CLOUD COMPUTING

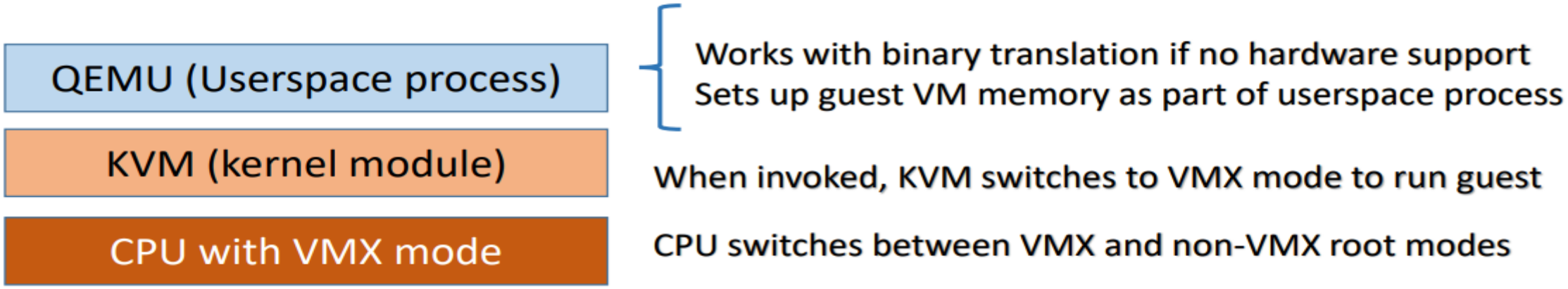
Session 5

Dr. S. Prabakeran
Guest Faculty

BITS Pilani
Pilani Campus

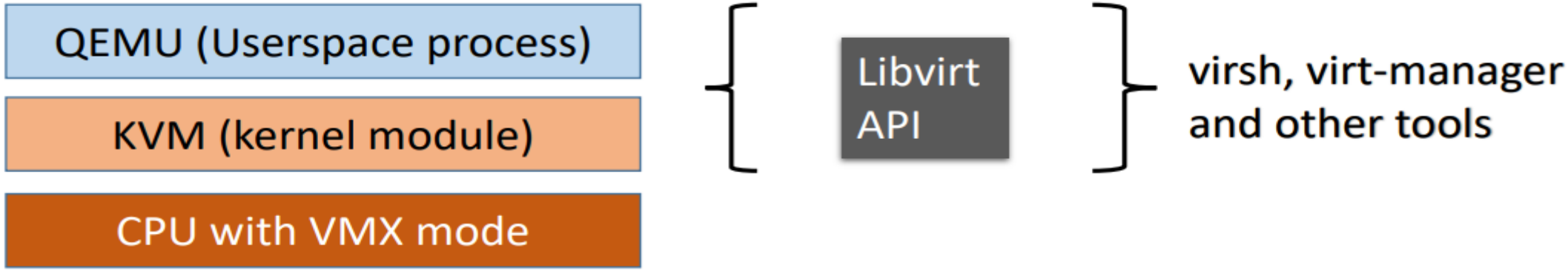
Hardware-assisted Virtualization

- Modern technique, after hardware support for virtualization introduced in CPUs
  - Original x86 CPUs did not support virtualization
  - Intel VT-X or AMD-V support is widely available in modern systems
  - Special CPU mode of operation called VMX mode for running VMs
- Many hypervisors use this H/W feature, e.g., QEMU/KVM in Linux



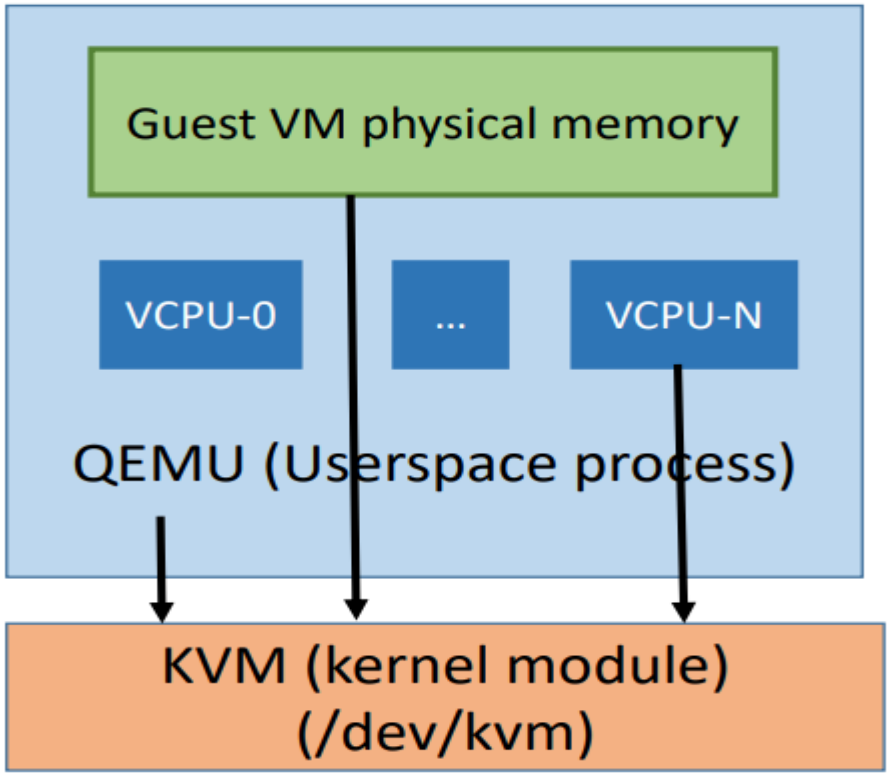
Libvirt and QEMU/KVM

- When you install QEMU/KVM on Linux, libvirt is also installed
  - A set of tools manage hypervisors, including QEMU/KVM
  - A daemon runs on the system and communicates with hypervisors
  - Exposes an API using which hypervisors can be managed, VM created etc.
  - Commandline tool (virsh) and GUI (virt-manager) use this API to manage VMs



QEMU architecture

- QEMU is userspace process
- KVM exposes a dummy device
  - QEMU talks to KVM via open/ioctl syscalls
- Allocates memory via mmap for guest VM physical memory
- Creates one thread for each virtual CPU (VCPU) in guest
- Multiple file descriptors to /dev/kvm (one for QEMU, one for VM, one for VCPU and so on)
  - ioctl on fds to talk to KVM
- Host OS sees QEMU as a regular multi-threaded process



QEMU operation

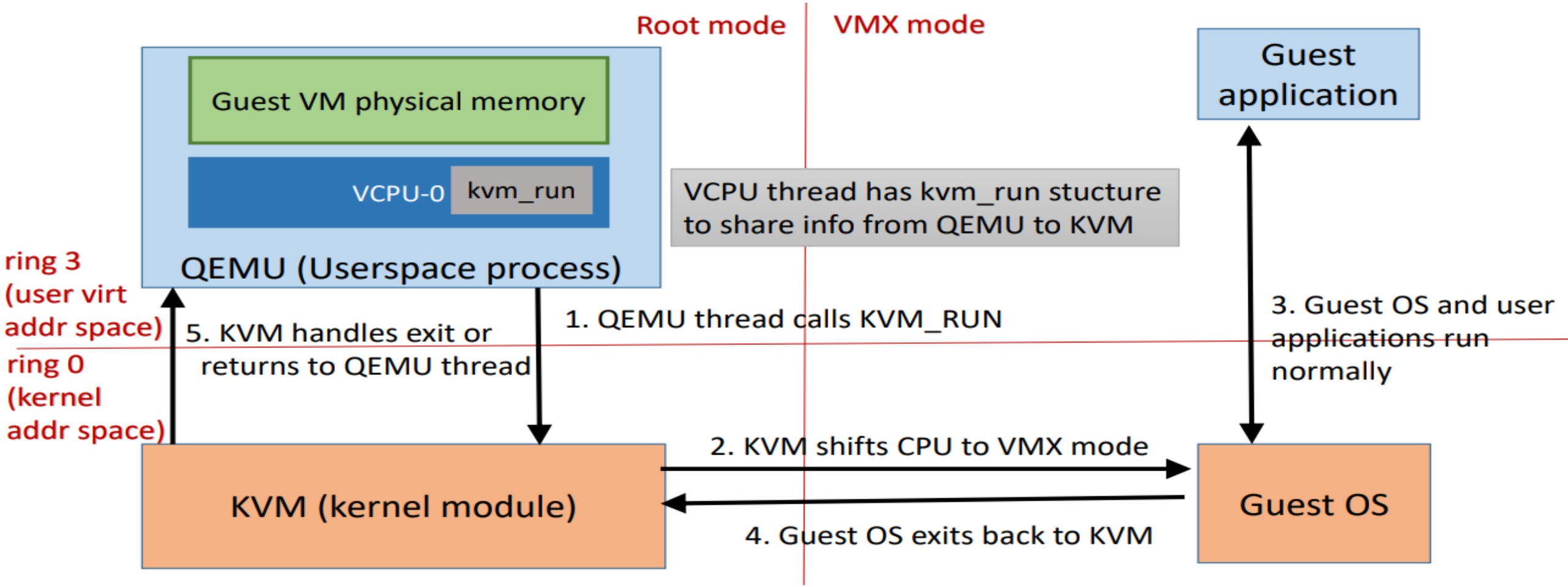
```
open(/dev/kvm)
ioctl(qemu_fd, KVM_CREATE_VM)
ioctl(vm_fd, KVM_CREATE_VCPU)

for(;;) { //each VCPU runs this loop
  ioctl(vcpu_fd, KVM_RUN)
  switch(exit_reason) {
    case KVM_EXIT_IO: //do I/O
    case KVM_EXIT_HLT:
  }
}
```

This ioctl system call blocks this thread, KVM switches to VMX mode, runs guest VM

Returns to QEMU on host when VM exits from VMX mode.

QEMU handles exit and returns to guest VM

QEMU/KVM operation

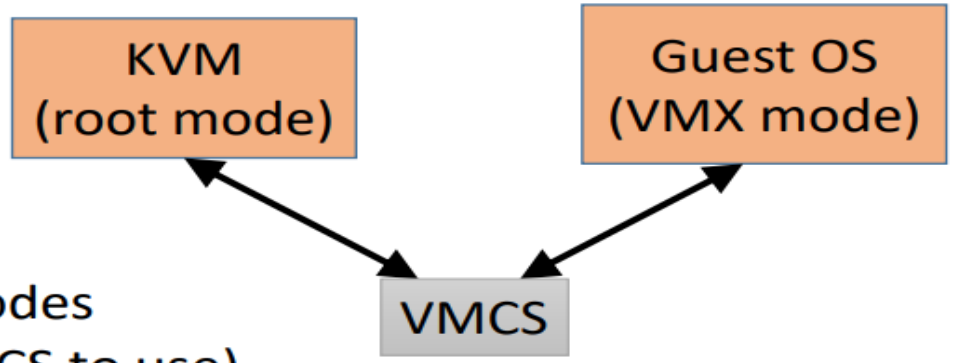


VMX mode

- Special CPU instructions to enter and exit VMX mode
  - VMLAUNCH, VMRESUME invoked by KVM to enter VMX mode
  - VMEXIT invoked by guest OS to exit VMX mode
- On VMX entry/exit instructions, CPU switches context between host OS to guest OS
  - Page tables (address space), CPU register values etc switched
  - Hardware manages the mode switch
- Where is CPU context stored during mode switch?
  - Cannot be stored in host OS or guest OS data structures alone (why?)
  - VMCS (VM control structure), also called VMCB (VM control block)

VM control structure (VMCS)

VM control structure (VMCS)

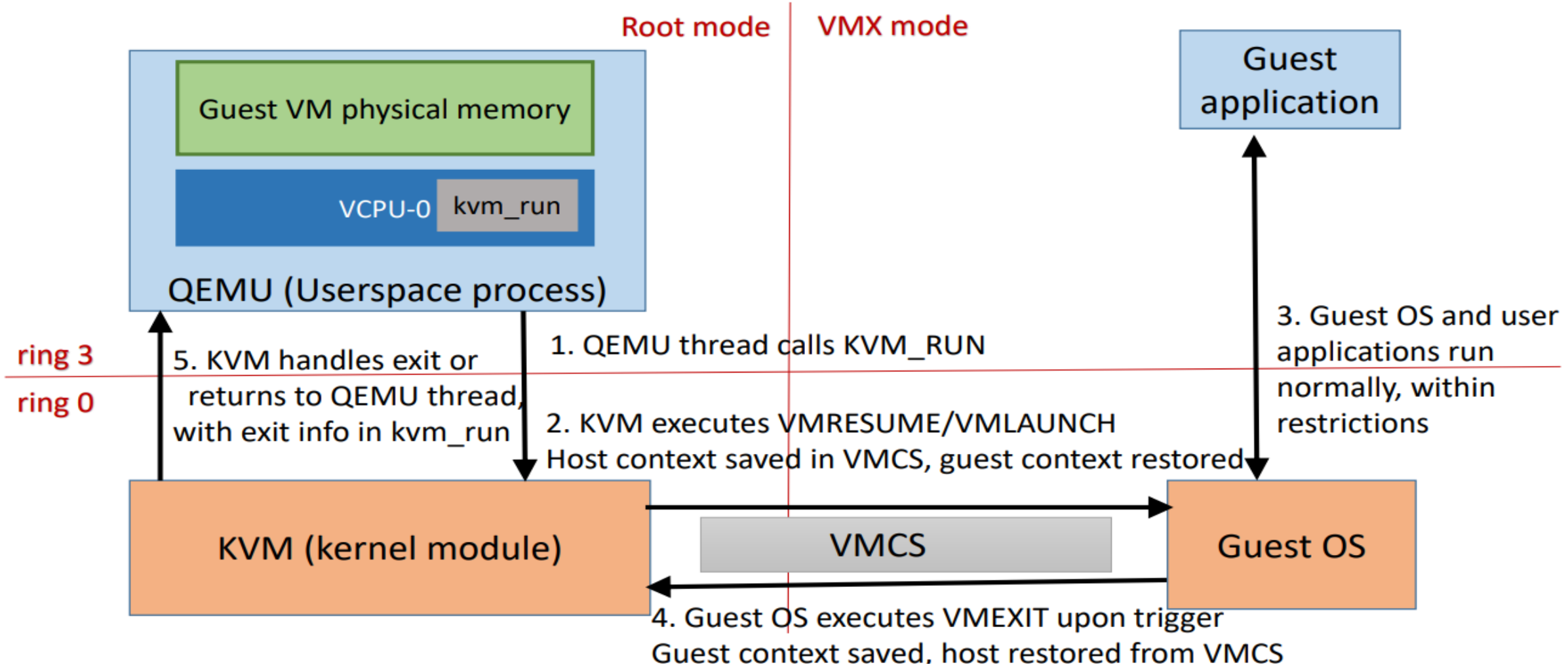


- What is VMCS?
  - Common memory area accessible in both modes
  - One VMCS per VM (KVM tells CPU which VMCS to use)
- What is stored in VMCS?
  - Host CPU context: Stored when launching VM, restored on VM exit
  - Guest CPU context: Stored on VM exit, restored when VM is run
  - Guest entry/execution/exit control area: KVM can configure guest memory and CPU context, which instructions and events should cause VM to exit
  - Exit information: Exit reason and any other exit-related information
- VMCS information (e.g., exit reason) exchanged with QEMU via kvm_run structure
  - VMCS only accessible to KVM in kernel mode, not to QEMU userspace

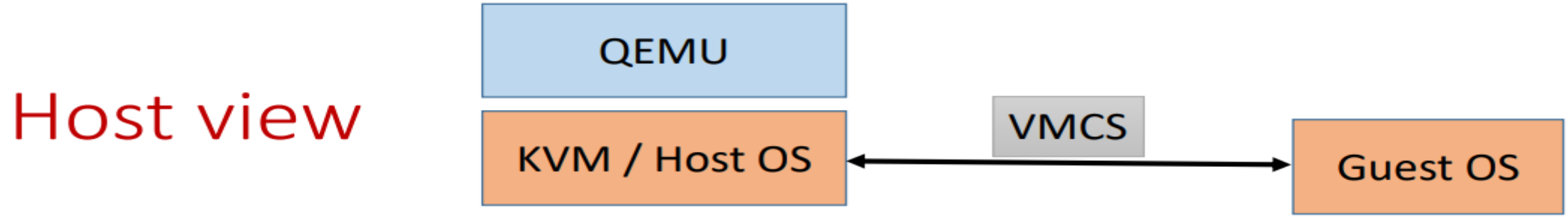
VMX mode execution

- How is guest OS execution in VMX mode different?
- Restrictions on guest OS execution, configurable exits to KVM
  - Guest OS exits to KVM on certain instructions (e.g., I/O device access)
- No hardware access to guest, emulated by KVM
  - Guest OS usually exits on interrupts (interrupts handled by KVM, assigned to the appropriate host or guest OS)
  - KVM can inject virtual interrupts to guest OS during VMX mode entry
- All of the above controlled by KVM via VMCS
- Mimics the trap-and-emulate architecture with hardware support
  - Guest runs in a (special) ring 0, but trap-and-emulate achieved

QEMU/KVM operation revisited



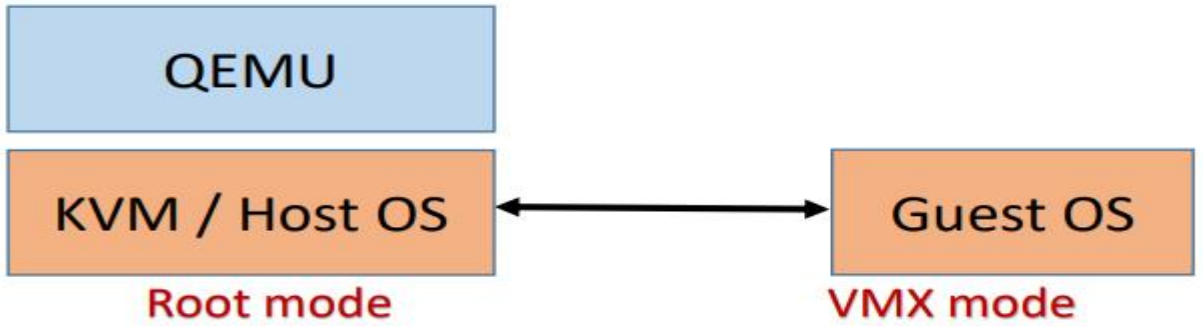
Host view



- Host sees QEMU as regular multithreaded process
  - Process that has memory-mapped memory, talks to KVM device via ioctl calls
  - Multiple QEMU VCPU threads can be scheduled in parallel on multiple cores
- When KVM launches a VM, host OS context is stored in VMCS
  - Host OS execution is suspended (all host processes stop)
  - CPU loads guest OS context and guest OS starts running
- When guest OS exits, host OS context is restored from VMCS
  - Host OS resumes in KVM, where it stopped execution
  - KVM can return to QEMU, or host can switch to another process
  - Host OS is not aware of guest OS execution

Summary

Summary

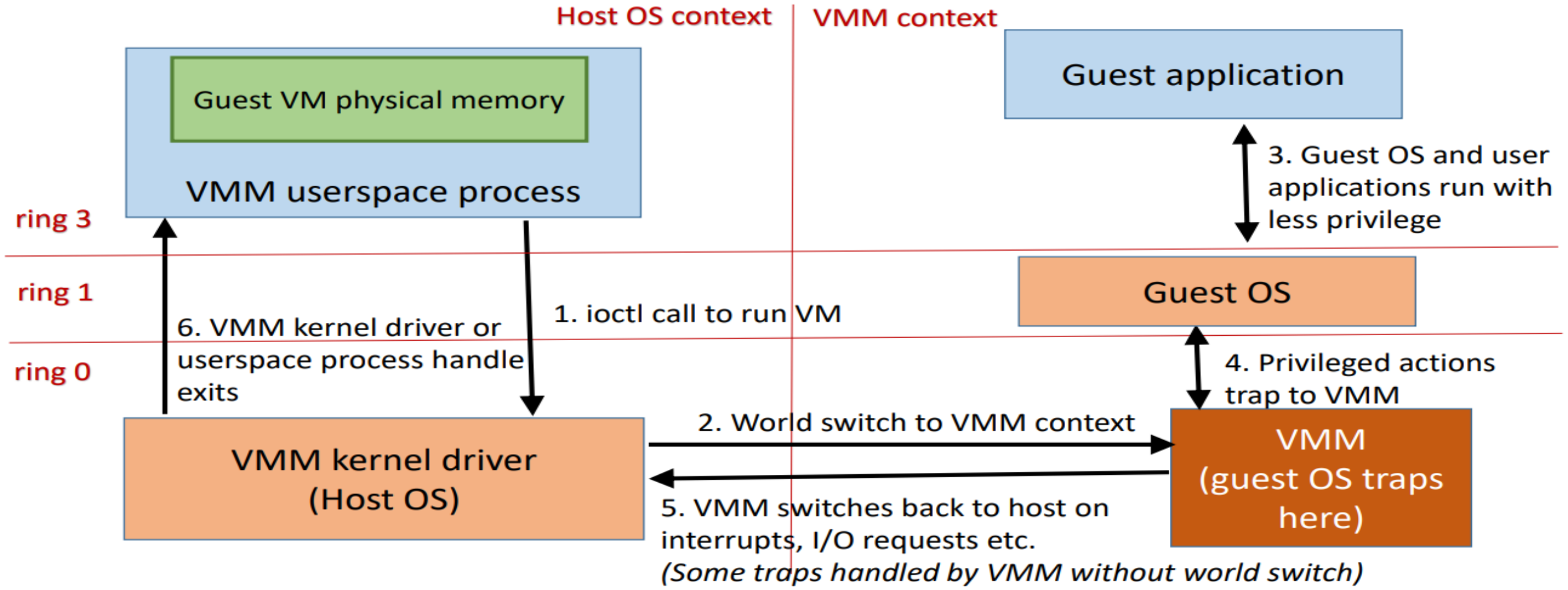


- Hardware-assisted CPU virtualization in QEMU/KVM
  - QEMU creates guest physical memory, one thread per VPCU
  - QEMU VCPU thread gives KVM_RUN command to KVM kernel module
  - KVM configures VM information in VMCS, launches guest OS in VMX mode
  - Guest OS runs natively on CPU until VM exit happens
  - Control returns to KVM/Host OS on VM exit
  - VM exits handled by KVM or QEMU
  - Host schedules QEMU like any other process, not aware of guest OS

Full virtualization

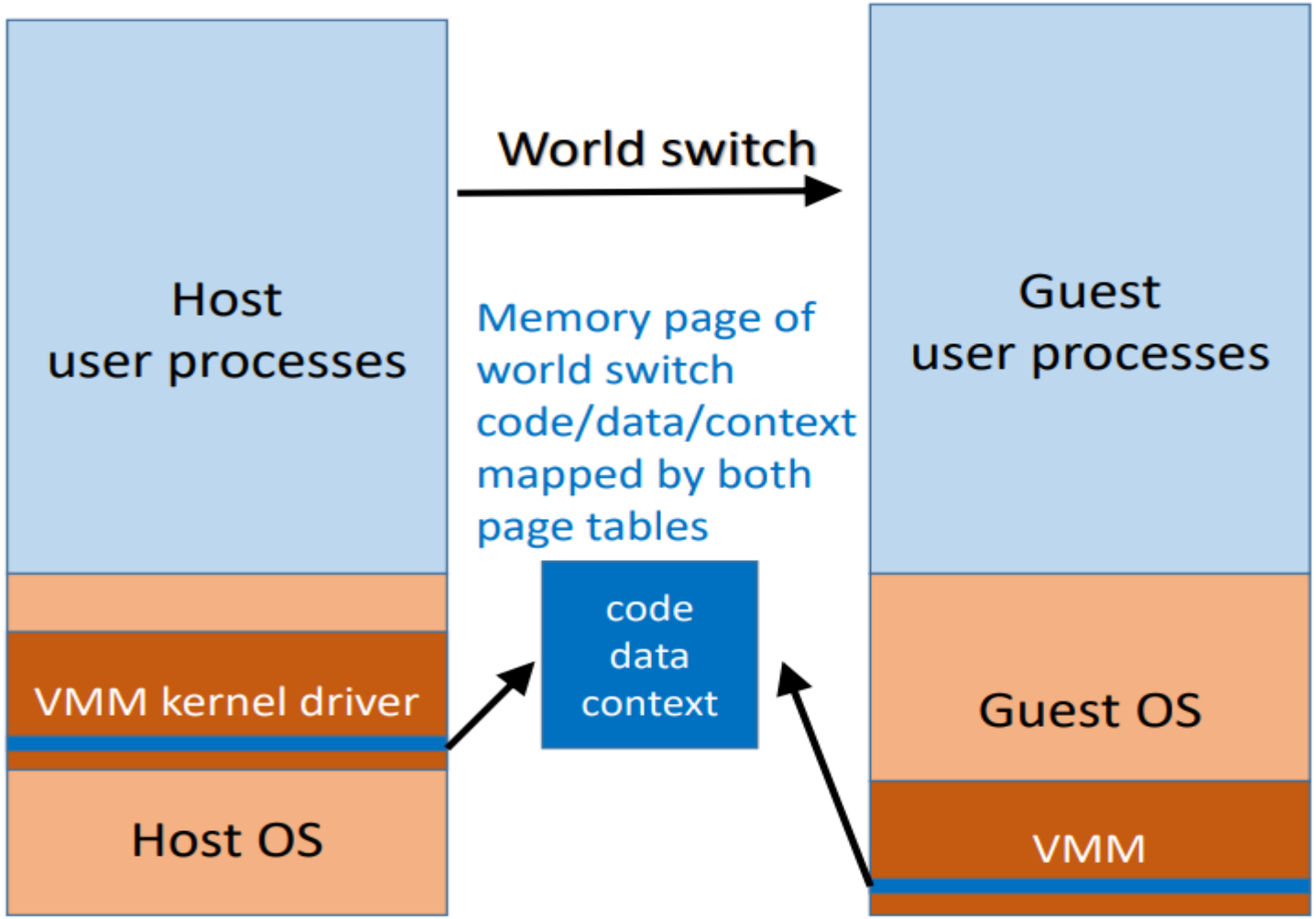
- x86 and other hardware lacked virtualization support
  - But cloud computing increased demand for virtualization
- VMWare workstation first to solve the problem of virtualization existing operating systems on x86 (basis for this lecture)
  - Type 2 hypervisor based on trap-and-emulate approach
- Key idea: dynamic (on a need basis) binary (not source) translation of OS instructions
  - Problematic OS instructions translated before execution
- Subsequently, hardware support for virtualization (previous lecture)
  - Binary translation is higher overhead than hardware-assisted virtualization
  - Used when hardware support not available

Full virtualization VMM architecture



Host and VMM contexts

- Each context has separate page tables, CPU registers, IDTs and so on
- VMM context: VMM occupies top 4MB of address space
- Memory page containing code/data of world switch mapped in both contexts
  - Host/VMM context saved/restored in this special “cross” page by VMM

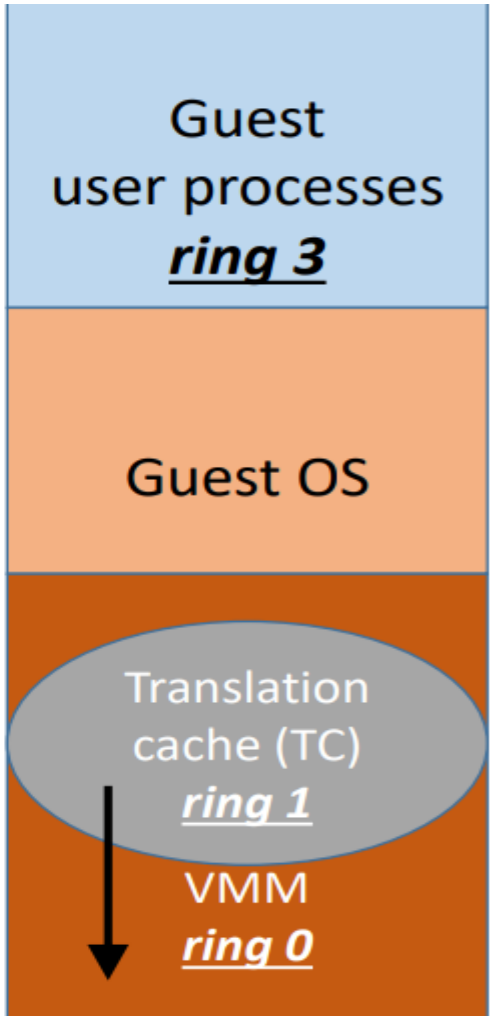


Understand difference with QEMU/KVM

- Where is context saved?
  - Common cross page mapped into both host and guest address spaces
  - KVM: Common memory (VMCS) accessible by CPU in both contexts via special instructions
- Privilege level of guest OS?
  - Guest OS runs in ring 1 (lower privilege). Instructions that do not run correctly at lower privilege level are suitably translated to trap to VMM
  - KVM: Guest OS runs in VMX ring 0. Some privileged instructions trigger exit to KVM
- How to trap to VMM?
  - VMM is located in top 4MB of guest address space , guest OS traps to VMM for privileged ops. World switch to host if VMM cannot handle trap in guest context
  - KVM: VMM is not in guest context, guest traps to VMM in host via VM exit

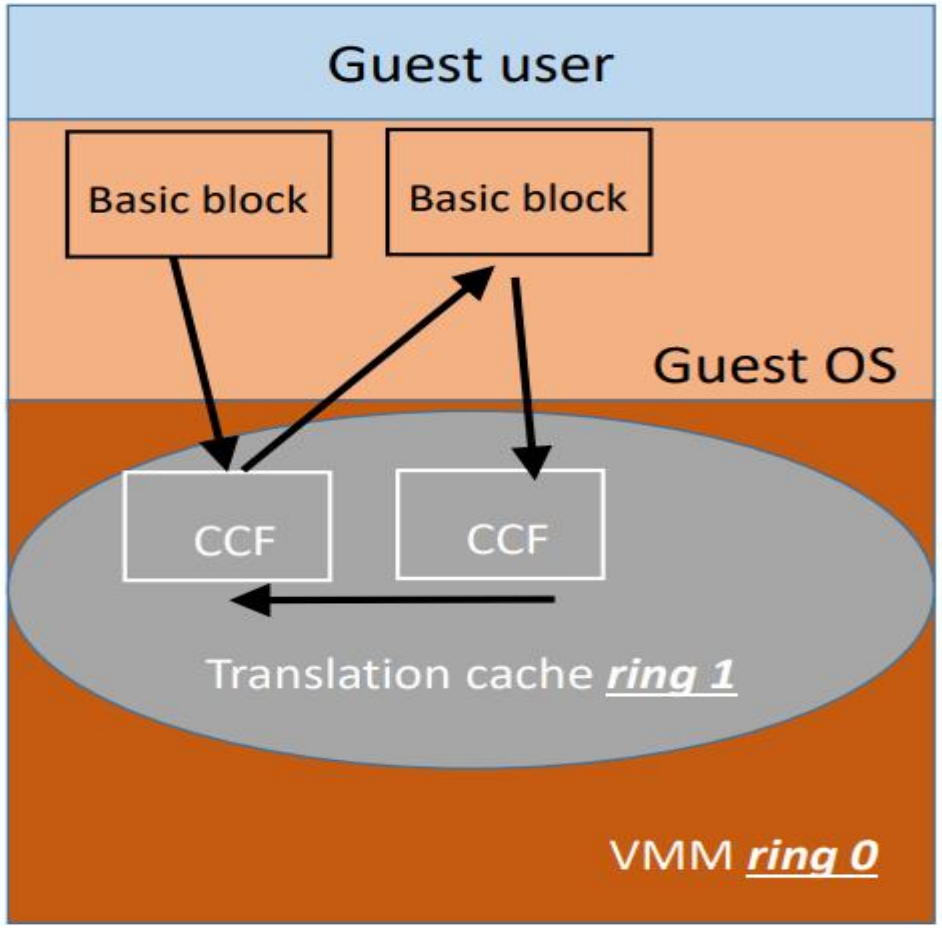
Binary translation

- Guest OS binary is translated instruction-by-instruction and stored in translation cache (TC)
  - Part of VMM memory
  - Most code stays same, unmodified
  - OS code modified to work correctly in ring 1
  - Sensitive but unprivileged instructions modified to trap
- Guest OS code executes from TC in ring 1
- Privileged OS code traps to VMM
  - E.g., I/O, set IDT, set CR3, other privileged ops
  - Emulated in VMM context or by switching to host
  - VMM sets sensitive data structures like IDT etc. (maintains shadow copies)



Dynamic binary translation

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



Summary

- VMWare workstation is example of full virtualization, where unmodified OS is run on x86 hardware via dynamic binary translation
  - VMM user process and kernel driver on host trigger world switch from host OS context to VMM context
  - World switch code/data is part of both host and VMM contexts, special cross page accessible in both modes has saved contexts
  - VMM is in top 4MB of address space in VMM context
  - Translated guest code runs in ring 1, traps to VMM in ring 0 for privileged operations (trap-and-emulate)
  - Traps handled by VMM in ring 0, or VMM exits to host OS for emulation
  - Segmentation used to protect VMM from guest OS

Xen and Paravirtualization

- Xen: most popular example of paravirtualized VMM
- Paravirtualization: modify guest OS to be amenable to virtualization
  - XenoLinux is a modified Linux OS that runs on Xen hypervisor
  - Application interface need not change
- Benefits: better performance than binary translation
- Disadvantages: requires source code changes to OS, porting effort
  - 1-2% code changes reported in the original Xen paper

Xen Architecture

- Type 1 hypervisor: runs directly over hardware
- Trap-and-emulate architecture
  - Xen runs in ring 0, guest OS in ring 1
  - Xen sits in the top 64MB of address space of guests
  - Guest OS traps to Xen to perform privileged actions
- A guest VM is called a domain
  - Special domain called dom0 runs control/management software

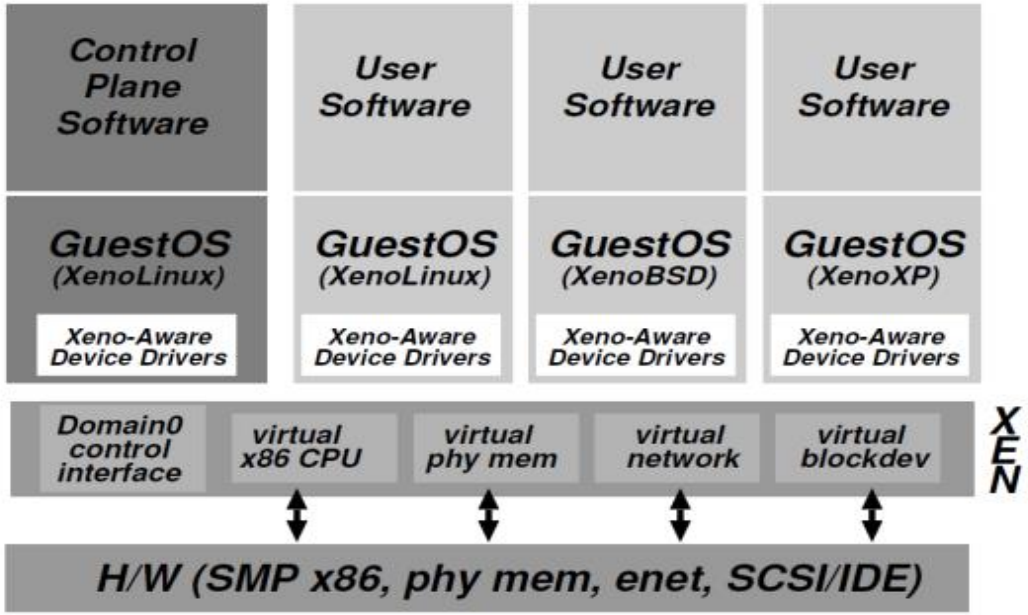


Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including *Domain0* running control software in a XenoLinux environment.

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

I/O virtualization in Xen

- Shared memory “rings” between guest domain and Xen/domain0
  - Front-end device driver in guest domain and backend in dom0
  - I/O requests placed in shared queue by guest domain
  - Request handled by Xen/domain0, responses placed in ring
  - Descriptors in queue: pointers to request data (DMA buffers with data for writes, empty DMA buffers for reads, etc.)
- Similar design to virtio
  - No copying of request data
  - Memory pages swapped between domains to exchange requests/responses
  - Batching for high performance

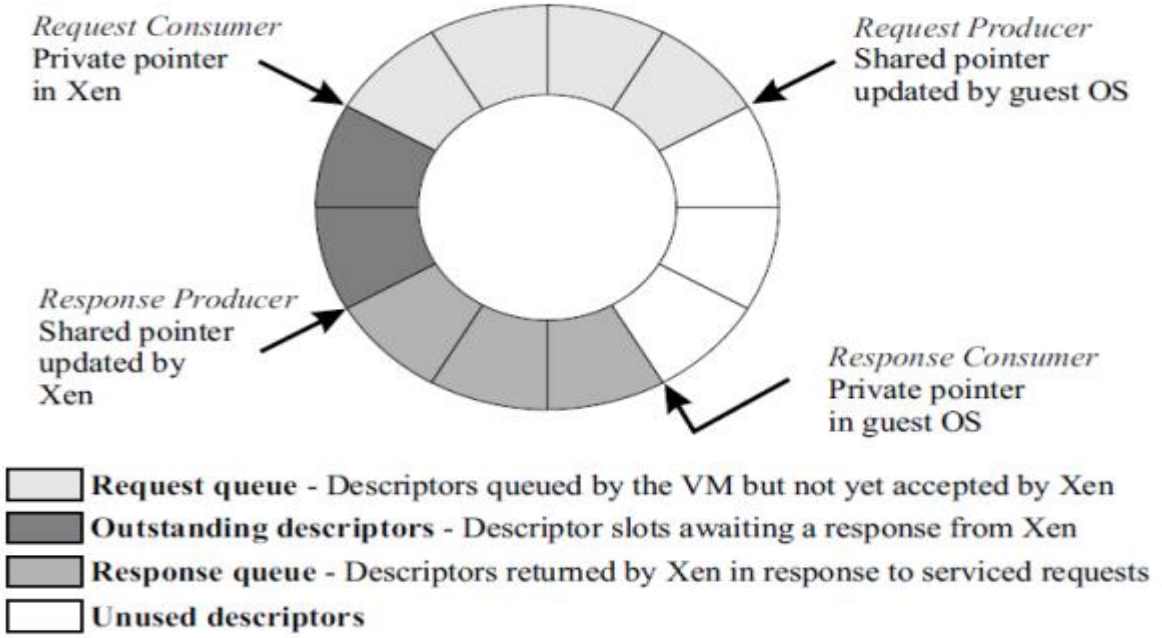
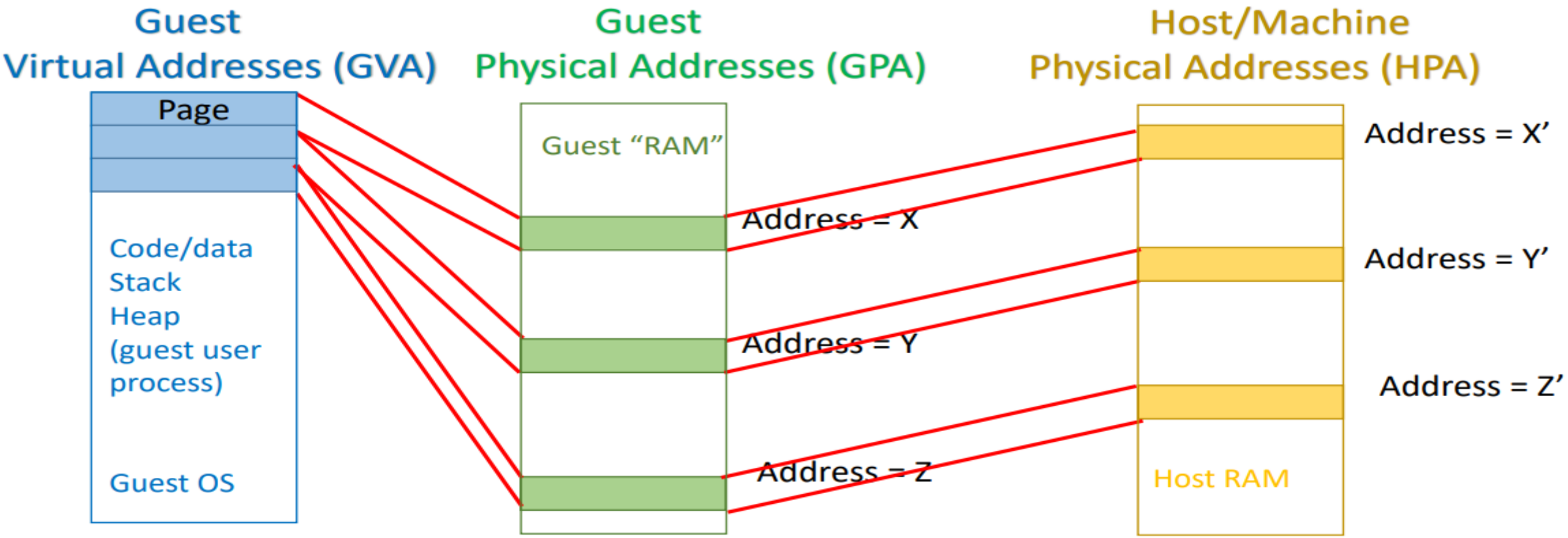


Figure 2: The structure of asynchronous I/O rings, which are used for data transfer between Xen and guest OSes.

Summary

- Xen: paravirtualization based hypervisor
- Guest OS modified to suit virtualization
- Trap-and-emulate via VMM
  - Guest in ring 1, VMM in ring 0
  - Guest traps to VMM for privileged operations
- Combined GVA→HPA page tables in guest memory
  - Read-only copy in guest
  - Updated via hypercalls to Xen
- I/O via shared rings between guest and Xen/domain0

Memory virtualization problem

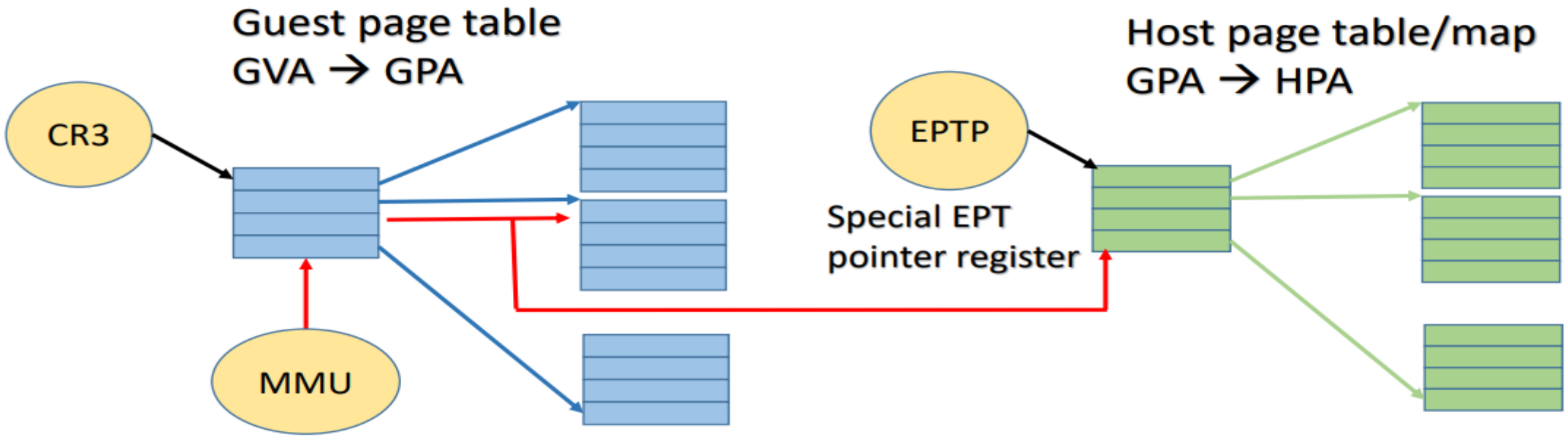


Guest "RAM" is actually memory of the userspace hypervisor process running on the host, which is mapped to host memory by the host's page table

Techniques for memory virtualization

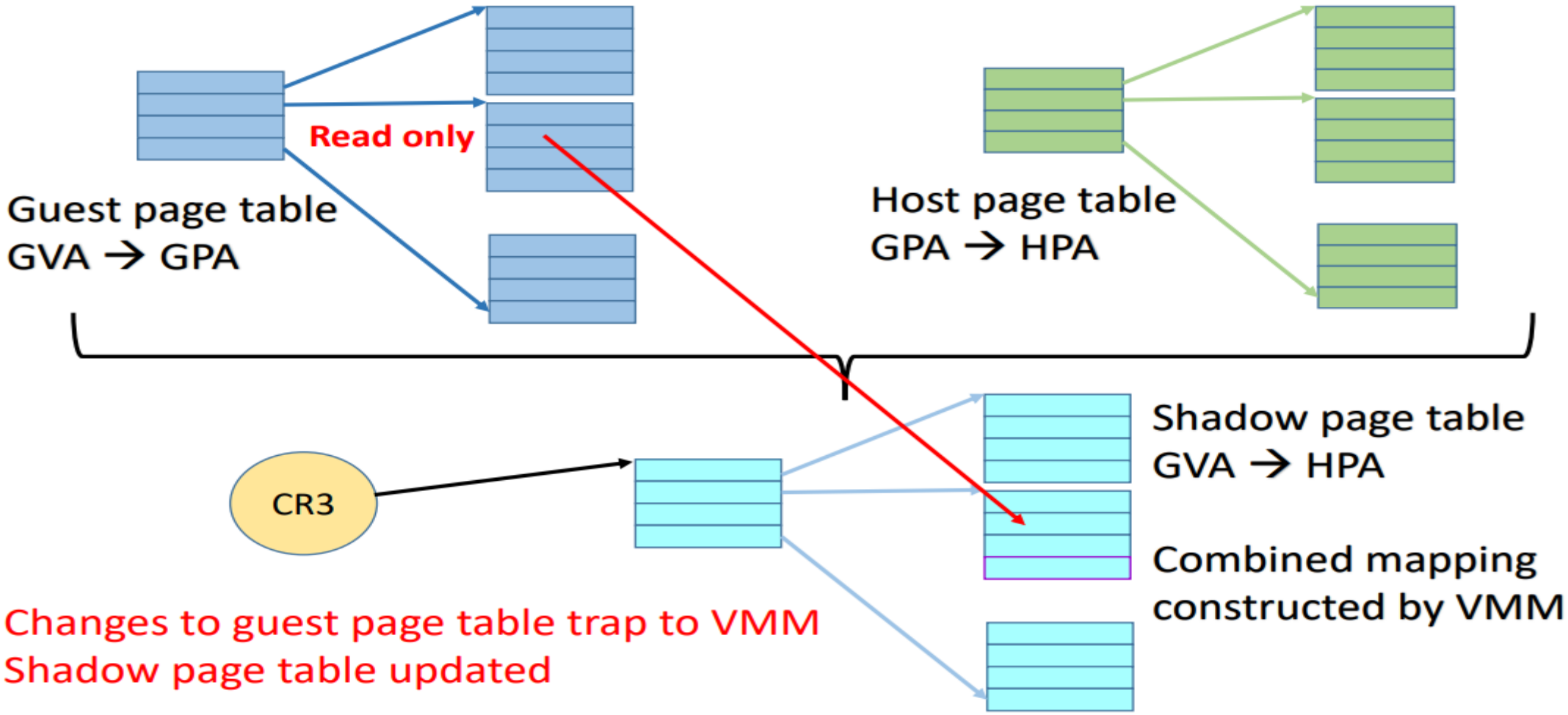
- Guest page table has GVA→GPA mapping
  - Each guest OS thinks it has access to all RAM starting at address 0
- VMM / Host OS has GPA→HPA mapping
  - Hypervisor knows mapping between host virtual address (HVA) and guest physical address (GPA), because it has setup its userspace memory as guest RAM
  - Host OS knows mapping from HVA to HPA for all processes, so it knows GPA→HPA
  - That is, guest "RAM" pages are userspace process pages that are distributed across host memory and host OS knows the physical locations of these guest "RAM" pages
- Which page table should MMU use?
- Shadow paging: VMM creates a combined mapping GVA→HPA and MMU is given a pointer to this page table
  - Used in VMWare workstation (full virtualization)
- Extended page tables (EPT): MMU hardware is aware of virtualization, takes pointers to two separate page tables
  - Used in QEMU/KVM

Extended page tables



- 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

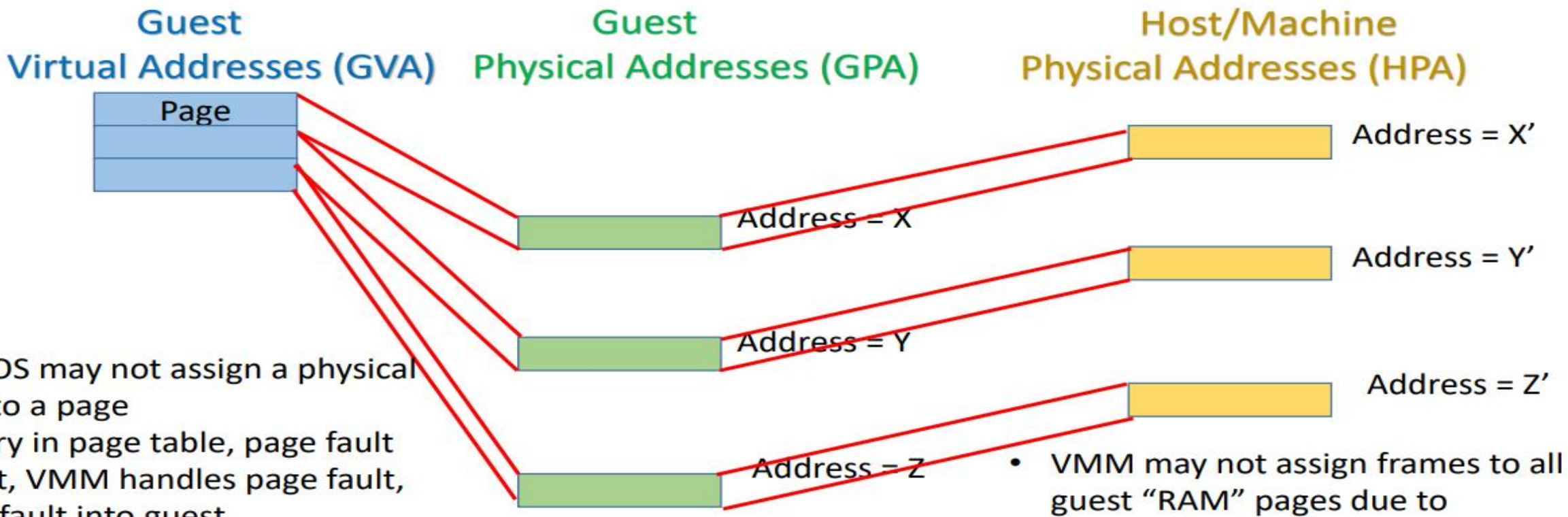
Shadow page tables



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)

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 technqiues

- VMM reclaims memory from guest “RAM” under memory pressure
- Uncooperative swapping: VMM reclaims some guest “RAM” pages and swaps them to disk
  - Page fault and memory assigned when guest accesses the page
  - May hurt performance, important pages may be swapped to disk
- Ballooning: VMM opens dummy device, requests pages from VM (“inflating the balloon”), then swaps these pages out
  - Since pages assigned to a device, pages not used by other processes in guest
  - Cooperative, guest can assign free pages
  - Lesser impact on performance, but reclamation takes time, requires guest changes
- Memory sharing: memory pages with identical content across VMs (OS images, zero pages etc.) shared between VMs
  - One physical frame mapped into page tables of multiple guests
  - Scan pages periodically to compute and match hash-based similarity of pages

Summary

- Memory virtualization
  - Extended page tables
  - Shadow page tables
- Memory reclamation techniques
  - Uncooperative swapping
  - Ballooning
  - Memory sharing

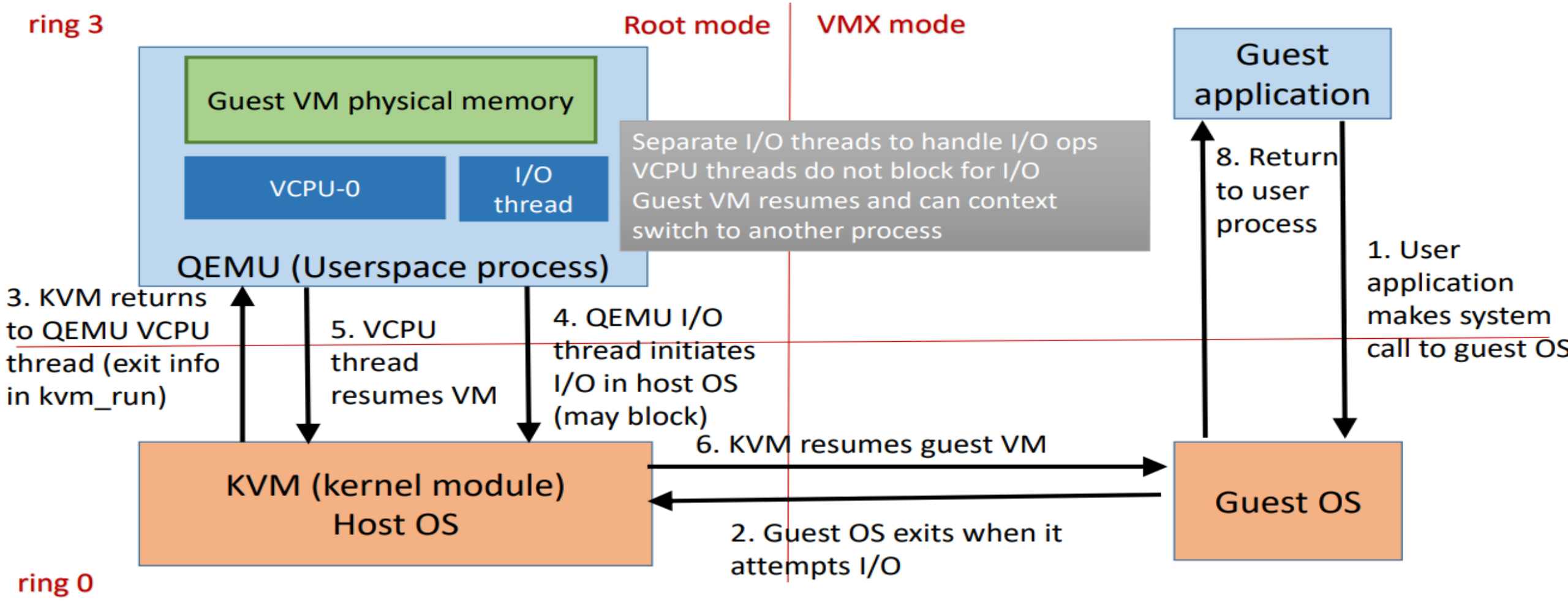
Techniques for I/O virtualization

- Guest OS cannot get full access to I/O devices
  - VMM must share I/O device access across guests
- Two ways to virtualize I/O devices:
  - Emulation: I/O access in guest traps to VMM, which performs I/O
  - Direct I/O or device passthrough: a slice of device is assigned directly to guest
- Many optimizations exist, only basics discussed here

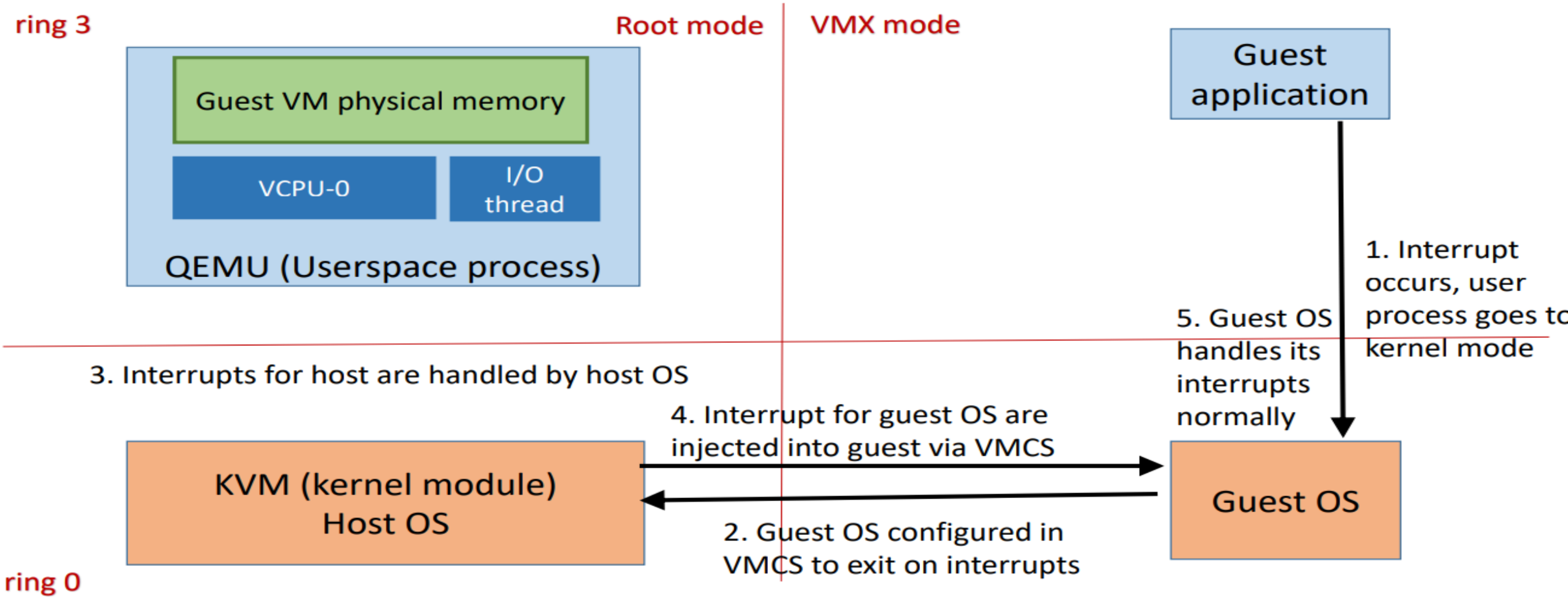
Communication between OS and device

- Device memory exposed as registers (command, status, data etc.)
  - I/O happens by reading/writing this memory
  - E.g., write command into device register to begin I/O
- OS can read/write device registers in two ways:
  - Explicit I/O: in/out instructions in x86 can write to device memory
  - Memory mapped I/O: Some memory addresses are assigned to device memory and are not to RAM. I/O happens by reading/writing this memory.
- Accessing device memory (via explicit I/O or memory mapped I/O) can be configured to trap to VMM
- Device raises interrupt when I/O completes (alternative to polling)
  - Modern I/O devices perform DMA (Direct Memory Access) and copy data from device memory to RAM before raising interrupt
  - Device driver provides physical address of DMA buffers to device

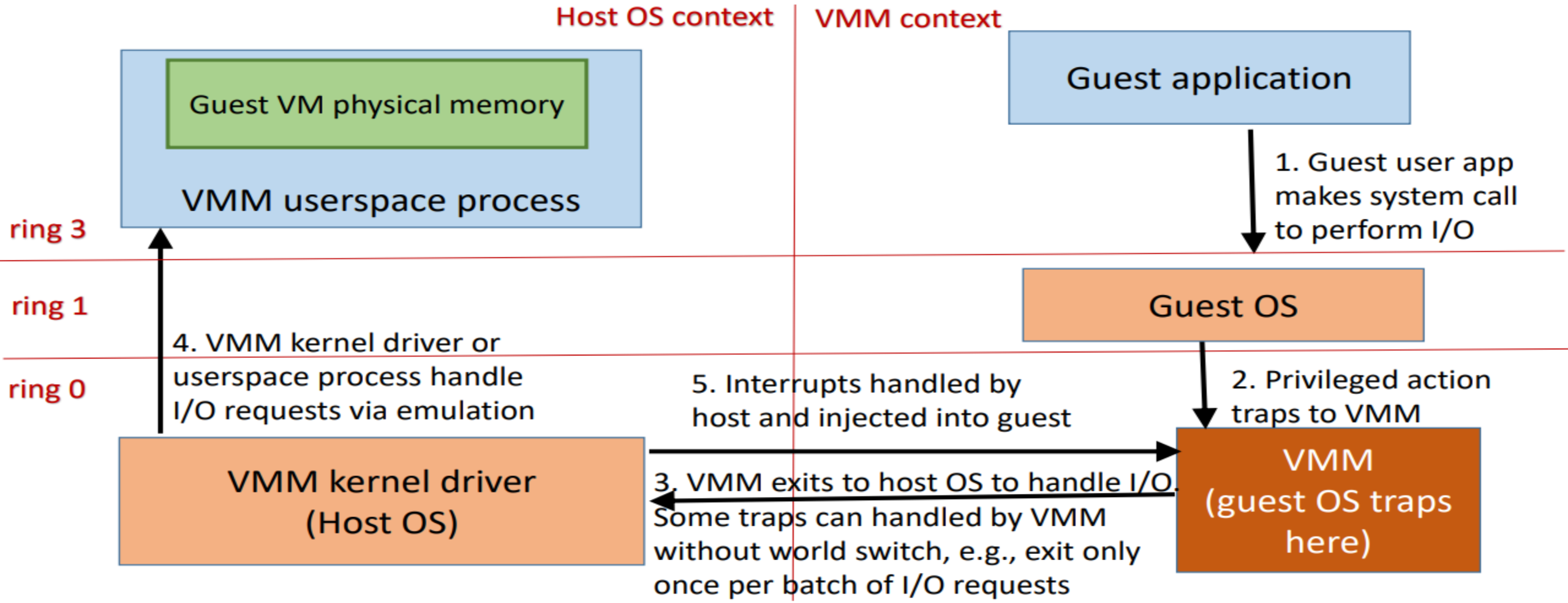
QEMU/KVM I/O handling



QEMU/KVM interrupt handling

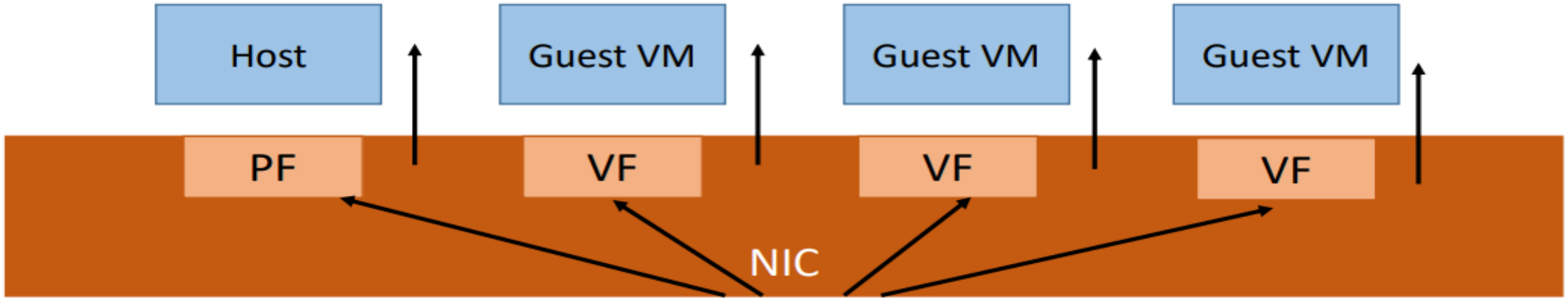


Full virtualization VMM architecture



Device passthrough or Direct I/O

- More efficient than device emulation
- Example: SR-IOV (Single Root IO Virtualization) in network devices
  - Network card has one physical function (PF) and many virtual functions (VFs)
  - PF managed by host OS, each VF assigned to one guest VM
  - Each VF is like a separate NIC, and is bound to a guest VM
  - Packets destined to the MAC address of VM are switched to corresponding VF



SR-IOV (Single Root IO Virtualization)

- SR-IOV NIC communicates directly with device driver in guest OS
  - Packets do not go to the host OS stack at all
  - Packets switched at Layer-2 using VM virtual device's MAC address
  - Packets DMA'ed directly into guest VM memory, host OS not involved
  - But, interrupts may still cause VM exit (interrupt can be for host too)
- Challenge: when guest device driver provides DMA buffers to VF, it can only provide guest physical addresses (GPA) of the buffer
  - NIC cannot access the DMA buffer memory using GPA alone
- SR-IOV capable NICs have an inbuilt MMU (IOMMU) to translate from GPA to HPA

Summary

Techniques for I/O virtualization

- Device emulation
- Device passthrough or direct I/O (SR-IOV)