I/O虚拟化简介

张正君

2021年12月

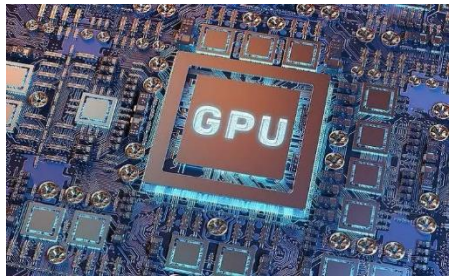
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录 Contents

目录 Contents

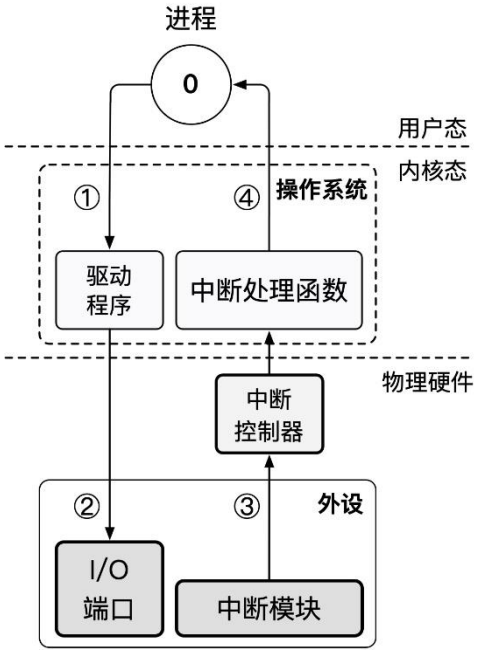
物理机上的I/O

- I/O过程
  - **数据交换** CPU与外部设备进行访问和数据交换的渠道
  - **I/O端口**
    - 控制寄存器
    - 状态寄存器
    - 数据寄存器
- I/O端口编址
  - 与内存统一编址，ARM
  - 独立编址，x86



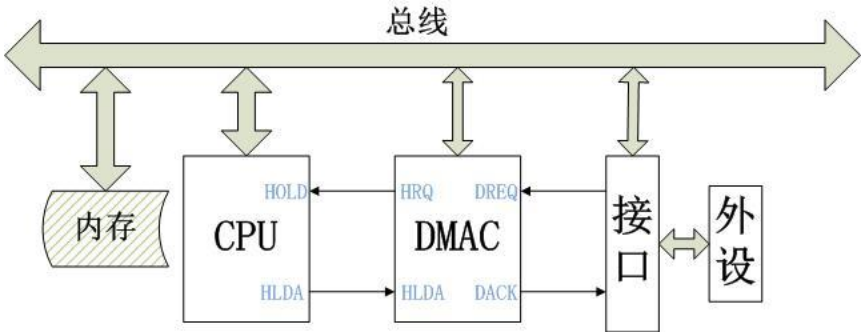
三种I/O访问方式

- PIO（x86架构为例）
  - 访问独立的I/O地址空间
  - 65536个8位I/O端口
  - 连续2个8位->16位，连续四个8位->32位
  - 0 ~ 0xffff，64K
  - **专用的端口访问指令**，IN/OUT
- MMIO
  - 被几乎所有架构支持，RISC仅支持MMIO
  - I/O端口与内存统一编址
  - **通用的内存访存指令**，movq %rax, (%rbx)



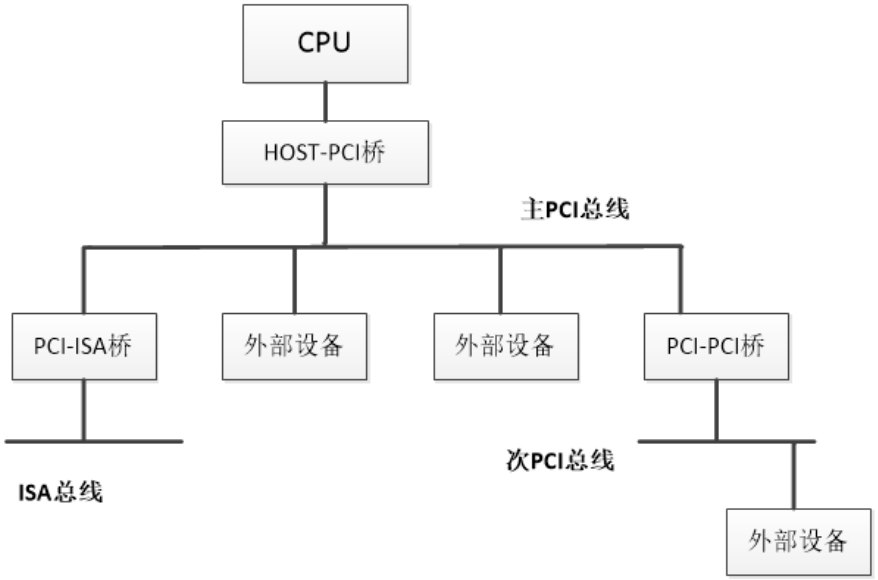
三种I/O访问方式

- DMA
  - 传输过程无须CPU控制
  - 外设<->内存， **直接数据传输**
  - 大批量数据传送
  - DMA控制器，DMAC
    - 接管地址总线
    - 动态修改地址指针



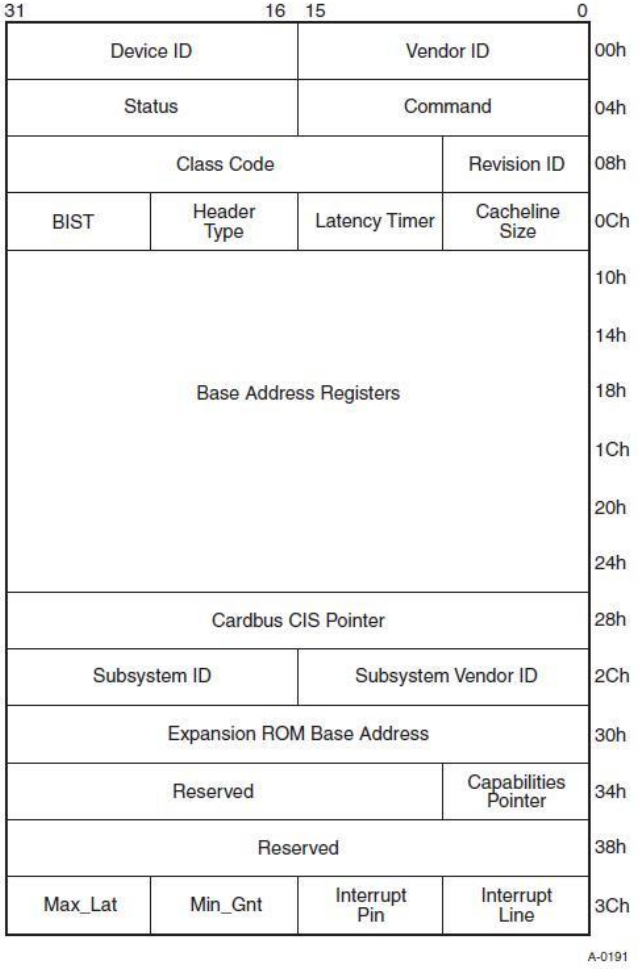
PCI设备简介——PCI总线

- PCI总线——树形结构
  - **根结点**，PCI 主桥
  - **子节点**，PCI-PCI桥、PCI-ISA桥
  - **叶节点**，PCI设备
  - 次级总线，PCI总线0，1，2 ……
- 设备描述符（BDF）
  - 标识PCI设备
  - Bus number，8位，设备所在PCI总线
  - Device number，5位，物理设备编号
  - Function number，3位，逻辑设备编号



PCI设备简介——PCI配置空间

- PCI配置空间
  - 256字节，一组设备寄存器
  - 前64字节，配置头，格式用途固定
  - 厂商预设默认值
- 基地址寄存器（BAR）
  - PIO地址/内存地址 （最后一位区分）
  - BIOS或OS设置
  - 不占用固定地址->热插拔



I/O虚拟化基本任务

- 访问截获
  - 限制VM对物理设备直接访问
  - 隔离性、安全性、资源共享
- 提供设备接口
  - 虚拟设备接口，例如QEMU暴露的虚拟 PCI设备
  - 直通设备接口，VT-d
- 实现设备功能
  - Hypervisor需要实现虚拟设备的功能逻辑

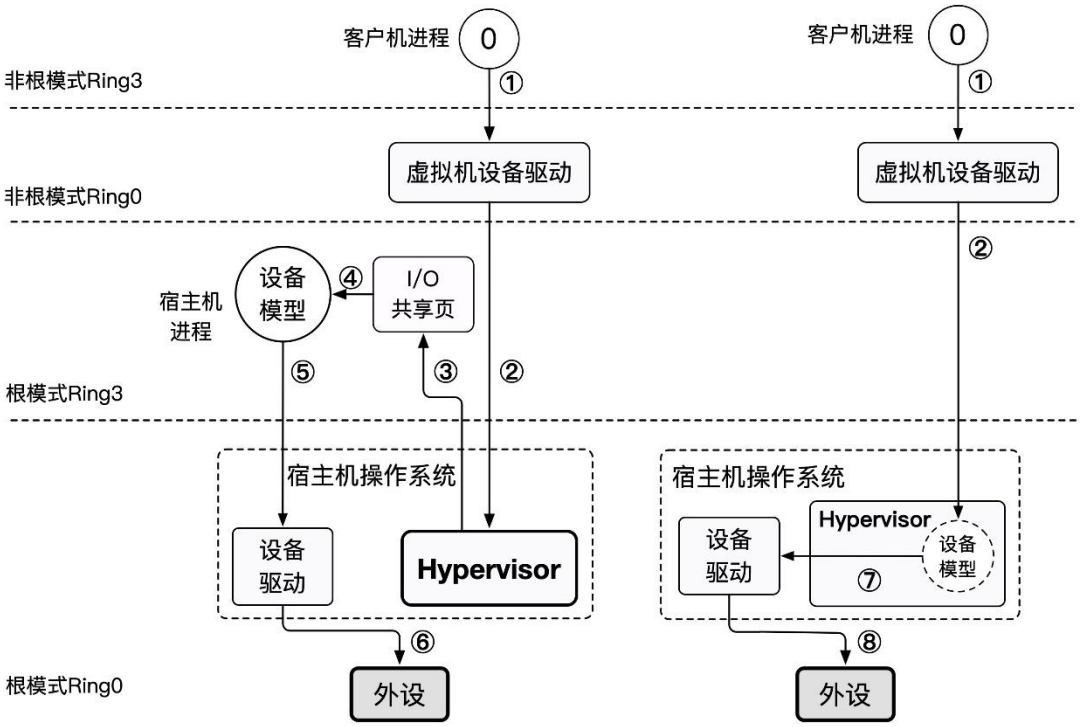
目录 Contents

软件实现的I/O虚拟化——设备模拟

- 虚拟设备抽象
  - 虚拟BIOS和客户机操作系统检测虚拟设备
  - 虚拟设备->虚拟的设备总线
  - I/O指令-> VM-Exit -> hypervisor
  - 对虚拟机完全“透明”
- 设备模型——两部分
  - 虚拟设备接口
  - 具体设备功能的软件实现

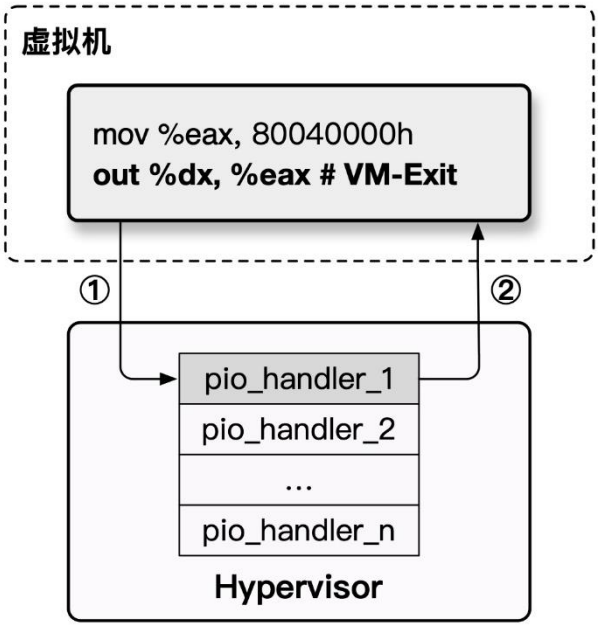
设备模型的两种运行环境

- 用户态设备模型
  - QEMU/KVM
  - 多次上下文切换
- Hypervisor中的设备模型
  - Xvisor
  - 避免多次上下文切换
  - 缩短I/O模拟路径
  - 代价：移植性



设备模拟——PIO模拟过程

- 虚拟机陷入过程
  - IN/OUT、INS/OUTS -> VM-Exit
  - 保留端口号、访问数据宽度、数据传输方向、数据传输方向
- Hypervisor中的处理过程
  - I/O端口对应的处理函数在设备模型初始化时会被注册到hypervisor中
  - 函数指针被组织成数组
  - 根据I/O端口号和访问数据宽度寻找相应PIO处理函数

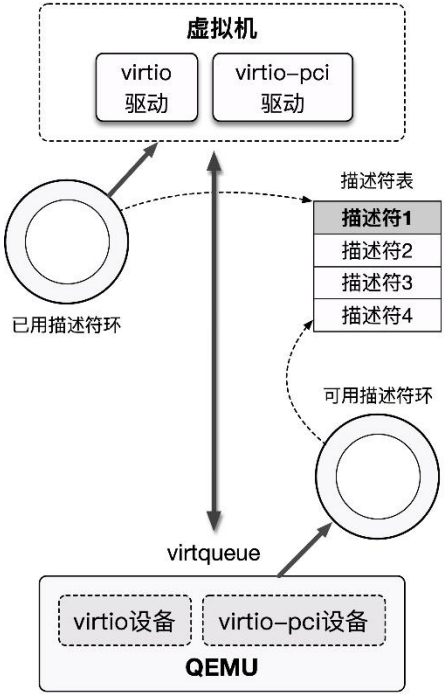


设备模拟——MMIO模拟过程

- 虚拟机陷入过程
  - 访存指令，非敏感指令
  - 影子页表 or EPT页表不存在相应页表项 -> 缺页异常 -> VM-Exit
- Hypervisor中的处理
  - MMIO内存区域较大，通常不采用PIO中的函数数组形式
  - 为MMIO区域注册一个MMIO处理函数
  - 处理函数定位到需要访问的I/O端口

软件实现的I/O虚拟化——Virtio半虚拟化

- virtqueue机制
  - 处理批量的异步I/O请求
  - 减少上下文切换次数
  - 基于共享内存机制
- Virtqueue机制具体实现——vring
  - 描述符表：保存一系列描述符，每一个描述符都被用来描述一块客户机内的内存区域
  - 可用描述符环：保存后端设备可以使用的描述符
  - 已用描述符环：后端驱动已经处理过并且尚未反馈给前端驱动的描述符



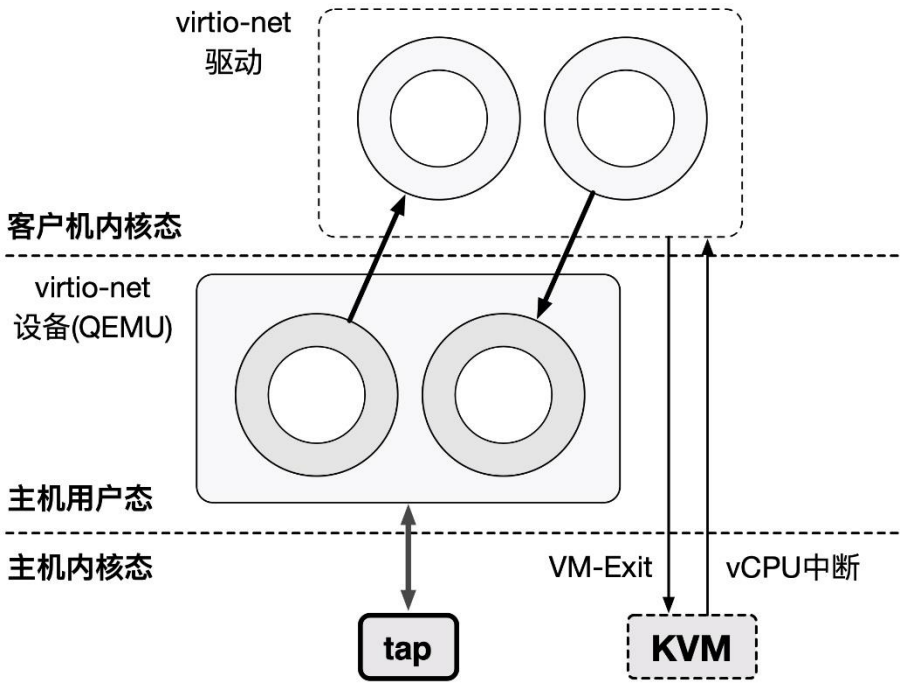
Virtqueue初始化

- Virtqueue的相关参数例如地址和大小都保存在virtio-header中。virtio-header存放在virtio-pci设备配置空间第一个BAR指向的I/O区域。
- 根据后端设备种类不同，一个前端驱动可能拥有多个队列。可以将virtqueue的索引写入virtio-header中的Queue Select寄存器以此来通知设备所要初始化的具体队列。
- 为了给virtqueue分配空间，驱动还需要知道virtqueue大小。前端驱动通过读virtio-header中的Queue Size寄存器，获得virtqueue内描述符的数量。
- 根据描述符数量计算并为virtqueue分配内存空间，并将内存空间的起始地址除以4096，转换成以页为单位的地址后写入virtio-header中的Queue Address 寄存器。后端设备收到该地址后，将改地址左移12位获得virtqueue的GPA，最后将该GPA转换为HVA。

Bits	32	32	32	16	16	16	8	8
Read/Write	R	R+W	R+W	R	R+W	R+W	R+W	R
Purpose	Device Features bits 0:31	Guest Features bits 0:31	Queue Address	Queue Size	Queue Select	Queue Notify	Device Status	ISR Status

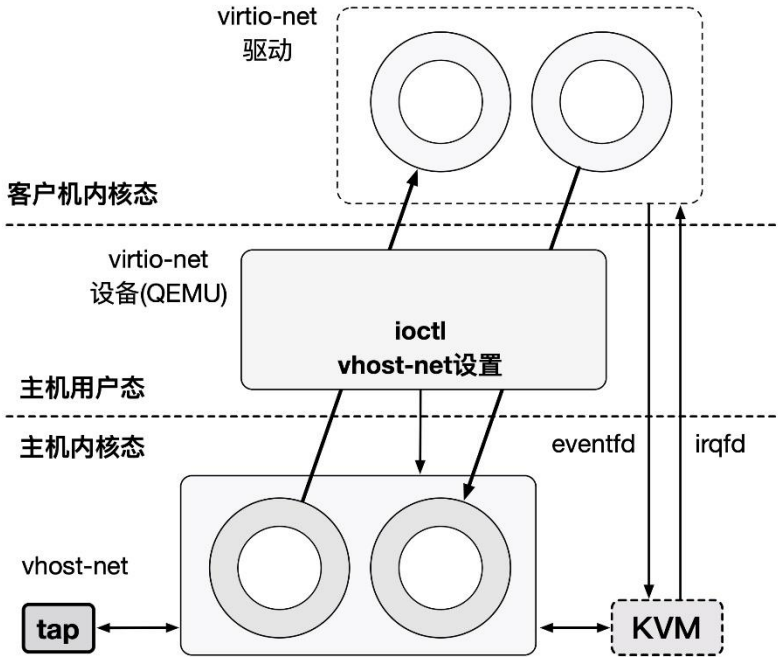
Virtio架构缺陷

- virtio能够支持各种不同的设备，基于virtio实现的网络架构通常被称为virtio-net
- 后端设备位于用户态QEMU进程，VCPU需要暂停执行
- 数据包接收涉及虚拟机内核与主机KVM模块之间、KVM模块与主机用户态QEMU进程之间、QEMU进程与主机内核之间的多次上下文切换



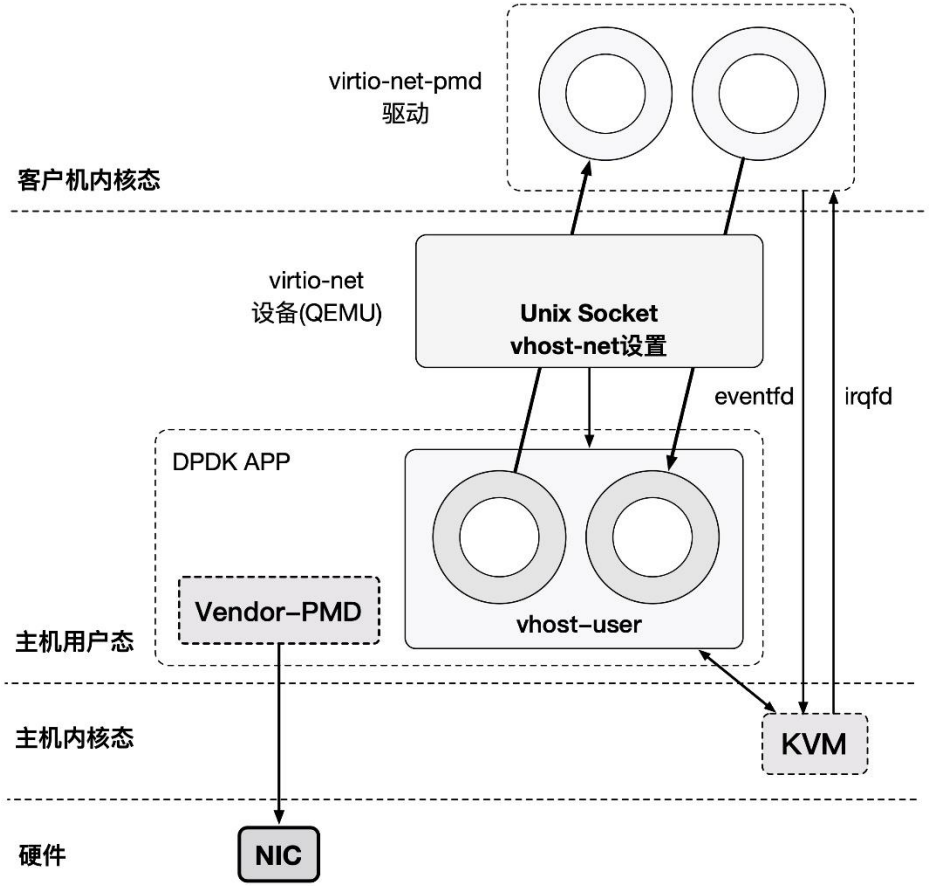
Vhost-net架构

- Vhost API协议
  - 数据交互的工作卸载到内核handler——vhost-net, 控制面和数据面解耦
  - 通过ioctl初始化vhost-net
  - 一对文件描述符ioeventfd/irqfd用于vhost发送和接收事件通知
- Vhost优势
  - 异步处理，不需要终止vCPU的执行，避免了上下文切换开销
  - 并不影响前端驱动的固有设计，vhost对前端驱动来说是透明的



Vhost-user架构

- Vhost-user
  - Vhost从宿主机内核迁移到宿主机用户态，一般集成在DPDK等用户态驱动中
  - 采用 UNIX 域套接字来建立QEMU进程与vhost-user之间的联系，进而初始化vhost-user
  - 事件通知机制与Vhost-net相同
  - 数据在用户态传递



硬件实现的I/O虚拟化——设备直通

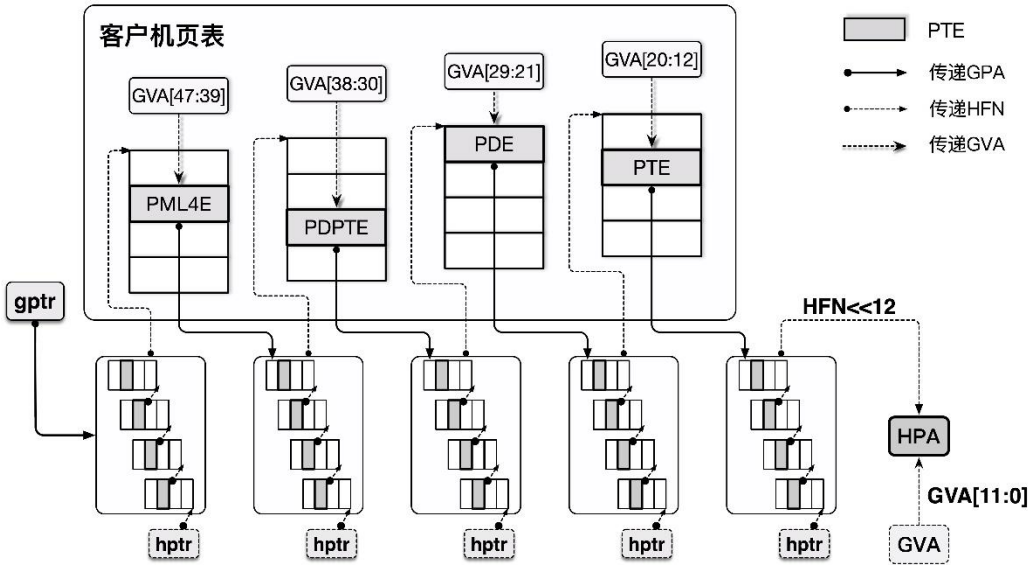
- 设备直通的优势
  - 不发生或发生少量VM-Exit
  - 性能接近裸机
- 设备直通需解决的问题
  - 如何保证虚拟机的原生驱动能够直接通过真实的I/O地址空间操作I/O设备?
  - DMA过程中如何控制外设访问到虚拟机所在的物理内存地址?

Intel VT-d——PIO处理

- I/O位图
  - 决定虚拟机是否可以直接访问某个端口
  - VM-Excution控制域，Use I/O bitmaps字段=1，启用I/O位图
  - VM-Excution控制域，I/O位图地址字段 ->两个大小为4KB的I/O位图
  - 端口在I/O位图中对应的位值为0，此时不会发生VM-Exit，客户机会直接访问该物理端口
- Hypervisor同时支持虚拟设备和直通设备
  - 端口映射表：防止两种设备端口产生冲突
  - 端口在I/O位图中对应的位值为1，发生VM-Exit， Hypervisor会根据映射关系将访问请求发送给直通设备的I/O端口

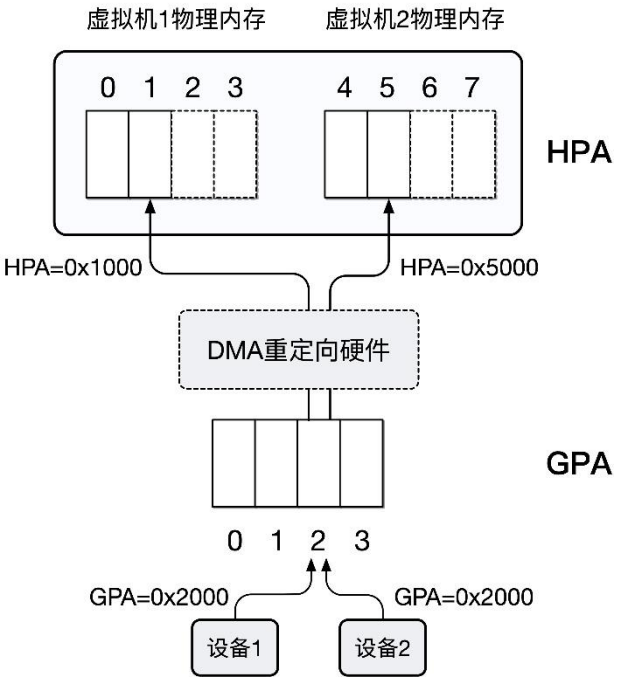
Intel VT-d——MMIO处理

- 利用Intel-VT提供的内存虚拟化支持
  - EPT Misconfiguration和EPT Violation-> VM-Exit
  - Hypervisor在EPT建立虚拟MMIO与物理MMIO映射页表项
  - 再次访问，不产生VM-Exit



Intel VT-d——DMA重映射

- 为什么需要DMA重映射?
  - VM驱动使用的DMA地址为GPA
  - 硬件操作，hypervisor无法帮助VM完成地址转换->引入硬件支持
- DMA重映射硬件
  - 位于PCI总线根部的北桥芯片
  - I/O页表，记录DMA地址映射关系，由hypervisor建立和维护



两种DMA请求

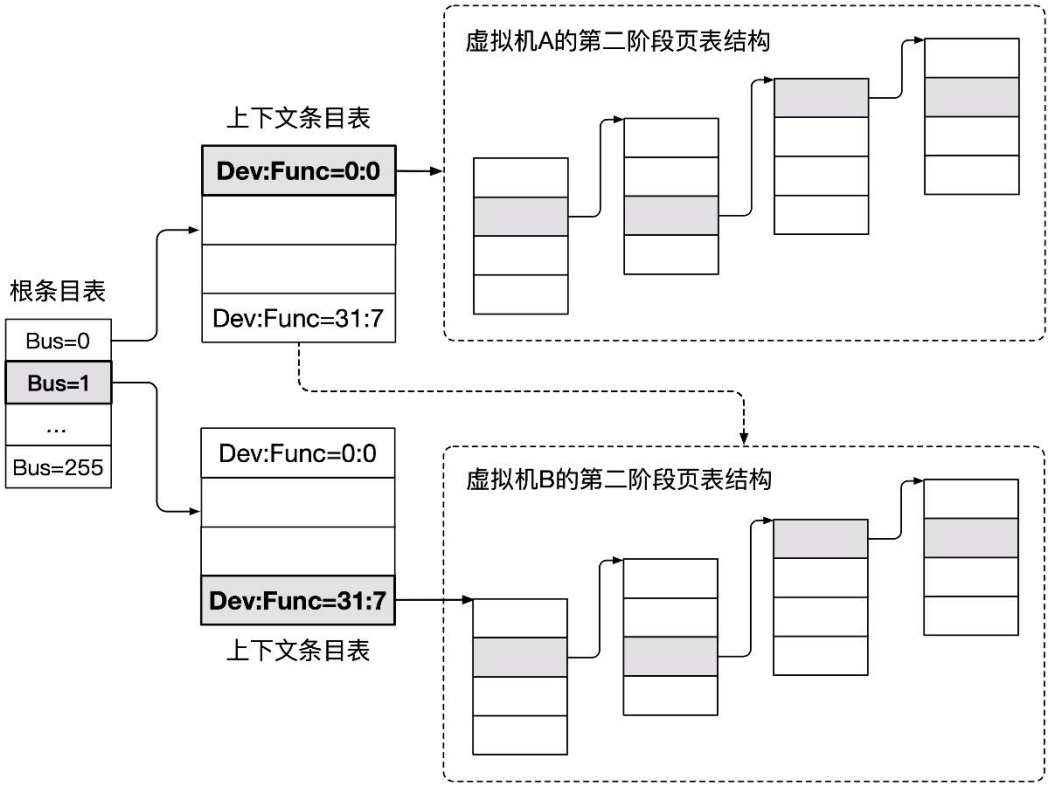
- 不带PASID的请求：
  - DMA地址一般是GPA和IOVA
  - 表明该请求的类型（读、写或原子操作）
  - DMA目标的地址、大小和发起请求的源设备的ID等信息
- 带有PASID的请求：
  - DMA地址一般是GVA和HVA
  - 只有具有Virtual Address Capability的PCI设备才能发出这类请求
  - 表明该请求的类型（读、写或原子操作）
  - DMA目标的地址、大小和发起请求的源设备的ID等信息
  - 定位进程地址空间的PASID

VT-d两种地址翻译模式

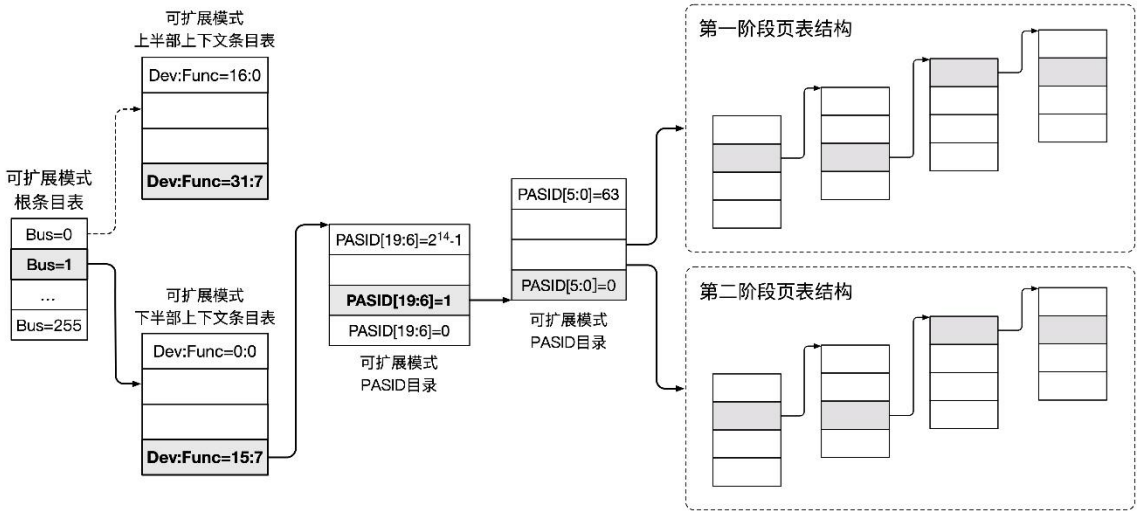
- Legacy模式
  - 仅支持不带PASID的DMA请求
  - 仅支持第二级地址转换，GPA->HPA
- Scalable模式
  - 同时支持不带PASID和带有PASID的DMA请求
  - 将不带PASID的请求转换为带PASID的请求
  - 同时支持两级地址转换，GVA->GPA，GPA->HPA

Legacy模式

- 根条目表
  - 4KB，256个根条目，对应256条PCI总线
  - 根条目中包含一个上下文条目表的指针
- 上下文条目表
  - 使用设备号和功能号进行索引
  - 第二级转换页表的基地址
  - 同一虚拟机的直通设备对应同一个第二阶段页表



Scalable模式



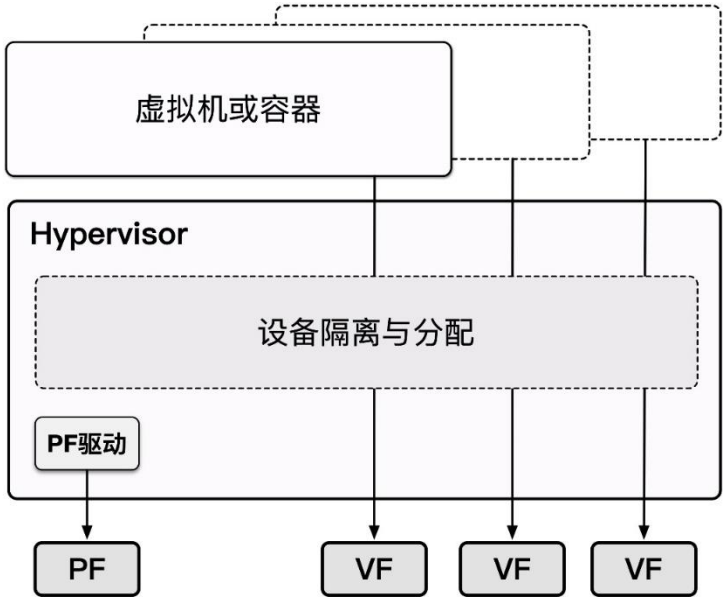
- 可扩展模式根条目表
  - 4KB，256个根条目，对应256条PCI总线
  - 根条目中包含两个上下文条目表的指针（上半部和下半部）
- 上下文条目表
  - 使用设备号和功能号索引，包含PASID
- PASID目录
  - 使用PASID[19:6]进行索引，包含PASID表地址指针
- PASID 表
  - 使用PASID[5:0]进行索引，包含第一级和第二级转换页表的指针
  - PGTT字段=001代表只进行第一阶段地址转换，010代表只进行第二阶段地址转换，011代表需要执行嵌套转换

VFIO

- 用户态驱动（UIO）
  - 绕开开源协议
  - 开发工作量少，易于调试，易于集成到特定应用（DPDK）
  - 缺点：无法动态申请DMA区域
- VFIO
  - 基于IOMMU的一套软件框架，用于开发用户态驱动
  - VFIO模块文件接口 /dev/vfio/$Group
  - 一组IOCTL
    - 配置IOMMU，将DMA地址映射到进程地址空间
    - 注册中断处理函数
    - ……

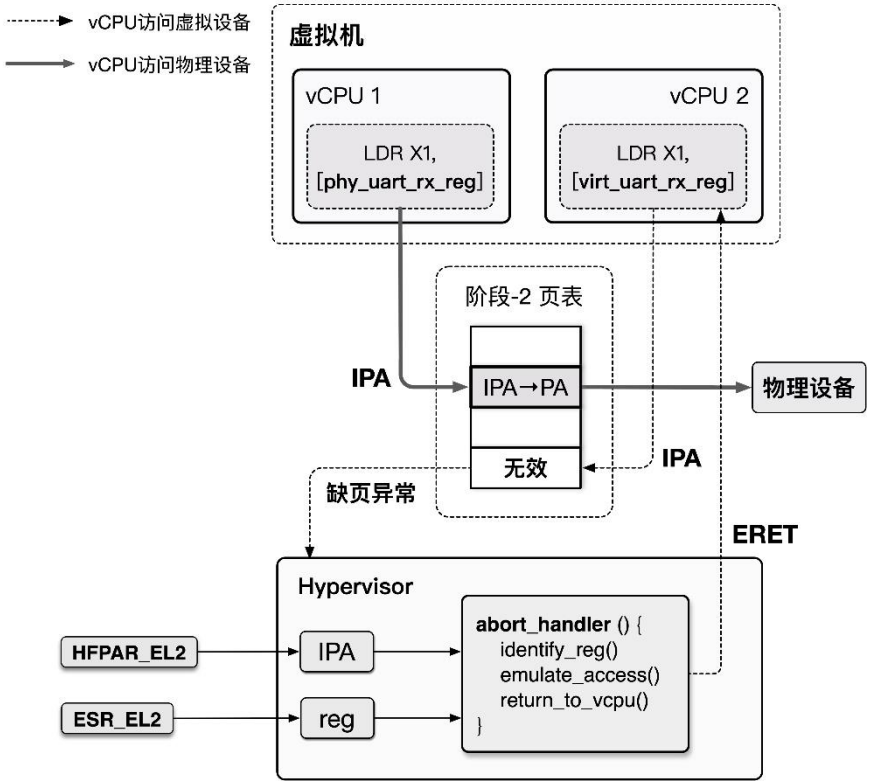
硬件实现的I/O虚拟化—— SR-IOV

- 设备直通的缺陷
  - 设备独占，利用率下降 eg.网卡
- SR-IOV
  - 硬件层面将物理设备虚拟成多个设备
  - PF： 一种支持SR-IOV的PCI功能
    - 将设备所有的物理资源划分为多个资源子集
    - 资源子集之上创建VF
  - VF：从PF中分离出的PCIe功能
    - 拥有专属的配置空间
    - 能够与VM直通

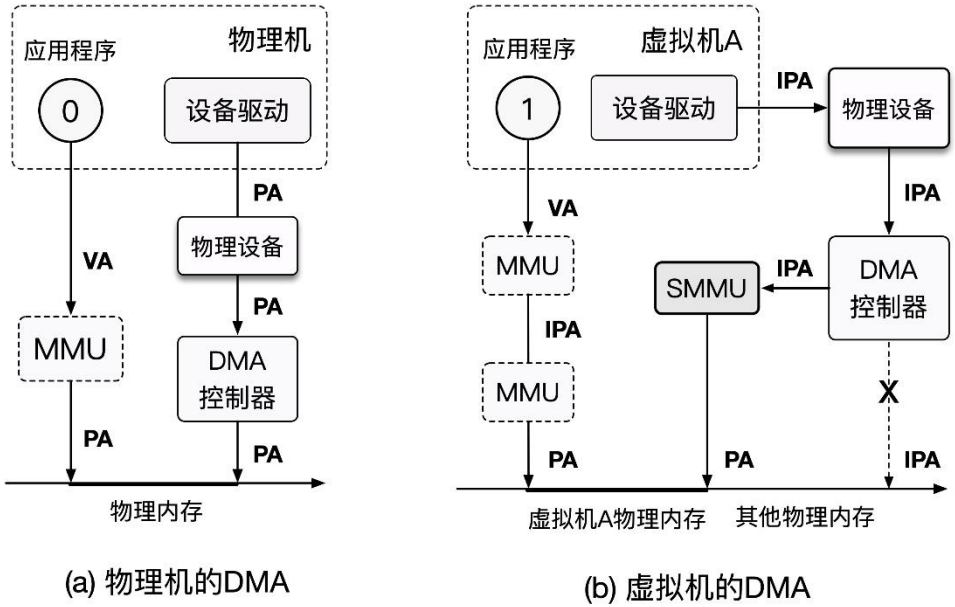


硬件实现的I/O虚拟化—— ARM-V8

- 物理直通设备MMIO
  - 阶段-2页表包含物理空间地址与虚拟机IPA之间的映射
  - 不触发VM-Exit
- 虚拟设备MMIO
  - 阶段-2 缺页异常
    - 将IPA地址填充到HPFAR_EL2寄存器中
    - 访问相关信息[Read,4 bytes, x0]，填充到ESR_EL2寄存器中
  - 调用emulate_access函数，完成MMIO的模拟
  - ERET指令将控制流返回给vCPU

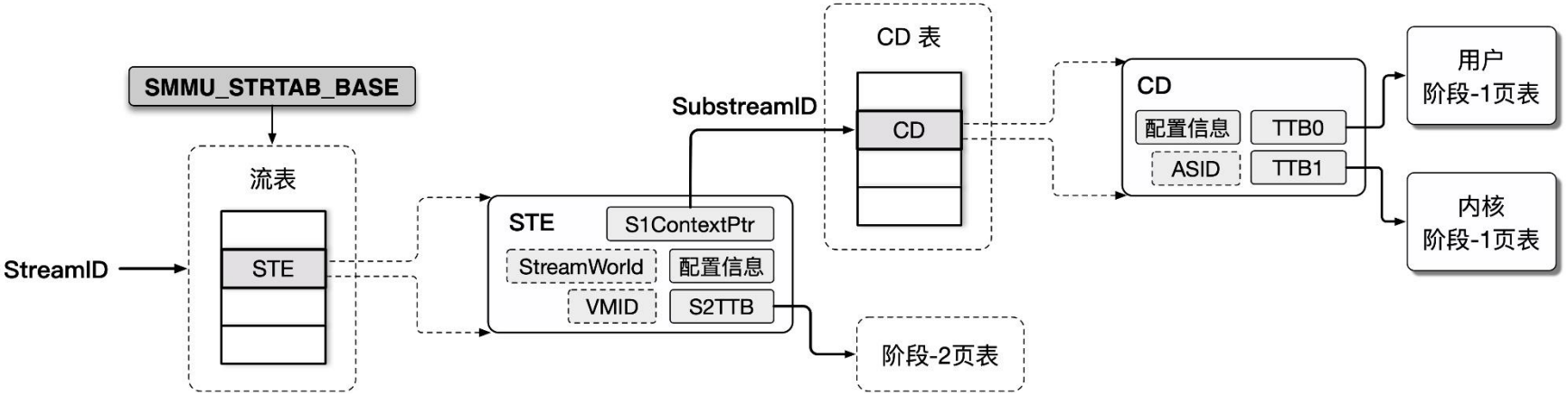


硬件实现的I/O虚拟化—— SMMU



- SMMU
  - 一种DMA重映射机制
  - IOMMU在ARM-V8架构下的解决方案，与VT-d类似
  - VT-d使用专用的I/O页表，SUMM与MMU共用一套阶段-2页表

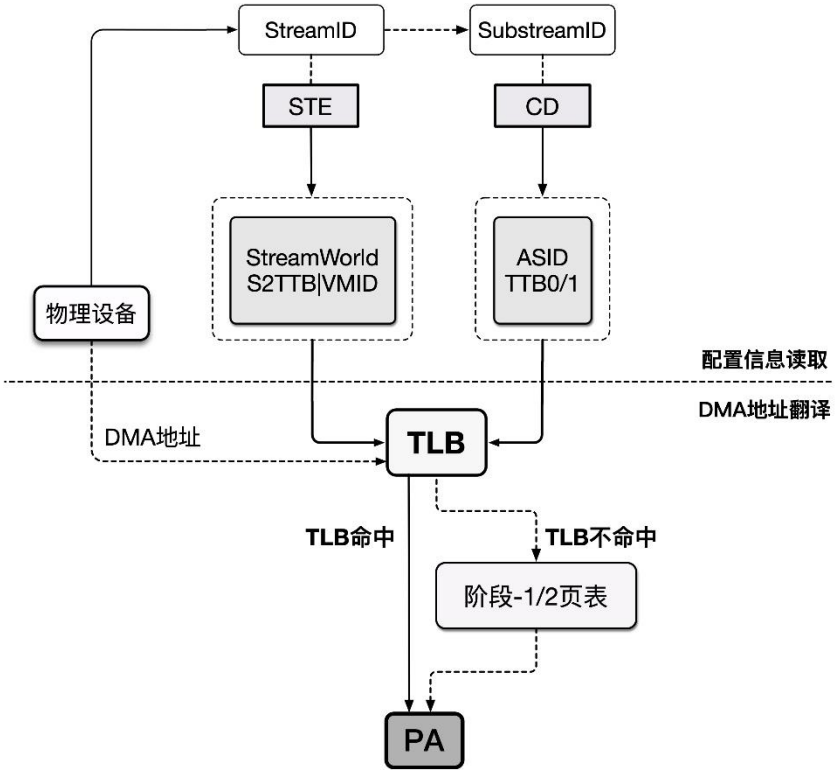
硬件实现的I/O虚拟化—— SMMU



- 流表：每个流表项对应一个设备
  - VMID:设备所属虚拟机
  - S2TTB：阶段-2转换页表基地址
  - S1上下文指针：指向阶段-1上下文描述符表(CD 表)
- CD表
  - TTB0 TTB1:分别保存用户空间和内核空间阶段-1页表
  - ASID：用于标记进程的地址空间

SMMU中的缓存机制

- SMMU从I/O 事务中获取设备标识符，即StreamID
- SMMU从SMMU_STRTAB_BASE寄存器中获取流表的基地址，并通过StreamID获取对应的ST
- 在开启阶段-1转换的情况下，通过SubstreamID定位到对应的CD，进而获取ASID和阶段-1页表基地址。在开启阶段-2转换的情况下，在STE中获取VMID和阶段-2页表基地址以及Stream World配置信息
- SMMU根据DMA地址、ASID、VMID、流世界查询TLB。如果TLB命中，可以直接获得目标物理地址以及访问权限信息。如果TLB未命中，通过相应地址翻译过程获得对应的目标物理地址，并将映射关系填充到TLB中
- 设备根据目标物理地址进行数据传输



目录 Contents

QEMU设备模型实现——edu设备为例

- QOM（QEMU 对象模型）
  - 同一类型设备间存在通用属性
  - Eg. 网卡->PCI设备->设备
  - C语言基础上实现的一套面向对象机制
- QOM中对象初始化流程
  - 将 TypeInfo 注册为TypeImpl
  - 创建对象类
  - 创建对象实例
  - 具现化对象实例
- edu设备
  - QEMU提供一个虚拟PCI设备

将 edu设备TypeInfo 注册为TypeImpl

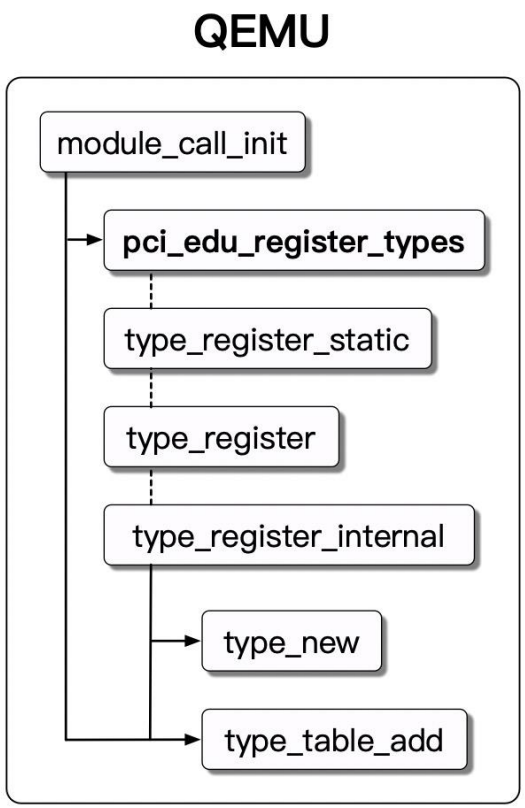
- TypeInfo
  - 对象类的描述
  - 类名、父类名、类初始化函数、类实例大小等描述性信息
  - edu设备代码中静态定义TypeInfo即edu_info
- TypeImpl
  - 由TypeInfo注册得到，存储在全局type_table中，用于创建对象实例
- type_init函数
  - 创建一个ModuleEntry，存储初始化函数指针type_register_static(&edu_info)

qemu-4.1.1/hw/misc/edu.c

```
static void pci_edu_register_types(void)
{
    static InterfaceInfo interfaces[] = {
        { INTERFACE_CONVENTIONAL_PCI_DEVICE },
        { },
    };
    static const TypeInfo edu_info = {
        .name          = TYPE_PCI_EDU_DEVICE,
        .parent        = TYPE_PCI_DEVICE,
        .instance_size = sizeof(EduState),
        .instance_init = edu_instance_init,
        .class_init    = edu_class_init,
        .interfaces = interfaces,
    };
    type_register_static(&edu_info);
}
type_init(pci_edu_register_types)
```

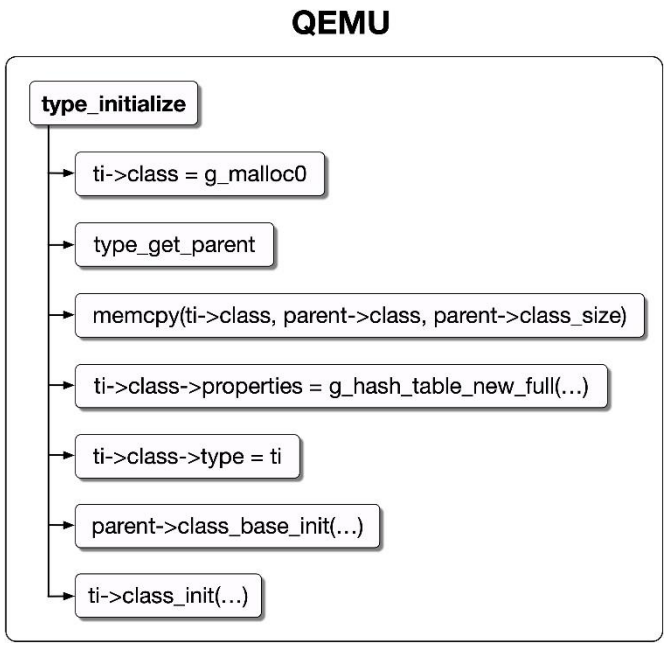
将 edu设备TypeInfo 注册为TypeImpl

- module_call_init函数
  - 遍历ModuleTypeList中的ModuleEntry并执行其存储的初始化函数。
  - 对于edu_info而言，其对应的初始化函数就是type_init函数传入的函数指针即type_register_static(&edu_info)函数
- type_register_internal函数
  - 调用type_new函数将TypeInfo转化为TypeImpl
- type_table_add函数
  - 注册TypeImpl到全局的type_table中



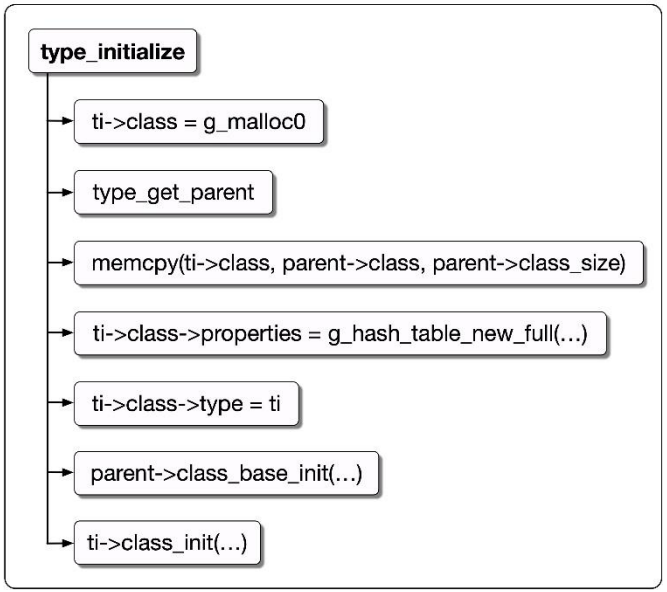
创建edu设备对象类

- 创建对象类的两种方式
  - 主动调用，object_class_get_list
  - 被动调用，object_new_with_type，object_initialize_with_type……
  - 两种方式最终都会调用type_initialize
- type_initialize
  - TypeImpl结构体作为参数
  - 对应对象类分配内存空间
  - 调用type_get_parent函数获取其父对象类的TypeImpl。递归调用，逐级向上初始化父对象类直至到达根对象类ObjectClass



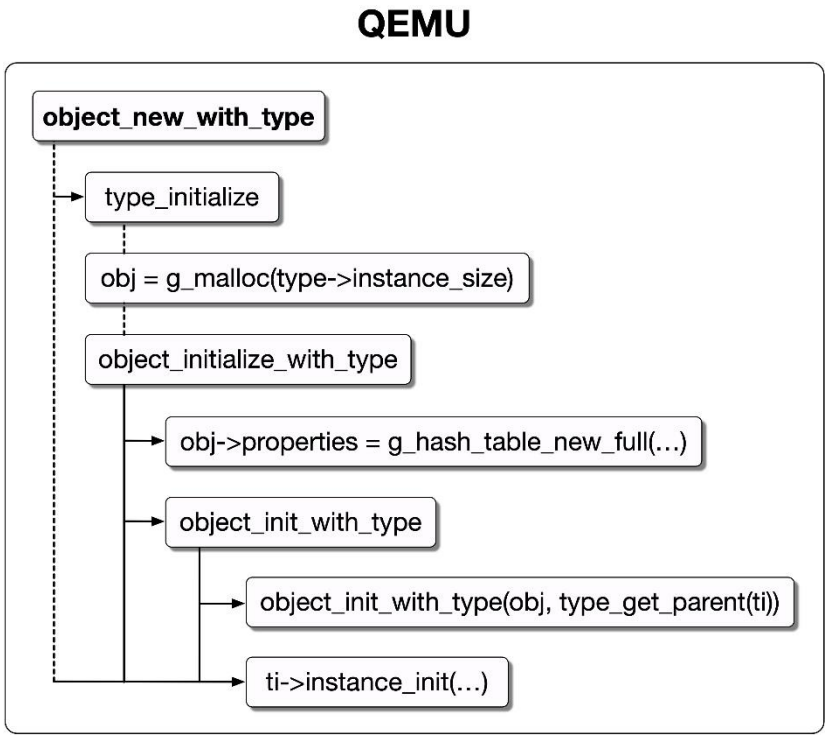
创建edu设备对象类

- memcpy
  - 将父对象类复制到其内存空间的前半部分
  - 目的：实现父类和子类直接的转换
- parent->class_base_init()
  - 初始化父类
- ti->class_init()
  - 即edu_class_init 函数，设置edu设备的厂商号与设备号等信息
  - 设置edu设备的具现化函数 pci_edu_realize



创建edu设备对象实例

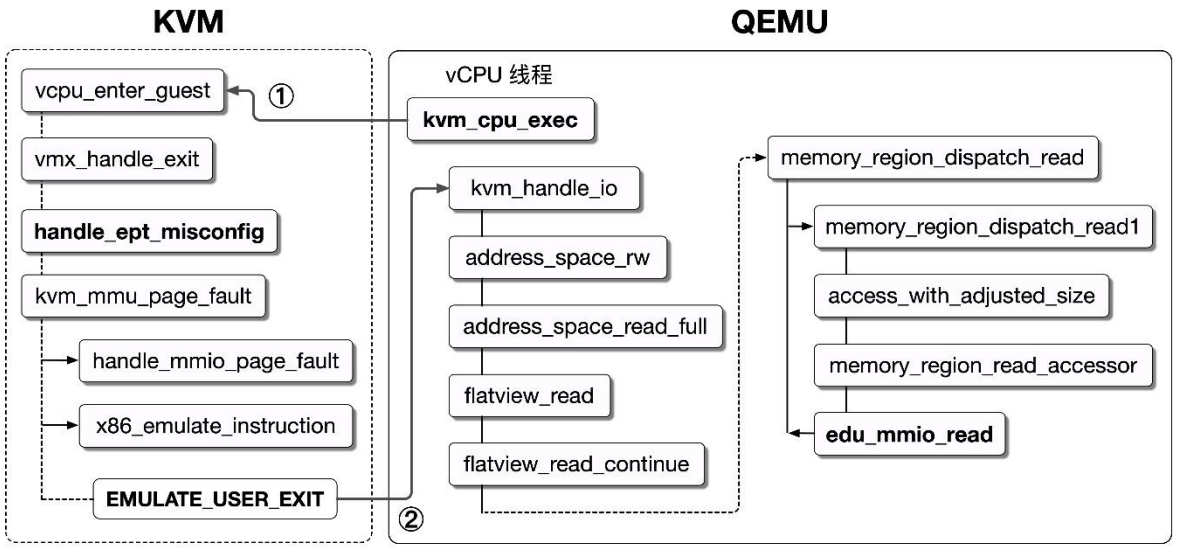
- 对象实例
  - 一个具体设备的抽象
  - 对应一个XXX State结构体，记录自身设备信息
- 创建edu设备对象实例
  - 一般的做法，启动命令行中添加-device edu参数
  - QEMU会检查到该参数后调用qdev_device_add函数创建edu设备
  - 为edu设备对象实例分配大小为sizeof(EduState)的内存空间



具现化edu设备对象实例

- EduState里的属性并未分配，必须具现化该对象实例
- edu设备具现化->pci_edu_realize
  - 初始化edu设备配置空间
    - 设置PCI配置空间中的Interrupt Pin寄存器(0X3D)的值为1，edu设备使用INTA#脚来申请中断
    - 设置PCI配置空间以开启MSI功能
    - 初始化一个MMIO内存区域，该内存区域大小为1MB，并指定该MMIO内存区域的读写函数
    - MMIO参数注册到设备配置空间的第0号BAR

edu设备MMIO处理过程



① QEMU 进行 ioctl 系统调用，进入 KVM ② 从 ioctl 系统调用返回到 QEMU

- 当虚拟机第一次访问edu设备的MMIO MemoryRegion时，由于先前没有给该MR分配RAM，因此会产生一个EPT Violation缺页异常
- KVM会检查客户机物理页号，memslots的范围中。如果不在，则将宿主机物理页号中写入KVM_PFN_NOSLOT
- KVM将对应的EPT页表属性标记上代表MMIO的特殊值，以后的读写都会引起EXIT_REASON_EPT_MISCONFIG类型的VM-Exit
- 由于edu设备是虚拟设备，需要返回QEMU中进行处理。QEMU检查exit_reason，由address_space_rw进行处理，最终回调edu设备中定义的MMIO读写函数

实验：为edu设备添加设备驱动

- 实验目标：通过添加设备驱动使用edu设备定义的功能
- 实验概述
  - 编写edu设备驱动，为用户提供设备使用接口
  - 为edu设备添加中断处理函数，并输出中断原因
  - 编写用户态测试程序，使用edu设备
- 驱动设计
  - file_operations 中的write函数和read函数按照PCI设备驱动编写的一般方法
  - 设计用于控制edu设备的多种功能的ioctl函数
  - 注册irq_handler用于中断处理
  - 修改edu设备源码，设置不同的irq_status，区分DMA读中断、DMA写中断、阶乘运算中断

启动虚拟机

- 实验第一步是在QEMU中启动带有edu设备的虚拟机，本次实验的启动参数如下

boot.sh

```
qemu-system-x86_64 -smp 2 -m 4096 -enable-kvm ubuntu.img -cdrom ./ubuntu-16.04.7.iso -device edu,dma_mask=0xFFFFFFFF -net nic -net user,hostfwd=tcp::2222-:22
```

- 进入虚拟机后，在终端输入lspci命令，根据edu的设备号以及vendor ID在pci设备列表中可以查询到edu设备被挂载到了0号总线的04号槽

lspci

```
00:04.0 Unclassified device [00ff]: Device 1234:11e8 (rev 10)
```

启动虚拟机

- 接着输入lspci -s 00:04.0 -vvv -xxxx命令会显示edu设备的基本信息，包括edu设备的中断信息、MMIO地址空间信息以及设备配置空间信息等

```
00:04.0 Unclassified device [00ff]: Device 1234:11e8 (rev 10)
        Subsystem: Red Hat, Inc. Device 1100
        Physical Slot: 4
        Interrupt: pin A routed to IRQ 10
        Region 0: Memory at fea00000 (32-bit, non-prefetchable) [size=1M]
        Capabilities: [40] MSI: Enable- Count=1/1 Maskable- 64bit+
                Address: 0000000000000000  Data: 0000
        Kernel driver in use: edu_pci
00: 34 12 e8 11 03 01 10 00 10 00 ff 00 00 00 00 00
10: 00 00 a0 fe 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 f4 1a 00 11
30: 00 00 00 00 40 00 00 00 00 00 00 00 0b 01 00 00
40: 05 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00
```

实现edu设备的五项功能

```
switch(cmd) {
    case DMA_WRITE_CMD: // 发起一次 DMA 写操作
        iowrite32(mmio+0x100, mmio + IO_DMA_DST);
        iowrite32(DMA_BASE, mmio + IO_DMA_SRC);
        iowrite32(4, mmio + IO_DMA_CNT);
        iowrite32(DMA_CMD|DMA_FROM_MEM|DMA_IRQ,mmio+ IO_DMA_CMD);
        msleep(1000);
        break;
    case DMA_READ_CMD: // 发起一次读操作
        iowrite32(DMA_BASE, mmio + IO_DMA_DST);
        iowrite32(mmio+0x104, mmio + IO_DMA_SRC);
        iowrite32(4, mmio + IO_DMA_CNT);
        iowrite32(DMA_CMD|DMA_TO_MEM|DMA_IRQ, mmio + IO_DMA_CMD);
        msleep(1000);
        break;
```

实现edu设备的五项功能

```
        case PRINT_EDUINFO_CMD: // 打印 edu 设备的基本信息
            if ((pci_resource_flags(pdev, BAR) & IORESOURCE_MEM) !=
IORESOURCE_MEM) {
                dev_err(&(pdev->dev), "pci_resource_flags\n");
                return 1;
            }
            resource_size_t start = pci_resource_start(pdev, BAR);
            resource_size_t end = pci_resource_end(pdev, BAR);
            pr_info("length %llx\n", (unsigned long long)(end + 1 - start));
            for (i = 0; i < 64u; ++i) {
            pci_read_config_byte(pdev, i, &val);
            pr_info("config %x %x\n", i, val);
    }
    pr_info("dev->irq %x\n", pdev->irq);
    for (i = 0; i < 0x98; i += 4) {
            pr_info("io %x %x\n", i, ioread32((void*)(mmio + i)));
    }
    break;
```

实现edu设备的五项功能

```
case SEND_INTERRUPT_CMD: // 写 0x60 寄存器，edu 设备发中断请求
    iowrite32(0x12345678, mmio + IO_IRQ_RAISE);
    break;
case FACTORIAL_CMD: // 发起一次阶乘运算
    iowrite32(0x80, mmio + IO_FACTORIA_IRQ);
    iowrite32(0xA, mmio + FACTORIA_VAL);
    msleep(1000);
    pr_info("computing result %x\n", ioread32((void*)(mmio +
FACTORIA_VAL)));
```

用户态测试程序运行结果

dmesg

```
[14410.274286] length 100000
[14410.274488] config 0 34
[14410.274514] config 1 12
[14410.274554] config 2 e8
[14410.274579] config 3 11
[14410.276934] dev->irq a
[14410.277161] io 0 10011ed
[14410.277269] io 4 0
[14410.277374] io 8 375f00
[14410.277631] io 20 80
[14410.277668] io 24 0
[14410.277716] io 28 ffffffff
[14410.285905] irq_handler irq = 10 dev = 245 irq_status = 12345678
[14411.303295] receive a FACTORIAL interrupter!
[14411.303362] irq_handler irq = 10 dev = 245 irq_status = 1
[14412.325635] computing result 375f00
[14412.427807] receive a DMA write interrupter!
[14412.427878] irq_handler irq = 10 dev = 245 irq_status = 101
[14414.453155] receive a DMA read interrupter!
[14414.453192] irq_handler irq = 10 dev = 245 irq_status = 100
```

用户态测试程序运行结果

- 驱动首先接收到了irq_status为0x12345679的设备中断，irq_status与pci_ioctl函数中设置的一致。

Log in QEMU

```
edu_raise_irq,irq_status=12345678,device_status=80
edu raise a irq
irqstatus is accessed irqstatus=12345678
edu_lower_irq
edu lower a irq
```

- 然后驱动再次接收到了irq_status为0x1的设备中断，并判断该中断为阶乘计算产生，最后输出了阶乘计算的结果0x375f00。

Log in QEMU

```
edu->status is been assigned a value
edu->fact is been assigned a value
edu->status is or 81 edu_raise_irq, irq_status=1, device_status=80
irqstatus is accessed irqstatus=1
edu_lower_irq
edu lower a irq
edu->fact is accessed edu->fact=375f00
```

用户态测试程序运行结果

- 紧接着测试程序会发起DMA命令，在edu设备中会设置与DMA相关的信息。pci_dma_read/pci_dma_write函数的返回值用于判断DMA操作是否成功完成。当DMA操作完成后，edu设备会返回相应DMA中断。

Log in QEMU

```
edu-dst set a value
edu-src set a value
edu-cnt set a value
edu-cmd set a value
edu_dma_timer
----------Dma write successfully-------------
edu_raise_irq,irq_status=101,device_status=80
irqstatus is accessed irqstatus=101
edu_lower_irq
edu lower a irq
```

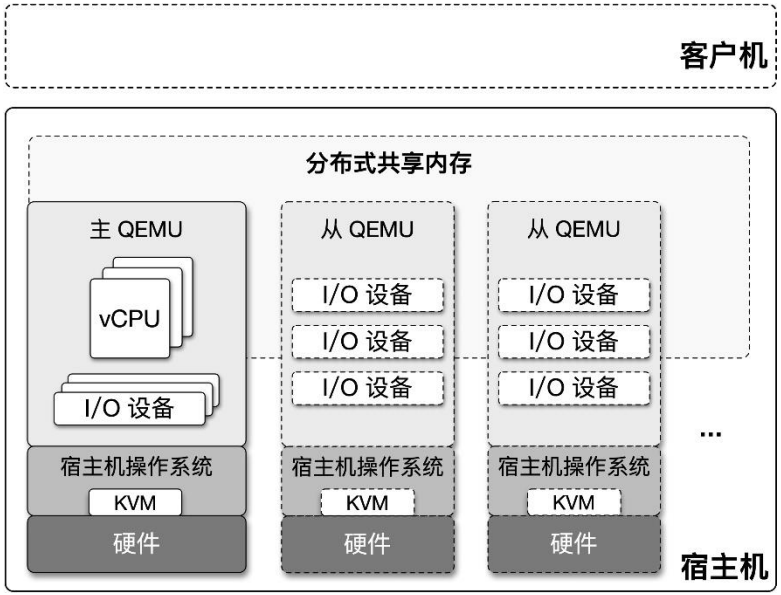
Log in QEMU

```
edu-dst set a value
edu-src set a value
edu-cnt set a value
edu-cmd set a value
edu_dma_timer
----------Dma read successfully-------------
edu_raise_irq,irq_status=100,device_status=80
irqstatus is accessed irqstatus=100
edu_lower_irq
edu lower a irq
```

目录 Contents

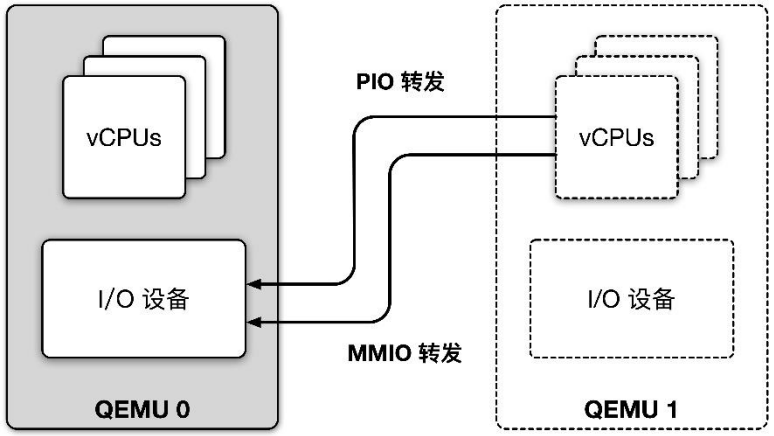
GiantVM I/O聚合

- 单台物理机
  - PCIe 数据通路（Lane）数目受到芯片组的限制往往十分有限
- GiantVM
  - 聚合多个物理机上的I/O设备
- Slave QEMU
  - I/O设备提供者
- Master QEMU
  - 注册全部I/O设备
  - I/O请求会被路由到物理设备所在的QEMU



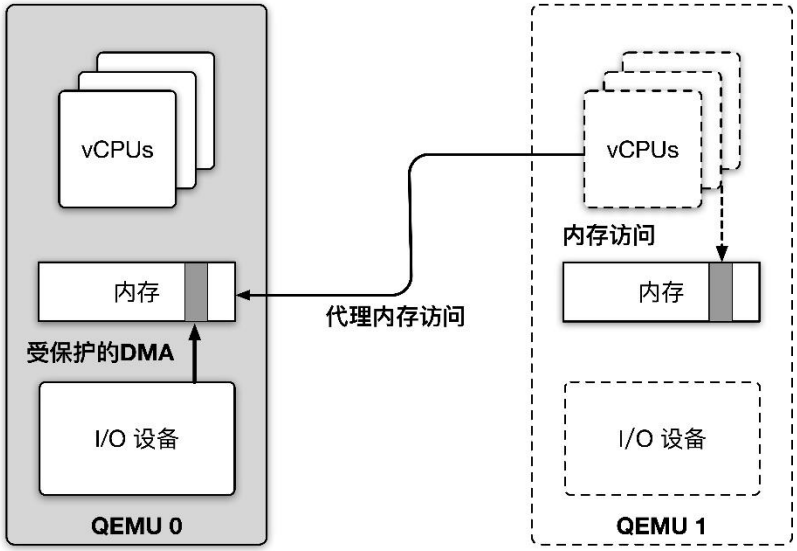
GiantVM PIO&MMIO转发

- PIO所操作的设备不属于当前QEMU
  - 根据端口号、数据、长度、方向格式化出一条网络消息发送给目的设备所在的物理节点
  - 特殊PIO：PAM0-PAM6 用于控制shadowing特性。对这类特殊的PIO操作进行广播，保证所有QEMU都能收到shadow操作，从而保证内存区域的一致性
- MMIO操作地址区域所属外设不在当前QEMU节点
  - 则由mmio_forwarding函数将其转发给设备所在的QEMU进行处理
  - 特殊地址：APIC相关区域无须转发



GiantVM DMA处理

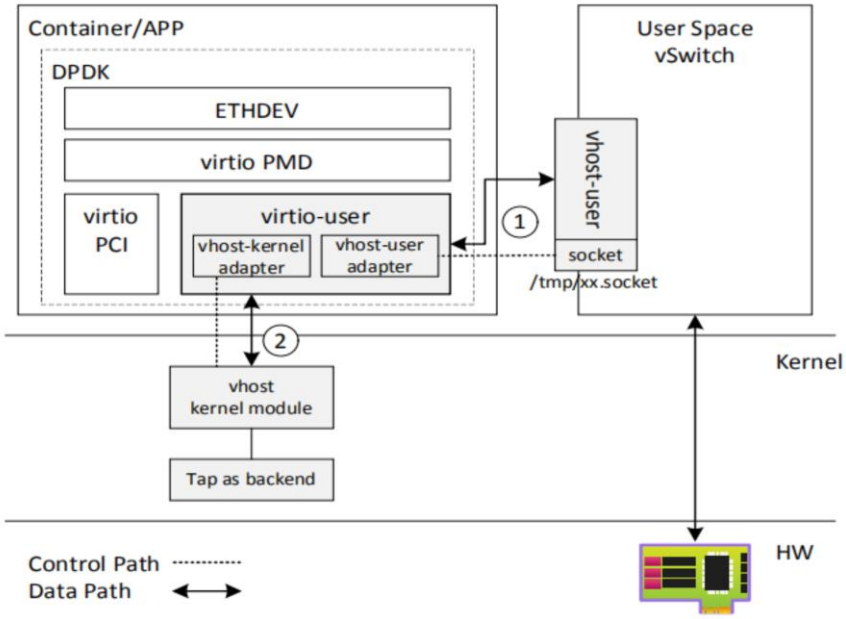
- 模拟设备DMA
  - Pin：手动获取并锁定内存的控制权
  - unpin：DMA操作完成后，解除对这块的内存的锁定
- 物理设备DMA
  - vIOMMU：对直通DMA进行权限控制
  - DMA区域升级为独占内存



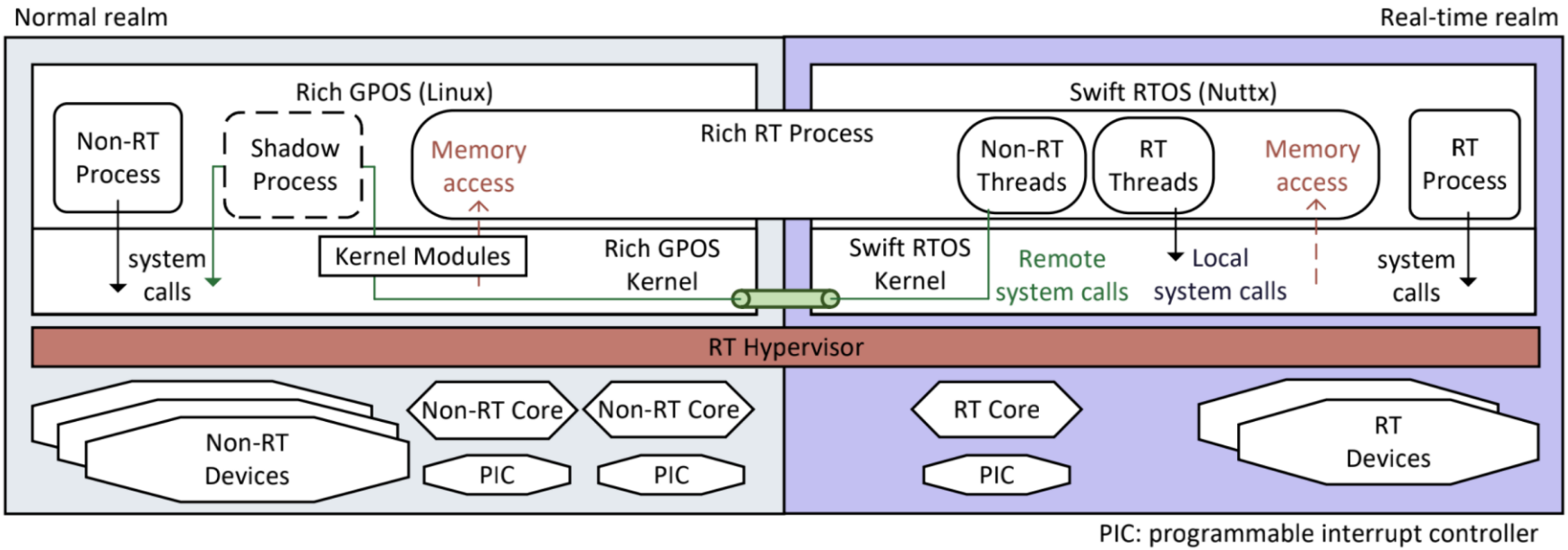
目录 Contents

VIRTIO-USER SIGCOMM 2017

- 新型虚拟设备
  - 集成在DPDK，应用于容器
  - 两种接口，vhost-net & vhost-user
  - 实现网络包kernel by pass
- 适用场景
  - 云平台网络功能虚拟化（NFV）



CRTOS VEE20



- 影子进程
  - GPOS中，负责创建Rich-realtime应用
  - 负责处理Rich-realtime应用的远程系统调用
- 设备租赁机制
  - 基于virtio system call 转发
  - RTOS中的应用->GPOS（Linux）设备

谢 谢！

本讲义内容基于

《深入浅出系统虚拟化：原理与实践》

（华为智能计算技术丛书）