virtio前端驱动详解

2016-11-08

前段时间大致整理了下virtIO后端驱动的工作模式以及原理，今天就从前端驱动的角度描述下目前Linux内核代码中的virtIO驱动是如何配合后端进行工作的。

注：本节代码参考Linux 内核3.11.1代码

virtIO驱动从架构上来讲可以分为两部分，一个是其作为PCI设备本身的驱动，此驱动需要提供一些基本的操作PCI设备本身的函数比如PCI设备的探测、删除、配置空间的设置和寄存器空间的读写等。而另一个就是其virtIO设备本身实现的功能驱动例如网络驱动、块设备驱动、console驱动等。所以我们要看还是分两部分，先介绍PCI设备本身的驱动，然后在介绍实际功能驱动。

一、PCI设备本身驱动

在前面的PCI系列文章中对Linux内核中PCI设备驱动做了分析，所以这里我们只分析和virtIO相关的部分。

二、功能驱动部分

其实大部分的功能在后端驱动已经介绍，只是有些功能是在前端实现的，比如说virtqueue的初始化、avail buffer的添加以及used buffer的消费，还有比较很重要的是前后端vring的同步。

鉴于前面已经有了基本的概念基础，那么我们直接从网络驱动下手，分析驱动从注册到接受数据的整个流程。（参考代码virtio-net.c）

看下网络驱动注册的操作函数：

```
1 static const struct net_device_ops virtnet_netdev = {
2     .ndo_open            = virtnet_open,
3     .ndo_stop            = virtnet_close,
4     .ndo_start_xmit      = start_xmit,
5     .ndo_validate_addr   = eth_validate_addr,
6     .ndo_set_mac_address = virtnet_set_mac_address,
7     .ndo_set_rx_mode     = virtnet_set_rx_mode,
8     .ndo_change_mtu          = virtnet_change_mtu,
9     .ndo_get_stats64     = virtnet_stats,
10    .ndo_vlan_rx_add_vid = virtnet_vlan_rx_add_vid,
11    .ndo_vlan_rx_kill_vid = virtnet_vlan_rx_kill_vid,
12    .ndo_select_queue     = virtnet_select_queue,
13 #ifdef CONFIG_NET_POLL_CONTROLLER
14    .ndo_poll_controller = virtnet_netpoll,
15 #endif
16 };
```

发送数据的函数为start_xmit，该函数接收来自网络协议栈的函数并写入到ring buffer中，然后通知后端驱动。

```
1 static netdev_tx_t start_xmit(struct sk_buff *skb, struct net_device *dev)
2 {
3     struct virtnet_info *vi = netdev_priv(dev);
4     int qnum = skb_get_queue_mapping(skb);
5     struct send_queue *sq = &vi->sq[qnum];
6     int err;
```


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

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(21050)
2. PCI 设备详解一(15726)
3. 进程的挂起、阻塞和睡眠(13609)
4. Linux下桥接模式详解一(13391)
5. virtio后端驱动详解(10504)

推荐排行榜

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

```
7
8       /* Free up any pending old buffers before queueing new ones. */
9       free_old_xmit_skbs(sq);
10
11      /* Try to transmit */
12      err = xmit_skb(sq, skb);
13
14      /* This should not happen! */
15      if (unlikely(err)) {
16          dev->stats.tx_fifo_errors++;
17          if (net_ratelimit())
18              dev_warn(&dev->dev,
19                       "Unexpected TXQ (%d) queue failure: %d\n", qnum, err);
20          dev->stats.tx_dropped++;
21          kfree_skb(skb);
22          return NETDEV_TX_OK;
23      }
24      /*通知后端驱动*/
25      virtqueue_kick(sq->vq);
26
27      /* Don't wait up for transmitted skbs to be freed. */
28      skb_orphan(skb);
29      nf_reset(skb);
30
31      /* Apparently nice girls don't return TX_BUSY; stop the queue
32       * before it gets out of hand.  Naturally, this wastes entries. */
33      if (sq->vq->num_free < 2+MAX_SKB_FRAGS) {
34          netif_stop_subqueue(dev, qnum);
35          if (unlikely(!virtqueue_enable_cb_delayed(sq->vq))) {
36              /* More just got used, free them then recheck. */
37              free_old_xmit_skbs(sq);
38              if (sq->vq->num_free >= 2+MAX_SKB_FRAGS) {
39                  netif_start_subqueue(dev, qnum);
40                  virtqueue_disable_cb(sq->vq);
41              }
42          }
43      }
44      return NETDEV_TX_OK;
45  }
```

函数中首先获取了buffer对应的发送队列sendqueue,调用了一个关键的函数**xmit_skb**，具体的添加buffer到queue中的操作就是在此函数实现的：

```
1  static int xmit_skb(struct send_queue *sq, struct sk_buff *skb)
2  {
3      struct skb_vnet_hdr *hdr = skb_vnet_hdr(skb);
4      const unsigned char *dest = ((struct ethhdr *)skb->data)->h_dest;
5      struct virtnet_info *vi = sq->vq->vdev->priv;
6      unsigned num_sg;
7
8      pr_debug("%s: xmit %p %pM\n", vi->dev->name, skb, dest);
9
10     if (skb->ip_summed == CHECKSUM_PARTIAL) {
11         hdr->hdr.flags = VIRTIO_NET_HDR_F_NEEDS_CSUM;
12         hdr->hdr.csum_start = skb_checksum_start_offset(skb);
13         hdr->hdr.csum_offset = skb->csum_offset;
14     } else {
15         hdr->hdr.flags = 0;
16         hdr->hdr.csum_offset = hdr->hdr.csum_start = 0;
17     }
18
19     if (skb_is_gso(skb)) {
20         hdr->hdr.hdr_len = skb_headlen(skb);
21         hdr->hdr.gso_size = skb_shinfo(skb)->gso_size;
22         if (skb_shinfo(skb)->gso_type & SKB_GSO_TCPV4)
23             hdr->hdr.gso_type = VIRTIO_NET_HDR_GSO_TCPV4;
24         else if (skb_shinfo(skb)->gso_type & SKB_GSO_TCPV6)
25             hdr->hdr.gso_type = VIRTIO_NET_HDR_GSO_TCPV6;
26         else if (skb_shinfo(skb)->gso_type & SKB_GSO_UDP)
27             hdr->hdr.gso_type = VIRTIO_NET_HDR_GSO_UDP;
28         else
29             BUG();
30         if (skb_shinfo(skb)->gso_type & SKB_GSO_TCP_ECN)
```

```
31              hdr->hdr.gso_type |= VIRTIO_NET_HDR_GSO_ECN;
32      } else {
33          hdr->hdr.gso_type = VIRTIO_NET_HDR_GSO_NONE;
34          hdr->hdr.gso_size = hdr->hdr.hdr_len = 0;
35      }
36
37      hdr->mhdr.num_buffers = 0;
38
39      /* Encode metadata header at front. 首个sg entry存储头部信息*/
40      if (vi->mergeable_rx_bufs)
41          sg_set_buf(sq->sg, &hdr->mhdr, sizeof hdr->mhdr);
42      else
43          sg_set_buf(sq->sg, &hdr->hdr, sizeof hdr->hdr);
44      /*映射数据到sg,当前在sq->sg里面已经记录数据的地址信息了*/
45      num_sg = skb_to_sgvec(skb, sq->sg + 1, 0, skb->len) + 1;
46      /*调用函数把sg 信息记录到队列中的desc中*/
47      return virtqueue_add_outbuf(sq->vq, sq->sg, num_sg, skb, GFP_ATOMIC);
48 }
```

这里我们先介绍下两种virtIO 头部：

```
1 struct skb_vnet_hdr {
2     union {
3         struct virtio_net_hdr hdr;
4         struct virtio_net_hdr_mrg_rxbuf mhdr;
5     };
6 };
```

里面包含一个union分别是virtio_net_hdr和virtio_net_hdr_mrg_rxbuf，前者是普通的数据包头部，后者是支持合并buffer的数据包的头部，并且virtio_net_hdr是virtio_net_hdr_mrg_rxbuf的一个内嵌结构，这样再看前面的函数代码

首先判断硬件是否已经添加了校验字段，设置virtio_net_hdr中相关的值；然后判断数据包是否是GSO类型，再次设置virtio_net_hdr相关字段的值。关于GSO类型，文章最后会介绍。设置好头部后，进入下一个if,判断设备是否支持合并buffer，是的话就调用函数sg_set_buf把virtio_net_hdr_mrg_rxbuf记录到首个sg table的第一个表项 中，否则添加virtio_net_hdr。这样设置好头部，就调用skb_to_sgvec函数把skb buffer记录到sg table中，然后调用virtqueue_add_outbuf把sg table转换到发送队列的ring desc中。回到上层的函数start_xmit中，在xmit_skb返回后，如果返回值正常，就调用virtqueue_kick函数通知后端驱动。

在通知后端驱动后判断剩余可用的desc是否小于2+MAX_SKB_FRAGS（为保证安全，一个数据包最多可能使用2+MAX_SKB_FRAGS个物理buffer，virtIO 头部占用一个，数据包头部占用一个，剩下的是数据包最大分片数），不小于的话需要调用netif_stop_subqueue禁止下一个数据包的发送。

下面回过头分析sg_set_buf、skb_to_sgvec和virtqueue_add_outbuf。

```
1 static inline void sg_set_buf(struct scatterlist *sg, const void *buf,
2                   unsigned int buflen)
3 {
4     sg_set_page(sg, virt_to_page(buf), buflen, offset_in_page(buf));
5 }
```

在分析的同时我们也看下scatter list是如何组织的。首先看参数sg是sg table 的指针，buf指向数据，buflen是数据的长度。可以看到函数中仅仅是调用了sg_set_page函数，所以这里具体的物理buffer块是按照页为单位的。由于buf并不一定是页对齐的，所以需要一个buf指针到所在页基址的偏移。

```
1 static inline void sg_set_page(struct scatterlist *sg, struct page *page,
2                    unsigned int len, unsigned int offset)
3 {
4     sg_assign_page(sg, page);
5     sg->offset = offset;//data在页面中的偏移
6     sg->length = len;//data的长度
7 }
```

到该函数中，page是一个指向一个页的指针，该函数中调用了sg_assign_page函数设置sg->page_lin指向page,这样在sg table entry和具体的buffer就联系起来了。然后把buffer的offset和length记录到sg entry中。

结合xmit_skb函数，那么在经过sg_set_buffer之后，sg table的第一个表项便和hdr->mhdr或者hdr->hdr联系起来。

Function skb_to_sgvec

```
1 int skb_to_sgvec(struct sk_buff *skb, struct scatterlist *sg, int offset, int len)
2 {
3 /*buffer数据存储在sg的个数*/
4     int nsg = __skb_to_sgvec(skb, sg, offset, len);
5 /*标记最后一个sg entry结束*/
6     sg_mark_end(&sg[nsg - 1]);
7
8     return nsg;
9 }
```

该函数直接调用了__skb_to_sgvec函数，有其实现具体的功能，然后设置最后一个entry为end end entry,以此表明sg list的结束。

```
 1 static int
 2 __skb_to_sgvec(struct sk_buff *skb, struct scatterlist *sg, int offset, int len)
 3 {
 4     int start = skb_headlen(skb);
 5     int i, copy = start - offset;
 6     struct sk_buff *frag_iter;
 7     int elt = 0;/*elt记录sg entry的个数*/
 8
 9     if (copy > 0) {/*copy是头部的长度*/
10         if (copy > len)/*头部大于总长度。。。几乎不可能*/
11             copy = len;
12         /*skb->data + offset是数据起始位置，尽管offset一般是0，所以可以看出头部是占用一个sg
entry*/
13         sg_set_buf(sg, skb->data + offset, copy);
14         elt++;
15         if ((len -= copy) == 0)
16             return elt;
17         /*offset记录数据copy的位置*/
18         offset += copy;
19     }
20     /*映射非线性数据即skb_shared_info相关的数据*/
21     for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
22         int end;
23
24         WARN_ON(start > offset + len);
25
26         end = start + skb_frag_size(&skb_shinfo(skb)->frags[i]);
27         if ((copy = end - offset) > 0) {
28             skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
29
30             if (copy > len)
31                 copy = len;
32             sg_set_page(&sg[elt], skb_frag_page(frag), copy,
33                     frag->page_offset+offset-start);
34             elt++;
35             if (!(len -= copy))
36                 return elt;
37             offset += copy;
38         }
39         start = end;
40     }
41
42     skb_walk_frags(skb, frag_iter) {
43         int end;
44
45         WARN_ON(start > offset + len);
46
47         end = start + frag_iter->len;
48         if ((copy = end - offset) > 0) {
49             if (copy > len)
```

```
50                 copy = len;
51             elt += __skb_to_sgvec(frag_iter, sg+elt, offset - start,
52                                   copy);
53             if ((len -= copy) == 0)
54                 return elt;
55             offset += copy;
56         }
57         start = end;
58     }
59     BUG_ON(len);
60     return elt;
61 }
```

该函数把一个完整的skbuffer记录到sg table,要搞清楚这些最好对sk_buffer结构比较清楚，而对sk_buffer结构可以参考其的有关专门的介绍。本节我们只介绍相关的部分，这里可以把skbuffer分成两部分：

1、skbuffer本身的数据

2、skb_shared_info记录的分片数据

而上面的函数也是把这两部分分开记录的，首先调用skb_headlen函数获取sk_buffer本身的头部以及数据（不包含分片数据），copy为实际的长度，不过这里传递进来的offset为0，所以copy即start，接着就调用了**sg_set_buf**函数把从skb_buffer->data+offset起始的有效数据记录到sg table,elt是一个变量记录使用的sg entry个数。

如果这里没有分片数据，那么直接返回elt,否则需要记录offset的位置，便于下次知道上次数据的记录位置。

下面一个for循环时完成第二部分数据的记录，即分片数据。sk_buffer->end指向一个skb_shared_info结构，该结构管理分片数据，nr_frags表示分片的数量，所以以此为基添加分片。

循环内部的内容有点混乱感觉，这里详解解释下：

注意一下几个变量：

/*
*len是未复制的数据的长度
*offset是已经复制的数据的长度
*copy是本次要复制的数据的长度
*start 是线性数据段的长度
*映射非线性数据即skb_shared_info相关的数据
*/

其实说实话我个人觉得这几个变量的命名很是失败，start和end咋一看容易让人感觉这是指针，但是没办法，说让咱写不出这种代码勒！

在循环之前，start是代表线性数据段的长度，offset在完成映射后就执行offset += copy，所以offset=start。

在循环中end=start +分片size，copy=end-offset,那么实际上，本次copy的长度也就是分片的size。如果分片size大于0，则表示分片存在数据，那么copy=end-offset必定大于0，直接调用sg_set_page函数把当前分片扩展成一个page然后映射到sg table.接着更新len，判断是否映射完成，即len是否为0，不为0的话更新offset。最后更新start=end.

下面遍历所有的sk_buffer->frag_list,对于每个sk_buffer,都调用__skb_to_sgvec对其中数据进行映射，最后返回elt即使用的sg entry个数。

/*关于sk_buffer，确实其组织方式很复杂，会单独讲解，碍于篇幅，就不在这里详细描述*/

回到skb_to_sgvec函数中，调用sg_mark_end对最后一个entry做末端标记。具体而言就是设置sg->page_link第二位为1：sg->page_link |= 0x02;

到这里就把buffer映射到了sg table中。那么如何把sg填入ring desc数组中呢？看virtqueue_add_outbuf

这里需要注意一下传入的data指针是skb，即数据的虚拟起始地址

```
1 int virtqueue_add_outbuf(struct virtqueue *vq,
2              struct scatterlist sg[], unsigned int num,
3              void *data,
4              gfp_t gfp)
5 {
6     return virtqueue_add(vq, &sg, sg_next_arr, num, 0, 1, 0, data, gfp);
7 }
```

这里就是简单的调用了下virtqueue_add

```
static inline int virtqueue_add(struct virtqueue *_vq,
                struct scatterlist *sgs[],
                struct scatterlist *(*next)
                  (struct scatterlist *, unsigned int *),
                unsigned int total_out,
                unsigned int total_in,
                unsigned int out_sgs,
                unsigned int in_sgs,
                void *data,
                gfp_t gfp)
{
    struct vring_virtqueue *vq = to_vvq(_vq);
    struct scatterlist *sg;
    unsigned int i, n, avail, uninitialized_var(prev), total_sg;
    int head;

    START_USE(vq);

    BUG_ON(data == NULL);

#ifdef DEBUG
    {
        ktime_t now = ktime_get();

        /* No kick or get, with .1 second between?  Warn. */
        if (vq->last_add_time_valid)
            WARN_ON(ktime_to_ms(ktime_sub(now, vq->last_add_time))
                        > 100);
        vq->last_add_time = now;
        vq->last_add_time_valid = true;
    }
#endif

    total_sg = total_in + total_out;
//这里判断是否支持间接描述符并且总的entry数要大于1且，vring里至少有一个空buffer
    /* If the host supports indirect descriptor tables, and we have multiple
     * buffers, then go indirect. FIXME: tune this threshold */
    if (vq->indirect && total_sg > 1 && vq->vq.num_free) {
        head = vring_add_indirect(vq, sgs, next, total_sg, total_out,
                      total_in,
                      out_sgs, in_sgs, gfp);
        if (likely(head >= 0))//如果执行成功，就直接执行add_head段
            goto add_head;
    }
    /*否则就可能是不支持间接描述符，那么这是需要有足够的desc来装载哪些entry*/

    BUG_ON(total_sg > vq->vring.num);
    BUG_ON(total_sg == 0);
/*如果可用的desc数量不够，则不能执行成功*/
    if (vq->vq.num_free < total_sg) {
        pr_debug("Can't add buf len %i - avail = %i\n",
             total_sg, vq->vq.num_free);
        /* FIXME: for historical reasons, we force a notify here if
         * there are outgoing parts to the buffer.  Presumably the
         * host should service the ring ASAP. */
        if (out_sgs)
            vq->notify(&vq->vq);
        END_USE(vq);
        return -ENOSPC;
    }

    /* We're about to use some buffers from the free list. */
    vq->vq.num_free -= total_sg;
/*可用的desc数量够的话就可以直接使用这些desc,针对desc的操作都是一样的*/
    head = i = vq->free_head;
    for (n = 0; n < out_sgs; n++) {
        for (sg = sgs[n]; sg; sg = next(sg, &total_out)) {
            vq->vring.desc[i].flags = VRING_DESC_F_NEXT;
            vq->vring.desc[i].addr = sg_phys(sg);
            vq->vring.desc[i].len = sg->length;
            prev = i;
```

```
72              i = vq->vring.desc[i].next;
73          }
74      }
75      for (; n < (out_sgs + in_sgs); n++) {
76          for (sg = sgs[n]; sg; sg = next(sg, &total_in)) {
77              vq->vring.desc[i].flags = VRING_DESC_F_NEXT|VRING_DESC_F_WRITE;
78              vq->vring.desc[i].addr = sg_phys(sg);
79              vq->vring.desc[i].len = sg->length;
80              prev = i;
81              i = vq->vring.desc[i].next;
82          }
83      }
84      /* Last one doesn't continue. */
85      vq->vring.desc[prev].flags &= ~VRING_DESC_F_NEXT;
86
87      /* Update free pointer */
88      vq->free_head = i;
89
90  add_head:
91      /* Set token. */
92      /*在客户机驱动写入数据到buffer以后，设置data数组以head为下标的内容为buffer的虚拟地址*/
93      vq->data[head] = data;
94
95      /* Put entry in available array (but don't update avail->idx until they
96       * do sync). */
97
98      //然后把本次传送所用到的描述符表的信息写入avail结构中
99      /*&应该是要保证idx小于vq->vring.num*/
100     avail = (vq->vring.avail->idx & (vq->vring.num-1));
101     /*设置avail_ring*/
102     vq->vring.avail->ring[avail] = head;
103
104     /* Descriptors and available array need to be set before we expose the
105      * new available array entries. */
106     virtio_wmb(vq->weak_barriers);
107     vq->vring.avail->idx++;
108     vq->num_added++;
109
110     /* This is very unlikely, but theoretically possible.  Kick
111      * just in case. */
112     if (unlikely(vq->num_added == (1 << 16) - 1))
113         virtqueue_kick(_vq);
114
115     pr_debug("Added buffer head %i to %p\n", head, vq);
116     END_USE(vq);
117
118     return 0;
119 }
```

该函数实现了把sg table中记录的信息，复制到发送队列的ring的desc数组中。看下该函数几个重要的参数：

struct virtqueue *_vq 添加的目的队列

struct scatterlist *sgs[] 要添加的sg table

struct scatterlist *(*next) 一个函数指针，用于获取下一个sg entry
(struct scatterlist *, unsigned int *)

unsigned int total_out 输出的sg entry的个数

unsigned int total_in 输入的sg entry的个数

unsigned int out_sgs 输出的sg list的个数，这里一个out_sgs代表一个完整的skb_buffer

unsigned int in_sgs 输入的sg list的个数，这里一个in_sgs代表一个完整的skb_buffer

void *data 一个指向sk_buffer的指针。

介绍完这些，下面的就很明确了，

首先判断是否支持队列是否支持indirect descriptor，首选也是使用这种方式，不过这种方式需要占用主描述符表的一个表项，并且在total_sg>1的时候使用（total_sg=1时只是使用主描述符表即可），如果满足条件就调用vring_add_indirect函数添加间接描述符表，并把sg table中记录的信息写入到描述符表中。

如果不支持，就只能使用主描述符表，此时主描述符表的空闲表项数必须大于等于total_sg，具体可用的数目记录在vq->vq.num_free，而首个可用的表项的下标记录在vq->free_head中，下面的for循环就依次把sg table中entry的信息记录到对应的desc表中，需要注意的是desc中的addr记录的是buffer的物

理地址，而sg是记录的页虚拟地址。下面的一个for循环是添加in_sg,**关于in_sg和out_sg，目前在网络驱动部分的发送队列只使用out_sg,而接受队列只使用in_sg,而控制队列就可能两个都使。这里我们忽略此点即可。**

最后依然需要设置最后一个desc为末端desc，并移动vq->free_head便于下次使用。

add_head后面的部分是和后端驱动相关的。

主要是在发送队列的ring[]中获取一项，写入前面写入的sk_buffer 对应的desc表中 的head,即首个描述符的下标。然后更新vq->vring.avail->idx。

到现在前端驱动已经设置完成，剩下就要通知后端驱动读取数据了，回到start_xmit函数中，看到调用了virtqueue_kick函数

```
1 void virtqueue_kick(struct virtqueue *vq)
2 {
3     if (virtqueue_kick_prepare(vq))
4         virtqueue_notify(vq);
5 }
```

virtqueue_kick函数调用了virtqueue_kick_prepare判断下当前是否应该notify后端，如果应该，就调用virtqueue_notify函数，该函数直接调用了vq->notify函数，参数是队列指针。

而具体实现的是下面的函数vp_notify在virtio_pci.c中

```
1 static void vp_notify(struct virtqueue *vq)
2 {
3     struct virtio_pci_device *vp_dev = to_vp_device(vq->vdev);
4
5     /* we write the queue's selector into the notification register to
6      * signal the other end */
7     iowrite16(vq->index, vp_dev->ioaddr + VIRTIO_PCI_QUEUE_NOTIFY);
8 }
```

可以看到，实际上前端通知后端仅仅是把队列的索引写入到对应的设备寄存器中,这样在后端qemu就会知道是哪个队列发生了add buffer，然后就从对应队列的buffer取出数据。

而对于前后端notify机制的分析，这里我们单独拿出一节来讲，感兴趣可以参考：

virtIO前后端notify机制详解

参考：

1. LInux 3.11.1内核源代码
2. qemu 2.7.0源代码
3. qemu 开发者的帮助

分类: KVM虚拟化技术, linux 内核源码分析, qemu

jack.chen
关注 - 12
粉丝 - 44
+加关注

« 上一篇： virtIO前后端notify机制详解
» 下一篇： 为何要写博客

posted @ 2016-11-15 15:48 jack.chen Views(5128) Comments(2) Edit 收藏

Post Comment

#1楼 2017-09-12 11:23 | 我不会放弃

留名，以后备用

支持(0) 反对(

#2楼 2019-12-27 15:13 | Linux-inside

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 selector into the notification register to
* signal the other end */
iowrite16(vq->index, (void __iomem *)vq->priv);
return true;
}
注意到vq->priv这个参数在virtio_pci_modern.c和virtio_pci_legacy.c这两个中是不同的。能否介绍下legacy和m
odern pci之间的差异?
谢谢。

支持(0) 反对(

刷新评论 刷新页面 返回顶部

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:
· virtio 简介
· virtio后端驱动详解
· virtIO前后端notify机制详解
· virtIO之VHOST工作原理简析
· 26.Linux-网卡驱动介绍以及制作虚拟网卡驱动(详解)
» 更多推荐...
Java经典面试题整理及答案详解（二）

最新 IT 新闻:
· 特斯拉：手握81亿美元现金 足以应对疫情
· Google Meet日活跃用户破1亿 每日新增用户约300万
· 快递柜收费，难救"穷疯"的丰巢
· 谷歌为Drive、Docs和Sheets等重新设计分享用户界面
· 没有中间商赚差价，我就会快乐吗?
» 更多新闻...

以马内利