[IO虚拟化 1](#_Toc105349101)

[硬件内存虚拟化 2](#_Toc105349102)

[二级翻译过程 3](#_Toc105349103)

[如何实现虚拟机级别的内存换页？ 5](#_Toc105349104)

[IO虚拟化 6](#_Toc105349105)

[设备模拟 8](#_Toc105349106)

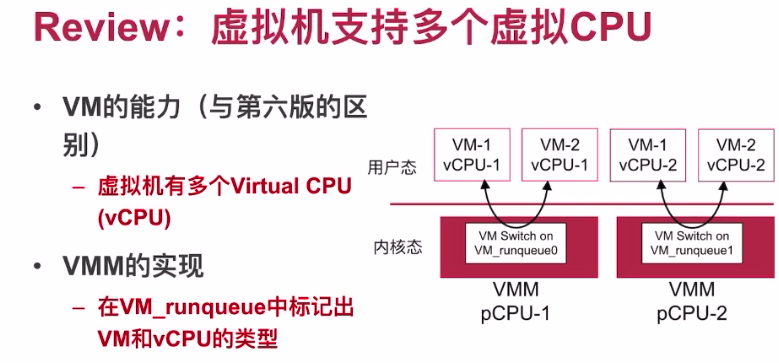
[半虚拟化 10](#_Toc105349107)

[设备直通 13](#_Toc105349108)

[SR-IOV 16](#_Toc105349109)

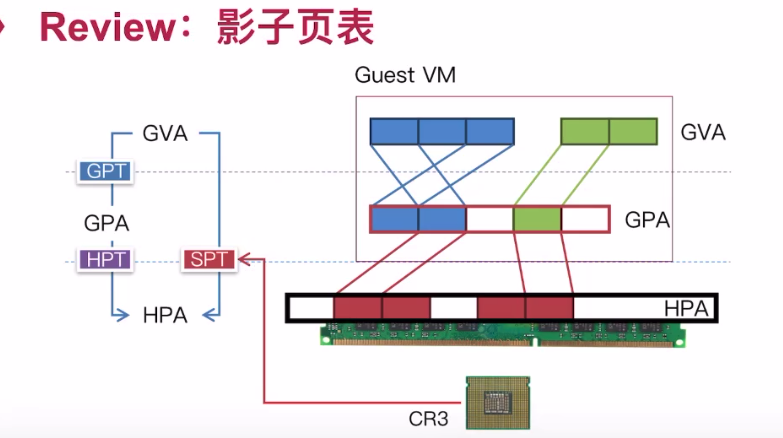
[中断虚拟化（讲的太快了。。） 18](#_Toc105349110)

# IO虚拟化



在内核中复用了线程切换的机制，只不过要在thread ctx里加一些vCPU的特殊寄存器。

cpu虚拟化的作用主要就是解决敏感指令不下陷的问题。



只有一个页表基地址寄存器如何完成两种页表翻译的过程。这就是我们之前用的影子页表的机制。写入CR3会被VMM捕捉到，然后VMM重新配置，将GPA翻译成HPA然后装入CR3的基地址寄存器。为了捕捉到guest对页表的修改，把guest对应的页表页设为read only。——这块可以参考课本，写的很详细。

对用户态实现一个页表，对内核实现一个页表。这就有俩页表。

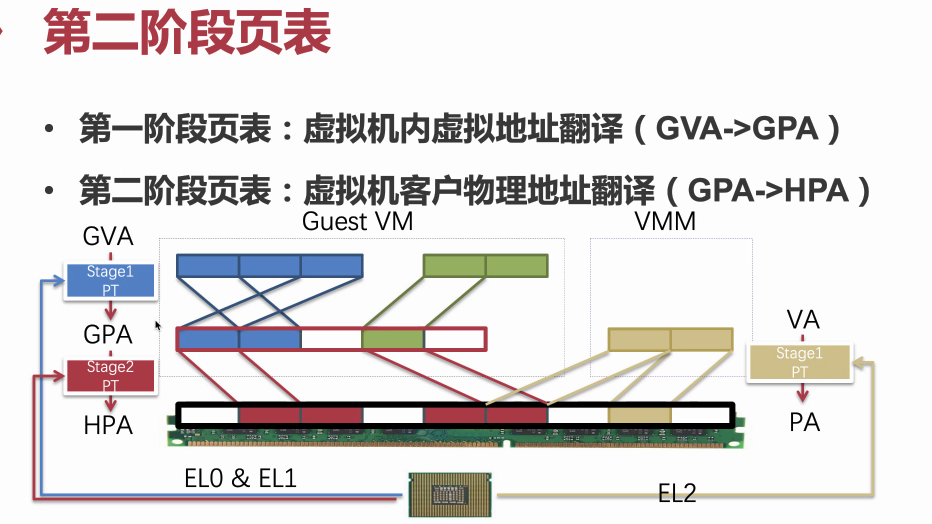
guest想装页表都会下陷VMM，10个进程有10个影子页表，但是再考虑内核和用户态的隔离必须为内核态和用户态分别创建——所以一共11个页表。

最后一个内存虚拟化的技术：硬件虚拟化，还是改硬件，之前不是硬件不行吗，只有一个页表。怎么改呢？虚拟机内部本来就有一个页表，第二阶段是底下VMM配置的页表，既然软件要两个页表，那我硬件就给你加两阶段的页表。

## 硬件内存虚拟化



这个新的由VMM控制的页表，把GPA翻译成HPA。

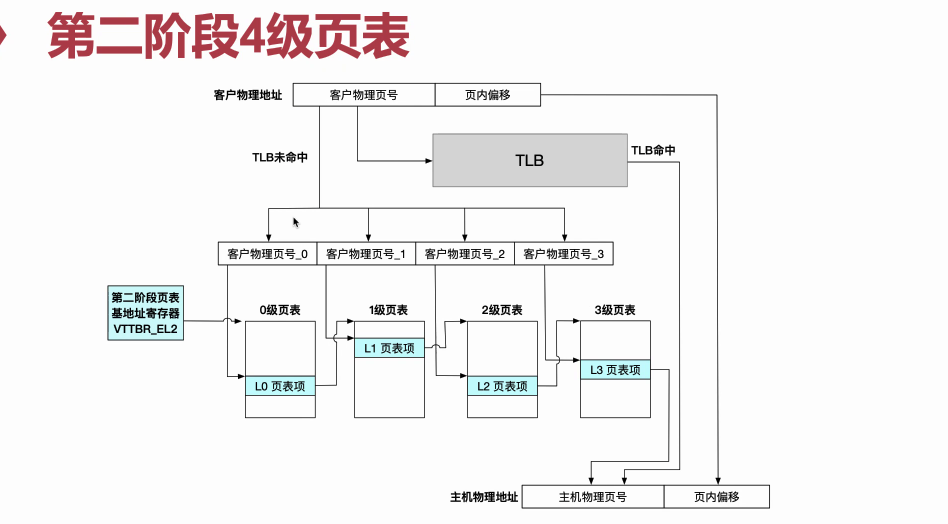


guest里GVA到GPA，完全由guest配置，不用trap。VMM只要配置stage2的页表——GPA-HPA，所有系统运行之前装上，虚拟机运行过程中每次配页表都无所谓，访存的时候翻译两次。

Q：不起虚拟机的时候VMM怎么办呢？

A：可以把虚拟化关掉，像过去一样配自己进程的页表。

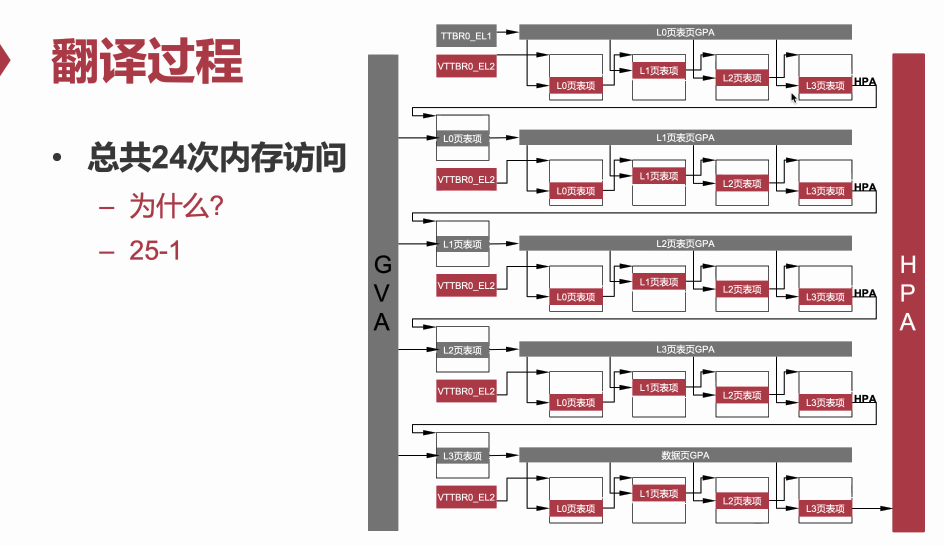
第二阶段页表结构和第一阶段完全一样。也是4级页表。



Q：TLB完全miss，一个GVA最终翻译到HPA要经过多少次内存访问？

A：是4+4吗还是4\*4，其实是25 - 1 = 24次。首先补充guest配置第一阶段页表的时候，每个页表项都是物理地址（GPA），GPA是不能直接访存的，要经过第二阶段翻译。

### 二级翻译过程



在一阶段里拿Lx页表项也要读。

stage 2页表存在VTTBR\_EL2：

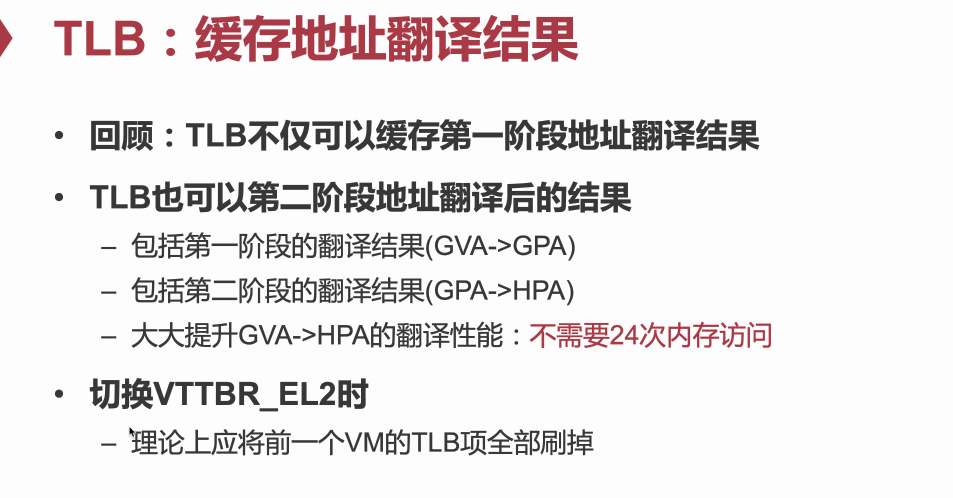


有了硬件虚拟化的第二阶段页表，4个vCPU，10个进程要配置多少个第二阶段页表呢？

答案是1个：因为是整个虚拟机范围内控制的，硬件只要保证一个GPA对应一个HPA就行了。

当有TLB的时候可以缓存两个阶段的翻译。

什么时候有问题：特殊workload：访存范围特别大，且随机还会导致大量的24次访问的性能开销。

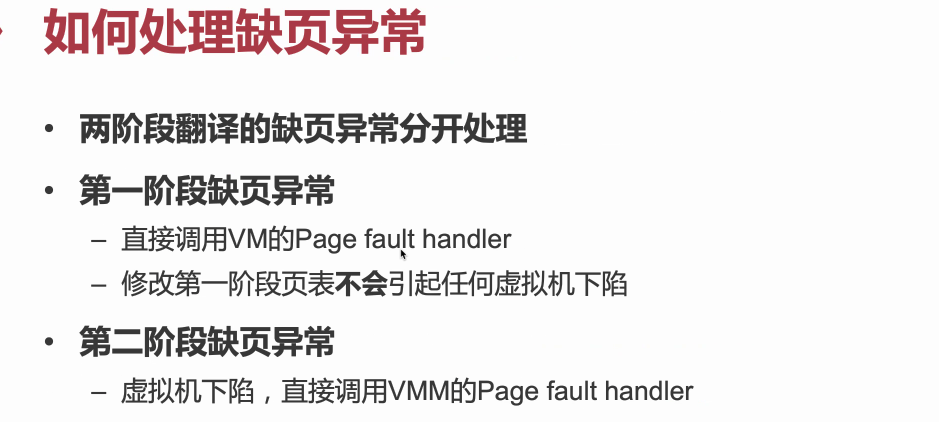


现在TLB里有tag，每个跟特定VM相关的tlb项都有一个vm id——VMID。所以不用刷了。

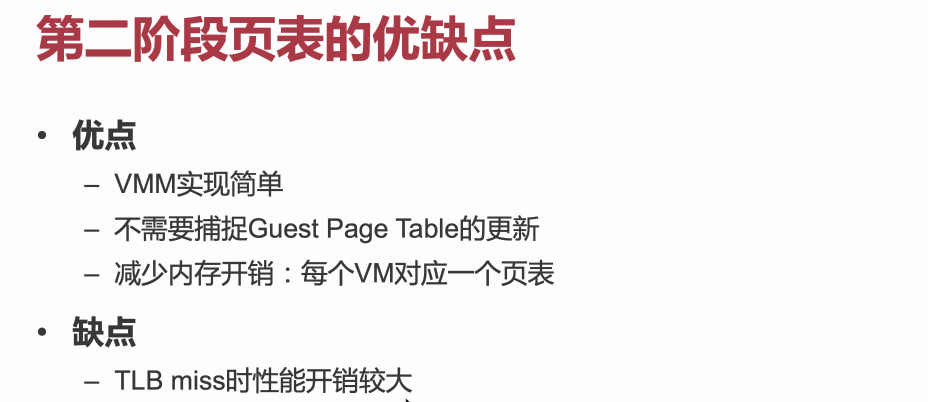


之前提到影子页表的时候，要改第一阶段页表会下陷然后触发同步，VMM已经介入了guest内部缺页异常的处理过程。

第一阶段缺页vm自己处理，第二阶段的缺页再vmm来处理。



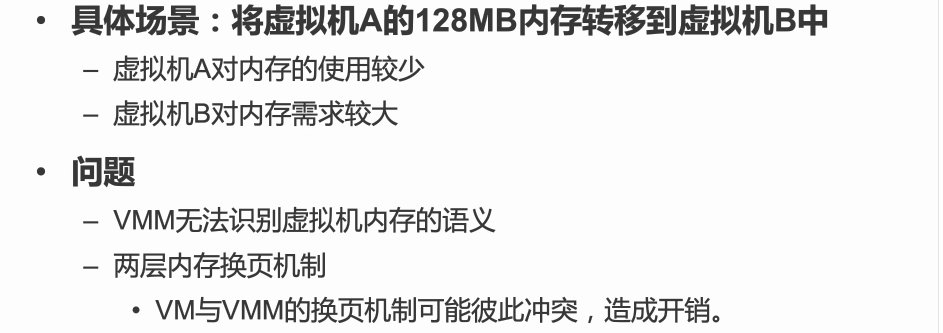
如果是GPA到HPA的缺页，这个是VMM维护的，所以应该触发一个stage 2的缺页，然后触发vmm的handler，这个过程应该对guest透明的。

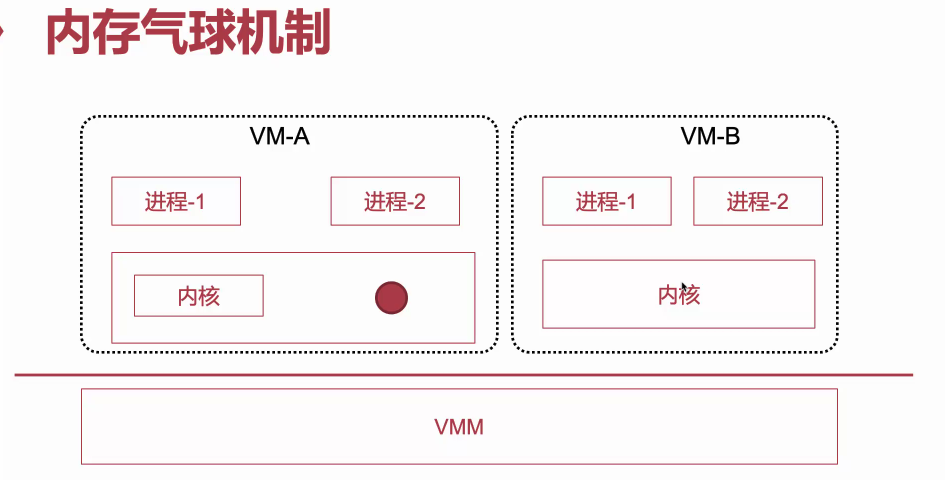


也不用像direct paging，打破虚拟化的透明性，guest的kernel也是不用改的。

有了二阶段页表后，VMM有了新的能力，所以很自然的想法能不能像os内部一样实现内存换页呢？4G物理内存，两个vm各用3G，能不能做到内存超售呢？——内存换页，暂时用不到的页存到磁盘上去。

### 如何实现虚拟机级别的内存换页？





LRU：最近最少用的内存从进程1 unmap掉（之前先把数据保存到disk上），把present bit置空。然后把这个页给进程2。

问题在于有两个换页的机制存在：VMM做LRU好还是vm内核做LRU好——哪个看到的信息更多，其实vm内核看到的信息更多，它有对于每一个内存页的更多的语义。

第二，guest内部有换页机制，这两个可能会造成冲突。虚拟机1的一些数据页已经给了2了，自己的页已经空了，然后它访问这个页，触发page fault，进vmm，vmm又把页从磁盘上读起来——然而它想读这个页的目的也是为了自己的换页，存到虚拟磁盘上——这就有可能造成冲突。如何解决呢？

内存气球机制：

我给guest装一个驱动，kernel module，正常的时候啥也不做，当vmm决定换页的时候，它就可以通知vm（怎么通知呢？upcall（与之前的hypercall相反的方向））vmm调用它，它直接取调内核的内存模块（kmalloc拿到128M空间），然后这个驱动把这个页的地址直接交给vmm，vmm拿到之后直接取guest里面unmap掉，present bit置空之后直接交给虚拟机B。这里不用存磁盘，因为这些页是调kmalloc分配出来的（这些值是没有用的）。vmm可以借用这个特殊语义减少一次写磁盘操作。

这个机制的cons & pros：

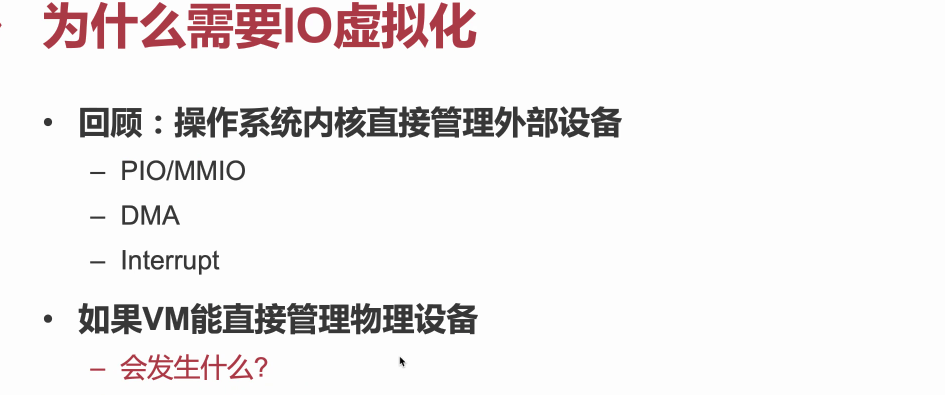
pros：不用大改guest内核。只要装一个kernel module即可。也不用改guest内部的LRU机制。通过kmalloc接口，如果内核内存空间不够，它会触发vm内核的swapping机制。这就变相触发了内核的LRU。一定程度上借用了内核的LRU机制。第三个好处：vmm不用存磁盘了。还有一个这个空间也可以直接还给虚拟机（气球变小）。

为什么叫bullon：两边配合增减这个气球的大小。

这就是全部内存虚拟化的内容了。

# IO虚拟化

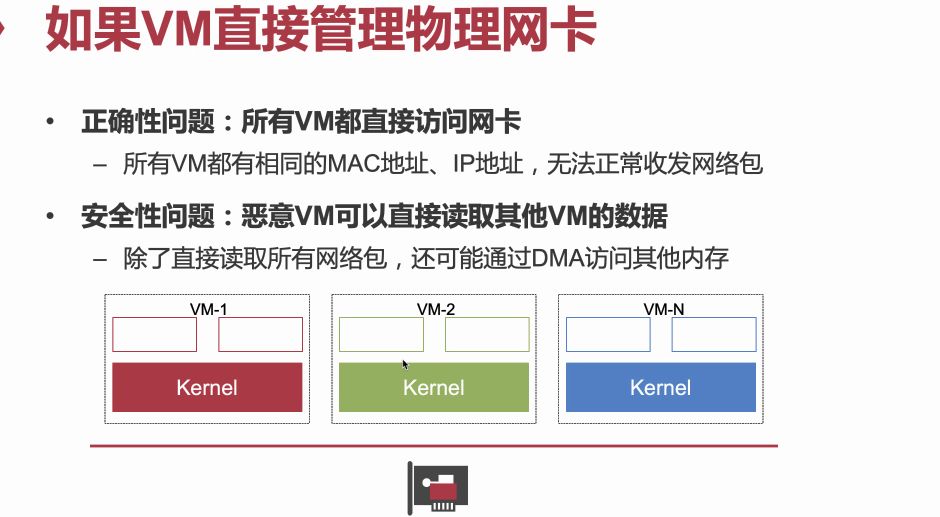
io虚拟化怎么做呢？和前面大同小异。



intel PIO

arm MMIO

数据交换：dma：异步，脱离cpu的行为

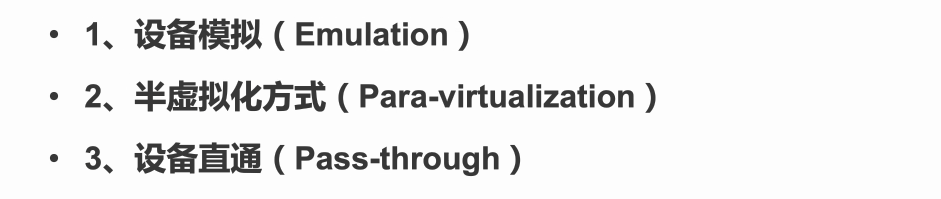


所有的虚拟机都有相同的MAC地址。

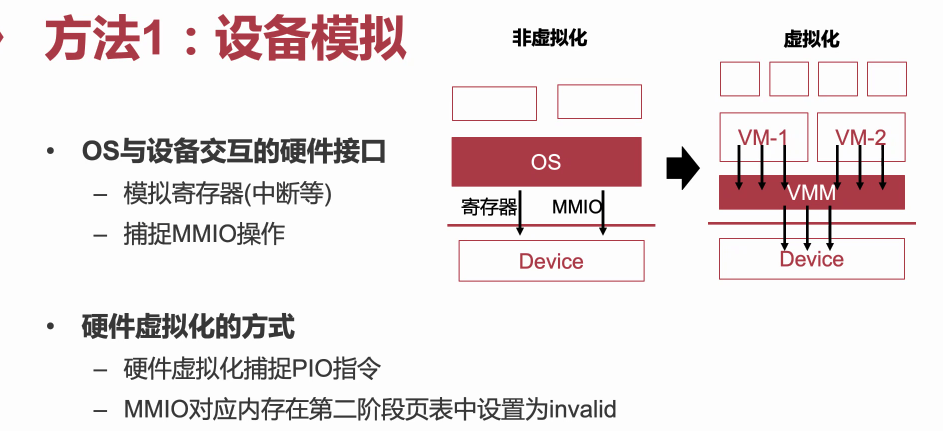
安全性：直接看到物理网卡可以看到其他人的网络包，对于网卡磁盘都适用。如果虚拟机A发出要求要格式化磁盘。



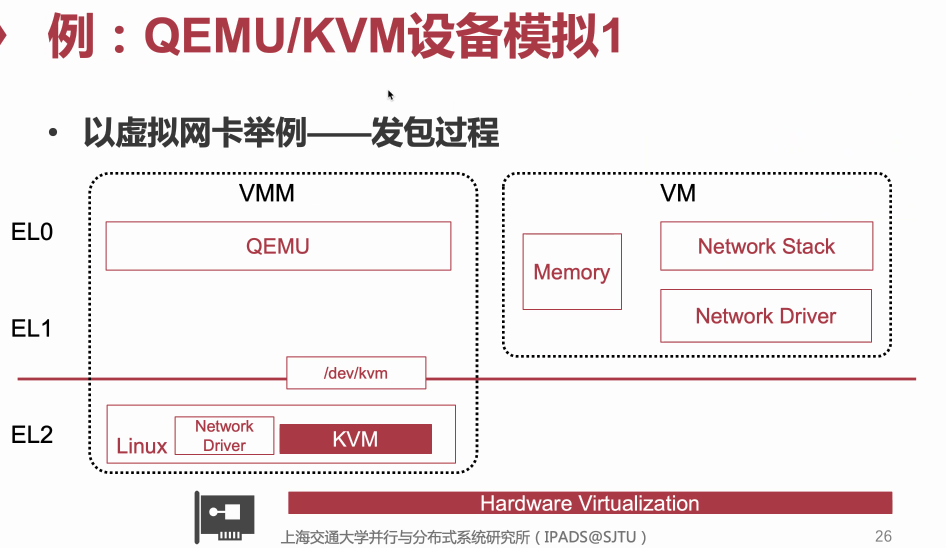
比如数据中心网卡可能有400G，每个vm可能只要20G。



### 设备模拟



用软件的方法把这三种接口都虚拟化出来，进而捕捉所有guest对设备的访问。



KVM出来之前qemu就是用软件的方式模拟设备，但世界上的驱动何其多，模拟所有的设备开销很大，所以qemu的策略是对大量使用的设备做模拟。为什么可以不做模拟呢？虚拟机内部的os有很多很多驱动，它启动的时候会看这个硬件上有什么设备。qemu只要告诉这个vm它有哪些设备就可以了。

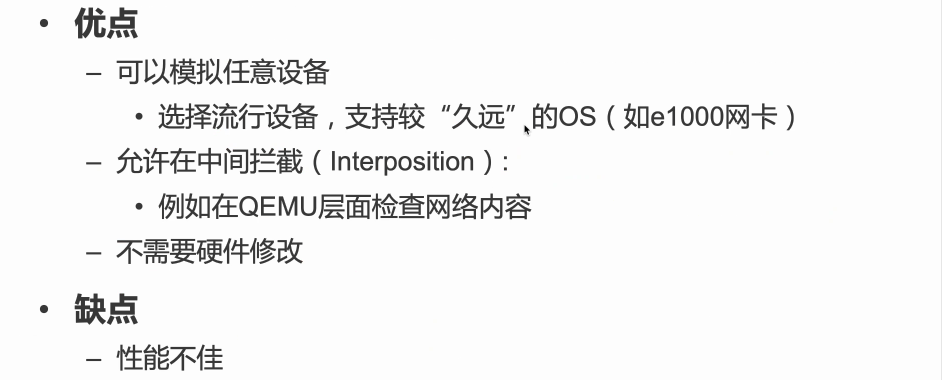
由qemu模拟E1000这个网卡，它如何配合KVM捕捉对网卡的访问。

看发包过程：VM exit MMIO一定会触发一次下陷，KVM看这次是MMIO的下陷，它拿到之后不知道怎么办？因为这是qemu主动去配的缺页。kvm只要恢复之前ioctl的位置就可以了。它要做的事是直接读虚拟机内存，qemu拿到真正的网络包之后调真正的system call。有两个问题：guest内部有一个网络协议栈，经过qemu又要调syscall，又要层层打包发出去，能不能做一些优化——有一种特殊类型的socket，raw socket可以直接把qemu准备的包发出去。但是这个优化方法有一个问题：要给qemu特殊的权限。

Q：第二个问题：为什么qemu能直接读到虚拟机内存呢？

其实这个虚拟机就是qemu创建出来的，这些所有内存都是qemu从kernel那通过mmap拿到的。

收包过程就是反过来的：网卡把数据从linux网络协议栈读出来，看看里面的IP是给虚拟机的，于是做个操作——把这个包扔到虚拟机里面，它要通知虚拟机（插入中断——调用虚拟机注册的interrupt handler，进到kvm里，然后进到VM里的interrupt handler，它看到自己有未接受的包）这里还有一个问题：所有的包先到驱动然后再进qemu用户态：这个路径很长。能不能做优化。

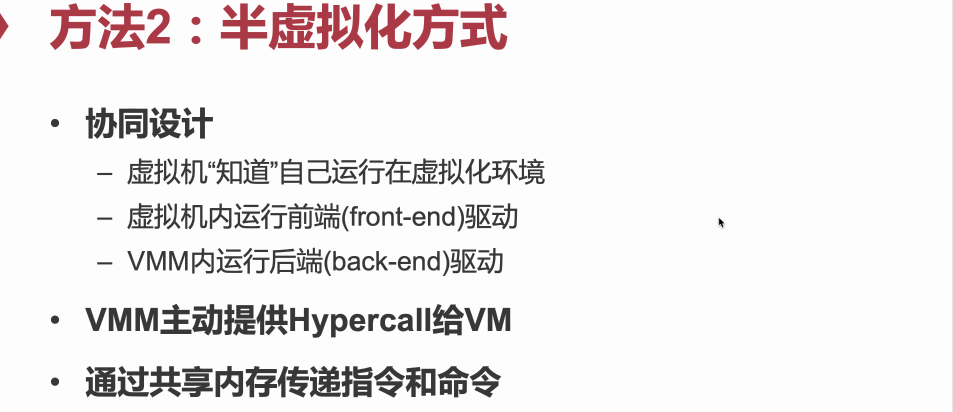


所有的dma，mmio都要经过qemu检查。不用在虚拟机内部做防火墙。

现在跑虚拟机不会对网卡，磁盘模拟，只会对不在乎性能的设备做模拟，比如console。

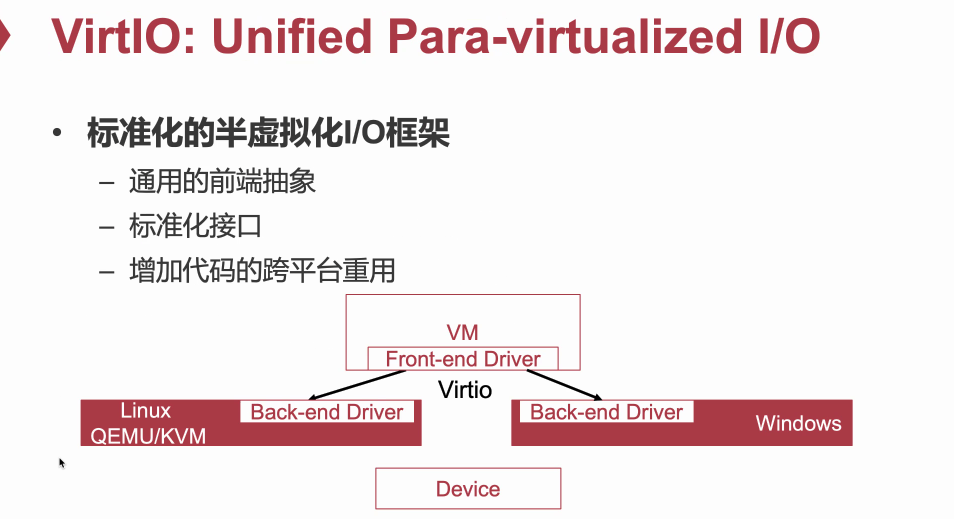
### 半虚拟化

课本P354



让虚拟机知道自己运行在虚拟化的环境里：Xen：在IO设备做拆解。

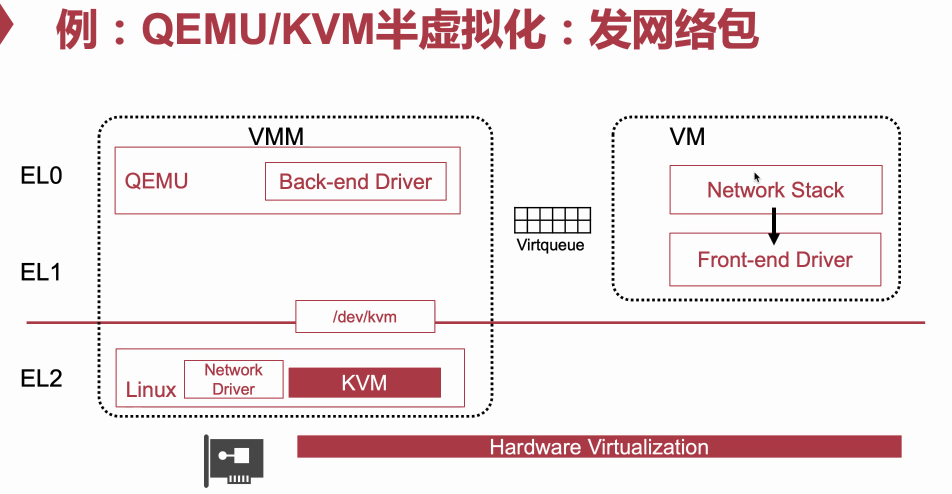
前端设备不用MMIO，DMA等等做通信。



形成了一套标准：不同的设备如何定义frontend，如何定义backend。只要遵循标准：backend可以跑在软件模拟的环境里——现在的趋势是能不能把backend搬到device上（device计算能力越来越强）

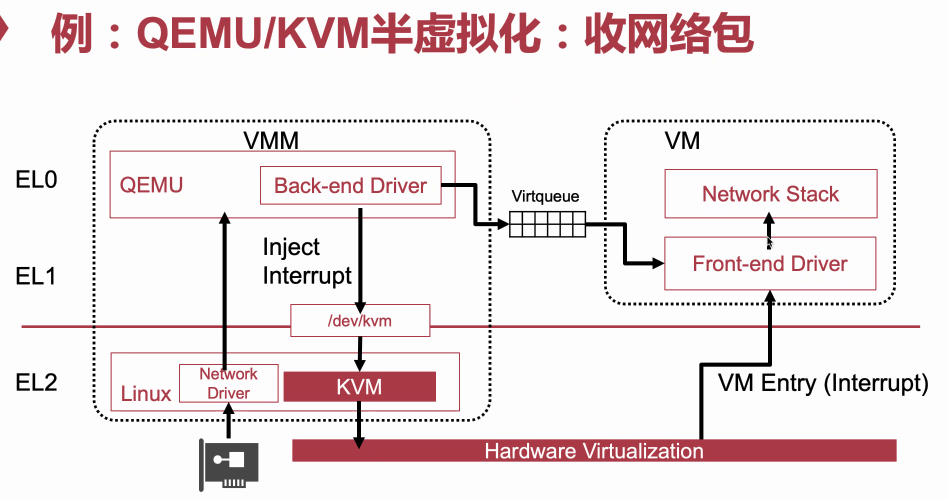


队列中的每个entry放什么什么都是有定义的：

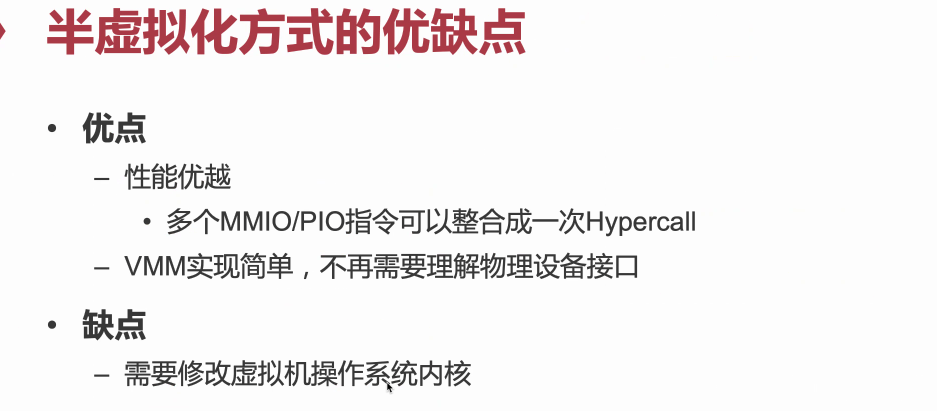


把这个网络包放到一个内存里面：next（一个entry放不下可以放好几个）主动调hypercall，进KVM，直接把控制权交给qemu（ioctl回复），backend去查virtqueue，然后再去发syscall通过host网络协议栈发出去。

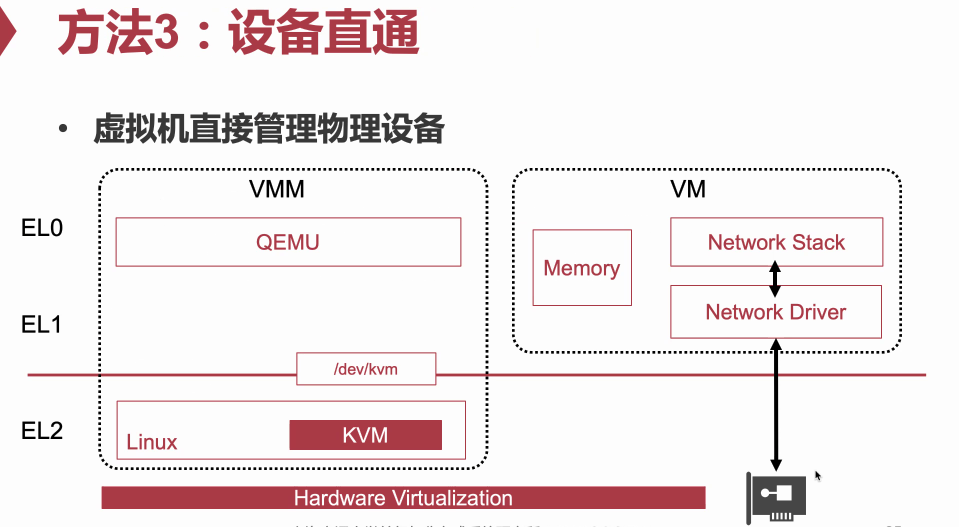
为什么快？Virtualqueue的设计：之前非常细粒度，每次dma都要转换，但现在知道跑在虚拟化环境里，可以做batch，把好多包窜在一块发出去，很多数据一起丢给VMM。之前很多次的round trip变成一次round trip。



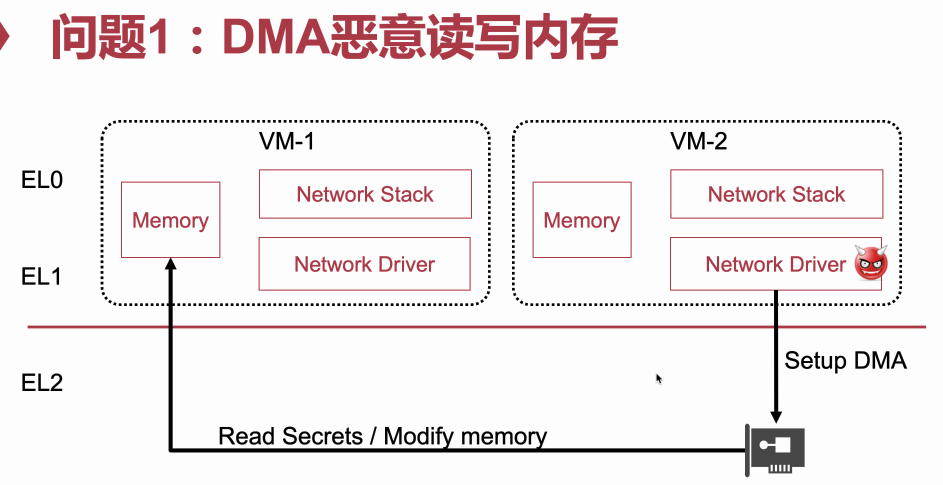
数据通路转发对于IO虚拟化造成不小开销：redhat提出了一个vhost，这个backend不用在qemu里，可以实现在kvm里。这就不用经过上面一层qemu了。为了性能，不得不把有些东西拉回来（按照架构来说应该放在qemu里）



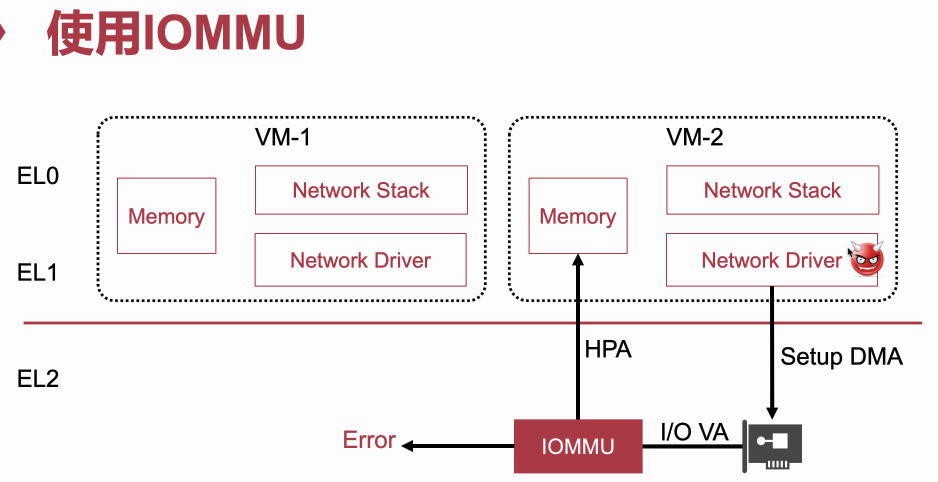
### 设备直通



把mmio的地址范围映射给虚拟机



dma是绕开cpu的，guest就可以用dma绕开cpu的检查，甚至可以直接写其他VM的内存。CPU里有页表，这里没有页表，那我再这个数据通路上加个页表不就好了吗？



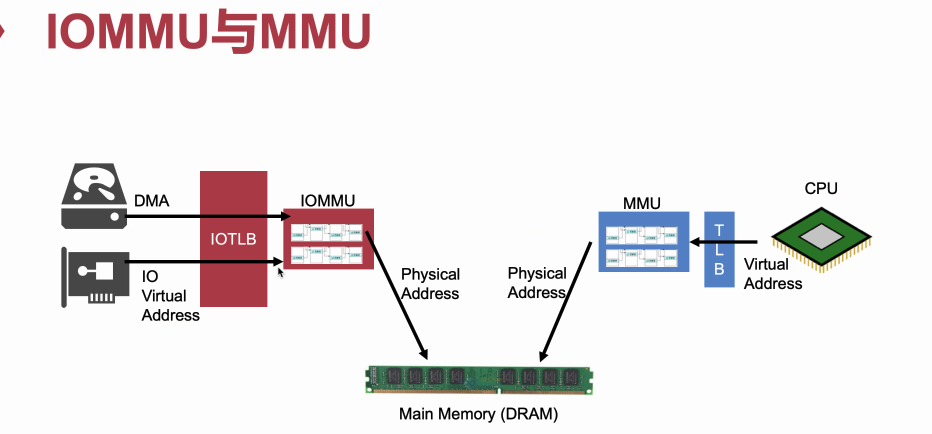
这个页表把GPA翻译成HPA。

这个IOMMU是在主板上的。由VMM配。

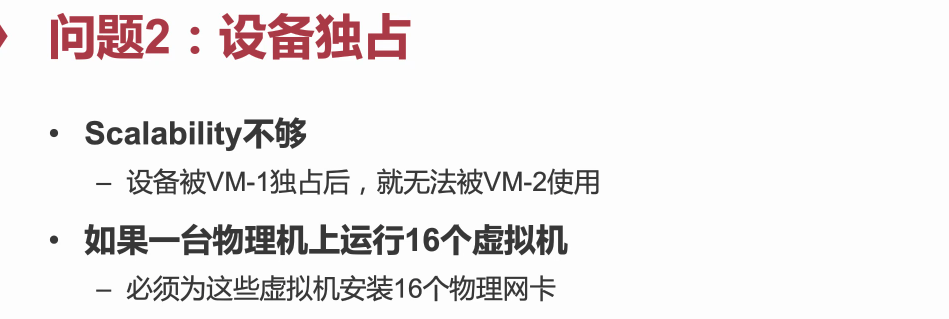
这个页表到底怎么配呢？该配什么内容呢？

最简单方法：直接用CPU配的stage 2的页表不就好了？

既然要维护一个统一的GPA的抽象。主流的KVM实现，直接把给CPU的stage2的页表直接扔给IOMMU。



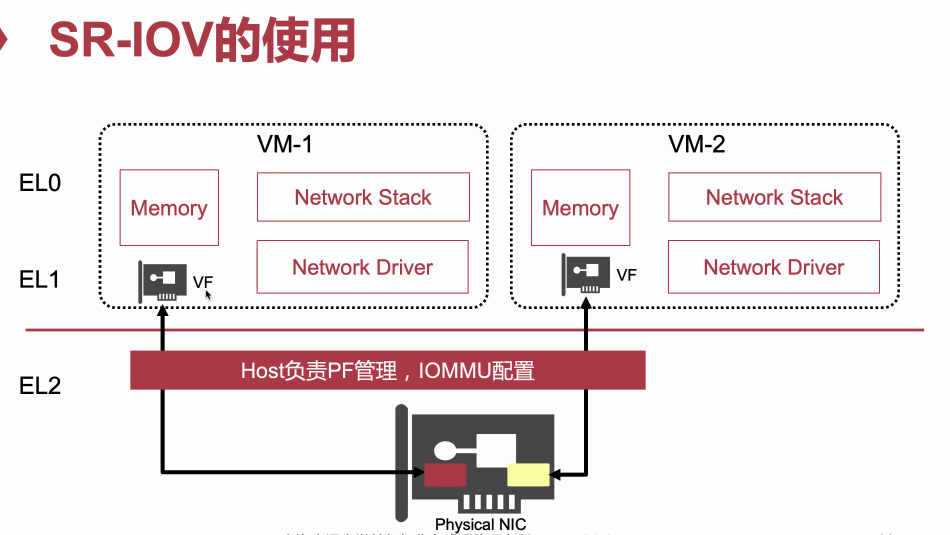


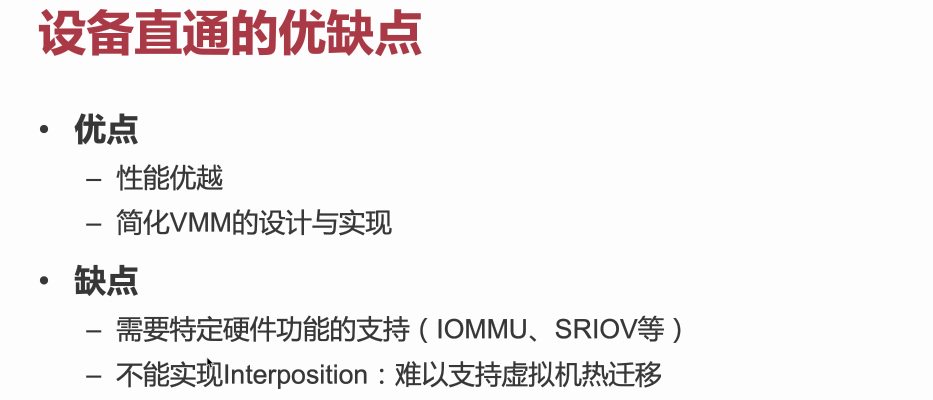


设备的厂商确立了一个规范，在设备层面做了一个虚拟化：即使是买了一块网卡，插在电脑上可以看到多个虚拟网卡。

### SR-IOV



、每个虚拟机都有独立的地址空间



backend都不需要，直接在虚拟机里装上对应设备的驱动就好了。

缺点依赖设备（逆序满足相应标准）

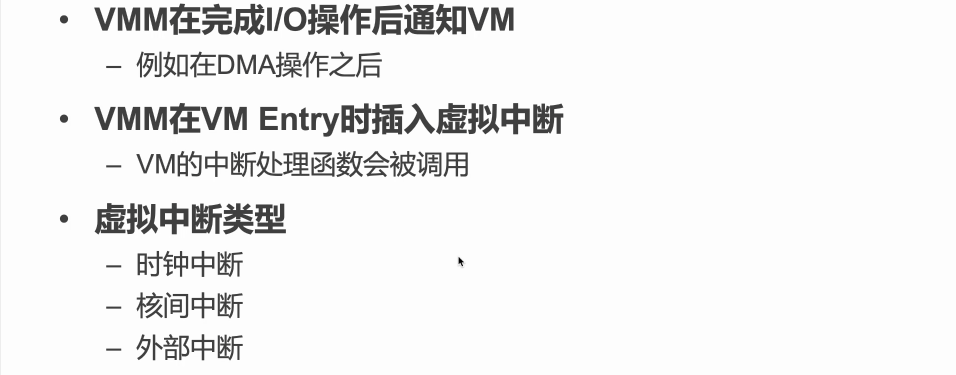
IOMMU仍然有一些问题

虚拟机迁移非常重要：服务器更新依赖热迁移把vm迁到新的服务器上面。

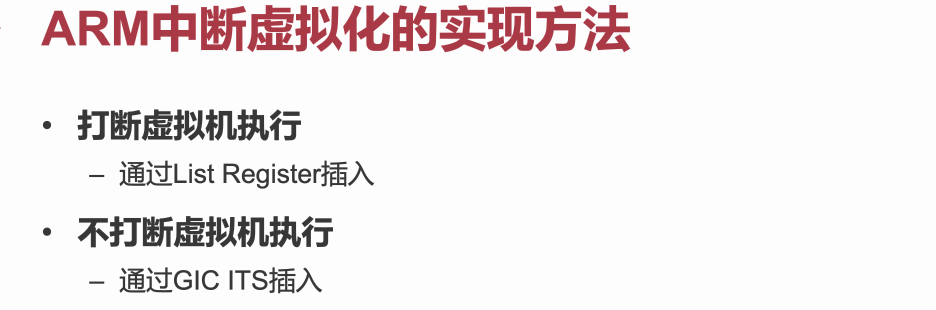


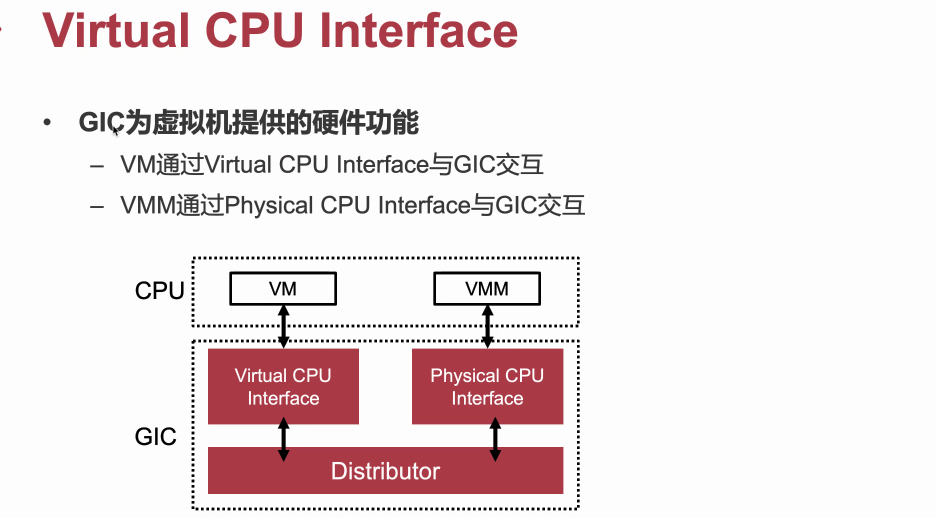
追求极致性能：云上也提供了设备直通。现在虚拟机里很多设备，用了各种技术：比如串口用的是模拟，网卡用的是半虚拟化，GPU用的是设备直通。

# 中断虚拟化（讲的太快了。。）



硬件提供了中断虚拟化的支持。





GIC做到了虚拟机看到的是virtual的虚拟机接口，vmm看到的是

过去插入中断要改guest ip然后跳转，有了GIC之后。



虚拟机最开运行的时候，VMM把物理中断和虚拟机内部中断做一个连接，物理中断来了，直接进GIC，整个过程不用VMM干预。过去先进到GIC然后VMM里触发一个中断触发虚拟机下陷。有了ITS直接就把翻译过程完成了，不经过VMM的翻译。不打断虚拟机的执行，也是未来的主流。

以上就是系统虚拟化的所有知识。