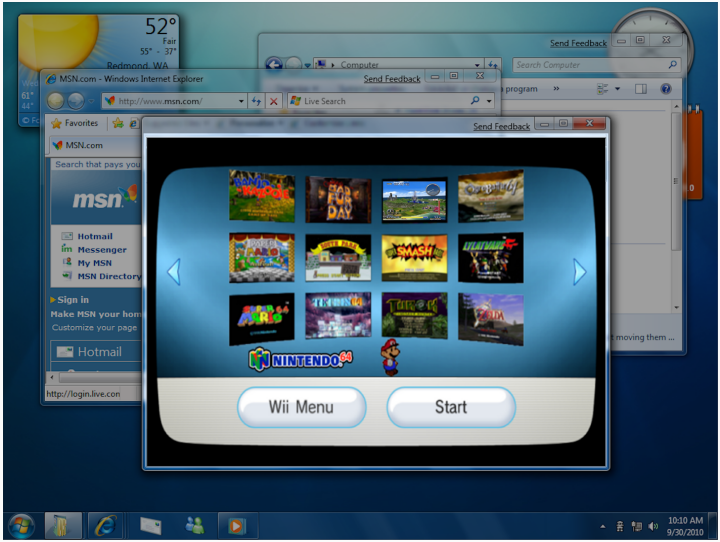



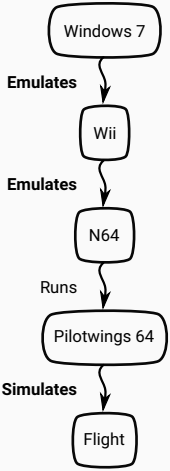
# What are Simulation, Emulation, Virtualization?



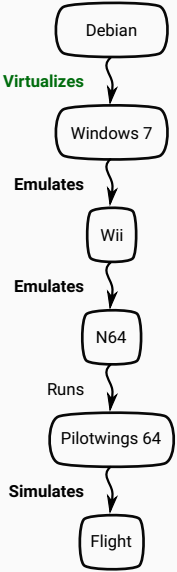



# What are Simulation, Emulation, Virtualization?

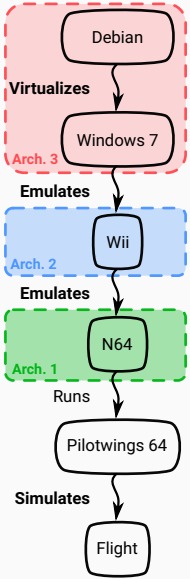


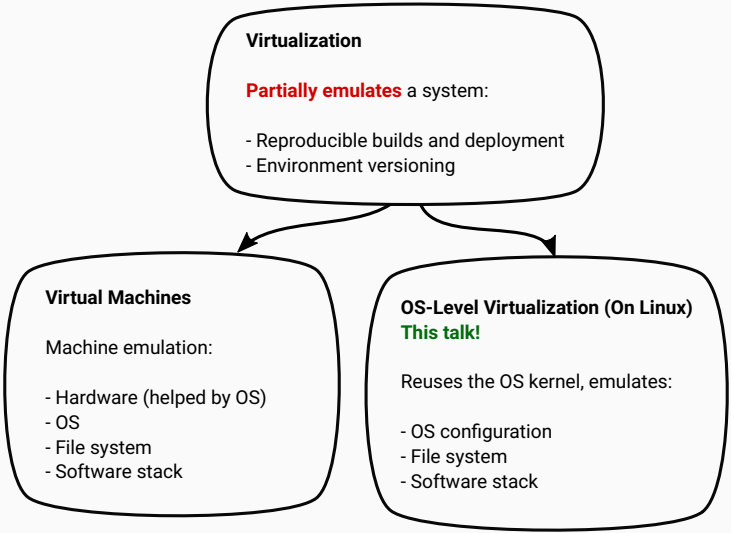
# What are Simulation, Emulation, Virtualization?

# What are Simulation, Emulation, Virtualization?

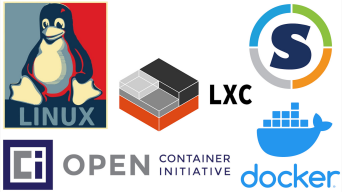
# OS-Level Virtualization

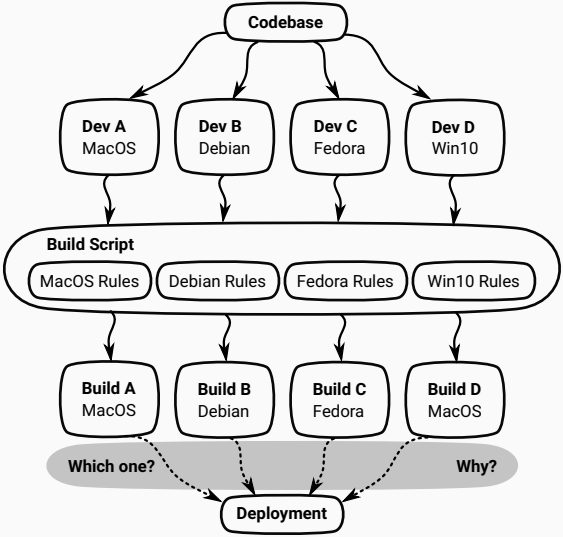


# OS-Level Virtualization: Scope of this Talk

## Scope

- Why should you use containers?
  - Reproducible builds
  - Environment versioning
  - It's also easier
- How do containers work?
- What tools are available?

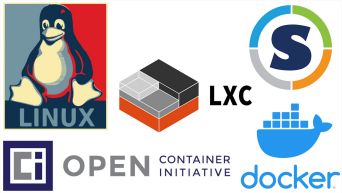


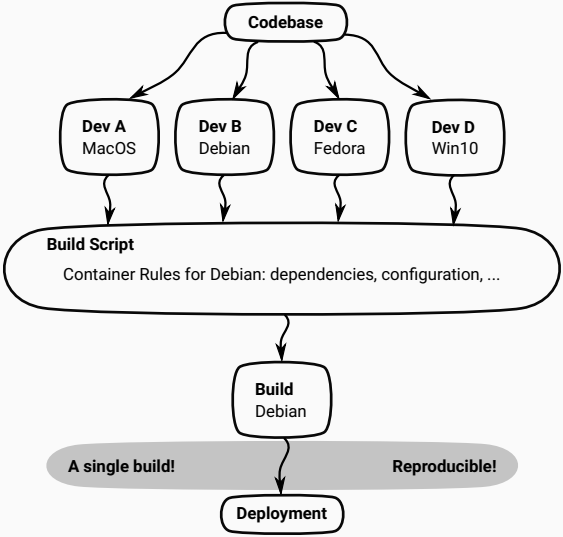



# OS-Level Virtualization: Scope of this Talk

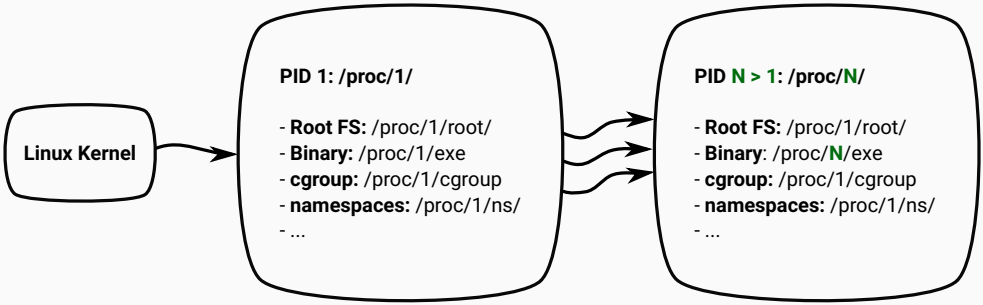
## Scope

- Why should you use containers?
  - Reproducible builds
  - Environment versioning
  - It's also easier
- How do containers work?
- What tools are available?

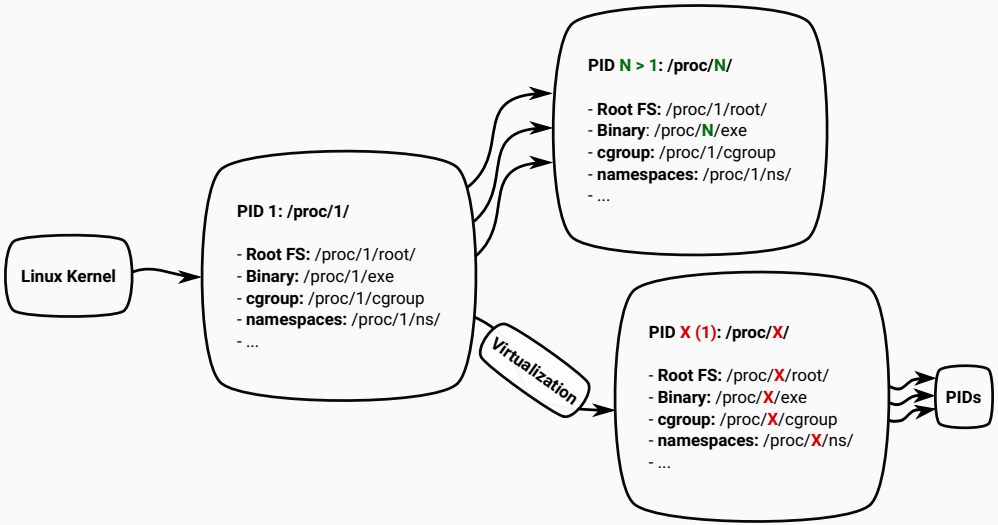


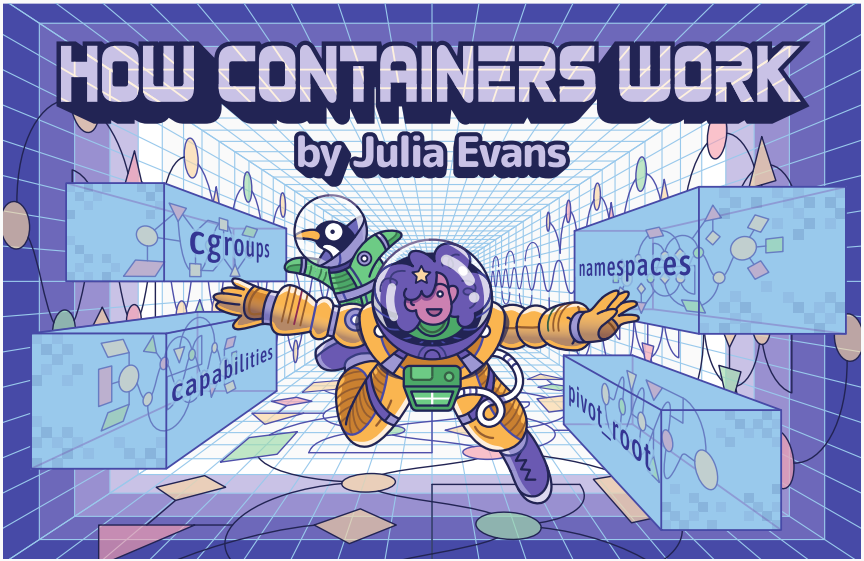
# OS-Level Virtualization on Linux



# OS-Level Virtualization on Linux

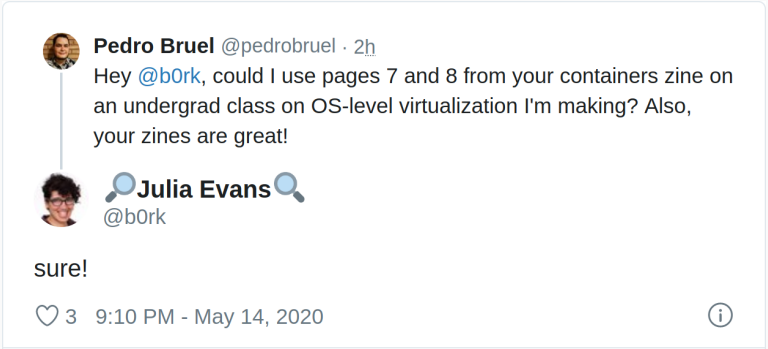


# How do Containers Work?



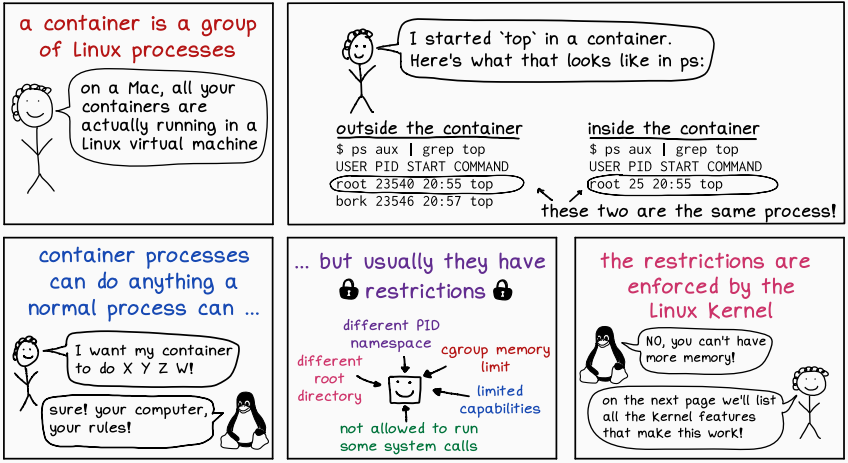
# How do Containers Work?

Images used with permission:



# containers = processes

7



# container kernel features

8

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


♥ 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: Obtaing 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=10000000000 "$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 depencies 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

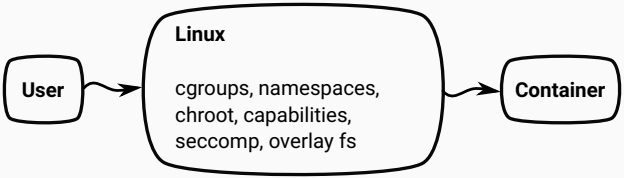
**Code**

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

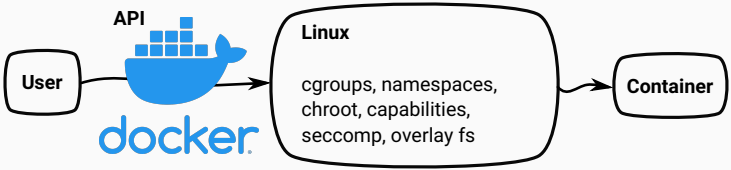
**Tutorials**

- btholt, Complete Intro to Containers

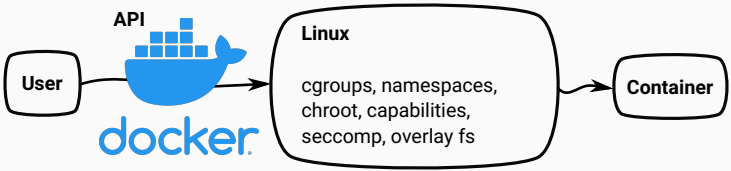
# The Docker API for Containers



# The Docker API for Containers



# The Docker API for Containers




## Reproducing the Alpine Container

```
#! /bin/bash

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

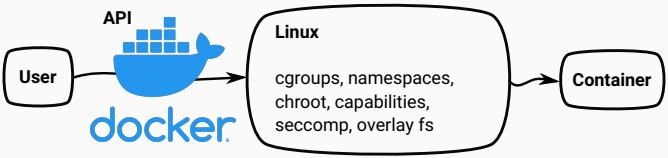
# The Docker API for Containers

Some API functions:

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

- Check the examples and the docs for more

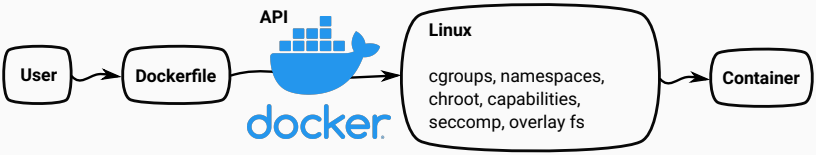
# Environment Versioning with Dockerfiles




## Dockerfiles

- Similar to makefiles
- Define container properties:
  - Versions of images from dockerhub
  - Environment variables
  - Install dependencies

# Environment Versioning with Dockerfiles




## Dockerfiles

- Similar to makefiles
- Define container properties:
  - Versions of images from dockerhub
  - Environment variables
  - Install dependencies

## 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 . .
```

### Building the Image

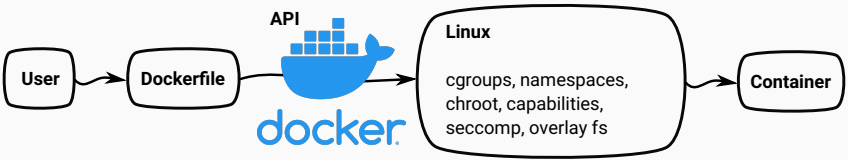
```
cd node-bulletin-board/bulletin-board-app
sudo docker image build --tag bulletinboard:1.0 .
sudo docker container run --publish 8000:8080 --detach --name bb bulletinboard:1.0
```

### Cleaning up

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

- Check the complete tutorial

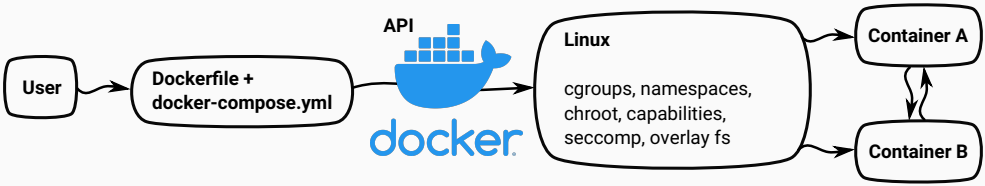
# Combining Services with Docker Compose




## 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)
```

### 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!
  - Reproducible builds
  - Environment versioning
  - It's easy
- Containers are processes
- Many tools are available: