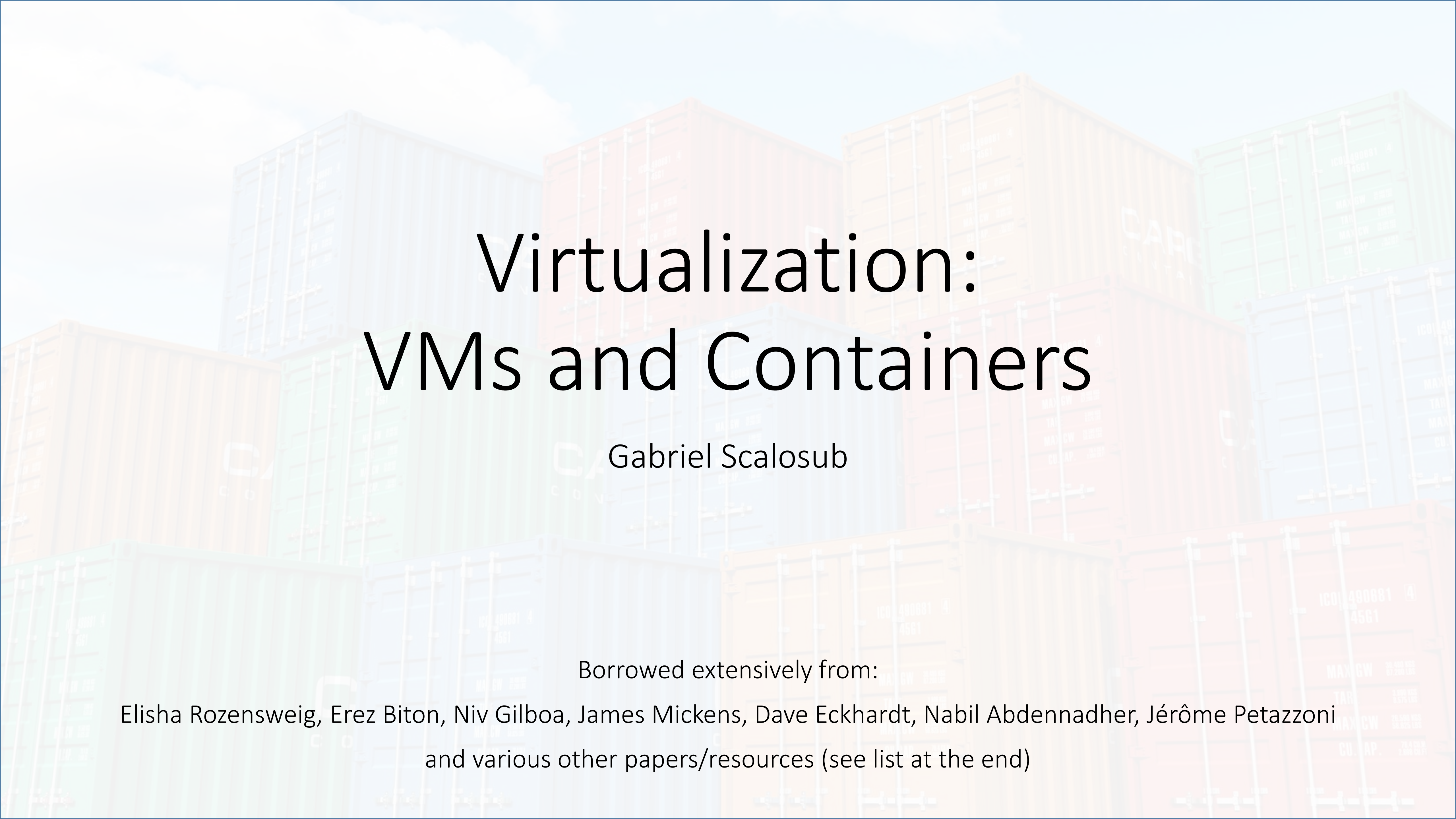


Virtualization: VMs and Containers

Gabriel Scalosub

Borrowed extensively from:

Elisha Rozensweig, Erez Biton, Niv Gilboa, James Mickens, Dave Eckhardt, Nabil Abdennadher, Jérôme Petazzoni

and various other papers/resources (see list at the end)

Outline

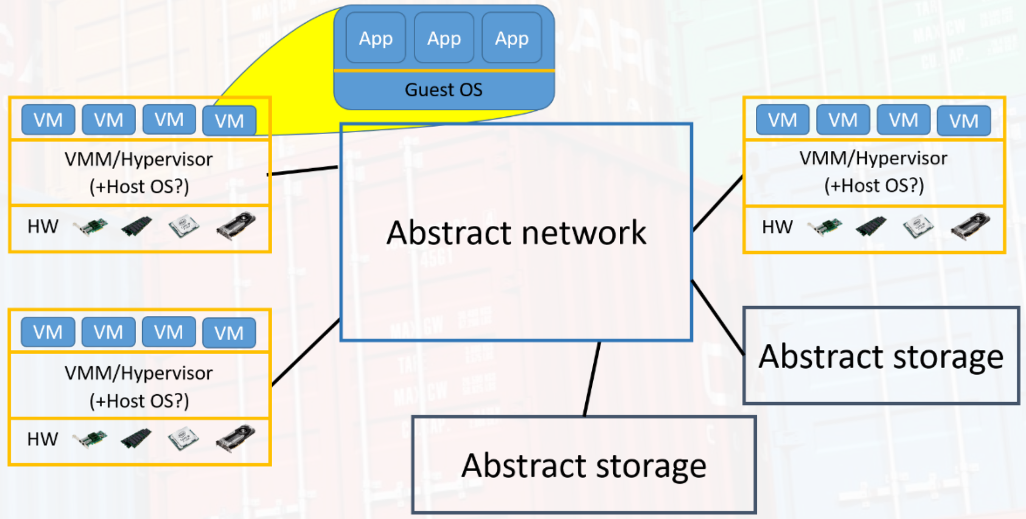
- (More) in-depth virtualization
- Containers
  - Docker
- Additional Virtualized frameworks
  - Unikernels

Outline

- (More) in-depth virtualization
- Containers
  - Docker
- Additional Virtualized frameworks
  - Unikernels

Recap: Virtualization (Previous Lecture)

- Raison d’être:
  - Distributed computing without dependency on physical resources
- What are we virtualizing?
  - Compute, storage, network
- (Some) Terminology:
  - **Host**: physical host, w/w-out host OS
  - **Guest**: user wishing to perform computation on a machine
  - **Virtual Machine (VM)**: complete guest environment (with OS) running on a host
  - **Hypervisor / VMM** (Virtual Machine Monitor): SW module managing virtualization
  - **Tenant**: user wishing to perform computation in the system (cluster of resources)
  - **Virtual CPU** (vCPU): VM CPU state, maintained by the VMM



Recap: Virtualization (Previous Lecture)

- "Formal" definition by Popek & Goldberg (1974)
- Efficient virtualization should satisfy:
  - Equivalence/Fidelity: VMM ≡ HW for any VM
  - Safety/Resource control : VMM controls all virtualized resources
  - Performance/Efficiency: major portion of VM instructions are executed natively
    - I.e., without VMM intervention
- Is virtualization possible?
  - Depends on Instruction Set Architecture (ISA)
  - Various techniques for satisfying the requirements (stay tuned…)
- Efficiency: different techniques have different pros/cons
  - No silver bullet

Virtualization: Emulation by Interpretation

- Every instruction issued by a guest OS VM:
  - Captured by the interpreter
  - Simulated (in SW) by the interpreter on behalf of guest OS
    - Maintains entire state for each guest OS
  - Read-decode-execute loop
- Benefits:
  - Isolation among guest OSs
    - Single interpreter simulating everything
  - Running ISAx guest OS on top of ISAy HW
- Main handicap: Slow!!! (sometimes it's good…)
- Examples:
  - BOCHS emulator, native QEMU (Quick EMUlator)

Source: https://c64emulator.111mb.de/

```
    **** COMMODORE 64 BASIC V2 ****
 64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
PRINT "HELLO WORLD"
HELLO WORLD

READY.
```

```
while(1){
      curr_instr = fetch(virtHw.PC);
      virtHw.PC += 4;
      switch(curr_instr){
            case ADD:
                  int sum = virtHw.regs[curr_instr.reg0] +
                            virtHw.regs[curr_instr.reg1];
                  virtHw.regs[curr_instr.reg0] = sum;
                  break;
            case SUB:
                  //...etc...
```

bochs
think inside the bochs.

Virtualization: Emulation by Binary Translation

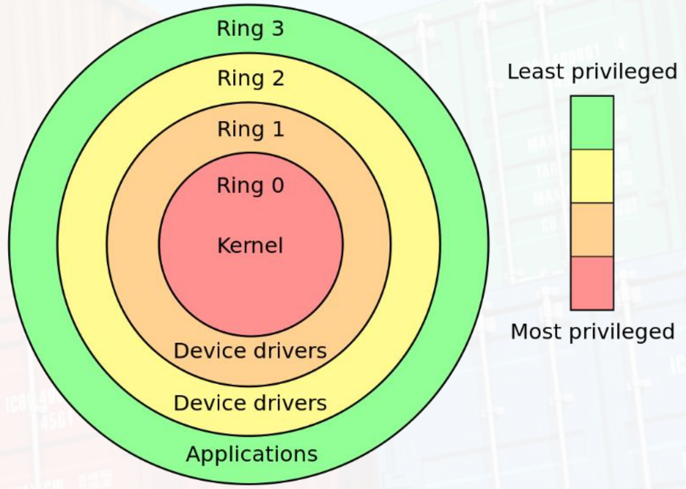
- Every instruction issued by a guest OS VM
  - Captured by the emulator
  - Translated to (a set of) instructions to be run on the real CPU
- BT vs Interpretation:
  - Both allow running ISAx guest OS on top of ISAy HW
  - BT has far less overhead -> much more efficient
- Do we ever need BT for guest OS-HW sharing same ISA?
  - E.g., why not issue the same instruction directly??
  - Depends on what we require from “virtualization”… (TBD shortly)

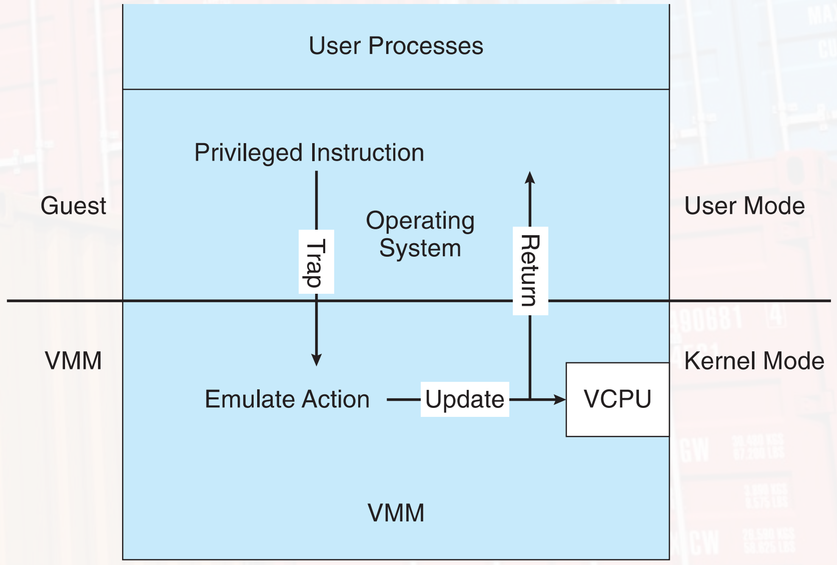
Virtualization: Trap-and-Emulate

- Maybe it is not required to capture all instructions?
  - Should we capture instructions like: **`add eax, ebx`** ?
- Capture only “sensitive” instructions
  - Required for isolation
  - Required for deceiving guest OS to think it runs directly on HW:
    - Guest OS runs in user-space, i.e., unprivileged mode, but
    - Thinks it’s running on the actual hardware, i.e., thinks it is privileged
    - How can we outwit the guest OS?

Virtualization: Trap-and-Emulate

- Privilege rings (execution mode)
  - Entire VM runs in user-mode – ring 3
    - Guest OS thinks it runs in kernel-mode – ring 0
  - VMM/hypervisor actually runs in kernel-mode – ring 0
- VM issues a non-privileged instruction
  - VMM: “Let it through!”
    - E.g., guest user-mode instruction
- VM issues a privileged instruction
  - CPU traps privileged instruction
    - E.g., some guest kernel-mode instruction
  - Control is passed on to VMM handler for emulating instruction (and updating vCPU)
  - Once completed, continue with next instruction





Source: Silberschatz et al.

Popek & Goldberg (1974)

- Efficient virtualization should satisfy:
  - Equivalence/Fidelity: of VMM to HW for any VM
  - Safety/Resource control : VMM controls all virtualized resources, VMs isolated
  - Performance/Efficiency: major portion of VM instructions are executed natively
    - I.e., without VMM intervention
- Types of CPU instructions:
  - Sensitive: depend/change HW configuration
    - HW configuration: registers, page tables, etc.
    - Reveal execution mode information (kernel/user)
  - Privileged: trap in user-mode, but don’t trap in kernel-mode
- Theorem: Efficient VMM is possible (for some instruction set) if every sensitive instruction is privileged.

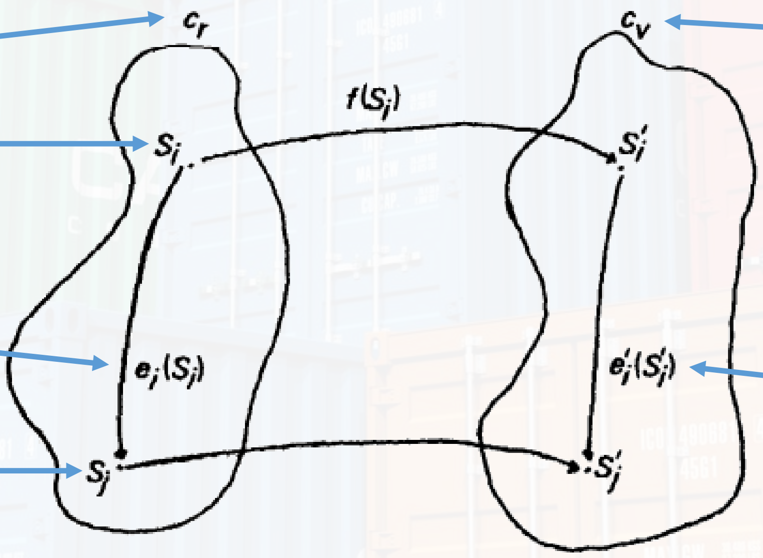
Popek & Goldberg (1974)

- Proof idea:
  - Efficient virtualization $\equiv$ Existence of Isomorphism
- Isomorphism between graph $G = (V, E)$ and $G' = (V', E')$
  - Mapping $\phi: V \mapsto V'$ satisfying
    $$(u, w) \in E \iff \big(\phi(u), \phi(w)\big) \in E'$$
  - In our case:
    - $V = C_r$: state space of real machine running on HW
    - $V' = C_v$: state space of VM running on top of VMM
    - State: memory and runtime system configuration (registers, tables, …)
    - E: instruction sequences on real machine causing transition from one state to another
    - E': instruction sequences of VM on top of VMM causing transition from one state to another
  - Note: suffices to consider only single instructions in E
    - Can always concatenate sequences in E'

Popek & Goldberg (1974)

- Proof idea:
  - Efficient virtualization ≡ Existence of Isomorphism

Fig. 2. The virtual machine map.



C_r
C_v
$f(S_i)$
S_i
S_i'
$e_i(S_i)$
$e_i'(S_i')$
S_j
S_j'
State space of real machine
Some state
Instruction on real machine
Some next state
State space of VM running on top of VMM
Instruction sequence e' maintaining isomorphism: emulating action e by VMM

Source: Popek & Goldberg

- $f: C_r \mapsto C_v$: Mapping of real machine state to VM state on top of VMM
- $e': C_v \mapsto C_v$: Isomorphism-maintaining mapping
  - Emulating actions in real machine space

Popek & Goldberg (1974)

Which Popek-Goldberg requirement is violated?
- Equivalence?
- Safety?
- Performance?

- Not every sensitive x86 instruction is privileged
  - Actually, quite a few such sensitive, unprivileged, instructions

only lower 22 bits of EFLAGS are used
0000 0000 0011 1111 1111 11**0**1 1111 1111

stack pointer

status register containing current state of processor

logic AND
value stored in right-operand

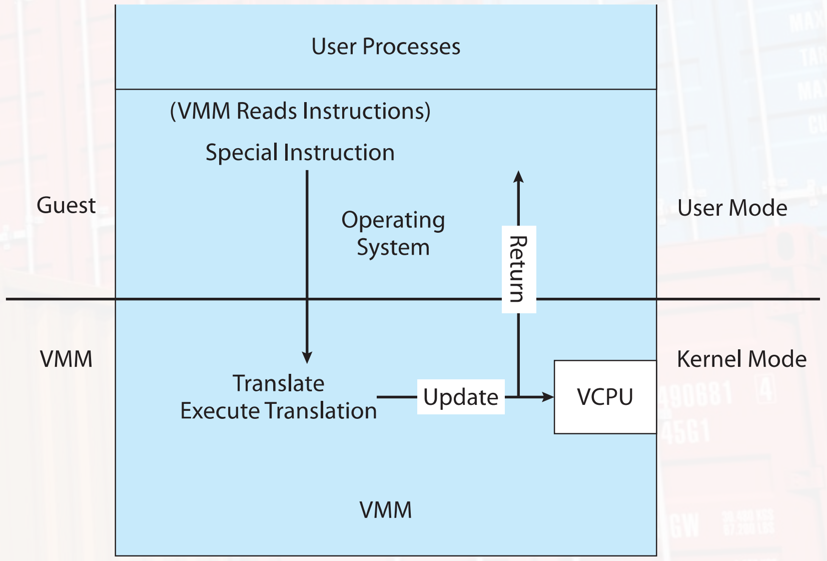
```
PUSHF                          # push %EFLAGS  onto stack
ANDL  $0x003FFDFF,  (%ESP)     # clear IF (interrupt flag) on stack
POPF                           # retrieve %EFLAGS  from stack
```

9th bit in EFLAGS

- Example:
  - POPF
    - If executed in kernel mode (ring 0):
      - Interrupts are off
      - E.g., so that kernel code doesn't get interrupted by a some interrupt handler
    - If executed in user mode (ring > 0)
      - Ignored by CPU… (that's what x86 does ☹)

Virtualization: Binary Translation (Again)

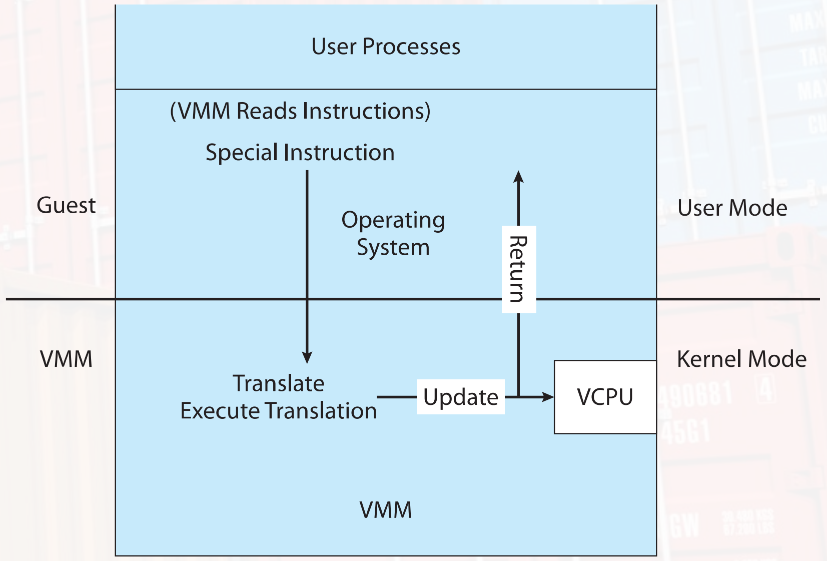
- Dynamic binary translation
  - VMM "Captures" sensitive guest OS instruction on the fly
    - Translates instruction, and executes translation
    - Updates vCPU, and returns execution to Guest OS
  - Non-sensitive instructions executed natively
- Optimizations of BT
  - Consider instructions in blocks
  - Cache translated instructions
- Static binary translation
  - Perform translation without actually running code
    - Done statically on the source code executable ("compilation"-like)
  - Hard to deal with branches which are only known in run-time



Source: Silberschatz et al.

Virtualization: Binary Translation (Again)

- Compared to emulation by interpretation:
  - Avoids the read-decode-execute loop per instruction
  - “Translate-once / use-multiple-times”
    - Large translation overhead (of instructions block)
    - Amortized by negligible reuse overhead
      - Caching of common blocks
- Workhorse of virtualization
  - VMware

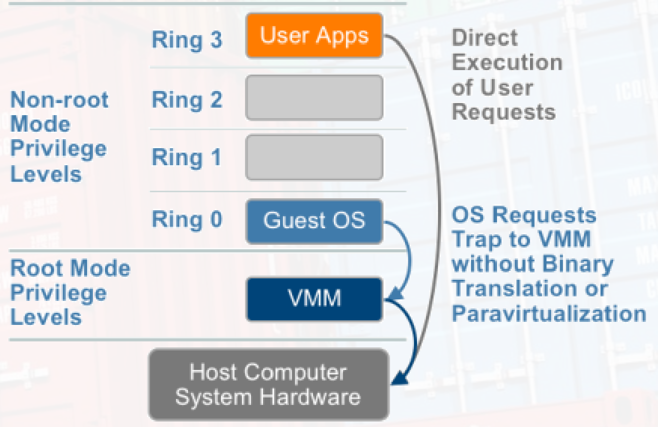


Source: Silberschatz et al.

Virtualization: Hardware Assisted Virtualization

- Hardware and ISA extensions that allow for virtualization
  - New instructions / abilities
- Key features
  - Introduces additional execution modes
    - Host-mode (or “root-mode”) for VMM, ring -1
      - beyond “standard” user/kernel- modes (Guest OS in guest-mode, ring 0)
  - Make sure to trap sensitive, previously unprivileged, instructions
    - vCPU state-awareness by the processor
  - Extended page tables (EPT): Memory Management Unit (MMU) support for virtualization
    - MMU aware of Guest-virtual mem. -> Guest-”physical” mem. -> Host-physical mem.
- Examples of HW providing such virtualization (mid 2000s)
  - Intel VT (Virtualization Technology), AMD-V
- Hypervisors use these frameworks (enhanced instruction sets)
  - VMware Fusion, Parallel Desktop
    - E.g., running Windows (or other OS) within Mac on Intel platforms

Source: VMware


Ring 3
User Apps
Direct Execution of User Requests
Non-root Mode Privilege Levels
Ring 2
Ring 1
Ring 0
Guest OS
OS Requests Trap to VMM without Binary Translation or Paravirtualization
Root Mode Privilege Levels
VMM
Host Computer System Hardware

Virtualization: Para-virtualization

- Guest OS is aware(!!) that it is being virtualized
  - Hypervisor/VMM offers special API for handling sensitive guest OS instructions
  - Hypercall: modified sensitive instruction, directly forwarded to hypervisor
  - Guest OS source code is recompiled and modified
    - Uses hypercalls
    - Does not run OS “as-is”
  - No trap-end-emulate
- E.g., Xen
  - Linux can be recompiled to run on Xen
    - Can the same be done for MS Windows?

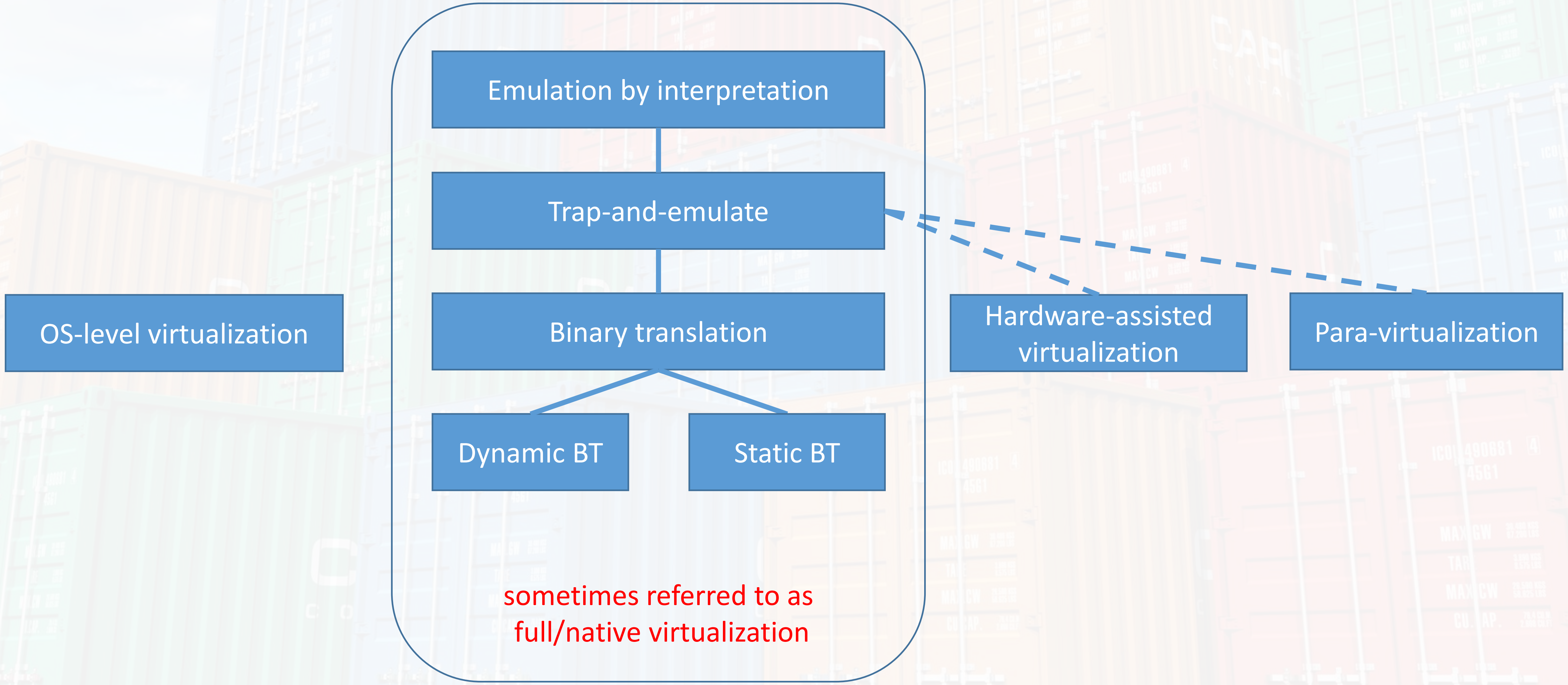
OS-level Virtualization

Jailbreaking anyone?

- Isolation of environments is provided by the host-OS
  - Kernel dependent!
    - Guests share all kernel functionalities, sometimes require a modified kernel
- Examples
  - Jails:
    - chroot jail: process has a modified root directory (restricted file-system view)
    - FreeBSD jail: virtual environment with isolated files, processes, users, IP address
  - Containers (generally, more flexible than jails):
    - Main tools:
      - cgroups (control groups): grouping processes into collections that share parameters/constraints
      - namespaces: arranges resources (pid, mnt, net, user, …) so visible only within namespace
    - Examples
      - Linux containers (LXC), OpenVZ (Virtuozzo), Docker containers… (stay tuned)

EUROPE'S ECLIPSING SENSATION
HOUDINI
THE WORLD'S HANDCUFF KING & PRISON BREAKER
NOTHING ON EARTH CAN HOLD HOUDINI A PRISONER

(Very) Informal Map of Virtualization


Emulation by interpretation
Trap-and-emulate
Binary translation
Dynamic BT
Static BT
OS-level virtualization
Hardware-assisted virtualization
Para-virtualization
sometimes referred to as full/native virtualization

VMMs/Hypervisors Types

- Type 1 / bare-metal / native
  - VMM runs on bare metal
    - “replaces” host OS
  - More efficient
  - Considered more secure
  - Examples: VMWare ESX/ESXi, MS Hyper-V, KVM, Xen
- Type 2 / hosted
  - Hosted VMM, on top of host OS
  - Host OS provides device drivers support
  - As simple as installing an application
  - Not as efficient as Type 1
  - Examples: Oracle VirtualBox, VMWare Workstation, Parallels, QEMU

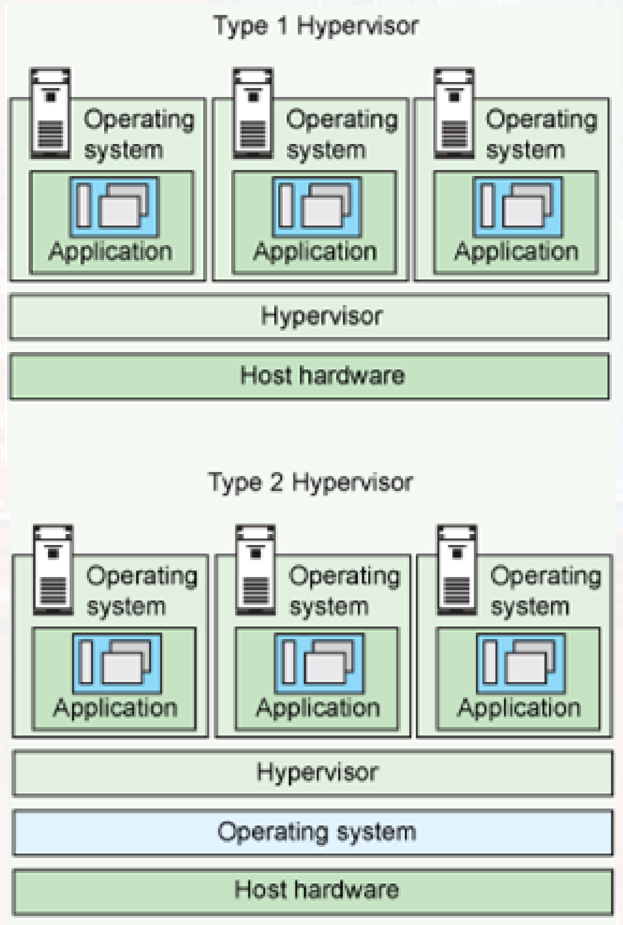


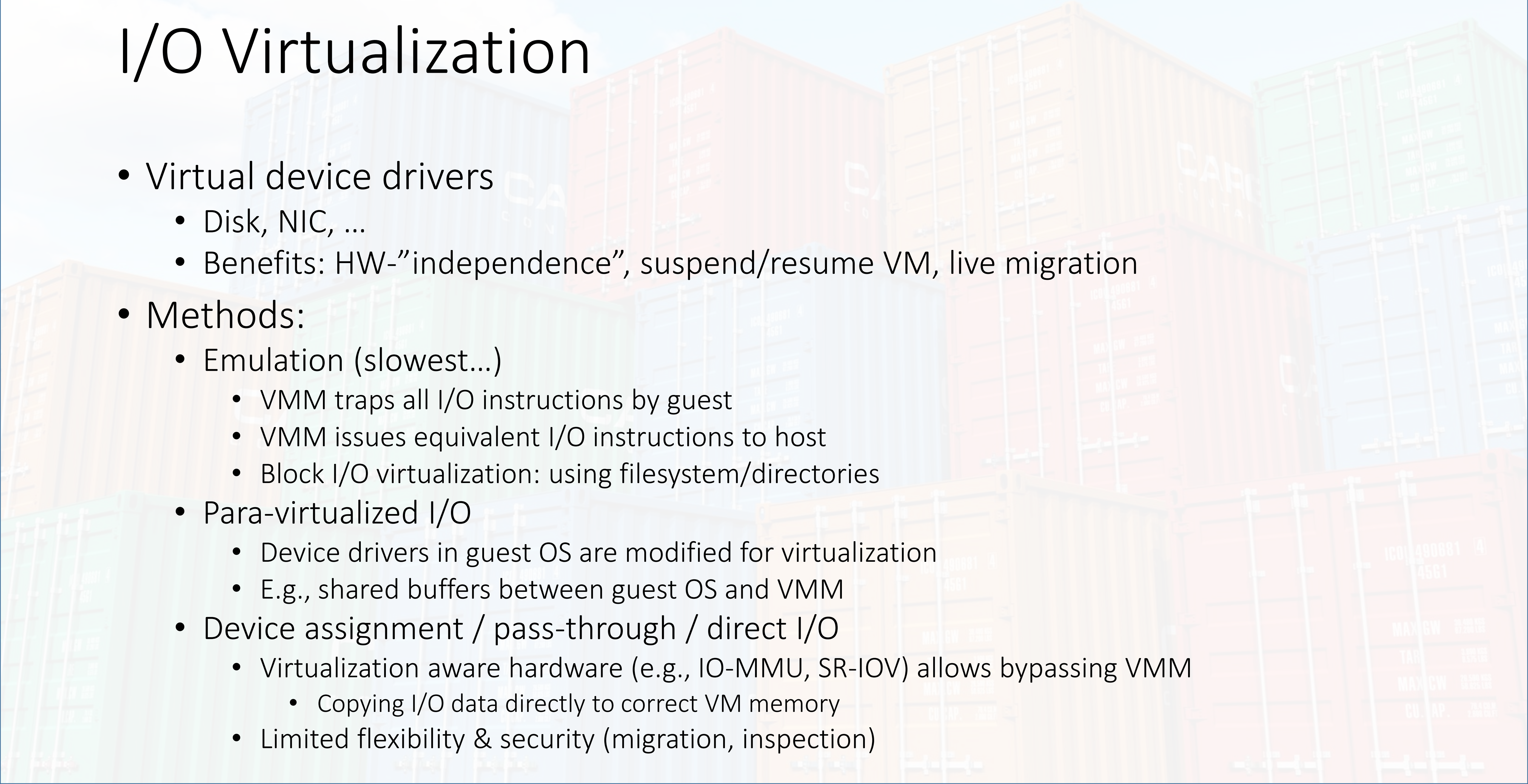
Image: IBM

Memory Virtualization

- Virtual memory maps to physical memory (via MMU)
  - Guest OS: guest virtual -> guest “physical”
  - VMM: guest “physical” -> host physical

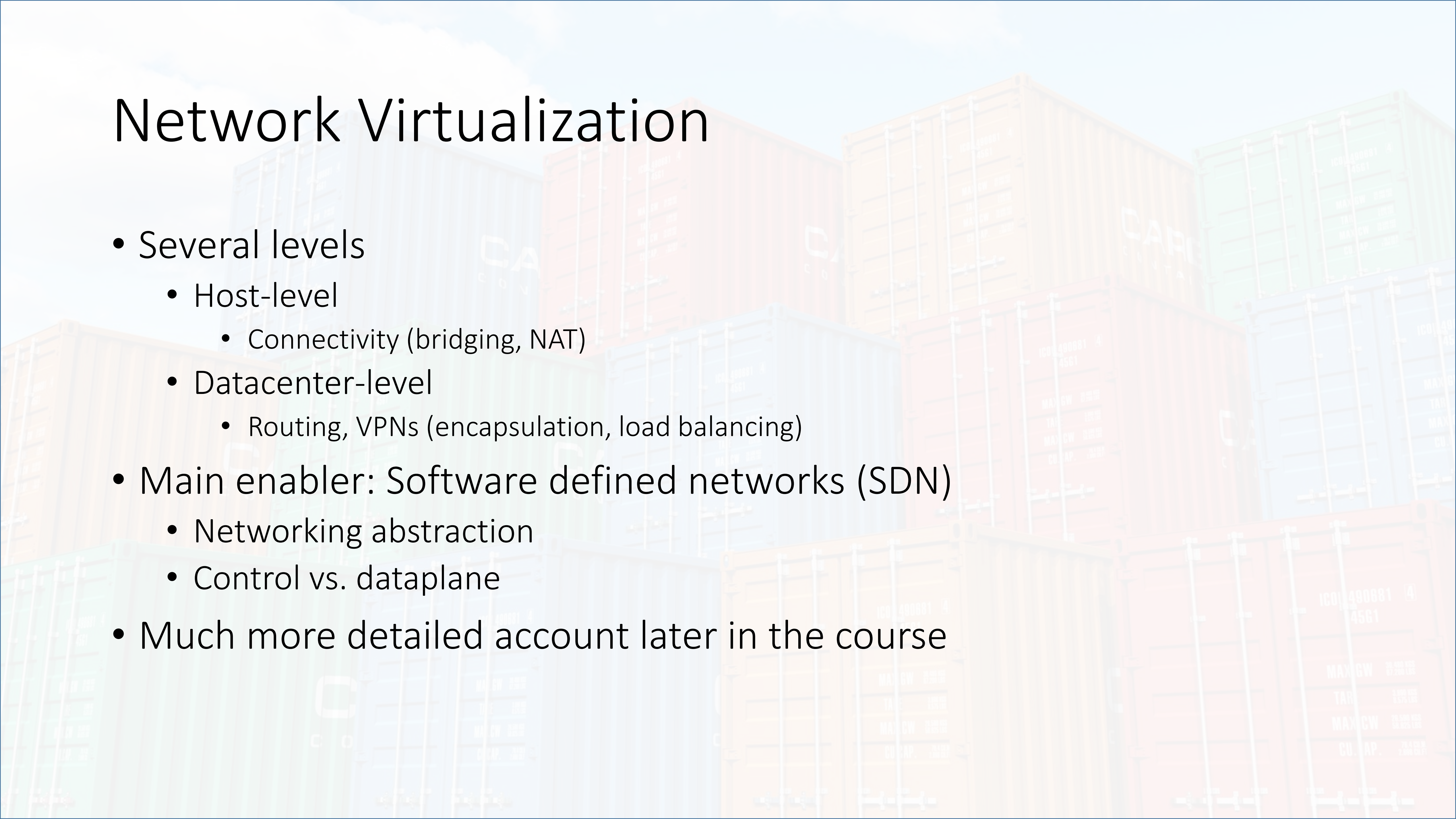
Memory Management Unit

- Shadow page tables
  - Managed by VMM
  - Copied onto guest MMU
    - When required
- Nested/Extended page tables
  - Managed directly by HW
  - In, e.g., Hardware-Assisted Virtualization



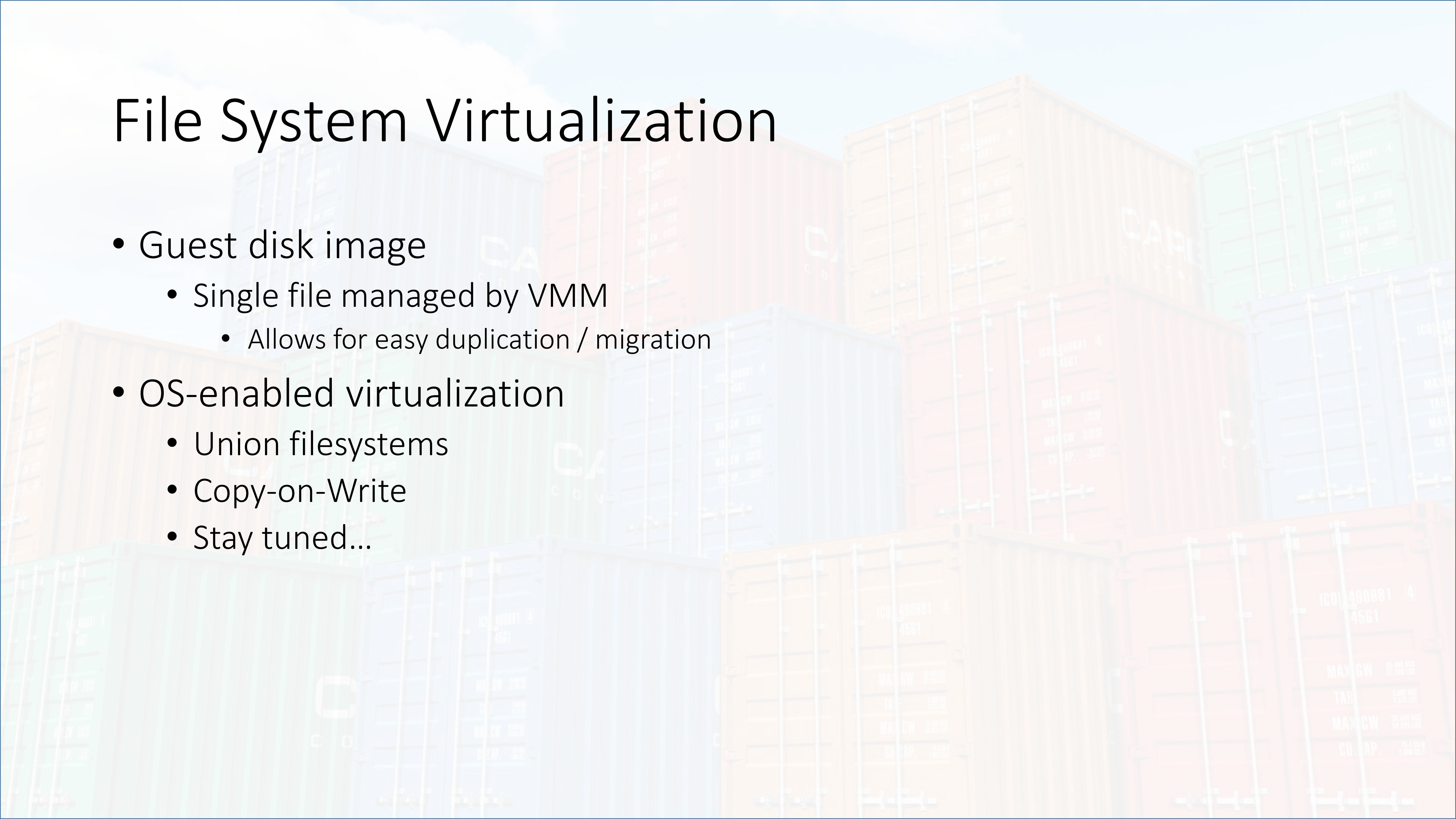
I/O Virtualization

- Virtual device drivers
  - Disk, NIC, …
  - Benefits: HW-”independence”, suspend/resume VM, live migration
- Methods:
  - Emulation (slowest…)
    - VMM traps all I/O instructions by guest
    - VMM issues equivalent I/O instructions to host
    - Block I/O virtualization: using filesystem/directories
  - Para-virtualized I/O
    - Device drivers in guest OS are modified for virtualization
    - E.g., shared buffers between guest OS and VMM
  - Device assignment / pass-through / direct I/O
    - Virtualization aware hardware (e.g., IO-MMU, SR-IOV) allows bypassing VMM
      - Copying I/O data directly to correct VM memory
    - Limited flexibility & security (migration, inspection)



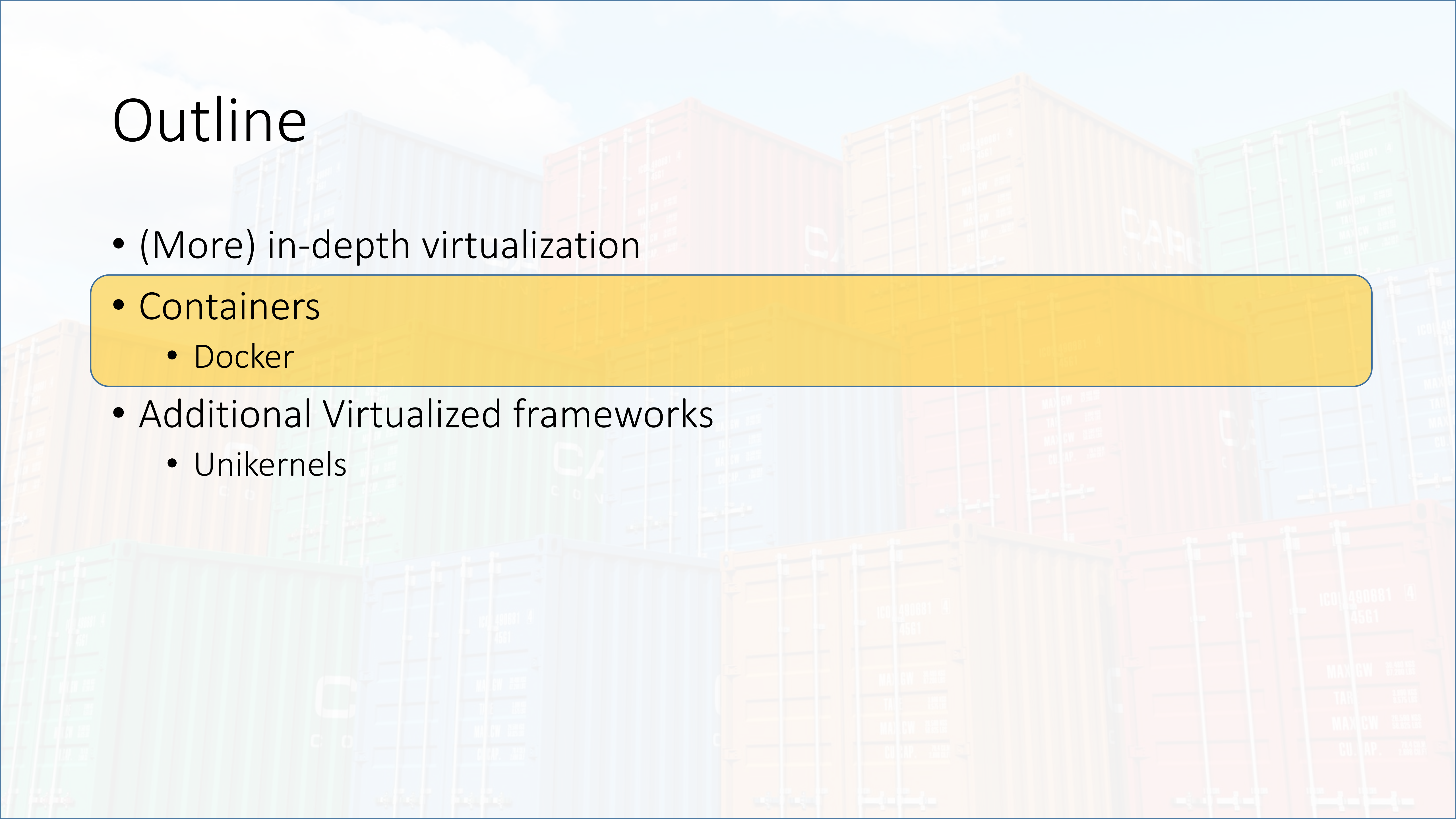
Network Virtualization

- Several levels
  - Host-level
    - Connectivity (bridging, NAT)
  - Datacenter-level
    - Routing, VPNs (encapsulation, load balancing)
- Main enabler: Software defined networks (SDN)
  - Networking abstraction
  - Control vs. dataplane
- Much more detailed account later in the course



File System Virtualization

- Guest disk image
  - Single file managed by VMM
    - Allows for easy duplication / migration
- OS-enabled virtualization
  - Union filesystems
  - Copy-on-Write
  - Stay tuned…



Outline

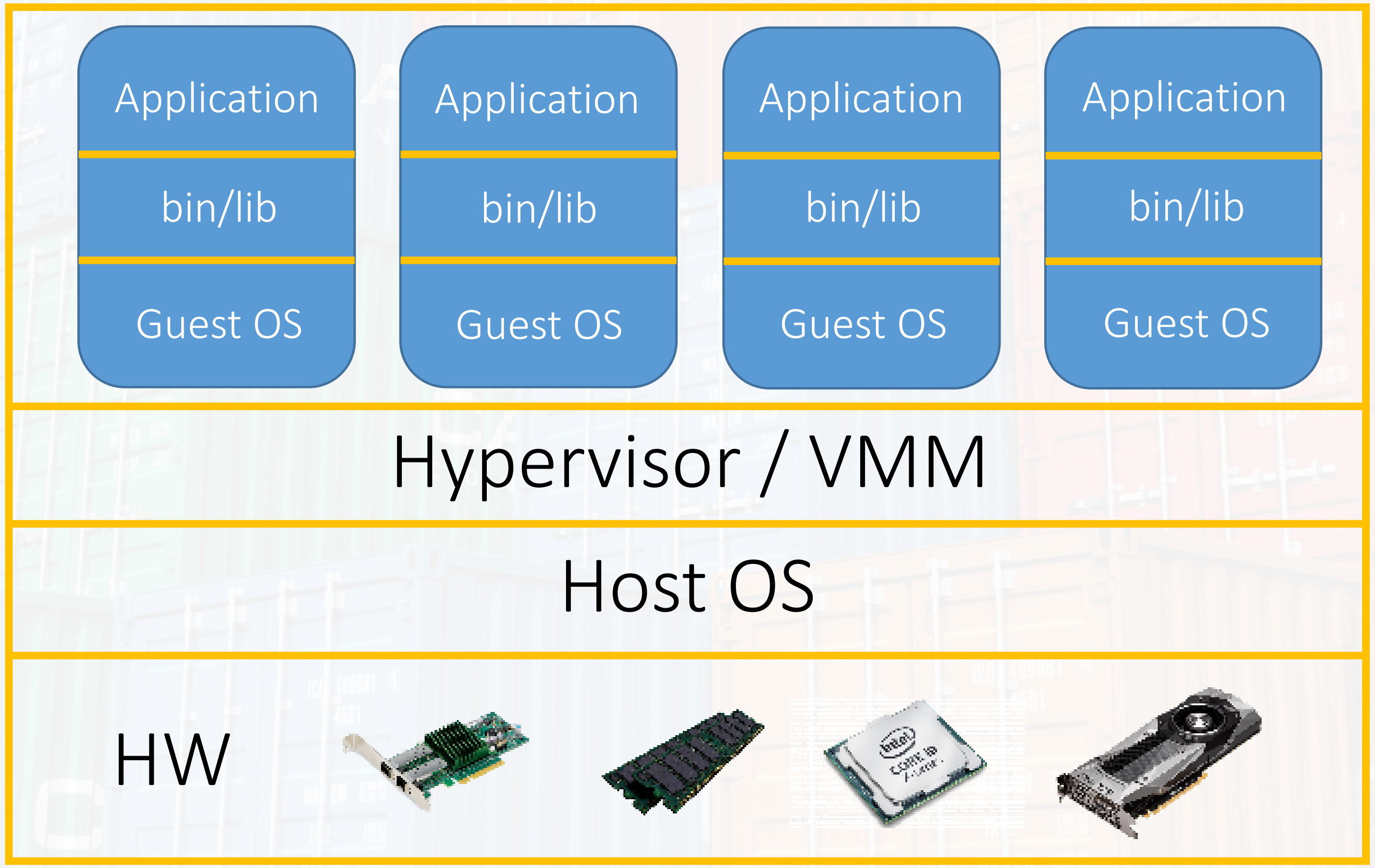
- (More) in-depth virtualization
- Containers
  - Docker
- Additional Virtualized frameworks
  - Unikernels

Introduction to Containers

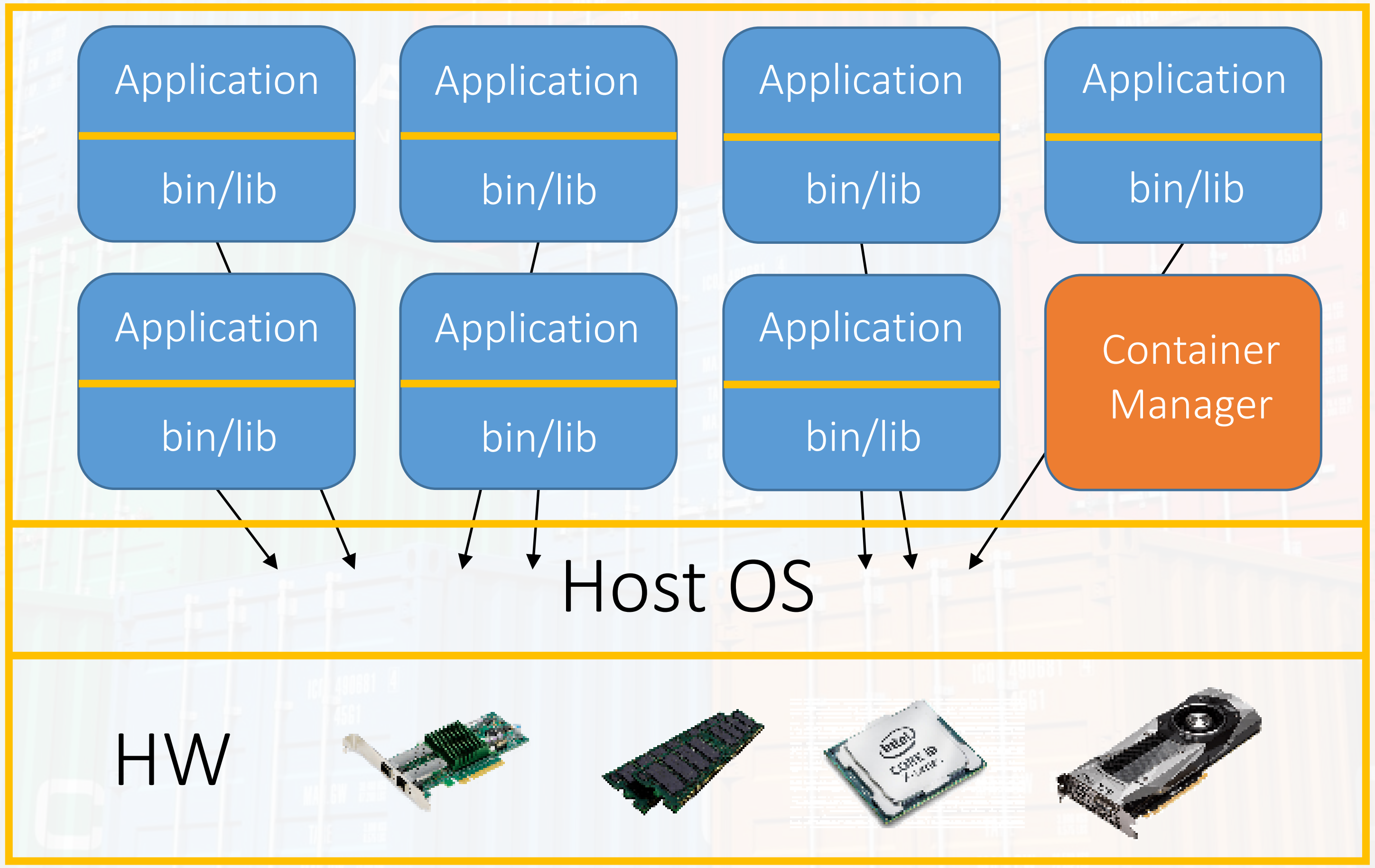
- A single package covering a complete runtime environment
  - Application
    - Dependencies, libraries, binaries, config files
  - Minimal
    - "Take only what you need to survive"
- Reliable operation of application
  - Across computing environments
  - Development -> Deployment (physical & virtual)

TAKE ONLY WHAT YOU NEED TO SURVIVE
WHAT THE HELL IS THIS?
HOT AIR
ITS MY INDUSTRIAL SIZED HAIR DRYER
AND I CAN'T LIVE WITHOUT IT!!

From VMs to Containers



From VMs to Containers



Containers Examples

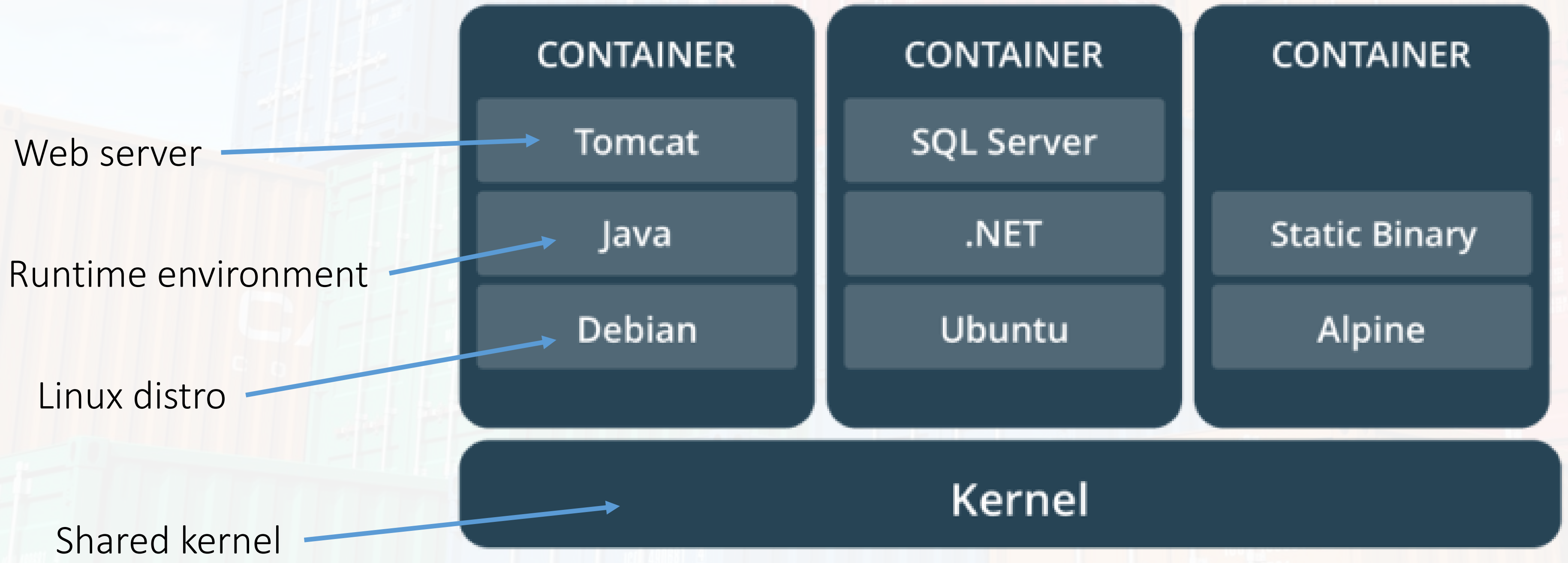


Image: Docker

Introduction to Docker

- Software development platform
  - Open Source, written in Go
  - Packaging applications in containers
- Main features
  - Portability, platform-independent
    - Assuming docker support is available
  - Reuse and Sharing
    - Modify any “template”
    - Public/private repositories
  - Simple interfaces
    - CLI, RESTful API
  - DevOps-oriented
    - Simplify process / reduce time from development to deployment

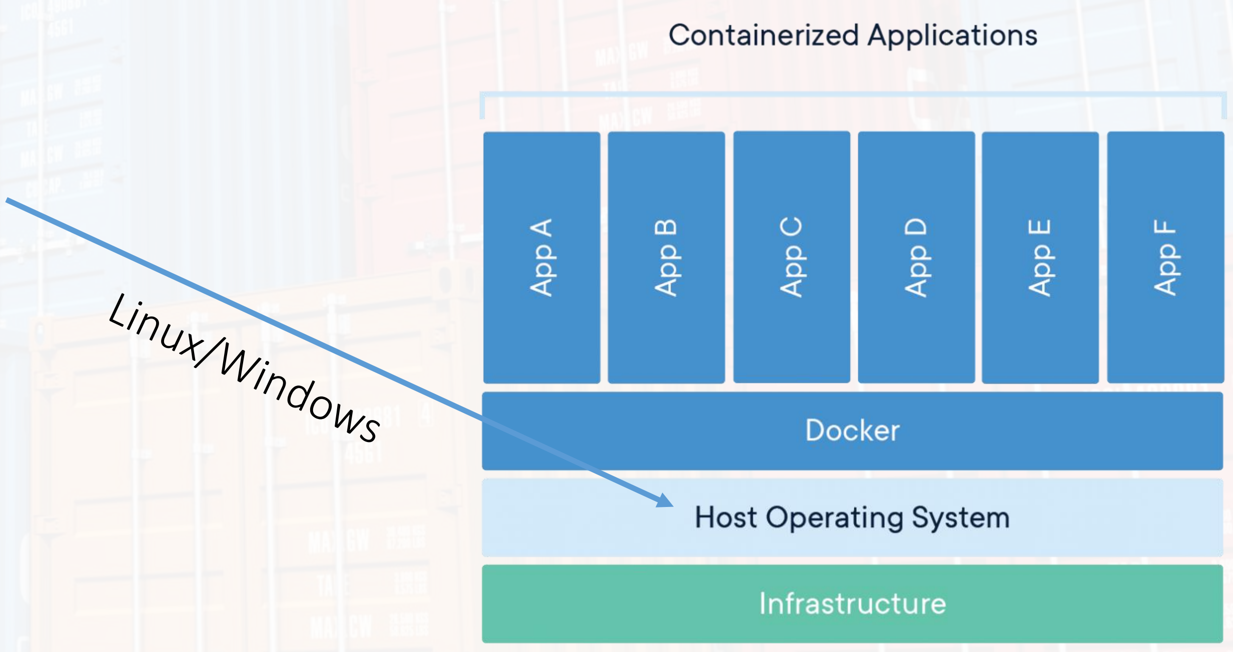


Image: Docker

Introduction to Docker: Architecture

- Docker engine
  - Docker deamon
  - Restful API
  - CLI
- Manages docker objects
  - Images (read-only, static definitions)
  - Containers (running images/applications)
  - Networks
  - Volumes

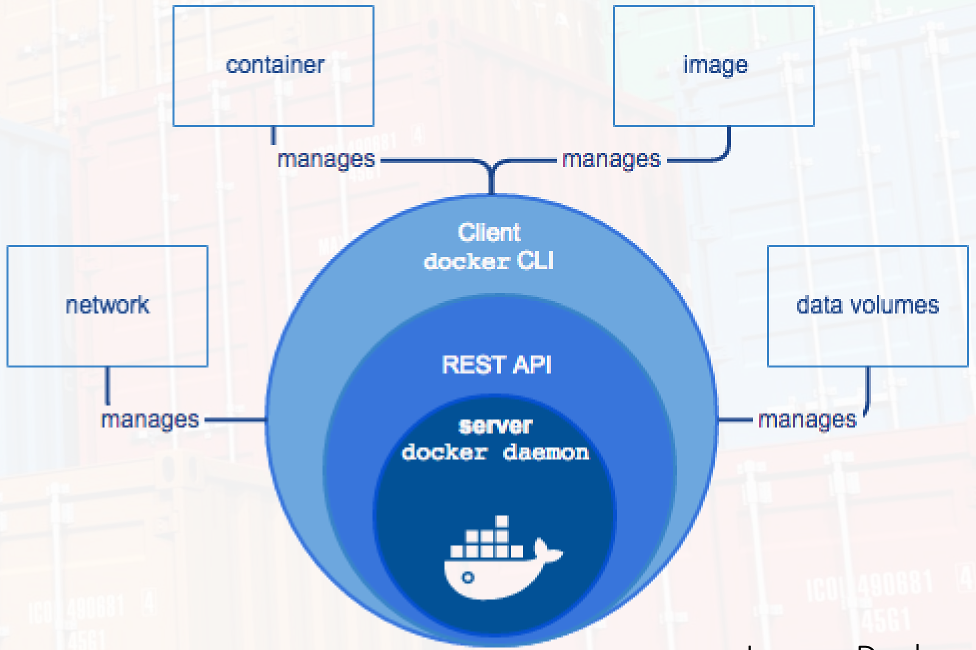


Image: Docker

Introduction to Docker: Architecture

- Docker daemon:
  - Performs building, running and distributing of containers
  - Listens to client requests via API
  - Inter-daemon communication
    - For managing services (stay tuned)
- Docker client:
  - User interaction interface with docker daemon (local/remote)
  - CLI-based, `docker` command
- Docker registry (e.g., Docker Hub):
  - Repository of docker images
    - Pull, run, push, build, etc. (follow the linestyles…)

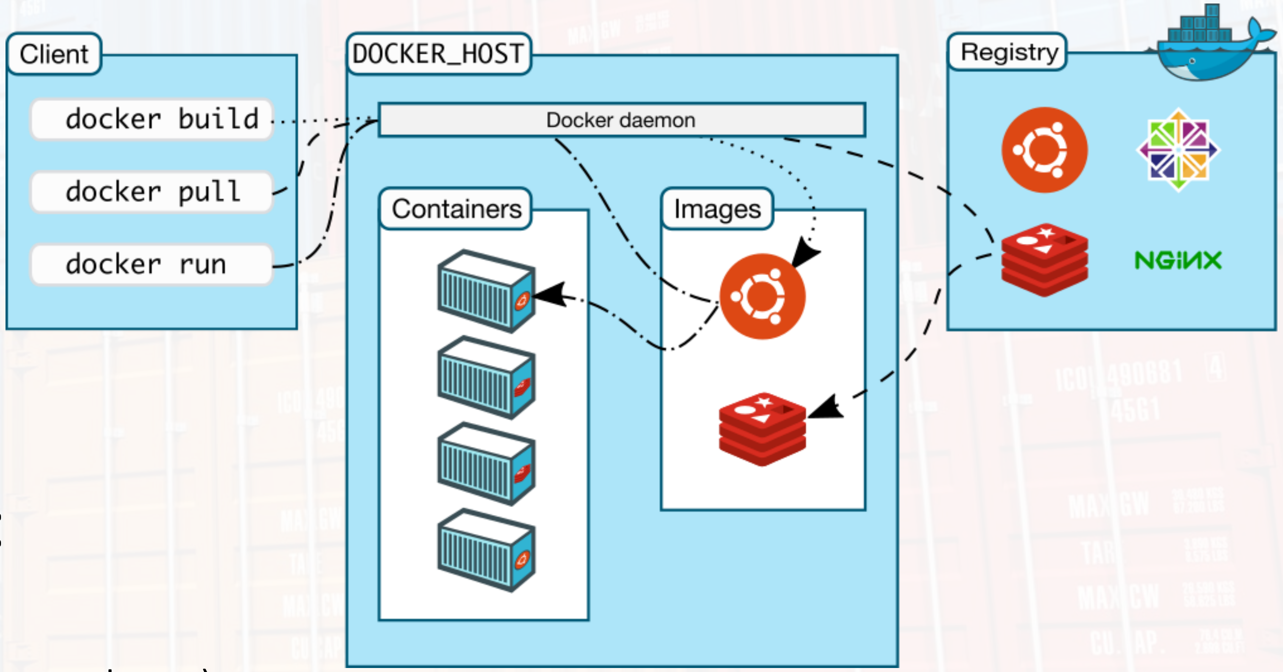
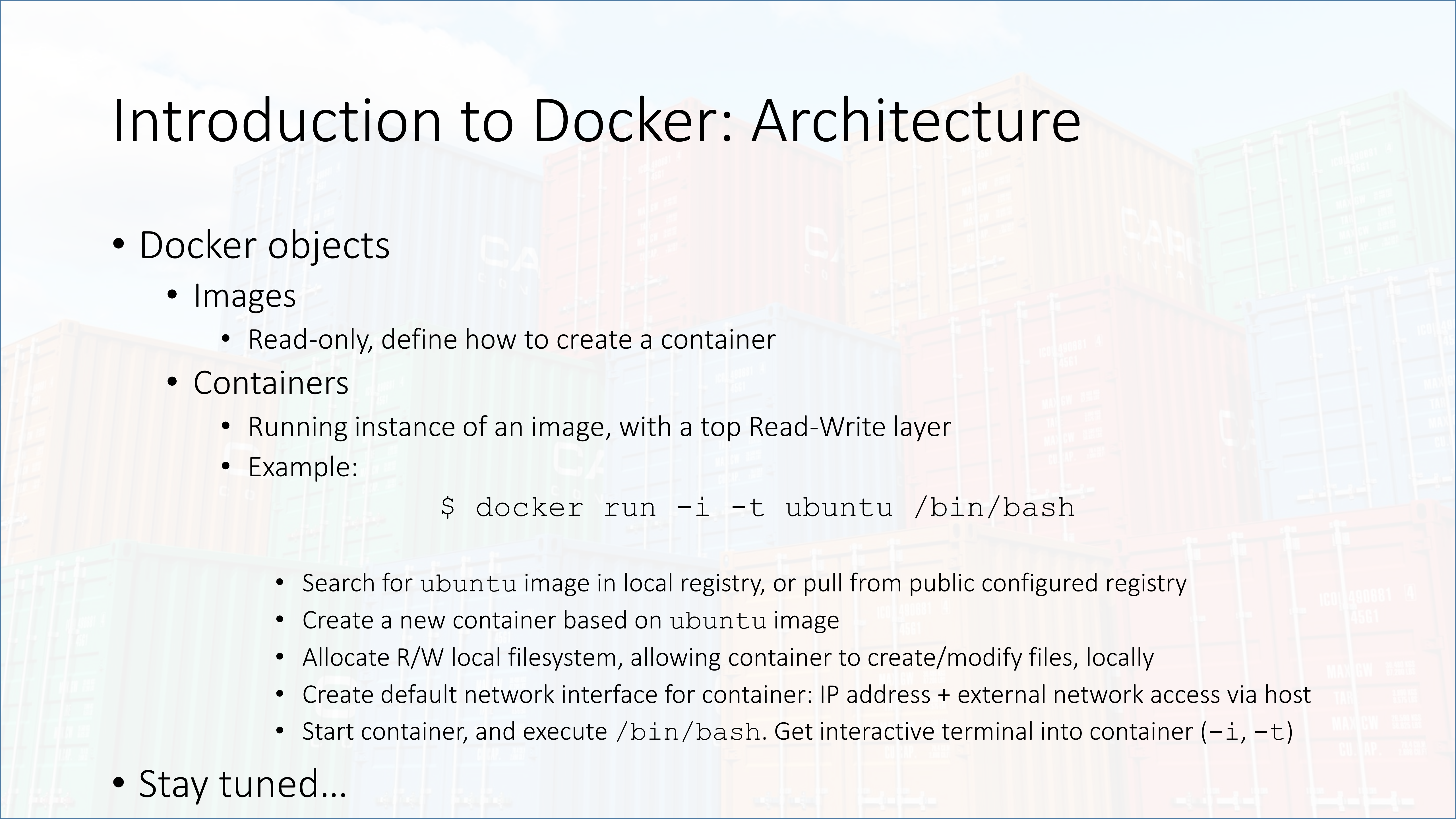


Image: Docker

Introduction to Docker: Architecture



- Docker objects
  - Images
    - Read-only, define how to create a container
  - Containers
    - Running instance of an image, with a top Read-Write layer
    - Example:

```
$ docker run -i -t ubuntu /bin/bash
```

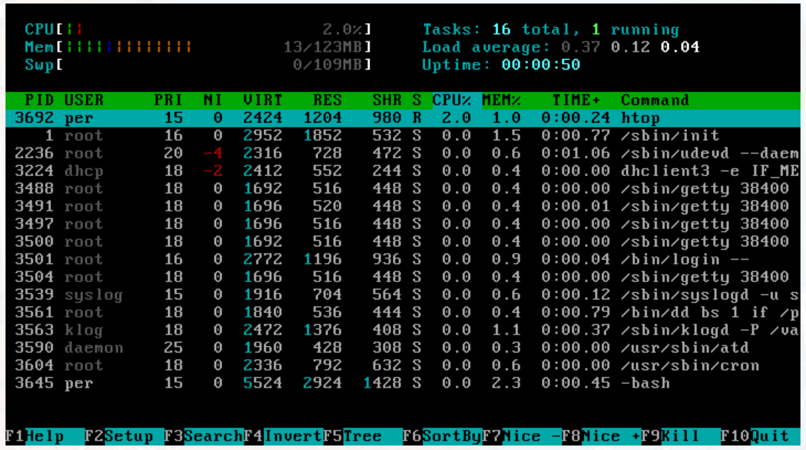
- Search for `ubuntu` image in local registry, or pull from public configured registry
- Create a new container based on `ubuntu` image
- Allocate R/W local filesystem, allowing container to create/modify files, locally
- Create default network interface for container: IP address + external network access via host
- Start container, and execute `/bin/bash`. Get interactive terminal into container (`–i`, `–t`)

- Stay tuned…

Containers Isolation

- Recap of processes in Linux
  - All processes are descendants of the `init` process (pid=1)
  - Tree hierarchy (`pstree -n -p`)
  - Processes inherit attributes of parents
- Container isolation uses:
  - cgroups
  - namespaces
- Examples:
  - LXC, Docker containers, RKT containers

Containers Isolation



```
  CPU[||                    2.0%]     Tasks: 16 total, 1 running
  Mem[||||||||||||||     13/123MB]     Load average: 0.37 0.12 0.04
  Swp[                   0/109MB]     Uptime: 00:00:50

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 3692 per        15   0  2424  1204   980 R  2.0  1.0  0:00.24 htop
    1 root       16   0  2952  1852   532 S  0.0  1.5  0:00.77 /sbin/init
 2236 root       20  -4  2316   728   472 S  0.0  0.6  0:01.06 /sbin/udevd --daem
 3224 dhcp       18  -2  2412   552   244 S  0.0  0.4  0:00.00 dhclient3 -e IF_ME
 3488 root       18   0  1692   516   448 S  0.0  0.4  0:00.00 /sbin/getty 38400
 3491 root       18   0  1696   520   448 S  0.0  0.4  0:00.01 /sbin/getty 38400
 3497 root       18   0  1696   516   448 S  0.0  0.4  0:00.00 /sbin/getty 38400
 3500 root       18   0  1692   516   448 S  0.0  0.4  0:00.00 /sbin/getty 38400
 3501 root       16   0  2772  1196   936 S  0.0  0.9  0:00.04 /bin/login --
 3504 root       18   0  1696   516   448 S  0.0  0.4  0:00.00 /sbin/getty 38400
 3539 syslog     15   0  1916   704   564 S  0.0  0.6  0:00.12 /sbin/syslogd -u s
 3561 root       18   0  1840   536   444 S  0.0  0.4  0:00.79 /bin/dd bs 1 if /p
 3563 klog       18   0  2472  1376   408 S  0.0  1.1  0:00.37 /sbin/klogd -P /va
 3590 daemon     25   0  1960   428   308 S  0.0  0.3  0:00.00 /usr/sbin/atd
 3604 root       18   0  2336   792   632 S  0.0  0.6  0:00.00 /usr/sbin/cron
 3645 per        15   0  5524  2924  1428 S  0.0  2.3  0:00.45 -bash

F1Help  F2Setup F3SearchF4InvertF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

- cgroups
  - What can a process use?
  - Per-subsystem (i.e., resource) hierarchy of processes
  - Subsystem: CPU, memory, network, disk
  - Restrict/prioritize/meter subsystem usage/access
    - Limits examples: weighted, hard-limit, …
  - Each process belongs to a single cgroup (per-subsystem)
    - Allows for complex combinations
    - Inheritance of restrictions
  - `nice` on steroids…
  - Demo: CPU restriction/prioritization using cgroups

Containers Isolation

- namespaces
  - What can a process see?

  - Local view of system
    - PID, mnt, net, uid,…
    - From within the container,
      it looks like we are the only ones there
      - “Virtual” PIDs
      - Isolated filesystem view (compared to host)
      - Isolated network from that on host
      - Root-like capabilities inside container, for a non-root host user
  - Demo: PID namespaces in Docker

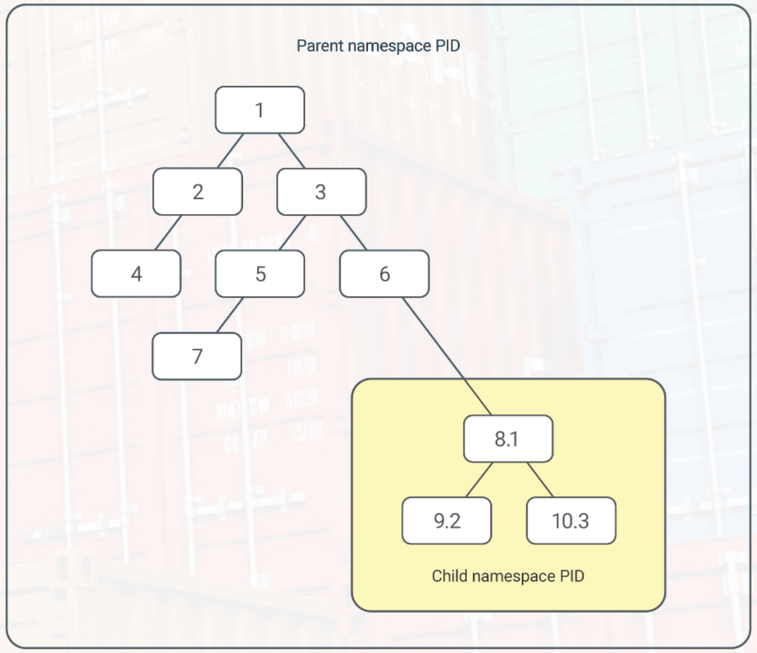
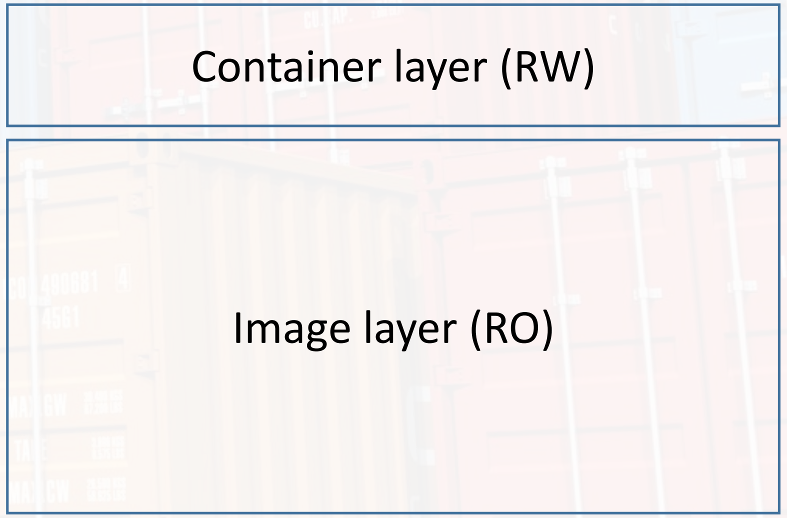


Image: selectel.com

Images and Containers

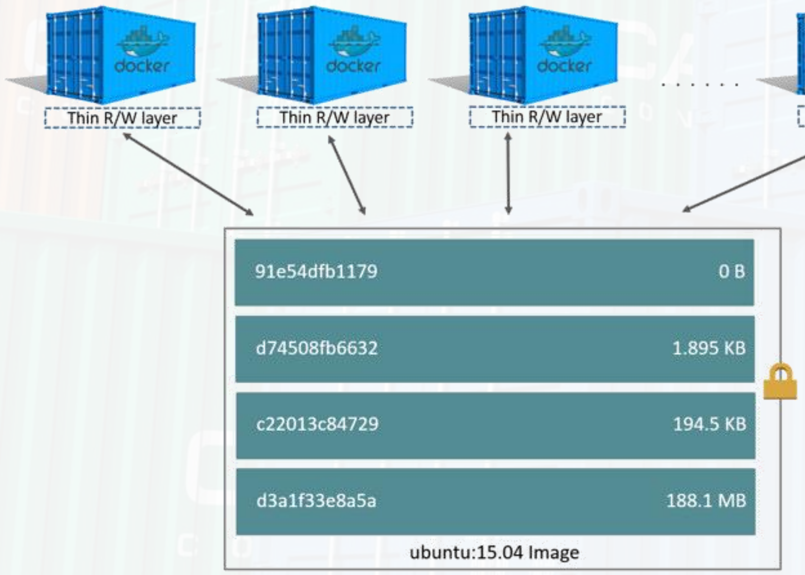
- Application image
  - Complete code (inc. runtime / bin) required for running application
- Container
  - Actual running instance of an application image

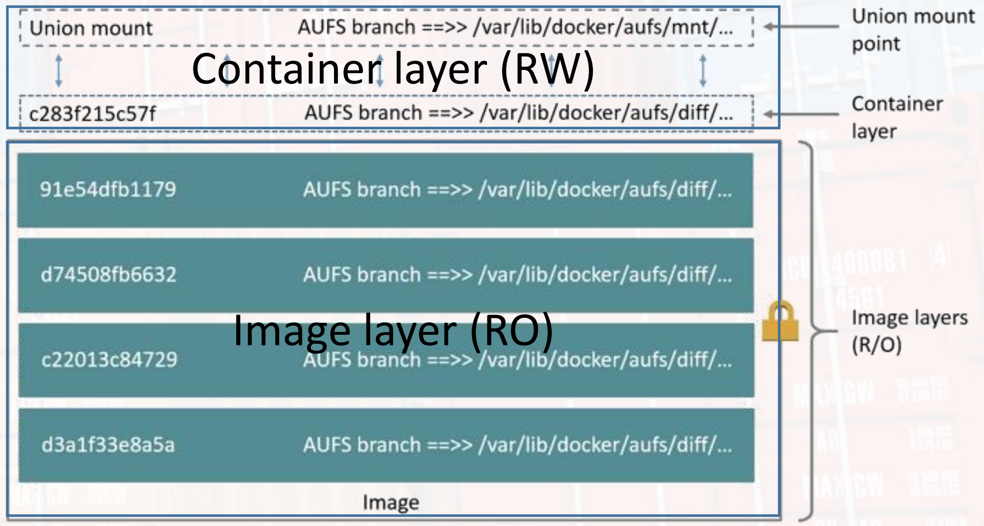


Images and Containers

- Layers
  - Sequence of files generated while docker builds an image using Dockerfile
  - RO Layers can be shared by various containers

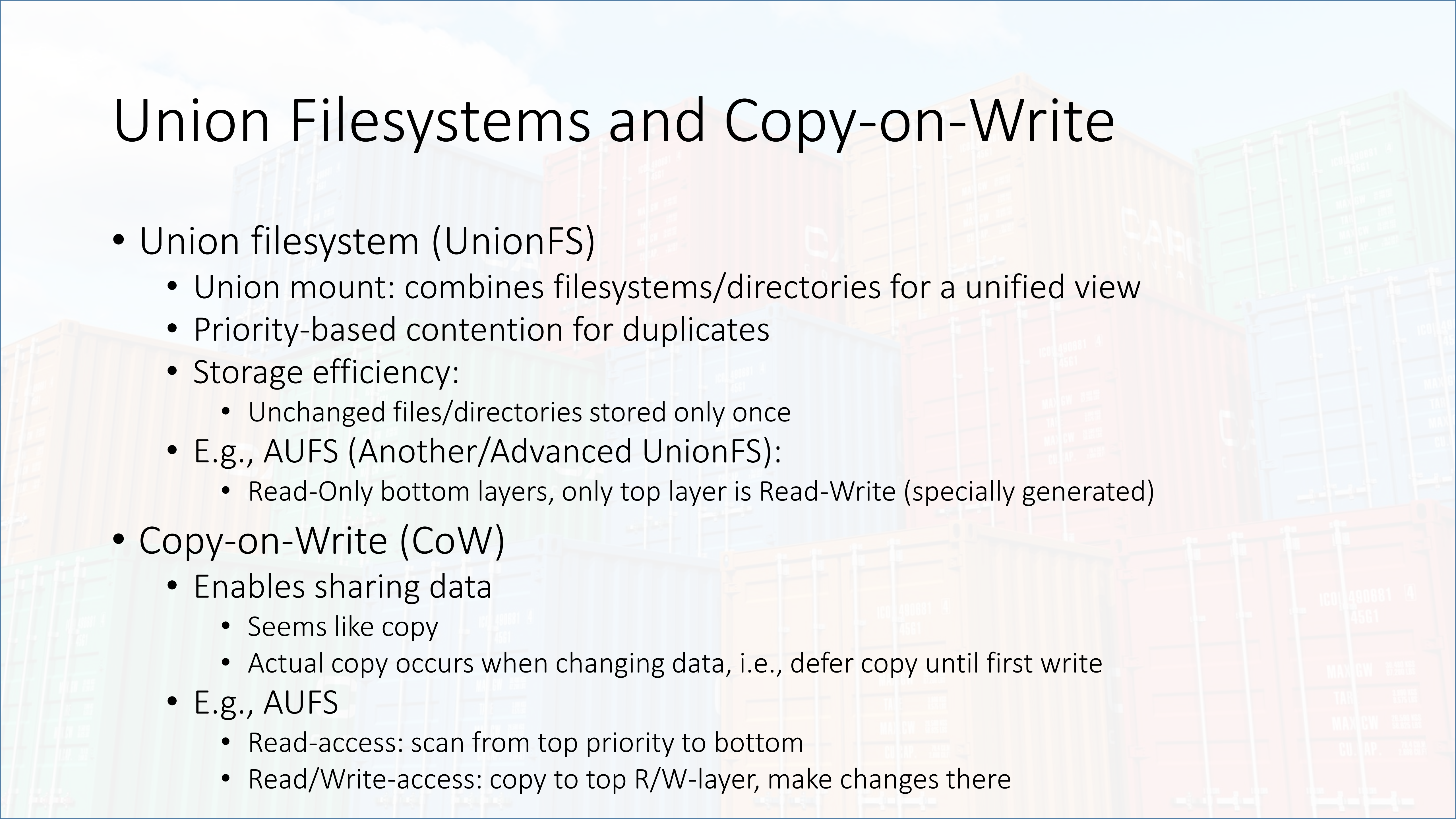
```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```





Union mount AUFS branch ==>> /var/lib/docker/aufs/mnt/...
Container layer (RW)
c283f215c57f AUFS branch ==>> /var/lib/docker/aufs/diff/...
91e54dfb1179 AUFS branch ==>> /var/lib/docker/aufs/diff/...
d74508fb6632 AUFS branch ==>> /var/lib/docker/aufs/diff/...
Image layer (RO)
c22013c84729 AUFS branch ==>> /var/lib/docker/aufs/diff/...
d3a1f33e8a5a AUFS branch ==>> /var/lib/docker/aufs/diff/...
Image
Union mount point
Container layer
Image layers (R/O)

Image: Docker

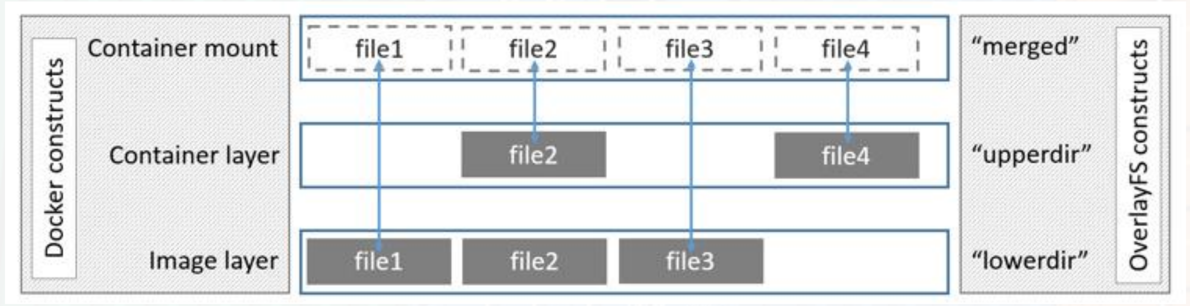


Union Filesystems and Copy-on-Write

- Union filesystem (UnionFS)
  - Union mount: combines filesystems/directories for a unified view
  - Priority-based contention for duplicates
  - Storage efficiency:
    - Unchanged files/directories stored only once
  - E.g., AUFS (Another/Advanced UnionFS):
    - Read-Only bottom layers, only top layer is Read-Write (specially generated)
- Copy-on-Write (CoW)
  - Enables sharing data
    - Seems like copy
    - Actual copy occurs when changing data, i.e., defer copy until first write
  - E.g., AUFS
    - Read-access: scan from top priority to bottom
    - Read/Write-access: copy to top R/W-layer, make changes there

Union Filesystems and Copy-on-Write

- UnionFS in Docker
  - Originally used AUFS, currently using overlay2
    - Image layer ≡ AUFS branch
  - Priority for duplicates: higher-layer



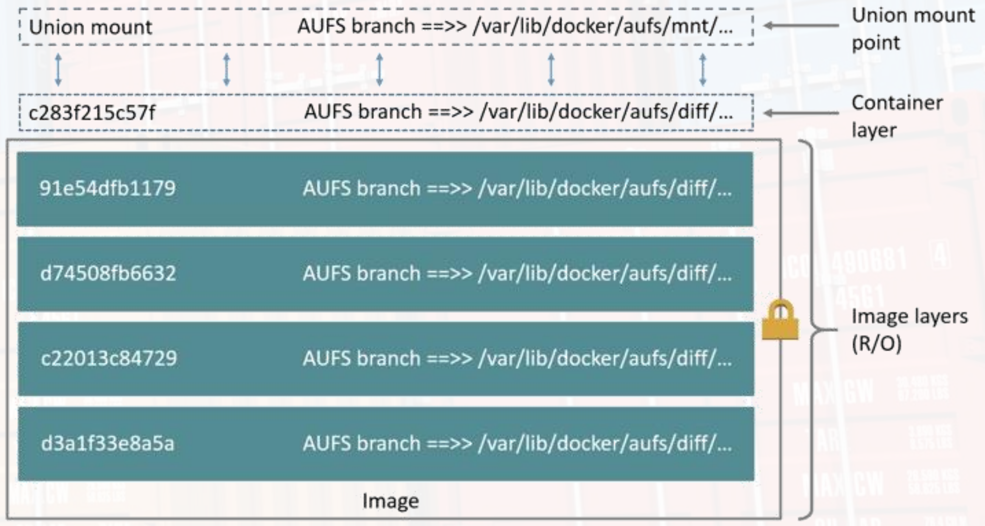
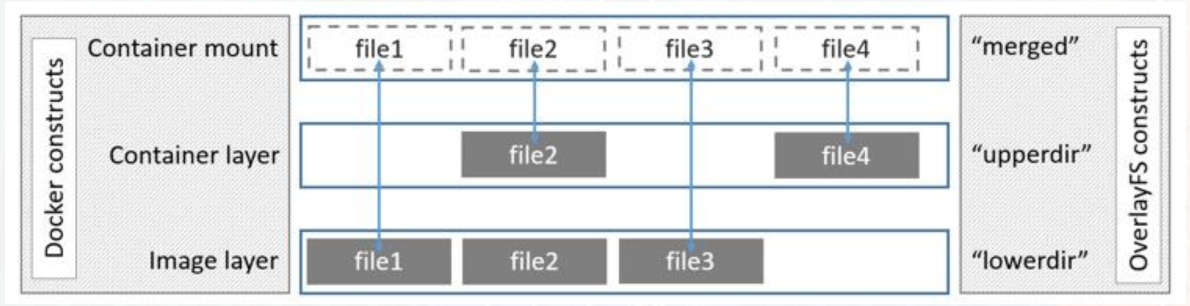

Union mount
AUFS branch ==>> /var/lib/docker/aufs/mnt/...
Union mount point
c283f215c57f
AUFS branch ==>> /var/lib/docker/aufs/diff/...
Container layer
91e54dfb1179
AUFS branch ==>> /var/lib/docker/aufs/diff/...
d74508fb6632
AUFS branch ==>> /var/lib/docker/aufs/diff/...
c22013c84729
AUFS branch ==>> /var/lib/docker/aufs/diff/...
d3a1f33e8a5a
AUFS branch ==>> /var/lib/docker/aufs/diff/...
Image layers (R/O)
Image

Image: Docker

Union Filesystems and Copy-on-Write

- CoW in Docker
  - Container layer (top) only keeps track of changes (diff)
  - Benefits:
    - No copying of image files
    - Instantaneous boot time from image
    - Container filesystem manipulation remains local
      - Does not affect RO layers
      - Does not affect other containers based on image



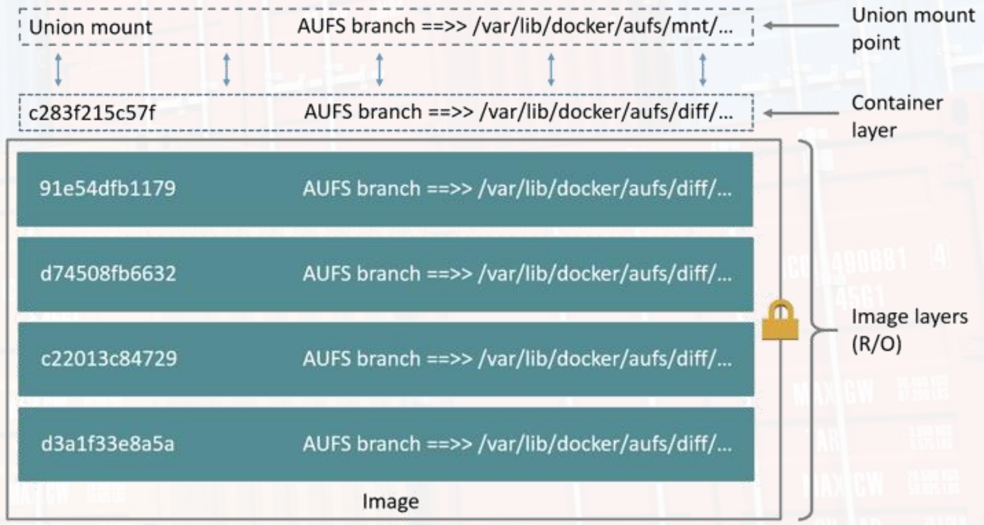
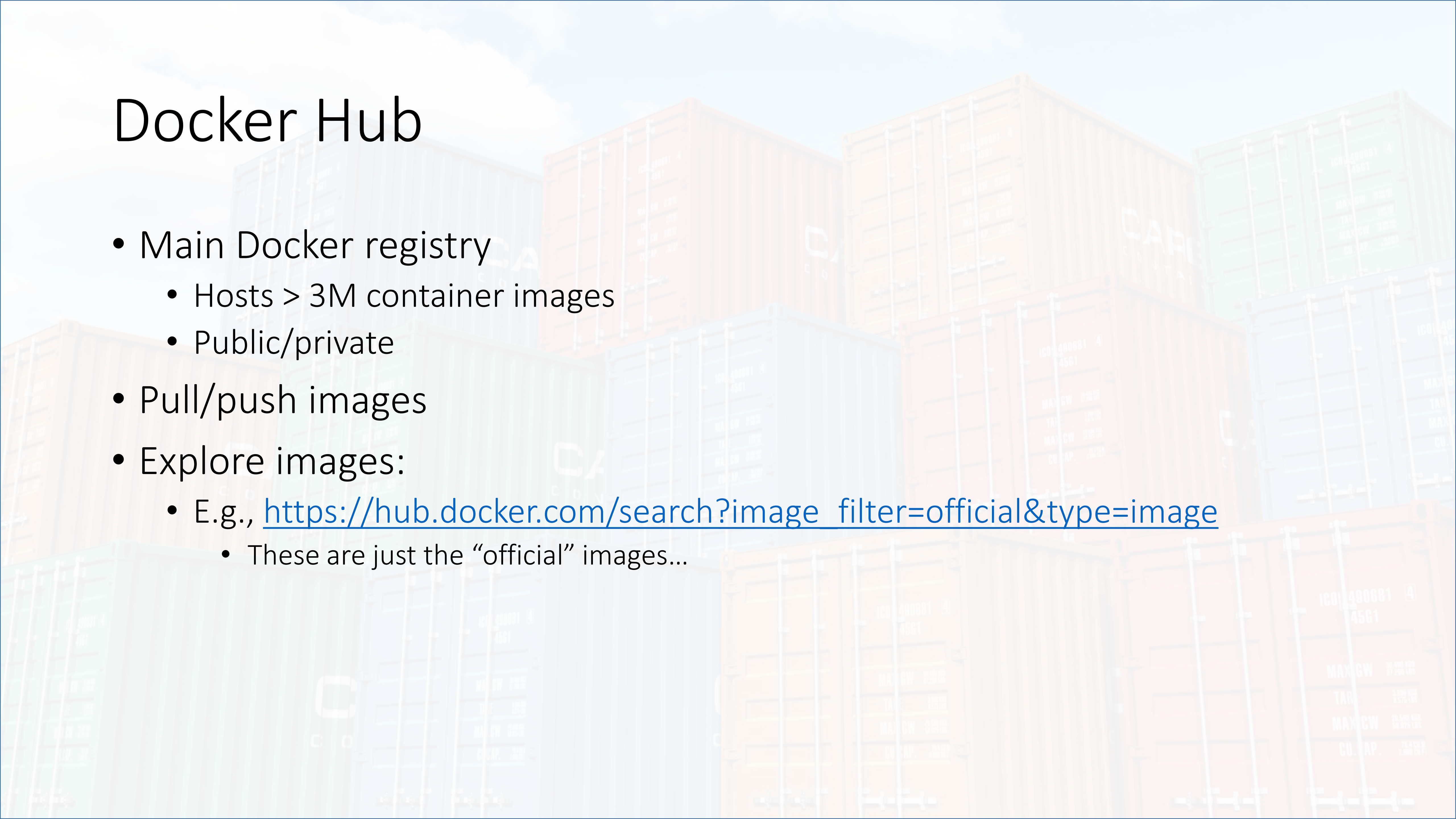
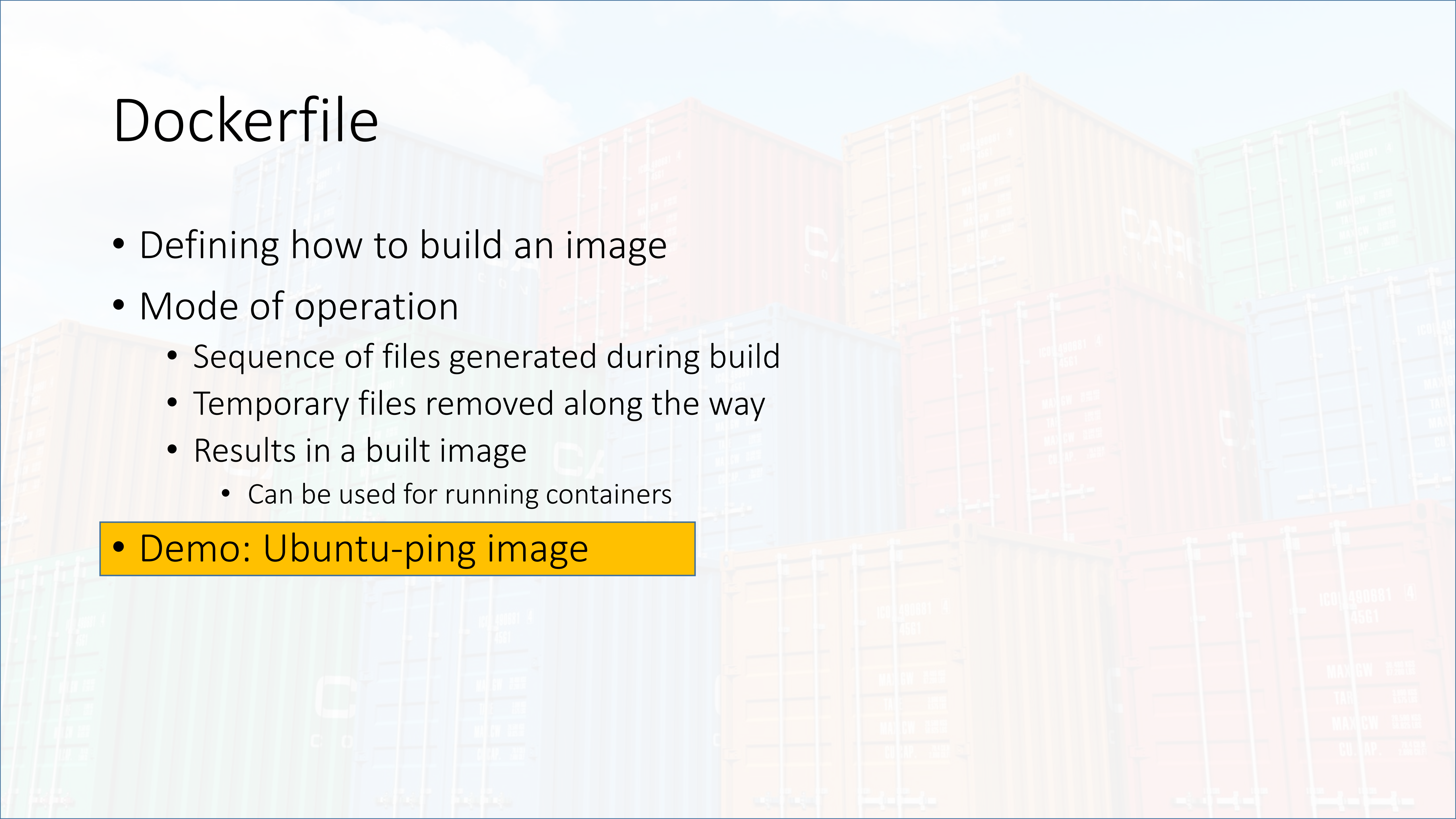

Union mount
AUFS branch ==>> /var/lib/docker/aufs/mnt/...
Union mount point
c283f215c57f
AUFS branch ==>> /var/lib/docker/aufs/diff/...
Container layer
91e54dfb1179
AUFS branch ==>> /var/lib/docker/aufs/diff/...
d74508fb6632
AUFS branch ==>> /var/lib/docker/aufs/diff/...
c22013c84729
AUFS branch ==>> /var/lib/docker/aufs/diff/...
d3a1f33e8a5a
AUFS branch ==>> /var/lib/docker/aufs/diff/...
Image layers (R/O)
Image

Image: Docker



Docker Hub

- Main Docker registry
  - Hosts > 3M container images
  - Public/private
- Pull/push images
- Explore images:
  - E.g., https://hub.docker.com/search?image_filter=official&type=image
    - These are just the “official” images…



Dockerfile

- Defining how to build an image
- Mode of operation
  - Sequence of files generated during build
  - Temporary files removed along the way
  - Results in a built image
    - Can be used for running containers
- Demo: Ubuntu-ping image

Docker Compose

- Deploying an application using containers
  - Each container provides a distinct service
- Structure
  - Multiple containers, single host
  - `docker-compose.yml` file
- Example:
  - Web server with a database
    - Define images to use per container
    - Define networking
    - …
  - Detailed example in the recitation

build web server using Dockerfile in current workdir

define container-host port mapping

```yaml
version: '3'
services:
  web:
    build: .
    ports:
     - "5000:5000"
  redis:
    image: "redis:alpine"
```

Image: Docker

use image for backend DB

VM-Containers Hybrids

- Managing private container-based cloud
  - Hosted on VM-based infrastructure

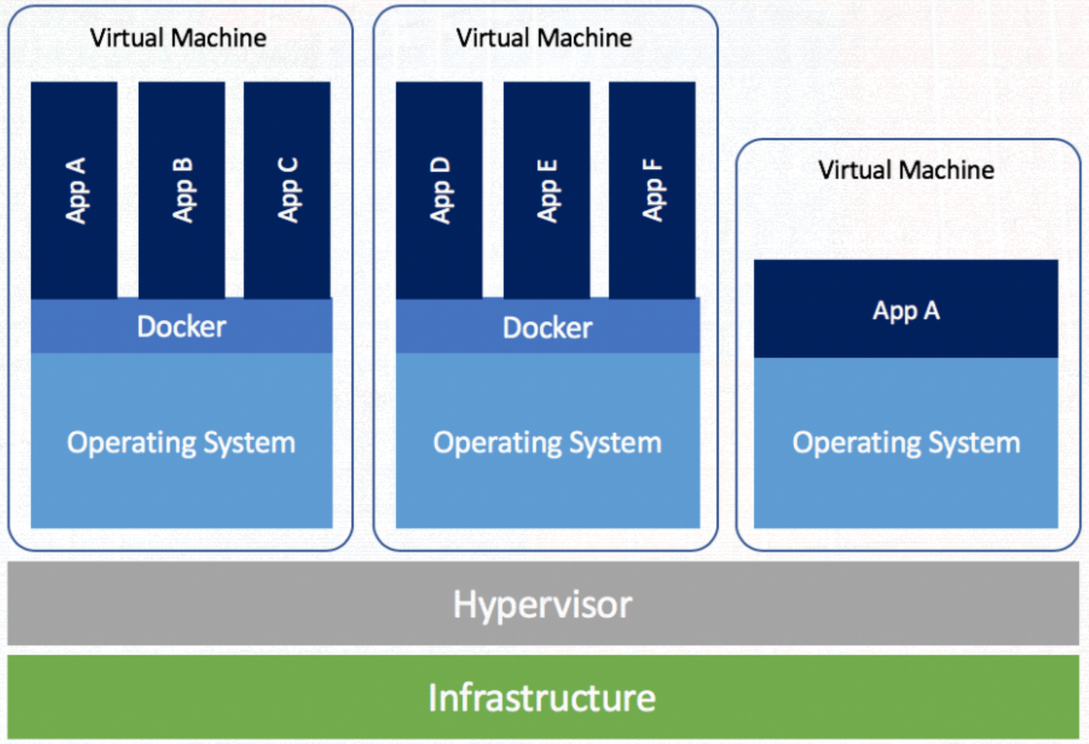


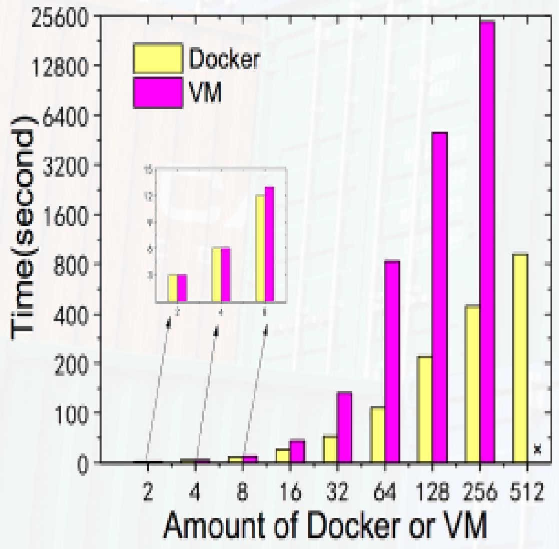
Image: Docker

Containers vs. VMs

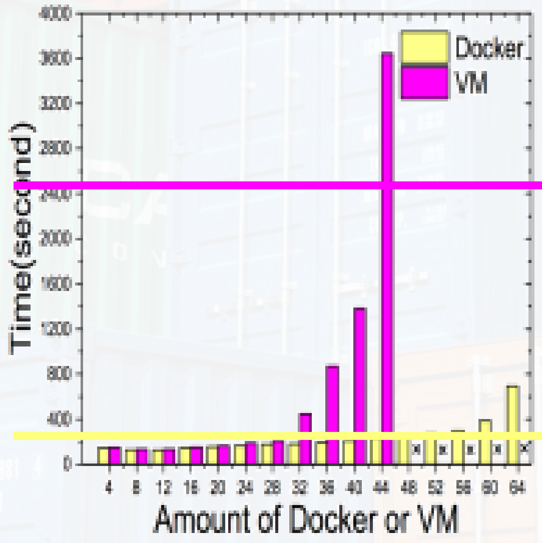
	Containers	VMs
Abstracting/virtualizing	Application environment and OS	Infrastructure and HW
Footprint size	10s MB	Several GB
Host Density	High	Low
Host utilization	High	Low (unused VM memory)
Boot time & Lifespan	Fast boot, supports very short lifespan	Slow boot, usually long lifespan
Hypervisor / OS	Share host OS kernel	Independent OS per VM No host-OS in bare-metal deployment
System diversity on host	Single OS per host	Multiple guest OS on same host
Security	Highly sensitive to host security	Less sensitive to host security
Portability	Simpler	Harder (e.g., paravirtualization changes)
Migration	Easy (small state footprint)	Harder

Containers vs. VMs

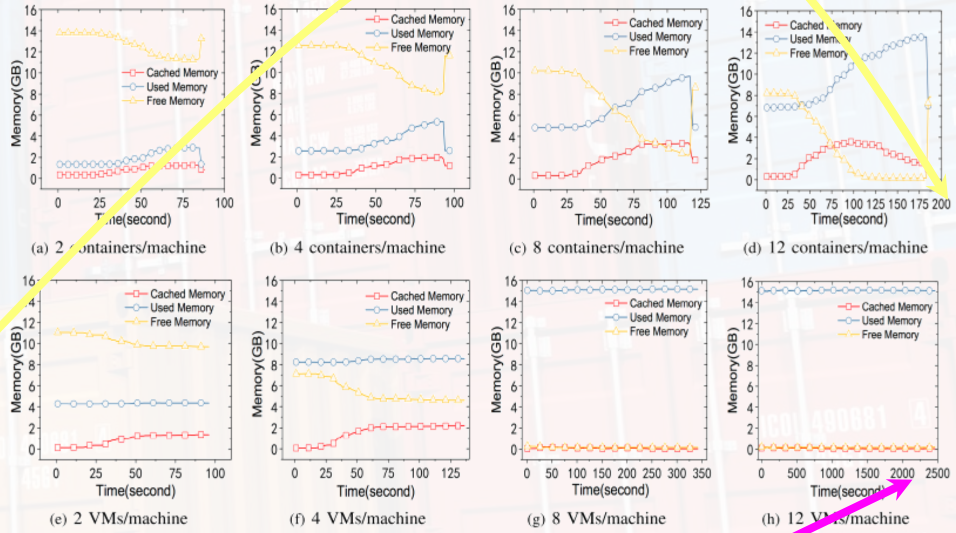
- Some concrete examples:



Bootup latency



Example workload execution

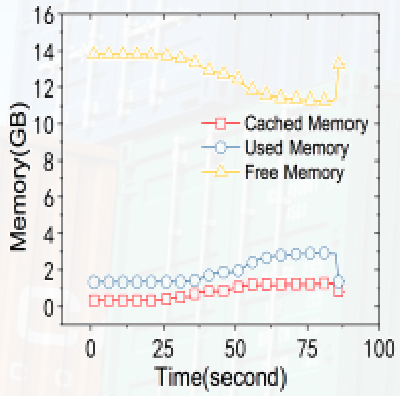


Used memory during example workload execution

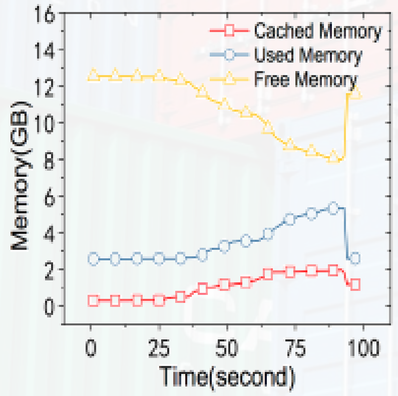
source: Zhang et al. 2018

Containers vs. VMs

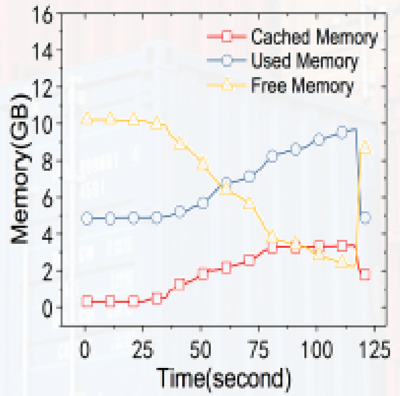
- Some concrete examples:



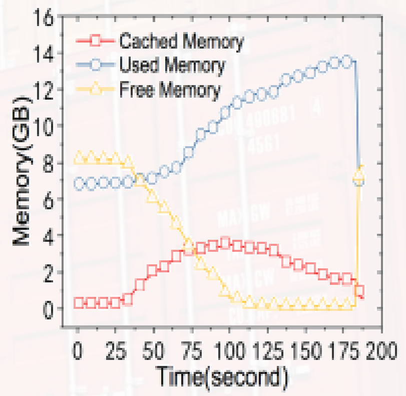
(a) 2 containers/machine



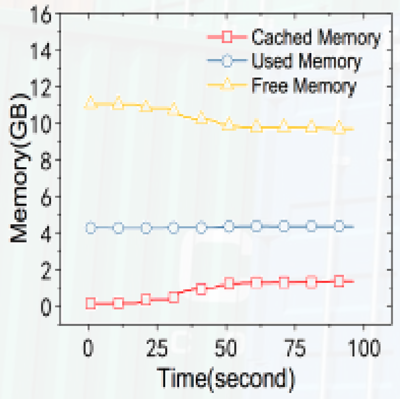
(b) 4 containers/machine



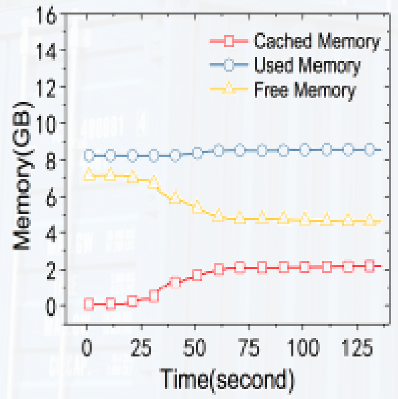
(c) 8 containers/machine



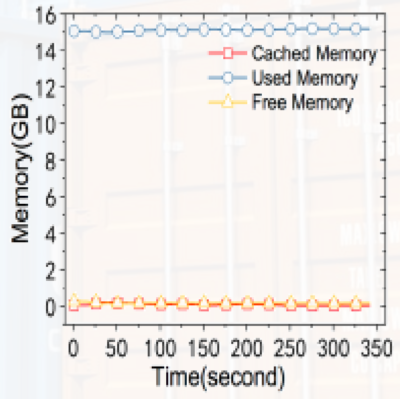
(d) 12 containers/machine



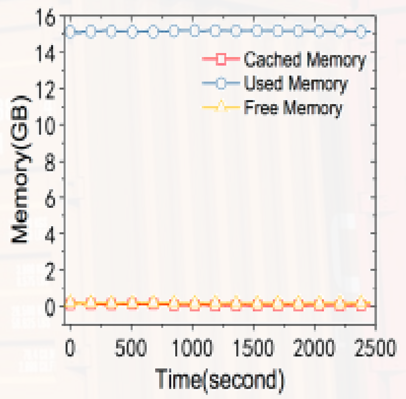
(e) 2 VMs/machine



(f) 4 VMs/machine



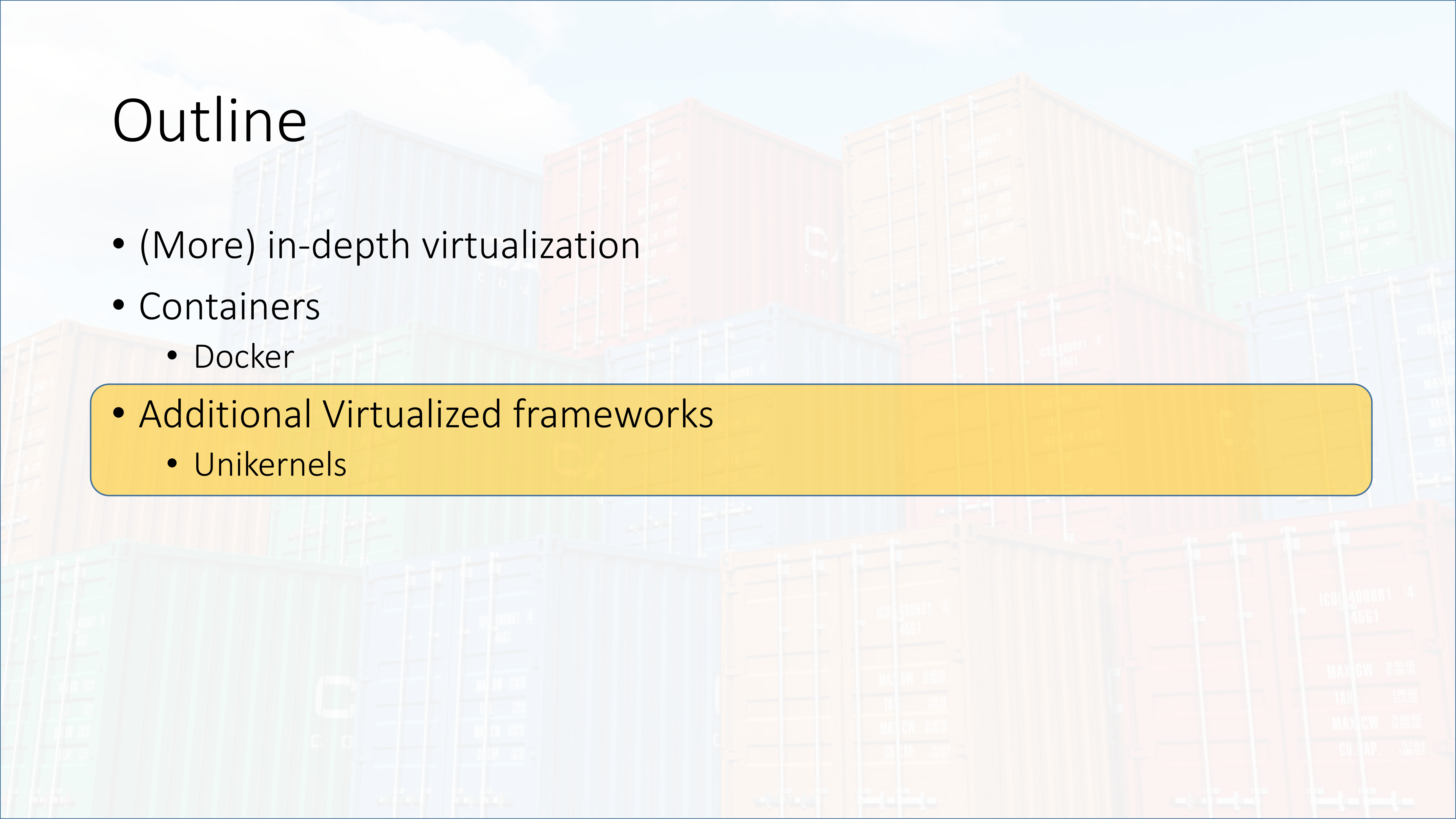
(g) 8 VMs/machine



(h) 12 VMs/machine

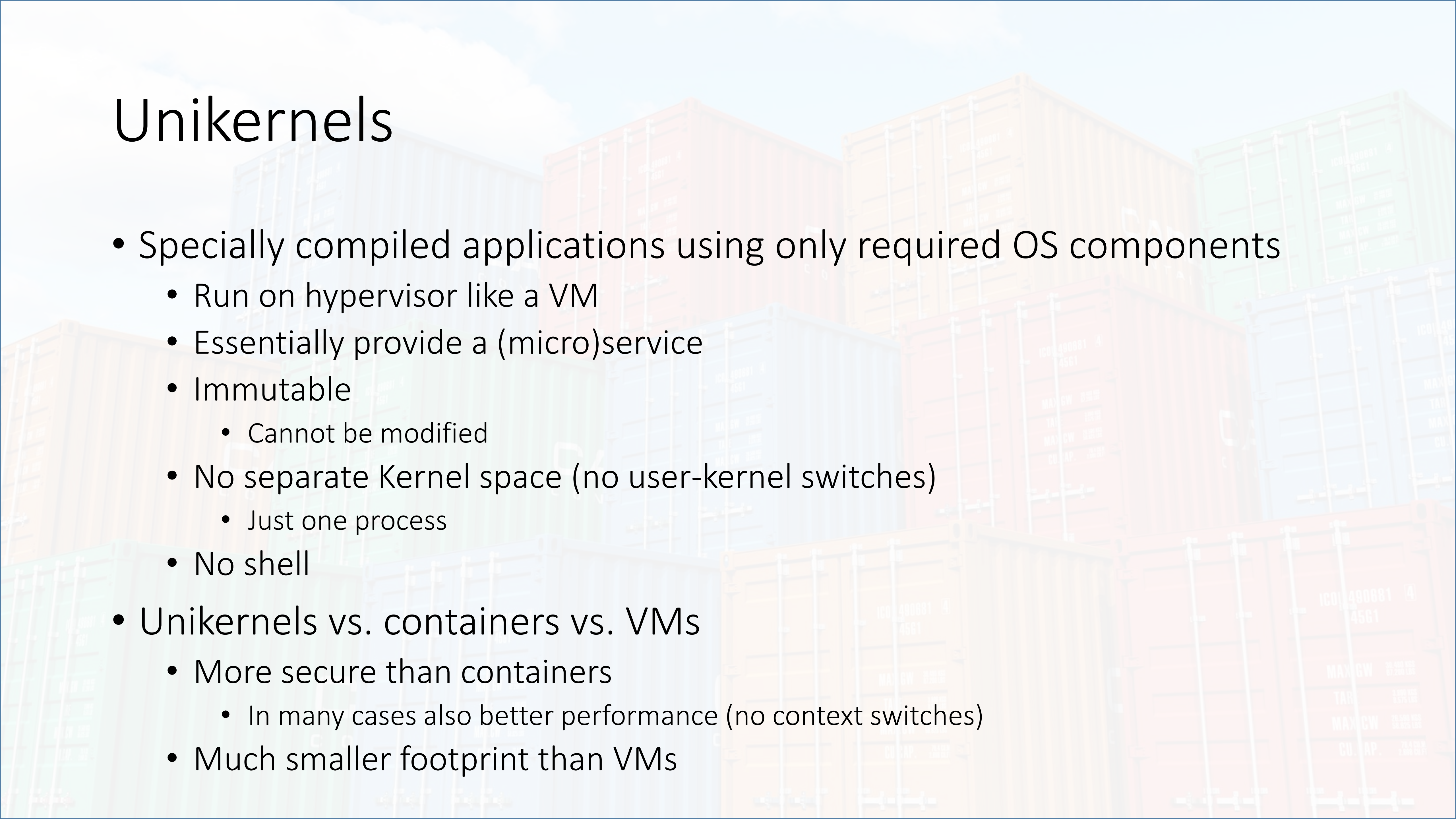
Used memory during
example workload execution

source: Zhang et al. 2018



Outline

- (More) in-depth virtualization
- Containers
  - Docker
- Additional Virtualized frameworks
  - Unikernels

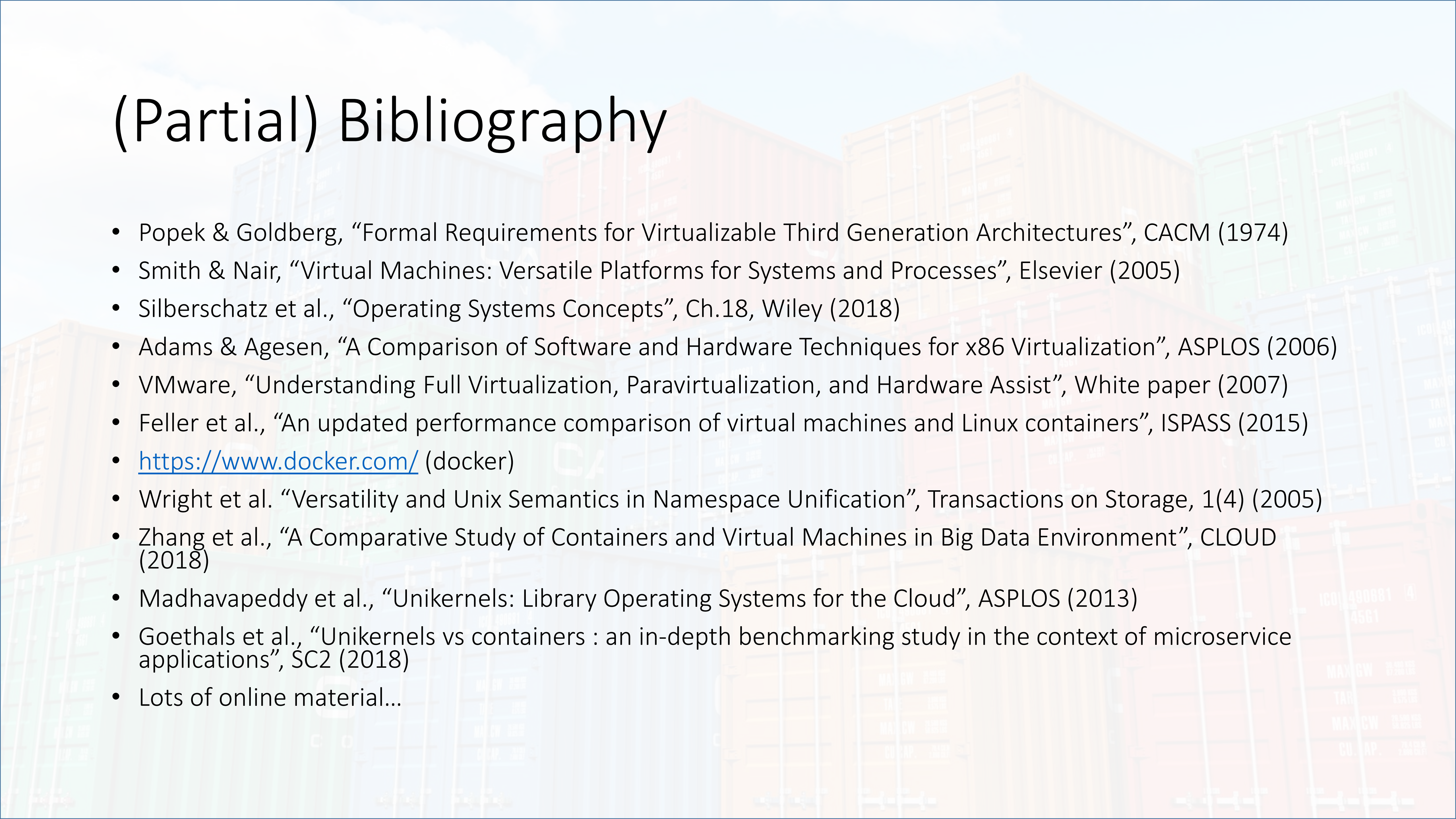


Unikernels

- Specially compiled applications using only required OS components
  - Run on hypervisor like a VM
  - Essentially provide a (micro)service
  - Immutable
    - Cannot be modified
  - No separate Kernel space (no user-kernel switches)
    - Just one process
  - No shell
- Unikernels vs. containers vs. VMs
  - More secure than containers
    - In many cases also better performance (no context switches)
  - Much smaller footprint than VMs

Unikernels

- Development “for unikernels”
  - Usually harder to develop
  - In most cases requires code be especially written to run on unikernel
    - E.g., sometimes no fork()…
- Examples of unikernel types:
  - MirageOS, Rumprun, OSv, HalVM, …
- Special tools for unikernels handling:
  - UniK: docker-like tool for compiling application source into unikernels
    - Supports various unikernel types
  - Solo5: sandboxed execution environment
    - Debugging



(Partial) Bibliography

- Popek & Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures”, CACM (1974)
- Smith & Nair, “Virtual Machines: Versatile Platforms for Systems and Processes”, Elsevier (2005)
- Silberschatz et al., “Operating Systems Concepts”, Ch.18, Wiley (2018)
- Adams & Agesen, “A Comparison of Software and Hardware Techniques for x86 Virtualization”, ASPLOS (2006)
- VMware, “Understanding Full Virtualization, Paravirtualization, and Hardware Assist”, White paper (2007)
- Feller et al., “An updated performance comparison of virtual machines and Linux containers”, ISPASS (2015)
- https://www.docker.com/ (docker)
- Wright et al. “Versatility and Unix Semantics in Namespace Unification”, Transactions on Storage, 1(4) (2005)
- Zhang et al., “A Comparative Study of Containers and Virtual Machines in Big Data Environment”, CLOUD (2018)
- Madhavapeddy et al., “Unikernels: Library Operating Systems for the Cloud”, ASPLOS (2013)
- Goethals et al., “Unikernels vs containers : an in-depth benchmarking study in the context of microservice applications”, SC2 (2018)
- Lots of online material…