virtio后端驱动详解

2016-10-08

virtIO是一种半虚拟化驱动，广泛用于在XEN平台和KVM虚拟化平台，用于提高客户机IO的效率，事实证明，virtIO极大的提高了VM IO 效率，配备virtIO前后端驱动的情况下，客户机IO效率基本达到和宿主机一样的水平。咱们本次的分析以qemu-kvm架构的虚拟化平台为基础，分析virtIO前后端驱动。当然后端就指有qemu实现的虚拟PCI设备，而前端自然就是客户操作系统中的virtIO驱动。需要前后配合才能完成数据的传输。

本节的重点在于首先对virtIO进行总体介绍，然后结合LInux 3.11.1内核代码对后端驱动进行分析。

一、virtIO 总体介绍

对于virtIO的基本介绍如前面所述，而virtIO的出现所解决的另一个问题就是给众多的虚拟化平台提供了一个统一的IO模型，KVM、XEN、VMWare等均可以利用virtIO进行IO虚拟化，在提高IO效率的同时也极大的减少了自家软件开发的工作量。那么对于virtIO基本介绍我们就不详细深入，事实上，前面已经足以说明virtIO是何方神圣，下面主要是深入内在分析virtIO 其实现原理。

二、现有设备虚拟化存在的问题

这里以网络驱动为例，其他的设备驱动类似。先看传统的利用简单虚拟网卡进行网络虚拟化的情况，这种情况下数据包的收发模式根物理机没有本质区别。虚拟网卡发现有数据包到来，需要先接收下来，这个过程会发生数据的复制，然后向客户机注入软件中断，客户机接收到信号，然后处理中断，最终完成数据包的处理。这种模式下工作效率的瓶颈主要两点：

1、数据的复制

2、根模式和非根模式的频繁切换

即使目前qemu的虚拟网卡已经使用DMA的方式直接把数据写入到客户机内存，然后通知客户机，但是仍然免不了数据复制带来的性能开销。

那么有没有一种方式能够彻底的解决上面两个问题呢？？当然要彻底消除根模式和非根模式的切换是不可能的，毕竟虚拟机还是有Hypervisor管理的，但是我们可以最大程度的减少这种不必要的切换。virtIO为这一问题提供了比较好的解决方案。

第一个问题：数据的复制

在virtIO实现了零复制。不管是什么虚拟化平台，虚拟机是运行在host内存中或者说虚拟机共享同一块内存，那么既然如此我们就没有必要在同一块内存不同区域之间复制数据，而可以进行简单的地址重映射即可。以KVM虚拟机为例，虚拟机运行在HOST的内存中，且在HOST上表现为一个普通的qemu进程。前面我们分析qemu网络虚拟化的时候已经分析，宿主机接收到数据包会根据目的MAC进行数据包的转发，如果是发往客户机的，则把数据包转发到一个虚拟端口（tap设备模拟），其本质实际上只是把数据共享到一个用户空间应用程序(通过虚拟设备)，这里我们就是指qemu，这个过程是不需要我们操心的。数据到了qemu,其实这里有一个Net client来接收，

第二个问题：根模式和非根模式的频繁切换、

这里考虑下为何会有线程或者说操作系统中用户模式和内核模式都是同样的道理，线程出现的很大一部分原因就是进程切换代价太高，需要保存和恢复的东西太多，以至于每次切换都要做很多重复的工作，这才有了线程或者说是轻量级的进程。那么在这里，根模式和非根模式也是这个道理，只是这个模式的切换比进程的切换需要消耗更多的资源，因为每次切换保存的不在是一个普通进程的上下文，而是一个虚拟机的上下文，尽管虚拟机在HOST上同样是表现为一个进程，但是其保存的资源更多。仍然以网络数据包的传送为例。传统的方式，物理网卡每接收到一个数据帧就需要中断CPU，让CPU处理调用相应的中断服务函数处理数据帧。那么虚拟网卡也是如此，HOST每转发一个数据包到客户机的虚拟网卡，在不使用DMA的情况下就一个数据帧触发一个软中断，客户机就必须VM-exit处理中断，然后VM-entry，该过程不仅包含了数据的复制还包含了根模式非根模式之间的频繁切换。而即使qemu采用DMA的方式把数据帧直接写入到客户机内存，然后通知客户机，同样免不了数据复制带来的开销。

三、 virtIO的实现

基于上面描述的问题，virtIo给出了比较好的而解决方案。 而事实上，**virtIO**的出现不仅仅是解决了效率的问题。其更是为各大虚拟化引擎提供了一个统一的外部设备驱动。

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(21001)
2. PCI 设备详解一(15689)
3. 进程的挂起、阻塞和睡眠(13566)
4. Linux下桥接模式详解一(13365)
5. virtio后端驱动详解(10468)

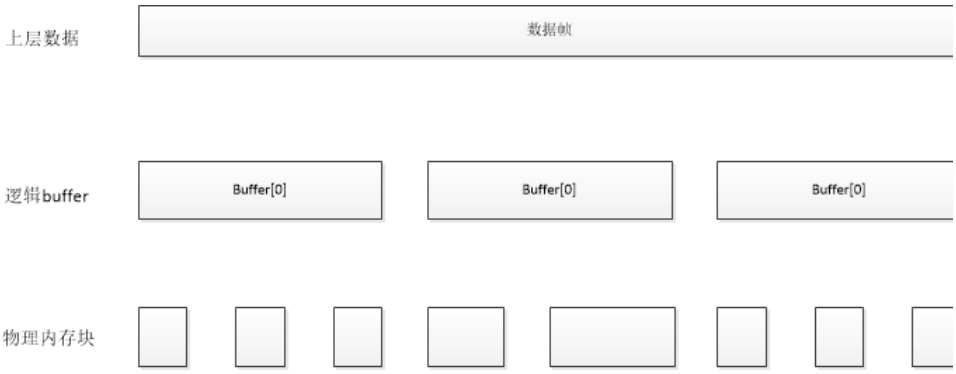
推荐排行榜

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)

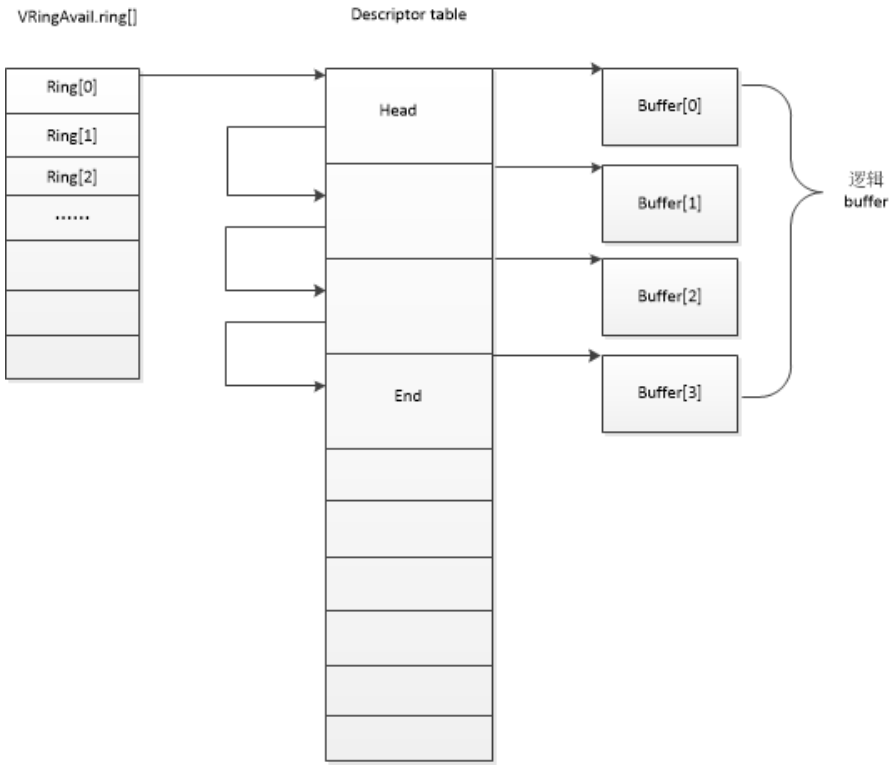
我们先从存储的角度分析一个数据帧。首先一个数据帧可能会需要多个buffer块才能完成存储；而一个buffer在这里指我们调用函数申请的虚拟地址空间，对应到物理内存可能包含有多个物理内存块。

qemu中用**VirtQueueElement**结构表示一个逻辑buffer，用**VRingDesc**结构描述一个物理内存块，用一个描述符数组集中管理所有的描述符。而前后端的配合通过两个ring来实现：**VRingAvail**和**VRingUsed**。当HOST需要向客户机发送数据时，先从对应的virtqueue获取客户机设置好的buffer空间（实际的buffer空间由客户机添加到virtqueue）,每次取出一个buffer，相关信息记录到**VirtQueueElement**结构中，然后对其进行地址映射，因为这里记录的buffer信息是客户机的物理地址，需要映射成HOST的虚拟地址才可以对其进行访问。每完成一个**VirtQueueElement** 即buffer的的写入,就需要记录**VirtQueueElement**相关信息到**VRingUsed**，并撤销地址映射。一个数据帧写入完成后会设置**VRingUsed**的idx字段并对客户机注入软件中断以通知客户机。

数据帧的逻辑存储结构如下：



而物理内存块由一个全局的描述符表统一管理，具体的管理如下图所示：



后端驱动工作模式正是如此，下面我们还是深入分析下几个重要的数据结构：

```
1 struct VirtQueue
2 {
3     VRing vring;//每个queue一个vring
4     hwaddr pa;//记录virtqueue中关联的描述符表的地址
5     /*last_avail_idx对应ring[]中的下标*/
6     uint16_t last_avail_idx;//上次写入的最后一个avail_ring的索引，下次给客户机发送的时候需要从avail_ring+1
7     /* Last used index value we have signalled on */
8     uint16_t signalled_used;
9
10    /* Last used index value we have signalled on */
```

```
11      bool signalled_used_valid;
12
13      /* Notification enabled? */
14      bool notification;
15
16      uint16_t queue_index;
17
18      int inuse;
19
20      uint16_t vector;
21      void (*handle_output)(VirtIODevice *vdev, VirtQueue *vq);
22      VirtIODevice *vdev;
23      EventNotifier guest_notifier;
24      EventNotifier host_notifier;
25  };
```

VirtQueue是一个虚拟队列，之所以称之为队列是从其管理buffer的角度。HOST和客户机也正是通过VirtQueue来操作buffer。每个buffer包含一个VRing结构，对buffer的操作实际上是通过VRing来管理的。pa是描述符表的物理地址。**last_avail_index**对应**VRingAvail**中ring[]数组的下标，表示上次最后使用的一个buffer首个desc对应的ring[]中的下标。这里听起来有点乱，么关系，后面会详细解释。暂且先介绍这几个。

```
1 typedef struct VRing
2 {
3     unsigned int num;//描述符表中表项的个数
4     unsigned int align;
5     hwaddr desc;//指向描述符表
6     hwaddr avail;//指向VRingAvail结构
7     hwaddr used;//指向VRingUsed结构
8 } VRing;
```

前面说过，**VRing**管理**buffer**，其实事实上，VRing是通过描述符表管理buffer的。究竟是怎么个管理法？这里num表示描述符表中的表项数。align是对其粒度。desc表示描述符表的物理地址。avail是**VRingAvail**的物理地址，而used是**VRingUsed**的物理地址。到这里是不是有点清楚了捏？？每一个描述符表项都对应一个物理块，参考下面的数据结构，每个表项都记录了其对应物理块的物理地址，长度，标志位，和next指针。同一buffer的不同物理块正是通过这个next指针连接起来的。现在应该比较清晰了吧！哈哈

```
1 typedef struct VRingDesc
2 {
3     uint64_t addr;//buffer 的地址
4     uint32_t len;//buffer的大小，需要注意当描述符作为节点连接一个描述符表时，描述符项的个数为
len/sizeof(VRingDesc)
5     uint16_t flags;
6     uint16_t next;
7 } VRingDesc;
```

```
 1 /*一个数据帧可能有多个VirtQueueElement，VirtQueueElement中的index */
 2 typedef struct VirtQueueElement
 3 {
 4     unsigned int index;
 5     unsigned int out_num;
 6     unsigned int in_num;
 7     /*in_addr和 out_addr保存的是客户机的物理地址，而in_sg和out_sg中的地址是host的虚拟地址，两者
之间需要映射*/
 8     hwaddr in_addr[VIRTQUEUE_MAX_SIZE];
 9     hwaddr out_addr[VIRTQUEUE_MAX_SIZE];
10     struct iovec in_sg[VIRTQUEUE_MAX_SIZE];
11     struct iovec out_sg[VIRTQUEUE_MAX_SIZE];
12 } VirtQueueElement;
```

再说这个VirtQueueElement，前面也说过其对应的是一个逻辑buffer块。而一个逻辑buffer块由多个物理内存块组成。index记录该逻辑buffer块的首个物理内存块对应的描述符在描述符表中的下标，out_num和in_num是指输出和输入块的数量。因为这里一个逻辑buffer可能包含可读区和可写区，即有些物理块是可读的而有些物理块是可写的，out_num记录可读块的数量，而in_num记录可写块的数量。in_addr是一个数组，记录的是可读块的物理地址，out_addr同理。但是由于物理地址是客户机的，所以要想在HOST访问，需要把这些地址映射成HOST的虚拟地址，下面两个就是保存的对应物理块在HOST的虚拟地址和长度。

```
typedef struct VRingAvail
{
    uint16_t flags;//限制host是否向客户机注入中断
    uint16_t idx;
    uint16_t ring[0];//这是一个索引数组，对应在描述符表中表项的下标，代表一个buffer的head，即一个buffer有多个description组成，其head会记录到
                     //ring数组中，使用的时候需要从ring数组中取出一个head才可以
} VRingAvail;

typedef struct VRingUsedElem
{
    uint32_t id;
    uint32_t len;//应该表示它代表的数据段的长度
} VRingUsedElem;

typedef struct VRingUsed
{
    uint16_t flags;//用于限制客户机是否增加buffer后是否通知host
    uint16_t idx;//
    VRingUsedElem ring[0];//意义同VRingAvail
} VRingUsed;
```

这三个放在一起说是因为这三个之间联系密切，而笔者也曾被这几个关系搞得晕头转向。先说VRingAvail和VRingUsed，两个字段基本一致，flags是标识位主要限制HOST和客户机的通知。VRing中的flags限制当HOST写入数据完成后是否向客户机注入中断，而VRingUsed中的flags限制当客户机增加buffer后，是否通知给HOST。这一点在高流量的情况下很有效。就像现在网络协议栈中的NAPI，为何采用中断加轮训而不是采用单纯的中断或者轮询。回到前面，二者也都有idx。VRingAvail中的idx表明客户机驱动下次添加buffer使用的ring下标，**VRingUsed**中的**idx**表明**qemu**下次添加**VRingUsedElem**使用的**ring**下标。

然后两者都有一个数组，**VRingAvail**中的ring数组记录的是可用buffer 的head index.即

if ring[0]=2

then desctable[2] 记录的就是一个逻辑buffer的首个物理块的信息。

virtqueue中的last_avail_idx记录ring[]数组中首个可用的buffer头部。即根据last_avail_idx查找ring[],根据ring[]数组得到desc表的下标。然后last_avail_idx++。

每次HOST向客户机发送数据就需要从这里获取一个**buffer head**。

当HOST完成数据的写入，可能会产生多个**VirtQueueElement**，即使用多个逻辑buffer，每个**VirtQueueElement**的信息记录到**VRingUsed**的**VRingUsedElem**数组中，一个元素对应一个**VRingUsedElem**结构，其中id记录对应**buffer**的**head**,**len**记录长度。

小结：上面结合重要的数据结构分析了**virtIO**后端驱动的工作模式，虽然笔者尽可能的想要分析清楚，但是总感觉表达能力有限，不足之处还请谅解！下面会结合**qemu**源代码就具体的网络数据包的接收，做简要的分析。

当然，开始还是从virtio_net_receive函数开始

```
static ssize_t virtio_net_receive(NetClientState *nc, const uint8_t *buf, size_t size)
{
    VirtIONet *n = qemu_get_nic_opaque(nc);
    VirtIONetQueue *q = virtio_net_get_subqueue(nc);
    VirtIODevice *vdev = VIRTIO_DEVICE(n);
    struct iovec mhdr_sg[VIRTQUEUE_MAX_SIZE];
    struct virtio_net_hdr_mrg_rxbuf mhdr;
    unsigned mhdr_cnt = 0;
    size_t offset, i, guest_offset;

    if (!virtio_net_can_receive(nc)) {
```

```
12          return -1;
13      }
14
15      /* hdr_len refers to the header we supply to the guest */
16      if (!virtio_net_has_buffers(q, size + n->guest_hdr_len - n->host_hdr_len)) {
17          return 0;
18      }
19
20      if (!receive_filter(n, buf, size))
21          return size;
22
23      offset = i = 0;
24      /*这里循环一次，就get到一个buffer*/
25      while (offset < size) {
26          VirtQueueElement elem;//一个elem表示一个buffer的数据
27          int len, total;
28          //sg作为一个指针，指向elem.in_sg，即sg=elem.in_sg。指向的是同一片内存区
29          const struct iovec *sg = elem.in_sg;
30
31          total = 0;
32          //从virtqueue中取出所有的描述符信息,返回的是in_sg和out_sg的数量总和，如果为0表示这队列并
没有实际的空间，无法装入数据
33          if (virtqueue_pop(q->rx_vq, &elem) == 0) {
34              if (i == 0)
35                  return -1;
36              error_report("virtio-net unexpected empty queue: "
37                      "i %zd mergeable %d offset %zd, size %zd, "
38                      "guest hdr len %zd, host hdr len %zd guest features 0x%x",
39                      i, n->mergeable_rx_bufs, offset, size,
40                      n->guest_hdr_len, n->host_hdr_len, vdev->guest_features);
41              exit(1);
42          }
43
44          if (elem.in_num < 1) {
45              error_report("virtio-net receive queue contains no in buffers");
46              exit(1);
47          }
48
49          if (i == 0) {
50              assert(offset == 0);
51              if (n->mergeable_rx_bufs) {
52                  mhdr_cnt = iov_copy(mhdr_sg, ARRAY_SIZE(mhdr_sg),
53                                      sg, elem.in_num,
54                                      offsetof(typeof(mhdr), num_buffers),
55                                      sizeof(mhdr.num_buffers));
56              }
57              receive_header(n, sg, elem.in_num, buf, size);
58              offset = n->host_hdr_len;
59              total += n->guest_hdr_len;
60              guest_offset = n->guest_hdr_len;
61          } else {
62              guest_offset = 0;
63          }
64          /* copy in packet.  ugh */
65          /*进行数据的写入*/
66          len = iov_from_buf(sg, elem.in_num, guest_offset,buf + offset, size - offset);
67          /*total表示完成复制的数据*/
68          total += len;
69          /*offset表示下一刻要复制的数据到buffer头的偏移*/
70          offset += len;
71          /* If buffers can't be merged, at this point we
72           * must have consumed the complete packet.
73           * Otherwise, drop it. */
74          if (!n->mergeable_rx_bufs && offset < size) {
75  #if 0
76
77  #endif
78              return size;
79          }
80
81          /* signal other side i 代表第几个buffer*/
82          /*到这里数据已经写入完成，需要撤销映射，更新ring的相关字段*/
83          virtqueue_fill(q->rx_vq, &elem, total, i++);
84      }
85
86      if (mhdr_cnt) {
87          stw_p(&mhdr.num_buffers, i);
88          iov_from_buf(mhdr_sg, mhdr_cnt,
89                       0,
```

```
90                      &mhdr.num_buffers, sizeof mhdr.num_buffers);
91      }
92      /*i表示数据包使用了多少个逻辑buffer即elem*/
93      virtqueue_flush(q->rx_vq, i);//这里会更新used_ring的idx，表明这些buffer 已经消费，可以回
收
94      virtio_notify(vdev, q->rx_vq);//通知客户机
95
96      return size;
97  }
```

参数想必不用过多解释，**NetClientState**代表当前网络接收端，buffer指向数据，size是数据的长度。前面的验证这里我们就暂且忽略了，重点从while循环开始，这里声明了一个局部变量**VirtQueueElement**用于从**virt_queue**中取元素，同时在循环体外边设置两个移动指针**offset**和i初始化成0.二者的意义后面自然明了，而循环体内部还有一个**toal**字段，我们最后一起说。往下看就调用了**virtqueue_pop**函数，该函数具体后面在分析，功能就是从队列中取出一个逻辑**buffer**，并记录相关信息到elem中。关于**VirtQueueElement**前面已经分析过；虾米那一个if是在开始写入数据之前，判断下是否支持合并**buffer**，是的话需要记录**buffer**的数量，并写入头部到**buffer**中（注意是最开始那个**buffer**块）。接着就要**iov_from_buf**函数往buffer块中写入数据了，每次按照一个逻辑buffer的大小进行写入直到写入数据结束。完成后调用**virtqueue_fill**函数，主要是撤销地址的映射并更新在**VRingUsed**中的**ring**[]映射，即获取一个**VRingUsedElem**，记录刚才完成写入的**VirtQueueElement**信息，主要是index和长度。就这样一直循环到写入数据结束。这样到最后就调用**virtqueue_flush**和**virtio_notify**函数更新**VRingUsed**中的**idx**并通知客户机。

大致处理流程是这样的，细节方面，我们看先如何从一个**virt_queue**中取出元素的：

```
1  int virtqueue_pop(VirtQueue *vq, VirtQueueElement *elem)
2  {
3      unsigned int i, head, max;
4      hwaddr desc_pa = vq->vring.desc;
5
6      if (!virtqueue_num_heads(vq, vq->last_avail_idx))
7          return 0;
8
9      /* When we start there are none of either input nor output. */
10     elem->out_num = elem->in_num = 0;
11     /*初始化成主描述表的表项数*/
12     max = vq->vring.num;
13     /*获取的是vringavail中的ring[avail]的值，对应描述符表中某个表项的下标*/
14     i = head = virtqueue_get_head(vq, vq->last_avail_idx++);
15
16     if (vq->vdev->guest_features & (1 << VIRTIO_RING_F_EVENT_IDX)) {
17         vring_avail_event(vq, vring_avail_idx(vq));//设置了
18         }
19     //如果描述符项指向的是另一个描述符数组
20     if (vring_desc_flags(desc_pa, i) & VRING_DESC_F_INDIRECT) {
21         //描述符数组的大小应该是VRingDesc大小的整数倍
22         if (vring_desc_len(desc_pa, i) % sizeof(VRingDesc)) {
23             error_report("Invalid size for indirect buffer table");
24             exit(1);
25         }
26
27         /* loop over the indirect descriptor table */
28         max = vring_desc_len(desc_pa, i) / sizeof(VRingDesc);
29         desc_pa = vring_desc_addr(desc_pa, i);//获取另一个描述符数组的地址
30         i = 0;//从第一个描述符项开始
31     }
32     /* Collect all the descriptors */
33     do {
34         struct iovec *sg;
35         //判断第i个描述符的属性
36         if (vring_desc_flags(desc_pa, i) & VRING_DESC_F_WRITE) {
37             if (elem->in_num >= ARRAY_SIZE(elem->in_sg)) {
38                 error_report("Too many write descriptors in indirect table");
39                 exit(1);
40             }
41             elem->in_addr[elem->in_num] = vring_desc_addr(desc_pa, i);
42             /**/
43             sg = &elem->in_sg[elem->in_num++];
44         } else {
45             if (elem->out_num >= ARRAY_SIZE(elem->out_sg)) {
46                 error_report("Too many read descriptors in indirect table");
47                 exit(1);
```

```
48              }
49              elem->out_addr[elem->out_num] = vring_desc_addr(desc_pa, i);
50              sg = &elem->out_sg[elem->out_num++];
51          }
52
53          sg->iov_len = vring_desc_len(desc_pa, i);
54
55          /* If we've got too many, that implies a descriptor loop. */
56          if ((elem->in_num + elem->out_num) > max) {
57              error_report("Looped descriptor");
58              exit(1);
59          }
60      } while ((i = virtqueue_next_desc(desc_pa, i, max)) != max);
61
62      /* Now map what we have collected */
63      /*现在获取到了客户机设置的avail space，现在需要map到host内存，然后写入数据*/
64      virtqueue_map_sg(elem->in_sg, elem->in_addr, elem->in_num, 1);
65      virtqueue_map_sg(elem->out_sg, elem->out_addr, elem->out_num, 0);
66      /*记录该数据段在整条数据帧中的索引*/
67      elem->index = head;
68      /*队列的使用元素加一*/
69      vq->inuse++;
70
71      trace_virtqueue_pop(vq, elem, elem->in_num, elem->out_num);
72      return elem->in_num + elem->out_num;
73  }
```

首先调用下函数**virtqueue_num_heads**检查下是否又可用的**buffer head**,然后初始化elem的**out_num**和**in_num**为0，接着以 **vq->last_avail_idx**为索引从**VRingAvail**的**ring**数组中获取一个**buffer head**索引，并赋值给i和**head**。**head**要作为**index**写入到**elem.index**,i作为一个计数器记录一共多少物理buffer，即多少个**desc**。接着判断客户机是否支持**VIRTIO_RING_F_EVENT_IDX**，对于这个看一下官方的解释：

```
1 /* The Guest publishes the used index for which it expects an interrupt
2  * at the end of the avail ring. Host should ignore the avail->flags field. */
3 /* The Host publishes the avail index for which it expects a kick
4  * at the end of the used ring. Guest should ignore the used->flags field. */
5 #define VIRTIO_RING_F_EVENT_IDX        29
```

简单点说，就是客户机希望**HOST**在使用完可用**buffer**后，中断虚拟机，忽略**avail->flags** ，同时HOST希望客户机使用完HOST添加的**used buffer**后，通知**HOST**，忽略 **used->flags**。

然后判断刚才获取的**head**指向的**desc**是否是一个**indirect**,是的话表明该描述符项指向一个单独的描述符表，此描述符表单独构成一个**buffer**。

然后从下面的**do**循环开始，就要开始获取各个物理**buffer**信息了，这里分为两类：可读和可写。可写的物理**buffer**地址（客户机物理地址）记录到**in_addr**数组中；可读的记录到**out_addr**数组中；并记录**in_num**和**out_num**；这样知道最后一个**desc**。获取完成后调用**virtqueue_map_sg**函数映射刚才我们获取到的各个物理buffer的地址，并记录到**in_sg**和**out_sg**数组中，这样才可以在HOST访问到。

这样经过**virtqueue_pop**函数，HOST已经获取了相应buffer的信息，下一步就是往buffer中写入数据了。

写入数据的过程就比较简单，这里就不赘述。那么写入数据完成后，调用的virtqueue_fill函数做了哪些工作呢?

```
1  void virtqueue_fill(VirtQueue *vq, const VirtQueueElement *elem,
2                      unsigned int len, unsigned int idx)
3  {
4      unsigned int offset;
5      int i;
6
7      trace_virtqueue_fill(vq, elem, len, idx);
8
9      offset = 0;
10     for (i = 0; i < elem->in_num; i++) {
11         size_t size = MIN(len - offset, elem->in_sg[i].iov_len);
12
13         cpu_physical_memory_unmap(elem->in_sg[i].iov_base,
14                                   elem->in_sg[i].iov_len,
15                                   1, size);
16
17         offset += size;
18     }
19
```

```
    for (i = 0; i < elem->out_num; i++)
        cpu_physical_memory_unmap(elem->out_sg[i].iov_base,
                                  elem->out_sg[i].iov_len,
                                  0, elem->out_sg[i].iov_len);
    /*idx表明这是第几个elem，一个elem代表一个buffer，而一个buffer会有一个description 的 head，这里idx对应head*/
    idx = (idx + vring_used_idx(vq)) % vq->vring.num;

    /* Get a pointer to the next entry in the used ring. */
    vring_used_ring_id(vq, idx, elem->index);
    vring_used_ring_len(vq, idx, len);
}
```

因为前面已经将数据写入，所以就HOST而言已经不需要再次访问前面映射的地址，那么这里就需要撤销映射以便于下次映射另外的物理buffer，同时需要在

VRingUsed的ring中添加元素映射我们使用过的elem，后面的**vring_used_ring_id**和**vring_used_ring_len**就是完成这样的工作。这样，待整条数据写入完成（可能使用多个elem即多个逻辑buffer），在最后调用**virtqueue_flush**函数集体更新**VRingUsed**的idx，最后调用**virtio_notify**通知客户机！！

---

virtIO和客户机Driver的通知机制：

这些需要结合具体的PCI设备架构， 之前有分析过PCI设备地址映射，这里简要说下，PCI设备的寄存器通过其BAR空间映射到IO地址空间或者内存地址空间，映射范围记录在BAR中，virtIO 作为一个PCI设备有几个控制寄存器或者理解成控制区域，大致布局为：

- **Common configuration**
- **Notifications**
- **ISR Status**
- **Device-specific configuration**
- **PCI configuration access**

在qemu中也都定义了对应的宏与之匹配

这里我们只关注ISR Status，它是一个32bit的寄存器，结构布局如下：

Bits	0	1	2 to 7
Purpose	Queue Interrupt	Device Configuration Interrupt	Reserved

当0位被设置的时候表明这是由virtQueue引起的中断

当1位被设置的时候表明这是因为配置变化引起的中断

在HOST完成数据写入，需要notify客户机时，调用前面提到的**virtio_notify**函数

```
void virtio_notify(VirtIODevice *vdev, VirtQueue *vq)
{
    if (!vring_notify(vdev, vq)) {
        return;
    }

    trace_virtio_notify(vdev, vq);
    vdev->isr |= 0x01;
    virtio_notify_vector(vdev, vq->vector);
}
```

可以看到这里设置了0位为1，然后调用**virtio_notify_vector**函数,**virtio_notify_vector**其实就是简单的调用了**virtio_pci_notify**函数，**virtio_pci_notify**这里根据不同的类型，采用不同的方式向客户机注入中断，这里主要由MSI-x和普通注入两种方式。

```
static void virtio_pci_notify(DeviceState *d, uint16_t vector)
{
    VirtIOPCIProxy *proxy = to_virtio_pci_proxy_fast(d);

    if (msix_enabled(&proxy->pci_dev))
        msix_notify(&proxy->pci_dev, vector);
    else {
        VirtIODevice *vdev = virtio_bus_get_device(&proxy->bus);
        pci_set_irq(&proxy->pci_dev, vdev->isr & 1);
    }
}
```

而客户机通知HOST就是想一个地址写入16位 的 **queue index**即可。比较关键的是**Queue Notify Address**的计算，这里我们下节介绍前端驱动的时候再讲！

后记：

由于笔者能力有限，文章中难免有错误或者不准确之处，若有老师发现，恳请指点！！谢谢！

参考：

1、**virtIO:Towards a De-Tacto Standard For Virtual I/O Devices**

2、**Virtual I/O Device (VIRTIO) Version 1.0**

2、**qemu1.7.1 源代码**

分类: KVM虚拟化技术, linux 内核源码分析

jack.chen
关注 - 12
粉丝 - 44
+加关注

« 上一篇： centos7手动编译安装Libvirt常见问题
» 下一篇： virtIO前后端notify机制详解

posted @ 2016-11-06 16:25 jack.chen Views(10468) Comments(5) Edit 收藏

Post Comment

#1楼 2017-08-02 21:52 | SimonZh

你好，VirtQueue数据结构下面的介绍中，每个buffer包含一个VRing结构是不是应该改成每个VirtQueue包含一个'ing结构。

支持(0) 反对(

#2楼 [楼主] 2017-08-10 18:29 | jack.chen

@ SimonZh
说"每个VirtQueue包含一个VRing结构"是没问题的，只是我想表达的意思是一个vring结构一次性记录一个逻辑buf r，仔细想想这么说的确容易造成误会，谢谢提醒！

支持(0) 反对(

#3楼 2017-11-09 22:32 | CheneyLin

每一个描述符表项都对应一个物理块？从图上看对应一个逻辑buffer吧？

支持(0) 反对(

#4楼 [楼主] 2017-11-10 13:12 | jack.chen

@ CheneyLin
一个逻辑buffer有多个物理块构成，每个表项对应一个物理块，图中的buffer[0]之类的只是为了显示一个逻辑buffe
的组成结构！谢谢

支持(0)　反对(

#5楼 2018-12-14 10:03 | dongyang626

在学习这块的知识。 特意回帖感谢博主～

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：
· virtio前端驱动详解
· virtio 简介
· DPDK数据包与内存专题-mempool内存池
· Linux设备驱动之mmap设备操作
· Linux字符设备驱动一一ioremap()函数解析
» 更多推荐...

最新 IT 新闻:
· MIUI 12发布：大幅改进界面效果 超40款机型支持升级
· 《我的世界》建筑团队还原迷宫都市欧拉丽 宏伟震撼
· .NET 5.0 Preview 3 发布
· 致癌的基因突变，可能比预想的来得更早
· 微软表示XSX已达成多项里程碑 将为玩家带来惊喜
» 更多新闻...

以马内利