Cloud Computing

Dr.P.Chinnasamy
Department of CS/IS

BITS Pilani
Pilani Campus

innovate achieve lead

BITS Pilani
Pilani Campus

CSI ZG527/ SE ZG527 –Memory Virtualization Problem, Container– L5

Recap

innovate achieve lead

- ✓ Hardware Assisted Virtualization
- ✓ Libvirt and QEMU/KVM
- ✓ Full Virtualization VMM Architecture
- ✓ Xen and Paravirtualization
- ✓ Xen Architecture

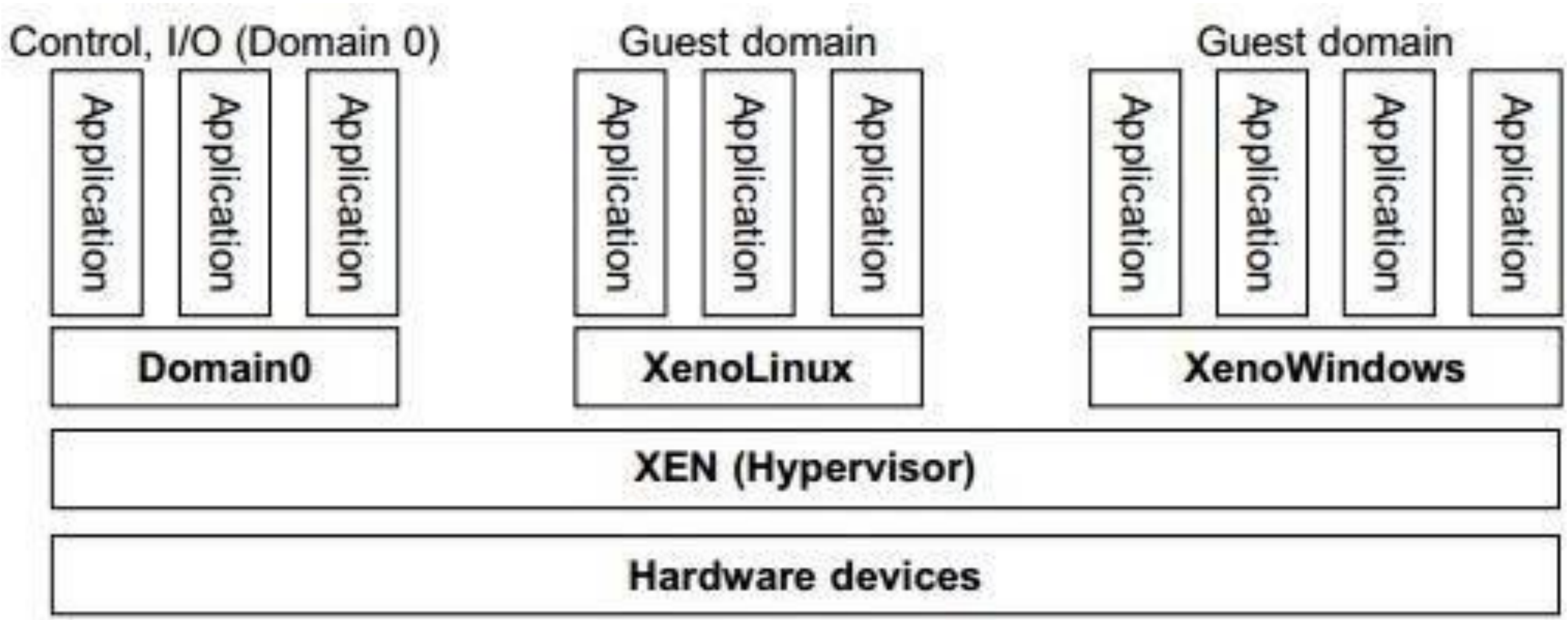
Agenda

- ✓ Memory Virtualization Problem
- ✓ Memory Reclamation Techniques
- ✓ Container
- ✓ Docker

Xen and Para Virtualization

- ✓ Para-virtualization is a **virtualization technique** where the guest OS is **aware that it is running in a virtualized environment**

Xen Architecture



Trap and Emulate Architecture

- ✓ The **Trap and Emulate** architecture is a fundamental technique used in **CPU virtualization**.
- ✓ It enables a **guest operating system (OS)** to run inside a **virtual machine (VM)** without modification, even if it executes privileged instructions.
- ✓ This method ensures that **guest OS operations that require direct hardware access** are intercepted (**trap**) by the hypervisor, which then simulates (**emulate**) their execution in a safe manner.

CPU Virtualization in Xen

- Guest OS code modified to not invoke any privileged instruction
  - Any privileged operation traps to Xen in ring 0
- Hypercalls: guest OS voluntarily invokes Xen to perform privileged ops
  - Much like system calls from user process to kernel
  - Synchronous: guest pauses while Xen services the hypercall
- Asynchronous event mechanism: communication from Xen to domain
  - Much like interrupts from hardware to kernel
  - Used to deliver hardware interrupts and other notifications to domain
  - Domain registers event handler callback functions

Trap Handling in Xen

- When trap/interrupt occurs, Xen copies the trap frame onto the guest OS kernel stack, invokes guest interrupt handler
- Guest registers an interrupt descriptor table with Xen to handle traps
  - Interrupt handlers validated by Xen (check that no privileged segments loaded)
- Guest trap handlers work off information on kernel stack, no modifications needed to guest OS code
  - Except page fault handler, which needs to read CR2 register to find faulting address (privileged operation)
  - Page fault handler modified to read faulting address from kernel stack (address placed on stack by Xen)
- What if interrupt handler still invokes privileged operations?
  - Traps to Xen again and Xen detects this “double fault” (trap followed by another trap from interrupt handler code) and terminates misbehaving guest

Memory Virtualization in Xen

- One copy of combined GVA→HPA page table maintained by guest OS
  - CR3 points to this page table
  - Like shadow page tables, but in guest memory, not in VMM
- Guest is given read-only access to guest "RAM" mappings (GPA→HPA)
  - Using this, guest can construct combined GVA→GPA mapping
- Guest page table is in guest memory, but validated by Xen
  - Guest marks its page table pages as read-only, cannot modify
  - When guest needs to update, it makes a hypercall to Xen to update page table
  - Xen validates updates (is guest accessing its slice of RAM?) and applies them
  - Batched updates for better performance
- Segment descriptor tables are also maintained similarly
  - Read-only copy in guest memory, updates validated and applied by Xen
  - Segments truncated to exclude top 64MB occupied by Xen

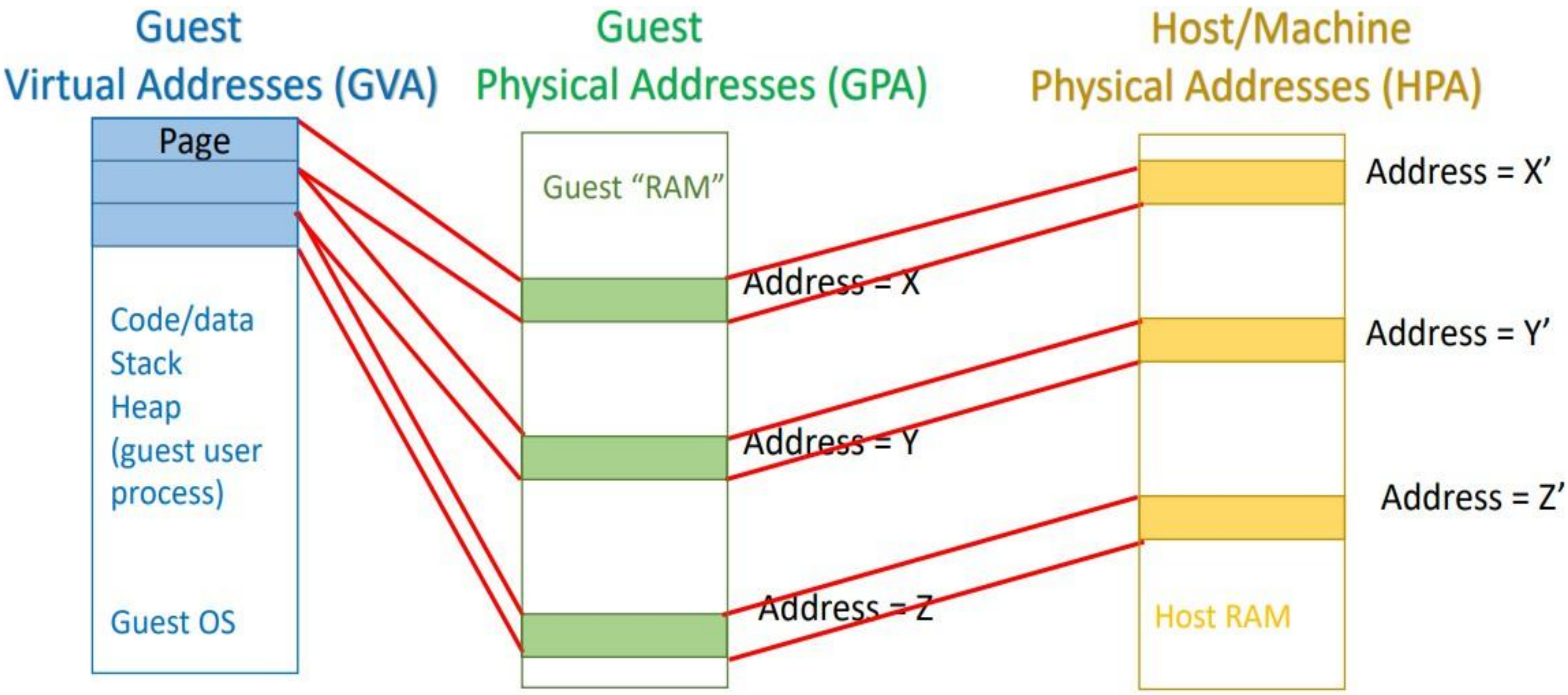
Memory Virtualization

- ✓ Memory virtualization is a critical component of system virtualization, allowing **multiple virtual machines (VMs) to share the physical memory of a host system.**
- ✓ However, it introduces several challenges related to **memory management, performance, and security.**

Virtual Memory Layers

- ✓ **Guest Virtual Address (GVA)** – Address used by the guest application.
- ✓ **Guest Physical Address (GPA)** – Address used by the guest OS.
- ✓ **Host Physical Address (HPA)** – Actual memory address on the host machine.

Virtual Memory Layers



Guest "RAM" is actually memory of the userspace hypervisor process running on the host, which is mapped to host memory by the host's page table

Importance of Memory Virtualization

- ✓ **Improves memory utilization.**
- ✓ **Enhances system security through isolation.**
- ✓ **Enables scalability in cloud computing.**
- ✓ **Supports multiple operating systems on a single machine.**

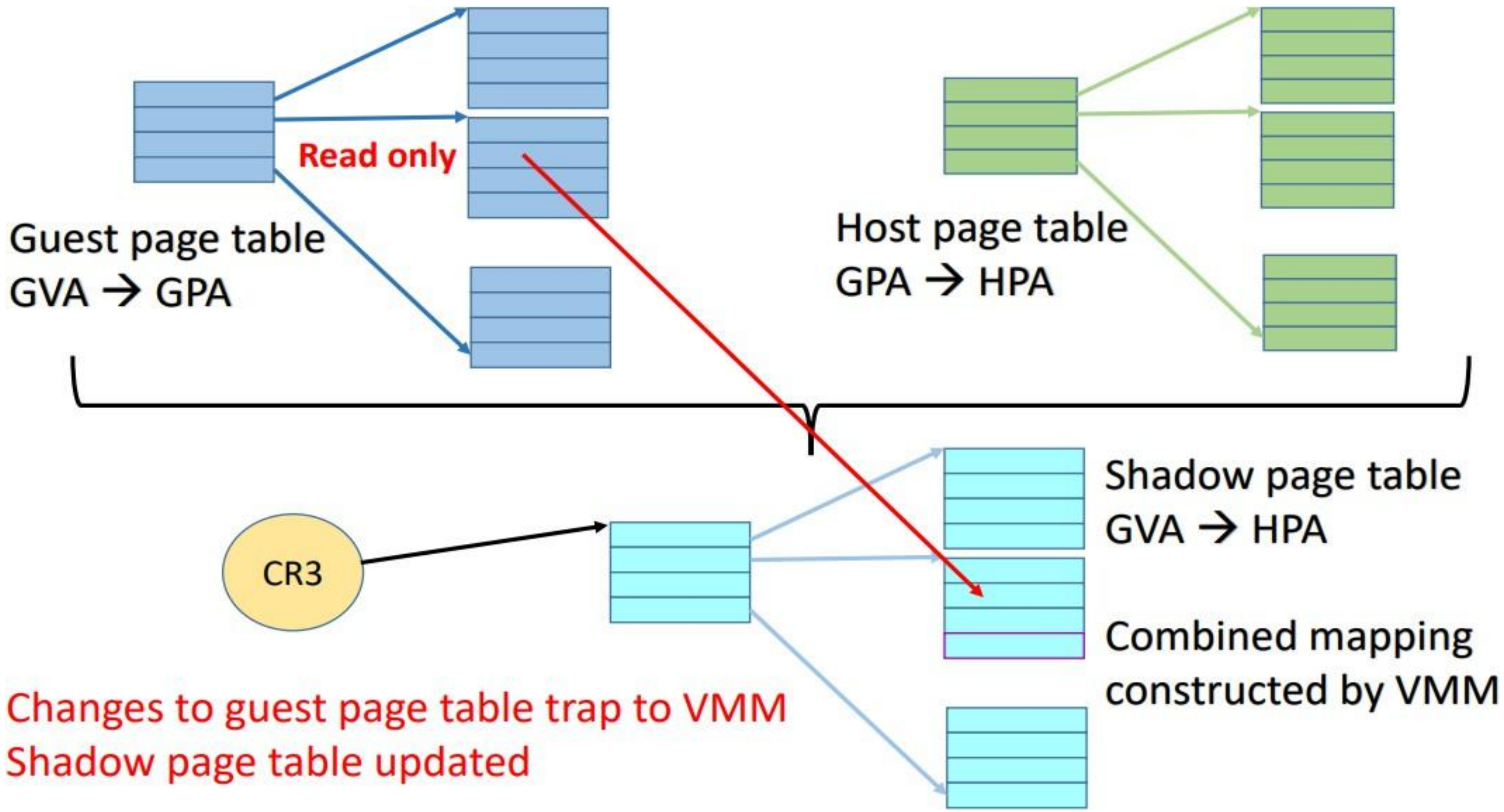
Techniques for Memory Virtualization

- Shadow Paging
- Extended Page Table (EPT)
- Demand Paging

Paging and Segmentation

- ✓ **Paging**: Divides **memory into fixed-size blocks** called pages, allowing efficient memory allocation.
- ✓ **Segmentation**: Divides **memory into variable-sized segments** based on program needs.

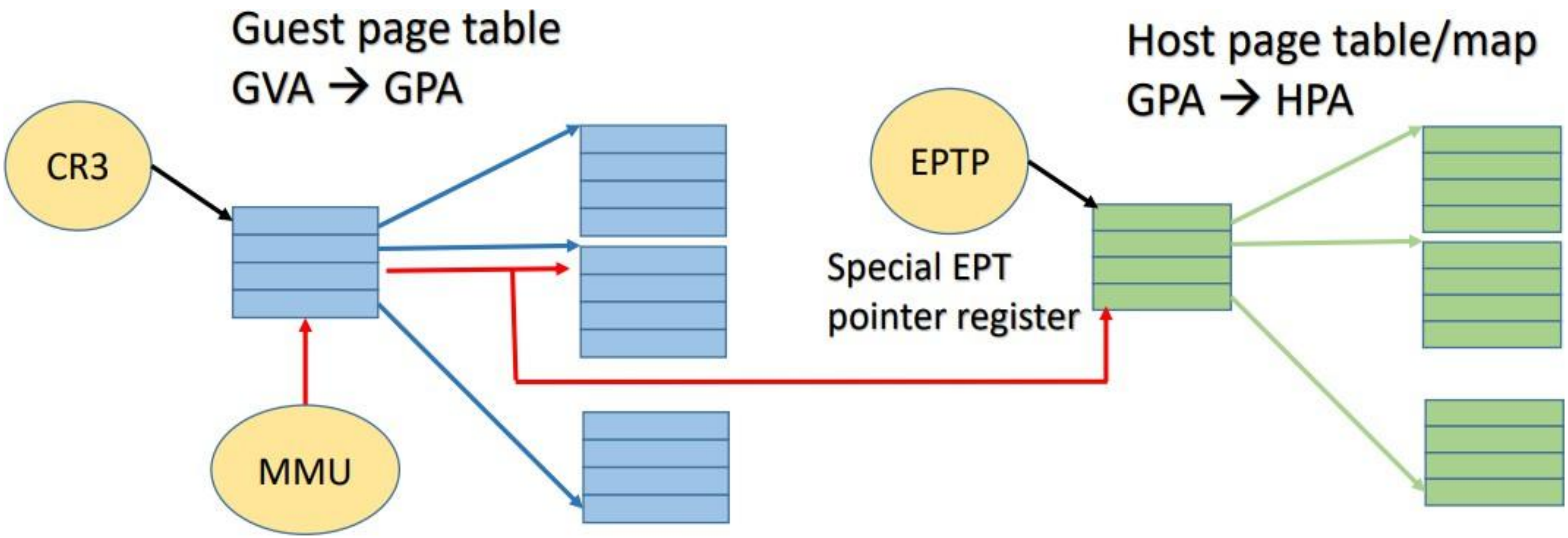
Shadow Paging



Maintaining Shadow Page Tables

- Guest writes to CR3, privileged operation traps to VMM
  - VMM marks the guest page table pages as read-only
  - VMM constructs shadow page table, sets CR3 to it
- Shadow page table can be built on demand
  - Start with empty page table, add entries on page faults
- Guest changes page table, traps to VMM, shadow entry updated
- Guest OS keeps multiple page tables of active processes in memory
  - On context switch, new page table used, but old page table still in memory
  - What about shadow page tables? How many in memory?
- Many design choices exist
  - VMM can discard old shadow page table on context switch, and rebuild it later (overhead during context switch)
  - VMM can maintain multiple shadow page tables of active processes (overhead to track changes to all page table pages)

Extended Page Table (EPT)

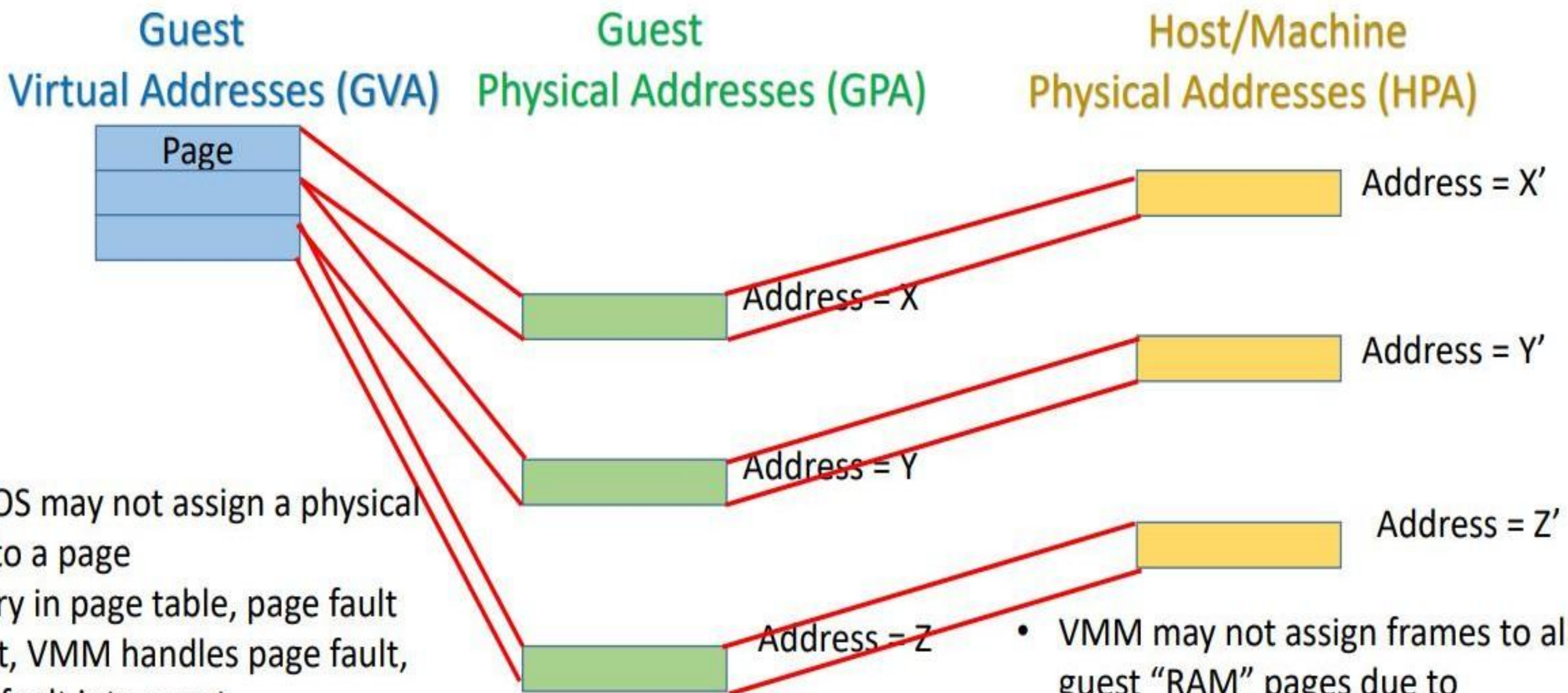


- Page table walk by MMU: Start walking guest page table using GVA
- Guest PTE (for every level page table walk) gives GPA (cannot use GPA to access memory)
- Use GPA, walk host page table to find HPA, then access memory page, then next level access
- Every step in guest page table walk requires walking N-level host page table
- N-level page tables in guest/host result in page table walk of NXN memory accesses

Demand Paging and Page Faults

- ✓ Demand paging is a technique **used in virtual memory systems where pages enter main memory only when requested or needed by the CPU**.
- ✓ The term **“page miss” or “page fault”** refers to a **situation where a referenced page is not found in the main memory**.

Demand Paging and Page Faults



- Guest OS may not assign a physical frame to a page
- No entry in page table, page fault
- VM exit, VMM handles page fault, injects fault into guest
- Guest assigns physical frame

- VMM may not assign frames to all guest "RAM" pages due to memory pressure
- Page fault is handled by VMM
- Guest OS not aware of this fault
- "Hidden" page faults

Memory reclamation Technique

- ✓ Memory reclamation is a crucial technique in operating systems and virtualization platforms to **optimize memory usage** by reclaiming unused or underutilized memory.
- ✓ It ensures efficient memory allocation, preventing **memory leaks, fragmentation, and performance degradation** in multi-process or multi-tenant environments like cloud computing.

Memory reclamation Technique

Ballooning

Transparent Page Sharing (TPS)

Kernal Same-page Merging (KSM)

Ballooning

- ✓ A hypervisor-controlled technique where **unused memory is reclaimed from idle VMs and reallocated** to VMs experiencing memory pressure.
- ✓ A **balloon driver** installed in the **guest OS** inflates to claim memory, forcing the **OS to release it back to the hypervisor.**
- ✓ **Example: VMware ESXi and KVM** use ballooning to optimize cloud-based VM memory allocation.

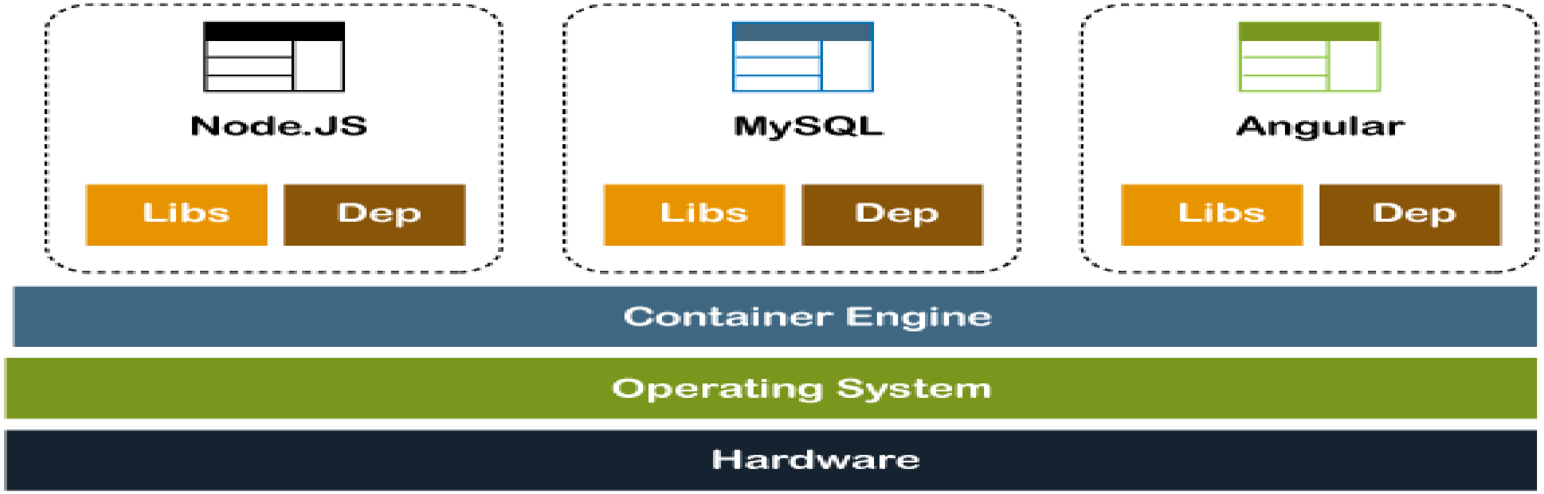
Transparent Page Sharing (TPS)

- ✓ **Identical memory pages across multiple VMs are merged into a single shared copy**.
- ✓ Reduces **redundant memory allocation, improving efficiency.**
- ✓ **Example: VMware ESXi and KVM** hypervisors use TPS to improve memory efficiency in cloud environments.

Kernal Same- page Merging (KSM)

- ✓ Works similarly to TPS but at the **kernel level** in Linux-based hypervisors.
- ✓ Merges identical pages across **processes and VMs** to optimize memory usage.
- ✓ Example: **KVM (Kernel-based Virtual Machine)** uses KSM to optimize VM workloads in **OpenStack cloud platforms.**

Container


Node.JS
MySQL
Angular
Libs
Dep
Libs
Dep
Libs
Dep
Container Engine
Operating System
Hardware

Container

- ✓ Modern software applications need to be **scalable, portable, and efficient** across different environments (development, testing, and production).
- ✓ **Containers** have emerged as a **powerful technology** to achieve these goals by **packaging applications with their dependencies into lightweight, isolated environments.**

Container

✓ **Why do we need containers when we already have virtual machines (VMs)?**

Inefficiency of Virtual Machines (VMs)

- ✓ VMs require a full operating system (OS) instance for each application, consuming significant system resources **(RAM, CPU, storage).**
- ✓ **Slow boot-up time** due to OS initialization.
- ✓ High overhead due to **hypervisor dependency.**

Containers share the same OS kernel, eliminating the need for multiple full OS instances, reducing overhead and improving performance.

Portability Issues in Traditional Deployment

- ✓ Applications behave **differently across environments (development, testing, and production).**
- ✓ **Dependency conflicts** occur due to mismatched libraries and software versions.

Containers encapsulate application code, dependencies, and configurations, ensuring consistency across different platforms (Windows, Linux, Cloud, etc.).

Multi-Cloud and Hybrid Cloud Compatibility

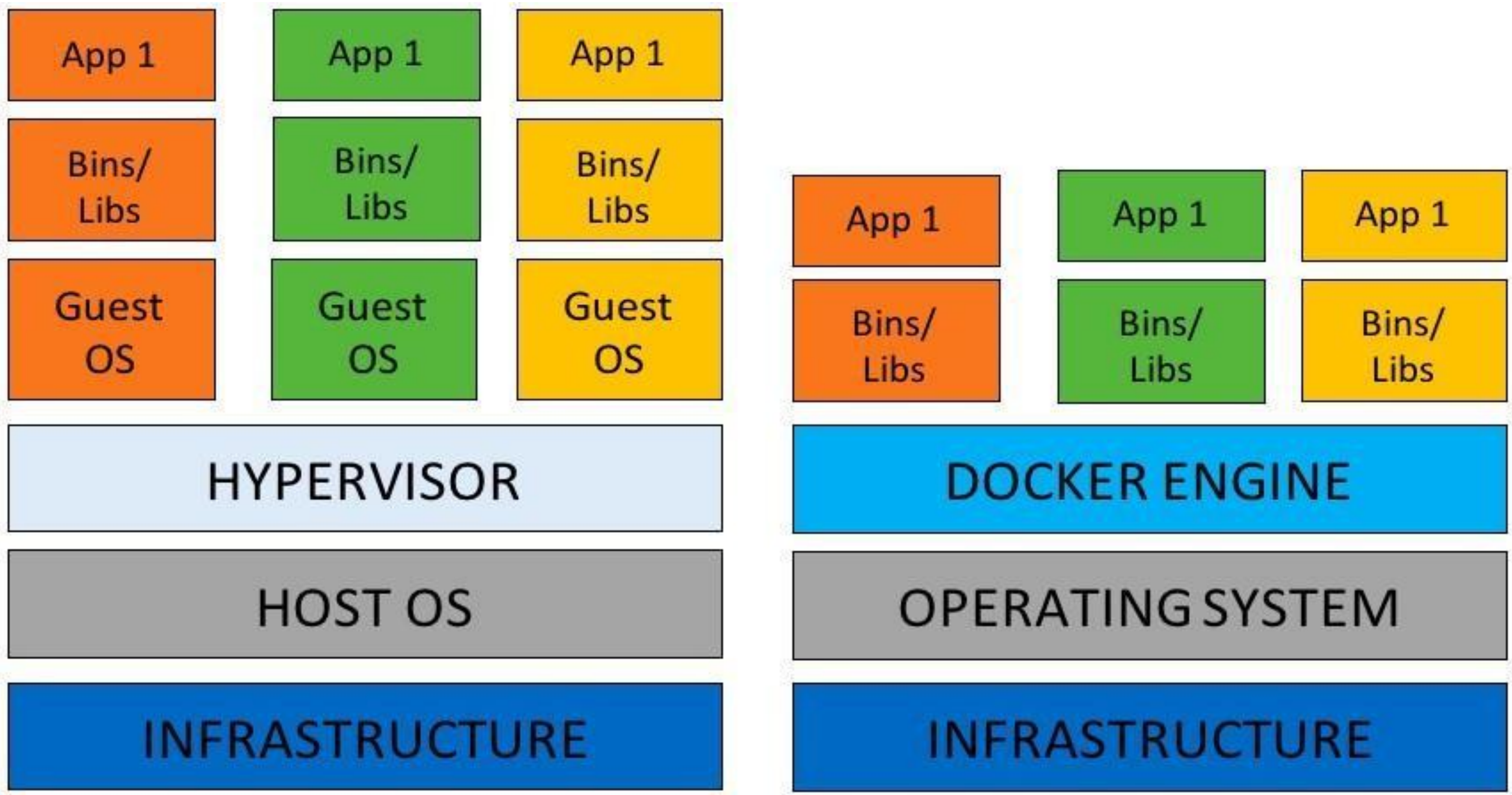
- ✓ Applications designed for one **cloud provider (AWS, Azure, Google Cloud)** may not work efficiently on another due to **dependency issues.**

Containers abstract the application from the underlying infrastructure, making them portable across different cloud providers.

Virtual Machine Vs Container

Feature	Containers	Virtual Machines (VMs)
Isolation	**Process-level**	**Full OS-level**
Startup Time	**Milliseconds**	**Minutes**
Resource Consumption	**Low** (Shares OS kernel)	**High** (Separate OS per VM)
Portability	**High** (Works across platforms)	**Limited**
Deployment Speed	**Fast**	**Slow**
Use Case	**Microservices, Cloud-native apps**	**Traditional monolithic applications**

Containers



App 1
Bins/ Libs
Guest OS
App 1
Bins/ Libs
Guest OS
App 1
Bins/ Libs
Guest OS
HYPERVISOR
HOST OS
INFRASTRUCTURE
App 1
Bins/ Libs
App 1
Bins/ Libs
App 1
Bins/ Libs
DOCKER ENGINE
OPERATING SYSTEM
INFRASTRUCTURE

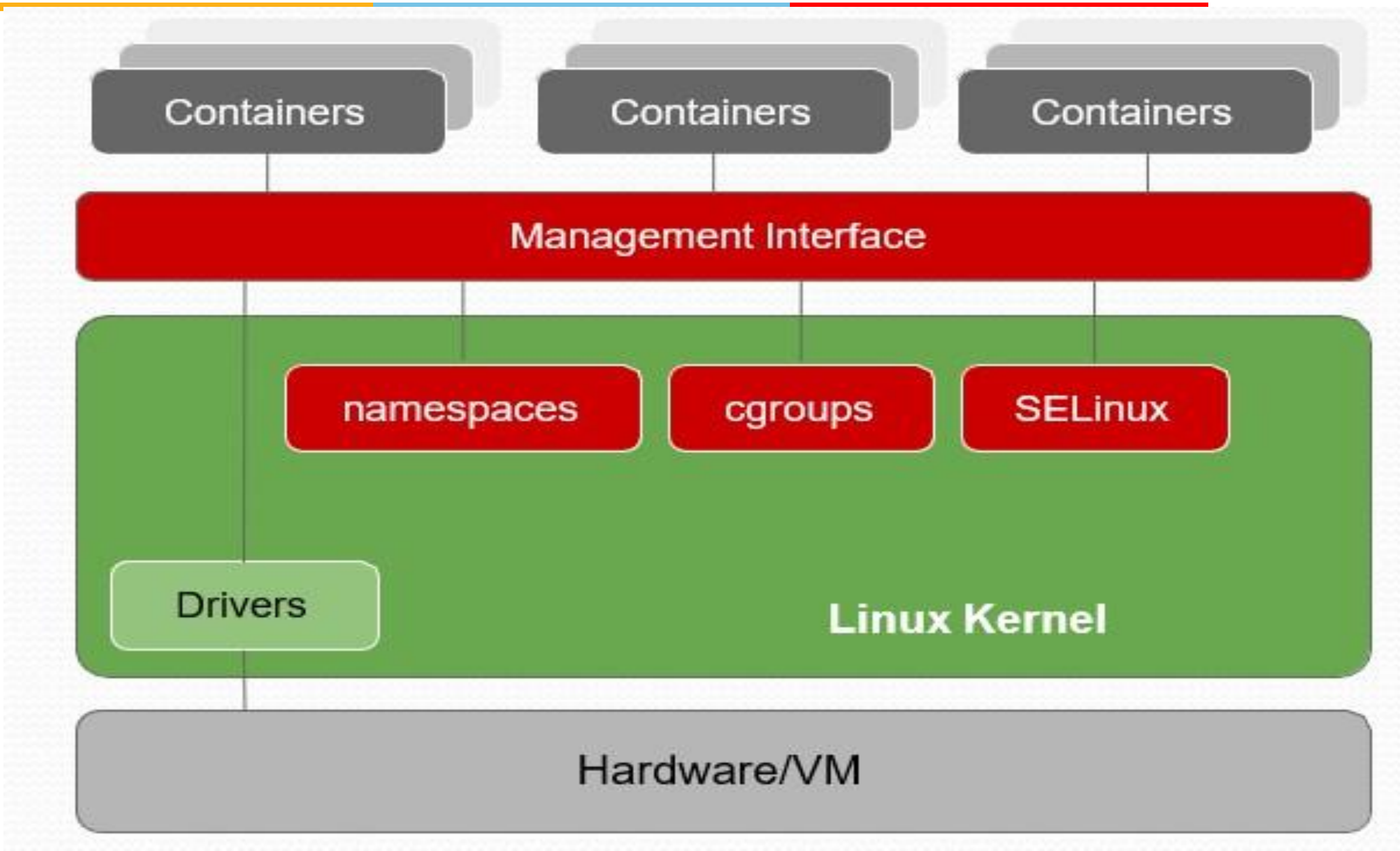
Containers

- ✓ A **container** is an **isolated, executable unit** that includes an **application and all its dependencies (libraries, configuration files, etc.)**

Key Characteristics of Containers

- ✓ **Lightweight** → Shares **OS kernel, avoiding the overhead of a full OS.**
- ✓ **Isolated** → **Runs independently** without interfering with other applications.
- ✓ **Portable** → Works **across different operating systems and cloud platforms.**
- ✓ **Efficient** → **Consumes fewer system resources** compared to VMs.

Linux Containers



Linux Containers LXC

- ✓ LXC is an abbreviation used for **Linux Containers** which is an operating system that is used **for running multiple Linux systems virtually** on a controlled host via a **single Linux kernel**.
- ✓ LXC bundles with the kernel's **Cgroups** to provide the functionality for the **process and network space** instead of creating a full virtual machine and provides an **isolated environment for the applications.**

Features of LXC

- ✓ It provides **Kernel namespaces such as IPC, mount, PID, network, and user.**
- ✓ It provides **Kernel capabilities**.
- ✓ **Control groups (Cgroups)**
- ✓ **Seccomp profiles**

LXC is not new

- ✓ **Lightweight virtualization.**
- ✓ **OS-level virtualization**
- ✓ Allow single host to **operate multiple isolated & resource-controlled**
- ✓ **Linux Instances** included in the **Linux kernel called LXC (Linux Container)**

Thank You