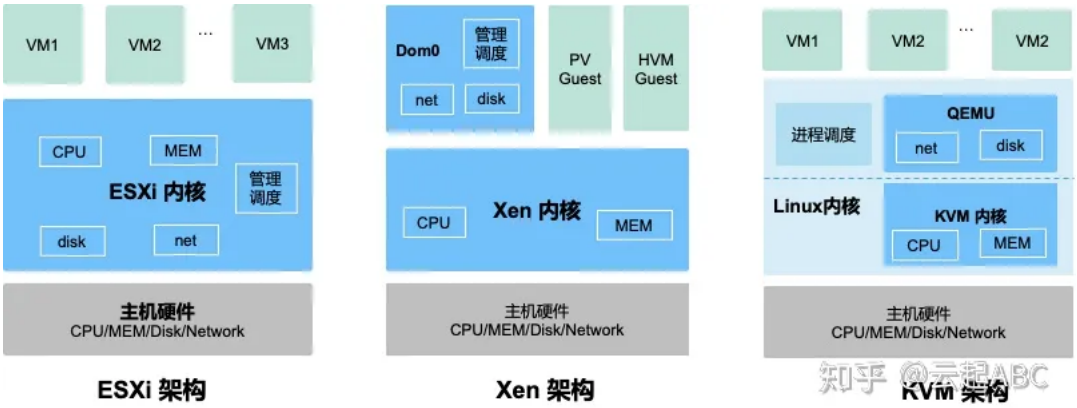
IO虚拟化

- 陈渝老师的课中，IO虚拟化这部分压根没有理解，正好在学习用户态的过程中完备了这部分的理解，先从2篇博客文章入手

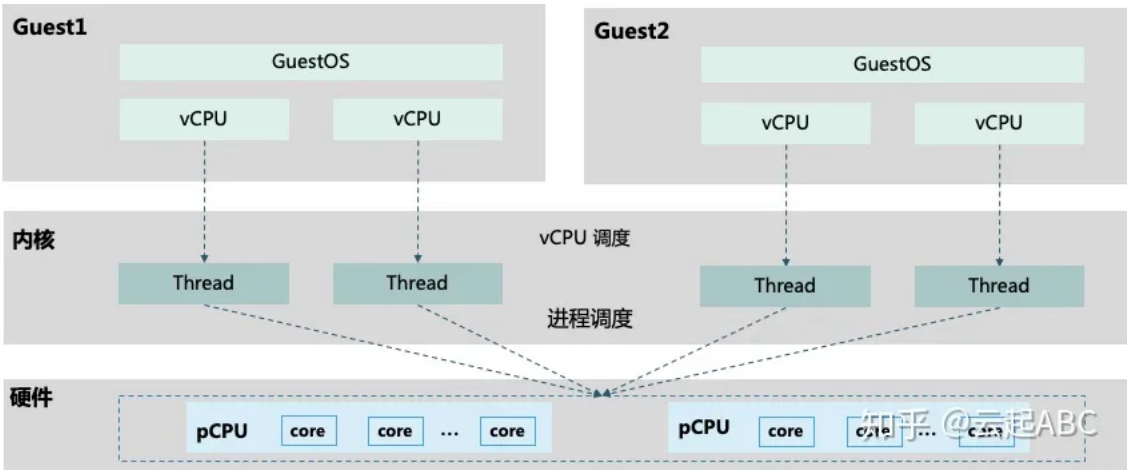
KVM 虚拟化详解

- ref
  - https://zhuanlan.zhihu.com/p/105499858
- **服务器虚拟化**是云计算最核心的技术，而**KVM**是当前最主流的开源的服务器虚拟化技术

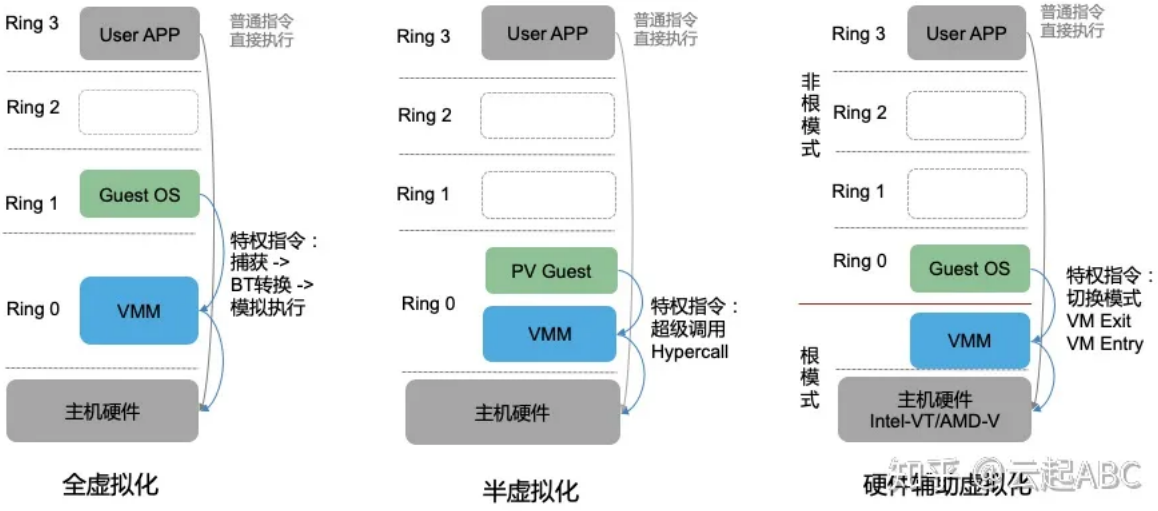


- 虚拟化架构对比
  - 在ESXi中，**所有**虚拟化功能都在内核实现
  - Xen内核仅实现CPU与内存虚拟化，IO虚拟化与调度管理由Domain0（主机上启动的第一个管理VM）实现
  - **KVM内核实现CPU与内存虚拟化**，**QEMU实现IO虚拟化**，通过Linux进程调度器实现VM管理

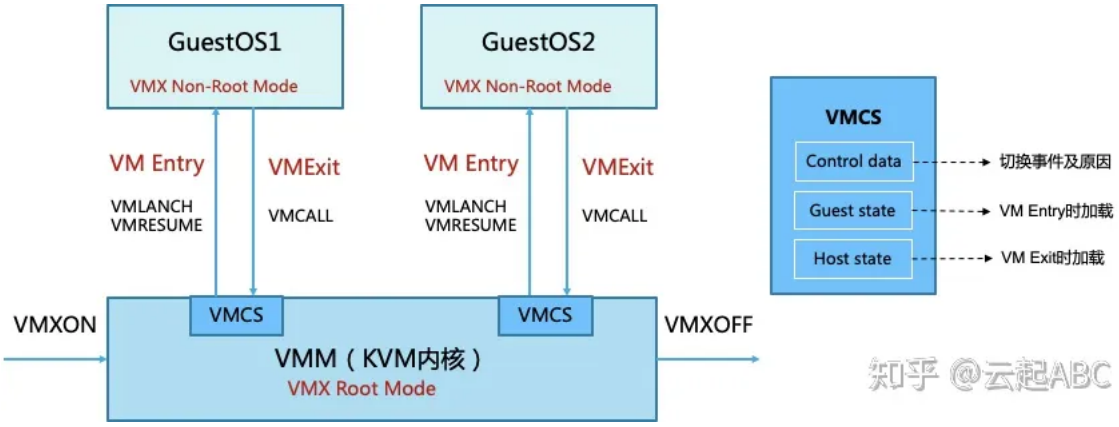
- KVM架构虚拟化具备两个核心模块
  - KVM内核模块
    - 负责CPU与内存虚拟化
  - QEMU设备模拟
    - 实现IO虚拟化与各设备模拟（磁盘、网卡、显卡、声卡等），通过IOCTL系统调用与KVM内核交互。**KVM仅支持基于硬件辅助的虚拟化**（如Intel-VT与AMD-V），在内核加载时，KVM先初始化内部数据结构，打开CPU控制寄存器CR4里面的虚拟化模式开关，**执行VMXON指令将Host OS设置为root模式**，并创建的特殊设备文件 `/dev/kvm` 等待来自用户空间的命令，然后由KVM内核与QEMU相互配合实现VM的管理。**KVM会复用部分Linux内核的能力，如进程管理调度、设备驱动，内存管理等**



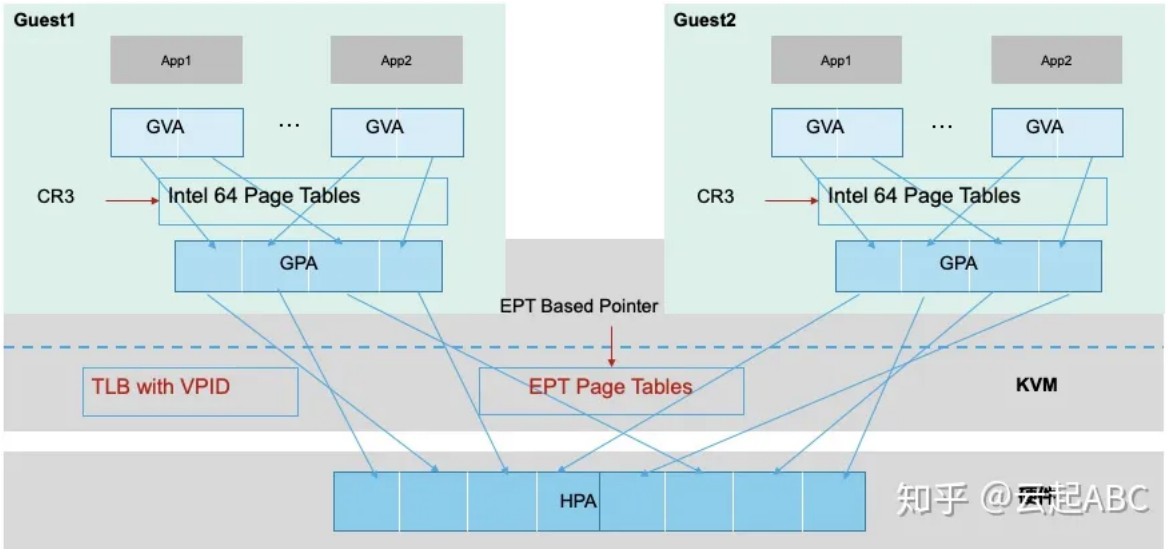
- CPU虚拟化



- 虚拟化类型对比
  - **ESXi属于全虚拟化**，VMM运行在Ring0完整模拟底层硬件；GuestOS运行在ring1，无需任何修改即可运行，执行特权指令时需要通过VMM进行异常捕获、二进制翻译BT（Binary Translation）和模拟执行
  - Xen支持**全虚拟化**（HVM Guest）与**半虚拟化**（PV Guest）；使用半虚拟化时，GuestOS运行在Ring 0需要进行修改（如安装PV Driver），通过Hypercall调用VMM处理特权指令，无需异常捕获与模拟执行
  - KVM是依赖于**硬件辅助的全虚拟化**（如Inter-VT、AMD-V），目前也通过virtio驱动实现半虚拟化以提升性能。Inter-VT引入新的执行模式，执行特权指令时两种模式可以切换
    - VMM运行在VMX Root模式
    - GuestOS运行在VMX Non-root模式

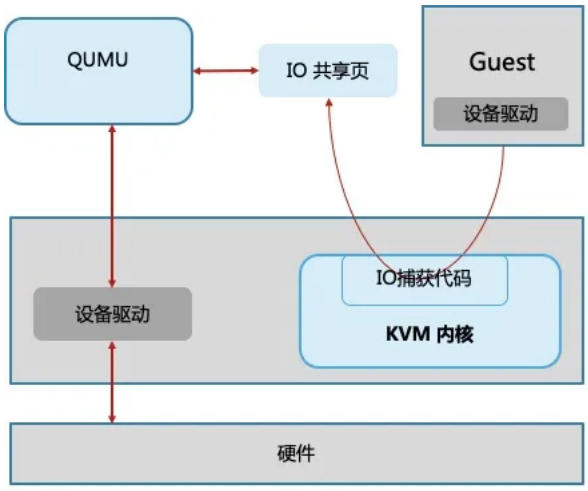


- KVM内核加载时执行VMXON指令进入VMX操作模式，VMM进入VMX Root模式，可执行VMXOFF指令退出。GuestOS执行特权或敏感指令时触发VM Exit，系统挂起GuestOS，通过VMCALL调用VMM切换到Root模式执行，VMExit开销是比较大的。VMM执行完成后，可执行VMLANCH或VMRESUME指令触发VM Entry切换到Non-root模式，系统自动加载GuestOS运行
  - 反正是有一个切换的过程

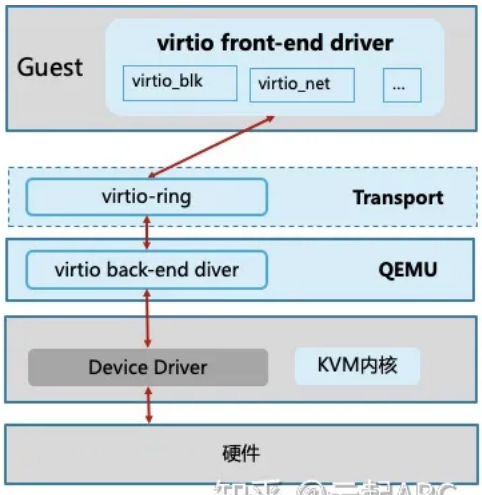


- 内存虚拟化，有点复杂，就这样吧

IO 虚拟化	类型	优点	不足
设备模拟	全虚拟化	兼容性好，无硬件依赖，GuestOS 不修改	IO 路径长，VMExit 较多，性能较差。
virtio 驱动	半虚拟化	减少 VM Exit 次数，性能较好	Guest 需要安装额外驱动，老系统无相关驱动。
设备直通 Pass-through	硬件辅助虚拟化	无需 VMM 处理，性能很好。	设备独占，PCI 设备数量有限，动态迁移受限。
设备共享 SR-IOV	硬件辅助虚拟化	性能很好，设备可共享。	动态迁移有限制。
DPDK/SPDK	硬件辅助虚拟化	用户态处理 IO，无需内核参与，性能非常好。	需要对系统进行改造



QEMU IO设备模拟架构



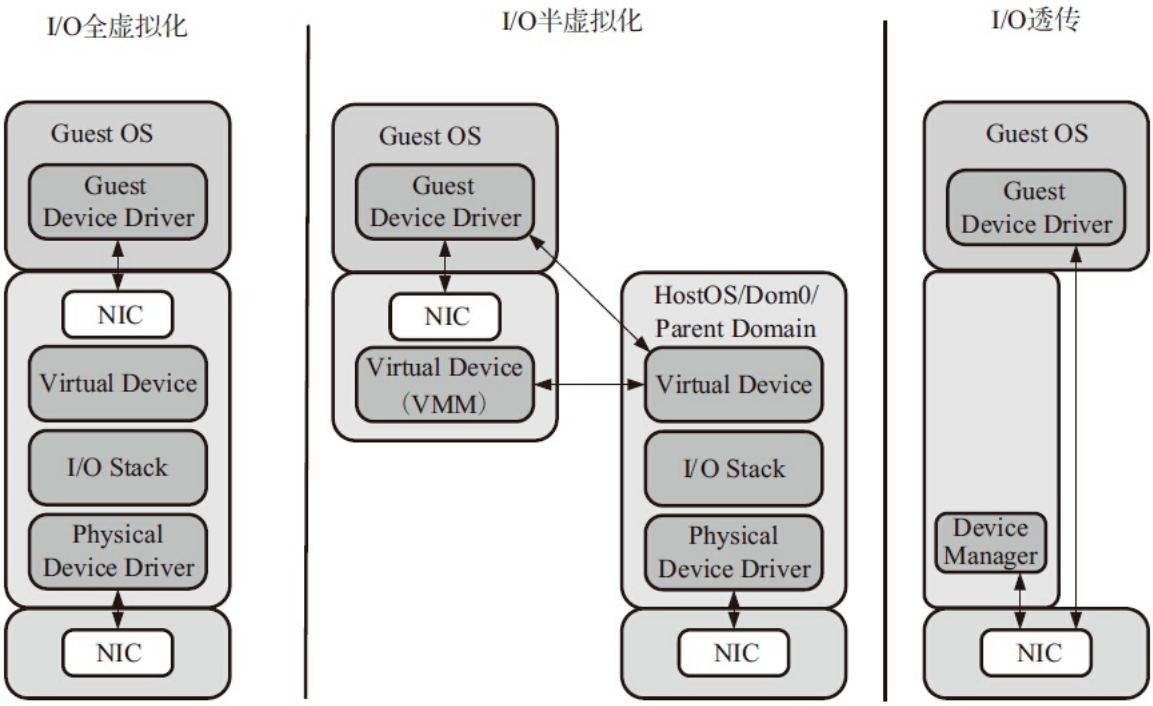
virtio半虚拟化架构

- **KVM中通过QEMU来模拟网卡或磁盘设备。**Guest发起IO操作时被KVM的内核捕获，处理后发送到IO共享页并通知QEMU；QEMU获取IO交给硬件模拟代码模拟IO操作，并发送IO请求到底层硬件处理，处理结果返回到IO共享页；然后通知IO捕获代码，读取结果并返回到Guest中

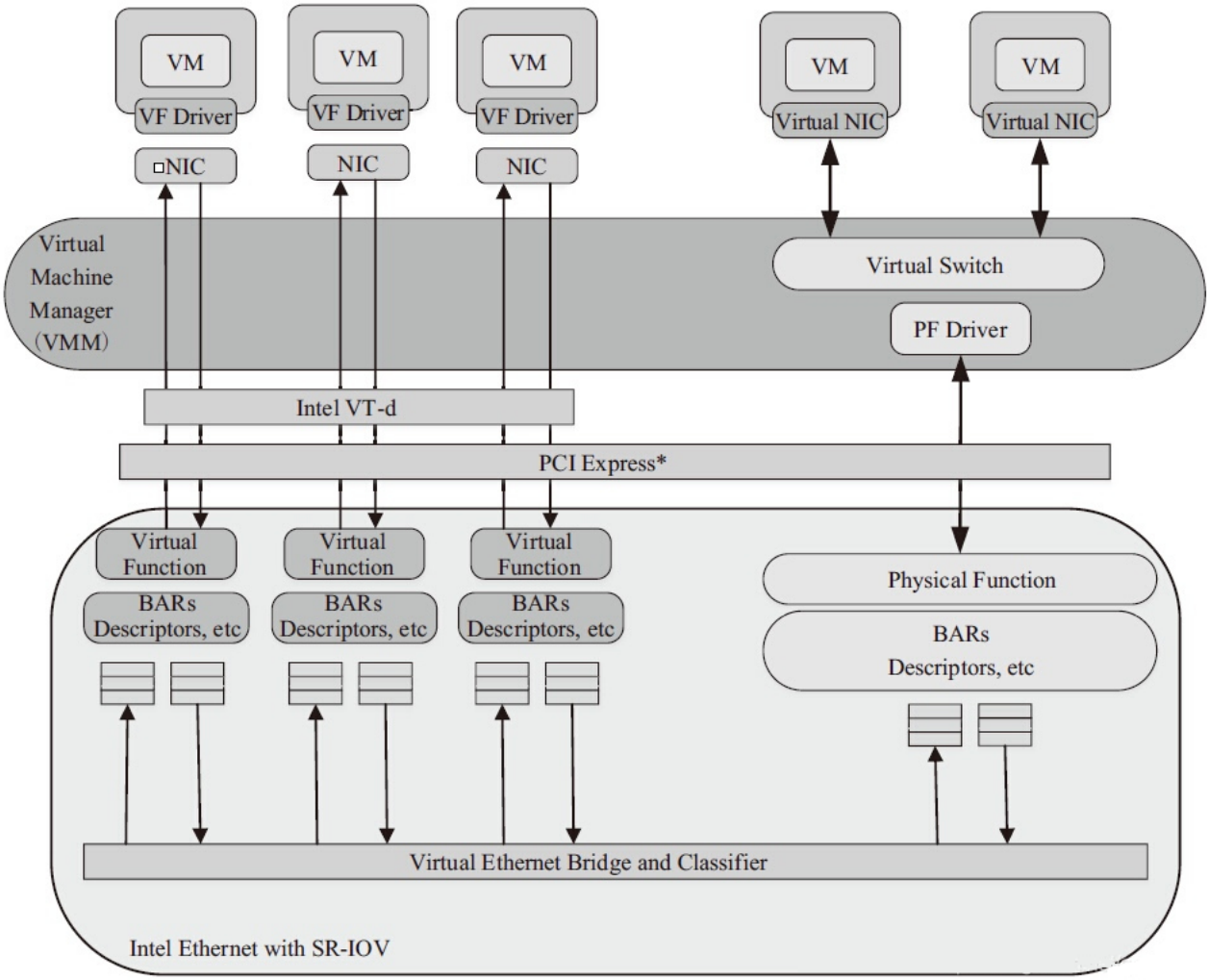
- IO虚拟化部分和之前理解的不太一样，**看一下IO虚拟化的发展过程**
  1. **在Guest中部署virtio前端驱动**，如virtio-net、virtio-blk、virtio_pci、virtio_balloon、virtio_scsi、virtio_console等
     - **然后在QEMU中部署对应的后端驱动（用户态）**，前后端之间定义了虚拟环形缓冲区队列virtio-ring，用于保存IO请求与执行信息
  2. virtio中后端驱动由用户空间的QEMU提供，但网络协议栈处于内核中，如果通过内核空间来处理网络IO，可以减少了网络IO处理过程中的多次上下文切换，从而提高网络吞吐量与性能。所以，**新的内核中提供vhost-net驱动，使前端网络驱动virtio-net的后端处理任务从用户态的QEMU改到Host内核空间执行**
     - 把**I/O处理模块**放在**QEMU进程之外**去实现的方案称为**vhost**
  3. 大规模云计算环境中会使用OVS（Open vSwitch）或SDN方案，而进程运行在用户态，如果继续用使用内核态的vhost-net，依然存在大量用户态与内核态的切换，所以引入了**vhost-user**（**内核态vhost功能在用户态实现**）。vhost-user定义了Master（QEMU进程）和slave（OVS进程）作为通信两端，Master与slave之间控制面通过共享的虚拟队列virtqueure交换控制逻辑，数据面通过共享内存交换信息。结合vhost-user、vSwitch与DPDK可以在用户态完成网络数据包交换处理，从而大幅提升了网络虚拟化性能
  4. 将设备直通给guest，这其实是我用的最多的一种，解释如下，确实无法share，只能单个guest独占
     - 基于硬件辅助虚拟化技术，KVM支持将Host的PCI/PCI-E物理设备（如网卡、磁盘、USB、显卡、GPU等）直接分配给Guest使用。**Guest的对该设备的IO操作与物理设备一样，不经过QEMU/KVM处理。**直通设备不能共享给多个Guest使用，且不能随Guest进行动态迁移，需要通过热插拔或**libvirt工具**来解决
  5. **设备共享SR-IOV**
     - 为了实现多个Guest可以共享同一个物理设备，PCI-SIG发布了SR-IOV（Single Root-IO Virtualization）标准，让一个物理设备支持多个虚拟功能接口，可以独立分配给不同的Guest使用
       - **原来如此**
  6. 下一步，dpdk/spdk就出来了，**其实本质也是用户态的vhost**

KVM IO虚拟化

- ref
  - https://github.com/0voice/Introduce_to_virtualization/blob/main/virtualization_type/io_virtualization/IO虚拟化.md



- I/O虚拟化包括管理虚拟设备和共享的物理硬件之间I/O请求的路由选择。目前，实现I/O虚拟化有三种方式
  - **I/O全虚拟化**
    - 宿主机截获客户机对I/O设备的访问请求，然后通过软件模拟真实的硬件。这种方式对客户机而言非常透明，无需考虑底层硬件的情况，**不需要修改操作系统**
      - 不需要修改操作系统，这个是最明显的特征，例如qemu模拟的ahci的sda等，驱动依然是ahci驱动，所以这就是**I/O全虚拟化**
  - **I/O半虚拟化**
    - 需要用到virtIO的虚拟化方式，很大的一个特点就是需要修改操作系统，设备的驱动为**virtxxx系列驱动**
  - **I/O透传**
    - 无需多言



- SR-IOV

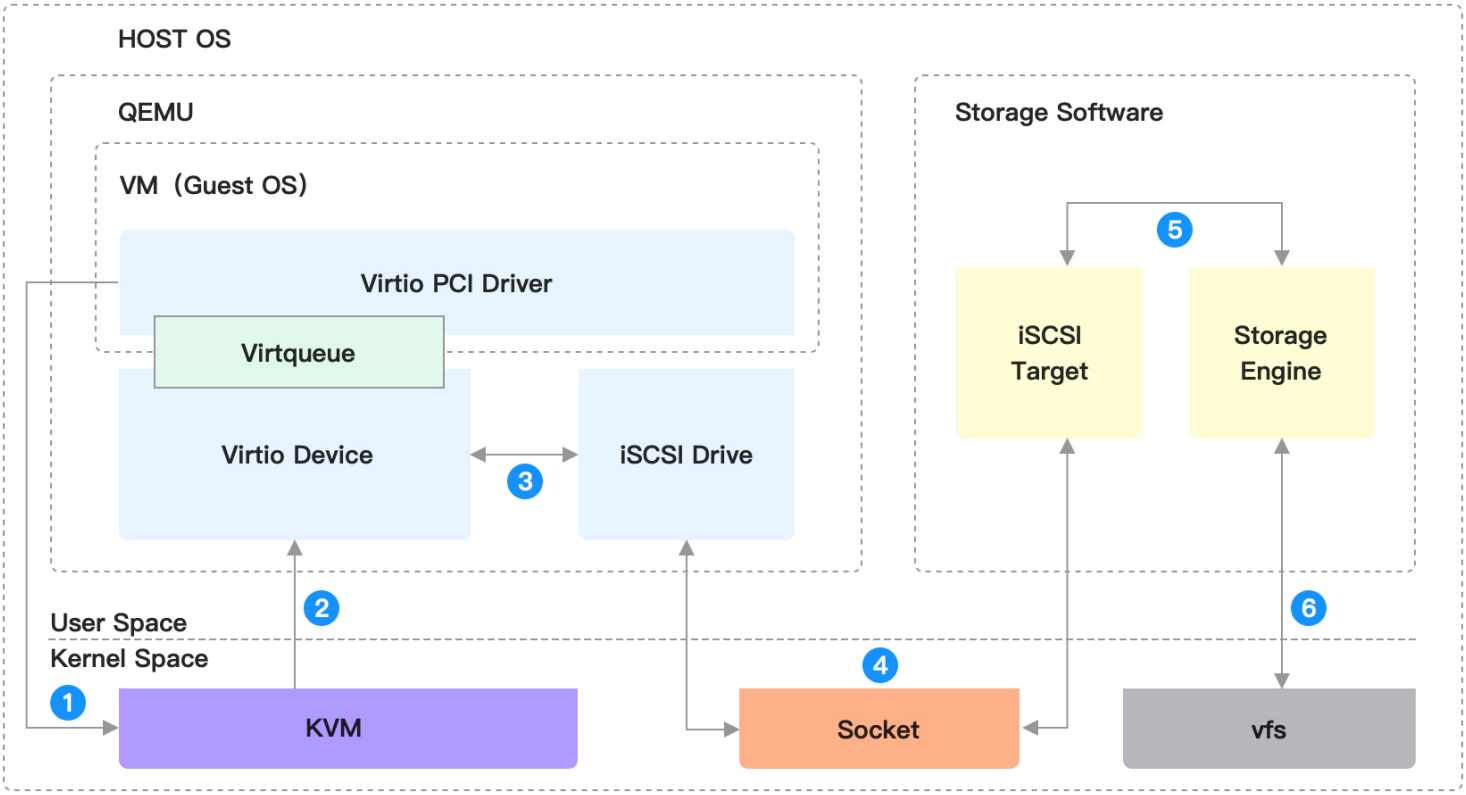
- virtio-net的后端驱动经历过
  - **从virtio-net后端**
    - virtio-net后端驱动的最基本要素是**虚拟队列机制、消息通知机制和中断机制**
      - 虚拟队列机制连接着客户机和宿主机的数据交互
      - 消息通知机制主要用于从客户机到宿主机的消息通知
      - **中断机制主要用于从宿主机到客户机的中断请求和处理**
  - **到内核态vhost-net**
    - vhost-net技术对virtio-net进行了优化，在内核中加入了vhost-net.ko模块，使得对网络数据可以在内核态得到处理。vhost-net通过卸载virtio-net在报文收发处理上的工作，使Qemu从virtio-net的虚拟队列工作中解放出来，减少上下文切换和数据包拷贝，进而提高报文收发的性能
    - 除此以外，宿主机上的vhost-net模块还需要承担报文到达和发送消息通知及中断的工作
  - **再到用户态vhost-user的演进过程**
    - 有些网络处理本身就位于用户态，**所以vhost又重新出现在了用户态**

SPDK Vhost-user 如何帮助超融合架构实现 I/O 存储性能提升

- ref
  - https://www.smartx.com/blog/2022/03/spdk-vhost-user/

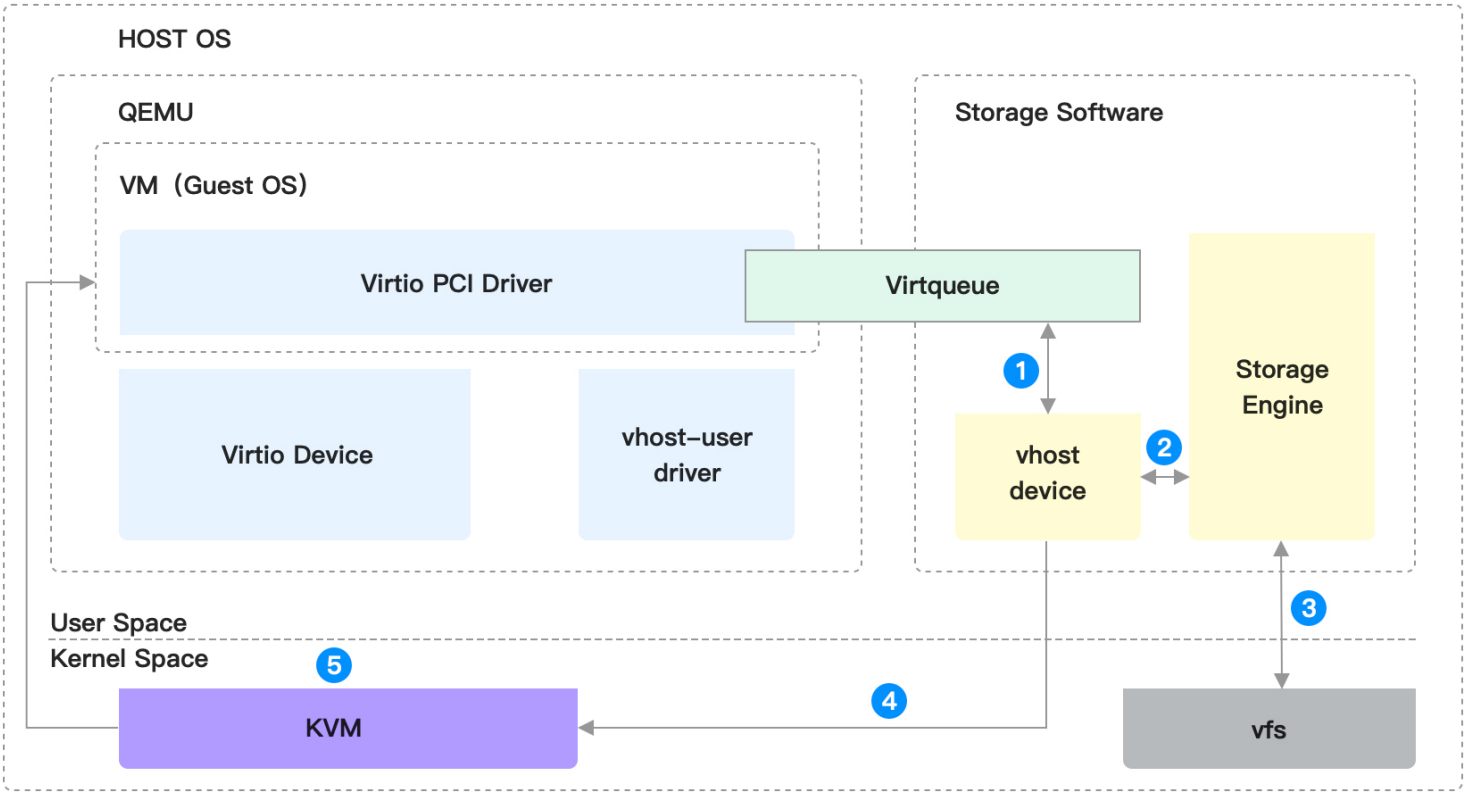
- 当前主流的I/O设备虚拟化方案
  - QEMU纯软件模拟，利用软件模拟I/O设备提供给虚拟机使用
    - 驱动什么的都无需修改，但是性能是最差的
  - **Virtio半虚拟方案**，**规范了前后端**模型，在虚拟机（Guest OS）中使用frontend驱动（Virtio Drive），在Hypervisor（QEMU）中使用backend设备（Virtio Device）提供I/O能力，通过减少**内存复制次数**和**VM陷入次数**，提升I/O性能，这种方案需要**安装Virtio驱动**
    - **但是virtio已经完完全全并入内核了**

- 在QEMU中，Virtio设备是为Guest操作系统模拟的PCI/PCIe设备，遵循PCI规范，具有配置空间、中断配置等功能。Virtio注册了PCI厂商ID（0x1AF4）和设备ID，不同的设备ID代表不同的设备类型，例如面向存储的virtio-blk（0x1001）和virtio-scsi设备ID（0x1004）



1. Guest发起I/O操作，Guest内核Virtio驱动写PCI配置空间，触发VM EXIT（这里由于**优化**的原因，不一定**每一次IO都会触发VM EXIT**，否则**半虚拟化Virtio的优势在哪里呢**），返回到Host KVM中（通知 KVM）
2. QEMU的vCPU线程从KVM内核态回到QEMU，让QEMU Device来处理Virtio Vring请求
3. QEMU通过iSCSI Drive发起存储连接（iscsi over tcp to localhost）
4. 通过Socket将请求连接到存储进程提供的iSCSI Target
5. 存储引擎接收请求并进行I/O处理
6. 存储引擎发起对本地存储介质的I/O
7. I/O操作结束，通过上述逆过程返回至Virtio后端Device，QEMU会向模拟的PCI发送中断通知，从而Guest基于该中断完成整个I/O流程

- Vhost加速
  - 如前所述，Virtio后端Device用于具体处理Guest的请求，负责I/O的响应，**把I/O处理模块放在QEMU进程之外去实现的方案称为vhost**。由于我们需要实现的优化目标是在两个用户态进程之间（超融合架构），所以采用vhost-user方案进行存储加速实现（vhost-kernel方案主要是将I/O负载卸载到内核完成，所在不在本文讨论）
  - Vhost-user的数据平面处理主要分为Master和Slave两个部分
    - 其中Master为virtqueue的供应方，一般由QEMU作为Master
    - 存储软件作为Slave，负责消费virtqueue中的I/O请求
  - Vhost-user的优势
    - 消除Guest内核更新PCI配置空间，QEMU捕获Guest的VMM陷入所带来的CPU上下文开销（后端处理线程采用轮询所有virtqueue）
    - 用户态进程间内存共享，优化数据复制效率



1. 当Guest发起I/O操作后，存储软件通过Polling机制感知新的请求动作，从virtqueue获取数据
2. 存储引擎接收请求并进行I/O处理
3. 存储引擎发起对本地存储介质的I/O
4. I/O操作完成后，由vhost device发送irqfd（eventfd）通知到KVM
5. KVM注入中断通知Guest OS完成I/O

那我用文件模拟的sda/nvme是什么虚拟化呢 —— 用了很久也没有思考过这个问题

- `DISK_PARAMETER := -drive id=disk,file=./disk/disk.img,format=raw,if=none -device ahci,id=ahci -device ide-hd,drive=disk,bus=ahci.0`
  - 全虚拟化，guest os无需改变
  - 但这种方式的缺点就是性能损耗较大，因为CPU必须耗费大量计算能力才能以软件方式模拟硬件操作。为提高性能，可以Qemu还提供有半虚拟化硬件，这时客户机操作系统会感知到Qemu环境的存在，并直接和虚拟机管理器配合工作
- `PARAMETER += -drive if=none,id=drive0,cache=none,aio=native,format=raw,file=./disk/disk.img -device virtio-blk-pci,drive=drive0,scsi=off`
  - 半虚拟化，使用virtio驱动
  - Qemu的半虚拟化硬件采用了virtio标准，并以virtio半虚拟化硬件形式实现，具体包括半虚拟化硬盘控制器，半虚拟化网卡，半虚拟化串口，半虚拟化SCSI控制器等
- 但是其实后端都是 `disk.img` ，即**宿主机的文件**
- 那这里就又引出了另一个好文
  - ref
    - https://blog.gmem.cc/kvm-qemu-study-note —— **qemu好文，有qemu参数，可作查询**

KVM和QEMU学习笔记 —— 当字典用的好文

- 半/全虚拟化的区别如下
  - 在全虚拟化状态下，**Guest OS不知道自己是虚拟机**，于是像发送普通的IO一样发送数据，**被Hypervisor拦截，转发给真正的硬件**
  - 在半虚拟化状态下，**Guest OS知道自己是虚拟机，因为驱动走的就不一样**（需安装半虚拟化驱动），所以数据直接发送给**半虚拟化设备**，经过特殊处理，发送给真正的硬件
- **virtio-blk**
  - 使用virto_blk驱动的硬盘，在客户机里对应的设备文件是 `/dev/vda` ，而IDE硬盘是 `/dev/hda` 、基于SATA的硬盘则显示为 `/dev/sda`
    - `PARAMETER += -drive if=none,id=drive0,cache=none,aio=native,format=raw,file=./disk/disk.img -device virtio-blk-pci,drive=drive0,scsi=o`
      - **linux config需要开启VIRTIO_BLK，能够看到vda的块设备**
        - vda的驱动与scsi一毛钱关系都没有的说哦，并不是 `sd.c` ，完全是不同的另一套的说

        - **所以virto的虚拟设备并非需要直接与后端真实设备关联啊，至少块设备是这样的**
- **virtio-net**
    - 要基于半虚拟化来访问网络，可以使用选项 `-device virtio-net-pci,netdev=network0`
        - **你应当总是考虑启用半虚拟化网卡**，因为性能会有很大的提升
- 最佳实践
    - 使用半虚拟化驱动virtio
        - 性能好：延迟低、吞吐量高
        - 纯虚拟设备的劣势：需要高吞吐能力的设备在硬件方面会有特殊的实现，这些纯虚拟设备是没法利用的
        - 网络、块设备、内存，都可以使用virtio
    - 虚拟机最好直接使用块设备做存储 —— 比如说 `disk.img` 就是文件系统中的
        - 性能好、无需管理宿主机的文件系统、无需管理稀疏文件
        - I/O 缓存以4K为边界
        - 宿主机最好使用ext3文件系统，ext4的barrier会影响性能
        - 选择正确的缓存策略，缓存模式推荐none，I/O调度器推荐Deadline I/O scheduler
    - CPU配置
        - 每个客户机相当于一个进程，而每个客户机的虚拟CPU相当于一个线程。因此超配CPU是可行的
        - CPU超配可能带来额外的上下文切换，影响性能
        - Pin CPU：可以将虚拟CPU Pin到一个物理CPU，或者一组共享缓存的物理CPU，便于缓存共享。缺点是Pin导致其它空闲CPU可能得不到利用
    - 内存配置
        - 使用内核特性KSM（Kernel Same Page Merging），KSM通过扫描将相同的内存区域设置为共享，并且Copy-on-write。共享内存节约可以内存空间，但是内存扫描同时影响性能
        - 尽量避免使用swap，可以设置/proc/sys/vm/swappiness=0
    - 网络配置
        - 使用tap类型的网络后端
        - 启用PCI passthough可以提高性能，但是影响迁移
    - TODO

IO虚拟化的继续理解 —— 我其实现在最不理解的就是这个为啥就能称之为IO虚拟化

tips

首先第一个关键的tips，对于Host软件来说，什么是IO设备

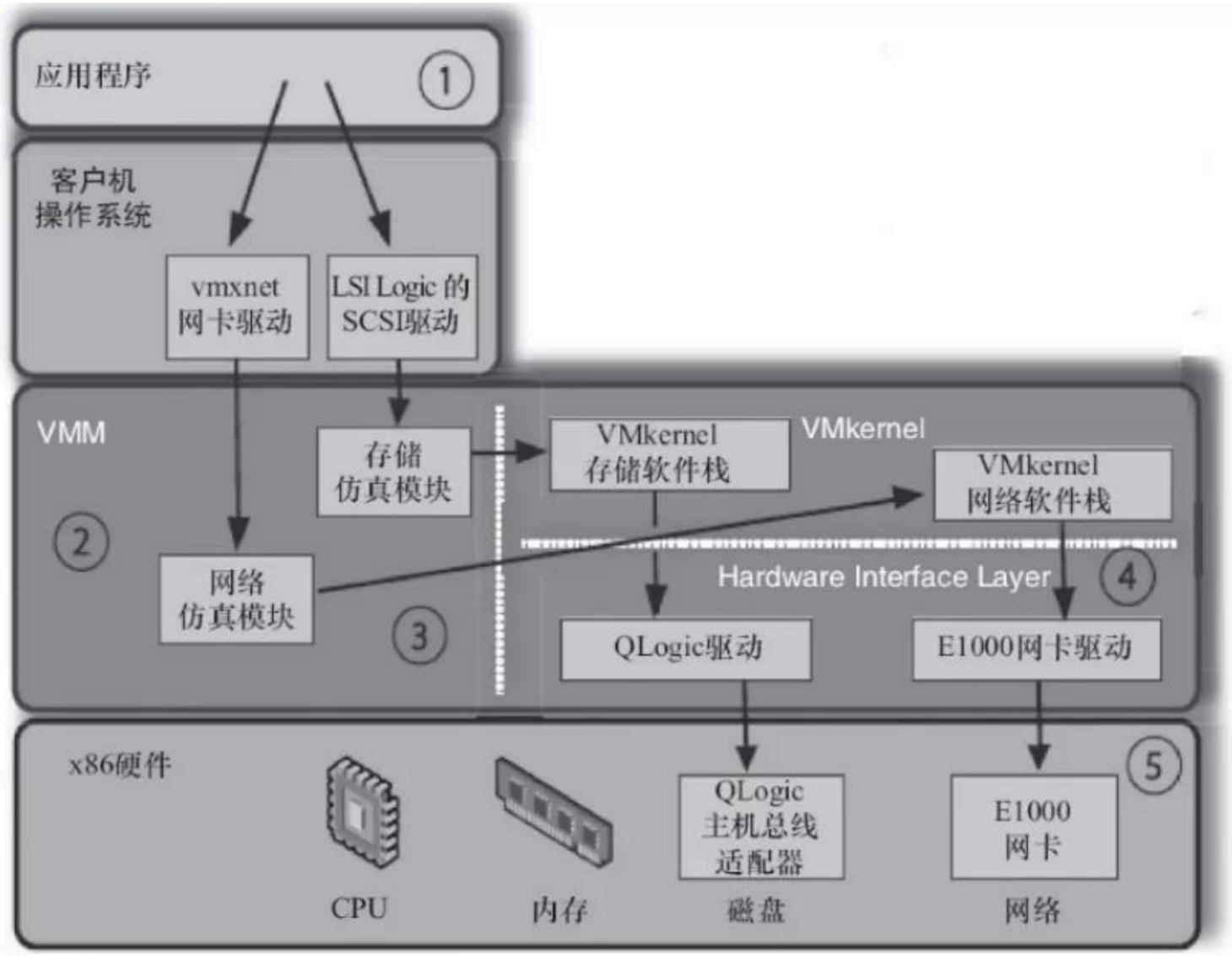
- 关键要素有**3**
    - **寄存器/DMA/中断**
- 而**IO虚拟化**的过程就是模拟上述**3要素**，并**截获**Guest OS中对寄存器的访问，然后通过软件的方式来模拟硬件

关于IO指令 —— 要认识到这是一条敏感指令

- 仅考虑最熟悉的**MMIO**，IO在Host的最后一步就是我们所熟知的**敲门铃**，在各种类型的IO驱动中，最后会敲门铃寄存器，实际上是**写PCIe BAR空间中的一个地址**
    - 该地址对应的写操作是一条**敏感指令**

典型的IO虚拟化分类

IO全虚拟化



- 可以简单描述一下我最开始的理解
  - 最简单的那种**虚拟磁盘**的情况，以宿主机的**文件**（后来证实这个并不是区分IO虚拟化分类的关键）作为**虚拟磁盘**的情况。我最开始理解**Guest OS**就是一个用户态的应用程序（这个理解也是片面的，总以为这个世界跑在裸金属上的**只有Linux Kernel**，而且看起来ESXi**最开始**可能都**没有用户态**的说法），而Guest OS作为一个用户态的应用程序，它的磁盘IO实际上就是宿主机的文件读写
    - 这句话的理解怎么说呢，也对也不对。对是因为实际情况确实是这样，不对的原因是忽视了太多的细节（**理解IO虚拟化的关键细节**）
    - 我最开始还以为这个真的就像是**用户态应用程序通过Read/Write系统调用操作文件系统的文件**一样简单，确实如果这样来思考的话，这和虚拟化有什么**蛋关系**呢
    - 我开始浅薄的理解
      - Guest OS（用户态应用程序）最后产生一个读写命令，这个读写命令被**某模块**识别并转换成对宿主机文件的读写
        - **两个错误**
          - 没有考虑最后的读写命令是一个敏感指令（**甚至于一开始都没有敏感指令的概念**）
          - 这个**某模块**是什么，**太想当然了**
- 再来描述一下如果**虚拟化的概念**被引入之后应该作何理解呢
  - 首先，在不修改Guest OS源码的情况下，IO的最后一步都是对PCIe BAR空间的一个写操作（考虑当前最常见的情况），这是一条敏感指令，**敏感指令的执行一定要被VMM捕获**
  - 其次，这个**某模块**是什么东西呢。以ESXi为例，这部分位于VMM内部。当然IO全虚拟化不仅仅ESXi会实现，其它（KVM+QEMU）都会实现，这些**某模块**都是有名字的
- 再按照现在的理解描述一下 —— 其本质是VMM需要截获虚拟机操作系统对外部设备的访问请求，通过软件的方式模拟出真实的物理设备的效果 —— **时刻更新**

    - IO的最后一锤子是对PCIe BAR空间寄存器的**写操作**，妥妥的**敏感指令。**
    - GuestOS执行特权或敏感指令时触发**VM Exit**（以KVM为例），系统挂起GuestOS，通过VMCALL调用VMM切换到Root模式执行，**VM Exit开销是比较大的**
    - VMM执行完成后，可执行VMLANCH或VMRESUME指令触发**VM Entry**切换到Non-root模式，**系统自动加载GuestOS运行**
- ESXi是典型的IO全虚拟化产品
    - Guest OS对这些虚拟设备的**每一次IO操作**都会陷入VMM中，由VMM对IO指令进行解析并映射到实际的物理设备，然后直接控制硬件完成
- **GuestOS**与**VMM**之间的**切换开销巨大**
    - 切换的是一个**虚拟机上下文**，尽管虚拟机在HOST上同样是表现为一个进程，**但是其关联的资源更多**

IO半虚拟化

- 看起来**IO全虚拟化**最大的开销就在于**VM Exit**，对于**IO密集型任务**几乎是无法接受的，其实**对于什么都无法接受**
- 说说我查阅资料之后的理解
    - 就**上文所述**，IO全虚拟化每一次IO都意味着一次**VM EXIT**（**性能杀手**），但是看起来是无解的，因为对于**原生的IO设备驱动**，最后**发起IO**的一定是**敏感指令**
    - 那怎么办呢，那就干掉问题的本质，**即原生的IO设备驱动**，既然原生的IO设备驱动可能会大量的访问寄存器或每一次IO都是敏感指令，**那就把驱动改了**
    - 说起来容易做起来难呀，**这意味着所有的驱动都得改一发**
        - 但是实际上这个想法不对，既然都要彻底推倒重来了，那为什么不彻底一点呢，最后统一到了**virtio中**
        - virtio是**全新**的驱动架构，意味着最后发起IO的行为是可控的，可以不是**敏感指令**
    - 为什么基本的说法都是Virtio可以显著的减少**VM EXIT**的次数呢，因为没有太仔细的阅读源码，网上的解释又**千奇百怪**（**都是理解不够透彻的**），所以这里就简单描述一下
        1. 相对于真实驱动，敏感指令可以更少，**这个规范是可以定的**，所以**virtio的寄存器交互**相比于**真实驱动**而言比较少
        2. 最后发起IO的行为是入队，**不是敏感指令。**但是入队之后的**通知**是**敏感指令**，入队之后不立刻通知（**batch**）也能够大幅度降低敏感指令的次数

IO透传

- 关于IO透传倒是充分理解了一下**硬件虚拟化技术VT-d**，之前没有理解它的作用，今日**多理解了一丢丢**
- 即使是IO透传，性能很高。但是在GuestOS中，向硬件寄存器写的可是GPA(Guest Physical Address)，对于真实的硬件，这个地址是需要转换的，**这其实是IOMMU的功能**
- 其实就是这个意思哈，需要由硬件来实现**中断重映射/IOMMU**等功能

所以最后这些实现为什么能够称之为IO虚拟化呢 —— 即什么是IO虚拟化呢

- 为了在虚拟机中能够执行IO操作，需要一定的**机制**来实现
    - **该机制就称为IO虚拟化**