

Virtualization

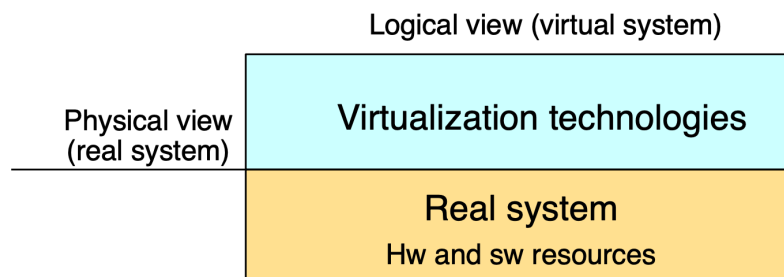
Corso di Sistemi Distribuiti e Cloud Computing A.A. 2024/25

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Virtualization

- High-level abstraction to hide details of underlying implementation
- Abstraction of **computing resources**
 - Logical view different from physical one

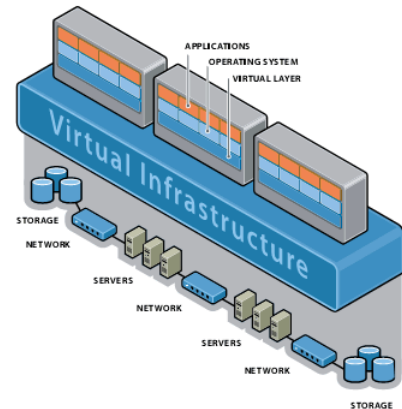


- How? **Decouple** user-perceived architecture and behavior of hw and sw resources from their physical realization
- Goals:
 - Portability, efficiency, reliability, security, ...

Virtualization of resources

- **System (hw and sw) resources virtualization**
 - Virtual machines, containers, unikernels, ...
- Storage virtualization
 - Storage Area Network (SAN), ...
- Network virtualization
 - Virtual LAN (VLAN), Virtual Private Network (VPN), ...
- Data center virtualization

← Our focus

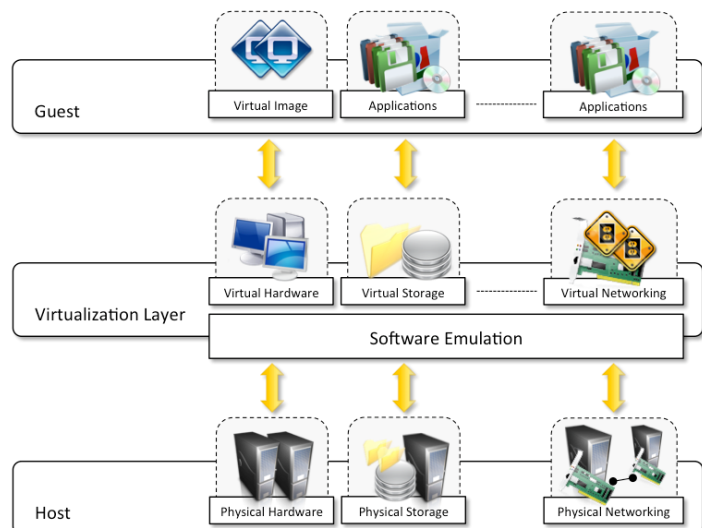


Valeria Cardellini - SDCC 2024/25

2

Components of virtualized environment

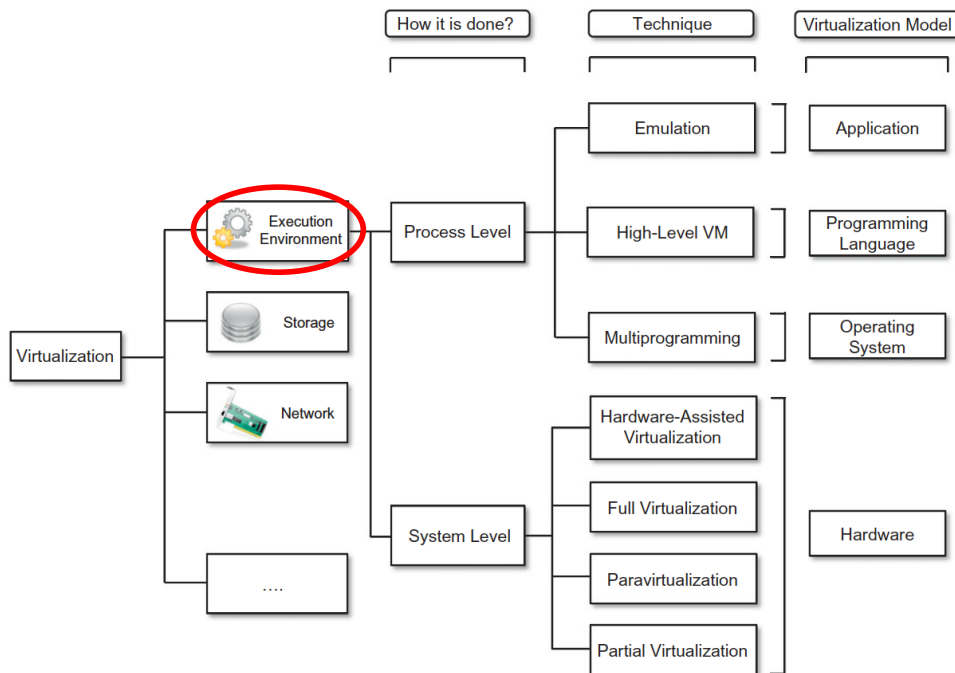
- Major components:
 - Guest
 - Host
 - Virtualization layer
- **Guest:** interacts with virtualization layer rather than with host
- **Host:** original environment where guest is supposed to be managed
- **Virtualization layer:** responsible for recreating same or different environment where guest will operate



Valeria Cardellini - SDCC 2024/25

3

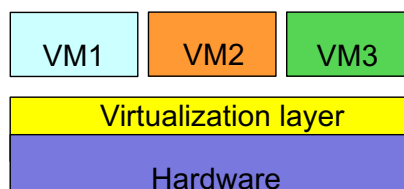
Taxonomy of virtualization techniques



- **Execution environment virtualization** is the oldest, most popular and developed area ⇒ our focus

Virtual Machine

- **Virtual Machine (VM)**: complete compute environment with its own isolated processing capabilities, memory, and communication channels
- Allows us to represent hw/sw resources of a physical machine differently from their reality
 - E.g., VM hw resources (CPU, network card, ...) different from physical resources of the real machine
 - E.g., VM sw resources (OS, ...) different from sw resources of the real machine
- A single physical machine can be used to host several VMs



Virtualization: a short history

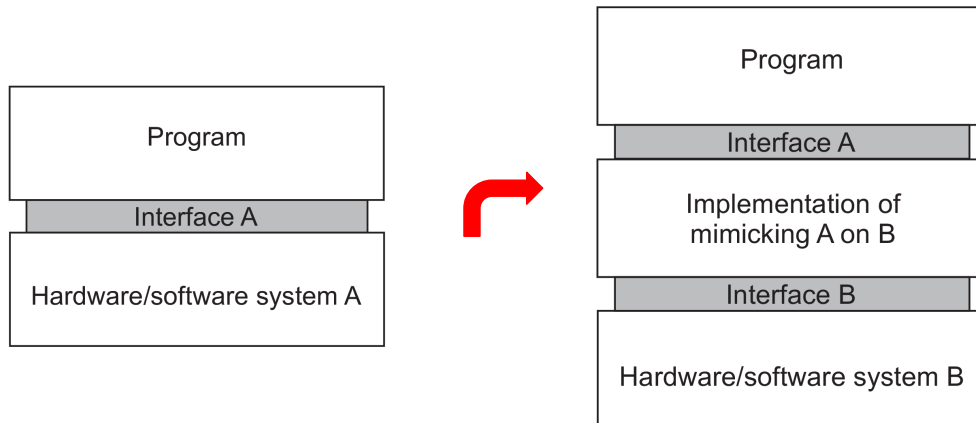
- Virtualization and VMs are an “old” idea in computer science
 - Dates back to the 1960s in a centralized context
 - Designed to allow legacy (existing) software to run on expensive mainframes and transparently share (scarce) physical resources
 - E.g., IBM System/360-67 mainframe
- In the 1980s, with the transition to PCs, the problem of transparently sharing computing resources was solved by multitasking OSs
 - Virtualization became less of an issue

Virtualization: a short history

- At the end of the 1990s, interest in virtualization revamped to make programming special-purpose hw less burdensome
 - VMware founded in 1998
- Moreover, management costs and under-utilization of hw and sw platforms exacerbate the need for virtualization solutions
 - Hw changes faster than sw (middleware and applications)
 - Management cost increases and application portability decreases
 - Sharing underutilized computing resources becomes important again to reduce infrastructure costs
- Nowadays, virtualization is a core technology for cloud computing

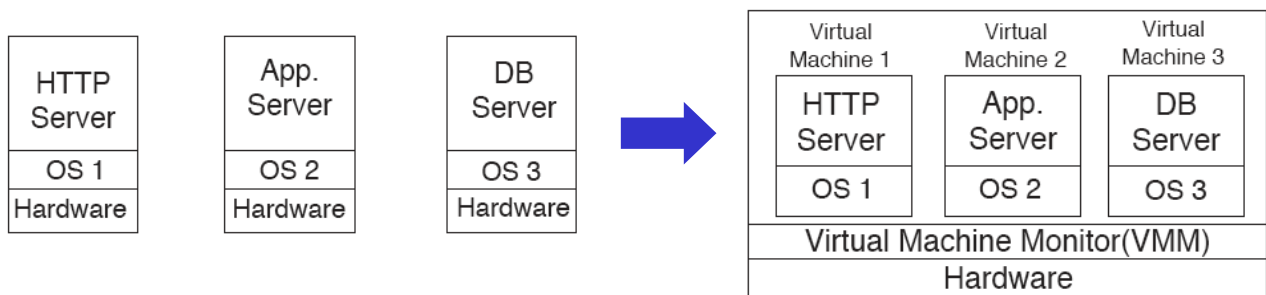
Virtualization: pros

- Facilitate **compatibility**, **portability**, **interoperability** and **migration** of applications and environments
 - Hw independence: create once, run everywhere
 - Legacy VMs: run old OSs or applications on new platforms



Virtualization: pros

- Enable **server consolidation** in data center
 - Goal: reduce number of physical servers and use them more efficiently
 - How? Multiplexing multiple VMs on same physical server
 - ✓ Reduce costs, energy consumption, and occupied space
 - ✓ Simplify server management, maintenance, and upgrade
 - ✓ Reduce service downtime through **live migration** of VMs



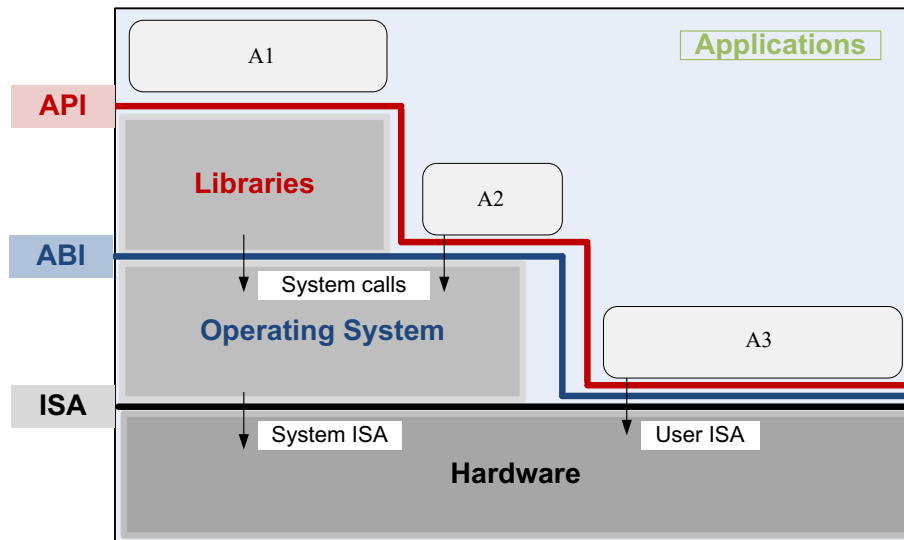
Virtualization: pros

- Enable **isolation of application components** that are malfunctioning or under security attacks, thus increasing application reliability and security
 - VMs running different components cannot access each other's resources
 - Software bugs, crashes, viruses inside VM do not harm other VMs running on same physical machine
- Enable **performance isolation** of different VMs
 - By **scheduling** shared physical resources among different VMs running on the same physical machine
- Enable **load balancing** on physical machines
 - By **migrating** VMs from a physical machine to another

Reasons to use virtualization

- Personal and educational
 - Run multiple OSs simultaneously on single machine
 - Simplify sw installation
 - Develop, test and debug applications
 - Simulate distributed environment on single machine
- Enterprise
 - Consolidate data center infrastructure and ensure **business continuity**
 - Encapsulate entire systems in single files (system images) that can be replicated, migrated or reinstalled on any server
 - Enable **DevOps**

Interfaces in computer system



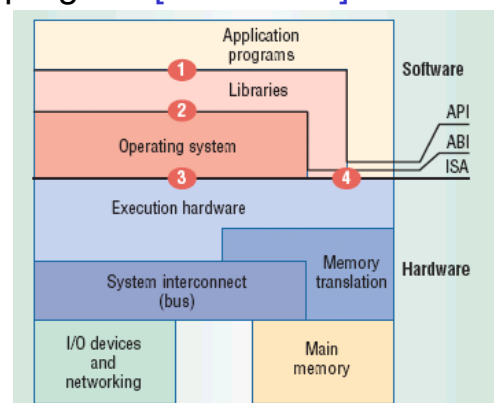
Applications:

- use library functions (A1)
- make system calls (A2)
- execute machine instructions (A3)

Interfaces in computer system and virtualization

At which level can we realize virtualization?

- Strictly related to computer system interfaces
 - **Hw/sw interface (system ISA)**: primarily for system resource management, *privileged* instructions executed only by OS [interface 3]
 - **Hw/sw interface (user-level ISA)**: primarily for computation, *non-privileged* instructions executed by any program [interface 4]
 - **System calls [interface 2]**:
 - **ABI** (Application Binary Interface): interface 2 + interface 4
 - **Library calls (API) [interface 1]**
- Essence of virtualization: **mimic behavior of these interfaces**



Smith and Nair, The architecture of virtual machines, IEEE Computers, 2005

Implementation levels of virtualization

- Virtualization can be implemented at various operational levels:
 - ISA level
 - **Hardware level** (aka *system VMs*)
 - **Operating system level** (aka *containers*)
 - Library level
 - User application level (aka *process VMs*)

← Our focus

Implementation levels of virtualization

- **ISA level**
 - Goal: **emulate a given ISA** by ISA of host machine
 - E.g., MIPS binary code can run on x86-based host with help of ISA emulation
 - ISA emulation can be done through *code interpretation* or *dynamic binary translation*
 - With code interpretation every source instruction is interpreted by emulator in order to execute native ISA instructions
 - Dynamic binary translation is faster: converts in blocks rather than instruction by instruction

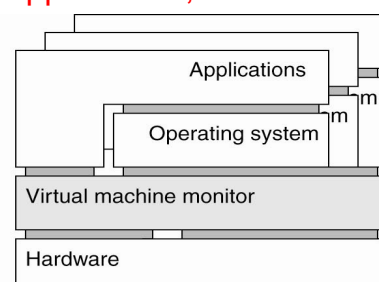
Implementation levels of virtualization

- **Hardware level** (aka *system VMs*)
 - Goal: **virtualize host resources** (processors, memory, and I/O devices)
 - Based on **Virtual Machine Monitor (VMM)**, aka *hypervisor*
 - VMM handles interaction with underlying hw for CPU, memory, and I/O resource access

Implementation levels of virtualization

- **Hardware level** (aka *system VMs*)
 - Provides a complete environment in which **multiple VMs** can coexist
 - **VMM** manages hardware resources and shares them among **multiple VMs** and provide isolation and protection of VMs
 - When a VM performs a **privileged instruction** or operation that directly interacts with shared hw, VMM intercepts the instruction, checks it for correctness, and performs it
 - Examples: VMware, KVM, Xen, Parallels, VirtualBox

Multiple instances of combinations
<applications, OS>



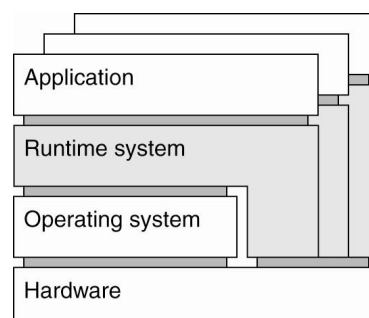
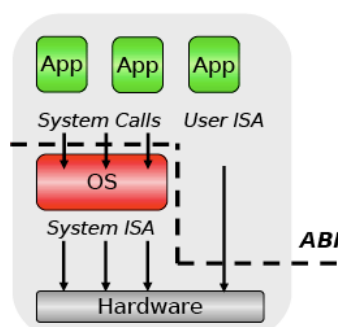
Implementation levels of virtualization

- **Operating system level** (aka *containers*)
 - Goal: create multiple isolated containers
 - Examples: Docker, Linux Containers, Podman
- **Library level**
 - Goal: create execution environment to run apps in a host environment that does not suite native apps
 - Rather than creating a VM to run full OS and apps
 - Examples:
 - Wine <https://www.winehq.org>: runs Windows apps on top of POSIX-compliant OS by translating Windows API calls into POSIX calls on-the-fly
 - Cygwin <https://cygwin.com>: “Get that Linux feeling – on Windows”

Implementation levels of virtualization

- **User application level** (aka *process VMs*)
 - Virtual platform that executes a **single process**
 - Provides **virtual ABI or API** to user application
 - Application is compiled into intermediary, portable code (e.g., Java bytecode) and executed in runtime environment provided by process VM
 - Examples: JVM, .NET CLR

Multiple instances of combinations
<application, runtime system>



Implementation levels of virtualization: summing up

- Relative merits of virtualization at different levels

Level of Virtualization	Functional Description	Example Packages	Merits, App Flexibility/ Isolation, Implementation Complexity		
Instruction Set Architecture	Emulation of a guest ISA by host	Dynamo, Bird, Bochs, Crusoe	Low performance, high app flexibility, median complexity and isolation		
Hardware-Level Virtualization	Virtualization on top of bare-metal hardware	XEN, VMWare, Virtual PC	High performance and complexity, median app flexibility, and good app isolation		
Operating System Level	Isolated containers of user app with isolated resources	Docker Engine, Jail, FVM	Highest performance, low app flexibility and best isolation, and average complexity		
Run-Time Library Level	Creating VM via run-time library through API hooks	Wine, vCUDA, WABI, LxRun	Average performance, low app flexibility and isolation, and low complexity		
User Application Level	Deploy HLL VMs at user app level	JVM, .NET CLR, Panot	Low performance and app flexibility, very high complexity and app isolation		

	Level of Implementation	Higher Performance	Application Flexibility	Implementation Complexity	Application Isolation
Instruction Set Architecture (ISA)	X	XXXXX	XXX	XXX	XXX
Hardware-Level Virtualization	XXXXX	XXX	XXXXX	XXXX	XXXX
Operating System Level	XXXXX	XX	XXX	XX	XX
Run-Time Library Support	XXX	XX	XX	XX	XX
User Application Level	XX	XX	XXXXX	XXXXX	XXXXX

Valeria Cardellini - SDCC 2024/25

20

System-level virtualization: terminology

- Let's first focus on **system-level virtualization** (achieved through VMM or hypervisor)
- Host**: base platform on top of which VMs are executed; made of:
 - Physical machine
 - Possible host OS
 - VMM
- Guest**: everything inside a single VM
 - Guest OS and applications executed inside the VM

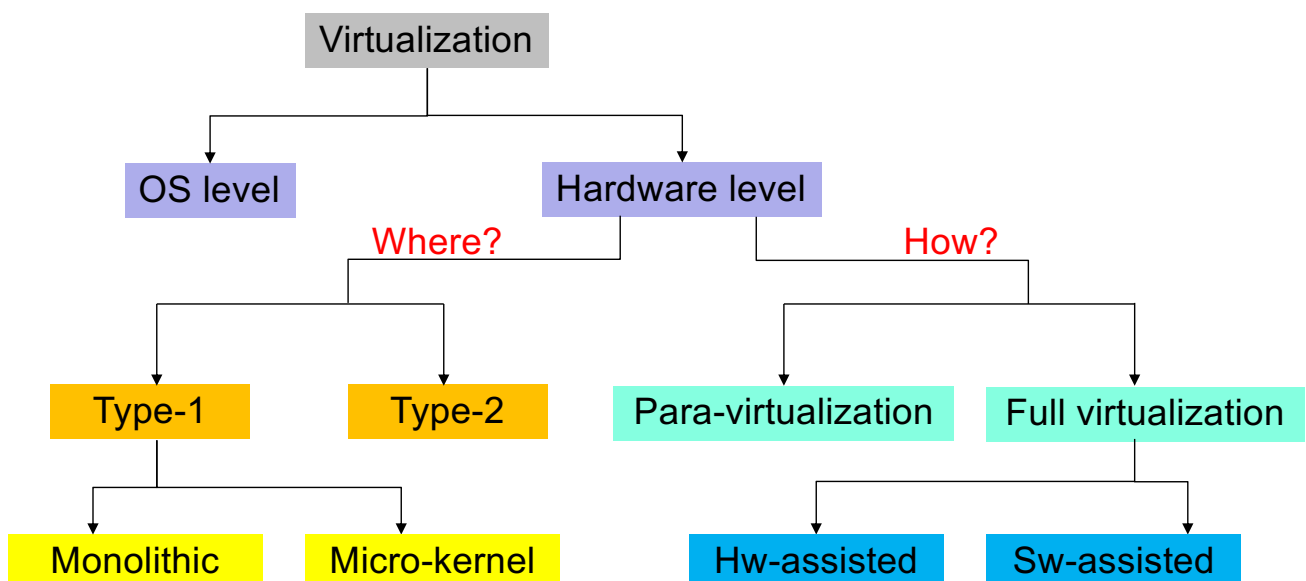
System-level virtualization: taxonomy

- Let's classify system-level virtualization solutions according to:
 - Where** to deploy VMM
 - System VMM** (aka *type-1*, *native* or *bare-metal* hypervisor)
 - Hosted VMM** (aka *type-2* hypervisor)
 - How** to virtualize instruction execution
 - Full virtualization**
 - Software-assisted*
 - Hardware-assisted*
 - Paravirtualization**

Valeria Cardellini - SDCC 2024/25

22

System-level virtualization: taxonomy



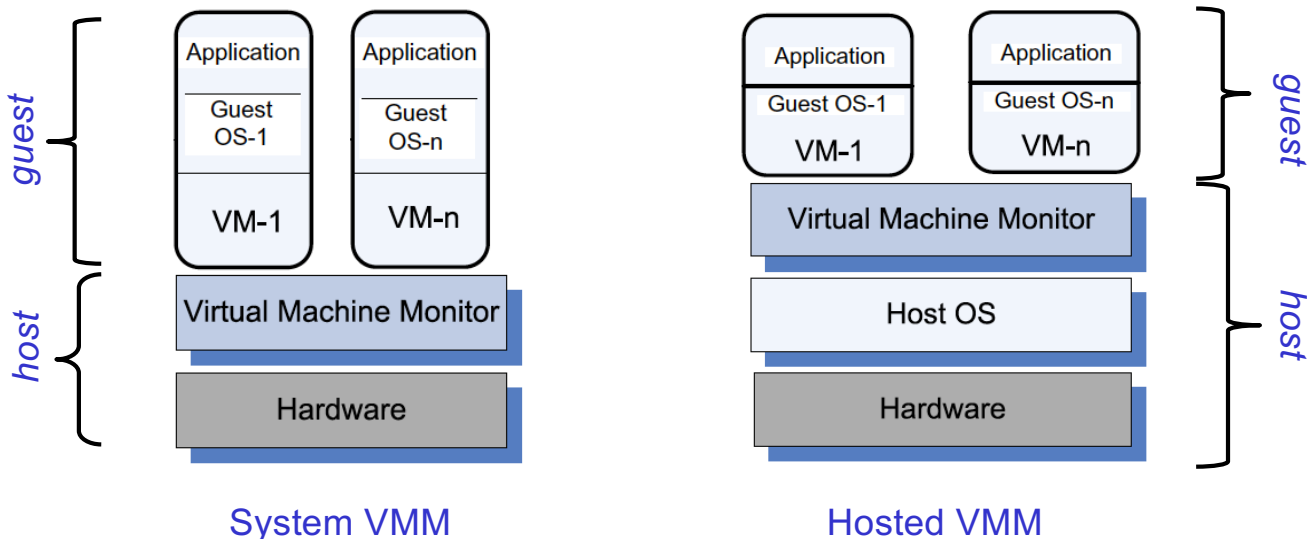
Valeria Cardellini - SDCC 2024/25

23

System vs. hosted VMM

In which level of the system architecture is VMM deployed?

- Directly on hardware: **system (or native) VMM**
- On top of host OS: **hosted VMM**



System vs. hosted VMM

- **System VMM** (*type-1*): runs directly on hw, offers virtualization features integrated into a simplified OS
 - VMM can have *microkernel* (only basic functions, no device drivers) or *monolithic* architecture
 - Examples: KVM, Microsoft Hyper-V, Nutanix AHV, VMware ESXi, Xen
- **Hosted VMM** (*type-2*): runs on top of host OS, accesses hw resources via host OS system calls
 - Interacts with host OS via ABI and emulates virtual hw ISA for guest OS
 - ✓ Can use host OS to manage devices and use low-level services (e.g., resource scheduling)
 - ✓ No need to change guest OS
 - ✗ Performance degradation with respect to system VMM
 - Examples: Parallels Desktop, VirtualBox

Full virtualization vs paravirtualization

How to manage interaction between VMs and VMM in order to access to system resources, i.e., how to manage the execution of privileged instructions that require direct access to hardware or other privileged resources?

- **Full virtualization**
- **Paravirtualization**

Comparison of platform virtualization software

https://en.wikipedia.org/wiki/Comparison_of_platform_virtualization_software

Full virtualization vs paravirtualization

- Full virtualization
 - VMM exposes to each VM simulated hw interfaces that are *functionally identical* to those of the underlying physical machine
 - VMM intercepts attempts to perform privileged instructions (e.g., I/O, TLB update) and emulates the expected behavior
 - Examples: KVM, VMware ESXi, Microsoft Hyper-V
- Paravirtualization
 - The VMM exposes to each VM simulated hw interfaces that are *functionally similar* (but not identical) to those of the underlying physical machine
 - Hardware is not emulated, but a minimal software layer (**Virtual Hardware API**) is created to ensure VM management and their isolation
 - Examples: Xen, Oracle VM, PikeOS

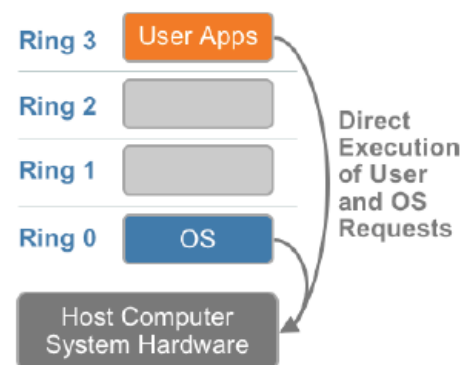
Full virtualization vs paravirtualization

- Full virtualization pros and cons
 - ✓ Run unmodified guest OSs
 - ✓ Complete isolation between VM instances: security, ease of emulating different architectures
 - ✗ VMM is more complex
 - ✗ Require processor collaboration to make virtualization more efficient: **why?**

Issues to address for system-level virtualization

- Non-virtualized processor architecture operates according to at least 2 **protection levels (rings)**: supervisor and user
 - **Ring 0**: most privileged (unrestricted access to system resources)
 - **Ring 3**: least privileged
- With virtualization
 - **VMM** operates in **supervisor** mode (ring 0)
 - **Guest OS** and applications (i.e., VM) operate in **user** mode (guest OS in ring 1 or 3)
 - **Ring deprivileging** problem: guest OS operates in a ring which is not its own \Rightarrow cannot execute privileged instructions (e.g., `lidt` in x86 to load interrupt descriptor table)
 - **Ring compression** problem: since applications and guest OS run at the same level, guest OS space must remain protected

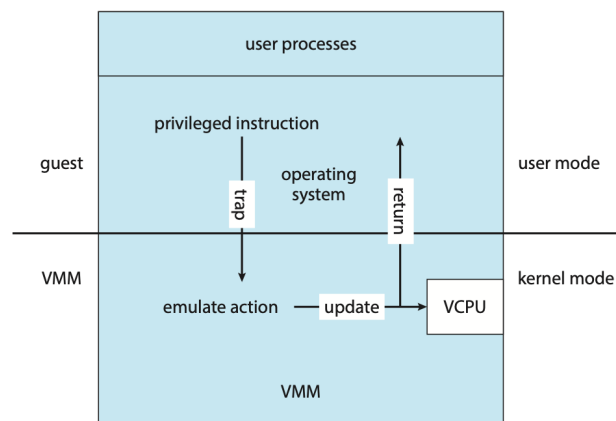
x86 architecture w/o virtualization



How to address ring deprivileging

- **Trap-and-emulate**

- When guest OS attempts to execute **privileged instructions** (which can run only in kernel mode), an exception (**trap**) must be notified to VMM and control must be transferred to it; VMM performs a safety check on the requested operation, executes (“**emulates**”) its behavior and returns result to guest OS
- Instead **non-privileged instructions** (all of them?) run by guest OS do not trap and are **directly executed**



Valeria Cardellini - SDCC 2024/25

30

Popek and Goldberg virtualization requirements

- Popek and Goldberg (1974): conditions and theorems under which a computer architecture can **support efficient virtualization**

Formal requirements for virtualizable third generation architectures,

<https://dl.acm.org/doi/pdf/10.1145/361011.361073>

- Conditions:

Equivalence

Resource control

Efficiency

1. A program running under the hypervisor should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
2. The hypervisor should be in complete control of the virtualized resources.
3. A statistically significant fraction of machine instructions must be executed without the intervention of the hypervisor.

Popek and Goldberg virtualization requirements

- ISA instructions are classified into 3 groups:
 - Privileged instructions**: must be executed in supervisor mode, **trap if executed in user mode**
 - Sensitive instructions**: change underlying resources (e.g., page tables) or observe information that indicates current privilege level (thus exposing that guest OS does not run on bare metal);
 - Control sensitive**: change the CPU configuration / CPU state (e.g., interrupt table, paging table)
 - Behavior sensitive**: reveal something about the CPU state
 - Innocuous instructions**: not sensitive
- Theorem: *For any conventional computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions*

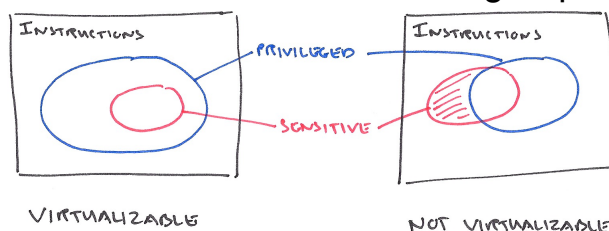
<https://blog.acolyer.org/2016/02/19/formal-requirements-for-virtualizable-third-generation-architectures>

Valeria Cardellini - SDCC 2024/25

32

Condition for virtualization

- The bad news: theorem condition is **not satisfied by all architectures**
 - There might be **sensitive but non-privileged** instructions that are executed in user mode without causing trap to OS



- Some CPU architectures are **non-virtualizable**
 - x86: plenty of sensitive non-privileged instructions
 - E.g., pushf (push the flags register on the stack)
 - MIPS: mostly virtualizable, but
 - \$k0, \$k1 (registers to save/restore state) are user-accessible
 - ARM: mostly virtualizable, but
 - Some instructions are undefined in user-mode

Condition for virtualization

- From Popek and Goldberg's theorem:
 - Need to virtualize both privileged instructions and sensitive but non-privileged instructions that are executed in user mode
- Issue:
 - Privileged instructions cause trap: ok
 - Non-privileged sensitive instructions do not cause trap
 - How to virtualize these instructions?
- 1st solution: **trap-and-emulate**
 - Privileged and non-privileged sensitive instructions cause trap and divert control to VMM
 - Seems easy but ... **how to implement it?**
- 2nd solution: **paravirtualization**
 - Modify guest OS, by either preventing non-privileged sensitive instructions or making them non-sensitive (i.e., changing the context)

Full virtualization: solutions

- How to realize trap mechanism?
- At **hardware level** if processor supports virtualization
 - Idea: introduce a new privilege level
 - Hypervisor privilege level, more privileged than kernel privilege level
 - All sensitive instruction trap to hypervisor level

hardware-assisted CPU virtualization

- At **software level** if processor does not support virtualization

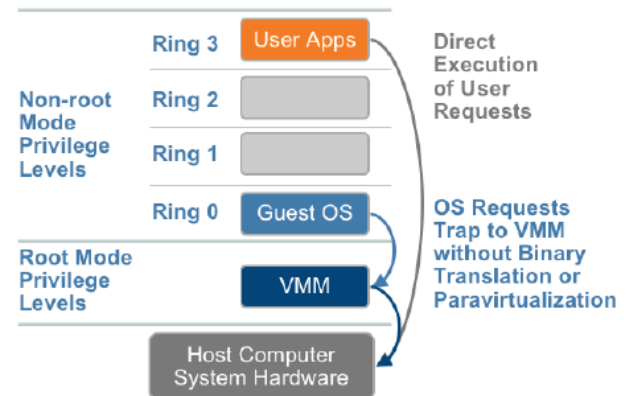
fast binary translation

- The elder solution

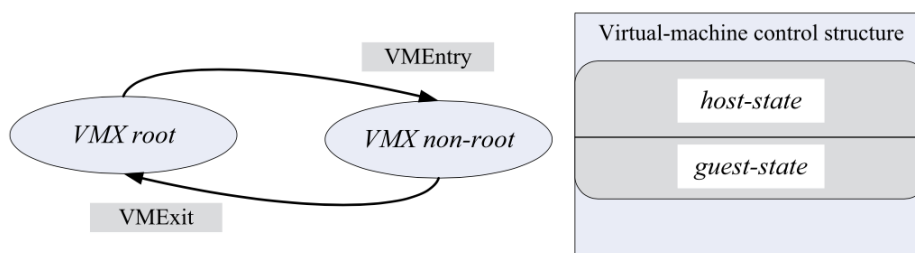
Hardware-assisted CPU virtualization

- Hardware-assisted CPU virtualization (Intel VT-x and AMD-V) provides two new CPU operating modes (**root mode** and **non-root mode**), each supporting all 4 x86 protection rings
 - VMM runs in root mode (Root-Ring 0), while guest OSs run in guest mode in their original privilege levels (Non-Root Ring 0): no longer ring deprivileging and ring compression issues
 - VMM can control guest execution through VM control data structures in memory

x86 architecture with full virtualization and **hardware-assisted CPU virtualization**



Hardware-assisted CPU virtualization: VT-x



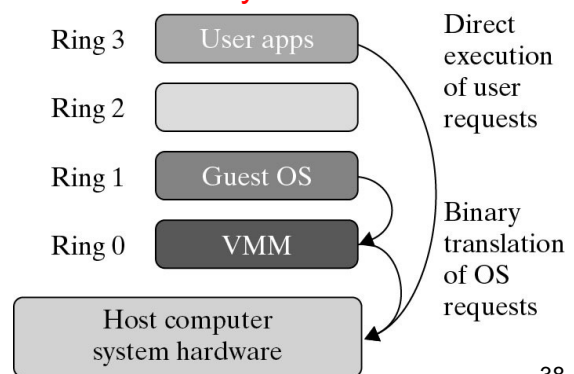
- **VMX root**: intended for hypervisor operations (like x86 without VT-x)
- **VMX non-root**: intended to support VMs
- When executing **VMEntry** operation, processor state is loaded from *guest-state* of VM scheduled to run, then control is transferred from hypervisor to VM
- **VMExit** saves processor state in *guest-state* area of running VM; it loads processor state from *host-state*, then transfers control to hypervisor

Fast binary translation

- VMM trap mechanism for privileged instructions is offered by processors with hardware support for virtualization
 - How to achieve full virtualization without hw support?
- **Fast binary translation**: VMM scans code before its execution to replace blocks containing privileged instructions with functionally equivalent blocks containing instructions for notifying exception to VMM

- Translated blocks are directly executed on hw and stored in cache for future reuse
- ✗ Higher complexity and lower performance wrt to hw-assisted virtualization

x86 architecture with full virtualization and fast binary translation

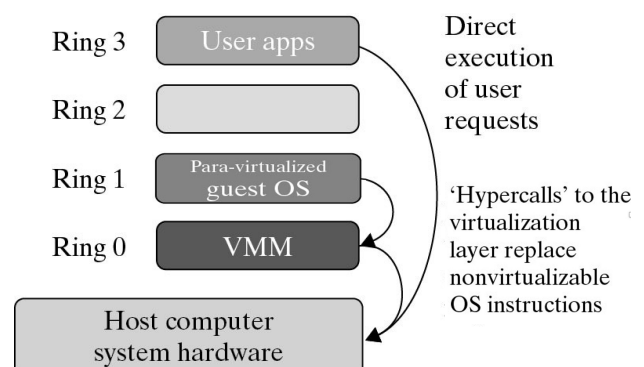


Paravirtualization

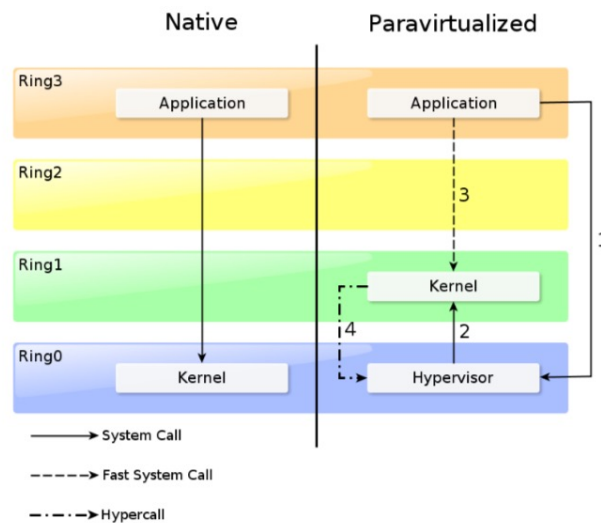
- **Non-transparent** virtualization solution
 - Guest OS kernel must be modified to let it invoke the virtual API exposed by VMM
- Non-virtualizable instructions are replaced by **hypercalls** that communicate directly with hypervisor
 - Hypercall: **software trap** from guest OS to hypervisor, just as syscall is software trap from app to kernel

hypercall : hypervisor = syscall : kernel

x86 architecture with paravirtualization



Paravirtualization: hypercall execution



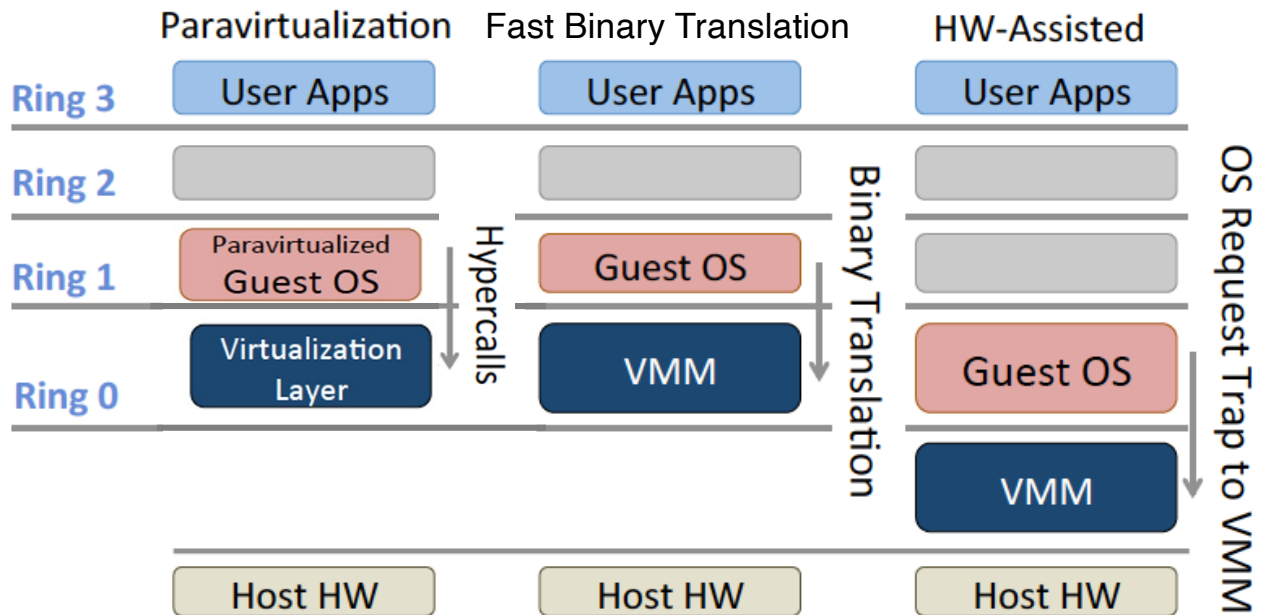
- When application running in VM issues a guest OS system call, through the hypercall the control flow jumps to hypervisor, which then passes control back to guest OS

Source: "The Definitive Guide to XEN hypervisor"

Paravirtualization: pros & cons

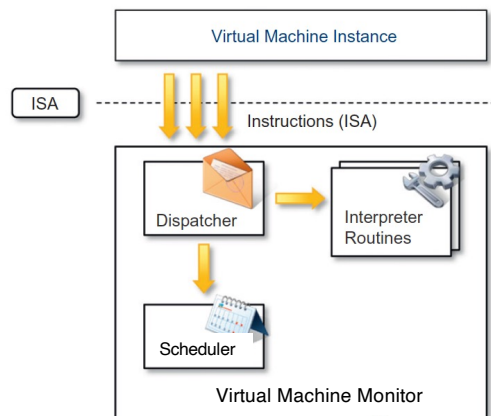
- Pros (vs full virtualization):
 - ✓ Relatively easier and more practical implementation
 - ✓ Less overhead wrt fast binary translation
 - ✓ Does not require virtualization extensions from host CPU as hw-assisted virtualization does
- Cons (vs full virtualization):
 - ✗ Requires source code availability of OS to modify guest OS and make it paravirtualized
 - ✗ Cost of maintaining paravirtualized OSs
 - Paravirtualized OS cannot run directly on hardware

Summing up different approaches



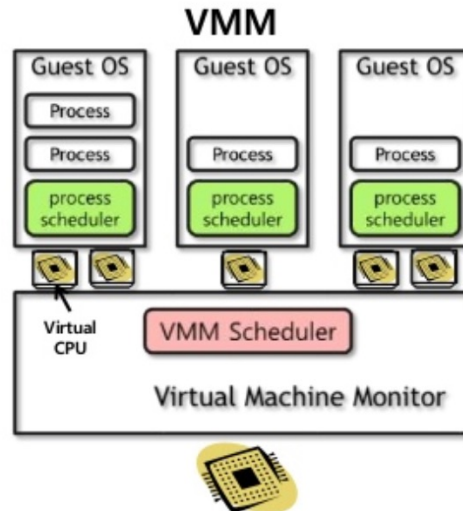
VMM reference architecture

- 3 main modules
 - **Dispatcher**: VMM entry point that reroutes privileged instructions issued by VMs to the other modules
 - **Scheduler**: invoked by dispatcher whenever VM executes privileged instruction for acquiring system resources; decides about their allocation to VMs
 - **Interpreter**: executes proper routine when VM executes a privileged instruction



VMM CPU scheduler

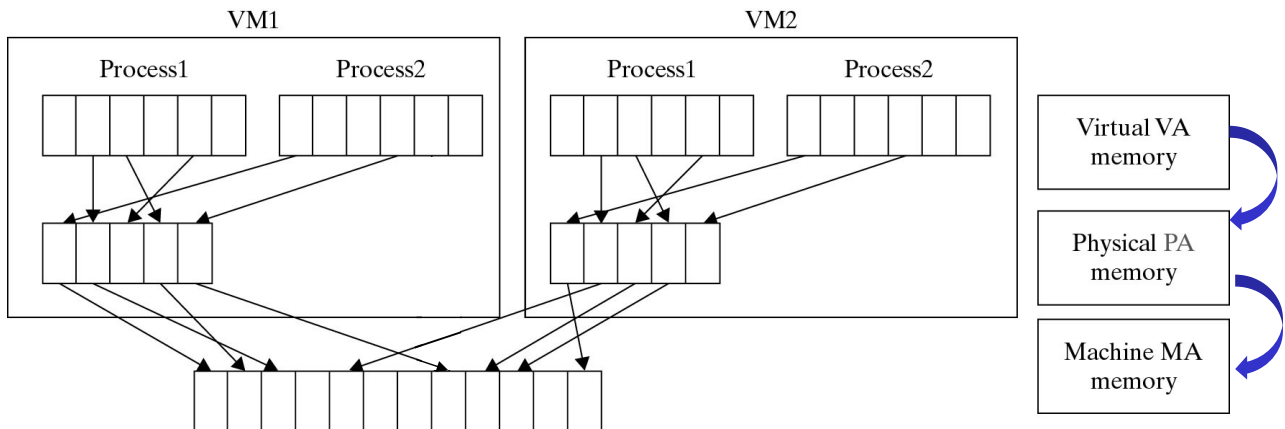
- Guest OS sees virtual CPUs
- Physical CPUs on host machine are multiplexed among VMs
- How to schedule virtual CPUs on physical CPUs?



Memory virtualization

- In *non-virtualized* environment
 - One-level memory mapping: from virtual memory to physical memory provided by page tables
 - MMU and TLB hardware components to optimize virtual memory performance
- In *virtualized* environment
 - All VMs share machine memory and VMM needs to partition it among VMs
 - **Two-level memory mapping**: from guest virtual memory to guest physical memory to host physical memory
- Some terminology
 - *Guest virtual memory*: memory visible to apps; continuous virtual address space presented by guest OS to apps
 - *Guest physical memory*: memory visible to guest OS
 - *Host physical memory*: actual hw memory visible to VMM

Two-level memory mapping



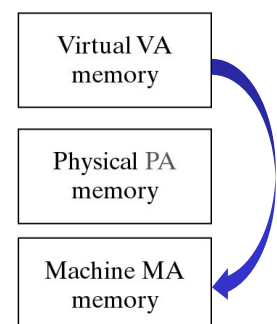
- Going from guest virtual memory to host physical memory requires two-level memory mapping

GVA (guest virtual address) → GPA (guest physical address) → HMA (host machine address)

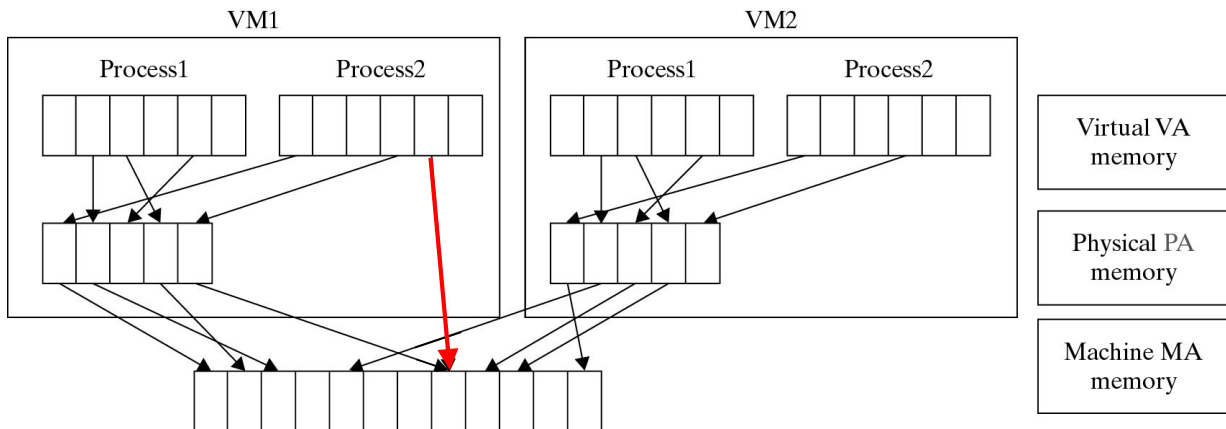
- Guest physical address \neq host machine address: why?
 - Hints: many VMs; what does guest OS expect about its memory?

Shadow page tables

- How to avoid unbearable performance drop due to extra memory mapping?
- VMM maintains **shadow page tables (SPTs)** and uses them to accelerate address mapping
 - So to achieve direct mapping from GVA to HPA
- SPT directly maps GVA to HPA
 - Guest OS creates and manages page tables (PTs) for its virtual address space without modification
 - But these PTs are not used by MMU hardware
 - VMM creates and manages PTs that map virtual pages directly to machine pages
 - These VMM PTs are the **shadow page tables** and are loaded into MMU
 - VMM needs to keep SPTs consistent with changes made by each guest OS to its PTs



Memory mapping with SPTs



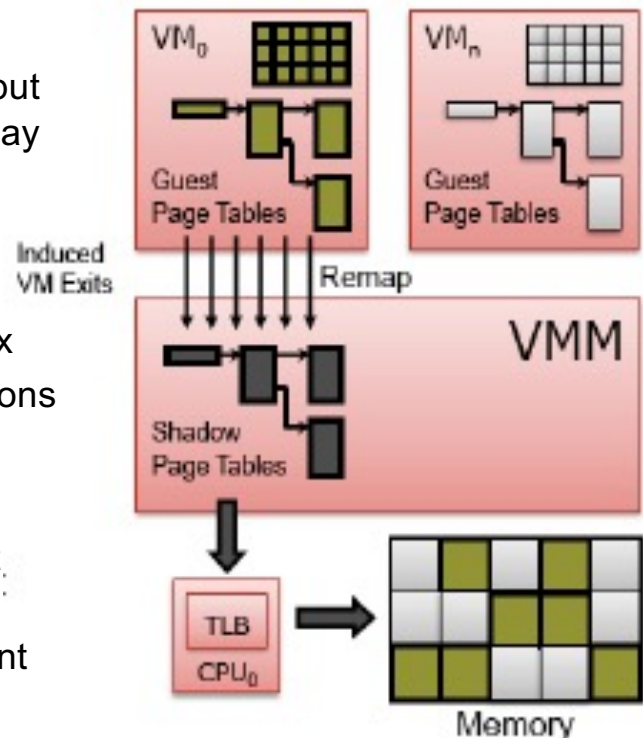
- VMM uses TLB hardware to map virtual memory directly to machine memory to avoid the two levels of translation on every access (red arrow)

SPT consistency

- When guest OS changes its PTs, VMM needs to update SPTs to enable a direct lookup
- How?
 - VMM maps guest OS PTs as read only
 - When guest OS writes to PTs, trap to VMM
 - VMM applies write to SPT and guest OS PT, then returns
 - Aka [memory tracing](#)
 - Adds overhead

Challenges in memory virtualization with SPT

- Address translation
 - Guest OS expects contiguous, zero-based physical memory, but underlying machine memory may be non contiguous: VMM must preserve this illusion
- Page table shadowing
 - SPT implementation is complex
 - VMM intercepts paging operations and constructs copy of PTs
- Overheads
 - SPTs consume significant host memory
 - SPTs need to be kept consistent with guest PTs
 - VM exits add to execution time

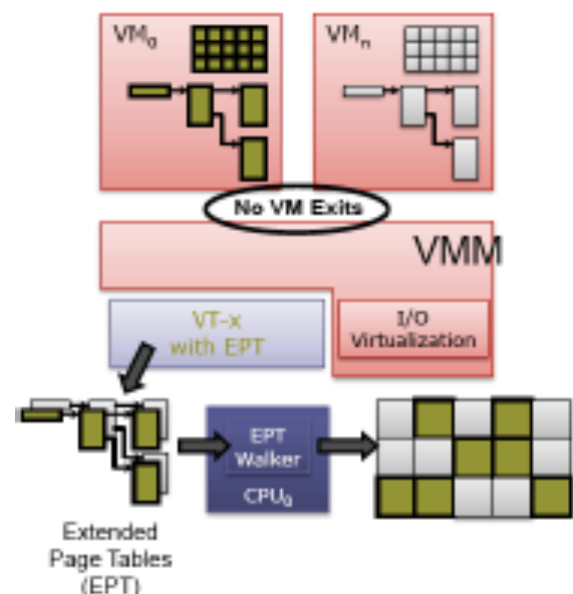


Valeria Cardellini - SDCC 2024/25

50

Hw support for memory virtualization

- SPT is a software-managed solution: let's consider a more efficient hardware solution
- **Second Level Address Translation (SLAT)** is the hardware-assisted solution for memory virtualization (Intel EPT and AMD RVI) to translate GVA into HPA
- Using SLAT significant performance gain with respect to SPT: around 50% for MMU intensive benchmarks



Valeria Cardellini - SDCC 2024/25

51

- Most notable example of **paravirtualization**
<https://www.xenproject.org/> born at Cambridge Univ.
 - Open-source **type-1** (system VMM) hypervisor with **microkernel** design
 - Offers to guest OS a virtual interface (**hypercall API**) to whom guest OS refers to access physical resources
 - Supports paravirtualization (**PV**), hardware-assisted virtualization (**HVM**) and their combination
 - With paravirtualization Xen requires PV-enabled guest OSs and PV drivers (part of Linux kernel and other OSs)
 - OSs ported to Xen: Linux, NetBSD, FreeBSD
 - Foundation for commercial virtualization products (e.g., XenServer, Oracle VM)
 - Xen distros for embedded systems (e.g., ARM, RISC-V)
 - Powers IaaS providers (e.g., Alibaba, Amazon, Rackspace)

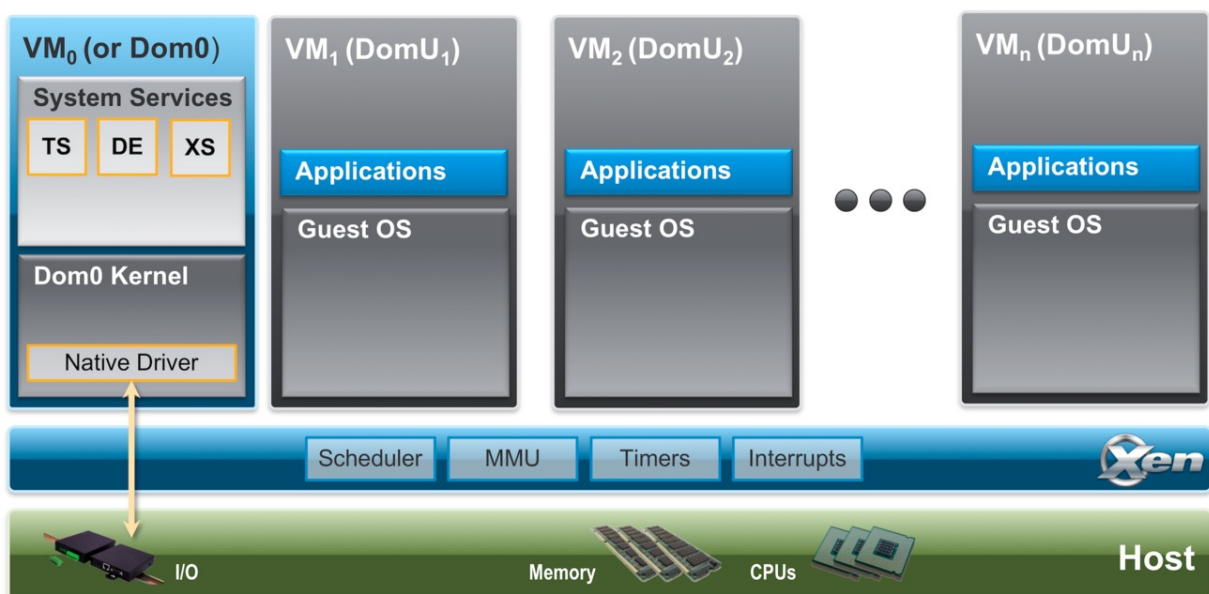
Xen: features

- Thin hypervisor model
 - 300K lines of code on x86, 65K on Arm
 - Small footprint and interface (around 1MB in size)
 - Microkernel design
 - Scalable: up to 4,095 host CPUs with 16Tb of RAM
 - More robust and secure than other hypervisors
 - But still vulnerable to attacks <https://xenbits.xen.org/xsa/>
- Continuously improved
- Flexibility in management
 - Tuning for performance
- Supports VM live migration
- Low overhead (within 2%) with respect to bare metal machine without virtualization

Xen: the origins

- Goal of Cambridge Univ. group who designed Xen (late 1990s, released in 2003):
 - Design hypervisor capable of scaling to ~100 VMs apps
 - At that time, only fast binary translation
 - Idea: “What if instead of trying to fool the guest kernel into thinking it’s running on real hardware, you just let the guest know that it was running in a virtual machine, and changed the interface you provide to make it easier to implement?”
- What can be paravirtualized?
 - Privileged instructions: replaced by hypercalls
 - Page tables (memory access)
 - Disk access and I/O devices
 - Interrupts and timers

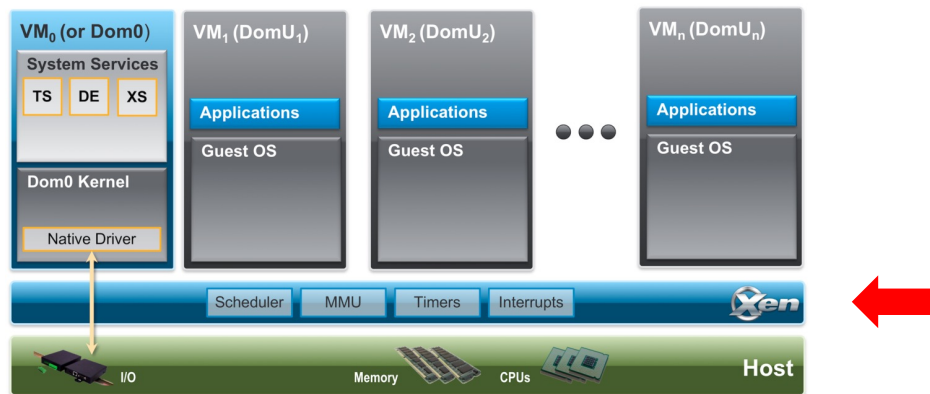
Xen: architecture



https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview

Xen architecture: hypervisor

- In charge of CPU scheduling, memory management, interrupt and timer handling
- Per-domain and per-vCPU info management



Xen architecture: domains

- 2 kinds of domains: **control domain** which starts and manages all the others **unprivileged domains**
- Guest domains: **DomU (unprivileged)**
 - Represent VM instances, each running a guest OS with apps
 - Run on virtual CPUs (vCPUs)
- **Dom0 (control domain)**: VM with special privileges
 - Handles device driver stack that provides access to physical devices to the other domains, so it responsible for multiplexing guest I/O
 - Provides management functions (e.g., create other domains)
 - Contains **system services**: Device Emulation (DS), XenStore/XenBus (XS), and Toolstack (TS)
 - Mandatory, initial domain started by Xen on boot

Dom0 components: XenStore and Toolstack

- **XenStore**: information storage space shared among domains
 - Stores configuration and status information
 - Implemented as hierarchical key-value storage
 - When values are changed in XS, a watch function notifies listeners (e.g., drivers) of changes of the key they have subscribed to
 - Communicates with guest VMs via shared memory using Dom0 privileges
- **Toolstack**: allows user to manage VM lifecycle (create, shutdown, pause, migrate) and configuration
 - To create a new VM, user provides a configuration file describing memory and CPU allocation and device configurations
 - Toolstack parses this file and writes this information in XenStore
 - Takes advantage of Dom0 privileges to map guest memory, to load kernel and virtual BIOS and to set up initial communication channels with XenStore and with virtual console when a new VM is created

CPU scheduler in Xen

- VMM scheduler decides, among all **virtual CPUs (vCPUs)** of VMs, which ones will execute on **physical CPUs (pCPUs)**
 - Further scheduling level with respect to OS ones (processes and user-level threads within processes)
- Xen allows to choose among different CPU schedulers
 - **Credit** scheduler is the default
- Scheduling algorithm goals:
 - Make sure that domains get **fair** share of pCPU
 - **Proportional share** algorithm: allocates pCPU in proportion to number of shares (weights) assigned to vCPUs
 - Keep pCPU busy
 - **Work-conserving** algorithm: does not allow pCPU to be idle when there is work to be done
 - Schedule with low latency

Credit scheduler

- Proportional fair share and work-conserving scheduler
- Each domain (including Domain0) is assigned a *weight* and a *cap* (tunable parameters)
 - Weight: relative pCPU allocation per domain (default 256)
 - Cap: maximum amount of CPU a domain can use
 - cap = 0 (default): vCPU can receive any extra CPU (i.e., work-conserving)
 - cap ≠ 0: limits amount of pCPU that vCPU receives (i.e., non work-conserving); expressed as % of pCPU (e.g., 100 = 1 pCPU, 50 = 0.5 pCPU)
- The scheduler transforms the weight into a *credit* allocation for each vCPU
 - The credit value represents the pCPU share that the domain is expected to have
 - As a vCPU runs, it consumes credit
 - If its credit value is negative, the domain is in *OVER* priority; otherwise, in *UNDER* priority

Valeria Cardellini - SDCC 2024/25

60

Credit scheduler: algorithm

- For each pCPU, the scheduler maintains a queue of vCPUs, with all the vCPUs in UNDER priority first, followed by vCPUs in OVER priority
 - Round-robin ordering within UNDER and OVER priorities
 - Scheduler picks the vCPU at the head of the queue
 - Selected vCPU receives 30 ms time slice before being preempted to run another vCPU
 - VCPUs in OVER priority cannot be scheduled unless there is no UNDER VCPUs in the queue
- The scheduler load balances vCPUs across pCPUs on SMP (symmetric multi-processor) host
 - Before a pCPU goes idle, it considers other pCPUs in order to find any UNDER credit vCPU: no pCPU is idle when there is runnable work in the system

https://wiki.xenproject.org/wiki/Credit_Scheduler

Xen and paravirtualization

- In most deployments, Xen no longer uses paravirtualization to virtualize CPU (as in original implementation)
 - HVM guests exploit hardware extensions to implement hardware-assisted virtualization
 - PV guests are still used on hardware not supporting virtualization extensions
- However, paravirtualization is important for I/O: instead of trying to (inefficiently) emulate hardware I/O devices, we can define a new virtual-only device and provide a driver for it
 - Xen supports hybrid variants (mix of PV and HVM) including PVH guests

https://wiki.xenproject.org/wiki/Understanding_the_Virtualization_Spectrum

VM portability

- **VM image**: a single file for each VM which contains a bootable OS, data files, and applications
- Virtual machine images come in different formats
- How to import and export VM images and avoid vendor lock-in?
- **Open Virtualization Format (OVF)**
 - Open industry standard (ISO 17203) for packaging and distributing VMs
 - Virtual-platform agnostic
 - Image stored in **.ova** file (Open Virtual Appliance)
 - VM configuration specified in XML format within a **.ovx** file
 - Supported by many hypervisors, including Hyper-V, VMware, VirtualBox, XenServer

VM resizing and migration

- Useful techniques to deploy and manage large-scale virtualized environments
 - **Dynamic resizing** for **vertical scaling** (scale up, scale down) of VMs
 - **Live migration** of VMs
 - Move VM between different physical machines (or even data centers) without stopping it

VM dynamic resizing

- Fine-grain mechanism with respect to migrating or rebooting VMs
 - Example: app running on a VM consumes a lot of resources, thus VM starts running out of RAM and CPU
 - Solution: **dynamically resize VM** (aka *warm* resizing)
- ✓ More cost-effective and faster than VM reboot
- ✗ Not supported by all virtualization products and guest OSs
- What can be resized **without stopping and rebooting** the VM?
 - **Number of virtual CPUs**
 - **Memory**

VM dynamic resizing: CPU

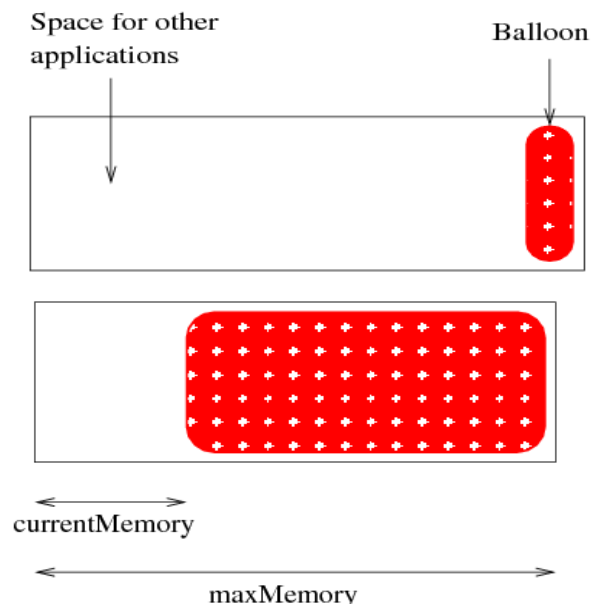
- Add or remove virtual CPUs (without turning off VM)
- Linux supports **CPU hot-plug/hot-unplug**
https://www.kernel.org/doc/html/latest/core-api/cpu_hotplug.html
 - Uses information in virtual file system `sysfs` (processor info is in `/sys/devices/system/cpu`)
 - `/sys/devices/system/cpu/cpuX` for `cpuX` ($X = 0, 1, 2, \dots$)
 - To turn on `cpu #5`:
`echo 1 > /sys/devices/system/cpu/cpu5/online`
 - To turn off `cpu #5`:
`echo 0 > /sys/devices/system/cpu/cpu5/online`
- VM CPU resizing can be managed using **virsh**
 - `virsh`: command line tool to configure and manage virtual machines, available with some hypervisors (KVM, Xen)
 - E.g., set the number of vCPUs while VM is running (cannot exceed max. number of vCPUs)
`virsh setvcpus <vm_name> <vcpu_count> --current`

Valeria Cardellini - SDCC 2024/25

66

VM dynamic resizing: memory

- Based on **memory ballooning**
 - Mechanism used by hypervisors (e.g., KVM, Xen and VMware) to pass memory back and forth between hypervisor and guest OSs
 - In KVM: `virtio_balloon` driver
- When balloon deflates: more memory to VM
 - Anyway, VM memory size cannot exceed `maxMemory`
- When balloon inflates: less memory to VM
 - Swap memory pages to disk



Valeria Cardellini - SDCC 2024/25

67

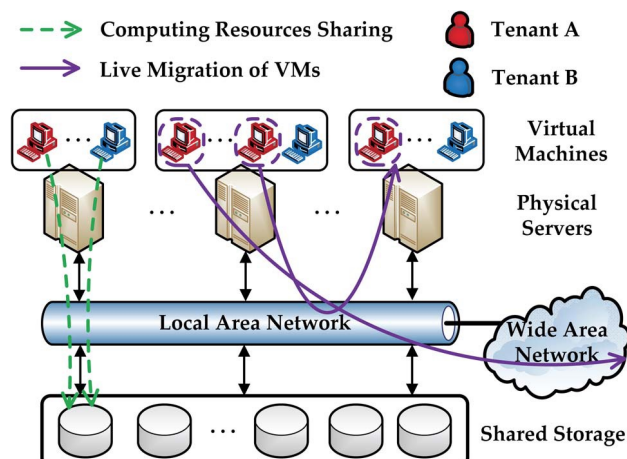
VM migration

- Pros
 - Useful in clusters and virtual data centers to:
 - ✓ Consolidate infrastructure
 - ✓ Add failover flexibility
 - ✓ Balance load
- Cons
 - ✓ Requires VMM support
 - ✓ Migration overhead is non-negligible
 - ✓ WAN migration is scarcely supported

VM migration

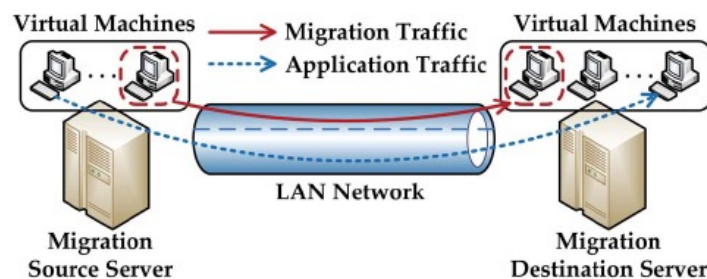
- Approaches to migrate VM instances between physical machines:
 - **Stop and copy**: shutdown source VM and transfer VM image to destination host, but downtime can be too long
 - VM image can be large and network bandwidth limited
 - **Live migration**: source VM is running during migration
 - Largely used by Google: > 1M migrations per month

Our focus



VM live migration

- Preliminary steps before starting VM live migration
 - **Setup** phase: determine source host, destination host and VM to migrate (goals of load balancing, energy efficiency, server consolidation) by means of proper algorithms
- What to migrate? **Memory, storage, network connections**
- How? In a **transparent** way wrt applications running inside VM
 - But migration transparency is hard to achieve, live migration still causes application **downtime**: how to limit it?



Valeria Cardellini - SDCC 2024/25

70

VM live migration: storage

- Let's consider **VM migration within a cluster environment**
- To **migrate storage**
 - Can use network-accessible and shared storage system
 - SAN (Storage Area Network) or cheaper NAS (Network Attached Server) or distributed file system (e.g., HDFS, GlusterFS, CEPH)
 - Without shared storage: source VMM stores all source VM data in an image file, which is transferred to destination host

Valeria Cardellini - SDCC 2024/25

71

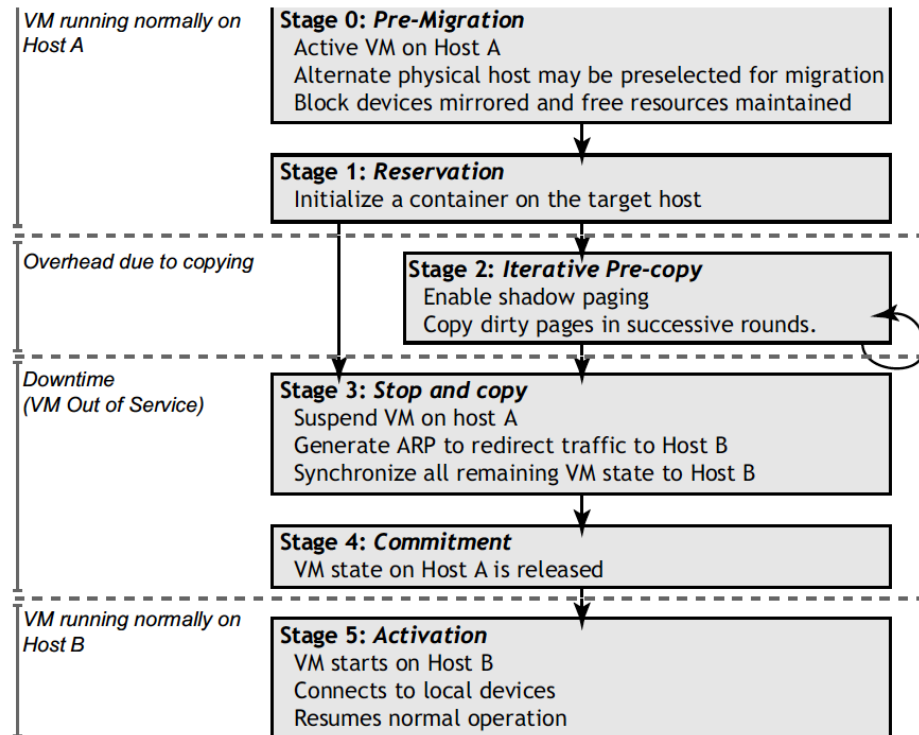
VM live migration: network

- To **migrate network connections**
 - Source VM has its own virtual IP address, which can be distinct from IP address of source host; can also have its own distinct virtual MAC address
 - VMM maintains a mapping of virtual IP and MAC addresses to their corresponding VMs
 - If source and destination hosts are connected to a single switched LAN, an **unsolicited ARP reply** from source host is provided, advertising that the IP has moved to a new location
 - A few in-flight packets might be lost
 - Alternatively, use forwarding mechanisms on source host

VM live migration: memory

- To **migrate memory** (including CPU and device state):
 1. **Pre-copy** phase: VMM copies in an **iterative** way the memory pages from source VM to destination VM **while** source VM is running
 - During iteration n those pages **dirty** during iteration $n-1$ are copied
 2. **Stop-and-copy** phase: source VM **is suspended** and the last dirty pages are copied, as well as CPU and device drivers states; VM applications do not run
 - **Downtime**: from some msec to sec, depending on memory size, application memory workload and network bandwidth
 3. **Commitment** and **reactivation** phases: destination VM is activated and recovers application execution; source VM is removed (and source host may be turned off)
- Known as **pre-copy** approach
 - Memory is copied **before** VM execution resumes at destination
 - Most popular solution (e.g, KVM, VMWare, Xen, Google CE)

VM live migration: overall process



Clark et al., Live Migration of Virtual Machines, NSDI 2005

https://www.usenix.org/legacy/event/nsdi05/tech/full_papers/clark/clark_html/

Valeria Cardellini - SDCC 2024/25

74

VM live migration: alternatives for memory

- Pre-copy cannot migrate in a transparent manner memory-intensive apps
 - E.g., for write-intensive memory app, pre-copy is unable to transfer memory faster than memory is dirtied by running app
- Two alternative approaches
 - Post-copy
 - Hybrid
- **Post-copy**
 - CPU and device state are transferred immediately to destination host followed by transfer of execution control to destination host
 - Memory is fetched on-demand if needed by the running VM on the destination host (*pull* approach)
 - ✓ Reduces downtime and total migration time
 - ✗ Incurs app degradation due to page faults which must be resolved over the network

Valeria Cardellini - SDCC 2024/25

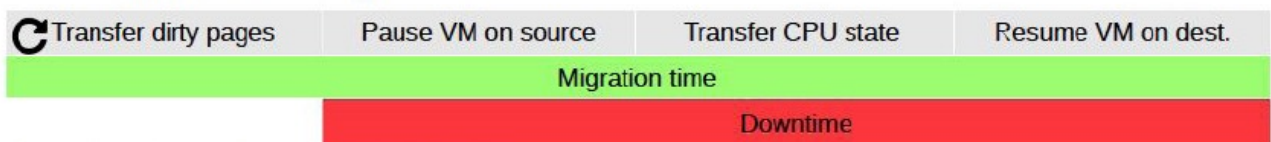
75

VM live migration: alternatives for memory

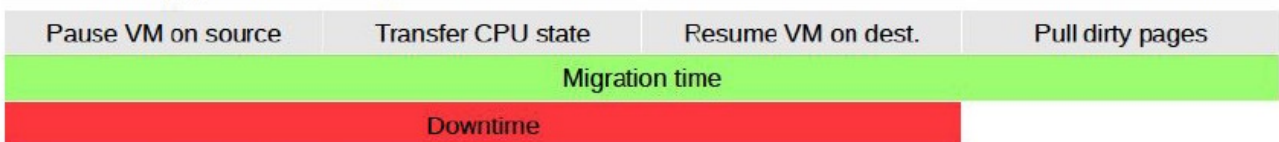
- **Hybrid**
 - Special case of post-copy migration: post-copy preceded by a bounded pre-copy stage
 - Idea: transfer a subset of the most frequently accessed memory pages before VM execution is switched to destination, so to reduce app performance degradation due to memory transfer after VM is resumed
 - ✓ Pre-copy stage reduces the number of future network-bound page faults as a large portion of VM memory is already pre-copied
- Most virtualization products support only pre-copy

VM live migration: alternatives for memory

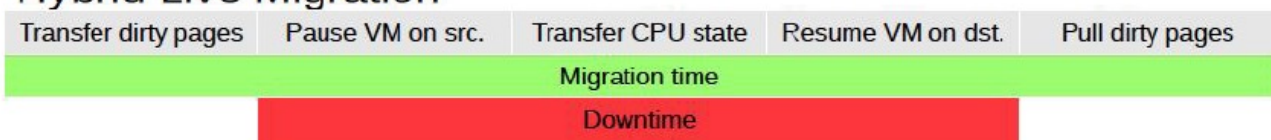
- Summary of approaches to migrate memory
- Pre-copy Live Migration



- Post-copy Live Migration



- Hybrid Live Migration



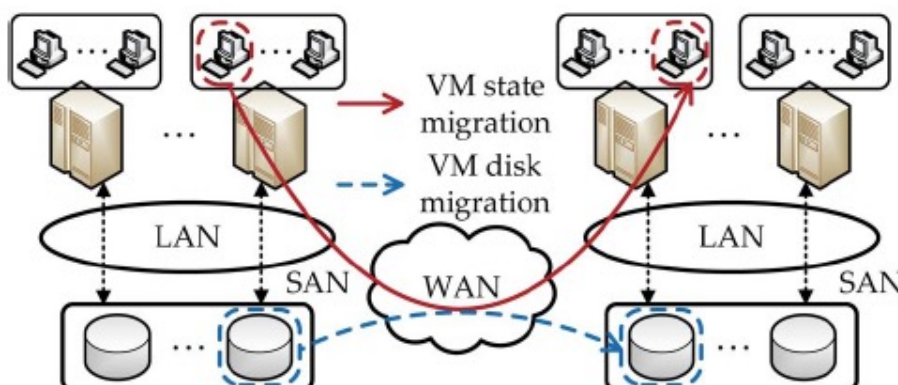
VM live migration and hypervisors

- VM live migration is supported by open-source and commercial hypervisors
 - E.g., KVM, Hyper-V, Xen, VirtualBox
- Can be managed using **virsh** with different options

```
$> virsh migrate --live [--undefinesource] \
[--copy-storage-all] [--copy-storage-inc] domain desturi
$> virsh migrate-setmaxdowntime domain downtime
$> virsh migrate-setspeed domain bandwidth
$> virsh migrate-getspeed domain
```

VM migration in WAN

- How to achieve **VM live migration across multiple geo-distributed data centers?**
 - Key challenge: maintain network connectivity and preserve open connections during and after migration
 - Limited support in open-source and commercial hypervisors



VM migration in WAN: storage

- Approaches to **migrate storage** in WAN
 - **Shared storage**
 - ✗ Storage access time can be too slow
 - **On-demand fetching**
 - Transfer only some blocks to destination and then fetch remaining blocks from source only when requested
 - ✗ Does not work if source crashes
 - **Pre-copy** plus **write throttling**
 - Pre-copy VM disk image to destination whilst VM continues to run, keep track of write operations on source (delta) and then apply delta on destination
 - If write rate at source is too fast, use *write throttling* to slow down VM disk writing so that migration can proceed

VM migration in WAN environments: network

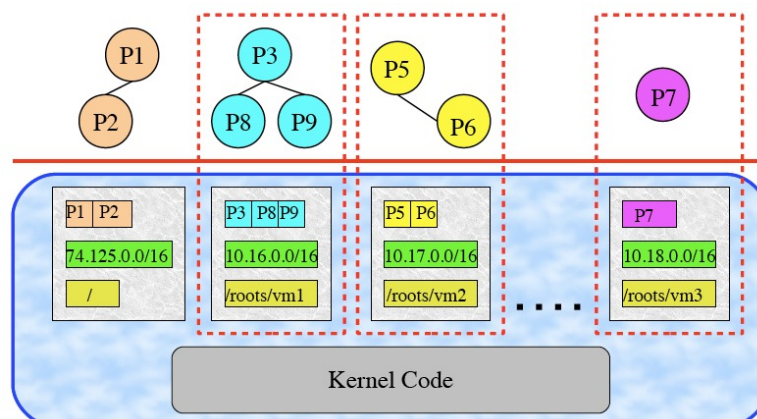
- Approaches to migrate **network connections** in WAN
 - **IP tunneling**
 - Set up an IP tunnel between old IP address at source VM and new IP address at destination VM
 - Use tunnel to forward all packets that arrive at source VM for old IP address
 - Once migration has completed and VM can respond at its new location, update the DNS entry with new IP address
 - Tear down the tunnel when no connections remain that use the old IP address
 - ✗ Does not work if source VM crashes
 - **Virtual Private Network (VPN)**
 - Use MPLS VPN to create the abstraction of a private network and address space shared by multiple data centers
 - **Software-Defined Networking (SDN)**
 - Change control plane, no need to change IP address!

OS-level virtualization

- So far system-level virtualization
- Let's now consider **operating system (OS) level virtualization** (or **container-based virtualization**)
- Allows to run multiple isolated (*sandboxed*) user-space instances on top of a **single OS**
 - Such instances are called:
 - **containers**
 - **jails**
 - **zones**
 - **virtual environments**

OS-level virtualization

- OS kernel allows the existence of multiple isolated user-space instances, called **containers**
- Each container has:
 - Its own set of processes, file systems, users, network interfaces with IP addresses, routing tables, firewall rules, ...
- Containers **share** the same OS kernel (e.g., Linux)

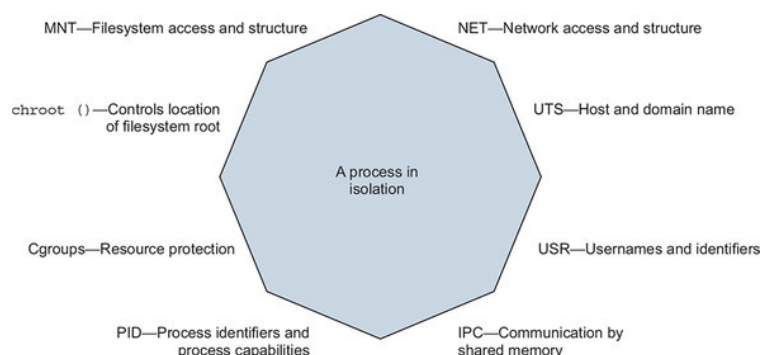


OS-level virtualization: mechanisms

- Which kernel mechanisms to manage containers?
 - Need to isolate processes from each other in terms of sw and hw (CPU, memory, ...) resources
- Main mechanisms offered by Unix-like OS kernel
 - **chroot** (change root directory)
 - Allows to change the apparent root folder for the current running process and its children
 - **cgroups** (Linux-specific)
 - Manage resources for groups of processes
 - **namespaces** (Linux-specific)
 - Per-process resource isolation

Mechanisms: namespaces

- Feature of Linux kernel that allows to **isolate** what a **set of processes** can see in the operating environment (processes, ports, files, ...)
- Kernel resources are partitioned so that one set of processes sees one set of resources, while another set of processes sees a different set of resources
- 6 different types of namespaces



Mechanisms: namespaces

- **mnt**: isolates mount points seen by a container
 - Virtually partitions the file system: processes running in separate mount namespaces cannot access files outside of their mount point
- **pid**: isolates PID space, so that each process only sees itself and its children (PID 1, 2, 3, ...)
- **network**: allows each container to have its dedicated network stack
 - Its own private routing table, set of IP addresses, socket listing, firewall, and other network-related resources
- **user**: isolates user and group IDs
 - E.g., allows a non-root user on host to be mapped with root user within container, without having actual root access to host

Mechanisms: namespaces

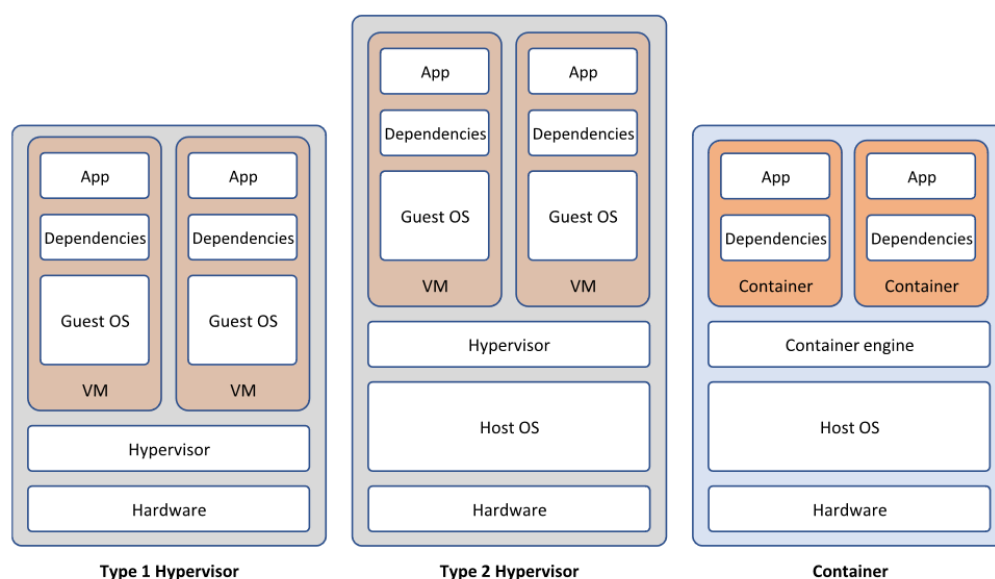
- **uts** (Unix timesharing): provides dedicated host and domain names
 - Allows processes to think they are running on differently named servers
- **ipc**: provides dedicated shared memory for IPC, e.g., different Posix message queues

Mechanisms: cgroups

- cgroups = **control groups**
- Allows to **limit, measure and isolate the use of hw resources** (CPU, memory, block I/O, network) of a **set of processes**
- Low-level filesystem interface similar to sysfs and procfs
 - By default mounted on /sys/fs/cgroup/ directory
- Mechanisms in a nutshell:
 - **namespaces** implements **information isolation**: what a container can see
 - **cgroups** implements **resource isolation**: how much resources a container can use

OS-level virtualization: pros

- VMM-based vs container-based virtualization



In a nutshell: lightweight vs. heavyweight

OS-level virtualization: pros

With respect to VMM-based virtualization (type-1)

- ✓ Minimal performance degradation
 - Apps invoke system calls directly, without VMM indirection
- ✓ Minimum startup and shutdown times
 - Seconds (even msec) per container, minutes per VM
- ✓ High density
 - Hundreds of containers on a single physical machine (PM)
- ✓ Smaller image (footprint)
 - Does not include OS kernel
- ✓ Ability to share memory pages among multiple containers running on same PM
- ✓ Increased portability and interoperability
- ✓ Containerized apps independent of execution environment

OS-level virtualization: cons

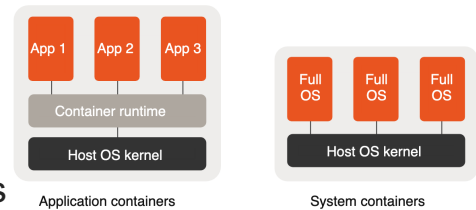
With respect to VMM-based virtualization (type-1)

- ✗ Less flexibility
 - Cannot run different OS kernels simultaneously on same PM
- ✗ Only native applications for supported OS kernel
 - E.g., cannot run Windows container on Linux host
- ✗ Less isolation and higher performance interference on shared system resources
 - Process-level isolation
- ✗ Higher risk of vulnerability and more threats
 - Vulnerability in OS kernel can compromise entire system
 - Since containers share OS kernel, a single compromised container could comprise host OS and other containers

OS-level virtualization: some products

- **Docker**

- The most popular **container engine**
- Provides **application containers**
- Supports Open Container Initiative (OCI), a set of standards for containers
<https://opencontainers.org>



- LXC (Linux Containers) <https://linuxcontainers.org/lxc/>
 - Supported by mainline Linux kernel
 - Provides **system containers** (full OS image)
- Podman <https://podman.io>
 - Supports OCI
 - Docker compatible CLI
- FreeBSD Jail
- OpenVZ (for Linux) <https://openvz.org>
- Virtuozzo Containers

Valeria Cardellini - SDCC 2024/25

92

OS-level virtualization: only Linux?

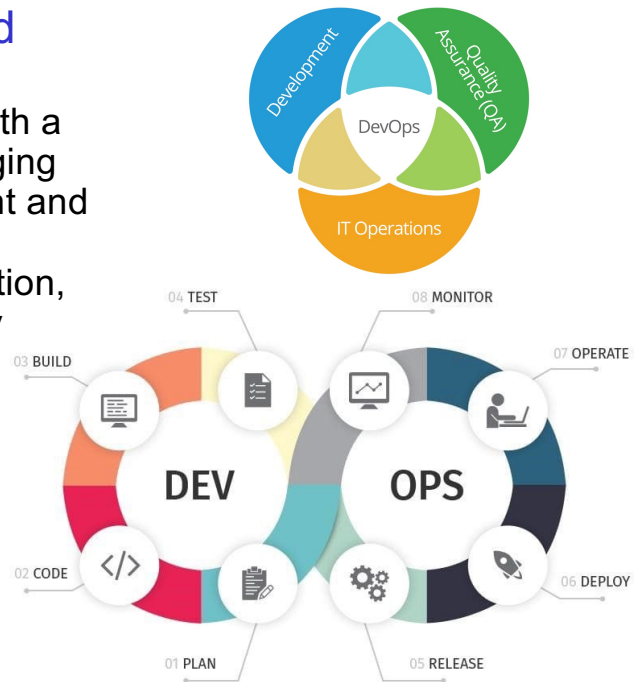
- Windows and OS X also support container-based virtualization
 - E.g., Docker Desktop
<https://www.docker.com/products/docker-desktop/>
- Alternative: install a VM with Linux as guest OS and then install a container-based virtualization product inside VM
 - ✗ Performance loss because of nested virtualization

Valeria Cardellini - SDCC 2024/25

93

Containers, DevOps and CI/CD

- Containers help in the shift to **DevOps** and **CI/CD** (Continuous Integration and Continuous Deployment)
- **DevOps = Development and Operations**
 - Development methodology with a set of practices aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment



Valeria Cardellini - SDCC 2024/25

94

Containers, DevOps and CI/CD

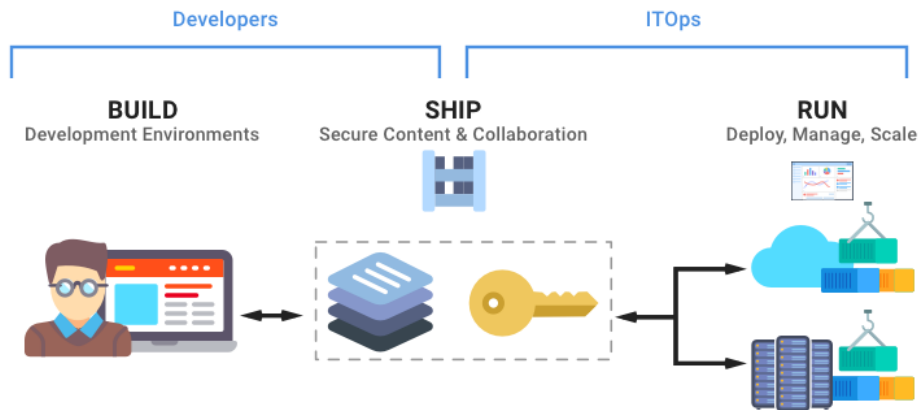
- **CI/CD = Continuous Integration and Continuous Delivery/Deployment**
 - Continuous integration: sw development practice that merges work of all developers working on same project
 - Continuous delivery: ensures reliable and frequent releases
- In DevOps culture, the two practices are combined to enable teams to ship software releases effectively, reliably, and frequently

Valeria Cardellini - SDCC 2024/25

95

Containers, DevOps and CI/CD

- Containers are become a standard to **build, package, share, and deploy apps and all their dependencies**
 - Containers (more than VMs) allow developers to build code collaboratively by sharing images while simplifying deployment to different environments without further configuration

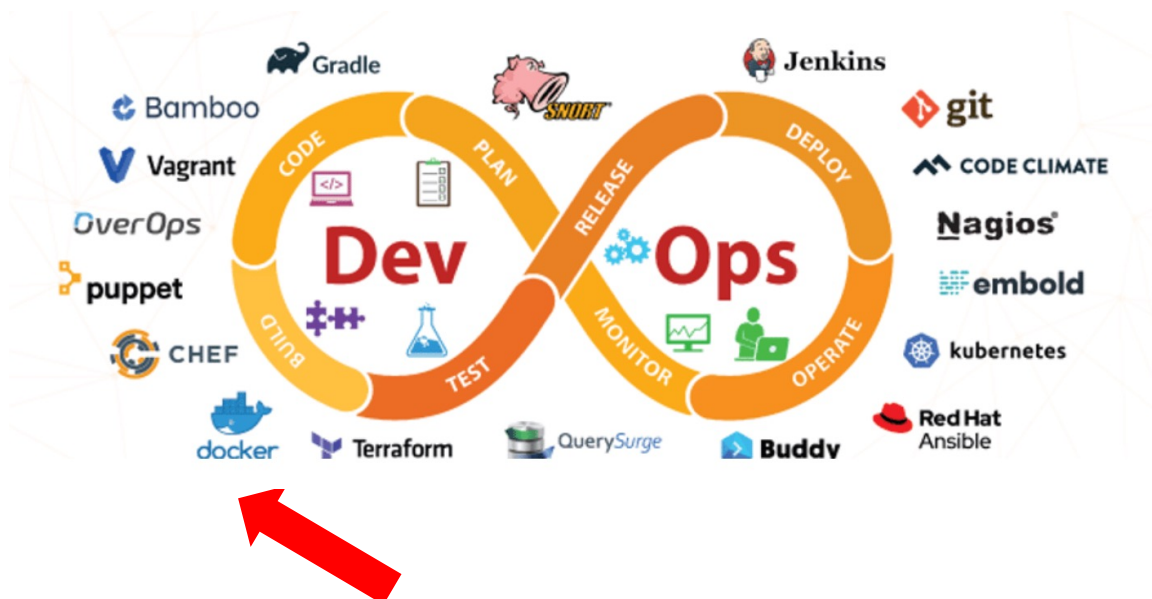


Valeria Cardellini - SDCC 2024/25

96

Containers, DevOps and CI/CD

- Some tools for DevOps



Valeria Cardellini - SDCC 2024/25

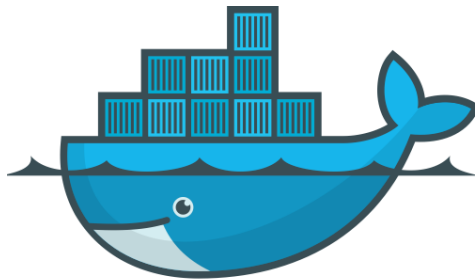
97

Containers, microservices, and serverless

- Using containers
 - App and all its dependencies into single package that can run almost anywhere
 - Using fewer resources than traditional VMs
- Containers are the key enabling technology for **microservices** and **serverless computing**
 - Wrap microservices and functions in containers

Docker

- Let's go into Docker details
<http://www.ce.uniroma2.it/courses/sdcc2425/slides/Docker.pdf>



Container resizing

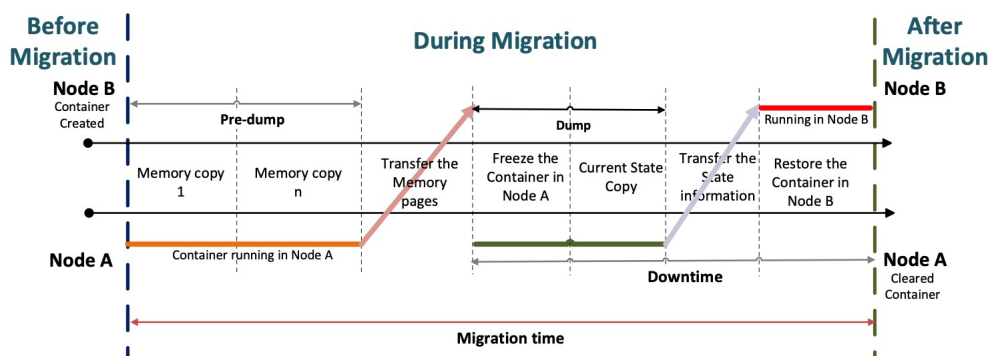
- As for VMs, we can **resize** and **migrate containers**
- Resizing (CPU, memory, I/O) changes dynamically container limits
 - On Docker (not supported for Windows)
`$docker update [OPTIONS] CONTAINER [CONTAINER...]`
 - Some example of container resizing
`$ docker update --cpu-shares 512 containerID`
`$ docker update --cpu-shares 512 -m 300M containerID`

Live migration of containers

- As for VM migration, we need to:
 - Save state
 - Transfer state
 - Restore from state
- State saving, transferring and restoring happen with frozen app: migration downtime
 - Use memory pre-copy or memory post-copy
- No native support in container engines, requires additional tool
- We also need to migrate container image (and volumes) and network connections

Live migration of containers

- Use CRIU tool to support live migration (in Docker and other container engines) through checkpointing and restoration <https://criu.org/>
 - During checkpoint, CRIU freezes running container at source host and collects information about its CPU state, memory content, and process tree
 - Collected information is passed on to destination host, and container is resumed
 - How to: <https://docs.docker.com/reference/cli/docker/checkpoint/>

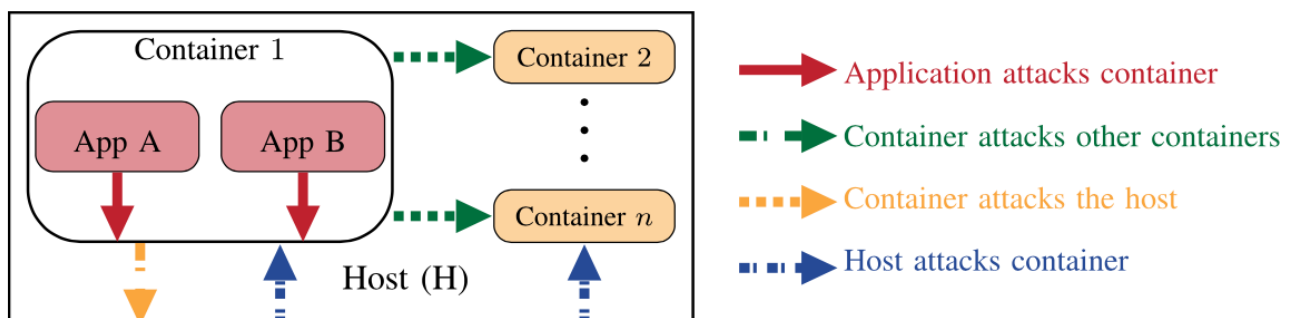


Valeria Cardellini - SDCC 2024/25

102

Container security

- Where attacks come from in a containerized environment?



- Example of attack: *container escape and privilege escalation*
 - Attacker can leverage containerized app's vulnerabilities to breach its isolation boundary, gaining access to host system's resources
 - Once attacker accesses host, it can escalate its privilege to access other containers or run harmful code on host

Valeria Cardellini - SDCC 2024/25

103

Container orchestration

- Sw platforms for managing the deployment of **multi-container packaged applications** in large-scale clusters
- Allow to configure, provision, deploy, monitor, and dynamically control containerized apps
 - Used to integrate and manage **containers at scale**
- Examples
 - **Docker Swarm**
 - **Kubernetes** (next lesson)
 - Nomad <https://www.nomadproject.io/>
- Also available as fully managed Cloud services

Containers in Cloud

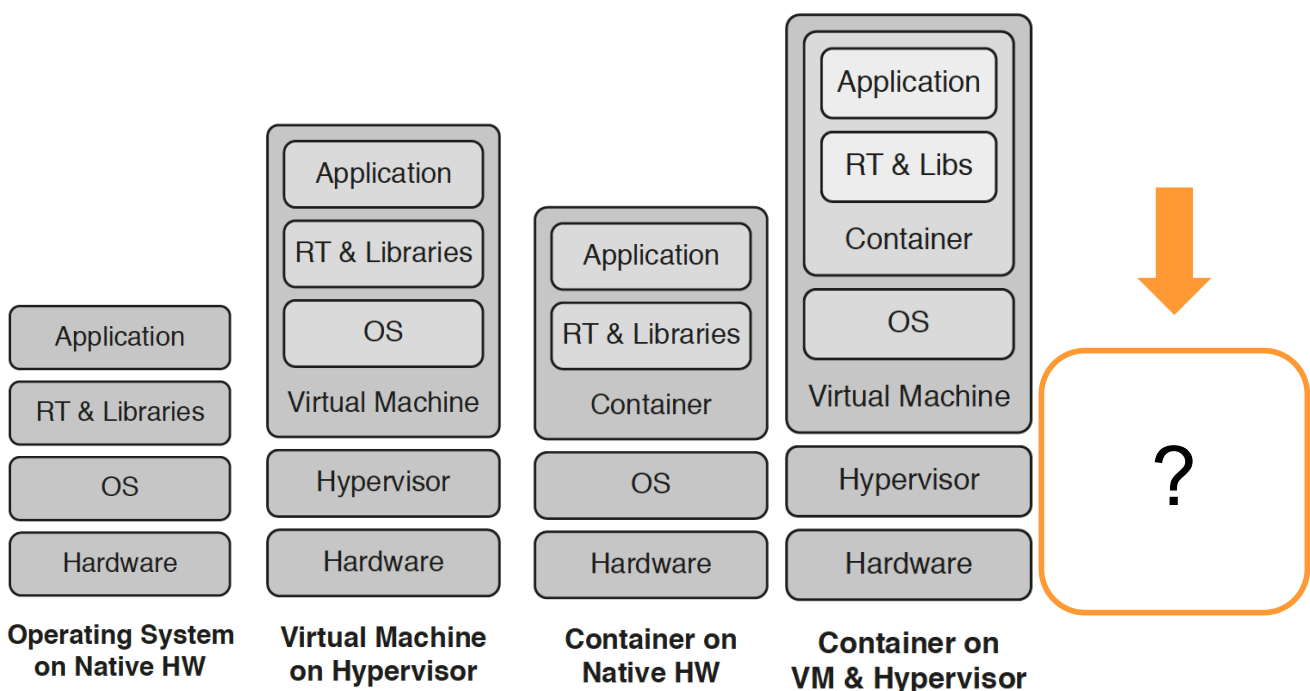
- Containers and container orchestration platforms as first-class Cloud services
- **Container-as-a-Service (CaaS)**
 - Allows developers to manage and deploy containerized applications using a Cloud-based platform
- Features: those of containers + cloud
 - Containerization and container orchestration
 - Networking
 - Platform configuration
 - Security and access control
 - Integration with other Cloud services
- Examples
 - Amazon Elastic Container Service <https://aws.amazon.com/ecs/>
 - Azure Container Instances <https://azure.microsoft.com/products/container-instances>
 - Google Cloud Run <https://cloud.google.com/run>

Hypervisors and containers in Cloud

- Which virtualization technology for IaaS providers?
 - ✓ Hypervisor-based virtualization: greater security, isolation, and flexibility (different OSs on same PM)
 - ✓ Container-based virtualization: smaller-size deployment and thus larger density, reduced startup and shutdown times
- Some question
 - Containers on top of bare metal or in VMs?
 - Are containers replacing VMs?

New lightweight virtualization approaches

- Deployment strategies examined so far

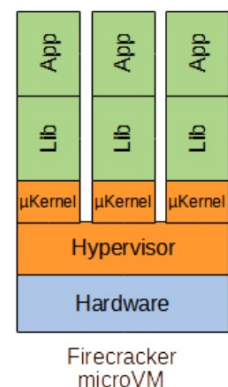


New lightweight approaches to virtualization

- Microservices, serverless computing, edge/fog computing, compute continuum demand for **low-overhead** (or **lightweight**) virtualization techniques, even lighter than containers
 - Additional requirement: improve security
- **MicroVM, lightweight OSs and unikernels**
 - Idea: reduce **OS overhead** and **attack surface**
 - OS overhead: services and tools coming with common OSs (shells, editors, core utils, and package managers) are not needed
 - Attack surface: images contain only the code that is strictly necessary for app to run, thus resulting in minimal attack surface

MicroVM runtimes

- **Tiny, specialized VMMs that run lightweight VMs (microVMs)**
- Goal: reduce memory footprint and improve security of virtualization layer
- **Firecracker**: VMM purpose-built by Amazon for creating and managing secure, efficient and multi-tenant microVMs <https://firecracker-microvm.github.io>
 - Why? To enable AWS Lambda and AWS Fargate
 - Based on KVM but with minimalist design (exclude unnecessary devices and guest functionality)
 - Open source, written in Rust
 - Runs app in **microVM**: < 125 ms startup time and <5 MB memory footprint
 - Scales to thousands of multi-tenant microVMs
 - Supported OS guests inside microVM: Linux and OSv

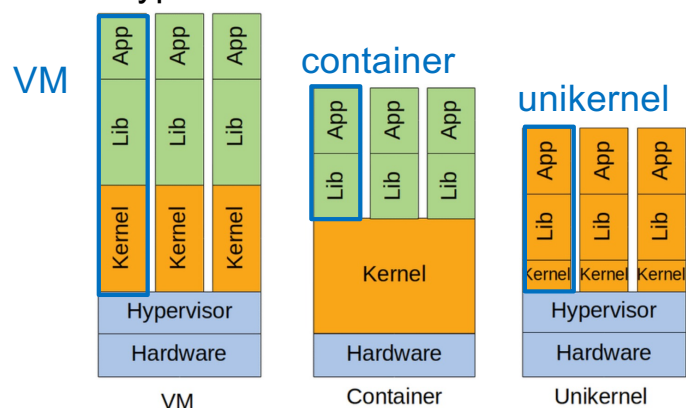


Lightweight operating systems

- Minimal, container-focused OSs, typically with monolithic kernel architecture
 - Special-purpose OSs to run containerized apps
- Fedora CoreOS <https://fedoraproject.org/it/coreos/>
 - Minimal, monolithic and compact Linux distribution
 - Only functionalities required for deploying apps inside containers, together with built-in mechanisms for service discovery, container management and configuration sharing
 - Goal: provide the best container host to run containerized workloads securely and at scale
 - Can be installed on bare metal or on hypervisor
 - Fast bootstrap and small memory footprint
 - Includes Docker and podman

Unikernels

- Specialized, small, lightweight, single-address-space operating system with kernel included as library within application (aka library OS)
 - Sort of very lightweight VM specialized to single app: executable directly into kernel, resulting in **monolithic process that runs entirely in kernel mode**
 - Built by compiling high-level language directly into specialized machine image that runs directly on **hypervisor**
 - Goal: isolation benefits of hypervisor without overhead of guest OS

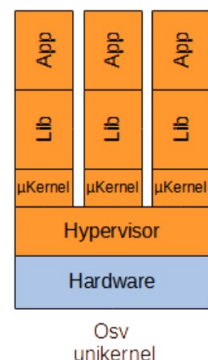


Unikernels: pros and cons

- Pros (from **specialized = high performance**)
 - ✓ Lightweight and small (minimal memory footprint)
 - ✓ Fast app execution (no context switching)
 - ✓ Fast boot (measured in ms)
 - ✓ Secure (reduced attack surface)See <https://www.youtube.com/watch?v=oHcHTFleNtg>
- Cons:
 - ✗ Engineering effort to port app to unikernel
 - ✗ Single language runtime
 - ✗ Limited debugging tools
- Good news: cons almost solved with recent unikernel frameworks

Unikernels: frameworks

- Frameworks (and programming language)
 - MirageOS (OCaml) <https://mirage.io>
 - OSv (C, C++, Go, Python, Java, Rust, ...)
 - Nanos <https://nanos.org/>
 - Unikernel Linux: patch to Linux and glibc to build unmodified programs as unikernel
<https://github.com/unikernelLinux/ukl>
 - Unikraft
- OSv <https://github.com/cloudius-systems/osv>
 - Designed to run single **unmodified Linux application on top of hypervisor** (e.g., KVM, Xen, VMWare, Firecracker)
 - Linux binary compatible unikernel
 - To run app on OSv, need to build image by fusing OSv kernel and app files together <https://github.com/cloudius-systems/capstan>
 - Open-source and fast
 - Can boot in ~5 ms on Firecracker using 11 MB of memory

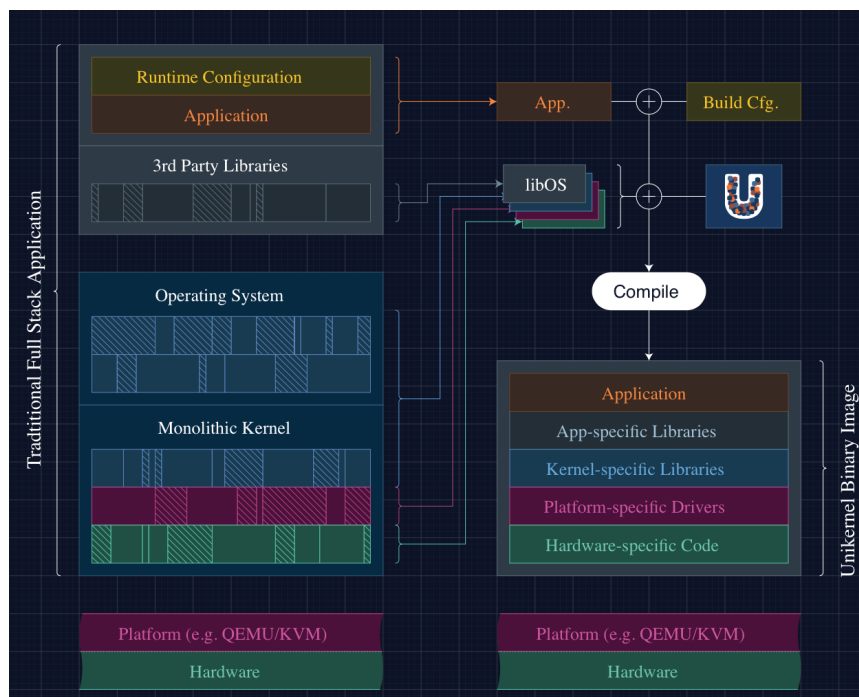


Unikernels: frameworks

- Unikraft <https://unikraft.org>
 - Fast, secure and open-source **Unikernel Development Kit**
 - Goal: build unikernels easily, quickly and without time-consuming expert work
 - Supports multiple hypervisors (e.g., Xen and KVM) and CPU architectures
 - Ability to run wide range of apps (even complex: Redis, Nginx, Memcached) and languages
 - POSIX compliant
 - Written in C

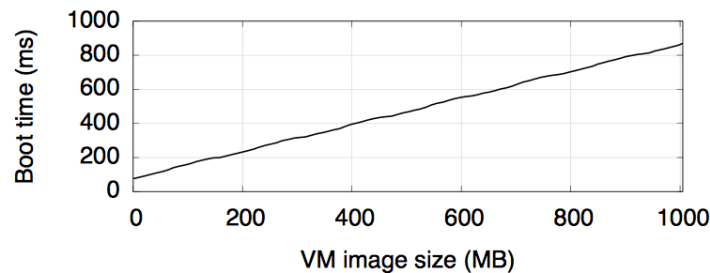
Unikernels: frameworks

- Unikraft

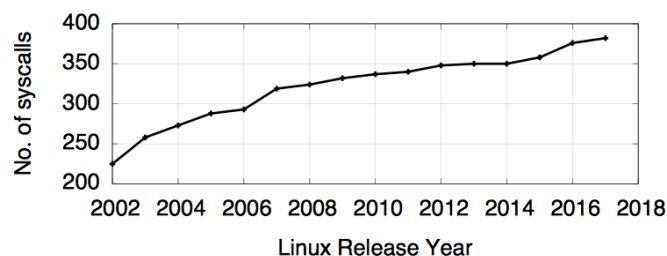


Performance of virtualization approaches

- VM boot times grow linearly with VM size



- Difficulties in securing containers due to growth of Linux syscall API



My VM is lighter (and safer) than your container, SOSP 2017
<https://dl.acm.org/doi/pdf/10.1145/3132747.3132763>

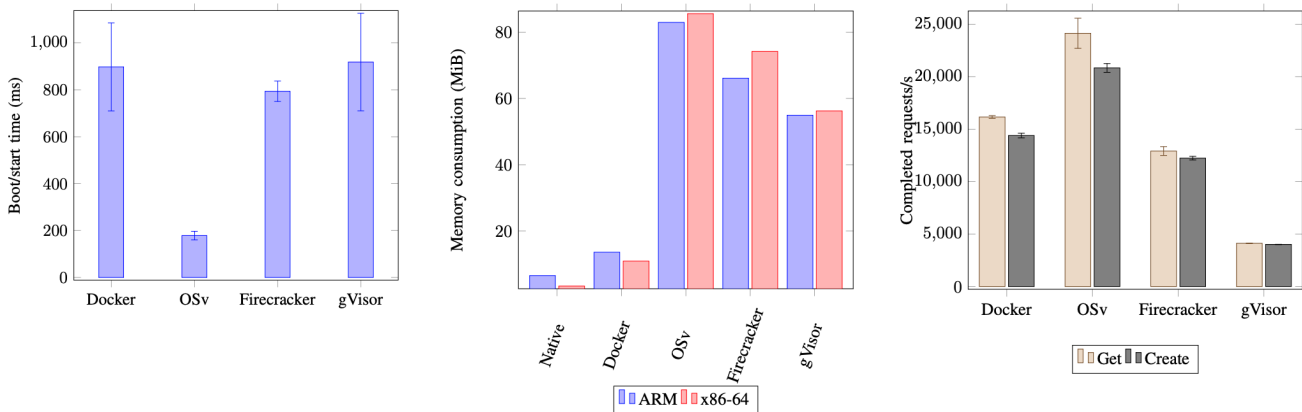
Performance of virtualization approaches

- Performance studies compare hypervisor vs. lightweight virtualization
- Overall result: overhead introduced by containers is almost negligible
 - Fast instantiation time
 - Small per-instance memory footprint
 - High density
- ... but paid in terms of security

Virtualization	Boot time	Image size	Memory footprint	Programming language dependance	Live migration
VM	~5/10 sec	~1 GB	~100 MB	No	Yes
Container	~0.8/1 sec	~50 MB	~5 MB	No	Non-native
Unikernel	<10 msec	<20 MB	~10 MB	Partially	No

Performance of virtualization approaches

- Lightweight virtualization is needed for edge computing and compute continuum
- Overall result: no clearly winning solution so far, each one has its own strengths and weaknesses



A functional and performance benchmark of lightweight virtualization platforms for edge computing, EDGE 2022 <https://ieeexplore.ieee.org/document/9860335>

Valeria Cardellini - SDCC 2024/25

118

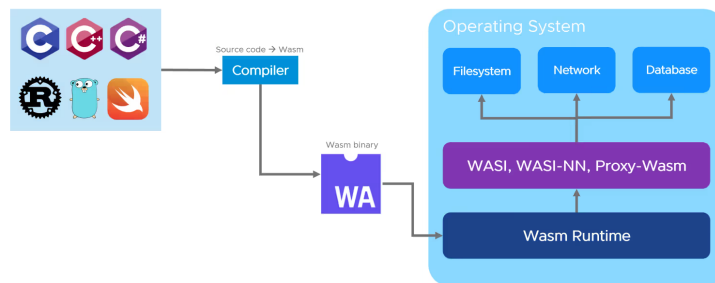
WebAssembly



- WebAssembly (**Wasm**): **safe, portable, binary code format** designed for efficient execution and compact representation <https://webassembly.org>
- Features
 - Open standard <https://www.w3.org/TR/wasm-core-2/>
 - Designed as portable compilation target for different programming languages
 - Born to safely execute JavaScript code in browsers
 - **Memory-safe, sandboxed** execution
 - Computational model based on **stack VM**

WebAssembly: features

- Wasm code is validated and executed in a **memory-safe, sandboxed** environment
 - Wasm interacts with host system via **WebAssembly System Interface** (WASI)
 - Wasm module cannot directly perform an OS system call due to sandboxing, but imports equivalent WASI functions instead
- Steps: write code in a variety of supported languages, compile it to Wasm, and execute in a Wasm runtime



Valeria Cardellini - SDCC 2024/25

120

WebAssembly: features

- Computational model is **stack-based VM**
https://en.wikipedia.org/wiki/Stack_machine
 - Code consists of sequences of instructions executed in order
 - Instructions manipulate values on an **operand stack**
 - Instructions pop argument values and push result values
 - Simple instructions perform basic operations on data, pop arguments from stack and push results back to it
 - E.g., `i64.add` takes two `i64` values from stack and pushes the result onto the stack
 - Control instructions alter control flow
- JVM is a famous example of stack-based VM


Valeria Cardellini - SDCC 2024/25

121

WebAssembly: example

- Factorial function written in C and its corresponding WebAssembly code after compilation
 - In .wat text format (human-readable textual representation of Wasm)

```
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```




```
(func (param i64) (result i64)  
    local.get 0          # put arg[0] on stack  
    i64.eqz              # compare top of stack to zero  
    if (result i64)      # if it is zero  
        i64.const 1      # put 1 on stack  
    else  
        local.get 0      # put arg[0] on stack  
        local.get 0      # put arg[0] on stack  
        i64.const 1      # put 1 on stack  
        i64.sub          # subtract the top 2 values in stack  
        call 0           # Call function #0 (return value is on the stack)  
        i64.mul          # Multiply  
    end)'
```

WebAssembly: example

- Factorial function written in C and its corresponding WebAssembly code after compilation
 - In .wasm binary format

```
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```



```
00 61 73 6D 01 00 00 00  
01 06 01 60 01 7E 01 7E  
03 02 01 00  
0A 17 01  
15 00  
20 00  
50  
04 7E  
42 01  
05  
20 00  
20 00  
42 01  
7D  
10 00  
7E  
0B  
0B
```

WebAssembly: pros and cons

- ✓ Efficient: near-native execution speed
- ✓ Secure: memory-safe sandboxed execution, which prevents data corruption and security breaches
- ✓ Language-, platform-, hardware-independent
 - Does not privilege any particular language
 - Can run as stand-alone VM
 - Can be compiled on all modern architectures, desktop or mobile devices and embedded systems
- ✗ In development
- ✗ Support is not equal for all languages
- ✗ Multiple runtimes (e.g., Wasmtime, Wasmer, WasmEdge) with different features: challenging choice

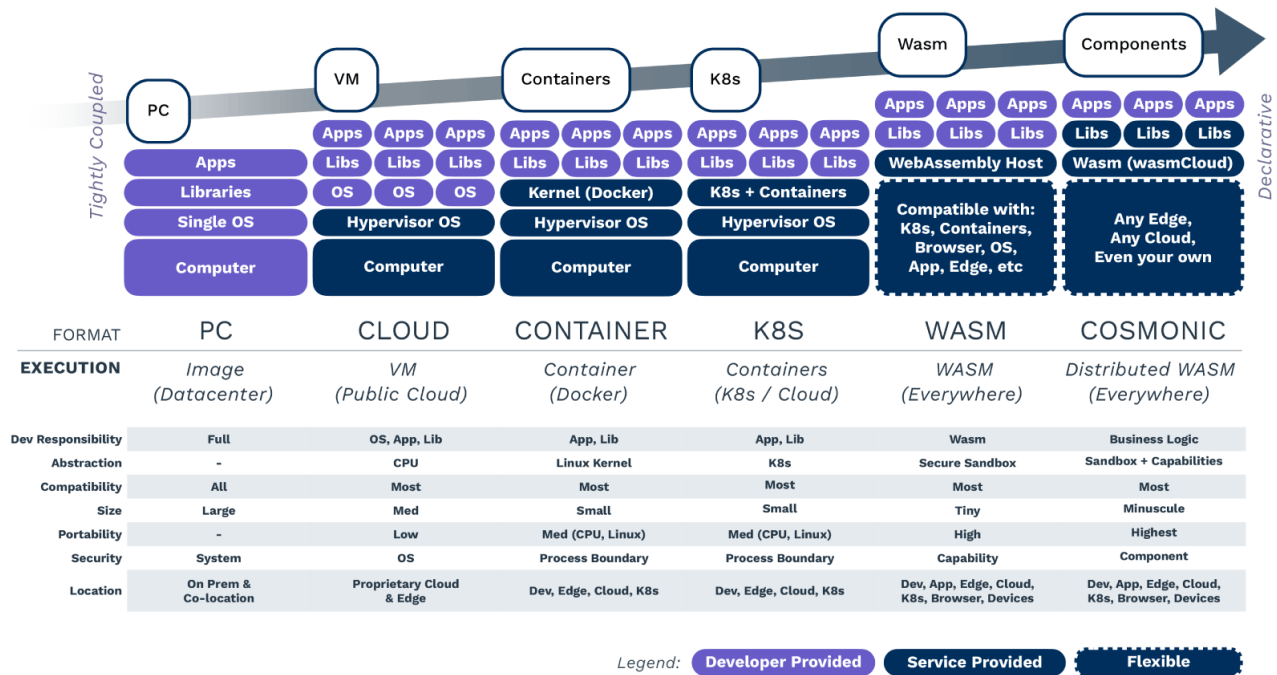
WebAssembly

- How to try: Wasm applications with Linux containers in Docker (beta)

<https://docs.docker.com/desktop/features/wasm/>

```
$ docker run \  
  --runtime=io.containerd.wasmedge.v1 \  
  --platform=wasi/wasm \  
  secondstate/rust-example-hello
```

The full scenario



References

- Section 3.2 of van Steen & Tanenbaum book
- Sections 4.1, 4.4, 4.13, 5.1-5.5, 5.8 of Marinescu book
- Smith and Nair, The architecture of virtual machines, IEEE Computer, 2005 <https://ieeexplore.ieee.org/iel5/2/30853/01430629.pdf>
- Bugnion et al., Hardware and software support for virtualization, 2017 <https://kartikgopalan.github.io/680v/books/HSSV.pdf>
- Agache et al., Firecracker: Lightweight virtualization for serverless applications, NDSI 2020 <https://www.usenix.org/conference/nsdi20/presentation/agache>
- Kuenzer et al., Unikraft: fast, specialized unikernels the easy way, EuroSys 2021 <https://dl.acm.org/doi/pdf/10.1145/3447786.3456248>