Virtualization Technologies

Hardware assisted virtualization

Reference:

- Material available on the course website

Performance penalty

- Hardware virtualization techniques result in running the guest OS completely into user space
- Every time the guest OS switches to kernel space, we have a context switch to run the VMM, which mimics the execution of a privileged instruction or other operations (e.g. I/O)
- Although guest OS kernel operations are limited in time, they represent a significant performance penalty, due to the noticeable overhead in performing hardware emulation and instruction emulation

Hardware Assisted Virtualization

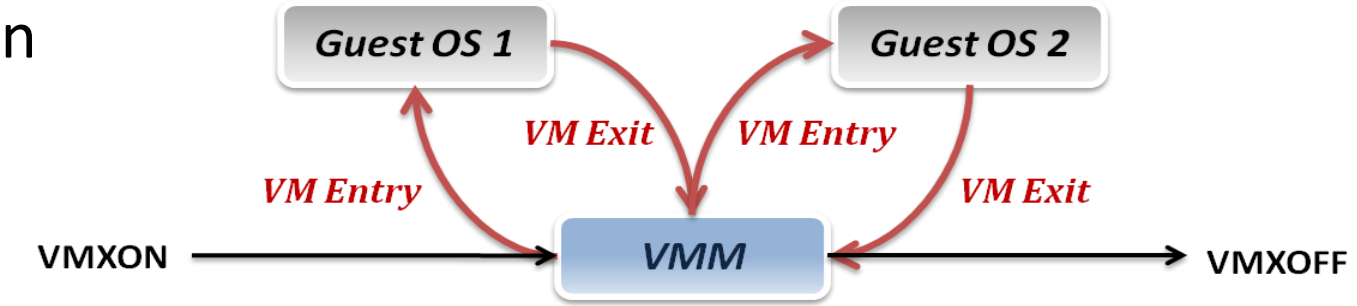
- In order to solve this problem and improve the performance both AMD and Intel have extended their CPU with new features starting from 2006
- Those features are explicitly designed to support efficient implementation of VMM-Hypervisor software
- Intel and AMD extensions are equivalent; thus, we focus on the Intel extension called VMX (Virtual Machine eXtension)
- The extension works on different aspects to minimize virtualization overhead

Root and non-root modes

- VMX technology introduces two **new operating modes** in Intel CPUs: the **root** and **non-root** modes
- These two modes are orthogonal to the already existing system and user modes
- The result is four possible combinations (in order of privilege):
  - root/system
  - root/user
  - non-root/system
  - non-root/user
- The root mode is intended for the VMM running on the host, while non-root mode is intended for the guest software running in the VM

Root and non-root modes

- The main purpose of these new modes is to put hardware-controlled limitations to the actions performed by the guest OS
- Whenever the guest OS tries to execute an instruction that would either violate the isolation of a VM or require software emulation, the hardware can trap and switch back to VMM execution
- Two new assembly instructions are introduced, the VMLAUNCH and VMRESUME, which both enable the root mode to launch the VM, *VM launch*
- Those instructions are only allowed in root/system mode
- The VM returns to the root mode and switch back to the VMM execution for a number of reasons, named *VM exits*

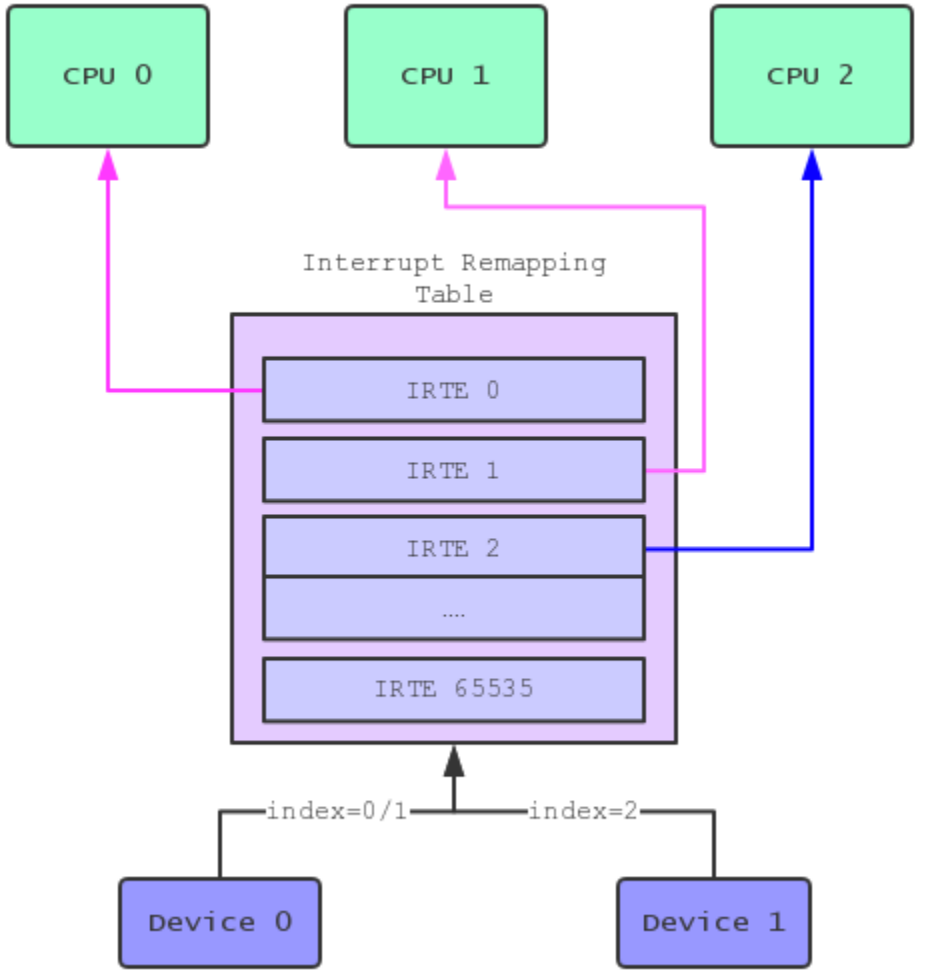


Root and non-root modes

- The root/non-root modes already allow to improve the efficiency, reducing the need for software emulation
- The non-root mode, via additional hardware support, allows the VM running in non-root mode to (we will see the details later):
  - Execute some privileged instructions
  - Handle directly (some) interrupts
- This without requiring VMM intervention for their emulation

Interrupt management

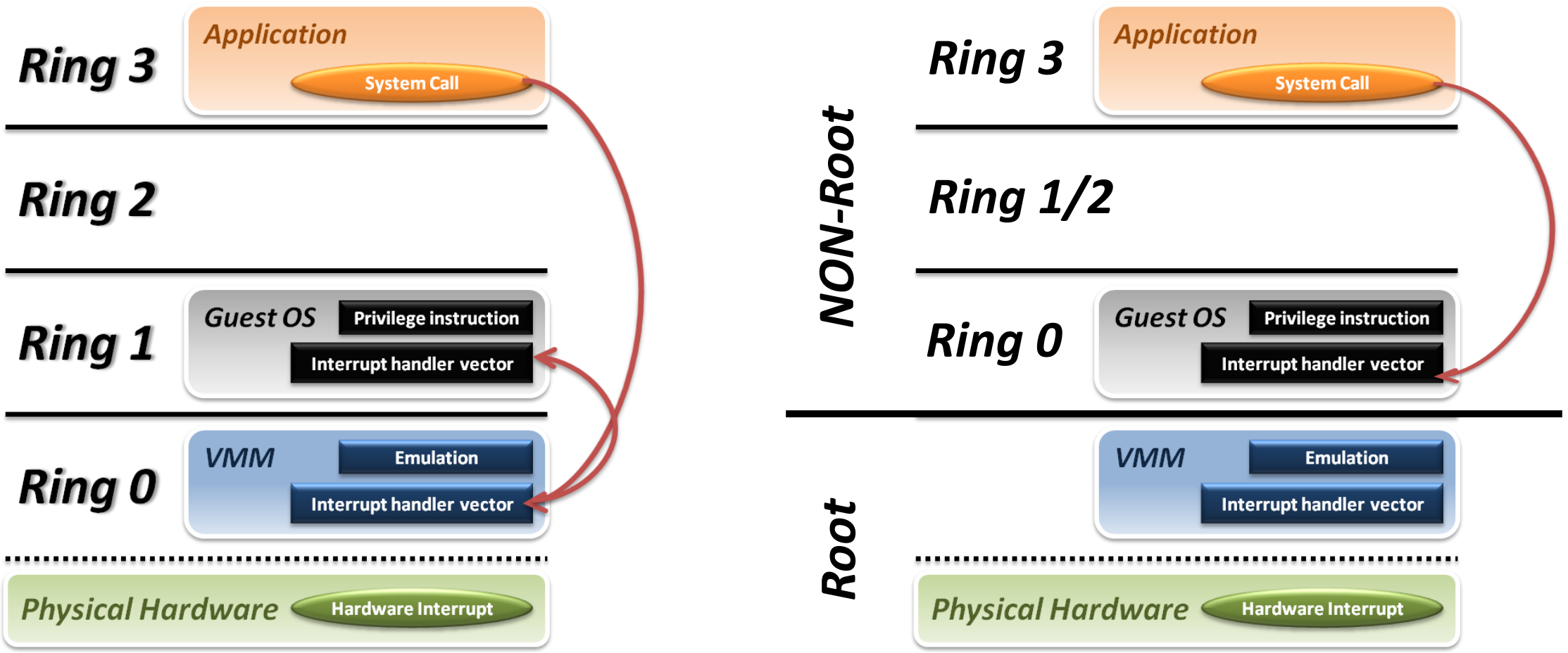
- To reroute interrupts an additional table is added, the Interrupt Remapping Table (IRT)
- The IRT allows to reroute every hardware interrupt directly by the VMM in a dynamic manner
- It has an entry for each possible interrupt request. Each entry maps the requests to the interrupt vector of a VM or to the interrupt vector of the VMM (or to other data structures – see later)
- Every time an interrupt is triggered, the IRT decides which Interrupt Vector handles the interrupt



Root and non-root modes

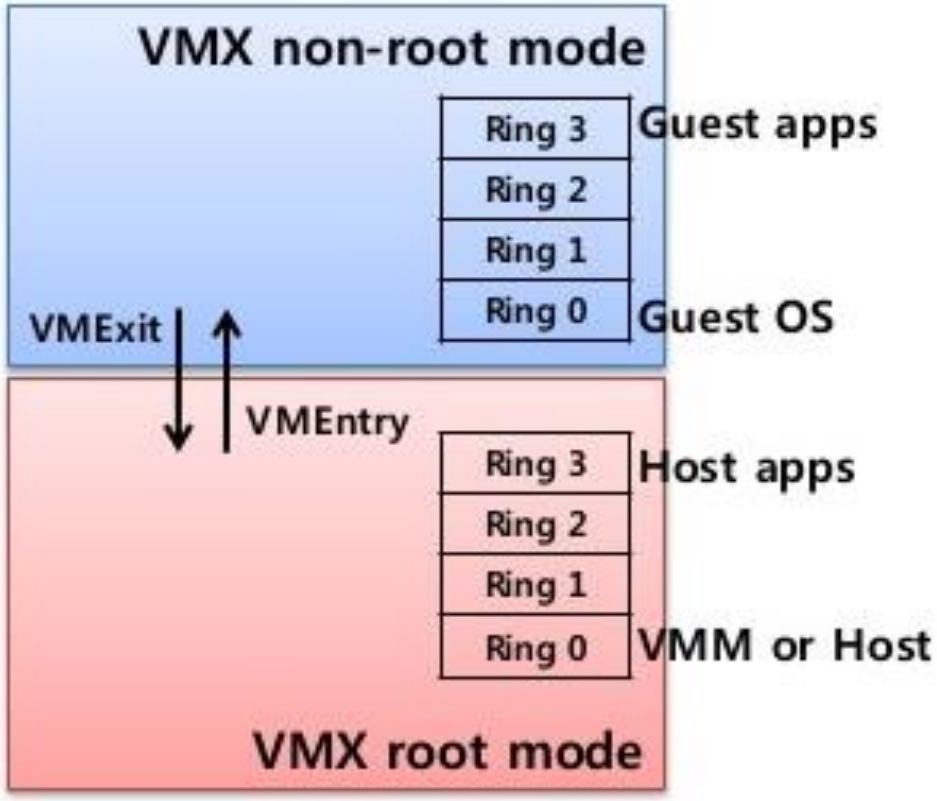
- An example is the need to emulate via software the execution of two instructions the INT and IRET, used to move from the user mode to the system mode and viceversa, used by user programs running inside the VM to invoke system calls
- Without hardware support, their execution was emulated via software by VMM, which was responsible for taking care of modifying the VM status in order to mimic the execution of the instructions (e.g. by modifying registries manually or triggering the execution of kernel space code)
- With root and non-root modes the execution of INT and IRET can be performed without the need to execute VMM code: whenever the INT is invoked the operating mode is switched from non-root/user to non-root/system, and viceversa for the IRET, this without the need to emulate the instruction

Example: INT execution on the VM



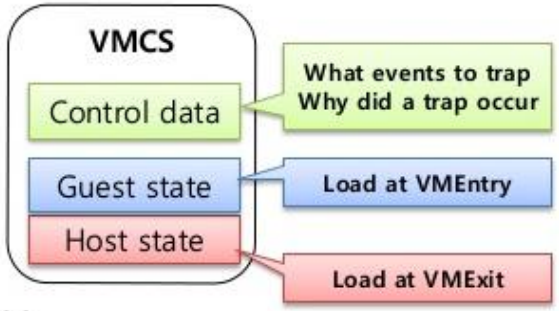
Root and non-root modes

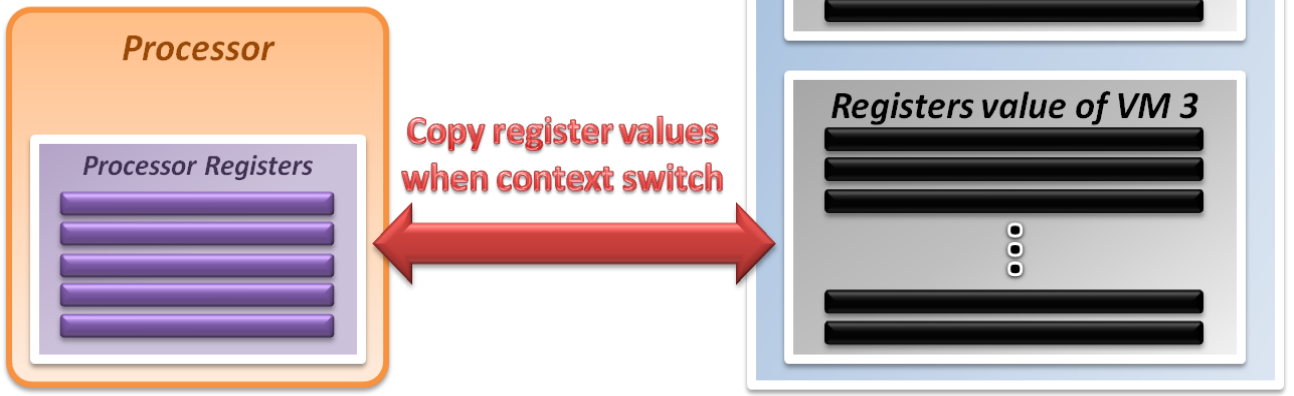
- The non-root/system and non-root/user modes allow to keep the distinction between user applications and OS kernel inside the VM
- In a similar manner, the root/system and root/user enables implementations in which the VMM is part of a standard OS running on the host, since userspace applications can run in root/user, while kernel OS and VMM operations are executed in root/system mode



Virtual Machine Control Structure

- Intel VMX introduces a new ***Virtual Machine Control Structure (VMCS)***, which is a control structure that contains all the information associated with the state of a VM
- The extension includes a set of instructions to manipulate such data structure that are used when the context switches to execute the VMM or the VM is restored





Virtual Machine Control Structure

- VMCS data structure has several fields that can be grouped as follows:
  - *Guest state*: stores the state of the (virtual) processor associated with the VM. The state is loaded from the structure to the processor during VM launch, while during a VM exit the state is stored back
  - *Host state*: store the state of the physical processor before the VM launch. It can contain the state of the VMM, running before VM launch. The state is restored during a VM exit, while it is saved at VM launch.
  - *VM execution control*: this field specifies what kind of operations and the instructions that are allowed and not allowed during non-root mode. Unallowed actions cause a VM exit
  - *VM enter control*: it contains several flags and fields that determine some optional behaviors of the root to non-root transition
  - *VM exit control*: likewise, but for the non-root to root transition
  - *VM exit reason*: it contains information related to the reason that caused the latest VM exit event

VMCS – Guest/Host states

- Among other information, the guest state contains the instruction pointer, that points to the first instruction to be executed after VM launch
- Likewise, the host state contains the value to be loaded into the instruction pointer at VM exit, to recover the execution of the VMM
- The VM exit reason section contains the code specifying the reason for the VM exit. Through this field the VMM can have information regarding the VM status and act accordingly

VMCS – VM execution control section

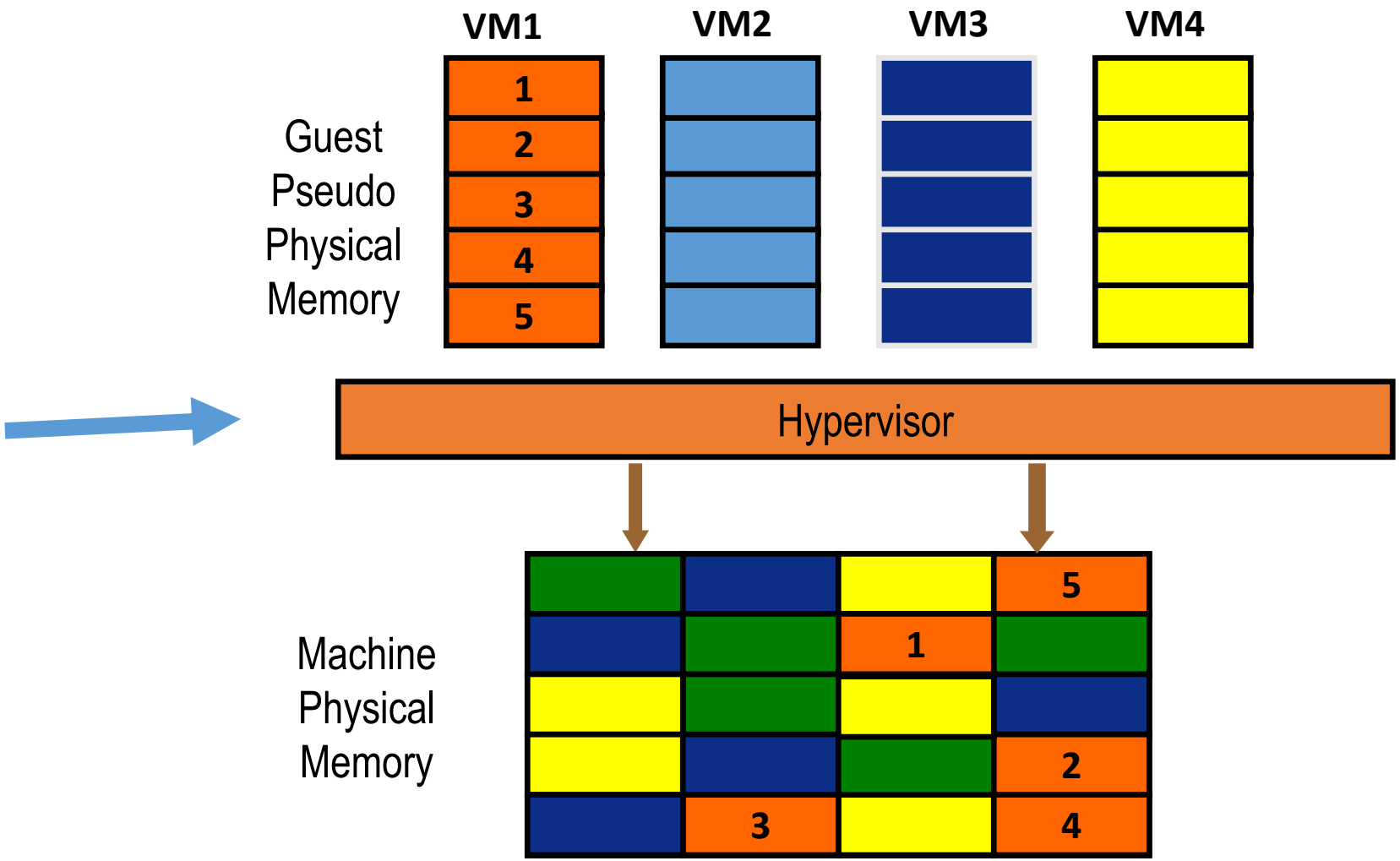
- The *VM execution control section* contains many flags, in particular to determine **interrupt and I/O management**, exploited in particular for *peripheral passthrough* (we will see that later):
  - a flag determines *what should happen when the CPU receives an external interrupt*, while running in non-root mode. Through this flag VMM can specify if the VM can handle interruptions directly or a VM exit must occur
  - a flag determines whether *some critical instructions should cause a VM exit or they can be executed by the VM*
  - *another set of flags specifies if I/O operations are allowed inside the VM or a VM exit is required*

VMCS – VM enter control

- In the ***VM enter control*** there are fields that can be used by the VMM to mimic a fake external interrupt or to inject exceptions and fault
  - *The VMM writes the vector of the desired interrupt into the VM enter control, during VM enter, the processor will perform all the actions to mimic the interrupt reception*, specifically:
    - it saves the state on the guest stack
    - it looks up the guest interrupt descriptor table to determine the address of the interrupt handler
    - it modifies the registries of the CPU to execute the handler
- Through this mechanism, fake interrupt can be generated into a VM by the VMM

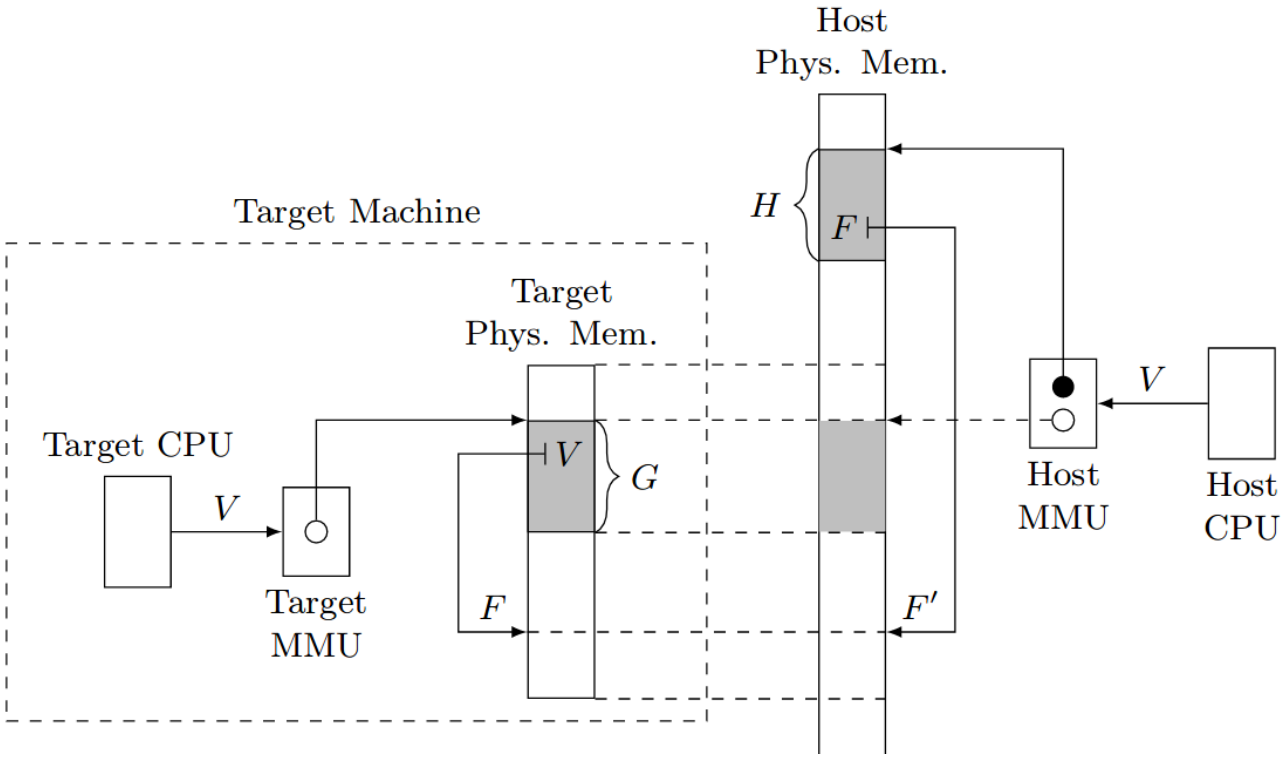
RAM Management

Hypervisor Goal

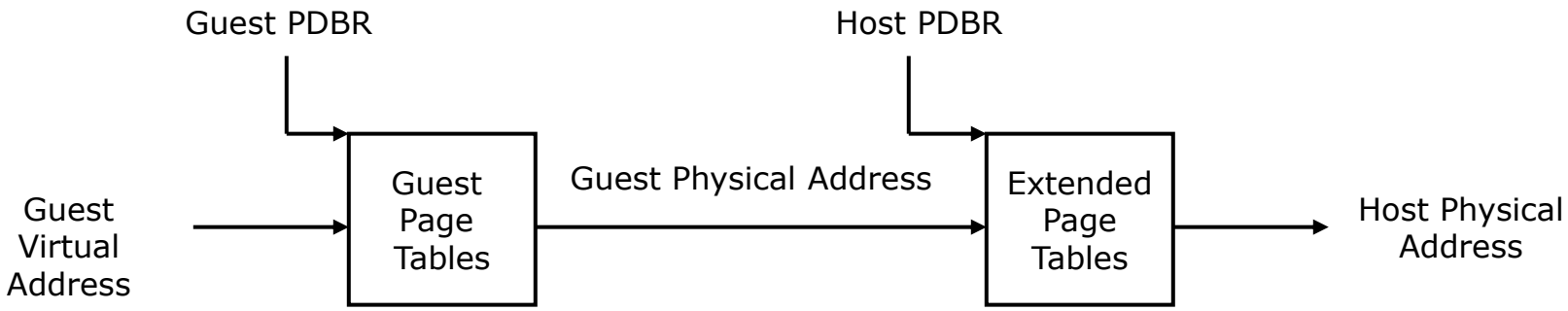


Extended Page Tables

- Both AMD and Intel have added some extensions to their hardware support to simplify the task of **virtualizing guest virtual memory**
- Specifically, they introduced the *extended page table*, that is available on the host but not on the hardware emulated for the VM
- The hardware implementing the host MMU is extended to have two PTBR pointers, one to the page table of the guest VM, implementing the G mapping, another one to the page table created by the VMM to implement the H mapping
- The extended hardware takes care of applying the composition of G mapping and H mapping, which are applied sequentially



Extended Page Tables



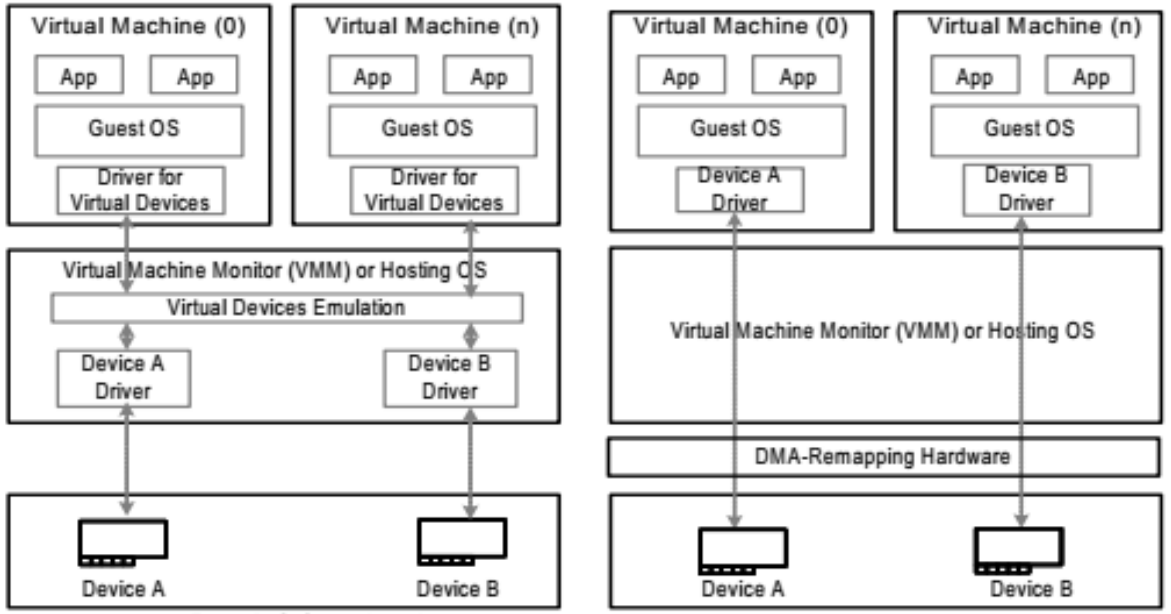
- Through this extension, there is no longer any need for VM exits, every time the guest VM modifies the page table
- This extension, however, increases the cost for address translation, which is more expensive, as more memory accesses must be performed by the hardware (for each translation 24 additional memory accesses are required)
- In order to reduce this overhead, the latest MMUs are usually equipped with a page table cache to reduce those accesses

Hardware passthrough

- I/O emulation poses the biggest performance challenge in virtual machines
- VMM has to participate in every interaction between the VM and the I/O peripherals (often emulated), thus resulting in several VM exits
- In order to overcome this issue, a solution is to give a VM direct and exclusive access to a peripheral
- This approach is called **passthrough** and maps the device directly to the guest VM without performing any translation
- This eliminates completely the need for VM exits, however, the hardware is not virtualized, and it must be dedicated to one VM and not shared
- In addition to this, the VM must deal with interacting directly with the hardware without additional abstraction layers
- *In order to implement this approach explicit hardware support is required*

Passed-through device

- In order to implement a passed-through device to a guest OS running into a VM, we need two main functionalities:
  - **Direct mapping** of the I/O (physical) **memory space** in the virtual physical memory of the VM, so the guest OS can write directly on the registries of the peripheral
  - **Direct assignment** of the **interrupts** linked with the peripheral, so the VM can handle the interrupts coming from the passed-through device, while the VMM handles all the others



I/O Mapping

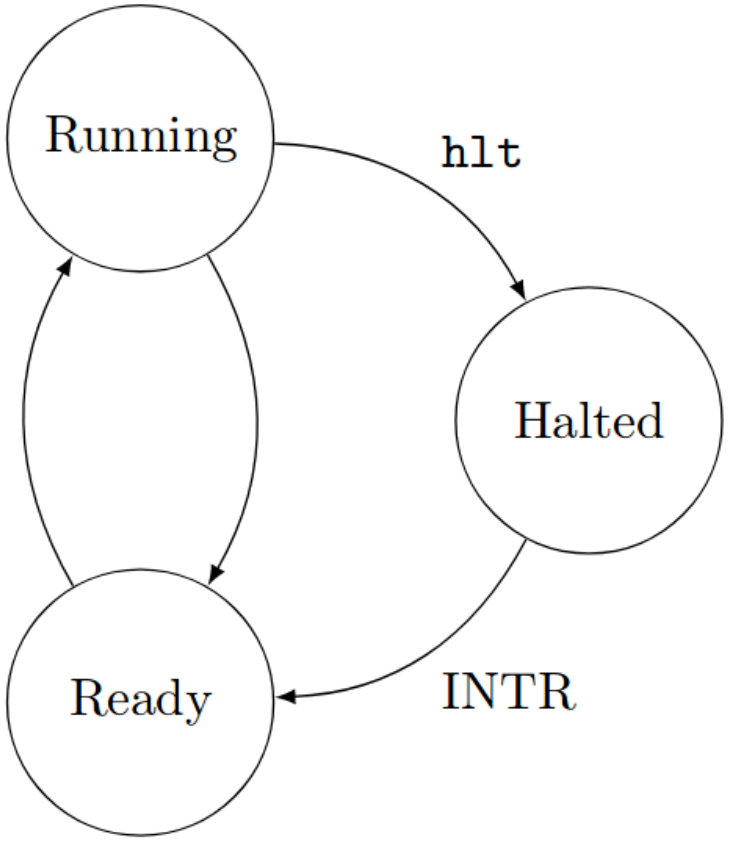
- For read/write to/from the registers of the passed-through device, we want the VM to be allowed to perform I/O read/write **without VMM intervention**
- For I/O reads and writes to other peripherals, we want the system to intercept VM operations, and give back control to VMM
- Intel VMX architecture includes in the VMCS an **I/O bitmap**, with one bit for each possible I/O address to specifically support hardware passthrough
- When a device passthrough is set for a VM, the I/O bitmap is modified in order to set all the bits corresponding to the addresses of the I/O space of the device, to grant direct access
- In non-root mode, the CPU first checks the bitmap when a read/write is performed in the I/O space: if the bit is set the instruction is completed, otherwise a VM exit is triggered to give back control to the VMM (e.g. to emulate hardware operations)

Interrupts

- Intel VMX extensions allow two options for the management of interrupts:
  1. the interrupt causes a VM exit, so the interrupt is managed by VMM
  2. the interrupt is managed directly by the VM without a VM exit
- For passed-through devices, the second option is adopted: the interrupt is handled directly by the guest VM that receives the interrupt and executes the handler
- In this case, the hardware is capable of directly looking at the Interruption Vector Table (IVT) of the guest OS and executing the corresponding handler of the guest OS
- The guest OS IVT is managed autonomously by the guest OS without VMM intervention
- That is implemented via the Interrupt Remapping Table
- The VM, however, might receive an interrupt while not running, to this aim the system needs to handle the interrupts coming from the passed-through device also when ***the VM is not running***

VM States

- A VM can be in ***running state***, when the VM CPU is actually using the host CPU, or in ***ready state***, when the VM CPU is stopped because the host CPU is currently being used by the VMM or some other VM
- A VM can switch between running and ready for VM exit or scheduled decision by the VMM scheduler
- When the guest software inside the VM executes the ***hlt*** instruction, the VMM typically intercepts the instruction and puts the VM into the ***halted state***
- A VM might resume from the halted state only if it receives an interrupt that is received to the VM CPU

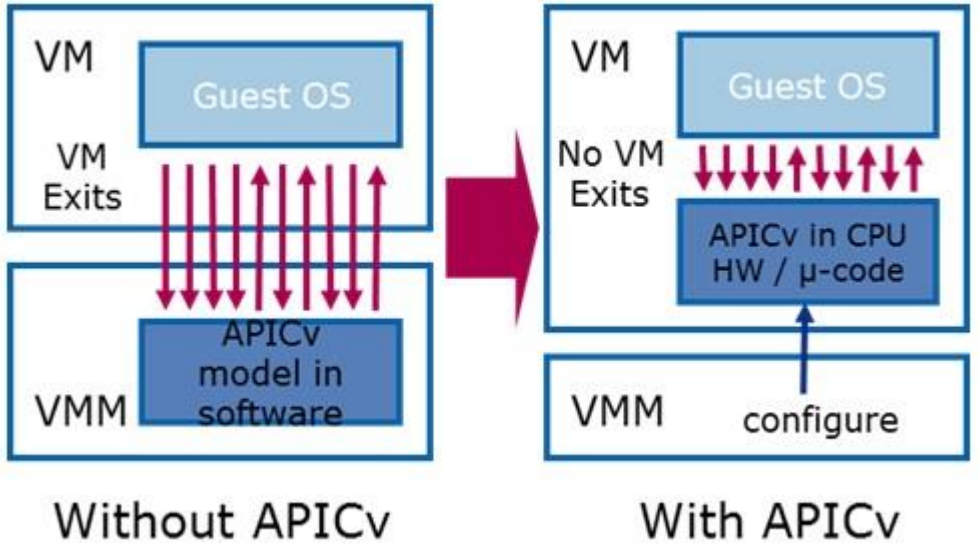


Interrupts management

- When a passed-through device is involved, the interrupts should be managed according to the VM state. Specifically, we have the following three possible cases:
  1. **VM running**: the interrupt should be handled directly by the processor, jumping to the guest OS interrupt handler, without VMM intervention
  2. **VM ready**: since the host CPU is currently running something else, e.g. another VM or VMM code, we want to store the interrupt request somewhere so that the VM CPU can handle it at the later time when the VM is scheduled for execution
  3. **VM halted**: the interrupt needs to be stored and the VM state needs to be changed to ready
- The **posted interrupt mechanism**, available in recent CPUs, allows the implementation of this mechanism

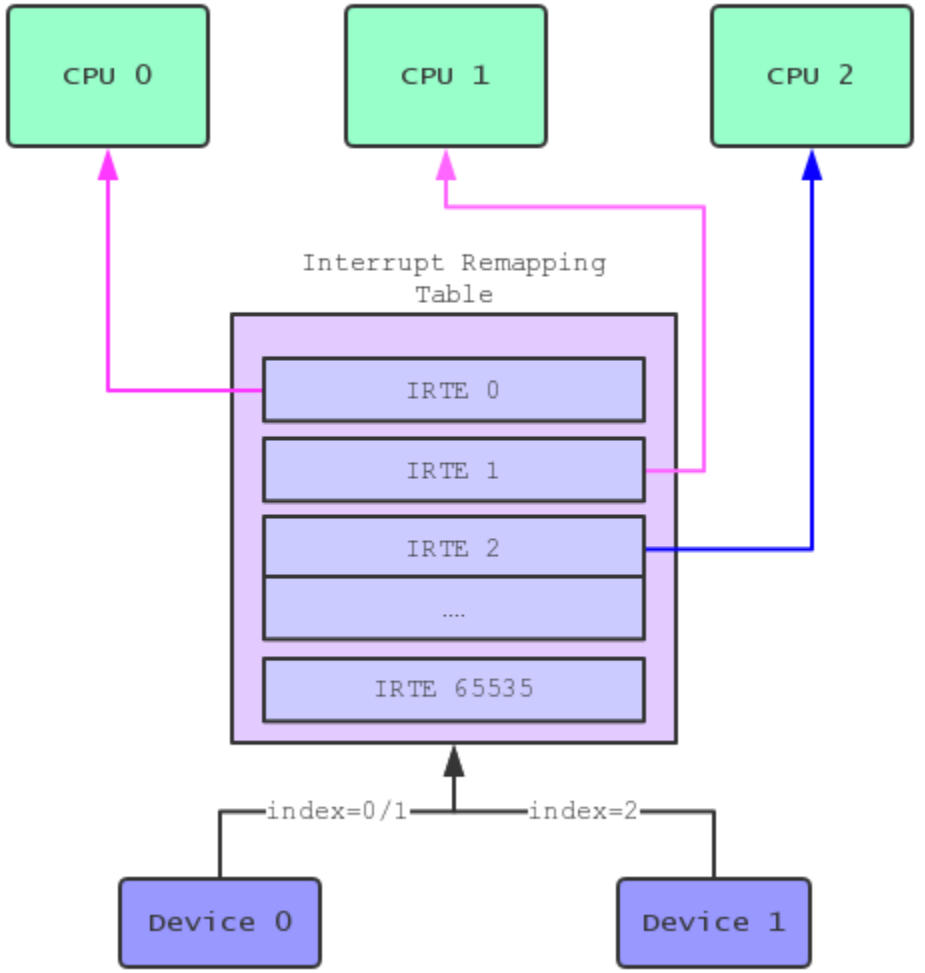
Posted Interrupts

- The posted interrupts mechanism allows to store interrupts signals (post) for later notification to the VM, without VMM intervention
- The mechanism exploits an additional data **for each VM**:
  - **The Posted Interrupt Descriptor** (PID)



Interrupt Remapping Table

- PID are used in the IRT
- Each entry in the IRT maps to an Interrupt Vector (if the interrupt must be handled by VMM) or a PID, which points to the actual interrupt vector if the VM is active
- Every time an interrupt is triggered, the IRT decides which Interrupt Vector or PID handles the interrupt



Posted Interrupt Descriptor

- The PID contains information on the status of the VM and stores the information on the interrupt notifications for asynchronous notification
- It contains all the data structures required to implement the posted interruption mechanism
- Specifically, it has the following fields:
  - A ***Posted Interrupt Request*** *(PIR)* structure, in which interrupts are posted
  - A ***Suppress Notification*** *(SN)* flag that determines whether the controller after posting the interrupt in the PIR must also notify the CPU (SN=0) or not (SN=1)
  - A ***Notification Vector*** (NV) that points to the actual interruption vector to notify interruptions to the VCPU of the VM

Interrupt delivery

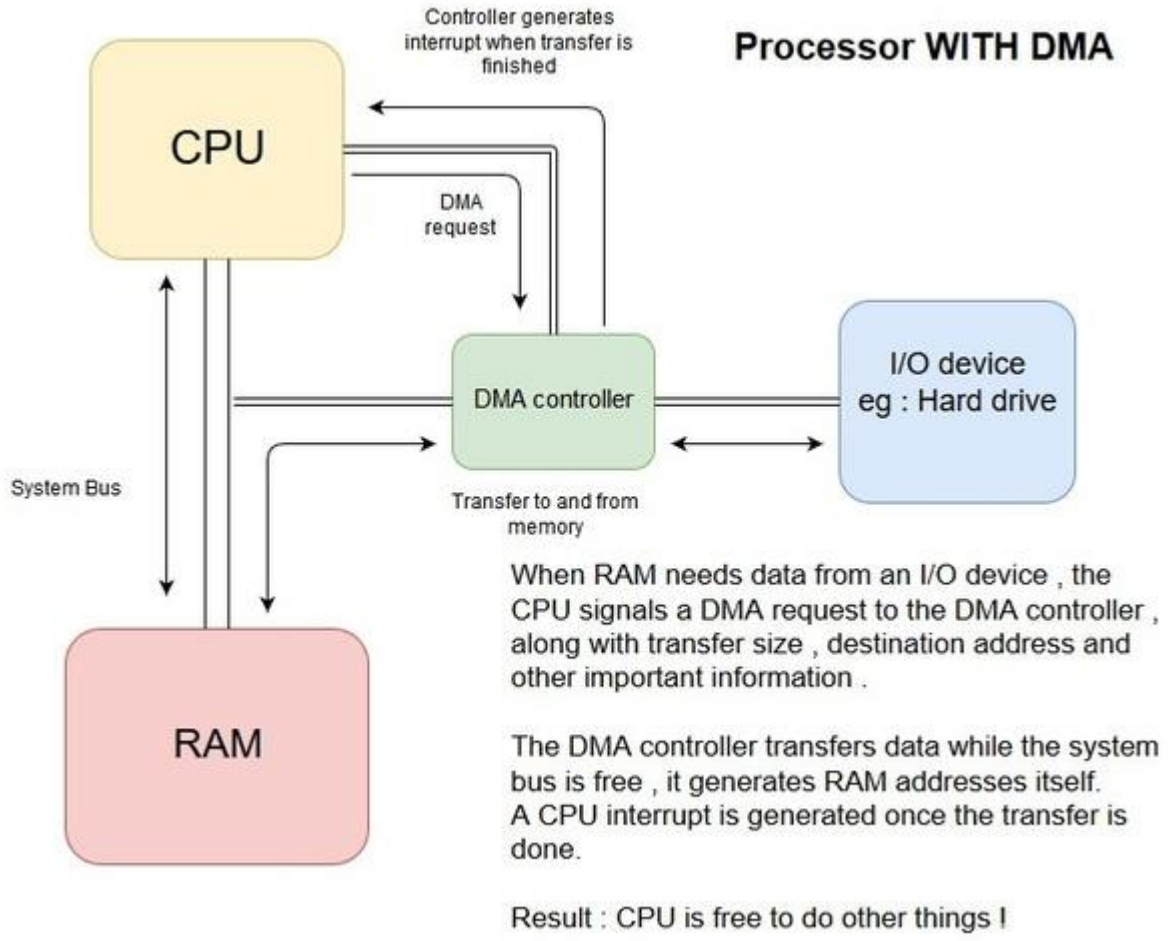
- When the interrupt controller needs to deliver the interrupt to the VM the corresponding entry in the IRT points to an PID instance
- In this case the following operations are performed:
  - It sets the proper bit in the PIR
  - If SN is 1 (e.g. because the VM is in ready state), it does nothing else
  - Otherwise:
    - If the VM is running it interrupts the processor by using the vector NV
    - If the VM is halted it triggers the awakening of the VM

PID update

- As the VM state changes, the VMM has to update the PID status accordingly:
  - When the VM goes into **running** state, VMM sets SN = 0 and NV to a vector named Active Notification Vector that points to the IDT of the VM
  - When the VM goes into **ready** state, VMM sets SN = 1
  - When the VM goes into **halted** state, VMM sets SN = 0 and NV to a special vector named Wakeup Notification Vector that triggers VM awakening
- *When the VMM changes the state of a VM to running, it must also look at the PIR, if there is any bit set, it must set accordingly the NV when entering the VM. This way the processor will process the interrupts that were posted while the VM was not running*

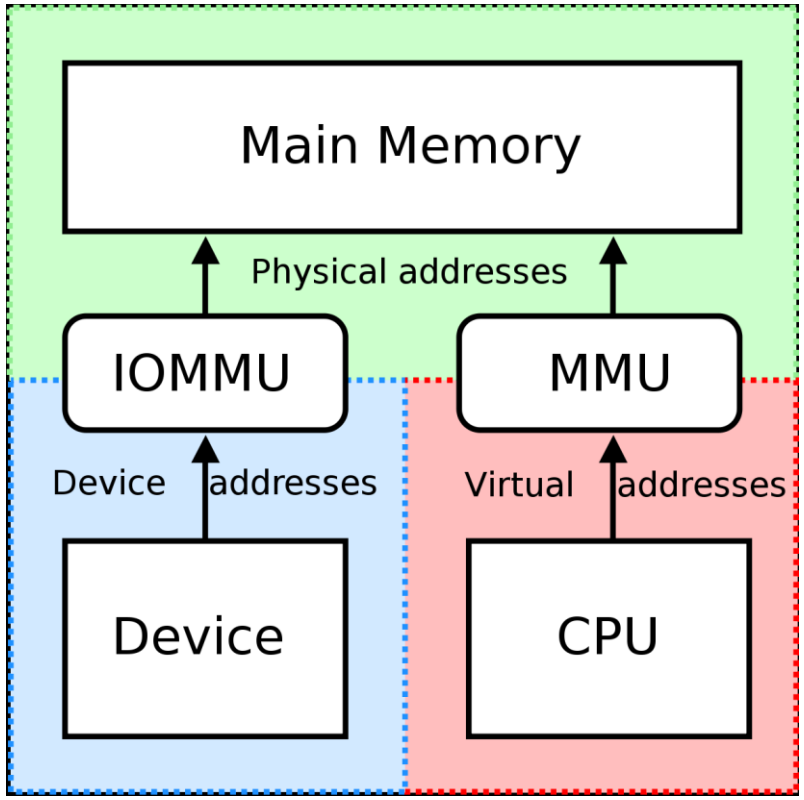
DMA devices

- DMA devices are I/O devices that reads data directly from RAM
- The CPU instructs the device by writing/reading on a small subset of registers, then it directly fetch the data from RAM
- DMA devices access directly the physical memory



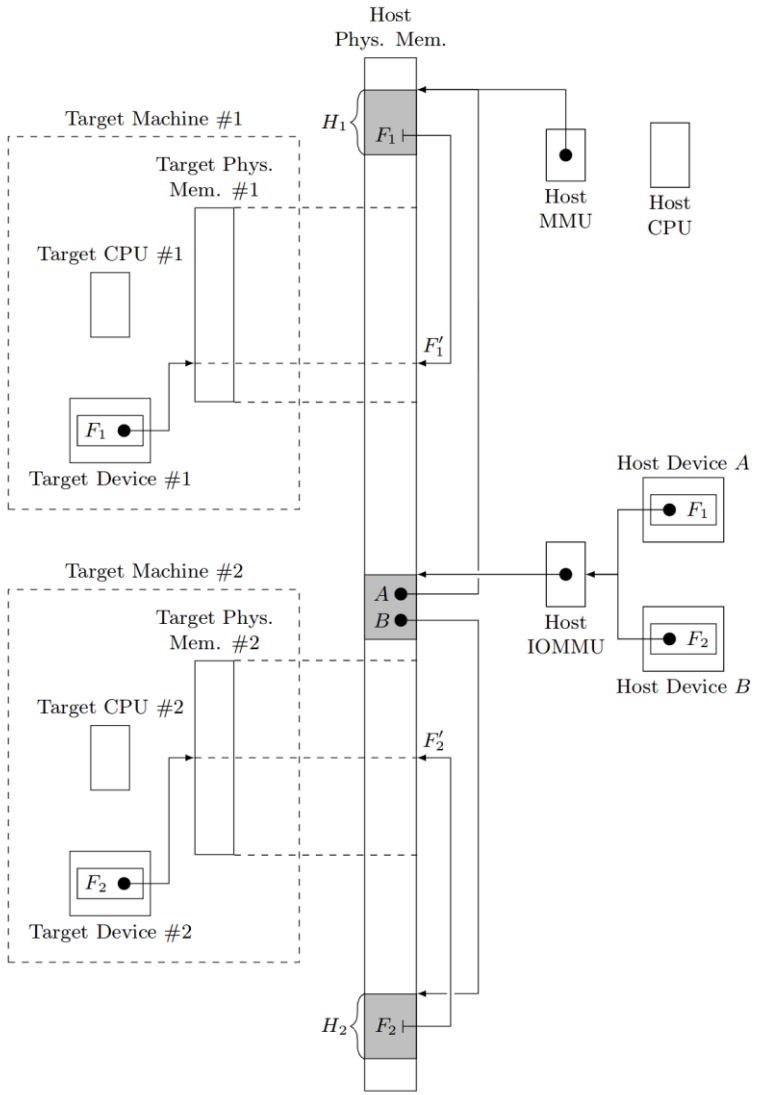
IOMMU

- In order to translate virtual addresses into physical addresses, DMA devices exploits a specific MMU circuit for I/O
- This IOMMU is responsible for translating virtual addresses into physical addresses for the I/O device to read/write from/to RAM
- The IOMMU is a general device that it was introduced into multiprogrammed systems that adopted virtual memory



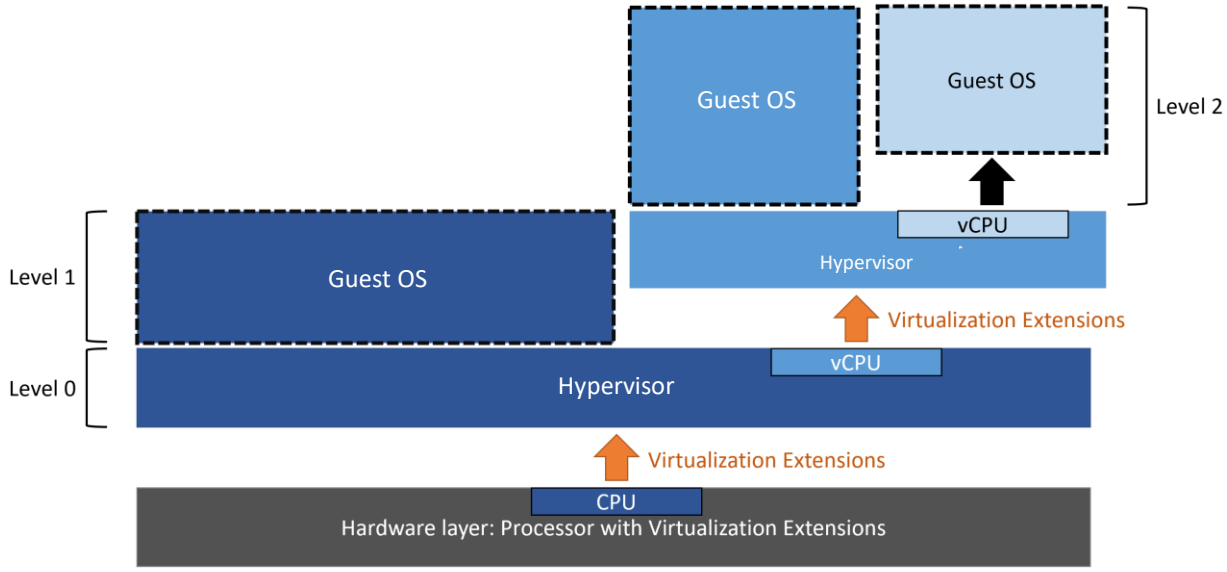
IOMMU and virtualization

- IOMMU can be used for virtualization already
- IOMMUs are also capable of triggering page faults interrupts, if they found pages that are not stored in ram



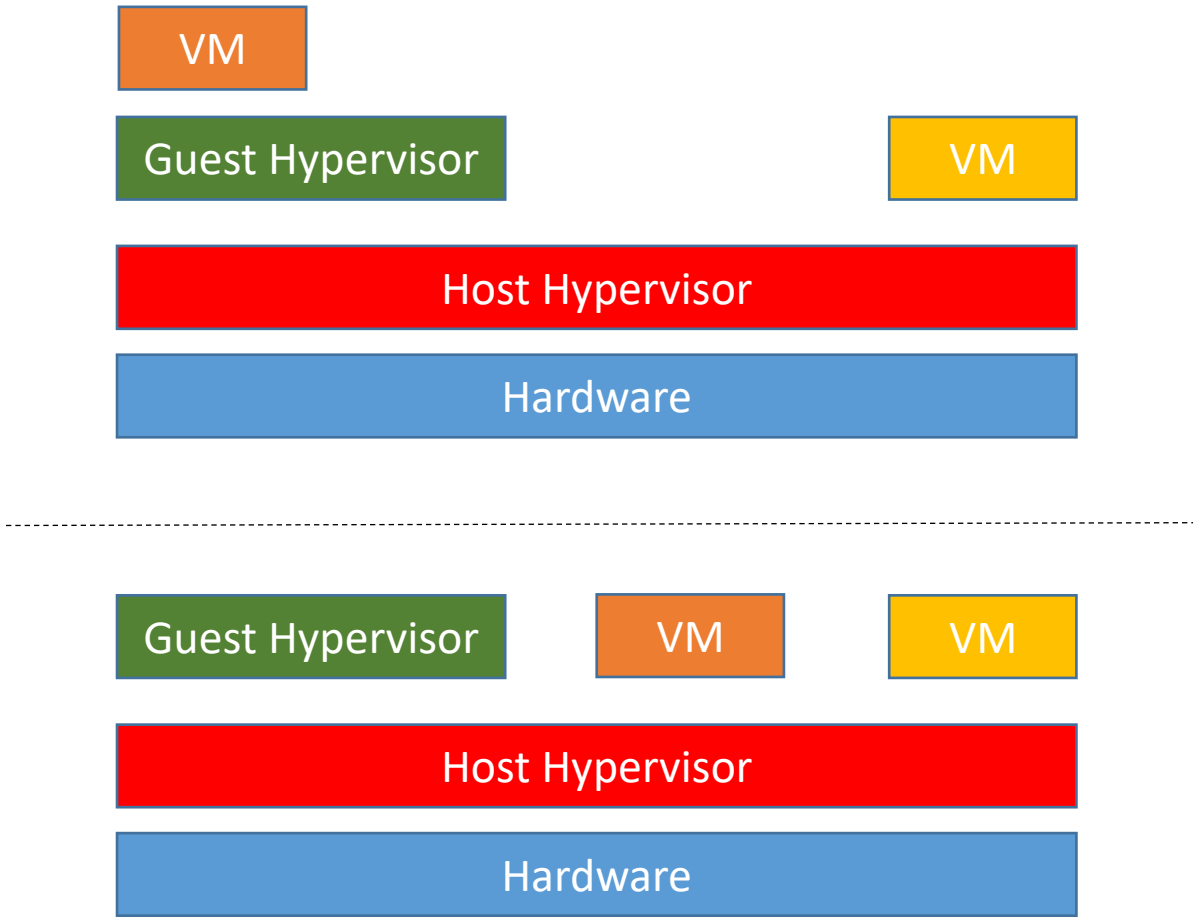
Nested Virtualization

- Nested virtualization describes a system in which the hypervisor runs inside a VM, which in turn is running inside a host hypervisor
- Nested virtualization is useful mainly for testing, although today is used also in solutions for production (it reduces the effort for upgrading the infrastructure)
- It allows cloud consumers to setup their own IaaS infrastructure on top of a set of VMs running on a IaaS infrastructure



Nested Virtualization Support

- Multi-level nested virtualization requires specific hardware support
- Intel and AMD only supports a **single virtualization level**
- Multiple virtualization levels can be multiplexed into a single virtualization level, i.e. the VMs running on top of the guest hypervisor are run as the other VMs running on top of the host hypervisor
- *Guest hypervisor access to hardware extensions for virtualization are emulated by the Host hypervisor*



KVM

- Kernel-based Virtual Machine (KVM) is a virtualization infrastructure of the Linux Kernel, added in 2007
- KVM allows to exploit hardware virtualization support to run VMs inside an execution environment nearly identical to the physical hardware
- KVM does not run as a normal program inside Linux, but it relies on the Linux kernel infrastructure to run
- KVM runs as a driver handling the new virtualization instructions exposed by the hardware
- KVM exposes a device /dev/kvm that is an interface allowing a user-space program to setup and configure the environment for a new VM
- A modified version of QEMU, kvm-userspace, exploits the KVM APIs to for running the guest VMs