Understanding the Kernel Network Layer

Breno Leitão Arnaldo Carvalho

June 25, 2009

Agenda

```
# lspci -vnn
02:00.0 Ethernet controller [0200]: Intel Corporation
82573L Gigabit Ethernet Controller [8086:109a]
```

Example

```
#define E1000_DEV_ID_82573L   0x109A
#define PCI_VENDOR_ID_INTEL   0x8086

static struct pci_device_id e1000_pci_tbl[] = {
...
{ PCI_VDEVICE(INTEL, E1000_DEV_ID_82573L), board_82573 },
...
}
```

Interrupt Handling

```
request_irq(unsigned int irq, irq_handler_t handler,
    unsigned long flags, const char *name, void *dev)
```

Computer's firmware assigns a unique interrupt number to the device during boot.

Interrupt Handling 2

Example

```
static irqreturn_t e1000_intr(int irq, void *data)

int irq_flags = IRQF_SHARED;
irq_handler_t handler = e1000_intr;
struct net_device *netdev = adapter->netdev;

err = request_irq(adapter->pdev->irq, handler,
  irq_flags, netdev->name, netdev);
```

The Interrupt Handler

Example

```
static irqreturn_t e1000_intr(int irq, void *data) {
                u32 rctl, icr = er32(ICR);

                if (!icr)
                        return IRQ_NONE;

        /* Body */

        return IRQ_HANDLED;
}
```

Receiving a frame

1. Packet received at the card
2. NIC generates an IRQ
3. Interrupt Handler called. `e1000_intr()`
4. `__napi_schedule(&adapter->napi)`
5. `Raise SoftIRq NET_RX_SOFTIRQ`

NET_RX_SOFTIRQ

1. `SoftIrq calls e1000_clean()` Registered using netif_napi_add()
2. calls `e1000_clean_rx_irq()`
   1. `netif_receive_skb()` - Process receive buffer from network
      1. `deliver_skb()` - Calls `packet_type->func()`

deliver_skb()

```
static struct packet_type ip_packet_type = {
        .type = cpu_to_be16(ETH_P_IP),
        .func = ip_rcv,
.....
}
```

ip_rcv

1. calls NF_HOOK() for NetFilter validation
2. if ok calls `ip_rcv_finish`
3. calls `ip_route_input()` to check route and remove spoofing
4. calls `dst_input()`
5. which `call skb->dst->input()` - Input packet from network to transport.
6. `.input= ip_local_deliver()`
7. another NF_HOOK() that call `ip_local_deliver_finish()`
8. that call `ipprot->handler(skb)`

TCP layer

```
static struct net_protocol tcp_protocol = {
        .handler =      tcp_v4_rcv,
        .err_handler =  tcp_v4_err,
        .gso_send_check = tcp_v4_gso_send_check,
        .gso_segment =  tcp_tso_segment,
        .gro_receive =  tcp4_gro_receive,
        .gro_complete = tcp4_gro_complete,
        .no_policy =    1,
        .netns_ok =     1,
};
```

tcp_v4_rcv

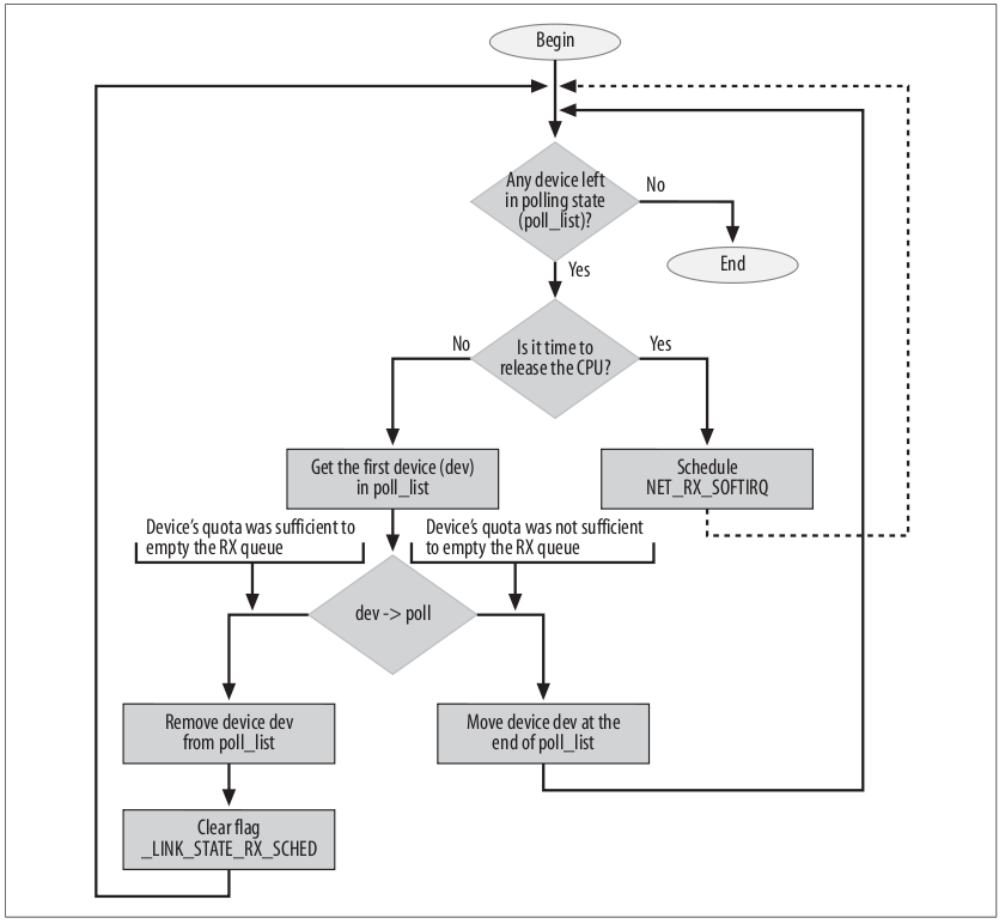
1. set some skb flags related to TCP
2. calls `tcp_v4_do_rcv()`
3. calls `tcp_v4_hnd_req()`, which tries to find, using `tcp_v4_hnd_req/inet_lookup_established()`, who is the socket that will handle that packet.
4. Calls **tcp_rcv_state_process()** which implement receiving procedure of RFC 793.
5. calls `tcp_validate_incoming()` that checks for sequence number and RST flag.
6. then if `TCP_ESTABLISHED` call `tcp_data_queue()`

tcp_data_queue()

```
/*  Queue data for delivery to the user.
 *  Packets in sequence go to the receive queue.
 *  Out of sequence packets to the out_of_order_queue.
 */
```

1. `if (th->fin) then tcp_fin();`
2. Depends on the type of socket
3. If in seq and in window, call `skb_copy_datagram_iovec()` to copy the data to the userspace using `memcpy_toiovec()`
4. `tcp_rcv_space_adjust()` adjusts space to copy the data
5. Wake up the waiter, through `sock_def_readable`
6. `sk->sk_data_ready = sock_def_readable`

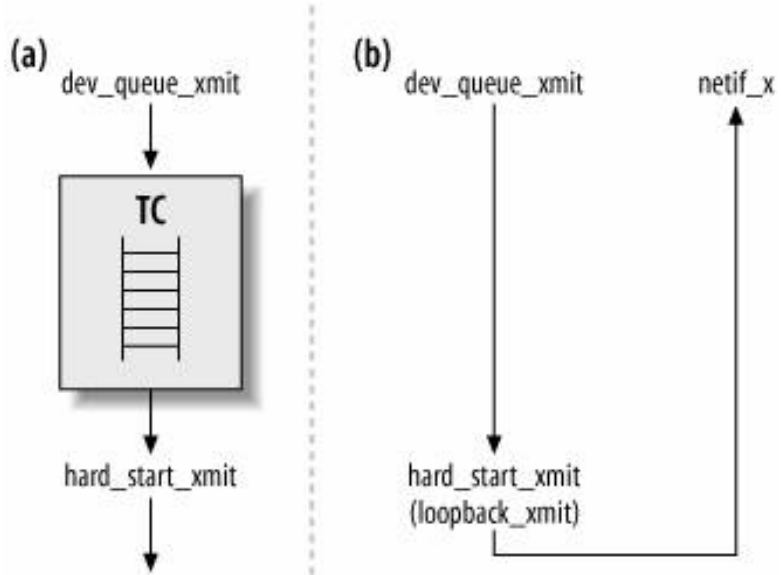
NAPI


Begin
Any device left in polling state (poll_list)?
No
End
Yes
No
Is it time to release the CPU?
Yes
Get the first device (dev) in poll_list
Schedule NET_RX_SOFTIRQ
Device's quota was sufficient to empty the RX queue
Device's quota was not sufficient to empty the RX queue
dev -> poll
Remove device dev from poll_list
Move device dev at the end of poll_list
Clear flag _LINK_STATE_RX_SCHED

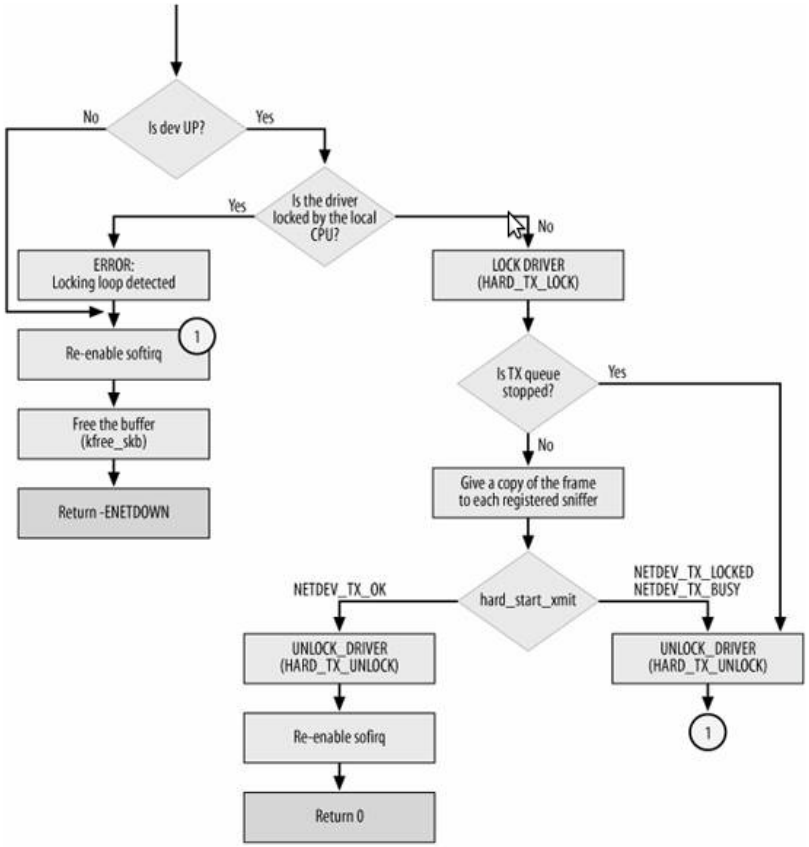
Transmitting

- packet_sendmsg() copies the packet to kernel space using `memcpy_fromiovec()`
- and calls `dev_queue_xmit(skb)` to send it;
- `dev_queue_xmit()` Queues a buffer for transmission to a network device.
- `dev_queue_xmit()` linearizes the buffer and do the checksum
- and calls `qdisc_enqueue_root()` that calls `sch->enqueue()`
- `sch` depends on the scheme you are using. Try `tc qdisc show`

Queuefull and queueless devices



Dev_queue_xmit()



Transmitting

Considering **fifo** scheme

```
struct Qdisc_ops pfifo_qdisc_ops __read_mostly = {
        .id             =       "pfifo",
        .priv_size      =       sizeof(struct fifo_sched_da
        .enqueue        =       pfifo_enqueue,
...
}
```

Transmitting

- `pfifo_enqueue()` calls `qdisc_enqueue_tail()` that enqueue the skb and return `NET_XMIT_SUCCESS`
- back to `dev_queue_xmit()`, it calls `qdisc_run()`
- `qdisc_run()` calls `__netif_schedule()`[1]
- `__netif_schedule` raises NET_TX_SOFTIRQ using `raise_softirq_irqoff(NET_TX_SOFTIRQ)`
- As expected `open_softirq(NET_TX_SOFTIRQ, net_tx_action);` was already called in `net_dev_init()` half century ago.

[1] Also call qdisc_run()

NET_TX_SOFTIRQ context

- net_tx_action() calls qdisc_restart which has

```
HARD_TX_LOCK(dev, txq, smp_processor_id());
if (!netif_tx_queue_stopped(txq) &&
    !netif_tx_queue_frozen(txq))
        ret = dev_hard_start_xmit(skb, dev, txq);
HARD_TX_UNLOCK(dev, txq);
```

dev_hard_start_xmit()

- it calls `rc = ops->ndo_start_xmit(skb, dev)`

```
static const struct net_device_ops e1000_netdev_ops = {
        .ndo_open               = e1000_open,
        .ndo_stop               = e1000_close,
        .ndo_start_xmit         = e1000_xmit_frame,
        .ndo_get_stats          = e1000_get_stats,
        .ndo_set_rx_mode        = e1000_set_rx_mode,
        .ndo_set_mac_address    = e1000_set_mac,
        .ndo_tx_timeout         = e1000_tx_timeout,
...
}
```

Entering into device driver functions

dev_hard_start_xmit()

- e1000_xmit_frame() calls e1000_tx_map() to set the TX DMA buffers
- then calls e1000_tx_queue()
- e1000_tx_queue() calls:

```
#define writel(val,addr) outl((val),(unsigned long)(addr))

writel(i, hw->hw_addr + tx_ring->tdt);
```

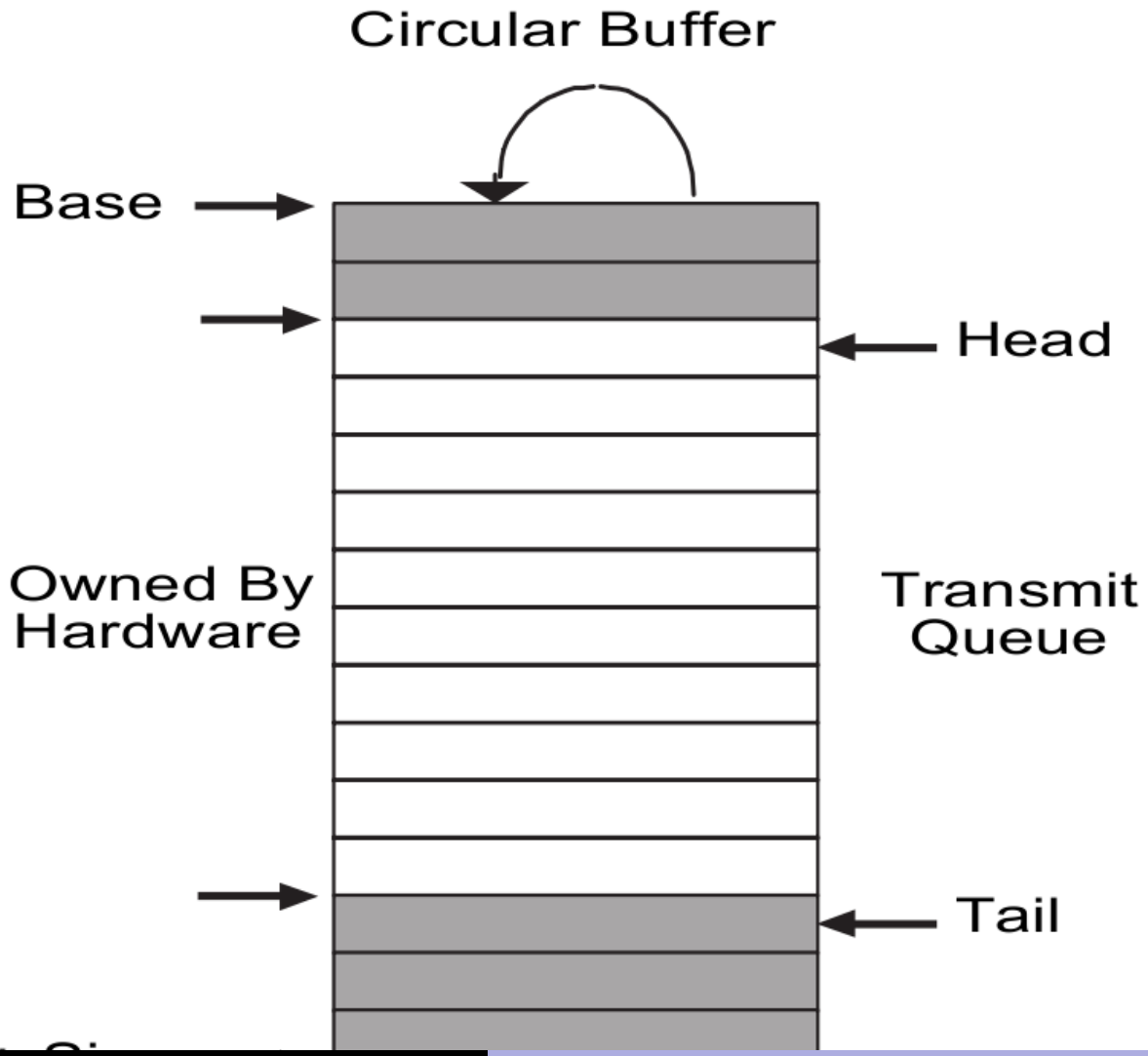
TX ring descriptor

```
struct e1000_tx_ring {
        /* pointer to the descriptor ring memory */
        void *desc;
        /* physical address of the descriptor ring */
        dma_addr_t dma;
        /* length of descriptor ring in bytes */
        unsigned int size;
        /* number of descriptors in the ring */
        unsigned int count;
        /* next descriptor to associate a buffer with */
        unsigned int next_to_use;
        /* next descriptor to check for DD status bit */
        unsigned int next_to_clean;
        /* array of buffer information structs */
        struct e1000_buffer *buffer_info;
```

e1000 spec

- Transmit Descriptor Head register (TDH) This register holds a value which is an offset from the base, and indicates the in–progress descriptor. There can be up to 64K descriptors in the circular buffer. Reading this register returns the value of "head" corresponding to descriptors already loaded in the output FIFO.
- Transmit Descriptor Tail register (TDT) This register holds a value which is an offset from the base, and indicates the location beyond the last descriptor hardware can process. This is the location where software writes the first new descriptor.

Kretprobe example



Thank you!

Doubts!?