Linux-Internals

Interprozess- und Netzwerkkommunikation
(Kernel 2.4)

Stephan Uhlmann <su@su2.info>
15.02.2002

Interprozesskommunikation

1. Primitive Kommunikation
   (a) Lock files
   (b) File locks
2. Pipes
3. Named Pipes
4. System V IPC
   (a) Semaphoren
   (b) Message queues
   (c) Shared memory
5. IPC mit Sockets

Lock files

- benutzt bestehende Mechanismen des VFS
- Prozess legt lock file an, um Belegung einer Ressource zu signalisieren
- Beispiel: Locking des Mailspools durch SMTP-/POP3-Server
- 1:n Beziehung
- für Datenaustausch ungeeignet

File locks

- benutzt bestehende Mechanismen des VFS
- Prozess lockt existierende Datei, um Belegung einer Ressource zu signalisieren
- gelockte Datei kann die gemeinsam genutzte Ressource sein (Datenaustausch)
- `flock()` (4.4BSD), `lockf()` (POSIX) und `fnctl()` (beides).
- 1:n Beziehung
- für Datenaustausch geeignet

Pipes

- `ls -la | more`
- `int pipe(int filedes[2]);` -> Inode mit 2 Dateideskriptoren (lesen&schreiben)
- Shell: erzeugt Pipe und 2x `fork()`
- Kindprozesse kopieren per `dup2(int oldfd, int newfd)` die jeweiligen Pipe-Dateidesktiptoren auf die Standardeingabe (fd 0) bzw. Standardausgabe (fd 1)
- Kindprozesse starten die Programme (ls, more)

Pipes

- im Kernel: im VFS ein `inode` und zwei `file` Objekte

- `inode->u` verweist auf ein `struct pipe_inode_info` [include/linux/pipe_fs_i.h]

```
struct pipe_inode_info {
  wait_queue_head_t wait;
  char *base;
  unsigned int len;
  unsigned int start;
  unsigned int readers;
  unsigned int writers;
  unsigned int waiting_readers;
  unsigned int waiting_writers;
  unsigned int r_counter;
  unsigned int w_counter;
};
```

Pipes

- Zugriff über normale `read()` und `write()` Systemaufrufe
- Im Kernel: `pipe_read()` und `pipe_write()` [fs/pipe.c]
- POSIX: bis 4 kB (`PIPE_BUF` [include/linux/limits.h]) atomar
- Pipe-Dateideskriptoren können von mehreren Prozessen genutzt werden, effektiv aber nur Datenaustausch zwischen 2 Prozessen -> 1:1

Named Pipes

- Nachteile Pipes: Prozesse müssen vom selben Elternprozess abstammen, Lebensdauer beschränkt auf die Lebensdauer der Prozesse
- `mkfifo Name`, Systemaufruf: `mknod()`, Kernel: Device anlegen
- `ls -la > fifo`
- `more fifo`
- Zugiff: wie normale Pipes
- auch hier effektiv nur 1:1 Kommunikation

System V IPC

- aus System V Unix Variante (<->BSD), kein POSIX Standard

- erlauben Rechtevergabe

```
struct kern_ipc_perm {
  key_t key;    /* Schlüssel */
  uid_t uid;    /* UID Eigentümer */
  gid_t gid;    /* GID Eigentümer */
  uid_t cuid;   /* UID Erzeuger */
  gid_t cgid;   /* GID Erzeuger */
  mode_t mode;  /* Zugriffsmodus */
  unsigned long seq;  /* Sequenznummer */
};
```

Semaphoren

- Speicherbereich der von Prozessen atomar abgefragt und verändert werden kann

- `int semget ( key_t key, int nsems, int semflg`) -> erzeugt Semaphoren-Array (`struct sem_array` [include/linux/sem.h])

```
struct sem_array {
  struct kern_ipc_perm sem_perm;        /* permissions .. see ipc.h */
  time_t sem_otime;                     /* last semop time */
  time_t sem_ctime;                     /* last change time */
  struct sem *sem_base;                 /* ptr to first semaphore in array */
  struct sem_queue *sem_pending;        /* pending operations to be processed */
  struct sem_queue **sem_pending_last;  /* last pending operation */
  struct sem_undo *undo;                /* undo requests on this array */
  unsigned long sem_nsems;              /* no. of semaphores in array */
};
```

Semaphoren

- eine einzelne Semaphore wird durch die `struct sem` implementiert:

```
struct sem {
int semval; /* current value */
int sempid; /* pid of last operation */
};
```

- Zugriff über den Systemaufruf

```
int semop ( int semid, struct sembuf *sops,
unsigned nsops )
```

- eine Operation auf einer Semaphore:

```
struct sembuf {
unsigned short sem_num; /* semaphore index
in array */
short sem_op; /* semaphore operation */
short sem_flg; /* operation flags */
};
```

- “undo”-Funktion: negierte Operationen in der `struct sem_undo`

- n:m Kommunikationsbeziehung, aber kein Datenaustausch

Message queues

- Verschicken von “Nachrichten”

- mit Typkennzeichnung

- Erzeugen einer message queue mit dem Systemaufruf
  `int msgget ( key_t key, int msgflg )`

- Im Kernel: `struct msg_queue` [ipc/msg.c]

```
struct msg_queue {
  struct kern_ipc_perm q_perm;
  time_t q_stime;          /* last msgsnd time */
  time_t q_rtime;          /* last msgrcv time */
  time_t q_ctime;          /* last change time */
  unsigned long q_cbytes;  /* current number of bytes on queue */
  unsigned long q_qnum;    /* number of messages in queue */
  unsigned long q_qbytes;  /* max number of bytes on queue */
  pid_t q_lspid;           /* pid of last msgsnd */
  pid_t q_lrpid;           /* last receive pid */

  struct list_head q_messages;
  struct list_head q_receivers;
  struct list_head q_senders;
};
```

Message queues

- Elemente von `q_messages`:
```
struct msg_msg {
struct list_head m_list;
long m_type;
int m_ts; /* message text size */
struct msg_msgseg* next;
/* the actual message follows immediately
*/
};
```

- `ssize_t msgrcv ( int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg )`

- `int msgsnd ( int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg )`

Message queues

- eine Nachricht:
```
struct msgbuf {
long mtype; /* type of message */
char mtext[1]; /* message text */
};
```

- Zugriff auf eine message queue zwar von mehreren Prozessen möglich, Datenaustausch jedoch 1:1, gelesene Nachrichten werden sofort aus der message queue entfernt

Shared memory

- gemeinsame Speicherbereiche verschiedener Prozesse
- schnellste Form von IPC
- keine Synchronisation
- Erzeugen eines shares memory Bereichs:
  `int shmget(key_t key, int size, int shmflg)`
- im Kernel: `struct shmid_kernel` [ipc/shm.c]

```
struct shmid_kernel /* private to the kernel */
{
  struct kern_ipc_perm shm_perm;
  struct file * shm_file;
  int id;
  unsigned long shm_nattch;
  unsigned long shm_segsz;
  time_t shm_atim;
  time_t shm_dtim;
  time_t shm_ctim;
  pid_t shm_cprid;
  pid_t shm_lprid;
};
```

Shared memory

- `shm_file`: eine “unlinked” (nicht sichtbare) Datei im FS (shmfs gemountet unter /dev/shm/)

- Zugriff über:
  ```
  void *shmat ( int shmid, const void *shmaddr,
  int shmflg )
  int shmdt ( const void *shmaddr)
  ```
  Zuordnen des shared memory Segments zum eigenen Datensegment (“attach”)

- shared memory kann von beliebig vielen Prozessen benutzt werden -> n:m Kommunikationsbeziehung

ipcs / ipcrm

```
han:~ # ipcs
------ Shared Memory Segments --------
key        shmid  owner   perms  bytes    nattch status
0x00000000 512    root    600    1056768   3      dest
0x00000000 513    wwwrun  600    46084     3      dest
0x74b8f640 514    oracle  640    374599680 14
0xd06f7024 515    oracle  640    255062016 10
0x91b0e5f8 516    oracle  640    143319040 11
------ Semaphore Arrays --------
key        semid  owner   perms  nsems  status
0x06999a36 1536   oracle  640     154
0x052b3e16 3073   oracle  640     154
0x0699aa36 4610   oracle  640     154
------ Message Queues --------
key        msqid  owner   perms  used-bytes   messages
han:~ #
```

IPC mit Sockets

- spezielle Variante der Sockets: Unix-Domain-Sockets
- gleiche Arbeitsweise wie normale Sockets
- `AF_UNIX` o. `AF_LOCAL` beim `socket()` Aufruf angeben
- Bei `connect()` Angabe eines `struct sockaddr_un` statt normalerweise `struct sockaddr`

IPC mit Sockets

- [include/linux/un.h]

```
#define UNIX_PATH_MAX 108

struct sockaddr_un {
sa_family_t sun_family; /* AF_UNIX */
char sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

- Dateiname repräsentiert den Socket im Dateisystem

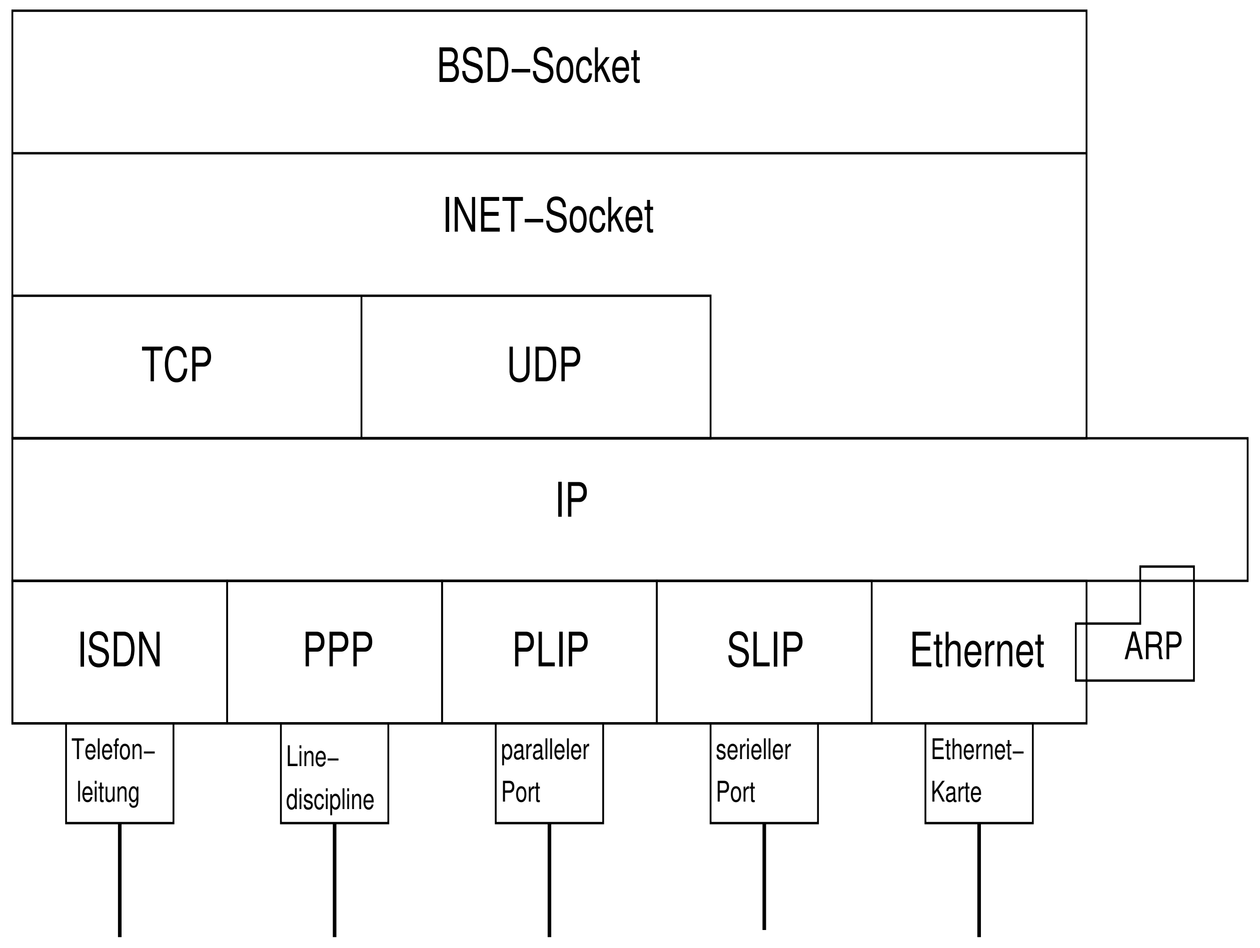
- Socket kann von vielen Prozessen genutzt werden, Datenaustausch jedoch nur 1:1

Zusammenfassung der Kommunikationsbeziehungen

IPC-Art	Beziehung	Datenaustausch
Lock files	1:n	nein
File locks	1:n	ja
Pipes	1:1	ja
Named Pipes	1:1	ja
Semaphoren	n:m	nein
Message queues	1:1	ja
Shared memory	n:m	ja
Sockets	1:n	ja

Netzwerkkommunikation

- BSD-Socket
- INET-Socket
- TCP/UDP
- IP
- Netzwerkgeräte


BSD–Socket
INET–Socket
TCP
UDP
IP
ISDN
PPP
PLIP
SLIP
Ethernet
ARP
Telefon–
leitung
Line–
discipline
paralleler
Port
serieller
Port
Ethernet–
Karte

BSD-Socket

- Schnittstelle für Programmierer: C-Bibliothek
  int socket(int domain, int type, int protocol);

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
int listen(int s, int backlog);
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
int send(int s, const void *msg, size_t len, int flags);
int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
int recv(int s, void *buf, size_t len, int flags);
int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

BSD-Socket

- ein Socket im Kernel: `struct socket` [include/linux/net.h]

```
struct socket {
  socket_state state;
  unsigned long flags;
  struct proto_ops *ops;
  struct inode *inode;
  struct fasync_struct *fasync_list;
  struct file *file;
  struct sock *sk;
  wait_queue_head_t wait;
  short type;
  unsigned char passcred;
};
```

```
struct sk_buff {
  struct sk_buff * next;       /* Next buffer in list */
  struct sk_buff * prev;       /* Previous buffer in list */
  struct sk_buff_head * list;  /* List we are on */
  struct sock *sk;             /* Socket we are owned by */
  struct timeval stamp;        /* Time we arrived */
  struct net_device *dev;      /* Device we arrived on/are leaving by */

  /* Transport layer header */
  union {
    struct tcphdr *th;
    struct udphdr *uh;
    struct icmphdr *icmph;
    struct igmphdr *igmph;
    struct iphdr *ipiph;
    unsigned char *raw;
  } h;

  /* Network layer header */
  union {
    struct iphdr *iph;
```

```
    struct ipv6hdr *ipv6h;
    struct arphdr *arph;
    struct ipxhdr *ipxh;
    unsigned char *raw;
  } nh;

  /* Link layer header */
  union {
    struct ethhdr *ethernet;
    unsigned char *raw;
  } mac;
 ...
  struct dst_entry *dst;

  unsigned int truesize;  /* Buffer size */
  unsigned char *head;    /* Head of buffer */
  unsigned char *data;    /* Data head pointer */
  unsigned char *tail;    /* Tail pointer */
  unsigned char *end;     /* End pointer */
 ...
};
```

BSD-Socket

- die in [net/socket.c] implementierten Funktionen greifen über den `ops` Eintrag des `struct socket` auf die Funktionen der INET-Socket Schicht zu

INET-Socket

- AF_INET Adressfamilie (TCP/IP)

- wichtigste Struktur: `struct sock` [include/net/sock.h] (gekürzt)

```
struct sock {
  __u32 daddr;
  __u16 dport;
  __u32 saddr;
  __u16 sport;
  struct sk_buff_head receive_queue;
  struct sk_buff_head write_queue;
struct proto *prot;
 union {
    struct tcp_opt af_tcp;
    struct raw_opt tp_raw4;
    struct raw6_opt tp_raw;
    struct spx_opt af_spx;
  } tp_pinfo;
  union {
    struct unix_opt af_unix;
    struct inet_opt af_inet;
    struct atalk_sock af_at;
    struct ipx_opt af_ipx;
    ...
  } protinfo;
  ...
};
```

INET-Socket

- Funktionen der AF_INET Adressfamilie implementiert in [net/ipv4/af_inet.c]

```
inet_stream_connect(struct socket *sock, struct sockaddr * uaddr, int addr_len, int flags);
inet_dgram_connect(struct socket *sock, struct sockaddr * uaddr, int addr_len, int flags); extern inet_accept(struct socket *sock,
inet_listen(struct socket *sock, int backlog);
inet_accept(struct socket *sock, struct socket *newsock, int flags);
inet_recvmsg(struct socket *sock, struct msghdr *ubuf, int size, int flags, struct scm_cookie *scm);
inet_sendmsg(struct socket *sock, struct msghdr *msg, int size, struct scm_cookie *scm);
inet_shutdown(struct socket *sock, int how);
inet_setsockopt(struct socket *sock, int level, int optname, char *optval, int optlen);
inet_getsockopt(struct socket *sock, int level, int optname, char *optval, int *optlen);
```

INET-Socket

- diese Funktionen werden aus der BSD-Socket Schicht aus aufgerufen
- rufen ihrerseits über den `prot` Eintrag der `struct sock` Funktionen der jeweiligen Transportschicht auf

```
struct proto {
  int (*connect)(struct sock *sk, struct sockaddr *uaddr, int addr_len);
  int (*setsockopt)(struct sock *sk, int level, int optname, char *optval, int optlen);
  int (*getsockopt)(struct sock *sk, int level, int optname, char *optval, int *option);
  int (*sendmsg)(struct sock *sk, struct msghdr *msg, int len);
  int (*recvmsg)(struct sock *sk, struct msghdr *msg, int len, int noblock, int flags, int *addr_len);
  int (*bind)(struct sock *sk, struct sockaddr *uaddr, int addr_len);
...
};
```

TCP/UDP

- Transportschicht
- transportiert “Nachrichten” nach Art des 4.4BSD message passing system: `struct msghdr` [include/linux/socket.h]

```
struct msghdr {
 void * msg_name;        /* Socket name */
 int msg_namelen;        /* Length of name */
 struct iovec * msg_iov; /* Data blocks */
 __kernel_size_t msg_iovlen;  /* Number of blocks */
 void * msg_control;     /* Per protocol magic (eg BSD file descriptor passing) */
 __kernel_size_t msg_controllen;  /* Length of cmsg list */
 unsigned msg_flags;
};
```

TCP/UDP

- Funktionen verarbeiten diese "Nachrichten"
  zu Pakete der `struct sk_buff`

- implementiert in [net/ipv4/udp.c], [net/ipv4/tcp.c],
  [net/ipv4/tcp_v4.c]

```
int udp_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
int tcp_sendmsg(struct sock *sk, struct msghdr *msg, int size)
int udp_sendmsg(struct sock *sk, struct msghdr *msg, int len)
int tcp_recvmsg(struct sock *sk, struct msghdr *msg, int len, int nonblock, int flags, int *addr_len)
int udp_recvmsg(struct sock *sk, struct msghdr *msg, int len, int nonblock, int flags, int *addr_len)
```

- benutzen Funktionen der IP-Schicht

IP

- allg. “Paketübermittlungsdienst”
- Paket (`struct sk_buff`) & Adresse -> Übermittlung
- keine Garantie, dass das Paket ankommt (->TCP)
- wichtige Struktur: `struct iphdr` [include/linux/ip.h]
- Unterteilung: Empfangen, Weiterleiten, Senden von IP-Paketen

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
  __u8 ihl:4,
  version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
  __u8 version:4,
  ihl:4;
#else
#error "Please fix <asm/byteorder.h>"
#endif
  __u8 tos;
  __u16 tot_len;
  __u16 id;
  __u16 frag_off;
  __u8 ttl;
  __u8 protocol;
  __u16 check;
  __u32 saddr;
  __u32 daddr;
  /*The options start here. */
};
```

IP - Empfangen:

1. Erhalt des Pakets
   `int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)`
   [net/ipv4/ip_input.c], Empfang eines `sk_buff`
   vom Netzwerkgerät `dev`

2. Überprüfen des IP-Headers
   RFC 1122:
   1. angegebene Länge des Paktes gültig?
   2. IP-Versionsnummer korrekt?
   3. IP Checksumme ok?

3. Vergleich der Zieladresse mit der eigenen Adresse
   `ip_rcv_finish(struct sk_buff *skb)`:

`ip_route_input()` ?
lokaler Rechner: `int ip_local_deliver(struct sk_buff *skb)`
sonst: `int ip_forward(struct sk_buff *skb)`

4. Defragmentierung des IP-Pakets
   Fragment? -> zusammen mit anderen Fragmenten defragmentieren
   fehlen Fragmente? -> Fragmentliste
   `struct sk_buff *ip_defrag(struct sk_buff *skb)`
   [net/ipv4/ip_fragment.c]

5. Übergabe des Pakets an das nächste Protokoll
   `ip_rcv_finish(): skb->dst->input(skb)` (Funktions-Zeiger der `input` Funktion der höheren Transportschicht.

IP - Weiterleiten

`int ip_forward(struct sk_buff *skb)`
[net/ipv4/ip_forward.c]

1. Überprüfen des `ttl` Feldes im IP-Header
   <=1 ? -> Paket verwerfen, ICMP-Nachricht an Absender

2. Überprüfen des Pakets auf fehlerhaftes/ungewolltes Routing
   (z.B: "strict routing" aber Ziel kein Gateway)

3. Kopieren des Paket-Headers
   copy on write des `sk_buff` Headers (`skb_cow()` [include/linux/skbuff.h])

4. Setzen/Ändern von IP Optionen
   Aufruf `void ip_forward_options(struct sk_buff *skb)` [net/ipv4/ip_options.c]

5. Senden
   Aufruf `int ip_send(struct sk_buff *skb)` (als `static inline` in [include/net/ip.h] implementiert).
   MTU Grösse überschritten? -> fragmentieren (`ip_fragment()`)
   dann per `int ip_finish_output(struct sk_buff *skb)` verschicken [net/ipv4/ip_output.c]

IP - Senden

1. Erhalt des Pakets

   (z.B. durch Aufruf der Funktion `int ip_queue_xmit(struct sk_buff *skb)` [net/ipv4/ip_output.c])

2. Ermitteln der Ziel-Route

   Aufruf `ip_route_output()` [include/net/route.h]
   -> Netzwerkgerät über das das Paket verschickt werden soll

3. Füllen des IP-Headers

   `skb_push()` [include/linux/skbuff.h] macht am Anfang des `sk_buff` Platz für IP-Header ->

füllen mit entsprechenden Werten (Quell- und Ziel-Adresse)

4. Fragmentierung des IP-Pakets

   `int ip_queue_xmit2(struct sk_buff *skb)`:
   MTU Grösse überschritten? -> `int ip_fragment()`
   [net/ipv4/ip_output.c]

5. Setzen der IP-Checksumme

   `void ip_send_check(struct iphdr *iph)` [net/ipv4/output.c]

6. Übergabe an das entsprechende Netzwerkgerät

   Aufruf `skb->dst->output(skb)` -> `dev_queue_xmit()`
   des Netzwerkgerätes

Netzwerkgeräte

- hardwareunabhängige Schnittstelle
- `struct net_device` [include/linux/netdevice.h] verwaltet ein abstaktes Netzwerkgerät
- Name (eth0), IRQ / IO-Adresse, Hardwareadresse (MAC bei Ethernet), Multicast-, Broadcastadresse, statistische Angaben (# empfangener & gesandter Pakete)

Netzwerkgeräte

- Funktionszeiger zur hardwarespezifischen Implementation:
```
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);
```

- werden von den hardwareunabhängigen Funktionen [net/core/dev.c] benutzt

- Bsp.: `void dev_queue_xmit(struct sk_buff *skb, struct net_device *dev)`: überprüft ob Netzwerkgerät gerade beschäftigt ist, überträgt Paket entweder sofort per `hard_start_xmit()` oder trägt es in Liste der noch zu sendenden Pakete des `struct net_device` ein

- ”Linux-Kernel-Programmierung”, Beck, Böhme, Dziadzka, Kunitz, Magnus, Verworner, Addison Wesley, ISBN 3-8273-1144-6
- ”The Linux Kernel”, David A Rusling, `http://www.linuxdoc.org/LDP/tlk/`
- ”Linux Kernel 2.4 Internals”, Tigran Aivazian, `http://www.moses.uklinux.net/patches/lki.html`
- ”Interprocess Communication in Unix”, John Shapley Gray, Prentice Hall, ISBN 0-13-186891-8
- Linux IP Networking, Glenn Herrin, `http://kernelnewbies.org/documents/ipnetworking/linuxipnetworking.html`
- ”Understanding the Linux Kernel”, Daniel P. Bovet, Marco Cesati, O’Reilly