

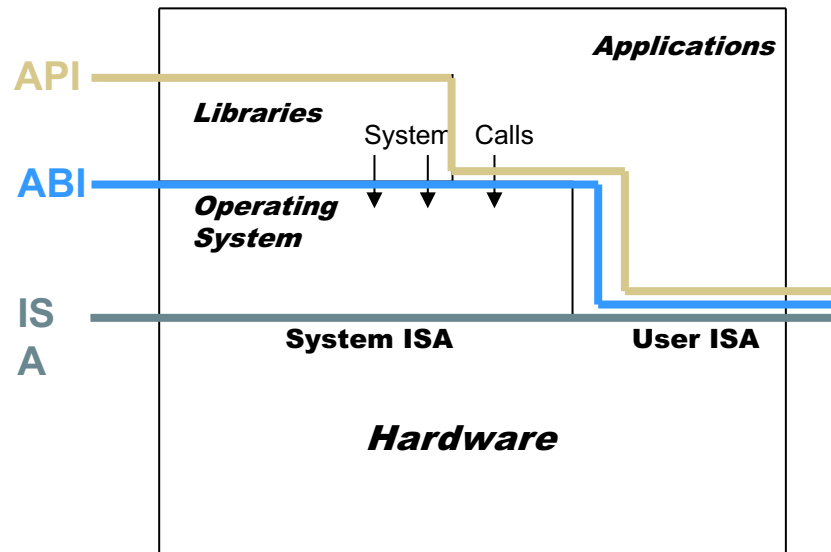
System Virtualization

IPADS, Shanghai Jiao Tong University

https://ipads.se.sjtu.edu.cn/pub/members/zeyu_mi

Virtualization Layers

- **At which layer?**
 - Hello world
 - Web game
 - Dota
 - Office 2013
 - Windows 8
 - Java applications
 - Python scripts
 - High Sierra



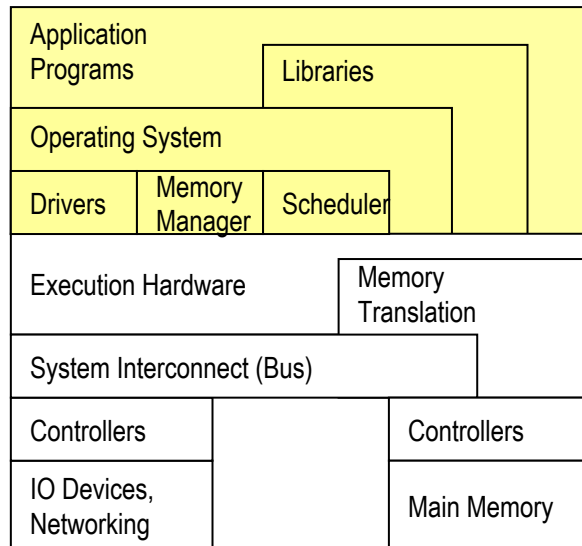
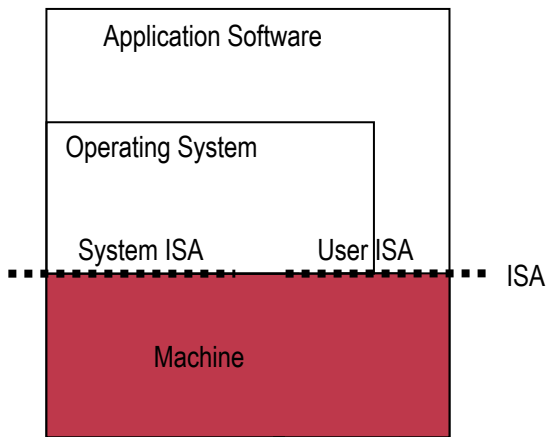
API – application programming interface

ABI – application binary interface

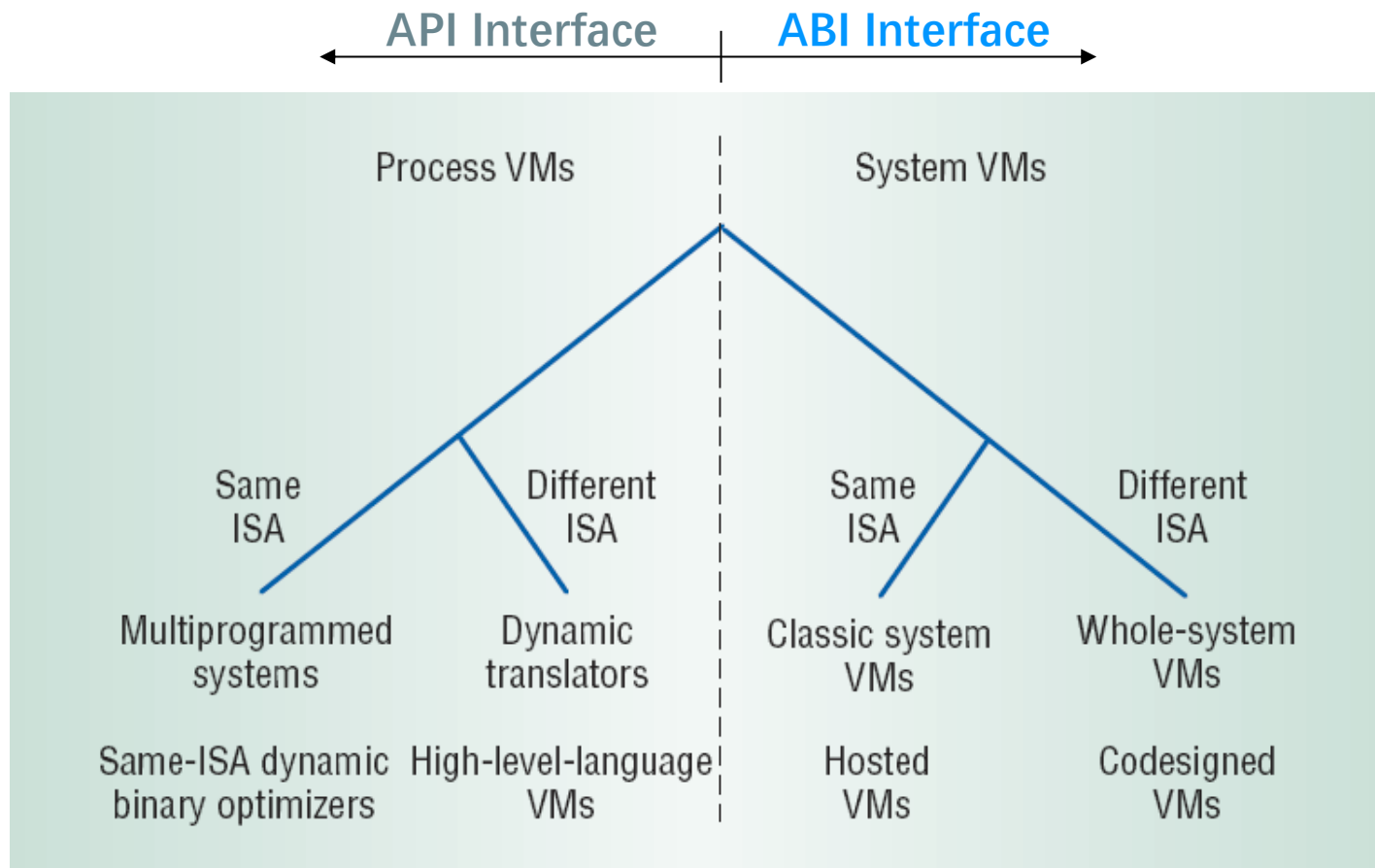
ISA – instruction set architecture

What is a VM and Where is the VM?

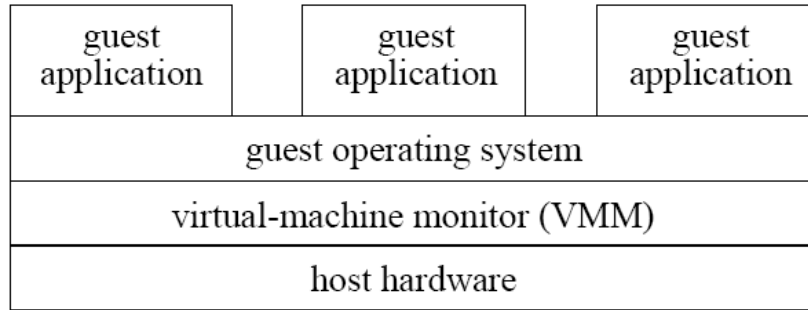
- Machine from the perspective of a system
 - ISA provides interface between system and machine



Design Space (Level vs. ISA)

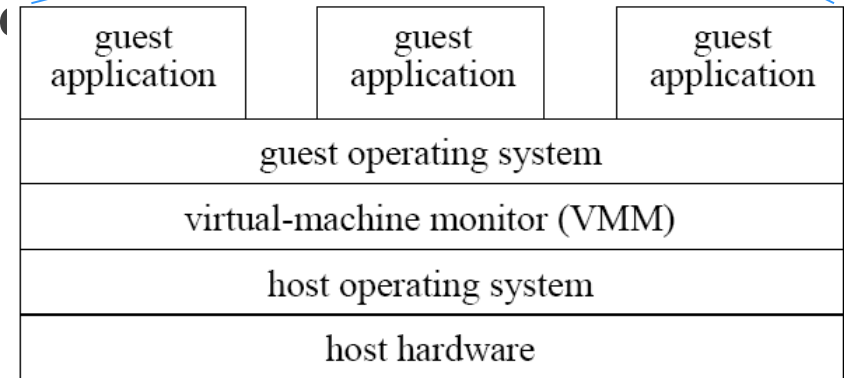
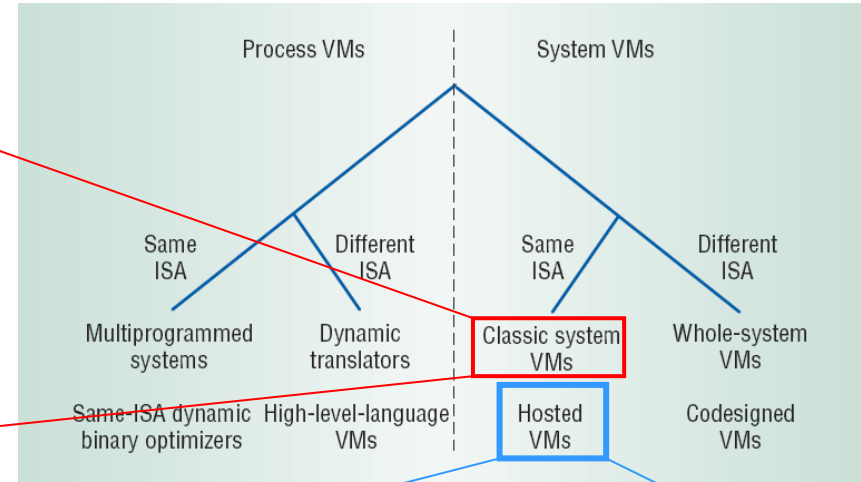


System VMMs



Type 1

- **Type 1: runs directly on hardware**
 - High performance
 - e.g., Xen, VMware ESX Server
- **Type 2: runs on Host OS**
 - Ease of construction/installation
 - e.g., VMware Workstation

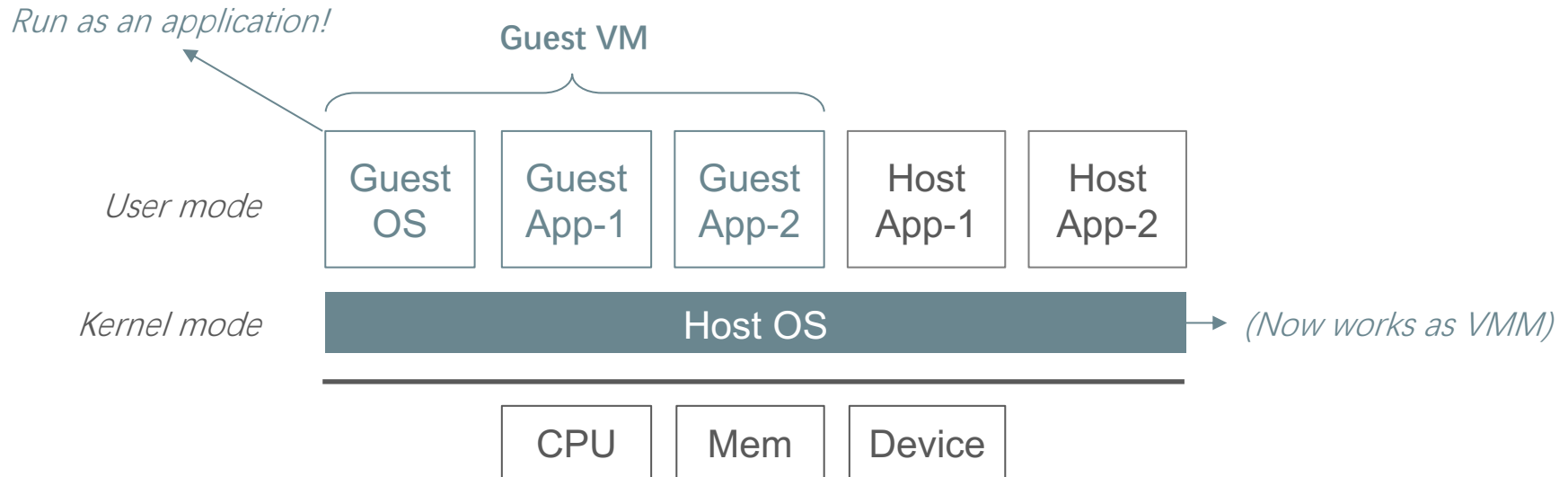


Type 2



CPU VIRTUALIZATION

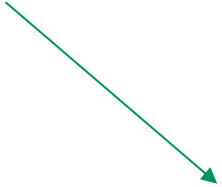
First Try: OS on OS



Run OS as an Application

- **Stuck at the very first instruction**

- `c/i`: disable interrupt
- It's a privilege instruction
- Cannot run in user mode!



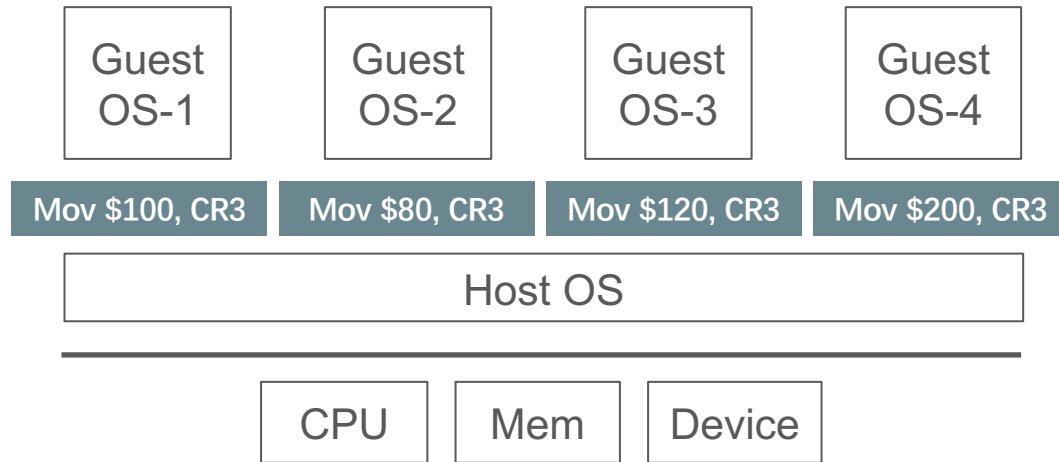
```
9
10 .code16
11 .globl start
12 start:
13     cli
14
```

xv6: bootasm.S

- **Similar instructions**

- E.g., change CR3, set IDT, etc.
- These instructions will change machine states

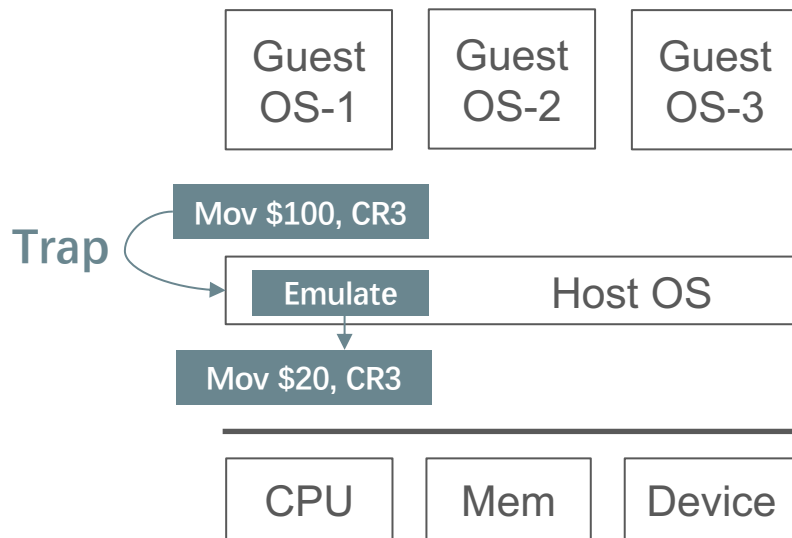
Challenges to Virtualized System ISA



- **Guest OS executes privilege instructions**
- **Guest OS are not allowed to run the privilege instructions**

Solution: Trap & Emulate

- **Trap**: running privilege instructions will trap to the VMM
- **Emulate**: those instructions are implemented as functions in the VMM
- System states are kept in VMM's memory, and are changed according



Now We Can Run OS as an Application

- **Host OS virtualizes system states**
 - Save guest system states in memory
- **When a guest OS runs `cli`**
 - CPU will trap to the host OS (`trap`)
 - Host OS will change the IF bit (`emulate`)
- **How does a host OS deliver interrupt to a guest OS?**
 - Similar as delivering a `signal` to an application
- **Problem solved? No...**

```
9
10 .code16
11 .globl start
12 start:
13     cli
14
```

xv6: bootasm.S

Problems of Trap & Emulate

- Not all architectures are "strictly virtualizable"
- An ISA is strictly **virtualizable** if, when executed in a lesser privileged mode:
 - All instructions that access privileged state trap
 - All instructions either trap or execute identically
- Trap costs may be high

X86 is not Strictly Virtualizable

- **Example: the *popf* instruction**
 - *popf* takes a word off the stack and puts it into the flags register
 - One flag in that register is the interrupt enable flag (IF)
 - At system level the IF flag is updated by *popf*
 - At user level the IF flag is *not* updated, and CPU **silently** drops updates to the IF
- **There **17** such instructions in X86**
 - SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT n, RET, STR, MOV

How to Deal with the 17 Instructions?

1. **Instruction Interpretation:** emulate them by software
2. **Binary translation:** translate them to other instructions
3. **Para-virtualization:** replace them in the source code
4. **New hardware:** change the CPU to fix the behavior

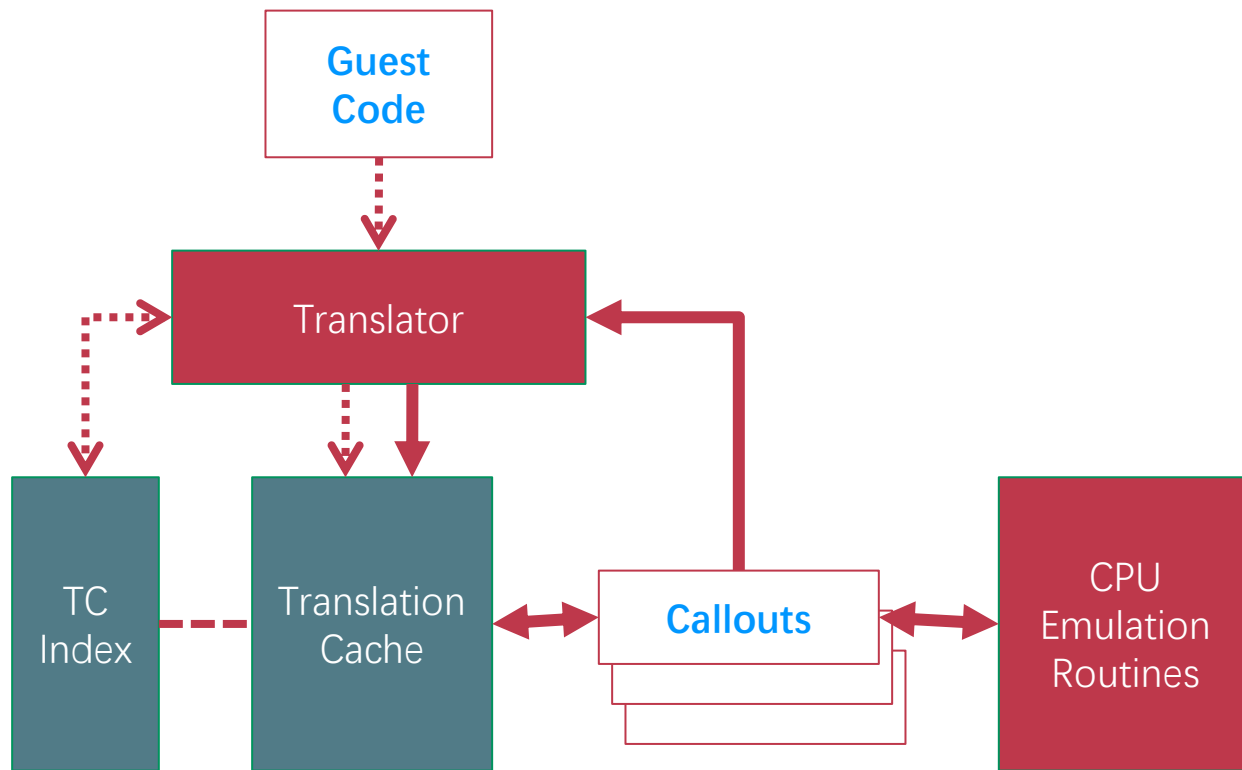
Sol-1: Instruction Interpretation

- Emulate Fetch/Decode/Execute pipeline [in software](#)
 - Emulate all the system status using memory
 - E.g., using an array `GPR[8]` for general purpose registers
 - None guest instruction executes directly on hardware
- E.g., Bochs

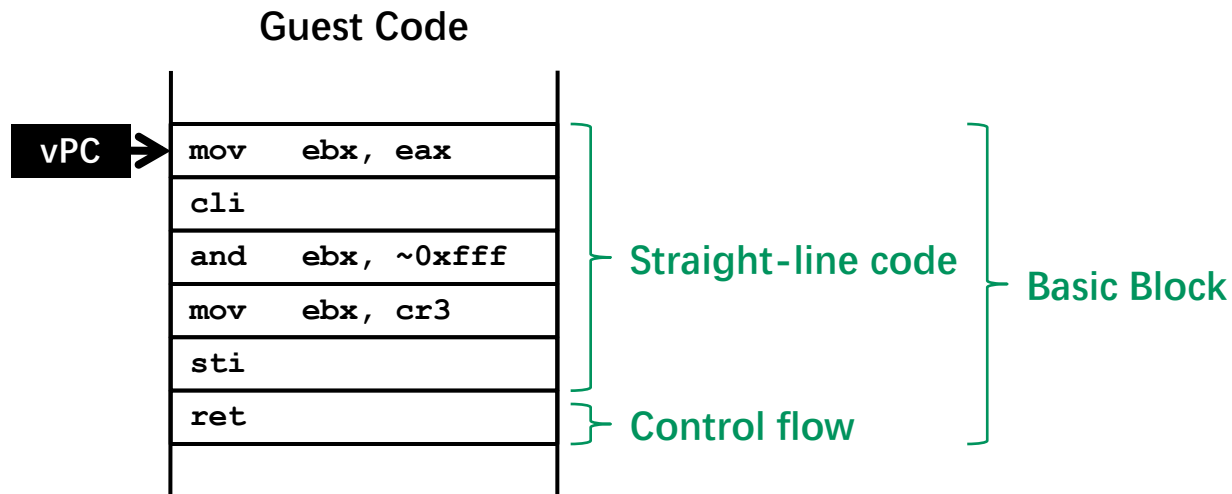
Sol-2: Binary Translator

- **Translate before execution**
 - Translation unit is basic block (why?)
 - Each basic block -> code cache
 - Translate the 17 instructions to function calls
 - Implemented by the VMM
- **E.g., VMware, Qemu**

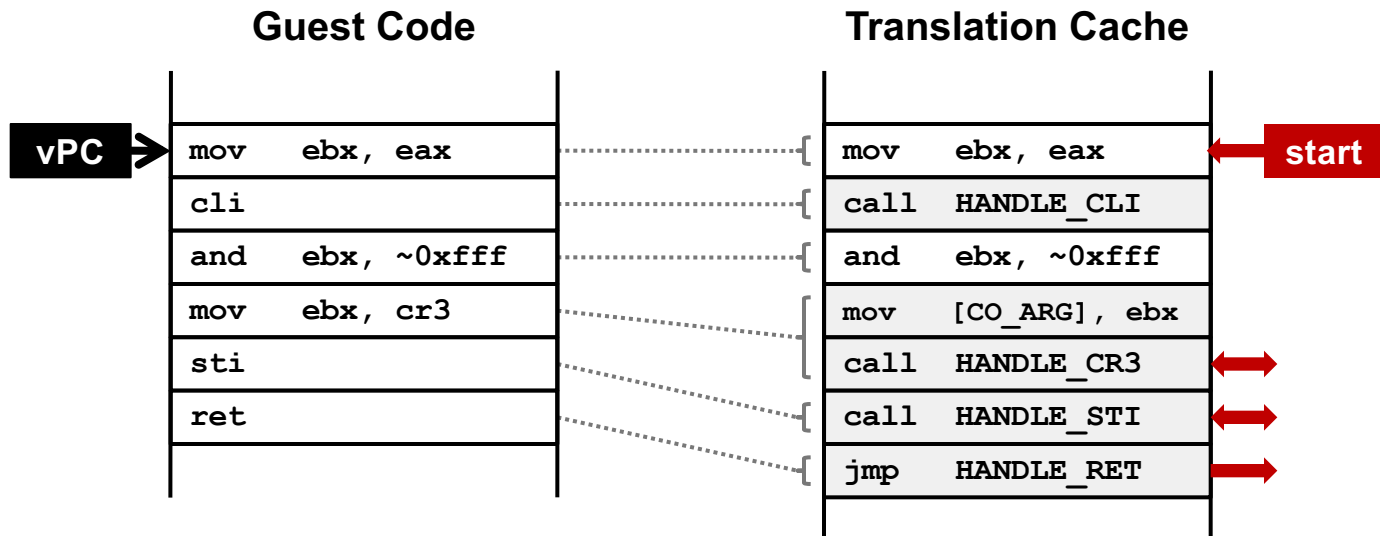
Architecture of a Binary Translator



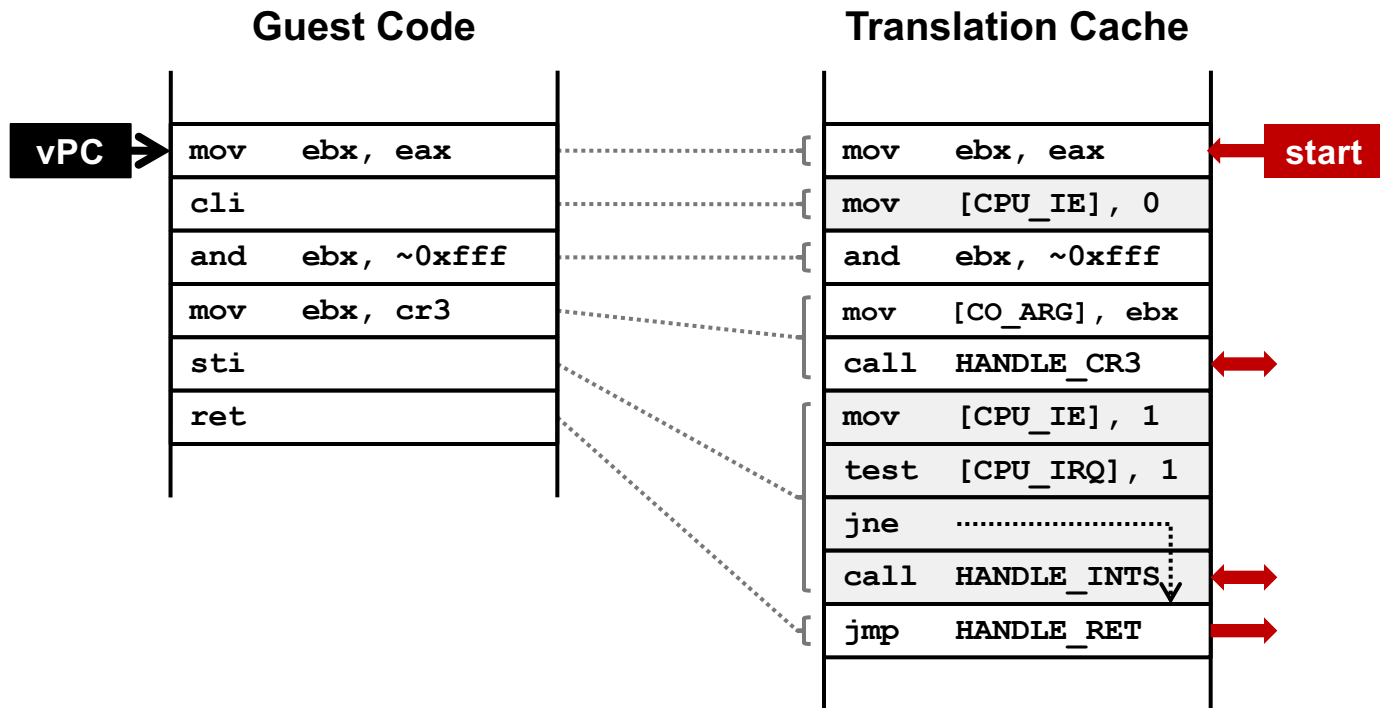
Basic Blocks



Binary Translation



Binary Translation



Basic Binary Translator

```
void BT_Run(void)
{
    CPUState.PC = _start;
    BT_Continue();
}

void BT_Continue(void)
{
    void *tcpc;

    tcpc = BTFindBB(CPUState.PC);

    if (!tcpc) {
        tcpc = BTTranslate(CPUState.PC);
    }

    RestoreRegsAndJump(tcpc);
}
```

```
void *BTTranslate(uint32 pc)
{
    void *start = TCTop;
    uint32 TCPC = pc;

    while (1) {
        inst = Fetch(TCPC);
        TCPC += 4;

        if (IsPrivileged(inst)) {
            EmitCallout();
        } else if (IsControlFlow(inst)) {
            EmitEndBB();
            break;
        } else {
            /* ident translation */
            EmitInst(inst);
        }
    }

    return start;
}
```

Basic Binary Translator – Part 2

```
void BT_CalloutSTI (BTSavedRegs regs)
{
    CPUState.PC = BTFindPC(regs.tcpc);
    CPUState.GPR[] = regs.GPR[];

    CPU_STI();

    CPUState.PC += 4;

    if (CPUState.IRQ
        && CPUState.IE) {
        CPUVector();
        BT_Continue();
        /* NOT_REACHED */
    }

    return;
}
```

Issues with Binary Translation

- **PC synchronization on interrupts**
 - Now interrupt will only happen at basic block boundary
 - But on real machine, interrupt may happen at any instruction
- **Carefully handle self-modifying code (SMC)**
 - Notified on writes to translated guest code

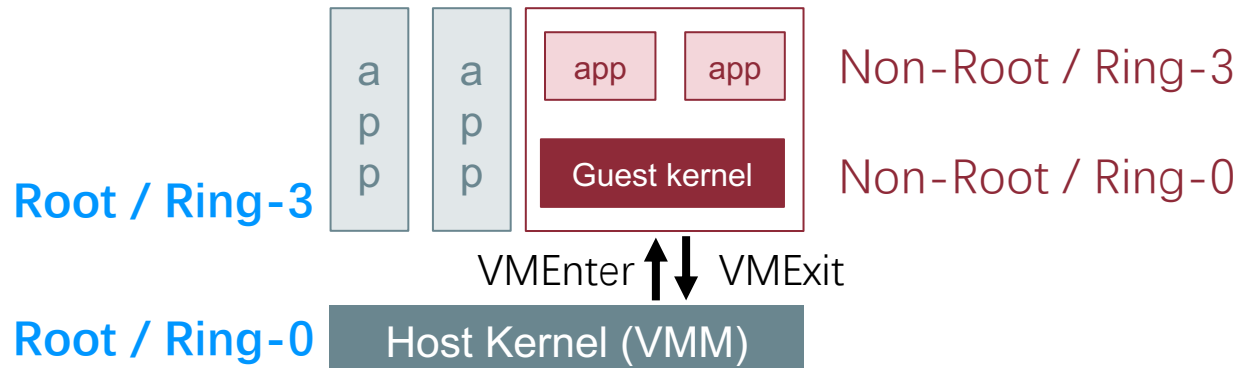
Sol-3: Para-virtualization

- **Modify OS and let it cooperate with the VMM**
 - Change sensitive instructions to calls to the VMM
 - Also known as **hypercall**
 - Hypercall can be seen as trap
- **E.g., Xen**
 - Widely used by industry like Amazon's EC2

Sol-4: Hardware Supported CPU Virtualization

- **VMX root operation:**
 - Full privileged, intended for Virtual Machine Monitor
- **VMX non-root operation:**
 - Not fully privileged, intended for guest software

Both forms of operation support all four privilege levels from 0 to 3

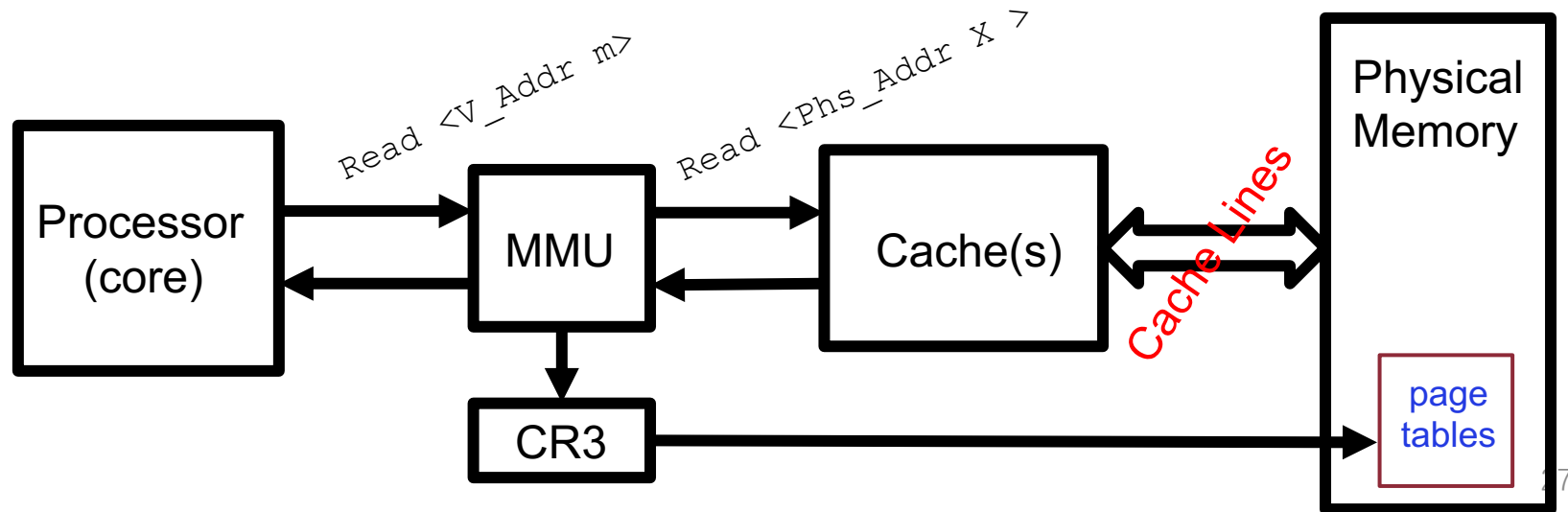




MEMORY VIRTUALIZATION

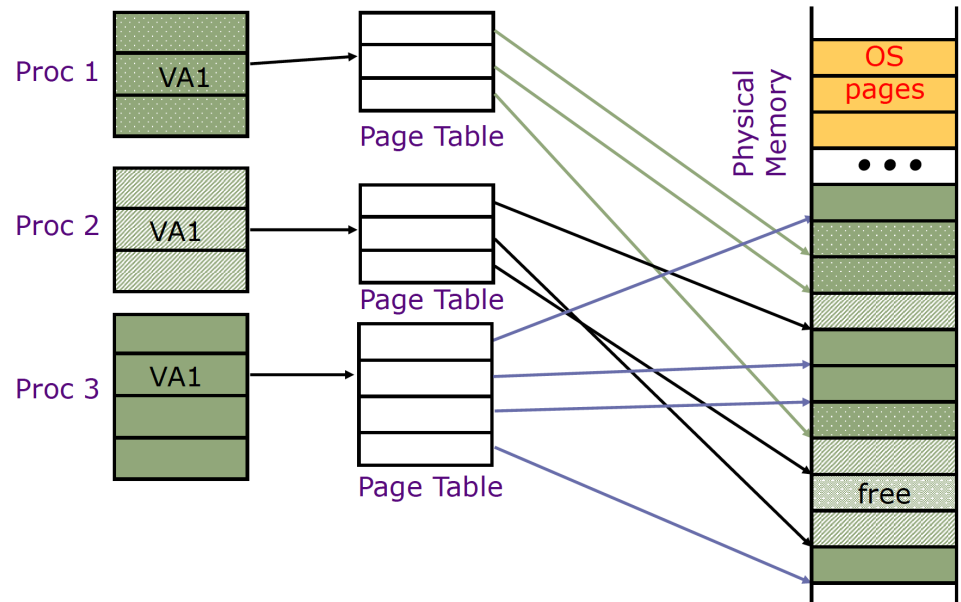
Memory Management Unit (MMU)

- OS configures page tables for processes
- Install a page table by modifying CR3 register
- MMU checks a page table when translating virtual addresses



Page Table

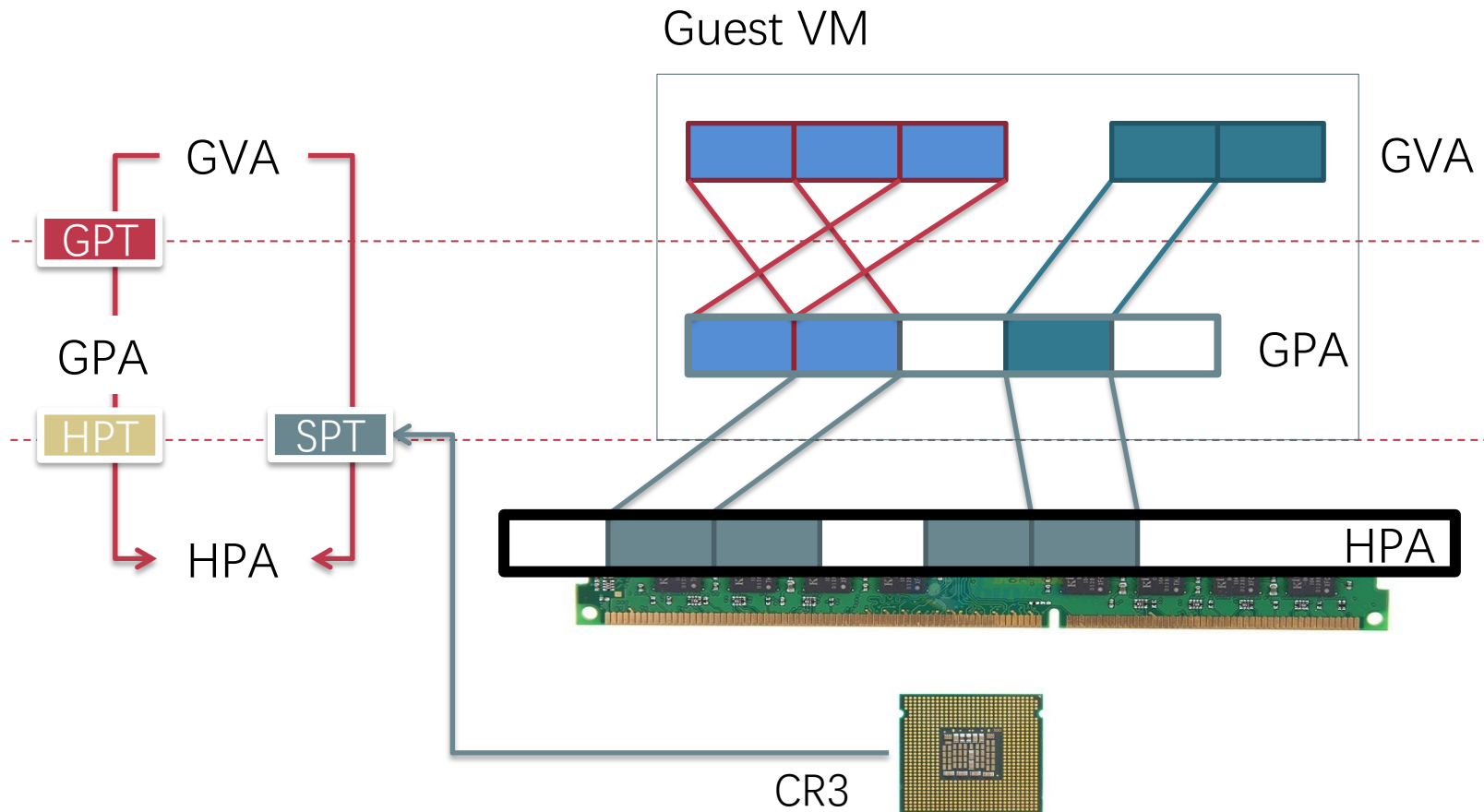
- Configured by the OS kernel
- Each process has its own page table
- Each page has different permission
 - Read
 - Write
 - Execution



Virtualizing the Page Tables

- **Terminology: 3 types of address now**
 - **GVA->GPA->HPA** (Guest virtual. Guest physical. Host physical)
 - Guest VM's page table contains GPA
- **Setting CR3 to point to guest page table would not work**
 - E.g., a processes in VM might access host physical address 0~1GB, which might not belong to that guest VM
 - Traditional solutions: **shadow paging & direct paging**
 - Today's solution: **new hardware**

Solution-1: Shadow Paging



Two Page Tables Become One

1. VMM intercepts guest OS setting the virtual CR3
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest physical address is translated into host physical address
4. Finally, VMM loads the host physical address of the shadow page table

Setup Shadow Page Table

```
set_cr3 (guest_page_table):  
    for GVA in 0 to 220  
        if guest_page_table[GVA] & PTE_P:  
            GPA = guest_page_table[GVA] >> 12  
            HPA = host_page_table[GPA] >> 12  
            shadow_page_table[GVA] = (HPA<<12)|PTE_P  
        else  
            shadow_page_table = 0  
    CR3 = PHYSICAL_ADDR(shadow_page_table)
```


Question

- **Assume that:**
 - There are 10 VMs running on a machine
 - Each VM contains 10 applications
- **Q: how many shadow page tables in total?**
 - Shadow page tables are per application
 - Guest page tables are per application
 - Host page tables are per VM

Hardware Supported Memory Virtualization

- **Hardware implementation**
 - Intel's EPT (Extended Page Table)
 - AMD's NPT (Nested Page Table)
- **Another table**
 - EPT for translation from **GPA to HPA**
 - EPT is controlled by the hypervisor
 - EPT is per-VM

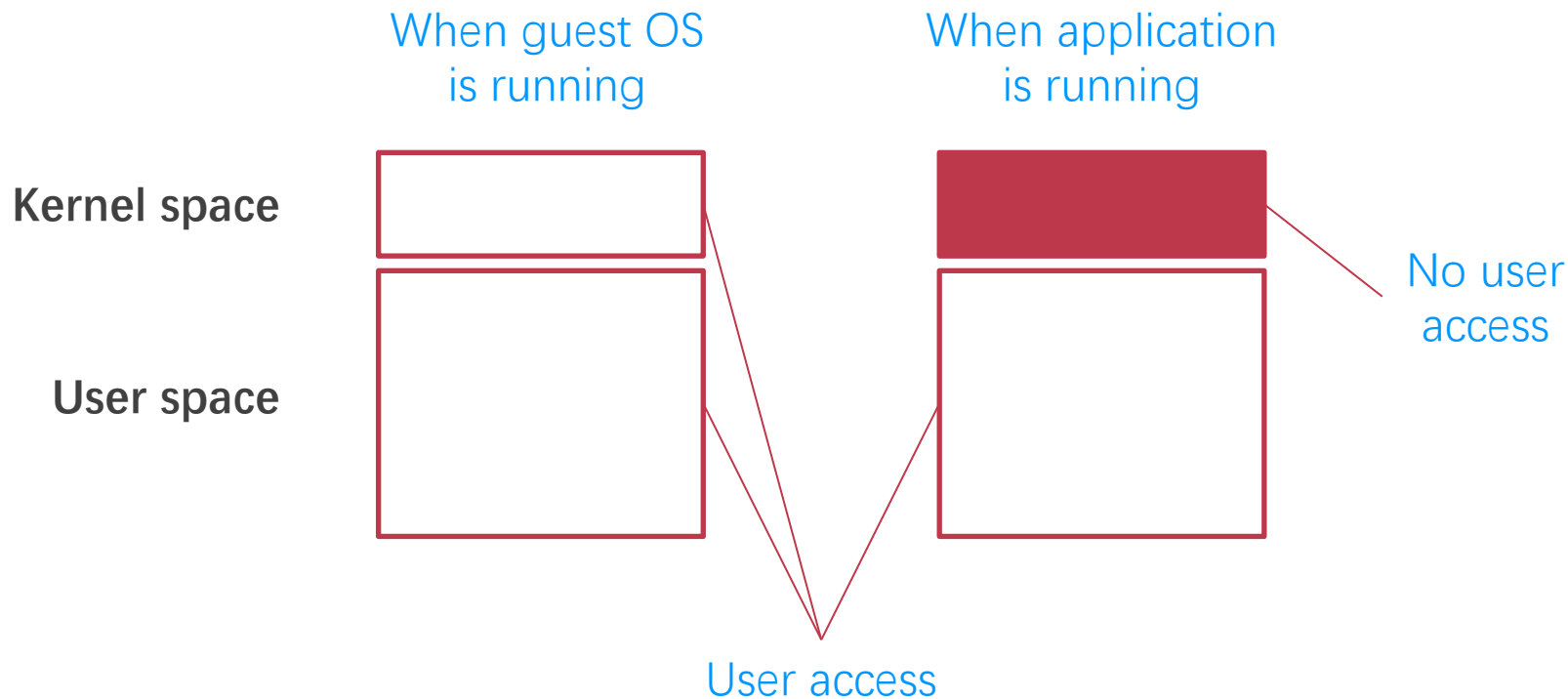
What if Guest OS Modifies its Own Page Table?

- **It will have no effect!**
 - Since CR3 is now pointing to the shadow page table
 - Need to synchronize the shadow page table with guest page table
- **Solution**
 - **VMM needs to intercept when guest OS modifies page table, and updates the shadow page table accordingly**
 1. Mark the guest table pages as read-only (in the shadow page table)
 2. If guest OS tries to modify its page tables, it triggers page fault
 3. VMM handles the page fault by updating shadow page table

What if a Guest App Access its Kernel Memory?

- **Remember that now the kernel is running in user mode**
 - It means any application may also access guest kernel's memory
 - How do we selectively allow / deny access to kernel-only pages?
- **One solution: split a shadow page table to two tables**
 - Two shadow page tables, one for user, one for kernel
 - When guest OS switches to user mode, VMM will switch the shadow page table as well, vice versa
 - Recall trap & emulate

Two Memory Views of Guest VM



▶ The Same Question

- **Assume that:**
 - There are 10 VMs running on a machine
 - Each VM contains 10 applications
- **Q: now, how many shadow page tables in total?**

Sol-2: Direct Paging (Para-virtualization)

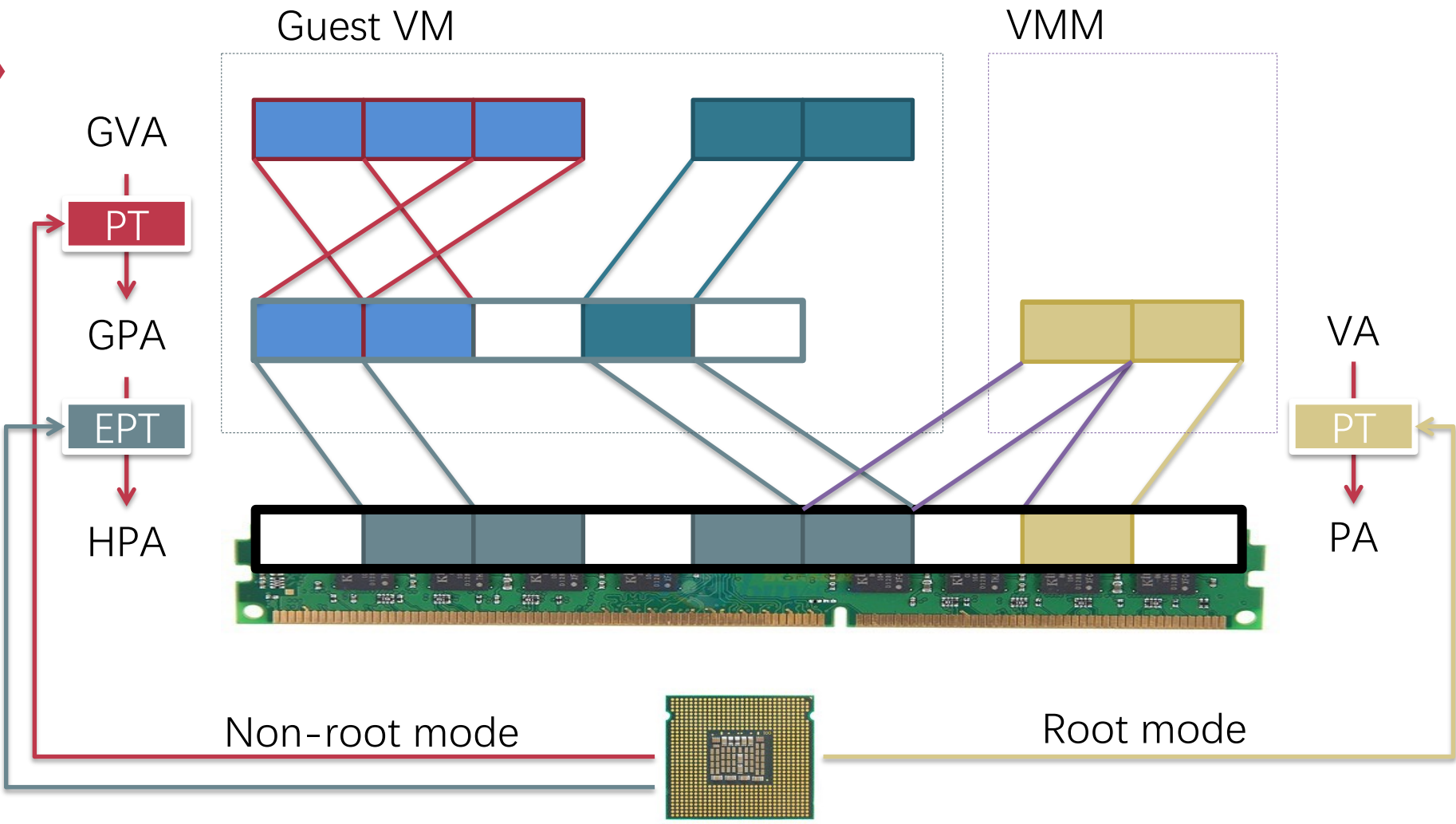
- **Modify the guest OS**
 - No GPA is needed, just GVA and HPA
 - Guest OS directly manages its HPA space
 - Use hypercall to let the VMM update the page table
 - The hardware CR3 will point to guest page table
- **VMM will check all the page table operations**
 - The guest page tables are read-only to the guest

Sol-2: Direct Paging (Para-virtualization)

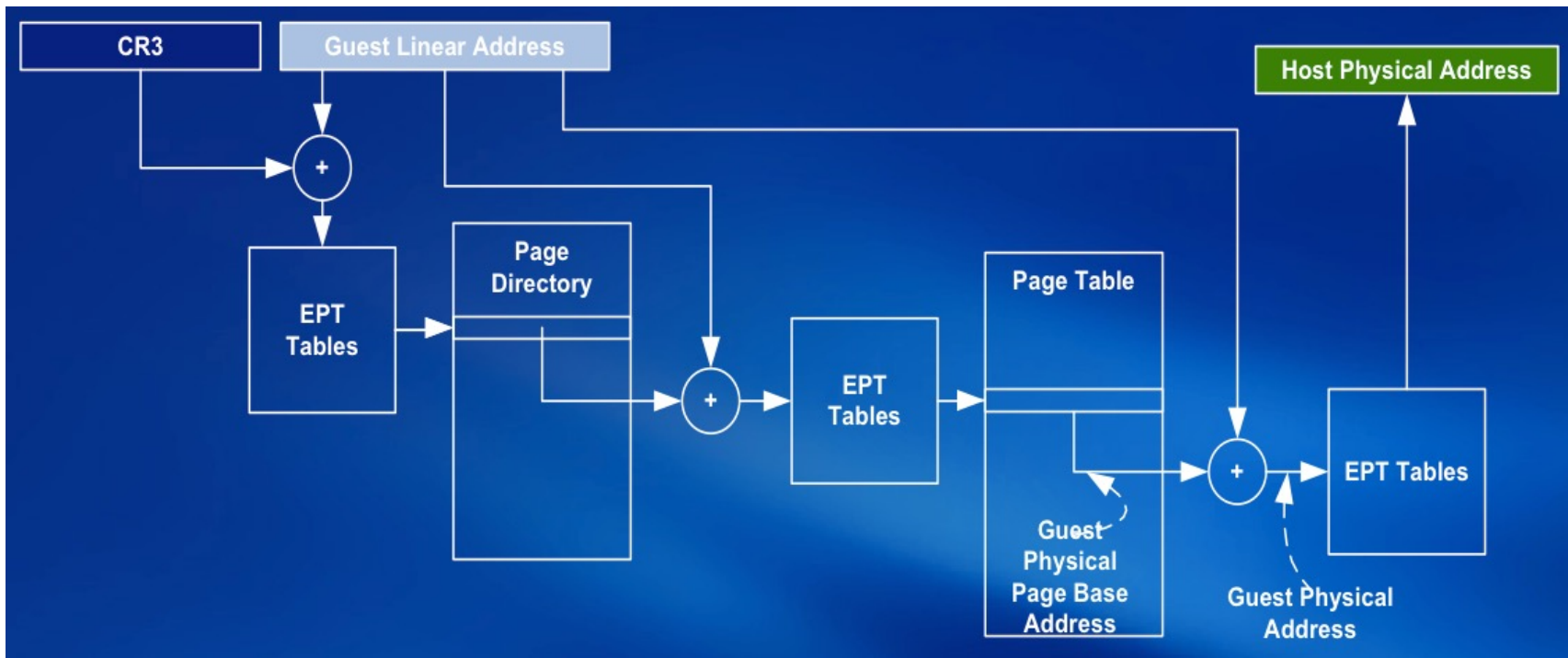
- **Positive**
 - Easy to implement and more clear architecture
 - Better performance: guest can batch to reduce trap
- **Negatives**
 - Not transparent to the guest OS
 - The guest now knows much info, e.g., HPA
 - May use such info to trigger *rowhammer* attacks

Sol-3: Hardware Supported Memory Virtualization

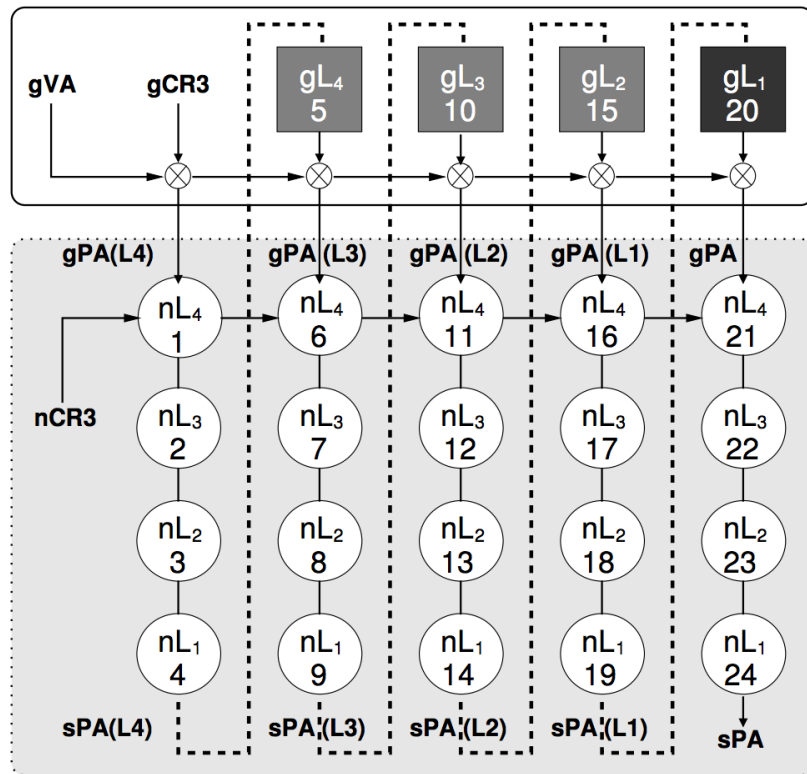
- **Hardware implementation**
 - Intel's EPT (Extended Page Table)
 - AMD's NPT (Nested Page Table)
- **Another table**
 - EPT for translation from **GPA to HPA**
 - EPT is controlled by the hypervisor
 - EPT is per-VM



EPT Translation: Details



EPT Increases Memory Access



$sPA == HPA$
 $nCR3 == hCR3$
 $nL_1 == hL_1$

One memory access from the guest VM may lead up to **20 memory accesses**!



I/O VIRTUALIZATION

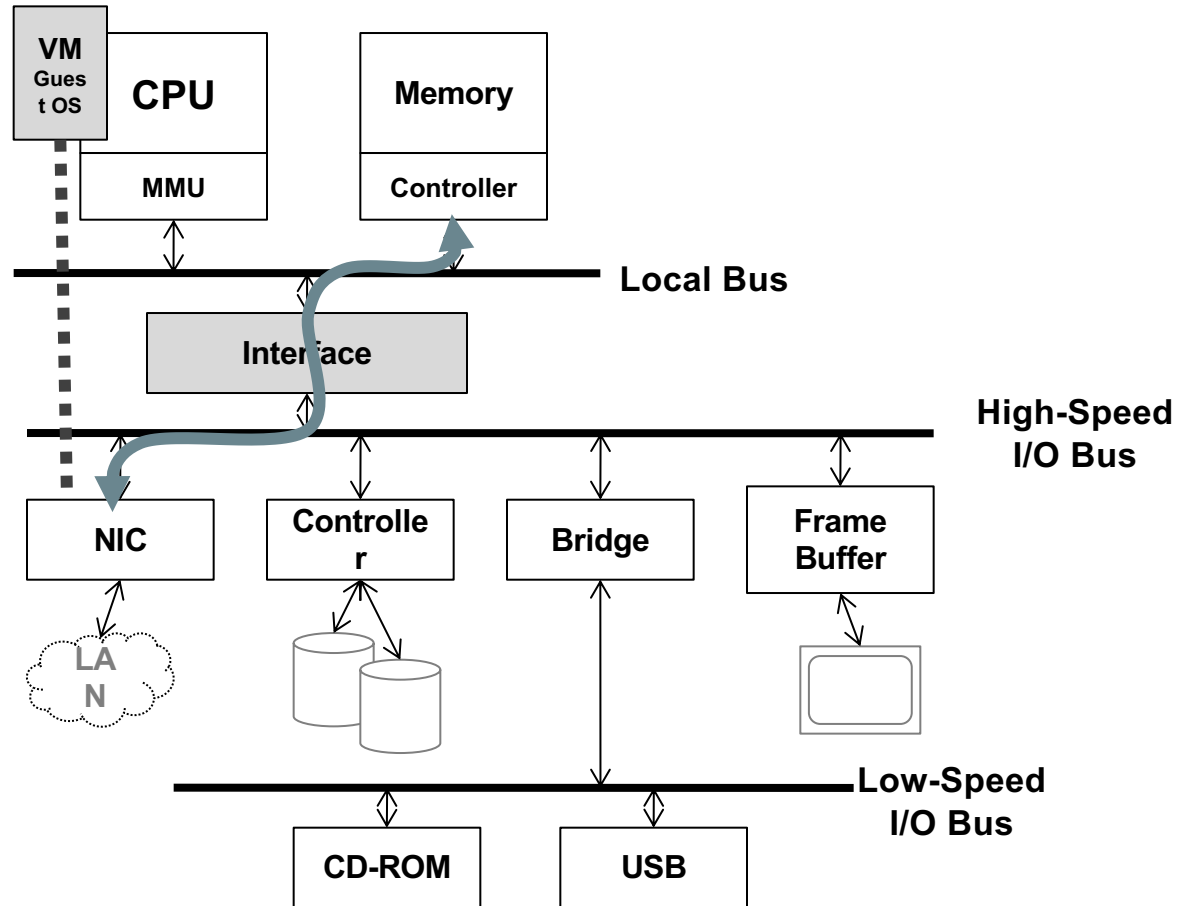
I/O Virtualization

- **Goal**
 - Multiplexing device to guest VMs
- **Challenges**
 - Each guest OS has its own driver
 - How can one device be controlled by multiple drivers?
 - What if one guest OS tries to format its disk?

Solutions for I/O Virtualization

1. **Direct access:** VM owns a device exclusively
2. **Device emulation:** VMM emulates device in software
3. **Para-virtualized:** split the drivers to guest and host
4. **Hardware assisted:** self-virtualization device

Sol-1: Direct Access Device Virtualization

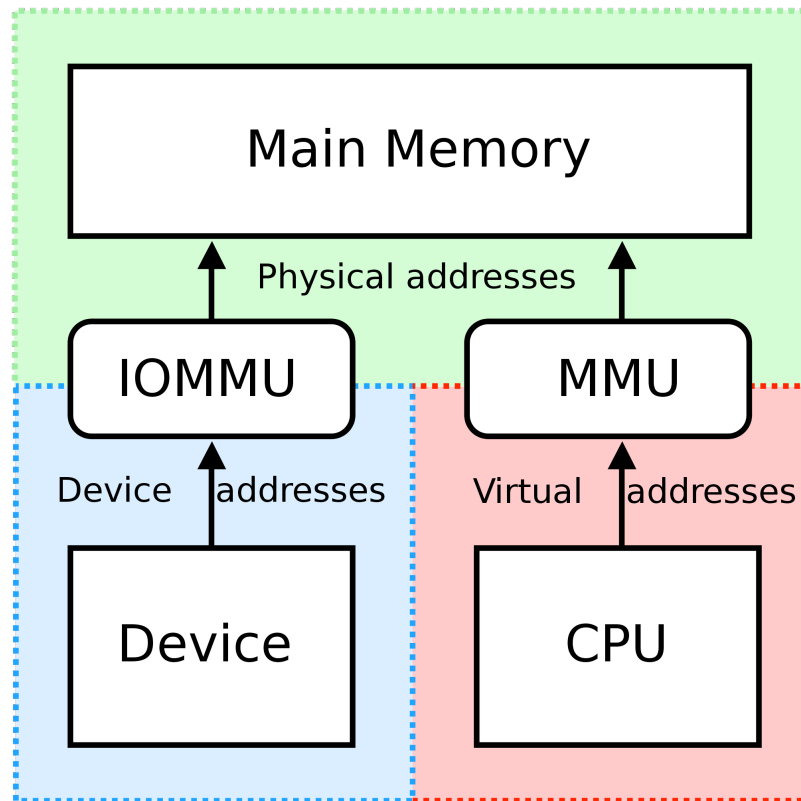


IOMMU: Page Tables for Devices

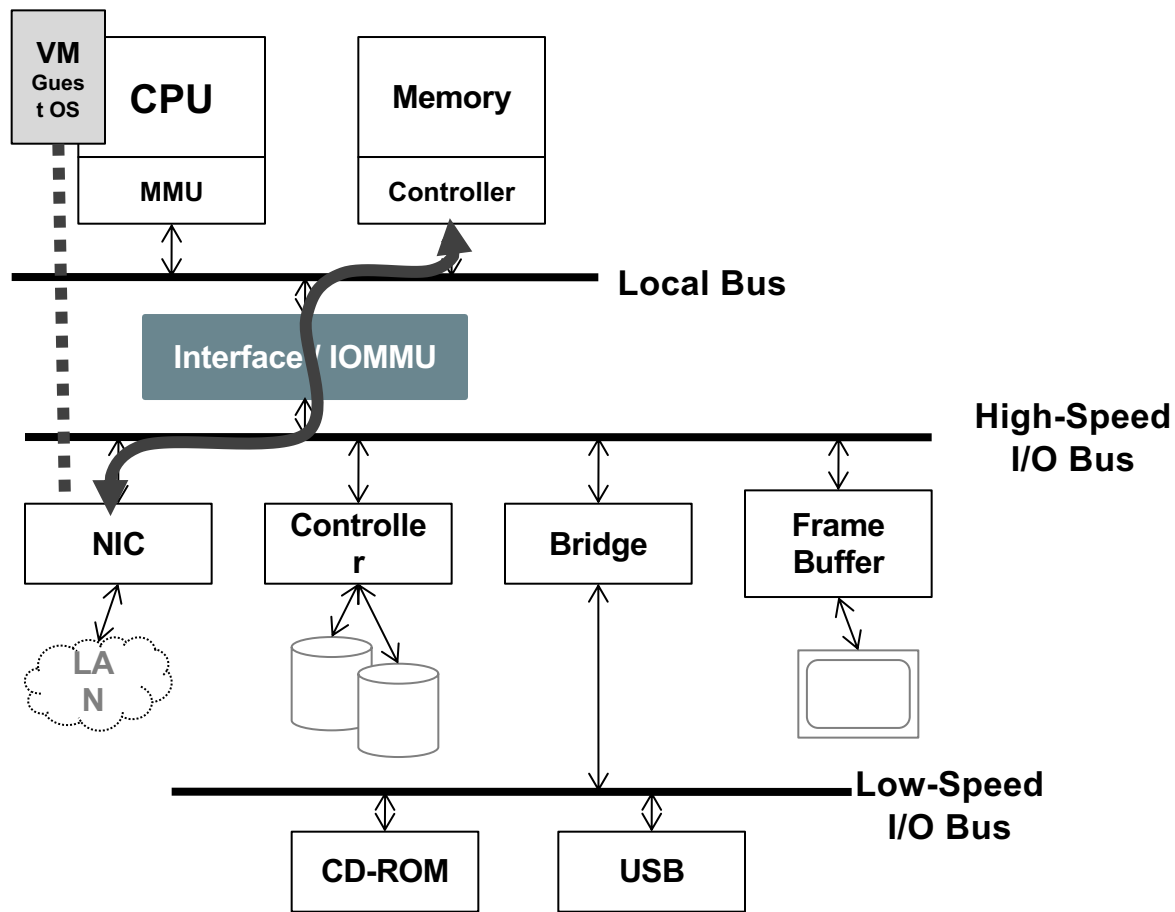
- **Allow guest OS direct access to underlying device**
 - Guest just reuses its own device driver
- **Q: What if a VM asks device to access memory of other VMs?**
 - It is possible because device accesses HPA in DMA
 - Thus a device can access any memory
- **Solution: Page tables for devices**
 - Another MMU: IOMMU for devices
 - Q: what addresses will IOMMU translate?

IOMMU for Direct Access

- VT-d architecture defines a multi-level page-table structure for DMA address translation
- Similar to IA-32 processor page-tables, enabling software to manage memory at 4 KB or larger page granularity



Direct Access Device Virtualization



Direct Access Device Virtualization

- **Positives**

- Fast, since the VM uses device just as native machine
- Simplify monitor: limited device drivers needed

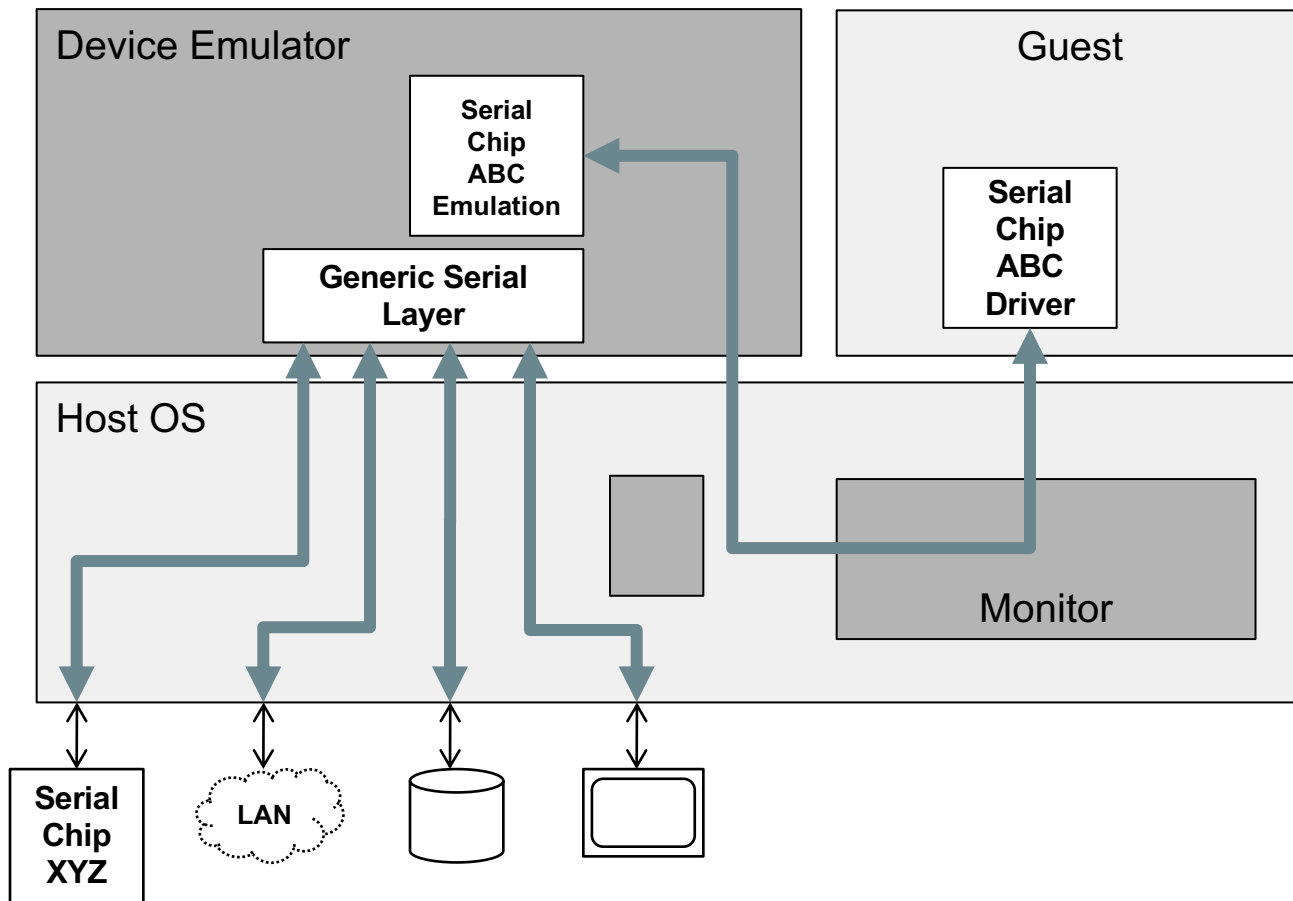
- **Negatives**

- Hardware interface visible to guest (bad for migration)
- Interposition is hard by definition (no way to trap & emulate)
- Now you need much more devices! (image 100 VMs)

Sol-2: Emulating Devices

- **Emulate a device**
 - Implement device logic in pure software
 - Can even emulate non-existing devices
- **For example**
 - VMware emulates a 8139 network card for VMs
 - Since 8139 is widely used, almost every OS has its driver
 - So no need to write a new driver

Example: Emulating Serial Port



Emulated Devices

- **Positives**

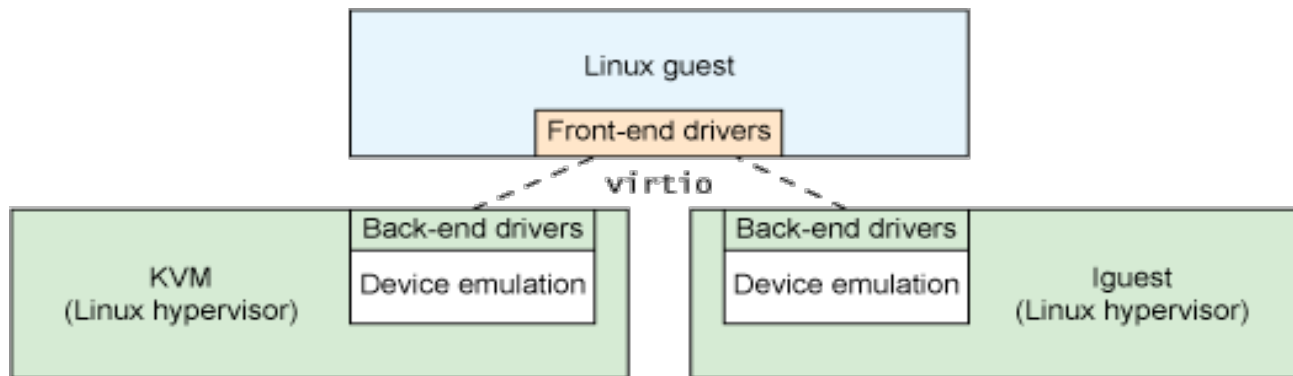
- Platform stability (good for migration)
- Allows interposition
- No special hardware support is needed

- **Negatives**

- Can be slow (it's software emulated)

Sol-3: Para-Virtualized Devices

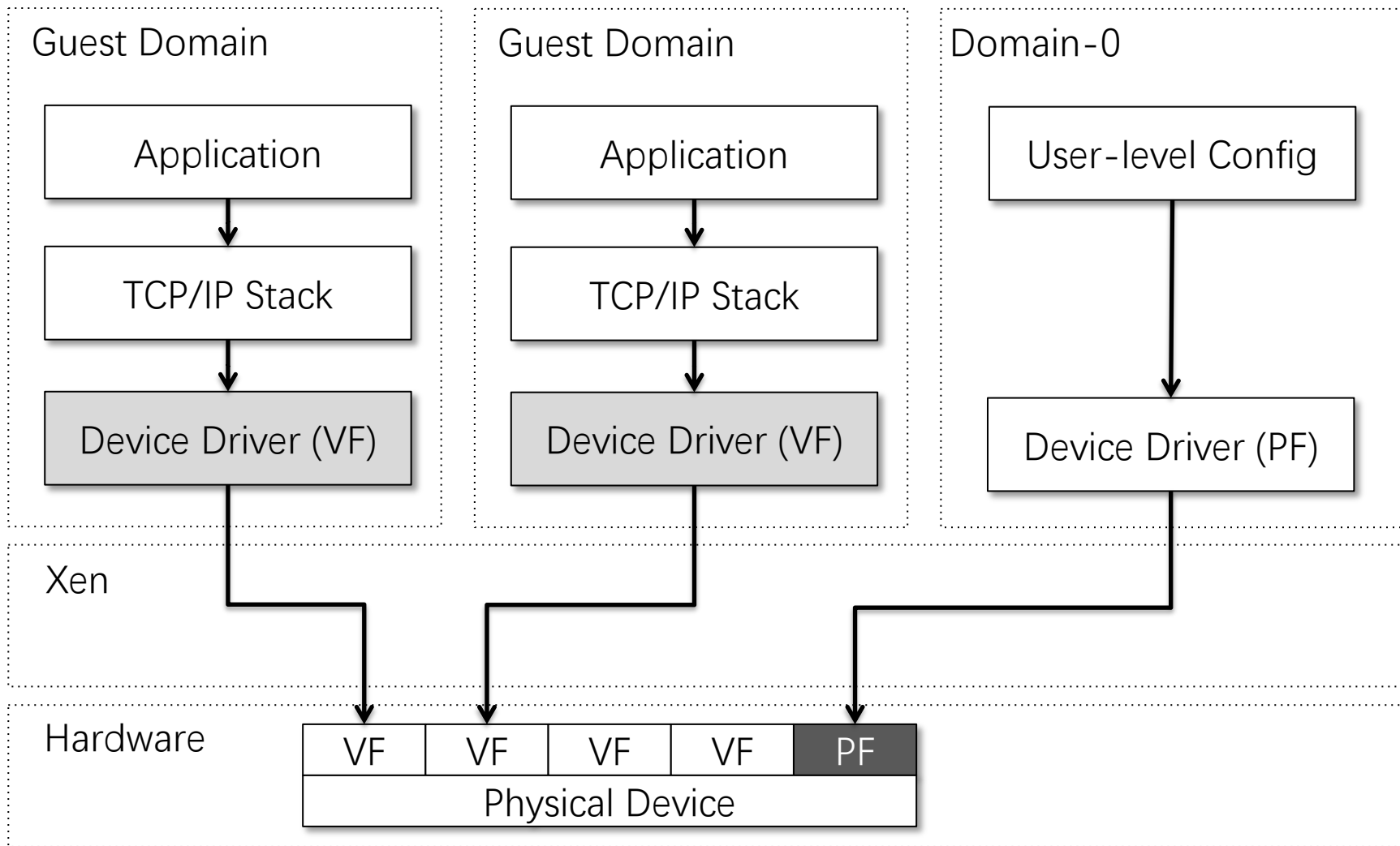
- **VMM offers new types of device**
 - The guest OS will run a new driver (front-end driver)
 - The VMM will run a back-end driver for each front-end
 - The VMM will finally run device driver to drive the device



VirtIO Architecture by KVM

Sol-4: Hardware Support for I/O Virtualization

- **VMM**
 - An SR-IOV-capable device can be configured to appear in the PCI configuration space as multiple functions
- **VM**
 - The VMM assigns one or more VFs to a VM by mapping the actual configuration space of the VFs to the configuration space presented to the virtual machine by the VMM



Intel Virtualization Technology Evolution

Vector 3:
I/O Focus

PCI-SIG

Standards for IO-device sharing:

- Multi-Context I/O Devices
- Endpoint Address Translation Caching
- Under definition in the PCI-SIG* IOVWG

Vector 2:
Platform Focus

VT-d

Hardware support for IO-device virtualization

- Device DMA remapping
- Direct assignment of I/O devices to VMs
- Interrupt Routing and Remapping

Vector 1:
Processor Focus

VT-x

VT-i

Establish foundation
for virtualization in the
IA-32 and
Itanium architectures...

... followed by on-going evolution of support:
Micro-architectural (e.g., lower VM switch times)
Architectural (e.g., Extended Page Tables)

VMM
Software
Evolution

Software-only VMMs

- Binary translation
- Paravirtualization

Simpler
and more Secure
VMM through
foundation
of virtualizable ISAs

Increasingly better CPU and I/O virtualization
performance and functionality as I/O devices
and VMMs exploit infrastructure provided
by VT-x, VT-i, VT-d

Past
No Hardware
Support

Today

VMM software evolution over
time with hardware support

Virtualization Technologies

Virtualization	Software Solution	Hardware Solution
CPU	<ul style="list-style-type: none">• Trap & Emulate• Instruction interpretation• Binary translation	<ul style="list-style-type: none">• VT-x<ul style="list-style-type: none">• Root / non-root mode• VMCS
	<ul style="list-style-type: none">• Para-virtualization: Replace 17 insns	
Memory	<ul style="list-style-type: none">• Shadow page table• Separating page tables for U/K	<ul style="list-style-type: none">• EPT
	<ul style="list-style-type: none">• Para-virtualization: Direct paging	
Device	<ul style="list-style-type: none">• Direct I/O• Device emulation	<ul style="list-style-type: none">• IOMMU• SR-IOV
	<ul style="list-style-type: none">• Para-virtualization: Front-end & back-end driver (e.g., virtio)	



SEMANTIC GAP

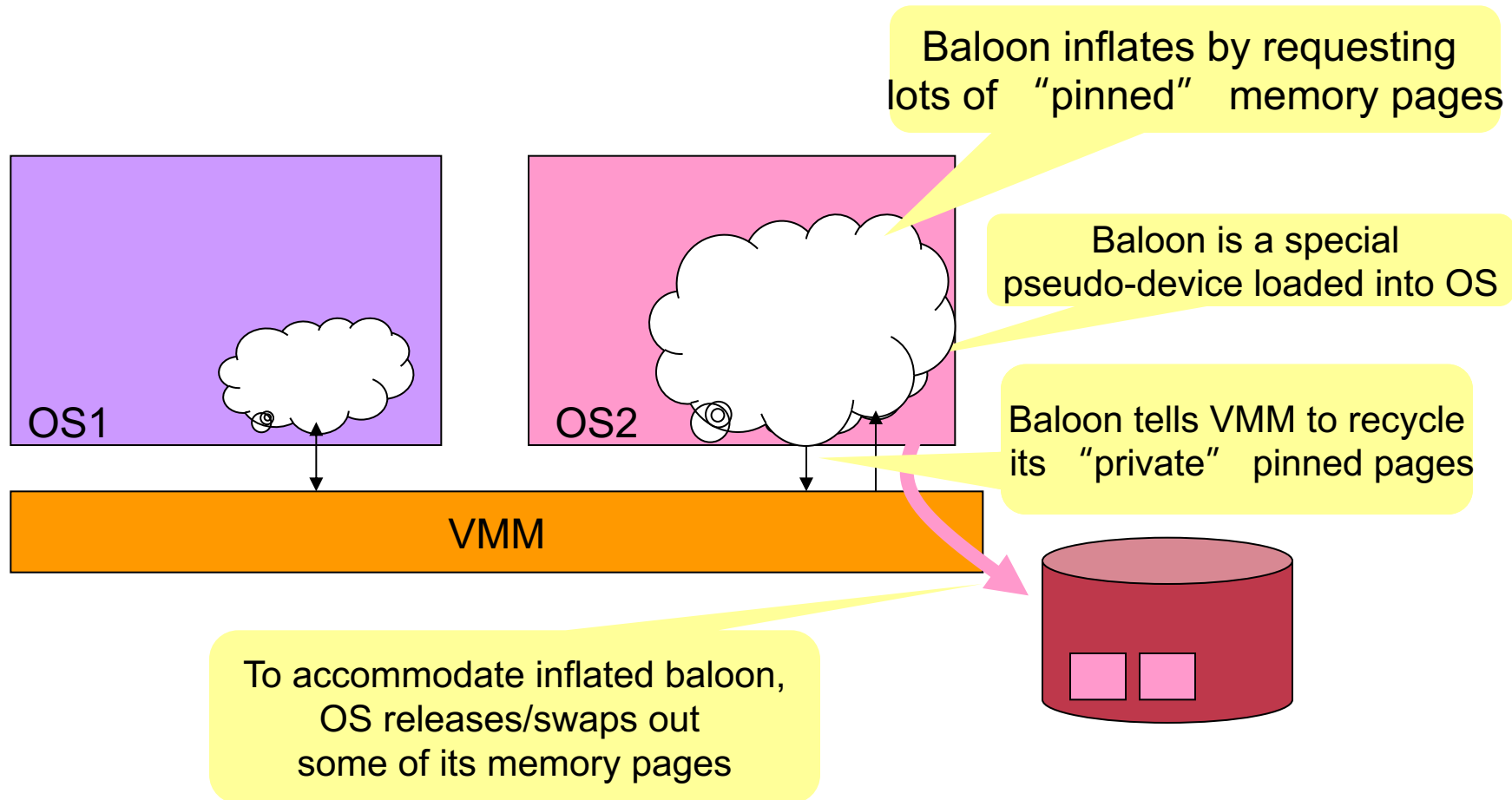
Managing Memory in VMM

- **Configure VMs to use more memory than actually available**
- **What happens when running out of memory?**
- **Strawman: use LRU paging at VMM**
 - OS already uses LRU → doubling paging
 - OS will recycle whatever page VMM just paged out
 - Better to do random eviction

ESX: Reclaiming Pages

- **Idea: trick OS to return memory to VMM**
- **OS is better at deciding what to swap**
 - Normally OS uses all available memory
 - E.g. buffer cache contains old pages, OS won't discard if it does not need memory
- **ESX trick: balloon driver**

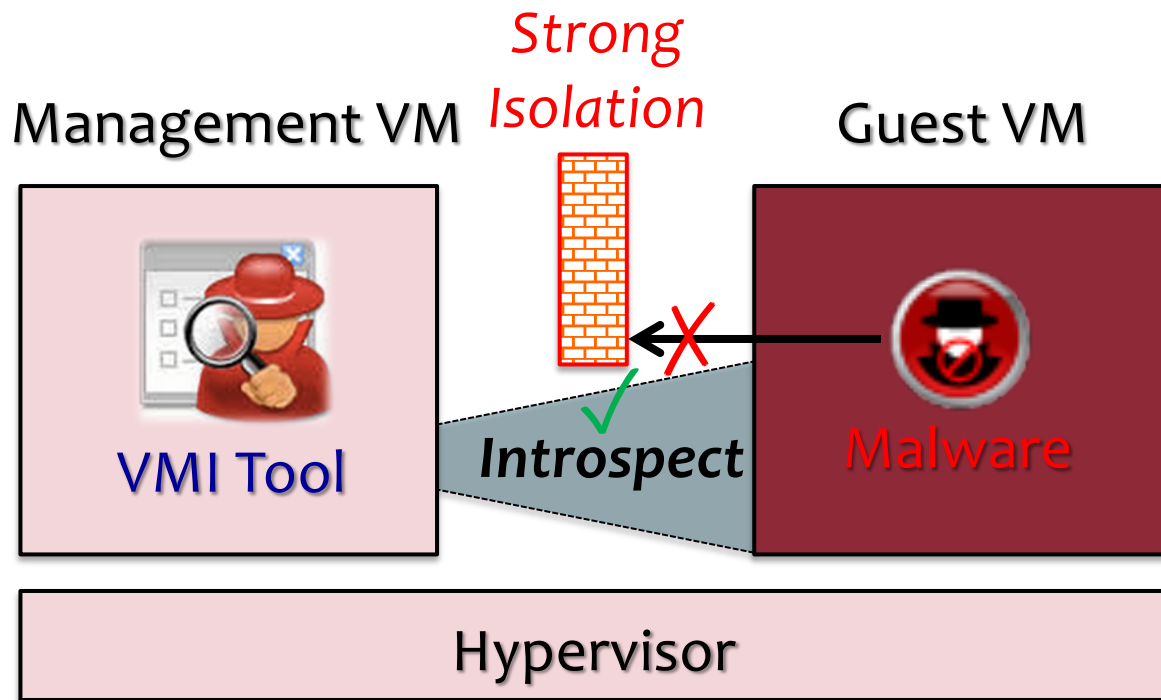
Memory Ballooning



ESX: Sharing Pages across VMs

- **Many VMs run same OS and programs**
 - Many Linux boxes with Apache server
- **Idea: use 1 machine page for identical physical pages**
- **Periodically scan to find identical machine pages**
 - Do copy-on-write to eliminate redundancy
- **Optimization: use a hash table keyed by hash(content)**
 - Allows quick lookup based on page content

Virtual Machine Introspection(VMI)



* T. Garfinkel, M. Rosenblum et al., “A virtual machine introspection based architecture for intrusion detection.” in *Proc. NDSS*, 2003.

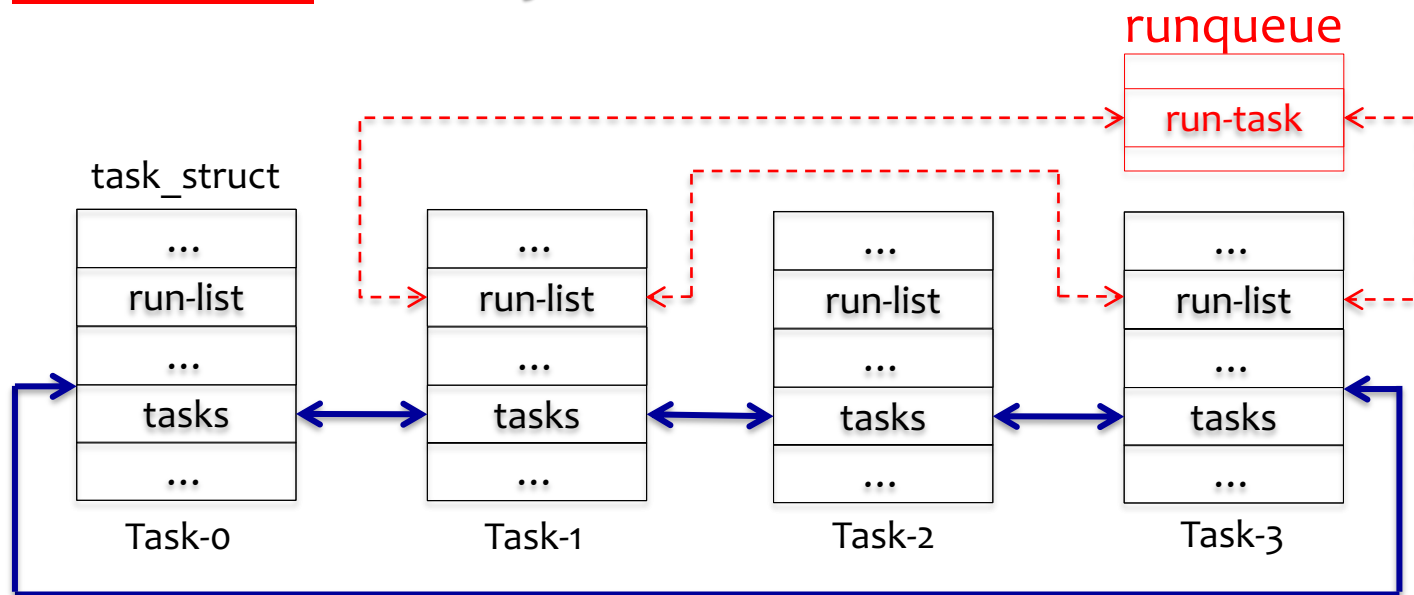
Virtual Machine Introspection

- **Benefits of VMI**
 - Non-intrusive to guest VMs
 - Better security due to strong isolation from vulnerable VMs
- **Widely used for numerous purposes**
 - Intrusion detection, malware analysis
 - Long line of research in architecture, systems and security community
 - Already integrated by VMware and other vendors

Example: Hidden Malware Detection

Task list used by ``ps aux`` to show all the tasks

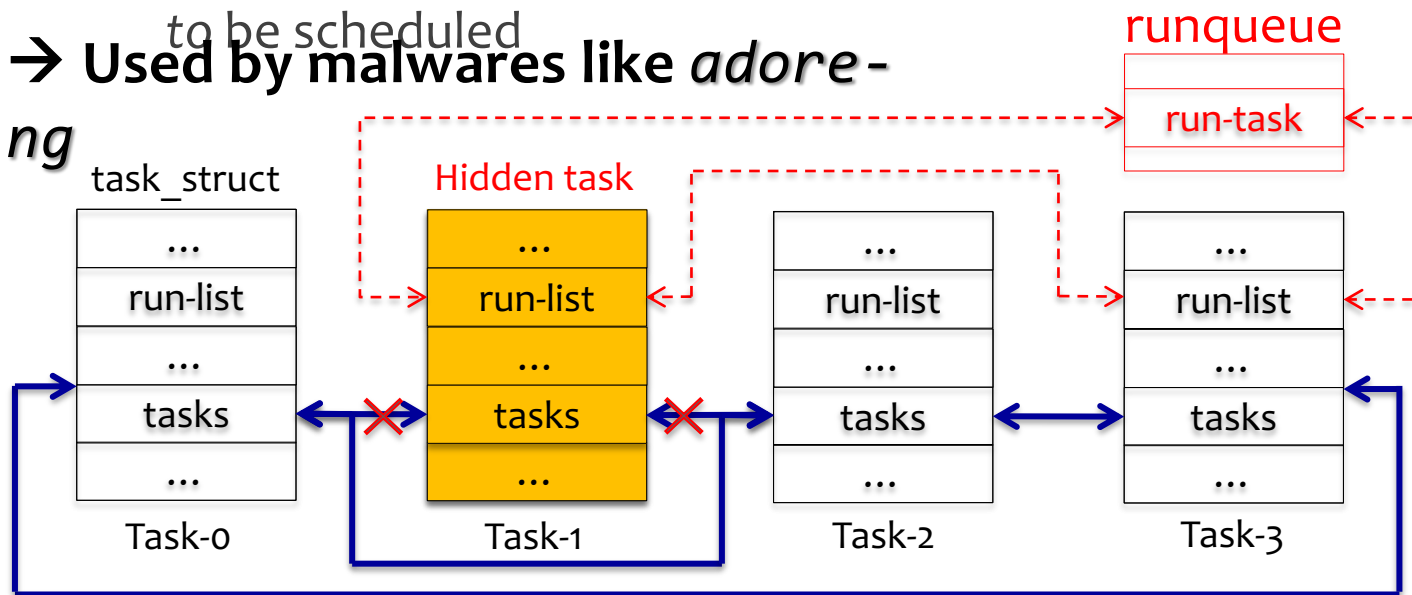
Runqueue used by scheduler to run a task



Example: Hidden Process

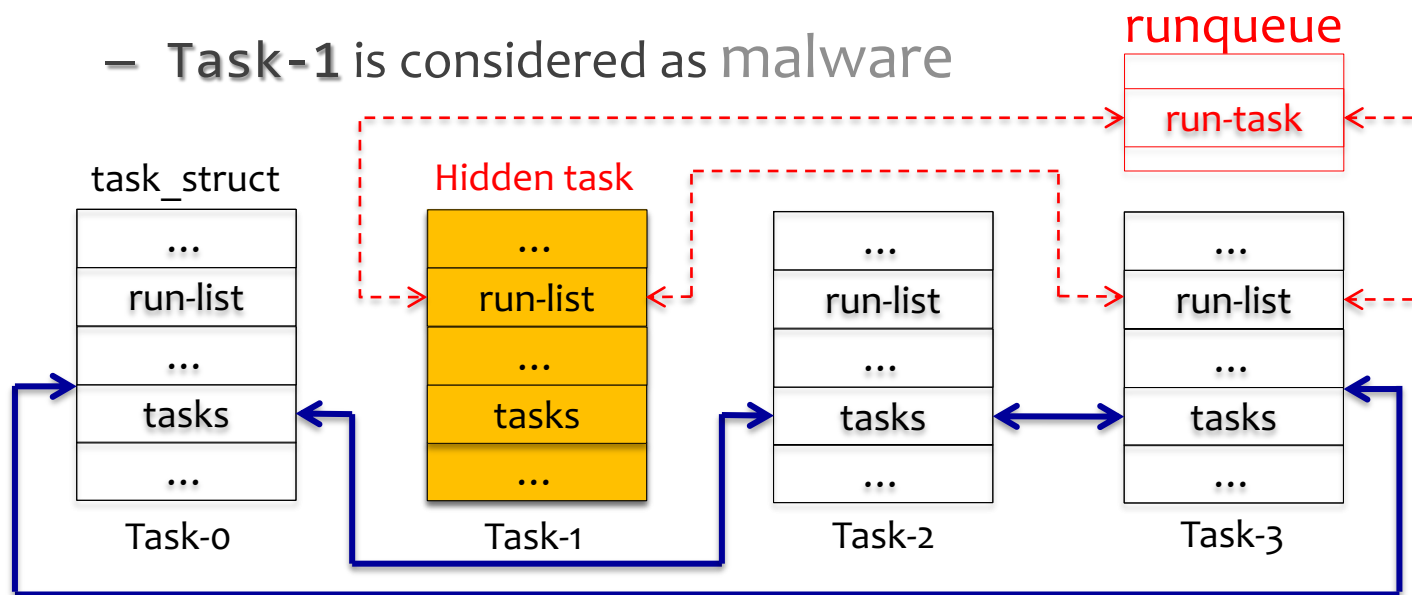
- Malware hides itself from ``ps aux``
 - Remove its own task from the *task list*
 - Invariants: still need to keep the task in *runqueue*

→ ^{to be scheduled} Used by malwares like *adore-ng*



VMI for Hidden Process Detection

- Find tasks in *runqueue* but not in *task list*
 - Run queue: 1, 3
 - Task list: 0, 2, 3
 - Task-1** is considered as malware



Example VMI Code

```
vmi_pause_vm()
```

```
/* traverse the tasks list */
```

```
foreach task in tasks_list:
```

```
    all_procs.add(task)
```

```
/* traverse the runqueue */
```

```
foreach run_task in runqueue:
```

```
    run_procs.add(run_task)
```

```
vmi_resume_vm()
```

```
/* check if there is any hidden running process */
```

```
foreach pid in run_procs:
```

```
    check_if_pid_in_all_procs(pid, all_procs)
```

```
alarm_if_needed()
```

Copy VM data

**Pause VM
to retrieve states**

Do VMI check

Challenges of VMI

- **Timely:** hard to decide when VMI is necessary
- **Disruptive:** Lengthy suspension time
- **Consistency:** May retrieve inconsistent data