High-level introduction to virtualization's low-level

Ronen Hod – Red Hat's KVM team

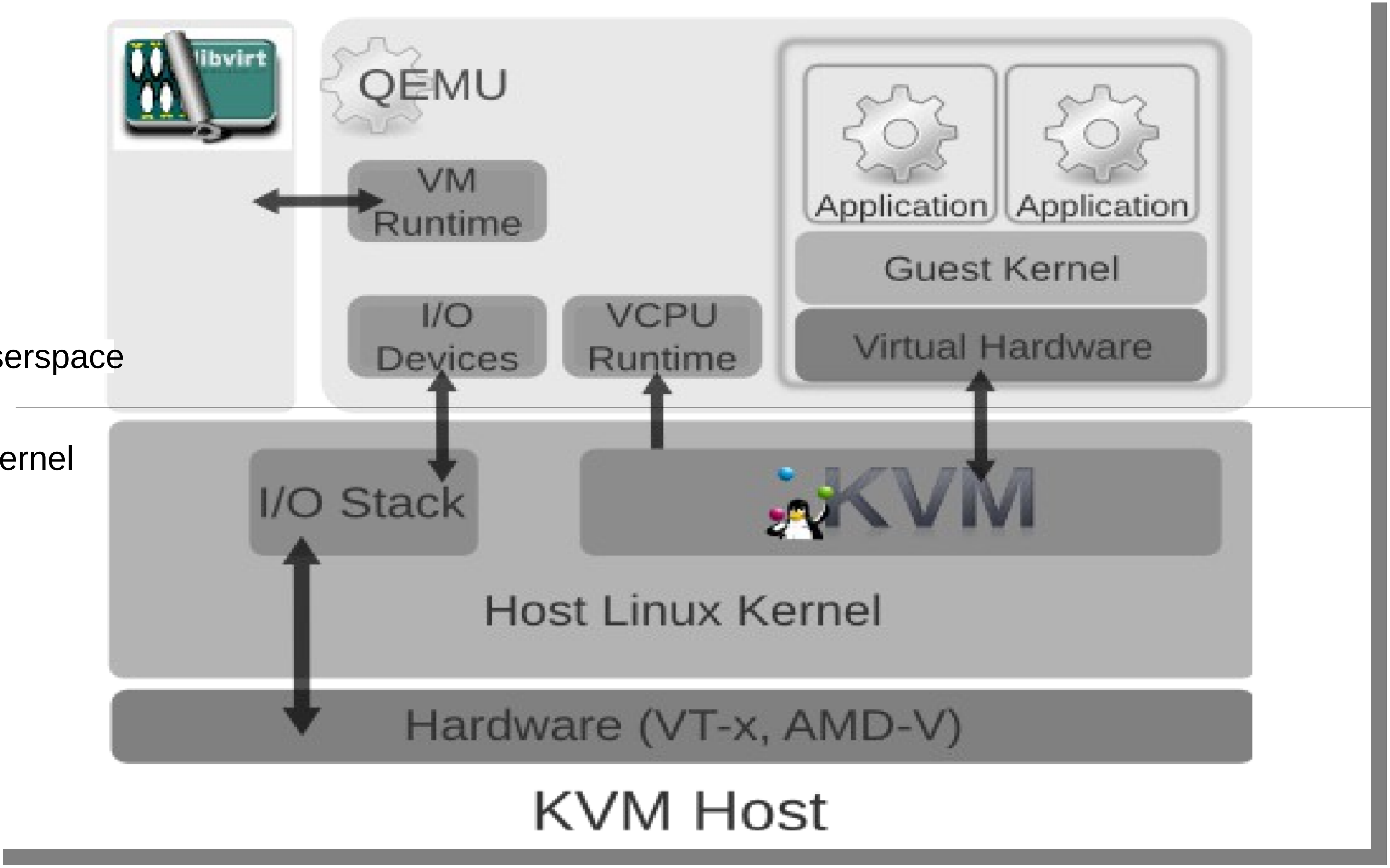
Aug 2013

Agenda

- Expectations setting
- Introduction to virtualization
- Introduction to memory management
- Analysis of 5 memory related assembly instructions
    - How it works on real hardware
    - Hypervisor's emulation

Expectations setting

KVM Architecture – Not today. We will dive all the way in.



Expectations setting

- This talk is at the level of “assembly” instructions
  - The common denominator of all software (BIOS, O/S, Java, browser, whatever).
- Virtualization is huge
  - We will focus on 5 variants of the X86 “MOV” instruction, and try to “fully” understand them.
- The focus is on memory management implications.
- We will spend time refreshing our native O/S knowledge
- The examples are simple and simplified. E.g., 32-bits addressing
- Do I promise to tell the whole truth and nothing but the truth?
  - Not really

Introduction to virtualization

What do we expect from a Hypervisor (VMM)

- Provide a VM (Virtual Machine), often referred to as Guest
- The Hypervisor emulates a hardware by ALL means
  - A VM can run code (“assembly” instructions) just like on real hardware, and it will work just the same.
    - What code?
      - BIOS, Operating system, applications
      - Code that accesses registers, memory, devices
      - Code that manages memory (paging system) and processes.
  - The (emulated) devices read/write to the VM's memory (using “DMA”)
  - The “Hardware” issues interrupts
- Timing is different from real hardware

Virtualization techniques

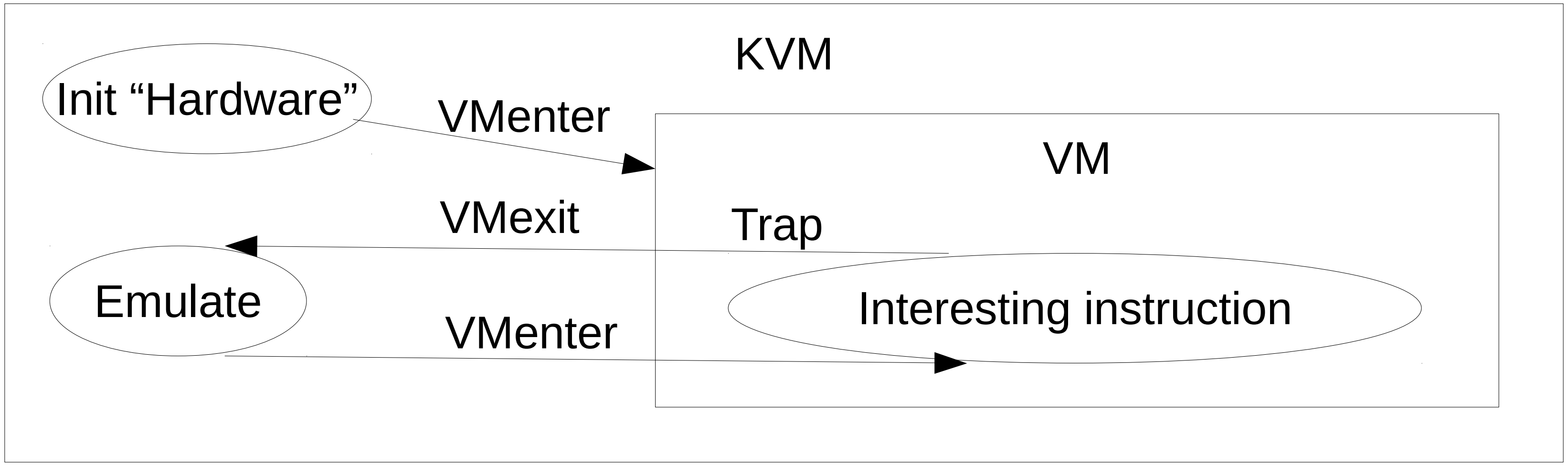
- Para-virtualization (old XEN)
- **Trap and Emulate**
- (dynamic) Binary translation (old VMWare)
- **Hardware assisted**

How does it work

- Efficiency – In practice, most of the time the Hypervisor simply lets the guest run on the real CPU and memory, not knowing what the guest is doing.
  - Access to Registers & memory, function calls, calculations, ...
- If the guest is executing an “interesting” assembly instruction then the hardware traps it, and the hypervisor emulates the instruction carefully
  - Examples of emulated instructions
    - Access to “hardware device” (MMIO / PIO)
    - Special instructions (CPUID, LGDT)
    - Access to interesting “registers” that have side effects

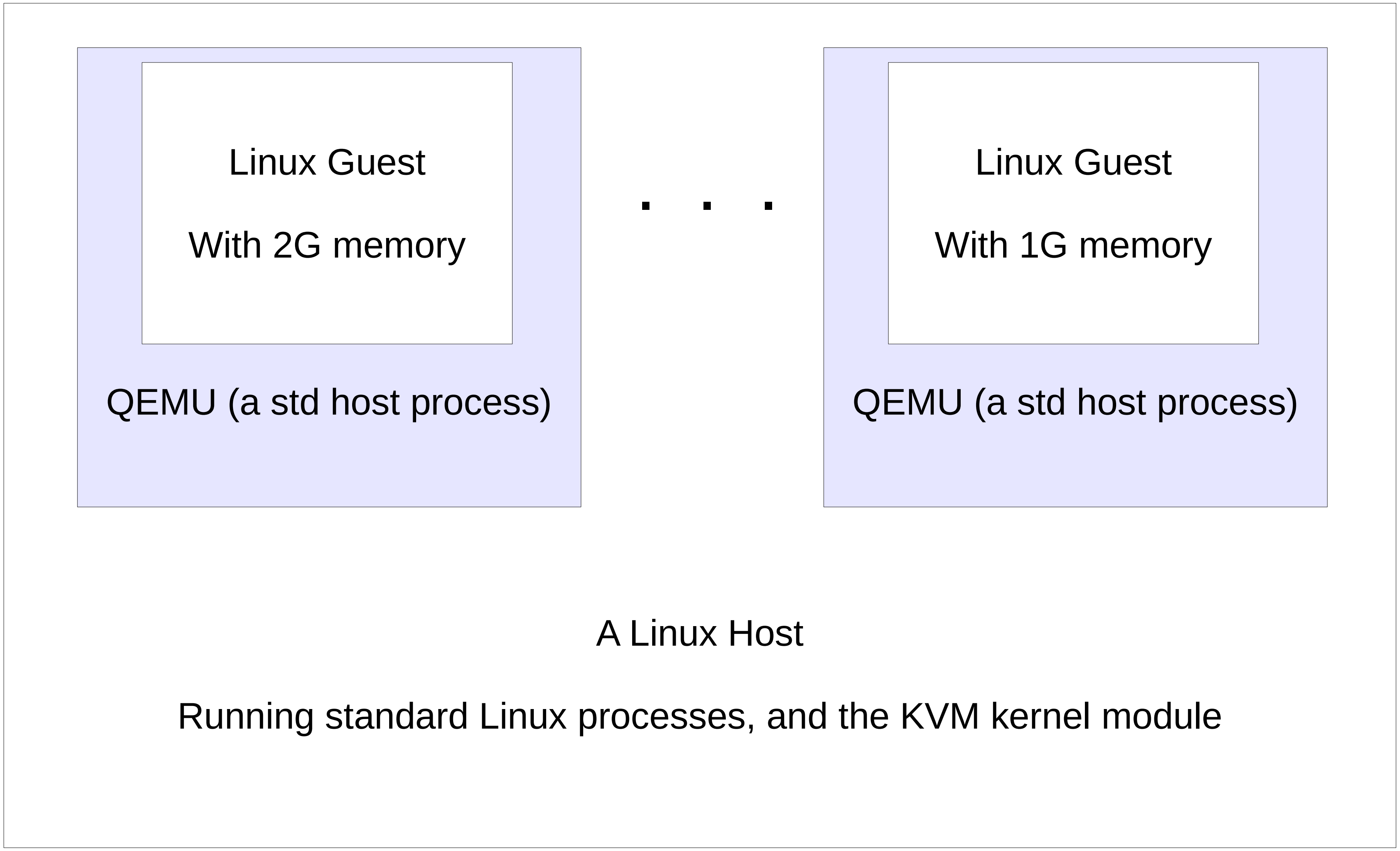
How does it work (cont.)

- Hardware support for virtualization
  - “luckily” modern x86 hardware is virtualizable, in the sense that every assembly instruction that the hypervisor needs to emulate can be trapped.



- Fully Virtualizable x86 hardware became available in 2005 (Pentium-4 662 and 672)
- VMCS – Virtual Machine Control Structure
  - VMenter/VMexit control and status

KVM Host & Guests (memory wise)


Linux Guest
With 2G memory
QEMU (a std host process)
. . .
Linux Guest
With 1G memory
QEMU (a std host process)
A Linux Host
Running standard Linux processes, and the KVM kernel module

Add to the confusion (side notes)

- With KVM, the Hypervisor itself is implemented mostly in QEMU, which is a standard Linux-host process (that runs in user-space).
- KVM is also the name of a kernel module (not just the technology).
  - The KVM-module runs code that requires kernel permissions (interrupts, memory management, ...)
  - A few performance critical devices are implemented in the kernel (Vhost)
- The guest's “physical memory” is a standard Linux-host memory which is part of QEMU's data structures

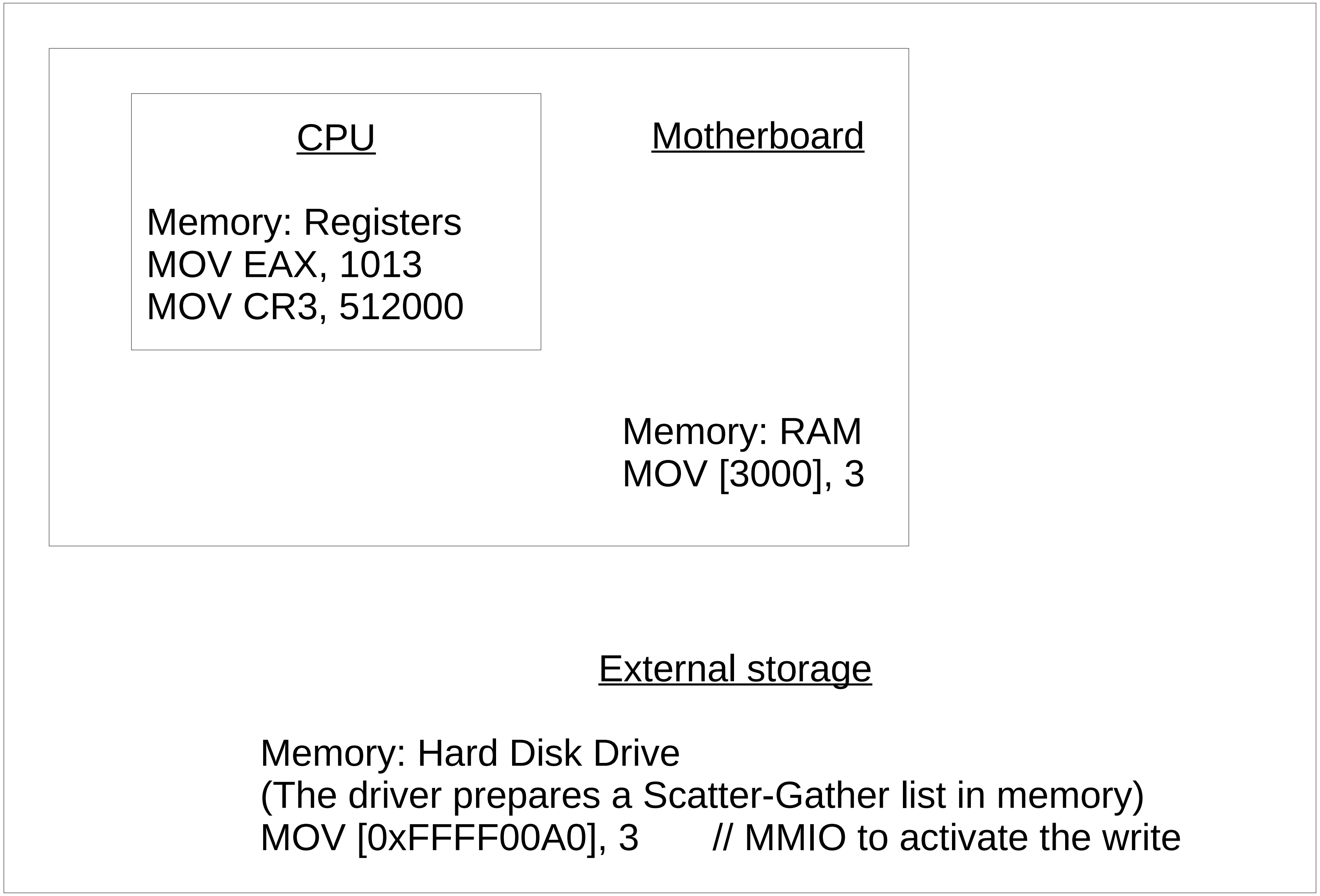
Add to the confusion (cont.)

- QEMU, just like any other standard (userspace) application
  - Is subject to scheduling (on the host) – it can be stopped and/or moved to a different physical CPU
  - Is subject to paging, its memory (that mostly comprises of the VM's memory) can be paged out to swap.
  - Can access host resources such as files and network in order to serve the guest
- All of the host facing activities have no effect on the VM's state. The guest's correctness is not affected by the host-side events
- We will ignore it from now on.

Let's examine one of the most popular instructions
Writing to a general purpose register

Mov EAX, EDX

3 memory levels and how they are accessed



Memory: Registers
```
MOV EAX, 1013
MOV CR3, 512000
```

Memory: RAM
```
MOV [3000], 3
```

Memory: Hard Disk Drive
(The driver prepares a Scatter-Gather list in memory)
```
MOV [0xFFFF00A0], 3       // MMIO to activate the write
```

Example 1: Mov EAX, EDX

- Explanation
  - EAX := EDX
  - Both EAX and EDX are memory registers (in the CPU)
- What can go wrong?
  - Nothing
  - Well, almost. The instruction itself can reside in memory that is not mapped. We will ignore it.
- What will happen (when the CPU is executing this guest's instruction).
  - The instruction is executed (as is, by the CPU)
  - The CPU will also advance the IP to the next instruction
  - KVM is unaware of it

Let's examine another popular instructions

Writing to memory

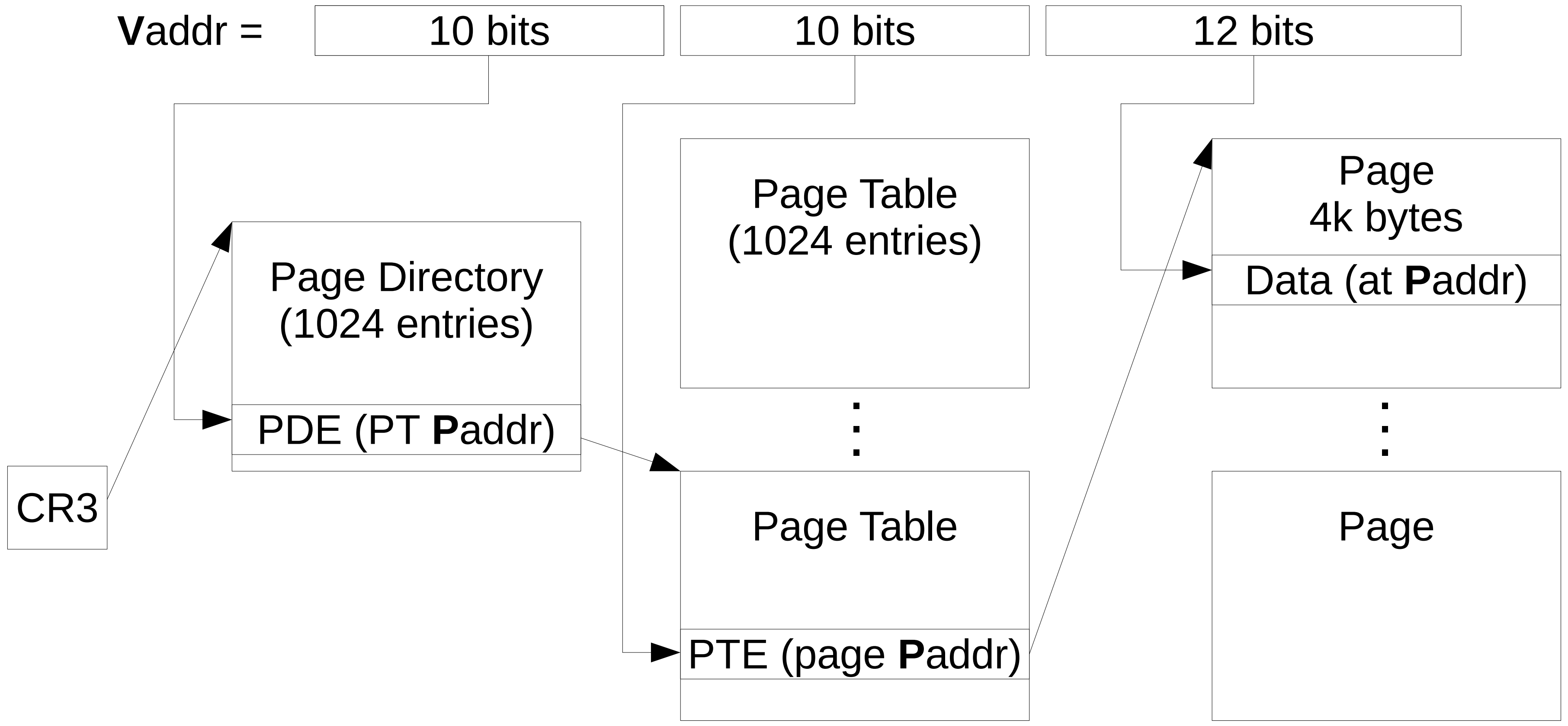
Mov [10000], EDX

A short introduction to virtual memory

- Note: **Virtual** memory has nothing to do with hypervisor's **virtual**ization. It works on the native machine
- Note that **V**irtual**addr** 10000 of different processes are distinct.

A short introduction to virtual memory (cont.)

- The MMU (inside the CPU) translates every **V**irtual (Linear) address to **P**hysical address.



A short introduction to virtual memory (cont.)

- A PDE/PTE also contains some bits for present, write-able, ...
- An attempt to access a non-present page triggers a page fault
- Where do these mysterious page-table reside, and how does the O/S update them?
  - A page-table is a standard 4k data page in memory
  - It plays the role of a PT only when it is used during the translation of a virtual address to physical address
  - A page table can be written just like any other memory page. Naturally, through a virtual address.
  - The O/S allocates PTs and updates their content
- Performance - A hardware cache (TLB) is used to accelerate the translation from Vaddr to Paddr

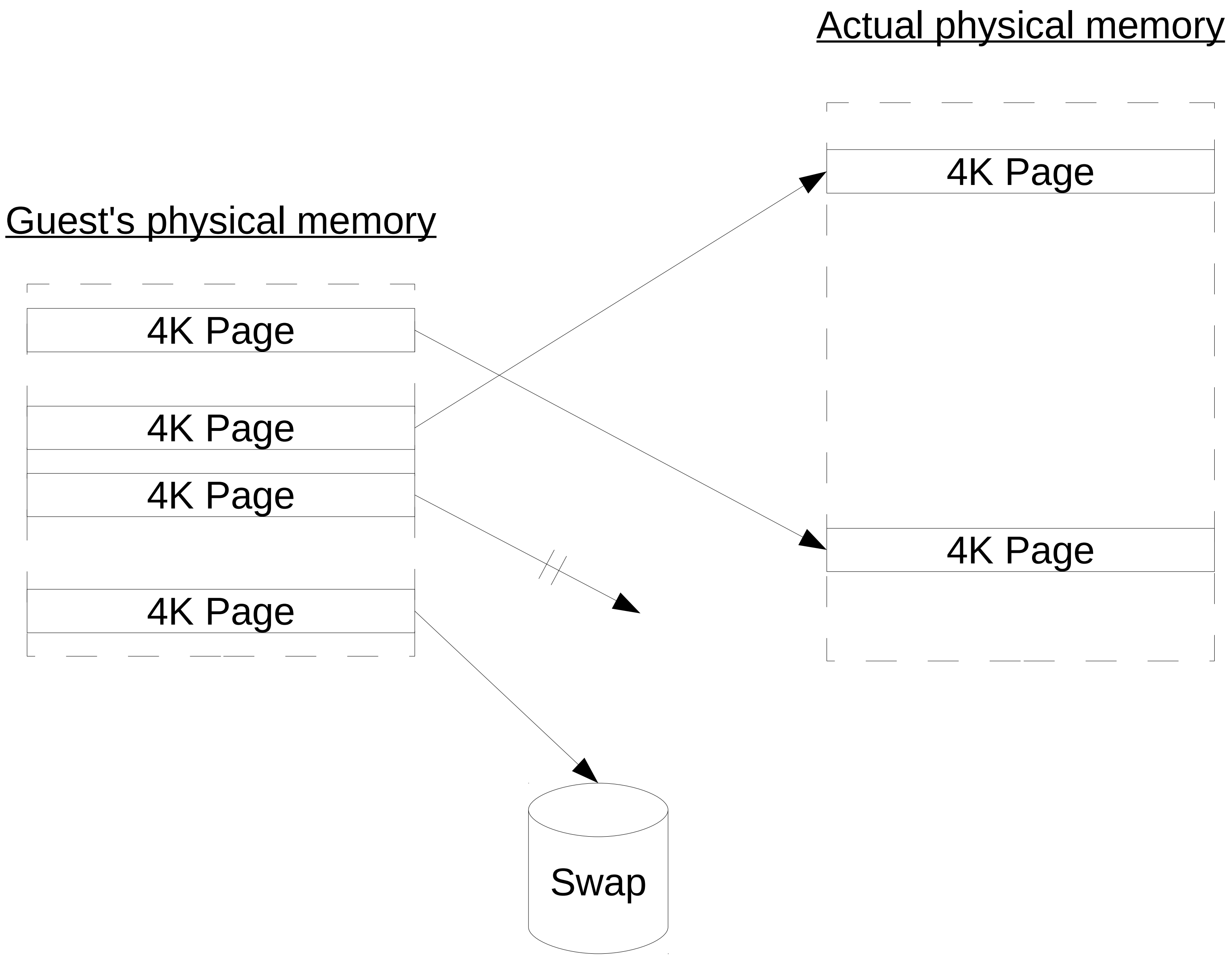
How hypervisors implement the virtualization of “virtual memory”

Memory virtualization

- There are several techniques to virtualize memory-access
  1. Emulate every instruction
     – This is what you get when running QEMU (in usermode), without the kernel's assistance that the KVM-module provides.
     – Can be accelerated using binary translation
  2. Shadow page tables
     – Involves a KVM kernel module
     – This technique is still in use for Nested-VM
  3. Using the hardware's EPT/NPT
     - Intel: Extended Page Tables
     - AMD: Nested Page Tables

     – KVM and other hypervisors use the hardware's support

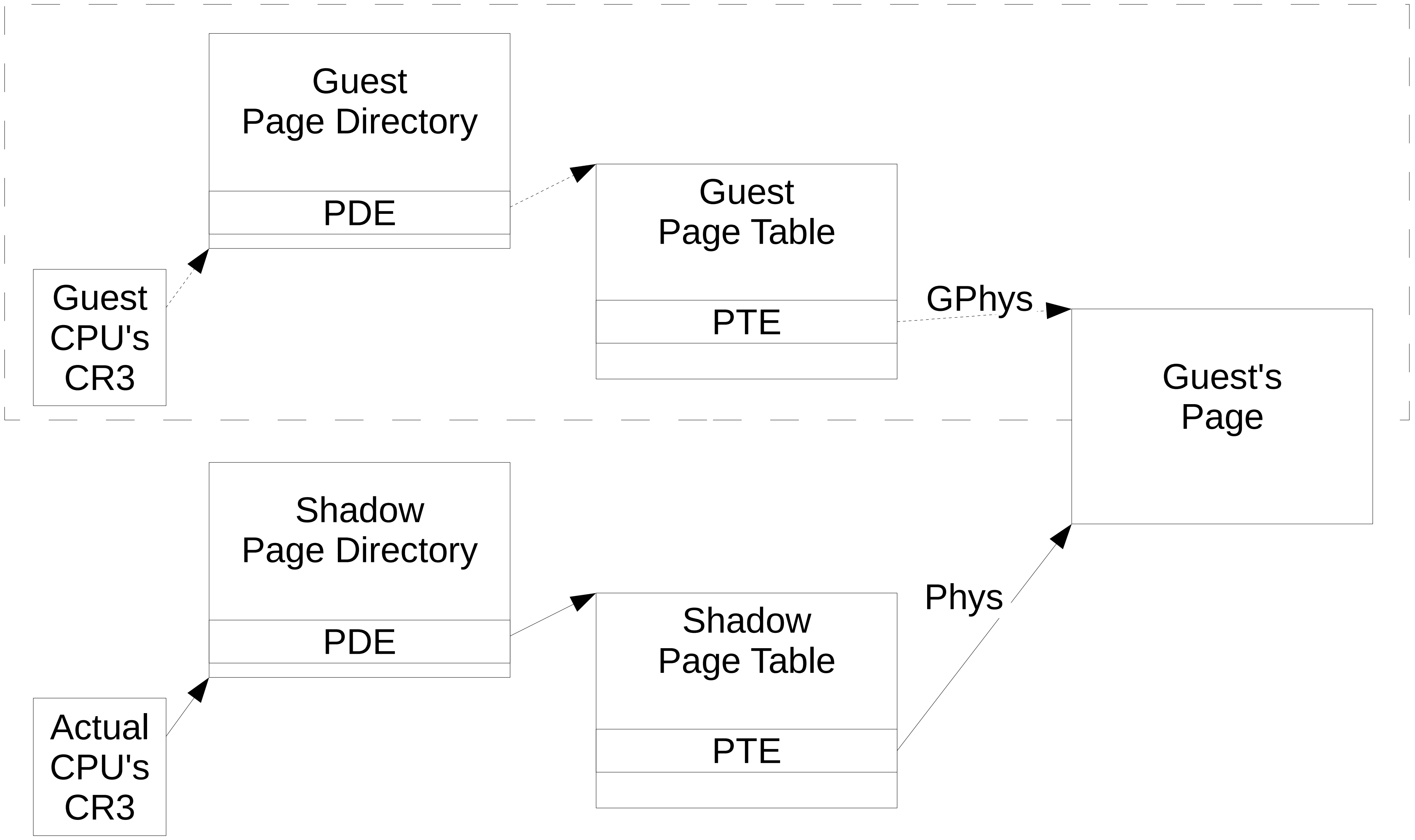
Guest's physical memory vs. Physical memory



Shadow page tables

- The Guest O/S is unaware of the fact that its “physical” memory is virtualized, and continues to manage its page-tables (upon Guest's memory allocation, swapping, attempts to write on the Zero-page, etc)
- KVM is maintaining a parallel (shadow) set of page-tables that implements the actual mapping to physical memory
  - I.e., the real CPU's CR3 points to the shadow PD

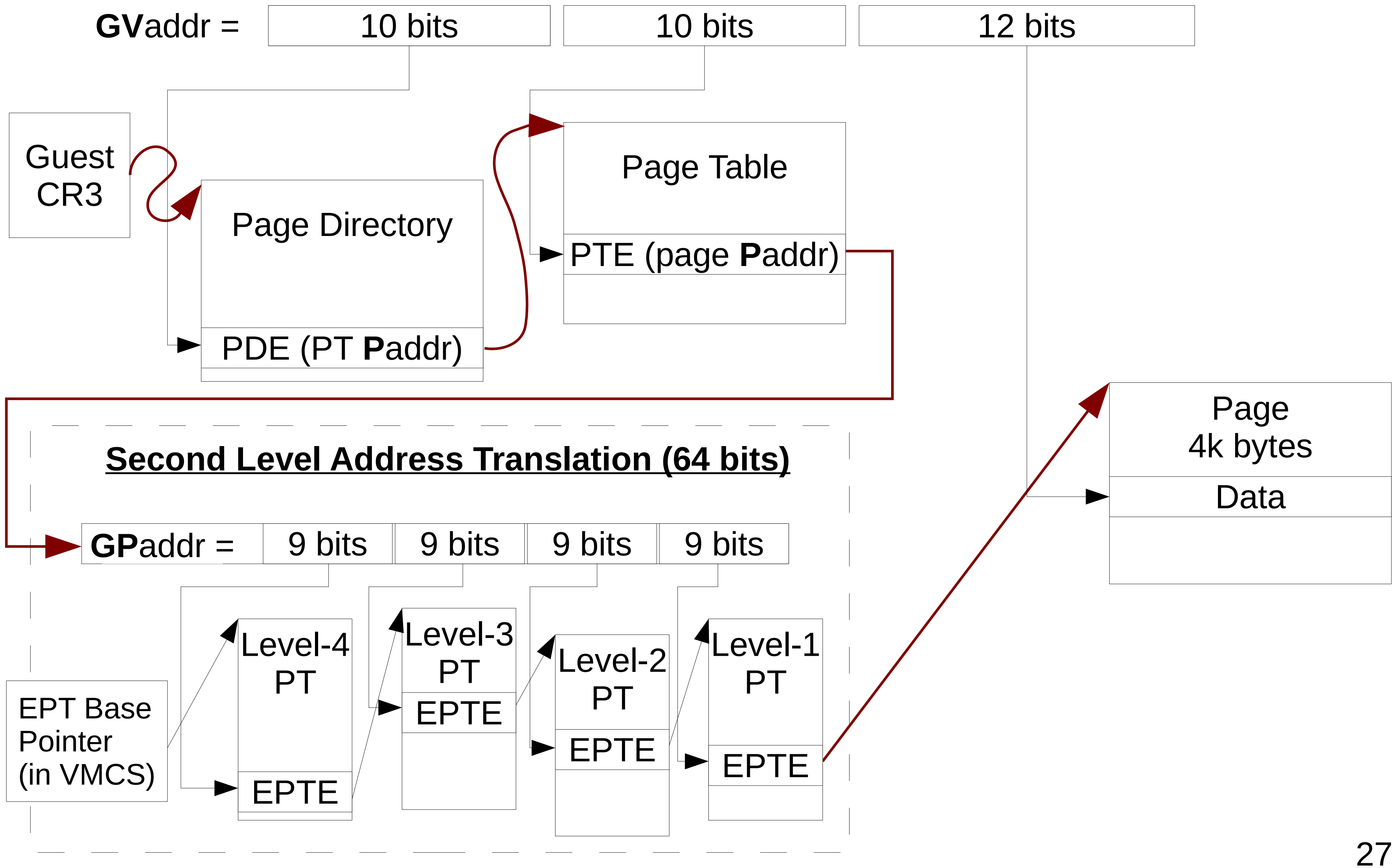
Shadow page tables (cont.)



Shadow page tables (cont.)

- Past hypervisors used to run using the shadow page tables
  - There were too many VM exits (accessed & present bits), and up to 25 mem accesses to complete
- With EPT (NPT), hardware support, the number of exits was reduced dramatically

Hardware support - EPT / NPT



Hardware support - EPT / NPT (cont.)

- The MMU support for calculating the Gphys address works just the same as on bare metal.
- The calculated Gphys goes through another level of translation in hardware (SLAT)
- Once again these EPT page tables reside in standard memory (naturaly, host mem)
- The TLB can cache the full translation – from Gvirt to physical address
- The main advantages
  - There is no exit to the hypervisor due to pure guest's page-faults.
  - Guest context switch that would trigger a shadow PTs recalculation (had shadow PTs been used) has no effect on the EPT tables.

Writing to memory

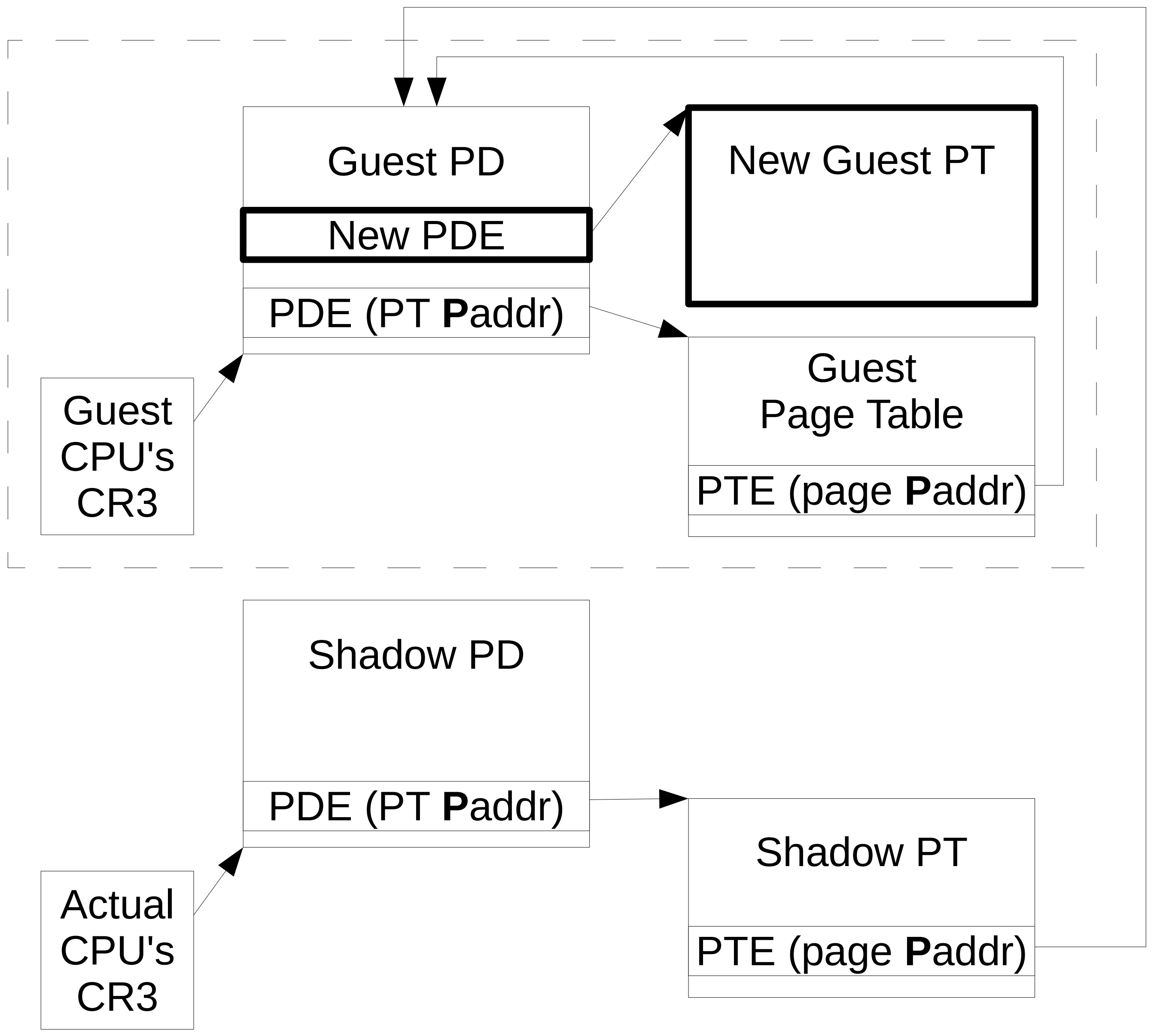
Example 2: Mov [addr], EDX

- Explanation
  - [addr] is the content of a memory
  - "addr" is a virtual address (guest wise)
- What can get complicated?
  - Guest PF - "addr" is not mapped or write-protected (according to the guest's PTE bits)
    - Let the guest handle it.
  - Shadow Pts issues
    - "addr" is inside a guest PT (that has a shadow PT)
      - Guest-PTs can be write-protected in order to track changes, and update their shadow accordingly
  - "addr" is not plain memory, it is used to access a (guest) device (MMIO)

(SKIP?) Example 2 (cont.): Mov [addr], EDX

- Say that shadow page tables are used, and that it is the first time the guest is accessing this page.
- Let's discuss a guest PF due to PDE
  - The MMU starts with CR3 (naturally the real one), uses the first 10 bits from the linear (virtual address) to locate the PDE (in the shadow PD), and since the PDE's Present bit is 0 issues a PF.
  - The hardware generates a "PF exception" vmexit (to KVM). Prior to H/W support it was a standard PF.
  - KVM notices that the guest's PDE's Present bit is 0, so it emulates a hardware PF for the guest, and let it run.
  - The guest's MM will find/create its PT, puts its address in the (guest's) PDE, set the present bit in the PDE
    - Note that in order for the guest to write on the PD it must be mapped (See next page)
  - The IP is set to rerun the instruction, and it is rerun.

Shadow PTs – Guest updates “New PDE”



The guest is updating its new PDE
A guest PTE must point to the PD (the PD as a data page).
A shadow PTE also points to the PD.

The Hypervisor needs to track changes to guest PTEs.

Shadow PDE's are Lazy evaluated

Writing to MMIO (Memory Mapped IO)

Example 3 - MMIO: Mov [addr], EDX

- Explanation
  - “addr” is a virtual address (guest wise), whose **G**Phys address is not memory. The guest uses it to access a device
- A hardware device (such as an IDE disk) specifies what will happen when writing a given value to an address in its MMIO range.
- As far as the Guest is concerned, KVM needs to accurately emulate the behavior of a real device.
- KVM will often “cheat” and implement the disk drive storage using a QCOW/raw file on the disk, or on the network.

Example 3 - MMIO: (cont.)

- Reading from disk to memory
  - Guest O/S prepares a description of the task
    - (Scatter Gather) List of disk-addr, mem-addr, length
  - Guest O/S writes a pointer to the list to the device's MMIO address
  - KVM has to emulate the storage device
    - Read the correct data from the guest's disk storage (probably a host QCOW file) into the guest's memory.
      - Note that the guest's physical memory is a virtual memory of the QEMU (host) process
    - When done, emulate the hardware interrupt that such drive would generate upon completion.

In what sense KVM is cool

- In the previous example (MMIO)
  - The hypervisor has to read from the guest's “disk” into actual memory
  - This is not a simple task.
    - There are many types of storage implementations, such as QCOW/raw files, SCSI/SATA, ISCSI, NAS, ...
    - The I/O of different VMs need to be scheduled and the disk access ordering should be optimized
    - Sometimes you need bounce-buffers, and sometimes you can use DMA to access the guest's memory.
    - While waiting for I/O the CPU can run something else.
  - KVM uses the host's services, and lets the host take care of all this.
    - KVM immediately benefits from every Linux optimization (scheduler, NUMA, THP, I/O, ...)
    - KVM immediately supports every hardware that is added to Linux

Miscellaneous topics

Example 4 – Interrupt Injection to Guest

- Real hardware implementation of interrupts
  - According to the interrupt type
    - Checks the interrupt-enabled and other conditions
  - Save the current context (IP, flags, SP, ...) in the stack
  - Move the IP to the interrupt handler's code (can be found in memory, in the IDT – Interrupt Descriptor Table)
  - Upon IRET, restore the saved context from the stack
- KVM emulates this behavior when it decides to inject an interrupt to the guest
- VT-x/AMD-V provide hardware support for interrupts injection
  - The Hypervisor can inject interrupts (set the VMCS's VM-entry before VMRESUME). H/W will take care of the guest's masks & priorities, and the actual injection.

How does a VM start

- Just like on real hardware (how else)
- QEMU puts the BIOS (default: SEABIOS) in the guest's memory (mapped at the end of the 4G)
- The guest's BSP's (Bootstrap Processor) state is initialized – registers, etc.
- The guest's APs (Application Processors) are uninitialized
  - Later the BSP will send IPIs to the Aps, and they will wake up and start running their code
- The BSP's IP is set to address 0xFFFFFFF0, and it starts running (assembly instructions)
  - The guest BIOS will scan the “hardware devices”, build tables, ..., load and run the O/S