Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Zusammenfassung

Inhaltsverzeichnis

1 Einführung

- Entwicklung des Linux-Kernels durch Linus Torvalds ab 1991
- Wird mit GNU-Software zu komplettem Betriebssystem (GNU/Linux)
- Lizenz: GPL und ähnlich (Open Source, Recht auf Weiterentwicklung)
- Stellt UNIX-System auf *Consumer*-Hardware (x86-PC) zur Verfügung
- Mittlerweile Unterstützung weiterer Prozessoren (PowerPC, ARM, Motorola, . . .)

Distributionen bieten einen einfachen Weg, Linux auf einem PC zu installieren. Eine Distribution enthält:

- kompilierter Kernel mit Modulen für übliche Hardware
- diverse Dienstprogramme
- häufig benutze Anwendungen (Office, Webbrowser, Mail-Client, . . .)
- Desktop (KDE und/oder Gnome)
- Verwaltungsprogramme (Netzwerkkonfiguration, Konf. von Diensten, Benutzerverwaltung etc.)
- Konzept für (De-)Installation und Update von Anwendungen

Neben „Exoten“ (Slackware, Gentoo) existieren zwei Familien von Linux-Distributionen

- Red Hat (inkl. Fedora, CentOS, Scientific Linux) verwendet als Paketformat RPM,
- Debian (inkl. (K)Ubuntu) verwendet für das Paket-Management APT/dpkg

OpenSuse basiert zwar nicht auf Red Hat, benutzt aber auch RPM.

2 System-Komponenten

2.1 Kernel

Der Linux-Kernel ist *monolithisch*, aber *modular*: Betriebssystem-Kern läuft in einem Prozess mit globalem Adressraum, aber es kann Kernel-Code bei Bedarf hinzugeladen oder entladen werden (Kernel-Module). Aktuell ist die Kernel-Version 2.6. Hautpsächlich ist der Kernel in C geschrieben, mit Teilen in Assembler.

Komponenten des Linux-Kernels:

- Prozess-Management,
- Speicher-Management,
- VFS (Virtuelles Dateisystem),

- Netzwerk-Layer,
- Gerätetreiber,
- Architektur-spezifischer Teil

Kernel-Module enthalten meist Treiber, z. B. für Dateisysteme oder Geräte. Jedes Kernel-Modul enthält mindestens je eine Funktion, die beim Laden bzw. Entladen des Moduls ausgeführt wird. Diese müssen entweder `init_module` und `cleanup_module` heißen oder über die Makros `module_init` bzw. `module_exit` bekannt gemacht werden. Zusätzlich sollte das Modul enthalten, wenn es unter der Lizenz GPL steht, da sonst der Kernel nach dem Laden als *tainted* markiert wird.

Befehle für Kernel-Module:

- `insmod`
- `rmmod`
- `lsmod`
- `modprobe` lädt ein Modul mit Auflösen von Abhängigkeiten.

Geräte-Treiber greifen i. A. auf ein Device zu, das durch ein Device-File unter `/dev` repräsentiert wird. Die Identifikation des Codes, der den Zugriff auf das Device realisiert, erfolgt über die *Major Number* des Devices.

2.2 Boot-Prozess

Als Boot-Loader wird heute meist GRUB (früher LILO) verwendet. Im Master Boot Record (MBR) steht dabei der Code, wo auf die Partition verwiesen wird, die später als `/boot` gemountet wird und in der sich die Konfiguration des Bootmenüs (mit einer oder mehrerer Boot-Option(en), z. B. verschiedene Kernel, alternative Betriebssysteme wie Windows) befindet. Bei Linux wird dann das komprimierte Kernel-Image entpackt und eine initiale RAM-Disk geladen (früher `initrd`, heute `initramfs`), die z. B. Kernelmodule enthält, mit denen vom Kernel aus überhaupt erst auf die Festplatte zugegriffen werden kann.

Anschließend wird das Init-Skript ausgeführt, das u. a. das `/proc`-Dateisystem mountet und anschließend in den in `/etc/inittab` spezifizierten Runlevel wechselt.

2.3 Festplatten

Festplatten als Block-Geräte werden als Device-Dateien unter `/dev` repräsentiert, wobei IDE-Festplatten als `hda`, `hdb`, ... und SCSI-Festplatten als `sda`, `sdb` etc. benannt werden. Innerhalb der Platten werden die Partionen nummerisch hochgezählt. Neben Partitionen mit Dateisystemen wie `ext2`, `ext3`, `vfat`, die in Verzeichniseinträge als Mountpoints gemountet werden (entweder manuell mit dem Befehl

`mount -t` *`typ dev dir`*

oder anhand der Konfiguration in `/etc/fstab`), existiert als besondere Partion(en) eine (oder mehrere) Swap-Partition(en), die als Auslagerung des Arbeitsspeichers benutzt werden.

Befehle zum Partionieren von Festplatten: `fdisk` und `cfdisk`, Formatieren von Festplatten mit einem Dateisystem: `mkfs -t TYP DEV` (wobei implizit das Executable `mkfs.TYP`, z. B. `mkfs.ext3` aufgerufen wird).

Dateien besitzen dabei *drei* Zeitstempel:

- Letzter Zugriff (auch rein lesend): `atime`
- Letztes Schreiben (Modifikation): `mtime`
- Letzte Inode-Änderung (z. B. Änderung Zugriffsrechte): `ctime`

2.4 Geräte

Device-Dateien unterhalb von `/dev` werden mit dem Befehl

`mknod TYP MAJOR MINOR DEVNAME`

angelegt, wobei der Typ des Devices angibt, ob es sich um ein Block-Device (`b`) oder ein Character-Device (`c`) handelt. Die Major Number bestimmt, welche Realisierung der File-Operationen benutzt wird. Die Minor Number wird der Funktion übergeben, die für das Öffnen des Devices zuständig ist und kann von dieser ausgewertet werden.

2.5 Dienste

Für den Start von Diensten (dies können z. B. elementare Dinge wie Netzwerkfunktionalität allgemein oder das Loggen von Systemmeldungen sein, aber auch „hochwertige“ Dienste wie ein Webserver oder ein Datenbankserver) werden Init-Skripte verwendet. Diese verstehen mindestens die Kommandozeilenparameter `start`, `stop` und `status`, häufig noch die Parameter `restart` (stoppen und dann neu starten), `condrestart` (Neustart nur, wenn der Dienst schon läuft) und `reload` (Neuladen der Konfigurationsdateien).

Die Init-Skripte befinden sich in dem Verzeichnis `/etc/rc.d/init.d` und können entweder direkt mit Pfadangabe oder über das Programm `service` gestartet werden, z. B.

`service httpd start`.

Beim Booten wechselt der Init-Prozess zum Abschluss in einen Runlevel, der Dienste definiert, die in diesem laufen (oder nicht laufen) sollen. Dies geschieht über symbolische Links auf Init-Skripte der Form `S??xxx` oder `K??xxx`, wobei `S` fürs Starten und `K` fürs Beenden und `??` für eine zweistellige Zahl steht, die die Reihenfolge bestimt. Hinweis: Beim `K` werden nur die Dienste beendet, die einen Eintrag unter `/var/subsys` gemacht haben.

Beim Aufruf eines Init-Skriptes mit `start` als Parameter passiert daher in der Regel folgendes:

1. Einlesen von Konfigurationsdaten für den Dienst (meist aus einer Datei in dem Verzeichnis `/etc/sysconfig`),
2. Starten eines Dämons,
3. bei Erfolg schreiben der PID-Datei unter `/var/run` mit der PID des Prozesses und
4. anlegen einer (leeren) Datei unter `/var/lock/subsys`.

Beim Stoppen wird der zu beendete Prozess dann anhand der PID-Datei ausgewählt.

Bedeutung der Runlevel (diese sind nicht einheitlich je Distribution, z. B. weicht Debian leicht von u. a. Liste ab):

- 1, 2: Single-User ohne/mit Netzwerk
- 3, 5: Multi-User ohne/mit X
- 0: Halt
- 6: Reboot

2.6 proc-Dateisystem

Das Verzeichnis `/proc` enthält ein virtuelles Dateisystem mit Systeminformationen, die wie Dateien gelesen werden können:

- `version`
- `cpuinfo`
- `meminfo`
- `devices`
- `mounts`
- `modules`

Weiterhin existiert für jeden Prozess ein Unterverzeichnis mit der PID als Name. Hierin befinden sich alle wichtigen Daten zu dem Prozess:

- `cmdline`
- `environ`
- `fd` als Unterverzeichnis mit Filedeskriptoren

2.7 Entwicklungsumgebung

- Shell (i. A. `bash`),
- Compiler (`gcc`, `g++`) und Linker (`ld`),
- Make zum automatischen Kompilieren anhand von Abhängigkeiten,
- Entwicklungsumgebungen (Eclipse mit CDT, Kylix, Kile)

3 Bash

Standard-Shell unter Linux ist die Bash, die Vorteile wie *Command Completion* (mit TAB-Taste) und eine History der ausgeführten Kommandos bietet. Neben Bash-internen Befehlen können Programme ausgeführt werden, die entweder mit vollständiger Pfadangabe aufgerufen werden oder innerhalb der in `$PATH` spezifizierten Verzeichnisse gesucht werden.

Befehle in der Bash können kombiniert werden:

cmd1; cmd2 cmd1 und cmd2 werden hintereinander ausgeführt,

cmd1 && cmd2 cmd2 wird dann ausgeführt, wenn cmd1 erfolgreich war,

cmd1 || **cmd2** cmd2 wird dann ausgeführt, wenn *cmd1* nicht erfolgreich war.

Bei den letzten beiden Varianten ist `cmd1` meist ein Kommando, was auf eine Bedingung testet:

- `[ -f DATEINAME ]` prüft, ob die angegebene Datei existiert,
- `[ -x DATEINAME ]` ob die Datei ausführbar ist;
- `[ -d DIR ]` prüft, ob das Verzeichnis existiert;
- `[ -z STRING ]` ist wahr bei leerem String,
- `[ -n STRING ]` ist wahr, wenn der String nicht leer ist.

Über so genannte *Pipes* können Ein- bzw. Ausgabe umgeleitet werden:

- `cmd < datei` holt die Standardeingabe (`stdin`, FD 0) aus der angegebenen Datei,
- `cmd > datei` leitet die Standardausgabe (`stdout`, FD 1) in die Datei um,
- `cmd 2> datei` leitet den Fehlerkanal (`stderr`, FD 2) um,
- `cmd1 | cmd2` macht die Ausgabe von `cmd1` zur Eingabe von `cmd2`.

Die Bash unterstützt bereits eine richtige Programmierung, d. h.

- eine Reihe von Bash-Befehlen kann in einer Datei gespeichert werden,
- Variablen können definiert werden,

- wobei der Name des Skripts in $0 und die Kommandozeilenparameter in $1, $2, ...
- die üblichen Kontrollstrukturen und Schleifen existieren (IF, CASE, WHILE).

Beispiel eines Shell-Skriptes, das alle ausführbaren Dateien in einem Verzeichnis der Reihe nach ausführt:

Listing 1: Beispiel Bash-Skript

```
#! /bin/bash

usage() {
    echo "Usage: $(basename $0) [--log logfile] [--stop-on-error] dirname"
}

log=
stop_error=no
while [ "$1" != "${1#-}" ]
do
    option=$1
    case $option in
        -)
            shift
            break
            ;;
        --log=*)
            log=${option#--log=}
            shift
            ;;
        --log)
            log=$2
            shift 2
            ;;
        --stop-on-error)
            stop_error=yes
            shift
            ;;
        *)
            usage
            echo "Unknown option $option"
            exit 1
            ;;
    esac
done

dir=${1%/}
```

```
if [ -z "$dir" ]
then
    usage
    exit 1
fi

if [ ! -d "$dir" ]
then
    echo "Directory $dir does not exist"
    exit 2
fi

echo "Executing all in $dir"

for f in $dir/*
do
    [ -d "$f" ] && continue
    [ -x "$f" ] || continue
    echo "Executing $f"
    if [ -z "$log" ]
    then
        $f
    else
        $f >$log
    fi
    ret=$?
    if [ $ret -ne 0 ]
    then
        echo "$f failed with return status $ret"
        [ "$stop_error" = "yes" ] && break
    fi
done
```

Eine besondere Bedeutung hat ein Kommentar der Form `#!` am Anfang einer Datei. Dieser bestimmt, mit welchem Interpreter ein Skript ausgeführt wird, so z. B.

```
#! /bin/bash
```

mit der Bash und

```
#! /usr/bin/perl
```

als Perl-Skript mit dem Perl-Interpreter.

4 Nutzerkonzept

Linux unterstützt die Definition verschiedener Nutzer und Gruppen sowie die Definition von Zugriffsrechten auf Dateien und Verzeichnisse in Abhängigkeit von Nutzer und Gruppenzugehörigkeit.

4.1 Nutzer

Ein User unter Linux hat einen Usernamen, optional einen Realnamen und gehört mindestens einer Gruppe an, der primären Gruppe.

Diese Daten werden in der Datei `/etc/passwd` definiert, worin die nummerische User-ID (UID), die Gruppen-ID (GID) der primären Gruppe stehen sowie das Homeverzeichnis und die Login-Shell des Users.

Den Nummern der Gruppen (GID) ist in `/etc/groups` jeweils der Name der Gruppe zugeordnet. Weiterhin stehen dort bei jeder Gruppe die Nutzer, die dieser als sekundärer, als zusätzlicher Gruppe angehören.

Befehle zum Hinzufügen, Entfernen oder Ändern eines Nutzers: `useradd`, `userdel`, `usermod` sowie grafische Nutzerverwaltung der Distribution.

4.2 Zugriffsrechte

Jede Datei und jedes Verzeichnis unter Linux ist einem User und einer Gruppe zugeordnet. Die Zugriffsrechte einer Datei bzw. Verzeichnisses werden dabei individuell eingestellt für

- den Besitzer der Datei/des Verzeichnisses
- für Nutzer, die der Gruppe angehören, der die Datei/das Verzeichnis gehört,
- für alle anderen.

Hierbei wird unterschieden nach Lesen, Schreiben und Ausführen.

Ändern der Zugriffsrechte:

`chmod [ugoa][+-][rwx]`

mit folgenden Bedeutungen:

- u, g, o, a: User, Group, Other, All
- +, −: Recht hinzufügen bzw. entfernen
- r, w, x: Read, Write, eXecute

Bei Verzeichnissen hat das x-Bit die Bedeutung, dass in das Verzeichnis gewechselt werden darf.

Bei ausführbaren Dateien gibt es noch die Möglichkeit, dass das Programm mit der *effektiven* User-ID (EUID) des Dateibesitzers bzw. der *effektiven* Group-ID der Gruppe, der die Datei gehört, (EGID) ausgeführt wird: `chmod [ug][+-]s`.

Mit `chmod [+-]t DIR` wird für ein Verzeichnis das Sticky-Bit gesetzt bzw. entfernt. Ist für ein Verzeichnis dieses Bit gesetzt, so darf nur der jeweilige Besitzer oder `root` darin Dateien löschen oder umbenennen.

5 Programmierung

5.1 Filedeskriptoren

Ein- und Ausgabe wird in C über nummerische Filedeskriptoren geregelt. Ein solcher Filedeskriptor kann einen der Standard-I/O-Kanäle `stdin`, `stdout` und `stderr` kennzeichnen, eine Datei, die über `open` geöffnet wurde, eine Pipe oder einen Socket.

Elementares Lesen und Schreiben funktioniert mit den Funktionen `read` und `write`, bei denen jeweils neben der Nummer des Filedeskriptors ein Char-Zeiger mit der Adresse übergeben wird, wo die zu schreibenden Daten anfangen bzw. wo im Speicher die zu lesenden Daten landen sollen, sowohl die Anzahl von Bytes die geschrieben bzw. (maximal) gelesen werden soll.

Mit `fdopen` kann das Lesen und Schreiben aus einem bzw. in einen Filedskriptor gepuffert werden, das dann z. B. mit `fgets` oder `fprintf` erfolgt. Für das Lesen und Schreiben aus bzw. in Dateien ist es aber bequemer, direkt die Funktion `fopen` zu verwenden.

5.2 Prozesse

Forken eines Prozesses mit `fork()`: Es wird eine Kopie des Prozesses angelegt. Filedeskriptoren und Daten werden übernommen. Eltern- und Kindprozess arbeiten ab dem Fork aber mit ihren eigenen Daten.

Bedeutung des Rückgabewertes von `fork()`:

- 0: Im Kindprozess
- > 0: Im Elternprozess, Wert entspricht PID des Kindprozesses
- < 0: Fehler

Hinweis: Virtuelles Memory unter Linux, Prinzip von *Copy-On-Write*, physikalische Memory Pages werden erst bei Änderung tatsächlich kopiert.

Beim Ende eines Kindprozesses ist der Elternprozess dafür verantwortlich, dass dieser aus der Prozesstabelle ausgetragen wird, sonst bleibt ein *Zombie* übrig. Hierfür existieren die Funktion `wait` und `waitpid`.

Mit den Funktionen `execve`, `execl`, `execv`, ... wird ein Programm gestartet, das den aktuellen Prozess ersetzt.

5.3 Pipes

Eine Pipe öffnet einen Datenkanal, wobei man je einen Filedeskriptor für lesendes und schreibendes Ende der Pipe enthält. Ermöglicht Interprozess-Kommunikation (IPC) zwischen Eltern- und Kindprozess nach dem Fork.

Das folgende Listing zeigt eine Pipe, die nach dem Fork dazu benutzt werden kann, vom Eltern- an den Kindprozess Daten zu schreiben (erkennbar daran, dass im Elternprozess die lesende, im Kindprozess die schreibende Seite der Pipe geschlossen wird).

Listing 2: Kombination von fork und pipe

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main() {
    int pipe_fd[2];
    if (pipe(pipe_fd) == -1) {
        printf("Fehler %d bei Pipe: %s\n", errno, strerror(errno));
        exit(EXIT_FAILURE);
    }

    int child = fork();
    if (child < 0) {
        printf("Fehler %d bei Fork: %s\n", errno, strerror(errno));
        exit(EXIT_FAILURE);
    }

    if (child>0) { // Elternprozess
        // Lesende Seite der Pipe schließen
        close(pipe_fd[0]);
        ...
        wait();
```

```
        return(0);
    } else { // Kindprozess
        // Schreibende Seite der Pipe schließen
        close(pipe_fd[1]);
        ...
        return(0);
    }
}
```

Mit den Befehlen `dup` bzw. `dup2` kann in einem Programm auch `stdin` oder `stdout` umgeleitet werden.

Die Funktion `popen` kombiniert `execve` mit Pipes, indem als Kindprozess ein anderes Programm gestartet wird, dessen Standard-Eingabe *oder* -Ausgabe über eine Pipe zugänglich ist.

5.4 Threads

Mit der Funktion `pthread` wird ein neuer Ausführungsstrang innerhalb des Prozesses angelegt, der eine beim Aufruf angegebene Funktion ggfs. mit Parametern ausführt. Hierbei laufen die Threads quasi gleichzeitig und teilen sich die globalen Variablen des Prozesses.

Bei Verwendung von Threads muss die pthread-Bibliothek hinzu gelinkt werden:

```
gcc -o progname obj1.o obj2.o -lpthread
```

Da nicht deterministisch ist, wann das System zwischen Threads hin- und herschaltet, können sich Threads manchmal gegenseitig negativ beeinflussen. Es kann daher notwendig sein, dass *kritische Bereiche* mit Mutexes oder Semaphores geschützt werden.

Ein anderer Problemfall mit Threads kann auftreten, wenn Threads sich gegenseitig blockieren, bezeichnet als Verklemmung oder Deadlock. Gedankenbeispiel für Deadlocks ist das Philosophenproblem.

Threads können auch über so genannte *Conditions* synchronisiert werden, wobei ein Thread sich solange wartend schlafen legt, bis er von einem anderen Thread über die Condition aufgeweckt wird. Musterfall für Conditions sind Erzeuger-Verbraucher-Szenarien.

5.5 Signale

Bei bestimmten Situationen wird einem Prozess ein Signal geschickt, z. B. bei Eingabe von Ctrl-C, Aufruf von `kill` oder dem Ende eines Kindprozesses. Eine Liste von wichtigen Signalen findet sich in Tab. 1.

Per Default wird ein Prozess beim Erhalten eines Signales beendet. Stattdessen kann aber auch definiert werden (mit Ausnahme von `SIGKILL` zum Töten oder `SIGSTOP` zum

Tabelle 1: Liste der Signale

Signal	Nr.	Bedeutung
SIGHUP	1	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Interrupt from keyboard
SIGQUIT	3	Quit from keyboard
SIGILL	4	Illegal Instruction
SIGABRT	6	Abort signal from abort(3)
SIGFPE	8	Floating point exception
SIGKILL	9	Kill signal
SIGSEGV	11	Invalid memory reference
SIGPIPE	13	Broken pipe: write to pipe with no readers
SIGALRM	14	Timer signal from alarm(2)
SIGTERM	15	Termination signal
SIGUSR1	10	User-defined signal 1
SIGUSR2	12	User-defined signal 2
SIGCHLD	17	Child stopped or terminated
SIGSTOP	19	Stop process
SIGTSTP	20	Stop typed at tty

Schlafenlegen eines Prozesses), dass ein Prozess bestimmte Signale einfach ignoriert oder einen selbst definierten *Signal Handler* ausführt. Hierfür existiert die Funktion `sigaction`.

Zum Senden von Signalen an einen Prozess existiert der Befehl `kill` sowohl als Programm als auch als C-Funktion (weitherhin gibt es noch die C-Funktionen `raise` und `abort`).

6 Kompilieren und Linken

Beim Kompilieren eines C-Quelltextes wird eine Object-Datei geschrieben. Diese enthält den Ausführungs-Code der in der Datei definierten Funktionen und Symbole auf Funktionen, die in dem Quelltext zwar bekannt waren (über eine Deklaration in einer Header-Datei) und verwendet wurden, aber hier nicht definiert wurden.

Aufgabe des Linkers ist es, diese Symbole aufzulösen, entweder, indem verschiedene Object-Dateien zusammen gelinkt werden oder die benutzten Symbole aus einer Bibliothekt entnommen werden:

```
gcc -o prog a.o b.o
```

oder

```
gcc -o prog a.o -lxyz
```

wobei im zweiten Fall Funktionen noch der Bibliothek `libxyz` entnommen werden.

Unter Linux werden zwei Formen von Bibliotheken unterschieden:

- Statische Bibliotheken, die aus Object-Dateien mit dem Programm `ar` erzeugt werden. Diese sind im Wesentlichen Archive von Object-Dateien, der ausführbare Code der hinzu gelinkten Symbole wird dann statisch in das ausführbare Programm hinzugefügt, nach dem Linken wird die Bibliothek also nicht mehr benötigt. Dateiendung: `.a`
- Dynamische Bibliotheken, die mit `gcc -shared` generiert werden. Hier landet im ausführbaren Programm nur eine Referenz auf die Bibliothek, der eigentliche Code einer Funktion wird dann erst zur Laufzeit aus der Bibliothek geholt. Daher ist das ausführbare Programm kleiner, aber die Bibliothekt wird auch noch zur Laufzeit benötigt. Dateiendung: `.so`

Dynamische Bibliotheken werden im so genannten Library Cache gesucht, in dem die Bibliotheken in den Verzeichnissen `/lib` und `/usr/lib` und in den Verzeichnissen, die in `/etc/ld.so.conf` (heute meist Einträge in `/etc/ld.so.conf.d`) indiziert werden. Dieser Cache wird mit dem Befehl `ldconfig` aktualisiert.

Zuätzliche Verzeichnisse werden durchsucht, wenn sie in der Umgebungsvariablen mit dem Namen `LD_LIBRARY_PATH` enthalten sind.

Der Linker erzeugt ein ausführbares Programm im Format ELF. Dieses enthält mehrere Sections:

- `.text` mit dem Maschinencode
- `.data` mit *initialisierten* globalen Variablen (schreibbar)
- `.rodata` mit *konstanten* globalen Variablen
- `.bss` mit nicht initialisierten globalen Variablen.

Beim Linken gegen dynamische Bibliotheken wird sich auf den SO-Namen der Bibliothek bezogen. So ist es möglich, verschiedenen, nicht kompatible Versionen einen Bibliothek auf einem System installiert zu haben.

Beispiel

Auf dem System sind die Header-Datei `abc.h` und `libabc.so` in der Version 1 installiert, wobei die Bibliothek die Versionsnummer 1.1 trägt. Dies wird durch folgende symbolische Links repräsentiert:

`libabc.so` → `libabc.so.1` → `libabc.so.1.1`

Wird nun ein Programm gegen diese Bibliothek gelinkt, so gegen den SO-Namen, also `libabc.so.1`.

Erfolgt nun eine *kompatible* Änderung (Header-Datei `abc.h` ändert sich nicht, also keine Änderung der API), so wird nur die letzte Ziffer hochgezählt und die Links entsprechend angepasst:

`libabc.so` → `libabc.so.1` → `libabc.so.1.2`

Vorher gelinkte Anwendungen (gelinkt gegen `libabc.so.1`) benutzen nun die kompatible Version `libabc.so.1.2`.

Ändert sich nun die API (erkennbar daran, das auch `abc.h` geändert wird), so wird auch der SO-Name geändert:

`libabc.so` → `libabc.so.2` → `libabc.so.2.1`

Die alte Version kann parallel installiert bleiben:

`libabc.so.1` → `libabc.so.1.2`

Neue Anwendungen werden also gegen die neue Version gelinkt, mit dem SO-Namen `libabc.so.2`, alte Programme, die noch gegen `libabc.so.1` gelinkt sind, funktionieren aber noch weiterhin, da auch die alte Version noch installiert ist.

7 Netzwerk

7.1 Konfiguration

Netzwerkadapter werden unter Linux mit `eth0`, `eth1` etc. durchnummeriert (natürlich muss innerhalb des Kernels, meist als Modul geladen, ein Treiber für die Netzwerk-Hardware existieren). Zusätzlich existiert ein Loopback-Device unter dem Namen `lo`, das den eigenen Rechner unter der IP-Adresse `127.0.0.1` erreichbar macht.

Die wichtigsten Daten zur Netzwerk-Konfiguration erhält man über `ifconfig`, nämlich zu den einzelnen Netzwerkadaptern

- MAC-Adresse der Karte
- zugeordnete IP-Adresse
- zugeordnete Netmask

Routen werden über das Programm `route` angezeigt und definiert.

Beispiel: Laut `ifconfig` habe `eth0` die IP 192.168.1.17 und es existiere folgende Route

```
Ziel            Router       Genmask          Iface
192.168.1.0     *            255.255.255.0    eth0
default         192.168.1.1  0.0.0.0          eth0
```

Verbindungen zu 127.0.0.1 gehen über `lo`, also per Loopback. Der Drucker mit der IP 192.168.1.5 befindet sich im gleichen Subnetz wie `eth0` (da ein Bit-And von eigener IP mit der Netmask und Bit-And der IP des Druckers mit der Netmask identisch sind). Also werden Pakete zum Drucker von `eth0` direkt geschickt. Die IP 209.185.149.99 liegt außerhalb des von der Netmask definierten Subnetzes, hier werden Pakete über das Gateway geschickt, das die IP 192.168.1.1 hat (z. B. DSL-Router).

Namensauflösung passiert über die in `/etc/resolv.conf` eingestellten DNS-Server.

Die meisten Linux-Distributionen erzeugen diese Datei und Routen-Einstellungen über grafische Verwaltungsprogramme für Netzwerkadapter, wobei im einfachsten Fall die Daten über DHCP von einem DHCP-Server geholt und dann die Dateien automatisch generiert werden.

7.2 Programmierung

Die verschiedenen Ebenen der Netzwerk-Kommunikation lassen sich vereinfacht so darstellen:

Netzwerkschicht Physikalische Übertragung der Pakete, z. B. Ethernet

Internet-Schicht Definiert das Format der Datenpakete über das Internet-Protokoll (IP)

Transport-Schicht Definiert, ob die Kommunikation verbindungsorientiert (TCP) oder verbindungslos (UDP) abläuft

Anwendungsschicht Realisiert das Protokoll für die Anwendung, wie HTTP, SMTP, NTP.

Grundlegender Befehl in der Socket-Programmierung ist die Funktion `socket`, die für

- eine Kommunikations-Familie (`AF_INET`, `AF_UNIX`),
- eine Kommunikations-Art (`SOCK_STREAM` für Stream-Verbindung, `SOCK_DGRAM` für verbindungslosen Austausch von Datentelegrammen)
- ein Protokoll (`PF_INET`, `PF_UNIX`, wird meist automatisch anhand der Familie bestimmt)

einen nummerischen Filedeskriptor generiert.

Der grundlegende Ablauf einer Client-Server-Verbindung über TCP/IP (dies geschieht z. B. beim Abruf einer WWW-Seite von einem Browser aus) ist in Abb. 1 dargestellt:

- Der Server bindet den Socket an einen oder mehrere Netzwerkadapter, auf denen er dann auf Verbindungen wartet. Beim Annehmen einer Verbindung mit `accept` wird dann ein Filedeskriptor für Client-Socket zurück gegeben, über den dann die Daten geschrieben und gelesen werden können.
- Der Client verbindet sich über den Socket unter Angabe der Adresse und des Ports des Zielrechners mit dem Server und kann dann darüber Daten lesen und schreiben.

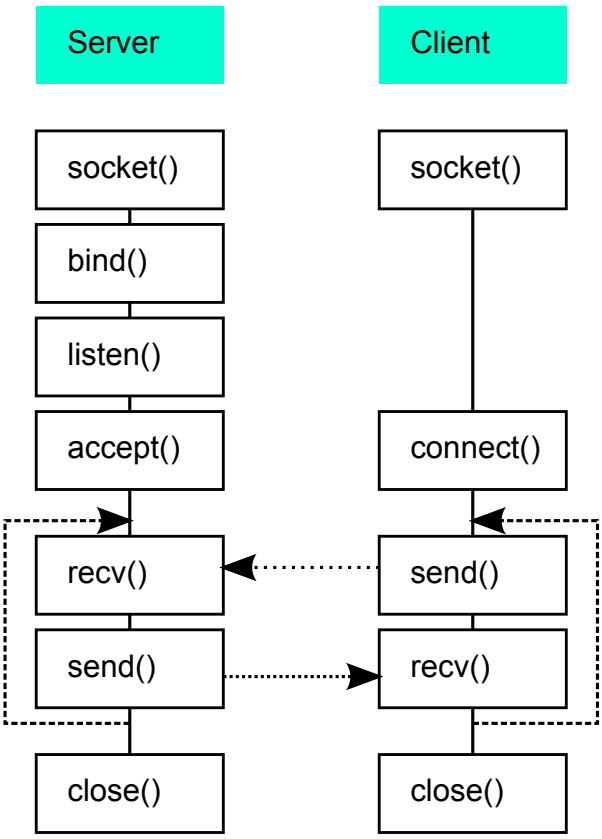

Abbildung 1: Ablauf einer Client-Server-Verbindung

Für Adressangaben wird eine generische Struktur vom Typ `sockaddr` erwartet. Aus Vereinfachungsgründen existieren für die einzelen Familien spezielle Strukturen, die hierzu binär kompatibel, aber bequemer zu füllen sind: `sockaddr_in`, `sockaddr_un`.

Zum Lesen und Schreiben können entweder die Funktionen `read` und `write` oder `recv` und `send` verwendet werden, wobei erstere die allgemeinen Funktionen zum Lesen/Schreiben mit Filedeskriptoren sind, letzter spezialisierte Varianten, bei denen noch zusätzliche Verbindungsoptionen angegeben werden können.

Bei verbindungsloser Kommunikation über UDP werden Daten-Telegramme von einem Rechner zu einem oder mehreren Rechnern geschickt. Hierbei erfolgt keine Kontrolle per Handshake, ob die Datagramme tatsächlich ankommen.

Da keine Verbindung erfolgt, muss der Sender bei jedem Datagramm die `sockaddr` des Empfängers oder der Empfänger angeben, wenn es per `sendto` verschickt wird. Der Empfänger liest die Pakte mit `recvfrom`.

Datagramme als Multicast gehen an beliebige Empfänger. Für Multicast gibt es einen speziellen IP-Bereich, 240.0.0.0 bis 239.255.255.255 (wobei 224.0.0.0 bis 224.0.0.255 reserviert ist). Von Seite des Senders besteht im Code kein Unterschied zwischen dem Senden eines UDP-Paketes an einen bestimmten Empfänger mit bekannter IP-Adresse oder senden an beliebige Empfänger unter Verwendung einer IP aus dem Multicast-Adressbereich

als Ziel-IP.

Lediglich der Empfänger muss hier modifiziert werden, dass er auf Multicasts hört:

Listing 3: Empfang von an 225.0.0.37, Port 12345 gerichtete Multicasts

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);

// Socket nicht blockieren, damit auch andere Empfänger lauschen können
int yes = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));

// Socket an alle Interfaces binden
sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(12345);
bind(fd, (struct sockaddr*) &addr, sizeof(addr));


// Multicast-Adresse hinzufügen
struct ip_mreq mreq;
mreq.imr_multiaddr.s_addr = inet_addr("225.0.0.37");
mreq.imr_interface.s_addr = htonl(INADDR_ANY);
setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

Zum Auflösen von Namen existiert die C-Funktion `gethostbyname`, die Anhand eines Namens aus dem DNS u. a. die IP-Adresse(n) liefert. In der Datei `/etc/services` sind bestimmten Protokollen (http, smtp, ftp, ntp) vordefinierte Protnummern zugeordnet. Einträge in diese Datei können von C aus über `getservbyname` abgefragt werden.

Statt Sockets auf IP-Basis sind auch *Unix Domain Sockets* (Familie `AF_UNIX`), bei denen die Kommunikation nicht über Netzwerk sondern über eine (virtuelle) Socket-Datei erfolgt. Dies stellt eine Alternative zur Interprozesskommunkation auf einem Rechner dar.