IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

系统虚拟化2

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

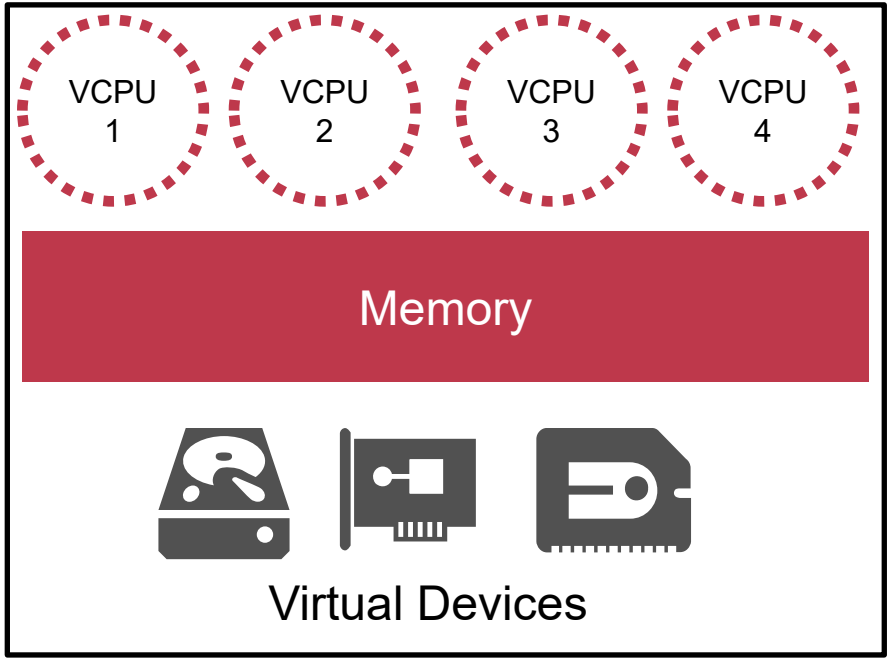
- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

回顾：处理器虚拟化

- **解释执行**
- **二进制翻译**
- **半虚拟化**
- **硬件虚拟化**
  - Intel VT-x
  - ARM VHE

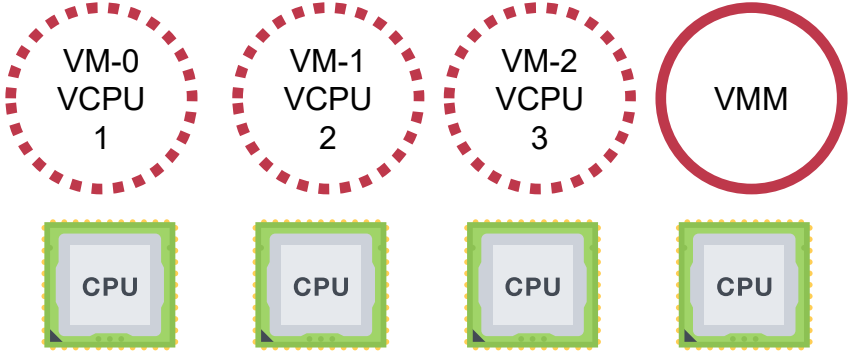
VM和VCPU

- **VM (Virtual Machine)**
  - 静态部分：VM的内存、设备等
  - 动态部分：VCPU
- **VCPU (Virtual CPU)**
  - 用线程模拟CPU
  - 虚拟寄存器
    - 程序计数器
    - 通用寄存器
    - 系统寄存器

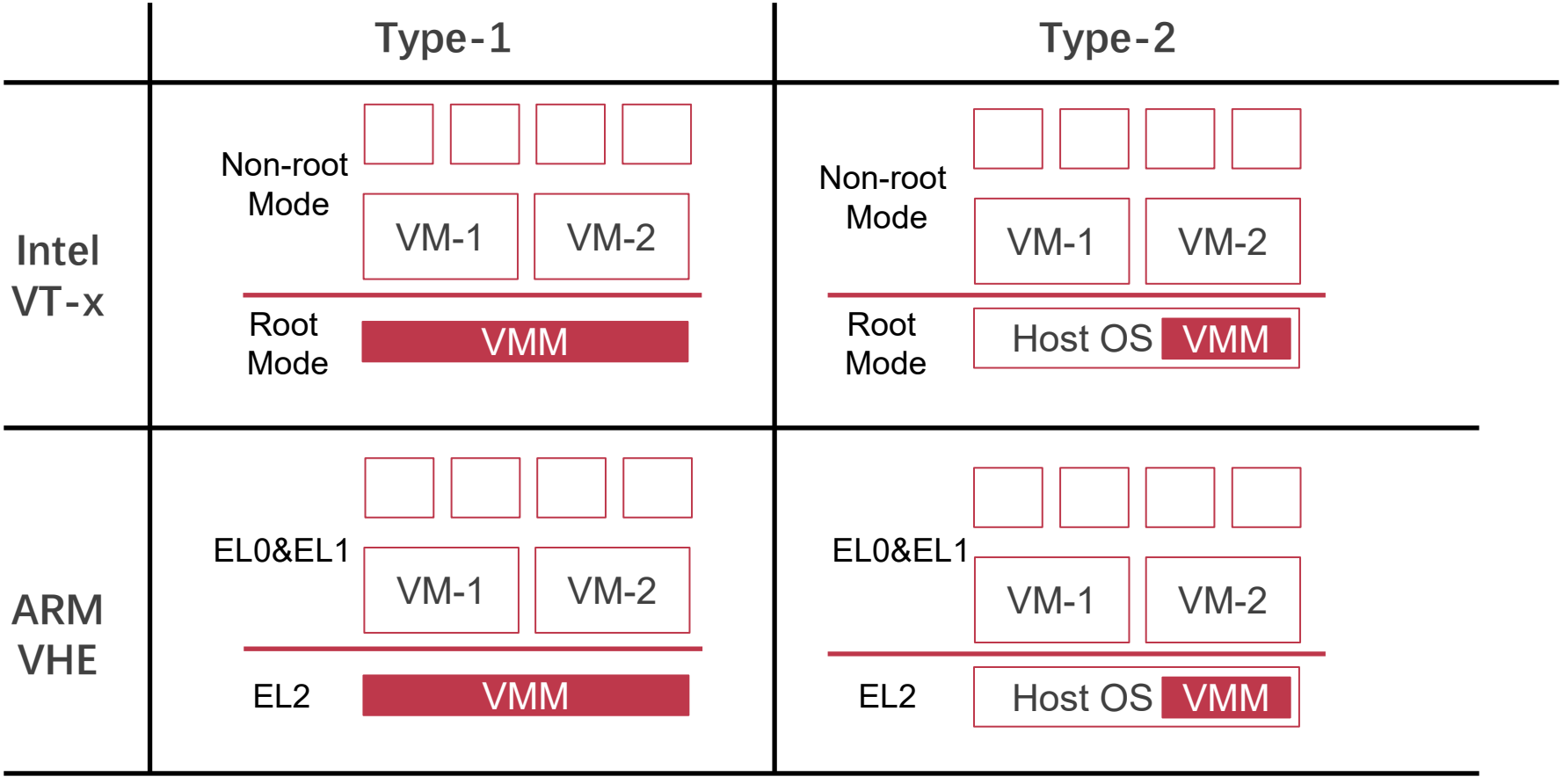


虚拟机的执行

- **虚拟机的执行与进程执行类似**
  - 进程的执行：OS将进程的每个线程调度在物理CPU核中执行
  - 虚拟机执行：VMM将VM的每个VCPU调度在物理CPU核中执行



Type-1和Type-2在VT-x和VHE下架构



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

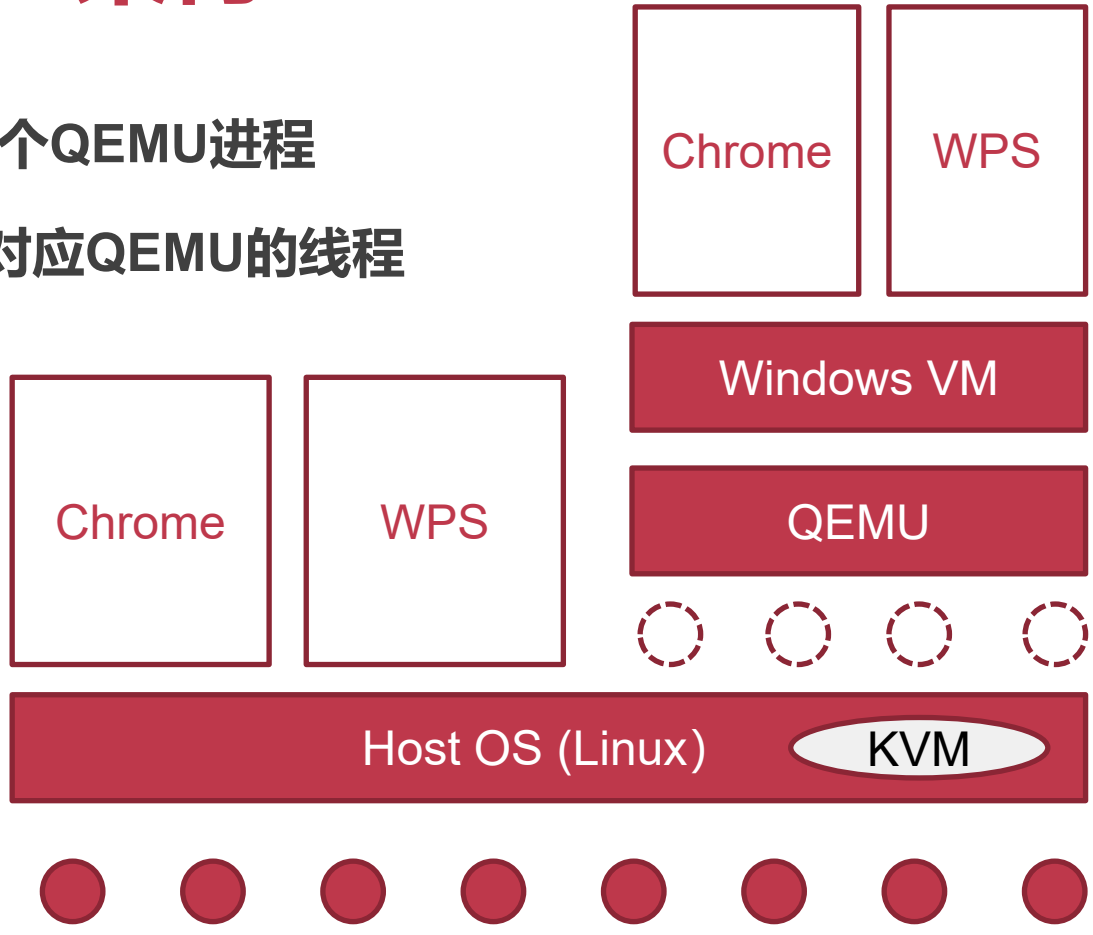
- 2005年11月，Intel发布了带有VT-x的两款Pentium 4处理器
- 2006年中期， Qumranet公司在内部开发KVM(Kernel-based Virtual Machine)，并于11月发布
- 2007年，KVM被整合进Linux 2.6.20
- 2008年9月，Redhat出资1亿700美元收购Qumranet
- 2009年，QEMU 0.10.1开始使用KVM，以替代其软件模拟的方案

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

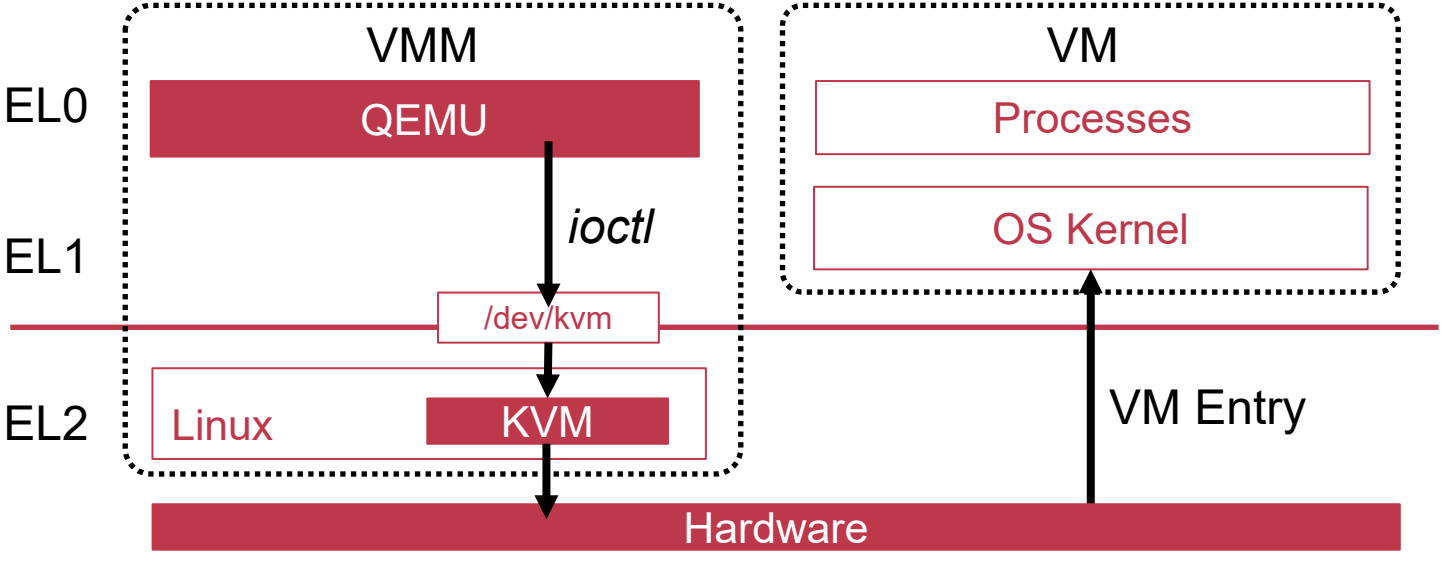
QEMU/KVM架构

- 1个虚拟机对应1个QEMU进程
- 虚拟机的VCPU对应QEMU的线程



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

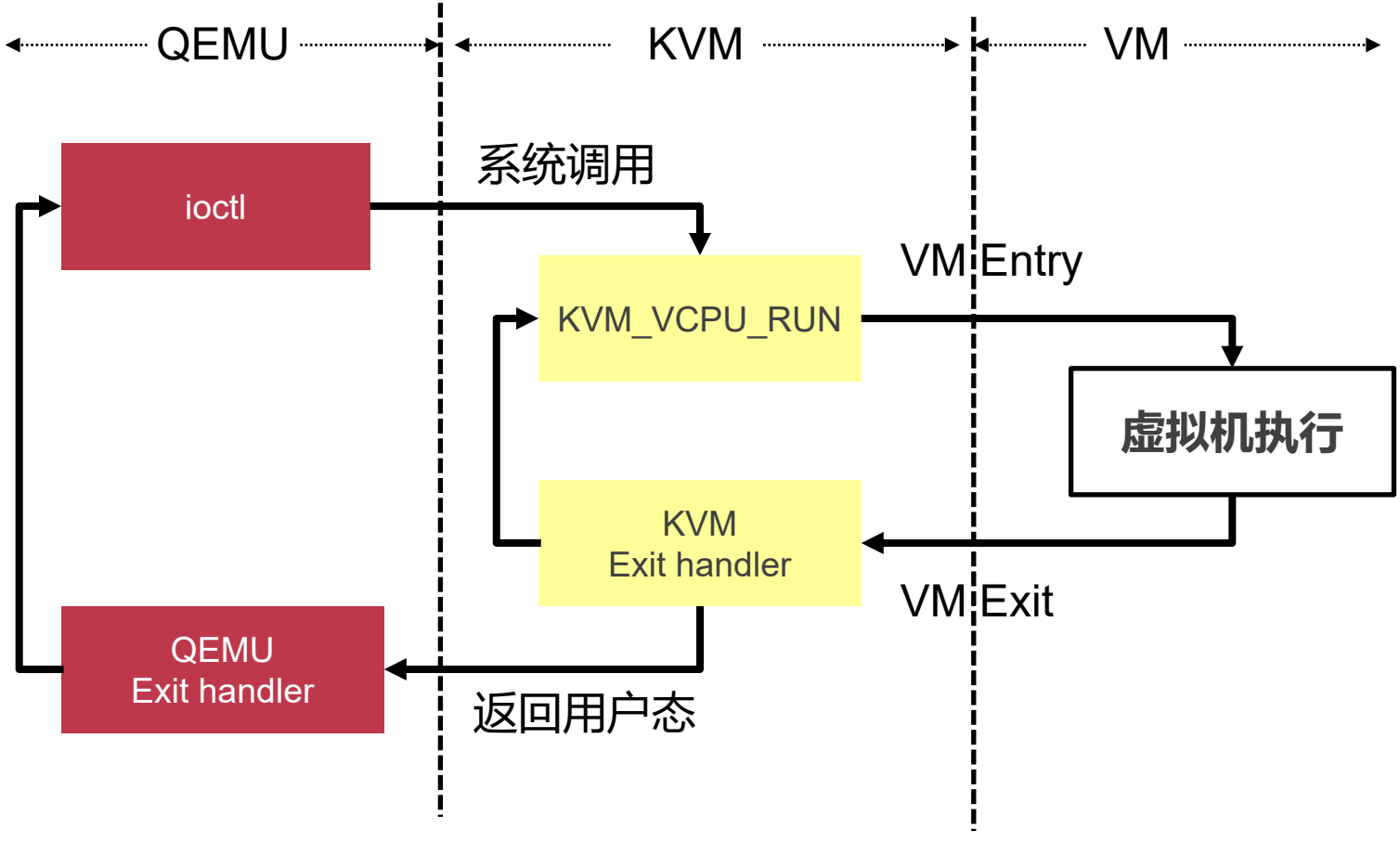
```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
while (true) {
      ioctl(KVM_RUN)
      exit_reason = get_exit_reason();
      switch (exit_reason) {
        case KVM_EXIT_IO:  /* ... */
          break;
        case KVM_EXIT_MMIO: /* ... */
          break;
      }
}
```

Invoke VMENTRY

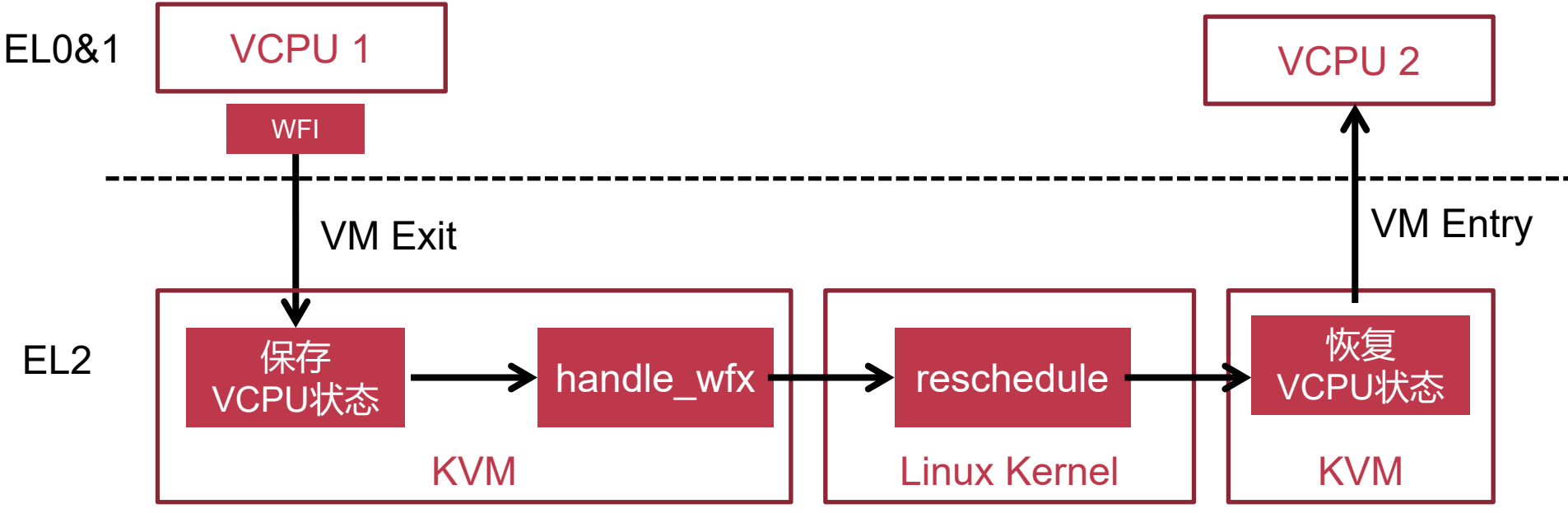
ioctl(KVM_RUN)时发生了什么

- **在ARM平台**
  - KVM主动加载VCPU对应的所有状态
  - 使用eret指令进入EL1或EL0（取决于VMExit时的状态）
    - PC切换成VMExit时对应的值，开始执行
- **在x86平台**
  - 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

QEMU/KVM的流程

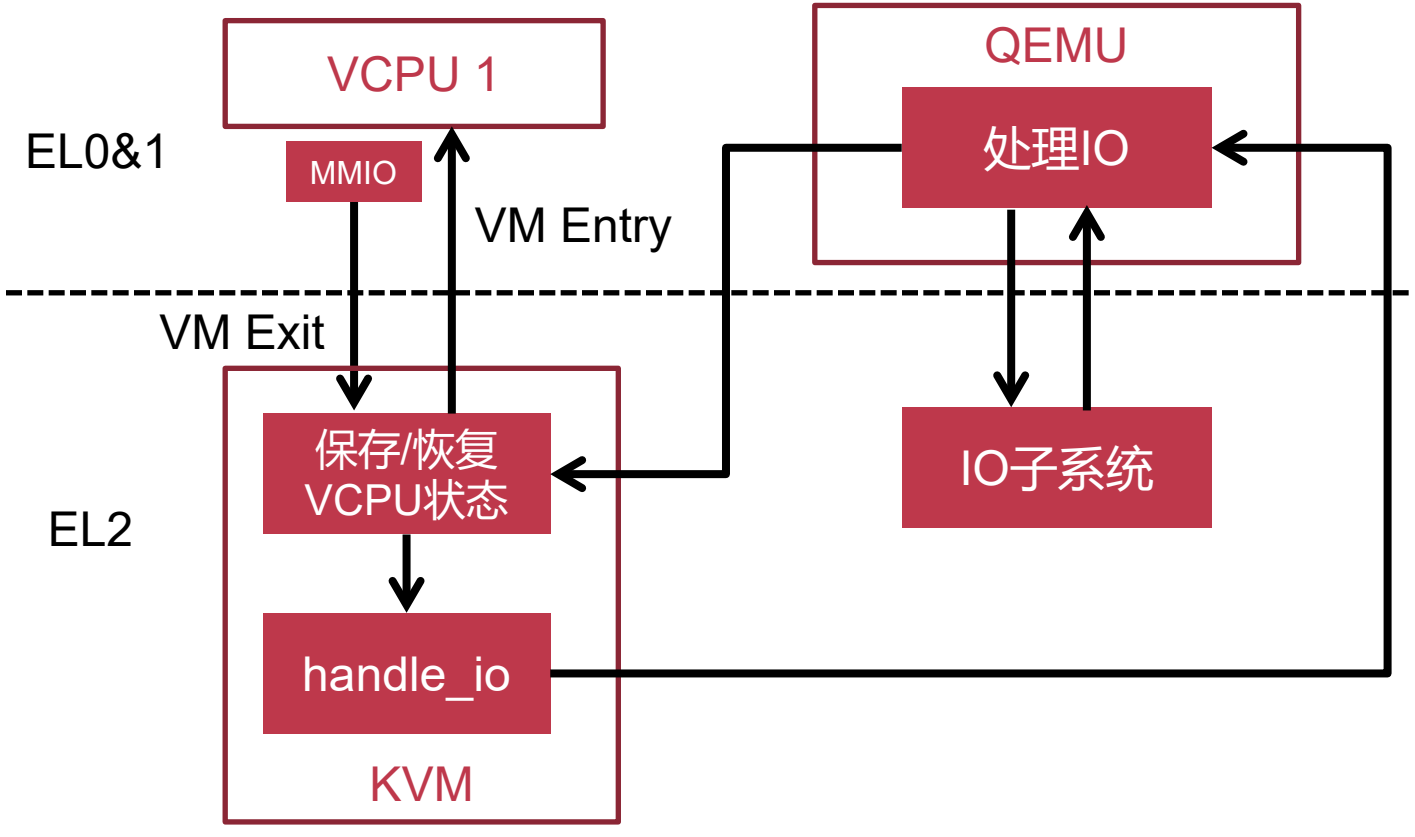

QEMU
KVM
VM
ioctl
系统调用
KVM_VCPU_RUN
VM Entry
虚拟机执行
KVM
Exit handler
VM Exit
QEMU
Exit handler
返回用户态

例：WFI指令VM Exit的处理流程



WFI：Wait For Interrupt，类似x86的halt指令

例：I/O指令VM Exit的处理流程

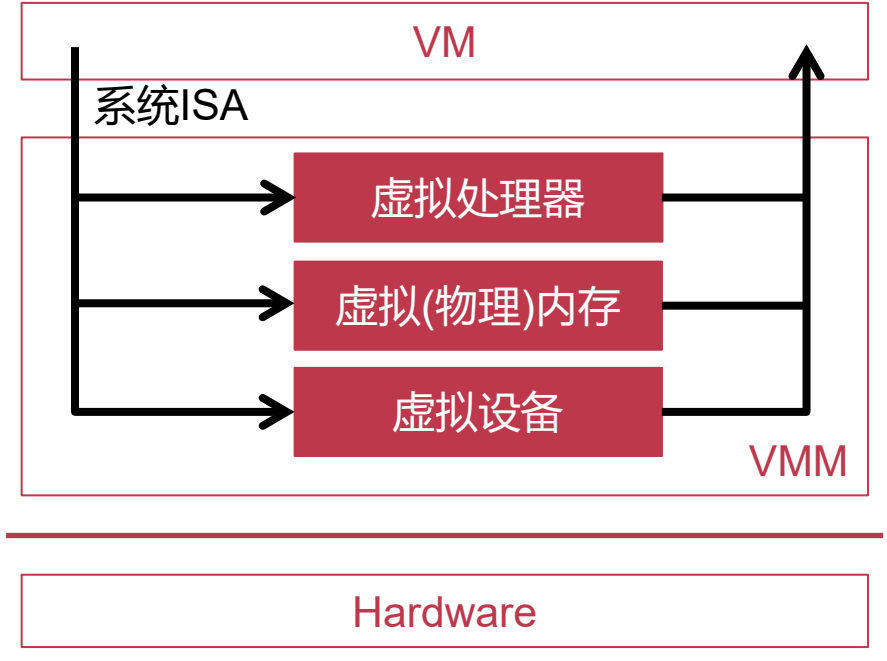


Memory Virtualization

内存虚拟化

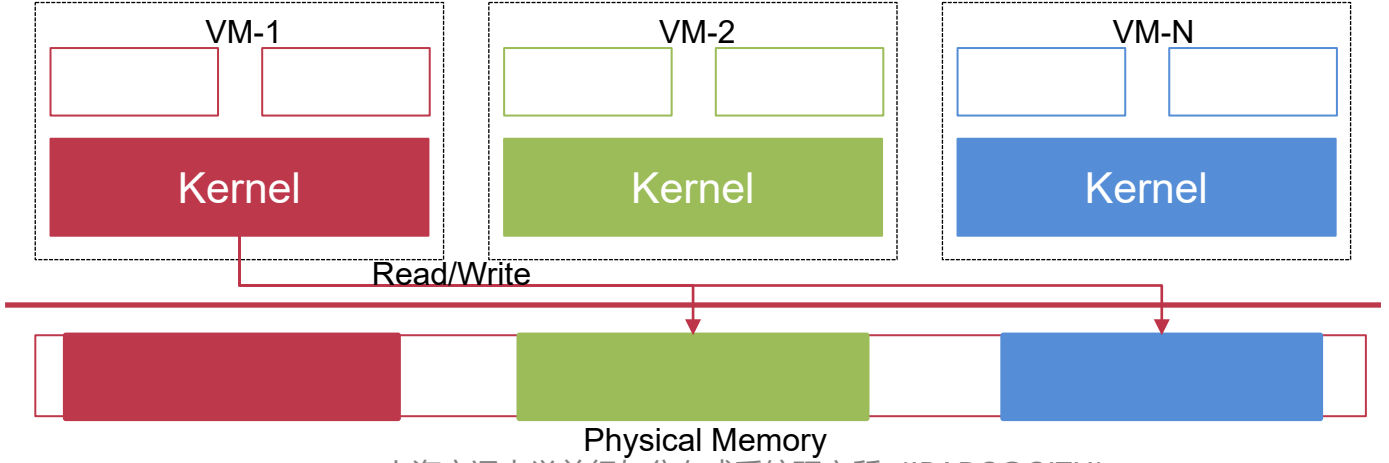
回顾：系统虚拟化的流程

- **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



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?

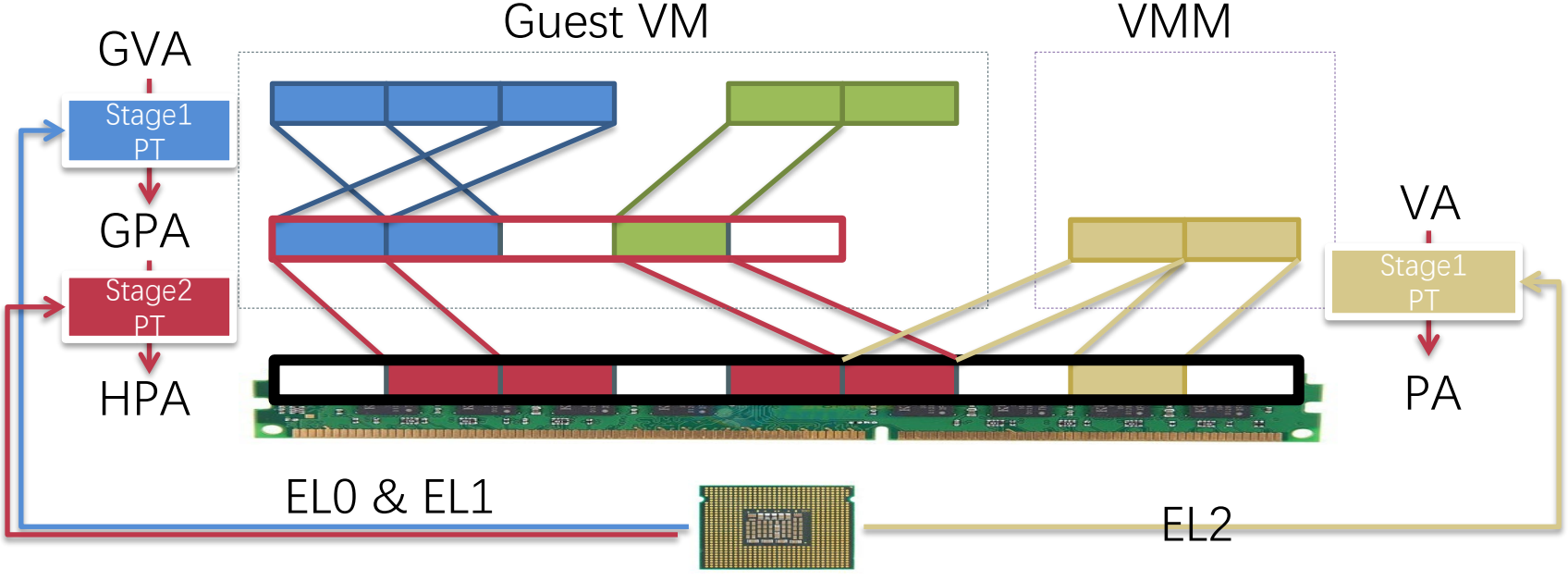
- 影子页表(Shadow Page Table)
- 直接页表(Direct Page Table)
- **硬件虚拟化**

硬件虚拟化对内存翻译的支持

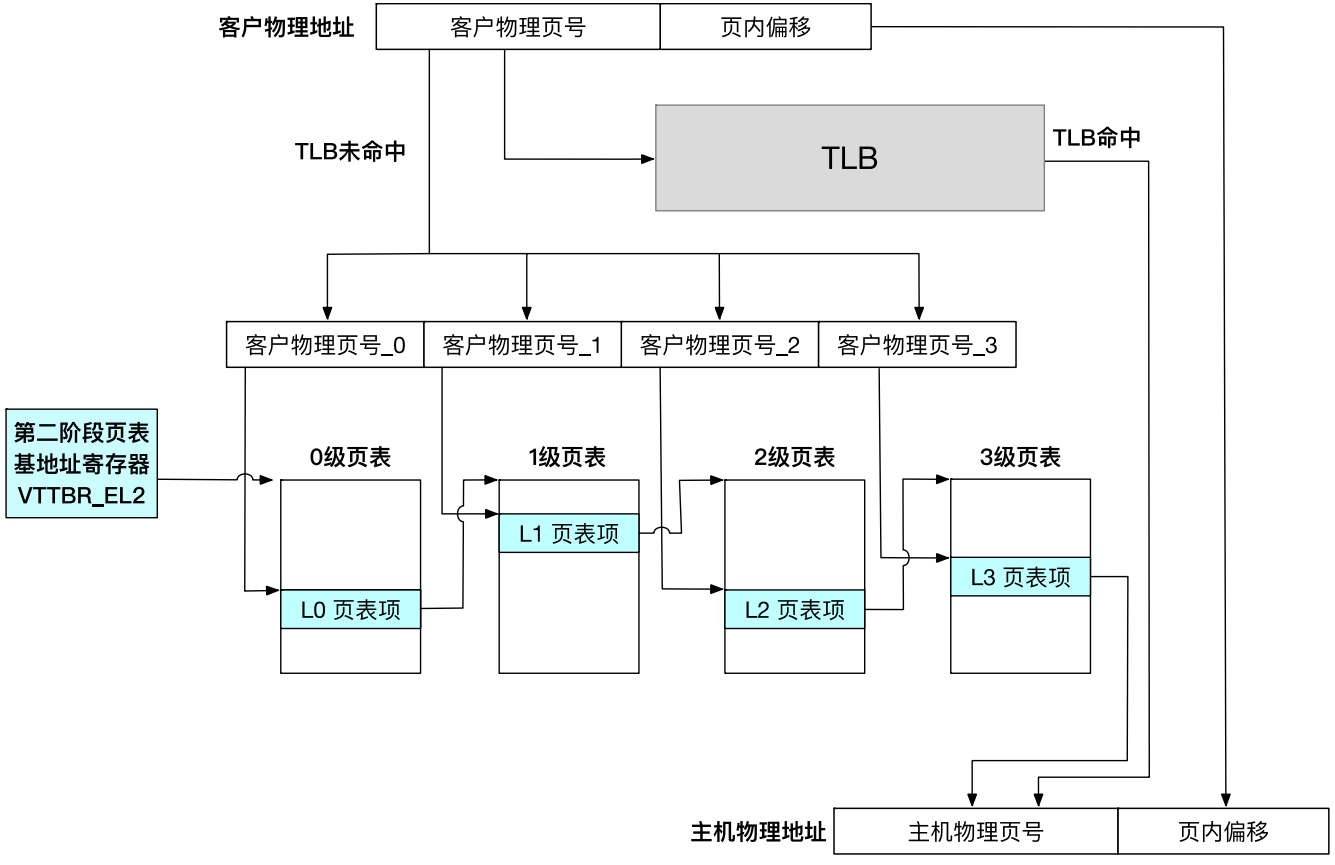
- **Intel VT-x和ARM硬件虚拟化都有对应的内存虚拟化**
  - Intel Extended Page Table (EPT)
  - ARM Stage-2 Page Table (第二阶段页表)
- **新的页表**
  - 将GPA翻译成HPA
  - 此表被VMM直接控制
  - 每一个VM有一个对应的页表

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

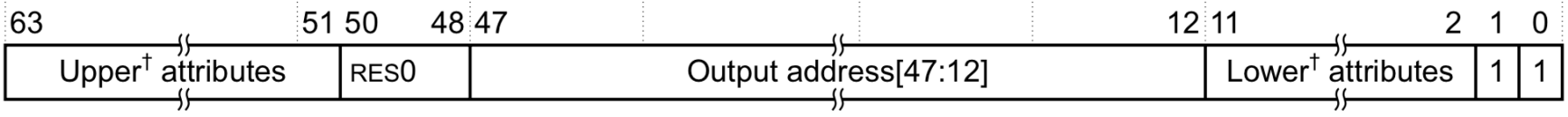

客户物理地址
客户物理页号
页内偏移
TLB未命中
TLB
TLB命中
客户物理页号_0
客户物理页号_1
客户物理页号_2
客户物理页号_3
第二阶段页表
基地址寄存器
VTTBR_EL2
0级页表
1级页表
2级页表
3级页表
L0 页表项
L1 页表项
L2 页表项
L3 页表项
主机物理地址
主机物理页号
页内偏移

VTTBR_EL2

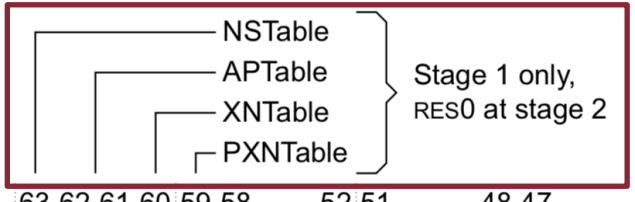
- **存储虚拟机第二阶段页表基地址**
  - 只有1个寄存器：VTTBR_EL2
- **对比第一阶段页表**
  - 有2个页表基地址寄存器：TTBR0_EL1、TTBR1_EL1
- **VMM在调度VM之前需要在VTTBR_EL2中写入此VM的第二阶段页表基地址**
- **第二阶段页表使能**
  - HCR_EL2第0位

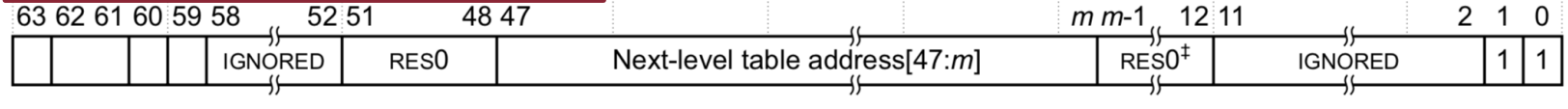
第二阶段页表项

- **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

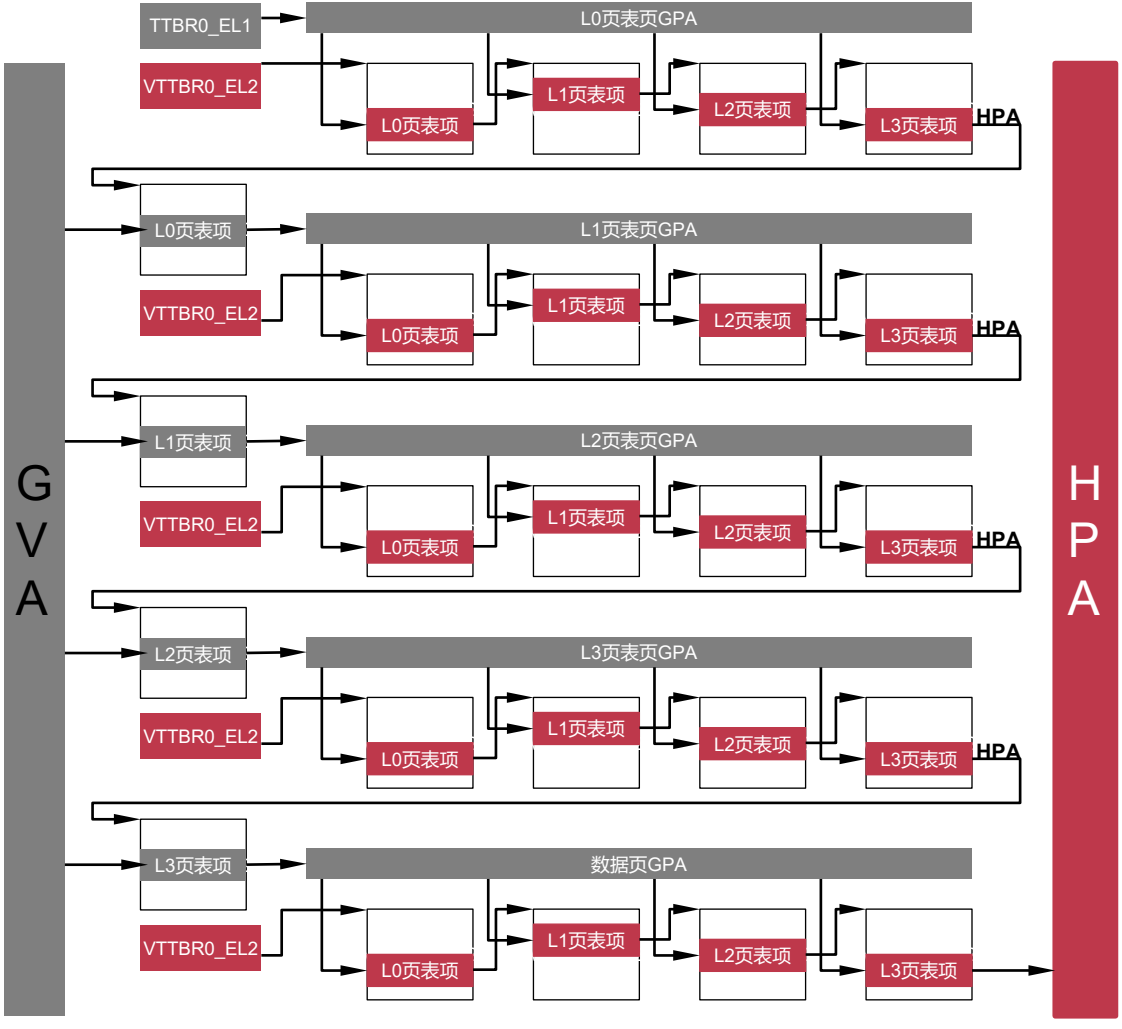




- **NSTable: 与TrustZone有关**
- **APTable：读写权限**
- **XNTable：执行权限**
- **PXNTable：特权级别软件的执行权限**

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

I/O Virtualization

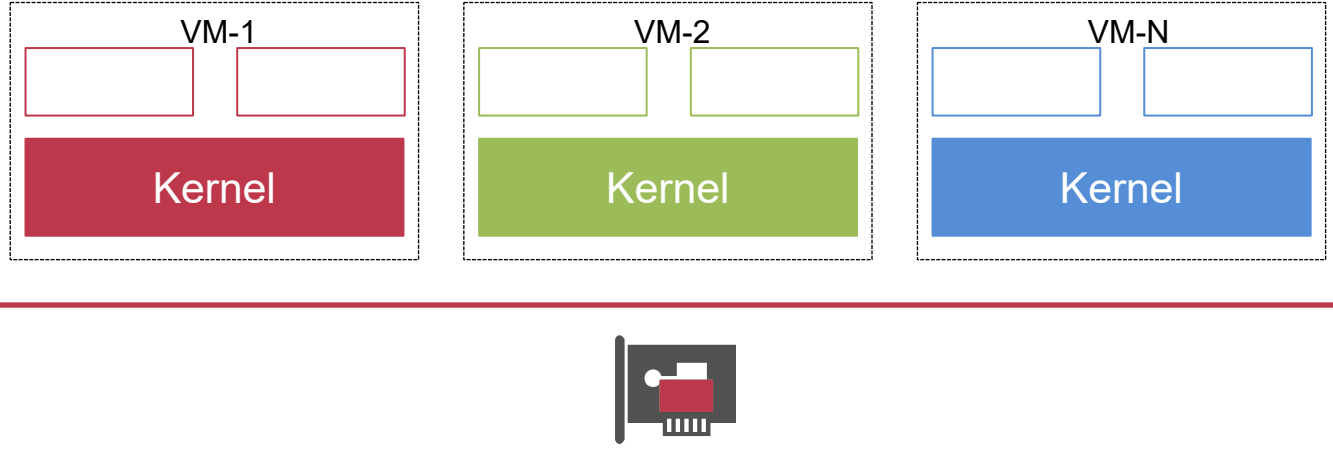
I/O虚拟化

为什么需要IO虚拟化

- **回顾：操作系统内核直接管理外部设备**
  - PIO
  - MMIO
  - DMA
  - Interrupt
- **如果VM能直接管理物理设备**
  - 会发生什么?

如果VM直接管理物理网卡

- **正确性问题：所有VM都直接访问网卡**
  - 所有VM都有相同的MAC地址、IP地址，无法正常收发网络包
- **安全性问题：恶意VM可以直接读取其他VM的数据**
  - 除了直接读取所有网络包，还可能通过DMA访问其他内存



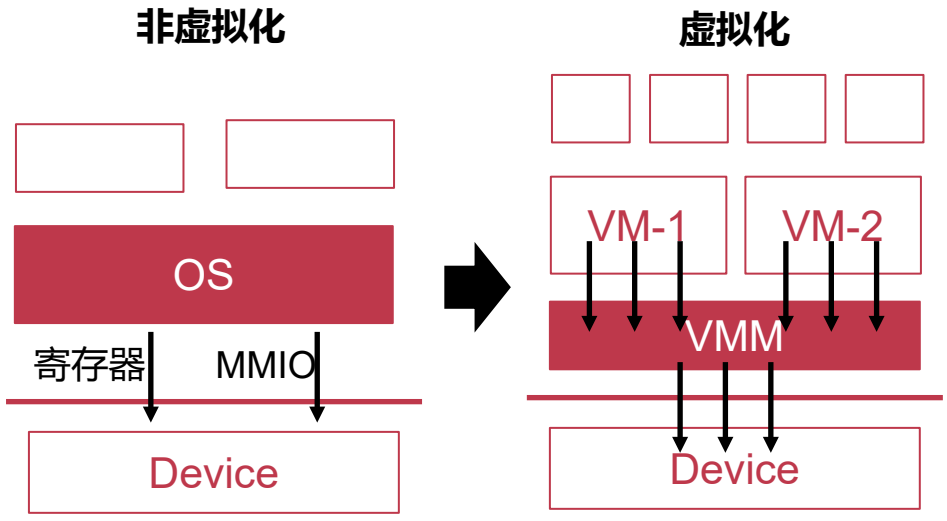
I/O虚拟化的目标

- **为虚拟机提供虚拟的外部设备**
  - 虚拟机正常使用设备
- **隔离不同虚拟机对外部设备的直接访问**
  - 实现I/O数据流和控制流的隔离
- **提高物理设备的利用资源**
  - 多个VM同时使用，可以提高物理设备的资源利用率

怎么实现I/O虚拟化?

- 设备模拟
- 半虚拟化方式
- 设备直通

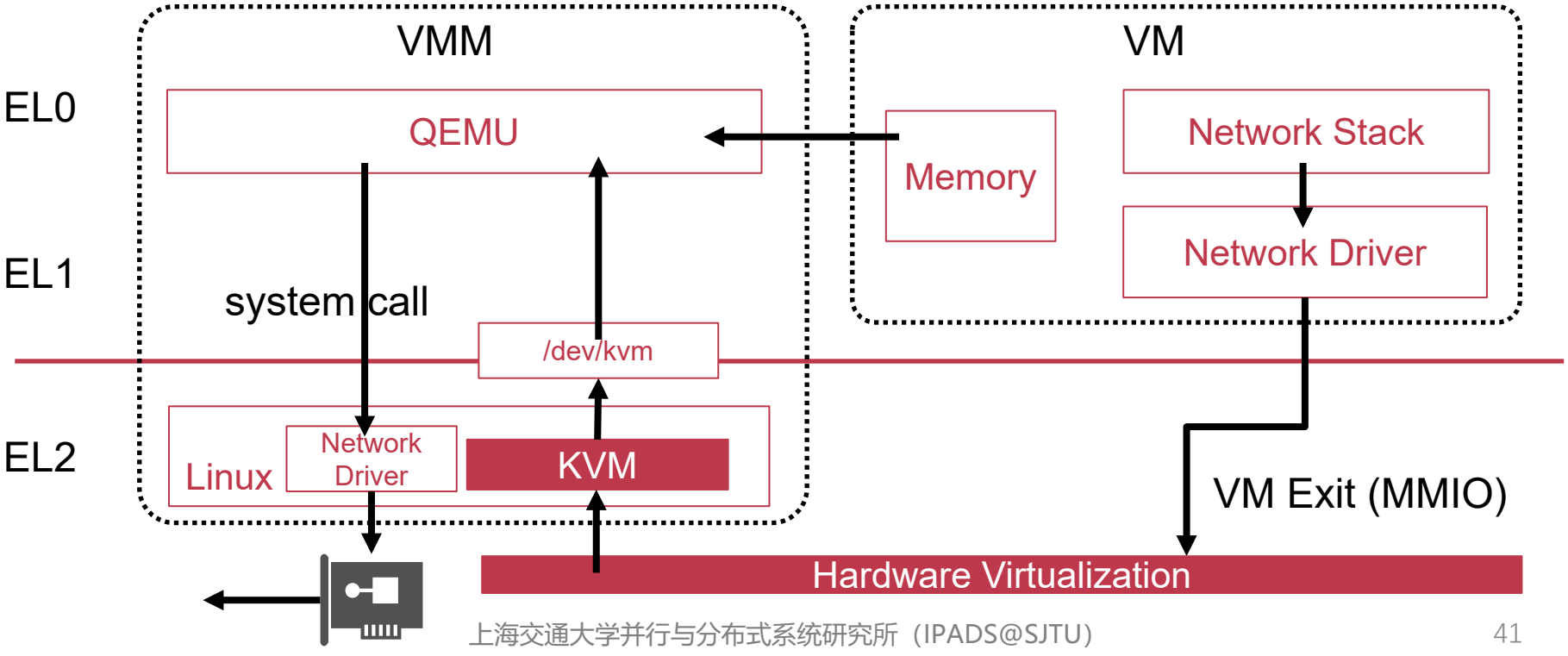
方法1：设备模拟



- **OS与设备交互的硬件接口**
  - 模拟寄存器(中断等)
  - 捕捉MMIO操作
- **硬件虚拟化的方式**
  - 硬件虚拟化捕捉PIO指令
  - MMIO对应内存在第二阶段页表中设置为invalid

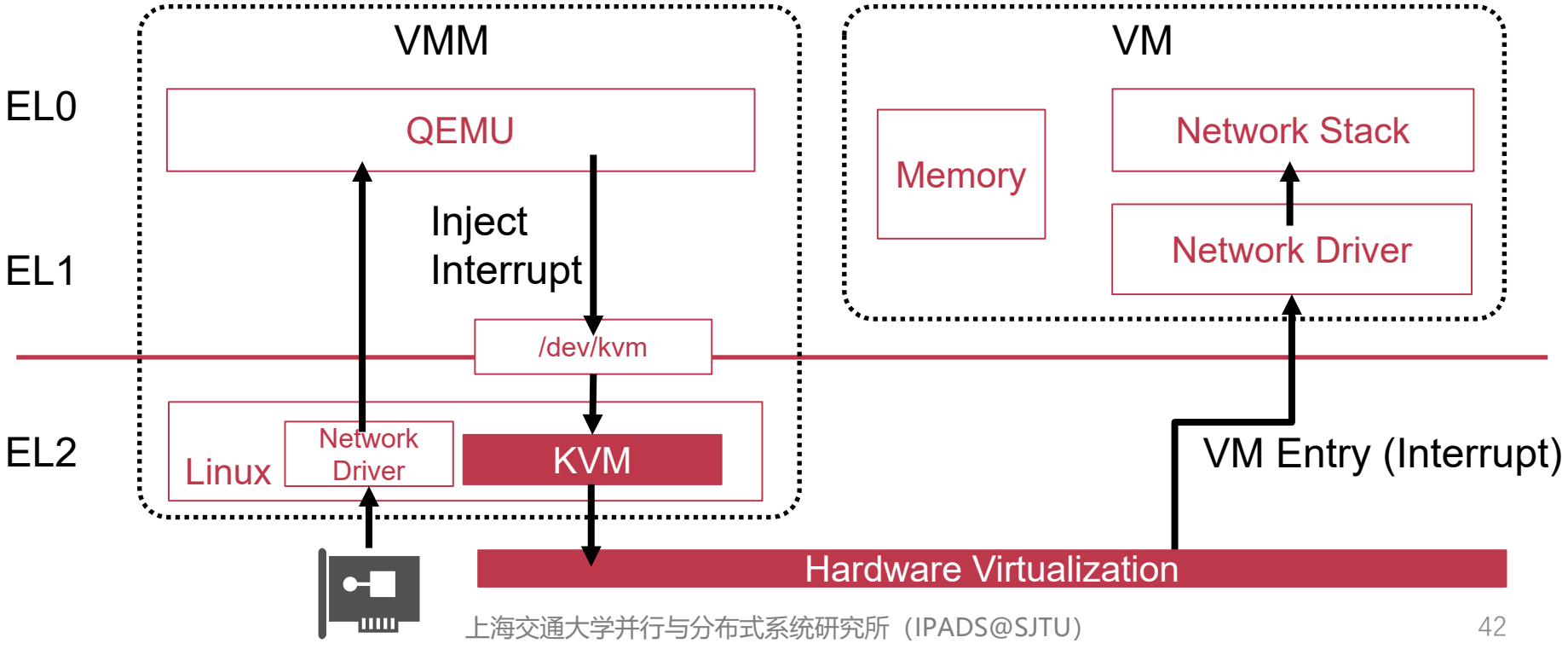
例：QEMU/KVM设备模拟1

- 以虚拟网卡举例——发包过程



例：QEMU/KVM设备模拟2

- 以虚拟网卡举例——收包过程



设备模拟的优缺点

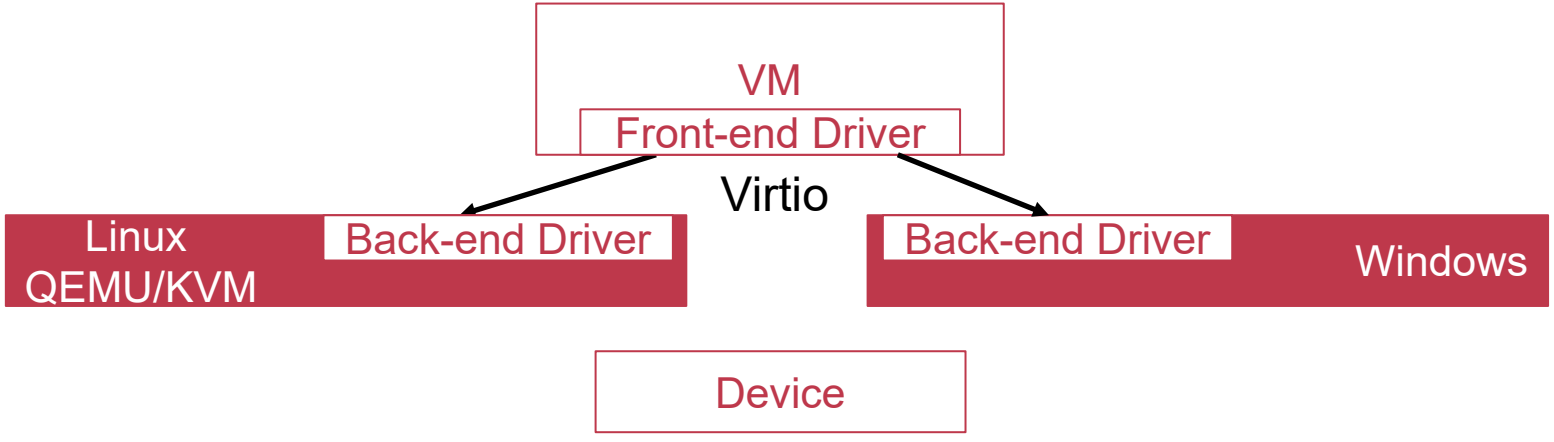
- **优点**
  - 可以模拟多种设备
    - 因而可以支持较“久远”的OS
  - 允许在中间拦截（Interposition）：
    - 例如在QEMU层面检查网络内容
  - 不需要硬件虚拟化
- **缺点**
  - 性能不佳

方法2：半虚拟化方式

- **协同设计**
  - 虚拟机“知道”自己运行在虚拟化环境
  - 虚拟机内运行前端(front-end)驱动
  - VMM内运行后端(back-end)驱动
- **VMM主动提供Hypercall给VM**
- **通过共享内存传递指令和命令**

VirtIO: Unified Para-virtualized I/O

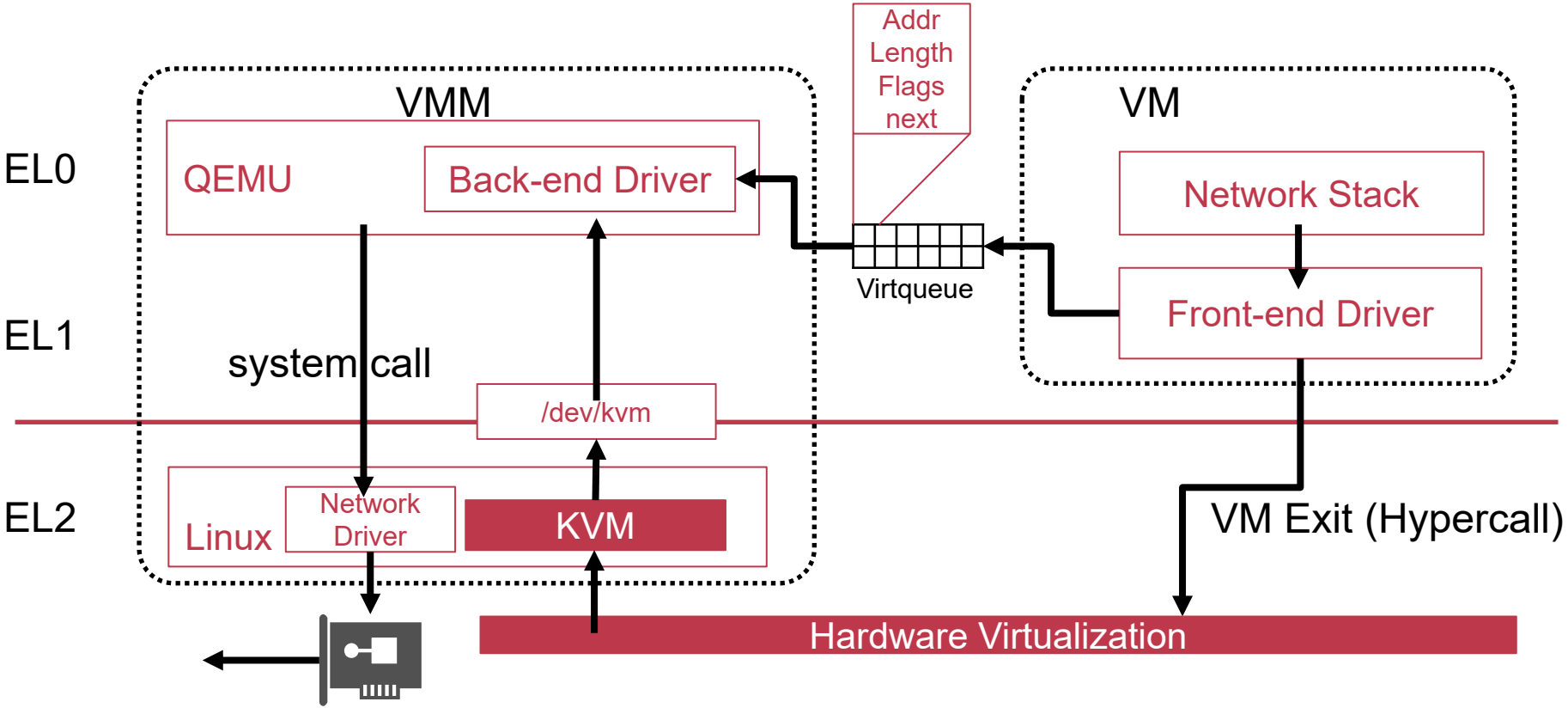
- **标准化的半虚拟化I/O框架**
  - 通用的前端抽象
  - 标准化接口
  - 增加代码的跨平台重用



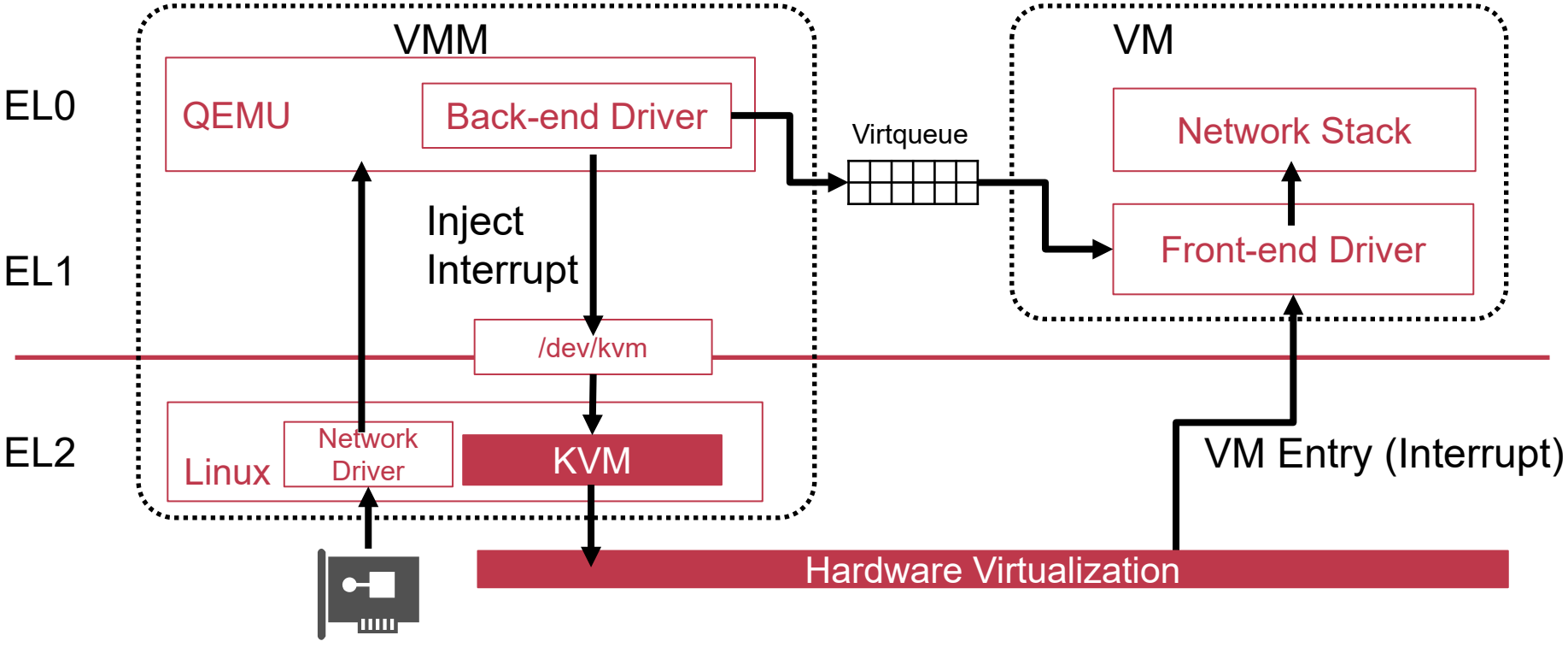
Virtqueue

- **VM和VMM之间传递I/O请求的队列**
- **3个部分**
  - Descriptor Table
    - 其中每一个descriptor描述了前后端共享的内存
    - 链表组织
  - Available Ring
    - 可用descriptor的索引，Ring Entry指向一个descriptor链表
  - Used Ring
    - 已用descriptor的索引

例：QEMU/KVM半虚拟化1



例：QEMU/KVM半虚拟化2

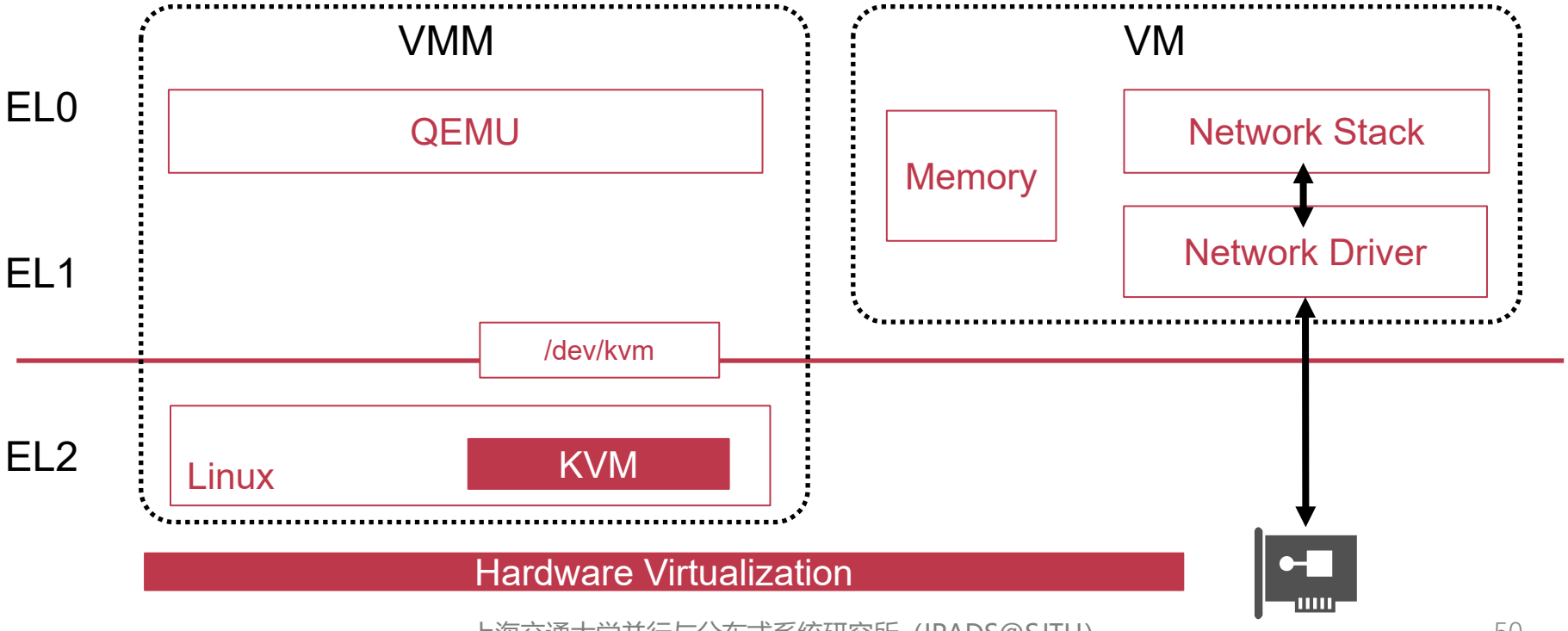


半虚拟化方式的优缺点

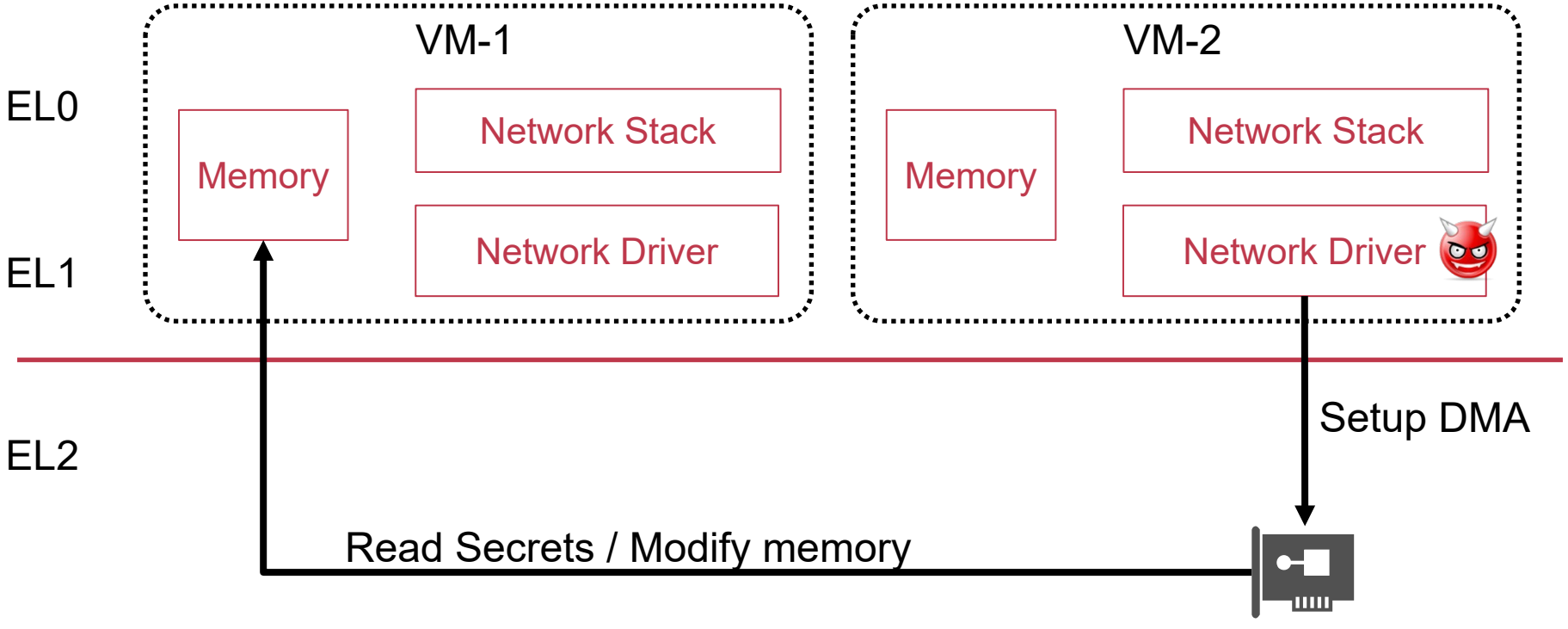
- **优点**
  - 性能优越
    - 多个MMIO/PIO指令可以整合成一次Hypercall
  - VMM实现简单，不再需要理解物理设备接口
- **缺点**
  - 需要修改虚拟机操作系统内核

方法3：设备直通

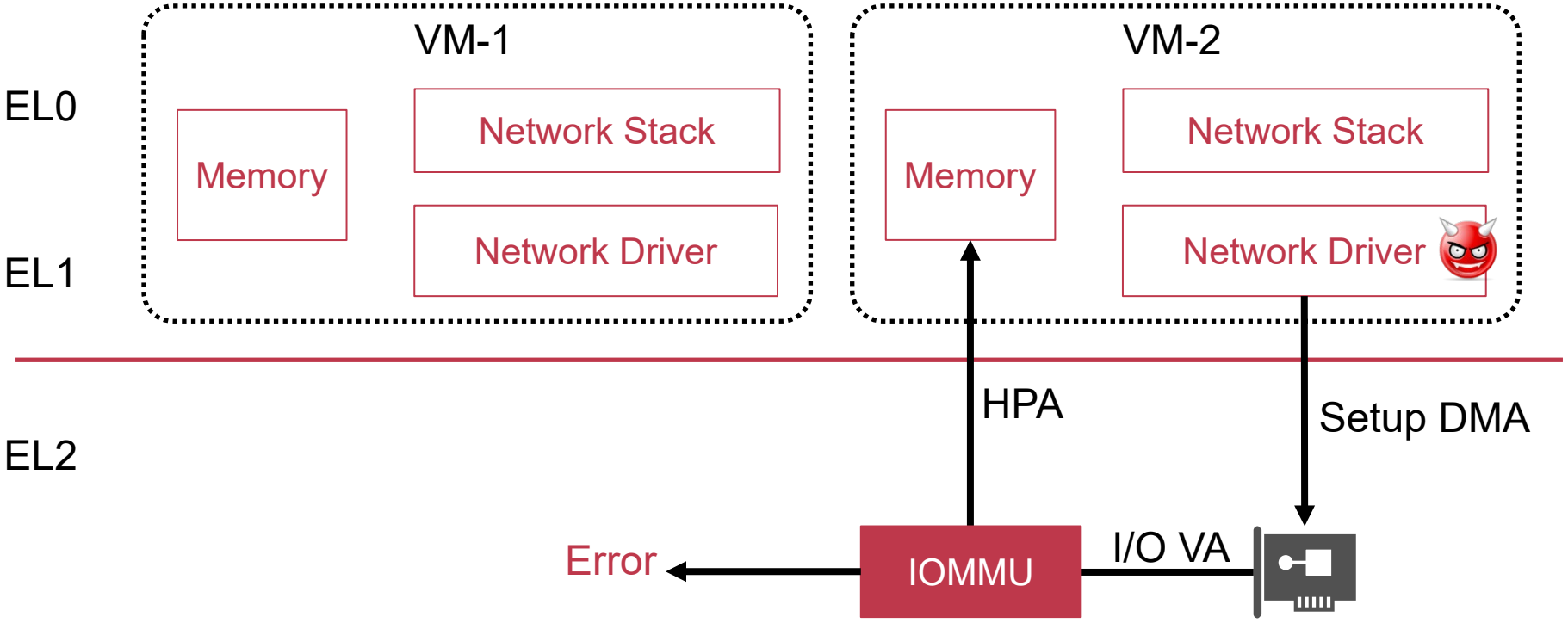
- 虚拟机直接管理物理设备



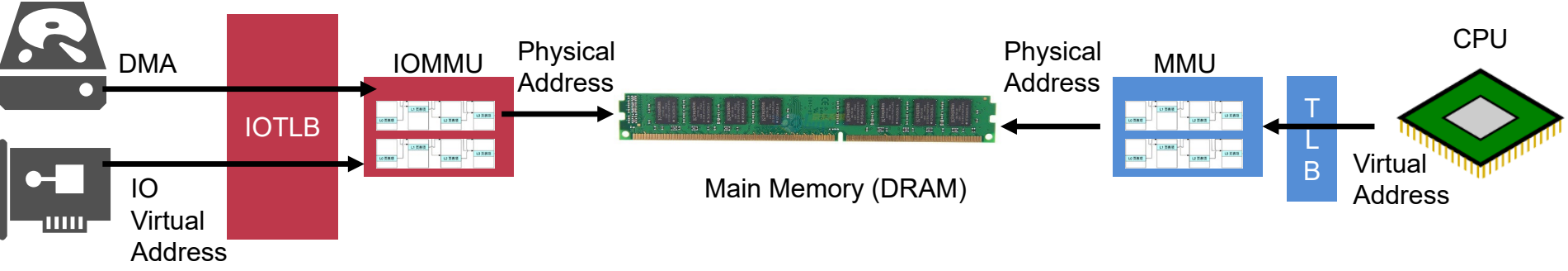
问题1：DMA恶意读写内存



使用IOMMU


VM-1
VM-2
EL0
EL1
EL2
Memory
Network Stack
Network Driver
Memory
Network Stack
Network Driver
HPA
Setup DMA
Error
IOMMU
I/O VA

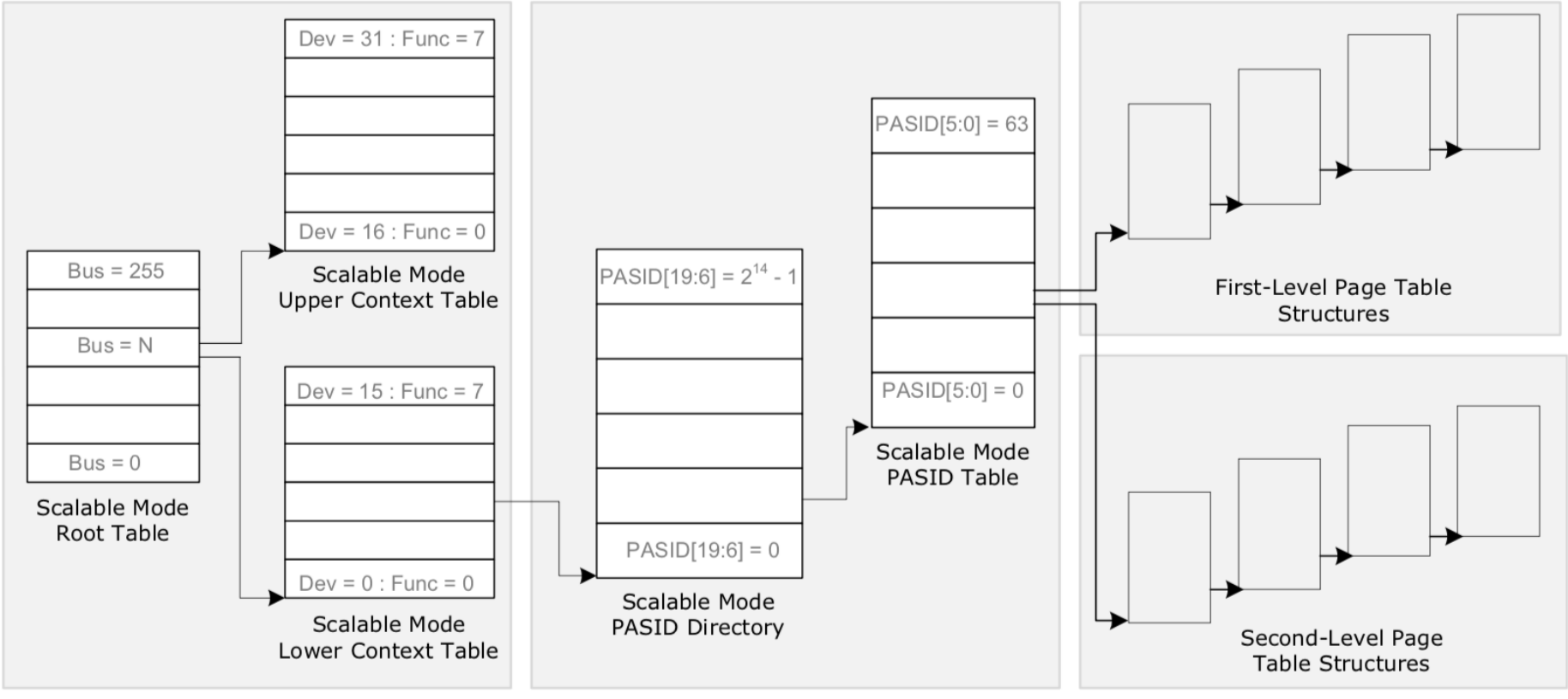
IOMMU与MMU


DMA
IO
Virtual
Address
IOTLB
IOMMU
Physical
Address
Main Memory (DRAM)
Physical
Address
MMU
T
L
B
CPU
Virtual
Address

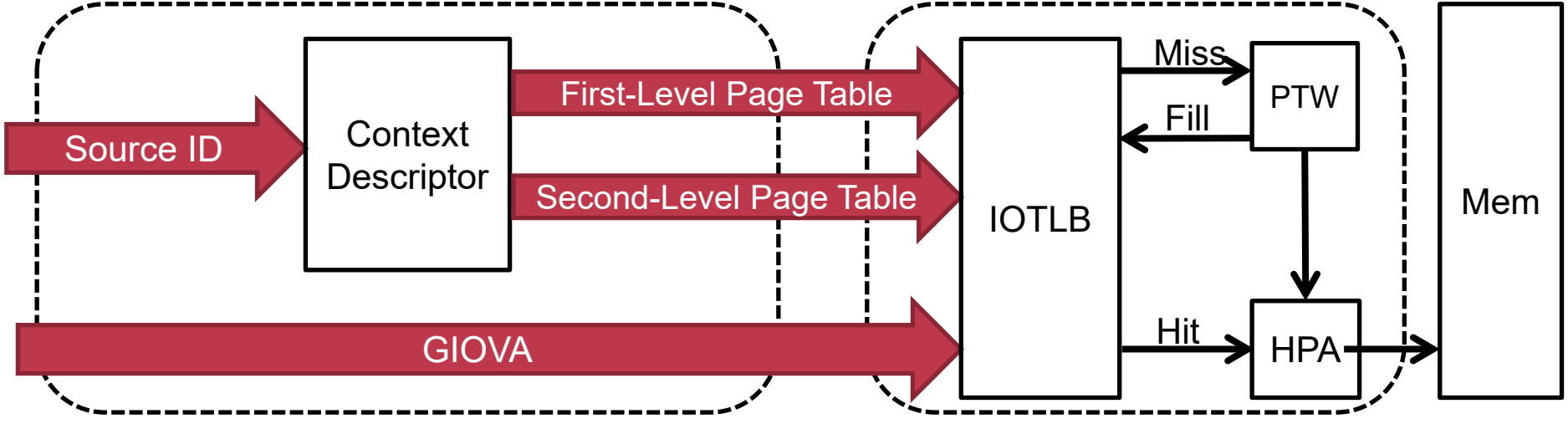
ARM SMMU

- **SMMU是ARM中IOMMU的实现**
  - System MMU
- **SMMU的设计与AARCH64 MMU一致**
  - 也存在两阶段地址翻译
  - 第一阶段：OS为进程配置：IOVA->GPA
  - 第二阶段：第一阶段翻译完之后进行第二阶段
    - VMM为VM配置：GPA->HPA

SMMU的页表


Dev = 31 : Func = 7
Dev = 16 : Func = 0
Scalable Mode
Upper Context Table
Bus = 255
Bus = N
Bus = 0
Scalable Mode
Root Table
Dev = 15 : Func = 7
Dev = 0 : Func = 0
Scalable Mode
Lower Context Table
PASID[19:6] = 2^{14} - 1
PASID[19:6] = 0
Scalable Mode
PASID Directory
PASID[5:0] = 63
PASID[5:0] = 0
Scalable Mode
PASID Table
First-Level Page Table
Structures
Second-Level Page
Table Structures

ARM SMMU翻译过程



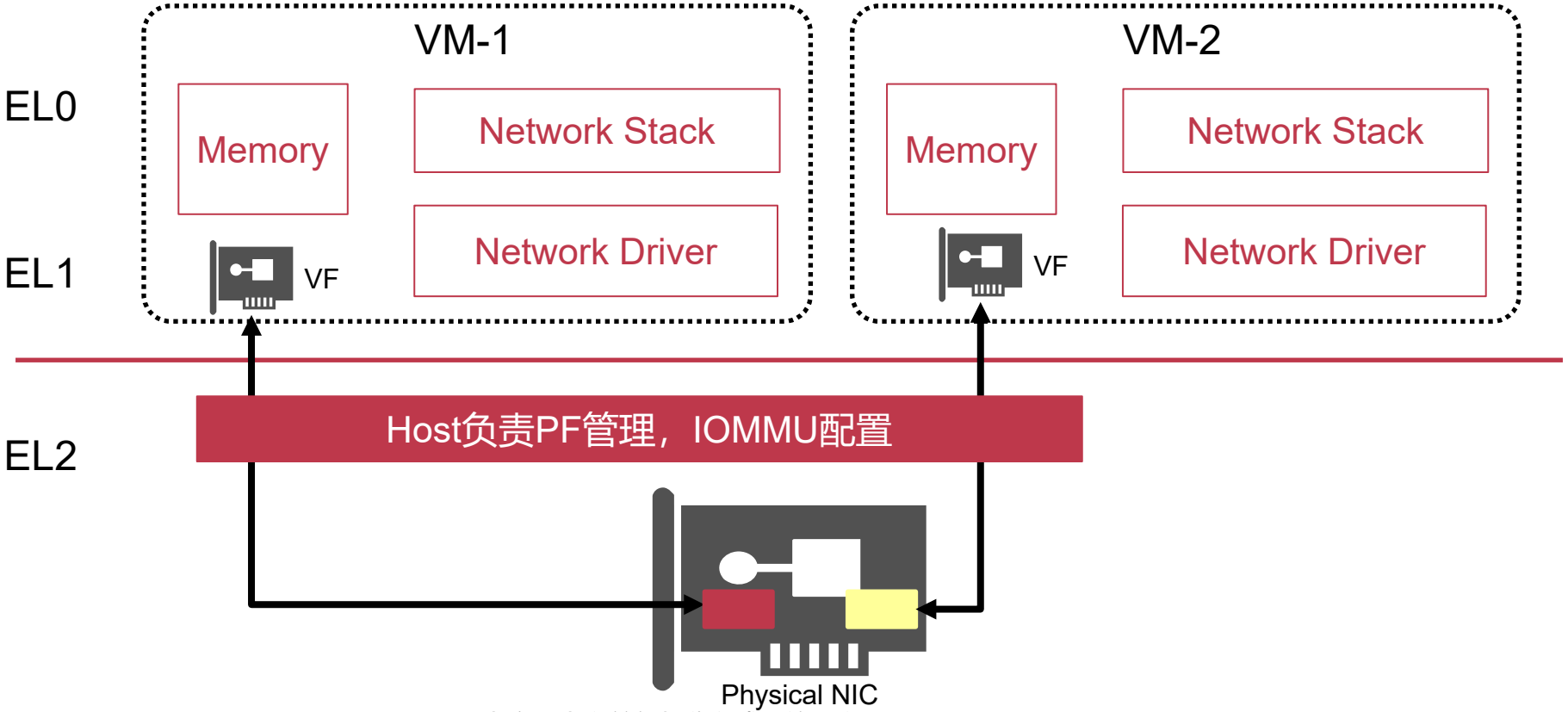
问题2：设备独占

- **Scalability不够**
  - 设备被VM-1独占后，就无法被VM-2使用
- **如果一台物理机上运行16个虚拟机**
  - 必须为这些虚拟机安装16个物理网卡

Single Root I/O Virtualization (SRIOV)

- **SR-IOV是PCI-SIG组织确定的标准**
- **满足SRIOV标准的设备，在设备层实现设备复用**
  - 能够创建多个Virtual Function(VF)，每一个VF分配给一个VM
    - 负责进行数据传输，属于数据面（Data-plane）
  - 物理设备被称为Physical Function(PF)，由Host管理
    - 负责进行配置和管理，属于控制面（Control-plane）
- **设备的功能**
  - 确保VF之间的数据流和控制流彼此不影响

SRIOV的使用


VM-1
VM-2
EL0
EL1
EL2
Memory
Network Stack
Network Driver
VF
Memory
Network Stack
Network Driver
VF
Host负责PF管理，IOMMU配置
Physical NIC

设备直通的优缺点

- **优点**
  - 性能优越
  - 简化VMM的设计与实现
- **缺点**
  - 需要特定硬件功能的支持（IOMMU、SRIOV等）
  - 不能实现Interposition：难以支持虚拟机热迁移

I/O虚拟化技术对比

	设备模拟	半虚拟化	设备直通
性能	差	中	好
修改虚拟机内核	否	驱动+修改	安装VF驱动
VMM复杂度	高	中	低
Interposition	有	有	无
是否依赖硬件功能	否	否	是
支持老版本OS	是	否	否

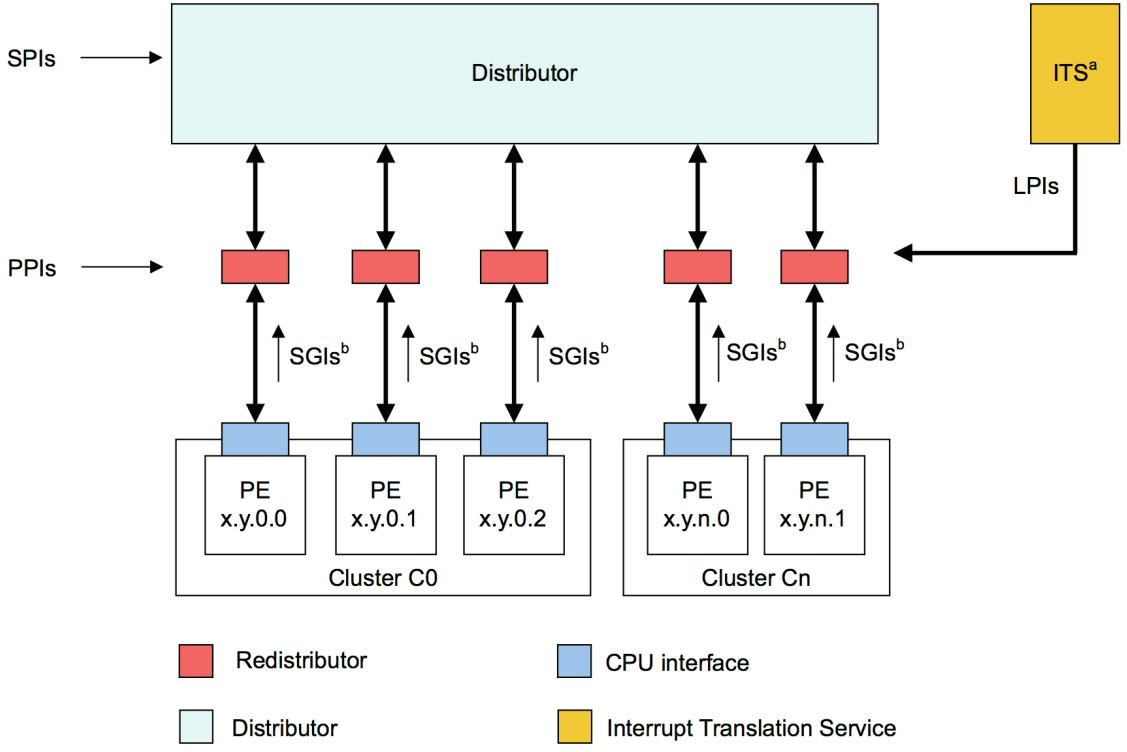
中断虚拟化

- **VMM在完成I/O操作后通知VM**
  - 例如在DMA操作之后
- **VMM在VM Entry时插入虚拟中断**
  - VM的中断处理函数会被调用
- **虚拟中断类型**
  - 时钟中断
  - 核间中断
  - 外部中断

ARM中断虚拟化的实现方法

- **打断虚拟机执行**
  - 通过List Register插入
- **不打断虚拟机执行**
  - 通过GIC ITS插入

回顾GIC

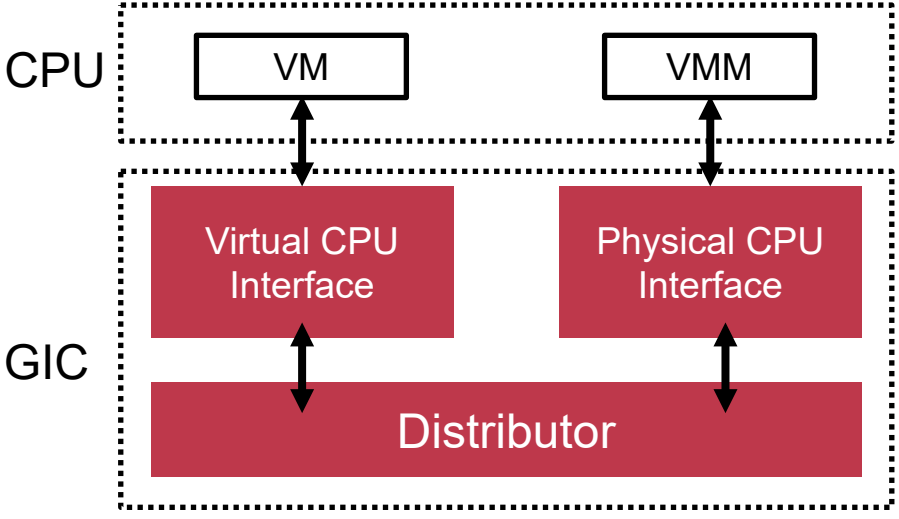


a. The inclusion of an ITS is optional, and there might be more than one ITS in a GIC.
b. SGIs are generated by a PE and routed through the Distributor.

Figure 3-2 GIC logical partitioning with an ITS

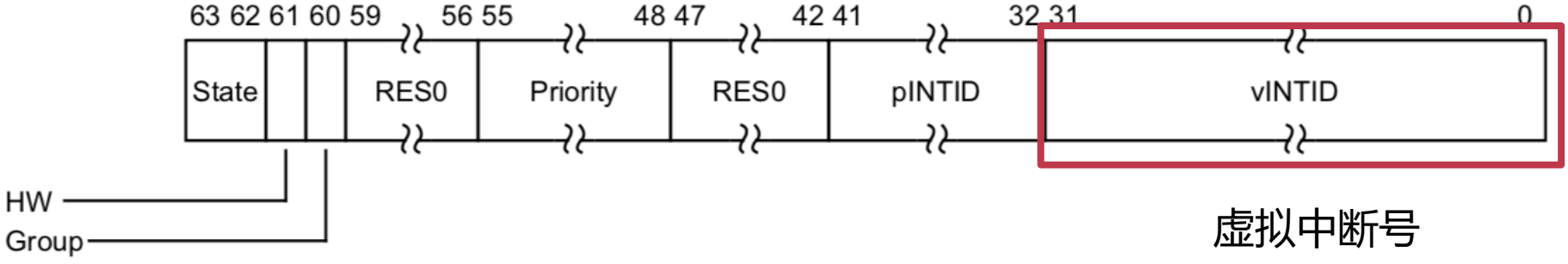
Virtual CPU Interface

- **GIC为虚拟机提供的硬件功能**
  - VM通过Virtual CPU Interface与GIC交互
  - VMM通过Physical CPU Interface与GIC交互



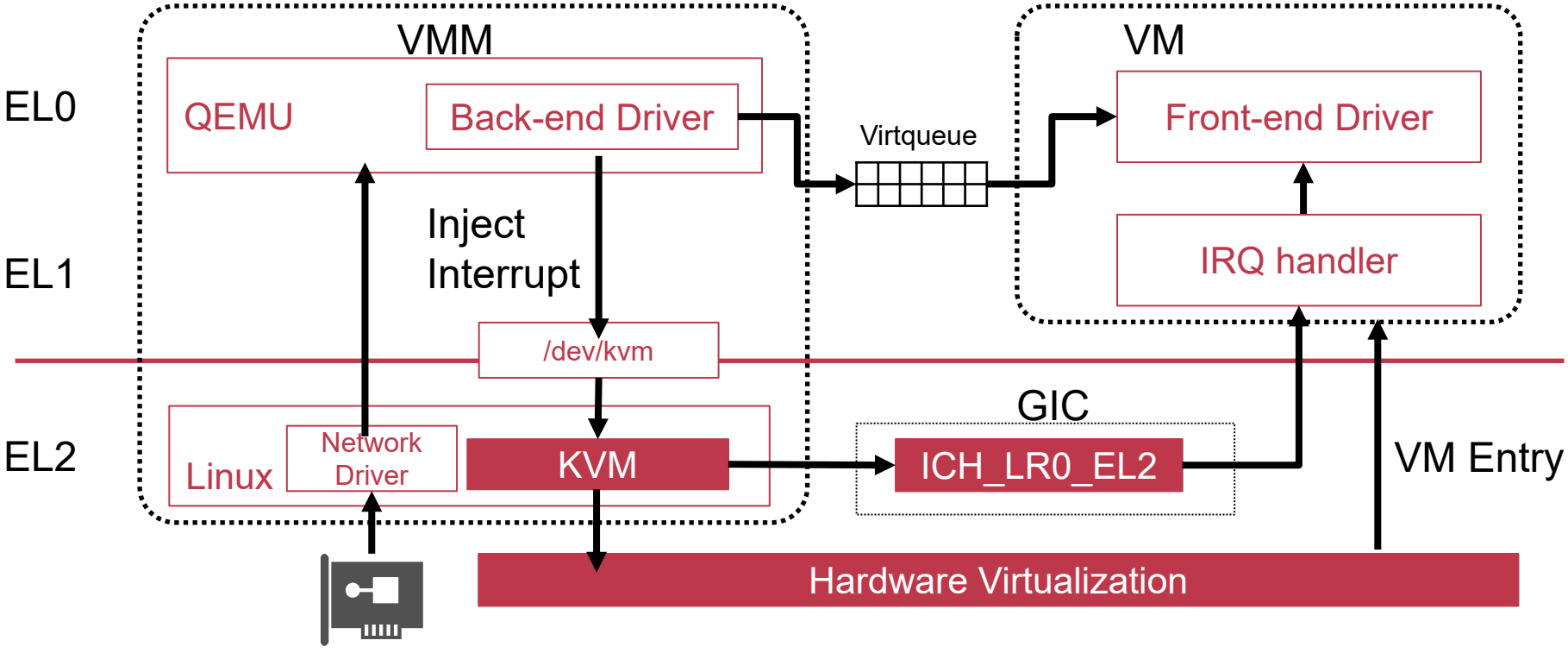
插入虚拟中断

- **通过GIC List Register插入中断**
  - 共有16个List Register
  - ICH_LR<n>_EL2 (n = 0…15)



插入虚拟中断：以半虚拟化举例

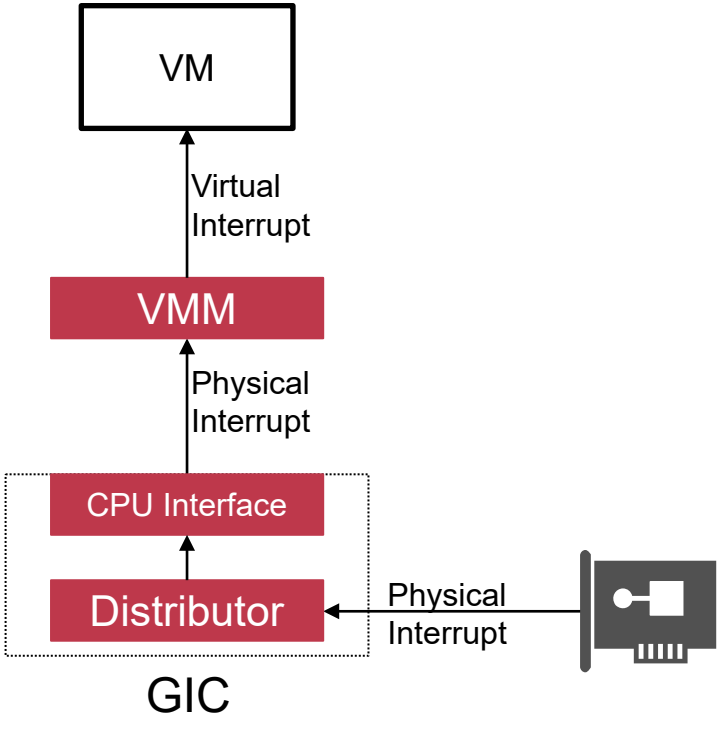
思考：这种方式有什么问题？



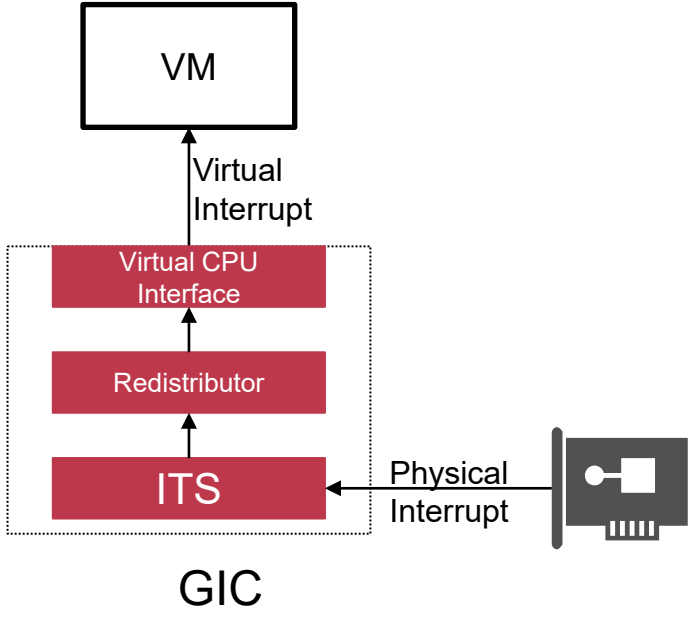
不打断虚拟机执行：GIC ITS

- **GIC第4版本推出了Direct injection of virtual interrupts**
  - 将物理设备的物理中断与虚拟中断绑定
  - 物理设备直接向虚拟机发送虚拟中断
- **VMM在运行VM前**
  - 配置GIC ITS (Interrupt Translation Service)
    - 建立物理中断与虚拟中断的映射
  - 映射内容
    - 设备与物理中断的映射
    - 分配虚拟中断号
    - 发送给哪些物理核上的虚拟处理器

虚拟中断的直接插入



GICv3中的物理中断插入



GICv4中的物理中断插入

下次课内容

- **轻量级虚拟化**