Understanding Linux Network Internals

可以随意转载。作者：金庆辉

Chapter 2. Critical Data Structures

"Show me your data," said the legendary software engineer, Frederick P. Brooks.(我想，大师级程序员应该能从关键数据结构中推测出整个系统的外貌。)

2.1. The Socket Buffer: sk_buff Structure

sk_buff数据结构是Linux网络代码中最重要的数据结构,它定义在<include/linux/skbuff.h>。它粗略的被划分为如下分类：

- Layout
- General
- Feature-specific
- Management functions

2.1.1. Networking Options and Kernel Structures

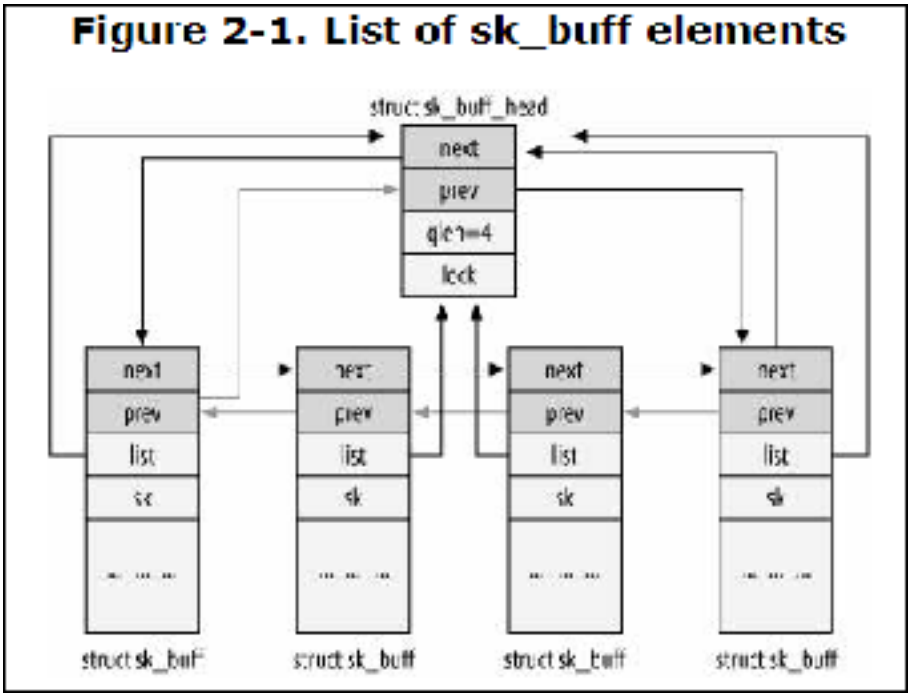
2.1.2. Layout Fields

sk_buff结构以双链表的形式组织在一起。为了让每个sk_buff结构能够快速的找到链表头，一个额外的数据结构sk_buff_head被插入到sk_buff链表的头部，它的数据结构是：

```
struct sk_buff_head {
    /* These two members must be first. */
    struct sk_buff    * next;
    struct sk_buff    * prev;

    _ _u32        qlen;
    spinlock_t    lock;
};
```

sk_buff_head和sk_buff头两个成员是一样的:next和prev指针。这就使这两个数据结构能同时存在于同一个链表中，即使sk_buff_head比起sk_buff少了很多成员。另外,同一个函数能用来操作sk_buff_head和sk_buff。

每个sk_buff含有一个指向sk_buff_head结构的指针，指针名字是list。下图是链表组织图：



sk_buff中其它成员:

1. struct sock *sk:这个指针指向拥有该buffer的socket的sock数据结构。当数据是被本地产生的或是被本地进程接受的,这个指针才有效。当buffer只是用来转发数据包(数据包的源地址和目的地址都不是本机),这个指针为NULL。
2. unsigned int len:buffer中数据块的大小。这个长度包括了buffer中的数据和分片中的数据。len值随着数据包在协议栈的不同层次中移动会改变。len也包含了协议头的大小。
3. unsigned int data_len:data_len仅仅计算分片中的数据大小。
4. unsigned int mac_len:MAC头的大小。
5. atomic_t users:sk_buff的使用计数。当使用计数为0时，sk_buff才会真正被释放。这个计数器只对sk_buff有效，buffer包含的真正的数据也有个计数器(dataref)，将在以后的章节中介绍。
6. **unsigned int truesize:它代表了buffer的总大小，包括sk_buffer结构。这个值被函数alloc_skb初始化为len+sizeof(sk_buff)。**

```
struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)
{
        ... ... ...
        skb->truesize = size + sizeof(struct sk_buff);
        ... ... ...
}
```

   当skb->len增加时会更新truesize。(还有疑点!!!)

7. unsigned char *head,*end,*data,*tail:它们代表了buffer的边界和buffer中数据的边界。head和end指向分配给buffer的空间的开始和结尾，data和tail指向真实数据的开始和结尾。如下图,在head和data之间能填入协议头,在tail和end之间能填入新数据。

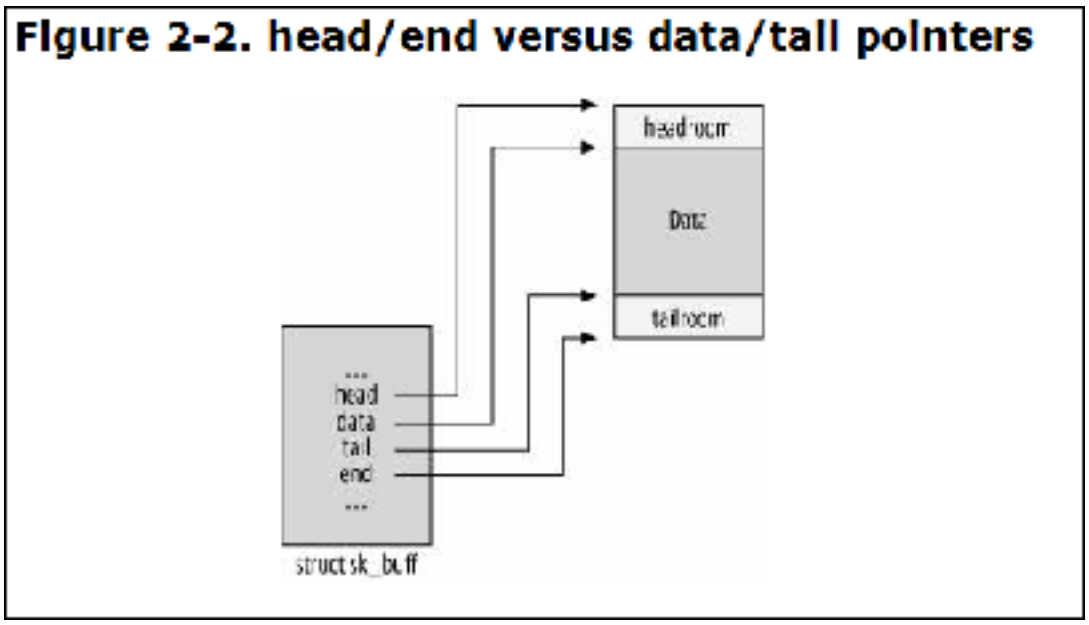


Figure 2-2. head/end versus data/tail pointers

8. void (*destructor)(...):当buffer被释放时将调用该函数。如果buffer不属于一个socket，该函数指针不被初始化，否则，它通常被初始化为sock_rfree或sock_wfree。这两个sock_XXX程序被用来更新socket所持有的内存总量。

2.1.3. General Fields

这一小节覆盖了sk_buff中与特定内核无关的成员：

1. struct timeval stamp:它通常只对接收到的数据包有意义，代表了包接收到时的时间戳，或偶尔代表包要被发送时的时间戳。
2. struct net_device *dev:它代表了一个网络设备。当数据包被接收时，设备驱动更新这个成员，来代表接收数据包的设备。当数据包被发送时，该成员代表了包被发送所使用的设备。
   一些网络功能可以把一些设备组织称一个单一的虚拟设备，通过虚拟设备驱动管理。当虚拟设备驱动被调用，dev指向了虚拟设备的net_device数据结构。驱动会从这组设备中选取一个特定的设备，然后把dev成员指向该设备的net_device数据结构。
3. struct net_device *input_dev:指向收到数据包的设备。如果数据包是本地产生的，它为NULL。这个成员主要被流量控制代码使用。
4. struct net_device *real_dev:只对虚拟设备有效，代表了与虚拟设备关联的真实设备。
5. union{...} h, union{...} nh, union{...} mac:它们都是指向TCP/IP协议栈中协议的指针：h指向L4，nh指向L3，mac指向L2。

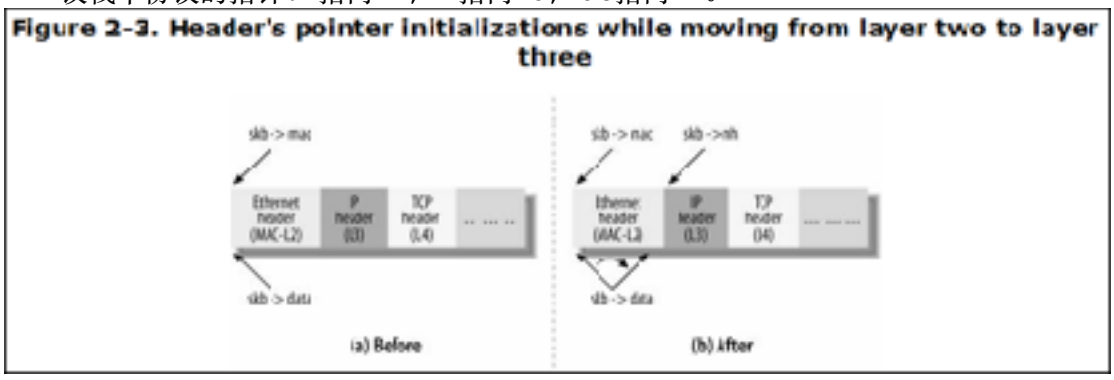


Figure 2-3. Header's pointer initializations while moving from layer two to layer three

6. struct dst_entry dst:用于路由子系统。
7. char cb[40]:这是"control buffer"或保存了每层的私有信息。

8. unsigned int csum, unsigned char ip_summed:校验和以及关联的状态flag。
9. unsigned char cloned:如果被设置，说明这个sk_buff结构是另一个sk_buff的克隆（它们使用同一个buffer）。
10. unsigned char pkt_type:这个成员用来把frame分类。可能的取值保存在include/linux/if_packet.h中。
11. __u32 priority:指示要被发送或转发的数据包的QoS等级。
12. unsigned short protocol:从L2判断出的协议类型，只要有:IP,IPv6和ARP。完整的列表在include/linux/if_ether.h中。通过该成员，来选定上层不同的处理函数。
13. unsigned short security:包的安全等级。原来是用在IPSec子系统中,现在已经作废了。

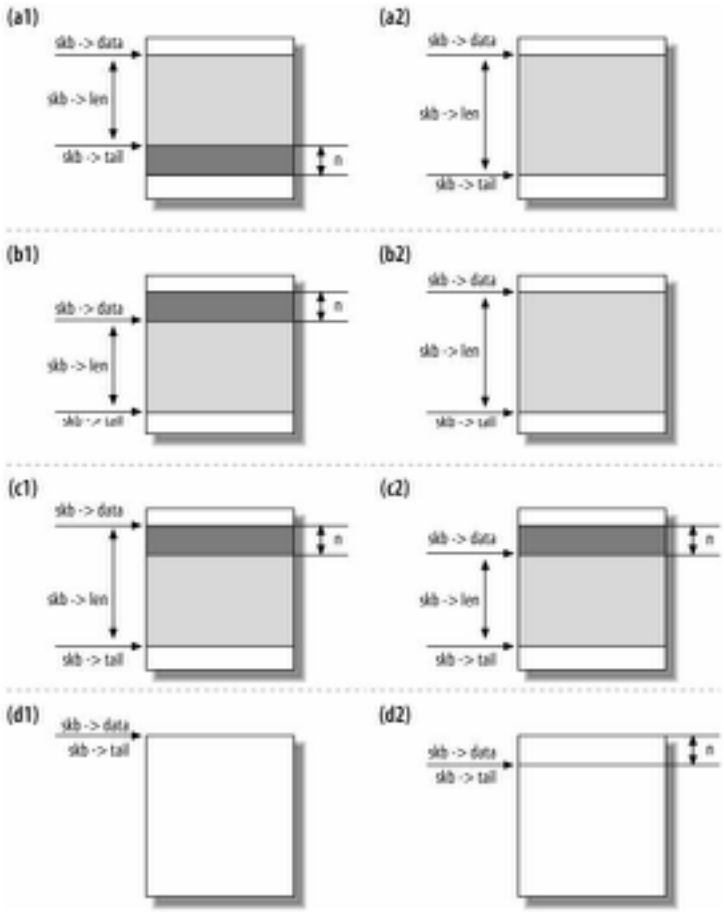
2.1.4. Feature-Specific Fields

Linux内核是模块化的，你可以选择使用或不使用某些特性。以下成员就是和特定内核特性相关的(例如Netfilter或QoS)：

1. unsigned long nfmark
   __u32 nfcache
   __u32 nfctinfo
   struct nf_conntrack *nfct
   unsigned int nfdebug
   struct nf_bridge_info *nf_bridge:这些参数用在Netfilter中。
2. union{...} private:用于High Performance Parallel Interface(HIPPI)
3. __u32 tc_index
   __u32 tc_verd
   __u32 tc_classid: 用于流量控制。
4. struct sec_path *sp:用于Ipsec协议。

2.1.5. Management Functions

Figure 2-4. Before and after: (a)skb_put, (b)skb_push, (c)skb_pull, and (d) skb_reserve



2.1.5.1. Allocating memory: alloc_skb and dev_alloc_skb

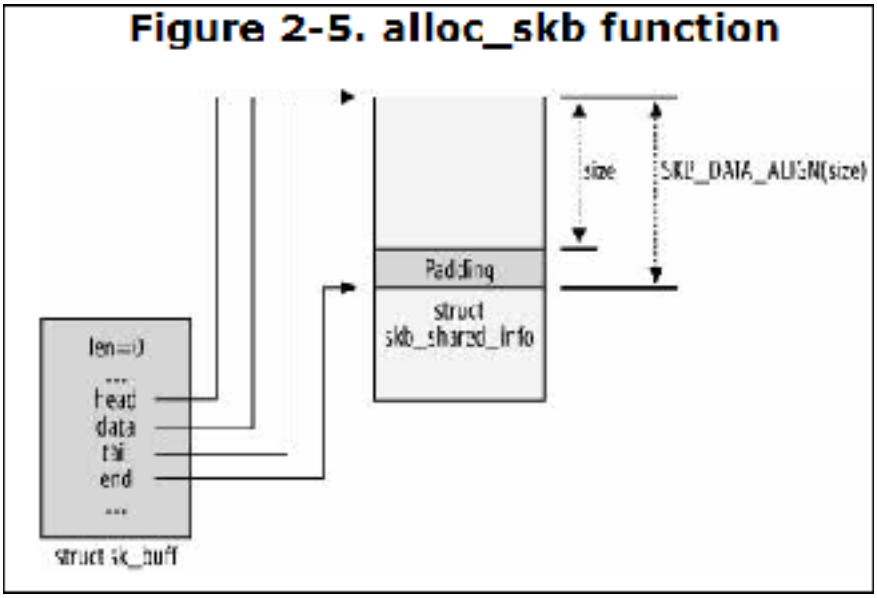
alloc_skb利用slab子系统分配一个sk_buff数据结构，然后调用kmalloc分配一个数据buffer。

```
    skb = kmem_cache_alloc(skbuff_head_cache, gfp_mask &
~__GFP_DMA);
    ... ... ...
    size = SKB_DATA_ALIGN(size);
         data = kmalloc(size + sizeof(struct skb_shared_info),
gfp_mask);
```

在调用kmalloc之前，size参数被宏SKB_DATA_ALIGN更新，使之对齐。

sk_buff中一些成员如下图初始化：

其中skb_shared_info块主要用来处理IP分片。



dev_alloc_skb是给驱动程序使用的，在中断上下文中调用的buffer分配函数。它是alloc_skb函数的简单包装，并多分配了16字节（为了优化目的），但指定了它是原子操作(GFP_ATOMIC)。

2.1.5.2. Freeing memory: kfree_skb and dev_kfree_skb

kfree_skb和dev_kfree_skb用于释放sk_buff。不过只有当sk_buff使用计数skb->users为1时才会被真正释放，否则只是简单的递减使用计数。

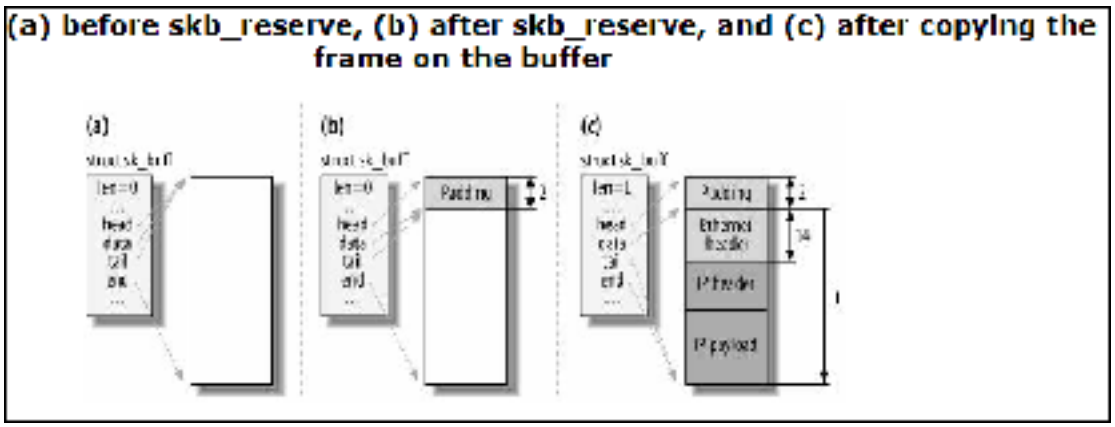
如果sk_buff真正要被释放了，并且sk_buff->destructor函数指针被初始化了，这个函数会被调用。

当kfree_skb真正释放sk_buff时，sk_buff指向的buffer底部的skb_shared_info数据结构指向的数据块也会被释放。sk_buff数据结构释放后会被回收到skbbuff_head_cache中。

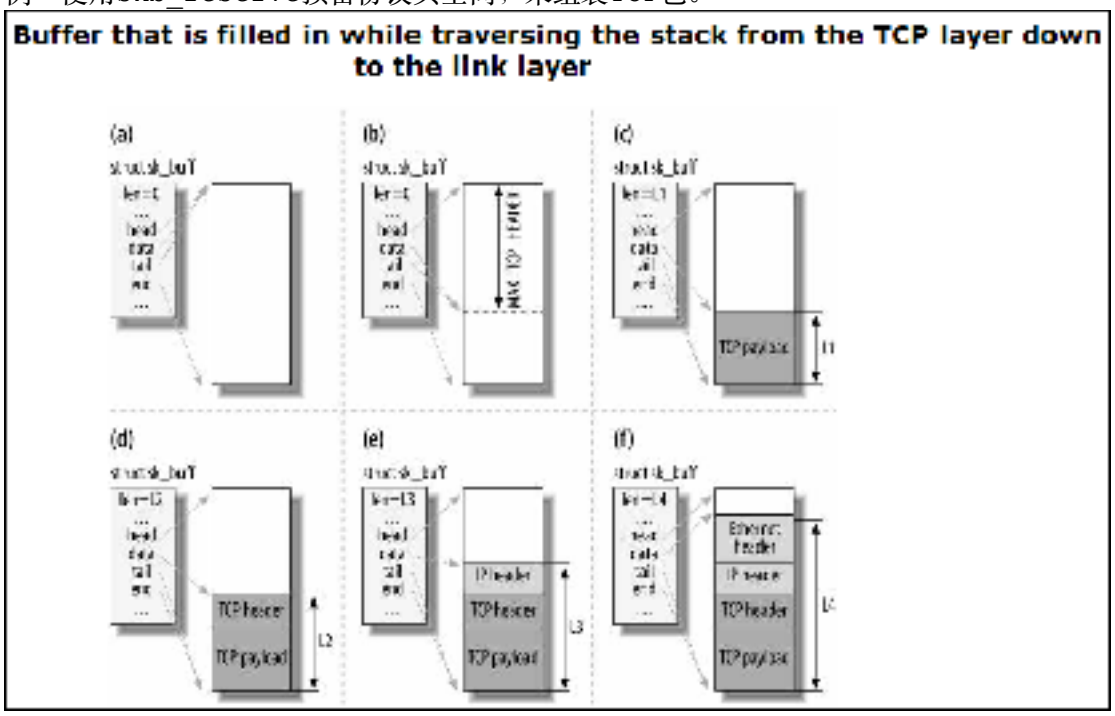
2.1.5.3. Data reservation and alignment: skb_reserve, skb_put, skb_push, and skb_pull

skb_reserve在buffer头部会预留出一些空间，以备之后插入协议头或者强迫数据对齐。

例：使用skb_reserve使IP头16字节对齐。



例：使用skb_reserve预留协议头空间，来组装TCP包。



2.1.5.4. The skb_shared_info structure and the skb_shinfo function

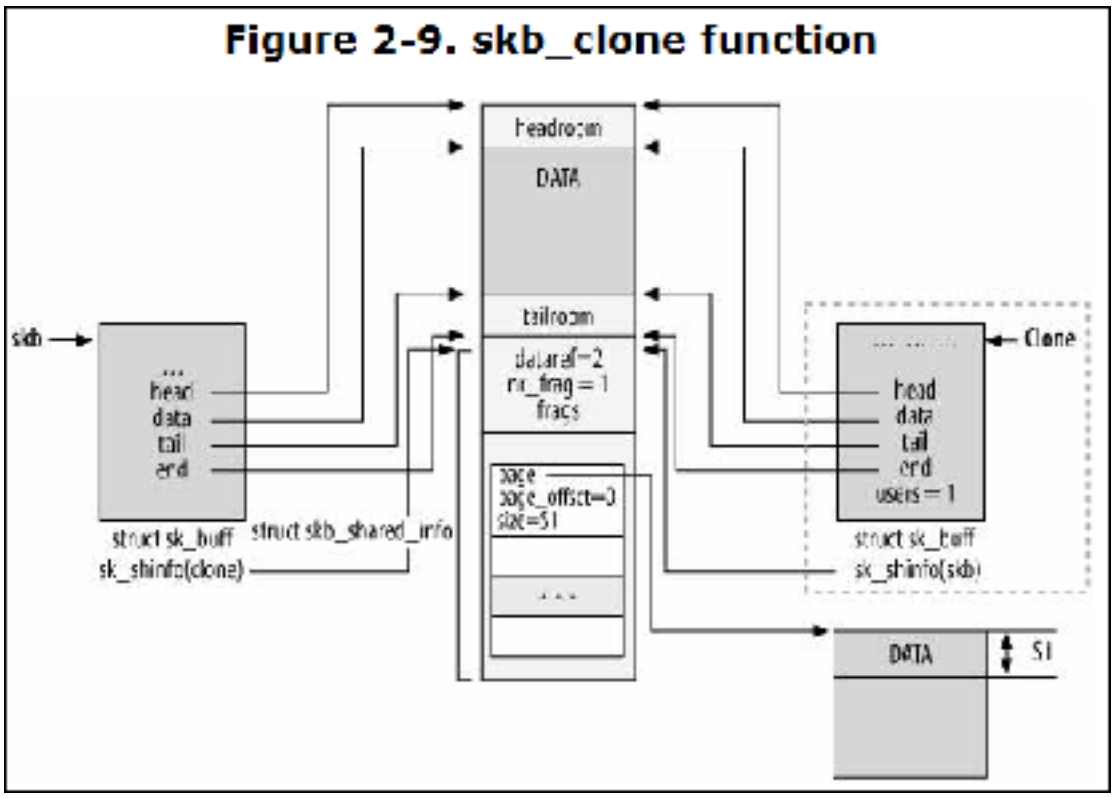
skb_shared_info位于数据buffer的末端，紧跟在end指针的后面。

```
struct skb_shared_info {
    atomic_t        dataref;
    unsigned int    nr_frags;
    unsigned short  tso_size;
    unsigned short  tso_seqs;
    struct sk_buff  *frag_list;
    skb_frag_t      frags[MAX_SKB_FRAGS];
};
```

dataref代表了数据buffer的用户数量。nr_frags,frag_list,和frags用来处理IP分片。tso_size和tso_seqs用于TCP segmentation offload。

2.1.5.5. Cloning and copying buffers

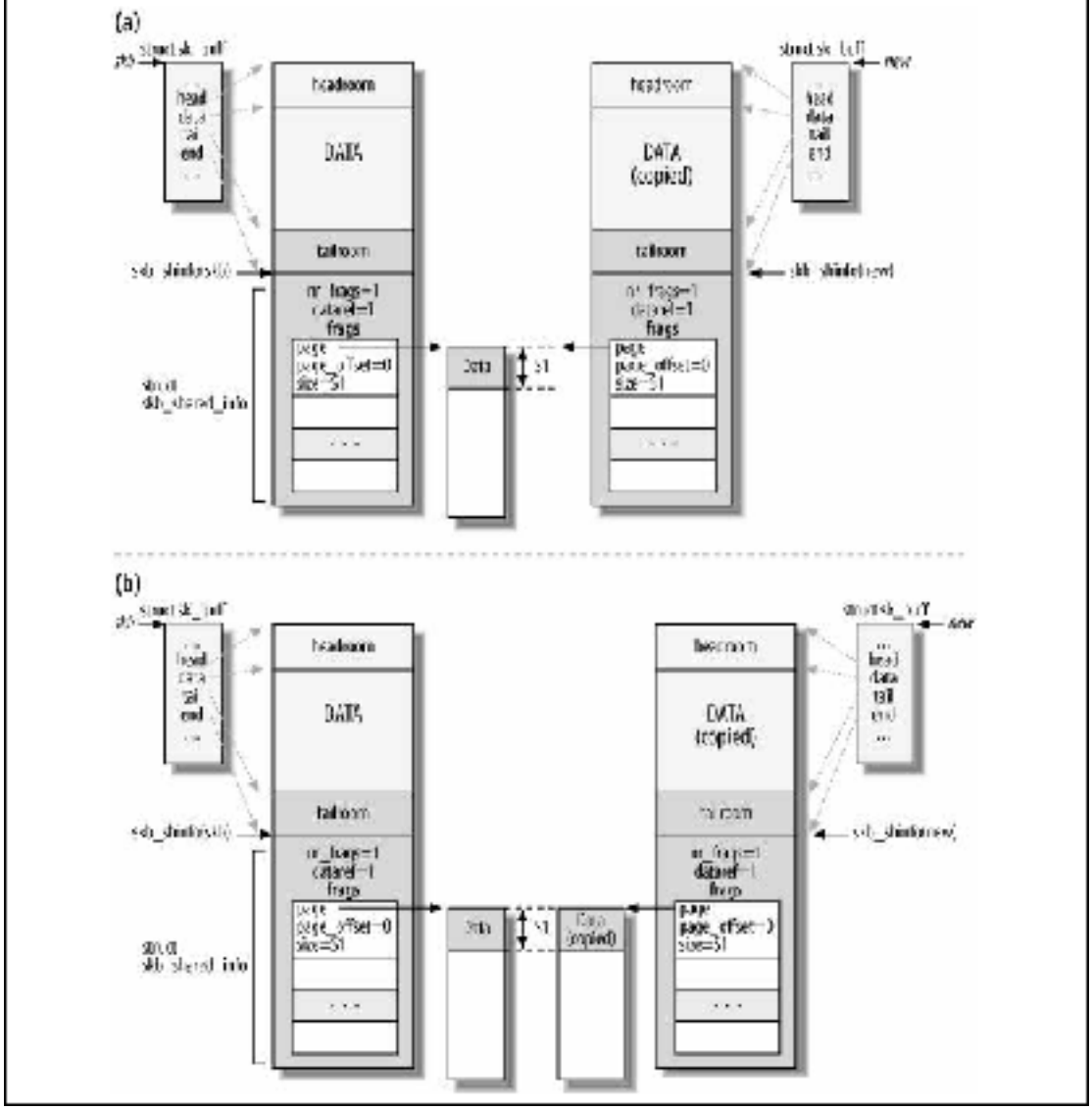
当一个buffer需要独立的被不同用户使用时，可以使用skb_clone函数，它只是克隆出了一个新的sk_buff，而没有对buffer进行复制。



被克隆出来的sk_buff没有链入任何链表，并且没有对socket的引用。原始的和被克隆的sk_buff的skb->cloned被设为1。

如果一个buffer被克隆了，那它就不能被修改。利用COW的思想。当需要修改的数据在skb->head和skb->end之间时，使用pskb_copy拷贝。如果要修改的是分片数据，必须使用skb_copy。

Figure 2-10. (a) pskb_copy function and (b) skb_copy function



2.1.5.6. List management functions

这些函数管理sk_buff的链表。在<include/linux/skbuff.h>和<net/core/skbuff.c>中有函数完整列表。以下是一些经常使用的函数：

1. skb_queue_head_init：初始化sk_buff_head结构，创建一个空队列。
2. skb_queue_head，skb_queue_tail：把一个缓冲区加入队列的头部或尾部。
3. skb_dequeue，skb_dequeue_tail:从头部或尾部取出一个元素。
4. skb_queue_purge：清空链表。
5. skb_queue_walk：循环遍历链表。

2.2. net_device Structure

所有设备的net_device结构都被放入了全局链表dev_base中。net_device结构中的成员被分为下列几类：

- Configuration
- Statistics
- Device status
- List management
- Traffic management

- Feature specific
- Generic
- Function pointers (or VFT)

2.2.1. Identifiers

net_device结构包含了3种标识符:

1. int ifindex:当设备注册时分配给每个设备的唯一的ID。
2. int iflink:主要用于隧道设备。
3. unsigned short dev_id:现在用在IPv6中。

2.2.2. Configuration

1. char name[IFNAMSIZ]:设备名字（例如:eth0）。
2. unsigned long mem_start,unsigned long mem_end:这两个成员描述了设备与内核通信的共享内存。它们仅被设备驱动访问；高层不需要关注它们。
3. unsigned long base_addr:设备自身的内存进行I/O映射后的起始地址。
4. unsigned int irq:设备使用的中断号。
5. unsigned char if_port:设备使用的port类型。
6. unsigned char dma:设备使用的DMA通道。
7. unsigned short flags:flags成员中一些bits代表了网络设备支持的功能（如: IFF_MULTICAST）,其它代表了状态（例如: IFF_UP或IFF_RUNNING）。
8. unsigned short gflags:主要用于兼容,现在几乎不使用了。
9. unsigned short priv_flags:被VLAN和Bridge虚拟设备使用。
10. int features:另一个flags位图,它保存了设备支持的其它一些功能。features向CPU报告了网卡所支持的功能，例如: 高内存DMA,硬件进行校验和。
11. unsigned mtu:设备能处理的最大frame。
12. unsigned short types:设备分类(例如:以太网，帧中继等等)。
13. unsigned short hard_header_len:以字节为单位的帧头部大小，例如: 以太网帧头是14字节。
14. unsigned char broadcast[MAX_ADDR_LEN]:链路层广播地址。
15. unsigned char dev_addr[MAX_ADDR_LEN]:设备的链路层地址。
16. unsigned char addr_len:设备的链路层地址长度。
17. promiscuity:混杂模式。

2.2.2.1. Interface types and ports

```
switch (dev->if_port) {
case IF_PORT_10BASE2:
    writeb((readb(addr) & 0xf8) | 1, addr);
    break;
case IF_PORT_10BASET:
    writeb((readb(addr) & 0xf8), addr);
    break;
}
```

2.2.2.2. Promiscuous mode

net_device->promiscuity指明一个设备是否处于混杂模式。它是一个计数器，因为可能多个用户会把设备配置成混杂模式。当这个值为非0时，设备处于混杂模式。

2.2.3. Statistics

net_device结构中并没有提供专门用来描述统计信息的成员，而是使用一个指针priv指向驱动的私有数据，来储存设备的信息。

在第8章可以看到，私有数据结构有时会分配在net_device结构后面。

2.2.4. Device Status

为了与NIC交互，每个设备驱动要维护一些信息，例如：时间戳，flags。在SMP系统中，内核必须确保不同CPU对同一个设备的并发访问被正确处理。以下一些net_device成员被用来保存这些信息：

1. unsigned long state:网络队列子系统使用的flags集合。
2. enum {...} reg_state:设备的注册状态。
3. unsigned long trans_start:最后一个帧开始发送的时间(以jiffies为单位)。这个成员被网卡用来检测错误的发生。
4. unsigned long last_rx:最后一个数据包接收时的时间(以jiffies为单位)。
5. struct net_device *master:一些协议允许一组设备被组织在一起看为一个单一的设备。这些协议包括：EQL，Bonding和TEQL(流量控制队列)。这组设备中的某一个被挑选出来称为master。这个成员就是指向master的指针。如果设备不是这样一个设备组的成员，该指针被设为NULL。
6. spinlock_t xmit_lock，int xmit_lock_owner：xmit_lock_owner是拥有xmit_lock锁的CPU的ID号。单CPU中它总是0，如果锁没有被CPU获取，该值为-1。
7. void *atalk_ptr
   void *ip_ptr
   void *dn_ptr
   void *ip6_ptr
   void *ec_ptr
   void *ax25_ptr：这6个成员指向给特定协议使用的私有数据。

2.2.5. List Management

net_device数据结构被插入了一个全局链表和2个哈希表中。下列成员被用来完成这些任务：

1. struct net_device *next:把每个net_device链入全局链表中。
2. struct hlist_node name_hlist
   struct hlist_node index_hlist：把net_device链入哈希表。

2.2.6. Link Layer Multicast

当一个设备被加入了很多多播组时，那么，设备简单的接收所有的多播帧将会比维持一张多播地址表再根据它来过滤接收到的帧要高效的多。net_device->flags中有一位指示了设备是否接收所有多播地址。这个标志位的设置和清除是受net_device->all_multi成员来控制的。

每个设备维护了dev_mc_list数据结构，链路层多播地址通过dev_mc_add和dev_mc_delete来添加和删除多播地址。net_device中相关成员有：

1. struct dev_mc_list *mc_list:指向这个设备的dev_mc_list结构队列头。
2. int mc_count:该设备的多播地址数量，也是mc_list指向的链表长度。
3. int allmulti:如果非0，设备监听所有多播地址。类似于promiscuity成员，allmulti也是一个引用计数而不是一个简单的布尔量。

2.2.7. Traffic Management

2.2.8. Feature Specific

2.2.9. Generic

1. atomic_t refcnt:引用计数。设备只有当计数器为0时才能被卸载。

2. int watchdog_timeo
   struct timer_list watchdog_timer: 第11章介绍。
3. int (*poll)(...)
   struct list_head poll_list
   int quota
   int weight:用在NAPI中，第10章将介绍。
4. const struct iw_handler_def *wireless_handlers
   struct iw_public_data *wireless_data:用于无线设备的额外参数。
5. struct list_head todo_list:用于卸载网络设备。
6. struct class_device class_dev:被新一代内核驱动结构所使用。

2.2.10. Function Pointers

Chapter 4. Notification Chains

4.1. Reasons for Notification Chains

4.2. Overview

一个notification chain是一个函数链表，当一个给定事件发生时，函数将被执行。每个函数让其它子系统知道某个事件发生了。
notifier只是简单的定义一个链表，然后每个notified子系统把回调函数注册到链表中。

4.3. Defining a Chain

notification chain中元素的类型是:

```
struct notifier_block
{
       int  (*notifier_call)(struct  notifier_block  *self,  unsigned
long, void *);
    struct notifier_block *next;
    int priority;
};
```

notifier_call是回调函数，next指向下一个链表元素，priority代表了回调函数的执行优先级，但实际中链表已是按优先级排好序了的。

4.4. Registering with a Chain

通过notifier_chain_register或notifier_chain_unregister注册或卸载。链表中每个notifier_block是按优先级排序的，相同优先级的notifier_block，新插入的在末尾。

4.5. Notifying Events on a Chain

使用notifier_call_chain函数来产生notifications，这个函数在kernel/sys.c中。这个函数只是顺序的调用链表中的回调函数。

```
int notifier_call_chain(struct notifier_block **n, unsigned long
val, void *v)
{
    int ret = NOTIFY_DONE;
    struct notifier_block *nb = *n;

    while (nb)
    {
        ret = nb->notifier_call(nb, val, v);
        if (ret & NOTIFY_STOP_MASK)
```

```
        {
            return ret;
        }
        nb = nb->next;
    }
    return ret;
}
```

三个参数：

1. n: notification chain。
2. val:事件类型（例如：NETDEV_REGISTER）。
3. v:给链表中回调函数使用的参数。这个参数在不同环境中有不同的含义。例如：当一个新网络设备注册进内核，v代表的是net_device数据结构。

回调函数返回值（定义在include/linux/notifier.h）：

1. NOTIFY_OK:notification成功完成。
2. NOTIFY_DONE:
3. NOTIFY_BAD:执行回调函数时出现了错误。停止继续调用回调函数。
4. NOTIFY_STOP:程序成功执行。剩下的回调函数不需要再执行了。
5. NOTIFY_STOP_MASK:被notifier_call_chain用来判断是停止执行下一个回调函数还是接着调用。

notifier_call_chain函数可能会在不同CPU上调用，作用在同一个notification chain上。回调函数的担负起互斥和串行换执行的责任。

4.6. Notification Chains for the Networking Subsystems

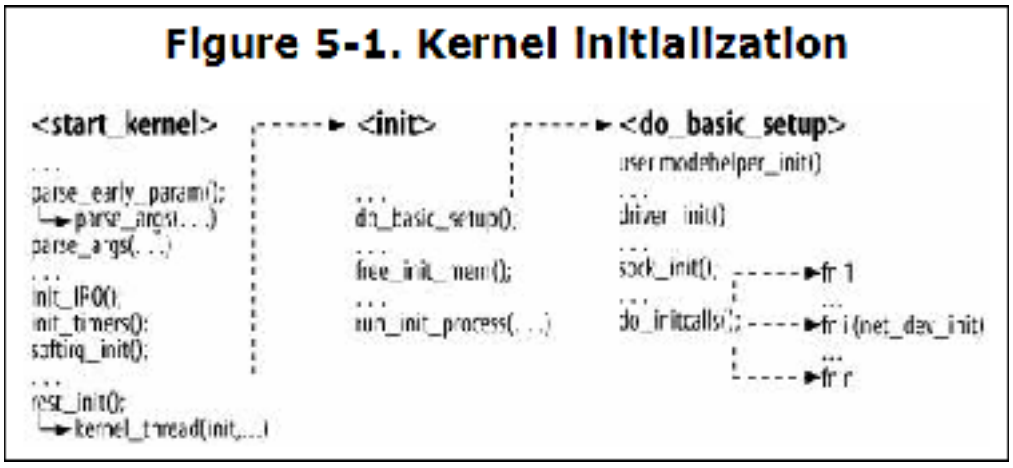
内核中定义了至少10种不同的notification chains。我们关注的是网络模块中使用的那几种，主要有：

1. inetaddr_chain:当在网络接口上插入，移除，改变IPv4地址时会发送notifications。
2. netdev_chain:注册网络设备时发送notifications。

Chapter 5. Network Device Initialization

5.1. System Initialization Overview

如下图，当内核启动时，它执行start_kernel，初始化了一组子系统。在start_kernel结束时，它调用init内核线程，这个线程执行了剩下的初始化工作。本章中涉及的绝大多数初始化活动在do_basic_setup中完成。



在不同初始化任务中，我们主要关注下面三种：

1. Boot-time options:处理通过boot loader(如LILO或GRUB)在启动时传递给内核的配置参数。
2. Interrupts and timers:通过init_IRQ和softirq_init分别初始化硬中断和软中断。
3. Initialization routines:内核子系统和内建的设备驱动通过do_initcalls初始化。

run_init_process决定了系统中第一个执行的进程，它将是所有其它进程的父亲;它的PID号是1，并且永远不会被挂起。

```
static void run_init_process(char *init_filename)
{
   argv_init[0] = init_filename;
   kernel_execve(init_filename, argv_init, envp_init);
 }
```

5.2. Device Registration and Initialization

一个网络设备要想被使用，它必须被内核识别以及连接到正确的驱动程序。驱动程序的私有数据结构中保存了驱动该设备的所有信息，以及与需要使用该设备的内核组件通信的所有信息。注册和初始化任务部分时由内核完成，部分是由设备驱动完成。下面是初始化的几个方面：

1. Hardware initialization:这个步骤是由设备驱动和generic bus layer(例如PCI和USB)共同完成的。设备驱动可以使用默认参数，也可以使用用户提供的参数来配置设备的特性（如：IRQ和I/O地址）。
2. Software initialization:在设备可以使用前，根据使用的网络协议和配置，用户需要提供配置参数，如：IP地址。
3. Feature initialization:Linux内核有很多网络可选特性。例如：流量控制，QoS。

5.3. Basic Goals of NIC Initialization

每个网络设备在内核中是用net_device数据结构。这个数据结构的初始化部分由设备驱动完成部分由内核完成。本章中，我们关注在用于设备/内核通信的资源，例如：

1. IRQ line:NIC需要分配一个IRQ来与内核通信。虚拟设备不需要分配IRQ:loopback设备就是个例子，它的活动在内部完成，所以不需要分配IRQ。
2. I/O ports and memory registration:驱动把设备的内存（例如控制寄存器）映射到系统内存中。这样驱动可以直接通过系统内存地址来访问。I/O ports and memory的注册和释放函数分别为：request_region和release_region。

5.4. Interaction Between Devices and Kernel

几乎所有的设备(包括NIC)与内核通信是通过2种方式：Polling，Interrupt。

5.4.1. Hardware Interrupts

当一个设备驱动注册NIC时，它需要被分配一个IRQ。通过request_irq()和free_irq()函数分别用来在一个给定IRQ上注册和释放中断处理函数，它们都是依赖于体系结构的函数：

1. int request_irq(unsigned int irq, void (*handler)(int, void*, struct pt_regs*), unsigned long irqflags, const char * devname, void *dev_id)

函数注册一个中断处理函数，首先需要确保中断号是合法并且没有分配给其它设备，除非是共享IRQ。

2. void free_irq(unsigned_int irq, void *dev_id)

给定设备标识符dev_id，这个函数移除了中断处理函数，如果该中断线上没有注册其它设备，它还会关闭该IRQ线。

5.4.1.1. Interrupt types

通过中断，NIC设备可以告诉驱动如下事件：

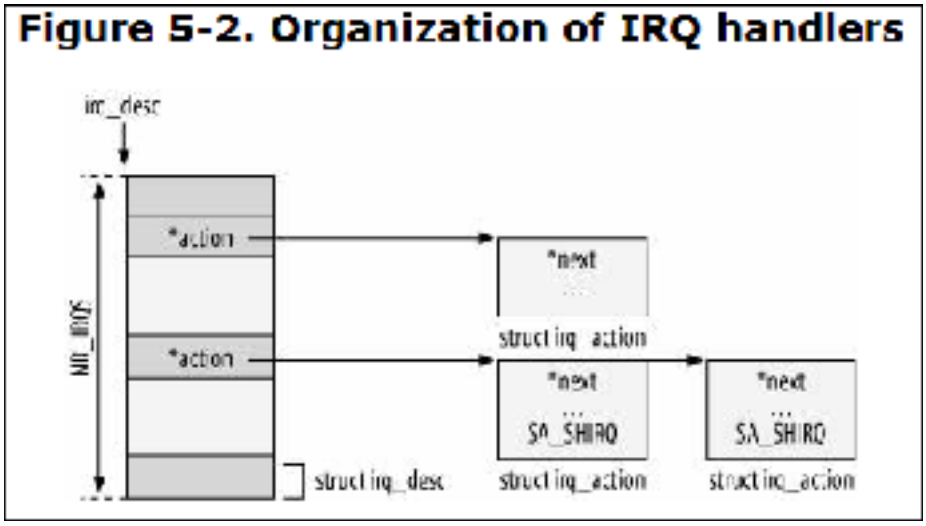
1. Reception of a frame:这是最常用和标准的情况。
2. Transmission failure: This kind of notification is generated on Ethernet devices only after a feature called exponential binary backoff has failed (this feature is implemented at the hardware level by the NIC). Note that the driver will not relay this notification to higher network layers; they will come to know about the failure by other means (timer timeouts, negative ACKs, etc.).
3. DMA transfer has complete successfully
4. Device has enough memory to handle a new transmission

5.4.1.2. Interrupt sharing

对于一组共享IRQ线的设备，它们都必须设置为允许共享IRQ。

5.4.1.3. Organization of IRQs to handler mappings

Figure 5-2. Organization of IRQ handlers



5.5. Initialization Options

5.6. Module Options

5.7. Initializing the Device Handling Layer: net_dev_init

网络组件的初始化，在启动阶段主要由net_dev_init执行。这个函数定义在net/core/dev.c

net_dev_init的主要部分有：

- 被2种网络软中断使用的per-CPU数据结构初始化。
- 当内核被编译为支持/proc文件系统，一些文件通过dev_proc_init和dev_mcast_init函数被添加入/proc。
- netdev_sysfs_init在sysfs中注册net class。创建在/sys/class/net下。
- net_random_init初始化一个per-CPU种子向量，它被net_random用来产生随机

数。

- The protocol-independent destination cache (DST), described in Chapter 33, is initialized with dst_init.
- 协议处理函数所在的向量ptype_base，用来多路分解入口流量。
- 当OFFLINE_SAMPLE符号有定义，内核设置了一个函数来有规律的收集设备队列长度的统计值。在这种情况下，net_dev_init需要创建一个定时器，来周期性的运行该函数。
- 一个回调函数注册进notification chain，这样，当CPU热拔插事件发生时能产生notification。回调函数是dev_cpu_callback。目前，唯一的事件是挂起CPU。当notification被接收，该CPU的ingress queue中的buffer出队，并被传递给netif_rx。

5.8. User-Space Helpers

内核在一些情况下会调用用户空间应用程序来处理事件。下面程序两个特别重要:

1. /sbin/modprobe: 当内核需要加载模块时调用。这个程序是the module-init-tools package的一部分。
2. /sbin/hotplug: 当内核检测到一个新设备从系统中被plug或unplug时调用。它的主要工作是根据设备标识符来加载正确的设备驱动。设备的标识符是根据它plug进的总线（例如: PCI）确定，相应的ID号是根据总线标准来定义的。这个程序是hotplug package的一部分。

内核提供了一个名为call_usermodehelper的函数来执行这些用户空间程序。函数允许调用者通过arg[]和env[]传递给应用程序参数。例如: arg[0]告诉call_usermodehelper哪个用户程序要被执行，arg[1]用来告诉用户程序使用哪个配置脚本。

下图展示了2个内核程序: request_module和kobject_hotplug，调用call_usermodehelper来分别调用/sbin/modprobe和/sbin/hotplug。

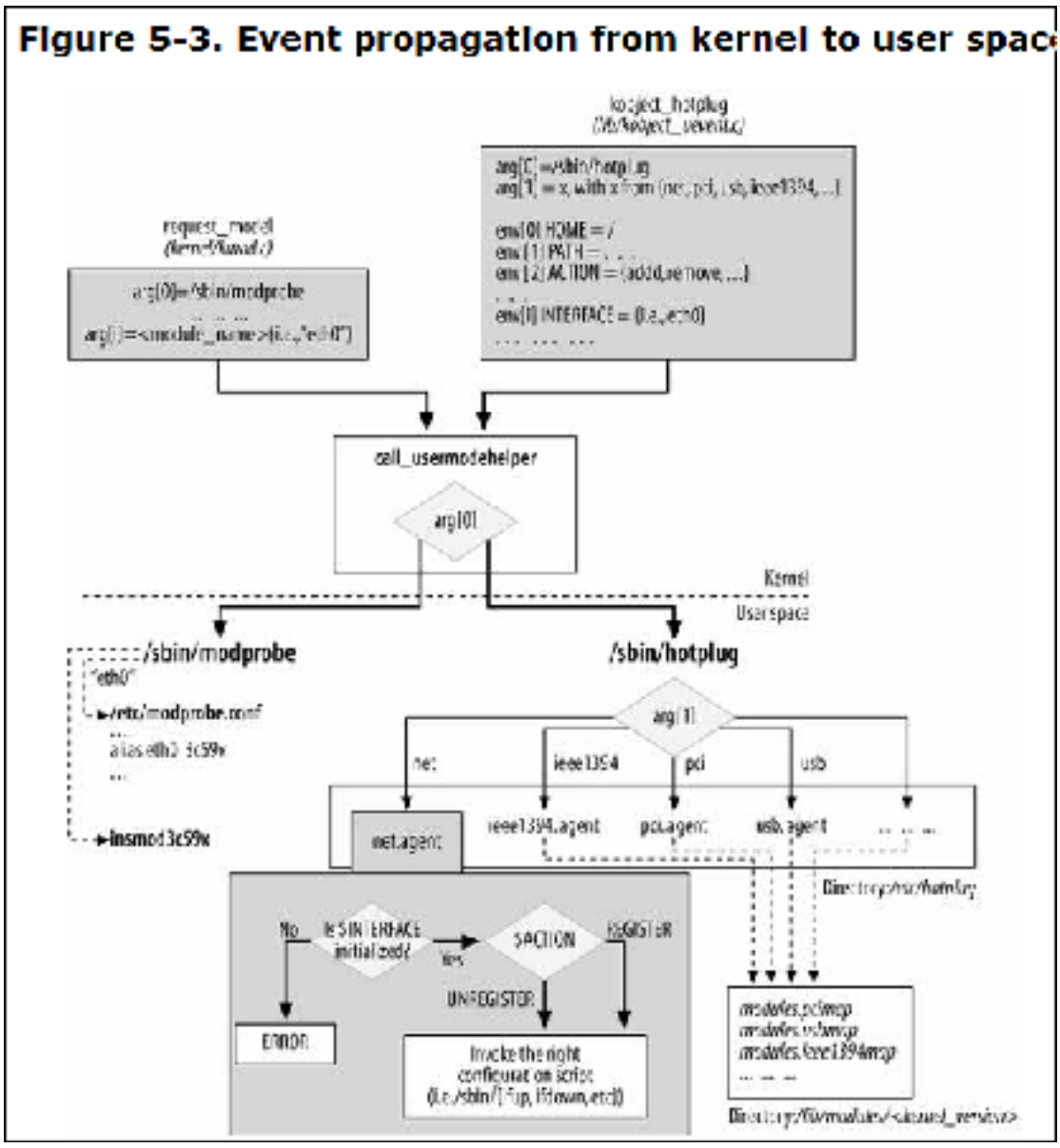


Figure 5-3. Event propagation from kernel to user space

5.8.1. kmod

kmod是内核模块加载器，它允许内核组件请求加载一个模块。

（lsmod、insmod、rmmod是一组实用工具所提供的三个命令，这组实用工具一般是和内核版本对应的，其1.3.57版本名为modules（modules-1.3.57.tar.gz），高一点的版本名为modutils（例如modutils-2.4.2.tar.gz）。最好保证你的系统中的模块实用工具的版本号（可以使用modinfo -V命令来查看）不低于内核版本号（可以使用uname -r来查看）。1.3.57版本的modules内容包括modprobe、depmod、genksyms、makecrc32、insmod、rmmod、lsmod、ksyms、kerneld等命令。其中modprobe和insmod命令类似，不过它要依赖于相关的配置文件；depmod用于生成模块依赖文件/lib/modules/kernel-version/modules.dep；genksyms和ksyms与内核函数的版本号有关（由于内核的不断更新，各个版本的内核函数各有不同，为了不会引起系统的崩溃，内核源程序中要对内核函数的版本号进行严格地控制）。在以后版本的实用工具中，使用kmod来取代了kerneld。kmod的功能和kerneld类似，但是它不能自动卸载模块。之所以采用kmod的原因在于kerneld是使用IPC通道实现的，相当于多经过了一层处理，另外kerneld的代码也比较复杂，kmod的代码数量也比kerneld少得多。）

内核提供多个程序，但我们主要关注request_module。这个函数用需要加载的模块名初始化arg[1]。/sbin/modprobe使用配置文件/etc/modprobe.conf来做不同的事，其中之一是判断从内核中接收到的模块名是否是其它模块的一个别名。

5.8.2. Hotplug

热拔插被引入Linux内核是为了实现即插即用(PnP)这个特性。这个特性允许内核检测到一个可热拔插设备的插入或移除，并能通知用户空间的应用程序，使得以后的部件可以装载它所需要的驱动，并使用相关的配置。

热拔插实际上也被用来在启动时维护不可热拔插的设备。它并不关心一个设备是在系统运行的时候插上的或者是系统引导的时候就已经插上了，两种情况下用户空间程序都会得到通报。用户空间的应用程序决定是否要对事件作出响应。

当编译内核模块时，目标文件缺省的是放在目录/lib/modules/kernel_version/，kernel_version代表了不同内核版本名字。在同一个目录下，能找到2个有趣的文件：modules.pcimap和modules.usbmap。这些文件分别包含了内核支持的设备的PCI IDs和USB Ids。文件同时也包含了每个设备ID对相应内核模块的引用。当用户空间程序接收到一个热拔插事件的notification时，它将使用这些文件来找到正确的设备驱动。

5.8.2.1. /sbin/hotplug

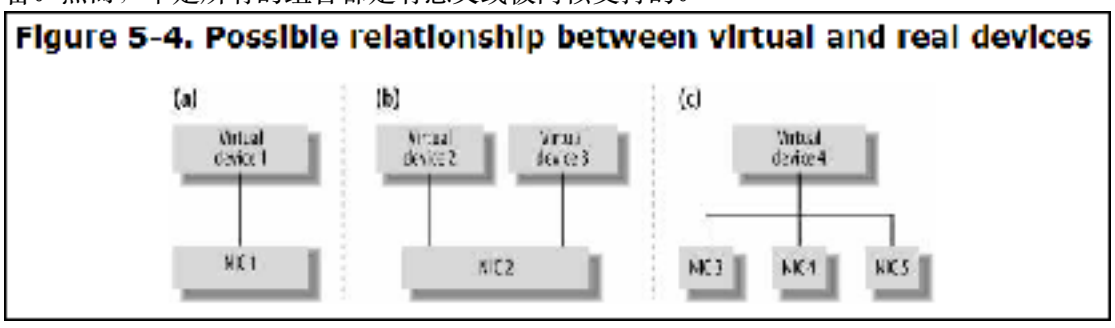
缺省的对应于Hotplug的用户空间程序是脚本/sbin/hotplug，它是Hotplug package的一部分。这个包能用缺省目录下/etc/hotplug/和/etc/hotplug.d/的文件来进行配置。

kobject_hotpulf()函数被内核调用来响应设备的添加和移除操作。kobject_hotpulf()初始化了arg[0]为/sbin/hotplug和arg[1]为。。。不写了。2.6.16中居然没找到该脚本。

5.9. Virtual Devices

一个虚拟设备是构建在一个或多个真实设备上的抽象。可以在其它虚拟设备上构建虚拟设备。然而，不是所有的组合都是有意义或被内核支持的。

Figure 5-4. Possible relationship between virtual and real devices



5.9.1. Examples of Virtual Devices

Linux允许定义不同种类的虚拟设备。这是一些例子：

1. Bonding：使用这个特性，一个虚拟设备把一组物理设备绑定成一组，并使它们看起来像一个设备。
2. 802.1Q：这是一个IEEE标准， 它用一个称为VLAN的首部扩充了802.3/以太网首部，允许创建虚拟网。
3. Bridging：网桥的桥接接口是虚拟出来的。详细参见第四章。
4. Aliasing interface：最初，别名接口主要的目的是为了使得一个实际的以太网

接口可以对应多个虚拟的接口（eth0:0 eth0:1等），每个虚拟接口都有自己的IP配置文件。现在，由于网络代码的改进，已经不需要通过定义多个虚拟接口来在一个网络接口卡上配置多个IP地址了。但是，在某些情况下（特别是路由选择），在一个网络接口卡上定义虚拟接口将更简单，比如可以使用更简单的配置文件。参考第三十章。

5. True equalizer（TEQL）：这是一个用于流量控制的排队规则。它的实现需要创建一个特殊的设备。TEQL的实现的思路和捆绑类似。
6. Tunnel interfaces: IP-over-IP Tunnel的实现和广义路径选择封装协议（GRE）的实现也依赖于虚拟设备。

5.9.2. Interaction with the Kernel Network Stack

内核中虚拟设备和真实设备的接口稍微有些不同。例如，它们在下列地方不同：

1. Initialization：绝大多数虚拟设备也像真实设备一样，被分配了一个net_device数据结构。通常，绝大多数虚拟设备net_device结构中的函数指针被初始化为真实设备对应函数的包装。
   然而，不是所有虚拟设备都被分配一个net_device结构。别名设备是个例子；它们被实现为对应真实设备的简单标签。
2. Configuration：通常都提供特别的用户空间工具用于虚拟设备的配置，特别是对那些只和这些设备相关且不能用ifconfig这类工具配置的高层的字段。
3. External interface：每个虚拟设备通常导出一个文件或一个包含一些文件的目录到/proc文件系统。这些文件的复杂和详细程度是依赖于具体的虚拟设备的类型和设计。
4. Transmission：如果虚拟设备与真实设备的关系不是一对一，用于传输数据包的程序需要包含对真实设备的选择。因为QoS是以每个设备为基础的，因此虚拟设备和实际设备之间的关系还和流量控制配置有关。
5. Reception：因为虚拟设备是软件对象，它们不需要和系统资源进行交互，如注册一个中断控制器、分配I/O端口或是I/O缓存。它们从进行这些操作的物理设备间接获得这些信息。对于不同类型的虚拟设备包的接收是不同的，802.1Q接口登记为一个以太类型，只有那些由相关实际设备接收的包含正确的协议ID的包才会传递给它。桥接接口则接收相关设备上的所有的数据（参考第十六章）。
6. External notifications：虚拟设备和真实设备都很关心内核组件产生的notifications。因为虚拟设备是实现在真实设备之上的，因此真实设备并不知道虚拟设备的逻辑实现，因此不能把notifications传递给虚拟设备。因为这个原因，notifications需要直接传给虚拟设备。
   和软件触发的notifications不同，硬件触发的notifications(例如：PCI电源管理)不能直接到达虚拟设备，因为没有与虚拟设备相关联的硬件。

5.11. Functions and Variables Featured in This Chapter

Name	Description
Functions and macros	
request_irq free_irq	Registers and releases, respectively, a callback handler for an IRQ line. The registration can be exclusive or shared.

Name	Description
request_region release_region	Allocates and releases I/O ports and I/O memory.
call_usermodehelper	Invokes a user-space helper application.
module_param	Macro used to define configuration parameters for modules.
net_dev_init	Initializes a piece of the networking code at boot time.
Global variables	
dev_boot_phase	Boolean flag used by legacy code to enforce the execution of net_dev_init before NIC device drivers register themselves.
irq_desc	Pointer to the vector of IRQ descriptors.
Data structure	
struct irq_action	Each IRQ line is defined by an instance of this structure. Among other fields, it includes a callback function.
net_device	Describes a network device.

Chapter 6. The PCI Layer and Network Interface Cards

6.1. Data Structures Featured in This Chapter

这里是一些被PCI层使用的关键数据结构。下面是我们必需掌握的。一个定义在include/linux/mod_devicetable.h，另一个定义在include/linux/pci.h。

1. pci_device_id：设备标识符。这不是Linux本地使用的的ID，而是根据PCI标准定义的ID。
2. pci_dev：每个PCI设备被分配了一个pci_dev数据结构，就像每个网络设备都被分配了一个net_device数据结构。pci_dev被内核用来引用一个PCI设备。
3. pci_driver：定义了PCI层与设备驱动之间的接口。这个结构基本上是由函数指针组成。所有的PCI设备使用它。

下面是pci_driver结构的主要成员的描述：

1. char *name：设备名字。
2. const struct pci_device_id *id_table：ID向量，把设备和它的驱动联系起来，供内核使用。

3. int (*probe)(struct pci_dev *dev, const struct pci_device_id *id)：当内核发现id_table与某个设备的ID号相匹配，这个函数就被PCI层调用。这个函数将开启硬件，分配net_device结构并且初始化和注册新设备。在这个函数中，驱动也可以分配任何其正常工作所用的额外的数据结构(例如，发送和接收过程中使用的缓冲区)。
4. void (*remove)(struct pci_dev *dev)：当驱动从内核中释放时，或一个拔插设备被移除时，该函数被调用。它是probe的相反操作的函数，会清除之前分配的数据结构和状态。
   网络设备使用这个函数释放分配的I/O端口和I/O内存，释放设备，释放net_device数据结构和其它被设备驱动在probe函数中分配的辅助数据结构。

6.2. Registering a PCI NIC Device Driver

PCI设备是唯一的使用参数组合来标识的设备，这些参数包括：vendor，model等等。这些参数在内核里使用数据结构pci_device_id储存：

```
struct pci_device_id {
     unsigned int vendor, device;
     unsigned int subvendor, subdevice;
     unsigned int class, class_mask;
     unsigned long driver_data;
};
```

这个结构中的很多字段是可以自解释的。vendor和device通常足够用来标识一个设备。subvendor和subdevice很少使用，通常设为统配符（PCI_ANY_ID）。class和class_mask代表设备所属的类；NETWORK是本章中我们所通常讨论的设备类。driver_data不是PCI_ID的一部分；它是驱动程序所使用的私有参数。

每个设备驱动在内核中注册了一个pci_device_id结构的向量，它表示了该设备驱动能够驱动的所有设备的ID号。

PCI设备驱动通过函数pci_register_driver()和pci_unregister_driver()在内核中注册和释放。

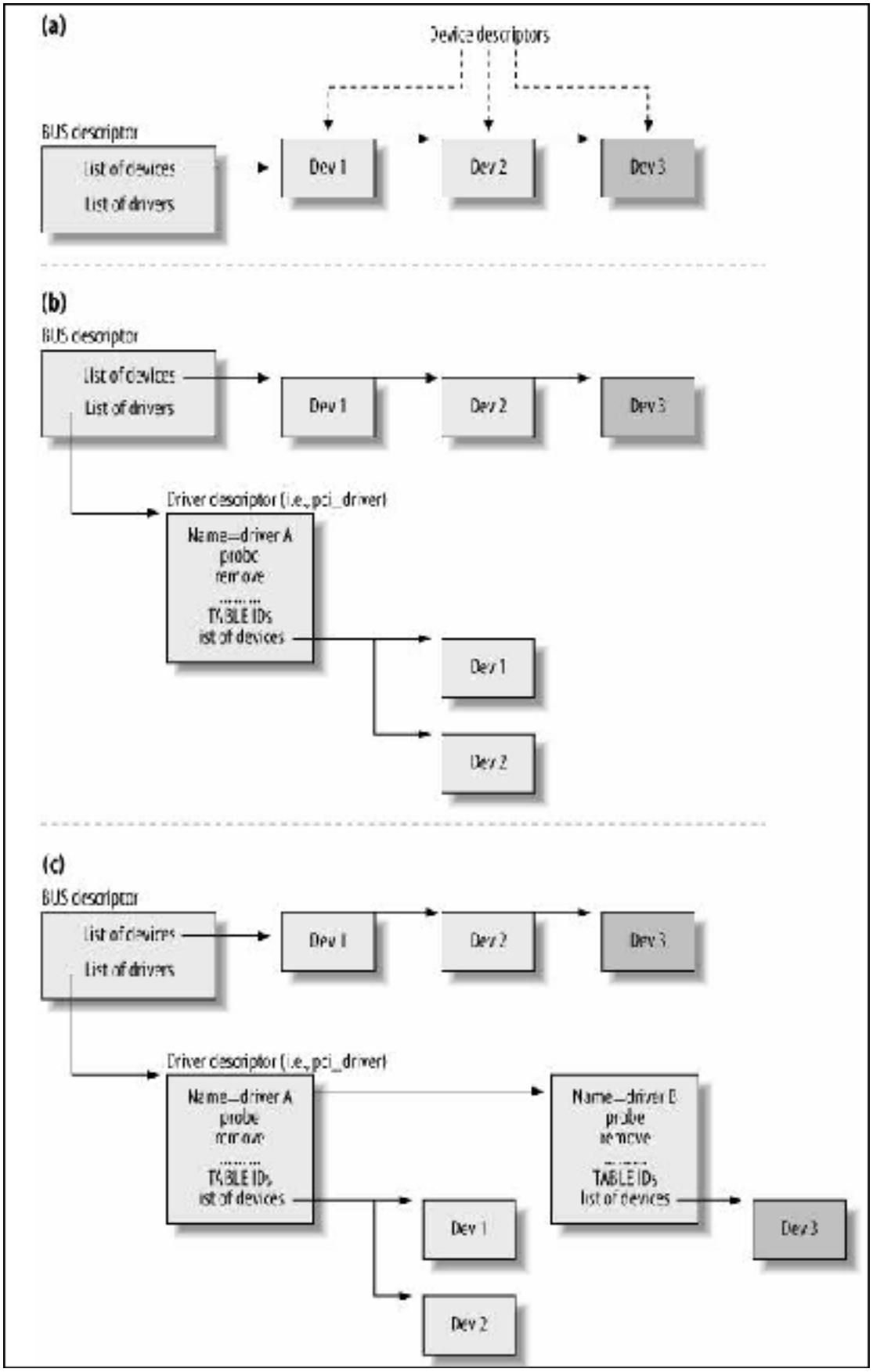
pci_register_driver()需要一个pci_driver数据结构作为参数。多亏了pci_driver的id_table向量，内核能知道哪些设备驱动可以处理，也多亏了pci_driver中的函数指针，使内核有一种机制把设备和它的驱动联系起来。

6.5. The Big Picture

当系统启动时，它创建了一个数据库，存储了启动时检测到的使用该总线的设备。在设备驱动加载之前，内核已经建立了它的数据库：以下图的三个PCI设备为例看看当设备驱动程序A和B加载的时候发生什么。

当设备驱动A被加载时，它通过调用pci_register_driver()注册进PCI层，并提供了该驱动的pci_driver结构。PCI层接着使用pci_driver->id_table来判断是否能匹配上之前检测到的PCI设备。接着创建驱动的设备链表。另外，对每一次匹配上设备，PCI层都会调用pci_driver->probe函数。

当驱动之后被卸载时，会调用pci_unregister_driver()函数。PCI层会遍历与驱动关联的设备，并调用驱动的remove函数。



Chapter 7. Kernel Infrastructure for Component Initialization

7.1. Boot-Time Kernel Options

Linux允许用户传递内核配置参数给boot loader，接着传递参数给内核。在内核启动阶段，内核调用2次parse_args来处理启动配置。

parse_args程序解析输入字符串，格式为：name_variable=value，寻找特定的关键字并调用正确的处理函数。

7.1.1. Registering a Keyword

内核组件能注册一个关键字和相应的处理函数，这是通过使用宏__setup完成：

```
__setup(string, function_handler)
```

下面是个例子：

```
__setup("netdev=", netdev_boot_setup);
```

略...

Chapter 8. Device Registration and Initialization

如果不考虑模块加载，注册和释放设备是对用户独立的；内核驱动它们。一个仅被注册的设备还不能运行。
开启和关闭一个设备需要用户的干预。一旦设备被内核注册，用户能通过命令看见，配置，和启动它。

8.1. When a Device Is Registered

网络设备的注册发生在下列情况：

Loading an NIC's device driver：一个NIC设备驱动会在启动时被初始化，如果它被编译进了内核，或者以模块在运行时初始化。无论何时发生初始化，所有被该驱动控制的NIC被注册。

Inserting a hot-pluggable network device：当用户插入一个热拔插NIC时，内核通知它的驱动，接着注册该设备。

8.2. When a Device Is Unregistered

Unloading an NIC device driver：如果设备驱动是以模块加载的，当模块被移除时，会对使用该驱动的设备调用pci_driver->remove程序。

Removing a hot-pluggable network device

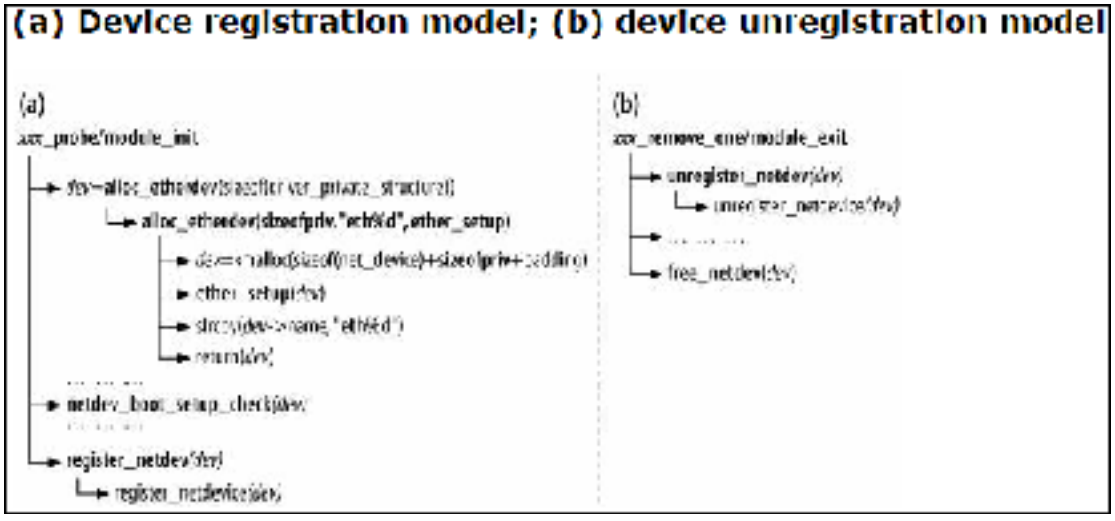
8.3. Allocating net_device Structures

网络设备被net_device数据结构代表。他们通常在内核代码中被命名为dev。这个数据结构由alloc_netdev()分配，它需要3个输入参数：

1. Size of private data structure
2. Device name
3. Setup routine：这个程序被用来初始化net_device的成员。

Table 8-1. Wrappers for the alloc_netdev function		
Network device type	**Wrapper name**	**Wrapper definition**
Ethernet	alloc_etherdev	return alloc_netdev(sizeof_priv, "eth%d", ether_setup);
Fiber Distributed Data Interface	alloc_fddidev	return alloc_netdev(sizeof_priv, "fddi%d", fddi_setup);
High Performace Parallel Interface	alloc_hippi_dev	return alloc_netdev(sizeof_priv, "hip%d", hippi_setup);
Token Ring	alloc_trdev	return alloc_netdev(sizeof_priv, "tr%d", tr_setup);
Fibre Channel	alloc_fcdev	return alloc_netdev(sizeof_priv, "fc%d", fc_setup);
Infrared Data Association	alloc_irdadev	return alloc_netdev(sizeof_priv, "irda%d", irda_device_setup);

8.4. Skeleton of NIC Registration and Unregistration



8.5. Device Initialization

net_device结构非常的大。它的成员被多个不同程序分成chunks来初始化：

1. Device drivers:IRQ，I/O内存，I/O端口的值依赖于硬件配置，它们的初始化是

通过设备驱动完成。

2. Device type:通过xxx_setup程序来进行初始化。例如，以太网设备就是通过ether_setup。
3. Features:强制的和可选的特性也需要被初始化。

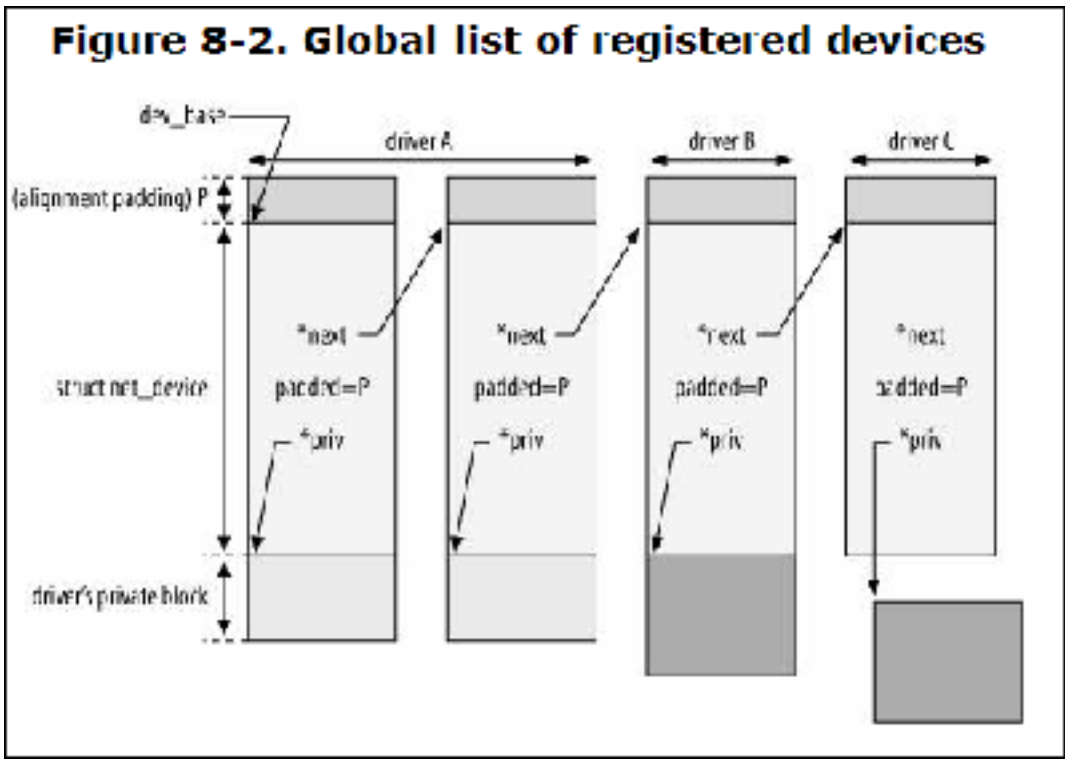
8.5.1. Device Driver Initializations

用设备驱动来初始化的net_device数据结构成员通常是使用xxx_probe函数来实现的。

8.5.2. Device Type Initialization: xxx_setup Functions

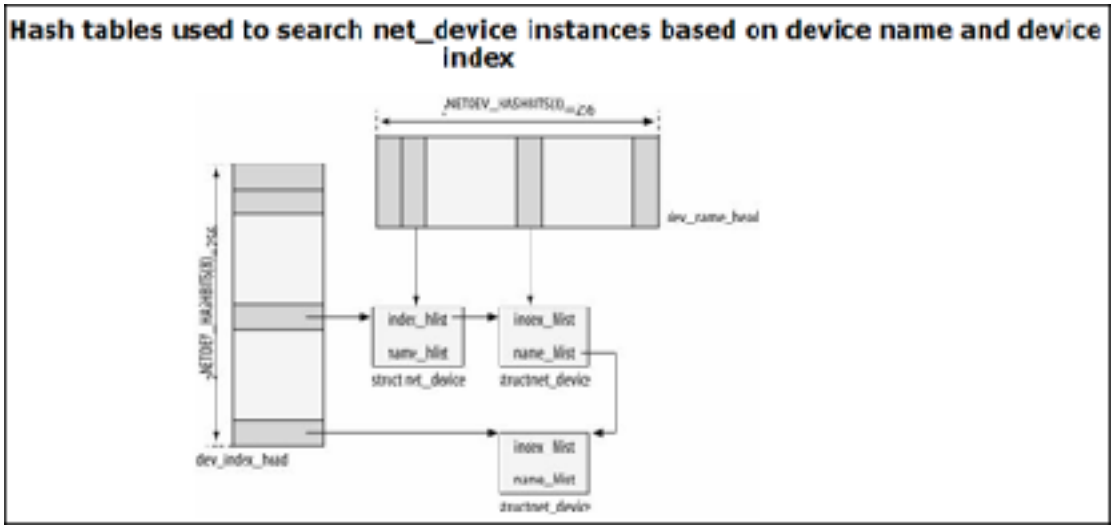
8.6. Organization of net_device Structures

Figure 8-2. Global list of registered devices



private block可以直接跟在net_device后面，也可以不连续。在net_device前面通常会有填充数据，这是为了对齐目的。因此，dev_base指向net_device的地址，dev->padded保存了填充量。

如下图：net_device被链入两个哈希表：dev_name_head，dev_index_head



8.7. Device State

net_device数据结构中有部分成员用来代表设备状态，包括：

1. flags:这是一个位图，用来保存不同的flag，绝大多数代表了设备的能力。然而，IFF_UP用来表示设备是否是激活的。
2. reg_state:设备注册状态。
3. state:设备使用的队列规则的状态。

8.8. Registering and Unregistering Devices

网络设备在内核中注册和释放是分别通过register_netdev()和unregister_netdev()。这两个函数是对程序register_netdevice()和unregister_netdevice()的简单包装。

下图展示了net_device可以被设置的注册状态，特别的：

- 在NETREG_UNINITIALIZED 和 NETREG_REGISTERED之间有许多中间状态的转变。
- net_device中的init和uninit能被设备驱动用来初始化和清空私有数据。他们主要被虚拟设备使用。
- 只有当所有对net_device的引用释放后，才会解除注册一个设备。因此，nedev_wait_allrefs()直到所有对net_device的引用释放后才会返回。
- 注册和释放设备都是通过netdev_run_todo()完成的。

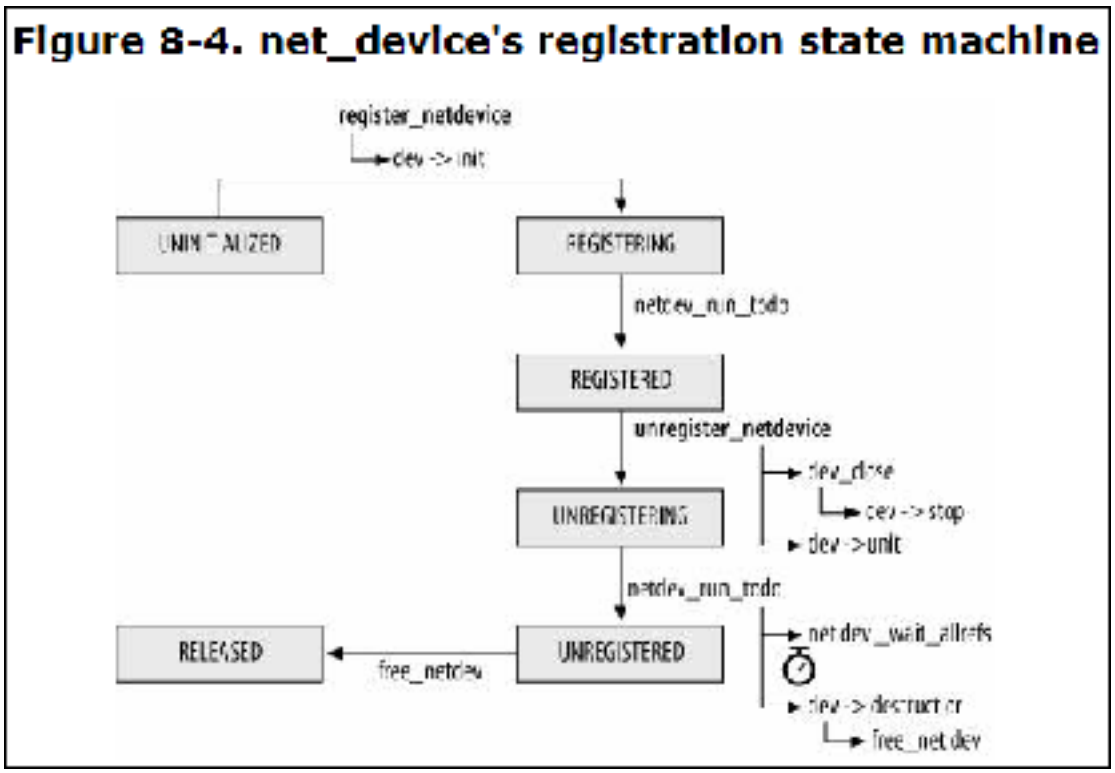


Figure 8-4. net_device's registration state machine

8.8.1. Split Operations: netdev_run_todo

`register_netdevice()`实现了设备注册部分流程，接着由`netdev_run_todo()`完成。

如下图，`net_device`数据结构被`Routing Netlink semaphore`保护着，通过`rtnl_lock`和`rtnl_unlock`函数来请求和释放锁。一旦`register_netdevice()`完成它的工作，它通过调用`net_set_todo()`将新分配的`net_device`结构链入`net_todo_list`。这个链表包含了已经被注册或释放完成的设备。这个链表不能被其它内核线程或周期性定时器处理；它仅被`register_netdev`函数在释放锁时间接处理。

`rtnl_unlock`不仅仅释放锁，也调用了`netdev_run_todo()`。这个函数浏览了`net_todo_list`并完成`net_device`注册。

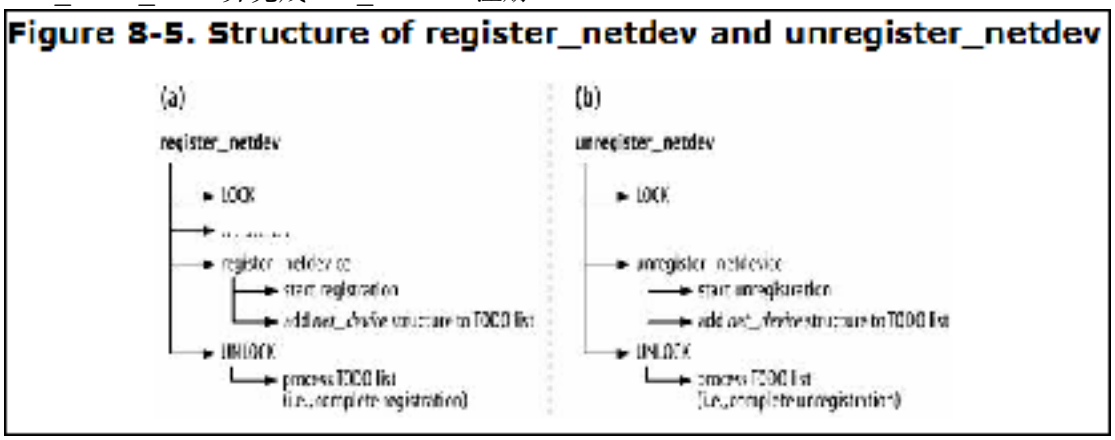


Figure 8-5. Structure of register_netdev and unregister_netdev

8.8.2. Device Registration Status Notification

内核组件和用户空间应用程序都关注一个网络设备的注册，释放，解除激活，激活。关于这些事件的`notification`能通过下列渠道发送：

1. netdev_chain: 内核组件能注册进这个notification chain。
2. Netlink's RTMGRP_LINK multicast group: 用户空间应用程序，例如监视工具或路由协议，能注册进RTnetlink的RTMGRP_LINK multicast group。

8.9. Device Registration

8.9.1. register_netdevice Function

register_netdevice()开始了设备注册并且调用了net_set_todo()，最终调用了netdev_run_todo()来完成注册。

下面是register_netdevice()执行的主要任务:

- 初始化net_device的某些成员，包括用于加锁的成员。
- 如果内核支持Divert功能，则分配一个配置块并链到dev->divert上。这是通过alloc_divert_blk()函数实现的。
- 如果设备驱动的dev->init有定义，执行它。
- 通过dev_new_index()函数分配给设备一个唯一的标识符。
- 把net_device添加到全局链表dev_base中，并插入到2个哈希表。
- 检测feature flags是否合法的组合在一起。
- 设置dev->state的__LINK_STATE_PRESENT标志位，使设备对系统可用。
- 通过dev_init_scheduler()函数初始化设备的队列规则，这是用在流量控制功能中的。
- 通过netdev_chain notification chain来通知所有对设备注册感兴趣的子系统。

当netdev_run_todo()被调用来完成注册时，它只是更新dev->reg_state并把设备注册金sysfs文件系统。

8.10. Device Unregistration

为了释放一个设备，内核和相关的设备驱动需要把注册时所执行的步骤取消掉，并且:

- 通过dev_close关闭设备。
- 释放所有相关资源(IRQ，I/O内存，I/O端口等等)。
- 把net_device结构从全局链表dev_base中和2个哈希表中移除。
- 一旦对net_device结构的引用变为0，释放net_device数据结构，驱动的私有数据结构，以及其它任何链在上面的内存块。net_device结构是通过free_netdev()完成。
- 移除添加在/proc和/sys文件系统中的相应文件。

8.10.1. unregister_netdevice Function

unregister_netdevice主要完成了下列任务:

- 如果设备没有被解除激活，首先调用dev_close关闭它。
- 把net_device结构从全局链表dev_list和2个哈希表中移除。
- 调用dev_shutdown把所有与设备相联系的队列规则被销毁。
- 一个NETDEV_UNREGISTER notification发送到netdev_chain中，让其它内核组件知道该事件。
- 用户空间被通知解除注册事件的发生。
- 任何链接到net_device上的数据块被释放。
- dev->init执行的步骤在这里将被dev->uninit取消。
- 某些功能，例如bonding，允许你把一些设备组合在一起并且把它们看成一个带有特

殊特性的单一虚拟设备。在这组设备中，其中一个通常被选为master，因为它将扮演特殊的角色。当设备被移除时，它应该把dev->master置为NULL。

最后，net_set_todo()被调用，用来让net_run_todo()完成解除注册。net_run_todo()把设备从sysfs中移除，把dev->reg_state改为NETREG_UNREGISTERED，等待直到引用变为0，通过调用dev->destructor完成解除注册。

8.10.2. Reference Counts

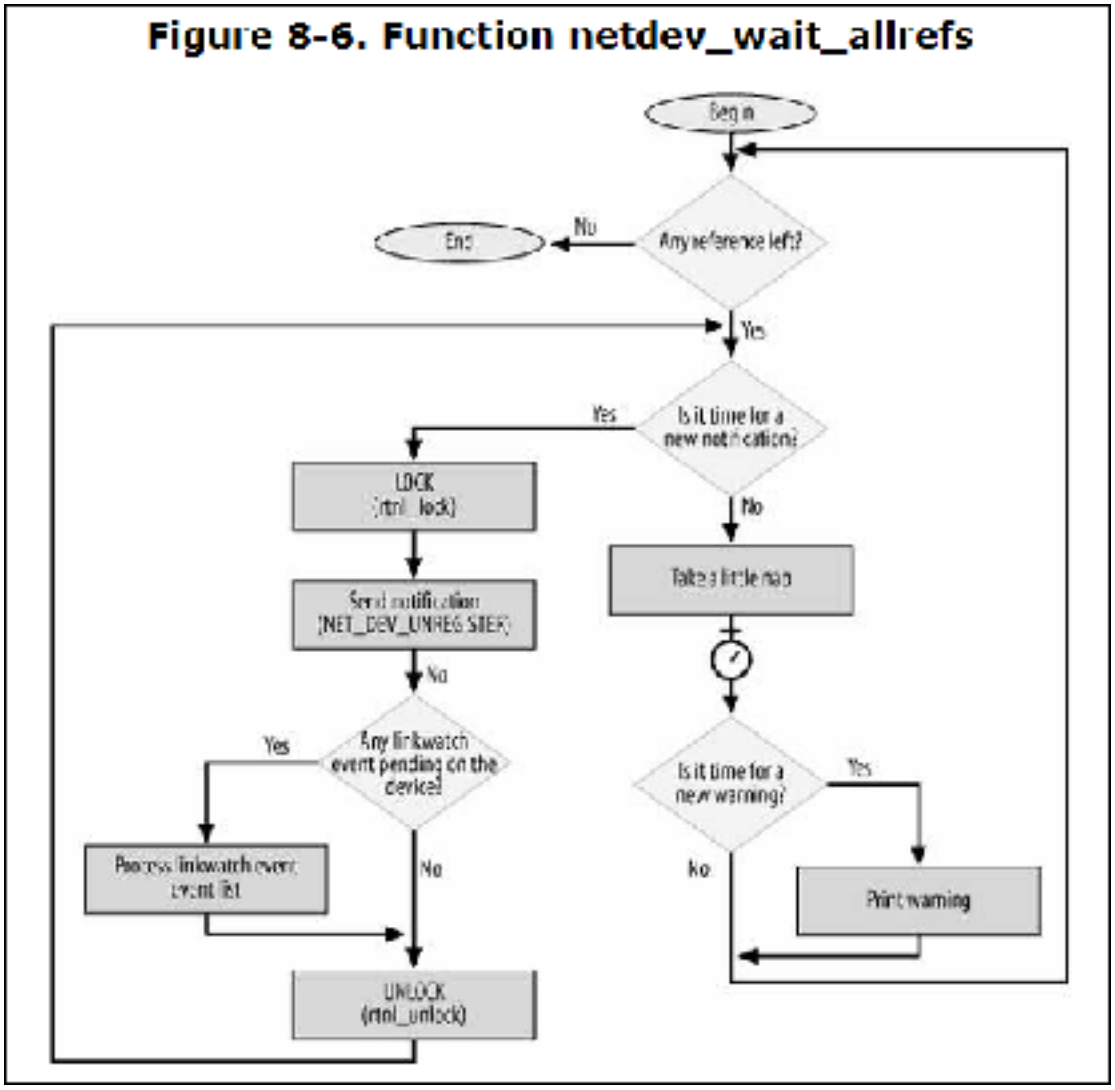
net_device结构直到所有对它的引用被释放后，才能被释放。计数器dev->refcnt保存了引用计数。这个计数器通过dev_hold和dev_put来更新。

当一个设备通过register_netdevice()注册，dev->refcnt被初始化为1。这个计数器只有在调用unregister_netdevice()时才能变成0。

总的来说，在unregister_netdevice()末尾调用dev_put()并不代表就能释放net_device结构：内核必须等待，直到所有引用被释放。因为设备被解除注册后将不再使用，内核需要通知所有引用了net_device的组件释放该引用。这一步是通过发送NETDEV_UNREGISTER notification给netdev_chain notification chain。这也意味着，引用了net_device的组件必须把注册进notification chain；否则，它将不能收到这种notification和采取恰当的行动。
unregister_netdevice()开始了解除注册流程并调用netdev_run_todo()完成它。netdev_run_todo()调用netdev_wait_allrefs()等待，直到所有对net_device结构的引用被释放。

8.10.2.1. Function netdev_wait_allrefs

netdev_wait_allrefs()函数由一个循环组成，仅当dev->refcnt变为0才会结束。每秒钟它发送一个NETDEV_UNREGISTER notification，每隔10秒在控制台上打印一个警告。剩下的时间它都在睡眠。函数直到所有对net_device的引用都释放后才会结束。



8.11. Enabling and Disabling a Network Device

一旦设备被注册，它就可以使用了，但是直到它被用户显式的激活才能传输和接受数据。激活设备是通过调用函数dev_open()。激活一个设备由下列任务组成：

- 如果dev->open有定义，调用它。不是所有设备驱动初始化了该函数。
- 设置dev->state的__LINK_STATE_START标志，dev->flags的IFF_UP标志，表明设备被激活。
- 调用dev_activate初始化用来进行流量控制的出口队列规则，启动看门狗定时器。如果没有用户配置了TC，分配一个FIFO队列。
- 发送一个NETDEV_UP notification给netdev_chain，通知感兴趣的内核组件设备现在被激活了。

设备需要被用户命令显式的解除激活，或被其它事件间接的接触激活。例如，在一个设备被解除注册前，它需要首先被解除激活。网络设备通过dev_close()来解除激活，它的主要任务有：

- 发送一个NETDEV_GOING_DOWN notification给netdev_chain来通知感兴趣内核组件，设备将要被解除激活。
- 调用dev_deactivate来关闭出口队列规则，确保设备能被用来传输数据。关闭看门狗定时器，因为它不再需要了。

- 清除dev->state的__LINK_STATE_START标志位来表明设备已经停止使用了。
- 如果一个轮询动作被调度来读取该设备上的进入数据包，等待该动作的完成。因为__LINK_STATE_START标志位已经被清除了，不会再有轮询被调度到该设备上，但是在标志位清除前会有轮询pending在该设备上。
- 如果dev->stop有定义，调用它。不是所有设备驱动初始化了该函数。
- 清除dev->flags的IFF_UP标志位表明设备停止了。
- 发送一个NETDEV_DOWN notification给netdev_chain通知感兴趣的内核组件，设备现在被停止了。

8.14. Virtual Devices

虚拟设备也需要像真实设备那样被注册和激活。但是，不同之处在于：

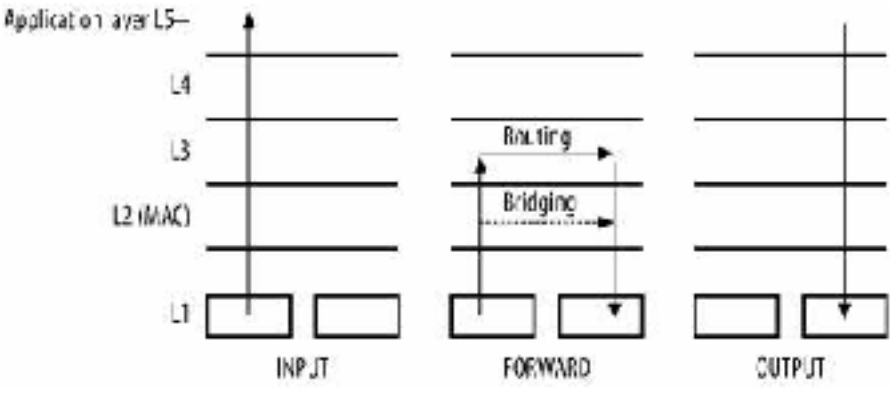
- 虚拟设备有时会调用register_netdevice()或unregister_netdevice()，而不是它们的包装。虚拟设备自己来加锁，因为它们可能需要把锁保持的时间比真实设备更长一些。
- 真实设备不能用用户命令来解除注册；它们只能被解除激活。真实设备只在它们的驱动被卸载时才会被解除注册。虚拟设备相反，可以用用户命令创建和解除注册。这一点可能要依赖于虚拟设备的具体设计。

Part III: Transmission and Reception

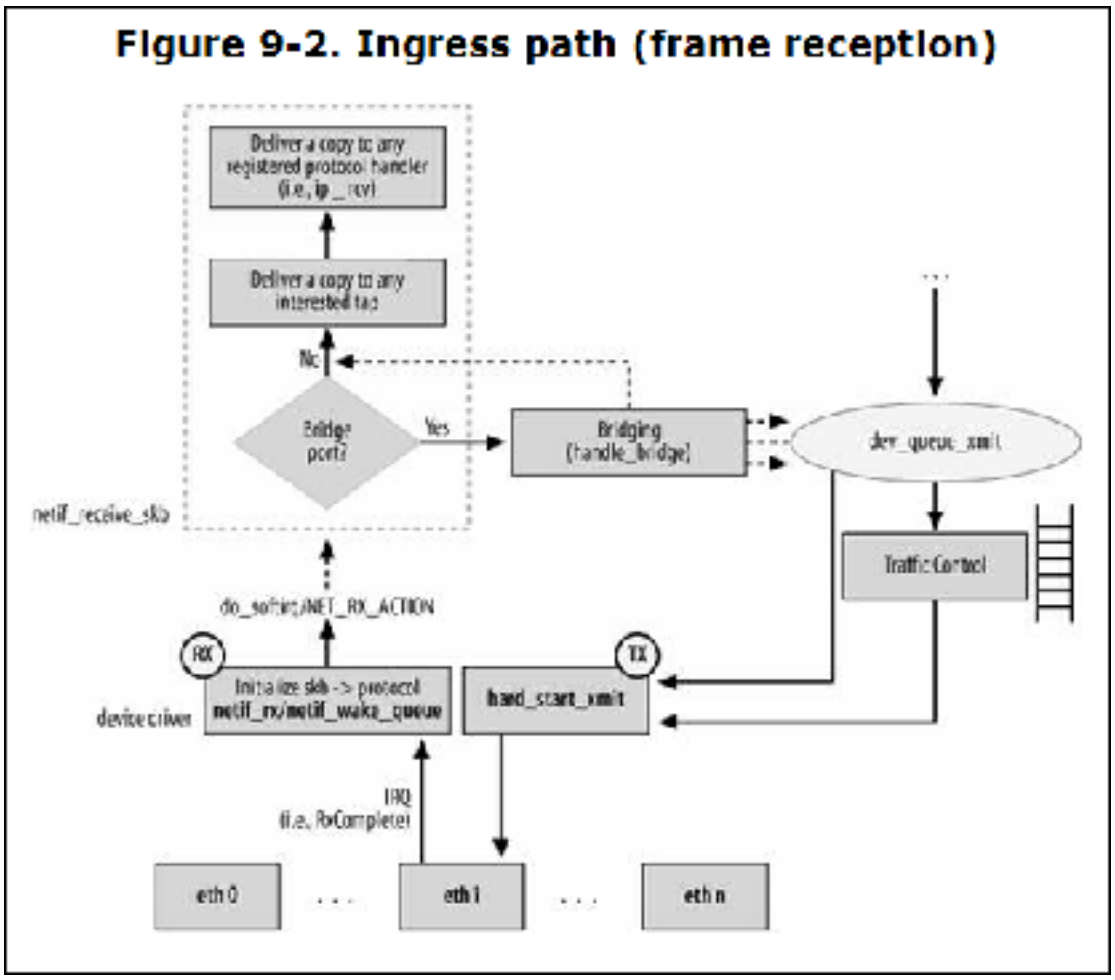
Chapter 9. Interrupts and Network Drivers

9.1. Decisions and Traffic Direction

Figure 9-1. Traffic directions



9.2. Notifying Drivers When Frames Are Received



9.2.1. Polling

略。。。

9.4. softnet_data Structure

每个CPU都有自己的数据帧队列。因为每个CPU都有自己的数据结构来管理入口和出口流量，所以没有必要在不同CPU之间加锁。用于该队列的数据结构叫softnet_data:

```
struct softnet_data
{
    int             throttle;
    int             cng_level;
    int             avg_blog;
    struct sk_buff_head    input_pkt_queue;
    struct list_head       poll_list;
    struct net_device      *output_queue;
    struct sk_buff         *completion_queue;
    struct net_device      backlog_dev;
}
```

这个数据结构包含了用于接收和传输的成员。换句话说，NET_RX_SOFTIRQ 和NET_TX_SOFTIRQ软中断都是使用该数据结构。入口数据帧进入input_pkt_queue，出口帧被放进特殊的队列，那个队列被TC控制而不是被软中断控制，但是，软中断仍然用来清除发送数据包后的buffers。

9.4.1. Fields of softnet_data

无论网卡驱动是否采用NAPI机制，它们都会使用softnet_data数据结构。

1. throttle，avg_blog，cng_level：这三个参数用在拥塞管理算法中。
2. input_pkt_queue：这个队列由net_dev_init()初始化，保存了进入的数据帧。他被用在非NAPI驱动中；在NAPI中被更新为用作私有队列。
3. backlog_dev：这是一个net_device类型的内嵌数据结构，它代表了一个被net_rx_action()调度执行的设备。这个成员被用在非NAPI驱动中。
4. poll_list：这是一个双向链表，挂载了设备，每个设备都有等待处理其中的数据帧。
5. output_queue，completion_queue：output_queue是一个设备链表，其中的设备带有需要传输的数据帧。completion_queue是一个buffer链表，代表了被成功传输的数据帧，所以需要被释放。

throttle被看成一个布尔变量，当CPU过载时被设为真，否则为假。它的值依赖于input_pkt_queue队列中的数据帧数量。当throttle被设置了，所有被该CPU接收的进入数据帧将被丢弃。

avg_blog代表了input_pkt_queue队列长度的平均权重；它的范围是0 ~ netdev_max_backlog。avg_blog被用来计算cng_level。

Chapter 10. Frame Reception

数据帧的接受的主要步骤：

- 拷贝数据帧到sk_buff数据结构。（如果DMA被使用了，驱动只需要初始化一个指针而不需要拷贝工作）
- 初始化sk_buff某些参数，它将在之后被用在网络层（例如skb->protocol，他将用来指示高层协议处理函数）。
- 更新设备的一些私有数据。
- 通知内核，新的数据帧接收到了。这是通过NET_RX_SOFTIRQ软中断来实现的。

10.2. Enabling and Disabling a Device

当net_device->state的__LINK_STATE_START被置位时，设备是激活状态。这个标识符通常在设备打开时(dev_open)置位，在设备关闭时(dev_close)清0。__LINK_STATE_START标识符能被netif_running()函数检测。

10.4. Notifying the Kernel of Frame Reception: NAPI and netif_rx

Linux驱动程序可以有两种方法通知内核新数据帧的到达：

- By means of the old function netif_rx:利用中断来处理数据帧。
- By means of the NAPI mechanism

下面的代码片断来自vortex_rx，使用的是早期的函数netif_rx，非NAPI模式的网卡驱动大多执行类似的功能：

```
    skb = dev_alloc_skb(pkt_len + 5);
        ... ... ...
    if (skb != NULL) {
        skb->dev = dev;
        skb_reserve(skb, 2);     /* Align IP on 16 byte boundaries
*/
                ... ... ...
                /* copy the DATA into the sk_buff structure */
```

```
            ... ... ...
        skb->protocol = eth_type_trans(skb, dev);
        netif_rx(skb);
        dev->last_rx = jiffies;
            ... ... ...
    }
```

首先，使用dev_alloc_skb()函数分配一个sk_buff数据结构，并把数据帧拷贝进去。注意到，在拷贝之前，代码使sk_buff预留了2个字节来使IP头对16字节对齐。以太网的L2头部是14字节，所以要预留2个字节来使IP头对齐。

eth_type_trans()函数被用来抽取协议名，然后保存到skb->protocol中。

10.4.1. Introduction to the New API (NAPI)

NAPI的主要思想很简单：它混合使用了中断和轮询。如果内核还没有处理完前一个数据帧时，新数据帧已经接收到了，那么驱动就不需要产生其它的中断：只需要让内核简单的去处理设备的输入队列（该设备的中断已经关闭了），并且当队列空了时再重新打开中断。这种方法集合了中断和轮询的优点。

从内核处理的角度，下面是NAPI方法的一些优点：

- 减少CPU负载（因为只有较少的中断）：在高负载的网络中，NAPI有很好的性能。但在低负载网络中，NAPI表现不佳，因为浪费了时间在轮询上。
- 更加的公平的处理多个设备：在后文中，我们将看见设备的入口队列是如何被公平的轮流处理。这确保了当其他设备高负载时，低负载的设备能够有可接受的延时。

10.4.2. net_device Fields Used by NAPI

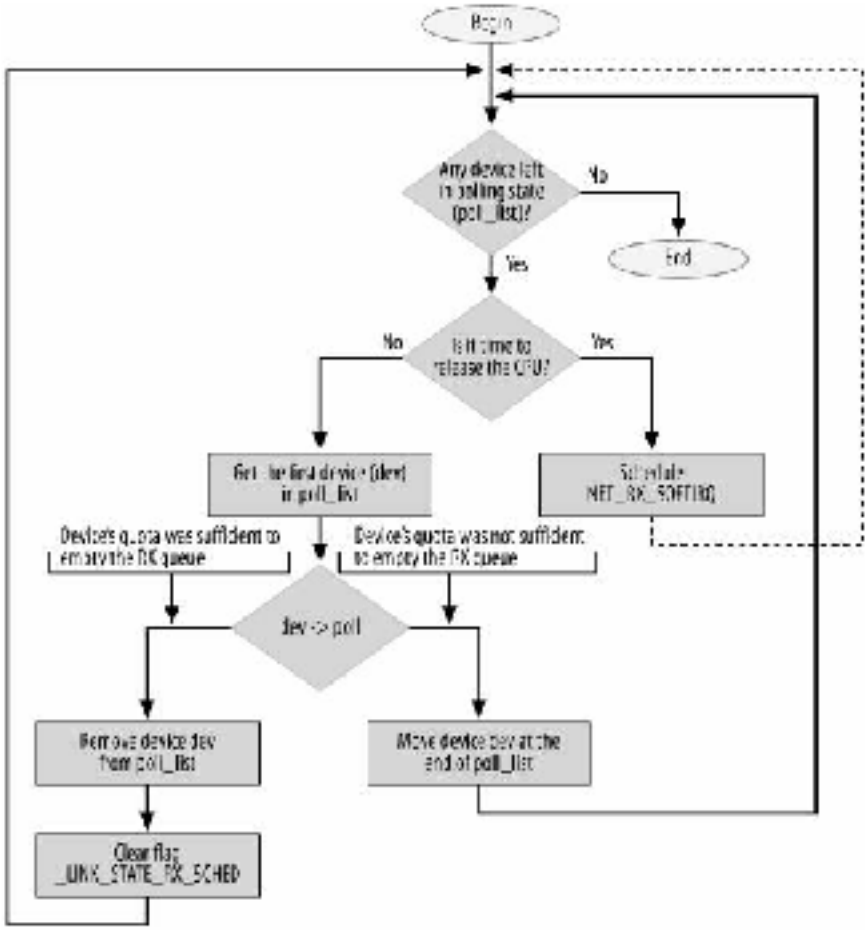
4个新成员被加进net_device结构，被NET_RX_SOFTIRQ软中断用来实现NAPI功能：

- poll：一个函数指针用来从设备入口队列中取下数据包。
- poll_list：设备链表，上面的设备都有新数据帧在入口队列中等待被处理。这些设备被称为处于轮询状态。链表头在softnet_data->poll_list。这个链表上的设备都关闭了中断并且内核当前正在轮询它们。
- quota，weight：quota是一个整数，代表了能被poll函数取下来的数据包最大个数。这个值得增加是以weight为单位，它被用来实现了不同设备之间的某种公平性。较低的quota意味着较低的延时以及其它设备有较少风险变成饥饿。换句话说，一个较少的quota增加了设备间切换的总数，造成了总体开销的增加。

10.4.3. net_rx_action and NAPI

下图展示了当内核轮询进入的网络流量时的情形。

Figure 10-1. net_rx_action function and NAPI overview



我们已经知道net_rx_action()函数是NET_RX_SOFTIRQ软中断的响应函数。net_rx_action()函数浏览处于轮询状态的设备的链表，并且对每个设备调用相应的poll函数来处理入口队列中的数据帧。

net_rx_action()函数限制了它的执行时间，当它的执行超过限制时间或处理了超过限制的数据帧，这个函数将重新调度。每个设备限制了每次调用poll方法能处理的数据帧数目。当一个设备不能清空它的入口队列，它只能等待到下一次执行poll方法。

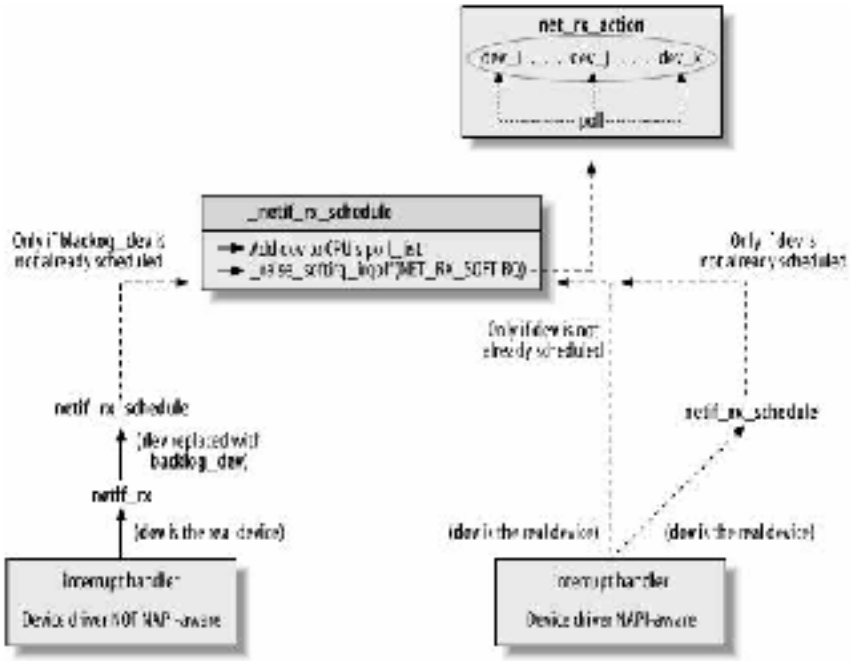
10.4.4. Old Versus New Driver Interfaces

下图展示了NAPI驱动和非NAPI驱动在通知内核新数据帧到达时的区别。

从设备驱动的角度看，在NAPI和非NAPI之间有2点差异：

1. NAPI驱动必须提供poll方法。
2. 接收数据帧所使用的函数不同：非NAPI驱动调用了netif_rx()函数，而NAPI驱动调用的是__netif_rx_schedule()函数。

Figure 10-2. NAPI-aware drivers versus non-NAPI-aware devices



10.4.5. Manipulating poll_list

添加设备进入poll_list是使用函数netif_rx_schedule()或__netif_rx_schedule()。把设备从poll_list中移除是通过使用函数netif_rx_complete()和__netif_rx_complete()。

设备能通过netif_poll_disable()和netif_poll_enable()暂时的开启和关闭轮询。但这并不代表设备驱动要使用非NAPI模型。例如，当设备需要被重新设置时，就可能会关闭轮询。

netif_rx_schedule()函数会过滤掉已经在poll_list(例如，__LINK_STATE_RX_SCHED被设置)中的设备。基于这个原因，如果一个驱动设置了该标志位但是还没有把设备添加进poll_list，它将关闭设备的轮询：设备不会添加进poll_list。netif_poll_disable()函数工作方式是：如果__LINK_STATE_RX_SCHED没有被设置，它简单的设置它并返回。否则，它等待该标志位被清除再设置它。

10.5. Old Interface Between Device Drivers and Kernel: First Part of netif_rx

当有进入的数据帧等待处理时，设备驱动将调用netif_rx()函数，它的工作是产生软中断来处理该帧。

netif_rx()函数通常被驱动程序在中断上下文中调用，不过也有例外，最明显的是在回路设备中会被调用。

不同的CPU可以同时执行netif_rx()函数，这不会造成问题，因为每个CPU与一个私有softnet_data结构相联系，这个结构保存了状态信息。

函数原型：int netif_rx(struct sk_buff *skb)

它的唯一的输入参数是一个被设备用来接收数据帧的buffer，返回值是拥塞等级的指示值。

netif_rx()函数的主要工作包括：

- 初始化sk_buff数据结构的某些成员（例如数据帧接收到的时间）。
- 把接收到的数据帧保存进CPU的私有输入队列，并且通过触发一个软中断NET_RX_SOFTIRQ来通知内核数据帧的到达。
- 更新有关于拥塞等级的统计值。

10.5.1. Initial Tasks of netif_rx

每个设备驱动也保存了统计值在dev->priv中。这些统计值包括接收到的数据帧数量，丢弃的数据帧数量，等等...它们是一个per-CPU变量。

10.5.2. Managing Queues and Scheduling the Bottom Half

输入队列由softnet->input_pkt_queue。每个输入队列有一个给定的最大值，保存在全局变量netdev_max_backlog中，它的值是300。这意味着每个CPU最多有300个数据帧在输入队列中等待被处理，不管系统中设备的数量是多少。

10.6. Congestion Management

拥塞管理是一个用来处理输入数据帧任务的重要组件。一个过载的CPU会变的不稳定和导致系统一个很大的延迟。拥塞管理机制用来确保系统的稳定性在高网络复杂下不会降低。通常在高流量负载情况下，用来减少CPU负载的方法有：

- 尽可能的减少中断的数量

这个机制通过在一个单独的中断中尽可能的处理多个数据帧或使用NAPI。

- 在输入路径中，尽可能早的丢弃数据帧

如果代码知道数据帧将被更高层丢弃，它能直接把数据帧丢弃，这样可以节省CPU时间。例如,如果一个设备驱动知道输入队列满了，它可以立即丢弃一个数据帧。

一个对输出路径的简单优化是:如果一个设备驱动没有系统资源来接收新的要被发送的数据帧，那么如果还要让内核刷新新数据帧给驱动将是对CPU的浪费。

在各种情况下，接收和传输数据帧，内核提供了一个函数集合来设置，清除，取得接收和传输队列的状态，从而允许设备驱动(接收帧)和内核(传输帧)来优化性能。

已经接收到的并在等待被处理的数据帧数量是拥塞等级的一个很好的标志。当设备驱动使用的是NAPI，驱动将不能实现任何拥塞控制机制。这是因为输入数据帧被保存在NIC的内存中或在被驱动管理的接收环形缓冲区中，内核不能跟踪流量的拥塞。相反的，当一个设备驱动不使用NAPI，数据帧将被添加到per-CPU队列中(softnet_data->input_pkt_queue)，并且内核跟踪了这个队列的拥塞等级。本章我们将考虑后一种情况。

队列中的当前数据帧数量不能准确的表示真实的拥塞等级。一个平均队列长度是队列状态的更好的表述。在Linux网络栈中，平均队列长度用softnet_data数据结构的两个成员代表：cng_level和avg_blog。

缺省的，每次一个数据帧被加入input_pkt_queue队列，avg_blog被更新并且一个相关的拥塞等级被计算，保存进cng_level。后者被用作netif_rx()的返回值。

input_pkt_queue队列中数据帧的数量不能超过一个最大值。当达到最大值时，之后的

数据帧将会被丢弃。

10.6.1. Congestion Management in netif_rx

当队列变成空时，softnet_data->throttle被清除。更精确的，当第一个数据帧被加入一个空队列，它将被netif_rx()清除。

10.7. Processing the NET_RX_SOFTIRQ: net_rx_action

net_rx_action()是一个底半部函数，用来处理进入的数据帧。当驱动通知内核新数据帧的到达时，将触发该函数。

数据帧能在2个位置等待net_rx_action()函数处理：

- 一个共享的per-CPU队列：非NAPI设备的中断处理函数，调用了netif_rx()函数把数据帧放入正在执行中断处理函数的CPU的softnet_data->input_pkt_queue队列。
- 设备内存：NAPI驱动使用的poll方法，直接从设备中提取数据帧(或者设备驱动的环形接收区)。

net_rx_action()的任务很简单：查看poll_list链表上的设备，并调用与之相关的poll函数，直到下列条件发生：

- 链表中没有设备。
- net_rx_action()已经运行了过长的时间，因此被建议释放CPU。
- 出队被处理的数据帧数量已经达到了一个给定上限(budget)。

队列的大小被限制为netdev_max_backlog的值。这个值被看做net_rx_action()的budget。因为net_rx_action()在开中断情况下运行，新的数据帧可以在net_rx_action()运行的同时被加入设备的输入队列。那么，帧的数量可能会大于buget，因此net_rx_action()不得不确保自己在这种情况下不能运行过长的时间。

```
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *queue = &_ _get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;
    int budget = netdev_max_backlog;

    local_irq_disable( );
```

如果当前设备还没有用完它的库存，那么它将得到一个取出buffer的机会。

```
 while (!list_empty(&queue->poll_list)) {
        struct net_device *dev;

        if (budget <= 0 || jiffies - start_time > 1)
            goto softnet_break;

        local_irq_enable( );

                    dev = list_entry(queue->poll_list.next, struct
net_device, poll_list);
```

如果dev->poll因为设备的库存不是足够大来取出所有的buffer而返回，设备将被移回poll_list的末尾：

```
        if (dev->quota <= 0 || dev->poll(dev, &budget)) {
            local_irq_disable( );
            list_del(&dev->poll_list);
            list_add_tail(&dev->poll_list, &queue->poll_list);
```

```
            if (dev->quota < 0)
                dev->quota += dev->weight;
            else
                dev->quota = dev->weight;
        } else {
```

注意到budget的地址被传进了poll函数；因为poll函数将返回一个新的budget值。net_rx_action()中的主循环检测budget，使之没有超过上限。换句话说，budget允许net_rx_action()和poll()函数在限制内协调好。

```
            dev_put(dev);
            local_irq_disable( );
        }
    }
out:
    local_irq_enable( );
    return;
```

最后一些代码，当net_rx_action()函数在buffer仍然还在入口队列中时被执行。在这种情况下，NET_RX_SOFTIRQ软中断将再次被调度，以至于net_rx_action接着能被调用，来处理剩下的buffer:

```
softnet_break:
    _ _get_cpu_var(netdev_rx_stat).time_squeeze++;
    _ _raise_softirq_irqoff(NET_RX_SOFTIRQ);
    goto out;
}
```

10.7.1. Backlog Processing: The process_backlog Poll Virtual Function

net_device数据结构的poll函数，它被net_rx_action()函数执行用来处理队列中的帧。当设备不使用NAPI时，这个函数在net_dev_init()中被缺省的初始化为process_backlog()。

在Linux2.6.12内核中，只有一小部分设备驱动使用NAPI，dev->poll被初始化为驱动自己的函数。本章我们将针对缺省的process_backlog()函数进行分析。

process_backlog()函数从一些初始化活动开始:

```
static int process_backlog(struct net_device *backlog_dev, int
*budget)
{
    int work = 0;
    int quota = min(backlog_dev->quota, *budget);
    struct softnet_data *queue = &_ _get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;
```

接下来是主循环，它尝试着从输入队列中取出所有buffers，这个循环仅在下列条件之一发生时被跳出:

- 队列变空。
- 设备的库存用完了。
- 函数运行了过长时间。

```
    for (;;) {
        struct sk_buff *skb;
        struct net_device *dev;

        local_irq_disable( );
        skb = _ _skb_dequeue(&queue->input_pkt_queue);
        if (!skb)
            goto job_done;
```

```
        local_irq_enable( );

        dev = skb->dev;

        netif_receive_skb(skb);

        dev_put(dev);

        work++;
        if (work >= quota || jiffies - start_time > 1)
            break;
```

netif_receive_skb()函数用来处理数据帧。它在NAPI和非NAPI中，被poll函数来使用。

设备的库存根据成功出队的buffer的数量来进行更新。输入参数budget也会被更新，因为它需要被net_rx_action()来追踪函数还能接着处理多少数据帧：

```
    backlog_dev->quota -= work;
    *budget -= work;
    return -1;
```

如果输入队列变空，主循环会跳转到job_done处。如果函数到达这里，throttle状态能被清0并且设备能被从poll_list移除。__LINK_STATE_RX_SCHED标志也被清0，因为设备在输入队列没有任何东西。

```
job_done:
    backlog_dev->quota -= work;
    *budget -= work;

    list_del(&backlog_dev->poll_list);
    smp_mb_ _before_clear_bit( );
    netif_poll_enable(backlog_dev);

    if (queue->throttle)
        queue->throttle = 0;
    local_irq_enable( );
    return 0;
}
```

10.7.2. Ingress Frame Processing

poll函数使用了netif_receive_skb()函数来处理进入的数据帧。

L2和L3层允许使用多种协议。每个设备驱动与一个特定的硬件类型相联系，因此它很容易解释L2头部并抽取信息来告诉哪个L3层协议要被使用。当net_rx_action()函数被调用时，L3协议类型已经被设备驱动抽取并保存到了skb->protocol中。

netif_receive_skv()函数的主要任务是：

- 把数据帧传递给已有的每个协议族。
- 把数据帧传递给与skb->protocol相关的L3协议处理函数。
- 在该层中需要被处理的某些特性，如bridging。

如果没有协议处理函数与skb->protocol相关联并且没有需要处理的特性（如bridging）需要消耗该数据帧，那么该帧将被丢弃。

在把输入数据帧传递给协议处理函数前，netif_receive_skb()函数必须处理一些能改

变帧目的地的特性。

Bonding允许一组网卡被组织在一起并被当作一个单一的网卡来对待。如果接收到数据帧的网卡属于这组网卡中的一个，sk_buff数据结构中对原有网卡的引用将被改为指向这组网卡的主设备。通过下面这个函数实现：

```
skb_bond(skb);
```

在真正的协议处理函数得到该数据帧之前，Diverter, ingress Traffic Control, 和bridging这些特性必须被处理。

当既没有bridging也没有输入数据帧的TC消耗该数据帧。它将被传递给L3协议处理函数。

此时，接收部分已经完成，它截至L3协议处理函数决定如果处理数据包：

- 把它传递给应用接收者。
- 丢弃。
- 转发。

转发这种选择对路由器是很常见的，但对单网卡的工作站并不常见。

10.7.2.1. Handling special features

netif_receive_skb()检查是否存在任何的Netpoll客户端要消耗数据帧。

TC一般在出口路径上实现Qos。然而，内核的最近的版本中，你能在入口流量上配置过滤器和动作。据此，ing_filter()可以决定输入的buffer要被丢弃或是被在别的地方被处理。

Diverter允许内核改变到另一个主机的数据帧的L2目的地址到本机，因此数据帧能被转发到本机。

handle_diverter()来决定是否改变目的MAC地址，同时skb->pkt_type必须被改变为PACKET_HOST。

Chapter 11. Frame Transmission

数据包的传输与接收过程对称：NET_TX_SOFTIRQ与NET_RX_SOFTIRQ相对，net_tx_actiong()与net_rx_actiong()相对，等等。下面是一些相似之处：

- poll_list是设备链表，其中的设备带有非空的接收队列。output_queue是带有需要发送的数据包的设备的链表。poll_list和output_queue是softnet_data数据结构的两个成员。
- 只有打开了的设备(__LINK_STATE_START被设置)能被调度来接收数据帧。只有允许发送数据帧的设备(__LINK_STATE_XOFF被设置)能被调度来发送。
- 当一个设备被调度来接收，它的__LINK_STATE_RX_SCHED被设置。当一个设备被调度用来发送数据帧，它的__LINK_STATE_SCHED被设置。

dev_queue_xmit()在出口路径上扮演了与netif_rx()在入口路径上扮演的相同的角色：在设备驱动的buffer和内核队列之间传输数据帧。当有设备正在等待传输数据帧和正在等待清除不需要使用的buffer时调用net_tx_action()函数。

11.1. Enabling and Disabling Transmissions

出口队列的状态由net_device->state中的__LINK_STATE_XOFF标志位代表。这个值能用下列函数来控制和检测：

- netif_start_queue：开启设备的发送功能。这个函数通常在设备被激活和重启一个停止了的设备时被调用。
- netif_stop_queue：关闭设备的发送功能。
- netif_queue_stopped：返回出口队列的状态：开启或禁止。

只有设备驱动能开启和禁止设备的发送功能。

11.1.1. Scheduling a Device for Transmission

为了发送数据帧，内核提供了dev_queue_xmit()函数，这个函数从设备的出口队列中取出一个数据帧然后传递给设备的hard_start_xmit方法。dev_queue_xmit()函数可能在某些情况下不能发送数据帧，例如：设备的出口队列被禁止了或被加锁了。为了处理这种情况，内核提供了__netif_schedule()函数，它能调度一个设备用来发送数据帧。

```
static inline void __netif_schedule(struct net_device *dev)
{
    if (!test_and_set_bit(_ _LINK_STATE_SCHED, &dev->state)) {
        unsigned long flags;
        struct softnet_data *sd;

        local_irq_save(flags);
           sd = &_ _get_cpu_var(softnet_data);
        dev->next_sched = sd->output_queue;
        sd->output_queue = dev;
        raise_softirq_irqoff(cpu, NET_TX_SOFTIRQ);
        local_irq_restore(flags);
    }
}
```

__netif_schedule()函数完成了2个主要的任务：

- 把设备添加到output_queue链表的头部。output_queue链表也是个per-CPU变量，它被NAPI和非NAPI设备使用。设备通过net_device->sched指针链入output_queue链表。
- 产生NET_TX_SOFTIRQ软中断，__LINK_STATE_SCHED被用来output_queue链表中的设备，因为这些设备需要发送数据帧。注意到如果设备已经被调度来发送，__netif_schedule()函数不做任何事情。

__netif_schedule()函数的两个包装函数：

netif_schedule()：

在调度设备发送数据帧前，确认设备是开启发送功能的。

```
      static inline void netif_schedule(struct net_device *dev)
      {
          if (!test_bit(_ _LINK_STATE_XOFF, &dev->state))
               __netif_schedule(dev);
}
```

netif_wake_queue()：

开启设备的发送功能，如果发送功能之前是被禁止的，调度设备来发送。

```
      static inline void netif_wake_queue(struct net_device *dev)
      {
          ...
```

```
            if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
                __netif_schedule(dev);
        }
```

test_and_clear_bit()清除__LINK_STATE_XOFF标志位并返回旧值。

设备驱动在下列情况下使用netif_queue()：

- 设备驱动使用一个看门狗定时器来把设备从禁止发送状态恢复。在这种情况下，函数net_device->tx_timeout通常重启网卡。
- 当设备发信号给驱动，它有足够的内存来发送给定大小的数据帧时，设备被唤醒。

11.1.2. Queuing Discipline Interface

几乎所有的设备都使用队列来调度出口流量，内核使用队列规则算法来安排数据帧，这样就能用最有效率的顺序来发送。

每个TC队列规则提供了不同函数指针，它们能被上层使用来完成不同的任务。最重要的函数有：

- enqueue：添加一个元素到队列中。
- dequeue：从队列中抽取一个元素。
- requeue：把之前抽取的元素重新入队（例如：发送失败）。

当一个设备被调度来发送数据帧，下一个帧是通过qdisc_run()函数来选择的，它间接调用了相关联的队列规则的dequeue函数。

qdisc_run()函数的真正工作是由qdisc_restart()函数完成的。

11.1.2.1. qdisc_restart function

网卡被调度来发送数据帧，有时是因为在出口队列中有数据帧正在等待被发送。另外一些时候，可能是因为出口队列被禁止了一段时间，因此之前传输失败会使一些数据帧在队列中。设备驱动并不知道是否有数据帧在队列中，所以它必须调度设备以免有数据帧在等待中。

```
int qdisc_restart(struct net_device *dev)
{
    struct Qdisc *q = dev->qdisc;
    struct sk_buff *skb;

    if ((skb = q->dequeue(q)) != NULL) {
```

假设dequeue函数成功返回。发送一个数据帧需要获得两个锁：

- 保护队列的锁（dev->queue_lock）。它将被qdisc_restart的调用者（dev_queue_xmit）获得。
- 驱动的数据帧发送程序hard_start_xmit()使用的锁（dev->xmit_lock）。如果设备驱动已经实现了它自己的锁结构，它只需设置dev->features中的NETIF_F_LLTX标志（无锁发送功能），从而告诉了上层不需要来获取dev->xmit_lock锁。NETIF_F_LLTX的使用允许内核优化发送数据路径。当然，如果队列是空的，就不需要获取锁。

注意到，qdisc_restart()并不在取出一个buffer时立即释放queue_lock，因为函数当在请求驱动的锁失败时会把buffer重新入队。当函数取得了驱动的锁时就会释放queue_lock，并在返回前再次请求queue_lock。最后，dev_queue_xmit()会释放

它。

当设备驱动不支持NETIF_F_LLTX并且驱动的锁已经被占用了(例如：spin_tyrlock返回0)，发送数据帧会失败。如果qdisc_restart()取得驱动上的锁失败，这意味着另一个CPU在通过同一个设备发送数据帧，此时，qdisc_restart()会把数据帧重新入队并重新调度设备发送数据帧，因为它不想等待。

```
            if (!spin_trylock(&dev->xmit_lock)) {
            collision:
                ...
                goto requeue;
            }
            ...
requeue:
        q->ops->requeue(skb, q);
        netif_schedule(dev);
```

一旦驱动的锁被成功获取，加在队列上的锁被释放，那么另外的CPU能访问这个队列。有时，如果NETIF_F_LLTX被设置了，就不需要获取驱动锁。无论哪种情况下，qdisc_restart()已经准备好了开始真正的工作。

```
            if (!netif_queue_stopped(dev)) {
                int ret;
                if (netdev_nit)
                    dev_queue_xmit_nit(skb, dev);

                ret = dev->hard_start_xmit(skb, dev);
                if (ret == NETDEV_TX_OK) {
                    if (!nolock) {
                        dev->xmit_lock_owner = -1;
                        spin_unlock(&dev->xmit_lock);
                    }
                    spin_lock(&dev->queue_lock);
                    return -1;
                }
                if (ret == NETDEV_TX_LOCKED && nolock) {
                    spin_lock(&dev->queue_lock);
                    goto collision;
                }
            }
```

在前面的章节中，qdisc_run()调用了netif_queue_stopped()来检测出口队列的状态，但是在这里，qdisc_restart()再次检测它。第二次检测并不是多余的。

设备驱动提供了dev->hard_start_xmit()函数来发送数据帧。该函数返回下列值：

- NETDEV_TX_OK:发送成功。buffer还没有被释放。驱动不会自己释放该buffer而是通过NET_TX_SOFTIRQ软中断让内核来释放。这比让驱动自己来释放更有效率些。
- NETDEV_YX_BUSY:驱动发现网卡没有足够的空间。当这种情况被检测到时，驱动通常也会调用netif_stop_queue()。
- NETDEV_TX_LOCKED:驱动被加锁了。这个返回值仅被支持NETIF_F_LLTX的驱动所使用。

总得来说，满足下列条件时，发送失败时，一个数据帧必须被放回队列：

- 队列被关闭(netif_queue_stopped(dev)为真)。
- 另一个CPU持有了驱动上的锁。

- 驱动发送失败(hard_start_xmit()没有返回NETDEV_TX_OK)。

11.1.3. dev_queue_xmit Function

这个函数是设备驱动执行发送的接口。dev_queue_xmit()函数通过两个途径来执行驱动的发送函数hard_start_xmit():

- Interfacing to Traffic Control (the QoS layer):通过qdisc_run()函数。
- Invoking hard_start_xmit directly:不使用TC结构的设备。

dev_queue_xmit()仅接收一个sk_buff作为输入。它包含了函数需要的所有信息。skb->dev是出口设备，skb->data指向payload的开始，skb->len是payload的长度。

dev_queue_xmit()的主要任务是:

- 检测数据帧是否由碎片组成，以及设备是否能通过scatter/gather DMA来处理它们；如果设备不能做到，就合并碎片。
- 确定L4的校验和(TCP/UDP)被计算了，除非设备是通过硬件来计算校验和。
- 选择哪个数据帧被发送(输入参数中的sk_buff可能不是要被发送的数据帧，因为发送队列的存在)。

下面的代码，如果skb_shinfo(skb)->frag_list非NULL说明payload是一个碎片链；否则，payload是一个单一的块。如果有碎片，代码检查scatter/gather DMA特性被设备支持，如果不支持，就合并碎片到一个单一的buffer中。

```
    if (skb_shinfo(skb)->frag_list &&
        !(dev->features&NETIF_F_FRAGLIST) &&
        _ _skb_linearize(skb, GFP_ATOMIC)) {
        goto out_kfree_skb;
    }

    if (skb_shinfo(skb)->nr_frags &&
        (!(dev->features&NETIF_F_SG) || illegal_highdma(dev,
skb)) &&
        _ _skb_linearize(skb, GFP_ATOMIC)) {
        goto out_kfree_skb;
    }
```

碎片的整理是通过__skb_linearize()函数实现的，它可能会由于下列原因而失败:

- 需要保存合并好的碎片的新buffer分配失败。
- sk_buff与其它子系统共享着(它的引用计数大于1)。在这种情况下，函数并不是真正的失败了，而是通过调用BUG()来产生一个警告。

接下来是校验和的计算，有软件实现的也有硬件实现的。

一旦校验和被处理了，所有的头部信息都准备好了；下一步是确定哪个数据帧要被发送。

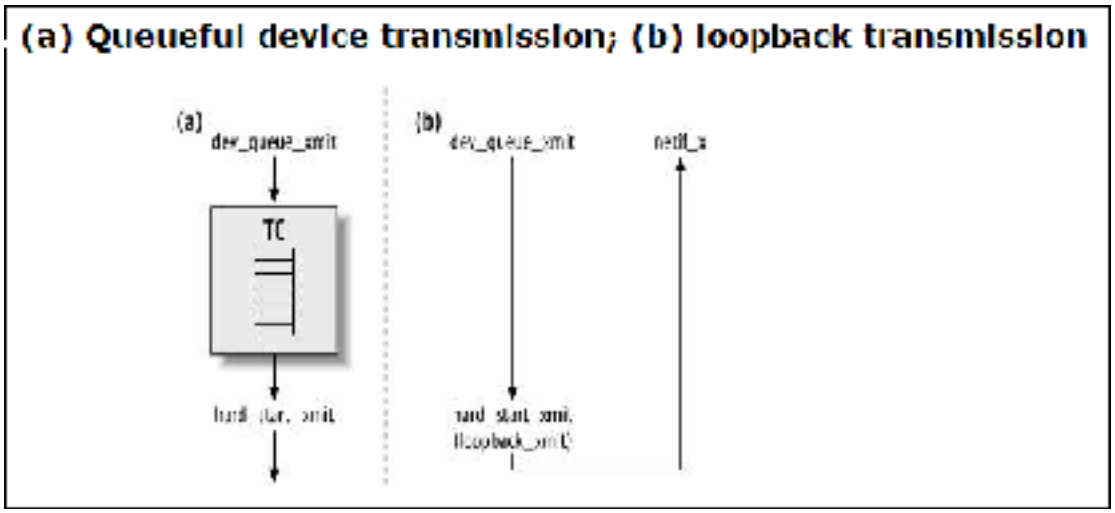
执行到这里时，函数的行为依赖于设备是否使用TC结构以及是否分配了一个队列规则。根据是否使用了队列规则以及那个队列被使用了，函数之前处理的buffer并不一定是接下来要被发送出去的数据帧。

11.1.3.1. Queueful devices

设备的队列规则是通过dev->qdisc来访问的。输入的数据帧通过enqueue函数入队，通过qdisc_run()函数来去除数据帧并发送。

11.1.3.2. Queueless devices

某些设备，例如环回设备，没有队列：当一个数据帧被发送时，它立即被发送。（因为没有地方可以把数据帧重新入队，如果碰到某些错误情况，数据帧将被丢弃）



11.1.4. Processing the NET_TX_SOFTIRQ: net_tx_action

net_tx_action()函数被设备通过raise_softirq_irqoff(NET_TX_SOFTIRQ)触发，它完成了2个主要任务：

- 当发送被开启时，调用netif_wake_queue()。在这种情况下，它确保了正在等待的数据帧被发送出去。
- 当发送完成时，调用dev_kfree_skb_irq()，通知内核buffer能被释放了。在这种情况下，它释放了与成功发送的buffer相关联的sk_buff结构。

释放一个buffer需要一些时间，因此通过调用软中断，把对它的处理推迟。设备驱动调用了dev_kfree_skb_irq()，它简单的把指向需要释放的buffer的指针添加到softnet_data->completion_queue链表中，让net_tx_action()随后完成真正的工作。

让我们来看看net_tx_action()函数如何完成这两个任务。

函数释放了completion_queue链表中的所有buffer。因为设备驱动可以在任何时候添加元素到这个链表中，所以需要关中断，为了使关中断时间尽可能的短，函数把softnet_data->completion_queue置为NULL，原来的值保存在了局部变量clist中。因此它能遍历链表并释放元素，同时设备驱动也能接着添加新的元素进completion_queue链表。

```
    if (sd->completion_queue) {
        struct sk_buff *clist;

        local_irq_disable( );
        clist = sd->completion_queue;
        sd->completion_queue = NULL;
        local_irq_enable( );

        while (clist != NULL) {
            struct sk_buff *skb = clist;
```

```
            clist = clist->next;

            BUG_TRAP(!atomic_read(&skb->users));
            __kfree_skb(skb);
        }
    }
```

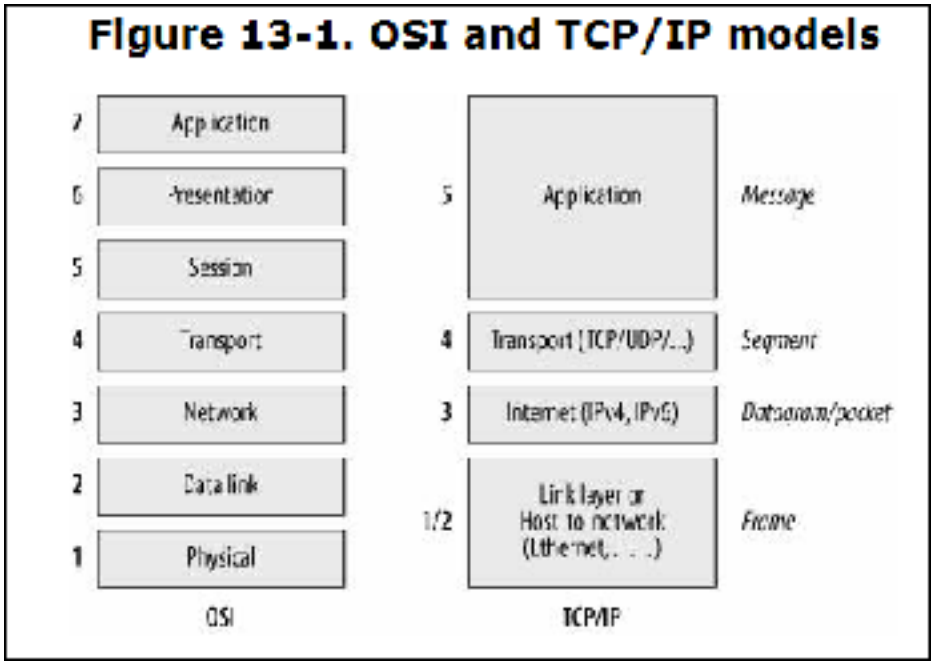
函数的另一半是发送数据帧，注意到对每个设备，在发送数据帧之前，函数需要获得设备的出口队列的锁(dev->queue_lock)。如果函数获得锁失败(因为另一个CPU已经持有了它)，它简单通过netif_schedule()来重新调度设备的发送。

```
    if (sd->output_queue) {
        struct net_device *head;

        local_irq_disable( );
        head = sd->output_queue;
        sd->output_queue = NULL;
        local_irq_enable( );

        while (head) {
            struct net_device *dev = head;
            head = head->next_sched;

            smp_mb__before_clear_bit( );
            clear_bit(__LINK_STATE_SCHED, &dev->state);

            if (spin_trylock(&dev->queue_lock)) {
                qdisc_run(dev);
                spin_unlock(&dev->queue_lock);
            } else {
                netif_schedule(dev);
            }
        }
    }
```

11.1.4.1. Watchdog timer

Chapter 13. Protocol Handlers

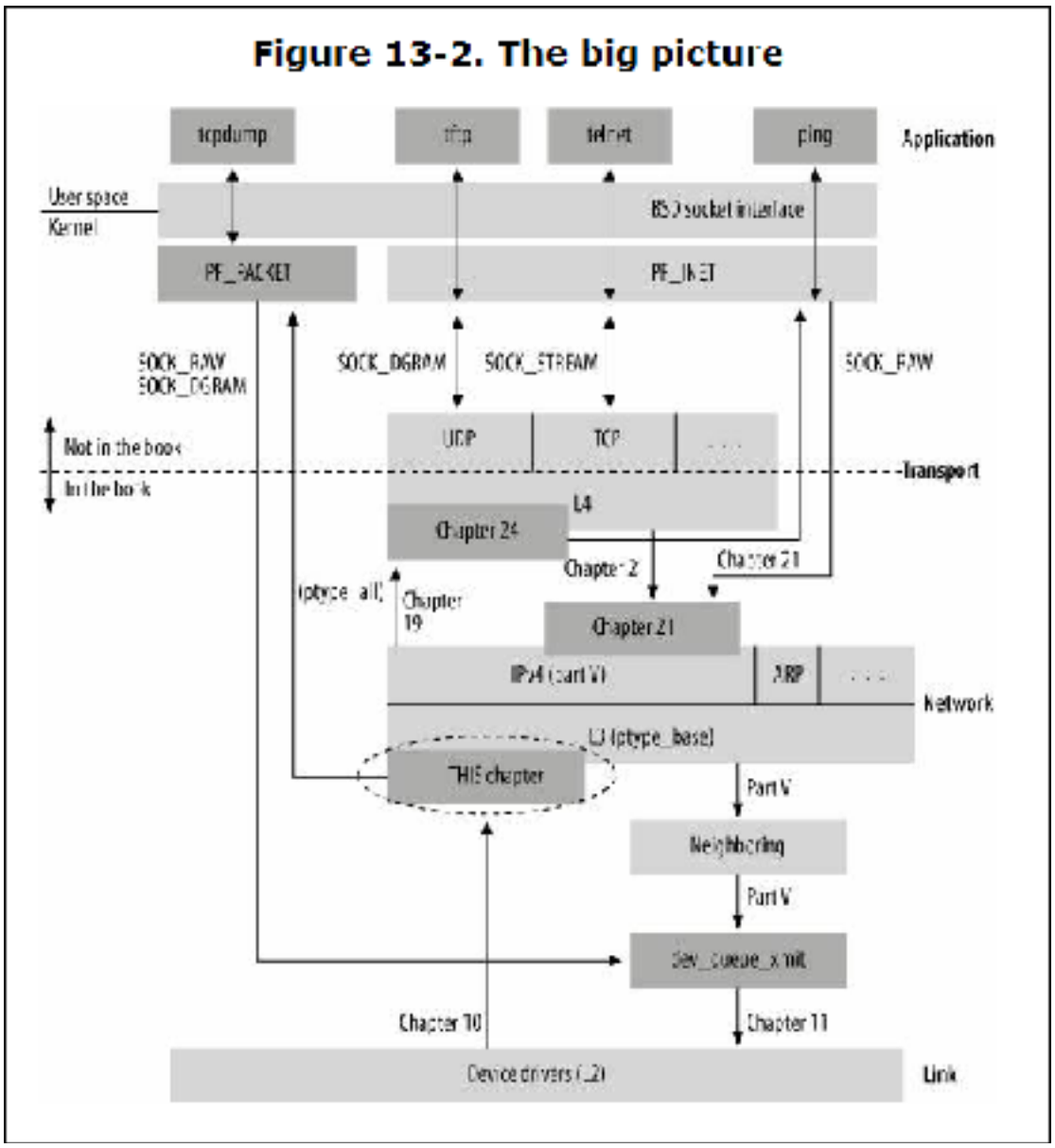
13.1. Overview of Network Stack



在每一层，有许多可用的协议。在最低一层，网卡使用预先定义好的协议来交换数据。网卡的驱动与该协议相关，所有进入网卡的数据都被假设遵从该协议；如果不是，错误将被报告，并且不会发生数据交换。

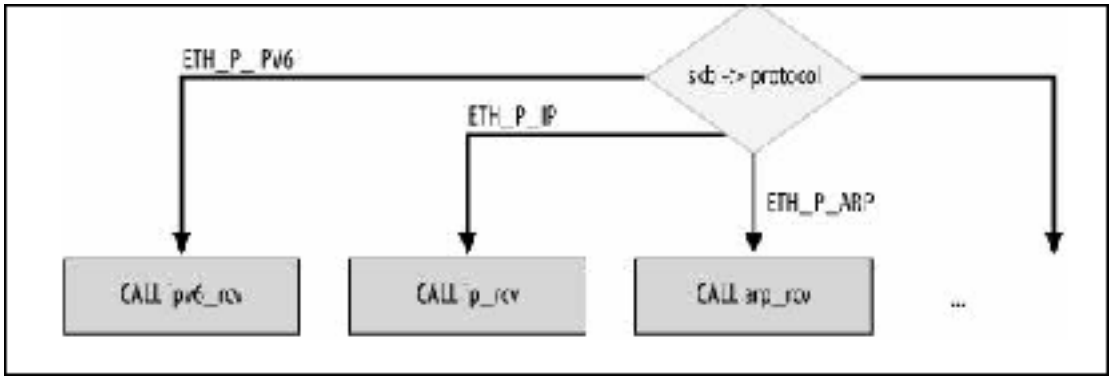
链入层使用的是`frame`，网络层使用的是`packet`，传输层使用的是`segment`，应用层使用的是`message`。

13.1.1. The Big Picture



13.2. Executing the Right Protocol Handler

当设备驱动接收到一个数据帧，它把数据帧保存在一个sk_buff数据结构中，并且把protocol成员初始化。这个成员的值将作为内核识别给定协议的标识符或一个数据帧的MAC头的成员。这个成员被内核函数netif_receive_skb()用来决定哪个处理函数被用来在L3层处理数据包。



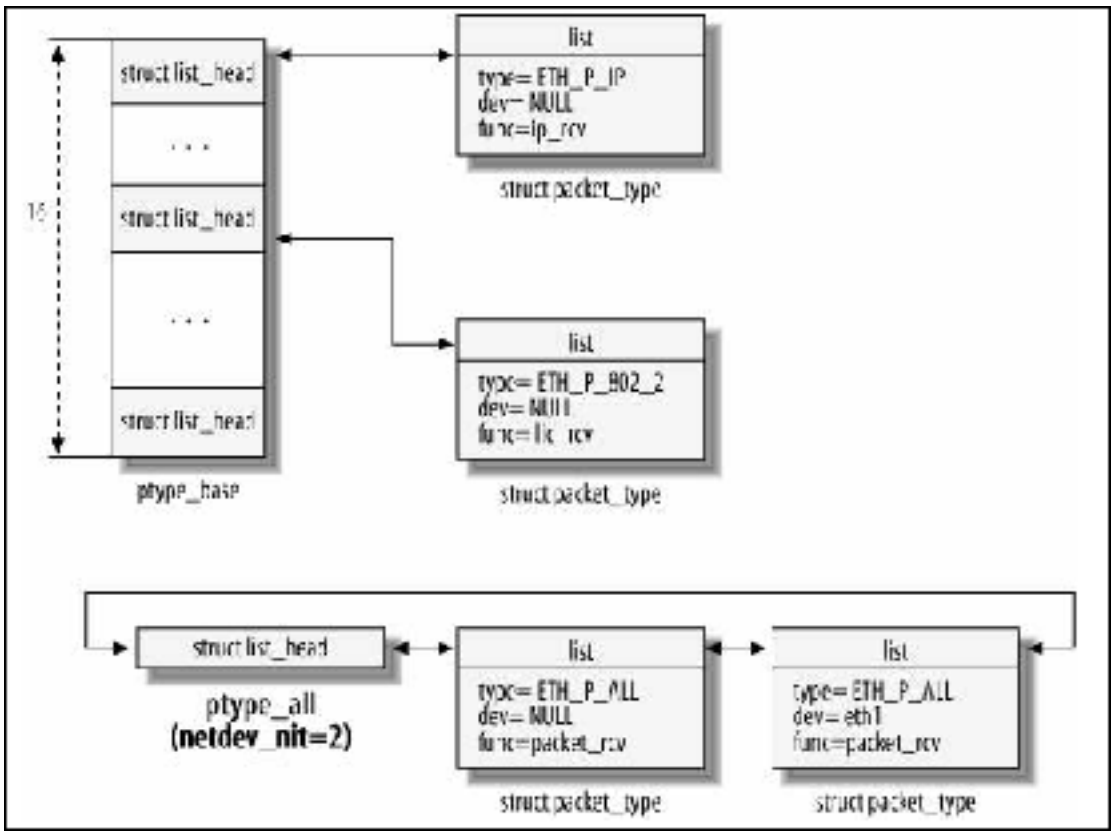
内核涉及的保存在skb->protocol成员中的值大多数在include/linux/if_ether.h

中，名字为ETH_P_XXX。

netif_receive_skb()函数把进入数据帧分配到合适的协议处理函数，如果没有合适的协议处理函数，该帧被丢弃。

13.3. Protocol Handler Organization

下图展示了不同的协议处理函数在内核中的组织。每种协议用packet_type数据结构描述。



为了加快访问速度，一个简单的哈希函数被用在大多数协议中。16个链表被组织进一个数组，这个数组用ptype_base指针指向。当一个协议注册时，使用dev_add_pack()函数，这个函数分配一个packet_type数据结构并散列到16个链表中的一个。

ETH_P_ALL协议被组织进一个链表，它由全局变量ptypr_all指向。这个链表中的协议数量被保存在netdev_nit。

13.4. Protocol Handler Registration

当协议注册时，内核调用dev_add_pack()函数，把packet_type数据结构传给该函数。

```
struct packet_type
{
    unsigned short        type;
    struct net_device       *dev;
     int              (*func) (struct sk_buff *, struct net_device
*,
                      struct packet_type *);
    void             *af_packet_priv;
    struct list_head    *list;
};
```

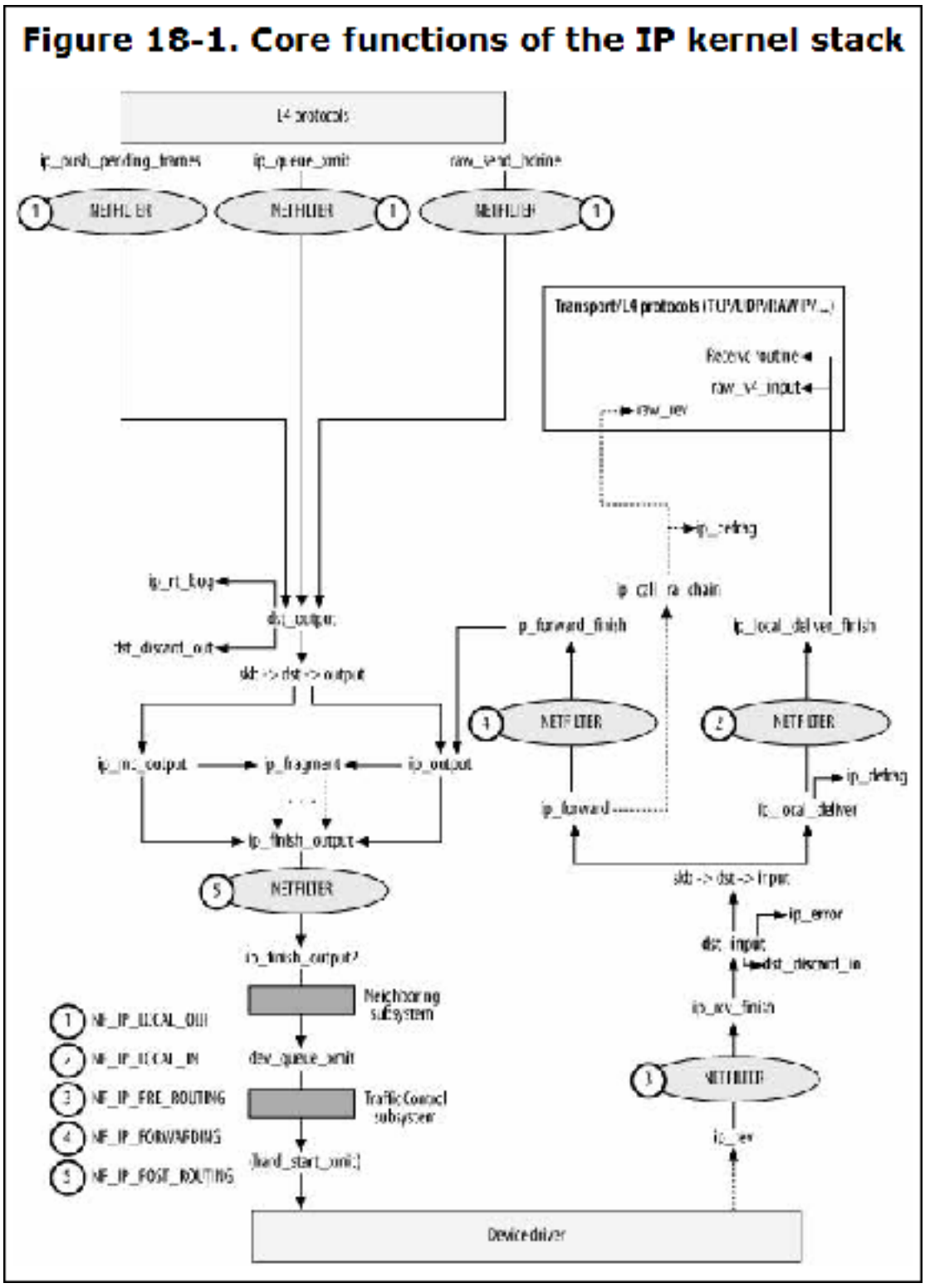
af_packet_priv: 被PF_PACKET socket使用。它是一个指针指向该packet_type创建者相关的sock数据结构。它被用来允许dev_xmit_nit()函数不传递一个buffer给发送者，并且被PF_PACKET接收函数传递进入数据包给合适的socket。

dev_remove_pack()函数与dev_add_pack()函数相反。

Chapter 18. Internet Protocol Version 4 (IPv4): Concepts

18.1. IP Protocol: The Big Picture

Figure 18-1. Core functions of the IP kernel stack



18.2. IP Header

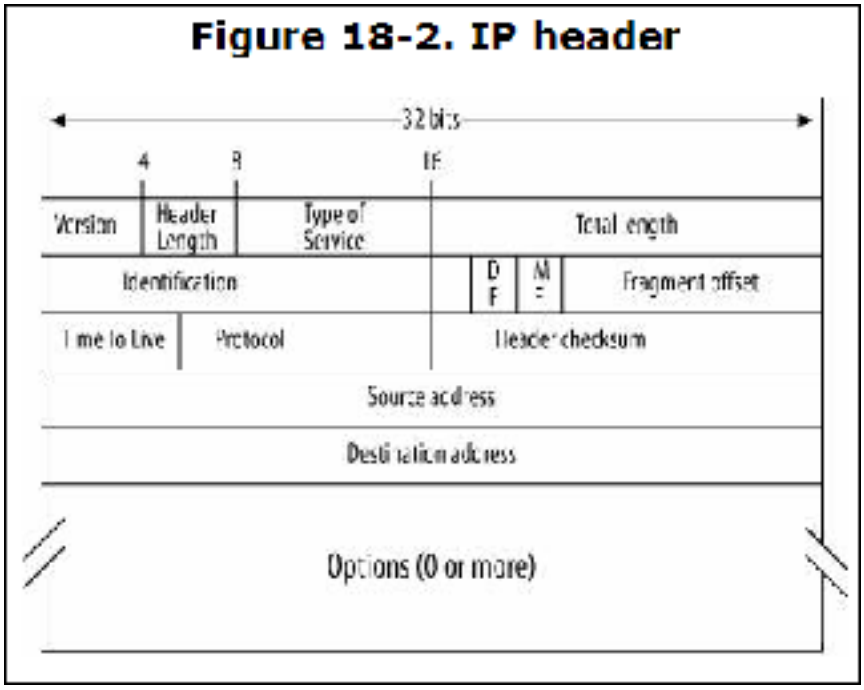
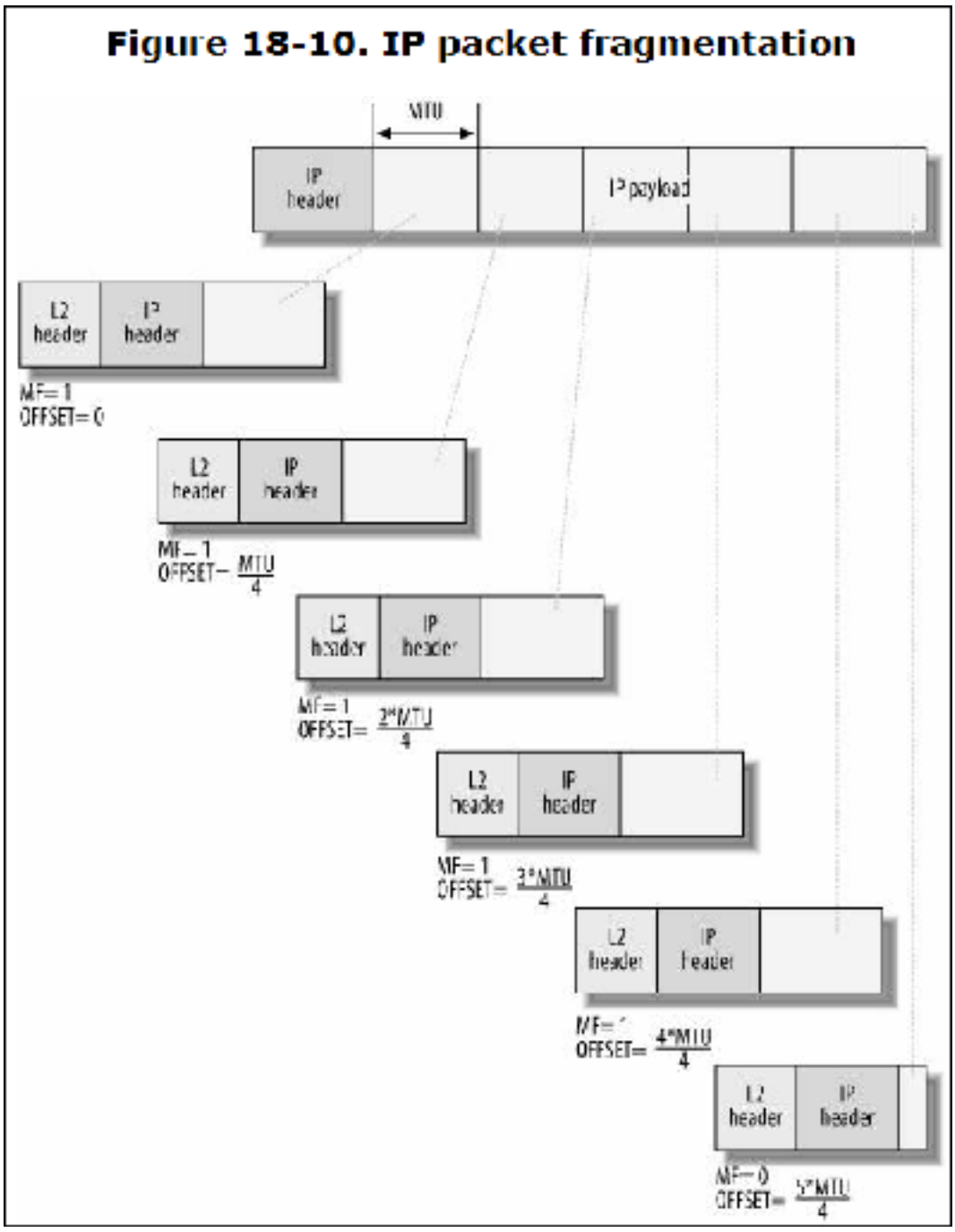


Figure 18-2. IP header

18.4. Packet Fragmentation/Defragmentation

数据包的分片和重组是IP协议的主要部分。IP协议定义了一个数据包的最大长度为64KB，因为数据包头部的len成员是16-bit值。当IP协议需要传输一个大小大于出口网卡的MTU数据包时，它需要把数据包分割成一些小包。

不管MTU的大小，分片程序把数据包分成一系列大小相等的分片，最后一块分片可能不相等。



被分片的的IP包通常在目的主机处被重组，但是中间设备（防火墙，NAT）需要查看整个IP数据包，所以需要重组它。

IP的checksum仅仅覆盖了IP头部（payload通常被更高的协议覆盖）。当分片被创建时，头部各不相同，因此checksum每次都要被重新计算。

18.4.1. Effect of Fragmentation on Higher Layers

数据包的分片和重组耗费了CPU时间和内存。在一个大流量环境下，资源消耗是极其大的。分片也导致了带宽的浪费，因为每个分片包含了L2和L3头部。如果分片太小，这个开销会更明显。

理论上，当L3层决定对数据包分片时，更上层协议并不知道。

尽管TCP和UDP并不知道分片/重组的过程，构建在这两个协议之上的应用却很关心分片/重组。由于担心效率的原因，一个对包延时敏感的应用程序会尝试尽可能避免分片。许多应用程序已经足够聪明，它通过考虑一些因素来尝试避免分片：

- 首先，内核并不是简单的使用出口网卡的MTU，而是使用一种叫路径MTU发现的功能来发现最大包大小，来避免在特定路径上的包分片。
- MTU被设置成576，在RFC791中，每个主机都必须接受最多576字节的包。这个限制减少了分片的可能性。

18.4.2. IP Header Fields Used by Fragmentation/Defragmentation

下面是用来处理分片/重组过程的IP头部成员：

- DF (Don't Fragment)：某些应用不希望分片，因此如果一个数据包超过MTU，它将被丢弃。
- MF (More Fragments)：如果一个数据包被分片了，除了最后一个分片，其它分片该位需要置位。数据包的接收者接收到最后一个分片包时就能知道原有数据包的大小，并不需要收到所有的分片包。
- Fragment Offset：分片包总是8字节对齐，该位是13-bit。offset为0说明该包是第一个分片。
- ID：对一个IP包的所有分片，ID都相同。

18.4.3. Examples of Problems with Fragmentation/Defragmentation

18.4.3.1. Retransmissions

IP包的发送者，只有等到上层协议通知它重传后才会发生重传。一个重传的数据包不会重用同一个ID。然而，主机仍然可能接收到同一个IP分片带有同样的ID，主机必须能够处理该情况。这种情况可能是因为L2层存在回路时。在第四部分有论述。

由于内核不将数据swap到磁盘中(只会swap用户态数据)，处理分片会带来内存的浪费。Linux给分片占用的内存数量设置了上限。

上层通过一些方法检测数据的丢失并且试图重发。由于只重发丢失的分片是不可能的，L4协议不得不重传整个IP包。每个重传会带来一些接收方不得不处理的特殊情况：

- Overlapping：一个分片可能包含前面到达的一些数据。重传的数据包有一个不同的ID，这样他们就不会产生混合。然而，不使用不同ID重传数据包的错误实现的操作系统，或者下一节将介绍的回路问题使重叠变得可能。
- Duplicates：这可以被认为是重叠的一种特殊情况，这里两个分片完全一样。一个分片被认为是重复的，如果它在相同的偏移开始并且有相同的长度。对实际的载荷内容不作检查。除非你在一个安全攻击中，否则没有理由同一个包的传输间载荷内容应该变化。前面提到的L2回路也是一个重复的来源。
- Reception once reassembly is already complete：这种情况下，IP层会将这个分片当作新的IP包的第一个分片。如果没有接收到全部的新分片，IP层会在垃圾回收过程中简单的清理这些重复；否则，它将再新建整个数据包，识别这个包重复的工作有上层协议来完成。

18.4.3.2. Associating fragments with their IP packets

为了识别分片属于哪个包，内核使用下面的参数进行判断：

- 源和目的IP地址
- IP包ID
- L4协议

我们不得不接受IP层偶尔会将来自完全不同的数据包的分片混合在一起的可能性。有些事情出了错。通常只有高层通过错误检查解决这个问题。

Chapter 19. Internet Protocol Version 4 (IPv4): Linux Foundations and Features

19.1. Main IPv4 Data Structures

这个章节介绍了IPv4协议使用的主要数据结构:

- iphdr structure:IP头部。
- ip_options structure:代表了IP包的选项内容。
- ip_cookie structure:这个结构组合了不同种类的被发送数据包所需的信息。
- ipq structure:IP包分片的集合。
- inet_peer structure:内核对每个最近访问过的远程主机都保存了这个结构的实例。所有的inet_peer结构的实例保存在一颗AVL树中。
- ipstats_mib structure:SNMP使用了一个MIB来收集系统的统计信息。ipstats_mib数据结构被用来保存关于IP层的统计信息。
- in_device structure: 这个数据结构保存了所有与一个网络设备IPv4相关的配置信息，例如用户使用ifconfig或ip命令下达的修改。这个数据结构通过net_device->ip_ptr链入net_device数据结构，它可以被in_dev_get和__in_dev_get抽取出来。这两个函数不同之处在于前者加载了所有可能的锁，而后者假设调用者已经加锁完毕了。
- in_ifaddr structure:当在一个网卡上配置了一个Ipv4地址，内核创建一个in_ifaddr结构。
- ipv4_devconf structure:ipv4_devconf数据结构被用来优化一个网络设备的行为。对每个设备都有一个实例，以及一个保存了缺省值(ipv4_devconf_dflt)的实例。
- ipv4_config structure:ipv4_devconf结构保存的是每个设备的配置，ipv4_config保存的是主机的配置。

19.1.1. Checksum-Related Fields from sk_buff and net_device Structures

19.1.1.1. net_device structure

net_device->features成员指明了设备的能力，其中有一小部分定义了设备的硬件校验和能力。下面是被用来控制校验和的标志位:

- NETIF_F_NO_CSUM:设备是可靠的，没有必要来使用任何L4校验。在loopback设备中使用了该特性。
- NETIF_F_IP_CSUM:设备能用硬件来计算L4校验和，但是仅用于IPv4的TCP和UDP。
- NETIF_F_HW_CSUM:设备能用硬件来计算任何协议的校验和。这个特性不常用。

19.1.1.2. sk_buff structure

skb->csum和skb->ip_summed根据skb指向的是接收到的包还是被发送的包会有不同的含义。

当数据包是被接收到的，skb->csum可以含有L4校验和。skb->ip_summed保存的是L4校验和的状态，它代表了设备驱动告诉L4层的信息。一旦L4接收程序接收到了buffer，它可以改变skb->ip_summed。

CHECKSUM_NONE:
在csum中的校验和无效，这可能是由于下列原因造成的:

- 设备没有提供硬件校验和。

设备用硬件计算校验和后发现数据帧已经损坏了。这时，设备驱动可以直接丢弃数据帧。但是一些设备驱动倾向于把ip_summed设置为CHECKSUM_NONE，并让软件再次计算一次校验和。注意到，如果数据帧将要被转发，路由器不会因为L4校验和错误而丢弃它。因此，设备驱动在L4校验和失败时不会丢弃数据帧，而是让L4层接收程序来验证它们。

- 校验和需要被重新计算和重新验证。

CHECKSUM_HW:
NIC已经计算了L4校验和并且拷贝到了skb->csum中。软件仅仅需要把关于伪头部的校验和添加到skb->csum中并验证最后的校验和。这个标志为被看作是后面的标志位的一种特殊情况。

CHECKSUM_UNNECESSARY:
NIC已经计算并验证了L4的校验和，以及伪头部的校验和，因此软件不用再进行任何的L4校验和的验证。
CHECKSUM_UNNECESSARY也能在这种情况下被设置：数据帧出错的几率很小，进行L4校验和的验证是一种浪费。例如loopback device。

当数据包是被用来发送的，csum代表了一个偏移，指向了buffer中的位置，网卡会从该位置来计算校验和。因此这个成员在数据包发送时被使用，而且还仅当校验和是被用硬件计算时。此时ip_summed的作用为:
CHECKSUM_NONE:
协议已经自己实现了校验和;设备不需要做任何事情。当你转发一个输入数据帧，L4校验和已经准备好了，因为它已经被发送主机计算好了;所以，不需要计算它。当ip_summed被设置为CHECKSUM_NONE，csum是无意义的。

CHECKSUM_HW:
协议已经在它的头部保存仅关于伪头部的校验和；设备将完成添加关于L4头和payload的校验和。

在发送数据包时，ip_summed不使用CHECKSUM_UNNECESSARY(它等同于CHECKSUM_NONE)。

在NIC激活时，如果设备驱动设置了NETIF_F_XXX_CSUM功能标志位，CHECKSUM_XXX标志位将被设置进每一个sk_buff中，不论是接收还是发送。

19.2. General Packet Handling

19.2.1. Protocol Initialization

IPv4协议通过ip_init()函数来初始化。下面是ip_init()函数完成的主要任务:

- 通过dev_add_pack()函数注册IP包的处理函数。这个处理函数为ip_rcv()。
- 初始化路由子系统，包括与协议独立的cache。
- 初始化用来管理IP peers的基础结构。

ip_init()在启动时被inet_init()函数调用，inet_init()函数初始化了与IPv4相关的子系统，包括L4协议。

19.2.2. Interaction with Netfilter

防火墙被加载在内核的点有:

- 包接收
- 包转发(路由决定前)
- 包转发(路由决定后)
- 包发送

在上面列出的每一种情况中，实现这些操作的函数被分割为2个部分，通常被称为do_something和do_something_finish(在某些情况下，名字是do_something和do_something2)。do_something仅仅包含了合法性检测和某些管理。真正的工作是在do_something_finish或do_something2中完成。do_something结束时会调用Netfilter函数NF_HOOK，它的参数有: hook函数所在的调用点(例如包接收时)，要被执行的回调函数。如果没有规则应用或简单的指示"向前进"，函数do_something_finish被执行。

NF_HOOK的输出函数可以是下列之一:

- do_something_finish的输出值。
- 如果因为过滤器的原因而使skb被丢弃了，返回-EPERM。
- 如果没有足够的内存来执行过滤操作，返回-ENOMEM。

19.2.3. Interaction with the Routing Subsystem

IP层需要与路由表在多个地方发生交互，例如在接收合传输一个数据包时。下面是三个被IP层用来与路由表交互的函数:

- ip_rout_input:决定输入数据包的目的地址。数据包可以被本地提交，转发或丢弃。
- ip_rout_output_flow:发送数据包时使用。这个函数返回下一跳网关和要使用的出口设备。
- dst_pmtu:给定一个路由表cache entry，返回与之相联系的Path Maximum Transmission(PMTU)。

ip_route_xxx函数，作用在了路由表中，这些函数是通过下列成员来做决定的:

- 目的IP地址。
- 源IP地址。
- ToS。
- 接收数据包时的接收设备。
- 允许发送数据的设备列表。

函数把路由表查询的结果保存在skb->dst中。它是一个dst_entry结构，这个结构包含了多个成员，包括input和output函数指针，用来完成数据包的接收和发送。如果查询失败，ip_route_xxx函数返回一个负值。

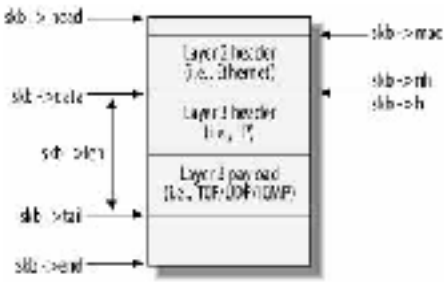
每个函数都使用了cache技术，这样得到一个目的地址相同的流的包将会很快。目的地址是这种方案最重要的一种考虑因素，他被用在cache中作为搜索的关键字。但是每个cache entry也包含了其它考虑参数，例如，cache跟踪了每个路由的PMTU。

19.2.4. Processing Input IP Packets

IP协议的处理函数主要是ip_rcv()。这个函数是一个经典的2步完成过程。绝大多数的处理发生在了ip_rcv_finish()中，它在Netfilter hook中调用。

sk_buff中的大多数成员在调用ip_rcv()前就被设置好了。下图展示了当ip_rcv()开始时sk_buff的成员的值。注意到skb->data，通常被用来指向payload，现在指向的是L3头部。

Figure 19-1. Part of sk_buff data structure at the beginning of ip_rcv



NIC的设备驱动将设置好L3协议在skb->protocol中，和数据包类型skb->pkt_type。例如以太网驱动，通过eth_type_trans()函数完成该过程。

当数据帧的L2目的地址不同与接收方网卡的地址时，skb->pkt_type被设置成PACKET_OTHERHOST。通常这些数据包被NIC自己丢弃。然而，如果网卡被设置成混杂模式，它将忽略L2地址而接收所有的数据包，并把它们传递到更高的层次。但是ip_rcv()并不关心到其它地址的数据包，只是简单的丢弃它们：

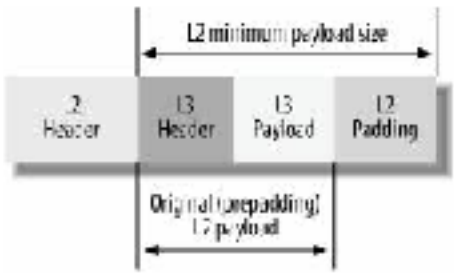
```
    if (skb->pkt_type == PACKET_OTHERHOST)
        goto drop;
```

skb_share_check()检查数据包的引用计数是否大于1，即内核其它部分是否引用了该buffer。在之前的章节中讨论的，嗅探器和其它用户可能会对数据包感兴趣，因此每个数据包包含了一个引用计数。netif_receive_skb()函数，它会在调用ip_rcv()前，将递增引用计数。如果协议处理函数看到引用计数大于1，它将创建buffer的拷贝，这样就能修改数据包。如果拷贝是必须的，然而内存分配却失败了，数据包将会被丢弃。

```
    if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
            IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
            goto out;
    }
```

pskb_may_pull()的工作是确保skb->data指向的区域包含的数据块至少有IP头那么大，因为每个IP包必须包含一个完整的IP头。

L2 padding needed to reach the minimum payload size



如果该数据包没有被丢弃，则会执行ip_rcv_finish()。

```
    return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev,
NULL, ip_rcv_finish);
```

19.2.5. The ip_rcv_finish Function

ip_rcv()仅仅只是对数据包做了些基本的合法性检测。因此当ip_rcv_finish()被调用时，它必须进行主要的处理过程，例如：

- 决定数据包是该发送到本地还是被转发。如果是第二种情况，需要找出出口设备和下一跳。
- 解析和处理一些IP选项。并不是所有的选项都是在这里被处理。

```
static inline int ip_rcv_finish(struct sk_buff *skb)
```

skb->nh在netif_receive_skb()函数中被初始化。在那个时候，L3协议还不知道，因此它被nh.raw初始化。现在，函数可以得到一个指向IP头的指针了。

```
struct net_device *dev = skb->dev;
struct iphdr *iph = skb->nh.iph;
```

skb->dst可能包含了关于路由的信息。如果路由信息还不知道，函数将询问路由子系统把数据包发送到哪里去，如果路由子系统回答目的地址不可达，数据包将被丢弃。

```
    if (skb->dst == NULL) {
         if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos,
dev))
             goto drop;
    }
```

接着函数更新一些被TC使用的统计值。

```
#ifdef CONFIG_NET_CLS_ROUTE
    if (skb->dst->tclassid) {
                    struct ip_rt_acct *st = ip_rt_acct +
256*smp_processor_id( );
        u32 idx = skb->dst->tclassid;
        st[idx&0xFF].o_packets++;
        st[idx&0xFF].o_bytes+=skb->len;
        st[(idx>>16)&0xFF].i_packets++;
        st[(idx>>16)&0xFF].i_bytes+=skb->len;
    }
#endif
```

当IP头的长度大于20字节，它意味着有选项要被处理。skb_cow()被调用，如果buffer被共享了，它将用来产生一个buffer的副本。因为可能需要修改IP头部。

```
    if (iph->ihl > 5) {
        struct ip_options *opt;
        if (skb_cow(skb, skb_headroom(skb))) {
                    IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
            goto drop;
        }
        iph = skb->nh.iph;
```

ip_options_compile()被用来IP头部中的IP选项。skb->cb中来保存一些私有信息。在这里。IP层使用它来保存IP选项解析的结果加上一些其他信息。

如果存在任何的错误选项，数据包被丢弃并且一个特别的ICMP消息发给数据包发送者，通知它这个问题。

ip_rcv_finish()最后会调用dst_input()。dst_input()中调用skb->dst->input，它根据数据包的目的地址被设置为ip_local_deliver或ip_forward。skb->dst->input最后完成了对数据包的处理。

Chapter 20. Internet Protocol Version 4 (IPv4): Forwarding and Local Delivery

20.1. Forwarding

转发也被分为两个函数：ip_forward()和ip_forward_finish()。

在这个时候，由于在ip_rcv_finish中ip_route_input的调用，sk_buff中包含了所有用来转发数据包的信息。转发由下列步骤组成：

1. 处理IP选项。
2. 根据IP头部成员，确认数据包能被转发。
3. 递减TTL，如果TTL为0则丢弃它。
4. 如果需要，处理分片。
5. 发送数据包到出口设备。

20.1.2. ip_forward Function

ip_forward()被ip_rcv_finish()函数调用，来处理所有进入的目的地不是本地的数据包。

```
int ip_forward(struct sk_buff *skb)
```

函数结束时，如果没有过滤规则禁止转发的话，就会要求Netfilter执行ip_forward_finish()。

```
       return NF_HOOK(PF_INET, NF_IP_FORWARD, skb, skb->dev, rt->u.dst.dev, ip_forward_finish);
```

20.1.3. ip_forward_finish Function

数据包最终是由dst_output发送：

```
static inline int ip_forward_finish(struct sk_buff *skb)
{
    struct ip_options * opt = &(IPCB(skb)->opt);

    IP_INC_STATS_BH(IPSTATS_MIB_OUTFORWDDATAGRAMS);

    if (unlikely(opt->optlen) {
            ip_forward_options(skb);

    return dst_output(skb);
}
```

20.1.4. dst_output Function

```
static inline int dst_output(struct sk_buff *skb)
{
        int err;

        for (;;) {
                err = skb->dst->output(&skb);
                if (likely(err = 0))
                        return err;
                if (unlikely(err != NET_XMIT_BYPASS))
                        return err;
        }
}
```

dst_output()调用了skb->dst->output，它被初始化为ip_output(单播)或ip_mc_output(多播)。

20.2. Local Delivery

在ip_rcv_finish()顶部，会调用ip_route_input()函数，如果数据包到达了它的目的主机，它将初始化skb->dst->input为ip_local_deliver()函数。而且，Netfilter被给予了最终的权力来决定do_someting函数是否允许调用对应的do_someting_finish函数来完成工作。

```
int ip_local_deliver(struct sk_buff *skb)
{
    if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
        skb = ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER);
        if (!skb)
            return 0;
    }

    return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
                   ip_local_deliver_finish);
}
```

分片重组是通过ip_defrag函数来执行的，如果完全重组了，返回指向原始数据的指针，否则返回NULL。

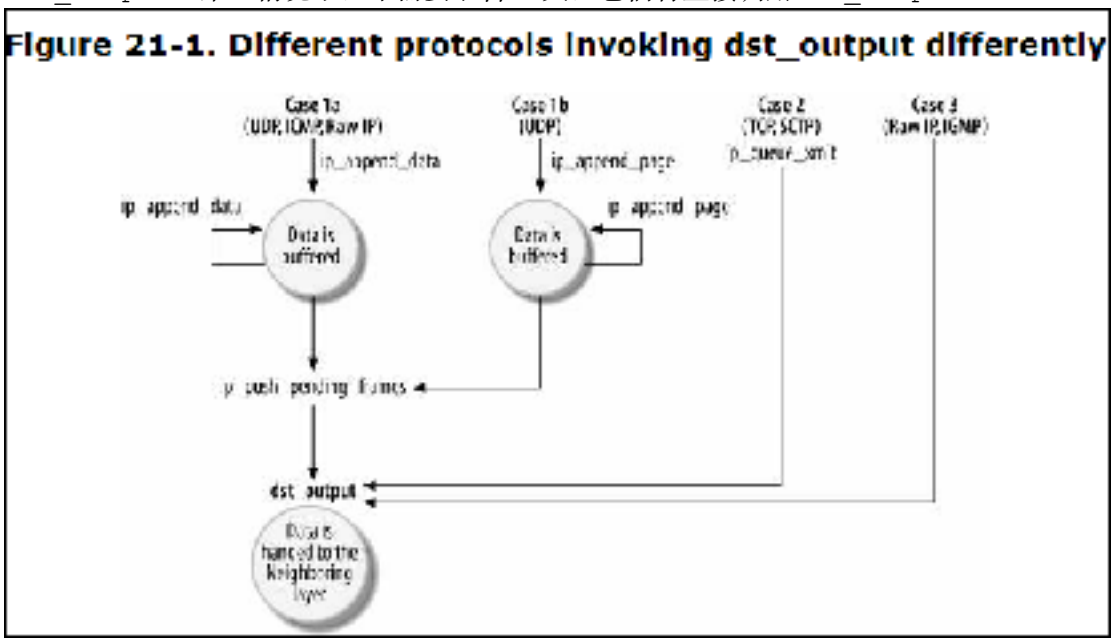
Chapter 21. Internet Protocol Version 4 (IPv4): Transmission

21.1. Key Functions That Perform Transmission

在L4协议中，TCP和SCTP将要为分片做很多准备;那么IP层工作量将减少。相反的，raw IP等把分片的工作全部交给了IP层。

下图展示了L4到L3最后步骤的发送过程。当L3完成任务后，将把数据包传给dst_output。某些情况下，不需要准备IP头，它们将直接调用dst_output。

Figure 21-1. Different protocols invoking dst_output differently



处理分片的函数集合有两种:

- ip_queue_xmit:

L4协议已经把数据分成合适的大小，IP层只需要添加一个IP头，而不需要考虑分片。

- ip_push_pending_frames and related functions:

L4协议调用该函数并不会考虑分片或者试图帮助下层分片。L4协议可以通过多次调用ip_append_data()把多次发送请求保存起来，而不用真正的发送任何数据包。

ip_append_data()并不是简单的缓存发送请求，而是透明的产生最优大小的数据分片，这样之后的IP层就能方便的进行IP分片，这大大提高了性能。

当L4协议需要刷新由ip_append_data()产生的输出队列时，协议调用ip_push_pending_frames()，它将做必要的分片并把最终产生的数据包传递给dst_output()。

ip_append_pages()和ip_append_data()类似。

其它一些函数被用在特定环境下的发送:

ip_build_and_send_pkt():被TCP用来发送SYN ACKs。

ip_send_reply():被TCP用来发送ACKs和Resets。

21.1.2. Relevant Socket Data Structures for Local Traffic

一个BSD socket在Linux中是被一个socket结构的实例代表。这个结构包含了一个指向sock数据结构的指针，在sock数据结构中保存了网络层的信息。sock数据结构实际上是一个更大的与特定协议相关的数据结构中的一个部分；对PF_INET sockets它是inet_sock。inet_sock的第一个成员是一个sock实例，其它部分保存了PF_INET私有信息。

```
struct inet_sock {
    struct sock sk;
    ... ... ...
    struct {
        ... ... ...
    } cork;
}
```

inet_sock->cork在ip_append_data()和ip_append_page()中扮演了一个关键角色：它保存了这两个函数进行数据分片所需的环境信息。

只要发送的数据包是本地产生的，每个sk_buff将通过skb->sk联系到它的sock实例。

21.1.3. The ip_queue_xmit Function

ip_queue_xmit()函数被TCP和SCTP协议使用。

```
int ip_queue_xmit(struct sk_buff *skb, int ipfragok)
```

skb:记住ip_queue_xmit()被用在处理本地产生的数据包;转发的数据包没有相关联的socket。

ipfragok:主要被SCTP使用的标志位，表明了分片是否被允许。

21.1.3.1. Setting the route

如果buffer已经被分配了合适的路由信息(skb->dst)，那么就不需要查询路由表。这种情况可能发生在SCTP协议中:

```
rt = (struct rtable *) skb->dst;
if (rt != NULL)
    goto packet_routed;
```

在其它情况下，ip_queue_xmit()检测是否有一个路由cache在socket数据结构中，如果有，确保它仍然是合法的。

```
        rt = (struct rtable *)_ _sk_dst_check(sk, 0);
```

如果socket还没有为数据包cache一个路由信息，或者此时已经失效了，ip_queue_xmit()需要查找一个新的路由，通过ip_route_output_flow()并把结果保存在sk数据结构中。

```
        if (rt == NULL) {
            u32 daddr;

            daddr = inet->daddr;
            if(opt && opt->srr)
                daddr = opt->faddr;

            {
                struct flowi fl = { .oif = sk->sk_bound_dev_if,
                                    .nl_u = { .ip4_u =
                                              { .daddr = daddr,
                                                .saddr = inet->saddr,
                                                .tos = RT_CONN_FLAGS(sk)
} },
                                  .proto = sk->sk_protocol,
                                  .uli_u = { .ports =
                                             { .sport = inet->sport,
                                               .dport = inet->dport } }
};

                if (ip_route_output_flow(&rt, &fl, sk, 0))
                    goto no_route;
            }
            _ _sk_dst_set(sk, &rt->u.dst);
            tcp_v4_setup_caps(sk, &rt->u.dst);
        }
```

21.1.3.2. Building the IP header

最终，Netfilter被调用，来决定是否把该数据包继续发送。

```
          return NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt-
>u.dst.dev, dst_output);
```

21.1.4. The ip_append_data Function

这个函数被L4协议使用，用来buffer要被发送的数据。这个函数并不发送数据，只是把数据放在合适大小的buffer中，之后再形成分片和发送。那么，它并不创建和控制任何的IP头部。

如果L4层想要快速的响应时间，它应该在ip_append_data()之后立即调用ip_push_pending_frames()。但是，L4层尽可能的buffer后再一次性发送将更加有效。

ip_append_data()的主要任务是:

- 把L4层的数据组织进buffer，这个buffer的大小将使IP分片的处理变得容易。
- 优化内存分配，需要考虑从上层来的信息和出口设备的能力。特别的:
  1. 如果上层通知将会有更多的发送请求在短时间内紧跟而来(通过MSG_MORE标志位)，那么应该分配一个大的buffer。

2. 如果出口设备支持Scatter/Gather I/O(NETIF_F_SG)，分片将会以页面来安排，这样可以优化内存处理。

- 考虑L4校验和。

```
int ip_append_data(struct sock *sk,
                   int getfrag(void *from, char *to, int offset,
int len,
                               int odd, struct sk_buff *skb),
                   void *from, int length, int transhdrlen,
                   struct ipcm_cookie *ipc, struct rtable *rt,
                   unsigned int flags)
```

下面是输入参数的含义：

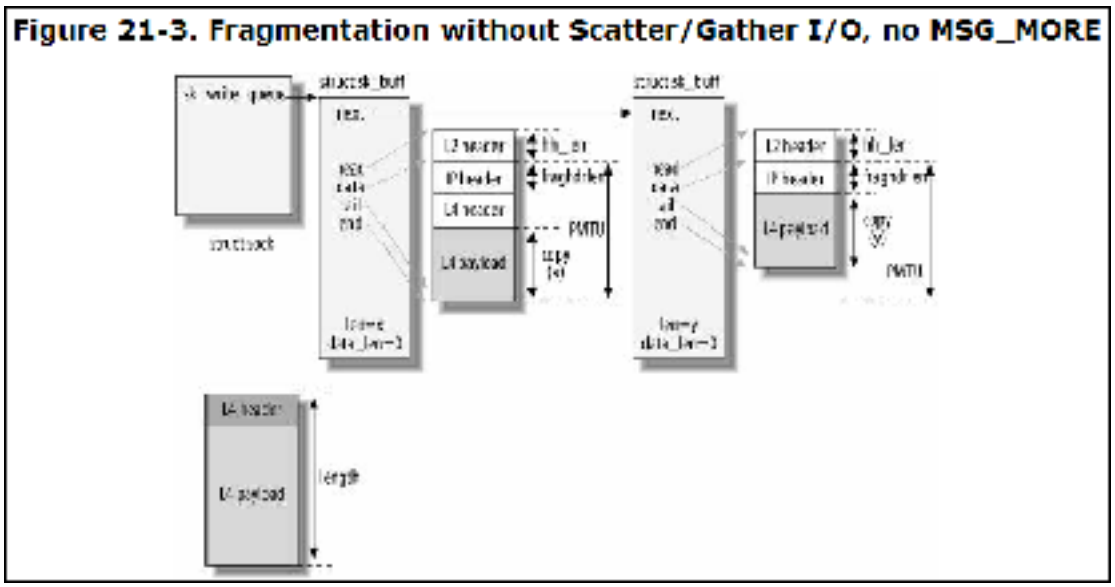
1. sk:包含了一些之后用于填充IP头(被ip_push_pending_frames()函数)的参数(例如：IP选项)。
2. form:指向L4层打算发送的数据的指针。这个指针可以是内核空间的，也可以是用户空间的，getfrag()函数的工作就是正确的处理它。
3. getfrag:用来从L4层拷贝数据到将要创建的数据分片中的函数。
4. length:要被发送的数据的总数(包括L4头部和payload)。
5. transhdrlen:L4头部大小。
6. ipc:转发数据包所需的信息。
7. rt:与数据包关联的路由表cache entry。
8. flags:MSG_XXX标志。

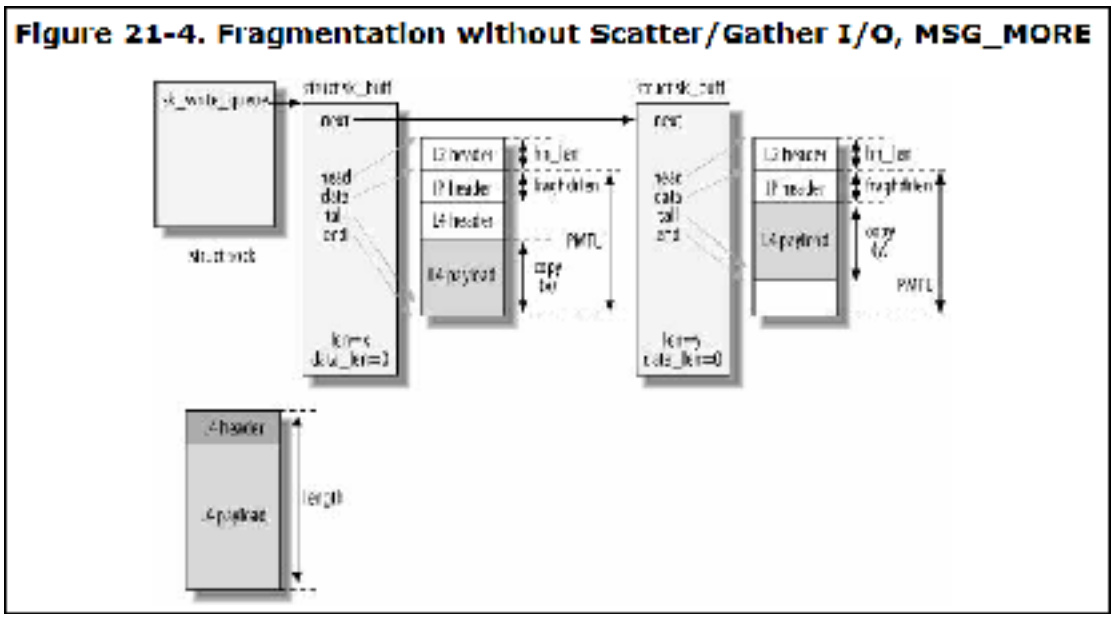
21.1.4.1. Basic memory allocation and buffer organization for ip_append_data

ip_append_data()能创建一个或多个sk_buff实例，每个实例代表了一个不同的IP包(或IP分片)。

L4头部在ip_push_pending_frames()函数中调用的函数(例:udp_push_pending_frames())填充。
L3头部(包括IP选项)将在ip_push_pending_frames()中填充。

Figure 21-3. Fragmentation without Scatter/Gather I/O, no MSG_MORE





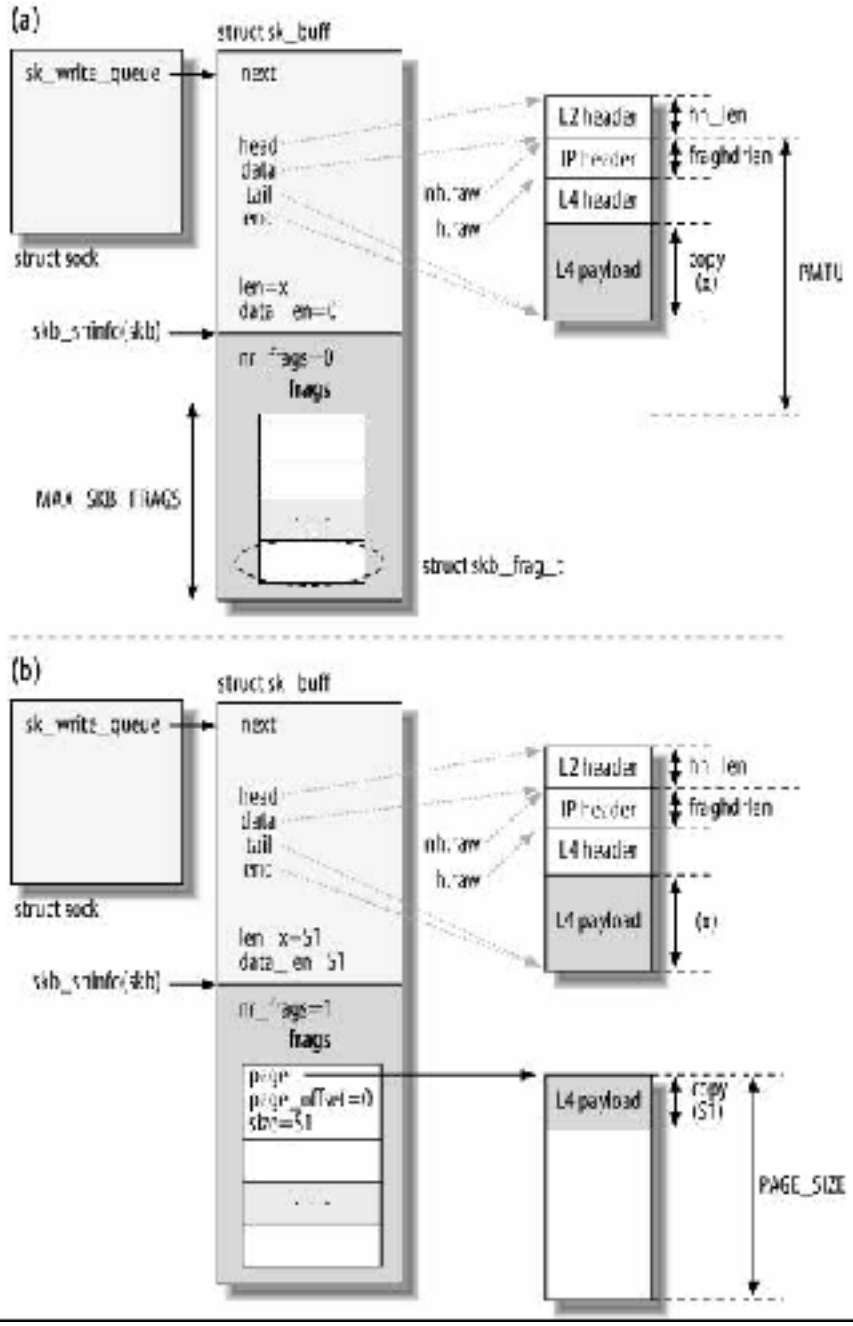
21.1.4.2. Memory allocation and buffer organization for ip_append_data with Scatter Gather I/O

当一个上层协议产生了许多小的数据项，L4层把它们存放在内核内存的不同buffer中。L3层接着会把这些项发送到一个IP包中。如果没有Scatter/Gather I/O，L3层不得不把数据拷贝到一个新的buffer中。如果设备支持Scatter/Gather I/O，数据直到离开主机前，其位置都不会发生改变。

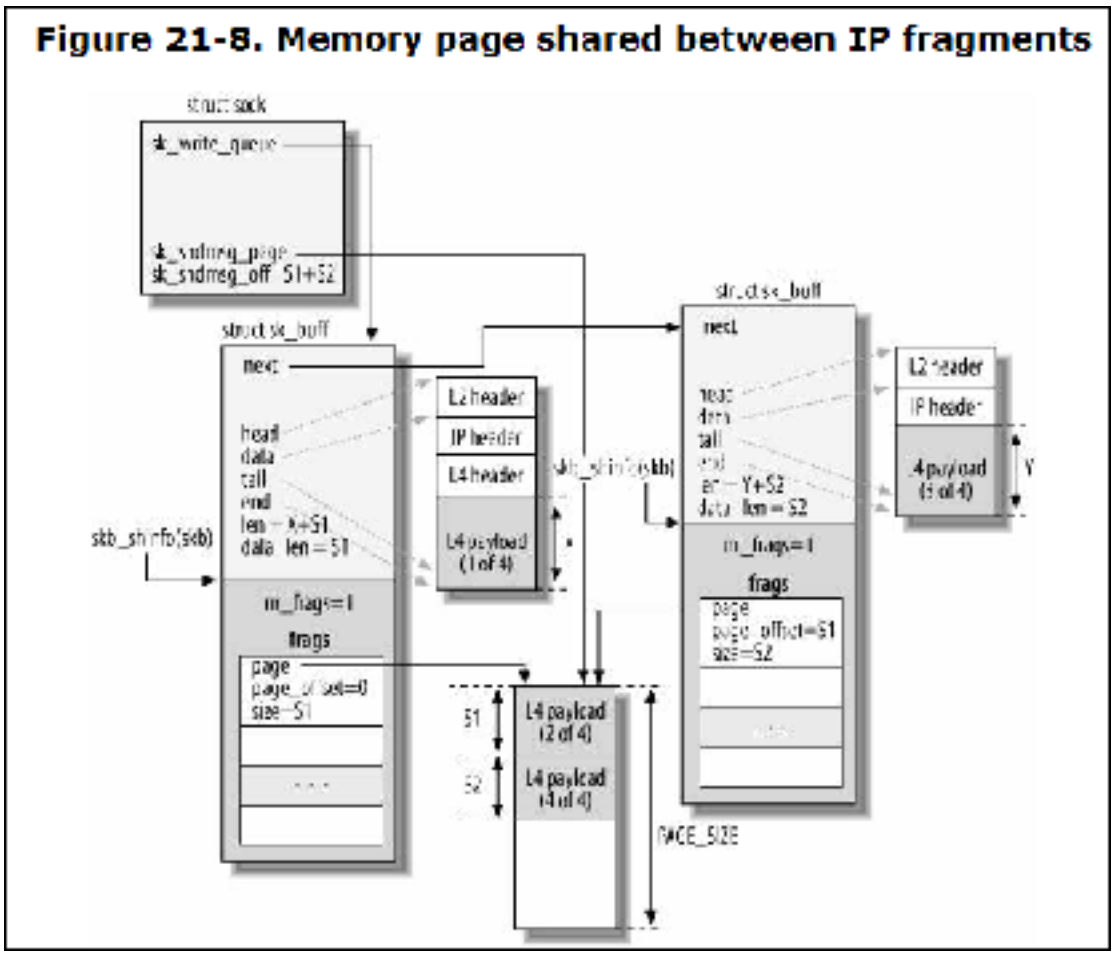
如果Scatter/Gather I/O被使用了，skb->data指向的内存数据块仅在首次被使用。接下来的数据块被拷贝进内存页面。Figures 21-5和21-6比较了被ip_append_data在第二次调用时接收的数据是如何保存的，此时Scatter/Gather I/O被使用：

- Figures 21-5(a)展示了在第一次调用后内存使用状况，Figures 21-5(b)展示了第二次调用后的内存状况。

Figure 21-5. ip_append_data with Scatter/Gather I/O



sock->sk_sndmsg_page指向指向当前socket的发送缓冲队列的最后一个skb的分片数据的最后一页，sock->sk_sndmsg_off表示最后一页分片数据的页内偏移。我们开始尝试把数据追加到最后一页中，如果不行，则分配新页，然后向新页拷贝数据，并更新sk_sndmsg_page和sk_sndmsg_off的值。



21.1.4.3. Key routines for handling fragmented buffers

- `skb_is_nonlinear()`:如果`buffer`是分片的，返回`true`。(例如`skb->data_len`为非`NULL`)。
- `skb_headlen()`:给定一个分片的`buffer`，返回在主`buffer`中的数据总数(不会计算`frags`中分片和`frag_list`链表)。
- `skb_pagelen()`:`skb_headlen+frags`分片中的数据，不考虑链入`frag_list`中的`buffer`。

21.1.4.4. Further handling of the buffers

只要`ip_append_data()`分配了一个新的`sk_buff`结构来处理一个新的数据分片(它将成为一个新的`IP`分片)，函数把这个分片加入一个名为`sw_write_queue`的队列，这个队列与`socket`结构中。随后的函数只需要把`IP`头加在数据分片上，并把它们传递到`L2`层(`dst_output`程序中)。

`sk_write_queue`链表是一个`FIFO`队列:

- 新的分片被添加到链表末尾。
- 仅当`sk_write_queue`中最后一个分片的大小到达最大值(`maxfraglen`)时，一个新的的分片才被创建并添加到链表中。

21.1.4.5. Setting the context

仅在第一个分片中需要包含传输层头部和可选的外部头部。因此，`transhdrlen`和`exthdrlen`在第一个分片之后的分片中都为`0`。

21.1.4.6. Getting ready for fragment generation

```
        hh_len = LL_RESERVED_SPACE(rt->u.dst.dev);
        fragheaderlen = sizeof(struct iphdr) + (opt ? opt->optlen
: 0);
                    maxfraglen = ((mtu - fragheaderlen) & ~7) +
fragheaderlen;
```

hh_len是L2头部长度。Fragheaderlen是IP头部大小，包括IP选项。maxfraglen是IP分片根据PMTU计算出来的最大大小。

21.1.4.7. Copying data into the fragments: getfrag

getfrag()接受4个输入参数(from，to，offset，len)，简单的把len字节的数据从from拷贝到to+offset中。同时，它还实现了L4校验和:当拷贝数据到内核buffer中时，它会根据skb->ip_summed来更新skb->csum。

21.1.4.8. Buffer allocation

ip_append_data()选择分配的buffer大小是根据:

- Single transmission versus multiple transmissions

如果ip_append_data()被告知之后将会有其它发送请求到达(如果MSG_MORE被设置了)，它将分配一个更大的buffer，以致之后需要发送的数据能被合并进同一个buffer中。

- Scatter/Gather I/O

如果设备能处理Scatter/Gather I/O，碎片存放在内存页面中将会更加的有效。

下面的代码根据上面提到的两个因素，来确定需要分配的buffer大小(alloclen)。

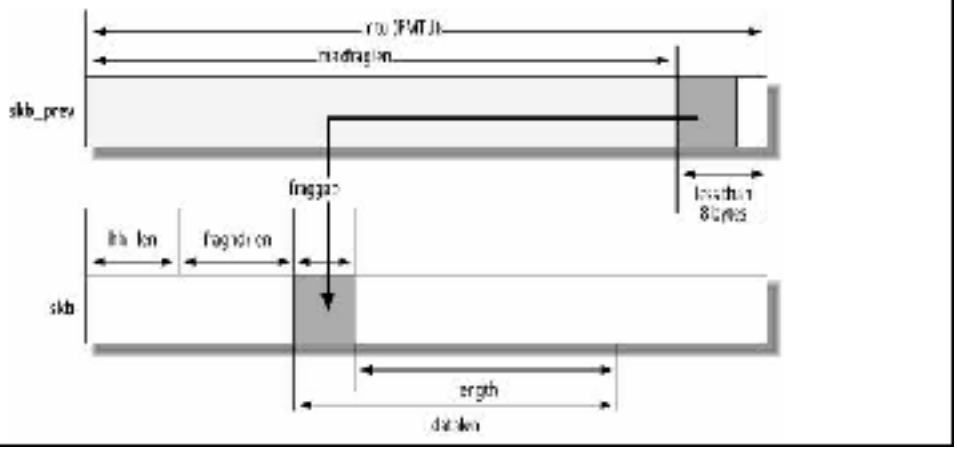
```
                if ((flags & MSG_MORE) &&
                    !(rt->u.dst.dev->features&NETIF_F_SG))
                    alloclen = mtu;
                else
                    alloclen = datalen + fragheaderlen;

                if (datalen == length)
                        alloclen += rt->u.dst.trailer_len;
```

datalen是将要拷贝到buffer中的数据总数。它的值是基于3个因素:剩下的数据长度(length)，适合一个分片的最大数据总数(fraghdrlen)，an optional carry from the previous buffer (fraggap)。

除了最后一个buffer(它含有最后一个IP分片)以外，所有的分片的payload必须是8字节对齐的。因此，当内核分配一个新的不是用在最后一个分片的buffer时，它可能需要把一些数据(0~7字节)从前一个buffer中移动到新分配的buffer的头部。换句话说，fraggap为0，除非下列情况发生:

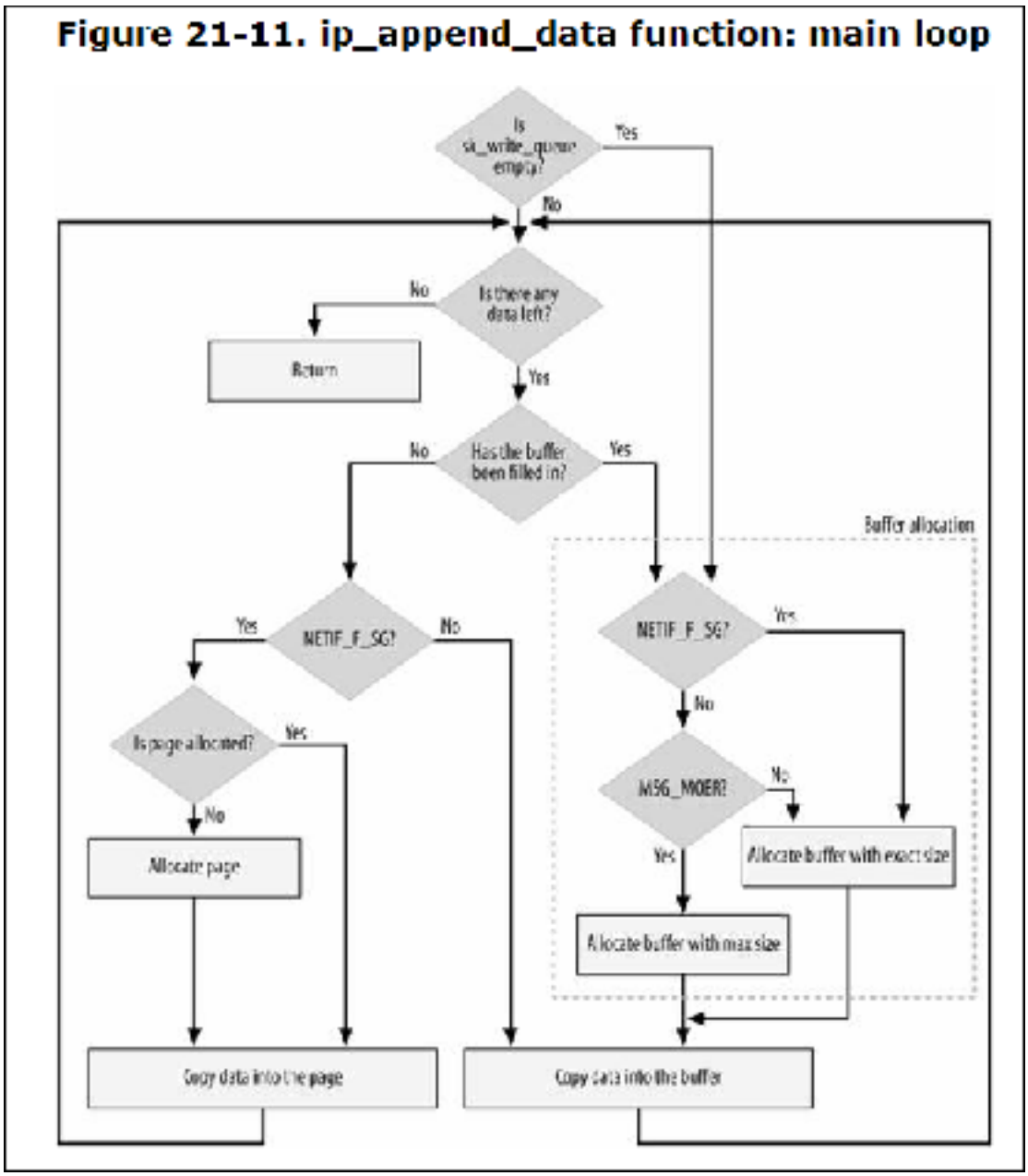
- PMTU不是8字节对齐。
- 当前IP分片还没到达PMTU。
- The size of the current IP fragment has passed the highest multiple of eight bytes that is less than or equal to the PMTU.

Figure 21-10. Respecting the 8-byte boundary rule on IP fragments



21.1.4.9. Main loop

Figure 21-11. ip_append_data function: main loop



初始的，length的值代表了ip_append_data()的调用者希望发送的数据总数。然而，一旦循环进入了，它的值代表着要剩下的将要被处理的数据。当length变为0时循环将结束。

我们已经知道了MSG_MORE指示了L4层期望更多的数据，NETIF_F_SG指示了设备是否支持Scatter/Gather I/O。

ip_append_queue()在下列条件发生时，分配一个新的sk_buff结构并把它加入sk_warite_queue队列：

- sk_warite_queue为空。
- sk_warite_queue的最后一个元素已经被完全填满。

在循环中，首先初始化copy为当前IP分片中剩下的空间总数：mtu-skb->len。如果剩下要被添加的数据(length)大于空闲空间总数，那么需要更多的IP分片。在这种情况下，copy被更新。因为要8字节对齐，copy会比最近的8字节边界值小。

21.1.5. The ip_append_page Function

ip_append_data()通过void *指针来接收数据的位置，ip_append_page()接收数据的位置是通过指向内存页面的指针和偏移，它将直接用来初始化frag中某个entry。

当添加一个新的分片到页面中时，ip_append_page首先尝试把新分片与之前的分片合并。为了实现这个，它先通过skb_can_coalesce()来检测是否可以合并。如果合并是允许的，需要做的事情就是更新前一个分片的长度来包含新的数据。

当合并是不被允许的，函数通过skb_fill_page_desc()来初始化新分片。

21.1.6. The ip_push_pending_frames Function

当L4层决定把通过ip_append_data()和ip_append_page()函数挂入sw_write_queue的分片发送出去时，只需简单的调用ip_push_pending_frames()。

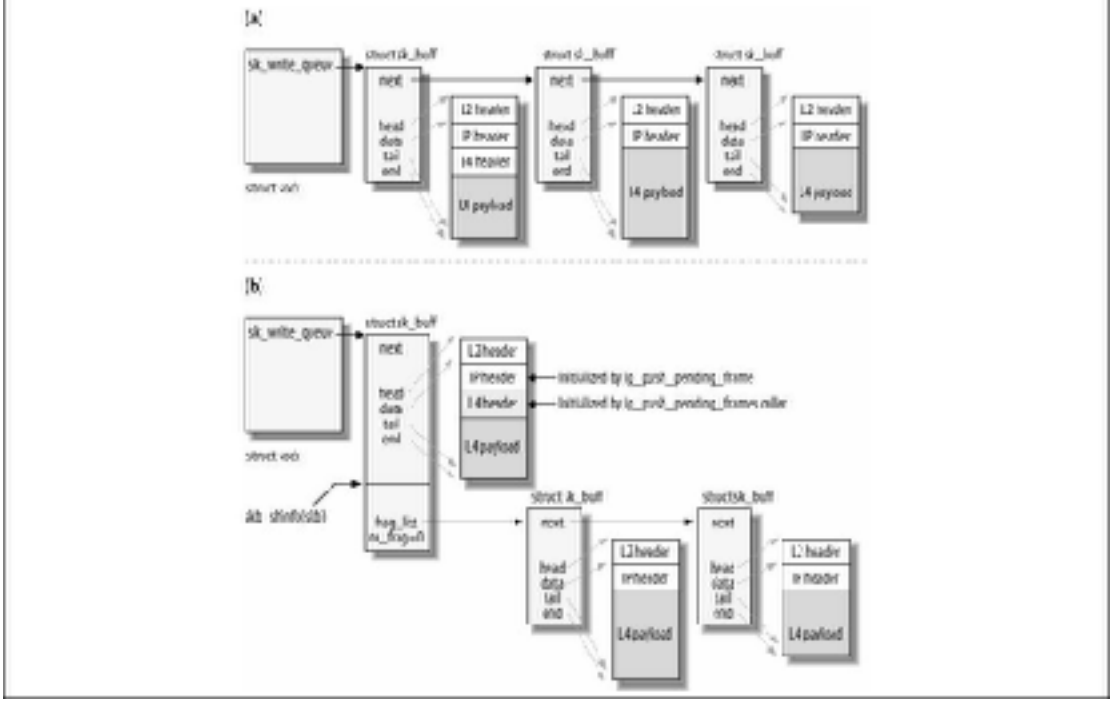
```
int ip_push_pending_frames(struct sock *sk)
```

函数会把第一个分片后的所有分片挂到第一个分片的frag_list中，并更新len和data_len来包括所有的分片。现在，数据已经离开了L4层，进入了IP层的处理范围。

注意到，nr_frags反映的是Scatter/Gather I/O buffer的数量，而不是IP分片的数量。

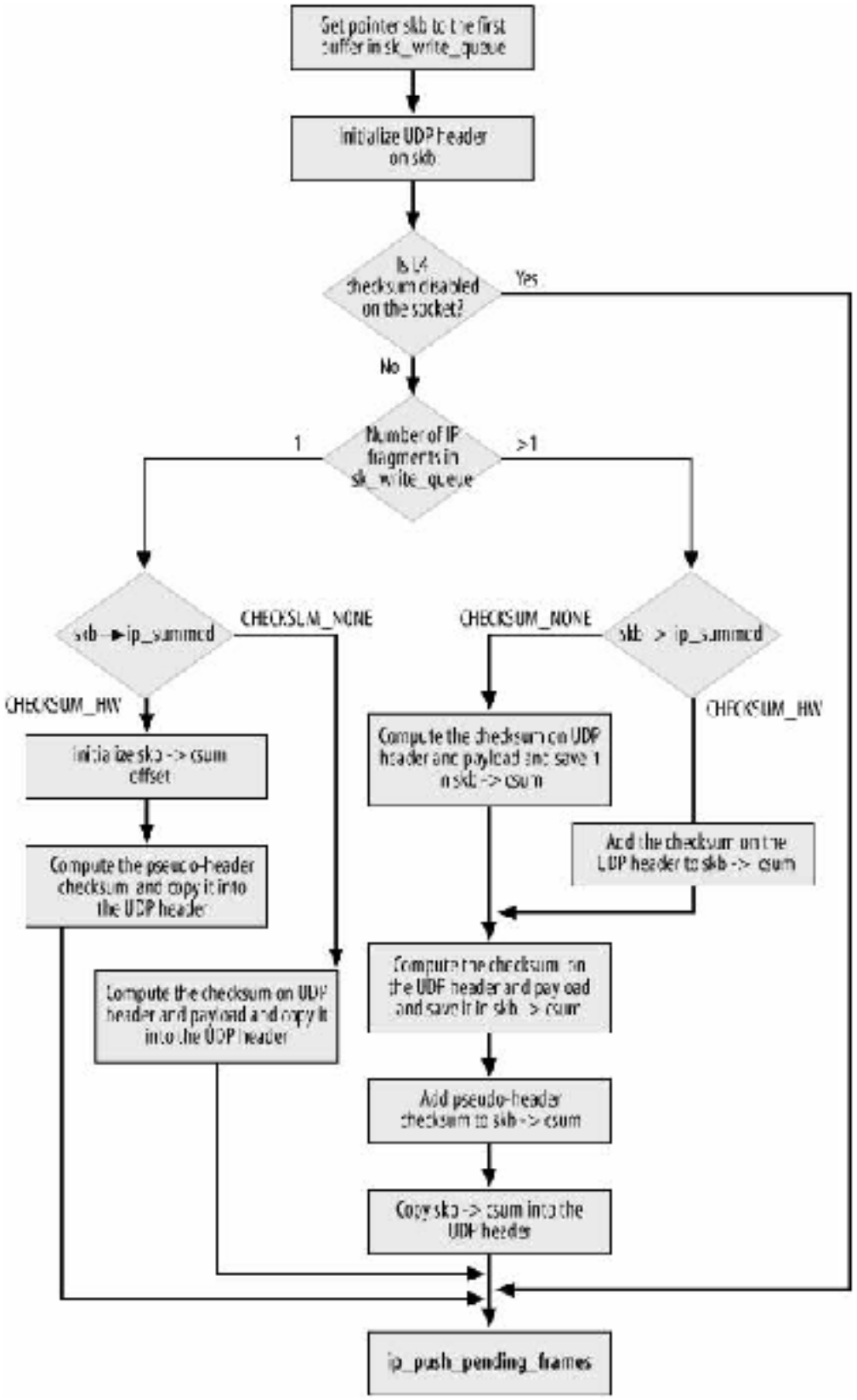
如果有多个分片，只有第一个分片被函数填充IP头部；其它分片的IP头部将在之后被填充。

Figure 21-12. (a) Before and (b) after removing buffers from the sk_write_queue queue



21.1.7. Putting Together the Transmission Functions

Figure 21-13. udp_push_pending_frames function



Chapter 22. Internet Protocol Version 4 (IPv4): Handling Fragmentation

22.1. IP Fragmentation

在之前的章节中看到的，dst_output()函数被本地产生的数据包和转发的数据包调用，其中调用的ip_fragment()函数能够:

1. Big chunks of data that need to be split into smaller parts.
分配新buffers，把原来大数据块分割拷贝到小buffer中去。
2. A list or array of data fragments that do not need to be fragmented further.
如果原来的buffers有足够的空间添加L3和L2头部，函数不需要进行内存拷贝。IP层只需要把IP头添加到每个分片中并计算校验和。

分片能被两种方法实现:快速方法和慢速方法。这两种方法都是通过ip_fragment()实现的。先复习下IP分片的主要任务:

1. 把L3的payload分割成小块，使之适合MTU。这个过程可能会有一些内存拷贝。最后一个分片的大小可能不同于其它分片。
2. 初始化每个分片的IP头，考虑到不是所有的IP选项都要复制到每个分片中。ip_options_fragment()实现了该工作。
3. 计算IP校验和。每个分片有一个不同的IP头，因此校验和必须重新计算。
4. 询问Netfilter，是否允许完成发送。
5. 更新必要的内核和SNMP统计信息。

22.1.1. Functions Involved with IP Fragmentation

22.1.2. The ip_fragment Function

```
int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))
```

输入参数:

skb:包含需要被分片IP包。包中已经包含了一个被初始化的IP头，它将被选择并复制进所有的分片。

output:用于发送分片的函数。

当ip_fragment接收到的sk_buff的数据已经分片了，将使用快速分片。这种情况发生在被L4层协议使用ip_append_data和ip_push_pending_frames产生的数据包。也有可能是通过ip_queue_xmit函数产生的数据包。

慢速分片用在其他情况:

- 转发数据包。
- 本地产生的流量，但在到达dst_output之前还没有被分片。
- 因为合法性检测，导致快速分片被禁止时。

22.1.3. Slow Fragmentation

在进入循环前，函数需要初始化一些局部变量。

ptr指向数据包的偏移，代表了分片；他将随着分片进行而移动。Left被初始化为IP包的长度。

22.1.4. Fast Fragmentation

当ip_fragment发现frag_list指针非NULL时，将使用快速分片。

22.2. IP Defragmentation

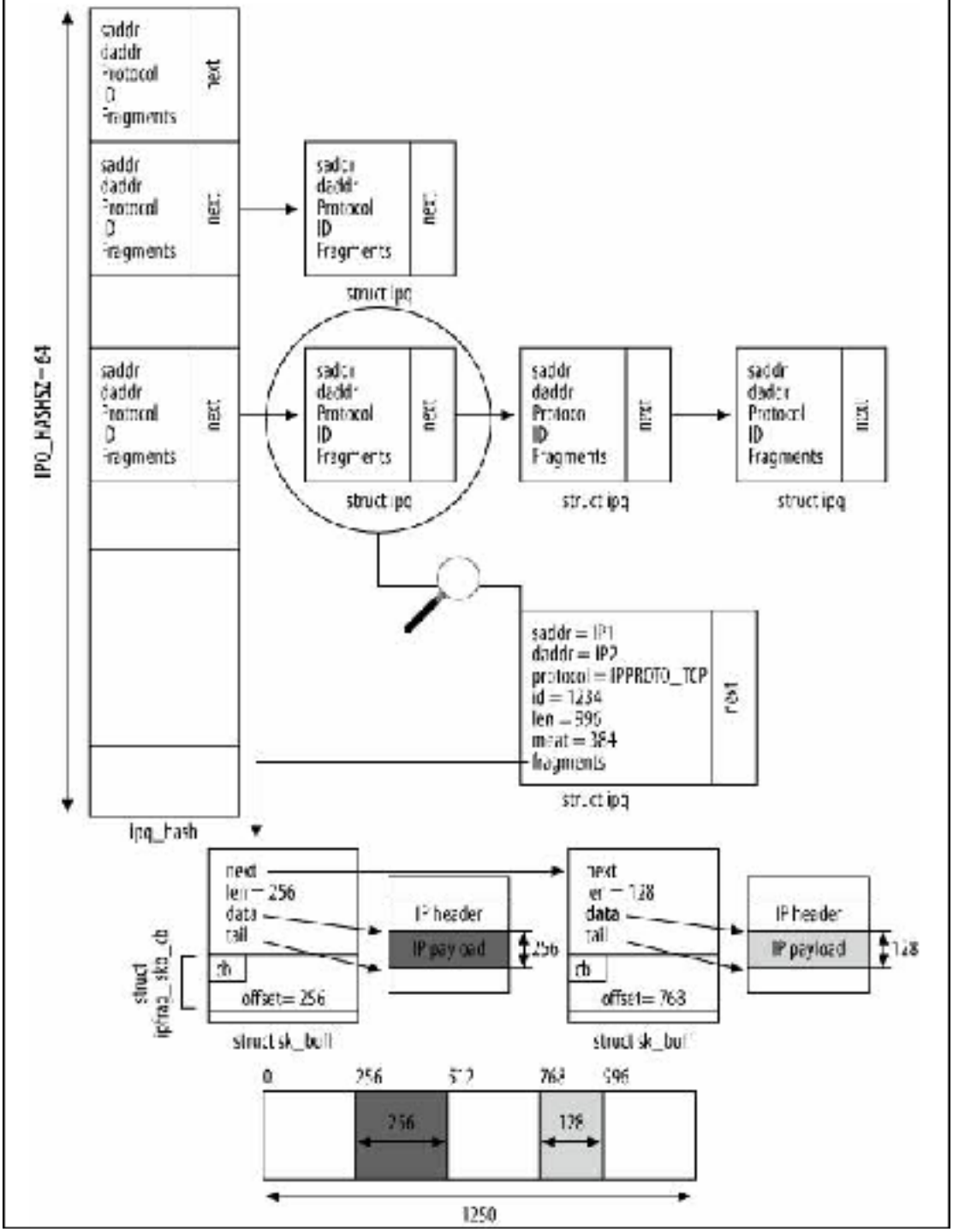
22.2.1. Organization of the IP Fragments Hash Table

当IP分片接收到时，它们被组织称一个哈希表，元素为struct ipq。

```
#define IPQ_HASHSZ    64
static struct ipq *ipq_hash[IPQ_HASHSZ];
```

在IP分片重组中，IP层使用sk_buff->cb来保存ipfrag_skb_cb数据结构，它包含了一个inet_skb_parm结构，被用来保存IP选项和标志。

Figure 22-1. Structure used to store IP fragments



22.2.2. Key Issues in Defragmentation

分片重组将面临的限制：

- 分片必须保存在内核内存中，直到他们被网络子系统完全处理完，内存是很宝贵的，所以必须有一种方法来限制内存的使用。
- 一个哈希表会变得不平衡，攻击者会利用哈希算法来使之变得不平衡，从而降低处理速度。Linux的哈希函数使用了一个额外的随机组件来确保哈希函数的输出较难被预测。

- 网络使用的是不可靠的媒介，因此分片可能会被丢失。因此，IP层必须对每个数据包维护一个定时器。校验和也必须有能力尽可能的检测出数据包的损坏。
- 如果源主机在给定时间内没有接收到数据包的确认，并且它的传输层实现了流量控制，它将重传数据。因此，同一个IP包的多个重复的分片可能在目的端被收到。更复杂的是，分片的边界可能不同。Linux内核对重传的数据包，给了它一个新的IP ID，这将帮助减少这种问题的发生的概率。不幸的是，IP ID在高速网络中会很快重复，因此，不同IP包的分片混杂的问题仍将存在。

22.2.3. Functions Involved with Defragmentation

分片重组的主要函数是ip_defrag()。它在每次调用时接收一个分片作为输入，并尝试着把它加入到合适的数据包中。函数仅在数据包被完全重组后才返回成功。

下面是被ip_defrag()使用的支撑程序：

- ip_evictor()：一个一个移除不完全数据包的ipq结构，直到分片使用的内存总量低于sysctl_ipfrag_low_thresh。
- ip_find()：查找与正在被处理的分片相联系的数据包。查找基于IP头的4个成员：ID，源和目的IP，L4协议。
- ip_frag_queue()：把一个给定分片加入到同一个IP包的分片的链表中。
- ip_frag_reasm()：一旦所有分片都接收到了，重组分片为一个IP包。

22.2.4. New ipq Instance Initialization

ip_defrag()的首要任务是搜寻分片需要添加到的数据包。为了找到这个数据包，函数调用ip_find()。如果分片恰巧是第一次出现，ip_find()会失败。在这种情况下，ip_find()通过ip_frag_create()函数创建一个新的ipq实例。无论是找到了结构还是新创建结构，ip_find()返回一个指向它的指针。ip_defrag()使用这个指针来插入新的分片到ipq结构中。

22.2.6.1. Handling overlaps

现在是处理数据帧重叠问题的时候了。这分为2个步骤：首先，函数处理偏移量较小的分片，接着处理较大偏移的分片。

如果新数据帧的prev!=NULL，这就意味着我们已经接收到了一个分片，它带有一个较小的offset。我们需要把新插入的分片与前一个分片的重叠部分移除。

```
        if (prev) {
            int i = (FRAG_CB(prev)->offset + prev->len) - offset;

            if (i > 0) {
                offset += i;
                if (end <= offset)
                    goto err;
                if (!pskb_pull(skb, i))
                    goto err;
                if (skb->ip_summed != CHECKSUM_UNNECESSARY)
                    skb->ip_summed = CHECKSUM_NONE;
            }
```

处理完前面的分片后，现在开始处理后一个分片可能发生的重叠。可能有两种情况：

- 一个或多个后面的分片被完全包含在新数据帧中。
- 一个后面的分片部分的与新数据帧重叠。

此时，next指针指向了offset值大于新数据帧的offset值的第一个分片。

```
        while (next && FRAG_CB(next)->offset < end) {
            int i = end - FRAG_CB(next)->offset;
```

如果重叠部分的大小比新分片的大小要小，这意味着函数到达了链表中最后一个重叠分片，并且接下来需要移除重叠部分。新分片随后将被加入。注意到，如果重叠部分在前一个分片中，函数需要从新分片中移除数据，但是当重叠部分在后一个分片中，函数移除后一个分片中的重叠数据。

```
            if (i < next->len) {
                if (!pskb_pull(next, i))
                    goto err;
                FRAG_CB(next)->offset += i;
                qp->meat -= i;
                if (next->ip_summed != CHECKSUM_UNNECESSARY)
                    next->ip_summed = CHECKSUM_NONE;
                break;
            } else {
```

如果被移除的分片是链表头，那么头指针必须更新：

```
                struct sk_buff *free_it = next;
                next = next->next;

                if (prev)
                    prev->next = next;
                else
                    qp->fragments = next;

                qp->meat -= free_it->len;
                frag_kfree_skb(free_it, NULL);
            }
        }
```

最后，在解析了所有可能发生的重叠情况后，函数把新分片插入链表并且更新qp中的一些参数。Qp特被移动到ipq_lru_list链表的尾部。

```
        qp->stamp = skb->stamp;
        qp->meat += skb->len;
        atomic_add(skb->truesize, &ip_frag_mem);

        if (offset == 0)
            qp->last_in |= FIRST_IN;
```

22.2.7. Garbage Collection

内核实现了2种IP分片的垃圾回收机制：

- 系统内存使用上限。
- 碎片重组定时器。

为了阻止IP分片重组系统的内存使用过度，内存使用限制被保存在sysctl_ipfrag_high_thresh变量。全局变量ip_frag_mem变量代表了分片当前内存的使用量。它在每次新分片从ipq_hash表中加入或移除时更新。当系统上限到达时，ip_evictor()被调用来释放一些内存。

```
        if (atomic_read(&ip_frag_mem) >
sysctl_ipfrag_high_thresh)
            ip_evictor( );
```

当一个IP包的分片首次加入ipq_hash表时，一个定时器启动。这个定时器用来丢弃该不

完整的包的所有分片。

22.2.8. Hash Table Reorganization

重新组织分片是通过一个定时器，缺省的每过10分钟定时器超时。

当定时器超时时，将执行函数ipfrag_secret_rebuild()。它每次执行时，都将通过get_random_bytes()产生一个随机数，并保存在全局变量ipfrag_hash_rnd中，这个值将用在ipqhashfn()哈希函数。接着，哈希表中的元素一个一个出列，重新哈希再插入。

ipq结构的重新组织不会影响ipq_lru_list链表。

Chapter 24. Layer Four Protocol and Raw IP Handling

24.2. L4 Protocol Registration

IPv4协议之上的L4协议使用net_protocol数据结构。它由3个成员组成：
int (*handler)(struct sk_buff *skb)
被L4协议注册的用来处理进入的数据包的函数。
void (*err_handler)(struct sk_buff *skb, u32 info)
被ICMP协议处理函数使用，来通知L4协议一个ICMP UNREACHABLE信息。
int no_policy
1代表没有必要对该协议检测IPsec策略

24.2.1. Registration: inet_add_protocol and inet_del_protocol

当协议是通过模块来实现时，协议的注册使用inet_add_protocol()函数，协议的解除注册时通过inet_del_protocol()函数。

所有的L4协议的inet_protocol结构的注册是被内核插入表inet_protos。

24.3. L3 to L4 Delivery: ip_local_deliver_finish

ip_local_deliver_finish()函数的主要任务是根据输入的IP包头的protocol成员找到正确的协议处理函数，以及把数据包交给处理函数。