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

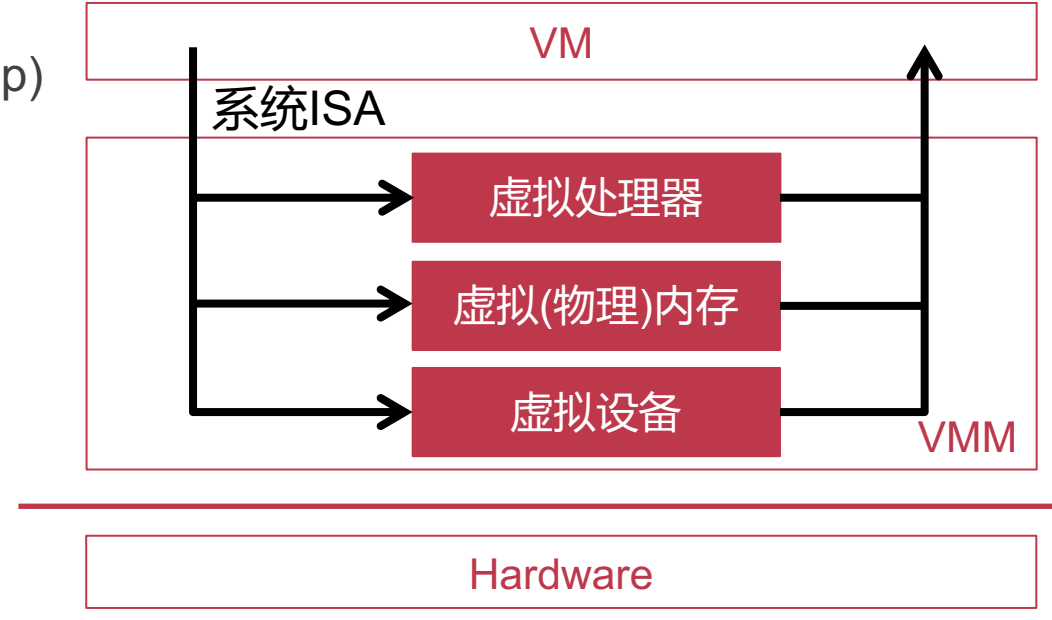
https://ipads.se.sjtu.edu.cn

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

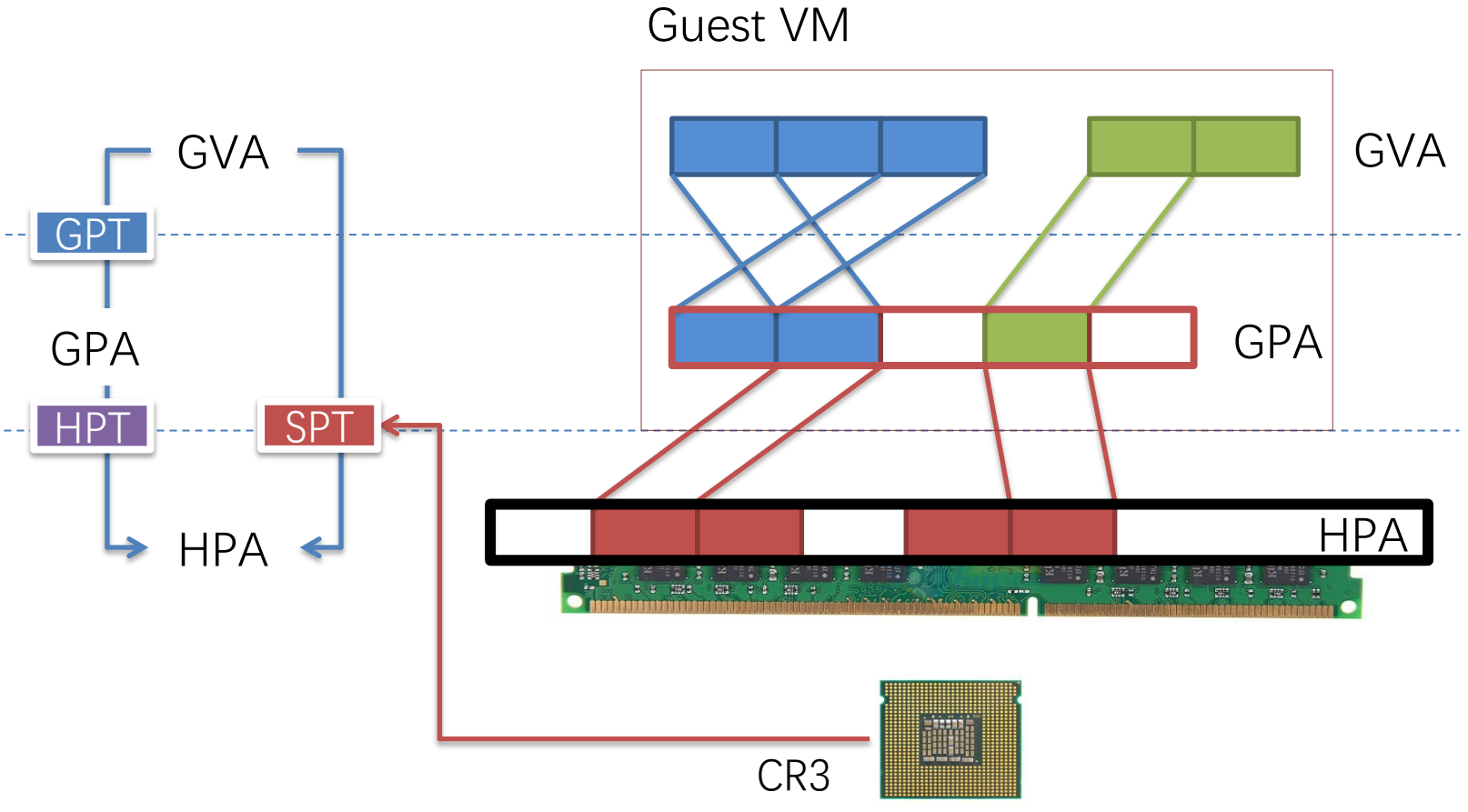


Review: 如何处理这些不会下陷的敏感指令？

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：影子页表

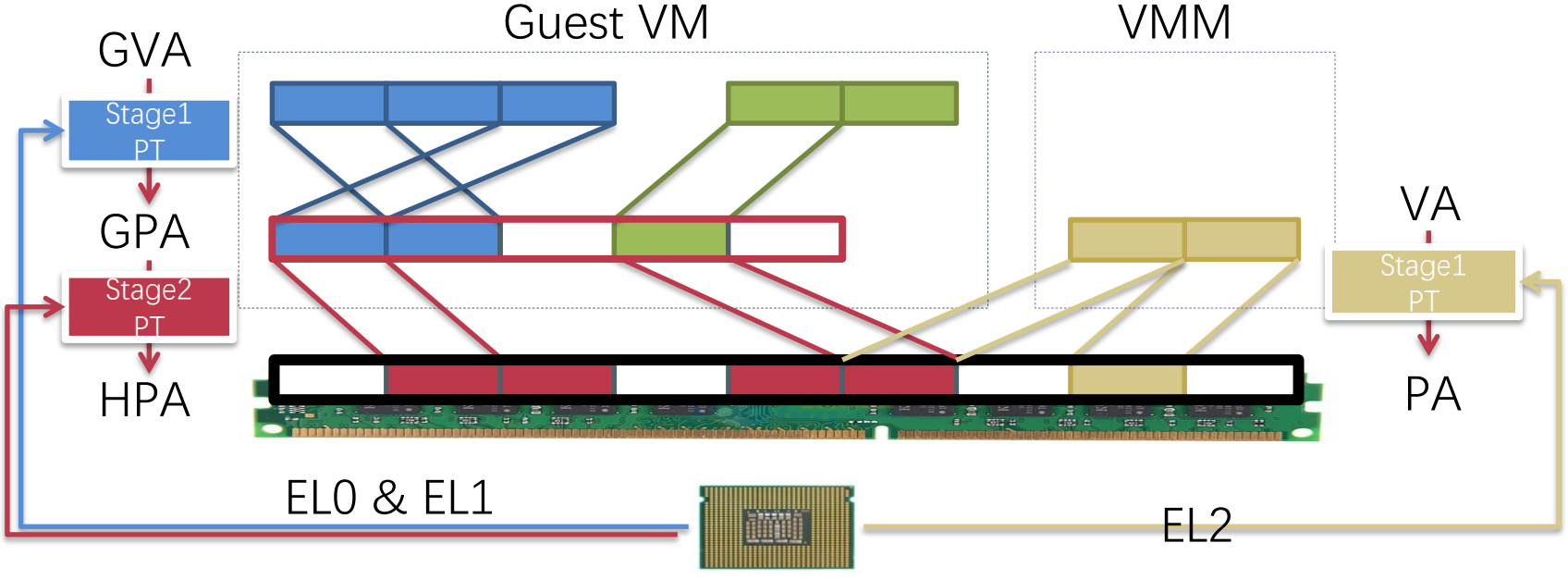


Review：Direct Paging (Para-virtualization)

- **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

Review：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）



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？

I/O Virtualization

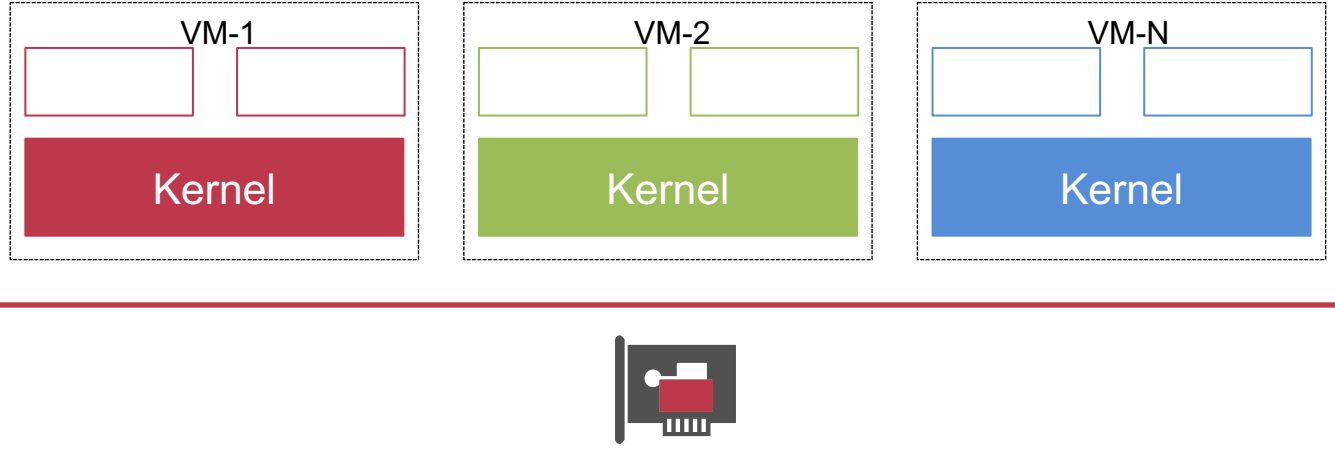
I/O虚拟化

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

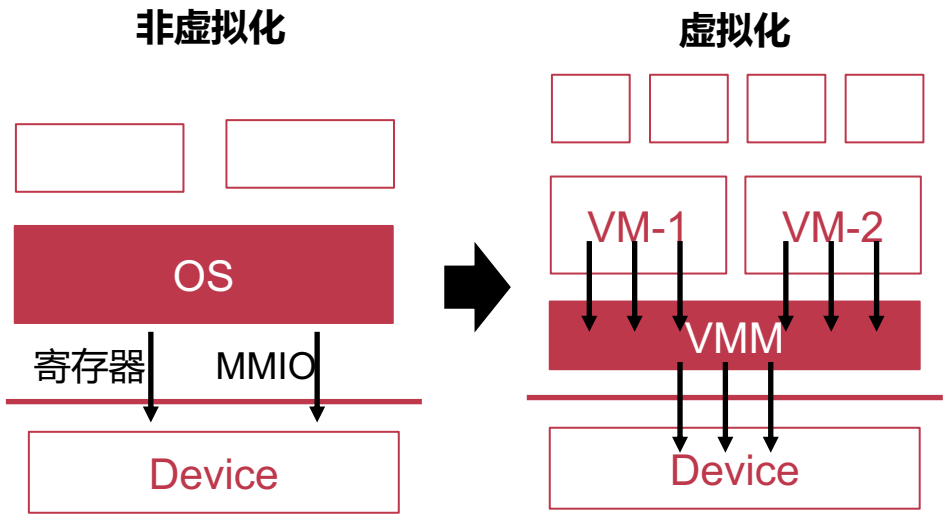
- **为虚拟机提供虚拟的外部设备**
  - 虚拟机正常使用设备
- **隔离不同虚拟机对外部设备的直接访问**
  - 实现I/O数据流和控制流的隔离
- **提高物理设备的利用资源**
  - 多个VM同时使用，可以提高物理设备的资源利用率

怎么实现I/O虚拟化?

- 1、设备模拟（Emulation）
- 2、半虚拟化方式（Para-virtualization）
- 3、设备直通（Pass-through）

方法1：设备模拟

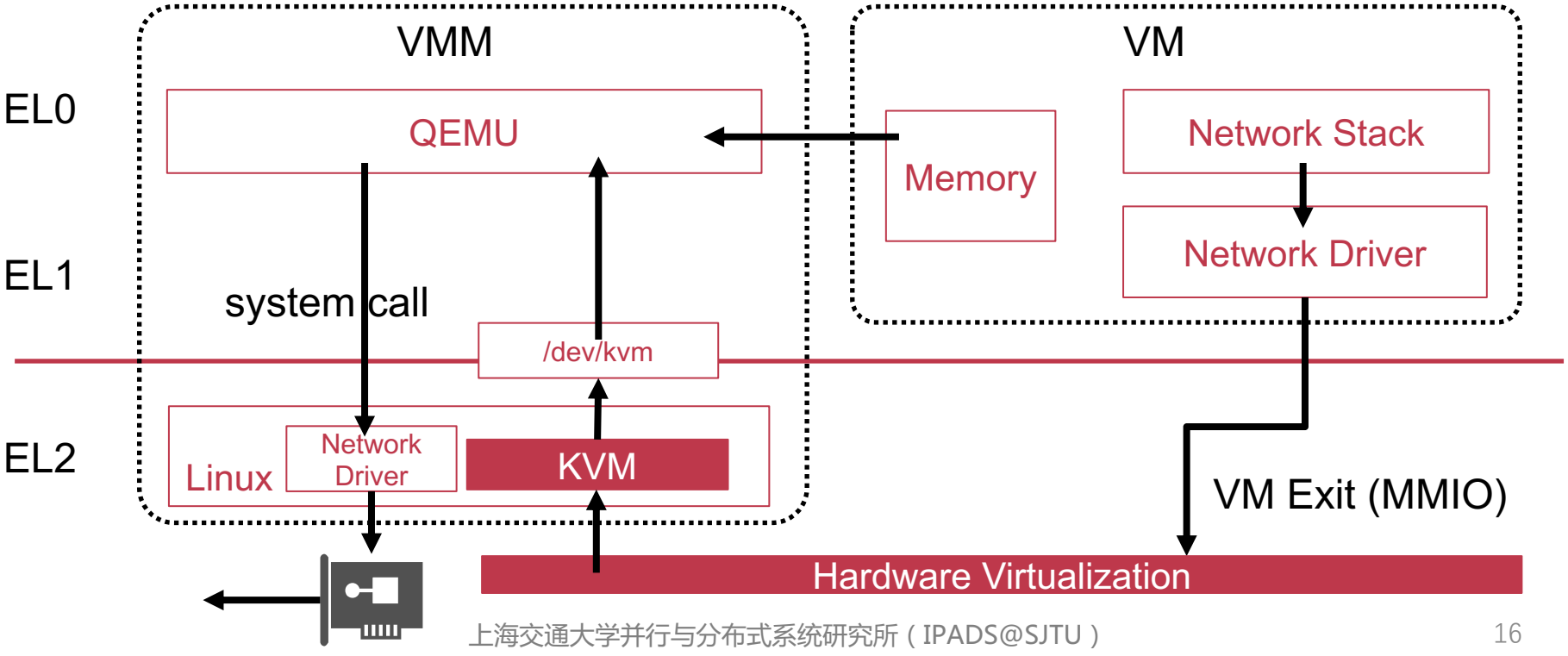
- **OS与设备交互的硬件接口**
  - 模拟寄存器(中断等)
  - 捕捉MMIO操作



- **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

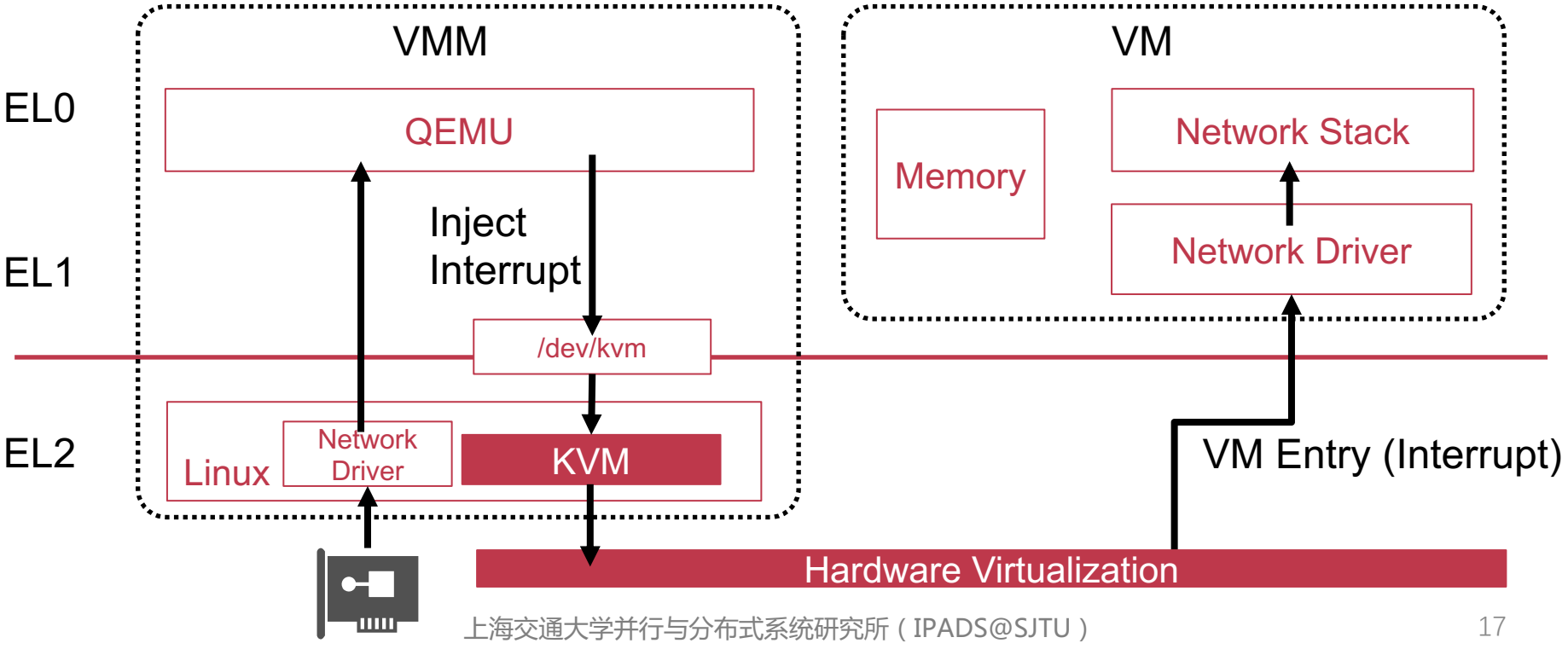
例：QEMU/KVM设备模拟1

- 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

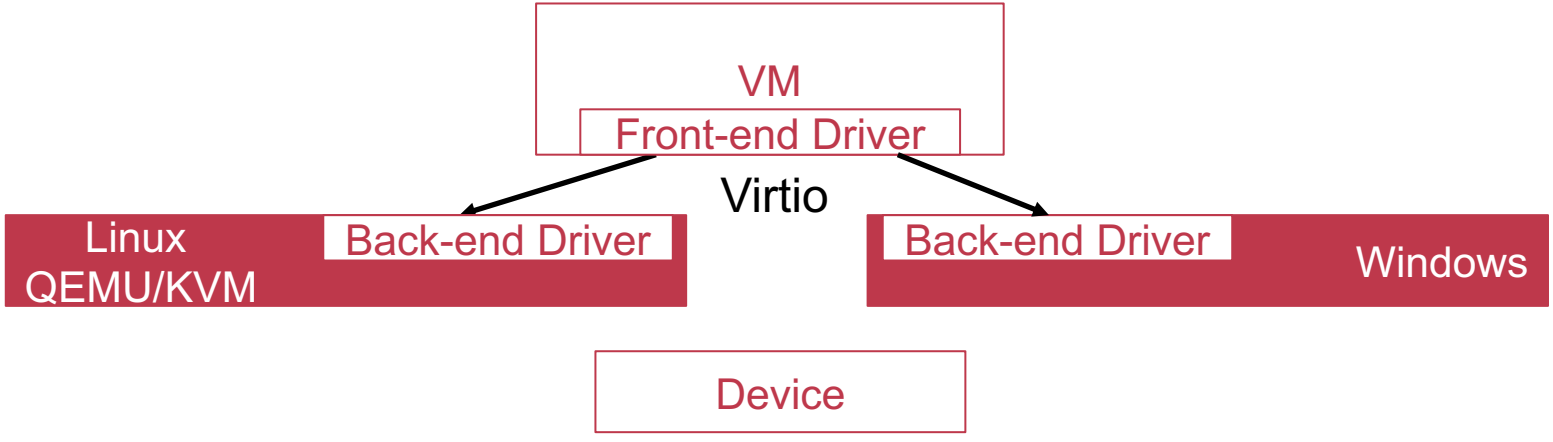
- **优点**
  - 可以模拟任意设备
    - 选择流行设备，支持较“久远”的OS（如e1000网卡）
  - 允许在中间拦截（Interposition）：
    - 例如在QEMU层面检查网络内容
  - 不需要硬件修改
- **缺点**
  - 性能不佳

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

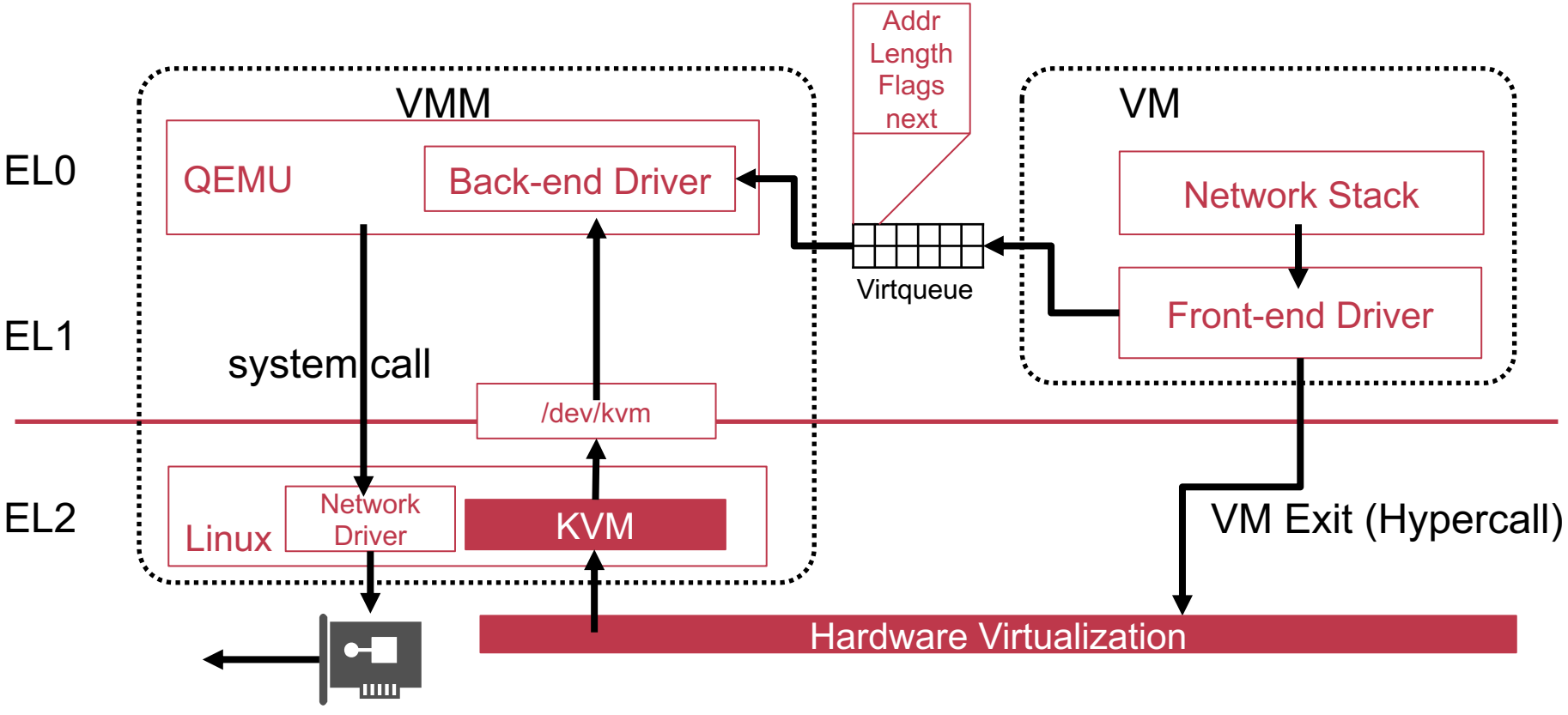
- **标准化的半虚拟化I/O框架**
  - 通用的前端抽象
  - 标准化接口
  - 增加代码的跨平台重用



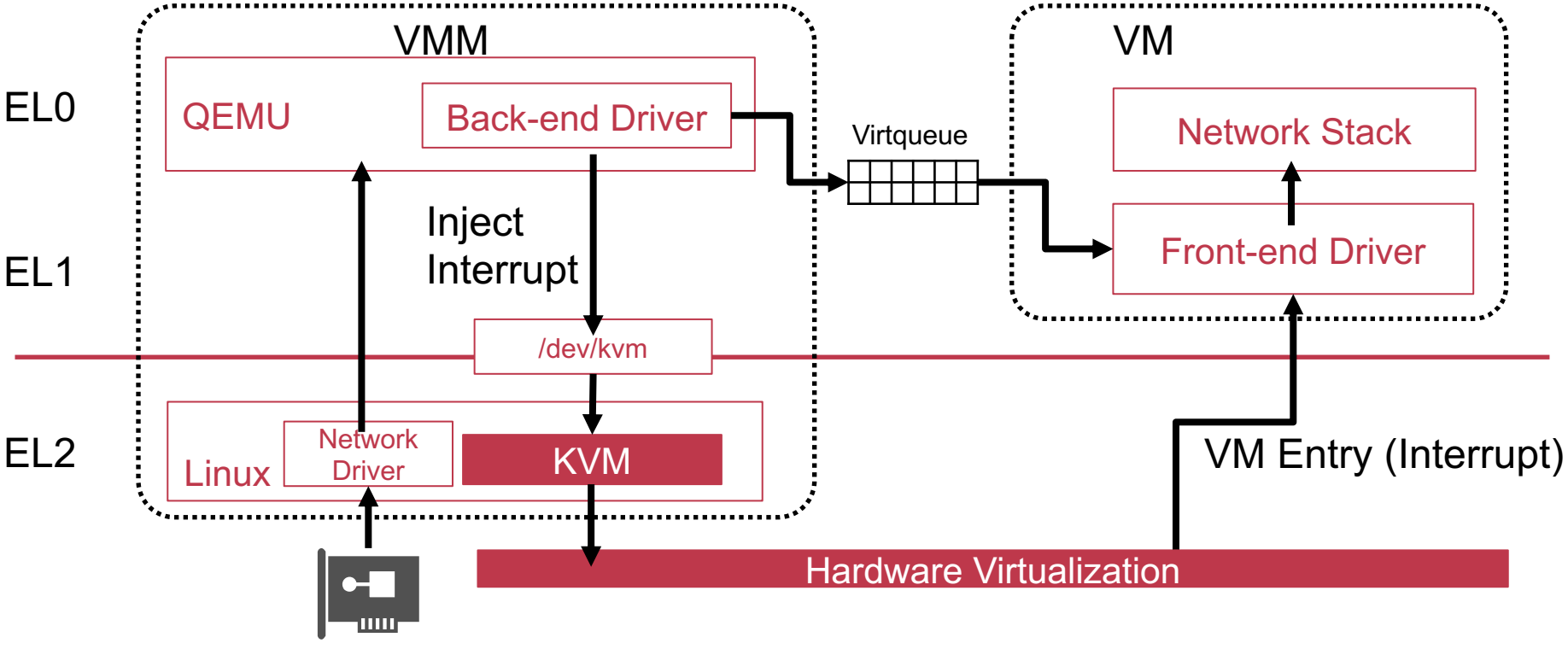
Virtqueue

- **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：发网络包



例：QEMU/KVM半虚拟化：收网络包

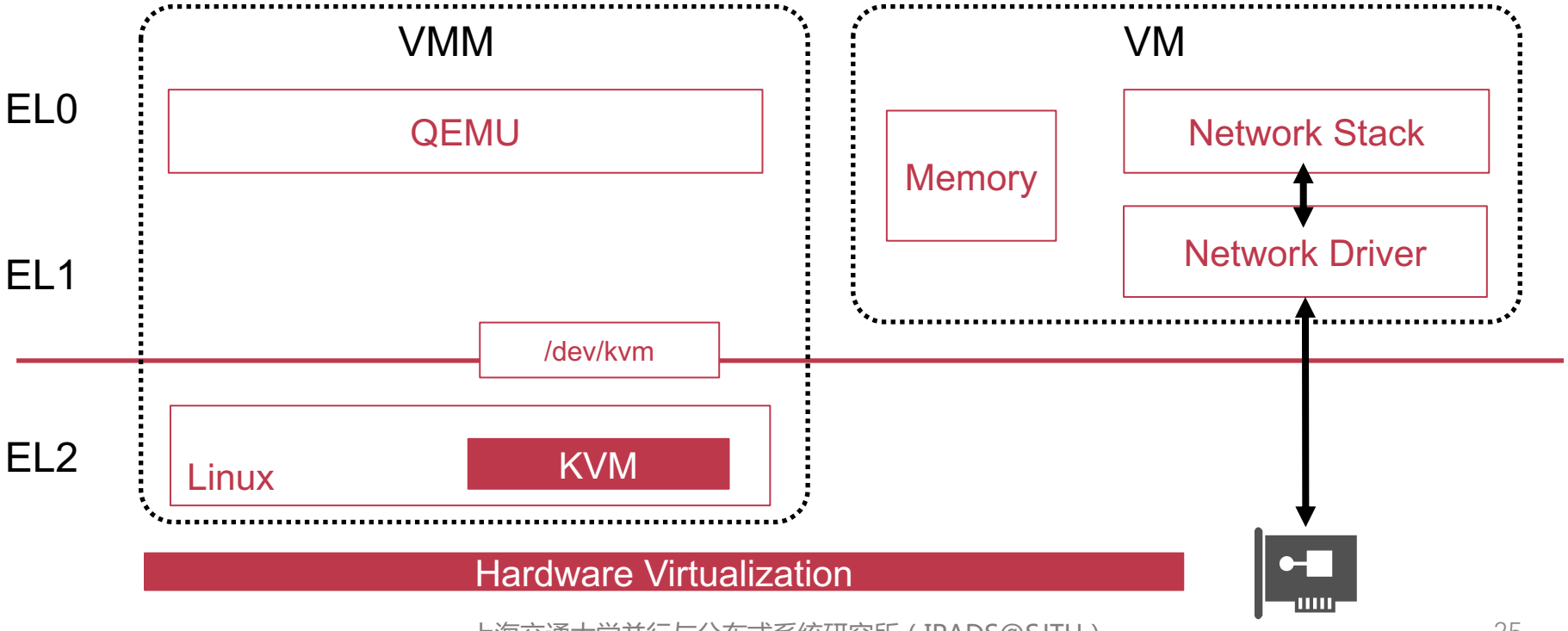


半虚拟化方式的优缺点

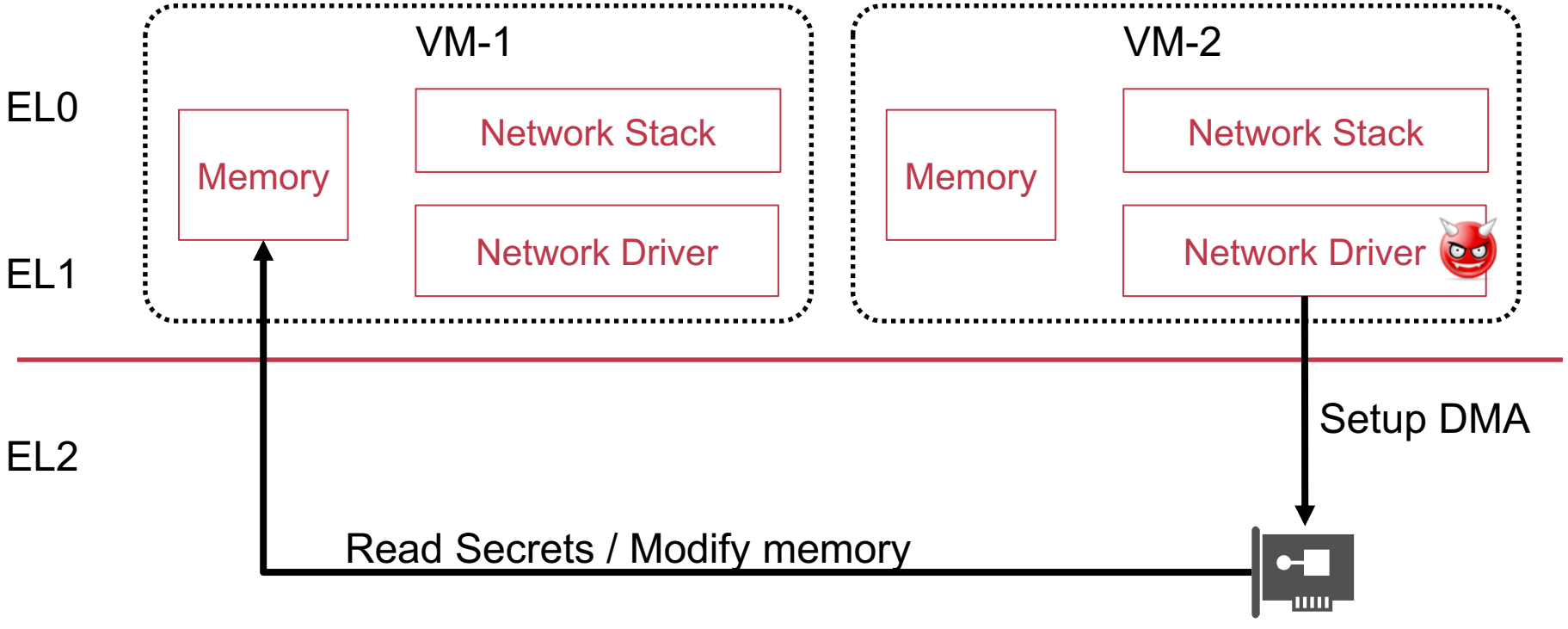
- **优点**
  - 性能优越
    - 多个MMIO/PIO指令可以整合成一次Hypercall
  - VMM实现简单，不再需要理解物理设备接口
- **缺点**
  - 需要修改虚拟机操作系统内核

方法3：设备直通

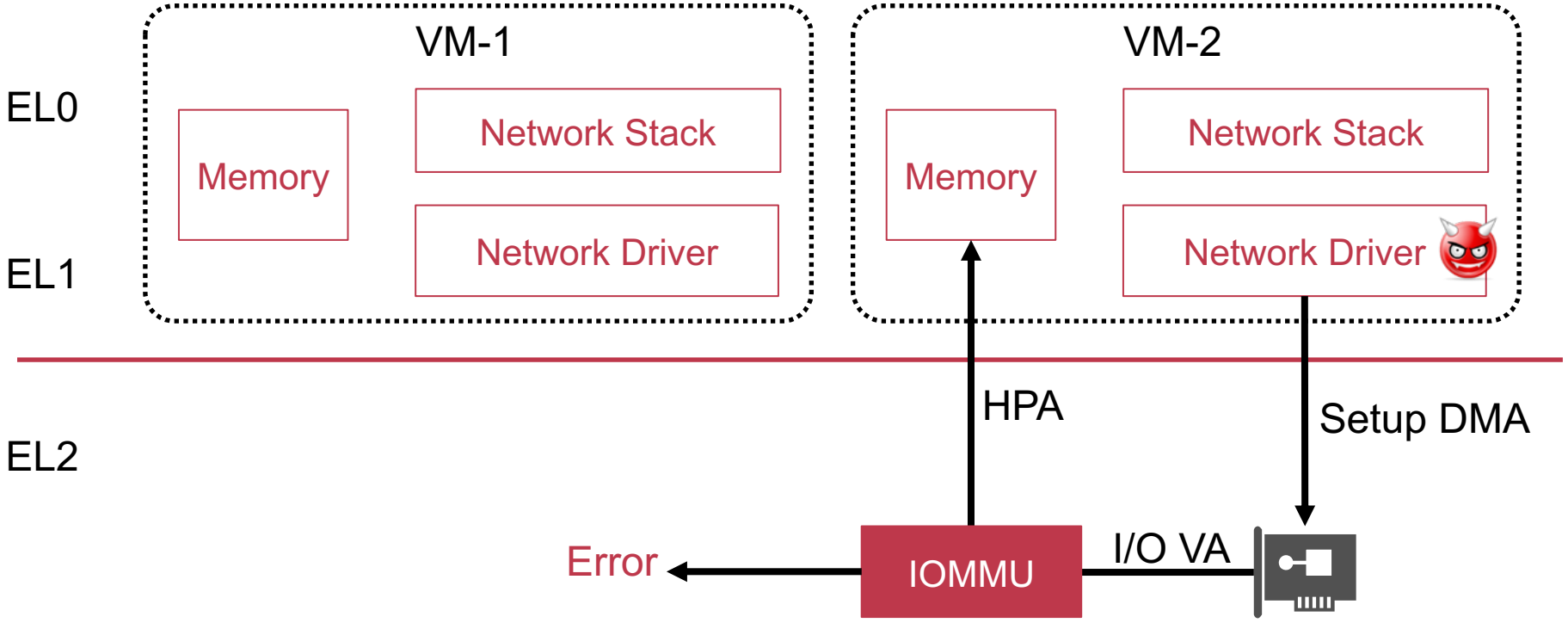
- 虚拟机直接管理物理设备



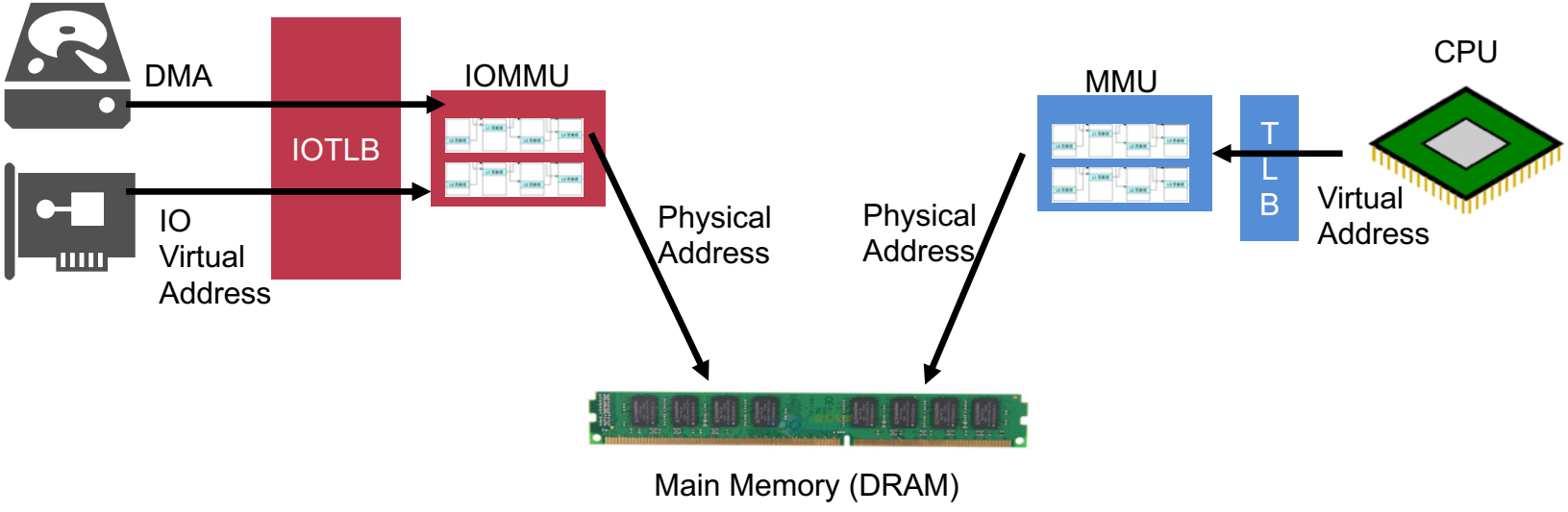
问题1：DMA恶意读写内存



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

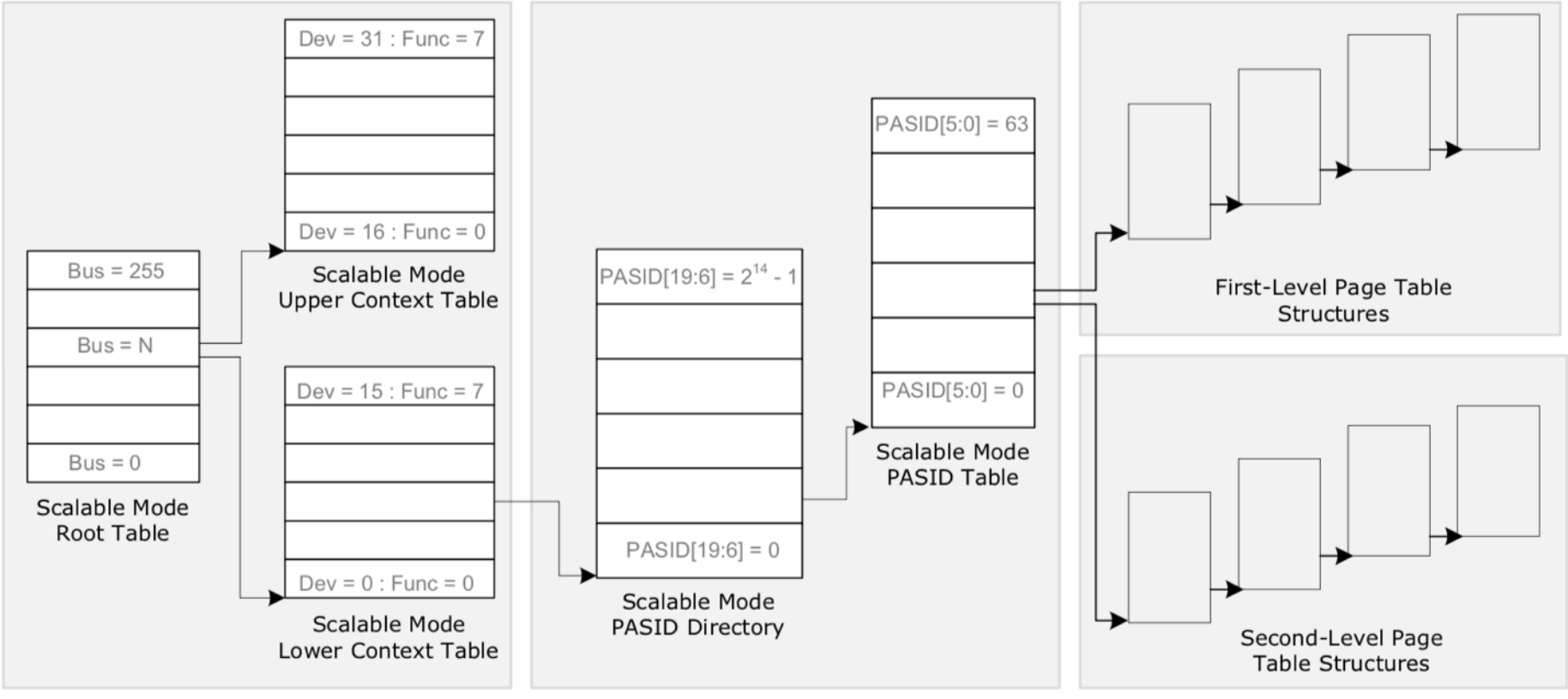
IOMMU与MMU


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

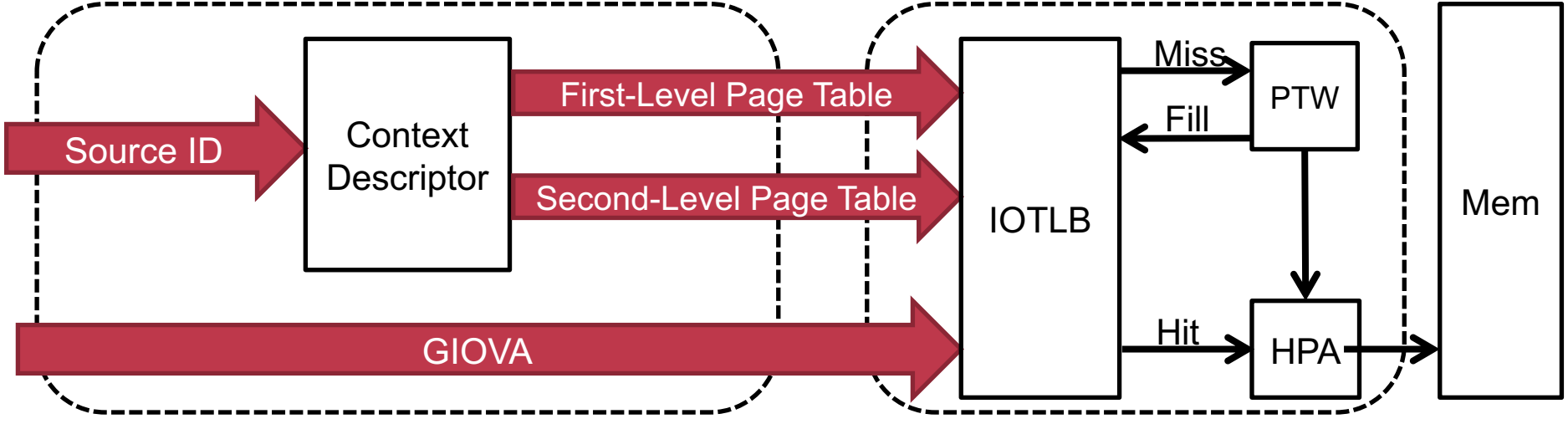
ARM SMMU

- **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



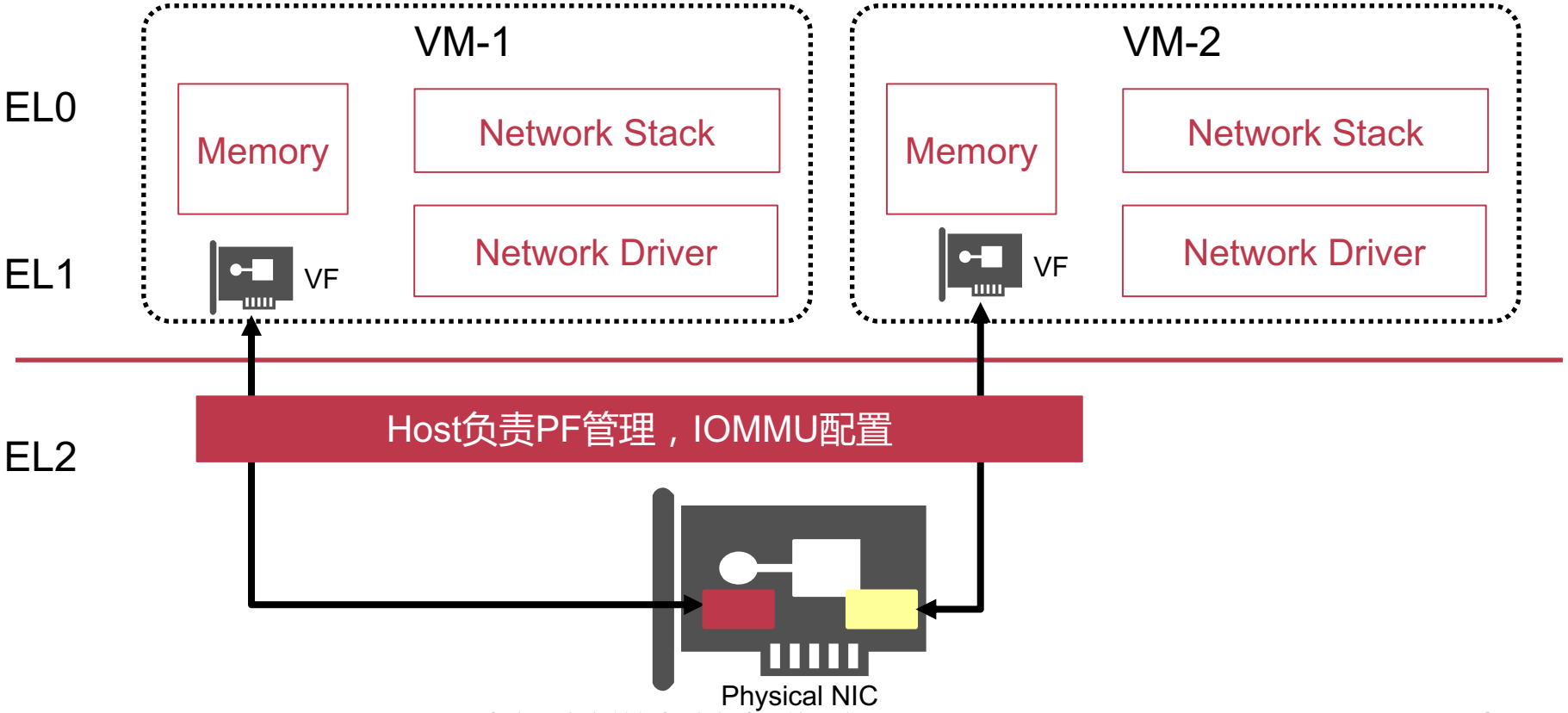
问题2：设备独占

- **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 (SR-IOV)

- **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

SR-IOV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

轻量级虚拟化

虚拟化太重！是否有更轻量级的隔离？

- **Windows Server允许多个用户同时远程桌面**
  - 多个用户可以共享一个操作系统，同时进行不同的工作
- **缺点：多个用户之间缺少隔离**
  - 例如：所有用户共同操作一个文件系统
- **如何想让每个用户看到的文件系统视图不同？**
  - 对每个用户可访问的文件系统做隔离

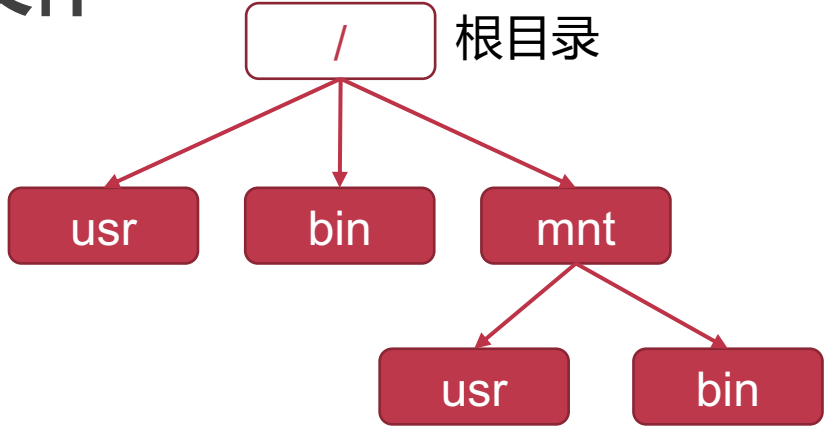
第一次尝试：CHROOT

文件系统视图的隔离

- **为每个执行环境提供单独的文件系统视图**
- **原理**
  - Unix系统中的“一切皆文件”设计理念
  - 对于用户态来说，文件系统相当重要
- **方法**
  - 改变文件系统的根目录，即chroot

Chroot效果

- 控制进程能够访问哪些目录子树
- 改变进程所属的根目录
- 进程只能看到根目录下属的文件



Chroot原理

- **进程只能从根目录向下开始查找文件**
  - 操作系统内部修改了根目录的位置
- **一个简单的设计**
  - 内核为每个用户记录一个根目录路径
  - 进程打开文件时内核从该用户的根目录开始查找
- **上述设计有什么问题？**
  - 遇到类似“..”的路径会发生什么？
  - 一个用户想要使不同进程有不同的根目录怎么办？

Chroot在Linux中的实现

- **特殊检查根目录下的“..”**
  - 使得“/..”与“/”等价
  - 无法通过“..”打破隔离
- **每个TCB都指向一个root目录**
  - 一个用户可以对多个进程chroot

```
struct fs_struct{
    ......
    struct path root, pwd;
};

struct task_struct{
    ......
    struct fs_struct  *fs;
    ......
};
```

正确使用Chroot

- **需要root权限才能变更根目录**
  - 也意味着chroot无法限制root用户
- **确保chroot有效**
  - 使用setuid消除目标进程的root权限

```
chdir("jail");
chroot(".");
setuid(UID); // UID > 0
```

基于文件系统Name Space隔离的限制

- **通过文件系统的name space来限制用户**
  - 如果用户直接通过inode访问，则可绕过
  - 因此不允许用户直接用inode访问文件
- **其它层也可以限制用户**
  - 例如：inode层可以限制用户

Chroot能否实现彻底的隔离？

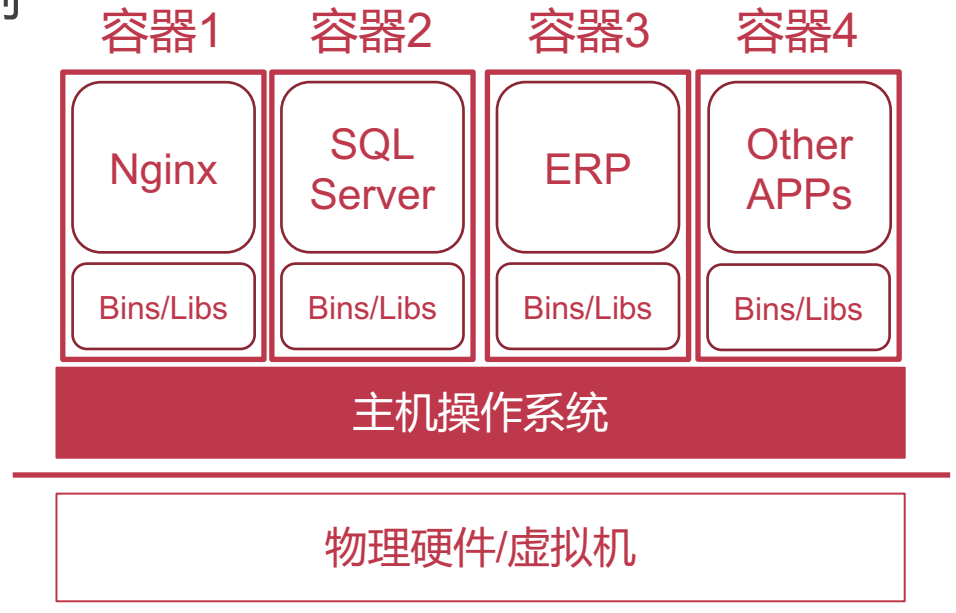
- **不同的执行环境想要共享一些文件怎么办？**
- **涉及到网络服务时会发生什么？**
  - 所有执行环境共用一个IP地址，所以无法区分许多服务
- **执行环境需要root权限该怎么办？**
  - 全局只有一个root用户，所以不同执行环境间可能相互影响

- **不能，因为还有许多资源被共享…**

LINUX CONTAINER

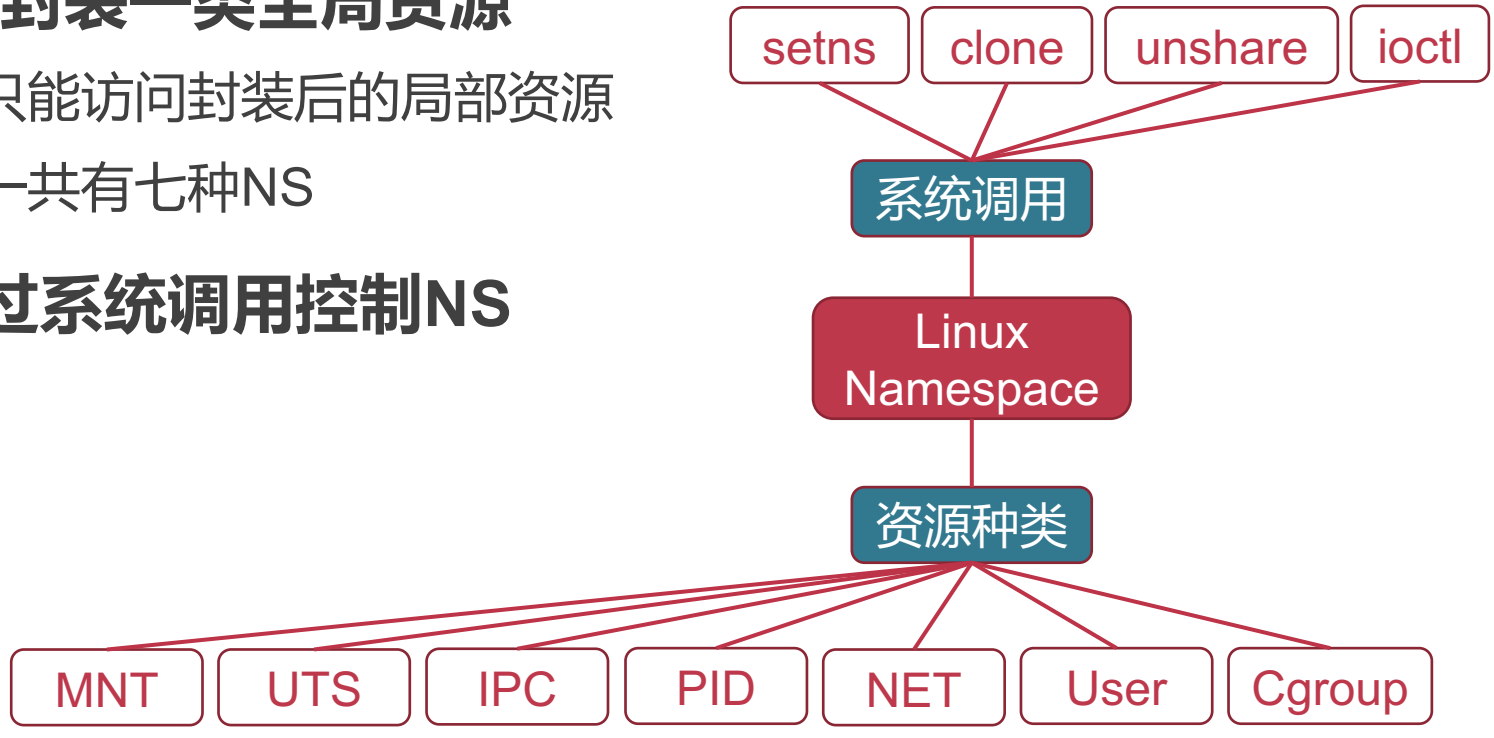
LinuX Container (LXC)

- **容器的概念**
  - 由Linux内核提供资源隔离机制
- **安全隔离**
  - Linux namespace
- **性能隔离**
  - Linux cgroup



Linux Namespace

- **每种NS封装一类全局资源**
  - 进程只能访问封装后的局部资源
  - 目前一共有七种NS
- **进程通过系统调用控制NS**

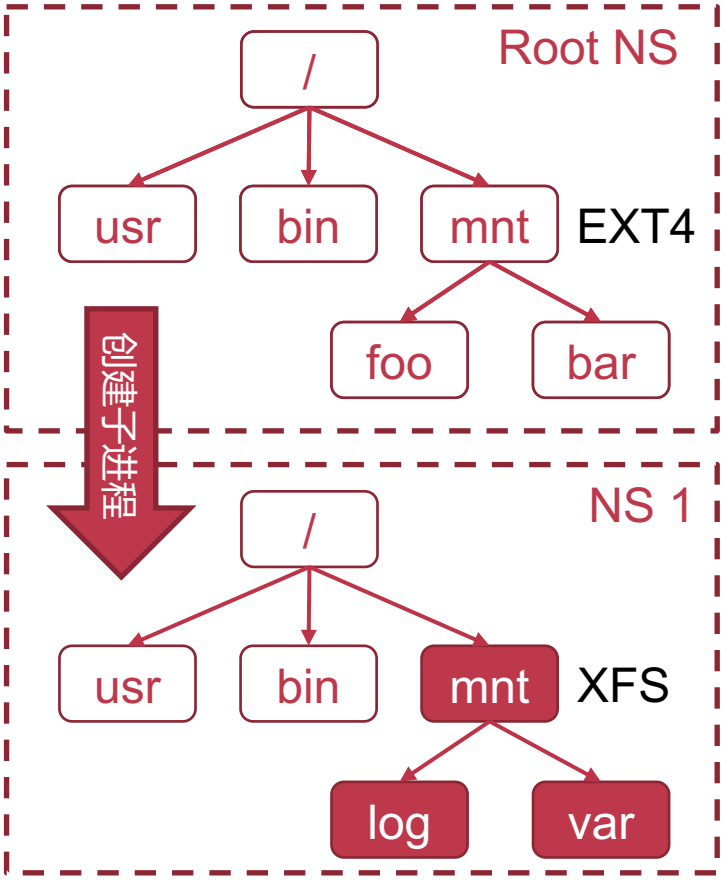


1、Mount Namespace

- **容器内外想要部分共享文件系统**
  - 如果容器内修改了一个挂载点会发生什么？
- **假设主机操作系统上运行了一个容器**
  - 主机操作系统准备从/mnt目录下的ext4文件系统中读取数据
  - 容器中进程在/mnt目录下挂载了一个xfs文件系统
  - 主机操作系统可能读到错误数据

Mount Namespace的实现

- **设计思路**
  - 在内核中分别记录每个NS中对于挂载点的修改
  - 访问挂载点时，内核根据当前NS的记录查找文件
- **每个NS有独立的文件系统树**
  - 新NS会拷贝一份父NS的文件系统树
  - 修改挂载点只会反映到自己NS的文件系统树



2、IPC Namespace

- **不同容器内的进程若共享IPC对象会发生什么？**

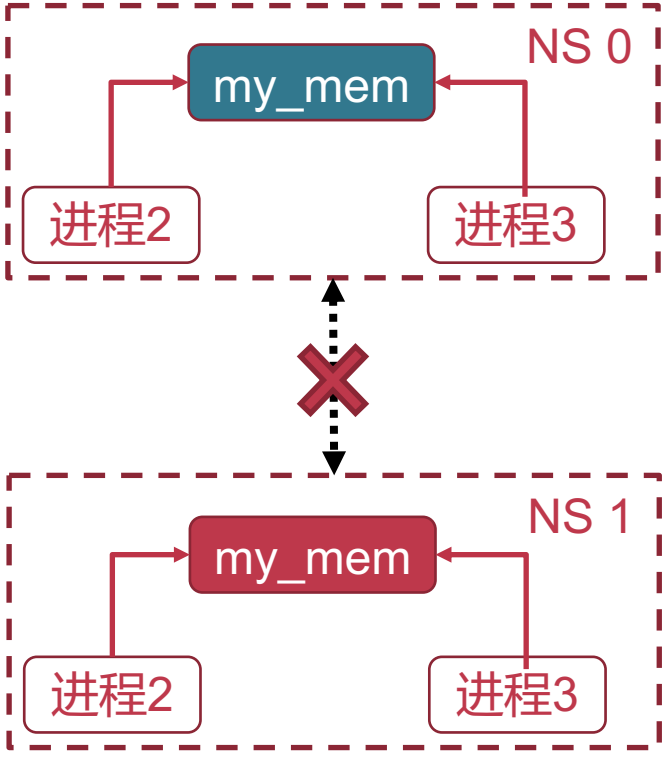
- **假设有两个容器A和B**
  - A中进程使用名为“my_mem”共享内存进行数据共享
  - B中进程也使用名为“my_mem”共享内存进行通信
  - B中进程可能收到A中进程的数据，导致出错以及数据泄露

IPC Namespace的设计

- **直接的想法**
  - 在内核中创建IPC对象时，贴上对应NS的标签
  - 进程访问IPC对象时内核来判断是否允许访问该对象
- **可能的问题**
  - 可能有timing side channel隐患
  - 对于同名的IPC对象不好处理
- **更进一步**
  - 将每个NS创建的IPC对象放在一起管理

IPC Namespace的实现

- **使每个IPC对象只能属于一个NS**
  - 每个NS单独记录属于自己的IPC对象
  - 进程只能在当前NS中寻找IPC对象
- **图例**
  - ID均为my_mem→不同的共享内存



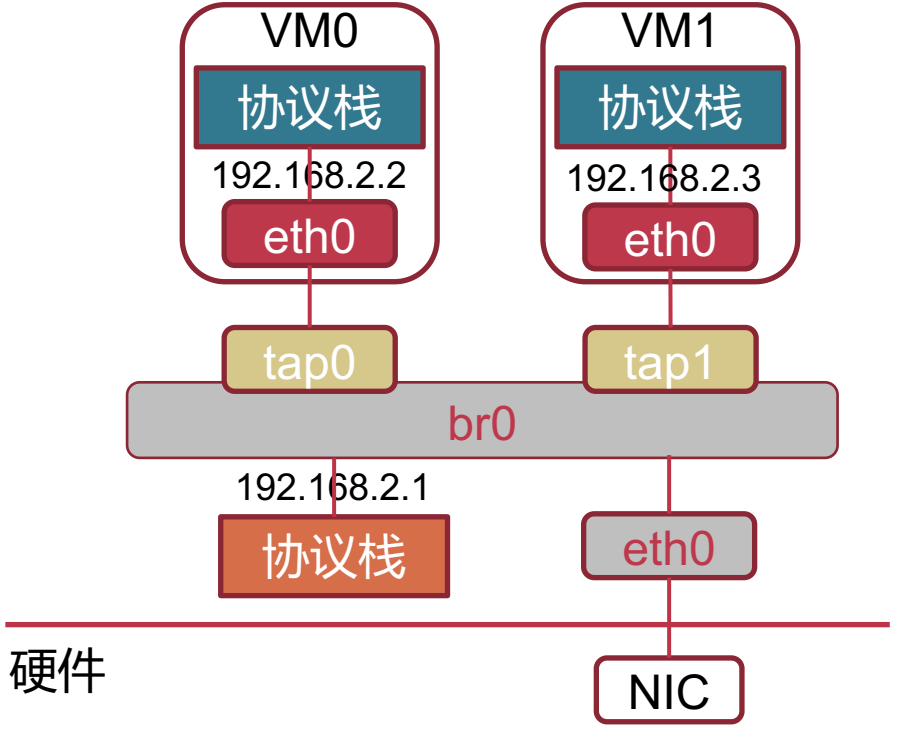
3、Network Namespace

- **不同的容器共用一个IP会发生什么？**

- **假设有两个提供网络服务容器**
  - 两个容器的外部用户向同一IP发送网络服务请求
  - 主机操作系统不知道该将网络包转发给哪个容器

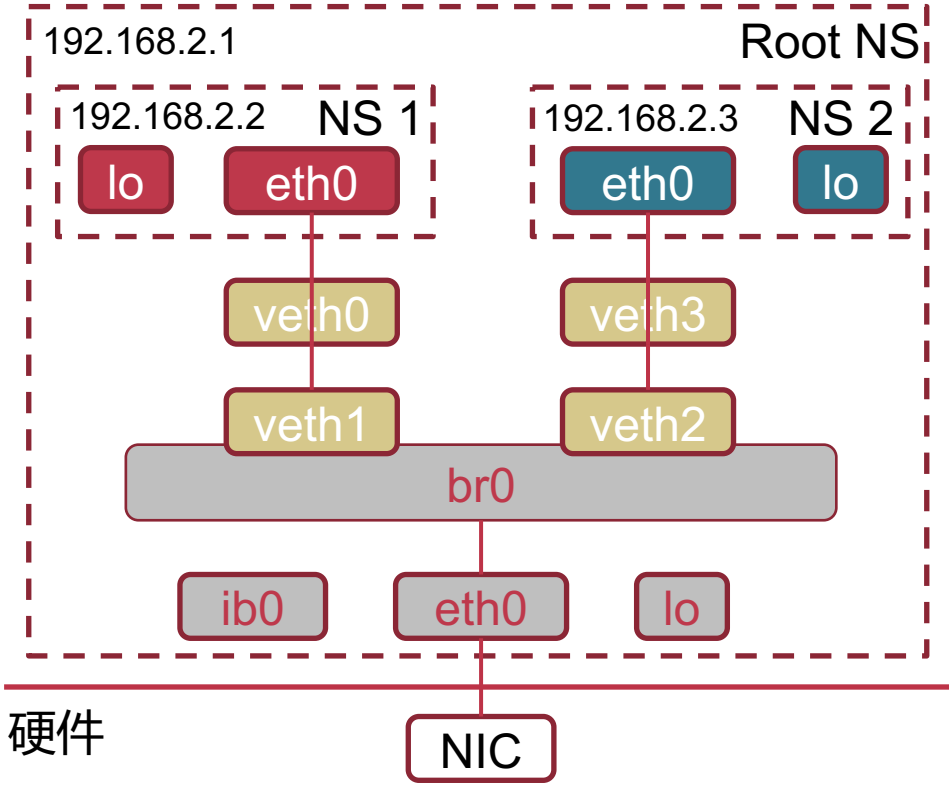
Linux对于多IP的支持

- **在虚拟机场景下很常见**
  - 每个虚拟机分配一个IP
    - IP绑定到各自的网络设备上
  - 内部的二级虚拟网络设备
    - br0: 虚拟网桥
    - tap: 虚拟网络设备
- **如何应用到容器场景？**



Network Namespace的实现

- **每个NS拥有一套独立的网络资源**
  - 包括IP地址、网络设备等
- **新NS默认只有一个loopback设备**
  - 其余设备需后续分配或从外部加入
- **图例**
  - 创建相连的veth虚拟设备对
  - 一端加入NS即可连通网络
  - 分配IP后可分别与外界通信



4、PID Namespace

- **容器内进程可以看到容器外进程的PID会发生什么？**

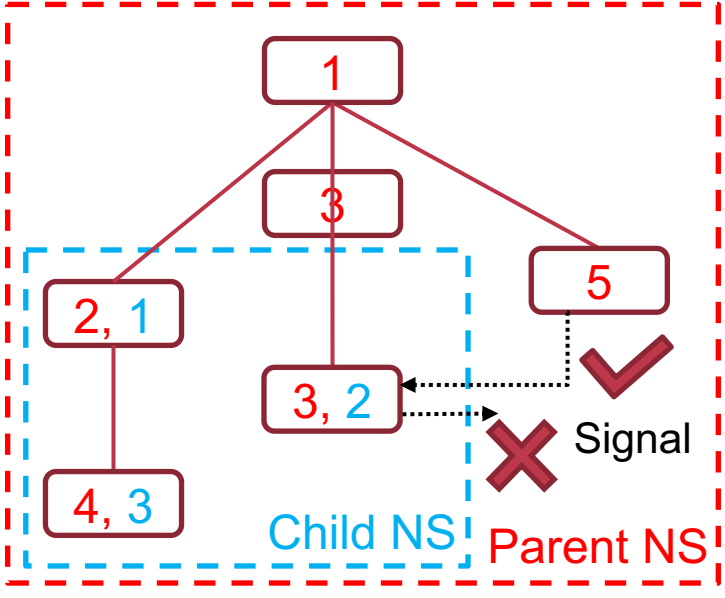
- **假设有容器内存在一个恶意进程**
  - 恶意进程向容器外进程发送SIGKILL信号
  - 主机操作系统或其他容器中的正常进程会被杀死

PID Namespace的设计

- **直接的想法**
  - 将每个NS中的进程放在一起管理，不同NS中的进程相互隔离
- **存在的问题**
  - 父子进程等进程间关系如何处理？
- **更进一步**
  - 允许父NS看到子NS中的进程，保留父子关系

PID Namespace的实现

- **对NS内外的PID进行单向隔离**
  - 外部能看到内部的进程，反之则不能
- **图例**
  - 子NS中的进程在父NS中也有PID
  - 进程只能看到当前NS的PID
  - 子NS中的进程无法向外发送信号



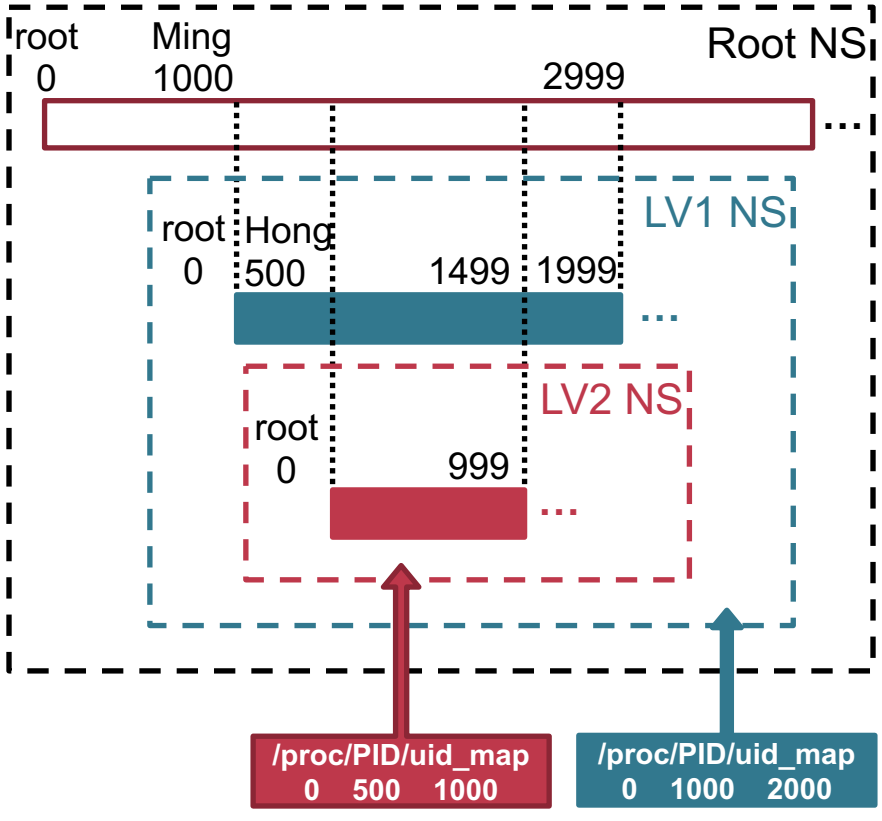
5、User Namespace

- **容器内外共享一个root用户会发生什么？**

- **假设一个恶意用户在容器内获取了root权限**
  - 恶意用户相当于拥有了整个系统的最高权限
  - 可以窃取其他容器甚至主机操作系统的隐私信息
  - 可以控制或破坏系统内的各种服务

User Namespace的实现

- **对NS内外的UID和GID进行映射**
  - 允许普通用户在容器内有更高权限
    - 基于Linux Capability机制
  - 容器内root用户在容器外无特权
    - 只是普通用户
- **图例**
  - 普通用户在子NS中是root用户



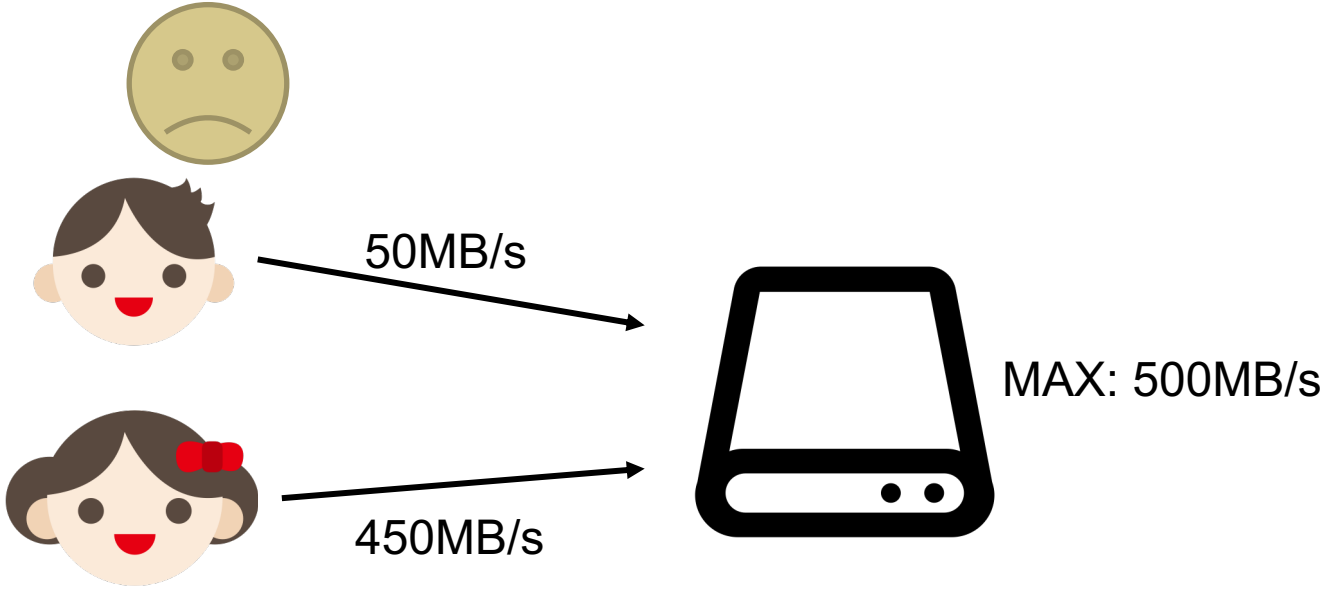
其他Namespace

- **6、UTS Namespace**
  - 每个NS拥有独立的hostname等名称
  - 便于分辨主机操作系统及其上的多个容器
- **7、Cgroup Namespace**
  - cgroupfs的实现向容器内暴露cgroup根目录
  - 增强隔离性：避免向容器内泄露主机操作系统信息
  - 增强可移植性：取消cgroup路径名依赖

性能隔离：CGROUPS

资源竞争问题

- 小明和小红同时访问磁盘



Control Cgroups (Cgroups)

- **Cgroups是什么**
  - Linux内核（从Linux2.6.24开始）提供的一种资源隔离的功能
- **Cgroups可以做什么**
  - 将线程分组
  - 对每组线程使用的多种物理资源进行限制和监控
- **怎么用Cgroups**
  - 名为cgroupfs的伪文件系统提供了用户接口

Cgroups的常用术语

- 任务（task）
- 控制组（cgroup）
- 子系统（subsystem）
- 层级 （hierarchy）

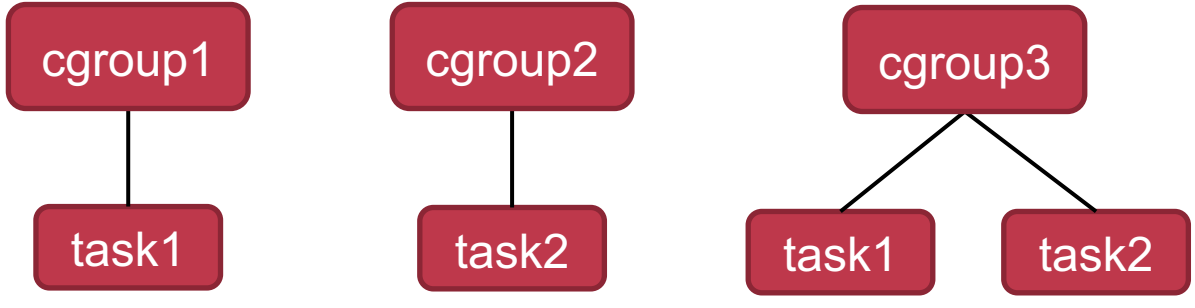
任务（Task）

- **系统中的一个线程**
  - 两个任务 task1和task2

task1　　task2

控制组（Control Group）

- **Cgroups进行资源监控和限制的单位**
- **任务的集合**
  - 控制组cgroup1包含task1
  - 控制组cgroup2包含task2
  - 控制组cgroup3由task1和task2组成



子系统（Sub-system）

- 可以跟踪或限制控制组使用该类型物理资源的内核组件
- 也被称为资源控制器

cpu　　cpuacct　　memory

层级（Hierarchy）

- 由控制组组成的树状结构
- 通过被挂载到文件系统中形成

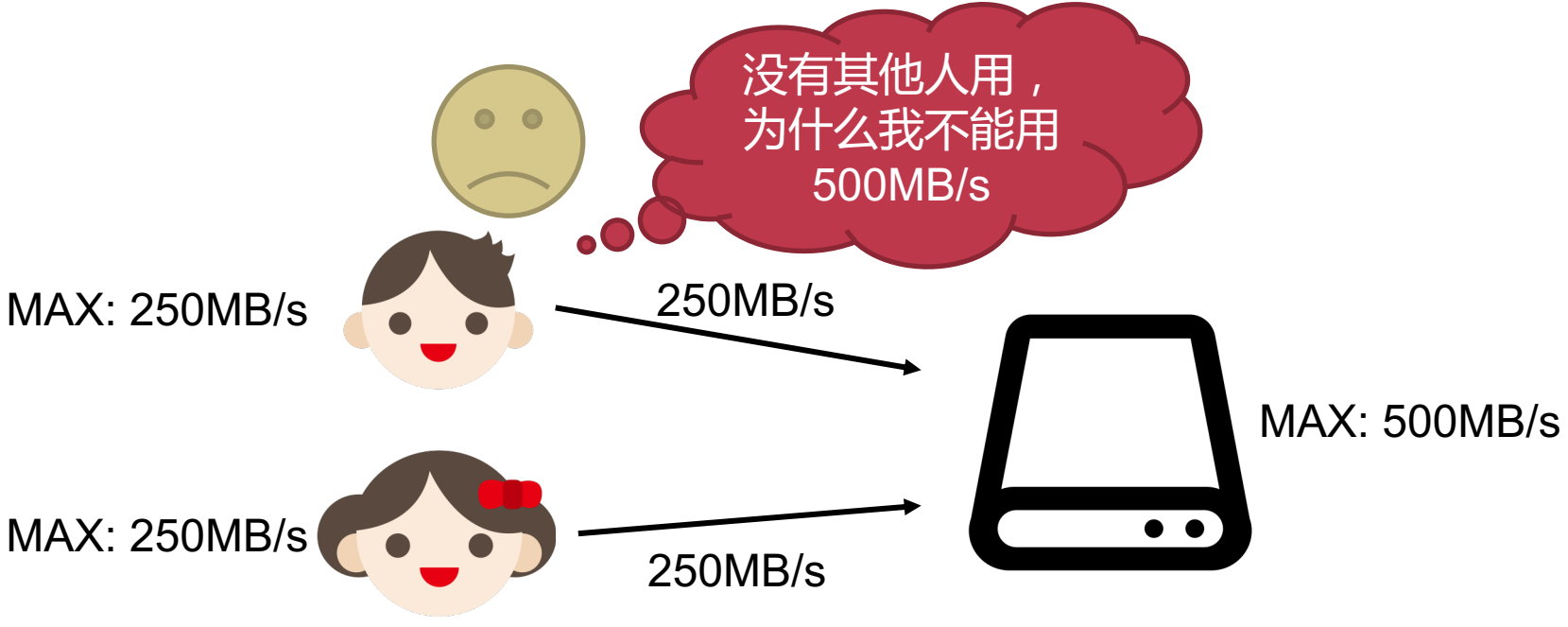
```
$ mount | grep "type cgroup "
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,name=systemd)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,rdma)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
```

资源控制模型

- **最大值**
  - 直接设置一个控制组所能使用的物理资源的最大值，例如：
    - 内存子系统：最多能使用1GB内存
    - 存储子系统：最大能使用100MB/s的磁盘IO

资源控制模型

- 小明和小红同时访问磁盘

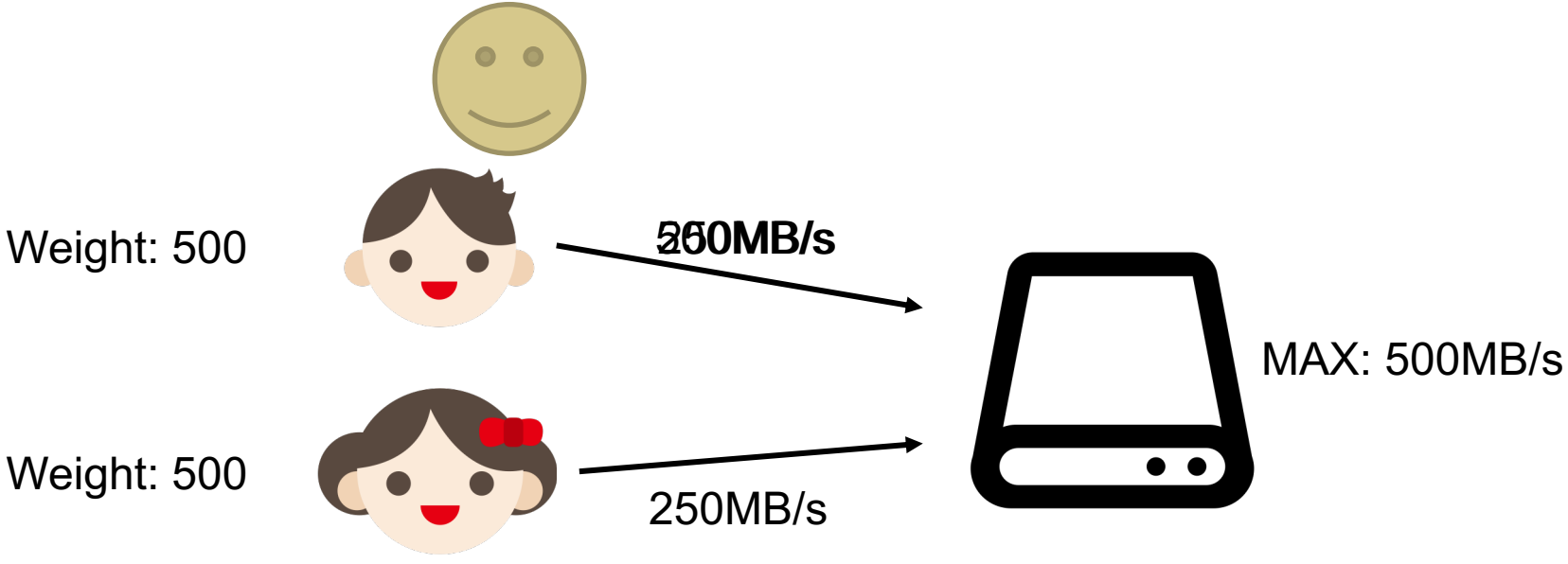


资源控制模型

- **最大值**
  - 直接设置一个控制组所能使用的物理资源的最大值，例如：
    - 内存子系统：最多能使用1GB内存
    - 存储子系统：最大能使用100MB/s的磁盘IO
- **比例**
  - 设置不同控制组使用同一物理资源时的资源分配比例，例如：
    - 存储子系统：两个控制组按照1：1的比例使用磁盘IO资源
    - CPU子系统：两个控制组按照2：1的比例使用CPU时间

资源控制模型

- 小明和小红同时访问磁盘



如何对任务使用资源进行监控和限制

- **Cgroups进行监控和限制的单位是什么？**
  - 控制组
- **如何知道一个控制组使用了多少物理资源？**
  - 计算该控制组所有任务使用的该物理资源的总和
- **如何限制一个控制组**
  - 使该控制组的所有任务使用的物理资源不超过这个限制
  - 在每个任务使用物理资源时，需要保证不违反该控制组的限制

最大IOPS/BPS限制（blkio.throttle）

- **实现于通用块抽象层**
  - 记录当前周期已经发出的IO数量和大小
  - 当新的bio到达时，判断分发了这个bio之后是否超过限制
    - 若未超过限制，则将这个bio分发给I/O调度器
    - 若超过限制，则计算等待时间，到时间再发送这个bio

总结

- 轻量级虚拟化是为了更好的启动性能和运行密度
- 两种思路
  - 更轻量级的虚拟化技术
  - 在内核中增加更多的name space
- 三种隔离技术
  - 虚拟机隔离
  - 容器隔离
  - 虚拟化容器