Linux-Internals
Interprozess- und Netzwerkkommunikation
(Kernel 2.4)

Stephan Uhlmann <su@su2.info>

15.02.2002

Inhaltsverzeichnis

1 Interprozesskommunikation

Why do you insist on utilizing this primitive linguistic communication? Your android brain is capable of so much more.
– Borg Queen, Star Trek: First Contact

Um Programmen im "Userspace" (ausserhalb des Kernels, normale Prozesse) Möglichkeiten zur Synchronisation und zum Datenaustausch zu bieten, muss das Betriebssystem verschiedene Möglichkeiten der Interprozesskommunikation (IPC) anbieten. Dies ist unter anderem notwendig, um Zugriffe auf gemeinsame Ressourcen zu koordinieren. Im Folgenden die Möglichkeiten die sich unter Linux bieten.

1.1 Primitive Kommunikation

Man kann schon bestehende Mechanismen des Virtual File Systems (VFS) verwenden, um sehr einfache Methoden der Interprozesskommunikation zu realisieren. Dies ist eine sehr unelegante Methode, wird jedoch trotzdem oft verwendet, da es leicht zu verstehen und implementieren ist.

1.1.1 Lock Files

Lock files sind Dateien die von einem Prozess angelegt werden, um anderen Prozessen zu signalisieren, dass er eine bestimmte Ressource belegt. Die anderen Prozesse warten so lange bis das lock file wieder verschwindet und legen dann selber wieder ein lock file an um die Ressource zu belegen.

Diese Methode wird z.B. häufig von SMTP und POP3 Servern benutzt, um den Zugriff auf das Mail Spool der verschiedenen Benutzer auf einem System zu synchronisieren.

Es besteht eine 1:n Beziehung zwischen den kommunizierenden Prozessen. Ein Prozess signalisiert durch setzen des lock files die Benutzung der Ressource und beliebig viele Prozesse können dies abfragen. Lock files sind nicht dazu geeignet Daten zwischen Prozessen auszutauschen.

1.1.2 File Locks

Das VFS bietet Möglichkeiten, Dateien durch sogenannte file locks zu sperren. Prozesse können sich nun bei der Nutzung einer gemeinsamen Ressource synchronisieren, indem sie ähnlich zu der Methode mit den lock files, wo die Datei ständig neu angelegt und wieder gelöscht wird, die eine Datei sperren und wieder entsperren. Dazu dienen die Systemaufrufe `flock()` (4.4BSD), `lockf()` (POSIX) und `fnctl()` (beides).

Häufig ist die gelockte Datei, die gemeinsam genutzte Ressource. Denn z.B. ist es auch möglich, zusätzlich zur Synchronisation, auch Datenaustausch zwischen den Prozessen zu betreiben, indem ein Prozess die Datei sperrt, Daten in sie hineinschreibt, wieder entsperrt und der nächste Prozess die Datei wieder sperrt, um die Daten auszulesen.

Es besteht eine 1:n Beziehung zwischen den kommunizierenden Prozessen. Ein Prozess signalisiert durch setzen des file locks die Benutzung der Ressource und beliebig viele Prozesse können dies abfragen. Es können Daten beliebiger Menge über die gemeinsam benutzte Datei ausgetauscht werden.

1.2 Pipes

Pipes stellen die klassische Methode der Interprozesskommunikation unter Unix dar, da sie sehr mächtig und einfach zu benutzen sind. Jedem dürfte eine Befehlszeile wie

```
ls -la | more
```

bekannt vorkommen. Die Shell startet die beiden Prozesse `ls` und `more` die über eine Pipe miteinander verbunden sind. `ls` schreibt seine Ausgabe in die Pipe und `more` liest sie daraus. Man kann theoretisch beliebig viele Prozesse auf diese Weise "aneinanderreihen".

Mit dem Systemaufruf `pipe()` erzeugt man eine Pipe.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Es wird ein temporärer Inode erzeugt und der Systemaufruf gibt zwei Dateideskriptoren zurück. `filedes[0]` zum Lesen und `filedes[1]` zum Schreiben. Auf diese Dateideskriptoren kann man wie auf eine normale Datei mit den Systemaufrufen `read()` und `write()` zugreifen.

Damit nun die beiden Prozesse `ls` und `more` mit einander kommunizieren können, müssen die beiden Prozesse jeweils die Dateideskriptoren des Ein-/Ausgabekanals der Pipe auf die Standardausgabe (Dateideskriptor 1) bzw. Standardeingabe (Dateideskriptor 0) kopieren. Dazu bedient man sich des Systemaufrufs `dup2()`. Ausgehend von der Shell passiert also folgendes:

1. Die Shell ruft `pipe()` auf und bekommt zwei Dateideskriptoren A und B zurück.
2. Die Shell führt für die beiden Programme zweimal `fork()` aus.
3. Die Shell schliesst wieder die beiden Dateideskriptoren A und B.

Der erste Kindprozess (für `ls`) tut folgendes:

1. Aufruf von `dup2(B,1)` um den schreibenden Dateideskriptor der Pipe auf den der Standardausgabe zu kopieren
2. Schliesst die von der Shell geerbten Dateideskriptoren A und B per `close()`.
3. Aufruf von `execve()` um das Programm `/bin/ls` auszufühen.

Der zweite Kindprozess (für `more`) tut analog folgendes:

1. Aufruf von `dup2(A,0)` um den lesenden Dateideskriptor der Pipe auf den der Standardeingabe zu kopieren.
2. Schliesst die von der Shell geerbten Dateideskriptoren A und B per `close()`.
3. Aufruf von `execve()` um das Programm `/bin/more` auszufühen.

Dies geschieht alles auf der Ebene der Systemaufrufe. Auf Kernelebene werden ein VFS `inode` Objekt und zwei `file` Objekte erzeugt (eins zum Lesen, eins zum Schreiben). Der `u` Eintrag des `inode` Objekts verweist auf eine Struktur `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;
};
```

`wait` ist eine Warteschlange der Prozesse, die darauf warten von der Pipe lesen zu können (z.B. weil gerade jemand etwas in die Pipe schreibt). `base` enthält die Startadresse der Speicherseite die für den Datenaustausch angelegt wurde, `start` den Startoffset des aktuell benutzen Bereichs der Speicherseite, `len` dessen Länge. Desweiteren gibt es noch diverse Zähler zur Synchronisation des Zugriffs auf diesen Inode von mehreren lesenden oder schreibenden Prozessen.

Der Zugriff auf eine Pipe kann von Prozessen im Userspace mit den normalen `read()` und `write()` Systemaufrufen passieren. Auf Kernelebene wird das durch die Funktionen `pipe_read()` und `pipe_write()` [fs/pipe.c] realisiert. Diese kopieren die Daten vom Userspace in die von der durch `pipe_inode_info` verwalteten Speicherseite in den Kernelspace und zurück. Sie kümmern sich auch um die Synchronisation und Fehlerbehandlung bei den verschiedensten Kombinationen von blockierenden und nicht-blockierenden Schreib- und Lesezugriffen.

[include/linux/limits.h] legt als Grösse für eine Pipe 4096 Bytes fest (PIPE_BUF), d.h. es können maximal 4 kB Daten "am Stück" in eine Pipe geschrieben werden. Nach POSIX Standard hat dies atomar zu erfolgen, d.h. ein Prozess der nur bis zu dieser Anzahl an Bytes Daten in eine Pipe schreibt, wird nicht unterbrochen und kann sicher sein, dass die Daten originalgetreu im Speicherbereich der Pipe angekommen sind.

Obwohl es theoretisch möglich ist, dass mehrere Prozesse den selben Dateideskriptor der Pipe benutzen, können Daten nur in 1:1 Beziehungen ausgetauscht werden. D.h. Daten die ein Prozess in die Pipe schreibt können nur von einem Prozess gelesen werden.

Pipes sind temporär, d.h. sie existieren nur solange einer der beteiligten Prozesse noch existiert. Möchte man eine Pipe über die Lebensdauer der Prozesse erhalten, nutzt man so genannte "Named Pipes".

1.3 Named Pipes

Named Pipes sind den normalen Pipes sehr ähnlich. Sie haben jedoch einen entscheidenden Vorteil: hiermit kann eine schon existierende Pipe von beliebigen Prozessen genutzt werden. Bei normalen Pipes müssen dazu die Prozesse vom selben Elternprozess abstammen (um an die Dateideskriptoren ranzukommen).

Man kann eine Named Pipe mit dem Befehl

```
mkfifo Name
```

erzeugen. Aufgrund des Befehls werden Named Pipes häufig auch als FIFOs bezeichnet. Dies ist zwar richtig, da sie nach dem FIFO (First In First Out) Prinzip funktionieren, jedoch arbeiten die normalen (temporären) Pipes nach dem selben Prinzip, würde also als unterscheidende Bezeichnung nicht passen.

Um das Beispiel von oben mit Named Pipes umzusetzen tut man folgendes (`fifo` ist die zuvor angelegte Named Pipe):

```
ls -la > fifo
```

Nun liest ein anderer Prozess die Daten aus der Named Pipe wieder aus:

```
more fifo
```

Die Named Pipe existiert weiter und kann weiter benutzt werden.

Im Gegensatz zu normalen Pipes wo in einem Schritt die Pipe erzeugt wird (ein `inode` Objekt, zwei `file` Objekte und die Speicherseite) und gleich zum Lesen und Schreiben geöffnet ist, werden Named Pipes von den Prozessen im Userspace geöffnet und geschlossen. Dabei beachtet Linux, dass es möglich ist, dass eine Pipe zum Lesen geöffnet wird bevor etwas hineingeschrieben wurde, sowie dass eine Pipe zum Schreiben geöffnet wird bevor sie auch zum Lesen geöffnet wurde.

Zum Anlegen einer Named Pipe bedient man sich des `mknod()` Systemaufrufs. Auf Kernelebene passiert dort das gleiche bei beim erzeugen einer Device Datei.

Das Lesen und Schreiben auf eine Named Pipe funktioniert auf die gleiche Weise wie bei normalen Pipes mit den Systemaufrufen `read()` und `write()` bzw. auf Kernelebene mit den Funktionen `pipe_read()` und `pipe_write()`.

Obwohl es auch hier es möglich ist, dass mehrere Prozesse die selbe Named Pipe benutzen, können Daten nur in 1:1 Beziehungen ausgetauscht werden. D.h. Daten die ein Prozess in die Pipe schreibt können nur von einem Prozess gelesen werden.

1.4 System V IPC

Eine der wichtigsten Varianten von Unix (neben BSD), das System V, definierte einige der klassischen Mittel der Interprozesskommunikation: Semaphore, Messagequeue und Shared Memory. Obwohl nicht im POSIX Standard enthalten, unterstützt Linux diese auch. Allen dreien ist gemeinsam, dass sie die Struktur `kern_ipc_perm` enthalten, in der die Zugriffsrechte auf das IPC Objekt verwaltet.

```
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 */
};
```

`key` ist eine einfache Nummer, die dynamisch beim Erstellen des IPC Objekts erzeugt wird und das IPC Objekt eindeutig referenziert. Die Sequenznummer wird zur Bestimmung der Objekt ID (s.u.) aus dem `key` verwendet.

Von Bedeutung ist unter Linux weiterhin der Systemaufruf

```
int ipc(unsigned int call, int first, int second, int third, void *ptr, long fifth)
```

Dieser ist dafür zuständig die IPC Funktionen (angegeben in `call`) in Funktionsaufrufe der jeweiligen Implementierungen im Kernel umzusetzen. Dies ist jedoch für normale Prozesse nicht sehr elegant und auch nicht portabel (da Linux-spezifisch). Deshalb benutzt man die jeweiligen Funktionen der C Bibliothek. Innerhalb des Kernels sind diese dann durch jeweilige `sys_` Aufrufe implementiert.

1.4.1 Semaphoren

Eine Semaphore ist im Prinzip ein Speicherbereich, dessen Wert von mehreren Prozessen ausgelesen und geändert werden kann. Dabei ist das Lesen und (in Abhängigkeit vom aktuellen Wert der Semaphore) Ändern des Wertes atomar, d.h. der Vorgang wird nicht von einem anderen Prozess unterbrochen, der evtl. zwischen dem Lese- und Schreibvorgang den Wert nochmals ändern könnte.

System V Semaphoren erweitern dieses Modell dahingehend, dass ein Array von Semaphoren erzeugt und durch eine einzige Operation verändert werden kann. Mit der Funktion

```
int semget ( key_t key, int nsems, int semflg )
```

werden `nsems` Semaphoren mit dem Schlüssel `key` erzeugt. Ist `key` noch nicht benutzt oder hat den besonderen Wert `IPC_PRIVATE`, so wird ein neues Array von Semaphoren angelegt und eine `semid` zurückgegeben, über die man Operationen auf diesem Semaphoren-Array ausführen kann. Referenziert `key` ein bereits existierendes Semaphoren-Array, so wird nur die damit assoziierte `semid` zurückgegeben.

Das Semaphoren-Array hat folgende Struktur [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 */
};
```

wobei `sem` folgende Struktur ist:

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

Wenn eine Operation nicht erfolgreich ausgeführt werden kann und der Prozess blockieren darf, so wird er in die `sem_pending` Queue eingereiht. In dieser werden alle Prozesse und deren Operationen auf dem Semaphoren-Array die noch auszuführen sind, gesammelt.

```
struct sem_queue {
  struct sem_queue * next;     /* next entry in the queue */
  struct sem_queue ** prev;    /* previous entry in the queue, *(q->prev) == q */
  struct task_struct* sleeper; /* this process */
  struct sem_undo * undo;      /* undo structure */
  int pid;                     /* process id of requesting process */
  int status;                  /* completion status of operation */
  struct sem_array * sma;      /* semaphore array for operations */
  int id;                      /* internal sem id */
  struct sembuf * sops;        /* array of pending operations */
  int nsops;                   /* number of operations */
  int alter;                   /* operation will alter semaphore */
};
```

System V Semaphoren erlauben es auch, dass bei Beenden eines Prozesses Operationen auf Semaphoren rückgängig gemacht werden. Dies soll sogenannte "deadlocks" verhindern. Dabei belegt ein Prozess per Semaphore eine Ressource, wird jedoch beendet bevor er die Ressource wieder freigeben kann und alle anderen Prozesse warten vergebens darauf, dass die Ressource durch die Semaphore wieder freigegeben wird.

Dazu verwaltet `task_struct` eine Liste mit "undo" Operationen, die auf die Semaphore angewandt werden, wenn der Prozess beendet wird.

```
struct sem_undo {
  struct sem_undo * proc_next; /* next entry on this process */
  struct sem_undo * id_next;   /* next entry on this semaphore set */
  int semid;                   /* semaphore set identifier */
  short * semadj;              /* array of adjustments, one per semaphore */
};
```

Wird der Wert einer Semaphore verändert, so wird der negative Wert der Änderung in dieser undo Liste gespeichert.

Ändern kann man den Wert einer Semaphore mit der Funktion

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

Auf das Semaphoren-Array, das durch `semid` referenziert wird, werden `nsops` Operationen ausgeführt. Beginnend ab der Adresse von `*sops` werden `nsops` Felder der Struktur `sembuf` ausgewertet.

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

`sem_num` ist der Index der zu verändernden Semaphore innerhalb des Semaphoren-Arrays. `sem_op` der Wert der auf den Wert der Semaphore (`semval`) addiert wird. Über das Flag `IPC_NOWAIT` im Feld `sem_flg` kann man das Blockieren des Prozesses bei der Ausführung der Operationen verhindern. Das Flag `SEM_UNDO` speichert die Operation in oben beschriebener undo-Liste, damit sie rückgängig gemacht werden kann, wenn der Prozess beendet wird.

`semop()` garantiert, dass entweder alle Operationen erfolgreich sind oder keine davon ausgeführt wird. Eine Operation ist als "erfolgreich" definiert, wenn der Wert der Semaphore nach der Operation grösser Null ist oder sowohl der aktuelle Wert der Semaphore als auch der Wert der Operation Null sind.

Semaphoren erlauben eine n:m Kommunikationsbeziehung zwischen Prozessen. Es können mehrere Prozesse gleichzeitig durch das Ändern der Semaphore vielen anderen Prozessen z.B. die Belegung/Freigabe einer Ressource signalisieren. Semaphoren eignen sich jedoch nicht für den Datenaustausch.

1.4.2 Message queues

Message queues, erlauben Prozessen Nachrichten zu schreiben, die von anderen Prozessen gelesen werden können. Diese Nachrichten sind einfach eine Folge von Bytes, können jedoch eine bestimmte Typkennzeichnung haben. Prozesse können den Empfang von Nachrichten auf einen bestimmten Typ einschränken. Die Grundlage der Implementierung unter Linux ist die Struktur `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;
};
```

Es ist wieder die Struktur `kern_ipc_perm` enthalten, welche die Zugriffsrechte regelt (s.o.). Neben weiteren Verwaltungsinformationen, werden drei Listen angelegt. `q_messages` ist die Liste der ungelesenen Nachrichten. Sie besteht aus Elementen der Struktur `msg_msg`.

```
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 */
};
```

`m_type` ist der Typ der Nachricht. `m_ts` die Grösse. `next` weist auf den nächsten Eintrag in der message queue.

```
struct msg_msgseg {
  struct msg_msgseg* next;
  /* the next part of the message follows immediately */
};
```

Die Daten der Nachricht selber, folgen immer direkt im Anschluss an das Feld `next`.

Die Liste `q_receivers` der `msg_queue` ist eine Liste der Prozesse die auf Daten in der message queue warten. Sie besteht aus Feldern der Struktur `msg_receiver`:

```
struct msg_receiver {
  struct list_head r_list;
  struct task_struct* r_tsk;
  int r_mode;
  long r_msgtype;
  long r_maxsize;
  struct msg_msg* volatile r_msg;
};
```

Der Prozess selber wird in der `task_struct* r_tsk` gespeichert. `r_msgtype` ist der Typ der Nachricht auf die der Prozess wartet, wobei der Wert Null bedeutet, dass der Prozess Nachrichten jeden Typs annimmt. `r_maxsize` ist die maximale Grösse der Nachricht in Bytes.

Die Liste `q_senders` der `msg_queue` ist eine Liste der Prozesse die darauf warten Daten in die message queue schreiben zu können, z.B. weil die message queue voll ist. Sie besteht aus Feldern der Struktur `msg_sender`:

```
struct msg_sender {
  struct list_head list;
  struct task_struct* tsk;
};
```

und diese besteht nur aus `task_struct`'s für die jeweiligen Prozesse.

Prozesse können mit der Funktion

```
int msgget ( key_t key, int msgflg )
```

eine message queue erzeugen. Ist `key` noch nicht benutzt oder hat den besonderen Wert `IPC_PRIVATE`, so wird eine neue messsage queue angelegt und eine `msqid` zurückgegeben, über die man Daten in die Queue schreiben oder von ihr lesen kann. Referenziert `key` eine bereits existierendes message queue, so wird nur die damit assoziierte `msqid` zurückgegeben.

Daten von einer message queue lesen, kann ein Prozess mit der Funktion

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

`msgid` ist der von `msgget()` zurückgegebene Wert. `msgp` ist die Adresse auf eine Struktur vom Typ `msgbuf` in der die Daten aus der message queue kopiert werden. `msgsz` ist die maximale Grösse der Nachricht. Ist die Nachricht grösser wird entweder eine Fehlermeldung zurückgegeben oder wenn das Flag `MSG_NOERROR` in `msgflg` gesetzt ist, die Nachricht abgeschnitten und der Rest geht verloren. `msgtyp` ist der Typ der zu empfangenden Nachricht. Wurde eine Nachricht von einem Prozess empfangen, wird sie aus der `q_messages` Liste entfernt.

Daten in eine message queue senden , kann ein Prozess mit der Funktion

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

`msgid` ist dabei wieder die ID der `message_queue`. `msgp` ein Zeiger auf eine Struktur `msgbuf` welche die Daten der Nachricht von der Grösse `msgsz` enthält. Über ein Flag `IPC_NOWAIT` kann der Prozess in `msgflg` signalisieren, dass er nicht blockieren möchte, wenn es im Moment nicht möglich ist die Nachricht zu senden (z.B. weil die message queue voll ist). Statt dessen kehrt `msgsnd()` dann gleich mit einer Fehlermeldung zurück.

`msgbuf` hat folgende Struktur:

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

Message queues erlauben zwar den Zugriff von mehreren Prozessen, die Daten können jedoch nur in 1:1 Beziehung ausgetauscht werden.

1.4.3 Shared Memory

Die dritte Variante der System V IPC ist das shared memory. Verschiedene Prozesse können einen gemeinsamen Speicherbereich nutzen und durch normale Maschinenbefehle zum Lesen und Schreiben Daten austauschen. Es ist die schnellste Form von IPC, da es keinerlei weitere Synchronisationsmechanismen gibt. Dies muss bei Bedarf "von Hand" implementiert werden (z.B. mit den schon erwähnten Semaphoren). Die Grundlage der Implementierung von shared memory unter Linux bildet die Struktur `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;
};
```

Wie schon bei den anderen System V IPC Varianten werden zuerst in `shm_perm` die Zugriffsrechte verwaltet. `shm_file` ist ein Eintrag zu einer "unlinked" Datei im shmfs (welches meist unter `/dev/shm` gemountet ist). "unlinked" bedeutet, sie ist zwar angelegt, belegt aber keinen `inode` und ist daher auch nicht im

Filesystem sichtbar. shm_nattach gibt die Anzahl der zugreifenden ("attached") Prozesse und shm_segsz die Segmentgrösse des Speicherbereichs aus. Danach folgen Timestamps der letzten Attach-, Detach- und Änderungsoperation sowie die PID des erzeugenden Prozesses und des Prozesses der zuletzt eine Operation auf den Speicherbereich ausgeführt hat.

Analog zu den anderen System V IPC Varianten kann ein Prozess mit der Funktion

```
int shmget(key_t key, int size, int shmflg)
```

ein neues shared memory Segment erzeugen oder sich die zu einem schon existierenden Segment mit dem Schlüssel key die passende shmid zurückgeben lassen.

Mit der Funktion

```
void *shmat ( int shmid, const void *shmaddr, int shmflg )
```

kann ein Prozess ein shared memory Segment mit der angegebenen shmid seinem eigenen Datensegment zuordnen. Entweder shmaddr ist 0, dann wird ein beliebiger freier Speicherbereich benutzt, oder die in shmaddr angegebene Adresse wird benutzt. In shmflg kann man über spezielle Flags die Wahl der Speicheradresse beeinflussen (z.B. Alignment auf eine bestimmte Blockgrösse).

Der Prozess kann nun durch Zugriff auf diesen "virtuellen" Speicherbereich in seinem Datensegment auf den shared memory Bereich zugreifen. Mit der Funktion

```
int shmdt ( const void *shmaddr)
```

kann man den Zugriff wieder aufheben.

Shared memory kann von mehreren Prozessen gleichzeitig genutzt werden und bietet daher eine n:m Kommunikationsbeziehung.

1.4.4 ipcs und ipcrm

Programme die System V IPC Mechanismen benutzen, können auch abstürzen. In diesem Falle werden die von ihnen belegten IPC Ressourcen nicht wieder freigegeben. Daher gibt es die beiden Tools ipcs und ipcrm, um IPC Ressourcen zu verwalten. Ersteres zeigt Informationen zu den aktuell belegten IPC Ressourcen an (soweit der Benutzer denn die entsprechenden Zugriffsrechte hat). Ein Beispiel der Ausgabe von ipcs auf einem Oracle-Server ist z.B.:

```
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:~ #
```

Mit `ipcrm` kann man nun bei Bedarf unter Angabe der jeweiligen id die IPC Ressource wieder freigeben.

Seit Linux Kernel 2.4 sind im Verzeichnis `/proc/sysvipc/` drei Dateien `sem`, `msg` und `shm` angelegt, aus denen sich die gleichen Informationen wie mit `ipcs` auslesen lassen.

1.5 IPC mit Sockets

Die bisher beschriebenen Möglichkeiten zur Interprozesskommunikation beschränkten sich auf die Kommunikation von Prozessen eines Rechners. Die Socket-Programmierschnittstelle ermöglicht es auch über Rechnergrenzen hinweg, über ein Netzwerk zu kommunizieren. Dies wird genauer im Abschnitt "Netzwerkkommunikation" beschrieben (auf Seite 17).

Eine spezielle Variante der Sockets, die Unix-Domain-Sockets, bieten jedoch auch auch eine einfache Möglichkeit zur lokalen Kommunikation innerhalb eines Rechners.

Unix-Domain-Sockets werden auf die gleiche Art und Weise erstellt und benutzt wie Sockets zur Netzwerkkommunikation. Man muss nur `AF_UNIX` bzw. `AF_LOCAL` als Adressfamilie beim `socket()` Aufruf angeben. Dann kann man mit Hilfe der Struktur `sockaddr_un` einen Dateinamen angeben, der den Socket im lokalen Dateisystem repräsentiert [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 */
};
```

Diese Struktur übergibt man dann dem `connect()` Aufruf als `struct sockaddr* serv_addr` wonach dann über den Dateideskriptor mit den üblichen Dateioperationen auf den Socket zugegriffen werden kann.

Auch wenn ein Socket von mehreren Prozessen genutzt werden kann, können Daten nur in 1:1 Beziehung über einen Socket ausgetauscht werden.

1.6 Zusammenfassung Kommunikationsbeziehungen

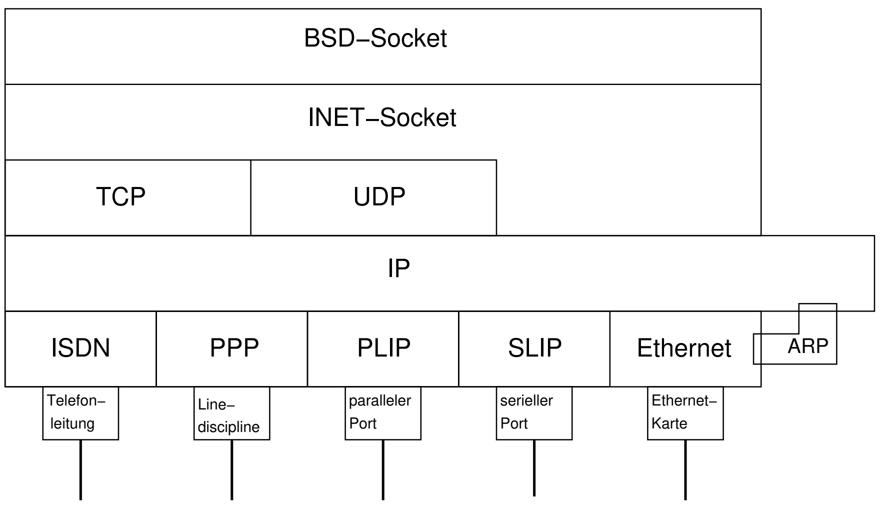
IPC-Art	Beziehung	Datenaustausch
Lock files	1:n	nein
File locks	1:n	ja
Pipes	1:1	ja
Named Pipes	1:1	ja
Semaphoren	n:m	nein
Message queues	1:1	ja
Shared memory	n:m	ja
Sockets	1:n	ja

2 Netzwerkkommunikation

Note that 120 sec is defined in the protocol as the maximum possible RTT.
I guess we'll have to use something other than TCP to talk to the University of Mars.
– aus net/ipv4/tcp_timer.c

Eines der Grundanforderungen an ein Betriebssystem ist es Netzwerkkommunikation zu unterstützen. Linux, selbst nur durch das Internet so schnell verbreitet und gewachsen, unterstützt das Internet Protocol (IP) seit ca. Version 0.96a. Die Netzwerkimplementation basiert auf 4.3 BSD Sockets mit einigen Erweiterungen.

Wie in anderen Bereichen des Linux Kernels, wurden auch bei der Netzwerkimplementation verschiedene Abstraktionsschichten eingeführt. Die folgende Darstellung verdeutlicht dies:



Folgend werden nun die einzelnen Schichten und ihre Implementation im Linux-Kernel erläutert. Jede Schicht interagiert dabei mit der Schicht über bzw. unter ihr, um die Daten zu transportieren.

Insgesamt wird darauf geachtet, dass die zu transportierenden Daten möglichst selten kopiert werden, da dies den Netzwerk-Datendurchsatz des Linux-Kernels entscheidend beeinflusst. Soll ein Datenpaket zwischen zwei Programmen der beiden Rechner A und B verschickt werden, werden diese Daten insgesamt nur viermal kopiert. Vom Userspace des Rechners A in den Kernelbereich, von dort zum Netzwerkgerät, dann vom Netzwerkgerät des Rechners B in dessen Kernelbereich und von dort in den Userspace des Programms auf Rechner B.

2.1 BSD-Socket

Als Programmierer benutzt man die Socket-Schnittstelle die in der C-Bibliothek zur Verfügung gestellt wird. Diese besteht aus den folgenden Funktionen:

```
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);
```

Erzeugt man mit `socket()` einen Socket, so gibt diese Funktion einen Dateideskriptor zurück über den man mit den anderen Funktionen den Socket benutzen kann. Es können auch normale I/O-Systemaufrufe wie `write()` und `read()` benutzt werden.

Diese Socket-Funktionen werden allein durch den Systemaufruf `sys_socketcall()` im Kernel implementiert [net/socket.c]:

```
asmlinkage long sys_socketcall(int call, unsigned long *args)
```

Dies ist eine der typischen Multiplexer-Systemaufrufe. `call` gibt an welcher Funktionsaufruf genau gemeint ist, gefolgt von der dynamischen Liste an Parametern. `sys_socketcall()` ruft dann die entsprechenden `sys_` Funktionen (z.B. `sys_socket()`) mit den passenden Parametern auf. Diese Funktionen bilden die oberste Schicht der Netzwerkimplementation im Linux-Kernel: die BSD-Socketschicht.

Eine der wichtigsten Strukturen innerhalb des Kernels ist die `struct socket` [include/linux/net.h]:

```
struct socket {
  socket_state state;
```

Hier wird der aktuelle Zustand des Sockets gespeichert. Z.B.: `SS_CONNECTED` oder `SS_UNCONNECTED` [include/linux/net.h].

```
  unsigned long flags;
```

Flags die bestimmte Eigenschaften des Sockets beschreiben (z.B. asynchron, nicht blockierend).

```
    struct proto_ops *ops;
```

Enthält für Sockets der `AF_INET` Adressfamilie spezifische Operationen (als Funktionspointer).

```
    struct inode *inode;
    struct fasync_struct *fasync_list;
    struct file *file;
```

Werden für Sockets der `AF_UNIX` Adressfamilie benötigt, um einen Socket einer Inode/Datei im Filesystem zuzuordnen.

```
    struct sock *sk;
```

Unterstruktur für die `AF_INET` Adressfamilie.

```
    wait_queue_head_t wait;
    short type;
```

Gibt den Typ des Sockets an. Gültige Werte sind `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP) und `SOCK_RAW` (IP) [include/asm/socket.h].

```
    unsigned char passcred;
  };
```

Eine weitere wichtige Struktur ist die `struct sk_buff` welche zur Verwaltung einzelner Datenpakete verwendet wird ([include/linux/skbuff.h], Auszugsweise da sehr umfangreich):

```
  struct sk_buff {
    /* These two members must be first. */
    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;
    struct spxhdr *spxh;
    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 */

...

};
```

Diese Struktur verwaltet einen allgemeinen Puffer, der bei der Übertragung von Daten zwischen den einzelnen Schichten verwendet wird. Dazu wird unter anderem das physikalische Netzwerkgerät in einem `struct net_device` gespeichert über das das Paket übertragen wird. Weiterhin werden jeweils ein Pointer auf die jeweiligen Header-Strukturen des transport, network und link layers innerhalb des Puffers gespeichert. `dst_entry` ist eine Struktur die das Ziel des Pakets definiert [include/net/dst.h].

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

2.2 INET-Socket

Die INET-Socket-Schicht verwaltet die netzwerkspezifischen Teile des Sockets für die `AF_INET` Adressfamilie. Hier ist unter anderem die Implementation der TCP/IP Protokoll-Suite in Linux festgeschrieben. Die Struktur `struct sock` [include/net/sock.h] bildet die Grundlage der Implementation. Da diese Struktur hier im einzelnen zu lang und zu detailliert wäre, seien hier nur die wichtigsten Elemente (stark gekürzt) erklärt:

```
struct sock {

  __u32 daddr;
  __u16 dport;
  __u32 saddr;
  __u16 sport;
```

Quell- und Zieladresse, sowie Portnummer.

```
  struct sk_buff_head receive_queue;
```

Liste von `sk_buff` (s.o.) Strukturen der auf den Socket ankommenden Pakete.

```
  struct sk_buff_head write_queue;
```

Liste von `sk_buff` (s.o.) Strukturen der vom den Socket zu sendenden Pakete.

```
struct proto *prot;
```

Struktur mit Zeigern auf die einzelnen Funktionen der Transport-Schicht.

```
 union {
    struct tcp_opt af_tcp;
    struct raw_opt tp_raw4;
    struct raw6_opt tp_raw;
    struct spx_opt af_spx;
  } tp_pinfo;
```

Struktur mit den Informationen zum jeweiligen Transport Protokoll. Bei TCP (`struct tcp_opt`) z.B. die Sequenznummern, diverse Zähler (z.B. für "slow start" und "congestion control"), Timestamps und statistische Daten.

```
  union {
    void *destruct_hook;
    struct unix_opt af_unix;
    struct inet_opt af_inet;
    struct atalk_sock af_at;
    struct ipx_opt af_ipx;
    x25_cb *x25;
    struct pppox_opt *pppox;
    struct netlink_opt *af_netlink;
    struct atm_vcc *af_atm;
    struct irda_sock *irda;
    struct wanpipe_opt *af_wanpipe;
    ...
  } protinfo;
```

Hier werden private Daten der jeweiligen Adressfamilie des benutzten Protokolls in der "network" Schicht gespeichert.

```
  ...
};
```

In [net/ipv4/af_inet.c] sind die einzelnen Funktionen der INET Adressfamilie implementiert. Dies sind im wesentlichen:

```
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);
```

Diese Funktionen werden von den Funktionen der BSD-Socketschicht aufgerufen und rufen ihrerseits wieder über den `prot` Eintrag in der `sock` Struktur die Funktionen der entsprechenden Transportschicht (TCP/UDP) auf.

Ein Auszug aus der Struktur `struct proto` welche die Zeiger auf diese Funktionen bereithält:

```
struct proto {
  void (*close)(struct sock *sk, long timeout);
  int (*connect)(struct sock *sk, struct sockaddr *uaddr, int addr_len);
```

```
  int (*disconnect)(struct sock *sk, int flags);
  struct sock * (*accept) (struct sock *sk, int flags, int *err);
  int (*ioctl)(struct sock *sk, int cmd, unsigned long arg);
  int (*init)(struct sock *sk);
  int (*destroy)(struct sock *sk);
  void (*shutdown)(struct sock *sk, int how);
  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);
...
};
```

2.3 TCP/UDP

Aufgerufen durch die Funktionen der INET-Schicht erledigen die verschiedenen TCP- und UDP-Funktionen die Arbeit für die Transport-Schicht im Netzwerkschichtenmodell.

Von besonderer Bedeutung ist hier die Struktur `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;
};
```

Sie implementiert eine allgemeine "Nachricht" nach Art des 4.4BSD message passing system.

Im folgenden die wichtigsten Funktionen der TCP/UDP Implementierungen.

`int udp_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)` [net/ipv4/udp.c]

`int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)` [net/ipv4/tcp_ipv4.c]

`int tcp_sendmsg(struct sock *sk, struct msghdr *msg, int size)` [net/ipv4/tcp.c]

`int udp_sendmsg(struct sock *sk, struct msghdr *msg, int len)` [net/ipv4/udp.c]

`int tcp_recvmsg(struct sock *sk, struct msghdr *msg, int len, int nonblock, int flags, int *addr_len)` [net/ipv4/tcp.c] `int udp_recvmsg(struct sock *sk, struct msghdr *msg, int len, int nonblock, int flags, int *addr_len)` [net/ipv4/udp.c]

Diese Funktionen verarbeiten die Nachrichten zu Paketen von `struct sk_buff` und rufen die entsprechenden Funktionen der IP-Schicht auf. Bei TCP werden vorher die bekannten Features zur verbindungsorientierten Kommunikation implementiert.

2.4 IP

Die IP-Schicht stellt einen allgemeinen "Paketübermittlungsdienst" zur Verfügung. Man übergibt ihr ein Paket und einen Adressaten und sie kümmert sich um die Übermittlung. Es wird nicht garantiert, dass das Paket auch ankommt. Dafür sind dann übergeordnete Schichten wie z.B. TCP zuständig.

Als wichtigste Datenstruktur der Implementierung der IP-Schicht im Linux Kernel ist die `struct iphdr` [include/linux/ip.h]:

```
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. */
};
```

Man kann die Aufgaben der IP-Schicht in das Empfangen, das Weiterleiten und Senden von IP-Paketen unterteilen. Dabei sieht der Ablauf wie folgt aus (alles IPv4):

Empfangen:

1. Erhalt des Pakets

   Mit der Funktion
   `int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)`
   [net/ipv4/ip_input.c] empfängt die IP-Schickt einen Puffer `skb` vom Netzwerkgerät `dev` der darunter liegenden Schicht.

2. Überprüfen des IP-Headers

   RFC 1122 konform wird in der obigen Funktion geprüft ob:
   1. die angegebene Länge des Paktes gültig ist (mind. die Länge eines IP Headers)
   2. die IP-Versionsnummer korrekt ist (IPv4 == 4)
   3. die IP Checksumme stimmt

3. Vergleich der Zieladresse mit der eigenen Adresse

   Ist das Paket nicht für den lokalen Rechner, dann wird es weitergeleitet. (In Funktion
   `ip_rcv_finish(struct sk_buff *skb)`, Abfrage von
   `ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev)`
   Ist es für den lokalen Rechner wird
   `int ip_local_deliver(struct sk_buff *skb)`
   aufgerufen, sonst
   `int ip_forward(struct sk_buff *skb)`
   [net/ipv4/ip_forward.c]

4. Defragmentierung des IP-Pakets

   Wurde das Paket nicht weitergeleitet, wird es per `ip_local_deliver()` auf dem Host weiterverarbeitet. Ist das Paket ein IP-Fragment, wird versucht mit Hilfe schon empfangender Pakete es zu defragmentieren. Ist das noch nicht möglich (weil noch Fragmente fehlen), dann wird das Paket in einer Fragmentliste gespeichert. Dies ist durch die Funktion `struct sk_buff *ip_defrag(struct sk_buff *skb)` [net/ipv4/ip_fragment.c] implementiert.

5. Übergabe des Pakets an das nächste Protokoll

   Die Funktion `ip_rcv_finish()` ruft die Funktion `skb->dst->input(skb)` auf, welche vorher auf den Funktions-Zeiger der entsprechenden `input` Funktion der höheren Transportschicht gesetzt wurde.

Weiterleiten:

1. Überprüfen des `ttl` Feldes im IP-Header

   Ist der Wert kleiner oder gleich 1, wird das Paket verworfen und eine entsprechende ICMP-Nachricht an den Absender geschickt.
   (in `int ip_forward(struct sk_buff *skb)` [net/ipv4/ip_forward.c])
   Wirklich dekrementiert wird das TTL Feld erst später.

2. Überprüfen des Pakets auf fehlerhaftes/ungewolltes Routing

   (zum Beispiel wenn "strict routing" verlangt wird und das Ziel kein Gateway ist)

3. Kopieren des Paket-Headers

   Da im folgenden der IP-Header verändert wird, muss der Header des `sk_buff` kopiert werden (copy on write, Funktion `skb_cow()` in [include/linux/skbuff.h]).

4. Setzen/Ändern von IP Optionen

   In `int ip_forward_finish(struct sk_buff *skb)` [net/ipv4/ip_forward.c]
   Aufruf von
   `void ip_forward_options(struct sk_buff *skb)` [net/ipv4/ip_options.c]

5. Senden

   In `int ip_forward_finish(struct sk_buff *skb)` [net/ipv4/ip_forward.c]
   Aufruf von
   `int ip_send(struct sk_buff *skb)` (als `static inline` in [include/net/ip.h] implementiert).
   Dort wird vorher überprüft ob das Paket die MTU Grösse überschritten hat, und dann entweder per
   `int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))`
   fragmentiert oder gleich per
   `int ip_finish_output(struct sk_buff *skb)`
   verschickt. (beide Funktionen implementiert in [net/ipv4/ip_output.c])

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 der Funktion `ip_route_output(&rt, daddr, sk->saddr, RT_CONN_FLAGS(sk), sk->bound_dev_if)` [include/net/route.h] um das Netzwerkgerät zu ermitteln, über welches das Paket verschickt werden soll.

3. Füllen des IP-Headers

   Mit Hilfe der Funktion `unsigned char *skb_push(struct sk_buff *skb, unsigned int len)` [include/linux/skbuff.h] wird am Anfang des `sk_buff` Puffers Platz für einen IP-Header geschaffen. Dieser wird dann mit den entsprechenden Werten wie Quell- und Ziel-Adresse gefüllt.

4. Fragmentierung des IP-Pakets

   In `int ip_queue_xmit2(struct sk_buff *skb)` wird überprüft ob das Paket die MTU Grösse überschritten hat. Ist das der Fall, dann wird es mit Hilfe der Funktion `int ip_fragment(struct sk_buff *skb, int (*output)(struct sk_buff*))` [net/ipv4/ip_output.c] in mehrere Teile zerlegt.

5. Setzen der IP-Checksumme

   Aufruf der Funktion `void ip_send_check(struct iphdr *iph)` [net/ipv4/output.c], welche die Checksumme berechnet und im IP-Header einträgt.

6. Übergabe an das entsprechende Netzwerkgerät

   Durch den Aufruf von `skb->dst->output(skb)` wird die `dev_queue_xmit()` des Netzwerkgerätes aufgerufen, welches sich dann um das Verschicken des Paktes kümmert.

2.5 Netzwerkgeräte

Da unterschiedliche Hardware zur Vernetzung von Rechnern eingesetzt werden kann, gibt es in Linux eine abstrakte Schnittstelle zur Netzwerkhardware. Dadurch können die oberen Netzwerkschichten unabhängig von der eingesetzten

Hardware implementiert werden. Die Datenstruktr `struct net_device` verwaltet solch ein abstraktes Netzwerkgerät [include/linux/netdevice.h]. Diese Datenstruktur enthält neben dem Namen (z.B. eth0) und Angaben zur Ansteuerung wie IRQ und IO-Adresse, auch Adressen wie die Broadcast-Adresse, Hardware-Adresse (MAC-Adresse bei Ethernet-Karten) und Multicast-MAC-Adressen sowie etliche statistische Angaben (Anzahl empfangender/gesandter Pakete).

`struct net_device` enthält weiterhin Pointer auf die Funktionen mit denen man das Gerät ansprechen kann. Die wichtigsten sind:

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

Initialisiert der Netzwerkgerät. Setzt z.B. die Adressen.

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

Öffnet das Netzwerkgrät, so dass Pakete übertragen werden können

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

Beendet die Übertragung von Paketen. Setzt alle Adressen auf `NULL`.

```
int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
```

Hardwareabhänige Funktion zum senden eines Pakets.

Diese Funktionen werden von den hardwareunabhägingen API Funktionen in [net/core/dev.c] benutzt. Zum Beispiel die Funktion `void dev_queue_xmit(struct sk_buff *skb, struct net_device *dev)` überprüft ob das Netzwerkgerät gerade beschäftigt ist und überträgt das Paket dann entweder sofort per `hard_start_xmit()` oder trägt es in eine Liste der noch zu sendenden Pakete innerhalb des `struct net_device` ein.

3 Referenzen

- "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

4 GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See Copyleft - GNU Project - Free Software Foundation (FSF) `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.