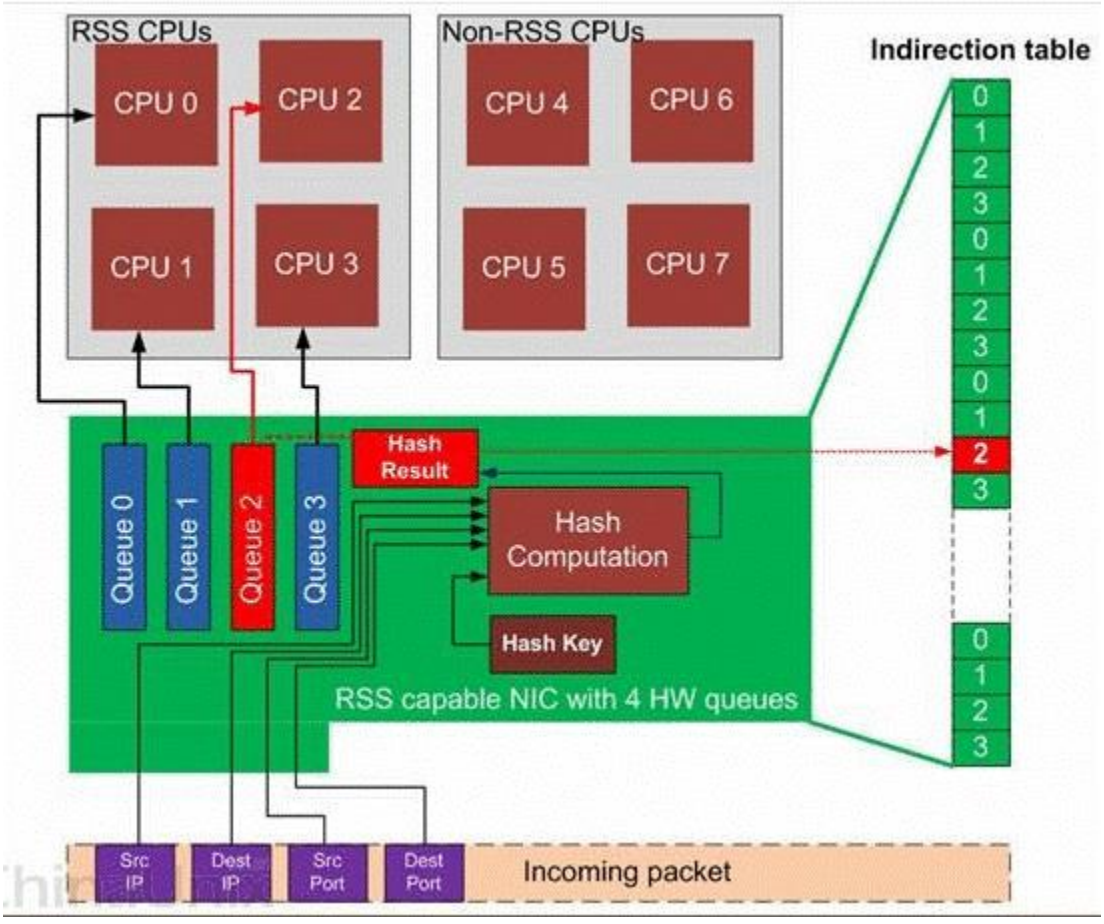
虚拟化难于理解的概念

多队列网卡

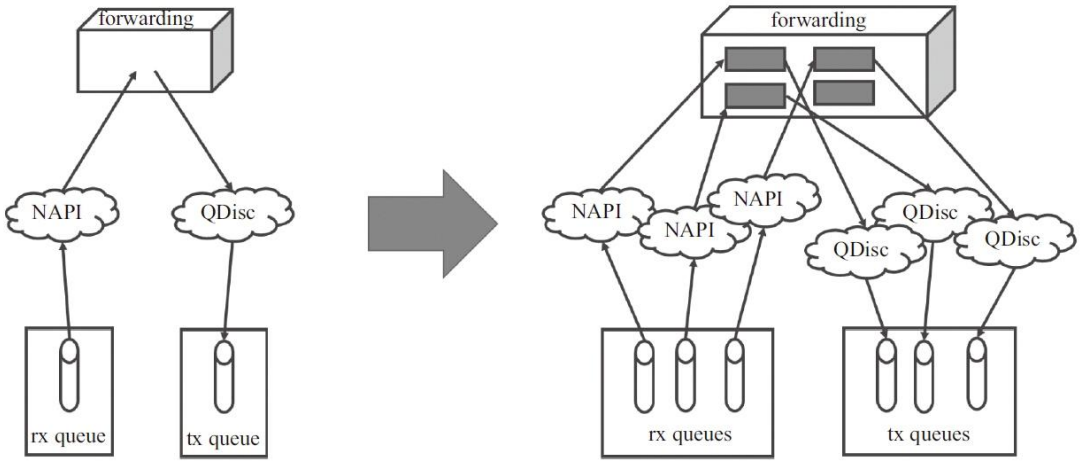
多队列网卡硬件实现

有四个硬件队列(Queue0, Queue1, Queue2, Queue3)，当收到报文时，通过 hash 包头的(sip, sport, dip, dport)四元组，将一条流总是收到相同队列，同时触发与该队列绑定的中断。




```
king@ubuntu:~$ ls /sys/class/net/eth0/queues/
rx-0  rx-2  rx-4  rx-6  tx-0  tx-2  tx-4  tx-6
rx-1  rx-3  rx-5  rx-7  tx-1  tx-3  tx-5  tx-7
```

内核对多队列网卡的支持



Linux 内核中，RPS（Receive Packet Steering）在接收端提供了这样的机制。RPS 主要是把软中断的负载均衡到 CPU 的各个 core 上，网卡驱动对每个流生成一个 hash 标识，这个 hash 值可以通过四元组（源 IP 地址 SIP，源四层端口 SPORT，目的 IP 地址 DIP，目的四层端口 DPORT）来计算，然后由中断处理的地方根据这个 hash 标识分配到相应的 core 上去，这样就可以比较充分地发挥多核的能力了。

NAPI 是 linux 上采用一种提高网络处理效率的技术，其核心概念就是不采用中断的方式读取数据，取而代之是采用中断唤醒数据接收的服务程序，然后 POLL 的方式轮询数据。

NAPI 的优点：

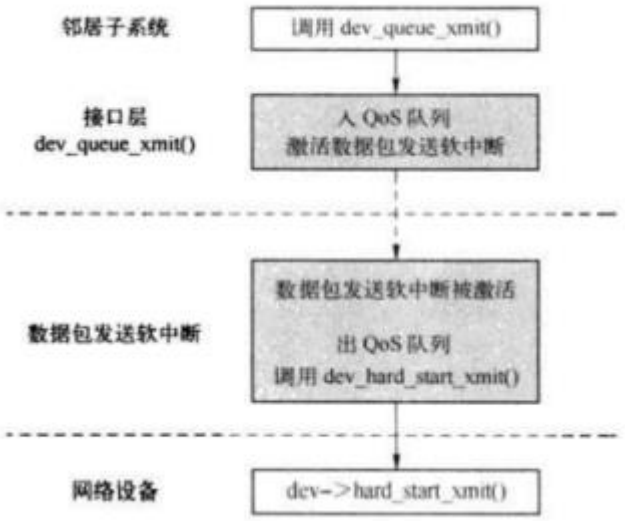
1. 中断缓和，在日常使用中，网卡产生的高达几 k/s，每次中断都需要系统来处理，是一个很大的压力，而 NAPI 使用轮询是禁止了网卡接收中断，减小处理中断的压力。
2. 数据包节流，NAPI 之前的 NIC 总在接收数据包之后产生一个 IRQ，接着在中断服务函数将 skb 加入本地 softnet，然后触发本地 NET_RX_SOFTIRQ 软中断后续处理。如果包速过高，因为 IRQ 的优先级高于 SoftIRQ，导致系统的大部分资源都在响应中断，但 softnet 的队列大小有限，接收到的超额数据包也只能丢掉，所以这时这个模型是在用宝贵的系统资源做无用功。而 NAPI 则在这样的情况下，直接把包丢掉，不会继续将需要丢掉的数据包扔给内核去处理，这样，网卡将需要丢掉的数据包尽可能的早丢弃掉，内核将不可见需要丢掉的数据包，这样也减少了内核的压力。

NAPI 的缺点：

1. 对于上层的应用程序而言，系统不能在每个数据包接收到的时候都可以及时地去处理它，而且随着传输速度增加，累计的数据包将会耗费大量的内存。
2. 另外一个问题是对于大的数据包处理比较困难，原因是大的数据包传送到网络层上的时候耗费的时间比短数据包长很多（即使是采用 DMA 方式）。所以，NAPI 技术适用于对高速率的短长度数据包的处理。

QDisc 是 queueing discipline 的简写，它是理解流量(traffic control)的基础。无

论何时，内核如果需要通过某个网络接口发送数据包，需要按照这个接口配置的 qdisc(排队规则)把数据包加入队列。然后，内核会尽可能多的从 qdisc 里面取出数据包，把它们交给网络适配器模块。最简单的 QDisc 是 pfifo，他不对进入的数据包做任何处理，数据包采用先入先出的方式。



多队列网卡的结构

Linux 内核的网卡结构体是由 net_device 表示，数据包是由 sk_buff 表示的

```
struct net_device {
    char            name[IFNAMSIZ];
    struct hlist_node   name_hlist;
    char            *ifalias;
    /*
     *  I/O specific fields
     *  FIXME: Merge these and struct ifmap into one
     */
    unsigned long       mem_end;
    unsigned long       mem_start;
    unsigned long       base_addr;
    int         irq;

    atomic_t        carrier_changes;

    /*
     *  Some hardware also needs these fields (state,dev_list,
     *  napi_list,unreg_list,close_list) but they are not
     *  part of the usual set specified in Space.c.
     */

    unsigned long       state;
```

接收端

```
int netif_rx(struct sk_buff *skb)
{
    trace_netif_rx_entry(skb);

    return netif_rx_internal(skb);
}
static int netif_rx_internal(struct sk_buff *skb)
{
    int ret;

    net_timestamp_check(netdev_tstamp_prequeue, skb);

    trace_netif_rx(skb);
#ifdef CONFIG_RPS
    if (static_key_false(&rps_needed)) {
        struct rps_dev_flow voidflow, *rflow = &voidflow;
        int cpu;

        preempt_disable();
        rcu_read_lock();

        cpu = get_rps_cpu(skb->dev, skb, &rflow);
        if (cpu < 0)
            cpu = smp_processor_id();

        ret = enqueue_to_backlog(skb, cpu, &rflow->last_qtail);

        rcu_read_unlock();
        preempt_enable();
    } else
#endif
    {
        unsigned int qtail;
        ret = enqueue_to_backlog(skb, get_cpu(), &qtail);
        put_cpu();
    }
```

发送端

```
int dev_queue_xmit(struct sk_buff *skb)
{
    return __dev_queue_xmit(skb, NULL);
}
```

```
static int __dev_queue_xmit(struct sk_buff *skb, void *accel_priv)
{
    struct net_device *dev = skb->dev;
    struct netdev_queue *txq;
    struct Qdisc *q;
    int rc = -ENOMEM;

    skb_reset_mac_header(skb);

    if (unlikely(skb_shinfo(skb)->tx_flags & SKBTX_SCHED_TSTAMP))
        __skb_tstamp_tx(skb, NULL, skb->sk, SCM_TSTAMP_SCHED);

    /* Disable soft irqs for various locks below. Also
     * stops preemption for RCU.
     */
    rcu_read_lock_bh();

    skb_update_prio(skb);

    /* If device/qdisc don't need skb->dst, release it right now whi
     * its hot in this cpu cache.
     */
    if (dev->priv_flags & IFF_XMIT_DST_RELEASE)
        skb_dst_drop(skb);

#ifdef CONFIG_NET_SWITCHDEV
    /* Don't forward if offload device already forwarded */
    if (skb->offload_fwd_mark &&
        skb->offload_fwd_mark == dev->offload_fwd_mark) {
        consume_skb(skb);
        rc = NET_XMIT_SUCCESS;
        goto ↓out;
    }
#endif

    txq = netdev_pick_tx(dev, skb, accel_priv);
    q = rcu_dereference_bh(txq->qdisc);

#ifdef CONFIG_NET_CLS_ACT
    skb->tc_verd = SET_TC_AT(skb->tc_verd, AT_EGRESS);
#endif
    trace_net_dev_queue(skb);
    if (q->enqueue) {
        rc = __dev_xmit_skb(skb, q, dev, txq);
        goto ↓out;
    }
```

```
struct netdev_queue *netdev_pick_tx(struct net_device *dev,
                                    struct sk_buff *skb,
                                    void *accel_priv)
{
    int queue_index = 0;

#ifdef CONFIG_XPS
    u32 sender_cpu = skb->sender_cpu - 1;

    if (sender_cpu >= (u32)NR_CPUS)
        skb->sender_cpu = raw_smp_processor_id() + 1;
#endif

    if (dev->real_num_tx_queues != 1) {
        const struct net_device_ops *ops = dev->netdev_ops;
        if (ops->ndo_select_queue)
            queue_index = ops->ndo_select_queue(dev, skb, accel_priv,
                                                __netdev_pick_tx);
        else
            queue_index = __netdev_pick_tx(dev, skb);

        if (!accel_priv)
            queue_index = netdev_cap_txqueue(dev, queue_index);
    }

    skb_set_queue_mapping(skb, queue_index);
    return netdev_get_tx_queue(dev, queue_index);
} ? end netdev_pick_tx ?

static u16 __netdev_pick_tx(struct net_device *dev, struct sk_buff *skb
{
    struct sock *sk = skb->sk;
    int queue_index = sk_tx_queue_get(sk);

    if (queue_index < 0 || skb->ooo_okay ||
        queue_index >= dev->real_num_tx_queues) {
        int new_index = get_xps_queue(dev, skb);
        if (new_index < 0)
            new_index = skb_tx_hash(dev, skb);

        if (queue_index != new_index && sk &&
            sk_fullsock(sk) &&
            rcu_access_pointer(sk->sk_dst_cache))
            sk_tx_queue_set(sk, new_index);

        queue_index = new_index;
    }

    return queue_index;
} ? end __netdev_pick_tx ?
```

DPDK 与多队列网卡

Dpdk 网口配置，

Int rte_eth_dev_configure(uint16_t port_id, uint16_t nb_rx_q, uint16_t nb_tx_q, const struct rte_eth_conf *dev_conf);

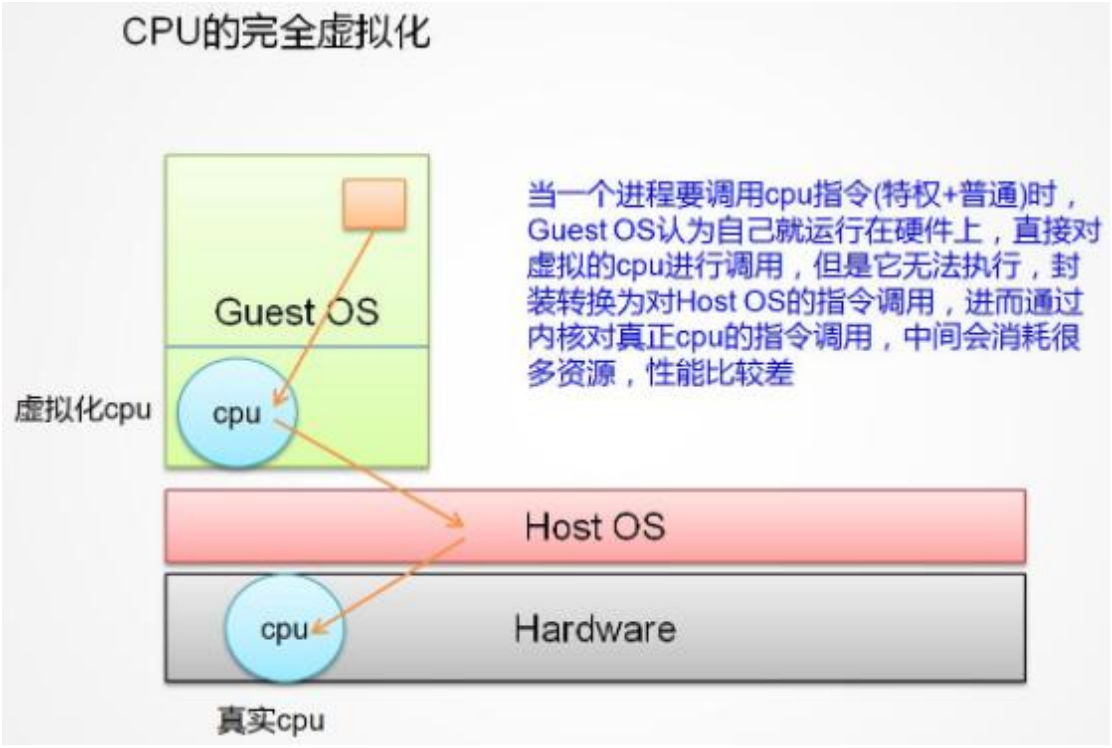
```
Int rte_eth_rx_queue_setup(uint16_t port_id, uint16_t rx_queue_id,
              uint16_t nb_rx_desc, unsigned int socket_id,
              const struct rte_eth_rxconf *rx_conf,
              struct rte_mempool *mp)

int rte_eth_tx_queue_setup(uint16_t port_id, uint16_t tx_queue_id,
              uint16_t nb_tx_desc, unsigned int socket_id,
              const struct rte_eth_txconf *tx_conf)


inline uint16_t rte_eth_rx_burst(uint16_t port_id, uint16_t queue_id,
struct rte_mbuf **rx_pkts, const uint16_t nb_pkts)

static inline uint16_t rte_eth_tx_burst(uint16_t port_id, uint16_t
queue_id, struct rte_mbuf **tx_pkts, uint16_t nb_pkts)
```

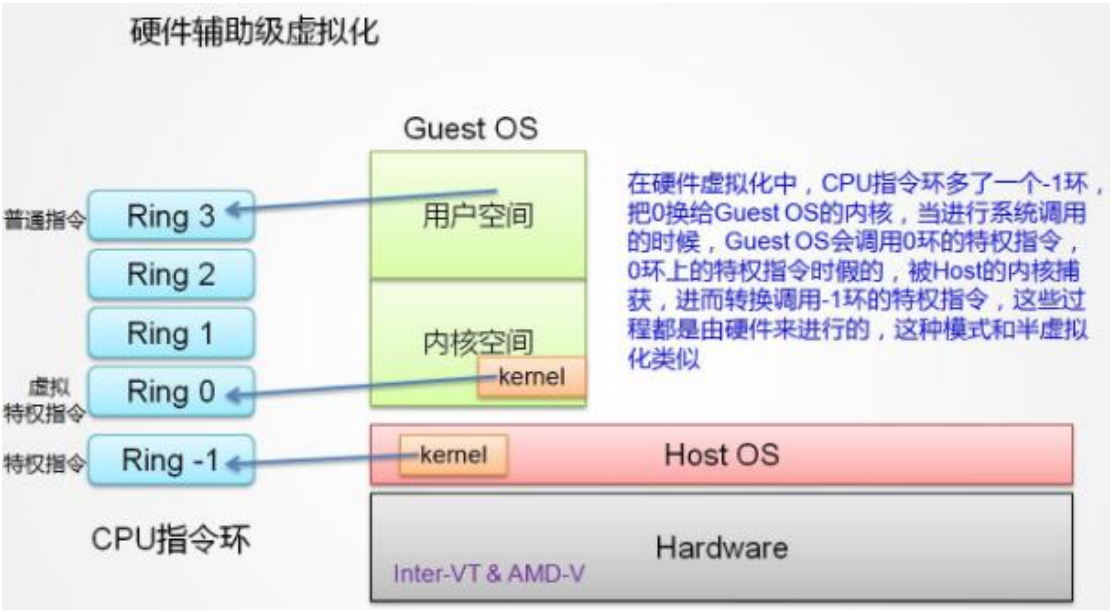
```
const int num_rx_queues = 1;
const int num_tx_queues = 1;
struct rte_eth_conf port_conf = port_conf_default;
if (rte_eth_dev_configure(g_dpdkPortId, num_rx_queues, num_tx_queues, &por
    rte_exit(EXIT_FAILURE, "rte_eth_dev_configure() failed.\n");
}

uint16_t nb_txd = TX_RING_SIZE;
uint16_t nb_rxd = RX_RING_SIZE;
rte_eth_dev_adjust_nb_rx_tx_desc(g_dpdkPortId, &nb_rxd, &nb_txd);

// Set up RX queue.
struct rte_eth_rxconf rxq_conf = dev_info.default_rxconf;
rxq_conf.offloads = local_port_conf.rxmode.offloads;
if (rte_eth_rx_queue_setup(g_dpdkPortId, DPDK_QUEUE_ID_RX, RX_RING_SIZE,
        rte_eth_dev_socket_id(g_dpdkPortId), &rxq_conf, mbuf_pool) < 0) {
    rte_exit(EXIT_FAILURE, "Couldn't setup RX queue.\n");
}

// Set up TX queue.
struct rte_eth_txconf txq_conf = dev_info.default_txconf;
txq_conf.offloads = local_port_conf.txmode.offloads;
if (rte_eth_tx_queue_setup(g_dpdkPortId, 0, nb_txd,
        rte_eth_dev_socket_id(g_dpdkPortId), &txq_conf) < 0) {
    rte_exit(EXIT_FAILURE, "Couldn't setup TX queue.\n");
}
```

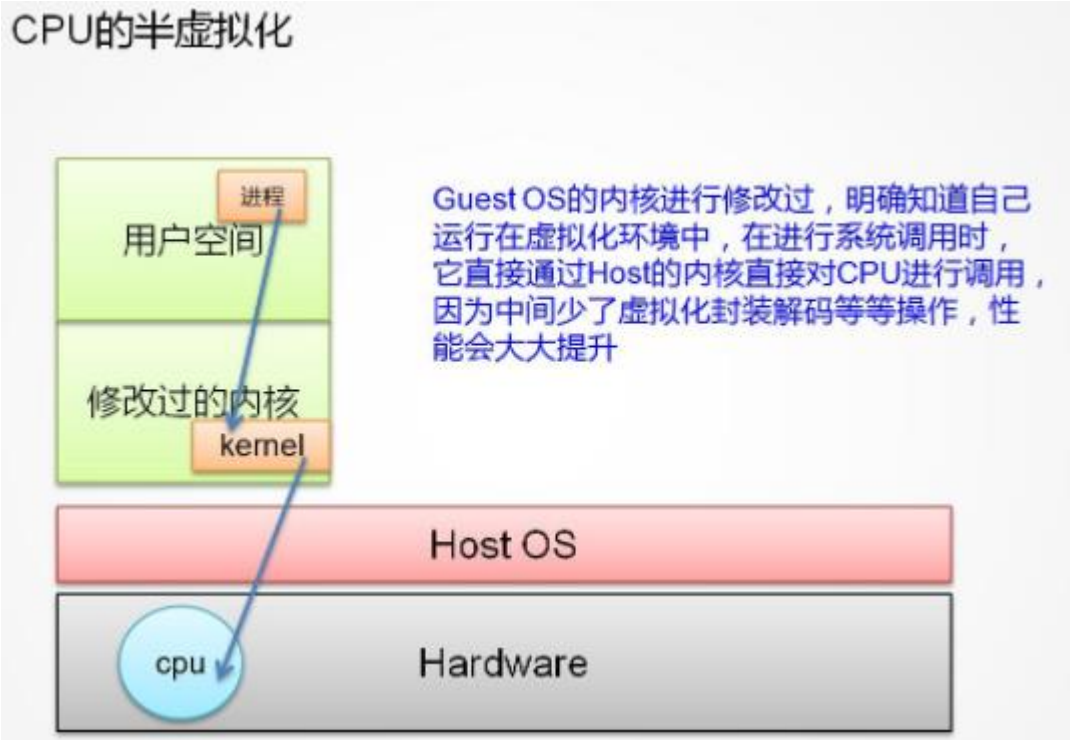
虚拟化

虚拟化是资源的逻辑表示，不受物理设备的约束。虚拟化技术的实现形式是在系统中加入一个虚拟化层，虚拟化层将下层的资源抽象成另一种形式的资源，提供给上层使用。

CPU 虚拟化


CPU的完全虚拟化
Guest OS
虚拟化cpu
cpu
Host OS
cpu
Hardware
真实cpu
当一个进程要调用cpu指令(特权+普通)时，Guest OS认为自己就运行在硬件上，直接对虚拟的cpu进行调用，但是它无法执行，封装转换为对Host OS的指令调用，进而通过内核对真正cpu的指令调用，中间会消耗很多资源，性能比较差


硬件辅助级虚拟化
Guest OS
普通指令
Ring 3
Ring 2
Ring 1
虚拟特权指令
Ring 0
特权指令
Ring -1
CPU指令环
用户空间
内核空间
kernel
kernel
Host OS
Hardware
Inter-VT & AMD-V
在硬件虚拟化中，CPU指令环多了一个-1环，把0换给Guest OS的内核，当进行系统调用的时候，Guest OS会调用0环的特权指令，0环上的特权指令时假的，被Host的内核捕获，进而转换调用-1环的特权指令，这些过程都是由硬件来进行的，这种模式和半虚拟化类似



内存虚拟化

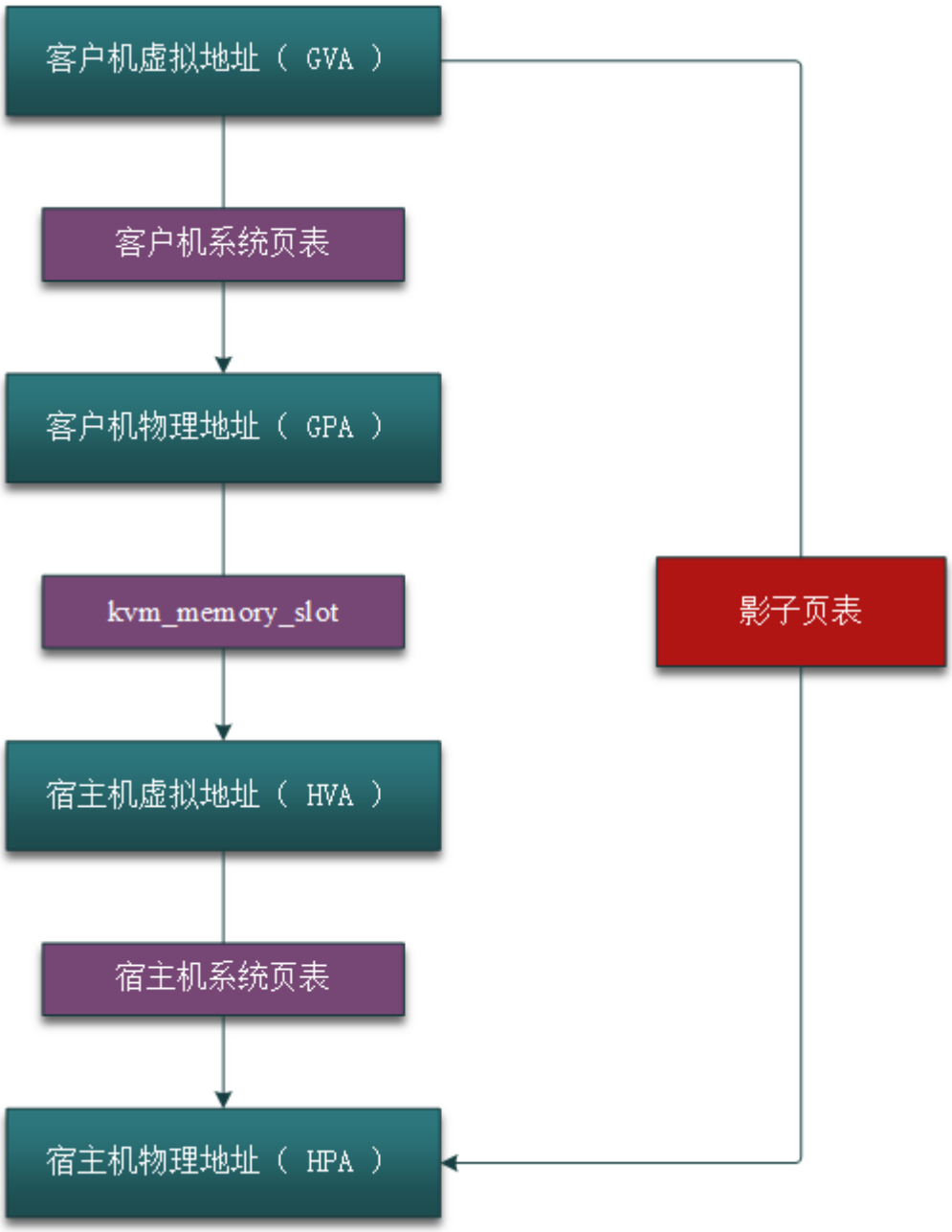
虚拟机本质上是 Host 机上的一个进程，按理说应该可以使用 Host 机的虚拟地址空间，但由于在虚拟化模式下，虚拟机处于非 Root 模式，无法直接访问 Root 模式下的 Host 机上的内存。

这个时候就需要 VMM 的介入，VMM 需要 intercept（截获）虚拟机的内存访问指令，然后 virtualize（模拟）Host 上的内存，相当于 VMM 在虚拟机的虚拟地址空间和 Host 机的虚拟地址空间中间增加了一层，即虚拟机的物理地址空间，也可以看作是 Qemu 的虚拟地址空间（稍微有点绕，但记住一点，虚拟机是由 Qemu 模拟生成的就比较清楚了）。

所以，内存软件虚拟化的目标就是要将虚拟机的虚拟地址（Guest Virtual Address, GVA）转化为 Host 的物理地址（Host Physical Address, HPA），中间要经过虚拟机的物理地址（Guest Physical Address, GPA）和 Host 虚拟地址（Host Virtual Address）的转化，即：

GVA -> GPA -> HVA -> HPA

其中前两步由虚拟机的系统页表完成，中间两步由 VMM 定义的映射表（由数据结构 kvm_memory_slot 记录）完成，它可以将连续的虚拟机物理地址映射成非连续的 Host 机虚拟地址，后面两步则由 Host 机的系统页表完成。如下图所示。



这样做得目的有两个：

1. 提供给虚拟机一个从零开始的连续的物理内存空间。
2. 在各虚拟机之间有效隔离、调度以及共享内存资源。

影子页表技术

接上图，我们可以看到，传统的内存虚拟化方式，虚拟机的每次内存访问都需要 VMM 介入，并由软件进行多次地址转换，其效率是非常低的。因此才有了影子页表技术和 EPT 技术。

影子页表简化了地址转换的过程，实现了 Guest 虚拟地址空间到 Host 物理地址空间的直接映射。

要实现这样的映射，必须为 Guest 的系统页表设计一套对应的影子页表，然后将影子页表装入 Host 的 MMU 中，这样当 Guest 访问 Host 内存时，就可以根据 MMU 中的影子页表映射关系，完成 GVA 到 HPA 的直接映射。而维护这套影子页表的工作则由 VMM 来完成。

由于 Guest 中的每个进程都有自己的虚拟地址空间，这就意味着 VMM 要为 Guest 中的每个进程页表都维护一套对应的影子页表，当 Guest 进程访问内存时，才将该进程的影子页表装入 Host 的 MMU 中，完成地址转换。

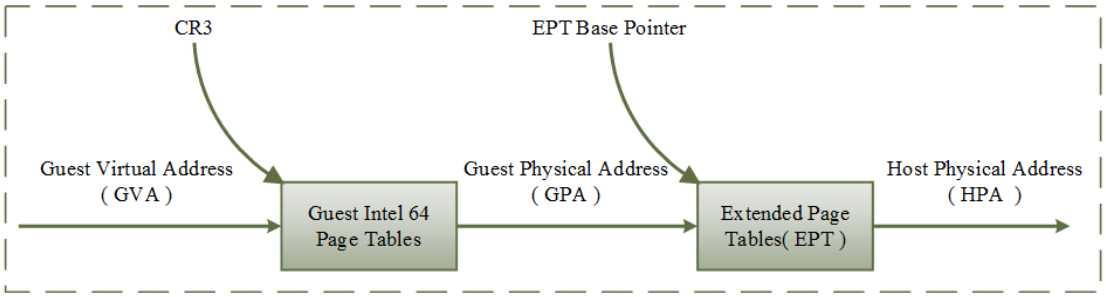
我们也看到，这种方式虽然减少了地址转换的次数，但本质上还是纯软件实现的，效率还是不高，而且 VMM 承担了太多影子页表的维护工作，设计不好。

为了改善这个问题，就提出了基于硬件的内存虚拟化方式，将这些繁琐的工作都交给硬件来完成，从而大大提高了效率。

EPT 技术（Extended Page Table）

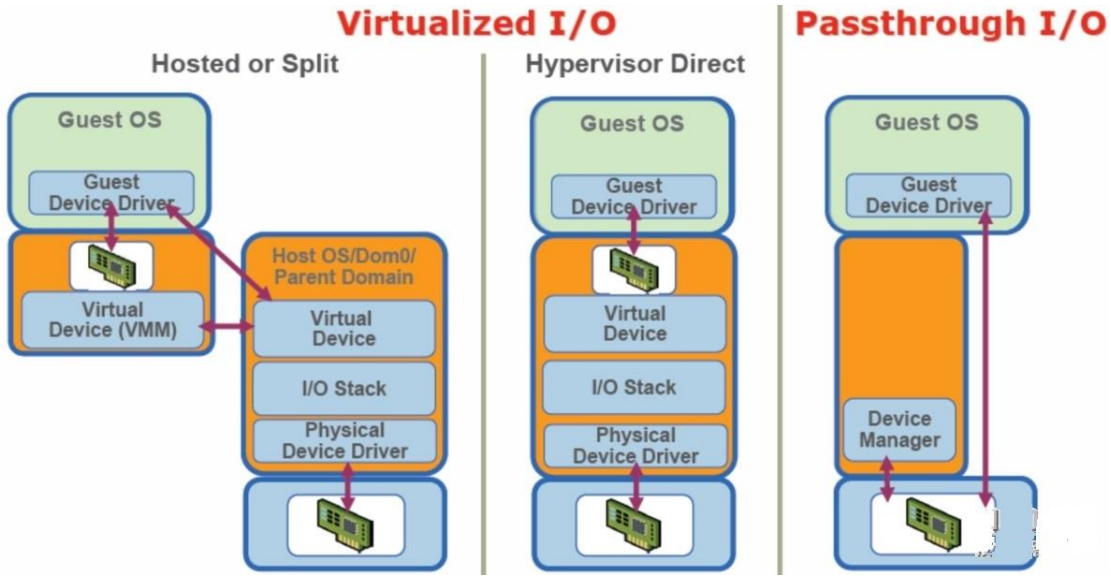
这方面 Intel 和 AMD 走在了最前面，Intel 的 EPT 和 AMD 的 NPT 是硬件辅助内存虚拟化的代表，两者在原理上类似，本文重点介绍一下 EPT 技术。

如下图是 EPT 的基本原理图示，EPT 在原有 CR3 页表地址映射的基础上，引入了 EPT 页表来实现另一层映射，这样，GVA->GPA->HPA 的两次地址转换都由硬件来完成



这里举一个小例子来说明整个地址转换的过程。假设现在 Guest 中某个进程需要访问内存，CPU 首先会访问 Guest 中的 CR3 页表来完成 GVA 到 GPA 的转换，如果 GPA 不为空，则 CPU 接着通过 EPT 页表来实现 GPA 到 HPA 的转换（实际上，CPU 会首先查看硬件 EPT TLB 或者缓存，如果没有对应的转换，才会进一步查看 EPT 页表），如果 HPA 为空呢，则 CPU 会抛出 EPT Violation 异常由 VMM 来处理。如果 GPA 地址为空，即缺页，则 CPU 产生缺页异常，注意，这里，如果是软件实现的方式，则会产生 VM-exit，但是硬件实现方式，并不会发生 VM-exit，而是按照一般的缺页中断处理，这种情况下，也就是交给 Guest 内核的中断处理程序处理。在中断处理程序中会产生 EXIT_REASON_EPT_VIOLATION，Guest 退出，VMM 截获到该异常后，分配物理地址并建立 GVA 到 HPA 的映射，并保存到 EPT 中，这样在下次访问的时候就可以完成从 GVA 到 HPA 的转换了。

I/O 虚拟化



I/O 全虚拟化（上图中间部分）

这种方式比较好理解，简单来说，就是通过纯软件的形式来模拟虚拟机的 I/O 请求。以 qemu-kvm 来举例，内核中的 kvm 模块负责截获 I/O 请求，然后通过事件通知告知给用户空间的设备模型 qemu，qemu 负责完成本次 I/O 请求的模拟。

优点：不需要对操作系统做修改，也不需要改驱动程序，因此这种方式对于多种虚拟化技术的「可移植性」和「兼容性」比较好。
缺点：纯软件形式模拟，自然性能不高，另外，虚拟机发出的 I/O 请求需要虚拟机和 VMM 之间的多次交互，产生大量的上下文切换，造成巨大的开销。

I/O 半虚拟化（上图左侧）

针对 I/O 全虚拟化纯软件模拟性能不高这一点，I/O 半虚拟化前进了一步。它提供了一种机制，使得 Guest 端与 Host 端可以建立连接，直接通信，摒弃了截获模拟这种方式，从而获得较高的性能。
值得注意的有两点：1）采用 I/O 环机制，使得 Guest 端和 Host 端可以共享内存，减少了虚拟机与 VMM 之间的交互；2）采用事件和回调的机制来实现 Guest 与 Host VMM 之间的通信。这样，在进行中断处理时，就可以直接采用事件和回调机制，无需进行上下文切换，减少了开销。
要实现这种方式， Guest 端和 Host 端需要采用类似于 C/S 的通信方式建立连接，这也就意味着要修改 Guest 和 Host 端操作系统内核相应的代码，使之满足这样的要求。为了描述方便，我们统称 Guest 端为前端，Host 端为后端。
前后端通常采用的实现方式是驱动的方式，即前后端分别构建通信的驱动模块，前端实现在内核的驱动程序中，后端实现在 qemu 中，然后前后端之间采用共享内存的方式传递数据。

关于这方面一个比较好的开源实现是 virtio

优点：性能较 I/O 全虚拟化有了较大的提升
缺点：要修改操作系统内核以及驱动程序，因此会存在移植性和适用性方面的问题，导致其使用受限。

I/O 直通或透传技术（上图右侧）

上面两种虚拟化方式，还是从软件层面上来实现，性能自然不会太高。最好的提高性能的方式还是从硬件上来解决。如果让虚拟机独占一个物理设备，像宿主机一样使用物理设备，那无疑性能是最好的。

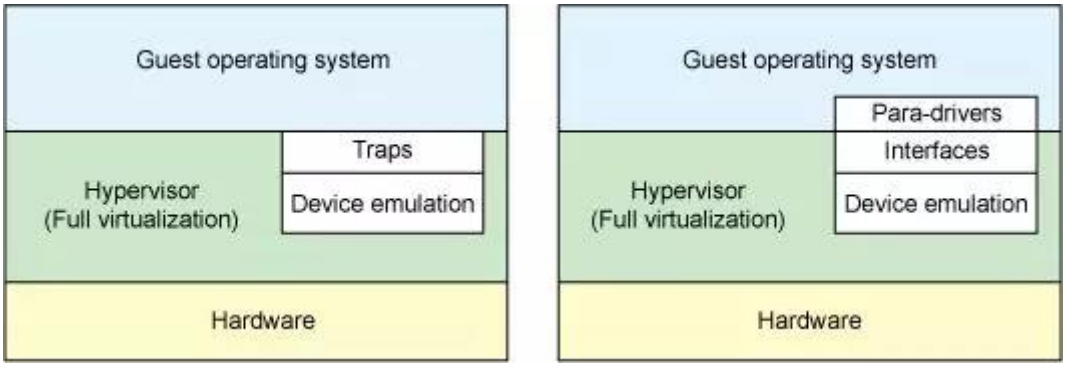
I/O 直通技术就是提出来完成这样一件事的。它通过硬件的辅助可以让虚拟机直接访问物理设备，而不需要通过 VMM 或被 VMM 所截获。

由于多个虚拟机直接访问物理设备，会涉及到内存的访问，而内存又是共享的，那怎么来隔离各个虚拟机对内存的访问呢，这里就要用到一门技术——IOMMU，简单说，IOMMU 就是用来隔离虚拟机对内存资源访问的。

I/O 直通技术需要硬件支持才能完成，这方面首选是 Intel 的 VT-d 技术，它通过对芯片级的改造来达到这样的要求，这种方式固然对性能有着质的提升，不需要修改操作系统，移植性也好。但该方式也是有一定限制的，这种方式仅限于物理资源丰富的机器，因为这种方式仅仅能满足一个设备分配给一个虚拟机，一旦一个设备被虚拟机占用了，其他虚拟机时无法使用该设备的。

Virtio

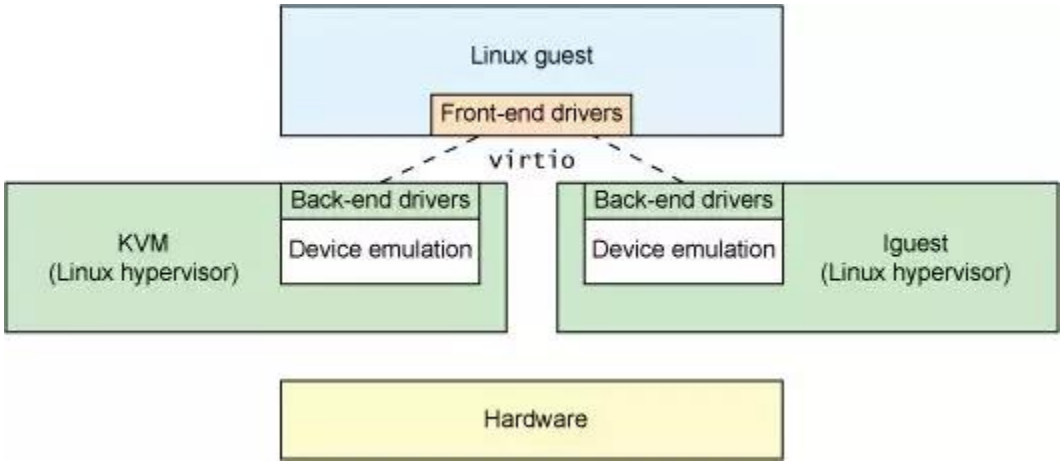
virtio 是一种 I/O 半虚拟化解决方案，是一套通用 I/O 设备虚拟化的程序，是对半虚拟化 Hypervisor 中的一组通用 I/O 设备的抽象。提供了一套上层应用与各 Hypervisor 虚拟化设备（KVM，Xen，VMware 等）之间的通信框架和编程接口，减少跨平台所带来的兼容性问题，大大提高驱动程序开发效率。



为什么是 virtio?

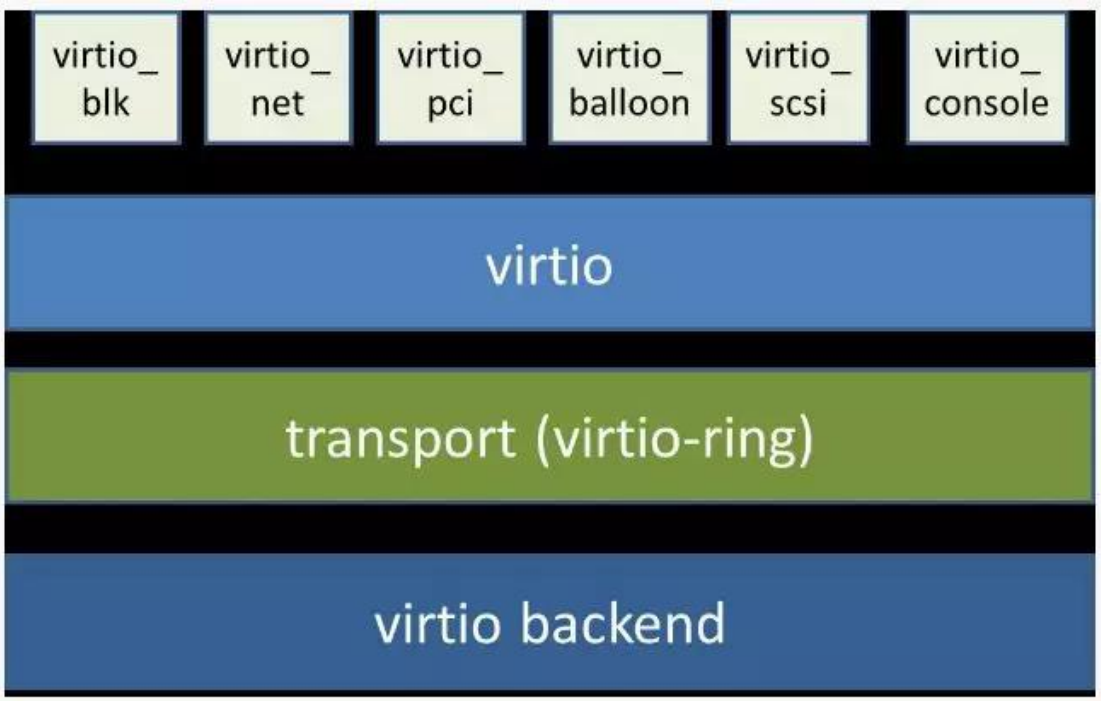
在完全虚拟化的解决方案中，guest VM 要使用底层 host 资源，需要 Hypervisor 来截获所有的请求指令，然后模拟出这些指令的行为，这样势必会带来很多性能上的开销。半虚拟化通过底层硬件辅助的方式，将部分没必要虚拟化的指令通过硬件来完成，Hypervisor 只负责完成部分指令的虚拟化，要做到这点，需要 guest 来配合，guest 完成不同设备的前端驱动程序，Hypervisor 配合 guest 完成相应的后端驱动程序，这样两者之间通过某种交互机制就可以实现高效的虚拟化过程。

由于不同 guest 前端设备其工作逻辑大同小异（如块设备、网络设备、PCI 设备、balloon 驱动等），单独为每个设备定义一套接口实属没有必要，而且还要考虑扩平台的兼容性问题，另外，不同后端 Hypervisor 的实现方式也大同小异（如 KVM、Xen 等），这个时候，就需要一套通用框架和标准接口（协议）来完成两者之间的交互过程，virtio 就是这样一套标准，它极大地解决了这些不通用的问题。



virtio 的架构

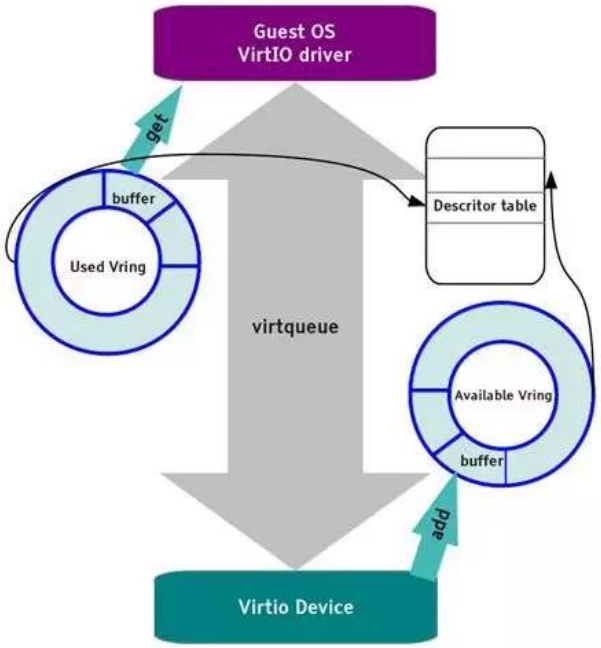
从总体上看，virtio 可以分为四层，包括前端 guest 中各种驱动程序模块，后端 Hypervisor（实现在 Qemu 上）上的处理程序模块，中间用于前后端通信的 virtio 层和 virtio-ring 层，virtio 这一层实现的是虚拟队列接口，算是前后端通信的桥梁，而 virtio-ring 则是该桥梁的具体实现，它实现了两个环形缓冲区，分别用于保存前端驱动程序和后端处理程序执行的信息。



严格来说，virtio 和 virtio-ring 可以看做是一层，virtio-ring 实现了 virtio 的具体通信机制和数据流程。或者这么理解可能更好，virtio 层属于控制层，负责前后端之间的通知机制（kick，notify）和控制流程，而 virtio-vring 则负责具体数据流转发。

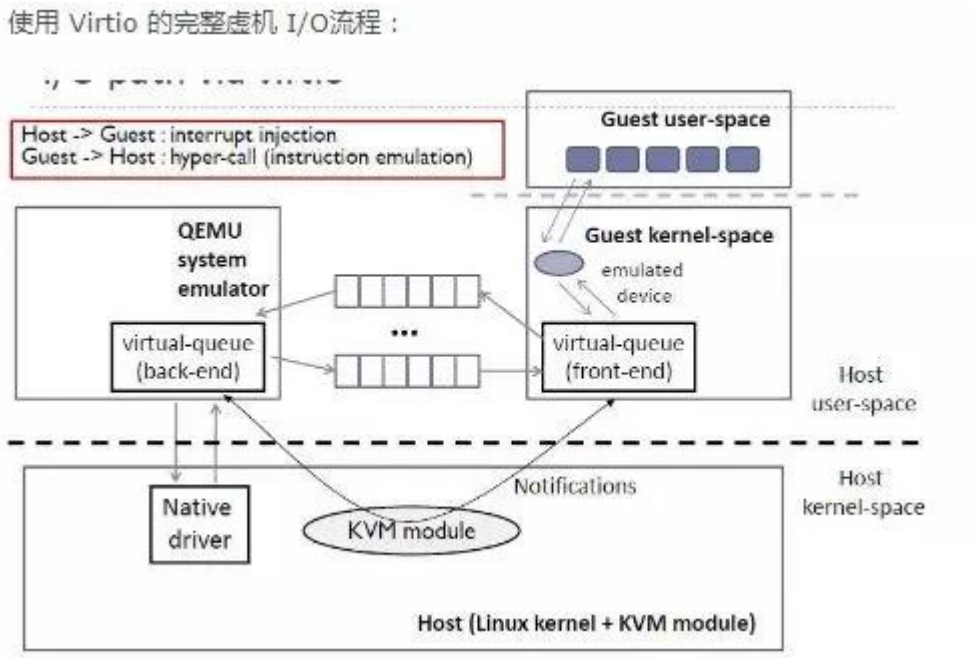
virtio 数据流交互机制

vring 主要通过两个环形缓冲区来完成数据流的转发，如下图所示。



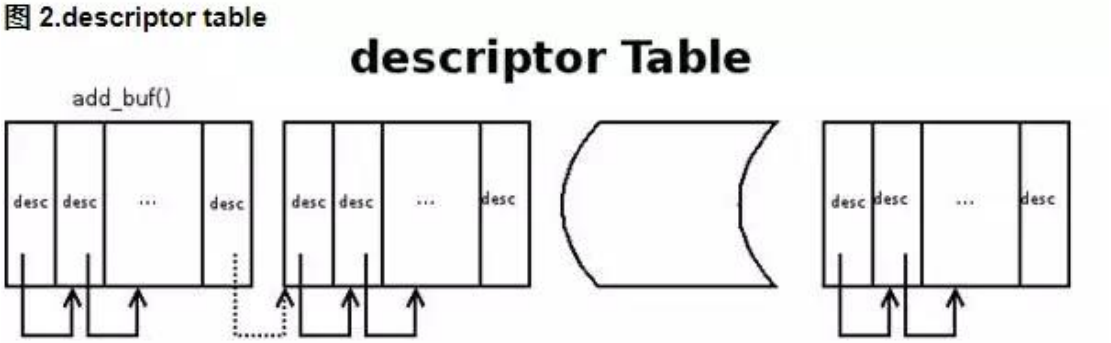
vring 包含三个部分，描述符数组 desc，可用的 available ring 和使用过的 used

ring。
desc 用于存储一些关联的描述符，每个描述符记录一个对 buffer 的描述，available ring 则用于 guest 端表示当前有哪些描述符是可用的，而 used ring 则表示 host 端哪些描述符已经被使用。
Virtio 使用 virtqueue 来实现 I/O 机制，每个 virtqueue 就是一个承载大量数据的队列，具体使用多少个队列取决于需求，例如，virtio 网络驱动程序（virtio-net）使用两个队列（一个用于接受，另一个用于发送），而 virtio 块驱动程序（virtio-blk）仅使用一个队列。
具体的，假设 guest 要向 host 发送数据，首先，guest 通过函数 virtqueue_add_buf 将存有数据的 buffer 添加到 virtqueue 中，然后调用 virtqueue_kick 函数，virtqueue_kick 调用 virtqueue_notify 函数，通过写入寄存器的方式来通知到 host。host 调用 virtqueue_get_buf 来获取 virtqueue 中收到的数据。



存放数据的 buffer 是一种分散-聚集的数组，由 desc 结构来承载，如下是一种常用的 desc 的结构：

图 2.descriptor table

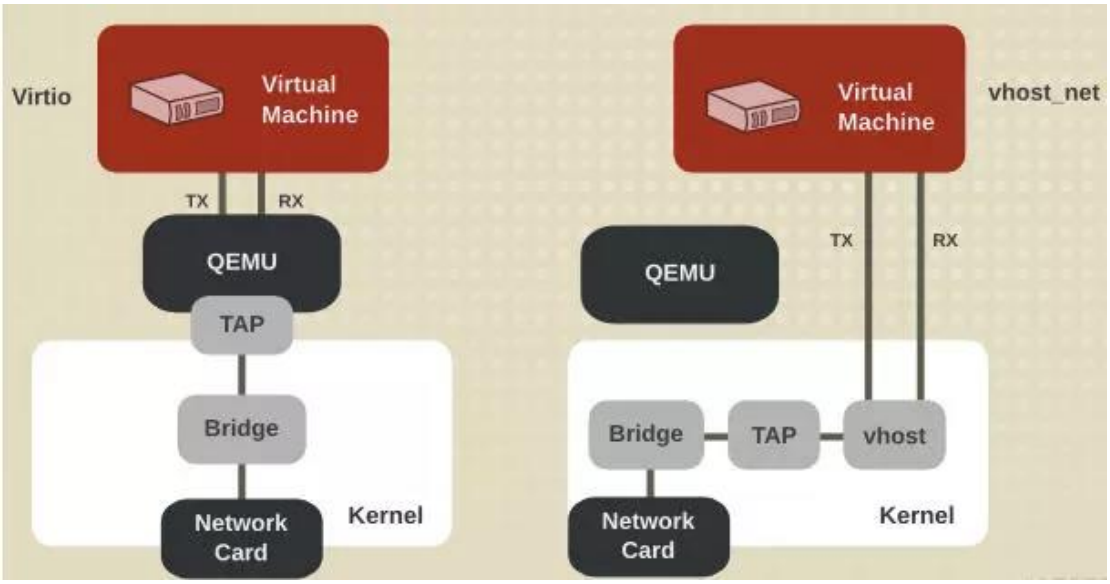


当 guest 向 virtqueue 中写数据时，实际上是向 desc 结构指向的 buffer 中填充数据，完了会更新 available ring，然后再通知 host。
当 host 收到接收数据的通知时，首先从 desc 指向的 buffer 中找到 available ring 中添加的 buffer，映射内存，同时更新 used ring，并通知 guest 接收数据完毕。
virtio 是 guest 与 host 之间通信的润滑剂，提供了一套通用框架和标准接口或协议来完成

两者之间的交互过程，极大地解决了各种驱动程序和不同虚拟化解决方案之间的适配问题。virtio 抽象了一套 vring 接口来完成 guest 和 host 之间的数据收发过程，结构新颖，接口清晰。

Vhost

vhost 是 virtio 的一种后端实现方案，在 virtio 简介中，我们已经提到 virtio 是一种半虚拟化的实现方案，需要虚拟机端和主机端都提供驱动才能完成通信，通常，virtio 主机端的驱动是实现在用户空间的 qemu 中，而 vhost 是实现在内核中，是内核的一个模块 vhost-net.ko。



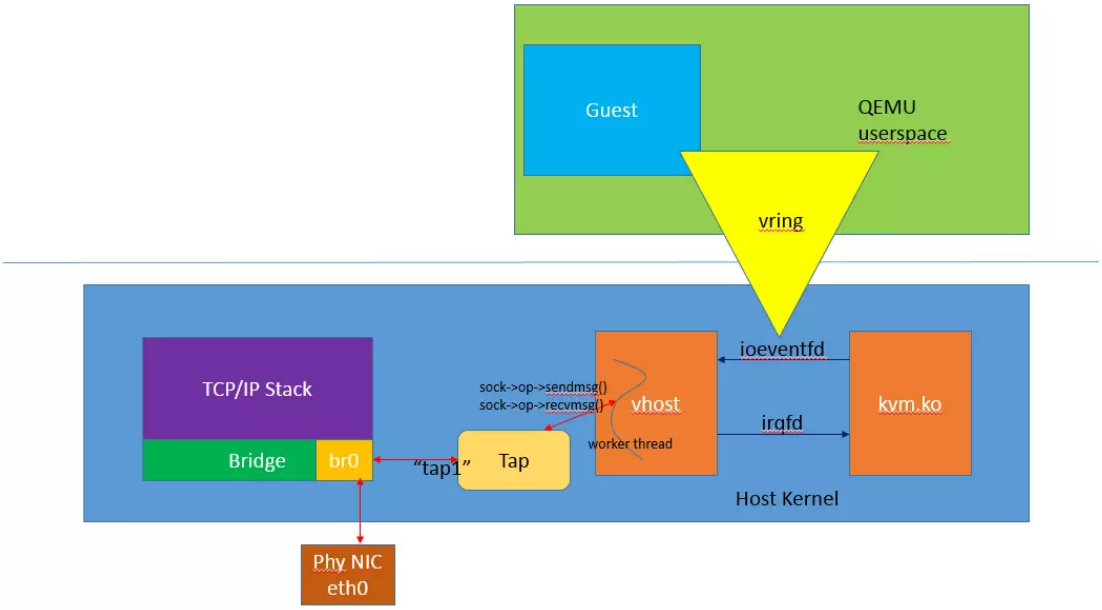
为什么要用 vhost

在 virtio 的机制中，guest 与 用户空间的 Hypervisor 通信，会造成多次的数据拷贝和 CPU 特权级的上下文切换。例如 guest 发包给外部网络，首先，guest 需要切换到 host kernel，然后 host kernel 会切换到 qemu 来处理 guest 的请求， Hypervisor 通过系统调用将数据包发送到外部网络后，会切换回 host kernel ， 最后再切换回 guest。这样漫长的路径无疑会带来性能上的损失。

vhost 正是在这样的背景下提出的一种改善方案，它是位于 host kernel 的一个模块，用于和 guest 直接通信，数据交换直接在 guest 和 host kernel 之间通过 virtqueue 来进行，qemu 不参与通信，但也没有完全退出舞台，它还要负责一些控制层面的事情，比如和 KVM 之间的控制指令的下发等。

vhost 的数据流程

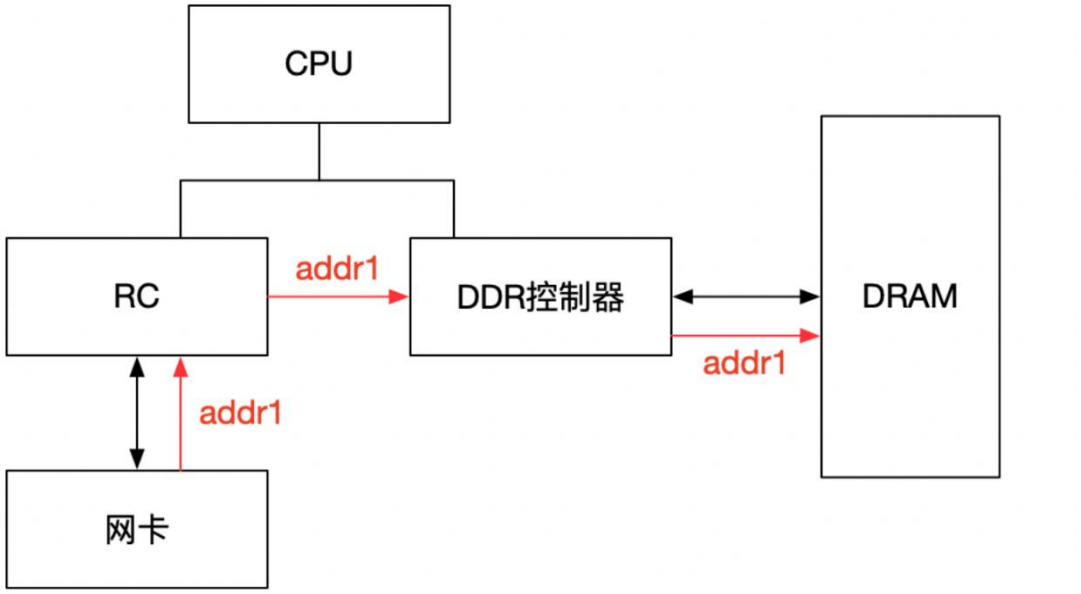
下图左半部分是 vhost 负责将数据发往外部网络的过程，右半部分是 vhost 大概的数据交互流程图。其中，qemu 还是需要负责 virtio 设备的适配模拟，负责用户空间某些管理控制事件的处理，而 vhost 实现较为纯净，以一个独立的模块完成 guest 和 host kernel 的数据交换过程。



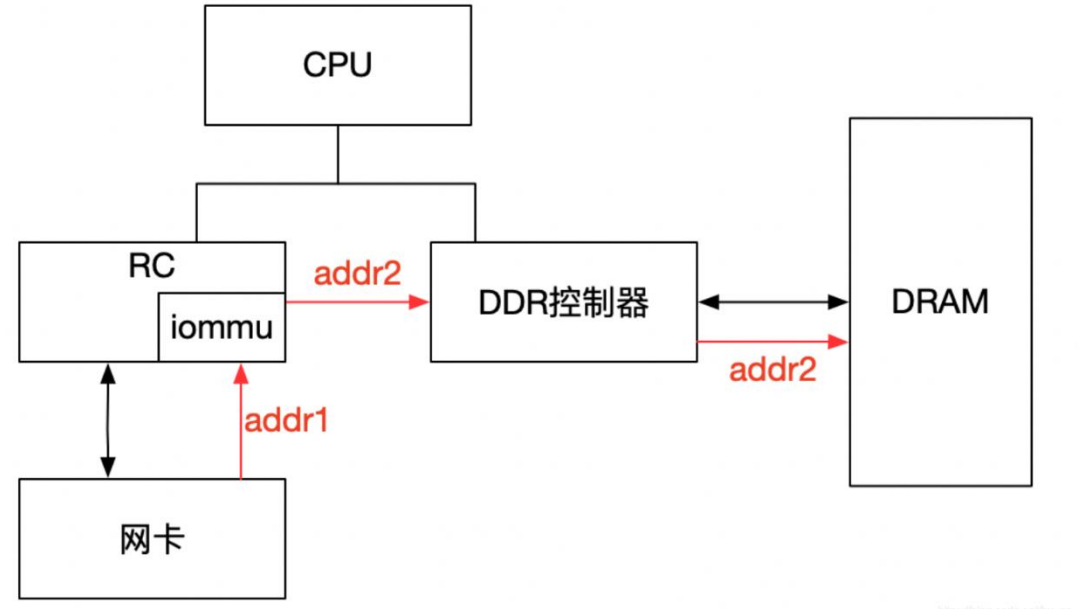
vhost 与 virtio 前端的通信主要采用一种事件驱动 eventfd 的机制来实现，guest 通知 vhost 的事件要借助 kvm.ko 模块来完成，vhost 初始化期间，会启动一个工作线程 work 来监听 eventfd，一旦 guest 发出对 vhost 的 kick event，kvm.ko 触发 ioeventfd 通知到 vhost，vhost 通过 virtqueue 的 avail ring 获取数据，并设置 used ring。同样，从 vhost 工作线程向 guest 通信时，也采用同样的机制，只不过这种情况发的是一个回调的 call envent，kvm.ko 触发 irqfd 通知 guest。

IOMMU

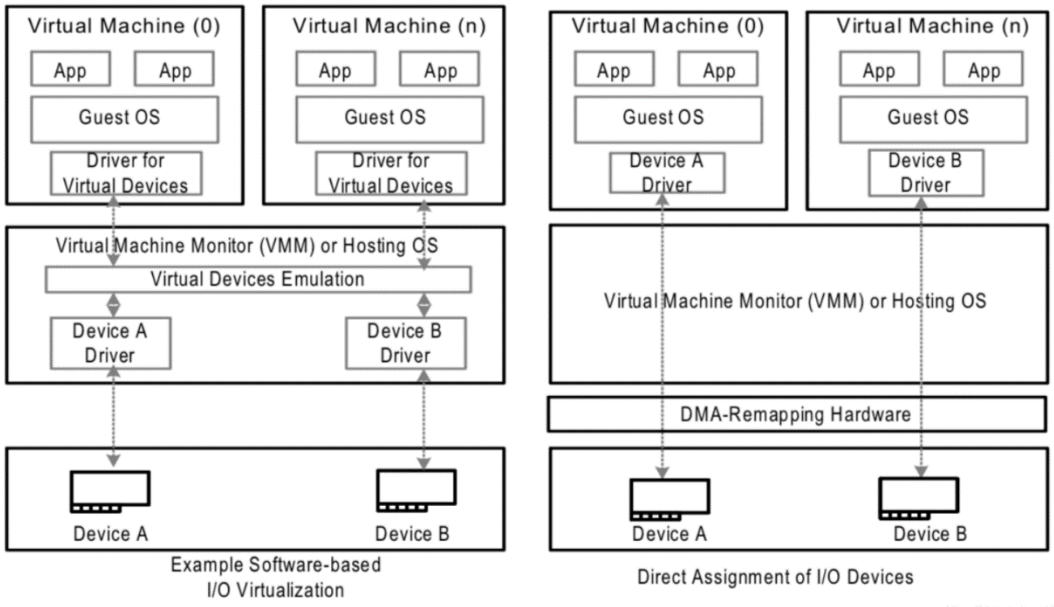
IOMMU 主要功能包括 DMA Remapping 和 Interrupt Remapping，这里主要讲解 DMA Remapping，Interrupt Remapping 会独立讲解。对于 DMA Remapping，IOMMU 与 MMU 类似。IOMMU 可以将一个设备访问地址转换为存储器地址，下图针对有无 IOMMU 情况说明 IOMMU 作用。



在没有 IOMMU 的情况下，网卡接收数据时地址转换流程，RC 会将网卡请求写入地址 addr1 直接发送到 DDR 控制器，然后访问 DRAM 上的 addr1 地址，这里的 RC 对网卡请求地址不做任何转换，网卡访问的地址必须是物理地址。



对于有 IOMMU 的情况，网卡请求写入地址 addr1 会被 IOMMU 转换为 addr2，然后发送到 DDR 控制器，最终访问的是 DRAM 上 addr2 地址，网卡访问的地址 addr1 会被 IOMMU 转换成真正的物理地址 addr2，这里可以将 addr1 理解为虚机地址。



左图是没有IOMMU的情况，对于此种情况虚机无法实现设备的透传，原因主要有两个：一是因为在没有 IOMMU 的情况下，设备必须访问真实的物理地址 HPA，而虚机可见的是GPA；二是如果让虚机填入真正的 HPA，那样的话相当于虚机可以直接访问物理地址，会有安全隐患。所以针对没有 IOMMU 的情况，不能用透传的方式，对于设备的直接访问都会有VMM 接管，这样就不会对虚机暴露 HPA。

右图是有 IOMMU 的情况，虚机可以将 GPA 直接写入到设备，当设备进行 DMA 传输时，设备请求地址 GPA 由 IOMMU 转换为 HPA（硬件自动完成），进而 DMA 操作真实的物理空间。IOMMU 的映射关系是由 VMM 维护的，HPA 对虚机不可见，保障了安全问题，利用 IOMMU 可实现设备的透传。这里先留一个问题，既然 IOMMU 可以将设备访问地址映射成真实的物理地址，那么对于右图中的 Device A 和 Device B，IOMMU 必须保证两个设备映射后的物理空间不能存在交集，否则两个虚机可以相互干扰，这和 IOMMU 的映射原理有关。

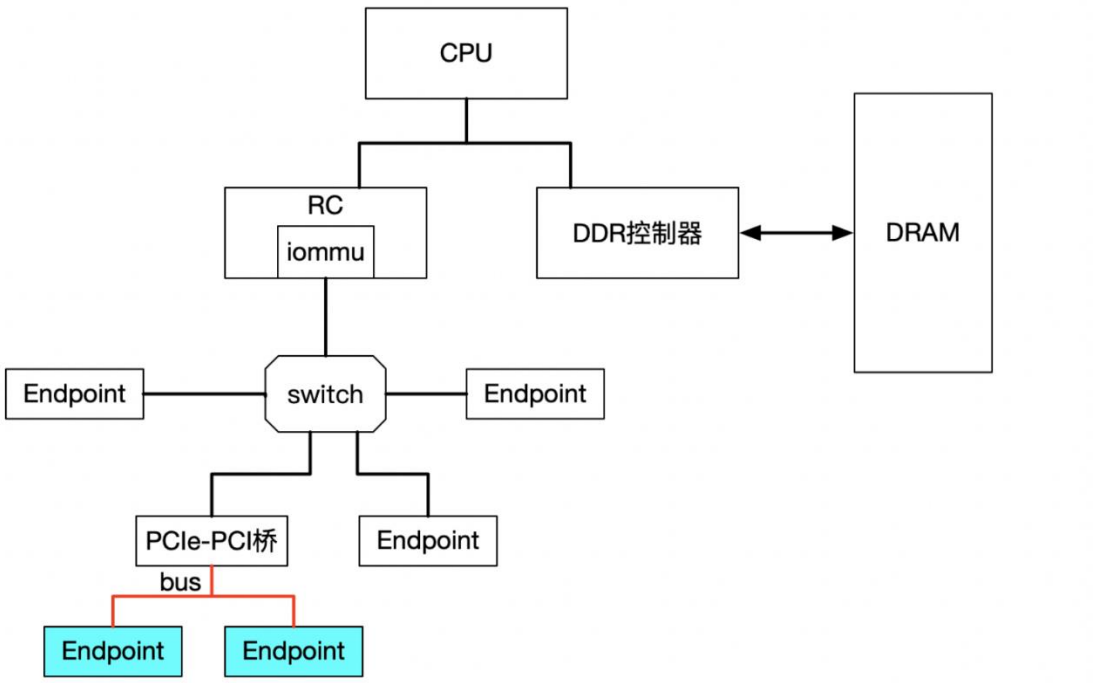
VFIO

VFIO就是内核针对IOMMU提供的软件框架，支持DMA Remapping和Interrupt Remapping，这里只讲 DMA Remapping。VFIO 利用 IOMMU 这个特性，可以屏蔽物理地址对上层的可见性，可以用来。开发用户态驱动，也可以实现设备透传。

Group 和 Container

1) Group：group 是 IOMMU 能够进行 DMA 隔离的最小硬件单元，一个 group 内可能只有一个 device，也可能有多个 device，这取决于物理平台上硬件的 IOMMU 拓扑结构。 设备直通的时候一个 group 里面的设备必须都直通给一个虚拟机。 不能够让一个 group 里的多个 device 分别从属于 2 个不同的 VM，也不允许部分 device 在 host 上而另一部分被分

配到 guest 里， 因为就这样一个 guest 中的 device 可以利用 DMA 攻击获取另外一个 guest 里的数据，就无法做到物理上的 DMA 隔离。
2) Container：对于虚机，Container 这里可以简单理解为一个 VM Domain 的物理内存空间。对于用户态驱动，Container 可以是多个 Group 的集合。



上图中 PCIe-PCI 桥下的两个设备，在发送 DMA 请求时，PCIe-PCI 桥会为下面两个设备生成 Source Identifier，其中 Bus 域为红色总线号 bus，device 和 func 域为 0。这样的话，PCIe-PCI 桥下的两个设备会找到同一个 Context Entry 和同一份页表，所以这两个设备不能分别给两个虚机使用，这两个设备就属于一个 Group。

透传设备具体实现

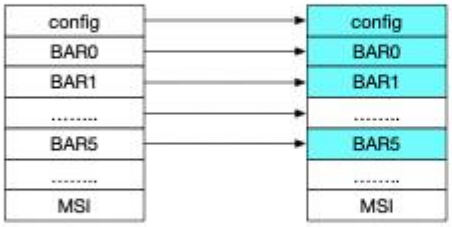
下面从一个物理机启动->虚机启动整个过程描述设备如何透传到虚机，并正常工作的 step 1，物理机启动，BIOS 完成 PCI 设备的配置，包括初始化 config 空间，分配 BAR 地址空间
step 2，由于内核开启 IOMMU 支持，会为当前设备分配 iommu group
step 3，加载 vfio 驱动，并与 mpw 卡关联
step 4，qemu 启动虚机，并将 mpw 卡设备透传给虚机
step 5，前面 4 个步骤中不涉及虚机，纯粹的物理机操作。当虚机启动后（假设虚机运行 linux 内核），会根据 qemu 构建的虚机 PCI 拓扑为 pci 设备初始化，包括配置 config 空间、分配 BAR 地址空间，这里过程与 step1 类似，但是行为差别很大。这里说明一下，对于透传设备、virtio 设备等，不同的设备实现也不同，这里仅讲述对于透传设备的实现
step 6，当虚机配置 mpw 卡的 config 空间时，会用到 in、out 系列 IO 指令，这样会造成虚机的 vm_exit，qemu 中可以截获这个行为，并选择是采取模拟的方式还是利用 vfio 提供的 io 接口实现，详细过程见第 1 节。这里有两个特殊处理，一个是 BAR 空间的配置，一个是中断的配置

Step 7， 首先透传的优势是高效，在虚机使用设备时，触发 vm_exit 的情况越少越好，但是又不能放给虚机过大权限，否则会影响到物理机，比如一些特殊的 config 寄存器，如 Max Payload，中断等。在虚机利用 in、out 指令时会 vm_exit，这个影响不大，目前大部分 PCI 设备的 BAR 空间都是采用 MMIO 的方式访问，当设备正常工作时，大部分采用的是 mmio 的方式。所以要尽可能的保证在 mmio 访问时，不会 vm_exit。也就是将 BAR 空间透传给虚机，根据 EPT 映射关系，如果物理机建立好了 GPA 到 HPA 的映射，就不会 vm_exit，以此来提供效率。先来说 HPA，HPA 即为物理机在 step1 中为 mpw 卡分配的 bar 地址。GPA 为 step5-6 中，虚机为透传设备分配的 BAR 空间，在 step6 中，已经记录了虚机对 BAR 空间的配置，所以也可以获取到 GPA，有了 HPA 和 GPA，建立 EPT 映射就可以实现 BAR 地址空间的透传。不过有一种情况需要特殊考虑，就是 MSI-X。因为 MSI-X 的 table 会放在 BAR 空间上，而虚机是不允许直接访问中断相关配置的，所以对于 MSI-X table 的相关 BAR 空间是不允许直接透传给虚机的

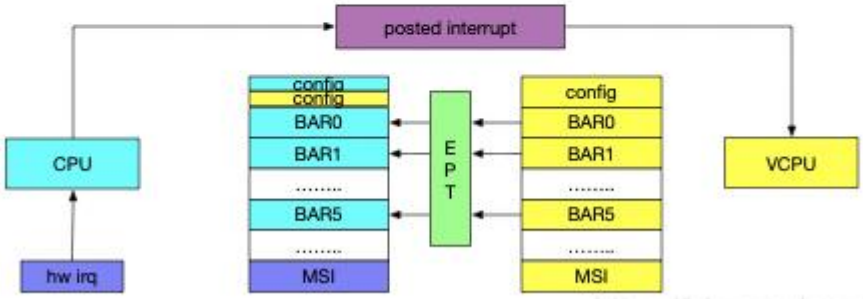
step 8，以使用 MSI 中断的情况为例，当虚拟配置透传设备的 MSI 相关寄存器时，会 vm_exit。qemu 会记录虚机内透传设备使用的中断号并传递给 kvm，同时利用 vfio 的 VFIO_DEVICE_SET_IRQS 命令在物理机中注册中断，当硬件设备产生中断时，首先物理机会执行服务程序，服务程序主要工作是发送 eventfd_signal，激活 kvm 中的 irqfd_inject，最终调用 deliver_posted_interrupt 向虚机注入中断，祥见第 3 节。

通过以上步骤，虚机可以访问设备的 config 空间(模拟或直接访问物理设备)，可以访问 BAR 空间(除 MSI-X table 外都可透传)，中断(利用 vfio VFIO_DEVICE_SET_IRQS、kvm 协同实现)。

1.上电启动，BIOS/kernel初始化PCI设备
1）初始化config空间
2）分配BAR空间，分配后可直接通过mmio访问
3）为热插拔预留资源
4）开启iommu=on支持，当前设备分配iommu group供vfio使用
5）为透传设备加载VFIO驱动，供qemu使用



2.设备透传给虚机，虚机启动
1）BIOS/kernel初始化设备，利用In/out指令访问设备config空间时会产生vm_exit，
由qemu和vfio决定是模拟还是透传给物理设备
2）对于BAR空间的配置，一定不能透传给物理设备，否则设备将无法正常工作，
qemu记录虚机试图配置的GPA地址，利用vfio提供接口实现GPA到HPA映射
3）虚机加载透传设备驱动，驱动程序中会注册中断函数，虚机初始化MSI，会vm_exit。
qemu会记录虚机内透传设备使用的中断号并传递给kvm，同时利用vfio的
VFIO_DEVICE_SET_IRQS命令在物理机中注册中断



VT-x 与 VT-d

VT-x:
原理：CPU 运行有 Ring0 ~ Ring3，一些底层操作必须 Ring0。如果没有 VT-x，虚拟机软件只能到 Ring1，那么有些内核级别的东西就必须靠软件模拟，而效率降低。 有了 VT-x，相当于多出来一套虚拟机的 Ring0 ~ Ring3，这样在虚拟机内的内核请求和虚拟机外的就等于性质上/效率上没有差别了，从而提高效率。

用途：提高虚拟机效率，让虚拟机没有 CPU 性能的短板（当然还是受限于你 CPU 本身的能力）。另外，在 32 位系统上要跑 64 位虚拟机的话，也必须要 VT-x 支持。

VT-d：

原理：是一种基于 North Bridge 北桥芯片（或者按照较新的说法：MCH）的硬件辅助虚拟化技术，通过在北桥中内置提供 DMA 虚拟化和 IRQ 虚拟化硬件，实现了新型的 I/O 虚 拟化方式，Intel VT-d 能够在虚拟环境中大大地提升 I/O 的可靠性、灵活性与性能。

用途：运用 VT-d 技术，虚拟机得以使用直接 I/O 设备分配方式或者 I/O 设备共享方式来代替传统的设备模拟/额外设备接口方式，从而大大提升了虚拟化的 I/O 性能，让虚拟机性能更接近于真实主机。