B4M36ESW: Efficient software

Lecture 13: Virtualization

Michal Sojka
michal.sojka@cvut.cz

May 20, 2019

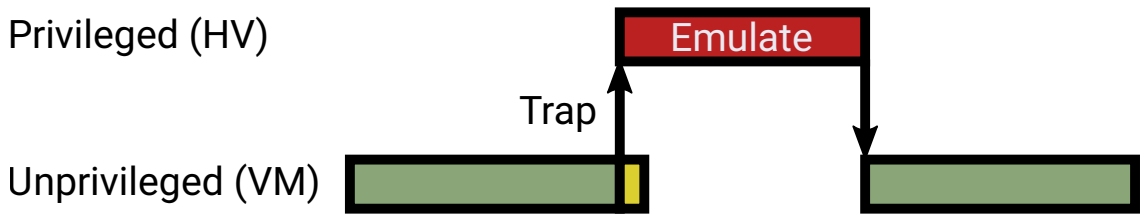
Outline

Outline

Virtualization

- **Definition:** Virtualization of the whole computing platform – the operating system thinks it runs on real hardware, but the hardware is largely emulated by the hypervisor (HV) and/or virtual machine monitor (VMM).
- Virtual machine (VM) vs. Java VM
  - Java VM interprets Java byte code and interacts with an operating system
  - VM executes native (machine) code and interacts with a hypervisor.
- VMs used since ’70s, mostly on IBM mainframes
  - Popek and Goldberg defined requirements for instruction set architecture (ISA) virtualization in their paper from 1974,
  - x86 became fully virtualizable in 2005.
- More detailed introduction to virtualization (from OSY course): `https://cw.fel.cvut.cz/b181/_media/courses/b4b35osy/lekce12_virt.pdf`

Trap-and-emulate

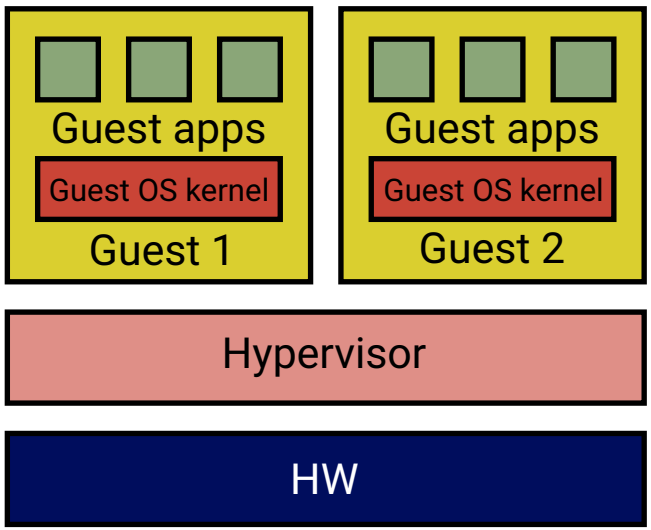


- Basic mechanism of virtualization
- Popek and Goldberg: "*All sensitive instructions must be privileged instructions*"
  - **Sensitive instruction**: Changes *global state*[1] or behaves differently depending on *global state* (e.g. cli, pushf on x86)
  - **Privileged instruction**: Unprivileged execution traps to the privileged mode (to the hypervisor via CPU exception)
  - on x86 `popf`, `pushf` and few other instructions were not privileged!
    - pushf stores all flags to stack including "global" interrupt flag (IF)
    - popf sets IF in privileged mode and ignores it in unprivileged mode (does not trap)
- Hypervisor (HV) can emulate the effect of sensitive instructions depending on the VM state (not the global state).

[1] Global state means a state that is common to all running VMs, not local to a single VM. For example, CPU reset signal is global.

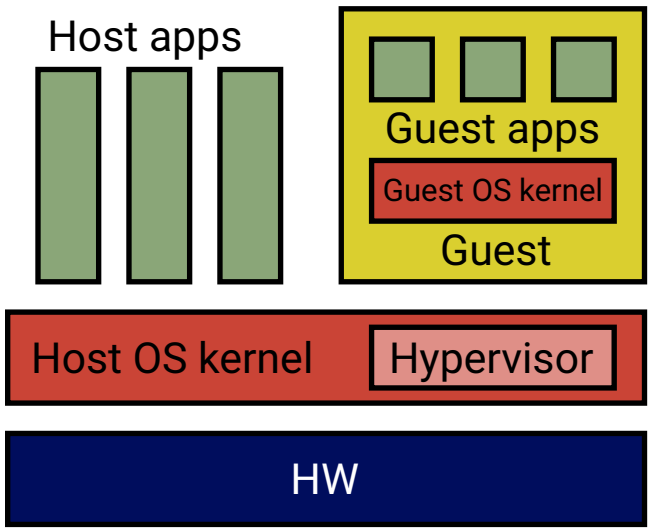
Hypervisor

- Privileged code that supervises execution of the VM, i.e. handles traps.
- Hypervisor types:



Bare-metal hypervisor

- Examples: Xen, VMware ESX, ...
- Idea: Avoid overheads of the general purpose OS.

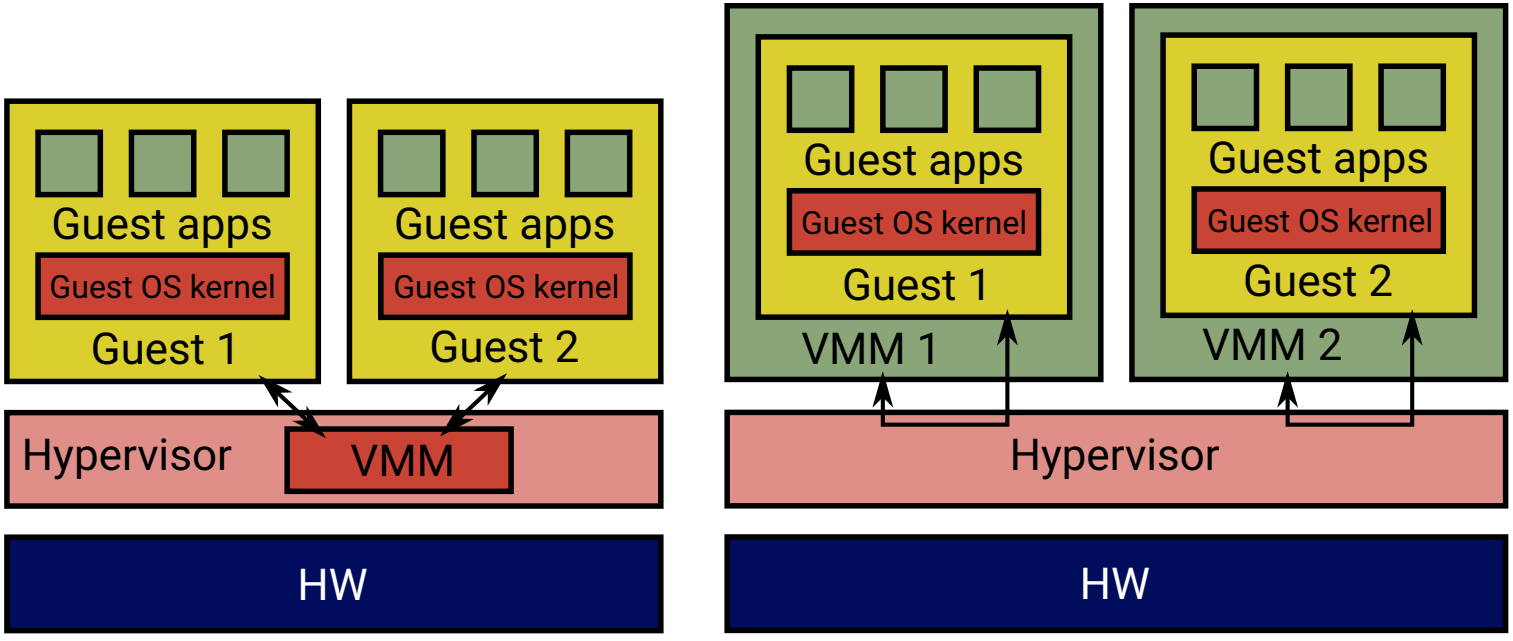


Hosted hypervisor

- Examples: KVM, VirtualBox, ...
- Idea: Why to reinvent the wheel?

- The boundary is blurry – many bare-metal hypervisors support native apps

Virtual Machine Monitor (VMM)



- Software that emulates the HW platform (network, graphics, storage, ...)
- Often implemented inside hypervisor (left) ⇒ people confuse VMM with hypervisors
- Today's platforms are complex (e.g. PC bears 40 years heritage, millions lines of emulation code)
- It is more secure to execute the VMM in user mode, outside of privileged mode (right, example: KVM & qemu)
- It is also slower, but see NOVA microhypervisor (TU Dresden), which implements this faster.

Questions

- How many privilege levels we need to implement virtualization?
  - Two are sufficient, but then, every guest system call, page fault etc. traps from the guest app to the hypervisor, which then arranges switch to the guest kernel – this is slow.
  - Hardware assisted virtualization – introduces more privilege levels (and other features) to make virtualization faster – see later.
- Why is virtualization needed at all? (My personal rant)
  - To some extent because the design of mainstream operating systems is not up to the current needs.
  - Current OSes do not offer sufficient isolation of applications and groups of applications. Many things such as user permissions, apply implicitly to the whole system.
  - Microkernel OSes, which solve this problem, were designed in the past without much success.
  - Now, people are adding “containers” to mainstream OSes, which is painful and often with security problems.
    - Making a microkernel from a monolithic kernel[2] is more difficult that starting with the microkernel from scratch.

[2] `https://marc.info/?l=linux-mm&m=154952428103397`

Outline

Hardware assisted virtualization

- Accelerates virtualized execution
- Differences between vendors (Intel, AMD, ARM, ...), core principles similar:
  - More privilege levels (x86 – root/non-root, ARMv8 EL0–3)
    - HW emulates things faster than SW
  - Nested paging
  - IO virtualization

Intel VMX

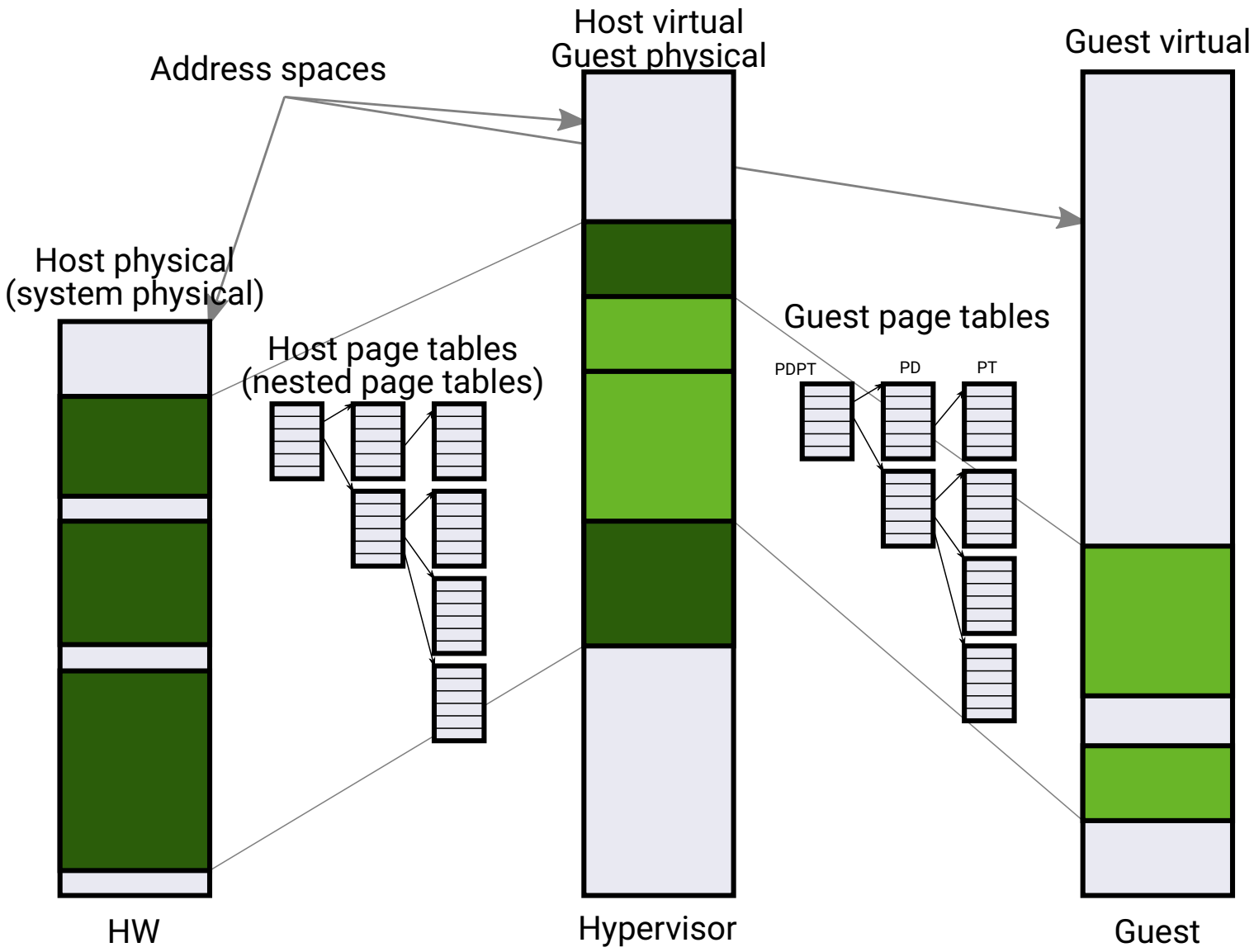
- VMX root operation
  - host rings 0–3
- VMX non-root operation
  - guest rings 0–3
  - in non-root mode, x86 ISA is fully virtualizable
- Transition root→non-root = **VM Enter**
  - instructions: vmlaunch, vmresume
- Transition non-root→root = **VM Exit**
  - instructions: vmresume, vmcall
  - faults (e.g. I/O)
- VM Control Structure (VMCS)
  - Data structure in memory that controls VMX execution (managed by the hypervisor/VMM)
  - (Re)stores host/guest state
  - “Large structure” ⇒ VM Enter/Exit has overhead
  - The overhead depends on what is (re)stored from/to VMCS (configurable)

VMCS (up to 4 KiB – e.g. 1024 B)

VM-execution control fields
Host state
Guest state
VM-exit information fields
VM-entry control fields
VM-exit control fields

Outline

Nested paging



Memory access overhead

- TLB misses and page faults are more expensive inside VM!
- Page walk in a VM (4-level page tables):

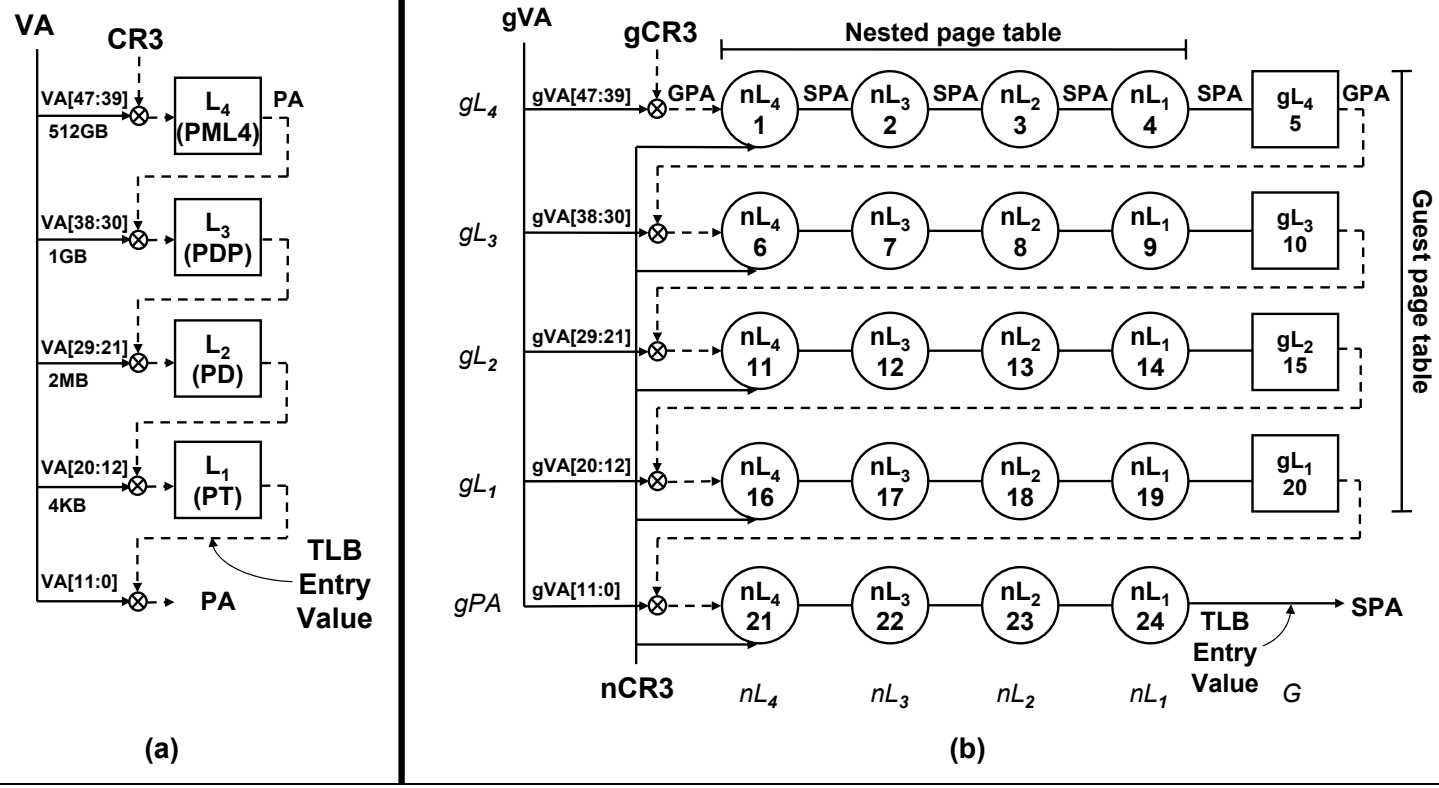


Figure 1. (a) Standard x86 page walk. (b) Two-dimensional page walk. Italics indicate column and row names; notations such as *{nL$_1$,gPA}* and *{G,gL$_1$}* indicate entries in the indicated columns and rows.

Source: Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. SIGOPS Oper. Syst. Rev. 42, 2 (March 2008), 26-35.
DOI: `https://doi.org/10.1145/1353535.1346286`

Page walk in a VM

4-level page tables (64-bit systems)

- 2D page walk – translation of guest virtual address to system physical address:
  - Access to each level of the guest page table has to be translated to system physical addresses via nested page table.
  - TLB miss in virtualized systems results in 24 memory accesses (non-virtualized systems need only 4 memory accesses).
  - If we are lucky, some of these memory accesses are served from a cache rather than from slow main memory.
  - Performance drop up to 15% (Intel), 38% (AMD)[3]
- Tagged TLBs (HW feature of modern CPUs)
  - No need to flush TLBs on process (or VM) switches (good)
  - Applications share TLBs with the hypervisor and VMM (bad)
- Recommendation: Use huge pages (2 MB) if possible
  - Bigger pages ⇒ less TLBs for the same amount of memory
  - Note: Older implementation of huge pages in Linux had performance problems and people recommended not using them. Now, the situation is different.

[3] Ulrich Drepper, The Cost of Virtualization, ACM Queue, Vol. 6 No. 1 – 2008

Outline

KVM

- Linux-based hosted hypervisor
- Abstracts hardware-assisted virtualization of different architectures behind `ioctl`-based API
- We will develop a miniature user-space VMM
  - Simplest hardware to virtualize: serial port

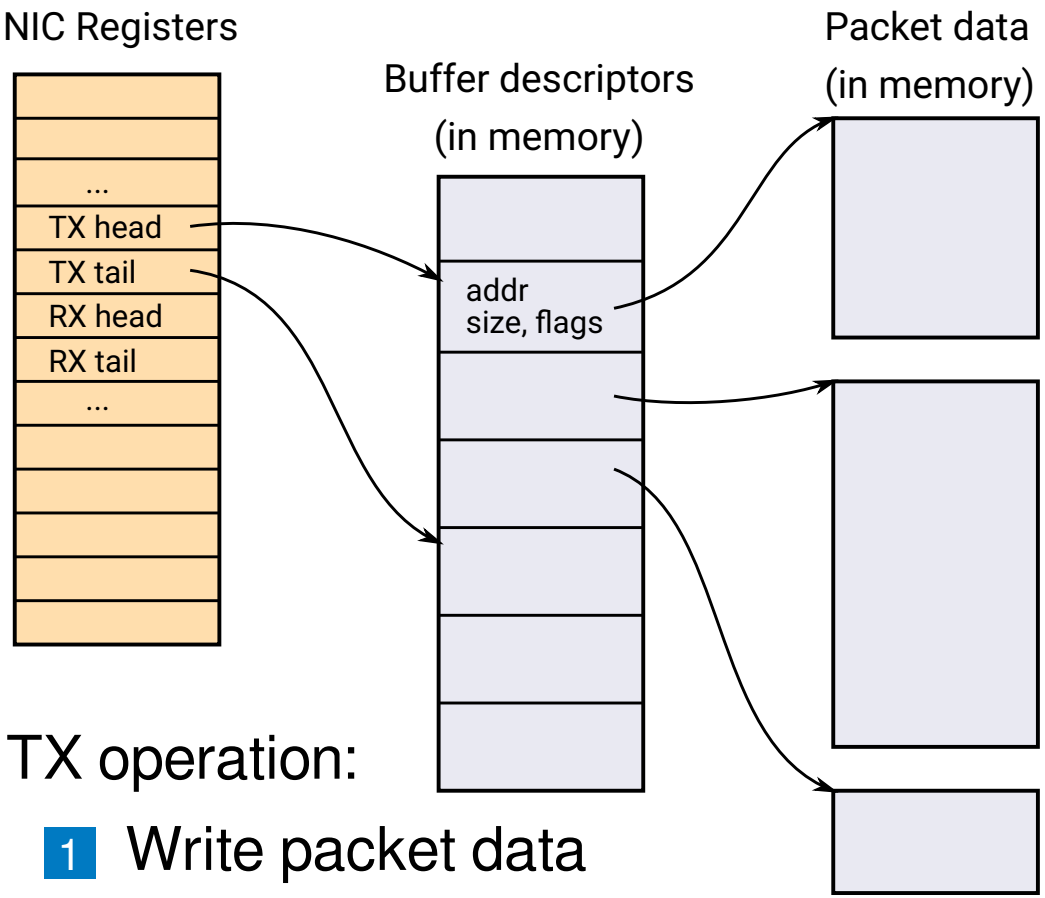
  1. Setup the VM's memory
  2. Load the code to execute
  3. Run the VM
  4. Handle the VM Exits and emulate serial port
  5. Goto 3

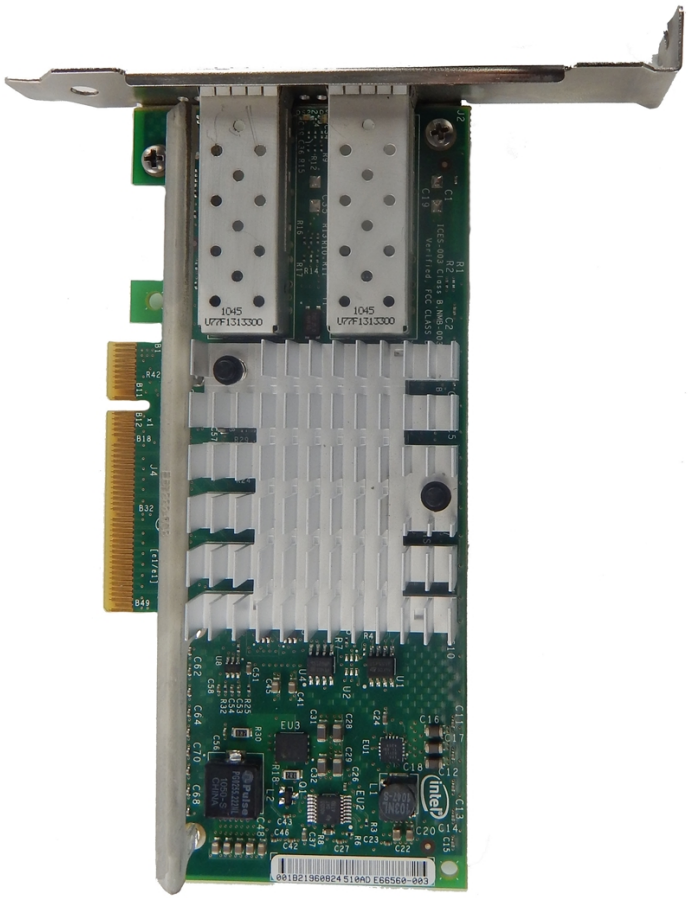
  - See also `https://lwn.net/Articles/658511/`

Outline

Outline

Network Interface Card & transmit operation

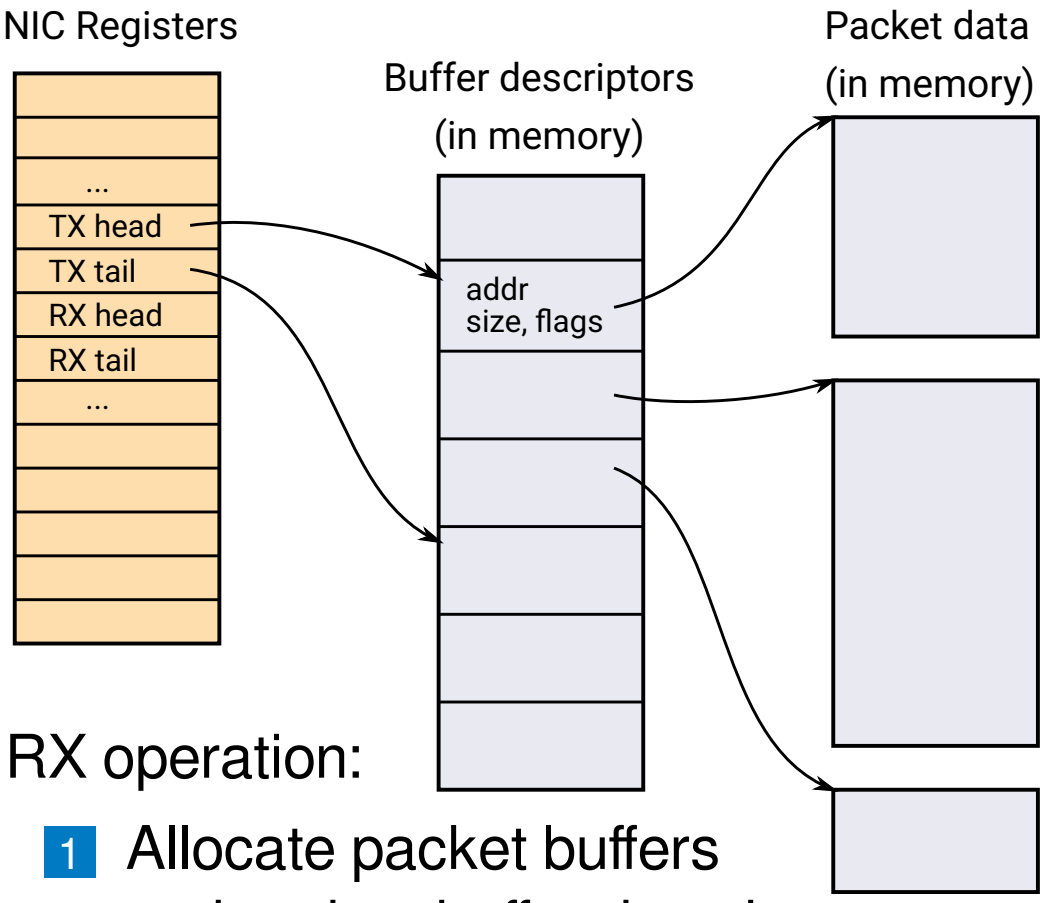


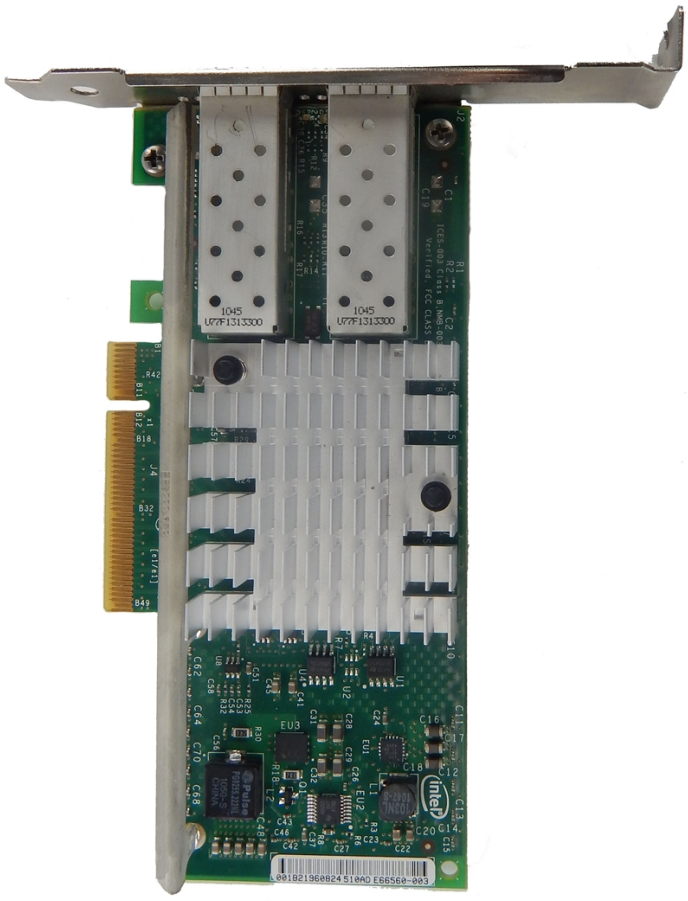


TX operation:

1. Write packet data
2. Fill in an empty buffer descriptor
3. Notify the NIC by writing TX tail reg

Network Interface Card & receive operation

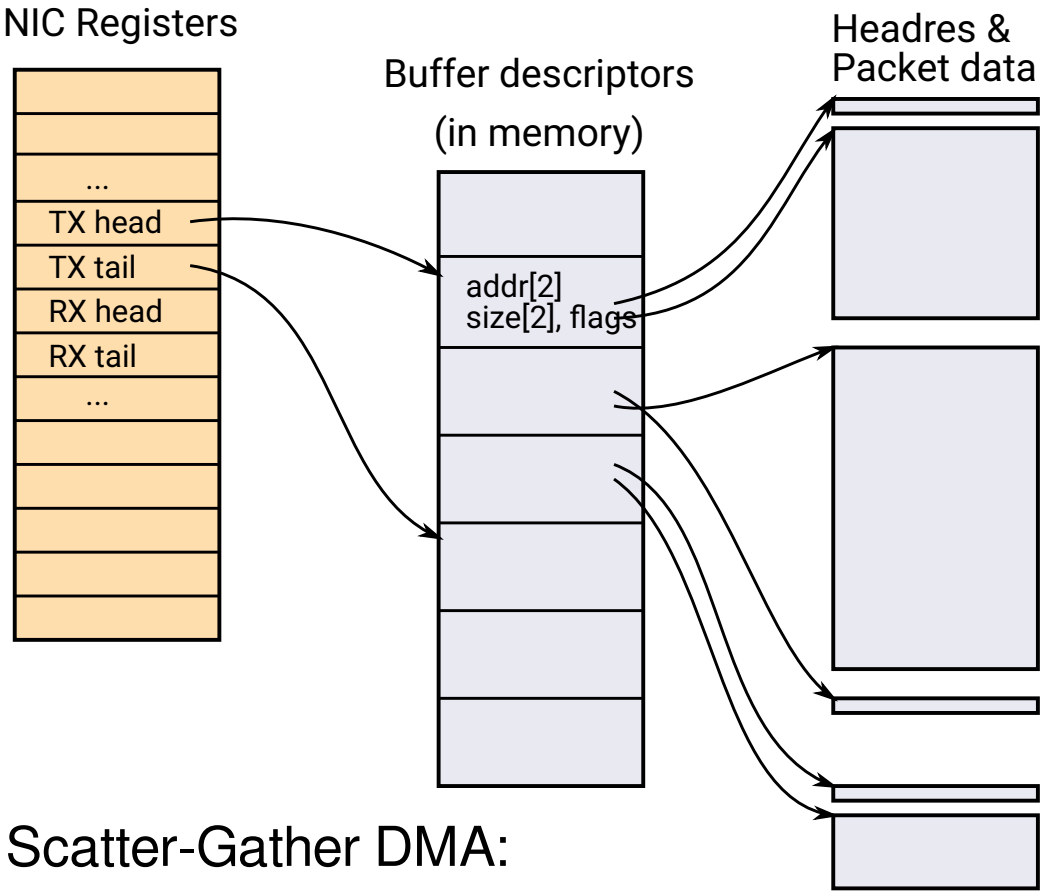


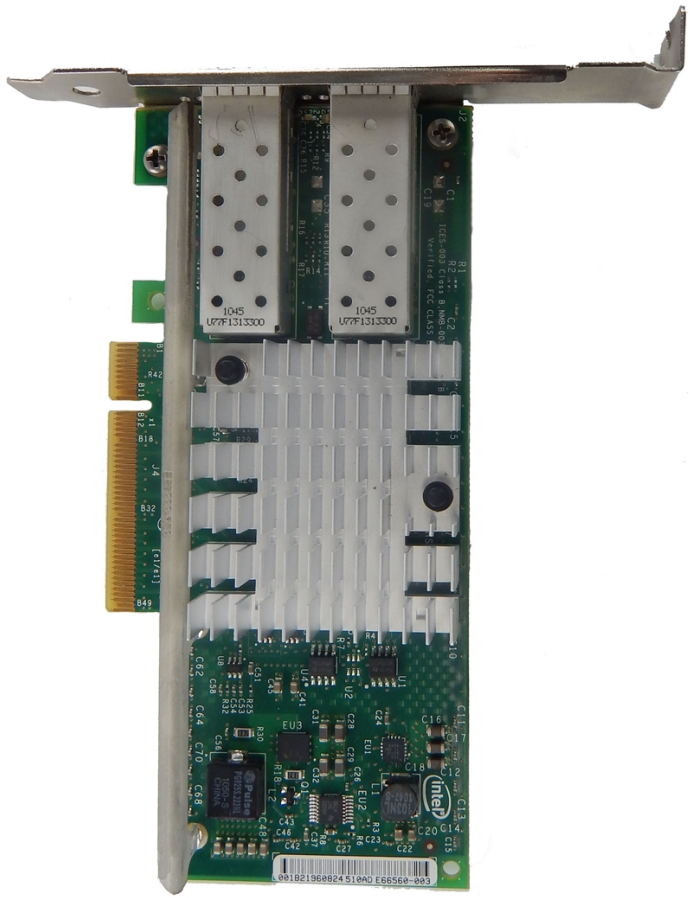


RX operation:

1. Allocate packet buffers and update buffer descriptors
2. Update RX head/tail regs
3. On packet reception, NIC stores the data to memory and generates an interrupt

Network Interface Card & SG DMA





Scatter-Gather DMA:

- NIC composes the final packet from several pieces scattered in memory
- Typically header (from OS) and data (from app)

Outline

NIC device emulation

- Trap accesses to NIC registers (memory-mapped IO)
- Upon write to TX tail, VMM iterates over queued buffers and sends them via real NIC (e.g. SOCK_RAW)
- Multiple packets can be sent during a single VM Exit ($\Rightarrow$ less overhead)
- Reception works similarly

- Not all hardware is “that nice” to virtualize
  - Several VM Exits per TX or RX
  - Registers that must be trapped are intermixed with non-sensitive (e.g. read-only) registers in a single page
    - $\Rightarrow$ Unnecessary VM Exits for some register accesses
- VMM must emulate not only RX/TX, but also management
  - Link negotiation, configuration, ...
  - More complex compared to RX/TX

Outline

Virtio

- It is neither easy nor necessary to emulate a real NIC
- Emulation of TX, RX and simple configuration (e.g. MAC address) is sufficient
- Why to implement different ring-buffer formats?

- Virtio[4]
  - Universal ring-buffer-based communication between VM and HV
  - Used for network, storage, serial line, ...
  - PCI-based probing & configuration – VMs can easily discover virtio devices

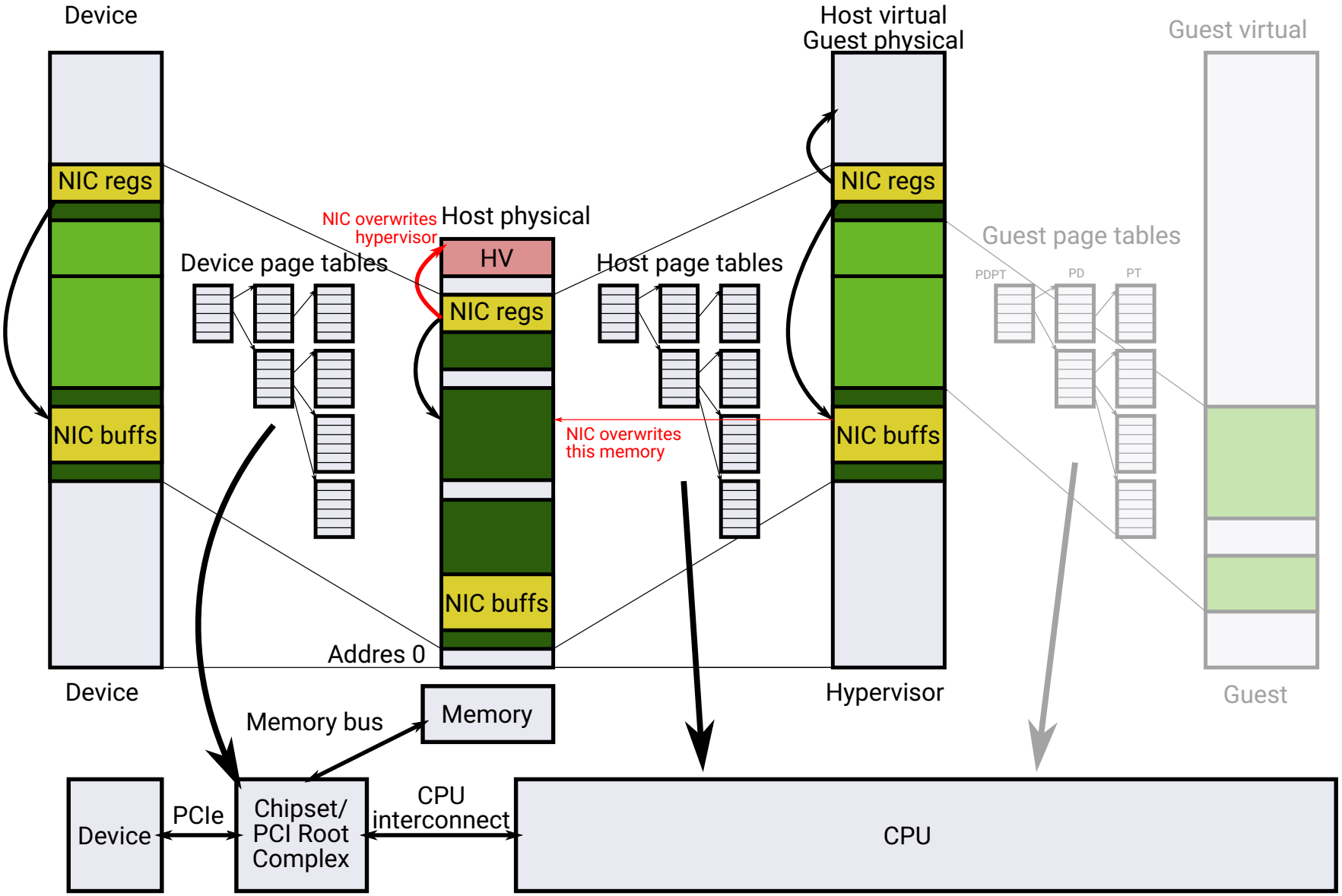
[4] R. Russell, virtio: Towards a De-Facto Standard For Virtual I/O Devices, ACM SIGOPS Operating Systems Review, 2008

Outline

PCI pass-through

- Even virtio needs one VM Exit per (a batch of) TX operation(s)
- If we don't want VM Exits, we may want to give a VM exclusive access to the NIC
- Few problems to solve...

PCI pass-through graphically



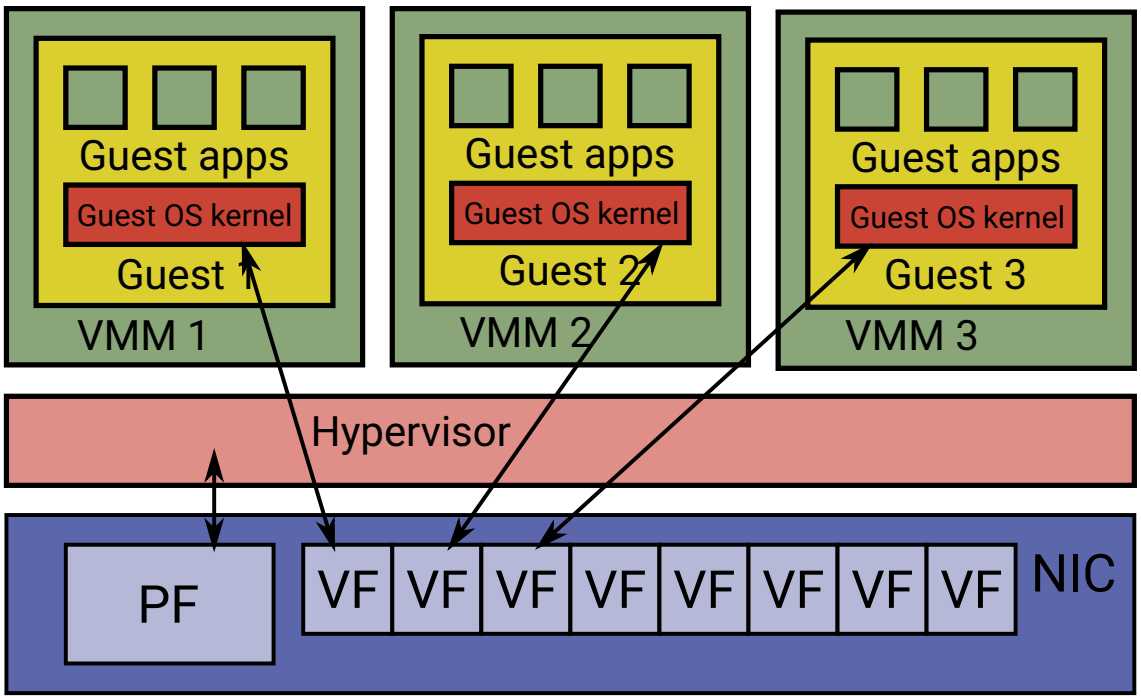
PCI pass-through

- Problems:
  1. Virtual address space (see previous slide)
     - Security: One VM could configure the NIC to read or write memory of other VM or even the hypervisor!
  2. Device interrupts: Host does not know how to acknowledge (silence) the interrupt – it has no driver for the device
     - Host injects the interrupt to the VM and returns from the IRQ handler
     - Host is interrupted again, because the VM didn't have chance to run and ack the interrupt
     - $\Rightarrow$ infinite loop
- Solution: Hardware support for direct use of devices in VMs
  1. IOMMU (AMD), VT-d (Intel), SMMU (ARM)
  2. Mask individual sources of interrupts without understanding the device
     - Hard with (parallel) PCI, where interrupt lines are shared between devices
     - Possible with Message Signaled Interrupts (MSI), PCI-express

Outline

Single-Root I/O Virtualization (SR-IOV)

- PCI pass-through is nice, but I have more VMs that want to communicate…
  - Each VM has an emulated NIC, VMM multiplexes the real NIC between VMs in software
  - What about performing the multiplexing in hardware?

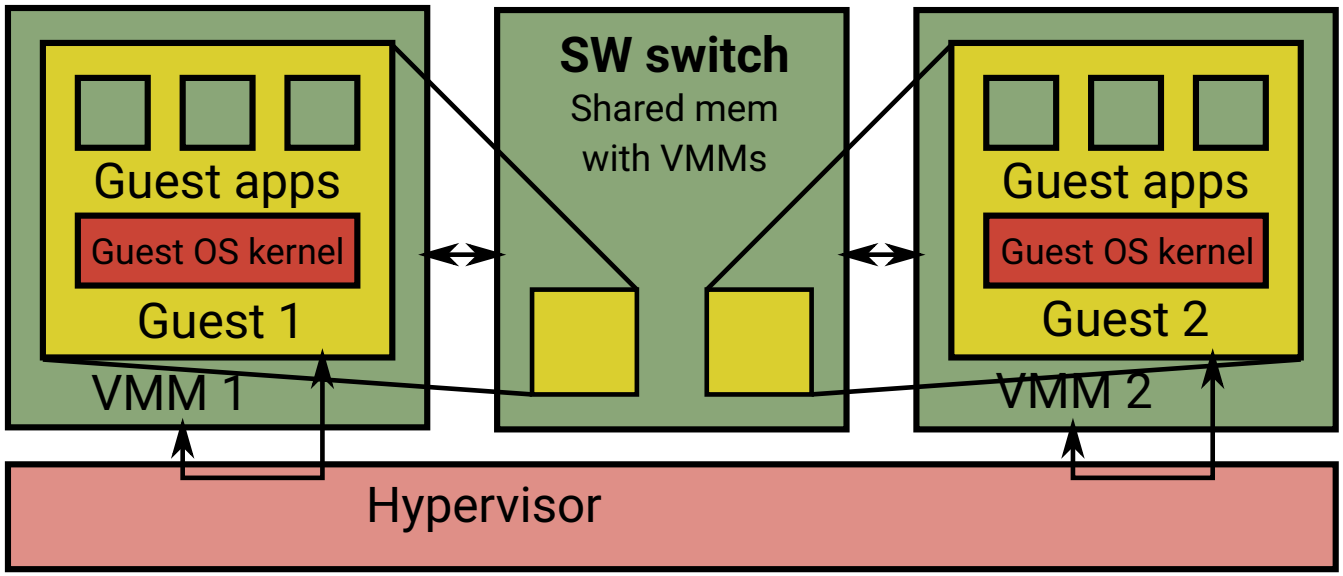


- SR-IOV
  - Besides “classic” physical function (PF), NIC implements several virtual functions (VFs)
  - Each VF provides simplified PCI interface and its own RX/TX ring buffers

Outline

Efficient inter-VM networking

Software-based Ethernet switch



- Packet stored in VM's memory
- VMM notified (VM Exit) e.g. via virtio's kick()
- VMM notifies the SW switch via standard IPC mechanism
- Switch does `memcpy()` of the packet from source VM to destination VM (into dest NIC ring buffer)
  - Note: The switch can see (`mmap()`) all VMs memory – the same as for real hardware NIC
- Dest VMM notifies the VM (injects an interrupt)

Optimizations

- OS networking stack is responsible for splitting application data to packets (e.g. TCP segmentation) and adding appropriate headers
- VMM sees many small packets and the switch does many small memcpy()s
- Receiver's networking stack strips packet headers and combines the payload to larger data chunks for application.

- TCP segmentation is not necessary for Inter-VM communication (overhead)!
- Modern NICs support TCP Segmentation Offload (TSO)/Large Receive Offload (LRO): Segmentation/reconstruction is done in hardware.
- If the virtual NIC supports TSO/LRO, Inter-VM communication is much faster, because whole TCP segments (in contrast to small packets) can be copied at once.

Outline

Summary

- Virtualization is just “another layer of indirection” and as such it adds overheads
- It is useful to know where the overheads are and how to mitigate them