

Virtualization Technologies

Multiprogramming recap and Full Virtualization Techniques (without hardware support)

Reference:

- Material available on the course website

Virtualization Requirements

- **Equivalence:** an OS running in a VM under the hypervisor should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly NON HA' ESSERE LOCO DENTRO LA MACCHINA
MA GENERALIST.
- **Resource control:** the hypervisor must be in complete control of the physical resources and the OS running in the VM must have complete control of the virtualized resource OS DEVE AVERE L'IMPRESSIONE DI CONTROLLARE IL HARDWARE
HYPERVISOR DEVE AVERE IL COMPLETO CONTROLLO DEL HARDWARE
- **Efficiency:** a statistically dominant fraction of machine instructions must be executed without Hypervisor intervention in order to ensure efficiency → NON ESSERE DETTO SOLO SE
- Remember: *the instruction set used by the virtual system and the actual hardware system is same* EVALUATION VS SIMULATION

Multiprogramming

QUALI SONO
A VIRTUALE

TUTTO QUESTO PER ESSEMPI
DI PROCEDURE CONTEMPORANEE

- Such requirements are similar to the ones for multiprogramming
- Through multiprogramming we emulate a machine with more processors (and other peripherals) than we have in the real hardware
- Each process runs on its own **virtual processor (VCPU)**
- Each virtual processor is reserved for the execution of a process
- Virtual processors are multiplexed over time on the host machine physical processors for the execution of a specific program
- Requirement: the process must have the impression that it has complete control on the processor, and it is the only process executed on it

Memory

Job 1
Ready

Job 2
Ready

Job 3
Ready

Job 4
Ready

Memory

Job 2
Ready

Job 3
Ready

Job 4
Ready

CPU



FISICO
(HARDWARE)

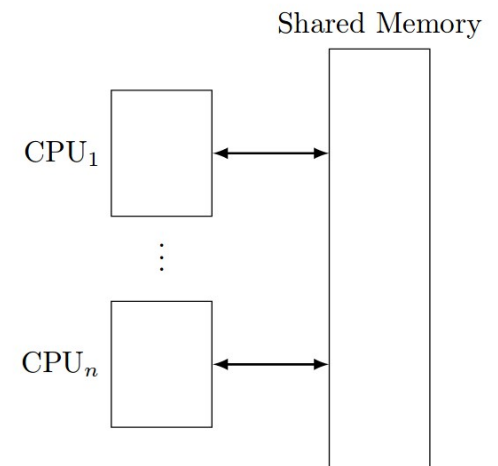
CPU



VCPU Execution

CONTEXT SWITCH : OPERAZIONE CHE UN PROCESSORE FISICO
Cambia per passare ad eseguire
un altro processo, salvando lo
stato corrente

- The OS implements multiprogramming by creating a virtual representation of the processor that contains a copy of the *state of the processor* (its registers)
- A machine with one physical processor emulates only one virtual processor at a time, this is obtained by loading the registers of the host processor with the values stored in the virtual processor representation and then letting the host processor continuing the execution
- At some time the execution is paused and the virtual processor data structures are updated with the current values of the host registers, then a new virtual processor can be selected for execution and so on



We assume that the OS adopts a shared memory model, i.e. all the Virtual CPUs share the same memory

CAMBIO
LOCALE:

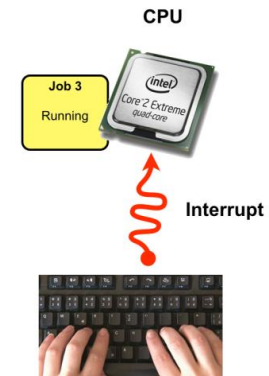
- 1) THUS
- 2) SCHEDULING
- 3) INTERRUPT → PERIFERICHE ESTERNE (HARDWARE)

Context Switch

- The operation of switching from one process to another (and the corresponding VCPU) is called *context switch*

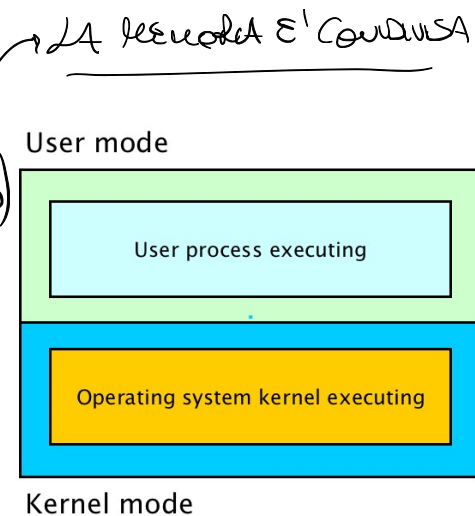
- ¹It is determined by a timer (the scheduler of the OS) or an ²interrupt (as a process that was waiting for a hardware input becomes ready)
- Context switch is typically implemented in software within the OS, which implements operations of loading and unloading the host processor registers in memory

Memory



Multiprogramming support

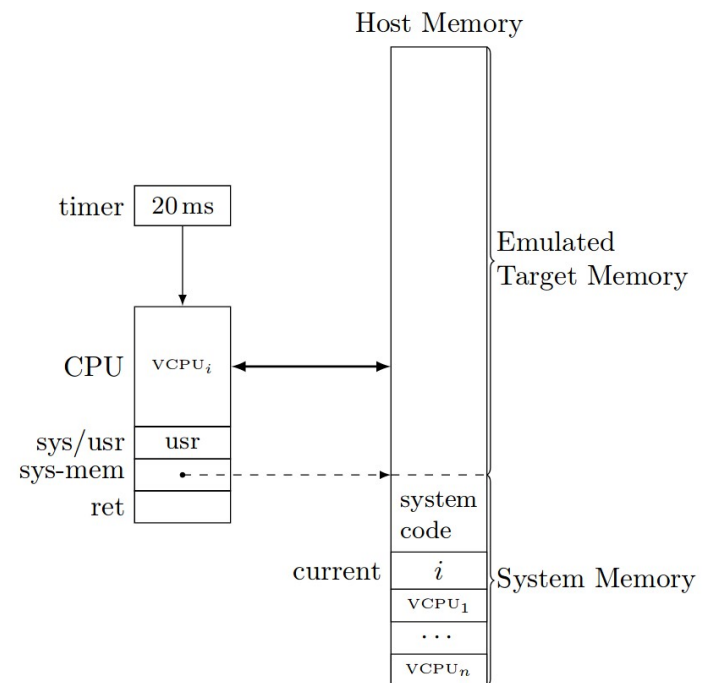
- The implementation of a complete multiprogramming environment, however, still requires some hardware support:
 - An *interruption*, e.g. a periodic timer, to trigger a periodic context switch (that is easy!)
 - A *memory protection mechanism* to isolate the VCPU representations in memory to prevent a process accessing the registry values of other VCPUs (HW)
- Modern host processors have support for different privilege levels
- They include at least two different privilege levels: a *system (or kernel) level* that it has access to all the memory, a *user level* that has access to only a subset that does not include VCPU data structures
- The code of the OS is executed in system level while the code of user processes is executed in user level, access to privileged memory in a code executed in user level is denied
- The timer interrupts that invoke the execution of the context switch functions runs a specific OS routine that is executed in system level so the context switch can be implemented properly



Multiprogramming support

- Memory protection support includes:
 - A *usr/sys registry*, a flag that specifies if the CPU is currently in system or user mode
 - A *sys-mem registry* that contains the starting address of the privileged memory where VCPUs data structures are stored
 - A *ret* register that stores a memory address to be used when returning from an interruption
- When the timer for context switch is triggered, the OS performs the following operations:
 - Saves the state of the host CPU on the VCPU representation
 - Selects a new VCPU to run and loads its state into the registers of the host CPU
 - Executes a special instruction to jump to the address stored in *ret* and return in user mode
- Recent CPUs includes also a set of instructions to implement operations (1) and (2) directly in hardware so their execution time is minimal

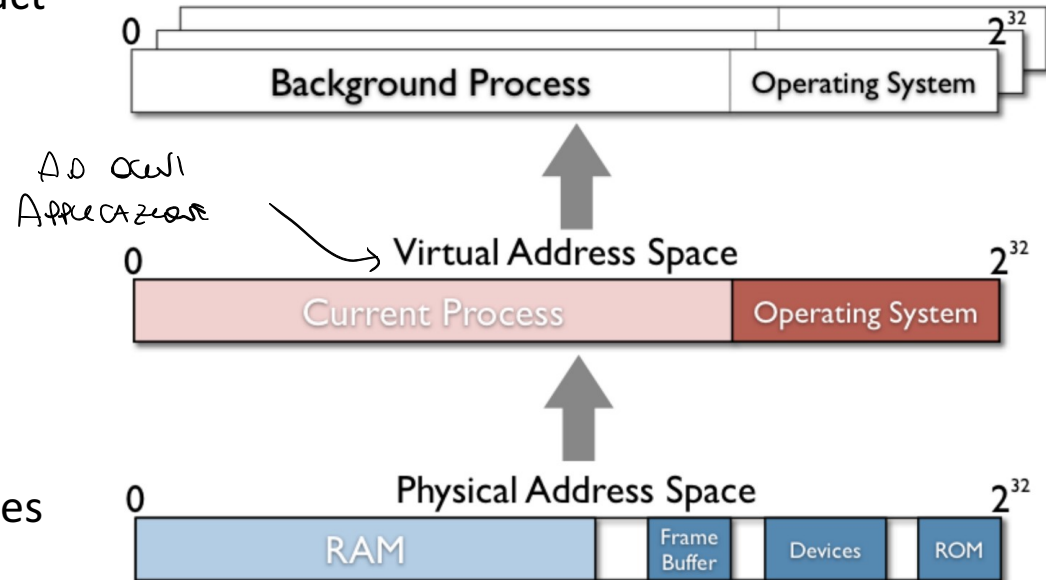
LOAD/SAVE



Virtual Memory

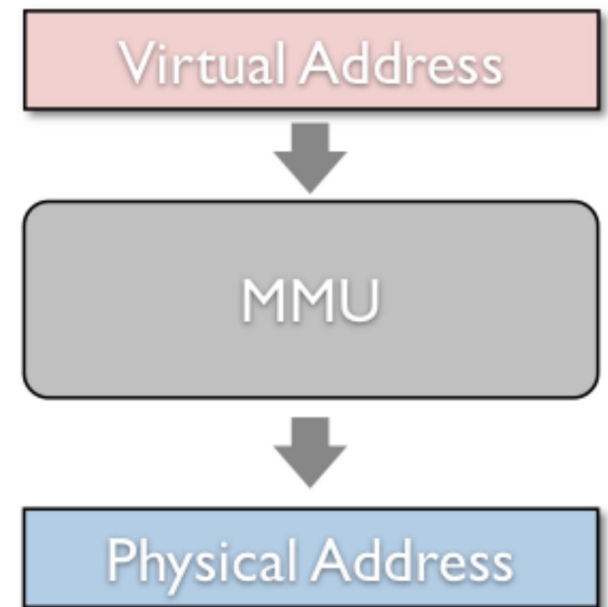
- Virtual Memory is a mechanism introduced to allow processes to abstract from the real memory available on a system and use virtual address space
- A Virtual Address Space is created and assigned to each application
- This allows each process to think that they can access a contiguous address space in memory, regardless of the physical allocation in RAM
- Data can be stored in RAM in not contiguous data segments in an independent manner from the addresses used by the application
- The result is that each process has the impression that can have access to the whole physical memory

Donc l'impression à l'application est l'illusion d'avoir un accès à la RAM A COMPTER DES ADRESSES



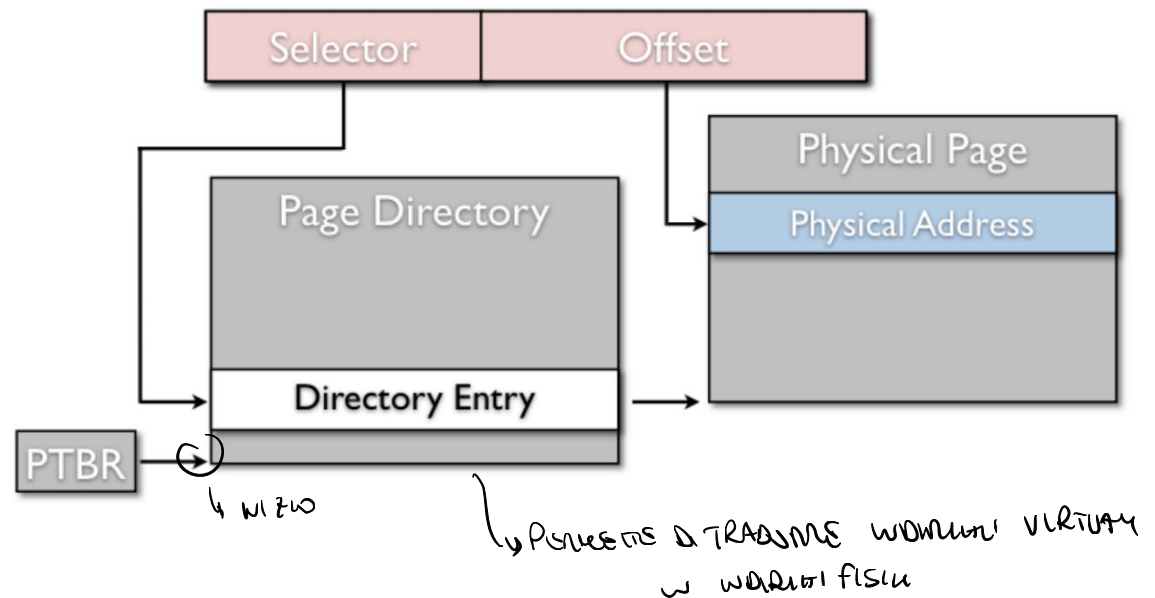
Memory Management Unit

- Virtual memory is implemented through two mechanisms:
 - Address translation
 - Virtual address spaces management
- Virtual address space management is performed by the OS, that takes care of managing the virtual address spaces created for each application
- The different virtual address spaces are eventually mapped on the physical memory that is shared among all the processes
- Address translation (from virtual to physical) is performed on the CPU by a specific hardware element called Memory Management Unit or MMU that takes care of translating the virtual address in the corresponding physical address based on a set



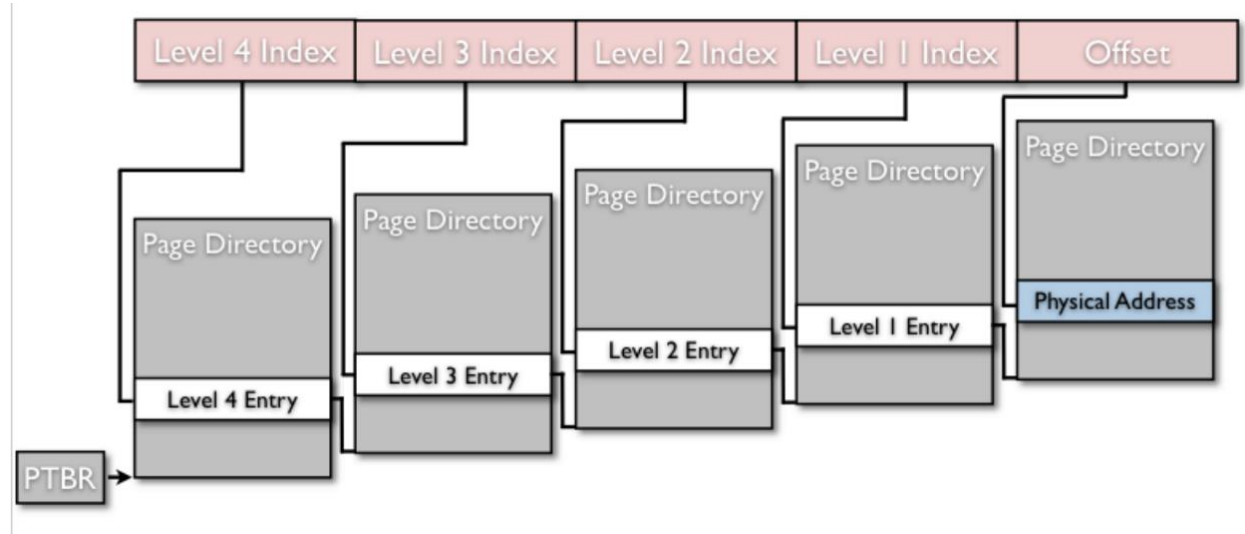
Page Table

- MMU manages the address translation according to a table named Page Table
- The CPU has a registry the PTBR (Page Table Base Register) that points to the physical address of the first byte of the page table
- The memory is organized into portions of equal size (pages), e.g. 4 KB
- The page table is organized into page directories, each one containing the information to translate a portion of the virtual address into the physical address
- The page table is managed by the OS



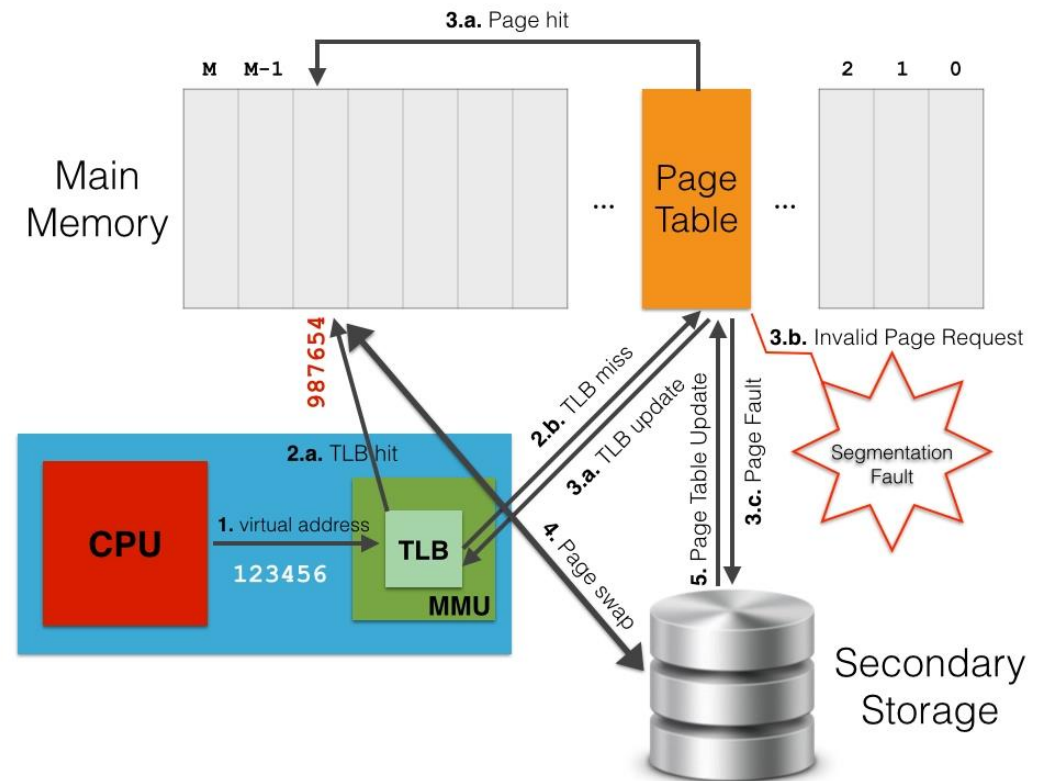
Multi-level Page Tables

- The page table is organized into a multi-level structure, multiple directories have to be accessed to translate the virtual address into a physical address
- The configuration depends on the hardware architecture and on the OS
- Hardware support is provided to manage the page table



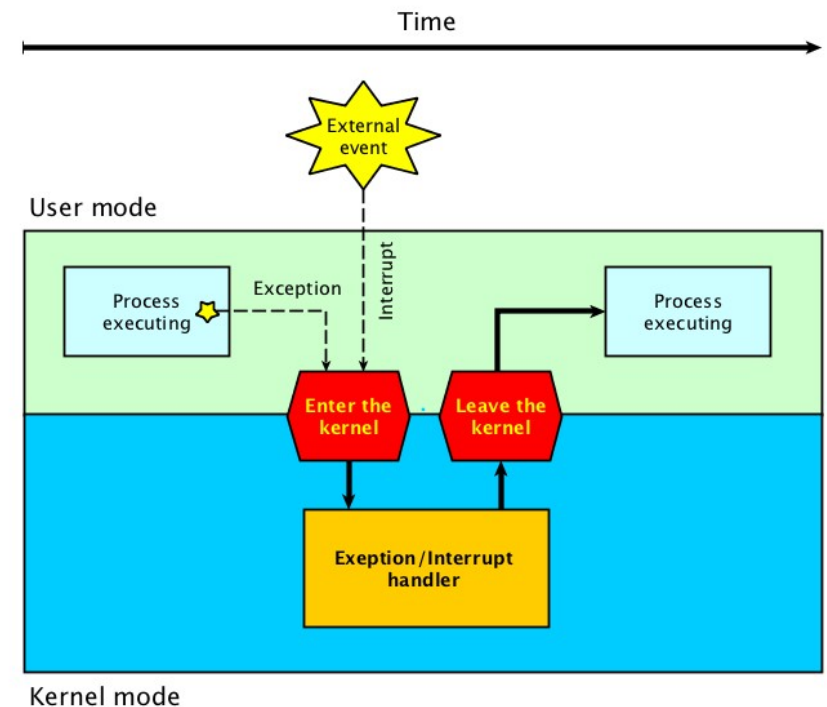
Segmentation

- The space of the virtual address can even exceed the actual capacity of main memory, using a secondary storage, e.g. a hard disk
- In this case, some pages can be stored outside RAM, when not used
- Every time a program tries to access a page stored in a secondary storage, a **page fault** is triggered and the OS is forced to retrieve the page and move it to RAM
- As a page is moved in RAM the page table is updated by the OS



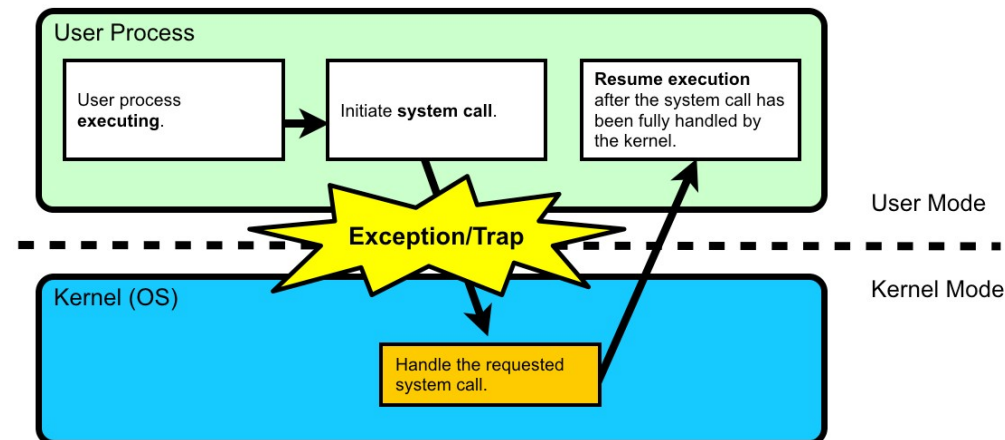
Interrupt/Exceptions

- Interrupts and exceptions are used to notify the system of events that needs immediate attention during program execution (e.g. a page fault)
- They alter the normal execution of the program triggering the execution of a function in kernel space
- When an exception or interrupt occurs, the transition from user mode to kernel mode is performed and a specific function of the OS associated with the event associated with the exception or interrupt is run. When the exception or interrupt has been handled, the execution resumes in user space



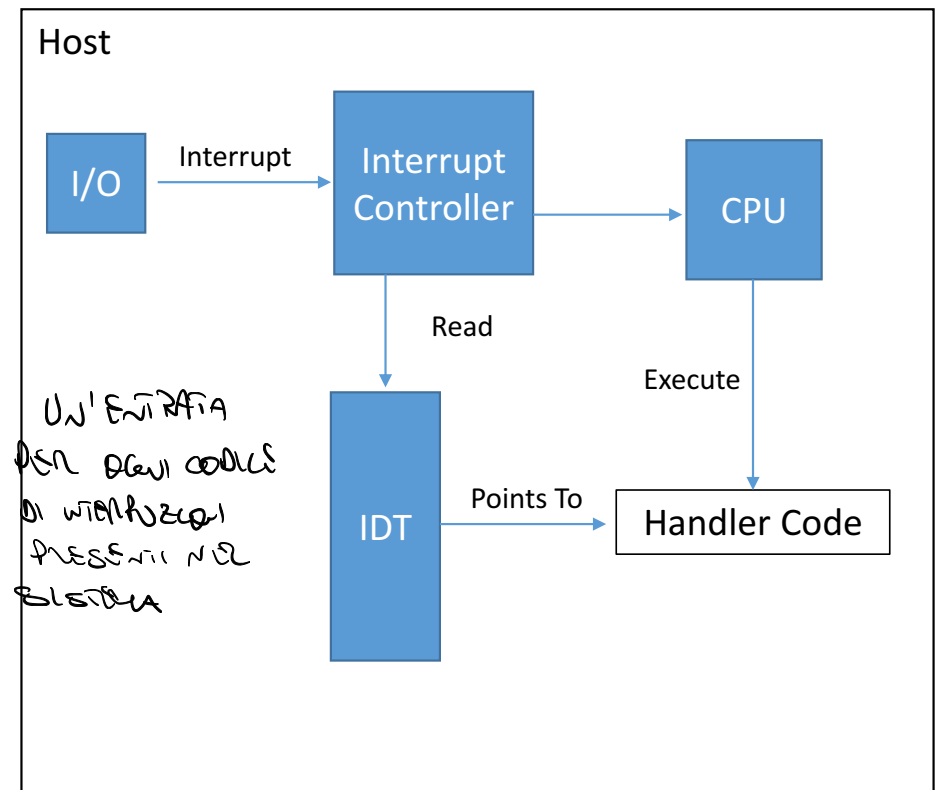
Interrupt/Exceptions

- Exceptions are internal and synchronous, they are used to handle internal program errors (e.g. division by zero, bad address, page fault)
- Another name for exception is trap. A trap (or exception) is a software generated interrupt. A trap is also used as system call and can be invoked in user space from programs.
- Interrupts are used to notify the CPU of external events
- Interrupts are generated by hardware devices outside the CPU at arbitrary times (e.g. a key pressed in the keyboard)



Interrupt descriptor table (IDT)

- The Interrupt Descriptor Table (ID~~V~~), also named Interrupt Vector (IV), is a table used by the processor to link interrupts and exception with handlers
- Each handler is a function in the kernel that performs operations linked with the interrupt
- The table is populated by the operating system

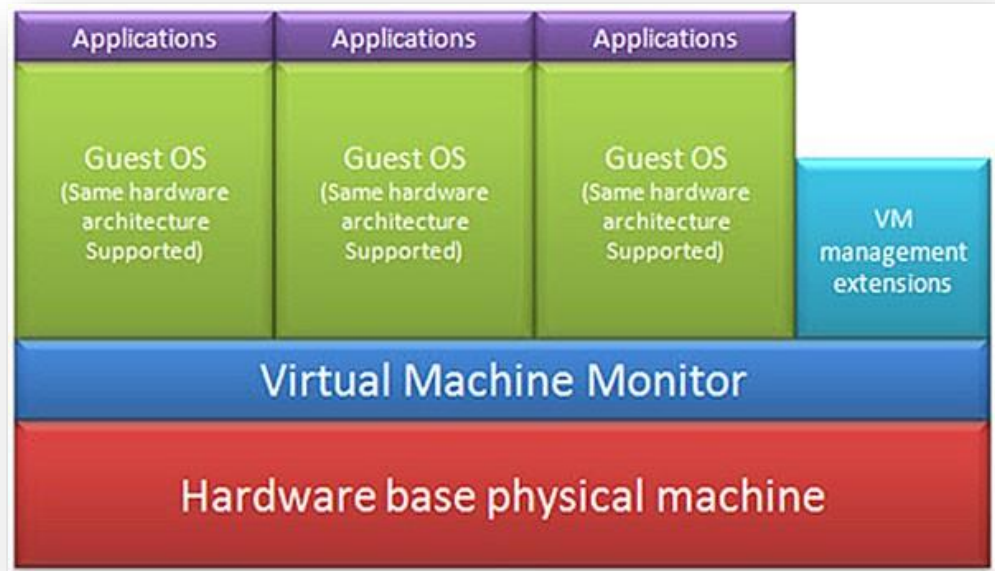


Virtualization – Full Virtualization

- Multiprogramming already includes some virtualization techniques that could be exploited to implement System Level virtualization
- Multiprogramming allows each process to have the impression that they have complete control on the CPU and RAM
- In a similar manner, the Hypervisor or Virtual Machine Monitor (VMM) must give VMs the impression they have complete control of the physical hardware, i.e. full processors, memory and I/O peripherals, to achieve system level virtualization
- The role of the VMM is similar to the role played by the OS Kernel, its functionalities, however, are far more extended than just supporting multiprogramming, due to the complexity of hiding the virtual environment to VM, a VM in fact is a multiprogrammed system itself and hosts an OS and not just a program

Hypervisor

- The idea is to implement the hypervisor to create a virtual representation of the system by reusing the hardware mechanisms that are already available
- If the target and host architectures are the same this can be implemented by minimizing the adoption of emulation, which introduces a significant overhead, due to the need for binary translation
- For the majority of time, the code of the OS/Software running in a VM can be executed directly on the hardware without the need for translation in order to reduce the penalty for the code that is executed into the VM



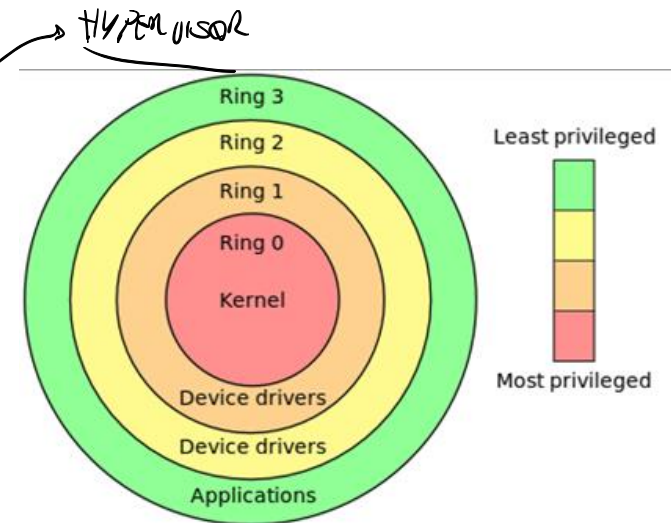
Virtualizing CPU

- The VMM adopts the same techniques adopted for creating VCPUs in a multiprogramming environment:
 - The VMM code is executed in system/kernel space, the guest OS code is executed in user space
 - The VMM loads the state of a VCPU into the host processor then it lets the host CPU to run the target code as it is, until the CPU finds an instruction that cannot be executed directly (we will see which instructions in a bit why)
 - As the context switch occurs the VMM regains control and *emulate the target instruction* that couldn't be executed by the guest OS in software and then re-load the virtual CPU into the host CPU to continue the execution
 - If multiple VMs are executed on the same physical host, the VMM could schedule a timer to trigger a context switch, to ensure fairness among different VMs running on the same hardware
 - In this case the VMM selects another VMs for execution whose VCPU status is loaded in the host CPU

Guest OS Execution

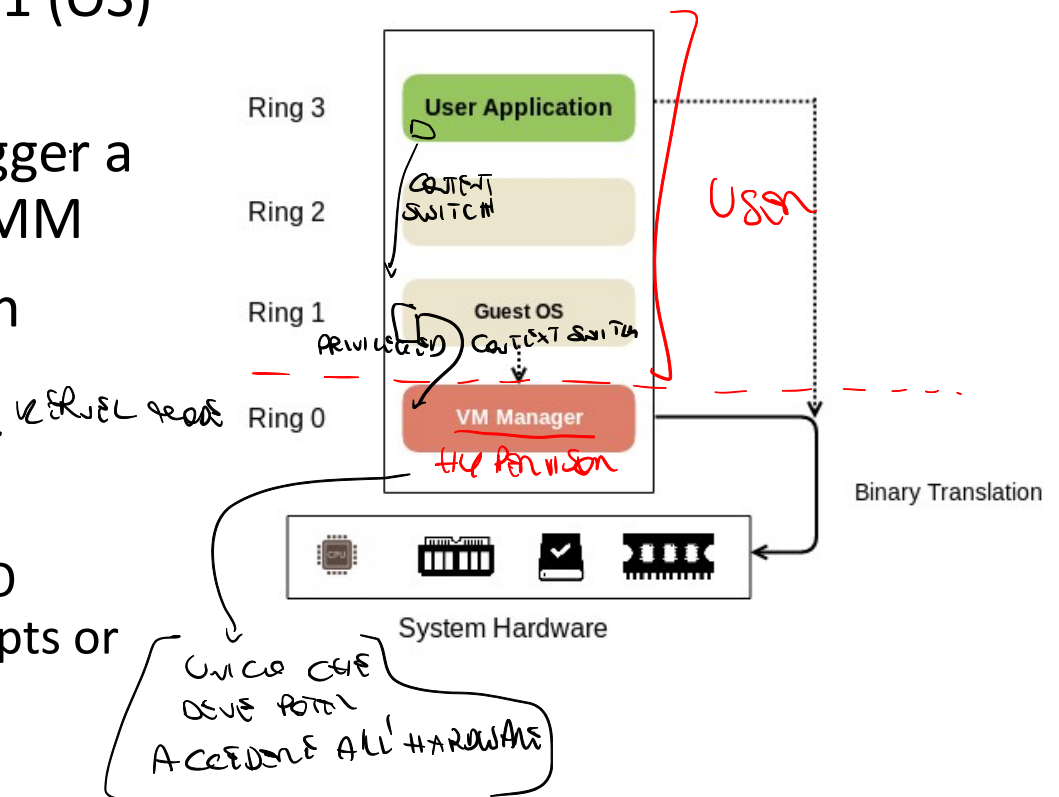
VMM = VIRTUAL MACHINE MONITOR

- The VMM has a very similar role as the OS kernel for multiprogramming, its role, however, it is different:
 - The Guest VM code would like to have complete control of the host hardware, and execute instructions not only at the user level but also at the system level
 - Therefore, VMM must emulate the entire processor, not only the user level functions but all the functionalities used at system level like multi-programming support, in order to support of the Guest OS
 - The code in execution on the Guest OS should have access to privileged registers and instructions
- In a multiprogrammed environment, every time a privileged instruction is executed the userspace process is killed by the kernel, we don't want the VMM to kill the VM but to manage the exception instead



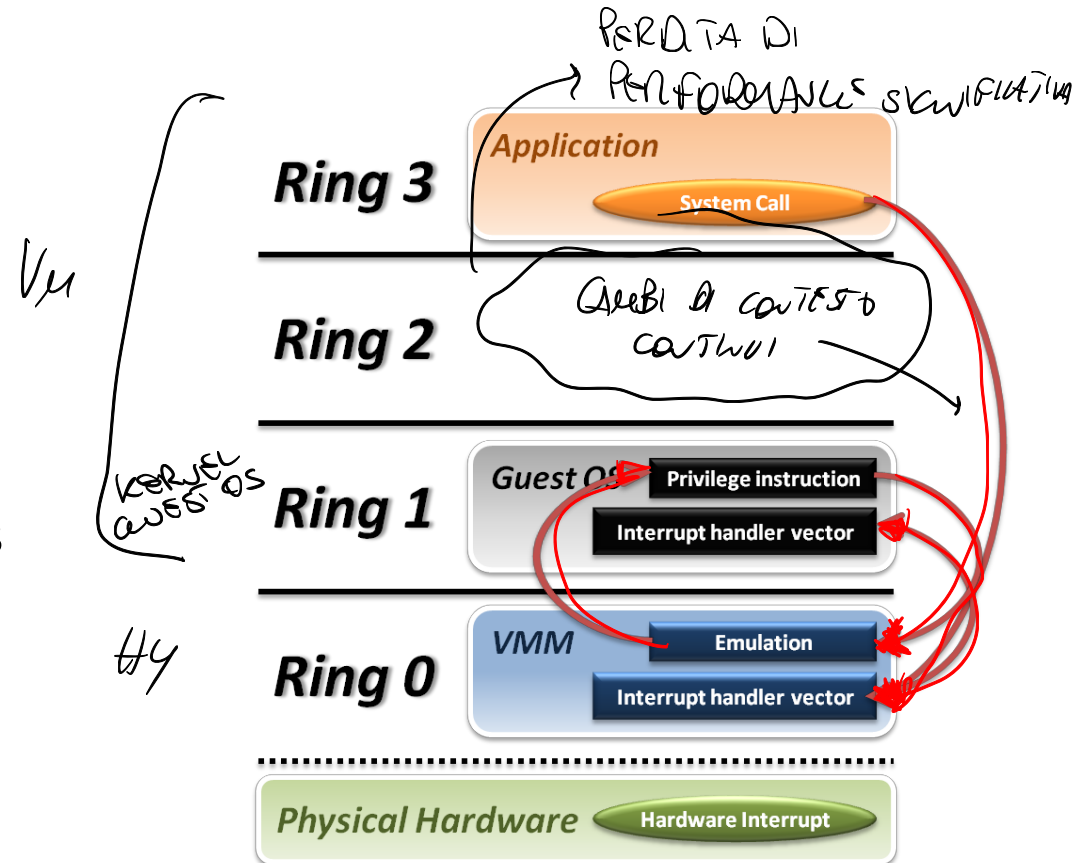
Trap and Emulate Virtualization Model

- Guest system (both OS and application code) is executed in user-space: Ring 1 (OS) Ring 3 (User app)
- *VMM exploits exceptions/traps* to trigger a context switch from the VM to the VMM
- Since the guest OS code is executed in userspace, every time *a privileged instruction is executed an exception is raised*
 - For instance, Instructions that access I/O devices, instruction related with interrupts or instruction that manipulate the MMU



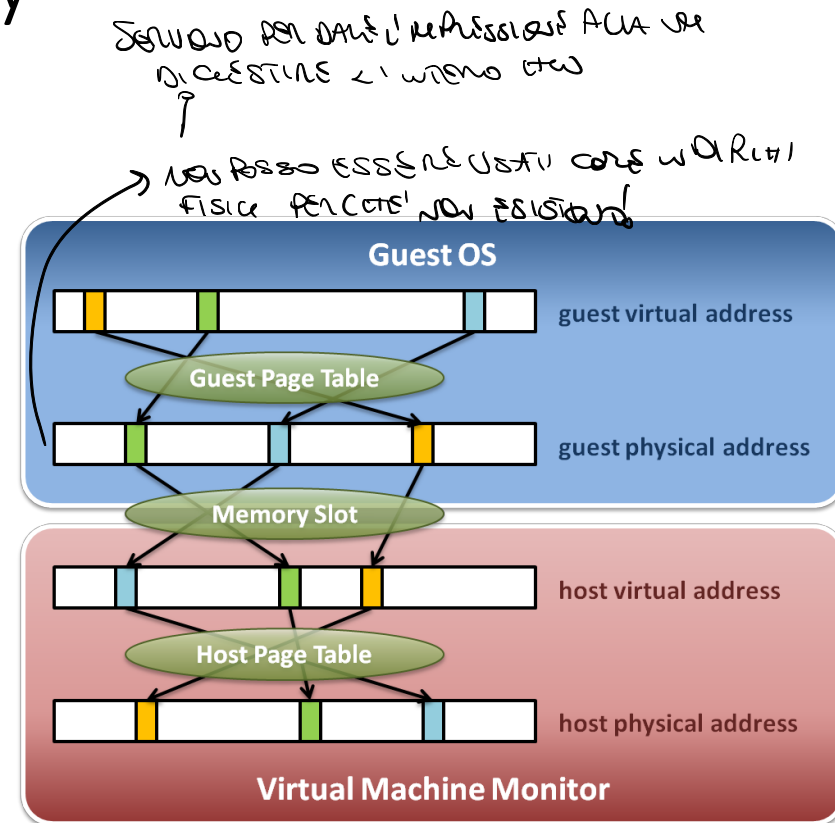
Trap and Emulate Virtualization Model

- The VMM (the trap code) then employs binary translation to execute the privileged instructions of the guest OS, emulating their behavior
- After the execution of the privileged instruction, the control is given back to the guest OS code
- If the set of privileged instructions is **limited** the performance are not affected significantly



Virtualizing Physical Memory

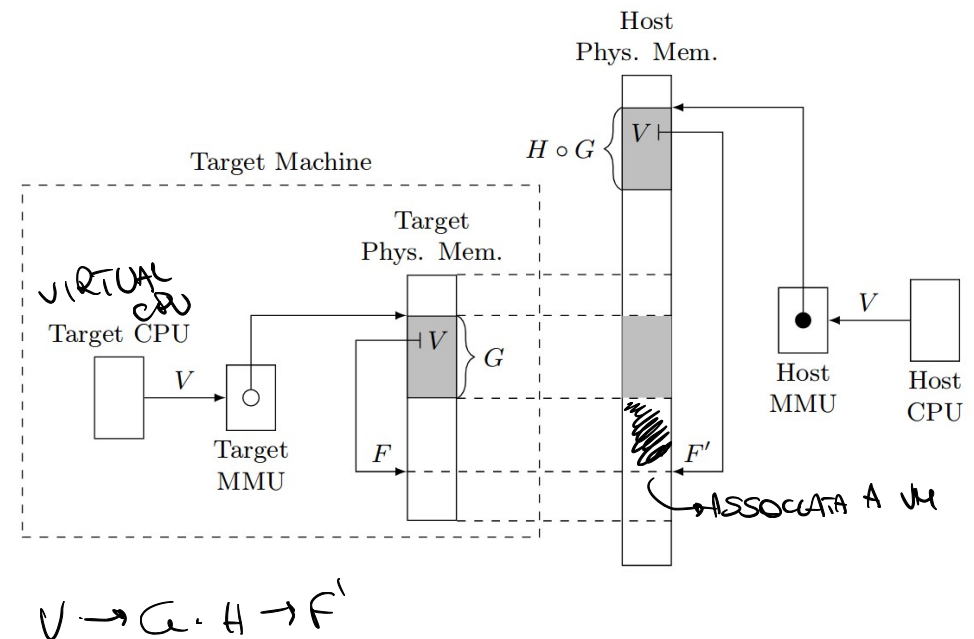
- The physical memory of the guest VM is typically implemented using a subset of the host physical memory (in a similar manner it is performed with multiprogramming)
- A part of the overall physical memory will be reserved for VMM execution
- The guest VM must have only access to the portion assigned to it and it must think that its memory starts from the physical address 0
- In order to map a portion of the physical memory to the host physical memory, the MMU unit of the host CPU is used
- The MMU of the host CPU is configured with *the address range in the physical memory assigned to each VM*, as the instructions of the VM are executed the addresses are translated
- If a pagefault occurs, the VMM takes care of retrieving the page and move it to memory
- Some operations from the guest OS are still trapped, e.g. operations that modify the page table so the proper translation is configured by the VMM



Virtual MMU

$G \rightarrow$ FUNCTION ONE TRADUCE
 $V \rightarrow f \rightarrow f'$
 =
 QUEST PHYSICAL
 ADDRESS

- The guest OS should be able to prepare and use its own translations from virtual address to the guest physical address (what the VM think is a real address) by creating and managing its own page table
- The guest OS should be able to define a G function, mapping the virtual address of the guest OS to the *guest* physical addresses, i.e. to map a virtual address V to a guest physical address F
- Considering that the guest physical memory is mapped in a portion of the physical memory of the host (F' in this case), the VMM is responsible to create and maintain a H function, which maps the guest physical addresses F to the host physical addresses F'
- To this aim the system needs a **Virtual MMU** that translates guest virtual addresses to host physical addresses by implementing the $G \cdot H$ mapping in a transparent manner (from the VM point of view)



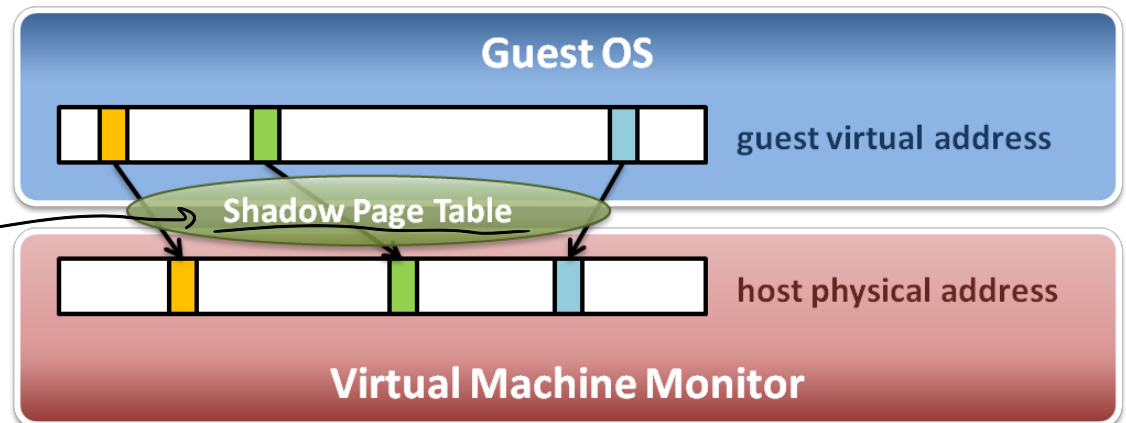
Brute Force Method

- The page table is modified in order to add an additional level (the shadow page tables) that implements the H function, i.e. the translation from the guest physical address to the host physical address

$$\begin{array}{ccc} V & \rightarrow & F' \\ || & & \\ G & \rightarrow & H \end{array}$$

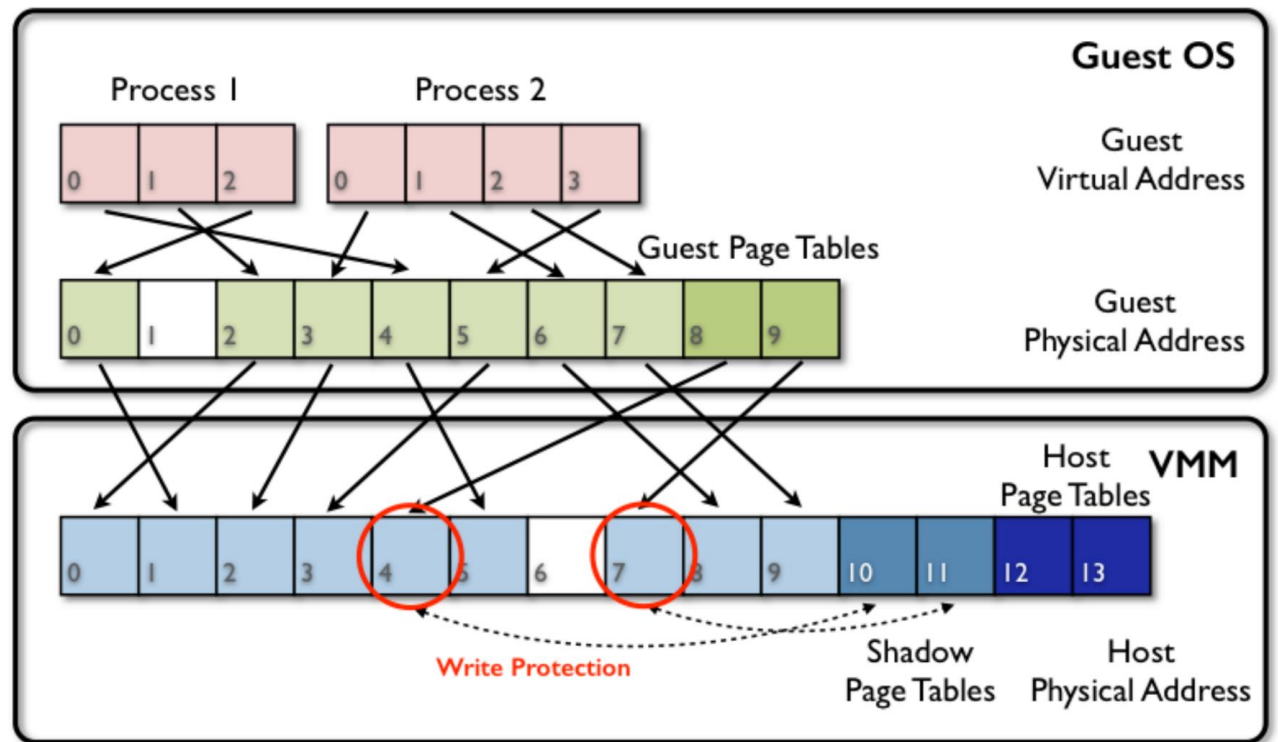
SERIAL FAVORITE CE PAGE TABLE
SOO ORGANIZE COORDINATION

CHES DEFINITIVE
DES PAGE TABLE



Shadow tables

- Shadow tables are set as write-protect, so any possible action to modify them causes a VM exit
- This method has a significant overhead as it requires several context switches and a continuous address translation by the VMM



Brute Force Method

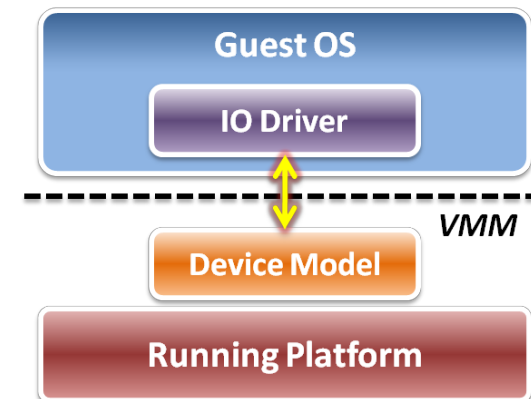
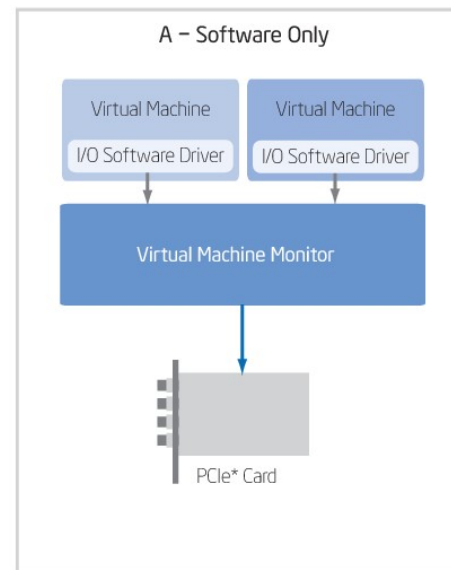
- In order to introduce this additional level and keep it updated, the VMM must trap all the possible actions related with the MMU and the page table, specifically:
 - Changing the PTBR (e.g. because the guest VM wants to replace the table completely)
 - Changing the page table entries in some directory (e.g. because the guest VM wants to change the mapping of some areas)
- Every time a trap intercepts a change in the page table, the VMM takes control and update (or add) a shadow table
- If a change in the PTBR occurs, the new table is analyzed and the shadow tables are added

Virtualizing I/O Devices

- I/O instructions are usually privileged instructions, i.e. they cannot be executed in userspace
- In a multiprogrammed environment processes access the I/O space through the abstraction implemented by the kernel
- The VMM instead must give the illusion to the guest VM that it is in control of the hardware and it can access it directly
- The VMM must emulate the real hardware creating a virtual representation implemented as a set of data structures in memory

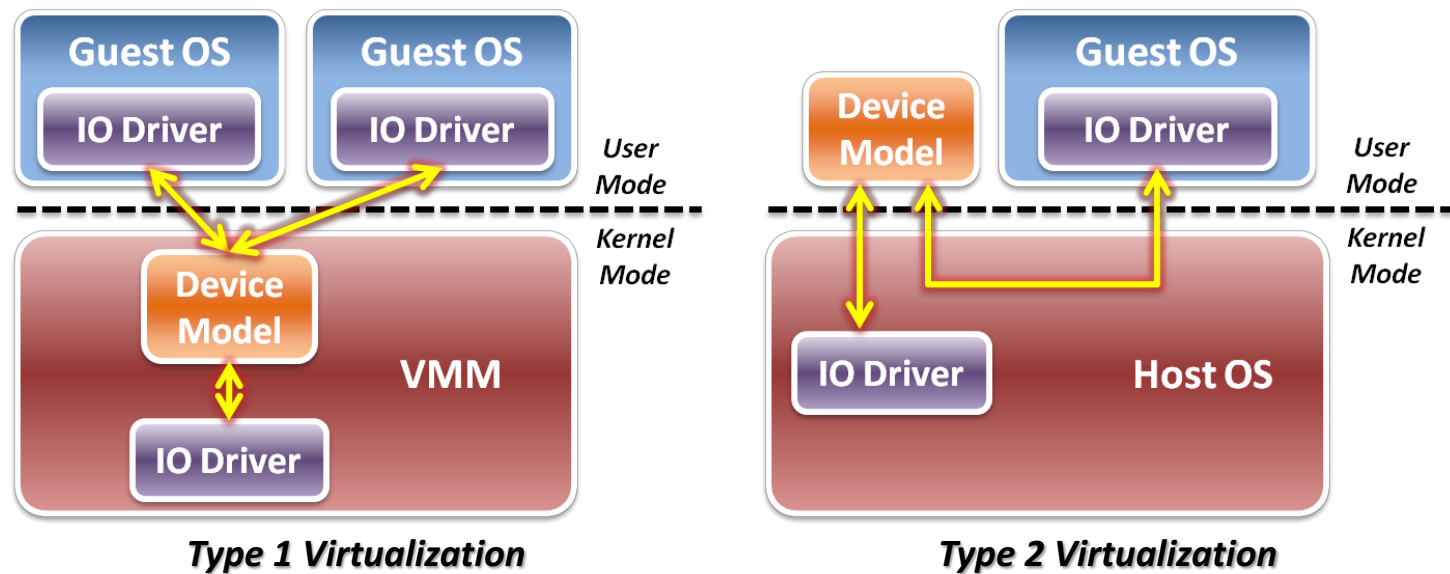
Virtualizing I/O Devices

- The *virtual representation (Device model)* of the peripherals are accessed by the guest VM, the VMM accesses/updates their representation every time a read/write is executed
- Eventually, the VMM is the only controller of the peripherals of the physical system, and they can translate the I/O instructions commands on the virtualized peripheral to an I/O instruction on the real device or emulate the device completely (it depends on the peripheral)



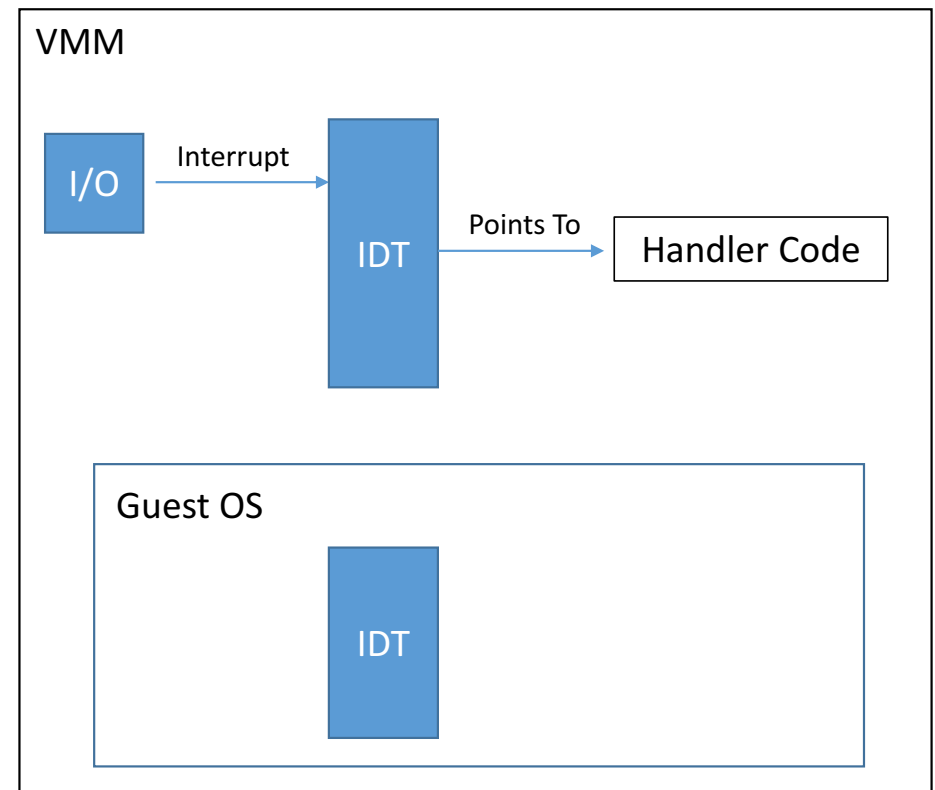
Device models implementation

- Device model implementation depends on the type of the Hypervisor:
 1. Device model is implemented as a part of VMM.
 2. Device model is running in user space as a stand alone service (the VMM is running in userspace as a service).



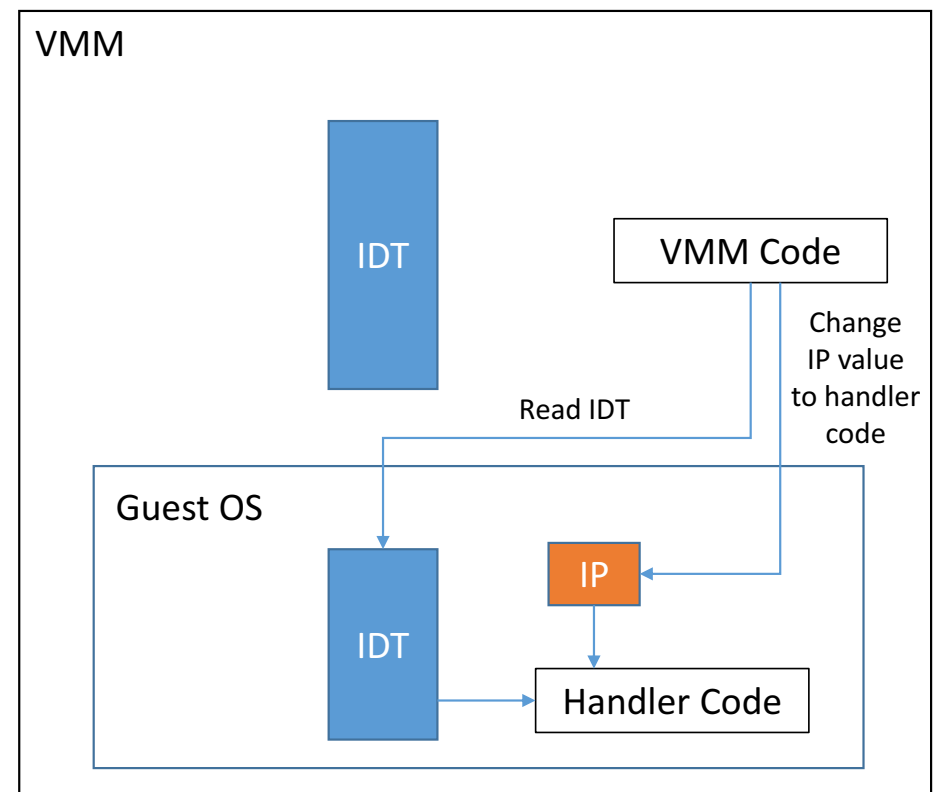
Interrupt Management

- The VMM must take care of the interrupts coming from the physical devices, it must handle them in a transparent manner, with respect to the guest VMs
- VMM must keep a host interrupt description table (inaccessible to the guests) that the host CPU uses to handle the interrupts
- Each guest VM have its own interrupt descriptor table managed by the guest OS
- The VMM must takes care of the interrupts that are generated from the virtual devices emulated by VMM for the guest OSs



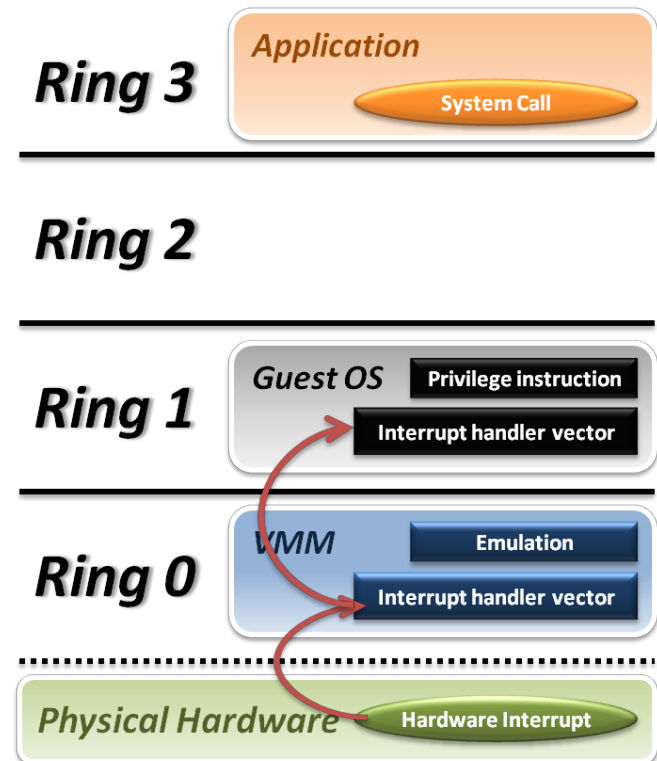
Virtualizing Interrupts

- When the VMM wants to emulate the reception of an interrupt in a VM (e.g. to emulate an input from a virtual device) it must look up the guest interruption table, and perform the operations required to execute the interrupt code in the guest OS:
 - Save the current state
 - Change the instruction pointer (the registry in the CPU that points to the next instruction to be executed) to the value of the first instruction in the interrupt code
 - As the VMM gives back control to the guest VM it will execute the interrupt code
- The VMM must consider the fact that the guest CPU could disable the interrupts, in that case it must wait until they are enabled again before emulating the interrupt reception



Hardware Interrupt

- VMM is the only one accessing the real hardware
- Interrupts from devices are handled by VMM
- Hardware interrupt triggers the execution of the interrupt at the VMM (specified inside the VMM Interrupt Handler Vector), even if a VM is running
- VMM handler might trigger the execution of the handler inside the Guest OS by emulating its call via software



QEMU

- Quick EMUlator is an open source emulator that performs hardware virtualization
- QEMU is a VMM software that emulates the machine hardware
- It supports a wide set of hardware emulations
- It can run a guest OS with an instruction set different from the one of the host through binary translation or run a VM with the same set
- In the latter case QEMU has an accelerator to speed up emulation in order to run some of the code of the guest OS as user mode code
- It is widely adopted in many other projects open source projects using virtualization