

Technische Universität München
Institut für Informatik – Lehrstuhl Prof. Feldmann

Study Project

Packet Capturing Using the Linux Netfilter Framework

Ivan Pronchev
(pronchev@in.tum.de)

Aufgabensteller: Prof. Anja Feldmann, Ph.D.
Betreuer: Dipl.-Inf. Fabian Schneider
Dipl.-Inf. Holger Dreger

Abgabedatum: 10. July 2006

Contents

1. Introduction	1
1.1 Motivation	1
1.2 Structure of this Paper	1
2. Background	2
2.1 Interrupts	2
2.2 Receive Livelock	2
2.3 Device Polling	2
2.4 Processing Multiple Frames During an Interrupt (NAPI)	2
2.5 Overview of the TCP/IP Kernel Stack	3
3. Main Data Structures	4
3.1 Linux Socket Buffer	4
3.2 Linux Network Device	7
3.3 Linux Socket Buffer Queue	9
3.4 Managing ingress/egress traffic	10
3.5 Protocol Handler	11
4. NAPI/Non-NAPI Frame Reception	13
4.1 net_dev_init	13
4.2 netif_rx	14
4.3 net_rx_action	16
4.4 process_backlog	17
4.5 netif_receive_skb	17
4.6 NAPI Frame Reception	19
5. Linux Netfilter Framework	21
5.1 nf_hook_ops	21
5.2 Netfilter Hooks in IPv4 Protocol Stack	22
5.3 Netfilter Framework in Details	23
6. Kernel Sniffer	26
6.1 Accessing File from Kernel Module	26
6.2 Design	27
6.3 Benchmarks	29
7. Summary	30
Bibliography	31

1. Introduction

Packet sniffers (also known as network analyzers) are software tools (usually) or hardware devices designed to log traffic passing over a network. As data streams travel back and forth over the network, a sniffer captures packets and eventually analyzes the contents. The following document covers functions and data structures from kernel 2.6.14 used to design a kernel sniffer. They also apply to any 2.6 kernel with slightly differences.

1.1 Motivation

The goal of the project is to build a kernel module as a sniffer using the Linux netfilter framework and to compare the capturing rates between the module and an existing capturing library (libpcap). The implementation of a sniffer as a kernel module allows increasing of the capturing rates, because it saves expensive copy operations from/to kernelspace in normal userlevel implementations and also reduces the number of context switches between kernel- and userspace. During the development of the kernel module different problems have occurred, because of the fact that a typical userspace application is moved to the kernelspace. The project deals not only with the networking of the kernel but also with other subsystems such as the memory management or the filesystem management which makes it interesting, exciting and sophisticated.

1.2 Structure of this Paper

The next chapter (2) covers some basic terms that are not directly related to the project but are useful to understand the differences between the two approaches (NAPI/non-NAPI) to receive data used by the network device drivers. Chapter 3 introduces some data structures used by the kernel to represent for example a network device or a packet. Chapter 4 deals with the packet path through the kernel and with the resulting differences between the two approaches. Chapter 5 describes the Linux netfilter framework and its design. The next chapter (6) begins with an overview of the Virtual File System (VFS); how file operations are working inside the kernel and after that follows a description of the capturing mechanism of the kernel module. The last chapter (7) summarizes the data structures and the functions mentioned in the paper.

2. Background

This chapter introduces two basic approaches used by network device drivers: interrupt or interrupt-pollled driven. Under certain conditions each method has advantages and disadvantages. The second one however has been developed later to avoid the receive livelock problem [ERLIK]. At last an overview of the TCP/IP kernel stack is presented.

2.1 Interrupts

Here the device driver instructs the device to generate a hardware interrupt when specific event occurs and the kernel, interrupted from its other activities, invokes an interrupt handler registered by the driver to take care of the device's need [ULNI]. When the event is a reception of a frame, the interrupt handler will queue the packet and will notify the kernel about it. With increasing traffic rates the amount of time, spent in interrupt handling, also increases and can compromise the system's stability. This technique is used by the most device drivers and represents the best option under low traffic loads.

2.2 Receive Livelock

The processing of an incoming packet through the kernel is split in two parts: first of all the driver saves an incoming packet into input queue accessible by the kernel and then the kernel processes the frame to an appropriate network layer. The code, that accepts a packet is executed in interrupt context and can preempt the processing part due to its higher priority. Under high network load [ULNI] the device generates an interrupt for every single packet which will lead to a system collapse, because the interrupt code will always preempt the processing code. At some point the input queue will be full, because the entire CPU time is spent in interrupt handling and the processing code, that is supposed to dequeue and process the frames from the queue, can not be executed. New frames cannot be queued since there is no available space and old packets cannot be processed, because there is no CPU time available for them. This condition is called receive-livelock.

2.3 Device Polling

"Device polling" refers to a technique for handling devices which does not rely on generated interrupts when they need attention, but on constantly checking a device whether it has anything to say [ULNI]. The disadvantage of this method is, that it can spend a lot of CPU time in checking for potential events and if the device can use interrupts it is recommended to avoid this method. Still, there are cases, where polling is the better choice. The main advantage of polling is in saving a bit of CPU cycles by reducing the number of context switches needed for interrupt handling.

2.4 Processing Multiple Frames During an Interrupt (NAPI)

To avoid the receive-livelock condition a new approach has been developed by Jamal Hadi Salim, Robert Olsson and Alexey Kuznetsov [BS], that reduces the number of interrupts generated by the system. The method is known as "new-API" or NAPI and uses the advantages of the polite behavior, where the interrupt – based handling of network devices is replaced with a mixed interrupt-polling mode. The device driver interrupts the CPU upon reception of the "first" packet, then the interrupt handler adds the device to a poll list and informs the kernel for the frame reception [ULNI]. The driver disables further interrupts for

incoming frames and all devices in the poll list are polled until certain number of packets is processed (to ensure fairness among devices). At last the driver re-enables receive interrupts. Under normal network load there are no differences between NAPI and non – NAPI device drivers, but a performance boost is achieved under heavy load.

2.5 Overview of the TCP/IP Kernel Stack

The Linux kernel supports many different network architectures and each of them describes how a specific network works. The architecture defines a set of layers and programs from each layer that communicate to one another by using a shared set of rules. For this project however the network architecture of interest is the IPv4 protocol stack and only related routines will be described. Figure 1 shows the different network layers and the correspondent protocols for each of them. At the bottom stands the link layer, that ensures the packet delivery from one network node to another and might provide methods to detect and possibly correct errors occurred during transmission. The device driver operates in this level and proceeds incoming /outgoing packets to/from the kernel and communicates directly with the network device. The next one is the network layer, responsible for end-to-end packet delivery and provides functions for network routing, fragmentation/defragmentation and error control. The transport layer provides transparent transfer of data between hosts and it is usually responsible for end-to-end error recovery and flow control and ensures complete data transfer.

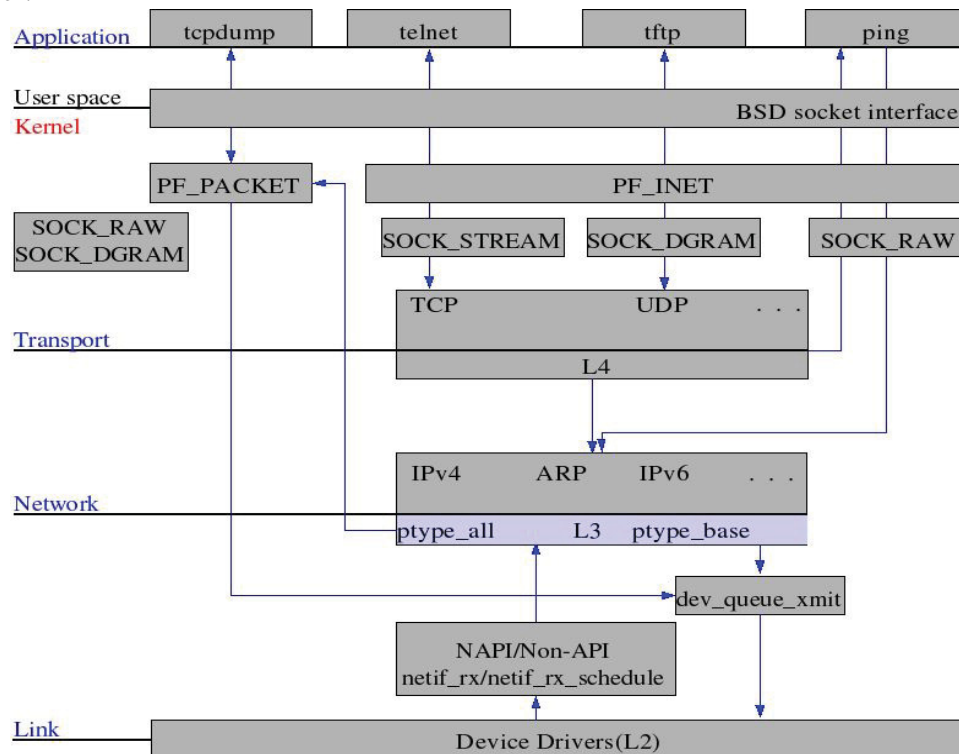


Figure 1: Packet flow within the network stack

The application layer at the top supports applications and end-user processes and everything at this layer is application-specific. The kernel sniffer works “somewhere at the top” of the link layer (protocol handlers) and inside the network layer (IPv4 netfilter hooks) that is why this paper describes the packet input traversal in these 2 layers.

3. Main Data Structures and Functions

This chapter will discuss main data structures and functions that are going to help to understand how device drivers work and how packet handling is implemented within the kernel.

3.1 Linux Socket Buffer

The socket buffer `sk_buff` is the structure used to address and manage a packet during its entire processing in the Linux kernel. A socket buffer consists of two parts: packet data, which stores the actual data of a packet that has been received or is about to be transmitted. And management data, in which the kernel maintains implementation specific data such as timers, pointers, counters etc. It is defined in `include/linux/skbuff.h` and from one kernel version to another there are slightly differences in some fields. Figure 2 shows some of the fields of `sk_buff` structure.

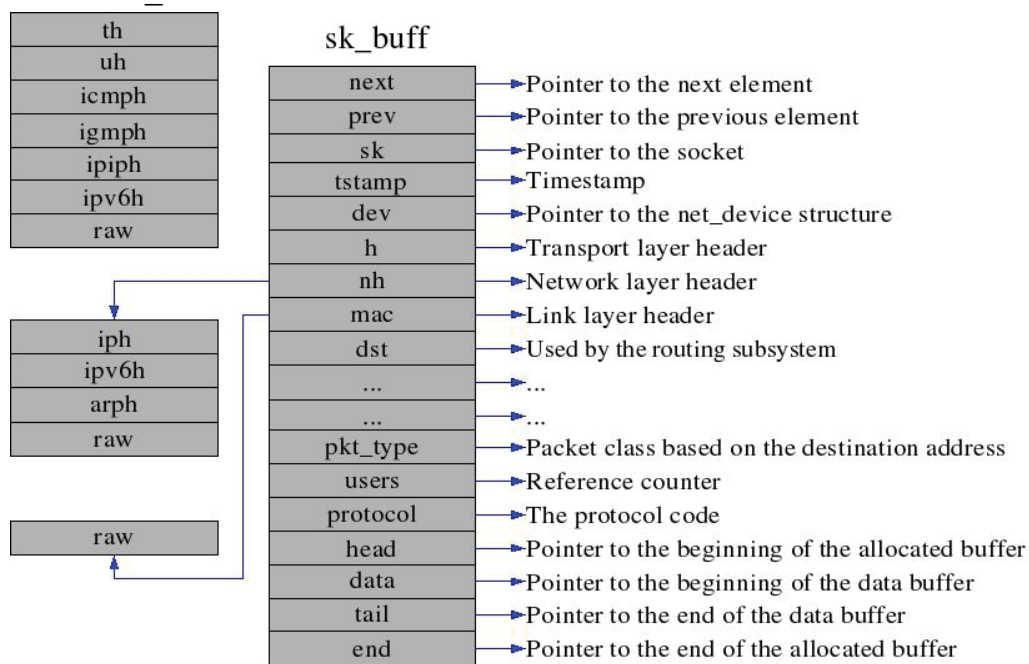


Figure 2: Fields of the `sk_buff` structure

```
struct sk_buff *next
struct sk_buff *prev
```

These fields are used to link the data structure in a double circular linked lists. There are generic `sk_buff` routines for adding packets to the front and to the end of these lists and for removing them.

```
struct sock *sk
```

This is a pointer to a sock data structure of a socket that owns the buffer. This pointer is needed when data is either locally generated or being received by a local process, because socket-related informations are used by transport layer protocols such as TCP and UDP and by user applications. When a buffer has to be forwarded this pointer is NULL.

```
struct skb_timeval tstamp
```

This field represents the time stamp of a packet that has been received and it is initialized during the reception. By default however this value is set to 0.0 to decrease the processing time of a single frame. Activation/deactivation of the time stamp is done by invoking the functions `net_enable_timestamp/net_disable_timestamp`, defined in `net/core/dev.c`.

```
struct net_device *dev
```

This field represents a network device receiving or transmitting this buffer. More detailed information about its type will be introduced later.

```
union {...} h  
union {...} nh  
union {...} mac
```

These are pointers to protocol headers for each network layer, `h` describes transport layer protocols (TCP, UDP), `nh` is the network layer protocol set and the `mac` field is for the link layer protocols. Each of these unions contain the `*raw` member that is used by the initialization of protocol and it can be used to access the protocol header instead the protocol specific pointer. Please refer to Figure 3.

```
struct dst_entry *dst
```

This member is used by the routing subsystem that is not covered in this document. For more details refer to the kernel source code or read O'Reilly's "Understanding Linux Network Internals" [ULNI].

```
__u8 pkt_type
```

This is the packet class based on the destination address: multicast, broadcast etc. The possible values are defined in `include/linux/if_packet.h`.

```
__be16 protocol
```

This is an unsigned short value that describes a network layer protocol. Because each protocol has its own function handler for processing incoming frames, the device driver informs the network layer what handler to use. The protocol specific values are defined in `include/linux/if_ether.h`.

```
atomic_t users
```

Reference counter or the number of entities using this socket buffer. It is used to avoid freeing the `sk_buff` structure when someone is still using it.

```
unsigned char *head, *data, *tail, *end
```

The `head` and `end` pointers point to the total location that can be used for packet data. The `data` and `tail` pointers point to currently valid packet data. The space between `head` and `data` is the so called headroom, and the space between `tail` and `end` is known as tailroom. The headroom and the tailroom allow a protocol to add protocol data in front of or after the currently valid packet data. The interaction with the actual packet data is depicted in Figure 3.

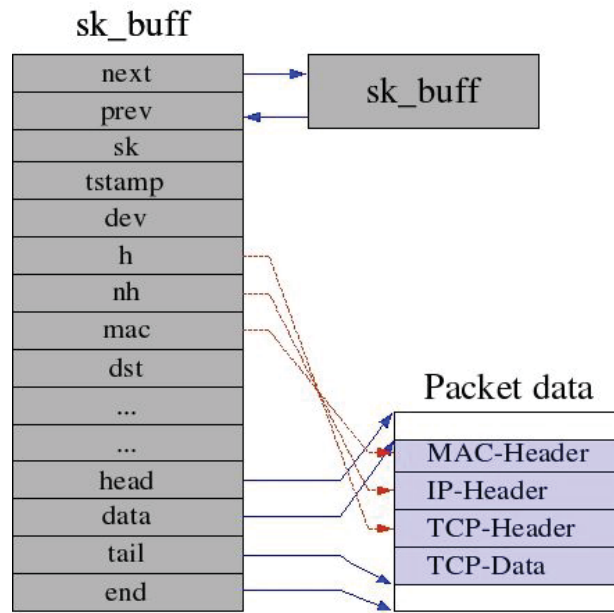


Figure 3: `sk_buff` and packet data

One of the problems having many network layers is the passing of a packet data from one layer to another, because each protocol needs to add protocol data by the transmission and to remove the data by reception. This make passing data buffers between the protocols difficult as each layer needs to find where its particular protocol headers and tails are. One solution is to copy buffers at each layer but that would be inefficient. Instead, Linux uses the `sk_buff` structure to pass data between the protocol layers and the network device drivers.

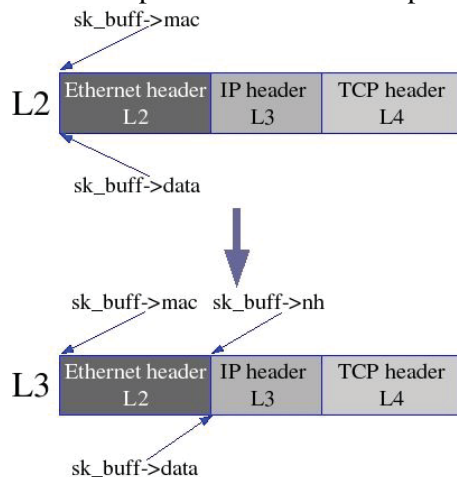


Figure 3.a: Pointers moving in packet reception

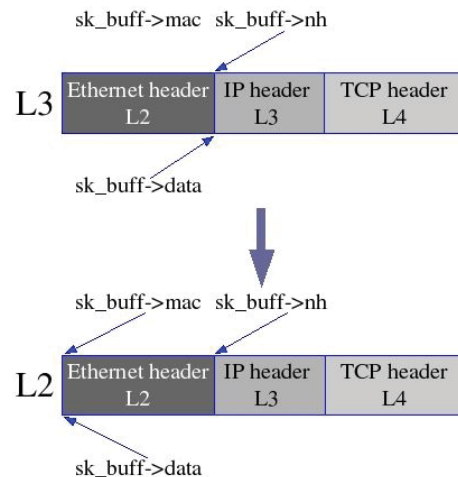


Figure 3.b P ointers moving when sending packets

While the `head` and `end` pointers are fixed, when the packet is allocated, the `data` and `tail` pointers are moved forward or backward when the packet is passed from one layer to another. Before passing a packet to another layer, the current layer initializes its own pointers and to prepare the packet for further proceSSION moves the `data` pointer to the beginning of the next layer. Figures 3.a and 3.b [ULNI] show how the `data` pointer is moved, when a packet is passed from link layer to network layer and from network layer to link layer.

3.2 Linux Network Device

Besides the socket buffer used to represent packets the core of the networking is the network device itself. In the Linux kernel every network device either a real one or a virtual one is represented through network device structure or `net_device` and it is defined in `include/linux/netdevice.h` (see Figure 4).

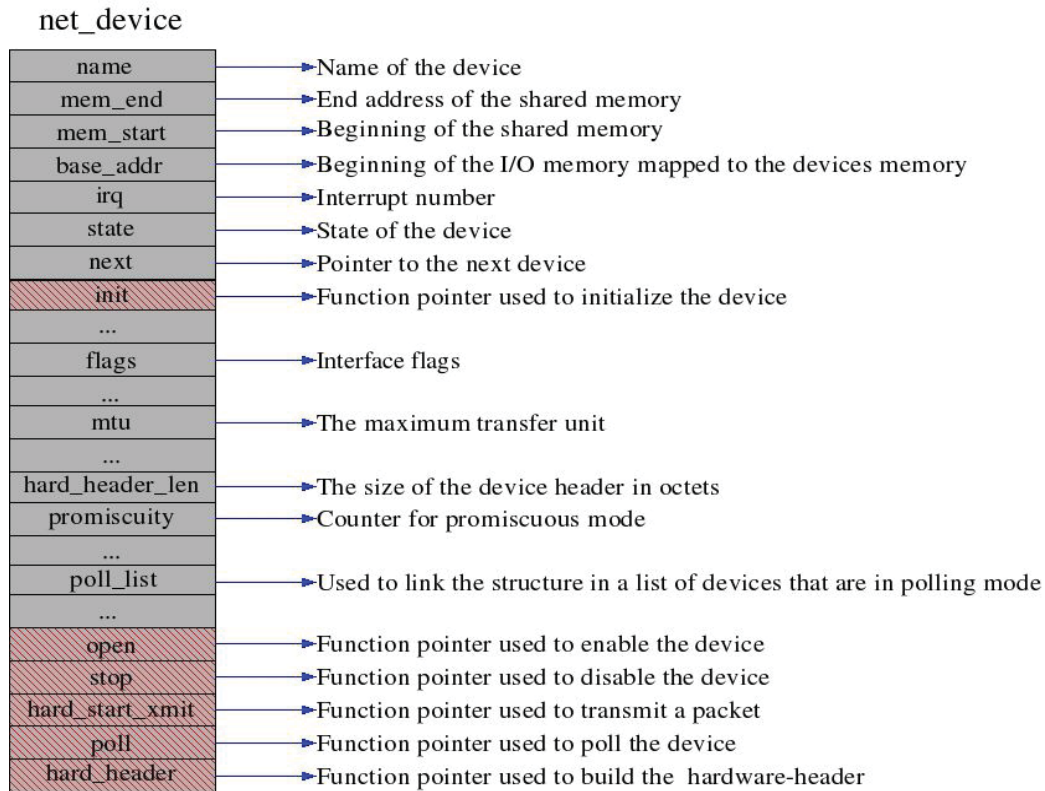


Figure 4: Fields of the `net_device` structure

```
char name[IFNAMSIZ]
```

This field stores the name of the network device and its size is up to 16 characters (`IFNAMSIZ=16`).

```
unsigned long mem_end
unsigned long mem_start
```

This fields describe the end and the beginning of the shared memory used by the device to communicate with the kernel.

```
unsigned long base_addr
```

This field stores the beginning address of the I/O memory mapped to the device's memory.

```
unsigned int irq
```

This field is the interrupt number used by the device. It can be shared among several devices.

```
unsigned long state
```

A set of flags used to represent the current state of the device.

`unsigned short flags`

Some bits in the `flags` field describe capabilities of the network device (`IFF_MULTICAST`) and others represent its status (`IFF_UP`: device is running, `IFF_DOWN`: device is stopped, `IFF_PROMISC`: device is in promiscuous mode). The different flags are defined in `include/linux/if.h`.

`unsigned mtu`

MTU is the abbreviation for Maximum Transfer Unit and it represents the maximum size of frames in bytes that the device can handle.

`unsigned short hard_header_len`

This value is the size of the device header in octets, for example the Ethernet header is 14 bytes long.

`int promiscuity`

This value is a counter that indicates whether a device is in promiscuous mode or not. The reason [ULNI] to be a counter rather than a simple flag is because several interested parties may ask for promiscuous mode. When the device enters in this mode the value is incremented by 1 and when leaves decreased by 1. The device is so long in promiscuous mode until the counter reaches 0.

`struct list_head poll_list`

As mentioned before the NAPI approach avoids the receive-livelock condition with reducing the number of generated interrupts by polling the device. As the name suggests this member is used to link the data structure in a global linked list of devices in polling mode.

```
int (*init)(struct net_device *dev)
int (*open)(struct net_device *dev)
int (*stop)(struct net_device *dev)
int (*hard_start_xmit)(struct sk_buff *skb,
                      struct net_device *dev)
int (*poll)(struct net_device *dev, int *quota)
int (*hard_header)(struct sk_buff *skb,
                  struct net_device *dev,
                  unsigned short type,
                  void *daddr, void *saddr,
                  unsigned len)
```

To be more generic the `net_device` structure contains also a set of function pointers that are device specific and implemented in the device driver. The first interesting member is the `init` function called once by the initialization of the network interface. The `open` and `stop` methods `open` or `stop` a device, for example `ifconfig eth0 up/down` will activate/deactivate the `eth0` device by calling the `open/stop` function. The function pointer `hard_start_xmit` initiates the transmission of the socket buffer. The method `poll` is provided for NAPI device drivers to proceed packets with disabled interrupts and the `hard_header` function builds the hardware header from source and destination addresses.

3.3 Linux Socket Buffer Queue

The kernel maintains `sk_buffs` structures in double linked lists, but its organization is more complicated than that of traditional linked lists. Each list is accessed by socket buffer head structure or `sk_buff_head`. It is defined in `include/linux/skbuff.h` and contains the members shown in Figure 5.

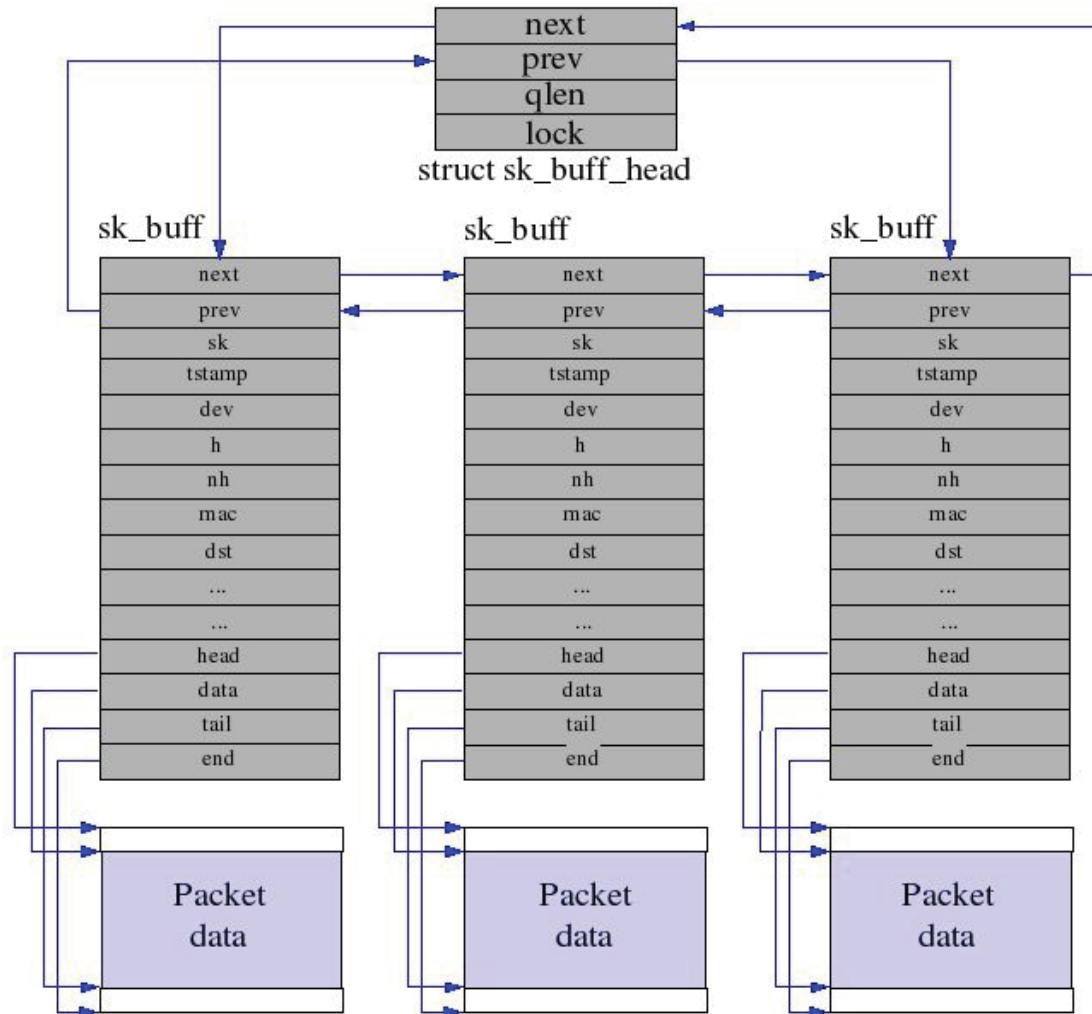


Figure 5: `sk_buff_head` interaction with `sk_buff`

```
struct sk_buff *next
struct sk_buff *prev
```

The `next` and `prev` fields are `sk_buff` pointers that are the boundaries of lists.

```
__u32 qlen
```

`qlen` is unsigned integer value and represents the number of elements in the list.

```
spinlock_t lock
```

This is synchronization primitive used to avoid race conditions among shared data and ensures in this case consistent list handling.

For more details see [UtLK] chapter 5 “Kernel Synchronization”.

3.4 Managing ingress/egress traffic

The next data structure is used to manage ingress/egress traffic and it is called `softnet_data`. As Figure 6 shows the structure includes both fields for reception and for transmission.

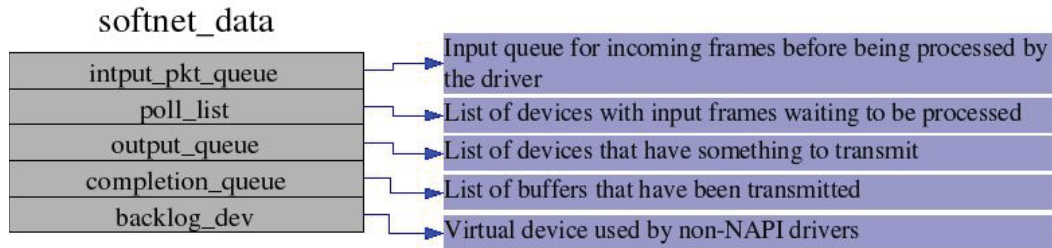


Figure 6: Fields of the `softnet_data` structure

```
struct net_device *output_queue
```

This is a list of devices that have something to transmit.

```
struct sk_buff_head input_pkt_queue
```

The `input_pkt_queue` member is a socket buffer queue, where incoming frames are stored before being processed by the driver. This queue is used only by non-NAPI device drivers and NAPI drivers are using their own queues.

```
struct list_head poll_list
```

Both NAPI and non-NAPI device drivers are inserted in this list when they have frames waiting to be processed. The `net_device` structure as mentioned before contains also a field `poll_list` but it is used to link the data structure in this list.

```
struct sk_buff *completion_queue
```

`completion_queue` is a list of socket buffers that have been successfully transmitted and can be released.

```
struct net_device backlog_dev
```

This field is used by non-NAPI device drivers and it represents a virtual device that has scheduled the `net_rx_action` software interrupt.

The `softnet_data` structure is defined per CPU so that no locking of the structure is needed among the different CPUs while processing the frames [ULNI].

3.5 Protocol Handler

When a packet has been received by the device driver it must be delivered to an appropriate network layer protocol (IPv4, ARP, IPv6) according to its protocol type. For each network layer protocol there is a protocol handler that proceeds the frames into its protocol stack. This is the `packet_type` structure (take a look at Figure 7) defined in `linux/include/netdevice.h`.

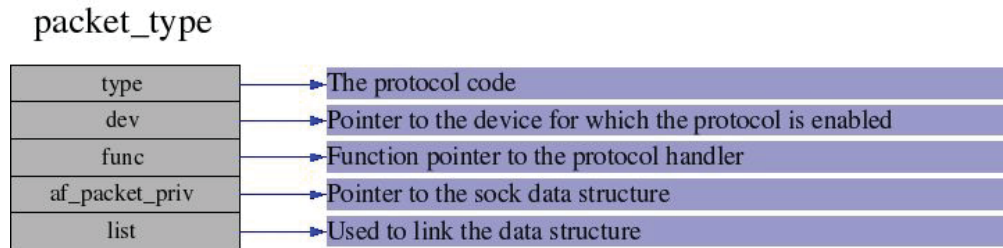


Figure 7: Fields of the `packet_type` structure

`__be16 type`

This is an unsigned short value that represents the protocol code and the possible values are defined in `include/linux/if_ether.h`.

`struct net_device *dev`

This is the network device for which the protocol is enabled [ULNI]. By setting this field to NULL means that the protocol is enabled for all devices. This parameter would allow to have different handlers for the different devices or one handler for a specific device.

`int (*func) (struct sk_buff *, struct net_device *, struct packet_type *, struct net_device *)`

The `func` member is the function handler that processes each frame into the network protocol stack. With invoking this handler a packet leaves the link layer and enters into the network layer. Table 1 shows some protocols and their correspondent function handlers.

`void *af_packet_priv`

The field is used by PF_PACKET sockets to deliver incoming frames to the right socket.

`struct list_head list`

This member is used to link the data structure in a linked list.

Each protocol registers itself by adding a `packet_type` data structure into either the `ptype_all` list or into the `ptype_base` hash table both defined in `net/core/dev.c`. The list `ptype_all` contains handlers for protocol sniffers while `ptype_base` hash table is hashed by protocol identifier and is used to decide which network layer protocol should receive the incoming packet.

When the fields of the `packet_type` data structure are initialized it can be inserted in the `ptype_all` list or in `ptype_base` hash table with `dev_add_pack` function defined in `net/core/dev.c`. This function checks the `packet_type`'s member `type` and decides if the structure must be linked in the list or in the hash table. Deletion of an entry from the list or from the hash table is done with `dev_remove_pack` function (wrapper to `__dev_remove_pack` function) also defined in `net/core/dev.c`.

Table 1: Some protocols and their function handlers

<i>Protocol</i>	<i>Protocol code</i>	<i>Function handler</i>	
ETH_P_IP	0x0800	ip_rcv	net/ipv4/ip_input.c
ETH_P_ARP	0x0806	arp_rcv	net/ipv4/arp.c
ETH_P_IPV6	0x86DD	ipv6_rcv	net/ipv6/ip6_input.c
ETH_P_IPX	0x8137	ipx_rcv	net/ipx/af_ipx.c
ETH_P_ALL	0x0003	packet_rcv	net/packet/af_packet.c
ETH_P_RARP	0x8035	ic_rarp_rcv	net/ipv4/ipconfig.c

4. NAPI/Non-NAPI Frame Reception

As mentioned before there are two modes for packet reception that may be implemented by the device driver: interrupt (non-NAPI) or interrupt-poll driven (NAPI). Let us start with non-NAPI frame reception. To become a more detailed and clear picture of the frame reception let see the initialization of the data structures discussed above that happens during the booting process in the `net_dev_init` (1) function.

4.1 `net_dev_init`

The `net_dev_init` (1) function defined in `net/core/dev.c` initializes the `ptype_all` list, the `ptype_base` hash table and the per-CPU defined `softnet_data` structures.

```
static int __init net_dev_init(void)
{
    ... ..
    1. The ptype_all list is initialized.
    List or hash table related functions are defined in include/linux/list.h.
        INIT_LIST_HEAD(&ptype_all);
    2. The sixteen lists in the ptype_base hash table are initialized.
        for (i = 0; i < 16; i++)
            INIT_LIST_HEAD(&ptype_base[i]);
    ... ..
    3. Initializes the softnet_data structures for each CPU.
        for (i = 0; i < NR_CPUS; i++) {
            struct softnet_data *queue;
    3.1 Defines for CPU i this softnet_data.
                queue = &per_cpu(softnet_data, i);
    3.2 Initializes the input queue for received packets input_pkt_queue
    by invoking the skb_queue_head_init function defined in include/linux/skbuff.h.
                skb_queue_head_init(&queue->input_pkt_queue);
    3.3 The queue for transmitted buffers is NULL because no packets have been transmitted yet.
                queue->completion_queue = NULL;
    3.4 Initializes the poll list where devices in polling mode will be inserted.
                INIT_LIST_HEAD(&queue->poll_list);
    ... ..
    3.5 Initializes the poll method for the virtual device backlog_dev with
    the process_backlog function defined in net/core/dev.c.
                queue->backlog_dev.poll = process_backlog;
    ... ..
        }
    ... ..
    4. Initializes the software interrupt handlers for transmission and reception of frames
    with net_tx_action/net_rx_action both defined in net/core/dev.c.
        open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
        open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
    ... ..
out:
    ... ..
}
```

1. `net_dev_init` function

The confusing part in `net_dev_init` (1) function is may be 3.5 - the initialization of the `poll` method for the virtual device `backlog_dev` that is used only by non-NAPI drivers. You would ask why non-NAPI drivers are providing `poll` method or the `process_backlog` (6) function? The new approach requires kernel changes which are going to result in old driver incompatibilities and the kernel will support only the new ones. To avoid this problem both NAPI and non-NAPI drivers must provide the `poll` method for packet processing, but non-NAPI haven't polite behavior. Therefore the `backlog_dev` virtual device is defined to provide the `process_backlog` (6) function as `poll` method and to process the packets from the input queue of the current CPU `input_pkt_queue`.

4.2 netif_rx

After the per-CPU `softnet_data`'s, the list `ptype_all` and the hash table `ptype_base` are initialized. We can start how device drivers are notifying the kernel for frame reception. In non-NAPI device drivers the code below is more or less similar and represents a generic interrupt handler (2). When a network device receives a frame the driver allocates `sk_buff` structure, stores the incoming packet in it. And to ensure that the packet is delivered to the appropriate network layer handler the `protocol` field of the `sk_buff` is set. For Ethernet devices this is done by invoking the `eth_type_trans` function defined in `net/ethernet/eth.c`. The important part is however the call of the `netif_rx` (3) function defined in `net/core/dev.c` that saves incoming frames into the input queue for the current CPU `input_pkt_queue`.

```
static irqreturn_t netdevice_interrupt(int irq, void *dev_id,
                                     struct pt_regs *regs)
{
    1. Allocate sk_buff structure for the incoming packet.
        skb = dev_alloc_skb(...);
        ... ..
        if (skb != NULL) {
            ... ..
    2. Assign the receiving device to the sk_buff .
        skb->dev = dev;
        ... ..
    3. Initialize the protocol field so that the packet is to be passed to an
        appropriate network layer protocol.
        skb->protocol = eth_type_trans(skb, dev);
        ... ..
    4. Invoke netif_rx function.
        netif_rx(skb);
        ... ..
        }
}
```

2. Generic non-NAPI interrupt handler

More information about Linux network device drivers can be found in [LDD] chapter 17 “Network Drivers”.

Now let us take a closer look in the `netif_rx(3)` function. It uses the incoming frame as argument and performs the following:

```
int netif_rx(struct sk_buff *skb)
{
    struct softnet_data *queue;
    ... ..
    1. If the device driver has not already time stamped the packet, it is done here.
        if (!skb->tstamp.off_sec)
            net_timestamp(skb);
        ... ..
    2. Get the softnet_data structure for current CPU where the packet will be queued.
        queue = &__get_cpu_var(softnet_data);
        ... ..
    3. The queue size is limited,so check if there is available space in the queue.
    (It can be modified with sysctl -w net.core.netdev_max_backlog=300)
        if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
            ... ..
        enqueue:
    4. Insert the packet into the softnet_data queue input_pkt_queue.
        __skb_queue_tail(&queue->input_pkt_queue, skb);
        ... ..
    5. Schedule software interrupt to process the packets. After that the kernel takes
    care for the packets and dequeues them from the queue.
        netif_rx_schedule(&queue->backlog_dev);
        ... ..
    }
    ... ..
}
```

3. netif_rx function

With the last step the kernel is notified for the received frame and this is done by invoking the `netif_rx_schedule` function defined in `include/linux/netdevice.h`. The function takes the device that receives the frame as an argument and it is a wrapper that checks if the device is running and that the software interrupt is not already scheduled. The “real job” is done by `__netif_rx_schedule(4)` also defined in `include/linux/netdevice.h`. It performs in general two things. First, the device for which the function is called is linked into the `softnet_data`'s `poll_list`. In the non-NAPI case this is the `softnet_data`'s virtual device `backlog_dev`. Second the software interrupt is scheduled.

```
static inline void __netif_rx_schedule(struct net_device *dev)
{
    ... ..
    1. Link the device into a list of devices in polling mode for the current CPU.
    This list is the softnet_data member poll_list.
    list_add_tail(&dev->poll_list,
        &__get_cpu_var(softnet_data).poll_list);
    ... ..
    2. Schedule the softirq for reception.
        __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

4. __netif_rx_schedule function

Now all structures of interest are initialized, the packet is inserted into the `input_pkt_queue` and the virtual device `backlog_dev` is linked into the `poll_list` of the current CPU. After the second step in the `__netif_rx_schedule` (4) function the device driver isn't anymore responsible for the packet and the kernel takes control over it. The `__raise_softirq_irqoff` function causes the execution of software interrupt handler and as previously shown in the `net_dev_init` (1) routine the `net_rx_action` (5) function has been installed as a handler for the softirq `NET_RX_SOFTIRQ`.

4.3 net_rx_action

As might be expected, its mission is to consume the packets from the input queues. This is done by running over the `poll_list` on the current CPU and for each device in the list its `poll` method is called. If `poll` returns value other than 0 it means that not all packets in the queue were dequeued (to ensure fairness amongst devices) and there are others waiting to be processed. For this reason the `net_device` structure that called its `poll` function is removed from the list and then reinserted. As you may be already noticed the `poll_list` is organized in FIFO-manner so the first inserted device in the list will be also processed first during the execution of the `net_rx_action` (5).

```
static void net_rx_action(struct softirq_action *h)
{
1. Get the softnet_data for the current CPU.
    struct softnet_data *queue=&__get_cpu_var(softnet_data);
    ... ..
2. While the poll_list contains devices it executes its poll method,remove the
the device from the poll_list and if after that the input queue still contains packets
reinsert the device into the poll_list.
    while (!list_empty(&queue->poll_list)) {
        struct net_device *dev;
        ... ..
2.1 Get the device from the poll_list.
        dev = list_entry(queue->poll_list.next, struct net_device, poll_list);
        ... ..
2.2 Call the poll method for this device.
        if (dev->quota <= 0 || dev->poll(dev, &budget)) {
            ... ..
2.3 If poll returns value different from 0 there are packets in the input queue
waiting to be processed. So remove the device form the poll_list.
            list_del(&dev->poll_list);
2.4 Reinsert the device into the poll_list.
            list_add_tail(&dev->poll_list, &queue->poll_list);
            ... ..
        } else {
            ... ..
        }
    }
    ... ..
}
```

5. net_rx_action function

4.4 process_backlog

In the `net_dev_init` (1) function the `poll` method of the virtual device `backlog_dev` is initialized with the `process_backlog` (6) function. It dequeues the frames from the `input_pkt_queue`, delivers them to an appropriate network layer protocol by calling the `netif_receive_skb` (7) function and if all packets are processed the `backlog_dev` is removed from `poll_list`.

```
static int process_backlog(struct net_device *backlog_dev,
                          int *budget)
{
    ... ..
    1. Get the softnet_data for the current CPU.
        struct softnet_data *queue=&__get_cpu_var(softnet_data);
        ... ..
        for (;;) {
            struct sk_buff *skb;
            struct net_device *dev;
            ... ..
            2. Dequeue a packet from the input_pkt_queue.
                skb = __skb_dequeue(&queue->input_pkt_queue);

            3. If the input_pkt_queue is empty the backlog_dev can be removed
            from the poll_list.
                if (!skb)
                    goto job_done;
                ... ..

            4. Deliver the packet to an appropriate network layer protocol.
                netif_receive_skb(skb);
                ... ..
        }
        ... ..
    job_done:
        ... ..
    5. Remove the backlog_dev from the poll_list.
        list_del(&backlog_dev->poll_list);
        ... ..
    return 0;
}
```

6. process_backlog function

4.5 netif_receive_skb

At this point we are at the end of the link layer and there is only one thing that must be done: to find an appropriate network layer protocol. This responsibility is taken by the `netif_receive_skb` (7) function. When I introduced the packet handler structure `packet_type` it was mentioned that `pptype_all` list and `pptype_base` hash table contain the registered protocol sniffers and protocol handlers. Both the list and the hash table were initialized in the `net_dev_init` (1) routine. The idea behind `netif_receive_skb` (7) is almost trivial: run over the list and the hash table and if an appropriate protocol handler is found invoke its function handler. This is done by a function wrapper `deliver_skb` defined in `net/core/dev.c` that first increments the reference counter of the packet and then calls the function handler.

```

int netif_receive_skb(struct sk_buff *skb)
{
1. Initialize the transport layer protocol and the network layer protocol pointers.
   skb->h.raw = skb->nh.raw = skb->data;
1.1 Calculate the MAC size.
   skb->mac_len = skb->nh.raw - skb->mac.raw;
   ... ..
2. For each entry in the ptype_all list check for registered protocol sniffers.
   list_for_each_entry_rcu(ptype, &ptype_all, list) {

2.1 Protocol handler wants to see all packets(its dev field is 0) or to
limit the packets to those received on a specific device.
   if (!ptype->dev || ptype->dev == skb->dev) {

2.2 If there is a registered sniffer for all devices or for a certain one call
its function handler through the deliver_skb.
   if (pt_prev)
       ret = deliver_skb(skb, pt_prev, orig_dev);
   pt_prev = ptype;
   }
}
... ..
3. Get the protocol value of the packet. This sk_buff field was initialized by
the device drivers. In the Ethernet case that was the eth_type_trans function.
   type = skb->protocol;
4. A Simple hash function finds the correspondent list in the hash table and for each entry
compares this time not only the packet_type's dev field with the sk_buff 's dev field, but
also the sk_buff's protocol field with the packet_type's protocol field.
   list_for_each_entry_rcu(ptype,
                           &ptype_base[ntohs(type)&15], list)
   {
       if (ptype->type == type &&
           (!ptype->dev || ptype->dev == skb->dev))
       {
           if (pt_prev)
               ret = deliver_skb(skb, pt_prev, orig_dev);
           pt_prev = ptype;
       }
   }
5. The searching for packet handlers is always one behind in search loop. So on exit from
the loop invoke the function handler for the last matched protocol.
   if (pt_prev) {
       ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
   } else {
6. If no protocol was matched the frame is discarded.
       kfree_skb(skb);
       ... ..
   }
out:
   ... ..
return ret;
}

```

7. netif_receive_skb function

4.6 NAPI Frame Reception

The differences between non-NAPI and NAPI drivers are that NAPI drivers must provide a `poll` method while non-NAPI are using the pseudo-polling process `process_backlog` (6) function and in the calling convention for scheduling software interrupt: non-NAPI drivers call `netif_rx` (3) function while NAPI drivers invoke `netif_rx_schedule` directly or `__netif_rx_schedule` (4) in combination with `netif_rx_schedule_prep` to check if a softirq is already scheduled [ULNI]. However not all devices can operate in NAPI mode, because the device must be able to store several packets on the card itself or in a DMA-ring and should be capable to disable interrupts for receiving packets [LDD]. Besides the device specific stuff in a NAPI device driver, the code below represents primitive interrupt handler (8) and `poll` method (9). The interrupt handler is modified so that `sk_buff` structures for the incoming packets are allocated in the `poll` method. The interrupts are disabled during the reception of the “first” packet, the device is inserted into the `poll_list` with `netif_rx_schedule` and software interrupt is scheduled. Then the processing of the frames is done as by non-NAPI drivers with `net_rx_action` (5) function that calls the device specific `poll` method. In general it must dequeue packets from a DMA-ring or a card memory, inserts the incoming data into socket buffers and to be delivered to an appropriate network layer it calls the `netif_receive_skb` (7) function. The last thing left to be done is to enable the interrupts if the receive buffer is empty.

```
static irqreturn_t netdevice_interrupt(int irq, void *dev_id,
                                       struct pt_regs *regs)
{
    ... ..
    1. Disable further interrupts.
    ... ..
    2. Insert the device into the poll_list for the current CPU.
       netif_rx_schedule(dev);
    ... ..
}
```

8. Generic NAPI interrupt handler

```

static int netdevice_poll (struct net_device *dev,int *budget)
{
1. Poll the device until the receive buffer(device memory or DMA-ring) contains
packets and certain amount of them are processed.
    while (Receive_Buffer_Not_Empty&&Processed_Packets<Quota) {
1.1 Allocate socket buffer.
        skb = dev_alloc_skb(packet_size + 2);
1.2 Initialize some of the sk_buff fields.
        skb->dev = dev;
        skb->protocol = eth_type_trans(skb,dev);
1.3 Deliver the packet to an appropriate network layer protocol.
        netif_receive_skb(skb);
        Processed_Packets++;
    }
2. If the receive buffer is empty the interrupts must be enabled.
    if (Receive_Buffer_Empty) {
2.1 Enable interrupts
2.2 The net_rx_action function must know that all packets are processed.
        return 0;
    }
2.3 The receive buffer contains still packets. Inform the net_rx_action function to remove and
reinsert the device into the poll_list.
return 1;
}

```

9. Generic NAPI poll method

5. Linux Netfilter Framework

After calling the `netif_receive_skb` (7) function incoming frames are leaving the link layer if appropriate network protocols are registered for them. However the protocol of interest is IPv4 and the accent will fall on the netfilter hooks in it. This chapter introduces some data structures and functions from the netfilter framework and explains its working mechanism. Linux netfilter is a framework responsible for packet mangling outside the normal Berkley socket interface [NF]. Each protocol defines set of “hooks” (IPv4 defines 5) which are well-defined points in a packet's traversal of that protocol stack [NF]. At each of this points the protocol calls the netfilter framework to “take care” of the passing packet. Parts of the kernel and kernel modules can register to the different hooks for each protocol and when a packet is passed to the netfilter framework, it checks if anyone has registered for that protocol and hook [NF]. Main features of the framework are:

1. stateless packet filtering
2. statefull packet filtering
3. network address and port translation
4. routing

5.1 nf_hook_ops

The hooks for the different protocols are defined in `include/linux/netfilter_xxx.h` where `xxx` denotes the protocol and their numbers differs between the protocol stacks: IPv4 defines 5 hooks, ARP – 3, DECnet – 7 . The registering to a netfilter hook is done with the `nf_hook_ops` data structure defined in `include/linux/netfilter.h` and its fields are explained in Figure 8 .

nf_hook_ops

list	→	Used to link the data structure
hook	→	Function pointer to the hook-function
owner	→	Module owner
pf	→	Protocol family
hooknum	→	Hook number
priority	→	Priority

Figure 8: Fields of the `nf_hook_ops` structure

```
struct list_head list
```

Used to link the data structure into two dimensional array `nf_hooks` that contains the registered hook objects.

```
nf_hookfn *hook
```

A function pointer called when a packet is passed to a hook and performs one of the following actions on a packet:

<code>NF_ACCEPT</code>	will allow a packet to continue the traversal through the protocol stack.
<code>NF_DROP</code>	will drop the packet
<code>NF_STOLEN</code>	take control over the packet, don't continue the traversal
<code>NF_QUEUE</code>	queue the packet (usually for userspace handling)
<code>NF_REPEAT</code>	call this hook again.

The five values are defined in `include/linux/netfilter.h` .

```
struct module *owner
```

Pointer to the kernel module that owns the `nf_hook_ops` structure.

```
int pf
```

The protocol family to that the hook belongs (ARP , IPv4 etc.). Except `NF_ARP` for ARP defined in `include/linux/netfilter_arp.h` the possible values are defined in `include/linux/socket.h` (netfilter hooks are not inserted in every protocol stack).

```
int hooknum
```

This represents the different hooks in a protocol stack. For example IPv4 defines 5 hooks defined in `include/linux/netfilter_ipv4.h`:

```
NF_IP_PRE_ROUTING
```

```
NF_IP_LOCAL_IN
```

```
NF_IP_FORWARD
```

```
NF_IP_LOCAL_OUT
```

```
NF_IP_POST_ROUTING
```

```
int priority
```

This is the priority of a hook entry. Each one is inserted according to its priority so that they are ordered in ascending priority. The values are protocol specific and defined in `include/linux/netfilter_xxx.h`.

5.2 Netfilter Hooks in IPv4 Protocol Stack

Figure 9 shows the IPv4 hooks. Every packet that enters the protocol stack is passed first to the prerouting hook `NF_IP_PRE_ROUTING`. Then follows a routing code which decides whether the frame is destined for another interface or a local process. The routing code may drop packets that are unroutable. If the packet is for a local process it enters the `NF_IP_LOCAL_IN` hook and if it is destined to another interface the `NF_IP_FORWARD` hook is called. The `NF_IP_LOCAL_OUT` hook is called for packets created by a local process and at the end every packet leaves the protocol stack through the `NF_IP_POST_ROUTING` hook [NF].

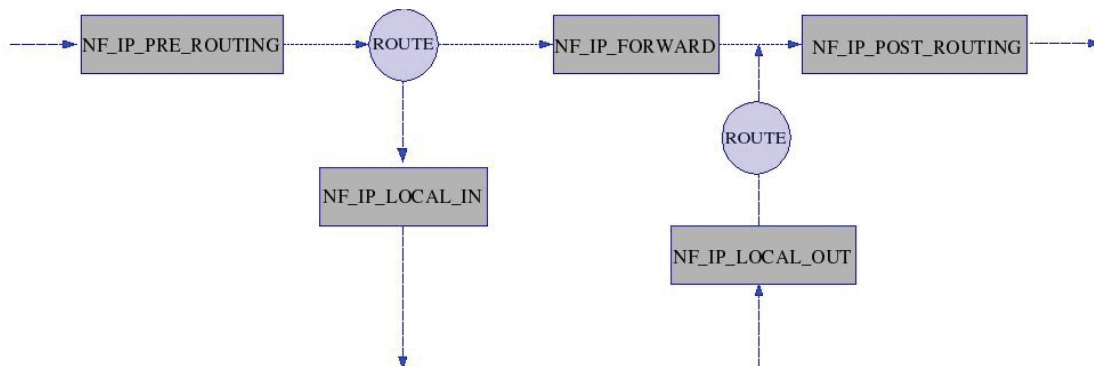


Figure 9: IPv4 netfilter hooks

5.3 Netfilter Framework in Details

As mentioned before the registered hook entries are stored in two dimensional array called `nf_hooks[NPROTO][NF_MAX_HOOKS]` defined in `net/netfilter/core.c` where `NPROTO` is the maximum number of protocols supported from the kernel (at the moment 32) and `NF_MAX_HOOKS` is the maximum number of hooks defined in a protocol stack (this is the DECnet protocol that defines 7 hooks) plus 1. Each element of the table is a list so that several hooks can register to the same hook but with different priority.

The `nf_register_hook(10)` function inserts a `nf_hook_ops` structure into the table according to its priority (the `priority` member of the structure) and to remove one there is the `nf_unregister_hook(11)` function both defined in `net/netfilter/core.c`.

```
int nf_register_hook(struct nf_hook_ops *reg)
{
    struct list_head *i;
    1. Find the correspondent entry in the table according to the protocol family and
    the hook number. Run over the list until a lower hook object's priority occur.
    list_for_each(i, &nf_hooks[reg->pf][reg->hooknum]) {
        if(reg->priority < ((struct nf_hook_ops *)i)->priority)
            break;
    }
    2. Insert the nf_hook_ops structure.
    list_add_rcu(&reg->list, i->prev);
    return 0;
}
```

10. nf_register_hook function

```
void nf_unregister_hook(struct nf_hook_ops *reg)
{
    1. Remove a hook object from the list.
    list_del_rcu(&reg->list);
    ... ..
}
```

11. nf_unregister_hook function

Each hook is called by a protocol stack with the `NF_HOOK(12)` wrapper defined in `include/linux/netfilter.h`. It checks if there are registered objects to a hook, calls the `nf_hook_slow` function defined in `net/netfilter/core.c` and if the packet is accepted from netfilter subsystem the wrapper calls the next protocol routine of the stack.

A general call looks like `NF_HOOK(protocol, hook, sk_buffer, input_device, output_device, next_protocol_call)`.

```
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn)
({int __ret;
    1. Check list entry from the nf_hooks table and if the packet is accepted.
    if (list_empty(&nf_hooks[pf][hook]) ||
    2. Call nf_hook_slow function.
    (__ret=nf_hook_slow(pf, hook, &(skb),
    indev, outdev, okfn, INT_MIN)) == 1)
    3. A packet is accepted and can continue its traversal. Invoke the next protocol call of the stack.
    __ret = (okfn)(skb);
    ret;})
```

12. NF_HOOK wrapper

For better explanation of the netfilter subsystem let see the `ip_rcv` (13) function where the prerouting `NF_IP_PRE_ROUTING` hook is placed. First it checks if the packet is destined to the machine so that packets that enter the protocol stack during promiscuous mode are dropped. General checks follow for different kinds of errors such as incorrect size, checksum etc.

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev,
           struct packet_type *pt, struct net_device *orig_dev)
{
    1. When the interface is in promiscuous mode drop all the crap
    that it receives, do not try to analyze it.
        if (skb->pkt_type == PACKET_OTHERHOST)
            goto drop;
        ... ..
    2. Call the prerouting netfilter hook.
        return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING,
                       skb, dev, NULL, ip_rcv_finish);

    3. By error discard the sk_buff structure.
inhdr_error:
    ... ..
drop:
    kfree_skb(skb);
out:
    ... ..
}
```

13. `ip_rcv` function

In the second step a frame enters the netfilter subsystem. In this case the protocol family is IPv4 or `PF_INET`, the hook is the prerouting `NF_IP_PRE_ROUTING`, `skb` is the packet, `dev` is the device that received the frame and the function `ip_rcv_finish` is the next protocol call of the stack. The `nf_hook_slow` (14) gets the correspondent list from the `nf_hooks` table according the protocol family and the hook. Then it runs over the list and for registered hook objects `nf_hook_ops` calls their function pointers. The iterating over the list is done with `nf_iterate` (15) routine also defined in `net/netfilter/core.c`.

```

int nf_hook_slow(int pf,unsigned int hook,
                 struct sk_buff **pskb,struct net_device *indev,
                 struct net_device *outdev,
                 int (*okfn)(struct sk_buff *), int hook_thresh)
{
1. Get the correspondent list from the nf_hooks table according to the protocol family and the hook.
    elem = &nf_hooks[pf][hook];
next_hook:
2. Start iterating over the list.
    verdict = nf_iterate(&nf_hooks[pf][hook], pskb, hook,
                        indev, outdev, &elem, okfn, hook_thresh);
3. Check return value.
    if (verdict == NF_ACCEPT || verdict == NF_STOP) {
3.1 Packet is accepted and can continue its traversal.
        ret = 1;
        goto unlock;
    } else if (verdict == NF_DROP) {
3.2 Packet is dropped.
        kfree_skb(*pskb);
        ret = -EPERM;
    } else if ....
    }
unlock:
    return ret;
}

```

14. nf_hook_slow function

```

unsigned int nf_iterate(struct list_head *head,
                       struct sk_buff **skb,
                       int hook, const struct net_device *indev,
                       const struct net_device *outdev,
                       struct list_head **i,
                       int (*okfn)(struct sk_buff *),
                       int hook_thresh)
{
    unsigned int verdict;
1. Run over the list.
    list_for_each_continue_rcu (*i, head) {
        struct nf_hook_ops *elem = (struct nf_hook_ops *)*i;
2. Call the nf_hook_ops function.
        verdict = elem->hook(hook,skb,indev,outdev,okfn);
3. Check return value.
        if (verdict != NF_ACCEPT) {
            ...
            if (verdict != NF_REPEAT)
                return verdict;
            *i = (*i)->prev;
        }
    }
return NF_ACCEPT;
}

```

15. nf_iterate function

6. Kernel Sniffer

At this point we know almost everything to create a kernel sniffer except a method to write a captured packet into a file. This chapter deals first with an overview of the Virtual File System (VFS) and then explains the capturing mechanism of the kernel module.

6.1 Accessing File from Kernel Module

Accessing a file from kernelspace is not as trivial as in userspace. File operations are not supposed to be done in kernelspace it is dangerous and can compromise the system stability. However there are several projects that are dealing with files in kernelspace and maybe the best known is the kernel-based web server TUX maintained by Ingó Molnár. The difficulty of accessing files from kernel module is that it can't simply invoke system calls such as `open()` or `write()`, because in that case data is moved from userspace to kernelspace and obviously a kernel module works in kernelspace. The solution of the problem is to use the Virtual Filesystem layer (see Figure 10). Linux supports many different filesystems and to enable the upper levels of the kernel to deal equally with all of these filesystems Linux defines an abstract layer known as Virtual Filesystem or VFS [UtLK]. Each lower level must provide an interface which conforms to VFS.

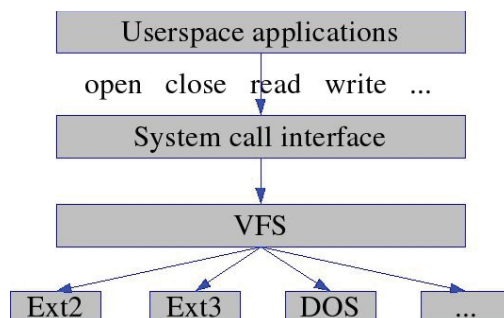


Figure 10: Virtual filesystem

The VFS contains a number of generic object types and methods which can be called on these objects. The basic objects are as follows and their interaction is depicted in Figure 11:

- | | |
|--|-------------------------------------|
| 1. file (information about an open file) | <code>include/linux/fs.h</code> |
| 2. inode (information about a file) | <code>include/linux/fs.h</code> |
| 3. dentry (information about a directory) | <code>include/linux/dcache.h</code> |
| 4. superblock (information about a filesystem) | <code>include/linux/fs.h</code> |

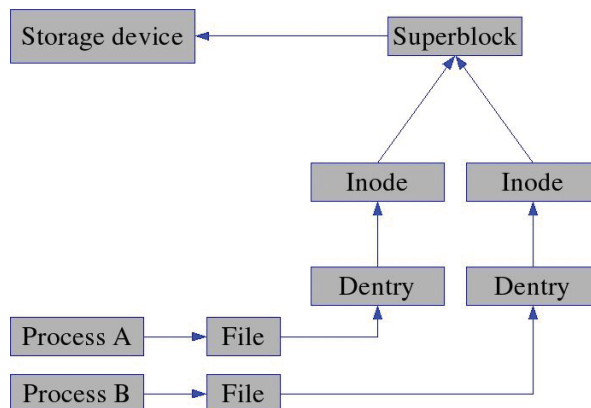


Figure 11: Virtual filesystem objects

For the project the component of interest is the file object that stores information about the interaction between an open file and a process. It is represented through a data structure `struct file` defined in `include/linux/fs.h`. The structure contains also a member `f_op` that is a number of methods for file handling (`struct file_operations` defined in `include/linux/fs.h`) and after a file is opened it can be accessed through this methods. For example when a file is opened it can be written with `file->f_op->write`. To initialize and open a `file` structure the kernel can not invoke a system call as mentioned above. But following the call chain of the system call `sys_open` defined in `fs/open.c` give us the function that opens a file: `filp_open` defined also in `fs/open.c`. After a file is opened it can be easily manipulated through the methods for file handling in it and it is closed with `filp_close` defined in `fs/open.c`. More information about VFS can be found in [UtLK] chapter 12 “The Virtual Filesystem”.

6.2 Design

First the kernel module should be registered to the netfilter subsystem and especially to the prerouting and postrouting hooks. So packets that enter the IPv4 protocol stack will be passed through the `NF_IP_PRE_ROUTING` hook and those that leave the machine through the `NF_IP_POST_ROUTING` hook. There is only one problem: packets that are not destined to the machine during promiscuous mode will be dropped from `ip_rcv` (13) function when they enters the IPv4 protocol stack(see 6. Linux Netfilter Framework). So how can the module capture those packets? The answer is the `ptype_all` list that contains protocol sniffers. Adding an entry in it ensures that every packet that enters the machine will be passed first to a protocol handler from the list and then eventually to one from the hash table `ptype_base` where the “real” handlers are stored. The protocol handler inserted in `ptype_all` must capture only those packets that are not destined to the machine otherwise packets destined to the machine will be captured twice: first from the protocol handler in `ptype_all` and second from the netfilter hooks. Now the kernel module can monitor each packet that enters or leaves the IPv4 protocol stack. However frames must be saved somewhere before being written into file. For this reason a sniffer usually uses a circular buffer of fixed size to store the captured packets. Nevertheless I chose another approach: to use the same methods as the kernel to maintain packets: fixed size queues. So when a packet is passed through one of these three points it will be enqueued in a socket buffer queue and dequeued after it is being written into a file. Figure 12 makes an overview of the sniffer. First the module creates a `file` object with `filp_open` function where the captured packets will be saved. When the file is initialized the pcap file header must be written with the write method of the file `file->f_op->write`. The next step is to set the device that are to be sniffed in promiscuous mode with `dev_set_promiscuity` and to enable the time stamp for packets with `net_enable_timestamp`. After that the packet capturing can begin and the module registers to the prerouting `NF_IP_PRE_ROUTING` hook, the postrouting `NF_IP_POST_ROUTING` hook with the `nf_register_hook` (10) function and inserts a `packet_type` handler in the `ptype_all` list with the `dev_add_packet` function to capture packets not destined to the machine. The last thing left to be done is to start a kernel thread that will dequeue frames from a socket buffer queue and writes them in to a file with `file->f_op->write`.

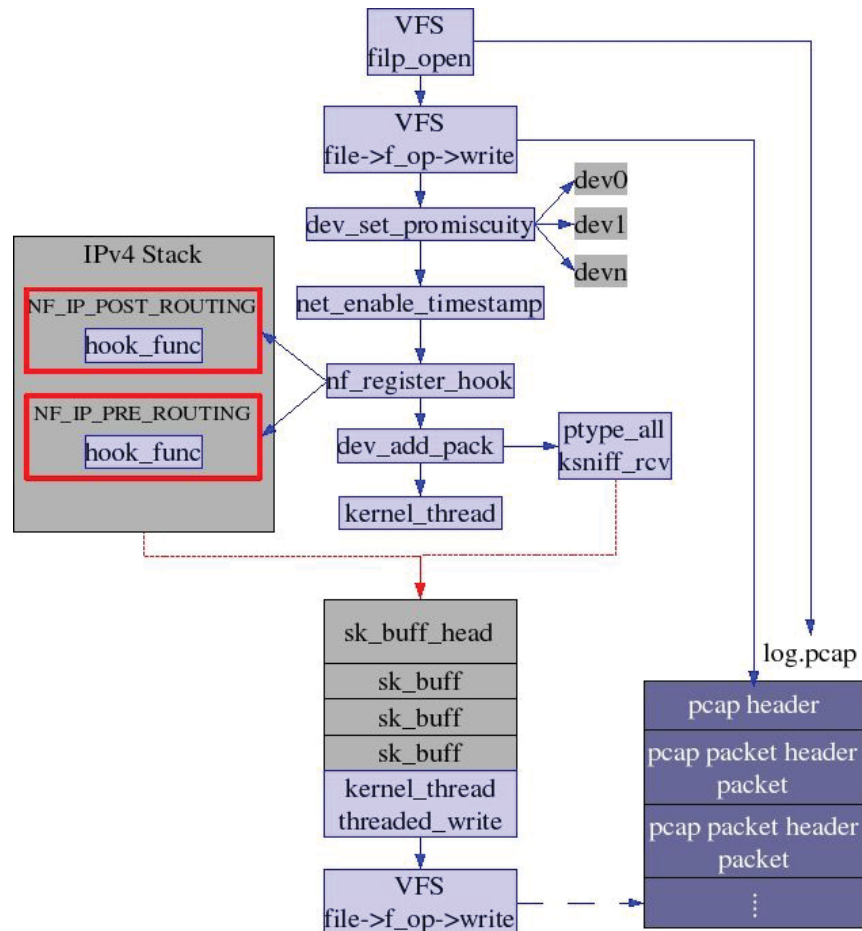


Figure 12: Kernel sniffer design

Packet capturing at high network rates is a challenge due to various hardware limitations. Disk I/O operations are very slow in comparison with other system activities and they are performance killer. A lot of time is spent for writing frames into a file and this causes packets lost, because the amount of data received from the network device is bigger then the disc throughput. Also invoking a write operation for every single packet will additionally increase this time and it will decrease the capturing rate. To reduce the number of write accesses the kernel thread responsible for dequeuing and writing processes multiple packets at once.

The kernel module can take the following arguments:

queue_size: The maximum queue size of the sniffer by default the size is 10000 packets.

device_name: Captures packets on a specific device by default all devices are sniffed.

logfile: File where captured packets are saved by default log.pcap in the module's directory

snapplen: Snarf snaplen bytes of data from each packet by default snaplen is 96 bytes.

Changing the queue size can slightly improve the capturing performance of the module.

When the kernel sniffer is inserted it creates entry in **/proc/ksniff** where different statistics of a network device are exported such as received packets, captured packets, errors etc.

6.3 Benchmarks

Below are listed some tests and comparisons with tcpdump done on Athlon XP 1800, RAM:256 MB and maximal disk's write speed ~ 34 MB/s .

<i>TEST 1 : kernel sniffer, snaplen=1500</i>	<i>TEST 1: tcpdump, snaplen=1500</i>
Packets:2000000 (1496byte,0frags) 70808pps 847Mb/sec (847432454bps) errors: 0 Captured packets:603874 Received packets:655560	Packets:2000000 (1496byte,0frags) 70800pps 847Mb/sec (847344015bps) errors: 0 589831 packets captured 661719 packets received by filter
<i>TEST 2: kernel sniffer, snaplen=96</i>	<i>TEST 2: tcpdump, snaplen=96</i>
Packets:2000000 (1496byte,0frags) 70799pps 847Mb/sec (847331807bps) errors: 0 Captured packets:647783 Received packets:647783	Packets:2000000 (1496byte,0frags) 70808pps 847Mb/sec (847431164bps) errors: 0 642799 packets captured 645014 packets received by filter
<i>TEST 3: kernel sniffer, snaplen=1500</i>	<i>TEST 3: tcpdump, snaplen=1500</i>
Packets:10.000.000 (1496byte,0frags) 47274pps 565Mb/sec (565784851bps) errors: 0 Captured packets:3791329 Received packets:9844006	Packets:10.000.000 (1496byte,0frags) 47088pps 563Mb/sec (563557308bps) errors: 0 3643704 packets captured 9930613 packets received by filter
<i>TEST 4: kernel sniffer, snaplen=96</i>	<i>TEST 4: tcpdump, snaplen=96</i>
Packets:10.000.000 (1496byte,0frags) 48294pps 577Mb/sec (577992364bps) errors: 0 Captured packets:9258105 Received packets:9813181	Packets:10.000.000 (1496byte,0frags) 48666pps 582Mb/sec (582438142bps) errors: 0 9582944 packets captured 9726366 packets received by filter

7. Summary

Data Structures

struct sk_buff	Data structure used to represent a packet.	4
struct net_device	Represents a network device.	7
struct sk_buff_head	Maintains packets in double circular linked list.	9
struct softnet_data	Used to manage ingress/egress traffic.	10
struct packet_type	Protocol handler.	11
nf_hook_ops	Data structure used to register to a netfilter hook.	21

Functions

net_enable_timestamp	Activate timestamp for sk_buffs .	5
net_disable_timestamp	Deactivate timestamp for sk_buffs .	5
dev_add_pack	Inserts a packet_type data structure in ptype_all list or ptype_base hash table.	11
dev_remove_pack	Removes a packet_type data structure from ptype_all list or ptype_base hash table.	11
net_dev_init	Called at a boot time to initialize the per-CPU softnet_data structures, the protocol sniffer list ptype_all , the protocol hash table ptype_base and software interrupt handlers for reception/transmission.	13
open_softirq	Registers a software interrupt handler.	13
netif_rx	Used by non-NAPI drivers to queue an incoming frames in input_pkt_queue and schedules a software interrupt with netif_rx_schedule .	14
netif_rx_schedule / __netif_rx_schedule	Inserts a device in a poll_list of the current CPU and schedule the software interrupt handler for frame reception.	15
__raise_softirq_irqoff	Schedules a software interrupt.	16
net_rx_action	The software interrupt handler that calls for each device in a poll_list its poll method.	16
process_backlog	poll virtual function used by non-NAPI device drivers.	17
netif_receive_skb	Processes incoming packets to an appropriate network layer.	17
NF_HOOK	Macro wrapper in a protocol stack that inserts a packet in the netfilter framework.	23
nf_register_hook	Registers a nf_hook_ops object to a netfilter hook.	23
nf_unregister_hook	Unregisters a nf_hook_ops object.	23
ip_rcv	Function handler called from netif_receive_skb for IPv4 frames.	24

Bibliography

- [ULNI] C. Benvenuti. *Understanding Linux Network Internals (first edition)*. O'Reilly
- [UtLK] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel (second edition)*. O'Reilly
- [LDD] J. Corbet, A. Rubini and G. Kroah-Hartman. *Linux Device Drivers (third edition)*. O'Reilly
- [ERLIK] J. C. Mogul and K. K. Ramakrishnan. *Eliminating Receive Livelock in an Interrupt-driven Kernel* (Jul 2006). www.usenix.org/publications/library/proceedings/sd96/full_papers/mogul.ps
- [BS] J. H. Salim, R. Olsson and A. Kuznetsov. *Beyond Softnet* (Jul 2006). www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf
- [LKD] Linux Kernel Documentation. linux/Documentation
- [TLDP] The Linux Documentation Project (Jul 2006). www.tldp.org
- [LJ] Linux Journal: The Original Monthly Magazine of the Linux Community (May 2006). www.linuxjournal.com
- [NF] netfilter: firewalling, NAT, and packet mangling for Linux (Jun 2006). www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html