IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

系统虚拟化

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

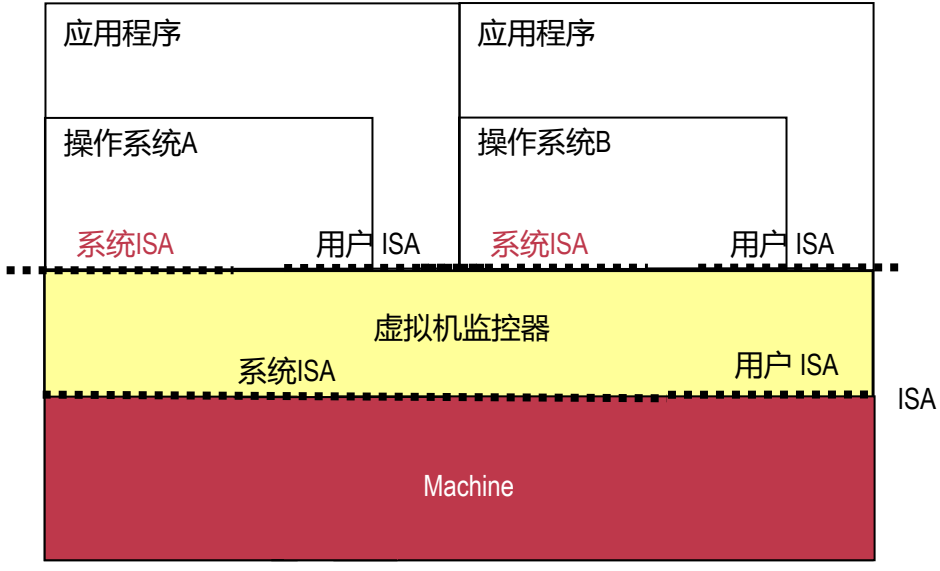
https://ipads.se.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

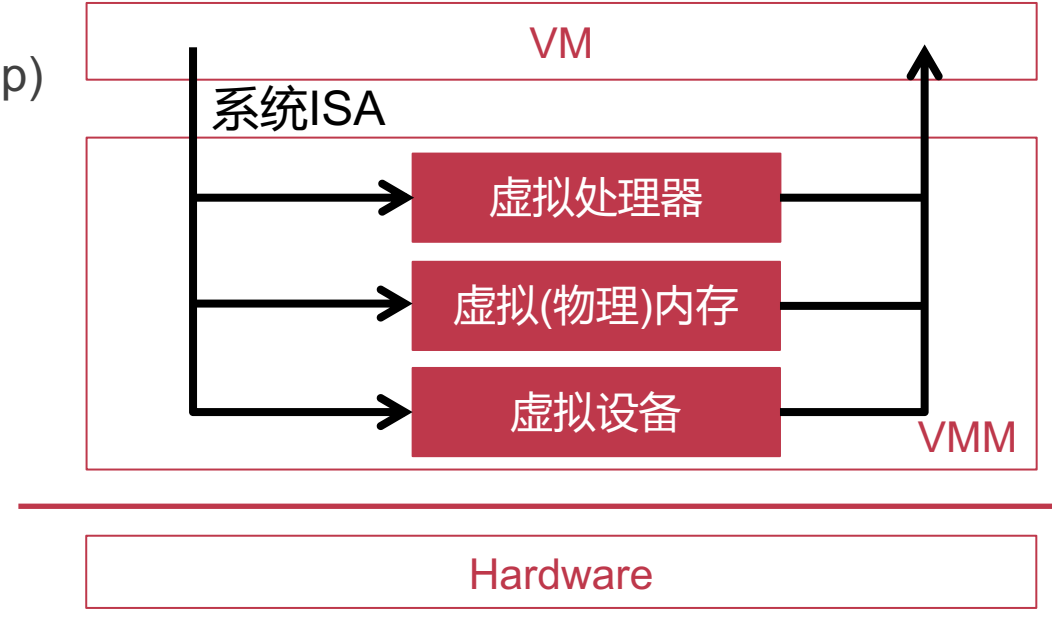
Review: 系统ISA

- **读写敏感寄存器**
  - sctrl_el1、ttbr0_el1/ttbr1_el1…
- **控制处理器行为**
  - 例如: WFI(陷入低功耗状态)
- **控制虚拟/物理内存**
  - 打开、配置、安装页表
- **控制外设**
  - DMA、中断



Review: 系统虚拟化的流程

- **第一步**
  - 捕捉所有系统ISA并陷入(Trap)
- **第二步**
  - 由具体指令实现相应虚拟化
    - 控制虚拟处理器行为
    - 控制虚拟内存行为
    - 控制虚拟设备行为
- **第三步**
  - 回到虚拟机继续执行



Review: 系统虚拟化技术

- **处理器虚拟化**
  - 捕捉系统ISA
  - 控制虚拟处理器的行为
- **内存虚拟化**
  - 提供“假”物理内存的抽象
- **设备虚拟化**
  - 提供虚拟的I/O设备

Review: ARM不是严格的可虚拟化架构

- **在ARM中：不是所有敏感指令都属于特权指令**
- **例子: CPSID/CPSIE指令**
  - CPSID和CPSIE分别可以关闭和打开中断
  - 内核态执行：PSTATE.{A, I, F} 可以被CPS指令修改
  - 在用户态执行：CPS 被当做NOP指令，不产生任何效果
    - 不是特权指令

Review: 如何处理这些不会下陷的敏感指令？

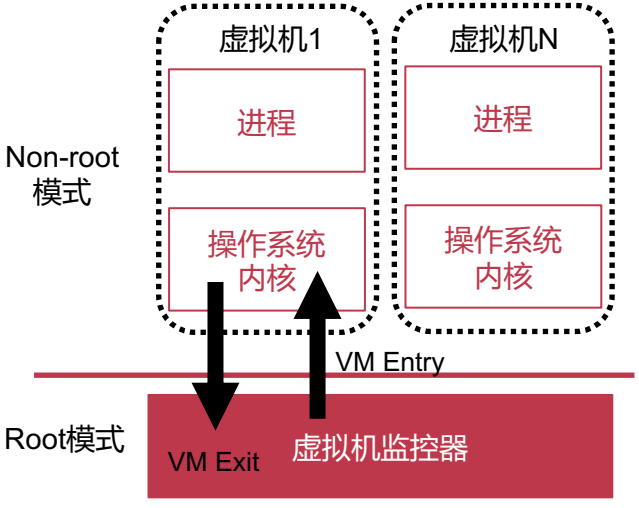
处理这些不会下陷的敏感指令，使得虚拟机中的操作系统能够运行在用户态（EL-0）

- **方法1：解释执行**
- **方法2：二进制翻译**
- **方法3：半虚拟化**
- **方法4：硬件虚拟化（改硬件）**

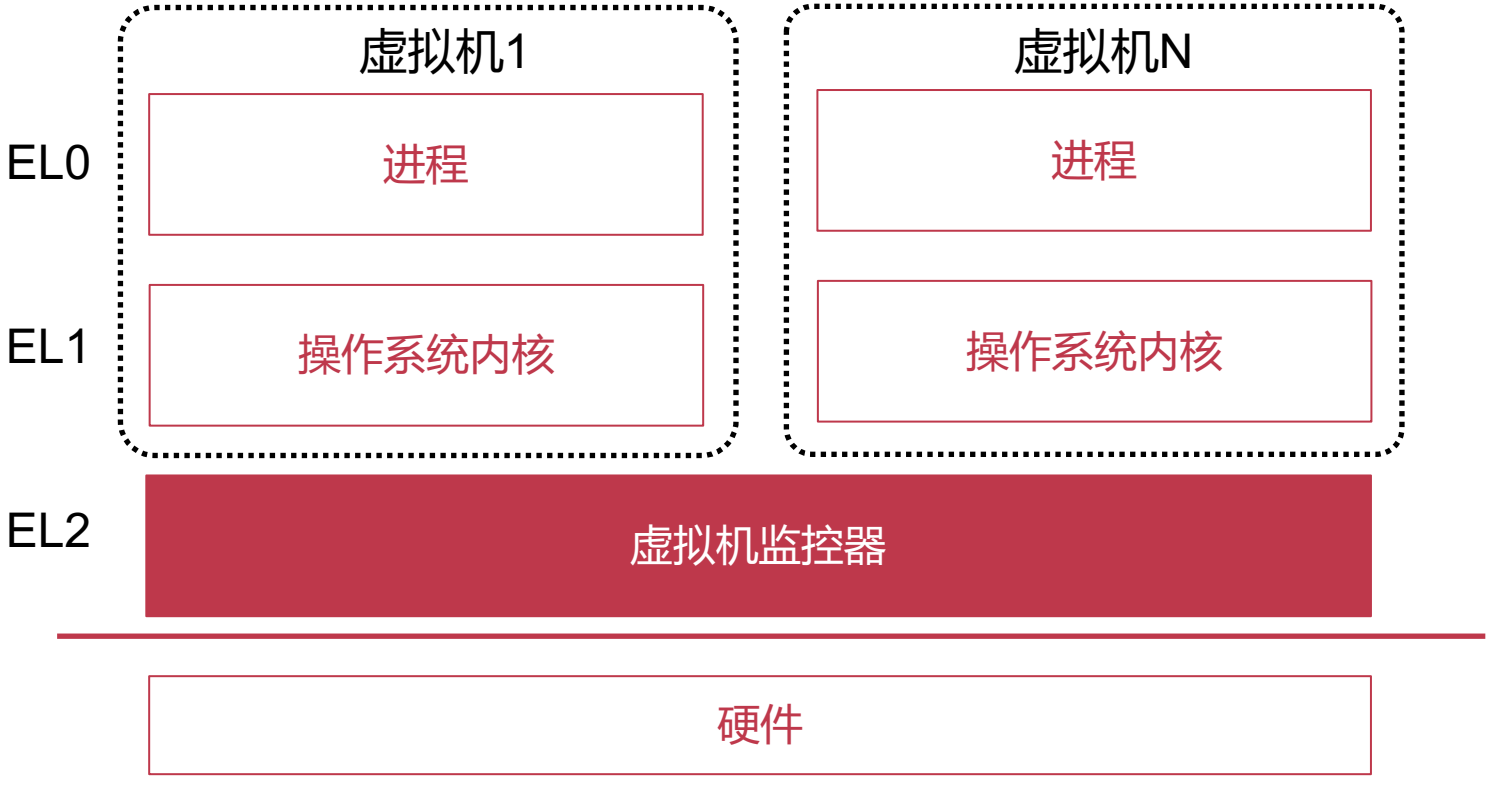
Review: Intel VT-x的处理器虚拟化

Review: x86中的VM Entry和VM Exit

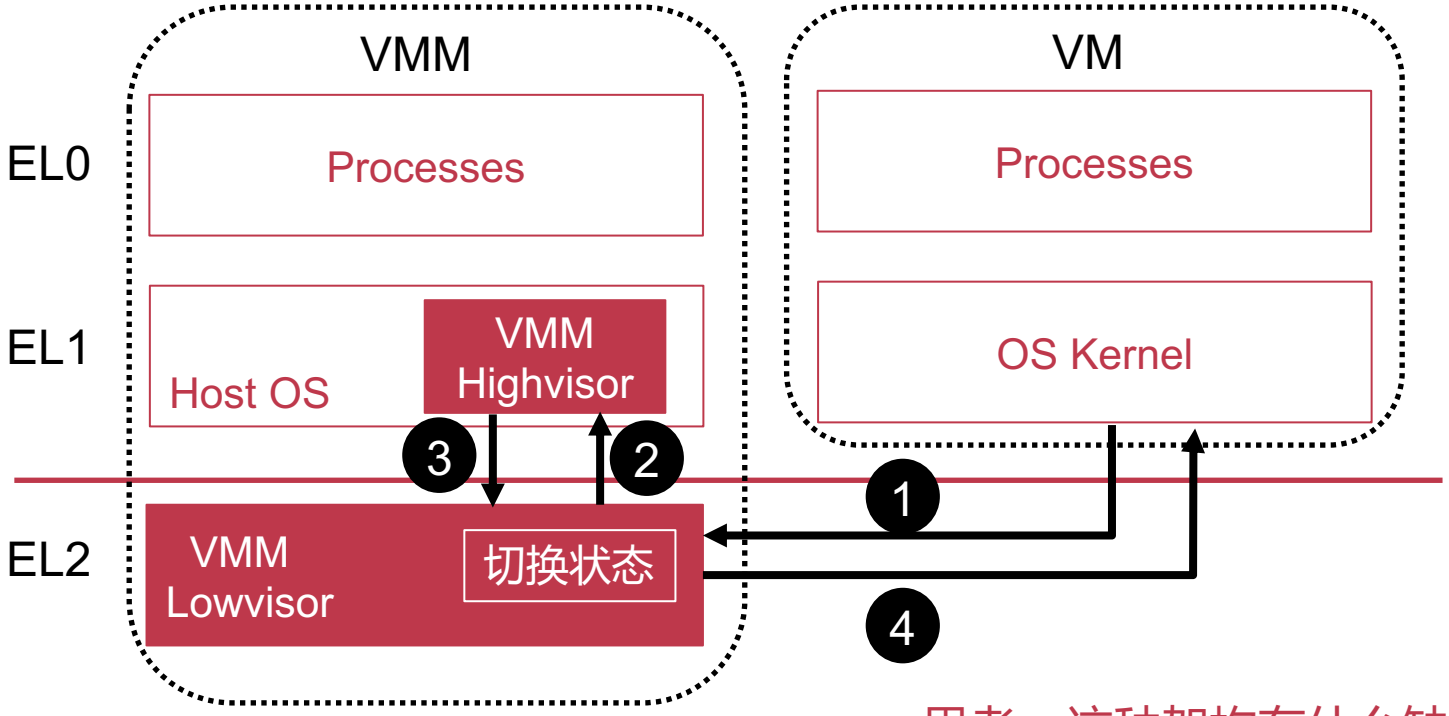
- **VM Entry**
  - 从VMM进入VM
  - 从Root模式切换到Non-root模式
  - 第一次启动虚拟机时使用VMLAUNCH指令
  - 后续的VM Entry使用VMRESUME指令
- **VM Exit**
  - 从VM回到VMM
  - 从Non-root模式切换到Root模式
  - 虚拟机执行敏感指令或发生事件(如外部中断)



Review: ARM的处理器虚拟化



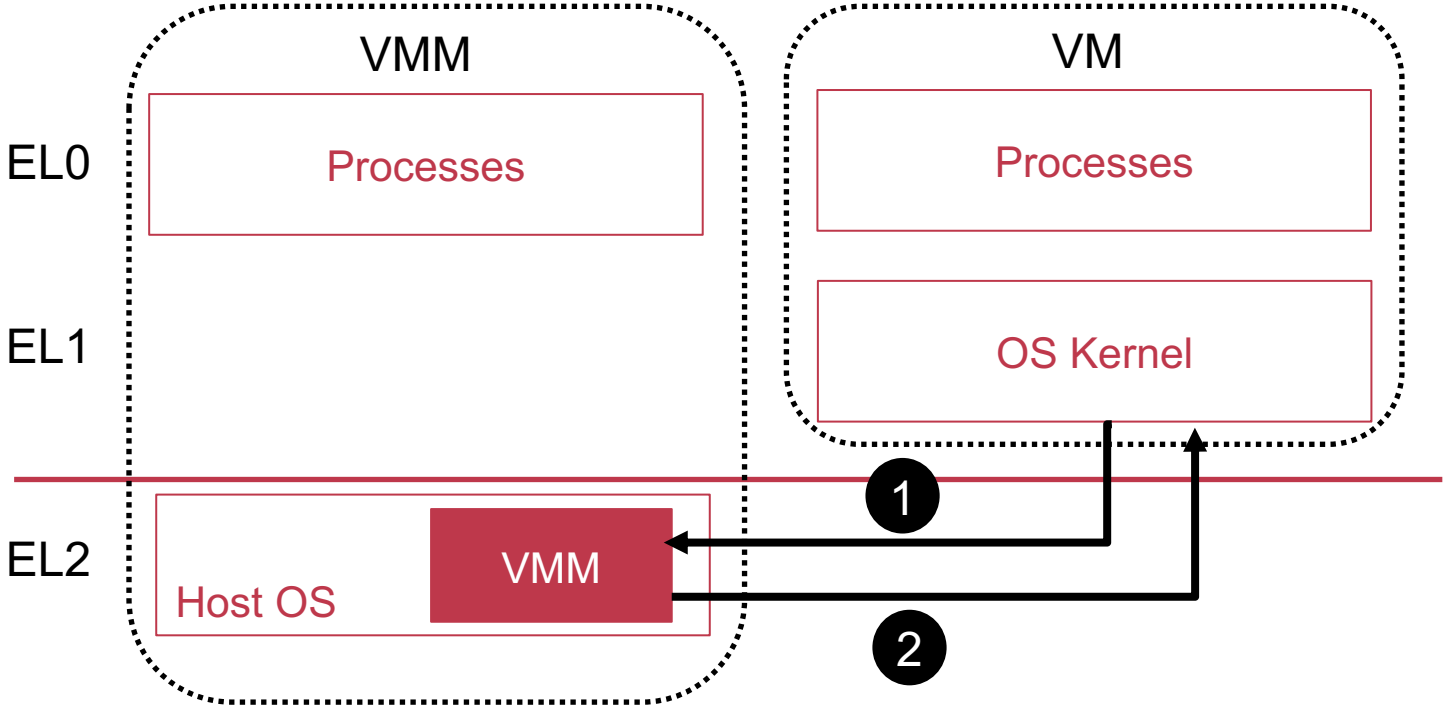
Review: ARMv8.0中的Type-2 VMM架构



思考：这种架构有什么缺点？

部件	代码行
KVM/ARM Highvisor	5094
KVM/ARM Lowvisor	718

Review: ARMv8.1中的Type-2 VMM架构



案例：QEMU/KVM

QEMU发展历史

- **2003年，法国程序员Fabrice Bellard发布了QEMU 0.1版本**
  - 目标是在非x86机器上使用动态二进制翻译技术模拟x86机器
- **2003-2006年**
  - 能模拟出多种不同架构的虚拟机，包括S390、ARM、MIPS、SPARC等
  - 在这阶段，QEMU一直使用软件方法进行模拟
    - 如二进制翻译技术

QEMU发展历史

Fabrice Bellard fabrice.bellard at free.fr
Sun Mar 23 14:46:47 CST 2003

- Previous message: SPI_GETGRADIENTCAPTIONS
- Next message: [announce] QEMU x86 emulator version 0.1
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Hi,

The first release of the QEMU x86 emulator is available at
http://bellard.org/qemu/. QEMU achieves a fast user space Linux x86
emulation on x86 and PowerPC Linux hosts by using dynamic translation.
Its main goal is to be able to run the Wine project on non-x86
architectures.

Fabrice.
```

KVM发展历史

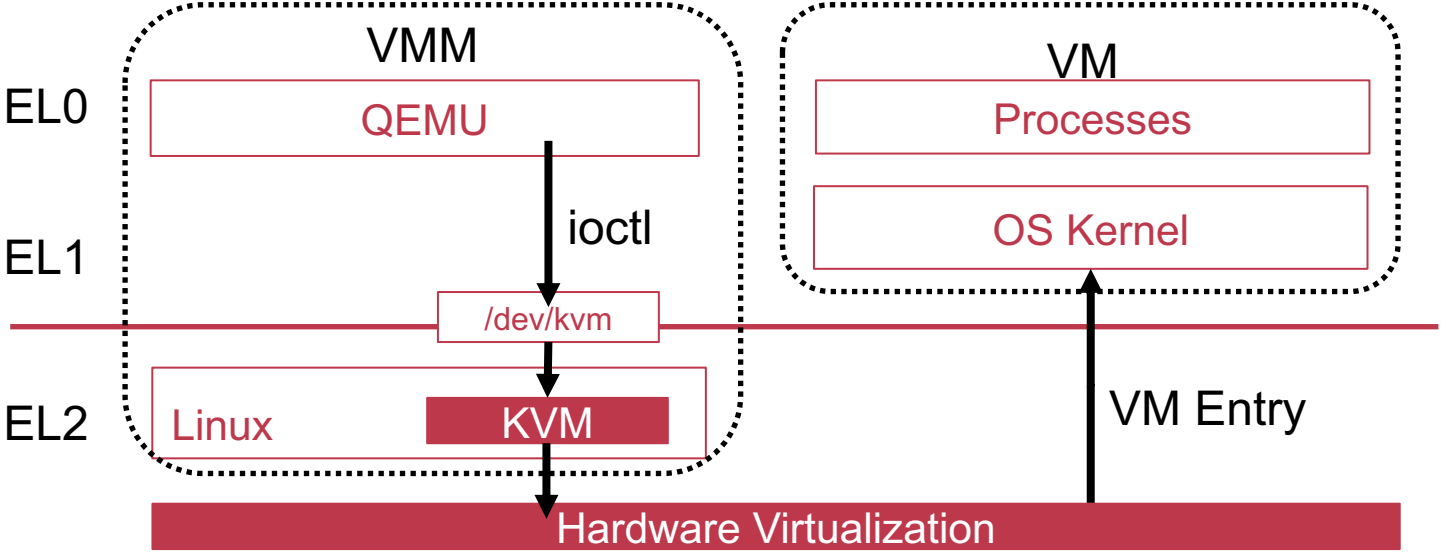
- 2005年11月，Intel发布带有VT-x的两款Pentium 4处理器
- 2006年中期， Qumranet公司在内部开发KVM(Kernel-based Virtual Machine)，并于11月发布
- 2007年，KVM被整合进Linux 2.6.20
- 2008年9月，Redhat出资1亿700美元收购Qumranet
- 2009年，QEMU 0.10.1开始使用KVM，以替代其软件模拟的方案

QEMU/KVM架构

- **QEMU运行在用户态，负责实现策略**
  - 也提供虚拟设备的支持
- **KVM以Linux内核模块运行，负责实现机制**
  - 可以直接使用Linux的功能
  - 例如内存管理、进程调度
  - 使用硬件虚拟化功能
- **两部分合作**
  - KVM捕捉所有敏感指令和事件，传递给QEMU
  - KVM不提供设备的虚拟化，需要使用QEMU的虚拟设备

QEMU使用KVM的用户态接口

- **QEMU使用/dev/kvm与内核态的KVM通信**
  - 使用ioctl向KVM传递不同的命令：CREATE_VM, CREATE_VCPU, KVM_RUN等



QEMU使用KVM的用户态接口

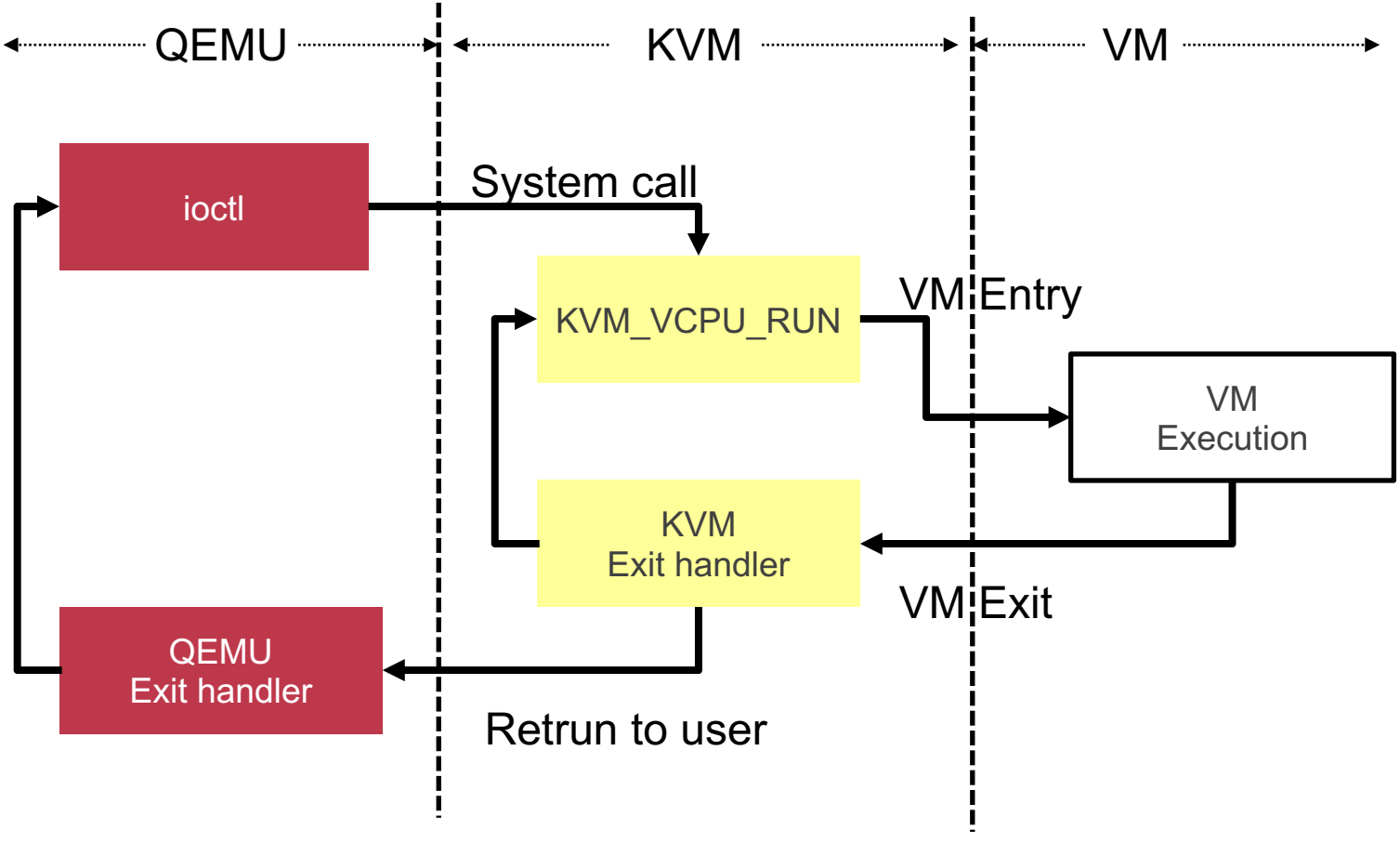
```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
while (true) {
      ioctl(KVM_RUN)
      exit_reason = get_exit_reason();
      switch (exit_reason) {
        case KVM_EXIT_IO:  /* ... */
          break;
        case KVM_EXIT_MMIO: /* ... */
          break;
      }
}
```

Invoke VMENTRY

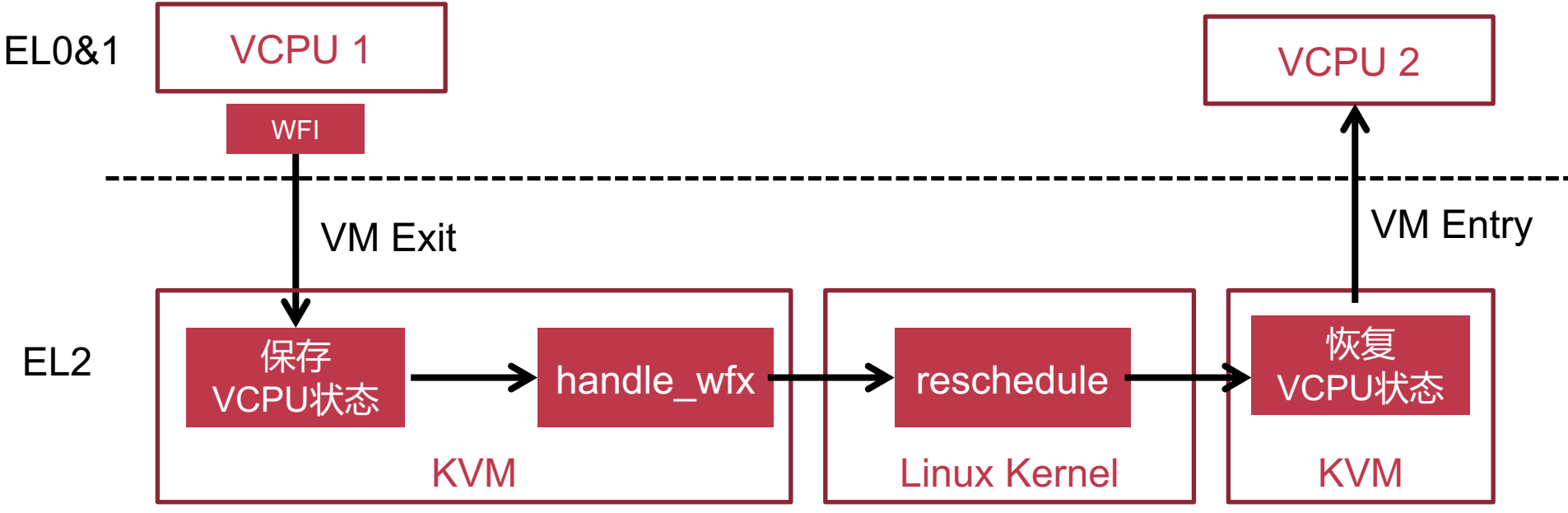
ioctl(KVM_RUN)时发生了什么

- **x86中**
  - KVM找到此VCPU对应的VMCS
  - 使用指令加载VMCS
  - VMLAUNCH/VMRESUME进入Non-root模式
    - 硬件自动同步状态
    - PC切换成VMCS->GUEST_RIP，开始执行
- **ARM中**
  - KVM主动加载VCPU对应的所有状态
  - 使用eret指令进入EL2
    - PC切换成ELR_EL2的值，开始执行

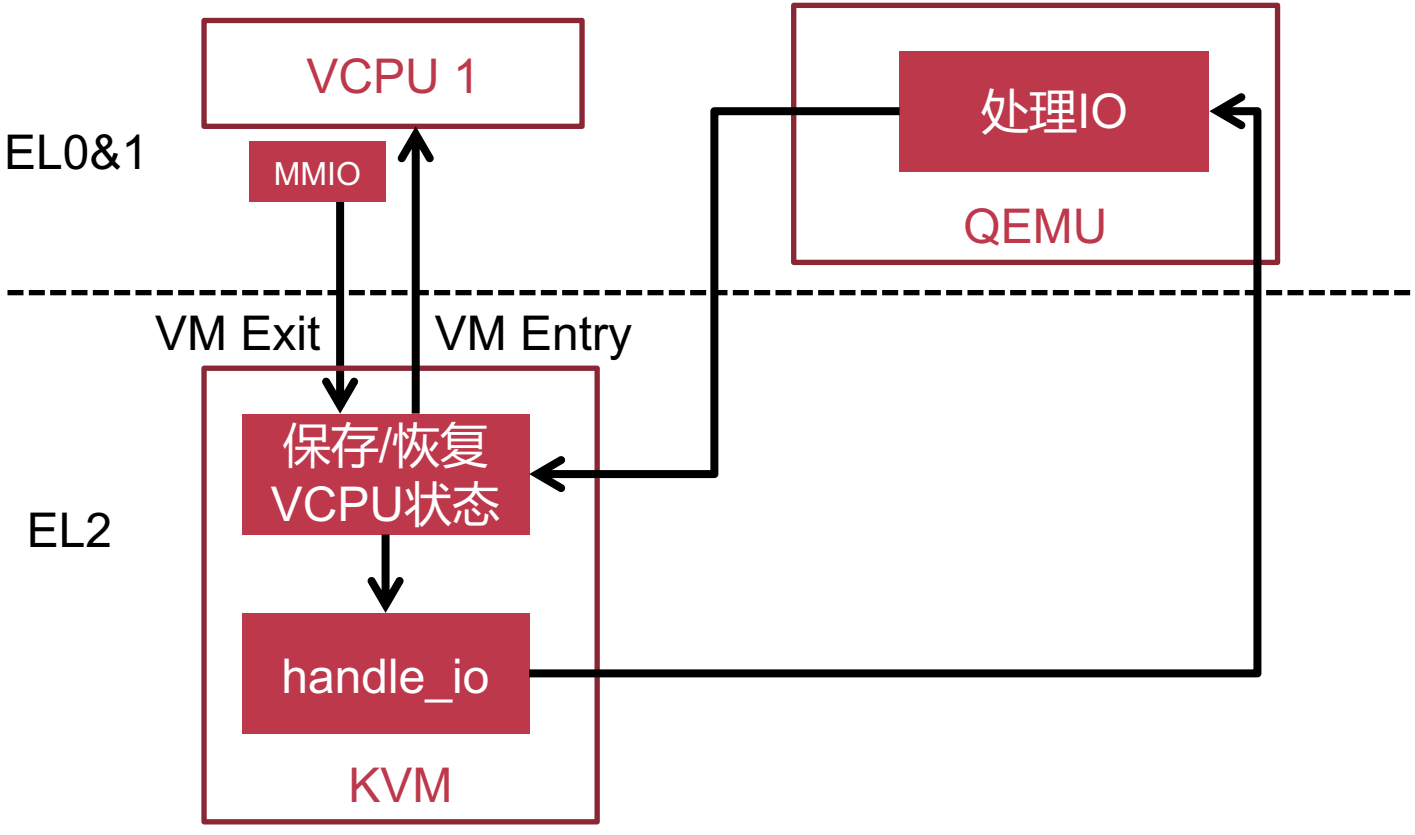
QEMU/KVM的流程


QEMU
KVM
VM
ioctl
System call
KVM_VCPU_RUN
VM Entry
VM
Execution
KVM
Exit handler
VM Exit
QEMU
Exit handler
Retrun to user

例：WFI指令VM Exit的处理流程



例：I/O指令VM Exit的处理流程

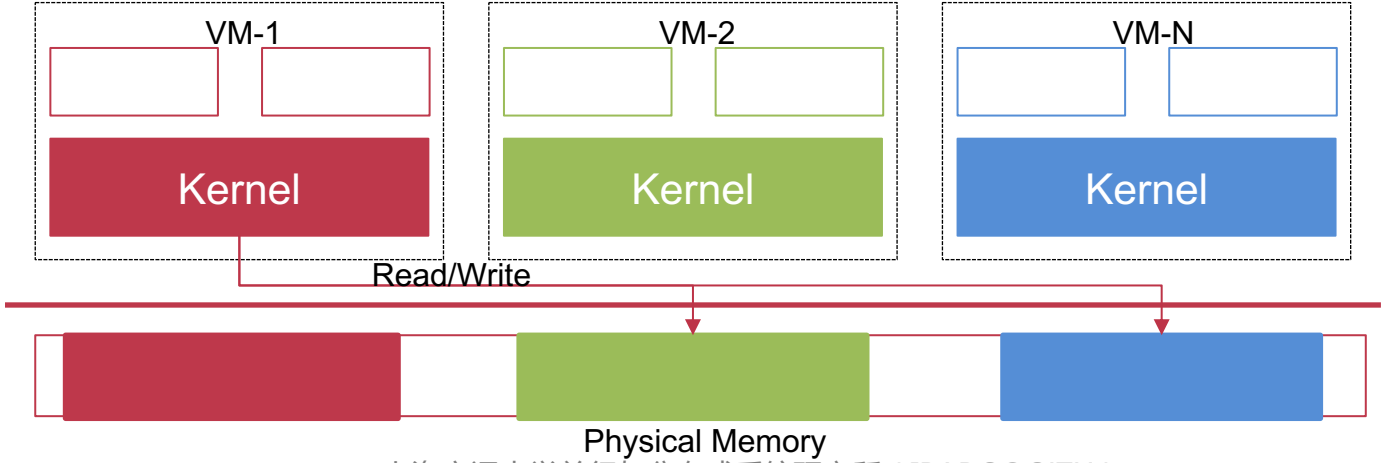


Memory Virtualization

内存虚拟化

为什么需要内存虚拟化?

- **操作系统内核直接管理物理内存**
  - 物理地址从0开始连续增长
  - 向上层进程提供虚拟内存的抽象
- **如果VM使用的是真实物理地址**



内存虚拟化的目标

- **为虚拟机提供虚拟的物理地址空间**
  - 物理地址从0开始连续增长
- **隔离不同虚拟机的物理地址空间**
  - VM-1无法访问其他的内存

三种地址

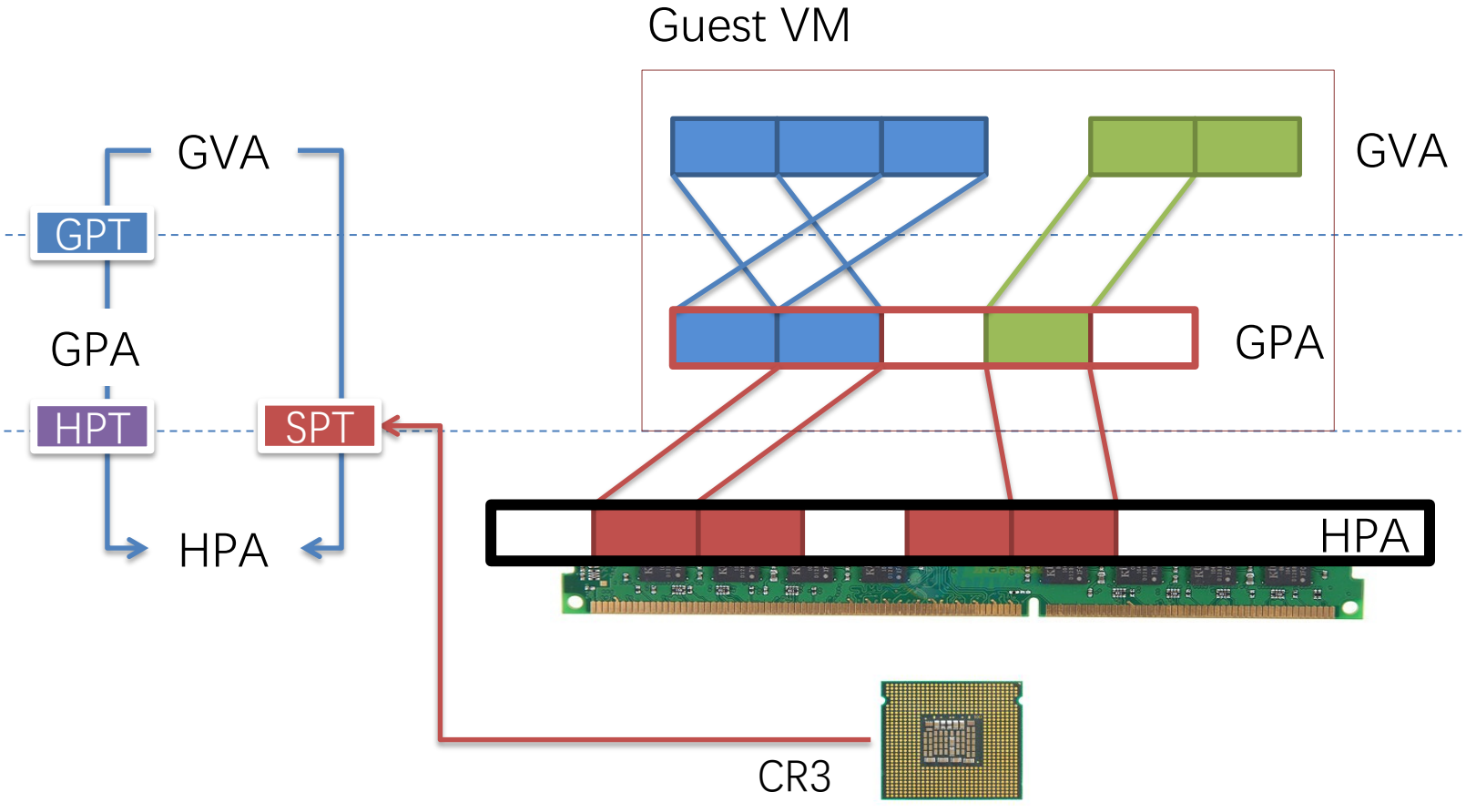
- **客户虚拟地址(Guest Virtual Address, GVA)**
  - 虚拟机内进程使用的虚拟地址
- **客户物理地址(Guest Physical Address, GPA)**
  - 虚拟机内使用的“假”物理地址
- **主机物理地址(Host Physical Address, HPA)**
  - 真实寻址的物理地址
  - GPA需要翻译成HPA才能访存

VMM管理

怎么实现内存虚拟化?

- 1、影子页表(Shadow Page Table)
- 2、直接页表(Direct Page Table)
- 3、硬件虚拟化

1、影子页表



2个页表 -> 1个页表

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

Shadow Page Table 设置

```
set_cr3 (guest_page_table):
    for GVA in 0 to 2^20
        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[GVA] = 0
     CR3 = PHYSICAL_ADDR(shadow_page_table)
```

Guest OS修改页表，如何生效？

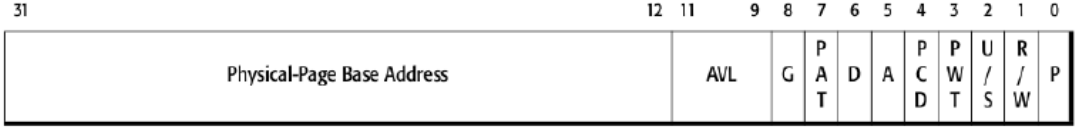
- **Real hardware would start using the new page table's mappings**
  - Virtual machine monitor has a separate shadow page table
- **Goal:**
  - VMM needs to intercept when guest OS modifies page table, update shadow page table accordingly
- **Technique:**
  - Use the read/write bit in the PTE to mark those pages read-only
  - If guest OS tries to modify them, hardware triggers page fault
  - Page fault handled by VMM: update shadow page table & restart guest

Guest内核如何与Guest应用隔离？

- **How do we selectively allow / deny access to kernel-only pages in guest PT?**
  - Hardware doesn't know about the virtual U/K bit
- **Idea:**
  - Generate two shadow page tables, one for U, one for K
  - When guest OS switches to U mode, VMM must invoke set_ptp(current, 0)

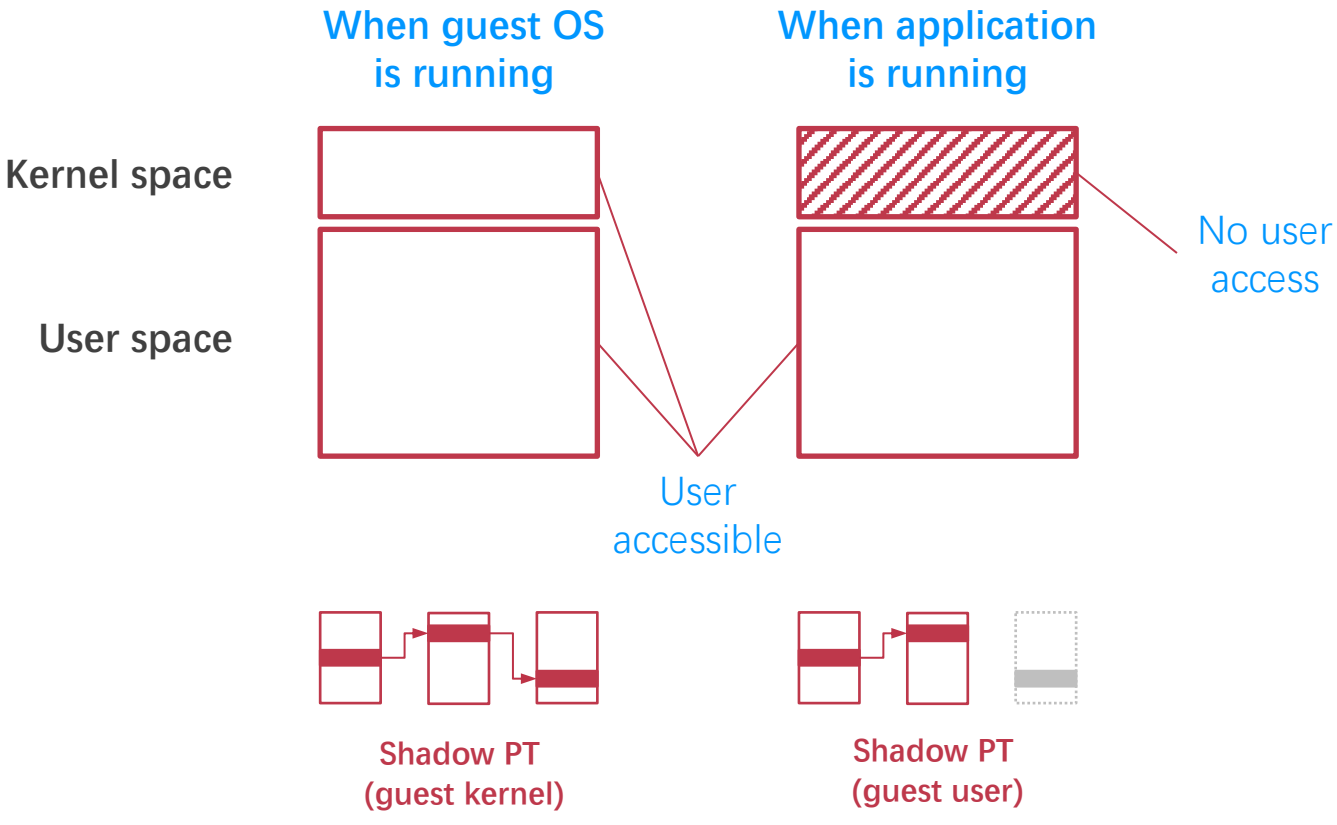
1个页表 -> 2个页表

Guest Kernel-only Pages




```
set_ptp(guest_pt, kmode):
    for gva in 0 .. 2^20:
        if guest_pt[gva] & PTE_P and
          (kmode or guest_pt[gva] & PTE_U):
            gpa = guest_pt[gva] >> 12
            hpa = host_pt[gpa] >> 12
            shadow_pt[gva] = (hpa << 12) | PTE_P | PTE_U
        else:
            shadow_pt[gva] = 0
    PTP = shadow_pt
```

Two Memory Views of Guest VM



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

2、Direct Paging (Para-virtualization)

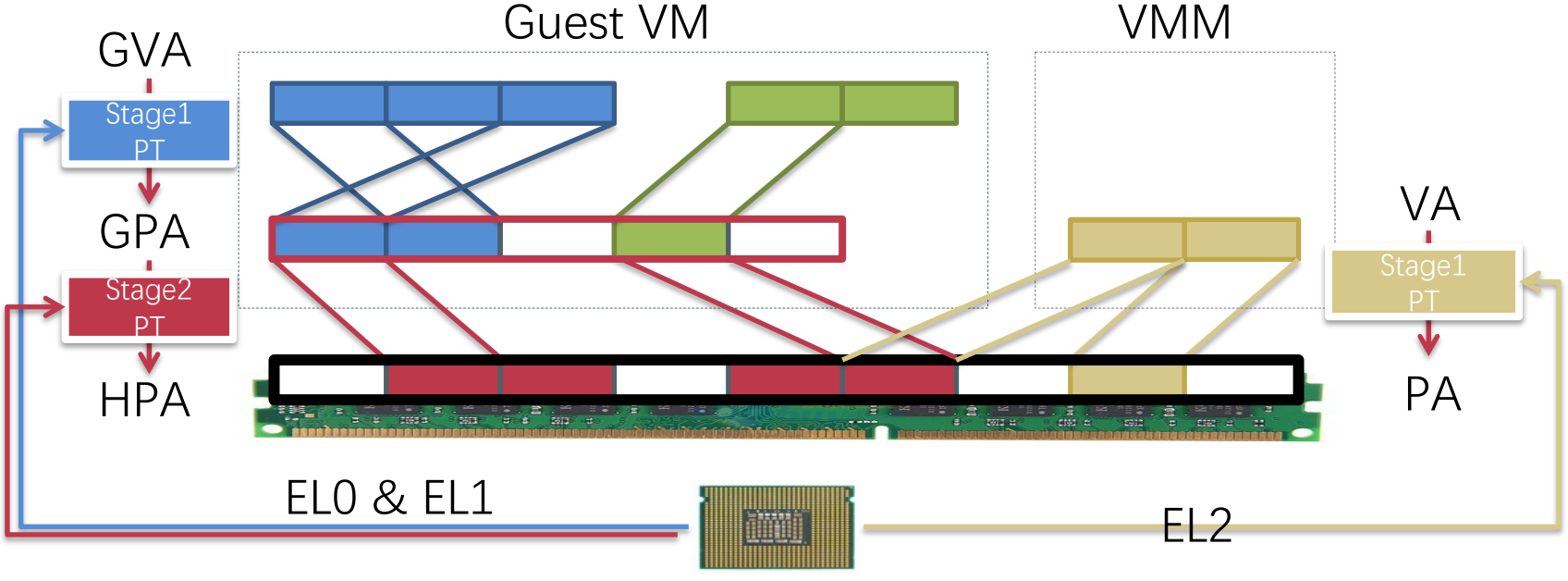
- **Positive**
  - Easy to implement and more clear architecture
  - Better performance: guest can batch to reduce trap
- **Negatives**
  - Not transparent to the guest OS
  - The guest now knows much info, e.g., HPA
    - May use such info to trigger *rowhammer* attacks

3、硬件虚拟化对内存翻译的支持

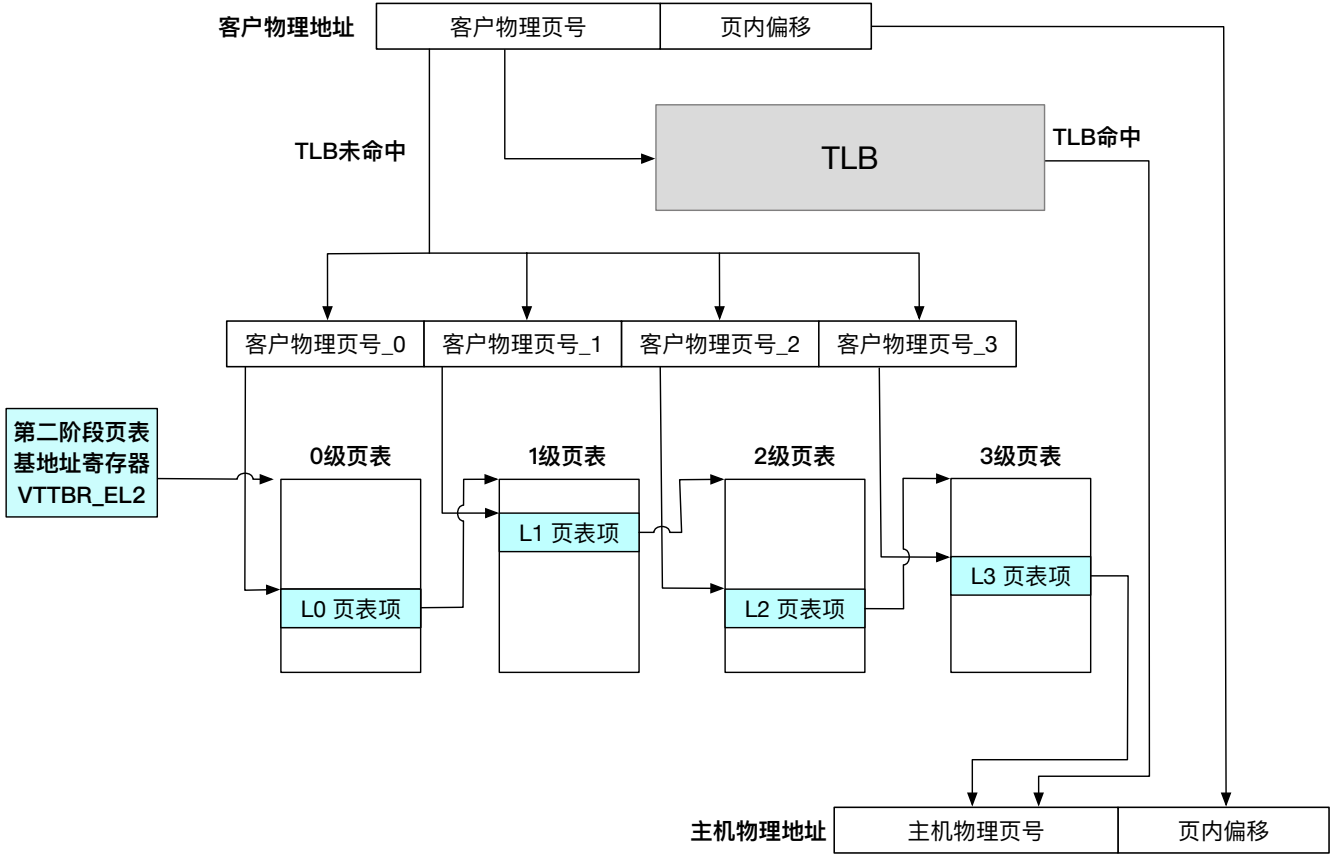
- **Intel VT-x和ARM硬件虚拟化都有对应的内存虚拟化**
  - Intel Extended Page Table (EPT)
  - ARM Stage-2 Page Table (第二阶段页表)
- **新的页表**
  - 将GPA翻译成HPA
  - 此表被VMM直接控制
  - 每一个VM有一个对应的页表

第二阶段页表

- 第一阶段页表：虚拟机内虚拟地址翻译（GVA->GPA）
- 第二阶段页表：虚拟机客户物理地址翻译（GPA->HPA）



第二阶段4级页表

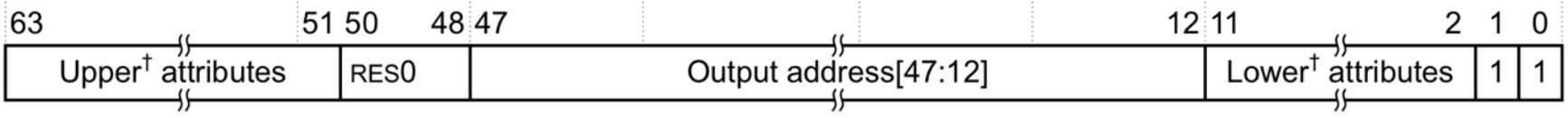

客户物理地址
客户物理页号
页内偏移
TLB未命中
TLB
TLB命中
客户物理页号_0
客户物理页号_1
客户物理页号_2
客户物理页号_3
第二阶段页表
基地址寄存器
VTTBR_EL2
0级页表
1级页表
2级页表
3级页表
L0 页表项
L1 页表项
L2 页表项
L3 页表项
主机物理地址
主机物理页号
页内偏移

VTTBR_EL2

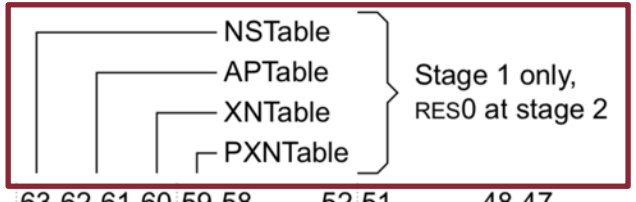
- **存储虚拟机第二阶段页表基地址**
  - 只有1个寄存器：VTTBR_EL2
- **对比第一阶段页表**
  - 有2个页表基地址寄存器：TTBR0_EL1、TTBR1_EL1
- **VMM在调度VM之前需要在VTTBR_EL2中写入此VM的第二阶段页表基地址**
- **第二阶段页表使能**
  - HCR_EL2第0位

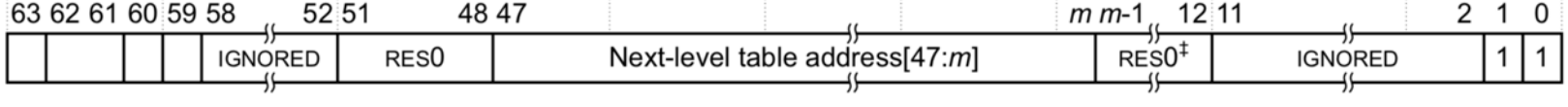
第二阶段页表项

- **第3级页表页中的页表项**
  - 与第一阶段页表完全一致



- **第0-2级页表页中的页表项**
  - 与第一阶段在高位有不同

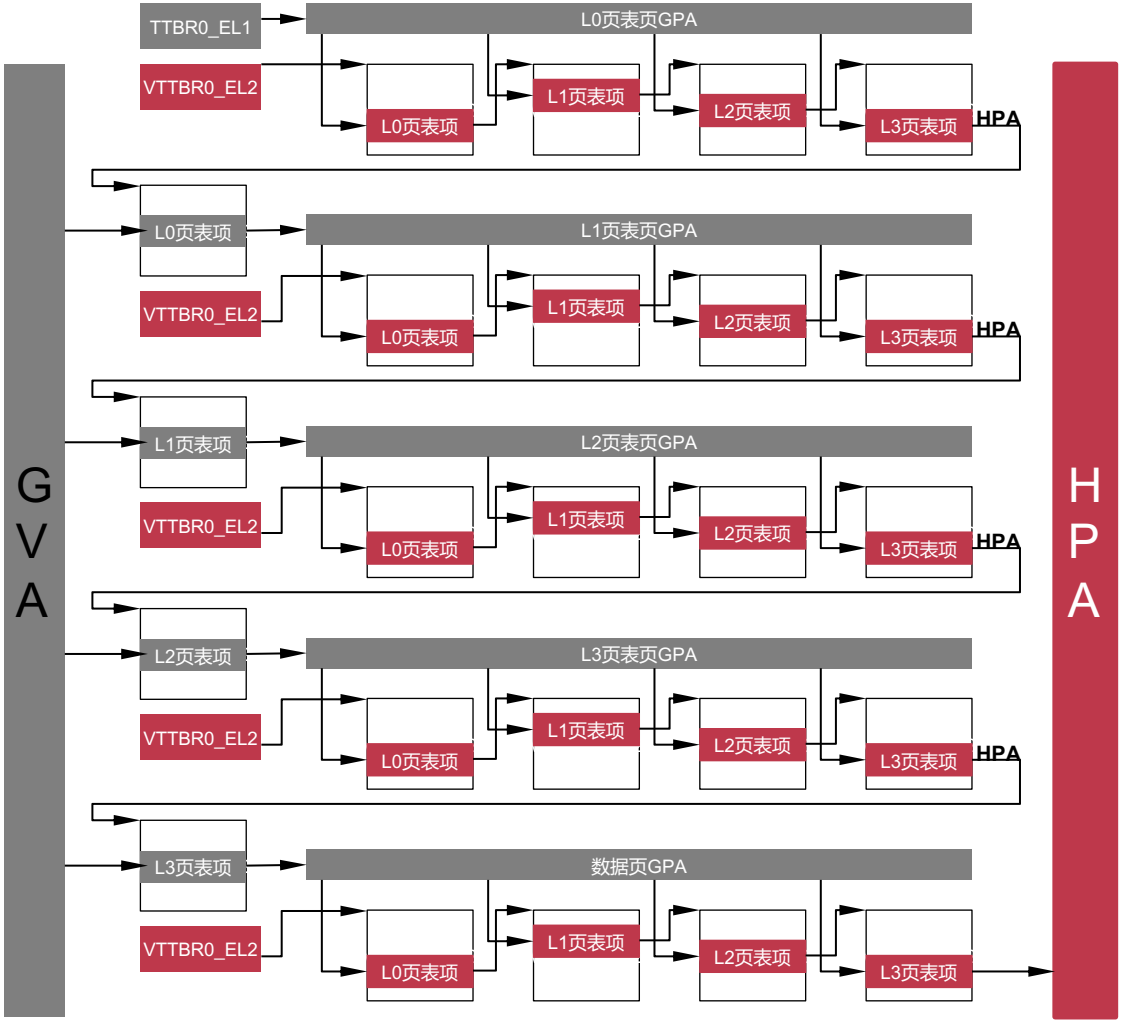




- **NSTable: 与TrustZone有关**
- **APTable：读写权限**
- **XNTable：执行权限**
- **PXNTable：特权级别软件的执行权限**

翻译过程

- **总共24次内存访问**
  - 为什么?
  - 25-1
  - 第一次访问寄存器



TLB：缓存地址翻译结果

- **回顾：TLB不仅可以缓存第一阶段地址翻译结果**
- **TLB也可以第二阶段地址翻译后的结果**
  - 包括第一阶段的翻译结果(GVA->GPA)
  - 包括第二阶段的翻译结果(GPA->HPA)
  - 大大提升GVA->HPA的翻译性能：不需要24次内存访问
- **切换VTTBR_EL2时**
  - 理论上应将前一个VM的TLB项全部刷掉

TLB刷新

- **刷TLB相关指令**
  - 清空全部
    - TLBI VMALLS12E1IS
  - 清空指定GVA
    - TLBI VAE1IS
  - 清空指定GPA
    - TLBI IPAS2E1IS
- **VMID (Virtual Machine IDentifier)**
  - VMM为不同进程分配8/16 VMID，将VMID填写在VTTBR_EL2的高8/16位
  - VMID位数由VTCR_EL2的第19位（VS位）决定
  - 避免刷新上个VM的TLB

如何处理缺页异常

- **两阶段翻译的缺页异常分开处理**
- **第一阶段缺页异常**
  - 直接调用VM的Page fault handler
  - 修改第一阶段页表**不会**引起任何虚拟机下陷
- **第二阶段缺页异常**
  - 虚拟机下陷，直接调用VMM的Page fault handler

第二阶段页表的优缺点

- **优点**
  - VMM实现简单
  - 不需要捕捉Guest Page Table的更新
  - 减少内存开销：每个VM对应一个页表
- **缺点**
  - TLB miss时性能开销较大

想一想：二阶段页表翻译还带了什么能力？