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

qemu网络虚拟化之数据流向分析三

2016-09-27

前篇文章通过分析源代码，大致描述了各个数据结构之间的关系是如何建立的，那么今天就从数据包的角度，分析下数据包是如何在这些数据结构中间流转的！

这部分内容需要结合前面两篇文章来看，我们还是按照从Tap设备->Hub->NIC的流程。

首先看Tap设备，在Tap.c文件中：

先看下Tap设备注册的处理函数

```
1 static NetClientInfo net_tap_info = {
2     .type = NET_CLIENT_OPTIONS_KIND_TAP,
3     .size = sizeof(TAPState),
4     .receive = tap_receive,
5     .receive_raw = tap_receive_raw,
6     .receive_iov = tap_receive_iov,
7     .poll = tap_poll,
8     .cleanup = tap_cleanup,
9 };
```

其中重点就是tap_receive，该函数中会根据情况调用tap_receive_raw，而tap_receive_iov是在使用IO向量的情况下使用的，逻辑上和tap_receive是一个级别，关于IO向量下面首先会简要分析下：

IO向量：

IO向量的主要目的是让调用在一次原子操作中连续读写多个缓冲区，从而提高效率。IO向量结构如下：

```
1 struct iovec {
2     void *iov_base;
3     size_t iov_len;
4 };
```

结构很简单，前者iov_base指向一个缓冲区，iov_len记录缓冲区的长度。一般会有一个iovec数组来描述IO向量，数组的个数就是缓冲区的个数。

先看下tap_receive函数

```
1 static ssize_t tap_receive(NetClientState *nc, const uint8_t *buf, size_t size)
2 {
3     TAPState *s = DO_UPCAST(TAPState, nc, nc);
4     struct iovec iov[1];
5
6     if (s->host_vnet_hdr_len && !s->using_vnet_hdr) {
7         return tap_receive_raw(nc, buf, size);
8     }
9
10    iov[0].iov_base = (char *)buf;
11    iov[0].iov_len  = size;
12
13    return tap_write_packet(s, iov, 1);
14 }
```

该函数接收从用户应用程序传递过来的数据，然后写入到设备文件中。在没有使用virtIO 的情况下是直接交给tap_receive_raw处理，否则设置IO向量并调用tap_write_packet函数

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(15806)
3. 进程的挂起、阻塞和睡眠(13713)
4. Linux下桥接模式详解一(13465)
5. virtio后端驱动详解(10538)

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)

```
1 static ssize_t tap_write_packet(TAPState *s, const struct iovec *iov, int iovcnt)
2 {
3     ssize_t len;
4
5     do {
6         len = writev(s->fd, iov, iovcnt);
7     } while (len == -1 && errno == EINTR);
8
9     if (len == -1 && errno == EAGAIN) {
10        tap_write_poll(s, true);
11        return 0;
12    }
13
14    return len;
15 }
```

在该函数中调用了writev函数进向TAPState->fd 进行数据的写入。这里数据被组织成IO向量，写入完成需要调用tap_write_poll更新下fd的处理函数(**原因**)。

而tap_receive_raw函数和tap_receive_iov函数本质上和tap_receive实现类似的功能，只有一些细枝末节的变化，这里就不在分析。

下面还是转换方向，从tap部分的发送函数说起,这里看tap_send函数

```
1 static void tap_send(void *opaque)
2 {
3     TAPState *s = opaque;
4     int size;
5
6     do {
7         uint8_t *buf = s->buf;
8
9         size = tap_read_packet(s->fd, s->buf, sizeof(s->buf));
10        if (size <= 0) {
11            break;
12        }
13       //如果设置了vnet头部长度，但是using_vnet_hdr为0，就在移动buffer指针且修正size。因为这里buffer里面包含了头部，但是设备并没有使用
14        if (s->host_vnet_hdr_len && !s->using_vnet_hdr) {
15            buf  += s->host_vnet_hdr_len;
16            size -= s->host_vnet_hdr_len;
17        }
18
19        size = qemu_send_packet_async(&s->nc, buf, size, tap_send_completed);
20        if (size == 0) {
21            tap_read_poll(s, false);
22        }
23    } while (size > 0 && qemu_can_send_packet(&s->nc));
24 }
```

这里通过一个循环反复调用tap_read_packet函数从打开的设备文件中读取数据，并每一次读取完毕调用qemu_send_packet_async函数进行数据的发送。前者比较简单，就是普通的读文件操作，后者我们来看下

```
1 size_t qemu_send_packet_async(NetClientState *sender,
2                               const uint8_t *buf, int size,
3                               NetPacketSent *sent_cb)
4 {
5     return qemu_send_packet_async_with_flags(sender, QEMU_NET_PACKET_FLAG_NONE,
6                                              buf, size, sent_cb);
7 }
```

可以看到这里调用了qemu_send_packet_async_with_flags函数，那么继续深入

```
static ssize_t qemu_send_packet_async_with_flags(NetClientState *sender,
                                                 unsigned flags,
                                                 const uint8_t *buf, int size,
                                                 NetPacketSent *sent_cb)
{
    NetQueue *queue;

#ifdef DEBUG_NET
    printf("qemu_send_packet_async:\n");
    hex_dump(stdout, buf, size);
#endif
//如果发送端口的peer指针为空会发送失败，即不存在目标网卡
    if (sender->link_down || !sender->peer) {
        return size;
    }
//获取对方的接收缓冲队列
    queue = sender->peer->incoming_queue;

    return qemu_net_queue_send(queue, sender, flags, buf, size, sent_cb);
```

这里就做了一下实质性的判断，查看下net client的连接是否打开并且对方net client是否存在，不满足条件直接返回，通过的话就获取对方的接收队列sender->peer->incoming_queue;注意这里是对方的接收队列，下面可以看到实际上只是把数据从buffer中复制到队列维护的链表中了。

然后调用qemu_net_queue_send函数进行发送

```
ssize_t qemu_net_queue_send(NetQueue *queue,
                            NetClientState *sender,
                            unsigned flags,
                            const uint8_t *data,
                            size_t size,
                            NetPacketSent *sent_cb)
{
    ssize_t ret;
//这里表示如果queue正在发送就直接把buffer附加到队列的package链表，当然是在条件允许的情况下
    if (queue->delivering || !qemu_can_send_packet(sender)) {
        qemu_net_queue_append(queue, sender, flags, data, size, sent_cb);
        return 0;
    }
//否则需要启动队列进行数据的发送
    ret = qemu_net_queue_deliver(queue, sender, flags, data, size);
    if (ret == 0) {
        qemu_net_queue_append(queue, sender, flags, data, size, sent_cb);
        return 0;
    }
//否则只能刷新下queue
    qemu_net_queue_flush(queue);

    return ret;
}
```

该函数就要做具体的工作了，首先判断队列是否正在进行发送，是的话直接调用qemu_net_queue_append函数把buffer附加到queue的发送链表中，否则还需要重新启动队列发送，然后在附加到发送链表。

假如都不成功就只能调用qemu_net_queue_flush函数重置下队列。

```
static void qemu_net_queue_append(NetQueue *queue,
                                  NetClientState *sender,
                                  unsigned flags,
                                  const uint8_t *buf,
                                  size_t size,
                                  NetPacketSent *sent_cb)
{
    NetPacket *packet;

```

```
    if (queue->nq_count >= queue->nq_maxlen && !sent_cb) {
        return; /* drop if queue full and no callback */
    }
    packet = g_malloc(sizeof(NetPacket) + size);
    packet->sender = sender;
    packet->flags = flags;
    packet->size = size;
    packet->sent_cb = sent_cb;
    memcpy(packet->data, buf, size);

    queue->nq_count++;
    QTAILQ_INSERT_TAIL(&queue->packets, packet, entry);
}
```

可以看到这里做的工作很简单，就是分配一个package把数据复制到里面，然后插入发送链表。

Hub 端

前面结合源代码大致分析了下Tap端数据的发送接收流程，本节介绍下Hub接收并转发数据包的流程，代码大部分都在hub.c中

相比前面的Tap。这里Hub完成的工作就要简单的多，代码量也要少很多，因为它其实并不分哪一端，只负责转发数据包，看下net_hub_receive函数

```
static ssize_t net_hub_receive(NetHub *hub, NetHubPort *source_port,
                               const uint8_t *buf, size_t len)
{
    NetHubPort *port;
//收到数据包就从其他端口转发
    QLIST_FOREACH(port, &hub->ports, next) {
        if (port == source_port) {
            continue;
        }

        qemu_send_packet(&port->nc, buf, len);
    }
    return len;
}
```

这里可以看到遍历Hub上的所有端口，然后调用qemu_send_packet函数对单个端口进行发送数据，其中忽略source_port。还有一个函数和这个函数相对就是net_hub_receive_iov，该函数以IO向量的方式对数据包做处理

```
static ssize_t net_hub_receive_iov(NetHub *hub, NetHubPort *source_port,
                                   const struct iovec *iov, int iovcnt)
{
    NetHubPort *port;
    ssize_t len = iov_size(iov, iovcnt);

    QLIST_FOREACH(port, &hub->ports, next) {
        if (port == source_port) {
            continue;
        }

        qemu_sendv_packet(&port->nc, iov, iovcnt);
    }
    return len;
}
```

前面依然是遍历端口，不同的最后调用qemu_sendv_packet函数

```
ssize_t
qemu_sendv_packet(NetClientState *nc, const struct iovec *iov, int iovcnt)
{
    return qemu_sendv_packet_async(nc, iov, iovcnt, NULL);
}
```

```
ssize_t qemu_sendv_packet_async(NetClientState *sender,
                                const struct iovec *iov, int iovcnt,
                                NetPacketSent *sent_cb)
{
    NetQueue *queue;

    if (sender->link_down || !sender->peer) {
        return iov_size(iov, iovcnt);
    }

    queue = sender->peer->incoming_queue;

    return qemu_net_queue_send_iov(queue, sender,
                                   QEMU_NET_PACKET_FLAG_NONE,
                                   iov, iovcnt, sent_cb);
}
```

该函数是核心函数，这里的内容和前面Tap发送函数有些类似，sender是Hub上的转发端口的NetClientState结构，这里发送的 方式也是向对方的incoming_queue copy数据，唯一的区别在于这里采用的IO向量的方式，前面IO向量我们忽略了，这里就分析下，直接看向队列的链表中添加package的函数qemu_net_queue_append_iov

```
static void qemu_net_queue_append_iov(NetQueue *queue,
                                      NetClientState *sender,
                                      unsigned flags,
                                      const struct iovec *iov,
                                      int iovcnt,
                                      NetPacketSent *sent_cb)
{
    NetPacket *packet;
    size_t max_len = 0;
    int i;

    if (queue->nq_count >= queue->nq_maxlen && !sent_cb) {
        return; /* drop if queue full and no callback */
    }
    for (i = 0; i < iovcnt; i++) {
        max_len += iov[i].iov_len;
    }

    packet = g_malloc(sizeof(NetPacket) + max_len);
    packet->sender = sender;
    packet->sent_cb = sent_cb;
    packet->flags = flags;
    packet->size = 0;

    for (i = 0; i < iovcnt; i++) {
        size_t len = iov[i].iov_len;

        memcpy(packet->data + packet->size, iov[i].iov_base, len);
        packet->size += len;
    }

    queue->nq_count++;
    QTAILQ_INSERT_TAIL(&queue->packets, packet, entry);
}
```

这里首先判断queue的发送链表是否已满，然后获取数据的长度，需要结合所有IO向量包含的数据长度和，最后申请一段内存做package，需要包含所有的数据以及NetPacket结构，并对package做一些参数的设置。然后逐项从向量代表的buffer中复制数据。最后在一次性的把整个package插入链表。

NIC端

终于到了关键的时刻，这里其实函数不多，但是函数体很庞大，我们看e1000网卡的接收数据流程，先看e1000_receive函数

```
 1 static ssize_t
 2 e1000_receive(NetClientState *nc, const uint8_t *buf, size_t size)
 3 {
 4     const struct iovec iov = {
 5         .iov_base = (uint8_t *)buf,
 6         .iov_len = size
 7     };
 8
 9     return e1000_receive_iov(nc, &iov, 1);
10 }
```

这里不管有没有使用IO向量都把数据封装到了一个向量里面，然后调用e1000_receive_iov函数，该函数的函数体比较庞大，按模块分析的话也并不难。

```
  2 static size_t e1000_receive_iov(NetClientState *nc, const struct iovec *iov, int
iovcnt)
  3 {
  4     E1000State *s = qemu_get_nic_opaque(nc);
  5     PCIDevice *d = PCI_DEVICE(s);
  6     struct e1000_rx_desc desc;
  7     dma_addr_t base;
  8     unsigned int n, rdt;
  9     uint32_t rdh_start;
 10     uint16_t vlan_special = 0;
 11     uint8_t vlan_status = 0;
 12     uint8_t min_buf[MIN_BUF_SIZE];
 13     struct iovec min_iov;
 14     uint8_t *filter_buf = iov->iov_base;
 15     size_t size = iov_size(iov, iovcnt);
 16     size_t iov_ofs = 0;
 17     size_t desc_offset;
 18     size_t desc_size;
 19     size_t total_size;
 20
 21     if (!(s->mac_reg[STATUS] & E1000_STATUS_LU)) {
 22         return -1;
 23     }
 24
 25     if (!(s->mac_reg[RCTL] & E1000_RCTL_EN)) {
 26         return -1;
 27     }
 28
 29     /* Pad to minimum Ethernet frame length */

 30     if (size < sizeof(min_buf)) {
 31         iov_to_buf(iov, iovcnt, 0, min_buf, size);
 32         memset(&min_buf[size], 0, sizeof(min_buf) - size);
 33         min_iov.iov_base = filter_buf = min_buf;
 34         min_iov.iov_len = size = sizeof(min_buf);
 35         iovcnt = 1;
 36         iov = &min_iov;
 37     } else if (iov->iov_len < MAXIMUM_ETHERNET_HDR_LEN) {
 38         /* This is very unlikely, but may happen. */
 39         iov_to_buf(iov, iovcnt, 0, min_buf, MAXIMUM_ETHERNET_HDR_LEN);
 40         filter_buf = min_buf;
 41     }
 42
```

```
    /* Discard oversized packets if !LPE and !SBP. */
    if ((size > MAXIMUM_ETHERNET_LPE_SIZE ||
        (size > MAXIMUM_ETHERNET_VLAN_SIZE
        && !(s->mac_reg[RCTL] & E1000_RCTL_LPE)))
        && !(s->mac_reg[RCTL] & E1000_RCTL_SBP)) {
        return size;
    }
//先对数据包进行过滤
    if (!receive_filter(s, filter_buf, size)) {
        return size;
    }
//如果网卡支持vlan并且数据包是vlan数据包
    if (vlan_enabled(s) && is_vlan_packet(s, filter_buf)) {
        vlan_special = cpu_to_le16(be16_to_cpup((uint16_t *)(filter_buf
                                                             + 14)));
        iov_ofs = 4;
        if (filter_buf == iov->iov_base) {
            memmove(filter_buf + 4, filter_buf, 12);//destination, src, count
            } else {
            iov_from_buf(iov, iovcnt, 4, filter_buf, 12);
            while (iov->iov_len <= iov_ofs) {
                iov_ofs -= iov->iov_len;
                iov++;
            }
        }
        vlan_status = E1000_RXD_STAT_VP;
        size -= 4;
    }

    rdh_start = s->mac_reg[RDH];
    desc_offset = 0;
    total_size = size + fcs_len(s);//加上crc校验
    if (!e1000_has_rxbufs(s, total_size)) {
            set_ics(s, 0, E1000_ICS_RXO);
            return -1;
    }
    do {
        desc_size = total_size - desc_offset;
        if (desc_size > s->rxbuf_size) {
            desc_size = s->rxbuf_size;
        }
        base = rx_desc_base(s) + sizeof(desc) * s->mac_reg[RDH];
        pci_dma_read(d, base, &desc, sizeof(desc));
        desc.special = vlan_special;
        desc.status |= (vlan_status | E1000_RXD_STAT_DD);
        if (desc.buffer_addr) {
            if (desc_offset < size) {
                size_t iov_copy;
                hwaddr ba = le64_to_cpu(desc.buffer_addr);
                size_t copy_size = size - desc_offset;
                if (copy_size > s->rxbuf_size) {
                    copy_size = s->rxbuf_size;
                }
                do {
                    iov_copy = MIN(copy_size, iov->iov_len - iov_ofs);
                    pci_dma_write(d, ba, iov->iov_base + iov_ofs, iov_copy);
                    copy_size -= iov_copy;
                    ba += iov_copy;
                    iov_ofs += iov_copy;
                    if (iov_ofs == iov->iov_len) {
                        iov++;
                        iov_ofs = 0;
                    }
                } while (copy_size);
            }
            desc_offset += desc_size;
            desc.length = cpu_to_le16(desc_size);
            if (desc_offset >= total_size) {
                desc.status |= E1000_RXD_STAT_EOP | E1000_RXD_STAT_IXSM;
            } else {
                /* Guest zeroing out status is not a hardware requirement.
                   Clear EOP in case guest didn't do it. */
                desc.status &= ~E1000_RXD_STAT_EOP;
            }
        } else { // as per intel docs; skip descriptors with null buf addr
            DBGOUT(RX, "Null RX descriptor!!\n");
        }
        pci_dma_write(d, base, &desc, sizeof(desc));

```

```
122         if (++s->mac_reg[RDH] * sizeof(desc) >= s->mac_reg[RDLEN])
123             s->mac_reg[RDH] = 0;
124         /* see comment in start_xmit; same here */
125         if (s->mac_reg[RDH] == rdh_start) {
126             DBGOUT(RXERR, "RDH wraparound @%x, RDT %x, RDLEN %x\n",
127                    rdh_start, s->mac_reg[RDT], s->mac_reg[RDLEN]);
128             set_ics(s, 0, E1000_ICS_RXO);
129             return -1;
130         }
131     } while (desc_offset < total_size);
132 
133     s->mac_reg[GPRC]++;
134     s->mac_reg[TPR]++;
135     /* TOR - Total Octets Received:
136      * This register includes bytes received in a packet from the <Destination
137      * Address> field through the <CRC> field, inclusively.
138      */
139     n = s->mac_reg[TORL] + size + /* Always include FCS length. */ 4;
140     if (n < s->mac_reg[TORL])
141         s->mac_reg[TORH]++;
142     s->mac_reg[TORL] = n;
143 
144     n = E1000_ICS_RXT0;
145     if ((rdt = s->mac_reg[RDT]) < s->mac_reg[RDH])
146         rdt += s->mac_reg[RDLEN] / sizeof(desc);
147     if (((rdt - s->mac_reg[RDH]) * sizeof(desc)) <= s->mac_reg[RDLEN] >>
148         s->rxbuf_min_shift)
149         n |= E1000_ICS_RXDMT0;
150 
151     set_ics(s, 0, n);
152 
153     return size;
154 }
```

结合上面的代码，首先进行的是判断数据的长度是否满足一个最小以太网帧的长度，如果不满足就必须按照以太网帧的最小长度对齐，即后面填充0即可。

然后丢弃超过最大标准的数据包；

接着就调用receive_filter函数对数据包进行过滤，这是数据链路层的过滤，需要判断数据包的类型（广播、组播或者网卡是混杂模式都直接接收），如果是单播需要分析链路层头部，比对MAC地址。

然后下面的do循环中就开始数据的写入，这是直接采用DMA的方式吧数据直接写入到客户机内存，然后向客户机注入软中断通知客户机。

写入的方式比较复杂，但是主要是逻辑混乱，也不难理解，这里就不重点描述。

最后写入完成调用set_ics注入软中断。剩下的就是客户机的操作了。

而E1000的发送函数就是**start_xmit**函数，位于E1000.c中。

```
static void
start_xmit(E1000State *s)
{
    PCIDevice *d = PCI_DEVICE(s);
    dma_addr_t base;
    struct e1000_tx_desc desc;
    uint32_t tdh_start = s->mac_reg[TDH], cause = E1000_ICS_TXQE;

    if (!(s->mac_reg[TCTL] & E1000_TCTL_EN)) {
        DBGOUT(TX, "tx disabled\n");
        return;
    }

    while (s->mac_reg[TDH] != s->mac_reg[TDT]) {
        base = tx_desc_base(s) +
               sizeof(struct e1000_tx_desc) * s->mac_reg[TDH];
        pci_dma_read(d, base, &desc, sizeof(desc));

        DBGOUT(TX, "index %d: %p : %x %x\n", s->mac_reg[TDH],
               (void *)(intptr_t)desc.buffer_addr, desc.lower.data,
               desc.upper.data);

        process_tx_desc(s, &desc);
        cause |= txdesc_writeback(s, base, &desc);
```

```
        if (++s->mac_reg[TDH] * sizeof(desc) >= s->mac_reg[TDLEN])
            s->mac_reg[TDH] = 0;
        /*
         * the following could happen only if guest sw assigns
         * bogus values to TDT/TDLEN.
         * there's nothing too intelligent we could do about this.
         */
        if (s->mac_reg[TDH] == tdh_start) {
            DBGOUT(TXERR, "TDH wraparound @%x, TDT %x, TDLEN %x\n",
                   tdh_start, s->mac_reg[TDT], s->mac_reg[TDLEN]);
            break;
        }
    }
    set_ics(s, 0, cause);
}
```

具体的步骤和接收数据的模式类似，网卡的发送寄存器会包含数据包的head和tail，如果两者不一致就说明有新数据包。然后获取发送缓冲区的地址，注意这里需要先获取对应本次传输数据的e1000_tx_desc结构，这也是首次调用**pci_dma_read**函数的作用,该结构中记录了数据buffer的实际地址，这个地址是需要再次通过ＤＭＡ读取。获取到desc描述符后，就调用**process_tx_desc(s, &desc)**函数进行具体的传输数据DMA操作。

```
static void
process_tx_desc(E1000State *s, struct e1000_tx_desc *dp)
{
    PCIDevice *d = PCI_DEVICE(s);
    uint32_t txd_lower = le32_to_cpu(dp->lower.data);
    uint32_t dtype = txd_lower & (E1000_TXD_CMD_DEXT | E1000_TXD_DTYP_D);
    unsigned int split_size = txd_lower & 0xffff, bytes, sz, op;
    unsigned int msh = 0xfffff;
    uint64_t addr;
    struct e1000_context_desc *xp = (struct e1000_context_desc *)dp;
    struct e1000_tx *tp = &s->tx;

    s->mit_ide |= (txd_lower & E1000_TXD_CMD_IDE);
    if (dtype == E1000_TXD_CMD_DEXT) {    // context descriptor
        op = le32_to_cpu(xp->cmd_and_length);
        tp->ipcss = xp->lower_setup.ip_fields.ipcss;
        tp->ipcso = xp->lower_setup.ip_fields.ipcso;
        tp->ipcse = le16_to_cpu(xp->lower_setup.ip_fields.ipcse);
        tp->tucss = xp->upper_setup.tcp_fields.tucss;
        tp->tucso = xp->upper_setup.tcp_fields.tucso;
        tp->tucse = le16_to_cpu(xp->upper_setup.tcp_fields.tucse);
        tp->paylen = op & 0xfffff;
        tp->hdr_len = xp->tcp_seg_setup.fields.hdr_len;
        tp->mss = le16_to_cpu(xp->tcp_seg_setup.fields.mss);
        tp->ip = (op & E1000_TXD_CMD_IP) ? 1 : 0;
        tp->tcp = (op & E1000_TXD_CMD_TCP) ? 1 : 0;
        tp->tse = (op & E1000_TXD_CMD_TSE) ? 1 : 0;
        tp->tso_frames = 0;
        if (tp->tucso == 0) {    // this is probably wrong
            DBGOUT(TXSUM, "TCP/UDP: cso 0!\n");
            tp->tucso = tp->tucss + (tp->tcp ? 16 : 6);
        }
        return;
    } else if (dtype == (E1000_TXD_CMD_DEXT | E1000_TXD_DTYP_D)) {
        // data descriptor
        if (tp->size == 0) {
            tp->sum_needed = le32_to_cpu(dp->upper.data) >> 8;
        }
        tp->cptse = ( txd_lower & E1000_TXD_CMD_TSE ) ? 1 : 0;
    } else {
        // legacy descriptor
        tp->cptse = 0;
    }

    if (vlan_enabled(s) && is_vlan_txd(txd_lower) &&
        (tp->cptse || txd_lower & E1000_TXD_CMD_EOP)) {
        tp->vlan_needed = 1;
        stw_be_p(tp->vlan_header,
                      le16_to_cpup((uint16_t *)(s->mac_reg + VET)));
        stw_be_p(tp->vlan_header + 2,
                      le16_to_cpu(dp->upper.fields.special));
    }
    /*这里就是获取客户机中数据buffer的地址*/
```

```
    addr = le64_to_cpu(dp->buffer_addr);
    if (tp->tse && tp->cptse) {
        msh = tp->hdr_len + tp->mss;
        do {
            bytes = split_size;
            if (tp->size + bytes > msh)
                bytes = msh - tp->size;

            bytes = MIN(sizeof(tp->data) - tp->size, bytes);
            pci_dma_read(d, addr, tp->data + tp->size, bytes);
            sz = tp->size + bytes;
            if (sz >= tp->hdr_len && tp->size < tp->hdr_len) {
                memmove(tp->header, tp->data, tp->hdr_len);
            }
            tp->size = sz;
            addr += bytes;
            if (sz == msh) {
                xmit_seg(s);
                memmove(tp->data, tp->header, tp->hdr_len);
                tp->size = tp->hdr_len;
            }
        } while (split_size -= bytes);
    } else if (!tp->tse && tp->cptse) {
        // context descriptor TSE is not set, while data descriptor TSE is set
        DBGOUT(TXERR, "TCP segmentation error\n");
    } else {
        split_size = MIN(sizeof(tp->data) - tp->size, split_size);
        pci_dma_read(d, addr, tp->data + tp->size, split_size);
        tp->size += split_size;
    }

    if (!(txd_lower & E1000_TXD_CMD_EOP))
        return;
    if (!(tp->tse && tp->cptse && tp->size < tp->hdr_len)) {
        xmit_seg(s);
    }
    tp->tso_frames = 0;
    tp->sum_needed = 0;
    tp->vlan_needed = 0;
    tp->size = 0;
    tp->cptse = 0;
```

函数中需要首先判断描述符是什么类型。比较关键的是**E1000_TXD_CMD_DEXT**和**E1000_TXD_DTYP_D**，

然后调用**pci_dma_read**函数把内存（客户机内存）中的数据读到设备缓冲区中，这点和实际的DMA道理是一样的。

然后调用**xmit_seg**

```
static void
xmit_seg(E1000State *s)
{
    uint16_t len, *sp;
    unsigned int frames = s->tx.tso_frames, css, sofar, n;
    struct e1000_tx *tp = &s->tx;

    if (tp->tse && tp->cptse) {
        css = tp->ipcss;
        DBGOUT(TXSUM, "frames %d size %d ipcss %d\n",
               frames, tp->size, css);
        if (tp->ip) {        // IPv4
            stw_be_p(tp->data+css+2, tp->size - css);
            stw_be_p(tp->data+css+4,
                     be16_to_cpup((uint16_t *)(tp->data+css+4))+frames);
        } else               // IPv6
            stw_be_p(tp->data+css+4, tp->size - css);
        css = tp->tucss;
        len = tp->size - css;
        DBGOUT(TXSUM, "tcp %d tucss %d len %d\n", tp->tcp, css, len);
        if (tp->tcp) {
            sofar = frames * tp->mss;
            stl_be_p(tp->data+css+4, ldl_be_p(tp->data+css+4)+sofar); /* seq */
            if (tp->paylen - sofar > tp->mss)
                tp->data[css + 13] &= ~9;        // PSH, FIN
        } else    // UDP
```

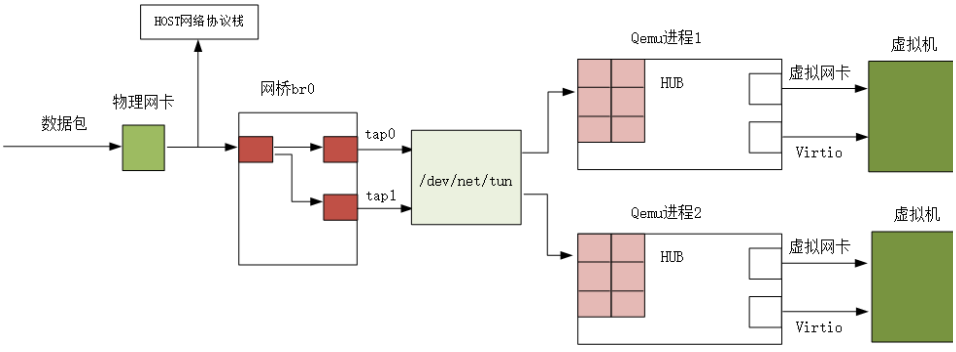
```
            stw_be_p(tp->data+css+4, len);
        if (tp->sum_needed & E1000_TXD_POPTS_TXSM) {
            unsigned int phsum;
            // add pseudo-header length before checksum calculation
            sp = (uint16_t *)(tp->data + tp->tucso);
            phsum = be16_to_cpup(sp) + len;
            phsum = (phsum >> 16) + (phsum & 0xffff);
            stw_be_p(sp, phsum);
        }
        tp->tso_frames++;
    }

    if (tp->sum_needed & E1000_TXD_POPTS_TXSM)
        putsum(tp->data, tp->size, tp->tucso, tp->tucss, tp->tucse);
    if (tp->sum_needed & E1000_TXD_POPTS_IXSM)
        putsum(tp->data, tp->size, tp->ipcso, tp->ipcss, tp->ipcse);
    if (tp->vlan_needed) {
        memmove(tp->vlan, tp->data, 4);
        memmove(tp->data, tp->data + 4, 8);
        memcpy(tp->data + 8, tp->vlan_header, 4);
        e1000_send_packet(s, tp->vlan, tp->size + 4);
    } else
        e1000_send_packet(s, tp->data, tp->size);
    s->mac_reg[TPT]++;
    s->mac_reg[GPTC]++;
    n = s->mac_reg[TOTL];
    if ((s->mac_reg[TOTL] += s->tx.size) < n)
        s->mac_reg[TOTH]++;
}
```

然后调用**e1000_send_packet**

```
static void
e1000_send_packet(E1000State *s, const uint8_t *buf, int size)
{
    NetClientState *nc = qemu_get_queue(s->nic);
    if (s->phy_reg[PHY_CTRL] & MII_CR_LOOPBACK) {
        nc->info->receive(nc, buf, size);
    } else {
        qemu_send_packet(nc, buf, size);
    }
}
```

最后就是**qemu_send_packet**，这就是之前我们分析过的函数了！！

最后放一张逻辑图示，旨在说明 数据包的流向，注意是逻辑图：



参考：qemu源码

linux内核源码

分类：linux 内核源码分析, KVM虚拟化技术, qemu

好文要顶 关注我 收藏该文


jack.chen
关注 - 12
粉丝 - 44
+加关注

0 0

« 上一篇： 软中断和tasklet介绍
» 下一篇： linux下的KSM内存共享机制分析

posted @ 2017-05-09 12:45 jack.chen Views(1094) Comments(0) Edit 收藏

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。

【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
【推荐】阿里毕玄16篇文章，深度讲解Java开发、系统设计、职业发展
【推荐】《Flutter in action》开放下载！闲鱼Flutter企业级实践精选

相关博文：
· 理解 Linux 网络栈（3）：QEMU/KVM + VxLAN 环境下的 Segmentation Offloading 技术（发送端）
· QEMU网络模式(一)——bridge
· linux网络流程分析（一）---网卡驱动
· virtIO前后端notify机制详解
· virtIO之VHOST工作原理简析
» 更多推荐...

Java经典面试题整理及答案详解（二）

最新 IT 新闻:
· 腾讯在列！微软宣布超140家工作室为Xbox Series X开发游戏
· 黑客声称从微软GitHub私人数据库当中盗取500GB数据
· IBM开源用于简化AI模型开发的Elyra工具包
· 中国网民人均安装63个App：腾讯系一家独大
· Lyft颁布新规：强制要求乘客和司机佩戴口罩
» 更多新闻...

Copyright © 2020 jack.chen
Powered by .NET Core on Kubernetes

以马内利