Virtualization

AGISIT

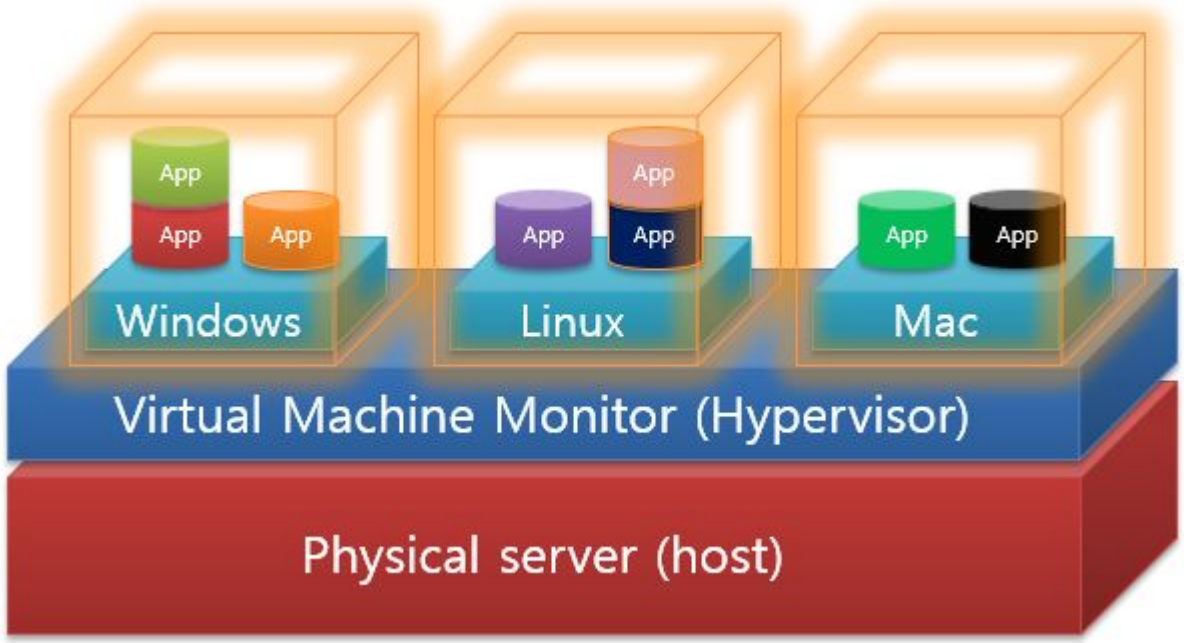
MEIC/METI - DEI - Instituto Superior Técnico

Agenda

- Virtual machines
- Hypervisors
  - Full-virtualization
  - Paravirtualization
  - Hardware-assisted virtualization
- Virtualization of virtual memory
- I/O virtualization
- Virtual machine migration
- Automation of VM creation
- Containers
- MicroVMs and Unikernels

What is server virtualization?

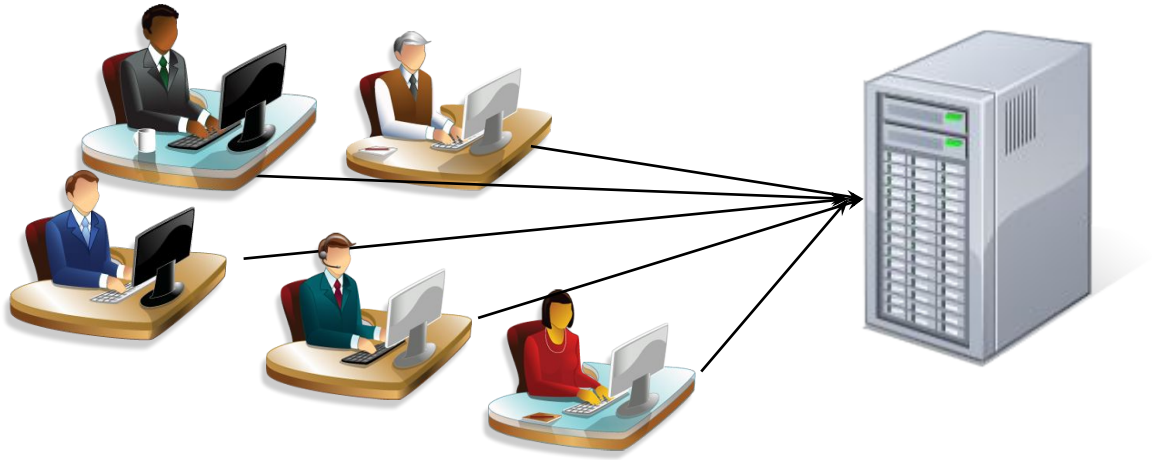
- One physical server hosts several Virtual Machines (VMs)
- Each VM runs its own operating system and applications, under the **illusion of having its own physical server**
- A VM is unaware that it is **sharing** the underlying hardware with other VMs
- The Virtual Machine Monitor (VMM), a.k.a. Hypervisor, isolates and orchestrates the various VMs



- **What is a hypervisor?** (with examples)

Birth of virtualization

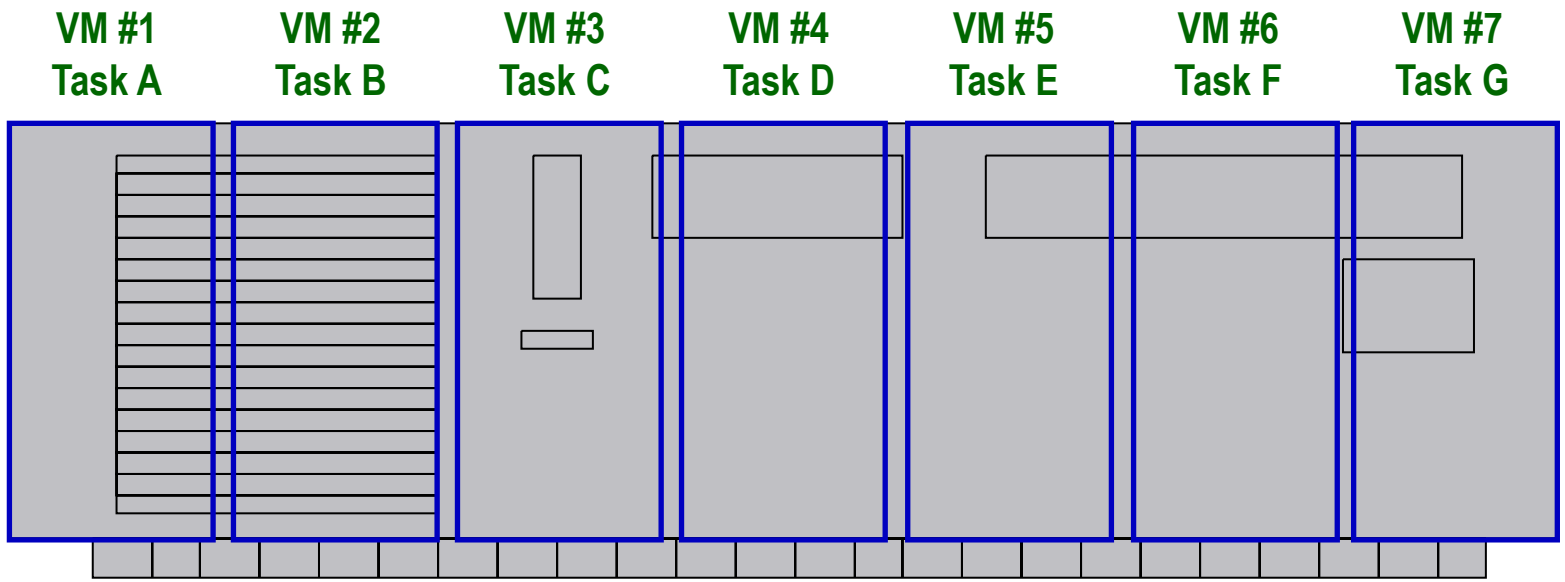
- 1964: IBM CP-40 (support for **more than user**, simultaneously)
- 1967: IBM CP-40 (first **hypervisor**)
- 1972: IBM VM/370 (**virtual memory** and redesigned hypervisor)



- 1960s-1970s: Powerful, expensive and time-shared computers (**mainframes**)

Mainframe virtualization

- Split the computer into multiple virtual machines, so different “tasks” can be run separately and independently on the same mainframe.
- If one “**task**” fails, others are unaffected

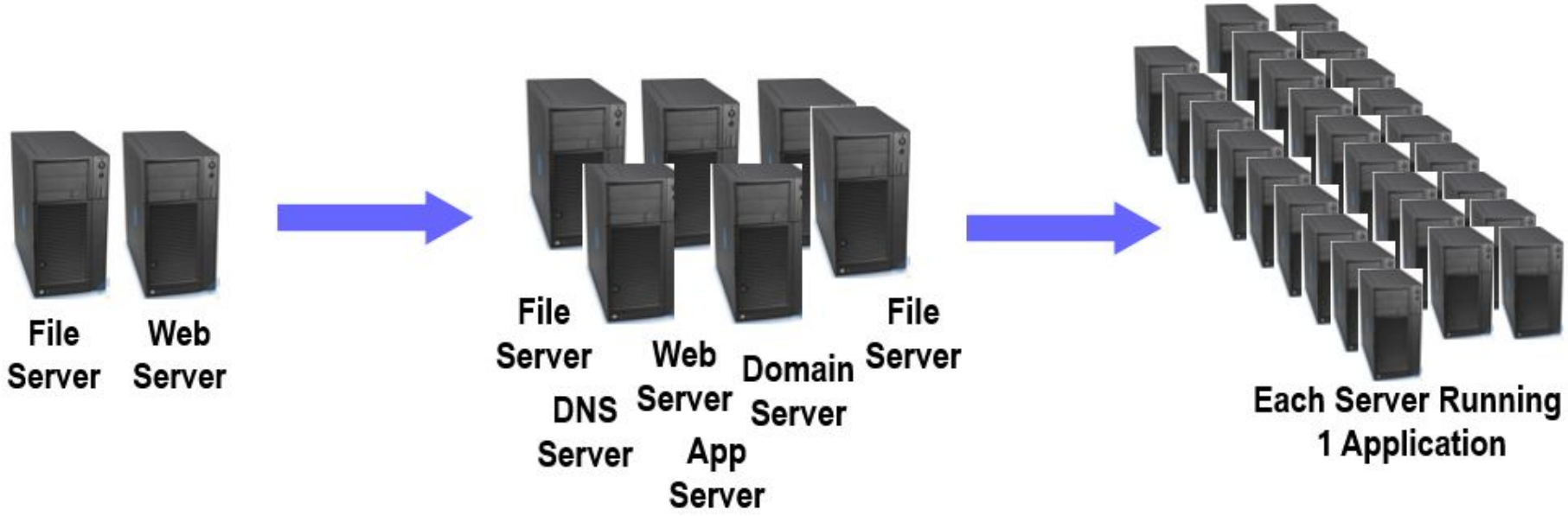


Logical VMs on a mainframe

Stagnation of virtualization (1980s-1990s)

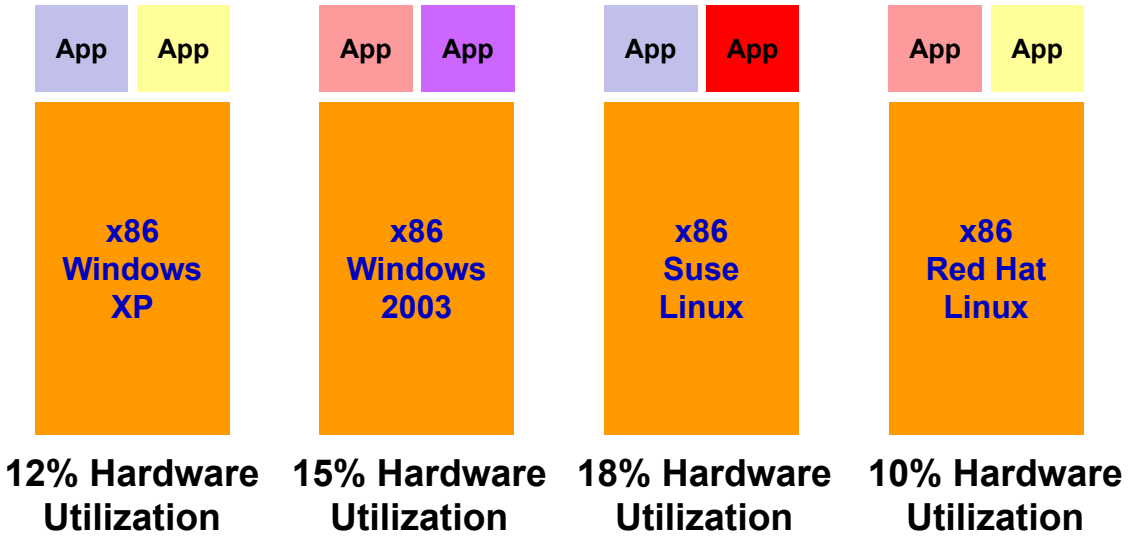
- Emergence of the PC:
  - **Cheap** hardware: **sharing not** needed
  - **Single-user** applications
- Servers:
  - High volume manufacturing changed server paradigm from custom-built to **general-purpose** processors
  - Applications **share the same OS**
  - If an application crashes, the whole server can crash (**poor isolation**)
  - Processors with no hardware support for virtualization

IT infrastructures in the late 1990s – early 2000s



- Intel/AMD servers are now very popular (known as “x86” servers)
- Each server runs one operating system such as Windows or Linux
- Typically, one OS and one major application per server (for isolation)
- Physical **server sprawl** is inevitable
- **Power, cooling, and rack space** become a problem

Pre-virtualization scenario (circa 2000)



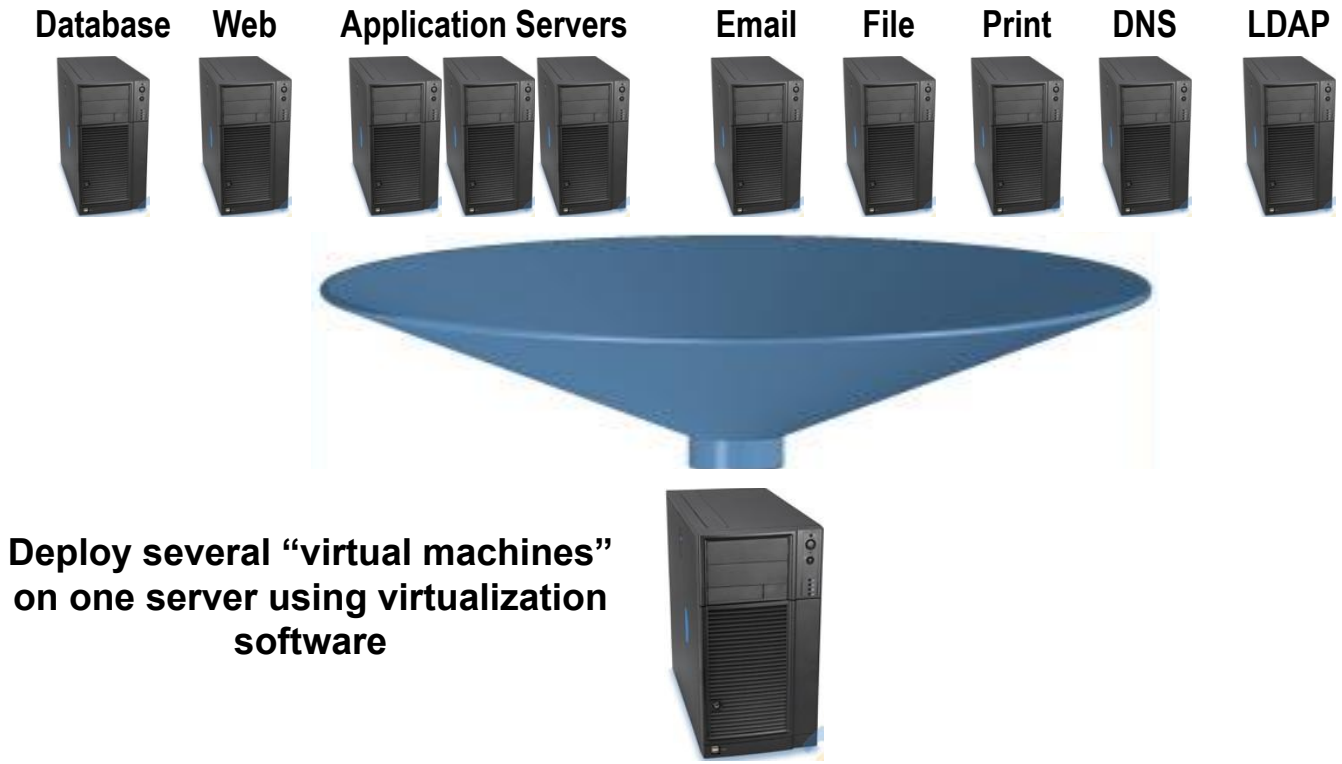
- Applications can affect each other
- Each server: one OS, one or very few applications (for isolation)
- Big disadvantage: **hardware utilization is very low**
  - most of the times it is under 20%

Focus on reducing footprint

- Blades and racks
- Density increased, space reduced
- Power density and heat/cooling problems increased
- **The more powerful the CPU, the lower the server utilization!**
- Average server utilization ranges between 10-20%
- Still one major application per server

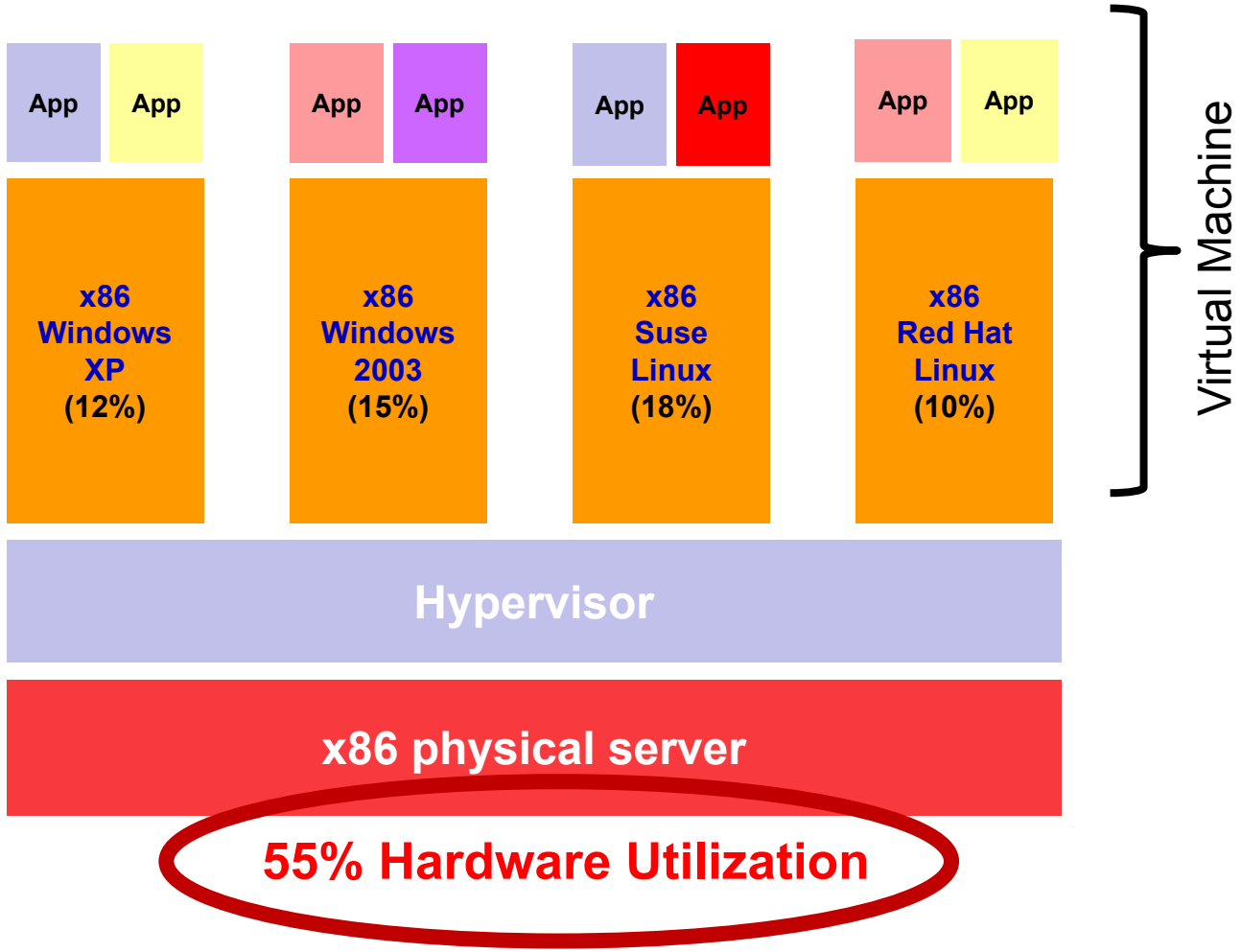
Solution: virtualization

- Apply mainframe virtualization concepts to servers
- Use virtualization software to make one physical server run several virtual machines
- Virtual machine: an instance of an operating system with several applications (isolated from others)



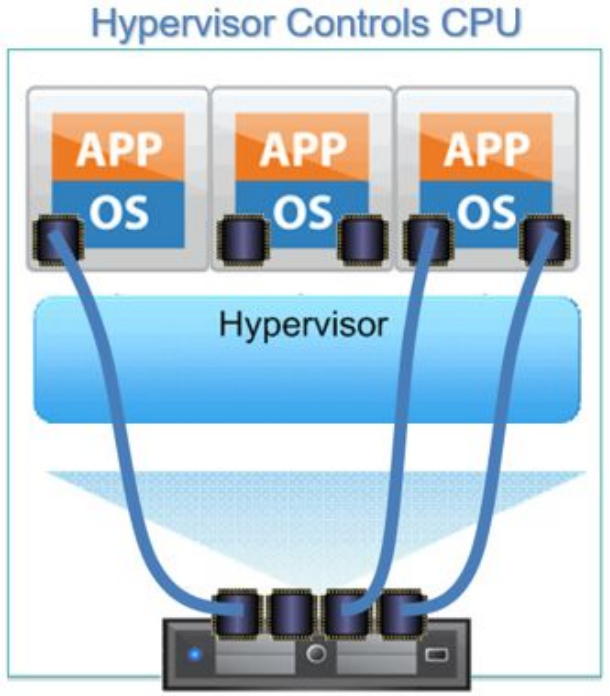
Post-virtualization scenario

- Each application runs on its own operating system
- Each operating system **does not know it is sharing** the underlying hardware with others
- An hypervisor manages the virtual machines
- Much **better hardware utilization**, without compromising application protection



Hypervisor scheduling: access to CPU

- Each OS “thinks” it has the host's processor, memory, and other resources, all to itself
- Actually, the **hypervisor controls the host processor** and resources, allocating what is needed to each operating system, in turn (round-robin)
- It also makes sure that the virtual machines (guest OS and applications) **cannot disrupt each other**



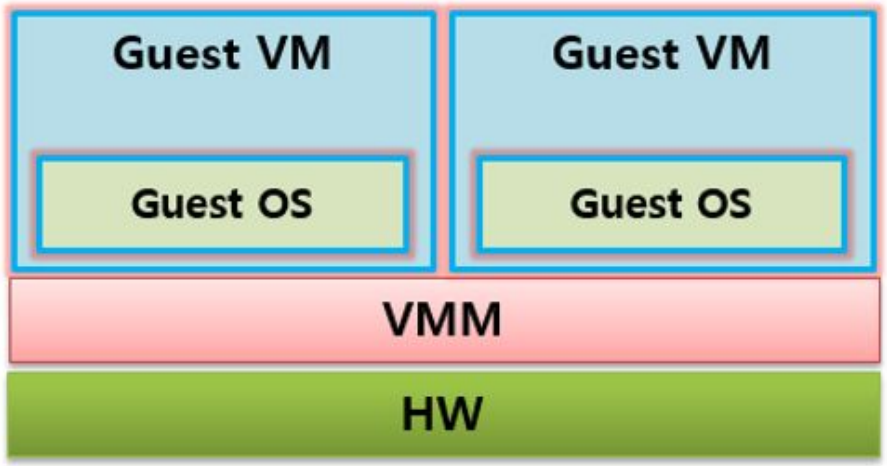
Hypervisor schedules VMs on each physical Core/CPU/Hyperthread

Complete control on how Cores are assigned to vCPUs

CPU will be used for hypervisor, virtual switches, etc.

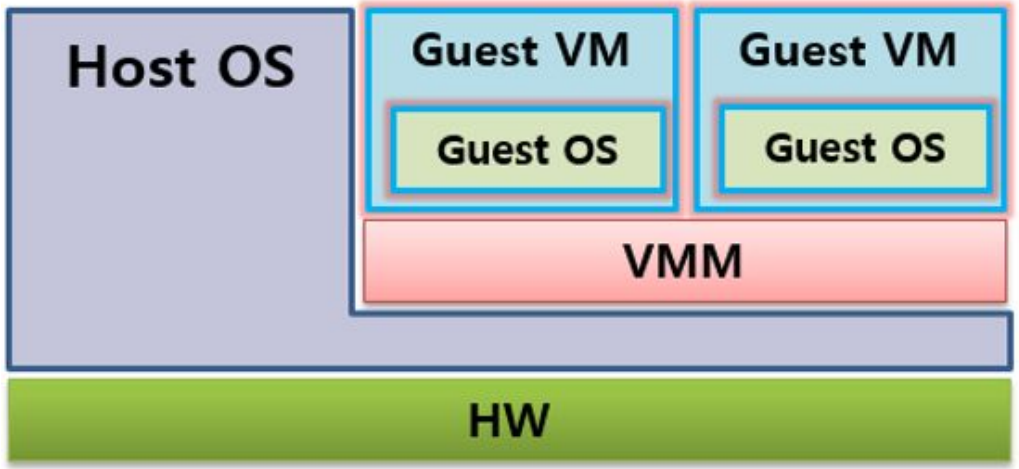
Hypervisor types

Type-1: VMM on HW (bare-metal)



- Xen, VMware ESX server, Hyper-V
- Mostly for server, but not limited
- VMM by default
- OS-independent VMM

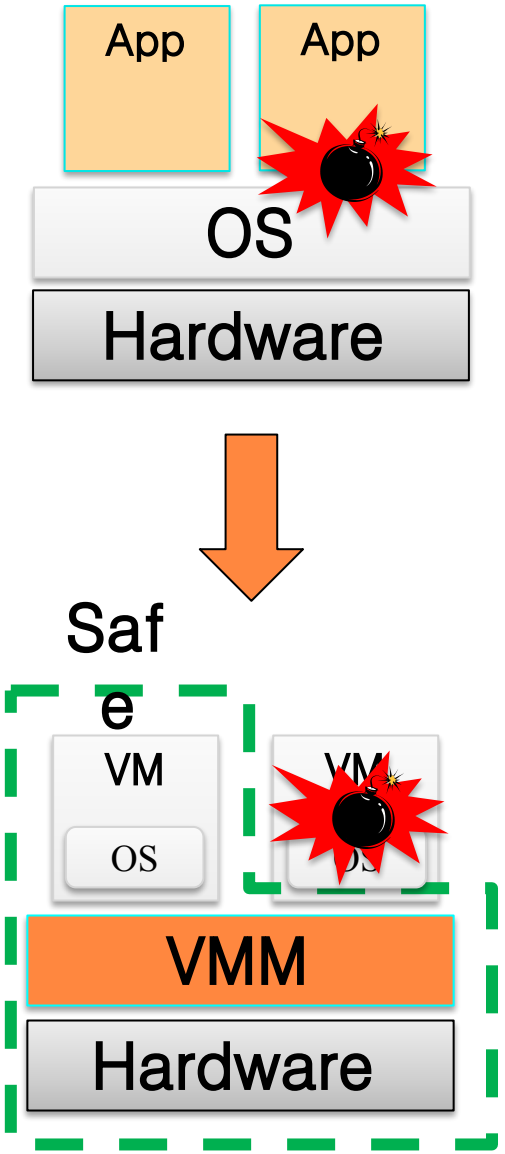
Type-2: VMM on Host OS



- KVM, VMware Workstation, VirtualBox
- Mostly for client devices, but not limited
- VMM on demand
- OS-dependent VMM

Major benefit: isolation

- **Strong isolation** between co-located VMs
  - fault containment
- Safe from bugs and malicious attacks
- A VMM (Virtual Machine Manager)
  - is much smaller than an OS
    - **small TCB** (Trusted Computing Base)
- A fault, bug or attack:
  - In a process, may crash the whole OS
  - In a VM, cannot affect other VMs
- Host-based hypervisors can still be affected by their host OS



Major pre-virtualization challenges

- Physical server sprawl
  - Power, space and cooling: one of the largest IT budget line items
  - One-application-per-server: high costs (equipment and administration)
- Low server and infrastructure utilization rates
  - Result in excessive acquisition and maintenance costs
- High business continuity costs
  - High Availability and Disaster Recovery solutions built around additional hardware are very expensive
- Ability to respond to dynamic business needs is hampered
  - Provisioning new servers/applications often a lengthy and tedious process (days/weeks)
- Securing environments
  - Security often accomplished through physical isolation: costly

Major results from virtualization

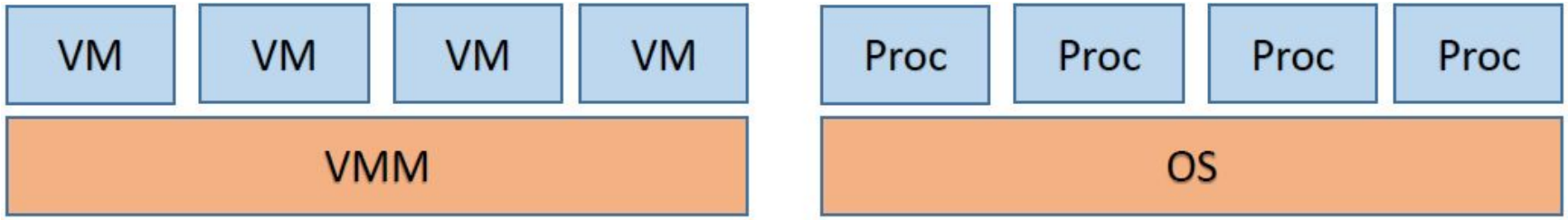
- Server sprawl (actually increased, but now virtually)
  - Power, space and cooling: substantially reduced
  - One-application-per-server: ~~high~~ moderate costs (~~equipment and~~ administration)
- High server and infrastructure utilization rates
  - Better use of less hardware: lower acquisition and maintenance costs
- Lower business continuity costs
  - High Availability and Disaster Recovery solutions built around additional ~~hardware~~ virtual machines are ~~very~~ less expensive (easy creation, migration and failover)
- Ability to respond to dynamic business needs is expedited
  - Provisioning new servers/applications usually a swift and easy process
- Securing environments
  - Security ~~often~~ accomplished through ~~physical~~ virtual isolation: ~~costly~~ essentially for free

Agenda

- Virtual machines
- Hypervisors
  - Full-virtualization
  - Paravirtualization
  - Hardware-assisted virtualization
- Virtualization of virtual memory
- I/O virtualization
- Virtual machine migration
- Automation of VM creation
- Containers
- MicroVMs and Unikernels

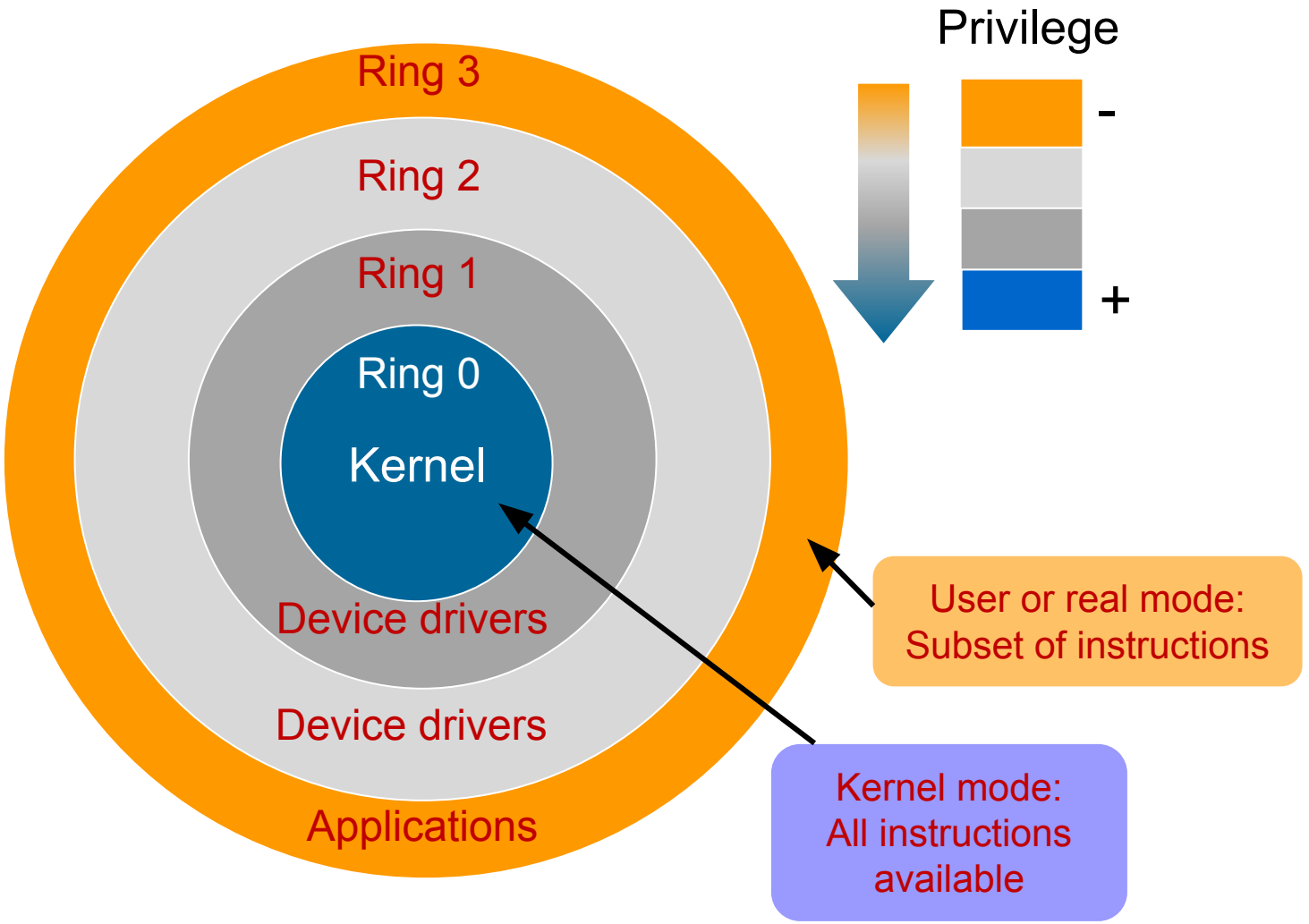
How does compute virtualization work?

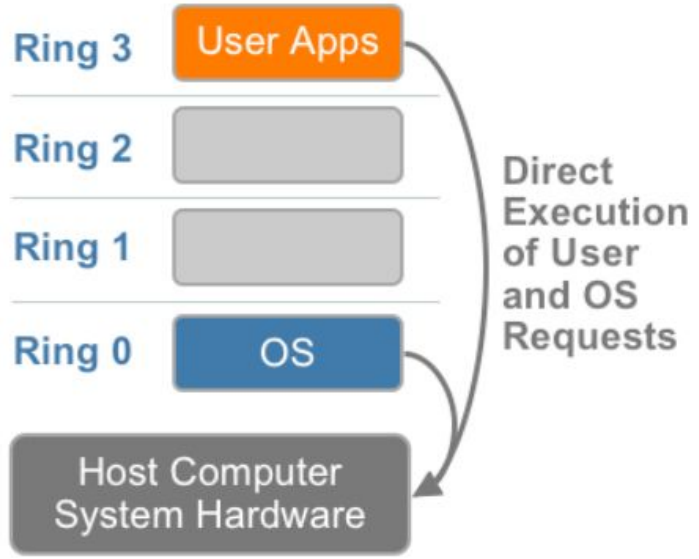
- A physical machine multiplexes the underlying CPU by the various VMs
  - **Similar** to how an **OS multiplexes processes** on CPU (time-sharing)



- The VMM performs machine switch (much like context switch)
  - Run a VM for a bit, save its state, switch to another VM, and so on…
- What is the problem?
  - Unlike user processes, a guest **OS expects** to have the **highest privileges** in the machine, to have unrestricted access to hardware, and to be able to run privileged instructions
  - To keep VMs isolated from each other, **guest** OSs must be **less privileged** than the **VMM**

x86 privilege rings (no virtualization)

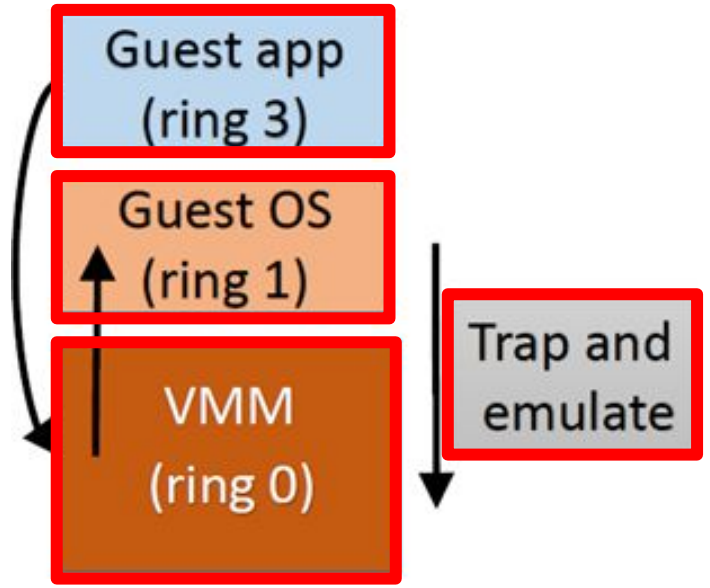




- OS kernels expect to run in Ring 0 with full privileges
- If a privileged instruction executes in Ring > 0, it traps (an exception occurs and the kernel is invoked)
- Access control is enforced by the CPU

De-privileging the guest OS in x86

- Where does the VMM fit in?
- The **VMM runs in ring 0** (with full privileges)
- The **guest OS runs in ring 1** (cannot execute privileged instructions, which trap into ring 0)
- System calls by the application that used to trap into the guest OS now trap into the VMM
  - The VMM doesn't know how to handle the trap
  - The VMM calls the usual guest OS trap handler
  - The guest OS trap handler returns (privileged instruction, so it traps to the VMM)
  - The VMM returns to the application



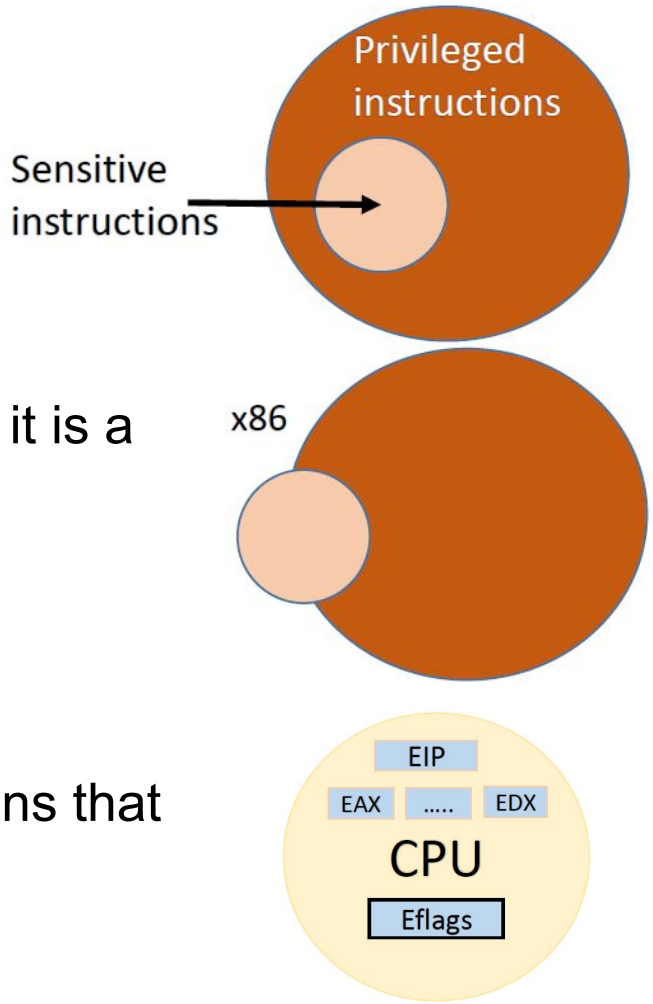
- All **privileged instructions** executed by the guest OS **trap** to the **VMM**, which emulates its effect (but under its control): trap and emulate

The Popek-Goldberg formalism (1974)

- Gerald Popek and Robert Goldberg formalized the requirements for a processor to **support virtualization efficiently** by trap and emulate:
  - **Sensitive instruction**: can change system state, or its semantics depend on system state
  - **Privileged instruction**: can run in privileged mode only (traps otherwise)
  - Theorem: virtualization is possible if all sensitive instructions are privileged
- *"A virtual machine is an efficient, isolated duplicate of the real machine"*
  - **Efficiency**: Non-sensitive instructions should execute directly on the hardware (virtualization should have a minor effect on performance)
  - **Isolation**: executed programs may not affect the system resources or other programs
  - **Equivalence**: the behavior of a program executing under the VMM should be the same as if the program is executed directly on the hardware (except possibly for timing and resource availability)

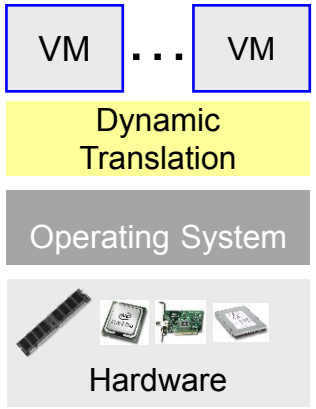
Problems with virtualization of x86

- Sensitive instructions must be a subset of privileged instructions
  - The x86 (32-bit) architecture does not satisfy this!
- For example, the POPF instruction in x86:
  - Pops values from the stack to the Eflags register (CPU flags)
  - IF (interrupt flag) indicates if interrupts are enabled
  - Executed in ring 0, all flags are set normally
  - Executed in ring 1, only some flags are set (e.g., IF is not set, since it is a privileged flag), and does not trap if ring > 0!
- This means that:
  - POPF is a sensitive instruction, but not privileged
  - Behaves differently when executed in different privilege levels
  - The x86 instruction set contains 17 sensitive, unprivileged instructions that do not trap if ring > 0
  - The x86 was not designed with virtualization in mind!

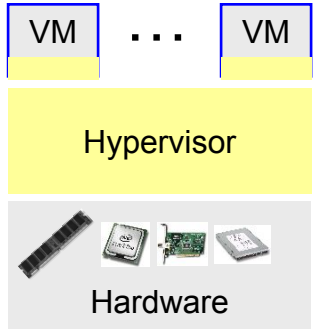


Evolution of virtualization

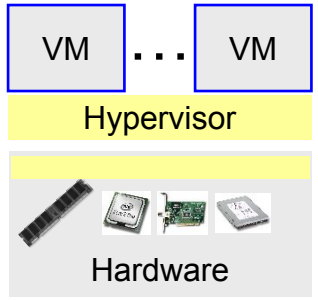
- 1st Generation (1999):
  Full virtualization
  (Binary translation)
  - Software-based
  - VMware



- 2nd Generation(2003):
  Paravirtualization
  - Cooperative virtualization
  - Modified guest
  - VMware, Xen



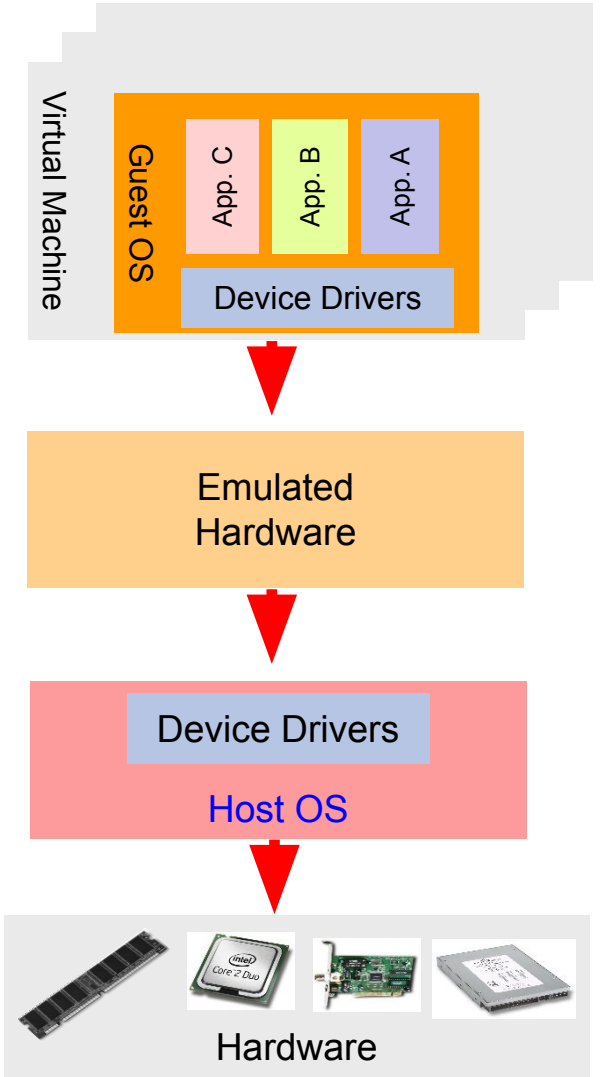
- 3rd Generation (2005):
  HW-assisted virtualization
  - Silicon-based
  - Unmodified guest
  - VMware, KVM, Hyper-V



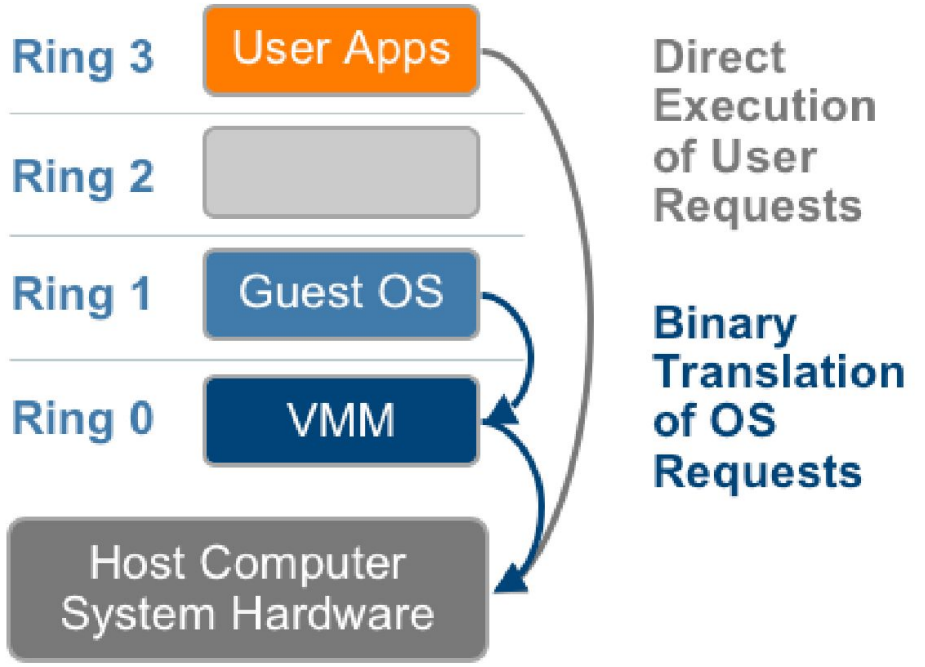
Virtualization Logic

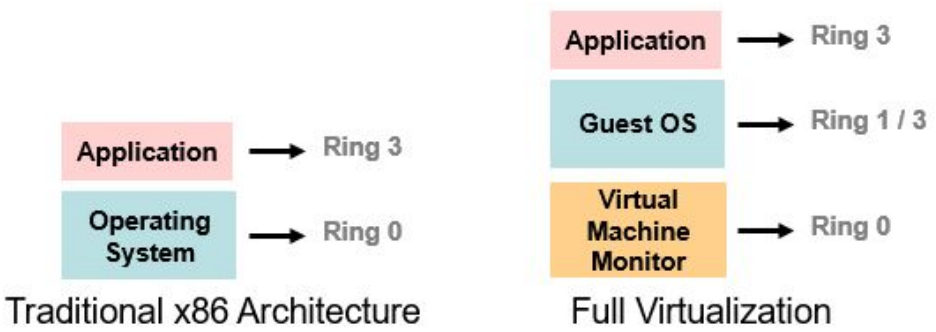
Full virtualization

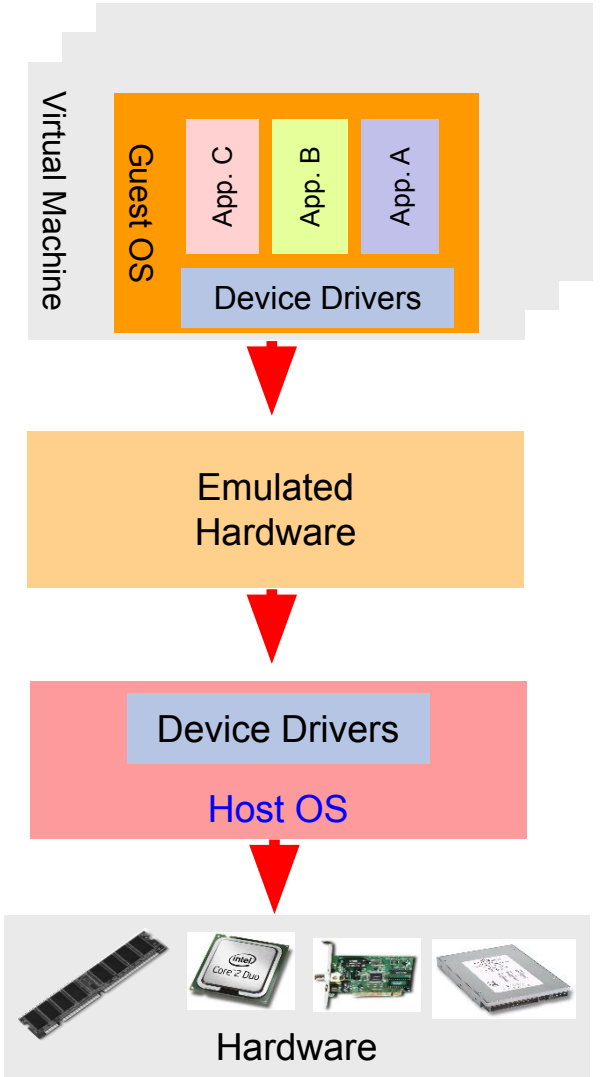
- 1st Generation offering of x86 server virtualization
- Dynamic binary translation
  - The **guest OS** is **scanned** at run-time and sensitive but unprivileged instructions are replaced by **explicit trap instructions** (to invoke emulation routines)
  - Privileged instructions already trap in rings > 0
  - Non-sensitive (innocuous) instructions run at native speed
  - The host OS provides basic system functionality (used by emulation routines to implement virtual resources)
  - The guest OS doesn't see this emulated environment
- The host **OS kernel runs at ring 0** and the **guest OS at ring 1**, trapping (by SW or by HW to ring 0)
- Lower than native performance!



Full virtualization

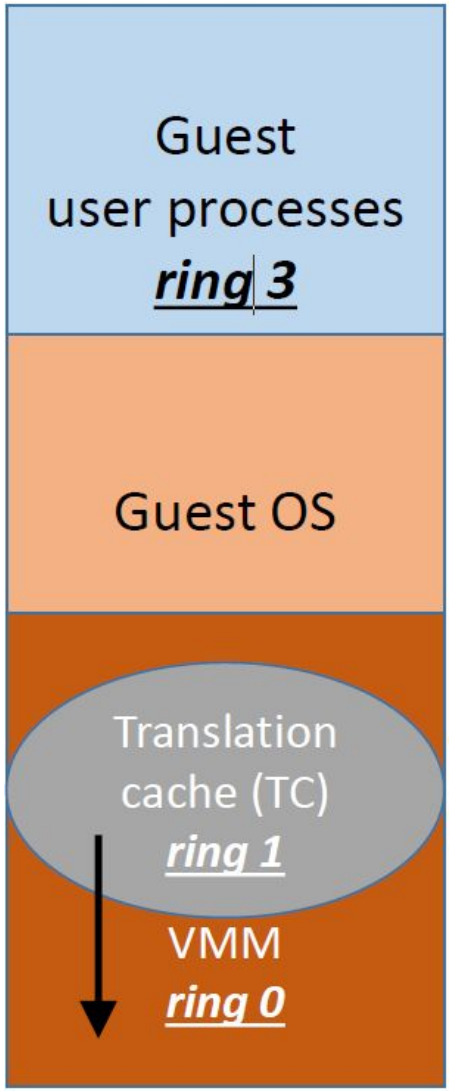






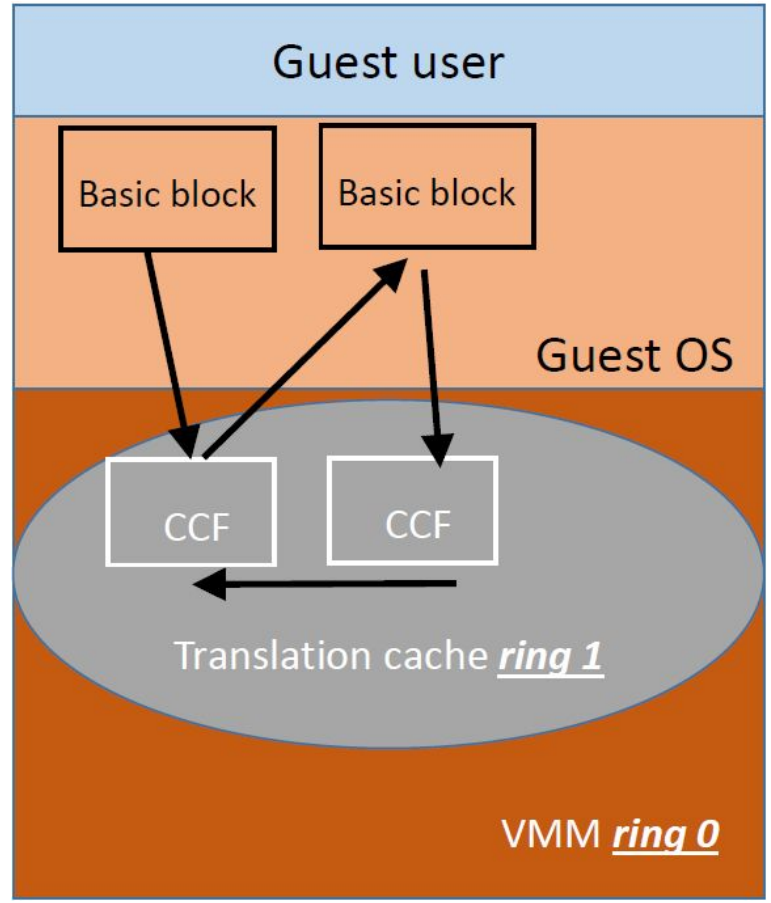
Binary Translation

- Guest OS binary is translated instruction by instruction and stored in translation cache (TC)
  - Part of VMM memory
  - Most code stays the same, unmodified
  - Sensitive but unprivileged instructions modified to trap
  - Guest OS code is thus modified to work correctly in ring 1
- Guest OS code executes from TC in ring 1
- Privileged OS instructions trap to VMM

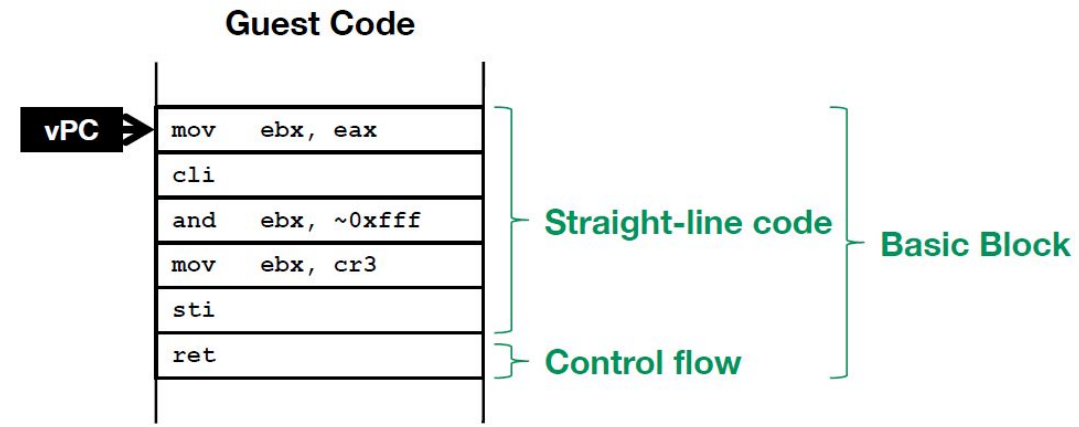


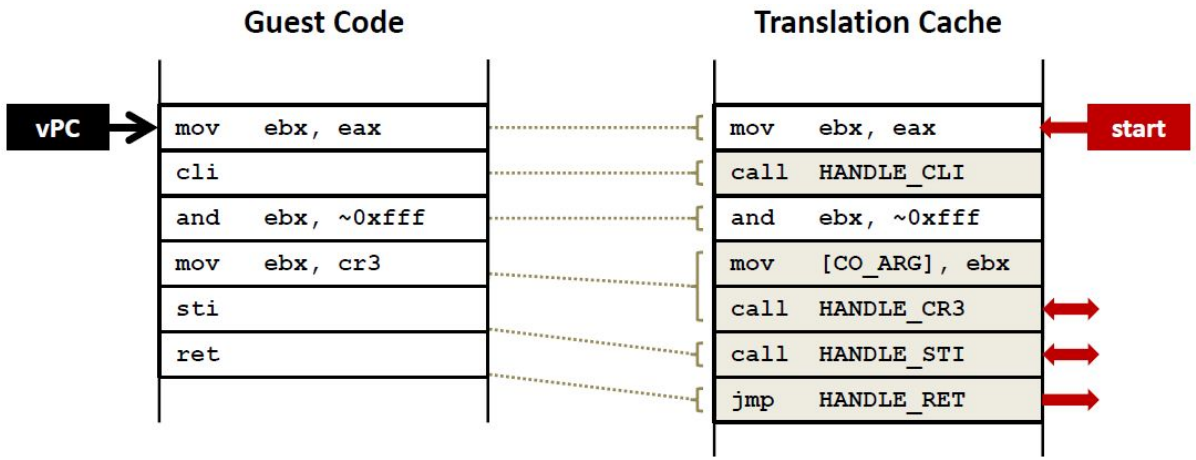
Basic blocks and compiled code fragments

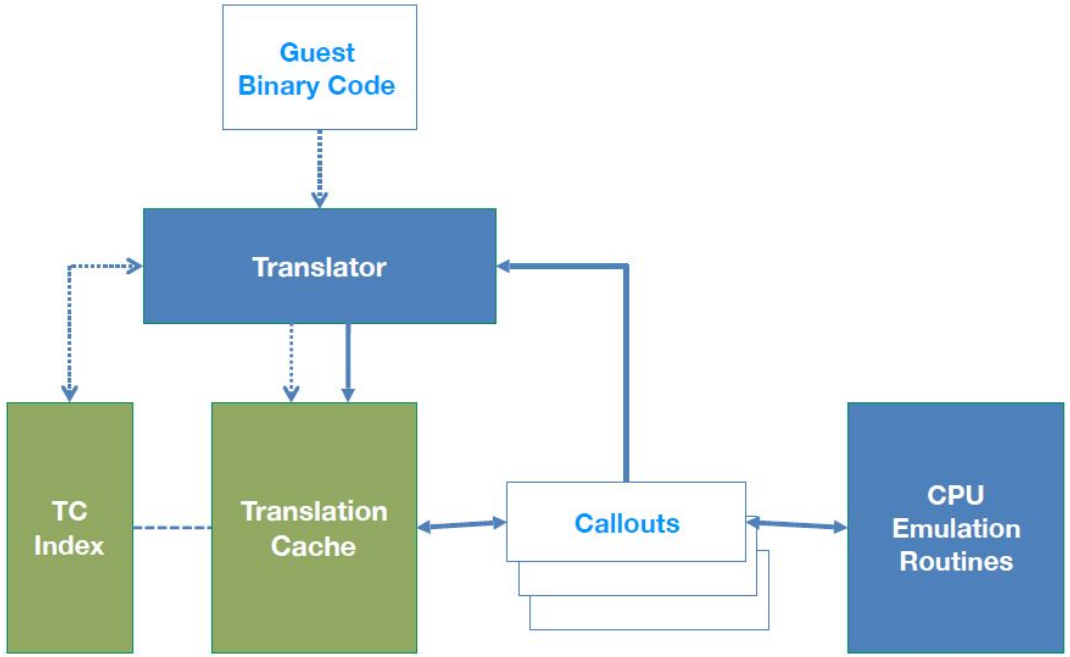
- VMM translator logic (ring 0) **translates guest code** one **basic block** at a time to produce a compiled code fragment (CCF)
  - Basic block = sequence of instructions until a jump/call/return
- Once a CCF is created, move it to ring 1 to run translated guest code
- Once a CCF ends, invoke the VMM translator, compute the next instruction to jump to, translate, run CCF, and so on
- If the next CCF is already in TC, jump directly to it without invoking the VMM translator
  - Optimization **called *chaining***



How binary translation works

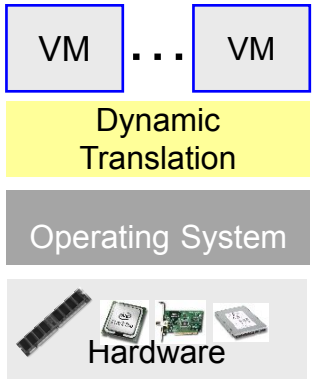




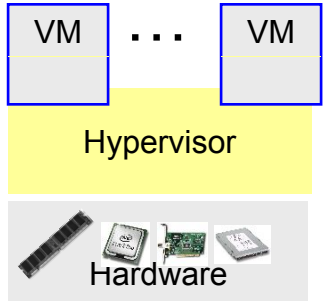


Evolution of virtualization

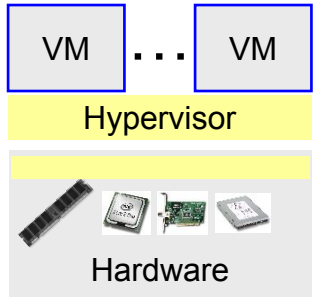
- 1st Generation (1999):
  Full virtualization
  (Binary translation)
  - Software-based
  - VMware



- 2nd Generation(2003):
  Paravirtualization
  - Cooperative virtualization
  - Modified guest
  - VMware, Xen



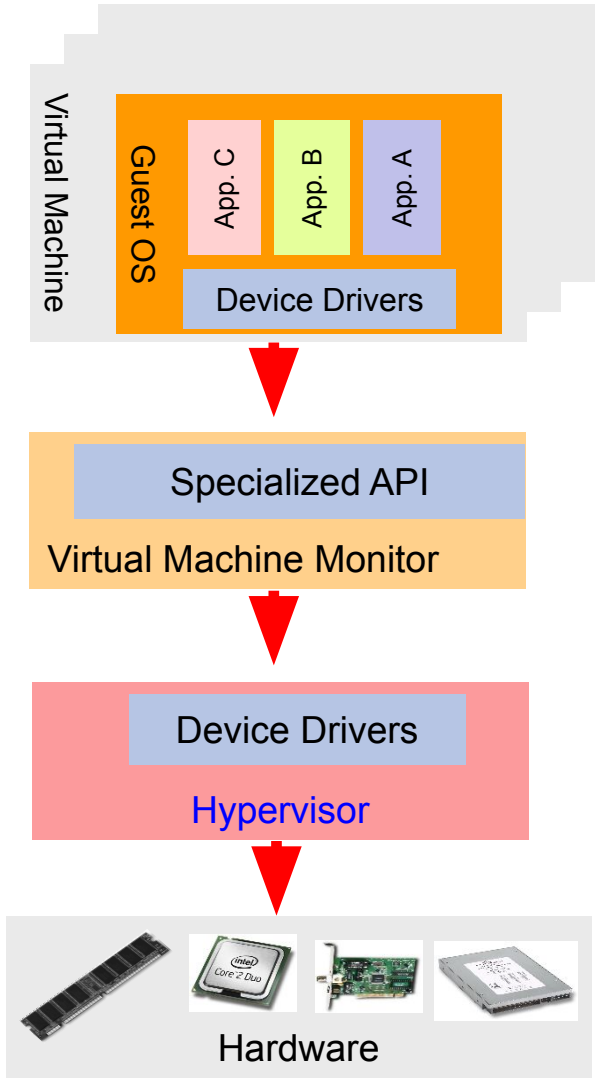
- 3rd Generation (2005):
  HW-assisted virtualization
  - Silicon-based
  - Unmodified guest
  - VMware, KVM, Hyper-V



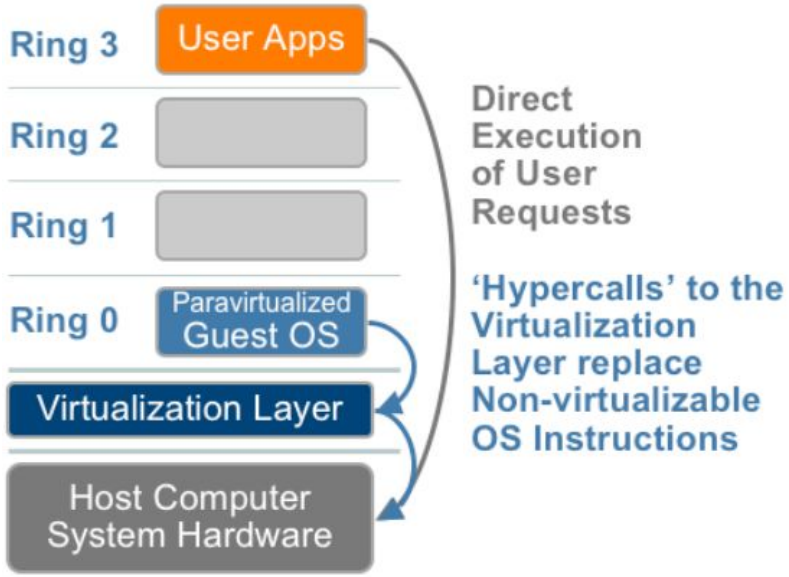
Virtualization Logic

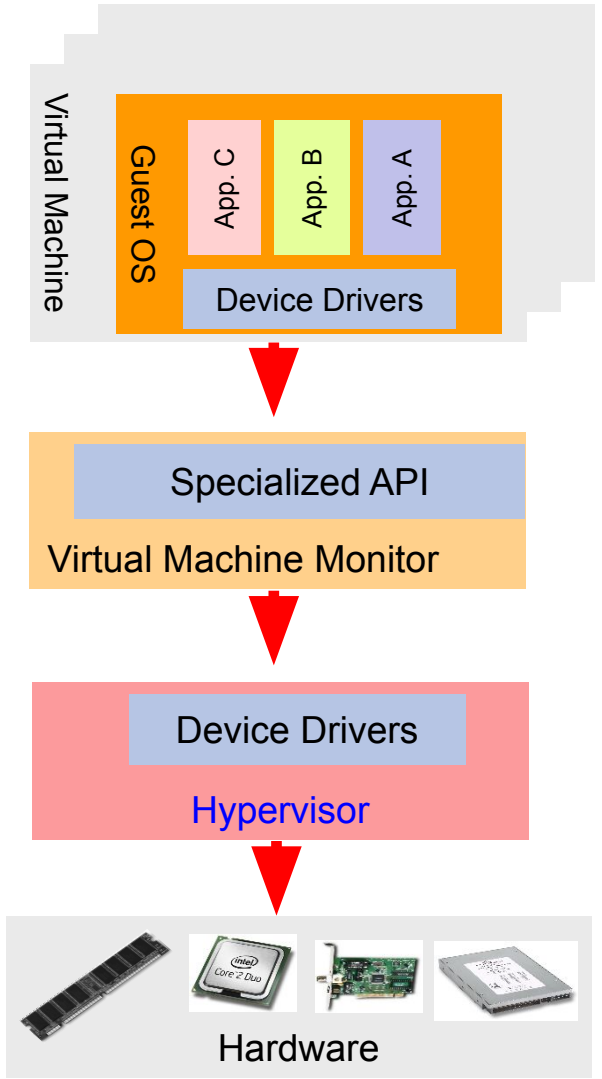
Paravirtualization

- The **guest OS is rewritten** to replace privileged instructions with "**hypercalls**" (calls to the hypervisor), making it suitable to run at Ring 0
- The VMM is responsible for handling the virtualization requests and putting them to the hardware
  - The guest operating system uses a **specialized API** to talk to the VMM and to execute the functionality of sensitive, unprivileged instructions
  - Instruction translation by the VMM is no longer necessary (better performance)
  - But the **guest** is now **aware of virtualization** (requires source code changes to OS, porting effort)



Paravirtualization

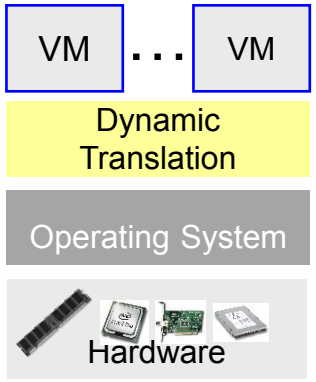




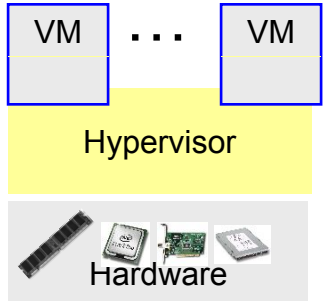
- Since hypercalls replace sensitive instructions, the **guest OS can run in ring 0** (with the hypervisor) or 1
- **Tight coupling**: guest kernel must be recompiled when hypervisor is updated

Evolution of virtualization

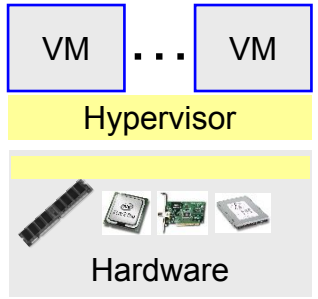
- 1st Generation (1999):
  Full virtualization
  (Binary translation)
  - Software-based
  - VMware



- 2nd Generation(2003):
  Paravirtualization
  - Cooperative virtualization
  - Modified guest
  - VMware, Xen

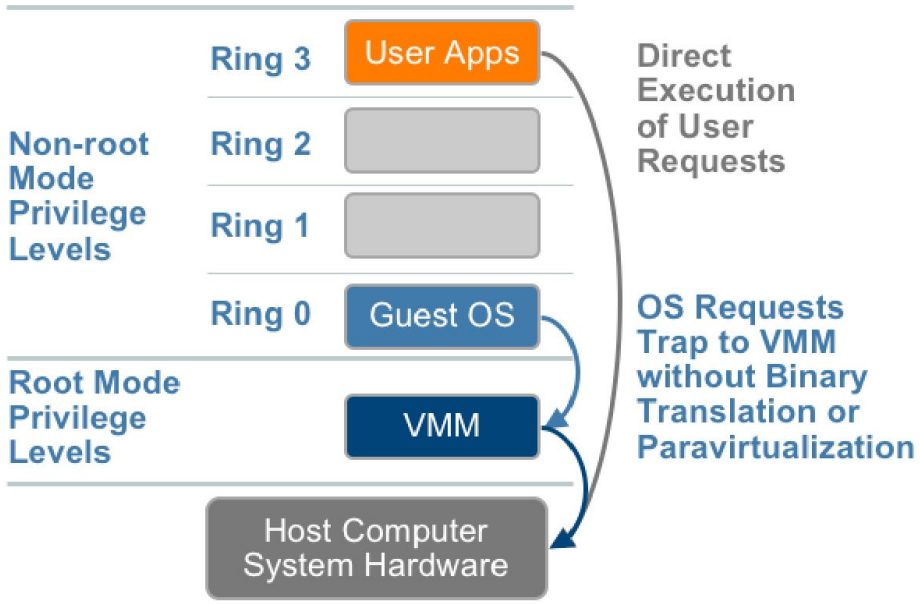


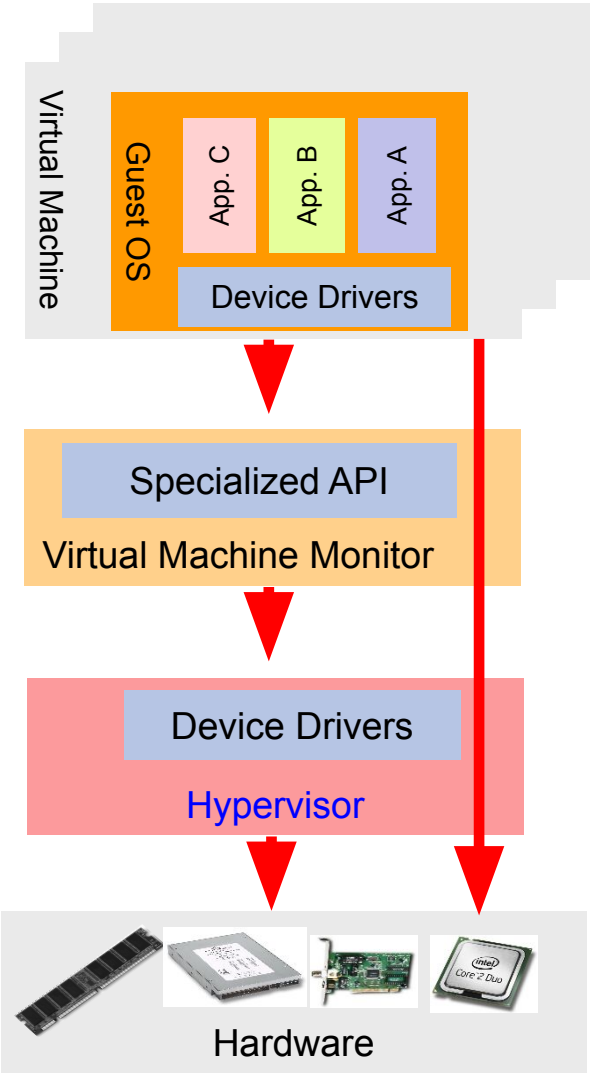
- 3rd Generation (2005):
  HW-assisted virtualization
  - Silicon-based
  - Unmodified guest
  - VMware, KVM, Hyper-V



Virtualization Logic

Hardware-assisted virtualization

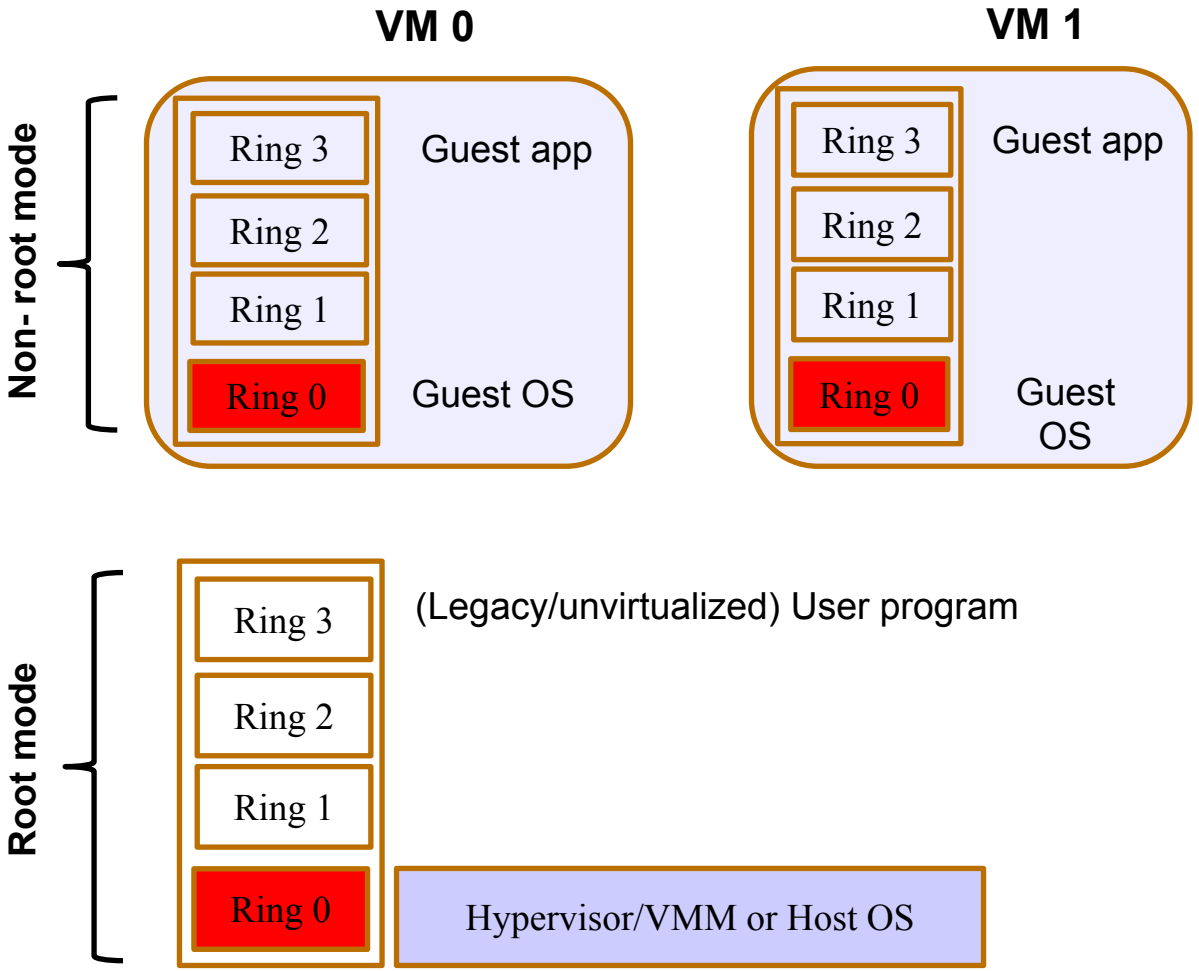




- The VMM runs in a more privileged ring than 0 (a virtual -1 ring, or root mode)
- The virtual machine (guest OS and applications) run unmodified in non-root mode, using rings as usual

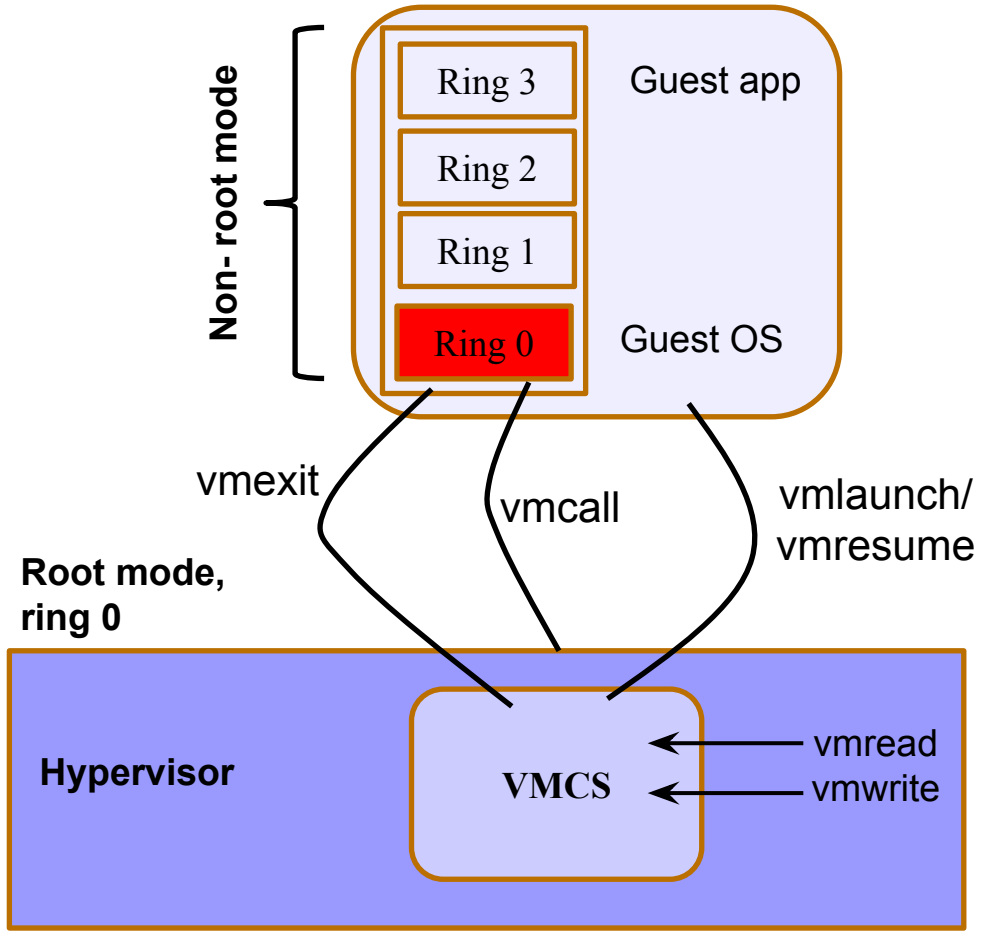
Root and non-root modes (Intel VT-x/AMD-v)

- The new mode **duplicates the state** of the processor (e.g., flags)
- Each mode has its own 64-bit address space and page table
- At any moment, the processor is in one mode and **changing** between modes is HW **atomic**
- In non-root mode, sensitive instructions either change just the non-root state or trap to root-mode
- **Popek & Goldberg criteria are met!**



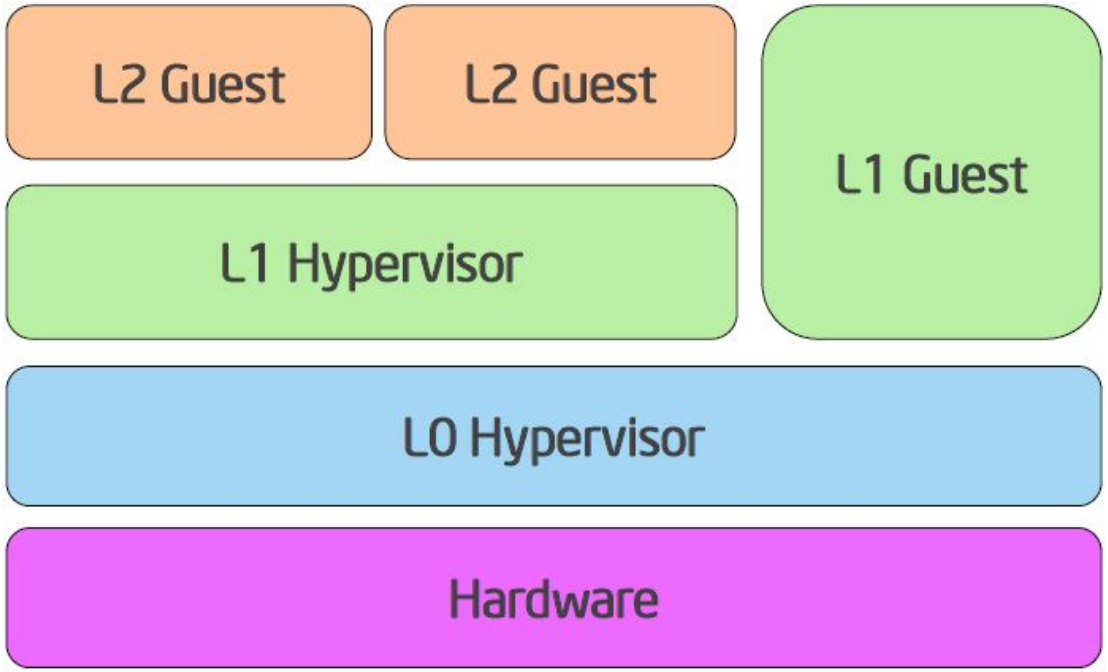
Changing between modes (Intel VT-x)

- The **hypervisor initializes/modifies VMCS** (Virtual Machine Control Structure), containing the VM state
- The hypervisor executes a vmlaunch/vmresume to run/resume the VM (loads the current processor state from the VMCS)
- The VM runs in non-root until it traps or an interrupt occurs (vmexit transition, stores the current processor state into the VMCS)
- The hypervisor analyzes the reason for vmexit and takes action on it
- The VM can explicitly make an hypercall (vmcall)
- The Hypervisor has access to the VMCS via vmread and vmwrite instructions
- These instructions are extensions to the x86 ISA



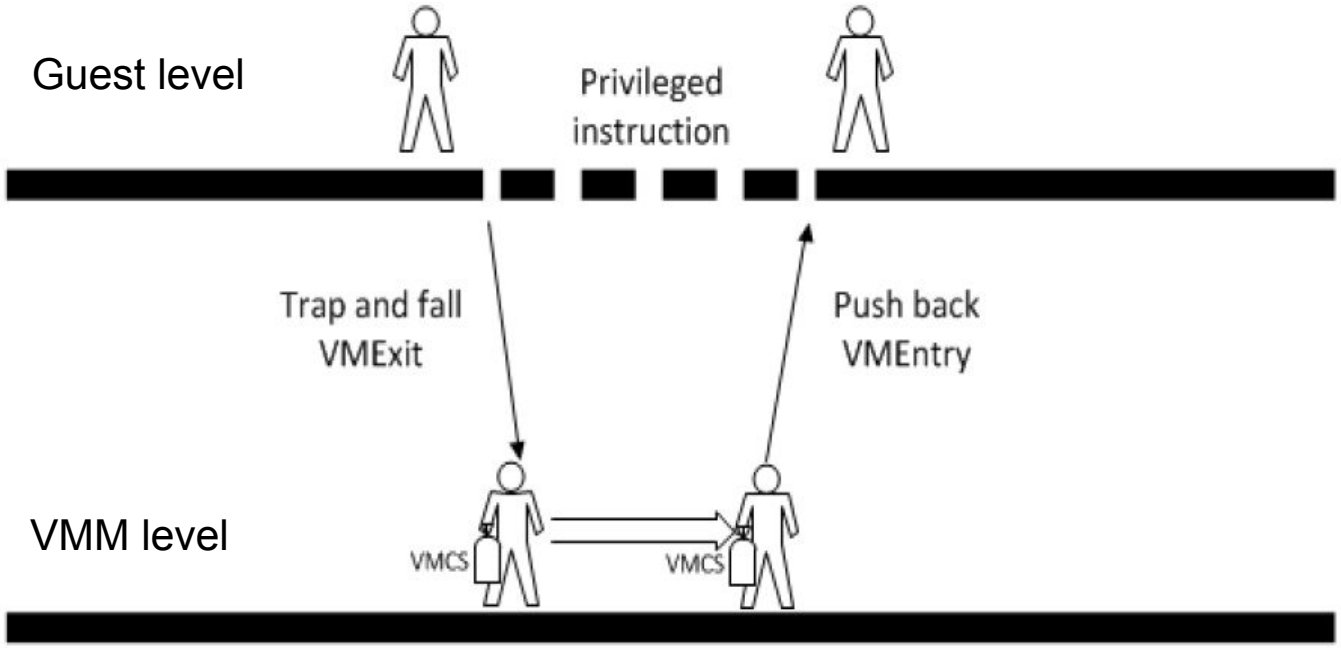
Nested virtualization

- Concept: running virtual machines inside virtual machines
- Main usefulness: testing or developing a full-blown virtual environment with isolation (in a virtual machine, without requiring a physical machine)



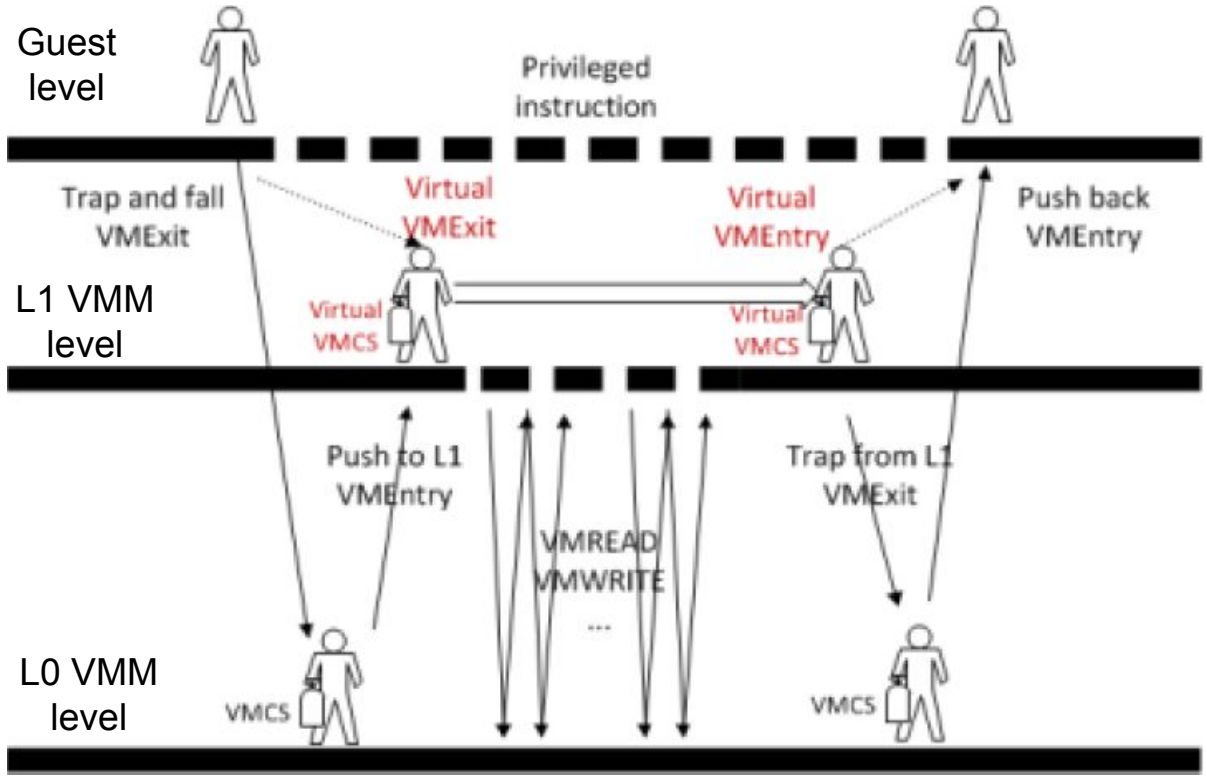
Non-nested (one-level) virtualization

- Guest vCPU executes privileged instruction
- Guest traps into VMM (VM exit)
- VMM emulates the instruction on behalf of guest
- VMM updates guest EIP and goes back to guest (VM entry - vmresume)
- Guest vCPU continues running
- The VMCS (Virtual Machine Control Structure):
  - Stores guest state
  - Controls VMExit/VMEntry behavior
  - Is stored in on-chip memory
  - Optimizes VM Exit/Entry



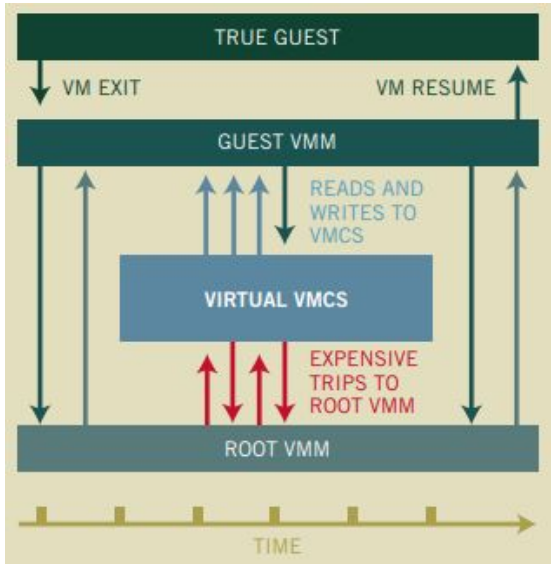
Trap-and-emulate nested virtualization

- L1 VMCSs are virtualized (software-based)
- Guest vcpu executes a privileged instruction
- Guest traps into L0 VMM (VM exit), not L1 VMM
- L0 VMM checks the status, prepares the virtual VMCS, and performs the VM entry into L1 VMM
- L1 VMM emulates the instruction
- When finished, L1 VMM calls vmresume, trying to get back to guest
- L0 VMM gets the VM exit (due to vmresume executed in non-root mode), and issues the real vmresume (virtual VM Entry)
- Guest vcpu continues running
- Each time the L1 VMM accesses the virtual VMCS (vmread and vmwrite are root-mode instructions), a trap occurs!
- Many VM exits and VM entry transitions

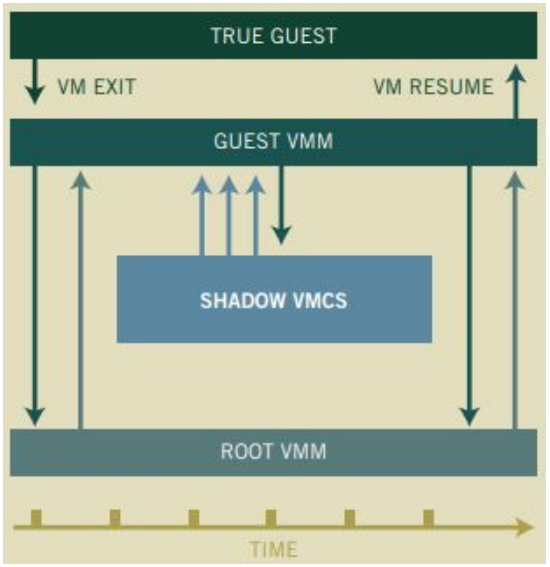


Trap-and-emulate vs VMCS shadowing

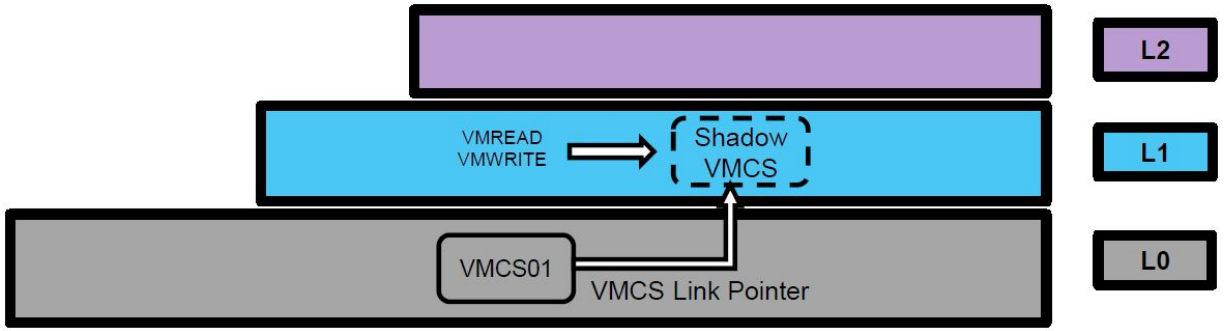
- Guest VMMs cannot use hardware support for virtualization directly.
- Must go through **L0 VMM**, which **emulates** the hardware for the **VMMs in the levels above**
- This emulation can cause many VM Exits and VM Entry, between the guest and the root VMMs
- Each **vmread/vmwrite to the virtual VMCS** by the guest VMM entails a **trap** to the L0 VMM
- The virtual (L1) and physical (L0) VMCSs need to be synchronized when changes occur
- **Hardware support for virtual VMCSs** was introduced in the Haswell architecture (2013)
- It allows L0 to define virtual (**shadow) VMCSs** in hardware, which can be accessed by L1 in a controlled manner, **without traps**!
- **Synchronization** between L1 and L0 VCMSs is **still needed**



Trap-and-emulate nested virtualization



VMCS shadowing nested virtualization

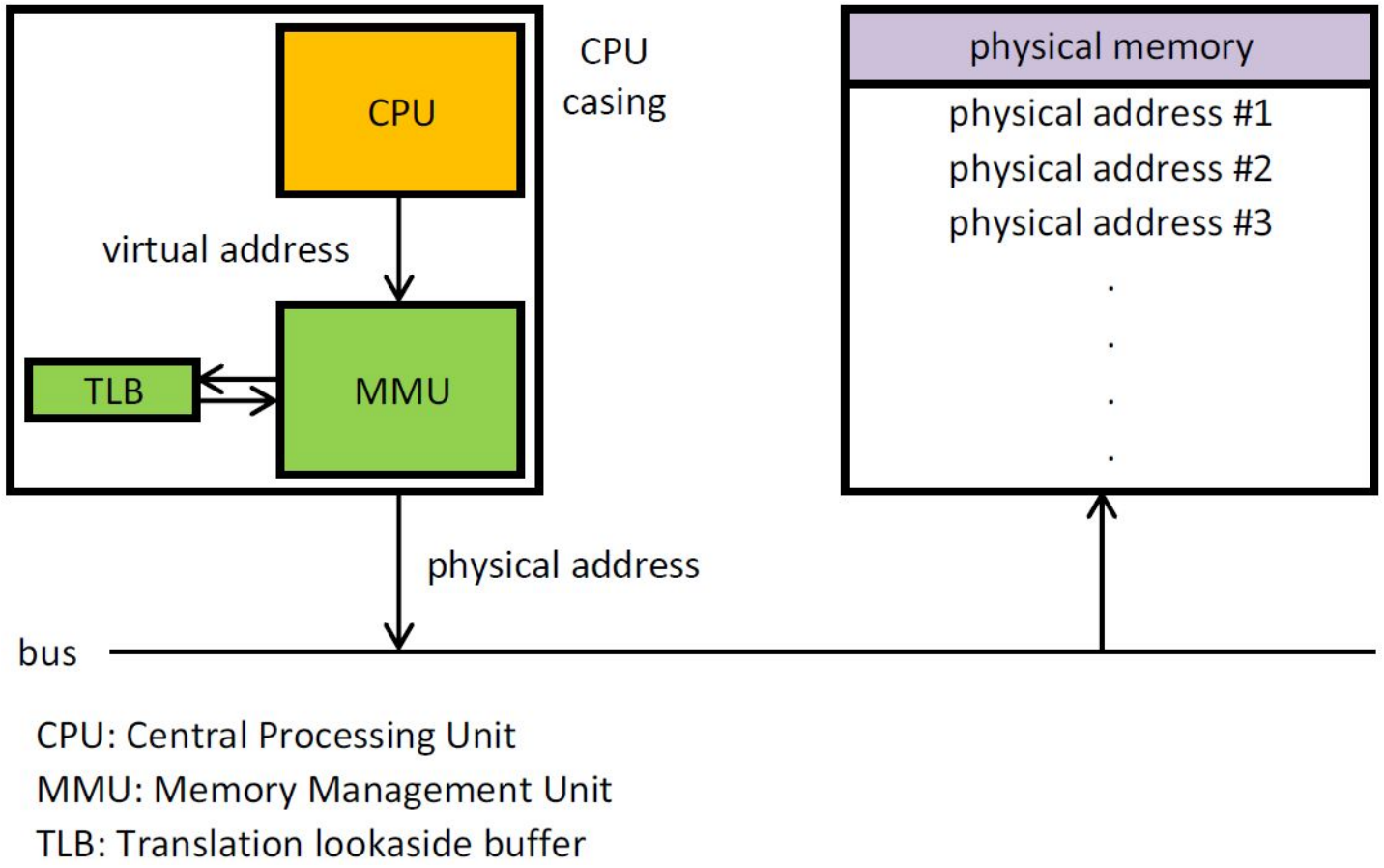


Agenda

- Virtual machines
- Hypervisors
  - Full-virtualization
  - Paravirtualization
  - Hardware-assisted virtualization
- Virtualization of virtual memory
- I/O virtualization
- Virtual machine migration
- Automation of VM creation
- Containers
- MicroVMs and Unikernels

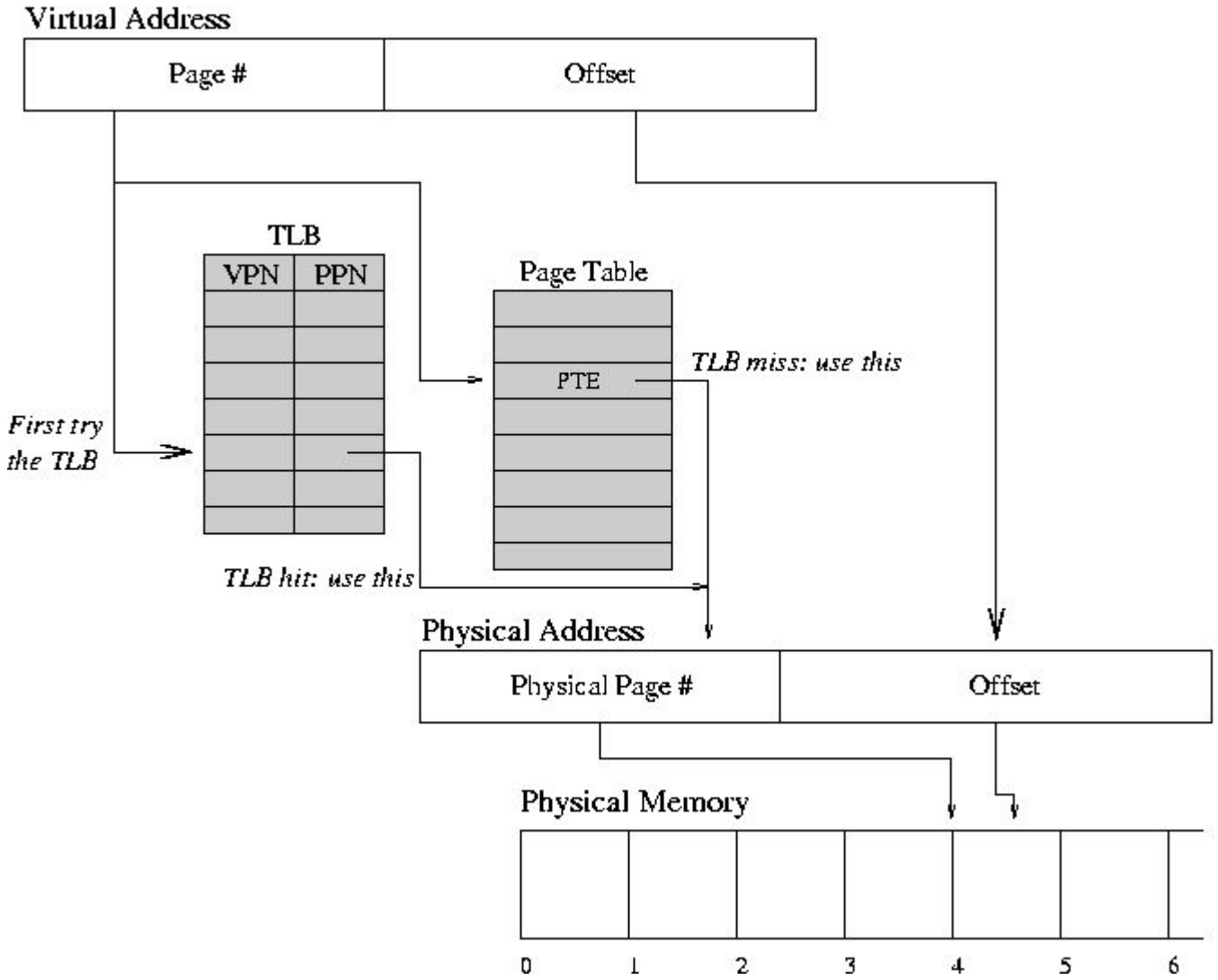
x86-64 virtual memory (no virtualization)

- Applications operate on virtual memory
  - virtual addresses
- Memory management unit (MMU) is responsible for converting virtual addresses into physical addresses
- a TLB caches the conversion for performance



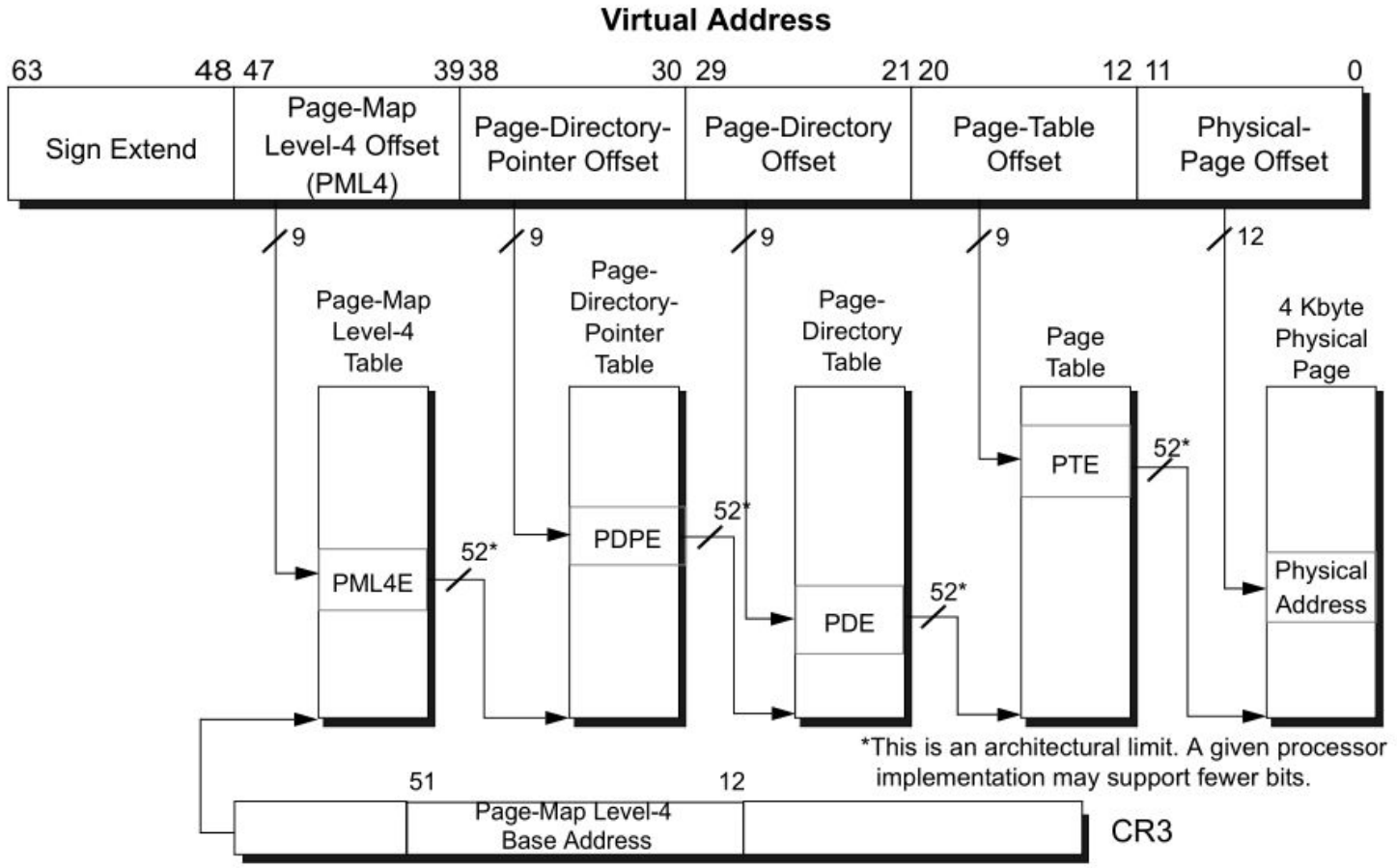
x86-64 virtual memory (no virtualization)

- Virtual addresses are split into two parts:
  - page number/index
  - offset
- Page number is converted into a physical page number
- Offset is appended to the physical page number
- TLB works as a fast (hardware) conversion table



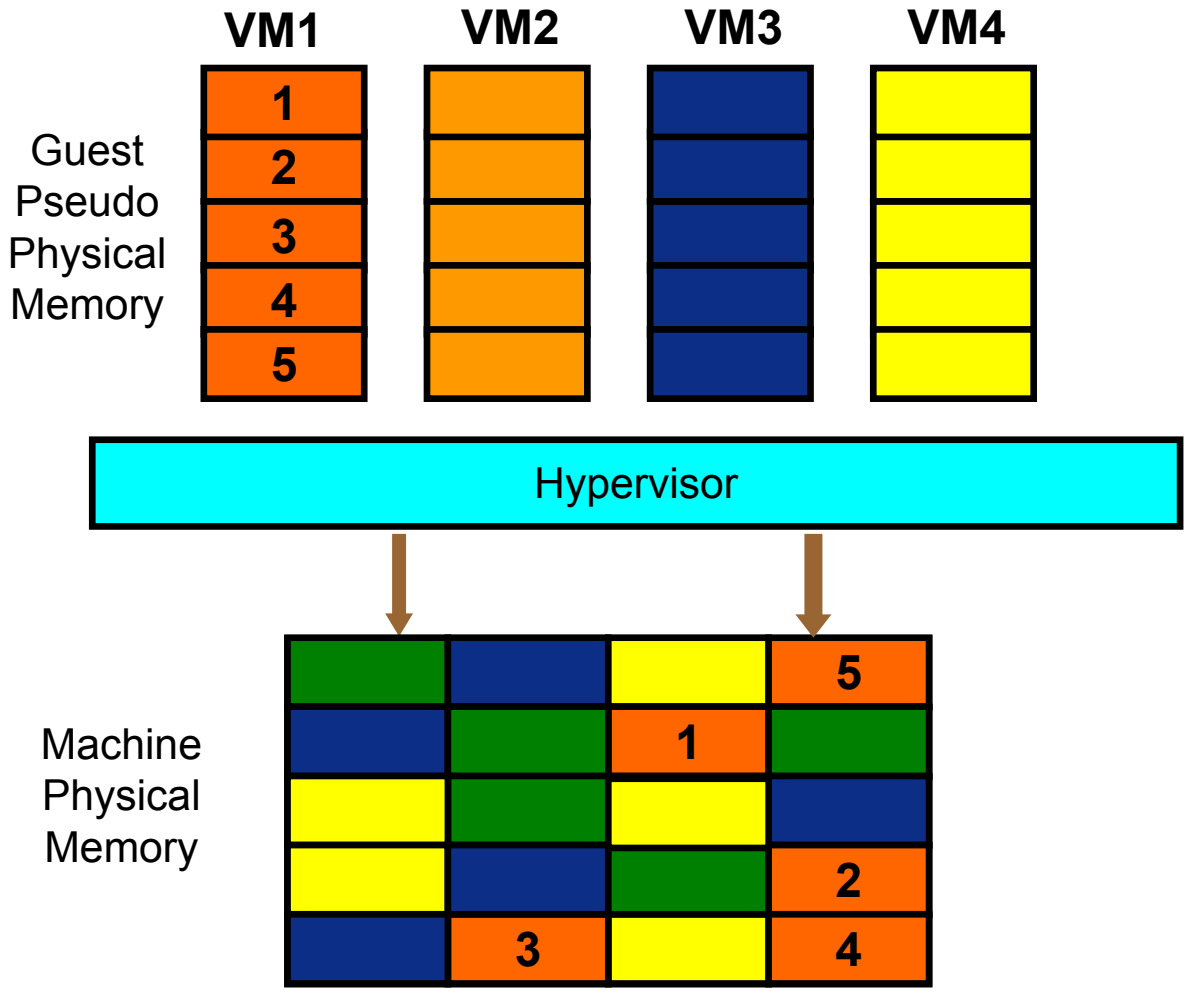
x86-64 virtual memory (no virtualization)

- 48-bit virtual addresses
- 4-level page table tree
- CR3 – register that points to the root page
- Initially, a memory access involves walking the page table tree (4 extra memory accesses)
- TLBs (Translation Lookaside Buffers) are essential to speed this up
- Each virtual address space (process) has its own table tree

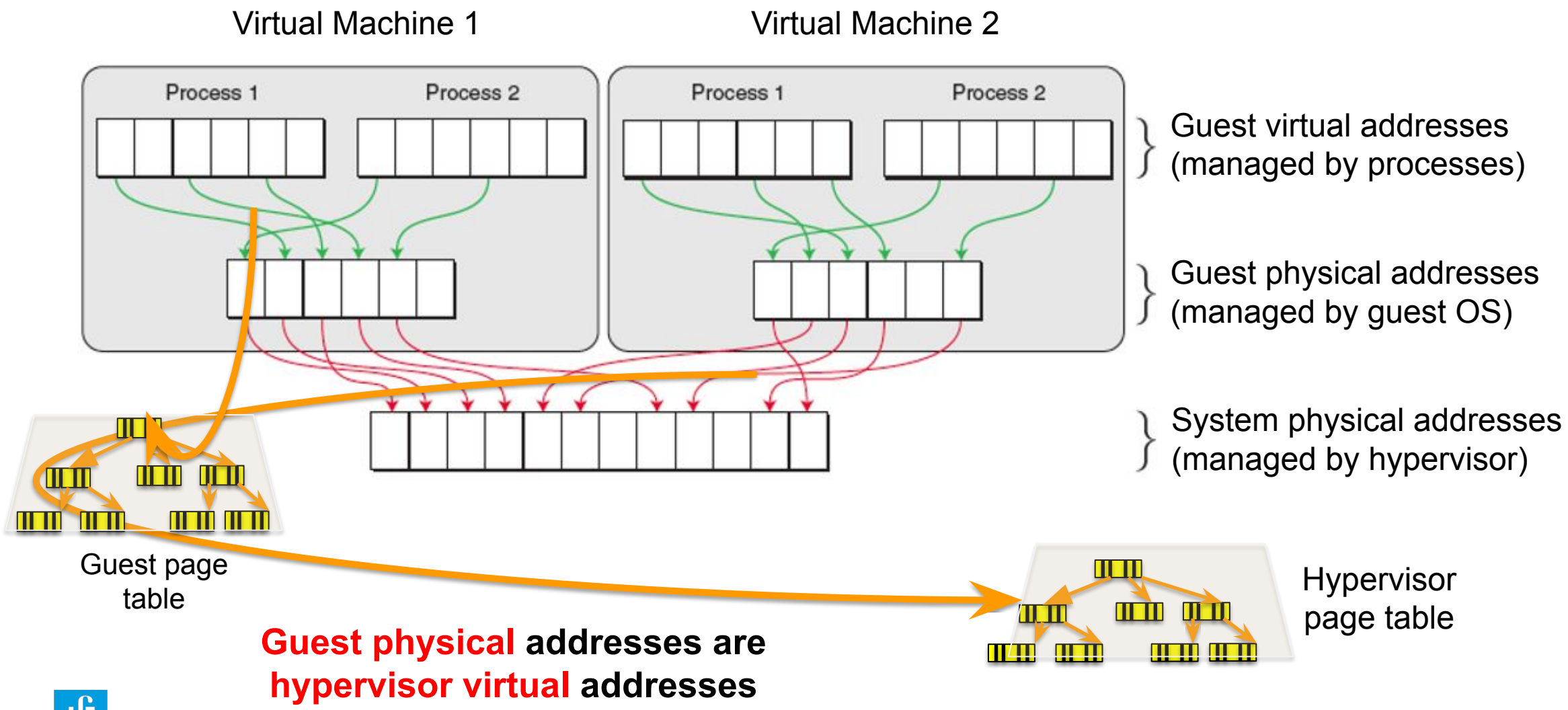


Virtualizing virtual memory

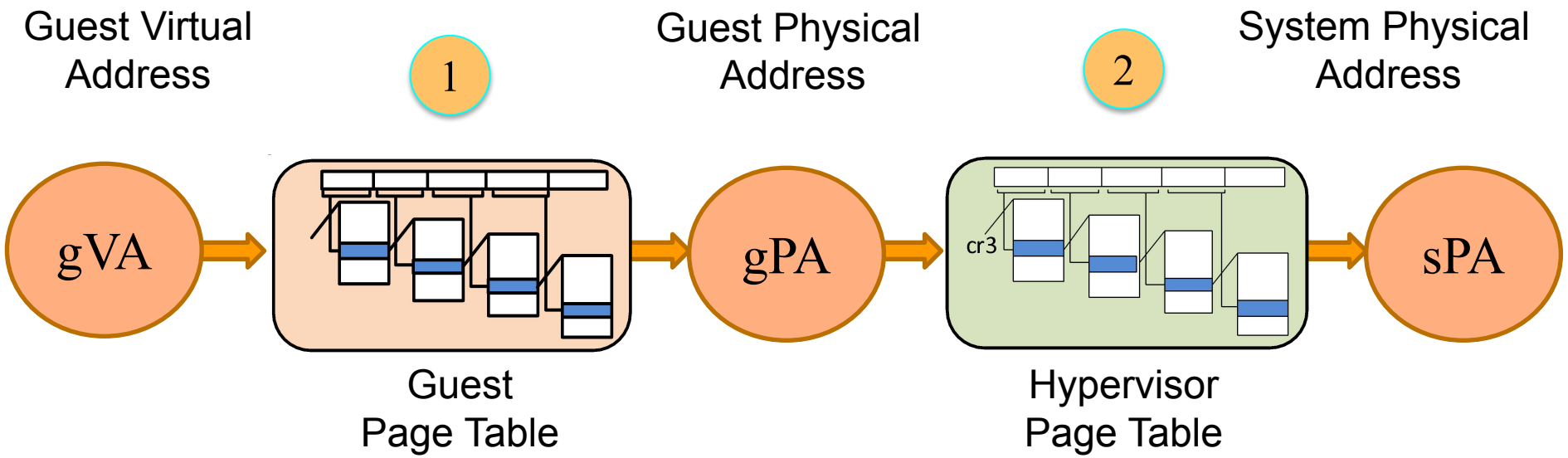
- Virtual memory works by mapping virtual to physical addresses (page tables)
- All modern x86 CPUs include a memory management unit (**MMU**) and a translation lookaside buffer (**TLB**) to optimize virtual memory
- The MMUs must be virtualized (each VM sees its own “physical” memory)
- The **real physical memory** is **dynamically allocated** to VMs



Virtualizing virtual memory

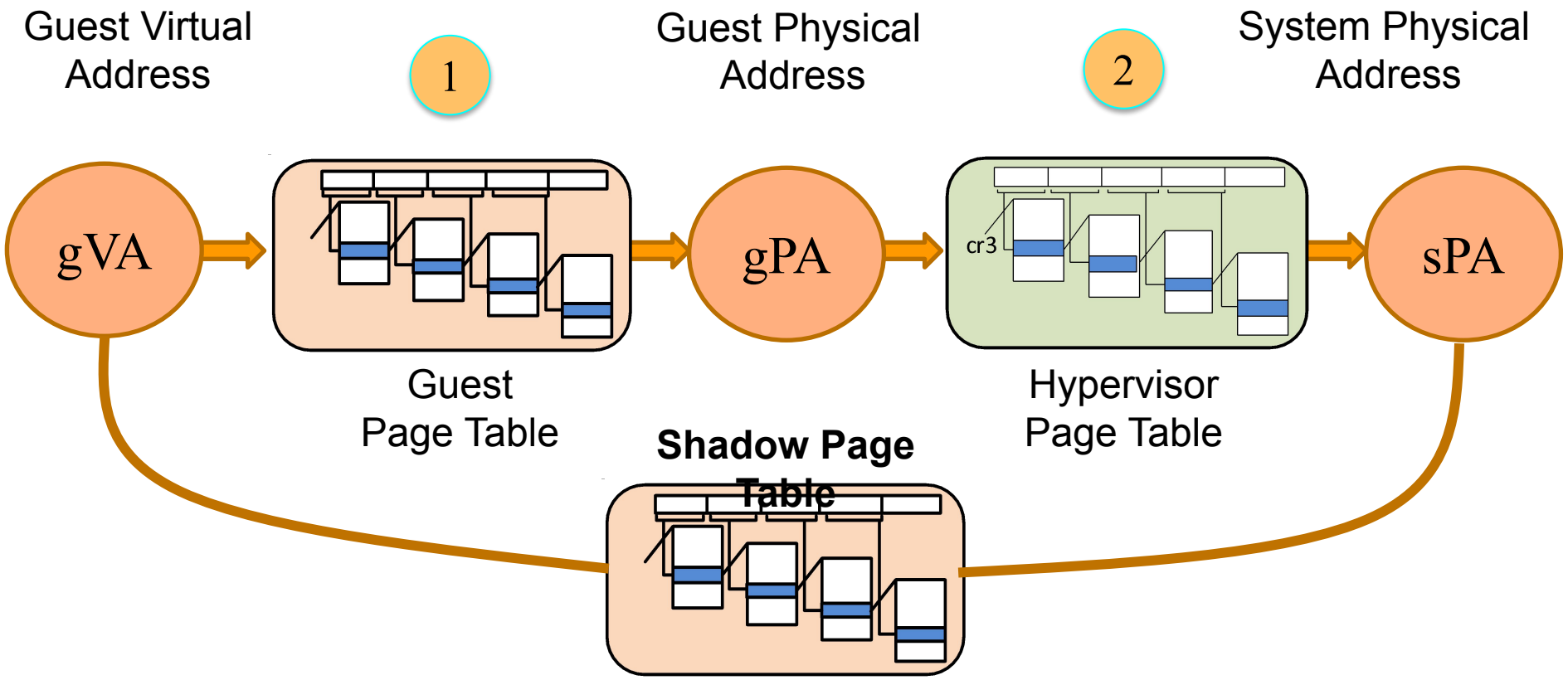


Two-level address translation



- There are two levels of address translation in each VM memory access
- CR3 – register in x86 points to the directory page
- Hypervisor page table is conceptually an extension of the guest page table
  - Shadow page table (software-based solution)
  - Extended (Intel) / Nested (AMD) page table (hardware-assisted solution)

Shadow page table



- Guest CR3 – register points to the shadow page table
- Maps a guest virtual address to a system physical address directly

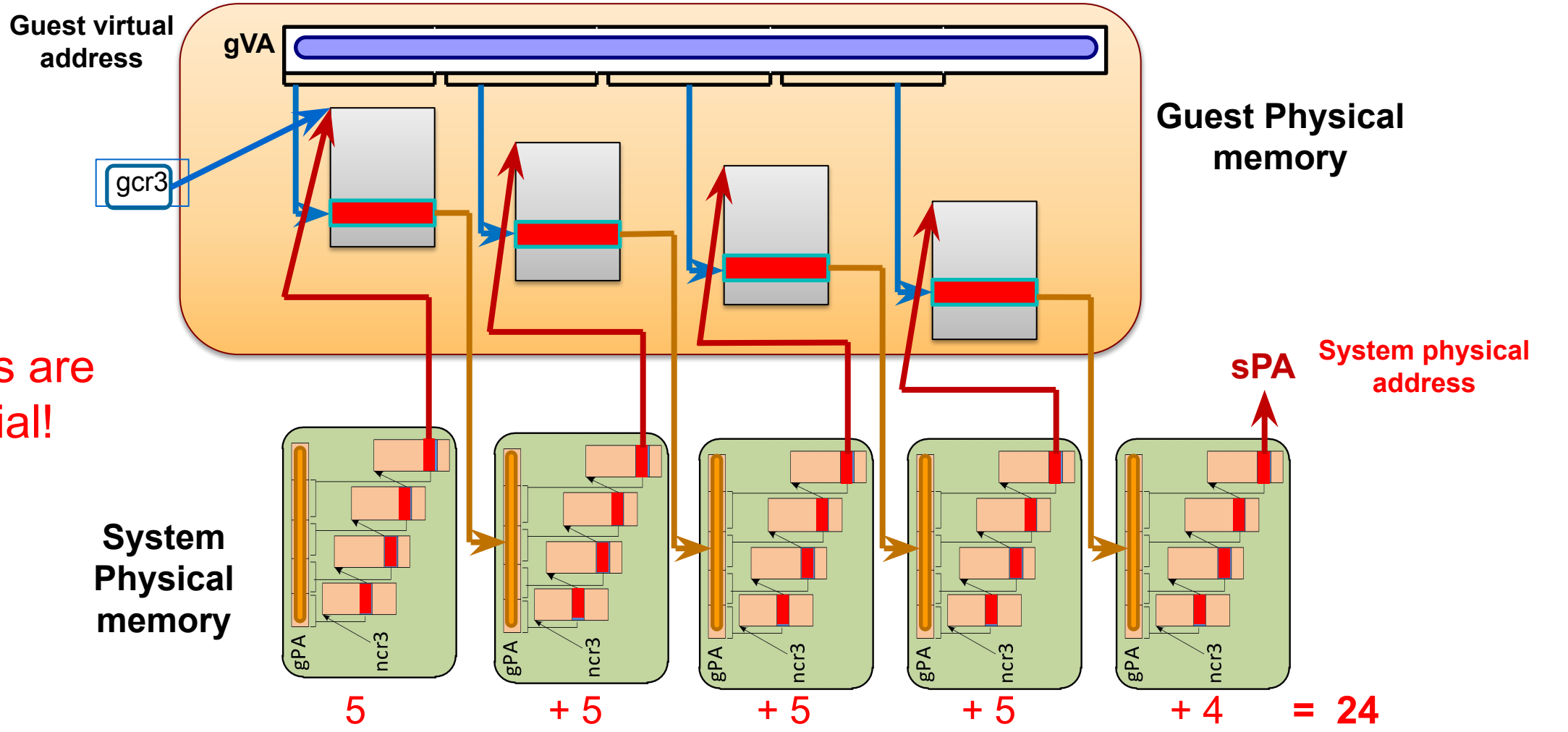
Problems with shadow page tables

- Anytime the **guest OS modifies** the guest page table, the hypervisor needs to **update the shadow page table**
  - Solution: Write protect guest page tables
  - Any write access to the guest page table generates a page trap to the hypervisor
  - Drawback: **Many page traps** for applications that modify page tables
- For **every guest application there is one shadow page table**
- Every time the guest application **context switch** occurs, a trap to the hypervisor is needed to change cr3 to point to the new shadow page table
- **Software-based solution, poor performance**

Extended (Intel) / Nested (AMD) page table

- Make **hardware** aware of **two levels** of address translation: guest and nested page table
- Make hardware page walker walk both guest page table and nested page table on a TLB miss (**two dimensional page walk**)
- Two cr3 registers:
  - **gCR3**: points to guest directory page
  - **nCR3**: points to nested directory page
- Eliminates the need of shadow page table
- Same table for all guest processes
- Problem: **latency** with TLB misses (lengthy two dimensional page walk)
  - Caches minimize the problem
  - Modern processors have hardware support for extended/nested page tables

Two dimensional page walk in hardware



- TLBs are crucial!

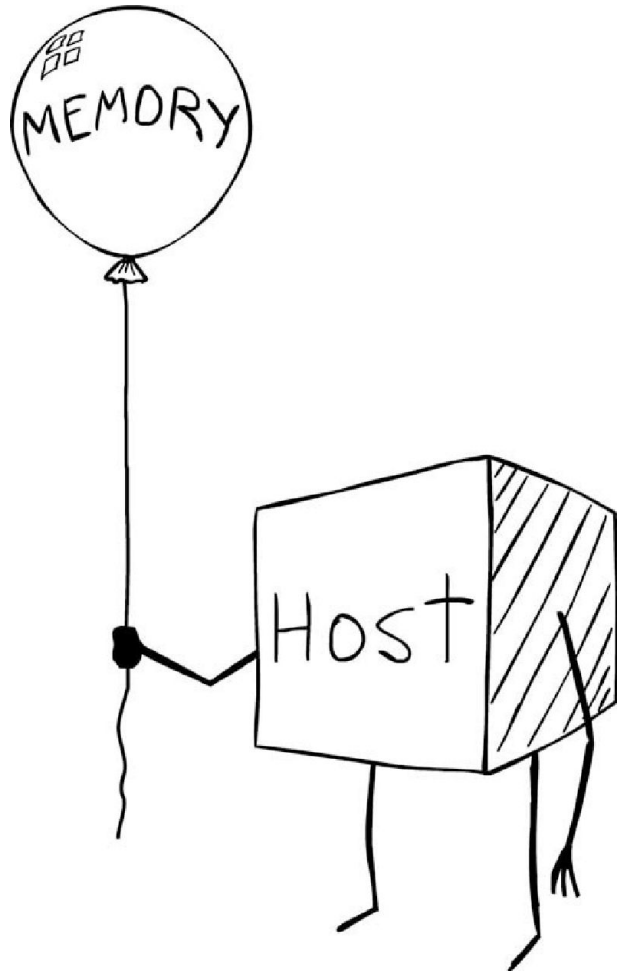
Maximum possible memory accesses for one two dimensional page walk

Memory management with virtualization

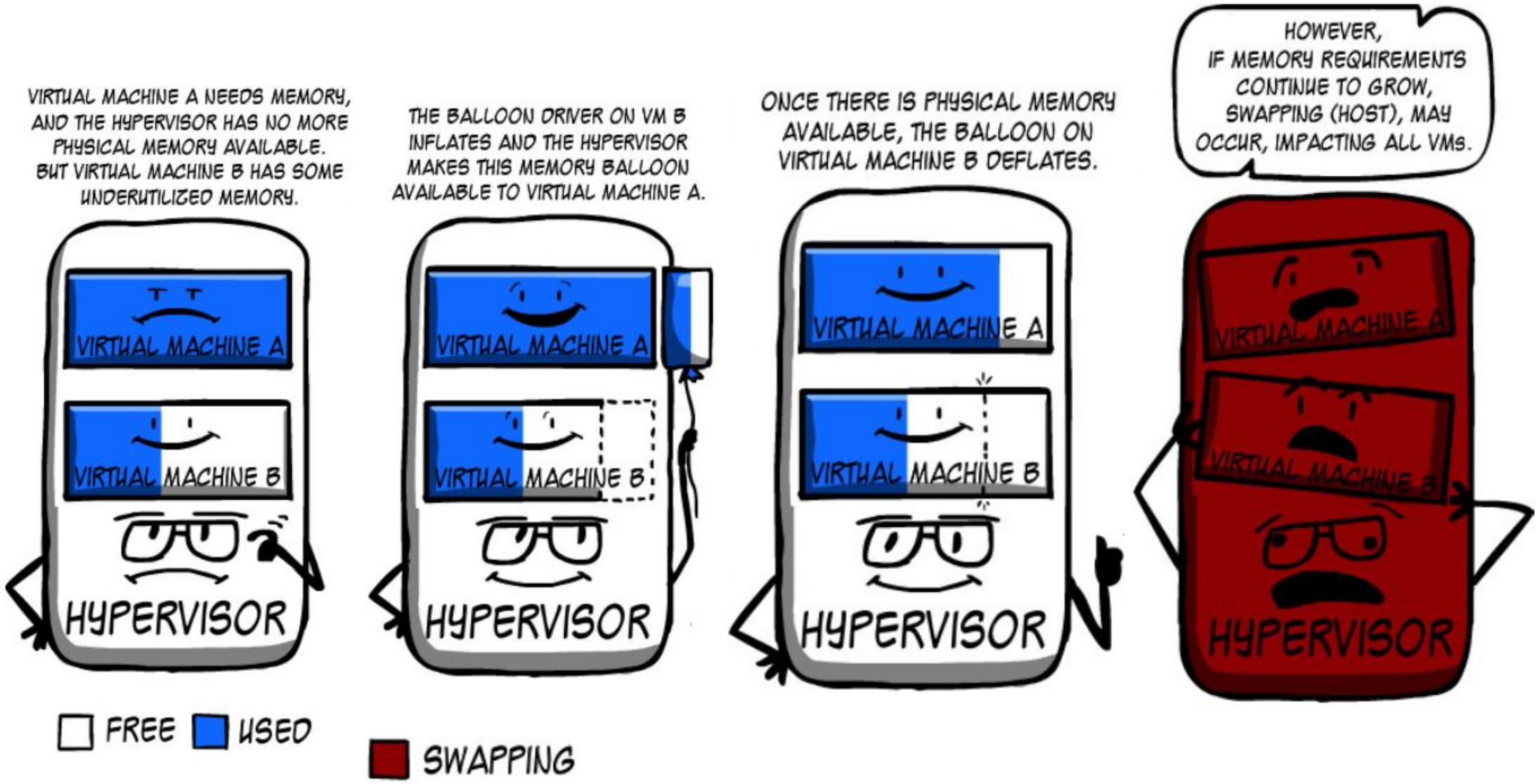
- The memory of a **physical** host has to be **shared** by **VMs** and other eventual host applications
- Various **techniques** need to be tuned up and **dynamically managed** (just like conventional virtual memory needs to do):
  - Deduplication (read-only pages with the same content are shared)
  - Over-commitment
  - Ballooning
  - Guest OS swapping
  - Hypervisor swapping

Memory Over-Commitment

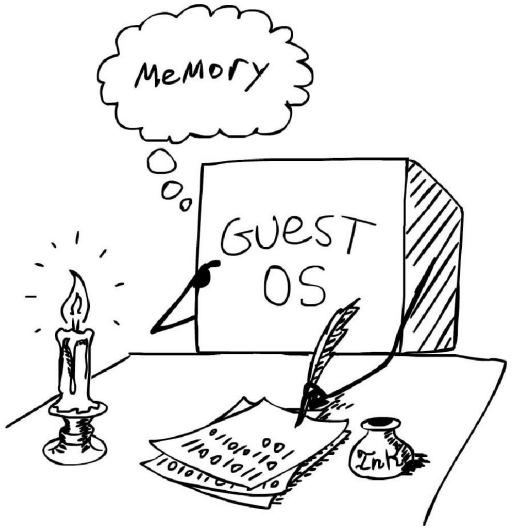
- The hypervisor can allocate more memory than the physical host actually has (e.g., if the host server has **8 GB** of physical memory available, we can provision **4 VMs with 4 GB** allocated each)
- But VMs can use all its memory (or the host can reduce the memory available to the hypervisor)
- The hypervisor can identify and reallocate unused memory from other VM's, using a technique called memory ballooning
- This is implemented by the balloon driver, installed on the guest OS (Operating System).



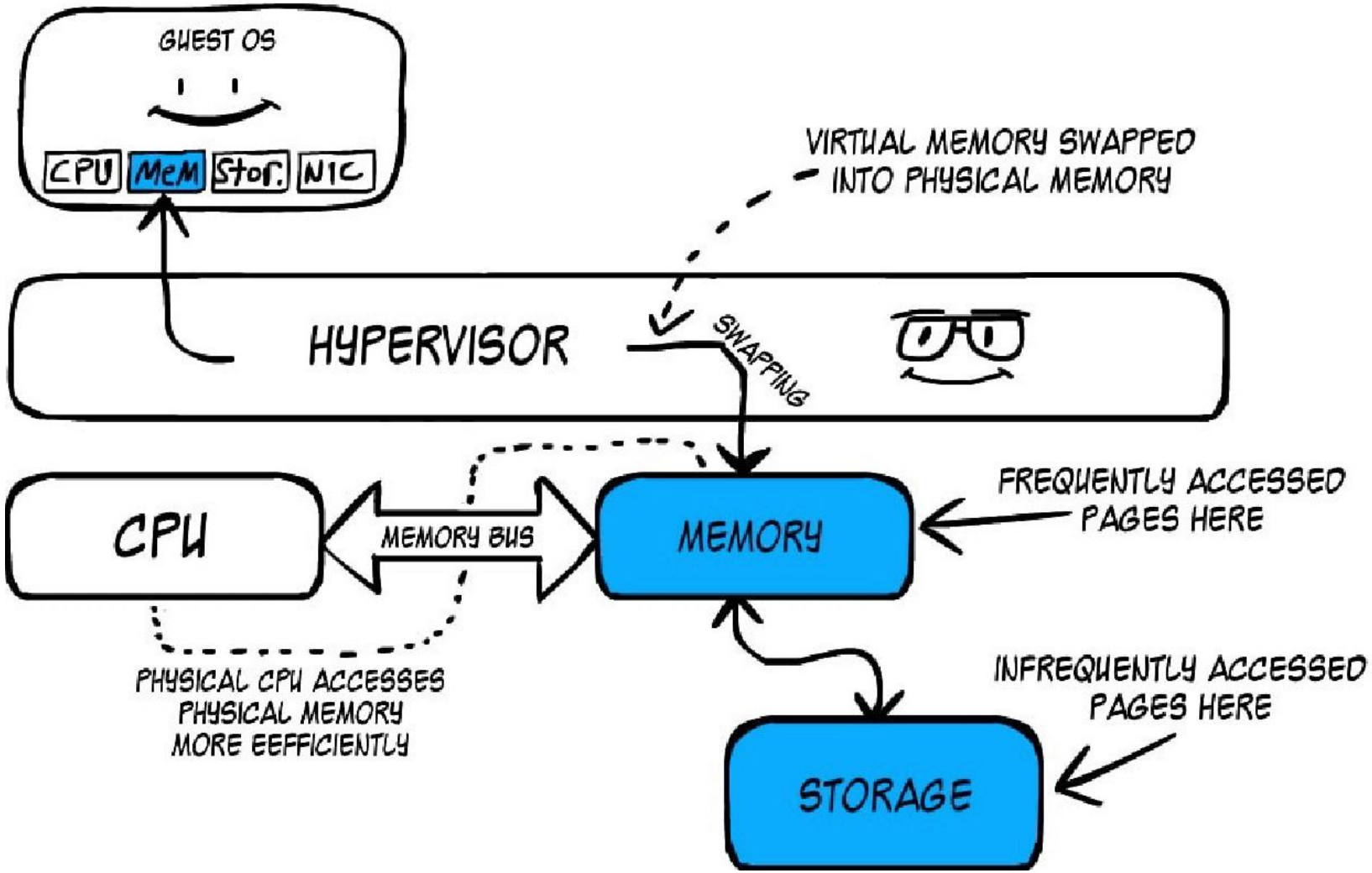
Memory ballooning



Guest OS swapping

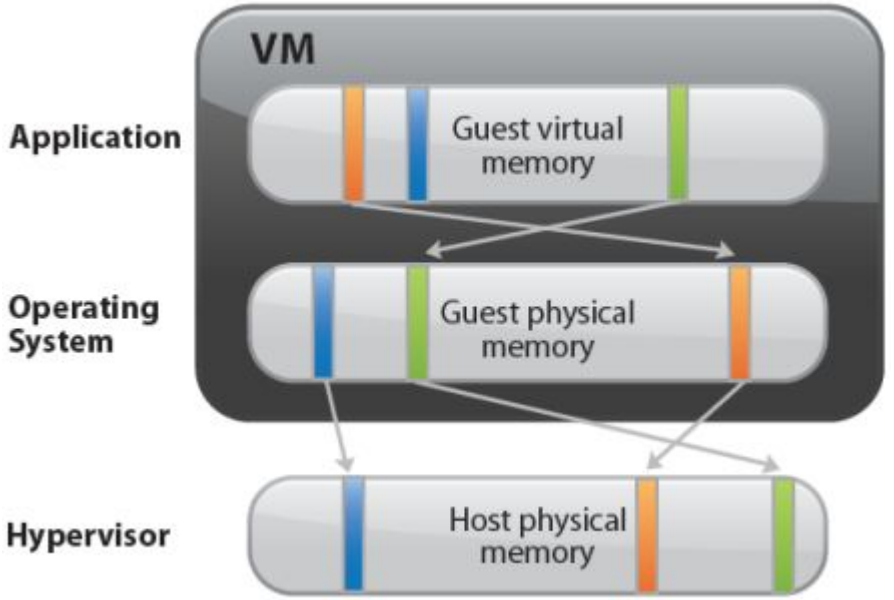


- Guest OS can swap virtual pages into real physical pages, if available, as usual
- The hypervisor can steal them from other VMs

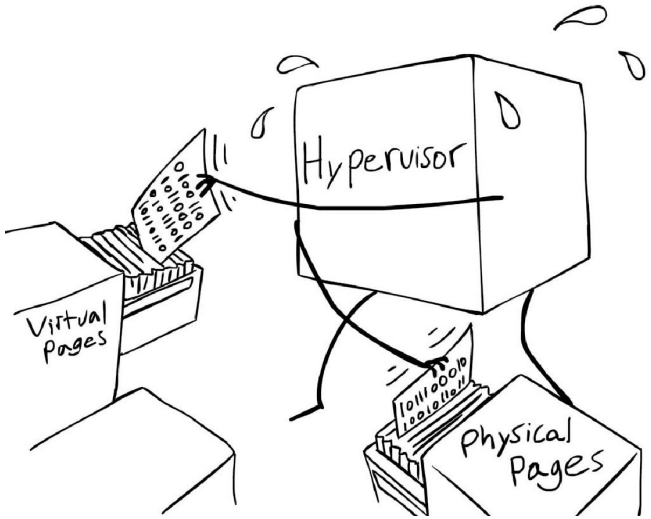


Guest vs hypervisor swapping

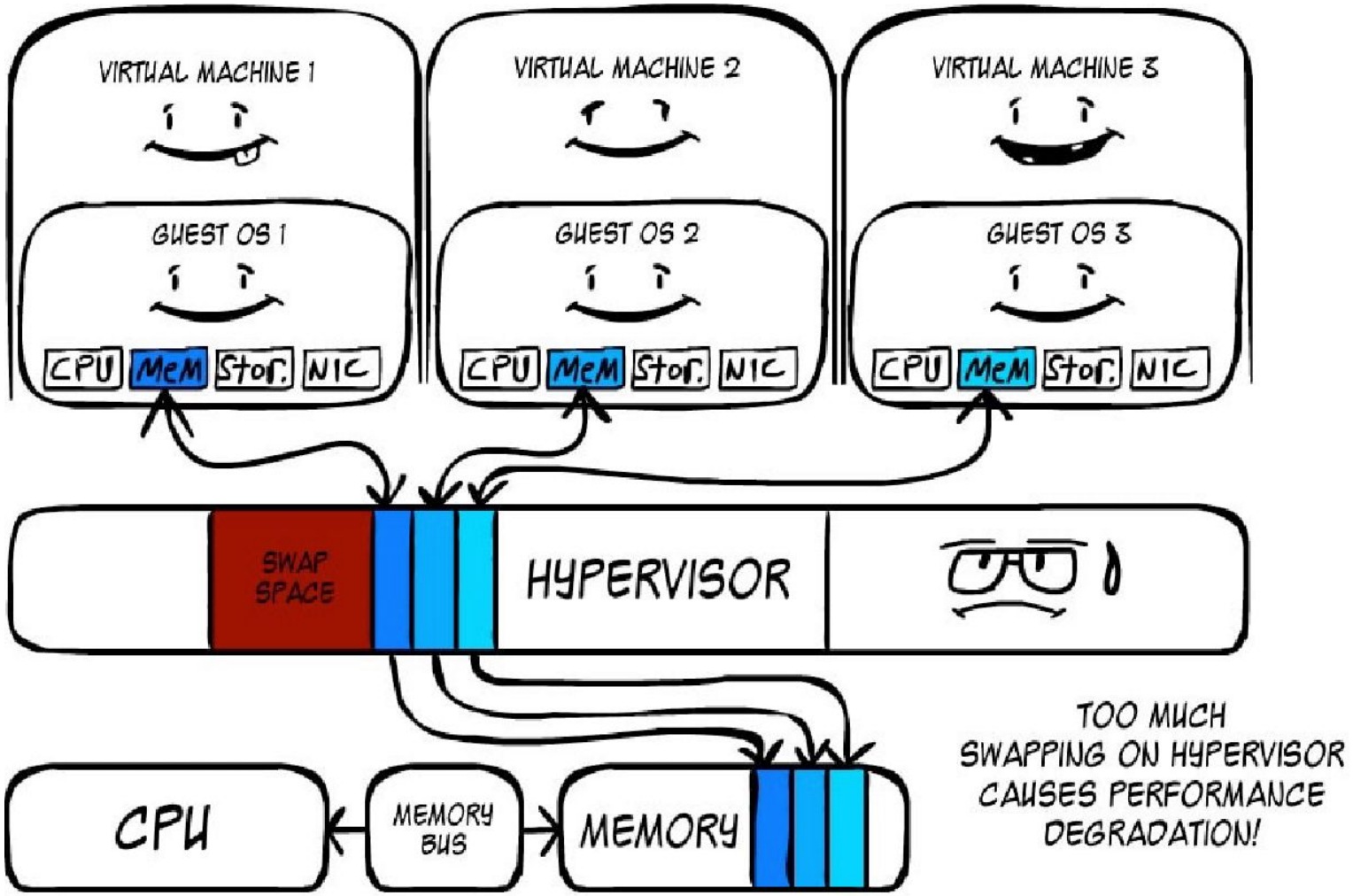
- The guest OS:
  - Does not know that its “physical” memory is not actually physical
  - Assumes it owns all the physical memory in the system
  - Does not tell which “physical” pages are used or free
- The hypervisor:
  - **Knows** easily when the **guest needs a swapped page** back into memory (a page fault occurs)
  - But **it does not know when a page has been freed**
- Each VM has its own swap file
- The hypervisor can use this to kick guest pages to disk, if needed (e.g., due to memory over-commitment), **but does not know which pages are free or not**
- This can hamper performance. Must be tuned.



Hypervisor swapping



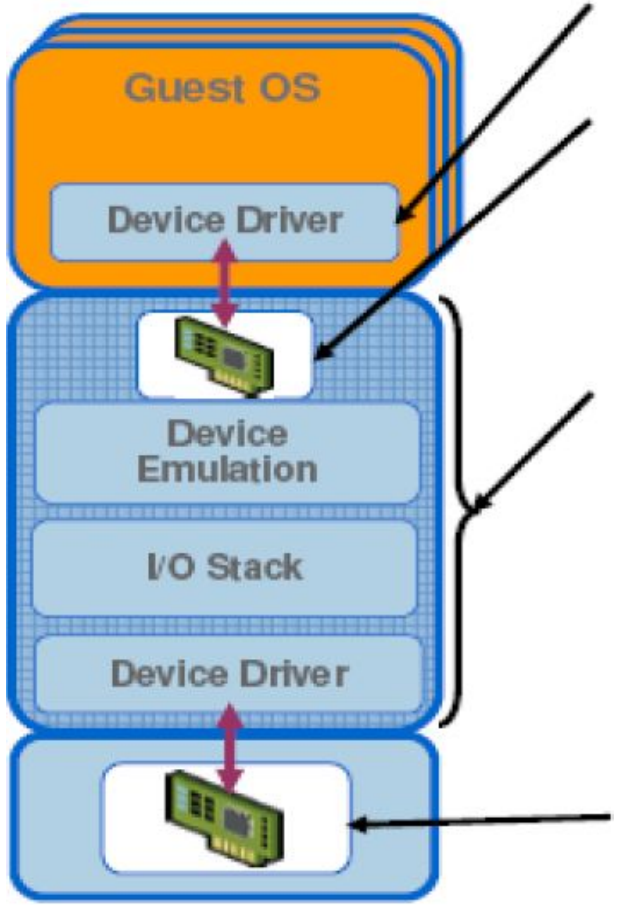
- Swapping at the hypervisor level is blind to actual memory usage
- Can impact even VMs that need no swapping at all
- Should be last resort!



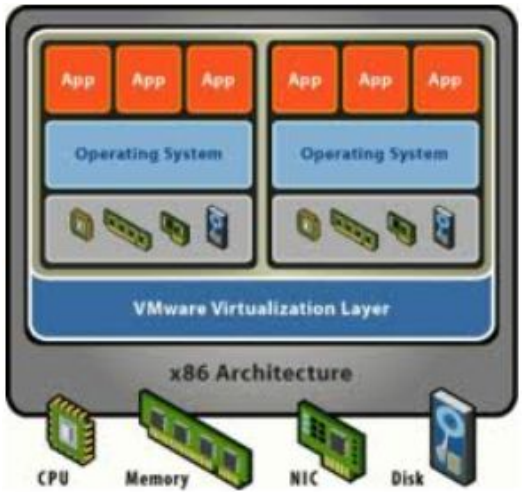
Agenda

- Virtual machines
- Hypervisors
  - Full-virtualization
  - Paravirtualization
  - Hardware-assisted virtualization
- Virtualization of virtual memory
- I/O virtualization
- Virtual machine migration
- Automation of VM creation
- Containers
- MicroVMs and Unikernels

I/O virtualization



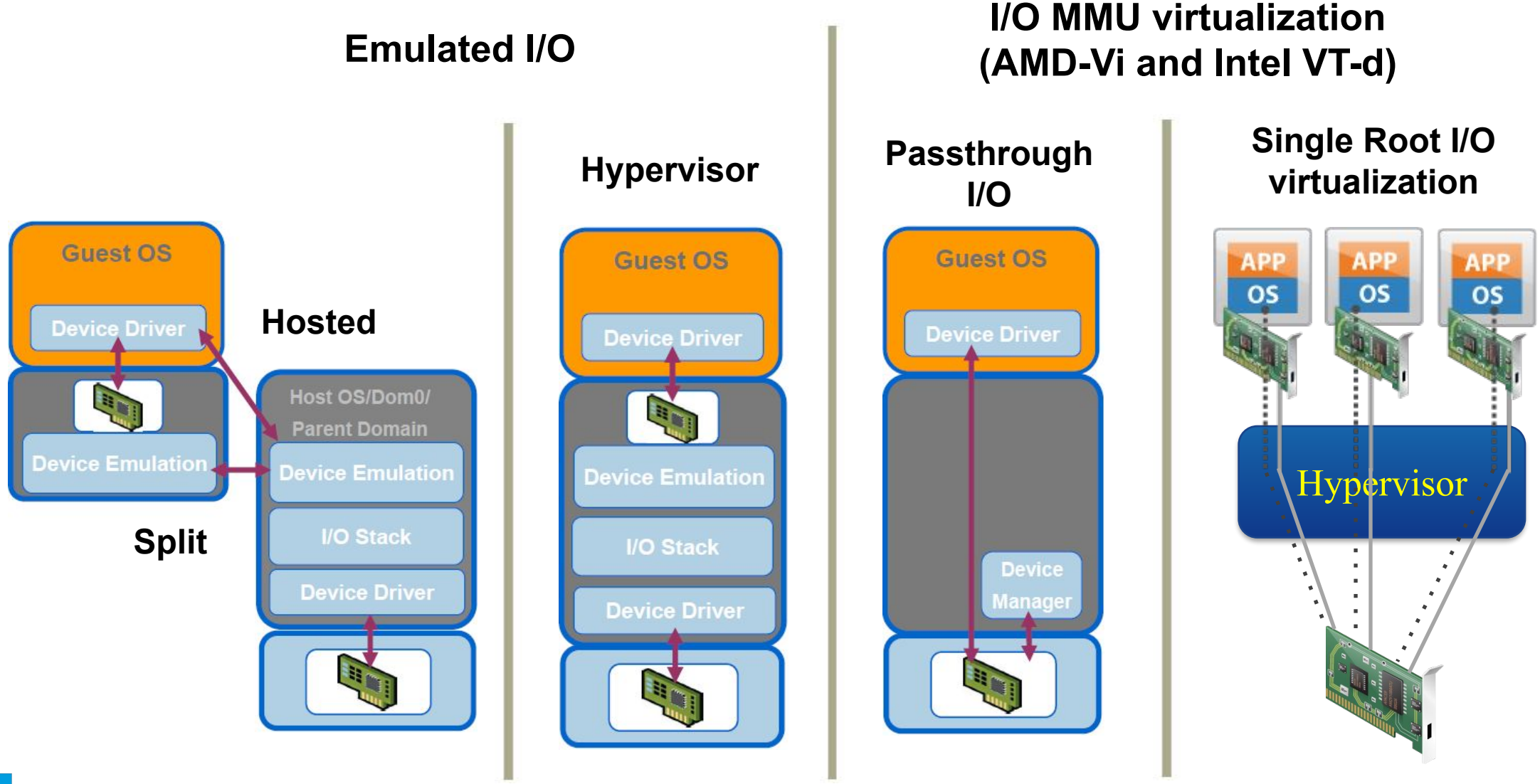
- Guest device driver
- Virtual device
  - Models an existing device, or
  - Models an idealized device
- Virtualization layer
  - Emulates the virtual device
  - Remaps guest and real I/O addresses
  - Multiplexes and drives physical device
  - May provide additional features, e.g., NIC teaming
- Real device
  - Physical hardware
  - Likely to be different from virtual device



I/O issues

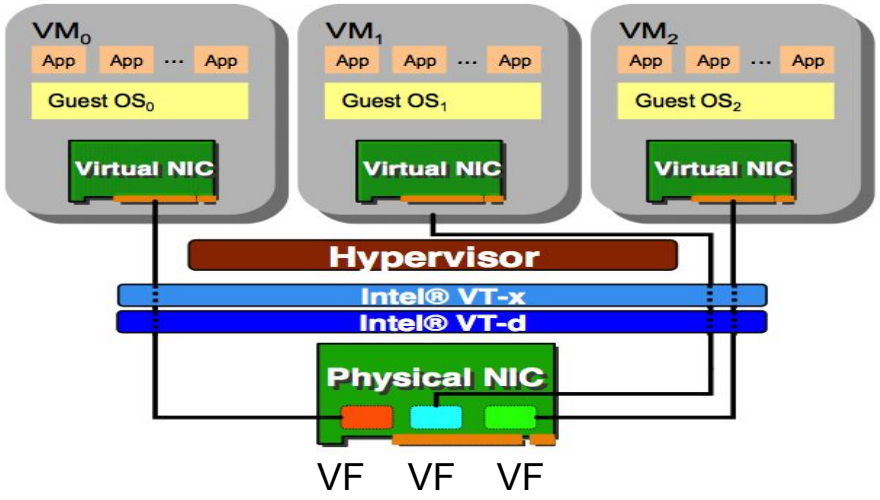
- I/O virtualization (IOV) involves sharing a single I/O resource between multiple virtual machines
- Problems:
  - **Lots of I/O devices** (with diverse characteristics), making virtualization challenging
  - Writing **device drivers** for all I/O devices in the **VMM layer** is **not a practical** option
- Possible solutions:
  - **Hosted**: Channel guest VM I/O requests through a trusted host VM running a popular OS, with already existing device drivers (emulating real devices)
  - **Split**: A front end driver in the guest works in tandem with a back end driver in the VMM.
  - **Hypervisor-based**: Place drivers in the hypervisor itself (for most common devices)
  - **Passthrough**: Link a guest VM directly to a device (cannot be shared)
  - **Single Root-I/O Virtualization**: Use hardware support from standard I/O specifications to virtualize and share physical devices (e.g., PCI-based)

I/O virtualization implementations



Single Root I/O virtualization (SR-IOV)

- PCI-SIG specification:
  - Directly assigned to VMs
  - Hypervisor bypassing
  - Evolution of passthrough I/O
- One Physical Function (PF):
  - Primary function of the device (base to VFs)
  - Advertises the device's SR-IOV capabilities
- One or more Virtual Functions (VF):
  - Lightweight (subset) PCIe functions
  - Share one or more of the device’s resources
  - Assigned to each guest as in passthrough
- Intel VT-x and VT-d:
  - Support for CPU and I/O virtualization



Agenda

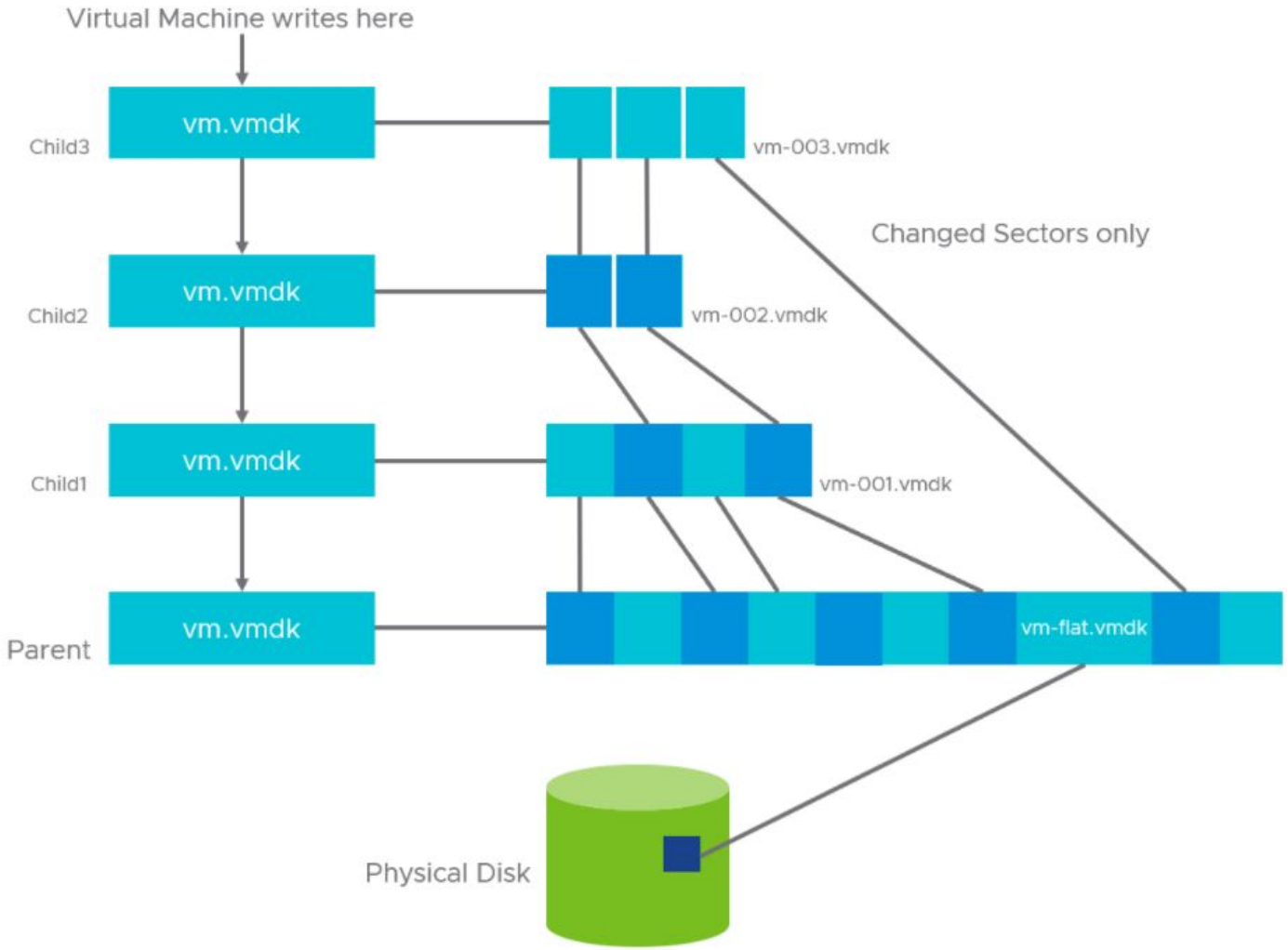
- Virtual machines
- Hypervisors
  - Full-virtualization
  - Paravirtualization
  - Hardware-assisted virtualization
- Virtualization of virtual memory
- I/O virtualization
- Virtual machine migration
- Automation of VM creation
- Containers
- MicroVMs and Unikernels

VM snapshots/checkpoints

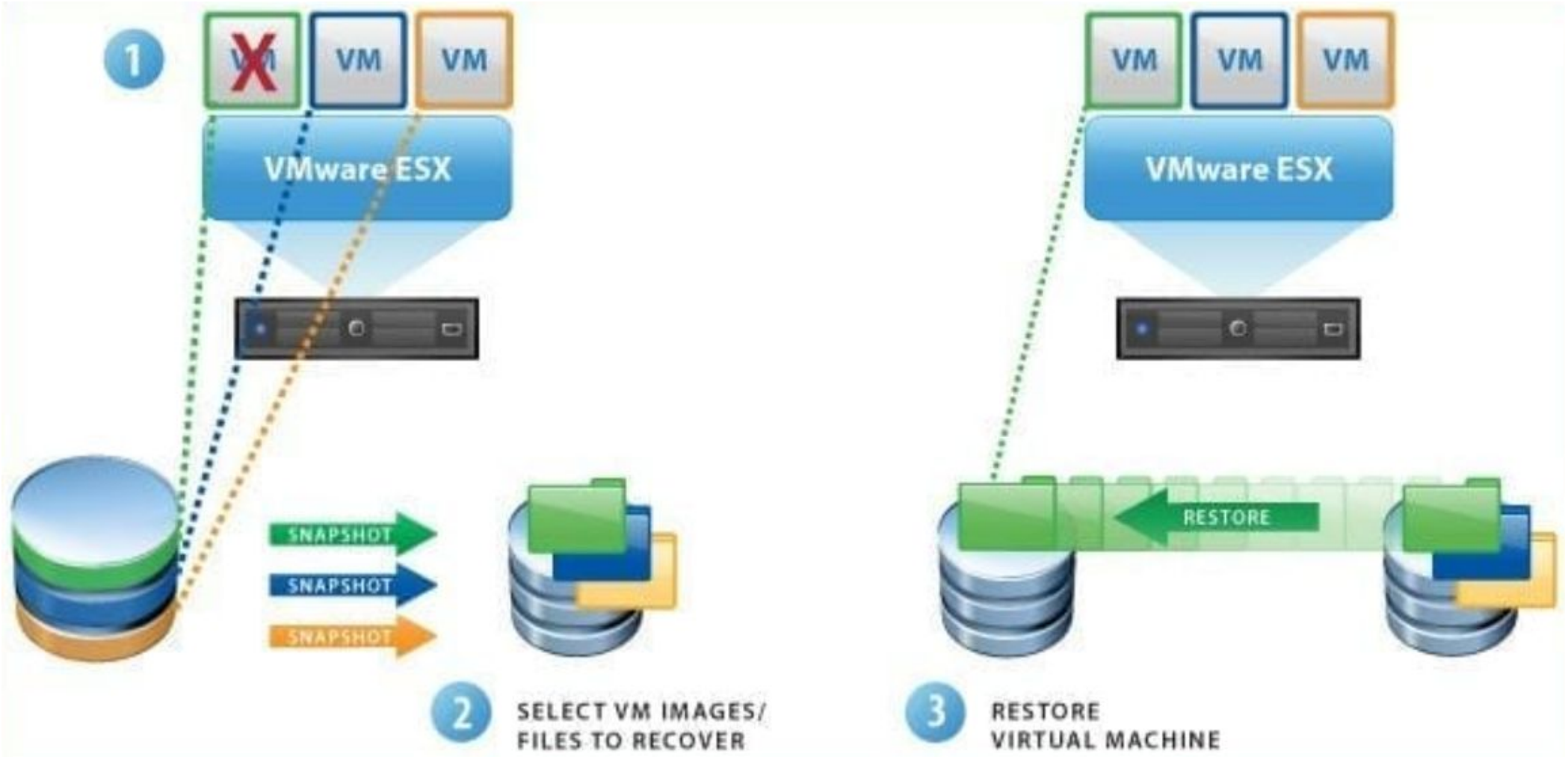
- A VM snapshot persists the state of a virtual machine (VM) at a specified time. It includes:
  - The state of all the virtual machine's **storage**;
  - The contents of the virtual machine's **memory**;
  - The virtual machine execution **state**.
- Can be used for:
  - Restoring the VM to a previous state (similar to the restore points of Windows)
  - Migration of the VM to another server
  - Suspension/resumption of the VM
  - Replication, by creating several VMs from the same snapshot
  - Backup, although not a good practice

Snapshot chain

- When a snapshot is taken, the current state is frozen.
- The snapshot includes a reference to that state and all the changes made subsequently, so it grows!
- The snapshot uses the VM disk
- Running a VM on snapshots has a performance penalty (changes only)
- Chain size should be 2 or 3 max
- Maintaining snapshots active for a long time can lead to instabilities
- Should be used primarily for tests (recovery if needed) or migration
- Bad solution for backup (includes just changes, not the original state)

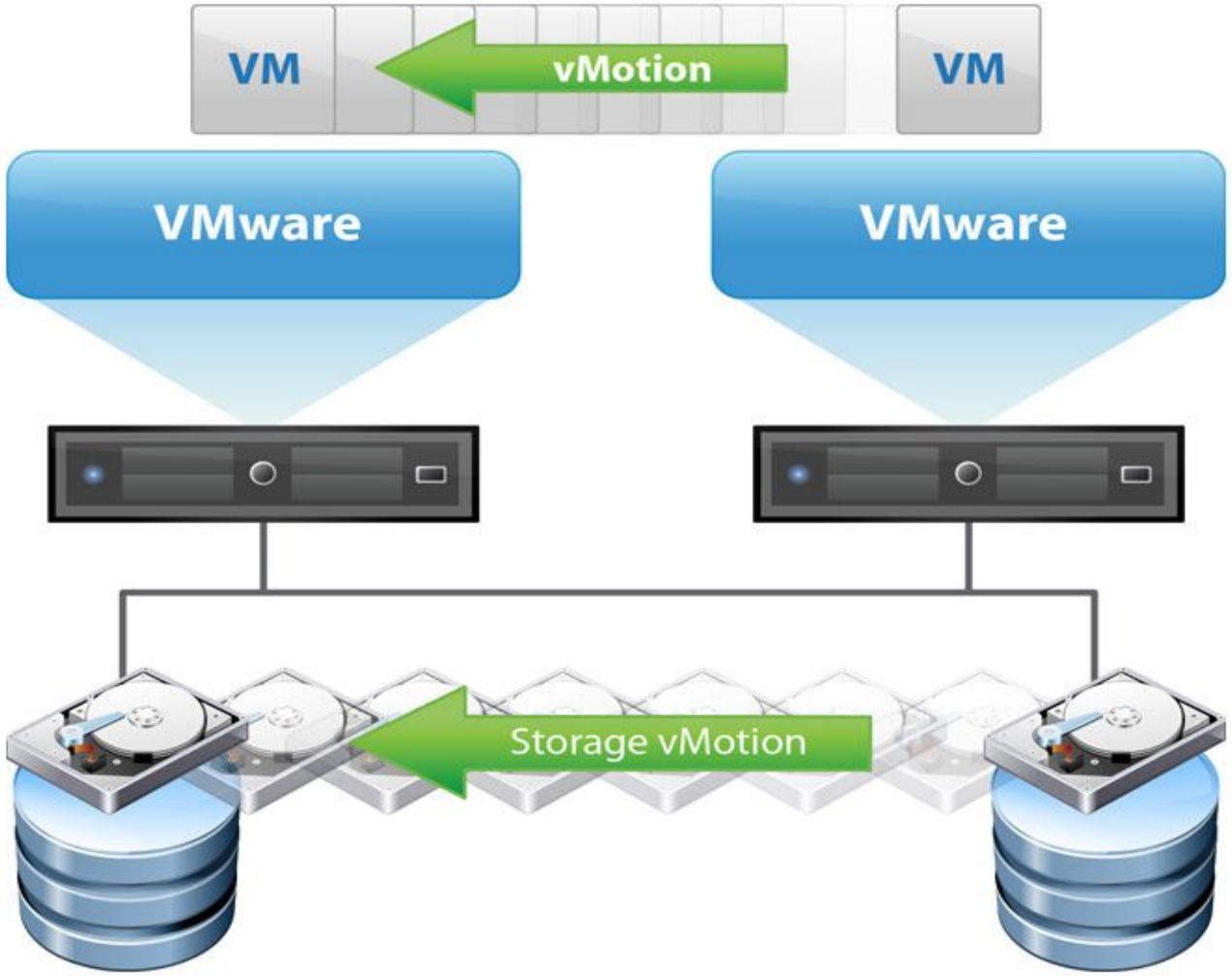


Restoring a VM from a snapshot

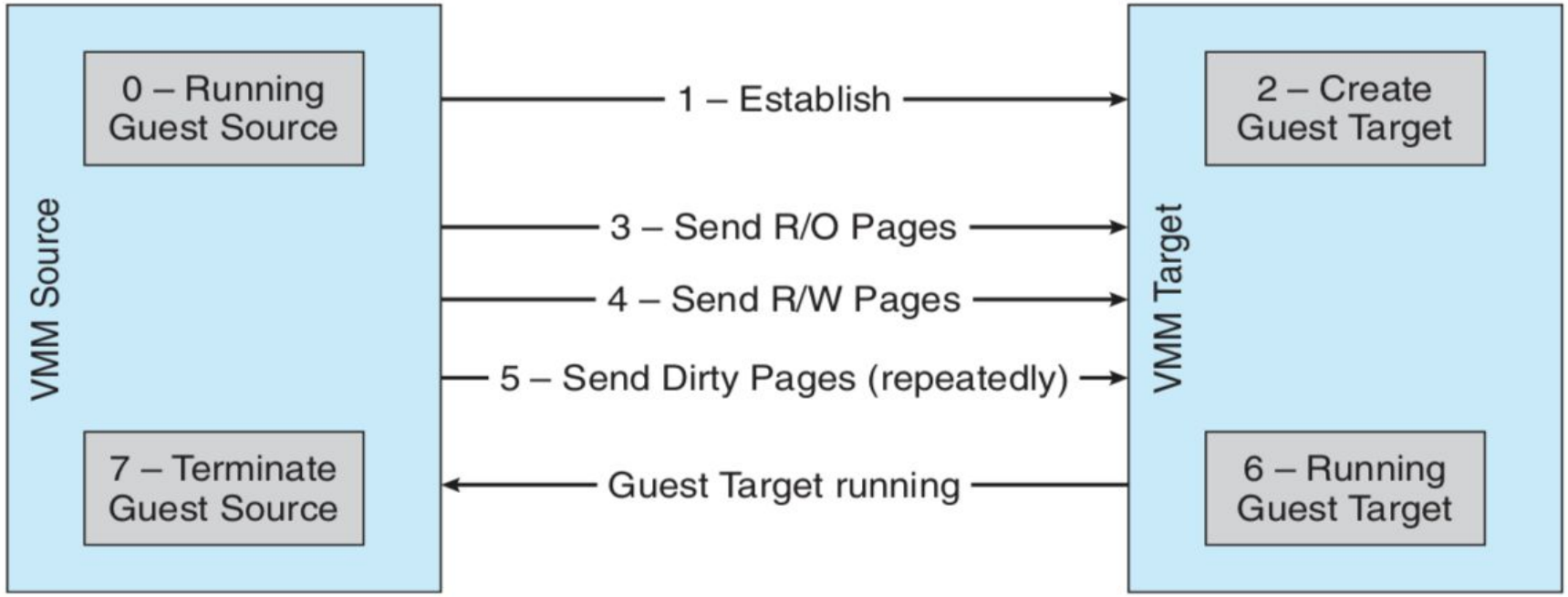


Virtual machine migration

- VM Migration refers to the process of **moving** a **running virtual** machine or application between different physical machines
- **Live migration** implies not disconnecting the client or application
- In simpler words, moving a virtual machine from one host to another

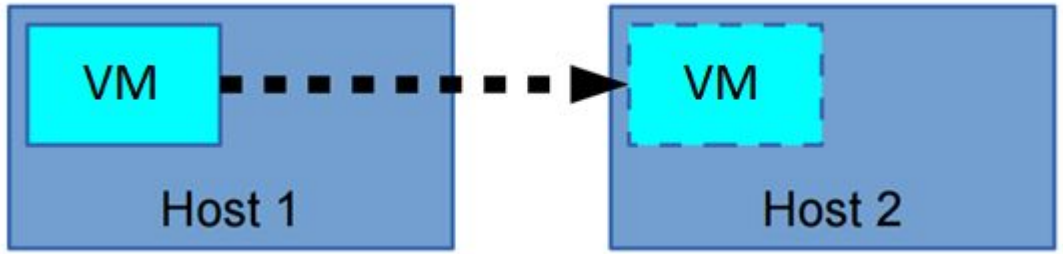


Live VM migration



- At some point, when **most pages** have been **sent**, the VMM source **freezes** the Guest source, sends its vCPU's **final state** and dirty pages, and tells VMM target to start running the Guest, which terminates at source
- Freeze time: on the order of 100 milliseconds

Migration: motivations and types

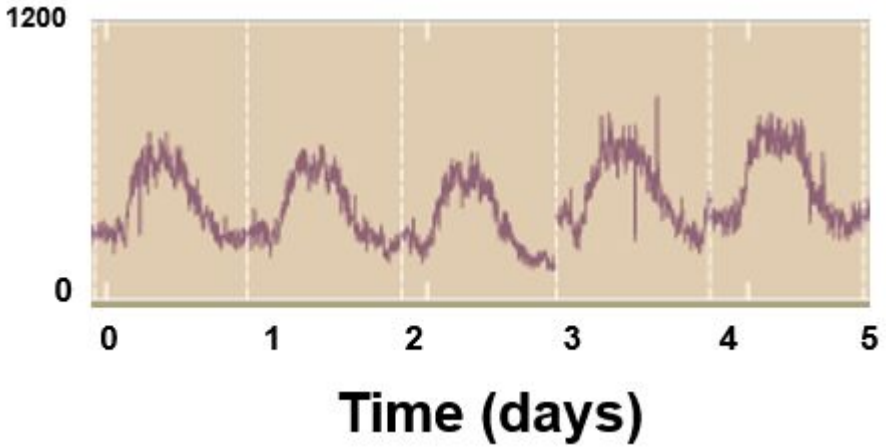


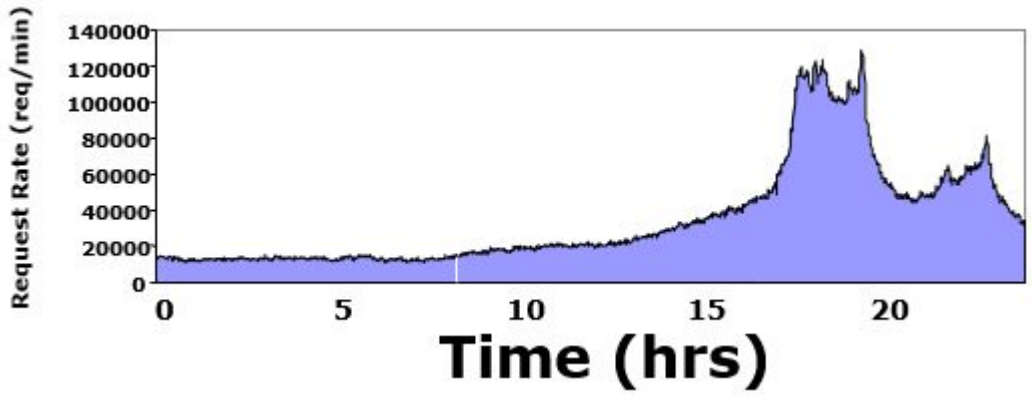
- Load balancing
  - Move VMs to a less busy host
  - VM consolidation
- Fast startups
  - Prewarmed VM snapshot
- Maintenance
  - Move VMs off a host before It is shut down
- Recovery From Host Failure
  - Restart VM on a different host

- COLD migration
  - Shut down VM on host 1, restart on host 2
- WARM migration
  - Suspend VM on host 1, copy across RAM and CPU registers, continue on host 2 (some seconds later)
- LIVE migration
  - Copy across RAM while VM continues to run. Mark "dirty" (changed) RAM pages & re-copy.

Dynamic workloads

- Applications can experience highly dynamic workloads
  - Timescale variations
  - Transient spikes and flash crowds





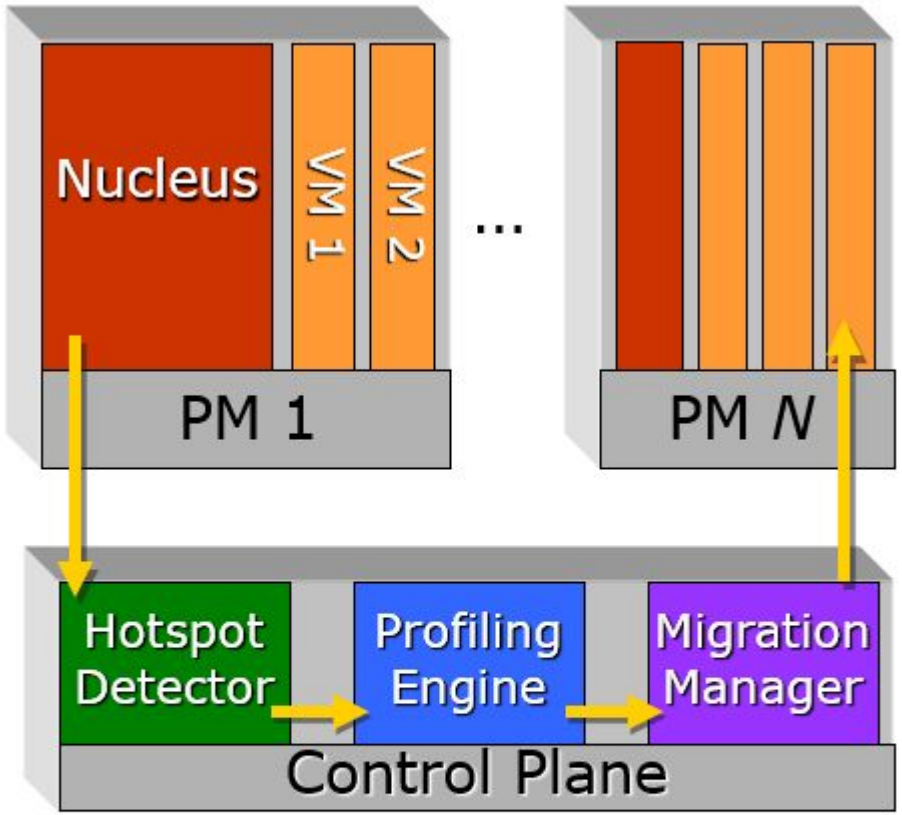
- Resources need to be **provisioned** to meet these **changing** demands
- Hotspots form if resource demand **exceeds** provisioned **capacity**

Solutions to the dynamic workload problem

- Static over-provisioning (allocate for peak load)
  - Wastes resources
  - Not suitable for dynamic workloads
  - Difficult to predict peak resource requirements
- Dynamic provisioning (allocate based on workload)
  - Requires automation and dynamic provisioning tools
  - Limited adaptability after provisioning
- VM migration and dynamic resource allocation
  - Dynamic load balancing (automatically detect and mitigate hotspots through VM migration)
  - When to migrate a VM?
  - Where to should a VM migrated?
  - How much of each resource should be allocated to each VM?

VM migration architecture

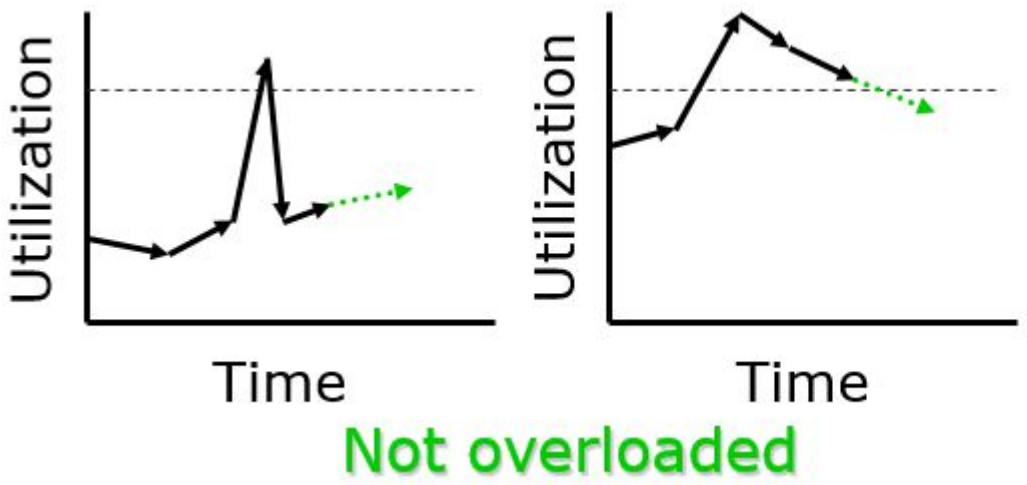
- Nucleus
  - Monitor resources
  - Report to control plane
  - One per server
- Control Plane
  - Centralized server
- Hotspot Detector
  - Detect when a hotspot occurs
- Profiling Engine
  - Decide how much of resources to allocate
- Migration Manager
  - Determine where to migrate



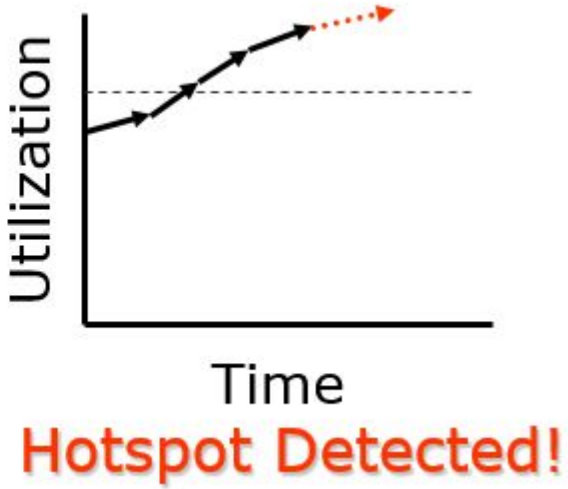
PM = Physical Machine
VM = Virtual Machine

Hotspot Detection – When?

- Resource thresholds
  - Potential hotspot if utilization exceeds threshold
- Only trigger for sustained overload
  - Must be overloaded for K out of N measurements

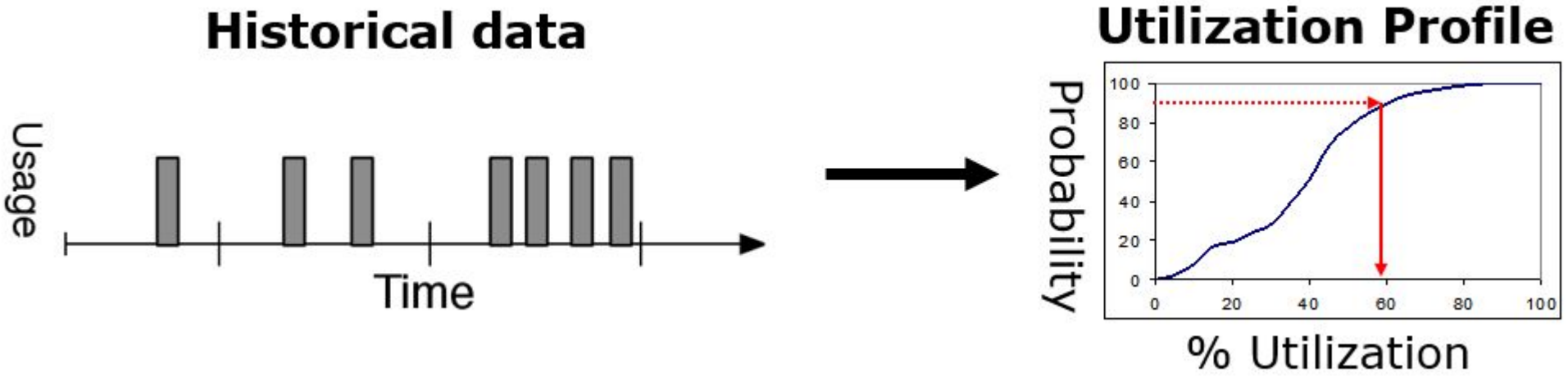


- Autoregressive time series model
  - Use historical data to predict future values
  - Minimize impact of transient spikes



Resource Profiling – How much?

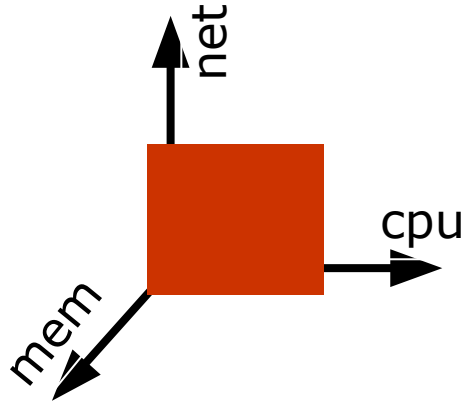
- How much of each resource should be given to a VM?
  - Create distribution from time series
  - Provision to meet peaks of recent workload



Determining placement – Where to?

- Migrate VMs from overloaded to underloaded servers

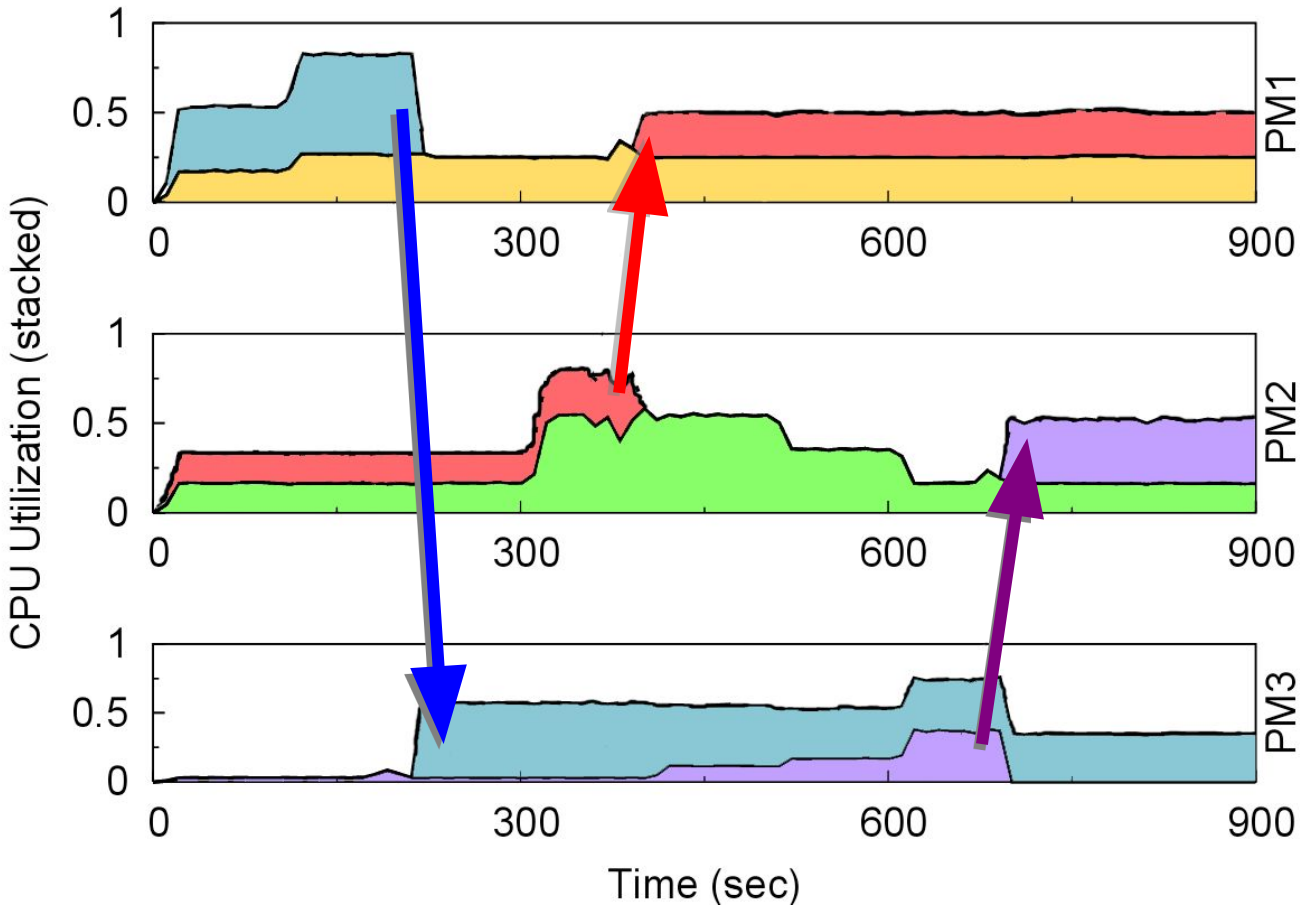
$$\text{Volume} = \frac{1}{1\text{-cpu}} * \frac{1}{1\text{-net}} * \frac{1}{1\text{-mem}}$$



- Use Volume to find most loaded servers
  - Captures load on multiple **resource dimensions**
  - Highly **loaded** servers are targeted **first**
- Migrations incur overhead
  - Migration **cost** determined by **VM size**
  - Migrate the VM with **highest** Volume/VM size **ratio**
  - **Maximize** the amount of load **transferred** while **minimizing** the **overhead** of migrations

Migration effectiveness example

- 3 Physical servers, 5 virtual mach
- Migration triggered when CPU usa
- 3 hotspots were detected and miti
- The system is continuously monit



Agenda

- Virtual machines
- Hypervisors
  - Full-virtualization
  - Paravirtualization
  - Hardware-assisted virtualization
- Virtualization of virtual memory
- I/O virtualization
- Virtual machine migration
- Automation of VM creation
- Containers
- MicroVMs
- Unikernels

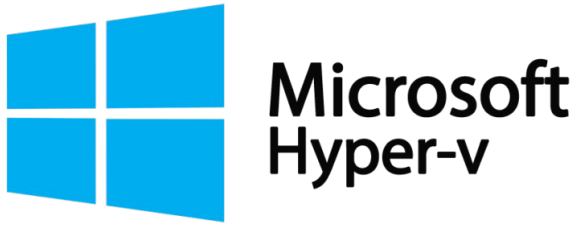
How to create Virtual Machines in VirtualBox?

- Download and install VirtualBox
- Host OSs supported:
  - Windows
  - Mac OS X
  - Linux
  - Solaris (Oracle)
- Requirement (again): virtualization activated in boot settings (UEFI/BIOS)

- **How to create a VM in VirtualBox** (Windows hosted)

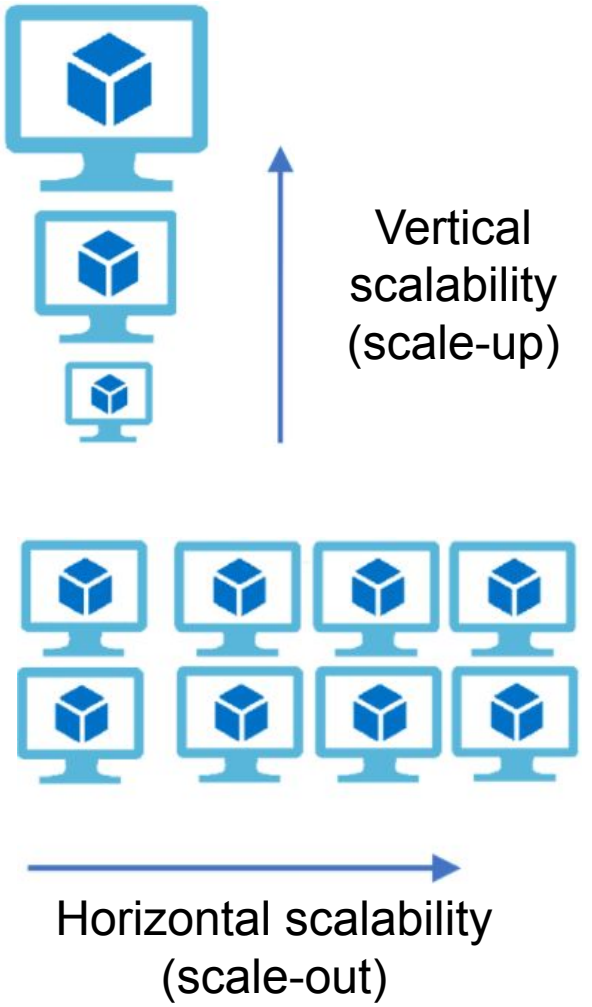
Should VM creation be manual?

- VMs are very flexible and useful, but their **manual** creation is **cumbersome** and **error prone**!
- Each new VM created requires all these steps (creation, configuration, OS and apps installation, etc.)
- Adequate to play with one or two VMs, but not for professional IT infrastructures!
- These can have many VMs and change frequently
- Apply the DRY (**Don't Repeat Yourself**) principle, as opposed to the WET (**Write Every Time**) principle:
  - Define reusable templates and patterns, rather than repeating manual procedures
  - Automate operations based on these templates and patterns



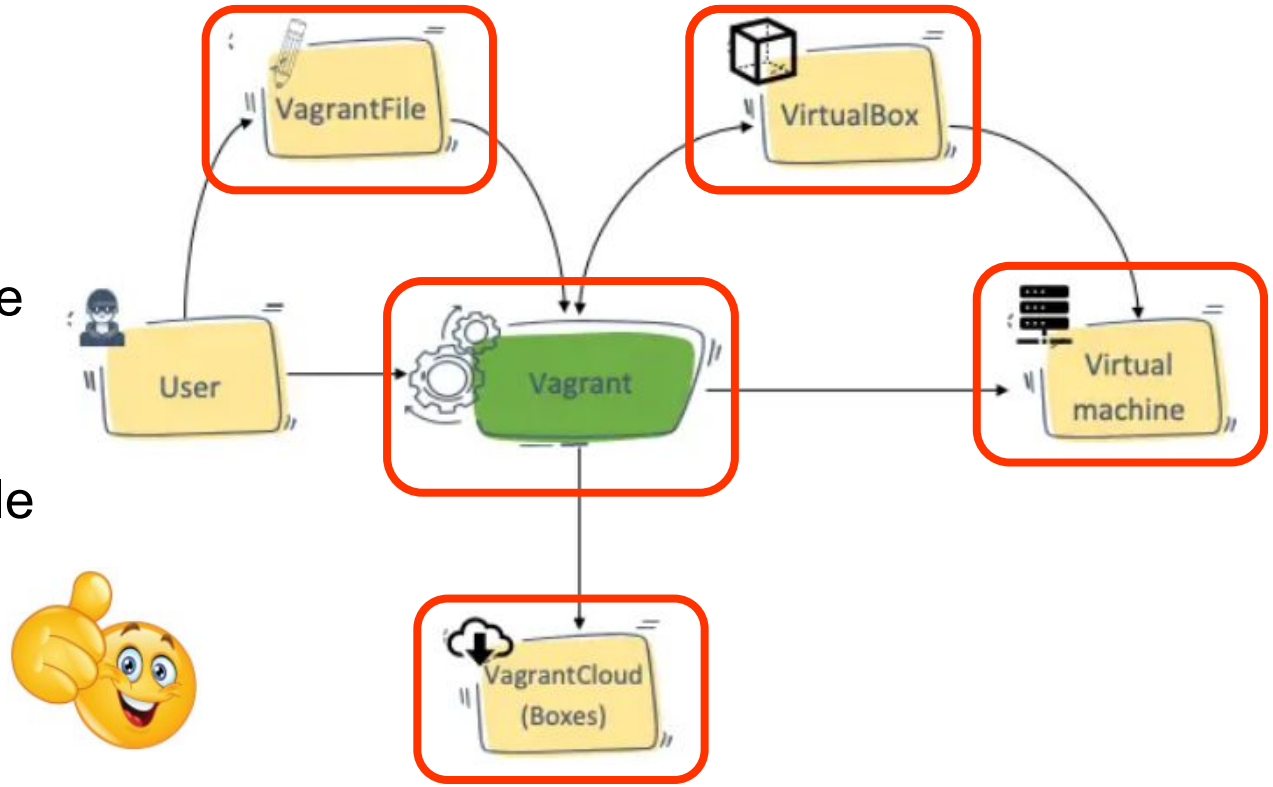
Automation is needed

- The only constant thing in IT is **change**!
- IT infrastructures are always changing its VMs:
  - Scale vertically: better characteristics (memory, power, …)
  - Scale horizontally: replicate VMs for more processing power
  - New VMs, with different characteristics and/or applications
  - Replicate and modify development environments
  - Sandboxed environments for testing applications
- Virtualization enables easy, dynamic, and repetitive creation of VMs
- Manual creation and configuration of VMs is only practical in trivial cases
- Creation/configuration of VMs needs to be automated



Vagrant: a tool for automating virtualization

- Vagrant: Open-source software **tool** for **building and maintaining** portable **virtual** software development **environments**
- The **VagrantFile** contains the definition and configuration of VMs
- VirtualBox **provides the hypervisor**
- Vagrant also supports Hyper-V, VMware, …
- **Vagrant Cloud** is a repository of already made VM images (“boxes”)
- A simple command (“vagrant up”) creates and configures the VMs according to the Vagrantfile
- Now VMs can be created in an automated, reproducible, and deterministic way



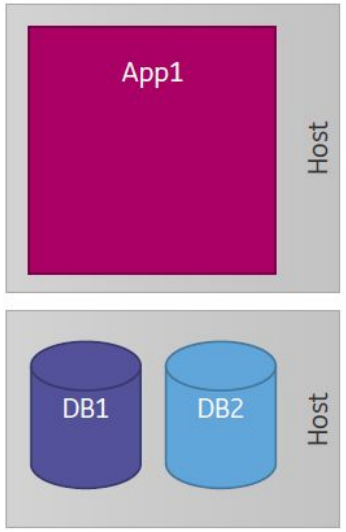
Agenda

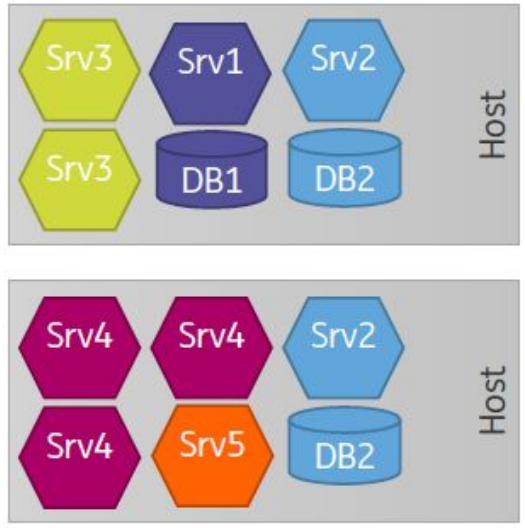
- Virtual machines
- Hypervisors
  - Full-virtualization
  - Paravirtualization
  - Hardware-assisted virtualization
- Virtualization of virtual memory
- I/O virtualization
- Virtual machine migration
- Automation of VM creation
- Containers
- MicroVMs and Unikernels

Virtualization Granularity

- **VMs** are **heavy-duty** components (full-fledged OS, large footprint, long boot times), better suited for large-grained (monolithic) applications
- However, the dynamics of modern applications (**constantly changing and changing at different paces**) led towards an interacting network of a significant number of small applications (microservices)

Monolithic environment

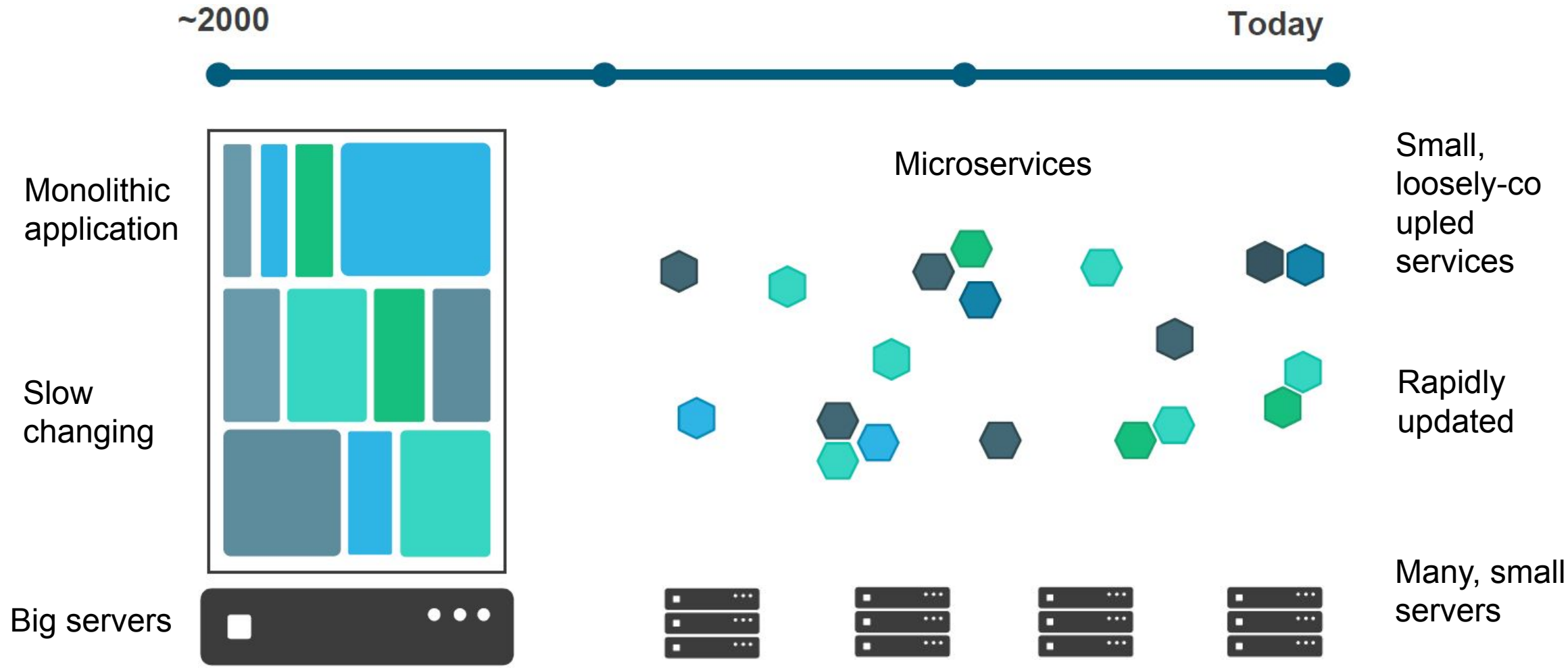




Microservices environment

- Changes are easier: a small change affects only a few microservices

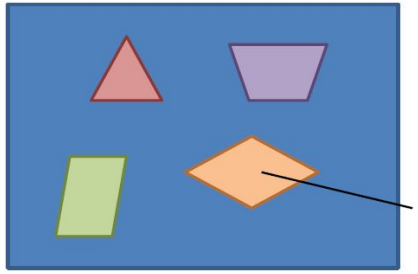
From monolithic apps to microservices



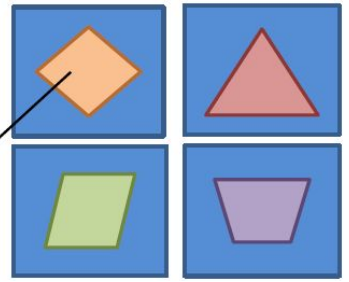
Scaling out applications

- If performance requires scaling out (more “workers”):
  - monolithic applications must be entirely replicated
  - with microservices, scaling can be done just for the components that require it

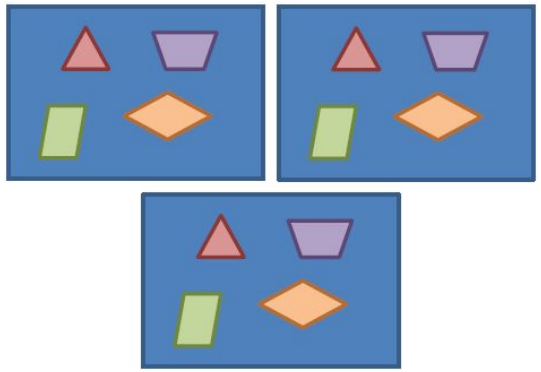
Monolithic environment

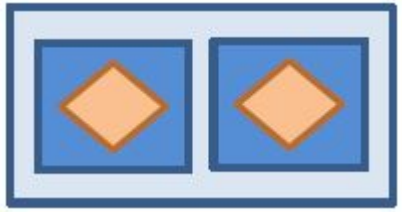


Bottleneck component

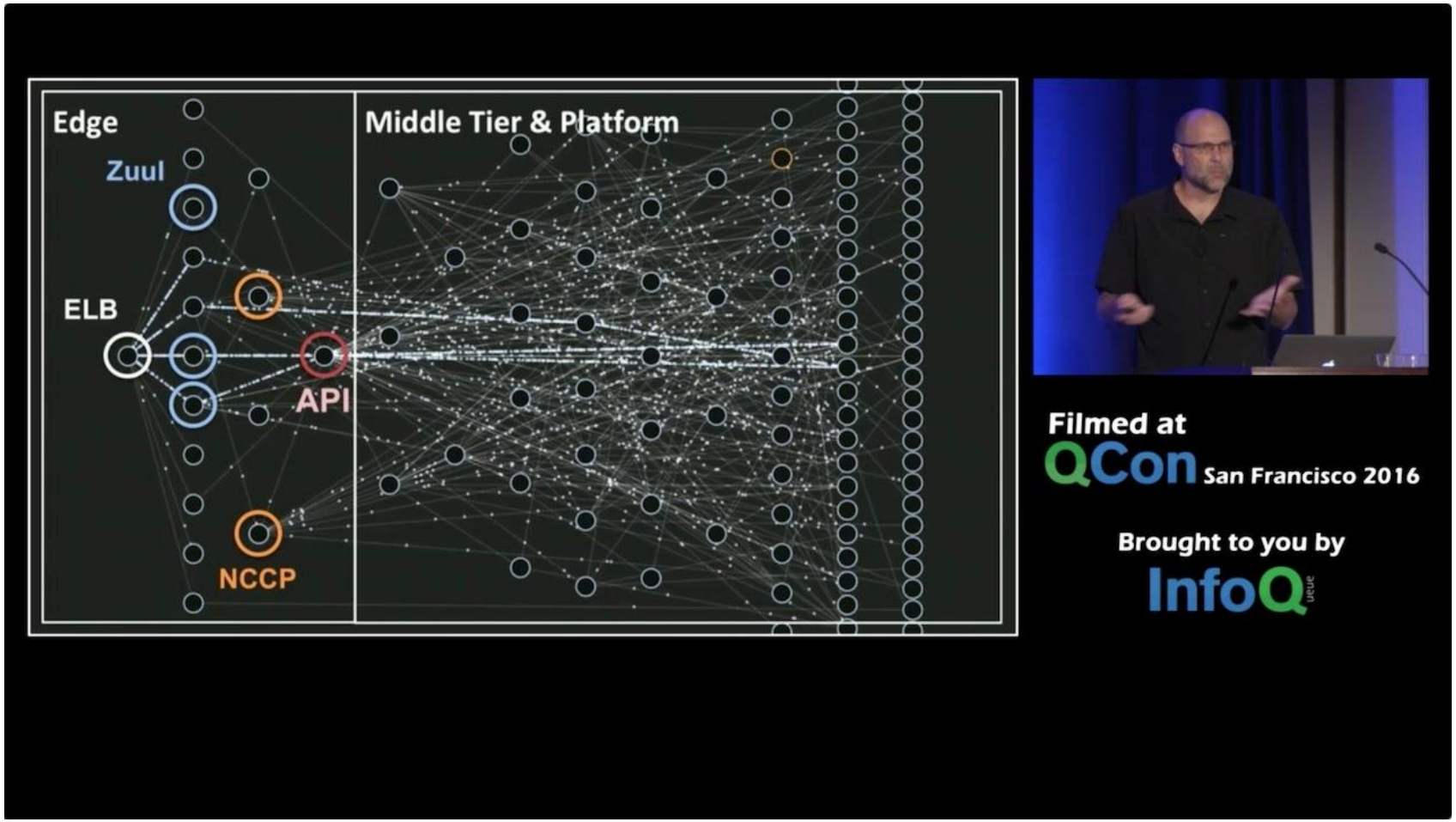


Microservices environment





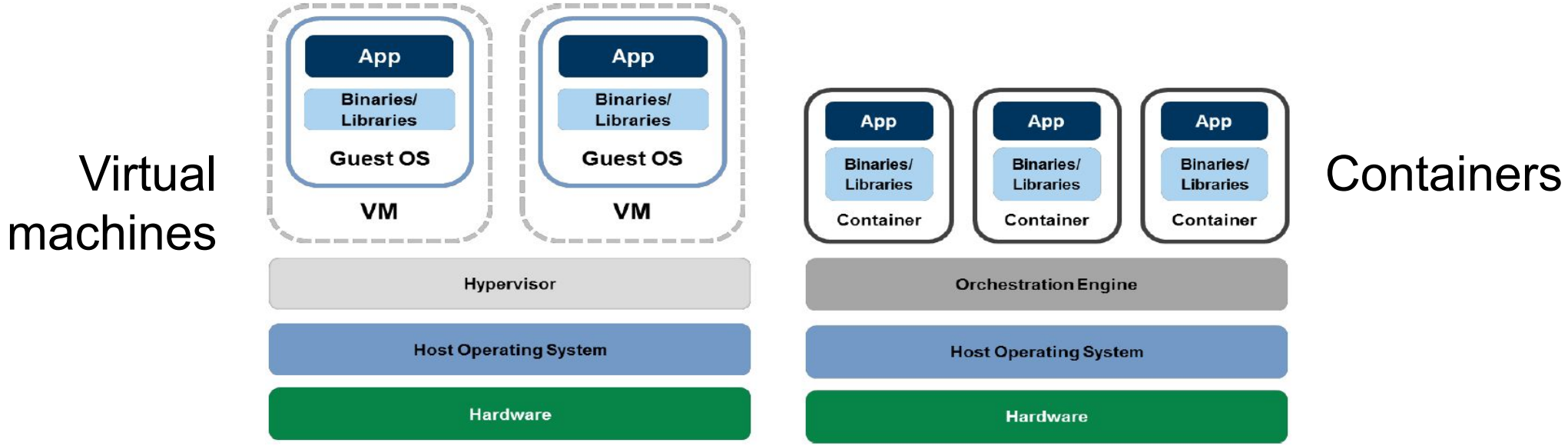
Microservices at Netflix



Mastering Chaos - A Netflix Guide to Microservices

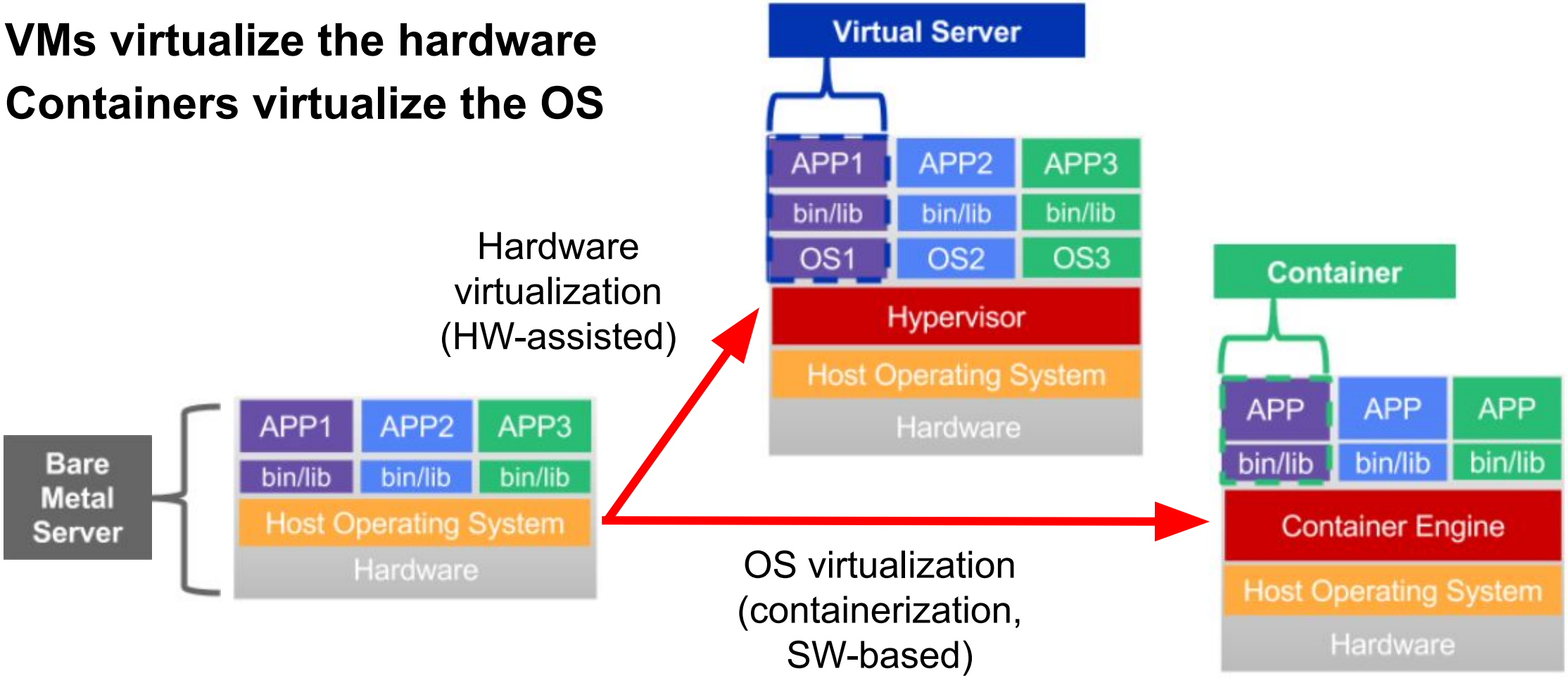
Containers as a virtualization mechanism

- To support this trend, a sort of “lightweight VM” (container) is needed, with a smaller footprint, lower latency, and faster boot times:
  - Use **OS mechanisms for isolation** (e.g., cgroups, namespaces), without HW support
  - No hypervisor managing guest OSs → **Orchestration engine** sharing a single OS
  - Include all that is required by the app in the container (e.g., binaries, libs, and configuration files, but not the OS) and nothing more (bare minimum, single process)



Hardware and OS virtualization

- **VMs virtualize the hardware**
- **Containers virtualize the OS**



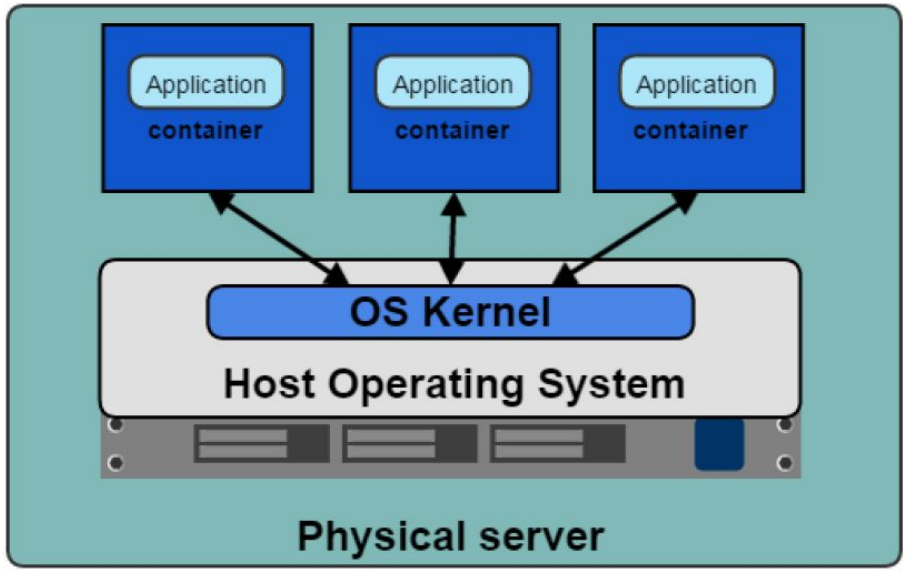
Virtual machines vs containers

Virtual machine (stand-alone and unshared OS)

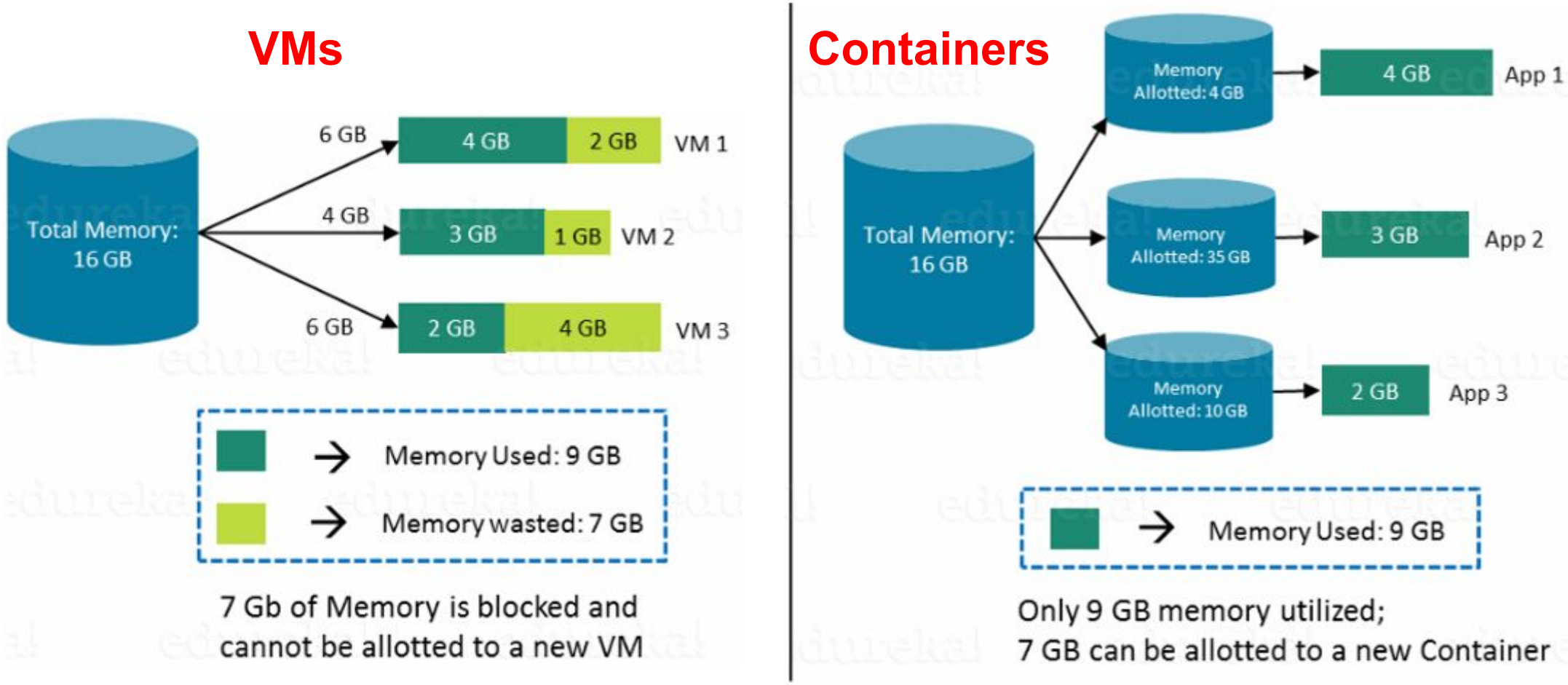
Containers (share the building infrastructure – the OS)

Containerization

- Containerization uses the kernel on the host operating system to run **multiple root file systems** (evolution of *chroot*)
- Each container has its own root directory and can only access the file tree underneath
- A container has its **own space and resources can be limited** (within host capabilities):
  - CPUs, GPUs
  - Memory
  - I/O (storage, networking)



Memory usage: VMs vs containers



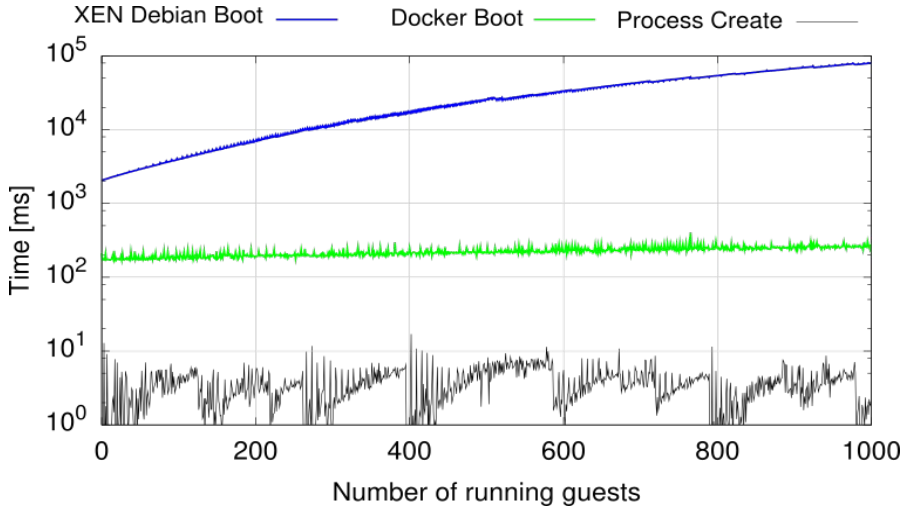
- Containers share the OS and can be more **precise in memory requirements**

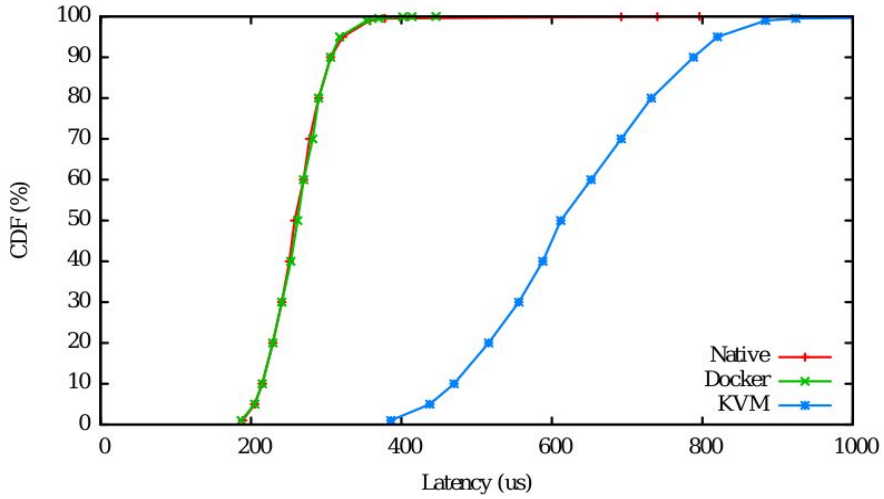
Boot time and latency

- VM boot time:
  - Much higher than that of containers
  - Increases with the number of VMs in server, unlike containers (grows just a bit)

- Latency (time to start processing a request):
  - Container is almost identical to native
  - VMs much slower (storage virtualization is heavy)

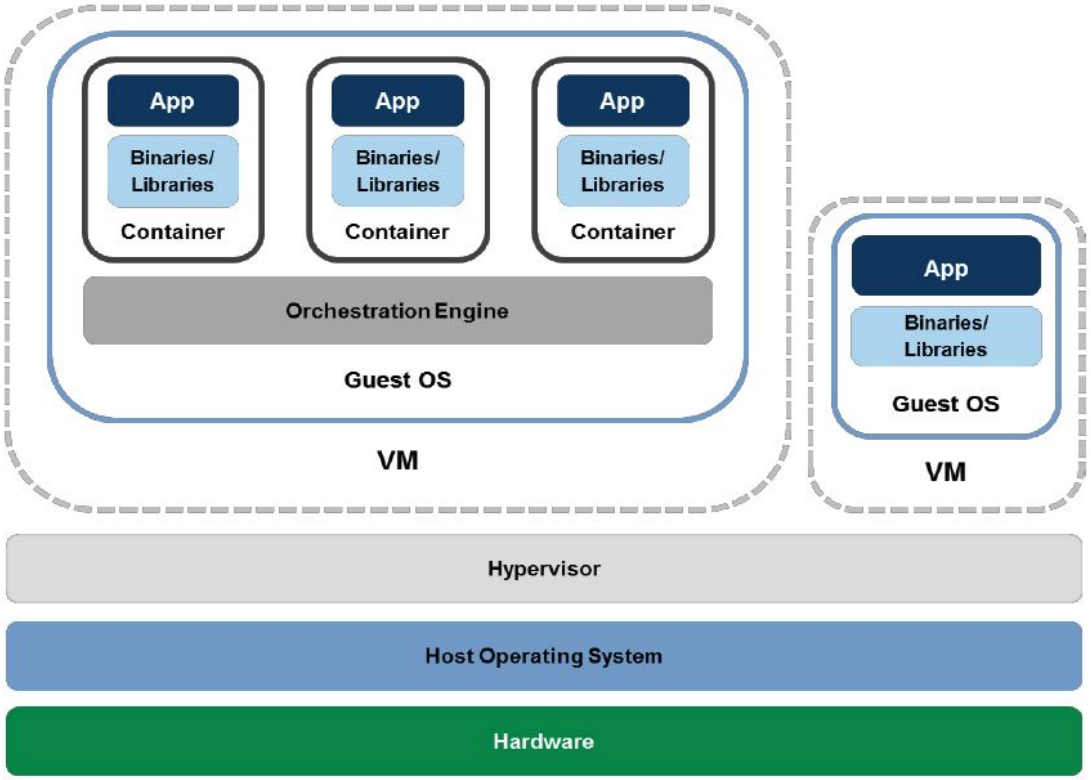
- The CDF (Cumulative Distribution Function) indicates the probability that the latency will take a value less than or equal to the indicated value





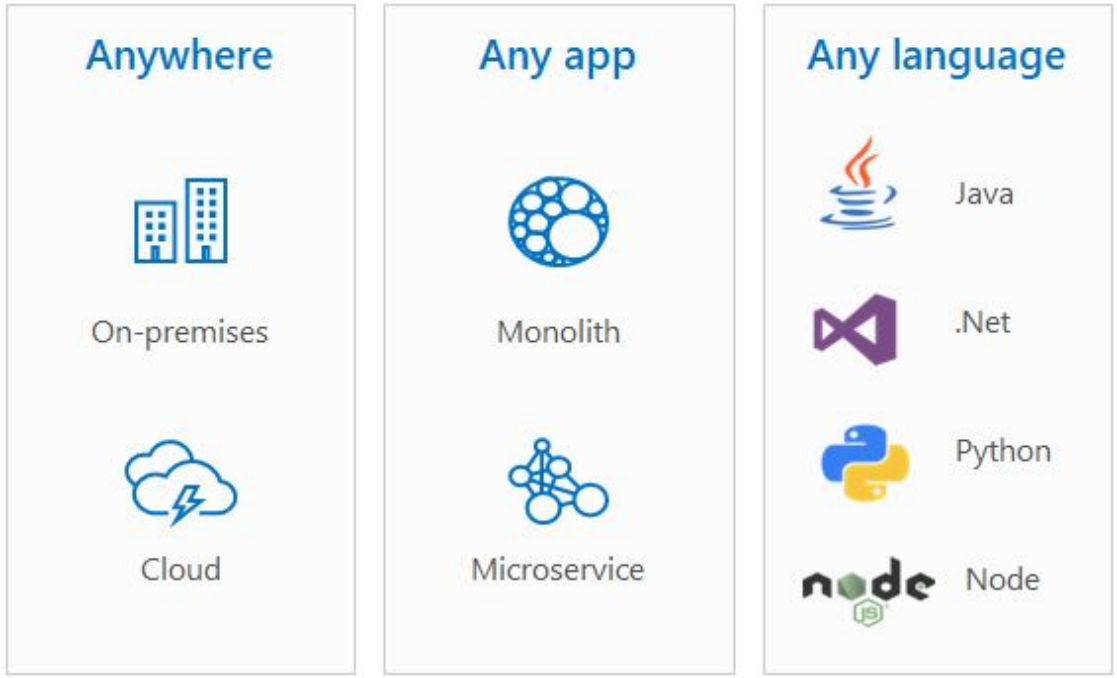
Containers and VMs

- **Sharing** the OS **reduces** the footprint (many more containers per host)
- Container snapshots are much smaller than those of VMs (OS not included)
- Containers boot much faster than VMs because the OS is already running
- Containers are not as isolated as VMs, but they can run inside VMs
- **Containers cannot** be **migrated** between different OSs
- Since they are self-contained, all containers can be used in the same way by the same API, without caring about what they contain

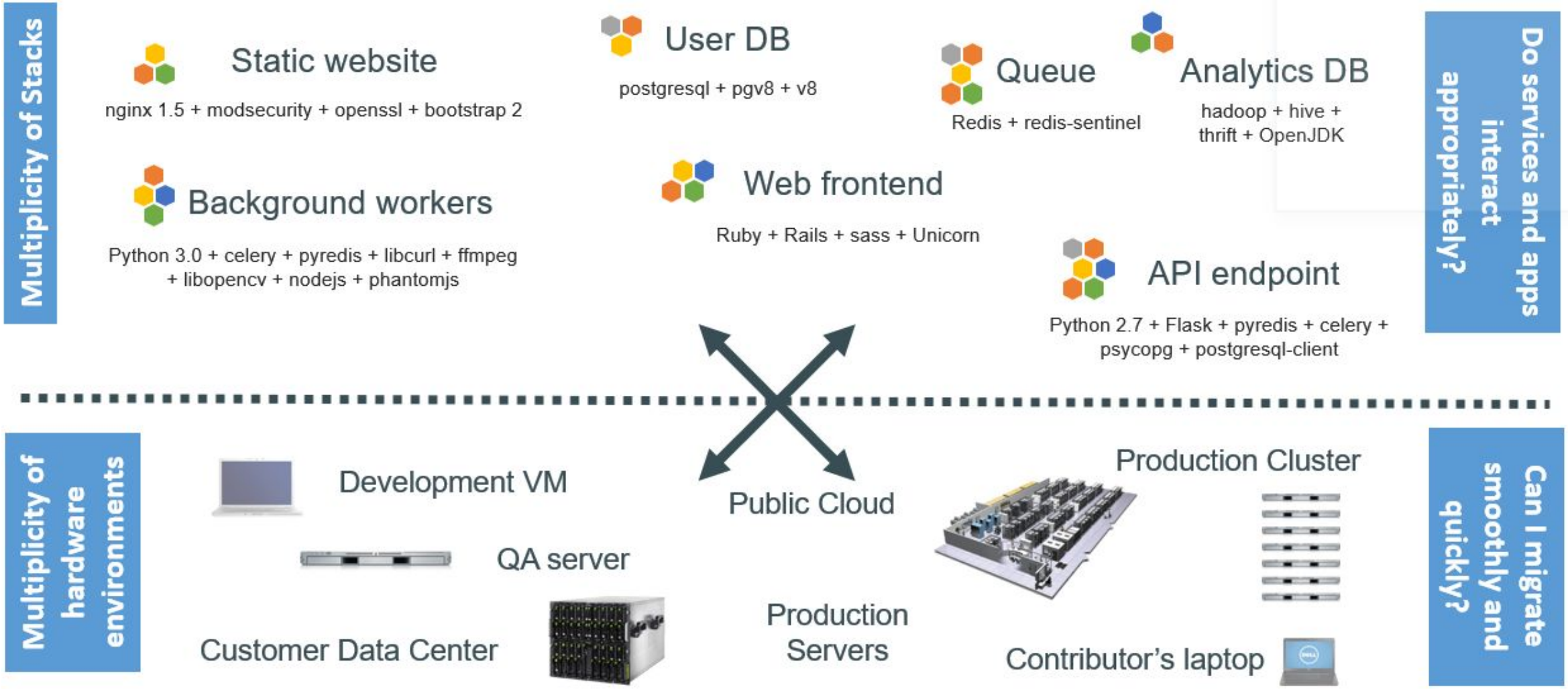


Containers - an application packaging system

- **Solve** not just the **virtualization problem** (duplication of reality and isolation) but also the **deployment problem** (packaging, although in the same OS)
- Container: a standard way to package an application and all its dependencies so that it can be moved between environments and run without changes.
- Containers include all dependencies required by their application: everything outside the container can be standardized.
- Similar to a graph with nodes: these can be different internally, but graph traversal and processing is universal.
- Build, ship, and run any anywhere



The interaction/migration problem

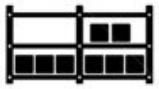


How to seamlessly deploy apps on platforms?

	Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
Static website	?	?	?	?	?	?	?
Web frontend	?	?	?	?	?	?	?
Background workers	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?

Cargo Transport Pre-1960

Same old M x N problem

	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
	?	?	?	?	?	?	?
							

Good old days: dockers in ship docks

- Docker: worker to load/unload ships at the docks
- Manual labor, item by item

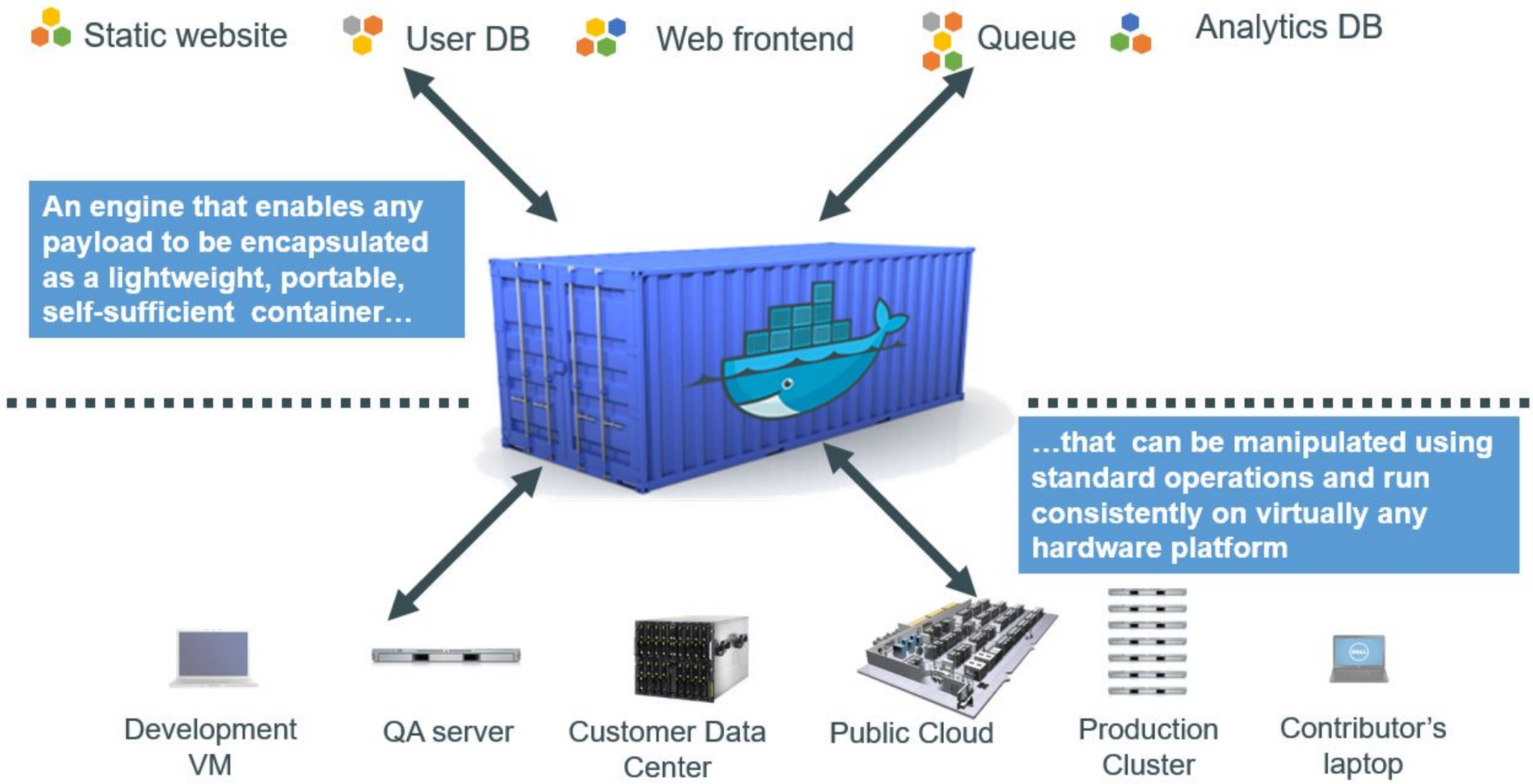
Solution: intermodal shipping container

- 1956: first intermodal shipping container (Malcom McLean)

Container ships, trains and trucks

Container-handling hubs

Docker is a shipping container system for code



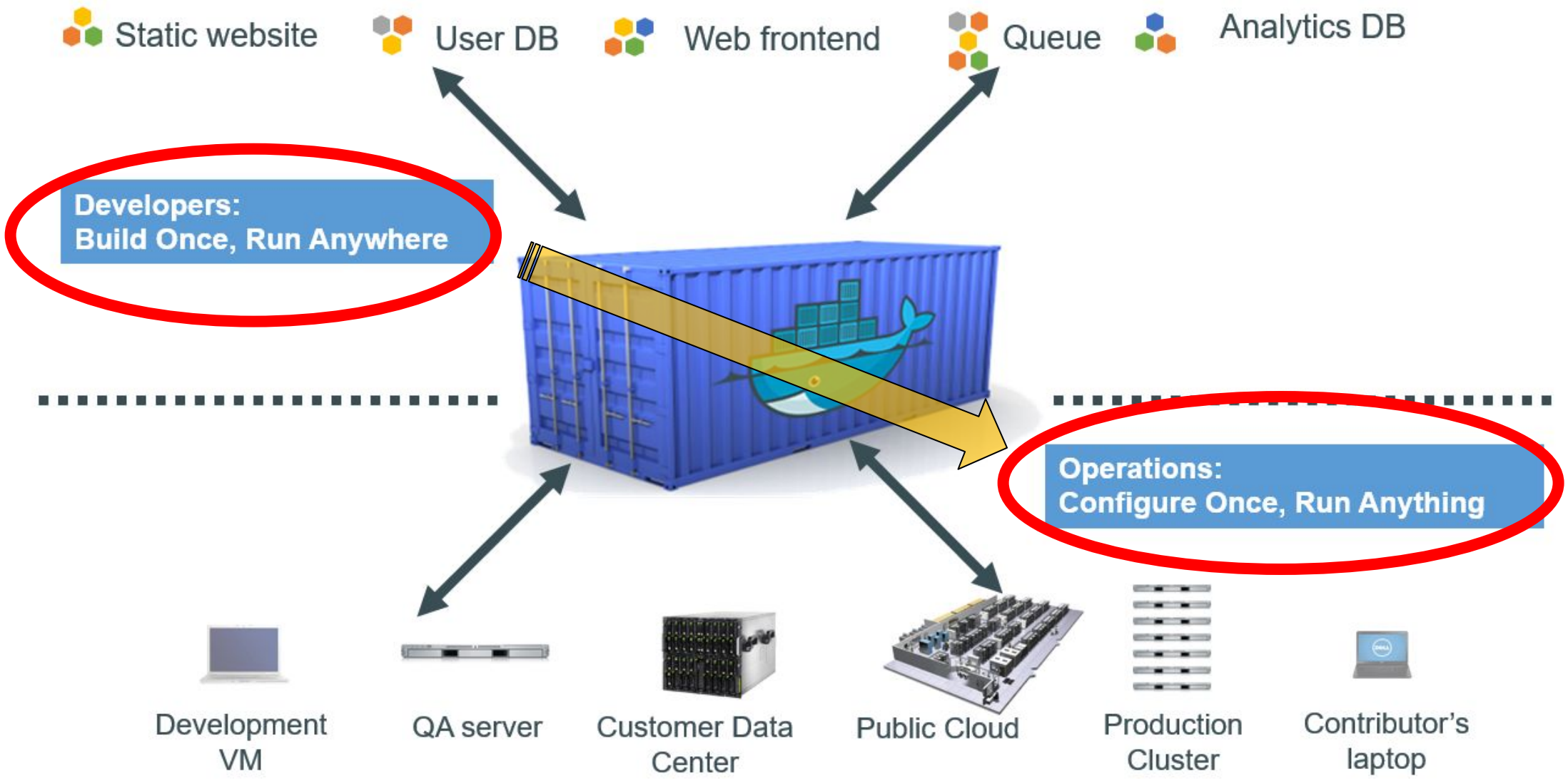
Separation of concerns

- The Developers team worries about what is **inside** the container
  - Code
  - Libraries
  - Package Manager
  - Other applications
  - Data
- Servers with same type of OS are compatible

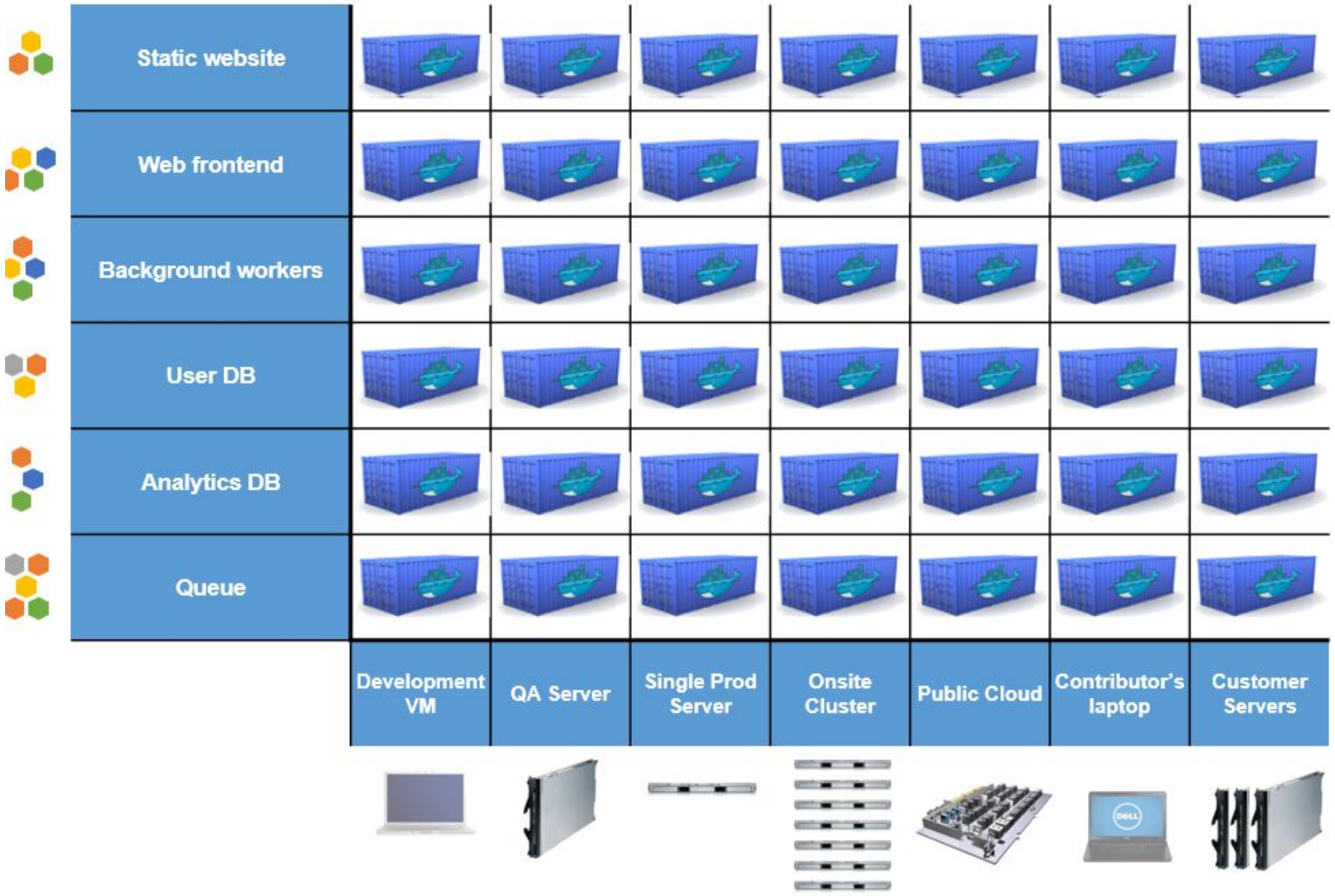
- The Operations team worries about what is **outside** the container
  - Logging
  - Remote access
  - Monitoring
  - Network configuration
- All containers start, stop, copy, attach, etc., the same way

- External dependencies are limited to the OS kernel (the rest is included)
- Docker enables SW container automation (replaces “human dockers”)

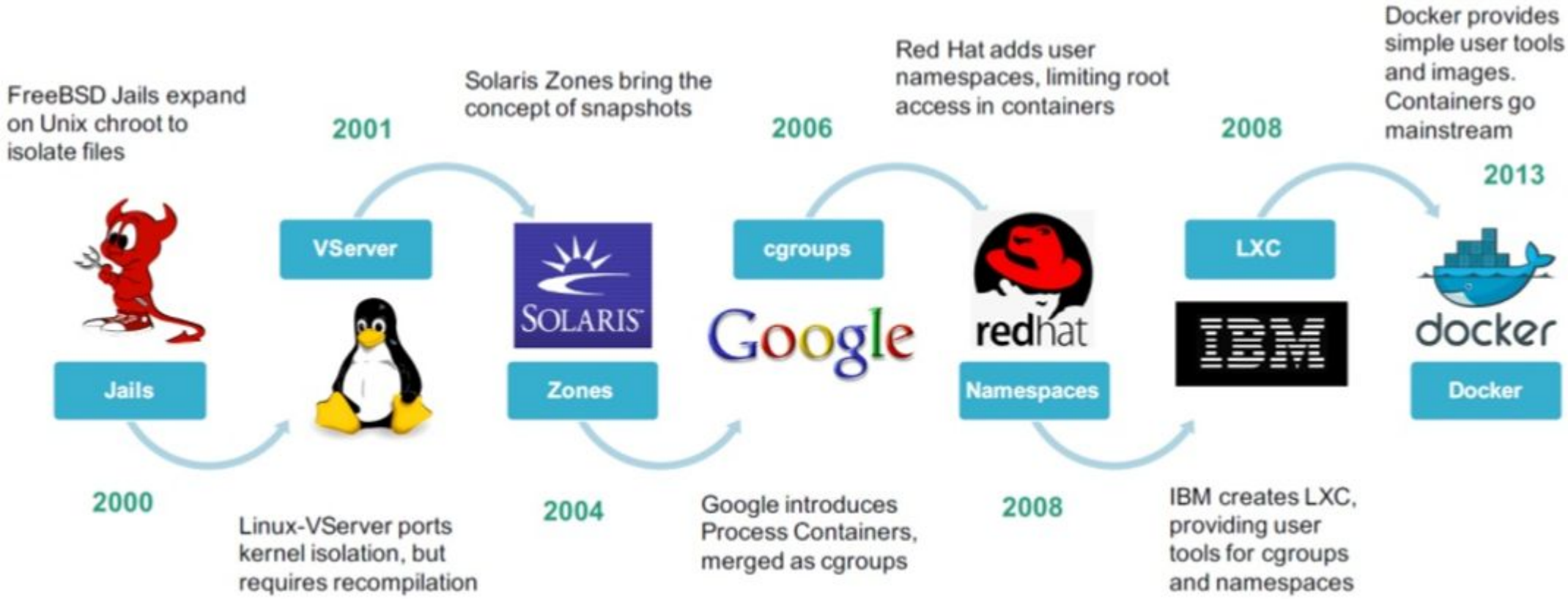
Docker bridges Developers and Operations



Solved: seamlessly deploy apps on platforms

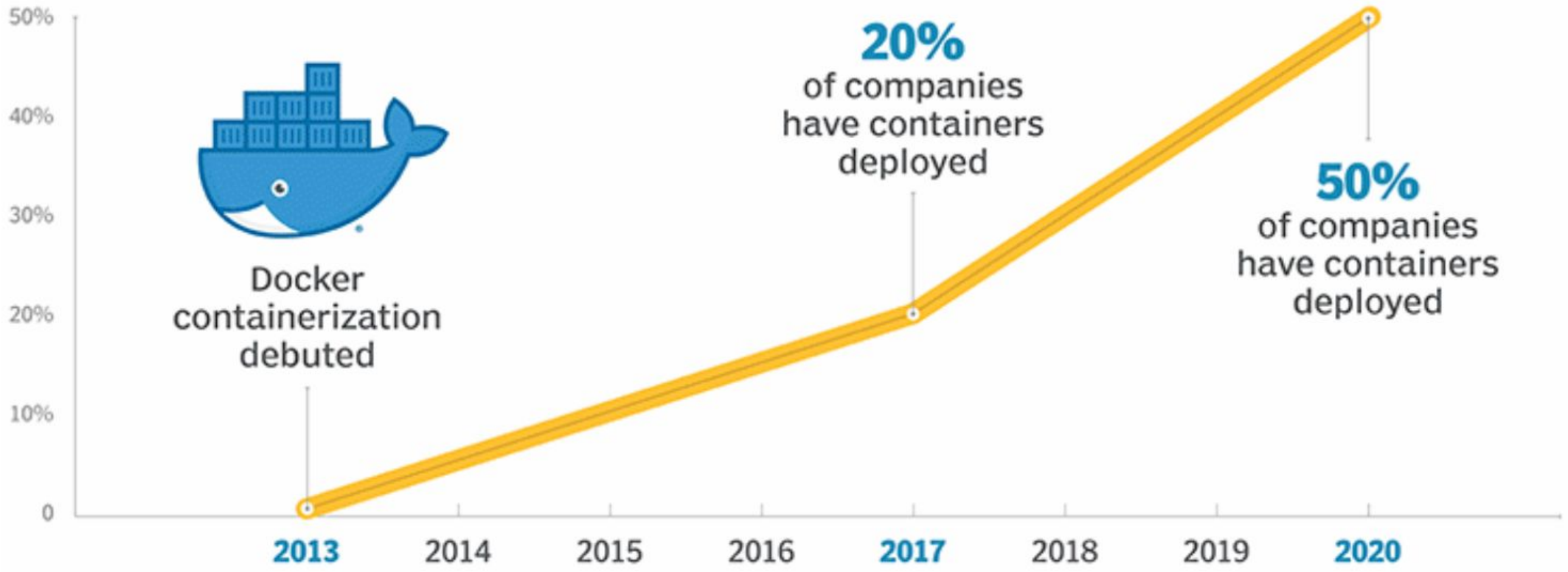


History in software containers



- Today, everybody supports containers (Linux, Windows, Mac OS, AWS, Azure, Google Cloud, etc.)

Containerization timeline (Gartner)



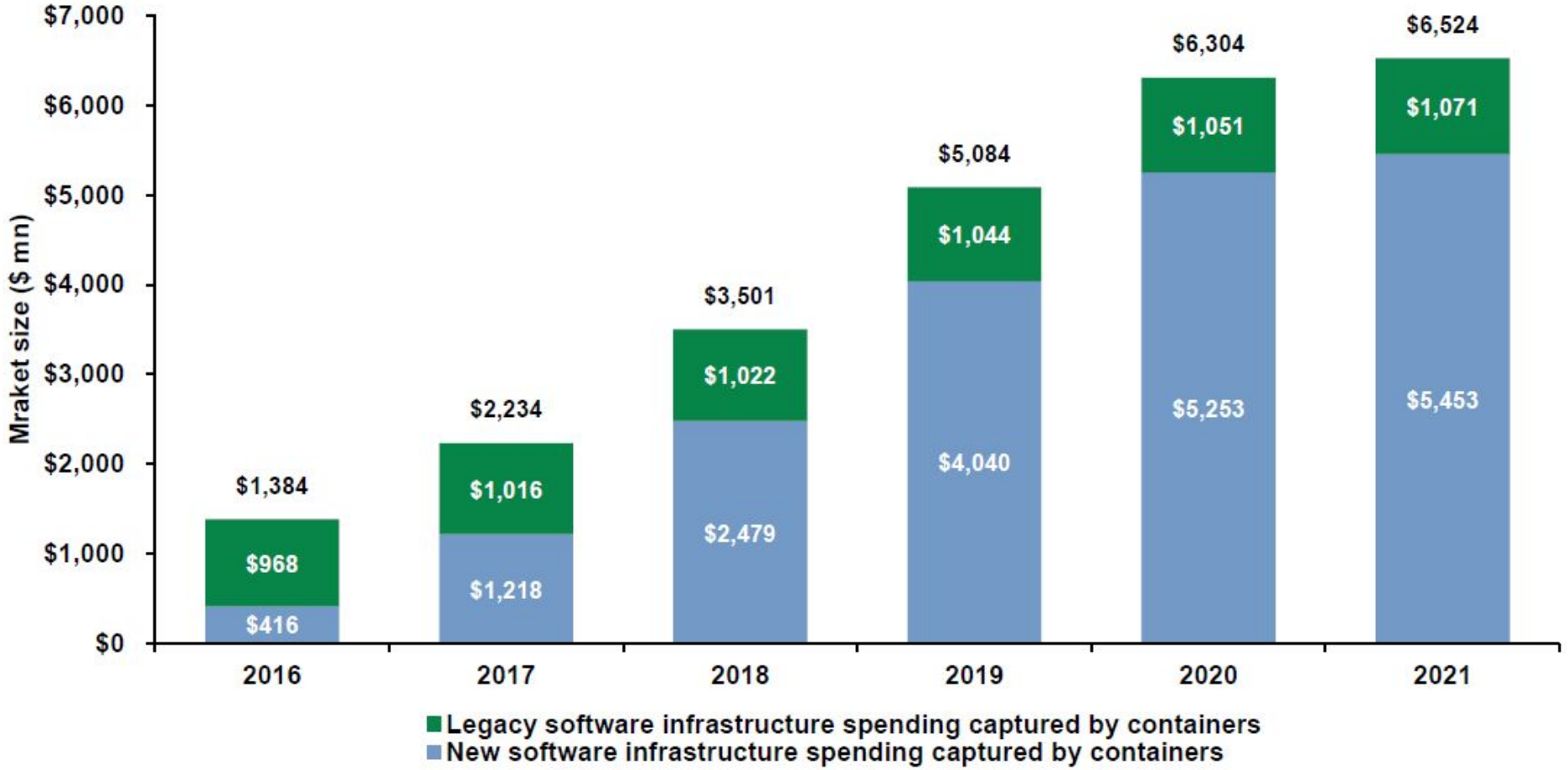
- Containers are becoming pervasive, at an increasing rate

Open Container Initiative

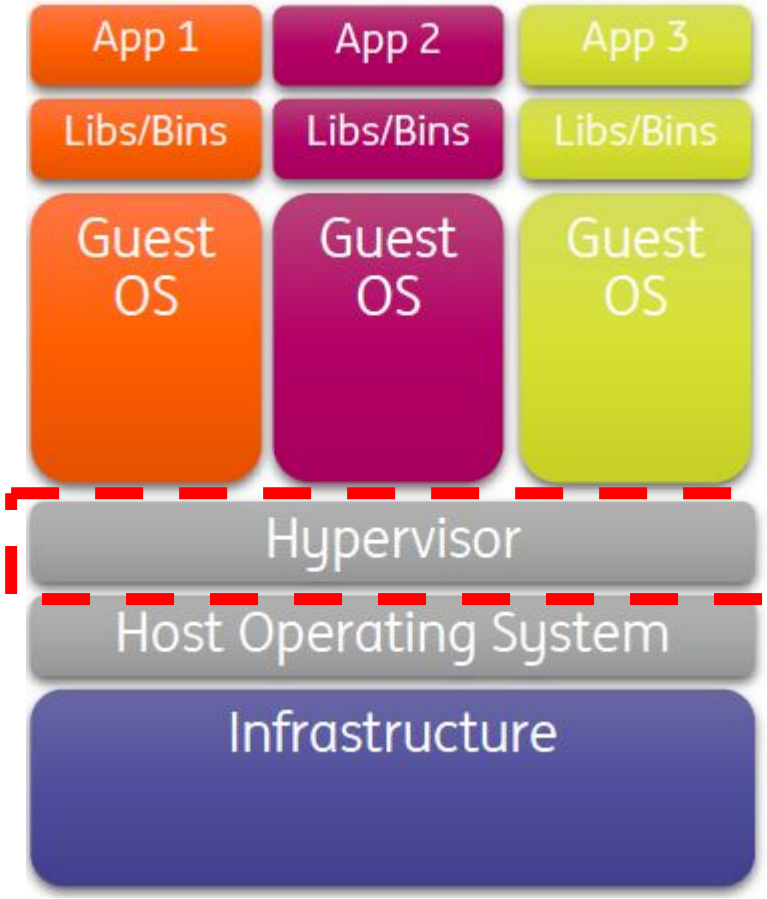
- Started in 2015 to promote open container **standards** (formats and runtime)
- Two specifications (version 1.0 in May 2021):
  - **Runtime Specification** (runtime-spec): outlines how to run a “file system bundle” that is unpacked from an image
  - **Image Specification** (image-spec): the container image format specification

The market relevance of containers

- Estimated total container market in 2021: ~$7bn

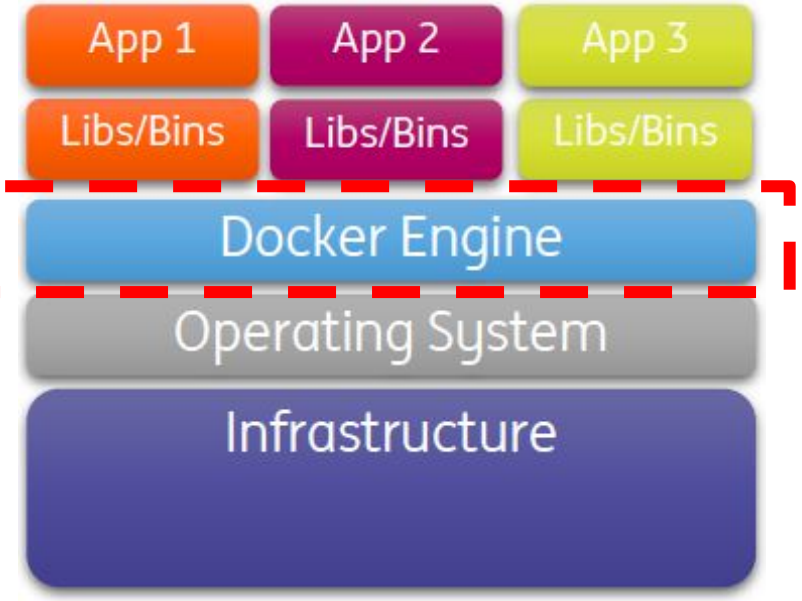


From VMs to Containers



VM environment

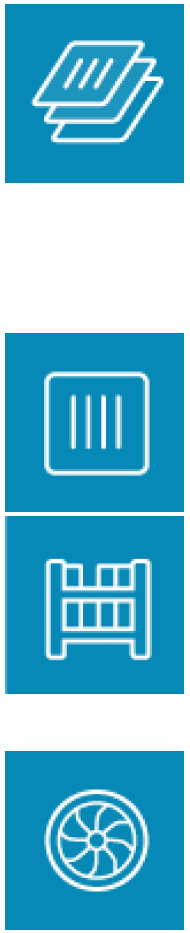
Docker is an open-source platform to build, ship, and run distributed applications, using the container mechanisms



Docker environment

There is no "bare-metal" Docker Engine!

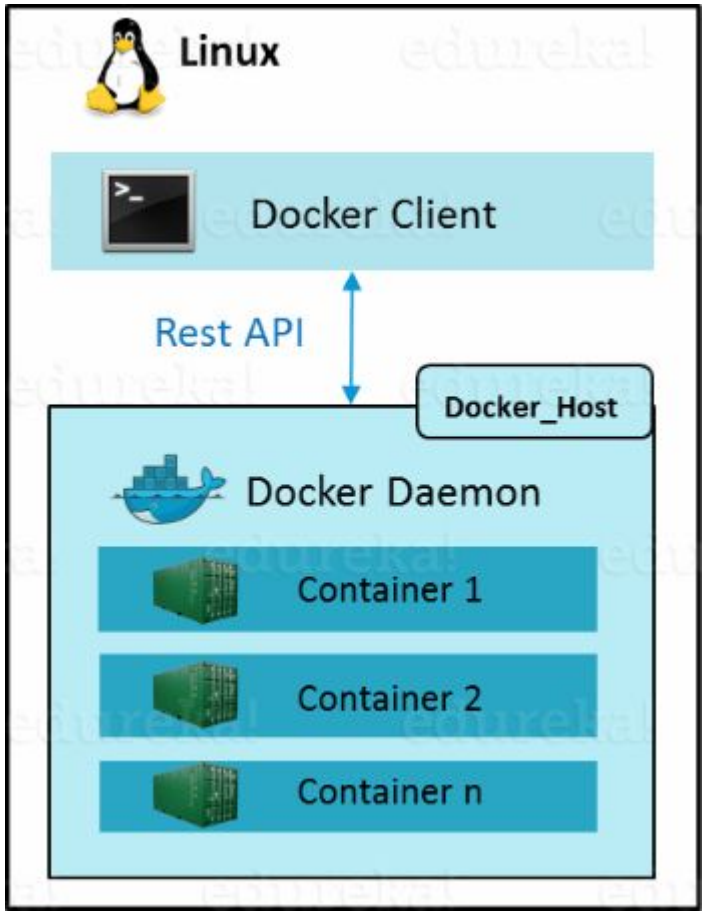
Docker – a shipping container for code

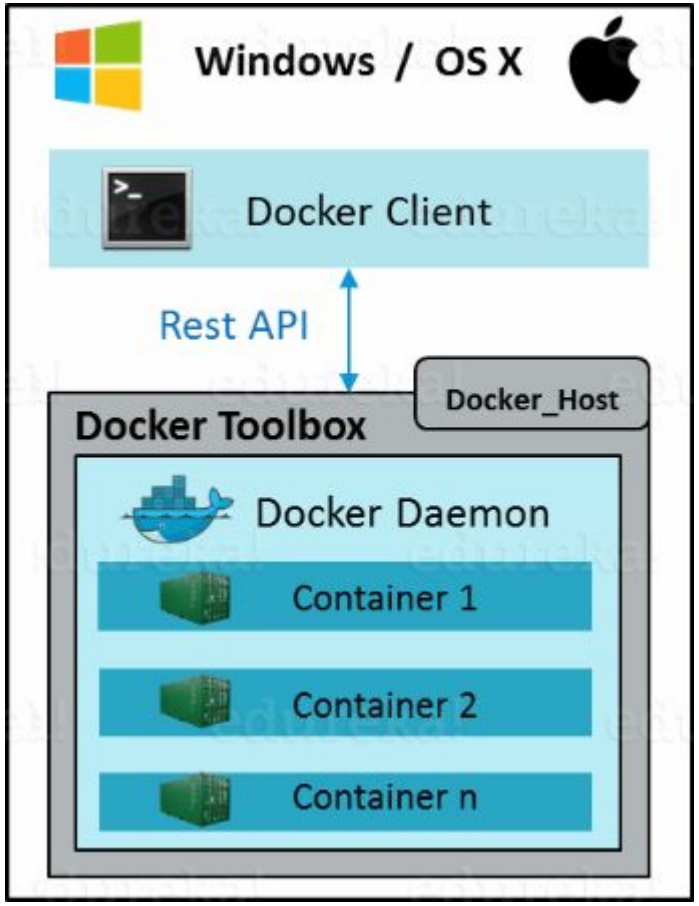


- Image
  - Immutable (unchangeable) file that contains the code, libraries, dependencies, and other files needed for an application to run (in layers)
  - Created from a Dockerfile or as a snapshot of a running container
- Container
  - An instance of an image with a writable layer, so that it can run
- Docker Hub (Docker's official container registry) / any Docker registry
  - Stores, distributes and shares container images
  - Available in cloud or enterprise repositories with images ready to use
- Docker Engine
  - A server with a long-running daemon process (dockerd)
  - Creates, ships and runs application containers
  - Runs on any physical server or virtual machine, locally or in public clouds

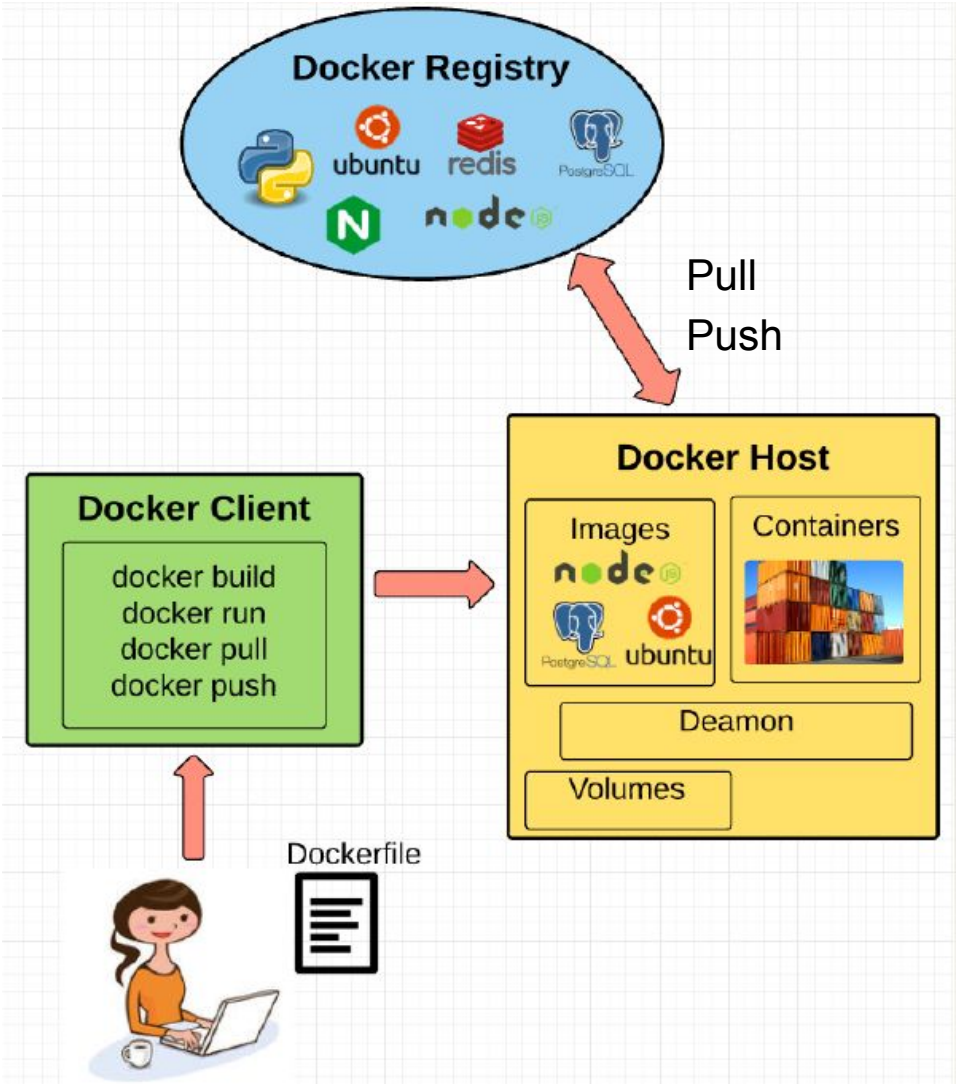
Docker Engine

- A client-server application that uses:
  - A server (a long-running process, the Docker Daemon)
  - A command line interface (CLI) client, or
  - A REST API for communication between the CLI client and Docker Daemon

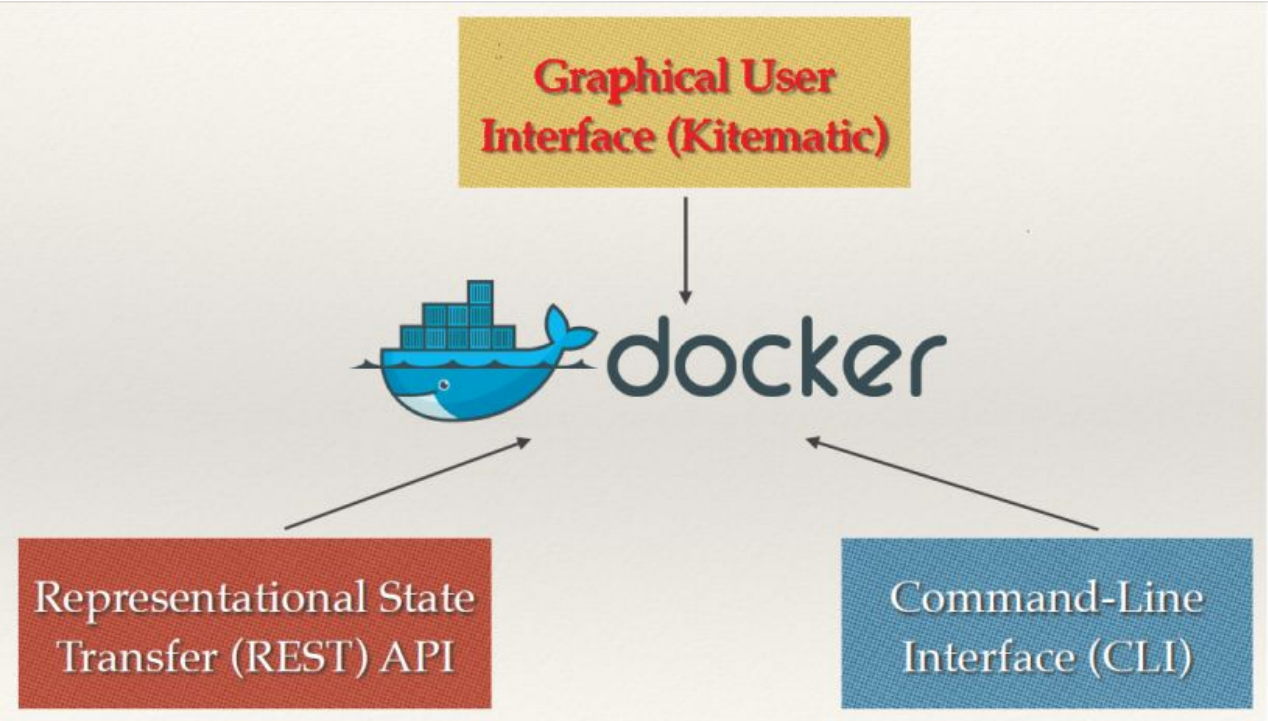




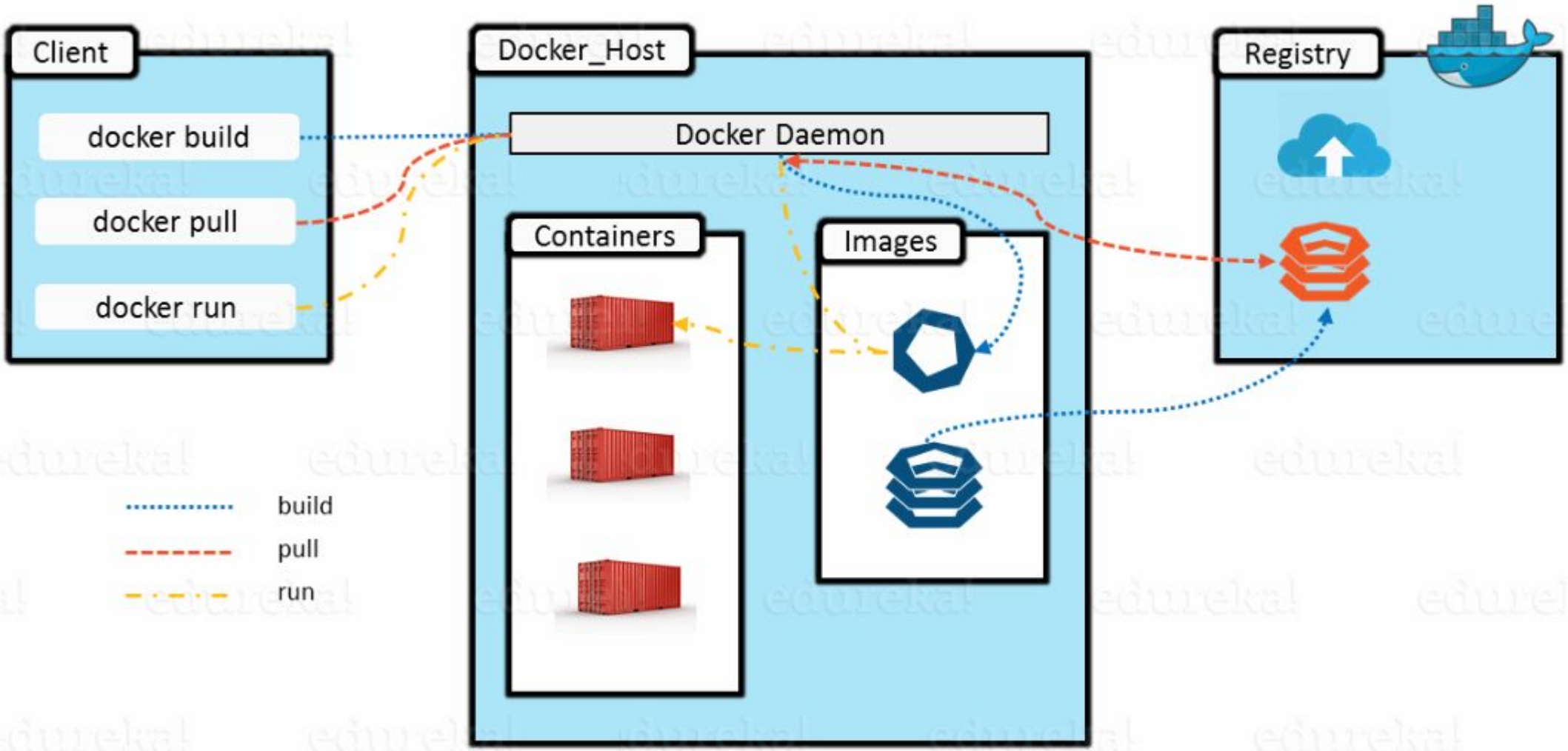
Docker architecture



- A client communicates with the Docker Daemon to execute commands

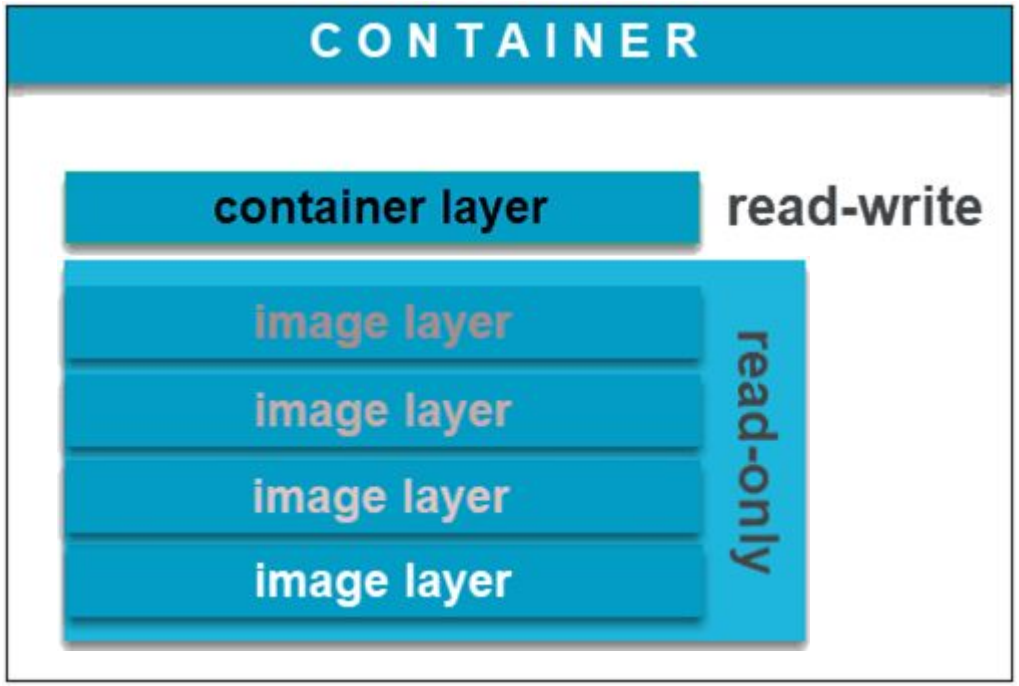


Docker in action

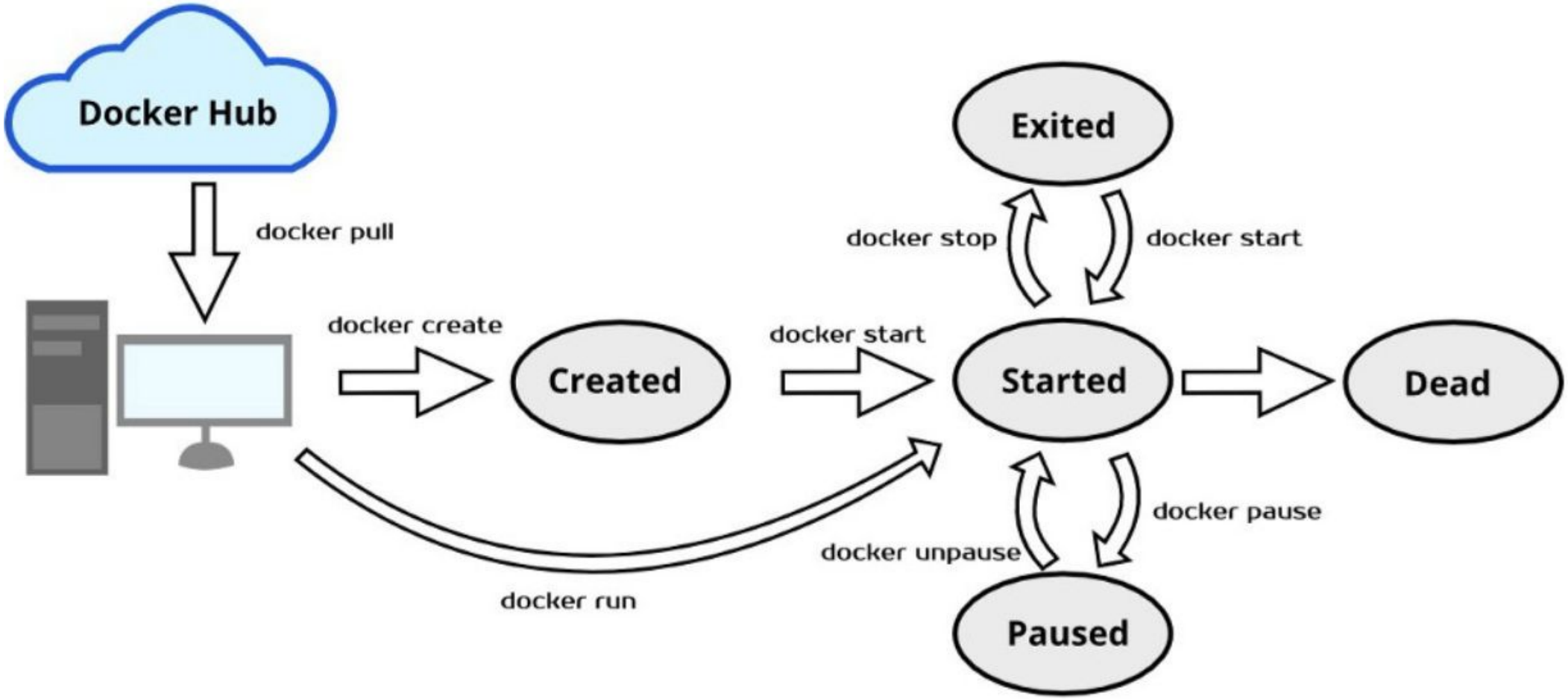


Docker commands: containers and images

- docker build:
  - Builds a new image from a Dockerfile
  - This is a read-only template and cannot be run
- docker pull/push:
  - Retrieves/sends an image from/to a registry
- docker create:
  - creates a container from an image with a writable layer on top and prepares it to run
- docker start:
  - executes a container
- docker run:
  - Creates a container and starts it (create + start)

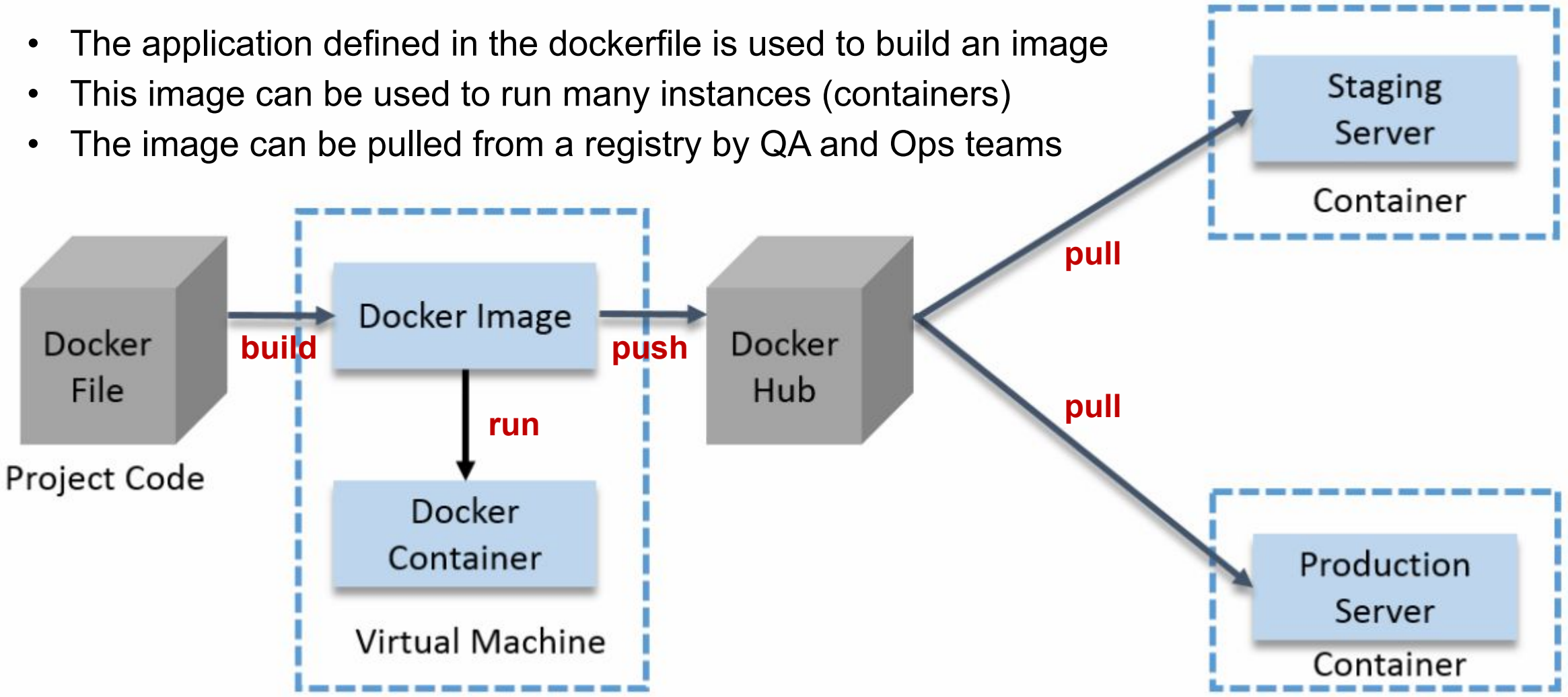


The lifecycle of a container



Defining, building and running containers

- The application defined in the dockerfile is used to build an image
- This image can be used to run many instances (containers)
- The image can be pulled from a registry by QA and Ops teams



Dockerfile (example for Linux)

```
1  # our base image
2  FROM alpine:latest
3
4  # Install python and pip
5  RUN apk add --update py-pip
6
7  # upgrade pip
8  RUN pip install --upgrade pip
9
10 # install Python modules needed by the Python app
11 COPY requirements.txt /usr/src/app/
12 RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
13
14 # copy files required for the app to run
15 COPY app.py /usr/src/app/
16 COPY templates/index.html /usr/src/app/templates/
17
18 # tell the port number the container should expose
19 EXPOSE 5000
20
21 # run the application
22 CMD ["python", "/usr/src/app/app.py"]
```

Python Package Index (package manager)

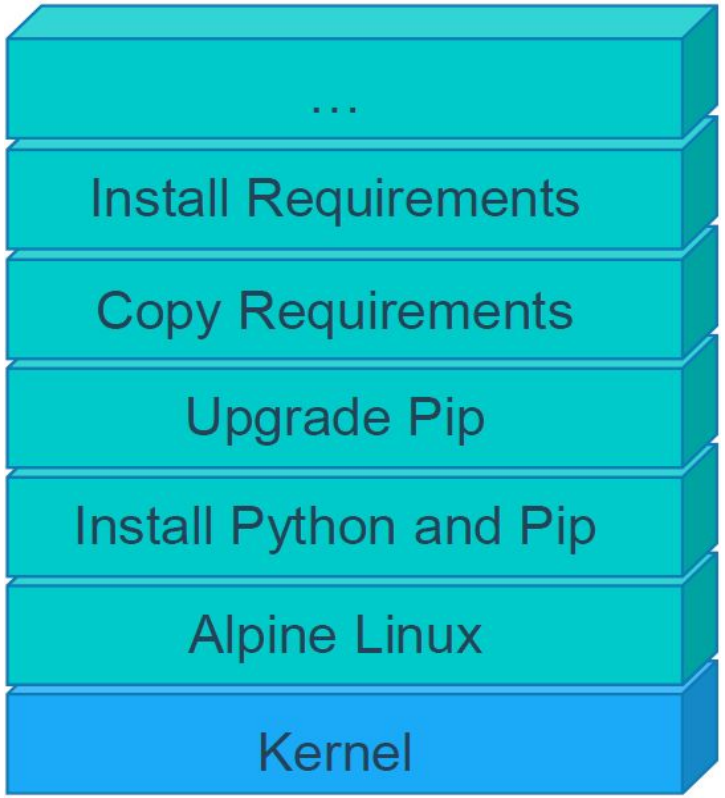
Install dependencies

- Instructions to build a Docker image
- Looks very similar to “OS native” commands
- FROM: parent image
- RUN: run command in a shell
- COPY: copy files from host to container file system
- EXPOSE: port to listen
- CMD: default setting to run container

Building an image from the Dockerfile

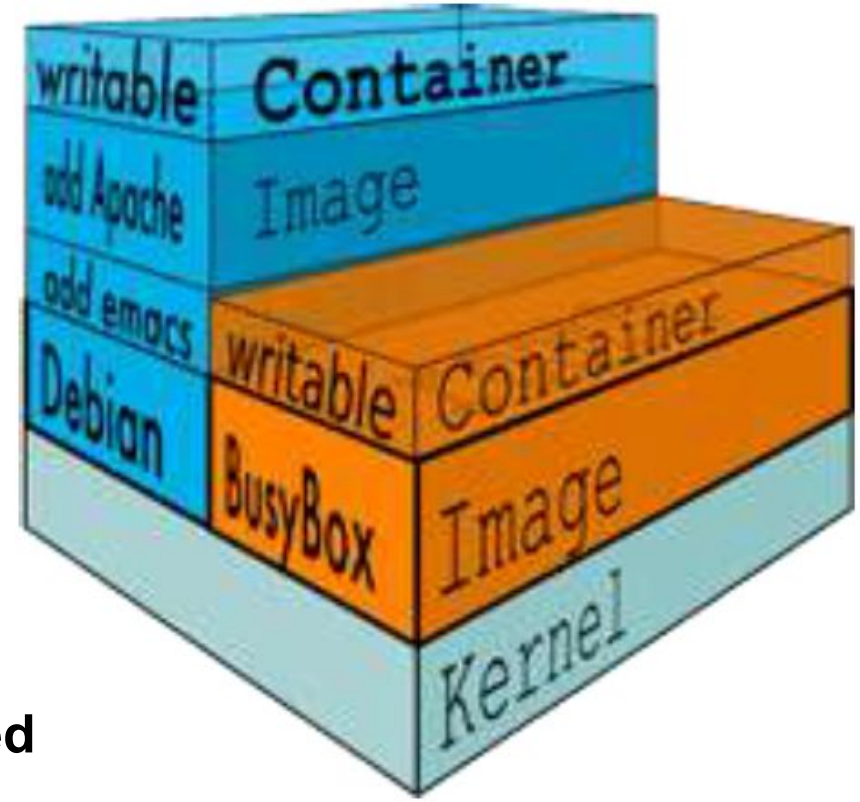
```
1  # our base image
2  FROM alpine:latest
3
4  # Install python and pip
5  RUN apk add --update py-pip
6
7  # upgrade pip
8  RUN pip install --upgrade pip
9
10 # install Python modules needed by the Python app
11 COPY requirements.txt /usr/src/app/
12 RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
13
14 # copy files required for the app to run
15 COPY app.py /usr/src/app/
16 COPY templates/index.html /usr/src/app/templates/
17
18 # tell the port number the container should expose
19 EXPOSE 5000
20
21 # run the application
22 CMD ["python", "/usr/src/app/app.py"]
```

- Each command stacks a new layer



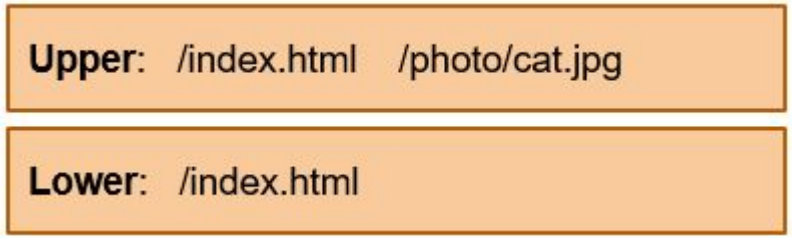
Images and containers

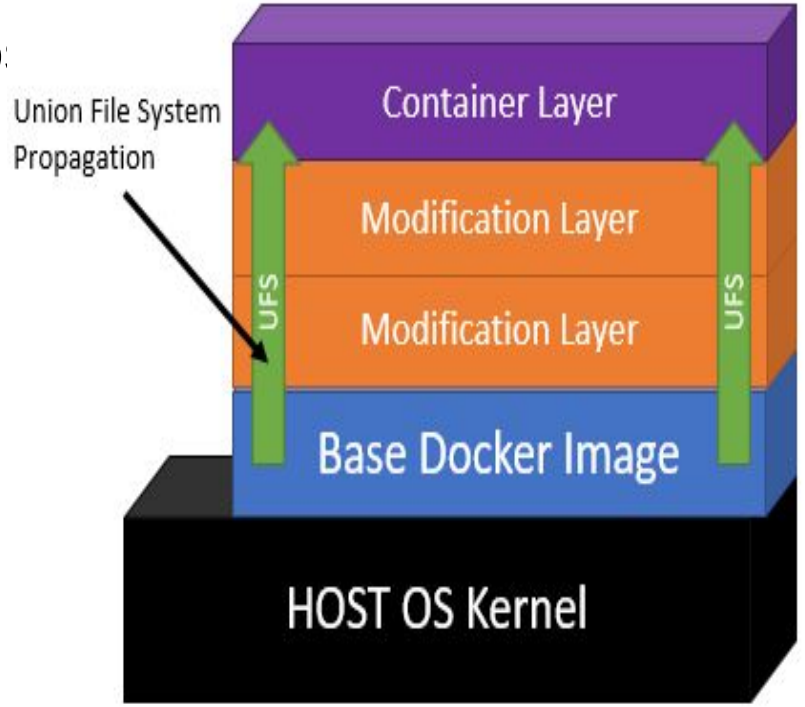
- An image is built with several layers:
  - Base image (pulled from a Docker registry)
  - Added files, binaries, libraries
  - App platforms (e.g., MongoDB, apache, node.js)
  - Configuration changes
  - Networking
  - Custom code
- A container is created from an image:
  - Adds a top layer (writable) as a working space
  - **Non-topmost layers** are read-only and can be **shared with other containers and cached**
  - Docker will pull the dependencies required to install/run



Union file system

- Each layer has its own set of files
- Each layer “sees” its set of files merged with tho
  file system)

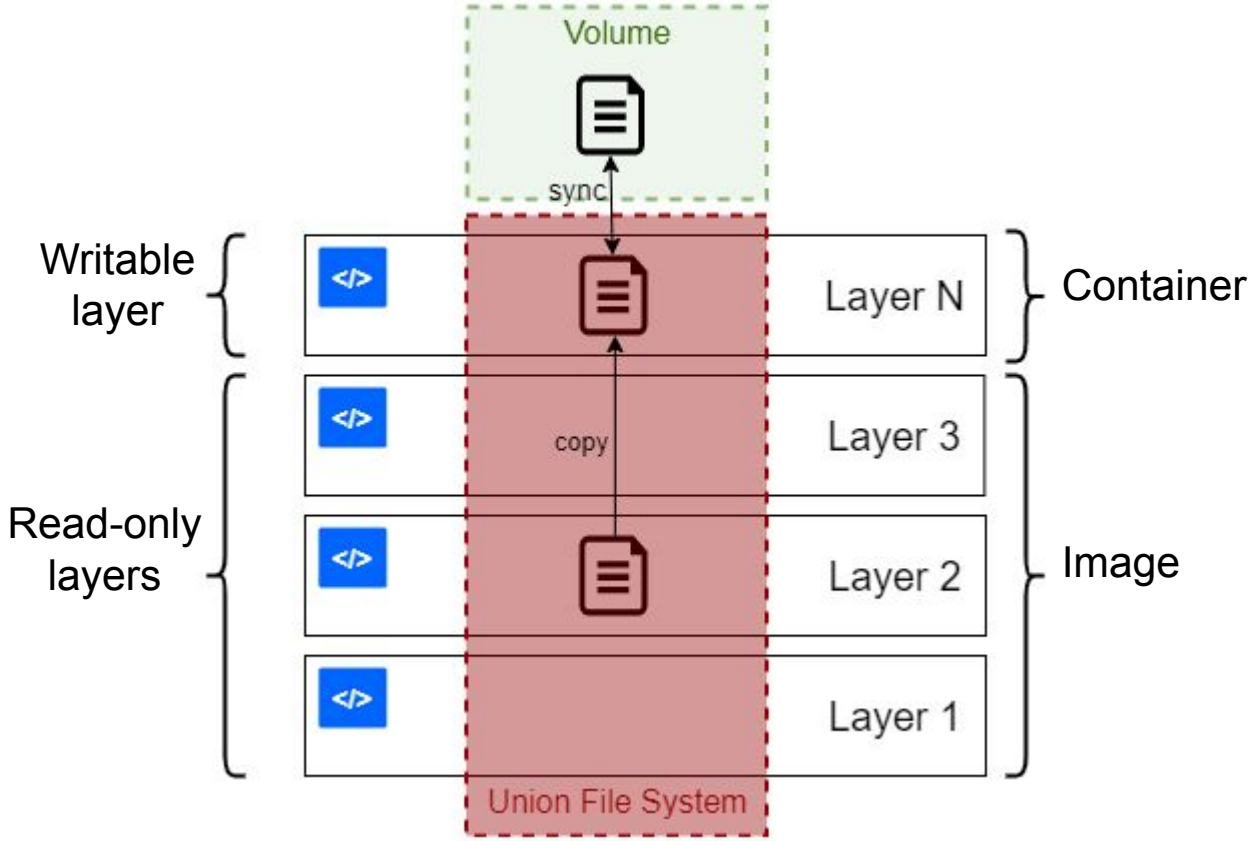




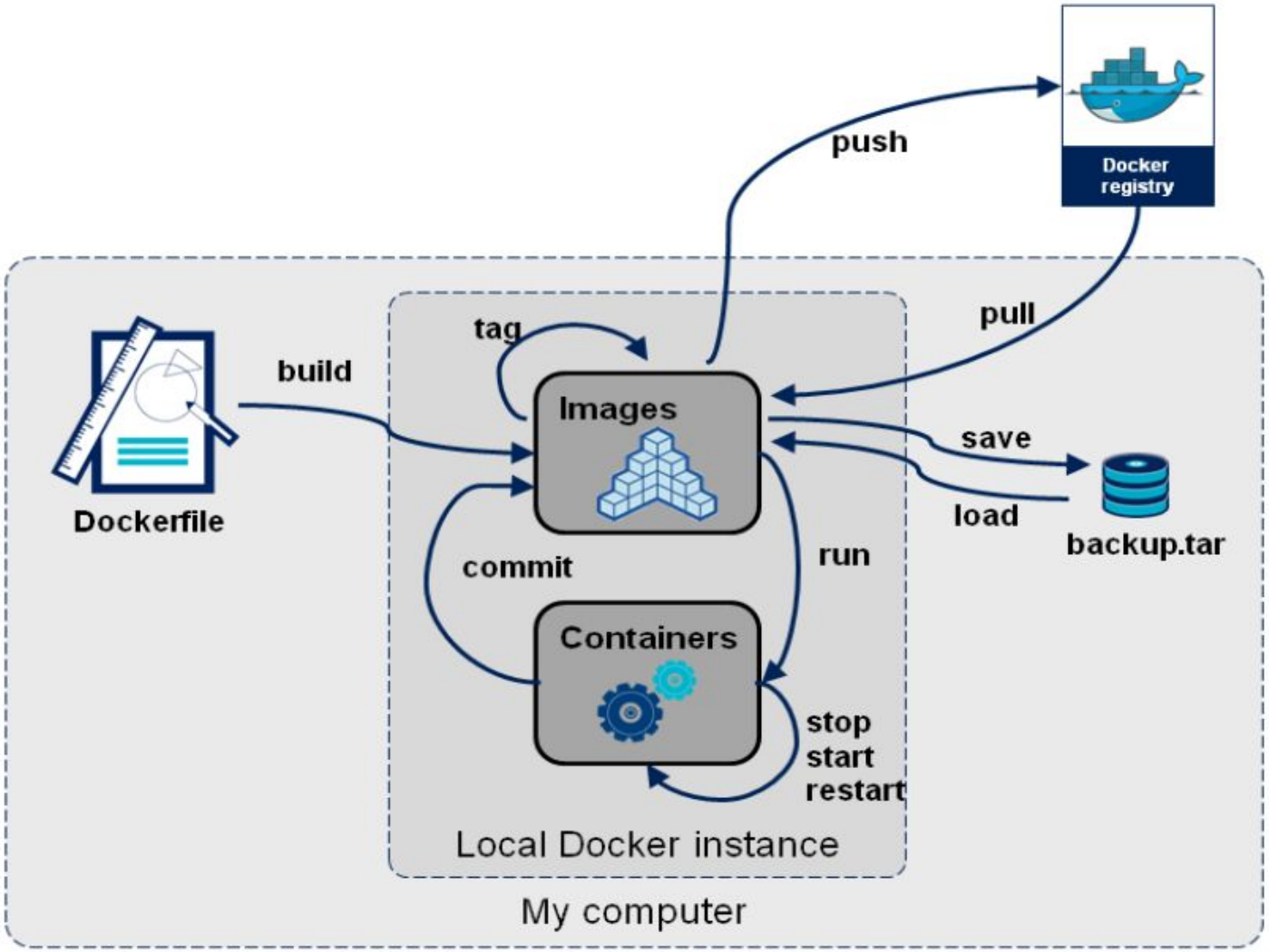
- In case of duplicates, the upper layer gets priority (layers can modify the context produced by lower layers)
- Files are immutable, except those created in the topmost (writable) layer

Volumes (persistent storage)

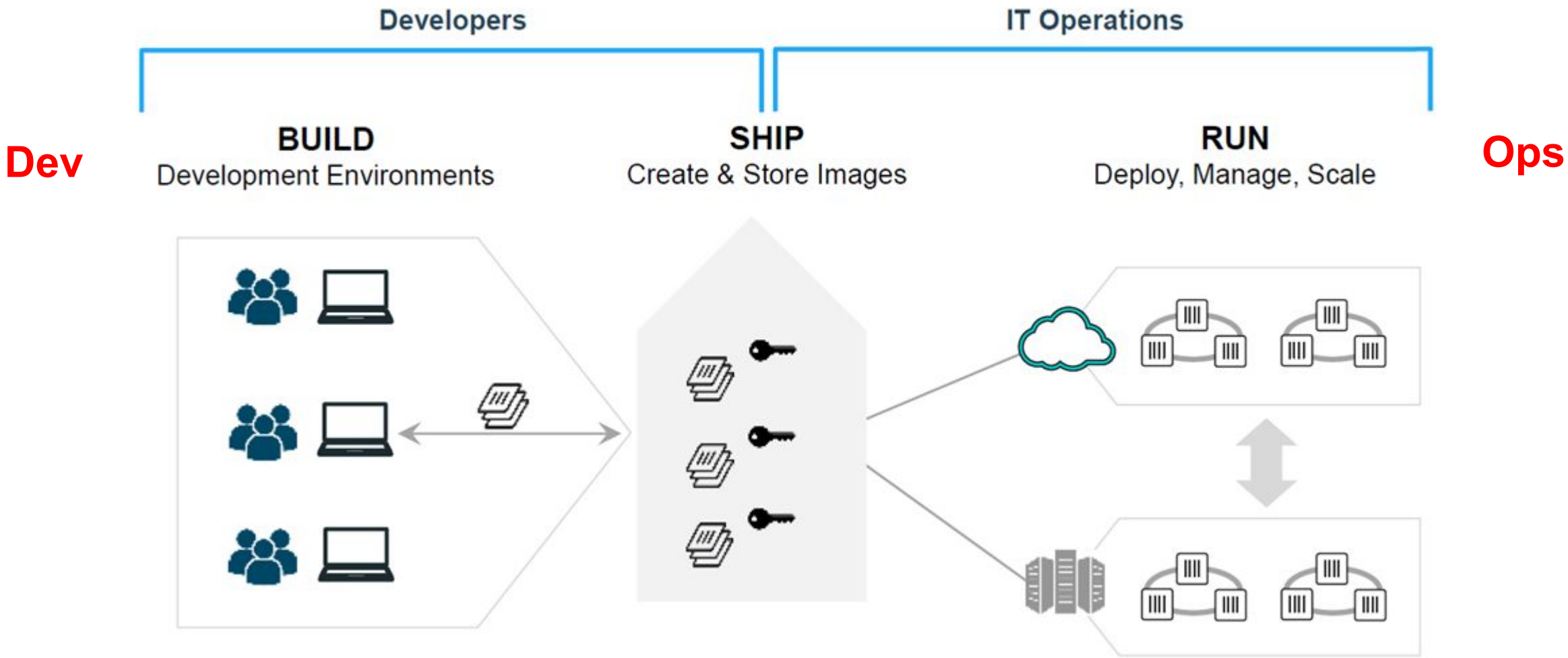
- What if the container wants to change a file in a read-only layer?
  - CoW (Copy-on-Write)
  - The changed copy hides the original
- When the container is stopped or deleted, the writable layer data (and changes) are lost
- Volumes (**persistent storage**):
  - can be mounted and shared between containers
  - They use host storage, but under container control
  - Volumes persist even after containers have been deleted



Developing and deploying containers

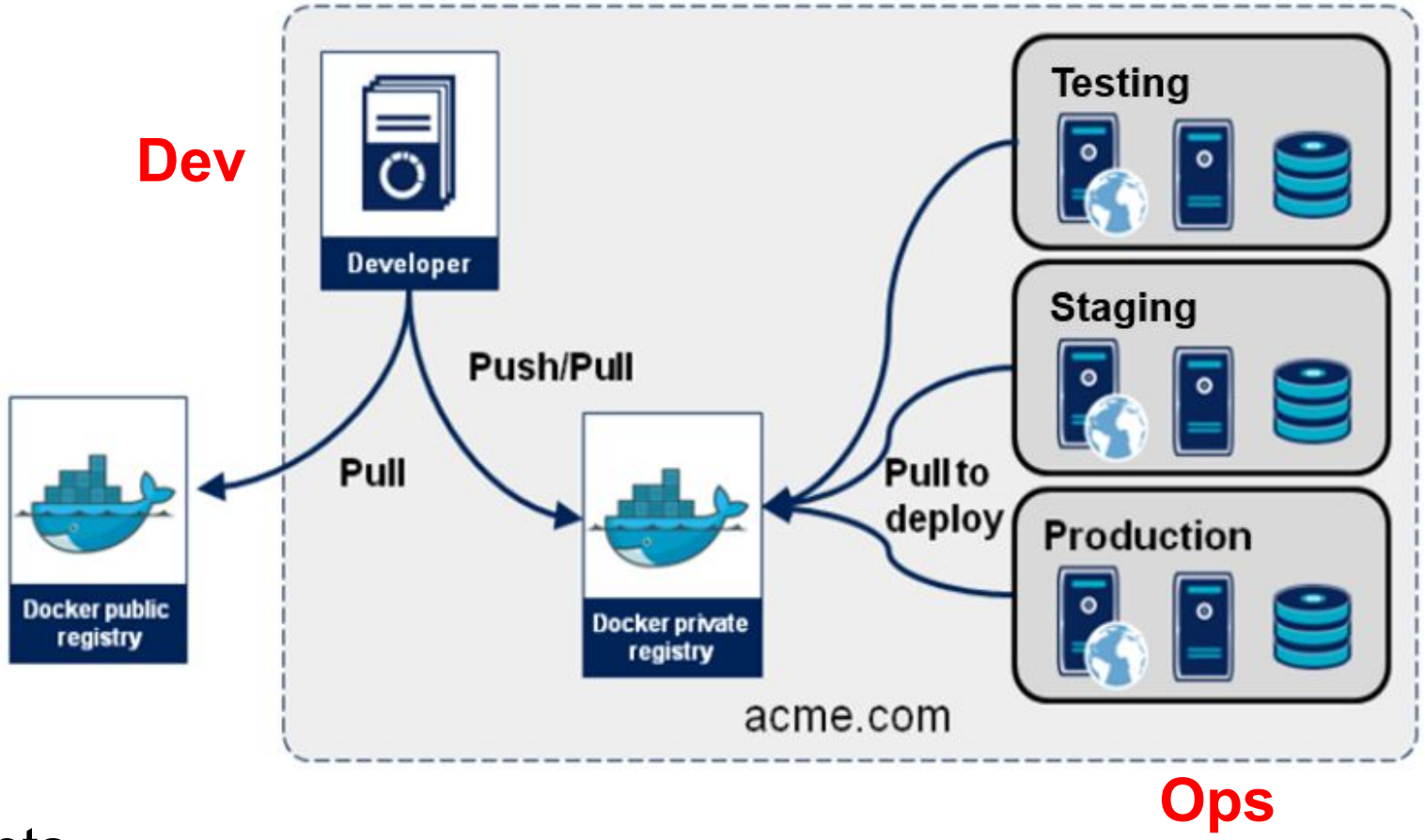


Containers: from Dev to Ops

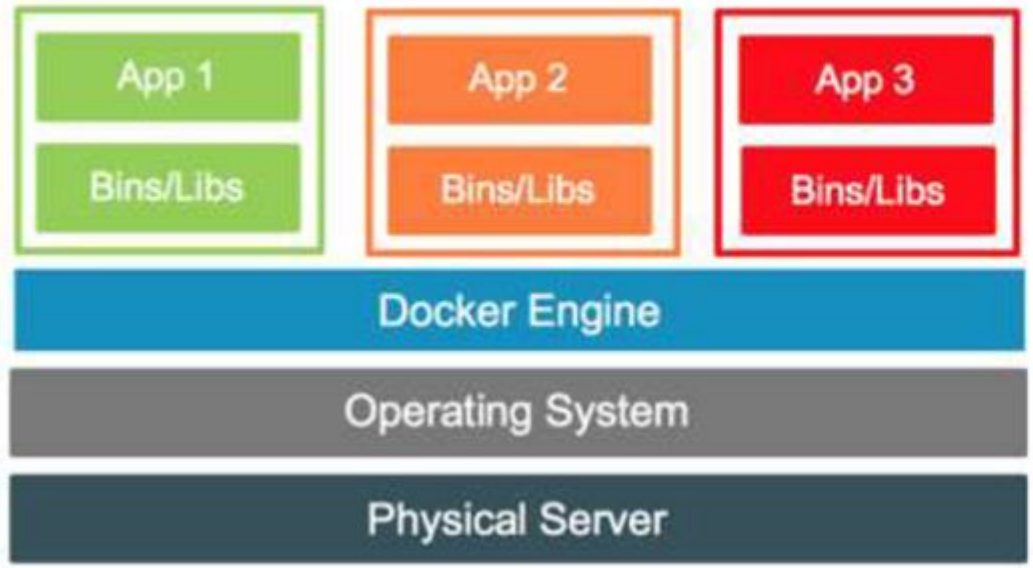


Using the Docker ecosystem

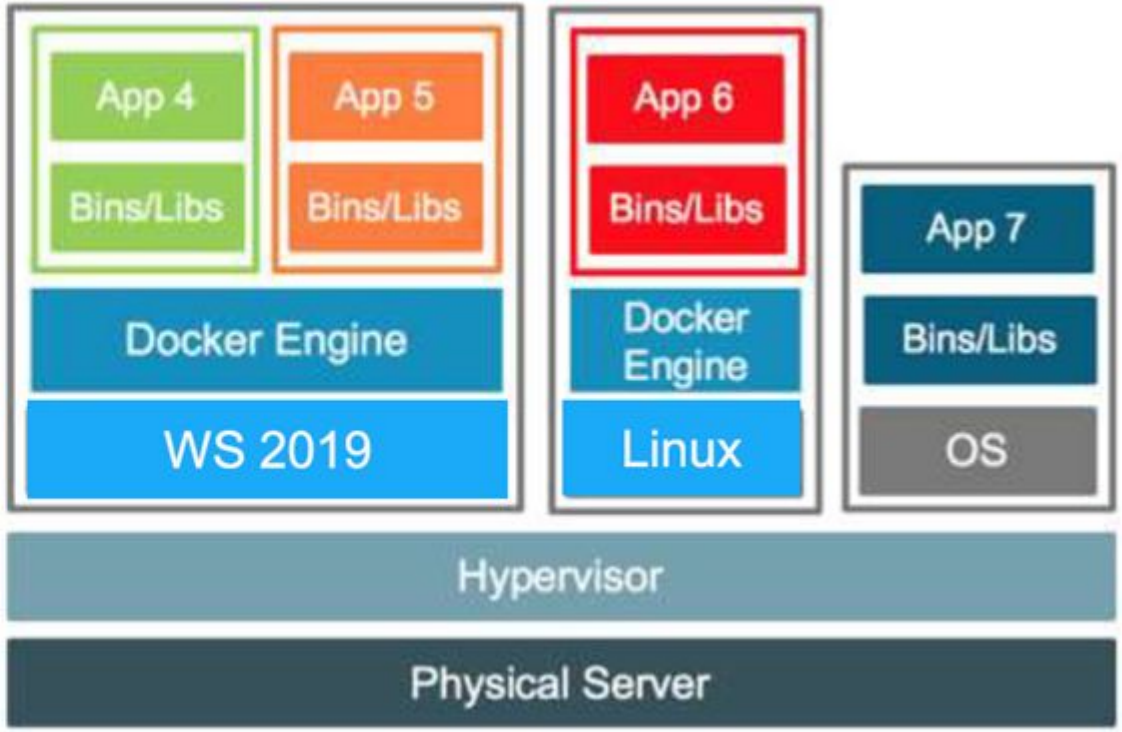
- Registries store and version container images
- Dev use the Docker Hub public registry to get ready-to-use container images
- Dev store new images in the private registry
- Dev build containers once
- Ops pull and deploy them anywhere
- Portable between environments



Containers can coexist with other technologies



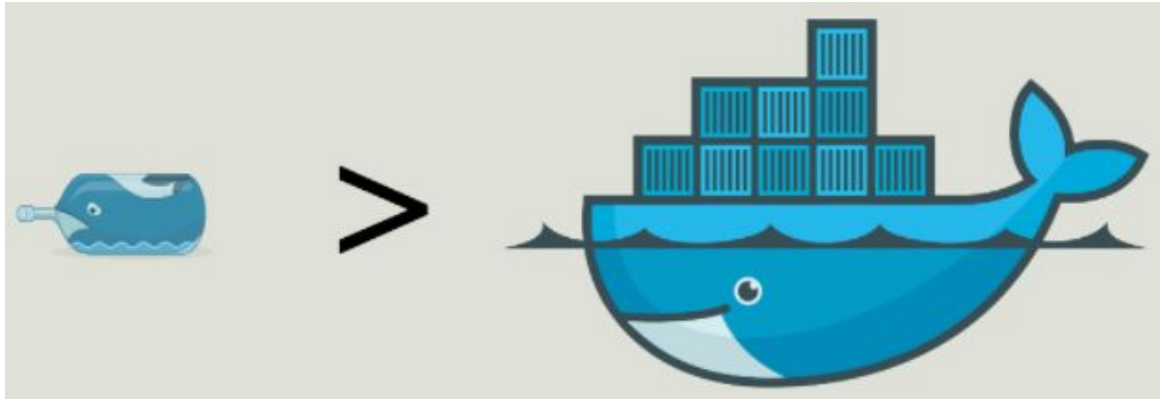
Containers on a physical host

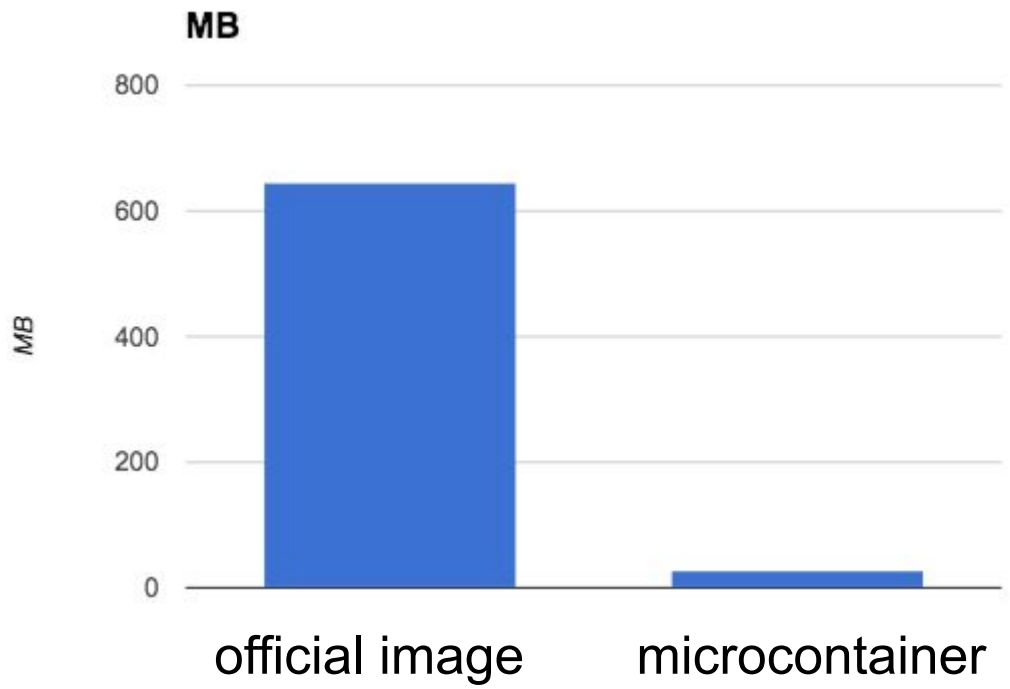


Containers on virtual machines

Microcontainers: small is beautiful

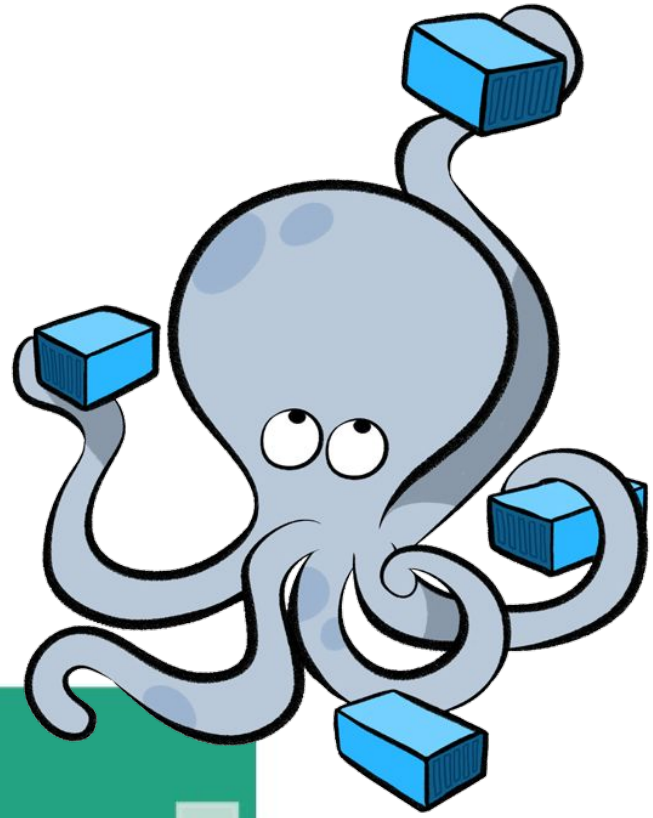
- Docker packages an application along with all its dependencies into a nice self-contained image
- But Docker's official repositories are usually **large bundles** (e.g., language and system libs)
- For simple applications, start with "FROM: **scratch**" and add just the dependencies needed
- A "Hello world" example in Node.js:
  - official image (644 MB)
  - Scratch image + Node.js (29 MB)





Docker Compose

- Docker Compose is a tool for defining and running **multi-container applications** with Docker
- A multi-container application is defined in a **single file**
- A single command does everything that needs to be done to get the application up and running

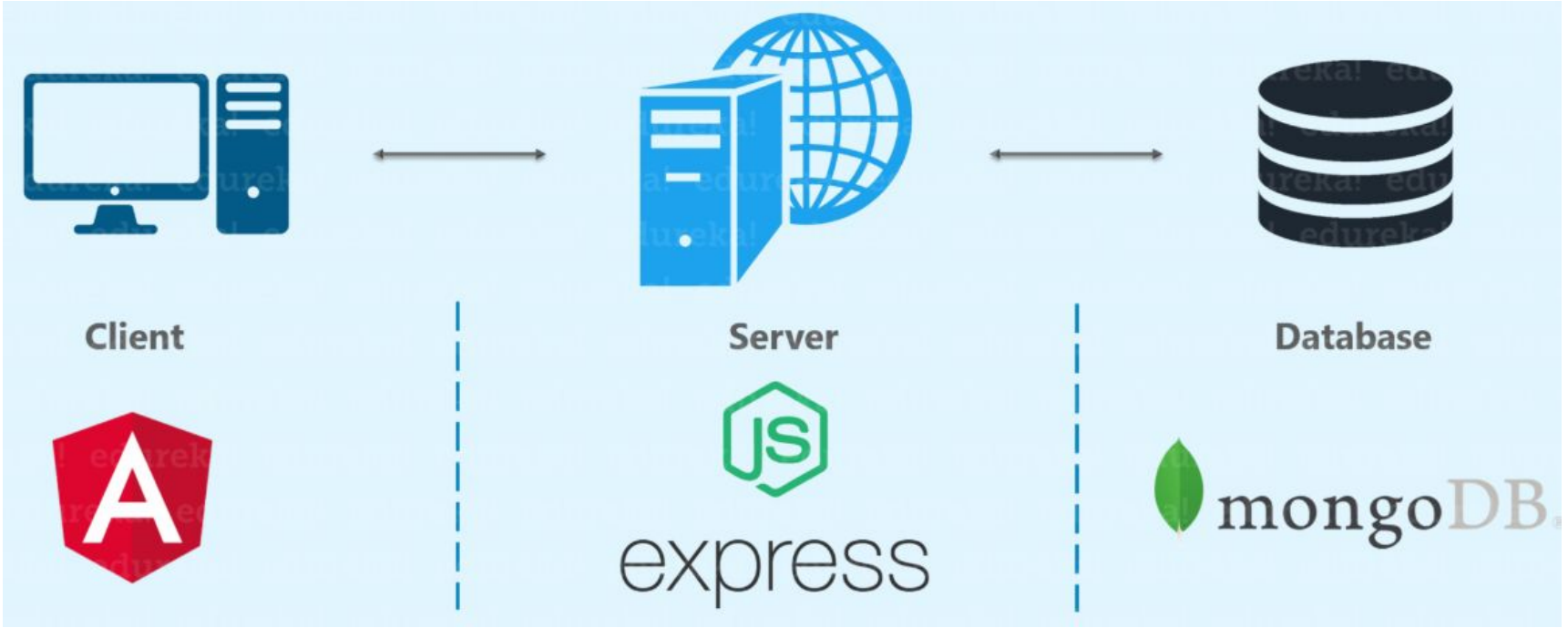


Define your app's environment with a Dockerfile so it can be reproduced anywhere.

Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment:

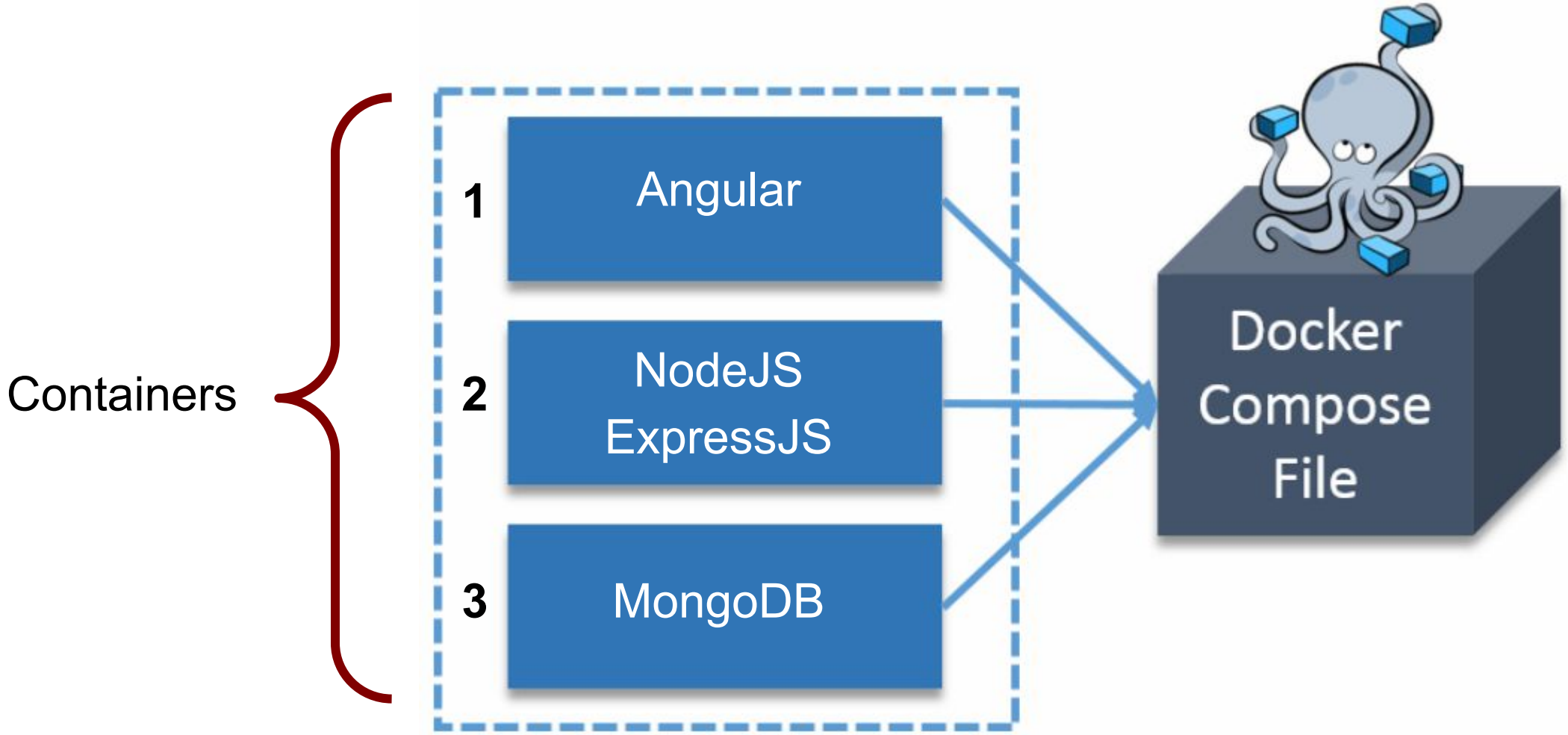
Lastly, run docker-compose up and Compose will start and run your entire app.

Example: full stack web application



- Angular – Front end web application platform
- NodeJS – Server-side JavaScript run-time environment
- ExpressJS – Back end web application framework for Node.js
- MongoDB – Back end NoSQL database

Docker Compose file



Dockerfiles for front end and back end

Dockerfile for Angular

```
1  FROM node:6
2  RUN mkdir -p /usr/src/app
3  WORKDIR /usr/src/app
4  COPY package.json /usr/src/app
5  RUN npm cache clean
6  RUN npm install
7  COPY . /usr/src/app
8  EXPOSE 4200
9  CMD ["npm","start"]
```

- Pull a base “node:6” image
- Create a directory and make it the working directory
- “package.json” contains dependencies needed by NPM (JavaScript package manager) to build Angular
- Clear the NPM cache and download what is needed
- Copy everything to the container’s working directory
- Expose port 4200 for communicating with back end server
- Run NPM to build the Angular application

Dockerfile for Node & Express

```
1  FROM node:6
2  RUN mkdir -p /usr/src/app
3  WORKDIR /usr/src/app
4  COPY package.json /usr/src/app
5  RUN npm cache clean
6  RUN npm install
7  COPY . /usr/src/app
8  EXPOSE 3000
9  CMD ["npm","start"]
```

- Similar to the previous Dockerfile, but:
  - “package.json” now refers to nodeJS and ExpressJS
  - Port exposed is 3000
- The database container will use a MongoDB registry image (no need for a dockerfile)

Docker Compose File (YAML)

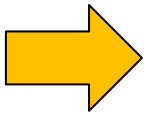
```yaml
version: '3.0' # specify docker-compose version

# Define the services/ containers to be run
services:
 angular: # name of the first service
  build: angular-app # specify the directory of the Dockerfile
  ports:
  - "4200:4200" # specify port mapping

 express: # name of the second service
  build: express-server # specify the directory of the Dockerfile
  ports:
  - "3000:3000" #specify ports mapping

 database: # name of the third service
  image: mongo # specify image to build container from
  ports:
  - "27017:27017" # specify port forwarding
```

- Docker Compose version
- “services” defines the three services (containers)
- “build” specifies a directory with a dockerfile
- Each service specifies the port number to use to receive/send requests from/to other services
- Service “database” does not build an image, since it uses an already existing image

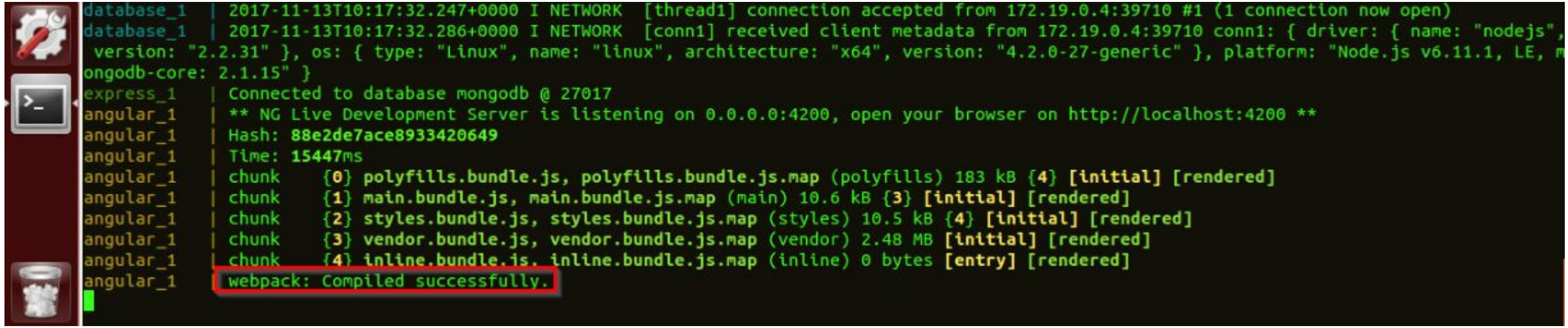
```
docker-compose build
docker-compose up
```

- Build / rebuild the services
- Create / start the containers

Executing the Docker Compose file



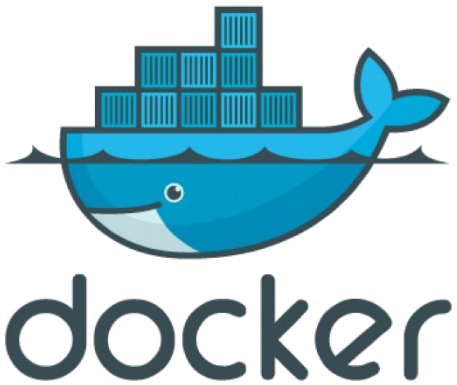
Running the containers



- Services accessible at:
  - localhost:4200 – Angular App (Front-end)
  - localhost:3000 – Express Server & NodeJS (Back-end/ Server-side)
  - localhost:27017 – MongoDB (Database)

Docker vs Vagrant

- Docker is an open-source platform for **building**, **shipping**, and **running** distributed applications (as containers)
  - Emphasis on application **portability**
- Vagrant is an open-source software tool for **building** and **maintaining** portable **virtual** software development environments in a reproducible way
  - Emphasis on development environment (entire system) portability/reproducibility
- Both Docker and Vagrant:
  - are based on configuration files (Dockerfile and Vagrantfile)
  - solve the problems related to sharing environment setups so that multiple instances of the same application act in a predictable, reproducible, and repeatable manner



Docker and Vagrant

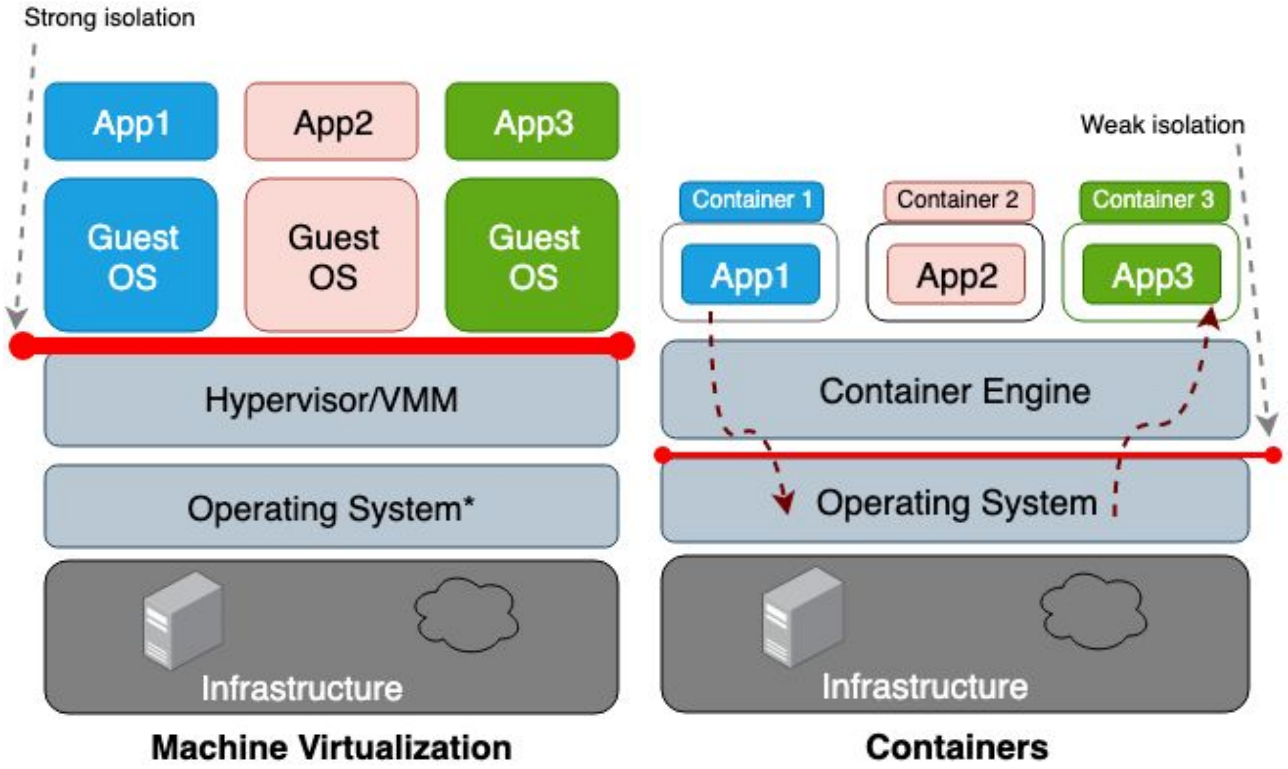
- Docker can be used as a **Vagrant provider**:
  - Containers instead of virtual machines
  - Specify the image rather than the box
  - Image automatically pulled from a registry
  - If the host does not support Docker containers natively, Vagrant automatically spins up a "host VM" to run Docker
- Vagrant can also be used to **build custom images** from a Dockerfile
  - Specify the directory where the Dockerfile is (same as the Vagrantfile in this example)
  - “vagrant up” automatically builds the image from the Dockerfile and runs the container

```
Vagrant.configure("2") do |config|
  config.vm.provider "docker" do |d|
    d.image = "nginx:latest"
    d.ports = [“8080:80”]
    d.name = “nginx-container”
  end
end
```

```
Vagrant.configure("2") do |config|
  config.vm.provider "docker" do |d|
    d.build_dir = "."
  end
end
```

“VMs vs containers” dilemma

- Virtual Machines
  - Enhanced security and isolation
  - Full OS, heavyweight
  - Large footprint, long boot times
- Containers
  - Performance and resource efficiency
  - OS not included, lightweight
  - Less secure than virtual machines (OS sharing, wider surface of attack)



- What if we could have both benefits?
  - The enhanced security and isolation of virtual machines, and
  - The performance and resource efficiency of containers

Agenda

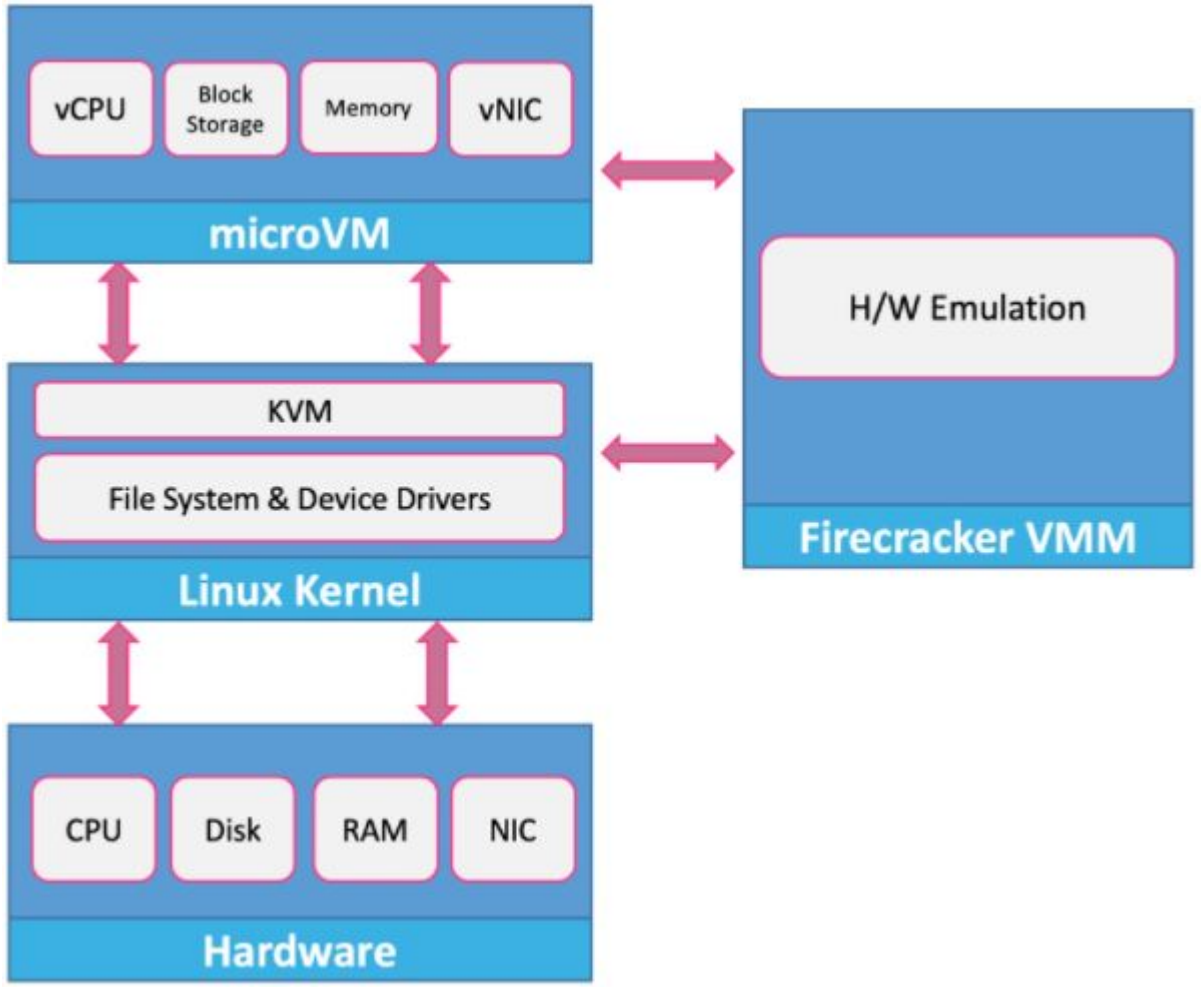
- Virtual machines
- Hypervisors
  - Full-virtualization
  - Paravirtualization
  - Hardware-assisted virtualization
- Virtualization of virtual memory
- I/O virtualization
- Virtual machine migration
- Automation of VM creation
- Containers
- MicroVMs and Unikernels

Solutions to lightweight isolation

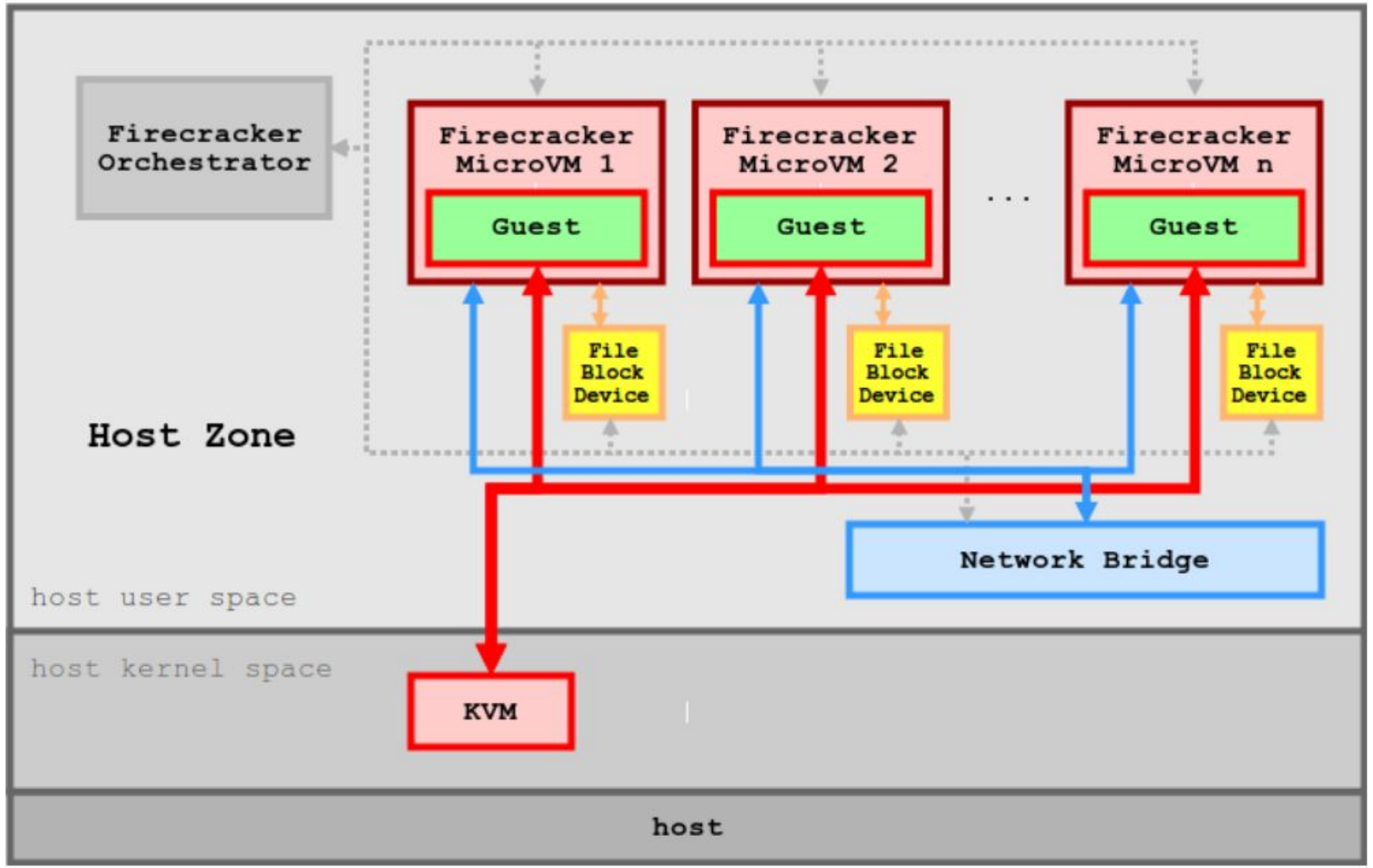
- MicroVMs (trimmed down hypervisor)
  - Provide only a minimal set of features and emulated devices
  - Footprint as low as 5 MB and boot times as low as 125 milliseconds
  - Example: Firecracker (originally, an AWS project)
- Unikernels (recompile the kernel, use only what is needed)
  - Lightweight, immutable OS compiled specifically to run a single application (one process), with the minimal device drivers and OS libraries necessary to support the application
  - The result is a machine image that can run without the need for a host operating system
  - Example: MirageOS
- Container sandboxes (restrict system calls)
  - Provide a “kernel proxy” for each container (an intermediate layer of control and isolation)
  - Does not require recompilation or kernel downsizing, but entail a performance penalty
  - Example: gVisor (Google project)

MicroVMs: Firecracker

- Trimmed-down type-2 VMM (50K lines of code, compared to the 1.4 million lines of QEMU)
  - Minimalist functionality and I/O emulation
  - Millisecond launch time can be as low 125ms
  - Memory footprint as low as 5 MB
- Relies on Kernel Virtual Machine (hypervisor in every Linux)
- Optimized for simple, transient, low-latency, and high numbers of processes

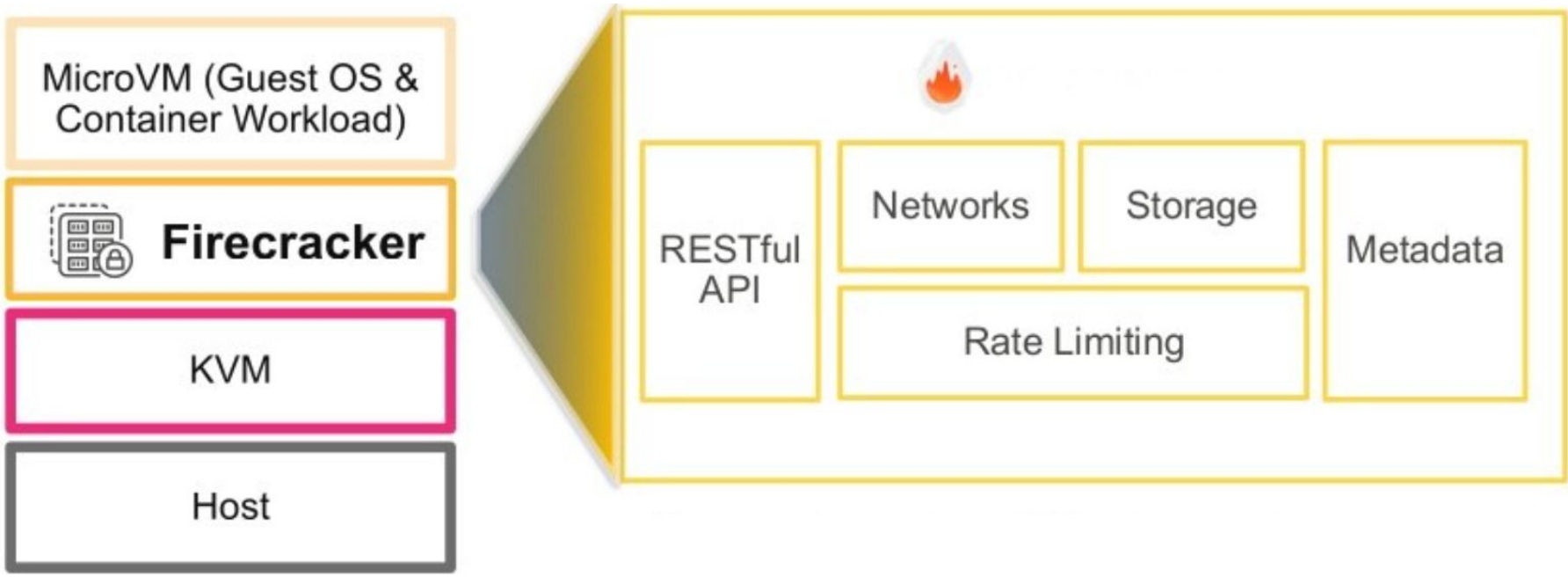


Firecracker architecture



Firecracker management

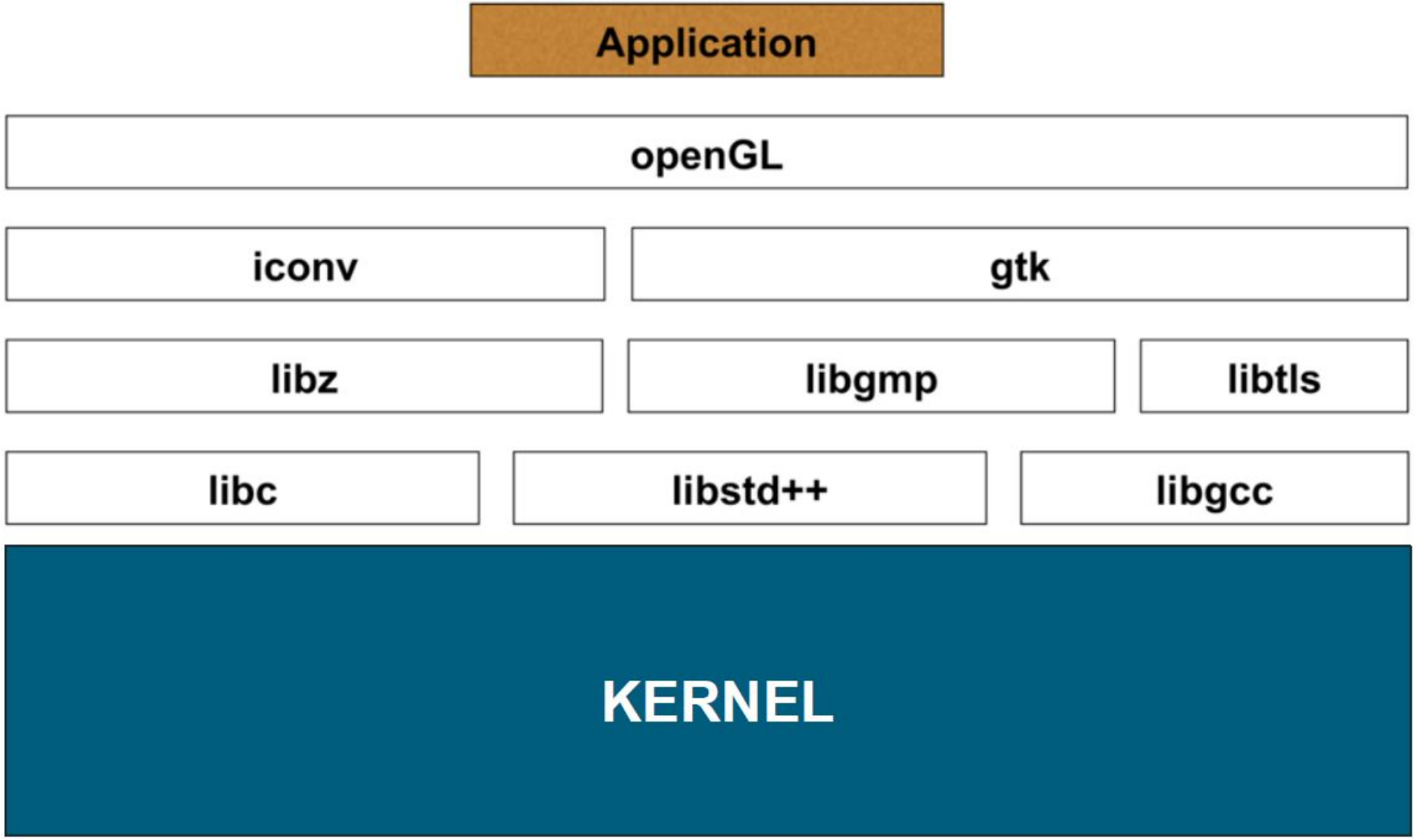
- RESTful API to , e.g.:
  - Spin up and configure the microVMs
  - Rate limiter: allocates storage and network resources for regular and burst activity levels
  - Metadata service: how configuration information is securely shared between the guest and host OS



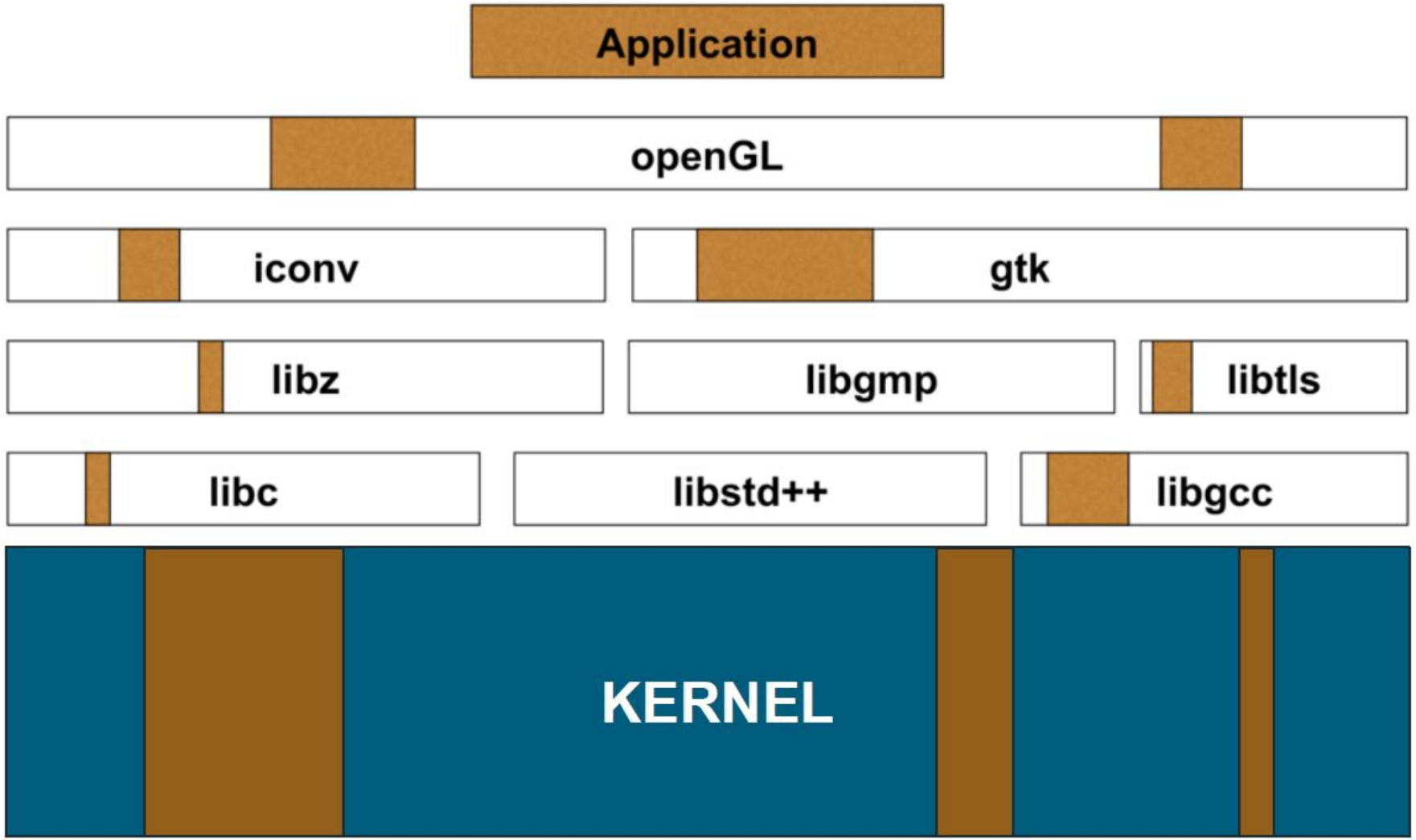
Unikernels: rightsizing the application stack

- No matter how simple the application API, the OS kernel is always a huge dependency
- The Linux kernel is around **28 million lines of code**
- This represents more than a third of the full OS
- Most applications require only a fraction of the kernel functionality

Traditional software stack



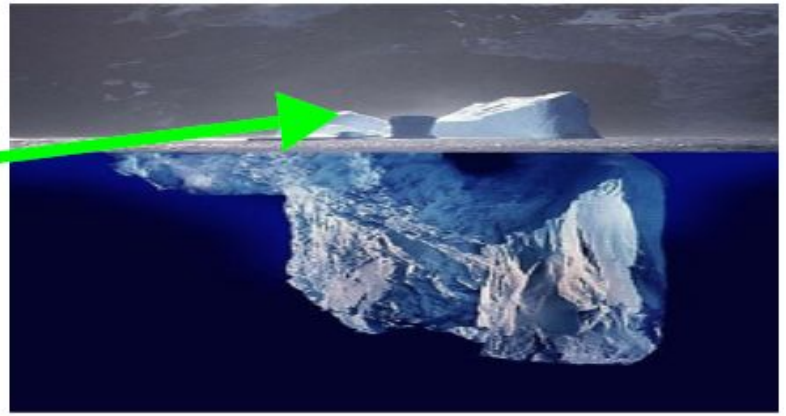
Only a small part is actually used



The Unikernel concept

- Compile your source code bundled with a custom “operating system”
- This includes only the functionality required to support the application logic
- This can run directly on top of a hypervisor or on bare metal (no host OS)

Code you want to run



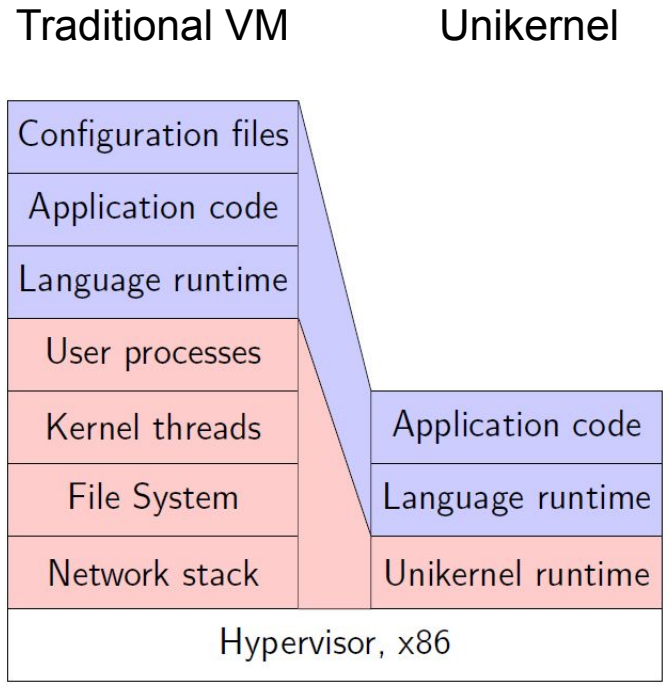
+

Operating system libraries

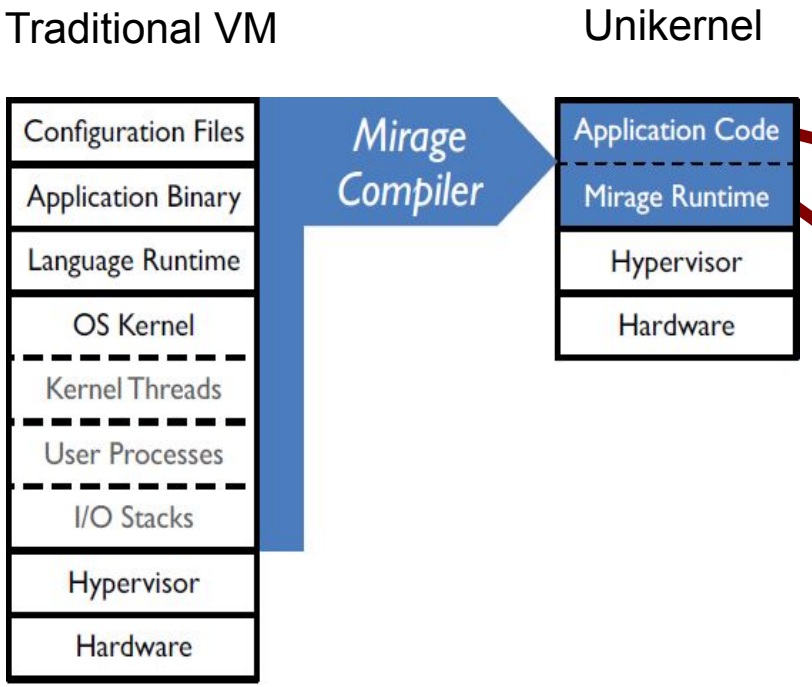
=

Standalone unikernel

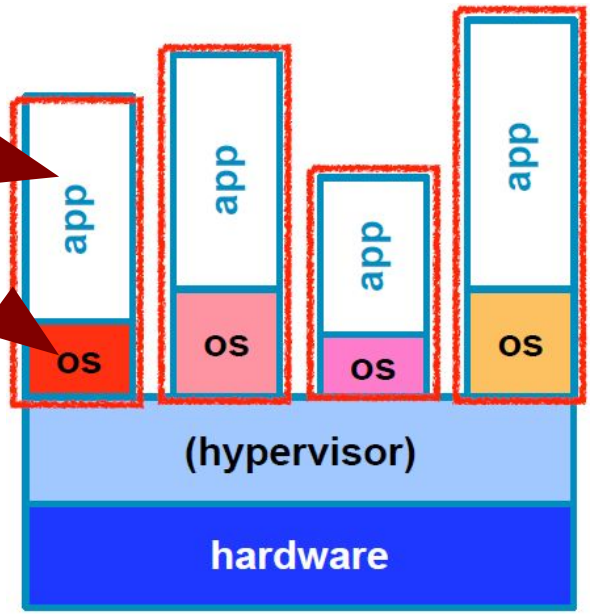
Unikernels: rightsizing running code



Unikernel concept



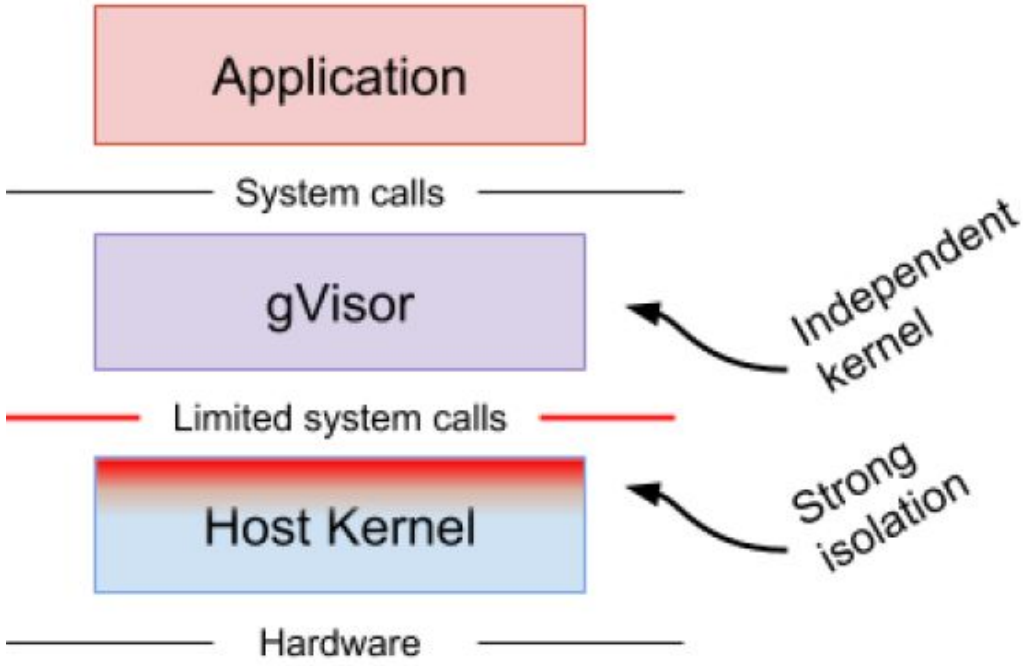
MirageOS example



Running unikernels
(rightsized)

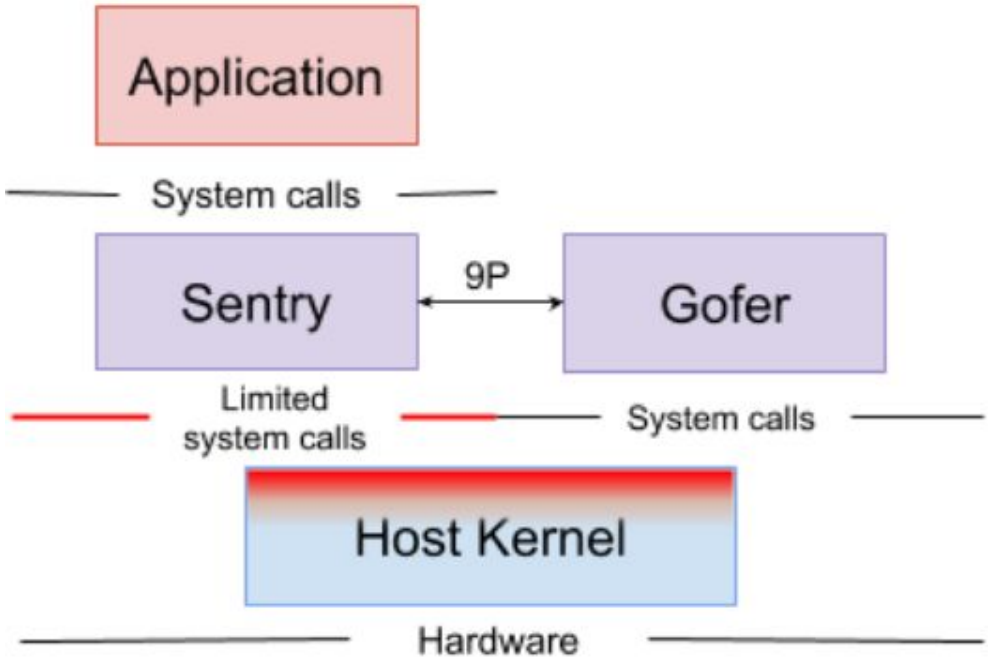
Container sandboxing: gVisor

- gVisor (Google) intercepts application system calls and acts as the guest kernel, without the need for translation through virtualized hardware
- Can be thought of as a merged guest kernel and VMM
- It provides a flexible resource footprint (based on threads and memory mappings, not fixed guest physical resources)
- But this incurs in a higher per-system call overhead

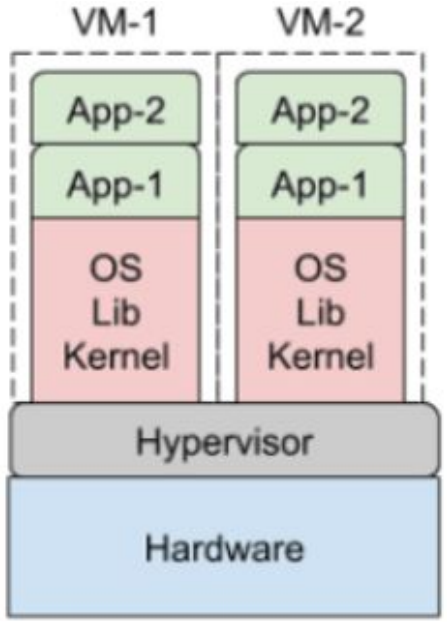


Components of gVisor

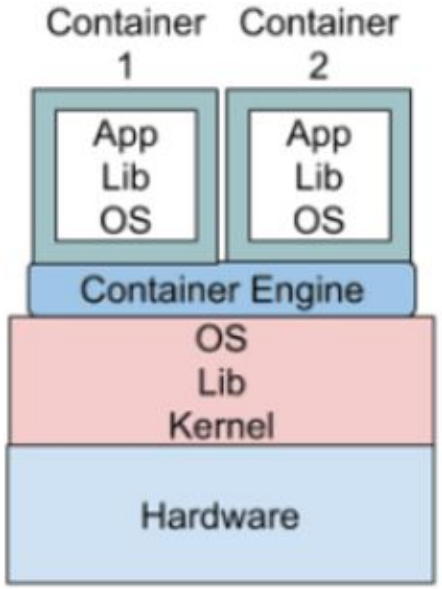
- A gVisor sandbox consists of multiple processes, which provide an environment in which one or more containers can be run.
- Each sandbox has its own isolated instance of the **Sentry**:
  - a kernel that runs the containers and **intercepts** and **responds** to **system calls** made by the application
- Each container running in the sandbox has its own isolated instance of a **Gofer**:
  - Provides **controlled file system** access to the containers (accessed via the 9P protocol)



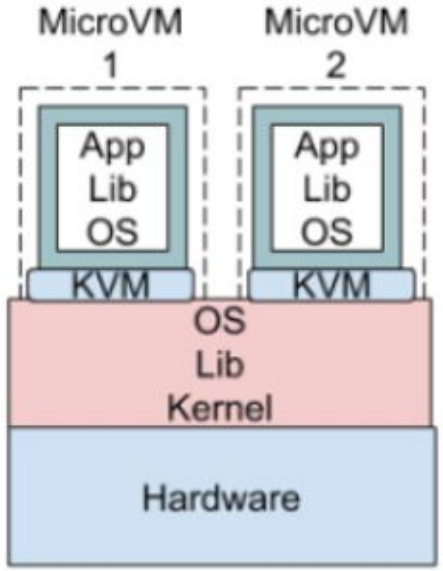
Comparison of main virtualization strategies



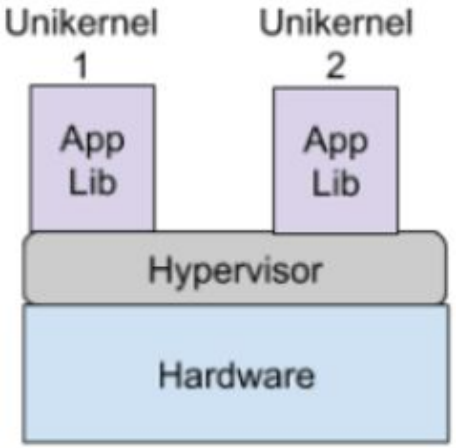
Virtual machines



Containers



MicroVMs



Unikernels

Thank you! Any questions?