A roaring journey through sk_buff and net_device

From Userspace through the Networking Subsystem into the Driver – and back again

Florian Westphal, Hagen Paul Pfeifer
fw@strlen.de — hagen@jauu.net

Hochschule Furtwangen,
Computer Networking – Fakultät Informatik
Furtwangen, Germany

Jan. 17th 2008
Preface

► Requirements:
  • Low latency
  • High throughput
  • Low CPU and memory utilization
  • Fair behavior against other protocols and components

► Driver specific code is based on e1000 adapter (exceptions are marked)

► No e1000 feature show today (sorry – that presentation was held last time ;-)

/* The code following below sending ACKs in SYN-RECV and TIME-WAIT states
   outside socket context is ugly, certainly. What can I do? */
NIC Initialization

- **Initialization**: `pci_register_driver()` → `e1000_probe()`
- `request_irq()` → registers IRQ handler
- `e1000_open()` *(called when if is made active)*
  1. Allocate transmit descriptors `e1000_setup_all_tx_resources()`
  2. Allocate receive descriptors `e1000_setup_all_rx_resources()`
  3. Power up `e1000_power_up_phy()`
  4. Tell firmware that we are the NIC is now open `e1000_get_hw_control()`
  5. Allocate interrupt `e1000_request_irq()`
  6. `e1000_configure_rx()`
- **BTW**: `SA_SAMPLE_RANDOM`
Virtual vs. Real Devices

▶ Each network device is represented by a instance of net_device structure

▶ Virtual Devices:
  - Build on top of a real (or virtual) device
  - Bonding, VLAN (802.1Q), IPIP, GRE, ...
  - Similar handling like real devices (register device et cetera)

▶ Real Devices:
  - RTL 8139/8169/8168/8101 ;-

▶ Mappings $n : m$
Frame Arrival – Hippie Revival

- **Interrupt Handler:** `e1000_intr() → __netif_rx_schedule()`
- Interrupt handler branch to arrival workmode
- Get RX ring address (and current offset) (`e1000_clean_rx_irq_PS()`)
- Get frame size and status from DMA buffer (`E1000_WRITE_REG, le32_to_cpu() and friends`)
- **Receive Checksum Offload** `e1000_rx_checksum()`
- Allocate new buffer: `dev_alloc_skb()` (non-NAPI)
- `skb_copy_to_linear_data`
- **Get protocol:** `eth_type_trans()` and update statistics
- `net/core/dev.c:netif_rx() → save data in CPU input queue (Limit: net.core.netdev_max_backlog) and netif_rx_schedule()`
- **NAPI:** `netif_rx_schedule()` and `netif_rx_schedule_prep()` directly
Frame Transmission

- hard_start_xmit(): driver/hardware specific network stack → Hardware entry point
- hard_start_xmit() → NETIF_F_LLTX (Duplicate Transmission Locking)
- e1000_xmit_frame
  1. tx_ring = adapter->tx_ring;
  2. Sanity checks (skb->len <= 0, adapter workarounds and friends)
  3. Count frags: count += TXD_USE_COUNT(len, max_txd_pwr); (thousands of errata)
4. **Flush** `e1000_tx_queue()`

```c
static void e1000_tx_queue(struct e1000_adapter *adapter,
    struct e1000_tx_ring *tx_ring, int tx_flags, int count)
{
    [...] while (count--)
    {
        buffer_info = &tx_ring->buffer_info[i];
        tx_desc = E1000_TX_DESC(*tx_ring, i);
        tx_desc->buffer_addr = cpu_to_le64(buffer_info->dma);
        tx_desc->lower.data = cpu_to_le32(txd_lower | buffer_info->length);
        tx_desc->upper.data = cpu_to_le32(txd_upper);
        if (unlikely(++i == tx_ring->count)) i = 0;
    }
    [...] writeln(i, adapter->hw.hw_addr + tx_ring->tdt);
    [...] }
```
Queuing Disciplines

- Each NIC has a assigned queuing discipline
- The egress queue is handled by tc
- L2 Congestion Management: **Ingress Path:** throtteling? → UDP? TCP? No (ECN? Maybe!)
### Protocol Support

<table>
<thead>
<tr>
<th>Protocol</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETH_P_IP</td>
<td>net/ipv4/ip_input.c:ip_rcv()</td>
</tr>
<tr>
<td>ETH_P_ARP</td>
<td>net/ipv4/arp.c:arp_rcv()</td>
</tr>
<tr>
<td>ETH_P_IPV6</td>
<td>net/ipv6/ip6_input.c:ipv6_rcv()</td>
</tr>
</tbody>
</table>
Software IRQ

- To delay work (IRQ handler isn’t the right place)
- Backlog queue per CPU
- CPU IRQ affinity
- After system call or IRQ handler returns
- Optimized for SMP/CMP systems
- NET_RX_SOFTIRQ
- 4 ? S< 0:00 [ksoftirqd/0]
NIC Data Mode: poll vs. interrupt

► Interrupt based:
  • NIC informs the driver if new data is available
  • Interrupt: new data, transmission failures and DMA transfer completed
    (e1000_clean_tx_irq())
  • Queues the frame for further processing

► Polling based:
  • Driver check the device constantly if new data is available
  • Pure polling is rare!

► Currently: NAPI (“interrupt-polled-driven”, “site:jauu.net filetype:pdf napi”)
Network Driver Principles

▶ Each device driver register themselves (register_netdevice(); linked list of network devices)

▶ include/linux/netdevice.h: struct net_device:
  • unsigned long mem_end, mem_start, base_addr, irq;
  • unsigned long state;
  • unsigned long features;
    • NETIF_F_SG, NETIF_F_HW_CSUM, NETIF_F_HIGHDMA, ...
  • int (*poll) (struct net_device *dev, int *quota);
  • struct Qdisc *qdisc;
  • int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
View From Userspace

- Socket Descriptor (int fd)
- can perform I/O on socket descriptor depending on socket state
- Various syscalls to create sockets/change state, etc
  - `socket()`, `listen()`, `connect()`, etc.
- Kernel keeps track of socket state
- *real* communication (the protocol itself) handled by kernel
- Kernel maps each process’ descriptor to a structure
How to tell if descriptor is a socket?

- `current->files`: open file table structure
- contains list of `struct file`
- if `(file->f_op == &socket_file_ops)` return `file->private_data`;
- `f_op/socket_file_ops`: `struct file_operations`
  - function pointers for `read, write, ioctl, mmap, open, ...` hence the name: all deal with file operations
  - `socket_file_ops` is the `file_operations` structure for sockets
  - if `file->f_op` is something other than the socket fops, this is not a socket ;)
- `->private_data` points to a socket structure
socket structure

- Represents a Socket

- identifies:
  - socket type (SOCK_STREAM, etc).
  - socket state (SS_CONNECTED, etc).

- contains pointers to various other structures, incl. proto_ops and struct sock

- also contains wait queue/wakelist, etc.
sock structure

- Network layer representation of sockets
- large structure (≈ 60 members)
- contains protocol id, packets send/receive queue heads, listen backlog, timers, peercred, ...
- also has some callbacks:

```c
void (*sk_state_change)(struct sock *sk);
void (*sk_data_ready)(struct sock *sk, int bytes);
void (*sk_write_space)(struct sock *sk);
void (*sk_error_report)(struct sock *sk);
int (*sk_backlog_rcv)(struct sock *sk, struct sk_buff *skb);
void (*sk_destruct)(struct sock *sk);
```
struct proto_ops

Recap:

- `fd` → `struct file`
- `struct file` has `f_ops` (== `socket_file_ops` in case of sockets)
- `struct file` also has a pointer to private data (which points to socket structure)
- `socket structure` has `struct sock` (see previous slide). Also has `proto_ops`.

`struct proto_ops` contains the (family dependent) implementation of socket functions: `bind`, `connect`, `setsockopt`, ...

Example (simplified):

```c
asmlinkage long sys_listen(int fd, int backlog) {
    struct socket *sock;

    sock = sockfd_lookup_light(fd, &err, &fput_needed);

    return sock->ops->listen(sock, backlog);
}
```
**struct proto**

- **struct proto**: socket layer → transport layer interface. Example:

  ```c
  struct proto tcp_prot = {
  .name = "TCP",
  .owner = THIS_MODULE,
  .close = tcp_close,
  .connect = tcp_v4_connect,
  };
  ```

- **struct inet_protosw**: transport → network interface. Example:

  ```c
  static struct inet_protosw inetsw_array[] = {
  {
  .type = SOCK_STREAM,
  .protocol = IPPROTO_TCP,
  };
  ```
AF_INET internals

net/ipv4/af_inet.c:

const struct proto_ops inet_stream_ops = {
        .family = PF_INET,
[..]
}

const struct proto_ops inet_dgram_ops = {
        .family = PF_INET,
[..]
}

Linux AF_INET implementation holds valid proto_ops inside an array. Assignment to sock structure depends on socket(2) arguments

static struct inet_protosw inetsw_array[] = {
        {
            .type = SOCK_STREAM, .protocol = IPPROTO_TCP,
            .prot = &tcp_prot, .ops = &inet_stream_ops,
[..]
Socket creation

Userspace does: `socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)`

- kernel allocates a new inode/socket. BTW: `grep sockfs /proc/filesystems`
- kernel sets `sock->type` as specified by User
- checks if family (=AF_INET in our case) is known
  `(net_proto_family[family] != NULL)`
- calls `net_proto_family[family] -> create`
  - create function must be implemented by all address families
  - address families register themselves at the socket layer at initialization
  - in our case create will be `inet_create()`
- `inet_create()` searches `inet_protosw inetsw_array[]` for the requested type/protocol pair
Socket creation (2)

- sets sock->ops and other values as specified in inetsw_array.
- allocates new struct sock (sk), records struct proto as specified in inetsw_array (in our case &tcp_prot)
- finally calls sk->sk_prot->init() (i.e. tcp_v4_init_sock, set in &tcp_prot)
  - sets TCP specific stuff: ssthresh, mss_cache, tcp_init_congestion_ops, etc.
From write to the wire...

Lets have a look what happens when data ist written to a socket via write.

- kernel looks up the corresponding struct file.
- we end up inside `vfs_write()`, which calls `file->f_op->aio_write()` (i.e. `socket_file_ops`)
- eventually we end up in `sock_sendmsg()`, which then calls `sock->ops->sendmsg` (i.e. `inet_protosw`s entry for `SOCK_STREAM/IPPROTO_TCP`: `inet_stream_ops`)

- now we are at the TCP level (`sock->ops->sendmsg` is `tcp_sendmsg`).
- will look at TCP state (connecting, being shut down, . . . )
- fetches a skb from write queue
- if no skb: allocate new one, or: `sk_stream_wait_memory()`
skbuffs

- struct skbuff: The most important data structure in the Linux networking subsystem.
- every packet received/sent is handled using the skbuff structure
- problems to solve:
  - Memory accounting.
  - Queueing of packets.
  - parsing of layer 2/3/4 protocol information.
  - insertion of additional headers at the beginning of packet, etc.
skbuff mapped to a packet

sk_buff

struct sk_buff *next;
struct sk_buff *prev;
[...
skb_data_t mac_header;
skb_data_t network_header;
skb_data_t transport_header;
skb_data_t tail;
skb_data_t end;
unsigned char *head;
unsigned char *data;
[...]

next/prev: List management (think ’receive/send queue this skb is on’)

skb_data_t: pointer or offset (unsigned int, 64 bit platforms)
struct sk_buff

- struct sock *sk;: An skb is mapped to a socket, e.g. for memory accounting
- ktime_t tstamp;: skb-timestamping (packet sniffer, TCP_CONG_RTT_STAMP, ...): net_timestamp(), i.e. normally unused
- struct net_device *dev;: interface skb arrived on/leaves by
- struct dst_entry *dst;: Destination cache/routing. Keeps track of pmtu and other properties; also deals with route (e.g. link down).
  - Destination cache/routing. Keeps track of pmtu and other properties; also deals with route (e.g. link down).
  - Has struct dst_ops which are implemented by each (network) protocol
- char cb[48]: e.g. TCP control block (sequence number, flag, SACK, ...)
- keeps track of total length, data length, cloned etc.
- optional pointers for _NF_CONTRACK, bridge, traffic shaping, ...
skb_headroom

> skb_headroom/_tailroom(): return number of bytes left at head/tail

> http://www.skbuff.net/skbbasic.html
skb_push/_pull

- skb_push: adjusts headroom for tailroom adjustment: skb_put/_trim
- http://www.skbuff.net/skbbasic.html
Sending a TCP frame

- recap: We are sending data via TCP, `tcp_sendmsg` has picked an skb to use.
- checks `skb_tailroom()`. If nonzero, calls `skb_add_data` which copies data from userspace into skbuff.
- if tailroom exhausted, use fragment list (`skb_shinfo(skb)->nr_frags`)
- if fraglist unusable (pageslots busy, !(sk->sk_route_caps & NETIF_F_SG)): push skb and alloc new segment
- eventually calls `tcp_write_xmit`
  - Does MTU probing (`tcp_mtu_probe`), depending on TCP state
  - takes first skb from send queue
    - calls `tcp_transmit_skb(skb, ...)` and advances `send_head`, i.e. ’packet is sent’.
    - `tcp_transmit_skb`: builds TCP header and hands skb to IP layer
      (ip_queue_xmit(), via `icsk->icsk_af_ops->queue_xmit(skb, ..)`)
Sending IP frame

- ip_queue_xmit(): make sure packet can be routed (sets skb->dst)
- Builds IP header
- Packet is handed to netfilter
- If everything ok: skb->dst->output(skb); (ip_output()).
  - sets skb->dev = skb->dst->dev
  - Packet is handed to netfilter (Postrouteing!), calls ip_finish_output if ok.
  - finally: dst->neighbour->output(...)
Almost done... need Layer 2 address

➤ Our journey through the protocol stack is almost done: net/ipv4/arp.c

```c
static struct neigh_ops arp-generic_ops = {
    .family = AF_INET,
[..]
    .output = neigh_resolve_output,
};

static struct neigh_ops arp-direct_ops = {
    .family = AF_INET,
    .output = dev_queue_xmit,

    skb_queue_tail(&neigh->arp_queue, skb) if NUD_INCOMPLETE
```
dev_queue_xmit

- has to linearize the skb, e.g. if device doesn’t support DMA from highmem and at least one page is highmem
- if device dev->qdisc != NULL, skb is enqueued now q->enqueue(skb, q);
- now a queue run is triggered (unless device is stopped...), eventually calls qdisc_restart()
  - dequeues skb from the qdisc, acquires per-cpu TX lock
  - ret = dev->hard_start_xmit

```c
switch (ret) {
    case NETDEV_TX_OK: /* Driver sent out skb successfully */
        [..]
    default: /* Driver returned NETDEV_TX_BUSY - requeue skb */
        ret = dev_requeue_skb(skb, dev, q);
```