Hardware-assisted virtualization: x86
Lecture for the Embedded Systems Course
CSD, University of Crete (May 5 & 7, 2025)

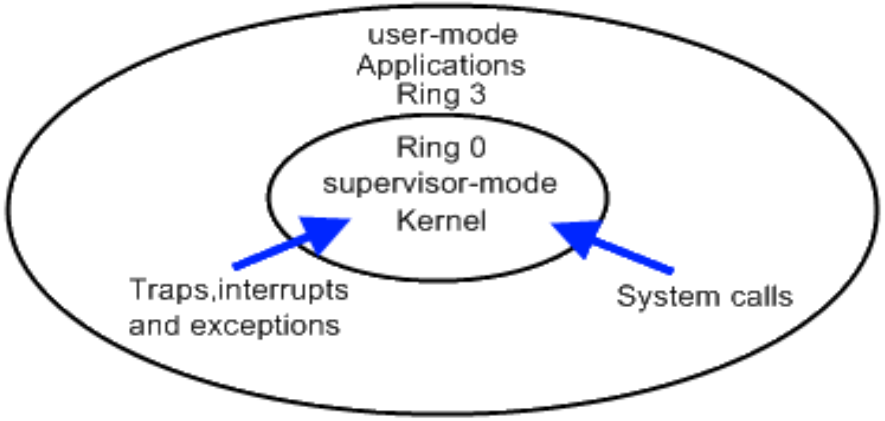
- Manolis Marazakis (maraz@ics.forth.gr)

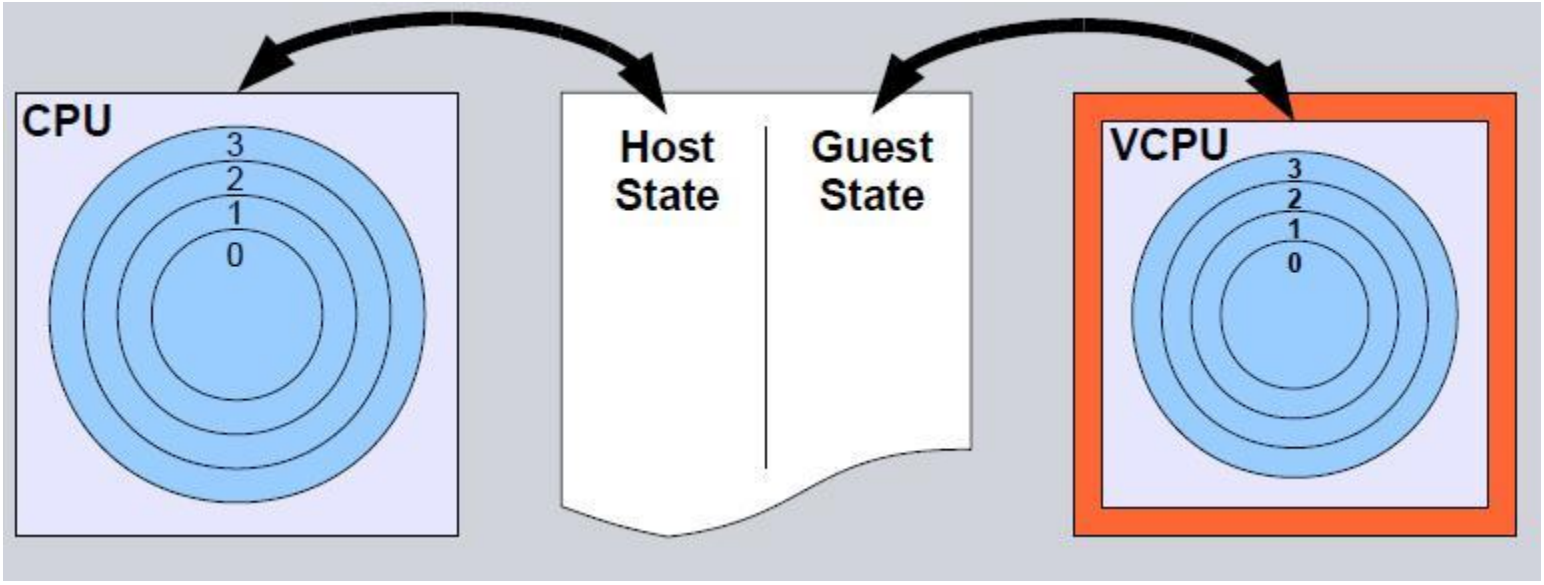
FORTH-ICS Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)

Virtualization of the x86 instruction set

- Trap-and-Emulate
- Alternatives:
  - Binary translation (e.g. early VMware), with “code substitution”
  - CPU para-virtualization (e.g. Xen), with “hyper-calls”
  - Hardware-assisted virtualization
    - Objective: reduce need for binary translation and emulation
    - Host + Guest state, per core
    - “World switch”
    - Intel VT-x, AMD SVM
    - Intel EPT (extended page tables), AMD NPT (nested page tables)
      - Add a level of translation for guest physical memory
      - R/W/X bits → can generate faults for physical memory accesses
    - vCPU-tagged address spaces → avoid TLB flushes

Virtualized CPU

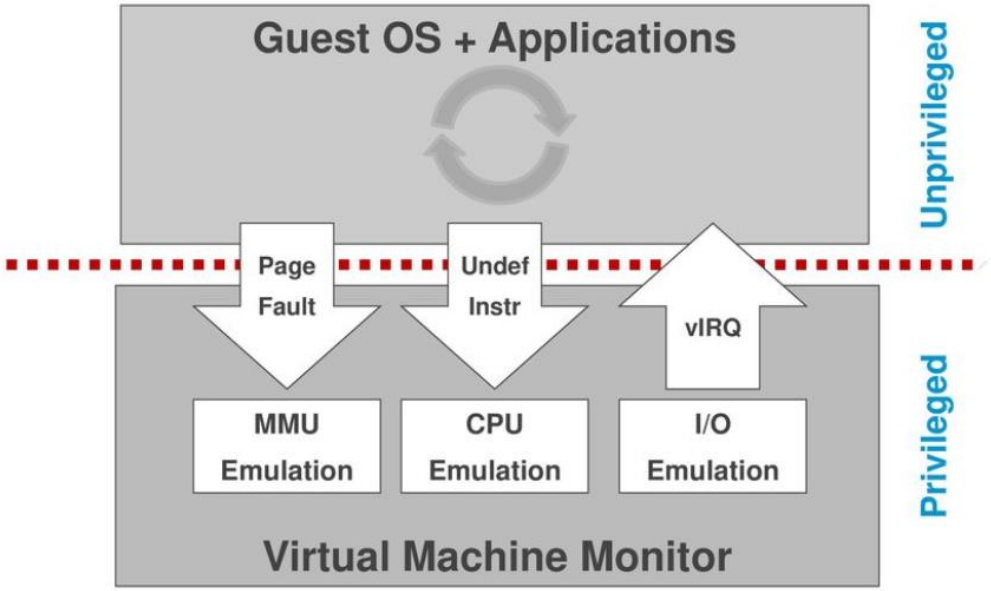


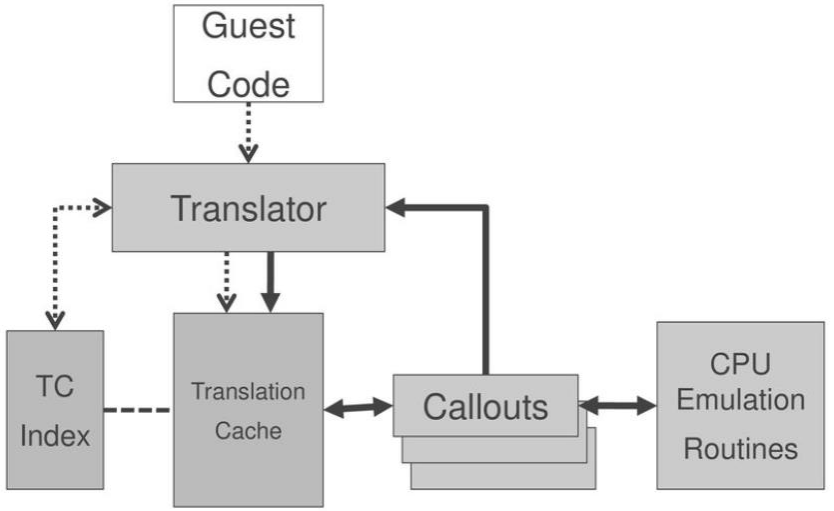


Trap-and-Emulate

- VMM and Guest OS :
  - System Call (ring 3)
    - CPU will trap to interrupt handler vector of VMM (ring 0).
    - VMM jump back into guest OS (ring 1).
  - Hardware Interrupt
    - Hardware makes CPU trap to interrupt handler of VMM.
    - VMM jump to corresponding interrupt handler of guest OS.
  - Privileged Instruction
    - Running privilege instructions in guest OS will trap to VMM for instruction emulation.
    - After emulation, VMM jumps back to guest OS.

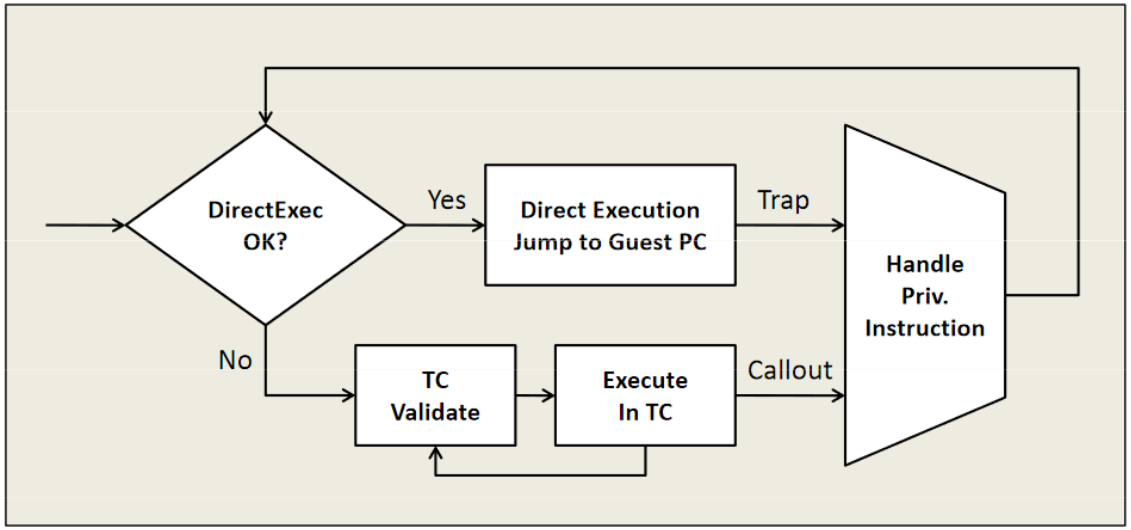
Trap&Emulate + Binary Translation



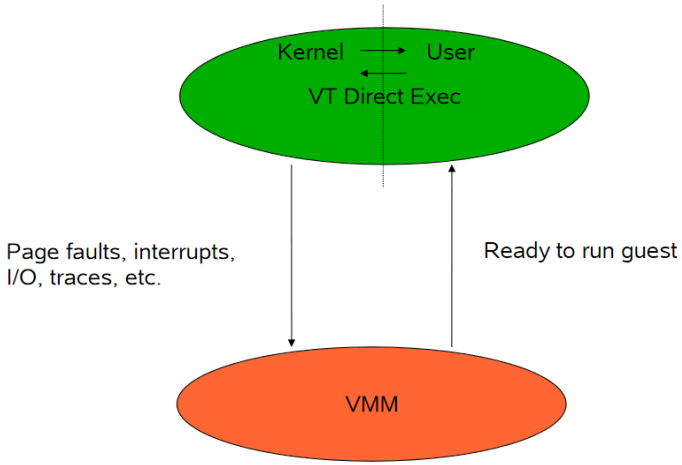


[source: Scott Devine (VMware)]

Hybrid binary translation approach



[VMware, US Patent 6,397, 242]

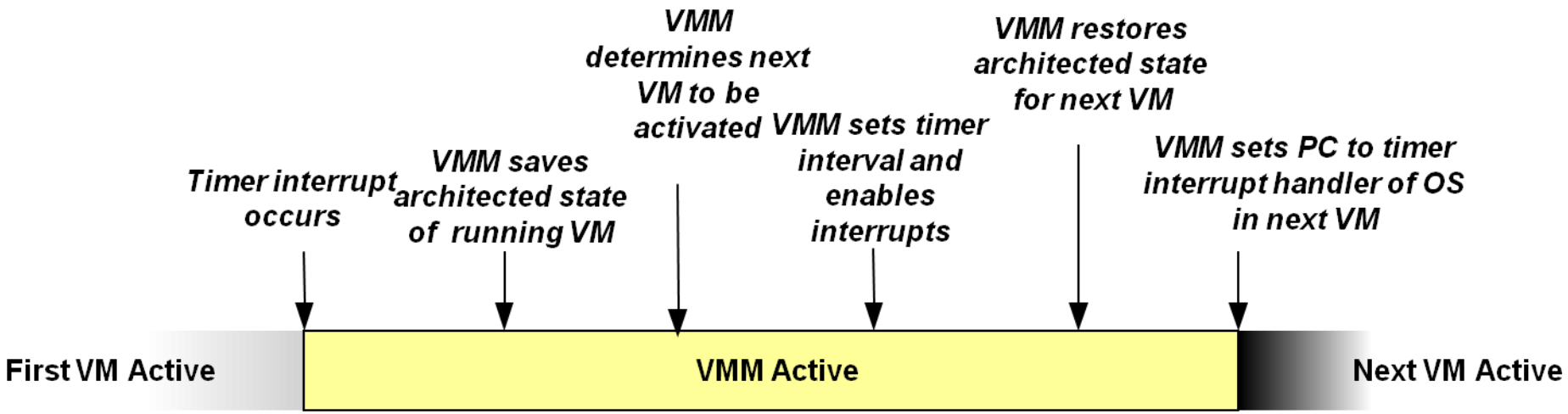


- Binary translation for “sensitive” code
- Direct execution (Trap&Emulate) for most of the code

Switching between VMs

- Timer Interrupt in running VM.
- Context switch to VMM.
- VMM saves state of running VM.
- VMM determines next VM to execute.
- VMM sets timer interrupt.
- VMM restores state of next VM.
- VMM sets PC to timer interrupt handler of next VM.
- Next VM active.

- VMM will hold the system states of all VMs in memory
  - When VMM context-switches from one VM to another:
    - Write the register values back to memory
    - Copy the register values of next guest OS to CPU registers.



Intel “Vanderpool” VT-x [1/3]

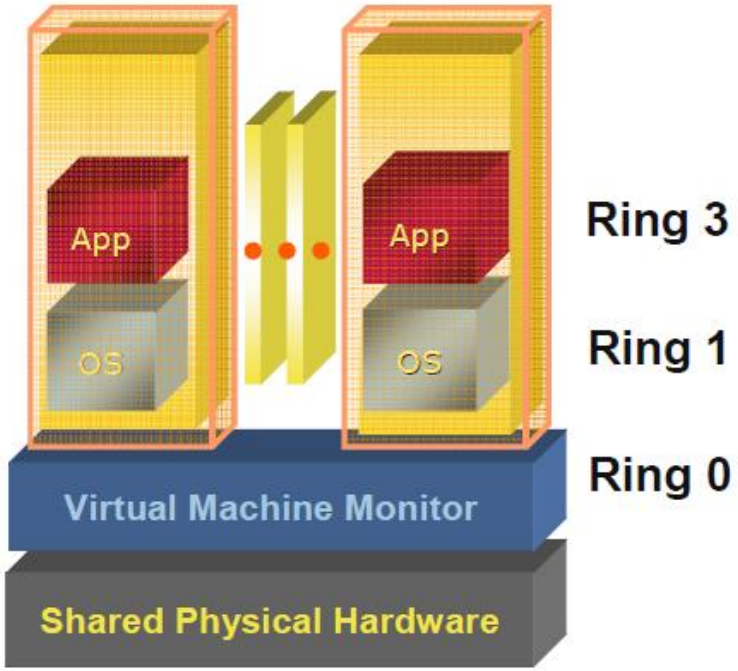
- VMX Root Operation (Root Mode)
  - All instruction behaviors in this mode are no different to traditional ones.
  - All legacy software can run in this mode correctly.
  - VMM should run in this mode and control all system resources.
- VMX Non-Root Operation (Non-Root Mode)
  - All sensitive instruction behaviors in this mode are redefined.
  - The sensitive instructions will trap to Root Mode.
  - Guest OS should run in this mode and be fully virtualized through typical “*trap and emulation model*”.

VM Control Structure (VMCS)

- specifies guest state (configured by VMM)
- manages VMX non-root operation and VMX transitions

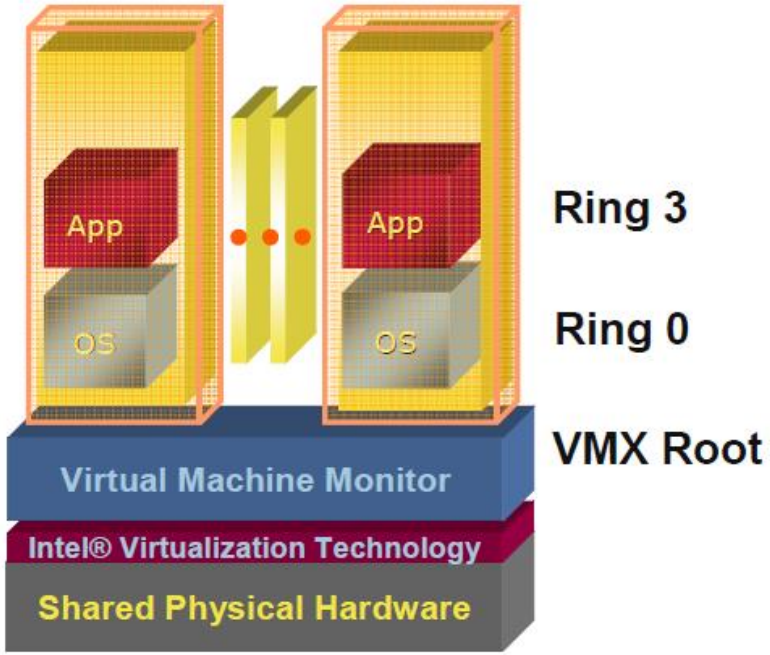
Intel VT-x [2/3]

Pre VT-x



- VMM de-privileges the guest OS into Ring 1, and takes up Ring 0
- Guest OS is unaware that it is not running in traditional ring 0 privilege
  - Requires compute intensive SW translation to mitigate

With VT-x



- VMM has its own privileged level where it executes
- No need to de-privilege the guest OS
- OSes run directly on the hardware

Intel VT-x [3/3]

- **VPID**: 16-bit virtual-processor-ID field in VMCS
  - Tag for cached linear translations
- No flush of TLBs on VM entry or VM exit if VPID active
  - Gen.1 VT-x forced TLB flush on each VMX transition → performance lost on all VM-exit and most VM-entry transitions.
- TLB entries of different VMs can all co-exist in the TLB.

Three abstractions of memory:

- Machine address space
- Physical address spaces
- Virtual address spaces

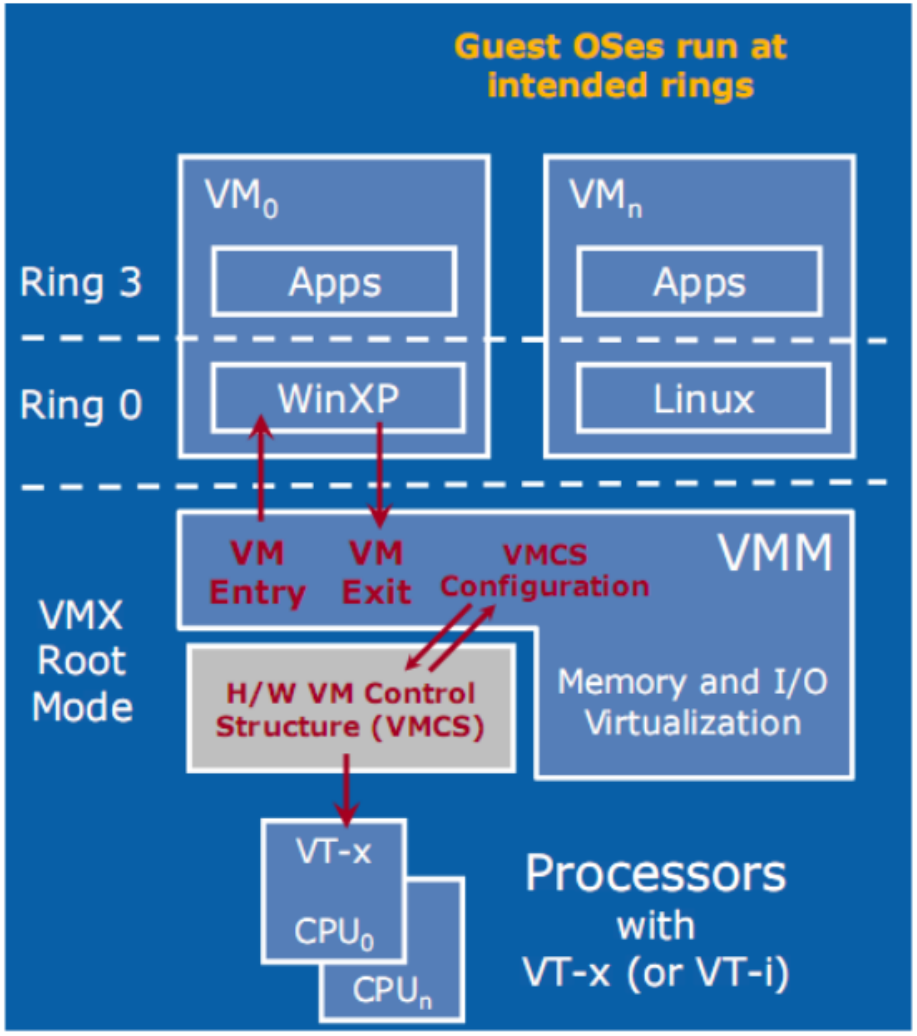
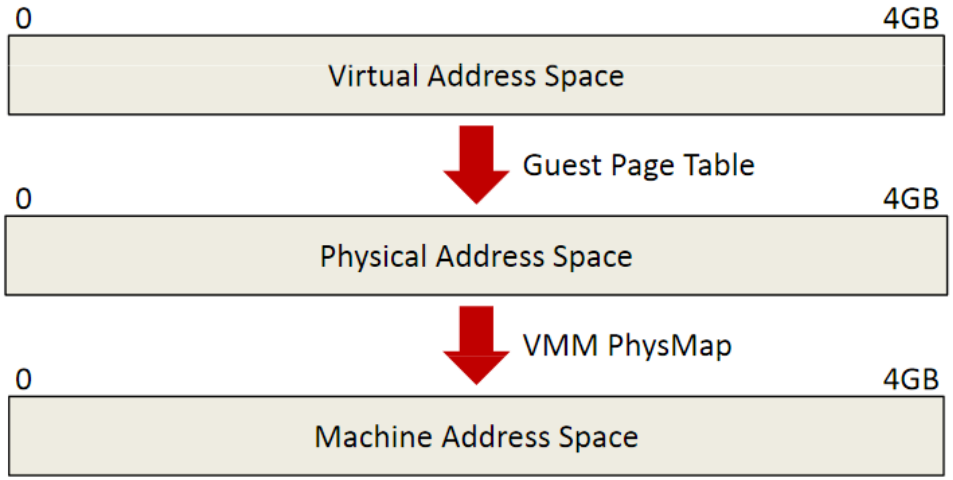
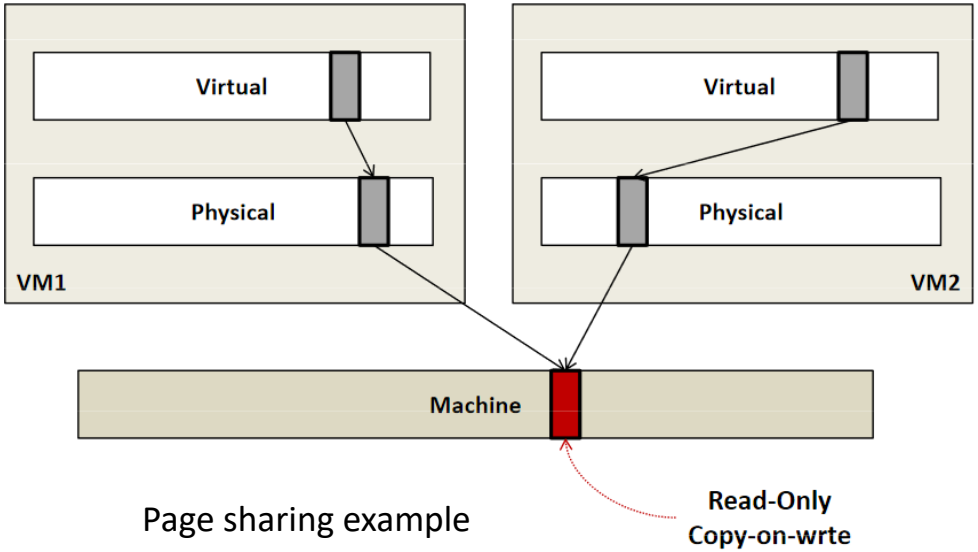


Image source: https://www.anandtech.com/show/2480/9

Memory virtualization



Motivation for nested page tables



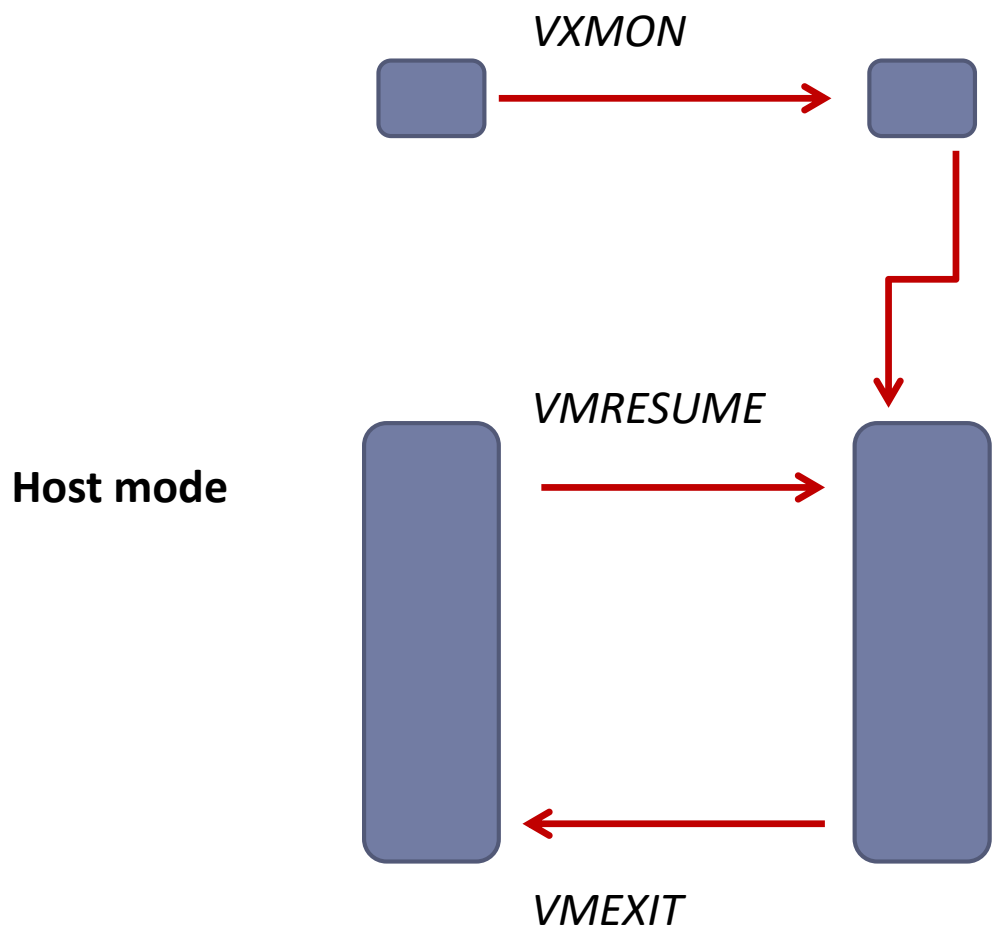
[source: Scott Devine (VMware)]

Page sharing example

VMCS: *Virtual Machine Control Structure (4KB page)*

- State Area
  - Store host OS system state upon VM-Entry.
  - Store guest OS system state upon VM-Exit.
- Control Area
  - Control instruction behaviors in Non-Root Mode.
  - Control VM-Entry and VM-Exit process.
- Exit Information
  - Provide the **VM-Exit reason** and associated hardware information.
- When VM Entry or VM Exit occur, CPU will automatically read/write corresponding information into VMCS.
  - VMM is responsible for VMCS configuration.

Intel VMX (Virtual Machine Extensions)



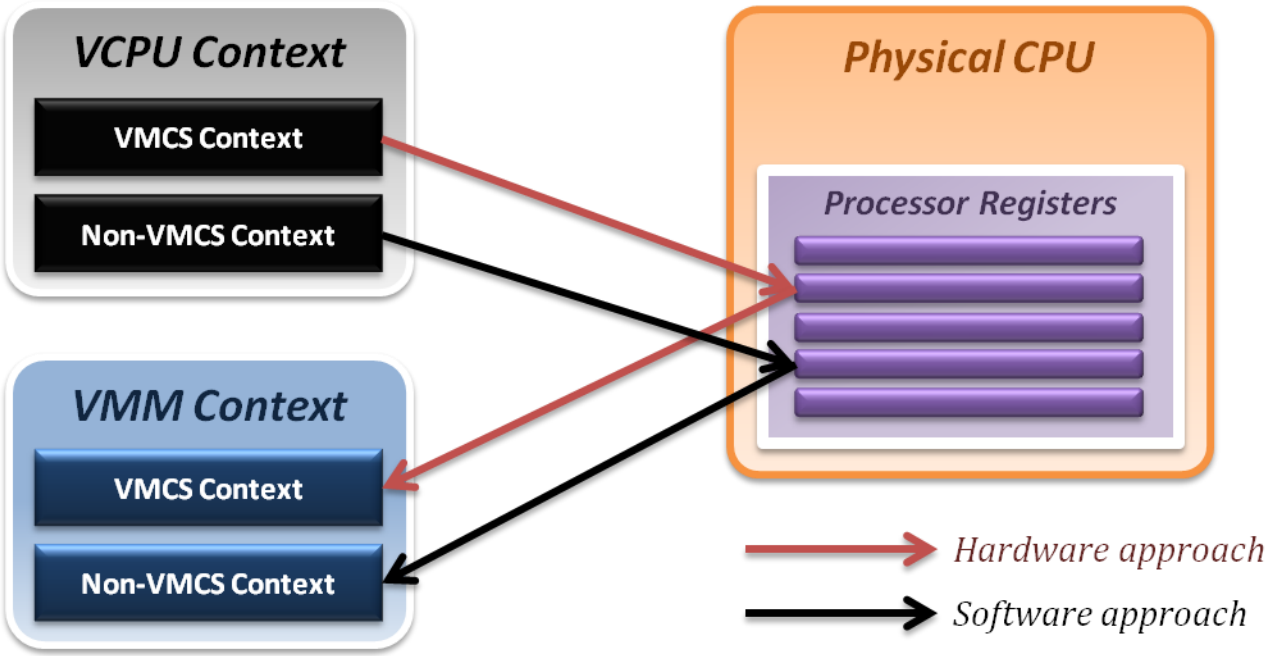
Expensive transitions:

- VM entry/exit
- Mode switch (world switch)

Some “exits” are heavier than others

System state management

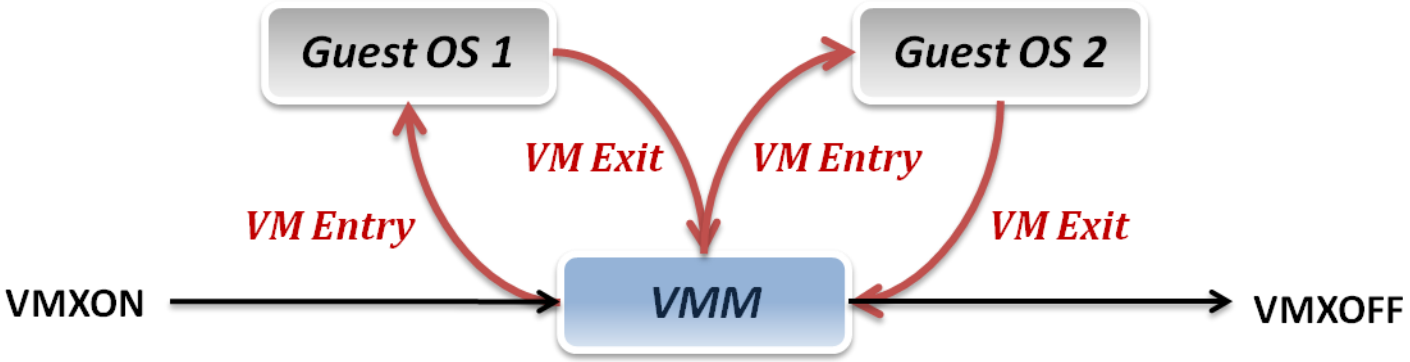
- Binding virtual machine to virtual CPU
  - VCPU (Virtual CPU) consists of two parts:
    - VMCS: maintains virtual system states, which is managed by hardware.
    - Non-VMCS: maintains other non-essential system information, which is managed by software.
  - VMM needs to handle the Non-VMCS part.



World Switch

Registers and address space swapped in one atomic operation.

- VMM switch different virtual machines with Intel VT-x :
  - VMXON/VMXOFF
    - These two instructions are used to turn on/off CPU Root Mode.
  - VM Entry
    - This is usually caused by the execution of **VMLAUNCH/VMRESUME** instructions, which will switch CPU mode from Root Mode to Non-Root Mode.
  - VM Exit
    - This may be caused by many reasons, such as hardware interrupts or sensitive instruction executions.
    - Switch CPU mode from Non-Root Mode to Root Mode.



Shadow page tables in the VMM

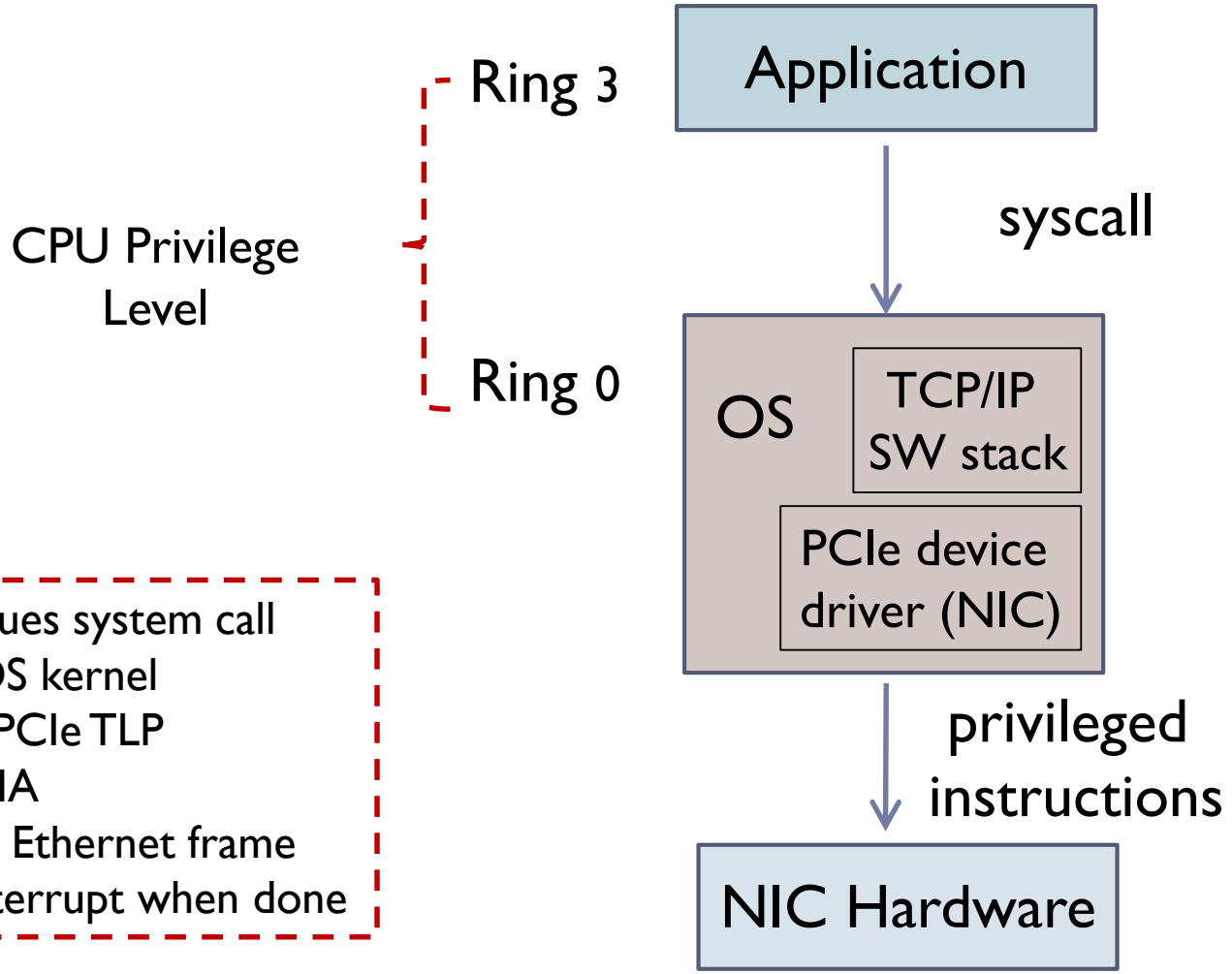
- VMM maintains shadow page tables that map guest-virtual pages directly to machine pages.
  - Guest modifications to page tables synced to VMM's shadow page tables.
- Guest OS page tables are marked as read-only.
  - Modifications of page tables by guest OS trigger trap to VMM.
- Shadow page tables are sync'ed to the guest OS tables.
- Maintaining consistency between guest page tables and shadow page tables leads to an overhead →VMM traps
  - Loss of performance due to TLB flush on every "world-switch".
  - Memory overhead due to shadow copying of guest page tables.

EPT: Extended Page Table

- Instead of walking along with only one page table hierarchy, EPT implements one additional page table hierarchy:
  - One page table is maintained by guest OS, which is used to generate guest physical address.
  - Another page table is maintained by VMM, which is used to map guest physical address to host physical address.
- For each memory access operation, the EPT MMU will directly get guest physical address from guest page table, and then get host physical address by the VMM mapping table automatically.
  - Benefit: Guest page table updates are not trapped → VM exits reduced.
    - Also: reduced memory footprint
  - Cost: TLB miss is very costly since guest-physical address to machine address needs an extra EPT walk (for each stage address translation).

One memory access from the guest VM may lead up to 20 memory accesses (for 4-level page table).

I/O (NIC) without virtualization

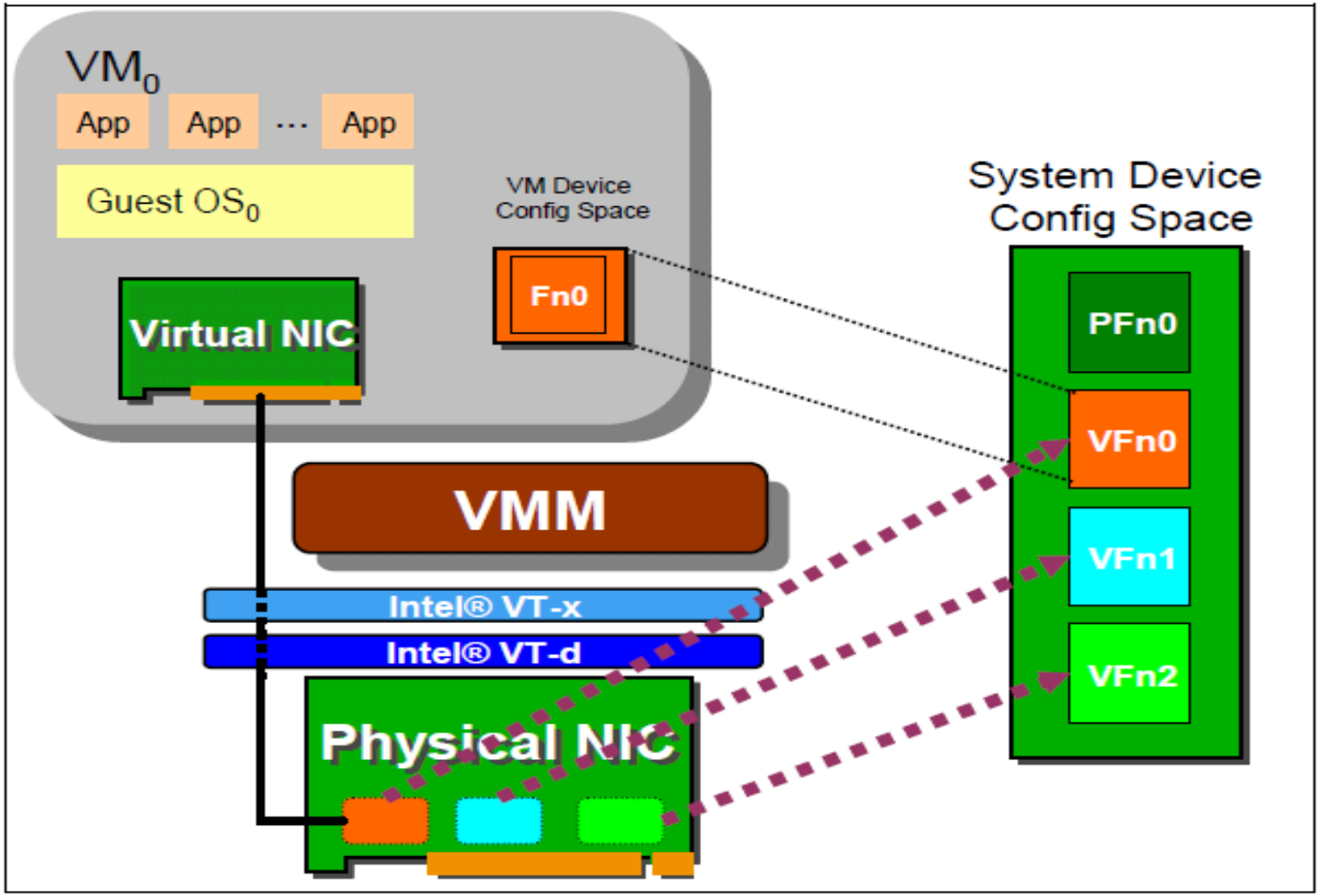


- Application issues system call
  - Trap to OS kernel
- Driver: issues PCIe TLP
  - Setup DMA
- NIC: transmits Ethernet frame
  - Assert interrupt when done

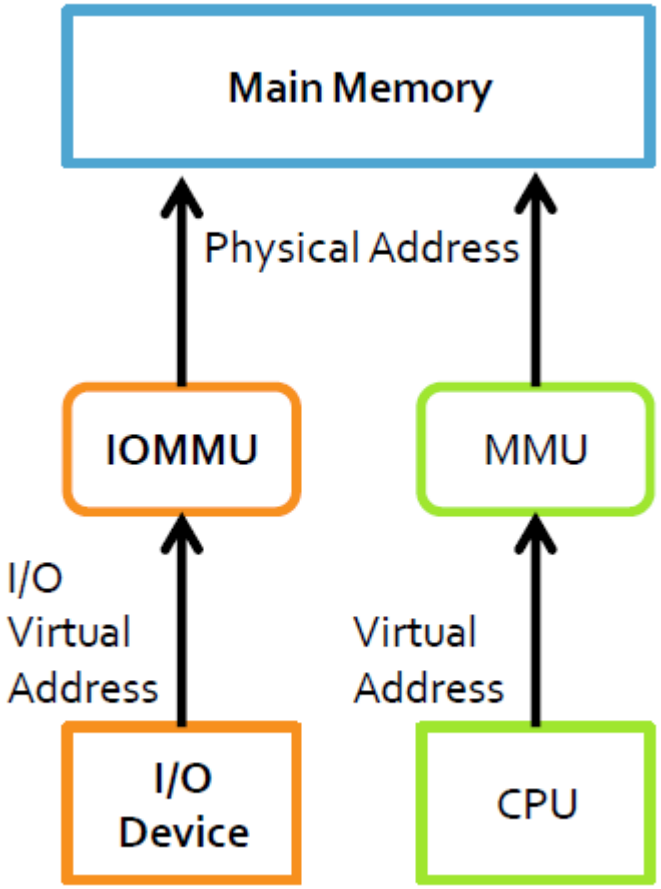
SR-IOV

- PCI-SIG Single Root I/O Virtualization and Sharing spec.
  - Physical Functions (PF)
    - Full configuration & management capability
  - Virtual Functions (VF)
    - “lightweight” functions: contain data-movement resources, with a reduced set of configuration resources
- An SR-IOV-capable device can be configured to appear (to the VMM) in the PCI configuration space as multiple functions
- The VMM assigns VF(s) to a VM by mapping the configuration space of the VF(s) to the configuration space presented to the VM.

SR-IOV : NIC example



IOMMU concept



- Indirection between addresses used for DMA and physical addresses
  - Similar MMU, for address translation in the device context
- Devices can only access memory addresses "present" in their protection domain
- I/O virtualization through direct device assignment

IOMMU

- Faster I/O via Pass-Through Devices
  - Exclusively used devices can be directly exposed to guest VM, without introducing device virtualization code
  - However, erroneous/malicious DMA operations are capable of corrupting/attacking memory spaces
  - Options:
    - Remapping of DMA operations by hardware (e.g. Intel VT-d)
    - Virtualizable devices (e.g. PCI-Express SR-IOV)

- I/O MMU
  - allow a guest OS running under a VMM to have direct control of a device
  - fine-grain control of device access to system memory
  - transparent to device drivers
  - software: swiotlb (bounce buffers), xen grant tables
  - hardware: AMD GART, IBM Calgary, AMD Pacifica

kvm components

- "hypervisor" module (/dev/kvm character device)
  - … heavily relies on Linux kernel subsystems
  - ioctl() API calls for requests
  - 1 file descriptor per "resource":
    - System: VM creation, capabilities
    - VM: CPU initialization, memory management, interrupts
    - Virtual CPU (vCPU): access to execution state
- Platform emulator (qemu)
- Misc. modules & tools:
  - KSM (memory deduplication for VM images)
  - Libvirt.org tools (virsh, virt-manager)

kvm Hypervisor API

- ioctl() system calls to /dev/kvm
  - Create new VM
  - Provision memory to VM
  - Read/write vCPU registers
  - Inject interrupt into vCPU
  - "Run" a vCPU

qemu-kvm for user-space:

- Config. VMs and I/O devices
- Execute Guest code via /dev/kvm
- I/O emulation

- /dev/kvm == 'communication channel' bet. kvm and QEMU
- Communication bet. kvm and VM instances:
  - kvm assumes x86 architecture with virtualization extensions
  - [Intel VT-x / AMD-V] VMCS + vmx instructions
    - VMXON, VMXOFF, VMLAUNCH, VMRESUME
    - [root →non-root] VM-Entry
    - [non-root → root] VM-Exit
    - VMCS: VMREAD, VMWRITE

Qemu main event loop

```
fd = open("/dev/kvm", O_RDWR);
ioctl(fd, KVM_CREATE_VM, …);
ioctl(fd, KVM_CREATE_VCPU, …);
for(;;) {
  ioctl(fd, KVM_RUN, …);
  switch(exit_reason) {
    case EXIT_REASON_IO_INSTRUCTION: … break;
    case EXIT_REASON_TASK_SWITCH: … break;
    case EXIT_REASON_PENDING_INTERRUPT: … break;
     …
  }
}
```

Event handling:

- timers
- I/O
- monitor commands

Event handling via select/poll system calls to wait on multiple file descriptors

VM creation with kvm

```
int fd_kvm = open("/dev/kvm", O_RDWR);
int fd_vm = ioctl(fd_kvm, KVM_CREATE_VM, 0);

ioctl(fd_vm, KVM_SET_TSS_ADDR, 0xfffffffffffd000);
ioctl(fd_vm, KVM_CREATE_IRQCHIP, 0);
```

1 process per Guest

1 thread per vCPU

+ 1 thread for the main event loop

+ ‘offload’ threads

Adding physical memory to a VM with kvm

```
void *addr = mmap(NULL, 10 * MB, PROT_READ | PROT_WRITE,
MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
struct kvm_userspace_memory_region region = {
    .slot = 0,
    .flags = 0, // Can be Read Only
    .guest_phys_addr = 0x100000,
    .memory_size = 10 * MB,
    .userspace_addr = (__u64)addr
};
ioctl(fd_vm, KVM_SET_MEMORY_REGION, &region);
```

vCPU initialization with kvm

```
int fd_vcpu = ioctl(fd_vm, KVM_CREATE_VCPU, 0);
struct kvm_regs regs;

ioctl(fd_vcpu, KVM_GET_REGS, &regs);
regs.rflags = 0x02;
regs.rip = 0x0100f000;
ioctl(fd_vcpu, KVM_SET_REGS, &regs);
```

Running a vCPU in kvm

```
int kvm_run_size = ioctl(fd_kvm, KVM_GET_VCPU_MMAP_SIZE, 0);
// access to the arguments of ioctl(KVM_RUN)
struct kvm_run *run_state =
        mmap(NULL, kvm_run_size, PROT_READ | PROT_WRITE,
        MAP_PRIVATE, fd_vcpu, 0);
for (;;) {
        int res = ioctl(fd_vcpu, KVM_RUN, 0);
        switch (run_state->exit_reason) {
            // use run_state to gather information about the exit
        }
}
```

Safely execute Guest code directly on the Host CPU (relying on virtualization support in the CPU)

Programmed I/O (PIO) in kvm

```
struct kvm_ioeventfd {
        __u64 datamatch;
        __u64 addr;       /* legal pio/mmio address */
        __u32 len;        /* 1, 2, 4, or 8 bytes    */
        __s32 fd;
        __u32 flags;
        __u8  pad[36];
};
```

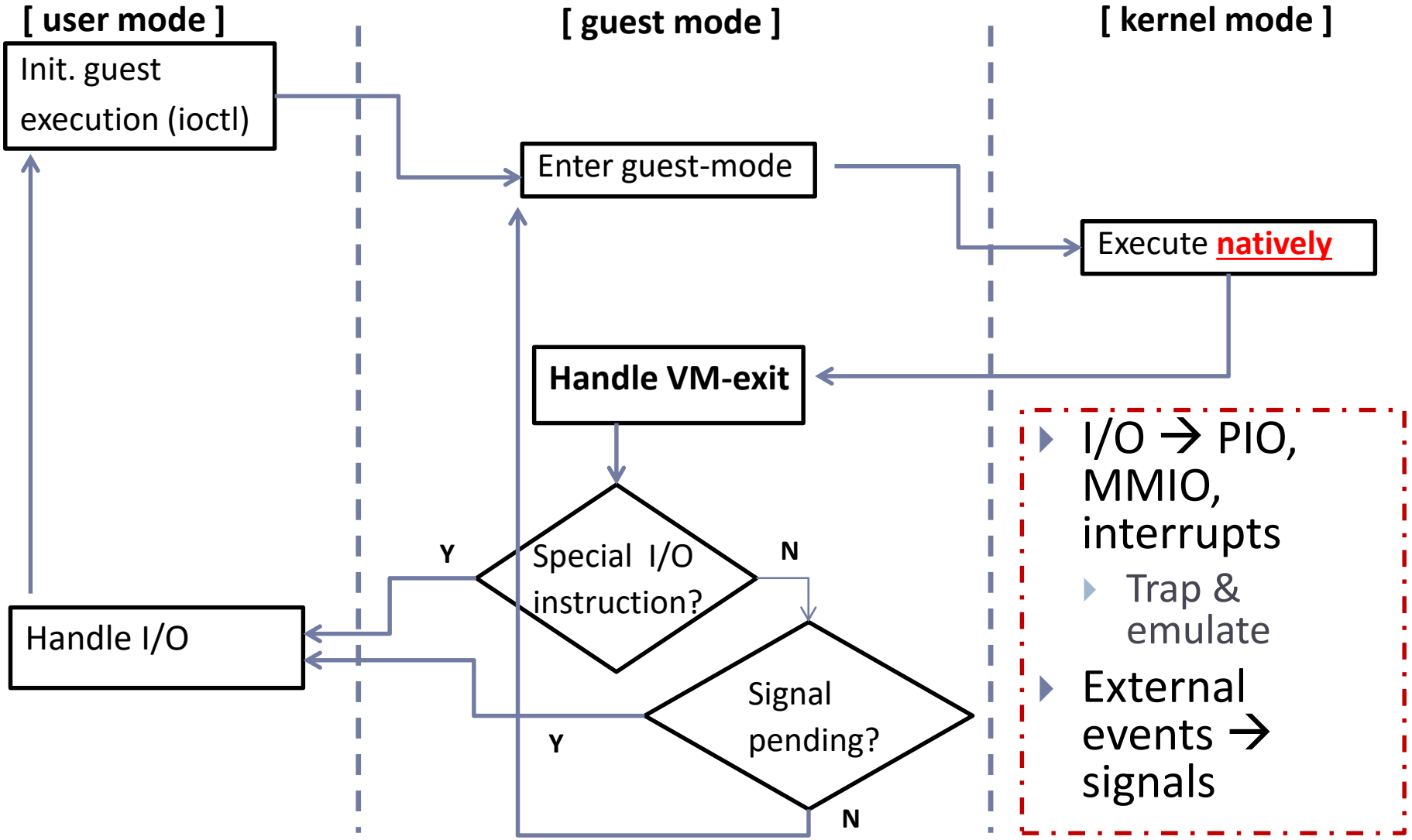
A guest write in the registered address will signal the provided event instead of triggering an exit.

Memory-mapped I/O (MMIO) in kvm

Exit reason : KVM_EXIT_MMIO

```
struct {
        __u64 phys_addr;
        __u8 data[8];
        __u32 len;
        __u8 is_write;
} mmio;
```

kvm guest execution flow



Sources (1)

- www.linux-kvm.org
- www.xen.org
- Mendel Rosenblum, Carl Waldspurger: "**I/O Virtualization**" ACM Queue, Volume 9, issue 11, November 22, 2011 URL: http://queue.acm.org/detail.cfm?id=2071256
- Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, **Xen and the Art of Virtualization**, SOSP'03
- **Xen (v.3.0. for x86) Interface Manual**
  - http://pdub.net/proj/usenix08boston/xen_drive/resources/developer_manuals/interface.pdf
- Jun Nakajima, Asit Mallick, Ian Pratt, Keir Fraser, **X86-64 XenLinux: Architecture, Implementation, and Optimizations,** OLS 2006
- **Xen: finishing the job,** lwn.net - 2009

Sources (2)

- Avi Kivity, et al: **kvm: The Linux Virtual Machine Monitor**, Proceedings of the Linux Symposium, 2007
  - http://www.linux-kvm.com/sites/default/files/kivity-Reprint.pdf
  - http://kerneltrap.org/node/8088
- Muli Ben-Yehuda, Eran Borovik, Michael Factor, Eran Rom, Avishay Traeger, Ben-Ami Yassour
  Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization, FAST '12
- Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, Dan Tsafrir **ELI: Bare-Metal Performance for I/O Virtualization**, ASPLOS '12
- Alex Landau, Muli Ben-Yehuda, Abel Gordon
  SplitX: Split Guest/Hypervisor Execution on Multi-Core, WIOV '11
- Abel Gordon, Muli Ben-Yehuda, Dennis Filimonov, Maor Dahan, **VAMOS: Virtualization Aware Middleware**, WIOV '11

Sources (3)

- 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**, *OSDI '10*
- Ben-Ami Yassour, Muli Ben-Yehuda, Orit Wasserman **On the DMA Mapping Problem in Direct Device Assignment**, *SYSTOR '10*
- Alex Landau, David Hadas, Muli Ben-Yehuda, **Plugging the Hypervisor Abstraction Leaks Caused by Virtual Networking**, *SYSTOR '10*
- Nadav Amit, Muli Ben-Yehuda, Dan Tsafir, Assaf Schuster, **vIOMMU: Efficient IOMMU Emulation**, *USENIX ATC 2011*
- *KVM ioctl() API: https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt*

QEMU Monitor Protocol

- Protocol allowing control of a QEMU instance
  - Text-based, easy to parse data format
  - JSON: JavaScript Object Notation (RFC 4627)
  - Details at https://wiki.qemu.org/Documentation/QMP
- qemu [...] **-qmp tcp:localhost:4444**,server,nowait –monitor stdio
  - telnet localhost **4444**
  - Returns QMP greeting banner
  - *qmp_capabilities* command: { "execute": "qmp_capabilities" }
  - QMP enters command mode
    - list of supported commands: { "execute": "query-commands" }
  - See also: **qmp-shell** (Under the scripts/qmp/ directory of the QEMU source tree)
- qmp-shell ./qmp-sock
- (QEMU) device_add driver=e1000 id=net1

Cost of VM exits

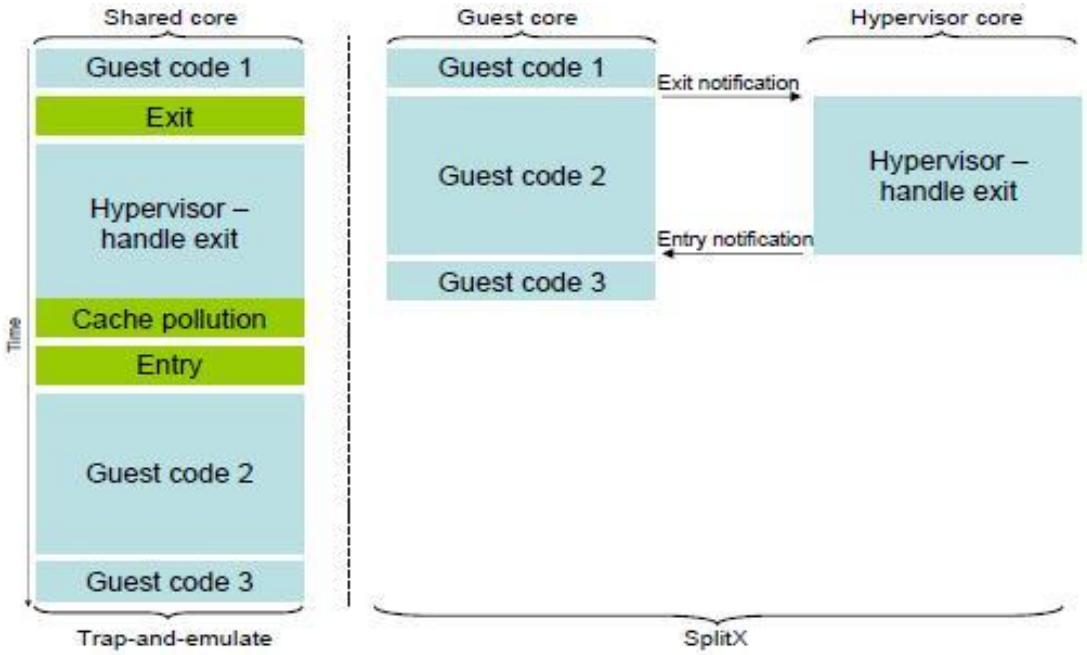
[source: Landau, et al, WIOV 2011]

Exit Type	Number of Exits	Cycle Cost/Exit
External interrupt	8961	363,000
I/O instruction	10042	85,000
APIC access	691249	18,000
EPT violation	645	12,000

- netperf client run on 1 GbE, with para-virtualized NIC
- Total run: ~7.1×10^{10} cycles vs ~5.2×10^{10} cycles for bare-metal
- **35% slow-down** due to the guest and hypervisor sharing the same core

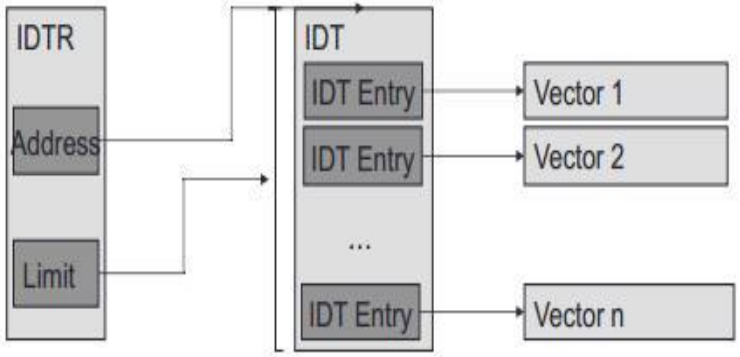
Acceleration of VMs (SplitX)

- Cost of an VM exit → 3 components
  - Direct (CPU "World switch")
  - Synchronous (due to exit processing in hypervisor)
  - Indirect (slowdown from having 2 contexts on same core → cache pollution)

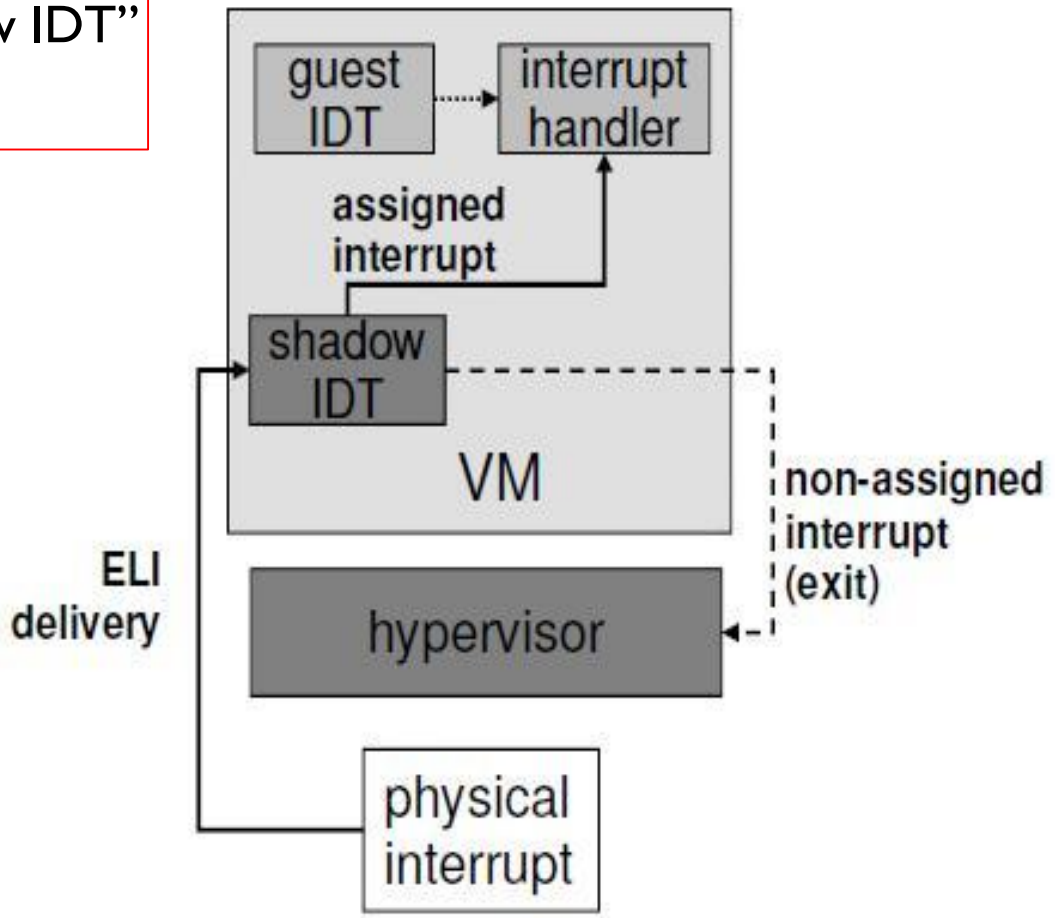


Acceleration of VMs (ELI)

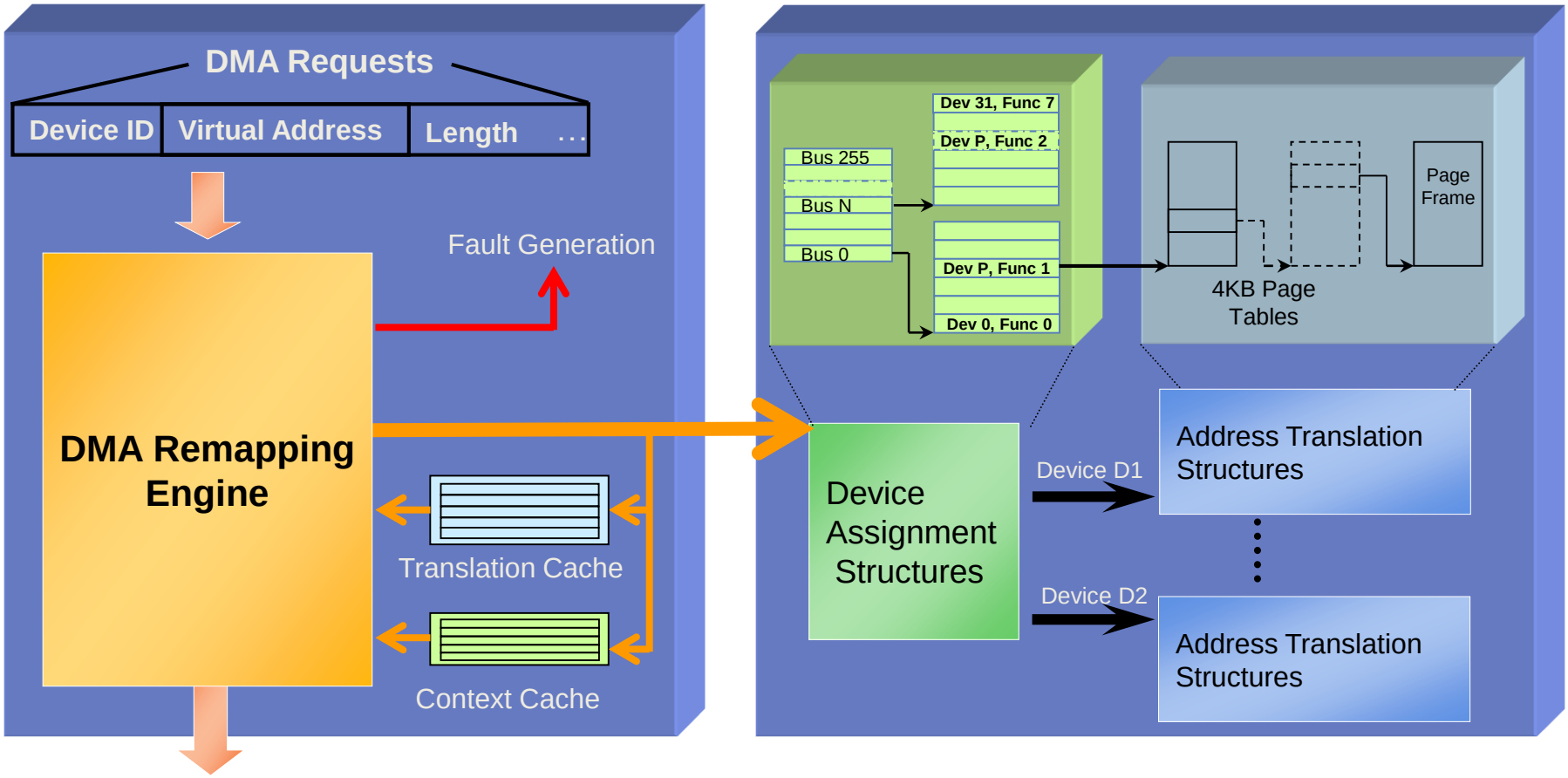
Instead of running the guest with its own IDT, run the guest with "shadow IDT" … prepared by the host



- I/O devices raise interrupts
- CPU temporarily stops the currently executing code
- CPU jumps to a pre-specified interrupt handler



Intel VT-d



Memory Access with System Physical Address

Memory-resident Partitioning And Translation Structures

AMD's IOMMU architecture

[source: AMD IOMMU specification (revision 2)]