Distributed Systems

15-440/640

Fall 2018

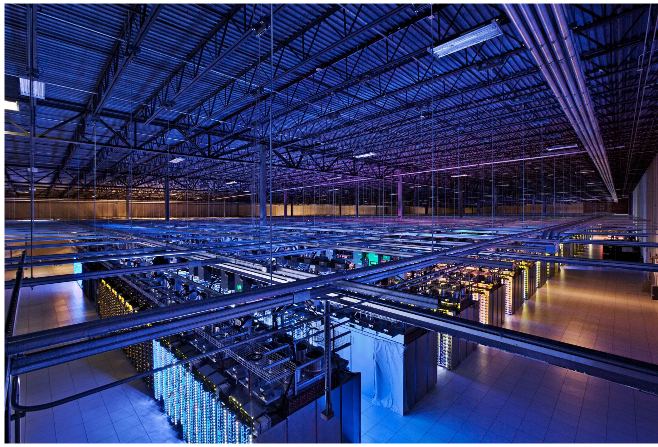
20– Virtualization Techniques

Readings: book chapter on Virtual Machines from the Wisconsin OS book

Load Scalability

Characteristic of good design for distributed systems

- small marginal load due to each additional client
- maximum # of clients with fixed # servers
- aggressive caching helps if workload is right



Need: ability to dynamically grow resources

- hard to do with real resources
  - → purchase of new servers, storage, networks, etc.
  - → growing/shrinking over small timeframes/quanta not feasible
- made possible by **virtualization**
  - → primarily VMs, but extends to other resources as well
  - → e.g “software-defined networking” virtualizes network components
  - → e.g. “software-defined storage”virtualizes storage components

Success of Virtualization

Virtualization can transform CAPEX into OPEX

- CAPEX → "capital expenses"
- OPEX → "operational expenses"

  → smaller incremental investments, different accounting rules

  → great boon for startups and small mature companies
- cloud owner (e.g. Amazon) incurs CAPEX
- cloud users (e.g. startup) incurs only OPEX

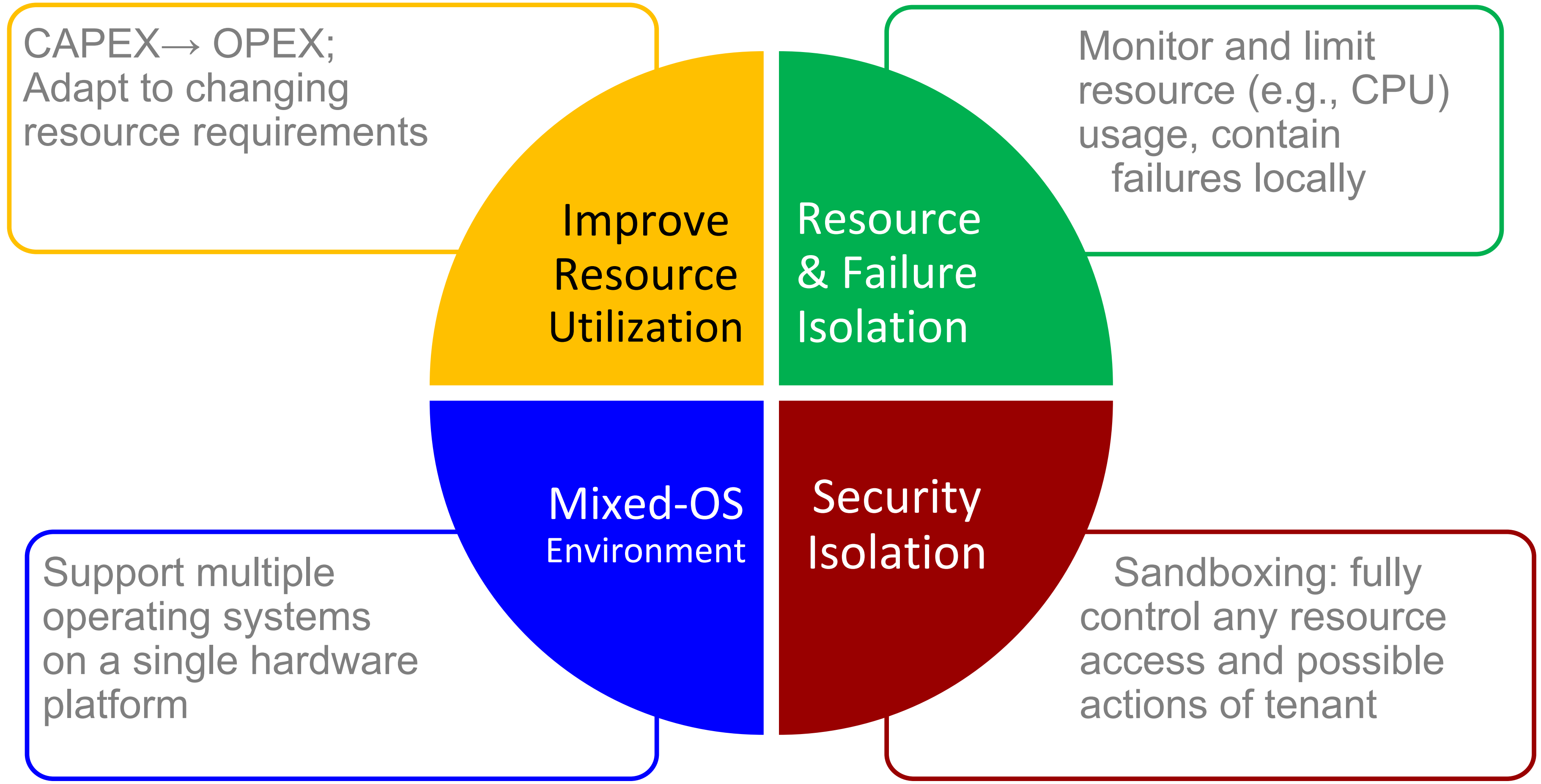
  → cloud owner makes a profit from OPEX pricing

  → like the difference between renting and buying a home

Flexible allocation of resources in cloud → **"elasticity"**

- "EC2" in "Amazon EC2" stands for "elastic cloud computing"

More Reasons for Virtualization


CAPEX→ OPEX; Adapt to changing resource requirements
Improve Resource Utilization
Resource & Failure Isolation
Monitor and limit resource (e.g., CPU) usage, contain failures locally
Mixed-OS Environment
Support multiple operating systems on a single hardware platform
Security Isolation
Sandboxing: fully control any resource access and possible actions of tenant

Roots of VM Technology

Roots of today's VMs reach back to 1960s

M44/44X (IBM), CTSS (MIT), {CP-40, CP-67, CP/CMS} (IBM) VM/370 (IBM product, 1972)

What was the driving force?

- Hardware very expensive (mainframes) → few machines
- Explosion of effort in low-level system software
- Pain point: need real hardware for testing

→ "nearly identical" not good enough

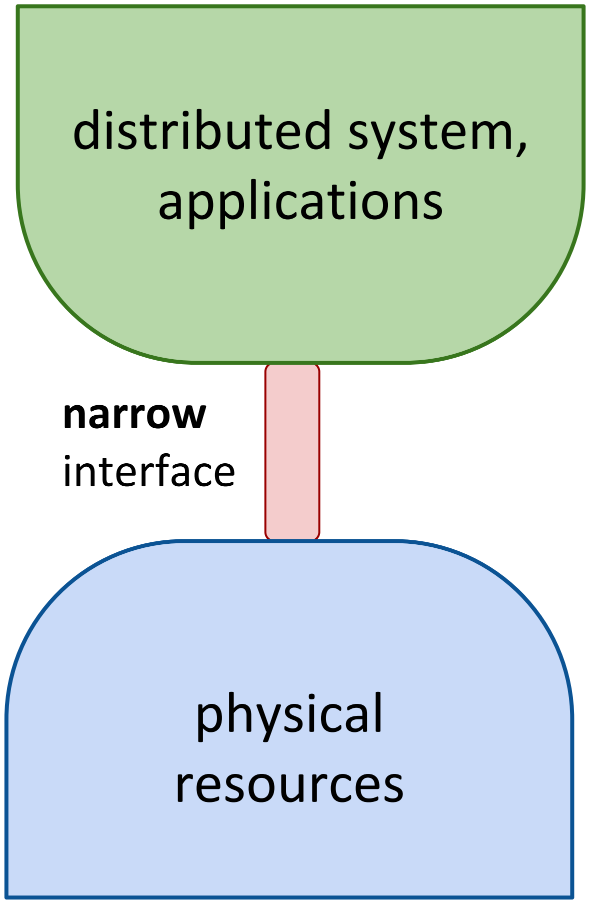
Hardware virtualization wins big

- enhances productivity of system software development
- new software runs concurrently with older versions
- "innovation" multiple concurrent users on same machine

The Strange History of VMs

mid-1960s to early 1970s	birth and emergence
early 1970s to late 1970s	extensive commercial use (VM/CMS)
late 1970s to early 1980s	emergence of personal computers (IBM PC)
late 1980s to late 1990s	"demise" of VMs
late 1990s	rebirth of VMs (VMware)
early 2000s	resurgence of research interest in VMs
late 2000s to present	explosion of commercial interest (cloud computing)

Virtualization Techniques



Separate
- physical characteristics of resources
- from the way in which other systems, applications, or end users interact

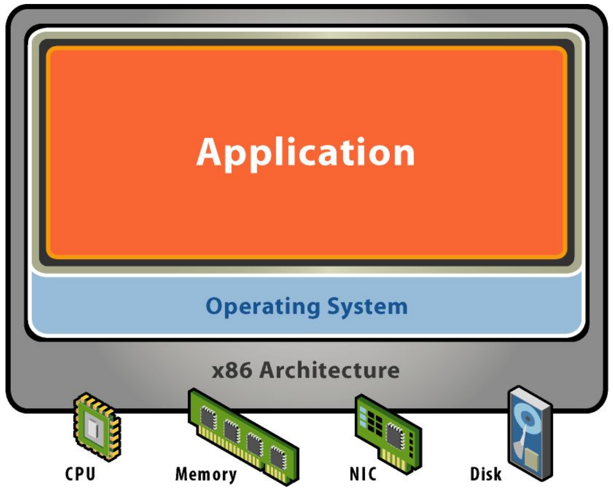
Why Is Hardware Special?

Narrow & stable waistline critical
- narrow: freer innovation
- narrow: vendor neutrality
- stable: longevity / ubiquity

Wide interfaces → brittle abstractions
- hard to: deploy, sustain, scale
- e.g., software interface: processes

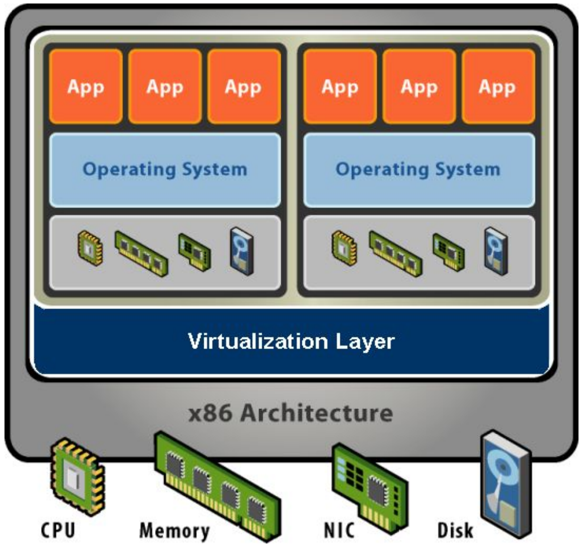
Adapted from: Ken Birman

Starting Point: Physical Machine



- Physical Hardware
  - Processors, memory, chipset, I/O devices, etc.
  - Resources often grossly underutilized
- Software
  - Tightly coupled to physical hardware
  - Single active OS instance
  - OS controls hardware

Virtualizing This Machine

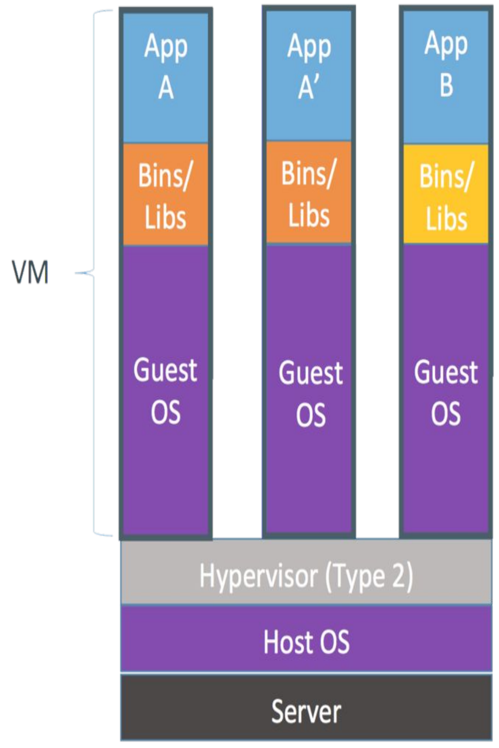


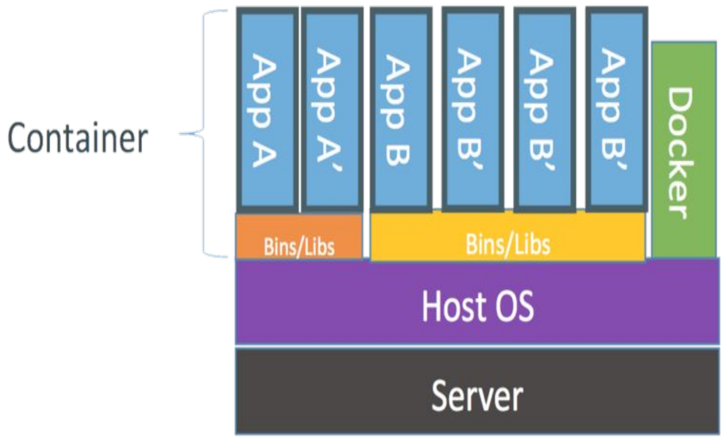
- Software Abstraction
  - Behaves like hardware
  - Encapsulates all OS and application state
- Virtualization Layer
  - Extra level of *indirection*
  - Decouples hardware, OS
  - Enforces isolation
  - Multiplexes physical hardware across tenants

Two main types of virtualization layers

Types of Virtualization

- System virtualization
  - Virtualizing the entire hardware interface
- Container virtualization
  - Virtualizing OS resources between processes





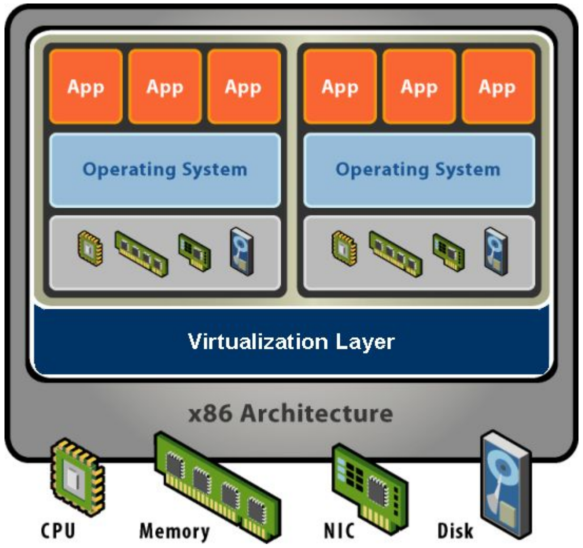
Topics Today

Motivation

System Virtualization

Container Virtualization

Virtual Machines



- Implemented via Virtual Machine Monitor (VMM, aka hypervisor)
- Classic Definition (Popek and Goldberg ’74)

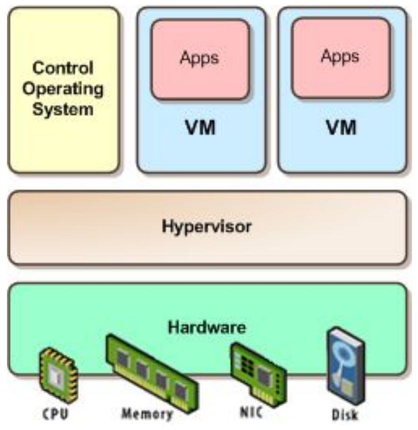
A virtual machine is … an efficient, isolated duplicate of the real machine. … the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

- VMM Properties
  - **Fidelity**: Programs running in the virtualized environment run identically to running natively.
  - **Performance**: A statistically dominant subset of the instructions must be executed directly on the CPU.
  - **Safety and isolation**: The VMM must completely control access to system resources.

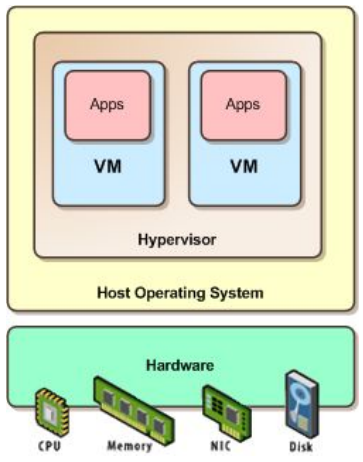
Again: two types (within system virtualization)

Types of System Virtualization

- Type 1: Native/Bare metal
  - Higher performance
  - Xen, Hyper-V



- Type 2: Hosted
  - Easier to install
  - Leverage host's device drivers
  - VMware Workstation, Parallels



Picture from: https://itechthoughts.wordpress.com/tag/full-virtualization/

Requirements on VMs

- Isolation
  - Fault isolation
  - Performance isolation (+ software isolation, …)

Resource & Failure Isolation

- Encapsulation
  - Cleanly capture all VM state
  - Enables VM snapshots, clones

Mixed-OS Environment

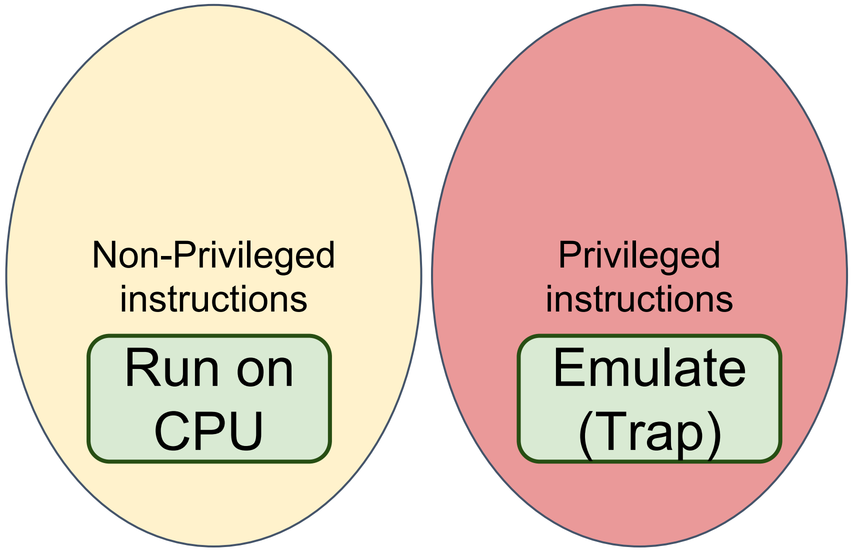
- Portability
  - Independent of physical hardware
  - Enables migration of live, running VMs (freeze, suspend,…)
  - Clone VMs easily, make copies

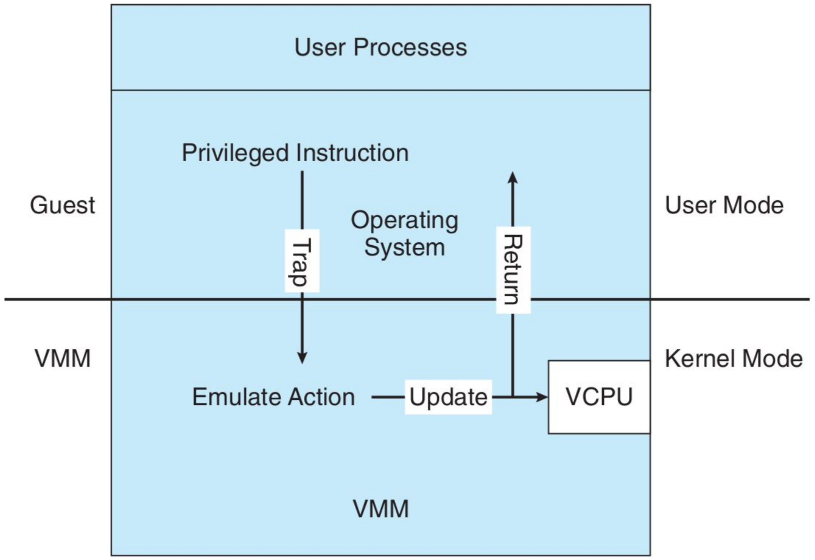
Improved Resource Utilization

- Interposition
  - Transformations on **instructions, memory, I/O**
  - Enables transparent resource overcommitment, compression, replication …

Security Isolation

Efficient CPU Virtualization





- **Non-privileged instructions** (e.g., Load from mem): Run as native machine
- **Privileged instructions** (e.g., Update CPU state, Manipulate page table): Trap to VMM

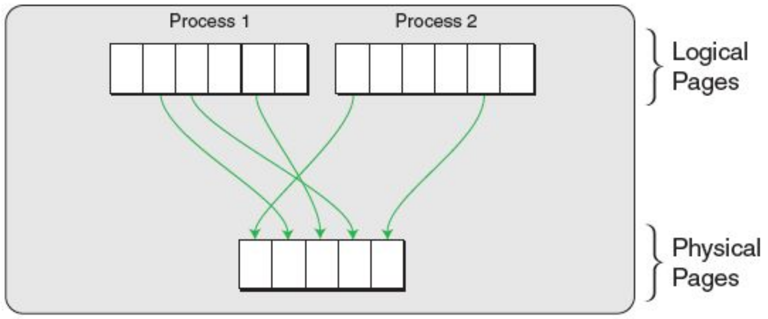
This is called Trap and Emulate
→ Full Control for VMM

More complex in reality (some privileged instructions don't trap) → Processor support VT-x, AMD-V

OS Memory Abstraction

- OS assumes that it has full control over memory
  - Management: Assumes it owns it all
  - Mapping: Assumes it can map any Virtual⟶ Physical

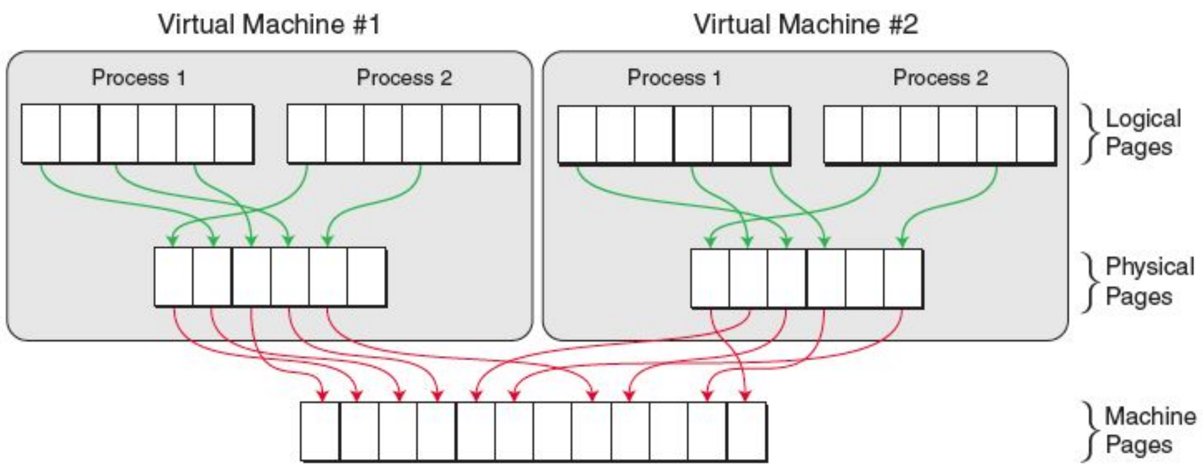
Native machine



- However, VMM partitions memory among VMs
  - VMM needs to assign hardware pages to VMs
  - VMM needs to control mapping for isolation
    - Cannot allow OS to map any Virtual ⇒ hardware page

Daniel S. Berger Adapted from: Alex Snoeren 15-440 Fall 2018 Carnegie Mellon University

Virtualized Memory: Three Levels of Abstraction

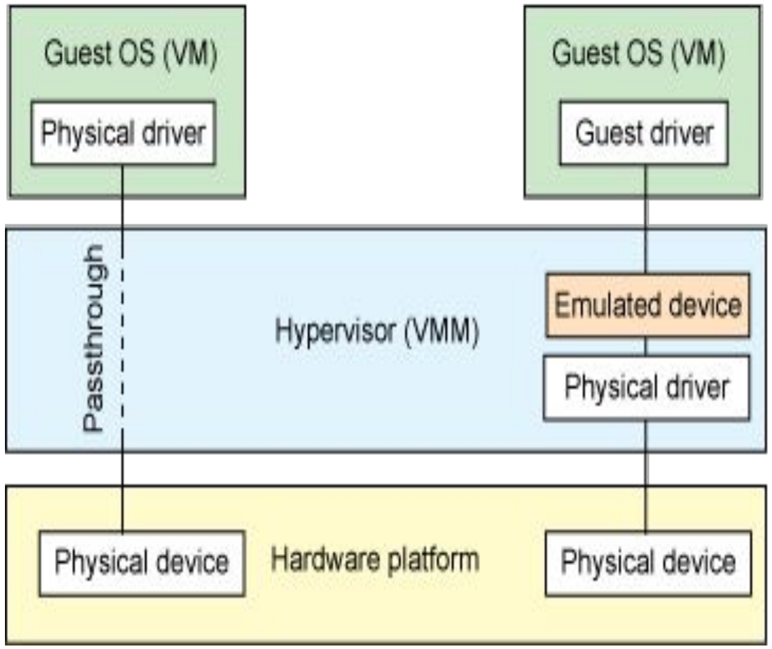


- **Logical**: process address space in a VM
- **Physical**: abstraction of hardware memory. Managed by guest OS
- **Machine**: actual hardware memory (e.g. 2GB of DRAM)

Subtle challenges in real implementations
(e.g., page table updates don't trap)

I/O Virtualization

- Direct access: VMs can directly access devices
  - Requires H/W support (e.g., DMA passthrough, SR-IOV)
- Shared access: VMM provides an emulated device and routes I/O data to and from the device and VMs

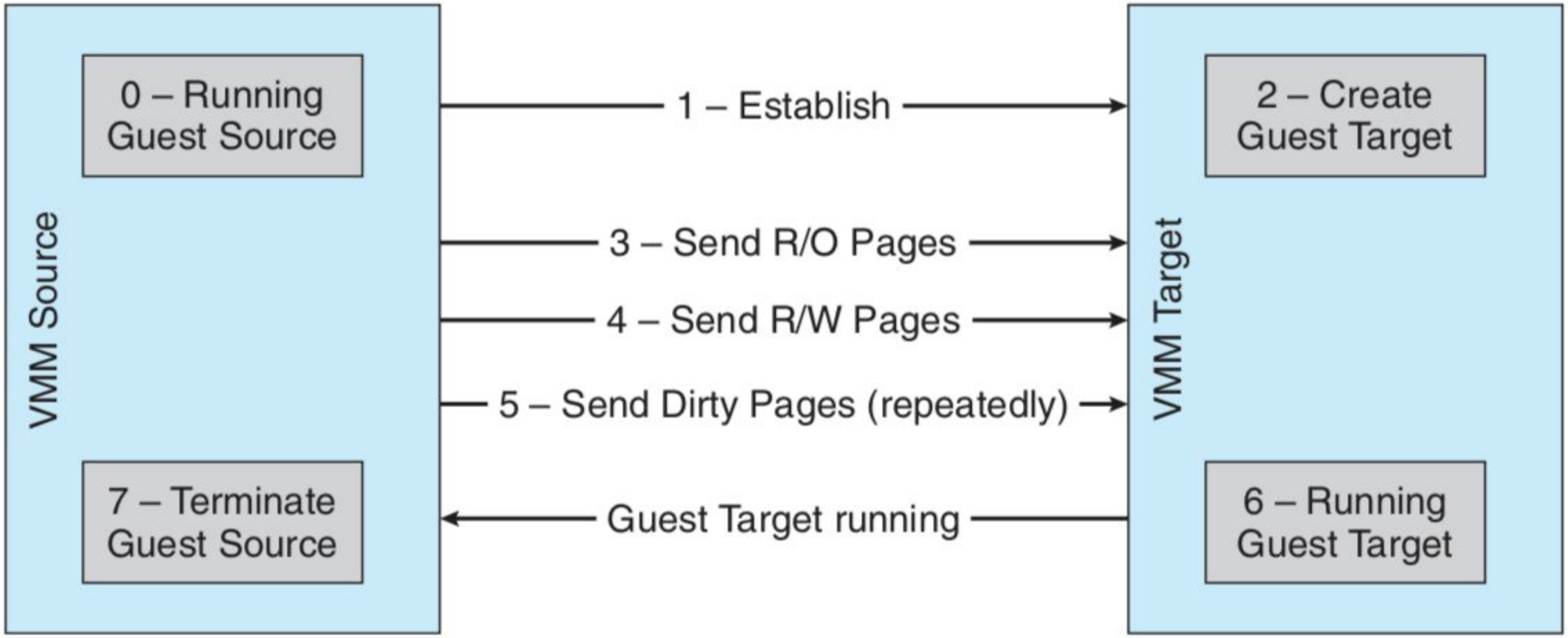


- VMM provides "virtual disks"
  - Type 1 VMM – store guest root disks and config information within file system provided by VMM as a disk image
  - Type 2 VMM – store the same info as files in the host OS' file system

Live migration

- Running guest OS can be moved between systems, without interrupting user access to the guest or its apps
- Supported by type 1 hypervisors
- Very useful for resource management, no downtime, etc
- How does it work?
  1. Source VMM connects to the target VMM
  2. Target VMM creates a new guest (e.g. create a new VCPU, etc)
  3. Source sends all read-only guest memory pages to the target
  4. Source sends all RD/WR pages to the target, marking them clean
  5. Source repeats step 4, as some pages may be modified ⇒ dirty
  6. When cycle of steps 4 and 5 becomes very short, source VMM freezes guest, sends VCPU's final state, sends final dirty pages, and tells target to start running the guest
  7. Target acknowledges that guest running ⇒ source terminates guest

Live migration



Virtual Machine Summary

- VMMs multiplex virtual machines on hardware
  - Virtualize CPU, Memory, and I/O devices
  - Run OSes in VMs, apps in OSes unmodified
  - Run different versions, kinds of OSes simultaneously

- Support for virtualization built into CPUs
  - Goal is to fully virtualize architecture
  - Required for transparent trap-and-emulate

Virtual machines add significant overhead.

Topics Today

Motivation

System Virtualization (VMs)

Container Virtualization

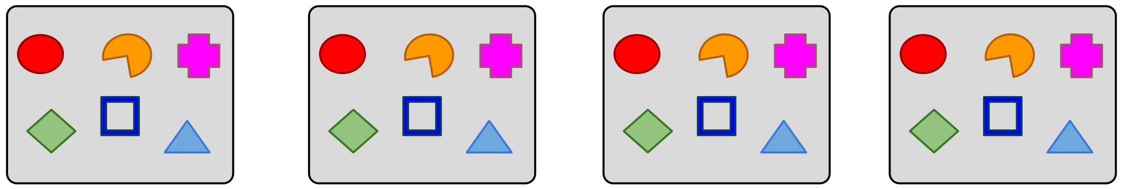
- Motivation for Containers
- Implementation in Linux
- Practical Implications

Motivation for Containers

Architecture of web applications is changing

Classical architecture

Monolithic application
100 engineers
Release / month
Horizontal scale out



Components

- ● Login
- ◔ Personification
- ◆ Renderer
- ☐ Ads
- ✚ Suggestions
- ▲ Encoders

Potential limitations of this architecture?

.WAR too big for IDE?

Async release of updates?

Change tech of a component?

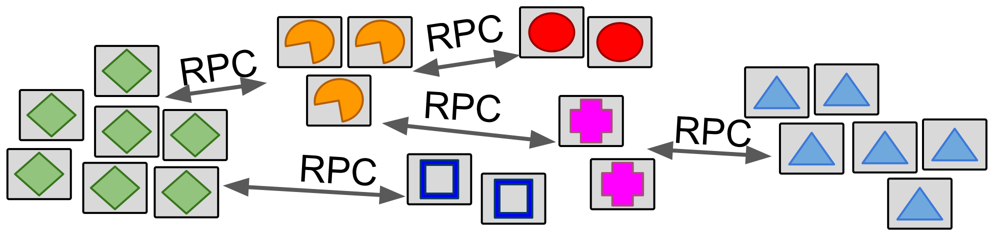
Failure Isolation?

Motivation for Containers

Changing architecture of web applications

New architecture: components → "micro services"

Define API between components
10-20 engineers / component
Components release and scale independently



Components
- 🔴 Login
- ◔ Personification
- ◆ Renderer
- □ Ads
- ✚ Suggestions
- △ Encoders

Potential limitations of this new architecture?

Per-component overhead?

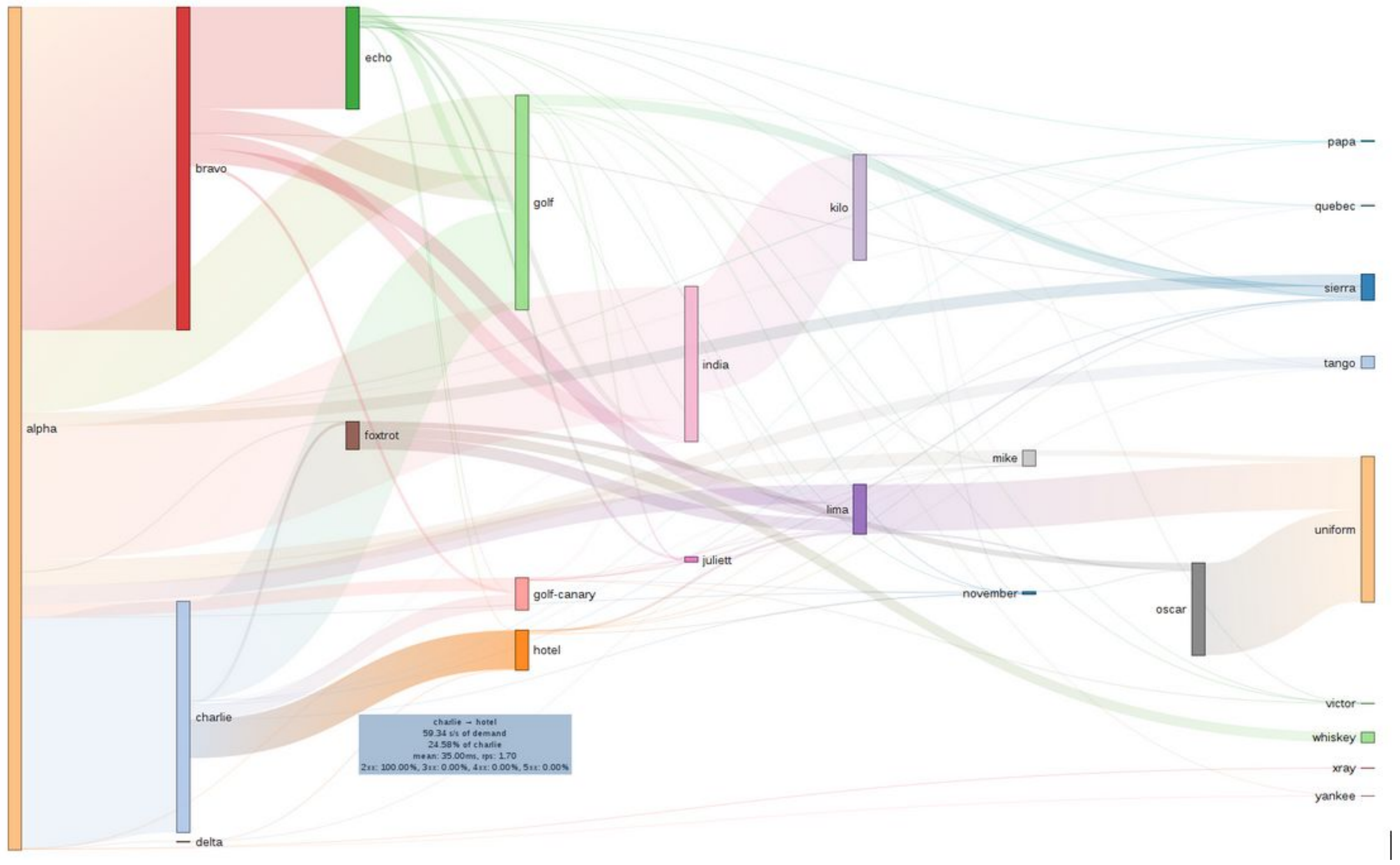
How to define services?

API/Communication latency?

Prominent Example: Netflix

Migration to micro services: 2008-2016

Hundreds of services, complex dependencies



Why Container Virtualization?

Overhead associated with deploying on VMs

- I/O overhead
- OS-startup overhead per VM
- Memory/Disk overhead (duplicate data)

Overhead becomes dominant at scale: thousands of VMs / server

Perception: VM have too much overhead!

New idea:

- Multiple isolated instances of programs
- Running in user-space (shared kernel)
- Instances see only resources (files, devices) assigned to their container

Other names: OS-level virtualization, partitions, jails (FreeBSD jail, chroot jail)

Requirements on Containers

- Isolation and encapsulation
  - Fault and performance isolation
  - Encapsulation of environment, libraries, etc.

Resource & Failure Isolation

- Low overhead
  - Fast instantiation / startup
  - Small per-operation overhead (I/O, ..)

Improved Resource Utilization

- Reduced Portability

Mixed-OS Environment

- ~~Interposition~~ (no hypervisor)

Security Isolation

Implementation

Key problems:

- Isolating which resources containers see
- Isolating resource usage
- Efficient per-container filesystems

Resource View Isolation

Problem: containers should only see "their" resources, and are the only users of their resource

(e.g., process IDs (PIDs), hostnames, users IDs (UIDs), interprocess communication (IPC))

Solution: each process is assigned a "namespace"

- Syscalls only show resources within own namespace
- Subprocesses inherit namespace

Current implementation: namespace implementation per resource type (PIDs, UIDs, networks, IPC), in Linux since 2006

Practical implication:

- Containers feel like VMs, can get root
- Security relies on kernel, containers make direct syscalls

Resource Usage Isolation

Problem: meter resource usage and enforce hard limits per container

(e.g., limit memory usage, priorities for CPU and I/O usage)

Solution: usage counters for groups of processes (cgroups)

- Compressible resources (CPU, I/O bandwidth): rate limiting
- Non-compressible resources (Memory/disk space): require terminating containers (e,g., OOM killer per cgroup)

Current implementation: cgroups/kernfs, in Linux since 2013/2014

Practical implication:

- Efficiency: 1000s of containers on a single host
- Small overhead per memory allocation, and in CPU scheduler

Filesystem Isolation

Problem: per-container filesystems without overhead of a "virtual disk" for each container

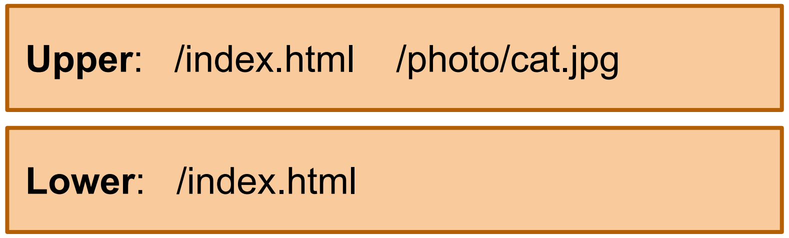
Solution: layering of filesystems (copy on write):

- Read-write ("upper") layer that keeps per-container file changes
- Read-only ("lower") layer for original files

Current implementation: OverlayFS, in Linux since 2014

Practical implication:

- Instant container startup
- "Upper" layer is ephemeral

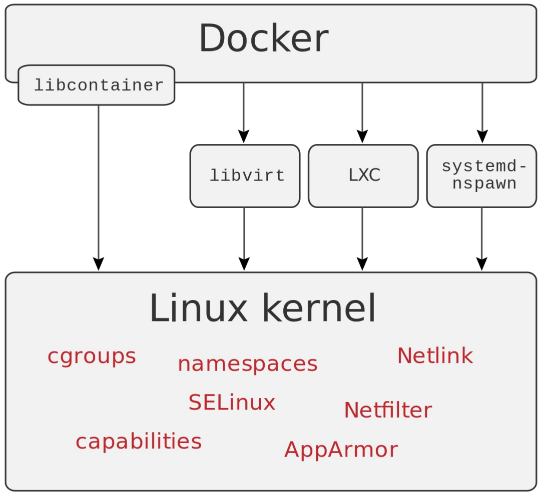


The Container Ecosystem

Docker OPEN CONTAINER INITIATIVE (also: LXC, Google lmctfy)

Libcontainer (written in GO)

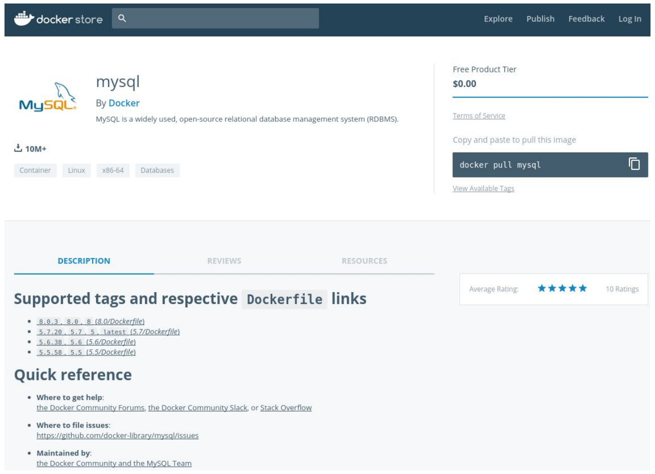
- Automates using kernel features (namespaces, cgroups, OverlayFS)
- Container-image configuration language



```
FROM golang

WORKDIR /go/src
COPY ./src .
RUN go-wrapper install monitor

CMD ./start.sh
```


docker store Explore Publish Feedback Log In

mysql
By Docker
MySQL is a widely used, open-source relational database management system (RDBMS).

10M+

Container Linux x86-64 Databases

Free Product Tier
$0.00

Terms of Service

Copy and paste to pull this image

```
docker pull mysql
```

View Available Tags

DESCRIPTION REVIEWS RESOURCES

Average Rating: ★★★★★ 10 Ratings

Supported tags and respective `Dockerfile` links

- `8.0.3`, `8.0`, `8` (*8.0/Dockerfile*)
- `5.7.20`, `5.7`, `5`, `latest` (*5.7/Dockerfile*)
- `5.6.38`, `5.6` (*5.6/Dockerfile*)
- `5.5.58`, `5.5` (*5.5/Dockerfile*)

Quick reference

- **Where to get help**:
  the Docker Community Forums, the Docker Community Slack, or Stack Overflow
- **Where to file issues**:
  https://github.com/docker-library/mysql/issues
- **Maintained by**:
  the Docker Community and the MySQL Team

Advantages of Containers

Fast boot times:

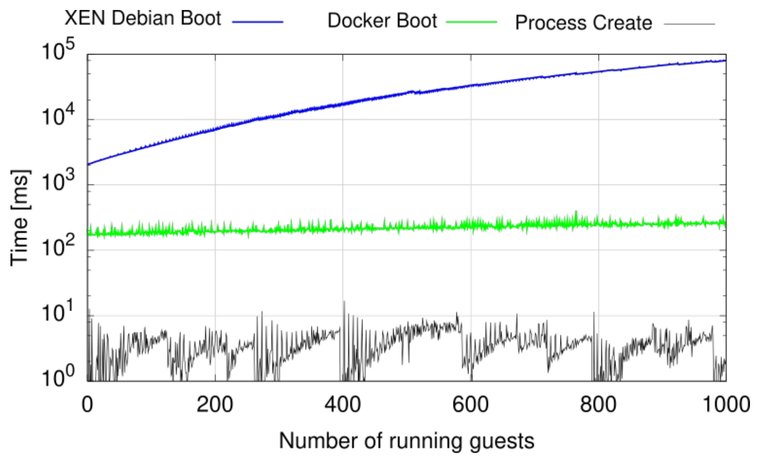
- 100s of milliseconds
- (10s-100s of seconds for VMs)

High density:

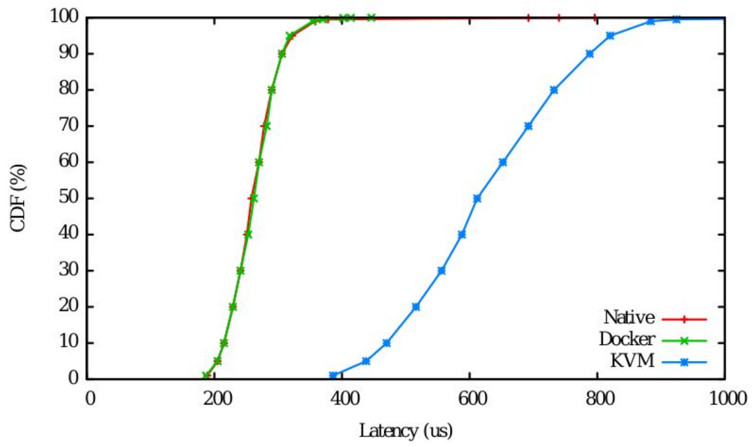
- 1000s of containers per machine

Very small I/O overhead

Require no CPU support



Manco, Filipe, et al. "My VM is Lighter (and Safer) than your Container." SOSP 2017.



Felter, et al. An Updated Performance Comparison of Virtual Machines and Linux Containers, IBM Report 2014.

Limitations of Containers

Implementation Complexity

- Much more complex (“wider”) interface for processes
- Need to configure namespace, cgroup, overlayfs (and more)

Less general than VMs

- Can only run the same Operation System (shared OS)

Harder to migrate than VMs

- State of containers is not fully encapsulated, state leaks into host OS
- In practice: no container migration. Instead: containers are ephemeral - just terminate old one and start new one

Large attack surface under adversarial behavior

- Containers typically have access to all syscalls
  - Linux offers 400 syscalls (10 new syscalls / year)
- One approach: syscall filtering (very complicated)

Summary

VMs

Strengths: strong isolation guarantees, can run different OSs

VM migration practical

Weaknesses: OS startup, disk,memory, and hypervisor overhead

Containers

Strength: fast startup times, negligible I/O overheads, very high density

Weaknesses: weak security isolation

In practice: techniques complement each other

Use VMs to isolate between different users, and containers to isolate different applications/services of a single user