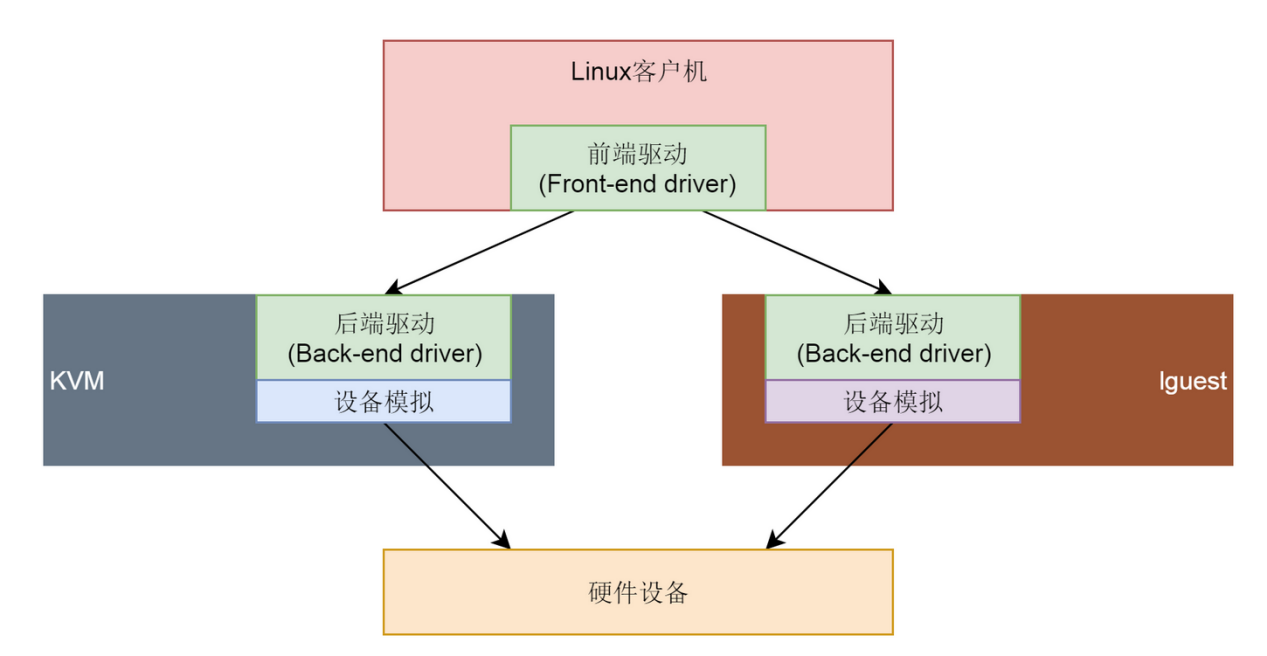
<https://blog.csdn.net/kunkliu/article/details/129782266>

<https://www.51cto.com/article/660499.html>

# 半虚拟化概述

## virtio运行结构



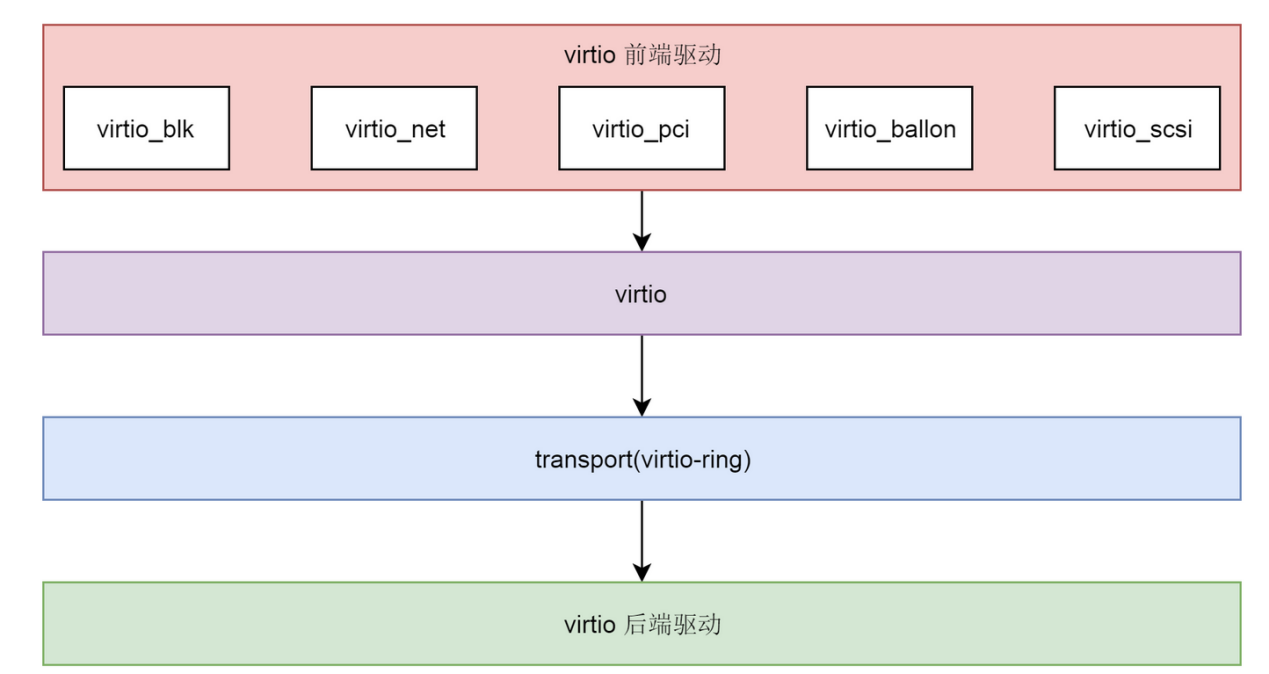
① virtio表示[虚拟化](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9F%E5%8C%96&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/kunkliu/article/details/_blank)IO，用于实现设备半虚拟化，即虚拟机中运行的操作系统需要加载特殊的驱动（e.g. virtio-net）且虚拟机知道自己是虚拟机

相较于基于完全模拟的全虚拟化，基于virtio的半虚拟化可以提升设备访问性能

② 运行在虚拟机中的部分称为前端驱动，负责对虚拟机提供统一的接口

③ 运行在宿主机中的部分称为后端驱动，负责适配不同的物理硬件设备

## virtio架构层次



### virtio前端驱动

① 运行在虚拟机中

② 针对不同类型的设备有不同的驱动程序，但是与后端驱动交互的接口都是统一的

③ 本文分析virtio-net模块，源码位于drivers/net/virtio\_net.c

### virtio层

① virtio层实现虚拟队列接口，作为前后端通信的桥梁

② 不同类型的设备使用的虚拟队列数量不同，比如virtio-net使用两个队列，一个用于接收，另一个用于发送

③ 源码位于drivers/virtio/virtio.c

### virtio-ring层

① virtio-ring层是虚拟队列的具体实现

② 源码位于driver/virtio/virtio\_ring.c

### virtio后端驱动

① 运行在宿主机中

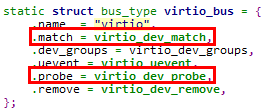
② 实现virtio后端的逻辑，主要是操作硬件设备，比如向内核协议栈发送一个网络包完成虚拟机对于网络的操作

③ 在Qemu + KVM虚拟化环境中，源码位于Qemu源码中（尚未详细分析）。后续将分析seL4中的后端实现

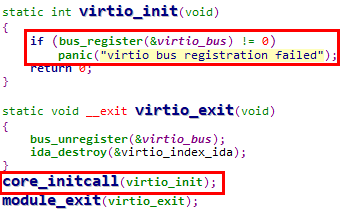
# Linux virtio核心数据结构

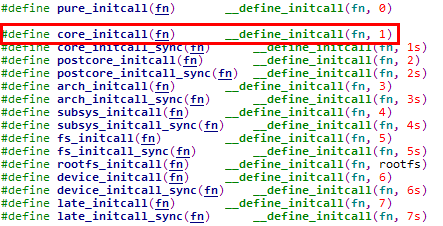
## virtio\_bus结构

struct bus\_type是基于总线驱动模型的公共数据结构，定义新的bus，就是填充该结构。virtio\_bus定义在drivers/virtio/virtio.c中，具体如下，



### 说明1：注册virtio\_bus

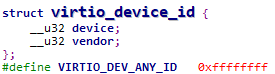




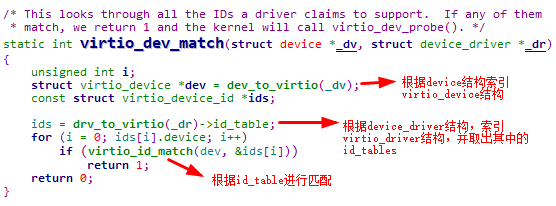
virtio\_bus以core\_initcall的方式被注册，该方式注册的启动顺序优先级很高（作为对比，module\_init最终对应的是device\_initcall），在使用中要注意不同组件的启动顺序

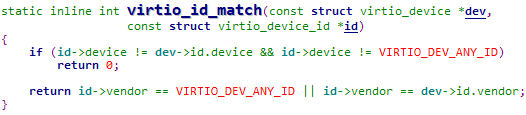
### 说明2：virtio\_dev\_match函数

virtio驱动的match涉及到virtio\_device\_id结构，在virtio\_device结构中包含该结构；在virtio\_driver中则是包含该驱动支持的virtio\_device\_id列表



具体的match流程如下，



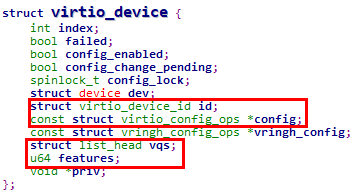


可见virtio驱动的match函数先匹配device字段，后匹配vendor字段，二者都满足条件时match成功

补充：从virtio\_dev\_match的流程可以看出，virtio\_driver中的id\_table必须以id->device = 0结尾，以便结束循环

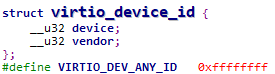
## virtio\_device结构

struct virtio\_device定义在include/linux/virtio.h中，具体如下，

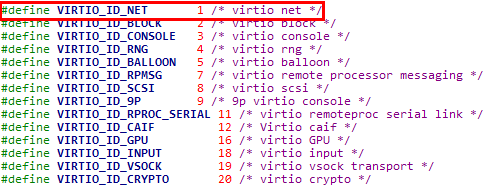


重点说明如下4个字段，

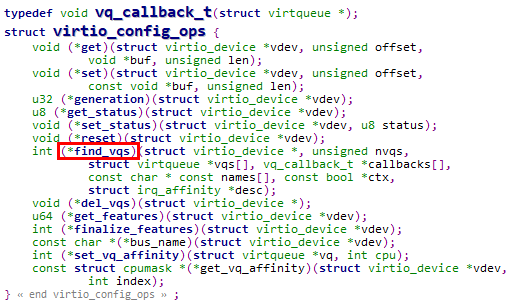
### struct virtio\_device\_id id



其中的device成员标识了当前virtio\_device的用途，virtio-net是其中的一种，



### const struct virtio\_config\_ops \*config



virtio\_config\_ops操作集中的函数主要与virtio\_device的配置相关，主要有如下2类操作，

① 实例化 / 反实例化virtqueue，其中要特别注意find\_vqs函数，该函数用于实例化virtio\_device所持有的virtqueue

②. 获取 / 设置virtio\_device的属性与状态，相关属性均在虚拟机虚拟出的PCI配置空间

### struct list\_head vqs

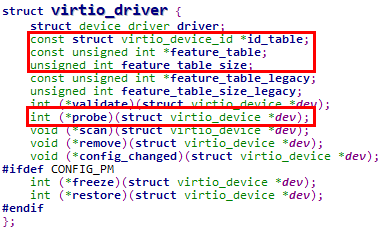
virtio\_device持有的virtqueue链表，virtio-net中建立了2条virtqueue（虚拟队列）

### u64 features

virtio\_driver & virtio\_device同时支持的通信特性，也就是前后端最终协商的通信特性

## virtio\_driver结构

struct virtio\_driver定义在include/linux/virtio.h中，具体如下，



重点说明如下4个字段，

### const struct virtio\_device\_id \*id\_table

对应virtio\_device结构中的id成员，virtio\_device中标识的是当前device的id属性；而virtio\_driver中的id\_table则是当前driver支持的所有id列表

### const unsigned int \*feature\_table & unsigned int feature\_table\_size

feature列表包含了当前driver支持的所有virtio传输属性，feature\_table\_size则说明属性数组的元素个数

### probe函数

virtio\_driver层面注册的probe函数，如上文所述，virtio\_bus层面也注册了probe函数，在Linux总线驱动框架 & virtio核心层中，当virtio\_device & virtio\_driver匹配成功后，先调用bus层面的probe函数，在virtio\_bus层面的probe函数中，又调用了virtio\_driver层面的probe函数

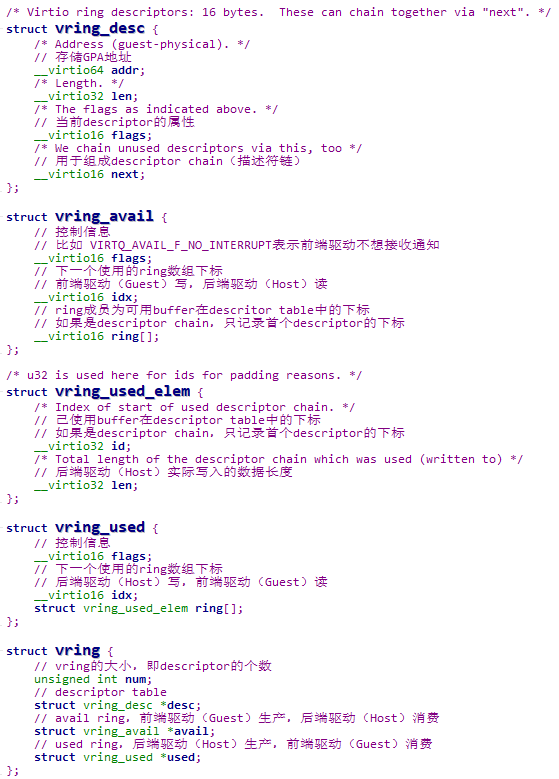
## virtqueue结构

struct virtqueue定义在include/linux/virtio.h中，具体如下，



## vring结构

struct virtqueue定义在include/uapi/linux/virtio\_ring.h中，具体如下，​​​​​​​



一定要结合一个后端驱动进行分析，可以对照rpmsg-lite分析

可以就分析rpmsg-lite对virtqueue的操作部分，不用上升到rpmsg协议的部分

### 说明1：vring的三个构成区域

① Destcriptor Table：描述内存buffer，主要包括addr & len等信息

② Avail Ring：用于前端驱动（Guest）通知后端驱动（Host）有可用的描述符

e.g. 前端驱动有一个报文需要发送，需要将其加入Avail Ring，之后通知后端驱动读取

③ Used Ring：用于后端驱动（Host）通知前端驱动（Guest）有可用的描述符，或者是后端驱动已将前端驱动提供的描述符使用完毕

e.g. 后端驱动有一个报文需要发送，需要将其加入Used Ring，之后通知前端驱动读取

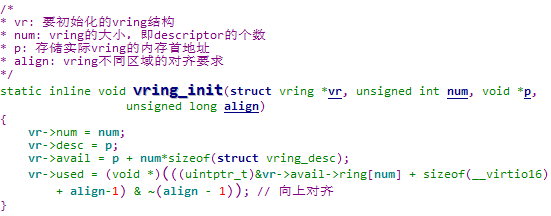
可见avail & used的命名都是站在Host的角度进行的

### 说明2：vring的存储

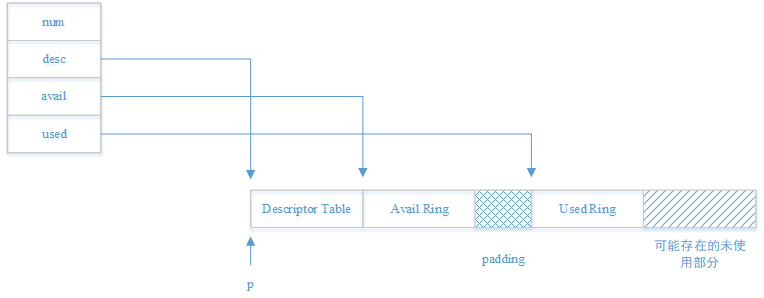
vring结构只是用于描述vring在内存中的布局（因此包含的都是指针变量），实际用于通信的vring是存储在内存中

上文提到的vring的三个区域是在内存中连续存储的，而且是存储在Guest & Host共享的一片连续内存中

我们可以通过vring\_init函数理解vring存储结构的布局



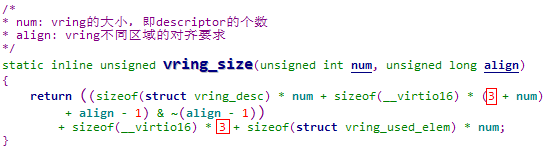
实际vring的内存布局如下图所示，



在计算used ring的起始地址时，在avail->ring[num]的地址之后又加了sizeof(\_\_virtio16)，也就是增加了2B，是为了容纳avail ring末尾的used\_event，该机制详见下文（是一种控制中断触发频率的机制）

### 说明3：实际vring的大小

实际vring的大小可以通过vring\_size函数获得

  
① 计算avail ring时加3，分别为flags、idx和used\_event

② 计算used ring时加3，分别为flags、idx和avail\_event

③ 计算过程中，包含了为满足对齐要求padding的空间

### 说明4：used\_event与avail\_event机制概述

这2个字段均与virtio设备的VIRTIO\_RING\_F\_EVENT\_IDX特性有关，由于virtio驱动触发对方中断将导致CPU反复进出虚拟机 & 宿主机模式，从而降低性能，因此需要控制触发中断频率的机制

① avail ring中的used\_event

a. 由前端驱动（Geust）设置，标识希望后端驱动（Host）触发中断的阈值

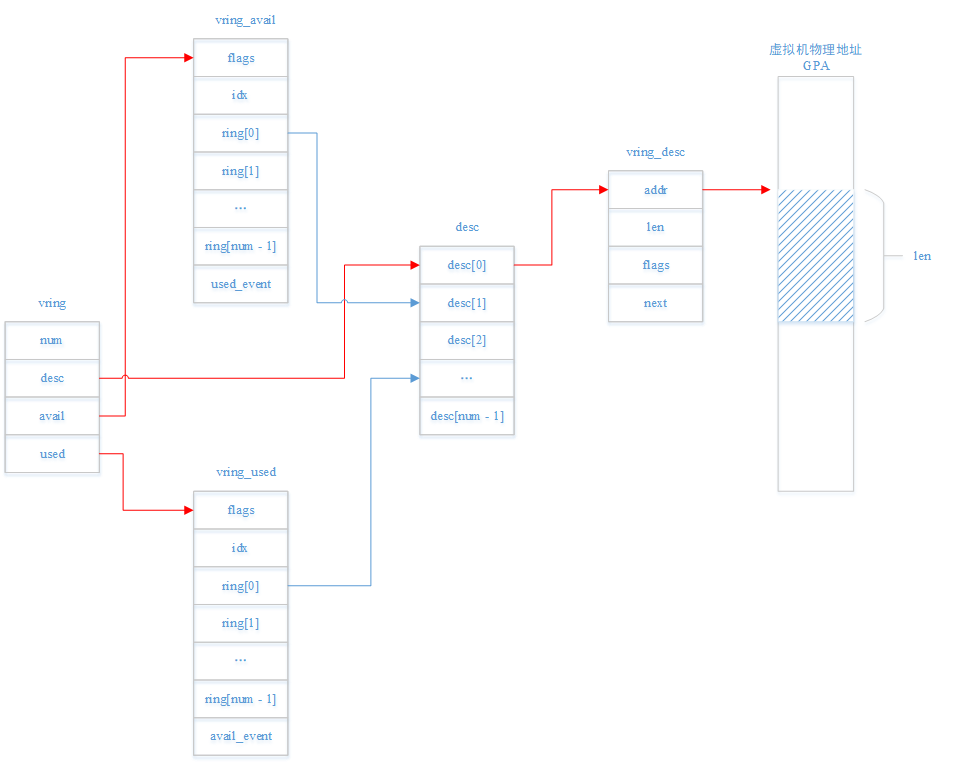
b. 后端驱动（Host）在向Used Ring加入buffer后，检查Used Ring中的idx字段，只有达到阈值才触发中断

② used\_ring中的avail\_event

a. 由后端驱动（Host）设置，标识希望前端驱动（Guest）触发中断的阈值

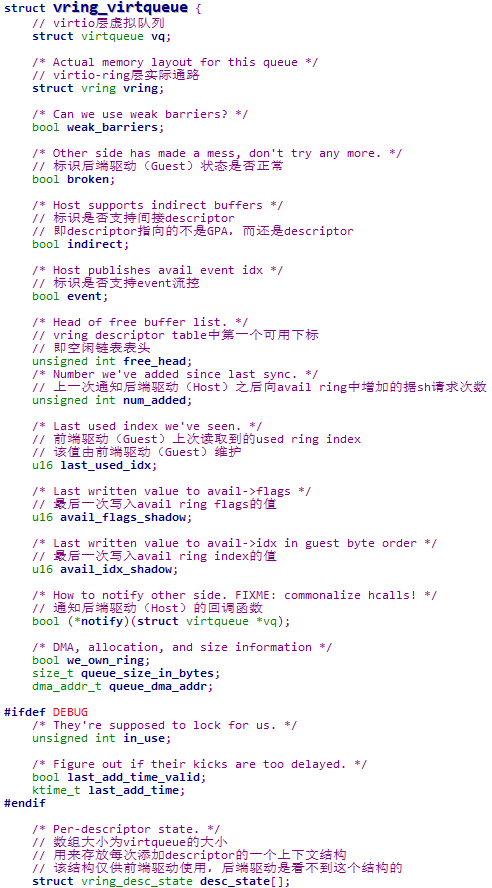
b. 前端驱动（Guest）在向Avail Ring加入buffer后，检查Avail Ring的idx字段，只有达到阈值才触发中断

综上所属，vring结构的构成如下图所示，

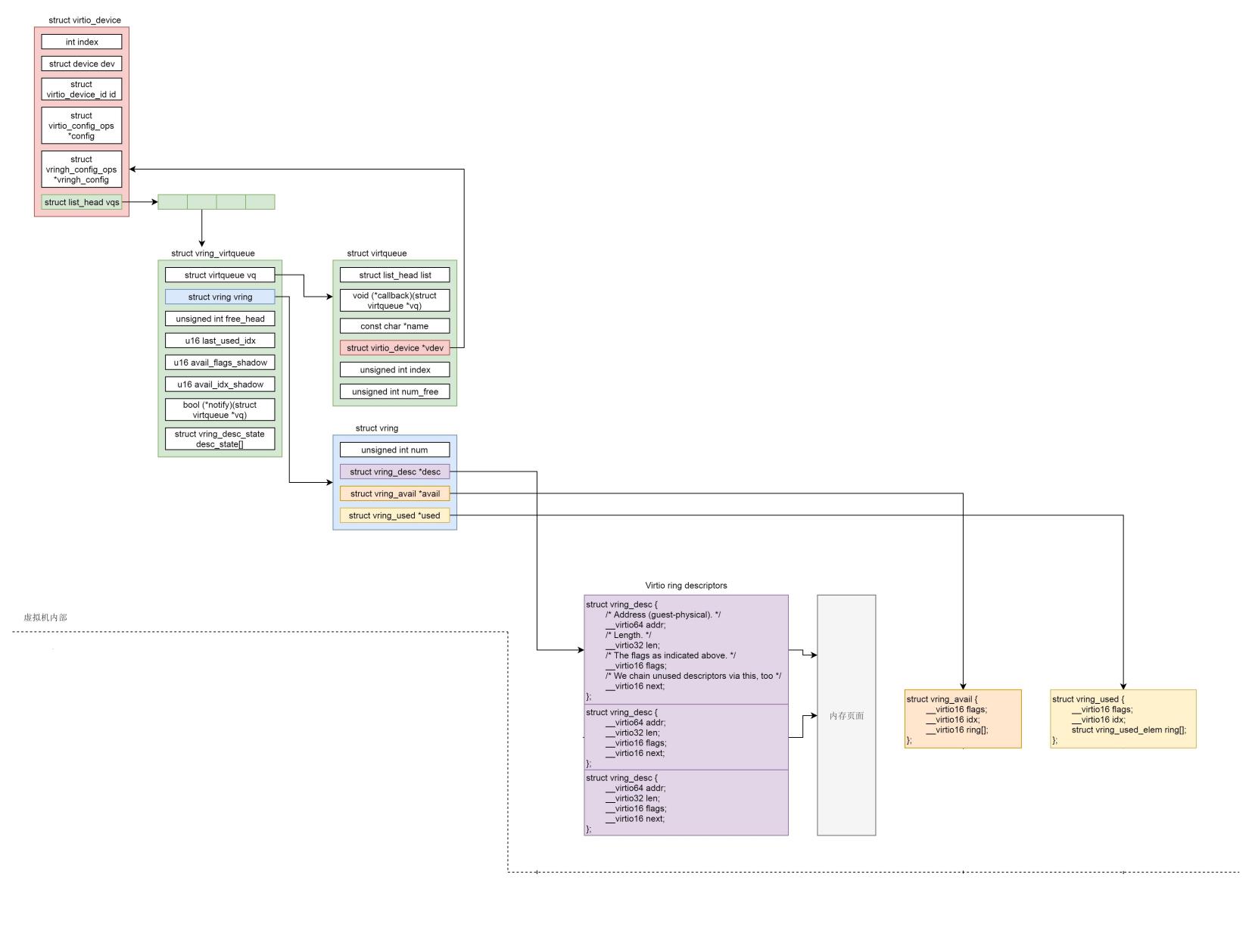


## vring\_virtqueue结构

vring\_virtqueue结构用于描述前端驱动（Guest）中的一条虚拟队列



总结：virtio\_device / vring\_virtqueue / virtqueue / vring结构之间的关系



# virtio操作

如上文所述，virtio框架向虚拟机中的前端驱动提供了统一的IO操作接口，我们下面就分析这些操作

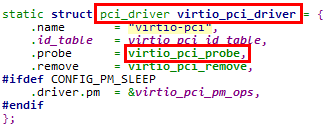
在理解了virtio的操作之后，配合不同虚拟设备的属性，就比较容易理解虚拟设备前端驱动的实现。比如virtio-net就是网卡驱动 + virtio操作

## 创建virtqueue

### 核心流程梳理（重点）

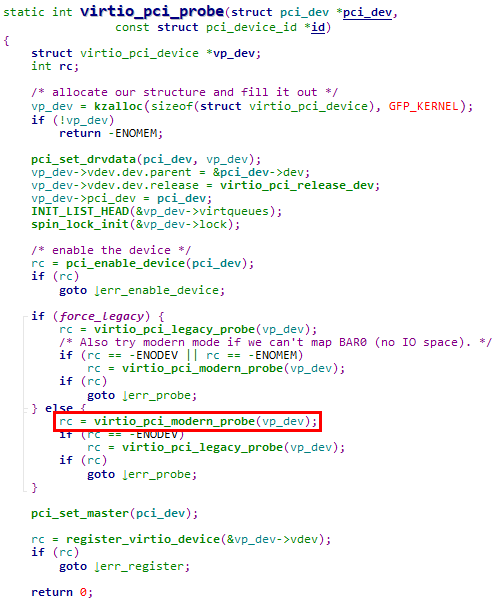
#### virtio\_pci\_probe阶段

在virtio框架中，**首先向虚拟机注册的是一个pci\_driver，后续所有vitio设备均是以虚拟pci设备的形式注册**，因此第1步流程即是运行virtio\_pci\_probe函数



#### virtio\_pci\_modern\_probe阶段

在virtio\_pci\_probe函数中，会调用virtio\_pci\_modern\_probe函数，进行进一步初始化



在virtio\_pci\_probe的最后阶段，会注册virtio\_device，该操作会触发virtio驱动的probe函数被调用，在该函数中，会触发find\_vqs回调函数被调用

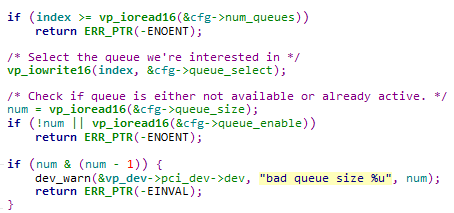
下面我们以virtio-net前端驱动为例，说明创建virtqueue的流程

下面我们就分析最核心的几个函数

### setup\_vq函数分析

setup\_vq函数有如下3个核心步骤，

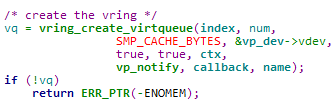
#### 检查virtqueue配置参数



前端驱动读取PCI配置空间中的参数，判断其合法性

其中要注意virtqueue的长度（queue\_size）必须是2的幂次，因为后续需要通过简单的位与运算实现绕回

#### 实际生成virtqueue



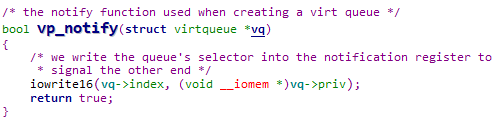
实际生成virtqueue通过vring\_create\_virtqueue函数实现，此处需要注意如下3点，

① 对齐要求

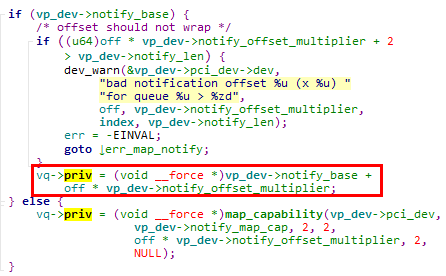
如上文所述，vring结构在内存布局上有对齐要求，该要求在创建virtqueue时传递，就是此处的SMP\_CACHE\_BYTES宏

② notify hook函数

notify hook函数用于前端驱动（Guest）触发后端驱动（Host）中断，通知其有数据需要处理

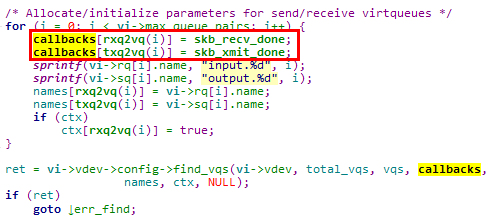


这里的notification register也在PCI配置空间中，该地址在setup\_vq函数中指定

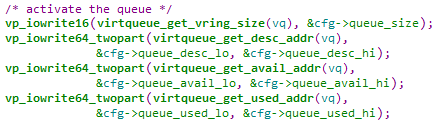


③ callback hook函数

callback hook函数在virtqueue被触发时调用，以virtio-net驱动为例，callback hook函数在virtnet\_find\_vqs函数中指定



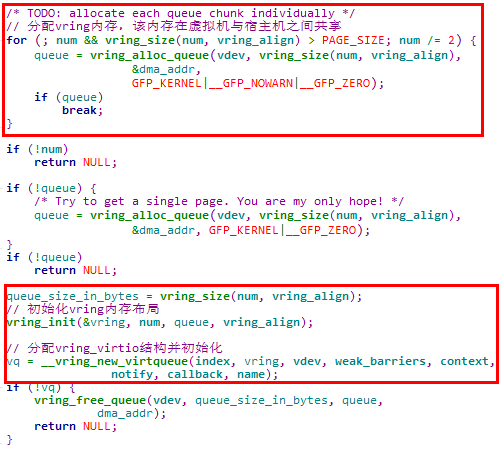
#### 同步GPA到宿主机



virtqueue作为前端驱动与后端驱动的交互媒介，需要在虚拟机和宿主机中同步这段共享内存的地址

调用vring\_create\_vritqueue函数生成的virtqueue，分配的内存为GPA，需要将其同步到宿主机，宿主机才能将其转换为HVA使用（因为虚拟机的GPA就是宿主机分配的）

### vring\_create\_virtqueue函数分析



可见vring的内存被分配在连续的1个或多个page中，而且如果内存条件不满足，会动态调整vring的长度（num变量）

### \_\_vring\_new\_virtqueue函数分析



\_\_vring\_new\_virtqueue函数的实现注释已经比较清楚了，需要说明的是，该函数返回的是virtqueue结构。在Linux的virtio层实现中，代码会根据需要在virtqueue与vring\_virtqueue结构间进行转换

## 前端驱动发送数据

### 流程概要

step 1：从descriptor table中取出描述符

step 2：根据要发送的数据填充并组织描述符

step 3：将组织好的描述符加入avail ring

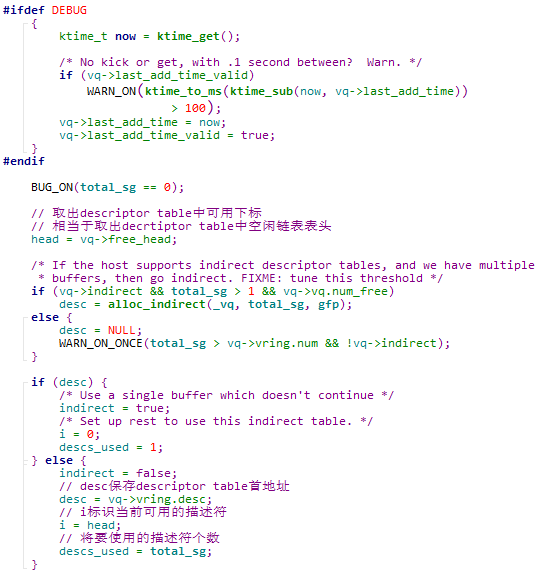
step 4：触发后端驱动中断，通知其处理数据

说明：vring的描述符结构与scatterlist结构是绝配

### virtqueu\_add函数分析

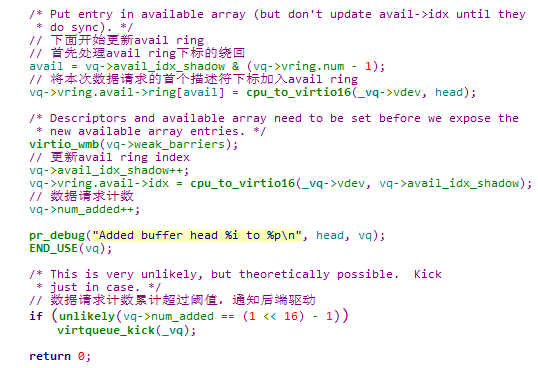
前端驱动发送数据的核心为virtqueue\_add函数，下面给出该函数的分析



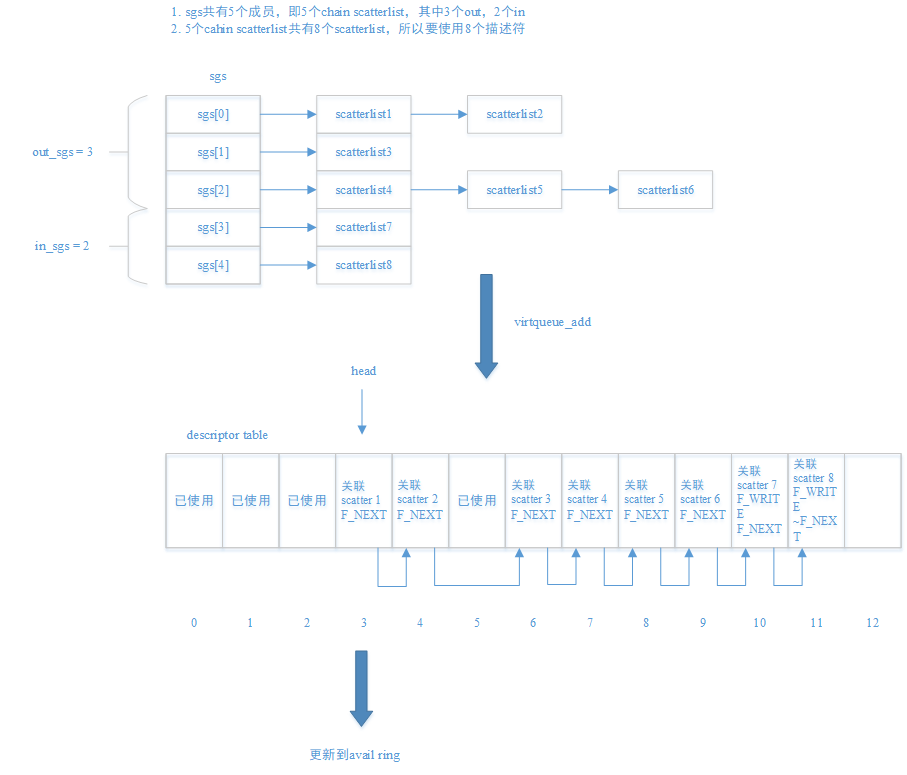








下面通过一张图展现该过程，



说明1：可见virtqueue\_add函数封装了一次数据请求，所有out & in请求均组织为一个decriptor chain提交到avail ring

从上文分析可见，如果将out & in数据请求组织在一起，将使得接收端的处理逻辑非常复杂。因此在实际使用中（e.g. virtio-net，rpmsg），一般为out & in的数据请求单独建立virtqueue，即输入和输出使用不同的虚拟队列

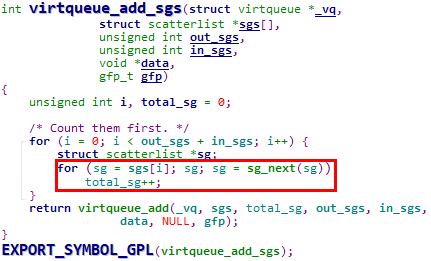
说明2：由上图可知，descriptor table以静态链表的方式管理，因此空闲链表中各个描述符在物理上不一定是连续的，而是依靠描述符中的next域维护链接关系

说明3：对virtqueue\_add函数的使用

virtqueue\_add函数被封装为如下4种方式供前端驱动调用，

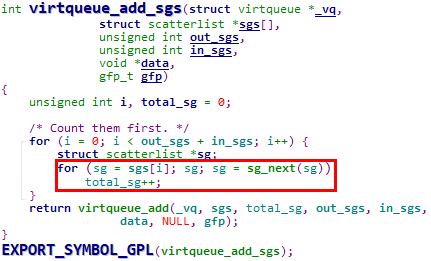
① virtqueue\_add\_sgs

可以同时提交out & in数据请求，且个数可设置



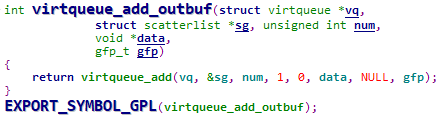
① virtqueue\_add\_sgs

可以同时提交out & in数据请求，且个数可设置



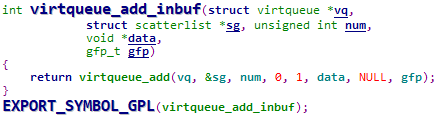
② virtqueue\_add\_outbuf

只提交一个out数据请求



③ virtqueue\_add\_inbuf

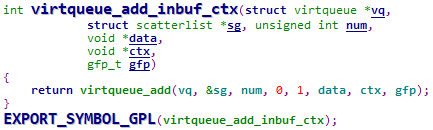
只提交一个in数据请求



④ virtqueue\_add\_inbuf\_ctx

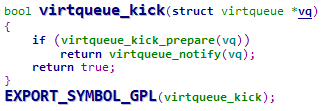
只提交一个in数据请求，且携带上下文信息

注意：ctx上下文信息与inderect特性是互斥的



## 前端驱动触发中断

前端驱动通过virtqueue\_kick函数通知后端驱动有数据需要处理

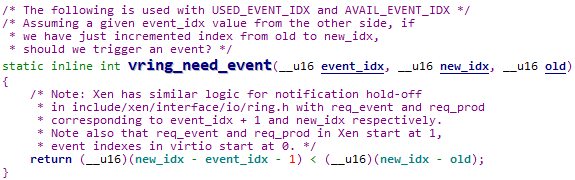


其中virqueue\_kick\_prepare函数判断是否需要触发中断，virtqueue\_notify函数实际触发中断

### virtqueue\_kick\_prepare函数分析

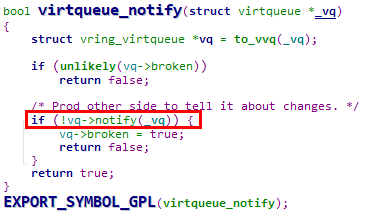


说明：vring\_need\_event函数实现

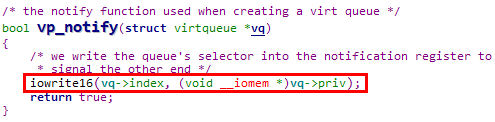


只有当event\_idx在[old, new - 1]范围时，才会允许触发中断

### virtqueue\_notify函数分析



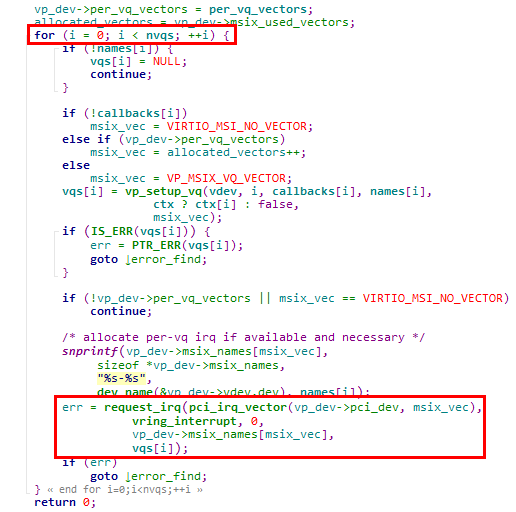
virtqueue\_notify会调用上文介绍的notify回调函数，实现对后端驱动的通知，在本文环境中，该回调函数为vp\_notify函数



## 前端驱动被触发中断

### 注册中断处理函数

在创建virtqueue时，会为每条virtqueue注册中断，可参考vp\_find\_vqs\_msix函数

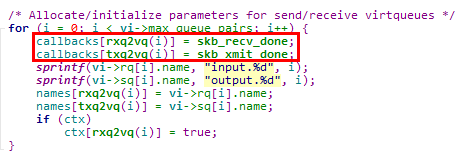


可见中断处理函数为vring\_interrupt，注意这里注册的是msi中断(总线中断)

### vring\_interrupt函数分析

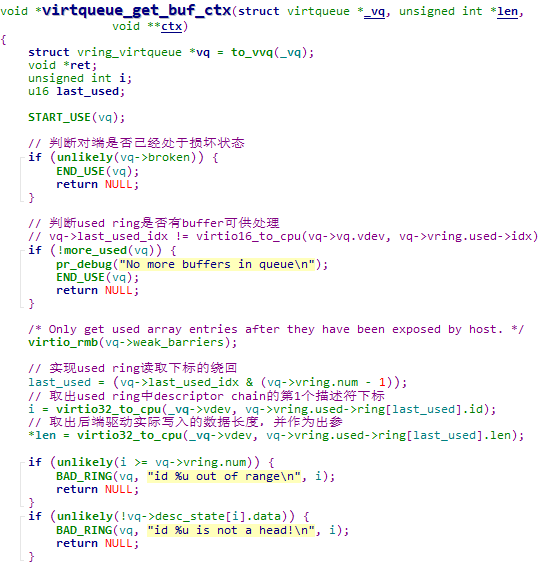


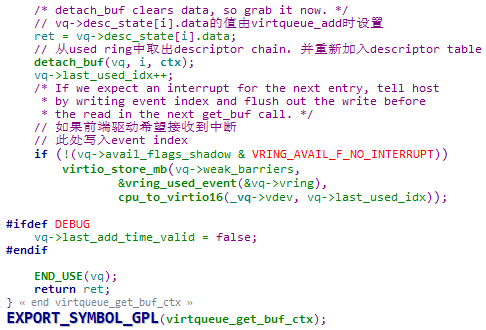
vring\_interrupt函数的核心操作是调用创建virtqueue时注册的callback回调函数，以virtio-net模块为例，接收和发送队列注册的callback回调函数如下



## 前端驱动接收数据

### virtqueue\_get\_buf\_ctx函数分析





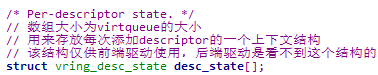
说明：virtqueue\_get\_buf\_ctx函数的返回值为vq->desc\_state[i].data，该值在调用virtqueue\_add时设置

virtqueue\_add写入该值，目的就是用于索引buffer（the token identifying the buffer）

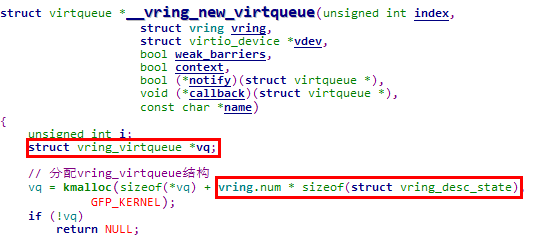
### 补充：对vring\_desc\_state desc\_state结构中data成员的使用

#### 在vring\_virtqueue结构中定义

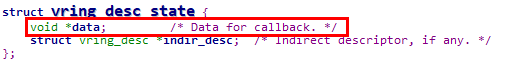
如上文所述，在vring\_virtqueue结构中定义了desc\_state数组，根据注释，该结构描述了每个描述符的状态（更好的理解是每个描述符有一个）



数组大小为virtqueue大小，空间随vring\_virtqueue结构一同分配，该数组用于存储每次数据传输请求的上下文

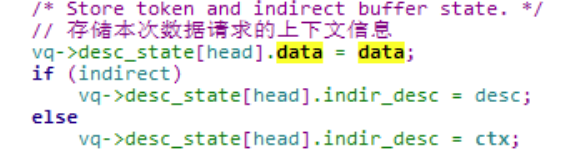


vring\_desc\_state结构如下，



我们这里就是讨论其中data成员的使用

#### 在virtqueue\_add函数中设置



这里注意2点，

① 填入data的值

此处填入的值为virtqueue\_add函数的入参data

② 填入desc\_state数组的下标

此处使用的下标为head，为本次数据请求的chain descriptor的首个描述符下标

#### 在virtqueue\_get\_buf\_ctx中读取

IMG_256

此处使用的下标i是used ring中取出的chain descriptor中首个描述符的下标，这里对应了一次vritqueue\_add加入的数据请求

此处就将当时virtqueue\_add函数写入的data作为返回值

说明：这里就可以看出virtio机制设计的巧妙之处，后端驱动在使用不同的chain descriptor后不需要按取出的顺序归还

这里有2点机制上的保障，

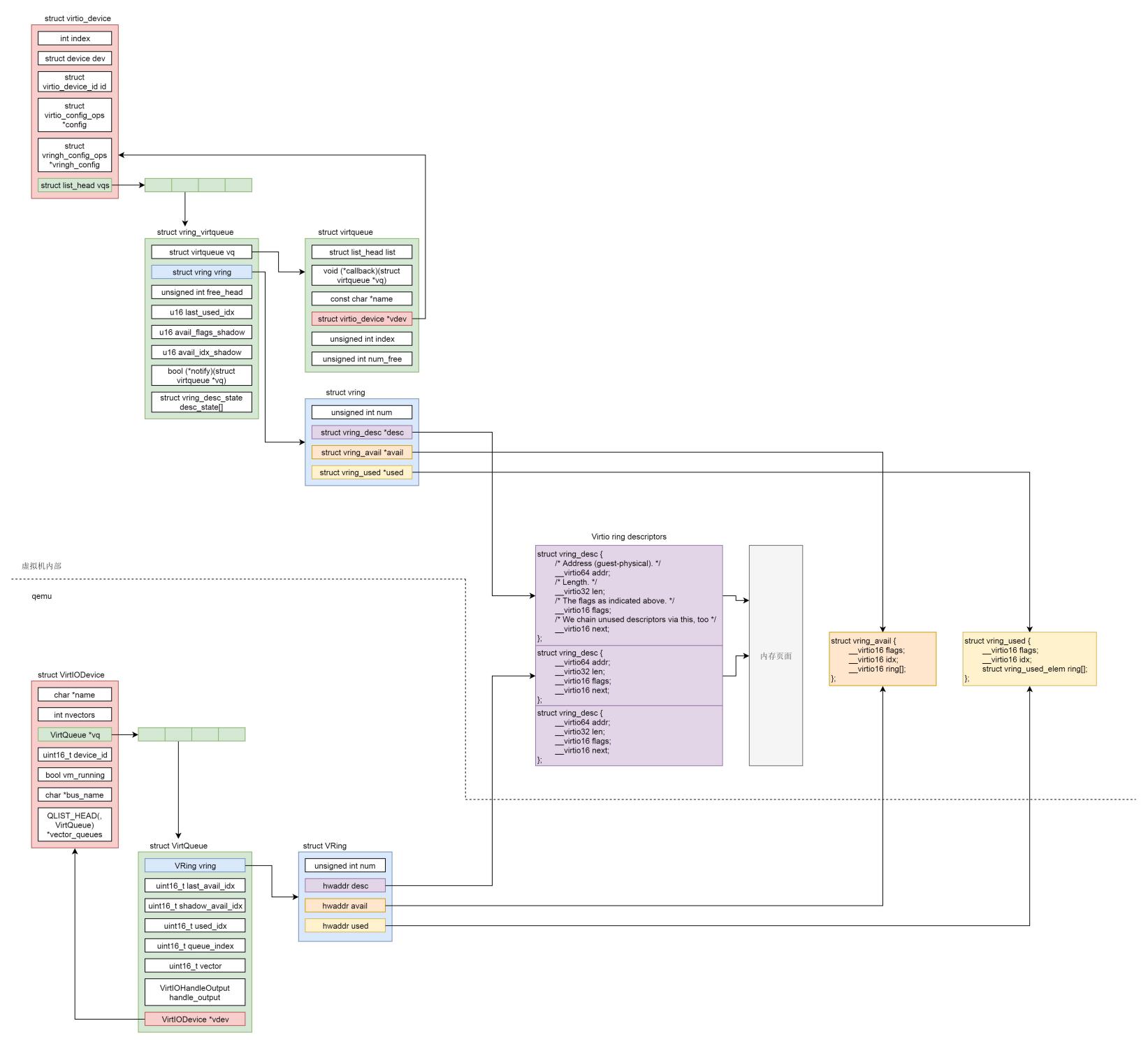
① descriptor table使用静态链表方式管理

② desc\_state数组按描述符管理

### detach\_buf函数分析



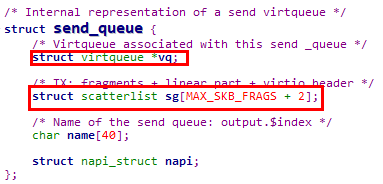
最终给出一张图，就是虚拟机和宿主机指向同一段内存，以实现二者之间的交互



# virio-net前端驱动分析

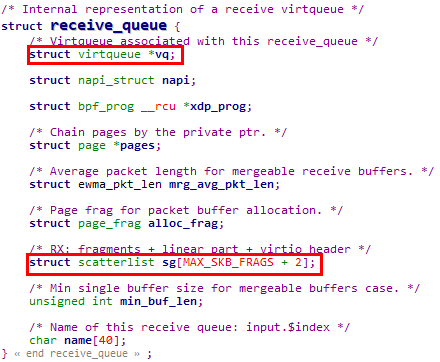
## 重要数据结构

### send\_queue结构



send\_queue结构是对virtqueue的封装，是virtio-net的发送队列，即数据流向从前端驱动（Guest）到后端驱动（Host）

### receive\_queue结构



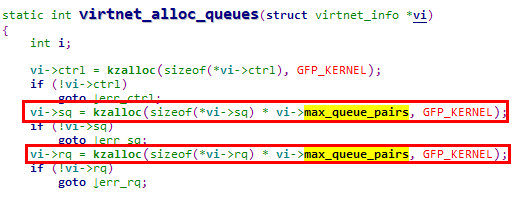
receive\_queue结构也是对virtqueue的封装，是virtio-net的接收队列，即数据流向从后端驱动（Host）到前端驱动（Guest）

说明：multiqueue virtio-net

virtio-net前端驱动支持multiqueue机制，也就是允许有多对send\_queue & receive\_queue，在virtnet\_probe过程中会检查宿主机的设置，获取收发队列的对数

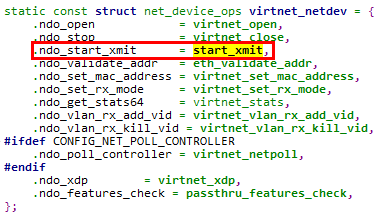
IMG_256

这样在创建virtqueue时，会根据配置项分配内存



但是在一般情况下，均使用1条send\_queue + 1条receive\_queue，且没有控制队列

### virtnet\_netdev callback数组



在Linux中，net\_device结构描述了一个网络设备，其中的net\_device\_ops则包含了该网络设备的操作方法集

其中特别注意ndo\_start\_xmit callback函数，该函数为网卡发送报文时使用的函数

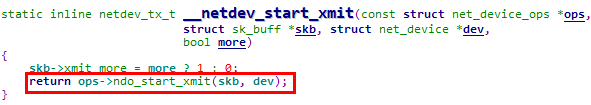
## 发送报文流程

### 到达start\_xmit函数

① 虚拟机中的进程发送网络包时，仍然通过文件系统和socket调用网络协议栈到达网络设备层。只不过此时不是到达普通的网络设备，而是virtio-net前端驱动

② virtio-net前端驱动作为网卡设备驱动层，接收IP层传输下来的二层网络数据包

③ 发送网络包的流程最终将调用net\_device\_ops结构中的ndo\_start\_xmit回调函数，在virtio-net驱动中，就是start\_xmit函数



### start\_xmit函数主要流程

与virtio框架相关的只有2个步骤，

① 调用xmit\_skb函数将网络包写入virtqueue

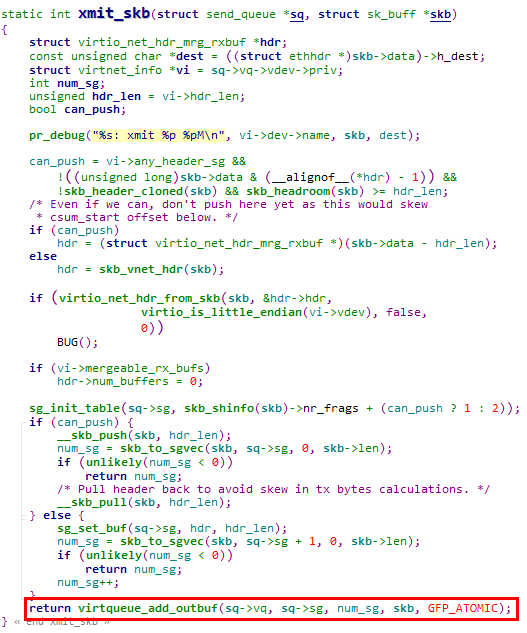
IMG_256

② 触发后端驱动中断

IMG_257

virtqueue\_kick函数在上文已有说明，此处说明一下xmit\_skb函数的实现

### xmit\_skb函数



xmit\_skb函数将sk\_buff映射到scatterlist中，之后调用virtqueue\_add\_outbuf函数将数据请求加入send\_queue的avail ring

说明：这里传递给data的值为skb，也就是要发送的skb的地址。注意，skb的地址值是一个GVA（Guest Virtual Address），因此只在虚拟机中使用

## 接受报文流程

### NAPI接收网络包流程概述

① 传统的网络收包流程完全靠中断驱动，当网络包到达十分频繁时，就会频繁触发中断，进而影响系统的整体性能

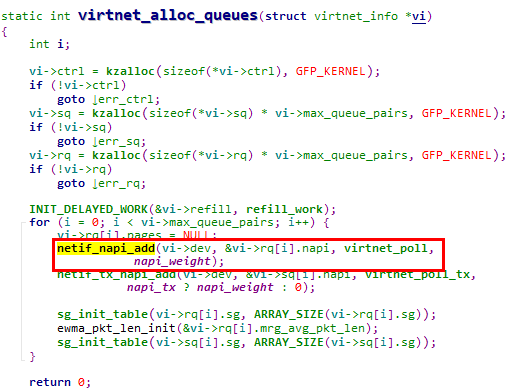
② NAPI方式的核心就是当有数据包到达时，集中处理网络包，之后再去处理其他事情

③ NAPI的处理流程是，当一些网络包到达触发中断时，内核处理完这些网络包之后，主动轮询poll网卡，主动去接收到来的网络包。如果一直有，就一直处理，等处理告一段落再返回

当再有下一批网络包到达时，再中断，再轮询poll。这样就会大大减少中断的数量，提升网络处理的效率

说明：注册NAPI收包poll函数

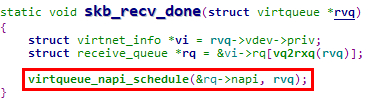
在virtio-net前端驱动中，在probe过程中，会调用netif\_napi\_add函数注册收包poll函数

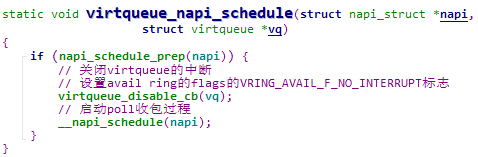


可见此处注册的函数为virtnet\_poll

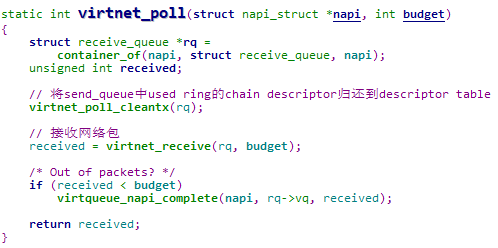
### 可见此处注册的函数为virtnet\_poll

如上文所述，virtqueue的中断处理函数最终会调用到创建virtqueue时注册的callback回调函数，该函数为 skb\_recv\_done，这也就是virtio-net前端驱动的收包中断顶半部操作



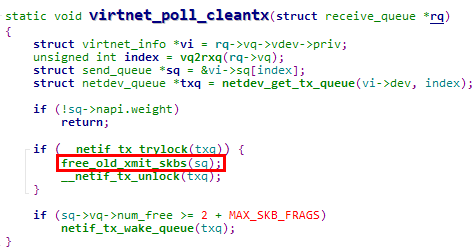


### virtnet\_poll函数分析



说明：virtnet\_poll\_cleantx函数分析

在接收数据报文之前，先调用了virtnet\_poll\_cleantx函数处理了send\_queue



其中的核心为free\_old\_xmit\_skbs函数，分析如下，



这里也很好地体现了vring\_desc\_state结构中data成员的使用，

① 前端驱动发送报文时，将含有报文的skb写入data成员，数据请求加入avail ring

② 后端驱动处理完数据请求后，将chain descriptor从avail ring加入used ring

③ 前端驱动在处理后端驱动已使用的chain descriptor时，从data成员中取出skb地址，并释放sk\_buffer

### virtnet\_receive函数分析前奏

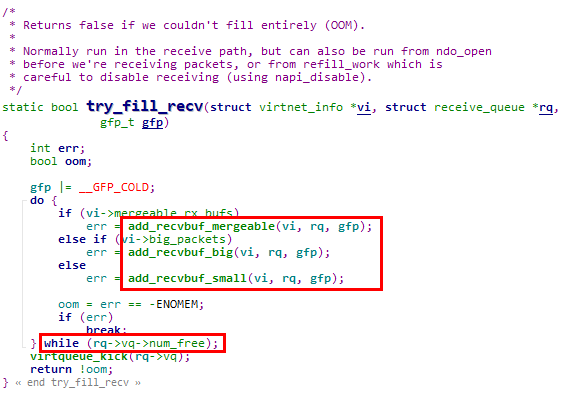
首先思考一个问题，receive\_queue中的avail ring是何时填充的 ?

receive\_queue的数据流向是从后端驱动到前端驱动，但是前端驱动需要先将数据请求加入avail ring，这样后端驱动在要发送网络包时，才能从avail ring中取出可用的chain descriptor

而且这里还带来另外一个问题，前端驱动是不知道后端驱动所要发送的报文大小的，那么该如何组织descriptor ring呢 ?

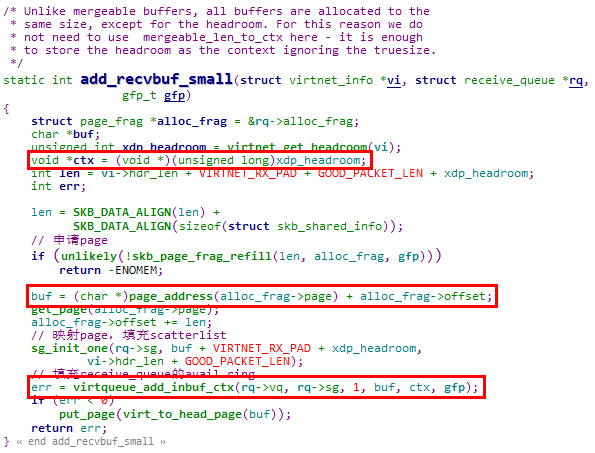
结合上文，这里解题的线索就是virtqueue\_add\_inbuf & virtqueue\_add\_inbuf\_ctx函数在virtio-net前端驱动中的调用

这样我们就很容易地找到关键的函数，try\_fill\_recv !



可见try\_fill\_recv函数会将所有可用的描述符均加入receive\_queue的avail ring，供后端驱动使用

我们分析add\_recvbuf\_small函数，另外两种情况需要后端驱动配置支持



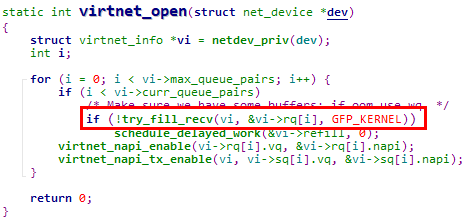
这里需要注意调用virtqueue\_add\_inbuf\_ctx的2个参数，因为后续的接收报文流程会使用

data：实参为buf，即分配的内存页面的GVA

ctx：实参为ctx，值为xdp\_headroom

说明：try\_fill\_recv函数的调用时机

① 打开网卡时

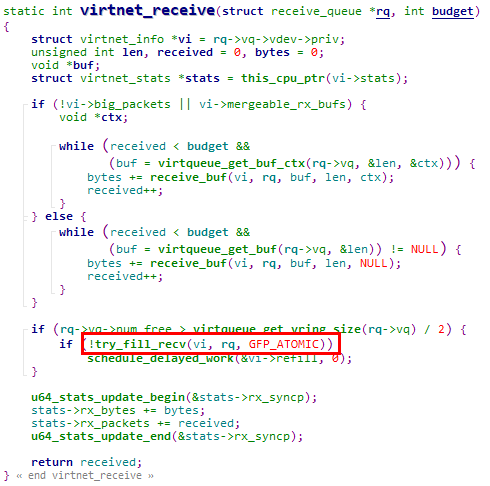


其中调度vi->refill工作，也会导致try\_fill\_recv函数被调用

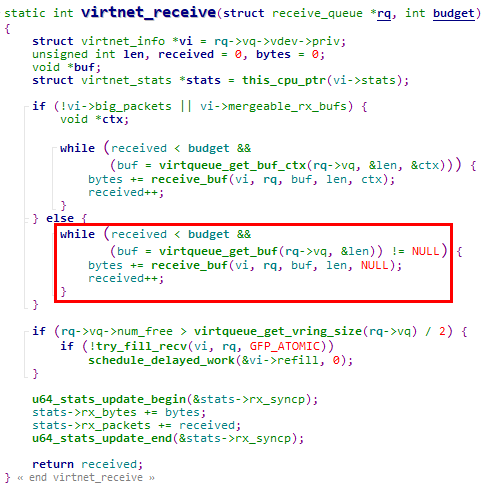
② 网卡restore时



③ 接收报文时，也就是接下来要分析的函数



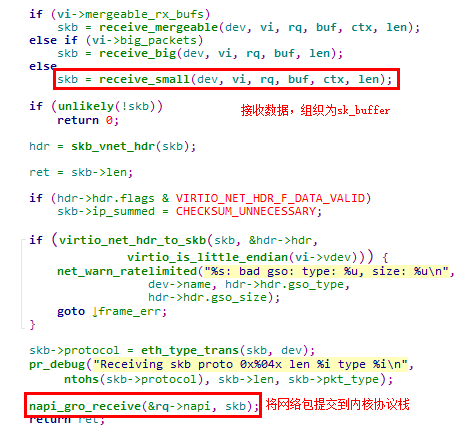
### virtnet\_receive函数分析



① 调用virtqueue\_get\_buf函数

将receive\_queue中used ring的chain descriptor归还descriptor table，返回的buf就是上文分析的分配的内存的GVA，该地址在虚拟机中可以使用

② 调用receive\_buf函数接收报文数据



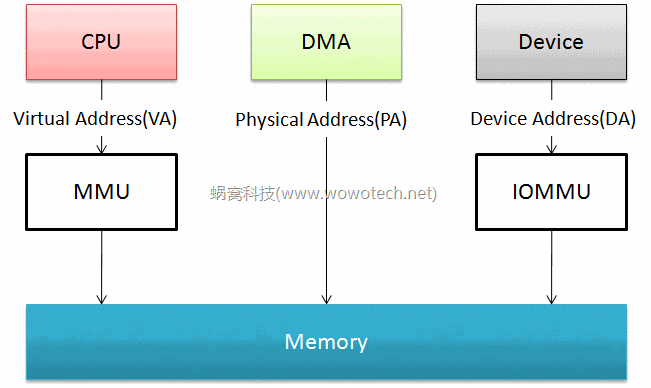
至此，virtio-net前端驱动接收报文的工作就结束后，后续就是虚拟机Linux内核网络协议栈的工作了

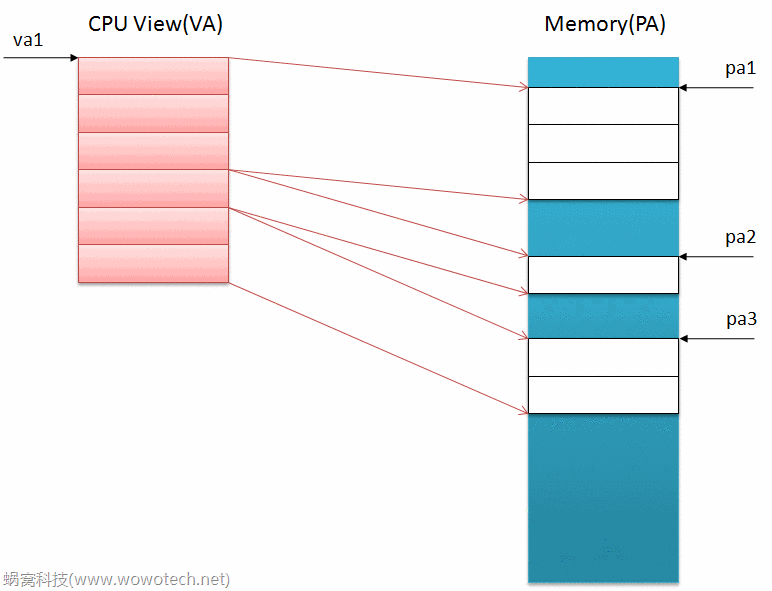
# Linux virtio-net中对内存的使用

## scatterlist 实现分析

### catterlist 产生背景

scatterlist用于汇总分散的物理内存（以页为单位），并以数组的形式组织起来，典型的应用场景如下图所示，





在一个系统中，CPU、DMA和Device通过不同的方式使用内存，

① CPU通过MMU以虚拟地址（VA）访问内存

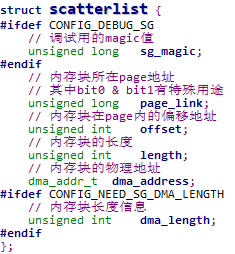
② DMA直接以物理地址（PA）访问内存

③ Device通过自己的IOMMU以设备地址（DA）访问内存

如果访问的内存虚拟地址连续但是物理地址不连续，CPU的访问没有问题，但是当需要将内存地址交给DMA进行传输时，只能以不连续的物理内存块的方式传递

而scatterlist就是用户汇总这些不连续的物理内存块的方式

### scatterlist结构

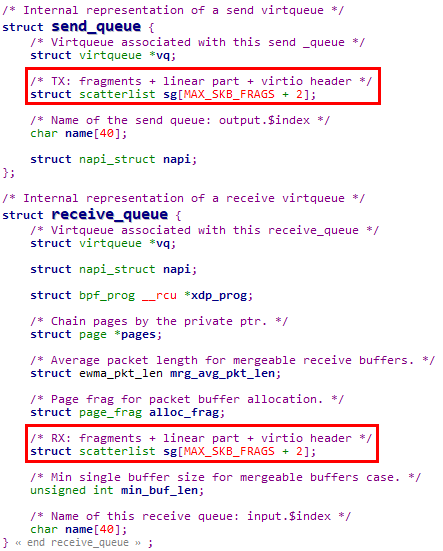


scatterlist以page为单位，描述了一个物理地址连续的内存块

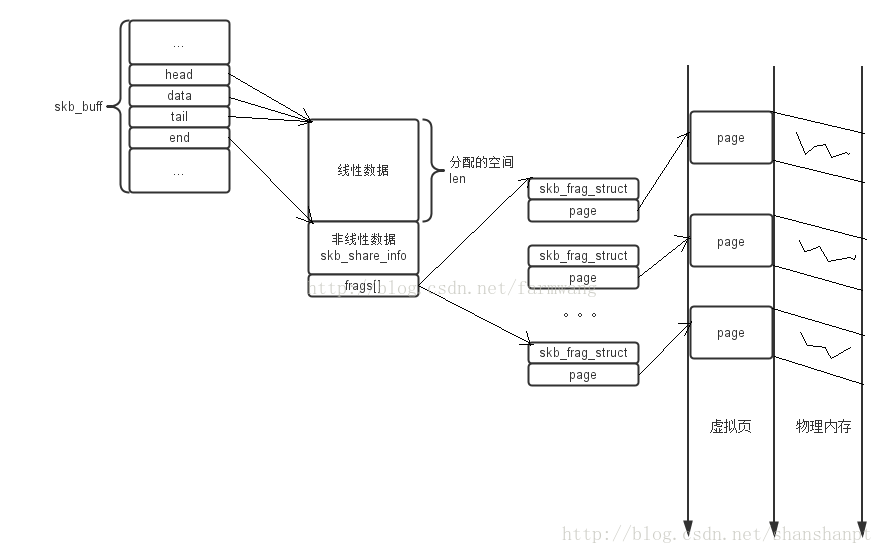
说明1：如果要组织的连续物理内存超过一页怎么办 ?

要组织的连续物理内存超过一页是常态，所以单个scatterlist结构是没啥实际用途的。在实际使用中，Linux内核默认将scatterlist组织为数组使用

在virtio-net前端驱动中，收发队列中均包含了scatterlist数组



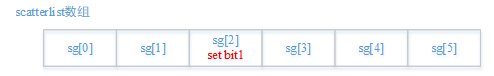
需要注意的是，这里scatterlist数组的大小与sk\_buff中分片的个数是匹配的，这里增加的2个scatterlist分别用于存放sk\_buff的线性数据部分和virtio-net的头部信息，可以参考下图理解



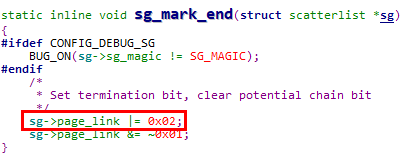
说明2：page\_link中bit1的作用

page\_link中的bit1是数组有效成员终止位，因为一次传输不一定使用scatterlist数组的所有成员，因此需要对最后一个有效的成员进行标记

下图中，一个scatterlist数组有6个成员，但是本次传输只使用其中3个

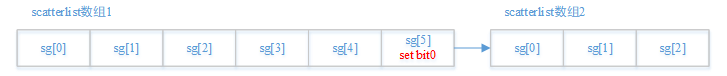


Linux内核代码中通过如下接口设置 & 检查该标志位

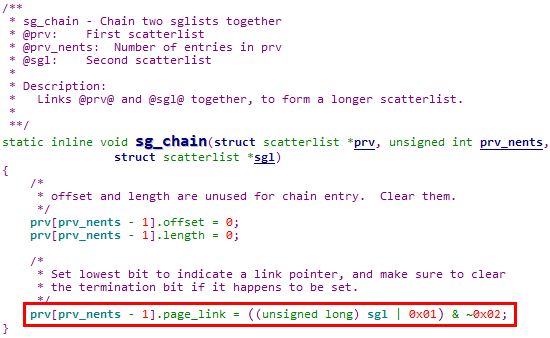


说明3：page\_link中bit0的作用

page\_link中的bit0是sacatterlist数组链接标志，用于实现将2个scatterlist数组链接起来。如果bit0置1，则该page\_link指向的不是一个page结构，而是指向另一个scatterlist数组



Linux内核代码中通过如下接口设置 & 检查该标志位



可见如果需要链接2个scatterlist数组，前一个数组的最后一个成员不能指向有效page

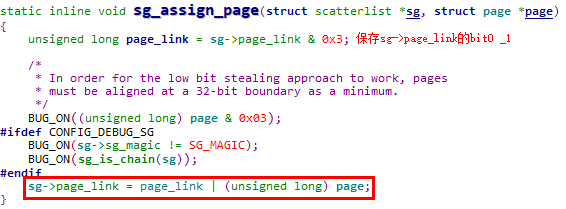
看到这里，就更容易理解之前分析的virtqueue\_add函数

### scatterlist常用API

#### sg\_init\_table

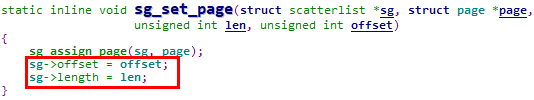


#### ****sg\_assign\_page****



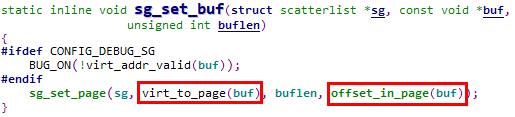
sg\_assign\_page函数将一个page与一个scatterlist关联起来

#### sg\_set\_page



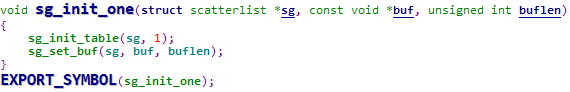
sg\_set\_page在关联page的基础上，设置了内存块的偏移量与长度

#### sg\_set\_buf



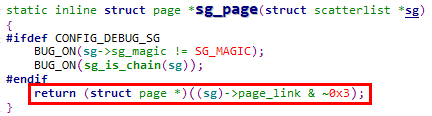
sg\_set\_buf函数是最常用的关联内存块与scatterlist的API，此处传入的buf参数为内存块起始的虚拟地址

#### sg\_init\_one



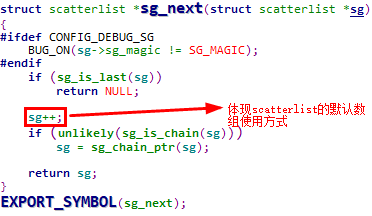
sg\_init\_one用于初始化一个scatterlist结构，并与一个内存块关联（该内存块必须在1个page内）

#### sg\_page



sg\_page返回与scatterlist关联的物理页面地址

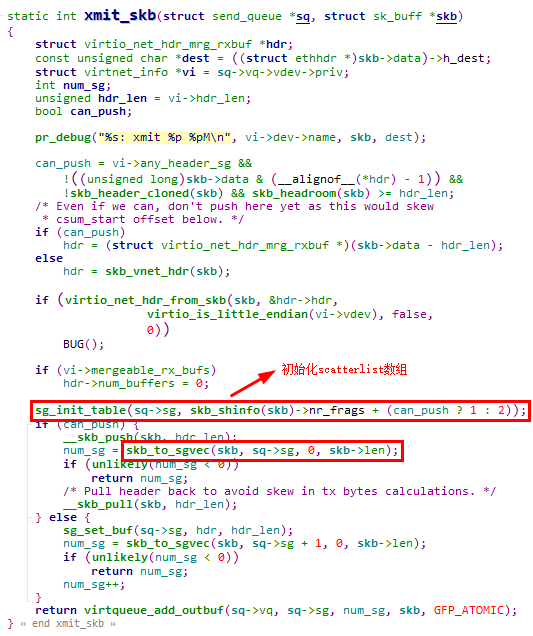
#### sg\_next



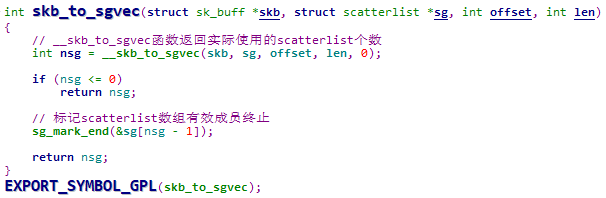
sg\_next用于取出scatterlist数组中的下一个成员，如果达到终止成员，则返回NULL

## virtio-net发送数据中的内存操作

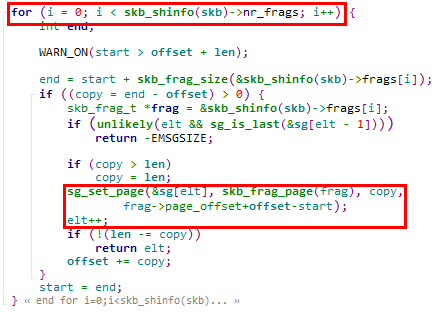
### 将sk\_buff关联到scatterlist数组



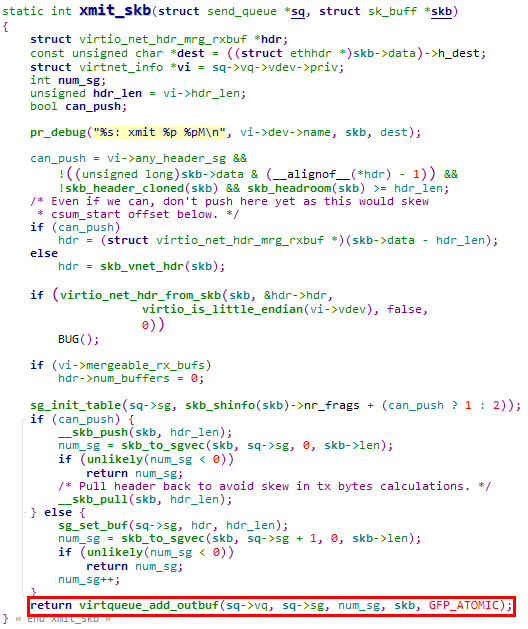
这里的核心是skb\_to\_sgvec函数，该函数用于将sk\_buff中存储报文用的各个page关联到scatterlist数组，下面分析该函数



在\_\_skb\_to\_sgvec函数中，将sk\_buff的逐个分片都关联到scatterlist数组中



### 将scatterlist数组映射到vring描述符



这里其实就回到了我们之前分析的virtqueue\_add函数

## virtio-net接收数据中的内存操作

备忘录：

topic 2：seL4中如何对接virtio-net

topic 3：virtio-net的上下游模块

topic 4：宿主机如何注册pci device，可以先分析qemu的实现思路

topic 5：SKB buffer的使用（这个属于网络相关知识点的补强）