

54 | 存储虚拟化（下）：如何建立自己保管的单独档案库？

2019-07-31 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 16:25 大小 15.05M



上一节，我们讲了 qemu 启动过程中的存储虚拟化。好了，现在 qemu 启动了，硬盘设备文件已经打开了。那如果我们要往虚拟机的一个进程写入一个文件，该怎么做呢？最终这个文件又是如何落到宿主机上的硬盘文件的呢？这一节，我们一起来看看。

前端设备驱动 virtio_blk

虚拟机里面的进程写入一个文件，当然要通过文件系统。整个过程和咱们在[文件系统](#)那一节讲的过程没有区别。只是到了设备驱动层，我们看到的就不是普通的硬盘驱动了，而是 virtio 的驱动。

virtio 的驱动程序代码在 Linux 操作系统的源代码里面，文件名叫 drivers/block/virtio_blk.c。

```

1 static int __init init(void)
2 {
3     int error;
4     virtblk_wq = alloc_workqueue("virtio-blk", 0, 0);
5     major = register_blkdev(0, "virtblk");
6     error = register_virtio_driver(&virtio_blk);
7     .....
8 }
9
10 module_init(init);
11 module_exit(fini);
12
13 MODULE_DEVICE_TABLE(virtio, id_table);
14 MODULE_DESCRIPTION("Virtio block driver");
15 MODULE_LICENSE("GPL");
16
17 static struct virtio_driver virtio_blk = {
18     .....
19     .driver.name           = KBUILD_MODNAME,
20     .driver.owner          = THIS_MODULE,
21     .id_table              = id_table,
22     .probe                 = virtblk_probe,
23     .remove                = virtblk_remove,
24     .....
25 };

```

前面我们介绍过设备驱动程序，从这里的代码中，我们能看到非常熟悉的结构。它会创建一个 workqueue，注册一个块设备，并获得一个主设备号，然后注册一个驱动函数 virtio_blk。

当一个设备驱动作为一个内核模块被初始化的时候，probe 函数会被调用，因而我们来看一下 virtblk_probe。

```

1 static int virtblk_probe(struct virtio_device *vdev)
2 {
3     struct virtio_blk *vblk;
4     struct request_queue *q;
5     .....
6     vdev->priv = vblk = kmalloc(sizeof(*vblk), GFP_KERNEL);
7     vblk->vdev = vdev;
8     vblk->sg_elems = sg_elems;
9     INIT_WORK(&vblk->config_work, virtblk_config_changed_work);

```

```

10 .....
11     err = init_vq(vblk);
12 .....
13     vblk->disk = alloc_disk(1 << PART_BITS);
14     memset(&vblk->tag_set, 0, sizeof(vblk->tag_set));
15     vblk->tag_set.ops = &virtio_mq_ops;
16     vblk->tag_set.queue_depth = virtblk_queue_depth;
17     vblk->tag_set.numa_node = NUMA_NO_NODE;
18     vblk->tag_set.flags = BLK_MQ_F_SHOULD_MERGE;
19     vblk->tag_set.cmd_size =
20         sizeof(struct virtblk_req) +
21         sizeof(struct scatterlist) * sg_elems;
22     vblk->tag_set.driver_data = vblk;
23     vblk->tag_set.nr_hw_queues = vblk->num_vqs;
24     err = blk_mq_alloc_tag_set(&vblk->tag_set);
25 .....
26     q = blk_mq_init_queue(&vblk->tag_set);
27     vblk->disk->queue = q;
28     q->queuedata = vblk;
29     virtblk_name_format("vd", index, vblk->disk->disk_name, DISK_NAME_LEN);
30     vblk->disk->major = major;
31     vblk->disk->first_minor = index_to_minor(index);
32     vblk->disk->private_data = vblk;
33     vblk->disk->fops = &virtblk_fops;
34     vblk->disk->flags |= GENHD_FL_EXT_DEVT;
35     vblk->index = index;
36 .....
37     device_add_disk(&vdev->dev, vblk->disk);
38     err = device_create_file(disk_to_dev(vblk->disk), &dev_attr_serial);
39 .....
40 }


```

在 virtblk_probe 中，我们首先看到的是 struct request_queue，这是每一个块设备都有一个的队列。还记得吗？它有两个函数，一个是 make_request_fn 函数，用于生成 request；另一个是 request_fn 函数，用于处理 request。

这个 request_queue 的初始化过程在 blk_mq_init_queue 中。它会调用 blk_mq_init_allocated_queue->blk_queue_make_request。在这里面，我们可以将 make_request_fn 函数设置为 blk_mq_make_request，也就是说，一旦上层有写入请求，我们就通过 blk_mq_make_request 这个函数，将请求放入 request_queue 队列中。

另外，在 virtblk_probe 中，我们会初始化一个 gendisk。前面我们也讲了，每一个块设备都有这样一个结构。


在 virtblk_probe 中，还有一件重要的事情就是，init_vq 会来初始化 virtqueue。

 复制代码

```
1 static int init_vq(struct virtio_blk *vblk)
2 {
3     int err;
4     int i;
5     vq_callback_t **callbacks;
6     const char **names;
7     struct virtqueue **vqs;
8     unsigned short num_vqs;
9     struct virtio_device *vdev = vblk->vdev;
10    .....
11    vblk->vqs = kmalloc_array(num_vqs, sizeof(*vblk->vqs), GFP_KERNEL);
12    names = kmalloc_array(num_vqs, sizeof(*names), GFP_KERNEL);
13    callbacks = kmalloc_array(num_vqs, sizeof(*callbacks), GFP_KERNEL);
14    vqs = kmalloc_array(num_vqs, sizeof(*vqs), GFP_KERNEL);
15    .....
16    for (i = 0; i < num_vqs; i++) {
17        callbacks[i] = virtblk_done;
18        names[i] = vblk->vqs[i].name;
19    }
20
21    /* Discover virtqueues and write information to configuration. */
22    err = virtio_find_vqs(vdev, num_vqs, vqs, callbacks, names, &desc);
23
24    for (i = 0; i < num_vqs; i++) {
25        vblk->vqs[i].vq = vqs[i];
26    }
27    vblk->num_vqs = num_vqs;
28    .....
29 }
```

按照上面的原理来说，virtqueue 是一个介于客户机前端和 qemu 后端的一个结构，用于在这两端之间传递数据。这里建立的 struct virtqueue 是客户机前端对于队列的管理的数据结构，在客户机的 linux 内核中通过 kmalloc_array 进行分配。

而队列的实体需要通过函数 virtio_find_vqs 查找或者生成，所以这里我们还把 callback 函数指定为 virtblk_done。当 buffer 使用发生变化时，我们需要调用这个 callback 函数进行通知。

 复制代码

```
1 static inline
```




```

2 int virtio_find_vqs(struct virtio_device *vdev, unsigned nvqs,
3                     struct virtqueue *vqs[], vq_callback_t *callbacks[],
4                     const char * const names[],
5                     struct irq_affinity *desc)
6 {
7     return vdev->config->find_vqs(vdev, nvqs, vqs, callbacks, names, NULL, desc);
8 }
9
10 static const struct virtio_config_ops virtio_pci_config_ops = {
11     .get          = vp_get,
12     .set          = vp_set,
13     .generation   = vp_generation,
14     .get_status   = vp_get_status,
15     .set_status   = vp_set_status,
16     .reset        = vp_reset,
17     .find_vqs     = vp_modern_find_vqs,
18     .del_vqs      = vp_del_vqs,
19     .get_features  = vp_get_features,
20     .finalize_features = vp_finalize_features,
21     .bus_name      = vp_bus_name,
22     .set_vq_affinity = vp_set_vq_affinity,
23     .get_vq_affinity = vp_get_vq_affinity,
24 };

```

根据 virtio_config_ops 的定义，virtio_find_vqs 会调用 vp_modern_find_vqs。


 复制代码

```

1 static int vp_modern_find_vqs(struct virtio_device *vdev, unsigned nvqs,
2                               struct virtqueue *vqs[],
3                               vq_callback_t *callbacks[],
4                               const char * const names[], const bool *ctx,
5                               struct irq_affinity *desc)
6 {
7     struct virtio_pci_device *vp_dev = to_vp_device(vdev);
8     struct virtqueue *vq;
9     int rc = vp_find_vqs(vdev, nvqs, vqs, callbacks, names, ctx, desc);
10    /* Select and activate all queues. Has to be done last: once we do
11     * this, there's no way to go back except reset.
12     */
13    list_for_each_entry(vq, &vdev->vqs, list) {
14        vp_iowrite16(vq->index, &vp_dev->common->queue_select);
15        vp_iowrite16(1, &vp_dev->common->queue_enable);
16    }
17
18    return 0;
19 }

```

在 `vp_modern_find_vqs` 中，`vp_find_vqs` 会调用 `vp_find_vqs_intx`。

 复制代码

```
1 static int vp_find_vqs_intx(struct virtio_device *vdev, unsigned nvqs,
2                             struct virtqueue *vqs[], vq_callback_t *callbacks[],
3                             const char * const names[], const bool *ctx)
4 {
5     struct virtio_pci_device *vp_dev = to_vp_device(vdev);
6     int i, err;
7
8     vp_dev->vqs = kcalloc(nvqs, sizeof(*vp_dev->vqs), GFP_KERNEL);
9     err = request_irq(vp_dev->pci_dev->irq, vp_interrupt, IRQF_SHARED,
10                      dev_name(&vdev->dev), vp_dev);
11     vp_dev->intx_enabled = 1;
12     vp_dev->per_vq_vectors = false;
13     for (i = 0; i < nvqs; ++i) {
14         vqs[i] = vp_setup_vq(vdev, i, callbacks[i], names[i],
15                             ctx ? ctx[i] : false,
16                             VIRTIO_MSI_NO_VECTOR);
17         .....
18     }
19 }
```

在 `vp_find_vqs_intx` 中，我们通过 `request_irq` 注册一个中断处理函数 `vp_interrupt`，当设备的配置信息发生改变，会产生一个中断，当设备向队列中写入信息时，也会会产生一个中断，我们称为 `vq` 中断，中断处理函数需要调用相应的队列的回调函数。

然后，我们根据队列的数目，依次调用 `vp_setup_vq`，完成 `virtqueue`、`vring` 的分配和初始化。

 复制代码

```
1 static struct virtqueue *vp_setup_vq(struct virtio_device *vdev, unsigned index,
2                                       void (*callback)(struct virtqueue *vq),
3                                       const char *name,
4                                       bool ctx,
5                                       u16 msix_vec)
6 {
7     struct virtio_pci_device *vp_dev = to_vp_device(vdev);
8     struct virtio_pci_vq_info *info = kmalloc(sizeof *info, GFP_KERNEL);
9     struct virtqueue *vq;
10     unsigned long flags;
```

```

11 .....
12     vq = vp_dev->setup_vq(vp_dev, info, index, callback, name, ctx,
13                           msix_vec);
14     info->vq = vq;
15     if (callback) {
16         spin_lock_irqsave(&vp_dev->lock, flags);
17         list_add(&info->node, &vp_dev->virtqueues);
18         spin_unlock_irqrestore(&vp_dev->lock, flags);
19     } else {
20         INIT_LIST_HEAD(&info->node);
21     }
22     vp_dev->vqs[index] = info;
23     return vq;
24 }
25
26 static struct virtqueue *setup_vq(struct virtio_pci_device *vp_dev,
27                                   struct virtio_pci_vq_info *info,
28                                   unsigned index,
29                                   void (*callback)(struct virtqueue *vq),
30                                   const char *name,
31                                   bool ctx,
32                                   u16 msix_vec)
33 {
34     struct virtio_pci_common_cfg __iomem *cfg = vp_dev->common;
35     struct virtqueue *vq;
36     u16 num, off;
37     int err;
38
39     /* Select the queue we're interested in */
40     vp_iowrite16(index, &cfg->queue_select);
41
42     /* Check if queue is either not available or already active. */
43     num = vp_ioread16(&cfg->queue_size);
44
45     /* get offset of notification word for this vq */
46     off = vp_ioread16(&cfg->queue_notify_off);
47
48     info->msix_vector = msix_vec;
49
50     /* create the vring */
51     vq = vring_create_virtqueue(index, num,
52                                 SMP_CACHE_BYTES, &vp_dev->vdev,
53                                 true, true, ctx,
54                                 vp_notify, callback, name);
55
56     /* activate the queue */
57     vp_iowrite16(vring_get_vring_size(vq), &cfg->queue_size);
58     vp_iowrite64_twopart(vring_get_desc_addr(vq),
59                          &cfg->queue_desc_lo, &cfg->queue_desc_hi);
60     vp_iowrite64_twopart(vring_get_avail_addr(vq),
61                          &cfg->queue_avail_lo, &cfg->queue_avail_hi);
62     vp_iowrite64_twopart(vring_get_used_addr(vq),
63                          &cfg->queue_used_lo, &cfg->queue_used_hi);

```


```

63 .....
64     return vq;
65 }
66
67 struct virtqueue *vring_create_virtqueue(
68     unsigned int index,
69     unsigned int num,
70     unsigned int vring_align,
71     struct virtio_device *vdev,
72     bool weak_barriers,
73     bool may_reduce_num,
74     bool context,
75     bool (*notify)(struct virtqueue *),
76     void (*callback)(struct virtqueue *),
77     const char *name)
78 {
79     struct virtqueue *vq;
80     void *queue = NULL;
81     dma_addr_t dma_addr;
82     size_t queue_size_in_bytes;
83     struct vring vring;
84
85     /* TODO: allocate each queue chunk individually */
86     for (; num && vring_size(num, vring_align) > PAGE_SIZE; num /= 2) {
87         queue = vring_alloc_queue(vdev, vring_size(num, vring_align),
88                                 &dma_addr,
89                                 GFP_KERNEL | __GFP_NOWARN | __GFP_ZERO);
90         if (queue)
91             break;
92     }
93
94     if (!queue) {
95         /* Try to get a single page. You are my only hope! */
96         queue = vring_alloc_queue(vdev, vring_size(num, vring_align),
97                                 &dma_addr, GFP_KERNEL | __GFP_ZERO);
98     }
99
100     queue_size_in_bytes = vring_size(num, vring_align);
101     vring_init(&vring, num, queue, vring_align);
102
103     vq = __vring_new_virtqueue(index, vring, vdev, weak_barriers, context, notify,
104
105     to_vvq(vq)->queue_dma_addr = dma_addr;
106     to_vvq(vq)->queue_size_in_bytes = queue_size_in_bytes;
107     to_vvq(vq)->we_own_ring = true;
108
109     return vq;
110 }

```


在 `vring_create_virtqueue` 中，我们会调用 `vring_alloc_queue`，来创建队列所需要的内存空间，然后调用 `vring_init` 初始化结构 `struct vring`，来管理队列的内存空间，调用 `__vring_new_virtqueue`，来创建 `struct vring_virtqueue`。

这个结构的一开始，是 `struct virtqueue`，它也是 `struct virtqueue` 的一个扩展，紧接着后面就是 `struct vring`。

 复制代码

```
1 struct vring_virtqueue {
2     struct virtqueue vq;
3
4     /* Actual memory layout for this queue */
5     struct vring vring;
6     .....
7 }
```

至此我们发现，虚拟机里面的 `virtio` 的前端是这样的结构：`struct virtio_device` 里面有一个 `struct vring_virtqueue`，在 `struct vring_virtqueue` 里面有一个 `struct vring`。

中间 `virtio` 队列的管理

还记不记得我们上面讲 `qemu` 初始化的时候，`virtio` 的后端有数据结构 `VirtIODevice`，`VirtQueue` 和 `vring` 一模一样，前端和后端对应起来，都应该指向刚才创建的那一段内存。


现在的问题是，我们刚才分配的内存存在客户机的内核里面，如何告知 `qemu` 来访问这段内存呢？

别忘了，`qemu` 模拟出来的 `virtio block device` 只是一个 `PCI` 设备。对于客户机来讲，这是一个外部设备，我们可以通过给外部设备发送指令的方式告知外部设备，这就是代码中 `vp_iowrite16` 的作用。它会调用专门给外部设备发送指令的函数 `iowrite`，告诉外部的 `PCI` 设备。

告知的有三个地址 `virtqueue_get_desc_addr`、`virtqueue_get_avail_addr`，`virtqueue_get_used_addr`。从客户机角度来看，这里面的地址都是物理地址，也即

GPA (Guest Physical Address)。因为只有物理地址才是客户机和 qemu 程序都认可的地址，本来客户机的物理内存也是 qemu 模拟出来的。


在 qemu 中，对 PCI 总线添加一个设备的时候，我们会调用 virtio_pci_device_plugged。

 复制代码

```
1 static void virtio_pci_device_plugged(DeviceState *d, Error **errp)
2 {
3     VirtIOPCIProxy *proxy = VIRTIO_PCI(d);
4     .....
5     memory_region_init_io(&proxy->bar, OBJECT(proxy),
6                           &virtio_pci_config_ops,
7                           proxy, "virtio-pci", size);
8     .....
9 }
10
11 static const MemoryRegionOps virtio_pci_config_ops = {
12     .read = virtio_pci_config_read,
13     .write = virtio_pci_config_write,
14     .impl = {
15         .min_access_size = 1,
16         .max_access_size = 4,
17     },
18     .endianness = DEVICE_LITTLE_ENDIAN,
19 };
```

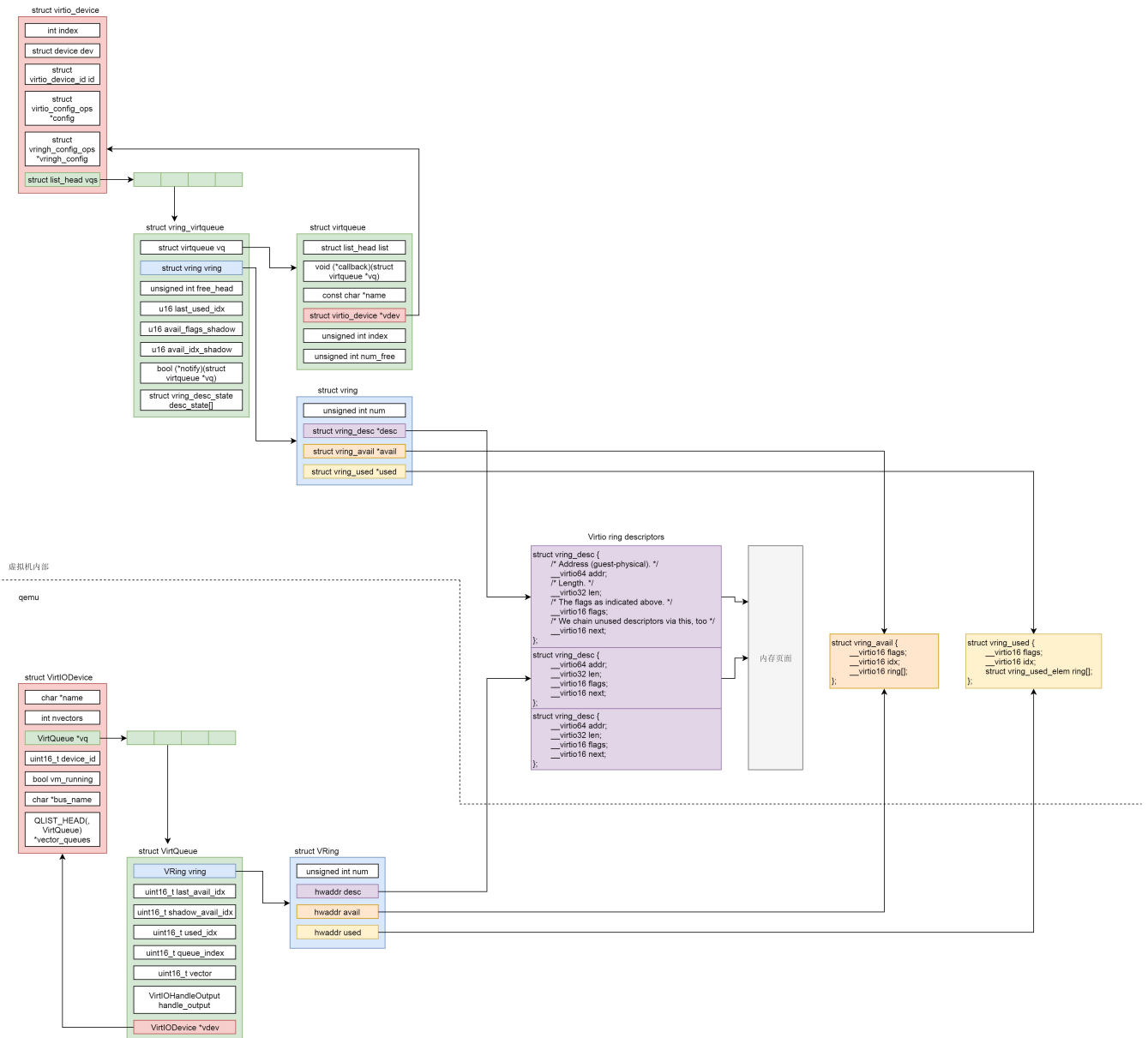
在这里面，对于这个加载的设备进行 I/O 操作，会映射到读写某一块内存空间，对应的操作为 virtio_pci_config_ops，也即写入这块内存空间，这就相当于对于这个 PCI 设备进行某种配置。

对 PCI 设备进行配置的时候，会有这样的调用链：virtio_pci_config_write->virtio_ioport_write->virtio_queue_set_addr。设置 virtio 的 queue 的地址是一项很重要的操作。

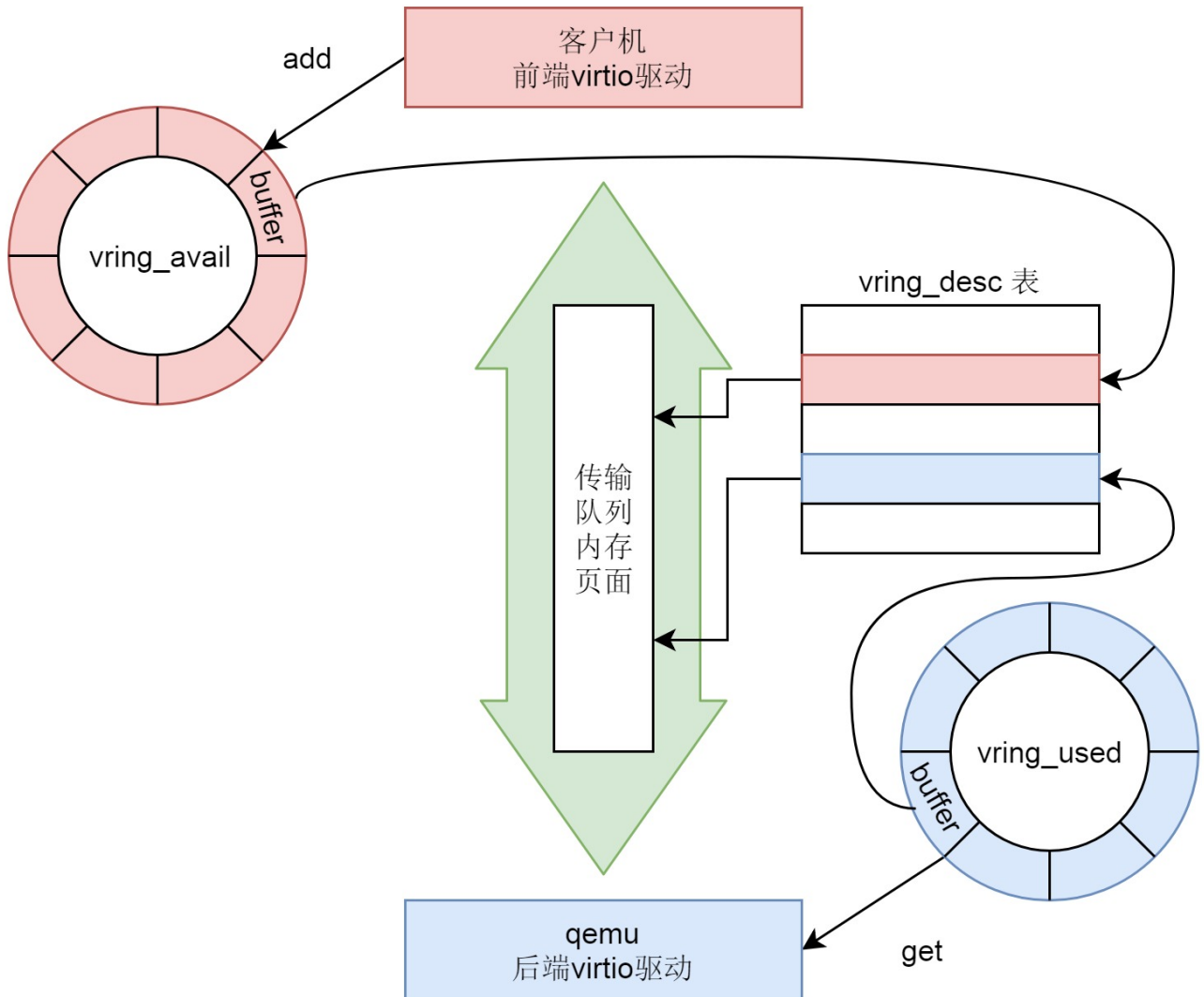
 复制代码

```
1 void virtio_queue_set_addr(VirtIODevice *vdev, int n, hwaddr addr)
2 {
3     vdev->vq[n].vring.desc = addr;
4     virtio_queue_update_rings(vdev, n);
5 }
```

从这里我们可以看出，qemu 后端的 VirtIODevice 的 VirtQueue 的 vring 的地址，被设置成了刚才给队列分配的内存的 GPA。



接着，我们来看一下这个队列的格式。



复制代码

```

1 /* Virtio ring descriptors: 16 bytes. These can chain together via "next". */
2 struct vring_desc {
3     /* Address (guest-physical). */
4     __virtio64 addr;
5     /* Length. */
6     __virtio32 len;
7     /* The flags as indicated above. */
8     __virtio16 flags;
9     /* We chain unused descriptors via this, too */
10    __virtio16 next;
11 };
12
13 struct vring_avail {
14     __virtio16 flags;
15     __virtio16 idx;
16     __virtio16 ring[];
17 };
18
19 /* u32 is used here for ids for padding reasons. */
20 struct vring_used_elem {

```

```

21      /* Index of start of used descriptor chain. */
22      __virtio32 id;
23      /* Total length of the descriptor chain which was used (written to) */
24      __virtio32 len;
25 };
26
27 struct vring_used {
28     __virtio16 flags;
29     __virtio16 idx;
30     struct vring_used_elem ring[];
31 };
32
33 struct vring {
34     unsigned int num;
35
36     struct vring_desc *desc;
37
38     struct vring_avail *avail;
39
40     struct vring_used *used;
41 };

```

vring 包含三个成员：

vring_desc 指向分配的内存块，用于存放客户机和 qemu 之间传输的数据。


avail->ring[] 是发送端维护的环形队列，指向需要接收端处理的 vring_desc。

used->ring[] 是接收端维护的环形队列，指向自己已经处理过了的 vring_desc。

数据写入的流程

接下来，我们来看，真的写入一个数据的时候，会发生什么。

按照上面 virtio 驱动初始化的时候的逻辑，blk_mq_make_request 会被调用。这个函数比较复杂，会分成多个分支，但是最终都会调用到 request_queue 的 virtio_mq_ops 的 queue_rq 函数。

 复制代码

```

1 struct request_queue *q = rq->q;
2 q->mq_ops->queue_rq(hctx, &bd);
3
4 static const struct blk_mq_ops virtio_mq_ops = {


```

```

5      .queue_rq      = virtio_queue_rq,
6      .complete      = virtblk_request_done,
7      .init_request   = virtblk_init_request,
8      .map_queues     = virtblk_map_queues,
9  };

```

根据 `virtio_mq_ops` 的定义，我们现在要调用 `virtio_queue_rq`。

 复制代码


```

1  static blk_status_t virtio_queue_rq(struct blk_mq_hw_ctx *hctx,
2                                     const struct blk_mq_queue_data *bd)
3  {
4      struct virtio_blk *vblk = hctx->queue->queuedata;
5      struct request *req = bd->rq;
6      struct virtblk_req *vbr = blk_mq_rq_to_pdu(req);
7      .....
8      err = virtblk_add_req(vblk->vqs[qid].vq, vbr, vbr->sg, num);
9      .....
10     if (notify)
11         virtqueue_notify(vblk->vqs[qid].vq);
12     return BLK_STS_OK;
13 }

```

在 `virtio_queue_rq` 中，我们会将请求写入的数据，通过 `virtblk_add_req` 放入 `struct virtqueue`。

因此，接下来的调用链为：`virtblk_add_req->virtqueue_add_sgs->virtqueue_add`。

 复制代码

```

1  static inline int virtqueue_add(struct virtqueue *_vq,
2                                 struct scatterlist *sgs[],
3                                 unsigned int total_sg,
4                                 unsigned int out_sgs,
5                                 unsigned int in_sgs,
6                                 void *data,
7                                 void *ctx,
8                                 gfp_t gfp)
9  {
10     struct vring_virtqueue *vq = to_vvq(_vq);
11     struct scatterlist *sg;
12     struct vring_desc *desc;

```



```

13     unsigned int i, n, avail, descs_used, uninitialized_var(prev), err_idx;
14     int head;
15     bool indirect;
16     .....
17     head = vq->free_head;
18
19     indirect = false;
20     desc = vq->vring.desc;
21     i = head;
22     descs_used = total_sg;
23
24     for (n = 0; n < out_sgs; n++) {
25         for (sg = sgs[n]; sg; sg = sg_next(sg)) {
26             dma_addr_t addr = vring_map_one_sg(vq, sg, DMA_TO_DEVICE);
27             .....
28                 desc[i].flags = cpu_to_virtio16(_vq->vdev, VRING_DESC_F_NEXT);
29                 desc[i].addr = cpu_to_virtio64(_vq->vdev, addr);
30                 desc[i].len = cpu_to_virtio32(_vq->vdev, sg->length);
31                 prev = i;
32                 i = virtio16_to_cpu(_vq->vdev, desc[i].next);
33             }
34     }
35
36     /* Last one doesn't continue. */
37     desc[prev].flags &= cpu_to_virtio16(_vq->vdev, ~VRING_DESC_F_NEXT);
38
39     /* We're using some buffers from the free list. */
40     vq->vq.num_free -= descs_used;
41
42     /* Update free pointer */
43     vq->free_head = i;
44
45     /* Store token and indirect buffer state. */
46     vq->desc_state[head].data = data;
47
48     /* Put entry in available array (but don't update avail->idx until they do sync
49     avail = vq->avail_idx_shadow & (vq->vring.num - 1);
50     vq->vring.avail->ring[avail] = cpu_to_virtio16(_vq->vdev, head);
51
52     /* Descriptors and available array need to be set before we expose the new avail
53     virtio_wmb(vq->weak_barriers);
54     vq->avail_idx_shadow++;
55     vq->vring.avail->idx = cpu_to_virtio16(_vq->vdev, vq->avail_idx_shadow);
56     vq->num_added++;
57     .....
58     return 0;
59 }

```


在 `virtqueue_add` 函数中，我们能看到，`free_head` 指向的整个内存块空闲链表的起始位置，用 `head` 变量记住这个起始位置。

接下来，`i` 也指向这个起始位置，然后是一个 `for` 循环，将数据放到内存块里面，放的过程中，`next` 不断指向下一个空闲位置，这样空闲的内存块被不断的占用。等所有的写入都结束了，`i` 就会指向这次存放的内存块的下一个空闲位置，然后 `free_head` 就指向 `i`，因为前面的都填满了。

至此，从 `head` 到 `i` 之间的内存块，就是这次写入的全部数据。

于是，在 `vring` 的 `avail` 变量中，在 `ring[]` 数组中分配新的一项，在 `avail` 的位置，`avail` 的计算是 `avail_idx_shadow & (vq->vring.num - 1)`，其中，`avail_idx_shadow` 是上一次的 `avail` 的位置。这里如果超过了 `ring[]` 数组的下标，则重新跳到起始位置，就说明是一个环。这次分配的新的 `avail` 的位置就存放新写入的从 `head` 到 `i` 之间的内存块。然后是 `avail_idx_shadow++`，这说明这一块内存可以被接收方读取了。

接下来，我们回到 `virtio_queue_rq`，调用 `virtqueue_notify` 通知接收方。而 `virtqueue_notify` 会调用 `vp_notify`。

 复制代码

```
1 bool vp_notify(struct virtqueue *vq)
2 {
3     /* we write the queue's selector into the notification register to
4      * signal the other end */
5     iowrite16(vq->index, (void __iomem *)vq->priv);
6     return true;
7 }
```

然后，我们写入一个 I/O 会触发 VM exit。我们在解析 CPU 的时候看到过这个逻辑。

 复制代码


```
1 int kvm_cpu_exec(CPUState *cpu)
2 {
3     struct kvm_run *run = cpu->kvm_run;
4     int ret, run_ret;
5     .....
6     run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
7     .....
```

```

8      switch (run->exit_reason) {
9          case KVM_EXIT_IO:
10             DPRINTF("handle_io\n");
11             /* Called outside BQL */
12             kvm_handle_io(run->io.port, attrs,
13                           (uint8_t *)run + run->io.data_offset,
14                           run->io.direction,
15                           run->io.size,
16                           run->io.count);
17             ret = 0;
18             break;
19         }
20         .....
21     }

```

这次写入的也是一个 I/O 的内存空间，同样会触发 `virtio_ioport_write`，这次会调用 `virtio_queue_notify`。

 复制代码


```

1 void virtio_queue_notify(VirtIODevice *vdev, int n)
2 {
3     VirtQueue *vq = &vdev->vq[n];
4     .....
5     if (vq->handle_aio_output) {
6         event_notifier_set(&vq->host_notifier);
7     } else if (vq->handle_output) {
8         vq->handle_output(vdev, vq);
9     }
10 }

```

`virtio_queue_notify` 会调用 `VirtQueue` 的 `handle_output` 函数，前面我们已经设置过这个函数了，是 `virtio_blk_handle_output`。

接下来的调用链为：`virtio_blk_handle_output->virtio_blk_handle_output_do->virtio_blk_handle_vq`。

 复制代码

```

1 bool virtio_blk_handle_vq(VirtIOBlock *s, VirtQueue *vq)
2 {
3     VirtIOBlockReq *req;
4     MultiReqBuffer mrb = {};

```

```

5     bool progress = false;
6     .....
7     do {
8         virtio_queue_set_notification(vq, 0);
9
10        while ((req = virtio_blk_get_request(s, vq))) {
11            progress = true;
12            if (virtio_blk_handle_request(req, &mrbs)) {
13                virtqueue_detach_element(req->vq, &req->elem, 0);
14                virtio_blk_free_request(req);
15                break;
16            }
17        }
18
19        virtio_queue_set_notification(vq, 1);
20    } while (!virtio_queue_empty(vq));
21
22    if (mrbs.num_reqs) {
23        virtio_blk_submit_multireq(s->blk, &mrbs);
24    }
25    .....
26    return progress;
27 }

```

在 `virtio_blk_handle_vq` 中，有一个 `while` 循环，在循环中调用函数 `virtio_blk_get_request` 从 `vq` 中取出请求，然后调用 `virtio_blk_handle_request` 处理从 `vq` 中取出的请求。

我们先来看 `virtio_blk_get_request`。

 复制代码

```

1 static VirtIOBlockReq *virtio_blk_get_request(VirtIOBlock *s, VirtQueue *vq)
2 {
3     VirtIOBlockReq *req = virtqueue_pop(vq, sizeof(VirtIOBlockReq));
4
5     if (req) {
6         virtio_blk_init_request(s, vq, req);
7     }
8     return req;
9 }
10
11 void *virtqueue_pop(VirtQueue *vq, size_t sz)
12 {
13     unsigned int i, head, max;
14     VRingMemoryRegionCaches *caches;

```

```

15     MemoryRegionCache *desc_cache;
16     int64_t len;
17     VirtIODevice *vdev = vq->vdev;
18     VirtQueueElement *elem = NULL;
19     unsigned out_num, in_num, elem_entries;
20     hwaddr addr[VIRTQUEUE_MAX_SIZE];
21     struct iovec iov[VIRTQUEUE_MAX_SIZE];
22     VRingDesc desc;
23     int rc;
24     .....
25     /* When we start there are none of either input nor output. */
26     out_num = in_num = elem_entries = 0;
27
28     max = vq->vring.num;
29
30     i = head;
31
32     caches = vring_get_region_caches(vq);
33     desc_cache = &caches->desc;
34     vring_desc_read(vdev, &desc, desc_cache, i);
35     .....
36     /* Collect all the descriptors */
37     do {
38         bool map_ok;
39
40         if (desc.flags & VRING_DESC_F_WRITE) {
41             map_ok = virtqueue_map_desc(vdev, &in_num, addr + out_num,
42                                         iov + out_num,
43                                         VIRTQUEUE_MAX_SIZE - out_num, true,
44                                         desc.addr, desc.len);
45         } else {
46             map_ok = virtqueue_map_desc(vdev, &out_num, addr, iov,
47                                         VIRTQUEUE_MAX_SIZE, false,
48                                         desc.addr, desc.len);
49         }
50     } while (rc == VIRTQUEUE_READ_DESC_MORE);
51     .....
52     rc = virtqueue_read_next_desc(vdev, &desc, desc_cache, max, &i);
53     .....
54     /* Now copy what we have collected and mapped */
55     elem = virtqueue_alloc_element(sz, out_num, in_num);
56     elem->index = head;
57     for (i = 0; i < out_num; i++) {
58         elem->out_addr[i] = addr[i];
59         elem->out_sg[i] = iov[i];
60     }
61     for (i = 0; i < in_num; i++) {
62         elem->in_addr[i] = addr[out_num + i];
63         elem->in_sg[i] = iov[out_num + i];
64     }
65
66     vq->inuse++;

```

```

67 .....
68     return elem;
69 }

```

我们可以看到，`virtio_blk_get_request` 会调用 `virtqueue_pop`。在这里面，我们能看到对于 `vring` 的操作，也即从这里面将客户机里面写入的数据读取出来，放到 `VirtIOBlockReq` 结构中。

接下来，我们就要调用 `virtio_blk_handle_request` 处理这些数据。所以接下来的调用链为：`virtio_blk_handle_request->virtio_blk_submit_multireq->submit_requests`。

 复制代码

```


1 static inline void submit_requests(BlockBackend *blk, MultiReqBuffer *mrbs, int start, int num_reqs) {
2 {
3     QEMUIOVector *qiov = &mrbs->reqs[start]->qiov;
4     int64_t sector_num = mrbs->reqs[start]->sector_num;
5     bool is_write = mrbs->is_write;
6
7     if (num_reqs > 1) {
8         int i;
9         struct iovec *tmp_iov = qiov->iov;
10        int tmp_niov = qiov->niov;
11        qemu_iovec_init(qiov, niov);
12
13        for (i = 0; i < tmp_niov; i++) {
14            qemu_iovec_add(qiov, tmp_iov[i].iov_base, tmp_iov[i].iov_len);
15        }
16
17        for (i = start + 1; i < start + num_reqs; i++) {
18            qemu_iovec_concat(qiov, &mrbs->reqs[i]->qiov, 0,
19                             mrbs->reqs[i]->qiov.size);
20            mrbs->reqs[i - 1]->mr_next = mrbs->reqs[i];
21        }
22
23        block_acct_merge_done(blk_get_stats(blk),
24                              is_write ? BLOCK_ACCT_WRITE : BLOCK_ACCT_READ,
25                              num_reqs - 1);
26    }
27
28    if (is_write) {
29        blk_aio_pwritev(blk, sector_num << BDRV_SECTOR_BITS, qiov, 0,
30                       virtio_blk_rw_complete, mrbs->reqs[start]);
31    } else {
32        blk_aio_preadv(blk, sector_num << BDRV_SECTOR_BITS, qiov, 0,
33                      virtio_blk_rw_complete, mrbs->reqs[start]);
34    }
35 }

```



```
34     }
35 }
```

在 submit_requests 中，我们看到了 BlockBackend。这是在 qemu 启动的时候，打开 qcow2 文件的时候生成的，现在我们可以用它来写入文件了，调用的是 blk_aio_pwritev。

 复制代码

```
1 BlockAIOCB *blk_aio_pwritev(BlockBackend *blk, int64_t offset,
2                             QEMUIOVector *qiov, BdrvRequestFlags flags,
3                             BlockCompletionFunc *cb, void *opaque)
4 {
5     return blk_aio_prvw(blk, offset, qiov->size, qiov,
6                         blk_aio_write_entry, flags, cb, opaque);
7 }
8
9 static BlockAIOCB *blk_aio_prvw(BlockBackend *blk, int64_t offset, int bytes,
10                                void *iobuf, CoroutineEntry co_entry,
11                                BdrvRequestFlags flags,
12                                BlockCompletionFunc *cb, void *opaque)
13 {
14     BlkAioEmAIOCB *acb;
15     Coroutine *co;
16     acb = blk_aio_get(&blk_aio_em_aiocb_info, blk, cb, opaque);
17     acb->rwco = (BlkRwCo) {
18         .blk      = blk,
19         .offset   = offset,
20         .iobuf    = iobuf,
21         .flags    = flags,
22         .ret      = NOT_DONE,
23     };
24     acb->bytes = bytes;
25     acb->has_returned = false;
26
27     co = qemu_coroutine_create(co_entry, acb);
28     bdrv_coroutine_enter(blk_bs(blk), co);
29
30     acb->has_returned = true;
31     return &acb->common;
32 }
```

在 blk_aio_pwritev 中，我们看到，又是创建了一个协程来进行写入。写入完毕之后调用 virtio_blk_rw_complete->virtio_blk_req_complete。

```

1 static void virtio_blk_req_complete(VirtIOBlockReq *req, unsigned char status)
2 {
3     VirtIOBlock *s = req->dev;
4     VirtIODevice *vdev = VIRTIO_DEVICE(s);
5
6     trace_virtio_blk_req_complete(vdev, req, status);
7
8     stb_p(&req->in->status, status);
9     virtqueue_push(req->vq, &req->elem, req->in_len);
10    virtio_notify(vdev, req->vq);
11 }

```

在 `virtio_blk_req_complete` 中，我们先是调用 `virtqueue_push`，更新 `vring` 中 `used` 变量，表示这部分已经写入完毕，空间可以回收利用了。但是，这部分的改变仅仅改变了 `qemu` 后端的 `vring`，我们还需要通知客户机中 `virtio` 前端的 `vring` 的值，因而要调用 `virtio_notify`。`virtio_notify` 会调用 `virtio_irq` 发送一个中断。

还记得咱们前面注册过一个中断处理函数 `vp_interrupt` 吗？它就是干这个事情的。

```

1 static irqreturn_t vp_interrupt(int irq, void *opaque)
2 {
3     struct virtio_pci_device *vp_dev = opaque;
4     u8 isr;
5
6     /* reading the ISR has the effect of also clearing it so it's very
7      * important to save off the value. */
8     isr = ioread8(vp_dev->isr);
9
10    /* Configuration change? Tell driver if it wants to know. */
11    if (isr & VIRTIO_PCI_ISR_CONFIG)
12        vp_config_changed(irq, opaque);
13
14    return vp_vring_interrupt(irq, opaque);
15 }

```

就像前面说的一样 `vp_interrupt` 这个中断处理函数，一是处理配置变化，二是处理 I/O 结束。第二种的调用链为：`vp_interrupt->vp_vring_interrupt->vring_interrupt`。

```

1 irqreturn_t vring_interrupt(int irq, void *_vq)
2 {
3     struct vring_virtqueue *vq = to_vvq(_vq);
4     .....
5     if (vq->vq.callback)
6         vq->vq.callback(&vq->vq);
7
8     return IRQ_HANDLED;
9 }

```

在 `vring_interrupt` 中，我们会调用 `callback` 函数，这个也是在前面注册过的，是 `virtblk_done`。

接下来的调用链为：`virtblk_done->virtqueue_get_buf->virtqueue_get_buf_ctx`。

```

1 void *virtqueue_get_buf_ctx(struct virtqueue *_vq, unsigned int *len,
2                             void **ctx)
3 {
4     struct vring_virtqueue *vq = to_vvq(_vq);
5     void *ret;
6     unsigned int i;
7     u16 last_used;
8     .....
9     last_used = (vq->last_used_idx & (vq->vring.num - 1));
10    i = virtio32_to_cpu(_vq->vdev, vq->vring.used->ring[last_used].id);
11    *len = virtio32_to_cpu(_vq->vdev, vq->vring.used->ring[last_used].len);
12    .....
13    /* detach_buf clears data, so grab it now. */
14    ret = vq->desc_state[i].data;
15    detach_buf(vq, i, ctx);
16    vq->last_used_idx++;
17    .....
18    return ret;
19 }

```

在 `virtqueue_get_buf_ctx` 中，我们可以看到，`virtio` 前端的 `vring` 中的 `last_used_idx` 加一，说明这块数据 `qemu` 后端已经消费完毕。我们可以通过 `detach_buf` 将其放入空闲队列中，留给以后的写入请求使用。

至此，整个存储虚拟化的写入流程才全部完成。

总结时刻

下面我们来总结一下存储虚拟化的场景下，整个写入的过程。

在虚拟机里面，应用层调用 write 系统调用写入文件。

write 系统调用进入虚拟机里面的内核，经过 VFS，通用块设备层，I/O 调度层，到达块设备驱动。

虚拟机里面的块设备驱动是 virtio_blk，它和通用的块设备驱动一样，有一个 request queue，另外有一个函数 make_request_fn 会被设置为 blk_mq_make_request，这个函数用于将请求放入队列。

虚拟机里面的块设备驱动是 virtio_blk 会注册一个中断处理函数 vp_interrupt。当 qemu 写入完成之后，它会通知虚拟机里面的块设备驱动。

blk_mq_make_request 最终调用 virtqueue_add，将请求添加到传输队列 virtqueue 中，然后调用 virtqueue_notify 通知 qemu。

在 qemu 中，本来虚拟机正处于 KVM_RUN 的状态，也即处于客户机状态。

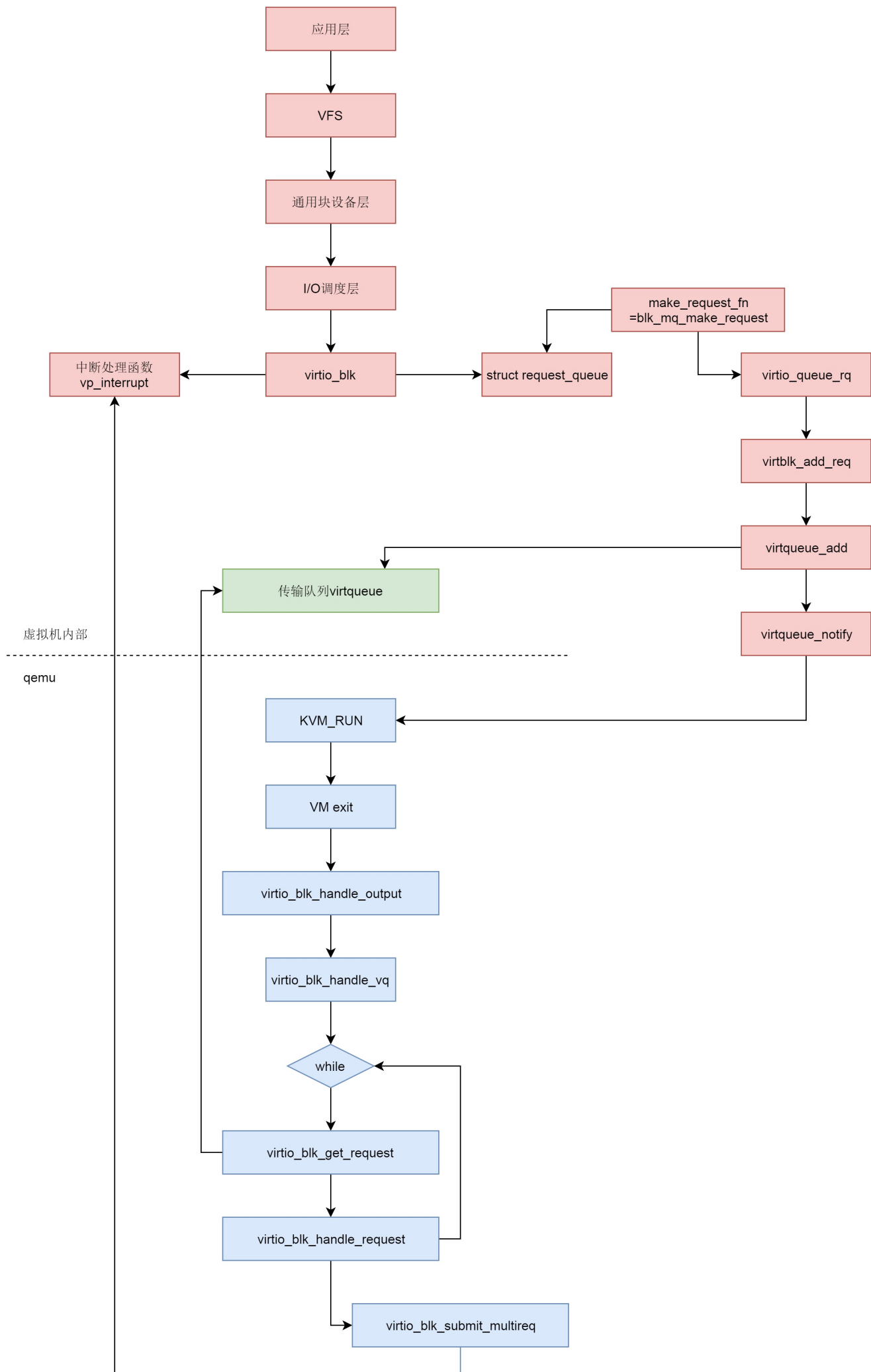
qemu 收到通知后，通过 VM exit 指令退出客户机状态，进入宿主机状态，根据退出原因，得知有 I/O 需要处理。

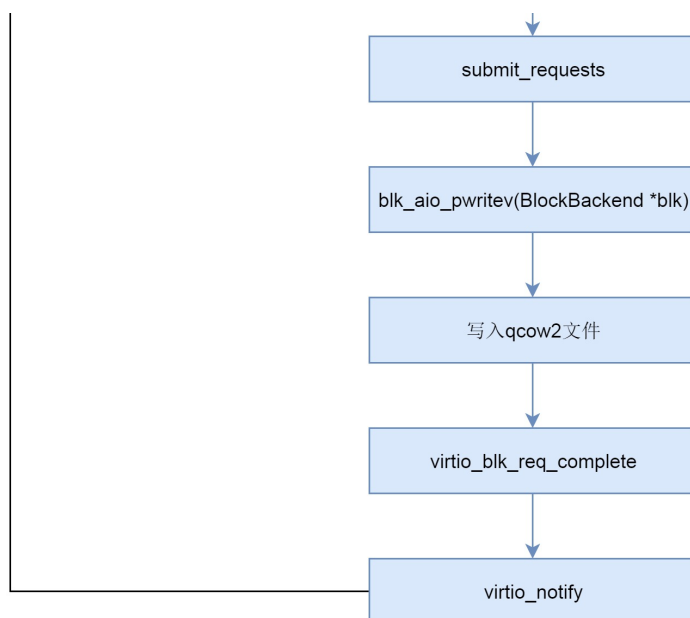
qemu 调用 virtio_blk_handle_output，最终调用 virtio_blk_handle_vq。

virtio_blk_handle_vq 里面有一个循环，在循环中，virtio_blk_get_request 函数从传输队列中拿出请求，然后调用 virtio_blk_handle_request 处理请求。

virtio_blk_handle_request 会调用 blk_aio_pwritev，通过 BlockBackend 驱动写入 qcow2 文件。

写入完毕之后，virtio_blk_req_complete 会调用 virtio_notify 通知虚拟机里面的驱动。数据写入完成，刚才注册的中断处理函数 vp_interrupt 会收到这个通知。





课堂练习

请你沿着代码，仔细分析并牢记 virtqueue 的结构以及写入和读取方式。这个结构在下面的网络传输过程中，还要起大作用。

欢迎留言和我分享你的疑惑和见解，也欢迎收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。



趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院
云计算技术部首席架构师



新版升级：点击「👤请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

上一篇 53 | 存储虚拟化（上）：如何建立自己保管的单独档案库？

下一篇 55 | 网络虚拟化：如何成立独立的合作部？

精选留言 (1)

写留言



没心没肺

2019-07-31

每次看到文中说还记得什么什么吗，我心里总是默默回答:不记得🤔

1

2