Linux Device Drivers – network driver

Jernej Vičič

Overview

1. Introduction
2. SNULL
   - IP number assignment
   - Packet transfer
3. Kernel interface
   - Device registration
   - Device initialization
   - net_device structure
   - net_device struct
   - net_device struct
   - Opening and closing
   - Packet send

Introduction

- network world,
- network device,
- application similar to (mounted) block device,
- block device registers disks and methods,
    - receive
    - send
    - blocks
    - uses request function
- similar for network devices:
    - receive
    - send
    - packets

introduction, differences

- differences between mounted disks and packets delivery:
  - disk has a separate file in /dev
  - network device does not
  - network has its own namespace
  - network has its own set of operations

introduction, differences

- vsocket
- software object that is distinct from the interface
- imultiple functions read, write
- multiple sockets on single network device

introduction, differences

- block devices respond to kernel requests
- network devices get packages from the environment
- packages are sent to the kernel
- the kernel interface is designed differently
- network devices support administrative tasks
  - defining network addresses
  - changing the transfer parameters
  - traffic statistics and errors
- API reflects those differences

introduction

- completely protocol-independent
- valid for:
  - network protocols:
    - Internet protocol [IP],
    - IPX,
    - other,
  - hardware protocols:
    - Ethernet,
    - token ring,
    - other.
- communication between the driver and the kernel is done by individual packages,
- one network packet at a time.

introduction, terminology

- octet:
  - 8 bits,
  - the smallest data unit of network devices,
  - we almost never use bytes,
- header:
  - a set of octets,
  - attaches them to the package front (prepend),
  - attaches them at the routing between layers,
  - example of data flow TCP connection (next slide).

Introduction, TCP example

- data sent via TCP,
- network subsystem breaks data into packages,
- adds a TCP header:
  - describes where each package belongs,
- lower level adds IP header:
  - describes where the package will be sent,
- if packages travel through Ethernet (hardware):
  - we need an Ethernet header,
- drivers are not interested in higher level headers:
  - drivers provide (create, read) hardware heads,
  - in our example the Ethernet header.

SNULL

- *SNULL* – Simple Network Utility for Loading Localities,,
- driver of the network device,
- driver that does not talk to the "actual" devices,
- works like a loopback device,
- simulates actual operations,
- simulates communication with actual servers,
- does not send hardware requests.

SNULL

- only supports IP protocol,
- driver modifies the packets because there are no remote servers,
- so must know the protocol,
- changes content (changes source / target addresses, ...),

IP number assignment

- module produces two interfaces,
- what we send to one interface, it returns to the other interface,
- simulates the operation of two external links,
- definition of IP numbers for this is not enough,
- the kernel would find that both the source and the target are on this computer,
- it would do all the necessary operations without the driver,
- driver of the external address "area" and sends it to another interface,
- the destination address from the external address of the other interface.

IP number assignment

- kind of "hidden" loopback,
- turns on the least important bit of the third octet of the IP number and network (C class),
- packages sent to the network A:
  - connected to *sn0*,
  - first interface,
- appear:
  - as packages of the network B,
  - connected to *sn1*,
  - second interface,

IP number assignment

- *snullnet0* - the network associated with the *sn0* interface,
- *snullnet1* - the network associated with the *sn1* interface,
- addresses of these two networks should differ only in the least significant bit of the third octet,
- mask should be 24-bit,
- *local0*:
    - IP interface *sn0*,
    - belongs to the network *snullnet0*,
- *local1*:
    - IP interface *sn1*,
    - belongs to the network *snullnet1*,

IP number assignment

- IPs differ only in the least significant bit of the third and fourth octets,
- *remote0* - (virtual) computer on the network *snullnet0*:
  - fourth octet has to be the same as *local1*,
  - each packet sent to *remote0* arrives to *local1*,
  - package changes the head as if it came from the computer *remote0*,
- *remote1* - (virtual) computer on the network *snullnet1*:
  - fourth octet has to be the same as *local0*,
  - each packet sent to *remote1* arrives to emphlocal0,
  - package changes the head, like it came from the computer *remote1*,

IP number assignment

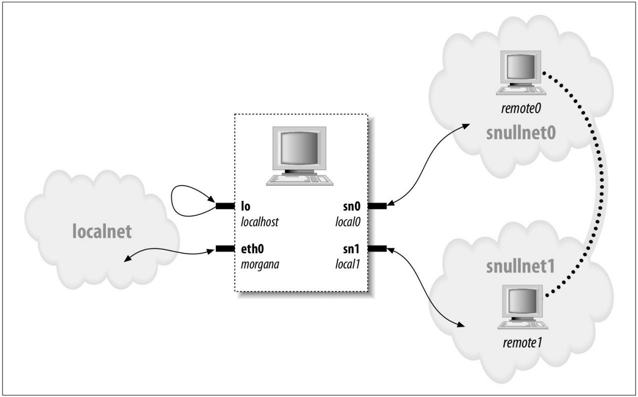

Figure: Interfaces.

IP number assignment

- Network numbers:
    - */etc/networks*,
    - *snullnet0 – 192.168.0.0*,
    - *snullnet1 – 192.168.1.0*,
- Computer numbers (hosts):
    - */etc/hosts*,
    - *192.168.0.1 – local0*,
    - *192.168.0.2 – remote0*,
    - *192.168.1.2 – local1*,
    - *192.168.1.1 – remote1*,

IP number assignment

```
ifconfig sn0 local0 netmask 255.255.255.0
ifconfig sn1 local1 netmask 255.255.255.0
```

IP number assignment

- ping *remote0* and *remote1*:

```
morgana% ping -c 2 remote0
64 bytes from 192.168.0.99: icmp_seq=0 ttl=64 time=1.6 ms
64 bytes from 192.168.0.99: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss

morgana% ping -c 2 remote1
64 bytes from 192.168.1.88: icmp_seq=0 ttl=64 time=1.8 ms
64 bytes from 192.168.1.88: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss
```

IP number assignment

- if we send a packet to a computer on a known network that the driver does not "translate"
- package appears on another interface, but it is ignored by the following:
  - package for 192.168.0.32,
  - goes to the *sn0* interface,
  - appears on *sn1*,
  - destination address 192.168.1.32,
  - packet is ignored.

Packet transfer

- driver emulates *Ethernet* protocol,
- 10base-T, 100base-T, ali Gigabit,
- *tcpdump* can be used,

Packet transfer

- *snull* works only with IP packets,
- corrupts all other packets,
- changes packet headers:
    - source,
    - destination,
    - checksum,

Kernel interface

- *loopback.c*,
- *plip.c*,
- *e100.c*,

Device registration

- we have no major/minor numbers,
- for each new interface sets the data structure,
- structure *net_device*,
- *snull* has pointers in two such structures: for *sn0* and *sn1*:

Device registration

- pointers to struct *net_device*:

```
struct net_device *snull_devs[2];
```

- structure contains *kobject*,
- reference-counted,
- exported via sysfs,
- we associate it dynamically with the function:

```
struct net_device *alloc_netdev(int sizeof_priv,
    const char *name,
        void (*setup)(struct net_device *));
```

Device registration

- *sizeof_priv* – size of private data area,
- *name* – interface name,
- *setup* – pointer to init function.

Device registration

- pointers to *net_device*:

```
snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn0",
      snull_init);
snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn1",
      snull_init);
if (snull_devs[0] = = NULL || snull_devs[1] = = NULL)
  goto out;
```

- helper functions (wrappers),

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

Device registration

- send struct to function *register_netdev*:

```
for (i = 0; i < 2; i++)
  if ((result = register_netdev(snull_devs[i])))
    printk("snull: error %i registering device %s",
        result, snull_devs[i]->name);
```

Device initialization

- mostly done in function *snull_init*:

```
ether_setup(dev); /* assign some of the fields */
dev->open = snull_open;
dev->stop = snull_release;
dev->set_config = snull_config;
dev->hard_start_xmit = snull_tx;
dev->do_ioctl = snull_ioctl;
dev->get_stats = snull_stats;
dev->rebuild_header = snull_rebuild_header;
dev->hard_header = snull_header;
dev->tx_timeout = snull_tx_timeout;
dev->watchdog_timeo = timeout;
/* keep the default flags, just add NOARP */
dev->flags |= IFF_NOARP;
dev->features |= NETIF_F_NO_CSUM;
dev->hard_header_cache = NULL; /* Disable caching */
```

Device initialization

- mostly set pointers to driver functions,
- *IFF_NOARP*:
  - interface can not use the ARP protocol,
  - Address Resolution Protocol (ARP),
  - IP addresses are translated into Ethernet medium access control (MAC) addresses,
  - *snull* does not need (does not send data),
- *hard_header_cache* - disables caching of ARP responses (since they are not),
- *tx_timeout, watchdog_timeo* - timeout when downloading.

Device initialization

- *priv* field in struct *net_device*:

```
struct snull_priv *priv = netdev_priv(dev);
```

- direct access in not OK,
- use *netdev_priv*,

priv and snull

- *priv* field in struct *net_device*:

```
struct snull_priv {
  struct net_device_stats stats;
  int status;
  struct snull_packet *ppool;
  struct snull_packet *rx_queue; /* List of incoming packet
  int rx_int_enabled;
  int tx_packetlen;
  u8 *tx_packetdata;
  struct sk_buff *skb;
  spinlock_t lock;
};
```

Unloading

```
void snull_cleanup(void)
{
  int i;
  for (i = 0; i < 2; i++) {
    if (snull_devs[i]) {
      unregister_netdev(snull_devs[i]);
      snull_teardown_pool(snull_devs[i]);
      free_netdev(snull_devs[i]);
    }
  }
  return;
}
```

net_device structure

- fields can be divided into 3 groups:
    - Global Information,
    - Hardware Information, low-level hardware information,
    - Interface Information, information about the interface.

net_device struct

- some functions can be NULL,
- some functions can be omitted, *ether_setup* takes care,
- lists all the functions of the network driver:

```
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 (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned
      short type, void *daddr, void *saddr, unsigned len);
```

net_device struct

```
int (*rebuild_header)(struct sk_buff *skb);

void (*tx_timeout)(struct net_device *dev);


struct net_device_stats *(*get_stats)(struct net_device *dev);


int (*set_config)(struct net_device *dev, struct ifmap *map);
```

- other functions are optional.

net_device struct

```
//gonilik ju spreminja ob vsakem začetku prenosa paketa in ob vsakem
//prejetem paketu; trans_start se uporablja za določitev smrtnih objemov pri
//pošiljanju, last_rx se ne uporablja (prihodnost)
unsigned long trans_start;
unsigned long last_rx;

//najmanjši čas, ki mora preteči, da se prenosni nivo odloči in pokliče
//funkcijo tx_timeout
int watchdog_timeo;

//ekvivalent filp->private_data, dostopa se prek etdev_priv
void *priv;

//podpora multicast
struct dev_mc_list *mc_list;
int mc_count;

//preprečuje simultane klice gonilnikove funkcije hard_start_xmit
spinlock_t xmit_lock;
int xmit_lock_owner;
```

Opening and closing

- the kernel opens/closes an interface when this function is called *ifconfig*,
- open:
- when ifconfig iz used to implement a new address:
  - specifies the address with the call ioctl(SIOCSIFADDR) (Socket I/O Control Set Interface Address),
  - puts the IFF_UP bit into the dev-> flag with ioctl(SIOCSIFFLAGS) (Socket I/O Control Set Interface Flags),
  - last command "starts" the interface (turn on),
- close:
- when we use ifconfig down:
  - calls ioctl(SIOCSIFFLAGS), which cleans the IFF_UP bit,
  - calls the *stop* method.

Opening and closing

- similar tasks as a character driver:
  - *open* - requires system resources and tells the interface to boot,
  - *sopt* - stops the interface and releases system resources,
- network drivers also take care of:
  - copies the MAC address from the device to $dev->dev\_addr$,
  - start interface queue for downloads:

```
void netif_start_queue(struct net_device *dev);
```

Function open (snull)

```
int snull_open(struct net_device *dev)
{
  /* request_region( ), request_irq( ), .... (like fops->open) */
  /*
  * Assign the hardware address of the board: use "\0SNULx", where
  * x is 0 or 1. The first byte is ’\0’ to avoid being a multicast
  * address (the first byte of multicast addrs is odd).
  */
  memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
  if (dev = = snull_devs[1])
    dev->dev_addr[ETH_ALEN-1]++; /* \0SNUL1 */
  netif_start_queue(dev);
  return 0;
}
```

Function stop (snull)

```
int snull_release(struct net_device *dev)
{
  /* release ports, irq and such -- like fops->close */
  netif_stop_queue(dev); /* can’t transmit any more */
  return 0;
}
```

- *netif_stop_queue* – mark the device as unable to send packets,

Packet send

- the most important tasks: sending and receiving packages,
- "Transmission refers to the act of sending a packet over a network link",
- when the kernel wants to send the package, call *hard_start_transmit*,
- load packet to queue,
- each package is stored in the socket buffer structure *struct sk_buff)*,,
- name originates from network connection abstraction - *socket*,
- connection may have nothing to do with sockets,

Packet send

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
  int len;
  char *data, shortpkt[ETH_ZLEN];
  struct snull_priv *priv = netdev_priv(dev);
  data = skb->data;
  len = skb->len;
  if (len < ETH_ZLEN) {
    memset(shortpkt, 0, ETH_ZLEN);
    memcpy(shortpkt, skb->data, skb->len);
    len = ETH_ZLEN;
    data = shortpkt;
  }
  dev->trans_start = jiffies; /* save the timestamp */
  /* Remember the skb, so we can free it at interrupt time */
  priv->skb = skb;
  /* actual deliver of data is device-specific, and not shown here */
  snull_hw_tx(data, len, dev);
  return 0; /* Our simple device can not fail */
}
```

Packet send

- true data transfer is "hidden" in the special function *snull_hw_tx*,
- this function only checks the package,
- if everything is OK, call the *snull_hw_tx* function.
- if the package is smaller than the minimum allowed value,
- add "zero" - *zero padding*,
- many drivers have problems in this regard (memory leak),
- minimum length - 60 bytes.

Packet send

- “hardware-related” transmission function *snull_hw_tx*,
- omitted, too dependent on hardware (Ethernet),
- we will look at the exercises (again, only briefly).

Packet receive

- harder than transmitting,
- we allocate *sk_buff*,
- send to 1 layer higher,
- two ways to receive packages:
  - *interrupt driven* - most drivers,
  - *polled* - rare, high-bandwidth adapters,

Packet receive

- separate parts that are dependent on hardware,
- function *snull_rx* is called from the interrupt handler snull,
- it is called after the hardware has already received the package,
- package is already in memory,
- *snull_rx* gets a pointer to the data and length of the package,
- only sends the package to higher levels,
- sends additional information at this time.

Packet receive

```
void snull_rx(struct net_device *dev, struct snull_packet *pkt)
{
  struct sk_buff *skb;
  struct snull_priv *priv = netdev_priv(dev);
  /*
  * The packet has been retrieved from the transmission
  * medium. Build an skb around it, so upper layers can handle it
  */
  skb = dev_alloc_skb(pkt->datalen + 2);
  if (!skb) {
    if (printk_ratelimit( ))
      printk(KERN_NOTICE "snull rx: low on mem - packet dropped\n");
    priv->stats.rx_dropped++;
    goto out;
  }
  memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
  /* Write metadata, and then pass to the receive level */
  skb->dev = dev;
  skb->protocol = eth_type_trans(skb, dev);
  skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
  priv->stats.rx_packets++;
  priv->stats.rx_bytes += pkt->datalen;
  netif_rx(skb);
out:
  return;
}
```

Packet receive

- allocate buffer for packet *dev_alloc_skb*,
- function *printk_ratelimit* returns 0, if too much text was written to console,
- otherwise the sysstem could hang,
- packet is copyed to buffer with *memcpy*,
- packet delivery information:
  - *skb_put* change *end-of-data pointer* in buffer and return pointer to new space,
  - $skb- > protocol = eth\_type\_trans(skb, dev);$
  - $skb- > ip\_summed = CHECKSUM\_UNNECESSARY;$
  - $priv- > stats.rx\_packets{+}{+};$
  - $priv- > stats.rx\_bytes += pkt- > datalen;$
  - $netif\_rx(skb);$

Changes in connection state

- network connection is part of the external environment,
- we have no control over external factors,
- the network subsystem needs to know when the link goes up / down),
- the network subsystem offers some functions for providing information,

Changes in connection state

- *carrier state* - the presence means that the hardware is ready for work,
- if someone pulls out the cable, the carier disappears, the connection goes "down",
- driver can explicitly specify/test carrier state:

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);

int netif_carrier_ok(struct net_device *dev);
```

MAC address definition

- Ethernet,
- medium access control (MAC),
- MAC is unique number for interface,
- 3 usages:
  - ARP with Ethernet,
  - ARP over Ethernet,
  - headers that are not Ethernet.

MAC address definition

- ARP – Address Resolution Protocol (ARP).
- supported by kernel,
- the driver does nothing, only helps the kernel in creation of physical layer header (Ethernet).

Changes in connection state

- retrieves information from the kernel and forms an Ethernet header,

```
int snull_header(struct sk_buff *skb, struct net_device *dev,
    unsigned short type, void *daddr, void *saddr,
    unsigned int len)
{
  struct ethhdr *eth = (struct ethhdr *)skb_push(skb,ETH_HLEN);
  eth->h_proto = htons(type);
  memcpy(eth->h_source, saddr ? saddr : dev->dev_addr, dev->addr_len);
  memcpy(eth->h_dest, daddr ? daddr : dev->dev_addr, dev->addr_len);
  eth->h_dest[ETH_ALEN-1] ^= 0x01; /* dest is us xor 1 */
  return (dev->hard_header_len);
}
```

Statistics

- method *get_stats*,
- returns a pointer to device statistics:

```
struct net_device_stats *snull_stats(struct net_device *dev
{
  struct snull_priv *priv = netdev_priv(dev);
  return &priv->stats;
}
```

- returns a struct *net_device_stats*.

Statistics

```
unsigned long rx_packets;
unsigned long tx_packets;
//število vseh uspešnih paketov (sprejetih in oddanih)

unsigned long rx_bytes;
unsigned long tx_bytes;
//število vseh bajtov  (sprejetih in oddanih)

unsigned long rx_errors;
unsigned long tx_errors;
//število vseh napačno oddanih/sprejetih paketov

unsigned long rx_dropped;
unsigned long tx_dropped;
//število zavrženih paketov (dropped)

unsigned long collisions;
//število vseh trkov zaradi zastojev na mediju

unsigned long multicast;
//število vseh sprejetih multicast paketov
```

Ethtool

- a tool for reviewing network devices,
- controls various interface parameters:
  - speed,
  - media type,
  - duplex operation,
  - DMA ring setup,
  - hardware checksumming,
  - wake-on-LAN operation,
  - etc.,

Ethtool

```
sudo ethtool eth0

Settings for eth0:
  Supported ports: [ TP ]
  Supported link modes:   10baseT/Half 10baseT/Full
                          100baseT/Half 100baseT/Full
                          1000baseT/Full
  Supports auto-negotiation: Yes
  Advertised link modes:  10baseT/Half 10baseT/Full
                          100baseT/Half 100baseT/Full
                          1000baseT/Full
  Advertised pause frame use: No
  Advertised auto-negotiation: Yes
  Link partner advertised link modes:  Not reported
  Link partner advertised pause frame use: No
  Link partner advertised auto-negotiation: No
  Speed: 100Mb/s
  Duplex: Full
  Port: Twisted Pair
  PHYAD: 1
  Transceiver: internal
  Auto-negotiation: on
  MDI-X: off
  Supports Wake-on: pumbag
  Wake-on: g
  Current message level: 0x00000001 (1)
```