



A few more things about system-level virtual machines

Renaud Lachaize



Acknowledgements

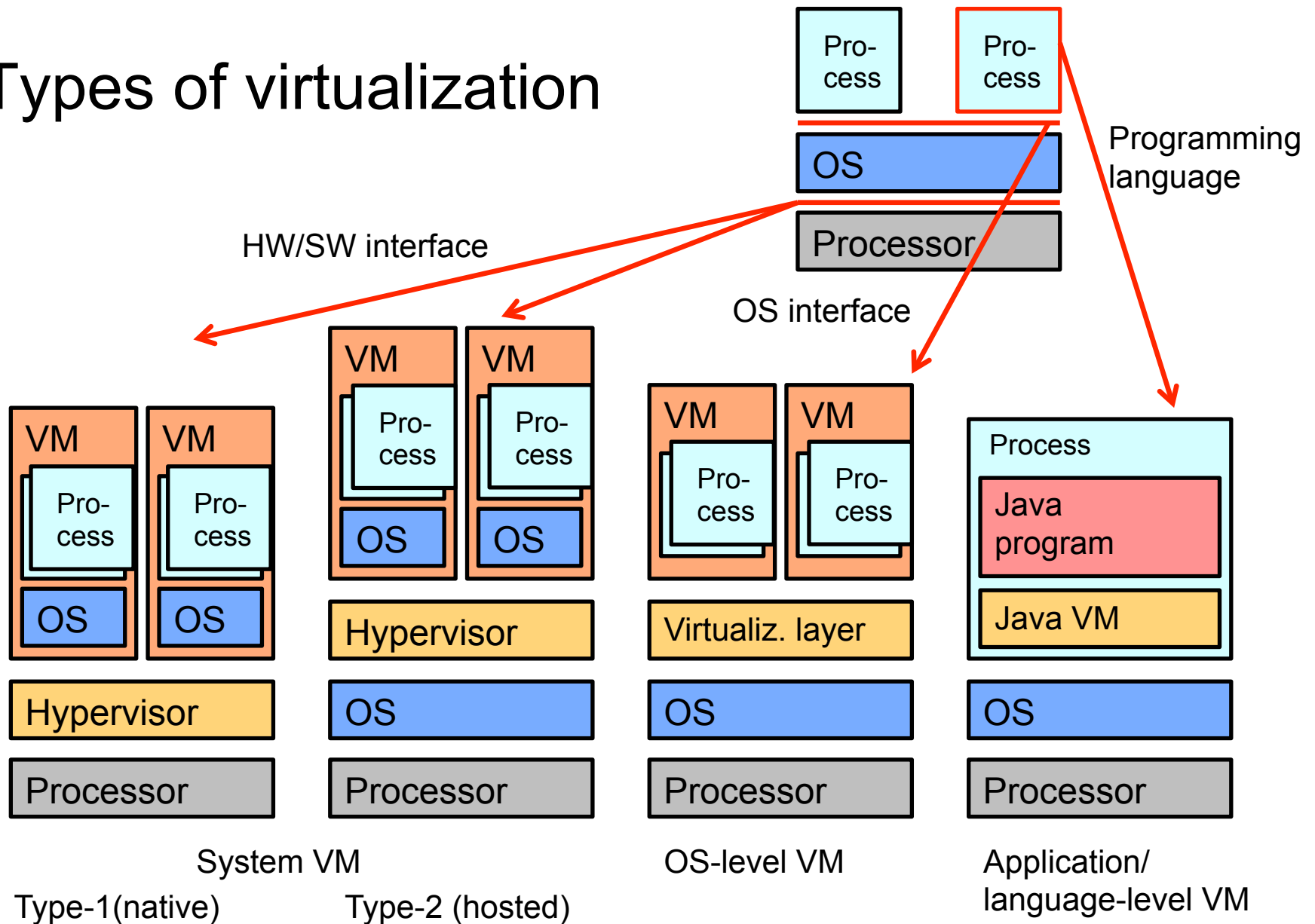
- These slides include material courtesy of:
 - Prof. Olivier Gruber, UJF
 - Prof. Gernot Heiser, UNSW (COMP9242 class)



Outline

- Type-I and type-II hypervisors
- Optimizations for memory virtualization
- I/O virtualization
- Dealing with problematic ISAs
 - Paravirtualization
 - ISA extensions for virtualization support
- Case studies

Types of virtualization





System-level virtualization

- A system-level **virtual machine (VM)** is an efficient, isolated duplicate of a real machine
 - CPU(s), memory, I/O devices
- **Duplicate:** Programs cannot distinguish between real or virtual hardware, except for:
 - Fewer resources (and potentially different between executions)
 - Some timing differences (when dealing with devices)
- **Isolated:** Several VMs execute without interfering with each other
- **Efficient:** VM should execute at speed close to that of real hardware
 - Usually requires that most instructions are executed directly by real hardware

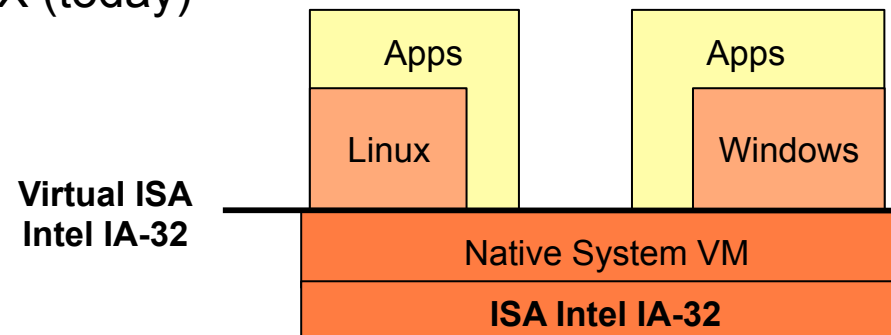


System-level virtualization (continued)

- **Note:** This lecture focuses on system-level virtualization in which the VM exports the same Instruction Set Architecture (ISA) as the underlying real machine (or a subset of this ISA)
- **Virtual Machine Monitor (VMM) or Hypervisor**
 - Generic terms for the software implementing the VM
 - Unless explicitly mentioned otherwise, we will use “VMM” and “Hypervisor” interchangeably
 - As often (but not always) in the literature
- Two main types of same-ISA VMMs
 - Classic VMMs, also known as “Type-I” or “Bare metal” or “native”
 - Hosted VMMs or “Type-II”

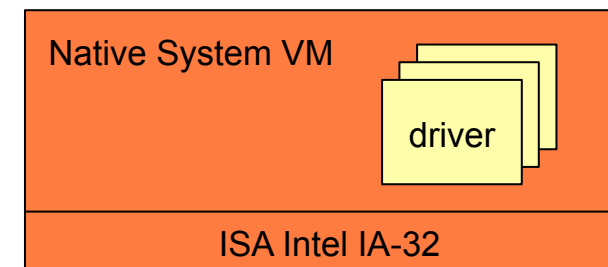
Type-I Virtual machine monitor

- Directly on bare hardware
 - Multi-tasking of virtual machines
 - Mediate access to shared resources
- Run in kernel mode
 - **All other software runs in user mode**
 - Including guest operating systems in the virtual machines
 - Privileged instructions in guest cause a trap into VMM
- Examples:
 - IBM CP/CMS (1960s) and VM/370 (1970s)
 - VMware ESX (today)



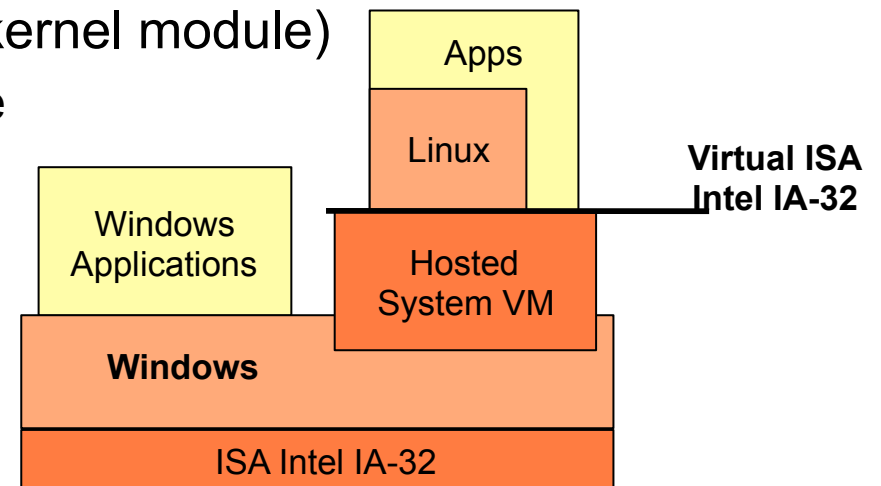
Discussing Type-I VMMs

- Must be installed on bare metal
 - Need to wipe out any pre-installed operating system
- Usually must have the adequate drivers
 - The VMM virtualizes the I/O devices
 - It is therefore the VMM that interfaces directly with the I/O devices
 - So it needs drivers for the hardware
 - The VMM can virtualize generic I/O devices
 - From specific hardware ones



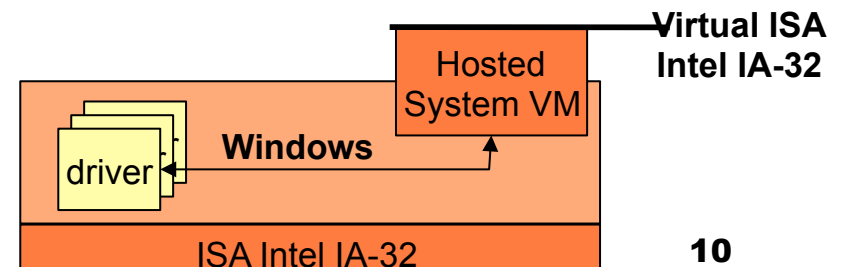
Type-II Virtual machine monitor

- Installed on a host operating system
 - As a “regular” application
 - Hosts one or several guest operating systems
 - Examples: VMware workstation/fusion/player, VirtualBox
- Runs in mix mode
 - Most of the VMM code runs in user mode
 - Some in kernel mode (through kernel module)
 - In order to improve performance

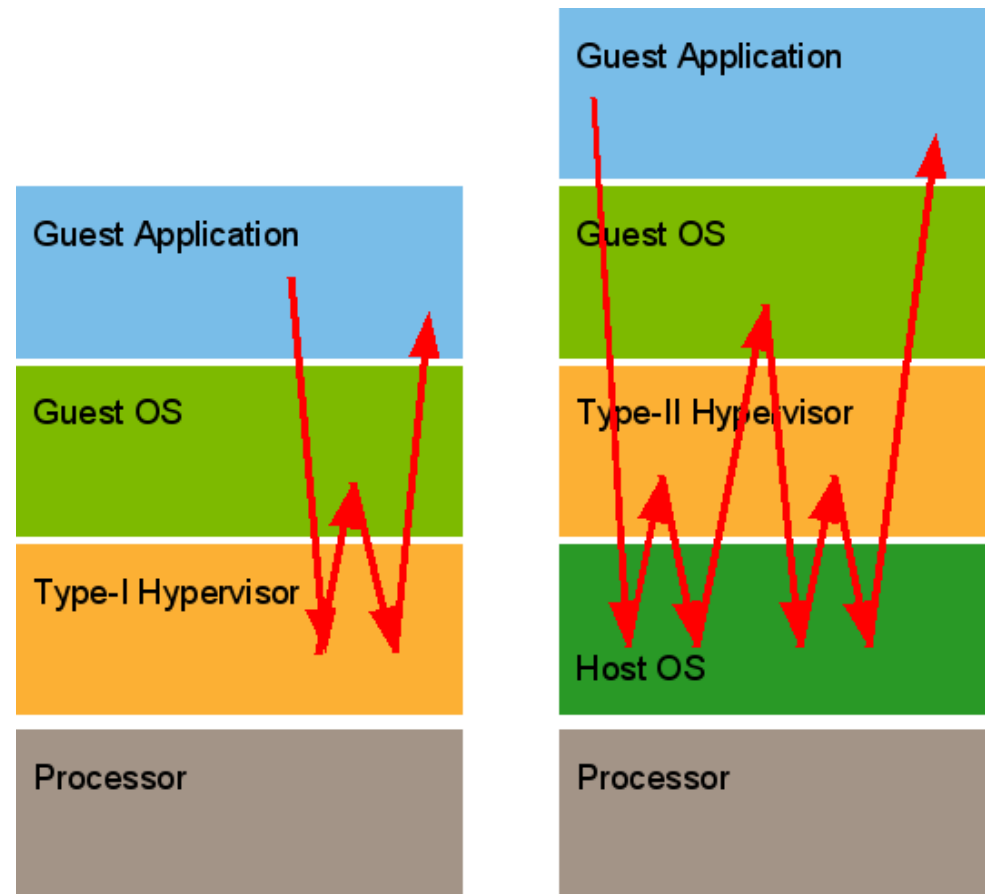


Discussing Type-II VMMs

- Can be installed on an existing operating system
 - End users do not need to wipe-out their disk
 - Can leverage the drivers from the host operating system
 - Can virtualize generic devices (like type-I VMMs)
- Can integrate in the host operating system
 - Can appear as a window on the host desktop
 - Can even provide cut & paste abilities
- Generally less efficient than native/type-I approach
 - See next slide



Mode switches in Type-I and Type-II hypervisors





Scheduling: Virtual CPUs

- A VMM provides a guest VM with the illusion of one or several virtual CPUs (“vCPUs”)
 - Akin to single-threaded vs multi-threaded process in a traditional OS
- Physical CPU management
 - Uniprocessor: the VMM time-shares the CPU between the different vCPUs (possibly related to different guest VMs)
 - Multiprocessor:
 - Physical partitioning: static and exclusive mapping between a given vCPU and a given pCPU
 - Logical partitioning: time sharing of a pCPU between multiple vCPUs
 - VMM can possibly implement a mix of both policies (and dynamically switch between them for a given CPU)



Outline

- Type-I and type-II hypervisors
- **Optimizations for memory virtualization**
- I/O virtualization
- Dealing with problematic ISAs
 - Paravirtualization
 - ISA extensions for virtualization support
- Case studies



Optimizations for memory virtualization

- We will study two different kinds of optimizations:
 - 1) How to decrease the performance overhead of shadow page table management?
 - 2) How to (efficiently) overcommit “guest-physical” memory?



Optimizations for memory virtualization

- We will study two different kinds of optimizations:
 - 1) **How to decrease the performance overhead of shadow page table management?**
 - 2) How to (efficiently) overcommit “guest-physical” memory?



Optimized management of shadow page tables

■ **Context: machine with architected page table**

- Here, we use “page table” as a generic expression to refer to the data structure that defines the virtual memory mappings within an address space
- Processor hardware is aware of page table format and automatically queries page table in memory when necessary
- Example: Intel processors

■ **How to virtualize such a machine?**

- If there is no hardware support for virtualization, the VMM must manage shadow page tables (studied in previous lectures)
- We will briefly review the principles and then discuss potential optimizations



A reminder on shadow page tables (1/5)

- MMU configuration

- ☐ Guest OS believes that it is running directly on the raw hardware (with full privileges) and that it can configure the page tables (PTs)
- ☐ ... but, in fact, **only the VMM can configure the hardware MMU**
- ☐ In other words:
 - The guest kernel allocates and sets up PTs in RAM
 - ... but the VMM uses its own set of PTs (based on, but not identical to the PTs of the guest) to configure the hardware MMU



A reminder on shadow page tables (2/5)

- The guest OS manages, for each process, one PT to define the mappings between virtual addresses and “guest-physical” addresses (a.k.a. “real” addresses)

- The VMM manages:
 - For each guest VM: mapping between “guest-physical” addresses and “host-physical” (i.e., truly physical) addresses
 - For each process of each guest VM: mapping between (guest) virtual addresses and (host) physical addresses
 - This mapping is used to configure the hardware MMU (because the MMU is only aware of one level of memory virtualization)

A reminder on shadow page tables (3/5)

guest-virtual
to host-physical
mappings

1000	500
2000	1000
7000	1000

(hardware) MMU page table
(configured by VMM,
according to the other tables)

1 per guest process

guest-virtual
to guest-physical
mappings

1000	5000
2000	1500
7000	3000

Guest page table
(configured by guest OS,
monitored by VMM)

1 per guest process

guest-physical
to host-physical
mappings

1500	1000
3000	1000
5000	500

Managed by VMM

1 per guest VM



A reminder on shadow page tables (4/5)

- When must the VMM update the PTs used by the hardware MMU?
 - (1) When there is a change in the “virtual to guest-physical” mappings of a process
 - Such changes are triggered by the guest OSes
 - Update of current process address space
 - Or switch to another process address space
 - **Must be detected by the VMM** (details on next slides)
 - (2) Also when there is a change in the “guest-physical to host-physical” mappings of a guest VM
 - Such changes are triggered by the VMM (so, easy to notice by the VMM)



A reminder on shadow page tables (5/5)

- The VMM must keep track of accesses to PT by guest OSes
 - Write accesses must be caught by VMM
 - Necessary to notice the modifications of “virtual to guest-physical” mappings and update the hardware MMU PT accordingly
 - Read accesses must also be caught
 - Otherwise, guest OS may notice that the contents of the hardware MMU PT are different from what it believes (i.e., the VMM would not be invisible to the guest)




How can the VMM detect the accesses to PTs by the guest OSes?

- Attempts by guest OS to read/write “current PT root” register (e.g., CR3 on Intel) trigger traps
- Attempts by guest OS to read/write guest PT pages (i.e., pages that belong to the PT) can trigger traps
 - If VMM configures the hardware MMU PT to forbid read and write accesses to such pages for the guest



How to efficiently manage shadow page tables?

- So far, we have described an approach that consists in trapping to the VMM upon any guest access to guest PT
 - (as explained on the previous slides)
- However, such an approach is costly (many traps, each with a significant overhead)
 - We can do better (less traps)
 - We will study two more efficient approaches:
 - Lazy shadow update
 - “Virtual TLB”



Efficient management of shadow page tables

The “lazy shadow update” approach (1/3)

■ Observation:

- When the guest kernel updates a PT (either for adding new mappings or invalidating old ones), it often performs several consecutive updates
 - For example, a given mapping may span multiple page table entries
- Usually, a PT update must only become effective once the guest kernel transfers control back to user mode

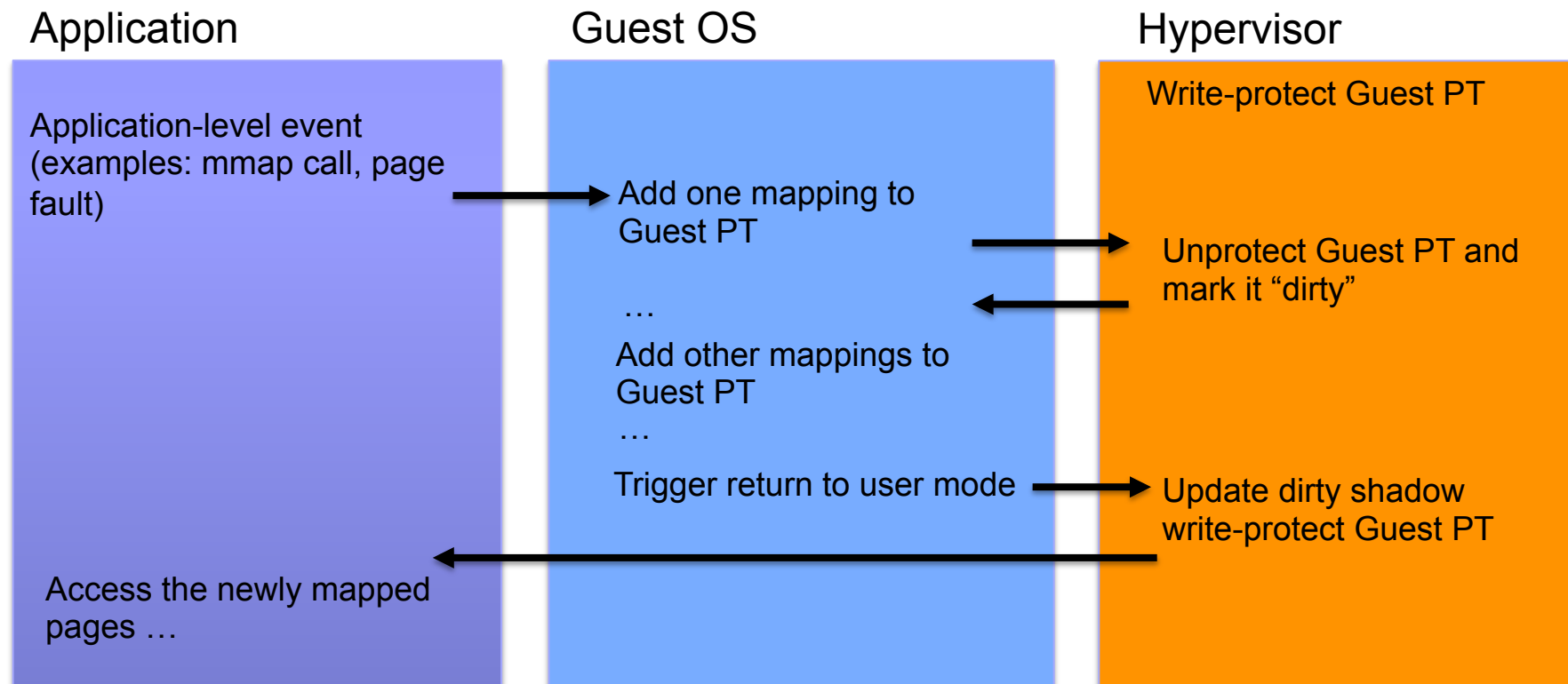
■ Idea:

- Control guest PT write protection (configured by VMM) so that we do not trigger 1 trap per update (instead, only 1 trap per group of updates)
- Upon first modification, disable guest PT write-protection
- Upon synchronization point, re-enable guest PT write-protection

Efficient management of shadow page tables

The “lazy shadow update” approach (2/3)

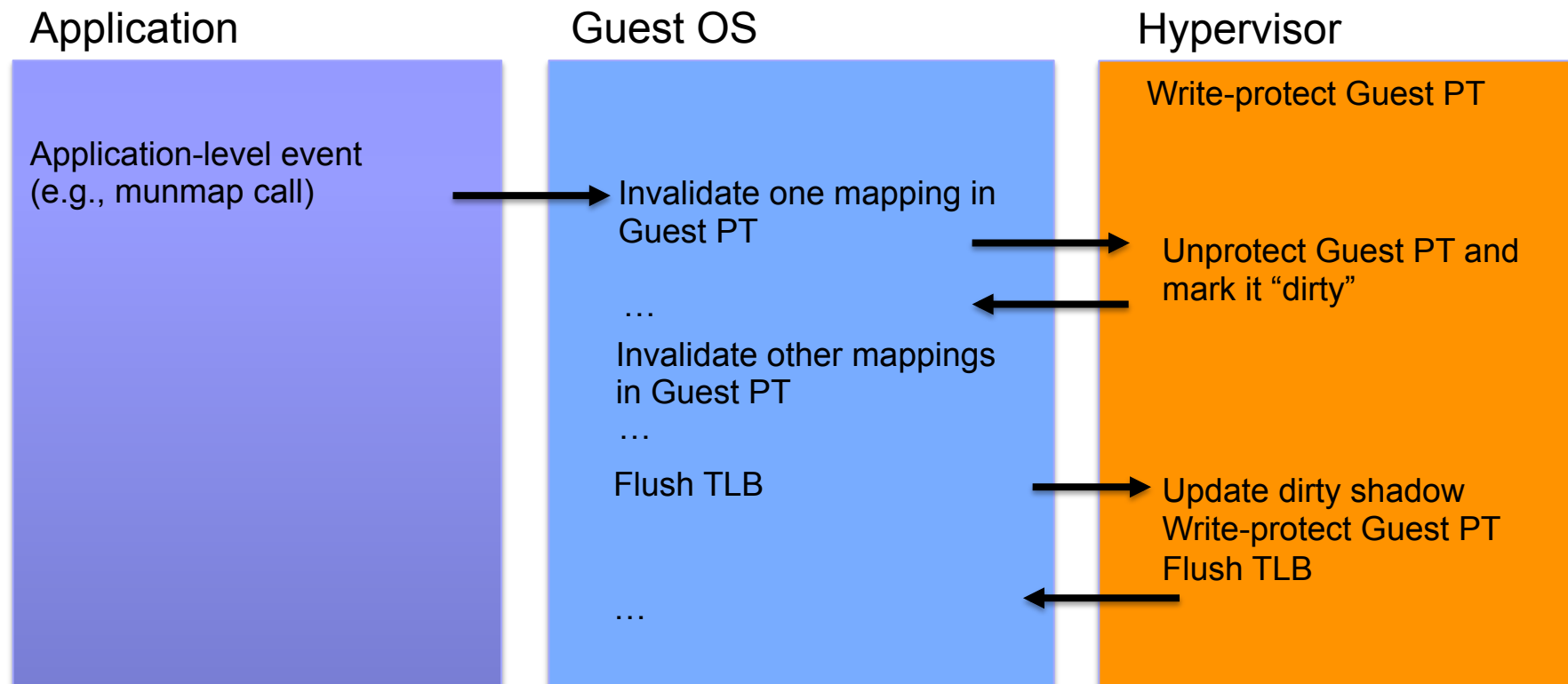
■ Case 1: Adding new mappings



Efficient management of shadow page tables

The “lazy shadow update” approach (3/3)

■ Case 2: Invalidating mappings





Efficient management of shadow page tables

The “Virtual TLB” approach (1/7)

- Main idea:

- ☐ The guest PT does not have to be always immediately synchronized with the hardware MMU PT
- ☐ So, we can avoid some traps

- More precisely:

- ☐ What happens if the guest OS adds a new mapping in guest PT without synchronizing with the MMU PT?
 - First access to new mapping by the guest application will trigger trap to VMM
 - The VMM can then check the guest PT and the VMM information and, if OK, update the MMU PT accordingly



Efficient management of shadow page tables

The “Virtual TLB” approach (2/7)

- More precisely (continued):
 - The MMU PT configured by the VMM can be “out-of-sync” with respect to the guest PT as long as the following conditions are respected:
 - The MMU PT must never contain mappings that have been destroyed in the guest PT
 - The MMU PT must never contain mappings whose permissions have become more restrictive in the guest PT
 - For example, the MMU PT must not keep the mapping for a page set to read-write if the guest PT has modified it to read-only



Efficient management of shadow page tables

The “Virtual TLB” approach (3/7)

- In other words:
 - **The MMU PT set by the VMM can be seen as a “cache”**
 - This cache holds mappings between guest-virtual and host-physical addresses
 - **This cache cannot contain “stale” information**
 - I.e., the cached information should not lead us to perform address translations that are not allowed anymore
 - If the guest OS destroys a “guest-virtual to guest-physical” mapping, the cache must be immediately updated
 - If the VMM destroys a “guest-physical to host-physical mapping”, the cache must be immediately updated (but this is cheap because we are already in the VMM context – No trap required)
 - The same rules apply for “demoted” mappings: if a mapping becomes more restrictive (i.e., from read-write to read-only, or from user to kernel-only), the cache must be immediately updated



Efficient management of shadow page tables

The “Virtual TLB” approach (4/7)

- In other words (continued):
 - **The contents of the cache may be incomplete. It may lack:**
 - Some mappings that have been recently added
 - Some mappings that have been recently “promoted” (i.e., made more permissive: from read-only to read-write, or from kernel to user)
 - In such cases, a trap will occur and the VMM will check the reference information (contents of guest PT + VMM information on “guest-physical to “host-physical” mappings) and, if necessary, update the cache



Efficient management of shadow page tables

The “Virtual TLB” approach (5/7)

- Overall, the hardware MMU PT configured by the VMM has the same kind of behavior as a TLB in a traditional OS
 - I.e., a cache with “weak consistency” semantics
 - Hence the name “Virtual TLB” (VTLB)
- For traditional TLB, reference information (checked upon TLB miss) = page table of current process
- For VTLB, reference information = current guest PT combined with VMM table on guest-physical to host-physical mappings (for current guest)



Efficient management of shadow page tables

The “Virtual TLB” approach (6/7)

- **VTLB consists in the hardware MMU PT + the processor TLB**
 - VMM initializes an empty VTLB and starts guest execution
 - A memory access from guest may trigger trap to VMM (page fault)
 - VMM checks reference information and, if access OK, copies the needed translation in the hardware MMU PT (= VTLB refill), then resumes guest execution
 - Note: VTLB is just a cache, so the VMM is allowed to discard some hardware MMU PTs and restart from empty state



Efficient management of shadow page tables

The “Virtual TLB” approach (7/7)

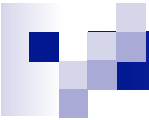
■ Synchronization points:

- **Upon page faults triggered by guest applications:** to update (fill in) VTLB with new or promoted mappings
- **Upon TLB flushes triggered by guest OS:** to remove obsolete mappings (or demote old permissions) in VTLB
- **Upon update of the “current PT root” register** (e.g., CR3 on Intel) **triggered by the guest OS:** this means that we are switching to another process address space so we must flush the VTLB



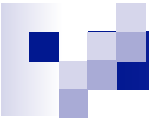
Optimizations for memory virtualization

- We will study two different kinds of optimizations:
 - 1) How to decrease the performance overhead of shadow page table management?
 - 2) **How to (efficiently) overcommit “guest-physical” memory?**



Efficient management of overcommitted guest-physical memory (1/5)

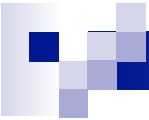
- We assume a Type-I VMM (i.e., native)
- **Context: Server consolidation**
 - Many physical servers under-utilized
 - Use (system) virtual machines to improve hardware utilization and lower costs (hardware, electricity, ...)
 - Put many VMs on the same physical server
- Over-committing memory
 - Like “classical” virtual memory
 - **Sum of capacities of guest-physical RAMs > capacity of physical RAM**
 - The VMM can swap out pages of the guest-physical RAMs
 - Allows “aggressive” server consolidation



Efficient management of overcommitted guest-physical memory (2/5)

– **Page sharing for space savings**

- Fact: Multiple VMs running the same guest OS have a lot of common pages
 - Examples: text segments, zeroed pages
- **Idea: Keeping a single copy of such duplicate pages in physical memory would help freeing some space**
- ... which would allow increasing the amount of over-committed memory
- **How to do it?**
- The VMM detects pages with the same content (using hashing and then full comparison)
 - Mostly performed as a background activity
 - But sometimes on-the-fly (e.g., before swapping out a page)
- Copy-on-write allows mapping duplicates to a single physical copy



Efficient management of overcommitted guest-physical memory (3/5)

– **Memory reclamation**

- Because of the memory over-commitment, the VMM may have to reclaim physical memory pages from a guest VM
- **Idea: let the guest OS decide about the most relevant pages to be evicted**
- Why?
- Because making eviction decisions only at the VMM level may result in bad performance
 - Poorly informed choices, hard to find a good generic page replacement policy
 - “Double paging” issue: VMM may decide to page out a given page just before the guest OS decides to page out the same page



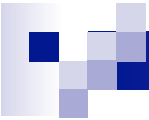
Efficient management of overcommitted guest-physical memory (4/5)

– Memory reclamation (continued)

- **Principle (“ballooning”) :**

- Load a pseudo-device driver into guest, which collaborates with the VMM
- To reclaim memory, the VMM instructs the driver to request memory (“inflate balloon”)
- The guest OS will itself select pages to be swapped out
- The VMM can then reuse the memory hoarded by the balloon driver
- The balloon can be later deflated when the memory pressure is reduced

See VMware ESX paper [Waldspurger] for further details



Efficient management of overcommitted guest-physical memory (5/5)

■ Uncooperative memory reclamation

- Ballooning is efficient but is not always possible
 - Guest may not have/want balloon driver
 - Balloons may reach an upper bound (memory limit)
 - Balloon may not be able to react fast enough
 - Balloon may be temporarily unavailable due to inner guest activity (e.g., booting)
- So the VMM may have to make eviction decisions by itself regarding the guest pages
- This raises many issues (beyond the previously-described “double paging” problem)
- See the Vswapper paper [Amit et al.] for details on the problems and their solutions

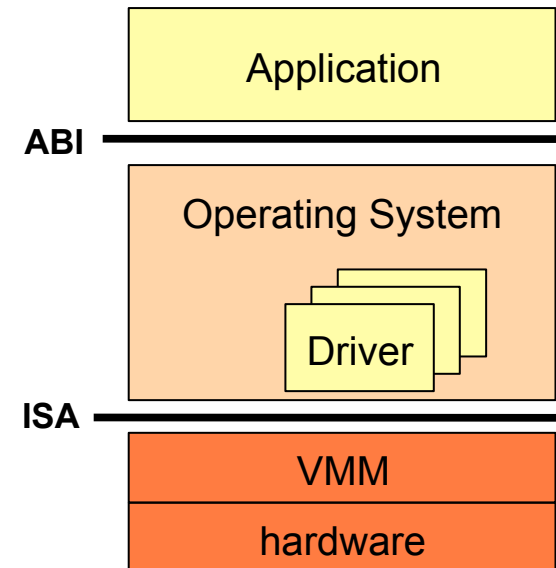


Outline

- Type-I and type-II hypervisors
- Optimizations for memory virtualization
- **I/O virtualization**
- Dealing with problematic ISAs
 - Paravirtualization
 - ISA extensions for virtualization support
- Case studies

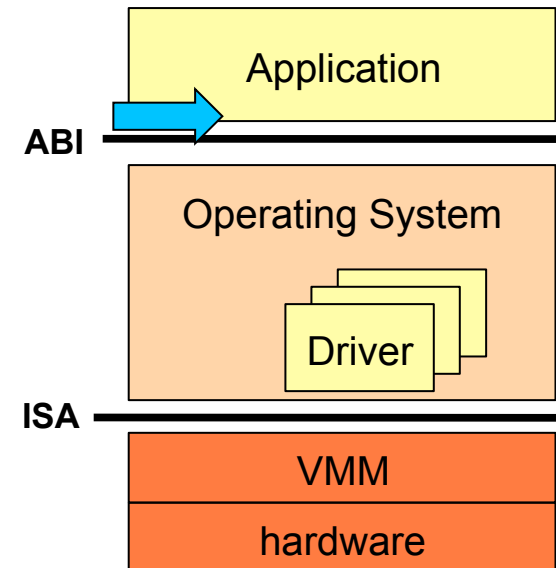
Virtualizing I/O activity

- Three possible levels
 - At the system call interface (ABI)
 - At the device driver interface
 - At the operation-level interface (ISA)



Virtualizing I/O activity

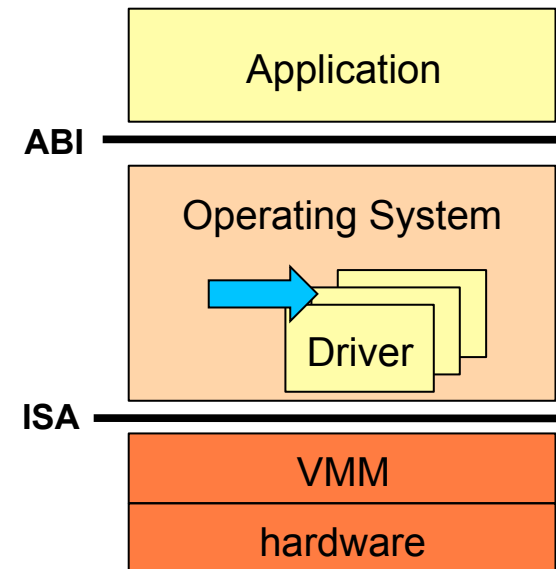
- At the **system call interface**
 - Could be more efficient in theory
 - Capture the original I/O at the ABI level
 - Emulate it entirely in one shot
 - Daunting task
 - The VMM needs an ABI mirroring the guest OS ABIs with many system calls
 - Very OS specific
 - Need precise emulation of the different I/O behaviors of the different guest OS
 - Can only be done if the VMM team has intimate knowledge of the guest operating systems



Virtualizing I/O activity

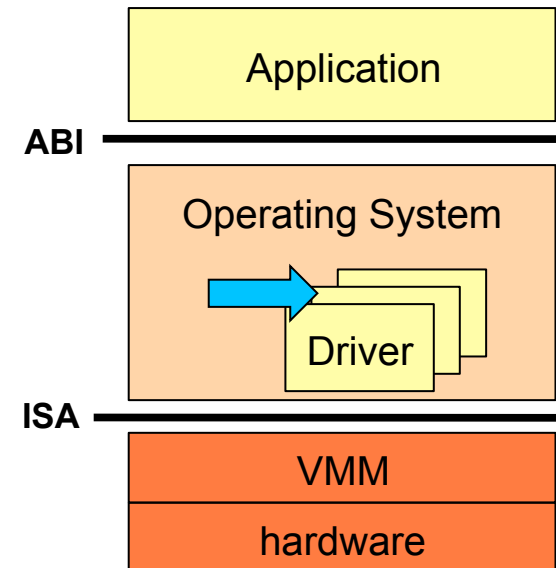
- At the **device driver interface**

- A natural interception point
 - Easy to emulate and redirect calls to the driver of the physical device
 - A guest driver simply redirects requests to the VMM using traps
- Also provides opportunities for performance optimizations
 - The device driver in the guest knows that there is an underlying VMM
 - The code of the guest driver can be optimized to minimize the number of traps to the VMM
- This approach is also known under other names:
 - “Split driver” (driver stub in guest interacting with VMM)
 - “Paravirtualized I/O” (because the guest driver is aware of the underlying VMM)



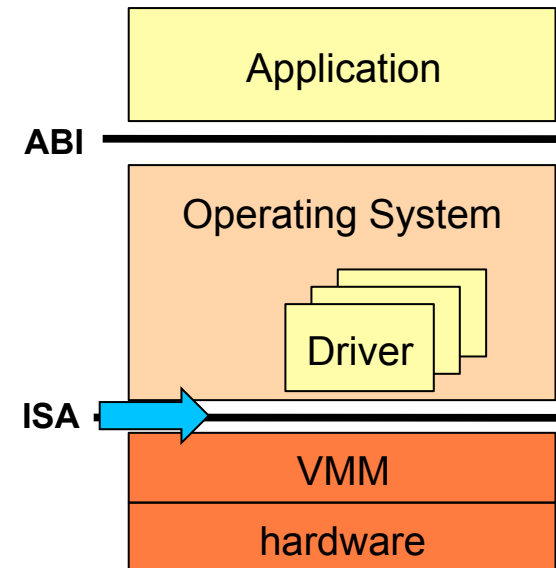
Virtualizing I/O activity

- At the **device driver interface** (continued)
 - Not general
 - Require some knowledge of the guest OS and of its internal device driver interface
 - Does not work if the VMM is intended to host arbitrary/esoteric operating systems
 - But often practical enough
 - VMM for mainstream operating systems
 - Such as Windows or Linux
 - The VMM only needs to support a small number of virtual devices (e.g., one type of virtual NIC, one type of virtual disk, etc.)



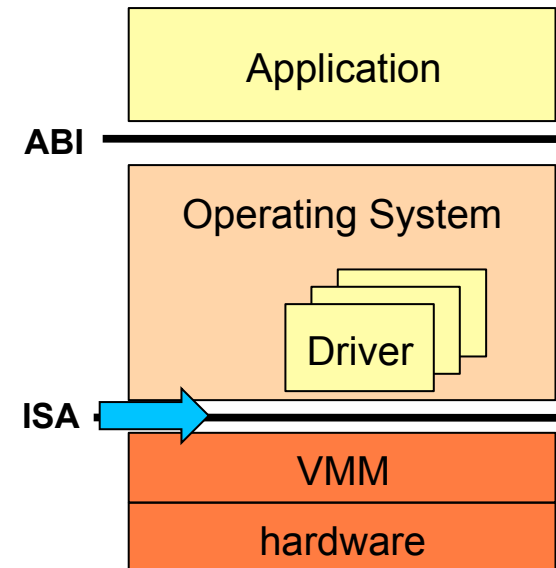
Virtualizing I/O activity

- At the **operation-level interface** (a.k.a. device interface)
 - Easy to intercept
 - Through special load/store instructions (e.g. in/out on IA-32) or regular load/store at special memory locations
 - Either privileged instructions or protected memory locations
 - The driver in the guest OS accesses the device “as usual”
 - Drawbacks
 - Performance issues (frequent traps) with many ISAs.
 - Thus, may not work with most devices (timing constraints violated)



Virtualizing I/O activity

- At the **operation-level interface** (continued)
 - Special (successful) example: IBM System/370
 - Well-defined ISA for I/O operations, using “channels” to I/O processors
 - With Channel Command Words (CCWs)
 - Includes an address in memory, length of data to transfer and a command
 - Use privileged instructions to send CCWs
 - Can be easily and efficiently virtualized





Virtualizing I/O activity

- **Regardless of the chosen interception level, the VMM must be involved to interact with the physical devices**
 - **Easy to implement for Type-II VMMs**
 - The VMM is just an application
 - Thus, the VMM can transparently leverage the device drivers provided by the underlying host OS
 - Typically through higher-level OS facilities (e.g., syscalls for file and socket management)
 - **What about Type-I VMMs?**
 - The VMM must implement mechanisms and policies for sharing the physical devices (between multiple guests)
 - The VMM must handle the interactions with the physical devices (see details on the next slides)

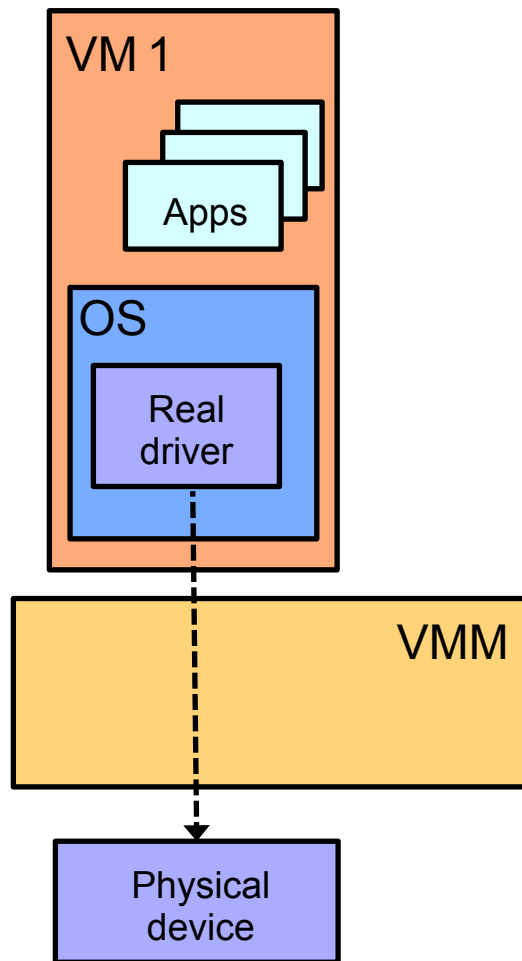


Virtualizing I/O activity

- Interaction with physical devices **for Type-I VMMs:**
 - Requires to reimplement/port all (real) drivers inside the VMM
 - Consequence: daunting task or limited portability of the VMM
- However, there are **possible workarounds:**
 - Using a complete OS as the VMM (e.g., Linux KVM)
 - “Pass-through”: grant direct device access to driver running in guest VM (may raise security issues – discussed/addressed later)
 - “Driver VM”: Using a specific VM to host the drivers

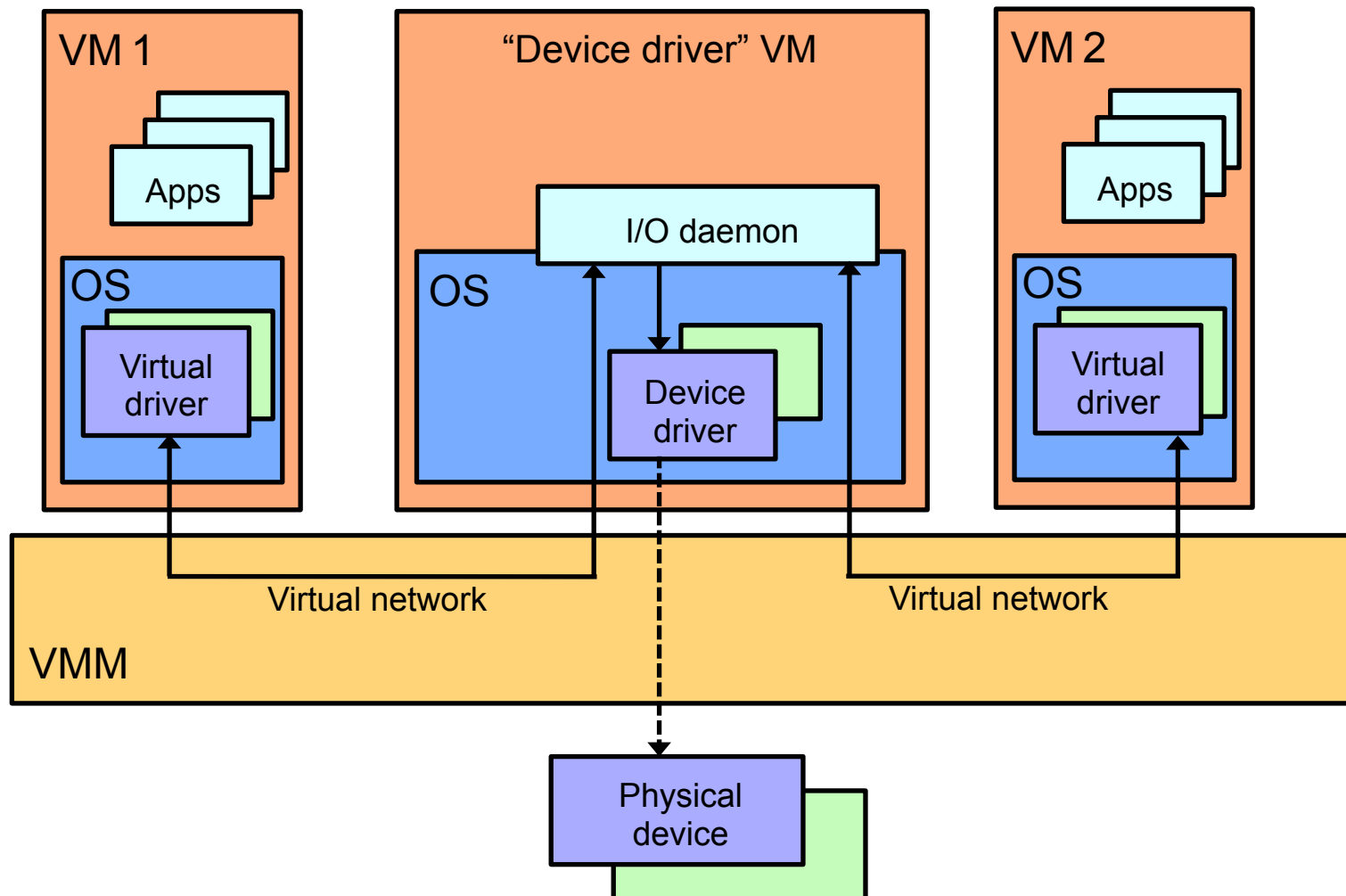
Virtualizing I/O activity

- “Pass through” approach



Virtualizing I/O activity

- “Driver VM” approach





Outline

- Type-I and type-II hypervisors
- Optimizations for memory virtualization
- I/O virtualization
- Dealing with problematic ISAs
 - Paravirtualization
 - ISA extensions for virtualization support
- Case studies



Dealing with problematic ISAs

- Non-virtualizable ISAs like Intel IA-32
- Basic solution: interpretation
 - Very slow
- More efficient approaches
 - Binary translation
 - Scan and patch sensitive instructions in guest binary code
 - Execute most of the instructions natively
 - Does not require guest source code
 - Paravirtualization
 - Hardware/ISA extensions providing support for virtualization



Para-virtualization (a.k.a. “impure virtualization”)

- Another solution to improve the emulation (performance) of “non-virtualizable” ISAs
- Idea: Modify ISA used by guest OS
 - Remove (i.e. do not use) instructions and ISA features which complicate virtualization
 - Use explicit calls to hypervisor (“hypercalls”)
 - Use a higher-level API to reduce the number of traps
- Examples: Denali, Xen
- Result: generally outperforms “pure” virtualization and binary translation
- Drawbacks:
 - Unlike binary translation, requires source code
 - Significant engineering effort (manual port of guest OS)
 - Needs to be repeated for each guest-ISA-hypervisor combination
 - Para-virtualized guest source code needs to be kept in sync with native guest source code



Para-virtualization

- Allows introducing new optimizations
 - Higher-level interfaces => less traps to the VMM
 - Additional tricks can also reduce the number of traps:
 - Relocating VM state (see example 1 below)
 - Lazy update of VM state (see example 2 next)
 - Example 1: Maintain some virtual machine state inside the VM
 - Relocate the interrupt-enable bit into virtual register
 - This requires changing the guest's idea of where this bit lives
 - In this way, the guest OS can update it without (expensive) VMM invocation
 - The hypervisor knows about VMM-local virtual state and can act accordingly
 - E.g., queue virtual interrupt until guest enables it in the virtual register



Paravirtualization

- Allows introducing new optimizations (continued)
 - Example 2: Lazy update of virtual machine state
 - The VM state is maintained by the hypervisor (as usual)
 - Optimization:
 - keep “local” copy of VM state inside paravirtualized guest OS
 - Allow temporary inconsistency between local copy and real VM state
 - Synchronize state on next forced hypervisor invocation (i.e., only when it becomes really necessary to update the real VM state)
 - For instance:
 - If the guest OS kernel performs a series of state updates that only impact the behavior of user-level code (e.g., enable FPU) ...
 - ... Then, these updates can be buffered (by the guest OS) and reflected to the real VM state only upon exit from the guest OS kernel



Paravirtualization revisited: When VMMs and microkernels meet ...

- Reminder: we have previously described paravirtualization in the context of virtual machine monitors (VMMs)
 - Idea: Modify ISA used by guest kernel in order to improve performance
 - Guest kernel is “aware” of the VMM and issues higher-level requests (hypercalls) to it
- The frontier between a (Type-I) VMM supporting paravirtualization and a microkernel is thin
 - In both cases, we have a small kernel providing protection and multiplexing low-level resources
 - However, there are some differences



VMMs vs microkernels

■ Similarities

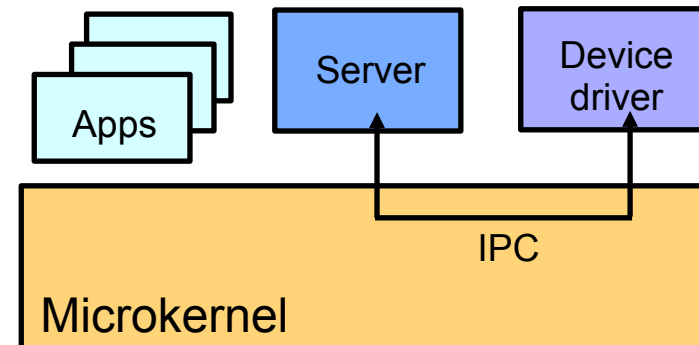
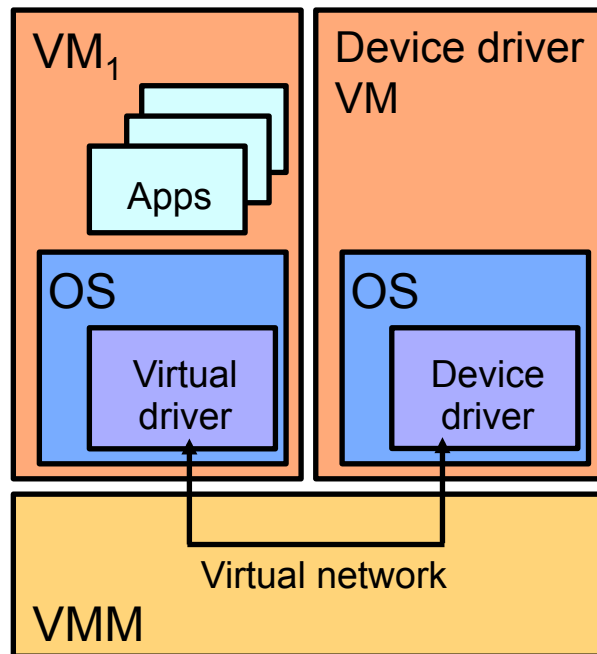
- Both contain all code executing at highest privilege level
- Both need to abstract hardware resources
 - VMM: abstraction closely models hardware
 - Microkernel: abstraction designed to support wide range of systems

■ Optimized for different use cases

- VMMs designed for virtual machines
 - API is hardware-like to ease guest ports
- Microkernels designed for multi-server systems
 - Seem to provide more OS-like abstractions

VMMs vs microkernels

Closer look at I/O and communication



- Communication is critical for I/O
 - Microkernel IPC is highly optimized
 - VMM inter-VM communication is frequently a bottleneck



VMMs vs microkernels

Similarities are growing

- **Hypervisors are becoming more microkernel-like**

- ☐ Tendency to move drivers out of the VMM
- ☐ Rising concerns regarding the security risks of VMM due to their TCB size and attack surface
 - Attempts to make VMMs less monolithic

- **Microkernels are increasingly used to support virtualization**

- ☐ To support legacy OS environments/applications
- ☐ To isolate mission critical applications from other applications

For more details, read: G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence point of Microkernels and Hypervisors. Proceedings of APSys 2010.



VMMs vs microkernels: Summary

VMMs:

- Communication is Achilles heel

- ☐ More important than expected
- ☐ Critical for I/O
- ☐ Plenty of improvement attempts

- Most VMMs have big TCBs:

- ☐ Make it hard/infeasible to achieve high assurance of security/safety
- ☐ In contrast, microkernel implementations can be proved correct

Microkernels:

- Not ideal for virtualization

- ☐ API not very effective
 - L4 virtualization performance close to hypervisor
 - ... but effort much higher
- ☐ Virtualization needed for legacy support



Outline

- Type-I and type-II hypervisors
- Optimizations for memory virtualization
- I/O virtualization
- Dealing with problematic ISAs
 - Paravirtualization
 - **ISA extensions for virtualization support**
- Case studies



ISA extensions for virtualization support

- Goal: extend non-virtualizable legacy ISAs to facilitate/accelerate the operation of VMMs
- Main ideas:
 - Eliminate the need to run all guest code in user (unprivileged) mode
 - Provide hardware assists for common tasks of a VMM
- Several dimensions must be considered:
 - Execution containers (CPU virtualization)
 - Memory virtualization
 - I/O virtualization
- We will detail each dimension (assuming that we have a Type-I VMM)
- Case study: x86 architecture (Intel/AMD)
 - Terminology and some details differ between Intel and AMD versions
 - ... but the general principles are the same

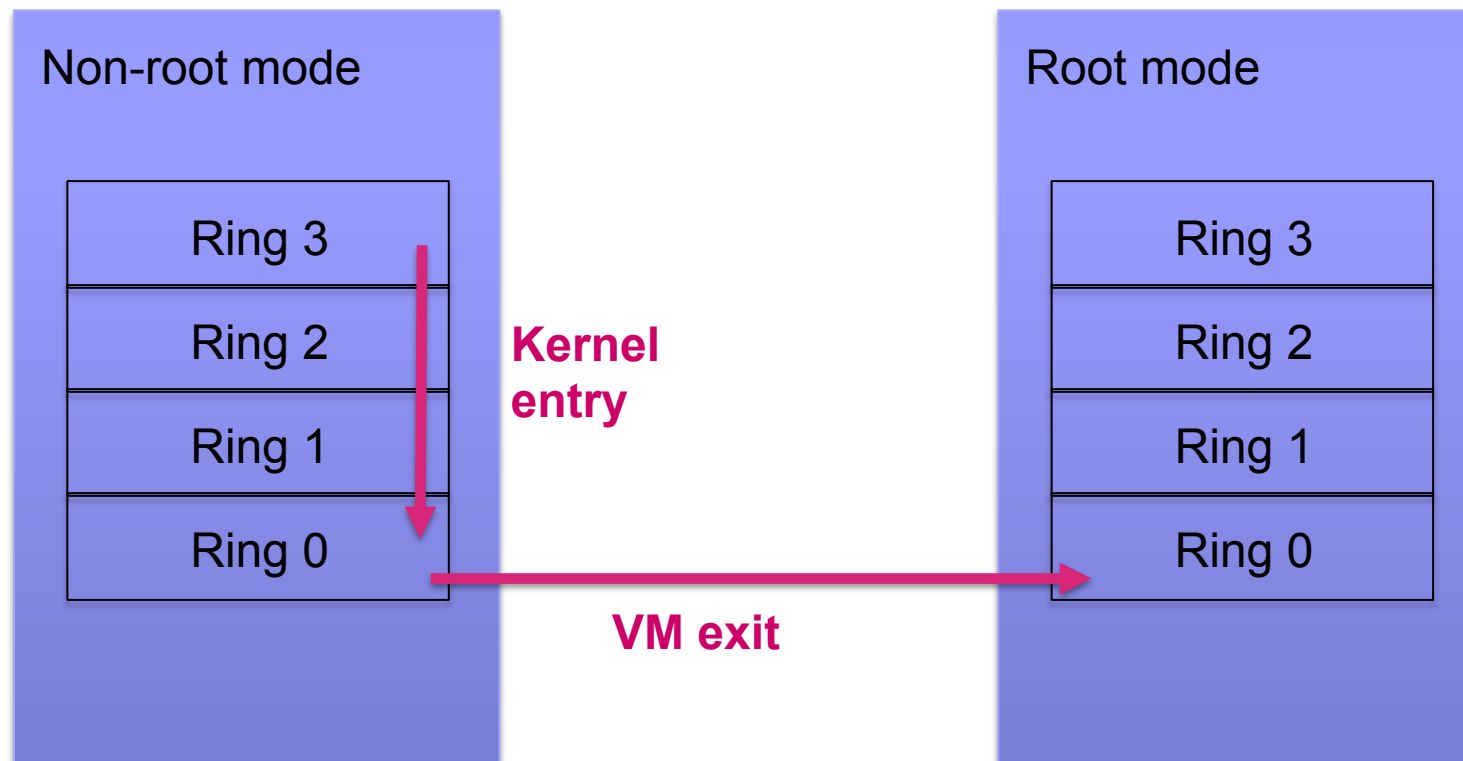


ISA extensions for virtualization support

- Notion of “execution containers”
 - Introduce 2 processors modes:
 - “Root mode” for VMM code
 - “Non-root mode” for the rest of the code
 - These modes are orthogonal to privilege rings (i.e., on Intel processor, each mode has 4 ring levels: 0 to 3)
 - In root mode:
 - The processor behaves likes “traditional” hardware
 - In non-root-mode:
 - All sensitive instructions trap to root mode (“VM exit”)
 - These traps can be very expensive

ISA extensions for virtualization support

- Intel x86 VT-x: non-root mode / root mode





ISA extensions for virtualization support

- Notion of “execution containers” (continued)
 - New instructions to switch between modes (the names below are from Intel)
 - **vmxon/vmxoff**: enable/disable root mode
 - **vmlaunch**: start a new VM and switch to non-root mode
 - **vmresume**: reenter the context of an existing VM
 - **vmcall**: explicit exit from a VM context (e.g. for hypercalls)



ISA extensions for virtualization support

- Optimizations for execution containers
 - The hardware uses extra registers to shadow some privileged state (segment registers, page table pointer, interrupt mask ...)
 - This state is automatically swapped by the hardware on VM entry/exit
 - Goal: reducing management costs of VM architected state and number of traps
 - Hypervisor-configurable register makes some VM exits optional
 - Allows delegating the management of some events to guest VM
 - Examples: selected exceptions such as syscalls, I/O bitmaps, maybe some sources of interrupts
 - Goal: reducing the number of exits



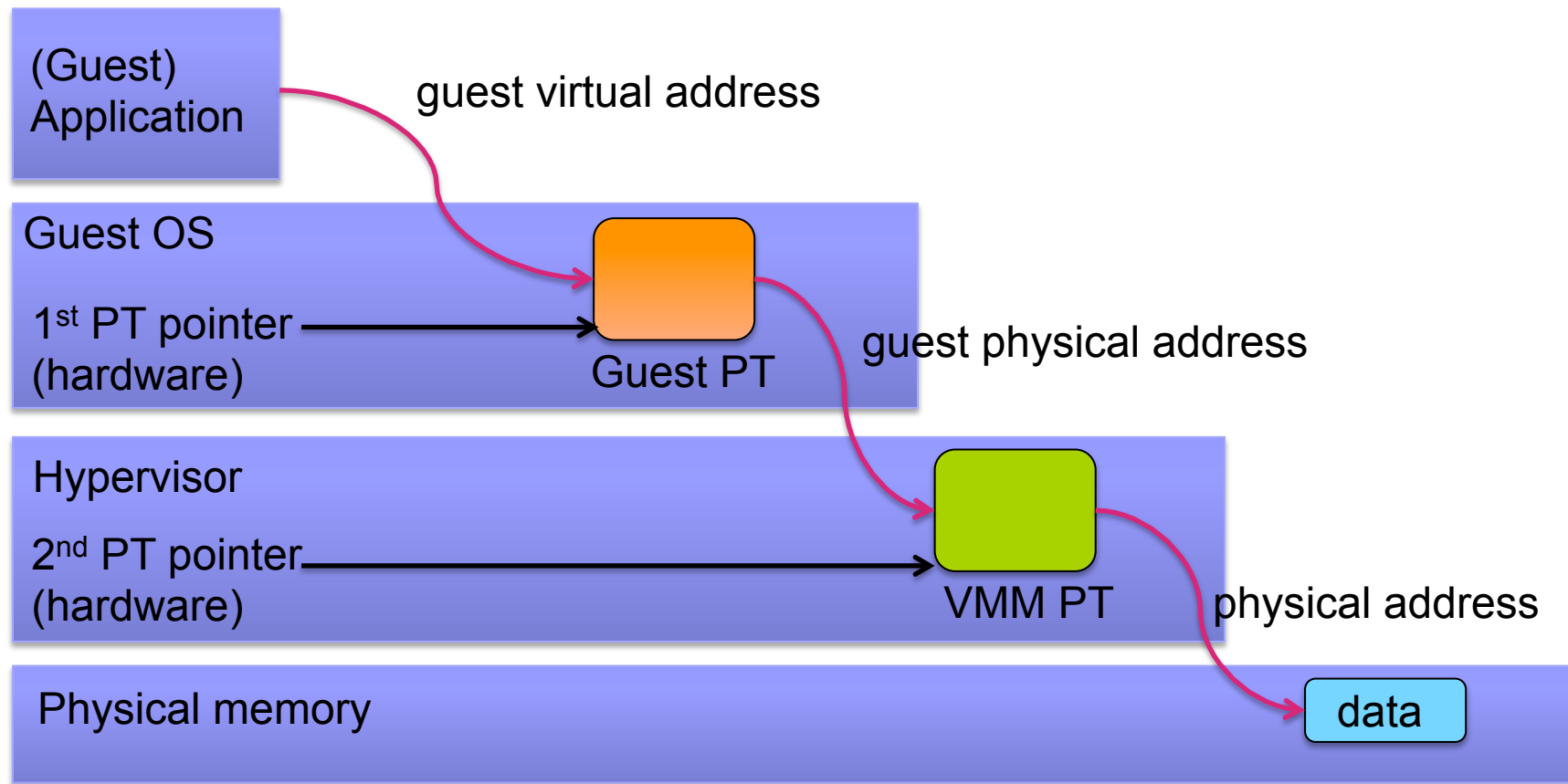
ISA extensions for virtualization support

■ Memory virtualization

- Notion of “extended page tables” (EPT) (a.k.a. “nested page tables”)
- The processor hardware is aware of both the guest’s and the VMM’s current page tables
 - Guest PT maps virtual addresses to guest-physical addresses
 - Host PT (configured by VMM for guest VM) maps guest-physical addresses to physical addresses
- When necessary during guest execution (TLB miss), the hardware can automatically walk the two PTs to determine the mapping between a virtual address and a (host) physical address

ISA extensions for virtualization support

■ Memory virtualization (continued)





ISA extensions for virtualization support

- Memory virtualization (continued)

- **Benefits of EPT**

- No need to trap/exit from guest when it updates a guest PT (introduction of new mappings, modification/invalidation of existing mappings)
 - No need to trap/exit from guest upon page fault due to guest PT configuration (e.g., if guest has swapped out page or write-protected page for copy-on-write). Guest kernel will handle such page faults directly.
 - So, overall less exits to VMM than with shadow page tables
 - Also, simpler design/implementation of the VMM



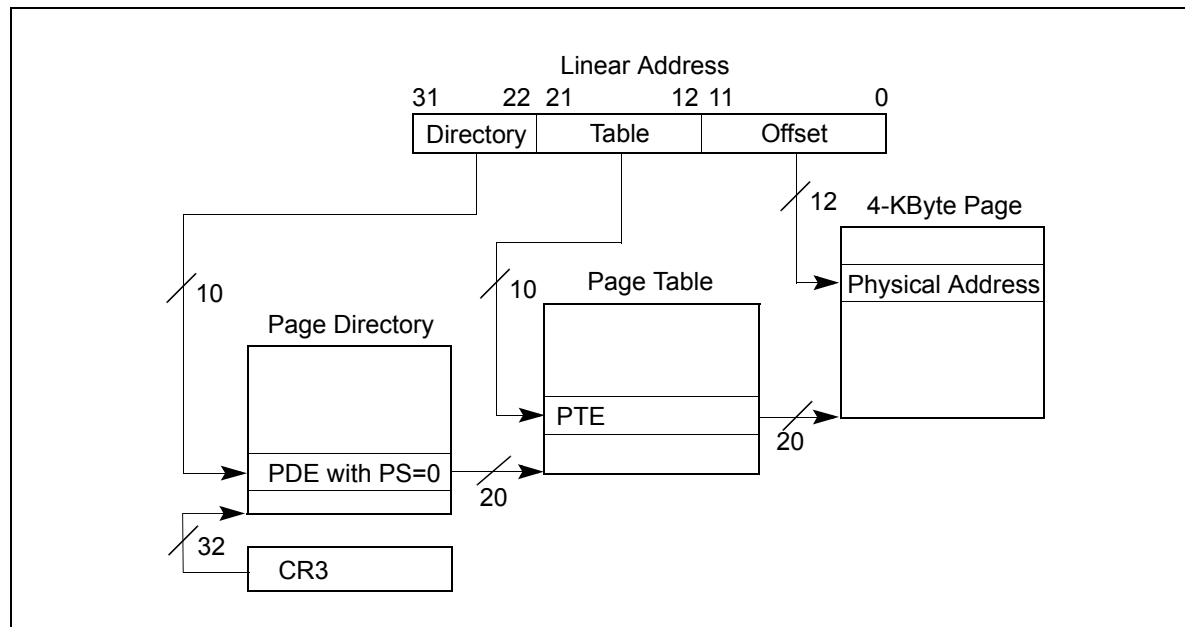
ISA extensions for virtualization support

- Memory virtualization (continued)
 - **Downsides of EPT**
 - TLB misses may be very costly
 - If non-virtualized PT format has N hierarchical levels, then worst-case TLB miss for virtualized guest may have $O(N^2)$ cost
 - Using large pages may help mitigating the costs

Reminder: Page table structure of x86 processors (without hypervisor and EPT)

■ IA-32:

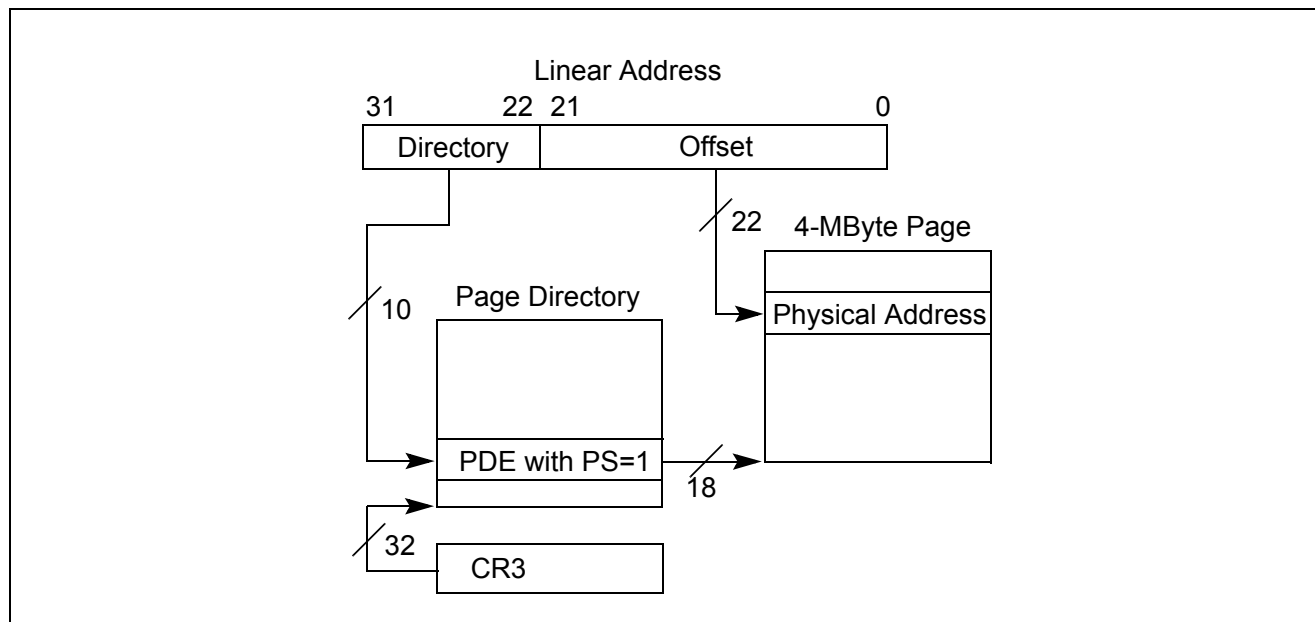
- 32-bit virtual addresses, 32-bit physical addresses
- With page size of 4 kB ($= 2^{12}$ bytes)



Reminder: Page table structure of x86 processors (without hypervisor and EPT)

■ IA-32:

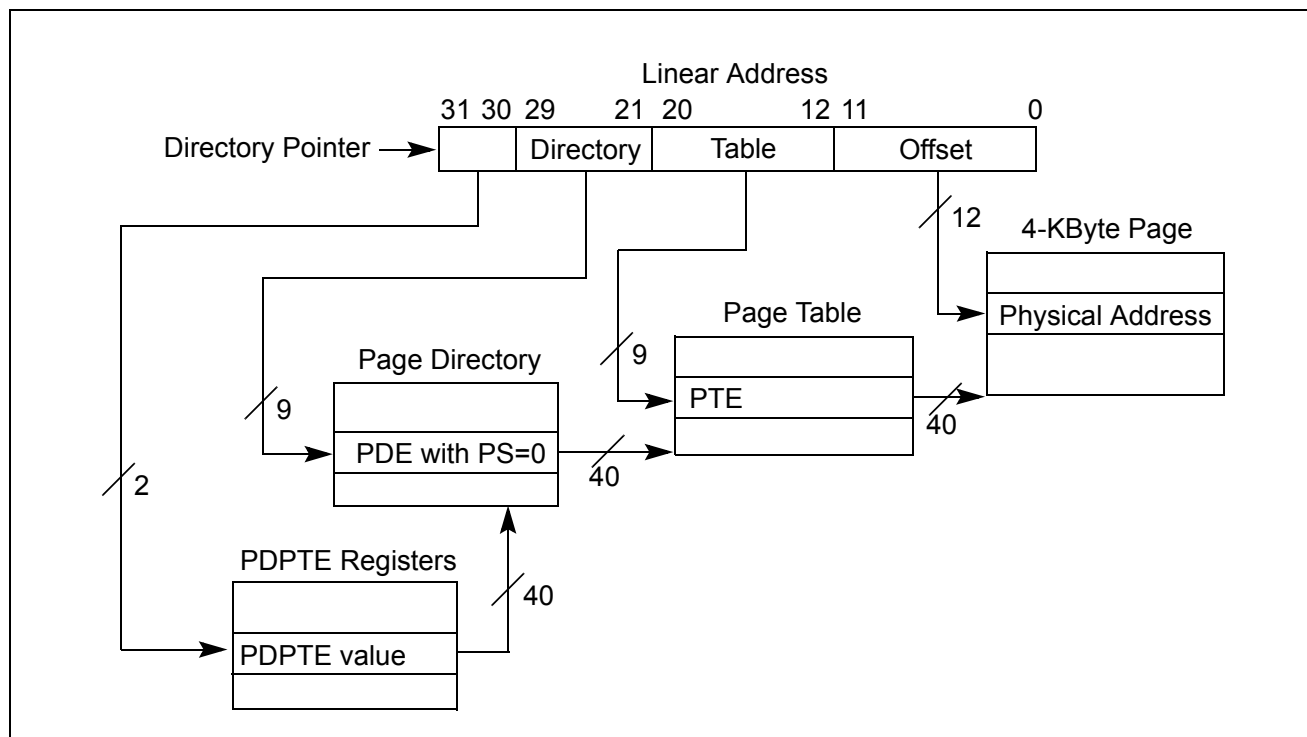
- 32-bit virtual addresses, 32-bit physical addresses
- With page size of 4 MB ($= 2^{22}$ bytes)



Reminder: Page table structure of x86 processors (without hypervisor and EPT)

■ IA-32 PAE:

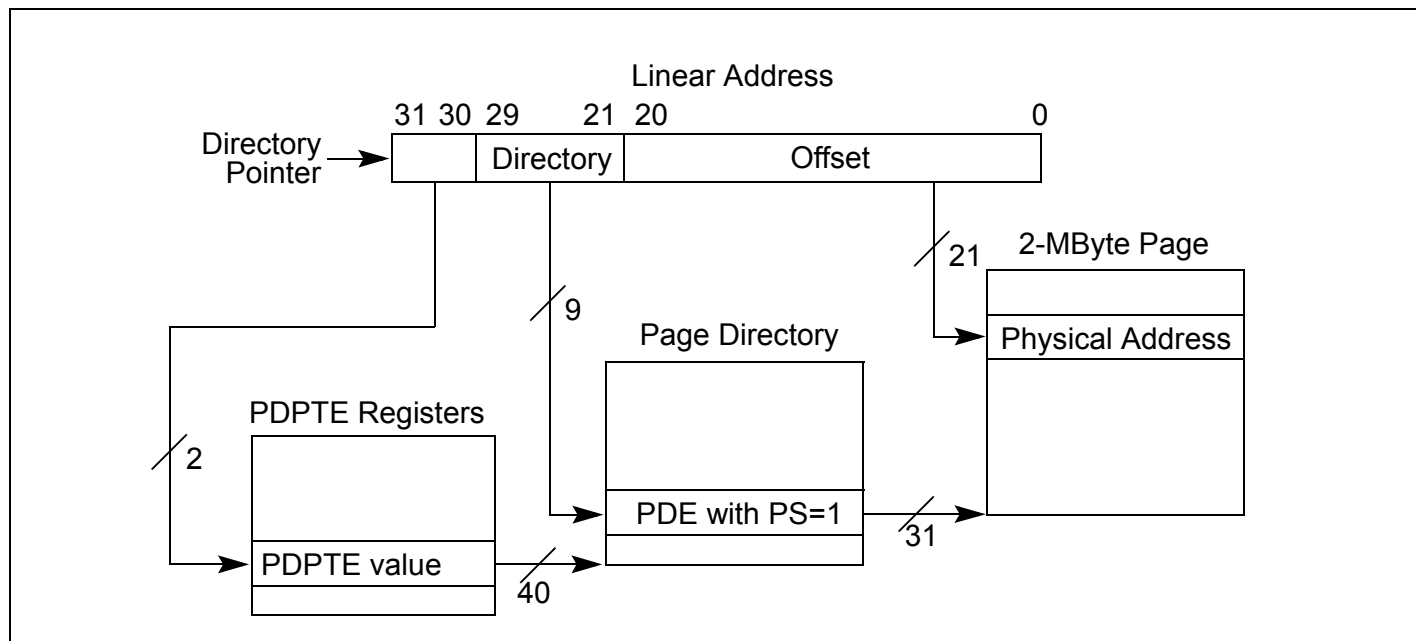
- 32-bit virtual addresses, 36-bit physical addresses
- With page size of 4 kB ($= 2^{12}$ bytes)



Reminder: Page table structure of x86 processors (without hypervisor and EPT)

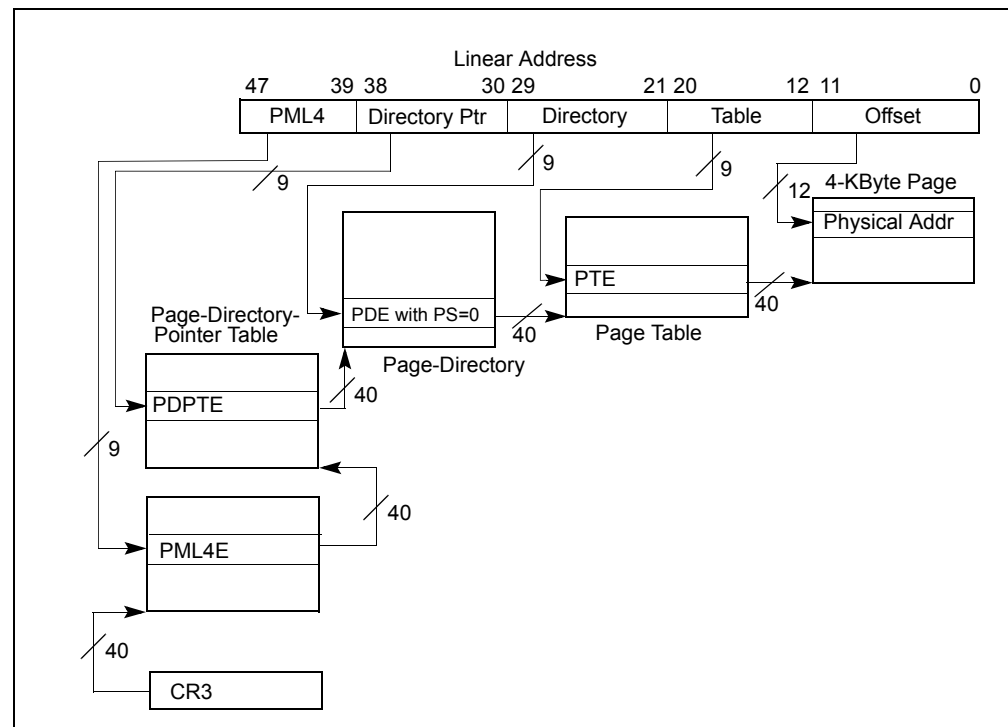
■ IA-32 PAE:

- 32-bit virtual addresses, 36-bit physical addresses
- With page size of 2 MB ($= 2^{21}$ bytes)



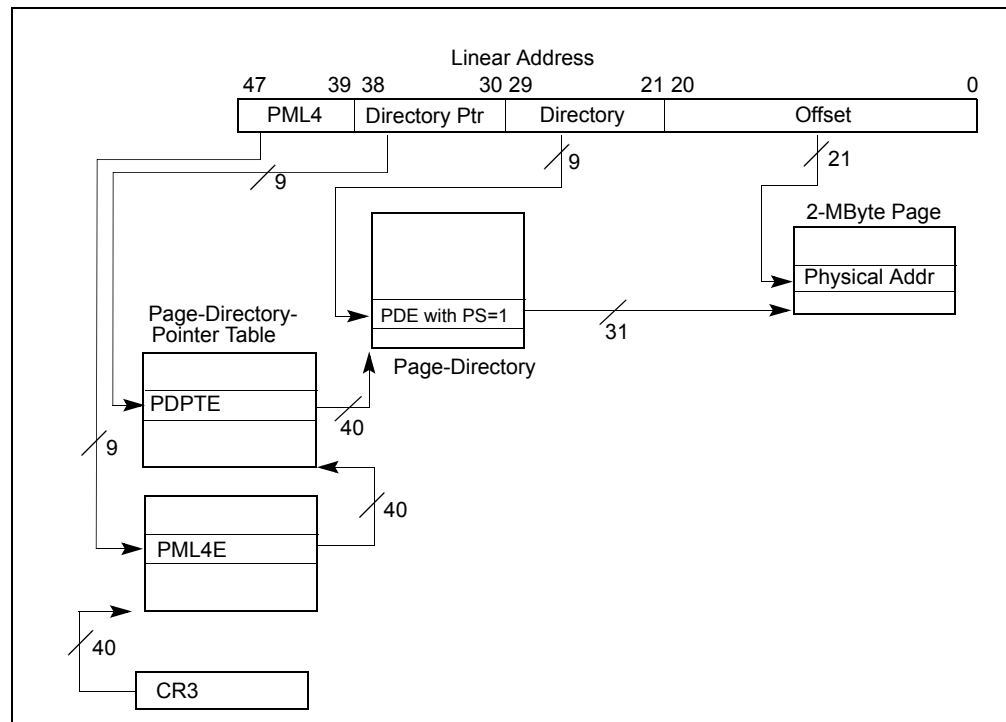
Reminder: Page table structure of x86 processors (without hypervisor and EPT)

- Intel 64:
 - 48-bit virtual addresses, 48-bit physical addresses
 - Actually, 64-bit virtual addresses but the top bits are ignored
 - Design extensible up to 52-bit addresses
 - With page size of 4 kB(= 2^{12} bytes)



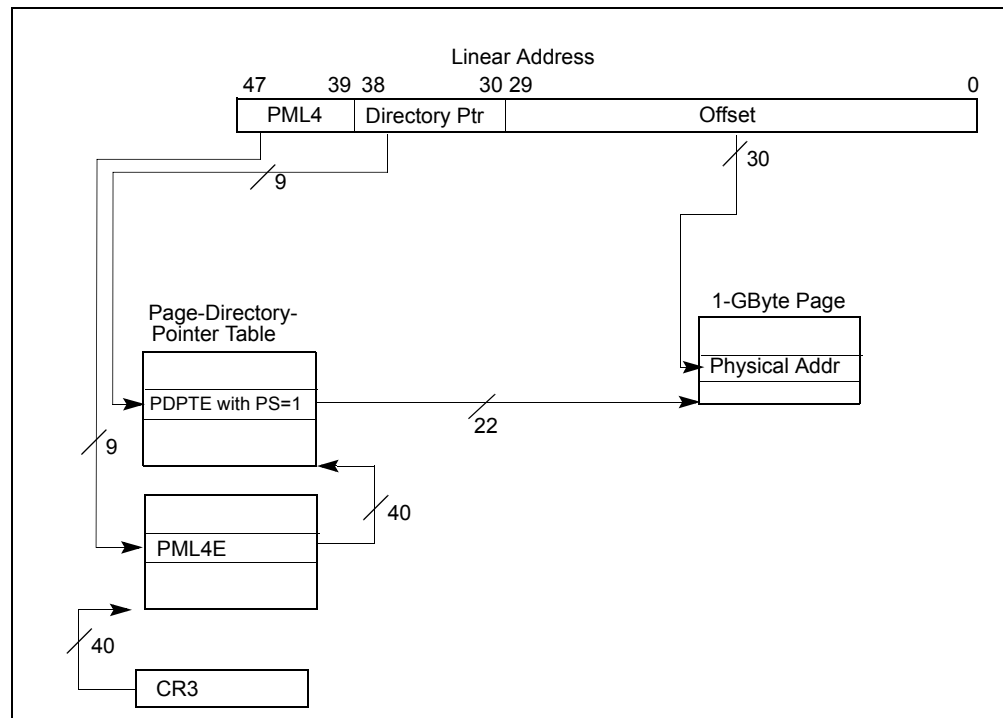
Reminder: Page table structure of x86 processors (without hypervisor and EPT)

- Intel 64:
 - 48-bit virtual addresses, 48-bit physical addresses
 - Actually, 64-bit virtual addresses but the top bits are ignored
 - Design extensible up to 52-bit addresses
 - With page size of 2 MB(= 2^{21} bytes)



Reminder: Page table structure of x86 processors (without hypervisor and EPT)

- Intel 64:
 - 48-bit virtual addresses, 48-bit physical addresses
 - Actually, 64-bit virtual addresses but the top bits are ignored
 - Design extensible up to 52-bit addresses
 - With page size of 1 GB(= 2^{30} bytes)





ISA extensions for virtualization support

- I/O virtualization

- IOMMU – Principles

- A traditional MMU translates virtual memory addresses (seen by instructions executed by the processor) into physical memory addresses

- An IOMMU does the same thing for DMA-capable devices

- Interposed between devices and memory bus
 - Maps device-visible addresses to physical addresses

- Benefits

- Protection against misbehaving devices (or device drivers)
 - Removes device restrictions on non-contiguous memory pages
 - Removes device restrictions on memory address space visibility



ISA extensions for virtualization support

- I/O virtualization (continued)
 - IOMMU – Usage
 - Allows direct, safe and simple management of DMA operations by guest VMs
 - A driver running in a guest OS kernel can be granted direct (and exclusive) access to a device, without compromising the safety of the hypervisor
 - Also useful for safe management of DMA operations by user-level device drivers in microkernels



ISA extensions for virtualization support

- I/O virtualization (continued)
 - Virtualizable devices
 - Example: SR-IOV (Single-Root I/O Virtualization)
 - A physical network interface that implement SR-IOV can appear as multiple devices
 - This simplifies what a hypervisor needs to do when a physical device is multiplexed to several guest virtual machines
 - Allows overcoming restriction of “simple” IOMMU: multiple VMs can have direct access to a shared device



ISA extensions for virtualization support

- I/O virtualization (continued)
 - Direct device assignment to the guest is the most efficient approach, and is facilitated by SR-IOV
 - Remaining problem: interrupt delivery
 - On current x86 hardware, only two possible configurations: all the HW interrupts are either (1) delivered to the guest or (2) to the host
 - Configuration (1) is insecure – generally avoided
 - Configuration (2) is inefficient (exits from the guests hurt the performance for I/O-intensive applications)
 - Solution: Use configuration (1) but make it secure:
 - By forcing the guest to redirect some interrupts to the VMM
 - Using a shadow interrupt descriptor table, stored by the VMM in the guest memory, invisible and non-writable by the guest
 - See the ELI paper [Gordon et al.] for details



ISA extensions for virtualization support

- Note: the main motivations for hardware support are:
 - Simplified support of unmodified guest code
 - Performance gains
- Simplified support is a reality
 - Several VMMs rely on hardware support to run unmodified guest VMs
 - Examples: Xen, KVM (whole Linux kernel as hypervisor)
 - IOMMU can help a lot with securing device drivers
- Performance
 - Can be disappointing compared to binary translation or paravirtualization
 - Essentially because of the number and costs of traps to root mode
 - Has improved with the integration of memory virtualization, and also thanks to less costly exits in recent hardware implementations



Case studies

- Some particular VMM architectures
 - Linux KVM
 - Xen
 - Microhypervisors
- Nested virtualization



Case studies

- Note: We will focus on x86 hardware.
- However, many of the studied systems have also been adapted to the specific (and sometimes quite different) features of other platforms (e.g., ARM)



The (Linux) KVM hypervisor

- “Kernel-based virtual machine”
- Idea: turn standard OS into hypervisor
 - By running the (host) Linux kernel in VT-x root mode
 - (+) Can reuse lots of Linux drivers and facilities (CPU scheduling, monitoring tools, etc.)
 - (-) Huge trusted computing base (attack surface)
 - Note: This approach rather fits in the “type-1” category of hypervisors (but is often debatably assimilated to a “type-2” approach)



The (Linux) KVM hypervisor – additional details (1/2)

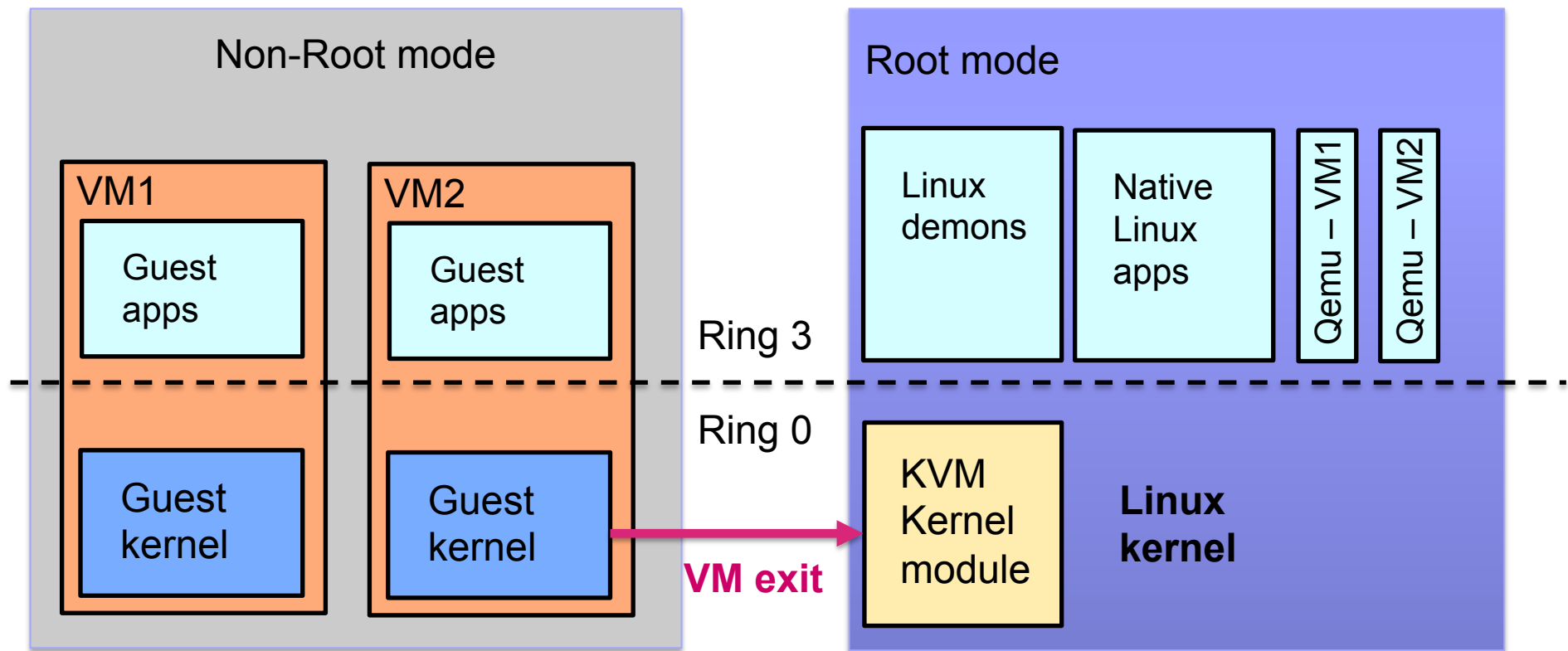
- The KVM infrastructure is made of **several components**:
 - A **loadable (Linux) kernel module** – that enables and interacts with Intel/AMD hardware extensions for virtualization. Runs (like the rest of the host Linux kernel) in “root mode”, ring 0.
 - A **qemu-kvm process** (one process per guest virtual machine)
 - Based on the Qemu emulator
 - Roles:
 - Encapsulates the logical resources required to execute the guest virtual machine: threads, memory
 - Instruction emulation + I/O device emulation
 - Runs in root mode, ring 3 (i.e., like a native Linux application)
 - Interacts with the kernel module (to request privileged operations) via a custom device driver



The (Linux) KVM hypervisor – additional details (2/2)

- A qemu-kvm process uses 1 Linux thread for each virtual CPU (vCPU) of the guest virtual machine
- Upon request from a kvm-qemu thread, the kernel module will trigger a switch to enter “non-root” mode, in order to run the guest
- Upon a VM exit triggered by the VT hardware, the kernel module forwards (if necessary) the event to qemu-kvm
- In addition to the vCPU threads, a qemu-kvm process uses additional threads for other purposes/services: I/O device emulation, remote display, live migration, etc.
- The memory pages of the guest virtual machine are allocated from the memory address space of the qemu-kvm process

The (Linux) KVM hypervisor





The Xen architecture

- The Xen architecture consists in two main components
- **Xen VMM**
 - Most privileged software layer
 - Performs basic virtualization operations and resource management
 - Exports a control interface to authorized entities (typically, only the control VM)
- **Control VM (a.k.a. “Domain 0” VM)**
 - Is a full (Linux-based) operating system with specific applications/demons
 - Trusted by the VMM
 - Launched at boot time by the VMM
 - Implements high-level policies (e.g., admission control for new VMs)
 - Interacts with the VMM to manage the other VMs
 - Create / Destroy / Suspend / Resume / ... a given VM
 - Configure virtual devices, filtering rules, quotas
 - Hosts VM administration services
 - Hosts device emulation + drivers for physical devices
 - Hosts other services (e.g., boot management, live migration)



The Xen architecture

- Note: in Xen terminology, “domain” = “guest VM”
- Design discussion
 - The split design helps separating basic mechanisms from high-level policies
 - Also, simplifies the implementation of high-level VM administration/setup tools (no need to write kernel-level code – can be implemented as “regular” applications)
 - The Control VM is part of the TCB (Trusted Computing Base)
 - ... but does not run with full privileges on the CPU. This allows catching some bugs.



The Xen architecture

- Xen was historically designed before the introduction of hardware-assisted virtualization in x86 processors
- The original version required to modify the source code of guest OSes (paravirtualization)
- Xen was later extended to (optionally) support hardware-assisted virtualization in order to run unmodified guests (and later also modified guests)
- The virtualization spectrum is actually more complex.
 - For details, see http://wiki.xen.org/wiki/Virtualization_Spectrum

The Xen architecture

The virtualization spectrum

(Figure from http://wiki.xen.org/wiki/Virtualization_Spectrum)

The Paravirtualization Spectrum

	Disk / Network	Interrupts, Timers	Emulated Motherboard, Legacy boot	Privileged Instructions and pagetables	
V Virtualized					
P Paravirtualized					
Full Virtualization (FV)	V	V	V	V	HVM mode
FV with PV disk, network	P	V	V	V	
PVHVM	P	P	V	V	
PVH	P	P	P	V	PV mode
Full Paravirtualized (PV)	P	P	P	P	



The Xen architecture

For 32-bit x86 guests:

- A paravirtualized guest kernel is modified to run in ring 1:
 - I.e., the guest kernel effectively runs in ring 1 and knows about it
 - Instead of effectively running in ring 3 and believing that it runs in ring 0
- Benefits
 - No need to use a different page table for the guest application and the guest kernel in order to enforce protection of the guest kernel
 - Allows the guest application to directly invoke the guest kernel without trapping to the VMM
- Notes:
 - Guest applications are unmodified (preserved ABI compatibility)
 - Most privileged instructions that remain in the paravirtualized guest kernel code are explicit hypercalls

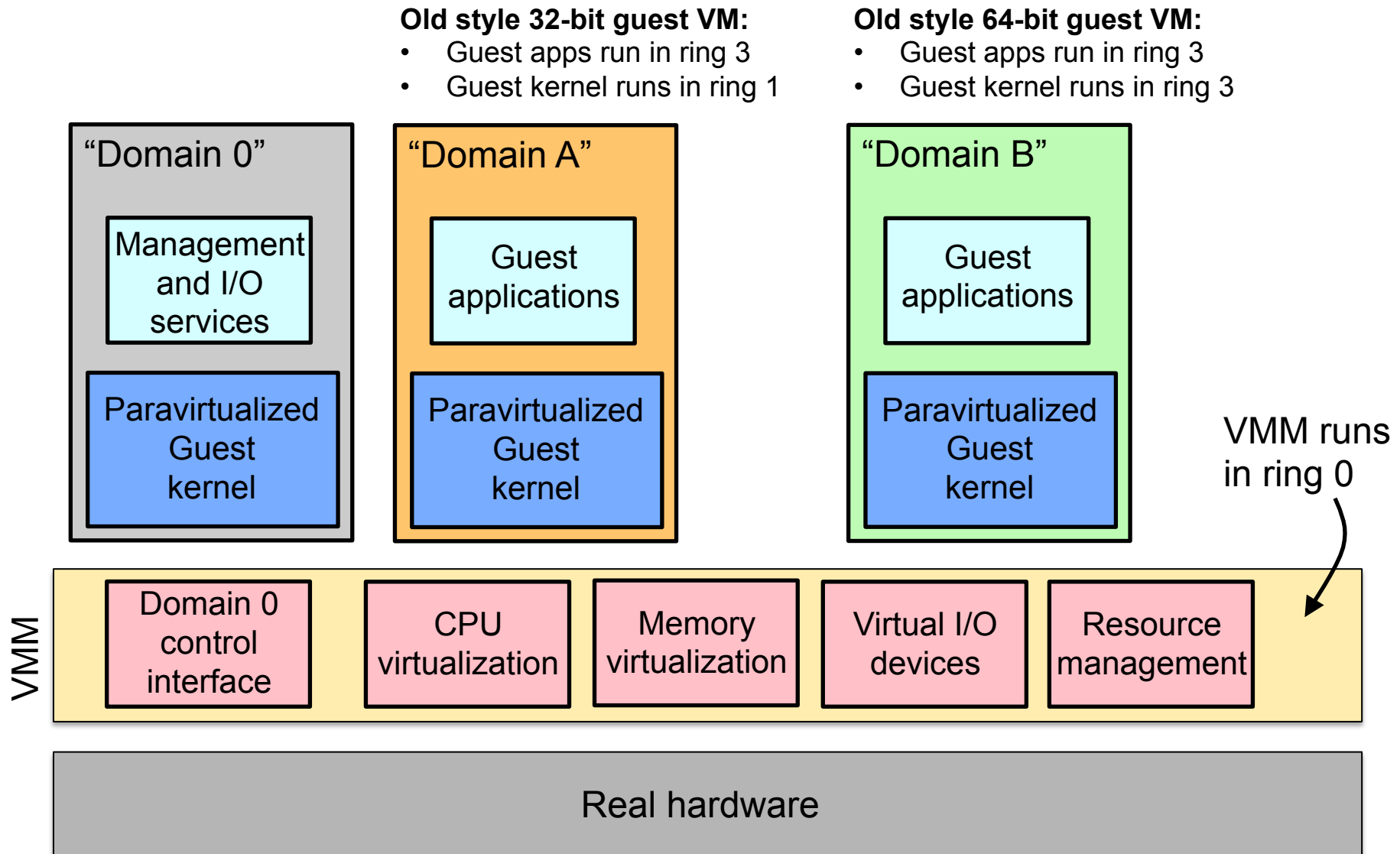


The Xen architecture

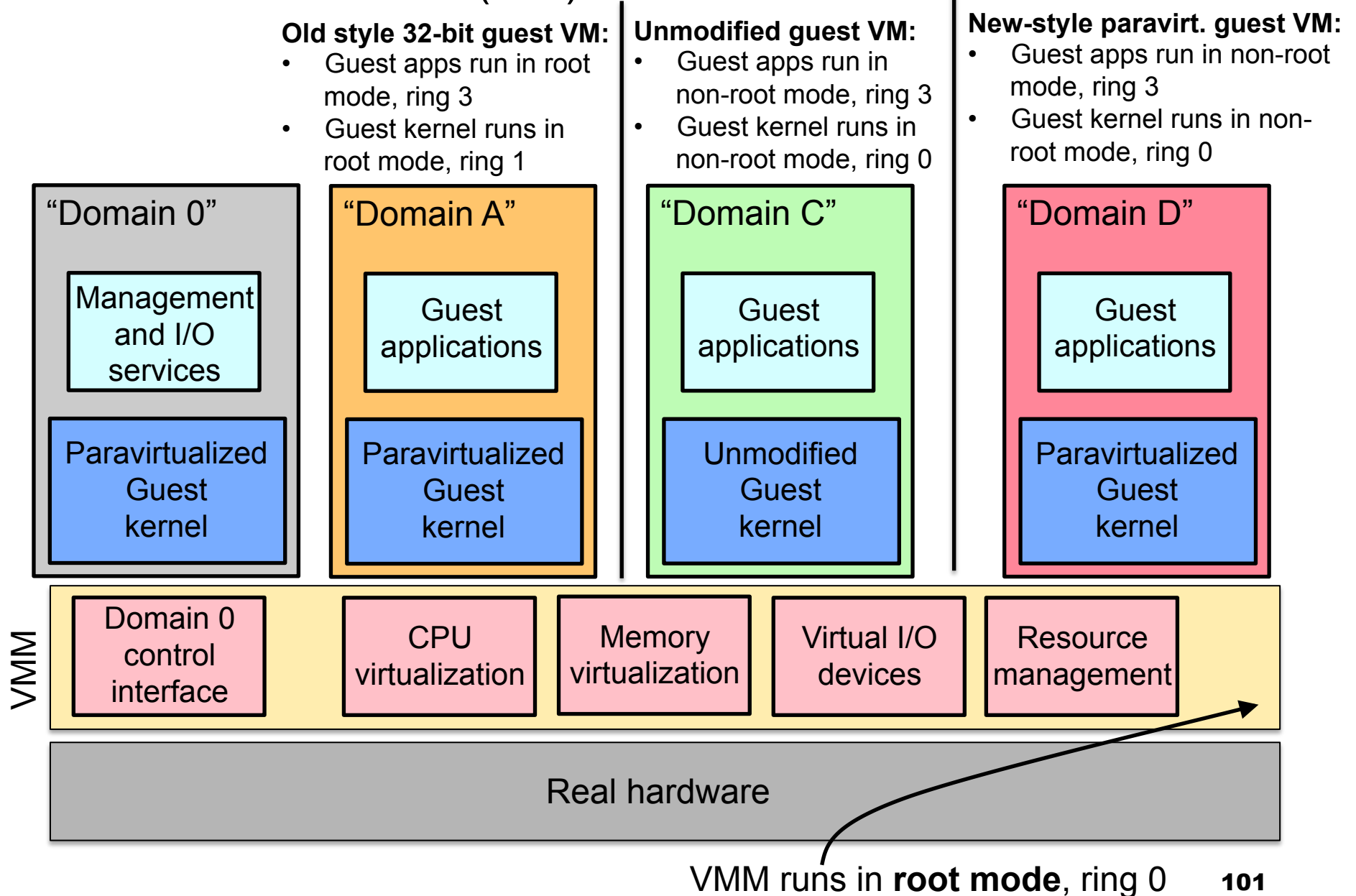
For 64-bit x86 guests:

- The previously described approach cannot be employed because it relied on hardware support for segmentation, which was stripped down in 64-bit versions of the x86 processors
- So, for a paravirtualized 64-bit guest on x86, both the guest kernel and the guest applications have to run in ring 3
- This introduces more traps to the VMM

The Xen architecture (x86) – Historical version (without HW support)



The Xen architecture (x86) – Current version





Microhypervisors

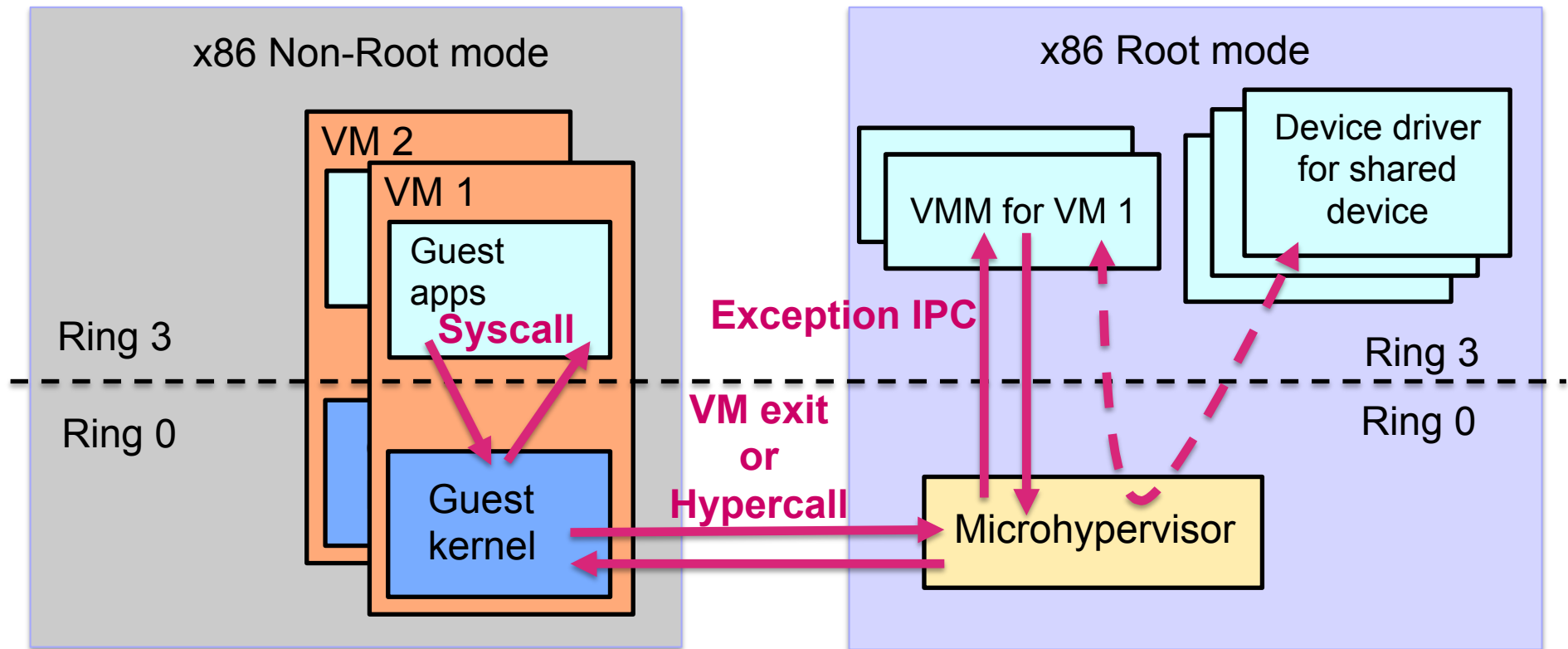
- Context/Motivation
 - Like traditional operating systems, traditional VMMs are monolithic and have a very large code base
 - This makes them likely to have bugs and security holes
- Idea: Use a microkernel-like approach (or even a microkernel) to build a more reliable/secure hypervisor system
- Examples:
 - NOVA microhypervisor [Steinberg and Kauer, EuroSys 2010]
 - seL4 + VMM component follows a similar approach



Microhypervisors

- **Warning: In this context, the terminology differs from the usual one**
 - Usually, “Hypervisor” and “VMM” are interchangeable terms
 - Here, two different notions:
 - **“Hypervisor”**: Privileged component, controls hardware resources
 - **“VMM”**: Unprivileged component, in charge of emulating sensitive instructions + I/O devices for a given guest VM, 1 instance per guest VM

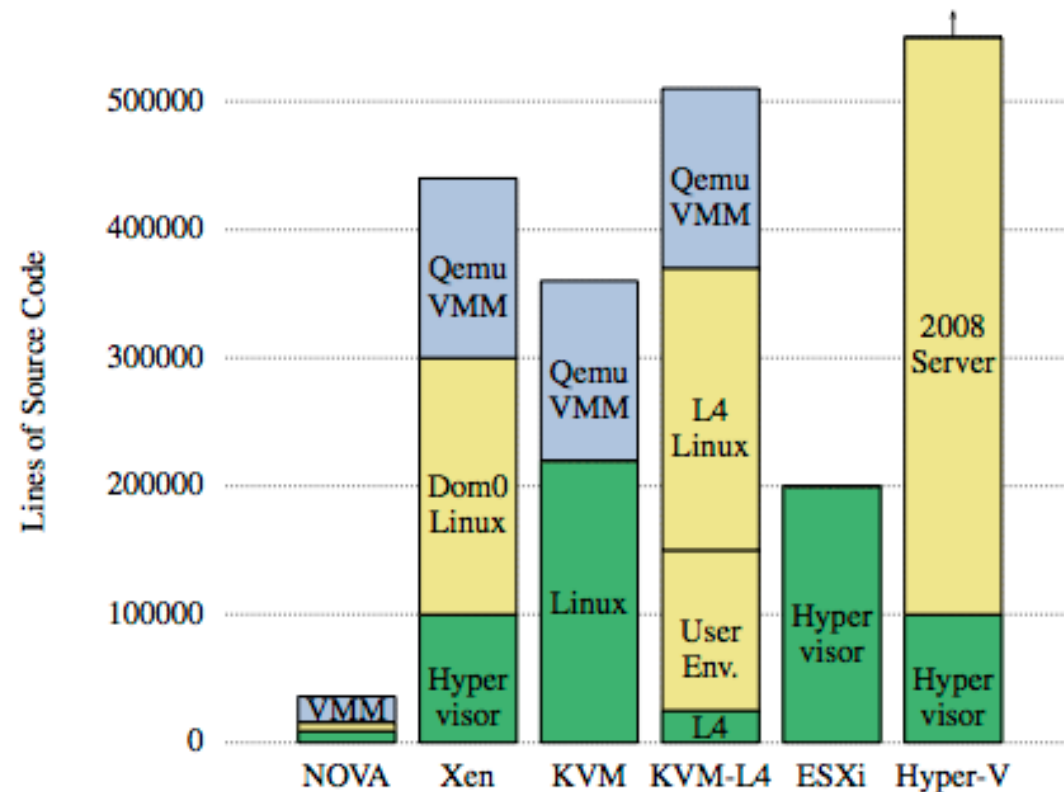
Microhypervisor architecture (x86 platform)



Microhypervisors

- Comparing the sizes of the TCB size of different hypervisors

- “TCB” = Trusted Computing Base
- The part of the system that can circumvent security and must be trusted



See the NOVA paper for details



Nested virtualization

- Running multiple VMMs on top of each other
 - Typically two VMMs:
 - The outer one (a.k.a. “Level 0” or “L0” software layer)
 - The inner one (a.k.a. “Level 1” or “L1” software layer)
 - But the following discussions can be generalized to an arbitrary level of nesting
 - Note: for simplification, we will only focus on Type-I VMMs



Nested virtualization

- Motivating examples:
 - Simplified debugging/testing of hypervisors
 - Operating systems with a built-in VMM
 - Example 1: Windows 7 with WinXP mode
 - Example 2: Linux/KVM
 - (IAAS) Cloud computing: allowing customers to deploy arbitrary VMMs on top of the provider's VMM
 - Portability, features (e.g., live migration) – see “Xen-Blanket” [Williams et al.]
 - Security guarantees – see “CloudVisor” [Zhang et al.]



Nested virtualization

- We assume (most general case):
 - An unmodified guest (not aware of the presence of L1 VMM, nor L0 VMM)
 - An unmodified L1 VMM (not aware of the presence of L0 VMM)

- Possible setups
 - L0 can be based either on hardware-assisted (HW) or on software-only (SW) virtualization
 - Similarly, L1 can be based either on HW or SW virtualization
 - In total, there are 4 possible combinations
 - For the rest of the discussion, **we will focus on the HW/HW setup**

- What are the problems?
 - Functionality
 - The L0 VMM must emulate the hardware virtualization features for the L1 VMM
 - Performance
 - See details on the next slides



Nested virtualization

- Main challenge: performance
 - A naïve implementation will have a very high overhead because the hardware supports a single level of virtualization

- Two main reasons
 - Emulation of hardware support for virtualization
 - Systematic exits to L0



Nested virtualization

- Performance problem #1: emulation of hardware support
 - In a naïve implementation, only the L0 VMM is able to use the hardware virtualization facilities (and only to run L1)
 - Each VMX operation performed by the L1 VMM will:
 - Trigger an exit to L0
 - Require to execute many (software-only) VMX emulation actions at L0 (much slower than true hardware execution of VMX instructions)
 - To execute the code of the (L2) guest (during VMX emulation for L1), the L0 VMM will need to use a software-only virtualization approach - typically, binary translation
 - All the patched sensitive instructions of the guest must necessarily trigger exits to L0
 - This imposes the use of shadow page tables instead of extended page tables for L2
 - This also prevents the use of direct device assignment to L2 (because L0 cannot use the IOMMU for L2)



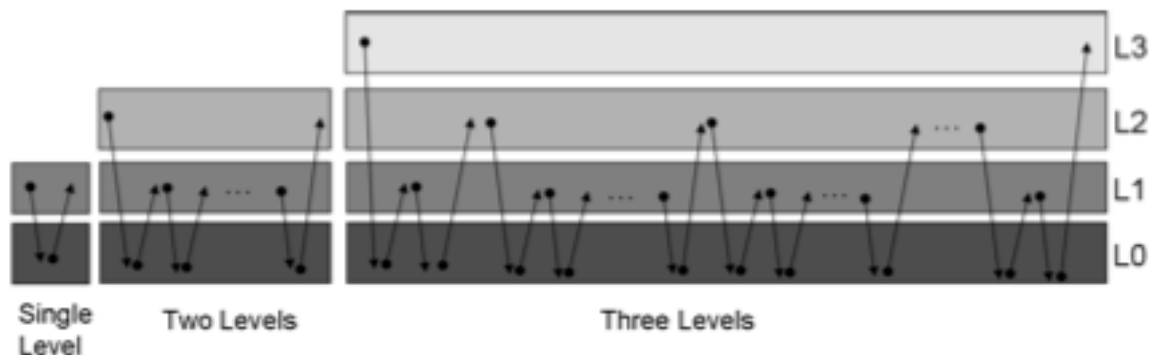
Nested virtualization

- Performance problem #2: systematic exits to L0
 - Regardless of the level in which a trap occurred, execution returns to the L0 (level 0) trap handler
 - Besides, to handle a single exit from L2, the L1 VMM must often perform many privileged operations
 - E.g., read and write the VMCS, update MMU settings, disable/enable interrupts, etc.
 - Those operations must trap to L0
 - This phenomenon is known as “exit multiplication”

Nested virtualization

■ Exit multiplication – Illustration

- Setup 1, without nesting: guest OS (L1) running above a single VMM (L0)
- Setup 2, with nesting: same guest VM (L2) running above inner VMM (L1), running above outer VMM (L0)
- Let us consider a single exit from the guest to the VMM in setup 1
- In setup 2, the same scenario (single exit from guest to inner VMM) may trigger 40-50 exits from inner VMM (L1) to outer VMM (L0)
- Things get worse with additional levels of nesting





Nested virtualization

- Possible approaches for efficient nested virtualization:
 - Architectural support for multiple levels of virtualization
 - Each VMM directly handles all the traps caused by sensitive instructions of any guest VMM running directly on top of it
 - Modifying the complete software stack (including OSes/VMMs) to introduce optimizations
- But, unfortunately:
 - Current x86 processors only have architectural support for a single level of virtualization
 - In many cases, it is not possible to modify the code of a legacy OS/VMM

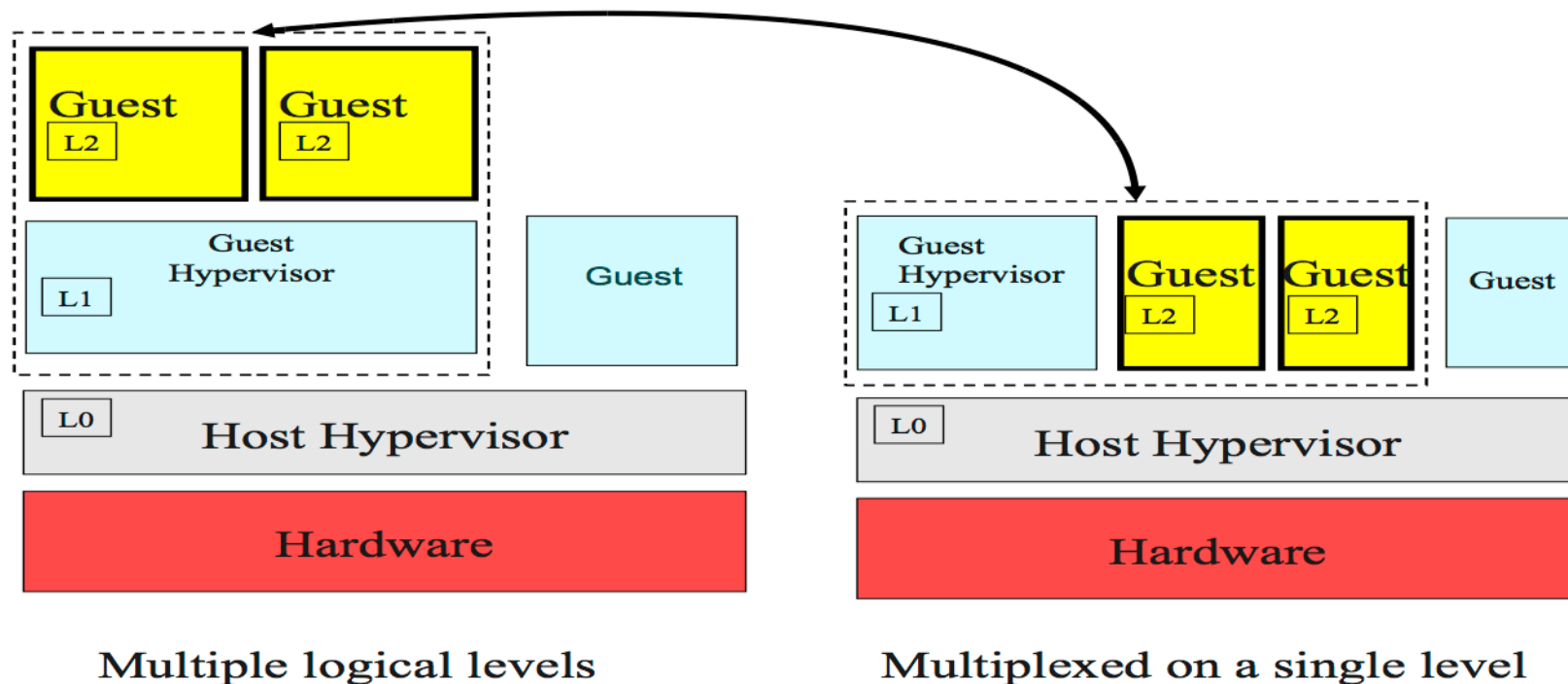


Nested virtualization

- **How can we nonetheless achieve efficient nested virtualization under the above assumptions?**
- By combining several techniques to:
 - Transparently merge/collapse virtualization levels so that many virtualization operations performed by L1 for L2 can be performed in hardware instead of being completely emulated in software by L0
 - Reduce the number/frequency of exits to L0
 - Reduce the overhead of each exit
- Read the “Turtles” paper for details (focused on x86 processors) [Ben-Yehuda et al. OSDI 2010]

Nested virtualization

- Transparently merging/collapsing virtualization levels





Nested virtualization

- Transparently merging/collapsing virtualization levels
 - Benefits:
 - Makes it possible for L1 virtualization (for L2 guest) to be hardware accelerated
 - Also, as a side effect, reduces the number of exits to L0 (especially for memory and I/O virtualization)
 - L0 VMM multiplexes the hardware between L1 and L2 guests layers:
 - L2 guest is actually executed as a L1 guest (from the point of view of the hardware)
 - Thus, the L2 guest VM runs at the same level as its underlying L1 VMM
 - Yet, the L1 VMM remains in full control of the guest VM
 - Neither the guest OS nor the L1 VMM are aware of this trick (only the L0 VMM needs to know about it)



Nested virtualization

- Details on nested CPU virtualization
 - **Reminder: we assume that L0 and L1 are both configured to use hardware VMCS features for virtualizing the CPU**
 - “VMX” instructions (i.e., instructions related to VM management) can only execute successfully in root mode.
 - In a nested setup, the L0 VMM must emulate such instructions for the L1 VMM (which believes that it runs in root mode)

 - L0 manages a VMCS (named $VMCS_{0,1}$) to run L1
 - L1 manages a VMCS (named $VMCS_{1,2}$) to run L2
 - All the VMX instructions executed by L1 will trigger exits to L0. This makes L0 aware of the creation/modification of $VMCS_{1,2}$
 - To collapse the virtualization levels, L0 must actually build another VMCS ($VMCS_{0,2}$) to run L2, by “merging” the contents of $VMCS_{0,1}$ and $VMCS_{1,2}$
 - Exits from L2 will always reach L0
 - L0 will selectively forward those exits to L1 (depending on the cause)



Nested virtualization

■ Details on nested memory virtualization

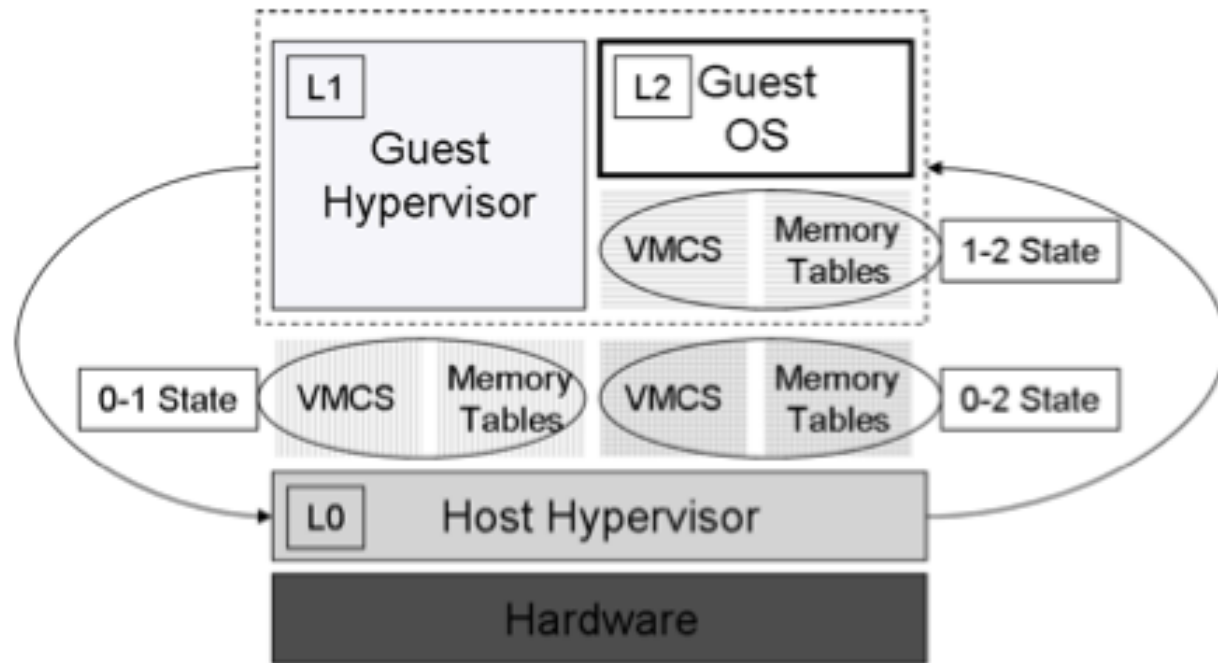
- **Reminder: We assume that L0 and L1 are both configured to use hardware nested paging**
- L0 manages an extended page table (named $EPT_{0,1}$) to run L1
- L1 manages an extended page table (named $EPT_{1,2}$) to run L2
- All the EPT manipulations executed by L1 will trigger exits to L0. This makes L0 aware of the creation/modification of $EPT_{1,2}$

- To collapse the virtualization levels, L0 must actually build and maintain another EPT ($EPT_{0,2}$) to run L2, by “merging” the contents of $EPT_{0,1}$ and $EPT_{1,2}$
- $EPT_{0,2}$ directly maps L2 physical addresses to L0 physical addresses

- Using hardware EPT support for the L2 guest (compared to shadow page tables) can significantly improve performance by reducing the number of exits
 - In this setup, page faults triggered by the guest applications do not cause exits (only an access to a guest physical page that is not mapped in the EPT table will cause an exit).

Nested virtualization

- Summary of nested CPU and memory virtualization





Nested virtualization

- Details on nested I/O virtualization
 - There are 3 possible approaches to I/O virtualization between L0 and L1:
 - Emulation of the I/O device
 - Paravirtualized I/O device
 - Direct device assignment of the I/O device to the guest (based on IOMMU)
 - Similarly, the same set of approaches is available between L1 and L2
 - At the scale of the nested setup, there are thus 9 possible configurations for I/O virtualization. Some configurations are much more efficient than others.
 - Here, **we focus on the “direct-direct” configuration.**
 - This is based on the assumption that the physical device can be dedicated to a single guest (or that this device supports SR-IOV to virtualize itself).



Nested virtualization

- Details on nested I/O virtualization (continued)
 - How to efficiently implement the “direct-direct” configuration?
 - L0 emulates the IOMMU for L1
 - L1 configures the IOMMU with its chosen L2-to-L1 mappings: the IOMMU seen by L1 maps L2 physical address to L1 physical address
 - L0 configures the real IOMMU with L2-to-L0 mappings: the real IOMMU maps L2 physical addresses to L0 physical addresses



Nested virtualization

- Additional optimizations

- There are two main places where a guest of a nested hypervisor is slower than the same guest running on a bare-metal hypervisor:
 - Transitions between L1 and L2 are slower than between L0 and L1
 - The exit-handling code running in L1 is slower than the same code running in L0
- We will discuss each problem in turn
- Reminder: we still assume that L2 and L1 are unmodified. So, we focus on optimizations at the level of L0



Nested virtualization

- Optimizing transitions between L1 and L2
 - Each such transition requires to go through L0
 - Most of the time in L0 is spent merging the VMCSes
 - Time can be saved if L0 only copies data when the VMCS contents have changed



Nested virtualization

■ Optimizing exit handling in L1


- Many instructions in the exit-handling code executed by L1 will trigger exits to L0
- In particular, **Intel processors use specific vmread/vmwrite instructions to access the contents of a VMCS. These instructions are privileged and trigger exits to L0.**
- Possible optimization: use binary translation within the L0 VMM to replace the L1 vmread/vmwrite instructions with load/store instructions
- Other possibilities (more demanding):
 - L1 modifications: Introduce paravirtualization in L1 to obtain a more efficient pseudo-VMX interface between L1 and L0 (less exits)
 - Hardware modifications: Replace vmread/vmwrite with standard load/store instructions (see AMD processors)
 - Hardware modifications: Keep vmread/vmwrite instructions but introduce hardware support for shadow VMCS that can be manipulated by L1 without exits (see “VMCS shadowing” feature in recent versions of Intel processors)



References

- General notions

- A. Tanenbaum and H. Bos. Modern Operating Systems 4th edition. Prentice Hall, 2014. Chapter 7: Virtualization and the Cloud.
- (Older) J. Smith and R. Nair. Virtual machines: versatile platforms for systems and processes. Morgan Kaufmann 2005.




References (continued)

■ Architectural support for x86 virtualization

- Gil Neiger et al. Intel virtualization technology: hardware support for efficient processor virtualization. Intel technology Journal 10(3), August 2006.
- Darren Abramson et al. Intel virtualization technology for directed I/O. Intel technology Journal 10(3), August 2006.
- Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS), March 2008. Seattle, WA
- Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In Proceedings of the 2012 USENIX conference on Annual Technical Conference.

■ Comparison of binary translation vs. hardware-assisted virtualization on x86

- Keith Adams and Ole Agesen. A comparison of the software and hardware techniques for x86 virtualization. In Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 2006. **Warning: the studied x86 hardware at that time did not feature support for memory virtualization**
- VMware. Software and hardware techniques for virtualization. 2009.
http://www.vmware.com/files/pdf/software_hardware_tech_x86_virt.pdf
- O. Agesen, A. Garthwaite, J. Sheldon, P. Subrahmanyam. The evolution of an x86 virtual machine monitor. ACM SIGOPS Operating System Review. Volume 44 Issue 4, December 2010.




References (continued)

- Memory optimizations in a type-1 VMM

- Carl A. Waldspurger. Memory resource management in VMware ESX server. In Proceedings of the 5th Symposium on Operating Systems Design and Implementations, Boston, MA, USA, 2002.
- Nadav Amit, Dan Tsafir, Assaf Schuster. VSwapper: A memory swapper for virtualized environments. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March, 2014, Salt Lake City, USA.

- I/O virtualization

- Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001.
- Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, Dan Tsafir. ELI: Bare-Metal Performance for I/O Virtualization Proceedings of ASPLOS '12: Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems. London, UK, 2012.
- Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, Razya Ladelsky. Efficient and Scalable Paravirtual I/O Systems. Proceedings of the 2013 USENIX Annual Technical Conference.




References (continued)

■ Paravirtualization

- Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002), Boston, MA, December 2002.
- Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Proc. 19th Symp. on Operating Systems Principles, pages 164–177, Bolton Landing, NY, USA, October 2003
- Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie and Gernot Heiser. Pre-virtualization: Soft layering for virtual machines. Asia-Pacific Computer Systems Architecture Conference, pp. 1-9, Hsinchu, Taiwan, August, 2008

■ Paravirtualization and microkernels


- Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ -kernel-based systems. In Proc. Symp. on Operating Systems Principles, pages 66–77, St. Malo, France, October 1997. **129**



References (continued)

- Modular VMM architectures


- Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems (EuroSys '10)*. 2010.
- Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. 2011.



References (continued)

■ Nested virtualization


- With architectural support (IBM z/VM): D. L. Osisek et al. Esa/390 Interpretive-execution architecture, foundation for vm/esa. IBM Systems Journal 30, 1 (1991).
- Without architectural support (x86): Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI), Vancouver, BC, Canada, October 2010.
- Use cases:
 - Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In Symp. on Operating Systems Principles, pages 203–216, Cascais, Portugal, 2011.
 - Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In Proceedings of ACM EuroSys, Bern, Switzerland, April 2012.



References (continued)

■ Misc. Topics / VMware

- An overview of VMware's VMM and its evolution over the years (with a particular focus on binary translation but also some more general details on system VMs principles and hardware support):
 - O. Agesen, A. Garthwaite, J. Sheldon, P. Subrahmanyam. The evolution of an x86 virtual machine monitor. ACM SIGOPS Operating System Review. Volume 44 Issue 4, December 2010, pages 3-18.
- A detailed description of the first version of the first VMware VMM product (Vmware workstation – Type II hypervisor)
 - E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, E. Wang. Bringing virtualization to the x86 architecture with the original Vmware Workstation. ACM Transactions on Computer Systems. Vol. 30. No. 4. November 2012.



References (continued)

- Hardware support for virtualization on ARM processors (and corresponding VMM design)
 - Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on ARM. Asia-Pacific Workshop on Systems (APSys), Shanghai, China, July, 2011.
 - Christoffer Dall and Jason Nieh. KVM/ARM: The design and implementation of the Linux ARM hypervisor. In International Conference on Architectural Support for Programming Languages and Operating Systems, pages 333–347, Salt Lake City, UT, USA, March 2014.