Kleinrechnerbetriebssysteme (Linux)

Vorlesung im Sommersemester 99
Dozent: Thomas Schöbel−Theurer
Uni Stuttgart

Ziele: − Betriebssystemkonzepte am Beispiel Linux
− Mitarbeit an Kernel−Projekten

Version 3.0
9. März 2000
Claus Möhrle

Inhalt

1

2 Einführung / Historie

2.1 Phase 1: Ur−Unix "Experimentierphase"

Multics (1965, MIT +General Electronics + Bell)

1969 Ausstieg von Bell ¢ Ken Thompson + Dennis Ritchie (waren bei Bell angestellt) ¢ quasi "arbeitslos"
¢ Spiel "Space Travel" für PDP−7 (Process Date P..?)
¢ Thompson entwickelt Filesystem (für das Spiel) (Assembler)
Name: "UNIX" als Persiflage auf Multics

1971 Auf PDP−11 portiert, 16k Kernel, 8k User, 512k Disc, 64k/file
Thompson entwickelt "B", angelehnt an BCPL (Martin Rigers)
war interpretativ ¢ Perfomanceproblem ¢ Entwicklung von "C"
¢ Versuch ein Betriebssystem portierbar zu machen!

1973 Neuimplementierung in C, ca. 25 Installationen

1974 Paper in CACM, freie Lizenzen für Universitäten, u.a. Berkley (Berkely Software Distribution)

2.2 Phase 2: Expansion

1977 Erste Portierung (Interdate 8/32)
Erste kommerzielle Anwendung

1978 Portierung auf VAX, 32 Bit!
(UNIX, 32V ¢ 3BSD 1979)

1979 Version 7, wurde stark von Anwendern verbessert ¢ **Kooperativer Entwicklungsstil**
BSD: wird von DARPA gefördert um TCP/IP zu implementieren

1980 4BSD, damit Trennung BSD ¢ Sys V
Beiträge von BSD: csh, virtual memory, FFS (Fast File System), reliable signals, sockets (Netzwerk)
bis 1993 Einschub bei BSD: Sponsor DARPA steigt aus, BSD I übernimmt kommerzielle Vermarktung, gerichtliche Auseinandersetzung mit AT & T ¢ FreeBSD, NetBSD

2.3 Phase 3: Kommerzialisierung

SunOS(1982, 4.2 BSD), später Solaris (basiert auf Sys V)
XENIX (Microsoft und SCO)
AIX (IBM), HP−UX, Ultrix (DEC) ¢ DEC (OSF/1)
Beiträge der "Kommerziellen":
NFS (Sun), Vnode / VFS (Sun), Journaling FS (IBM), Multiprozessor (Ultrix)

weitere Impulse: Mach (ca. 1986) ab Version 3.0 Microkernel,
¢ OSF/1 und Nextstep

Rangeleien:

1987 Allianz zwischen AT & T und Sun, später UI genannt

1988 Antwort: Restliche Hersteller schließen sich zur OSF zusammen. Eigenes Konzept, von denen heute nur noch Motif Bedeutung hat.

1990 (ab) streit plötzlich zu Ende, da Microsoft die Bühne betritt

1991 Auflösung von UI, OSF gibt die ambitionierten Pläne auf (z.B. Distributed Management Environment)
₩ SV R4 (System V Release 4)
integriert Features auch von BSD, SunOS und XENIX
¢ Unternummerierung SV R4.2

2.4 Phase 4: Standardisierung (ab ca. 1986)

´90 IEEE Standart 1003.x, auch POSIX genannt (**P**ortable **O**perating **S**ystem **I**nterface for Uni**x**)

1984 X/Open ist der Name einer internationalen Gruppe von Computerherstellern.
Ab ca. 1984 ₩ XPG, z.B. XPG 4 (**X**/Open **P**ortability **G**uide), siebenbändiges Handbuch zur Portierung von Programmen. XPEG definiert jedoch zusätzliche Merkmale, die nicht in POSIX.1 enthalten sind.
ê kein Standart, sondern de Facto Standart zu einer "Plattform" verknüpfen

1989 ANSIC (**A**merican **N**ational **S**tandards **I**nstitutes) , auch außerhalb der Unix–Welt!

1989 FIPS – **F**ederal **I**nformation **P**rocessing **S**tandard; Regulierung der USA. Verschärfung von POSIX – POSIX.FIPS schreibt einige Funktionen vor die in POSIX nur optional sind. Wer an die US–Regierung ein POSIX ähnliches System verkaufen will, muss POSIX.FIPS beachten.

1989 SVID (**S**ystem **V** **I**nterface **D**efinition) ab 1989 enthält BSD– Kompatibilitäts– Library
In der dritten Auflage von SVID wird die Funktionalität definiert, die von UNIX–Systemen, die mit dem System V Release 4 kompatibel sein wollen, erfüllt werden muss.

₩ Trend zur Integration

2.5 Phase 5: Freie Reimplementierung der Standards

GNU–Projekt (www.gnu.org), ab Anfang der 80´iger (Richard Stallman) GPL (General Public Licence)
Ziel: eigenes freies Betriebssystem unter GPL www.gnu.org/philisophy/why_free.html

₩ Lizenzbedingungen als "Motor" der Weiterentwicklung
GNU Hauptbeiträge: gcc, emacs, exteen nachhaltige Programmierungsumgebung, ¢ BS namens "Hurd[1]", Hurd stagniert immer noch, Zeichen des Verfalls. Linux Anleihen wie ext2.

Linux ab ca. 1991 komplette Neuimplementierung des Standards unter GPL, greift auf vorhandene Literatur zurück (Spezifikation) und implementiert sie "quick & dirty"

Entwicklungsstile:	Kirche	¢	Basar
	BSD		Linux

[1] Hird of Unix–replacing Daemons, Hird = Hurd of Interface representing depth

3 Die Posix–Programmierschnittstelle (Übersicht)

(Portable Operating System Interface for Unix)

a) **File I/O** (ungepuffert)
b) **Files and Directories**
c) Standard I/O Library (gepuffert)
d) System Data Files (z.B. /dev/null ¢ wenn hier hineingeschrieben wird, verschwindet das Geschriebene ¢ "Papierkorb"
/etc/passwd)
e) **Environment of an Unix Process**
f) Process Groups/Relationships
g) Signals
h) Terminal I/O
i) Daemon Process
j) **Advanced I/O**
k) Interprocess Communication
l) Advanced IPC
m) Pseudo Terminal
n) Process Control

3.1 File I/O (a)

Folgende Funktionen werden oft Funktionen zur ungepufferten Ein– /Ausgabe benannt, da jeder Aufruf von *read()* oder *write()* einen Systemaufruf in den Kern auslöst.

Grundthema: Pathnames ¢ Filedescriptor
statisch ¢ beständig — dynamisch ¢ Stromausfall ¢ weg!
Handl, dass zur Laufzeit erzeugt wird und eine Datei identifiziert (nicht negative Ganzzahl).

Zwei verschiedene Prozesse können durch ihre eigenen Filedescriptor die selbe Datei zur selben Zeit öffnen!

Notwendig aus Performancegründen.
Vordefinierte Filedescriptoren:

```
#define     STDIN_FILENO      0     -|
#define     STDOUT_FILENO     1      |- <unistd.h>
#define     STDERR_FILENO     2     -|

1) int      open  (const  char* pathname,
                          int   oflag,
                          /*mode_t mode*/);
```

Werte für oflag:

min. eins:
`O_RDONLY`
`O_WRONLY` (nur für das eine Filehandle, nicht für die Datei)
`O_RDWR`

beliebig:
`O_APPEND` positioniert ans Ende, arbeitet atomar!
(atomar für alle Schreiboperationen, jede OP hängt hinten an, kein Überschreiben!)
¢ Bsp.: Logfile mehrerer Daemons. Alle schreiben in die gleiche Datei. Die eine OP muss abgeschlossen sein, bevor die nächste startet.

Semaphor {

`O_CREAT` erstellt, falls nicht existiert
`O_EXCEL` liefert Fehler, wenn Datei bereits existiert ¢ atomar!
(nur wenn auch die Option O_CREATE angeben wurde)

O_TRUNC	löscht Dateiinhalt (Länge wird auf 0 gesetzt) ¢ atomar (falls die Datei existiert und erfolgreich zum Schreiben ohne Lesezugriff oder Lesen und Schreiben geöffnet wird)
O_NONEBLOCK	Spezialmodus (nicht blockierend)
O_SYNC	Synchrones I/O "Durchschreiben" ¢ Aufrufer wird blockiert bis Datei geschrieben.

₩ liefert den niedrigsten noch nicht verwendetet Filedescriptor zurück
êReturncode < 0 bedeutet Fehler!

```
2) int      creat       (const char*  pathname,
                         mode_t mode);
```
₩ `open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);`

Ein Nachteil dieser Funktion ist, dass Dateien nur zum Schreiben geöffnet werden.
Besser:
`open(pathname, O_RDWR|O_CREAT|O_TRUNC, mode);`
(Lesen und Schreiben | Datei erstellen, wenn nicht existiert | Länge auf 0 setzten)

```
3) int      close       (int filedes);
```

Beim schließen einer Datei werden auch alle Datensatzsperren aufgehoben, die ein Prozess in der Datei vorgenommen hat. Wenn ein Prozess beendet wird, schließt der Systemkern automatisch alle geöffneten Dateien.

```
4) off_t    lseek       (int        filedes,
                         off_t      offset,
                         int        whence);
```
Werte für whence: (whence: von wo?)

`SEEK_SET` ¢ der Wert des Parameters *offset* wird zum Dateianfang addiert.
`SEEK_CUR` ¢ der Wert des Parameters *offset* (positiv od. negativ) wird zur aktuellen Position addiert
`SEEK_END` ¢ der Wert des Parameters *offset* (positiv od. negativ) wird zur Dateigröße addiert

ê Man darf hinter EOF positionieren, anschließendes write() erzeugt ein "Loch" ¢ **sparce file**"
(nicht geschrieben Bytes in der Datei werden als 0 gelesen)

```
5) ssize_t  read  (int        filedes,
                   void*      buff,
                   size_t     nbytes);
```

Es wird am aktuellen Datei–Offset mit dem Lesen begonnen (siehe *lseek()*). Vor der Rückgabe des Ergebnisses, wird der Offset–Wert um die Anzahl der gelesenen Bytes erhöht.
Rückgabe: Anzahl gelesener Bytes, 0 = EOF, –1 Fehler

```
6) ssize_t  write (int        filedes,
                   void*      buff,
                   size_t  nbytes);
```

Der Rückgabewert hat in der Regel den selben Wert wie der Parameter *nbytes*. Ist dies nicht der Fall ist ein Fehler aufgetreten (z.B. Platte voll). Nach erfolgreichem Schreiben, wird der Datei–Offset um die Anzahl der geschriebenen Byte erhöht. Wenn die Option O_APPEND im Aufruf der Funktion *open()*verwendet wird, wird der aktuelle Datei–Offset vor jedem Schreiben auf das Dateiende gesetzt.

```
7) int      dup   (int        filedes);
   int   dup2  (int        filedes,
                      int        filedes2);
```
Ein bestehender Filedescriptor lässt sich mit diesen beiden Funktion duplizieren. *dup()* liefert den niedrigsten noch nicht verwendetet Filedescriptor zurück, bei *dup2()* können wir den Wert des neuen Filedescriptor mit *filedes2* festlegen; falls der durch filedes2 bezeichnete Descriptor bereits geöffnet ist, wird er zuerst geschlossen. Dabei zeigt der neue Filedescriptor auf den selben Eintrag in der Dateitabelle wie der mit dem Parameter *fildes* angebende Filedescriptor. Jeder Descriptor ist jedoch mit einer eigenen Gruppe von Filedescriptor–Flags ausgestattet.

```
8) int      fcntl (int        filedes,
                   int        cmd,
```

```
                  ...                  );
```
Ändert Eigenschaften des Filedescriptors (teilweise implementierungsspezifisch). Zum Beispiel das *Close–On–Exe*c Flag, das einen Descriptor beim Starten eines neuen Programms durch exec() schließt.

```
9) int      ioctl (int  filedes,
                  int  request,
                  ...                  );
```
Ändert Eigenschaften des Dateninhalts, vor allem bei Geräten, oft stark geräteabhängig.

10)Files and Directories (b)

```
1) int   stat   (           char*       pathname,
                  struct  stat*      buf);
```
Diese Funktion gibt eine Datenstruktur mit Informationen über die durch den Parameter *pathname* bezeichnete Datei aus.

```
int   fstat (           int         filedes,
                  struct     stat*      buf);
```
Diese Funktion ermittelt Informationen über die Datei, die unter dem Filedescriptor *filedes* bereits geöffnet ist. Das f bei *fstat* steht für den Filedescriptor!

```
int   lstat (const      char*       pathname,
                  struct  stat*      buf);
```
Diese Funktion ähnelt der Funktion *stat*; wenn es sich aber bei der bezeichneten Datei jedoch um einen symbolischen Verweis handelt, gibt sie Informationen über den Verweis und nicht über die Datei!
¢ Sym**l**ink

```
struct stat {
      mode_t       st_mode;     Dateityp und Modus (Zugriffsberechtigungen)
      ino_t        st_ino;      i–node number (Seriennummer)
      dev_t        st_dev;      device number (filesystem) [Gerätenummer (Dateisystem)]
      dev_t        st_rdev;     device number for special files
      nlink_t      st_nlink;    number of links (Anzahl der Verweise)
      uid_t        st_uid;      user ID of owner
      gid_t        st_gid;      group ID of owner
      off_t        st_size;     size in bytes for regular files (Größe in Bytes)
      time_t       st_atime;    time of last access (letzter Zugriff: read())
      time_t       st_mtime;    time of last modification (letzte Änderung: write())
      time_t       st_ctime;    time of last status change (Dateistatusänderung: chown(),
chmod())
      long         st_blksize;  best I/O block size (geeignetste I/O–Blockgröße)
      long         st_blocks;   number if 512–byte blocks allocated (Anzahl der
zugewiesenen 512                             Byte großen Blöcke)
};
```

Dateitypen (Dateitypmakros aus <*sys/stat.h*>):

Regular File `S_ISREG()`
: Die Interpretation des Inhalts einer regulären Datei werden der Anwendung überlassen. Dem Systemkern ist es also egal, ob die Dateien Text– oder Binärdaten enthalten.

Directory File `S_ISDIR()`
: Verzeichnisdatei. Eine Datei, die den Namen von anderen Dateien und Zeiger auf Informationen über diese Dateien enthält.

Character Device `S_ISCHR()`
: Zeichenorientierte Gerätedateien. Ein Dateityp, der für bestimmt Gerätetypen des Systems verwendet wird.

Block Device `S_ISBLK()`
: Blockorientierte Dateien. Ein Dateityp, der für gewöhnliche bei Festplatten– oder Diskettenlaufwerken verwendet wird.

FIFO (named pipe) `S_ISFIFO()`
: Ein Dateityp, der für die Kommunikation zwischen Prozessen verwendet wird.

Socket `S_ISSOCK()`
: Ein Dateityp, der für die Netzwerkkommunikation zwischen Prozessen verwendet wird.

Symlinks `S_ISLINK()`
: Symbolische Verweise. Ein Dateityp der auf eine andere Datei verweist.

Verwendung der Makros:

```
[?]
stat (/home/user/test, &buf)
if (S_ISREG(buf.st_mode)
      cout << "regular file!";
if (S_ISDIR(buf.st_mode)
      cout << "directory!";
[?]
```

2)
```
int   access      (const      char*       pathname,
                              int         mode);
```
mode–Konstanten:

- R_OK Leseberechtigung prüfen
- W_OK Schreibberechtigung prüfen
- X_OK Ausführungsberechtigung prüfen
- F_OK Vorhandensein prüfen

Diese Funktion überprüft die Zugriffsberechtigung anhand der tatsächlichen Benutzernummer und der tatsächlichen Gruppennummer. Testet ob Operation möglich (mit read userid). Diese Funktion ist anzuwenden, wenn man vor einem *open*()–aufruf wissen möchte, ob die richtigen Rechte gesetzt sind.

3)
```
mode_t      umask       (mode_t     cmask);
```
Setzt default Permissions eines Prozesses für das Erzeugen neuer Dateien und Verzeichnisse.
Bei den Funktionen *open()* und *creat()* können mit dem Parameter *mode* die Zugriffsberechtigungen gesetzt werden. Bits, die in der Bitschutzmaske gesetzt sind, werden mit dem Parameter *mode* für die Datei deaktiviert.

Beispiel:
[..]
```
umask (S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

creat ("test",    S_IRUSR | S_IWUSR | S_IRGRP |
                  S_IWGRP | S_IROTH | S_IWOTH);
[?]
```
Die Bits für Lesen (Gruppe, Andere) und Schreiben (Gruppe, Andere) werden in den Aufruf von *creat()* deaktiviert ¢ die Datei *test* hat nur die Rechte für Lesen (Benutzer) und Schreiben (Benutzer)

4)
```
int   chmod       (const      char*       pathname,
                              mode_t      mode);
int   fchmod      (           int         filedes,
                              mode_t      mode);
```
Diese Funktionen können die Zugriffsberechtigungen bei <u>bestehenden</u> Dateien ändern.
chmod bearbeitet die mit dem Parameter *pathname* bezeichnete Datei, während die Funktion *fchmod* eine bereits geöffnete Datei bearbeitet.

5)
```
int   chown       (const      char*       pathname,
                              uid_t       owner
                              gid_t       group);

int   fchmod      (           int         filedes,
                              uid_t       owner,
                              gid_t       group);

int   lchown      (const      char*       pathname,
                              uid_t       owner,
                              gid_t       group);
```

Diese Funktionen erlauben uns, die Benutzernummer und die Gruppennummer von Dateien zu verändern.

chown ¢ pathname
fchown ¢ Bereits geöffnete Datei.
lchown ¢ Der Eigentümer des Verweises wird verändert nicht die Datei auf die er verweist!

6)
```
int   truncate    (const      char*       pathname,
int   ftruncate   (           int         filedes,
```

```
                                        off_t       length);
                                        off_t       length);
```

Verkürzt Dateiinhalt. Hierbei werden Daten vom Ende der Datei abgeschnitten. Bestehende Dateien werden auf *length* Byte verkürzt. War die vorherige Größe der Datei größer als *length*, kann auf die Daten hinter *length* nicht mehr zugegriffen werden. Ist die Dateigröße kleiner als *length*, ist das Ergebnis Systemabhängig.
(Siehe *open* (?, O_RDWR | *O_TRUNC* |?) ¢ Sonderfall, der eine Datei auf 0 verkürzt!)

```
7) int   link        (const     char*       existingpath,
                      const     char*       newpath);
```

Erzeugt Hardlinks. Die Verweise zeigen direkt auf den i–node einer Datei.

```
   int   unlink      (const     char*       pathname);
```

ê funktioniert auch mit geöffneten Dateien!

Anwendung bei Temp–Dateien, die dann automatisch beim Beenden der Anwendung gelöscht werden und nicht als Leichen übrig bleiben (kill –9). Die Datei wird nach dem Aufruf von *unlink* nicht gelöscht, da sie immer noch von der Anwendung geöffnet ist. Nur wenn die Anwendung mit der Funktion *close* die Datei schließt oder terminiert (in diesem Fall schließt der Systemkern alle geöffneten Dateien) wird die Datei gelöscht. Falls *pathname* einen symbolischen Verweis bezeichnet, bezieht sich die Funktion *unlink* auf den Verweis und nicht auf die Datei.

```
   int   rename      (const     char*       oldname,
                      const     char*       newname);
```

ê atomar, während der Änderung wird die Datei gesperrt!

Es gibt drei Fälle zu Unterscheiden:

1. Falls *oldname* eine Datei bezeichnet, die kein Verzeichnis ist, wird die Datei umbenannt. In diesem Fall kann *newname*, falls dieser Parameter angegebenen wird, kein Verzeichnis bezeichnen. Falls der mit *newname* angegebene Name existiert (kein Verzeichnis darstellt), wird er entfernt und *oldname* wird in *newname* umbenannt. Wir müssen in dem Verzeichnis, das *oldname* enthält, und in dem Verzeichnis, das *newname* enthält, über ein Zugriffsberechtigung zum Schreiben verfügen da wir beide Verzeichnisse verändern.
2. Falls *oldname* ein Verzeichnis bezeichnet, wird das Verzeichnis umbenannt. Falls der mit *newname* angegebene Name existiert, muss er ein leeres Verzeichnis bezeichnen. Wenn *newname* existiert (und ein leeres Verzeichnis ist), wird es entfernt und *oldname* wird in *newname* umbenannt. Verzeichnisse können auch nur dann umbenennt werden, wenn *newname* nicht dieselbe Pfadangabe wie *oldname* enthält. Bsp_ */usr/foo* ¢ */usr/foo/testdir*.
3. Ein Sonderfall liegt vor, wenn *oldname* und *newname* dieselbe Datei bezeichnen. In diesem fall kehrt die Funktion fehlerfrei zurück und nimmt keine Änderungen vor.

```
8) int   symlink     (const     char* actualpath,
                      const     char* sympath);
   int   readlink    (const     char* pathname,
                                char* buff,
                                int   bufsize);
```

Im Unterschied zu den festen Verweisen, von denen in 7) die Rede war und direkt auf einen I–node einer Datei zeigen, stellen symbolische Verweise indirekte Zeiger auf Dateien dar. Mit der Einführung von symbolischen Verweisen wollte man Beschränkungen von festen Verweisen umgehen:

a) bei festen Verweisen ist es normalerweise erforderlich, dass der Verweis und die Datei sich im selben Dateisystem befinden.
b) Normalerweise kann nur der Supervisor feste Verweise auf Verzeichnisse definieren.

Symbolische Verweise werden normalerweise dazu verwendet, um eine Datei oder eine ganze Verzeichnisstruktur an eine andere Position im System zu verlagern.

symlink(): Der symbolischer Verweis wird auch dann erstellt, wenn die mit *actualpath* angegebenen Datei nicht existiert. Da die Funktion *open()* symbolische Verweise zurückverfolgt, brauchen wir ein Verfahren, mit dem wir den Verweis selbst öffnen und den im Verweis enthaltenen Namen lesen können. Die Funktion *readlink()* erfüllt diese Anforderungen. Wenn die Funktion fehlerfrei ausgeführt wird, gibt sie die Anzahl der gelesenen Bytes in *buf* aus. Die Daten des symbolischen Verweises, die in *buf* zurückgegeben werden, sind nicht nullterminiert.

Folgende Funktionen verfolgen einen symbolischen Verweis zurück, d.h. der übergebene Pfadname (an die Funktion) bezieht sich auf die Datei, auf die der Verweis zeigt:

access()
chdir()
chmod()
creat()
chown() (implementierungsabhängig)

```
9) int   mkdir       (const      char*       pathname,
                                 mode_t      name);
   int   rmdir       (const      char*       pathname);
```

mkdir erstellt ein neues, leeres Verzeichnis. Bei den im *mode* angegebene Zugriffsberechtigungen, sollte darauf geachtet werden, dass zumindest eines der Ausführungsbits aktiviert ist, um den Zugriff auf Dateinamen innerhalb des Verzeichnisses zu ermöglichen.
Wenn der Verzeichniszähler des Verzeichnisses beim Aufruf vom *rmdir* den Wert 0 annimmt und kein andere Prozess auf das Verzeichnis zugreift, wird der von diesem Verzeichnis besetzte Speicherplatz geleert und damit verfügbar. Falls min. ein andere Prozess auf das Verzeichnis zugreift und der Zähler auf 0 steht, dann werden die Einträge Punkt und Punkt−Punkt entfernt. Darüber hinaus können keine neuen Dateien oder Verzeichnisse in diesem Verzeichnis erstellt werden. Der Speicherplatz wird aber erst wieder frei, wenn der letzte Prozess das Verzeichnis schließt.

```
10)DIR *      opendir     (const      char* pathname);
      struct      dirent * readdir  (DIR * dp);
      void        rewinddir         (DIR * dp);
      int         closdir           (DIR * dp);
```

DIR ist eine interne Struktur, in der die vier Funktionen die Informationen über das Verzeichnis, das sie gerade lesen, speichern. Die Funktion *opendir()* gibt einen Zeiger auf eine Struktur vom Typ *DIR* zurück, der dann von anderen drei Funktionen eingelesen wird. *opendir()* initialisiert auch Werte so, dass mit dem ersten Aufruf von *readdir()* der erste Eintrag im Verzeichnis gelesen wird. Die Reihenfolge der Einträge innerhalb der Verzeichnisse ist implementierungsabhängig. Für gewöhnlich sind Einträge nicht alphabetisch sortiert.

Die Struktur *dirent* ist implementierungsabhängig, enthält aber mindestens diese zwei Einträge.

```
struct dirent{
            ino_t d_ino;       // I-Knotennummer
            char  d_name;      // nullterminierter Dateiname
      };
```

```
11)int   chdir       (const      char* pathname);
      int   fchdir      (           int   filedes);
```

Wir können über einen Aufruf der Funktionen das Arbeitsverzeichnis des aufrufenden Prozesses verändern. Wir können das neue Arbeitsverzeichnis entweder durch einen *pathname* oder durch einen Filedescriptor(*filedes*) bezeichnen. In dem Arbeitsverzeichnis eines Prozesses geht die Suche nach relativen Pfadnamen (Pfadnamen, die nicht mit einem "/" beginnen) aus.

```
12)void  sync        (void);
      int   fsync       (int  filedes);
```

Traditionelle Unix−Implementierungen verfügen über einen Cache−Puffer im Systemkern, der bei den meisten Festplatten−I/O−Operationen verwendet wird. Wenn wir mit der Funktion *write()* Daten schreiben, kopiert der Systemkern die Daten normalerweise in einen seiner Puffer und fügt sie in die I/O−Warteschlange ein. Man nennt diesen Vorgang *verzögertes Schreiben*.
Der Systemkern schreibt schließlich sämtliche in den Puffern enthaltenen Datenblöcke auf die Festplatte. Normalerweise geschieht dies, wenn der Puffer für andere Datenblöcke benötigt wird. Mit den Funktionen *sync()* und *fsync()* wird die Konsistenz zwischen dem tatsächlichen auf der Festplatte vorhanden Dateisystem und dem Inhalt des Cache−Puffers sichergestellt.

Die Funktion *sync* reiht sämtliche modifizierten Pufferdaten in die Warteschlange zum Schreiben ein und kehrt zurück. Sie wartet **nicht**, bis die tatsächlichen I/O−Operation ausgeführt wird.

Die Funktion *sync()* wird normalerweise alle 30 Sekunden von einem Systemdämon (*update*) aufgerufen. Damit wird sichergestellt, dass die Datenpuffer des Systemkerns regelmäßig gelöscht werden.

Die Funktion *fsync()* bearbeitet nur **eine** Datei (die durch den Filedescriptor *fildes* angegeben wird) und **wartet**, bis die tatsächliche I/O–Operation ausgeführt wurde, bevor sie zurückkehrt! Beachte das Flag *O_SYNC* beim öffnen (*open()*) einer Datei ¢ synchrones Schreiben, hier wartet der Prozess, bis die Datei wirklich geschrieben wurde. Anwendung der Funktion ist bei Datenbanken zu finden.

ê Reflektiert Tatsache, dass im Kernel Puffer vorhanden sind!

3.3 Environment of a Unix Process (e)

1) Hauptprogramm–Interface

```
int      main       (int       argc,
                     char*     argv[]);
```

Wenn ein C–Programm vom Systemkern gestartet wird, wird vor dem Aufruf der Funktion *main()* eine spezielle Startroutine aufgerufen. In der ausführbaren Datei ist diese Startroutine als Startadresse des Programms verzeichnet. Diese Startroutine liest Werte vom Systemkern (die Befehlszeilenargumente und die Umgebungsvariablen) und setzt die Programmvariablen so, dass die Funktion *main()*, wie oben gezeigt, aufgerufen wird.

2) Terminierung:

normale

- Rückkehr aus main() (mittels *return*())
- `exit()   void exit (int status)`
  status ist der Rückgabewert an den Systemkern, um eventuelle Fehler zu verarbeiten.

anormale

- `abort()  void abort(void)`
  Anormale Beendigung eines Prozesses, da hierdurch das Signal SIGABRT generiert wird.
- Signale (siehe kill)
  Der Prozess empfängt bestimmte Signale. Das Signal kann von dem Prozess selbst, von einem anderen Prozess oder vom Systemkern generiert werden. Signale werden z.B. vom Systemkern generiert wenn ein Prozess auf einen Speicherbereich bezug nimmt, der außerhalb seines Adressbereichs liegt, oder eine Division durch 0 durchführt.

Unabhängig davon, wie Prozesse beendet werden, führt der Systemkern schlussendlich denselben Code aus. Er schließt alle geöffneten Descriptoren des Prozesses, gibt den Speicher frei, usw.

3.4 Process Control (f)

Erstellung neuer Prozesse, die Ausführung von Programmen und die Beendigung von Prozessen. Es gibt einige besondere Prozesse:

Nummer 0: Scheduler
Nummer 1: init
Nummer 2: Pagedaemon

1) Prozessidentifikation (PID)
Jedem Prozess ist eine eindeutige (nichtnegative Ganzzahl) Prozessnummer zugeordnet.

`pid_t getpid      (void)`
Prozessnummer des aufrufenden Prozesses (Kindprozess).

`pid_t getppid     (void)`
Nummer des dem aufrufendem Prozess übergeordneten Prozesses (Elternprozess).

ê Beachte, dass bei den Funktionen kein Rückgabewert für den Fehlerfall definiert ist!

2) Kreieren neuer Prozesse
Es gibt nur eine Weise, auf die der Systemkern Prozesse erzeugen kann, nämlich indem ein bestehender Prozess die Funktion *fork* aufruft (dies gilt nicht für besondere Prozesse wie den Swapper, init und den Pagedaemon).

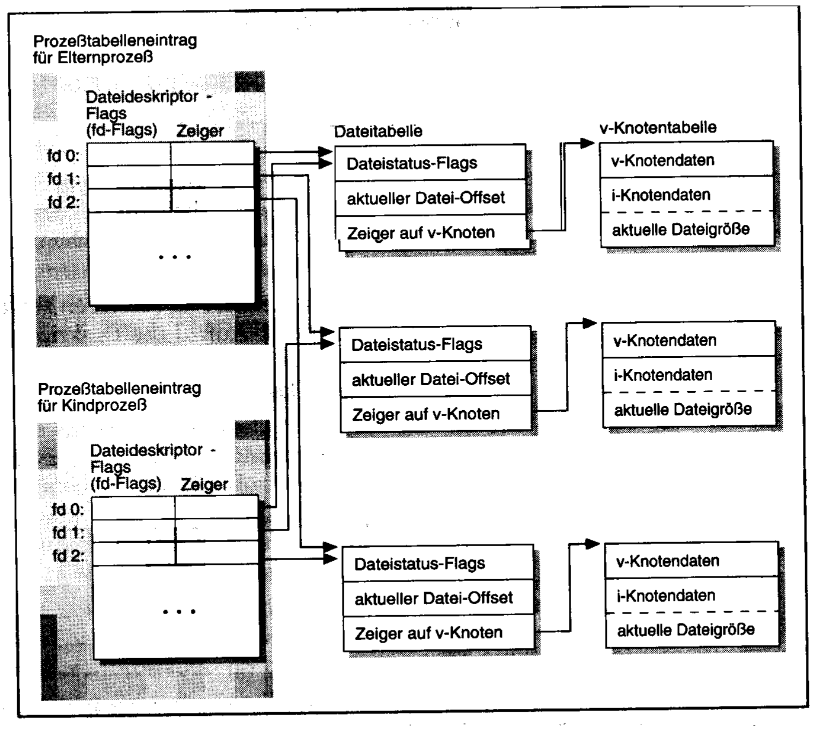
`pid_t fork        (void)`

Rückgabewerte: 0 an Kindprozess, Vaterprozess bekommt PID des Kindes zurück.

Der neue, von *fork()* kreierte Prozess, wird Kindprozess genannt. Diese Funktion wird nur einmal aufgerufen, liefert jedoch zwei Rückgabewerte. Die Rückgabewerte unterscheiden sich nur dadurch, dass dem Kindprozess immer der Wert 0, dem Elternprozess dagegen die Prozessnummer des neuen Kindprozesses übergeben wird. Die Prozessnummer des Kindprozesses wird von dem Elternprozess deshalb übergeben, da Prozesse über mehrere Kindprozesse verfügen können. Daher gibt es auch keine Funktion, mit der ein Prozess die Prozessnummer seiner Kindprozesse ermitteln kann. *fork()* übergibt dem Kindprozess den Wert 0, weil jeder Prozess nur ein Elternprozess haben kann. Der Kindprozess kann daher auch immer mit dem Aufruf der Funktion *getppid()* die Prozessnummer seines Elternprozesses ermitteln.
Der Kindprozess ist eine Kopie des Elternprozesses. Der Kindprozess erhält beispielsweise eine Kopie vom Datenbereich, Stack und Heap des Elternprozesses.

ê Es handelt sich hier um eine Kopie des Speicherbereichs ¢ er wird nicht gemeinsam benutzt!

Viele der heutigen Implementierungen kopieren die Daten, den Heap und den Stack des Elternprozesses nicht vollständig, da nach einem Aufruf von *fork()* oft ein *exec()*–Aufruf folgt. Statt dessen verwendet man copy–on–wirte (bei Schreibversuch kopieren). Diese Speicherbereiche werden vom Eltern– und Kindprozess gemeinsam benutzt und werden vom Systemkern schreibgeschützt. Wenn einer der beiden Prozesse versucht den Bereich zu verändern, erstellt der Systemkern nur eine Kopie des betreffenden Speicherbereichs.
Eine Eigenschaft von *fork()* ist auch, dass alle im Elternprozess geöffneten Descriptoren im Kindprozess dupliziert werden (gleiches Ergebnis wie wenn man die Funktion *dup*() für jeden Descriptor aufruft). Eltern– und Kindprozess greifen bei jedem geöffneten Descriptor gemeinsam auf den zugehörigen Dateitabelleneintrag zu.



Wenn sowohl Kind– als auch Elternprozess zum selben Dateidescriptor schreiben, und nicht in irgendeiner Weise synchronisiert werden, werden ihre Ausgaben vermischt!

Der Kindprozess erbt zusätzlich folgende Eigenschaften des Elternprozess:

- reale Benutzernummern, ...
- Prozessgruppennummern
- Sitzungsnummer
- aktuelles Arbeitsverzeichnis
- Stammverzeichnis
- Dateischutzbitmaske

- Umgebung
- ...

Unterschiedlich bei Eltern– und Kindprozess sind:
- Rückgabewert von fork
- Prozessnummern
- Dateisperren; vom Elternprozess verhängte Dateisperren werden nicht vererbt
- Timer; die laufenden Timer werden beim Kindprozess gelöscht
- ...

3)
```
pid_t    wait        (int* statloc);
pid_t    waitpid     (pid_ pid,
                      int* statloc,
                      int  options);
```
(wartet, bis irgendein Kindprozess terminiert)

Wenn ein Prozess normal oder anormal terminiert, wird der Elternprozess vom Systemkern durch das *SIGCHLD* –Signal darüber benachrichtet. Das Signal muss nicht beachtet werden!
Diese Funktionen können folgendes bewirken:
- der Prozess wird blockiert (falls alle Kindprozesse noch laufen) oder
- der Prozess kehrt sofort mit dem Beendigungsstatus des Kindprozesses zurück (falls der Kindprozess beendet wurde und darauf wartet, dass sein Beendigungsstatus abgerufen wird) oder
- der Prozess kehrt sofort zurück und gibt einen Fehler aus (falls es keine Kindprozesse hat).

Im einzelnen:
- *wait()* blockiert den aufrufenden Prozess, bis ein Kindprozess terminiert.
- *waitpid()* wartet nicht (¢ kein Blockieren!), bis der erste Kindprozess beendet wurde – es sind eine Reihe von Optionen verfügbar, mit denen sich steuern lässt, auf welchen Prozess gewartet werden soll. Gibt einen Fehler aus wenn der angegebene Prozess nicht existiert oder kein Kindprozess des aufrufenden Prozesses ist.

Wenn ein Kindprozess bereits beendet ist und den Status eines Zombies hat, kehrt *wait()* sofort zurück und gibt den Status des Kindprozesses aus. Ansonsten wird der aufrufende Prozess so lange blockiert, bis ein Kindprozess terminiert.

4)
```
int    excel    (const    char* pathname,
                          char* arg 0,
                          ...,
                          (char*)0[2]);
int    execle   (         ...
                          char* const envp[]);
int    execlp   (const    char*     filename,
                          ...   );
int    execv    (const    char* pathname,
                          char* const argv[]);
int    execve   (const    char* pathname
                          char* const argv[],
                          char* const envp[]);
int    execvp   (const    char* filename,
                          char* const argv[]);
```
(execlp) Unterschied: sucht in $Path

```
if (!fork()) {
   execl("/bin/ls","ls","-l",NULL);
};
```

Mit der Funktion *fork()* wird ein neuer Prozess kreiert, der dann über den einer *exec()*–Funktion ein anderes Programm startet. Wenn ein Prozess eine der *exec()*–Funktionen aufruft, wird dieser Prozess vollständig durch das neue Programm ersetzt, und das neue Programm beginnt die Ausführung mit seiner *main()*–Funktion. Die Prozessnummer wird hierbei nicht verändert, da kein neuer Prozess erzeugt wird. *exec()* ersetzt einfach den bestehenden Prozess (sein Textsegment, Datensegment, Heap und Stack) durch ein ganz neues von der Festplatte geladenes Programm.

Funktion	***pathname***	***filename***	***Parameterliste***	***argv[]***	***environ***	***envp[]***
execl	•		•		•	•

[2] Nullpointer

execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
Kennbuschstabe		***p***	***l***	***v***		***e***

p: Der Funktion muss ein Dateiname übergeben werden. Weiter sucht die Funktion in dem von der Umgebungsvariablen PATH angegebenen Suchpfad nach der Datei sucht.
l: Zeigt an, dass die Funktion eine Werteliste (*char* arg*) als Parameter erwartet.
v: Parameterangabe in Form eines Vektors (*argv[]*) wird erwartet.
e: Weist daraufhin, dass die Funktion nicht die aktuelle Umgebung berücksichtigt, sondern ein Array im Format *envo[]* liest, das den Suchpfad definiert.

Nach einem *exec*–Aufruf verändert sich die Prozessnummer nicht. Das neue Programm erbt jedoch noch weitere Eigenschaften vom aufrufenden Prozess:
- Prozessnummer des Elternprozesses
- aktuelles Arbeitsverzeichnis
- Stammverzeichnis
- Dateisperren (denn es wird kein neuer Prozess kreiert, sondern ein anderes Programm in diesen Prozess geladen und gestartet!)
- Dateischutzbitmaske
- ...

Beispiel:
```
[?]
pid = fork();
if (pid == 0) {
      execle      ("home/user/program", "program",
                "myarg1", "myarg2", (char *) 0, env_init);
[?]
if (pid == 0) {
   execlp      ("program", "program", "only_1_arg", (char *) 0,
                 env_init);
[?]
```

3.5 Advanced I/O (j)

1) File and Record Locking (Datensatzsperren)
?die Fähigkeit eines Prozesses, andere Prozesse davon abzuhalten, einen bestimmten Bereich einer Datei zu verändern, während der erste Prozess in diesem <u>Bereich</u> liest oder schreibt (¢ "Bereichssperren").

```
int   fcntl (int  filedes,
                   int  cmd,
                   /* struct flock xptr /*);
```

Diese Funktion kann jeden beliebigen Bereich einer Datei sperren, die ganze Datei ebenso gut wie ein einzelnes Byte!

cmd ist:
- **F_GETLK** bestimmt, ob die durch *flockptr* beschriebene Sperre durch eine andere Sperre verhindert wird. Falls eine Sperre existiert, die die Einrichtung unserer Sperre verhindern würde, dann wird die Struktur, auf die *flockptr* zeigt, mit den Informationen über die vorhandene Sperre überschrieben. Falls keine solche Sperre vorhanden ist, wird die Struktur, auf die *flockptr* zeigt, bis auf das Element *l_type* unverändert gelassen. Dieses Element wird auf F_UNLCK gesetzt.
- **F_SETLK** richtet die durch *flockptr* beschriebene Sperre ein. Wenn wir eine Sperre zum Lesen oder Schreiben (mit einem Wert von *F_RDLCK* bzw. *F_WRLCK* für *l_type*) anfordern und die Verträglichkeitsregel verhindert, dass das System die Sperre einrichtet, dann kehrt *fcntl* sofort zurück und setzt *errno* auf *EACCES* oder *EAGAIN*.
- **F_SETLKW** stellt eine blockierende Variante von *F_SETLK* dar (das W im Namen steht für "wait"). Falls die angeforderte Sperre zum Lesen oder Schreiben nicht eingerichtet werden kann,

weil ein andere Prozess einen Teil des gewünschten Bereichs gesperrt hat, wird der aufrufende Prozess angehalten. Er wird fortgesetzt, falls ein Signal auftritt, das er abfängt.

```
struct      flock {
      short l_type;     /* F_RDLCL, F_WRLCK, F_UNLCK */
      off_t l_start;    /* Offset in Byte relativ zu l_whence */
      short l_whence3;  /* SEEK_SET, SEEK_CUR, SEEK_END */
      off_t l_len;      /* Länge in Byte; 0 bedeutet Sperren bis EOF */
      pid_t l_pid;      /* nur bei F_GETLK */ Outputparam.
}
```

Die Struktur (*flock*)beschreibt:

- die Art der gewünschten Sperre: *F_RDLCK* (Sperre für gemeinsames Lesen), *F_WRLCK* (Sperre für exklusives Schreiben) und *F_UNLOCK* (hebt die Sperre für einen bestimmten bereich auf),
- das Offset des ersten Bytes des Bereichs, der gesperrt oder freigegeben wird (*l_start* und *l_whence*)
- die Größe des Bereichs (*l_len*).

Implizite Vererbung und Freigabe von Sperren:

a) Zu einer Sperre gehören ein Prozess und eine Datei (pid / filedes):
   - Wenn ein Prozess beendet wird, werden alle seine Sperren freigegeben.
   - Wenn ein Filedescriptor geschlossen wird, werden alle Sperren für diesen Prozess in der Datei, auf die der Filedescriptor verweist, freigegeben.
b) Sperren werden nie über einen *fork()*–Befehl an einen Kindprozess vererbt. Wenn also der Elternteil eine Sperre einrichtet und dann fork aufruft, wird der Kindprozess hinsichtlich der eingerichteten Sperre als fremder Prozess betrachtet. Der Kindprozess muss *fcntl()* aufrufen, um für alle mit *fork()* vererbten Descriptoren seine eignen Sperren einzurichten. Das ist auch Sinnvoll, denn schließlich sind Sperren dafür gedacht, zu verhindern, dass mehrere Prozesse gleichzeitig dieselbe Datei schreiben.
c) Über *exec()* können Sperren an ein neues Programm vererbt werden.

ê Locks gehören zu einem (pid / filedes)–Paar werden aber bei *dup()* und *fork()* nicht vererbt!
bei *exec()* werden sie vererbt!

ê Locking ist "advisory", d.h. hängt von Kooperationsbereitschaft ab!

2) Multiplexing von I/O
Wenn man aus zwei Filedescriptoren abwechselnd lesen möchte wird die Multiplex–I/O–Methode verwendet. In diesem Fall können wir für keinen dieser Descriptoren ein blockierendes *read()* verwenden, weil unter dem einem Descriptor Daten auftauchen können, während der andere durch ein *read()* blockiert ist. Dazu erstellen wir eine Liste mit den Filedescriptoren, die uns interessieren und rufen eine Funktion auf, die erst zurückkehrt, wenn einer der Filedescriptoren für I/O bereit ist. Die Funktion teilt bei der Rückkehr mit, welche Descriptoren bereit sind. Dadurch wird ein permanentes Abfragen (*read*) (auch Polling genannt) verhindert, bei dem die Descriptoren abwechselnd auf Daten überprüft werden.

```
int   select      (           int         maxfd,
                                    fd_set*     readfds,
                                    fd_set*     writefds,
                                    fd_set*     exeptfds,
                        struct      timeval*    tvptr);
```

Die Parameter der *select()*–Funktion teilen dem Systemkern folgendes mit:

- an welchen Descriptoren wir interessiert sind,
- welche Bedingungen uns bei jedem Descriptor interessiert (ob wir lesen oder schreiben oder von Fehlermeldungen erfahren wollen)
- wie lange wir warten wollen. (Wir können unbegrenzt lange, für eine bestimmte Zeit oder überhaupt nicht warten.)

Bei der Rückkehr von *select()* liefert der Systemkern folgende Informationen:

- Die Gesamtzahl der Descriptoren, die bereit sind.
- Welche Descriptoren für jede der drei Bedingungen bereit sind (Lesen, Schreiben oder Fehler).

Mit Hilfe dieser Infos können wir die geeigneten I/O–Funktionen aufrufen (für gewöhnlich *read()* oder *write()*) und wissen dabei, dass die Funktion nicht blockieren wird.

Wie Lange wollen wir warten?

```
struct      timeval {
```

[3] woher, von wo *[wens]*

```
      long  tv_seconds; //Sekunden
      long  tv_usec;    //und Mikrosekunden
};
```

`tvptr == NULL:`
unendliches warten. Die Rückkehr erfolgt, wenn einer der angegebenen Descriptoren bereit ist oder wenn ein Signal abgefangen wird.
`sonst:`
Timeout, evtl. auch gar nicht warten.

<u>Rückgabewerte:</u>
–1 : Es ist ein Fehler aufgetreten, oder ein Signal wurde abgefangen
0 : keine Descriptoren sind bereit (¢ Timeout wurde erreicht)
>0 : Anzahl der Descriptoren die bereit sind.

Die Funktion *poll()* ähnelt der Funktion *select()*, hat aber eine andere Programmierschnittstelle. *poll()* ist mit dem Datenstrom verknüpft, obwohl man die Funktion mit jedem Descriptor verwenden kann.

```
int   poll  (struct     pollfd          fdarray[],
                              unsignedlong      nfds,
                              int               timeout);
```

Anstatt für jede Bedingung (Lesen, Schreiben, Fehler) eine Descriptorengruppe zu initialisieren, wie wir das bei *select()* getan haben, initialisieren wir bei *poll()* ein Array mit *poolfd*–Strukturen, wobei jedes Element eine Descriptornummer und die Bedingung angibt, für die wir uns bei diesem Descriptor interessieren.

```
struct      pollfd {
      int   fd;          /* Descriptor prüfen, wenn < 0, ignorieren */
      short events;      /* Die interessierenden Ereignisse auf fd */
      short revents;     /* Die eingetretenen Ereignisse auf fd */
};
```

nfds:
Anzahl der Elemente in *fdarray.*
timeout:
= = INFTIM : unbegrenzt warten (<stropts.h> = –1)
= = 0 : überhaupt nicht warten
> 0 : *timeout* in Millisekunden warten

3) Scatter read, Gatherwrite

Die Funktionen *readv()* und *writev()* ermöglichen es, in einem einzigen Funktionsaufruf aus unzusammenhängenden Puffern zu lesen und in sie zu schreiben.

```
ssize_t     readv (           int   filedes,
                  struct      iorec iov[],
                              int   iovent);

ssize_t     writev(           int   filedes,
                  struct      iorec iov[],
                              int   iovent);
```

Die Anzahl der Elemente im Array *iov* wird durch *iovcnt* angegeben.
Rückgabewert beider Funktionen:
Anzahl der gelesenen oder geschriebenen Bytes, –1 Fehler.

```
struct      iorec {           // "Puffert" Zugriffe
      void*       iov_base;   // Startadresse des Puffers
      size_t      io_len;     // Puffergröße
};
```

writev() sammelt die Ausgabedaten in den Puffern der Reihenfolge iov[0], iov[1] bis iov[iovent–1]. Die Funktion liefert die Anzahl der geschriebenen Bytes – normalerweise sollte sie mit der Summe alle Pufferlängen übereinstimmen.

readv() verteilt in der selben Reihenfolge die Daten auf die einzelnen Puffer. Die Funktion füllt immer erst einen Puffer, bevor sie sich den nächsten vornimmt, *readv()* liefert die Gesamtzahl der gelesenen Bytes. Der Rückgabewert beträgt *Null*, wenn keine weiteren Daten mehr verfügbar sind und die Funktion auf das Dateiende stößt.

4) Memory Mapped I/O (I/O über ein Speicherabbild)

Bei der Ein– /Ausgabe über ein Speicherabbild (memory mapped I/O) wird eine Festplattendatei auf einen Puffer im Arbeitsspeicher abgebildet. Wenn wir Daten aus dem Puffer holen, wird der entsprechende Bereich der Datei gelesen, und wenn wir Daten in den Puffer schreiben, werden die entsprechenden Bytes automatisch in die Datei geschrieben. Wir können damit I/O–Operationen ausführen, <u>ohne</u> *read()* oder *write()* aufzurufen.
Wenn wir von dieser Möglichkeit Gebrauch machen wollen, müssen wir den Systemkern anweisen, eine bestimmte Datei auf einen Bereich im Arbeitsspeicher abzubilden. Dazu dient die Funktion *mmap*.

```
caddr_t     mmap  (caddr_t    addr,
                   size_t     len,
                   int        prot,
                   int        flag,
                   int        filedes,
                   off_t      offs);
```

Der Datentyp *caddr_t* wird meistens als *char* * definiert. Im Parameter *addr* können wir angeben, an welcher Adresse der abgebildete Bereich beginnen soll. Normalerweise sollte *addr* auf *Null* gesetzt werden, damit das System bei der Wahl der Startadresse freie Hand hat. Die Funktion gibt die Startadresse des abgebildeten Bereichs zurück.
filedes ist der Descriptor der Datei, die abgebildet werden soll. Die Datei muss zuvor geöffnet worden sein. *len* steht für die Anzahl der abzubildenden Bytes und *off* repräsentiert den Datei–Offset der abzubildenden Bytes.
Der Parameter *prot* spezifiziert den Zugriffsschutz auf den abgebildeten Bereich.

4 Architektur von Linux (Übersicht)

4.1 Grundprinzipien

- Monolithischer Kernel (kein Mikrokernel aber dynamisch nachladbare Module, z.B. für ein selten benötigtes Dateisystem)

  Der eigentliche Kern stellt bei einem Mikrokernel nur das absolut notwendige Minimum an Funktionalität (Interprozesskommunikation und Speicherverwaltung) zur Verfügung und kann deswegen klein und kompakt implementiert werden. Auf diesen Mikrokernel aufbauend, wird die restliche Funktionalität des Betriebssystems in eigenständige Prozesse ausgelagert, die mit dem Mikrokernel über eine wohldefiniertre Schnittstelle kommunizieren. Der große Vorteil dieser Architektur ist eine auf den ersten Blick wartungsfreundliche Struktur des Systems. Einzelne Komponenten arbeiten unabhängig voneinander, können sich nicht ungewollt beeinflussen und sind leichter austauschbar. Die Entwicklung neuer Komponenten wird vereinfacht. Daraus ergibt sich auch ein Nachteil eines Mikrokernels, denn er erzwingt die Einhaltung der definierten Schnittstellen zwischen den einzelnen Komponenten und erschwert damit trickreiche Optimierungen. Außerdem ist die im Mikrokernel benötigte Interprozesskommunikation auf heutigen Hardwarearchitekturen aufwendiger als einfache Funktionsaufrufe. Das System wird hierdurch etwas langsamer als traditionelle monolithische Kerne. (Hardware immer schneller, System besser Wartbar ¢ Zukunft!)
  Linux hingegen entstand auf der "langsamen" 386–Architektur. Gutes Laufzeitverhalten durch ausreizen aller Optimierungsmöglichkeiten stand bei der Entwicklung im Vordergrund. Das ist der Grund warum Linux in der klassischen monolithischen Kernelarchitektur realisiert wurde. Ein weiterer Grund ist sicherlich, dass eine Mikrokernel–Architektur ein sorgfältiges Systemdesign bedingt, da Linux evolutionär entstanden ist, war dies einfach nicht möglich.

- Systemschnittstellen zentral (in <linux/*.h>)
- Architekturabhängige Teile (in <asm/*.h>)
- Entwicklung stark evolutionär, Grundprinzipien von [Tanenbaum] und [Bach]

- ***Subsysteme***

- Hilfsroutinen allgemeiner Art (z.B. memset(), memcpy(), ...)
- Low–Level Synchronisationsmittel (z.B. Interrupt–Sperren, Spin locks, Semaphore)
- High–Level Scheduling (Internes Koroutinen–Modell ¢ Probleme mit SMP[4])
  ₩ Koroutinen: (nur eine kann laufen!)
  Transfer (self, other)
  start (other)
  ¢ Unterbrechungsstelle ist fest (bekannt). Anzahl der Transferbefehle im Code.
  Aus der Sicht eines unter Linux ablaufenden Prozeses stellt sich der Kern als Anbieter von Dienstleistungen dar. Einzelne Prozesse existieren unabhänhig nebeneinander und können sich nicht driekt beeinflusen. Der eigene Speicherbereich ist vorallem vor dem Zugriff fremder Prozesse geschützt. Die interne Sicht auf ein Linux–System ist etwas anders. Auf dem Rechner läuft nur ein Programm – das Betriebssystem –, welches auf alle Resourcen zugreifen kann. Die verschiedenen Tasks werden durch Koroutinen realisiert, d.h. jede Task entscheidet selbst, ob und wann sie die Steuerung an eine andere Task abgibt (dies wird auch als kooperatives Multitasking bezeichnet). Eine Konsequenz daraus ist, dass ein Fehler in der Kernprogrammierung das ganze System blockieren kann.
- Low–Level Speicherverwaltung (Kernelspeicher, Pages)
- Architekturabhängige Routinen für virtuelle Adressräume
- Architekturunabhängige Routinen für virtuelle Adressräume
- High–Level–Routinen für Paging und Swapping
- Gerätetreiber–Grundfunktionen (z.B. IO–requests)
- Gerätetreiber, zum Teil mit interner Hierarchie, z.B. SCSI
- VFS–Layer
- einzelne Dateisysteme, z.B. minix, ext2
- Netzwerk, inzwischen stark modularisiert

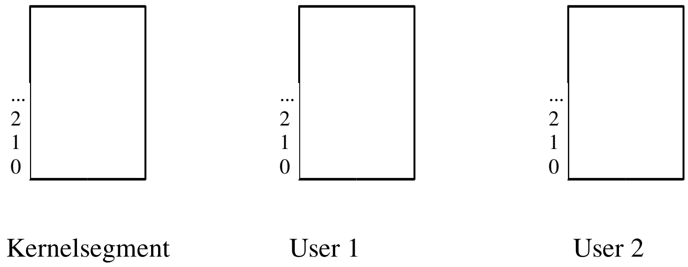
---
[4] Symmetric Multi Processing

ê Im Prinzip "darf" jeder Kernelteil auf jeden anderen zugreifen.
₩ keine exakte Hierarchie!

5 Grundlegende Kernelstrukturen und Funktionen

5.1 Prozessmodelle

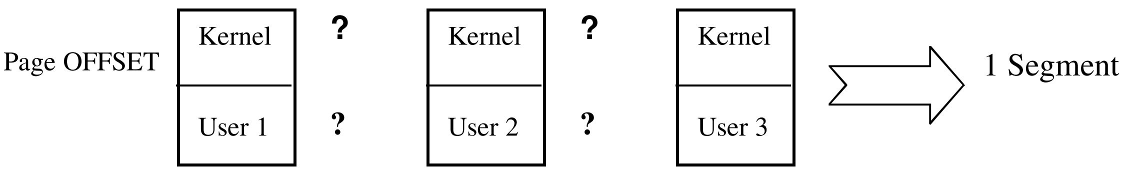
5.1.1 Ursprüngliches Modell i386 (2 Segmente Kernel)



Usermodus: DS[5] ¢ User–Segment
Kernelmodus: DS ¢ Kernel–Segment
FS ¢ User–Segment

Kernel: virtuelle Adresse = physikalische Adresse

5.1.2 Heutiges Modell (wegen anderen Architekturen)

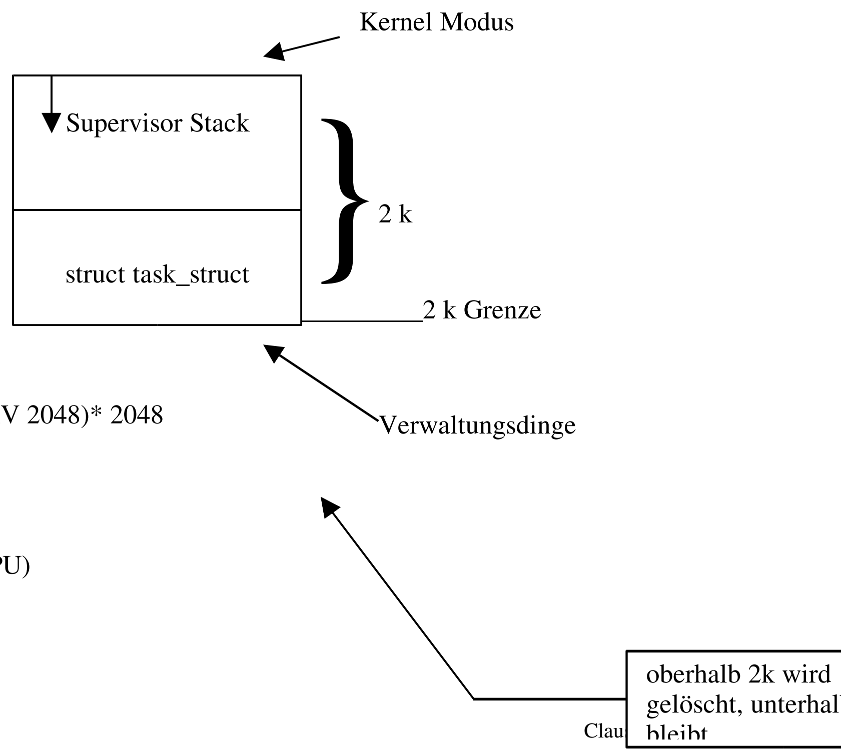


Dieses Modell wurde eingeführt, da mache Prozessoren die Adressierung von nur 1 Segment erlauben (Dec, Alpha)

ê Umschaltung der Adressräume muss ausdrücklich veranlasst werden. (internes Koroutinen–Modell)
ê Schutz: Zugriff auf Kernelsegment ist im User–Modus verboten (sonst könnte sich jeder User zum Admin machen!), "harte Grenze".

₩ Trennung User–Stack und Kernel–Stack

Implementierungstrick
pseudo–variable current:
(Zeiger auf) ¢ Prozess der gerade läuft



Mit 2 Maschinenbefehlen
MOV SP ¢ ACCU
AND ACCU, ~ ((1 << 11) –1) (ACCU DIV 2048)* 2048

(2^{11} –1) invertiert

Jeder Prozess hat SU–Stack (mind. für jede CPU)

5 Prozessorregister

ê Rekursionen im Kernelmodus sehr problematisch, Stack ist beschränkt. (512 Prozesse * 2k = 1 MB)
ê Bei SMP hat jeder Prozessor seinen eigenen Supervisor−Stackpointer, aber trotzdem den gemeinsamen Kernel−Adressraum ¢ SMP− Synchronisationsproblem.
ê Kernel−Pointer und User−Pointer sind unterschiedlich!
ê Beide sind virtuell: Physikalische Adresse = virtuelle Kerneladresse + page_OFFSET

Zugriff auf User−Segment:
Der Zugriff der Daten im User−Segment kann nie direkt, sondern nur über Makros oder Funktionen erfolgen.

```
int   access_ok   (type, addr, size);

type ? {VARIFY_READ, VARIFY_WRITE}
```

überprüft, ob lesend bzw. schreibend auf einen Useradressbereich zugegriffen werden kann.

```
varify_area ();
```

Makro:

```
get_user (result, addr)
```

Lesen eines skalaren Wertes aus dem Usersegment, wobei der *result*−Typ automatisch vom Typ von *addr* bestimmt wird.

analog:

```
put_usr (x, ptr)
```

Schreiben eines skalaren Wertes aus dem Usersegment.

```
unsignedlong      copy_to_user      (to, from, n)
unsignedlong      copy_from_user    (to, from, n)
```

Kopieren von Daten in/aus dem Usersegment.

Pointertypen:
Pointer auf Userdaten
Pointer in Kernel
auf physikalische Adresse

ê Aufpassen, welche Art von Pointer übergeben wird!

5.2 Synchronisationsmittel auf Task−Ebene

5.2.1 Unterste Ebene: void schedule (void)

Sucht einen (anderen) lauffähigen Prozess, schaltet Koroutine auf diesen Prozess um.
ê Prozess wird einfach durch Setzen / Löschen von current ¢ *state* = <*state*>

state:

TASK_RUNNING
Der Prozess wartet auf seine Zuteilung oder läuft gerade.

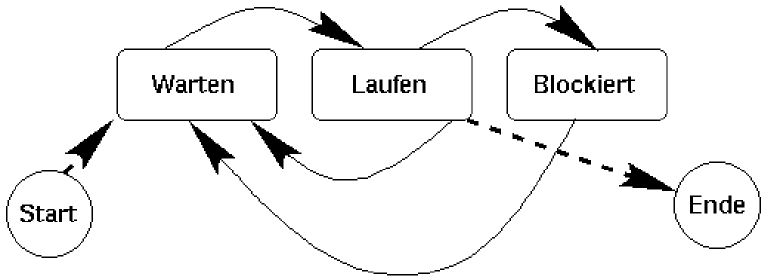
TASK_INTERUPTIBLE
TASK_UNINTERUPTIPLE
Der Prozess wartet auf bestimmte externe Ereignisse. Der Unterschied zwischen diesen beiden Werten besteht darin, dass im Zustand *TASK_INTERUPTIBLE* ein Prozess durch Signale wieder aktiviert werden kann, während er im Zustand *TASK_UNINTERUPTIPLE* in der Regel direkt oder indirekt auf eine Hardwarebedingung wartet und damit keine Signale akzeptiert.

TASK_STOPPED
Beschreibt einen Prozess dessen Ausführung angehalten worden ist.

TASK_ZOMBIE
Beschriebt einen Prozess, der beendet worden ist, dessen Taskstruktur sich aber noch in der Prozesstabelle befinden muss.

Der Scheduler ist für die Zuteilung der Ressource "Prozessor" (also Rechenzeit) an die einzelnen Prozesse verantwortlich. Es ist möglich, dass ein Prozess der Klase SCHED_FIFO so lange laufen kann wie er möchte bis er die Steuerung freiwillig abgibt. Nur ein Prozess mit höhere Echtzeitpriorität kann diesen Prozess unterbrechen.
Die Funktion *schedule()* besteht aus drei Teilen. Zuerst werden Routinen die regelmäßig aufgerufen werden müssen, gestartet. Danach wird der Prozess mit der höchsten Priorität bestimmt. Als drittes wird der neue Prozess zum aktuellen Prozess und der Scheduler hat seine Arbeit erledigt.

Ein Prozess kann folgende Zustände annehmen:



Bei der Wahl des nächsten Prozesses überprüft der Scheduler ob das Attribut *state* der *task_struct* Struktur auf TASK_RUNNING (Warten) gesetzt ist, denn nur dann ist der Prozess auch bereit. Unter besonderen Umständen (warten auf ein bestimmtes Ereignis z.B. I/O–Operationen) kann es passieren, dass ein laufender Prozess in den blockierenden Zustand wechselt wobei das *state* Attribut hierbei auf TASK_INTERUPTIBLE oder TASK_UNINTERUPTIPLE gesetzt wird. In diesem Fall wird der Scheduler diesen Prozess bei der nächsten Wahl nicht mehr berücksichtigen. Wenn das bestimmte Ereignis eingetreten ist, wird das *state* Attribut wieder auf TASK_RUNNING gesetzt und beim nächsten Durchlauf des Schedulers berücksichtigt.

5.2.2 Mittlerer Ebene: ADT *wait_queue*

Oftmals ist ein Prozess vom Eintreten bestimmter Bedingungen abhängig. Sei es, dass der Systemaufruf *read()* darauf warten muss, dass die Daten von der Festplatte in den Speicherbereich des Prozesses geladen werden, oder dass ein Elternprozess mit *wait()* auf das Ende eines Kindprozesses wartet. In jedem dieser Fälle ist nicht bekannt, wie lange ein Prozess warten muss.
Dieses "Warten bis eine Bedingung erfüllt ist" wird in Linux mit Hilfe der Warteschlange (*Waitques*) realisiert. Ein Warteschlange ist nichts anderes als eine zyklische Liste, welche als Element Zeiger in die Prozesstabelle enthält.

```
void  add_wait_queue    (struct wait_queue **queue,
                         struct wait_queue *entry);

void  remove_wait_queue (struct wait_queue **queue,
                         struct wait_queue *entry);
```

Ein Prozess, der auf ein bestimmtes Ereignis warten will, trägt sich nun in eine solche Warteschlange ein und gibt die Steuerung ab (Koroutinen–Modell). Zu jedem möglichen Ereignis gibt es eine Warteschlange. Wenn das entsprechende Ereignis eintritt, werden alle Prozesse in dieser Warteschlange wieder aktiviert und können weiterarbeiten. *queue* enthält jeweils die zu modifizierende Warteschlange (das entsprechende Ereignis).

Diese Semantik wird durch die Funktion

```
void  [interruptible_] sleep_on (struct wait_queue ** q)
```

realisiert. Sie setzt den Status des Prozesses (*current–> state*) auf den Wert *TASK_UNINTERUPTIPLE* bzw. *TASK_INTERUPTIBLE*, tragen den aktuellen Prozess (*current*) in die Warteschlange ein und rufen den *Scheduler* auf. Damit gibt der Prozess die Steuerung freiwillig ab.

Der Prozess wird erst wieder aktiviert, wenn der Prozessstatus auf TASK_RUNNING gesetzt wurde. Das geschieht in der Regel dadurch, dass ein anderer Prozess das Makro

```
void  wake_up     (struct     wait_queue **q);
```

aufruft, um alle in der Warteschlange eingetragenen Prozesse zu wecken.

5.2.3 High−Level−Ebene: Semaphore

Mit Hilfe von Warteschlangen stellt *Linux* auch Semaphore bereit. Diese dienen der Synchronisation verschiedener Routinen des Kerns auf gemeinsam benutze Datenstrukturen. Diese Semaphore sind nicht mit dem für Anwendungsprogramme bereitgestellten Semaphoren von Unix System V zu verwechseln.

```
struct semaphore {
        int         count;
        struct            wait_queue *wait;
        unsigned long     owner, owner_depth;
        int         waking;
 }
```

Ein Semaphor gilt als belegt, wenn *count* einen Wert kleiner oder gleich 0 hat. In die Warteschlange tragen sich alle Prozesse ein, die den Semaphor belegen wollen. Sie werden dann benachrichtet, wenn er von einem anderen Prozess freigegeben wird. *owner* enthält einen Verweis auf die Task, die den Semaphor gerade belegt hat, *owner_depth* gibt an, wie oft diese Task den Semaphor belegt hält. Zum Belegen oder Freigeben von Semaphoren gibt es zwei Hilfsfunktionen:

```
void  down  (struct     semaphore*  sem);  /* P-Operation */
void  up    (struct     semaphore*  sem);  /* V-Operation */
```

5.2.4 Weiteres

Weitere High−Level−Funktionen insbesondere 2 verschiedene Timer−Implementierungen

5.3 Interrupts

Zur Kommunikation der Hardware mit dem Betriebsystem werden Interrupts verwendet. Beim Entwurf einer Interruptroutine steht der Programmierer vor einem Problem. Zum einen muss die eigentliche Interruptroutine die Hardware so schnell wie möglich bedienen und dann den Prozessor für andere Aufgaben so schnell wie möglich für die Bearbeitung weiterer Interrupts, wieder freigeben. Zum anderen kann so ein Interrupt aber auch die Verarbeitung einer größeren Datenmenge auslösen.
Um dieses Problem zu lösen, erfolgt die Interruptbehandlung unter Linux in zwei Phasen. Zuerst wird die zeitkritische Kommunikation mit der Hardware durchgeführt, hierbei sind eventuell andere Interrupts gesperrt. Die eigentliche Verarbeitung der Daten erfolgt asynchron zum eigentlichen Interrupt in den sogenannten "Bottom Halfs" der Interrupthandler. Dies sind Funktionen, die zu einem späteren Zeitpunkt aufgerufen werden. Sie können ihrerseits auch wieder von anderen Interrupts unterbrochen werden.

ê Können "jederzeit" auftreten ¢ liegen quer zum Koroutinen−Modell
¢ besondere Sorgfalt notwendig, z.B. darf in Interrupts kein Schedule() aufgerufen werden! (Zerstörung des Koroutinen−Modells, *schedule*() könnte rekursiv unterbrochen werden) Re−Entranz−Problem (Wiedereintrittsproblem)
₩ Interrupts müssen gesperrt werden können, bei allen Prozessortypen über das Statusregister.

Makros: `__cli() /* Interrupts sperren */` (lokal)
`__sti() /* Interrupts wieder zulässig*/` (lokal)
`save_flags (flags)` ¢ Interrupts merken
`restore_flags (flags)` ¢ Interrupts wiederherstellen

cli() und sti() für alle Interrupts auf allen Prozessoren (bei SMP−Kernel)

ê Interrupt−Programmierer müssen aufpassen, keine Kernelfunktionen aufzurufen, die blockieren können!
ê In Interrupts darf man nicht warten!

Beispiel Interruptbetrieb:
Im Interruptbetrieb benachrichtigt das Gerät die CPU über einen Interruptkanal (IRQ), wenn es eine Operation beendet hat. Die CPU unterbricht den laufenden Betrieb und führt eine Interruptserviceroutine (ISR) aus. Innerhalb der ISR erfolgt dann die weitere Kommunikation mit dem Gerät. So wird ein Prozess, der auf die parallele Schnittstelle im Interruptbetrieb schreiben will, vom Gerätetreiber im Interruptbetrieb nach dem Schreiben eines Zeichens angehalten. Kann die parallele Schnittstelle weitere Zeichen entgegennehmen, löst sie einen IRQ aus. Die behandelnde ISR weckt den Prozess daraufhin wieder (*current−>sate* = TASK_RUNNING) und der Vorgang wiederholt sich.
Zuerst wird die Schnittstelle ermittelt, die den Interrupt auslöste, und danach der wartende Prozess mit *wake_up()* "wachgeküsst" .

5.4 Synchronisationsmittel auf CPU−Ebene (SMP)

Bei Symmetric Multi Processing (SMP) können mehrere Prozessoren in kritischen Abschnitten um die selbe Ressource konkurrieren. Um dieses Problem zu lösen, wurden in den Linux−Kernel *Spin Locks* eingeführt. Sie realisieren den gegenseitigen Ausschluss von Prozessoren im Kernel. Der kritische Abschnitt kann nur von dem Prozessor ausgeführt werden, der sich gerade im Besitz des Spin Locks befindet.

ê liegt nochmals "quer", sowohl zum Koroutinen−Modell als auch zu den Interrupts!
¢ Spinnlocks

ADT spinlock_t; ("bringe und lösche")

```
void  spin_lock_init    (lock);
void  spin_lock         (lock);
void  spin_unlock       (lock);
int   spin_trylock      (lock);
```

Wegen Problemen mit Interrupts:

```
spin_[un]lock_irq (lock)
```

verhindert Interrupts für die Dauer des gehaltenen Locks.

ê Bei Schachtelung mehrerer Locks geht das schief!
Lösung:

```
spin_[un]lock_irq_save (lock);
```

Um einen kritischen Bereich zu betreten, muss *spin_lock()* aufgerufen werden. Falls sich ein Prozessor bereits in diesem Bereich befindet, bleibt der aktuelle Prozessor so lange im Spinlock[6] gefangen, bis der andere Prozessor den kritischen Bereich verlassen hat.
Verlässt ein Prozessor einen kritischen Bereich, muss *spin_unlock()* aufgerufen werden. Vergisst man diesen Aufruf, kann schnell der Rechner einfreieren, da danach kein Prozessor mehr diesen Bereich betreten kann.
Mit Hilfe von *spin_trylock()* kann abgefragt werden, ob ein Bereich betreten werden kann. Liefert diese den Wert 0, ist der Bereich momentan belegt. Ansonsten ist er frei. Ein weiteres *spin_lock()* ist nicht mehr erforderlich, da *spin_trylock()* den Spinlock belegt.

Weitere Möglichkeiten in <asm/atomic.h> atomare Operation für Zähler:

```
z.B.   atomic_inc()
       atomic_dec()
       atomic_add()
       atomic_sub()
```

Diese Funktionen erlauben es, den Wert einer atomaren Variable durch Addition oder Subtraktion eines Werts zu ändern.

5.5 Allgemeine Hilfsroutinen

Meldungen an Systemkonsole. Funktion um Kontroll− und Debugmeldungen ausgeben zu können. Es werden nur Mitteilungen die wichtig genug sind auf der Konsole ausgegeben (d.h. in der Priorität höher als das Loglevel liegen). Das Loglevel kann durch den Systemruf *syslog* eingestellt werden.

`int printk (char* fmt, ...)` eingeschränkte Version von printf

```
void  panic (char*      str);
```

Diese Funktion ist ein *printk()* mit der festen Priorität KERN_EMERG. Zusätzlich wird die Kernfunktion *sys_sync()* aufgerufen.

```
<linux/string.h>
char* strcpy      (           char*       ziel,
                   const      char*       quelle);
```

kopiert den String quelle einschließlich Nullbyte.

```
char* strncpy     (           char*       ziel,
                   const      char*       quelle,
                              size_t      count );
```

nur die ersten count−Zeichen werden kopiert.

---

[6] Der "gefangene" Prozessor muss dann endlos "Runden drehen". Daher der Name Spinlock.

```
size_t       strlen       (char*       str);
```
liefert sie Anzahl der Zeichen in str, ohne das Nullbyte.

Folgende Funktionen bearbeiten Speicherbereiche. Normalerweise handelt es sich um Speicher in Form von Zeichenketten, die nicht durch ein Nullbyte abgeschlossen sind.
```
void*        memset       (void*       addr,
                           int         inival,
                           size_t      len);
```
füllt den mit addr adressierten Bereich der Größe *len* mit *inival*

```
void*        memcpy       (void*       ziel,
                           void*       quelle,
                           size_t      len);
```
der Bereich *ziel* wird mit den ersten *len* Zeichen aus *quelle* gefüllt.

Folgende Funktion setzt das Bit *nr* an der Adresse *addr*.
```
<asm/bitops.h>
void         set_bit      (int   nr,
                           void* addr);
```

5.6 Speicherallokation im Kernel

Der physische Speicher wird in Pages strukturiert. Die Größe der Pages ist durch das Makro PAGE_SIZE definiert. Für 32–Bit–Architekturen wie x86 beträgt die Größe meist 4 KB.

5.6.1 Low–Level

```
unsigned    long  __get_free_page   (int   gfp_mask)
```
¢ Pointer ¢ Typecast notwendig!

`GFP_KERNEL`
(GFP_USER ¢ der laufende Prozess darf zum Auslagern von Speicherseiten unterbrochen werden)
`GFP_ATOMIC`
(Die Funktion *__get_free_page* darf den laufenden Prozess nicht unterbrechen, aber sollte nach Möglichkeit eine Speicherseite zurückliefern) ¢ garantiert kein *schedule().*

`GFP_KERNEL | GFP_DMA` oder
`GFP_ATOMIC | GFP_DMA` ¢ liegt unter 16 MB–Grenze (braucht man nur bei Intel)

Eigenschaften:

- Speicherbereich hat Größe `PAGE_SIZE`
- beginnt an einer `PAGE_SIZE`–Grenze

```
unsigned    long  __get_free_pages  (int              gfp_mask,
                                     unsigned long    gfp_order);
```

₩ Größe ist (PAGE_SIZE << gfp_order). Ein Block mit der Ordnung x ist 2^x Speicherseiten groß.

Weiterer Versionen ohne __ löschen den allokiertern Speicher

ê Allokierter Speicher beginnt an der Größen–Grenze

5.6.2 Objekt–Caches

<linux/slab.h>
<linux/simp.h>

ADT:
```
kmem_cache_t*     kmem_cache_create (const      char*              name,
                                                size_t             size,
                                                size_t             offset,
                                                unsigned long      flags,
                                                void(x)(void*,  kmem_cache_t*

                                                ...                      );
```

Konstruktor, wird von der Cache–Verwaltung beim <u>ersten Erzeugen eines Objekts aufgerufen</u>

Mit *kmem_cache_create* wird ein neuer Cache für Objekte der Größe *size* angelegt. Mit *offset* kann ein Alignment für die Objekte, kleiner als die Objekte selber angefordert werden.

Die Allokation und Freigabe einzelner Objekte ist mit

```
void* kmem_cache_alloc  (kmem_cache_t      *,
                         int               gfp_mask);

void  kmem_cache_free   (kmem_cache_t      *,
                         void*             obgp);
```

möglich. Bei *kmem_cache_alloc()* können in den Flags die Priorität für das Allokieren einer Speicherseite angegeben werden.

Idee:
Objekt muss nur einmal konstruiert werden, `kmem_cache_free()` gibt Objekte zurück, die noch (teilweise) sinnvolle Informationen enthalten.

5.6.3 Heterogener Heap im Kernel Speicher

```
void* kmalloc     (size_t     size,
                   int        gfp_mask);
```

versucht Speicher mit der Größe *size* zu reservieren. Es ist die Reservierung von Speicher bis zu 128kByte möglich. Der Speicher ist aus dem **physikalischen Speicherbereich** allokiert.

```
void  kfree       (const      void  *obj);
```

so kann der reservierte Speicher wieder freigegeben werden.

5.6.4 Virtueller Adreßraum

```
void* vmalloc     (size_t     size);
void  vfree       (void*      addr);
```

Blendet vielfaches von Speicherseiten in den **virtuellen Adressraum** des Kernels ein, lässt Lücken von mind. 1 Seite im virtuellen Adressraum.
¢ allokierter Speicher beginnt an Page–Grenzen

5.7 Wichtige grundlegende ADT'S im Kernel

5.7.1 struct task_struct

(sehr umfangreich) beinhaltet sowohl User– als auch Kerneldaten für Prozesse.
Dazu globales Array

```
struct task_struct *task[NR_TASK];
```

Das Array ist auf max. 512 Prozesse beschränkt!

5.7.2 struct mm_struct

Beschreibt Layout eines virtuellen Adreßraumes.
¢ neu 1 Instanz pro *task_struct–Instanz* notwendig.

5.7.3 struct vm_area_struct

Beschreibt Segment eines virtuellen Adressraumes.

ê Eine *vm_area_struct–Instanz* kann Mitglied in mehreren *mm_struct–Instanzen* sein.
¢ Inoffizieller *clone()*– Systemaufruf

5.7.3.1 struct vm_operations_struct
Enthält Methoden zur Verwaltung eines Segments.

5.7.4 struct page
Beschreibt physikalische Speicherseite
(1:1 Zuordnung zwischen *mem[]* und der Speicherseite)

5.7.5 struct inode
Beschreibt genau ein (passives) Dateiobjekt

ê bei Hardlinks ist nur 1 Instanz vorhanden
ê Objekt muss keinen Namen haben, z.B. anonyme Pipes

5.7.6 struct dentry
Beschreibt Namen für ein Dateiobjekt.

5.7.7 struct file
Beschreibt Kanal für ein Dateiobjekt (es kann mehrere Kanäle zu einer Datei geben)
ê Es können mehrere (unabhängige) Kanäle für dasselbe Dateiobjekt existieren!

5.7.8 struct super_block
Beschreibt Dateisystem–Typ, z.B. ext2, iso9660, nfs, ...

5.7.9 struct {inode, super, dentry, file}_operations
ê theoretisch sollten hier die zur jeweiligen Struktur gehörenden Methoden stehen, es herrscht aber Chaos!

5.7.10 struct buffer_head
Beschreibt Platten–Datenblöcke, zur Verwendung in Dateisystem–Implementierungen
ê Ursprünglicher Zweck war Buffer–Cache, dessen Funktion inzwischen teilweise durch einen Page–Cache realisiert ist!

6 Programmierstil und Tricks

ê C verführt zum Chaos, daher Dokumentation / Coding Style beachten!!

- Einrückung 8 Zeichen ¢ tiefe Schachtelungen vermeiden
- Funktionen sollten kurz und einfach sein und genau 1 Sache machen. KISS (**K**epp **I**t **S**imple And **S**tupid)
  ¢ komplizierte Funktionen aufbrechen ¢ Modularisierung
- Kommentare möglichst nur am Beginn einer Funktion, sollte das **WAS** erklären, nicht das **WIE**!

Generelle Attitüde von Linus Torvalds: Flags sind schlecht!
¢ Aber Realität sieht aber leider anders aus!

Aufbrechen in Teilfunktionen kostet theoretisch nichts, da man Funktionen <u>inline</u> deklarieren kann.
ê Praxiserfahrung zeigt das Gegenteil:
    Je mehr inline−Funktionen, desto mehr Prozessorregister werden gebraucht!
    ¢ zumindest bei Intel führt dies zu Register−Thrashing

₩ Funktionen sollten möglichst wenig Variablen benutzen und möglichst wenig Parameter.

6.1 C−Hacker−Tricks

Beruhen auf der Interpretation von Werten als Bitmuster. ¢ Programmierer sollte zweier Potenzen auswendig kennen.

1. (x << y) ¢ (x * 2^y)
2. (x & ((1 << y)−)) ¢ x % (2^y)
   ¢ cons, sofern y = const
   ¢ Hashtabellen werden häufig mit diesem Trick realisiert
   ¢ viele Tabellen haben zweier−potenzen−Größe
3. Constant−Expression−Evaluation möglichst überall nutzen

4. Das VFS

VFS = Virtual Filesystem Switch
Ziel: einheitliche Schnittstelle für unterschiedliche Dateisystem–Typen

7.1 Der DCache und der Inode–Cache Ausgangssituation

Um die Anzahl der Zugriffe auf die Festplatte beim Lesen und Schreiben der Inodes möglichst niedrig zuhalten, werden diese beiden Caches verwendet. Es handelt sich dabei um nichts anderes als zwei interne Zwischenspeicher, in dem sich die Daten von häufig gebrauchten Inodes bzw. Verzeichnissen befinden. Wenn das VFS auf eine solche Struktur zugreift, prüft es zuerst, ob sich die gewünschten Daten in den Caches befinden. War die Prüfung erfolgreich, können diese Daten verwendet werden, ohne dass irgendwelche anderen (zeitaufwendigen) Routinen zu ihrer Beschaffung aufgewendet werden müssen.

	Speichermodul	**Kernel–Space**	**User–Space**
Dateien	(Disk–Inode)	struct inode ¢ Inode–cache	struct stat
Dateinamen / Verzeichnisse	[fs–spezifische Datei mit name, inode–no P...?]	–––	struct dirent

ê Der Begriff "inode" hat zwei Bedeutungen:
1) Struktur auf dem Speichermedium
2) Struktur im Kernel–Space

Ursprünglich waren beide Bedeutungen gleich (binäres Abbild der Kernelstruktur auf der Platte)
>> !! Geht nicht mit den verschiedenen Dateisystem–Typen !! <<
¢ Übersetzung notwendig (¢ VFS)

Die Struktur Inode enthält im wesentlichen Informationen über die Datei. Zusätzlich befinden sich in dieser Struktur Verwaltungsinformationen sowie die dateisystemabhängige Union *u*.

```
struct inode{
      /* Verwaltungs-infos */
      unsigned long     i_ino;      /* Inode-Nummer */
      unsigned int      i_count;    /* Referenzzähler */
      kdev_t            i_dev;      /* Gerätenummer */
      int               i_nlink;    /* Hardlinks */
      mode_t            i_mode;     /* Typ der Inode */
      uid_t             i_uid:      /* Eigentümer */

      ...               ...   (siehe struct stat Seite 8)

      struct      semaphore         i_sem;
      struct      inode_operations* i_op;
      struct      super_block*      i_sb;

      ...

      union {
            struct      pipe_inode_info   pipe_i;
            struct      minix_inode_info  minix_i;
      } u;
```

7.1.1 Generische Inodeoperationen

```
struct inode*     get_empty_inode   (void);
```
liefert eine "freie" Inode.

Für den schnellen Zugriff auf Inodes sind die Inodes zusätzlich in einer offenen Hashtabelle *hash_table[]* untergebracht.

```
void              insert_inode_hash (struct inode*    inode);
void              remove_inode_hash (struct inode*    inode);
void              clear_inode       (struct inode*    inode);

struct inode*     iget              (struct superblock*     sb,
                                     unsigned long          ino);
```

Liefert die durch den Superblock *sb* und die Inode–Nummer *ino* angegebene Inode. Kann die Inode in der Hashtabelle gefunden werden, wird der Referenzzähler i_count erhöht.

```
void                iput                (struct inode*    inode);
```

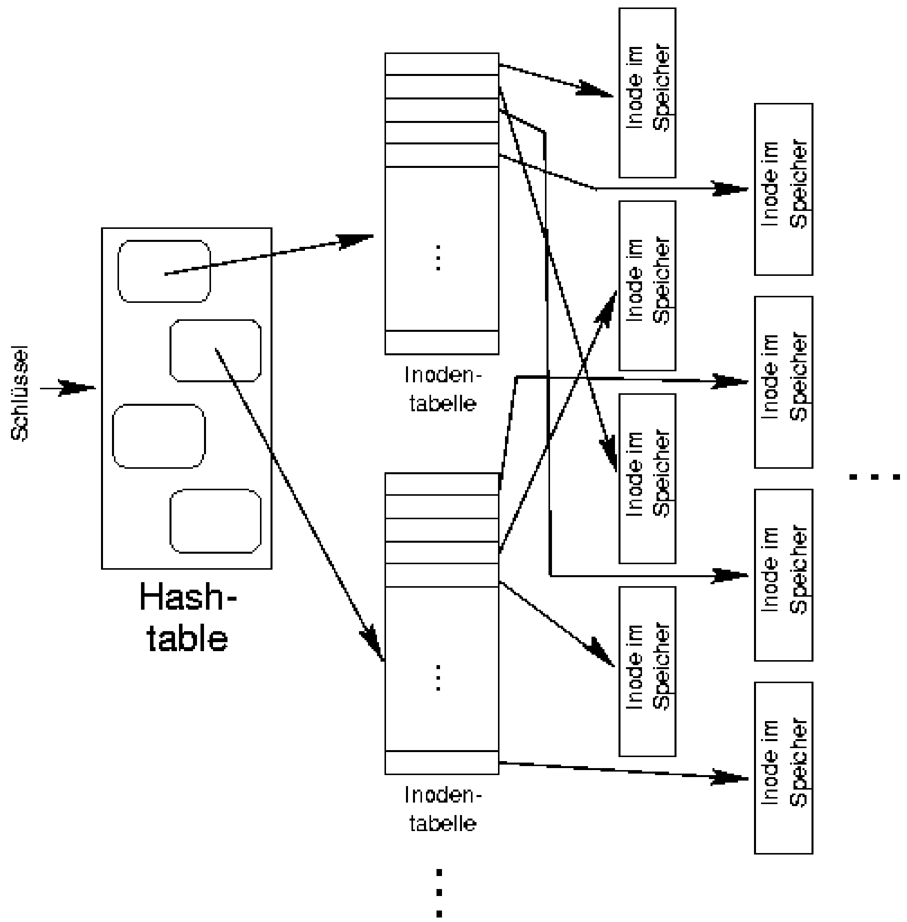
Eine mit *iget()* erhaltene Inode muss mit der Funktion *iput()* wieder "freigegeben" werden. Diese verringert den Referenzzähler um 1 und markiert die Inode–Struktur als "frei", wenn dieser 0 ist.

Inodes können von mehreren konkurrierenden Prozessen verwendet werden ¢
ê Race–Conditions
¢ Referenzzähler–Methode
i_count = 0: Inode nicht in Verwendung, darf aus Kernelspace entfernt werden
i_count > 0: Es gibt i_count Referenzen (Pointer), Inode darf nicht zerstört werden.

7.1.2 Funktionsweise eines Caches:

- Wenn Element benötigt wird ((*iget()*), wird es über den Primärindex (hier (dev, inoNr)–Paar) gesucht (z.B. über Hashtabelle)
  ¢ weiterer *iget()* sieht zwar zuerst in Hashtabelle nach, falls gefunden, wird i_count erhöht und der Inode–Pointer zurückgeliefert.
- Wenn Element nicht mehr benötigt wird *iput()* ausgeführt. Führt dies zu *i_count* = 0, wird eine LRU–Liste der inaktiven Elemente verwaltet.

ê Gesamtzahl der im Kernel–Memory gehaltenen Elemente ist begrenzt.
¢ letztes Element der LRU–Liste wird bei Bedarf wiederverwendet.



Was passiert allerdings, wenn sich eine Inode (noch) nicht in der Hashtabelle befindet? In diesem Fall muss eine VFS–Inode im Speicher mit den Werten der Inode aus dem untergeordneten Dateisystem belegt werden. Wie wird aber die VFS–Inode ermittelt, die mit den Daten belegt werden soll? Die Zahl der hashbaren Inodes ist kernelseitig begrenzt (und zwar durch die Konstante NR_IHASH, die üblicherweise auf 512 eingestellt ist). Wenn die Zahl noch nicht ausgeschöpft ist, ist die Situation einfach: Das Dateisystem braucht nur den benötigten Speicher zu reservieren und kann diesen zur Aufnahme von Inodes verwenden. Komplizierter wird das Ganze, wenn die Grenze bereits erreicht ist, da das Dateisystem in diesem Fall eine Inode aus dem Cache entfernen und dafür eine neue integrieren muss. Um zu entscheiden, welche Inodes entfernt werden soll, orientiert sich Linux daran, *wie wichtig* die Inode im Moment für das System ist. Mit Hilfe der Variablen *i_count,* kann festgestellt werden, von wie vielen Prozessen die Inode im Moment verwendet wird. Wenn dieser Zähler gleich 0 ist, wird die Inode nicht mehr gebraucht und kann daher aus dem Cache entfernt werden, um den benötigten Platz zu schaffen. Dieser Mechanismus ermöglicht es übrigens auch, sicherzustellen, dass die für das System wichtigen Inodes nie aus dem Cache entfernt werden, indem ihr *i_count* immer größer 0 gehalten wird.

7.1.3 Problem bei bisherigen Implementierungen der Namesauflösung:

1) Zugriff auf *dirents* findet nicht−zentral sowohl in *readdir()* als auch in *lookup()* statt.
   lookup() =
   a) Directory scannen, ob (name, ino)−Paar vorhanden
   b) *iget*(ino) aufrufen
      ¢ Inode−Pointer rurück liefern
2) *lookup()* durchsucht jedes Mal das gesamte Directory (wird sehr oft im Kernel aufgerufen)

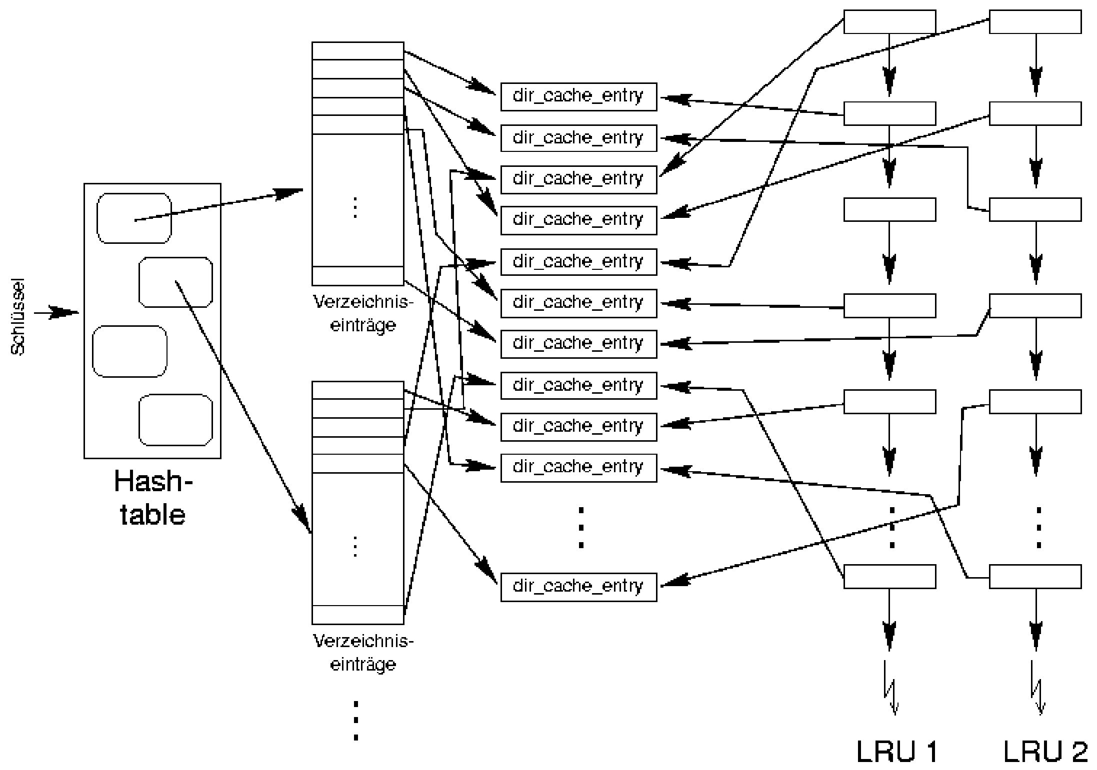
¢ "alter Dcache" war eine ad hoc − Methode für Problem 2) Hashtabelle mit (pfad, inode−Pointer)

7.1.4 Neuer DCache

	Speichermedium	Kernel−Space	User−Space
Datei−Inhalte	[disk−inode]	struct inode	struct stat
Datei−Namen	fs−spezifisches Format	struct dentry	struct dirent

```
struct dentry {
     int               d_count;    /* Referenzzähler */
     struct inode*     d_inode;
     struct dentry*    d_parent; ¢ Vater wird "gehalten"
     char*             d_name;
     list_link         d_subdirs;  /* Anker */
     list_link         d_child;    /* Mitglieder */
     list_link         d_hash;     /* zum schnellen Finden durch Namen */
     [...]
     [Flags und andere Dinge]
};
```

Der DCache arbeitet teilweise nach dem gleichen Prinzip wie der Inode−Cache, ist allerdings etwas komplizierter aufgebaut. Basis das Caches ist ebenfalls eine Hashing−Funktion. Die Werte der Tabelle sind wiederum Zeiger auf Listen mit Verzeichnisnamen (und den entsprechenden Inodes). Allerdings kann als Schlüssel für den Hash nicht nur wie beim InodeCache eine Kombination aus Inode−Nummer und untergeordnetem Gerät verwendet werden, sondern zusätzlich auch der Name des Verzeichnisses.
Weiter verfeinert wird die Caching−Technik durch Verwendung zweier LRU−Listen (Least Recently Used). Wenn ein Verzeichnis von einem Programm verwendet wird, setzt Linux dessen Namen, zusammen mit der Nummer der assoziierten Inode, in die Hashtabelle und damit in den Cache. Zusätzlich allerdings wird die gecachte Datenstruktur noch in die erste LRU−Liste aufgenommen und zwar genau an das Ende der Liste; das bisher letzte Element wird vorletztes Element und das bisher erste Element fällt vorne aus der Liste.
Wenn nun nochmals auf das Verzeichnis zugegriffen wird, wird der Eintrag ans Ende der zweiten LRU−Liste verschoben. Weitere Zugriffe auf das Verzeichnis bewirken jeweils, dass der Cache−Eintrag wieder ans Ende der zweiten LRU−Liste gesetzt wird.
Wird lange nicht mehr auf ein Verzeichnis zugegriffen, verschiebt sich der Eintrag immer weiter ans vordere Ende der LRU−Liste, bis er schließlich aus der Liste fällt.
Die maximale Zahl der Verzeichniseinträge in eine LRU beträgt defaultmäßig 128 (festgelegt durch DCACHE_SIZE), was eine Gesamtgröße von 256 Einträgen ergibt.
Ebenfalls interessant ist, dass nur Verzeichnisnamen bis zu einer maximalen Länge von 15 Zeichen (DCACHE_NAME_LEN) in die Hashtabelle aufgenommen werden. Da nur weinige Verzeichniseinträge diese Länge überschreiten, kann diese Beschränkung sowohl Speicherplatz als auch Rechenzeit sparen.



7.1.5 Generische Dentry–Operationen

```
struct dentry*    d_alloc     (          struct dentry*    parent,
                               const     struct qstr*      name);
```

wobei

```
struct qstr{
      char* name;
      int   len;
}
int               d_invalidate     (struct dentry*   entry);
struct dentry*    dget             (struct dentry*   entry);
void              dput             (struct dentry*   entry);
void              d_add            (struct dentry*   entry,
                                    struct inode*    inode);
```

ê Falls d_inode == NULL ¢ "negatives dentry" ₩ Name hat kein zugehöriges Inode!

```
void              d_move           (struct dentry*   entry,
                                    struct dentry*   newdentry);
```

(kopiert Inhalt hinüber)

```
struct dentry*    d_loockup        (struct dentry*   dir,
                                    struct qstr*     name);
void              d_delete         (struct dentry*   entry);
```

7.2 Der Buffer–Cache

Der Buffer Cache hat mit den eigentlichen Strukturen des Dateisystems keine Verbindung. Vielmehr beschleunigt er das Schrieben/Lesen der vom Dateisystem benötigten "Rohdaten" auf das untergeordnete Gerät, aus denen dann die für das Dateisystem notwendigen Informationen gewonnen werden.
Obwohl es in einem Linux–System üblicherweise verschiedene blockorientierte Geräte gibt, wird für alle nur ein einziger Cache verwendet, d.h. die Datenstrukturen des Caches gelten universell für alle Blockgeräte. Bei der Initialisierung des Caches werden zuerst Listen mit freien Puffern angelegt, die Daten zur Zwischenspeicherung aufnehmen können. Da sich die Blockgrößen der einzelnen Geräte voneinander unterscheiden, werden 5 verschiedene Listen angelegt, in denen jeweils Blöcke mit 512, 1024, 2048, 4096, 8192 Bytes abgelegt werden können. Alle Dateisystem–Module kommunizieren mit den Gerätetreiber über eine Datenstruktur, die die Bezeichnung *buffer_head* trägt. Die Liste, in der die freien Puffer verwaltet werden, enthält darum eben solche Strukturen.

Wenn nun ein Block gelesen werden soll, sucht Linux zuerst im eigentlichen Buffer−Cache danach. Der Cache besteht aus einer Hashtabelle, die auf Listen aus Blöcken zeigt, sowie einer LRU−Liste (der Schlüssel für den Hash wird dabei aus der Blocknummer und der hexadezimalen Kennzahl des assoziierten Geräts gebildet). Falls sich der Block im Hash befindet, kann er für die weiteren Operationen verwendet werden, ansonsten wird ein neuer Block aus der Liste der freien Blöcke entfernt, mit den entsprechenden Daten gefüllt und in den Hash integriert.
Neben den eigentlichen Nutzdaten wird unter anderem eine zusätzliche Variable *(b_state*) gespeichert, die den aktuellen Zustand des Buffers angibt. Sie ist ein Feld aus Bits, in dem verschiedene Werte kombiniert werden können; insgesamt existieren 8 Stück davon, wobei an dieser Stelle allerdings nur einer interessant ist: *BH_Dirty*. Wenn das entsprechende Bit gesetzt ist, zeigt dies, dass der Inhalt des Buffers verändert wurde und daher nicht mit den Daten auf der Festplatte übereinstimmt. Buffer mit diesem Status müssen von Zeit zu Zeit auf die Festplatte geschrieben werden, damit die Daten dauerhaft gespeichert sind. Dies übernimmt die Kernel−Routine *bdflush:* Wenn die Zahl der "dreckigen" (dirty) Puffer einen bestimmten Prozentsart der insgesamt zur Verfügung stehenden Puffer erreicht hat, wird diese Routine aufgerufen und "bereinigt" die Situation, indem es die wie gezeigt markierten Puffer auf die Festplatte verewigt.

```
struct buffer_head {
      unsigned long     b_blocknr;  /* Blocknummer */
      unsigned long     b_size;     /* Blockgröße */
      kdev_t            b_dev;      /* Gerät */
      unsigned long     b_state;    /* Pufferstatusbitmap */
      unsigned int      b_count;    /* Anzahl der Benutzer dieses Blocks */
      char*             b_data;     /* Zeiger auf Datenblock */
      [?]
};

struct buffer_head*     getblk      (kdev_t     dev,
                                     int        block,
                                     int        size);
```

¢ Versucht, den Puffer im Cache zu finden.
ê Puffer braucht nicht aktuell zu sein!

```
struct buffer_head*     bread       (kdev_t     dev,
                                     int        block,
                                     int        size);
```

Um einen Block zu lesen, ruft eine Systemroutine die Funktion *bread()* auf. Zuerst wird überprüft, ob für den Block *block* zum Gerät *dev* nicht schon ein Blockpuffer vorhanden ist. Wird der Puffer gefunden, und ist das Flag *BH_Uptodate* gesetzt, wird *bread()* mit der Rückgabe des Blockpuffers beendet. Ist das Flag nicht gesetzt, muss der Puffer durch lesen des externen Mediums aktualisiert werden und die Routine kann zurückkehren.

```
void        brelse      (struct buffer_head*    bh);
```

Mit b*relse*() (block release) soll der von *bread()* zurückgegebene Speicherblock, wenn er nicht mehr benötigt wird, freigegeben werden. Modifizierte Puffer werden hierbei asynchron zurückgeschrieben.

```
void        bforget     (struct buffer_head*    bh);
```

Zurückschreiben unnötig!

ê Zustandsmodell des Buffer−Caches ist sehr komplex!

Buffer−Cache realisiert ein <u>virtuelles</u> <u>Array</u> von Blöcken (á 512 Bytes)

₩ In Zukunft Byte−Offsets verwenden!

7.3 Der virtuelle Switch

Idee: virtuelle Methoden mittels Pointer auf Funktionen simulieren.
(mehrere in struct* −operationen gesammelt)

ê Stimmt nicht vollständig:

1) Man kann die Existenz einer Implementierung prüfen (ob pointer == null) ₩ default−operationen ausführen.
2) Man kann Ponter auf Funktionen <u>einzeln</u> zur Laufzeit auswechseln.
   Z.B.: In der Pipe−Implementierung.
   Je nach Open−Flags verschiedene Read/Write−Funktioenen eingeführt

₩ Prinzip der Funktions–Pointer ist echt mächtiger als OOP!

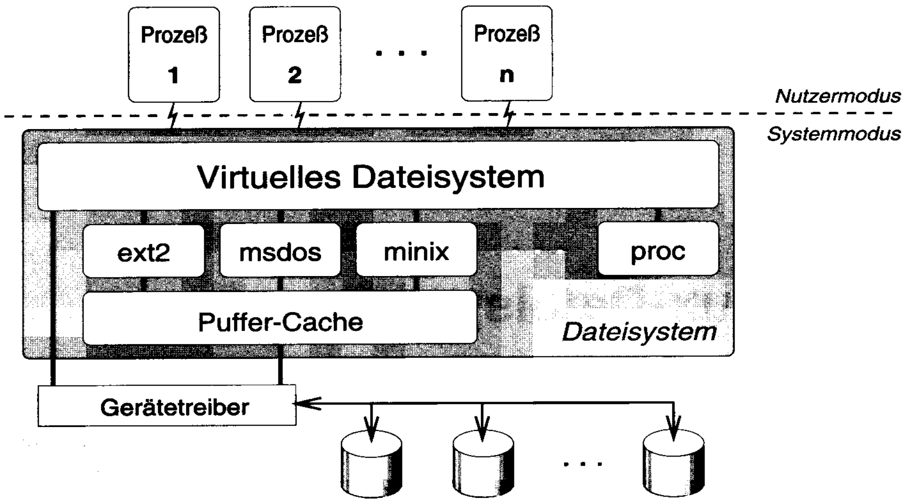
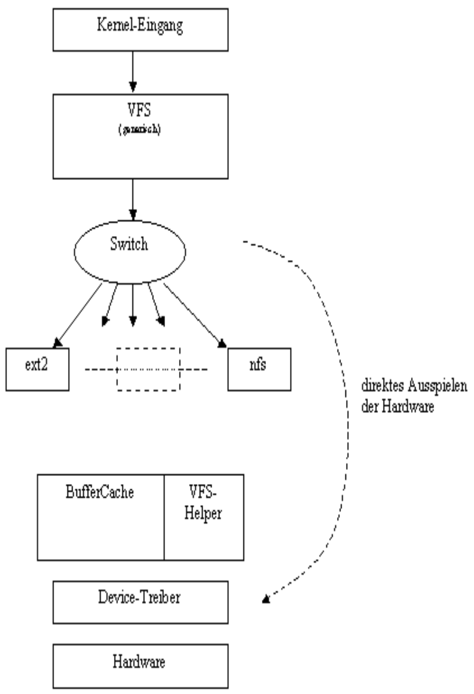


Abbildung 6.1: Die Schichten des Dateisystems

Architektur:



7.3.1 VFS–helper (Auszug)

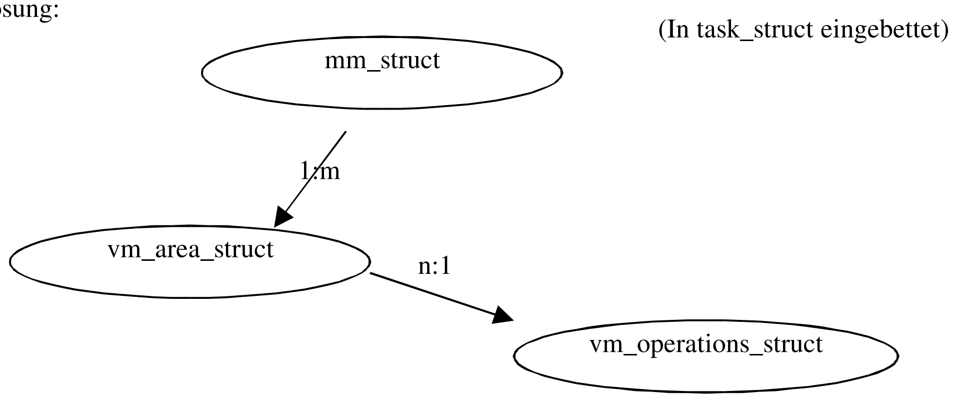
```
mit [un] register_filesystemtype    (struct file_system_type*     ini);
struct file_system_type {
      const       char* name;
      int         fs_flags;
      struct      super_block*(*read_super)     (struct super_block*    sb,
                                                 void*      data,
                                                 int        silent);
};
```

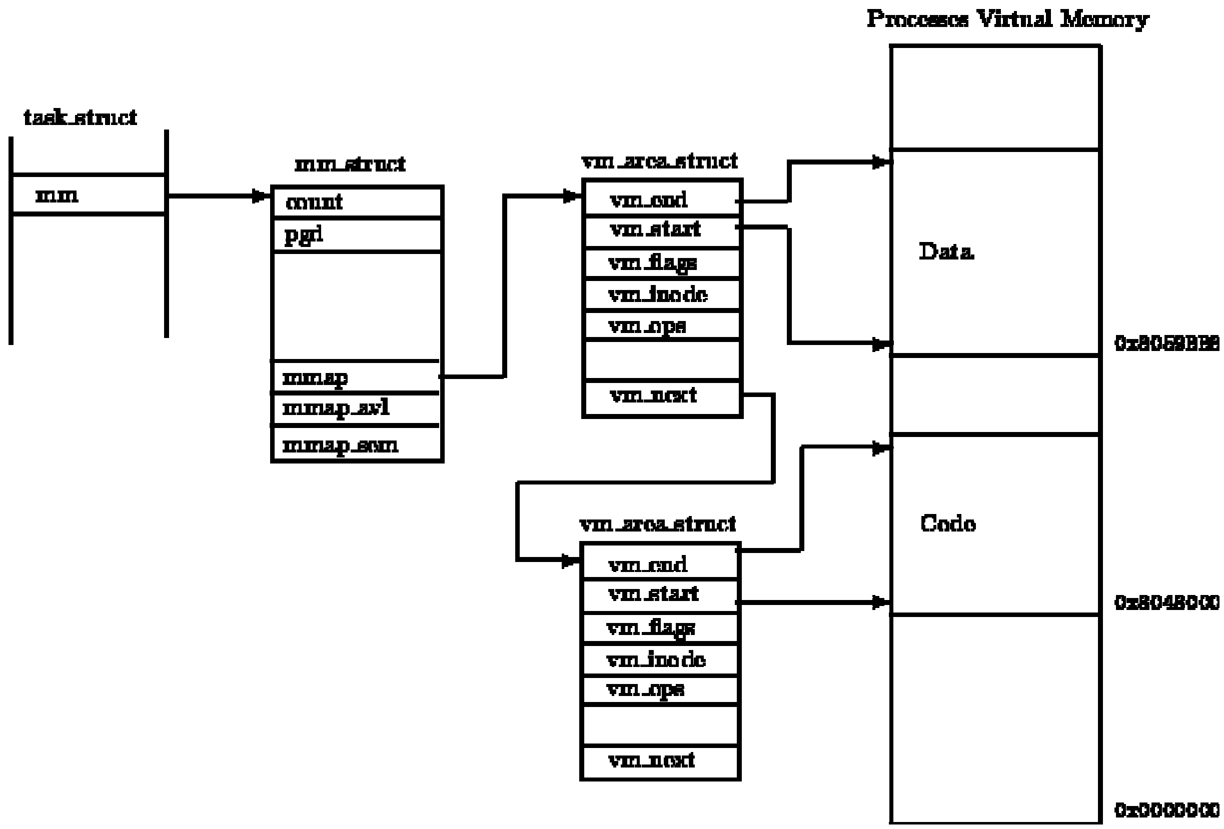
Über *read_super()* wird beim Mounten der Superblock gelesen, von dort aus sind Pointer auf root–dentry und *super__operations* erreichbar.

8 Virtual Memory

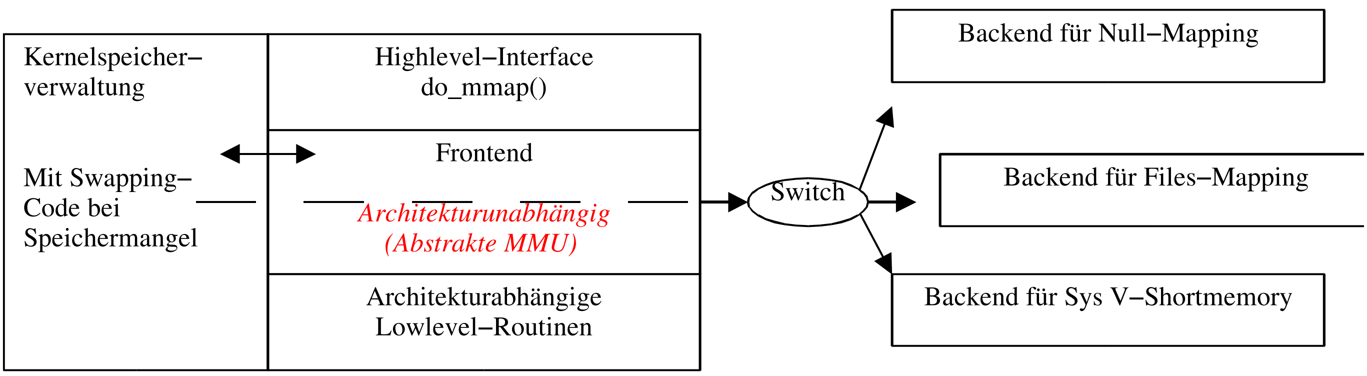
Problem: Nutzeradressraum besteht aus mehreren Bereichen, die aus unterschiedlichen Quellen stammen können.

Lösung:





Rudimentäres Konzept einer Virtualisierung



(MMU: Memory Management Unit ¢ gibt's nur bei Linux!)

ê Hauptproblem: fast alle Mapping/Unmapping−Operationen geschehen in Interrupt−Routinen! Statusinformationen sind in *vm_area_struct*, *struct page* und in PTE's (Page−Table−Entries) verteilt!

Vorgehensweise (Strategie):

a) Null−mapping:
Ist bei einem Aufruf vom do_map() file == null, dann spricht man von einem *anonymen Mapping*. Wird auf Speicherseiten lesend zugegriffen, wird ein Verweis auf immer dieselbe leere Seite eingefügt. Bei Schreiben einer Seite in einem anonymen Virtuellen Speicherbereich, wird eine leere Seite initialisiert.
Es gibt eine reservierte Speicherseite, die nur Null−Bytes enthält. Diese Seite wird mehrfach eingebettet, aber read−only. beim ersten Schreibzugriff wird durch wp_page() eine andere (gelöschte) Seite eingeblendet ¢ anonymes Mapping

b) File−Mapping: (startet mit leeren PTE)
nopage() holt die gewünschte Seite. Bei RW−Mapping und beim ersten Schreibzugriff wird die Seite auf dirty gesetzt.

c) Sys V ist vollkommen getrennt.

ê Der Swapping−Code ist separat von Virtualisierungs−Architektur
¢ treten sich gegenseitig auf die Füße.
¢ neue Architektur!

9 Literatur

- Beck et. al.: Linux−Kernel−Programmierung Addison−Wesly, 1977 a. Auflage
- Bach : The Design Of The Unix Operating System, Prentice Hall 1986
- Stevans : Advanced Programming in the Unix Environment, Addison−Wesly 1993
- Vahalia : Unix Internals, Prentice Hall 1996
- Salus : A Quarter Century Of Unix, Addison−Wesley 1994

10 Glossar

Begriff	***Erklärung***
Monolitischer Kernel	siehe Grundprinzipien, Grundprinzipien
Mikrokernel	siehe Grundprinzipien, Grundprinzipien
POSIX	Portable Operating System Interface. Eine Reihe von Bedingungen für portierbare Betriebssysteme
BSD	Berkely Software Distribution
DARPA	
VAX	
PDP	
Interdate	
AT&T	
SCO	Santa Cruz Operation
OSF	Open Software Foundation. Eine nicht kommerzielle Vereinigung, die sich die Entwicklung von Standards für Computer− und Netzwerksysteme auf ihre Fahnen geschrieben hat.
CACM	
MACH	Ein Betriebssystem, das den Einsatz von Multi−Prozessorsystemen und Multitasking ermöglicht.
Pipe	Verfahren, bei dem ein befehl Informationen zur weiteren Verarbeitung an einen anderen befehl weiterleitet.
OSF/1	Von der OSF entwickeltes Betriebssystem, das auf dem Mach−Kernel basiert.
NetBSD	Kostenlose Implementierung von UNIX, die auf BSD basiert.
XENIX	Unix−Version von Microsoft in Zusammenarbeit mit SCO
FreeBSD	INTEL−basierte, frei kopierbare Unix−Implementation