Cloud Virtualisation

Fog and Cloud Computing

Domenico Siracusa, Fondazione Bruno Kessler (FBK)

08/03/2021

Introduction

Preliminary information i

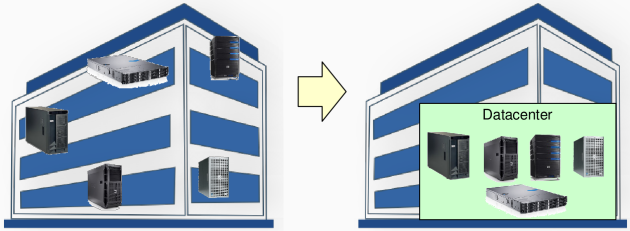
- This set of slides is based on a previous version created by Alex Palesandro and Prof. Fulvio Risso
- Material
  - Dan C. Marinescu, *Cloud Computing: Theory and Practice* (Chapter 10)
  - G. J. Popek and R.P. Goldberg, *Formal requirements for virtualizable third generation architecture*, Communications of the ACM, 17(7): 412-421, 1974
  - *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*, Vmware white paper
  - *Performance evaluation of Intel EPT hardware assist*, Vmware white paper

Preliminary information ii

- Additional readings
  - P. Barham et al., *Xen and the art of virtualization*, Proc. 19th ACM Symp. Operating System Principles, pp. 164-177, 2003
  - A. Kivity, *KVM: the Linux virtual machine monitor*, Proc. Linux symposium, Ottawa, pp.225-230, 2007
  - M. Rosenblum, T. Garfinkel, *Virtual machine monitors: current technology and future trends*, Computer, 38(5), pp.39-47, 2005
  - X. Zao, K. Borders, A. Prakash, *Virtual machine security systems*, Advances in computer science and engineering, pp. 339-365

The path toward computing virtualisation (1)

- Initially, servers have been installed across the company
- . . . then, moved to datacenters in order to preserve data and make management easier. . .
- . . . and discovered that there were so many servers around, mostly idle. . .
- . . . and that we cannot consolidate multiple apps on a single server



The “one application per server” rule

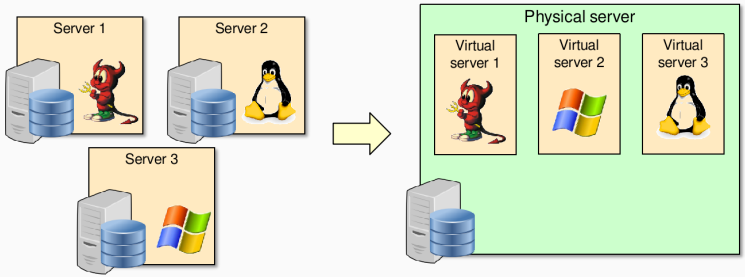
- The reason? The “one application per server” rule, due to the failure of popular OSes to provide
  - Configuration/shared components full isolation
    - App A requires shared library v. 1.0, App B requires v. 2.0
    - A is certified by the vendor only on OS version X, patch Y
    - If A runs in a different environment, the vendor will not be responsible of possible misbehavior
  - Temporal isolation for performance predictability
    - If A consumes a lot of CPU, performance of B will be affected
    - If A sends a of of traffic, performance of B will be affected
  - Strong spatial isolation for security and reliability
    - If A crashes, it may compromise B
    - A can send traffic directly to B, bypassing the (outside) firewall

The path toward computing virtualisation (2)

- Impossibility to consolidate multiple apps on the same server was a problem also for another reason
  - At the same time (late 90's), the Moore Law was still valid, but... no longer with the increase of the operating frequency (clock), but with the number of CPU cores
- Results? Huge amount of servers, massively underutilized, and consuming a lot of electrical power
  - A lot of CPU cores available, but not used because of the "one app per server" rule

Computing virtualisation in a nutshell

- Computing Virtualisation is a flexible way to share hardware resources (e.g. CPU, memory, I/O) between different (un-modified) operating systems
- *Virtualisation broadly describes the seperation of a service request from the underlying physical delivery of that service* (VMware definition)



Virtualisation in the past

- Proposed for the first time around 1960 on IBM 360 to share expensive hardware resources
  - Very common in the 60's, e.g. with mainframes
- Limited importance in 70-80's with the diffusion of cheap mini and personal computers
- Back in 90's as a new way to address new needs of reliability and consolidation of business servers
- Currently focus on x86 architecture

Virtualisation: advantages i

- Isolation
  - Critical applications could run in different and easily isolated OS
  - Different services could run on the same hosts with an improved degree of isolation
    - Malicious or misbehaving (e.g., bugs) applications or services cannot compromise services running in other VMs
    - We can assign different CPU cores to different VMs
- Consolidation
  - Completely different OSes could transparently run on the same hardware at the same time, saving hardware resources
    - Minimizing operating costs

Virtualisation: advantages ii

- Optimize energy consumption
- Flexibility and Agility
  - Complete control over the execution information of all VMs
    - Possibility to pause and restart OS execution
    - Possibility to migrate the VM (virtual server) to another host (even when running), e.g., to consolidate servers (e.g., during night), or to give it more resources (when a physical machine becomes overloaded)
- Possibility to duplicate a running VM, e.g., to address a peak load
- Disaster recovery
- Rapid deployment of new servers (spawn new instances)

Virtualisation: limitations

- Additional overhead in running the same application
  - Each application has its own OS running, hence more requirements in terms of disks, memory and CPU
  - The additional amount of resources strongly depends on the application and technology in use
  - However, this overhead is usually considered acceptable for a wide range of different applications and operating conditions
- More difficult to handle heterogeneous hardware
  - E.g., offering the access to special components (e.g., powerful GPUs, hardware offloading cards, etc.) to some applications

Virtualisation: usage scenarios

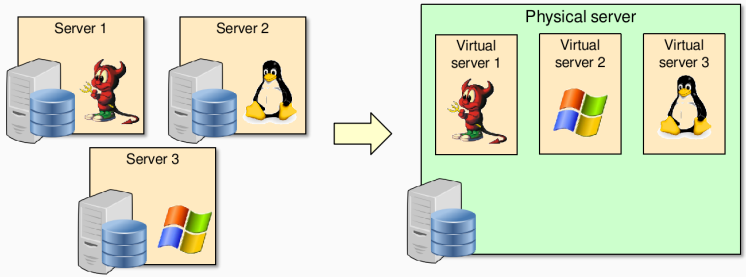
- Servers virtualisation
  - Several services, each one with their environment (e.g., their own OS) sharing the same hardware, with a given degree of isolation
- Workstation/Desktop virtualisation
  - Easy to share hardware facilities
  - Easier to restore previous configurations
  - Possibility to reproduce the same hardware/software configuration on several physical machines

Virtualisation: economic considerations

- Servers virtualisation is definitely more important economically
  - Workstation/Desktop virtualisation is not a huge economic driver; in fact, most of the products in that area are free
- Main products:
  - Server virtualisation: Vmware ESXi, Microsoft Hyper-V, Citrix XenServer, Linux KVM
  - Workstation/Desktop virtualisation: Oracle Virtualbox, Linux KVM, Vmware Workstation

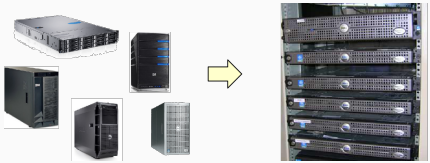
Common Off The Shelf (COTS) hardware i

- Focus has now shifted from real servers to virtual servers
  - No longer important to have a physical machine with a given set of characteristics (e.g., powerful CPU, 16GB RAM, etc.)
  - We can create dynamically a virtual server with the requested characteristics (e.g., CPU, memory, disks, number of NICs, ...), based on the actual necessities



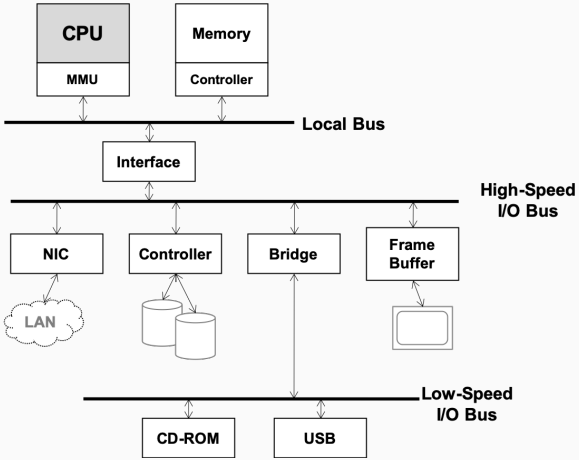
Common Off The Shelf (COTS) hardware ii

- Consequence: (corporate) users can buy tons of equivalent servers, with exactly the same hardware characteristics, and:
  - Aggregate them together in a datacenter
  - Virtualize their resources in order to create the specific set of virtual servers (a.k.a., VMs) we need
- Computing hardware becomes a commodity
  - Just buy the "cheapest" (e.g., white label) COTS hardware, asking for volume discounts



Initial definitions

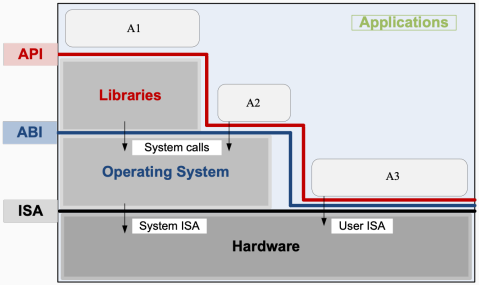
Computer System Organization



Layering in complex systems i

- Layering – a common approach to manage system complexity
  - Minimizes the interactions among the subsystems of a complex system
  - Simplifies the description of the subsystems; each subsystem is abstracted through its interfaces with the other subsystems
  - We are able to design, implement, and modify the individual subsystems independently
- Layering in a computer system
  - Hardware
  - Software
    - Operating system
    - Libraries
    - Applications

Layering in complex systems ii



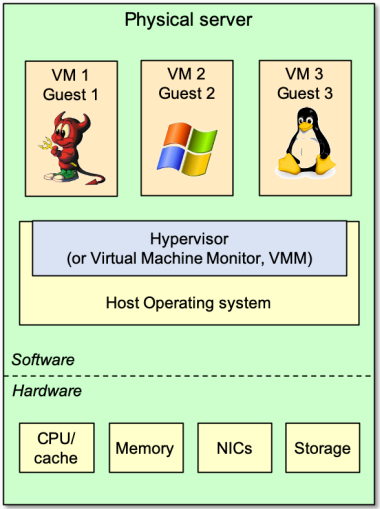
- An application
  - uses library functions (A1)
  - makes system calls (A2)
  - executes machine instructions (A3)

Interfaces

- Instruction Set Architecture (ISA) - at the boundary between hardware and software
- Application Binary Interface (ABI) - allows the ensemble consisting of the application and the library modules to access the hardware
  - ABI does not include privileged system instructions, instead it invokes system calls
- Application Program Interface (API) - defines the set of instructions the hardware was designed to execute and gives the application access to the ISA
  - Includes HLL library calls which often invoke system calls

Definitions

- Virtual Machine (VM)
  - Software emulation of a physical machine that executes OS+apps such as being in a physical one
- Host OS
  - OS running on the physical machine, in charge of the virtualisation of the hardware
  - It can include the Hypervisor (next slide)
- Guest OS
  - OS running in the VM, which should not be aware of running in a virtualized environment



Hypervisor (VMM) i

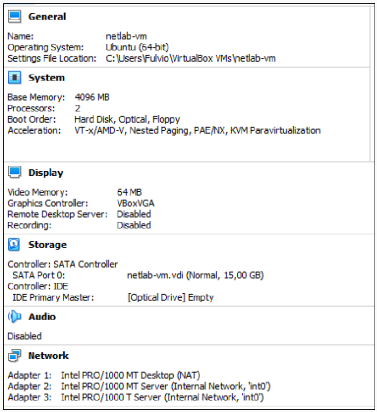
- The Hypervisor (or *Virtual Machine Monitor*, VMM) is the software in charge of the virtualisation process
- It has to virtualize the hardware resources, such as CPU, memory, and other devices (e.g., NICs)
- Virtualisation means
  - Assigning a distinct set of resources to each VM, when possible, and guaranteeing that each VM cannot get access outside its boundaries
    - E.g., the memory can be partitioned in disjoint spaces and assigned to different VMs
  - Arbitering the access to shared resources, in case those cannot be partitioned
    - E.g., unique NIC, shared among all VMs

Hypervisor (VMM) ii

- In practice, often a stripped-down OS
  - Often Linux-based
  - A limited set of native drivers to manage the hardware
  - A virtualisation layer exports a set of "standard" devices to the upper-layer OS
    - Usually, we do not virtualize the latest video card
    - However, most important characteristics of the hardware can be exploited "natively"
    - Enable hosted OS to support a limited set of hardware
- The hypervisor may be attacked
  - Much smaller and more defendable than a conventional OS

Virtual Hardware

- The VMM has to provide a "virtual hardware" to the guest VM, with the exact characteristics specified in a given hardware profile
- The real hardware may have a little to do with the virtualized hardware
  - E.g., the server has one NIC from vendor X, and the VM profile specifies two NICs from vendor Y



Virtualised components

- Computing virtualisation represents the most common approach for Cloud Computing, according to the IaaS model
  - A flexible way to share hardware resources between different (un-modified) operating systems
- This requires the VMM to virtualize:
  - CPU
  - Memory
  - I/O (e.g., NICs, storage, video, etc.)

CPU virtualisation

- VMM assigns to the VM one or more CPUs to execute Guest OS
- Assume CPU architecture of physical machine is the same of the virtualized one (e.g., Intel x64)
  - The Instruction Set Architecture (ISA) of the virtualized machine is the same (or a subset) of the physical ISA
- In case the ISA is different we need to rely on “emulation” instead of “virtualisation”
  - It requires the binary translation between two really different ISA, which is usually rather slow
    - e.g. Apple "Rosetta" transitioning from MIPS to intel CPUs
  - Although some particular CPU architectures are able to perform this binary translation automatically, this feature is not available on mainstream architectures (e.g., x64)

Definitions

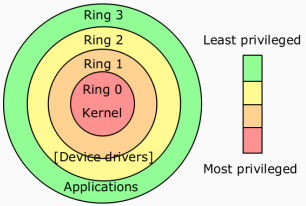
- Virtual machine: "A virtual machine is taken to be an efficient, isolated duplicate of the real machine" (Popek&Goldberg, 1974)
- A Virtual Machine Monitor (VMM) must satisfy three characteristics:
  1. It exports execution environments, VMs, essentially identical to the real machine
     - Applications and OSes can run unmodified
  2. It efficiently executes the virtualised system
     - A "statistically dominant portion of the virtual processor instructions should be executed by the real processor"
  3. The VMM should have the complete control of real system resources
     - Each virtualised system should be able to access only hardware resources under the authorisation of the VMM

Types of CPU virtualisation in the x86 world

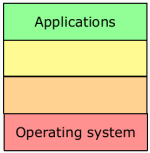
- Full Virtualisation (Vmware workstation, Virtual Box) - 1999
  - Guest OS can run unmodified in the hypervisor
- Paravirtualisation (Xen, Oracle OVM) - 2003
  - Guest OS need to be modified in order to be executed
- Hardware assisted Virtualisation (KVM) - 2007
  - Hypervisors exploit hardware features now included in the CPU/chipset

x86 Hardware Privilege Ring

- x86 defines 4 rings with different privilege levels
  - Ring 0: designed for the most privileged part of code, the OS kernel
  - Ring 1-3: designed to run application and services with different levels of trustworthiness
- Currently, only Rings 0 and 3 are used in modern OS



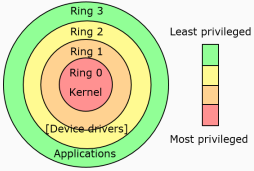
Traditional ring structure in Intel x86 CPUs



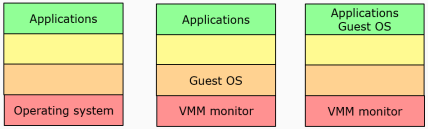
Traditional OS

x86 Hardware Privilege Ring and CPU virtualisation

- Virtualisation can use "ring de-priviledging", a technique that runs all guest software at priviledge level greater than zero, but:
  - 0/1/3 model: in the x86 architecture, some privileges with respect to memory accesses are granted to 0-2 rings, hence the guest OS can interfere with the VMM
  - 0/3/3 model (“ring compression”): the above problem is solved, but the GuestOS is no longer protected from malicious applications



Traditional ring structure in Intel x86 CPUs



Traditional OS

Virtualized OS (0/1/3 model)

Virtualized OS (0/3/3 model)

Definition: priviledged instruction

- A privileged instruction is a CPU instruction that needs to be executed in a privileged hardware context
- It generates a *trap* if called when the CPU is running in the wrong context (e.g., at ring 3)
- Examples of instructions that cannot be allowed in user-level applications
  - "HALT" instruction: a user-defined applications cannot halt a computer from running
  - I/O instructions: a user-defined application cannot directly interact with a specific I/O device, unless the OS guarantees exclusive access to it
- A privileged instruction cannot be executed by a guest OS running at a ring greater than 0

Definition: sensitive instruction

- A sensitive instruction is CPU instruction leaking information about physical state of processor
- Example: the x86 PUSH %CS register can read the code segment selector (%cs) and learn its current privileged level
  - The Code Segment includes 2-bits for the Requested Privilege Level, which enables who executes this instruction to know which privileged level the software is currently running into
- In order to be virtualizable, all sensitive instructions of a CPU must be privileged

Traps i

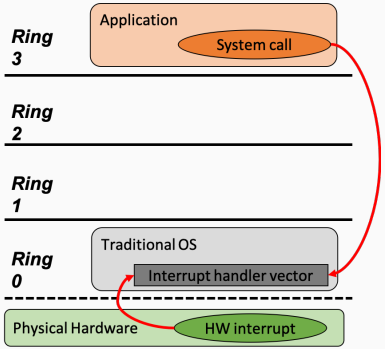
- What is trap?
  - When CPU is running in user mode, some internal or external events, which need to be handled in kernel mode, take place
  - Then CPU will jump to hardware exception handler vector, and execute system operations in kernel mode

Traps ii

- When does a trap occur?
  - Exception
    - Invoked when unexpected error or system malfunction occur
    - For example, execute privilege instructions in user mode
  - System Call
    - Invoked by application in user mode
    - For example, application ask OS for system I/O
  - Hardware Interrupts
    - Invoked by some hardware events in any mode
    - For example, hardware clock timer trigger event

A note on system calls implementation i

- Traditional OS:
  - When application invokes a system call:
    - CPU will trap to interrupt handler vector in OS
    - CPU will switch to kernel mode (Ring 0) and execute OS instructions
- When hardware event:
  - Hardware will interrupt CPU execution, and jump to interrupt handler in OS

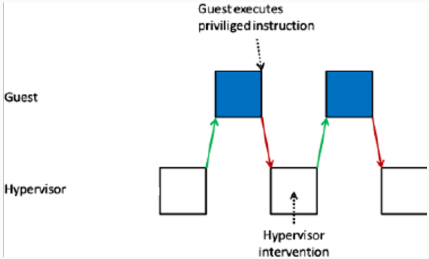


A note on system calls implementation ii

- Traditional way:
  - Userland code (e.g. glibc) generates a software interrupt (instruction `INT xx`)
  - The generic interrupt routine of the OS is started, which determines where to jump in the OS code to serve the above interrupt
  - Kernel jumps to the requested code, serves the interrupt, then returns to the caller (instruction `IRET`)
    - Requires to load and parse the content of several memory locations (rather slow)
- New way: use SYSENTER/SYSEXIT (or SYSCALL/SYSRETURN in x64)
  - Userland code writes the memory address of the target kernel routine in a specific register, then calls SYSENTER and kernel runs in a very fast transition
- Interesting reading: John Gulbrandsen, "System Call Optimization with the SYSENTER Instruction," 2004

Trap & Emulate (T&E) paradigm i

- Model proposed by Popek & Golberg in 1974
  - Guest OS executed in an unpriviledged domain
  - When Guest OS executes a privileged instructions, processor launces a trap that is intercepted by the VMM
  - VMM emulates the effect of the privileged instructions for the guest OS (if legitimate) and gives control back to guest OS execution

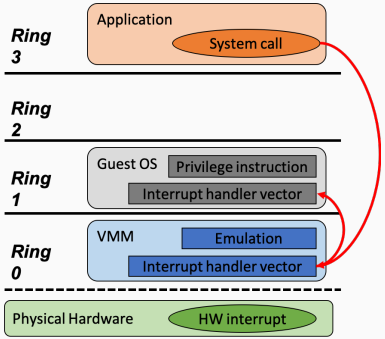


Trap & Emulate (T&E) paradigm ii

- What does VMM do with traps that occur within the virtual machine?
  - If trap is caused by an application, pass trap to the guest OS
  - If trap is caused by guest OS, handle the trap by adjusting the state of the VM
- Need to require operating system support
  - All traps and exceptions originating inside the VM must be handled by the VMM
  - Most of the time guest apps and guest OS simply use the physical processor normally

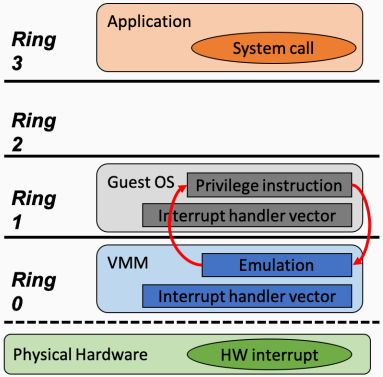
Trap & Emulate (T&E) paradigm iii

- System Call
  - CPU will trap to interrupt handler vector of VMM
  - VMM jump back into guest OS
    - extra context switch
    - gets worse if the guest is not able to handle the interrupt routing
    - time to execute single syscall could be 10x for the same call in the host OS



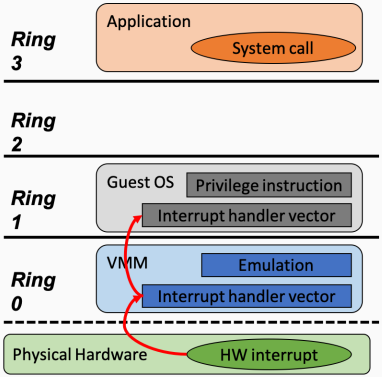
Trap & Emulate (T&E) paradigm iv

- Privileged Instruction
  - Running privileged instructions in guest OS will be trapped to VMM for instruction emulation
  - After emulation, VMM jump back to guest OS



Trap & Emulate (T&E) paradigm v

- Hardware Interrupt
  - Hardware make CPU trap to interrupt handler of VMM
  - VMM jump to corresponding interrupt handler of guest OS



Traditional x86 architecture is not easily virtualisable

- In the T&E, each privileged instruction, if executed in an unprivileged domain, has to generate a (time-consuming) trap detectable by the hypervisor
- This is not always true for all CPU architectures
  - In particular, for those, like x86, conceived with a minimalistic approach and when the virtualisation was not popular at all
  - x86/x64 presents some sensitive instructions that do not trap if executed in a unprivileged level, such as *POPA*, *POPF*
  - The x86/x64 architecture is said to be "non-virtualizable"
- Therefore:
  - Several sensitive instructions are not detectable by the VMM
  - Consequentially, the VMM cannot emulate the correct behavior during the execution

Traditional x86 architecture is not easily virtualisable: readings

- Very nice readings about the problems present in the original x86 architecture:
  - "Intel Technology Journal," Volume 10, Issue 03, August 2006. Available at `https://www.intel.com/content/dam/www/public/us/en/documents/research/2006-vol10-iss-3-intel-technology-journal.pdf`
  - "Bringing Virtualisation to the x86 Architecture with the Original VMware Workstation," ACM Transactions on Computer Systems, November 2012. Available at `https://dl.acm.org/doi/10.1145/2382553.2382554`
  - Much shorter paper: "Intel Virtualization Technology," IEEE Computer Magazine, May 2005. Available at `https://ieeexplore.ieee.org/document/1430631`

Possible solutions

- Parse the instruction stream and detect all sensitive instructions dynamically
  - Interpretation (BOCHS, JSLinux)
    - Old and slow technique, not presented here; emulating a single ASM instruction may originate an overhead of at least one order of magnitude
  - Binary translation (VMWare, QEMU)
    - No OS source modification, but performance overhead
- Change the operating system: paravirtualisation (Xen, L4, Denali, Hyper-V)
  - Near-native performance, but OS modifications
- Make all sensitive instructions privileged!
  - Hardware supported virtualisation (Xen, KVM, VMWare): Intel VT-x, AMD SVM

Dynamic Binary Translation (DBT) i

- Proposed by VMware in 1998 to overcome x86/x64 architectural limits (initially x86)
- Fully virtualized approach
  - Again, guest OS does not need to be modified
  - Enables x86/x64 virtualisation without hardware assistance or modification to source code
- Idea: the VMM dynamically translates x86 "non-virtualizable" ISA to virtualizable one at run-time
  - Dynamic: translation is done on the fly during execution and interleaved with code execution
  - Binary: VMM translates the binary code, not the source code
    - For better performance translation is done on blocks of code and not on single instructions

Dynamic Binary Translation (DBT) ii

- Compatibility
  - No need of a specific HW support
  - No need of an OS modification
- Performance
  - Virtualisation overhead is significantly higher than other techniques
  - Translation speed can be improved by adopting caching techniques to recognize (and translate) significant instruction patterns
  - Several instructions or execution patterns (system calls) are significantly slower than real execution

Vmware workstation: T&E + DBT

- Original VMware VMM combined a T&E engine with a system-level DBT
- Guest OS is executed in Ring 1 and VMM at Ring 0
- VMM inspects dynamically the code of the Guest OS and replaces non-trappable portions of the code (sensitive instructions) with "safe" instructions
  - In some cases, the resulting code requires the intervention of the VMM (e.g., interacting with the hardware)
  - In other cases, the resulting code is able to perform all its operations without the VMM intervention
    - e.g., access to GuestOS structures, while “normal” (privileged) instructions would return the equivalent structure present at VMM level (ring 0)

Para-Virtualisation (PV) i

- Main idea: let the guest know that it is running in a VM
  - Guest OS knows that, in some cases, it has to give the control to the VMM
  - Para-virtualisation is no longer Full Virtualisation
- OS Guest is explicitly modified to be virtualizable, changing the interface provided to make it easier to implement
  - System calls are replaced with specific hypervisor calls (hypercall)
  - Non-virtualizable instructions are replaced with hypercalls
    - They do not produce traps
  - No more trap&emulate
    - Also access to the page tables is para-virtualized
  - ABIs of guest OS will not change, hence application could be executed without any modification

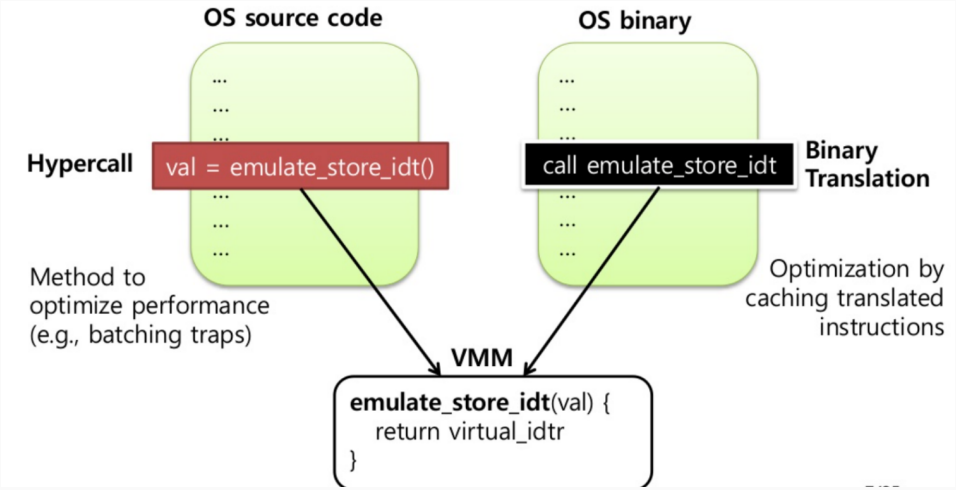
Para-Virtualisation (PV) ii

- Guest OS is explicitly deprivileged
  - It is executed in RING 1
- Efficient mechanisms are introduced in the Guest OS kernel to ease the communication with the hypervisor:
  - Guest-to-Hypervisor: privileged instructions are replaced with synchronous para-virtualized equivalents (hypercalls)
  - Hypervisor-to-Guest: hypervisor could notify certain events asynchronously to the guest
    - Like interrupts, Guest OS could defer their handling

Para-Virtualisation (PV) iii

- Performance
  - PV does not need to emulate motherboard or device bus
  - No longer dynamic translation; the OS has to be patched ahead of time
  - Implementation is simpler, hence much faster, than DBT
- Compatibility
  - Only modifiable OS could be virtualized
  - No Windows
  - OK Linux and BSD

Source-level vs. binary-level modifications



Hardware-assisted virtualisation (HVM) i

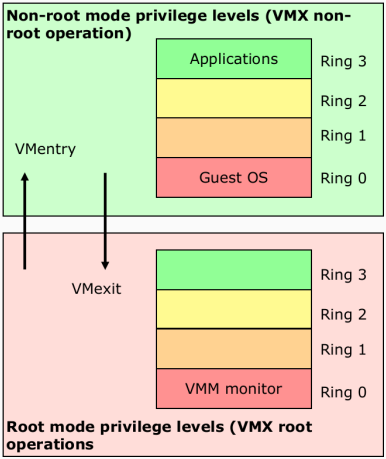
- Two problems still open:
  - Make easier the implementation of a hypervisor: even in PV, the VMM is still complex and tricky
  - Provide efficient x86 Full Virtualisation: several OS could not be modified and DBT is slow
- HVM aims at providing a solution to those problems by proposing an efficient "Trap & Emulate" approach to virtualisation thanks to an additional hardware support
- First implementation of Intel VT-X / AMD SVM (now AMD-Vi) in 2005
  - Functionally equivalent implementation in Intel and AMD CPUs, but incompatible

Hardware-assisted virtualisation (HVM) ii

- Idea: avoid sensitive instructions, either because are "promoted" as privileged, or because the VMM can dynamically configure which instructions have to be trapped
  - Some instructions (e.g., INVD, Invalidate CPU Internal Caches) cause VM exits unconditionally and therefore can never be executed in VMX non-root operation.
  - Other instructions (e.g., INVLPG, Invalidate TLB Entries) and all events can be configured to do so conditionally, using the VM execution control fields in the VMCS

Hardware-assisted virtualisation (HVM) iii

- Processor is provided with a new running mode (VMX, Virtual Machine eXtensions)
- When VMX is turned on CPU enables two different running modes called "Operating Levels", equivalent to a new set of rings
  - Root Mode: higher class of privilege
  - Non-Root Mode: keeps existing four rings
- The VMM runs in fully privileged Root Mode
- Guest OS run in VMX non-root operating level in Ring 0, applications in VMX non-root Ring 3

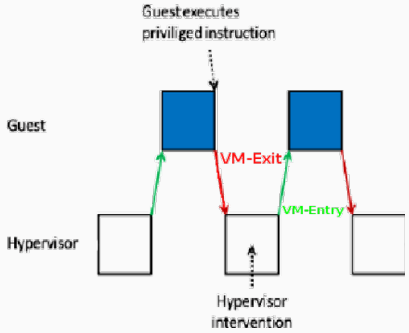


Details: VMX instructions

- If sytem code tries to execute instruction violating isolation of the VMM or that must be emulated via software, hardware traps it and switch back to the VMM
- CPU enters non-root mode via the new VMLAUNCH and VMRESUME instructions, and it returns to root mode for a number of reasons, collectively called VM exits
- VM exits should return control to the VMM, which should complete the emulation of the action that the guest code was trying to execute, then give control back to the guest by re-entering non-root mode. All the new VM instructions are only allowed in root/system mode
- e.g, while in non-root mode, the 2 INT istruction may cause a switch from non-root/user to non-root/system, and the IRET instruction may return from non-root/system to non-root/user

HVM iv

- Intel's VTx allow a VMM to specify several conditions
  - According to this configuration, the actions attempted by a Guest VM will get "trapped"
- When a trapping condition is triggered, VMM takes the execution control and it emulates the correct behavior
- Transition between Root and Non-Root levels
  - VM entry: VMM to guest transition
  - VM exit: guest to VMM transition
- Registers and address space swapped in a single atomic operation
- Such transitions are main source of overhead



Cost for VMEntry/VMExit

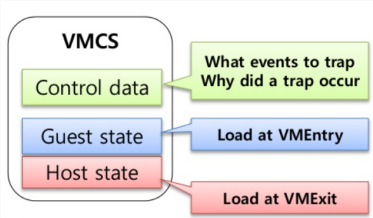
Microarchitecture	Launch Date	Cycles
Prescott	3Q2005	3963
Merom	2Q2006	1579
Penryn	1Q2008	1266
Nehalem	3Q2009	1009
Westmere	1Q2010	761
Sandy Bridge	1Q2011	784

Table 1: Hardware round-trip latency.

Agesen Ole, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. "Software techniques for avoiding hardware virtualisation exits." In Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 373-385. 2012. Available at http://www.cs.columbia.edu/~cdall/candidacy/pdf/Agesen2012.pdf

Virtual Machine Control Structure (VMCS)

- VT-x (SVM) introduces a new memory structure, VMCS (VMCB)
  - It mirrors all register modifications needed to set a certain configuration in the guest OS
  - Concretely it represents the control panel of the VM storing information about:
    - Guest state
    - Host Processor information
    - Control data (e.g. trapping conditions)
  - Introduced dedicated instructions to modify it (VMWRITE / VMREAD)



HVM: summary

- Hardware Assistance technology greatly simplifies VMM implementation
  - Transparent way to make "Full virtualisation"
  - HVM mode minimizes VMM intervention
  - No VM Entry/VM Exit for system call triggered by user applications
- HVM reduces virtualisation overhead
  - Round-trip time for VM Entry/VM Exit represents the most important source of virtualisation overhead
  - CPU manufactures are pushing for more efficient VM entry/exit to reduce the cost of this operation
- KVM provides hardware-assisted virtualisation

CPU virtualisation: summary

- DBT features the best compatibility but it is slow
  - Used mainly for legacy hardware without HVM
  - Still used nowadays e.g., in desktop virtualisation (Virtualbox)
- Paravirtualisation could achieve great performances but it could virtualize only a subset of OS
  - Fully implemented only for Linux and BSD guests
- HVM is the most used technique nowadays
  - Performance could vary a lot according to CPU generation

Memory virtualisation

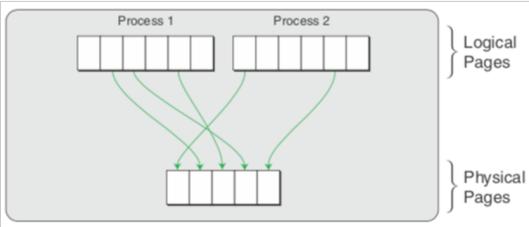
- Modern operating systems use “memory paging” as a technique to access as contiguous dispersed locations in the physical memory
- Each OS keeps a set of tables to translate the virtual memory addresses to physical addresses
- X86/x64 can handle this task in hardware with a dedicated Memory Management Unit (MMU)

Virtual memory

- Processes use virtual addresses
  - Addresses start at 0
  - OS lays processes down on pages
- MMU (Memory management unit)
  - Translates virtual to physical addresses
    - Maintains page table (big hash table) for the mapping
  - TLB (Translation lookaside buffer): cache of recently used page translations

Mapping virtual to physical

- OS maintains mapping of logical page numbers (LPNs) to physical page numbers (PPNs) in page table structures
- When logical address is accessed, hardware walks these page tables to determine the corresponding physical address
- For faster memory access, x86 hardware caches the most recently used LPN->PPN mappings into TLB

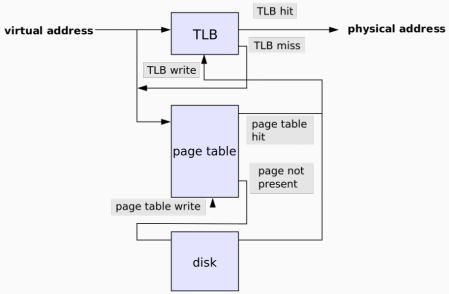


Why virtual memory

- Simplicity
  - Every process gets illusion of whole address space
- Isolation
  - Every process is protected from every other
- Optimization
  - Reduces space requirements

Translation Lookaside Buffer (TLB)

- TLB: fast, fully associative memory
  - Caches page table entries
  - Stores page numbers (key) and frame (value) in which they are stored
- TLB Sizes: 8 to 2048 entries

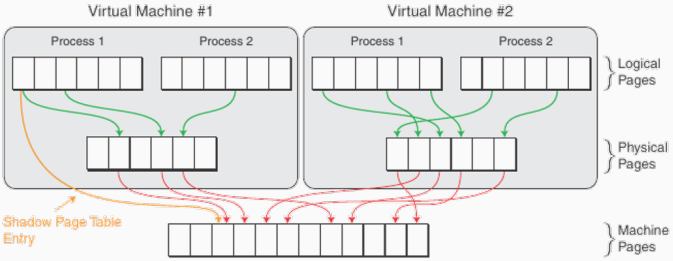


Memory Virtualisation: Shadow Page Table i

- An extra level of translation is required with VMs
  - Guest virtual address -> Guest physical address -> Machine physical address
- To avoid two address translations, a “Shadow Page Table” is introduced
  - The shadow page table stores and keep track of the mapping between Guest logical address and Machine physical pages
  - It is invisible from the guest point of view and it will be used by the CPU for a translation, when the guest is active

Memory Virtualisation: Shadow Page Table ii

- PPN->MPN maintained by VMM in internal data structures
- LPN->MPN stored by VMM in shadow page table exposed to HW
- Most recently used LPN->MPN in HW TLB



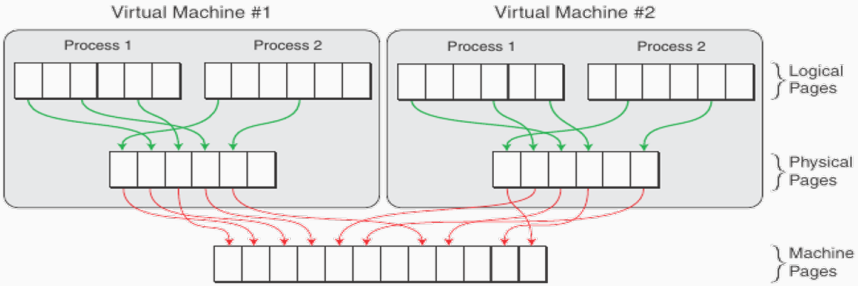
Memory Virtualisation: Shadow Page Table iii

- The VMM is in charge of keeping the Shadow Page Table synchronized with the guest OS page table
  - OS will modify its ordinary page tables
- An important extra overhead is introduced in presence of page faults
  - An extra page table has to be kept synchronized when Guest update its own page tables -> overhead for apps for which the guest frequently updates its page table

Memory Virtualisation: EPT/RVI i

- In order to avoid Shadow Page Table overhead, Intel/AMD introduces Extended Page Table / Rapid Virtualisation Indexing
  - Different technologies implementing the same concept
- Traditional page tables translate LPN->PPN (guest table walk)
- VMM maintains PPN->MPN mappings in an additional level of page tables, called nested
  - Both the traditional (guest) page tables and the nested page tables are exposed to the CPU
  - When a logical address is accessed, the hardware walks the guest page tables as in the case of native execution (no virtualisation), but for every PPN accessed during the guest page table walk, the hardware also walks the nested page tables to determine the corresponding MPN

Memory Virtualisation: EPT/RVI ii



Memory Virtualisation: EPT/RVI iii

- This approach removes the need of VMExit associated with page table virtualisation and update
  - Guest could now keep update its page table without any overhead
- EPT/RVI introduce a “double-level” page walk performed by the hardware
  - No need for the shadow table anymore
  - This increases the cost of a single page walk
  - TLB cache becomes critical to guarantee good performance
  - For memory intensive task, the utilization of larger pages of memory (“Huge Pages”) could increase the TLB hit ratio thanks to the two TLB present in Intel processors
    - One TLB for normal pages, one for huge pages

Memory virtualisation – Tagged TLBs

- Additional optimization of EPT
- Each TLB line has an identifier (Virtual Processor ID) for each virtual processor
- When a line has to be accessed, the ID prevents wrong access to other virtual processors cache lines
- This technique allows several virtual processors coexist on the TLB at the same time
  - The tagged TLBs eliminate the need for TLB flushes on every VMEntry and VMExit
  - Great importance with EPT/RVI where TLB page miss cause an important overhead

Memory Virtualisation: summary

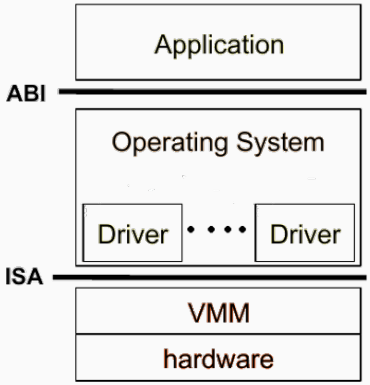
- Shadow Page Table technique used only in absence of EPT/RVI support
- EPT/RVI used whenever is possible
    - Bigger pages could be used in order to limit the pressure on the TLB cache

I/O virtualisation

- Several techniques could be adopted for virtualizing I/O devices
  - Device emulation
  - Para-virtualized device
  - Direct assignment
- The choice of the technique that has to be used depends on:
  - The type of device
  - Shared/Dedicated to a single Guest OS

I/O virtualisation: device emulation

- VMM proposes to the Guest OS an emulated device, which implements in software an hardware specification
- Guest OS completely ignore that the device is emulated
- Guest OS will use the same driver used with an equivalent physical device
- VMM has to remap device communication with the physical device in real OS

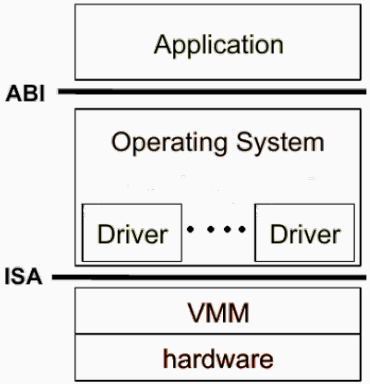


Device emulation: pros and cons

- An approach simple and easy to set up
  - No need to install any dedicated driver to handle device I/O operations
  - A single physical device could be multiplexed with several emulated device
- However:
  - I/O operations are generally slower than physical ones, with higher latency
    - Particularly critical in case of devices with high I/O (NIC, disk)
  - I/O operations may increase substantially the CPU load
    - This represent an extra-work that is completely absent in case of a physical device

I/O virtualisation: para-virtualized device

- Guest OS is enriched with dedicated drivers
- Similar to para-virtualisation
  - The main difference is that para-virtualisation needs core modules of the kernel to be modified
    - This requires the kernel to be patched
  - Instead, para-virtualized drivers could be added as external modules to the most part of modern OS
    - Here we require dedicated device drivers to be added/replaced



Para-virtualized device: status

- PV drivers are provided from modern hypervisors for several kind of devices, such as:
  - Network cards
  - Disks
  - Graphical video cards
- In addition, special devices could be conceived to optimize resource consumption in virtualized OS
  - Memory Ballooning driver

A possible PV driver: memory ballooning

- The VMM, during the initialization of a VM, defines its memory size
- Normally, the VMM needs to allocate statically all the memory assigned to the guest
  - The actual utilization in the guest OS is unknown to the hypervisor
- In order to avoid to reserve memory that is not used by the guest, the hypervisor exploits a "Memory Ballooning" PV driver installed in the Guest OS
  - This driver provides to the hypervisor the information about current memory occupation of the Guest
  - This allows the hypervisor to over-commit memory allocation to several guests

I/O virtualisation: direct assignment

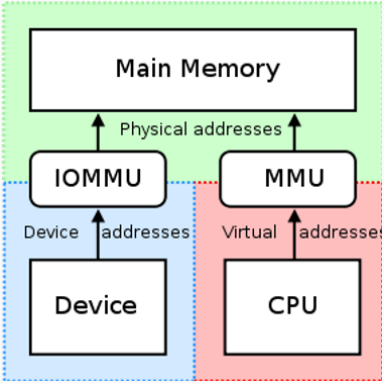
- Direct assignment allows the VM to directly communicate with the physical device
  - Also called “Device pass-through”
- Guest OS will handle the device with the traditional (dedicated) driver
  - No need for an additional driver in the host OS, as the device will be totally handled by the guest OS
- This technique requires the exclusive assignment of a device to a VM
  - No device multiplexing over several VMs
- "PCI passthrough" is an example of this technique

Direct assignment: problems

- Despite its apparently simple approach, the "direct access" to hardware devices is very complex indeed
  - Direct Memory Access (DMA) cycles have to be performed on *guest* physical addresses
  - Devices do not know the mapping between guest and host physical addresses for a particular VM
  - This could potentially lead to a memory corruption
  - Memory corruption can be avoided if the VMM or host OS intercepts the I/O operation and performs the correct translation
  - But this is slow and can introduce a significant overhead in I/O operations

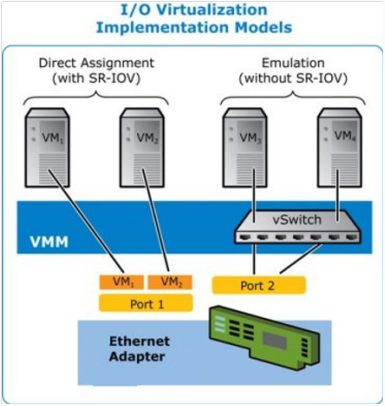
IOMMU

- Dedicated extension introduced by hardware manufacturers to boost and make direct assignment easier to be implemented in hypervisor
- Like the MMU, the IOMMU:
  - Remaps the addresses accessed by the hardware according to the same table used to map guest-physical address to host-physical addresses
  - DMA cycles will safely access the correct memory locations



SR-IOV (Single Root Input/Output Virtualisation)

- The PCI-e standard defines SR-IOV as a mechanism to allow several directly assigned devices to be shared among VMs
  - Specially designed for network cards
- SR-IOV defines the possibility from the device to present several virtual devices, "virtual functions" to the OS
- The VMM will directly assigning a virtual function to a VM
- The hardware will handle itself the device multiplexing



I/O virtualisation: summary

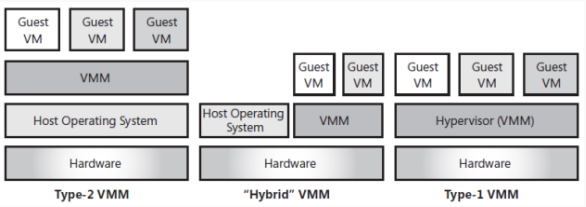
- Device emulation is slow and computationally expensive but offers a great flexibility
  - Used for devices that are not critical in terms of performance or legacy OS
- Para-virtualisation is flexible and faster even if still computationally expensive
- Direct assignment is the fastest and less expensive technique but requires exclusive allocation of the device
  - In case of a network card, SR-IOV mitigates the problem of exclusive allocation of the device

Hypervisor architectures

- Traditionally, hypervisors presents mainly two architectures, type-1 and type-2
- They follows different design criteria:
  - Performance
  - Easiness of deployment & utilization
- Hybrid approaches are possible as well

Hypervisor architectures - Type 1

- The hypervisor runs directly on bare metal
  - No extra layers between Hypervisor and hardware
  - Normally able to provide the best performance
  - Installation and deployment may be cumbersome
    - The hypervisor is basically an Operating System where all the unnecessary components (e.g., printing, etc.) are stripped down
    - The problem is having the proper drivers for all your hardware

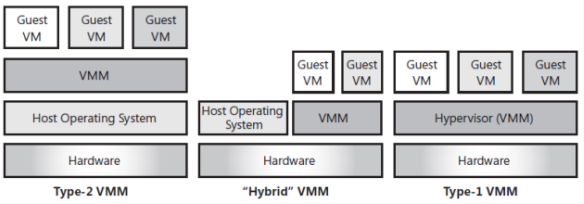


Type 1 - Popular implementations

- Vmware ESXi
- Xen
- Microsoft Hyper-V
  - When enabling Hyper-V on your Windows 10 desktop, your operating system will no longer executed on the bare hardware, but as VM
  - Hence, running a further hypervisor (e.g., VirtualBox) in Windows would not be a great choice, as we are asking the system to use nested virtualisation
  - Instead, when Hyper-V is enabled, any further virtualized OS executed by Hyper-V would work in parallel with your Windows 10

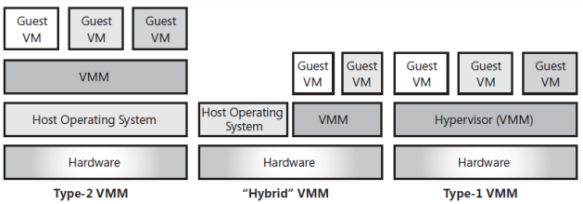
Hypervisor architectures - Type 2

- The hypervisor runs on a host OS, as a normal application
  - Installation and VM creation are simple tasks
  - Normally less performing than type 1
- Available implementations:
  - VirtualBox
  - VMware workstation



Hypervisor architectures - Hybrid approach

- The hypervisor is now implemented in the OS kernel
  - The host OS is itself the hypervisor, thanks to a specific component
- Normally simple to install and deploy
  - Drivers and support coming from the mainstream OS
- Very good performance, KVM is an example of hybrid approach



OS-level virtualisation

Full virtualisation with VMs: pro and cons

Advantages

- Compatible with existing applications
- Support for different operating systems
- Each application (i.e., VM) can have its own execution environment
  - Kernel version, libraries
- Excellent isolation, backed by hardware mechanisms
  - CPU, memory

Problems

- Overhead for the necessity to execute the guest OS
  - Memory (hundred of MB)
  - CPU
- Necessity to configure and keep up-to-date each instance of guest OS
  - "Operations" are a non-negligible cost in real life
- OS booting time (tens of seconds or more) may not be acceptable

Going cloud native

- A recap on the historical perspective
  - First, we had individual servers (with Operating Systems, applications, et)
  - Then, we had datacenters, with many servers that look like "cattle"
  - But... do we really need different operating systems?
    - Required for historical reasons
    - Required if we want to support real users (desktop environment)
    - Required if we want to support different hardware (e.g., peripherals, etc.)
  - In cloud, we want to have the same commodity hardware, we don't have desktop environments, hence we can achieve greater operational efficiency if we reduce the number of OS to just one: Linux

Idea behind lightweight virtualisation

- Create a system that can guarantee the nice properties of computer virtualisation (scalability, elasticity, isolation) but that consumes less resources
- Lightweight virtualisation is appropriate when:
  - No need for a classical virtual machine
  - The overhead of a classical VM is not acceptable
  - We would like to have an isolated environment that is quick to deploy, migrate and dispose with possibly little or no overhead at all
  - We would like to scale both vertically (thousands of "lightweight VMs" on the same machine) and horizontally (deploy the "lightweight VMs" on many different machines available in a data center)

Lightweight virtualisation

- Use Operating System-level virtualisation or Application-level virtualisation instead of full hardware virtualisation
  - Both are software virtualisation technologies
- In case of OS-level virtualisation, the “hypervisor” is the Linux kernel itself
  - No longer required a dedicated hypervisor
- Virtual environments, replace classical VMs
  - Virtual environments are also know as virtual private servers, jails, containers
  - Virtual environments feature a given extent of resource management and isolation, usually less than what is achievable with VMs
  - Apps are executed inside these virtual isolated environments

Requirements for lightweight virtualisation

- Fine-grained control of resources of the physical machine
  - Possibility to partition and control resources (e.g., CPU cores, RAM) among the different environments
- Security and isolation guarantees
  - Each virtual environment should be assigned to a different app/user, and avoid that a misbehavior in a virtual environment affects the others
- Possibility to manage the entire datacenter as an unique entity, such as with cloud toolkits
  - Even better, capability to integrate lightweight virtualisation with a cloud toolkit in order to have the flexibility to deploy VM, containers, etc., upon request

Possible technologies

- In theory, several different choices, but, in the end, a few answers
  - Linux containers (LXC) and LXC-based software
  - Other Operating Systems look irrelevant here
- Technologies actually used
  - Linux cgroups and namespaces
  - Linux Containers (LXC)
  - Docker

Process isolation: motivation

- Community recognised need to implement strong process isolation in Linux Kernel
  - Server running multiple services wants to be sure that possible intrusion on a service does not compromise the entire machine
  - Necessity to safely execute arbitrary or unknown programs on your server
    - e.g., Amazon Lambda
    - e.g., code submitted by third parties, such as students, coding events (hackathon), continuous integration (automatic testing of new software releases)
  - This requires strong process isolation, without adopting techniques such as hardware virtualisation (overheads)
    - e.g., a different operating system is not needed

cgroups and namespaces

- Originally developed to strengthen isolation between processes
  - Nothing to do with full virtualization
- Cgroups and namespaces can be leveraged to create a new form of lightweight virtualization, without the overhead associated to full virtualization
- Controllable properties
  - CPU and Disk quotas
  - I/O rate and Memory limit
  - Network isolation
  - Check-pointing and live migration
  - File system isolation
  - Root privilege isolation

Linux cgroups

- Linux kernel feature to limit, account, isolate or deny resource usage to processes or groups of processes
  - Represents the elementary brick that enables a fine-grained control to single processes and resources, providing a way to implement OS-level virtualisation
- Consists of two parts: kernel feature and user-space tools that handle the kernel control groups mechanism
- Example: CPU quota
  - The advantage of control groups over *nice* or *cpulimit* is that the limits are applied to a set of processes, rather than to just one
  - For instance, CoreOS, the minimal Linux distribution designed for massive server deployments, controls the upgrade processes with a cgroup: downloading and installing of system updates does not affect system performance

Linux cgroups documentation

- cgroups are quite complex to use
- Some good documentation at
  - Red Hat `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/`
  - Kernel.org `https://www.kernel.org/doc/Documentation/cgroup-v2.txt`

Cgroup features

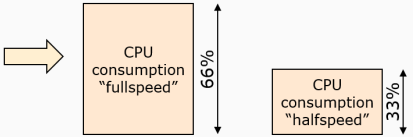
- Resource limiting
  - Groups can be set to not exceed a configured memory limit, which also includes the file system cache
- Prioritization
  - Some groups may get a larger share of CPU utilization, Disk I/O throughput, or Network bandwidth
- Accounting
  - Measure group resource usage, e.g. for billing purposes
- Control
  - Freezing groups of processes, their checkpointing and restarting

Cgroups example: limiting the CPU consumption

```
netlab@VM:~$ sudo apt install cgroup-tools

# Copy and past this text (till second EOF) in a terminal
# to create a script that does an infinite loop
netlab@VM:~$ cat <<'EOF' > infinite_full.sh
#!/bin/bash
while true
do
    i=i+1
done
EOF

netlab@VM:~$ chmod +x infinite_full.sh

netlab@VM:~$ cp infinite_full.sh infinite_half1.sh
netlab@VM:~$ cp infinite_full.sh infinite_half2.sh

netlab@VM:~$ sudo cgcreate -g cpu:/cpufullspeed
netlab@VM:~$ sudo cgcreate -g cpu:/cpuhalfspeed

netlab@VM:~$ sudo cgset -r cpu.shares=1024 cpufullspeed
netlab@VM:~$ sudo cgset -r cpu.shares=512 cpuhalfspeed

netlab@VM:~$ sudo cgexec -g cpu:cpufullspeed ./infinite_full.sh &
netlab@VM:~$ sudo cgexec -g cpu:cpuhalfspeed ./infinite_half1.sh &
netlab@VM:~$ sudo cgexec -g cpu:cpuhalfspeed ./infinite_half2.sh &
```

Output of the 'top' command (on a single-core machine)

```
Tasks: 165 total,   4 running, 161 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   1990.8 total,    410.9 free,    491.1 used,   1088.7 buff/cache
MiB Swap:    711.4 total,    711.4 free,      0.0 used.   1309.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 5179 root      20   0   22612   3492   3256 R  66.3   0.2   1:30.24 infinite_full.sh
 5222 root      20   0   22612   3560   3332 R  16.2   0.2   0:20.08 infinite_half1.sh
 5152 root      20   0   22612   3484   3260 R  16.7   0.2   0:19.57 infinite_half2.sh
```

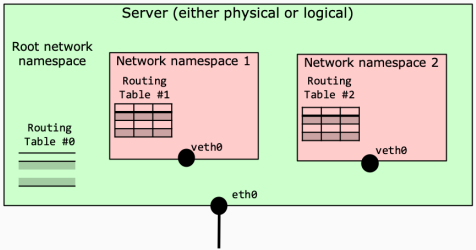


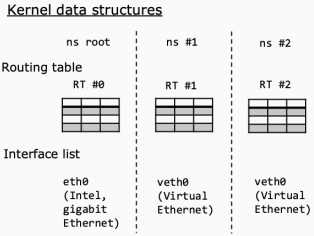
Linux namespaces

- Feature of the Linux kernel, technically not part of cgroups, but highly related
  - Process groups separated to not "see" resources of a given class in other groups
  - Initial release in 2002, kernel 2.4.19, with "mount" namespaces and growing ever since
- Create distinct virtual environments e.g., networking, file system, and more
  - e.g., two namespaces have completely independent networking stacks, e.g., virtual interfaces, IP addresses associated to it, etc.
  - e.g., two namespaces have visibility on completely independent file systems, such as different the /etc folder, etc.
- Currently defined seven different *types* of namespaces (see later)
  - Within each *type* of namespace, we can create N different namespaces, e.g. within the network namespace, there can be namespace root, ns1, ns2, ...

Example: kernel partitioning among namespaces

- Kernel has to create different independent instances of the same data structure
  - E.g., routing table for the network namespace
- Each object (e.g., process) is associated to a given namespace and can access to objects belonging to the same namespace
  - When the access to a given data structure is requested, the kernel uses the "namespaceid" to retrieve data from the proper structure





Linux namespaces: available types

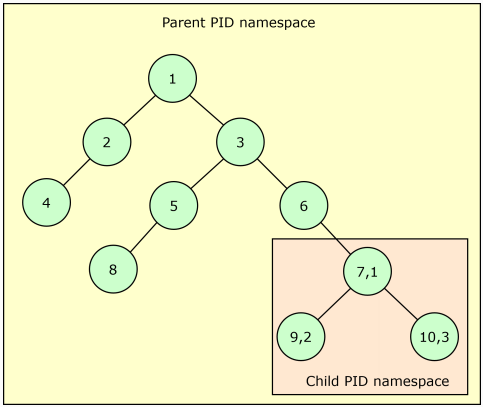
Namespace	Constant	Isolates
IPC	CLONE_NEWIPC	System V IPC, POSIX message queues
Network	CLONE_NEWNET	Network devices, stacks, ports, etc.
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain name
Cgroup	CLONE_NEWCGROUP	Control groups

Linux namespaces: process namespace i

- In Linux, processes originate a single process tree
  - Each process has a parent... till init(1)
  - Processes may have children (when fork() is called)
- A process with enough privileges and under some conditions can inspect another process by attaching a tracer to it or may kill/suspend it
- PID namespace enable multiple "nested" process trees
  - Each process tree represent an entirely isolated set of processes
  - Processes belonging to a given process tree do not even know the existence of other parallel process trees, hence cannot inspect or kill processes in other process trees

Linux namespaces: process namespace ii

- PID namespace allows a process to create a new tree, with its own PID 1 process
  - The first process remains in the original tree and knows about its child; in fact, processes in the parent namespace have a complete view of processes in the child namespace, as if they were any other process in the parent namespace
  - However, child becomes the root of its own process tree and it does not know anything about the originating process tree



Linux namespaces: process namespace iii

- A single process can now have multiple PIDs associated to it, one for each namespace it falls under

https://elixir.bootlin.com/linux/latest/source/include/linux/pid.h

```
/*
 * struct upid is used to get the id of the struct pid, as it is
 * seen in particular namespace. Later the struct pid is found with
 * find_pid_ns() using the int nr and struct pid_namespace *ns.
 */

struct upid {
          int nr;                    //PID value
          struct pid_namespace *ns;  //Namespace where this is relevant
};

struct pid
{
          refcount_t count;
          unsigned int level;
          /* lists of tasks that use this pid */
          struct hlist_head tasks[PIDTYPE_MAX];
          /* wait queue for pidfd notifications */
          wait_queue_head_t wait_pidfd;
          struct rcu_head rcu;
          struct upid numbers[1];  //Array of uPIDs
};
```

Linux namespaces: process namespace iv

Source code

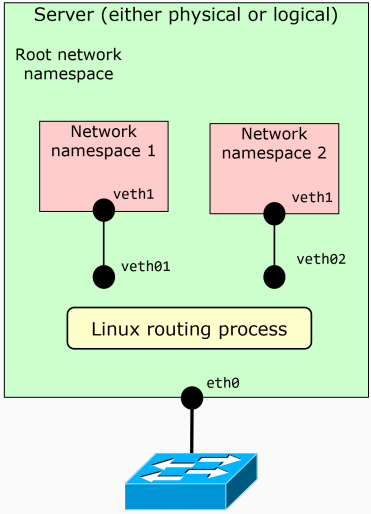
```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

static char child_stack[1048576];

static int child_fn(void *arg) {
  printf("Child PID: %ld\n", (long) getpid());
  printf("Parent PID: %ld\n", (long) getppid());
  sleep(1000);
  return 0;
}

int main() {
  pid_t child_pid = clone(child_fn, child_stack + sizeof(child_stack),
            CLONE_NEWPID | SIGCHLD, NULL);
  printf("clone() = %ld\n", (long) child_pid);
  printf("Main PID: %ld\n", (long) getpid());

  waitpid(child_pid, NULL, 0);
  return 0;
}
```

Output example

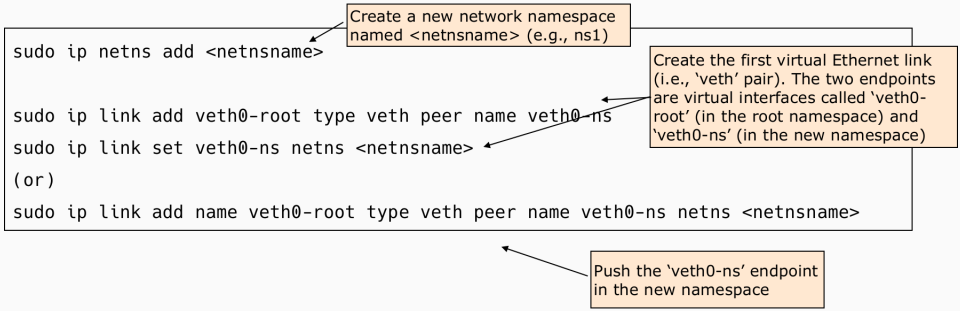
```
netlab@VM:~/sample/bin$ sudo ./sample
clone() = 10381
Main PID: 10380
Child PID: 1
Parent PID: 0


netlab@VM:~/sample/bin$ ps ax |grep sample
10379 pts/1    S+     0:00 sudo ./sample
10380 pts/1    S+     0:00 ./sample
10381 pts/1    S+     0:00 ./sample
```

Linux namespaces: network namespace i

- Network namespace allows two processes to perceive a completely different network setup
  - Interfaces, routing table, firewalling rules, etc.
  - Even the loopback interface is different
- Once a network namespace is created, we should create additional "virtual" network interfaces which span multiple namespaces
  - Virtual interfaces ("veth") are a network abstraction of a wire with two ends, in different namespaces
  - Veth allow traffic to cross the namespace borders and be delivered to another namespace
  - Bridging/routing process in root namespace enable traffic delivery to destination



Linux namespaces: network namespace ii



Linux namespaces: network namespace iii

- These commands establish a pipe-like connection between these two namespaces
  - Parent namespace retains the veth0-root device, and passes the veth0-ns device to the child namespace
  - Anything that enters one of the ends, comes out through the other end, just as we expect from a real Ethernet connection between two real nodes
  - Both sides of this virtual Ethernet connection can be provided with IP addresses
- Names are arbitrary; names can be the same in different namespaces (e.g., two veth0 in two different namespaces)

Linux namespaces: network namespace iv

- All network commands are still available in network namespaces, although with visibility limited to the namespace resources
- Relevant command

```
[sudo] ip netns exec <namespace_name> <command>
```

- Example

```
sudo ip netns exec ns1 tcpdump
```

Linux namespaces: network namespace v

```
netlab@VM:~$ sudo ip netns add ns1

netlab@VM:~$ sudo ip link add name veth0 type veth peer name veth1 netns ns1

netlab@VM:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:c0:81:04 brd ff:ff:ff:ff:ff:ff
3: veth0@if2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 6e:ce:ab:3d:19:9a brd ff:ff:ff:ff:ff:ff link-netns ns1

netlab@VM:~$ sudo ip netns exec ns1 ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: veth1@if6: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 9a:a4:a6:77:fb:94 brd ff:ff:ff:ff:ff:ff link-netnsid 0

netlab@VM:~$ sudo ip link add name veth2 type veth peer name veth3

netlab@VM:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:c0:81:04 brd ff:ff:ff:ff:ff:ff
3: veth0@if2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 6e:ce:ab:3d:19:9a brd ff:ff:ff:ff:ff:ff link-netns ns1
4: veth3@veth2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 92:c7:93:32:db:21 brd ff:ff:ff:ff:ff:ff
5: veth2@veth3: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether c2:29:ab:43:85:07 brd ff:ff:ff:ff:ff:ff
```

The veth is created with one end in the *root* namespace, the other in namespace *ns1*

The veth is created entirely in the *root* namespace, hence both ends are here. We need another command "sudo ip link set veth3 netns ns1" to push it in namespace

Linux namespaces: network namespace vi

Source code

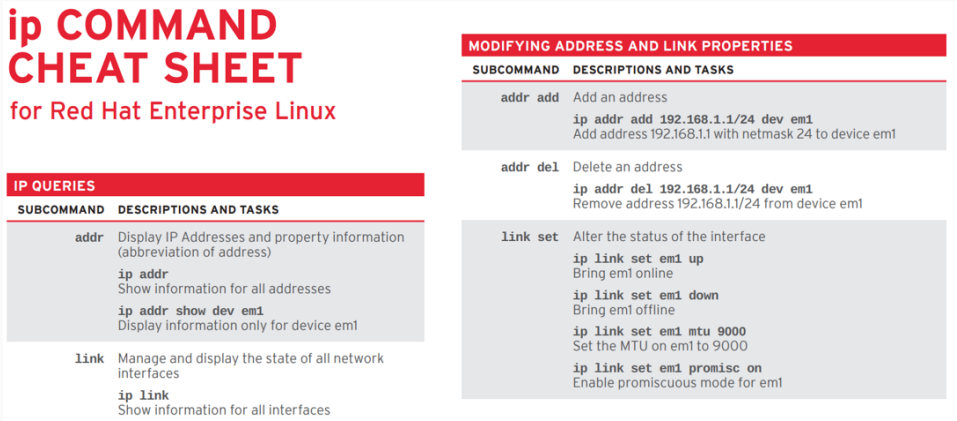
```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

static char child_stack[1048576];

static int child_fn(void *arg) {
  printf("\nNew 'net' namespace:\n");
  system("ip link");
  return 0;
}

int main() {
  printf("Original 'net' namespace:\n");
  system("ip link");

  /* We could create a process using the new PID namespace here */
  pid_t child_pid = clone(child_fn, child_stack + sizeof(child_stack),
                /*CLONE_NEWPID |*/ CLONE_NEWNET | SIGCHLD, NULL);

  waitpid(child_pid, NULL, 0);
  return 0;
}
```

Output example

```
netlab@VM:~/sample/bin$ sudo ./sample
Original 'net' namespace:
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel
state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:c0:81:04 brd ff:ff:ff:ff:ff:ff

New 'net' namespace:
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Note that the loopback is ‘down’ and no other network interfaces are present

Reference: Linux IP commands cheat sheet



ip COMMAND CHEAT SHEET

for Red Hat Enterprise Linux

IP QUERIES

SUBCOMMAND	DESCRIPTIONS AND TASKS
`addr`	Display IP Addresses and property information (abbreviation of address) **`ip addr`** Show information for all addresses **`ip addr show dev em1`** Display information only for device em1
`link`	Manage and display the state of all network interfaces **`ip link`** Show information for all interfaces

MODIFYING ADDRESS AND LINK PROPERTIES

SUBCOMMAND	DESCRIPTIONS AND TASKS
`addr add`	Add an address **`ip addr add 192.168.1.1/24 dev em1`** Add address 192.168.1.1 with netmask 24 to device em1
`addr del`	Delete an address **`ip addr del 192.168.1.1/24 dev em1`** Remove address 192.168.1.1/24 from device em1
`link set`	Alter the status of the interface **`ip link set em1 up`** Bring em1 online **`ip link set em1 down`** Bring em1 offline **`ip link set em1 mtu 9000`** Set the MTU on em1 to 9000 **`ip link set em1 promisc on`** Enable promiscuous mode for em1

https://access.redhat.com/sites/default/files/attachments/rh_ip_command_cheatsheet_1214_jcs_print.pdf

Other namespaces in short i

- Mount namespace enables the creation of a completely new file system, with the desired structure
  - E.g., disk partitions (and permissions, e.g., read-only), mount points, folders
  - Similar to chroot (empty root to populate), but more flexible (cloned data structure)
- UTS namespace is a very simple namespace that provides isolation of two system identifiers: the hostname and the NIS domain name
- User namespace allows a process to have root privileges within the namespace, without giving it that access to processes outside of the namespace

Other namespaces in short i

- IPC namespace creates private inter-process communication resources (e.g., System V IPC, POSIX messages) for the isolated process
- cgroup namespace provides a mechanism to virtualize the view of the "/proc/$PID/cgroup" file and cgroup mounts

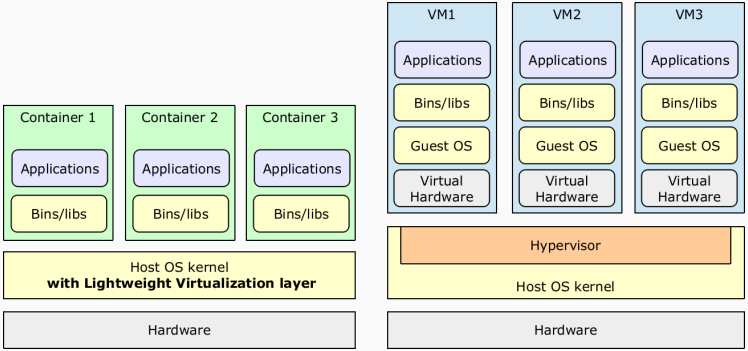
Beyond process isolation: limitation of cgroups and namespaces

- cgroups and namespaces show some limitations beyond process isolation
- In fact, they provide an answer to the "virtualization" on a single server
  - Cannot be used, as is, to handle an entire datacenter (need to be integrated with CMS)
- Flexible (each feature, e.g., namespace can be tuned) but difficult to use
  - A lot of commands/scripting required to turn a full container on
- Cannot guarantee application portability, such as in case of VMs
  - VMs can be packaged and started on any server
  - Not an easy way to "package" an isolated app and make it running on another server
- Cgroups and Namespaces should be extended to handle a datacenter at scale

Linux containers: overview

- Containers provide lightweight virtualisation that allows to isolate processes and resources without the complexities of full virtualisation
- Linux container is an OS-level virtualisation method for running multiple isolated Linux systems (containers) on a single control host
    - The Linux kernel is shared across all containers

Containers vs. Hypervisors



Containers: what they do...

- They group processes together inside isolated containers (to which different resources can be assigned)
- They share the same operating system as the host
- Inside the box they look like a VM (or better, a machine)
- Outside the box they look like normal processes (in the host, I see processes of all containers)

Containers: . . . and what they don't

- They do not emulate hardware
- They do not run different kernels or OSs
- Security is not an out-of-the-box feature and it must not be taken for granted
  - still an appropriate level of security can be achieved

Containers vs VMs i

- Containers are:
  - Faster than real VM (to boot, freeze, dispose and to orchestrate)
  - Lighter than VM: less CPU, less memory, no virtualisation overhead (e.g., instruction emulation)
    - Denser than VM: due to the inferior resource consumption
    - Container virtualisation technology can practically achieve the same performances of native execution
- VMs
  - Provide better isolation (e.g., protect also from kernel exploits)
  - Better security, due to the limited points of attacks (the hypervisor is usually very tiny)
  - Enable the usage of different OSs

Containers vs VMs ii

- Depending on your view, containers are at the same time:
  - Infrastructure primitives (aka “lightweight VMs”)
  - Application management and configuration systems
- An Infrastructure engineer will see them as the former; a developer will see them as the latter

Containers: characteristics

- Agile application creation and deployment
  - Increased ease and efficiency of container image creation compared to VM images
- Cloud and OS distribution portability
  - Run on Ubuntu, RHEL, CoreOS, and anywhere else
- Application-centric management
  - abstraction: from OS on virtual hardware to application on OS using logical resources
- Resource isolation
  - Predictable application performance
- Resource utilization
  - High efficiency and density

Pet vs cattle vs chicken vs insect

- Pet vs cattle analogy was invented to show difference between (individual, unique) servers and datacenter, now extended

Unique servers, Virtual Machines

Replaceable servers and VMs

Even more dense and lightweight containers

Extremely short-lived functions (serverless)

Russ McKendrick, "The pets and cattle analogy demonstrates how serverless fits into the software infrastructure landscape." Available at https://hub.packtpub.com/pets-cattle-analogy-demonstrates-how-serverless-fits-software-infrastructure-landscape/

- LXC = cgroups + namespaces (+ some other stuff)
  - No kernel patches required, vanilla kernel is ok
- Collection of kernel features that can be used to isolate processes in different ways and a userspace tool to use all of these features to create full-fledged containers
- Elementary features are still usable on their own, without LXC

LXC features

- Kernel namespaces (ipc, uts, mount, pid, network and user)
- Chroots
- Control groups (cgroups)
- Kernel capabilities
- Apparmor and SELinux profiles
- Seccomp policies

Why LXC?

- Because it is easier than each single elementary component
- For instance
  - Cgroups provides only resource management
  - If we want also isolation, we need to add also namespaces
  - If we want also security, we need to add Apparmor and/or SELinux
  - But what about also live migration, etc. . .
- LXC does not provide everything, but it provides several features
  - LXC allows to configure each container with the list of features it needs, specified in its configuration file
  - We can assign different resources to different containers
  - We can have different levels of isolation between different containers (and the host)

LXC limitations

- Checkpointing and migration
  - Missing features in vanilla linux kernel, we need to user other tools (CRIU), which are not yet 100% working
- Resource isolation
  - We must configure containers not to "see" resources
  - Not always guaranteed; e.g., the resource quota of a container could be affected by the behavior of others
- Do not guarantee portability
  - A LXC created on a first server cannot be ported to other servers, while VMs are instead portable (at least across servers that have the same hypervisor)


docker
Build, Ship, Run, Any App Anywhere
From Dev
To Ops
Any App
ASP.NET
.NET
Java
MORE
docker
Any OS
Windows
Linux

Docker focuses on applications

- So far, developers experience an hard way to package applications for deployment
  - Package your app in multiple formats (e.g., .debian, redhat, etc)
  - Create multiple packages for each OS version (e.g., Ubuntu 19.10, Ubuntu 19.04, etc), as dependencies may be different and required libraries may be present in different versions
  - Be ready to create new packages as soon as a new OS flavor / version is released
  - Still, users may complain that your app does not start because of some mistakes in the packaging process
- Not even considering that there are different OSs out there (Windows, MacOS, etc)
- Docker focuses on applications, simplifying their deployment and execution by creating a lightweight, portable, self-contained "package" that runs anywhere
  - Similar to what "intermodal shipping containers" did in the area of goods transportation (ships, trucks, etc.)

Docker in a nutshell

- Objectives
  - Clean separation of environments (of different processes)
  - Sandbox with defined resources (e.g., through cgroups and namespaces)
  - Unified environment to handle applications (e.g. "run" command for all apps)
  - Transparent and seamless networking (Docker bridge, DHCP, transparent NAT, etc)
- What Docker is not
  - Not a virtualisation engine (leverages existing primitives such as cgroups and namespaces) and no support for different kernels
  - Does not leverage hardware primitives (e.g., CPU extensions)
- Docker focuses on applications, simplifying their deployment and execution

Docker deploys apps reliably and consistently

- If an app works locally, it works on the server (and in any other place)
  - Run side-by-side containers with their own versions of dependencies
- Your app runs everywhere, with the exact behavior
  - Regardless of the kernel version
  - Regardless of the host distro
  - Physical or virtual, cloud or not

Separation of concerns

Developer (inside the container)

- App
- Libraries and dependencies
- "Manifest" (e.g., used TCP/UDP ports)
  - Optional and not always embedded in the container
- [Code]
- [Data]

DevOps (outside the container)

- Logging
- Remote access
- Network configuration
- Monitoring

Docker vs LXC

- Docker is optimized for the deployment of applications, as opposed to machines
  - LXC focuses on containers as lightweight machines, basically servers that boot faster and need less RAM
- Docker containers are portable across machines, while LXC need to be rebuilt on the target machine
  - A Docker can run "anywhere" (although CPU architecture must match) with just one build, and is isolated from the host
- Docker has with a simplified (command line) interface and more powerful management tools
- Docker supports resource management and isolation, versioning, component re-use, automatic build and sharing

Docker: additional features compared to LXC

- Application portability
- Union file system
- Automatic build
- Focus on running applications
- Versioning
- Component re-use
- Sharing (Docker server)
- Better documentation and ecosystem
- Integration with OpenStack, Kubernetes and all cloud vendors

Docker Image

- An immutable template for containers
- Can be pulled and pushed towards a registry
- Image names have the form [registry/] [user/] name [:tag]

Docker Container

- An instance of an image
- Can be started, stopped, restarted, ...
- Maintains changes within the filesystems
- New image can be created from current container state (although not recommended, use Dockerfile instead)
  - The default for the tag is *latest*

Docker Registry

- Entity similar to a software repository, that keeps the available Docker Images
- Can be either public (e.g., Docker Hub, `https://hub.docker.com/`) or private
  - Images can be private (e.g., associated to a given user/group), even on a public repository
- Some commands:
  - Searching in the registry:
    - `docker search <term>`
  - Download or update an image from the registry (and cache locally):
    - `docker pull <image>`
  - Upload an image to the registry (keep it private or make it public):
    - `docker push <image>`

Docker Images (locally)

- "Pulled" images are cached locally and can be executed
- Some commands:
  - List downloaded images:
    - `docker images`
  - Delete a local image:
    - `docker rmi <image>` (or)
    - `docker image rm <image>`
  - Run an image
    - `docker run [options] <image>`

Docker run: very common options

- When running a container, two options may be very useful:
  - Expose a TCP/UDP port of the container on the host
  - Mount a folder on the host (e.g., with persistent data) in the container file system (consider also network shares)
- Those are simple options of the docker run command
  - Publish port 80 from container nginx as port 8080 on host:
    - `docker run -p 8080:80 nginx`
  - Mount local directory /html as directory /usr/share/nginx/html in the container nginx:
    - `docker run -v /html:/usr/share/nginx/html mynginx`
    - Note 1: /usr/share/nginx/html is where nginx expects HTML files
    - Note 2: "Mount" just means "make available"

Docker containers: some common commands

- Show running containers:
  - `docker ps`
- Show all containers (shows also terminated containers):
  - `docker ps -a`
- Show metadata of a container
  - `docker inspect <container>`
    - Returns JSON with all the info about the container (e.g. current IP/MAC address, log path, image name, etc)
  - docker inspect –format='{{.Image}}' <container> (to show a specific metadata)
    - Shows only a specific metadata from the above JSON

Commands to handle container lifecycle

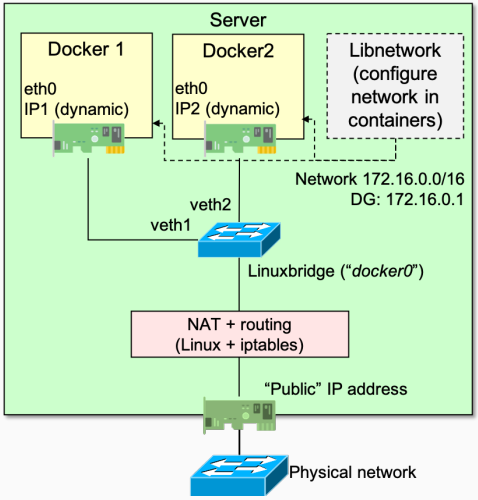
- Control your containers:
  - `docker run <image..>` (start a new container from an image)
  - `docker start <container..>` (start stopped container)
  - `docker stop <container..>` (stop gracefully a container with SIGTERM signal)
  - `docker kill <container..>` (kill running container by sending a SIGKILL signal)
  - `docker rm <container..>`
    - Remove container from `docker ps -a` list
    - State of container is lost, preventing user to save it to another container
    - `docker start` does not work on a "removed" container
    - Does not remove image, `docker run` still works

Commands for interactions and debugging

- Run a command in an existing container, e.g start a shell:
  - `docker exec <container> <command>`
  - `docker exec -it <container> bash`
- See the logs (stdout) of the container:
  - `docker logs -f <container>`
- Copy files from and to Docker container:
  - `docker cp <source> <destination>`
  - `docker cp my_webserver:/etc/nginx/nginx.conf ~/`

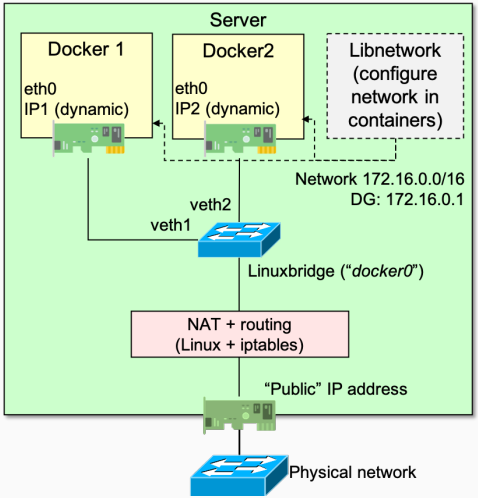
Docker network i

- By default, all containers are attached to a docker0 bridge (linuxbridge)
- Docker automatically implements a private network, with its IP address space and the appropriate routing table + iptables rules to enable Internet connectivity (outbound)
- Inbound connectivity is automatically provided to all connections started *inside* and if a port of the container is exposed outside



Docker network ii

- Network operations are carried out through the "libnetwork" component, which implements the Container Network Model (CNM)
- This component decides the IP address to use within the container, and configures the inside endpoint of the veth with the above address
  - Address assignment does not use DHCP

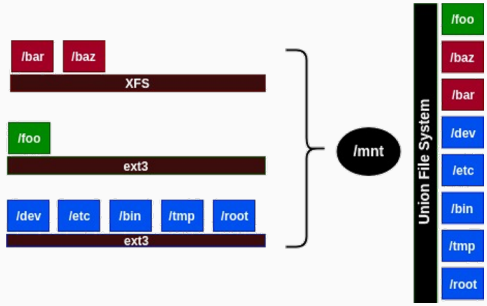


File system in containers

- Containers must replicate entire file system required for them to run
  - Isolation requirement: prevents container to access some files already on the host
  - Portability requirement: container should containing all files required to run
- This means that the size of a container (even minimal) can be rather huge, i.e., in the order to 100MB or more
  - Remember: only OS kernel is shared with the host; all the rest is duplicated!
- Starting a process (i.e., container) already present on the host may require a few milliseconds
- Transferring the image from a remote location can require several seconds (100MB is ~1s on a 1Gbps network) => non-negligible delay in the container startup time

Union File System

- Union file system works on top of the other file-systems
- It gives a single coherent and unified view to files and directories of separate file-system
- In other words, it mounts multiple directories to a single root
  - It is more of a mounting mechanism than a file system
- UnionFS, AUFS, OverlayFS are some examples of the union file system
  - Docker, by default, uses OverlayFS



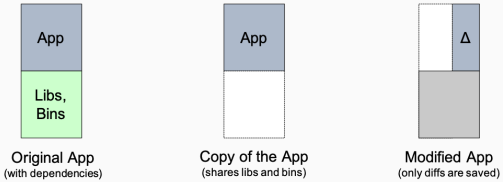
https://blog.knoldus.com/unionfs-a-file-system-of-a-container/

Properties of a Union File System

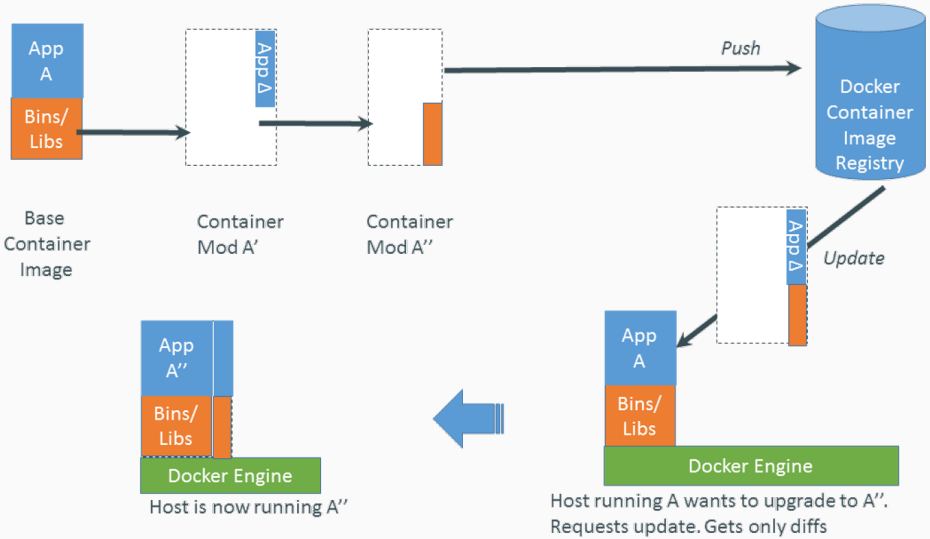
- Logical merge of multiple layers
- Read-only lower layers, writable upper layer
- Start reading from the upper layer then defaults to lower layers
- Copy on Write (CoW)
  - If process wants to modify existing data, OS copies data *only for that process* to use (all other processes continue to use the original data)
- Simulate removal from lower directory through whiteout file
  - File removed from the union mount directory would directly remove file from "upper" directory, simulate removal from "lower" directory by creating "whiteout" file
  - This file exists only in "union" directory, without physically appearing in either the "upper" or "lower" directories

Union File Systems: advantages

- Only differences are stored
  - E.g., in case an application is modified, we do not need to copy also shared bin/libs
  - E.g., two apps can be created using same set of base "layers" (only apps are copied)
- Reduces
  - Disk footprint on the target host
  - Loading time when launching containers (each individual layer can be cached)

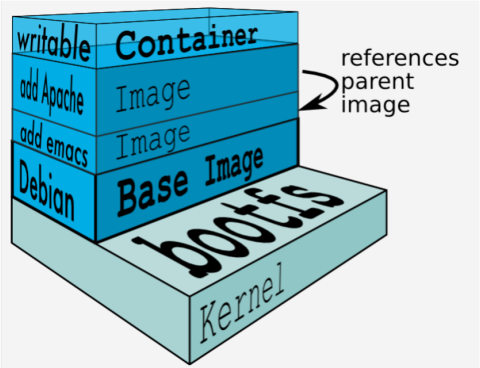


Upgrading a container



Docker file system

- Docker uses a layered file system
  - Default is OverlalyFS, through storage driver "overlay2"
- A complex image can result from the composition of different layers
  - Each layer can be either read-only or read/write
- Many containers can share the same set of base images



Automatic build

- Docker allow developers to automatically create a container starting from its composing elements
  - E.g., a container can be created by compiling the application source code from scratch
  - E.g., another container can be created by using the package manager of Linux distro
- The recipe used to create the container is stored in a special file, called "Dockerfile"
- Developers are free to use make, maven, chef, puppet, salt, debian packages, rpms, source tarballs, bash scripts, or any combination of the above, regardless of the configuration of the machines
- Note: if we start the build process with a strictly reduced base image (e.g., stripped down OS image), we can create an image that has only the software we need, avoiding the "default" software that is always installed in a vanilla copy of the operating system

Dockerfile example i

```
###############################################################
# Dockerfile to build MongoDB container images, based on Ubuntu

# Set the base image to Ubuntu, with a specific tag (1910)
FROM Ubuntu:1910

# File Author / Maintainer
MAINTAINER Example McAuthor

# Update the repository sources list. Not strictly needed, but good practice
RUN apt-get update

################## BEGIN INSTALLATION ######################
# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

# Add MongoDB to the Ubuntu default repository sources list
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' |
tee /etc/apt/sources.list.d/mongodb.list
```

https://www.digitalocean.com/community/tutorials/docker-explained-using-dockerfiles-to-automate-building-of-images

Dockerfile example ii

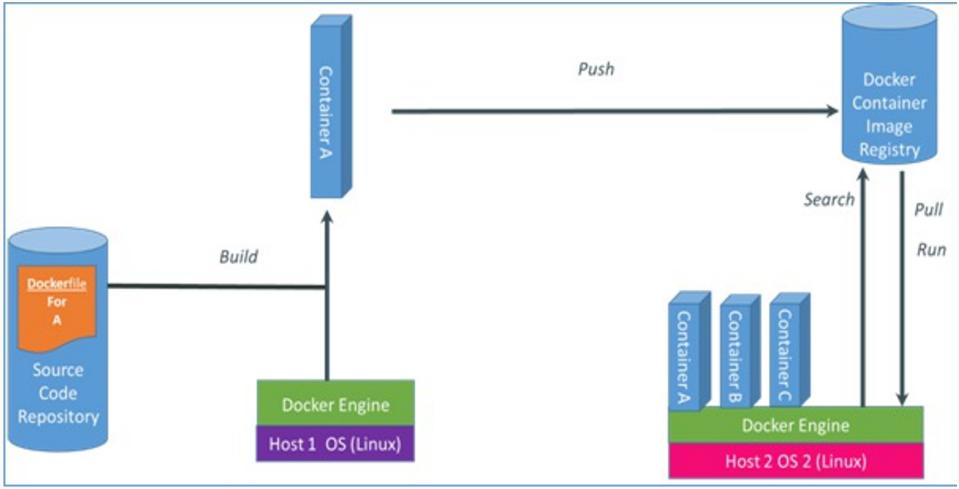
```
# Update the repository sources list once more
RUN apt-get update

# Install MongoDB package (.deb)
RUN apt-get install -y mongodb-10gen

# Create the default data directory
RUN mkdir -p /data/db

##################### INSTALLATION END #####################

# Expose the default port
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

Now we have to build the image and give it a name:

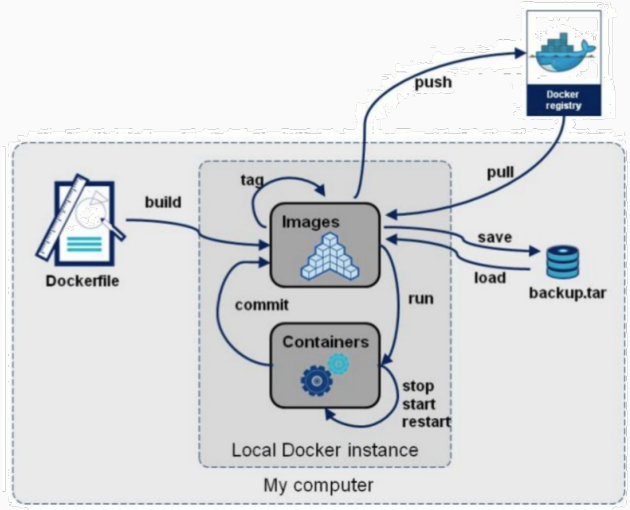
```
docker build --pull -t mymongo .
```

`--pull`: Pull a newer version of the image
`-t mymongo`: name the new image “mymongo”
`.`: look for the “Dockerfile” in the current folder

Docker components and deployment workflow



Docker lifecycle



Resource consumption: some examples

- Memory and CPU
  - A container with a tiny "hello world" program takes about 670 KB of additional RAM compared to the same program running on a vanilla OS
  - The additional CPU consumption of the same program running in a container or on the base host is negligible
- Disk
  - A container for the Ubuntu 14.04 LTS may take up 250MB on disk
  - Installing openjdk-7-jre on adds about 140MB

Why is Docker used (and appreciated)

- Continuous delivery
  - Deliver software more often and with less errors
  - No time spent on dev-to-ops handoffs
- Improved security
  - Containers help isolate components of your system and provides control over them
- Run anything, anywhere
  - All languages, all databases, all operating systems
  - Any distribution, any cloud, any machine
- Reproducibility
  - Reduces the times we say “it worked on my machine”

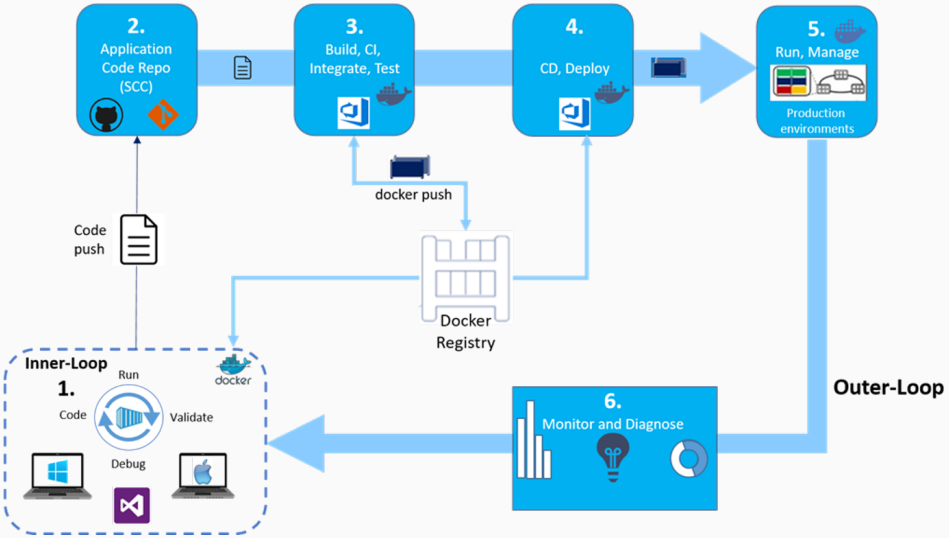
Orchestration

- DockerSwarm: orchestrator for Docker
  - It turns a pool of Docker hosts into a virtual, single host, handling the resources of the entire datacenter
  - Loosing attraction in favor of Kubernetes
- Kubernetes
  - Part of the PaaS lab

Docker Compose

- Compose is a tool for defining and running multi-container Docker applications
- A specific YAML file is used to configure the application individual services
  - E.g., web frontend, database engine, storage
- Compose enables to create and start all the services from the same configuration, with a single command
- Works better if coupled with Docker Swarm
  - Also in this case, its functions are often carried out with Kubernetes

The overall DevOps loop



OS-level (lightweight) virtualisation: summary

- Lightweight virtualisation is definitely important and it is increasingly used in big enterprises
  - More efficient use of computing resources
  - Reduced operating costs (e.g., update of each individual kernel in different VMs)
  - Enable Multiple processes to coexist, with (strong) isolation properties
  - Several cloud companies are already offering commercial services
- Docker is a well-established technology for container virtualisation
  - Although is it more appreciated because has revolutionized the way software is packaged than for lightweight virtualisation