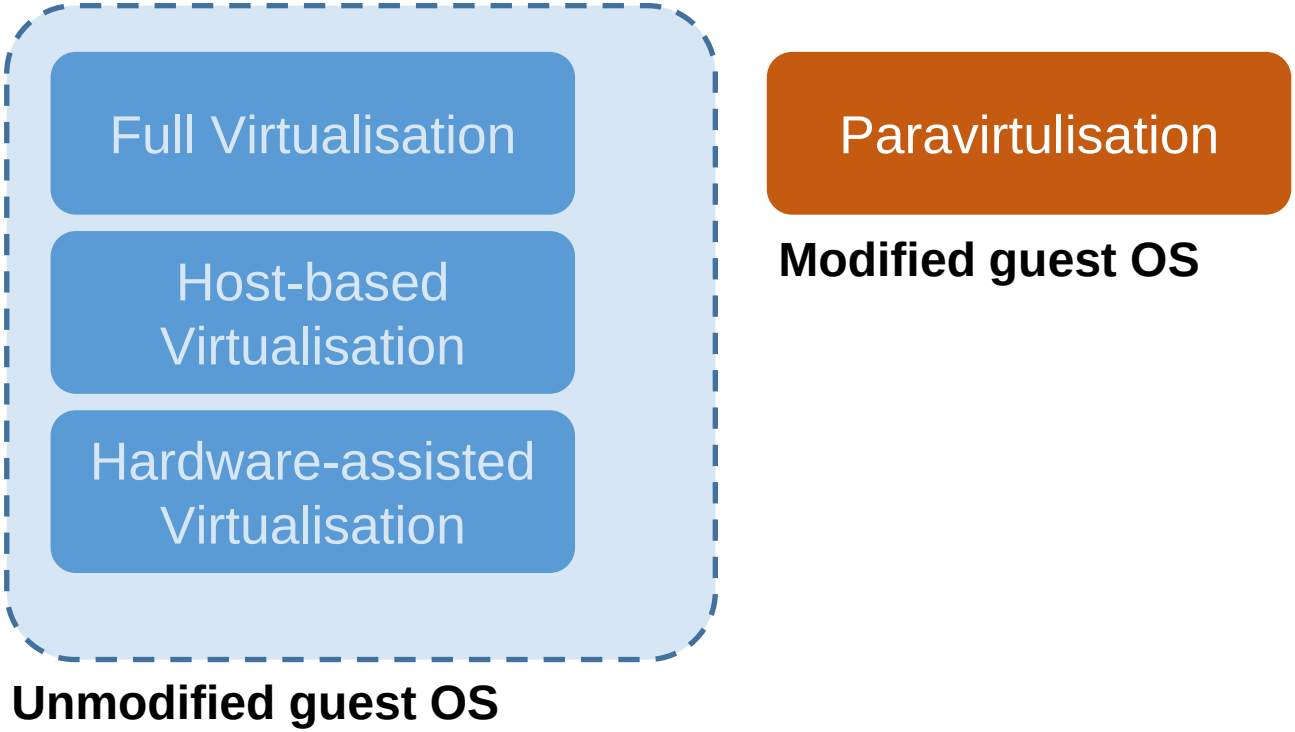
Virtualisation Technologies

CCS3341 Cloud Computing

Dr S. Veloudis

Virtualisation

- Main types of CPU virtualisation:



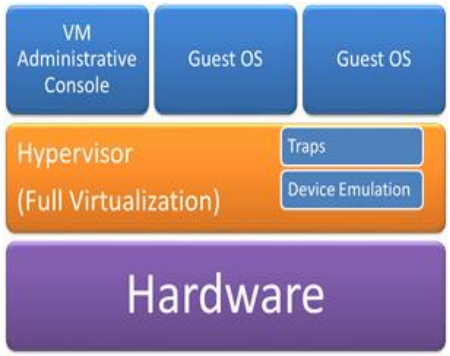
Full Virtualisation

- Full virtualisation provides complete hardware emulation
- The hardware emulator is called the hypervisor (or Virtual Machine Monitor – VMM)
- The hypervisor creates and presents a simulated hardware interface to each guest OS
  - In other words, the hypervisor simulates in software a complete hardware environment (the so-called virtualisation environment) with all the functionality required for interacting with a guest OS
    - **The guest OS is not aware that it is not running natively**
    - All software that can run natively on the hardware of the host machine can also run in the virtual machine (VM)
    - Full virtualisation provides complete isolation between guest OSs
  - But how is this possible?

Note: The term hardware here includes **CPU**, **memory**, **storage**, **network interfaces**, and other devices required for the guest operating system to function properly

Full Virtualisation – Binary Translation

- A guest OS issues instructions which in a non-virtualised environment would directly execute on, and potentially affect, the hardware
  - Such an effect on the underlying hardware could in turn affect other guest machines on the same host
  - These calls are trapped by the hypervisor and are executed virtually (this is often referred to as trap and emulate)
    - They are translated into one or more instructions that do execute on the real CPU but affect instead the simulated hardware
    - Thus by the term 'virtual execution' we refer to the execution of an instruction whose hardware effect may be on the software-emulated hardware (not on the real hardware)
- The trapping and virtual execution of instructions is referred to as binary translation



Full Virtualisation – Binary Translation

- Example: IDT
  - The IDT (Interrupt Description Table) is a data structure used by the CPU to handle interrupts and software exceptions
  - When a hardware device generates an interrupt, or when a software exception occurs, the CPU uses the IDT to determine the appropriate interrupt or exception handler
    - The IDT maps interrupt/exception codes to corresponding interrupt/exception handlers (routines) in memory
  - Any modification to the IDT may significantly impact system stability/security
  - The following instruction loads an IDT effectively replacing the existing IDT

```
lidt [idt_descriptor]
```

Note (Interrupts): Application SW is not allowed to directly interact with hardware devices (except, of course, from memory addresses and general-purpose CPU registers). There two main reasons for this:

1. Different versions of the same SW would be required to interact with hardware devices of the same kind but from different manufacturers.
2. Security: it is dangerous to allow SW to alter the state of hardware devices!

To interact with hardware devices, application SW issues interrupts. Interrupts provide an API through which OS services are called: these service are called interrupt handlers and undertake the task of interacting with the hardware devices in a controlled and secure manner.

Full Virtualisation – Binary Translation

- Example: IDT
  - The `lidt` instruction is unsafe
  - It may be binary-translated to

```
mov edx, [idt_descriptor.base]
mov eax, [idt_descriptor.limit]
mov [vm_idt_descriptor_base], edx
mov [vm_idt_descriptor_limit], eax
```

`vm_idt_descriptor_base` and `vm_idt_descriptor_limit` are data structures that represent the virtual machine's IDT in software, and the `mov` instructions copy the base address and limit values from the IDT descriptor into these data structures

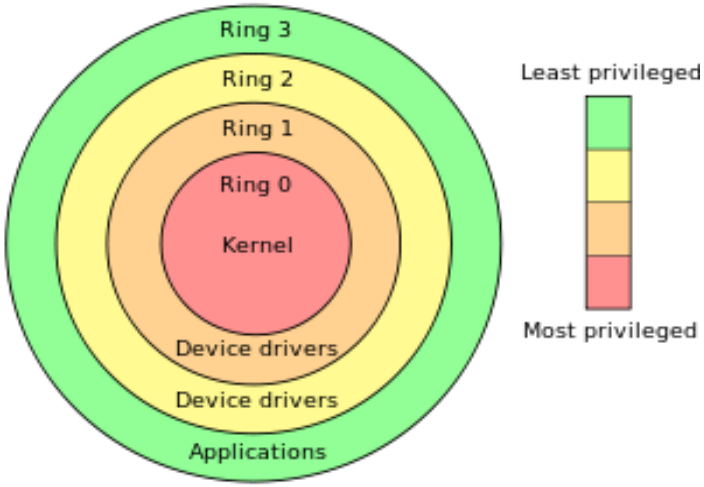
Note: Instructions that alter critical data structures such as the IDT, the GDT (Global Descriptor Table), the control registers (CR0-CR4), and the **page tables**, are typically only issued from within OS routines; they are not encountered in application SW. However, certain application SW instructions may indirectly trigger "OS-level" instructions. The **`fork()`** instruction in C, for example, may cause a page fault and subsequent exception to update the page table

Full Virtualisation – Binary Translation

- Binary translation incurs a significant performance penalty
  - It is generally much faster to run an instruction directly on hardware rather than to emulate its execution in software
    - The virtual execution of an instruction gives rise to other instructions, the ones that perform the actual emulation, which are executed on the real hardware and affect the emulated hardware
    - This set of 'other' instructions requires more time to execute than a single instruction…
- Evidently, the lesser instructions we emulate the better the performance
- Therefore, the hypervisor does not trap all instructions
  - Non-critical instructions run directly on the host hardware
  - Critical (or privileged or unsage) instructions are emulated in software
- But which instructions are critical and which are non-critical?

Full Virtualisation

- OSs provide different levels of access to resources by implementing privilege rings
- Privilege rings are arranged in a hierarchy from the most privileged (usually numbered zero) to the least privileged (usually with the highest ring number)



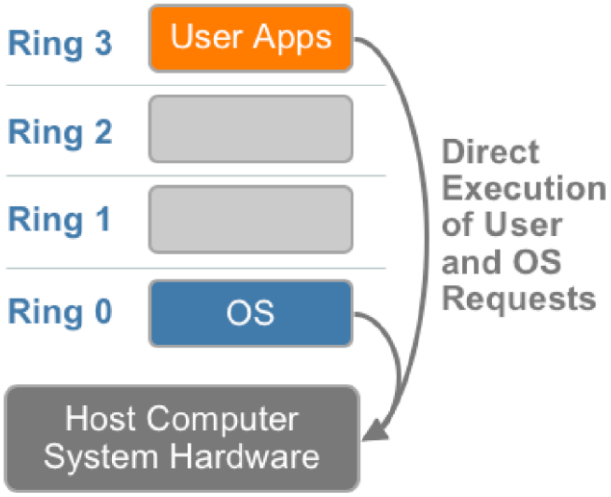
x86 architecture

- Ring 0 issues instructions directly to the physical hardware (CPU, memory)
- Special interfaces (effectively APIs) between rings allow an outer ring to access an inner ring's resources in a predefined and secure manner
  - The special interfaces are interrupts that invoke system calls issued from lower-numbered rings

Full Virtualisation

- Therefore, with this approach:
  - Non-critical application-level instructions (user apps) neither control the hardware nor threaten the security/integrity of the host system
    - They can therefore be safely executed on the real underlying hardware
    - Of course, this requires that the architecture of the underlying CPU is compatible with the application-level instructions
  - A guest machine cannot be affected by the critical instructions issued by another guest machine

Note: Rings are implemented in hardware through the Descriptor Privilege Level (DPL) bits: 2 bits in segment descriptors that control the privilege level(s) that may access a memory segment. Segments are contiguous memory addresses that hold data or code and which enjoy the same level of protection (e.g. all memory addresses are accessible from Ring 3). Segments are usually in the order to a few hundred KiBs each.



Full Virtualisation

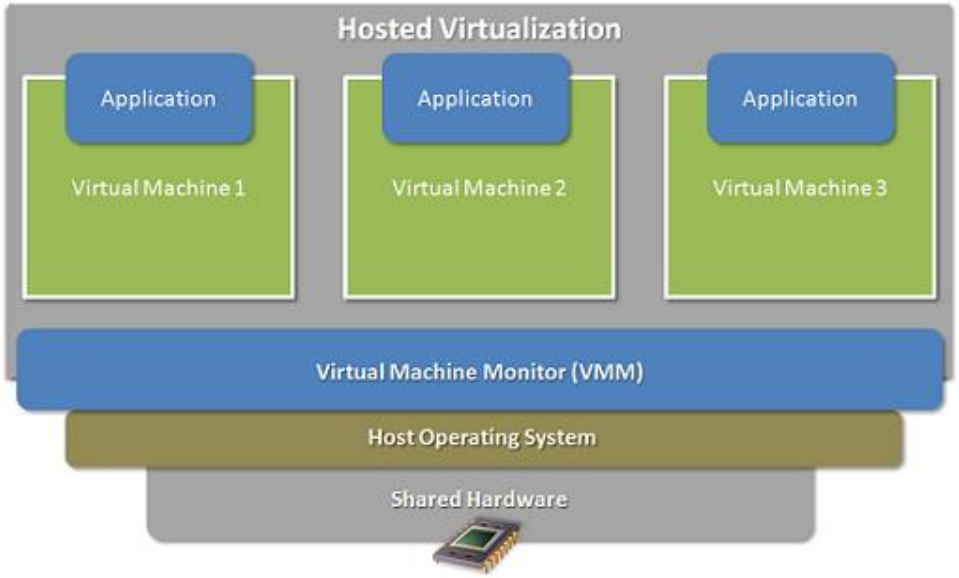
- The hypervisor used in full virtualisation is a.k.a. as type 1
- To further optimise performance, binary translation is typically implemented through the aid of a cache
  - Keeps recently translated instructions to speed up the translation of similar subsequent instructions
- Emulation is a special case of full virtualisation whereby all instructions (both critical ones and non-critical ones) are subjected to binary translation
  - Allows the simulation of hardware different from the one of the host machine
    - An Android emulator running on a Windows box (Windows has a different processor architecture than an Android device)
    - MAME (Multiple Arcade Machine Emulator) allows us to play arcade games designed for old architectures
  - The problem with emulation is, of course, performance…

Full Virtualisation

- To recap
  - Type-1 hypervisors are installed directly on the hardware and completely replace the host OS
  - System calls from the guest OSs are trapped
  - If the effect of these calls is disruptive for other guest machines, these calls only affect the emulated hardware
- Advantages
  - True isolation of VMs
  - Allows dissimilar guest OSs
  - No modifications to the guest OS
  - User-level instructions run at native speed
  - Facilitates VM portability
- Disadvantages
  - Performance penalty from binary translation
  - Compatibility with driver updates

Host-based Virtualisation

- A.k.a. OS-level virtualisation or hosted virtualization
- This is an alternate VM architecture whereby a virtualisation layer (i.e., the hypervisor or VMM) is built on top of the host OS
  - The host OS (and not the hypervisor) is responsible for managing the underlying (real) hardware
- Guest OSs are installed and run on top of this virtualisation layer



Host-based Virtualisation

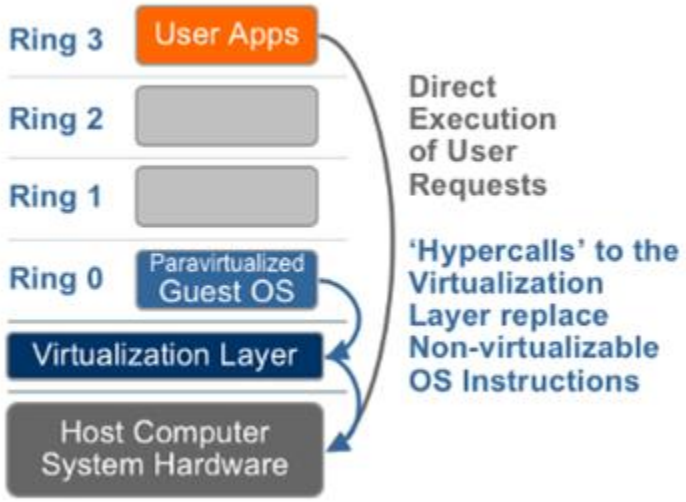
- The hypervisor is thus no different than any other application running in user space
- The hypervisor still provides a separate software emulation of the hardware environment for each guest OS
  - System calls from a guest OS that may affect the underlying hardware in a manner disruptive for other guest machines, only affect the virtualised hardware
  - Other system calls that simply use (but not disruptively affect) the underlying hardware are trapped by the hypervisor and passed over to the host OS which then executes them on the real hardware
- Non-critical instructions execute directly on the real hardware
- The hypervisor used in host-based virtualisation is a.k.a. type 2

Host-based Virtualisation

- Advantages:
  - The host OS is not replaced
  - True isolation of VMs
  - Allows dissimilar guest OSs
  - No modifications to the guest OS required
  - The deployment of the hypervisor is simplified (just installed as an application on top of the OS – no need to install it on the bare metal)
- Disadvantages
  - Performance
    - Recall that the hypervisor traps systems calls and passes them over to the host OS which then attempts to execute them on the real hardware
    - This involves more overhead than the bare-metal type-1 hypervisor that runs directly on hardware

Paravirtualisation

- With this virtualisation technique, the guest OS is substantially modified
  - The critical instructions that need to interact with the hardware of the guest OS are replaced by hypercalls that trap directly into the hypervisor
  - This replacement takes place at compile-time rather than at run-time
  - Hypercalls are essentially APIs through which the functionality of the hypervisor may be requested; they are thus a kind of interrupts
  - A paravirtualised guest OS is not able to run directly on hardware
- Note that guest OSs are aware that they are running virtually and not natively on the host machine



Paravirtualisation

- As in full virtualization, the hypervisor provides SW-emulated HW to ensure VM isolation
- Unlike full virtualization, no binary translation needs to be performed
- Advantages
  - Increased performance - the performance advantage of paravirtualisation over the other kinds of virtualisation may vary widely depending on the workload
- Disadvantages
  - Inapplicable to 'closed' OSs (e.g., Windows)
  - Poor portability
  - Deep OS kernel modifications are error prone and can give rise to support and maintainability issues (e.g. OS updates, patches, etc.)

Xen

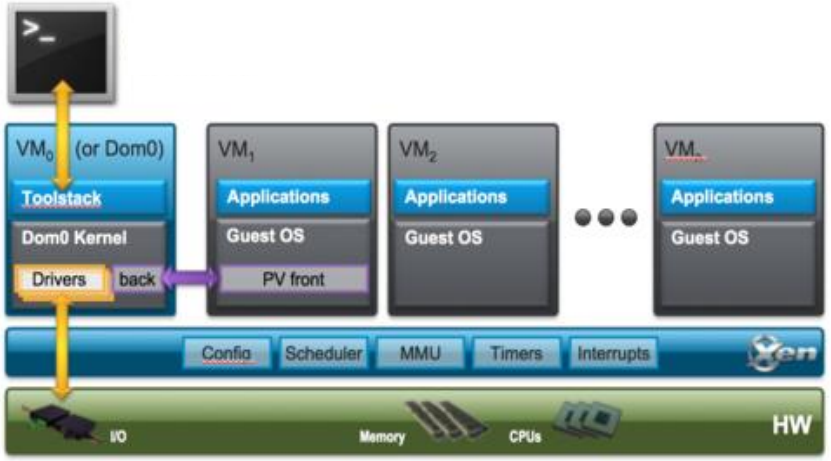
- Xen is a paravirtualisation hypervisor developed by the University of Cambridge
- A Xen hypervisor is responsible for:
  - CPU scheduling of VM tasks
  - Memory partitioning between VMs (ensuring isolation)
  - Abstracting HW from VMs
- To keep the Xen hypervisor generic and 'thin', typical hypervisor functionality is delegated to the first VM launched – the so-called Domain0/Dom0
  - The hypervisor is unaware of networking, external storage, or any other common I/O function
- Dom0:
  - Creates and configures DomUs
  - Accesses physical I/O resources (disk, network)
  - Access HW devices

Note: Dom0 runs a Unix-like OS

Xen

contd
.

- Xen abstracts VMs from hardware devices by means of frontend and backend drivers
  - Frontend drivers reside at user domains (VMs other than Dom0); they are software modules that are invoked each time a guest OS wants to invoke a driver
  - Frontend drivers make hypercalls (special privileged instructions provided by the hypervisor) to communicate with backend drivers
  - Frontend drivers are agnostic to HW devices
  - Backend drivers reside at Dom0
  - Backend drivers are real device drivers that are installed along with



Note: User domains are called DomUs

Xen

contd
.

- Frontend drivers have no hardware access
- Frontend and backend drivers facilitate VM portability
  - Each guest OS only needs to be modified to provide support for the generic device classes
  - Eliminates the need to support each possible physical device
  - Facilitates updates and maintenance
- Frontend and backend drivers communicate through the Xenstore: holds configuration and status information about each domain for example:
  - network interfaces (IP addresses assigned to them, current network traffic statistics)
  - disk images (locations, access permissions)
  - current state of the hypervisor (including state of each VM)
  - performance metrics

Xen

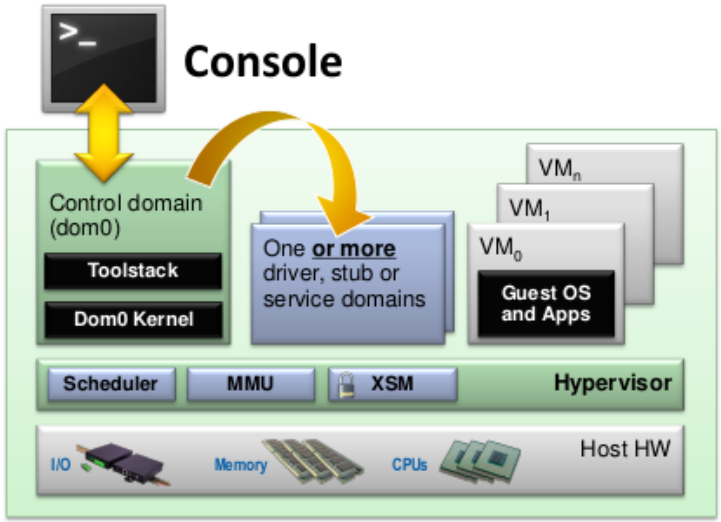
- Xenstore:
  - No-SQL store
  - Has a hierarchical tree-like structure
  - Keys are organized in paths as in file systems (where they keys are the filenames)
  - This way related configuration information can be grouped together

```
/
|-- vm
|   |-- domain1
|   |   |-- name = "VM 1"
|   |   |-- memory = "512 MB"
|   |
|   |-- domain2
|       |-- name = "VM 2"
|       |-- memory = "1 GB"
|
|-- devices
|   |-- pci
|       |-- 00:00.0
|       |   |-- vendor_id = "1234"
|       |   |-- device_id = "5678"
|
|-- network
|   |-- interface1
|   |   |-- mac = "00:11:22:33:44:55"
|   |   |-- ip = "192.168.1.2"
|
|-- storage
    |-- disk1
        |-- capacity = "100 GB"
```

Xen

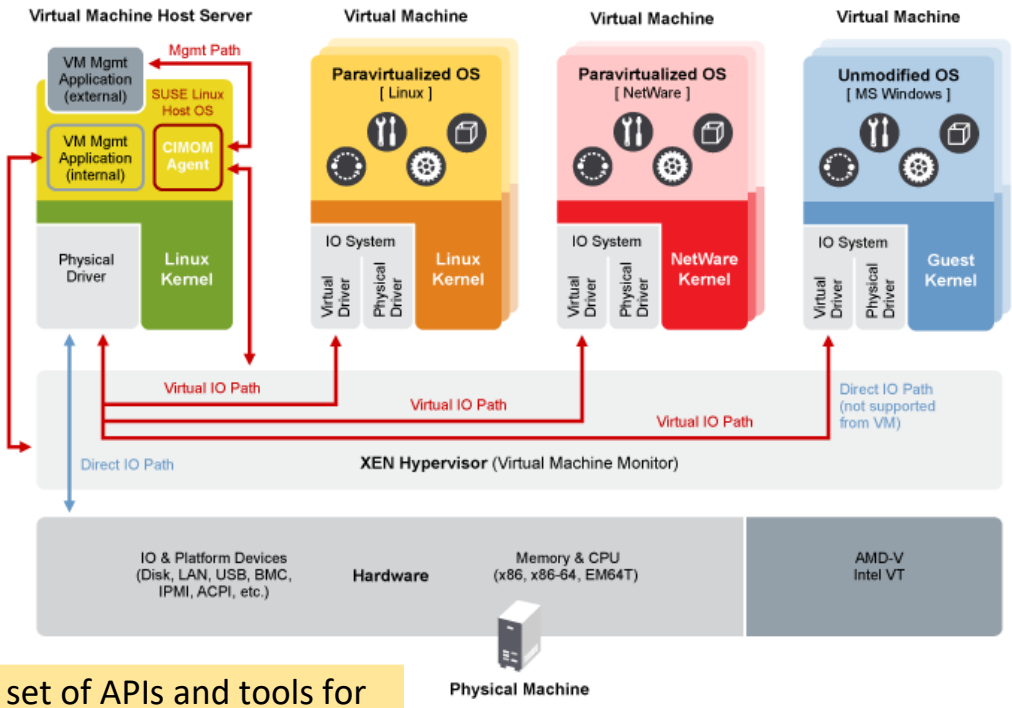
- The Xen hypervisor is responsible for conveying hypercalls to Dom0 in case a DomU is to perform an unsafe operation
  - The operation is performed by Dom0 on the emulated hardware maintained for each VM
- Communication between different domains is achieved through event channels

Note: Event channels are implemented as pairs of shared memory pages. In each pair, one page is in the memory space of the sending domain and one in the memory space of the receiving domain



Xen

- More recent versions of Xen also support full virtualisation and can therefore run closed OSs – e.g. Windows



- **Note**: "Xen Management API" (XAPI) is a set of APIs and tools for managing Xen-based virtualization environments
- It enables administrators to interact with the Xen hypervisor and create/configure/remove VMs
- It can be used with any programming language that supports HTTP/RESTful APIs; it also includes a command-line interface and a web-based (GUI) for managing virtualization resources
- XAPI communicates with the Xen hypervisor through the Xenstore

Hardware-assisted Virtualisation

- Full virtualisation examined hitherto is software-based
  - Virtual execution of instructions affects software-emulated hardware (not the real hardware)
- However, full virtualisation suffers significant performance penalties
- To overcome these drawbacks, another form of full virtualisation, hardware-assisted virtualisation, is introduced
  - Circa 2007, both leading chip manufacturers, Intel and AMD, launched virtualisation technologies that were CPU-supported:
    - VT-x (Intel)
    - AMD-V
- Both VT-x and AMD-V share the same basic structure (although not entirely equivalent)
  - Indicatively, we shall examine VT-x's structure

Hardware-assisted Virtualisation – VT-x

VT-x introduces **virtualisation extensions**

HW components and SW structures that aim at reducing the virtualisation overhead

- CPU registers
- In-memory data structures
- HW structures on I/O devices

VM Exit and **VM Entry instructions** are VMX instructions introduced by VT-x

- VT-x defines two modes of CPU operation

Root operation
Intended for hypervisor execution ('host' mode)

Non-root operation
Intended for VM execution ('guest' mode)

Each time a guest OS attempts to execute an unsafe instruction, a **VM Exit instruction** is executed and control is transferred to the hypervisor which emulates the behaviour of the instruction. Control is transferred back to the guest OS (**VM Entry instruction**) when this emulation completes

Shadow Registers

VT-x introduces the concept of **shadow** registers

Certain important registers (e.g. **CR0, CR3, CR4, DR7)** are duplicated (shadowed) on the CPU chip

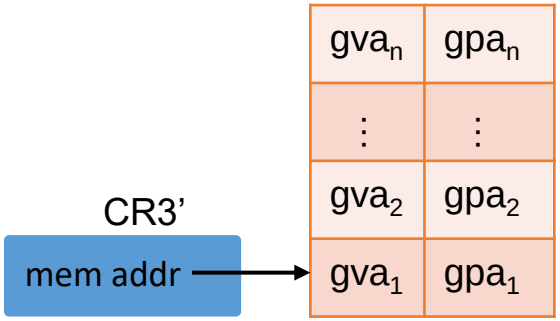
The purpose of shadowing is to reduce the number of times the hypervisor needs to intervene when a guest OS tries to change an important register

NOTE: Other important registers such as the IDTR or the GDTR are **not** shadowed because their changes are infrequent and don't justify wasting space on the CPU chip for shadow registers…

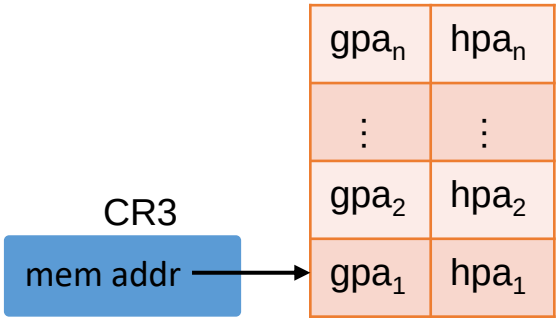
Consider for instance CR3 and let CR3' be its shadow. CR3 points to the page table maintained by the hypervisor, and CR3' to a virtual Page Table (vPT) maintained by the guest OS. The guest can freely change CR3' without such changes **necessarily** requiring hypervisor intervention

Justification:

Consider the virtual Page Table pointed by CR3'. It maps guest virtual addresses (GVAs) to guest physical addresses (GPAs). The latter are virtual addresses that the guest OS thinks are physical:

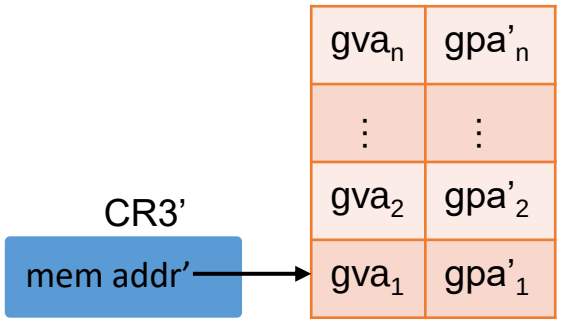


Consider the Page Table pointed by CR3. It maps GPAs to real physical addresses (HPAs):



Shadow Registers

Consider now that the guest OS changes CR3'. This change likely causes changes to the virtual PT mappings:



Depending on the changes, the hypervisor may, or may not, need to intervene and change the mappings into its PT as well

If $\{gpa'_1, \dots, gpa'_n\}$ (i.e., the right column of the vPT above) is equal to $\{gpa_1, \dots, gpa_n\}$ (i.e., the right column of the vPT in the previous slide) then the change in CR3' alters the way in which $gva_1, \dots gva_n$ are mapped to GPAs, but these GPAs are essentially the same as previously

In other words, for any i, gva_i was mapped to gpa_i, and now gets mapped to a gpa'_i such that gpa'_i was previously the mapping of another GVA

The fact that the new GPA (gpa'_i) that gva_i maps to was previously in the vPT means that it is also in the PT that is maintained by the hypervisor (see previous slide) and **hence its mapping to an HPA is already known**

No hypervisor intervention required!

On the other hand, if gpa'_i is a new address that was not previously in the vPT and is not mapped to an HPA by the PT, then when the hypervisor tries to map gpa_i to an HPA a **page fault** occurs and the **hypervisor needs to intervene** (a VM Exit occurs)

Extended Page Tables

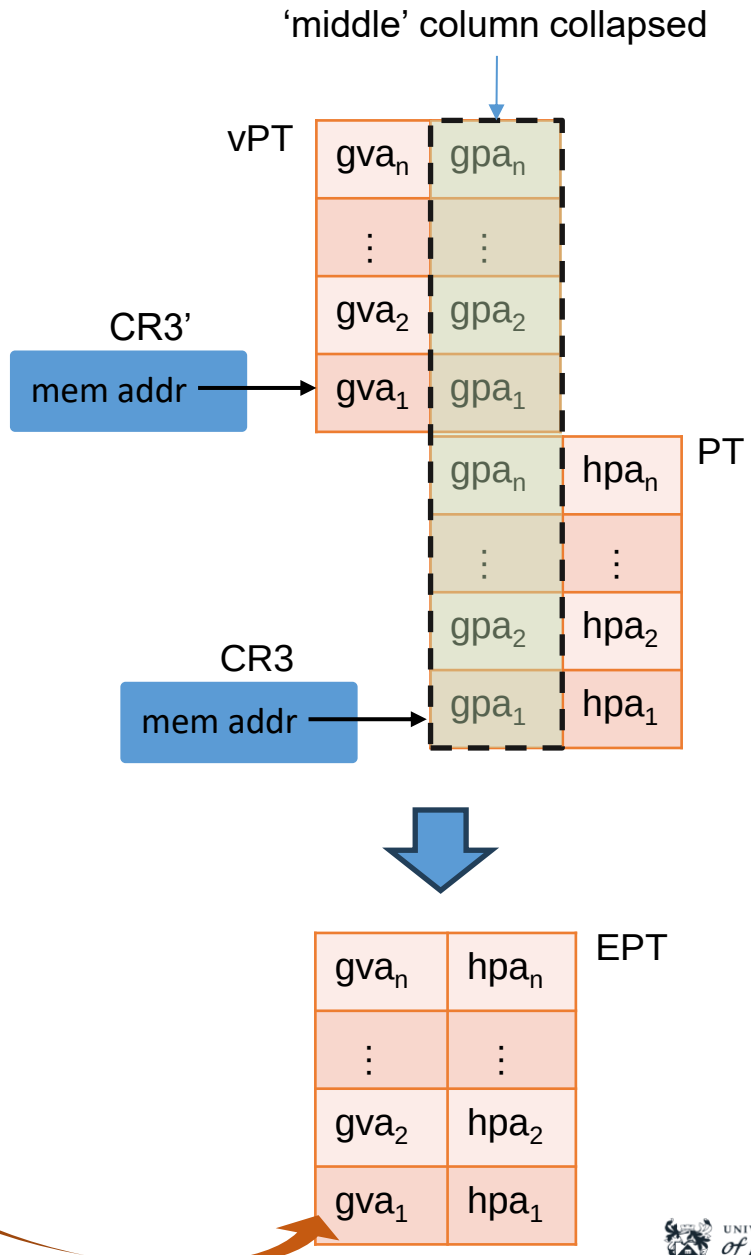
Observe how in the previous case we restrict the need for hypervisor interventions only to cases when the new GPAs are not already mapped by the PT

This contrasts with full virtualization where any change to CR3 necessarily traps to the hypervisor which then binary translates this change to affect instead a memory structure that emulates the CR3; **this incurs a significant performance penalty**

Entails memory accesses which are time consuming

Extended Page Table (EPTs) introduce a **new optimization** that eliminates the 2-level mapping from GPAs to HPAs of slide 25

Extended Page Table (EPTs) map directly GVAs to HPAs and are maintained by the hypervisor alongside PTs



VMCS

VT-x introduces the **Virtual Machine Control Structure (VMCS)** to control interaction between hosts and guests

- VMCS resides in hypervisor memory space (RAM)
- There is one VMCS per VM

The VMCS serves as a **bridge** between the VM's state information in memory and the CPU's execution context

The VMCS contains various fields that describe VM state

CONTROL FIELDS				
Pin-Based VM-Execution Controls	External-interrupt exiting	NMI exiting	Virtual NMIs	
	Activate VMX-preemption timer	Process posted interrupts		
Primary processor-based VM-execution controls	Interrupt-window exiting	Use TSC offsetting		
	HLT exiting	INVLPG exiting	MWAIT exiting	RDPMC exiting
	RDTSC exiting	CR3-load exiting	CR3-store exiting	CR8-load exiting
	CR8-store exiting	Use TPR shadow	NMI-window exiting	MOV-DR exiting
	Unconditional I/O exiting	Use I/O bitmaps	Monitor trap flag	Use MSR bitmaps
	MONITOR exiting	PAUSE exiting	Activate secondary controls	
Secondary processor-based VM-execution controls	Virtualize APIC accesses	Enable EPT	Descriptor-table exiting	Enable RDTSCP
	Virtualize x2APIC mode	Enable VPID	WBINVD exiting	Unrestricted guest
	APIC-register virtualization	Virtual-interrupt delivery	PAUSE-loop exiting	
	RDRAND exiting	Enable INVPCID	Enable VM functions	VMCS shadowing
	Enable ENCLS exiting	RDSEED exiting	Enable PML	EPT-violation #VE
	Conceal VMX non-root operation from Intel PT	Enable XSAVES/XRSTORS		
	Mode-based execute control for EPT	Use TSC scaling		
Exception Bitmap	I/O-Bitmap Addresses	TSC-offset		
Guest/Host Masks for CR0	Guest/Host Masks for CR4	Read Shadows for CR0	Read Shadows for CR4	
CR3-target value 0	CR3-target value 1	CR3-target value 2	CR3-target value 3	CR3-target count
APIC Virtualization	APIC-access address	Virtual-APIC address	TPR threshold	
	EOI-exit bitmap 0	EOI-exit bitmap 1	EOI-exit bitmap 2	EOI-exit bitmap 3
	Posted-interrupt notification vector	Posted-interrupt descriptor address		
Read bitmap for low MSRs	Read bitmap for high MSRs	Write bitmap for low MSRs	Write bitmap for low MSRs	
Executive-VMCS Pointer	Extended-Page-Table Pointer	Virtual-Processor Identifier		
PLE_Gap	PLE_Window	VM-function controls	VMREAD bitmap	VMWRITE bitmap
ENCLS-exiting bitmap	PML address			
Virtualization-exception information address	EPTP index	XSS-exiting bitmap		

VM-EXIT CONTROL FIELDS					
VM-Exit Controls	Save debug controls	Host address space size	Load IA32_PERF_GLOBAL_CTRL		
	Acknowledge interrupt on exit	Save IA32_PAT	Load IA32_PAT	Save IA32_EFER	Load IA32_EFER
	Save VMX preemption timer value	Clear IA32_BNDCFGS	Conceal VM exits from Intel PT		
VM-Exit Controls for MSRs	VM-exit MSR-store count	VM-exit MSR-store address			
	VM-exit MSR-load count	VM-exit MSR-load address			

VM-EXIT INFORMATION FIELDS				
Basic VM-Exit Information	Exit reason	Exit qualification		
	Guest-linear address	Guest-physical address		
VM Exits Due to Vectored Events	VM-exit interruption information	VM-exit interruption error code		
VM Exits That Occur During Event Delivery	IDT-vectoring information	IDT-vectoring error code		
VM Exits Due to Instruction Execution	VM-exit instruction length	VM-exit instruction information		
	I/O RCX	I/O RSI	I/O RDI	I/O RIP
VM-instruction error field				

- Natural-Width fields.
- 16-bits fields.
- 32-bits fields.
- 64-bits fields.

Complex!

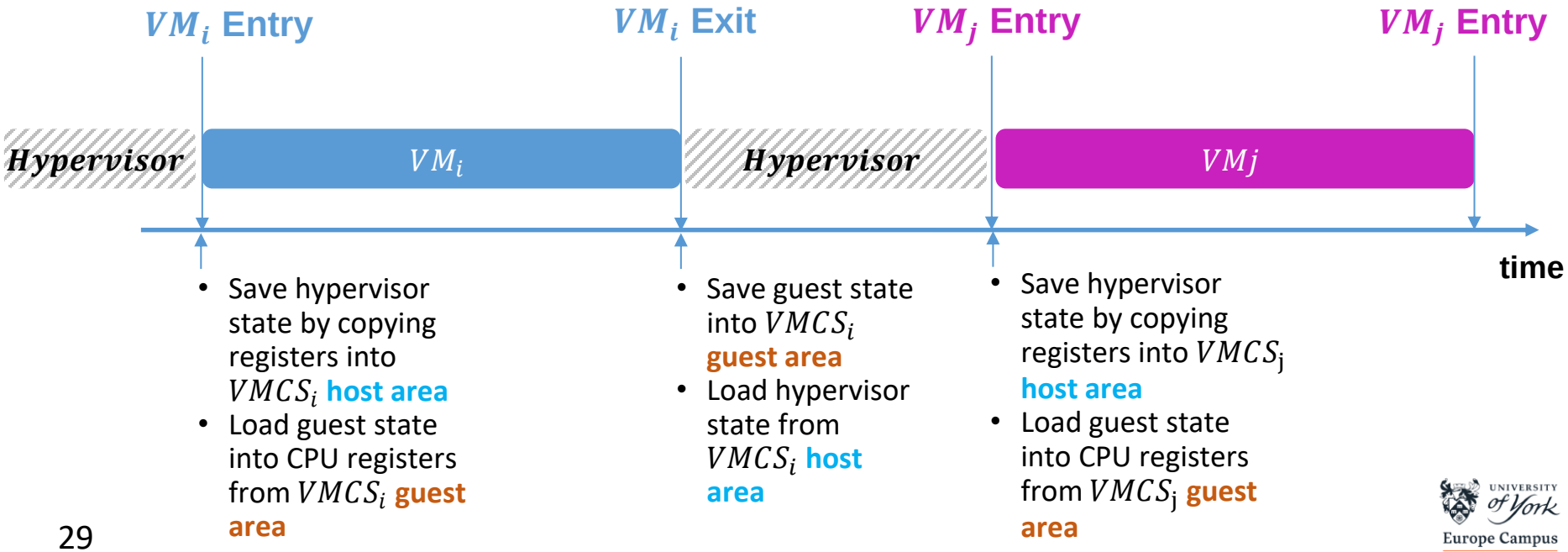
VMCS Fields

Guest-state Area

- Responsible for holding guest state by storing the content of registers **upon VM Exit** i.e., when a VM was previously running and it now is about to exit
- This state is then reloaded to the CPU registers when a VM entry occurs and the VM is to resume execution

Host-state Area

- Responsible for holding host state by storing the content of registers **upon VM Entry** i.e., when the hypervisor was previously running and it now is about to pass control to a VM
- This state is then reloaded to the CPU registers when a VM exit occurs and the VM is to resume execution



VMCS Fields

Host-state Area

- Observe that there are **multiple** host areas, one per VMCS
- Also observe that, after the VM_j Exit in the figure of the previous slide, the content of $VMCS_i$'s host state (that was saved upon VM_i Entry) **is never used again**…

Question:

Why don't we designate a **single memory area** and use it for storing host (hypervisor) state upon **each** VM Entry, and for loading host state into CPU upon **each** VM Exit?

That could work but it is **not as efficient performance-wise**…

- Suppose a single memory area is used
- Then, upon a VM Exit:
  - The hypervisor stores the CPU state in the guest area of the VMCS
  - The hypervisor performs a **new memory access** to the designated memory area where host state is stored
  - Resumes CPU state from this new area
  - The new memory access **incurs a performance penalty**
  - This is avoided if, instead, the hypervisor resumes CPU state from the VMCS which the hypervisor is accessing thus not require any further memory accesses

Performance is more important than a small waste of (inexpensive nowadays) RAM!

VMCS Fields

VM-Exit/Entry control fields

- **VM-Exit Reasons**: Indicate the reasons/conditions for a VM exit/entry (e.g., due to an interrupt request or unsafe instruction execution); useful for auditing
- **VM-Exit MSR-Store and MSR-Load**: Specify whether certain model-specific registers (MSRs) should be saved or loaded during VM exits/entry (in case of entry, this enables the VM to have access to specific MSRs while running)

See later!

VM-Execution control fields

- **Pin-based controls**: Determine which events can cause the VM to "pin" to the current physical processor (precluding migration to another core)
- **Processor-based controls**: Specify the conditions under which VM exits occur for various reasons, such as exceptions or specific instructions

- Let a VMs be running a high-priority application that requires consistent and low-latency access to a specialized GPU
- When the VM executes the instruction to start the GPU process, the hypervisor is configured to pin the VM to a specific processor core that is associated with the dedicated GPU. This configuration ensures that the VM consistently runs on a core with direct access to the GPU

VMX Instructions

VT-x introduces specialized instructions to enable, configure, and manage virtual machine operation

Key VMX instructions:

- **VMXON/OFF (Virtual Machine eXit ON/OFF):** enable/disable VMX operation. Sets/unsets VMX operation mode in CPU
- **VMPTRLD (VM-Pointer Load):** load the address of a VMCS from memory, making it the current VMCS for the VM that the processor is operating on
- **VMPTRST (VM-Pointer Save):** save the address of the current VMCS in memory, allowing for later retrieval or switching between VMCSs
- **VMCLEAR (VM-Clear):** clear a VMCS, making it available for reuse or for the creation of a new virtual machine
- **VMLAUNCH (VM-Launch):** launch a virtual machine. It is executed after loading the appropriate VMCS with the desired configuration
- **VMRESUME (VM-Resume)**: resume execution of a virtual machine that has been previously launched using VMLAUNCH
- **VMREAD and VMWRITE:** read and write various fields in the VMCS, allowing the hypervisor to configure and retrieve information about the virtual machine's state

VMX Instructions

These instructions enable **efficient** VMCS management

- Suppose for a moment that these instructions were not provided
- Emulating these instructions using standard x86 instructions incurs significant performance overhead

Example: emulating VMPTRLD

```
; ECX: Address of the VMCS in memory
test ecx, ecx

; Load memory address
mov eax, ecx
mov vmcs_pointer, eax

not_in_vmxon_state:
; Handle cases where the CPU is not in VMXON state
```

Checks if **ecx** is set to 0

With VMX instructions, this check is avoided!

Loads **ecx** into **vmcs_pointer**

With VMX instructions, this two-step copying is avoided!

VMX MSRs

VT-x uses special registers on the CPU chip – the **Model Specific Registers** (**MSRs**)

For various low-level purposes, including configuration and control of processor features, performance monitoring, power management, and other low-level CPU settings

VMCS MSRs:

- **IA32_VMX_BASIC:** contains such info as the VMCS revision identifier, the maximum number of VMCS fields, etc.
- **IA32_FEATURE_CONTROL:** enable/disable virtualization extensions; commonly used by the BIOS or firmware to configure virtualization support

VMCS MSRs (more):

- **IA32_VMX_PINBASED_CTLS:** Defines which control events are pinned.
- **IA32_VMX_PROCBASED_CTLS:** Defines which control events cause VM exits/entries
- **IA32_VMX_EXIT/ENTRY_CTLS:** Specifies additional info about VM exits and any additional processing that should be done during VM exits – e.g., whether the guest's IP and FLAGS register should be saved in the VMCS for analysis, etc.

These MSRs are loaded from, and their contents are saved to, the corresponding VM Execution and VM Exit/Entry control fields in the VMCS

Detailed exit information is essential for the hypervisor

Note that VMCS MSRs are not pointers to memory-resident information but contain the information themselves

Hardware Virtual I/O Support

Aim: optimize I/O operations, further improving VM performance

Includes features like **Single Root I/O Virtualization** (**SR-IOV**) and **Virtual Machine Device Queues** (**VMDq**)

Virtual devices incur significant performance penalties because they require hypervisor intervention (and the context switching that this entails) at each VM interaction with the virtual device

Hypervisor intervention aims at **emulation**: the VM OS thinks that it is interacting with the physical I/O device whilst in fact it is interacting with an interface that the hypervisor presents and which acts as a proxy interacting with the I/O device on behalf of the VM OS

Virtual I/O Devices

- Full virtualization uses **virtual I/O devices (or appliances)** for enabling VMs to access shared physical I/O devices
- These are virtualized interfaces created and maintained by the hypervisor to isolate VM usage of physical I/O devices

Examples: Virtual NICs, virtual disks, virtual GPUs, virtual USB controllers, …

Hardware Virtual I/O Support

Example: vNIC

All network traffic sent to or from the vNIC is mediated by the hypervisor which forwards traffic between the VM and the physical (NIC) using network bridging or network address translation (NAT)

The VMs are assigned by the hypervisor virtual IP and physical addresses, as well as unique TCP/UDP port numbers

Egress traffic

- Egress traffic from a VM uses these addresses and port numbers in data packet headers (datagrams and frames) to indicate the packets' origin
- These addresses are however not recognizable across the network and the hypervisor replaces them with real IP and physical addresses of the physical NIC

Ingress traffic

- Ingress traffic to a VM arrives at the physical NIC and gets intercepted by the hypervisor which replaces the IP and physical addresses in the headers of data packets with the virtual IP and physical addresses of the destination VM
- The destination VM is resolved on the basis of TCP/UDP port numbers (each VM is assigned a different port number)

- Evidently such frequent hypervisor interventions significantly hinder performance
- Same applies to all kinds of virtual I/O devices!

Hardware Virtual I/O Support

SR-IOV

Aims at avoiding the performance overhead associated with frequent hypervisor interventions

It replaces virtual I/O devices with **Virtual Functions (VFs)**

Virtual functions are **lightweight HW** versions of **Physical Functions (PFs) – i.e., of the HW interfaces that I/O devices offer for communicating with the OS**

This is in contrast with virtual devices which are fully-fledged albeit SW implementations

- Each VF is assigned to a VM
- A VM interacts with its VF in the same way as it would interact through an OS driver with the real PF

Example: NIC

- A NIC's PF is a hardware interface that interacts directly with the physical layer of the TCP/IP stack of a machine's OS
- The hypervisor creates multiple VFs for a NIC's PF (one per VM)
- Each VF interacts directly with the physical layer of the TCP/IP stack of the VM's OS
- This means that the physical layer has direct access to the VFs queues and registers and packet transmission is offloaded directly to the VF's hardware

Note that VMs now use real IP/MAC addresses as opposed to virtual ones!

Hardware Virtual I/O Support

VF Lightweightedness:

- A PF has a complete set of HW resources including control and status registers, queue management, and administrative capabilities
- VFs in contrast only have access to registers and queues that support basic data transmission and reception
- VFs are assigned dedicated queues
- PF is responsible for maintaining HW isolation between VFs