邮箱：zhunxun@gmail.com

< 2020年4月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

搜索

找找看

谷歌搜索

PostCategories

C语言(2)
IO Virtualization(3)
KVM虚拟化技术(26)
linux 内核源码分析(61)
Linux日常应用(3)
linux时间子系统(3)
qemu(10)
seLinux(1)
windows内核(5)
调试技巧(2)
内存管理(8)
日常技能(3)
容器技术(2)
生活杂谈(1)
网络(5)
文件系统(4)
硬件(4)

PostArchives

2018/4(1)
2018/2(1)
2018/1(3)
2017/12(2)
2017/11(4)
2017/9(3)
2017/8(1)
2017/7(8)
2017/6(6)
2017/5(9)
2017/4(15)
2017/3(5)
2017/2(1)
2016/12(1)
2016/11(11)
2016/10(8)
2016/9(13)

ArticleCategories

时态分析(1)

Recent Comments

1. Re:virtio前端驱动详解
我看了下，Linux-4.18.2中的vp_notify()函数。bool vp_notify(struct virtqueue *vq){ /* we write the queue's selec...
--Linux-inside
2. Re:virtIO之VHOST工作原理简析

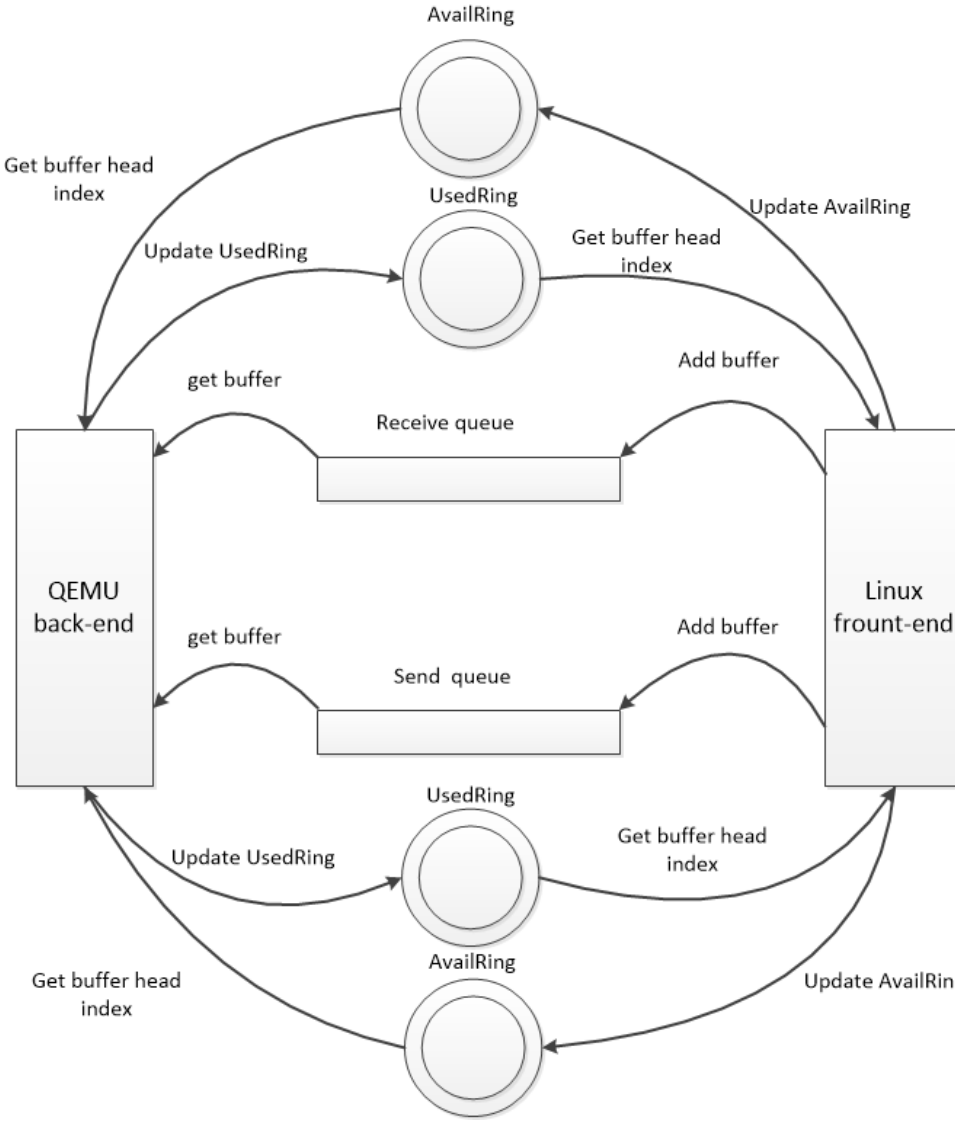
virtIO前后端notify机制详解

2016-11-15

本来这是在前端驱动后期分析的，但是这部分内容比较多，且分析了后端notify前端的机制，所以还是单独拿出一节分析比较好！

还是拿网络驱动部分做案例，网络驱动部分有两个队列，（忽略控制队列）：接收队列和发送队列；每个队列都对应一个virtqueue,两个队列之间是互不影响的。

前后端利用virtqueue的方式如下图所示：



这里再详细的描述下，当两个queue都需要客户机填充buffer，ReceiveQueue需要客户机 driver提前填充分配好的空buffer，然后记录到availRing，并在**恰当的时机**通知后端,当外部网络有数据包到达时，qemu后端就从availRing 中获取一个buffer，然后填充数据，完事后记录buffer head index到usedRing.最后在**恰当的时机**通知客户机（向客户机注入中断），客户机接收到信号便知道有数据包到达，这里只需要从usedRing 中获取到index，然后取data数组的第i个元素即可。因为在客户机填充buffer的时候把逻辑buffer的指针保存在data数组中。

而SendQueue同样需要客户机去填充，只不过这里是当客户机需要发送数据包时，把数据包构造成逻辑buffer，然后填充到send Queue,并在**恰当的时机**通知后端，qemu后端收到通知就知道那个队列有请求到达，如果当前没有处理其他数据包就着手处理这个数据包。具体就同样是从AvailRing中取出buffer head index，然后从描述符表中get到buffer，这时就需要从buffer中copy数据了，因为要把数据包从

再问一个问题，从设置ioeventfd那个流程来看的话是guest发起一个IO，首先会陷入到kvm中，然后由kvm向qemu发送一个IO到来的event，最后IO才被处理，是这样的吗?

--Linux-inside

3. Re:virtIO之VHOST工作原理简析

你好。设置ioeventfd这个部分和guest里面的virtio前端驱动有关系吗?
设置ioeventfd和virtio前端驱动是如何发生联系起来的？谢谢。

--Linux-inside

4. Re:QEMU IO事件处理框架

良心博主，怎么停跟了，太可惜了。

--黄铁牛

5. Re:linux 逆向映射机制浅析

小哥哥520脱单了么

--黄铁牛

Top Posts

1. 详解操作系统中断(21053)
2. PCI 设备详解一(15734)
3. 进程的挂起、阻塞和睡眠(13613)
4. Linux下桥接模式详解一(13391)
5. virtio后端驱动详解(10508)

推荐排行榜

1. 进程的挂起、阻塞和睡眠(6)
2. qemu-kvm内存虚拟化1(2)
3. 为何要写博客(2)
4. virtIO前后端notify机制详解(2)
5. 详解操作系统中断(2)

host发送出去，然后更新usedRing。最后同样要在**恰当的时机**通知客户机。注意这里客户机同样需要从usedRing 中get index，但是这里主要是用于delay notify,因为数据包由客户机构造，其占用的buffer并不能重复使用，只是每次有数据包就把其构造成buffer而已。

以上便是基本的使用sendqueue和receive的原理，但是还有一点上面我没有提到，就是通知的那个**恰当的时机，**那么这个恰当的实际究竟是什么时候呢？？在virtIO中有两种方式控制前后端的notify.

1、flags字段

2、事件触发

1、在vring_avail和vring_used的flags字段，控制前后端的通信。vring_used中的flags用于通知driver端，当add一个buffer的时候不用notify后端。而vring_avail中的flags用于通知qemu端，当消费一个buffer的时候不用interrupt 客户机。

2、在virtIO中又加入了另一种机制，需要由driver和qemu自己判断是否需要通知，也就是设置一个限额，当一端添加buffer或者消费buffer的数量达到指定数目，就触发事件，从而发生notify或者interrupt。在有这种机制的情况下就忽略了前面所说的flags。

这里我们以receiveQueue为例，分析下前后端的delay notify机制。

在front driver端:

客户机driver通过NAPI接收数据时，会在可用buffer不足的时候调用函数添加，具体就是try_fill_recv：

```
static bool try_fill_recv(struct receive_queue *rq, gfp_t gfp)
{
    struct virtnet_info *vi = rq->vq->vdev->priv;
    int err;
    bool oom;
    /*循环，每循环一次添加一个buffer，一直到填充满，即描述符表满*/
    do {
        if (vi->mergeable_rx_bufs)
            err = add_recvbuf_mergeable(rq, gfp);
        else if (vi->big_packets)
            err = add_recvbuf_big(rq, gfp);
        else
            err = add_recvbuf_small(rq, gfp);
        oom = err == -ENOMEM;
        if (err)
            break;
        ++rq->num;
    } while (rq->vq->num_free);
    if (unlikely(rq->num > rq->max))
        rq->max = rq->num;
    /*通知后端*/
    virtqueue_kick(rq->vq);
    return !oom;
}
```

至于添加的是哪种类型的buffer，我们这里并不关心，循环结束就调用virtqueue_kick(rq->vq)函数，此时参数是接收队列的virtqueue,

接下来就调用到了virtqueue_kick_prepare函数，该函数判断当前应不应该通知后端。先看下函数的代码：

```
 1 bool virtqueue_kick_prepare(struct virtqueue *_vq)
 2 {
 3     struct vring_virtqueue *vq = to_vvq(_vq);
 4     u16 new, old;
 5     bool needs_kick;
 6
 7     START_USE(vq);
 8     /* We need to expose available array entries before checking avail
 9      * event. */
10     virtio_mb(vq->weak_barriers);
11
12     old = vq->vring.avail->idx - vq->num_added;
13     new = vq->vring.avail->idx;
14     vq->num_added = 0;
15
16 #ifdef DEBUG
17     if (vq->last_add_time_valid) {
18         WARN_ON(ktime_to_ms(ktime_sub(ktime_get(),
19                             vq->last_add_time)) > 100);
```

```
20      }
21      vq->last_add_time_valid = false;
22  #endif
23
24      if (vq->event) {
25          needs_kick = vring_need_event(vring_avail_event(&vq->vring),
26                                        new, old);
27      } else {
28          needs_kick = !(vq->vring.used->flags & VRING_USED_F_NO_NOTIFY);
29      }
30      END_USE(vq);
31      return needs_kick;
```

这里面涉及到几个变量，old是add_sg之前的avail.idx，而new是当前的avail.idx，还有一个是vring_avail_event(&vq->vring)，看具体的实现：

```
1 #define vring_avail_event(vr) (*(__u16 *)&(vr)->used->ring[(vr)->num])
```

可以看到这里是VRingUsed中的ring数组最后一项的值，该值在后端驱动从virtqueue中pop一个elem之前设置成相应队列的下一个将要使用的index,即last_avail_index。

看下vring_need_event函数：

```
1 static inline int vring_need_event(__u16 event_idx, __u16 new_idx, __u16 old)
2 {
3      /* Note: Xen has similar logic for notification hold-off
4       * in include/xen/interface/io/ring.h with req_event and req_prod
5       * corresponding to event_idx + 1 and new_idx respectively.
6       * Note also that req_event and req_prod in Xen start at 1,
7       * event indexes in virtio start at 0. */
8      return (__u16)(new_idx - event_idx - 1) < (__u16)(new_idx - old);
9 }
```

前后端通过对比 (__u16)(new_idx - event_idx - 1) < (__u16)(new_idx - old)来判断是否需要notify后端，这在数据量比较大的时候显得很实用。在初始状态下，即在qemu一个buffer还没有使用的情况下，event_idx必然是0，那么此时这里的判断肯定为真，所以notify后端。后端收到通知就从virtqueue中pop buffer,同时在此之前需要设置event_idx,代码见qemu virtio.c的virtqueue_pop函数：

```
 1 void *virtqueue_pop(VirtQueue *vq, size_t sz)
 2 {
 3      ......
 4
 5      i = head = virtqueue_get_head(vq, vq->last_avail_idx++);
 6      if (virtio_vdev_has_feature(vdev, VIRTIO_RING_F_EVENT_IDX))     {
 7          vring_set_avail_event(vq, vq->last_avail_idx);
 8      }
10      ......
11 }
```

如果是初始化状态，即当前是首次执行virtqueue_pop函数，last_avail_idx=0，在++后就成了1，然后设置此值到UsedRing.ring[]数组的最后一项：

```
1 static inline void vring_set_avail_event(VirtQueue *vq, uint16_t val)
2 {
3      hwaddr pa;
4      if (!vq->notification) {
5          return;
6      }
7      pa = vq->vring.used + offsetof(VRingUsed, ring[vq->vring.num]);
8      virtio_stw_phys(vq->vdev, pa, val);
9 }
```

设置成功后就执行pop之后的处理，写入数据完成后，调用后端的 virtio_notify(vdev, q->rx_vq)函数。该函数执行前同样需要判断是否需要notify,具体函数为virtio_should_notify

```
bool virtio_should_notify(VirtIODevice *vdev, VirtQueue *vq)
{
    uint16_t old, new;
    bool v;
    /* We need to expose used array entries before checking used event. */
    smp_mb();
    /* Always notify when queue is empty (when feature acknowledge) */
    if (virtio_vdev_has_feature(vdev, VIRTIO_F_NOTIFY_ON_EMPTY) &&
        !vq->inuse && virtio_queue_empty(vq)) {
        return true;
    }

    if (!virtio_vdev_has_feature(vdev, VIRTIO_RING_F_EVENT_IDX)) {
        return !(vring_avail_flags(vq) & VRING_AVAIL_F_NO_INTERRUPT);
    }

    v = vq->signalled_used_valid;
    vq->signalled_used_valid = true;
    old = vq->signalled_used;
    new = vq->signalled_used = vq->used_idx;
    return !v || vring_need_event(vring_get_used_event(vq), new, old);
}
```

该函数逻辑和前端driver总的判断函数大致类似，但是还是有些不同，首先，如果队列为空即当前没有可用buffer了，那么必然会notify前端；

接着判断是否支持这样事件触发式的方式即VIRTIO_RING_F_EVENT_IDX，如果不支持，就通过flags字段来判断。而如果支持，就通过事件触发来通知。

这里有两个条件：第一个是v = vq->signalled_used_valid和 vring_need_event(vring_get_used_event(vq), new, old)

v = vq->signalled_used_valid在初始化的时候被设置成false，表示还没有向前端做任何通知，而后每次的virtio_should_notify中就会设置成true，并更新vq->signalled_used = vq->used_idx；所以如果是首次尝试通知前端，则总能成功，否则需要判断 vring_need_event(vring_get_used_event(vq), new, old)，该函数具体是根前面逻辑是一样的，正如前面所说，这是第一次尝试通知，所以总能成功。而vring_get_used_event(vq)是 **VRingAvail.ring[]**数组的最后一项的值，该值在客户机driver中被设置

在次回到linux driver中，就会从usedRing中取buffer，同样每取出一个buffer就会设置used_event，代码见virtio_ring.c的virtqueue_get_buf函数,设置的值是vq->last_used_idx，记录客户机处理位置。

```
1 void *virtqueue_get_buf(struct virtqueue *_vq, unsigned int *len)
2 {
3
4     ......
5     if (!(vq->vring.avail->flags & VRING_AVAIL_F_NO_INTERRUPT)) {
6         vring_used_event(&vq->vring) = vq->last_used_idx;
7         virtio_mb(vq->weak_barriers);
8     }
9     ......
10
11 }
```

到目前为止，基本一次完整的交互已经完成了，但是由于是初次交互，前后端的delay机制都没起作用，判断条件中使用到的event_idx已经更新了，假如说首次add 8个buffer，然后通知了后端，并且后端使用了三个buffer并首次notify前端，此时 后端向第4个buffer中写数据，last_avail_idx=4（从0开始），那么used_event=4，此时前端发现可用buffer不足，需要添加，那么本次添加了5个，即new=8+5=13，old=8,new-old=5,而此时new-used_event-1=8,条件不满足，所以此时前端driver添加的buffer就不用notify后端。而话说这段时间后端又处理好了第二个数据包，使用了3个buffer。但不幸，前端还在处理第二个buffer，即last_used_idx=2,则used_event=2;对于后端来讲new-old=3,new-used_event-1=3,条件不满足，所以也不用通知。这样delay notify的机制便显示出效果了。笔者认为这其实本质上就是一场速度的对决，为了保证公平，即使一方处理快，也不能任意向另一端发送数据，只能待对方处理的差不多了你才能发，这样发送一方可以歇歇，而接受一方也不会因为处理不及而丢弃，从而造成浪费！哈哈，真是无规矩不成方圆！

具体通知方式：

前面已经提到前端或者后端完成某个操作需要通知另一端的时候需要某种notify机制。这个notify机制是啥呢？这里分为两个方向

1、guest->host

前面也已经介绍，当前端想通知后端时，会调用virtqueue_kick函数，继而调用virtqueue_notify，对应virtqueue结构中的notify函数，在初始化的时候被初始化成vp_notify（virtio_pci.c中），看下该函数的实现

```
static void vp_notify(struct virtqueue *vq)
{
    struct virtio_pci_device *vp_dev = to_vp_device(vq->vdev);

    /* we write the queue's selector into the notification register to
     * signal the other end */
    iowrite16(vq->index, vp_dev->ioaddr + VIRTIO_PCI_QUEUE_NOTIFY);
}
```

可以看到这里仅仅是吧vq的index编号写入到设备的IO地址空间中，实际上就是设备对应的PCI配置空间中VIRTIO_PCI_QUEUE_NOTIFY位置。这里执行IO操作会引发VM-exit，继而退出到KVM->qemu中处理。看下后端驱动的处理方式。在qemu代码中virtio-pci.c文件中有函数virtio_ioport_write专门处理前端驱动的IO写操作，看

```
case VIRTIO_PCI_QUEUE_NOTIFY:
        if (val < VIRTIO_PCI_QUEUE_MAX) {
            virtio_queue_notify(vdev, val);
        }
        break;
```

这里首先判断队列号是否在合法范围内，然后调用virtio_queue_notify函数，而最终会调用到virtio_queue_notify_vq，该函数其实仅仅调用了VirtQueue结构中绑定的处理函数handle_output，该函数根据不同的设备有不同的实现，比如网卡有网卡的实现，而块设备有块设备的实现。以网卡为例看看创建VirtQueue的时候给绑定的是哪个函数。在virtio-net,c中的virtio_net_init，可以看到这里给接收队列绑定的是virtio_net_handle_rx，而给发送队列绑定的是virtio_net_handle_tx_bh或者virtio_net_handle_tx_timer。而对于块设备则对应的是virtio_blk_handle_output函数。

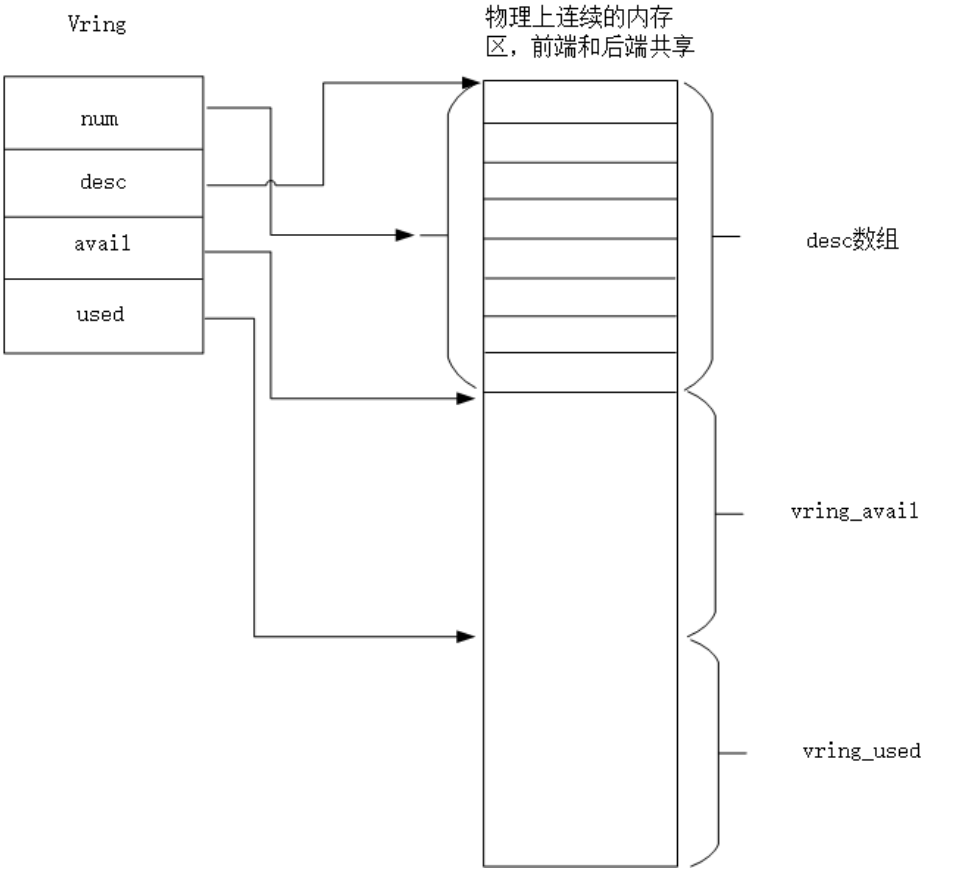
2、host->guest

host通知guest当然是通过注入中断的方式，首先调用的是virtio_notify，继而调用virtio_notify_vector并把中断向量作为参数传递进去。这里就调用了设备关联的notify函数，具体实现为virtio_pci_notify函数，常规中断（非MSI）会调用qemu_set_irq，在8259a中断控制器的情况下回调用kvm_pic_set_irq，然后到了kvm_set_irq，这里就会通过kvm_vm_ioctl和KVM交互，接口为KVM_IRQ_LINE，通知KVM对guest进行中断的注入。KVm里的kvm_vm_ioctl函数会对此调用进行处理，具体就是调用kvm_vm_ioctl_irq_line，之后就调用kvm_set_irq函数进行注入了。之后的流程参看中断虚拟化部分。

共享内存

前面提到，在guest通知host的时候，是把队列的索引写入到了配置空间的VIRTIO_PCI_QUEUE_NOTIFY字段，但是仅仅一个索引是怎么找到指定的队列，且数据时什么时候到达后端的呢？这就用到了共享内存。我们知道的是前后端的确通过共享内存的方式传递数据，但是数据的地址是怎么传递到后端的，这是个问题。本小节主要分析下这个问题。

为了便于理解我们先阐述其原理，然后结合代码看具体的实现。实际上前后端在初始化后就共享了一段连续的内存区，注意这里是物理上连续的内存区（GPA），由客户机内部初始化队列的时候分配，所以这里就是需要和伙伴系统交互。这段内存区的结构如下图所示



对于vring了解的朋友应该很熟悉这个结构，没错，这就是通过vring管理的结构，换句话说，前后端直接共享的其实是vring。也就是说针对同一个队列（比如网卡的发送队列），前后端已经形成一种协议，通过这段内存区交换数据的地址信息。在把数据的地址信息写入到desc数组中后，仅仅需要通知另一端，另一端就知道从哪里取出数据。当然还是通过desc数组。具体数据的传递过程参见其他小结。因此在初始化阶段，前端分配好内存区，并初始化好前端的vring后，就把内存区的信息传递到后端，后端也利用这个内存区的信息初始化队列相关的vring。这样vring就在前后端保持了一致。原理就是如此，下面看具体初始化代码：

前端：

virtnet_probe->init_vqs->virtnet_find_vqs->vi->vdev->config->find_vqs(vp_find_vqs)->vp_try_to_find_vqs->setup_vq,在setup_vp中通过IO端口和后端交互完成前面我们说的协议。看下该函数

```
static struct virtqueue *setup_vq(struct virtio_device *vdev, unsigned index,
                  void (*callback)(struct virtqueue *vq),
                  const char *name,
                  u16 msix_vec)
{
    struct virtio_pci_device *vp_dev = to_vp_device(vdev);
    struct virtio_pci_vq_info *info;
    struct virtqueue *vq;
    unsigned long flags, size;
    u16 num;
    int err;

    /* Select the queue we're interested in */
    iowrite16(index, vp_dev->ioaddr + VIRTIO_PCI_QUEUE_SEL);

    /* Check if queue is either not available or already active. */
    num = ioread16(vp_dev->ioaddr + VIRTIO_PCI_QUEUE_NUM);
    if (!num || ioread32(vp_dev->ioaddr + VIRTIO_PCI_QUEUE_PFN))
        return ERR_PTR(-ENOENT);

    /* allocate and fill out our structure the represents an active
     * queue */
    info = kmalloc(sizeof(struct virtio_pci_vq_info), GFP_KERNEL);
    if (!info)
        return ERR_PTR(-ENOMEM);

    info->num = num;
    info->msix_vector = msix_vec;
```

```
    size = PAGE_ALIGN(vring_size(num, VIRTIO_PCI_VRING_ALIGN));
    info->queue = alloc_pages_exact(size, GFP_KERNEL|__GFP_ZERO);
    if (info->queue == NULL) {
        err = -ENOMEM;
        goto out_info;
    }

    /* activate the queue */
    /*这里实际上把info->queue的GPA(页框号写入到了设备的VIRTIO_PCI_QUEUE_PFN)*/
    iowrite32(virt_to_phys(info->queue) >> VIRTIO_PCI_QUEUE_ADDR_SHIFT,
          vp_dev->ioaddr + VIRTIO_PCI_QUEUE_PFN);

    /* create the vring */
    vq = vring_new_virtqueue(index, info->num, VIRTIO_PCI_VRING_ALIGN, vdev,
                 true, info->queue, vp_notify, callback, name);
    if (!vq) {
        err = -ENOMEM;
        goto out_activate_queue;
    }

    vq->priv = info;
    info->vq = vq;

    if (msix_vec != VIRTIO_MSI_NO_VECTOR) {
        iowrite16(msix_vec, vp_dev->ioaddr + VIRTIO_MSI_QUEUE_VECTOR);
        msix_vec = ioread16(vp_dev->ioaddr + VIRTIO_MSI_QUEUE_VECTOR);
        if (msix_vec == VIRTIO_MSI_NO_VECTOR) {
            err = -EBUSY;
            goto out_assign;
        }
    }

    if (callback) {
        spin_lock_irqsave(&vp_dev->lock, flags);
        list_add(&info->node, &vp_dev->virtqueues);
        spin_unlock_irqrestore(&vp_dev->lock, flags);
    } else {
        INIT_LIST_HEAD(&info->node);
    }

    return vq;

out_assign:
    vring_del_virtqueue(vq);
out_activate_queue:
    iowrite32(0, vp_dev->ioaddr + VIRTIO_PCI_QUEUE_PFN);
    free_pages_exact(info->queue, size);
out_info:
    kfree(info);
    return ERR_PTR(err);
}
```

注意协商的步骤，首先通过VIRTIO_PCI_QUEUE_SEL标记本次操作的队列索引，因为每个队列都有自己的vring，即需要自己的共享内存区。然后检查队列是否可用，这是通过VIRTIO_PCI_QUEUE_NUM，如果返回的结果是0，则表示没有队列可用，则返回错误。接着通过VIRTIO_PCI_QUEUE_PFN检查是否已经激活，如果已经激活，同样返回错误。这些检查通过就可以予以初始化了，具体先分配一个中间结构virtio_pci_vq_info，这不是重点，后面通过alloc_pages_exact向伙伴系统分配了不小于size的连续物理内存，等会我们再说size的问题，然后把这块物理页框号（GPA>>VIRTIO_PCI_QUEUE_ADDR_SHIFT）写入到VIRTIO_PCI_QUEUE_PFN，这样后端就会得到这块内存区的信息。然后我们先看下前端利用这块内存区做了什么？看下面的vring_new_virtqueue函数，该函数中调用vring_init来初始化vring

```
static inline void vring_init(struct vring *vr, unsigned int num, void *p,
                  unsigned long align)
{
    vr->num = num;
    vr->desc = p;
    vr->avail = p + num*sizeof(struct vring_desc);
    vr->used = (void *)(((unsigned long)&vr->avail->ring[num] + sizeof(__u16)
        + align-1) & ~(align - 1));
}
```

这个函数正好体现了我们前面那个结构图。这样前端vring就初始化好了。对队列填充数据时就是根据这个vring填充信息。

后端（qemu端）

主要操作都在virtio_ioport_write中，我们只关注三个case

```
case VIRTIO_PCI_QUEUE_PFN:
        /*pa就是desc的GPA*/
        pa = (target_phys_addr_t)val << VIRTIO_PCI_QUEUE_ADDR_SHIFT;
        if (pa == 0) {
            virtio_pci_stop_ioeventfd(proxy);
            virtio_reset(proxy->vdev);
            msix_unuse_all_vectors(&proxy->pci_dev);
        }
        else
            virtio_queue_set_addr(vdev, vdev->queue_sel, pa);
        break;
    case VIRTIO_PCI_QUEUE_SEL:
        if (val < VIRTIO_PCI_QUEUE_MAX)
            vdev->queue_sel = val;
        break;
    case VIRTIO_PCI_QUEUE_NOTIFY:
        if (val < VIRTIO_PCI_QUEUE_MAX) {
            virtio_queue_notify(vdev, val);
        }
        break;
```

可以看到在VIRTIO_PCI_QUEUE_SEL时候，仅仅是标记了下设备中的queue_sel表示当前操作的队列索引。下面在通过VIRTIO_PCI_QUEUE_PFN传递地址的时候，调用virtio_queue_set_addr设置后端相关队列的vring该函数实现较简单

```
void virtio_queue_set_addr(VirtIODevice *vdev, int n, target_phys_addr_t addr)
{
    vdev->vq[n].pa = addr;
    virtqueue_init(&vdev->vq[n]);
}
```

```
static void virtqueue_init(VirtQueue *vq)
{
    target_phys_addr_t pa = vq->pa;

    vq->vring.desc = pa;
    vq->vring.avail = pa + vq->vring.num * sizeof(VRingDesc);
    vq->vring.used = vring_align(vq->vring.avail +
                                 offsetof(VRingAvail, ring[vq->vring.num]),
                                 VIRTIO_PCI_VRING_ALIGN);
}
```

看到这里有么有很面熟，没错，这个函数和前端初始化vring的函数很是类似，这样前后端的vring就同步起来了……

而在guest通知后端的时候，通过VIRTIO_PCI_QUEUE_NOTIFY接口，该函数调用了virtio_queue_notify_vq继而调用 vq->handle_output……就这样，后端就得到通知着手处理了！

后记：

到此，virtIO部分已经分析的差不多了，分析期间真实感觉到了自己知识的匮乏，其间多次向开发者求助，并均得到认真回复，在此在此感谢这些优秀的开发者。有时候看内核代码就感觉工程师和硬件在干仗，站在工程师的角度，需要尽其所能榨取硬件的性能。大到实现算法的优化，小到分析程序执行流的概率，从而针对编译做优化。站在硬件的角度，你处理不好，我就不给你工作。而从这方面，工程师自然是完胜，并且还在不遗余力的朝着胜利的另一个境界挺近，即征服硬件！哈哈，不过谁都知道，这是一场没有胜负的战争，工程师自然优秀，但是，因为工程师内部的竞争，这样战斗将永无休止！！唉，瞎扯淡了，各位朋友，下篇文章见！

分类：KVM虚拟化技术, linux 内核源码分析, qemu, IO Virtualization


jack.chen
关注 - 12
粉丝 - 44
+加关注

« 上一篇： virtio后端驱动详解
» 下一篇： virtio前端驱动详解

posted @ 2016-11-15 15:43 jack.chen Views(6924) Comments(5) Edit 收藏

Post Comment

#1楼 2016-11-27 17:32 | 非专业OPS

分析的不错，继续加油，期待佳作。

支持(0) 反对(

#2楼 2017-05-26 16:50 | 小白爱虚拟

还看不太懂

支持(0) 反对(

#3楼 [楼主] 2017-06-02 14:54 | jack.chen

@ 小白爱虚拟
具体实现的确有些复杂，需要结合源代码认真分析.....

支持(0) 反对(

#4楼 2017-08-04 14:38 | GYZheng

先赞一个!
请问下，这里的vring应该是前后端采用共享内存方式实现的吧？ 比如Guest发送I/O数据的过程中，涉及GuestOS -- > KVM ----- > Qemu，这些过程之中应该都是不涉及数据包的拷贝，只是最后Qemu需要拷贝出来，发送给tap 备，这样理解对么?

支持(0) 反对(

#5楼 [楼主] 2017-08-10 18:06 | jack.chen

@ GYZheng
没错，也正是这个原因需要前后端驱动的配合，数据准备好后直接通知对方知道即可！进一步的实现可以参看vhost-rnel和vhost-user.

支持(0) 反对(

刷新评论 刷新页面 返回顶

注册用户登录后才能发表评论，请 登录 或 注册， 访问 网站首页。

相关博文：
· Linux Kernel Vhost 架构
· virtio后端驱动详解
· virtio 简介
· virtIO之VHOST工作原理简析
· 理解 Linux 网络栈（3）：QEMU/KVM + VxLAN 环境下的 Segmentation Offloading 技术（发送端）
» 更多推荐...

合辑 | 学习python不可不知的开发者词条汇总!

最新 IT 新闻:
· 北京文化被实名举报财务造假 今日开盘一字跌停
· 腾讯云发布专属语音识别模型，加速在金融、音视频行业应用
· Facebook财报电话会议实录：电商广告营收增长强劲，将拉动总营收
· 紫光股份拟定增120亿 董事长：基础科学没钱办不成事
· 张朝阳：我5月份要尝试直播带货 搜狐不考虑私有化
» 更多新闻...

以马内利