DPU study -software

jeckypei@163.com

wechat: realjecky

GITHUB REPO: https://github.com/jeckypei/dpu-knowledge

DPU

DPU is not only for network, also for storage, and other HW acceleration.

VM,Container,Host all could use DPU.

VM: ovs

DPU Software work

SRIOV :

- Linux Kernel PF/VF driver --to do
- Windows VF driver --to do
- DPDK PF/VF driver --to do

VIRTIO:

- DPDK vhost driver
- Kernel vdpa mgmtdev driver --to do
- DPDK vdpa driver --to do

SDK: supply API to control and configure DPU

- upgrade firmware
- management for ip/route
- interfaces add/remove
- ovs configuration
- ....

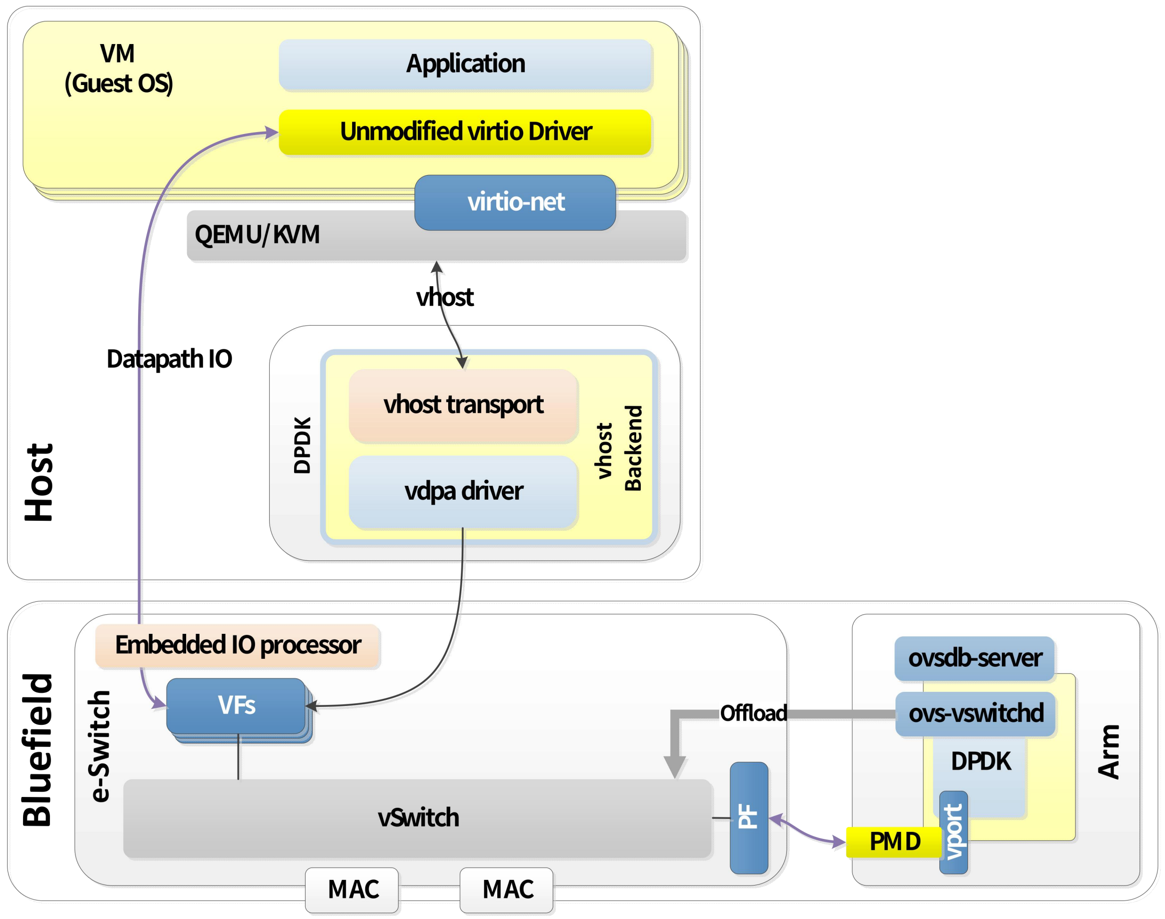
知识储备：

- VIRTIO , IOMMU/VFIO , DPDK , KVM/QEMU
- SRIOV , PCI
- OVS spec

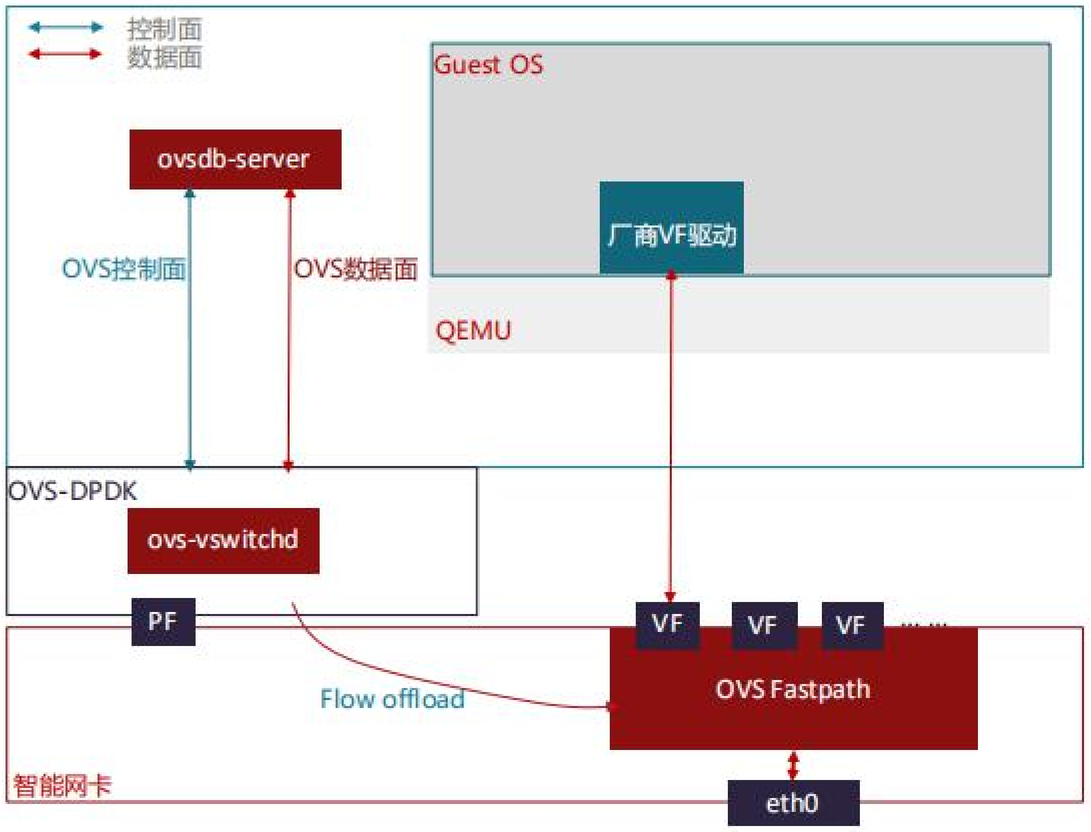
Docs about Smart NIC

- http://t.zoukankan.com/shaohef-p-12227496.html
- https://www.design-reuse.com/articles/46833/how-to-design-smartnics-using-fpgas-to-increase-server-compute-capacity.html

VM net HW Offload, solution1 --vDPA

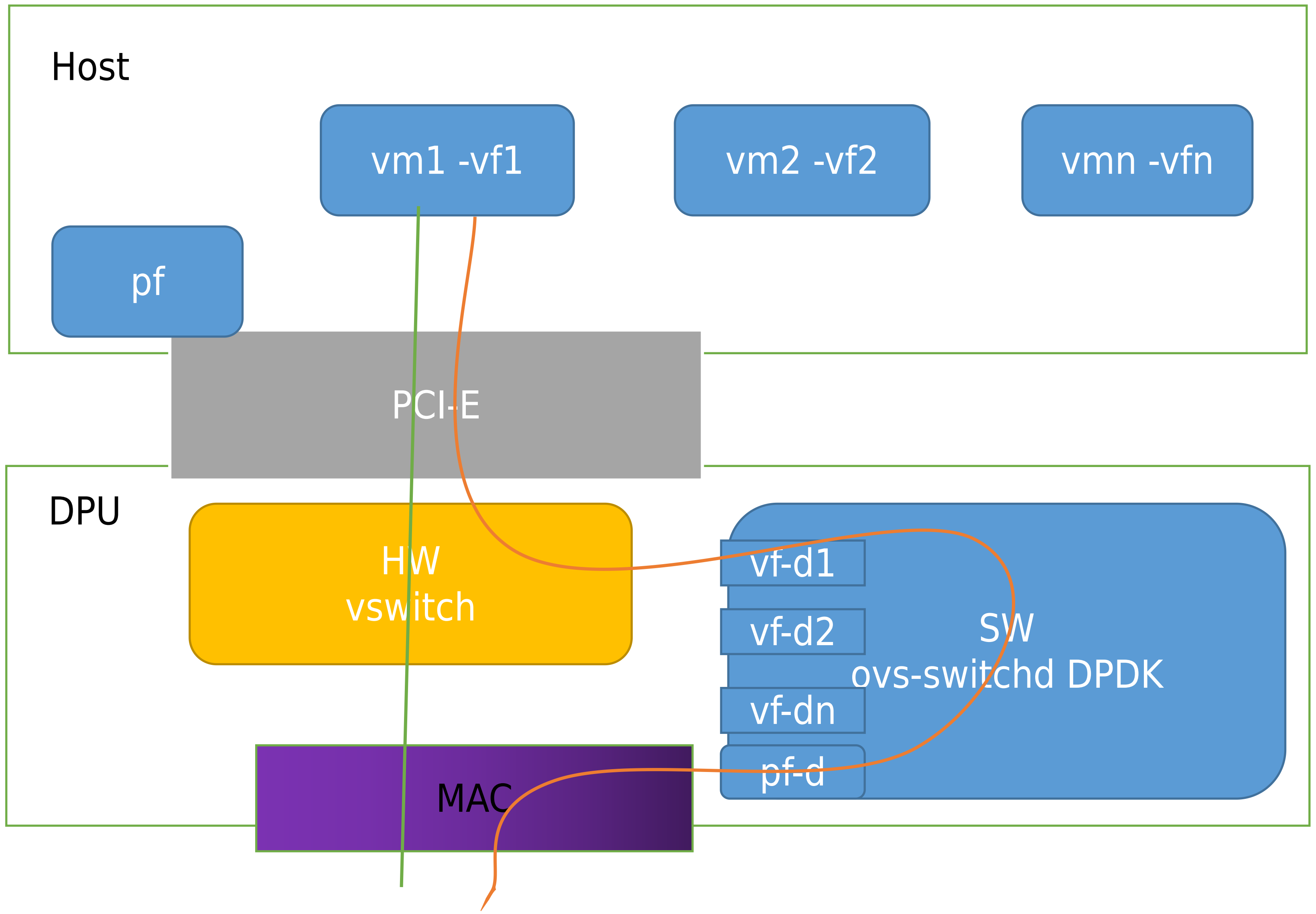


VM net HW Offload, solution2 --SRIOV



OVS Fastpath硬件卸载虚拟化网络

VM net HW Offload, solution-2.1 SIOV



KVM 虚拟机的热迁移 ---Live Migration

热迁移（又叫动态迁移、实时迁移），即虚拟机保存（ save ）/ 恢复 (restore) ：将整个虚拟机的运行状态完整保存下来，同时可以快速的恢复到原有硬件平台甚至是不同硬件平台上。恢复以后，虚拟机仍旧平滑运行，用户不会察觉到任何差异.

SRIO： VF PCI address 太实， 迁移的时候PCI address未必能保持一致

VIRT-IO ： 虚拟的，迁移容易保持一致。

VIRT-IO

Spec: https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html

// you should read this spec more and more times.

virtio is the most important interface for VM guest and Host .

VIRT-IO protocol -- split queue

```
struct vring {
        unsigned int num;
        vring_desc_t *desc;
        vring_avail_t *avail;
        vring_used_t *used;
};
```

VIRT-IO protocol -- split queue

```
/* Virtio ring descriptors: 16 bytes.  These can chain together via "next". */
struct vring_desc {
        /* Address (guest-physical). */
        __virtio64 addr;
        /* Length. */
        __virtio32 len;
        /* The flags as indicated above. */
        __virtio16 flags;
        /* We chain unused descriptors via this, too */
        __virtio16 next;  // link desc, scatter/gather
};
```

VIRT-IO protocol -- split queue

```
/* Virtio ring descriptors: 16 bytes.  These can chain together via "next". */
struct vring_desc {
        __virtio16 flags;
};
/* This marks a buffer as continuing via the next field. */
#define VRING_DESC_F_NEXT       1
/* This marks a buffer as write-only (otherwise read-only). */
#define VRING_DESC_F_WRITE      2
/* This means the buffer contains a list of buffer descriptors. */
#define VRING_DESC_F_INDIRECT   4
```

VIRT-IO protocol -- split queue

```
struct virtq_avail {
        #define VIRTQ_AVAIL_F_NO_INTERRUPT 1
        le16 flags;
        le16 idx;
        le16 ring[ /* Queue Size */ ];
        le16 used_event; /* Only if VIRTIO_F_EVENT_IDX */
};
```

VIRT-IO protocol -- split queue

VIRT-IO protocol -- split queue

VIRT-IO protocol -- split queue

VIRT-IO protocol -- split queue

VIRT-IO protocol -- split queue

- if Avail.idx Used.idx回环，是否会造成无法区空还是满？

  对 lastAvail， lastUsed 来说，不会因为溢出造成无法区分

  lastAail 追赶Avail.idx, lastUsed追赶Used.idx，但差距仅限于 0 ~ descriptor number

  这四个计数都是16 bits， descriptor table 最多15 bits。

  当溢出的时候，溢出后的值在0 ~ descriptor number之间，不会等于追赶者lastAail或者 lastUsed

In addition, the maximum queue size is 32768 (the highest power of 2 which fits in 16 bits), so the 16-bit idx

value can always distinguish between a full and empty buffer

VIRT-IO protocol -- split queue

VIRT-IO protocol -- split queue

VIRT-IO protocol -- split queue notify

Notify

1，VIRTIO_F_EVENT_IDX 不支持

avail notify: 由driver 决定alail.flags值，0 通知，1 不通知。

used notify: 由device决定used.flags值，0 通知，1 不通知。

2，VIRTIO_F_EVENT_IDX 支持

avail notify: 仅driver avail idx == used.avail_event值, 通知。

used notify：仅device used idx == avail.avail_event值, 通知。

VIRT-IO protocol packed queue

```
/* Actual memory layout for this queue. */
                    struct {
                            unsigned int num;
                            struct vring_packed_desc *desc;
                            struct vring_packed_desc_event *driver;
                            struct vring_packed_desc_event *device;
                    } vring;
```

VIRT-IO protocol packed queue

```
struct vring_packed_desc {
        /* Buffer Address. */
        __le64 addr;
        /* Buffer Length. */
        __le32 len;
        /* Buffer ID. */
        __le16 id;
        /* The flags depending on descriptor type. */
        __le16 flags;
};
```

VIRT-IO protocol packed queue

```
struct vring_packed_desc {
...
        __le16 flags;
...
};

/* This marks a buffer as continuing via the next field. */
#define VRING_DESC_F_NEXT       1
/* This marks a buffer as write-only (otherwise read-only). */
#define VRING_DESC_F_WRITE      2
/* This means the buffer contains a list of buffer descriptors. */
#define VRING_DESC_F_INDIRECT   4
#define VRING_PACKED_DESC_F_AVAIL       7
#define VRING_PACKED_DESC_F_USED        15
```

VIRT-IO protocol packed queue

VIRT-IO protocol packed queue --wrap counter

1. 计数索引关系

VMDriver.last_used_idx <= Device.next_used_idx <= Device.next_avail_idx <= VMDriver.next_avail_idx ：仅体现领先性

Device变量不用VM Driver 比较， 各自判断descriptor的flag标志位。

2. idx反转问题：

Dev and VM: 各自拥有一套 avail_wrap_counter, used_wrap_counter; 初始值为1， 他决定了VIRTQ_DESC_F_USED， VIRTQ_DESC_F_AVAIL是0有效，还是1有效。 wrap counter 为1是flag置1有效，wrap counter 为0， flag置0有效

每当反转的时候就取反，wrap couner 就是1变0，或者0变1.

wrap counter优势：VM driver 回收的时候不必再clear flag，因为反转了，所以A|U自然失效了。由于free_num限制VMDriver.next_avail_idx，所以不用担心会覆盖未回收的descriptor。

VMDriver.last_used_idx ，Device.next_used_idx，Device.next_avail_idx ，VMDriver.next_avail_idx 四个counter最多差一轮计数，wrap counter不会有覆盖的问题。

VIRT-IO protocol packed queue --scatter

VIRT-IO protocol packed queue --scatter

VIRT-IO protocol -- packed queue notify

```
struct pvirtq_event_suppress {
        le16 {
                desc_event_off : 15; /* Descriptor Ring Change Event Offset */
                desc_event_wrap : 1; /* Descriptor Ring Change Event Wrap Counter */
        } desc; /* If desc_event_flags set to RING_EVENT_FLAGS_DESC */

        le16 {
                desc_event_flags : 2, /* Descriptor Ring Change Event Flags */
                reserved : 14; /* Reserved, set to 0 */
        } flags;
};
#define RING_EVENT_FLAGS_ENABLE 0x0
#define RING_EVENT_FLAGS_DISABLE 0x1
* Enable events for a specific descriptor
* (as specified by Descriptor Ring Change Event Offset/Wrap Counter).
* Only valid if VIRTIO_F_RING_EVENT_IDX has been negotiated.
*/
#define RING_EVENT_FLAGS_DESC 0x2
```

VIRT-IO protocol -- packed queue notify

Descriptor Ring Change Event Offset If Event Flags set to descriptor specific event: offset within the

ring (in units of descriptor size). Event will only trigger when this descriptor is made available/used

respectively.

Descriptor Ring Change Event Wrap Counter If Event Flags set to descriptor specific event: offset within

the ring (in units of descriptor size). Event will only trigger when Ring Wrap Counter matches this value

and a descriptor is made available/used respectively

VIRT-IO protocol packed split compare

\type	Split Queue	Packed Queue
max size	32768	32768
size	2^n (0<=n <=15)	1 ~ 2^15
memory size	more	little could choice precise size
cache performance	low	high
complex	low	high

VIRT-IO protocol: pci configuration

VIRT-IO protocol: pci configuration

```
// Avail area
// Used area
```

VIRT-IO protocol: notify

VIRT-IO protocol: notify

VIRT-IO protocol: notify

VIRT-IO protocol: net card

VIRT-IO protocol: net card packet pre-header

Packets are transmitted by placing them in the transmitq1. . .transmitqN, and buffers for incoming packets are placed in the receiveq1. . .receiveqN. In each case, the packet itself is preceded by a header:

```
struct virtio_net_hdr {

#define VIRTIO_NET_HDR_F_NEEDS_CSUM

#define VIRTIO_NET_HDR_F_DATA_VALID

#define VIRTIO_NET_HDR_F_RSC_INFO

u8 flags;

#define VIRTIO_NET_HDR_GSO_NONE#define VIRTIO_NET_HDR_GSO_TCPV4

#define VIRTIO_NET_HDR_GSO_UDP

#define VIRTIO_NET_HDR_GSO_TCPV6

#define VIRTIO_NET_HDR_GSO_ECN

u8 gso_type;

le16 hdr_len;

le16 gso_size;le16 csum_start;

le16 csum_offset;

le16 num_buffers;

};
```

The controlq is used to control device features such as filtering.

VIRT-IO protocol: net card : Packet Transmission

5.1.6.2

Packet Transmission

Transmitting a single packet is simple, but varies depending on the different features the driver negotiated.

1. The driver can send a completely checksummed packet. In this case, flags will be zero, and gso_type will be VIRTIO_NET_HDR_GSO_NONE.

2. If the driver negotiated VIRTIO_NET_F_CSUM, it can skip checksumming the packet:

• flags has the VIRTIO_NET_HDR_F_NEEDS_CSUM set,

• csum_start is set to the offset within the packet to begin checksumming, and

• csum_offset indicates how many bytes after the csum_start the new (16 bit ones' complement) checksum is placed by the device.

• The TCP checksum field in the packet is set to the sum of the TCP pseudo header, so that replacing it by the ones' complement checksum of the TCP header and body will give the correct result.

Note: For example, consider a partially checksummed TCP (IPv4) packet. It will have a 14 byte ether-net header and 20 byte IP header followed by the TCP header (with the TCP checksum field 16 bytes into that header). csum_start will be 14+20 = 34 (the TCP checksum includes the header), and csum_offset will be 16.

VIRT-IO protocol: net card : Packet Transmission

3. If the driver negotiated VIRTIO_NET_F_HOST_TSO4, TSO6 or UFO, and the packet requires TCP segmentation or UDP fragmentation, then gso_type is set to VIRTIO_NET_HDR_GSO_TCPV4, TCPV6 or UDP. (Otherwise, it is set to VIRTIO_NET_HDR_GSO_NONE). In this case, packets larger than 1514 bytes can be transmitted: the metadata indicates how to replicate the packet header to cut it into smaller packets. The other gso fields are set:

• hdr_len is a hint to the device as to how much of the header needs to be kept to copy into each packet, usually set to the length of the headers, including the transport header 1 .

• gso_size is the maximum size of each packet beyond that header (ie. MSS).

• If the driver negotiated the VIRTIO_NET_F_HOST_ECN feature, the VIRTIO_NET_HDR_GSO_-ECN bit in gso_type indicates that the TCP packet has the ECN bit set 2 .

4. num_buffers is set to zero. This field is unused on transmitted packets.

5. The header and packet are added as one output descriptor to the transmitq, and the device is notified of the new entry (see 5.1.5 Device Initialization).

VIRT-IO protocol: net card : Incoming Packets

When a packet is copied into a buffer in the receiveq, the optimal path is to disable further used buffer notifications for the receiveq and process packets until no more are found, then re-enable them.

Processing incoming packets involves:

1. num_buffers indicates how many descriptors this packet is spread over (including this one): this will always be 1 if VIRTIO_NET_F_MRG_RXBUF was not negotiated. This allows receipt of large packets without having to allocate large buffers: a packet that does not fit in a single buffer can flow over to the next buffer, and so on. In this case, there will be at least num_buffers used buffers in the virtqueue, and the device chains them together to form a single packet in a way similar to how it would store it in a single buffer spread over multiple descriptors. The other buffers will not begin with a struct virtio_net_hdr.
2. If num_buffers is one, then the entire packet will be contained within this buffer, immediately following the struct virtio_net_hdr.
3. If the VIRTIO_NET_F_GUEST_CSUM feature was negotiated, the VIRTIO_NET_HDR_F_DATA_VALID bit in flags can be set: if so, device has validated the packet checksum. In case of multiple encapsulated protocols, one level of checksums has been validated.

VIRT-IO protocol: net card : Incoming Packets

When a packet is copied into a buffer in the receiveq, the optimal path is to disable further used buffer notifications for the receiveq and process packets until no more are found, then re-enable them.
Processing incoming packets involves:
1. num_buffers indicates how many descriptors this packet is spread over (including this one): this will always be 1 if VIRTIO_NET_F_MRG_RXBUF was not negotiated. This allows receipt of large packets without having to allocate large buffers: a packet that does not fit in a single buffer can flow over to the next buffer, and so on. In this case, there will be at least num_buffers used buffers in the virtqueue, and the device chains them together to form a single packet in a way similar to how it would store it in a single buffer spread over multiple descriptors. The other buffers will not begin with a struct virtio_net_hdr.
2. If num_buffers is one, then the entire packet will be contained within this buffer, immediately following the struct virtio_net_hdr.
3. If the VIRTIO_NET_F_GUEST_CSUM feature was negotiated, the VIRTIO_NET_HDR_F_DATA_VALID bit in flags can be set: if so, device has validated the packet checksum. In case of multiple encapsulated protocols, one level of checksums has been validated.

VIRT-IO protocol: net card : Incoming Packets

When a packet is copied into a buffer in the receiveq, the optimal path is to disable further used buffer notifications for the receiveq and process packets until no more are found, then re-enable them.
Processing incoming packets involves:
1. num_buffers indicates how many descriptors this packet is spread over (including this one): this will always be 1 if VIRTIO_NET_F_MRG_RXBUF was not negotiated. This allows receipt of large packets without having to allocate large buffers: a packet that does not fit in a single buffer can flow over to the next buffer, and so on. In this case, there will be at least num_buffers used buffers in the virtqueue, and the device chains them together to form a single packet in a way similar to how it would store it in a single buffer spread over multiple descriptors. The other buffers will not begin with a struct virtio_net_hdr.
2. If num_buffers is one, then the entire packet will be contained within this buffer, immediately following the struct virtio_net_hdr.
3. If the VIRTIO_NET_F_GUEST_CSUM feature was negotiated, the VIRTIO_NET_HDR_F_DATA_VALID bit in flags can be set: if so, device has validated the packet checksum. In case of multiple encapsulated protocols, one level of checksums has been validated.

VIRT-IO protocol: net card : Control Queue

VIRT-IO protocol: net card : Control Queue

VIRT-IO protocol: net card : Control Queue

VIRT-IO protocol: net card : Control Queue

VIRT-IO protocol: net card : Control Queue

VIRT-IO protocol: net card : Control Queue

VIRT-IO protocol: net card : Control Queue

VIRT-IO NET & VHOST

https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net#:~:text=virtio%20spec%20-%20The%20virtio%20spec%2C%20which%20is,rings%20layouts%20which%20are%20detailed%20in%20the%20spec.

VIRT-IO vhost-kernel

VIRT-IO notify

- vhost协议可以将允许VMM将virtio的数据面offload到另一个组件上，而这个组件正是vhost-net。在这套实现中，QEMU和vhost-net内核驱动使用ioctl来交换vhost消息，并且用eventfd来实现前后端的通知。当vhost-net内核驱动加载后，它会暴露一个字符设备在/dev/vhost-net。而QEMU会打开并初始化这个字符设备，并调用ioctl来与vhost-net进行控制面通信，其内容包含virtio的特性协商，将虚拟机内存映射传递给vhost-net等。对比最原始的virtio网络实现，控制平面在原有的基础上转变为vhost协议定义的ioctl操作（对于前端而言仍是通过PCI传输层协议暴露的接口），基于共享内存实现的Vring转变为virtio-net与vhost-net共享，数据平面的另一方转变为vhost-net，并且前后端通知方式也转为基于eventfd的实现。

VIRT-IO vhost-user protocol

https://qemu.readthedocs.io/en/latest/interop/vhost-user.html

VIRT-IO vhost-user Doc

Doc:

https://www.redhat.com/en/blog/how-vhost-user-came-being-virtio-networking-and-dpdk#:~:text=In%20the%20vhost-user%2Fvirtio-pmd%20architecture%20virtio%20uses%20DPDK%20both,vhost-user%20module%20are%20additional%20APIs%20inside%20that%20library.

code level : https://www.jianshu.com/p/ae54cb57e608

DPDK site: http://doc.dpdk.org/guides/prog_guide/vhost_lib.html#vhost-user-implementations

VIRT-IO vhost-user DPDK code

- drivers/net/vhost/rte_eth_vhost.c

```
static int
eth_dev_vhost_create(struct rte_vdev_device *dev, char *iface_name,
        int16_t queues, const unsigned int numa_node, uint64_t flags,
        uint64_t disable_flags)
{
...
        /* finally assign rx and tx ops */
        eth_dev->rx_pkt_burst = eth_vhost_rx;
        eth_dev->tx_pkt_burst = eth_vhost_tx;
...
}
```

VIRT-IO vhost-user example

- https://blog.csdn.net/yangye2014/article/details/78064735/

qemu argument: qemu-system-x86_64 -machine accel=kvm -cpu host -smp sockets=2,cores=2,threads=2 -m 2048M -object memory-backend-file,id=mem,size=2048M,mem-path=/dev/hugepages,share=on -drive file=/var/iso/virtual.img -drive file=/opt/share.img,if=virtio -mem-prealloc -numa node,memdev=mem -vnc 0.0.0.0:50 **--enable-kvm -chardev socket,id=char1,path=/tmp/sock0,server -netdev type=vhost-user,id=mynet1,chardev=char1,vhostforce -device virtio-net-pci**,netdev=mynet1,id=net1,mac=00:00:00:00:00:01

ovs:

ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev

ovs-vsctl add-port br0 vhost-user-1 -- set Interface vhost-user-1 type=dpdkvhostuserclient options:**vhost-server-path="/tmp/sock0"**

VIRTIO-HOST Guest driver code -transmit

```
static netdev_tx_t start_xmit(struct sk_buff *skb, struct net_device *dev)
{
...
bool kick = !netdev_xmit_more();
struct netdev_queue *txq = netdev_get_tx_queue(dev, qnum);
...
err = xmit_skb(sq, skb);

...
if (virtqueue_kick_prepare(sq->vq) && virtqueue_notify(sq->vq)) {
...
}

...

}
```

VIRTIO-HOST Guest driver code -recieve

```
static int virtnet_alloc_queues(struct virtnet_info *vi)
{
        for (i = 0; i < vi->max_queue_pairs; i++) {
                netif_napi_add(vi->dev, &vi->rq[i].napi, virtnet_poll,
                               napi_weight);
                netif_tx_napi_add(vi->dev, &vi->sq[i].napi, virtnet_poll_tx,
                                  napi_tx ? napi_weight : 0);
       }
}
```

VIRTIO-HOST Guest driver code -receive

no copy vring to skb

```
skb = build_skb(buf, buflen);
    if (!skb)
            goto err;
    skb_reserve(skb, headroom - delta);
    skb_put(skb, len);
    if (!xdp_prog) {
            buf += header_offset;
            memcpy(skb_vnet_hdr(skb), buf, vi->hdr_len);
    } /* keep zeroed vnet hdr since XDP is loaded */

    if (metasize)
            skb_metadata_set(skb, metasize);
```

IOMMU

- https://wiki.qemu.org/Features/VT-d
- https://www.kernel.org/doc/html/latest/x86/intel-iommu.html
- http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf
- https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- https://www.amd.com/en/support/tech-docs/amd-io-virtualization-technology-iommu-specification

- https://developer.arm.com/architectures/system-architectures/system-components/system-mmu-support
- https://developer.arm.com/documentation/ihi0070/latest

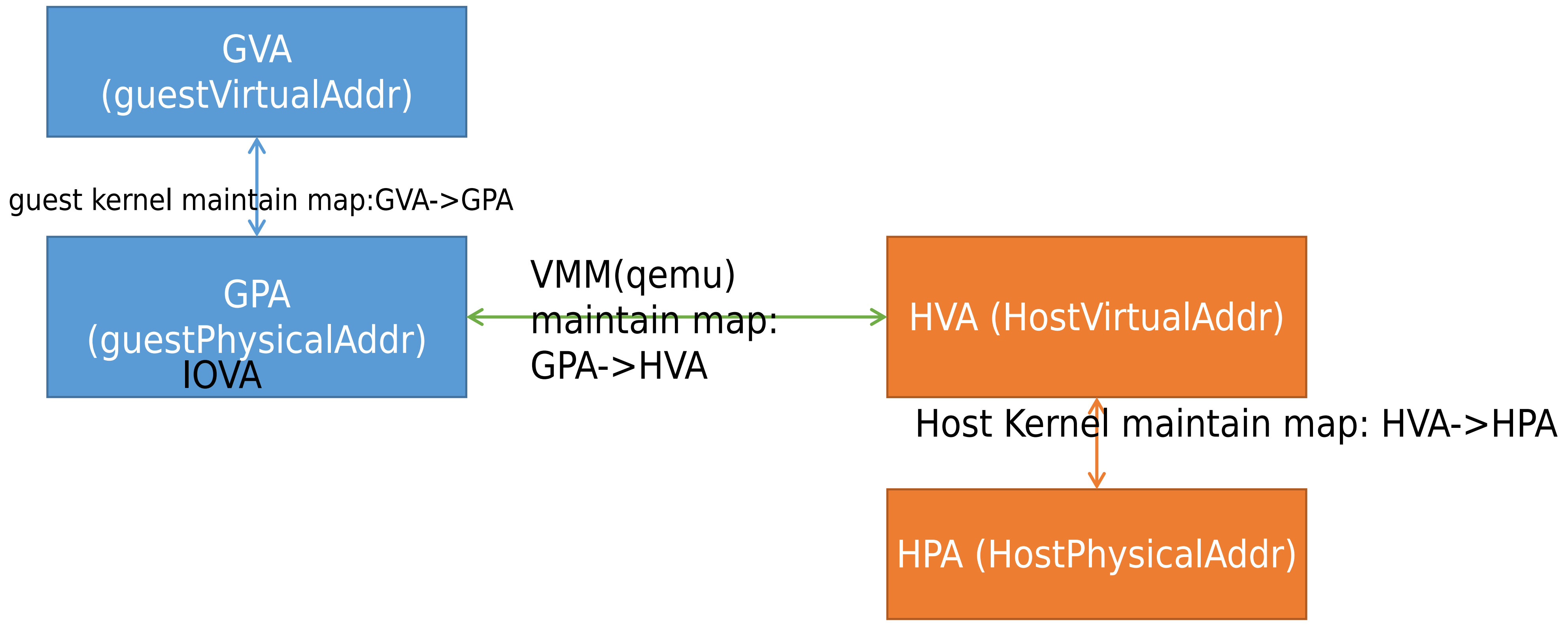
intel IOMMU

Intel ® Virtualization Technology for Directed I/O

intel IOMMU

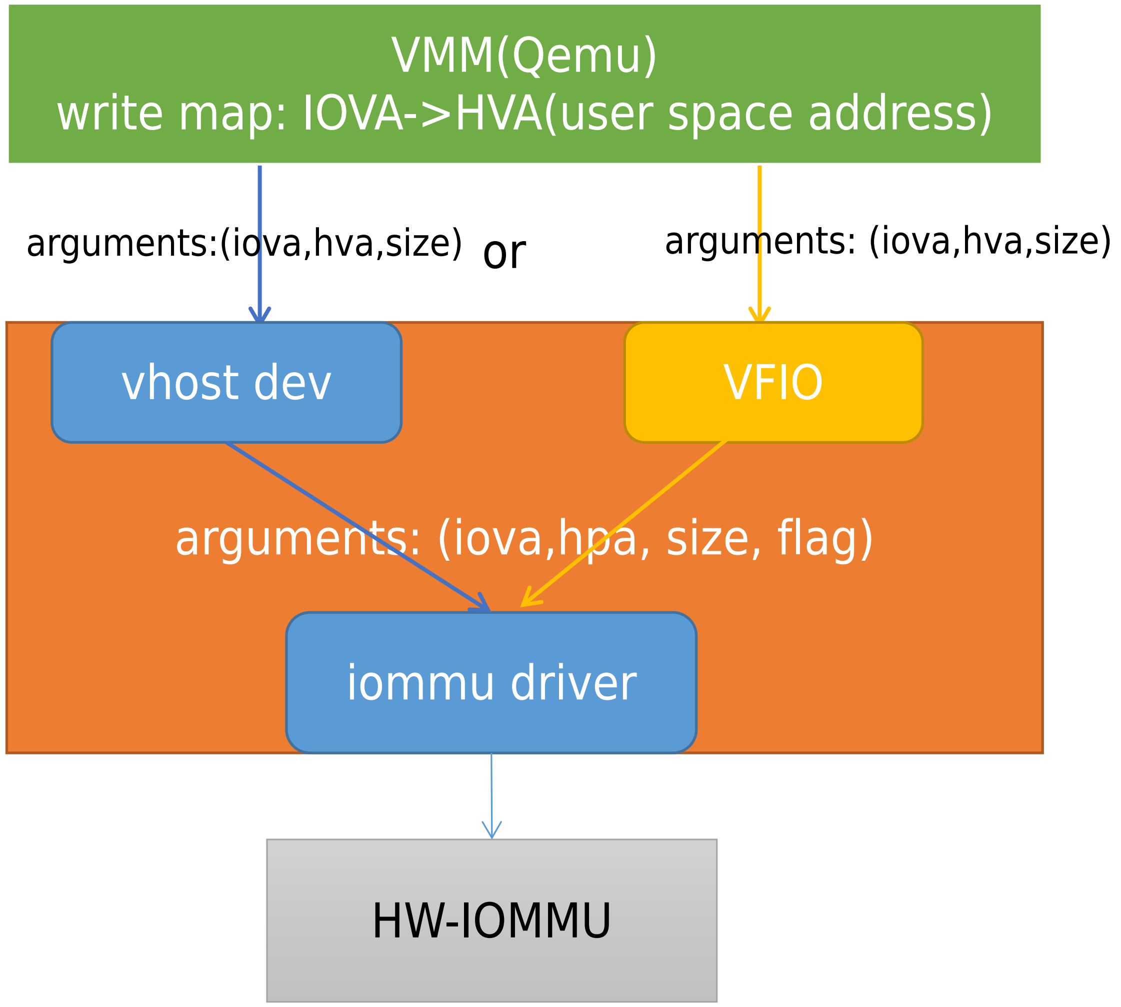
intel IOMMU

VM address translation&IOMMU


GVA (guestVirtualAddr)
guest kernel maintain map:GVA->GPA
GPA (guestPhysicalAddr)
IOVA
VMM(qemu) maintain map: GPA->HVA
HVA (HostVirtualAddr)
Host Kernel maintain map: HVA->HPA
HPA (HostPhysicalAddr)

1.Guest driver write IOVA of buffer to HW decriptor , Generally IOVA is equal to GPA, because VMM could map GPA->HVA->HPA, and finally write (IOVA->HPA) map to IOMMU.

VM address translation&IOMMU


VMM(Qemu)
write map: IOVA->HVA(user space address)
arguments:(iova,hva,size)
or
arguments: (iova,hva,size)
vhost dev
VFIO
arguments: (iova,hpa, size, flag)
iommu driver
HW-IOMMU

VIRT-IO & IOMMU

VIRTIO_F_ACCESS_PLATFORM(33) This feature indicates that the device can be used on a platform where device access to data in memory is limited and/or translated. E.g. this is the case if the device can be located behind an IOMMU that translates bus addresses from the device into physical addresses in memory, if the device can be limited to only access certain memory addresses or if special commands such as a cache flush can be needed to synchronise data in memory with the device. Whether accesses are actually limited or translated is described by platform-specific means. If this feature bit is set to 0, then the device has same access to memory addresses supplied to it as the driver has. In particular, the device will always use physical addresses matching addresses used by the driver (typically meaning physical addresses used by the CPU) and not translated further, and can access any address supplied to it by the driver. When clear, this overrides any platform-specific description of whether device access is limited or translated in any way, e.g. whether an IOMMU may be present.

VIRT-IO & IOMMU

A driver SHOULD accept VIRTIO_F_ACCESS_PLATFORM if it is offered, and it MUST then either disable the IOMMU or configure the IOMMU to translate bus addresses passed to the device into physical

addresses in memory. If VIRTIO_F_ACCESS_PLATFORM is not offered, then a driver MUST pass only physical addresses to the device.

vDPA

vDPA Doc

https://www.redhat.com/en/blog/achieving-network-wirespeed-open-standard-manner-introducing-vdpa

https://www.redhat.com/en/blog/introduction-vdpa-kernel-framework

https://www.redhat.com/en/blog/vdpa-kernel-framework-part-1-vdpa-bus-abstracting-hardware

https://www.redhat.com/en/blog/vdpa-kernel-framework-part-2-vdpa-bus-drivers-kernel-subsystem-interactions

https://www.redhat.com/en/blog/vdpa-kernel-framework-part-3-usage-vms-and-containers

https://www.redhat.com/en/blog/hands-vdpa-what-do-you-do-when-you-aint-got-hardware

https://www.redhat.com/en/blog/how-deep-does-vdpa-rabbit-hole-go

https://www.redhat.com/zh-tw/blog/vdpa-kernel-framework-part-2-vdpa-bus-drivers-kernel-subsystem-interactions

https://lwn.net/Articles/815318/

https://doc.dpdk.org/guides-19.02/nics/ifc.html

vfio: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/driver-api/vfio.rst

VDPA tools

- userspace command: vdpa , it is in iproute2

```
vdpa
Usage: vdpa [ OPTIONS ] OBJECT { COMMAND | help }
where  OBJECT := { mgmtdev | dev }
       OPTIONS := { -V[ersion] | -n[o-nice-names] | -j[son] | -p[retty] | -v[erbose] }
-------------------------------------
vdpa dev help
Usage: vdpa dev show [ DEV ]
       vdpa dev add name NAME mgmtdev MANAGEMENTDEV [ mac MACADDR ] [ mtu MTU ]
       vdpa dev del DEV
Usage: vdpa dev config COMMAND [ OPTIONS ]
-------------------------------------
(p38) jecky@jecky-here:/work/linux-tools/iproute2-5.17.0/vdpa$ vdpa mgmtdev show
vdpasim_net:
  supported_classes net
(p38) jecky@jecky-here:/work/linux-tools/iproute2-5.17.0/vdpa$ sudo vdpa dev add name vdpa0  mgmtdev vdpasim_net
kernel answers: File exists
(p38) jecky@jecky-here:/work/linux-tools/iproute2-5.17.0/vdpa$ sudo vdpa dev add name vdpa1  mgmtdev vdpasim_net
(p38) jecky@jecky-here:/work/linux-tools/iproute2-5.17.0/vdpa$
(p38) jecky@jecky-here:/work/linux-tools/iproute2-5.17.0/vdpa$ vdpa dev show
vdpa0: type network mgmtdev vdpasim_net vendor_id 0 max_vqs 2 max_vq_size 256
vdpa1: type network mgmtdev vdpasim_net vendor_id 0 max_vqs 2 max_vq_size 256
```

VDPA driver

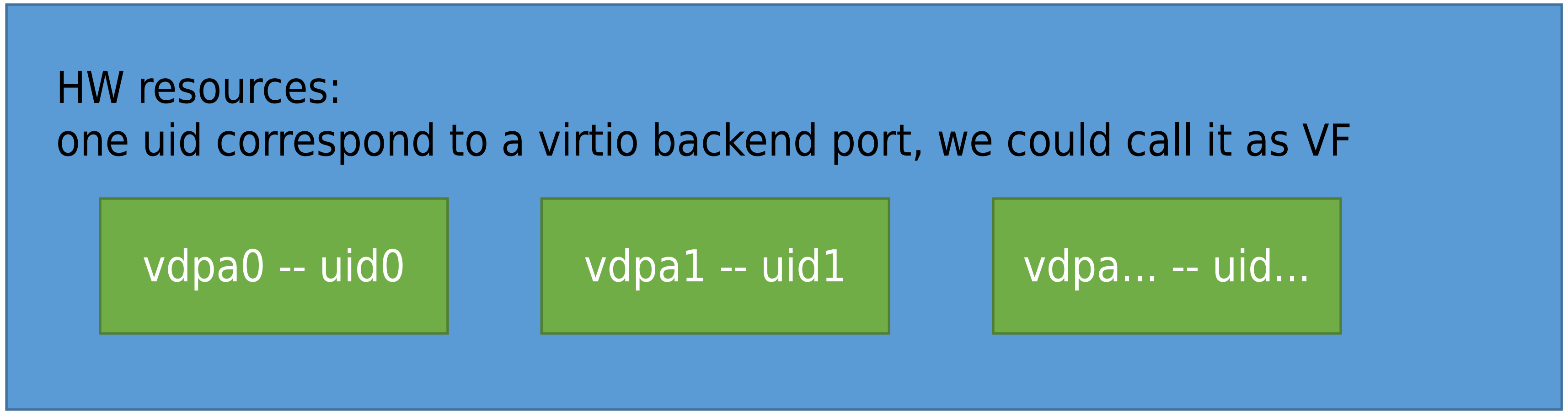
```
/* This driver drives both modern virtio devices and transitional
        * devices in modern mode.
        * vDPA requires feature bit VIRTIO_F_ACCESS_PLATFORM,
        * so legacy devices and transitional devices in legacy
        * mode will not work for vDPA, this driver will not
        * drive devices with legacy interface.
        */
```

You can see that vhost-vdpa bus driver was bound to vdpa0.

The second step is to switch the binding to virtio-vdpa.

```
# echo vdpa0 > /sys/bus/vdpa/drivers/vhost_vdpa/unbind
# echo vdpa0 > /sys/bus/vdpa/drivers/virtio_vdpa/bind
```

VDPA driver mlx5/net/mlx5_vnet.c


HW resources:
one uid correspond to a virtio backend port, we could call it as VF
vdpa0 -- uid0
vdpa1 -- uid1
vdpa... -- uid...

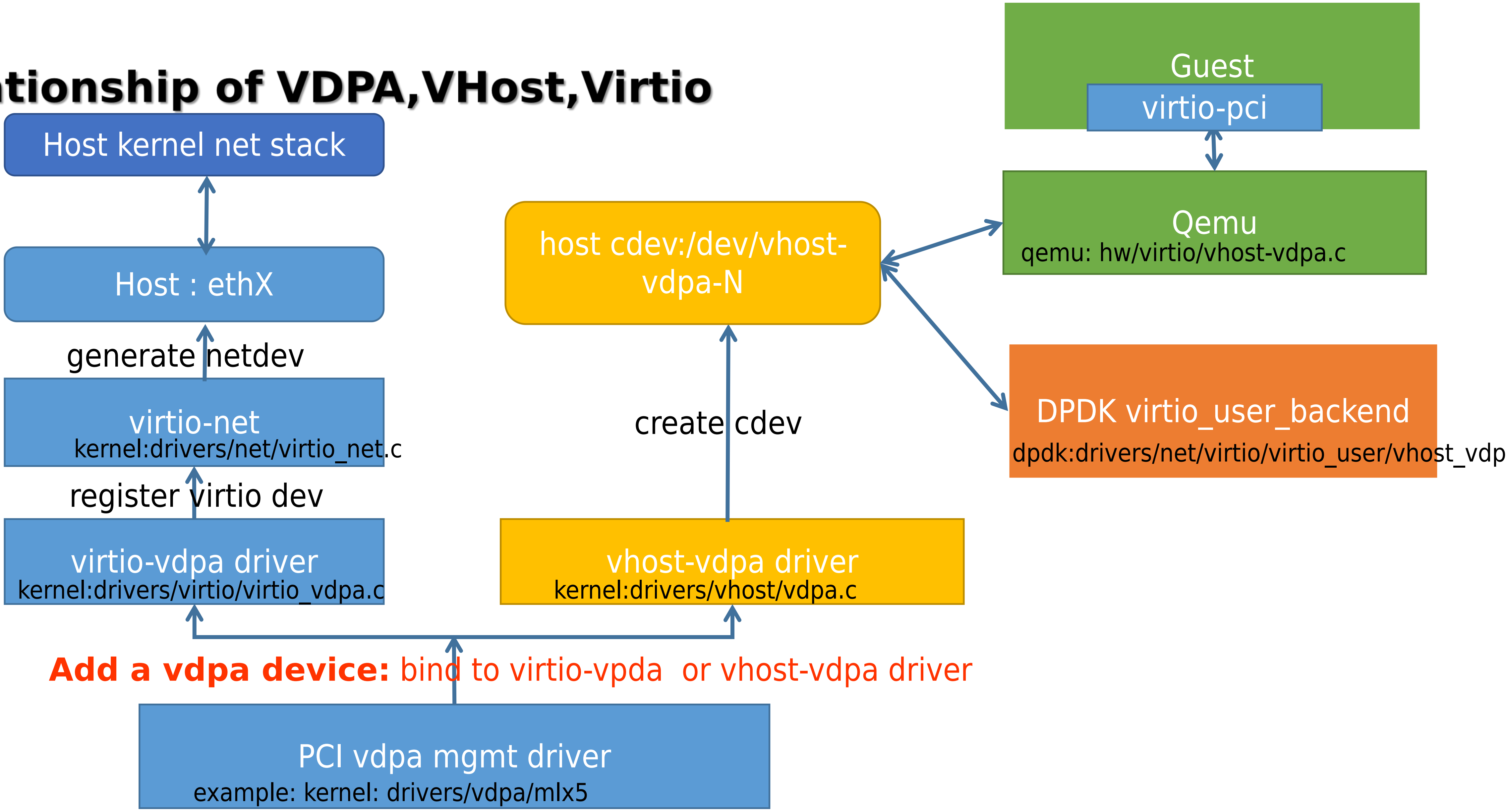
Register interface :

```
struct mlx5_ifc_general_obj_in_cmd_hdr_bits {
        u8      opcode[0x10];
        u8      uid[0x10];
        u8      vhca_tunnel_id[0x10];
        u8      obj_type[0x10];
        u8      obj_id[0x20];
        u8      reserved_at_60[0x20];
};
```

Example:

```
static int create_virtqueue(struct mlx5_vdpa_net *ndev, struct mlx5_vdpa_virtqueue *mvq)
{
...
 MLX5_SET(general_obj_in_cmd_hdr, cmd_hdr, opcode, MLX5_CMD_OP_CREATE_GENERAL_OBJECT);
 MLX5_SET(general_obj_in_cmd_hdr, cmd_hdr, obj_type, MLX5_OBJ_TYPE_VIRTIO_NET_Q);
 MLX5_SET(general_obj_in_cmd_hdr, cmd_hdr, uid, ndev->mvdev.res.uid);
...
}
```

relationship of VDPA,VHost,Virtio


Guest
virtio-pci
Qemu
qemu: hw/virtio/vhost-vdpa.c
DPDK virtio_user_backend
dpdk:drivers/net/virtio/virtio_user/vhost_vdp
Host kernel net stack
Host : ethX
generate netdev
virtio-net
kernel:drivers/net/virtio_net.c
register virtio dev
virtio-vdpa driver
kernel:drivers/virtio/virtio_vdpa.c
host cdev:/dev/vhost-vdpa-N
create cdev
vhost-vdpa driver
kernel:drivers/vhost/vdpa.c
Add a vdpa device: bind to virtio-vpda or vhost-vdpa driver
PCI vdpa mgmt driver
example: kernel: drivers/vdpa/mlx5

Please Read: https://www.redhat.com/en/blog/vdpa-kernel-framework-part-3-usage-vms-and-containers

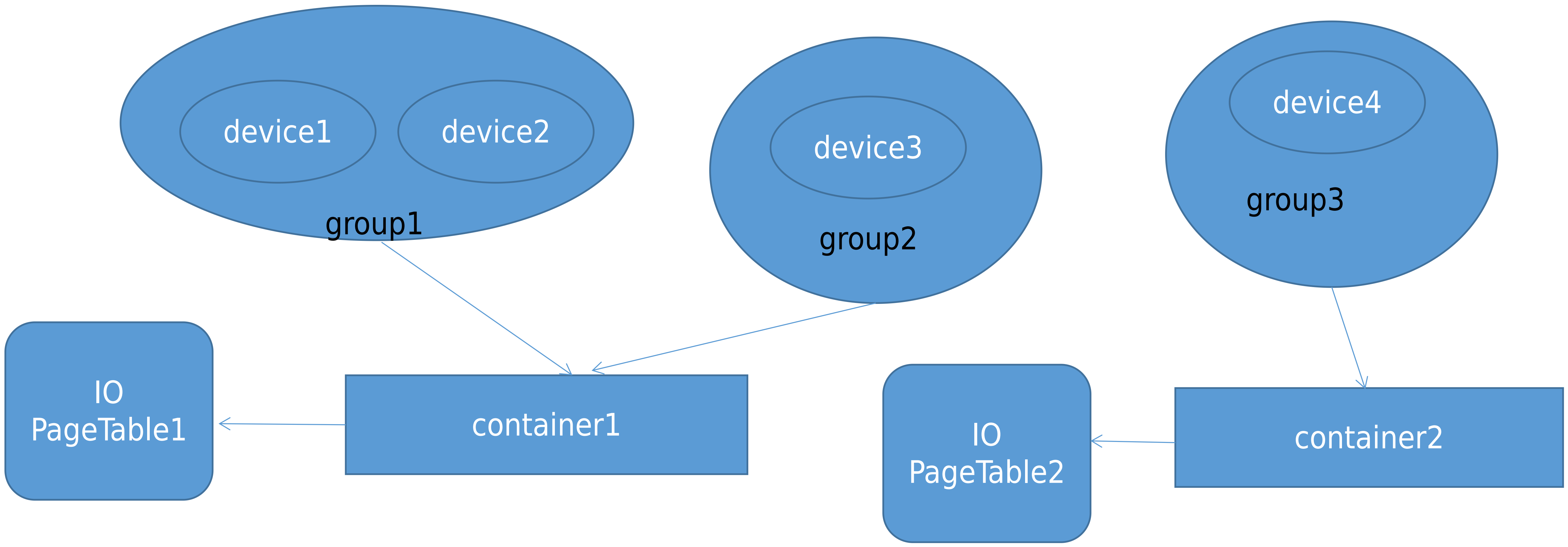
VDPA IOMMU

vendor/driver	on-chip IOMMU	notes
mlx5	yes	advantage: a) support all CPU b) one management pci dev support multi vdpa device , don’t depend on SRIOV
ifcvf	no	a) depend on CPU iommu feture support multi-vpda, should b) depend on SRIOV SRIOV support multiple bus-device-function to support different iommu domain in VM scenario.

VFIO

- https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/driver-api/vfio.rst
- https://www.kernel.org/doc/html/latest/driver-api/vfio.html

VFIO


device1
device2
group1
device3
group2
device4
group3
IO PageTable1
container1
IO PageTable2
container2

VFIO

vfio-containerM
vfio-groupX
share same vfio-container
vfio-containerN
vfio-groupY
vfio-groupZ

VFIO

https://www.cnblogs.com/yi-mu-xi/p/12370626.html

Linux内核设备驱动充分利用了“一切皆文件”的思想，VFIO驱动也不例外，VFIO中为了方便操作device, group, container等对象将它们和对应的设备文件进行绑定。 VFIO驱动在加载的时候会创建一个名为/dev/vfio/vfio的文件，而这个文件的句柄关联到了vfio_container上，用户态进程打开这个文件就可以初始化和访问vfio_container。 当我们把一个设备直通给虚拟机时，首先要做的就是将这个设备从host上进行解绑，即解除host上此设备的驱动，然后将设备驱动绑定为“vfio-pci”， 在完成绑定后会新增一个/dev/vfio/$groupid的文件，其中$groupid为此PCI设备的iommu group id， 这个id号是在操作系统加载iommu driver遍历扫描host上的PCI设备的时候就已经分配好的，可以使用readlink -f /sys/bus/pci/devices/$bdf/iommu_group来查询。 类似的，/dev/vfio/$groupid这个文件的句柄被关联到vfio_group上，用户态进程打开这个文件就可以管理这个iommu group里的设备。 然而VFIO中并没有为每个device单独创建一个文件，而是通过VFIO_GROUP_GET_DEVICE_FD这个ioctl来获取device的句柄，然后再通过这个句柄来管理设备。

VFIO框架中很重要的一部分是要完成DMA Remapping，即为Domain创建对应的IOMMU页表，这个部分是由vfio_iommu_driver来完成的。 vfio_container包含一个指针记录vfio_iommu_driver的信息，在x86上vfio_iommu_driver的具体实现是由vfio_iommu_type1来完成的。 其中包含了vfio_iommu, vfio_domain, vfio_group, vfio_dma等关键数据结构（注意这里是iommu里面的），

- vfio_iommu可以认为是和container概念相对应的iommu数据结构，在虚拟化场景下每个虚拟机的物理地址空间映射到一个vfio_iommu上。
- vfio_group可以认为是和group概念对应的iommu数据结构，它指向一个iommu_group对象，记录了着iommu_group的信息。
- vfio_domain这个概念尤其需要注意，这里绝不能把它理解成一个虚拟机domain，它是一个与DRHD（即IOMMU硬件）相关的概念， 它的出现就是为了应对多IOMMU硬件的场景，我们知道在大规格服务器上可能会有多个IOMMU硬件，不同的IOMMU硬件有可能存在差异， 例如IOMMU 0支持IOMMU_CACHE而IOMMU 1不支持IOMMU_CACHE（当然这种情况少见，大部分平台上硬件功能是具备一致性的），这时候我们不能直接将分别属于不同IOMMU硬件管理的设备直接加入到一个container中， 因为它们的IOMMU页表SNP bit是不一致的。 因此，一种合理的解决办法就是把一个container划分多个vfio_domain，当然在大多数情况下我们只需要一个vfio_domain就足够了。 处在同一个vfio_domain中的设备共享IOMMU页表区域，不同的vfio_domain的页表属性又可以不一致，这样我们就可以支持跨IOMMU硬件的设备直通的混合场景。

VFIO

如果device 是一个硬件拓扑上是独立那么这个设备构成了一个IOMMU group。如果多个设备在硬件是互联的，需要相互访问数据，那么这些设备需要放到一个IOMMU group 中隔离起来。

vfio_group is not configured by software.

it depend on the HW.

VFIO

```
#./usertools/dpdk-devbind.py  --bind=vfio-pci 0000:02:00.0
#readlink  sys/devices/pci0000:00/0000:00:1c.7/0000:02:00.0/iommu_group
../../../../kernel/iommu_groups/24
#find /sys/kernel/iommu_groups/
/sys/kernel/iommu_groups/
/sys/kernel/iommu_groups/55
/sys/kernel/iommu_groups/55/devices
/sys/kernel/iommu_groups/55/devices/0000:90:0c.1
/sys/kernel/iommu_groups/55/type
/sys/kernel/iommu_groups/55/reserved_regions
......
/sys/kernel/iommu_groups/35
```

PCI

- https://blog.csdn.net/u013253075/article/details/119045277

PCI

- https://blog.csdn.net/u013253075/article/details/119045277

PCI

在上图PCIe系统中有几种设备类型，Root Complex、Switch、Bridge、Endpoint等，下面分别介绍其概念。

Root Complex：简称RC，CPU和PCIe总线之间的接口，可能包含几个组件(处理器接口、DRAM接口等)，甚至可能包含几个芯片。RC位于PCI倒立树拓扑的“根”，并代表CPU与系统的其余部分进行通信。但是，规范并没有仔细定义它，而是给出了必需和可选功能的列表。从广义上讲，RC可以理解为系统CPU和PCIe拓扑之间的接口，PCIe端口在配置空间中被标记为“根端口”。

Bridge：桥提供了与其他总线(如PCI或PCI- x，甚至是另一个PCIe总线)的接口。如图中显示的桥接有时被称为“转发桥接”，它允许旧的PCI或PCIX卡插入新系统。相反的类型或“反向桥接”允许一个新的PCIe卡插入一个旧的PCI系统。

Switch：提供扩展或聚合能力，并允许更多的设备连接到一个PCIe端口。它们充当包路由器，根据地址或其他路由信息识别给定包需要走哪条路径。是一种PCIe转PCIe的桥。

Endpoint：处于PCIe总线系统拓扑结构中的最末端，一般作为总线操作的发起者（initiator，类似于PCI总线中的主机）或者终结者（Completers，类似于PCI总线中的从机）。显然，Endpoint只能接受来自上级拓扑的数据包或者向上级拓扑发送数据包。细分Endpoint类型的话，分为Lagacy PCIe Endpoint和Native PCIe Endpoint，Lagacy PCIe Endpoint是指那些原本准备设计为PCI-X总线接口的设备，但是却被改为PCIe接口的设备。而Native PCIe Endpoint则是标准的PCIe设备。其中，Lagacy PCIe Endpoint可以使用一些在Native PCIe Endpoint禁止使用的操作，如IO Space和Locked Request等。Native PCIe Endpoint则全部通过Memory Map来进行操作，因此，Native PCIe Endpoint也被称为Memory Mapped Devices（MMIO Devices）

SRIOV

- https://composter.com.ua/documents/sr-iov1_1_20Jan10_cb.pdf

SRIOV is for virtualization. could be used in VM as passthrough mode.

SRIOV

SRIOV

SRIOV

SRIOV

SRIOV

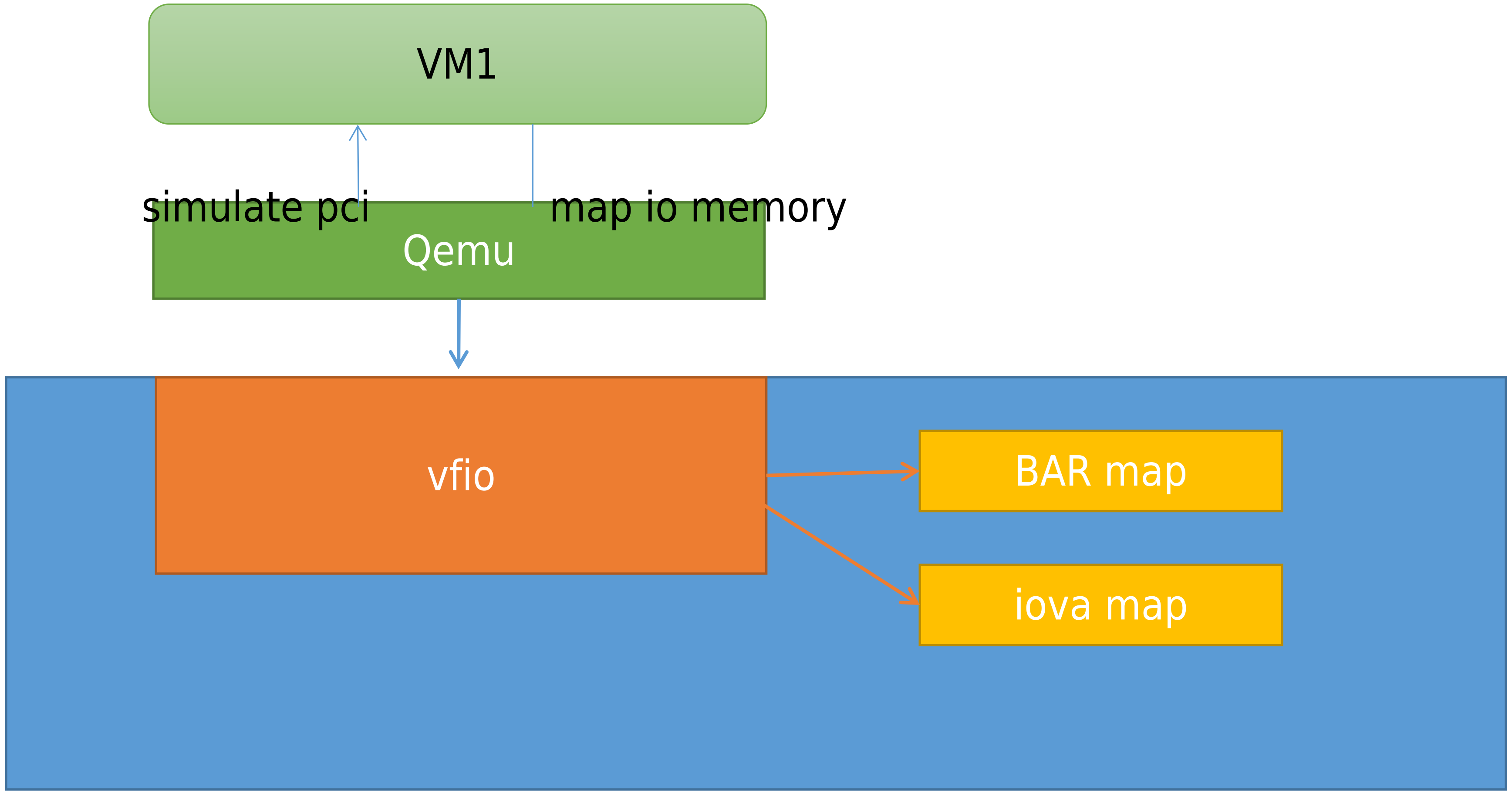
SRIOV

SRIOV

SRIOV VM passthrough

- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/sect-pci_devices-pci_passthrough

SRIOV QEMU



SRIOV VM passthrough

```
...
 <devices>
   ...
   <interface type='hostdev' managed='yes'>
     <source>
       <address type='pci' domain='0' bus='11' slot='16' function='0'/>
     </source>
     <mac address='52:54:00:6d:90:02'>
     <vlan>
       <tag id='42'/>
     </vlan>
     <virtualport type='802.1Qbh'>
       <parameters profileid='finance'/>
     </virtualport>
   </interface>
   ...
 </devices>
virsh attach-device MyGuest /tmp/new-interface.xml --live --config
```

SRIOV VM VF driver example

kernel :

drivers/net/ethernet/intel/igbvf

drivers/net/ethernet/intel/ixgbevf

QEMU simulate PCI

https://blog.csdn.net/weixin_43780260/article/details/104410063

https://www.qemu.org/docs/master/system/target-i386.html

https://www.qemu.org/docs/master/system/i386/pc.html

SIOV

- https://www.opencompute.org/documents/ocp-scalable-io-virtualization-technical-specification-revision-1-v1-2-pdf
- https://wcm-ciqa.intel.com/content/dam/develop/public/us/en/documents/intel-scalable-io-virtualization-technical-specification.pdf

P4

- https://p4.org/
- https://p4.org/specs/

P4 lang : header vs struct

p4 is like C lang .

A header declaration introduces a new identifier in the current scope; the type can be referred to

using this identifier. A header is similar to a struct in C, containing all the specified fields. However,

in addition, a header also contains a hidden Boolean “validity” field. When the “validity” bit is true

we say that the “header is valid”. When a local variable with a header type is declared, its “validity”

bit is automatically set to false . The “validity” bit can be manipulated by using the header methods

isValid() , setValid() , and setInvalid() , as described in Section 8.16.

P4 lang : example

https://opennetworking.org/news-and-events/blog/getting-started-with-p4/

compile: p4c -b bmv2 test.p4 -o test.bmv2

run the p4 example:

sudo simple_switch --interface 0@veth0 --interface 1@veth2 --interface 2@veth4 **test.bmv2/test.json &**

simple_switch is from repo: https://github.com/p4lang/behavioral-model.git

So the HW should understand target json file .

P4 OVS

- https://opennetworking.org/news-and-events/blog/p4-and-open-vswitch/
- https://github.com/osinstom/P4-OvS
- http://www.openvswitch.org//support/slides/p4.pdf

eBPF offload

https://ebpf.io/

eBPF is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in an operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules.

eBPF offload

https://ebpf.io/

eBPF is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in an operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules.

eBPF offload

https://www.kernel.org/doc/html/latest/bpf/instruction-set.html

eBPF has 10 general purpose registers and a read-only frame pointer register, all of which are 64-bits wide.

The eBPF calling convention is defined as:

R0: return value from function calls, and exit value for eBPF programs

R1 - R5: arguments for function calls

R6 - R9: callee saved registers that function calls will preserve

R10: read-only frame pointer to access stack

R0 - R5 are scratch registers and eBPF programs needs to spill/fill them if necessary across calls.

eBPF offload

conclusion:

- performance perspective : register number is limited to 11. eBPF should not be the best choice for H/W Offload.
- ecosystem perspective : Maybe, eBPF is a good choice for current ebpf applications(specially in k8s).