CLOUD COMPUTING

Compute Virtualization - Types of Hypervisors

Dr. H.L. Phalachandra

Department of Computer Science and Engineering

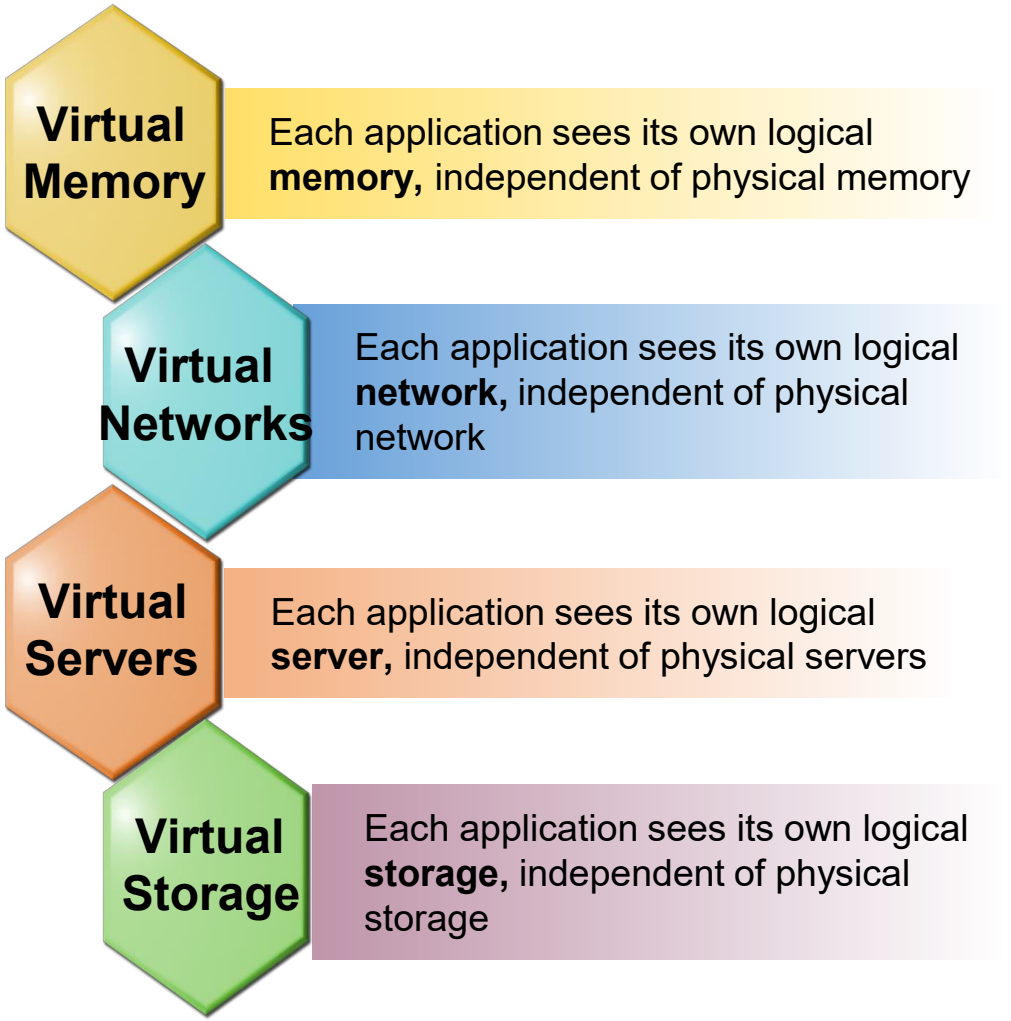
Acknowledgements:

Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

CLOUD COMPUTING

What is Virtualization?

- Virtualization is a technique of abstracting physical resources (this could be compute, memory, storage disks, networking) in to a logical view
- Virtualization increases utilization and capability of the IT infrastructure
- Simplifies resource management by pooling and sharing resources
- Significantly reduces downtime both Planned and unplanned
- Improves performance of IT infrastructure



CLOUD COMPUTING

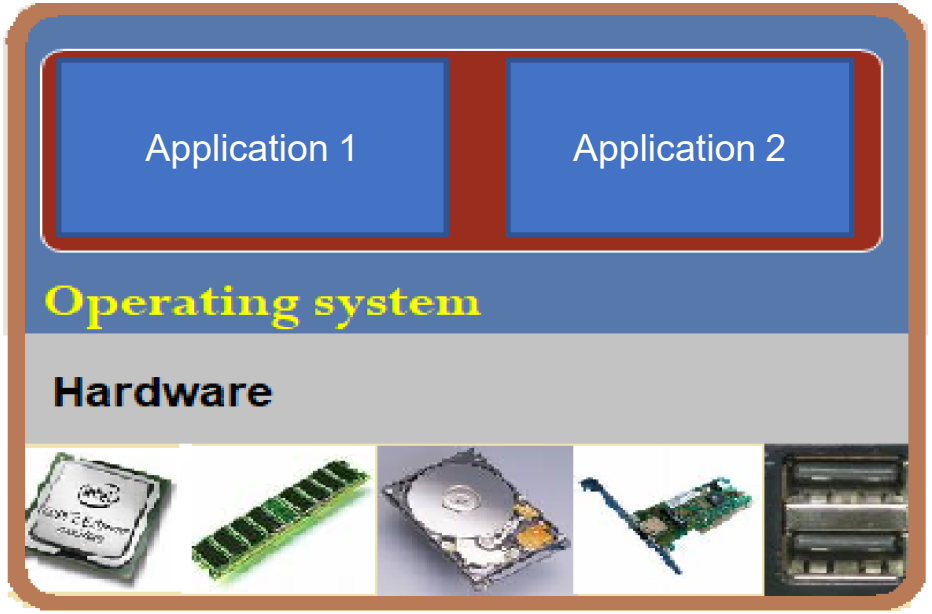
Compute Virtualization : Its relationship to Cloud Computing

- Compute (sometimes also known as Server) virtualization :
  - Can be considered as a framework or methodology of dividing the resources of a computer/server into multiple execution environments
  - Its achieved by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others.
  - Can be looked at as a practice of presenting and partitioning computing resources in a logical way rather than partitioning according to physical reality
- VM (or a Virtual Machine) is the coarse granular view of the Virtual Compute at the next abstract level, and is logically identical to a Physical machine with an ability to run an OS
- Virtualization enables sharing of the same physical resources across different users and is a key technology for Cloud Computing
- VM which uses this virtualization technology is one of the prominent ways in which compute resources are provisioned in a cloud environment

CLOUD COMPUTING

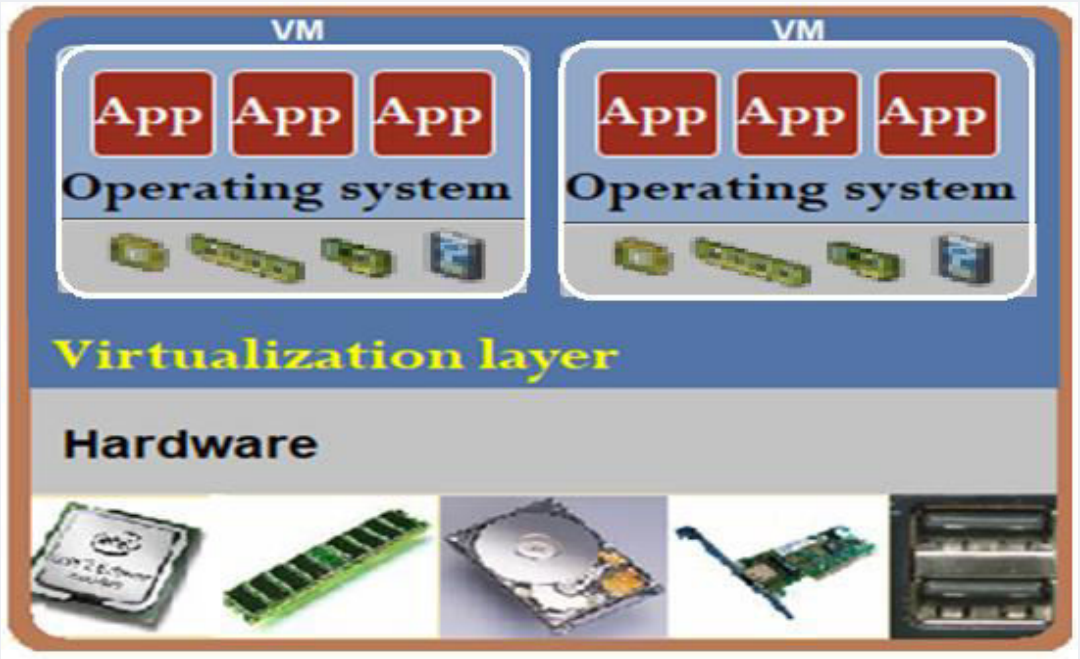
Traditional computer without virtualization (bare metal) & After Virtualization

Before Server Virtualization:



- Single operating system image per machine
- Software and hardware tightly couple
- OS handles the bare hardware (CPU, memory, and I/O) directly.
- Running multiple applications on same machine can create conflicts
- Underutilized resources

After Server Virtualization:



- Virtual Machines (VMs) break dependencies between operating system and hardware
- Manages operating system and application as single unit by encapsulating them into VMs
- Strong fault and security isolation
- Hardware-independent

CLOUD COMPUTING

Compute Virtualization : Why Virtualization

• Server consolidation	• Increase hardware utilization by running multiple applications in isolation on same physical server
• Workload mobility	• Move applications from one server to another
• Development and test	• Easily provision virtual resources for test and dev
• Business Continuity Management	• Ability granularly live-migrate a VM to other physical servers which are part of a Business Continuity environment
• Support OS diversity	• Can run both Linux and Windows on same h/w
• Security/Isolation	• Hypervisor separates the VMs from each other and isolates VMs from H/W
• High Availability/Load Balancing	• Ability to live-migrate a VM to other physical server
• Rapid provisioning	• On demand provisioning of hardware resources
• Encapsulation	• The execution environment of an application is encapsulated within VM
• Lower Costs	• Increased Efficiencies, flexibility and responsiveness

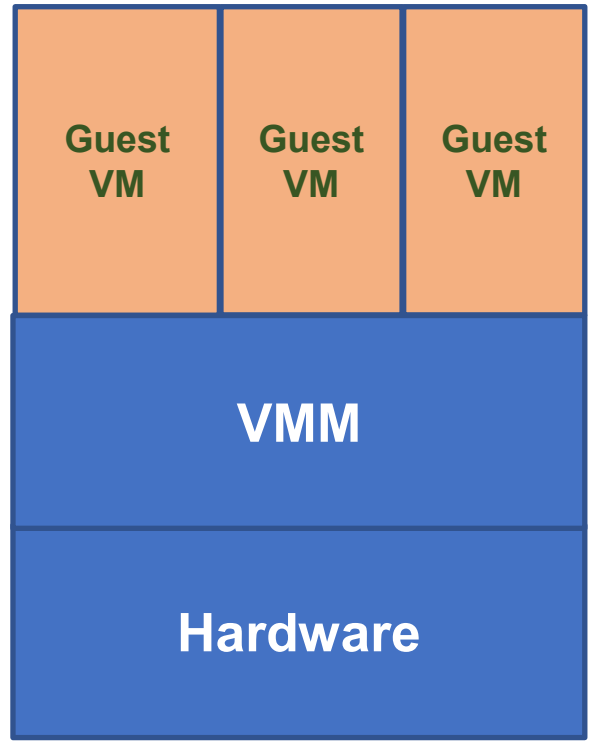
CLOUD COMPUTING

Compute Virtualization : How is this implemented?

Typically, a layer of software that provides the illusion of a "real" machine to multiple instances of "*virtual machines* is traditionally called a *Virtual Machine Monitor (VMM)* or a "*Hypervisor*"

- **Hardware-level virtualization** inserts the VMM (or the virtualization layer) between real hardware and traditional operating systems. This VMM directly interacts with the Hardware and thus is also called **Bare Metal VMM or Hypervisor of Type 1**.
- VMM manages the hardware resources of a computing system. Each time programs access the hardware, the VMM captures the process
- VMM acts as a traditional OS & leading to three requirements for a VMM:
  - VMM should provide an environment for programs which is essentially identical to the original machine.
  - Programs running in this environment should show at worst, only minor reduction in performance.
  - VMM should be in complete control of the system resources.

E.g. Xen, VMWare ESX server, IBM CP/CMS, Windows Virtualization (2008)



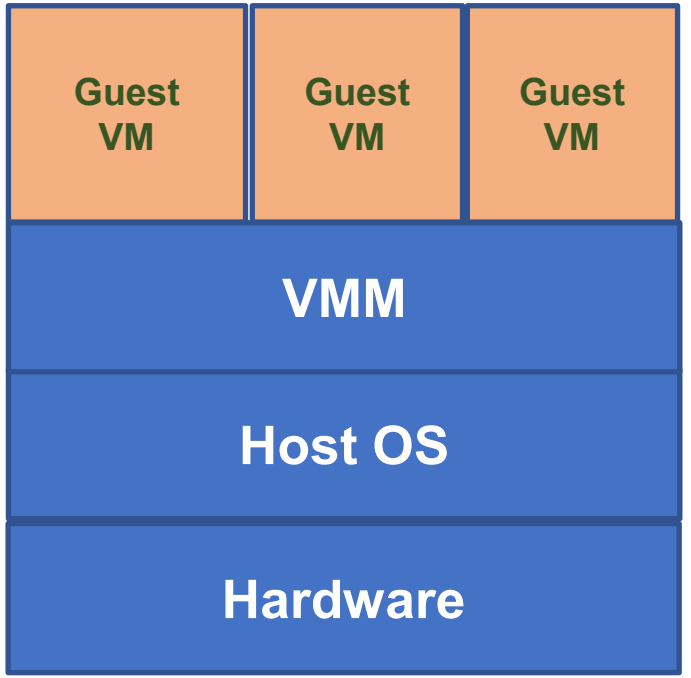
CLOUD COMPUTING

Compute Virtualization : How is this implemented? - II

- This VMM when implemented on top of the OS as in the figure would typically be called as **Hosted VMM** or **Hosted Hypervisor or Hypervisor** of **Type 2**
- VMM maintains a software level representation of physical hardware and interposes on the operations and redirects it as per the policies

  E.g. VMWare workstation or Oracle VirtualBox
- VMM could also be implemented as part of the OS too.

  E.g. KVM (Kernel Based Virtual Machine) for Linux

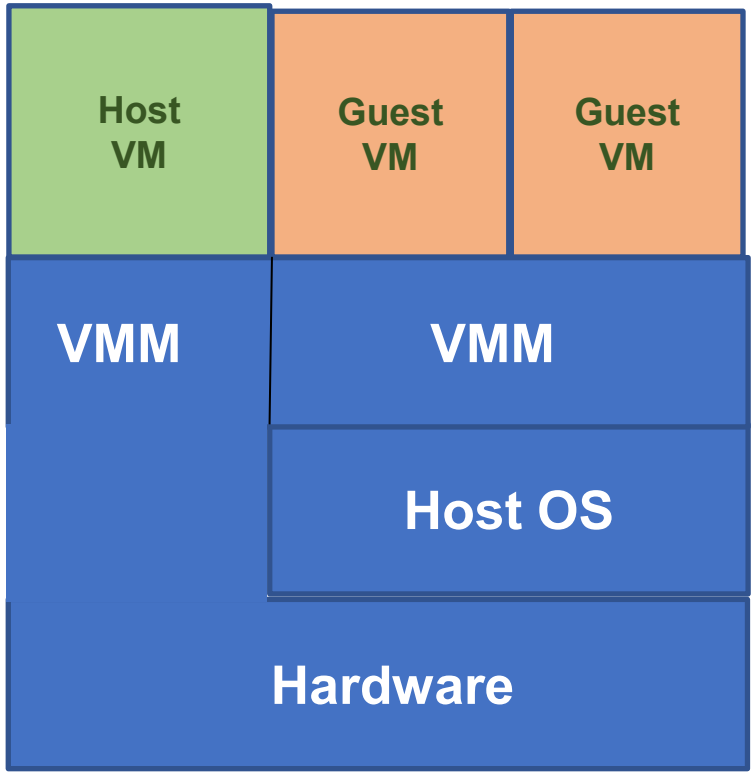


CLOUD COMPUTING

Compute Virtualization : How is this implemented? - III

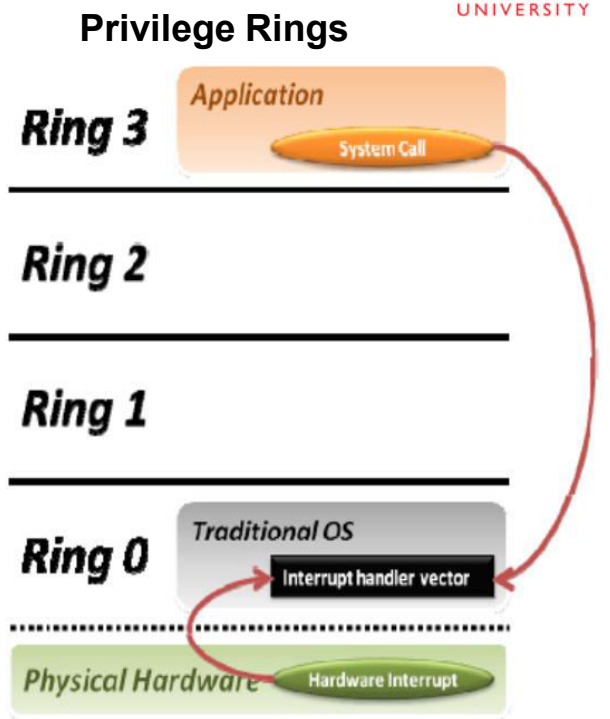
- VMM implementation when supporting both the bare-metal and as a hosted application on top of the OS as in the figure would typically be called as **Hybrid VMM** or **Hybrid Hypervisor**

  E.g. MS Virtual Server, MS Virtual PC



CLOUD COMPUTING

Physical Machine (setting the context from the OS last Sem)



Physical machines have real hardware resources like CPU, memory, I/O etc.

These are managed/orchestrated into a single execution environment by the OS by having

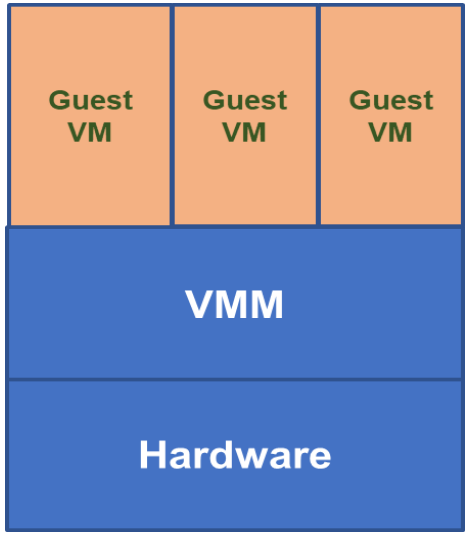
- Exclusive access to these hardware resources through some hardware interfaces
- Uses instructions defined by the ISA (ISA-instruction set architectures) for the specific CPU architecture eg. X86
- Memory
- I/O

OSes provide dual modes, with the privileged or Kernel mode **(Ring 0)** having access to the physical resources (with operations like I/O instructions and Halt instructions, Turn off all Interrupts, Set the Timer, Context Switching, Clear the Memory or Remove a process from the Memory) and a non-privileged mode (or also know as **Ring 3 or User mode**, where safe instructions (like Load, store instructions Add, Subtract etc.) are supported

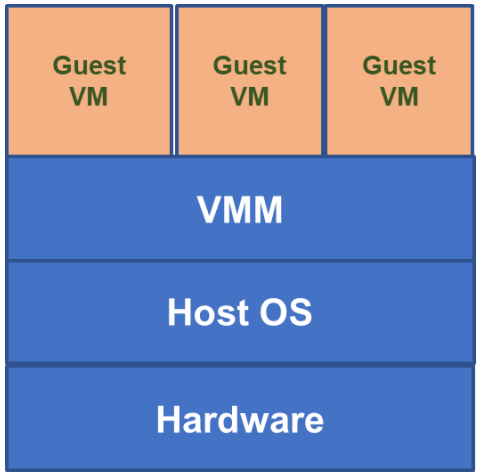
The User mode can do some of the IO and other actions only by invoking system calls. These system calls when executed, first sets mode bit to 0 (supervisory mode) and then executes the call. On completion of the action and before exiting from the I/O system call, it sets the bit to one (user mode) and transfers control back to user process. This approach is used as a means of not giving the supervisory control to any user process.

Hypervisor Type 1 vs. Type 2

Criteria	Type 1 hypervisor	Type 2 hypervisor
AKA	Bare-metal or Native	Hosted
Definition	Runs directly on the system with VMs running on them	Runs on a conventional Operating System
Virtualization	Hardware Virtualization	OS Virtualization
Operation	Guest OS and applications run on the hypervisor	Runs as an application on the host OS
Scalability	Better Scalability	Not so much, because of its reliance on the underlying OS.
Setup/Installation	Simple, as long as you have the necessary hardware support	Lot simpler setup, as you already have an Operating System.
System Independence	Has direct access to hardware along with virtual machines it hosts	Are not allowed to directly access the host hardware and its resources
Speed	Faster	Slower because of the system's dependency
Performance	Higher-performance as there's no middle layer	Comparatively has reduced performance rate as it runs with extra overhead
Security	More Secure	Less Secure, as any problem in the base operating system affects the entire system including the protected Hypervisor
Examples	• VMware ESXi • Microsoft Hyper-V • Citrix XenServer	• VMware Workstation Player • Microsoft Virtual PC • Sun's VirtualBox



Type 1



Type 2

Additional references

What is a Hypervisor?
https://www.vmware.com/topics/glossary/content/hypervisor
https://youtu.be/EvXn2QiL3gs
https://www.vmware.com/topics/glossary/content/bare-metal-hypervisor
Types of hypervisor
https://www.youtube.com/watch?v=dckSc61WS6w
Bare metal vs hypervisor
https://blog.servermania.com/bare-metal-vs-hypervisor-which-is-right-for-your-project/

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

CLOUD COMPUTING

Paravirtualization and Transparent Virtualization

Prof. Venkatesh Prasad
Dr. H.L. Phalachandra

Department of Computer Science and Engineering

Acknowledgements:
Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

Other types of Virtualization

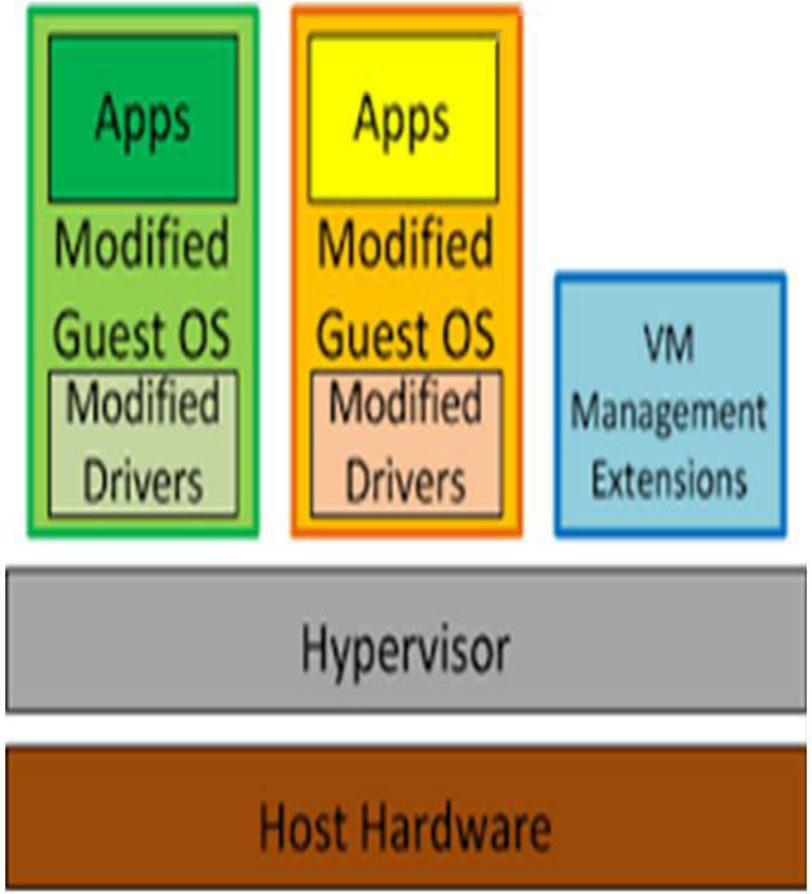
Paravirtualization

It's a virtualization technique that presents an software interface to VMs that is similar but not identical to that of the underlying hardware

- OS to be explicitly ported to run on top of the VMM/Hypervisor
  - Xen
- VMM provides APIs for guest OS
- Useful if source code of OS is modifiable
  - IBM: MVS, VM
  - Linux
  - Microsoft: Windows

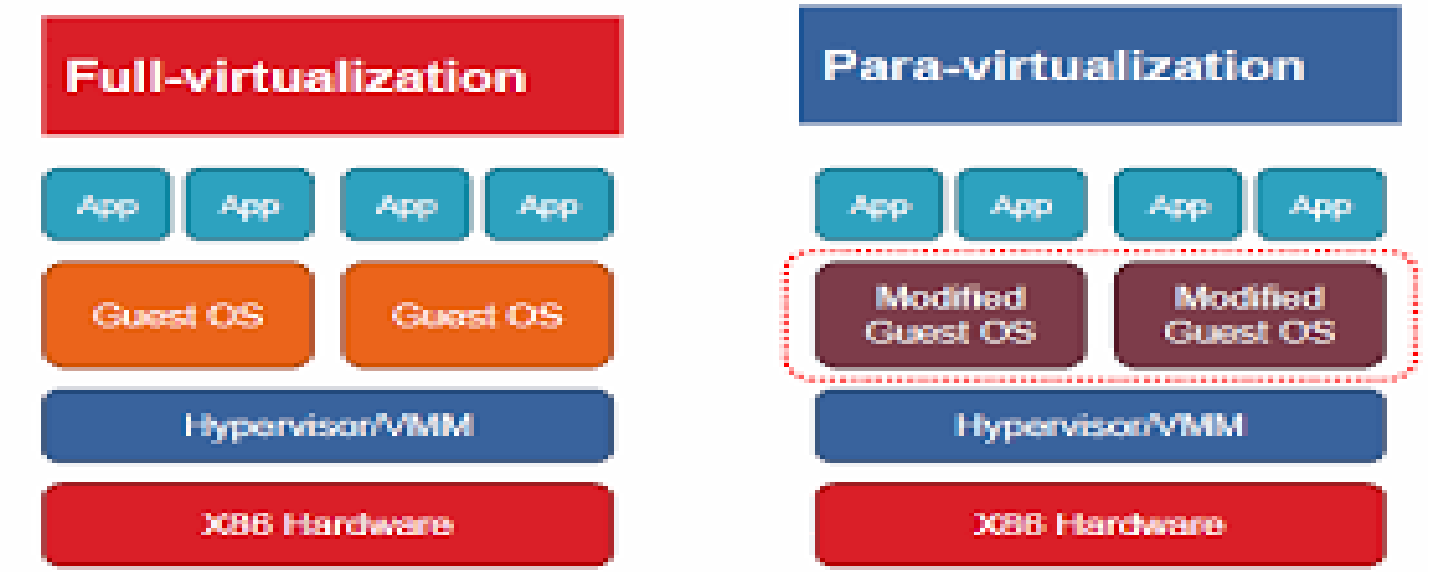
Full (**transparent**) virtualization

- Provides complete simulation of the underlying hardware
- OS runs without modification
  - VMWare
  - kvm



CLOUD COMPUTING

Full vs Para Virtualization



- Full virtualization architecture intercepts and emulates privileged and sensitive instructions at runtime
- Para-virtualization handles these instructions at compile time.
  - The guest OS kernel is modified to replace privileged and sensitive instructions with hypercalls to the hypervisor or VMM.
  - Xen assumes such a para-virtualization architecture.

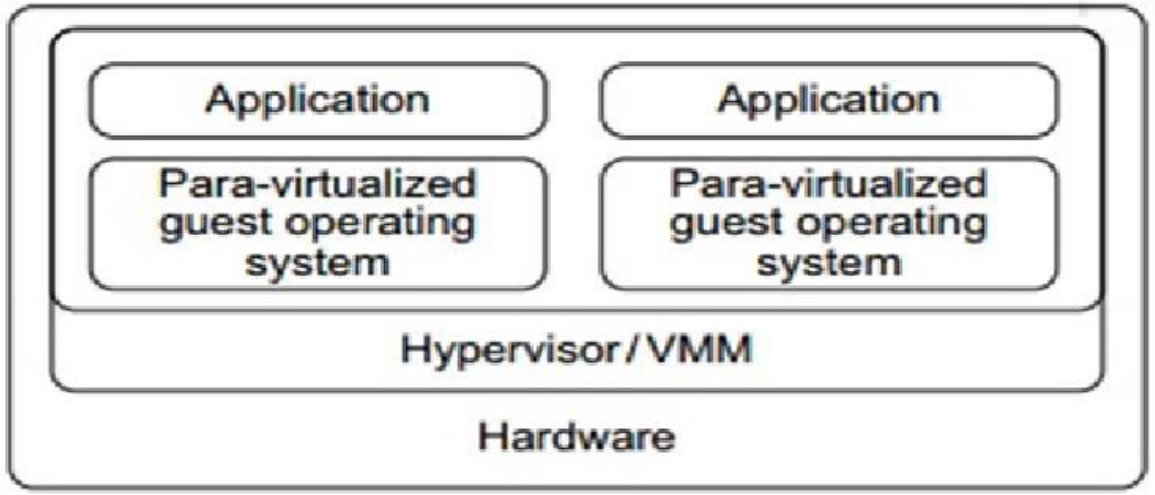
CLOUD COMPUTING

Paravirtualization Overview

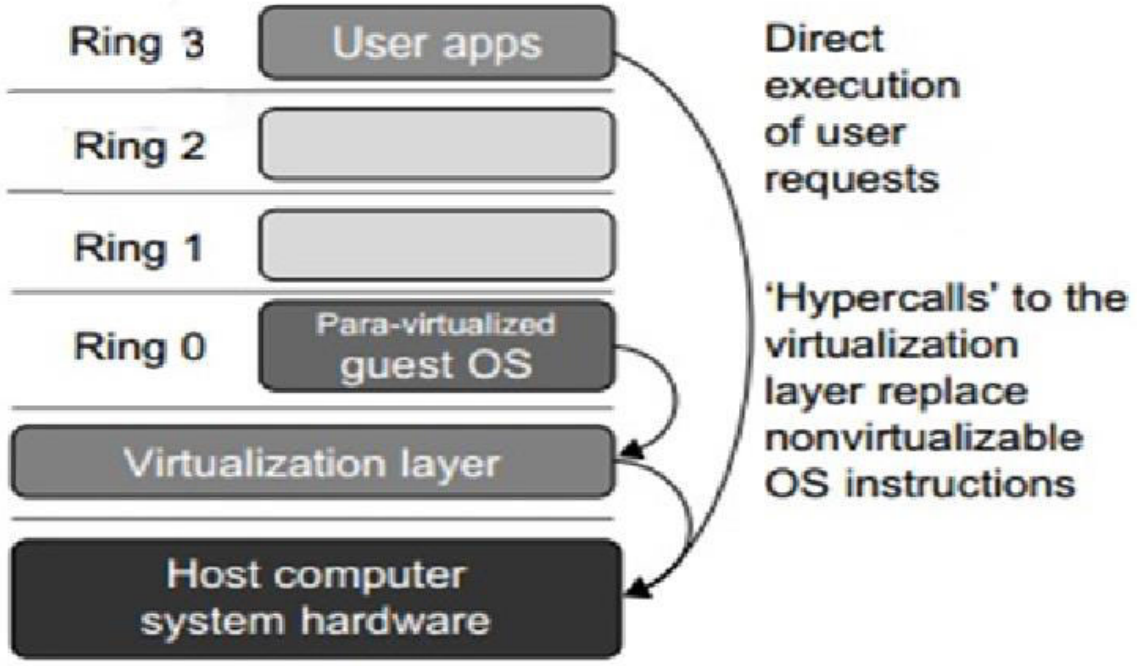
- Paravirtualization needs to modify the guest OS.
  - Modify guest so that it interfaces better with hypervisor
  - Leverage hypervisor APIs (hypercall)
  - Special I/O APIs vs emulating hardware I/O
  - For improved performance
- A para-virtualized VM provides special APIs requiring substantial OS modifications in user applications
- Performance degradation is a critical issue of a virtualized system.
- No one wants to use a VM if it is much slower than using a physical machine.
- Para-virtualization attempts to reduce the virtualization overhead, and thus improve performance by modifying only the guest OS kernel.

CLOUD COMPUTING

Paravirtualization Overview (Cont.)



Para-virtualized VM architecture, which involves modifying the guest OS kernel to replace nonvirtualizable instructions with hypercalls for the hypervisor or the VMM to carry out the virtualization process



The use of a para-virtualized guest OS assisted by an intelligent compiler to replace nonvirtualizable OS instructions by hypercalls.

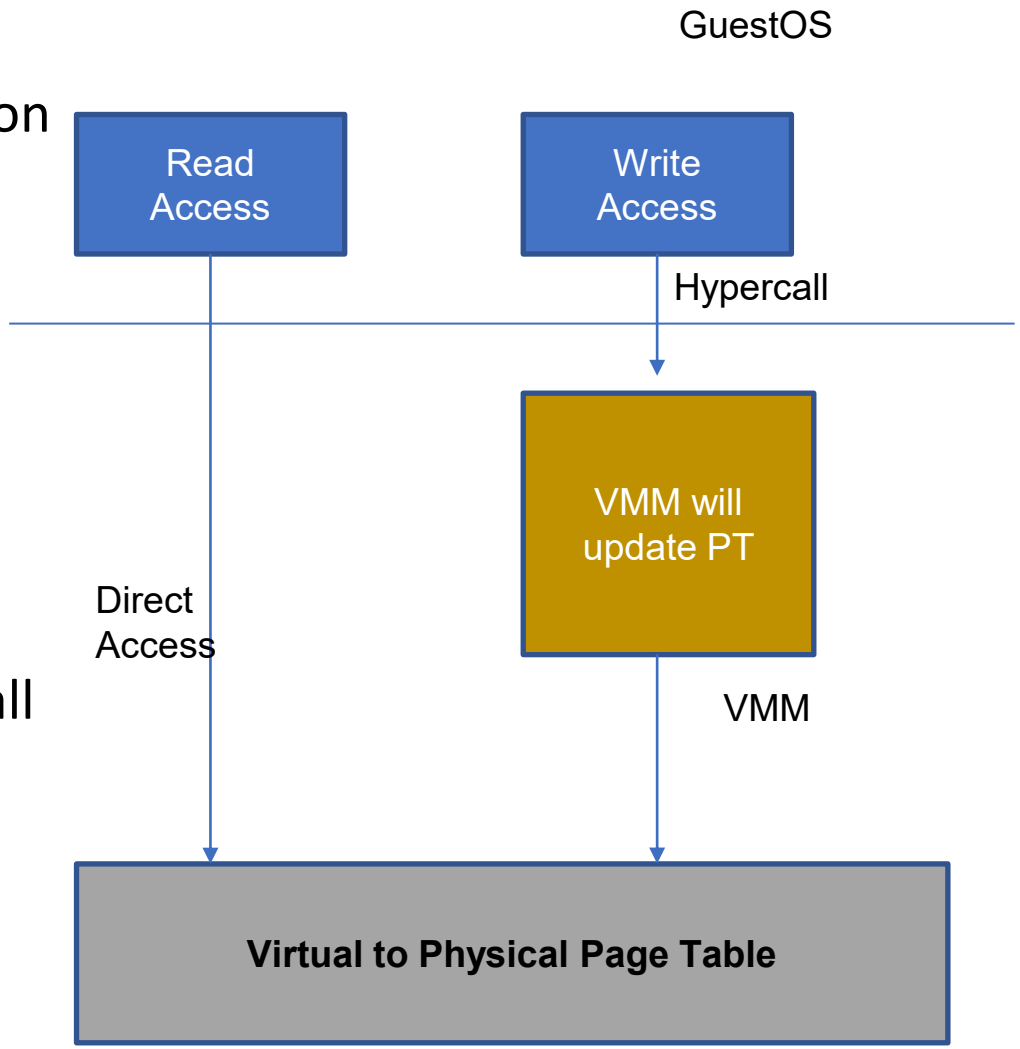
Paravirtualization Overview (Cont.)

- The guest operating systems are para-virtualized.
- They are assisted by an intelligent compiler to replace the nonvirtualizable OS instructions by hypercalls
- The traditional x86 processor offers four instruction execution rings: Rings 0, 1, 2, and 3. The lower the ring number, the higher the privilege of instruction being executed.
- The OS is responsible for managing the hardware and the privileged
- instructions to execute at Ring 0, while user-level applications run at Ring 3.

CLOUD COMPUTING

Paravirtualization Example

- Xen writeable page tables
  - Guest OS has enabled writeable page tables option
  - Guest OS tries to write page tables
  - Xen makes that page table page temporarily writeable
    - But removes the page from the page table
  - Re-starts guest
  - Guest can continue to modify page table, then call a Xen routine
  - Xen then updates the real page table



Paravirtualization Example (Cont.)

- Suppose Guest OS wants to start Oracle; Oracle needs 1GB of virtual memory
- Page table stores mapping of virtual memory to physical memory
  - Assuming 4K pages, there are 1GB/4K = 250,000 page table entries
- Under pure trap and emulate virtualization, there will be a trap for each entry
  - 250,000 traps

Paravirtualization Example (Cont.)

- Since we need a page table with 250,000 entries, the page table itself is stored in memory
  - Suppose we need 4MB for the page table
  - This is 4MB/4K = 1000 pages
- With Xen writeable page table support
  - The first time the guest OS tries to write a page in the page table, there will be a trap
  - Xen will remove page from page table
  - After that, the guest OS can modify entries in page as needed
  - When completed; guest OS will call Xen API
  - Xen will validate page table entries and put page back in page table
- 1000 traps vs 250,000 traps

Paravirtualization Issues

- First, its *compatibility and portability* may be in doubt, because it must support the un-modified OS as well.
- Second, the *cost of maintaining para-virtualized OSes is high*, because they may require deep OS kernel modifications.
- Finally, the *performance* advantage of para-virtualization varies greatly due to workload variations.
- Compared with full virtualization, para-virtualization is relatively easy and more practical.
- The main problem in full virtualization is its *low performance in binary translation*. To speed up binary translation is difficult. Therefore, many virtualization products employ the para-virtualization architecture. The popular Xen, KVM, and VMware ESX are good examples.

KVM

- This is a Linux para-virtualization system—a part of the Linux version 2.6.20 kernel.
- Memory management and scheduling activities are carried out by the existing Linux kernel. The KVM does the rest, which makes it simpler than the hypervisor that controls the entire machine.
- KVM is a *hardware-assisted para-virtualization tool*, which improves performance and supports unmodified guest OSes such as Windows, Linux, Solaris, and other UNIX variants.

CLOUD COMPUTING

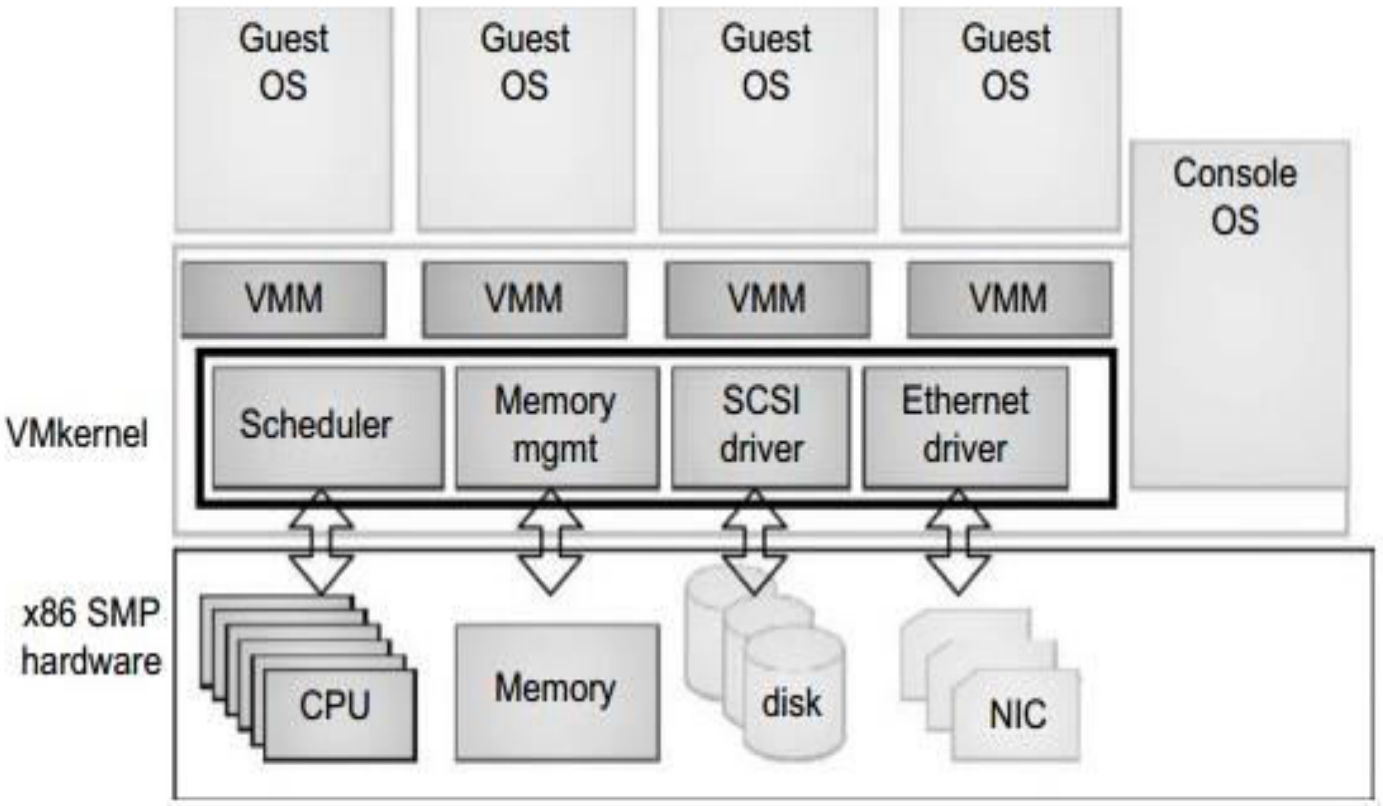
Paravirtualization with Compiler Support

- Unlike the full virtualization architecture which intercepts and emulates privileged and sensitive instructions at runtime, para-virtualization handles these instructions at compile time.
- The guest OS kernel is modified to replace the privileged and sensitive instructions with hypercalls to the hypervisor or VMM.
- Xen assumes such a para-virtualization architecture.
- The guest OS running in a guest domain may run at Ring 1 instead of at Ring 0.
  - This implies that the guest OS may not be able to execute some privileged and sensitive instructions.
- The privileged instructions are implemented by hypercalls to the hypervisor.
- After replacing the instructions with hypercalls, the modified guest OS emulates the behavior of the original guest OS.
- On an UNIX system, a system call involves an interrupt or service routine. The hypercalls apply a dedicated service routine in Xen.

CLOUD COMPUTING

VMware ESX Server architecture using Para-Virtualization

- ESX is a VMM or a hypervisor for bare-metal x86 symmetric multiprocessing (SMP) servers.
- It accesses hardware resources such as I/O directly and has complete resource management control.
- To improve performance, the ESX server employs a para-virtualization architecture in which the VM kernel interacts directly with the hardware without involving the host OS.



An ESX-enabled server consists of four components: a virtualization layer, a resource manager, hardware interface components, and a service console.

CLOUD COMPUTING

VMware ESX Server architecture using Para-Virtualization (Cont.)

- The VMM layer virtualizes the physical hardware resources such as CPU, memory, network and disk controllers, and human interface devices. Every VM has its own set of virtual hardware resources.
- The resource manager allocates CPU, memory disk, and network bandwidth and maps them to the virtual hardware resource set of each VM created.
- Hardware interface components are the device drivers and the VMware ESX Server File System.
- The service console is responsible for booting the system, initiating the execution of the VMM and resource manager, and relinquishing control to those layers. It also facilitates the process for system administrators.

Summary

- Transparent virtualization
  - Trap-and-emulate (to be discussed in detail in the next lecture)
  - Binary translation
- Paravirtualization
- Example: Page tables
  - Transparent: shadow page tables (to be discussed in detail under Memory virtualization)
  - Paravirtualization: special page table APIs (Xen)

Additional References

Para vs Full vs Transparent virtualization
https://www.geeksforgeeks.org/difference-between-full-virtualization-and-paravirtualization/
https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

CLOUD COMPUTING

Virtualization Software Techniques
Trap & emulate, binary translation

Prof. Venkatesh Prasad
Dr. H.L. Phalachandra

Department of Computer Science and Engineering

Acknowledgements:
Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

Direct Execution (Recap)

Idea: Run *most* instructions of the VM directly on the hardware

Advantage: Performance *close* to native execution

Challenge: How to ensure isolation/protection ?

If all instructions of the VM are directly executed then VMM has no control !

Where the original intent for a VMM and VM is to run an unmodified guest OS on the VM as if it were executing directly on the hardware, an alternative is to modify the guest OS replacing hard to virtualize instructions and/or providing more convenient abstractions especially of IO devices. Paravirtualization often results in more efficient code, but of course limits the range of usable guest OSs and the convenience of their use.

Direct Execution + Trap and Emulate (Recap)

Idea:

- Trap to hypervisor when the VM tries to execute an instruction that could change the state of the system/take control (i.e., impact safety/isolation).
- Emulate execution of these instruction in hypervisor
- Direct execution of any other innocuous instructions on h/w that cannot impact other VMs or the hypervisor

CLOUD COMPUTING

Virtualization (Types) - Recap

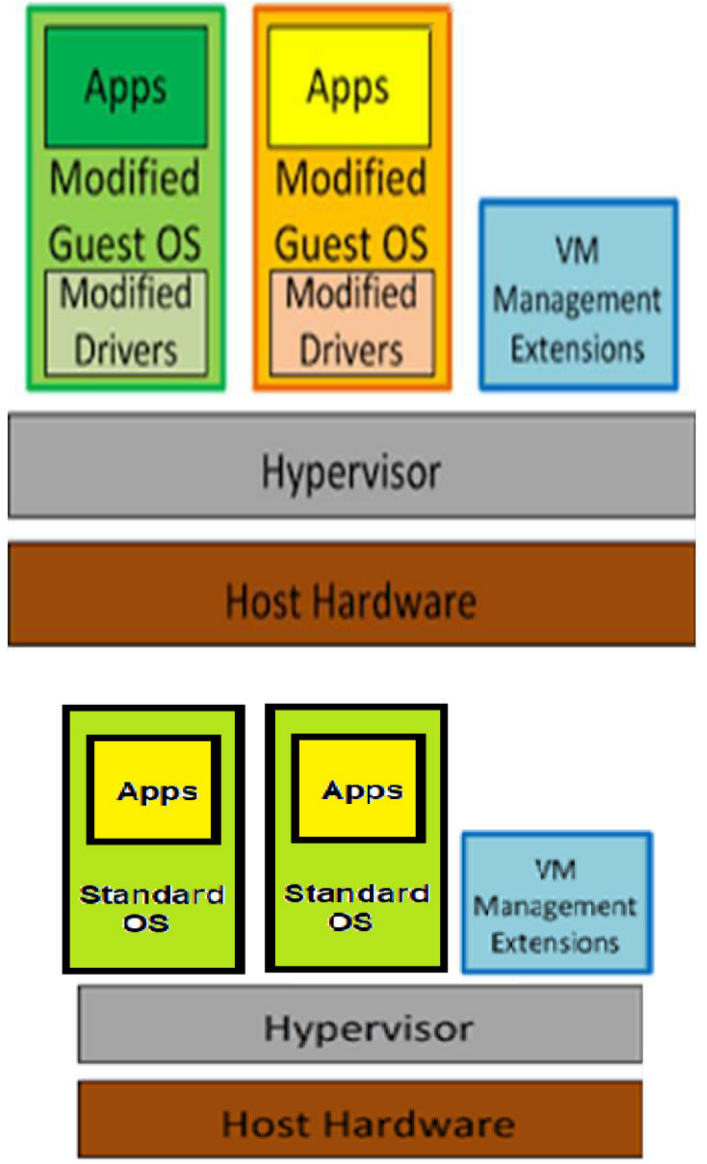
Paravirtualization

It's a virtualization technique that presents an software interface to VMs that is similar but not identical to that of the underlying hardware

- OS to be explicitly ported to run on top of the VMM/Hypervisor
  - Xen
- VMM provides APIs for guest OS
- Useful if source code of OS is modifiable
  - IBM: MVS, VM
  - Linux
  - Microsoft: Windows

Full (**transparent**) virtualization

- Provides complete simulation of the underlying hardware
- OS runs without modification
  - VMWare
  - kvm



How to do Trap and Emulate?

- Generally two categories of instructions(assembly):
  - User instructions:
    - Typically compute instructions
      e.g. *add, mult, ld, store, jmp*
  - System instructions:
    - Typically for system management
      e.g. iret, invlpg, hlt, in, out

invlpg— priviledged instruction to invalidate TLB Entries

How to do Trap and Emulate? (Cont.)

Two modes of CPU operation:

User mode (in x86-64 typically ring 3)

Privileged mode (in x86-64 ring 0)

Attempt to execute system instructions in user mode generates trap/*general protection fault* (gpf)

System state:

Example, **c**ontrol **r**egisters like *cr3*

Access to system state in user mode trigger gpf

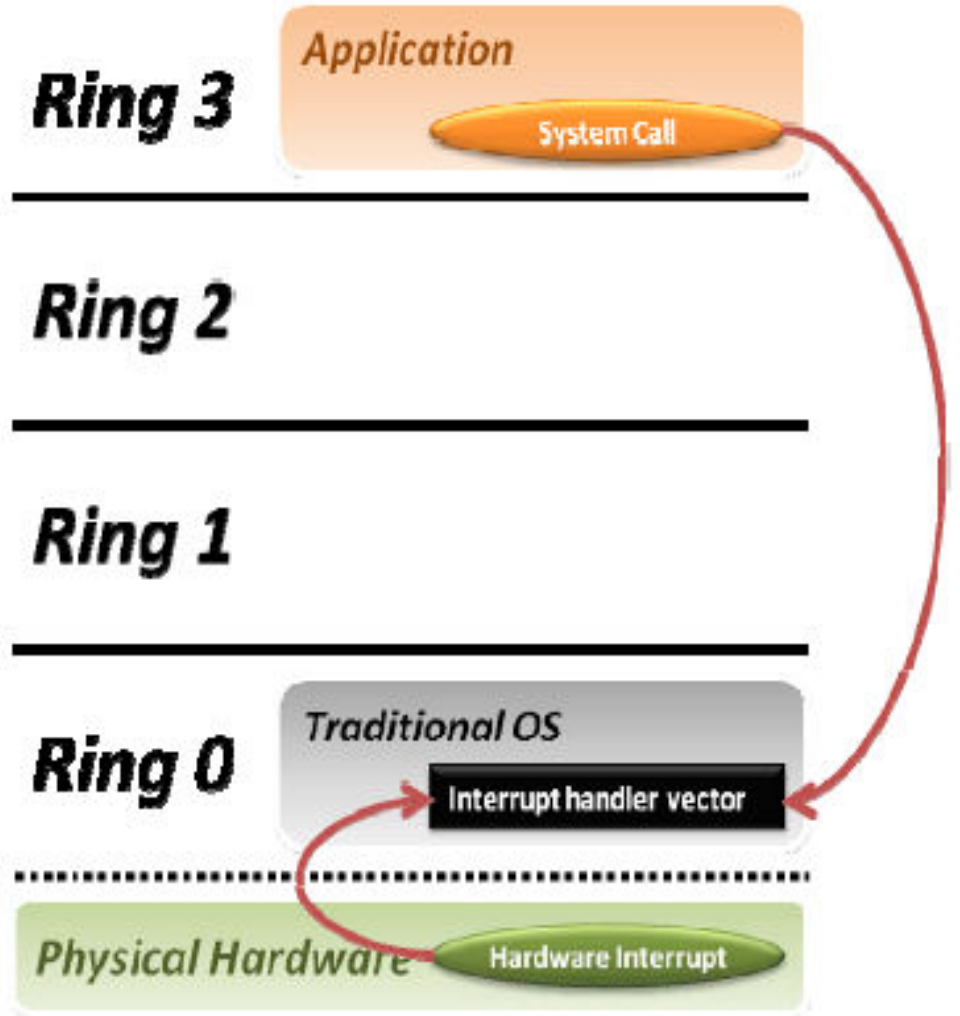
Idea:

Run the VM in user mode (ring 3), while the hypervisor in privileged mode (ring 0)

Anytime VM tries to execute an system instruction or tries to access system state, trap to VMM

Trap and Emulate

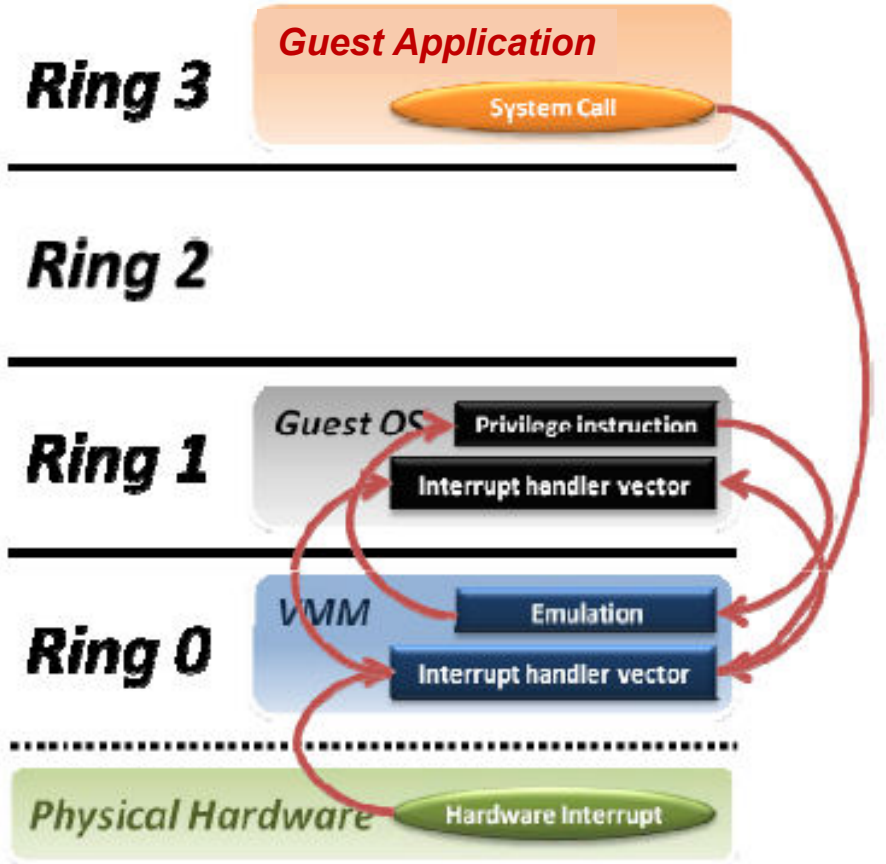
- Privilege Rings in Hardware
  - Ring 0,1,2,3 in x86 CPUs
- Typically OS runs in high privilege mode (Ring 0)
- User applications run in Ring 3 (less privilege)
- But with a VMM (functioning like OS), the
  - VMM runs in ring 0
  - Which ring does Guest OS run in?
- Guest OS will need to run with privileges but not fully privileged like host OS/VMM
  - Guest OS will also need to be protected from guest apps which would be in ring 3
  - Typically would run in Ring 1 and traps to VMM – at 0



Trap and Emulate Contd.

- If Guest application has to handle syscall/interrupt
  - Special trap instr (int n), traps to VMM
  - VMM doesn't know how to handle trap
  - VMM jumps to guest OS trap handler
  - Trap handled by guest OS normally
- Guest OS performs return from trap
  - Privileged instr, traps to VMM
  - VMM jumps to corresponding user process
- Any privileged action by guest OS traps to VMM

  Example: set IDT, set CR3, access hardware
  - emulated by VMM
  - Then go back to the Guest OS



Issues/Problems with Trap and Emulate Virtualization technique

1. Performance Overhead

   E.g. trapping privileged instructions

   - Trap costs may be high

2. Not all architectures support it

   - x86 architecture was easily or not completely virtualizable and needed workarounds
   - Depends on the fact that executing an instruction at a lower privilege than required will cause a **trap** to the VMM.

Recent x86 systems try to facilitate virtualization by dealing with the sensitive instructions that are not privileged and by facilitating memory management with hardware. They have introduced a new mode, root mode. Now the mode will consist of the usual four levels paired with a normal/root mode. All sensitive instructions cause transfer to the root mode.

Issues/Problems with Trap and Emulate Virtualization technique

3. Some x86 instructions which change hardware state (sensitive instructions) run in both privileged and unprivileged modes
   - Will behave differently when guest OS is in ring 0 vs in less privileged ring 1
   - OS behaves incorrectly in ring1, will not trap to VMM

     Example: popf (pop flags)
     - Can be used in user mode to change ALU flags
     - Can be used in privileged mode to change system state flag (e.g., interrupt delivery flag)
     - Trouble: No trap is popf is attempted to alter interrupt flag in user mode --- CPU just ignores it !
       - Behavior sensitive instruction but is not privileged

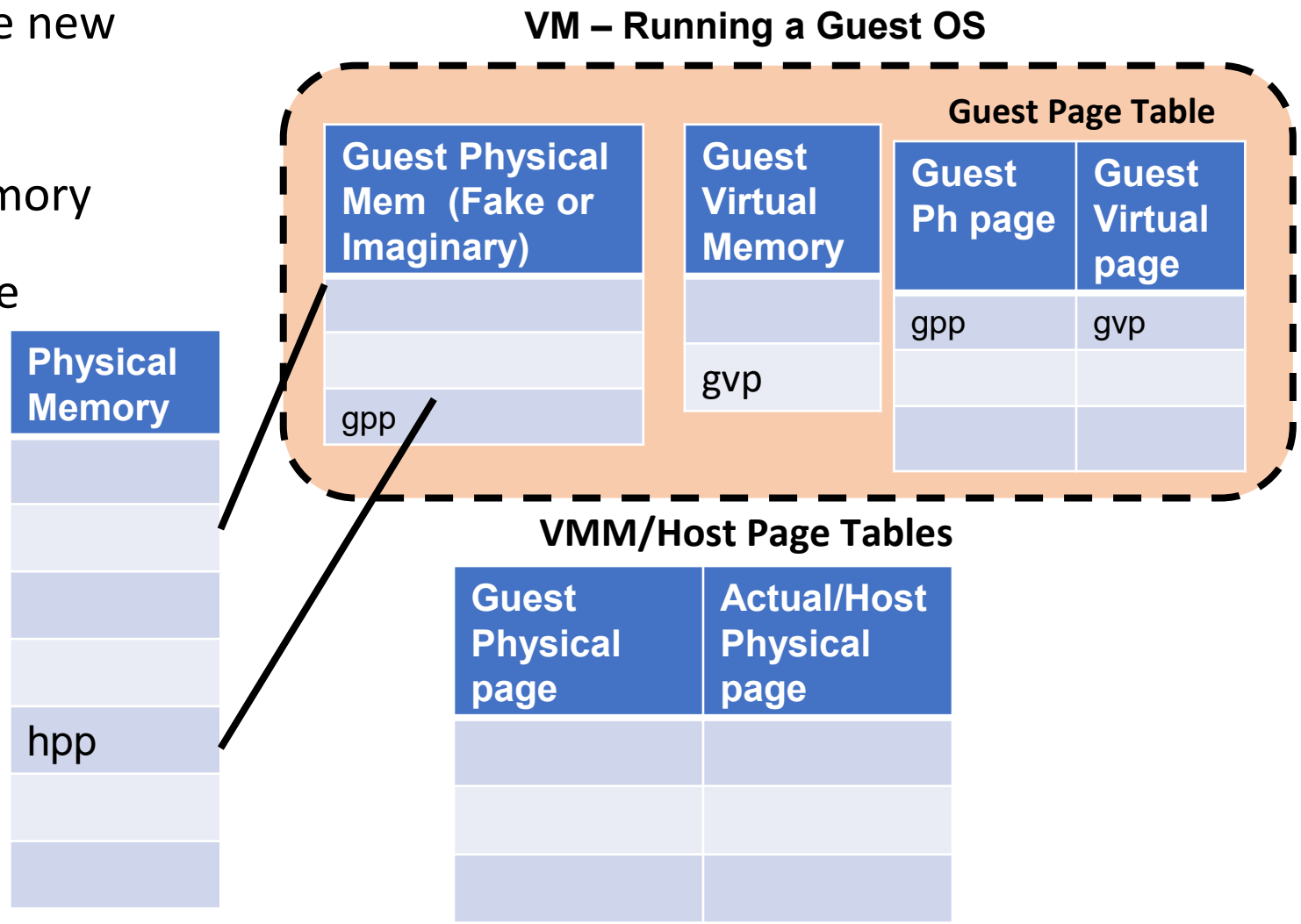
     Example: *pushf*
     - simply reveals that the kernel is running in user mode
   - There were about 17 such instructions in x86 E.g. SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT, RET, STR, MOV in the initial architecuture
   - X86 ISA (pre-2005) does not meet the Popek & Goldberg requirements for virtualization

Problems with Trap and Emulate Virtualization technique

4. Guest OS may realize it is running at lower privilege level
   - Some registers in x86 reflect CPU privilege level (code segment/CS)
5. In trying to keep the Guest OS at the lower privileged level, memory protection is another challenge as physical memory is a shared resource.
   - So may need to not allow Guest OS to directly allow memory access and map fake physical pages needing two levels of virtual address translation.
   - Virtual page to fake physical page and then the fake physical page to real machine frame/page. Makes it complex to handle this.

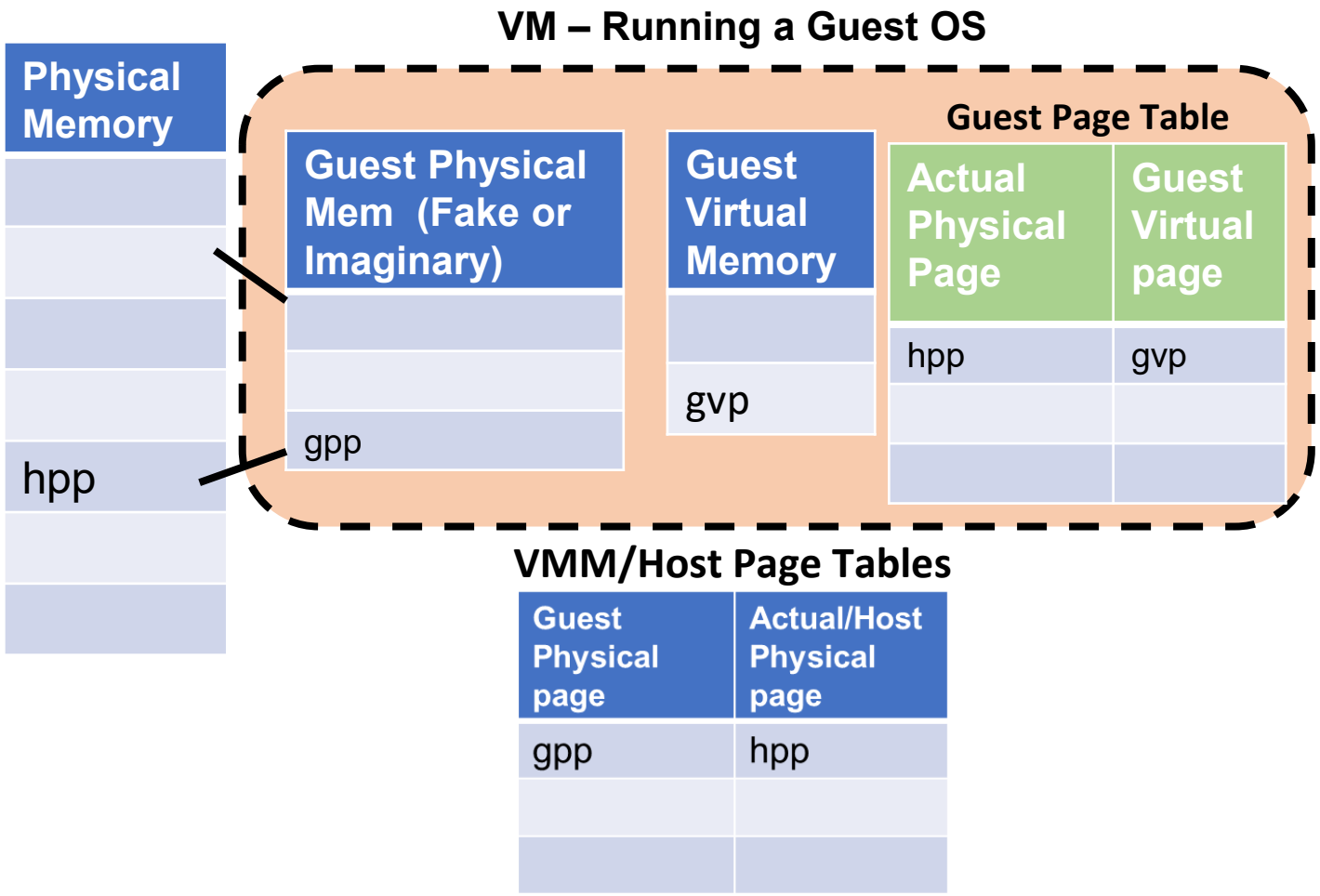
Trap and Emulate technique : Exercise

- Guest OS (e.g., Linux) is trying to define new page *gvp* for process
- It finds an unused page *gpp* in VM memory
- It tries to load (gvp,gpp) into page table
- hpp is a free physical page
- What happens?



Trap and Emulate Exercise Solution

1. Guest OS tries to modify page tables so that gvp points to gpp
2. Hypervisor traps this privileged instruction
3. Hypervisor maps (if not already done) gpp to a host physical page hpp
4. Loads the Guest page table to get gvp to point to hpp by copying it from VMM/Host page table. (to enhance usability)
5. Other page table operations (e.g., read) have to be trapped as well
6. We have skipped TLB manipulation



As an implementation the new Page table in Green is also called Shadow Paging/Page table

Why did these problems crop up during Virtualization & What is Strictly Virtualizable

- ***Why these problems***?
  - OSes not developed to run at a lower privilege level
  - Instruction set architecture of x86 is not easily virtualizable (x86 wasn't designed with virtualization in mind)
- A processor or mode of a processor is ***strictly virtualizable*** if, when executed in a lesser privileged mode:
  - all instructions that access privileged state should trap
  - all instructions either trap or execute identically

Again this another operational definition.

This idea behind strictly virtualizable is whether trap and emulate will work.

Binary Translation Technique

- Binary translation is one specific approach to implementing full virtualization that does not require hardware virtualization features -say like the VT instructions for x86 which support the 17 non virtualization instructions for trap and emulation.
- Not having support of the VT instructions limit the public IaaS VMs to be virtualizable further by Users .. Say like EC@ instance being virtualized further.
- Binary translation involves the hypervisor examining the virtual guest code for "unsafe or sensitive" but unprivileged instructions, and translating these into "safe" or privileged equivalents, and then executing the translated code
- Thus in this approach, since there is no sensitive unprivileged instructions (because they got translated) there is no need for a trap to be generated and handled
- This helps in the Transparent or full virtualization technique
- Supports virtualization of x86 architecture
- Similar technique was used in DEC, but was used for Virtualization as we see by VMWare
- Conceptually simple

E.g. Replacing the *popf* instruction with trap to kernel and emulation

Binary Translation Technique - Functioning

- The monitor inspects the next sequence of instructions. An instruction sequence is typically defined as the next basic block, that is all instructions up to the next control transfer instruction such as a branch. Such a code will be executed from start to finish by the CPU and hence is ideal for translation. Post this there is a call-out to other pieces of code which will need to be emulated.
- Each instruction is translated and the translation is copied into a ***translation cache***.
- Instructions are translated as follows:
  - Instructions which pose no problems can be copied into the translation cache with modification. We call these “ident” translations.
  - Some simple dangerous instructions can be translated into a short sequence emulation code and directly into the translation cache. This is known as “inline” translation. An example is the modification of the Interrupt Enable flag.
  - Other dangerous instructions need to be performed by emulation code in the VMM for which calls are to be made which are called “Call-outs”. An example of these is a change to the page table base.

Binary Translation Complexities

- Control flow changes
  - Translation can change addresses of instructions
  - Branches have to be re-translated
  - Keep track of branch addresses

```
isPrime:  mov   %ecx, %edi ; %ecx = %edi (a)
          mov   %esi, $2   ; i = 2
          cmp   %esi, %ecx ; is i >= a?
          jge   prime      ; jump if yes
```

```
isPrime':  mov %ecx, %edi    ; IDENT
           mov %esi, $2
           cmp %esi, %ecx
           jge [takenAddr]   ; JCC
           jmp [fallthrAddr]
```

Additional references

Binary Translation, Trap & Emulate
https://www.sciencedirect.com/topics/computer-science/binary-translation

http://www.cs.cmu.edu/~410-f06/lectures/L31_Virtualization.pdf

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

CLOUD COMPUTING

Hardware Virtualization

Venkatesh Prasad
Department of Computer Science

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

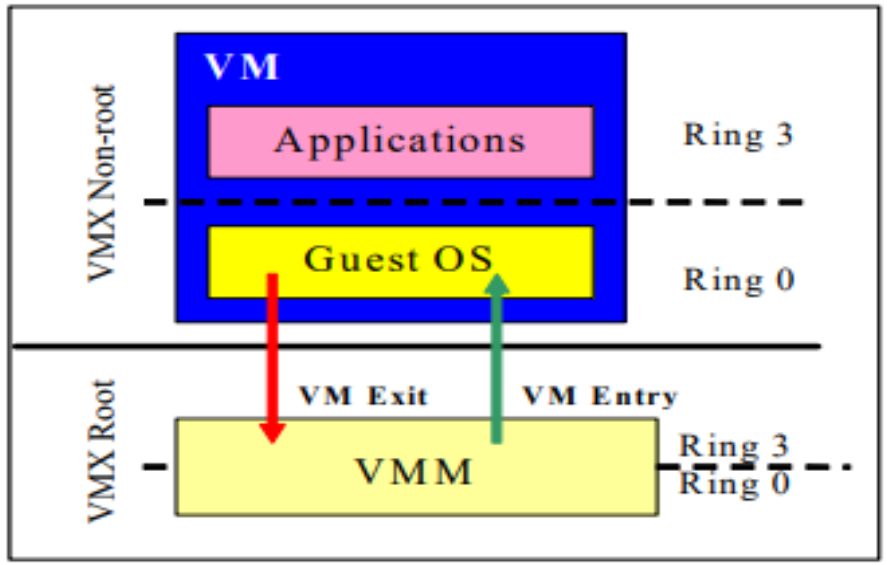
1. Slides of Prof Arkaprava Basu and Prof Sorav Bansal
2. Some slides, conceptual text and diagram from Scott Devine, Vmware
3. Text book and Reference books prescribed in the syllabus
4. Other Internet sources

Techniques to virtualize x86

- Paravirtualization: rewrite guest OS code to be virtualizable
  - Guest OS won't invoke privileged operations, makes "hypercalls" to VMM
  - Needs OS source code changes, cannot work with unmodified OS
  - Example: Xen hypervisor
- Full virtualization: CPU instructions of guest OS are translated to be virtualizable
  - Sensitive instructions translated to trap to VMM
  - Dynamic (on the fly) binary translation, so works with unmodified OS
  - Higher overhead than paravirtualization
  - Example: VMWare workstation

Techniques to virtualize x86 (Cont.)

- Hardware assisted virtualization: Eg. KVM in Linux
  - CPU has a special VMX mode of execution
  - Two kinds of VMX transitions: VMX Root and VMX Non-Root
    - X86 has 4 rings in both VMX root and Non-root modes
- VMM enters VMX mode to run guest OS in (special) ring 0
- Exit back to VMM on triggers (VMM retains control)



Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain instructions (such as the new VMCALL instruction) and events cause VM exits to the VMM. Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources.

Workaround for x86-32: Paravirtualization

- Co-design of guest OS and hypervisor
  - Advantage: Simplicity, work around corner cases
  - Disadvantage: Need **modified** guest OS
  - Example: Xen hypervisor

- Trick to virtualize x86-32:
  - Block all 17 instructions that are sensitive but not privileged
    - Just modify the guest OS to #undef them
  - Instead provide ***hypercalls*** from guest OS to VMM
    - Hypercalls are like system calls but from guest OS to VMM

Hardware assisted virtualization: Intel VT-x/AMD-v for x86-64

Two challenges of virtualizing x86-64 ISA:

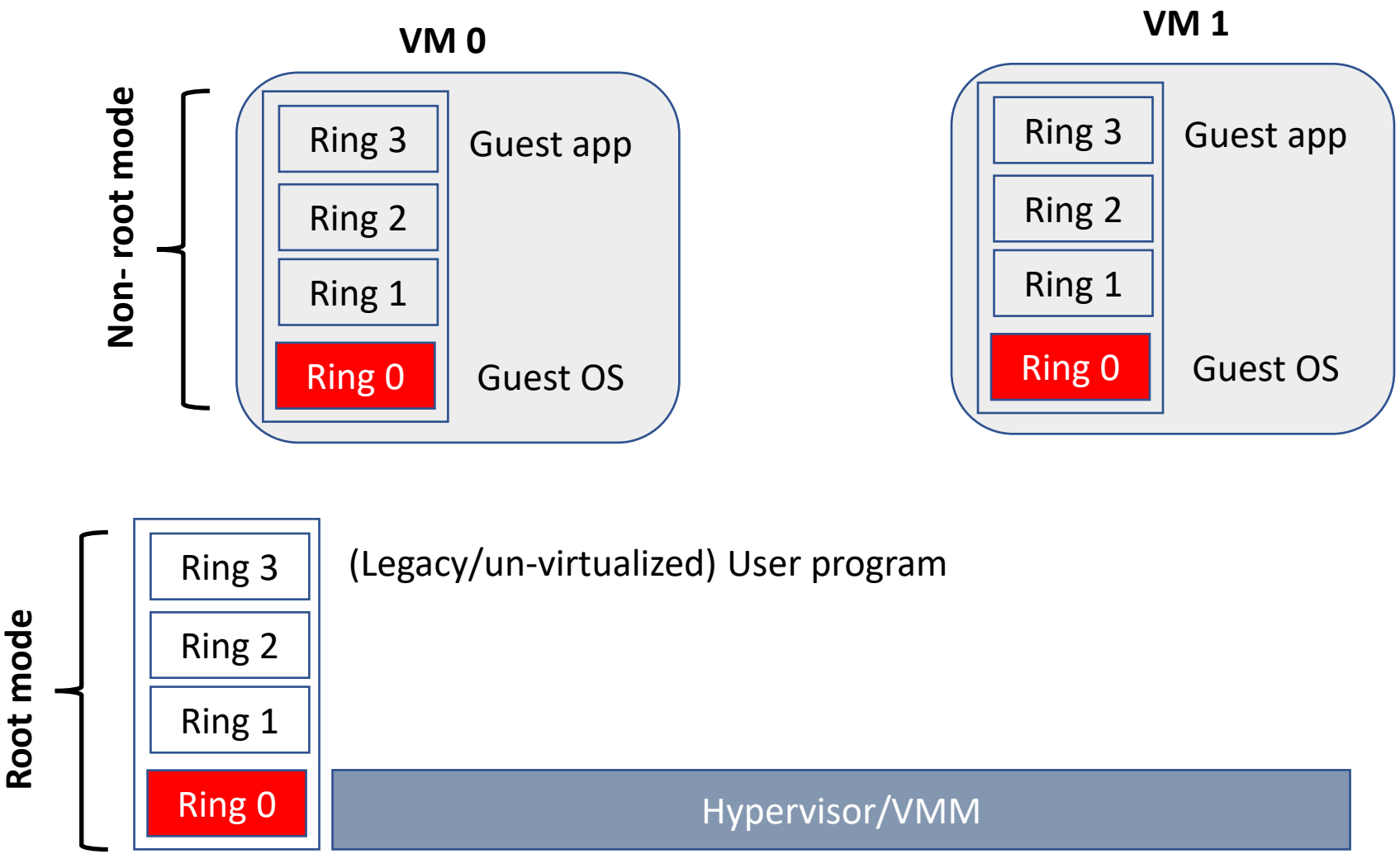
- How to hide system/privileged state from the VM?
- How to ensure a VM cannot directly change system state (e.g., interrupt flags) of the processor?

Hardware assisted virtualization (Cont.)

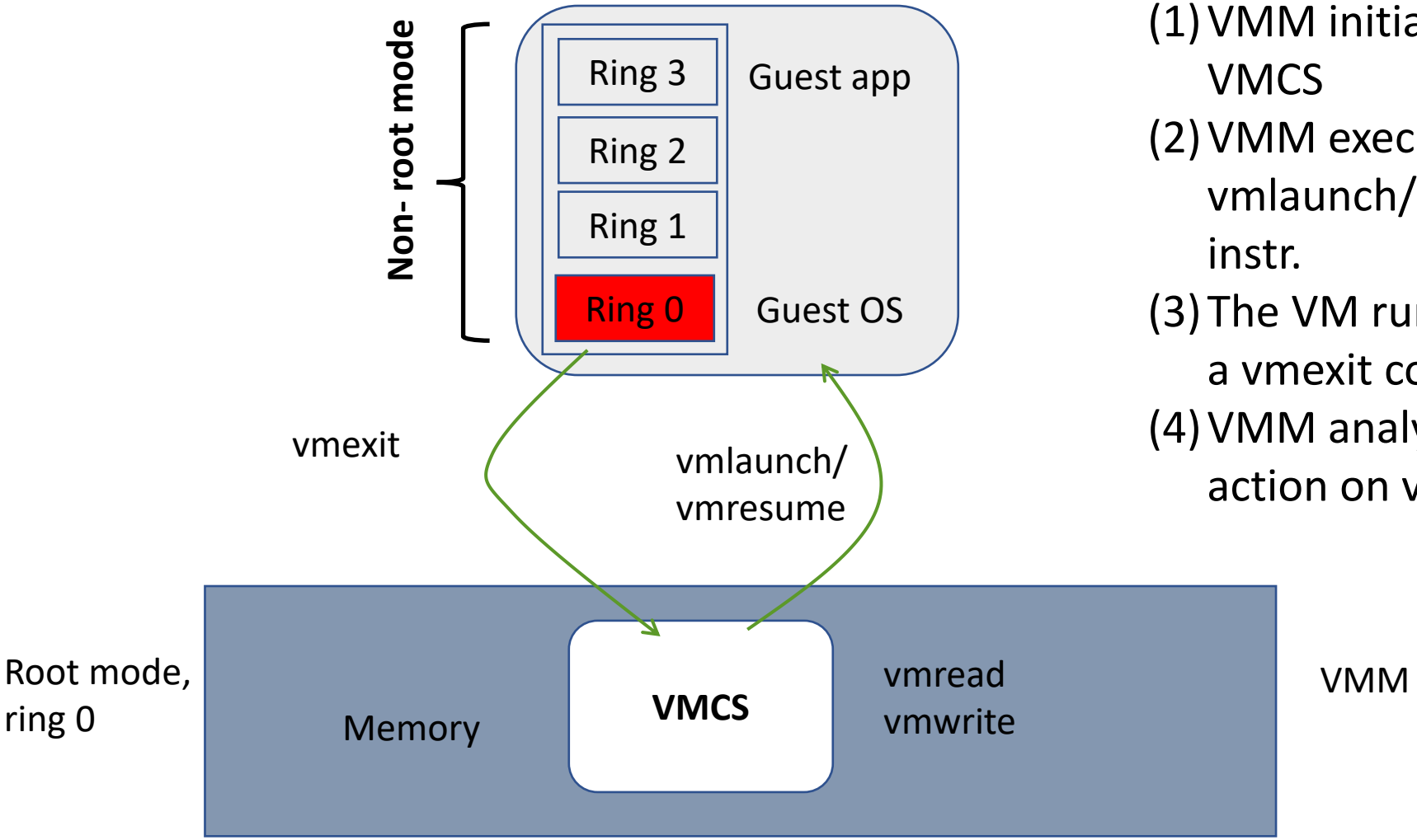
- Solution idea:
  - Two *new* modes of operation: root and non-root
    - Each mode has complete set of execution rings (0-3)
    - New instructions to switch between modes
  - H/W state is duplicated for each operation mode
  - Hypervisor runs in root mode and VMs in non-root mode
  - When any “sensitive” instruction executed in non-root mode it either (1) executed by processor on duplicated state or (2) trap to hypervisor

CLOUD COMPUTING

Hardware assisted virtualization (Cont.)



Using Intel VT-x/AMD-v (KVM way)



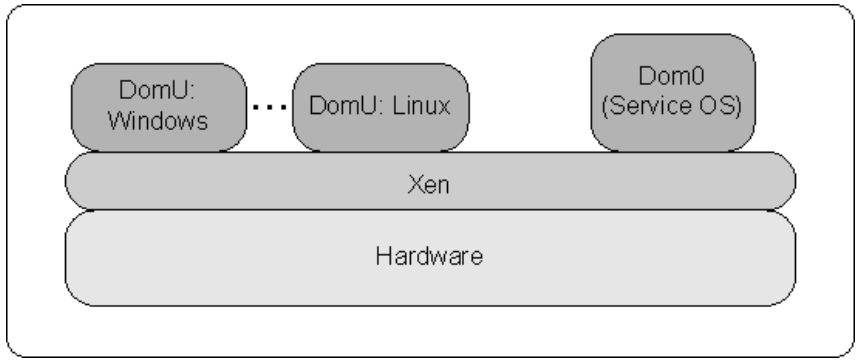
(1) VMM initializes/modifies VMCS
(2) VMM executes vmlaunch/vmresume instr.
(3) The VM runs until it hits a vmexit condition
(4) VMM analyzes and takes action on vmexit

Current status of H/W assist

Almost all VMMs make use of h/w assist today
KVM is integrated part of Linux and makes heavy use of h/w assist
Even Xen and VMWare workstation uses it

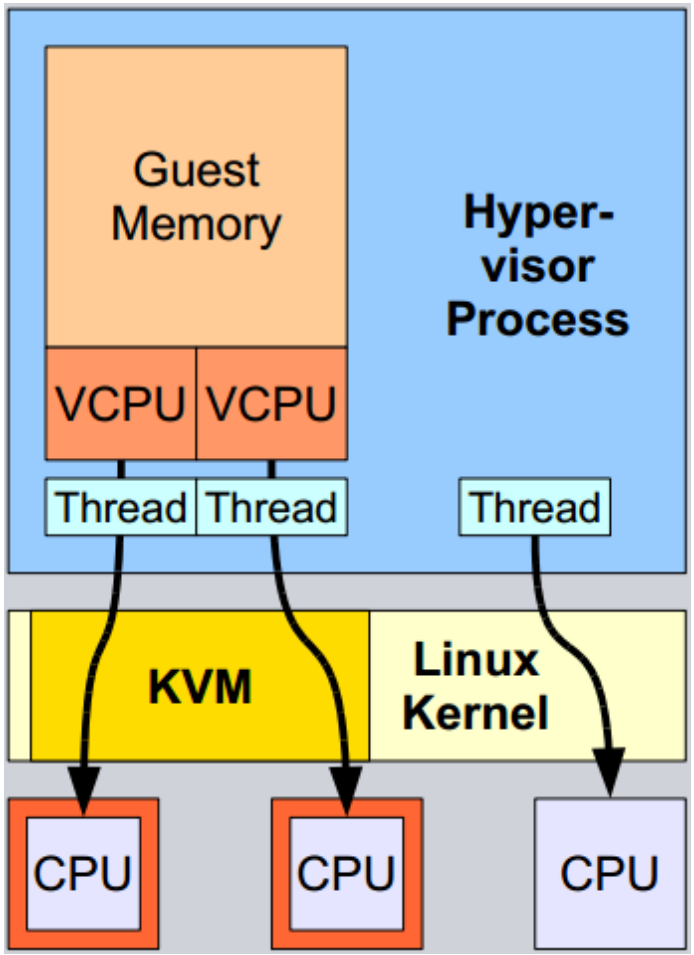
Xen architecture

- Paravirtualization hypervisor
- VMs called *domains*
- Dom0
  - Special domain
  - Based upon Linux
  - Handles all I/O
  - Windows: uses filter drivers
- Xen hypervisor
  - All functions other than I/O
  - Trap and emulate
  - Binary translation
    - Sensitive instructions replaced by hypervisor calls



KVM Architecture – Processor Support

- Extends Linux kernel to add hypervisor functionality
  - Leverage Linux code for scheduling, paging, …
  - Uses VT-x, EPT
- Each VM appears as a Linux process



CLOUD COMPUTING

Additional Reading

www.vmware.com/pdf/asplos235_adams.pdf
https://www.cse.iitb.ac.in/~cs695/slides_pdf/04-hwvirt-kvmqemu.pdf
https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-tech-converged-application-platforms-paper.pdf

THANK YOU

Venkatesh Prasad
Department of Computer Science Engineering

venkateshprasad@pes.edu

CLOUD COMPUTING

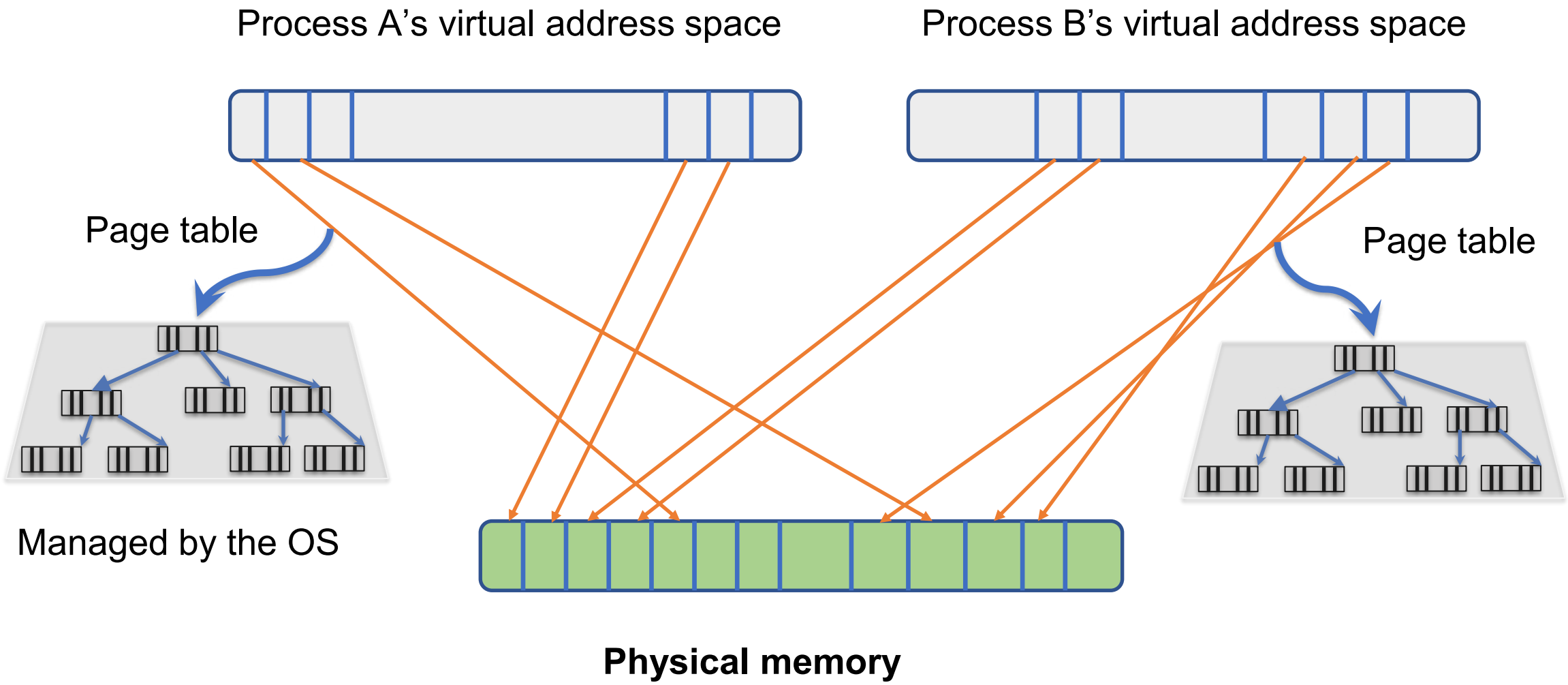
Virtualization – Memory and I/O

Prof. Venkatesh Prasad
Dr. H.L. Phalachandra

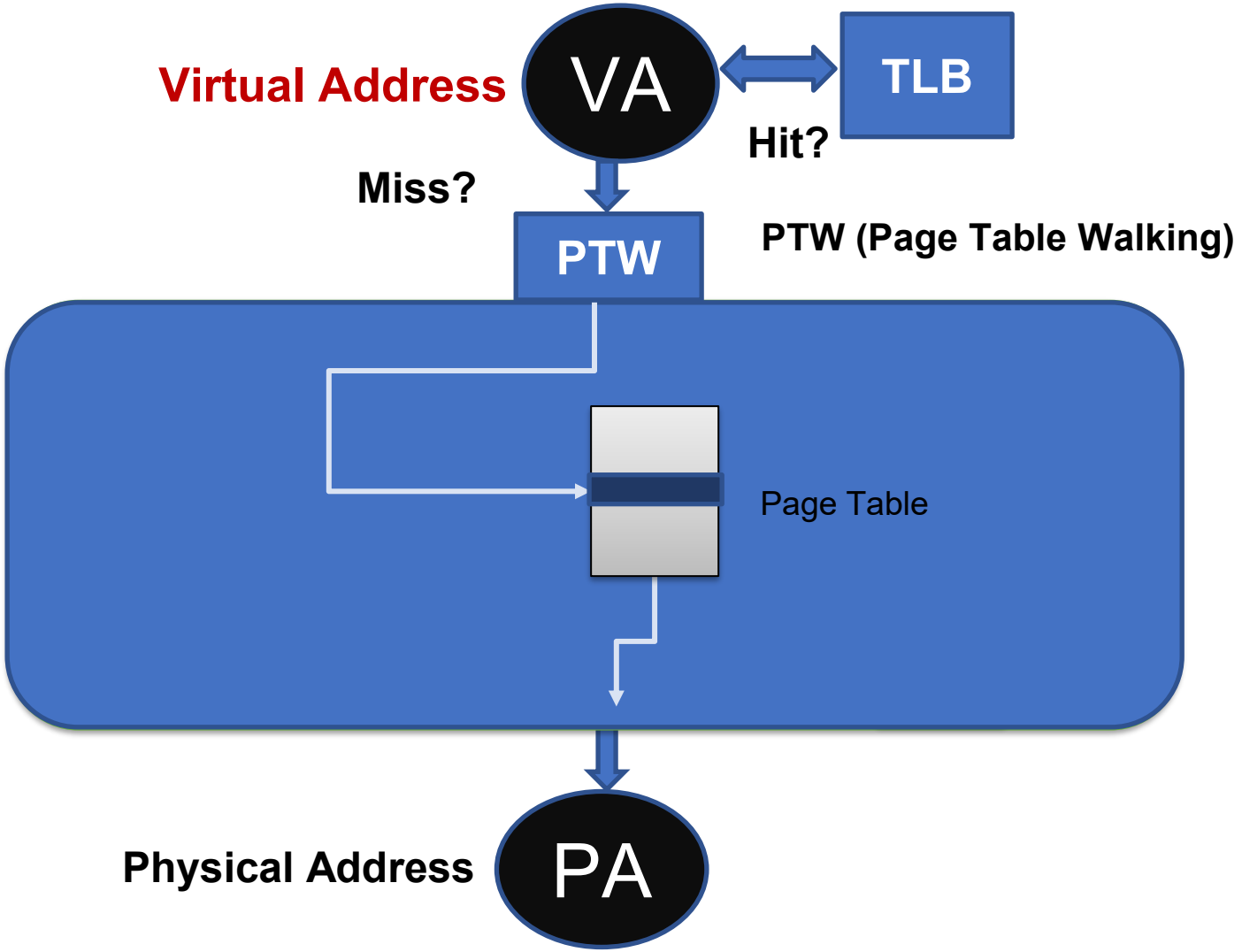
Department of Computer Science and Engineering

Acknowledgements:
Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

Recap of Virtualization in Un-virtualized system



Recap of Performing address translation in unvirtualized machine



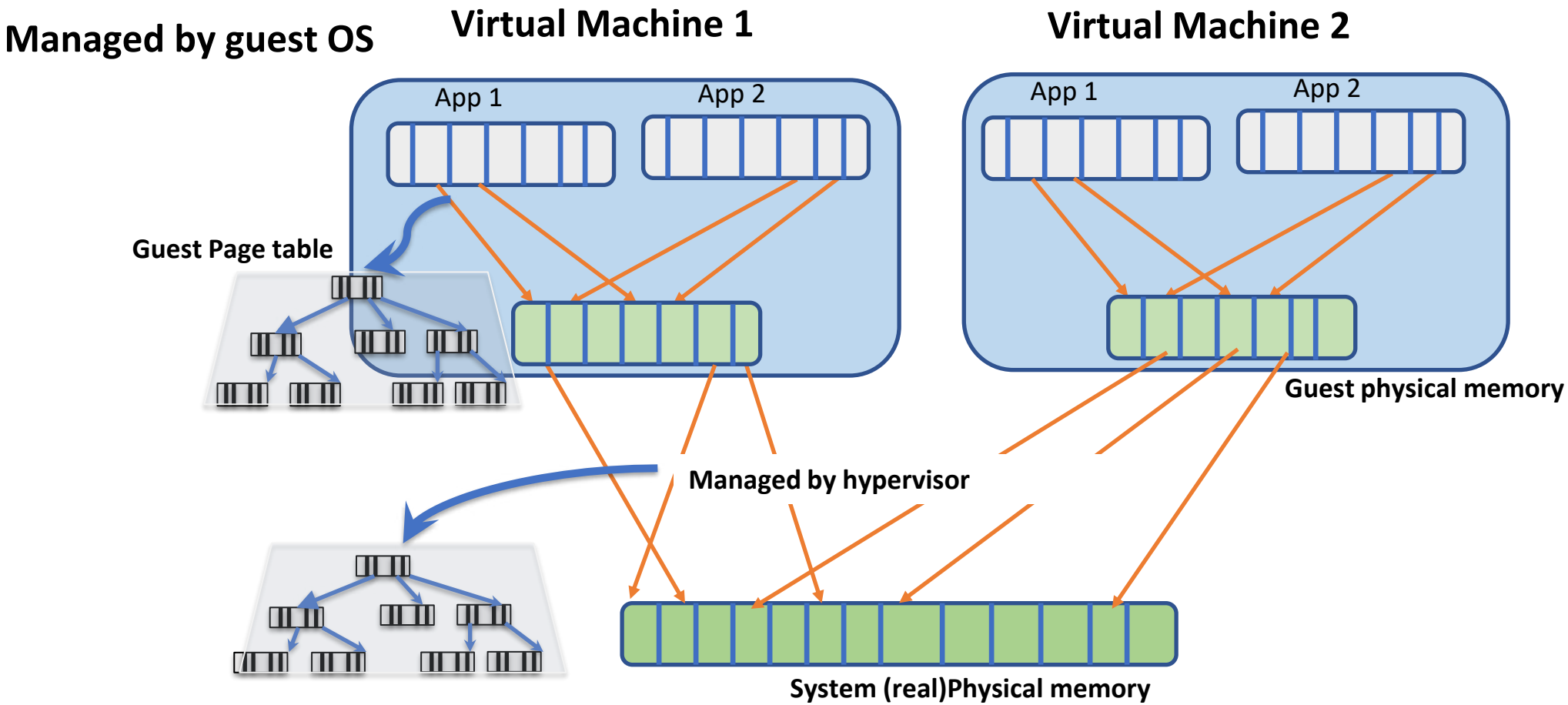
Page Walking (Recap)

- Software page walker is a OS handler that walks the page table
  - slower than hardware page table walker due to large address translation overhead
  - Example: SPARC (Sun/Oracle Machines)
- Hardware page walker generates load-like “instructions” to access page table
  - Example: x86 and ARM processors
- During the page walk, the page walker encounters physical addresses in CR3 register and in page table entries which point to the next level of the walk. The page walk ends when the data or leaf page is reached.
- Address translation is a very memory intensive operation because the page walker must access the memory hierarchy many times. To reduce this overhead, processors automatically store recent translations in an internal translation look-aside buffer (TLB).

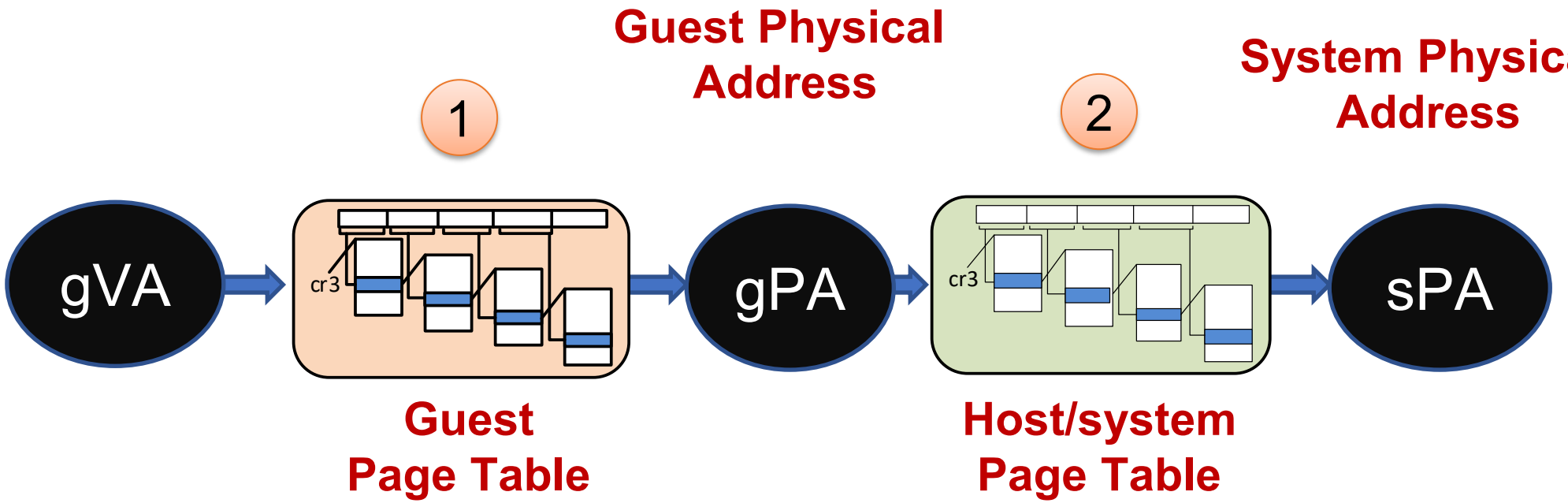
Requirement for memory virtualization

- **VMs or Guest OSes** on **VMs** should not have direct access to physical memory
- **Hypervisor** or the **VMM** should **only manage** the **Host Physical memory**
- **Guest OS** should be **fooled**/or should be made to believe that its accessing the physical memory

Virtualizing virtual memory



Virtualizing virtual memory (Cont.)



Two levels of address translation on each memory access by an application running inside a VM

Implementing two level address translation

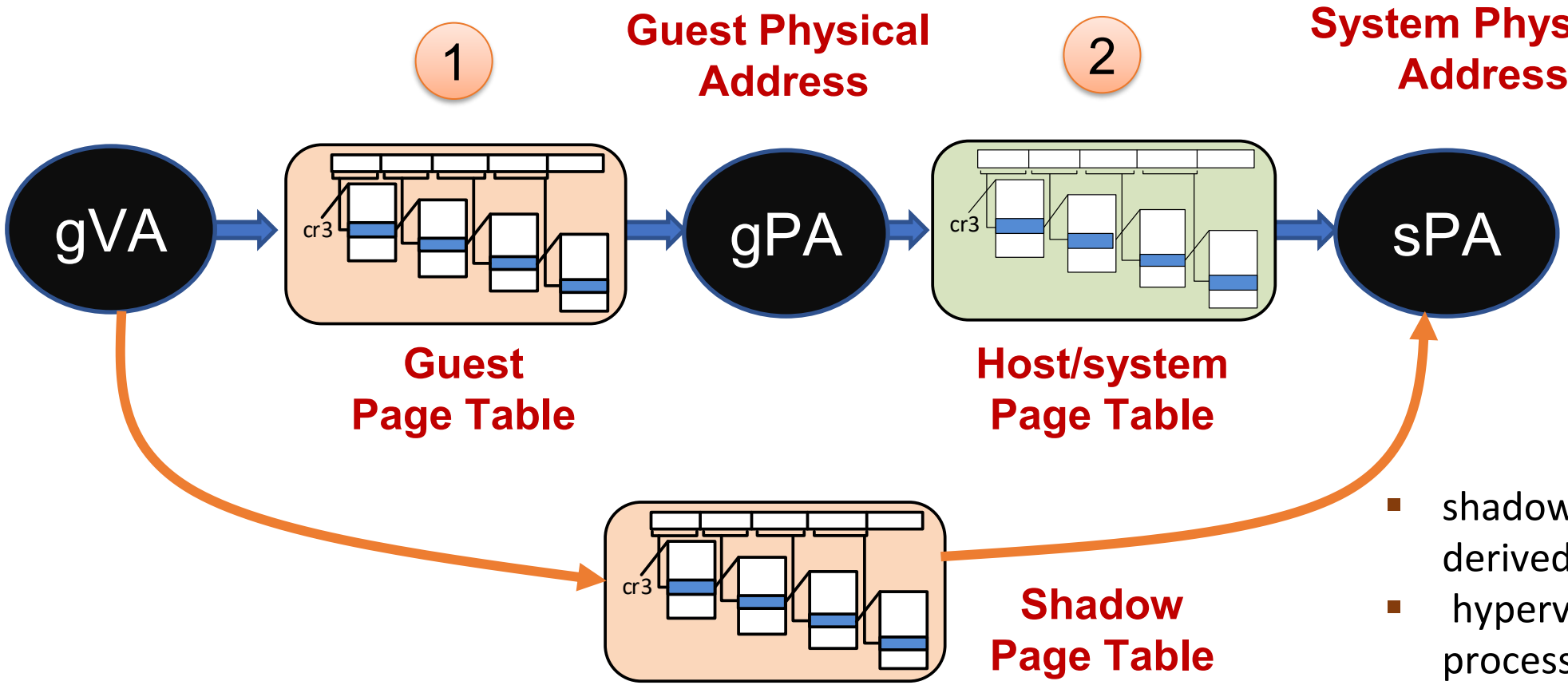
- **Shadow page table**
  - Each page table of the guest OS has a separate page table in the VMM corresponding to it, the VMM page table is called the shadow page table.
  - Shadow paging was developed as a ***software-only technique*** to virtualize memory before hardware support was implemented.
- **Nested/Extended page table**
  - ***Hardware supported technique***
  - A additional page table is set up by the hypervisor to perform a second level address translation

Shadow page table

- **Idea**: Let hypervisor “create” a shadow page table that maps guest VA to host PA directly
  - Made by combining guest page table with system page table
  - Hypervisor makes the *cr3* point to the shadow page table
  - A hardware or software page table walker (PTW) walks the shadow page table
  - Control Register CR3 points to the root of page table
  - Translation from Virtual Page Number (VPN) to Physical page frame number or fault occurs

Ref: https://www.cse.iitk.ac.in/users/biswap/CASS18/VM.pdf

Shadow page table

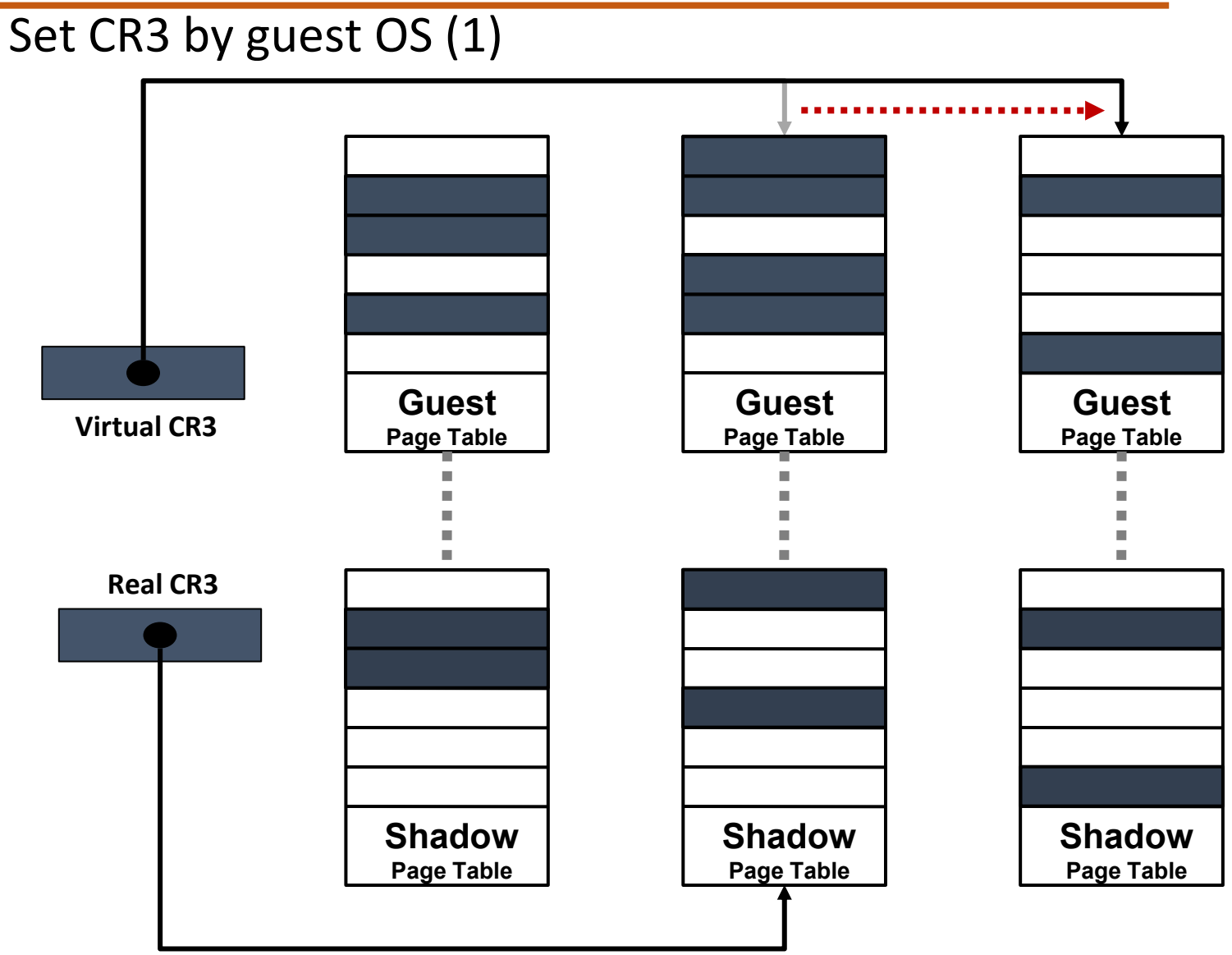


- shadow version of page table derived from gPT.
- hypervisor forces the processor to use sPT(shadow page table) to perform address translation.

Shadow page tables

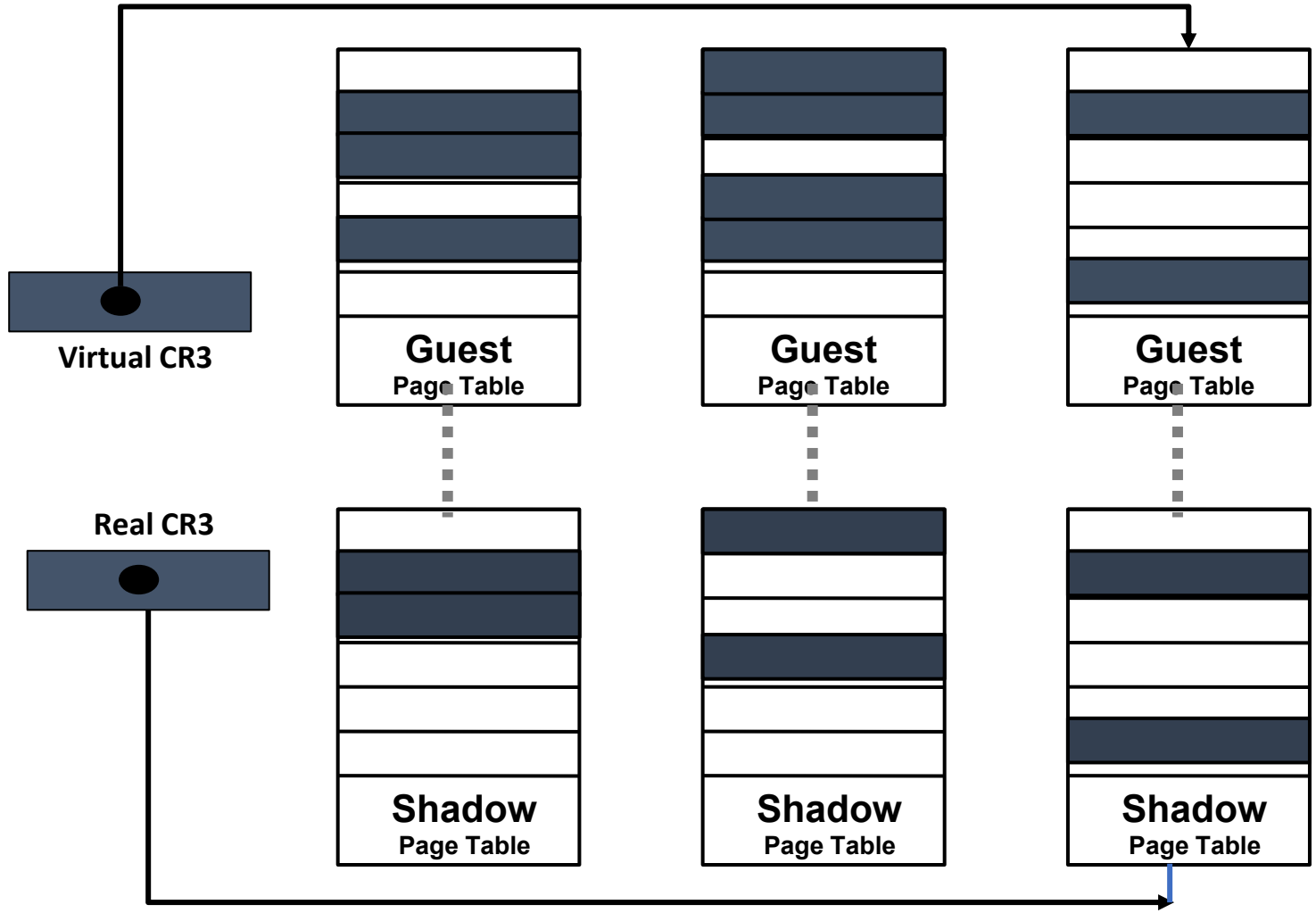
- VMM maintains shadow page tables that map guest-virtual pages directly to machine pages.
- Guest modifications to the guest Page tables (containing gVA->gPA) synced to VMM managed Shadow Page tables (containing gVA->sPA).
  - Guest OS page tables marked as read-only.
  - Modifications of guest page tables by guest OS -> trapped to VMM.
  - Shadow page tables synced to the guest OS page tables
- Software-based techniques maintain a shadow version of page table derived from guest page table (gPT).
- When the guest is active, the hypervisor forces the processor to use the shadow page table (sPT) to perform address translation.
- The sPT is not visible to the guest.

Shadow page tables (Cont.)



Shadow page tables (Cont.)

Set CR3 by guest OS (2)



Challenge of Shadow page table

- How to create a shadow page table?
  - Anytime guest OS modifies guest page table hypervisor needs to update shadow page table
  - Solution: Write protect guest page table
    - Any write access to guest page table would generate page fault → trap to hypervisor
  - Drawback: Many page faults for application that alters page tables
- For every guest application there is one shadow page table
- Every time guest application context switches trap to hypervisor to change cr3 to point to new shadow page table
- Maintaining consistency between guest page tables and shadow page tables leads to an overhead: VMM traps
- Loss of performance due to TLB flush on every “world-switch”.
- Memory overhead due to shadow copying of guest page tables.

Nested/Extended page table (Hardware Assisted)

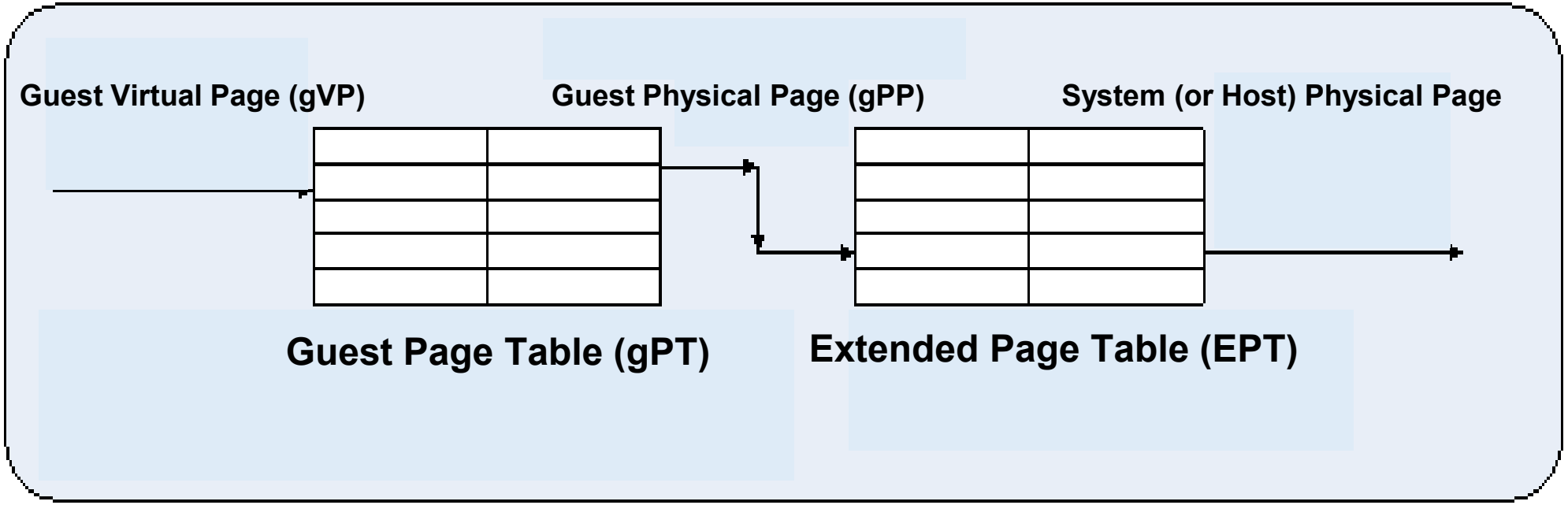
- Nested or Extended page table was introduced to avoid the software overheads of shadow paging.
- In nested paging, uses an additional or nested page table (NPT) to translate guest physical address to system physical address and thus guest will control its page tables
- Differs from shadow paging, that ones nested pages are populated, the hypervisor does not need to intercept and emulate guest page table modifications
- Both the guest and the hypervisor have their own copy of processor state like the CRO, CR3 etc.
- The gPT set up by the guest OS maps guest virtual address to guest physical address. Nested page tables (nPT) setup by the Hypervisor, maps guest physical address to system physical address
- When a guest attempts to reference memory using a virtual address, and nPT is enabled, the page walker to find the mapping, performs a two dimension walk using the gPT and the nPT to translate the guest virtual address to system physical address
- Summarily

  Idea : Make hardware aware of two levels of address translation the guest and nested page table

  - Make hardware page walker walk both guest page table and nested page table on a TLB miss → Two dimensional page walk
  - Two cr3s : gCR3 → gPT;
    nCR3 → nPT
  - Eliminates the need of shadow page table

Ref: http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf

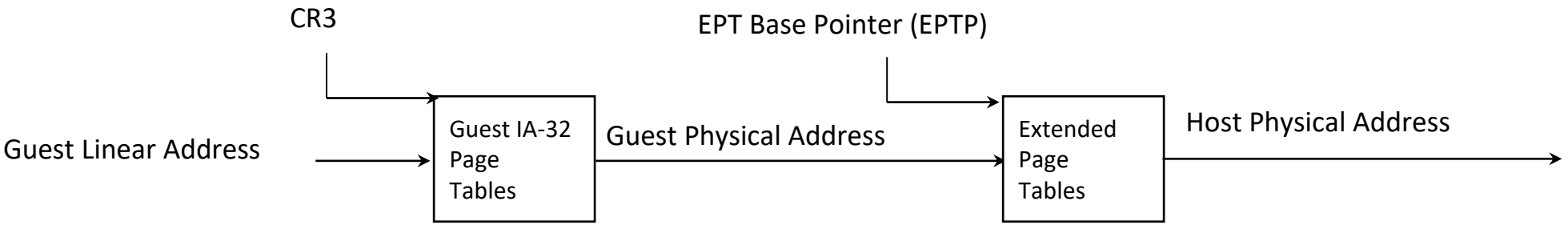
Extended page tables



Guest manages page tables (one per process running on the VM)
Hypervisor manages Extended Page Tables (EPT)

CLOUD COMPUTING

EPT – Another Summary



- Extended Page Table
- A new page-table structure, under the control of the VMM
  - Defines mapping between guest- and host-physical addresses
  - EPT base pointer (new VMCS field) points to the EPT page tables
  - EPT (optionally) activated on VM entry, deactivated on VM exit
- Guest has full control over its own IA-32 page tables
  - No VM exits due to guest page faults, INVLPG, or CR3 changes

when the guest removes a translation, it executes INVLPG to invalidate that translation in the TLB. The hypervisor intercepts this operation; it then removes the corresponding translation in sPT and executes INVLPG for the removed translation.

Virtual Machine Control Structure (VMCS) Field IDs used by the Hypervisor framework read and write functions.

EPT (Extended Page Tables) : Performance

- **Estimated EPT benefit is very dependent on workload**
  - Typical benefit estimated up to 20%[1]
  - Outliers exist (e.g., forkwait, Cygwin gcc, > 40%)
  - Benefit increases with number of virtual CPUs (relative to MP page table virtualization algorithm)
- **Secondary benefits of EPT:**
  - No need for complex page table virtualization algorithm
  - Reduced memory footprint compared with shadow page-table algorithms
    - Shadow page tables required for each guest user process
    - Single EPT supports entire VM

Summarily EPT improves Memory Virtualization Performance

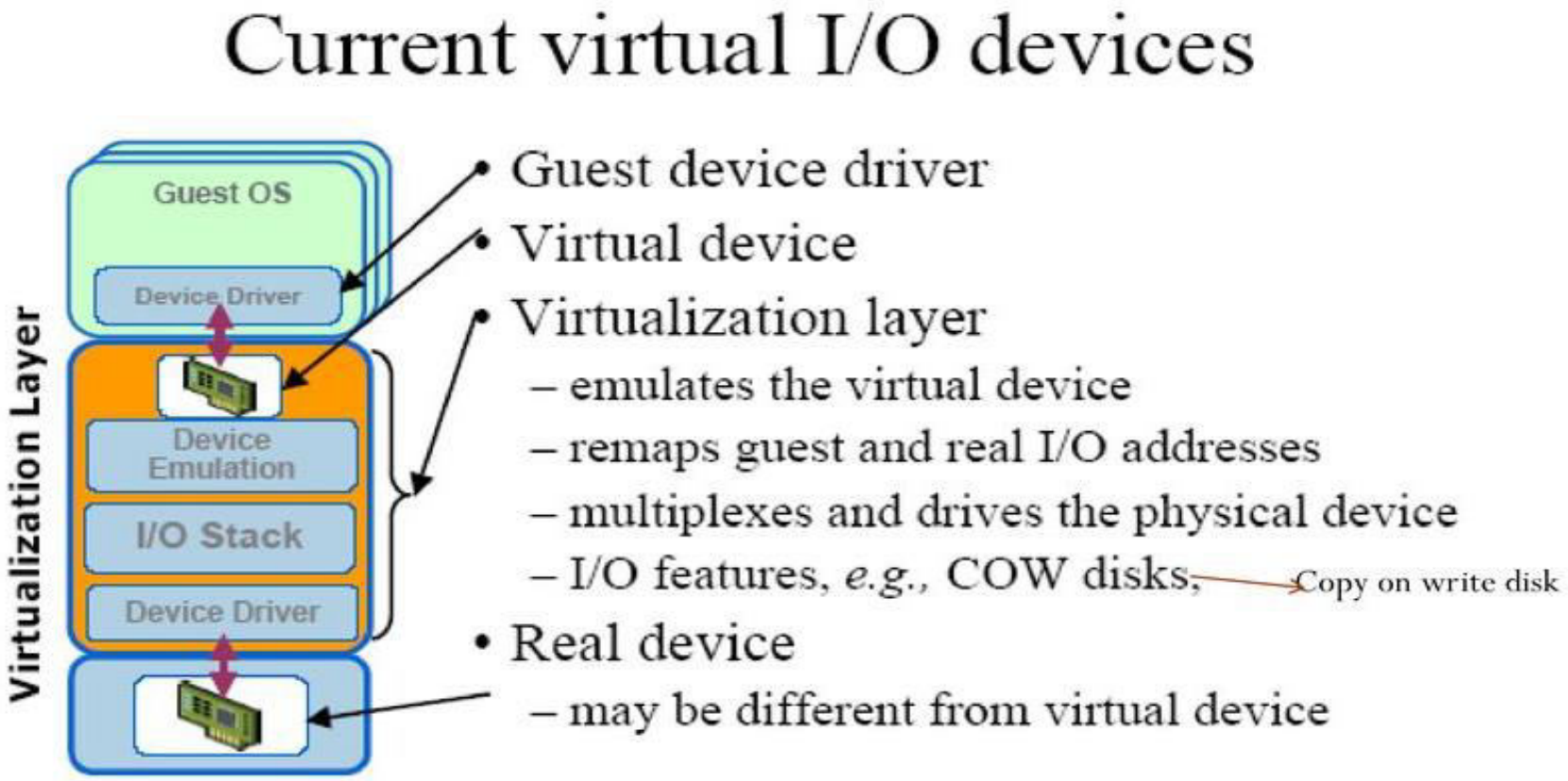
I/O virtualization

- I/O virtualization (IOV), or input/output virtualization, is technology that uses software to abstract upper-layer protocols from physical connections or physical transports.
- I/O virtualization involves managing the routing of I/O requests between virtual devices and the shared physical hardware.
- Three ways to implement I/O virtualization:
  1. Full device Emulation
  2. Para-Virtualization
  3. Direct I/O

1. Full device emulation

- Full device emulates well-known, real-world devices.
  - All the functions of a device or bus infrastructure, such as device enumeration, identification, interrupts, and DMA, are replicated in software.
  - This software is located in the VMM and acts as a virtual device. The I/O access requests of the guest OS are trapped in the VMM which interacts with the I/O devices.

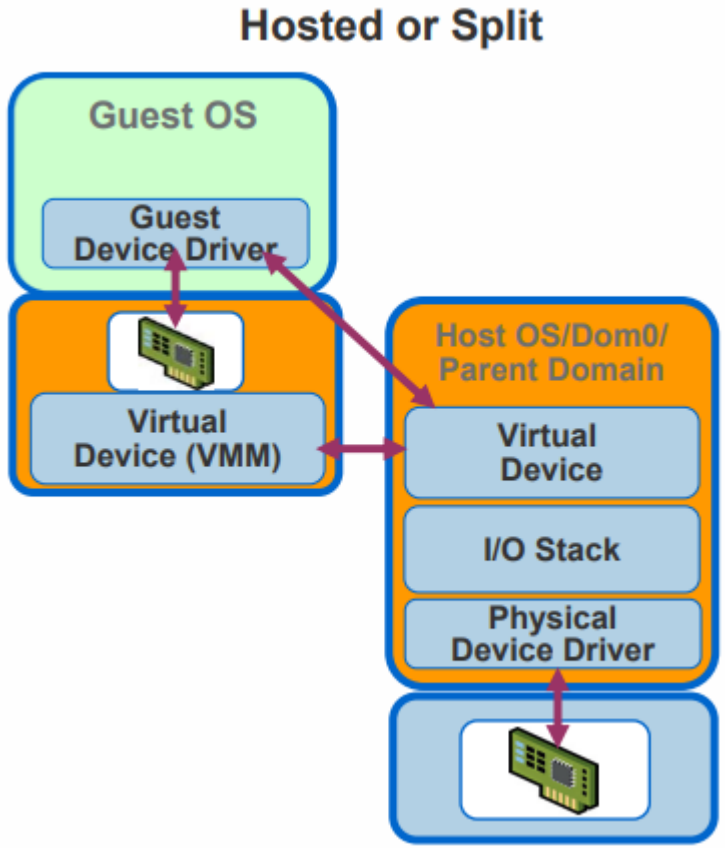
1. Full Device emulation for I/O Virtualization



Device emulation for I/O virtualization implemented inside the middle layer that maps real I/O devices into the virtual devices for the guest device driver to use.

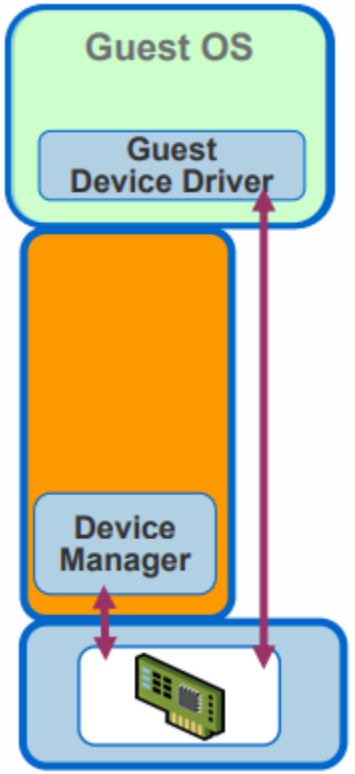
2. Para-I/O-Virtualization or Split Driver or Hosted I/O Virtualization

- It is also known as **Hosted** or the **split driver model** consisting of a frontend driver and a backend driver.
- The frontend driver is running in Domain U (User/Guest) and the backend driver is running in Domain 0. They interact with each other via a block of shared memory.
- The frontend driver manages the I/O requests of the guest OSes and the backend driver is responsible for managing the real I/O devices and multiplexing the I/O data of different VMs.
- Although para-I/O-virtualization achieves better device performance than full device emulation, it comes with a higher CPU overhead.
- Para virtualization method of I/O virtualization is typically used in Xen



3. Direct I/O (or Passthrough) I/O Virtualization

- Direct I/O virtualization lets the VM access devices directly.
- It can achieve close-to-native performance without high CPU costs.
- Enables a device to directly DMA to/from host memory.
  - This technique thus provides an ability to bypass the VMM's I/O emulation layer and therefore shows improved throughput for the VM's.
- The Intel VT-x technology allows a VM to have direct access to a physical address (if so configured by the VMM).
- This ability allows a device driver within a VM to be able to write directly to registers IO device (such as configuring DMA descriptors).
- Intel VT-d provides a similar capability for IO devices to be able to write directly to the memory space of a virtual machine, for example a DMA operation.



- The mechanism for doing direct assignment varies from one vendor to another. However, the basic idea is that the VMM utilizes and configures technologies such as Intel VT-x and Intel VT-d to perform address translation when sending data to and from an IO device.
  - The main concern with direct assignment is that it has limited scalability, a physical device can only be assigned to one VM.

Ref: https://virtualizationdeepdive.wordpress.com/about/6-2/ https://download3.vmware.com/vmworld/2006/tac0080.pdf

I/O virtualization : Advantages

Flexibility: Since I/O virtualization involves abstracting the upper layer protocols from the underlying physical connections, it offers greater flexibility in terms of hardware independence, utilization and faster provisioning in comparison with normal NIC and HBA cards.

Cost minimization: I/O virtualization methodology involves using fewer cables, cards and switch ports without compromising on network I/O performance.

Increased density: I/O virtualization increases the practical density of I/O by allowing more connections to exist in a given space.

Minimizing cables: The I/O virtualization helps to reduce the multiple cables needed to connect servers to storage and network.

Additional References

https://www.cse.iitb.ac.in/~cs695/slides_pdf/06-memvirt.pdf

https://www.cse.iitb.ac.in/~cs695/papers/iovirt.pdf

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

Xen I/O Virtualization

- Xen does not emulate hardware devices
- Exposes device abstractions for simplicity and performance
- I/O data transferred to/from guest via Xen using shared-memory buffers
- Virtualized interrupts: light-weight event delivery mechanism from Xen-guest
  - Update a bitmap in shared memory
  - Optional call-back handlers registered by O/S

CLOUD COMPUTING

Virtualization – Goldberg/Popek Principles

Dr. H.L. Phalachandra
Prof. Venkatesh Prasad

Department of Computer Science and Engineering

Acknowledgements:
Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

Background

- We discussed Cloud Computing as a paradigm which provides large number of benefits as discussed earlier, where
  - Computation happens somewhere in the cloud
  - Scalable Resources are made available to applications as a Service and at different service levels
- We also discussed that Virtualization is one of the fundamental technologies which enables Cloud Computing
- Virtualisation divides the resources of a computer into multiple execution environments using partial or complete machine simulation or emulation and allows it to be shared between different tenants or users.
- The typical granularity of the compute to which the resources of a single physical server is divided would be a Virtual Machine
- This is typically achieved using a layer of software, the Virtual Machine Manager or the hypervisor that provides illusion of a "real" machine to multiple instances of "virtual machines“

Background - 2

- This hypervisor could run directly on the real hardware or it could run as an application on top of a host operating system
- This VMM will perform the activities typically done by an OS and interfaces with the hardware and the VMs on either side.
- VMM or the Hypervisor uses two techniques “Trap and Emulate” and “Binary Translation” to function as the virtualization layer supporting the execution of a Guest Application, on a Guest OS on a VM.
- “Trap and Emulate” of these two is more prevalent as we discussed till now, has a requirement that all sensitive instructions which needs emulation in the Hypervisor can be trapped.
- Not all system architectures (e.g. the original x86 architecture) can support virtualization.
- Gerald Popek and Robert Goldberg in 1974 defined the requirements for an architecture to efficiently support virtualization.

Popek and Goldberg principles : Terminologies

As part of their research work they

1. Set out a set of terminologies

   - **Virtual machine:** Complete compute environment with its own isolated processing capabilities, memory and communication channels as created by the VMM.
     - Virtual machine is an efficient isolated duplicate of the physical machine
   - **Virtual Machine Monitor (VMM)/Hypervisor:** System software that creates and manages virtual machines that is
     - Efficient : All safe guest instructions run directly on hardware
     - Omnipotent: Only the VMM can manipulate sensitive state
     - Undetectable: A guest cannot determine that it is running atop a VMM
   - These are applicable for **Third generation machines** implying that the system has at least two operating modes user and supervisor with some instructions not available for user mode. Addressing is with respect to a relocation register and the system has an ability to perform table look ups etc.

Ref: https://cis.temple.edu/~ingargio/cis307/readings/virtualization.html#:~:text=Privileged%20Instructions%3A%20instructions%20that%20if,%2CSIDT%2CSLDT%2CSMSW.

Popek and Goldberg principles : Essential Characteristics

2. They came up with a few essential characteristics or requirements for the VMM
   - **Equivalence**: VMM provides an environment for programs which is essentially identical with the original machine, thus a program running on a VM should behave essentially identically to a program executing directly on the hardware (to indicate isolation from a **protection** perspective).
   - **Resource Control**: The VMM must be in total control of the virtualized resources. This would mean programs running under the VMM should not have an ability to access a resource which is not explicitly allocated and VMM should have an ability to regain control of resources allocated.
   - **Efficiency**: The great majority of the machine instructions must be executed without intervention of the VMM (i.e. without trap) or programs running in this environment show at worst only minor decreases in speed

Ref: https://cis.temple.edu/~ingargio/cis307/readings/virtualization.html#:~:text=Privileged%20Instructions%3A%20instructions%20that%20if,%2CSIDT%2CSLDT%2CSMSW.

Goldberg/Popek : Instructions

3. All instructions are bucketed into three types.
   - **Privileged Instructions :** These instructions cause a trap if the processor is not in the privileged mode
   - **Sensitive Instruction :** These access low-level machine states e.g. page tables, IO devices, privilege bits, which needs to be managed by the control software (i.e. an OS or a VMM/Hypervisor)
     - **Behavior sensitive:** Those whose behaviour or result depends on the mode or configuration of the hardware's mode. If guest OS executes these instructions at lower privilege levels, result will be wrong
       - E.g of such instructions on x86 are POP, PUSH, CALL, JMP, INT n, RET, LAR, LSL, VERR, VERW, MOV.
     - **Control sensitive:** those that attempt to change the configuration of resources in the system i.e. that modify the system registers.
       - E.g. of such instructions in x86 are: PUSHF,POPF,SGDT,SIDT,SLDT,SMSW.
   - **Safe Instructions :** Which are not sensitive

Popek and Goldberg : Theorems - 1

4. They also came up with a set of conditions which have been coded-in as a set of theorems, which are the requirements or conditions which could be used to determine or assure that an architecture is Virtualizable when building a system.

   This depends on the relations between control-sensitive, behaviour-sensitive and privileged instructions. The main result of Popek and Goldberg's analysis can be expressed as follows.

   Theorem 1: For any conventional third generation computer, a VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

   - The theorem states that to build a VMM it is sufficient that all instructions that could affect the correct functioning of the VMM (sensitive instructions) always trap and pass control to the VMM.
   - Non-privileged instructions must instead be executed natively (i.e., efficiently).

Ref: https://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements

Popek and Goldberg : Theorems (Cont.) - 2

4. (Cont.)

Theorem 2: A conventional third generation computer is recursively virtualizable if it is:

(a) virtualizable, and

(b) a VMM without any timing dependencies can be constructed for it.

- If architectures cannot be virtualized in the **classic** way, use **binary translation**, which replaces the sensitive instructions that do not generate traps
- Recursive virtualization is the ability to install a VM within a VM or the conditions under which a VMM that can run on a copy of itself can be built.

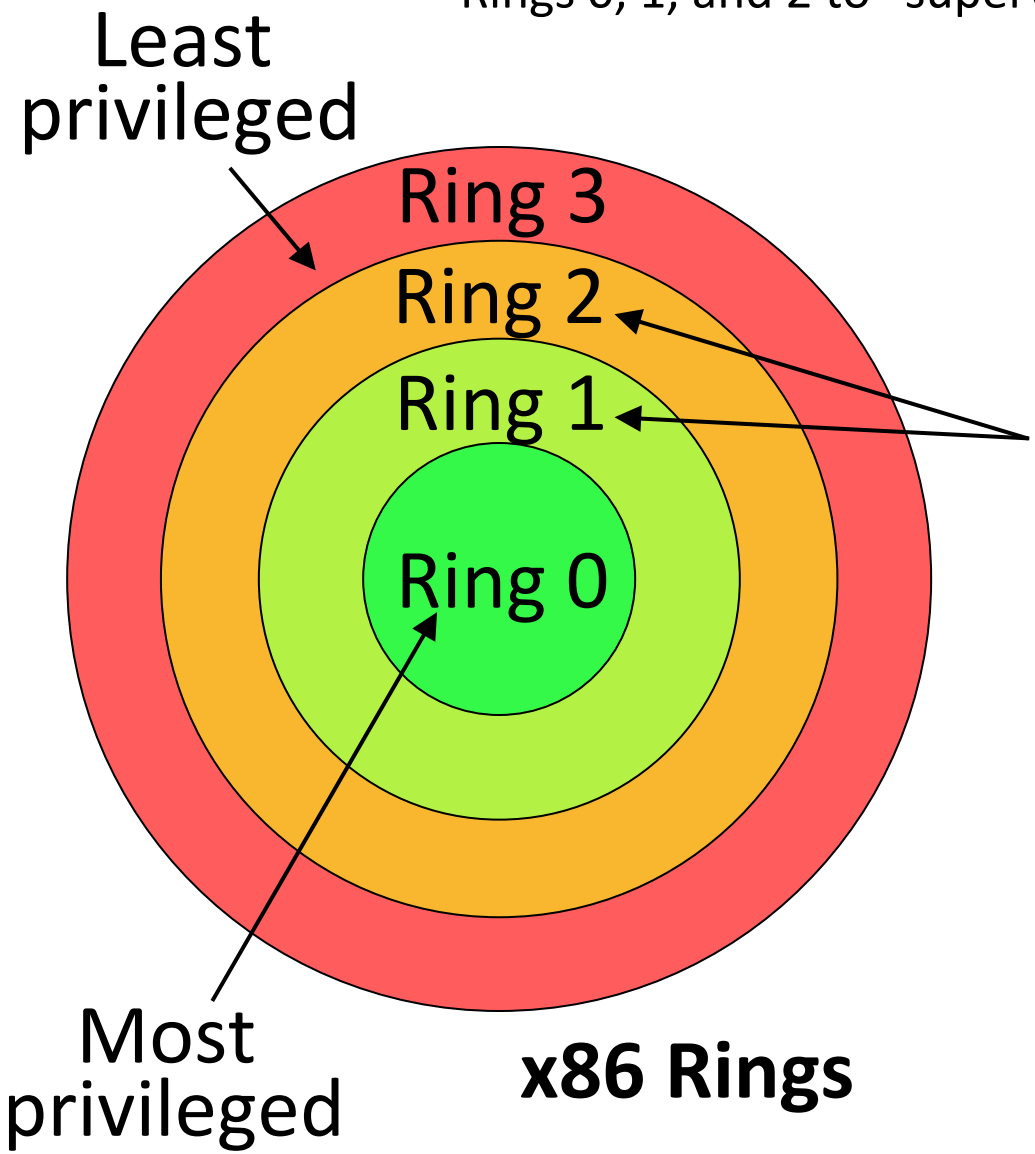
Theorem 3: A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions.

Ref: https://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements

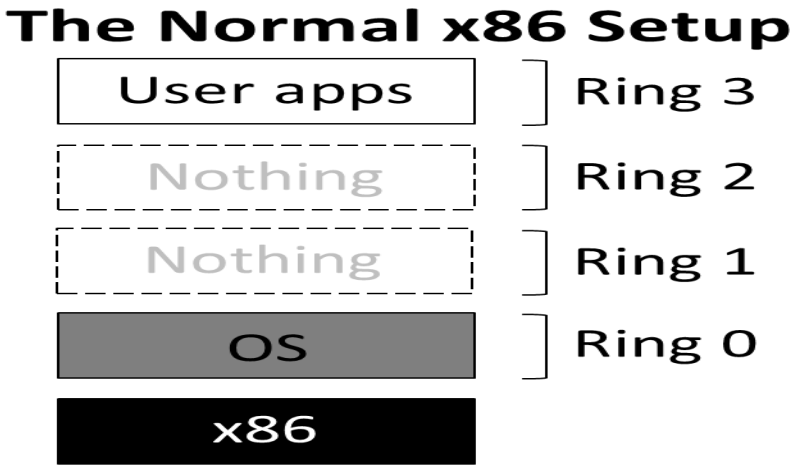
Does x86 support Popek-Goldberg virtualization?

Observation 1: Only Ring 0 can invoke privileged instructions. Rings 1, 2, and 3 trap when invoking privileged instructions.

Observation 2: An x86 PTE only has a single bit to indicate user/supervisor! The MMU maps Rings 0, 1, and 2 to "supervisor," and Ring 3 to "user."



- Intel originally thought that device drivers could be "isolated" here
- However, "isolated" drivers in Rings 1 or 2 can still access ring 0 memory pages, so isolation is weak in practice without additional segment-based protection (but segment support was dropped in x86-64)
- Also, most non-x86 chips only define two privilege levels, so portable OSes shouldn't leverage Rings 1 and 2



Popek and Goldberg: Terminology

- **Privileged instructions** cause a trap if the processor isn't in privileged mode
- **Sensitive instructions** access low-level machine state (e.g., page tables, IO devices, privilege bits) that should be managed by control software (i.e., an OS or a VMM)
- **Safe instructions** are not sensitive

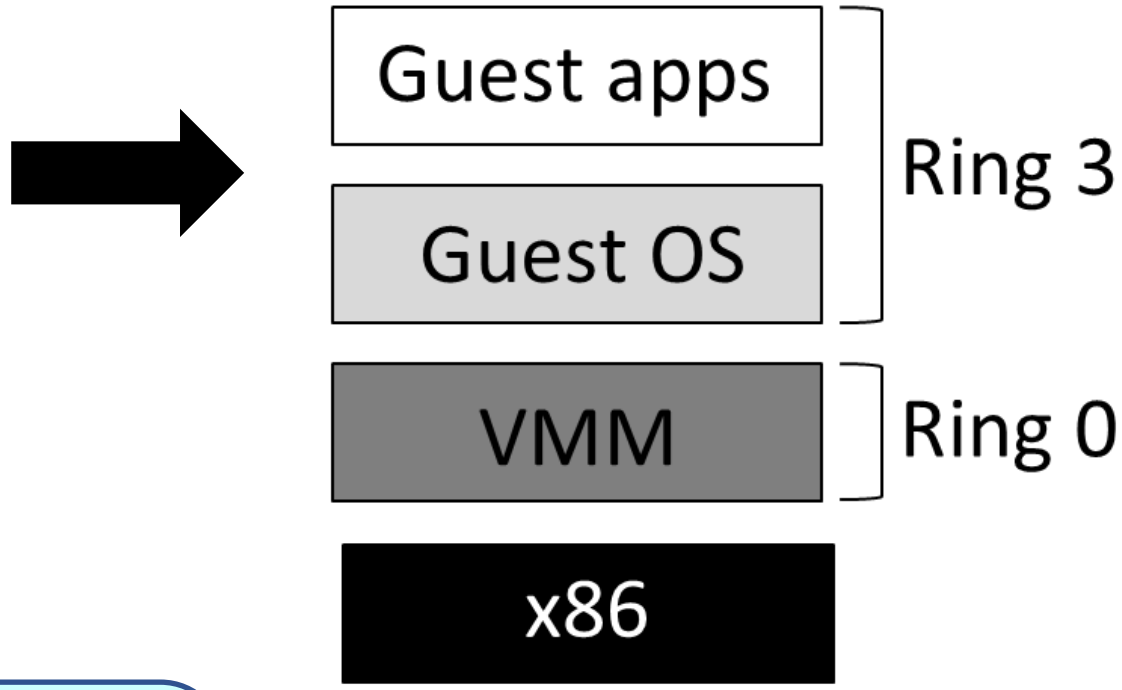
- A VMM is a control program that is:
  - **Efficient:** All safe guest instructions run directly on hardware
  - **Omnipotent:** Only the VMM can manipulate sensitive state
  - **Undetectable:** A guest cannot determine that it is running atop a VMM

Theorem: A virtual machine monitor can be constructed for architectures in which every sensitive instruction is privileged.

Guest app executes syscall to read from keyboard

- Hardware generates trap which is handled by VMM
- VMM vectors control to guest OS's interrupt handler
- Guest OS tries to execute inb to read a key from the keyboard
- Hardware traps to VMM
- VMM executes real inb
- VMM places result where the guest OS expects it to be
- VMM vectors control back to the guest OS
- OS does some processing and then does an iret or sysexit
- Hardware traps to VMM
- VMM returns to user-mode

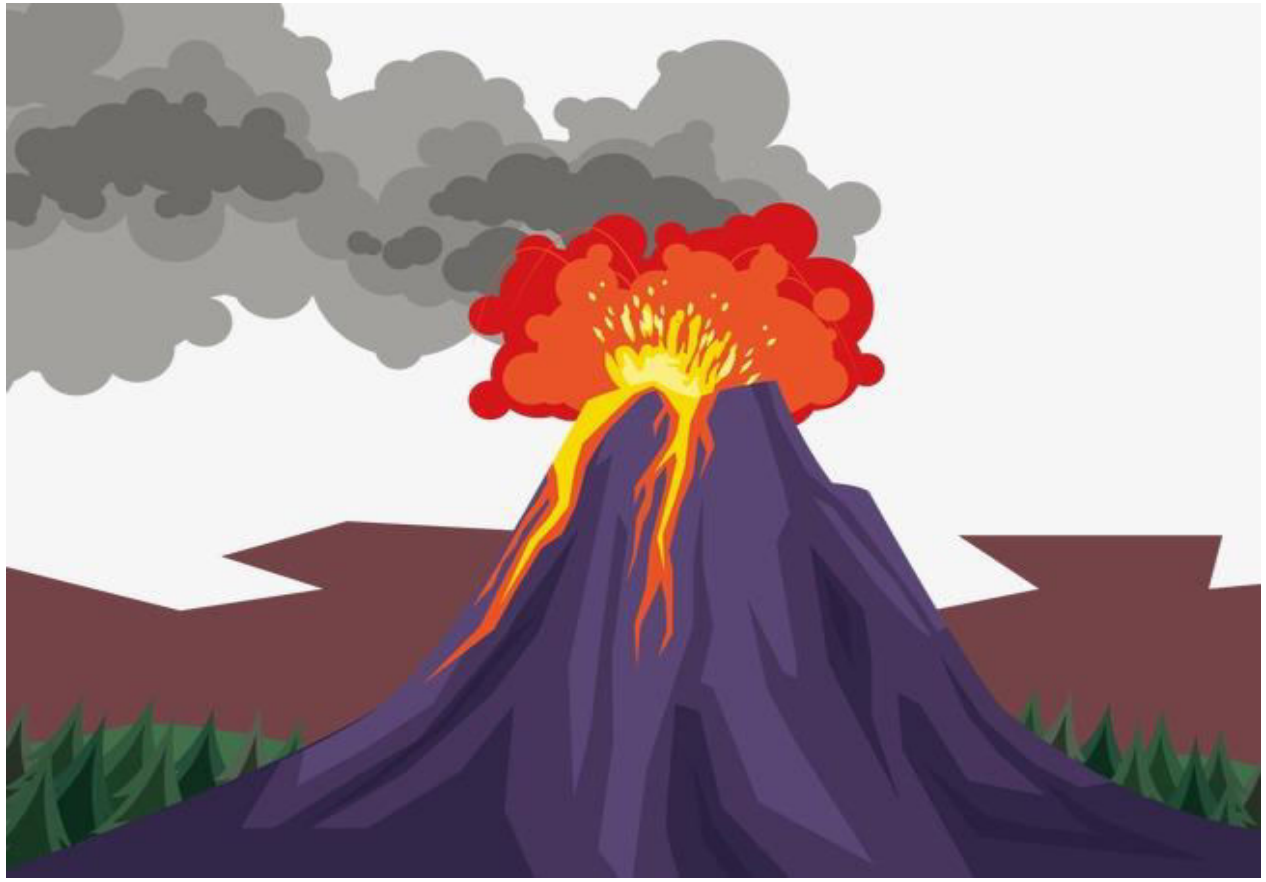
If the guest OS is mapped into the same address space as the guest apps, then the guest apps can read and write the guest OS's state!



There’s another problem . . .

Theorem: A virtual machine monitor can be constructed for architectures in which every sensitive instruction is privileged.

For many years, x86 had sensitive instructions that weren't privileged!



The x86 push instruction was not PG-virtualizable!

- **push** can push a register value onto the top of the stack
  - **%cs** register contains (among other things) 2 bits that represent the current privilege level
  - A guest OS running in Ring 3 could push **%cs** and see that the privilege level isn't Ring 0!
- To be virtualizable, **push** should cause a trap when invoked from Ring 3, allowing the VMM to push a fake **%cs** value which indicates that the guest OS is running in Ring 0

The x86 pushf/popf instructions weren't PG-virtualizable!

- **pushf**/**popf** read/write the **%eflags** register using the value on the top of the stack
  - Bit 9 of **%eflags** enables interrupts
  - In Ring 0, **popf** can set bit 9, but in Ring 3, CPU silently ignores **popf**!
- To be virtualizable, **pushf**/**popf** should cause traps in Ring 3
  - Allows VMM to detect when guest OS wants to changes its interrupt level (meaning that the VMM should change which interrupts it forwards to the guest OS)
- x86 had 17 non-PG-virtualizable instructions!

x86 architecture : Conformance to Popek & Goldberg requirements for virtualization

Summarily

- X86 ISA does not meet the Popek & Goldberg requirements for virtualization
- ISA contained 17+ sensitive , unprivileged instructions:
  - SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT, RET, STR, MOV
- Most simply reveal that the kernel is running in user mode
  - PUSHF
- Some execute inaccurately
  - POPF
- Virtualization is now possible, but requires workarounds or the hardware enhancements

Additional Reading

- https://enacademic.com/dic.nsf/enwiki/480245
- https://www.cs.cmu.edu/~410-s14/lectures/L30_Virtualization.pdf
- https://en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements#:~:text=The%20Popek%20and%20Goldberg%20virtualization,for%20Virtualizable%20Third%20Generation%20Architectures%22.
- https://blog.acolyer.org/2016/02/19/formal-requirements-for-virtualizable-third-generation-architectures/
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.4815&rep=rep1&type=pdf
- https://www.researchgate.net/publication/252019491_Recursive_virtual_machines_for_advanced_security_mechanisms

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

CLOUD COMPUTING

Virtualization – VM Migration

Dr. H.L. Phalachandra
Prof. Venkatesh Prasad

Department of Computer Science and Engineering

Acknowledgements:
Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

VM Migration

- VM migration is the process of moving a virtual machine from one host to another.
- There can be different kinds of migrations
  - **Cold migration** –migrating a powered-off VM (migrations can be across different CPU families)
  - **Non-live(Off-line) VM Migration:** Virtual machine at the source host is paused and then all states of source host is transferred to the target or destination host, and then working of virtual machine at the target host is resumed. This could have the drawback of a larger down time
  - **Live or Hot migration** – migrates a powered-on VM with no disruption to service.
    - including the memory content and all the information that define the virtual machine, such as BIOS, devices, MAC addresses, etc.
    - There are no dropped network connections and applications continue to run uninterrupted.
    - End users are not even aware that the VM has been migrated between two physical

  eg. vSphere vMotion

VM Migration – Why?

- Migrations are done for reasons such as
  - **Hardware Maintenance**
    - Migrates the VMs and the applications running on them to allow maintenance of the physical server
  - **Balancing of the workload**
    - Workload on a specific server or on few of the servers in the cluster of nodes can get unbalanced and live migrations are done to dynamically adjusting loads among nodes
  - **Scaling to a changing workload**
    - Changes in the workload or requirements of the Server may necessitate migration to a physical server which has the required resources needed
    - By supporting automatic scale-up and scale-down of a virtual cluster
  - **Consolidating the Servers for utilization**
    - Effective and optimal utilization of Servers for cost

VM Migration - Why

- Migrations are done for reasons such as (Cont.)
  - **Workload Mobility**
    - Applications based on the compliance requirements may need to move their VMs to different machines or locations
  - **Performance**
    - Mapping VMs onto the most appropriate physical node having the necessary resources to promote performance.
    - Migrating to nodes from nodes whose resources are over-utilized or environments where networks are clogged to shorten the response time of systems.
  - **Energy efficiencies**
    - Given that sustainability is the need for all data center's migrations are used to reduce the energy consumption (Servers, Zones, Airconditioning, Location)
  - **Availability Support, Business Continuity**
    - To support Availability requirements arising out of failures to equipment or environments

Non-Live or Cold migration process

- Non live migration is easiest of the migration technique.
- In this technique the current execution of VMs are suspended before migration and resumed after the migration phase.
- It is a simple technique as the service provided is stopped while migration process continues.
- The state of VM does not resume on the destination server until the migration process completes.
- Memory pages need to be migrated only once using this technique.
- The migration time is also very short and predictable.
- This technique simplifies the migration process but as the service provided is paused during the migration phase, application performance is degraded

Live migration process of VMs

Live migration is the current need in significant use case scenarios as significant number of applications and the services have availability expectations and Live migration supports offering of the same. In other words, the intention would be to migrate without impacting or even making aware the User application running on the VM.

- The live migration of VMs allows workloads of one node to transfer to another node.
- The considerations for choosing to migrate and where to migrate would be the cost or overhead involved for the migration and the total time taken for migration.
- VM migration is a resource-intensive procedure as VM's continuously demand appropriate CPU cycles, cache memory, memory capacity, and communication bandwidth.
- VM migration process typically degrades the performance of running applications and adversely affects efficiency of the data centers, particularly when Service Level Agreements (SLA) and critical business objectives are to be met.

Live VM migrations :

- Live migration used in significant use case scenarios as this supports availability expectations needed for significant number of applications and the.
- These are few objectives of Live migration
  - Migrate by efficiently utilizing the bandwidth to optimize the application performance
  - Minimize the application downtime
  - Reducing the overall time required for transfer by maximum factor
  - Summarily migrate without significantly impacting the User application running on the VM.
- There are two technique of live migration
  - Pre-copy technique
  - Post-copy technique

Live VM migrations : Pre-Copy technique - 1

In Pre-copy migration, VM will continue in running condition and the transfer of memory contents to the destination is done simultaneously. For pre-copy VM migration process there are mainly six stages

1. **Selecting the destination host**
2. **Reservation of Resources**

   This step makes preparations for the migration, including determining the migrating VM and the destination host.

   Although users could manually make a VM migrate to an appointed host, in most circumstances, migration is automatically started by strategies such as load balancing and server consolidation.

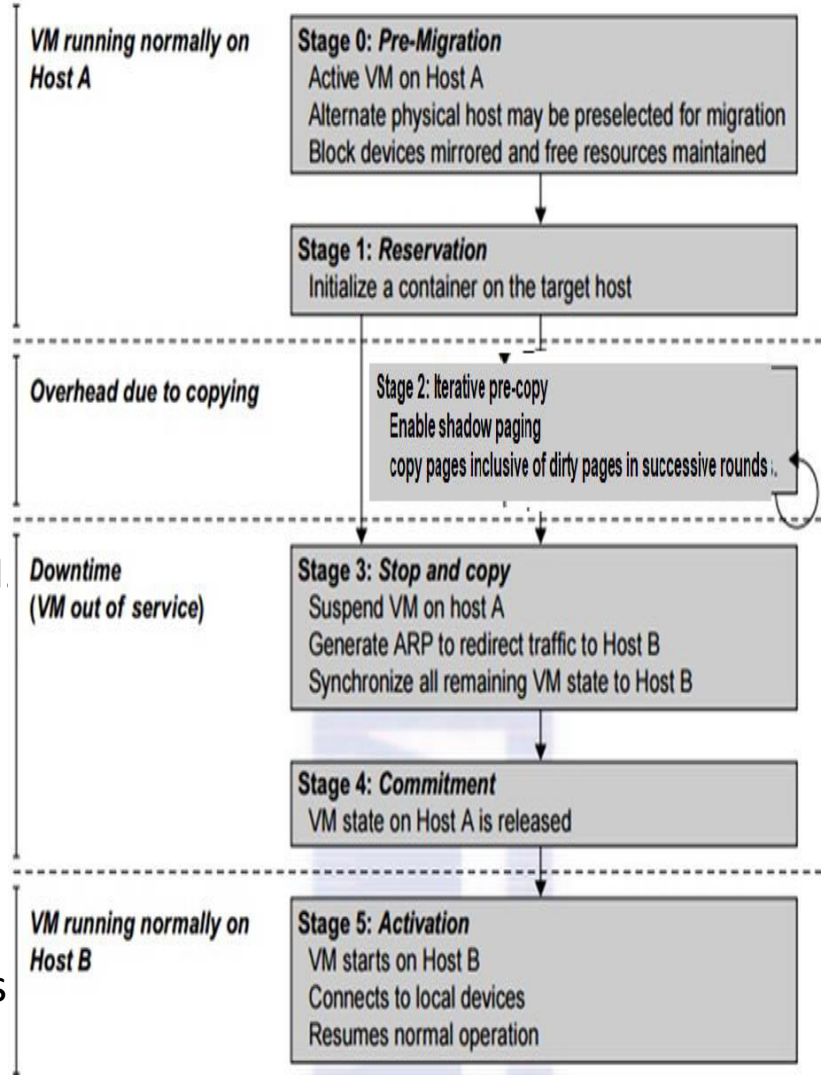
3. **Iterative pre-copying rounds**

   Since the whole execution state of the VM is stored in memory, sending the VM's memory to the destination node ensures continuity of the service provided by VM.

   All of the memory data is transferred in the first round, and then the migration controller recopies the memory data which is changed in the last round (dirty pages).

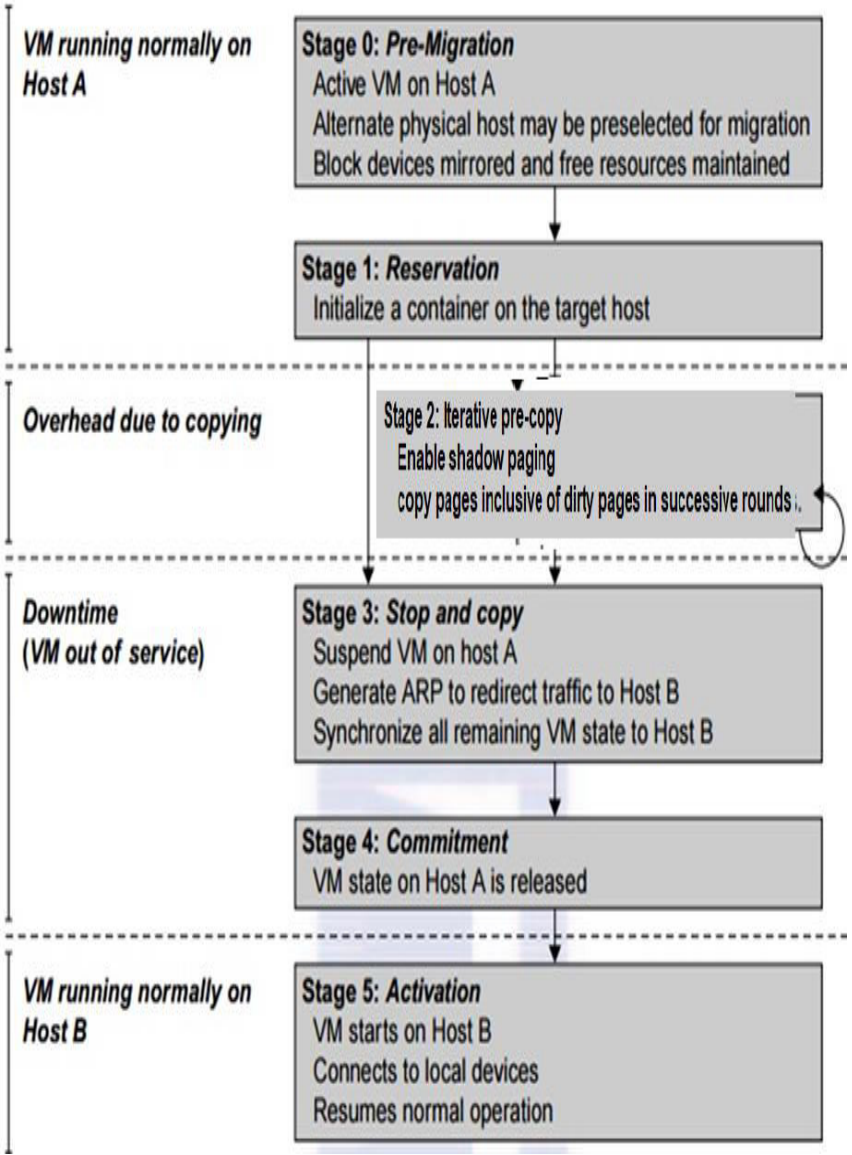
   These steps keep iterating until 1) Fixed threshold is reached 2) the dirty portion of the memory is small enough to handle the final copy.

   Although pre-copying memory is performed iteratively, the execution of programs is not obviously interrupted.



Live VM migrations : Pre-Copy technique - 2

4. **Stop and transfer VM State**
   Suspend the VM and copy the last portion of the data. This happens when the last round's memory data is transferred.
   Other non-memory data such as CPU and network states should be sent as well. During this step, the VM is stopped and its applications will no longer run. This "***service unavailable***" time is called the "***downtime***" of migration, which should be as short as possible so that it can be negligible to users.
5. **Commitment**
6. **VM activation at the destination server**
   Commit and activate the new host.
   After all the needed data is copied, on the destination host, the VM reloads the states and recovers the execution of programs in it, and the service provided by this VM continues.
   Then the network connection is redirected to the new VM and the dependency to the source host is cleared.
   The whole migration process finishes by removing the original VM from the source host.



Live VM migrations : Pre-Copy technique - 3

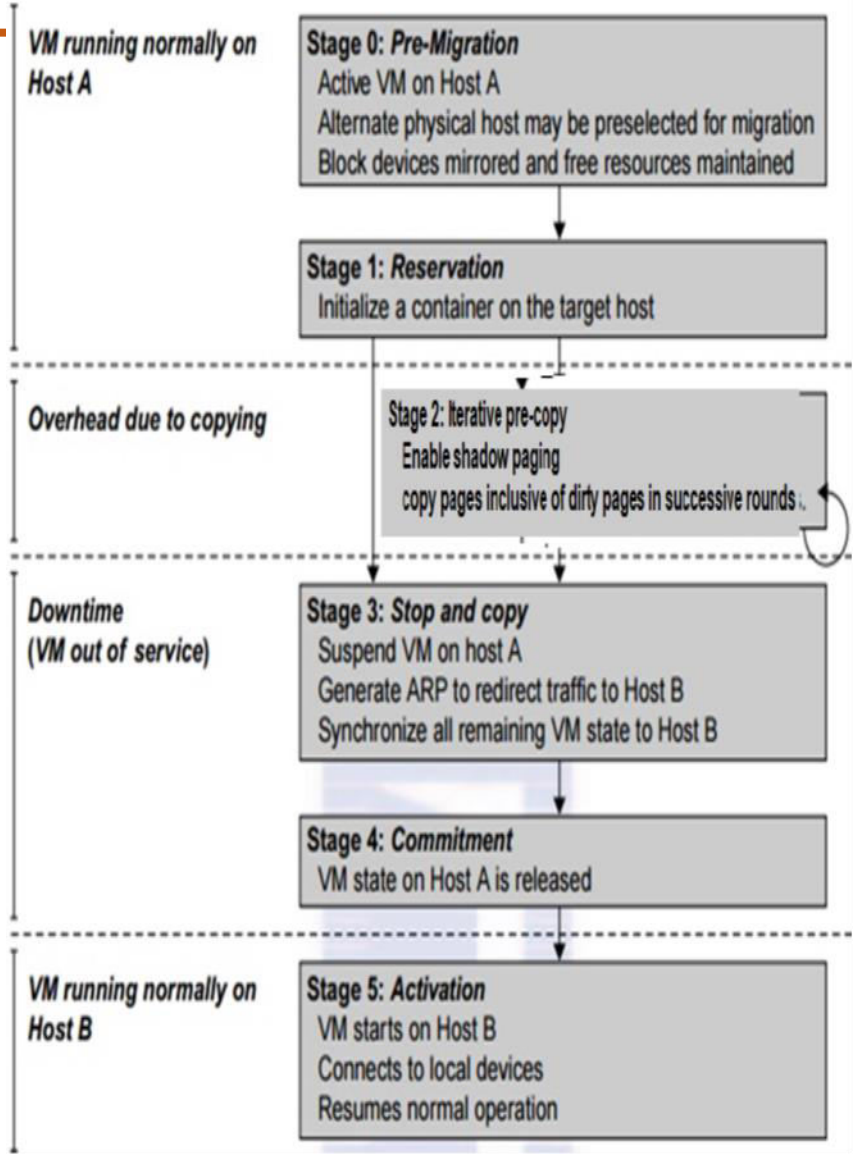
Advantages :

- Low VM downtime (required for copying the remaining dirty pages)
- Pre copy migration is advantageous in cases of few memory transfers

Disadvantage :

- Repeated copying of dirty pages can increase the whole migration time
- This could result in greater transfer time and downtime in case of more memory transfers

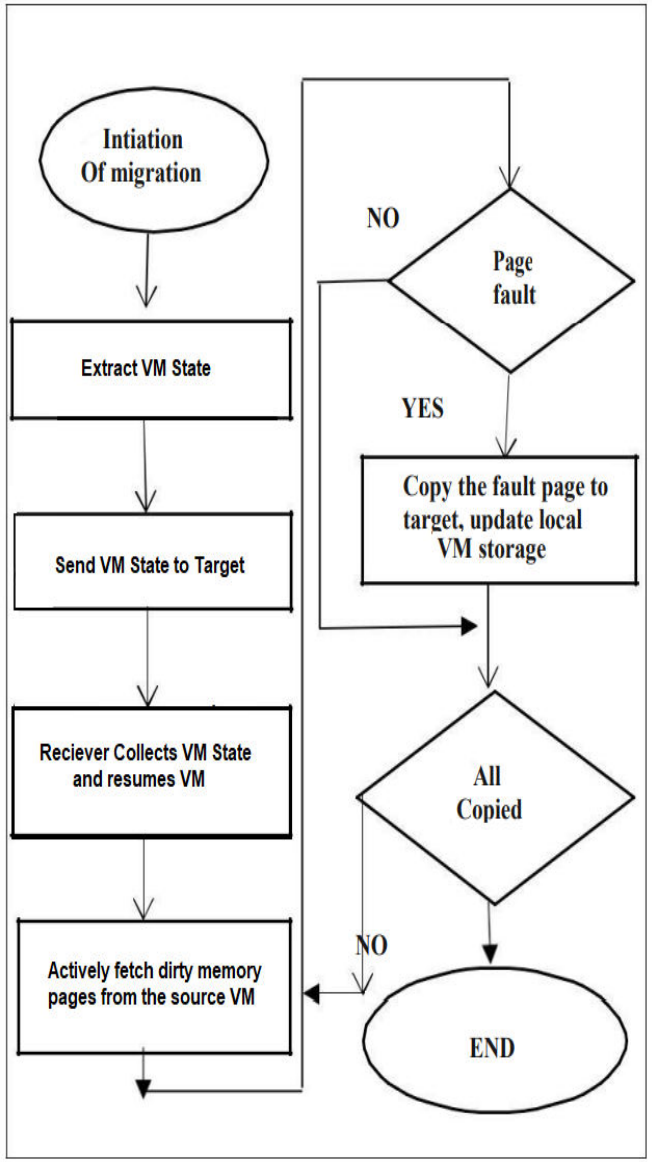
E.g. KVM, Xen, and VMware hypervisor use the pre-copy technique for live VM migration.



Live VM migrations : Post-Copy technique - 1

- In post-copy migration technique, processor state is transferred to the destination server before memory content
- The prior transfer of process states to the destination machine in this scheme enables the VM to resume instantly at the target machine.
- The memory contents of VM are copied to destination from source while VM is running on the target host almost at once.
- Memory faults are generated for the memory pages that have not been fetched yet, and where the missing pages are to be fetched from source machine.
- As excess and repetitive memory faults can result in significant disruption in the service quality, which is addressed by considering different techniques to lower the number of page faults.
  E.g. Demand Paging, Active Push, Memory Prepaging

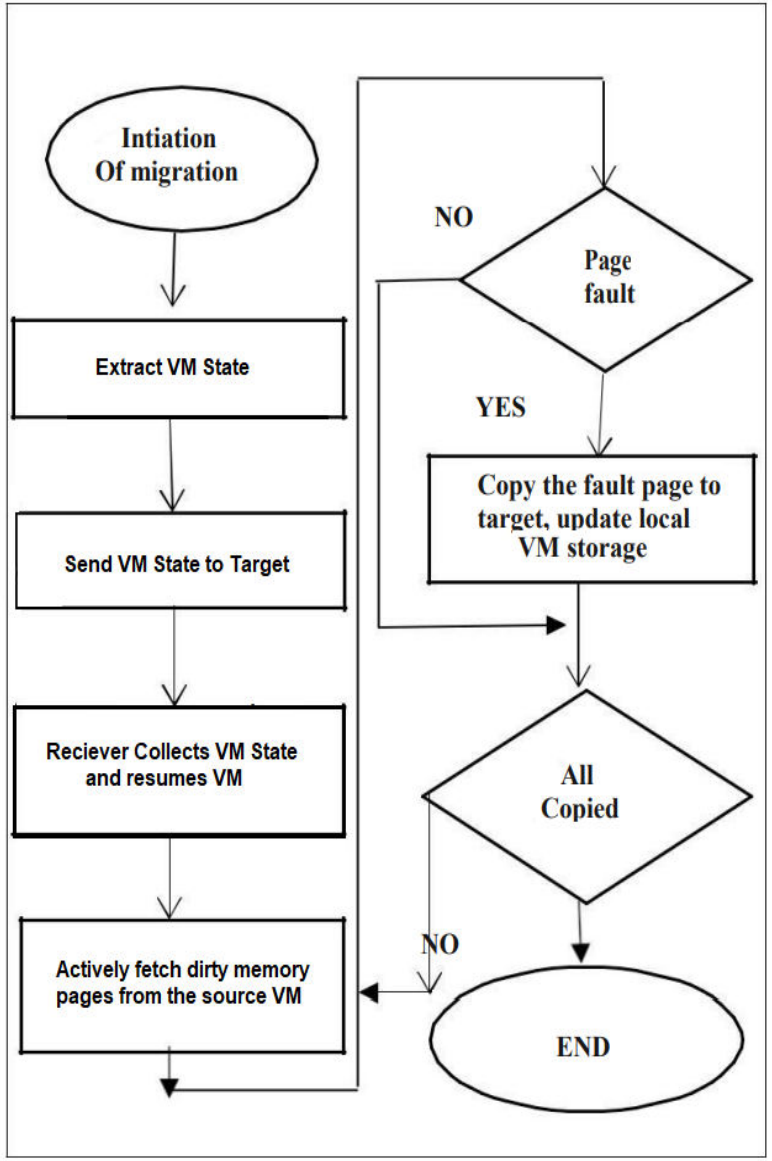
Demand paging: In this technique, when VM resumes on the target machine, requesting memory pages for read/write operation results in page faults, the faulty pages are serviced by retransmission from the source server. This servicing of faulty pages degrades the application performance but provides a simple and slowest option.



Live VM migrations : Post-Copy technique - 2

- ***Active push:*** It removes residual dependencies from the source server and it pro-actively pushes the VM pages to destination server even when VM is running on the destination server. If a page fault occurs at destination VM then demand paging is used to deal with fault. Therefore, pages are sent only once either via active push or demand paging.
- ***Memory prepaging*** approach assumes VM memory access in time and location and predicts the corresponding memory pages which has high demand probability and makes the page pushing to be more efficient and thus reducing memory faults

  These could have their service downtimes and total migration times affected by similar page re-transmission.



Live VM migrations : Post-Copy technique - 3

Disadvantages

- There is a challenge to reduce the overhead of Post copy migration technique occurs due to the generation of page faults at different time intervals during execution of VM at destination host. Each time, VM is suspended at destination host and waiting for re-transmission of required memory pages from source host to continue.
- There is a need to minimize the amount of data transfer (memory pages and CPU state) over the network, between clusters of hosts to save the money and time that is spent on network.
- The repetition in page fault detection is a main drawback of post copy approach, it should be optimized to reduce the number of cycles of memory re-transmission from source host it will increase the reliability of post copy migration technique

Issues to be handled with VM migration : Memory Migration

- Memory which needs to be migrated can be in a range of hundreds of megabytes to a few gigabytes in a typical system today, and it needs to be done in an efficient manner.
- The Internet Suspend-Resume (ISR) technique **exploits temporal locality** as memory states are likely to have considerable overlap in the suspended and the resumed instances of a VM. Temporal locality refers to the fact that the memory states differ only by the amount of work done since a VM was last suspended before being initiated for migration.
- Implementations use a representation of a tree of small subfiles which exists with both the suspended and resumed VM Instances. This caching helps to send only those which have changed, but the downtime may be high.

Refs: https://www.brainkart.com/article/Migration-of-Memory,-Files,-and-Network-Resources_11346/
https://www.researchgate.net/publication/3958192_Internet_suspendresume

Issues to be handled with VM migration : File System Migration (Cont.)

- To support VM migration, a system must provide each VM with a consistent, location-independent view of the file system that is available on all hosts.
- This may be achieved by providing each VM with its own virtual disk which the file system is mapped to and transport the contents of this virtual disk along with the other states of the VM. The current large capacity disks may make it feasible to send it over the network.

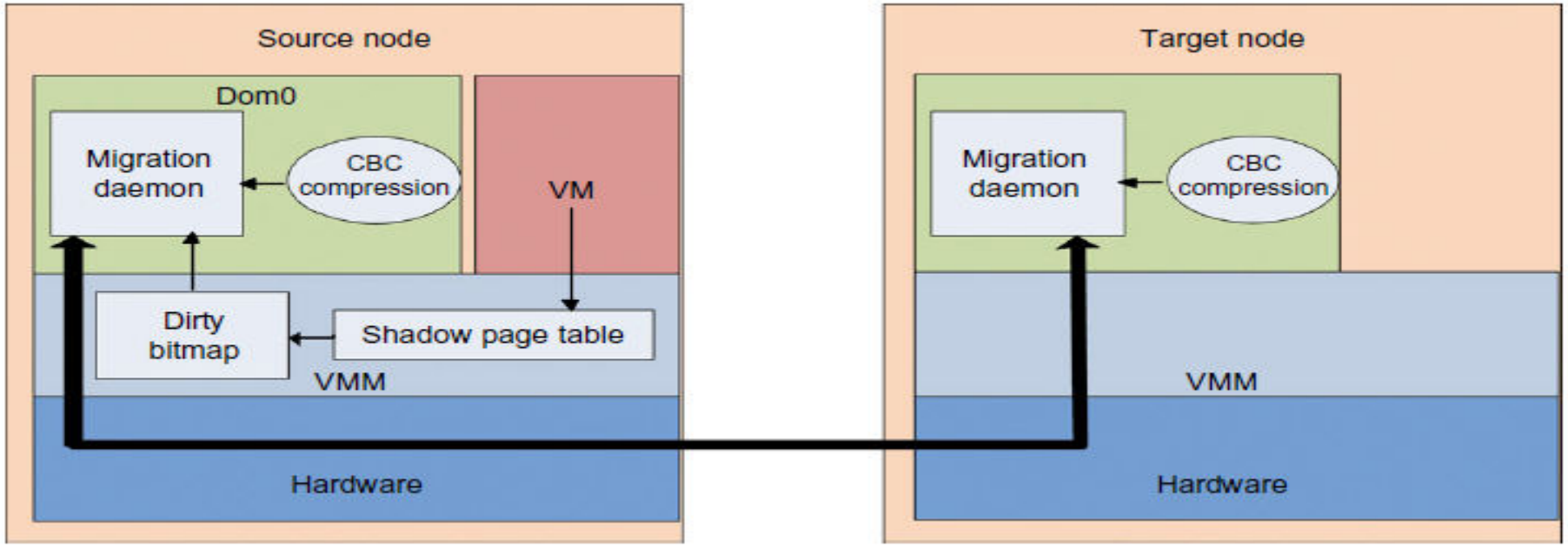
Another way is to have a global file system across all machines where a VM could be located. This way removes the need to copy files from one machine to another because all files are network accessible.

Issues to be handled with VM migration : Network Migration (Cont.)

Migrating VM should maintain all open network connections without relying on forwarding mechanisms on the original host or on support from mobility or redirection mechanisms.

- To enable remote systems to locate and communicate with a VM, each VM must be assigned a virtual IP address known to other entities. This address can be distinct from the IP address of the host machine where the VM is currently located.
- Each VM can also have its own distinct virtual MAC address.
- The VMM maintains a mapping of the virtual IP and MAC addresses to their corresponding VMs.
- In general, a migrating VM includes all the protocol states and carries its IP address with it.

Example: Live migration of VM between two Xen-enabled hosts



Domain 0 (or Dom0) performs tasks to create, terminate, or migrate to another host. Xen uses a send/recv model to transfer states across VMs.

Live migration of VM between two Xen-enabled hosts explained

- Migration daemons running in the management VMs are responsible for performing migration.
- Characteristic Based Compression (**CBC**) algorithm compresses the memory pages adaptively.
  - **Adaptive compression** is a type of data **compression** which changes compression algorithms based on the type of data being compressed.
- Shadow page tables in the VMM layer trace modifications to the memory page in migrated VMs during the precopy phase. Corresponding flags are set in a dirty bitmap
- At the start of each precopy round, the bitmap is sent to the migration daemon. Then, the bitmap is cleared and the shadow page tables are destroyed and re-created in the next round.
- The system resides in Xen's management VM. Memory pages denoted by bitmap are extracted and compressed before they are sent to the destination.
- The compressed data is then decompressed on the target.

Additional Reading

https://www.cse.iitb.ac.in/~cs695/papers/livemigration.pdf

How to migrate on premise VM to AWS
https://www.youtube.com/watch?v=GTHxL8U0PiI

How to migrate your VM to AWS
https://www.youtube.com/watch?v=_SpRpC2Ez9c

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

CLOUD COMPUTING

Lightweight virtualization – Containers, namespaces, cgroups
&
Deployment of cloud native applications through Docker, UnionFS

Dr. H.L. Phalachandra

Prof. Venkatesh Prasad

Department of Computer Science and Engineering

Acknowledgements:

Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

Containers – Definitions

- Linux Containers or LXC is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) which can run multiple workloads, on a control host running a single linux OS instance
- LXC provides a virtual environment that has its own process and network space and does not create a full-fledged virtual machine.
- LXC uses Linux kernel cgroups and namespace isolation functionality to achieve the same.

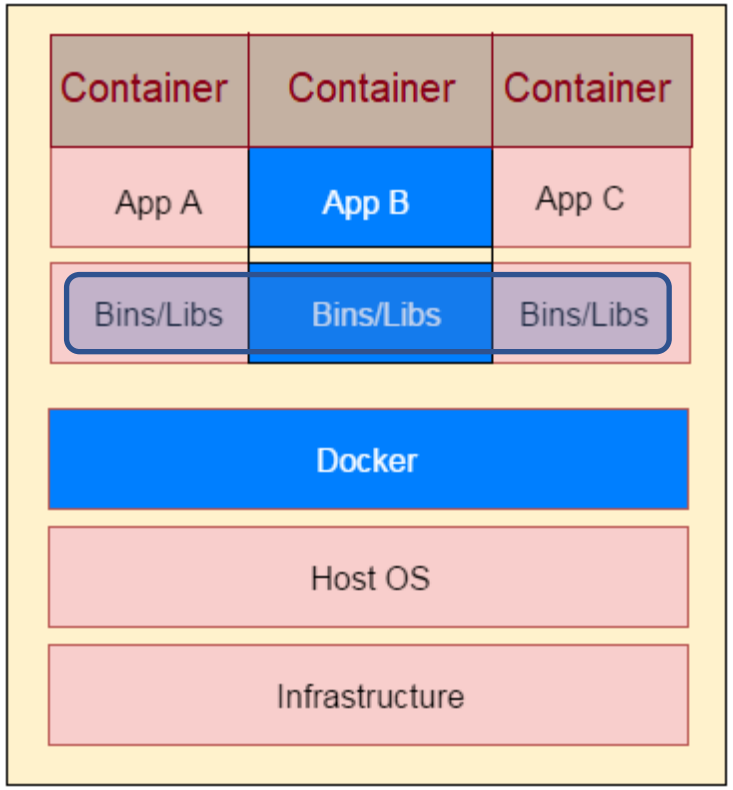
Motivation

- VMs support the objective of virtualizing the physical systems and allowing multiple tenants/applications to be isolated and share the physical resources along with Access control
- One of the challenges as observed is, this isolation achieved by or provided by VM is expensive
- Traditional OSs supporting multiple application processes, but share a disk, with all the processes capable of seeing the entire filesystem with access control built on top, and also share a network.
- Containers using a light weight mechanism, provide this virtualization extending the isolation provided by our traditional OS

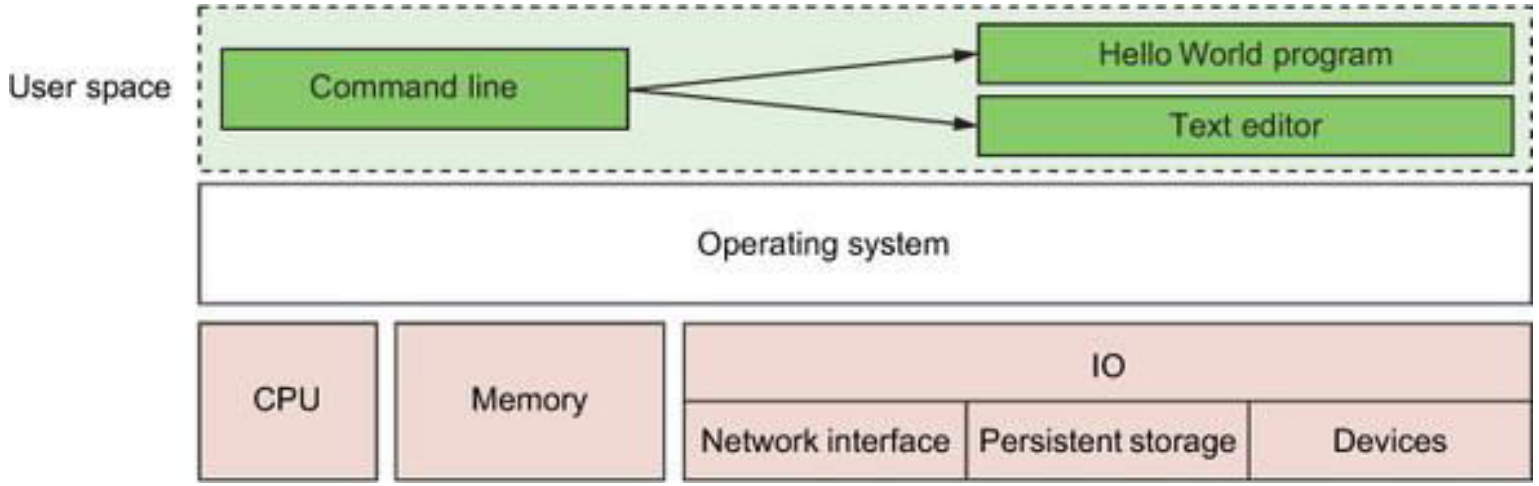
Lxc (tools) is an user space toolset for creating & managing Linux Containers – different from LXC Linux containers

Container Characteristics

- Containers sit on top of a physical server and its host OS.
- Each container shares the host OS kernel and, usually, the binaries and libraries too, but as read-only. This reduces the management overhead where a single OS needs to be maintained for bug fixes, patches, and so on.
- Containers are thus exceptionally “light”—they are only megabytes in size and take just seconds to start, versus gigabytes and minutes for a VM.
  - Container creation is similar to process creation and it has speed, agility and portability.
  - Thus containers have higher provisioning performance



Containers (Cont.)



A basic computer stack running two programs that were started from the command line

- Notice that the command-line interface, or CLI, runs in what is called user space memory, just like other programs that run on top of the operating system.
- Ideally, programs running in user space can't modify kernel space memory.
- Broadly speaking, the operating system is the interface between all user programs and the hardware that the computer is running on.

Contrasting Containers vs VMs

1. Each VM includes a separate OS image, which adds overhead in memory and storage footprint.

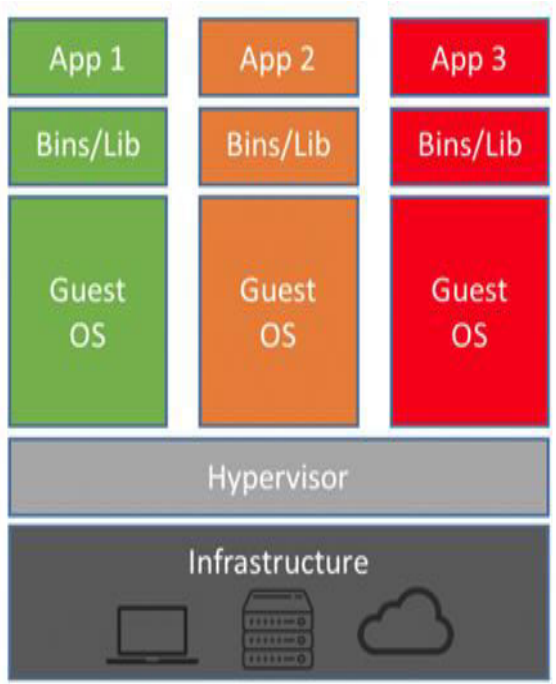
   Containers reduce management overhead as they share a common OS, only a single OS needs to be maintained for bug fixes, patches, and so on.

2. In terms of performance, VMs have to boot when provisioned making it slower, and also have I/O performance overhead

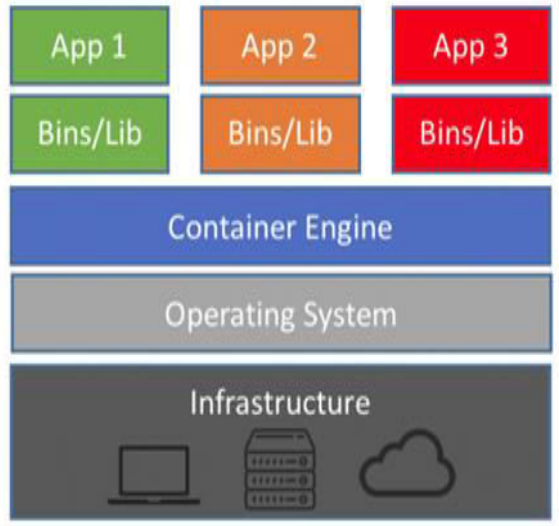
   Containers have higher performance as its creation is similar to process creation, so boots quickly and it has speed, agility and portability

3. VMs are more flexible as Hardware is virtualized to run multiple OS instances.

   Containers run on a single OS and also can support only Ubuntu containers of that type of OS where its running or containers cannot be of different OS variants



Machine Virtualization



Containers

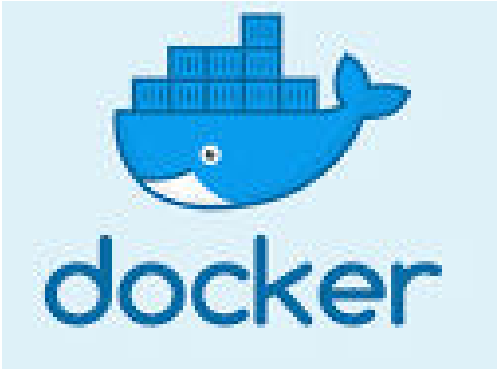
4. VMs consume more resources and come up slower than Containers which come up more quickly and consume fewer resources

CLOUD COMPUTING

VM vs Docker

Virtual Machine	Docker Container
Hardware-level process isolation	OS level process isolation
Each VM has a separate OS	Each container can share OS
Boots in minutes	Boots in seconds
VMs are of few GBs	Containers are lightweight (KBs/MBs)
Ready-made VMs are difficult to find	Pre-built docker containers are easily available
VMs can move to new host easily	Containers are destroyed and re-created rather than moving
Creating VM takes a relatively longer time	Containers can be created in seconds
More resource usage	Less resource usage

Docker



- Docker is an open platform tool which makes it easier to create, test, ship, deploy and to execute applications using containers.
- Docker containers allow us to separate the applications from the infrastructure enabling faster deployment of applications/software
- It significantly reduces the between writing code and running it in production by providing methodologies for shipping, testing and deploying code quickly
- It can be considered as a tool that helps to package and run an application in a loosely isolated environment called a container.
- It could be looked at as a PaaS product that uses OS level virtualization to deliver S/W packages
- Docker provides the isolation and security to allow many containers to run on a single server or virtual machine
- Its typical to find between 8 -18 containers running simultaneously on a single server/VM

Docker (Cont.)

- Containers created using Docker can contain everything needed to run an application, so you do not need to rely on what is currently installed on the host system
- Docker allows these containers to be shared in a way that it would work identically.
- Docker can be used with network applications such as web servers, databases, mail servers, with terminal applications like text editors, compilers, network analysis tools, and scripts.
  - In some cases, it's even used to run GUI applications such as web browsers and productivity software.
- Docker runs on Linux software on most systems.
  - Docker is also available as a native application for both macOS and Windows
  - Docker can run native Windows applications on modern Windows server machines.

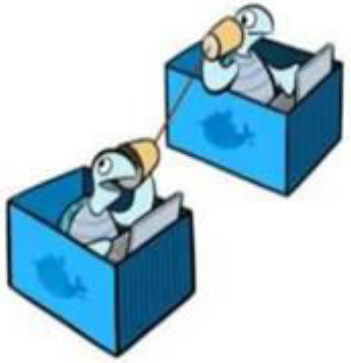
Docker Benefits

Portability, Shipping Applications

One App =

- binaries (exec, libs, etc.)
- data (assets, SQL DB, etc.)
- configs (/etc/config/files)
- logs

either in a container
or a *composition*



docker

Docker, Containers, and the Future of Application Delivery

Portability

Docker Promise: Build, Ship, Run !

- reliable deployments
- develop here, run there

docker

Docker, Containers, and the Future of Application Delivery

Build

Develop an app using Docker containers with any language and any toolchain.

Ship

Ship the "Dockerized" app and dependencies anywhere - to QA, teammates, or the cloud - without breaking anything.

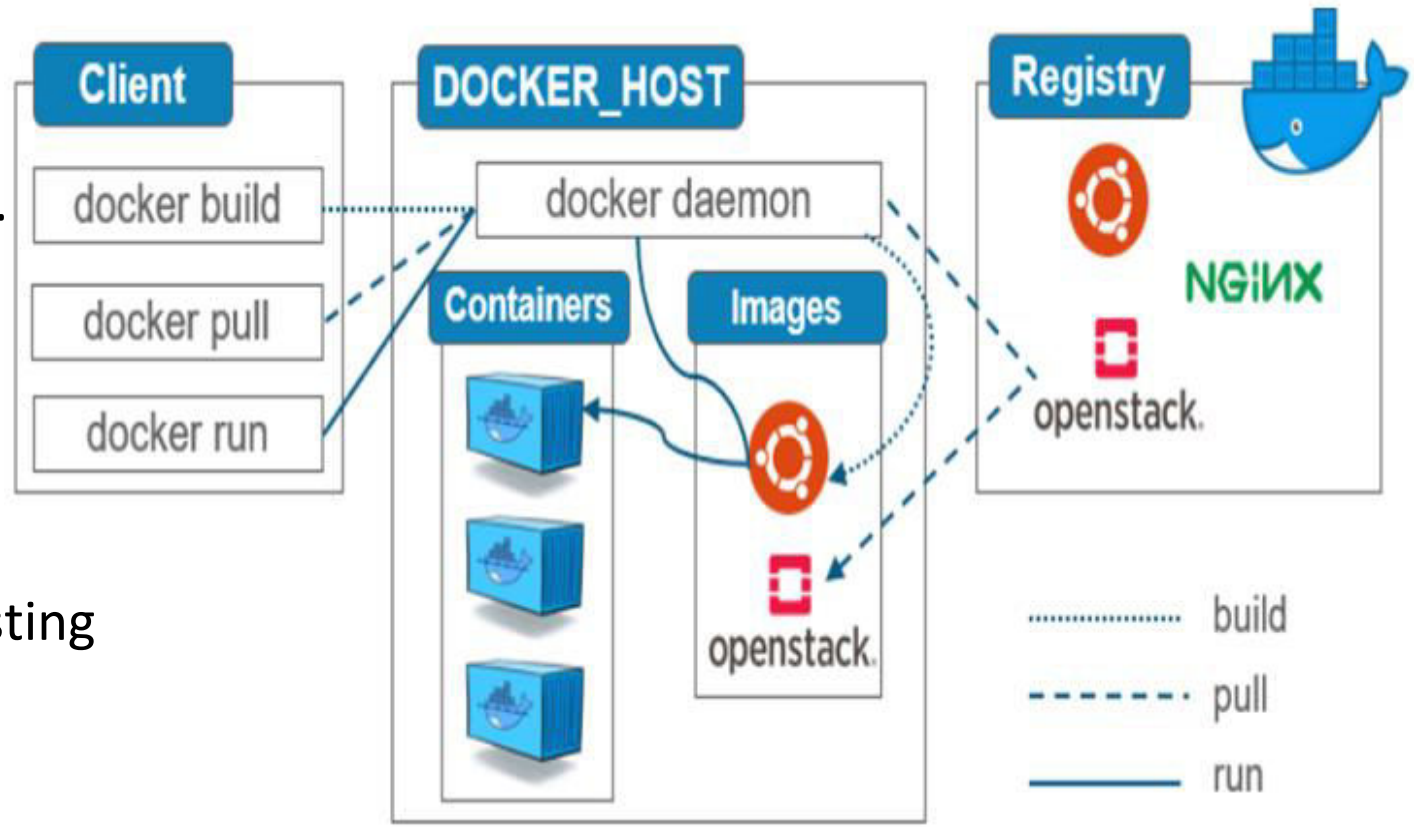
Run

Scale to 1000s of nodes, move between data centers and clouds, update with zero downtime and more.

Developers use Version Control Systems (VCS) like Git. DevOps also uses VCS for docs, scripts and Dockerfiles.
DevOps could use Dockerfile to describes how to build the image, and something like docker-compose.yml to describe how to orchestrate them.

Docker Architecture

- Docker uses a client-server architecture.
- The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.
- Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

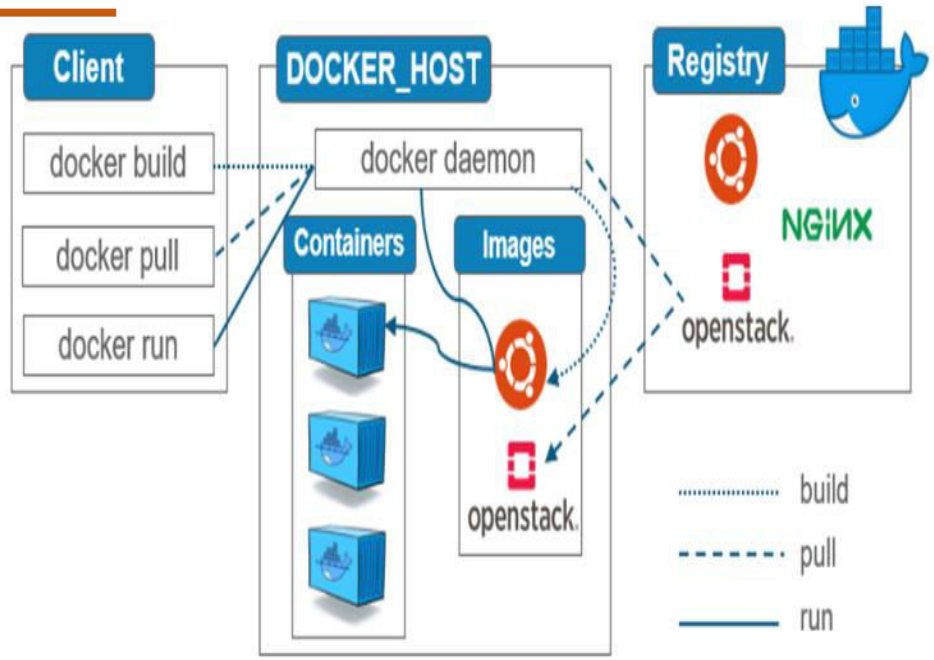


Reference:https://docs.docker.com/get-started/overview/

Docker Architecture

The Docker daemon

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
- A daemon can also communicate with other daemons to manage Docker services.



The Docker client

- The Docker client (docker) is the primary way that many Docker users interact with Docker.
- When you use commands such as **docker run**, the client sends these commands to 'dockerd' (docker daemon), which carries them out.
- The docker command uses the Docker API.
- The Docker client can communicate with more than one daemon.

Docker Architecture

Docker Host

- Docker Host has the Docker Daemon running and can host (like a private hub/registry) or connect (like to a public docker_hub/registry) to a Docker Registry which stores the Docker Images.
- The Docker Daemon running within Docker Host is responsible for the Docker Objects images and containers.

Docker Objects :

There are objects like

- Images
- Containers
- Networks
- Volumes
- Plugins and other objects which are created and used while using docker.

Docker Objects : Images

Images : This is an is a read-only template with instructions for creating a Docker container. This could be based on another image (available in the registry) with additional customizations. Eg. Image for a Webserver .. Original image of Ubuntu customized with installation and configuration of the webserver which is created only as a read-only image which can be deployed and an application can be run in the same.

This is done using a ***Dockerfile*** a script file which defines the syntax to indicate steps needed to create the image (and run using a run command).

Each instruction in a Dockerfile creates a ***layer*** in the image.

These Dockerfiles are distributed along with software that the author wants to be put into an image. In this case, you're not technically installing an image. Instead, you're following instructions to build an image.

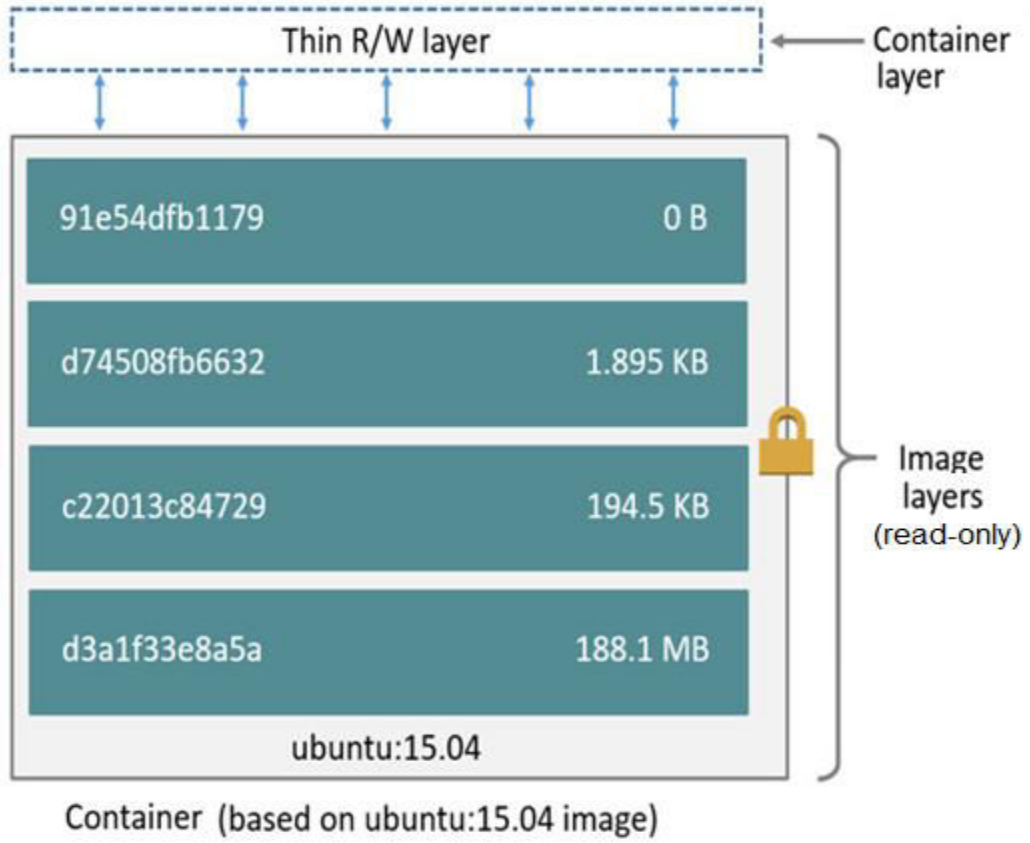
Docker Objects : Images – More on Dockerfile

- Distributing a Dockerfile is similar to distributing image files using your own distribution mechanisms. A common pattern is to distribute a Dockerfile with software from common version-control systems like Git.
  - If you have Git installed, you can try this by running an example from a public repository:
    - git clone https://github.com/dockerinaction/ch3_dockerfile.git
    - docker build -t dia_ch3/dockerfile:latest ch3_dockerfile

  In this example, you copy the project from a public source repository onto your computer and then build and install a Docker image by using the Dockerfile included with that project. The value provided to the -t option of docker build is the repository where you want to install the image.
- Building images from Dockerfile's is a light way to move projects around that fits into existing workflows. Could lead to an unfavourable experience in case of a drift in dependencies between the time when the Dockerfile was authored and when an image is built on a user's computer.
- Docker Images can be removed or cleaned up with
  - docker rmi dia_ch3/dockerfile
  - rm -rf ch3_dockerfile

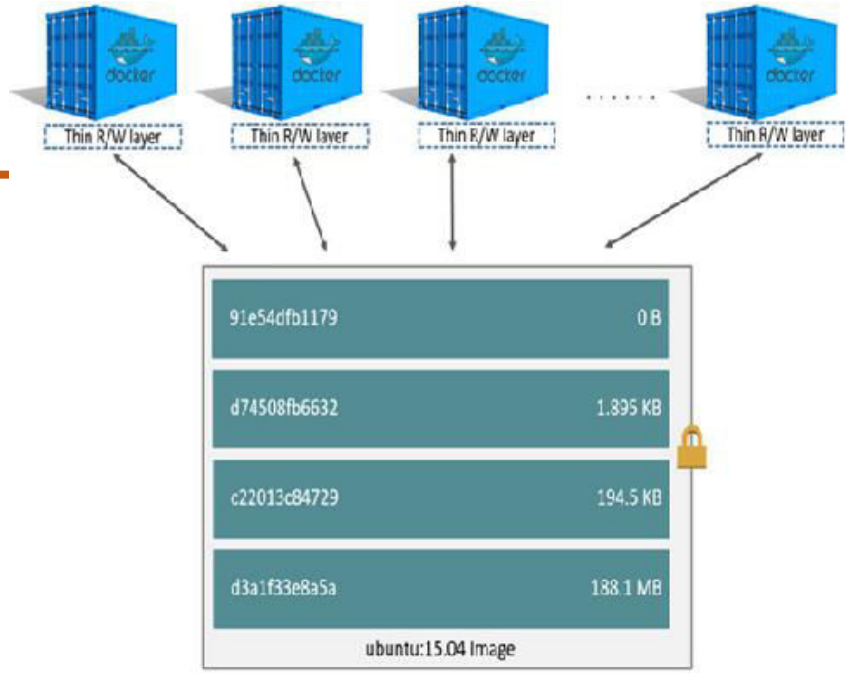
Docker Commands – Video Ref: https://www.youtube.com/watch?v=nXV6qihj5uw

Docker Objects : Images - layers



- Specification for a Docker Image is stored in Dockerfile
  - Should be only one for a container
  - Only the definition of the image
- Image is built from Dockerfile
- Specifies the read-only file systems in which various programs are installed E.g. web server + libraries
- Each instruction in a Dockerfile creates a ***layer*** in the image.
- A layer is set of files and file metadata that is packaged and distributed as an atomic unit.
  - Internally, Docker treats each layer like an image, and layers are often called intermediate images.
  - You can even promote a layer to an image by tagging it.
  - Most layers build upon a parent layer by applying filesystem changes to the parent.
  - Whenever there is a change in the Dockerfile and the image is rebuilt, only the ***incremental changes are rebuilt*** (making it light weight, small and fast when compared to other virtualized technologies)

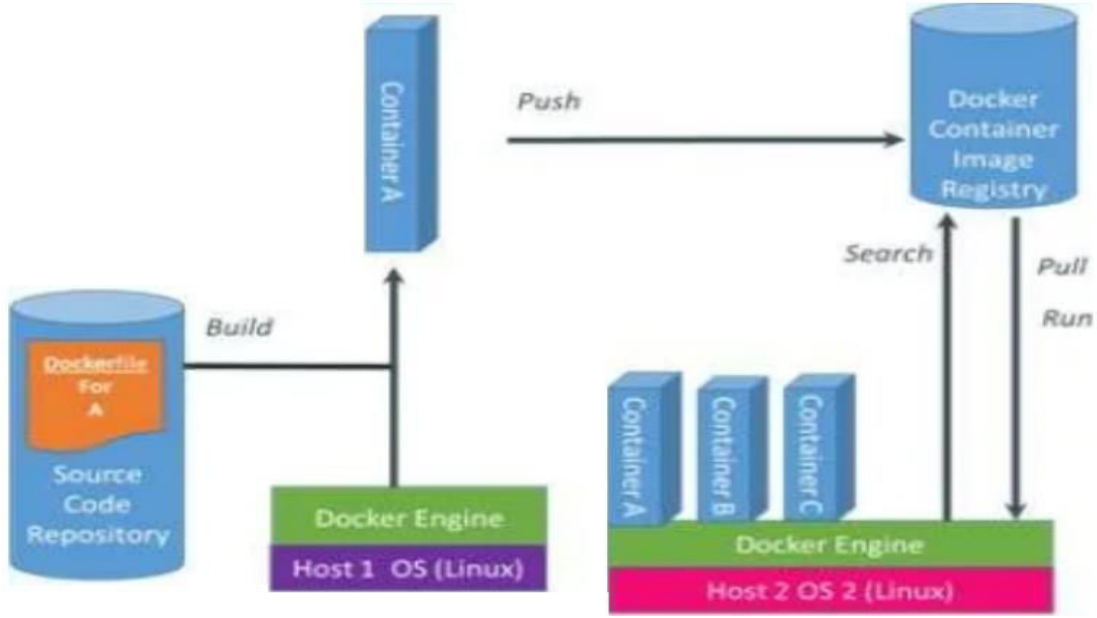
Docker Objects : Images & Docker Registries



- Image + temporary R/W file system
  - Used as temporary storage
  - Deleted when container is destroyed
- Multiple containers can use the same image & their own temporary storage

Docker registries:

- A Docker registry stores Docker images.
- ***Docker Hub*** is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default.
- You can even run your own ***private registry***.
- When you use the **docker pull or docker run** commands, the required images are pulled from your configured registry.
- When you use the docker push command, your image is pushed to your configured registry.



https://docs.docker.com/storage/storagedriver/

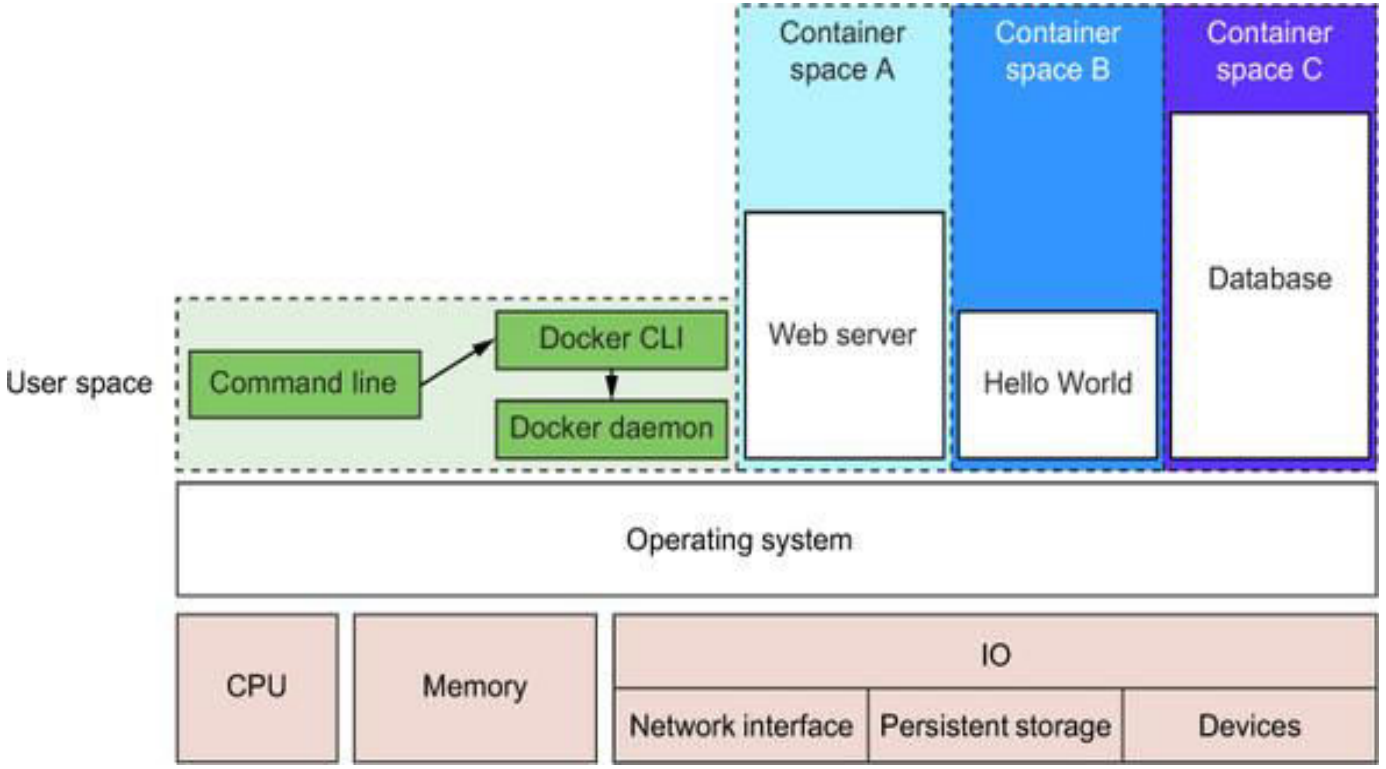
Docker Objects : Containers

- Docker Containers are the ready applications created from Docker Images. **Or** you can say they are running instances of the Images and they hold the entire package needed to run the application **Or** it could also be looked at as a runnable instance of an image **Or** an execution environment (sandbox).

- We can create, start, stop, move, or delete a container using the Docker API or CLI.
- A container can be connected to one or more networks. Storage could be attached to it, or even new image could be created based on its current state.
- A container is relatively well isolated from other containers and its host machine and this isolation can also be controlled. Processes in the container cannot access non-shared objects of other containers, and can only access a subset of the objects (like files) on the physical machine.
- Docker creates a set of name spaces when a container is created. This name spaces restrict what the objects processes in a container can see e.g. a subset of files
- A container is defined by its image and the configuration options provided to it when its created or started. When a container is removed, any changes to its state that are not stored in persistent storage will disappear

Docker running three containers on a basic Linux computer system

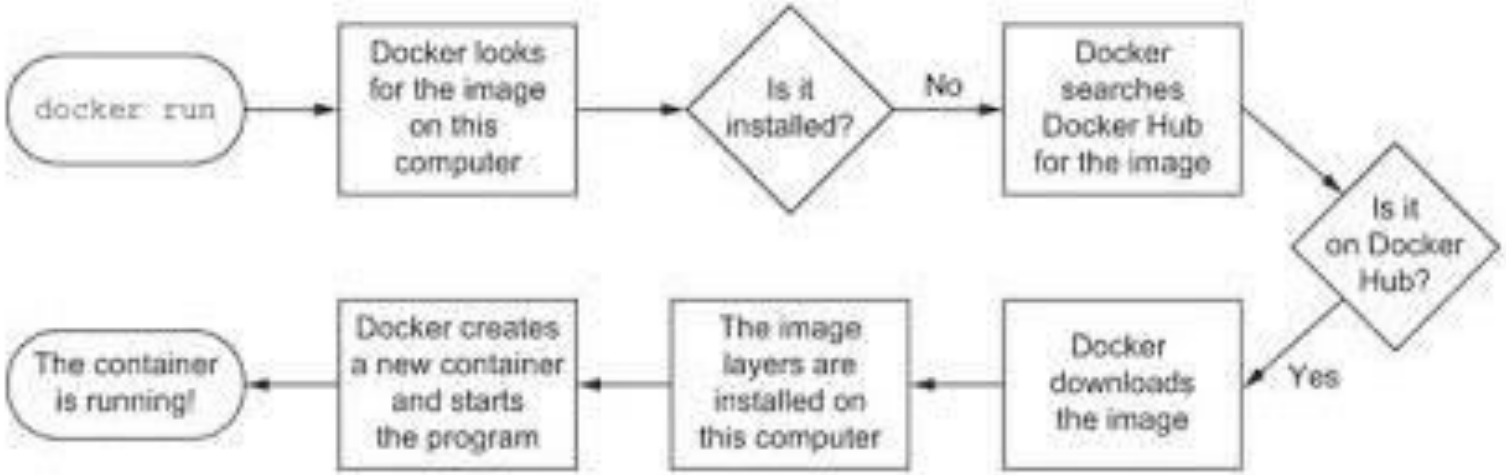
- Running Docker means running two programs in user space.
  - The first is the Docker engine that should always be running.
  - The second is the Docker CLI. This is the Docker program that users interact with to start, stop, or install software
- Each container is running as a child process of the Docker engine, wrapped with a container, and the delegate process is running in its own memory subspace of the user space.
  - Programs running inside a container can access only their own memory and resources as scoped by the container.



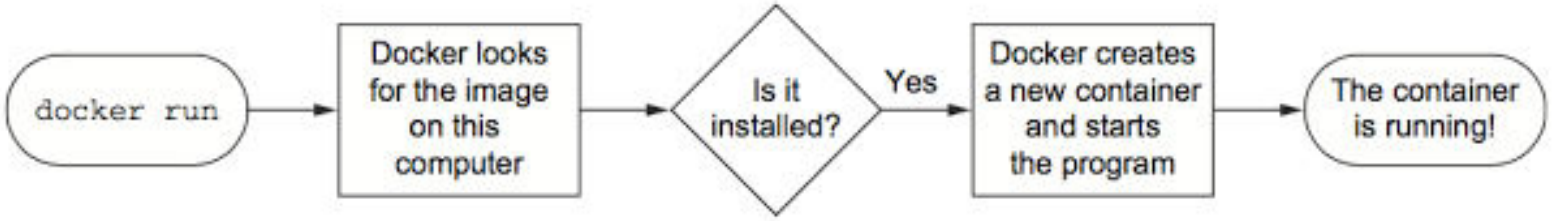
Docker can be looked at as a set of PaaS products that use OS-level virtualization to deliver software in packages called containers.

Running a program with Docker

- What happens after running **docker run**
- The image itself is a collection of files and metadata which includes the specific program to execute and other relevant configuration details

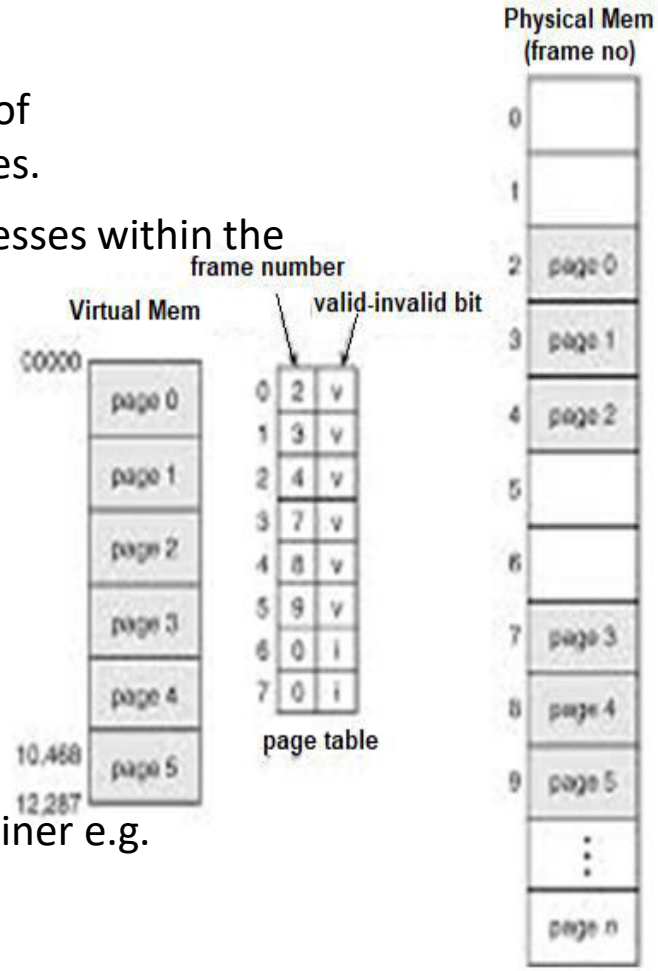


- Running docker run a second time.
- The image is already installed, so Docker can start the new container right away.



Namespaces – What's in a Name in CS?

- If you can't name an object, you can't access it. E.g. Web site – if name is hidden, can't access
- Paging : Processes can access only pages in its name spaces but cannot access physical page 1 (i.e. frame number 1 in the diagram)
- Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources and another set of processes sees a different set of resources.
- A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource
- The feature works by having the same namespace (with a name) for a group of resources and processes, but those namespaces refer to *distinct resources*
- A physical computer can have more than one namespaces (two or more)
- All the resources that a process sees can be considered a ***namespace***
  - The files seen by a process is the ***file namespace***
  - The network connections are part of ***network*** **namespace**
- Container Access is restricted to only subset of objects (e.g., files) on the physical machine using these namespaces. Restriction is applied on what can be seen by the object processes in a container e.g.
  - To restrict process (and container) to a subset of files use file namespace



Docker used Namespaces

- Docker builds containers at runtime using 10 major system features. Docker commands can be used to illustrate and modify features to suit the needs of the contained software and to fit the environment where the container will run.
- The specific features are as follows (a few of then are discussed in a little more detail):
  - PID namespace—Process isolation through an identifiers (PID number) – not aware of what happens outside its own processes
  - UTS namespace—allows for having multiple hostnames on a single physical host - Host and domain name (as other things like IP can change)
  - MNT namespace—isolate a set of mount points such that processes in different namespaces cannot view each others files. (almost like chroot) (Filesystem access & structure)
  - IPC namespace—provides isolation to container process communication over shared memory and having an ability to invite other processes to read from the shared memory

https://www.redhat.com/sysadmin/7-linux-namespaces

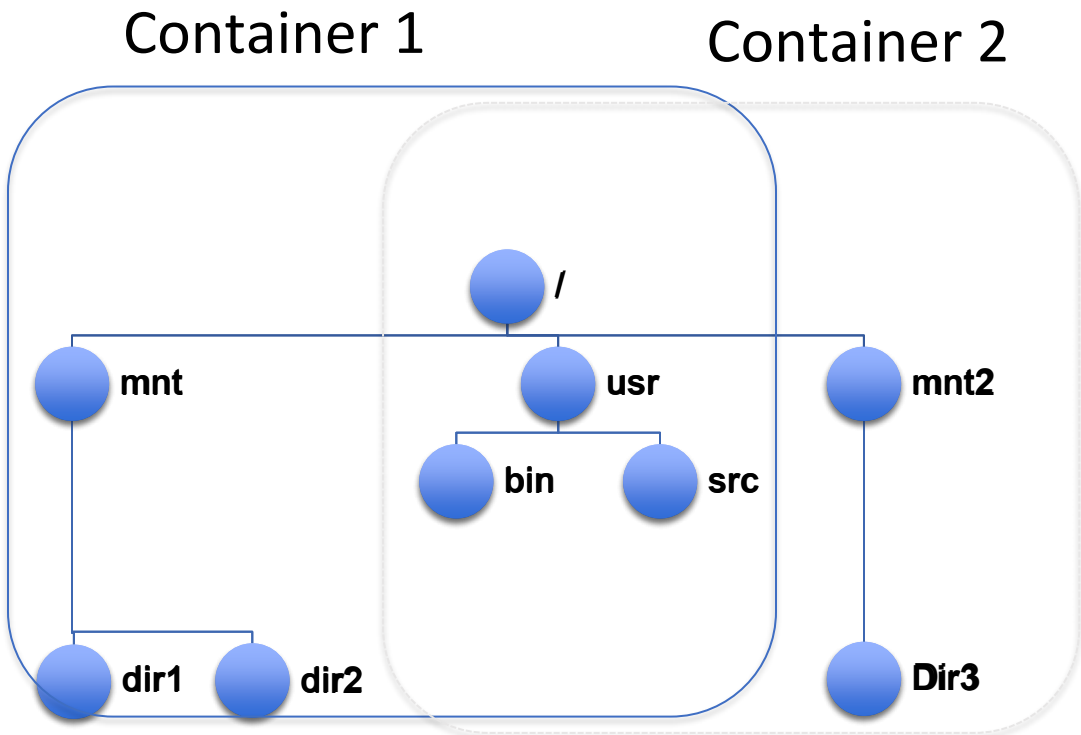
Docker used Namespaces (Cont.)

- NET namespace—allow processes inside each namespace instance to have access to a new IP address along with the full range of ports - Network access and structure
- USR namespace—provides user name-UID mapping isolating changes in names from the metadata within a container
- chroot syscall—Controls the location of the filesystem root
- cgroups—Controlling and accounting resources –rather than a hierarchical cgroup creation, there is a new cgroup with a root directory created to isolate resources and not allow navigation
- CAP drop—Operating system feature restrictions
- Security modules—Mandatory access controls

https://www.redhat.com/sysadmin/7-linux-namespaces

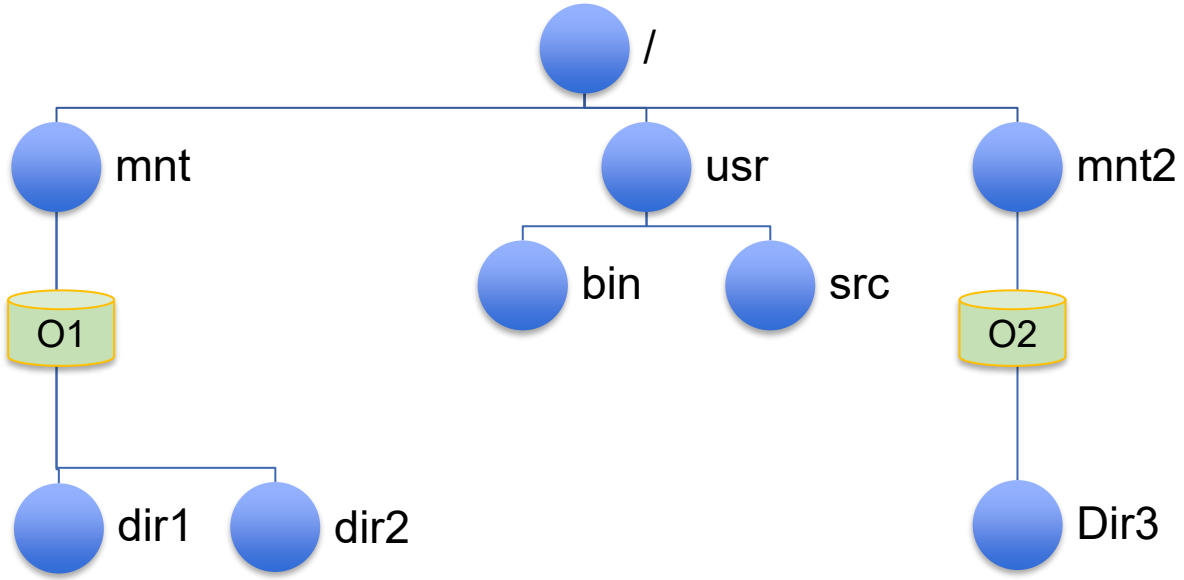
Access Control of objects : Illustration of name space (in this case MNT)

- Processes
  - in container 1 can access
    - Shared files in /usr
    - Non-shared /mnt
    - Cannot access /mnt2
  - in container 2 can access
    - Shared files in /usr
    - Non-shared /mnt2
    - Cannot access /mnt
- These are two different namespaces



Linux Filesystems before Namespaces

- */* is the *root file system*
  - Contains *vmunix* – the OS
  - *usr, bin, src* are all subdirectories
- *O1, O2* are volumes containing different versions of an application (e.g., Oracle)
- Mounted at */mnt* and */mnt2*
- This namespace is visible to all processes
- Access to files and directories controlled by access rights



Linux Filesystems after Namespaces

- File namespace: *mount namespace*
- *O1, O2* are volumes containing different versions of an application (e.g., Oracle)
  - Mounted at */mnt* and */mnt2*
- This namespace is visible to all processes
- Access to files and directories controlled by access rights
- Which namespace is process in?
  - Look at */proc/pid/ns*
  - *ls –l* for */proc/pidx/mnt* may show as below (4026531840 is the namespace id)
    - lrwxrwxrwx. 1 mtk mtk 0 Jan 8 04:12 mnt -> mnt:[4026531840]

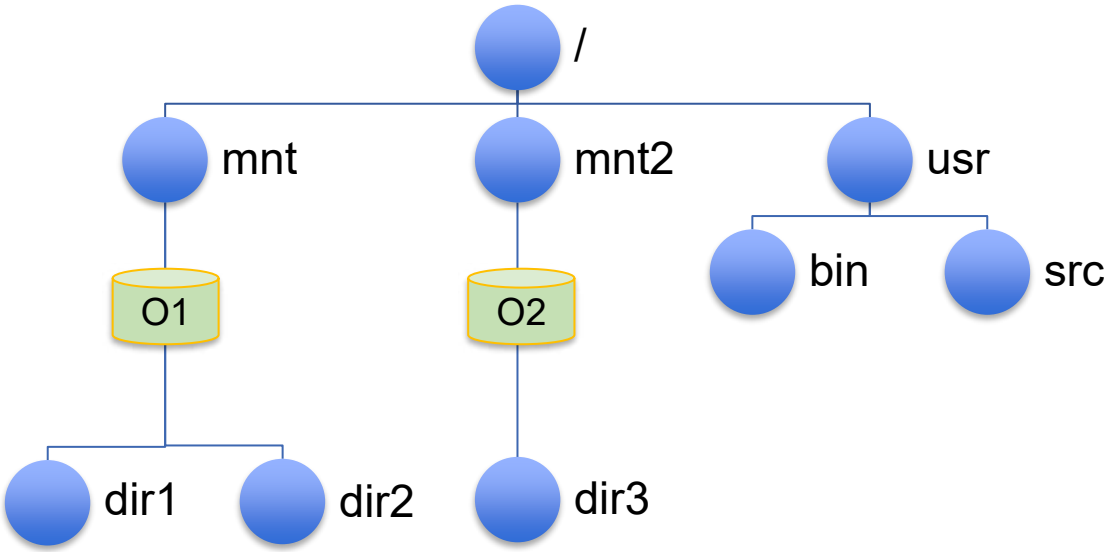


Illustration of Namespace Operations : E.g. MNT Namespace

Creation of Namespace

- To create a namespace, must create a process with that namespace
- *pid = clone(childFunc, stackTop, CLONE_NEWNS | SIGCHLD, argv[1]);*
- Creates a new child, like *fork()*
- *NEWNS* flag indicates that child has a new mount namespace
- Child can do *mount* and *umount* to modify namespace

Programs to be joining into a namespace

Allows program to join an existing namespace

int setns

- (int fd, // namespace to join
- int nstype); // type of ns

int unshare (int flags); // which namespace

- CLONE_NEWNS specifies mount namespace
- Similar to clone(); allows caller to create a new namespace

UTS (Unix Time Sharing) Namespace

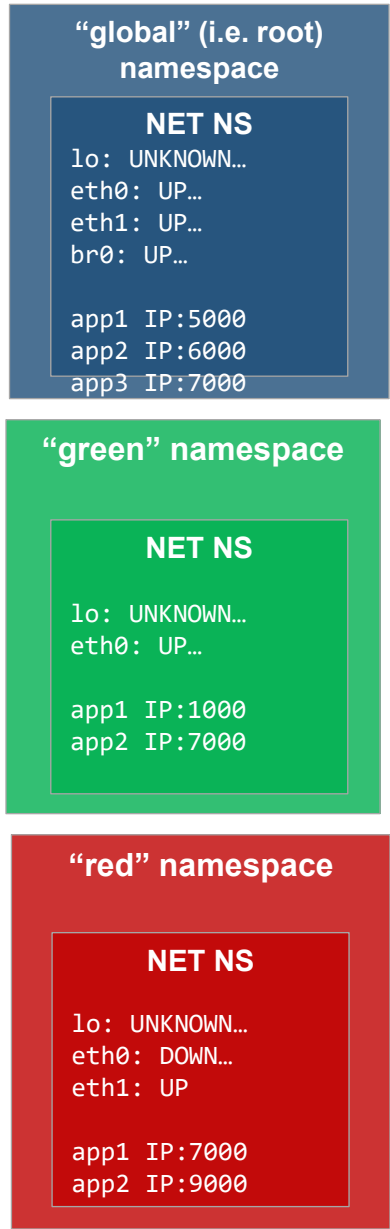
- Per namespace
  - Hostname
  - NIS domain name (Network Information Service)
- Reported by commands such as hostname
- Processes in namespace can change UTS values – only reflected in the child namespace
- Allows containers to have their own FQDN (Fully qualified Domain Name)

UTS namespace is about isolating hostnames.
The **UTS namespace** is used to isolate two specific elements of the system that relate to the uname system call.
The **UTS**(UNIX Time Sharing) **namespace** is named after the data structure used to store information returned by the uname system call.

CLOUD COMPUTING

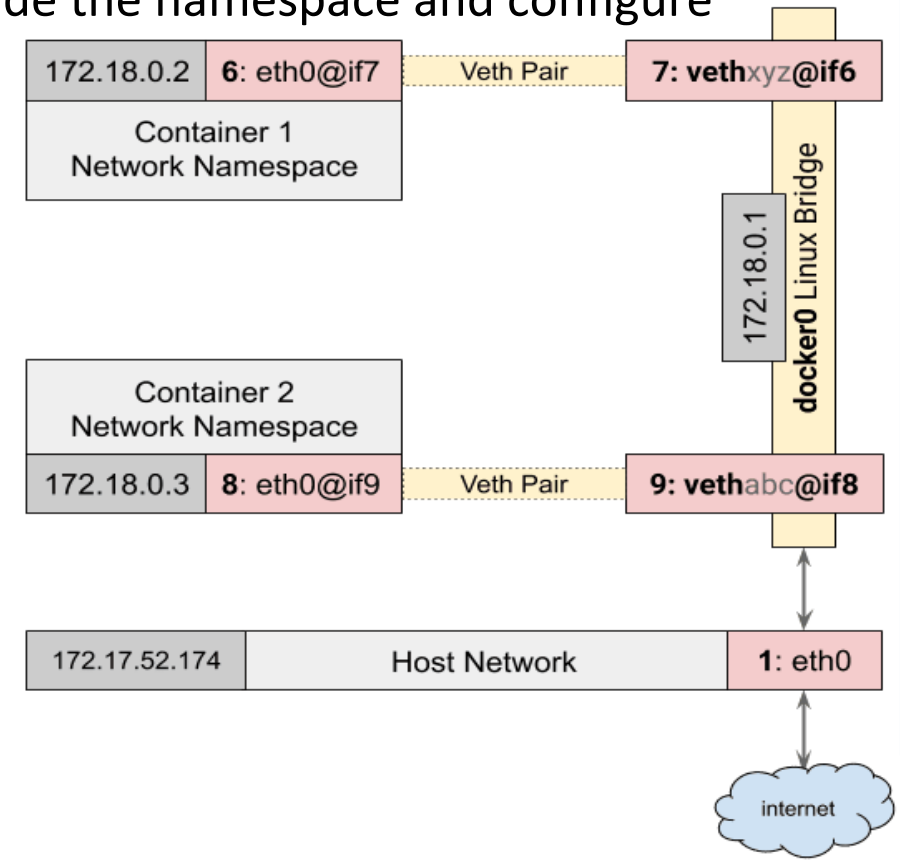
NET namespace

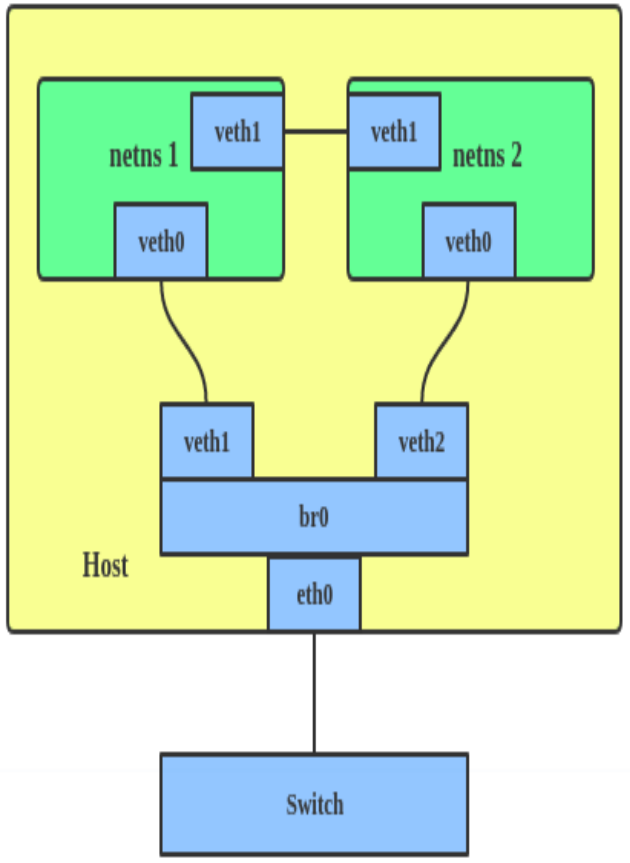
- It's a virtual network barrier encapsulating a process to isolate its network connectivity (in/out) and resources (i.e. network interfaces, route tables and rules) from linux core and other processes.
- It allow processes inside each namespace instance to have access to per namespace network objects
  - Network devices (ethernets)
  - Bridges
  - Routing tables
  - IP addresses
  - ports
- Various commands support network namespace such as ip



NET

- A VETH (virtual Ethernet) configuration as shown can be used when namespaces need to communicate to the main host namespace or between each other.
- veths – create veth pair, move one inside the namespace and configure
- Acts as a pipe between the 2 namespaces
- The VETH (virtual Ethernet) device is a local Ethernet tunnel.
- Devices are created in pairs
- Packets transmitted on one device in the pair are immediately received on the other device.
- When either device is down, the link state of the pair is down.

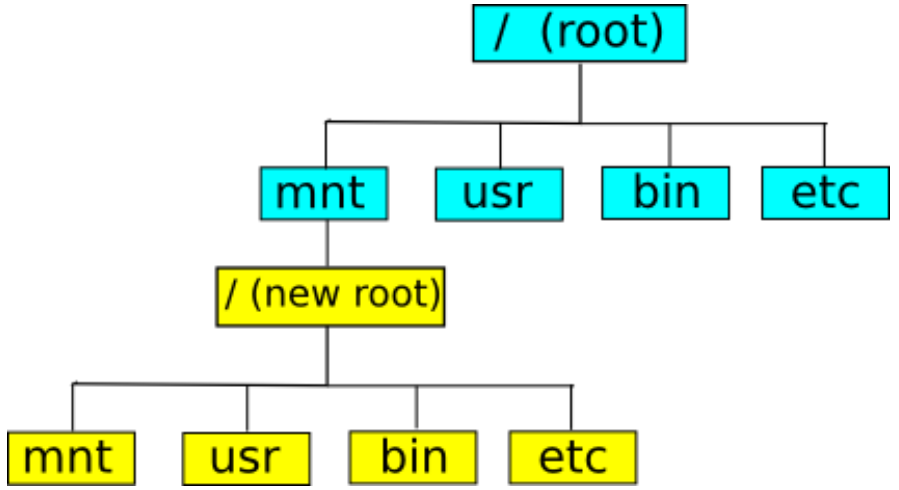




Ref: https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking/#:~:text=by%20Sabrina%20Dubroca.-,VETH,of%20the%20pair%20is%20down
https://dev.to/polarbit/how-docker-container-networking-works-mimic-it-using-linux-network-namespaces-9mj#:~:text=A%20%22veth%20pair%22%20is%20basically,bridge%2C%20creating%20something%20kinda%20LAN..

Filesystem root - chroot name space

- The OS has a file system started at root
  - /bin, and so on, contain programs and libraries
- If you want to run a program which uses a different version of /bin and so on
- This can be achieved using chroot
  - Mount new /bin on some place (say /mnt)
  - Issue chroot /mnt *command*
  - *command* will then see the root as /mnt, not the real root
- Using this technique, can give each process on system its own file system
- Changes the root directory for currently running processes as well as it children for
  - Search paths
  - Relative directories
- Using chroot can be escaped given proper capabilities, thus pivot_root is often used instead
  - chroot; points the processes file system root to new directory
  - pivot_root; detaches the new root and attaches it to process root directory
  - pivot_root info at https://man7.org/linux/man-pages/man8/pivot_root.8.html



Often used when building system images

- chroot to temp directory
- Download and install packages in chroot
- Compress chroot as a system root FS

Cgroups (or Control groups)

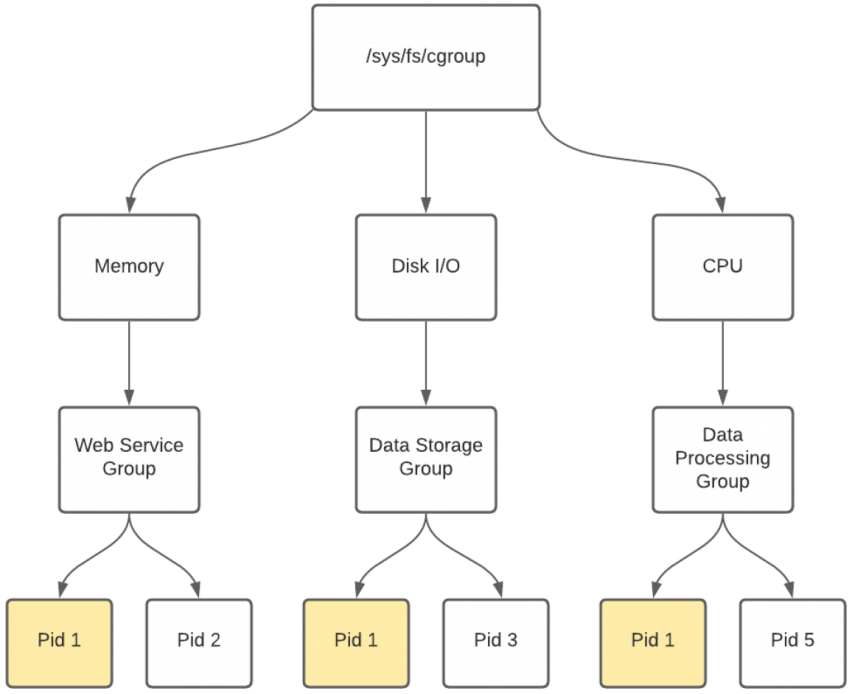
- Cgroup is a linux feature to limit, police, and account the resource usage for a set of processes. Docker uses cgroups to limit the system resources.
- Cgroups - provides mechanisms (fine grain control) to allocate, monitor and limit resources such as ***CPU time, system memory, block IO or disk bandwidth, network bandwidth, or combinations of these resources*** — among user-defined groups of tasks (processes) running on a system.
- Cgroups works on resource types. So it works by dividing resources into groups and then assigning tasks to those groups, deny access to certain resources, and even reconfigure our cgroups dynamically on a running system.
- When you install Docker binary on a linux box like ubuntu it will install cgroup related packages and create subsystem directories.
- Hardware resources can be appropriately divided up among tasks and users, increasing overall efficiency.

Linux cgroups - Functionality

- Access
  - which devices can be used per cgroup
- Resource limiting
  - memory, CPU, device accessibility, block I/O, etc.
- Prioritization
  - who gets more of the CPU, memory, etc.
- Accounting
  - resource usage per cgroup
- Control
  - freezing & check pointing
- Injection
  - packet tagging

Linux cgroups - usage

- cgroups are hierarchically structured where each of the groups are created for a resource with an number.
- Tasks are assigned to cgroups
- Each cgroups has a resource limitation
- There is a hierarchy for each resource

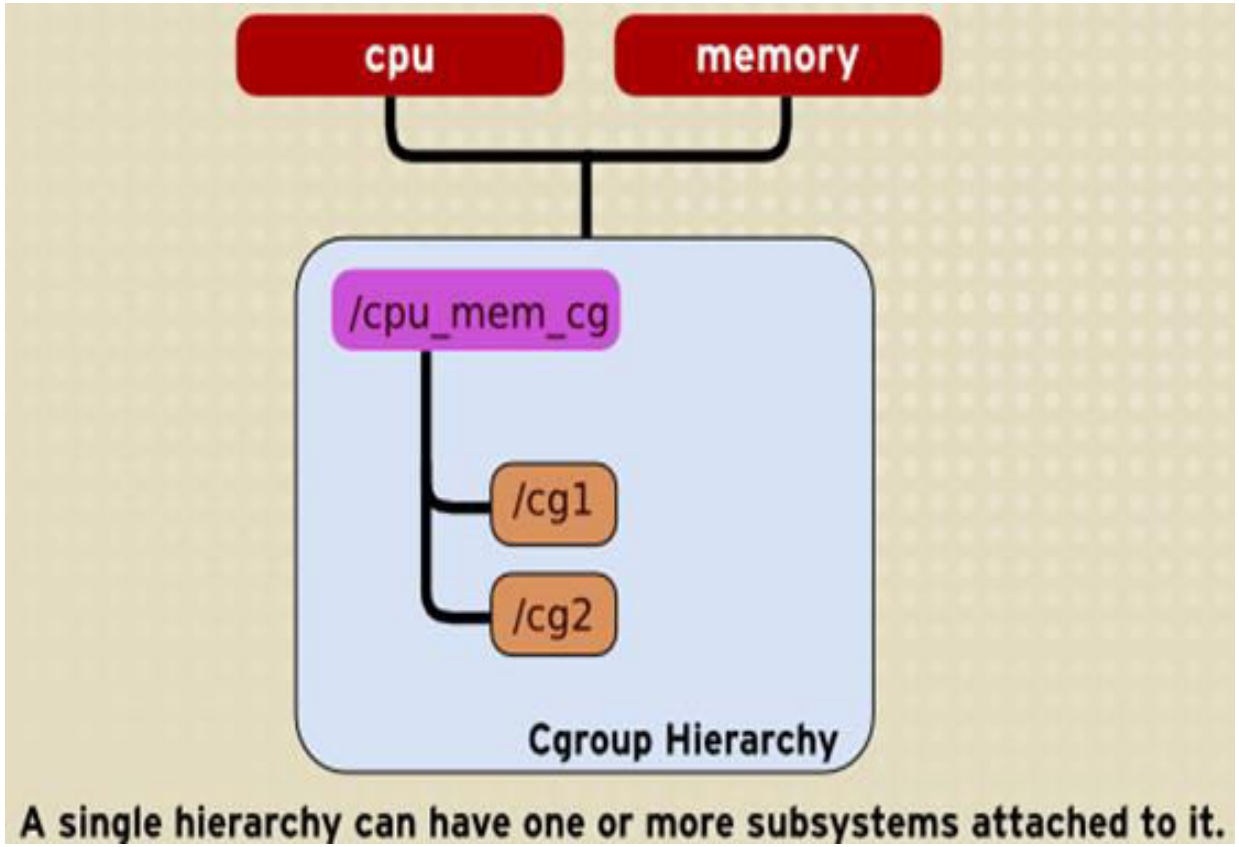


What resources can we limit?

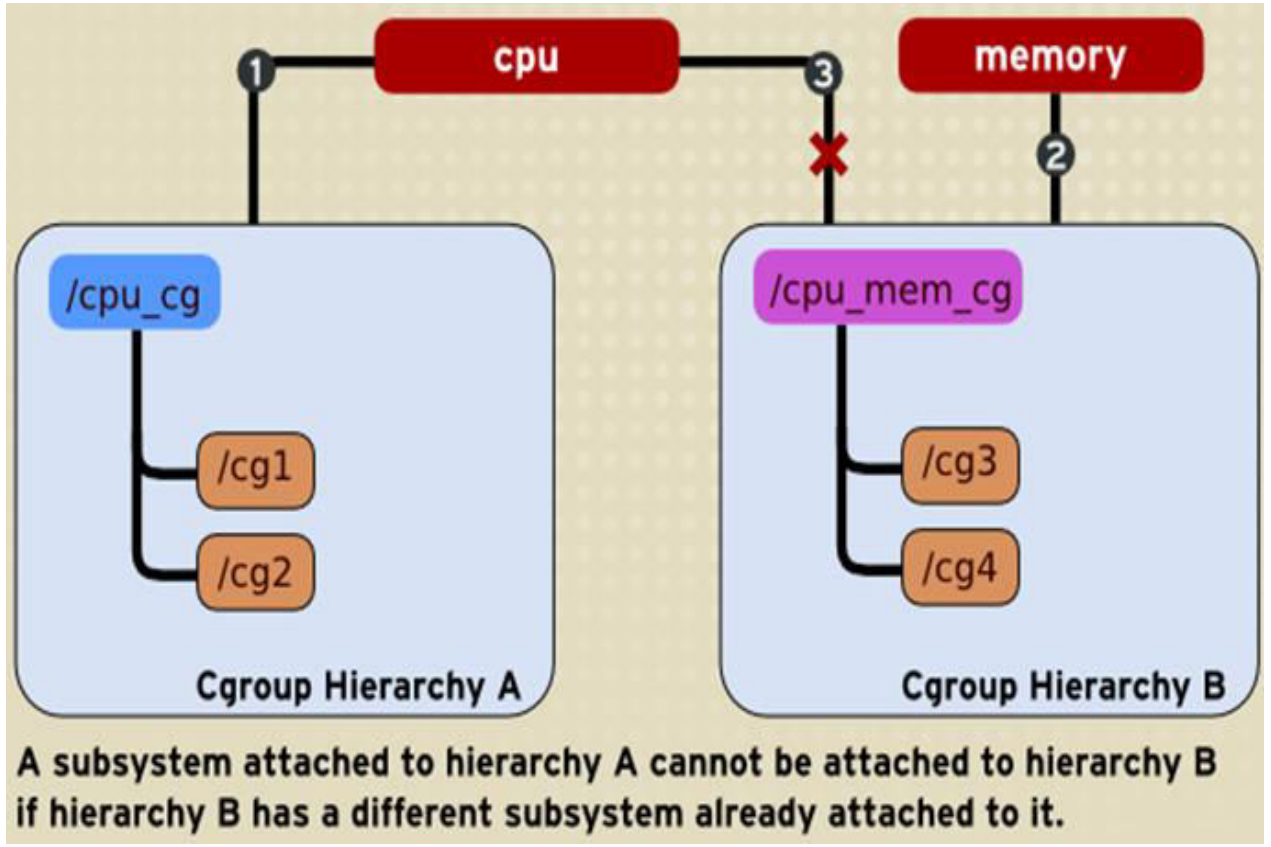
All of the following can be limited for a Cgroup

- Memory
- CPU
- Block IO
- Devices (which of the devices and allowing creation of devices ..)
- Network

cgroup Examples



A single hierarchy can have one or more subsystems attached to it.

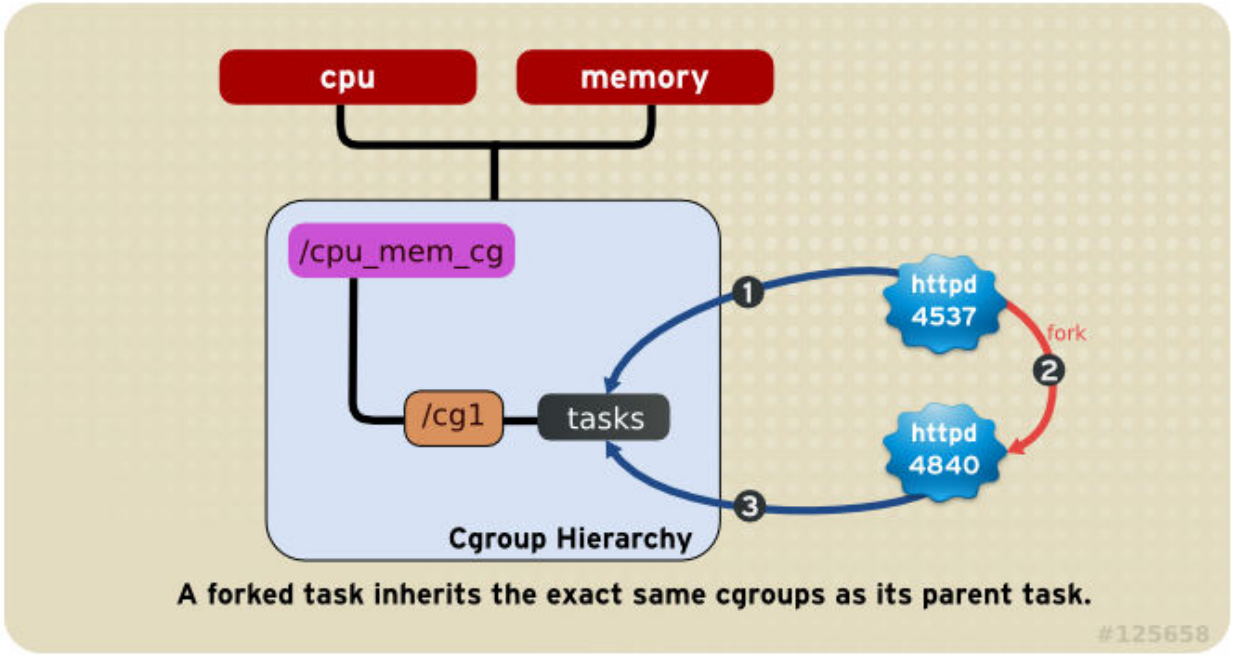


A subsystem attached to hierarchy A cannot be attached to hierarchy B if hierarchy B has a different subsystem already attached to it.

Each cgroup has a resource limitation associated with it

What resources can we limit?

- When a process creates a child process, the child process stays in the same cgroup
  - Good for servers such as NFS
  - Typical operation
    - Receive request
    - Fork child to process request
    - Child terminates when request complete
    - All children will have resource limitation of parent => the resource limitation of parent will apply to processing requests



Container Filesystem

- Programs running inside containers know nothing about image layers.
- From inside a container, the filesystem operates as though it's not running in a container or operating on an image.
- From the perspective of the container, it has exclusive copies of the files provided by the image. This is made possible with something called a *union filesystem (UFS).*
- Docker uses a variety of union filesystems and will select the best fit for your system.
- A union filesystem is part of a critical set of tools that combine to create effective filesystem isolation.
- The other tools are MNT namespaces and the chroot system call.

Union Filesystem

- Union is a type of a filesystem that can create an illusion of merging contents of several directories (created in layers) into one without modifying its original (physical) sources which can be shown in single, merged view
- Union file system operates by creating layers, making them very lightweight and fast.
- Docker Engine uses UnionFS to provide the building blocks for containers.
- Docker Engine can use multiple UnionFS variants, including AUFS, overlay2, btrfs, vfs, and DeviceMapper.

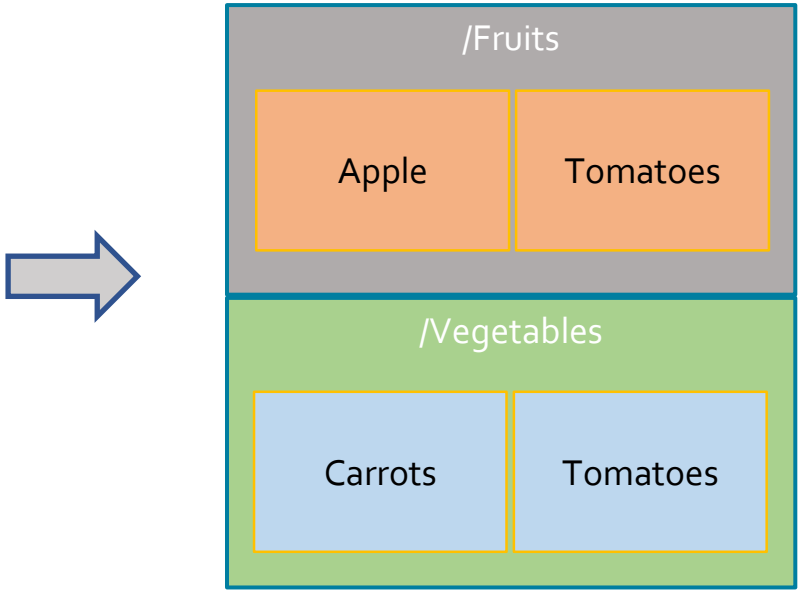
Reference:https://docs.docker.com/get-started/overview/

https://martinheinz.dev/blog/44

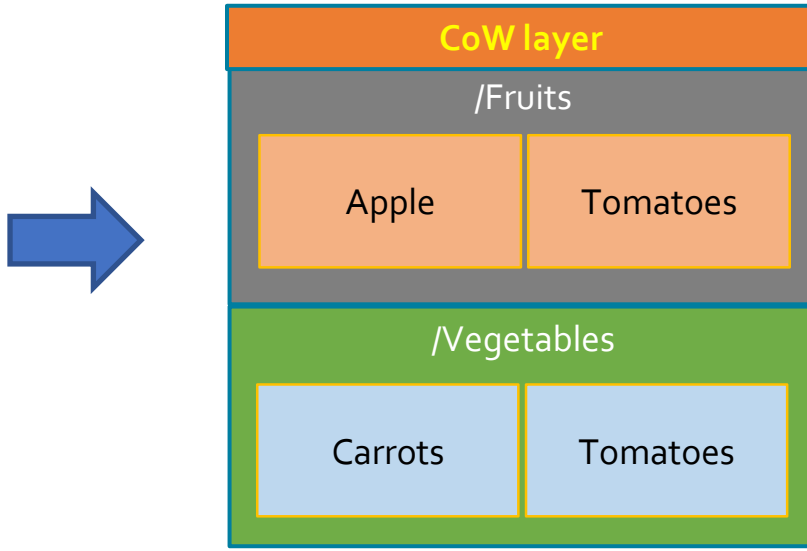
unionfs features - Layering

- unionfs permits layering of file systems
  - */Fruits* contains files *Apple, Tomato*
  - */Vegetables* contains *Carrots, Tomato*

  mount –t unionfs –o dirs=/Fruits:/Vegetables none /mnt/healthy
  - */mnt/healthy* has 3 files – *Apple, Tomato, Carrots*
  - *Tomato* comes from */Fruits* (1st in *dirs* option)
- As if /Fruits is layered on top of /Vegetables



- -o cow option on mount command enables copy on write
  - *If change is made to a file*
  - *Original file is not modified*
  - *New file is created in a hidden location*
- If /Fruits is mounted ro (read-only), then changes will be recorded in a temporary layer

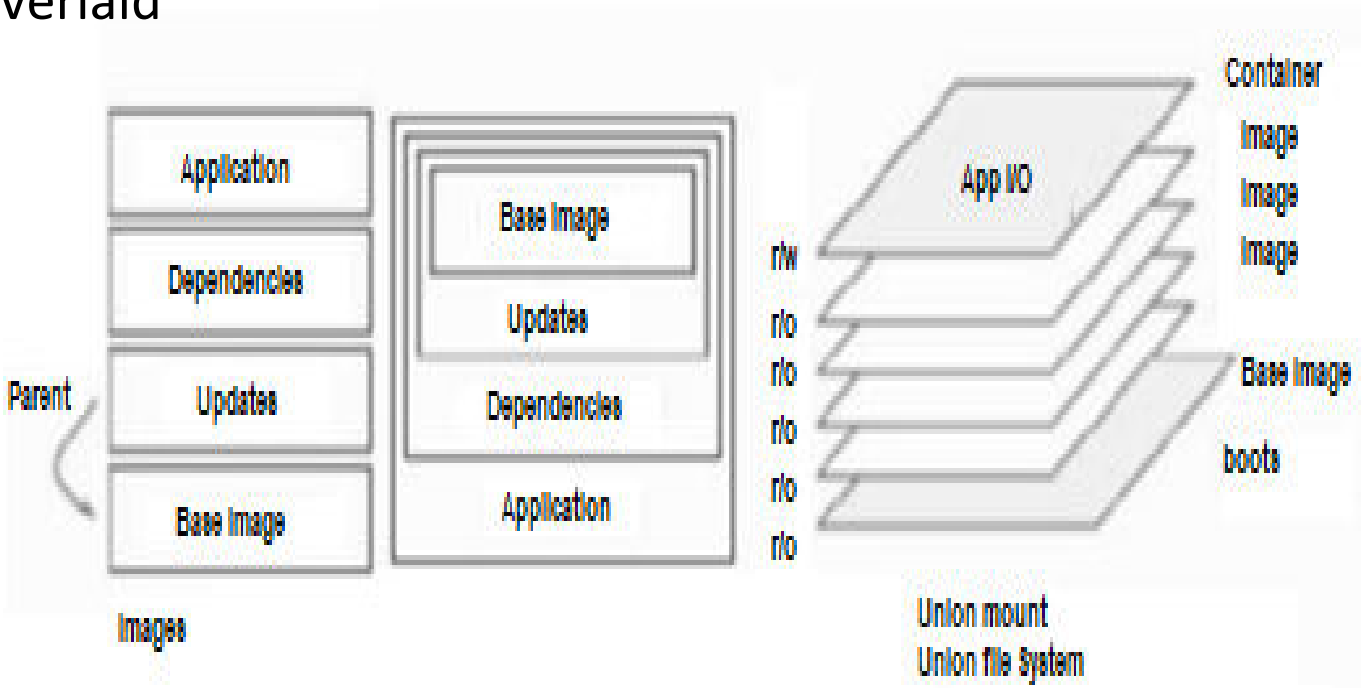


http://www.linuxjournal.com/article/7714

Docker and UnionFS

Incremental Images

- Union FS
  - Files from separate FS (branches) can be overlaid
  - Forming a single coherent FS
  - Branches may be read-only or read-write
- Docker Layers
  - Each layer is mounted on top of prior layers
  - First layer = base image (scratch, busybox, ubuntu, …)
  - A read-only layer = an image
  - The top read-write layer = container



https://www.slideshare.net/cm13/introduction-to-docker-48784005

Weaknesses of Union Filesystem

- Different filesystems have different rules about file attributes, sizes, names, and characters.
- Union filesystems are in a position where they often need to translate between the rules of different filesystems. In the best cases, they're able to provide acceptable translations. In the worst cases, features are omitted.
- Union filesystems use a pattern called *copy-on-write, and that makes implementing* memory-mapped files (the mmap system call) difficult.
- Most issues that arise with writing to the union filesystem can be addressed without changing the storage provider. These can be solved with volumes
- The union filesystem is not appropriate for working with long-lived data or sharing data between containers, or a container and the host.

Summary

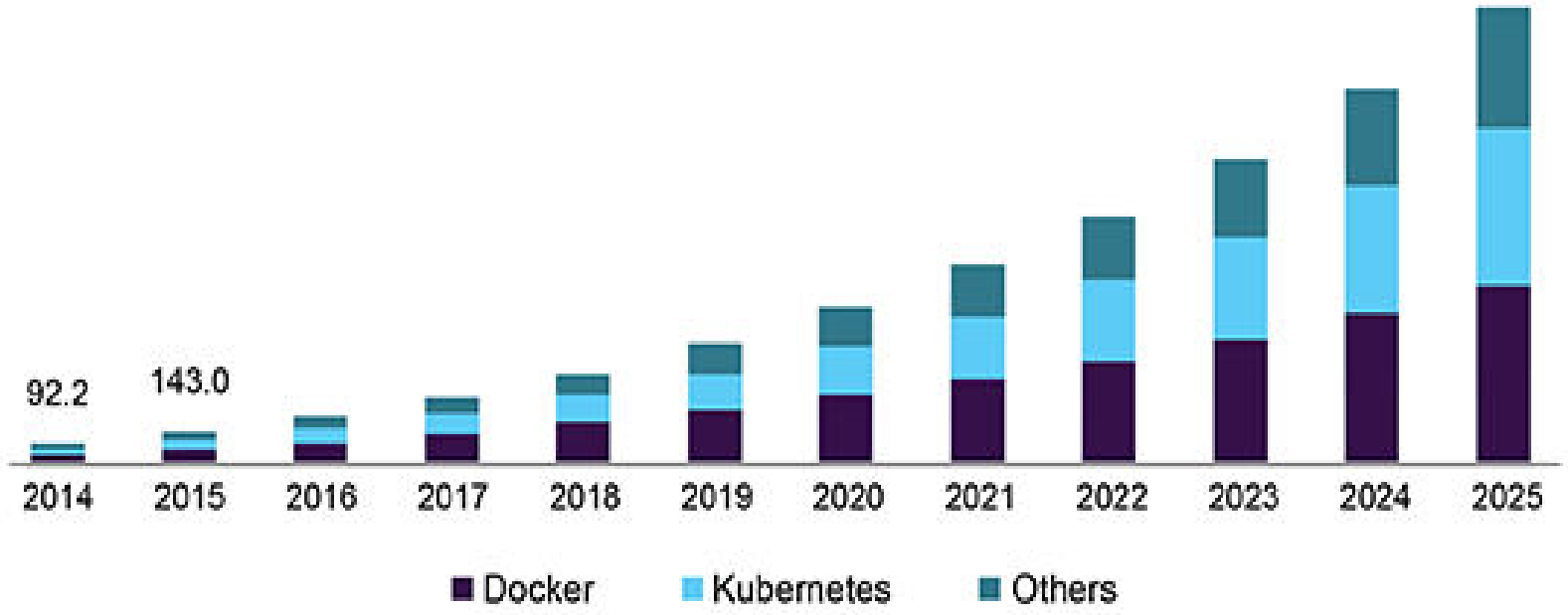
- Use namespaces for controlling resource access
  - Docker has developed their own namespace technology called **namespaces** to provide the isolated workspace called the container.
  - When you run a container, **Docker** creates a set of **namespaces** for that container.
  - These **namespaces** provide a layer of isolation.
- Use cgroups for resource sharing

CLOUD COMPUTING

Container Software

1. Docker
2. AWS Fargate
3. Google Kubernetes Engine
4. Amazon ECS
5. LXC
6. Microsoft Azure
7. Google Cloud Platform
8. Core OS

U.S. application container market size, by platform, 2014 - 2025 (USD Million)



Source: www.grandviewresearch.com

Additional References

Containers vs VMs
https://www.redhat.com/en/topics/containers/containers-vs-vms

Docker container
https://www.simplilearn.com/tutorials/docker-tutorial/what-is-docker-container

Understanding Linux containers
https://www.redhat.com/en/topics/containers
https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html

Additional Reading

- https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-Relationships_Between_Subsystems_Hierarchies_Control_Groups_and_Tasks.html
- https://en.wikipedia.org/wiki/Cgroups
- https://www.youtube.com/watch?v=sK5i-N34im8
  - Cgroups, namespaces and beyond: What are containers made from – Jerome Pettazzoni, Docker

  https://www.youtube.com/watch?v=nXV6qihj5uw

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

CLOUD COMPUTING

Lightweight virtualization – Containers, namespaces, cgroups
&
Deployment of cloud native applications through Docker, UnionFS

Dr. H.L. Phalachandra

Prof. Venkatesh Prasad

Department of Computer Science and Engineering

Acknowledgements:

Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

Containers – Definitions

- Linux Containers or LXC is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) which can run multiple workloads, on a control host running a single linux OS instance
- LXC provides a virtual environment that has its own process and network space and does not create a full-fledged virtual machine.
- LXC uses Linux kernel cgroups and namespace isolation functionality to achieve the same.

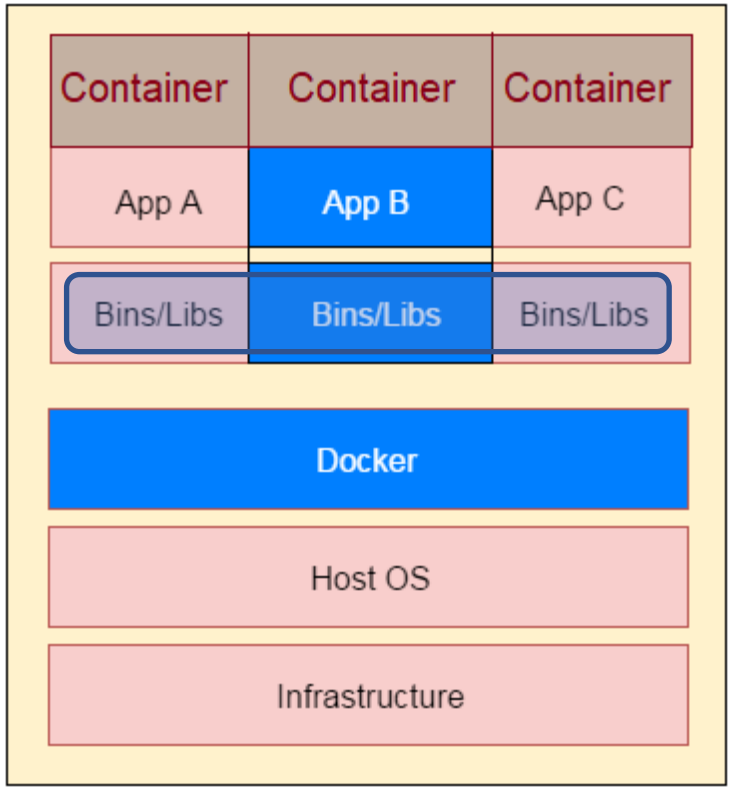
Motivation

- VMs support the objective of virtualizing the physical systems and allowing multiple tenants/applications to be isolated and share the physical resources along with Access control
- One of the challenges as observed is, this isolation achieved by or provided by VM is expensive
- Traditional OSs supporting multiple application processes, but share a disk, with all the processes capable of seeing the entire filesystem with access control built on top, and also share a network.
- Containers using a light weight mechanism, provide this virtualization extending the isolation provided by our traditional OS

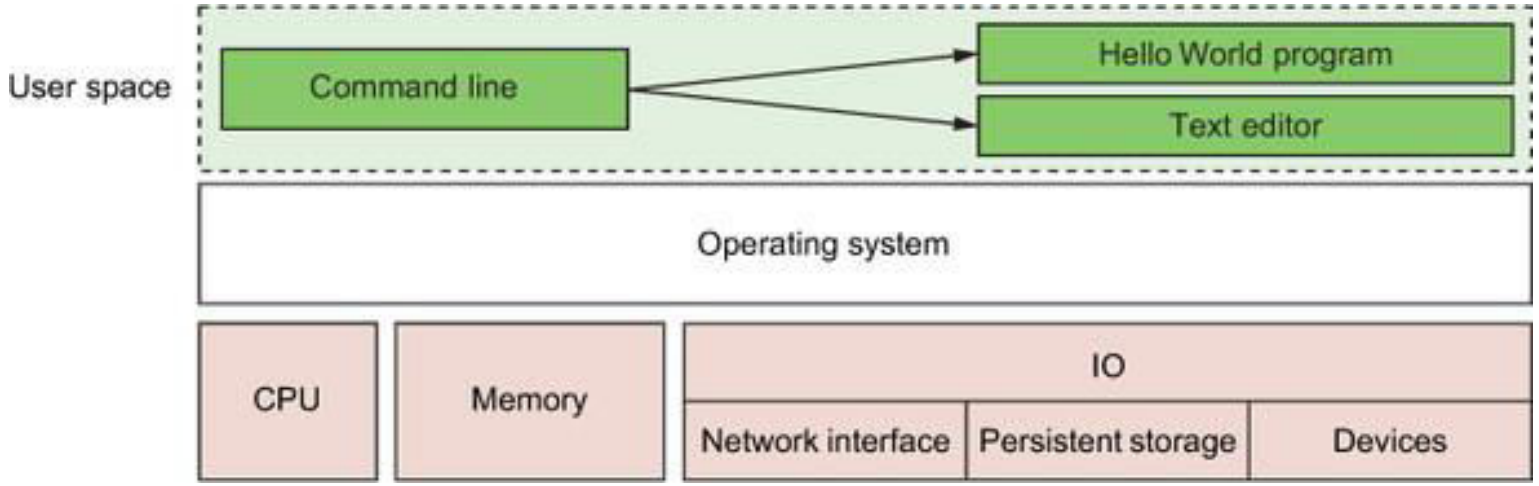
Lxc (tools) is an user space toolset for creating & managing Linux Containers – different from LXC Linux containers

Container Characteristics

- Containers sit on top of a physical server and its host OS.
- Each container shares the host OS kernel and, usually, the binaries and libraries too, but as read-only. This reduces the management overhead where a single OS needs to be maintained for bug fixes, patches, and so on.
- Containers are thus exceptionally “light”—they are only megabytes in size and take just seconds to start, versus gigabytes and minutes for a VM.
  - Container creation is similar to process creation and it has speed, agility and portability.
  - Thus containers have higher provisioning performance



Containers (Cont.)



A basic computer stack running two programs that were started from the command line

- Notice that the command-line interface, or CLI, runs in what is called user space memory, just like other programs that run on top of the operating system.
- Ideally, programs running in user space can't modify kernel space memory.
- Broadly speaking, the operating system is the interface between all user programs and the hardware that the computer is running on.

Contrasting Containers vs VMs

1. Each VM includes a separate OS image, which adds overhead in memory and storage footprint.

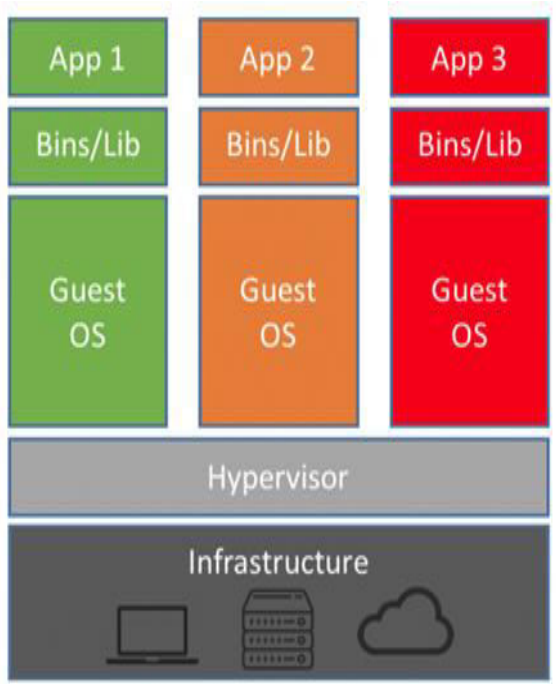
   Containers reduce management overhead as they share a common OS, only a single OS needs to be maintained for bug fixes, patches, and so on.

2. In terms of performance, VMs have to boot when provisioned making it slower, and also have I/O performance overhead

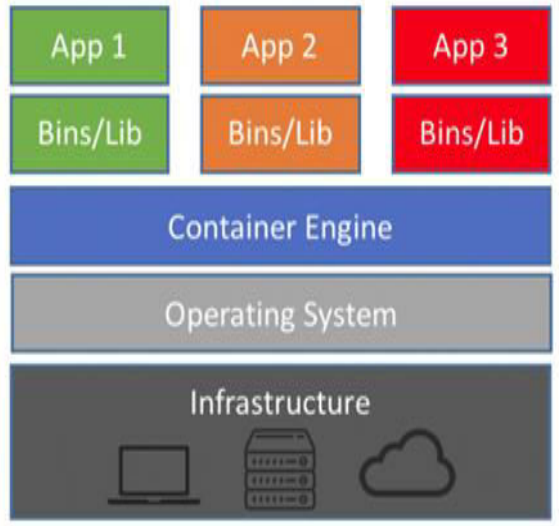
   Containers have higher performance as its creation is similar to process creation, so boots quickly and it has speed, agility and portability

3. VMs are more flexible as Hardware is virtualized to run multiple OS instances.

   Containers run on a single OS and also can support only Ubuntu containers of that type of OS where its running or containers cannot be of different OS variants



Machine Virtualization



Containers

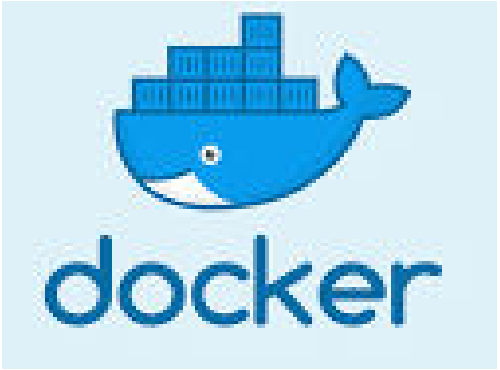
4. VMs consume more resources and come up slower than Containers which come up more quickly and consume fewer resources

CLOUD COMPUTING

VM vs Docker

Virtual Machine	Docker Container
Hardware-level process isolation	OS level process isolation
Each VM has a separate OS	Each container can share OS
Boots in minutes	Boots in seconds
VMs are of few GBs	Containers are lightweight (KBs/MBs)
Ready-made VMs are difficult to find	Pre-built docker containers are easily available
VMs can move to new host easily	Containers are destroyed and re-created rather than moving
Creating VM takes a relatively longer time	Containers can be created in seconds
More resource usage	Less resource usage

Docker



- Docker is an open platform tool which makes it easier to create, test, ship, deploy and to execute applications using containers.
- Docker containers allow us to separate the applications from the infrastructure enabling faster deployment of applications/software
- It significantly reduces the between writing code and running it in production by providing methodologies for shipping, testing and deploying code quickly
- It can be considered as a tool that helps to package and run an application in a loosely isolated environment called a container.
- It could be looked at as a PaaS product that uses OS level virtualization to deliver S/W packages
- Docker provides the isolation and security to allow many containers to run on a single server or virtual machine
- Its typical to find between 8 -18 containers running simultaneously on a single server/VM

Docker (Cont.)

- Containers created using Docker can contain everything needed to run an application, so you do not need to rely on what is currently installed on the host system
- Docker allows these containers to be shared in a way that it would work identically.
- Docker can be used with network applications such as web servers, databases, mail servers, with terminal applications like text editors, compilers, network analysis tools, and scripts.
  - In some cases, it's even used to run GUI applications such as web browsers and productivity software.
- Docker runs on Linux software on most systems.
  - Docker is also available as a native application for both macOS and Windows
  - Docker can run native Windows applications on modern Windows server machines.

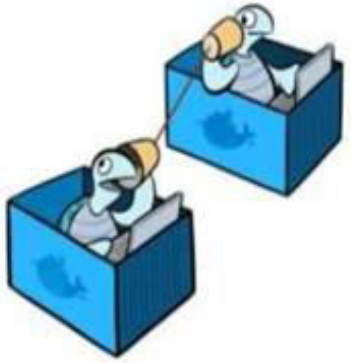
Docker Benefits

Portability, Shipping Applications

One App =

- binaries (exec, libs, etc.)
- data (assets, SQL DB, etc.)
- configs (/etc/config/files)
- logs

either in a container
or a *composition*



docker

Docker, Containers, and the Future of Application Delivery

Portability

Docker Promise: Build, Ship, Run !

- reliable deployments
- develop here, run there

docker

Docker, Containers, and the Future of Application Delivery

Build

Develop an app using Docker containers with any language and any toolchain.

Ship

Ship the "Dockerized" app and dependencies anywhere - to QA, teammates, or the cloud - without breaking anything.

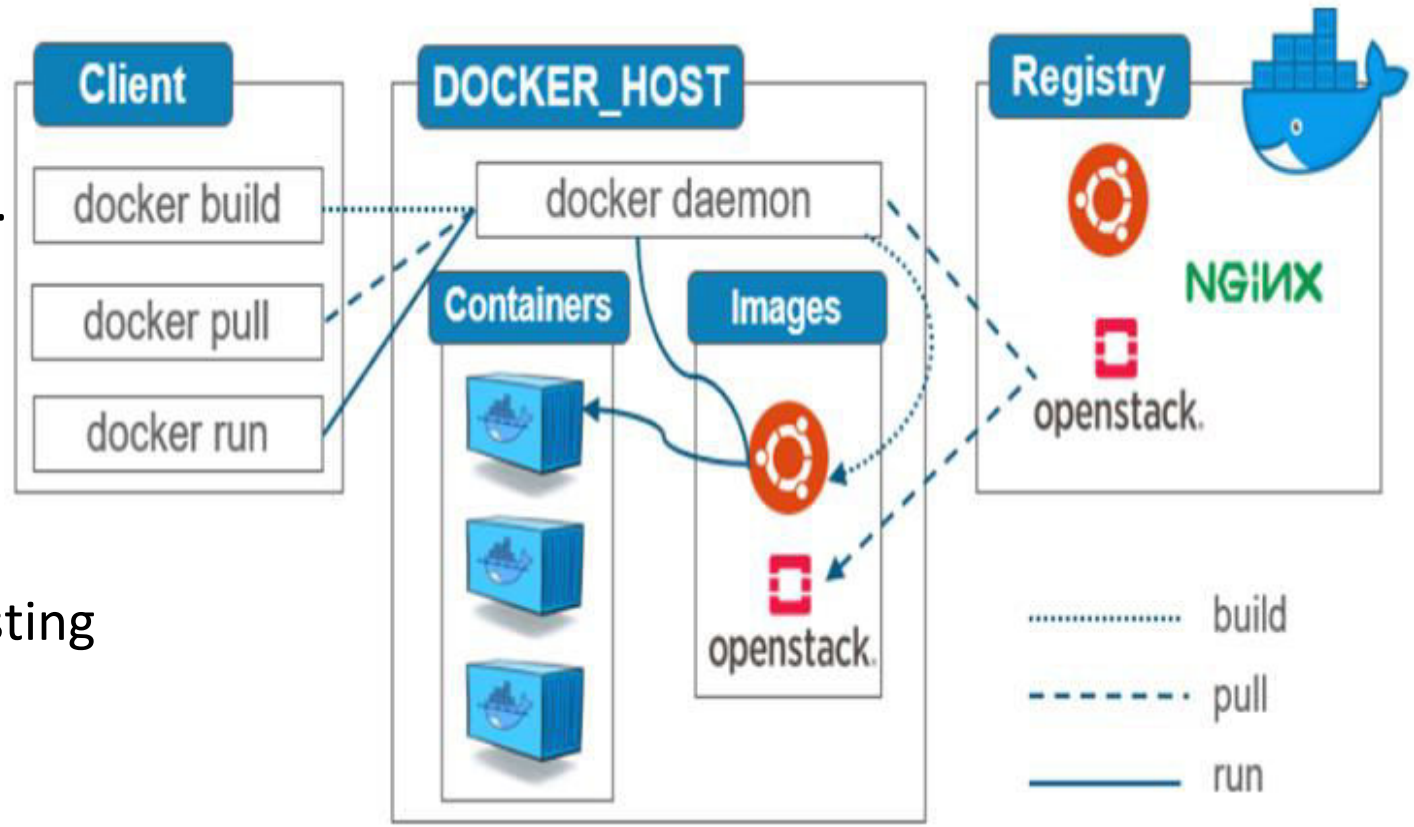
Run

Scale to 1000s of nodes, move between data centers and clouds, update with zero downtime and more.

Developers use Version Control Systems (VCS) like Git. DevOps also uses VCS for docs, scripts and Dockerfiles.
DevOps could use Dockerfile to describes how to build the image, and something like docker-compose.yml to describe how to orchestrate them.

Docker Architecture

- Docker uses a client-server architecture.
- The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.
- Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

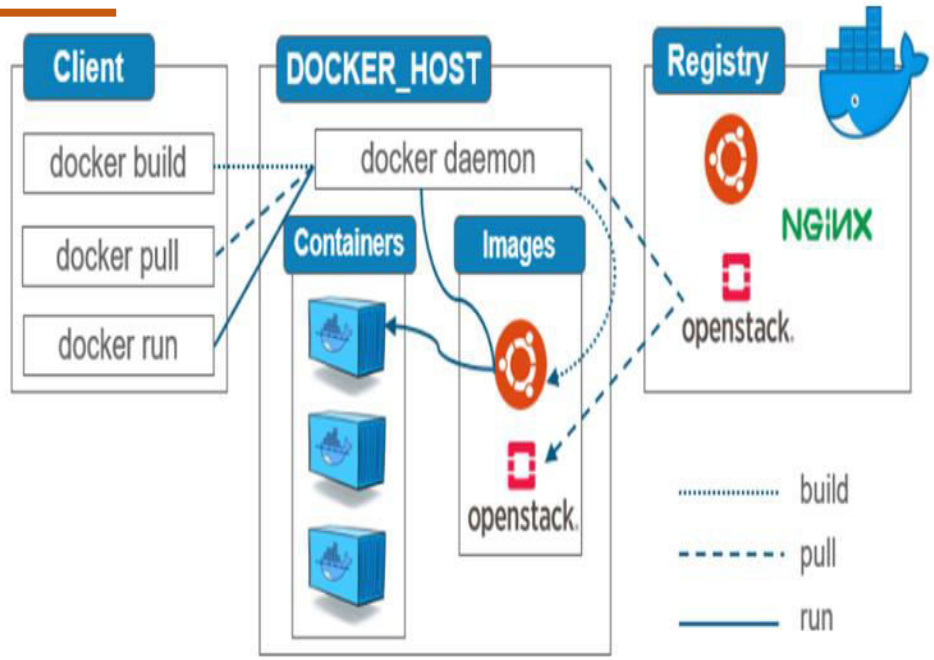


Reference:https://docs.docker.com/get-started/overview/

Docker Architecture

The Docker daemon

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
- A daemon can also communicate with other daemons to manage Docker services.



The Docker client

- The Docker client (docker) is the primary way that many Docker users interact with Docker.
- When you use commands such as **docker run**, the client sends these commands to 'dockerd' (docker daemon), which carries them out.
- The docker command uses the Docker API.
- The Docker client can communicate with more than one daemon.

Docker Architecture

Docker Host

- Docker Host has the Docker Daemon running and can host (like a private hub/registry) or connect (like to a public docker_hub/registry) to a Docker Registry which stores the Docker Images.
- The Docker Daemon running within Docker Host is responsible for the Docker Objects images and containers.

Docker Objects :

There are objects like

- Images
- Containers
- Networks
- Volumes
- Plugins and other objects which are created and used while using docker.

Docker Objects : Images

Images : This is an is a read-only template with instructions for creating a Docker container. This could be based on another image (available in the registry) with additional customizations. Eg. Image for a Webserver .. Original image of Ubuntu customized with installation and configuration of the webserver which is created only as a read-only image which can be deployed and an application can be run in the same.

This is done using a ***Dockerfile*** a script file which defines the syntax to indicate steps needed to create the image (and run using a run command).

Each instruction in a Dockerfile creates a ***layer*** in the image.

These Dockerfiles are distributed along with software that the author wants to be put into an image. In this case, you're not technically installing an image. Instead, you're following instructions to build an image.

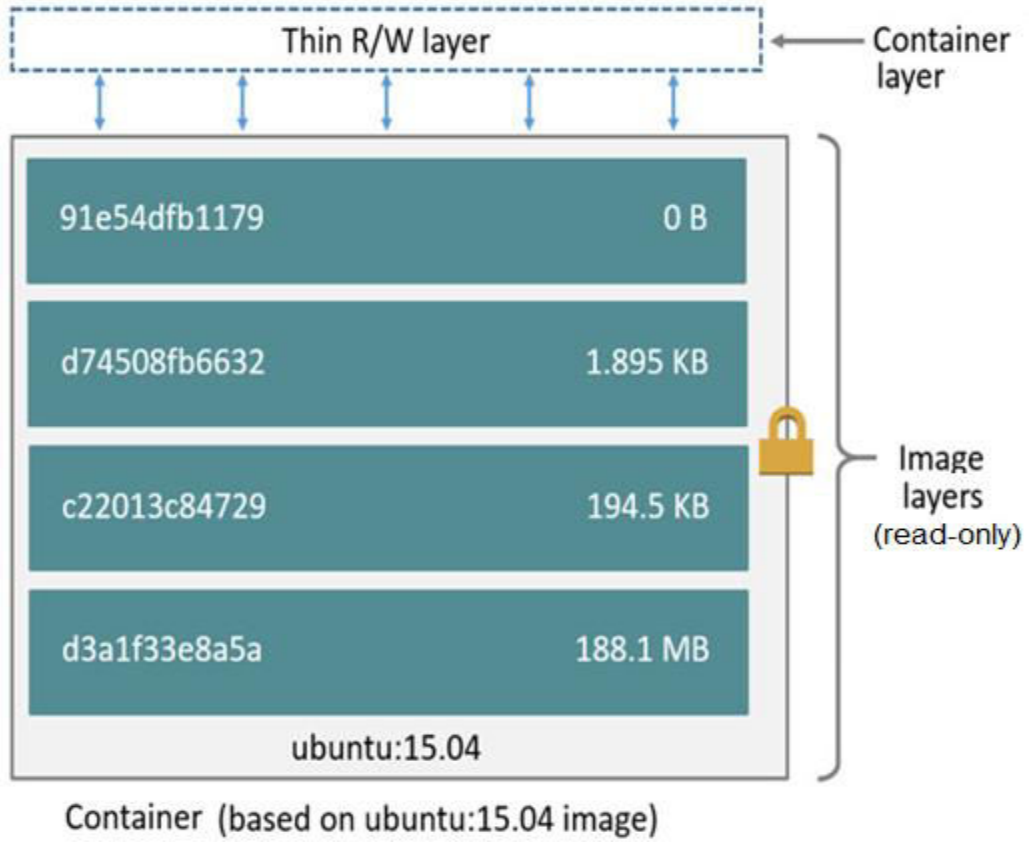
Docker Objects : Images – More on Dockerfile

- Distributing a Dockerfile is similar to distributing image files using your own distribution mechanisms. A common pattern is to distribute a Dockerfile with software from common version-control systems like Git.
  - If you have Git installed, you can try this by running an example from a public repository:
    - git clone https://github.com/dockerinaction/ch3_dockerfile.git
    - docker build -t dia_ch3/dockerfile:latest ch3_dockerfile

  In this example, you copy the project from a public source repository onto your computer and then build and install a Docker image by using the Dockerfile included with that project. The value provided to the -t option of docker build is the repository where you want to install the image.
- Building images from Dockerfile's is a light way to move projects around that fits into existing workflows. Could lead to an unfavourable experience in case of a drift in dependencies between the time when the Dockerfile was authored and when an image is built on a user's computer.
- Docker Images can be removed or cleaned up with
  - docker rmi dia_ch3/dockerfile
  - rm -rf ch3_dockerfile

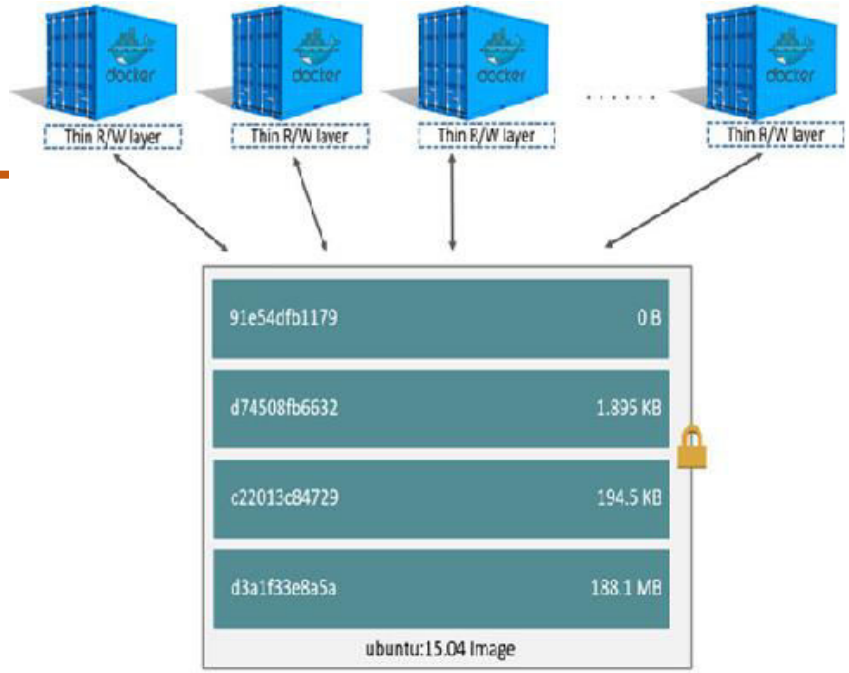
Docker Commands – Video Ref: https://www.youtube.com/watch?v=nXV6qihj5uw

Docker Objects : Images - layers



- Specification for a Docker Image is stored in Dockerfile
  - Should be only one for a container
  - Only the definition of the image
- Image is built from Dockerfile
- Specifies the read-only file systems in which various programs are installed E.g. web server + libraries
- Each instruction in a Dockerfile creates a ***layer*** in the image.
- A layer is set of files and file metadata that is packaged and distributed as an atomic unit.
  - Internally, Docker treats each layer like an image, and layers are often called intermediate images.
  - You can even promote a layer to an image by tagging it.
  - Most layers build upon a parent layer by applying filesystem changes to the parent.
  - Whenever there is a change in the Dockerfile and the image is rebuilt, only the ***incremental changes are rebuilt*** (making it light weight, small and fast when compared to other virtualized technologies)

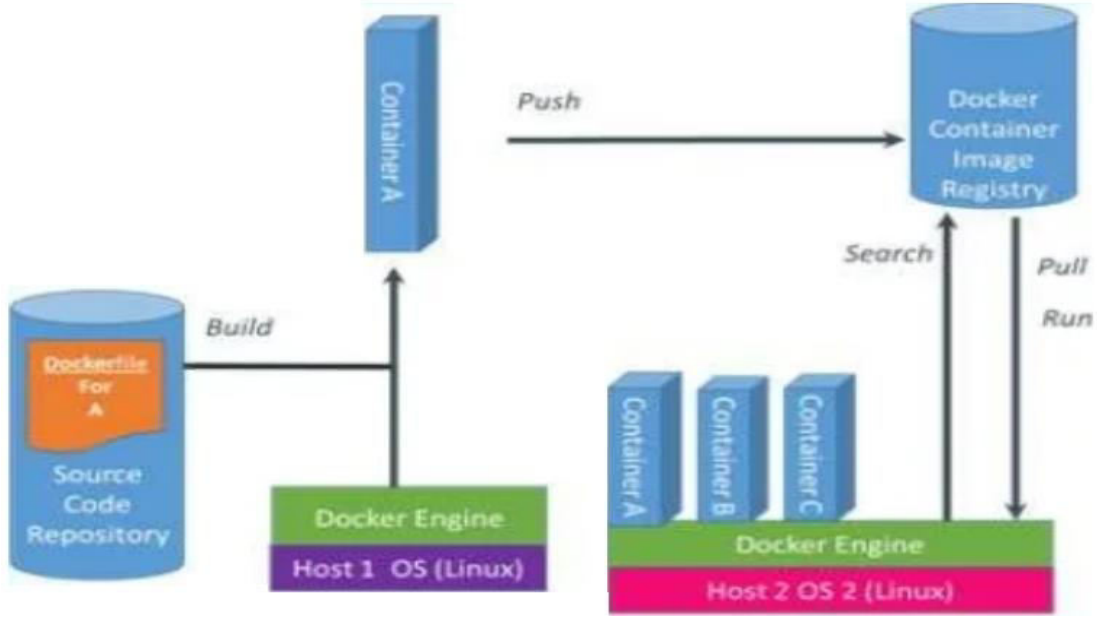
Docker Objects : Images & Docker Registries



- Image + temporary R/W file system
  - Used as temporary storage
  - Deleted when container is destroyed
- Multiple containers can use the same image & their own temporary storage

Docker registries:

- A Docker registry stores Docker images.
- ***Docker Hub*** is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default.
- You can even run your own ***private registry***.
- When you use the **docker pull or docker run** commands, the required images are pulled from your configured registry.
- When you use the docker push command, your image is pushed to your configured registry.



https://docs.docker.com/storage/storagedriver/

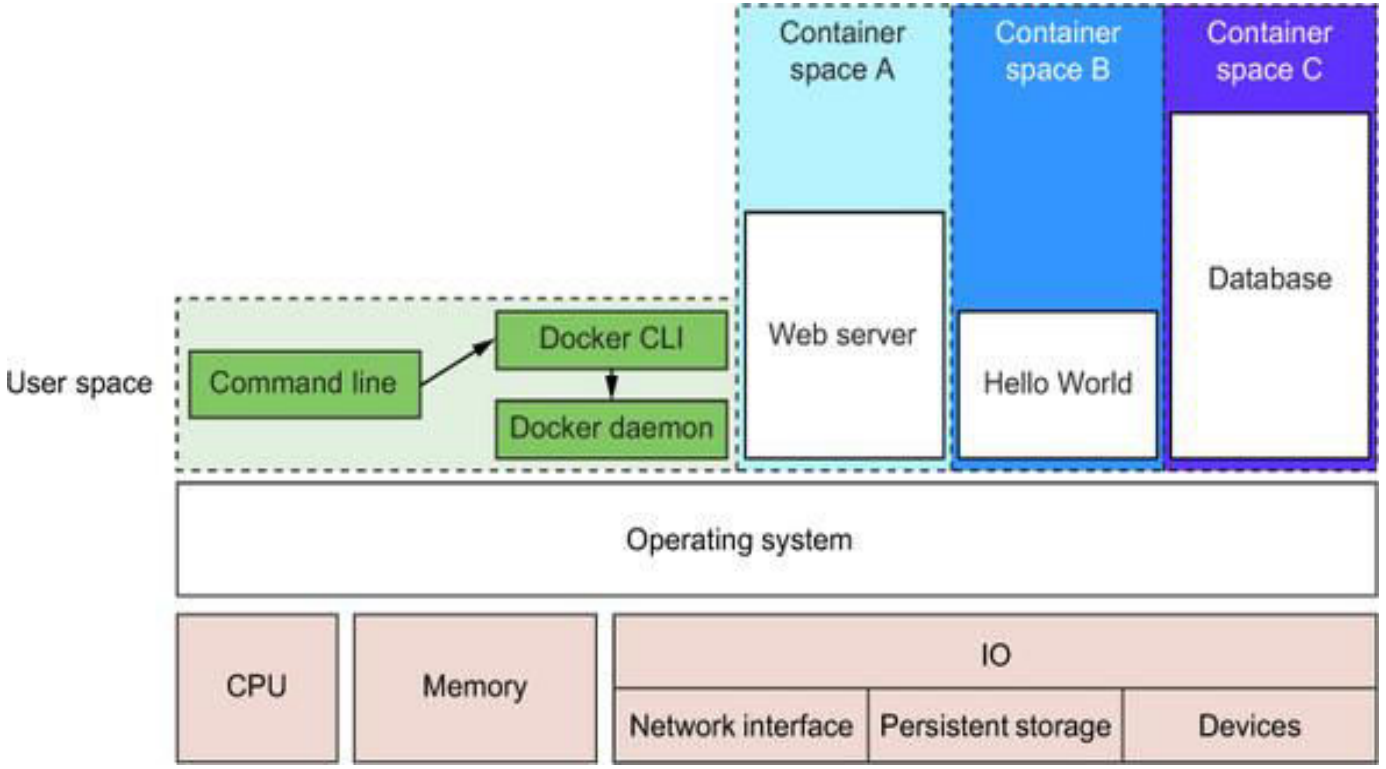
Docker Objects : Containers

- Docker Containers are the ready applications created from Docker Images. **Or** you can say they are running instances of the Images and they hold the entire package needed to run the application **Or** it could also be looked at as a runnable instance of an image **Or** an execution environment (sandbox).

- We can create, start, stop, move, or delete a container using the Docker API or CLI.
- A container can be connected to one or more networks. Storage could be attached to it, or even new image could be created based on its current state.
- A container is relatively well isolated from other containers and its host machine and this isolation can also be controlled. Processes in the container cannot access non-shared objects of other containers, and can only access a subset of the objects (like files) on the physical machine.
- Docker creates a set of name spaces when a container is created. This name spaces restrict what the objects processes in a container can see e.g. a subset of files
- A container is defined by its image and the configuration options provided to it when its created or started. When a container is removed, any changes to its state that are not stored in persistent storage will disappear

Docker running three containers on a basic Linux computer system

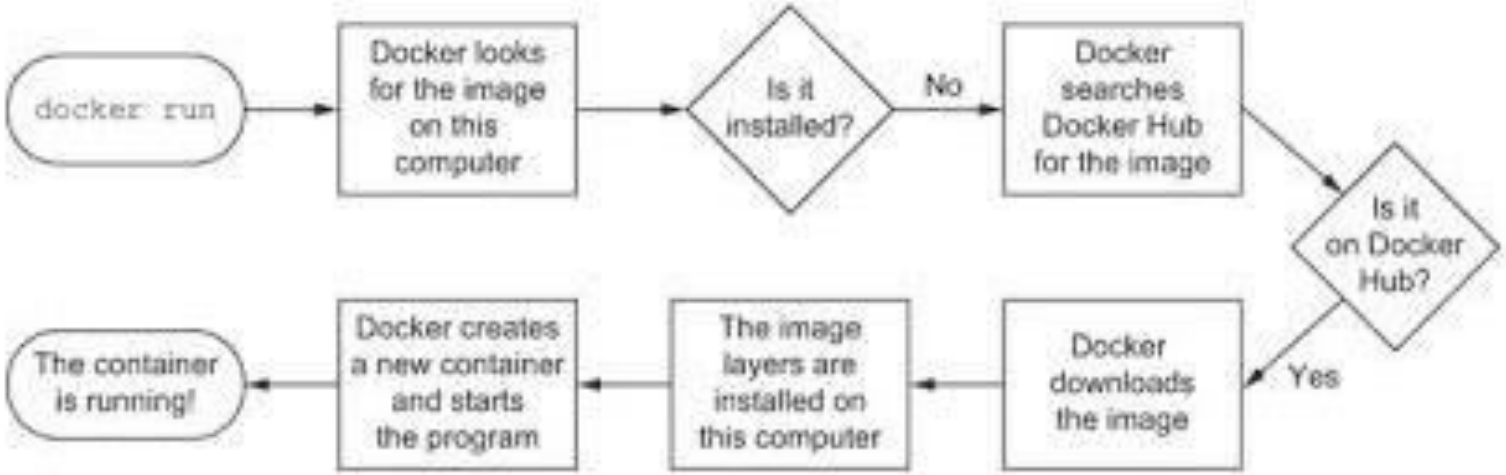
- Running Docker means running two programs in user space.
  - The first is the Docker engine that should always be running.
  - The second is the Docker CLI. This is the Docker program that users interact with to start, stop, or install software
- Each container is running as a child process of the Docker engine, wrapped with a container, and the delegate process is running in its own memory subspace of the user space.
  - Programs running inside a container can access only their own memory and resources as scoped by the container.



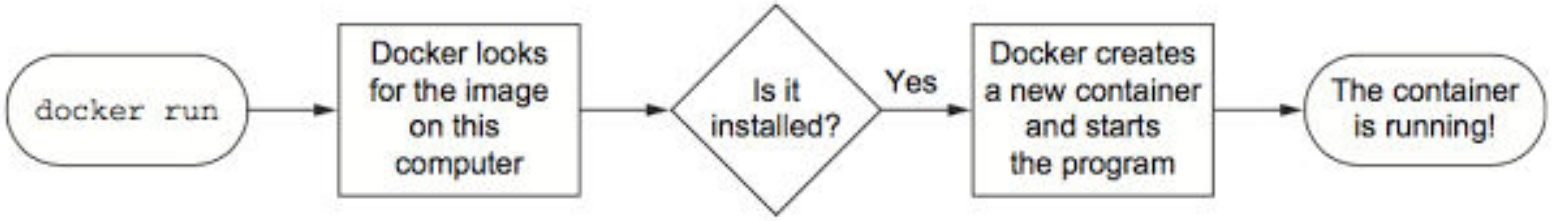
Docker can be looked at as a set of PaaS products that use OS-level virtualization to deliver software in packages called containers.

Running a program with Docker

- What happens after running **docker run**
- The image itself is a collection of files and metadata which includes the specific program to execute and other relevant configuration details

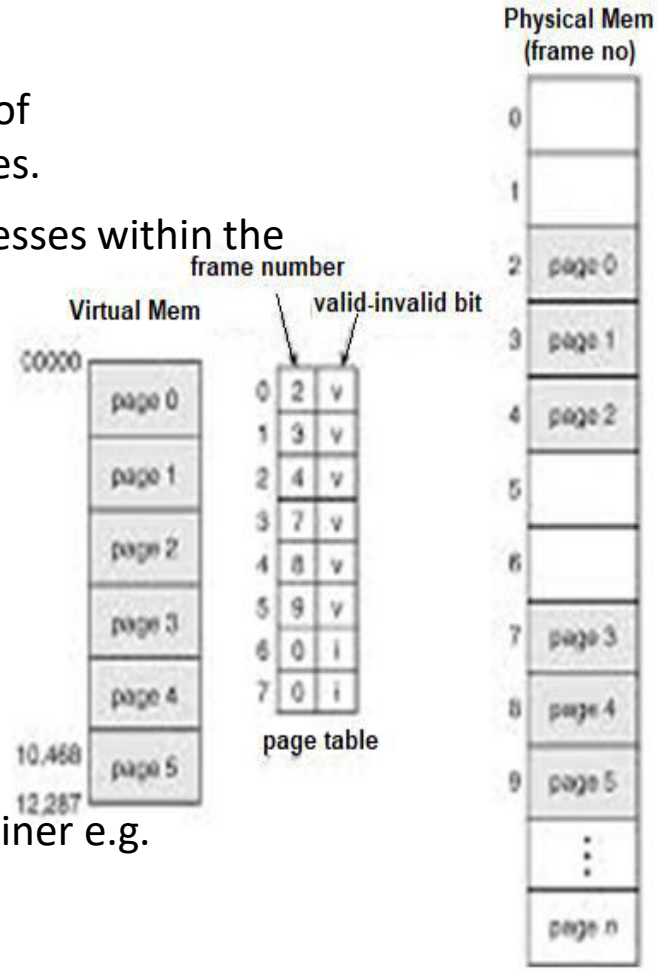


- Running docker run a second time.
- The image is already installed, so Docker can start the new container right away.



Namespaces – What's in a Name in CS?

- If you can't name an object, you can't access it. E.g. Web site – if name is hidden, can't access
- Paging : Processes can access only pages in its name spaces but cannot access physical page 1 (i.e. frame number 1 in the diagram)
- Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources and another set of processes sees a different set of resources.
- A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource
- The feature works by having the same namespace (with a name) for a group of resources and processes, but those namespaces refer to *distinct resources*
- A physical computer can have more than one namespaces (two or more)
- All the resources that a process sees can be considered a ***namespace***
  - The files seen by a process is the ***file namespace***
  - The network connections are part of ***network*** **namespace**
- Container Access is restricted to only subset of objects (e.g., files) on the physical machine using these namespaces. Restriction is applied on what can be seen by the object processes in a container e.g.
  - To restrict process (and container) to a subset of files use file namespace



Docker used Namespaces

- Docker builds containers at runtime using 10 major system features. Docker commands can be used to illustrate and modify features to suit the needs of the contained software and to fit the environment where the container will run.
- The specific features are as follows (a few of then are discussed in a little more detail):
  - PID namespace—Process isolation through an identifiers (PID number) – not aware of what happens outside its own processes
  - UTS namespace—allows for having multiple hostnames on a single physical host - Host and domain name (as other things like IP can change)
  - MNT namespace—isolate a set of mount points such that processes in different namespaces cannot view each others files. (almost like chroot) (Filesystem access & structure)
  - IPC namespace—provides isolation to container process communication over shared memory and having an ability to invite other processes to read from the shared memory

https://www.redhat.com/sysadmin/7-linux-namespaces

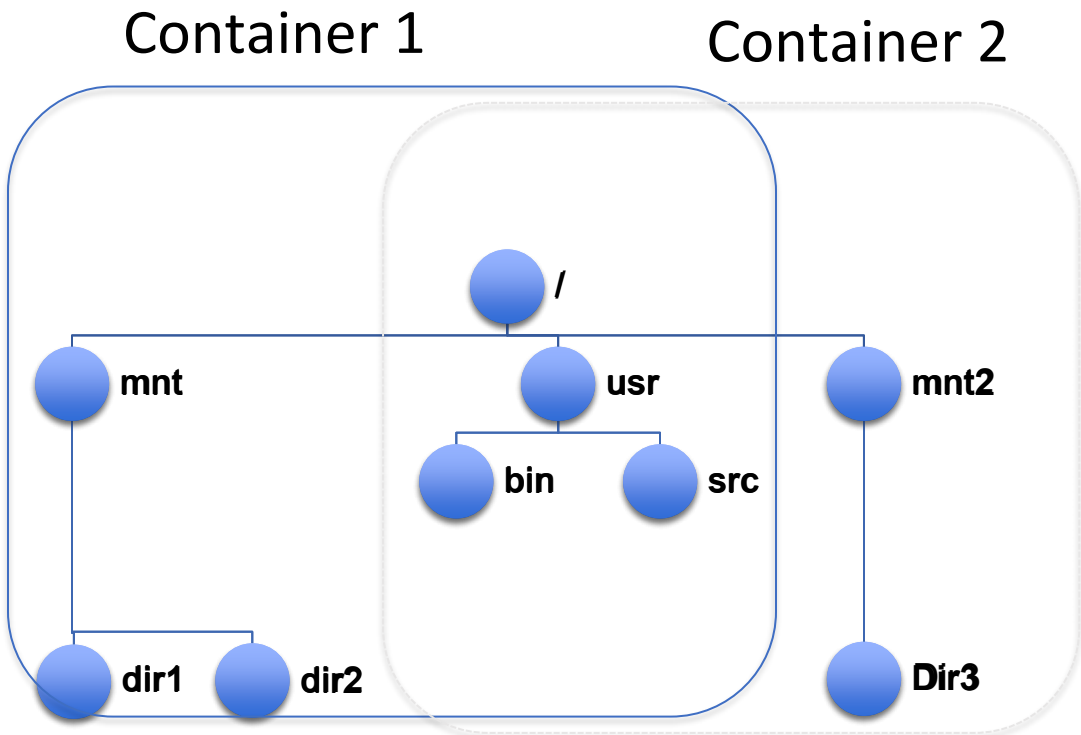
Docker used Namespaces (Cont.)

- NET namespace—allow processes inside each namespace instance to have access to a new IP address along with the full range of ports - Network access and structure
- USR namespace—provides user name-UID mapping isolating changes in names from the metadata within a container
- chroot syscall—Controls the location of the filesystem root
- cgroups—Controlling and accounting resources –rather than a hierarchical cgroup creation, there is a new cgroup with a root directory created to isolate resources and not allow navigation
- CAP drop—Operating system feature restrictions
- Security modules—Mandatory access controls

https://www.redhat.com/sysadmin/7-linux-namespaces

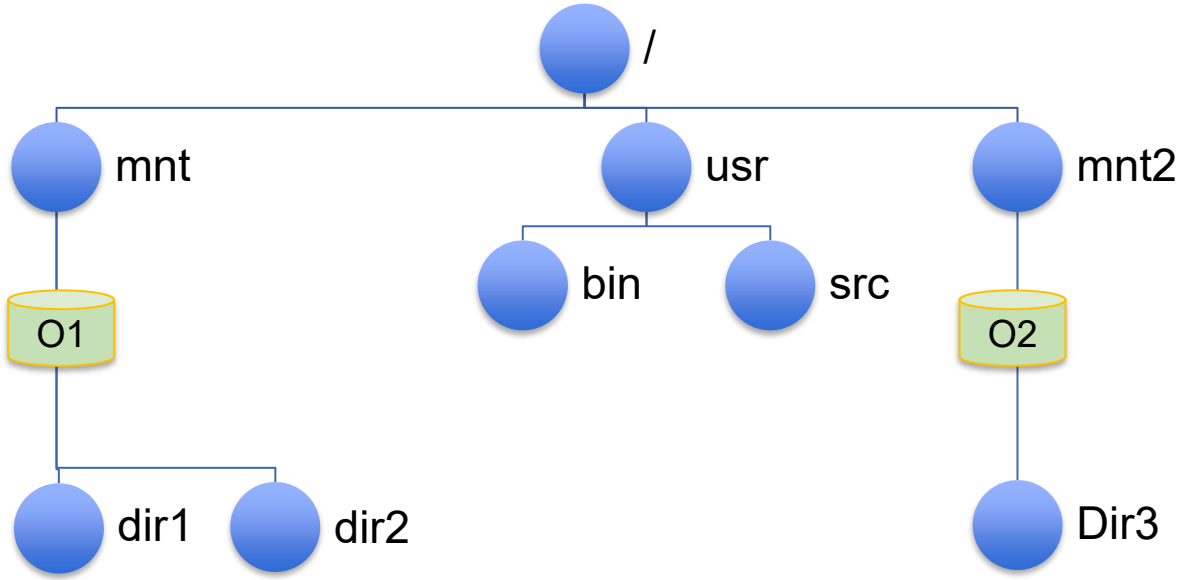
Access Control of objects : Illustration of name space (in this case MNT)

- Processes
  - in container 1 can access
    - Shared files in /usr
    - Non-shared /mnt
    - Cannot access /mnt2
  - in container 2 can access
    - Shared files in /usr
    - Non-shared /mnt2
    - Cannot access /mnt
- These are two different namespaces



Linux Filesystems before Namespaces

- */* is the *root file system*
  - Contains *vmunix* – the OS
  - *usr, bin, src* are all subdirectories
- *O1, O2* are volumes containing different versions of an application (e.g., Oracle)
- Mounted at */mnt* and */mnt2*
- This namespace is visible to all processes
- Access to files and directories controlled by access rights



Linux Filesystems after Namespaces

- File namespace: *mount namespace*
- *O1, O2* are volumes containing different versions of an application (e.g., Oracle)
  - Mounted at */mnt* and */mnt2*
- This namespace is visible to all processes
- Access to files and directories controlled by access rights
- Which namespace is process in?
  - Look at */proc/pid/ns*
  - *ls –l* for */proc/pidx/mnt* may show as below (4026531840 is the namespace id)
    - lrwxrwxrwx. 1 mtk mtk 0 Jan 8 04:12 mnt -> mnt:[4026531840]

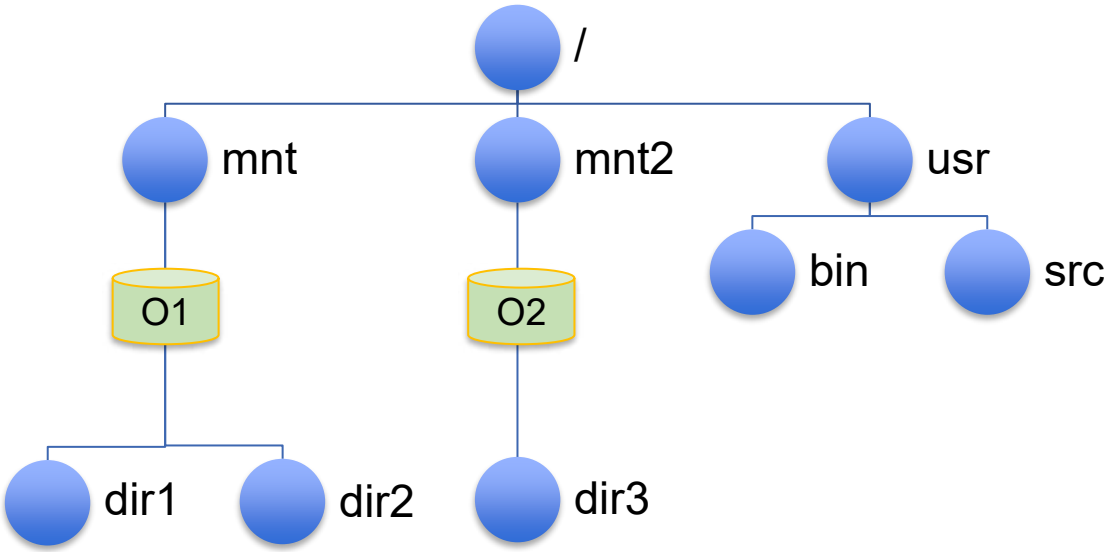


Illustration of Namespace Operations : E.g. MNT Namespace

Creation of Namespace

- To create a namespace, must create a process with that namespace
- *pid = clone(childFunc, stackTop, CLONE_NEWNS | SIGCHLD, argv[1]);*
- Creates a new child, like *fork()*
- *NEWNS* flag indicates that child has a new mount namespace
- Child can do *mount* and *umount* to modify namespace

Programs to be joining into a namespace

Allows program to join an existing namespace

int setns

- (int fd, // namespace to join
- int nstype); // type of ns

int unshare (int flags); // which namespace

- CLONE_NEWNS specifies mount namespace
- Similar to clone(); allows caller to create a new namespace

UTS (Unix Time Sharing) Namespace

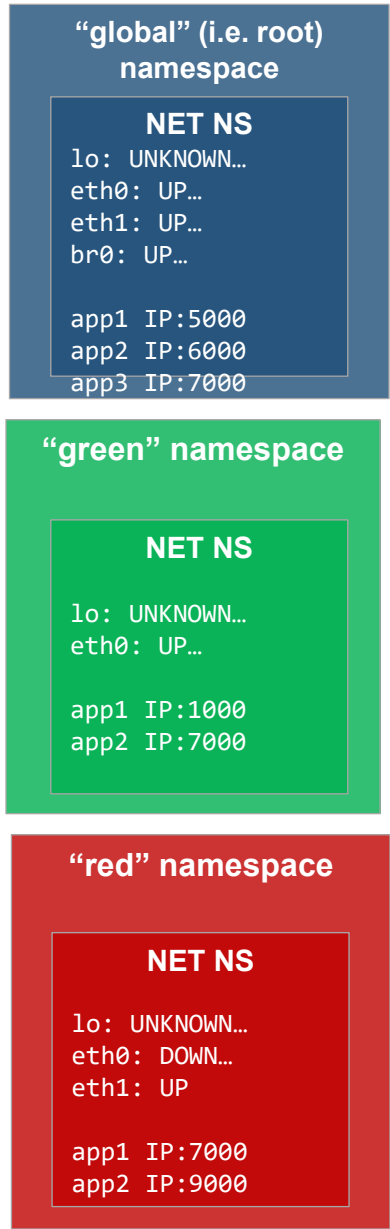
- Per namespace
  - Hostname
  - NIS domain name (Network Information Service)
- Reported by commands such as hostname
- Processes in namespace can change UTS values – only reflected in the child namespace
- Allows containers to have their own FQDN (Fully qualified Domain Name)

UTS namespace is about isolating hostnames.
The **UTS namespace** is used to isolate two specific elements of the system that relate to the uname system call.
The **UTS**(UNIX Time Sharing) **namespace** is named after the data structure used to store information returned by the uname system call.

CLOUD COMPUTING

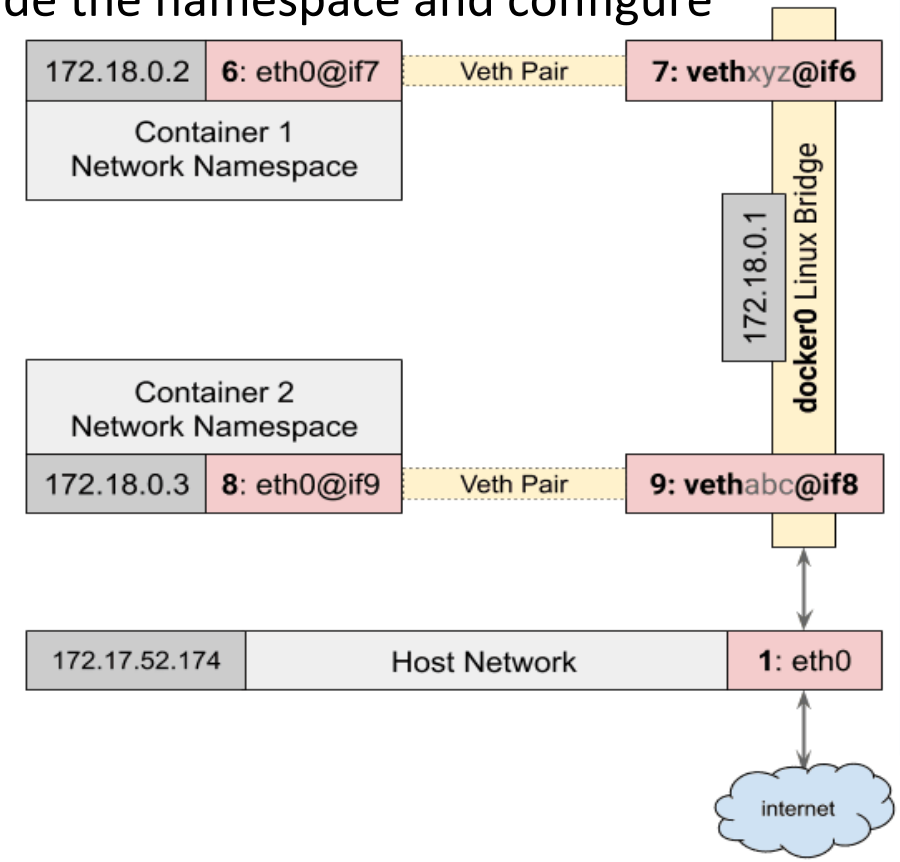
NET namespace

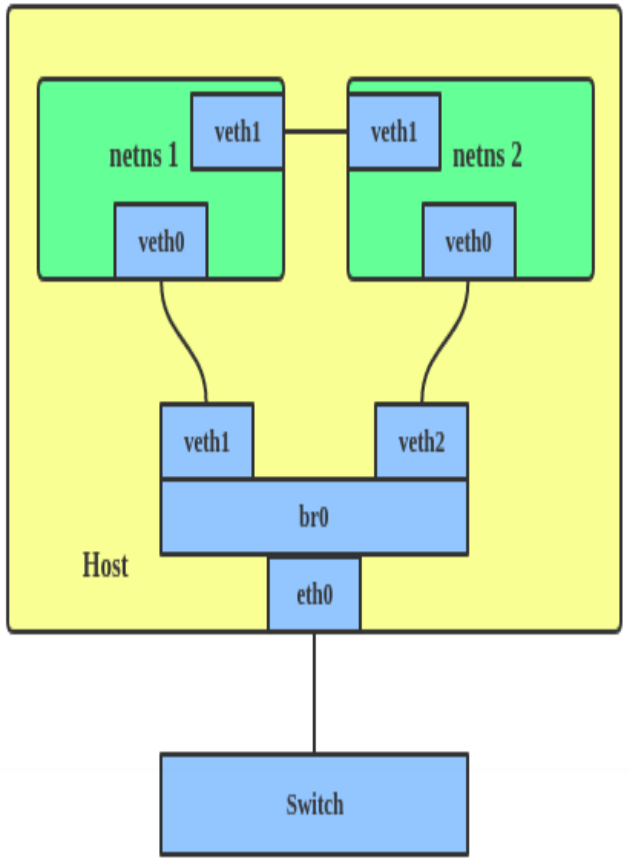
- It's a virtual network barrier encapsulating a process to isolate its network connectivity (in/out) and resources (i.e. network interfaces, route tables and rules) from linux core and other processes.
- It allow processes inside each namespace instance to have access to per namespace network objects
  - Network devices (ethernets)
  - Bridges
  - Routing tables
  - IP addresses
  - ports
- Various commands support network namespace such as ip



NET

- A VETH (virtual Ethernet) configuration as shown can be used when namespaces need to communicate to the main host namespace or between each other.
- veths – create veth pair, move one inside the namespace and configure
- Acts as a pipe between the 2 namespaces
- The VETH (virtual Ethernet) device is a local Ethernet tunnel.
- Devices are created in pairs
- Packets transmitted on one device in the pair are immediately received on the other device.
- When either device is down, the link state of the pair is down.

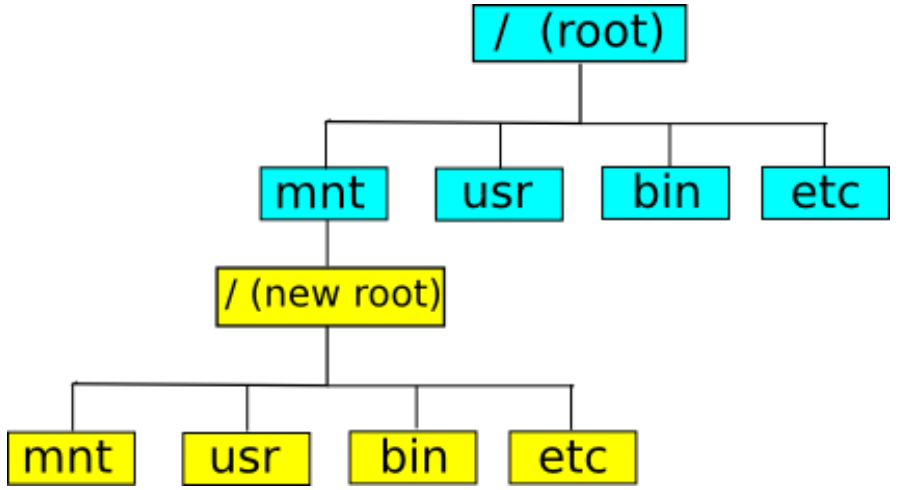




Ref: https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking/#:~:text=by%20Sabrina%20Dubroca.-,VETH,of%20the%20pair%20is%20down
https://dev.to/polarbit/how-docker-container-networking-works-mimic-it-using-linux-network-namespaces-9mj#:~:text=A%20%22veth%20pair%22%20is%20basically,bridge%2C%20creating%20something%20kinda%20LAN..

Filesystem root - chroot name space

- The OS has a file system started at root
  - /bin, and so on, contain programs and libraries
- If you want to run a program which uses a different version of /bin and so on
- This can be achieved using chroot
  - Mount new /bin on some place (say /mnt)
  - Issue chroot /mnt *command*
  - *command* will then see the root as /mnt, not the real root
- Using this technique, can give each process on system its own file system
- Changes the root directory for currently running processes as well as it children for
  - Search paths
  - Relative directories
- Using chroot can be escaped given proper capabilities, thus pivot_root is often used instead
  - chroot; points the processes file system root to new directory
  - pivot_root; detaches the new root and attaches it to process root directory
  - pivot_root info at https://man7.org/linux/man-pages/man8/pivot_root.8.html



Often used when building system images

- chroot to temp directory
- Download and install packages in chroot
- Compress chroot as a system root FS

Cgroups (or Control groups)

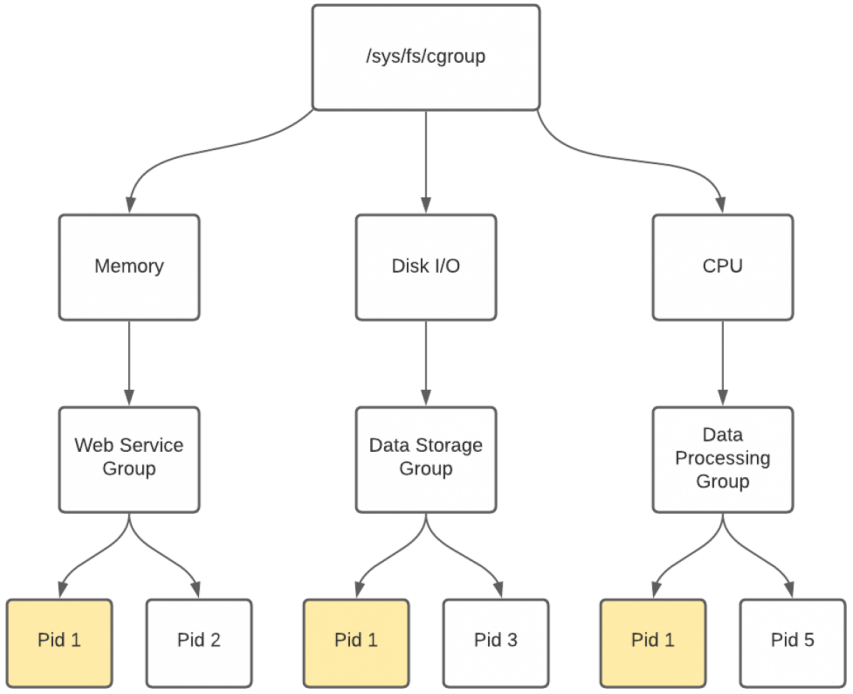
- Cgroup is a linux feature to limit, police, and account the resource usage for a set of processes. Docker uses cgroups to limit the system resources.
- Cgroups - provides mechanisms (fine grain control) to allocate, monitor and limit resources such as ***CPU time, system memory, block IO or disk bandwidth, network bandwidth, or combinations of these resources*** — among user-defined groups of tasks (processes) running on a system.
- Cgroups works on resource types. So it works by dividing resources into groups and then assigning tasks to those groups, deny access to certain resources, and even reconfigure our cgroups dynamically on a running system.
- When you install Docker binary on a linux box like ubuntu it will install cgroup related packages and create subsystem directories.
- Hardware resources can be appropriately divided up among tasks and users, increasing overall efficiency.

Linux cgroups - Functionality

- Access
  - which devices can be used per cgroup
- Resource limiting
  - memory, CPU, device accessibility, block I/O, etc.
- Prioritization
  - who gets more of the CPU, memory, etc.
- Accounting
  - resource usage per cgroup
- Control
  - freezing & check pointing
- Injection
  - packet tagging

Linux cgroups - usage

- cgroups are hierarchically structured where each of the groups are created for a resource with an number.
- Tasks are assigned to cgroups
- Each cgroups has a resource limitation
- There is a hierarchy for each resource

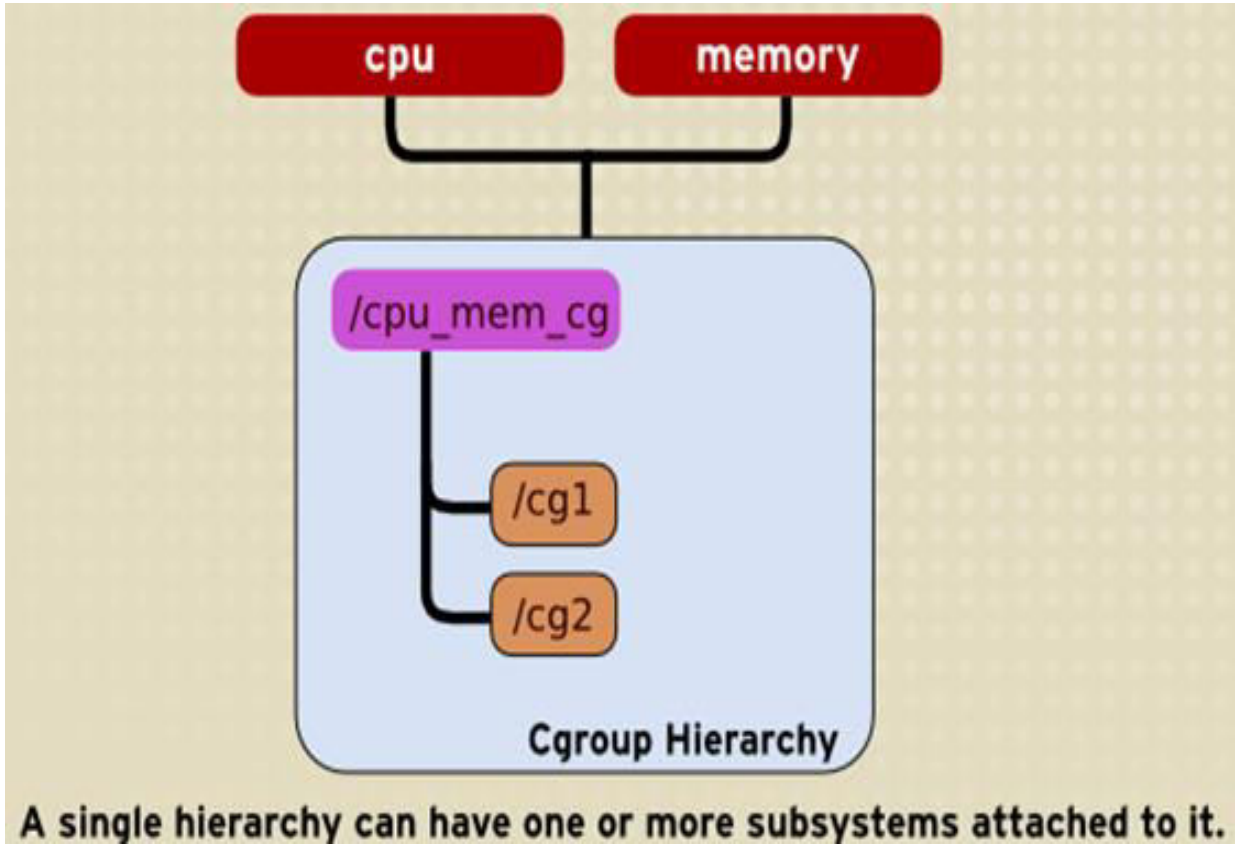


What resources can we limit?

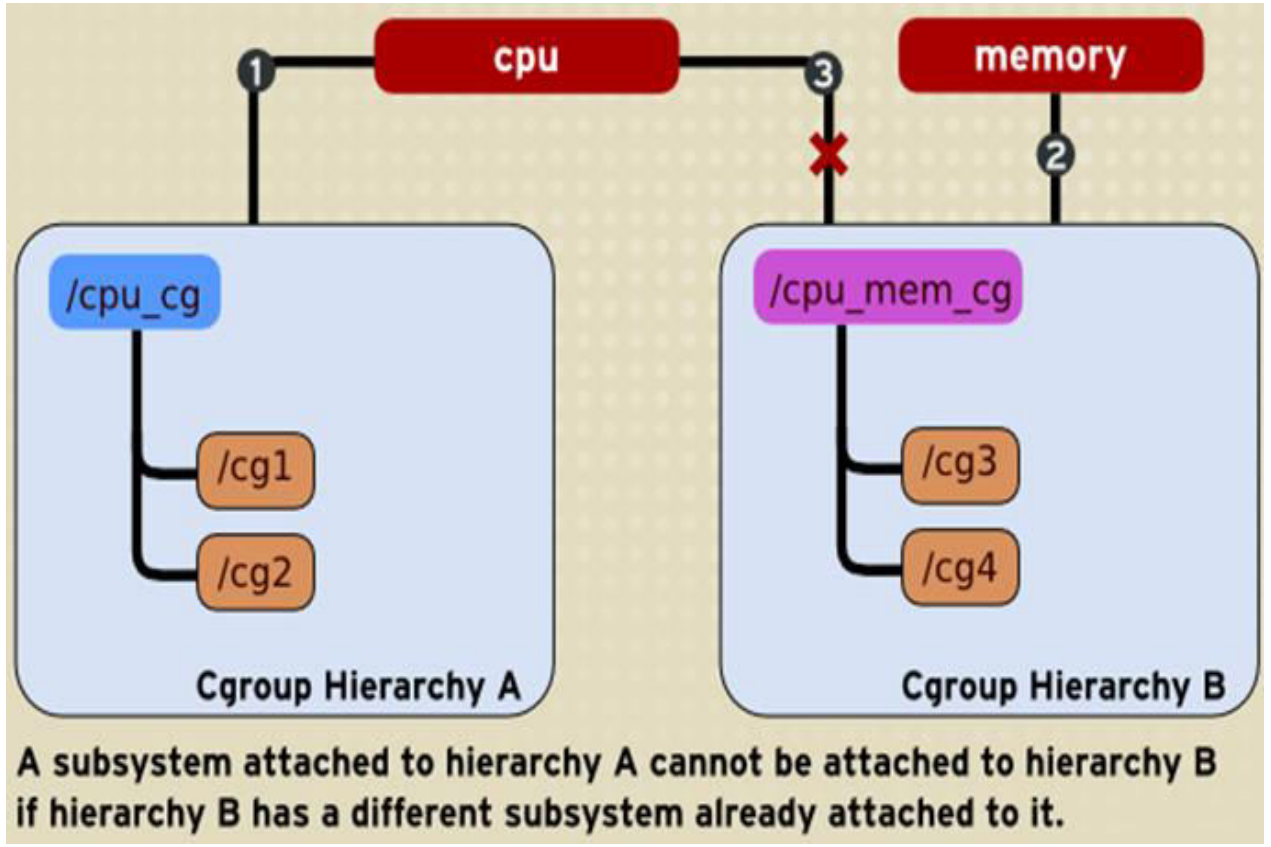
All of the following can be limited for a Cgroup

- Memory
- CPU
- Block IO
- Devices (which of the devices and allowing creation of devices ..)
- Network

cgroup Examples



A single hierarchy can have one or more subsystems attached to it.

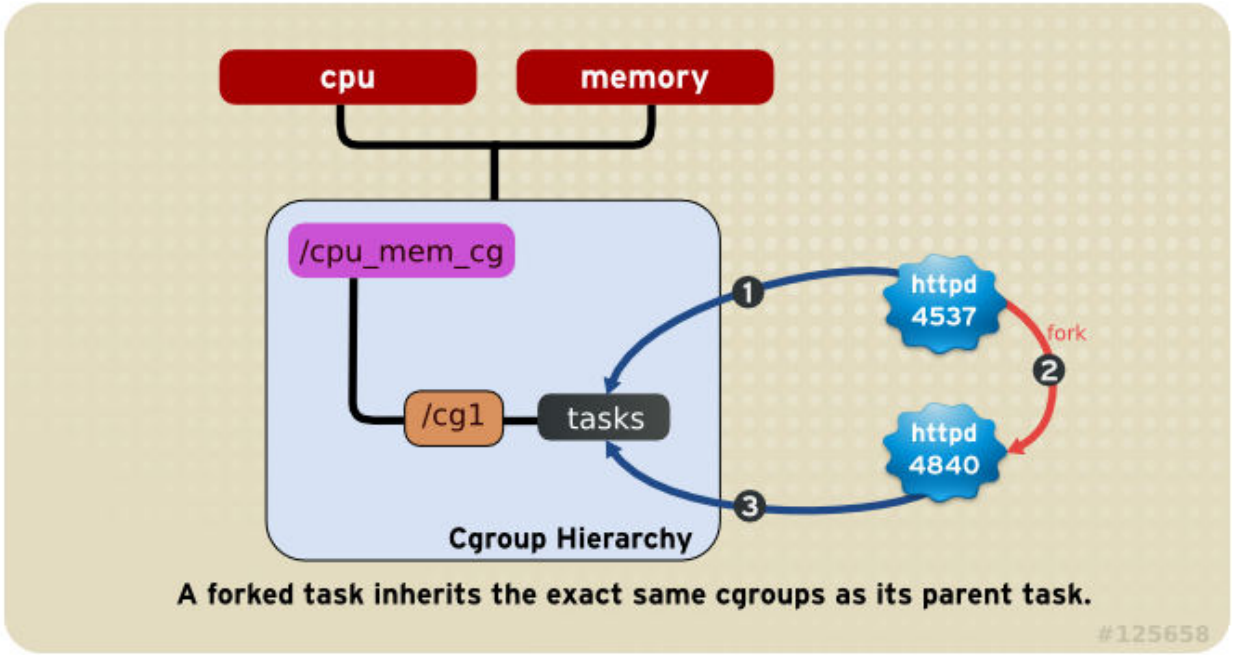


A subsystem attached to hierarchy A cannot be attached to hierarchy B if hierarchy B has a different subsystem already attached to it.

Each cgroup has a resource limitation associated with it

What resources can we limit?

- When a process creates a child process, the child process stays in the same cgroup
  - Good for servers such as NFS
  - Typical operation
    - Receive request
    - Fork child to process request
    - Child terminates when request complete
    - All children will have resource limitation of parent => the resource limitation of parent will apply to processing requests



Container Filesystem

- Programs running inside containers know nothing about image layers.
- From inside a container, the filesystem operates as though it's not running in a container or operating on an image.
- From the perspective of the container, it has exclusive copies of the files provided by the image. This is made possible with something called a *union filesystem (UFS).*
- Docker uses a variety of union filesystems and will select the best fit for your system.
- A union filesystem is part of a critical set of tools that combine to create effective filesystem isolation.
- The other tools are MNT namespaces and the chroot system call.

Union Filesystem

- Union is a type of a filesystem that can create an illusion of merging contents of several directories (created in layers) into one without modifying its original (physical) sources which can be shown in single, merged view
- Union file system operates by creating layers, making them very lightweight and fast.
- Docker Engine uses UnionFS to provide the building blocks for containers.
- Docker Engine can use multiple UnionFS variants, including AUFS, overlay2, btrfs, vfs, and DeviceMapper.

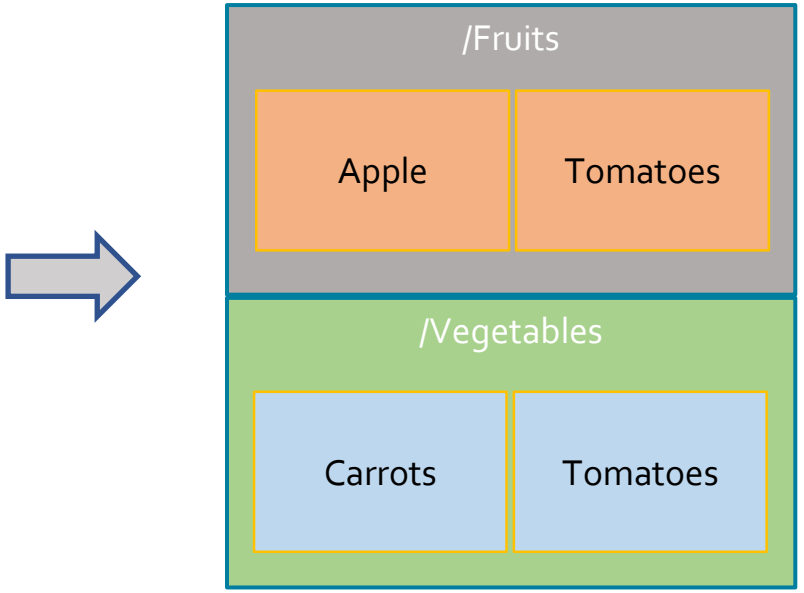
Reference:https://docs.docker.com/get-started/overview/

https://martinheinz.dev/blog/44

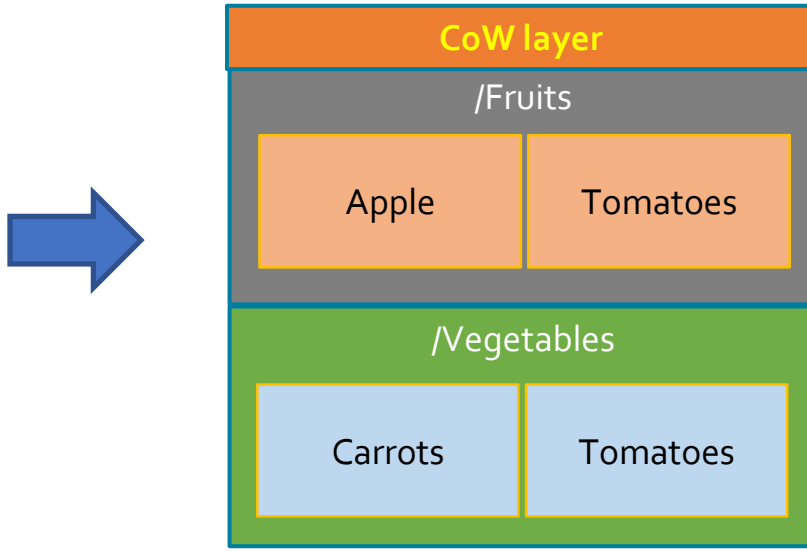
unionfs features - Layering

- unionfs permits layering of file systems
  - */Fruits* contains files *Apple, Tomato*
  - */Vegetables* contains *Carrots, Tomato*

  mount –t unionfs –o dirs=/Fruits:/Vegetables none /mnt/healthy
  - */mnt/healthy* has 3 files – *Apple, Tomato, Carrots*
  - *Tomato* comes from */Fruits* (1st in *dirs* option)
- As if /Fruits is layered on top of /Vegetables



- -o cow option on mount command enables copy on write
  - *If change is made to a file*
  - *Original file is not modified*
  - *New file is created in a hidden location*
- If /Fruits is mounted ro (read-only), then changes will be recorded in a temporary layer

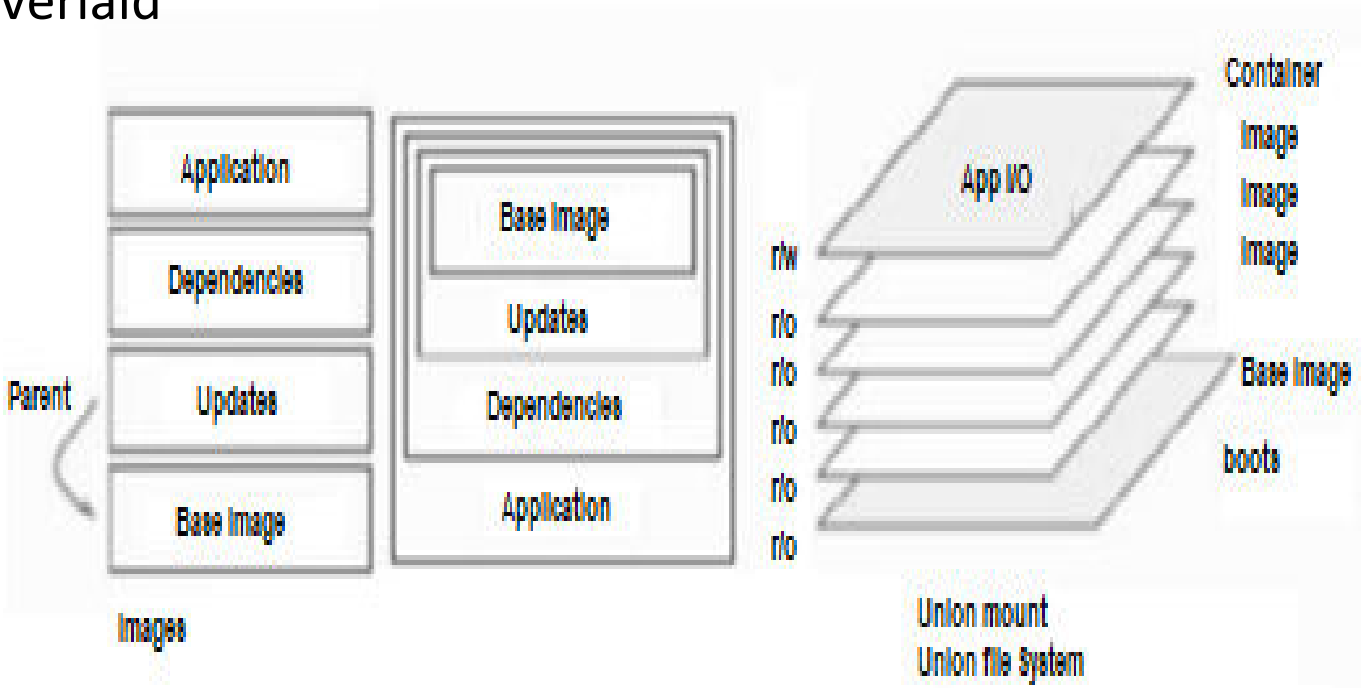


http://www.linuxjournal.com/article/7714

Docker and UnionFS

Incremental Images

- Union FS
  - Files from separate FS (branches) can be overlaid
  - Forming a single coherent FS
  - Branches may be read-only or read-write
- Docker Layers
  - Each layer is mounted on top of prior layers
  - First layer = base image (scratch, busybox, ubuntu, …)
  - A read-only layer = an image
  - The top read-write layer = container



https://www.slideshare.net/cm13/introduction-to-docker-48784005

Weaknesses of Union Filesystem

- Different filesystems have different rules about file attributes, sizes, names, and characters.
- Union filesystems are in a position where they often need to translate between the rules of different filesystems. In the best cases, they're able to provide acceptable translations. In the worst cases, features are omitted.
- Union filesystems use a pattern called *copy-on-write, and that makes implementing* memory-mapped files (the mmap system call) difficult.
- Most issues that arise with writing to the union filesystem can be addressed without changing the storage provider. These can be solved with volumes
- The union filesystem is not appropriate for working with long-lived data or sharing data between containers, or a container and the host.

Summary

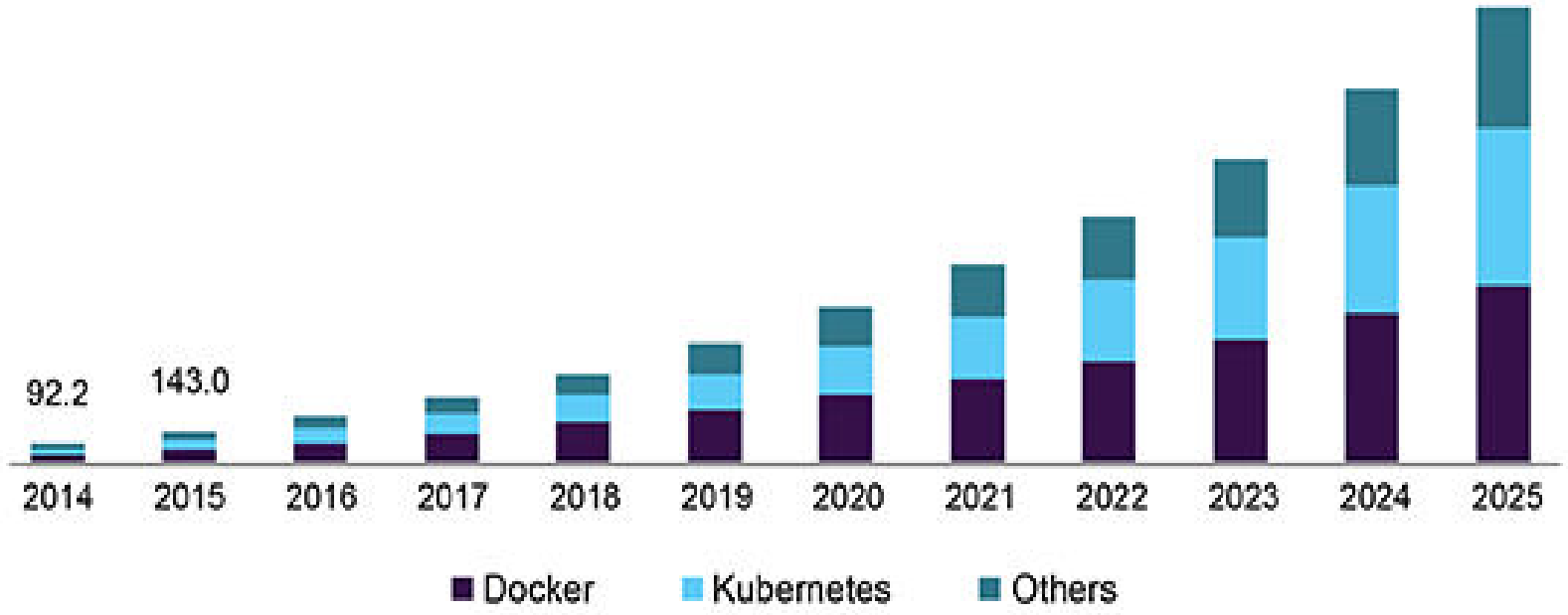
- Use namespaces for controlling resource access
  - Docker has developed their own namespace technology called **namespaces** to provide the isolated workspace called the container.
  - When you run a container, **Docker** creates a set of **namespaces** for that container.
  - These **namespaces** provide a layer of isolation.
- Use cgroups for resource sharing

CLOUD COMPUTING

Container Software

1. Docker
2. AWS Fargate
3. Google Kubernetes Engine
4. Amazon ECS
5. LXC
6. Microsoft Azure
7. Google Cloud Platform
8. Core OS

U.S. application container market size, by platform, 2014 - 2025 (USD Million)



Source: www.grandviewresearch.com

Additional References

Containers vs VMs
https://www.redhat.com/en/topics/containers/containers-vs-vms

Docker container
https://www.simplilearn.com/tutorials/docker-tutorial/what-is-docker-container

Understanding Linux containers
https://www.redhat.com/en/topics/containers
https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html

Additional Reading

- https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-Relationships_Between_Subsystems_Hierarchies_Control_Groups_and_Tasks.html
- https://en.wikipedia.org/wiki/Cgroups
- https://www.youtube.com/watch?v=sK5i-N34im8
  - Cgroups, namespaces and beyond: What are containers made from – Jerome Pettazzoni, Docker

  https://www.youtube.com/watch?v=nXV6qihj5uw

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

CLOUD COMPUTING

DevOps

Dr. H.L. Phalachandra

Prof. Venkatesh Prasad

Department of Computer Science and Engineering

Acknowledgements:

Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

Definition

- DevOps is a set of practices that combines software development and IT operations (development and operations) where each of them are independent specialist jobs.
- It aims to shorten the system development life cycle (SDLC) and increase an organizational ability to deliver (Deploy and Support) applications and services
- DevOps is complementary with Agile software development; several DevOps aspects came from Agile methodology

DevOps Definition (Cont.)

- **DevOps** is a set of practices that automates the processes between software development and IT teams, in order that they can build, test, and release software faster and more reliably

Development Org

Operations Org

Have quite different goals and incentives, which often conflicted with each other.

- It follows Plan, Build, Deploy, Operate cycle.
- Docker and Kubernetes are 2 of 10 most used DevOps tools now. Git is another one.

Motivational Problem

- You have built and tested the containerized application
- You learnt how to deploy your application using Docker
- As customers require more features,
  - What do you do?
- Let’s look at some engineering issues

https://www.iltam.org/check_download_nopass.php?forcedownload=1&file=files/DevOpsconfpresentationSandrine1.pptx&no_encrypt=true&dlpassword=20426

SDLC – Software Development Life Cycle

- Customer generates requirements
- Do a design
- Different teams work on different components
- Integrated together
- Then tested for
  - Functionality
  - Similarity to customer deployment
- Then for deployment, need to talk to customers IT team.
  - Manages deployment, scaling, run time issues with the application

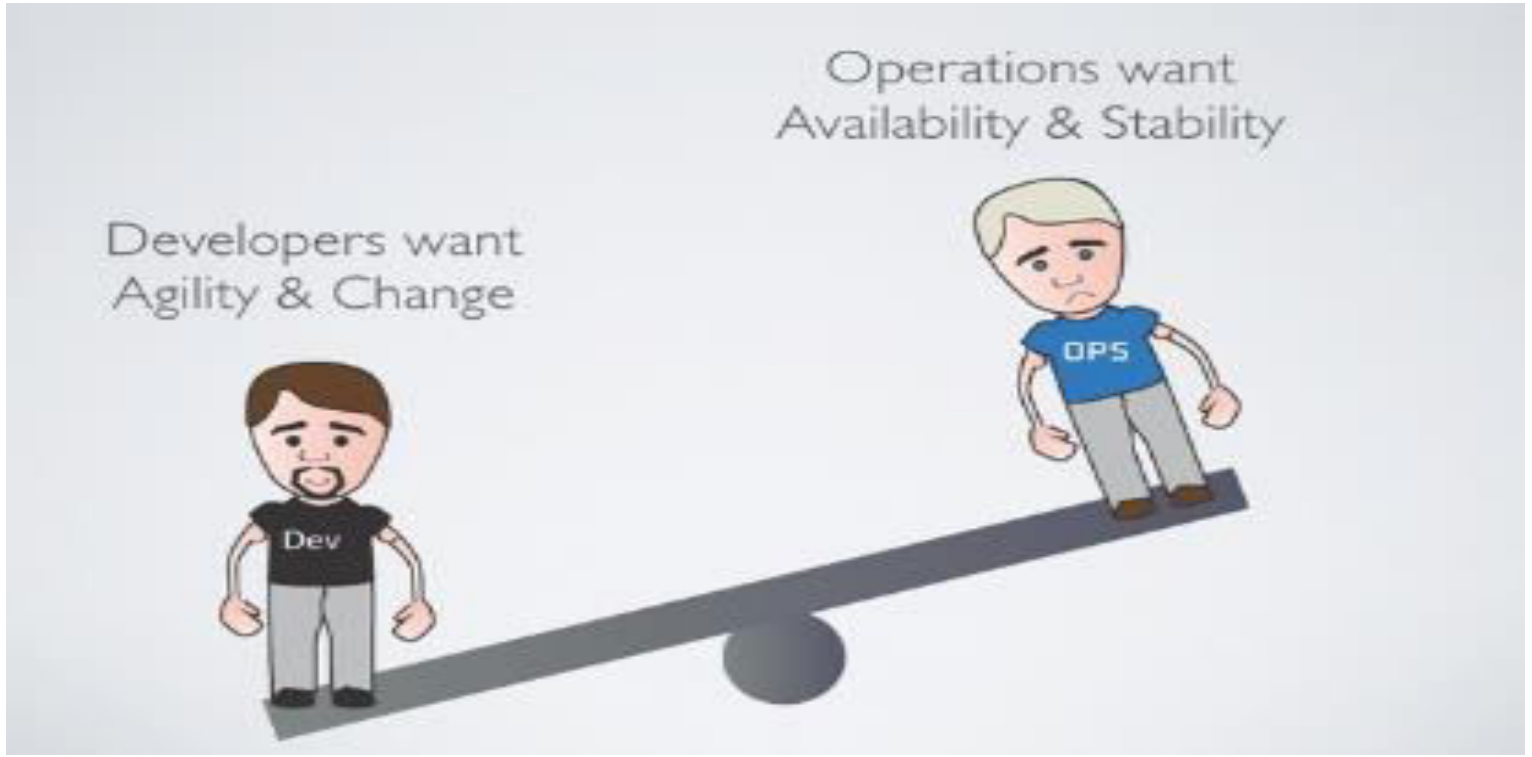
CLOUD COMPUTING

Dev & Ops Typical Conversation

Put the current release live, NOW!
It works on my machine
We need this Yesterday
You are using the wrong version

- What are the dependencies?
- No machines available…
- Which DB?
- High Availability?
- Scalability?

Make the balance between Dev & Ops

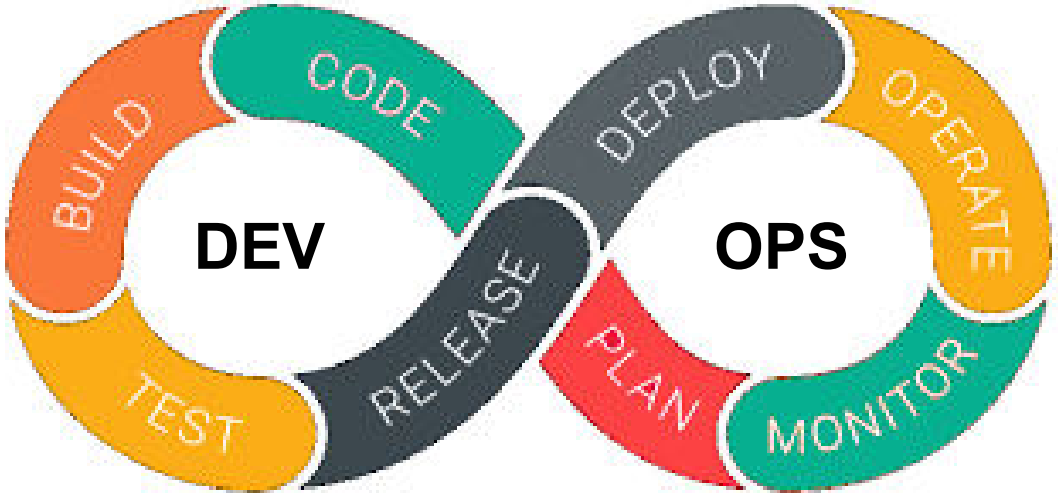


In pre-cloud era, requires a lot of manual intervention like procuring machines, etc.

With cloud – automation is possible

Cloud SDLC

- Customer generates requirements
- Do a design
  - Different teams work on different components
  - Integrated together
  - Then tested for
    - Functionality
    - Similarity to customer deployment
  - Then for deployment, need to talk to cloud IT team.
    - Handles deployment, scaling, monitoring



So, what are the challenges?

CLOUD COMPUTING

Advantages

- Faster Time to Market – *checkins more often, so can build and test more often*.
- Quality of Code/Release - *improved because of frequent testing*
- Integration *Often*
- Deploy Often – *changes reach the customers often*
- Automated – *Repeatable and faster*
- Every one is happy (Hopefully!!)

Automation : Principles of Continuous Delivery

Principle #1

Every build is a potential release

Principle #3

Automate wherever possible

Principle #2

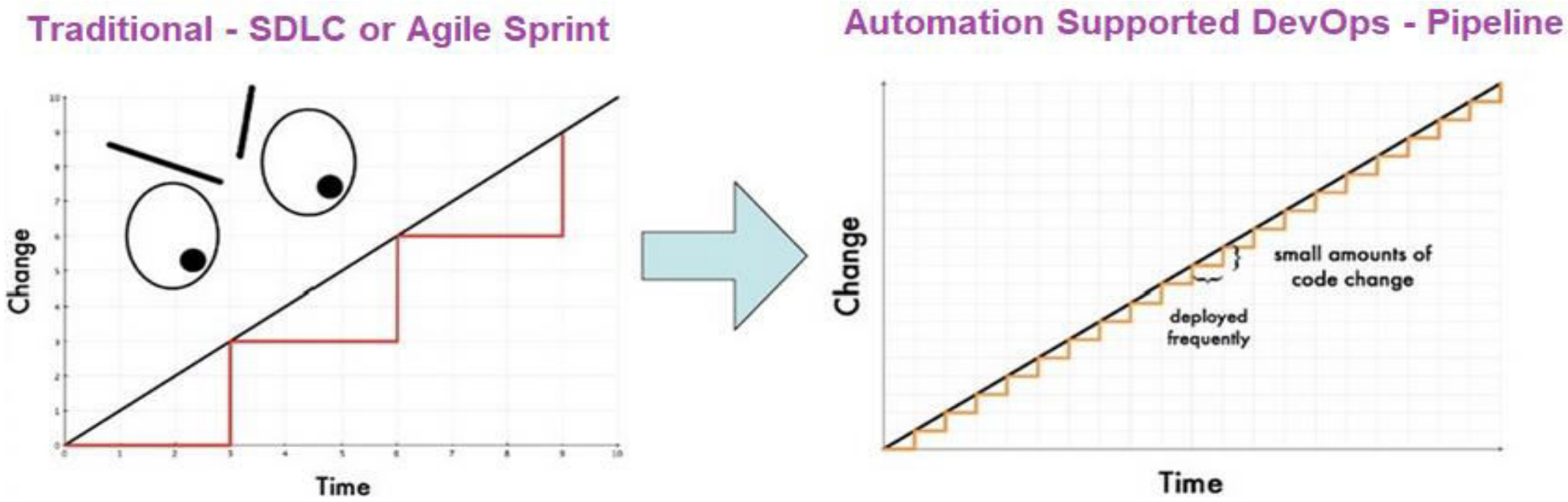
Eliminate manual bottlenecks

Principle #4

Have automated tests you can trust

Keep it Simple

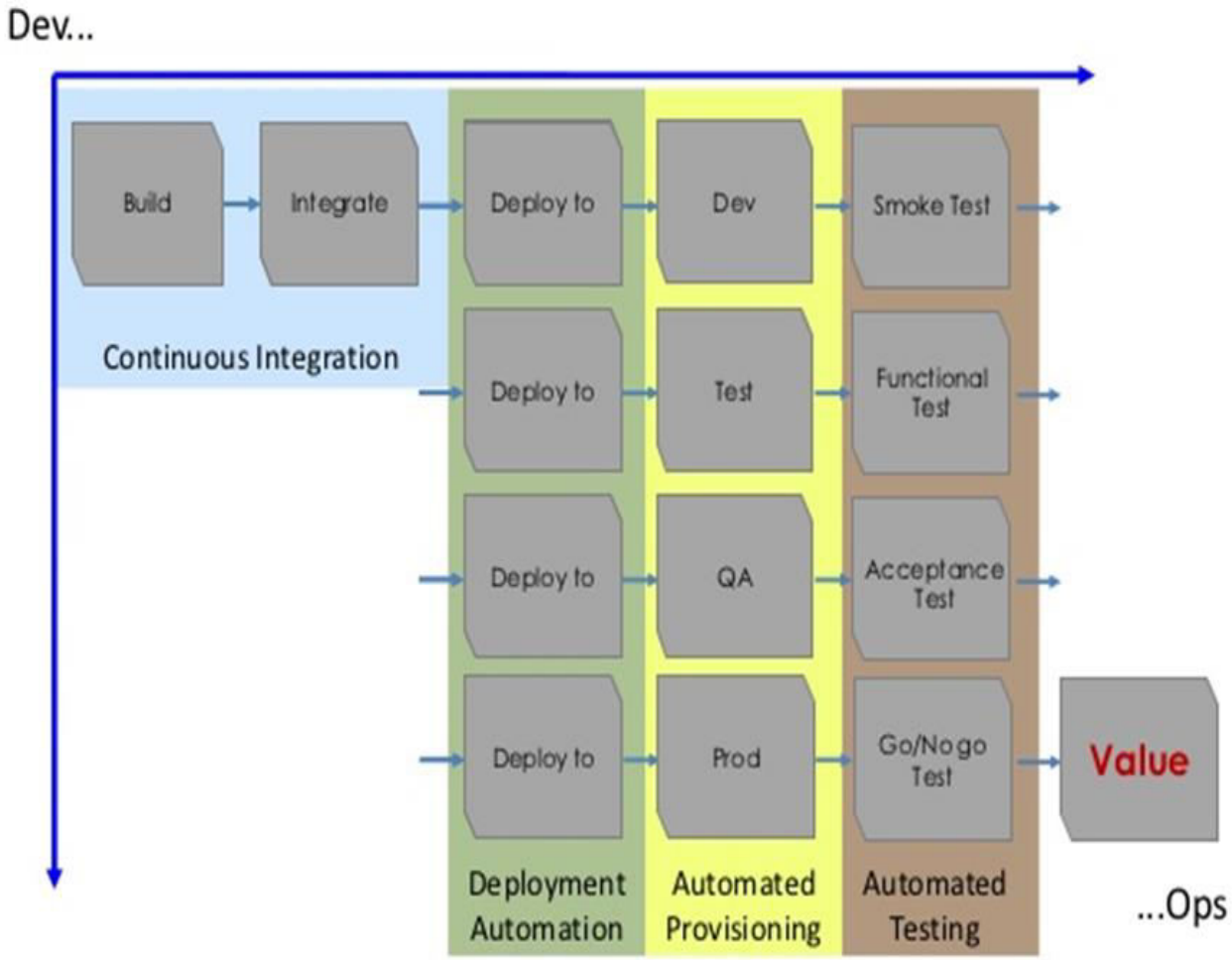
- Reduce risk of Release
- Make small changes and Test it every cycle
- Small changes can make the difference for Ops & Dev



CLOUD COMPUTING

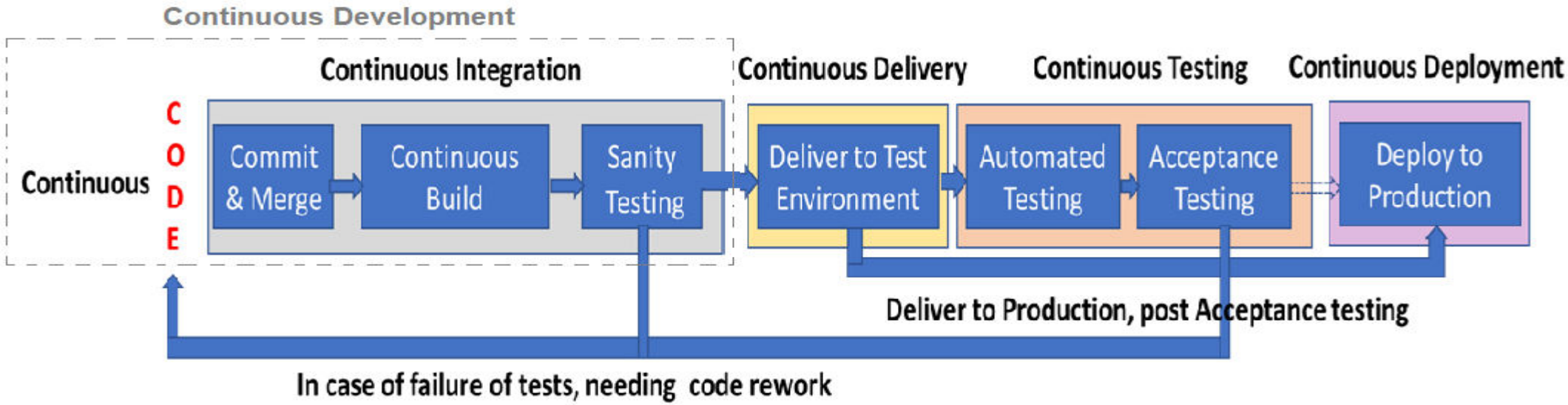
Automate all the things

- Programming Model to enable scale
- Manage source code
- Reproducible Build
  - Build on a Prod-Like environment
  - No more “Works on my machine”
- Test
  - Testing reduces risks
  - Make you more confident
- Deploy
  - Deploy to Dev
  - Deploy to QA
  - Deploy to Pre-Prod
  - Prod



CLOUD COMPUTING

DevOps Pipeline



Typical pipeline for containerized applications might look like the following:

1. A developer pushes his/her code changes to Git.

2. The build system automatically builds the current version of the code and runs some sanity tests.

3. If all tests pass, the container image will be published into the central container registry.

4. The newly built container is deployed automatically to a staging environment.

5. The staging environment undergoes some automated acceptance tests.

6. The verified container image is deployed to production.

Continuous Development

- This includes continuous code development, continuous integration and continuous Build
  - Developers distributed across geographies
  - Consistent coding style
  - Need to keep the changes in sync
  - Can’t rely on mailing individual changes
  - Everyone needs to have a common view of the code.
- **Continuous integration (CI):** the automatic merging of the code, Requires every module to compile (Build) correctly and then do sanity testing of developers’ changes against the mainline branch.
- If you’re making changes on a branch that would break the build when merged to the mainline, continuous integration will let you know that right away, rather than waiting until you finish your branch and do the final merge.
- Tool : Source Code Version Management Systems like git, ClearCase, Maven, ANT, Jenkins, Travis CI, or Drone are often used to automate build pipelines

Continuous Delivery / Continuous Testing / Continuous Deployment

Continuous delivery is the frequent shipping of sanitized builds to a given environment (such as test or production) for automatic developments.

- This could be done using tools like Jenkins the open source Java based application which automates many parts of the software delivery pipeline

Continuous Testing

- Continuous testing is the process of executing predominantly automated tests as part of the software delivery pipeline for validating the code to ensure that the application works as envisaged and is free of bugs or defects and can be continuously deployed
- These tests are typically designed to be executed with minimum wait times and provide the earliest possible feedback & support detection (or prevention) of the risks that are most critical for the business or organization that is associated with the software release candidate.

(Cont.)

Continuous testing (Cont.)

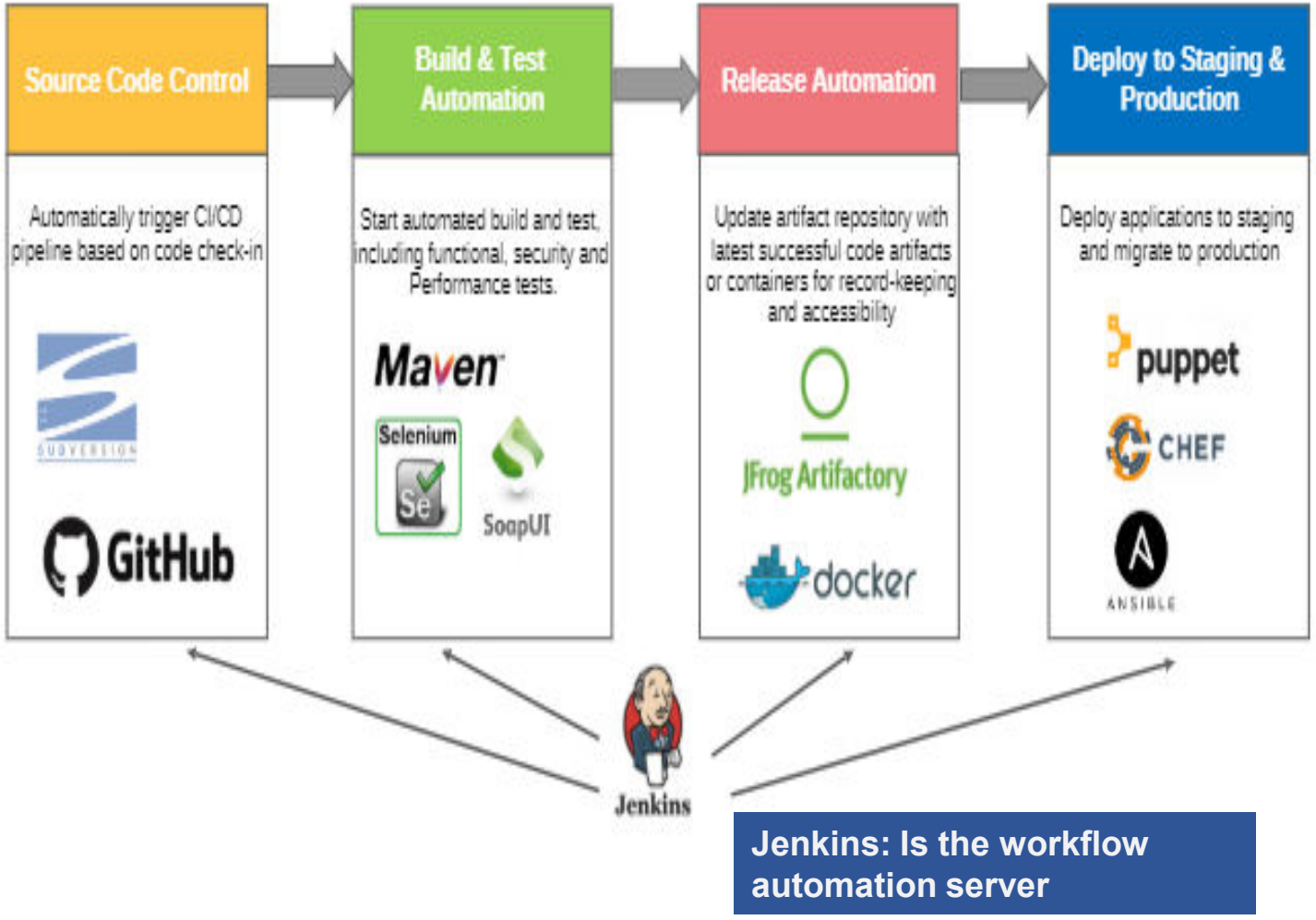
This would involve activities like

- Copy individual binaries to staging environment to the right directories
- Configure it correctly, ensure dependent libraries are installed
- Execute different categories of the planned tests (e.g. Integration tests, functionality, performance or acceptance tests) (Could use tools like Junit, Selenium)
- Generate reports
- **Continuous deployment (CD)** is the automatic deployment of successful builds to production managed centrally.
- Developers should be able to deploy new versions by either pushing a button, or merging a merge request, or pushing a Git release tag.

CLOUD COMPUTING

Continuous Deployment (Cont.)

- This is when the application is actually used by the customer Setup the environment
- Specify the type of infrastructure that you need in terms of machine types – Chef Provisioning (Machines, storage)
- Consider Immutable infrastructure which does not change enabling easier automation
- Specify the configuration that you need – Puppet, Chef Recipe/Cookbooks
- The set of docker images and their scaling policy, setup load balancer etc.



Jenkins: Is the workflow automation server

Tools

Tools to setup the infrastructure: Chef/Puppet/Kubernetes

Docker – create a docker image with all dependencies for each of the services

Jenkins

- **Jenkins** is a very widely adopted CD tool. There is also a newer dedicated side project for running Jenkins in Kubernetes cluster, **JenkinsX**.
- Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.
- Jenkins can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed.

Additional References

https://www.guru99.com/jenkins-tutorial.html

https://www.guru99.com/devops-tutorial.html

References: (useful for Cloud lab experiment)
https://www.jenkins.io/doc/
https://www.jenkins.io/doc/book/pipeline/
https://jenkins-x.io/

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu

CLOUD COMPUTING

Orchestration and Kubernetes

Dr. H.L. Phalachandra

Prof. Venkatesh Prasad

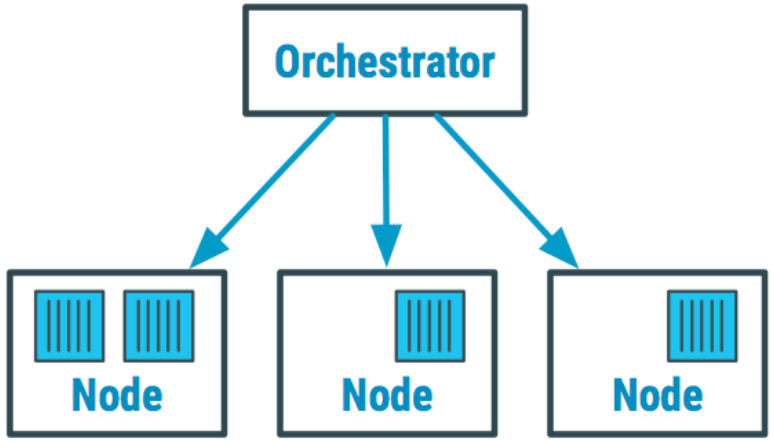
Department of Computer Science and Engineering

Acknowledgements:

Most information in the slide deck presented through the Unit 2 of the course have been created by **Prof. Venkatesh Prasad** and would like to acknowledge and thank him for the same. There have been some information which I might have leveraged from the content of **Dr. K.V. Subramaniam's, Dr. Arkaprava Basu and Dr. Sorav Bansal's** lecture contents too. I may have supplemented the same with contents from books and other sources from Internet and would like to sincerely thank, acknowledge and reiterate that the credit/rights for the same remain with the original authors/publishers only. These are intended for class room presentation only.

Container and Orchestration Recap - Basics

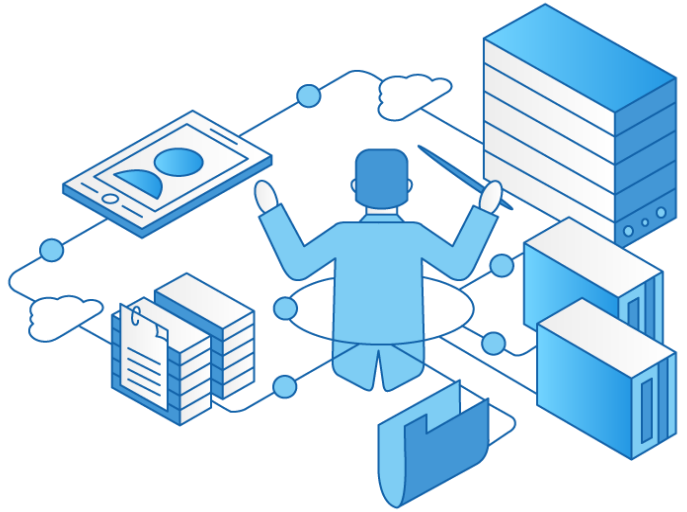
- **Containers** offer a way to package code, a runtime, system tools, system libraries, and configs altogether. This shipment is a lightweight, standalone executable. This way, your application will behave the same every time no matter where it runs (e.g. Ubuntu, Windows, etc.).
- Container orchestration automates the deployment, management, scaling, and networking of containers.This could involve deploying containers on a compute cluster consisting of multiple nodes
- Container orchestration can be used in any environment where you use containers, to help deploy the same application across different environments without needing to redesign it.



- Managing the lifecycle of containers with Orchestration supports DevOps teams to integrate into CI/CD workflows
- Container Management tools extend lifecycle management capabilities to complex, multi-container workloads deployed on a cluster of machines

https://devopedia.org/container-orchestration

Container orchestration



- **Container orchestrator:** Its a piece of centralized management software that allocates and schedules containers to run on a pool of servers (different machines in a cluster):
  - Operations teams, too, find their workload greatly simplified by containers.
  - Instead of having to maintain a sprawling estate of machines of various kinds, architectures, and operating systems, all they have to do is run a *container orchestrator*
- The terms ***Orchestration*** and ***Scheduling*** are often used loosely as synonyms.
  - Strictly though, orchestration in this context means coordinating and sequencing different activities in service of a common goal (like the musicians in an orchestra).
  - Scheduling means managing the resources available and assigning/deploying workloads where they can most efficiently be run.

Container orchestration (Cont.)

- An important activity of container orchestration is cluster management: joining multiple physical or virtual servers into a unified, reliable, fault-tolerant, apparently seamless group.
- Containers providing this computation environment is considered **Immutable**
  - Immutable infrastructure is an infrastructure paradigm in which servers are never modified after they're deployed.
  - If something needs to be updated, fixed, or modified in any way, new servers built from a common image with the appropriate changes are provisioned to replace the old ones.
  - The benefits of an immutable infrastructure include more consistency and reliability in your infrastructure and a simpler, more predictable deployment process.
  - It mitigates or entirely prevents issues that are common in mutable infrastructures, like configuration drift
  - It also enables comprehensive deployment automation, fast server provisioning in a cloud computing environment.

Container orchestration (Cont.)

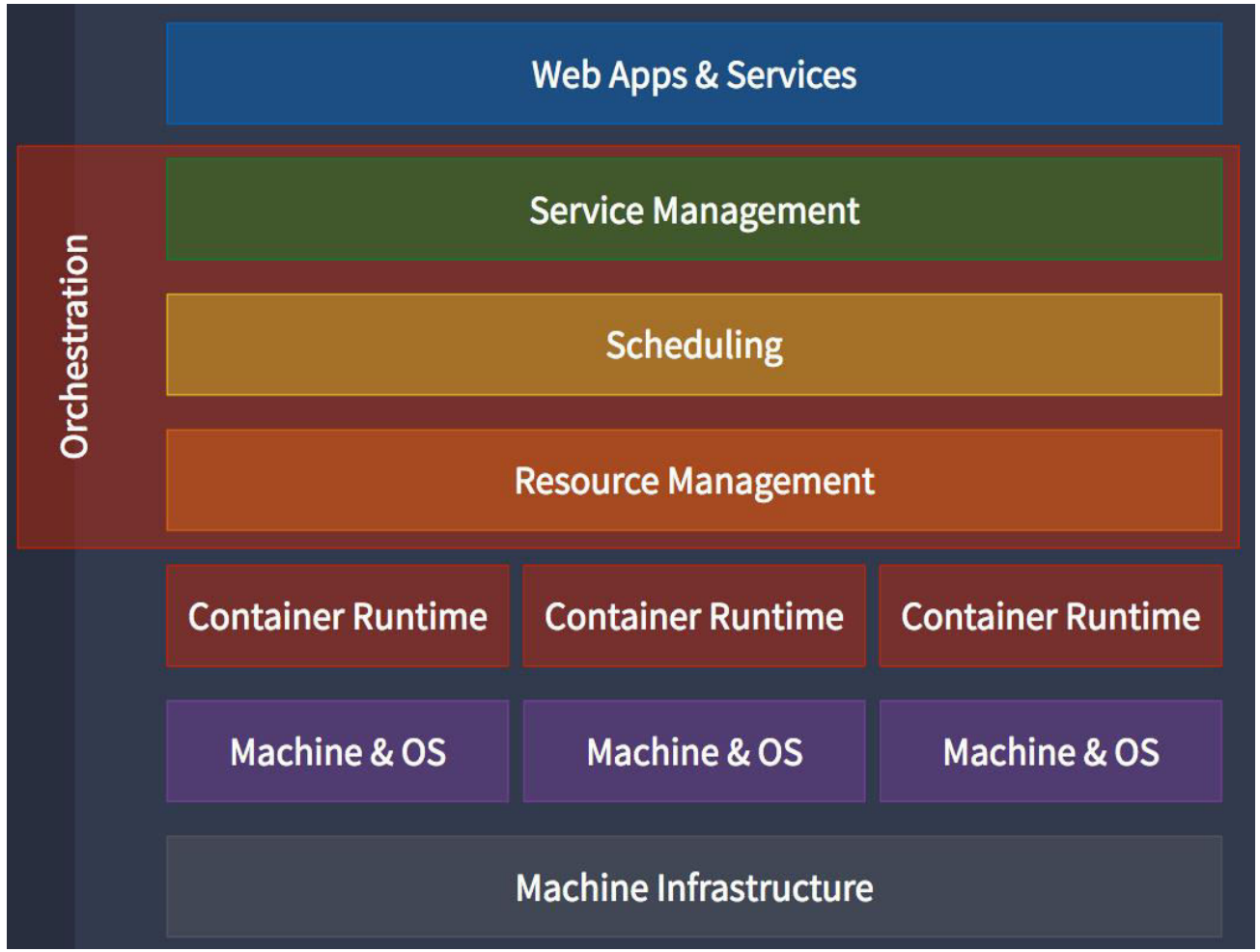
- **Kubernetes** is the most pervasive container orchestration platform to address these challenges.
  - Kubernetes is an open-source container management (orchestration) tool.
  - It's container management responsibilities include container deployment, scaling & descaling of containers & container load balancing.
  - It lets you manage complex application deployments quickly in a predictable and reliable manner.

Container orchestration (Cont.)

- **Container orchestration** is used to control and automate many tasks:
  - Provisioning and deployment
  - Configuration and scheduling
  - Resource allocation between containers and manage the networking between them
  - Container availability and redundancy management
  - Scaling or removing containers based on balancing workloads across host infrastructure
  - Load balancing and traffic routing
  - Monitoring container and host health
  - Configuring applications based on the container in which they will run
  - Keeping interactions between containers secure
  - Migration of containers from one host to another for supporting failures, shortage of resources etc.

Where does Container orchestration fit within the system stack?

Container Orchestration mediates between the apps/services and the container runtimes



- **Service Management:** **Labels** *(represents Key-Value pairs that are attached to objects such as pods and used to specify identifying attributes of objects)* , **groups** (*collection of related functionality*), **namespaces** (*a way to organize clusters into virtual sub-clusters*), **dependencies**, **load balancing**, **readiness checks**.
- **Scheduling:** Allocation, replication, (container) resurrection, rescheduling, rolling deployment, upgrades, downgrades.
- **Resource Management:** Memory, CPU, GPU, volumes, ports, IPs.

Container orchestration – Different Tools which support Orchestration

Docker Swarm: Provides native clustering functionality for Docker containers, which turns a group of Docker engines into a single, virtual Docker engine.

Google Container Engine: Google Container Engine, built on Kubernetes, can run Docker containers on the Google Cloud.

Kubernetes: An orchestration system for Docker containers. It handles scheduling and manages workloads based on user-defined parameters.

Amazon ECS: The ECS supports Docker containers and lets you run applications on a managed cluster of Amazon EC2 instances.

Kubernetes (also know as K8s) Basics

PES UNIVERSITY

Kubernetes as seen earlier is a container or microservice platform that orchestrates computing, networking, and storage infrastructure workloads. Kubernetes extends how we scale containerized applications so that we can enjoy all the benefits of a truly immutable infrastructure

Understanding the K8S, will need the understanding the objects used as the building block of the application lifecycle. This . This include

1. **Basic Building objects**
   - **Pod**. A group of one or more containers.
   - **Service**. An abstraction that defines a logical set of pods as well as the policy and an endpoint for accessing them.
   - **Volume**. An abstraction that lets us persist data. (This is necessary because containers are ephemeral—meaning data is deleted when the container is deleted.)
   - **Namespace**. A segment of the cluster dedicated to a certain purpose, for example a certain project or team of devs. Typically used to create multiple virtual Kubernetes clusters within a single cluster

2. **Controllers** which are several higher-level abstractions
   Basic K8S objects include
   - **ReplicaSet (RS).** Ensures the desired amount of pod is what's running.
   - **Deployment.** Offers declarative updates for pods in an RS.
   - **StatefulSet.** A workload API object that manages stateful applications, such as databases.
   - **DaemonSet.** Ensures that all or some worker nodes run a copy of a pod. This is useful for daemon applications like Fluentd.
   - **Job.** Creates one or more pods, runs a certain task(s) to completion, then deletes the pod(s)

https://www.bmc.com/blogs/what-is-kubernetes/

http://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Kubernetes Building Blocks : Pod

- Pod is a single or group of containers that share storage and network within a Kubernetes configuration, telling those containers how to behave.
- These are group of containers which are running microservices, of an elastic application and which are communicating with each other (typically, in a non-containerized setup run together on one server).
- Pods share storage, Linux namespaces, Cgroups, IP, port address space and can communicate with each other over localhost networking.
- Each pod is assigned an IP address on which it can be accessed by other pods within a cluster. Applications within a pod have access to shared volumes – helpful for when you need data to persist beyond the lifetime of a pod.
- The containers are co-located, hence share resources and are always scheduled together.
- Pods are not intended to live long. They are created, destroyed and re-created on demand, based on the state of the server and the service itself.

Kubernetes Building Blocks : Pod (Cont)

- A Pod is scheduled or assigned to a node by the scheduler and the kubelet starts creating containers for that Pod using a container runtime. There are three possible container states: Waiting, Running, and Terminated.
- Pods follow a defined lifecycle, starting in the Pending phase, moving through Running if at least one of its primary containers starts OK, and then through either the Succeeded or Failed phases depending on whether any container in the Pod terminated in failure.
- Pods are only scheduled once in their lifetime. Once a Pod is scheduled (assigned) to a Node, the Pod runs on that Node until it stops or is terminated
- Whilst a Pod is running, the kubelet is able to restart containers to handle some kind of faults. Within a Pod, Kubernetes tracks different container states and determines what action to take to make the Pod healthy again.
- Volumes exists as long as that specific Pod (with that exact UID) exists unless its
  an persistent volume for shared storage between the containers
- Pods do not, by themselves, self-heal. If a Pod is scheduled to a node that then fails, the Pod is deleted;

https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

Kubernetes - Service

Services is coupling of a set of pods to a policy by which to access them. Services are used to expose containerised applications to originations from outside the cluster

- Pods in a Kubernetes deployment are regularly created and destroyed, causing their IP addresses to change constantly. It will create discoverability issues for the deployed application, making it difficult for the application frontend to identify which pods to connect.
- To address this K8s introduced the concept of a service, which is an abstraction on top of a number of pods which has a Virtual IP address assigned to it which is exposed and used by the service proxies running on top of it.
- Other services communicate with the service via this Virtual IP address, and the service keeps track of the changes in IP addresses and DNS names of the pods and map them to this virtual IP.
- A Kubernetes service is a logical collection of pods in a Kubernetes cluster.
- This is where we can configure load balancing for our numerous pods and expose them via a service.

Kubernetes Architecture

A K8S cluster is made of a

- **Master node**, which exposes the API, schedules deployments, and generally manages the cluster. This is the entry point of all administrative tasks. The master node is the one taking care of orchestrating the worker nodes, where the actual services are running.
- ***Worker nodes*** *(multiple of them)* can be responsible for container runtime, like Docker.

Master node components

- These master components comprise a master node:
  - **Kube-apiserver**. Exposes the API is the entry points for all the REST commands used to control the cluster
  - **etcd.** Key value stores all cluster data. (Can be run on the same server as a master node or on a dedicated cluster) It's mainly used for shared configuration and service discovery with info like jobs being scheduled, created and deployed, pod/service details and state, namespaces and replication information
  - **Kube-scheduler**. Has information on the resources available. Schedules new pods on worker nodes.
  - **Kube-controller-manager**. Runs the controllers. Uses apiserver to watch the shared state of the cluster and makes corrective changes to the current state to change it to the desired one.
    E.g. of a controller is the Replication controller, which takes care of the number of pods in the system as configured by the user.

Kubernetes Architectu

Worker node components

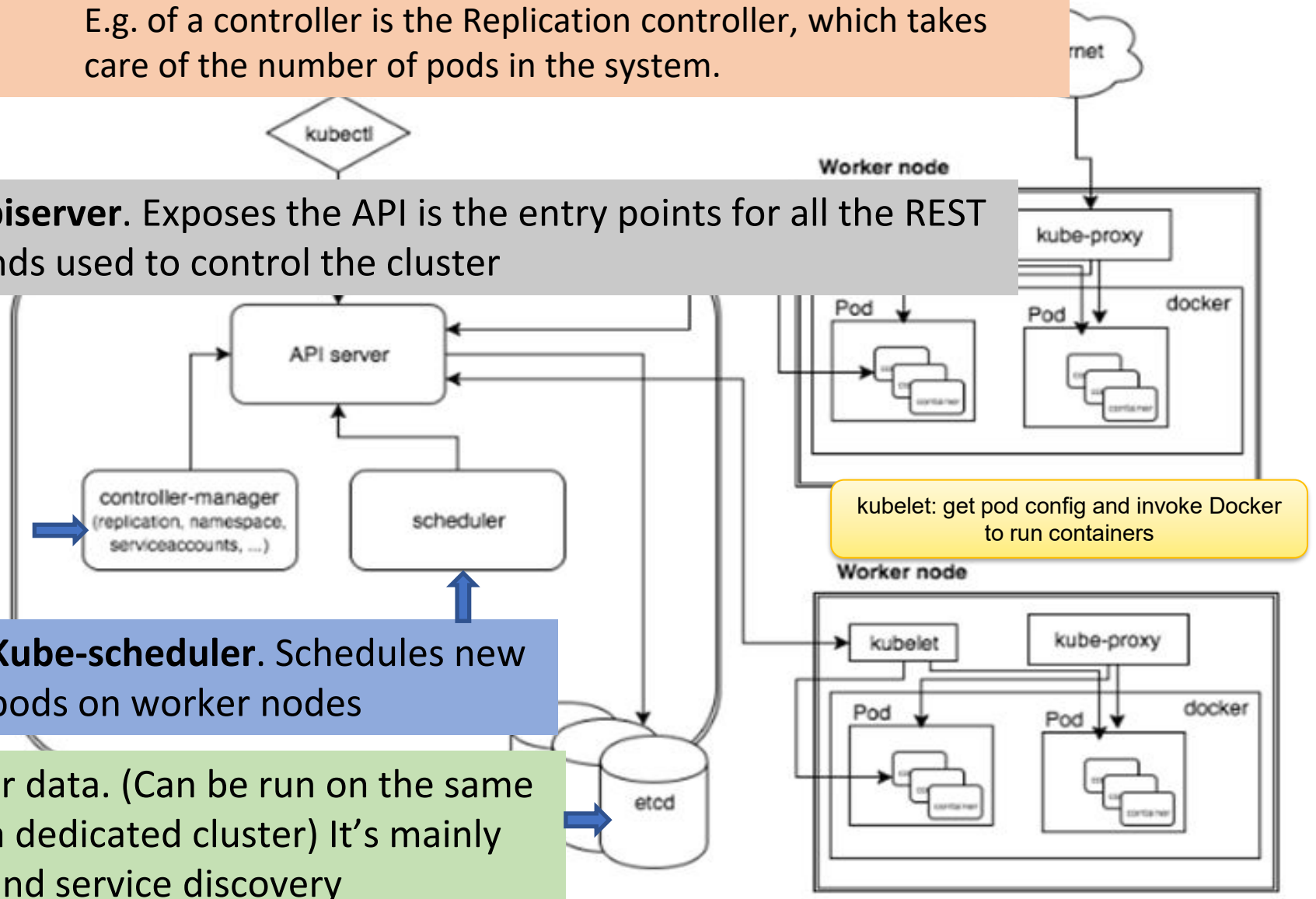
- **Kubelet**. Agent that gets the configuration of a pod from the apiserver that the descr are up and running.
- **Kube-proxy**. acts as a network proxy and a load balancer for a service on a single worker node. It takes care of the network routing for TCP and UDP packets..
- **Container runtime**. Runs containers.

- **Kube-controller-manager**. Runs the controllers. Uses apiserver to watch the shared state of the cluster and makes corrective changes to the current state to change it to the desired one. E.g. of a controller is the Replication controller, which takes care of the number of pods in the system.

- **Kube-apiserver**. Exposes the API is the entry points for all the REST commands used to control the cluster

Kube-scheduler. Schedules new pods on worker nodes

- **etcd.** Key value stores all cluster data. (Can be run on the same server as a master node or on a dedicated cluster) It's mainly used for shared configuration and service discovery



https://x-team.com/blog/introduction-kubernetes-architecture

Kubernetes Functioning

- Kubernetes, you will describe the configuration of an application using say a JSON file. The configuration file tells the configuration management tool where to find the container images, how to establish a network, and where to store logs.
- When deploying a new container, the container management tool automatically schedules the deployment to a cluster and finds the right host, taking into account any defined requirements or restrictions. The orchestration tool then manages the container's lifecycle based on the specifications that were determined in the compose file.

Kubernetes Benefits

- **Horizontal scaling.** Scale your application as needed from command line or UI.
- **Automated rollouts and rollbacks.** Roll out changes in a controlled fashion while monitoring the health of your application in the container and if something goes wrong, K8S automatically rolls back the change.
- **Service discovery and load balancing.** Kubernetes can expose a containers using a DNS name or IP and based on the load, can do load balancing by distribute traffic.
- **Storage orchestration.** Automatically mount local or public cloud or a network storage.
- **Secret and configuration management.** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
- **Self-healing.** The platform heals many problems: restarting failed containers, replacing and rescheduling containers as nodes die, killing containers that don't respond to your user-defined health check, and waiting to advertise containers to clients until they're ready.
- **Batch execution.** Manage your batch and Continuous Integration workloads and replace failed containers.
- **Automatic binpacking.** Automatically schedules containers based on resource requirements like CPU and memory which the containers need along with other constraints into the same host

THANK YOU

Dr. H.L. Phalachandra

phalachandra@pes.edu