Kapitel 23

Betriebssysteme

23.1 Literatur

- **M. Bach:** The Design of the UNIX Operating System. Prentice Hall 1986
- **D. Comer:** Operating System Design: The XINU Approach. Prentice Hall 1984
- **P.A. Darnell, P.E. Margolis:** Software Engineering in C. Springer 1988
- **H. Herold:** UNIX-Shells. Addison-Wesley 1992
- **H. Herold, M. Mayer:** SCCS und RCS - Versionsverwaltung unter UNIX. Addison-Wesley 1995
- **B.W. Kernighan, R. Pike:** Der UNIX-Werkzeugkasten. Hanser 1987
- **M.J. Rochkind:** UNIX-Programmierung für Fortgeschrittene. Hanser 1988
- **A.-T. Schreiner:** System-Programmierung in UNIX - Teil1: Werkzeuge. Teubner 1984
- **W.R. Stevens:** UNIX Network Programming. Prentice Hall 1990
- **A. S. Tanenbaum:** Operating Systems - Design and Implementation. Prentice Hall 1987

23.2 Betriebssystem – Definition

DIN 44300 schreibt vor:

- „Zum Betriebssystem zählen die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.“

Mit einfachen Worten:

- Nach dem Einschalten des Rechners muß ein erstes Programm gestartet werden, das alle Ressourcen des Rechners verwaltet.
  Dieses Programm nennt man “Betriebssystem”.
- Das Betriebssystem ist ein Programm, das immer arbeitet, solange der Rechner in Betrieb ist.

- Die Hauptaufgabe des Betriebssystems besteht im Vermitteln zwischen dem Benutzer / Benutzerprogramm und der Hardware.
  Zu diesem Zweck stellt das Betriebssystem alle Leistungen des Rechners den Anwenderprogrammen zur Verfügung, schützt aber gleichzeitig auch Hardware und Software vor unberechtigtem Gebrauch.

23.3 Aufgaben eines Betriebssystems

- **Ressource Manager**
  Das Betriebssystem kontrolliert alle Hardware- und Software-Komponenten eines Rechners und teilt sie **effizient** den einzelnen Nachfragern zu.
  Das Betriebssystem stellt Basis-Dienstleistungen (Dateizugriff, Prozessmanagement) zur Verfügung, auf denen Anwendungsprogramme aufsetzen können.

- **Extended Machine** (*Virtual Machine Abstraction*)
  Das Betriebssystem besteht aus einer (oder mehreren) Schichten von Software, die über der „nackten“ Hardware liegen: *Virtual Machine*.
  Diese ist einfacher zu verstehen und zu programmieren. Komplexe Hardware-Details verbergen sich hinter einfachen und einheitlichen *extended instructions*.
  Der Programmierer erhält vom Betriebssystem eine angenehmere Schnittstelle zur Hardware als von der reinen Hardware *(instruction set, register structure, memory organisation, I/O structur, bus structure, . . .)*. Er muß sich nicht mehr um sämtliche maschinenabhängige Details kümmern.

23.4 Schichtenmodell

Layered System - Extended Machine:

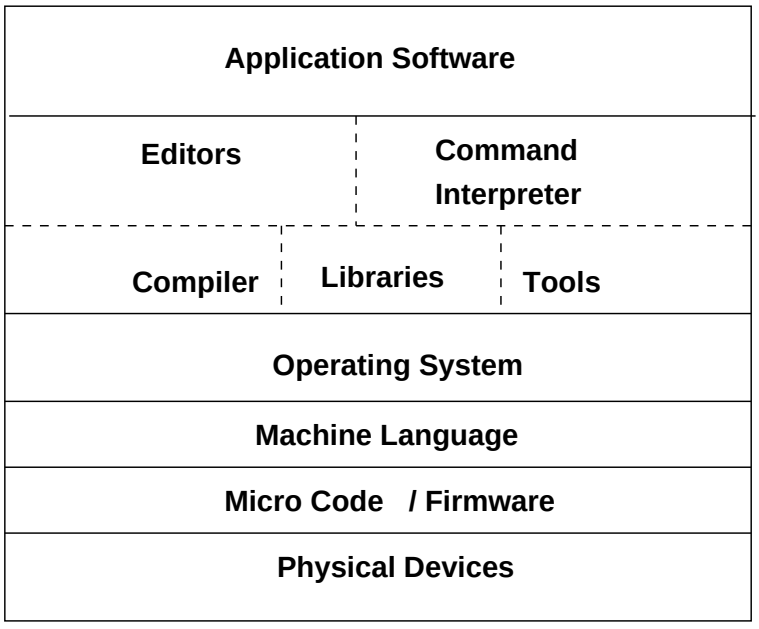

Abbildung 23.1: Layered System

- *Physical Devices*
  IC Chips, Drähte, Platten, Stromversorgung, ...
- *Microcode*
  „Primitive“ Software, die die *Physical Devices* direkt kontrolliert und sich teilweise direkt auf den Geräten befindet.
  Bietet der nächsten Schicht eine einheitlichere Schnittstelle zu den *Physical Devices*. Diese Schicht verbirgt bereits etwas von den Details der direkten Gerätesteuerung.
- *Machine Language*
  Schnittstelle zwischen Hardware und Software.
  50 - 300 Instruktionen (ADD, MOVE, JUMP, ...)
  Die meisten Instruktionen dienen dem Bewegen, Laden und Speichern von Daten im Hauptspeicher und in den Registern, dem Ausführen von arithmetischen Operationen, oder dem Vergleichen von Werten.
- *Operating System*
  Vermittelt zwischen Benutzer/Benutzerprogrammen und der Hardware.
  Verbirgt die Komplexität der Hardware vor dem Programmierer.
  Bietet angenehmere Instruktionen für's Programmieren:
  `READ FROM FILE` statt
  `Move head of disk 1 to track 231, lower head and ...`
- *System Software*
  Compiler, Editoren, Kommando Interpreter, Dienstprogramme — diese Programme gehören nicht zum Kern des Betriebssystems, werden aber meist vom Hersteller des Betriebssystem-Kernels mitgeliefert.
  Benutzer können diese Programme durch andere / eigene ersetzen, was bei den Komponenten des Kernels nicht möglich ist.
- *Application Software*
  Von Benutzern zur Lösung ihrer Probleme geschrieben.

23.5 Layered System — UNIX

Schalenmodell des UNIX Betriebssystems:

- Abstraktion
  Die Hardware versorgt den Kernel mit Basis-Dienstleistungen. Das Betriebssystem versorgt alle Programme mit System-Dienstleistungen. Es isoliert die Programme von Hardware-Eigenheiten. Der Programmierer kann mit abstrakteren Objekten arbeiten. Programme werden hardwareunabhängig und dadurch leichter portierbar.
- Interaktion
  Programme interagieren mit dem Betriebssystem-Kernel durch Aufrufe von genau definierten **System Calls**, um vom Kernel Dienstleistungen zu erhalten und um mit dem Kernel Daten auszutauschen.
- Kombination
  Die Benutzerprogramme und „Kommandos“ befinden sich in der gleichen Schicht. Sie können aber aufeinander aufbauen, sich gegenseitig aufrufen und / oder Daten miteinander austauschen.
  Beispiel: `cc` koordiniert `cpp` (Preprozessor), `comp` (übersetzen in Assembler), `as` (Assembler) und `ld`.

„Kernel — Dienstleistungen“:

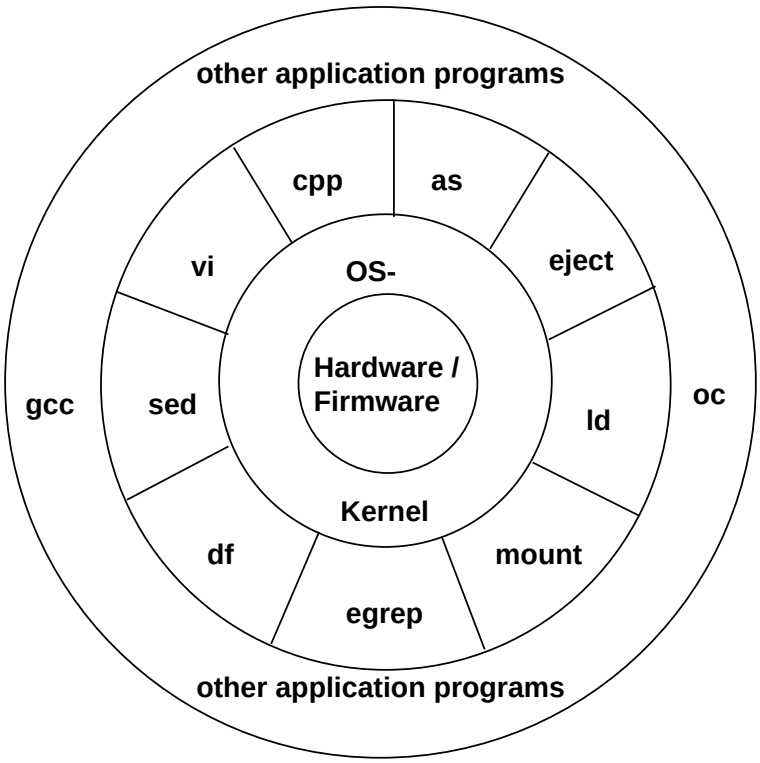

Abbildung 23.2: UNIX — Schalenmodell

- System Calls
  Alle Programme fordern Dienstleistungen des Kernels per System Calls an. Vom Programm aus erfolgt dies in Form eines Funktionsaufrufs.
  Die System Calls definieren die **Schnittstelle** zwischen Kernel und Benutzerprogramm.

- Subsysteme
  Die System Calls lassen sich nach ihrer Funktion den beiden großen Subsystemen im Kernel zuordnen:
  I/O Subsystem und **Process Subsystem**
  Das I/O Subsystem manipuliert die sogenannten statischen Objekte (Files), das Process Subsystem die dynamischen Objekte (Prozesse) des Rechners.

- Zugriffskontrolle
  Benutzerprogramme haben keinen direkten Zugriff auf Ressourcen im Rechner (CPU, Hauptspeicher, Platten, . . .). Die System Calls bieten den einzig möglichen, aber gut kontrollierbaren Zugang zu den Ressourcen.

- Transparenz
  Die Dienstleistungen des Kernels sind transparent.
  Beispiel: Datei, Gerät, Pipe = unformatierter Bytestrom

Kapitel 24

Aufbau des UNIX Betriebssystems

24.1 Struktur – Übersicht

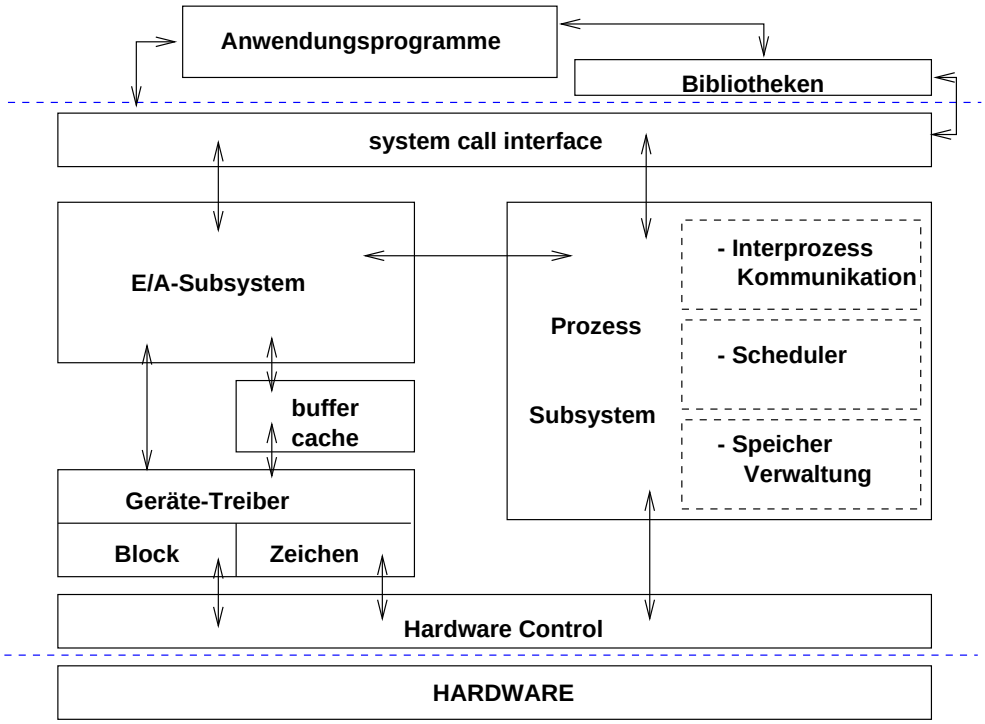

Abbildung 24.1: UNIX — Aufbau

Kapitel 25

Das I/O-Subsystem

25.1 Einführung

In einem UNIX-System wird jede Art von Information in Dateien (*Files*) gespeichert. Ein File ist schlicht ein Behälter für Information.

Aufgaben:

- Zugriff auf Geräte (== Files !!) ermöglichen
- Kontrolle der Zugriffsrechte auf Files
- Speichern und Wiederbeschaffen von Benutzerdaten
- Bereitstellen von Platten-Speicherplatz
- Verwalten von freiem Plattenplatz

Die Interaktion von Prozessen mit dem I/O Subsystem erfolgt über System Calls oder die Standard I/O-Bibliothek.

Beispiele:

chdir	Katalog wechseln
chmod	Zugriffsrechte ändern
chown	Besitzer ändern
close	Schließen einer Datei
creat	Erzeugen einer Datei („alt“)
dup	Filedeskriptor duplizieren
dup2	4.3BSD, ähnlich zu dup
fnctl	Eigenschaften einer offenen Datei ändern
ioctl	ähnlich zu fcntl, geräte-spezifisch
link	Neuer Name für bestehende Datei
lseek	Positionieren in einer Datei
mknod	*special files* erzeugen
mount	Filesystem in bestehendes „einhängen“
open	Öffnen/Erzeugen einer Datei
pipe	*unnamed pipe* erzeugen
read	Lesen aus einer Datei
stat	Attribute einer Datei lesen
umount	Filesystem „aushängen“
unlink	Namen entfernen
write	Schreiben in eine Datei

Das I/O Subsystem benutzt einen Puffermechanismus beim Zugriff auf Daten, die auf einem **block device** (z.B. Festplatte) lagern: **Buffer Cache**.
Block devices sind Geräte, die Daten in beliebig adressierbaren Blöcken speichern und übertragen können.

Char devices sind Geräte auf die ungepuffert zugegriffen wird, gemäß dem Modell „unformatierter sequentieller Bytestrom“. Beispiele: Terminal, Drucker.

Device Driver heißen die Kernel Module, die die Operationen der „physical devices“ im Detail kontrollieren. Sie haben eine *device independent* Schnittstelle nach oben zu den übrigen Kernel-Routinen und eine *device dependent* Schnittstelle nach unten zur Hardware hin.
Neuartig am UNIX **File System** ist die Idee, praktisch **alle** Objekte des Systems als Files zu repräsentieren. Trotz dieses gewaltigen Abstraktionsschrittes benötigt UNIX nur wenige verschiedene Dateiarten (**File Types**):

- **ordinary file** oder **regular file**:
  Container für jede Art von Daten (Text wie binär).
- **directory file**:
  Verzeichnis für die Namen von Files und zugehörige Verweise auf „Verwaltungsinformation“ (**Inode**).
- **named pipes** (**FIFO – First-In-First-Out**):
  zur unidirektionalen Prozeßkommunikation
- **device special files** (character special file / block special file):
  Geräte und Hardware allgemein, zu Files abstrahiert (Terminal, Drucker, Platten, Memory, . . .).
- **symbolic link:**
  Verweis auf den Pfadnamen einer anderen (u.U. nicht existierenden) Datei

- **socket**:
  Kommunikationsstruktur, insb. in Rechnernetzen, 4.3BSD
- und einige wenige mehr (System V, BSD)

25.2 Der Dateibegriff

- Definition: „Every file is a sequence of bytes.“

Zu einer Datei gehören

- ein oder auch mehrere Namen
- Inhalt und Aufbewahrungsort (Menge von Blöcken auf der Platte)
- Verwaltungsinformationen (Besitzer, erlaubter Zugriff, Zeitstempel, Länge, Dateityp, . . .)

UNIX verlangt / unterstellt bei regulären (gewöhnlichen) Dateien keinerlei Struktur und unterstützt auch keine. Die Konzepte „variabel oder konstant lange Records“ sind im Kernel von UNIX nicht implementiert. Ein File wird abstrakt als Datenstrom repräsentiert.

25.3 File System

Ein UNIX-Dateisystem besteht zumeist aus mehreren Dateisystemen (und Festplatten) - ein spezielles Dateisystem ist das **root file system**.

Layout:

Abbildung 25.1: Filesystem — Aufbau

- **boot block**
  enthält beim **root file system** den sog. **bootstrap code**, der beim „Hochfahren“ des Betriebssystems zuerst geladen wird — bei anderen Filesystemen ist er vorhanden, aber leer
- **super block**
  - Größe des Filesystems
  - Anzahl freier Blöcke
  - Liste der freien Blöcke
  - Zeiger auf nächsten freien Block in der Liste der freien Blöcke
  - Größe der Inode-Liste
  - Anzahl freier Inodes
  - Liste der freien Inodes
  - Zeiger auf nächste freie Inode

  - „Sperr-Felder“ für Liste der freien Blöcke / Inodes
  - Anzeigefeld, ob *super block* verändert wurde

- **Inode List**
  siehe unten — eine **Inode** ist die sog. **root-inode**, die den Wurzel-Katalog beschreibt und beim Zusammenfügen von Dateisystemen (**System Call mount**) wichtig ist.

25.4 Directory

Dieser Dateityp weist eine Struktur auf:

Lokaler Dateiname	Verweis auf Inode-Liste (Index)
.	4711
..	4709
onefile	47119
otherfile	471110
subdir_1	471130

Damit ergibt sich die bekannte Benutzer-Sicht: Hierarchisches Dateisystem oder Dateibaum! Die Blätter im Baum sind *ordinary files, named pipes* oder *device special files*, die anderen Knoten *directories*.

Anmerkung:
Man nennt deshalb Directories auch **Kataloge** oder **Verzeichnisse**.

25.5 Filesystem — Standard Directories

- **root directory**:
  Der Katalog am oberen Ende des Dateibaums / heißt Wurzel (**root**). Dieser Katalog ist direkter oder indirekter Vorgänger aller Files und Directories im gesamten Dateibaum und hat selbst keinen Vorgänger.
- **top level directories**:
  Im oberen Bereich des Dateibaums befinden sich Kataloge mit verschiedenen Systemdaten.
  - **/bin** — ausführbare Programme (*binaries*)
  - **/lib** — Bibliotheken und Hilfsdateien
  - **/tmp** — Platz für temporäre Dateien, benutzbar für alle User
  - **/etc** — Konfigurationsdateien und Systemverwaltungskommandos
  - **/proc** — *process information pseudo-filesystem*, benutzt als Schnittstelle zu den Kernel Datenstrukturen (mehr z.B. **man -S5 proc**)
  - **/dev** — Gerätedateien (*device special files*)
  - **/usr** — Userdaten und eher lokale Systemdateien
    (/usr/bin, /usr/lib, /usr/tmp /usr/man)
- **/usr/spool** — Zwischenablage für Druckaufträge, externe Post, usw. (*spooling area*)
- **/usr/local** — lokale Software, meist ebenfalls unterteilt in Kataloge mit den Namen *bin, lib, man* und *src*.

Dateinamen:

Dateinamen sind Verweise von einem Directory zu einem File. Sie bestehen aus Zahlen, Buchstaben und Sonderzeichen. Erlaubt sind im Prinzip alle Zeichen außer / (Slash) und

\0 (Null-Byte). Verschiedene Zeichen tragen jedoch eine Sonderbedeutung bei der Kommandoeingabe (Shell) und eignen sich daher nur bedingt für Dateinamen.
Dateinamen sind in älteren UNIX Versionen auf eine Länge von 14 Zeichen begrenzt, in neueren Versionen auf 256 (genauer: 255 + Null-Byte) Zeichen oder mehr.

25.6 Links

(siehe Kommando **ln**)

Hardlink:

```
ln name anothername
```

Syntax 25.1: Kommando ln – Hardlink

Die bisher über den Namen `name` erreichbaren Daten sind jetzt auch über den Namen `anothername` erreichbar. Beim Zugriff besteht zwischen `name` und `anothername` keinerlei Unterschied, da letztendlich auf den gleichen physikalischen Speicherplatz auf der Festplatte zugegriffen wird.
Hardlinks können nur auf *ordinary files* oder *device special files* eingerichtet werden. Beide Objekte, `name` und `anothername`, müssen sich im **gleichen** Filesystem befinden. Der Eintrag `name` muß bereits existieren.

Softlink / Symbolic Link:

```
ln -s name anothername
```

Syntax 25.2: Kommando ln – Symbolic Link

Das bisher über den Namen `name` erreichbare Daten-Objekt ist jetzt auch über den Namen `anothername` erreichbar. Falls das Daten-Objekt `name` existiert, so bestehen beim Zugriff auf diese Daten via `anothername` aus Sicht des Benutzer keinerlei Unterschiede zum Zugriff über `name`.
Im Kernel laufen jedoch zusätzliche Operationen ab (für `anothername` wird eine Inode angelegt!). Softlinks sind folglich „teurer“ als Hardlinks.
Der Eintrag `name` muß nicht unbedingt existieren. Softlinks können außerdem auch auf Directories eingerichtet werden und können auch auf andere Filesysteme (und physikalische Platten) verweisen.

Konsequenz:

- Durch Hardlinks wird aus dem Dateibaum ein zyklenfreier gerichteter Graph.
- Durch Softlinks können auch Zyklen im Filesystem entstehen!!!

25.7 Zugriffschutz

- Benutzergruppen
  Zur Identifikation eines Benutzers gehört der **Username** und der **Groupname**.
  Während der Username pro System eindeutig ist, fassen die Gruppen jeweils mehrere Benutzer (zum Beispiel alle Mitarbeiter eines bestimmten Projekts oder alle Mitarbeiter einer Abteilung) zu einer Einheit zusammen.

Ein Benutzer kann eine eigene Gruppe bilden oder zusammen mit anderen Benutzern in einer oder in mehreren verschiedenen Gruppen eingetragen sein. Die Gruppenzuordnung jedes Benutzers läßt sich dynamisch verändern. Die jeweils aktuelle Gruppenzugehörigkeit wirkt sich auf die Zugriffsrechte des Benutzers aus.

- Benutzerklassen
  Die Zugriffsrechte für alle Objekte des Filesystems sind nach drei Klassen gestaffelt und für jede Klasse individuell definierbar.
  - Klasse 1: **user**
    die Rechte des Besitzers des Objekts
  - Klasse 2: **group**
    die Rechte für die Mitglieder in dieser Gruppe.
  - Klasse 3: **other**
    die Rechte aller nicht durch die Klassen 1 und 2 erfaßten Benutzer.

  Wie die Benutzer des Systems tragen auch alle Files einen Username und einen Groupname. Der Username definiert den Besitzer. Der Groupname bestimmt die *group*-Klasse für den Zugriffsschutz.

- Individueller Schutz
  Der Schutzmechanismus in UNIX ermöglicht das individuelle Festlegen von Zugriffsrechten für jedes Objekt im Dateibaum. Der Besitzer kann bei jedem Objekt die Rechte für jede der drei Benutzerklassen separat festlegen: **Kommando chmod**

- **Zugriffsrechte**
  - **r** (read)
    Berechtigt zum Lesen einer Datei oder eines Katalogs.
  - **w** (write)
    Berechtigt zum Verändern einer Datei oder eines Katalogs. Bei Katalogen erlaubt dies primär das Hinzufügen von neuen Einträgen. Details zum Löschen von Einträgen s.u.
  - **x** (execute)
    Berechtigt zum Ausführen einer Datei oder zum Zugriff (**Kommando cd**) auf einen Katalog.

- Konsequenzen
  Dieser Mechanismus kann sowohl einzelne Dateien als auch ganze Teilbäume vor dem Zugriff einiger, vieler oder aller Benutzer schützen.
  Beispiel: **/etc/passwd**
  In früheren UNIX Versionen genügte bereits das Schreib-Recht für einen Katalog, um alle darin befindlichen Dateien löschen zu können. In neueren UNIX Versionen kann der Besitzer des Katalogs verlangen, dass zusätzlich mindestens eine der folgenden Bedingungen erfüllt ist, bevor ein Benutzer eine Datei aus dem Katalog löschen kann:
  - Benutzer ist Besitzer der Datei
  - Benutzer ist Besitzer des Katalogs
  - Benutzer hat Schreib-Recht auf die Datei
  - Benutzer ist Super-User

Will der Katalogbesitzer diese Sicherheitsmaßnahme aktivieren, muß er das Kommando **chmod +t** auf seinen Katalog anwenden.
Der **Superuser** ist ein besonders ausgezeichneter Benutzer (Username **root**, id=0), bei dem sämtliche Mechanismen des Zugriffsschutz **nicht** greifen.

VORSICHT bei Operationen als Super-User!

25.8 Inode

Die Datenstruktur **Inode** ist die interne Repräsentation einer Datei. Die **Inode Number** ist ein Index in die Inode-Liste und ist die **eindeutige** Identifikation einer Datei, **nicht** der Dateiname.

In der *Inode* enthalten sind sämtliche Informationen, die eine Datei beschreiben (siehe z.B. **Kommando ls -l**):

owner swg
group staff
type regular file
perms rw-r–r–
last access: Wed Jun 21 09:25:11 2000
last modification: Sun May 21 11:50:15 2000
last change: Wed Jun 21 05:24:20 2000
link count 2
size 600 Bytes
disk addresses

- *access* — wann wurde der Inhalt zuletzt gelesen?
- *modification* — wann wurde der Inhalt der Datei zuletzt verändert?
- *change* — wann wurde die Inode zuletzt verändert?

Blockadressen:
Herstellen der logischen Sicht (Byte-Folge) aus der „Menge“ der jeweiligen Plattenblöcke:

Durch die einfach, doppelt und dreifach indirekte Adressierung können - je nach gewählter Blockgröße - Dateigrößen von bis zu 100 GB erreicht werden.
Die eigentliche Beschränkung liegt allerdings in der Angabe der Dateigröße: mit 32 Bit läßt sich eine Angabe von max. 4 GB realisieren!
(Ab Solaris 8 wird die Dateigröße mit 64 Bit beschrieben und dadurch könnten die Dateigrößen theoretisch im Terrabytebereich liegen.)
Abhängig vom aktuellen Speicherplatz gibt es diese Datenstruktur in zwei verschiedenen Ausprägungen:

- als Vektorelement in der Inode-Liste eines Filesystems.
- Inode als Element der **Kernel Inode** (**in-core Inode**) Tabelle im Hauptspeicher.

Beide Ausprägungen sind funktional identisch, nur im Aufbau etwas unterschiedlich.
Auf diesen (öffentlichen) Teil der Inode kann man vom Programm aus zugreifen: (/usr/include/sys/stat.h)

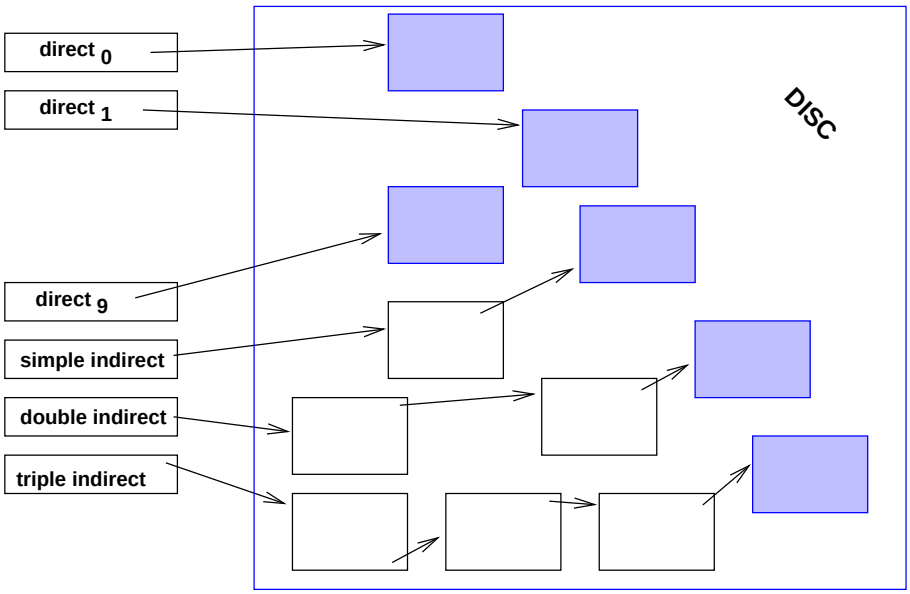

Abbildung 25.2: Blockadressen in der Inode

```
/* sys/stat.h: stat structure, used by stat(2) and fstat(2) */
struct stat {
  dev_t   st_dev;   /*device number*/
  ino_t   st_ino;   /*inode number*/
  mode_t  st_mode;  /*mode and type of file*/
  nlink_t st_nlink; /*number of links to file*/
  uid_t   st_uid;   /*user ID of the file's owner*/
  gid_t   st_gid;   /*group ID of the file's group*/
  dev_t   st_rdev;  /*ID of (raw)device if block/char special*/
  off_t   st_size;  /*file size in bytes*/
  time_t  st_atime; /*time of last access*/
  time_t  st_mtime; /*time of last data modification*/
  time_t  st_ctime; /*time of last file status change*/
};
```

(In der Variable `st_mode` sind zusätzlich auch die Zugriffsrechte der Datei aufgeführt. Siehe `man mknod`)

Zusätzliche Einträge der **In-core Inode:**

- Status
  - Inode gesperrt
  - Prozess wartet auf Freigabe
  - Inhalt ist anders als bei **disk inode**
  - Datei ist ein **mount point**
- Logische Geräte-Nummer
- Eigene **inode number**

- Referenz-Zähler - gibt die Zahl der aktiven Instanzen dieser Datei an (siehe *file tables*)

Auslesen der Inode-Informationen: stat() und **fstat()**

```
/* ftype.c: Datei-Typ und Zeitstempel ausgeben -
 * Namen aus Kommandozeile
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char *argv[]) {
   int i; char * p;
   struct stat statbuffer; /*aus stat.h -> spaeter mehr*/
   for(i=1; i<argc; i++) {
      printf("File >>%s<<\n", argv[i]);
         /* System Call stat */
      if (stat(argv[i], &statbuffer) < 0) {
         perror("stat error"); exit(1);
      }
      printf("\tlast access:       %s\n",
             ctime(&statbuffer.st_atime) );
      printf("\tlast modification: %s\n",
             ctime(&statbuffer.st_mtime) );
      printf("\tlast change:       %s\n",
             ctime(&statbuffer.st_ctime) );
      switch (statbuffer.st_mode & S_IFMT) {
         case S_IFDIR: p = "directory"; break;
         case S_IFCHR: p = "char special"; break;
         case S_IFBLK: p = "block special"; break;
         case S_IFREG: p = "regular"; break;
         case S_IFLNK: p = "sym link"; break ;
         case S_IFSOCK: p = "socket"; break;
         case S_IFIFO: p = "fifo"; break;
         default: p = "UNKNOWN"; break;
      }
      printf("\ttype:              %s\n", p);
   }
   exit(0);
}
```

liefert:

```
hypatia$ gcc -Wall -o ftype.exe ftype.c
hypatia$ ftype.exe ftype.c ftype.exe ..
File >>ftype.c<<
last access:       Sun May 21 11:45:33 2000

last modification: Sun May 21 11:45:05 2000

last change:       Sun May 21 11:45:05 2000

type:              regular
File >>ftype.exe<<
last access:       Sun May 21 11:45:46 2000

last modification: Sun May 21 11:45:33 2000

last change:       Sun May 21 11:45:33 2000

type:              regular
File >>..<<
last access:       Sun May 21 11:31:50 2000

last modification: Sat Dec 11 07:41:17 1999

last change:       Sat May 20 20:53:41 2000

type:              directory
hypatia$
```

Inode und **Links**:

Jedes File wird durch **genau eine** Inode repräsentiert, es kann aber **mehrere Dateinamen** besitzen. Ein **Dateiname** ist ein Verweis, genannt **Link**, aus einem Directory heraus auf einen Inode. Wieviele Directory-Einträge auf denselben Inode verweisen, spiegelt sich als **link count** im Inode-Feld *nlink* wieder.

```
hypatia$ pwd
/home/swg/soft/soft1/9/progs
hypatia$  ls -al
total 11
drwxr-xr-x   2 swg      users        1024 Dec 11 07:46 .
drwxr-xr-x   5 swg      users        1024 Dec 11 07:41 ..
-rw-r--r--   1 swg      users        1190 Dec 11 06:59 ftype.c
-rwxr-xr-x   1 swg      users        4921 Dec 11 06:59 ftype.exe
-rw-r--r--   1 swg      users         143 Dec 11 06:59 ftype.out
-rw-r--r--   1 swg      users         780 Dec 11 06:59 ftype.se
-rw-r--r--   1 swg      users           0 Dec 11 07:46 typescript
hypatia$ ls -al ..
total 6
drwxr-xr-x   5 swg      users        1024 Dec 11 07:41 .
drwxr-xr-x  21 swg      users        2048 Dec 11 07:45 ..
drwxr-xr-x   2 swg      users        1024 Dec 11 07:35 pics
```

```
drwxr-xr-x   2 swg      users        1024 Dec 11 07:46 progs
drwxr-xr-x   2 swg      users        1024 Dec 11 07:44 src
hypatia$
```

Links können aus verschiedenen Directories heraus auf ein und denselben Inode installiert sein. Die Directories müssen jedoch alle zum selben Filesystem gehören, da Directory-Einträge nur die `inum` enthalten (und keine Identifikation des Filesystems).

Eine Ausnahme dazu stellen die symbolischen Links dar (BSD 4.3 und System V ab Rel 4.0). Für einen symbolischen Link legt das I/O Subsystem eine zweite Inode an und speichert den Pfadnamen des Objekts ab, auf das der Link „zeigt“. Der *link count* im Inode des Objekts wird nicht erhöht. Das Objekt muß nicht einmal existieren, wenn der symbolische Link installiert wird.

25.9 System Calls für die I/O-Verbindungen

25.9.1 System Call open()

```
#include <fcntl.h>

int open(char *path,   /* path name */
         int oflag     /* option flag, e.g. r/w */
 );
int open(char *path,   /* path name */
         int oflag,    /* option flag */
         mode_t mode   /* if creat: permissions */
        );
/* returns file descriptor or -1 on error */
```

Syntax 25.3: System Call open()

```
#include <unistd.h>

int close ( /* close file */
int fd /* file descriptor */
  );
/* returns 0 on success or -1 on error */
```

Syntax 25.4: System Call close()

Der System Call **open()** mit **zwei** Argumenten stellt eine I/O-Verbindung her zwischen einem existierenden File und dem ausführenden Prozess.

open() kennt drei Arten von I/O-Verbindungen:

oflag	Makro	Prozess kann das File
0	O_RDONLY	nur lesen
1	O_WRONLY	nur schreiben
2	O_RDWR	lesen und schreiben

Für `oflag` enthält **fcntl.h** die Definitionen der symbolischen Konstanten.
open() schlägt fehl, wenn der Prozess die nötigen Rechte für `path` nicht besitzt oder `path` nicht existiert (nur in der ersten Version von *open()*).
Eine erfolgreiche Ausführung des System Calls liefert einen **Filedeskriptor** zurück. Dies ist eine kleine, positive Zahl, die bei allen nun folgenden I/O-Operationen (*read()*, *write()*, ..., *close()*) diese I/O-Verbindung identifiziert. Sie stellt einen Index in eine entsprechende Tabelle dar. Für **stdin** ist der Filedeskriptor 0, für **stdout** 1 und für **stderr** 2. Diese sind beim Start eines Prozesses typischerweise automatisch geöffnet!

Der System Call **close()** beendet eine I/O-Verbindung, die durch einen Filedeskriptor repräsentiert ist.

Da der Kernel die Zahl der I/O-Verbindungen pro Prozess limitiert ist es ratsam, nicht mehr gebrauchte I/O-Verbindungen mit *close()* zu schließen.

Terminiert ein Prozess, so führt der Kernel einen impliziten *close()* System Call für alle noch offenen I/O-Verbindungen aus.
Die allgemeinere Version von **open()** hat **drei Parameter**: der dritte ist nur relevant, wenn eine Datei geoffnet werden soll, die noch nicht existiert. In diesem Fall soll die Datei ggf. angelegt werden (über den zweiten Parameter festzulegen); dabei müssen dann die Zugriffsrechte auf diese neue Datei festgelegt werden. Sie ist voll kompatibel zu der älteren Form mit 2 Argumenten.

Der C-Compiler erlaubt variabel viele Argumente bei Funktionsaufrufen. Die gerufene Funktion muß wissen, wieviele Argumente gültige Werte enthalten. *open()* entscheidet anhand des 2. Arguments, ob das 3. Argument genutzt wird.

Die **oflags** können nicht nur die Werte **O_RDONLY**, **O_WRONLY** oder **O_RDWR** annehmen, sondern differenziertere Flagwerte. Diese entstehen durch `OR`-en von einem oder mehreren der folgenden Flagwerte zu dem r/w Flag:

oflag	Bedeutung
O_CREAT	Falls das File nicht existiert, wird es angelegt. Vgl. **creat()**. Nur bei diesem Flagwert findet das 3. Argument Anwendung.
O_TRUNC	Falls das File existiert, wird seine Länge auf 0 verkürzt, d.h., der alte Inhalt wird „gelöscht“.
O_APPEND	Dieses Flag stellt sicher, dass jede Write-Operation am Ende der Datei erfolgt, auch bei konkurrierenden Write-Operationen verschiedener Prozesse.
O_EXCL	Zusammen mit dem `O_CREAT`-Flag stellt dieses Flag sicher, dass `open()` fehlschlägt, wenn `path` bereits existiert.
O_NONBLOCK	Prozess blockiert nicht beim Öffnen eines noch nicht bereiten IPC-Kanals. Zusätzlich wirkt dieses Flag auf nachfolgende `read()` und `write()` System Calls.
O_NDELAY **O_SYNC**	Bei jedem `write()` auf dieses File blockiert der Prozess bis die Write-Operation physikalisch abgeschlossen ist.

Beliebige Kombinationen sind möglich, aber nicht alle sinnvoll!
Beispiele:

- **creat()**-Aufgabe: Datei zum Schreiben öffnen. Falls Datei noch nicht existiert, anlegen. Falls Datei bereits existiert, auf Länge 0 bringen (alten Inhalt löschen).

  ```
  open(path, O_WRONLY | O_CREAT | O_TRUNC, perms)
  ```

- „Harmloseres“ `creat()`: Datei zum Schreiben öffnen. Falls die Datei noch nicht existiert, anlegen. Auf keinen Fall Daten löschen.

  ```
  open(path, O_WRONLY | O_CREAT, perms)
  ```

- Datei zum Schreiben und Lesen öffnen. Falls Datei noch nicht existiert, anlegen. Mit `creat()` und `open()` (alte Form):

  ```
  if((fd = creat( path, perms )) < 0)
        sysfatal(path, "can't creat \"%s\"", path);
  if (close(fd) < 0  ||  (fd = open(path, O_RDWR)) < 0)
  ```

```
        sysfatal(path, "can't reopen \"%s\"", path);
```

Soll außerdem ein alter Inhalt erhalten bleiben, müssen einige weitere System Calls ausgeführt werden.

- Die neue Form von **open()** erlaubt dagegen eine differenzierte Anforderung der Kernel-Dienstleitstung in nur einem System Call:

```
if((fd = open(path, O_RDWR | O_CREAT, perms)) < 0)+
      sysfatal(path, "can't get r/w access to \"%s\"", path);
```

Bemerkung:
In der neuen Form umfaßt `open()` alle Funktionen der älteren Form von `open()` plus die Funktion des System Calls `creat()`. Durch die neuen Flagwerte ist aber eine wesentlich differenziertere Anwendung der einzelnen Funktionen möglich.

One may well ask why creat is still in UNIX as a system call. (. . .) The answer is probably emotional; creat has been a system call since the very beginning, and has evidently been granted tenure. [Marc J. Rochkind, Seite 26]

25.10 System Call write()

```
/* write to a file descriptor */

# include <unistd.h>

int write ( int fd,        /* file descriptor */
            char *buf,     /* buffer address  */
            int nbytes     /* number of bytes to write */
          );
/* returns number of bytes written or -1 on error */
```

Syntax 25.5: System Call write()

`write()` schreibt aus dem Hauptspeicher in eine I/O-Verbindung. `nbytes` bestimmt die Datenmenge, `buf` die Startadresse im Hauptspeicher und der File Deskriptor `fd` die I/O-Verbindung.

Eine `write`-Operation ist ein 1:1 Kopiervorgang. Es findet keinerlei Daten-Konvertierung oder Veränderung statt. Führt die I/O-Verbindung zu einem *regular file*, so bestimmt der *File Offset Pointer* im Open-File-Table-Slot (OFT – siehe weiter hinten) das Ziel des Kopiervorgangs. Die übertragenen Bytes überschreiben bereits vorher gespeicherte Daten. Anschließend zeigt der File Offset Pointer hinter das letzte übertragene Byte.

Achtung:
Die Modellannahme, dass die Daten nach Abschluß des `write()` System Calls physikalisch auf die Platte geschrieben wurden, ist nicht vollständig korrekt. Das I/O Subsystem kopiert die Daten in den Buffer Cache und gibt die Kontolle zurück, etwa mit folgender Meldung:

- *I've taken note of your request, and rest assured that your file deskriptor is OK, I've copied your data successfully, and there's enough disk space. Later, when it's convenient for me, and if I'm still alive, I'll put your data on the disk where it belongs. If I discover an error then I'll try to print something on the console, but I won't tell you about it (indeed, you may have terminated by then). If you, or any other process, tries to read this data before I've written it out, I'll give it to you from the buffer cache, so, if all goes well, you'll never be able to find out when and if I've completed your request. You may ask no further questions. Trust me. And thank me for the speedy reply.* [Marc J. Rochkind, Seite 29]

Mit dem **O_SYNC** Flag könnte man dies verhindern. Ein deutlicher Performanceverlust wäre jedoch die Folge.

25.11 Return-Wert von System Calls

Grundregel:

Jeder System Call zeigt mit seinem Return-Wert an, ob die Ausführung erfolgreich war oder ob eine Fehlersituation aufgetreten ist. Ein stabil codiertes Programm überprüft den Return-Wert bei jedem System Call.

Beispiel:

```
if( ( fd = creat(path, 0444) ) < 0 )
    perror( path),  exit(1);
```

Wenn der System Call **creat()** (Erzeugen einer Datei - alte Version, siehe *open()*) bei der Ausführung im Kernel Mode in irgendwelche Probleme läuft, liefert er **-1** als Ergebnis.

Frage:
Auf welches Problem traf *creat()*?
Woher weiß **perror()**, welches Problem aufgetreten ist?

errno:
Tritt bei der Ausführung eines System Calls ein Fehler auf, liefert der Aufruf einen eigentlich unmöglichen Wert zurück. Dies ist meist **-1** oder ein *NULL*-Zeiger. Die Details beschreibt die Definition der einzelnen System Calls im Kapitel 2 der *Reference Manuals*.

Neben diesem Fehlerindikator macht der Kernel in der globalen Variable *errno* eine Fehlernummer verfügbar. Mit Hilfe dieser Fehlernummer kann *perror()* oder eine selber geschriebene Funktion eine entsprechende Fehlermeldung produzieren.

```
void myperror(char * sys_call) {
  extern int errno;
  fprintf(stderr, "Error %d (%s)\n",
              errno, sys_call);
}
```

Für aussagekräftigere Fehlermeldungen kann man *errno* als Index in den globalen Vektor **sys_errlist[]** mit den Standard-Fehlermeldungen benutzen, siehe dazu auch **man perror**.
liefert:

```
hypatia$ gcc -Wall -o myerror myerror.c
hypatia$ ls -l xxx
-r--r--r--   1 swg      users           0 Dec 12 19:16 xxx
hypatia$ myerror
ERROR: Permission denied
hypatia$
```

Die Datei **/usr/include/errno.h** definiert symbolische Konstanten für alle vom Kernel benutzten Fehlernummern.

```
/* myerror.c */

# include <stdio.h>
# include <errno.h>
# include <fcntl.h>

/* int errno, char *sys_errlist[] und int sys_nerr
 * sind in stdio.h definiert
 */

void myperror2(char * sys_call) {

  if ( 0 <= errno  &&  errno < sys_nerr ){
    fprintf(stderr, "ERROR: ");
    fprintf(stderr, "%s\n", sys_errlist[ errno ]);
  } else
    fprintf(stderr, "Error %d occurred!\n", errno);
}

int main() {
   int fd;
   if( (fd = open("xxx", O_WRONLY)) < 0)
       myperror2("open"), exit(1);
   exit(0);
}
```

errno	Makro	error message
1	EPERM	No permission
2	ENOENT	No such file or directory
3	ESRCH	No such process
4	EINTR	interrupted system call
	…	
19	ENODEV	No such device
20	ENOTDIR	Not a directory
	…	
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Not a typewriter
	…	
64	ENONET	Machine is not on the network
65	ENOPKG	Package not installed
	…	

Anwendung von errno:
Wird ein “langsamer” System Call (z.B. **read()**) durch ein Interrupt-Signal unterbrochen, so liefert er -1 als Fehlerindikator zurück, setzt aber `errno` auf den Wert `EINTR`. Dies bedeutet, dass eigentlich **kein** Fehler aufgetreten ist, der System Call nur “gestört” worden war. Das Programm kann nun entscheiden, ob es den System Call erneut aufsetzt oder auf die Unterbrechung hin anders weiterarbeitet.

```
extern int errno;
   ...
do {
    errno = 0;  /* clear before system call */
    nread = read( fd, buf, BUFSIZ ); /* slow system call */
} while ( nread < 0  &&  errno == EINTR );

if ( nread < 0 )    /* there was a real error */
  perror( "read" ),   exit( 1 );
```

25.12 System Call lseek()

```
/* reposition read/write file offset */

# include <unistd.h>

long lseek( int fd,          /* file descriptor */
            long offset,     /* offset in file  */
            int whence       /* interpretation of offset */
          );
/* returns file offset pointer or -1 on error */
```

Syntax 25.6: System Call lseek()

`lseek()` verändert den *File Offset Pointer* OFT-Slot. Diesen neuen Startpunkt für die nächste `read`- oder `write`–Operation bestimmt `lseek()` aus `offset` und `whence`:

whence (int)	whence (Makro)	Neue Position
0	SEEK_SET	`offset` Bytes vom Dateianfang
1	SEEK_CUR	momentane Position plus `offset`
2	SEEK_END	Dateiende plus `offset`

`offset` kann auch eine negative Zahl sein. Die symbolischen Konstanten für `whence` sind in der Header-Datei **unistd.h** definiert.

`lseek()` kann nicht auf alle I/O-Verbindungen angewendet werden. Bei einem Terminal oder einer Pipe hat der *File Offset Pointer* keine sinnvolle Bedeutung. `lseek()` signalisiert deshalb einen Fehler mit dem Rückgabewert -1. Im Erfolgsfall gibt `lseek()` den neuen Wert des *File Offset Pointer* zurück.

Implizites lseek:

Enthalten die `oflags` im OFT-Slot einer I/O-Verbindung **O_APPEND**, so verursacht jeder `write` System Call vor dem Datentransfer eine implizite `lseek`-Operation, die den *File Offset Pointer* zuverlässig ans Ende der Datei positioniert.

Zwei explizite System Call Aufrufe, ein `lseek()`, gefolgt von einem `write()`, sind in einem multiuser / multitasking Betriebssystem wie UNIX nicht notwendig eine **atomare** Operation. Ein konkurrierender Prozess könnte seinen `write` System Call zwischen dem `lseek` und dem `write` System Call des ersten Prozesses ausführen. Dadurch wäre der **File Offset Pointer** des ersten Prozesses nicht mehr auf das neue Ende der Datei positioniert und der erste Prozess würde die Daten des zweiten Prozesses überschreiben. Das

`O_APPEND` Flag ermöglicht dagegen das sichere und geordnete Anfügen an eine Datei. Der implizite `lseek` und der `write` System Call laufen als eine atomare Operation ab.

25.13 System Call read()

```
/* read from a file descriptor */

# include <unistd.h>

int read( int fd,          /* file descriptor */
          char *buf,       /* buffer address  */
          int nbytes       /* number of bytes to read */
        );
/* returns number of bytes read, 0 on EOF, or -1 on error */
```

Syntax 25.7: System Call read()

`read()` liest `nbytes` von der I/O-Verbindung `fd` in den Puffer mit der Startadresse `buf` im Adressraum des Prozesses. Die Datenübertragung beginnt mit dem Byte, auf das der *File Offset Pointer* im OFT-Slot zeigt. Nach Abschluß des `read()` System Calls zeigt der *File Offset Pointer* auf das dem letzten übertragenen Byte folgende Byte.

Der Rückgabewert von `read()` — Anzahl der gelesenen Bytes — kann kleiner ausfallen als `nbytes`. Dies tritt auf, wenn über die I/O-Verbindung im Moment weniger als `nbytes` Daten zur Verfügung stehen.

Der Rückgabewert 0 zeigt **EOF** und -1 einen Fehler an. Im Fehlerfall gibt **errno** genauere Auskunft über die Fehlerursache.

Besonderheit: Non-Blocking Read

- Durch die Abstraktion aller Datenquellen/ziele zu Files kann mit `read` nicht nur von (Platten-) Dateien, sondern auch von Terminals, Pipes und Stream-Devices gelesen werden.

- Wenn über eine derartige I/O-Verbindung gerade keine Daten zur Verfügung stehen und ein Prozess führt darauf einen `read` System Call aus, dann blockiert der Prozess bis Daten eintreffen. Wurde jedoch beim `open()` oder durch `fcntl()` das **O_NDEALY** / **O_NONBLOCK** Flag für diese I/O-Verbindung gesetzt, so kehrt `read` ohne zu blockieren zurück. Bei Terminals und Pipes produziert `read` als Rückgabewert 0, was ein Entdecken von **EOF** unmöglich macht.

  Dieses Problem vermeidet `read` bei den neueren Stream-Devices, hier kommt der Wert -1 zurück und `errno` enthält den Wert `EAGAIN`.

25.14 System Calls link() und unlink()

25.14.1 Übersicht

Mit **unlink()** entfernt man Dateien wieder aus dem Directory, d.h. der Namenseintrag wird gelöscht, der Link Count der Datei wird dekrementiert:

```
/* unlink file */
# include <unistd.h>

int unlink ( char *path /* path name */ );
/* returns 0 on success or -1 on error */
```

Syntax 25.8: System Call unlink()

Besteht bereits ein Verweis aus einem Directory heraus auf eine Inode (kein Directory!), so kann man mit **link()** einen weiteren Verweis (**hardlink**) innerhalb desselben File Systems auf diese Inode installieren.

```
/* make a new name for a file */

# include <unistd.h>
int link (char *oldpath,     /* existing file */
          char *newpath      /* new directory entry */
         );
/* returns 0 on success or -1 on error */
```

Syntax 25.9: System Call link()

Der System Call `link()` inkrementiert den Link Count in der Inode um den Wert 1, `unlink()` dekrementiert ihn um 1.

Einen symbolischen Link installiert man mit folgendem System Call:

```
/* make a new name for a file */

# include <unistd.h>

int symlink ( char *oldpath,   /* existing file */
              char *newpath    /* new directory entry */
            );
/* returns 0 on success or -1 on error */
```

Syntax 25.10: System Call symlink()

Symbolische Links bestehen lediglich aus einem Pfadnamen. Sie können sowohl über File System Grenzen hinweg als auch auf Directories wie auch auf nicht existente Objekte installiert werden. Beim `open()` unterscheiden sie sich aus Benutzersicht durch nichts von Hardlinks, im Kernel aber sehr wohl.
Sämtliche Operationen unterliegen dem üblichen UNIX Zugriffsschutz. Nur mit den entsprechenden Rechten können Prozesse Directories lesen oder verändern.

25.14.2 link()

Der System Call `link()` installiert einen zusätzlichen Verweis auf eine bereits existierende Datei, genauer, auf eine Inode. Dazu trägt der System Call lediglich einen Verweis in das durch den Directory-Prefix von `newpath` bestimmte Directory ein. Der Verweis besteht aus der File-Komponente von `newpath` und der gleichen `inum` wie bei dem Eintrag `oldpath`. Außerdem erhöht `link()` den Link Count im Inode um 1.

Links können zwar aus verschiedenen Directories auf dieselbe Inode installiert werden, müssen aber innerhalb des selben File Systems bleiben. Auf Directories können keine Links installiert werden. Das mit `oldpath` bezeichnete Objekt muß bereits existieren.

Für den Eintrag braucht der Prozess (**euid** / **egid**egid) Write-Recht auf das Directory.

`link()` schlägt fehl, wenn das Objekt `newpath` bereits existiert.

25.14.3 unlink()

`unlink()` entfernt den Eintrag `path` aus dem entsprechenden Directory und verringert den Link Count in der zugehörigen Inode um 1. Wird der Link Count dadurch zu 0, kann das I/O Subsystem den durch die Datei belegten Plattenplatz und Inode freigeben.

Um einen Directory-Eintrag entfernen zu können und damit eine Datei zu löschen, braucht der Prozess (**euid** / **egid**) nur Write-Recht für das Directory. Die Rechte der Datei bleiben unberücksichtigt.

Dieses potentielle Sicherheitsloch wurde in neueren UNIX Versionen geschlossen. Setzt der Besitzer das **sticky bit** für sein Directory, so kann ein Prozess dort nur dann Einträge löschen, wenn zusätzlich zum Write-Recht auf das Directory noch eine der folgenden Bedingungen erfüllt ist:

- `euid` / `egid` des Prozesses hat Write-Recht auf die Datei
- `euid` des Prozesses ist Besitzer der Datei
- `euid` des Prozesses ist Besitzer des Directory
- `euid` des Prozesses ist 0 (Superuser).

Datei löschen? — Freigeben des Plattenplatzes bedeutet, dass das I/O Subsystem die der Datei zugeordneten Blöcke aus dem Inode entfernt und in die *Free List* einhängt.
Bei der nächsten Anforderung für neue Blöcke können die gerade freigegebenen Blöcke bereits wiederverwendet werden. Wann dies tatsächlich geschieht, hängt von den gerade aktiven Prozessen und vom *Free List* Algorithmus ab.

Besonderheit:
Das I/O Subsystem verzögert das Freigeben von Inode und Plattenplatz solange noch ein Prozess die Datei geöffnet hat. Da jedoch der Directory-Eintrag entfernt wurde, ist ein weiterer Zugriff auf die Datei, mit `open()` oder `stat()`, über den Namen nicht mehr möglich.
Die File Deskriptoren der Prozesse bilden die „letzte“ Verbindung zu der Datei. Sie können für `read()`, `write()`, `fstat()`, ..., und `close()` benutzt werden. Nach dem letzten expliziten oder impliziten `close()` System Call mit einem File Deskriptor, der noch auf die Datei verwiesen hat, gibt das I/O Subsystem die Daten endgültig frei.

Dieses Verhalten des Kernels vereinfacht den Gebrauch (und das ordentliche Entfernen) von **temporären Dateien**:

```
if((fd = open(TMPFILE, mode, perms )) < 0)
            sysfatal(TMPFILE, "can't creat temp file");
else
             unlink(TMPFILE);
```

Der Prozess kann die temporäre Datei benutzen, wenn er jedoch – egal aus welchem Grund – terminiert, gibt das I/O Subsystem automatisch allen Plattenplatz frei, den die Datei belegt hat. Es entstehen keine unliebsamen „Uberbleibsel".

25.15 System Call fcntl()

```
/* manipulate file descriptor - see manpages */

# include <unistd.h>
# include <fcntl.h>

int fcntl ( int fd,       /* file descriptor */
            int cmd,      /* command  */
            int arg       /* argument */
          );
/* returns value depending on cmd or -1 on error */
```

Syntax 25.11: System Call fcntl()

`fcntl()` verändert die Eigenschaften von bereits offenen I/O-Verbindungen. Er manipuliert die `oflag`-Bits im OFT-Slot. Damit können nachträglich Flagbits wie **O_APPEND** oder **O_SYNC** für eine I/O-Verbindung gesetzt oder gelöscht werden.

Beispiel:

```
#include <fcntl.h>

void clear_append(int fd)      /* clears O_APPEND flagbit */
{
   int oflags;

   if ( (oflags = fcntl( fd, F_GETFL, 0 )) < 0 )
      sysfatal( "fcntl()", "can't get oflags" );

   if ( (oflags & O_APPEND) != O_APPEND )
      return;      /* not set */

   oflags &= ~O_APPEND;   /* clear flagbit */

   if ( fcntl( fd, F_SETFL, oflags ) < 0 )
      sysfatal( "fcntl()", "can't set oflags" );
}
```

25.16 System Call ioctl()

```
/* control device */

# include <sys/ioctl.h>

int ioctl( int fd,       /* file descriptor */
           int cmd,      /* command  */
           int arg       /* argument */
         );
/* returns 0 on success or -1 on error */
```

Syntax 25.12: System Call ioctl()

`ioctl()` verändert die Eigenschaften von offenen I/O-Verbindungen zu Geräten oder IPC-Kanälen. Ein Prozess kann damit direkt auf einen *Device Driver* Einfluß nehmen. Deshalb sind auch die erlaubten Kommandos sehr abhängig von den einzelnen Gerätetreibern und davon, welche Treibermodule in den Kernel konfiguriert wurden (siehe dazu die entsprechenden Manpages!).

Beispiel: Echo abschalten

```
/* ---------- echo.c -----------*/
#include <unistd.h>
#include <termio.h>

/* switch off echo-ing characters typed in */
void no_echo( int fd ) {
   struct termio buf;
   if ( ioctl( fd, TCGETA, &buf ) < 0 ) {
      perror("ioctl()");
      /*can't get line parms (no_echo)*/
      exit(1);
   }
   if ( (buf.c_lflag & ECHO) != ECHO )
      return;      /* not set */
   buf.c_lflag &= ~ECHO;   /* clear bit */
   if ( ioctl( fd, TCSETA, &buf ) < 0 ) {
      perror("ioctl()");
      /*can't set line parms (no_echo)*/
      exit(2);
   }
}
```

```
/* switch on echo-ing characters typed in */
void set_echo( int fd ) {
   struct termio buf;
   if ( ioctl( fd, TCGETA, &buf ) < 0 ) {
      perror("ioctl()");
      /*can't get line parms (set_echo)*/
      exit(3);
   }
   if ( (buf.c_lflag & ECHO) == ECHO )
      return;      /* already set */
   buf.c_lflag |= ECHO;   /* set bit */
   if ( ioctl( fd, TCSETA, &buf ) < 0 ) {
      perror("ioctl()");
      /*can't set line parms (set_echo)*/
      exit(4);
   }
}
```

25.17 Synchronisation (mutual exclusion)

25.17.1 Generelles

Problem - Concurrency:

Mehrere Prozesse arbeiten lesend und schreibend auf einer Datei. Dabei kann es durchaus passieren, dass ein Prozess einen Datensatz liest, der während dessen von einem anderen Prozess in Teilen verändert wird („Mischmasch"). Mit einem einfachen Beispiel soll das Problem verdeutlicht werden.

Beispiel:
Mehrere Prozesse benötigen einen fortlaufenden Zähler, z.B. um für Datensätze einen internen Schlüssel zu vergeben; dazu lesen sie die Zahl für den nächsten Schlüssel aus einer Datei und schreiben ihn inkrementiert wieder zurück. In der folgenden Implementierung wird zwischen dem Lesen der Zahl und dem Zurückschreiben eine zufällig gewählte Zeit gewartet. Dieses Warten (Suspendieren des Prozesses kann mit dem **sleep()** realisiert werden; die Zufallszahl für die Anzahl der zu wartenden Sekunden kann mit dem Pseudo-Zufallsgenerator **random()**, der Startwert für die Folge der Pseudo-Zufallszahlen mit der Funktion **srandom()** jeweils aus der C-Bibliothek bestimmt werden:

```
/* ----------- incr_nolock.h -------------*/
#ifndef INCR_NOLOCK_H
#define INCR_NOLOCK_H
# include <stdio.h>
# include <fcntl.h>
# include <stdlib.h>
# include <unistd.h>
# define SEQFILE "seqno"      /*filename*/
# define MAXBUF   100         /* max length of seqno*/
int incr(int /*delay*/);
#endif
```

```
/* ----------- incr_nolock.c -------------*/
# include "incr-nolock.h"

int incr(int delay) {
   int fd, n, seqno; char buffer[MAXBUF + 1];
   if ( (fd = open(SEQFILE,O_RDWR) ) < 0 ) {
      perror("SEQFILE"); return -1;
   }
   /* rewind before read: */ lseek(fd, 0L, 0);
   if ( (n = read(fd, buffer, MAXBUF) ) <= 0) {
      perror("read"); return -1;
   }
   buffer[n] = '\0';
   if ( (n = sscanf(buffer, "%d\n", &seqno)) != 1) {
         fprintf(stderr, "sscanf error\n");
         return -1;
   }
   fprintf(stderr, "Read: %d\n", seqno);
   seqno++;
   sprintf(buffer, "%03d\n", seqno);
   n = strlen(buffer);
   /* wait delay sec: */ sleep(delay);
   /* rewind before write: */ lseek(fd,0L,0);
   if (write(fd, buffer, n) != n) {
      perror("write");
      return -1;
   }
   close(fd); return 0;
}
```

```
/* ----------- main.c -------------*/
# include "incr-nolock.h"

#define MAXDELAY 5
#define MAXREPEAT 5

int main(int argc, char ** argv) {
   int i = 1, seed = 1, delay;
   if( argc > 1) sscanf(argv[1], "%d", & seed);
   srandom(seed);
   sleep(delay = random() % MAXDELAY);
   while( (i <= MAXREPEAT) && (incr(delay) == 0)) {
         sleep(delay = random() % MAXDELAY); i++;
   }
   exit(0);
}
```

Ausführung:

```
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -c incr-nolock.c
gcc -Wall -o incr main.o incr-nolock.o
# incr ist das ausfuehrbare Programm
hypatia$ cat seqno
001
hypatia$ incr 2 2> s1 & incr 5 2> s2 &
hypatia$ cat seqno
007
hypatia$ cat s1
Read: 1
Read: 2
Read: 3
Read: 4
Read: 6
hypatia$ cat s2
Read: 1
Read: 2
Read: 3
Read: 5
Read: 6
hypatia$
```

Wie man hier leicht sieht, steht der gemeinsame Zähler nicht auf 11 (jeder der beiden Prozesse sollte um 5 weiterzählen)!

- Lösung: Semaphore

  Eine Semaphore ist eine "Ampel", die immer nur **einem** Prozess grün (Datei benutzen erlaubt) und allen anderen Prozessen "rot" (Benutzung z.Zt. nicht erlaubt) signalisiert.

  Alle Prozesse verpflichten sich, nur bei "grün" bestimmten Code auszuführen, z.B. auf eine gemeinsame Datei zuzugreifen. Dadurch, dass immer nur ein Prozess "grün" erhält (*mutual exclusion*), synchronisieren sich die Prozesse über die Semaphore.

- Codiertechnik

  Eine Semaphore bewacht die Anweisungen des kritischen Teils. Bevor ein Prozess den kritischen Teil betritt, muss er sich von der Semaphore Grün signalisieren lassen (warten, bis "grün" gesetzt ist). Bei Betreten des kritischen Bereichs wird die Semaphore auf "rot" gesetzt, wenn der Prozess den kritischen Teil verläßt, teilt er dies der Semaphore mit (wieder auf "grün" setzen). Anschließend kann die Semaphore einem anderen Prozess "grün" signalisieren. Der Notation des Informatiker **Dijkstra** folgend nennt man diese Operationen `P(sema)` (Semaphore/Strecke für sich reservieren – *protect*) und `V(sema)` (Semaphore/Strecke freigeben, holl. *vrej*).

In einem Programm könnte dies etwa folgendermaßen aussehen:

```
/* non-critical region */
...

P(semaphore);  /* get sema */
/* begin critical region */
...
/* end critical region */
V(semaphore);  /* release sema */

/* non-critical region */
...
```

Anmerkung:
Ist die Semaphore auf "rot", so wartet der Prozess (Operation `P`) solange, bis sie (von wem auch immer) auf "grün" gesetzt wird.

Problem: Deadlock

- zwei Prozesse arbeiten z.B. gleichzeitig auf zwei Dateien
- jeder hat eine Datei gesperrt und wartet darauf, dass der andere seine Datei freigibt, um weiterarbeiten zu können

25.17.2 Synchronisation mit link()

Der System Call **link()** installiert in einem Directory einen zusätzlichen Verweis auf eine Inode. Sollte bereits ein Eintrag mit diesem Namen existieren, schlägt `link()` fehl — unabhängig von der Userid.

Die `P()`-Operation besteht aus einem **open()**-Aufruf und aus, evtl. wiederholten, `link()`-Aufrufen.

Die `V()`-Operation besteht aus einem **unlink()**-Aufruf.

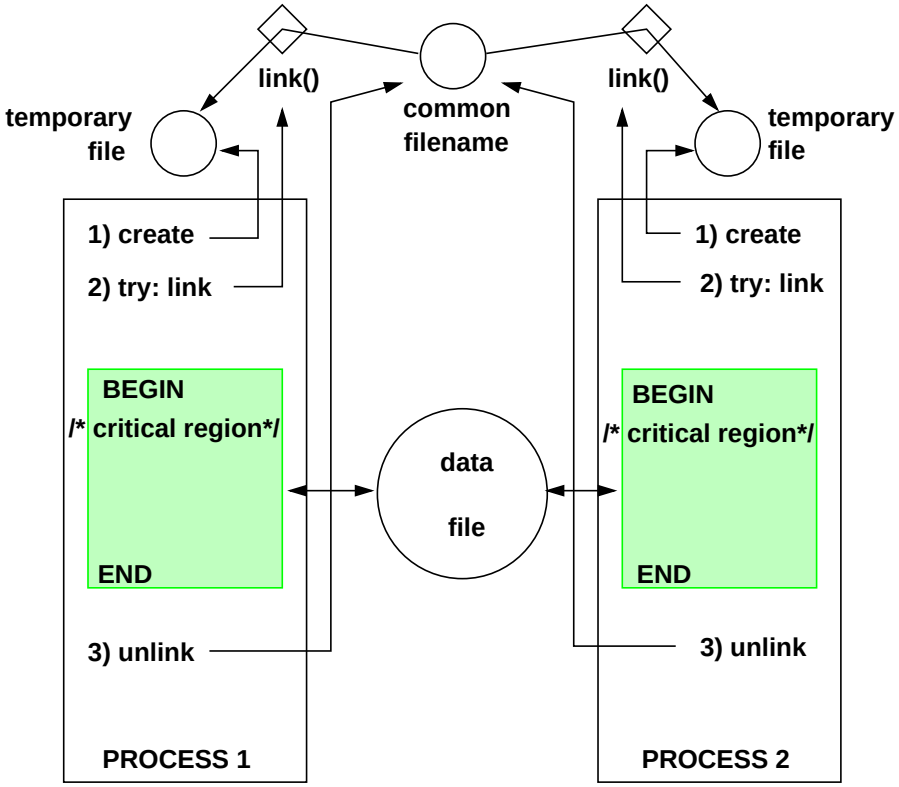

Abbildung 25.3: Synchronisation mit link()

Realisierung:

- Zur Erzeugung einer temporären Datei (Eindeutigkeit des Namens) wird die Bibliotheksfunktion **mkstemp()** aus **stdlib.h** verwendet.
- Das Programm besteht aus den Funktionen `my_lock()` und `my_ulock()` zur Sperrung des Dateizugriffs.
- Der Rest ist identisch mit vorigem Beispiel, ausser dass beim Zugriff auf den Zähler synchronisiert wird.

```
/* ------------- lock.h -----------*/
#ifndef MYLOCK_H
#define MYLOCK_H

#define MAXTRIES  3
#define NAPTIME   1
#define TMPFILE   "LCK_XXXXXX"

int my_lock( char * );
int my_ulock ( char * );
/* argument: common lockfile name */

#endif


/* ----------- incr.h -------------*/
#ifndef INCR_NOLOCK_H
#define INCR_NOLOCK_H
# define SEQFILE "seqno"   /*filename*/
# define MAXBUF  100       /* max length of seqno*/
int incr(int /*delay*/);
#endif
```

Ausführung:

```
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -c incr.c
gcc -Wall -c lock.c
gcc -Wall -o incr main.o incr.o lock.o
# incr ist das ausfuehrbare Programm
hypatia$ cat seqno
001
hypatia$ incr 2 2>s1 & incr 100 2>s2 &
hypatia$ cat seqno
011
hypatia$ cat s1
Entering critical region
Read: 1
Leaving critical region
Entering critical region
Read: 4
Leaving critical region
Entering critical region
Read: 7
Leaving critical region
Entering critical region
Read: 8
Leaving critical region
Entering critical region
Read: 10
Leaving critical region
hypatia$ cat s2
```

```
/* --------- lock.c --------- */

# include "lock.h"

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

char * my_tmpfile;
/* = { TMPFILE }; */

int my_lock( char *lockfile ) {
  extern int errno;
  int tmpfd, link_ret, tries = 0;

  my_tmpfile = (char *)calloc(256, 1);
  strcpy(my_tmpfile, TMPFILE);
  /* generate an unique tempfile name */
  if ( mkstemp( my_tmpfile ) == -1 ){
    perror("mkstemp"); exit(1);
  }
  /* create tmp file, then close it */
  if((tmpfd=open(my_tmpfile,O_WRONLY | O_CREAT,0664))<0){
     perror("TMPFILE open"); exit(1);
  }
  if ( close( tmpfd ) < 0 ) {
     perror("close"); exit(1);
  }
  /* now try to link tmp file to lock file */
  while((link_ret=link(my_tmpfile,lockfile))<0
        && errno==EEXIST ) {
    if ( ++tries > MAXTRIES )
       return -1;  /* couldn't get the lock */
    sleep( NAPTIME );
  }
  if ( link_ret < 0 )  exit(2);
  return 0;
}

int my_ulock(char * lockfile){
  if( (unlink( lockfile ) < 0)
      || ( unlink(my_tmpfile) < 0)) {
     perror("unlink"); exit(1);
  }
  return 0;
}
```

```
/* ----------- incr.c -------------*/
# include <stdio.h>
# include <fcntl.h>
# include <stdlib.h>
# include <unistd.h>
# include "incr.h"
# include "lock.h"

# define COMMON_NAME "XXX"
int incr(int delay) {
   int fd, n, seqno; char buffer[MAXBUF + 1];

   fprintf(stderr, "Entering critical region\n");
   if(my_lock(COMMON_NAME) != 0) {
       fprintf(stderr, "Couldn't lock!\n");
       exit(1);
   }

   if ( (fd = open(SEQFILE,O_RDWR) ) < 0 ) {
      perror("open SEQFILE"); exit(1);
   }
   /* rewind before read: */ lseek(fd, 0L, 0);
   if ( (n = read(fd, buffer, MAXBUF) ) <= 0) {
      perror("read"); exit(1);
   }
   buffer[n] = '\0';
   if ( (n = sscanf(buffer, "%d\n", &seqno)) != 1) {
         fprintf(stderr, "sscanf error\n");
         return -1;
   }
   fprintf(stderr, "Read: %d\n", seqno);
   seqno++;
   sprintf(buffer, "%03d\n", seqno);
   n = strlen(buffer);
   /* wait delay sec: */ sleep(delay);
   /* rewind before write: */ lseek(fd,0L,0);
   if (write(fd, buffer, n) != n) {
      perror("write"); exit(1);
   }
   close(fd);
   fprintf(stderr, "Leaving critical region\n");
   my_ulock(COMMON_NAME);
   return 0;
}
```

```
/* ----------- main.c ------------*/
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include "incr.h"

#define MAXDELAY 5
#define MAXREPEAT 5

int main(int argc, char ** argv) {
   int i = 1, seed = 1, delay;
   if( argc > 1) sscanf(argv[1], "%d", & seed);
   srandom(seed);
   sleep(delay = random() % MAXDELAY);
   while( (i <= MAXREPEAT) && (incr(delay) == 0)) {
         sleep(delay = random() % MAXDELAY); i++;
   }
   exit(0);
}
```

```
Entering critical region
Read: 2
Leaving critical region
Entering critical region
Read: 3
Leaving critical region
Entering critical region
Read: 5
Leaving critical region
Entering critical region
Read: 6
Leaving critical region
Entering critical region
Read: 9
Leaving critical region
hypatia$
```

25.17.3 Synchronisation mit lockf()

File Locking und *Record Locking* wurde erst relativ spät in UNIX eingeführt. Einige ältere System V Versionen erhielten einen System Call `lockf()`, andere, neuere (aktuelle) Versionen erweitern die Funktionalität von **fcntl()** und stellen eine Bibliotheksfunktion `lockf()` zur Verfügung.

Im einfachsten Fall kann `lockf()` eine Datei komplett vor simultanem Zugriff schützen und somit als Semaphore brauchbar machen. `lockf()` bietet aber zusätzlich die Möglichkeit, auf genau definierten Teilen einer Datei exklusiven Zugriff durchzusetzen.

Folgende Kommandos unterstützt `lockf()`:

```
/* file and record locking */

#include <sys/types.h>
#include <fcntl.h>

int lockf( int fd,       /* file descriptor */
           int cmd,      /* command  */
           int size      /* size of locked region */
         );
/* returns 0 on success or -1 on error */
```

Syntax 25.13: System Call lockf()

cmd	Bedeutung
F_LOCK	Eine Region der Datei für exklusiven Gebrauch nur durch den ausführenden Prozess reservieren.
F_ULOCK	Vorher reservierte Region der Datei wieder freigeben.
F_TEST	Eine Region der Datei auf Reservierungen durch andere Prozesse überprüfen.
F_TLOCK	Falls in der Region keine Reservierung durch einen anderen Prozess besteht, die Region für den ausführenden Prozess reservieren, sonst Rückkehr mit Fehleranzeige. Prozess blockiert nicht!

Der Bereich der Datei, auf den das Lock-Kommando angewendet werden soll, bestimmt sich aus dem Wert des *File Offset Pointer* im OFT-Slot und dem Wert des Argumentes `size`.

- Unteilbare Operation

Ein Aufruf mit `F_LOCK` blockiert solange, bis das Lock gesetzt werden kann. Um das Blockieren zu verhindern, könnte man eine Sequenz von `lockf()`-Aufrufen mit `F_TEST` und `F_LOCK` benutzen. Zwischen den beiden Aufrufen kann aber ein anderer Prozess erfolgreich `F_LOCK` anwenden, worauf der erste Prozess doch blockiert. Abhilfe schafft hier `F_TLOCK`, was `F_TEST` und `F_LOCK` zu einer atomaren, nicht blockierenden Operation zusammenfaßt.

- System Call oder Büchereifunktion `lockf()`

Der System Call `lockf()` bzw. die Büchereifunktion `lockf()` schlagen fehl, wenn auf die Datei gerade ein `LOCK` installiert ist — unabhängig von der Userid des ausführenden Prozesses.
Die `P()`-Operation besteht aus, evtl. wiederholten, `lockf()`-Aufrufen mit `F_TLOCK` als Kommando.
Die `V()`-Operation besteht aus einem `lockf()`-Aufruf mit `F_ULOCK` als Kommando.

- Realisierung:
  Im Vergleich zum vorigen Programm ändert sich im wesentlichen nur die Funktionen `my_lock()` und `my_ulock()`!

Kapitel 26

Datenstrukturen für I/O-Verbindungen

26.1 UFDT, OFT und KIT

Vom Prozess zum File via Kernel:

Der Kernel verwaltet in seinem Bereich des Hauptspeichers eine Reihe von Datenstrukturen, die die System Call Serviceroutinen beim Zugriff auf Files benutzen.

Die **UFDT**(User File Descriptor Table) identifiziert alle offenen I/O-Verbindungen eines Prozesses. Der Kernel legt diese Tabelle für jeden Prozeß aber im Kernel-Adreßraum an. Sie gehört zum Kontext des Prozesses. Jeder Filedeskriptor eines Prozesses ist Index in seine UFDT. Als einzige Komponente enthalten die Slots der UFDT einen Zeiger in die **OFT** (Open File Table) des Kernels.

Die *Open File Table* existiert nur einmal im Adressraum des Kernels. Alle Slots der *OFT* enthalten

- einen Zeiger in die Kernel Inode Tabelle,
- die **oflag**-Bits,
- einen **File Offset Pointer**
- und einen Referenz-Zähler.

Die *oflag*-Bits stammen aus dem **open()**-Aufruf. Der *File Offset Pointer* legt fest, bei welchem Byte die nächste **read()** oder **write()** Operation stattfindet. Der Referenz-Zähler zählt mit, wie viele *UFDT* Slots auf diesen *OFT* Slot zeigen.

Die *Kernel Inode Table* existiert nur einmal. Der Kernel koordiniert darüber sämtliche Zugriffe auf Files. Die *in-core* Version der Inodes enthält alle Komponenten der Plattenversion plus Informationen, die nur bei offenen I/O-Verbindungen von Bedeutung sind: ein Referenz-Zähler, die Device Nummer, ...
Jede offene Datei belegt genau einen Slot, unabhängig wie oft und von wievielen Prozessen sie geöffnet wurde.

26.2 File Zugriff

File Deskriptoren:

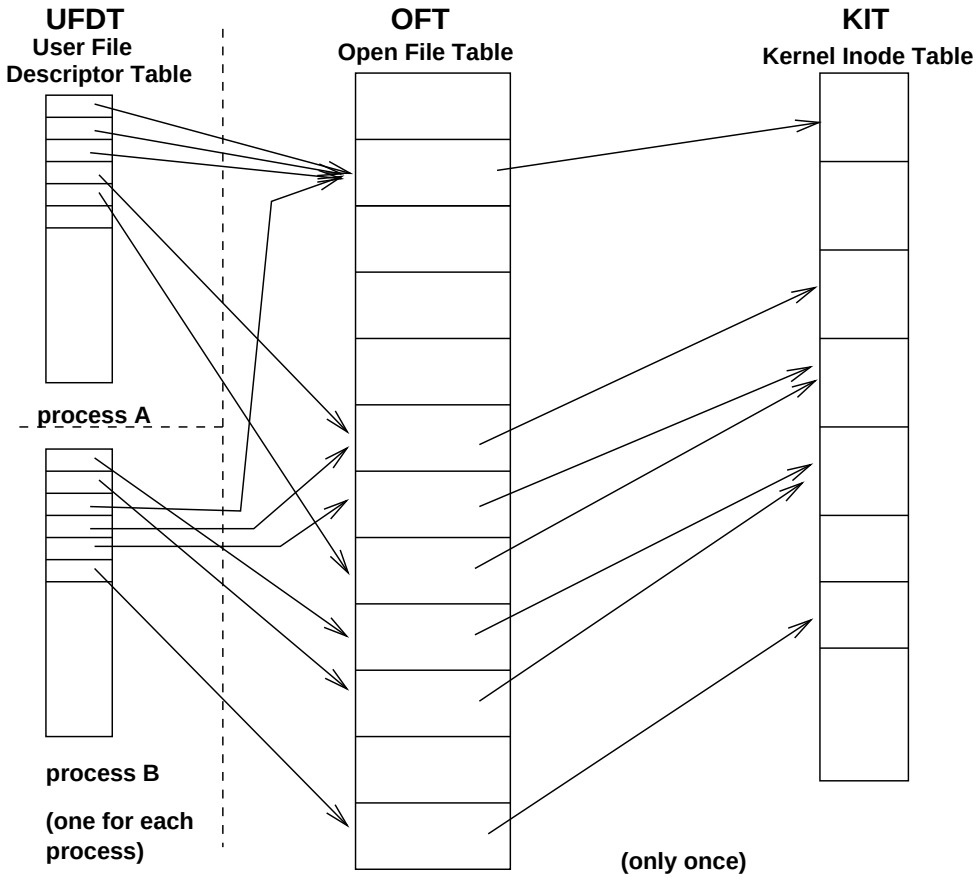

Abbildung 26.1: File Tabellen

- eine kleine Integer-Zahl als Index in die *UFDT*
  (Maximum: **NOFILE** aus **/usr/include/sys/param.h**
- **0** (**stdin**), **1** (**stdout**), **2** (**stderr**) — Konvention

Prozesse haben vier ID's:

- real user ID
- real group ID
- effective user ID
- effective group ID

Zugriff eines Prozesses auf Datei:

1. Ist die effektive User-Id 0 (superuser), so ist Zugriff erlaubt.
2. Stimmt die effektive User-ID des Prozesses mit dem Besitzer der Datei überein und sind die Zugriffsrechte entsprechend dem Zugriff (read, write) richtig gesetzt, so ist Zugriff möglich.
3. Stimmt die effektive User-ID des Prozesses nicht mit dem Besitzer der Datei überein
   und
   stimmt die effektive Group-ID des Prozesses mit der Group-Id des Besitzers der Datei überein
   und
   und stimmen Zugriffswunsch und Zugriffsrechte überein,
   so ist der Zugriff erlaubt.
4. Stimmt die effektive User-ID des Prozesses nicht mit dem Besitzer der Datei überein
   und
   stimmt die effektive Group-ID des Prozesses nicht mit der Group-Id des Besitzers der Datei überein
   und
   und stimmen Zugriffswunsch und Zugriffsrechte für *others* überein,
   so ist der Zugriff erlaubt.

File Access Mode Word zur Darstellung obiger Dateiattribute:

Oktalwert	Bedeutung
04000	set user ID on execution
02000	set group ID on execution
01000	save text image after execution (*sticky bit*)
00400	Read by user
00200	Write by user
00100	Execute by user
00040	Read by group
00020	Write by group
00010	Execute by group
00004	Read by other
00002	Write by other
00001	Execute by other

Herausfinden

- der User-Rechte: Maske 0700 (Dual: 000 . . . 000111000000)
- der Group-Rechte: Maske 070 (Dual: 000 . . . 000111000)
- der Other-Rechte: Maske 07 (Dual: 000 . . . 000111)

sticky bit:
ist das Programm ausführbar als *read-only*, so verbleibt eine Kopie des Textes im Swap-Bereich (nach Beendigung des Programms) und kann beim nächsten Start schneller geladen werden.
File Mode Creation Mask:
Jeder Prozeß hat eine Maske zum Anlegen der Rechte neuer Dateien, die mit dem **umask()** System Call gesetzt wird:

```
NAME
       umask - set file creation mask

SYNOPSIS
       #include <sys/stat.h>

       int umask(int mask);

DESCRIPTION
       umask sets the umask to mask & 0777.

       The umask is used by open(2) to set initial file permissions
       on a newly-created file.  Specifically,  permissions
       in  the umask are turned off from 0666 (so, for example,
       the common umask default value of 022 results in new files
       being  created  with  permissions
       0666 & ~022 = 0755 = rw-r--r--).

RETURN VALUE
       This system call always succeeds and the previous value of
       the mask is returned.

CONFORMING TO
       SVr4, SVID, POSIX, X/OPEN, BSD 4.3

SEE ALSO
       creat(2), open(2)
```

26.2.1 Manipulation der I/O-Datenstrukturen

- Manipulation nur per System Call:

Um die Integrität der I/O-Datenstrukturen zu gewährleisten, behält der Kernel sämtliche Tabellen in seinem Adreßraum. Prozesse können deshalb nur durch System Calls auf diese Tabellen einwirken.

UFDT	User File Descriptor Table
OFT	Open File Table
KIT	Kernel Inode Table

- **open()** belegt immer je einen Slot in der UFDT und der OFT. Der Referenz-Zähler im OFT-Slot wird mit 1 initialisiert. Bestand keine Verbindung zu der Datei, belegt *open()* auch einen Slot in der KIT und initialisiert diesen neuen in-core Inode. Besteht bereits eine Verbindung zu der Datei (vom selbem oder einem anderen Prozess), wird der schon für die Datei reservierte KIT-Slot verwendet. Der Kernel installiert nur den Zeiger vom OFT-Slot auf den KIT-Slot und inkrementiert den Referenz-Zähler in diesem in-core Inode.

  Folge: mehrere OFT-Slots können auf den selben KIT- Slot zeigen. Der Kernel hält von jedem Platten-Inode maximal eine Kopie *in-core*.

- **close()** gibt einen Slot in der UFDT frei und dekrementiert den Referenz-Zähler im OFT-Slot. Wird dieser Referenz-Zähler dadurch zu 0, kann auch der OFT-Slot freigegeben und der Referenz-Zähler im KIT-Slot dekrementiert werden. Wird nun dieser Referenz-Zähler zu 0, gibt der Kernel auch den KIT-Slot frei. Sollte im Inode der **Link Count** 0 sein, kann der Kernel jetzt auch den Platten-Inode und Plattenplatz freigeben.

- Der System Call **dup()** (siehe auch **dup2()**) belegt den niedersten, noch freien Slot in der UFDT, kopiert den Inhalt eines bereits belegten Slots dorthin und inkrementiert den Referenz-Zähler in dem OFT-Slot, auf den nun beide UFDT-Slots zeigen.

  Folge: mehrere UFDT-Slots eines Prozesses können auf den selben OFT-Slot zeigen.

- Der System Call **fork()** erzeugt einen neuen Prozess, indem er den kompletten Kontext des ausführenden Prozesses verdoppelt. Dabei muß der Kernel u.a. sowohl die UFDT kopieren als auch die Referenz-Zähler in den entsprechenden OFT-Slots inkrementieren.

  Folge: mehrere UFDT-Slots aus verschiedenen aber verwandten Prozessen können auf den selben OFT-Slot zeigen (Vererbung). Diese Prozesse können sich dadurch einen File Offset Pointer teilen.

Im folgenden soll die Belegung der **Kerneltabellen** über eine Folge von System Calls gezeigt werden; die Stdio-Verbindungen werden dabei nicht dargestellt — Ausgangspunkt:

- Parent Process
  - `fd1 = open("file1", O_RDONLY );`
  - UFDT-Slot belegen, OFT-Slot belegen, Refz = 1; KIT-Slot belegen, Refz = 1;
  - `fd2 = open( "file2", O_RDWR );`
  - `fork();`
- Kernel:
  - UFDT Vektor kopieren, in allen betroffenen OFT Slots: `++Refz;` die KIT Slots bleiben unverändert
- Child Process
  - `fd3 = open("file3", O_RDONLY);`
- Parent Process
  - `fd3 = open("file3", O_WRONLY);`

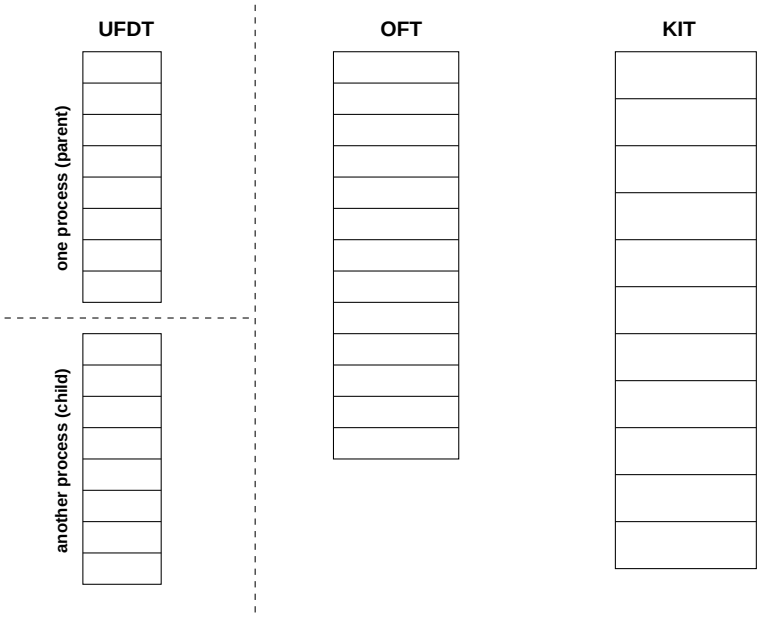

Abbildung 26.2: Kerneltabellen - Start

– UFDT Slot belegen, OFT Slot belegen, Refz = 1; bereits belegten KIT Slot benutzen, `++Refz;`

Belegung danach:

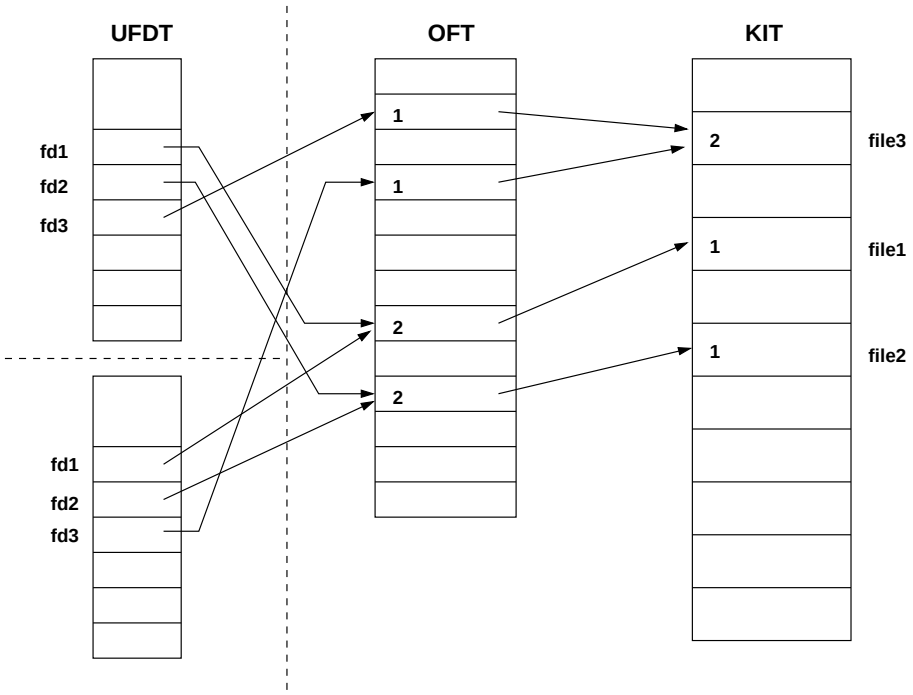

Abbildung 26.3: Kerneltabellen - Teil 1

- Parent Prozess
  - `fd4 = open("file4", O_RDONLY);`
- Child Process
  - `fd4 = open("file5", O_RDONLY);`
  - `close(fd1);`
  - UFDT Slot freigeben, im OFT Slot `--Refz; Refz > 0` ==> KIT Slot bleibt unverändert
  - `fd5 = dup(fd4);`
  - freien UFDT Slot belegen, im bereits belegten OFT Slot `++Refz;` KIT Slot bleibt unverändert
- Parent Process
  - `close(fd2);`
  - UFDT Slot freigeben, im OFT Slot `--Refz; Refz > 0` ==> KIT Slot bleibt unverändert

Belegung danach:

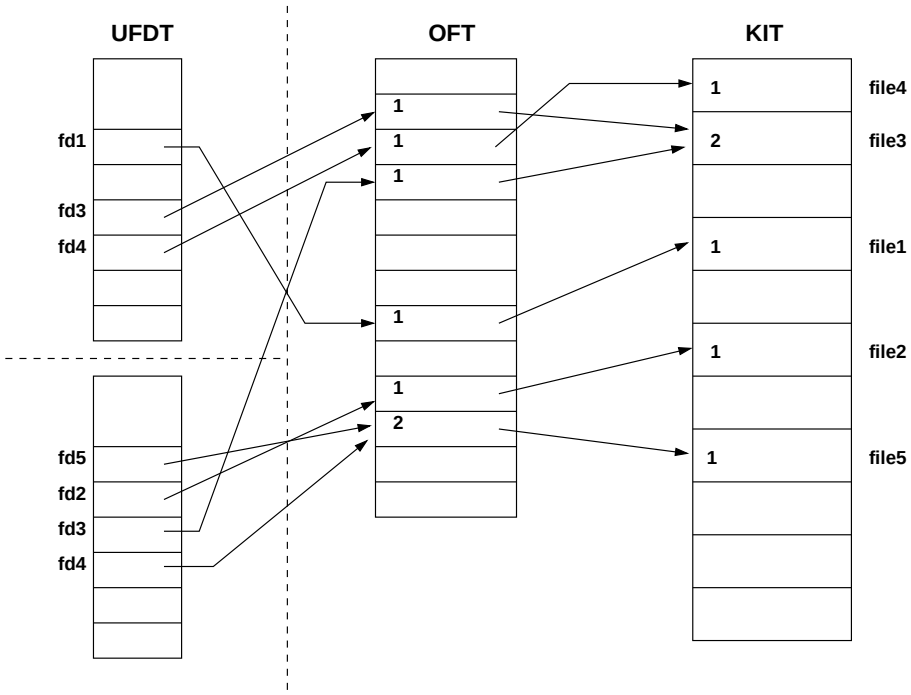

Abbildung 26.4: Kerneltabellen - Teil 2

- Child Process:
  - `close(fd4);`
  - UFDT Slot freigeben, im OFT Slot `--Refz; Refz > 0 ==>` KIT Slot bleibt unverändert
  - `close(fd5);`
  - UFDT Slot freigeben, im OFT Slot `--Refz; Refz == 0 ==>` OFT Slot freigeben und im KIT Slot `--Refz; Refz == 0 ==>` auch KIT Slot freigeben (Link Count?)
- Parent:
  - `close(fd1);`
  - UFDT Slot freigeben, ist im OFT Slot `--Refz == 0 ==>` freigeben, ist im KIT Slot `--Refz == 0 ==>` freigeben
  - `close(fd4);`

Belegung danach:

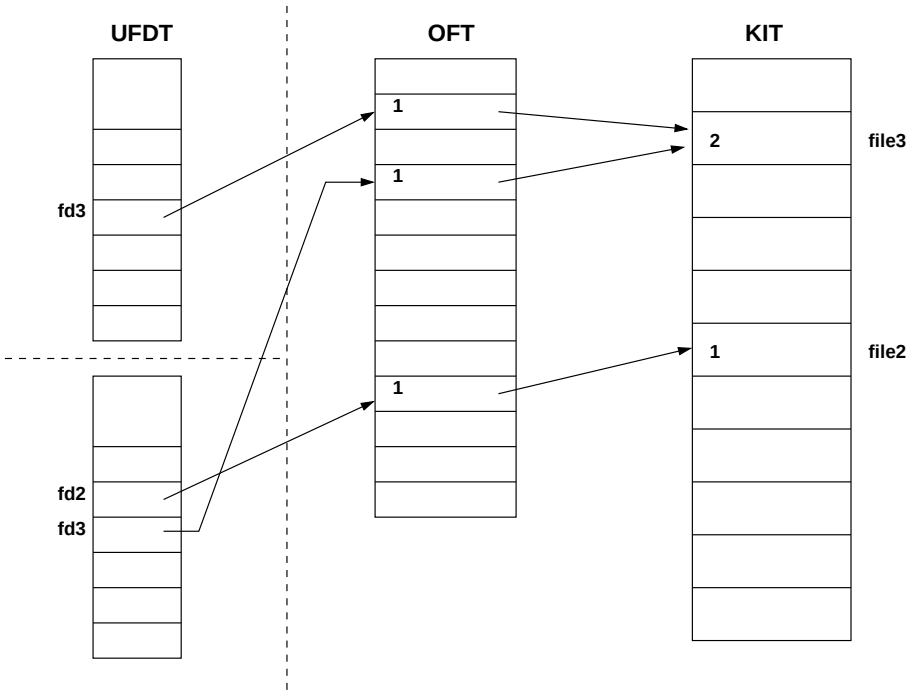

Abbildung 26.5: Kerneltabellen - Teil 3

- Child Prozess:
  - `close(fd2);`
  - UFDT Slot freigeben, OFT Slot `--Refz == 0 ==>` freigeben, KIT Slot `--Refz == 0 ==>` freigeben
- Parent Process:
  - `close(fd3);`
  - UFDT Slot freigeben, im OFT Slot `--Refz; Refz == 0 ==>` OFT Slot freigeben und im KIT Slot `--Refz; Refz > 0 ==>` KIT Slot noch nicht freigeben
- Child Process:
  - `close(fd3);`
  - UFDT Slot freigeben, OFT Slot `--Refz == 0 ==>` freigeben, KIT Slot `--Refz == 0 ==>` freigeben

Kapitel 27

Anhang

27.1 Das Process Subsystem

Aufgaben:

- Prozesse managen (kreieren, terminieren, synchronisieren)
- Interprocess Communication (**IPC**) ermöglichen
- Memory verwalten

Die Ausführung eines Programms heißt **Prozess**. Das Process Subsystem benötigt Service vom I/O Subsystem, wenn ein Programm gestartet (Image von Platte in Speicher) oder wenn Prozessdaten aus- / eingelagert werden müssen.

Benutzerprogramme/Prozesse erhalten vom Process Subsystem Serviceleistungen über System Calls wie:

brk	Datenadreßraum ändern
exec(*)	Text eines Prozesses mit neuem Text überlagern
exit	Prozess terminieren
fork	Erzeugen eines neuen Prozesses
kill	Signal senden
setuid	User Id ändern
signal	Mit Signalen umgehen
wait	Auf Termination eines erzeugten Prozesses warten

Der **Memory Management** Teil kümmert sich um die Zuteilung des Hauptspeichers an die einzelnen Prozesse. Er übernimmt das Ein- und Auslagern von Prozessdaten vom Primärspeicher zum Sekundärspeicher und umgekehrt:

- *swapping*: alle Prozessdaten werden verlagert
- *demand paging*: nur Teile des Adressraums, sog. *pages* werden vorgehalten

Der **Scheduler** teilt die CPU den einzelnen Prozessen zu. Er entscheidet anhand von Prioritäten und der Uhr (*timeslice*-Verfahren), welcher Prozess auf der CPU arbeiten darf.

Der **IPC** (Interprocess Communication) Teil versorgt alle Prozesse mit Dienstleistungen zur Kommunikation untereinander:

- **synchroner** Datenaustausch: Pipes, Message Queues, shared memory, sockets
- **asynchroner**Datenaustausch: Signale

27.2 System Calls – Execution Mode

- Execution Mode
  Die Hardware führt zu jedem Zeitpunkt entweder Anweisungen eines Benutzerprogramms oder Anweisungen im Kernel-Code aus.

  Bei der Ausführung von Anweisungen und beim Zugriff auf den Speicher unterscheidet die Hardware zwischen zwei Zuständen, in denen sie sich jeweils befinden kann:

  User Mode
  Darin befinden sich Prozesse beim Ausführen der meisten ihrer Anweisungen zur Lösung ihres anwendungsspezifischen Problems.

  Kernel Mode
  Darin befinden sich Prozesse dann, wenn sie Dienstleistungen des Kernels benötigen.

- **Instruction Set**
  Die *Machine Language* legt die Schnittstelle zwischen Hardware und Software fest. Der Übergang ist definiert durch das *Instruction Set*, die Menge aller Anweisungen, die die Hardware ausführen kann.
  Die Hardware unterteilt diese *Machine Language Instructions* in **privilegierte** (z.B. Manipulation von Prozessor Status Register) und **nicht-privilegierte** Anweisungen und stellt den Schutz dieser Anweisungen sicher:
  Im **User Mode** können nur nicht-privilegierte Anweisungen, im **Kernel Mode** können auch privilegierte Anweisungen ausgeführt werden.
  Damit können Prozesse im *User Mode* durch die Hardware von unkontrollierten Manipulationen an wichtigen Datenstrukturen und Registern des Kernels abgehalten werden.

- **Address Space**
  Für den Zugriff auf Speicheradressen stellt die Hardware ebenfalls einen zweistufigen Schutzmechanismus zur Verfügung. Jeder Bereich des Speichers, meist auf Seiten-Basis, gehört entweder zum **User Adressraum** oder **Kernel Adressraum**.
  Die Memory Management Hardware sorgt dafür, dass im *User Mode* nur auf Speicheradressen des *User Adressraums* und ausschließlich im *Kernel Mode* auf beide Bereiche zugegriffen werden kann.

- **Hardware Protection**
  Abstraktionen wie Prozess oder Benutzer sind auf Hardware-Ebene unbekannt. Die Hardware kann nur Teile des Adressraums und bestimmte Anweisungen vor unberechtigtem Zugriff schützen.

27.3 System Calls - Kernel Services

System Calls sind die einzige Möglichkeit, die ein Programm hat, um vom Kernel Dienstleistungen zu erhalten.

Modell:
Benutzerprogramm und Betriebssystem (Kernel) sind **nicht** zwei separate Einheiten, die

von einander unabhängig laufen. Sie operieren stattdessen als ein grosses logisches Programm mit vielen Unterprogrammen.
Kernel und Programm grenzen sich voneinander ab wie zwei Module eines gut strukturierten Programms. Beide Module besitzen genau definierte Einstiegspunkte für den gegenseitigen Aufruf.

Der Kernel ist speicherresidenter Code, Programme werden in den Speicher geladen und entfernt. Alle Programme nutzen gemeinsam denselben Code im Kernel Adressraum.

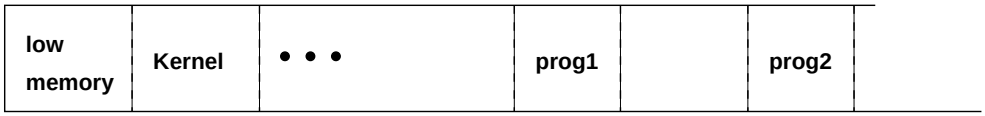

Abbildung 27.1: Adressraum

Aufruf- und Ablaufsequenz:

- Kernel arbeitet (*Kernel Mode*)
  Anweisungen im Kernel Adressraum werden ausgeführt.
- Kernel startet `prog1` (*Kernel Mode*) — Kernel richtet Umgebung (**Prozess Kontext**) für `prog1` im Speicher ein, schaltet in **User Mode** und weist die CPU an (Unterprogrammaufruf), die Anweisungen ab der Startadresse von `prog1` auszuführen.
- `prog1` arbeitet (*User Mode*)
  Dabei kann nur auf Anweisungen und Daten im Adressraum von `prog1` zugegriffen werden.
- `prog1` fordert Kernel Service an (noch *User Mode*) — Parameter hinterlegen — System Call ausführen (Unterprogrammaufruf) — Spezielle **trap**-Anweisung ausführen Übergang in *Kernel Mode*.
  Im *low memory* (Kernel Adressraum) ist festgelegt, wo nach einer *trap* die Ausführung weitergehen soll.
- Kernel arbeitet (*Kernel Mode*)
  Entsprechend dem aufgerufenen System Call wird eine Serviceroutine im Kernel Adressraum ausgewählt, die den geforderten Kernel Service liefert.
  Die Serviceroutine hat sowohl Zugriff auf Adressen des Kernel Adressraums als auch auf Adressen des User Adressraums.
  Als letzte Aktion veranlasst die Serviceroutine das Zurückschalten in den *User Mode* und den Rücksprung zur Stelle des System Call Aufrufs in `prog1`
- `prog1` arbeitet (*User Mode*)
- ...
- `prog1` terminiert (*User Mode*)
  Implizite oder explizite Ausführung des System Calls **exit()**.
  Rückkehr zum *Kernel Mode* und *Kernel* Adressraum.
- Kernel arbeitet (*Kernel Mode*)
  Kernel gibt alle von `prog1` belegten Speicherflächen (Umgebung) frei und beendet damit die Existenz von `prog1`

27.4 System Calls — Beispiel read()

```
count = read(fileDeskriptor, buffer, nbytes);
```

Syntax 27.1: read()

Beschreibung:
Vom offenen Kanal, identifiziert über eine kleine Zahl (**Filedeskriptor**), in den *buffer* bis zu *nbytes* Daten lesen.
Zurück kommt die Anzahl der tatsächlich gelesenen Bytes.

Ablauf:

- 1. Schritt
  `read()` ist ein normaler Aufruf einer Bibliotheksfunktion. Die diesem System Call entsprechende *Stub*-Funktion befindet sich in der Systembibliothek, die für jeden System Call eine *Stub*-Funktion enthält.
- 2. Schritt
  Die *Stub*-Funktion für *read()* führt die *trap*-Anweisung mit den entsprechenden Parametern aus und bewirkt damit einen Wechsel in den Kernel Mode.
- 3. Schritt
  Kernel-Routinen arbeiten, um den *read()* System Call zu befriedigen. Nach Abschluß aller Kernel-Routinen erfolgt der Rückwechsel in den User Mode und der Rücksprung zur *Stub*-Funktion.
- 4. Schritt
  Die *Stub*-Funktion kümmert sich noch um den Return-Wert und gibt ihrerseits zurück zum Aufrufer von *read()*.
- 5. Schritt
  Das Programm prüft den Rückgabewert des System Calls und gegebenfalls auch den Wert von **errno**.

  ```
  if((count = read(fileDeskriptor, buffer, nbytes)) < 0 ){
          perror( "read" );
          exit( 1 );
  }
  ```

  Anschließend kann das Programm mit den durch den System Call erhaltenen Daten weiterarbeiten.

Anmerkungen:

- Die meisten System Calls liefern als Rückgabewert `-1` bei Fehler, eine Zahl `>=0`, falls okay
- Viele System Calls sind hinter C-Bibliotheksfunktionen verborgen (C-Interface zu Unix)
- Das C-Interface enthält eine globale Variable `int errno`, die bei Auftreten eines Fehlers die Unix-Fehlernummer enthält.

- Die Datei **errno.h** enthält dazu Fehlertexte
- Bei erfolgreicher Ausführung eines System Calls wird die Fehlernummer (errno) **nicht** zurückgesetzt - also nur abfragen, falls tatsächlich Fehler passiert ist.

27.5 Benutzer- / Datei- / Prozess-Identifikatoren

Dateien, die von Prozessen geöffnet sind, werden ebenso wie die Prozesse selbst über Integer-Zahlen identifiziert.

- **Filedeskriptoren**:
  siehe System Calls im I/O-Subsystem
- **Prozess ID**:
  siehe Kommando **ps** oder Funktion **getpid()**
- **Real User ID**:
  siehe **/etc/passwd**

```
# include <stdio.h>
# include <unistd.h>

void main() {
printf("User Id: %d\n", getuid());
}
```

- **Real Group ID**:
  Benutzergruppen (siehe **/etc/group**, /etc/passwd)

```
# include <stdio.h>
# include <unistd.h>

void main() {
printf("Group Id: %d\n", getgid());
}
```

- **Effective User ID:**
  Im Normalfall identisch mit *real user ID*
  ABER: Manche Programme erhalten ein spezielles Flag (**set-user-ID-Bit**), das besagt: „Wenn dieses Programm ausgeführt wird, ändere die effektive User Id des Prozesses in die User ID des Besitzers der Datei“
  unsigned short geteuid();
- **Effective Group ID**: siehe oben;
  unsigned short getegid();

Abbildungsverzeichnis

Aufruf Syntax

Beispielprogramme

Index