vhost-user 分析1

2018-01-24

占个坑，准备下写vhost-user的东西

vhost-user是vhost-kernel又回到用户空间的实现，其基本思想和vhost-kernel很类似，不过之前在内核的部分现在有另外一个用户进程代替，可能是snapp或者dpdk等。在网上看相关资料较少，就简单介绍下。虽然和vhost-kernel实现的目标一致，但是具体的实现方式却有所不同。vhost-user下，UNIX本地socket代替了之前kernel模式下的设备文件进行进程间的通信（qemu和vhost-user app），而通过mmap的方式把ram映射到vhost-user app的进程空间实现内存的共享。其他的部分和vhost-kernel原理基本一致。这种情况下一般qemu作为client，而vhost-user app作为server如DPDK。而本文对于vhost-user server端的分析主要也是基于DPDK源码。本文主要分析涉及到的三个重要机制：qemu和vhost-user app的消息传递，guest memory和vhost-user app的共享，guest和vhost-user app的通知机制。

一、qemu和vhost-user app的消息传递

qemu和vhost-user app的消息传递是通过UNIX本地socket实现的，对应于kernel下每个ioctl的实现，这里vhost-user app必须对每个ioctl 提供自己的处理，DPDK下在vhost-user.c文件下的vhost_user_msg_handler函数，这里有一个核心的数据结构：VhostUserMsg，该结构是消息传递的载体，整个结构并不复杂

```
typedef struct VhostUserMsg {
    union {
        VhostUserRequest master;//qemu
        VhostUserSlaveRequest slave;//dpdk
    } request;

#define VHOST_USER_VERSION_MASK     0x3
#define VHOST_USER_REPLY_MASK       (0x1 << 2)
#define VHOST_USER_NEED_REPLY        (0x1 << 3)
    uint32_t flags;
    uint32_t size; /* the following payload size */
    union {
#define VHOST_USER_VRING_IDX_MASK   0xff
#define VHOST_USER_VRING_NOFD_MASK  (0x1<<8)
        uint64_t u64;
        struct vhost_vring_state state;
        struct vhost_vring_addr addr;
        VhostUserMemory memory;
        VhostUserLog    log;
        struct vhost_iotlb_msg iotlb;
    } payload;
    int fds[VHOST_MEMORY_MAX_NREGIONS];
} __attribute((packed)) VhostUserMsg;
```

既然是传递消息，其中必须包含消息的种类、消息的内容、消息内容的大小。而这些也是该结构的主要部分，首个union便标志该消息的种类。接下来的Flags表明该消息本身的一些性质，如是否需要回复等。size就是payload的大小，接下来的union是具体的消息内容，最后的fds是关联每一个memory RAM的fd数组。消息种类如下：

```
typedef enum VhostUserRequest {
    VHOST_USER_NONE = 0,
    VHOST_USER_GET_FEATURES = 1,
    VHOST_USER_SET_FEATURES = 2,
    VHOST_USER_SET_OWNER = 3,
    VHOST_USER_RESET_OWNER = 4,
    VHOST_USER_SET_MEM_TABLE = 5,
```


邮箱：zhunxun@gmail.com

< 2020年5月 >

日	一	二	三	四	五	六
26	27	28	29	30	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
31	1	2	3	4	5	6

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(21154)
2. PCI 设备详解一(15808)
3. 进程的挂起、阻塞和睡眠(13714)
4. Linux下桥接模式详解一(13467)
5. virtio后端驱动详解(10539)

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)

```
    VHOST_USER_SET_LOG_BASE = 6,
    VHOST_USER_SET_LOG_FD = 7,
    VHOST_USER_SET_VRING_NUM = 8,
    VHOST_USER_SET_VRING_ADDR = 9,
    VHOST_USER_SET_VRING_BASE = 10,
    VHOST_USER_GET_VRING_BASE = 11,
    VHOST_USER_SET_VRING_KICK = 12,
    VHOST_USER_SET_VRING_CALL = 13,
    VHOST_USER_SET_VRING_ERR = 14,
    VHOST_USER_GET_PROTOCOL_FEATURES = 15,
    VHOST_USER_SET_PROTOCOL_FEATURES = 16,
    VHOST_USER_GET_QUEUE_NUM = 17,
    VHOST_USER_SET_VRING_ENABLE = 18,
    VHOST_USER_SEND_RARP = 19,
    VHOST_USER_NET_SET_MTU = 20,
    VHOST_USER_SET_SLAVE_REQ_FD = 21,
    VHOST_USER_IOTLB_MSG = 22,
    VHOST_USER_MAX
} VhostUserRequest;
```

到目前为止并不复杂，我们下面看下消息本身的初始化机制，socket-file的路径会作为参数传递进来，在main函数中examples/vhost/，调用us_vhost_parse_socket_path对参数中的socket-fiile参数进行解析，保存在静态数组socket_files中，而后在main函数中有一个for循环，针对每个socket-file，会调用rte_vhost_driver_register函数注册vhost 驱动，该函数的核心功能就是为每个socket-fie创建本地socket，通过create_unix_socket函数。vhost中的socket结构通过create_unix_socket描述。在注册驱动之后，会根据具体的特性设置features。在最后会通过rte_vhost_driver_start启动vhost driver，该函数倒是值得一看：

```
int
rte_vhost_driver_start(const char *path)
{
    struct vhost_user_socket *vsocket;
    static pthread_t fdset_tid;

    pthread_mutex_lock(&vhost_user.mutex);
    vsocket = find_vhost_user_socket(path);
    pthread_mutex_unlock(&vhost_user.mutex);

    if (!vsocket)
        return -1;
    /*创建一个线程监听fdset*/
    if (fdset_tid == 0) {
        int ret = pthread_create(&fdset_tid, NULL, fdset_event_dispatch,
                    &vhost_user.fdset);
        if (ret < 0)
            RTE_LOG(ERR, VHOST_CONFIG,
                "failed to create fdset handling thread");
    }

    if (vsocket->is_server)
        return vhost_user_start_server(vsocket);
    else
        return vhost_user_start_client(vsocket);
}
```

函数参数是对应的socket-file的路径，进入函数内部，首先便是根据路径通过find_vhost_user_socket函数找到对应的vhost_user_socket结构，所有的vhost_user_socket以一个数组的形式保存在vhost_user数据结构中。接下来如果该socket确实存在，就创建一个线程，处理vhost-user的fd,这个作用我们后面再看，该线程绑定的函数为fdset_event_dispatch。这些工作完成后，就启动该socket了，起始qemu和vhost可以互做server和client，一般情况下vhsot是作为server存在。所以这里就调用了vhost_user_start_server。这里就是我们常见的socket编程操作了，调用bind……然后listen……，没什么好说的。后面调用了fdset_add函数，这是就是vhost处理消息fd的一个单独的机制，

```
int
fdset_add(struct fdset *pfdset, int fd, fd_cb rcb, fd_cb wcb, void *dat)
{
    int i;

    if (pfdset == NULL || fd == -1)
        return -1;
```

```
    pthread_mutex_lock(&pfdset->fd_mutex);
    i = pfdset->num < MAX_FDS ? pfdset->num++ : -1;
    if (i == -1) {
        fdset_shrink_nolock(pfdset);
        i = pfdset->num < MAX_FDS ? pfdset->num++ : -1;
        if (i == -1) {
            pthread_mutex_unlock(&pfdset->fd_mutex);
            return -2;
        }
    }

    fdset_add_fd(pfdset, i, fd, rcb, wcb, dat);
    pthread_mutex_unlock(&pfdset->fd_mutex);

    return 0;
}
```

简单来说就是该函数为对应的fd注册了一个处理函数，当该fd有信号时，就调用该函数，这里就是vhost_user_server_new_connection。具体是如何实现的呢？看下fdset_add_fd函数

```
static void
fdset_add_fd(struct fdset *pfdset, int idx, int fd,
    fd_cb rcb, fd_cb wcb, void *dat)
{
    struct fdentry *pfdentry = &pfdset->fd[idx];
    struct pollfd *pfd = &pfdset->rwfds[idx];

    pfdentry->fd  = fd;
    pfdentry->rcb = rcb;
    pfdentry->wcb = wcb;
    pfdentry->dat = dat;

    pfd->fd = fd;
    pfd->events  = rcb ? POLLIN : 0;
    pfd->events |= wcb ? POLLOUT : 0;
    pfd->revents = 0;
}
```

这里分成了两部分，一个是fdentry，一个是pollfd。前者保存具体的信息，后者用作poll操作，方便线程监听fd。参数中函数指针为第三个参数，所以这里pfd->events就是POLLIN。那么在会到处理线程的处理函数fdset_event_dispatch中，该函数会监听vhost_user.fdset中的rwfds，当某个fd有信号时，则进入处理流程

```
if (rcb && pfd->revents & (POLLIN | FDPOLLERR))
                rcb(fd, dat, &remove1);
if (wcb && pfd->revents & (POLLOUT | FDPOLLERR))
                wcb(fd, dat, &remove2);
```

这里的rcb便是前面针对fd注册的回调函数。再次回到vhost_user_server_new_connection函数中，当某个fd有信号时，这里指对应socket-file的fd，则该函数被调用，建立连接，然后调用vhost_user_add_connection函数。既然连接已经建立，则需要对该连接进行vhost的一些设置了，包括创建virtio_net设备附加到连接上，设置device名字等等。而关键的一步是为该fd添加回调函数，刚才的回调函数用于建立连接，在连接建立后就需要设置函数处理socket的msg了，这里便是vhost_user_read_cb。到这里正式进入msg的部分。该函数中调用了vhost_user_msg_handler，而该函数正是处理socket msg的核心函数。到这里消息处理的部分便介绍完成了。

二、guest memory和vhost-user app的共享

虽然qemu和vhost通过socket建立了联系，但是这信息量毕竟有限，重点是要传递的数据，难不成通过socket传递的？？当然不是，如果这样模式切换和数据复制估计会把系统撑死……这里主要也是用到共享内存的概念。核心机制和vhost-kernel类似，qemu也需要把guest的内存布局通过MSG传递给vhost-user，那么我们就从这里开始分析，在函数vhost_user_msg_handler中

```
    case VHOST_USER_SET_MEM_TABLE:
        ret = vhost_user_set_mem_table(dev, &msg);
        break;
```

在分析函数之前我们先看下几个数据结构

```
/*对应qemu端的region结构*/
typedef struct VhostUserMemoryRegion {
    uint64_t guest_phys_addr;//GPA of region
    uint64_t memory_size;    //size
    uint64_t userspace_addr;//HVA in qemu process
    uint64_t mmap_offset; //offset
} VhostUserMemoryRegion;

typedef struct VhostUserMemory {
    uint32_t nregions;//region num
    uint32_t padding;
    VhostUserMemoryRegion regions[VHOST_MEMORY_MAX_NREGIONS];//All region
} VhostUserMemory;
```

在vhsot端，对应的数据结构为

```
struct rte_vhost_mem_region {
    uint64_t guest_phys_addr;//GPA of region
    uint64_t guest_user_addr;//HVA in qemu process
    uint64_t host_user_addr;//HVA in vhost-user
    uint64_t size;//size
    void     *mmap_addr;//mmap base Address
    uint64_t mmap_size;
    int fd;//relative fd of region
};
```

意义都比较容易理解就不在多说，在virtio_net结构中保存有指向当前连接对应的memory结构rte_vhost_memory

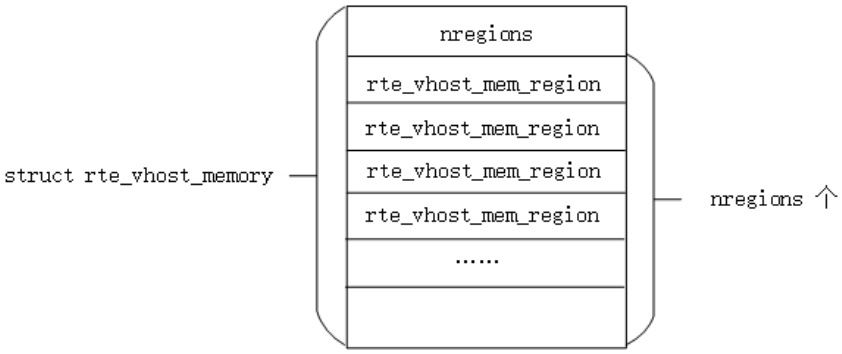
```
struct rte_vhost_memory {
    uint32_t nregions;
    struct rte_vhost_mem_region regions[];
};
```

OK，下面看代码，代码虽然较多，但是意义都比较容易理解，只看核心部分吧：

```
dev->mem = rte_zmalloc("vhost-mem-table", sizeof(struct rte_vhost_memory) +
        sizeof(struct rte_vhost_mem_region) * memory.nregions, 0);
    if (dev->mem == NULL) {
        RTE_LOG(ERR, VHOST_CONFIG,
            "(%d) failed to allocate memory for dev->mem\n",
            dev->vid);
        return -1;
    }
    /*region num*/
    dev->mem->nregions = memory.nregions;

    for (i = 0; i < memory.nregions; i++) {
        /*fd info*/
        fd  = pmsg->fds[i];
        reg = &dev->mem->regions[i];
        /*GPA of specific region*/
        reg->guest_phys_addr = memory.regions[i].guest_phys_addr;
        /*HVA in qemu address*/
        reg->guest_user_addr = memory.regions[i].userspace_addr;
        reg->size            = memory.regions[i].memory_size;
        reg->fd              = fd;
        /*offset in region*/
        mmap_offset = memory.regions[i].mmap_offset;
        mmap_size   = reg->size + mmap_offset;

        /* mmap() without flag of MAP_ANONYMOUS, should be called
         * with length argument aligned with hugepagesz at older
         * longterm version Linux, like 2.6.32 and 3.2.72, or
         * mmap() will fail with EINVAL.
         *
         * to avoid failure, make sure in caller to keep length
         * aligned.
         */
        alignment = get_blk_size(fd);
        if (alignment == (uint64_t)-1) {
            RTE_LOG(ERR, VHOST_CONFIG,
```

```
                "couldn't get hugepage size through fstat\n");
            goto err_mmap;
        }
        /*对齐*/
        mmap_size = RTE_ALIGN_CEIL(mmap_size, alignment);
        /*执行映射，这里就是本进程的虚拟地址了，为何能映射另一个进程的文件描述符呢?*/
        mmap_addr = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE,
                 MAP_SHARED | MAP_POPULATE, fd, 0);

        if (mmap_addr == MAP_FAILED) {
            RTE_LOG(ERR, VHOST_CONFIG,
                "mmap region %u failed.\n", i);
            goto err_mmap;
        }

        reg->mmap_addr = mmap_addr;
        reg->mmap_size = mmap_size;
        /*region Address in vhost process*/
        reg->host_user_addr = (uint64_t)(uintptr_t)mmap_addr +
                      mmap_offset;

        if (dev->dequeue_zero_copy)
            add_guest_pages(dev, reg, alignment);


    }
```

首先就是为dev分配mem空间，由此我们也可以得到该结构的布局



下面一个for循环对每个region先进行对应信息的复制，然后对该region的大小进行对其操作，接着通过mmap的方式对region关联的fd进行映射，这里便得到了region在vhost端的虚拟地址，但是region中GPA对应的虚拟地址还需要在mmap得到的虚拟地址上加上offset，该值也是作为参数传递进来的。到此，设置memory Table的工作基本完成，看下地址翻译过程呢?

```
/* Converts QEMU virtual address to Vhost virtual address. */
static uint64_t
qva_to_vva(struct virtio_net *dev, uint64_t qva)
{
    struct rte_vhost_mem_region *reg;
    uint32_t i;

    /* Find the region where the address lives. */
    for (i = 0; i < dev->mem->nregions; i++) {
        reg = &dev->mem->regions[i];

        if (qva >= reg->guest_user_addr &&
            qva <  reg->guest_user_addr + reg->size) {
            return qva - reg->guest_user_addr +
                   reg->host_user_addr;
        }
    }

    return 0;
}
```

相当简单把，核心思想是先使用QVA确定在哪一个region，然后取地址在region中的偏移，加上该regior在vhost-user映射的实际有效地址即reg->host_user_addr字段。这部分还有一个核心思想是fd的使用，vhost_user_set_mem_table直接从MSG中获取到了fd，然后直接把FD进行mmap映射，这点一时间让我难以理解，FD不是仅仅在进程内部有效么？怎么也可以共享了？？通过向开源社区请教，感叹自

己的知识面实在狭窄，这是Unix下一种通用的传递描述符的方式，怎么说呢？就是进程A的描述符可以通过特定的调用传递给进程B，进程B在自己的描述符表中分配一个位置给该描述符指针，因此实际上进程B使用的并不是A的FD，而是自己描述符表中的FD，但是两个进程的FD却指向同一个描述符表，就像是增加了一个引用而已。后面会专门对该机制进行详解，本文仅仅了解该作用即可。

三、vhost-user app的通知机制。

这里的通知机制和vhost kernel基本一致，都是通过eventfd的方式。因此这里就比较简单了

qemu端的代码：

```
file.fd = event_notifier_get_fd(virtio_queue_get_host_notifier(vvq));
r = dev->vhost_ops->vhost_set_vring_kick(dev, &file);
```

```
static int vhost_user_set_vring_kick(struct vhost_dev *dev,
                                     struct vhost_vring_file *file)
{
    return vhost_set_vring_file(dev, VHOST_USER_SET_VRING_KICK, file);
}
static int vhost_set_vring_file(struct vhost_dev *dev,
                                VhostUserRequest request,
                                struct vhost_vring_file *file)
{
    int fds[VHOST_MEMORY_MAX_NREGIONS];
    size_t fd_num = 0;
    VhostUserMsg msg = {
        .request = request,
        .flags = VHOST_USER_VERSION,
        .payload.u64 = file->index & VHOST_USER_VRING_IDX_MASK,
        .size = sizeof(msg.payload.u64),
    };

    if (ioeventfd_enabled() && file->fd > 0) {
        fds[fd_num++] = file->fd;
    } else {
        msg.payload.u64 |= VHOST_USER_VRING_NOFD_MASK;
    }

    if (vhost_user_write(dev, &msg, fds, fd_num) < 0) {
        return -1;
    }

    return 0;
}
```

可以看到这里实质上也是把eventfd的描述符传递给vhost-user。再看vhost-user端，在vhost_user_set_vring_kick中，关键的一句

```
vq->kickfd = file.fd;
```

其实这里的通知机制和kernel下没什么区别，不过是换到用户空间对eventfd进行操作而已，这里暂时不讨论了，后面有时间在补充！

以马内利！

参考资料：

qemu 2.7 源码

DPDK源码

分类：IO Virtualization, KVM虚拟化技术, qemu

jack.chen
关注 - 12
粉丝 - 44

+加关注

« 上一篇： 聊一聊Linux中的工作队列2
» 下一篇： QEMU IO事件处理框架

posted @ 2018-02-02 14:08 jack.chen Views(2739) Comments(8) Edit 收藏

Post Comment

#1楼 2018-03-03 12:25 | legolas_yang

感动，大神终于开始看vhost_user了，期待后续，我们做了这一块半年多，dpdk端的机制非常熟悉，就是qemu端i
于涉及很多内核的，不是很清楚。上面关于rte_vhost_memory_region，在具体实验中打印出来只有两三块，而在
端qemu代码中内存映射传的参数是address_space_memory，也就是把address_space_memory里所有的mer
ry_region_section给映射成共享内存了，但是为什么后端打印出来并不是整个虚拟机的物理内存，而是两三段分离
内存呢

支持(0) 反对(

#2楼 [楼主] 2018-03-06 10:31 | jack.chen

@ legolas_yang
按道理来讲不应该的呀，但是我确实没有试过，这两天有些忙，凑空我再看看，抱歉！！

支持(0) 反对(

#3楼 [楼主] 2018-03-07 19:41 | jack.chen

@ legolas_yang
你好，今天凑空调试了下，发现在qemu端也是这样的，虽然小的region不少，但是在提交之前会进行merge，最终
就是3-4个的样子，当然根据实际情况可能会有差异，但是不会很大！

支持(0) 反对(

#4楼 2018-03-08 12:17 | legolas_yang

对，是这样的，但是还有一个我比较纳闷的就是，mmap映射通过文件映射，然而在后端打印fd的st_blksize是0x4
00000，也就是说这一个文件的最小块大小是1GB，所以它每次映射最小单位是一个块，这是怎么做到的，可我系统
面的文件系统块大小也就4KB。

支持(0) 反对(

#5楼 2018-05-06 16:16 | avalanche

您好，我想请教一个问题，我看vhost-user代码时看到了两个变量last_used_idx和last_avail_idx，请问这是后端
理available ring和used ring时记录的临时变量吗？我的理解是后端读到available ring的index时，发现自己的las
avail_idx不等于index，然后就开始处理buffer并将last_avail_idx逐个加一，直到等于index。对于used ring来说
是逐个放入buffer并将last_used_idx加一，放完后将新的last_used_idx写入到used ring的index中，这样前端就
据上一次处理的index得知这次要处理的buffer个数，请问是这样运作的吗？

支持(0) 反对(

#6楼 [楼主] 2018-05-16 15:43 | jack.chen

@ avalanche
你好，今天凑空看了下，last_used_idx和last_avail_idx可以理解为临时变量，这主要是纪录对应ring上次使用到
哪里；而具体ring中的idx表示VRing中desctable所有的表项数目，这样如果从网卡接收数据，就需要根据实际数据
的数量和可用buffer的数量，取最小值来进行接收。而可用buffer的num就是idx-last_used_idx(针对接收队列来
讲)，发送队列类似。但是最后并没有将last_used_idx写入到used ring的index，而仅仅是针对后者做更新。前段
使用used ring的index的思路和后端使用available ring的index的思路类似，但是确实有点绕，需要好好思考！
PS：其实这里virtio、vhost-kernel、vhost-user的基本机制都是一致的，只是实现位置或者方法不同，可以参考之
前博文，欢迎交流！
http://www.cnblogs.com/ck1020/p/6044134.html
http://www.cnblogs.com/ck1020/p/5939777.html

支持(0) 反对(

#7楼 2018-05-16 16:07 | avalanche

@ jack.chen
好的 感谢您的解答！

支持(0) 反对(

#8楼 2019-04-24 13:55 | fengyd

@ jack.chen
对于vhost, last_avail_idx是在vhost端的vhost_virtqueue里维护的，

但是迁移过程是有QEMU控制的，QEMU会比较last_avail_idx和used_idx是不是合法，这里QEMU取的last_avail_idx又是保存在QEMU端 的VirtQueue里。
那vhost和QEMU里的last_avail_idx是怎么同步的？会同步吗？

迁移时，相关QEMU被调用到vritio_load的代码：

```
vdev->vq[i].inuse = (uint16_t)(vdev->vq[i].last_avail_idx -
vdev->vq[i].used_idx);
if (vdev->vq[i].inuse > vdev->vq[i].vring.num) {
error_report("VQ %d size 0x%x < last_avail_idx 0x%x - "
"used_idx 0x%x",
i, vdev->vq[i].vring.num,
vdev->vq[i].last_avail_idx,
vdev->vq[i].used_idx);
```


以马内利