

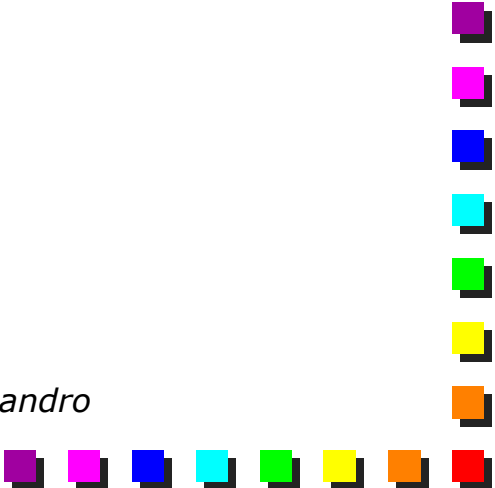


Computing Virtualization Technologies and Tools

Fulvio Riso

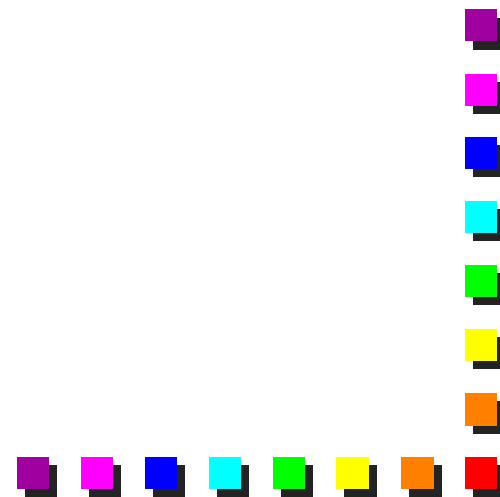
Politecnico di Torino

This set of slides is based on a previous version created by Alex Palesandro





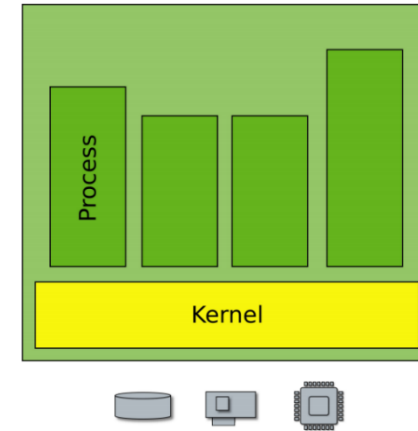
Part I
Introduction



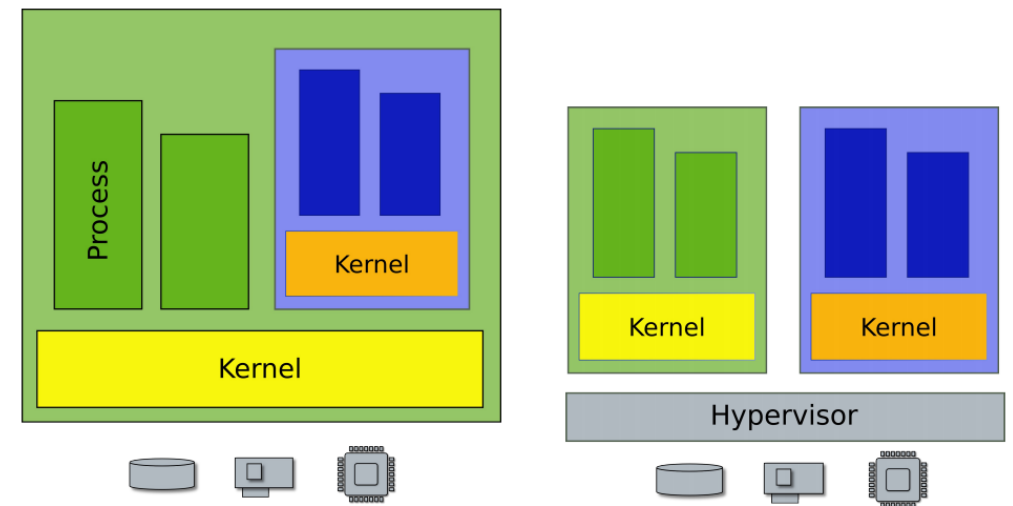
Introduction

- Virtual machines were popular in 60s-70s
 - Share resources of mainframe computers
 - Run multiple single-user operating systems
- Interest was lost in 80s-90s
 - Development of multi-user OS
 - Rapid drop in hardware cost
- Hardware support for virtualization is lost

Operating system on bare metal

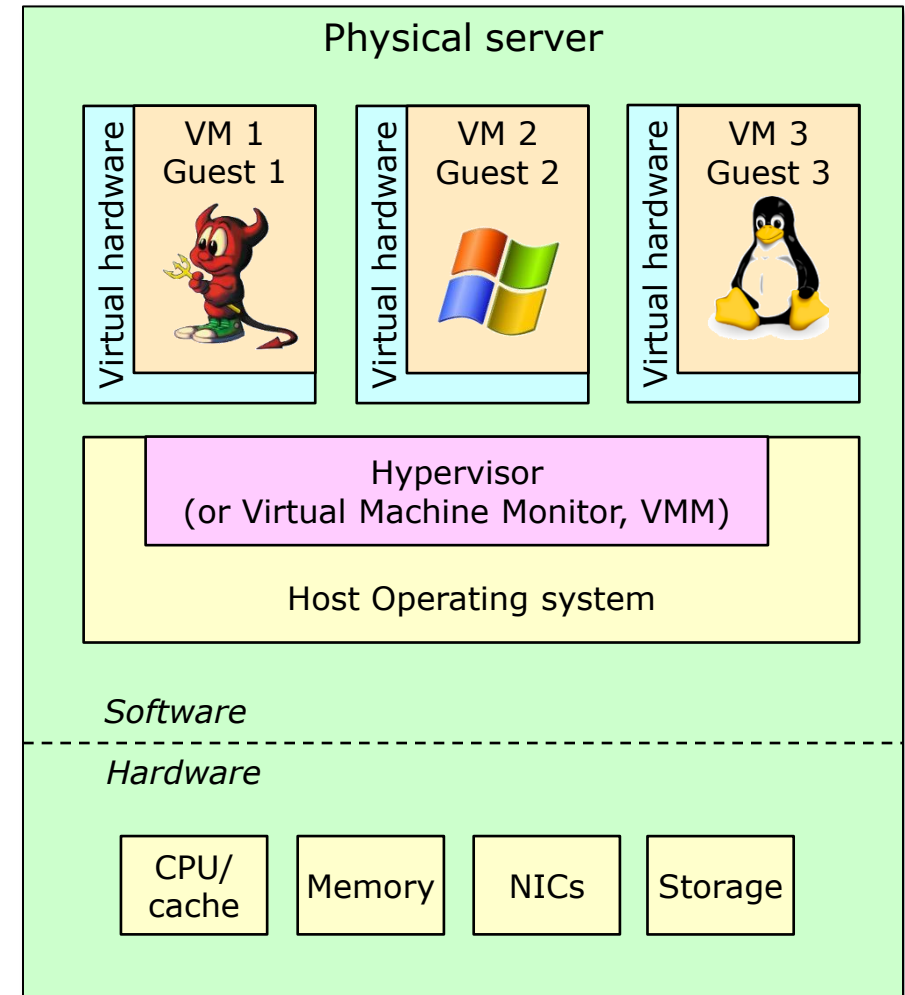


Operating system on virtual machines



Initial definitions

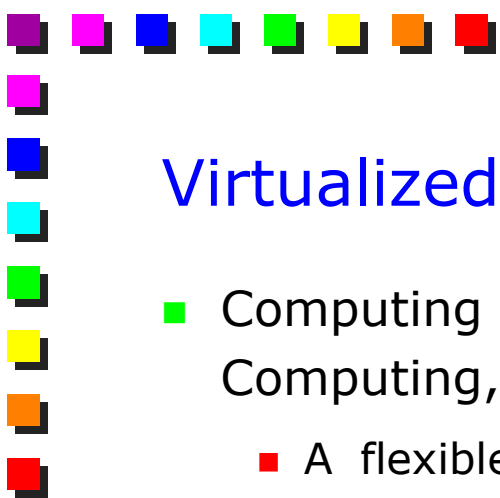
- Virtual Machine (VM)
 - Sw emulation of a physical machine that executes OS+apps such as being in a physical one
- Host OS
 - OS running on the physical machine, in charge of the virtualization of the hardware
 - It includes the Hypervisor (next slide)
- Guest OS
 - OS running in the VM, which should not be aware of running in a virtualized environment
- Hypervisor (or *Virtual Machine Monitor*, VMM)
 - Software in charge of the virtualization process; must virtualize the hardware resources, such as CPU, memory, and other devices (e.g., NICs)



Virtual hardware

- The “virtual hardware profile” defines the characteristics of the hardware provided to the Guest OS
 - The duty of the VMM is to “emulate” the virtual hardware, with the exact characteristics specified in the above hardware profile, i.e., allowing the Guest OS to run in the same way as the given hardware is really present
- The real hardware may have a little to do with the virtualized hardware
 - E.g., the server has *one* NIC from vendor X, and the VM profile specifies two NICs from vendor Y

General	
Name:	netlab-vm
Operating System:	Ubuntu (64-bit)
Settings File Location:	C:\Users\Fulvio\VirtualBox VMs\netlab-vm
System	
Base Memory:	4096 MB
Processors:	2
Boot Order:	Hard Disk, Optical, Floppy
Acceleration:	VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization
Display	
Video Memory:	64 MB
Graphics Controller:	VBoxVGA
Remote Desktop Server:	Disabled
Recording:	Disabled
Storage	
Controller:	SATA Controller
SATA Port 0:	netlab-vm.vdi (Normal, 15,00 GB)
Controller:	IDE
IDE Primary Master:	[Optical Drive] Empty
Audio	
Disabled	
Network	
Adapter 1:	Intel PRO/1000 MT Desktop (NAT)
Adapter 2:	Intel PRO/1000 MT Server (Internal Network, 'int0')
Adapter 3:	Intel PRO/1000 T Server (Internal Network, 'int0')



Virtualized components

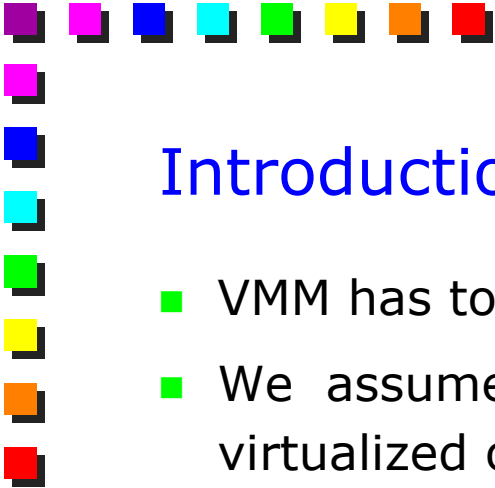
- Computing virtualization represents the most common approach for Cloud Computing, according to the IaaS model
 - A flexible way to share hardware resources between different (un-modified) operating systems
- This requires the VMM to virtualize:
 - CPU
 - Memory
 - I/O (e.g., NICs, storage, video, etc.)



Part II

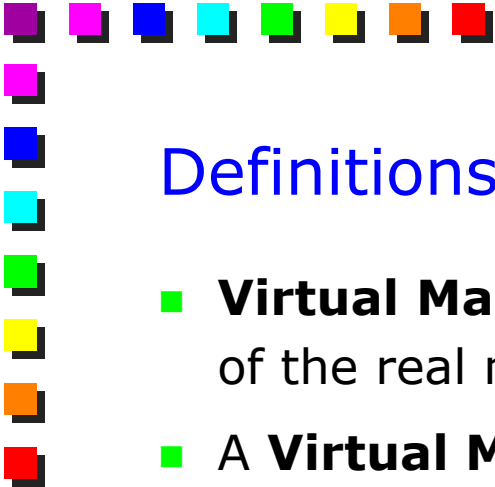
CPU virtualization





Introduction

- VMM has to assign one or more CPUs cores to the VM to execute the Guest OS
 - We assume that the CPU architecture of physical machine is the same of the virtualized one (e.g., Intel x86, x64)
 - Same Instruction Set Architecture (ISA) (or a subset) for host and VM
 - In case the ISA is different we need to rely on “emulation” instead of “virtualization”
 - It requires the binary translation between two completely different ISA, which is usually rather slow
 - Apple “Rosetta” used when transitioning from MIPS to Intel CPUs
 - [https://en.wikipedia.org/wiki/Rosetta_\(software\)](https://en.wikipedia.org/wiki/Rosetta_(software))
 - Although some CPUs are able to perform this binary translation automatically, this feature is not available on mainstream Intel architectures (e.g., x64)
- 

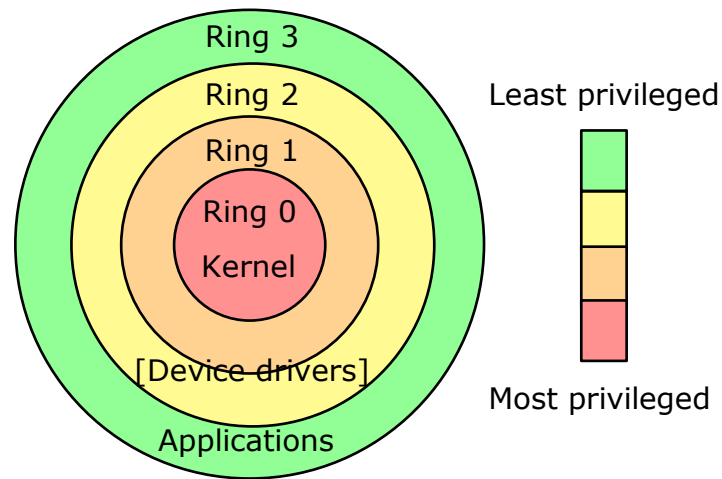


Definitions

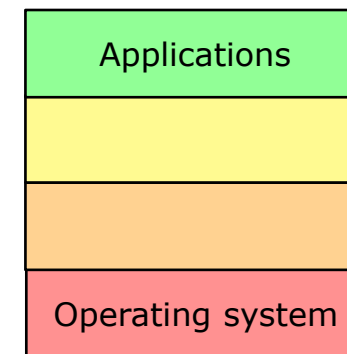
- **Virtual Machine:** “A virtual machine is taken to be an **efficient, isolated** duplicate of the real machine” (Popek&Golberg [1])
 - A **Virtual Machine Monitor** (VMM) must satisfy three characteristics:
 - First, it exports execution environments, VMs, essentially identical to the real machine
 - → Applications and OSes can run unmodified
 - Second, it executes efficiently the virtualized system
 - → A “statistically dominant portion of the virtual processor instructions should be executed by the real processor”
 - Third, the VMM should have the complete control of real system resources
 - → Each virtualized systems should be able to access only hardware resources under the authorization of the VMM
 - **Full Virtualization:** Guest OS has not to be aware of being virtualized
- 

x86 Hardware Privilege Rings

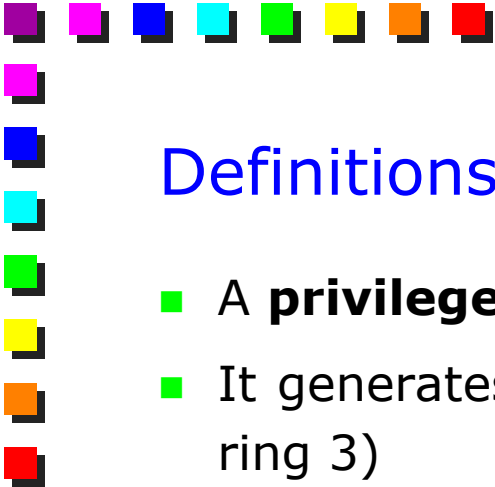
- x86 defines 4 rings with different privilege levels
 - **Ring 0:** designed for the most privileged part of code, the OS kernel
 - **Ring 1-3:** designed to run application and services with different level of trustworthiness
- Currently, only Rings 0 and 3 are used in modern OSs



Traditional ring structure
in Intel x86 CPUs



Traditional OS



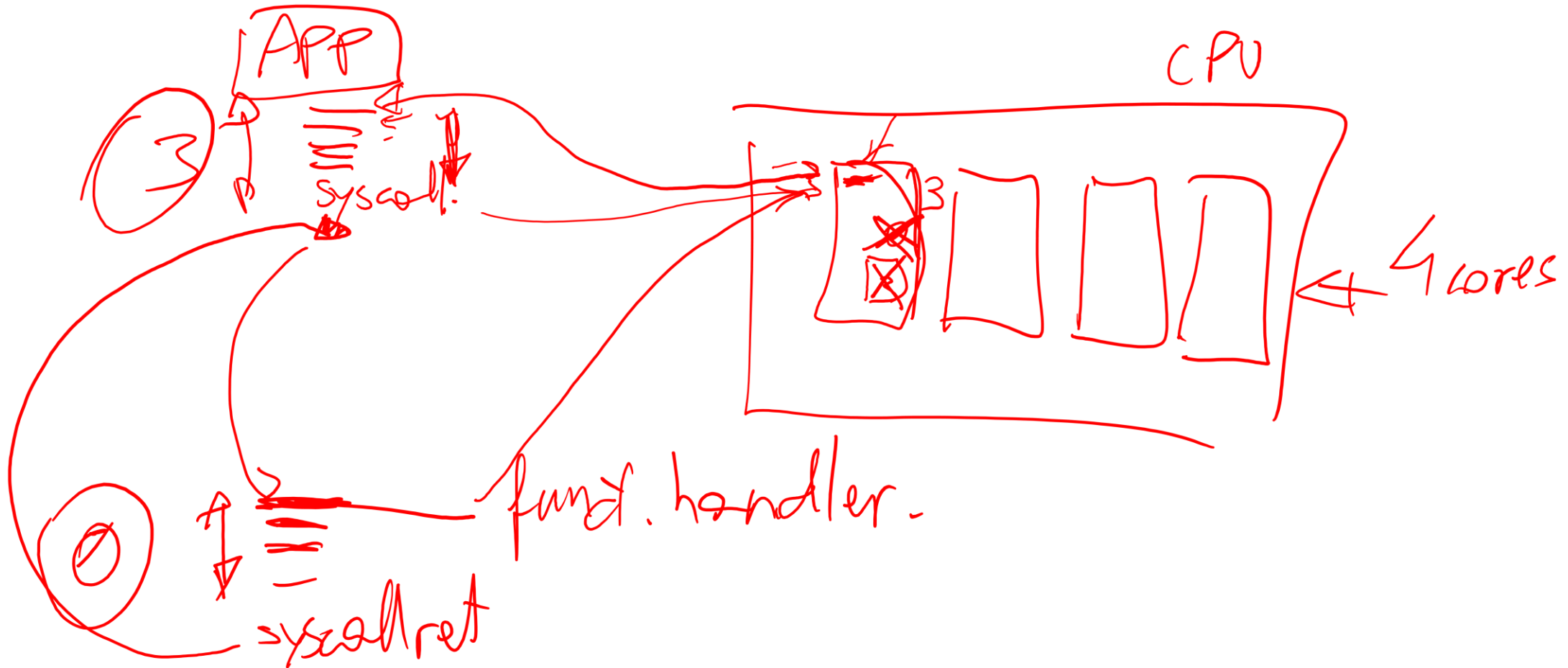
Definitions: Privileged instruction

- A **privileged instruction** needs to be executed in a privileged hardware context
 - It generates a trap if called when the CPU is running in the wrong context (e.g., at ring 3)
 - Examples of instructions that cannot be allowed in user-level applications:
 - "HALT" instruction: a user-defined applications cannot halt a computer from running
 - I/O instructions: a user-defined application cannot directly interact with a specific I/O device, unless the OS guarantees exclusive access to it
 - A privileged instruction cannot be executed by a guest OS running at a ring greater than 0
- 

How SysCalls and HW privileged instructions work (hand drawing)

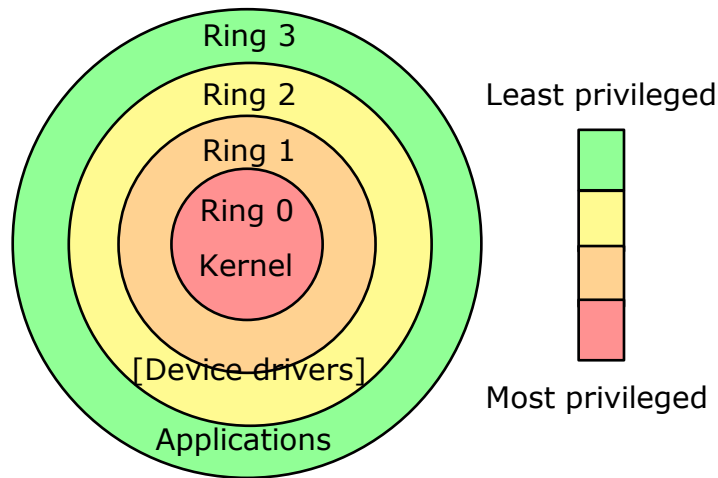
Apps → Syscalls → OS

OS → Hardware

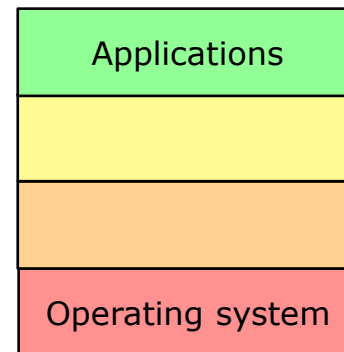


x86 Hardware Privilege Rings with virtualization

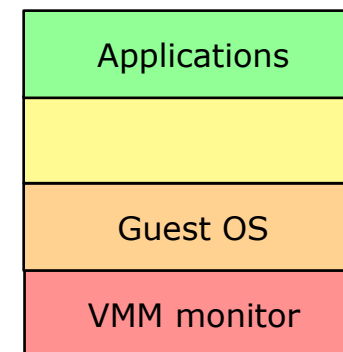
- Virtualization can use “ring de-privileging”, a technique that runs all guest software at privilege level greater than zero, but:
 - 0/1/3 model: in the x86 architecture, some privileges with respect to memory accesses are granted to 0-2 rings, hence the guest OS can interfere with the VMM
 - 0/3/3 model (“*ring compression*”): the above problem is solved, but the GuestOS is no longer protected from malicious applications



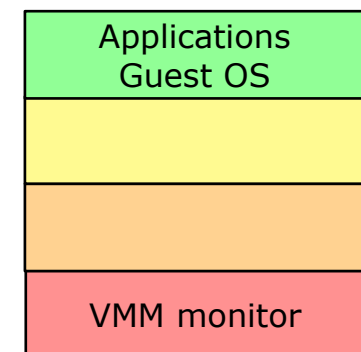
Traditional ring structure
in Intel x86 CPUs



Traditional OS



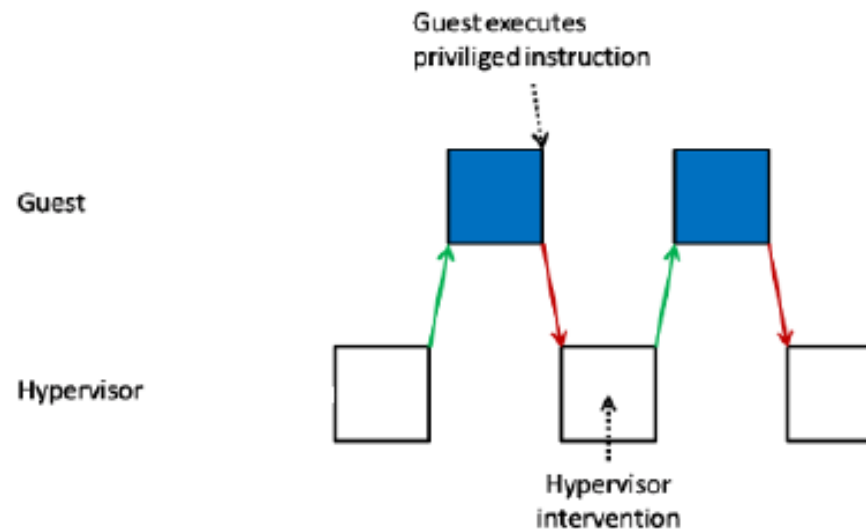
Virtualized OS
(0/1/3 model)



Virtualized OS
(0/3/3 model)

Traditional paradigm for virtualization: Trap & Emulate (T&E)

- Model proposed by Popek & Golberg in 1974
- Guest OS is executed in an unprivileged domain
- When Guest OS executes a privileged instruction, the processor launches a trap that is intercepted by the hypervisor (VMM)
- The VMM emulates the privileged instruction
- VMM gives control back to the Guest OS execution



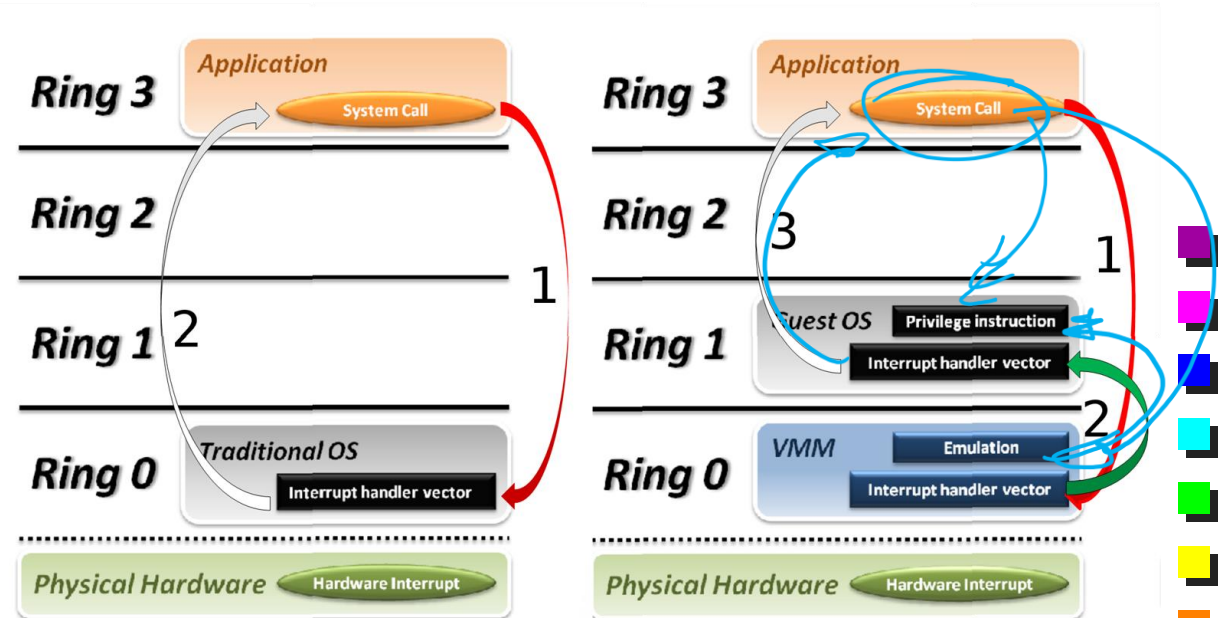


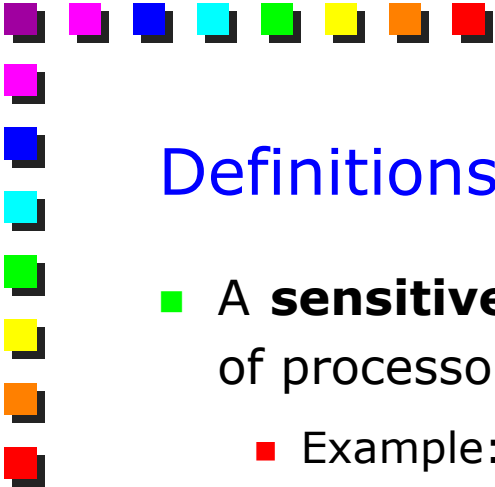
T&E and system calls (1)

- How are system calls (*syscalls*) implemented?
- The traditional (old) way:
 - Userland code (e.g., glibc) generates a software interrupt (instruction "INT xx")
 - The generic interrupt routine of the OS is started, which determines where to jump in the OS code to serve the above interrupt
 - The kernel jumps to the requested code, it serves the interrupt, then it returns to the caller (instruction "IRET")
 - This requires to load and parse the content of several memory locations, which is rather slow
- The new way: use SYSENTER/SYSEXIT (or SYSCALL/SYSRETURN in x64)
 - The userland code writes the memory address of the target kernel routine in a specific register, then calls SYSENTER
 - The kernel runs into a very fast transition, in which a few memory accesses are required (e.g., context switching, saving application state on the stack, etc)
- Very interesting reading: John Gulbrandsen, "System Call Optimization with the SYSENTER Instruction," 2004
 - <https://www.codeguru.com/cpp/misc/misc/system/article.php/c8223/System-Call-Optimization-with-the-SYSENTER-Instruction.htm>

T&E and system calls (2)

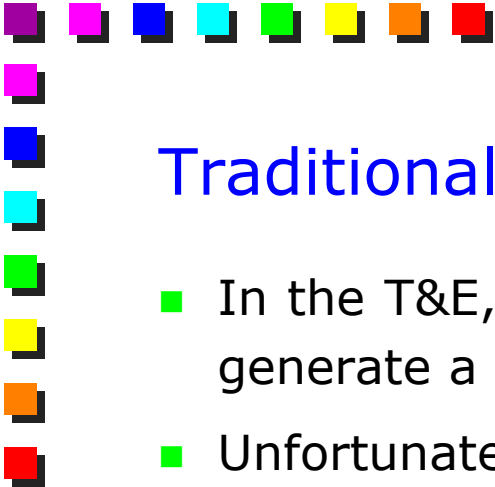
- VMM must intercept also system calls triggered by user apps (e.g., SYSENTER) passing the control to the guest OS
 - This system call cannot directly be intercepted by the guest OS due to the de-privileged ring of execution of guest OS (Ring 1)
- VMM has to intercept and give back control to the kernel of the guest
 - An extra context switch is introduced
 - System calls are always trapped and delivered to code running in Ring 0 (VMM in our case)
- The time to execute a single *syscall* could be 10x the time needed for the same call within the host OS





Definitions: Sensitive instruction

- A **sensitive instruction** is CPU instruction leaking information about physical state of processor
 - Example: the x86 PUSH %CS register can read the code segment selector (%cs) and learn its current privileged level
 - The Code Segment includes 2-bits for the Requested Privilege Level, which enables who executes this instruction to know which privileged level the software is currently running into
 - In order to be “easily” virtualizable, all *sensitive* instructions must be *privileged*
 - If not, the risk is that some programs can obtain information about the real hardware, and not on the actual (virtual) hardware they have
 - Very simplistic example: if an application can discover that it runs on a N-core CPU, it can create N threads to go faster. But what about the physical machine has N cores, but the VM has only a portion $M \ll N$? The application will not make the right decision (the optimal number of threads is M) and, in the end, will waste resources in un-necessary context switching.
- 



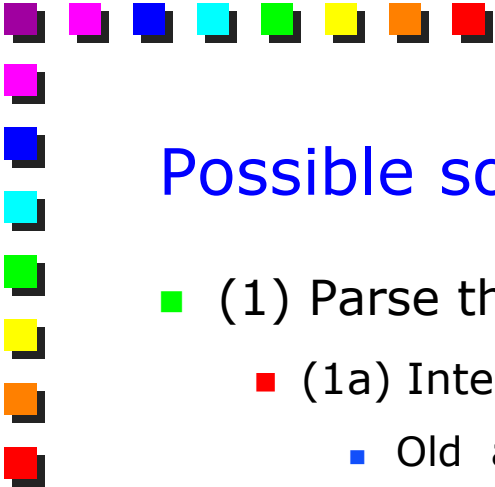
Traditional x86 architecture is not easily virtualizable (1)

- In the T&E, each privileged instruction, if executed in an unprivileged domain, must generate a trap detectable by the hypervisor
- Unfortunately, this is not true in the x86 architecture
 - X86 was conceived with a minimalistic approach, when the virtualization was NOT requested by customers
 - x86/x64 presents some sensitive instructions that do not trap if executed in an unprivileged level, such as *POPA*, *POPF* (i.e., not all sensitive instructions are privileged)
 - E.g., executing the x86 "PUSH %CS" instruction, the OS guest may know that is running at ring != 0



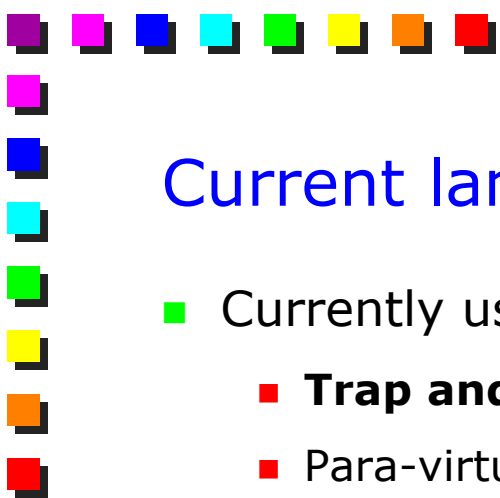
Traditional x86 architecture is not easily virtualizable (2)

- Therefore, the x86 architecture was said to be “non-virtualizable”
 - Therefore:
 - Several sensitive instructions are not detectable by the VMM
 - The VMM cannot emulate the correct behavior during the execution using the traditional T&E approach
 - Very nice readings about the problems present in the original x86 architecture:
 - “Intel Technology Journal,” Volume 10, Issue 03, August 2006. Available at <https://www.intel.com/content/dam/www/public/us/en/documents/research/2006-vol10-iss-3-intel-technology-journal.pdf>
 - “Bringing Virtualization to the x86 Architecture with the Original VMware Workstation,” ACM Transactions on Computer Systems, November 2012. Available at <https://dl.acm.org/doi/10.1145/2382553.2382554>
 - Much shorter paper: “Intel Virtualization Technology,” IEEE Computer Magazine, May 2005. Available at <https://ieeexplore.ieee.org/document/1430631>
- 

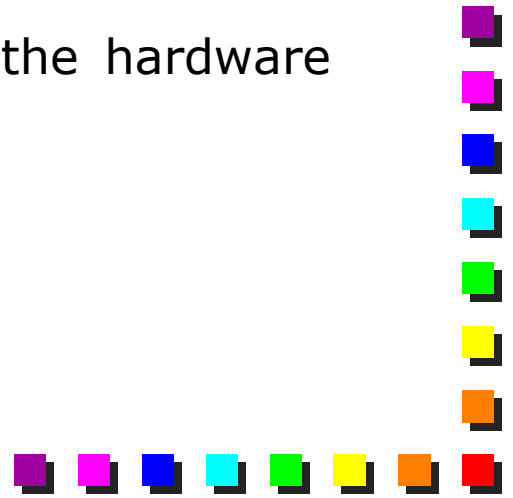


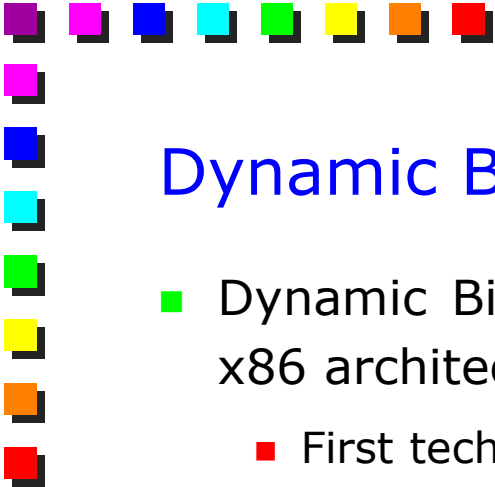
Possible solutions

- (1) Parse the instruction stream and detect all sensitive instructions dynamically
 - (1a) Interpretation (BOCHS, JSLinux)
 - Old and slow technique, not presented here; emulating a single ASM instruction may originate an overhead of at least one order of magnitude
 - (1b) Binary translation (VMWare, QEMU)
 - ☺ No OS source modification
 - ☹ Performance overhead
 - (2) Change the operating system
 - Paravirtualization (Xen, L4, Denali, Hyper-V)
 - ☺ Near-native performance
 - ☹ OS modifications
 - (3) Make all sensitive instructions privileged!
 - Hardware supported virtualization (Xen, KVM, VMWare): Intel VT-x, AMD SVM
- 



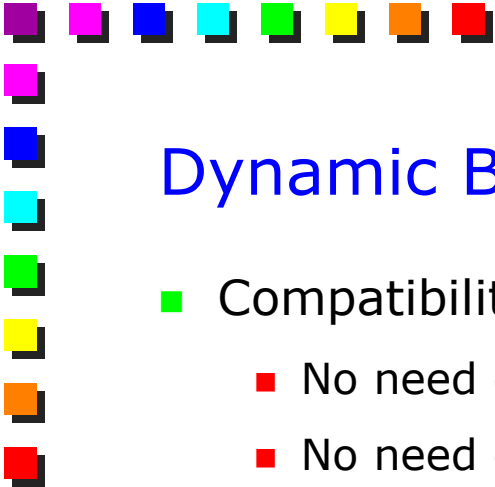
Current landscape

- Currently used virtualization techniques
 - **Trap and Emulate**
 - Para-virtualization (old XEN)
 - (dynamic) Binary translation (old VMWare)
 - **Hardware assisted**
 - 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
 - Access to “hardware devices” (MMIO / PIO)
 - Special instructions (CPUID, LGDT)
 - Access to interesting “registers” that have side effects
- 



Dynamic Binary Translation (1)

- Dynamic Binary Translation (DBT) was proposed by VMware in 1998 to overcome x86 architectural limits
 - First technique available on x86
 - Fully virtualized approach
 - Guest OS does not need to be modified
 - Enabled x86 virtualization without hardware assistance or modification of the source code
 - The idea: the VMM dynamically translates x86 “non-virtualizable” ISA in a virtualizable one at run-time
 - **Dynamic:** translation is done on the fly during execution and interleaved with code execution
 - **Binary:** VMM translates the *binary* code, not the *source* code
- 



Dynamic Binary Translation (2)

■ Compatibility

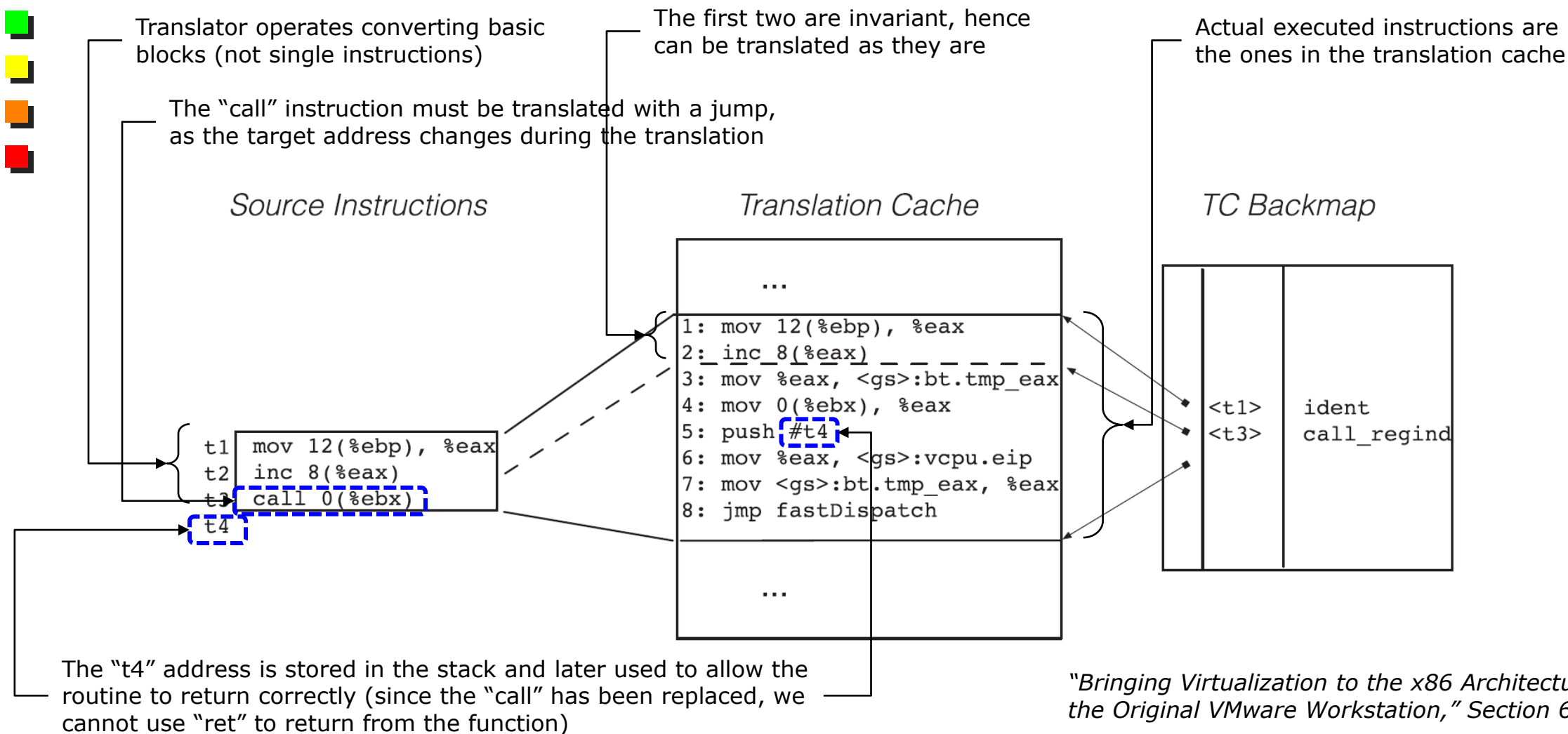
- No need of a specific HW support
- No need of a modified OS

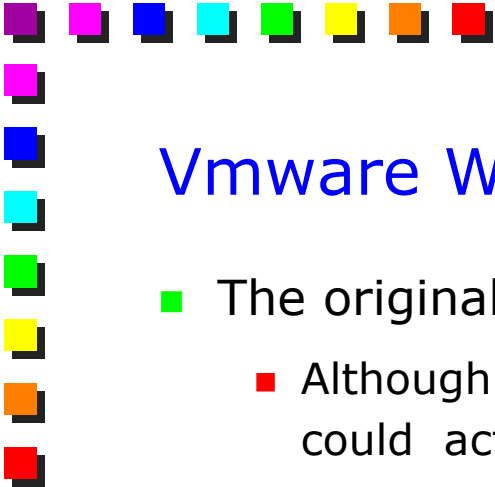
■ Performance

- Virtualization overhead is significantly higher than other techniques
 - Translation speed can be improved by adopting caching techniques to recognize (and translate) significant instruction patterns (assembly code “basic block”)
 - In fact, for better performance, the translation process was done on blocks of code and not on single instructions
 - Several instructions or execution patterns (system calls) are significantly slower than real execution
- 



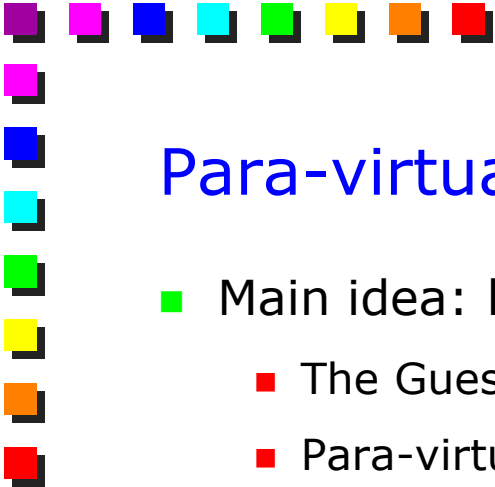
DBT example





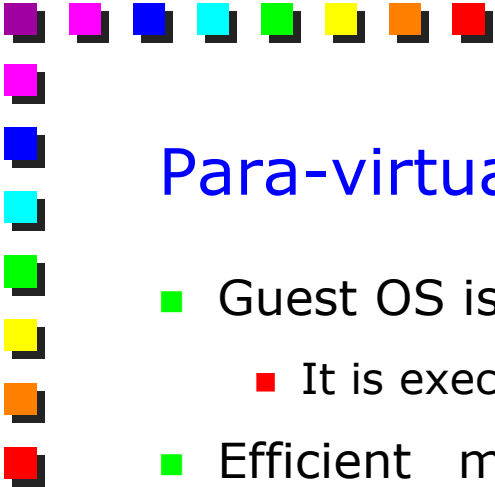
Vmware Workstation: T&E + DBT

- The original VMware VMM combined a T&E engine with a system-level DBT
 - Although trap-and-emulate could not be used to virtualize the entire x86 architecture, it could actually be used some of the time, e.g., during the execution of **application programs**, which accounted for most of the execution time on relevant workloads
 - According to this approach, *syscalls* can originate a huge performance penalty
- Execution privileges: Guest OS at Ring 1, VMM at Ring 0
 - VMM inspects dynamically the code of the Guest OS and replaces non-trappable portions of the code (*sensitive instructions*) with “safe” instructions
 - In some case, the resulting code requires the intervention of the VMM (e.g., interacting with the hardware)
 - In other cases, the resulting code is able to perform all its operations without the VMM intervention
 - E.g., access to GuestOS structures, while “normal” (privileged) instructions would return the equivalent structure present at VMM level (ring 0)



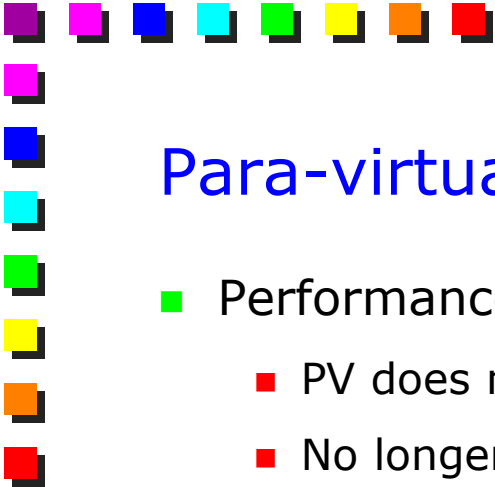
Para-virtualization (1)

- Main idea: let the guest know that it is running in a VM
 - The Guest OS knows that, in some cases, it has to give the control to the VMM
 - Para-virtualization gives up to the “Full Virtualization” requirements
- OS Guest is explicitly modified to be virtualizable, changing the interface provided to make it easier to implement
 - System calls are replaced with specific hypervisor calls (hypercall)
 - Non-virtualizable instructions are replaced with hypercalls
 - They do not produce traps
 - No more trap&emulate
 - Also access to the page tables is para-virtualized
 - ABIs of guest OS will not change, hence application could be executed without any modification



Para-virtualization (2)

- Guest OS is explicitly de-privileged
 - It is executed in RING 1
 - Efficient mechanisms are introduced in the Guest OS kernel to ease the communication with the hypervisor:
 - Guest-to-Hypervisor: privileged instructions are replaced with synchronous para-virtualized equivalents (hypercalls)
 - Hypervisor-to-Guest: hypervisor could notify certain events asynchronously to the guest
 - Like interrupts, Guest OS could defer their handling
 - PV guest does not need to start in 16 bits mode or go through the BIOS
 - PV boots directly the kernel in the desired mode (32 or 64-bits)
- 



Para-virtualization (3)

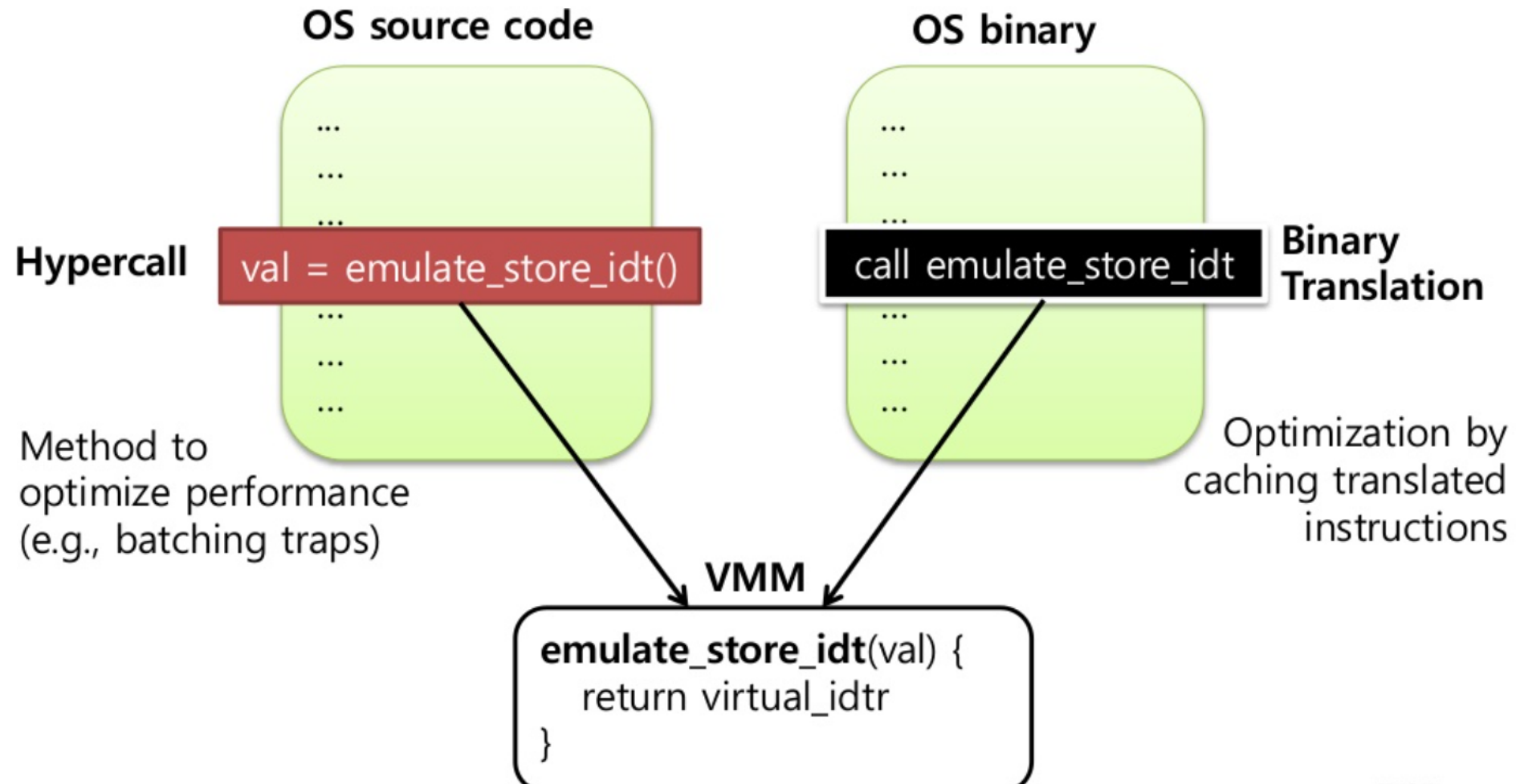
■ Performance

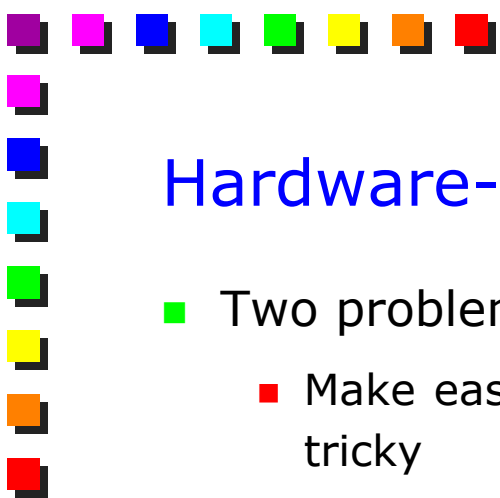
- PV does not need to emulate motherboard or device bus
- No longer dynamic translation; the OS has to be patched ahead of time
- The drop of “segmentation” memory protection mechanism in x86-64 impact performance on this architecture
- Implementation is simpler, hence much faster, than DBT

■ Compatibility

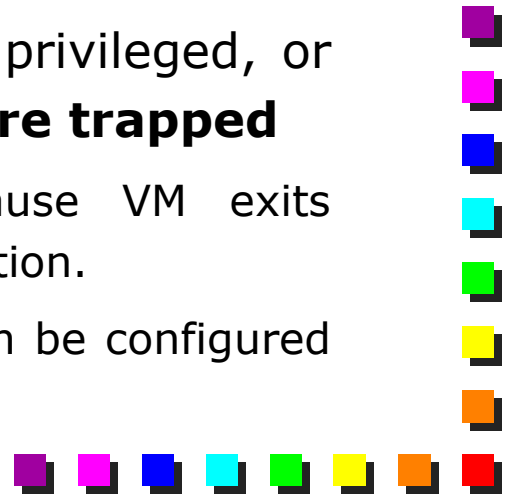
- Only modifiable OS could be virtualized
 - No Windows
 - OK Linux and BSD
- 

Source-level vs. binary-level modifications



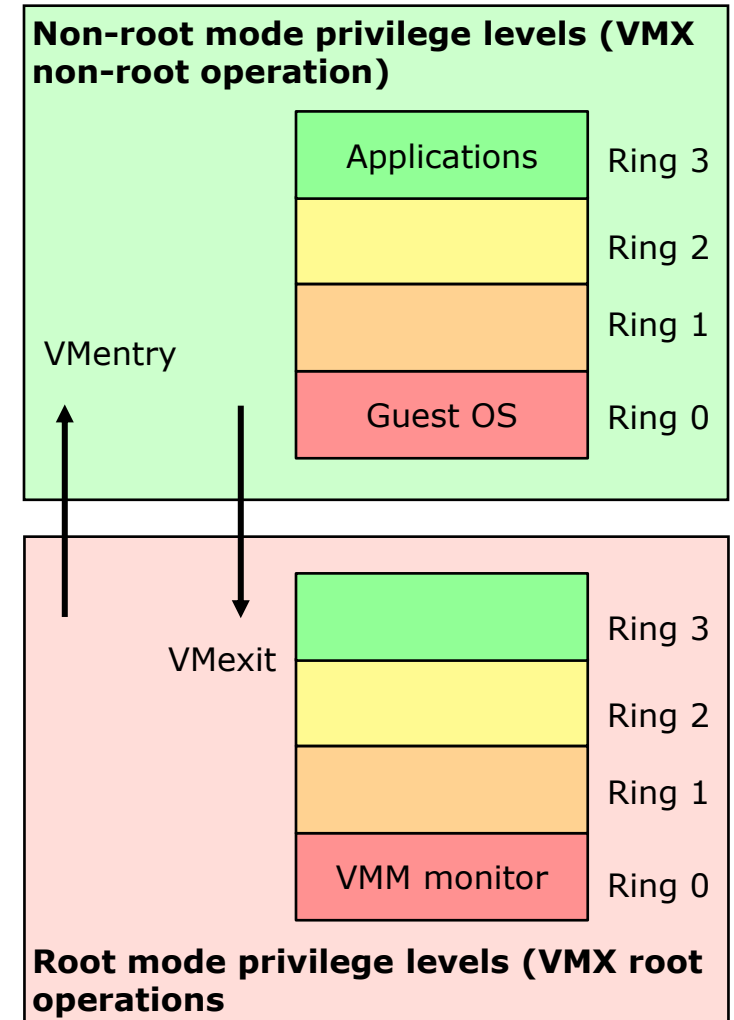


Hardware-assisted virtualization (HVM) (1)

- Two problems still open:
 - Make easier the implementation of a hypervisor: even in PV, the VMM is still complex and tricky
 - Provide efficient x86 Full Virtualization: several OS could not be modified and DBT is slow
 - HVM aims at providing a solution to those problems by proposing an efficient “Trap & Emulate” approach to virtualization thanks to additional hardware support
 - First implementation of Intel VT-X / AMD SVM in 2005
 - Functionally equivalent implementation in Intel and AMD CPUs, but incompatible
 - Idea: avoid sensitive instructions, either because are “promoted” as privileged, or because the **VMM can dynamically configure which instructions are trapped**
 - Some instructions (e.g., INVD, Invalidate CPU Internal Caches) cause VM exits unconditionally and therefore can never be executed in VMX non-root operation.
 - Other instructions (e.g., INVLPG, Invalidate TLB Entries) and all events can be configured to do so conditionally, using the VM execution control fields in the VMCS
- 

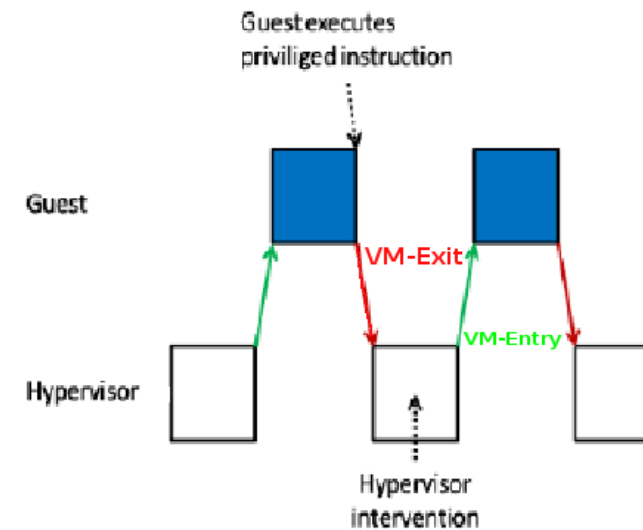
Hardware-assisted virtualization (HVM) (2)

- Processor has a new running mode (VMX root /non root)
 - Virtual Machine eXtensions
 - When the guest executes an instruction that would either violate the isolation of the VMM, or that must be emulated via SW, the HW can trap it and switch to the VMM
- When VMX is turned on, the CPU enables two different running modes, called "Operating Levels", equivalent to a new set of rings
 - Root Mode: higher class of privilege
 - Non-Root Mode: keeps existing four rings
- The VMM runs in fully privileged Root Mode
- Guest OS run in VMX non-root operating level in Ring 0
- Guest applications in VMX non-root Ring 3



HVM

- Intel's VTx allow a VMM to specify several conditions
 - According to this configuration, the actions attempted by a Guest VM will get "trapped"
- When a trapping condition is triggered, VMM takes the execution control, and it emulates the correct behavior
- Transition between Root and Non-Root levels are called:
 - VM entry: VMM to guest transition
 - VM exit: guest to VMM transition
- Registers and address space swapped in a single atomic operation
- Those transitions represent the main source of overhead





Cost for VMEntry/VMExit

Microarchitecture	Launch Date	Cycles
Prescott	3Q2005	3963
Merom	2Q2006	1579
Penryn	1Q2008	1266
Nehalem	3Q2009	1009
Westmere	1Q2010	761
Sandy Bridge	1Q2011	784

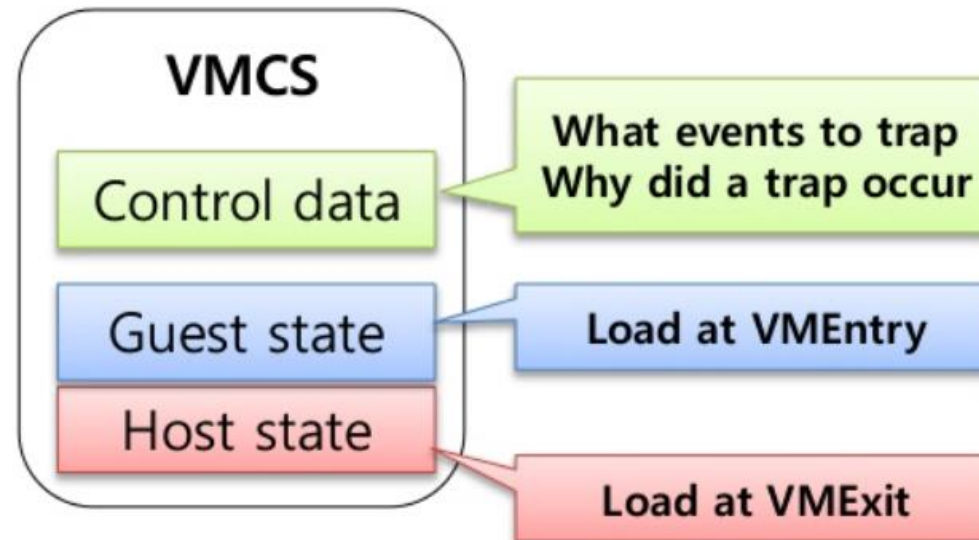
Table 1: Hardware round-trip latency.

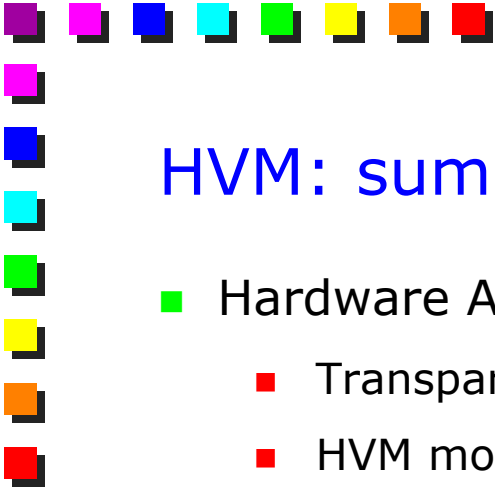
Agesen Ole, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. "Software techniques for avoiding hardware virtualization exits." In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pp. 373-385. 2012. Available at <http://www.cs.columbia.edu/~cdall/candidacy/pdf/Agesen2012.pdf>



Virtual Machine Control Structure (VMCS)

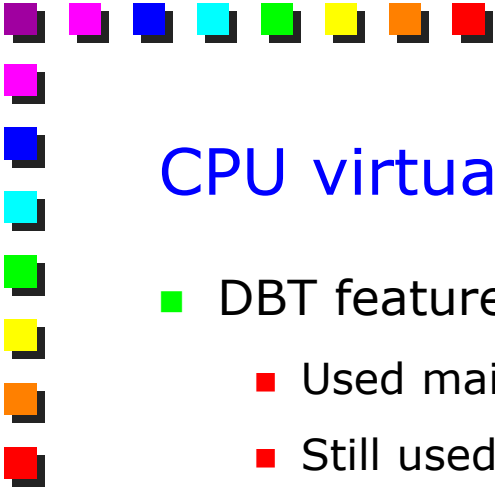
- VT-x (SVM) introduces a new memory structure, VMCS (VMCB)
 - It mirrors all register modifications needed to set a certain configuration in the guest OS
 - Concretely it represents the control panel of the VM storing information about:
 - Guest state
 - Host Processor information
 - Control data (e.g., trapping conditions)
 - Introduced dedicated instructions to modify it (VMWRITE / VMREAD)





HVM: summary

- Hardware Assistance technology greatly simplifies VMM implementation
 - Transparent way to make "Full virtualization"
 - HVM mode minimizes VMM intervention
 - No VM Entry/VM Exit for system calls triggered by user applications
- HVM reduces virtualization overhead
 - Round-trip time for VM Entry/VM Exit represents the most important source of virtualization overhead
 - CPU manufactures are pushing for more efficient VM entry/exit to reduce the cost of this operation



CPU virtualization: summary

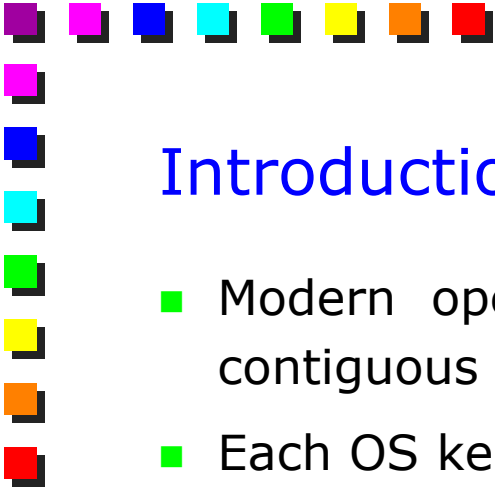
- DBT features the best compatibility, but it is slow
 - Used mainly for legacy hardware without HVM
 - Still used nowadays e.g., in desktop virtualization (Virtualbox)
- Paravirtualization could achieve great performance, but it could virtualize only a subset of OS
 - Fully implemented only for Linux and BSD guests
- HVM is the most used technique nowadays
 - Performance could vary a lot according to CPU generation



Part III

Memory virtualization

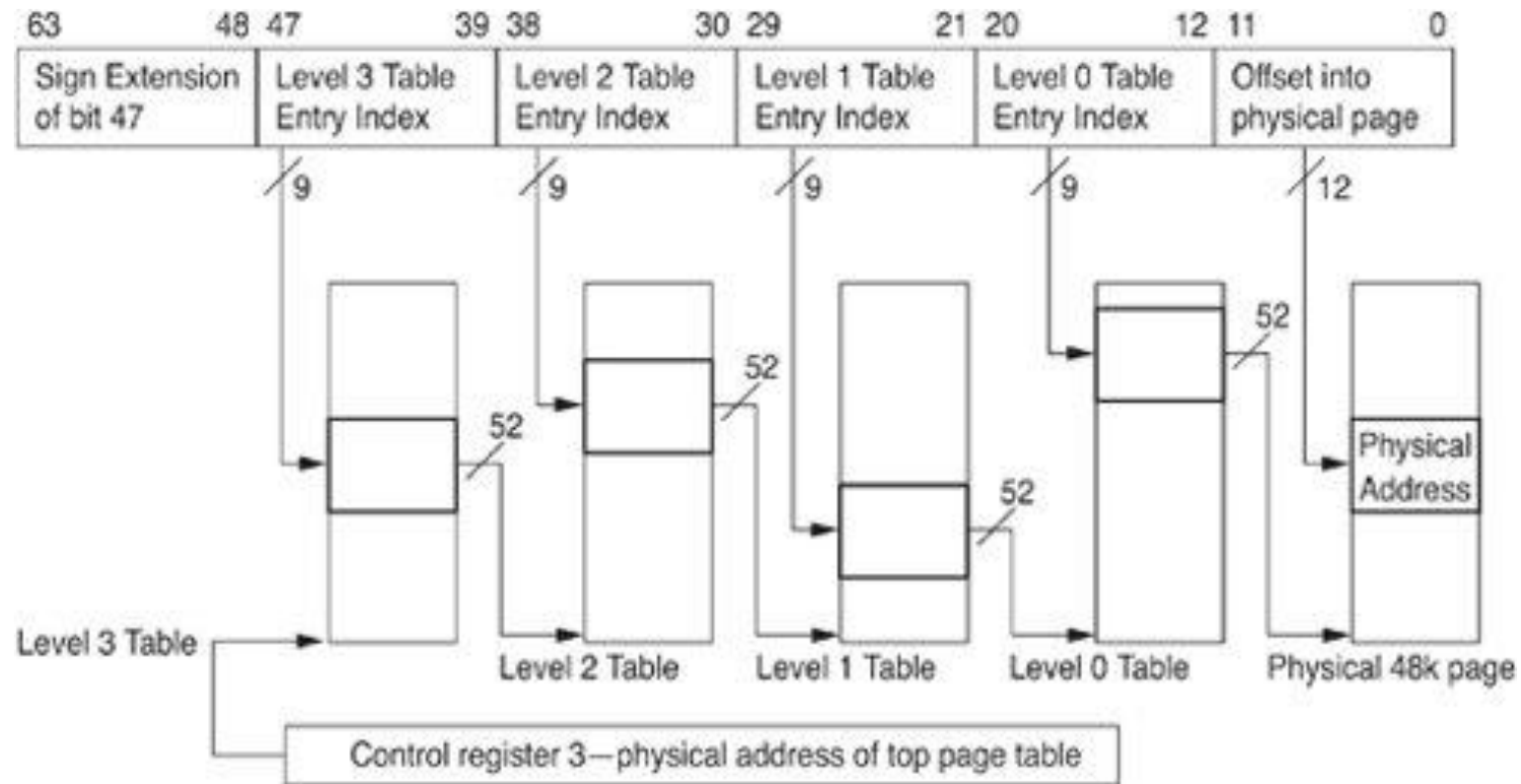




Introduction - Pagination

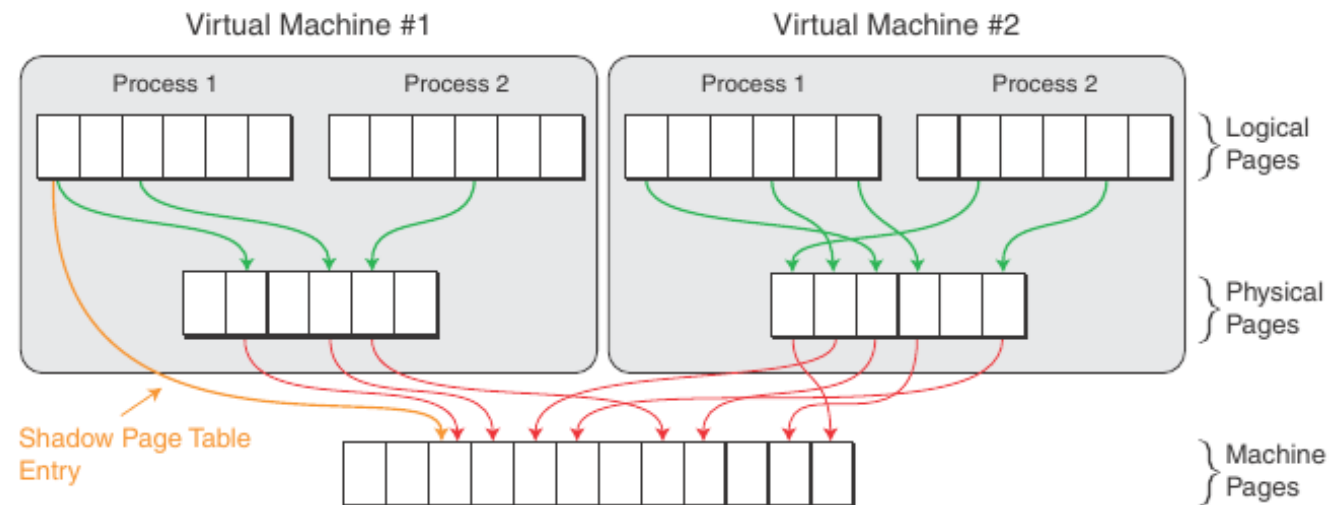
- Modern operating systems use “memory paging” as a technique to access as contiguous dispersed locations in the physical memory (Virtual Memory)
- Each OS keeps a set of tables to translates the virtual memory addresses to physical addresses
- X86/x64 can handle this task in hardware with a dedicated Memory Management Unit (MMU)

x64 Memory Paging



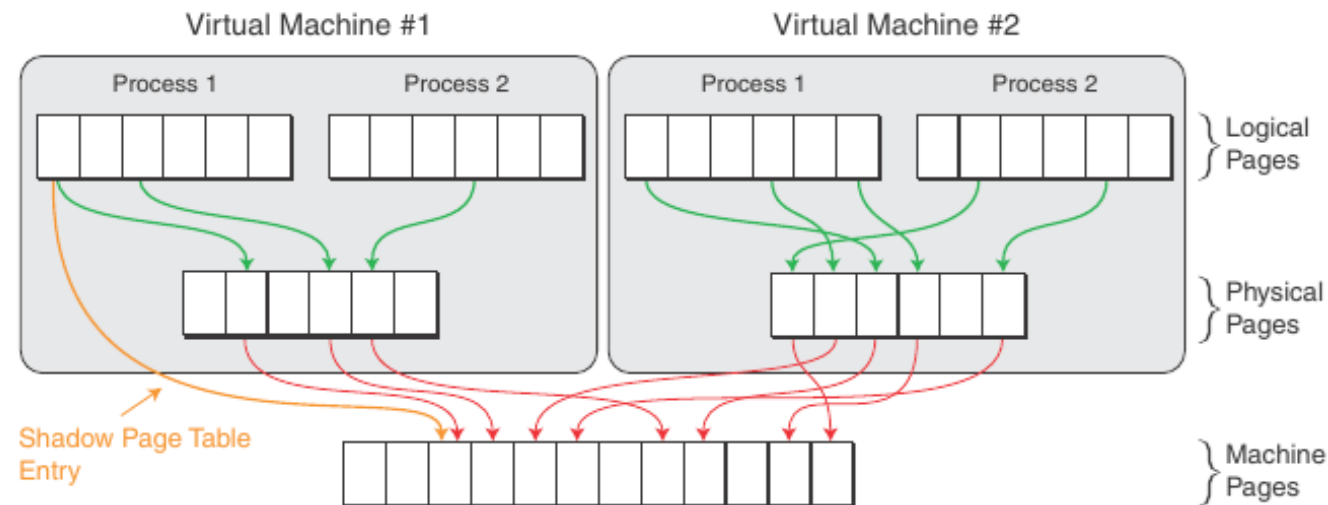
Memory Virtualization: Shadow Page Table (1)

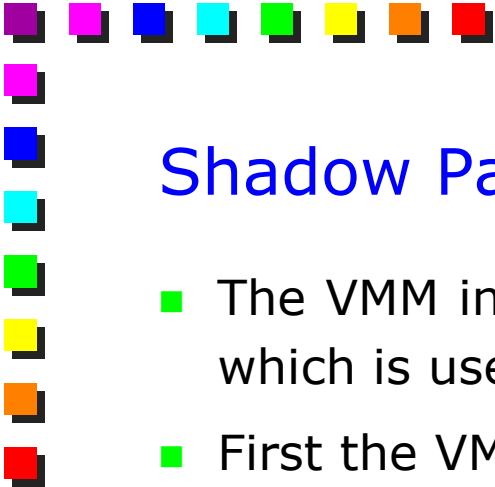
- An extra level of translation is required with VMs
 - Guest virtual address → Guest physical address → Machine physical address
- To avoid two address translations, a “Shadow Page Table” is introduced
 - The shadow page table stores and keep track of the mapping between Guest logical address and Machine physical pages
 - It is invisible from the guest point of view and it will be used by the CPU for a translation, when the guest is active



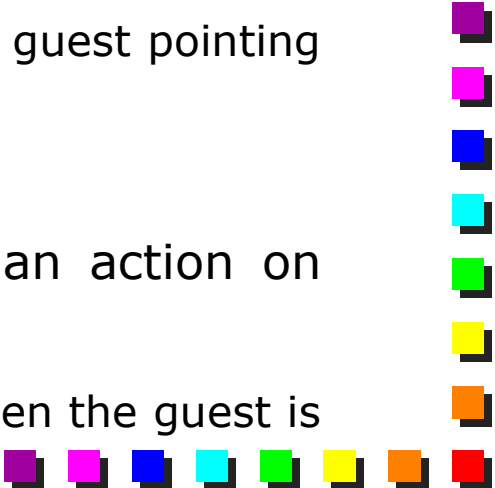
Memory Virtualization: Shadow Page Table (2)

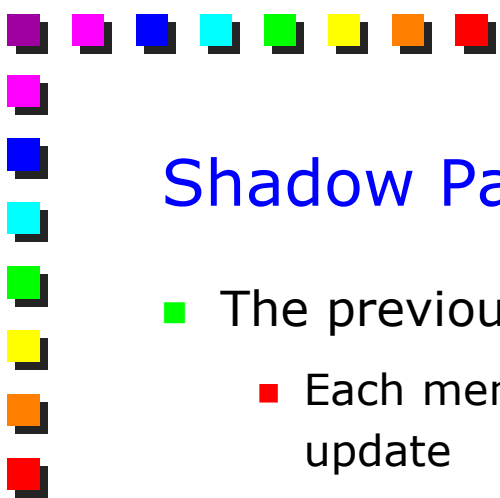
- The VMM is in charge of keeping the Shadow Page Table synchronized with the guest OS page table
 - OS will modify its ordinary page tables
- An important extra overhead is introduced in presence of page faults
 - An extra page table has to be kept synchronized when Guest update its own page tables → this require an extra VMExit





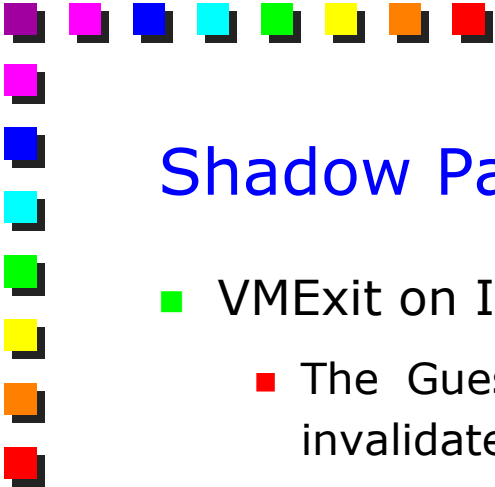
Shadow Page Tables – Brute Force Approach

- The VMM intercepts all the action performed on Guest Page tables and CR3 register which is used to point the Page Descriptor table.
 - First the VMM sets the VMCS to cause a VM-Exit each time that CR3 is modified
 - When it takes control upon a write of CR3, it can emulate the behavior
 - example: MOV %EAX, %CR3
 - a new page-table descriptor is going to be set up by the guest operating system
 - The VMM reads the Guest page tables set up by the guest and create the corresponding host page tables by translating them into physical addresses.
 - It writes its desired value into CR3 and then it gives back the control to the guest pointing to the next instruction (EIP+4)
 - The guest resumes the execution
 - The guest pages are set in read-only and when the guest perform an action on memory will cause a VM-EXIT.
 - This lets the VMM intervene loading the correct value in host page tables when the guest is trying to modify the guest page tables.
- 



Shadow Page Tables – Optimization

- The previous approach had too much overhead
 - Each memory access will cause a VM Exit when the Guest is trying to perform a page table update
- Adopting a lazy approach, to limit the amount of VM Exits
- Still trap in case of writes on %CR3
 - VMM should be noticed by the Page Descriptor initialization (Trap and Emulate)
- No write protection on the guest page tables
 - Guest Page Tables can be out of sync with Host Page Tables

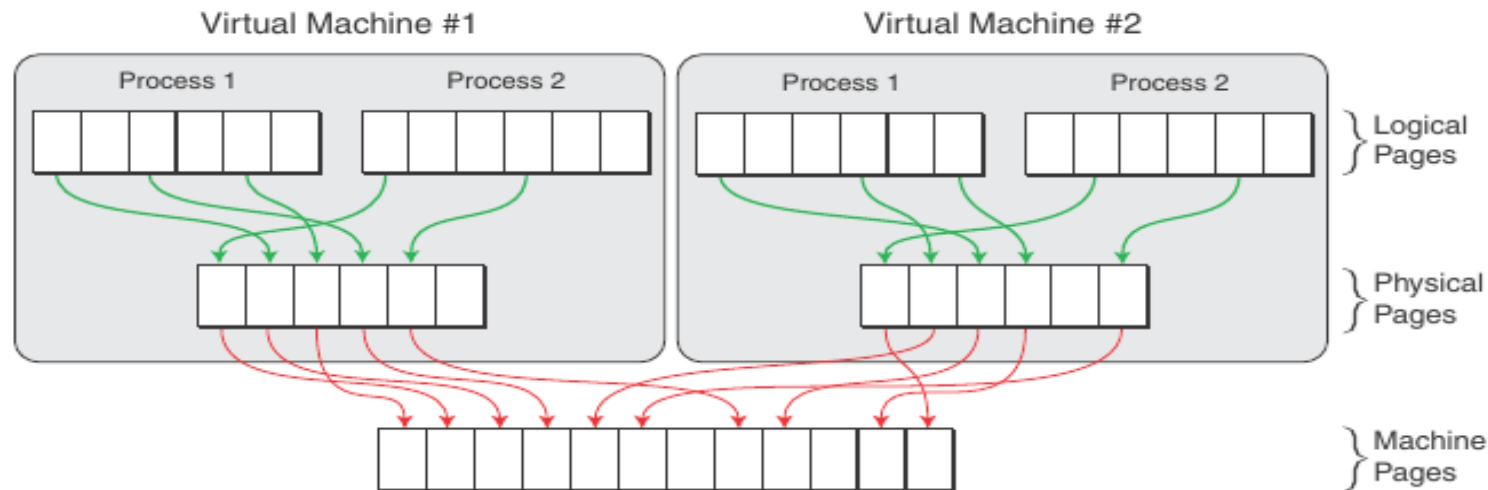


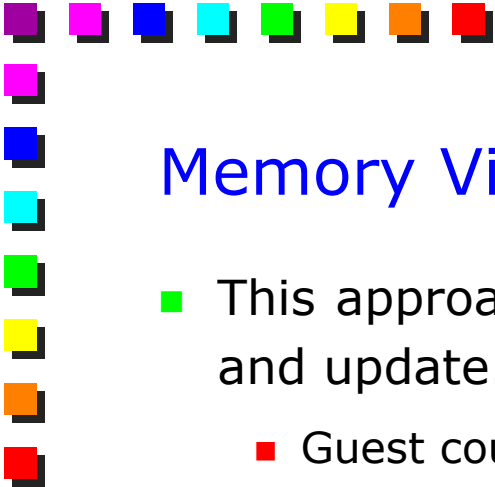
Shadow Page Tables – Optimization

- VMExit on INVLPG (TLB Flush)
 - The Guest OS removes some memory pages and updates the guest page tables and invalidate the TLB
 - But the Host Page tables still marks that page as present in memory
 - Trapping the TLB flushing, the VMM will mark as «not present» pages unloaded by the VMM
 - VMExits on Page Faults
 - VMM Trap and Emulate the absence of a page in memory (Seen by the guest as available in its page tables)
 - After the VMM have loaded the page, it resumes the guest execution by setting the guest %EIP to re-run
- 

Memory Virtualization: EPT/NPT (1)

- In order to avoid Shadow Page Table overhead, Intel/AMD introduces Extended/Nested Page Table
 - Different technologies implementing the same concept
- Traditional page tables translate from guest logical to physical address
- In addition, a separate set of page tables, the EPT, translates from guest physical address to machine physical address

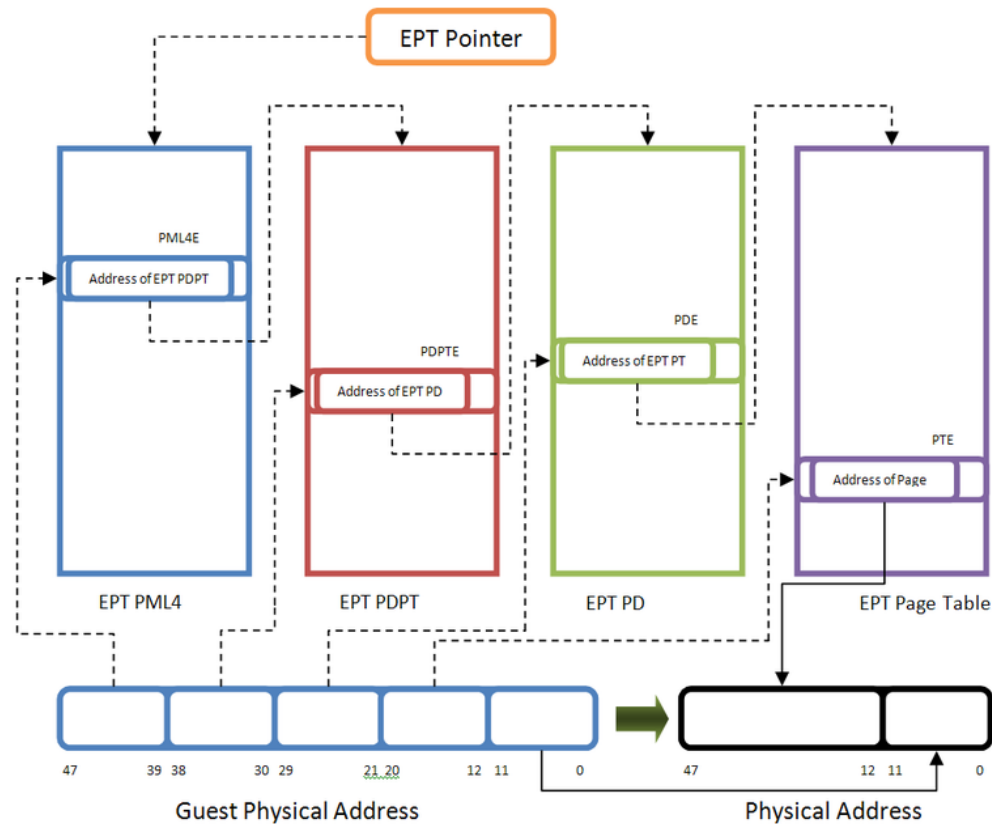




Memory Virtualization: EPT/NPT (2)

- This approach removes the need of VMExit associated with page table virtualization and update.
 - Guest could now keep update its page table without any overhead
- EPT/NPT introduce a “double-level” page walk
 - This increases the cost of a single page walk
 - TLB cache becomes critical to guarantee good performance
 - For memory intensive task, the utilization of larger pages of memory (“Huge Pages”) could increase the TLB hit ratio thanks to the two TLB present in Intel processors
 - One TLB for normal pages, one for huge pages

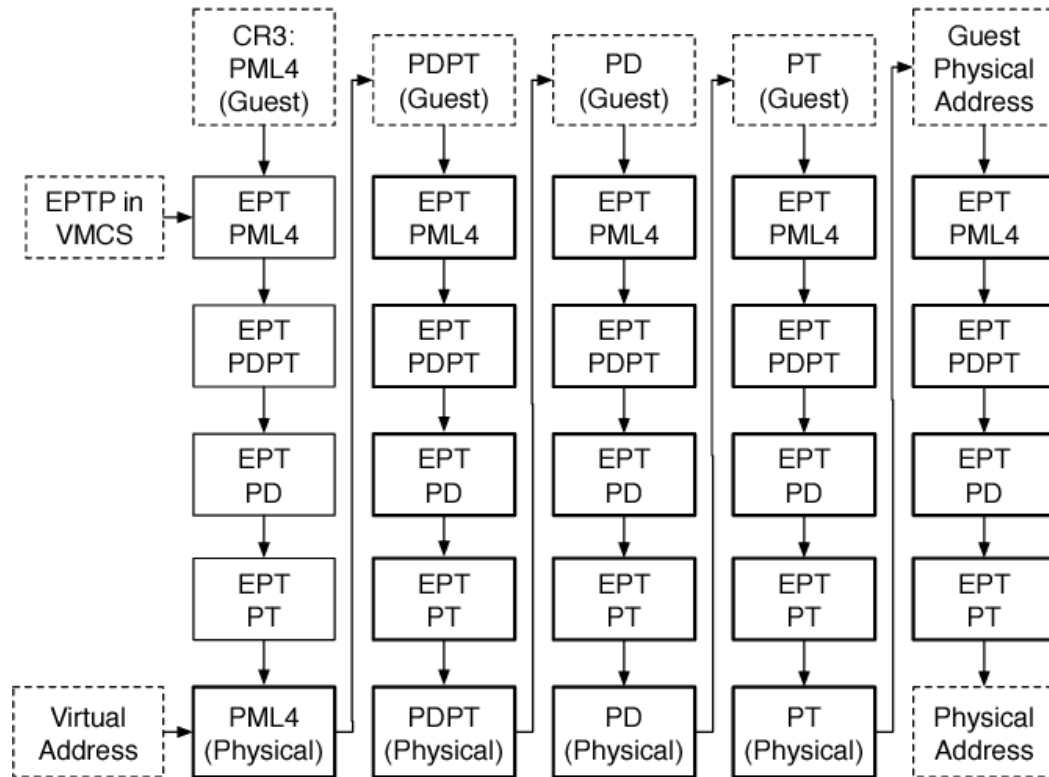
X64 – EPT Page Walk



EPT Address Translation [6]

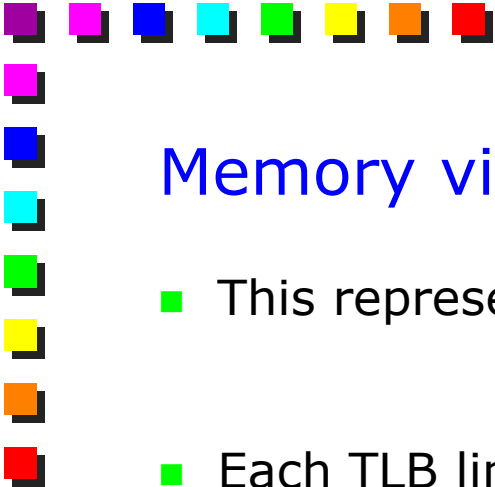
- Two layer page walk
 - Multiple memory access for every address that has to be translated
- The Guest Physical address should be translated again
 - For each part of this nested translation we have to pay further «hidden» memory access

X64 – EPT Page Walk



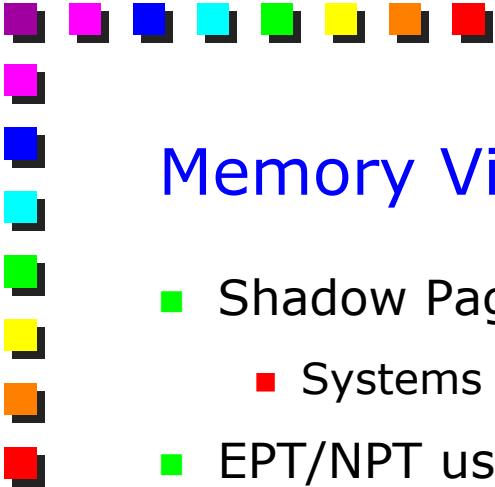
EPT Memory Access Summary [6]

- Two layer page walk
 - Multiple memory access for every address that has to be translated
- For each part of the address translation, the addresses of different levels Page Tables should be translated to the physical address
- For each memory access:
 - $(2 + 1) \times 2 = 6$ additional memory in x86 (2 Level of Paging)
 - $(4 + 1) \times 4 = 20$ additional memory access in x64
- “EPT provides performance gains of up to 48% for MMU-intensive benchmarks and up to 600% for MMU-intensive microbenchmark” [5]



Memory virtualization – Tagged TLBs

- This represents an additional optimization of EPT
 - Each TLB line has an identifier (Virtual Processor ID) for each virtual processor
 - When a line has to be accessed, the ID prevents wrong access to other virtual processors cache lines
 - This technique allows several virtual processors coexist on the TLB at the same time
 - The tagged TLBs eliminate the need for TLB flushes on every VMEntry and VMExit
 - Great importance with EPT/NPT where TLB page miss cause an important overhead
- 



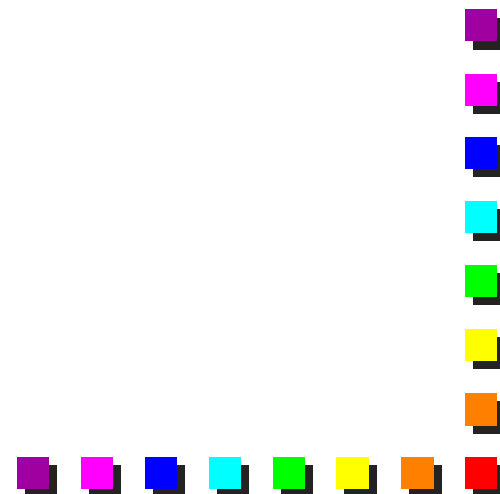
Memory Virtualization: summary

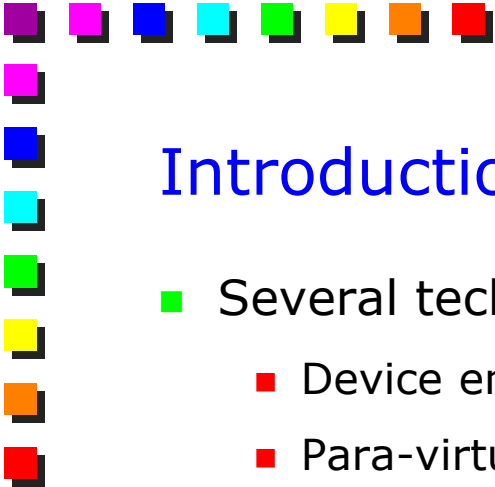
- Shadow Page Table technique used only in absence of EPT/NPT support
 - Systems with the first generation of hardware support do not have NPT/EPT technology
- EPT/NPT used whenever is possible
 - Bigger pages could be used to limit the pressure on the TLB cache



Part IV

I/O virtualization



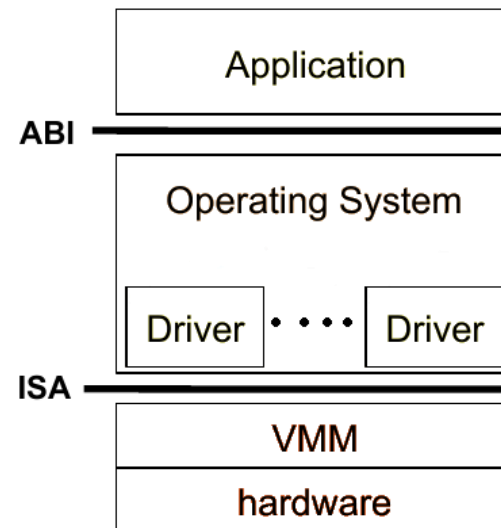


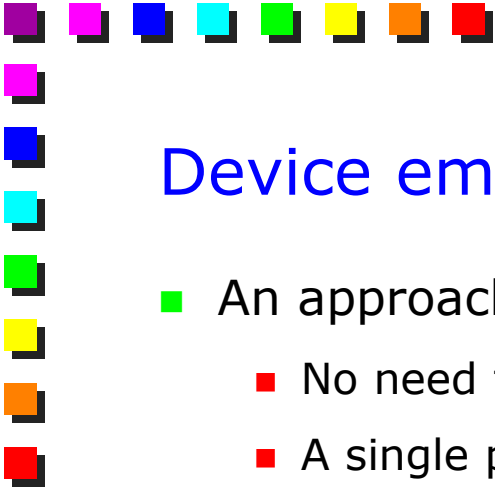
Introduction

- Several techniques could be adopted for virtualizing I/O devices
 - Device emulation
 - Para-virtualized device
 - Direct assignment
- The choice of the technique that has to be used depends on:
 - The type of device
 - Shared/Dedicated to a single Guest OS

I/O virtualization: device emulation

- VMM proposes to the Guest OS an emulated device, which implements in software an hardware specification
- Guest OS completely ignore that the device is emulated
- Guest OS will use the same driver used with an equivalent physical device.
- VMM has to remap device communication with the physical device in real OS



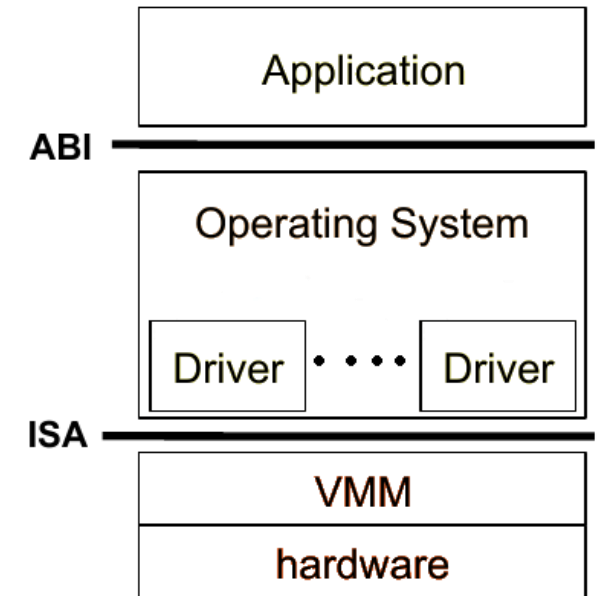


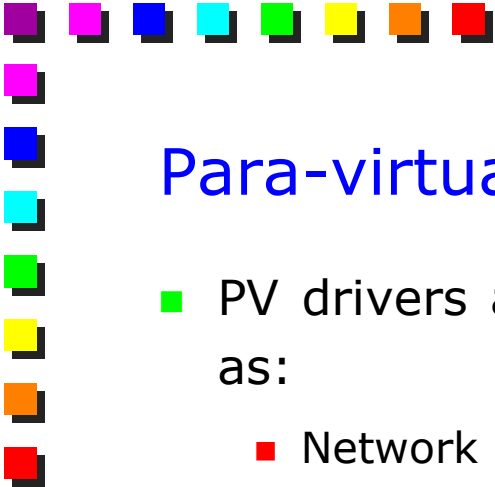
Device emulation: pros and cons

- An approach simple and easy to set up
 - No need to install any dedicated driver to handle device I/O operations
 - A single physical device could be multiplexed with several emulated device
- However:
 - I/O operations are generally slower than physical ones, with higher latency
 - Particularly critical in case of devices with high I/O (NIC, disk)
 - I/O operations may increase substantially the CPU load
 - This represent an extra-work that is completely absent in case of a physical device

I/O virtualization: para-virtualized device

- Guest OS is enriched with dedicated drivers
- Like para-virtualization
 - The main difference is that para-virtualization needs core modules of the kernel to be modified, hence, to patch the kernel
 - Instead, para-virtualized drivers could be added as external modules to the most part of modern OS
 - Here we require dedicated device drivers to be added/replaced



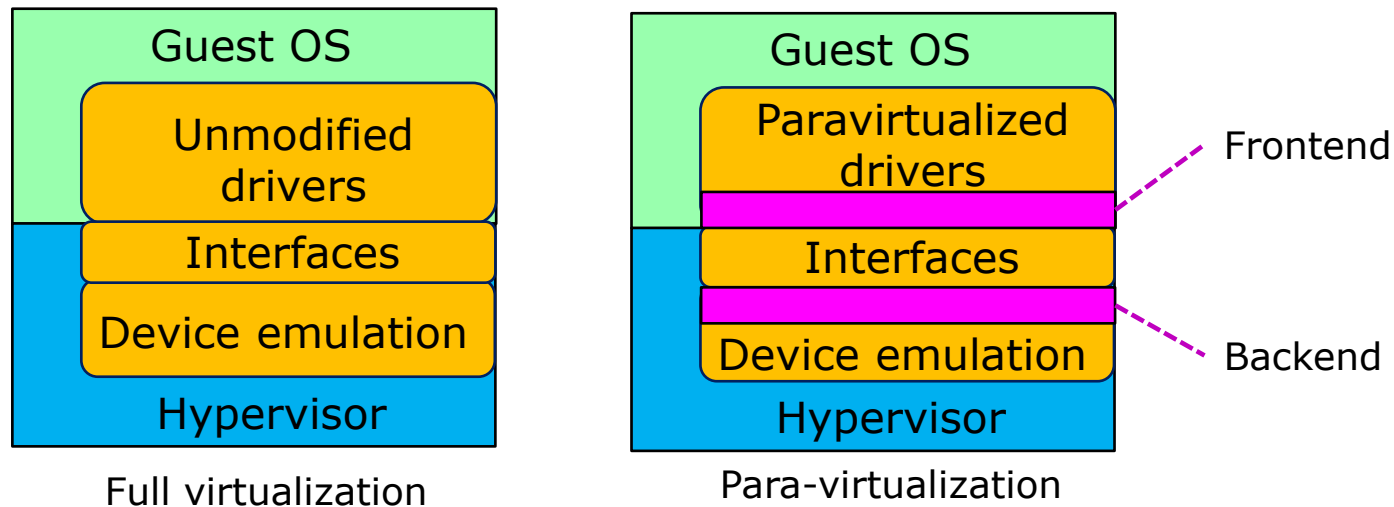


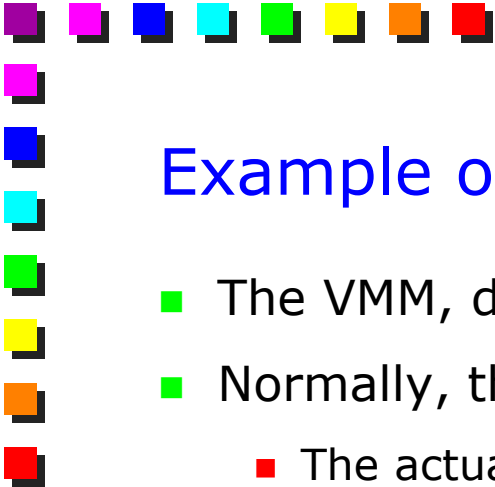
Para-virtualized device: status

- PV drivers are provided from modern hypervisors for several kind of devices, such as:
 - Network cards
 - Disks
 - Graphical video cards
- In addition, special devices could be conceived specially to optimize resource consumption in virtualized OS
 - Memory Ballooning driver

I/O virtualization: para-virtualization

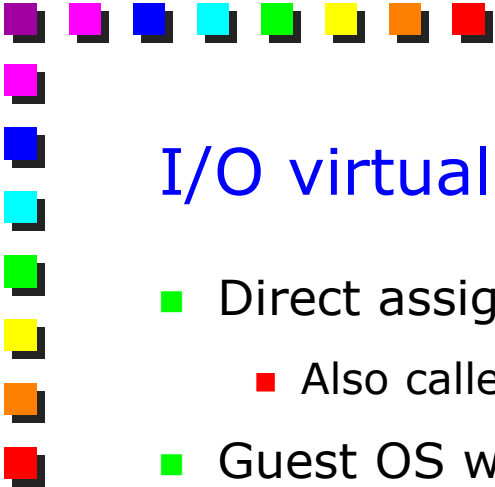
- Let the guest know that it is running in a virtual machine
- A para-virtualized driver consists of two components
 - **Frontend**: installed in the guest OS
 - **Backend**: installed in the hypervisor
- The frontend and the backend communicates through a protocol designed to be efficient in a virtualized environment





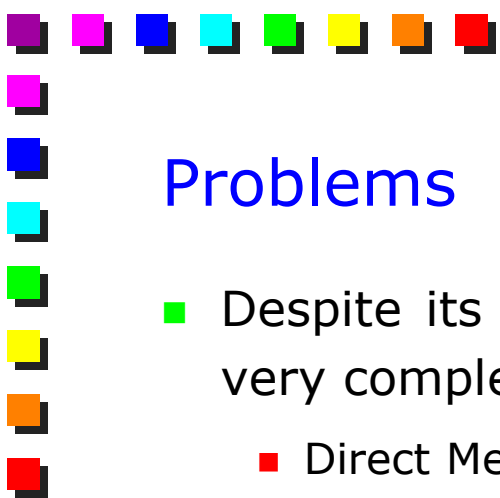
Example of PV-only device: Memory ballooning

- The VMM, during the initialization of a VM, defines its memory size
- Normally, the VMM needs to allocate statically all the memory assigned to the guest
 - The actual utilization in the guest OS is unknown to the hypervisor
- To reserve only the memory that is used by the guest, the hypervisor exploits a “Memory Ballooning” PV driver installed in the Guest OS
 - This driver provides to the hypervisor the information about current memory occupation of the Guest
 - This allows the hypervisor to over-commit memory allocation to several guests



I/O virtualization: direct assignment

- Direct assignment allows the VM to directly communicate with the physical device
 - Also called “Device pass-through”
- Guest OS will handle the device with the traditional (dedicated) driver
 - No need for an additional driver in the host OS, as the device will be totally handled by the guest OS
- This technique requires the exclusive assignment of a device to a VM
 - No device multiplexing over several VMs
- “PCI passthrough” is an example of this technique

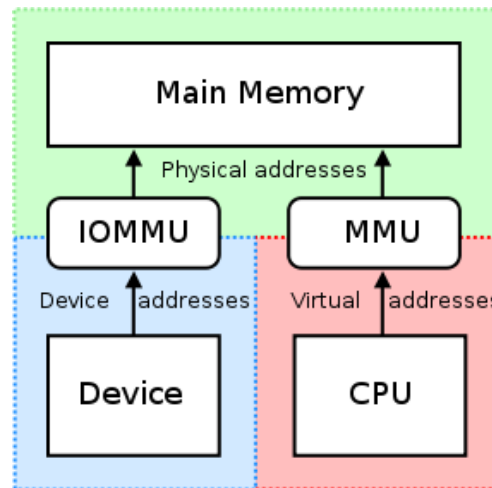


Problems

- Despite its apparently simple approach, the “direct access” to hardware devices is very complex indeed
 - Direct Memory Access (DMA) cycles have to be performed on **guest** physical addresses
 - Devices do not know the mapping between guest and host physical addresses for a particular VM
 - This could potentially lead to a memory corruption
 - Memory corruption can be avoided if the VMM or host OS intercepts the I/O operation and performs the correct translation
 - But this is slow and can introduce a significant overhead in I/O operations

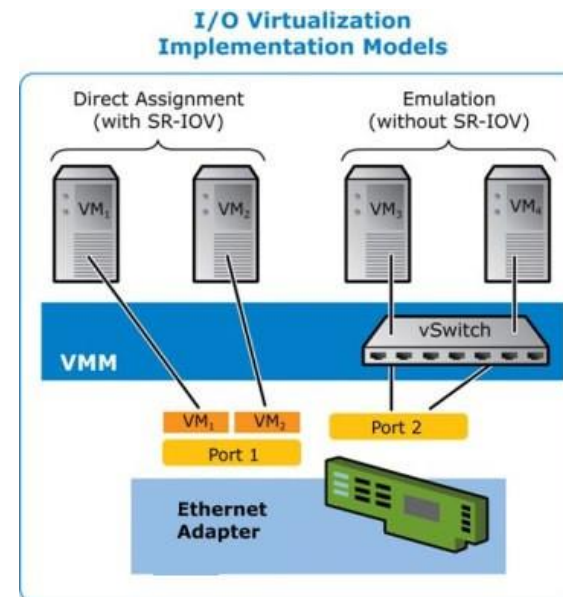
IOMMU

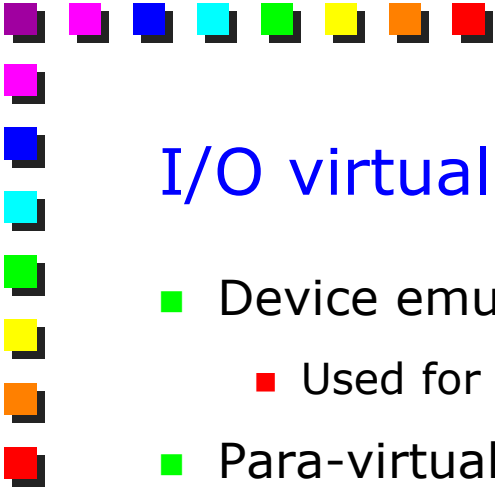
- Dedicated extension introduced by hardware manufacturers to boost and make direct assignment easier to be implemented in hypervisor
- Like the MMU, the IOMMU:
 - Remaps the addresses accessed by the hardware according to the same table used to map guest-physical address to host-physical addresses
 - DMA cycles will safely access the correct memory locations



SR-IOV

- The PCI-e standard defines SR-IOV as a mechanism to allow several directly assigned devices to be shared among VMs
 - Specially designed for network cards
- SR-IOV defines the possibility from the device to present several virtual devices, “virtual functions” to the OS
- The VMM will directly assigning a virtual function to a VM
- The hardware will handle itself the device multiplexing





I/O virtualization: summary

- Device emulation is slow and computationally expensive but offers a great flexibility
 - Used for devices that are not critical in terms of performance or legacy OS
- Para-virtualization is flexible and faster even if still computationally expensive
- Direct assignment is the fastest and less expensive technique but requires exclusive allocation of the device
 - In case of a network card, SR-IOV mitigates the problem of exclusive allocation of the device



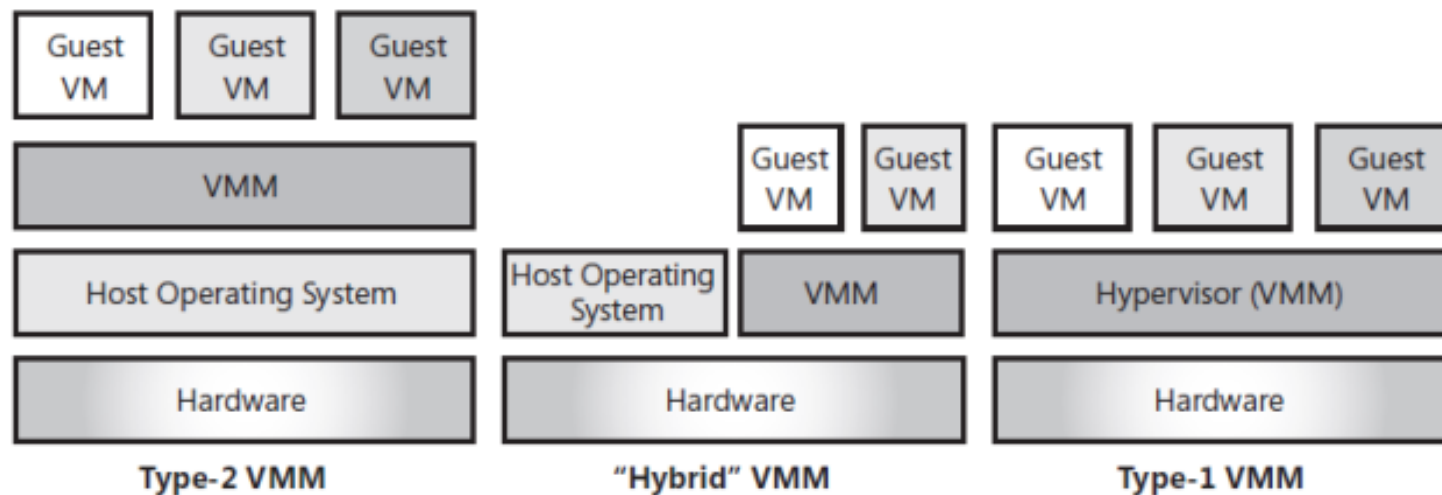
Part V

Hypervisor architectures



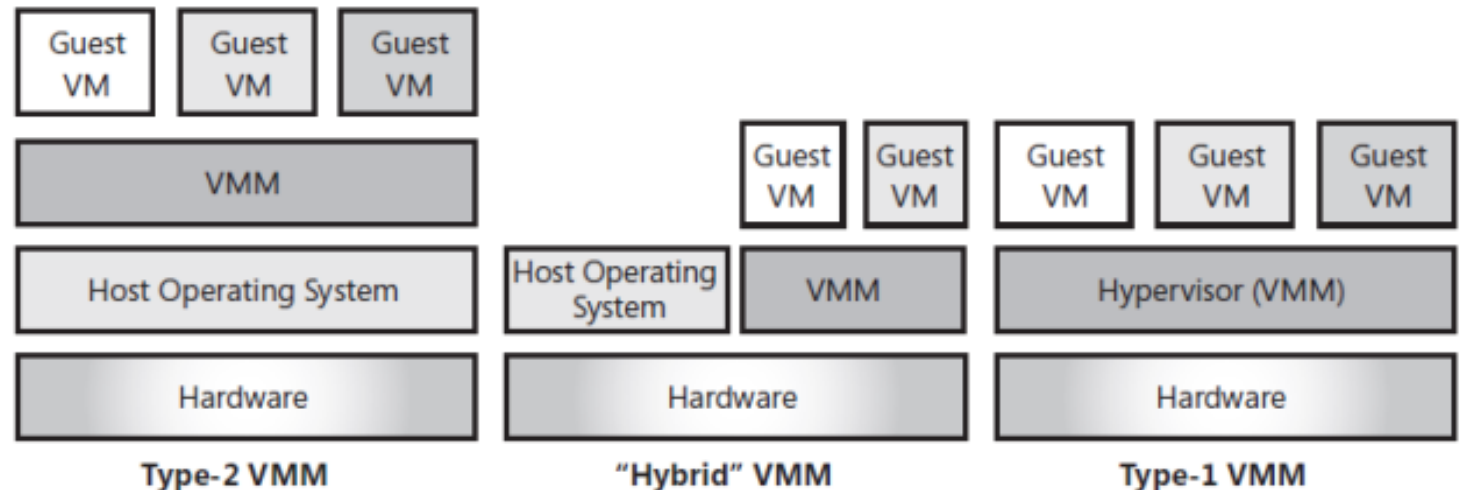
Introduction

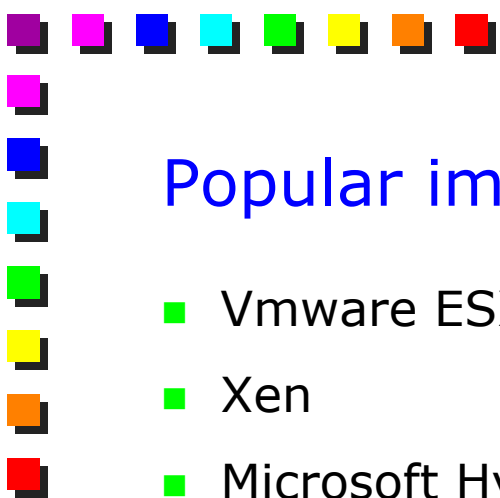
- Traditionally, hypervisors follow mainly two architectures, Type-1 and Type-2
- They follow different design criteria:
 - Performance
 - Ease of deployment & utilization
- Hybrid approaches are possible as well



Hypervisor architectures – Type 1

- The hypervisor runs directly on bare metal
 - No extra layers between Hypervisor and hardware
 - Normally able to provide the best performance
 - Installation and deployment may be cumbersome
 - The hypervisor is basically an Operating System where all the unnecessary components (e.g., printing, etc.) are stripped down
 - The problem is having the proper drivers for all your hardware
- Popular in server context



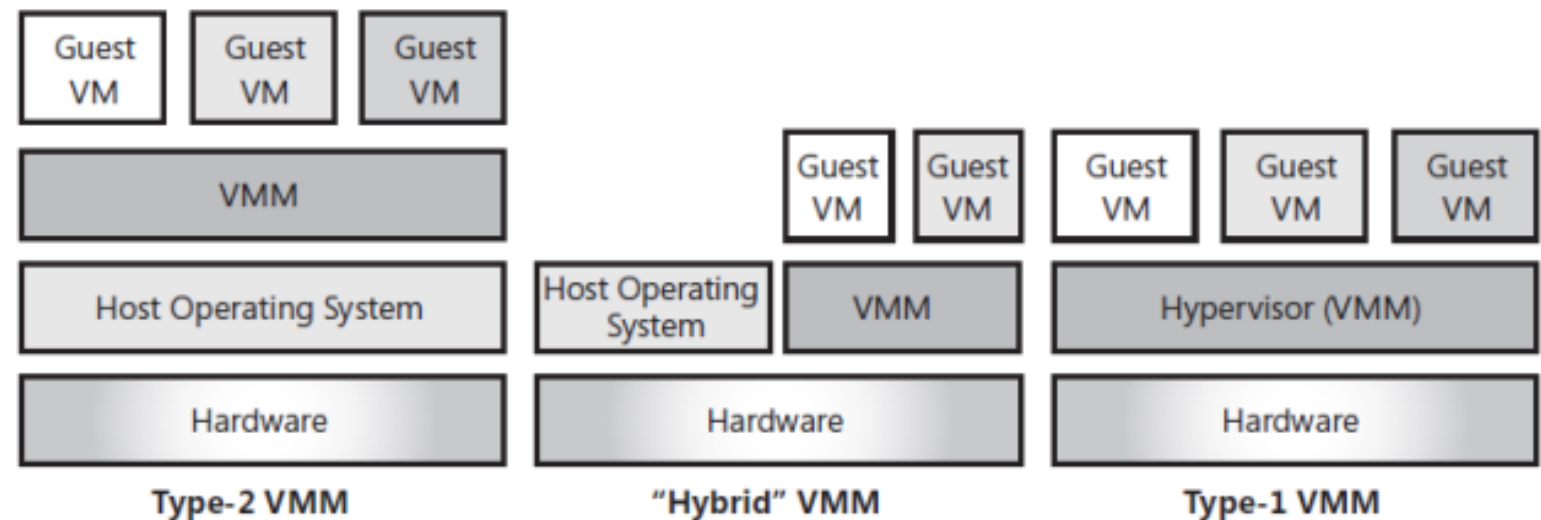


Popular implementations

- Vmware ESXi
- Xen
- Microsoft Hyper-V
 - When enabling Hyper-V on your Windows 10 desktop, your operating system will no longer executed on the bare hardware, but as VM
 - Hence, running a further hypervisor (e.g., VirtualBox) in Windows would not be a great choice, as we are asking the system to use nested virtualization
 - Instead, when Hyper-V is enabled, any further virtualized OS executed by Hyper-V would work in parallel with your Windows 10

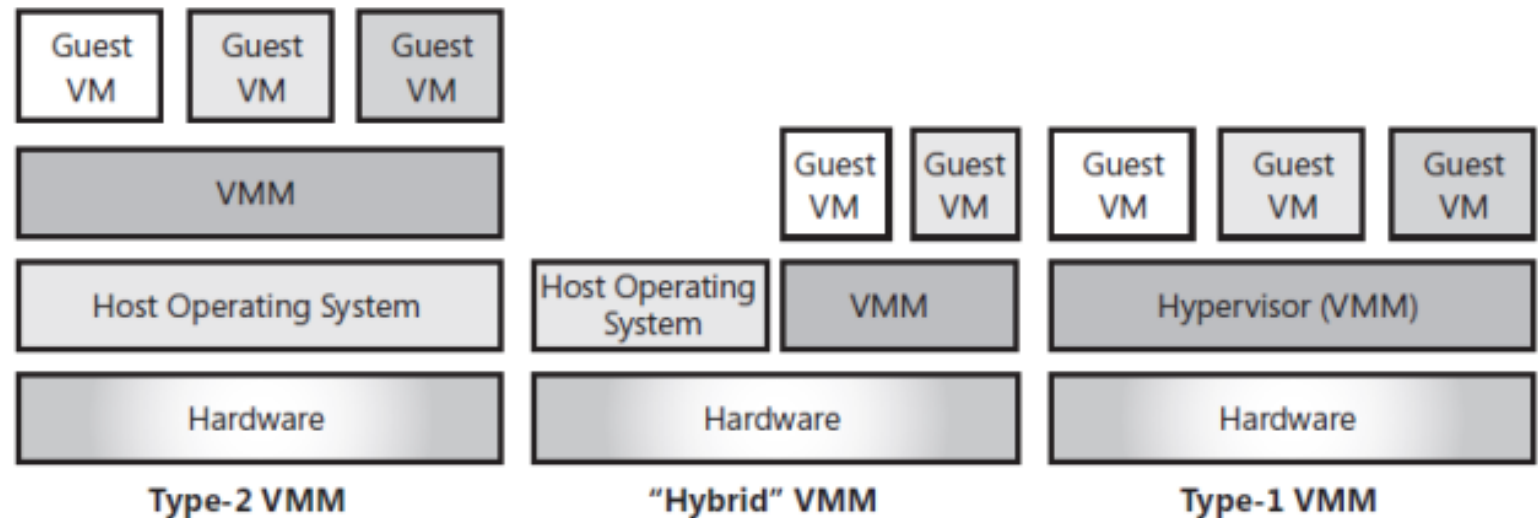
Hypervisor architectures – Type 2

- The hypervisor runs on a host OS, as a normal application
 - Installation and VM creation are simple tasks
 - Normally less performing than type 1
- Available implementations:
 - Oracle VirtualBox
 - VMware workstation



Hypervisor architectures – Hybrid approach

- The hypervisor is now implemented in the OS kernel
 - The host OS is itself the hypervisor, thanks to a specific component
- Normally simple to install and deploy
 - Drivers and support coming from the mainstream OS
- Very good performance
- Implementations
 - KVM (Linux)

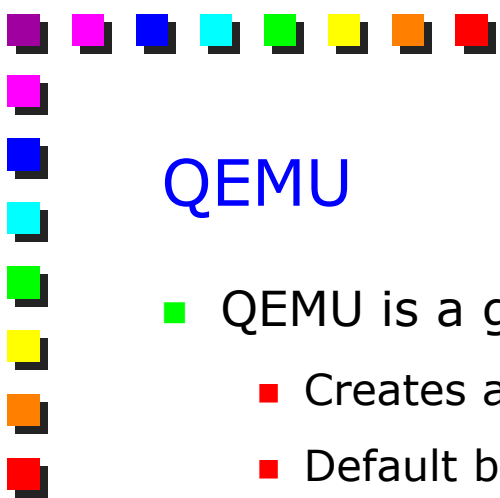




Part VI

Full virtualization in Linux: QEMU, KVM, VirtIO



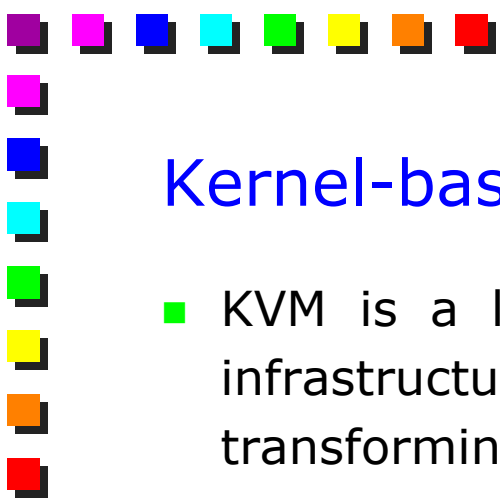


QEMU

- QEMU is a generic and open-source machine emulator and virtualizer
 - Creates and starts VMs, similar to other hypervisors (Vmware, etc)
 - Default behavior is based on **emulation** (not full virtualization)
 - It support Guests compiled for a different hardware architecture (e.g., ARM guest running on an Intel host)
- Two operating modes
 - User-mode emulation: it runs a single Linux program that is compiled for a different ISA
 - System-mode emulation (default): it emulates a full computing system, including peripherals, using binary translation
- Can be integrated with KVM (KVM-qemu) for hardware-assisted virtualization

Fabrice Bellard. QEMU, a fast and portable dynamic translator. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, USA, 2005.
Available at https://static.usenix.org/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.

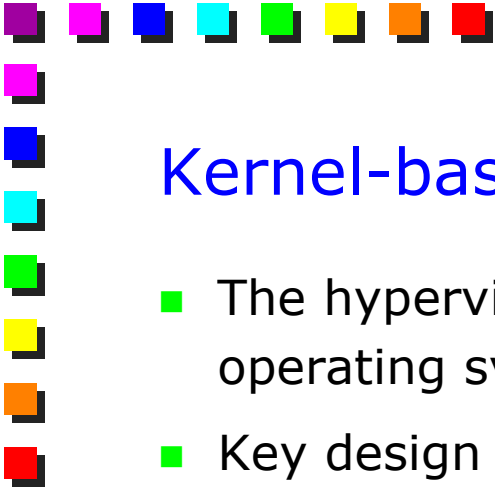




Kernel-based Virtual Machine (KVM)

- KVM is a loadable kernel module (kvm.ko) that provides the core virtualization infrastructure and a processor specific module (kvm-intel.ko / kvm-amd.ko), hence transforming the Linux kernel in a hypervisor
 - Loadable: Linux can be turned in a hypervisor dynamically
 - Full CPU virtualization: the guest is not aware of being virtualized, hence legacy/unmodified OS are supported
 - GuestOS must have been compiled for the same ISA of the host
- KVM leverages the hardware assisted virtualization (VMX root / non root operations)
 - The guest code is directly executed in the host CPUs
 - Requires Intel VT-X or AMD-V enabled CPUs
 - With KVM, Linux kernel runs at Ring 0, VMX root mode
 - Virtualbox is no longer able to run, while other systems (e.g., VMware Workstation) are still ok



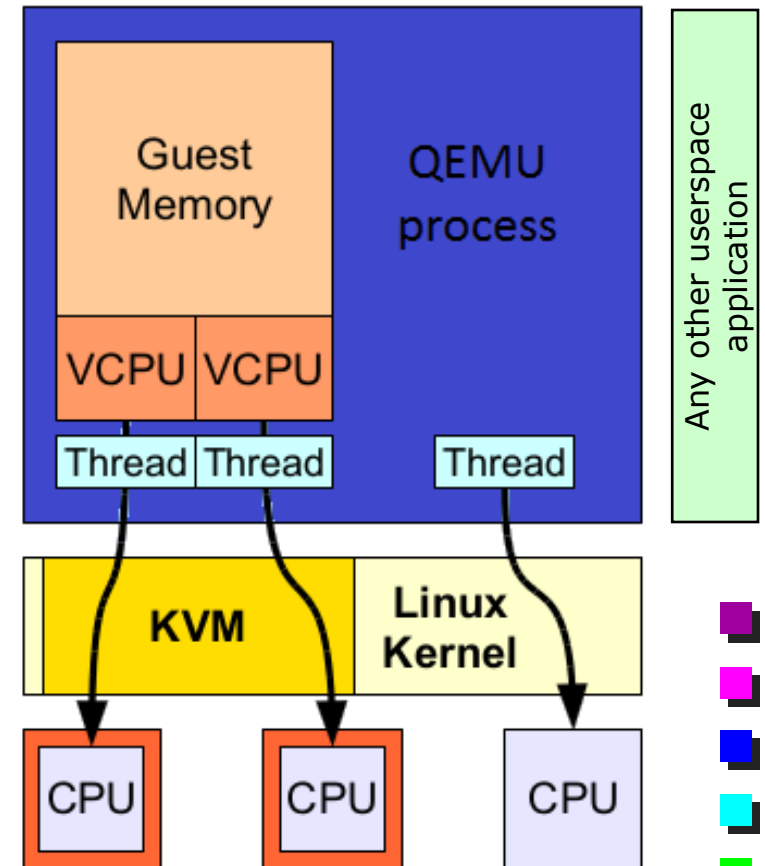


Kernel-based Virtual Machine (2)

- The hypervisor associates one or more CPUs to the VM in order to execute the guest operating system
- Key design choice: *"don't reinvent the wheel"*
 - A VMM is a specialized operating system that runs VMs rather than applications
 - Many components required by a VMM in addition to CPU and memory virtualization
 - Memory manager, process scheduler, I/O stack, etc.
 - The Linux kernel already implements all these features, so let's reuse them

Kernel-based Virtual Machine (KVM)

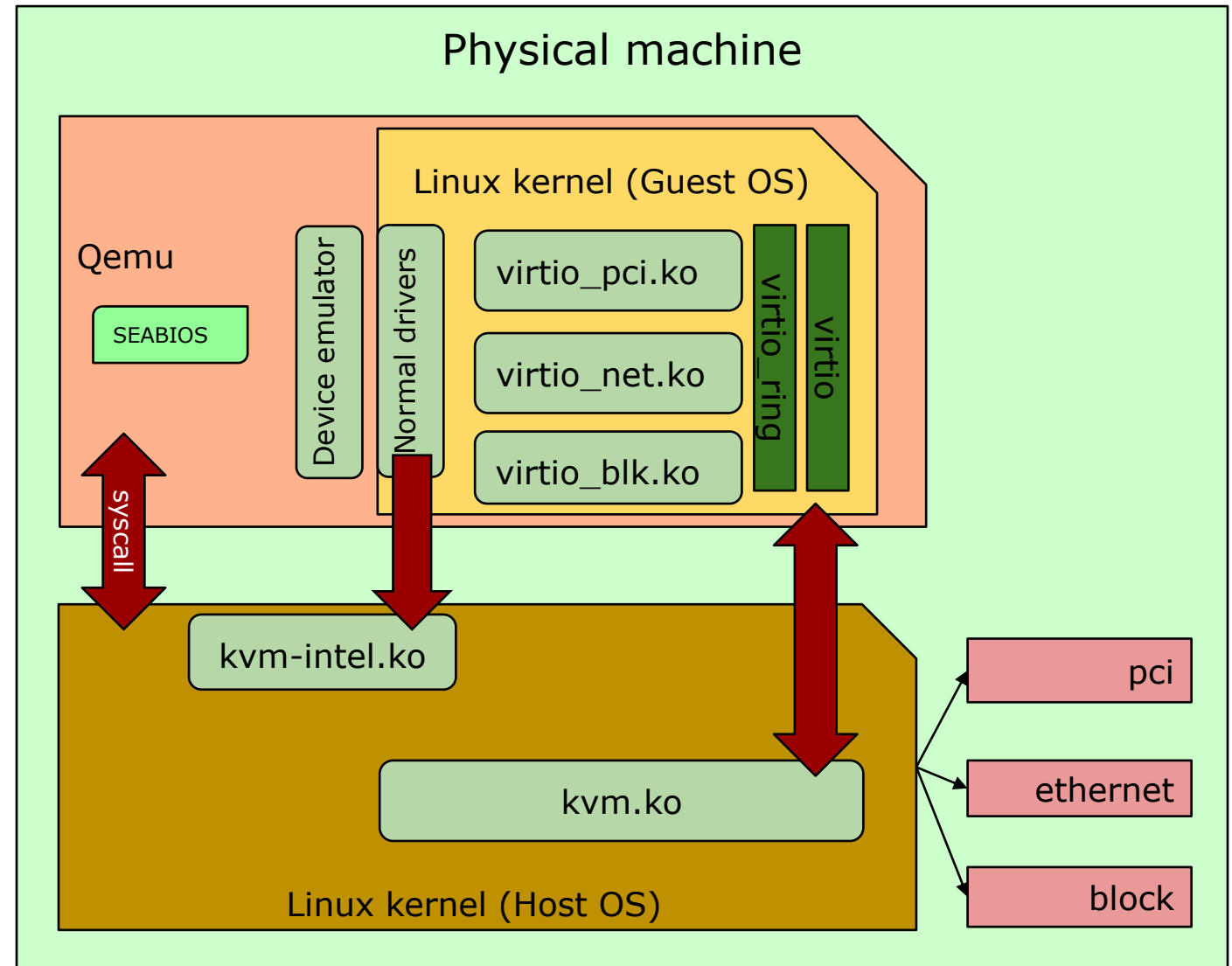
- Virtual machine created as a **single** userland process
 - KVM exposes the `/dev/kvm` interface, which can be used by userspace processes to spawn VMs
 - A VM is actually implemented as a regular Linux process
- VMs can contain
 - Memory
 - Virtual CPUs (vCPUs)
 - In-kernel device models
- Guest physical memory part of creating process' address space
- Each vCPU appears as a regular Linux thread in the host
 - KVM can benefit from all the features of the Linux kernel
 - e.g., the standard Linux scheduler

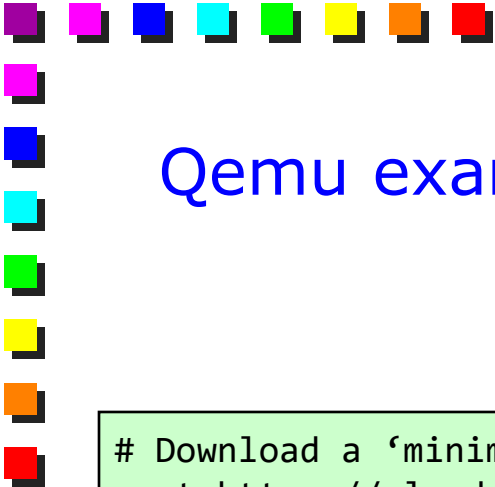


General organization

■ Main components

- kvm.ko
- kvm-intel.ko
- kvm-amd.ko
- qemu-kvm
 - Modified QEMU that uses KVM as hardware
- SEABIOS





Qemu example: (x64) emulated vs fully virtualized VM

```
# Download a 'minimal' cloud image from Ubuntu
wget https://cloud-images.ubuntu.com/minimal/releases/focal/release/ubuntu-20.04-minimal-cloudimg-amd64.img

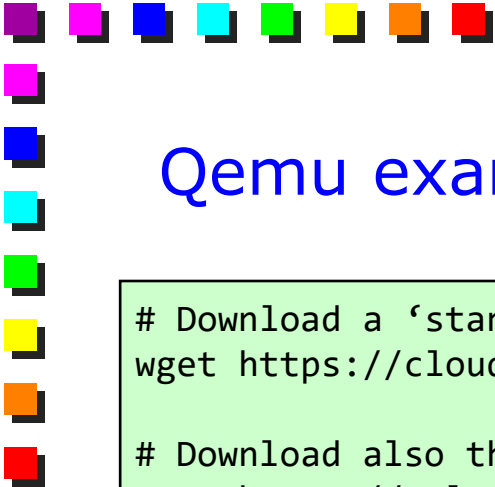
# Now, let's try to start it, using emulation (use the '-nographic' flag to start in a console window)
qemu-system-x86_64 -nographic ubuntu-20.04-minimal-cloudimg-amd64.img

# Please kill the process; ubuntu cloud images have the 'console login' disabled for security reasons

# Now, let's try to start the VM again, using kvm ('-enable-kvm' flag)
qemu-system-x86_64 -nographic -enable-kvm ubuntu-20.04-minimal-cloudimg-amd64.img

# You can easily compare the different booting speed in the two cases!
```





Qemu example: ARM emulated VM running on x86

```
# Download a 'standard' cloud image from Ubuntu (compiled for ARM binary architecture)
wget https://cloud-images.ubuntu.com/releases/hirsute/release/ubuntu-21.04-server-cloudimg-arm64.img

# Download also the BIOS
wget https://releases.linaro.org/components/kernel/uefi-linaro/16.02/release/qemu64/QEMU_EFI.fd

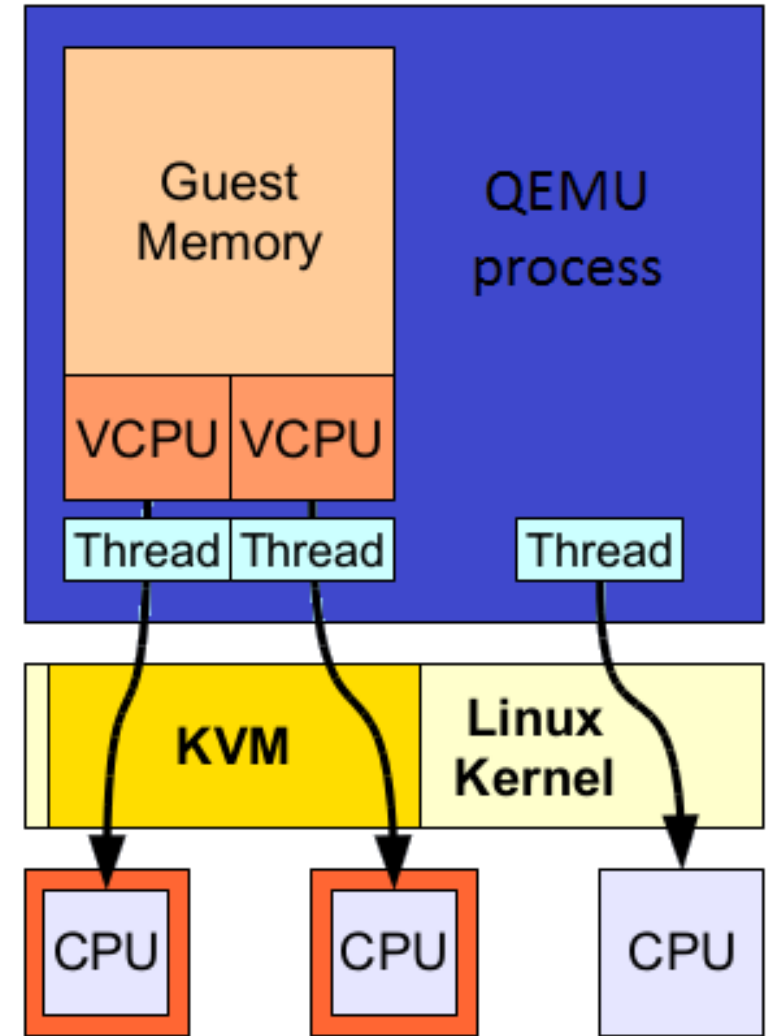
# Now, let's run the ARM64 image, where:
#   -m 1024 - 1024MB of system memory.
#   -M virt - emulate a generic QEMU ARM machine.
#   -bios QEMU_EFI.fd - the BIOS firmware file to use.
#   -nographic - output goes to the terminal (rather than opening a graphics-capable window).
#   -device virtio-blk-device,drive=image - create a Virtio block device called "image".
#   -drive if=none,id=image,file=ubuntu.img - create a drive using the "image" device and our ubuntu
#           cloud server image.

qemu-system-aarch64 -smp 2 -m 1024 -M virt -bios QEMU_EFI.fd -nographic -device virtio-blk-
device,drive=image -drive if=none,id=image,file=ubuntu-21.04-server-cloudimg-arm64.img

# More details and other options at https://gist.github.com/brianredbeard/36f5bc52f8e8602b8ac0fcdb8bacc4e3
```

QEMU

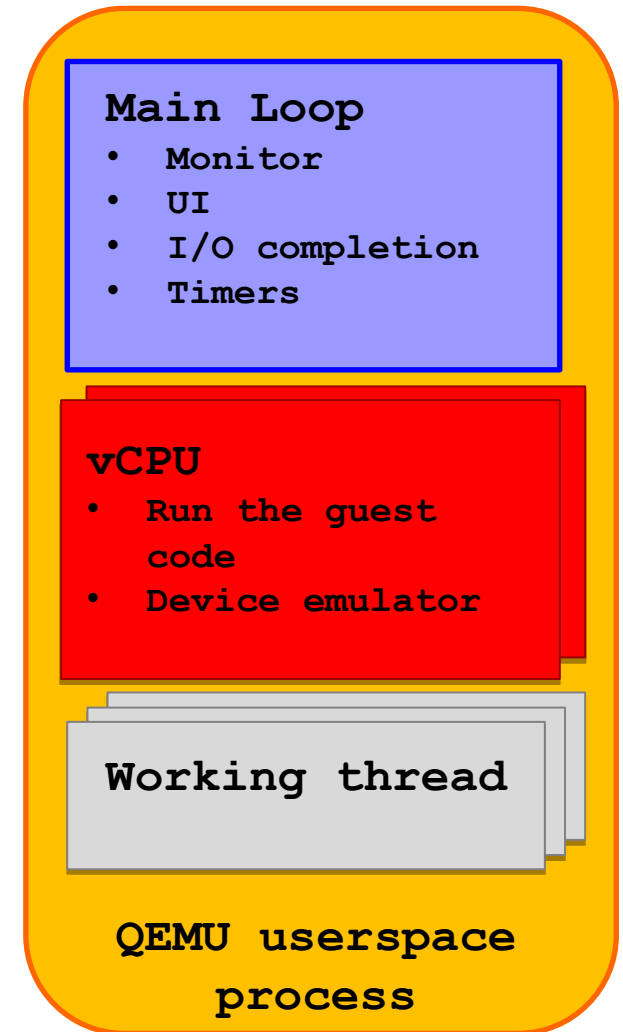
- Host user space process with the following responsibilities:
 - Set up a VM and its I/O devices
 - Execute guest code via KVM kernel module
 - Although other VMMs (e.g., Xen) are also supported as well
 - I/O emulation (and live migration)
 - I/O operations are in fact privileged, and must be emulated in the host
- Each QEMU process corresponds to a different VM
 - The process owns the guest RAM
 - Each guest vCPU is a different thread in the host
 - When the VM shuts down, the QEMU process terminates



QEMU

■ QEMU threading model

- Main loop thread:
 - Event handler
 - E.g., monitor commands sent by users, timers expiration
 - 1 thread per vCPU:
 - Run the guest code through KVM
 - Emulate devices
 - E.g., network cards, disk controller, USB bus, PCI bus
 - (several) working thread(s):
 - Execute expensive I/O operations in order to limit the time the vCPU is blocked
- ## ■ Only these threads are seen in the host
- The host does not have any knowledge of the threads executed in the guest



```

kvm = open("/dev/kvm");           // Open the '/dev/kvm' device to interact with KVM kernel module
vmfd = ioctl(kvm, KVM_CREATE_VM); // Create a VM, which includes everything associated with one emulated system
                                   // (e.g., memory, CPUs). KVM returns a handle to the VM
mem = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED); // Map files or devices into memory. In this case, it allocates a single
                                                                // page of memory to hold our code, starting at virtual address 0x1000
memcpy(mem, code, sizeof(code)); // Copy our machine code into it
ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &mem); // Tell KVM virtual machine to use this memory region
vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, 0); // Create a virtual CPU (which includes processor registers and other execution state) to
                                           // run that code. Currently one (index '0') CPU is created and a handle to it is returned
mmap_size = ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE); // Each vCPU has an associated 'struct kvm_run', used to communicate information about the
                                                // CPU between kernel and user space. When a "vmexit" occurs, this structure contains
                                                // information about why it was stopped. This call gets the size of this structure.
run = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_SHARED); // Map the 'kvm_run' structure into user space
sregs.rip = 0x1000; // Initialize instruction pointer in the CPU (with address defined in initial mmap() )
ioctl(vcpufd, KVM_SET_SREGS, &sregs); // Set up the initial states of CPU registers (code segment, instruction pointers, general
                                       // purpose registers such as RAX, etc.)

while (1)
{
    ioctl(vcpufd, KVM_RUN, NULL); // Start running instructions with the VCPU. It will return successfully each time
                                  // virtualization stops, such as for us to emulate hardware.
                                  // This is the reason we run it in a loop. More about transitions in the next slide.
                                  // Now, we have to emulate each single reason of exit

    switch (run->exit_reason) {
        case KVM_EXIT_HLT:
            // Emulate HLT instruction
            break;

        case KVM_EXIT_IO:
            // Triggering instruction in the VM code: out %eax, $0x3f8
            if (run->io.direction == KVM_EXIT_IO_OUT && // Check if OUT instruction wrote a one word (4 bytes) on port 0x3f8. If so,
                run->io.size == 4 && // it corresponds to a well-defined emulated hardware and QEMU knows what to do
                run->io.port == 0x3f8 &&
                run->io.count == 1)
            // Emulate the given 'OUT' instruction
            break;
    }
}

```

Full example available at <https://lwn.net/Articles/658511/>

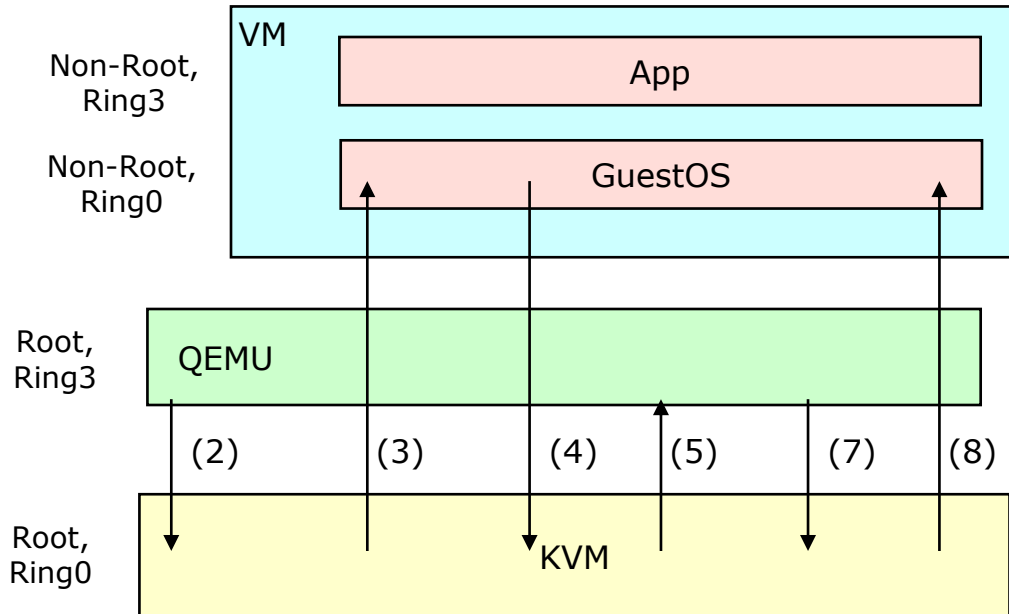


QEMU-KVM-VM: interaction example

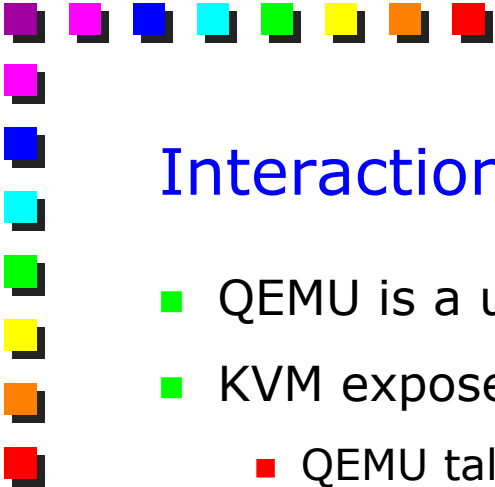
Main loop:

```
while (1)
{
    ioctl(vcpuFd, KVM_RUN, NULL);

    switch (run->exit_reason)
    {
        case KVM_REASON_1:
    }
}
```



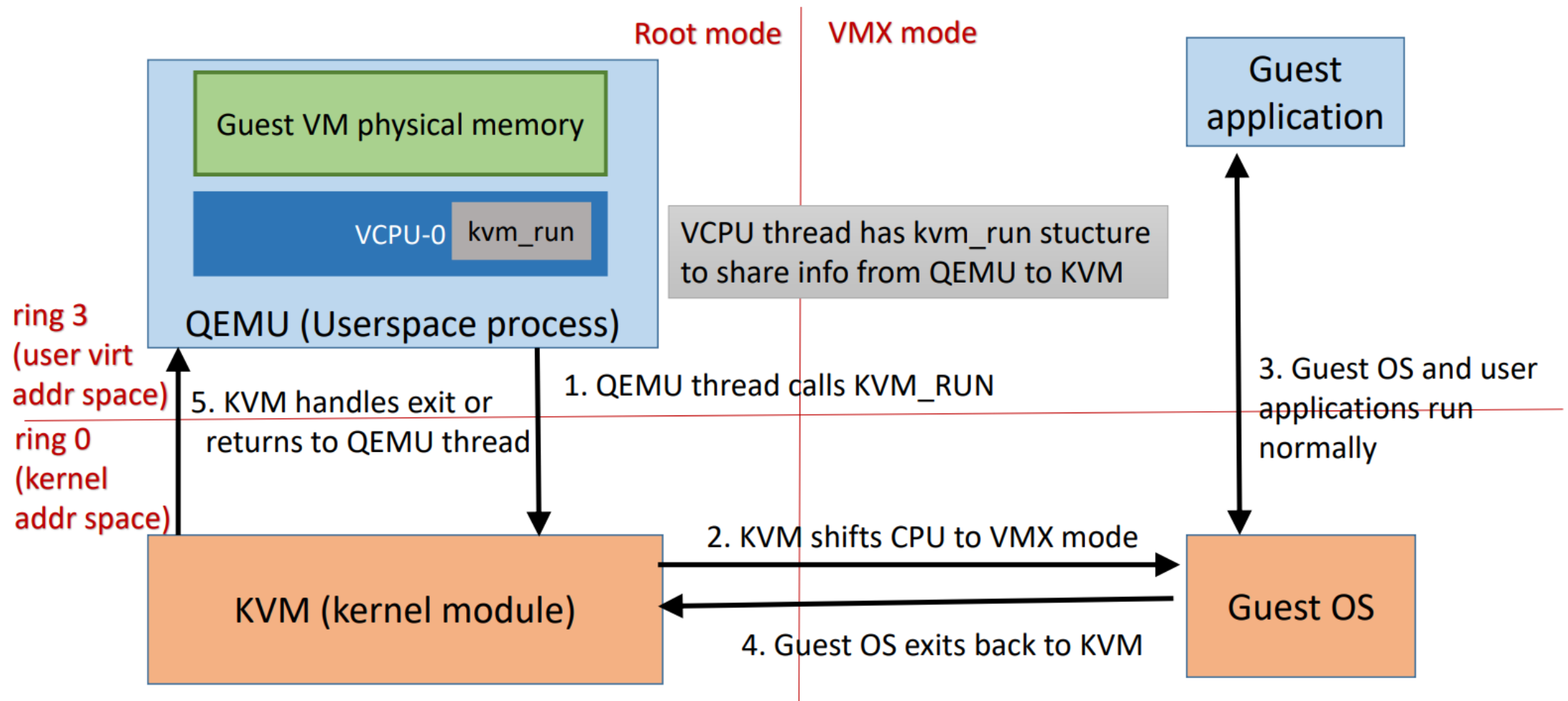
- (1) QEMU running (Ring 3-Root)
- (2) QEMU asks for ioctl to start the VM: transition to Ring 0-Root
- (3) KVM starts VM: transition to Ring 0-NoRoot (GuestOS)
- (4) VM generates a trap: transition to Ring 0-Root mode (KVM)
- (5) KVM returns control to QEMU (Ring 3-Root)
- (6) QEMU serves the 'vmexit' event and it emulates the device
- (7) QEMU asks for ioctl to resume the VM (same of step (2))
- (8) Same of step (3): KVM resumes VM, hence transition to Ring 0-NoRoot (GuestOS)



Interaction between QEMU and KVM (1)

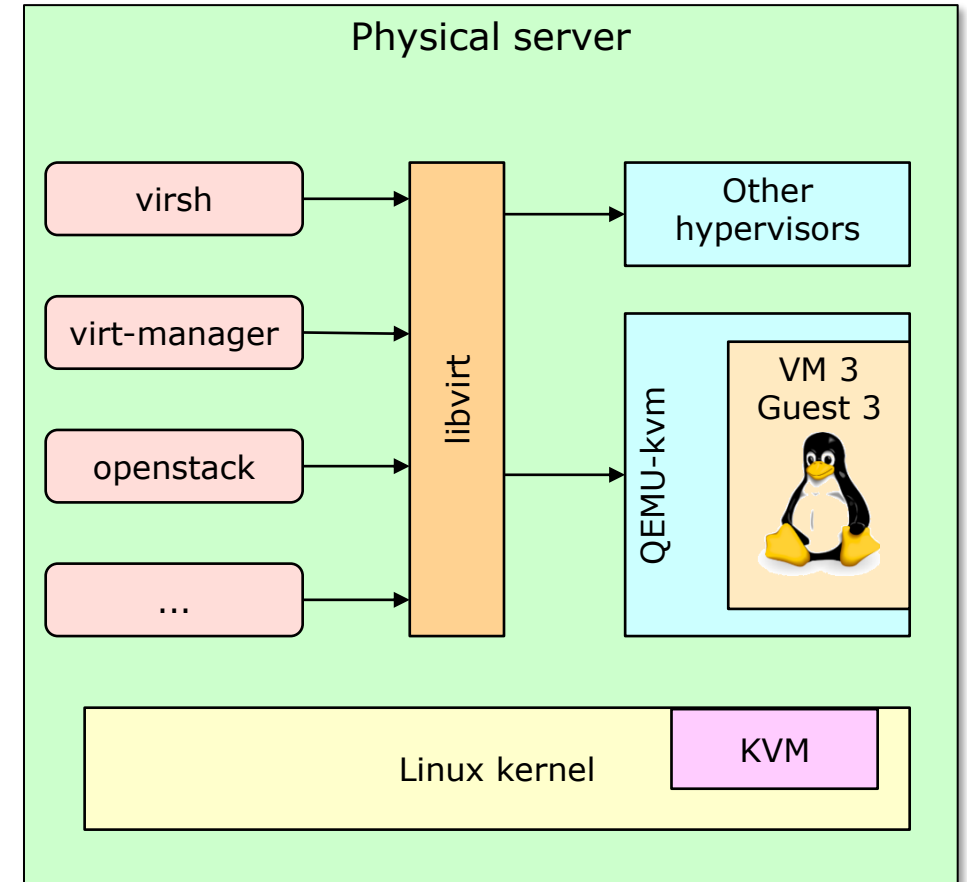
- QEMU is a userspace process
- KVM exposes a dummy device
 - QEMU talks to KVM via open/ioctl syscalls
- Allocates memory via mmap for guest VM physical memory
- Creates one thread for each virtual CPU (VCPU) in guest
- Multiple file descriptors to /dev/kvm (one for QEMU, one for VM, one for VCPU and so on)
 - ioctl on file descriptors to talk to KVM
- Host OS sees QEMU as a regular multithreaded process
 - QEMU runs at ring 3, VMX-root (simply because it is part of the HostOS)

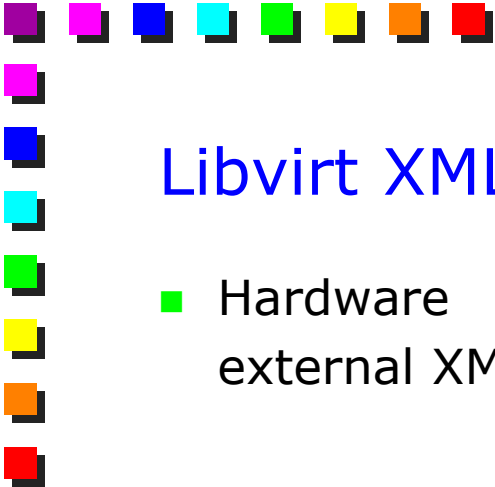
Interaction between QEMU and KVM (2)



Libvirt

- Virtualization api
- Hypervisor agnostic
 - Supports kvm, xen, vmware, virtualbox, etc.
- Provision, create, modify, monitor, control, migrate and stop domains
- Also included the “virsh” shell

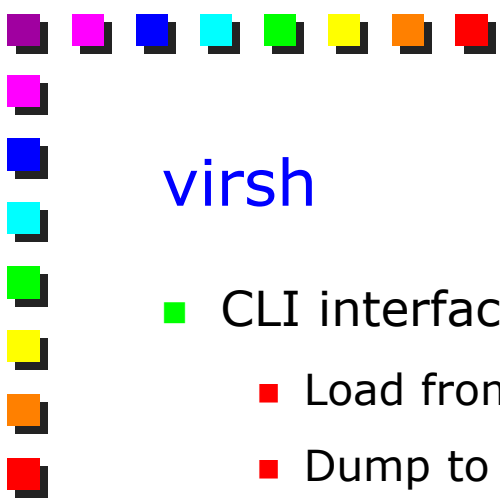




Libvirt XML

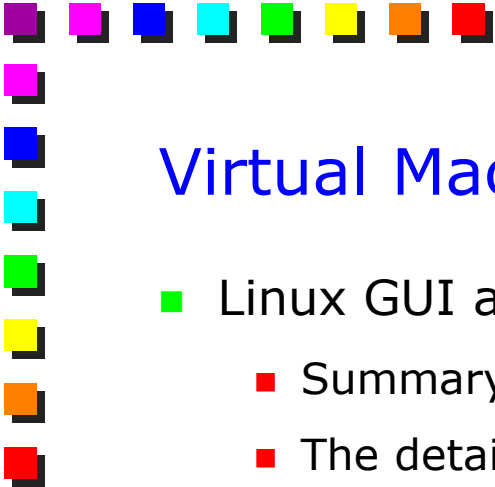
- Hardware profile stored in an external XML file

```
<domain type='kvm' id='2'>
  <name>domainname</name>
  <uuid>something-unique</uuid>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <vcpu placement='static'>1</vcpu>
  <os>
    <type arch='x86_64' machine='rhel6.3.0'>hvm</type>
    <boot dev='hd' />
  </os>
  <features>
    <acpi/><apic/><pae/>
  </features>
  <clock offset='utc' />
  <devices>
    <emulator>/usr/libexec/qemu-kvm</emulator>
```



virsh

- CLI interface for libvirt
 - Load from xml
 - Dump to xml
 - Edit xml
 - Start/stop/restart domains
 - Define storage pools
 - Define networks
 - Connect to machines
 - Connect to other hypervisors
 - Can be used with the hypervisors supported by libvirt, although it is usually used only for qemu/kvm
- 



Virtual Machine Manager (virt-manager)

- Linux GUI application for managing virtual machines, based on libvirt
 - Summary view of running domains, their live performance & resource utilization statistics
 - The detailed view about performance and utilization over time
 - Wizards enable the creation of new domains, and configuration/adjustment of a domain's resource allocation and virtual hardware
 - Includes an embedded VNC client viewer that a full graphical console to the guest domain
 - Supports for both **local** and **remote** hypervisors
 - Can be used with the hypervisors supported by libvirt, although it is usually used only for qemu/kvm
- 

Virt-manager

Virtual Machine Manager

File Edit View Help

Open [Play] [Pause] [Shutdown]

Name	CPU usage	Host CPU usage	Memory usage
▼ localhost (QEMU)			
f16 Running			
f17 Saved			
f18 Paused			
f18-q35 Shutoff			
f19 Shutoff			
f19-32 Shutoff			
f19-migrate Shutoff			

rhel7.3 on QEMU/KVM

File Virtual Machine View Send Key

[Monitor] [Lightbulb] [Play] [Pause] [Shutdown] [Add Hardware]

Overview Performance **CPU** Memory Boot Options VirtIO Disk 1 IDE CDROM 1 NIC :29:d2:a9 Tablet Mouse Keyboard Display Spice Sound: ich6 Serial 1 Channel qemu-ga Channel spice Video QXL Controller USB Controller PCI Controller IDE Controller VirtIO Serial USB Redirector 1 USB Redirector 2

CPU

Logical host CPUs: 4

Current allocation: 2 - +

Maximum allocation: 2 - +

Configuration

☐ Copy host CPU configuration

Model: Haswell-noTSX

► Topology

Add Hardware

Networking

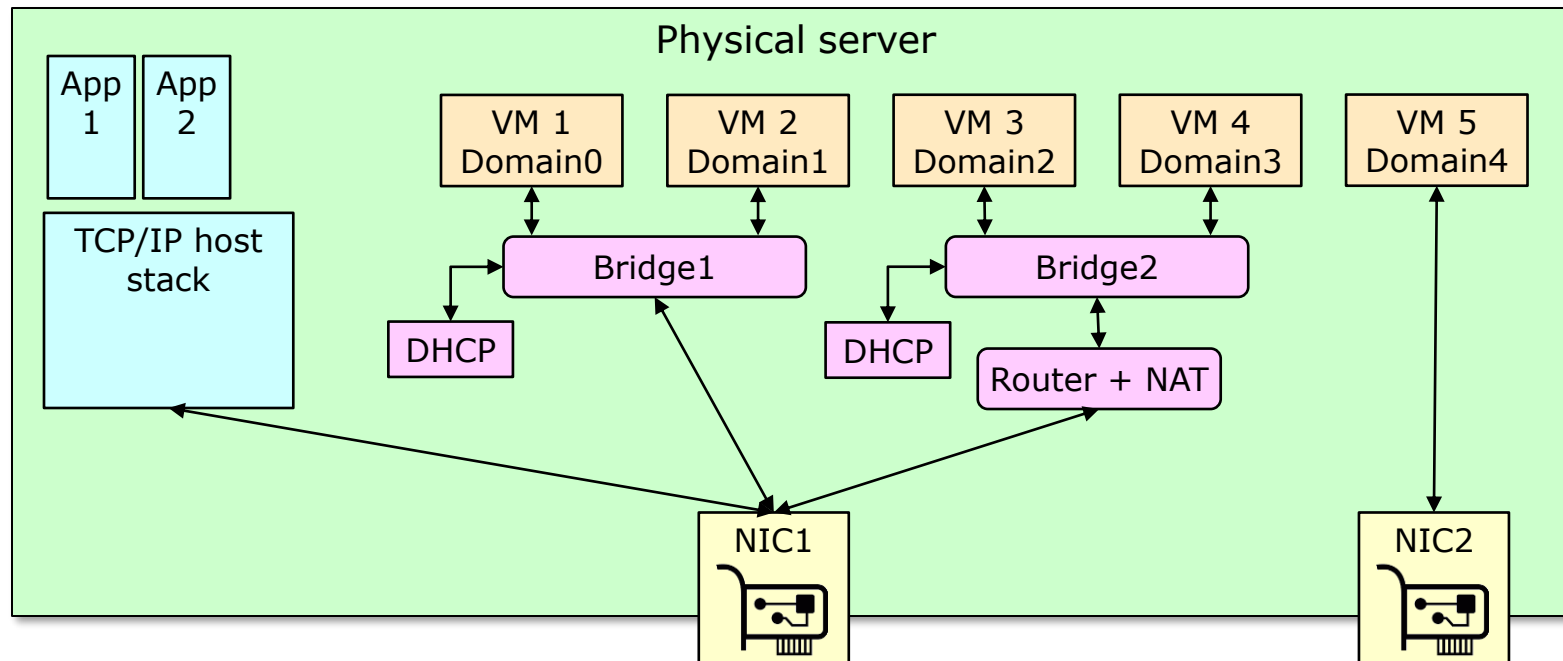
■ The short story

- bridge
- nat - dnsmasq
- physical device

■ More on a dedicated set of slides

■ Tools

- virt-manager
- bridge-utils ("brctl")
- dnsmasq - dns and dhcp

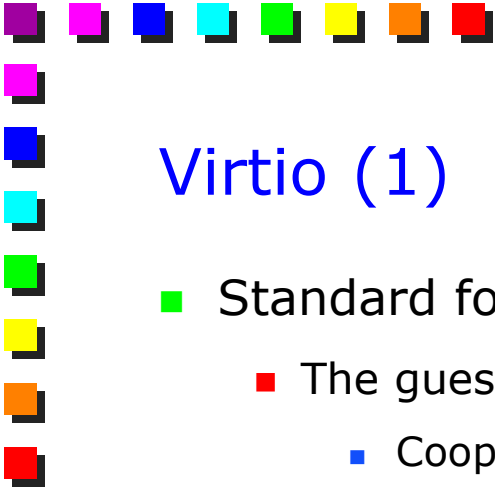




Vagrant

■ “Infrastructure as code”



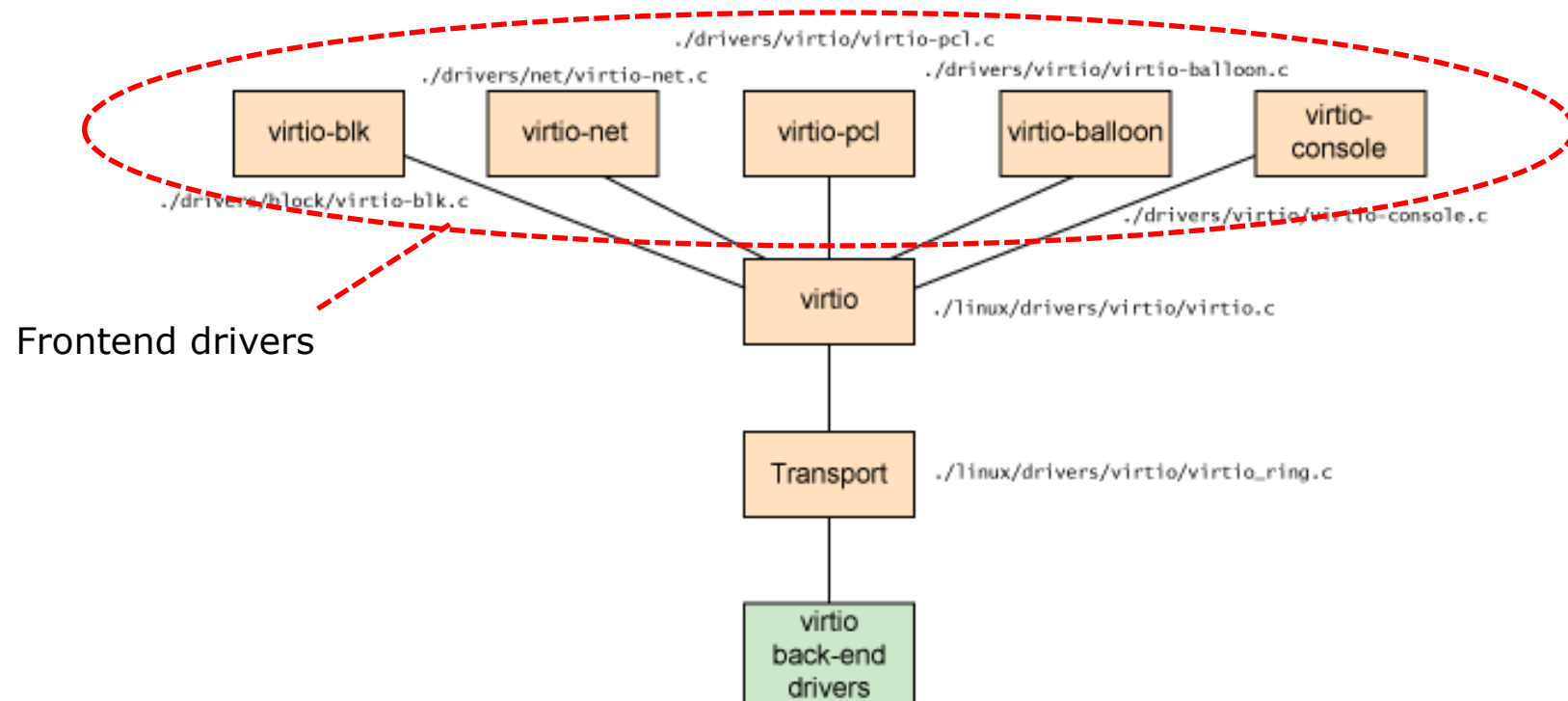


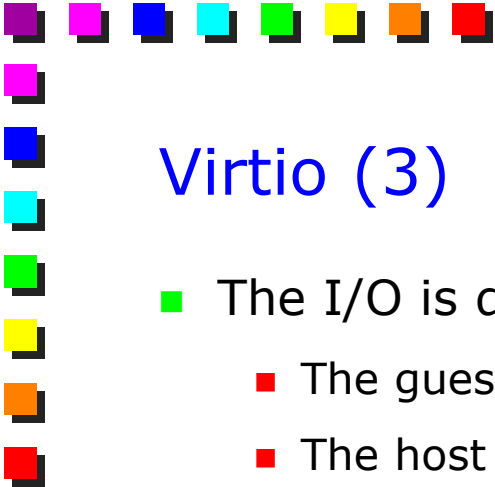
Virtio (1)

- Standard for paravirtualized device drivers
 - The guest device driver knows it is in a virtual environment
 - Cooperates with the hypervisor
 - Virtio provides better performance than fully virtualized devices
 - 2-3 times for network I/O
- Main goal: share as much code as possible among drivers
- Provides a common frontend for all the devices of the same type
 - E.g., all the network devices have the same frontend
 - The backend can be different, and it is hidden to the guest operating system
- Virtio devices appear as normal PCI devices to the guest VM
 - Vendor ID: 0x1AF4
 - Device ID between 0x1000 and 0x103F

Virtio (2)

- Virtio drivers implemented for
 - Block devices (e.g., disk), network devices, PCI emulation, serial devices (e.g., console)
 - Balloon memory driver





Virtio (3)

- The I/O is done through the *virtqueue* abstraction
 - The guest puts buffers into the queue
 - The host consumes buffers that are in the queue
 - Each buffer is a scatter-gather array
 - The structure of the data depends on the device type
- The number of queues per device depends on the device type
 - E.g; The virtio-net device has two mandatory virtqueues (TX and RX) and an optional virtqueue (control)
 - Control queue allows the guest to control network specific features on the host machine
 - E.g., promiscuous mode, hardware checksum calculations
- The driver registers a callback for the virtqueue when it is initialized



Scatter-gather I/O

- Some applications may need to read or write data to multiple buffers, which are separated in memory
 - Although this can be done with multiple calls to read and write, it is inefficient because there is overhead associated with each kernel call
- The GNU C Library provides special high-speed primitives to perform these scatter-gather operations in a single kernel call, and an emulation on any system that lacks these primitives
 - Defined in `sys/uio.h`

```
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <unistd.h>
#include <sys/uio.h>

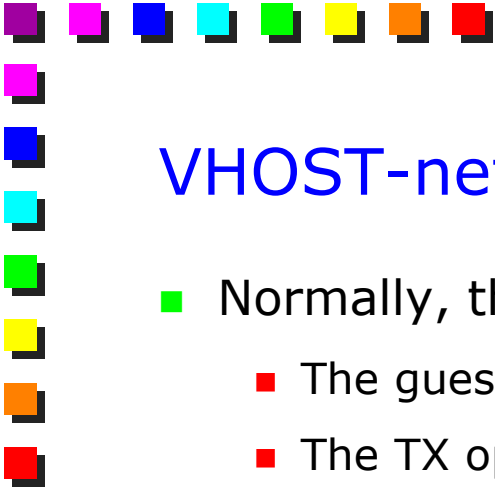
int main(int argc, char *argv[])
{
    const char buf1[] = "Hello, ";
    const char buf2[] = "Wikipedia ";
    const char buf3[] = "Community!\n";

    struct iovec bufs[] = {
        { .iov_base = (void *) buf1, .iov_len = sizeof(buf1)-1},
        { .iov_base = (void *) buf2, .iov_len = sizeof(buf2)-1},
        { .iov_base = (void *) buf3, .iov_len = sizeof(buf3)-1},
    };

    if (writev(STDOUT_FILENO, bufs, sizeof(bufs)/sizeof(*bufs))== -1)
    {
        perror("writev()");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

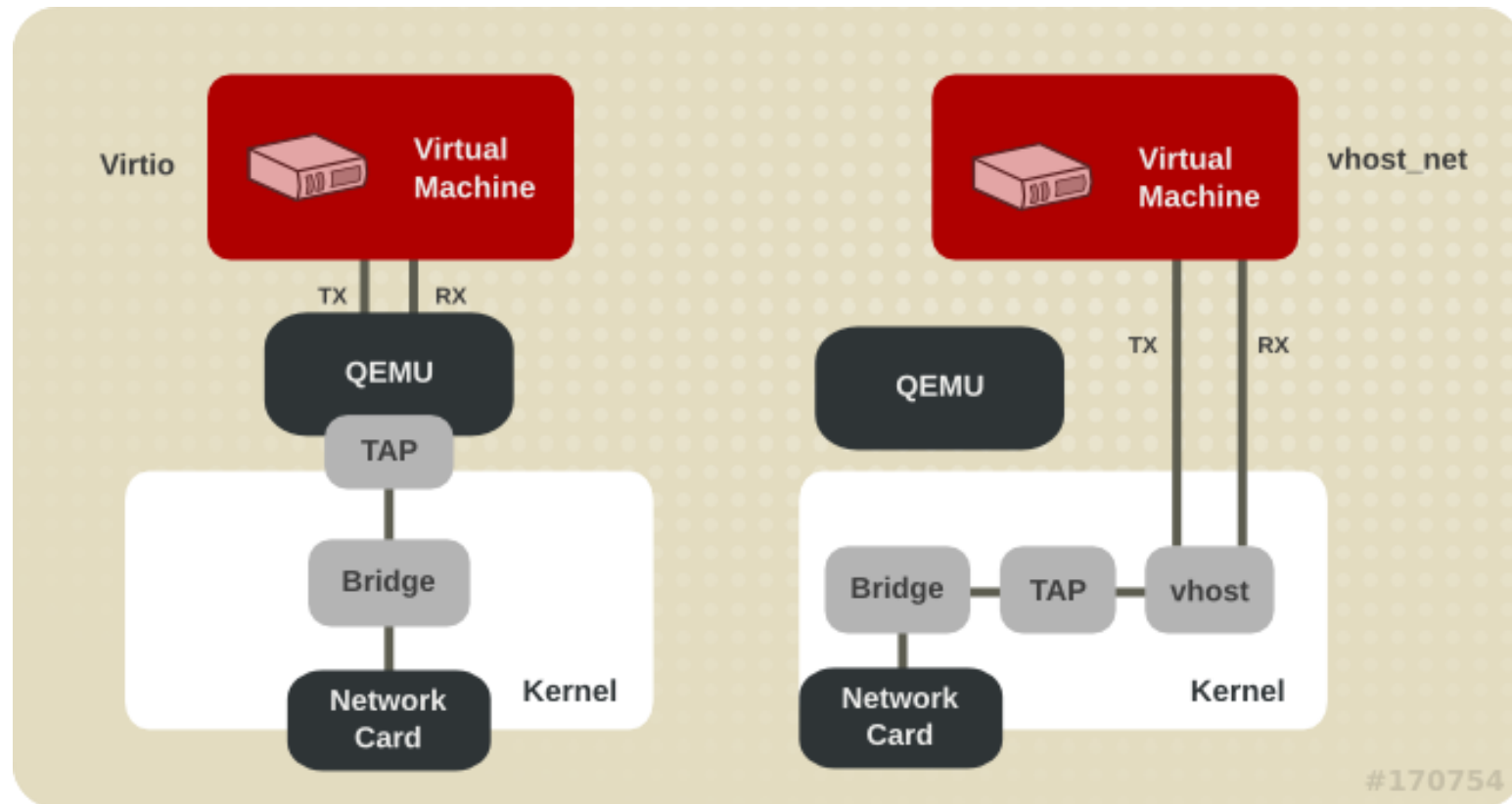
https://en.wikipedia.org/wiki/Vectored_I/O



VHOST-net (1)

- Normally, the QEMU userspace process emulates I/O access from the guest
 - The guest sends packets through the virtio-net driver
 - The TX operation is emulated by the virtio-net device in QEMU
 - This device exploits system calls to actually TX packets on the Tap interface
 - Packet TX: guest -> host userspace (QEMU) -> host kernel
- The VHOST-NET driver emulates the virtio-net network card in the host kernel
 - VHOST-NET puts virtio emulation code into the kernel
 - QEMU userspace is let out of the picture when TX/RX packets
 - Device emulation code directly calls into kernel subsystem
- No system calls from userspace
 - High throughput / low latency guest networking

VHOST-NET (2)





VHOST-NET - cont'd

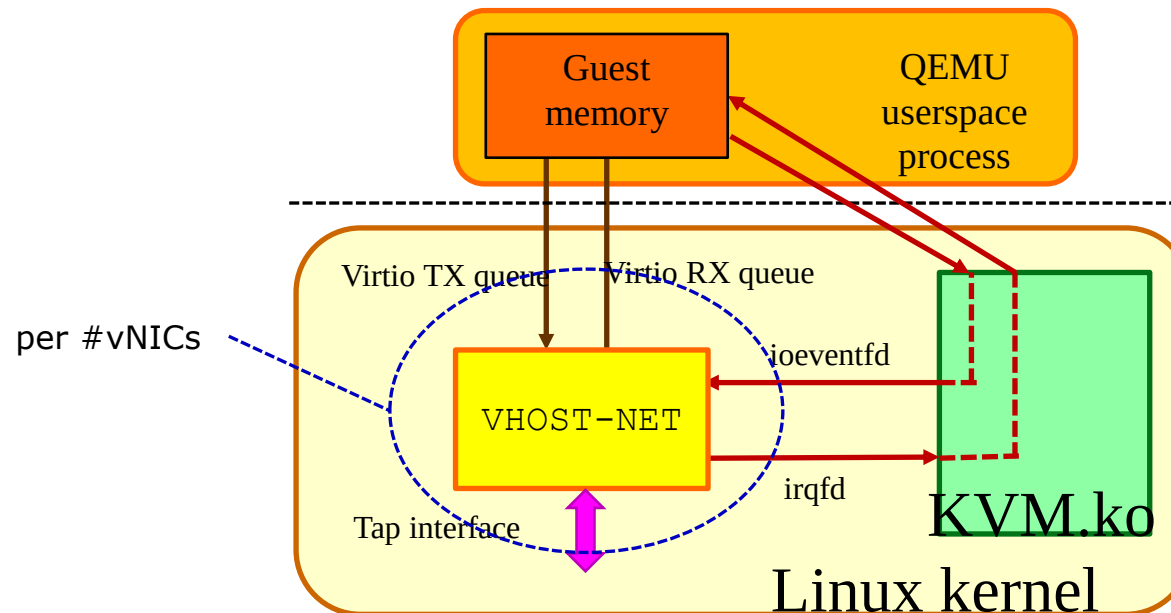
- The guest sees no difference between a VHOST-NET and a virtio-net device
- VHOST-NET is not a self-contained virtio device implementation:
 - Depends on userspace to handle control plane
 - QEMU still used for virtio feature negotiation
 - e.g., checksum offload
 - Data plane is done in-kernel
 - Performance are needed on data plane
- QEMU initializes VHOST-NET through `/dev/vhost-net` interface
 - A new kernel thread is created associated with the QEMU process
 - It is the "*vhost worker thread*"
 - Emulates the virtio-net device in the host kernel and handles IO events
 - Each vhost worker thread emulates a single vNIC



VHOST-NET - cont'd

■ The vhost worker thread

- Works with interrupt
 - No busy waiting (i.e., polling)
- Waits for *TX virtqueue* kicks
 - Takes packets from the queue and transmits them on a tap interface
- Waits for packets coming from the tap and places them on the *RX virtqueue*

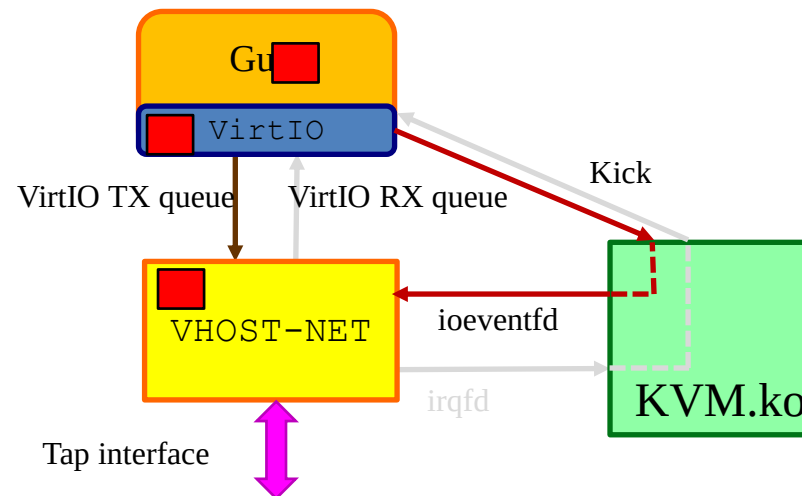




VHOST-NET - cont'd

- When the guest puts packets in the *TX virtqueue*, it has to kick
 - The kick must be brought to the VHOST-NET device
- VHOST instances are set up with an *eventfd* file descriptor that the vhost worker thread watches for activity
- `kvm.ko` has a feature called *ioeventfd* for taking an *eventfd* and hooking it up to a particular guest I/O exit
- QEMU userspace registers an *ioeventfd* for the VM exit triggered by the kick

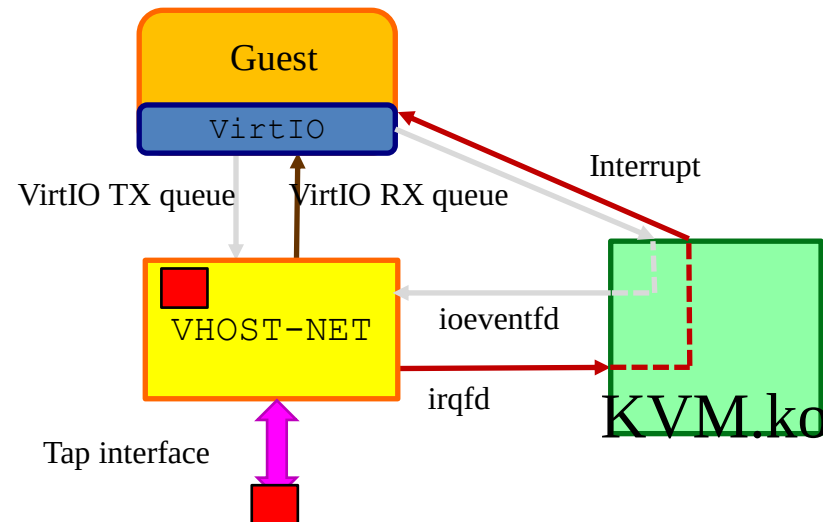
 *eventfd*: file descriptor used as an event wait/notify mechanism

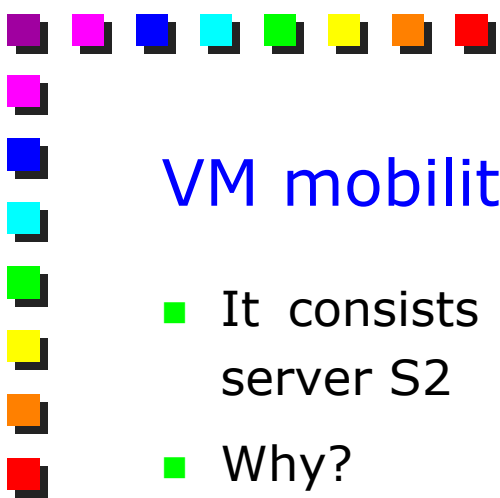




VHOST-NET - cont'd

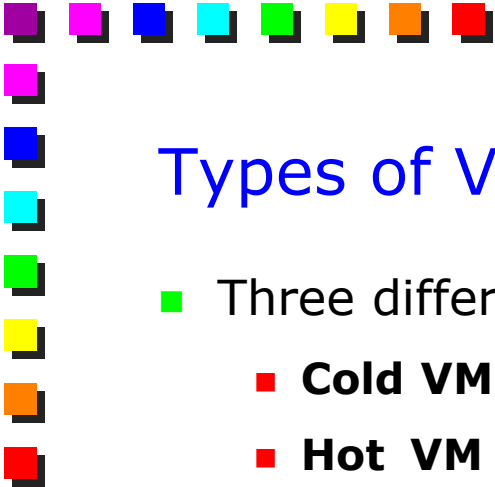
- On the receive path, the vhost worker thread has to interrupt the guest
- The vhost worker thread writes a file descriptor in order to kick the guest
- `kvm.ko` has a feature called *irqfd* that allows an *eventfd* file descriptor to trigger guest interrupt
- QEMU userspace registers a *irqfd* for the virtio PCI device interrupt and hands it to the VHOST instance





VM mobility

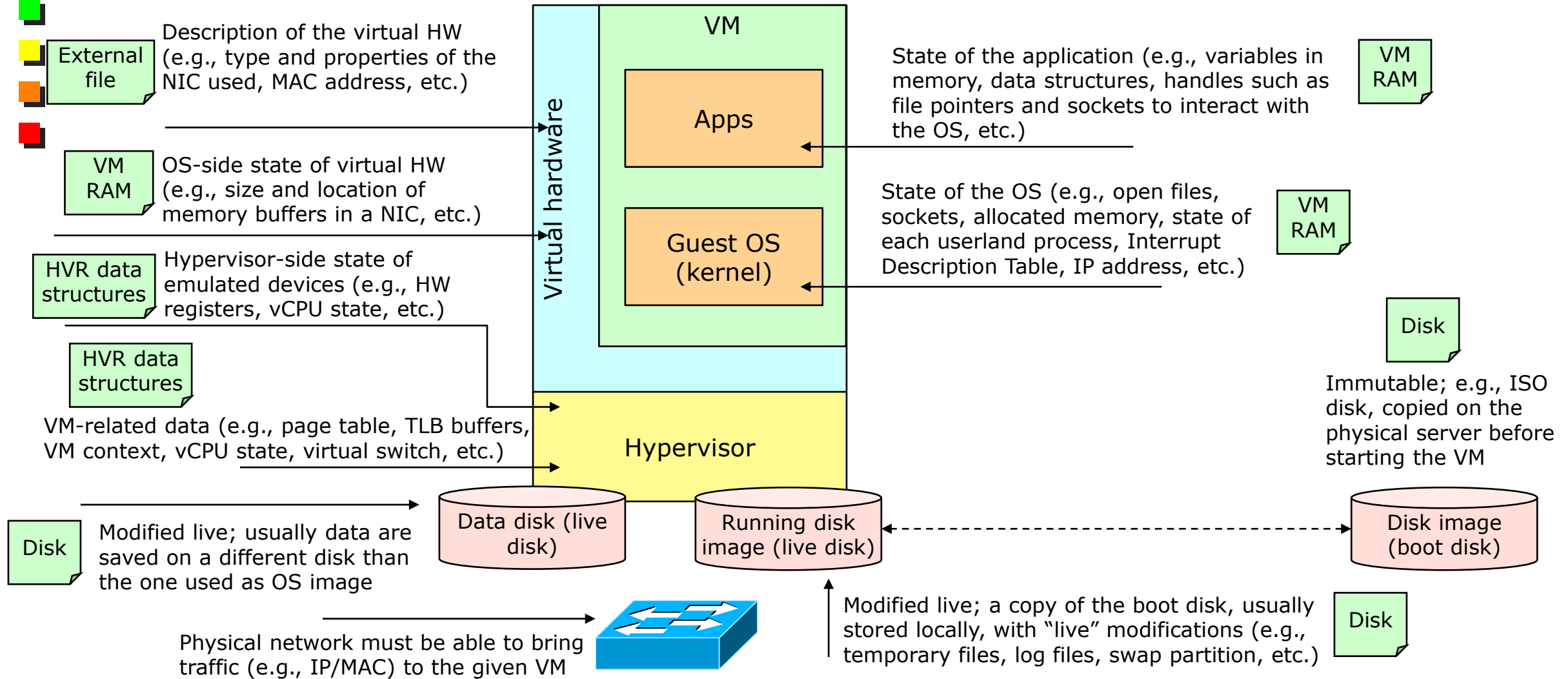
- It consists in taking a given VM, running on a server S1, and move to another server S2
- Why?
 - VM consolidation (on a few servers)
 - HW/SW maintenance (physical server may need to reboot)
 - High Availability / Disaster recovery
 -

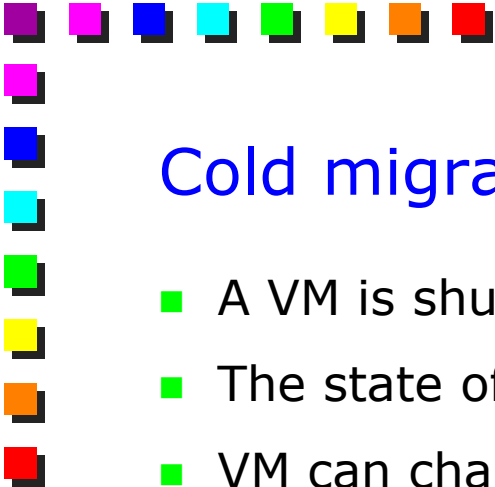


Types of VM mobility (or migration)

- Three different types (of increasing complexity)
 - **Cold VM migration:** a VM is shut down and restarted on another hypervisor
 - **Hot VM migration:** a running VM is moved from one hypervisor to another, *through PAUSE, MOVE, RESUME phases*
 - **Live VM migration:** a running VM is moved from one hypervisor to another, *without stopping the running VM*
- Often the term “VM migration” is used instead of “mobility”

An inside view of the state of the VM

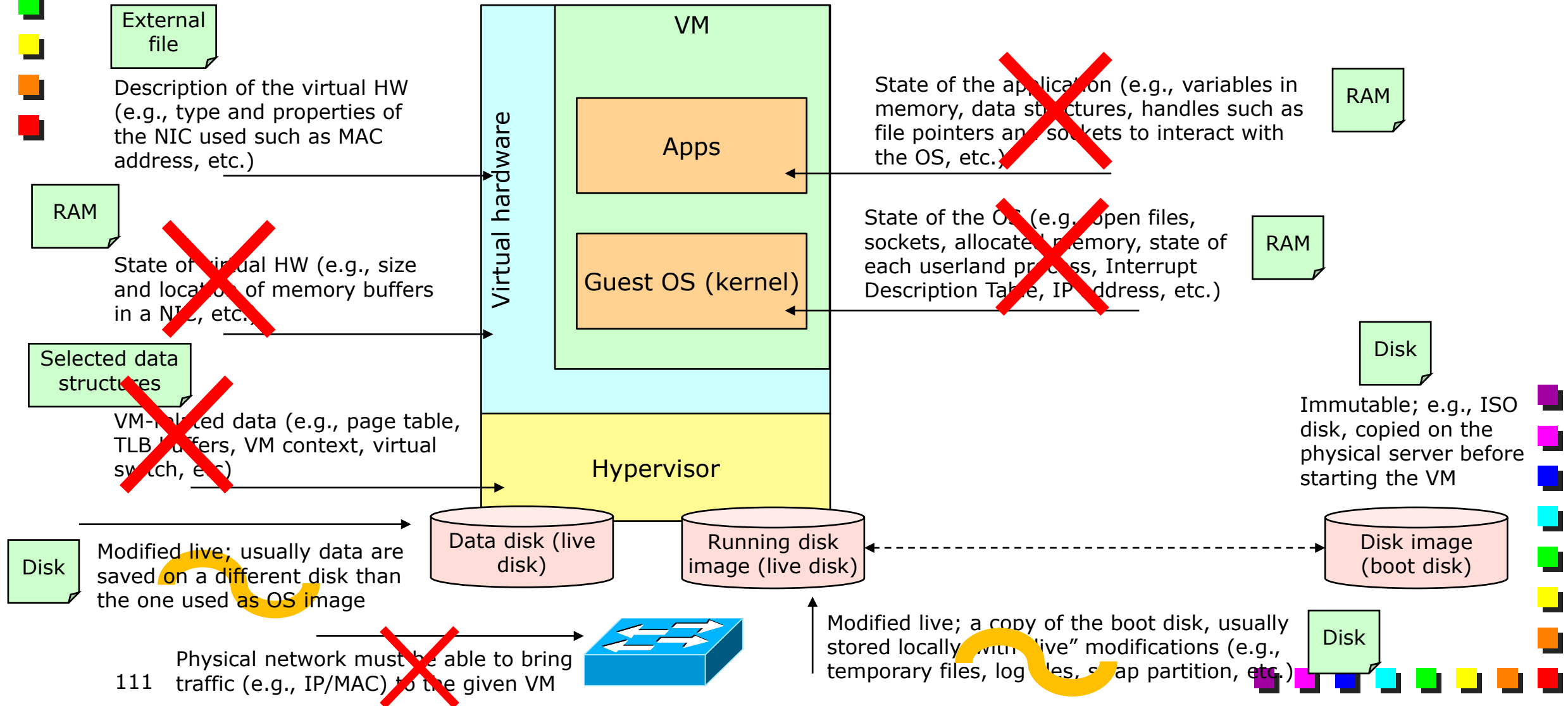


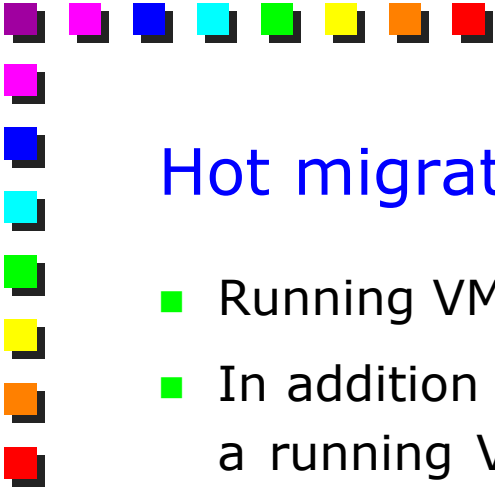


Cold migration

- A VM is shut down and restarted on another hypervisor
- The state of the VM and hypervisor do not need to be preserved
- VM can change IP address
- “Live” disks must be transferred, which is much easier if disks are not kept on the local server, but are stored on a remote location (e.g., iSCSI or similar)

Cold migration





Hot migration

- Running VM is moved from one hypervisor to another
 - In addition to data copied in the cold migration, we must copy the memory pages of a running VM to another hypervisor, and the state of the external resources (e.g., hypervisor, possible pass-through I/O devices)
 - We may need to re-configure the network
 - E.g., gratuitous ARP, assuming that the IP address of the VM does not change
 - However, here things may be more difficult than expected, based on the actual type of network connectivity (e.g., VM is connected to the physical network in a “direct bridge” mode, or through a NAT)
- 



Live migration

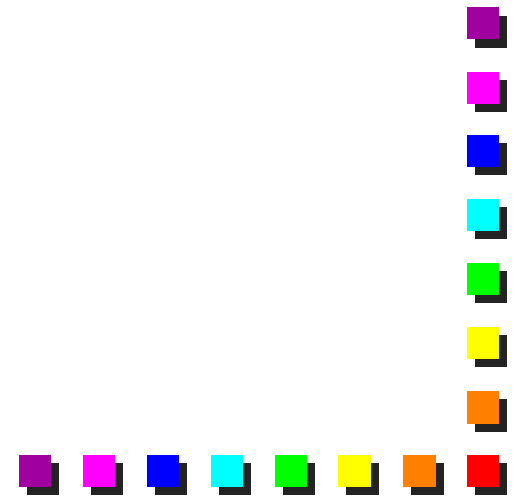
- Running VM is moved from one hypervisor to another, while **running**
- We must copy the memory pages of a running VM to another hypervisor, including pages that have been modified while the memory was transferred.
- Once copied most of the memory pages, the state of the VM is transferred to the destination hypervisor and restarted there





Part VII

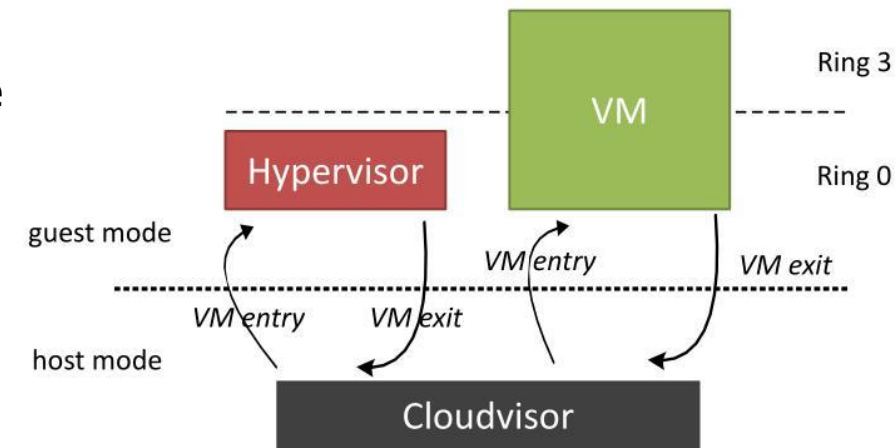
Nested Virtualization



Nested Virtualization

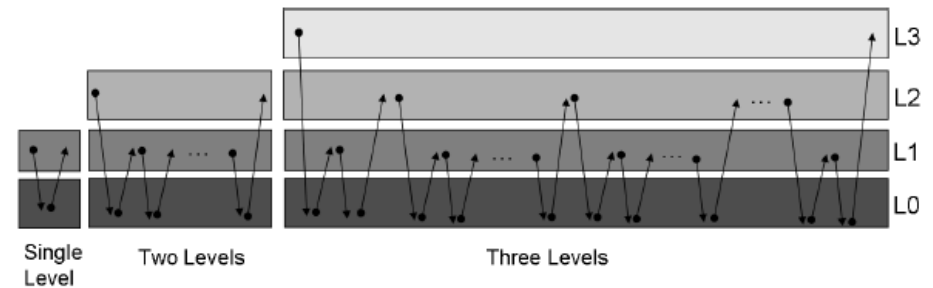
- Several use-cases require more than one layer of virtualization:
 - Continuous Integration of OS components (e.g. Device drivers)
 - Hypervisor-based security tools
 - Honeypots
 - Intrusion Detection
 - De-privileged hypervisors (Zhang et. Al, CloudVisor, 2011)
- Several works have been proposed in literature to handle nested virtualization in x86-64.
 - The Turtles Project: Design and Implementation of Nested Virtualization, B. Yehuda et al, OSDI 2010
- Prototype have been backported or inspired nested support in QEMU/KVM

Interposition with CloudVisor



Nested Virtualization - CPU

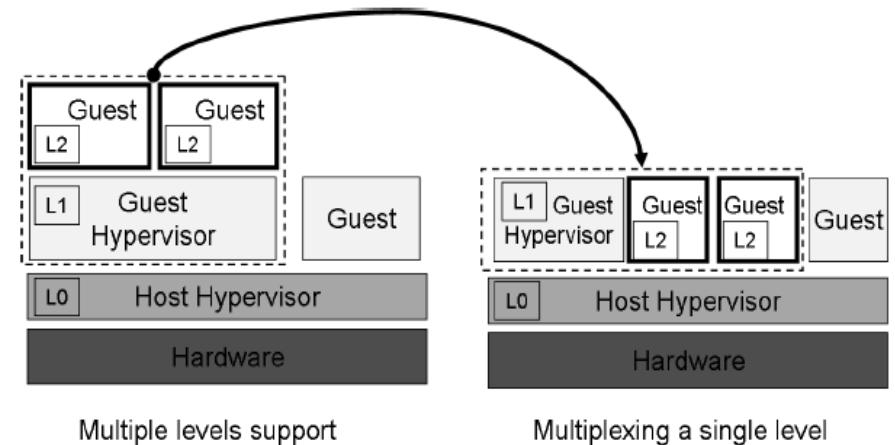
- The cost VMENTRY/VMEXIT
 - Introducing one further layer of virtualization, the number of VM exits increases linearly
 - When the hypervisor L1 tries to execute hypervisor code, thinking to be in a privileged context, this causes traps to L0 in order to correctly emulate VMX instructions
- This requires to extend the capabilities of L0 hypervisors
 - Shadow VMCS
 - Nested EPT



[3]

Nested Virtualization – VMCS management

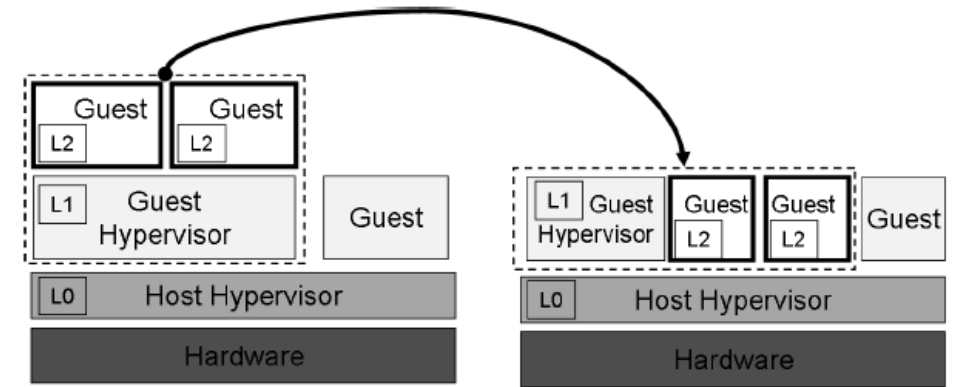
- A great part of the introduced overhead is due to the fact the many instructions executed by the L1 hypervisor trigger VMEXIT/VMENTRY to L0.
- L0 normally prepares a VMCS₀₋₁ to execute a guest L1 with a single level of virtualization
- In Nested Virtualization, when a new nested guest is created, the virtualized hypervisor tries to do the same for the nested guest L2, creating a VMCS₁₋₂.
- The L0 has to create a new VMCS₀₋₂, that represent the L2 state from the L0 perspective and update the former VMCS₀₋₁ to correctly trap and emulate interactions between L1 and L2.



[3]

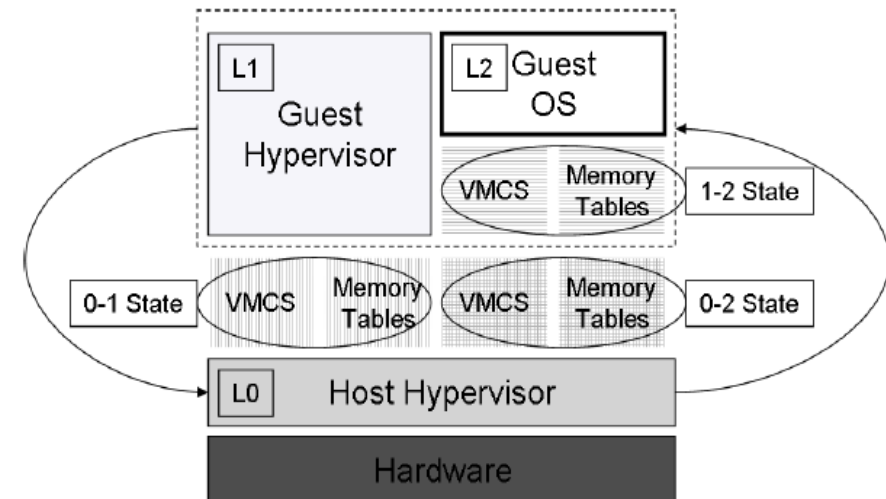
Nested Virtualization – VMCS management

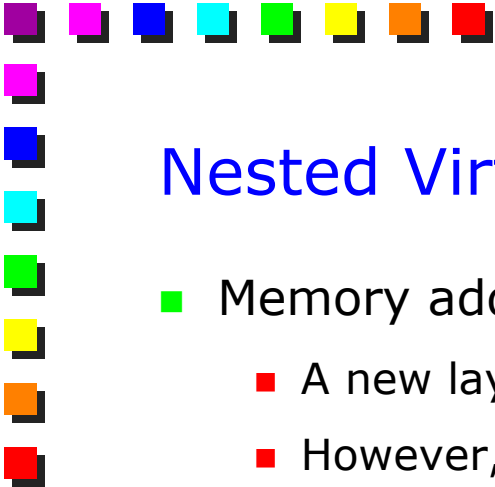
- The control data of the VMCS₁₋₂ and VMCS₀₋₁ has to be correctly merged in order to have a consistent behavior of the emulated processor in VMCS₀₋₂ (e.g. Consistent VMExit causes).
- Due to the Intel/AMD specification, every time the L1 hypervisor access VMCS_{1→2}, a VMExit is produced.
- Several optimizations are possible: Shadow VMCS (KVM) or para-virtualization (Turtles paper).
- Shadow VMCS, implemented in KVM, consists in defining a special VMCS structure which does not trigger every time VMEXITS. When L1 reads this VMCS structure, this does not trigger a VMEXIT.



Multiple levels support

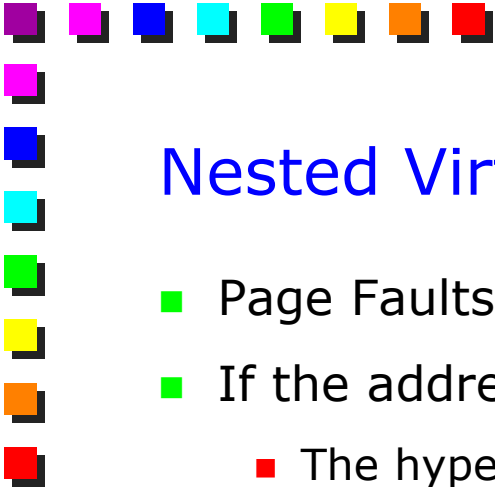
Multiplexing a single level





Nested Virtualization – EPT on top of EPT

- Memory address translation becomes more complex in nested virtualization
 - A new layer of translation is required to obtain physical addresses
 - However, EPT/NPT technologies does not provide support for more than two layers
 - EPT technologies can be adapted to obtain the desired 3 levels of indirections
 - Require ad-hoc trap to prevent the virtualized hypervisor to manipulate EPT
 - L0 hypervisor normally creates EPT0-1 for Hypervisor L1
 - L0 exposes the support for EPT to L1, which creates the EPT for L2 (EPT1-2)
 - Every modification of EPT1-2 should be trapped by the L0 to keep the EPT0-2 updated
 - L0 is now capable to create a compressed version of EPT tables mapping addresses of L2 to the physical addresses, reducing the number of page fault exits (which are very costly)
- 



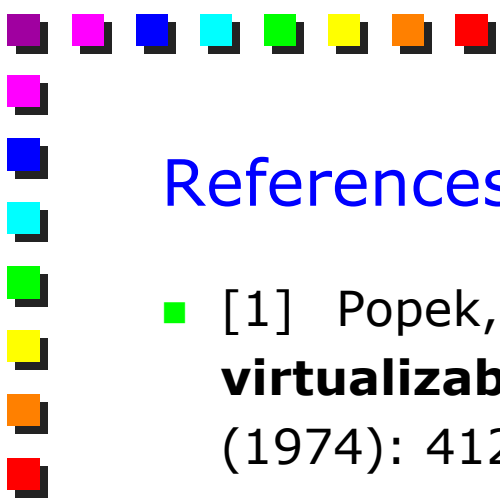
Nested Virtualization – EPT on top of EPT

- Page Faults are quite tricky to handle
- If the address has no available translation in EPT1-2
 - The hypervisor L0 lets L1 handle that by updating EPT1-2
 - L0 can now update EPT0-2, mapping the virtual address to a physical page
- Cache invalidation instructions invoked by L1 should be also trapped by L0 in order to correspondingly invalidate its cache.



Take-away messages

- Virtualization makes computer securely shared across different tenants
 - This is done by virtualizing CPU, Memory and Devices in Virtual Machines
- Virtualization technologies are often evaluated considering the **degree of isolation** and **performance overhead**
 - Hardware mechanisms can dramatically boost VM performance
- Slightly modification to virtualized OS can bring massive benefits in terms of speed up and flexibility (Para-virtualization)
 - PV Device drivers can be easily integrated in Operating System



References

- [1] Popek, Gerald J., and Robert P. Goldberg. **"Formal requirements for virtualizable third generation architectures."** Communications of the ACM 17.7 (1974): 412-421.
 - [2] Barham, Paul, et al. **"Xen and the art of virtualization."** ACM SIGOPS operating systems review. Vol. 37. No. 5. ACM, 2003.
 - [3] Ben-Yehuda, Muli, et al. **"The Turtles Project: Design and Implementation of Nested Virtualization."** Osd. Vol. 10. 2010.
 - [4] **Nesting Virtual Machines in Virtualization Test Frameworks**, O. Berghmans, 2010
 - [5] **Performance Evaluation of Intel EPT**, Nikhil Bhatia (VMware), 2009
 - [6] <https://rayanfam.com/topics/hypervisor-from-scratch-part-4/>, *Sina Karvandi*
- 