UNIX-Schnittstelle

7. Filesystem

Alle Beispiel-Quellen mittels SVN unter:
https://svn.informatik.hu-berlin.de/svn/unix-2014/File

7.1 Vorbemerkungen

Für Nutzer wichtige Kennzeichen für ein Filesystem

- Wie werden Files erzeugt?
- Wie werden Files benannt?
- Wie werden Files geschützt?
- Welche Operationen über Files gibt es?
- Detailfragen:
  - Abspeicherung von Files auf Medien
  - Freispeicherverwaltung

Prinzipielle Möglichkeiten der Datenspeicherung

1. Bildung von Segmenten (maximale Länge 2^32 Bytes)
   Ein Startprozess verwaltet die Segmente. Programme haben feste Segmentnummern. Bestimmte Segmente werden als Directories benutzt (Verweise auf andere Segmente). Daten werden in Segmenten gespeichert. Reuckgabe von Segmenten möglich.
   negativ: unpraktisch, statisch, störanfällig
   positiv: Daten als Teil des Adressraumes - schnell

2. Files sind benannte Objekte, die Daten, Programme und Sonstiges beinhalten können. Sie gehören nicht zum Adressraum des Prozesses.
   positiv: dynamisch

```
6.1.1 externe Aspekte von Filesystemen

a) Fileorganisation (für Nutzer sichtbar)

     1. Bytefolge (UNIX)
     2. Blockfolge (CP/M)
     3. Baumstruktur (ISAM - Indexed Sequential Access Method)
        Records werden entsprechen eines Schlüssels (key) in einem
        Baum einsortiert.
                                                                FL1
   Files sollen gerätunabhängig sein!!!!!
     D.h. Anwederprogramme sollen nicht berücksichtigen müssen,
     wo und wie das File gespeichert ist.
        Musterbeispiel:  UNIX   (Trick: mount)
        Weniger gut:   MS DOS:   a:\xyz\x.dat
                       CP/M:     a:xyz.dat
                       RSX/11M:  dk0:[100,50]delta.xyz;3

   Namesraum von Files möglichst frei wählbar.

b) Directories  (für Nutzer sichtbar)
     Mehrere Files werden in einer Directory zusammengefasst.
     Pro File ein Record in der Directory.
     In manchen Systemen sind die Directories ebenfalls Files.
     Ein Filesystem kann strukturiert sein.             FL2,FL3,FL6
```



```
7.1.2 Interne Aspekte von Filesystemen
--------------------------------------
7.1.2.1. Plattenspeichermanagement

   Files werden normalerweise auf Platten gespeichert, d.h. File-
   systeme verwalten Plattenspeicher.
   2 Strategien:
   1. Files werden als zusammenhängende Folge von Bytes gespeichert
      problematisch: Speicherzuweisung, Condens
   2. Files als Folge von Blöcken gespeichert, die nicht notwendig
      zusammenhängend auf der Platte stehen.
      momentan realisiert.
      Probleme: Grösse der Blöcke
                  (Zugriffszeit <--> Plattenauslastung)
                optimale Blockgrösse:
                  512 (physische Blockgrösse auf Platte) ....
                  4096 (Blockgrösse für HS-Managment)          IO7
      Freispeicherverwaltung
        1. Verkettet Liste, in der die Blocknummern der freien
           Blöcke enthalten sind
           Vorteil: - je weniger freie Blöcke je weniger Platz
                      wird benötigt
                    - schnell  (max. 20 MB = 40 Blöcke)
        2. Bitmapping: pro Block ein Bit (0-belegt, 1-frei)
           Problem: Speicherplatz (20 MB = 3 Blöcke,
                                   2 GB = 300 Blöcke!!)
                    Sicherheit
                    langsam
           Vorteil: einfache Verwaltung                         FL4
```

```
7.1.2.2. Organisation der Filespeicherung

   Problem: File besteht aus Blöcken. Wie kann die Reihefolge der
            zu einem File gehörenden Blöcke gespeichert werden?
   1. Verkettete List:
      Blocklänge: 1024:  Datenlänge 1022 + 2 Byte Pointer zum nächsten
                          Block
      Nachteil: Blocklänge keine 2er Potenzen
                - Direktzugriff schwer
                - Probleme bei E/A-Fehler
   2. FAT - File Allocation Table  (DOS)
      Pro Block eine Pointer zum nächsten in einer zentralen Tabelle
      Probleme: - ganze FAT ist notwendig um ein File zu bestimmen
                  bei grossen Platten - grosse Speicherbereiche im HS
                  notwendig
      Disketten: 12-Bit-Blocknummer, 4096 Blöcke möglich
      Harddisk:  16-Bit-Blocknummer, 32-MB Platte maximal       FL5
   3. i-node  (UNIX)
      10+256+256*256 +256*256*256 = 16.843.008 Blöcke
                                                                  FL8
7.1.2.3. Directory Struktur

   1. CP/M

   2. MS-DOS

   3. UNIX
                                                  FL9,FL10,FL11,FL12
```



```
7.1.2.4. File-System Wiedererstellung/Sicherung

   Probleme der Datensicherheit (gegen Verluste)

   1. Badblocks
      finden, gegen Benutzung sperren (format)

   2. dynamisches Backup
      Spiegelung von Platten (aufwendig, teuer aber sicher)

   3. statisches Backup zu festen Zeiten auf Backupmedium
      (Kapazität, Zeit, Backuplevel)

   4. Filesystemkonsistenz
      Hilfsprogramme. die die Konsistenz des Filesystems Prüfen
          a) Filekonsistenz
          b) Blockkonsistenz
               Aktionen: - Prüffeld auf 0
                         - alle i-nodes lesen und Blocks eintragen
                         - Freiblockliste lesen
                         - testen
```

```
mögliche Situationen
   1. 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
      1 1 0 1 0 1 1 1 1 0  0  1  1  1  0  0    blocks in use
      0 0 1 0 1 0 0 0 0 1  1  0  0  0  1  1    free blocks
                Konsistent

   2. 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
      1 1 0 1 0 1 1 1 1 0  0  1  1  1  0  0    blocks in use
      0 0 0 0 1 0 0 0 0 1  1  0  0  0  1  1    free blocks
          ^     missing block - ergänzen

   3. 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
      1 1 0 1 0 1 1 1 1 0  0  1  1  1  0  0    blocks in use
      0 0 1 0 2 0 0 0 0 1  1  0  0  0  1  1    free blocks
              ^  doppelter Block in freelist - neu

   4. 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
      1 1 0 1 0 2 1 1 1 0  0  1  1  1  0  0    blocks in use
      0 0 1 1 1 0 0 0 0 1  1  0  0  0  1  1    free blocks
            ^   ^     doppelter Dateblock - kompliziert
       Files kopieren, Files streichen, Freiblockliste neu
```


7.1.2.5. Filesystemdurchsatz

Das Lesen eines Blockes von der Platte ist ca. 10000 mal langsamer als das Lesen aus dem HS.

--> Pufferverwaltung ist sinnvoll, d.h. bestimmte Blöcke sind in internen Puffern zu halten und nicht sofort zu transportieren (cache)

Problem: Austauschalgorithmus wenn Cache voll ist:
- z.B. FIFO oder LRU
  - LRU sinnvoll aber Spezifika des FS berücksichtigen
    - kritische Blöcke: modifizierte i-nodes
    - notwendige Blöcke für nächsten Zugriff (doppelt indirekte Blöcke)
    - volle/halbvolle Datenblöcke
- sync–Systemruf

7.1.2.6. Sicherheitsfragen

Einteilung in:
- Nutzer
- Nutzergruppen
- other

Vergabe unterschiedlicher Zugriffsrechte:
- read
- write
- execute

Passwortschutz

Kerberos, ACL

```
7.2 Systemrufe Filesystem
=========================

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

int creat(const char *pathname, mode-t mode);

Erzeugen eines neuen Files mit dem Filenamen *pathname.
Identisch mit:

       open(pathname,0-WRONLY|O_CREAT|O_TRUNC,mode);

Bestehende Files werden gelöscht und die Länge auf 0 gesetzt,
Mode und Eigentümer bleiben erhalten.
mode spezifiziert die späteren Zugriffsrechte, wenn das File
neu erzeugt wurde.
Früher: Error, wenn File vorher existierte

Rückkehrwert:
  >=0 - Filedescriptor
   <0 - Fehler - siehe    open
```



```
#include <unistd.h>

int dup(int filedes)
int dup2(int filedes, int filedes2);

Doppeln des Filedescriptors fieldes.  Neuer Filedescriptor wird
durch den Rückkehrkode bereitgestellt.  Es ist der nächste
freie Filedescriptor.  Der neue Filedescriptor representiert
das gleiche File am gleichen Zugriffspunkt.
dup2() ist in POSIX nicht enthalten. dup2() benutzt als neuen
Filedescriptor den durch filedes2 spezifizierten.  Sollte sich
hinter filedes2 ein eröffnetes File verbergen, so wird dies
vorher geschlossen (close).  Äuivalent mit:

           fcntl(filedes1, F-DUPFD, fieldes2).

Rückkehrwert:
    >=0 -,neuer Filedescriptor
     <0 - Fehler
           EBADF   filedes oder filedes2 unzulässig
           EMFILE kein freier Filedescriptor mehr
```

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int filedes, off_t offset, int whence);

lseek() positioniert den den Zugriffspunkt des eröffneten Files
filedes an die durch offset und whence spezifizierte Stelle.
Bei lssek() wird kein E/A-Operation ausgeführt.  Die physische
Positionieroperation erfolgt erst bei der nächsten read() bzw.
write() Operation.
whence spezifiziert den Basiswert für die Positionieroperation:
   SEEK_SET - offset gibt die Anzahl der Bytes vom
              Fileanfang an
   SEEK_CUR - offset gibt die Anzahl der Bytes vom momentanen
              Wert des Zugriffspunkts an (offset: +, -)
   SEEK_END - offset gibt die Anzahl der Bytes vom Ende
              Files an (offset:

Rückkehrwert:
       >=0 - momentane Position des Zugriffspunktes
        <0 - Fehler
               EBADF - File nicht eröffnet
               EINVAL - whence ist unzulässig
               ESPIPE - Pipe
Beispiele:
     currpos    = lseek(fd,0,SEEK_CUR);
     filelength = lseek(fd,0,SEEK_END);
```



```
Beispiel:

  Anwendung von seek auf verschiedene Files

    Quelle:
           seek.c

    Ausführung:
          ./seek  </etc/motd
          cat  /etc/motd | ./seek
          mkfifo FIFO
          ./seek < FIFO   # bleibt stehen
          ./seek < FIFO &
          cp /etc/motd FIFO
```

Beispiel:

Holes File erzeugen und dieses File mit verschiedenen Programmen kopieren.

Quelle:

```
hole.c
```

Ausführung:

```
./hole
ls -lisa file.hole
od -bc file.hole
ls -l file.hole
du file.hole
cp file.hole xxx
du xxx
tar cvf file.hole.tar file.hole
ls -l file.hole.tar
du file.hole.tar
tar xvf file.hole.tar
```



```
#include <unistd.h>

int symlink(char *actualpath,  char *sympath);
```

symlink() erzeugt einen neuen Directory-Eintrag sympath für das File actualpath. Das File actualpath muss dabei nicht existieren.

Rückkehrwert:

```
 0 -  ok
-1 -  Fehler
      EACCES   -  kein Zugriff zu sympath
      EDQUOT   -  keine Quota bei Directory-Erweiterung
      EEXIST   -  File sympath existiert bereits
      EFAULT   -  unzulässiger Parameter (Adresse)
      EIO      -  E/A-Fehler
      ELOOP    -  zu viele symbolislche Links
      ENAMETOOLONG - Name zu lang
      ENOSPC - Filesystem ist voll
      ENOTDIR  -  kein Directory
      EROFS    -  read-only-Filesystem
```

```
#include <unistd.h>

int pipe(int filedes[2]);
```

pipe() eröffnet zwei Filedescriptoren, die miteinander verbunden sind. filedes[0] ist dabei zum Lesen eröffnet und filedes[1] ist zum Schreiben geöffnet. Mit Hilfe von fildes[0] können die Daten gelesen werden, die mit filedes[1] in die Pipe geschrieben wurden. Der Pipe-Mechanismus ist eine Halb-Duplex-Verbindung zwischen verwandten Prozessen (Vater-Sohn). Für die Kommunikation wird ein Puffer zur Verfügung gestellt, so dass ein Schreiben erst bei vollem Puffer zum Blockieren des schreibenden Prozesses führt. Wird der lesende Prozess beendet, so erhält der schreibende Prozess ein Signal SIGPIPE. EOF beim Lesen wird erst bei close() des schreibenden Prozesses erkannt,
Pufergröße pro Pipe: ca. 4 - 64 KB

Rückkehrwert:

```
  0 -   ok
 -1 -   Fehler
        EFAULT - Falscher Parameter (Adresse)
        EMFILE - zu viele Filedescriptoren belegt
        ENFILE - Systemtable full
```


Beispiele:

Aufruf des Programms pwd zur Bestimmung des Working Directories

```
Quelle:
     pipe.c    mit dup2

Ausführung:
    ./pipe

Quelle:
     pipef.c  mit fcntl

Ausführung:
    ./pipef
```

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

int fcntl(int filedes, int cmd);
int fcntl(int filedes, int cmd, long arg);
int fcntl(int filedes, int cmd, struct flock *arg);
```

fcntl() ermöglicht die Ausführung einer Reihe von Steuer- und Abfrageoperationen über dem eröffneten File filedes. cmd spezifiziert dabei das Komando und arg enthält weitere Informationen für das Komando.

```
cmd:
     F_DUPPD    - dup2:     newfd=fcntl(fd,F_DUPFD,fdn)  !!!!!!
     F_GETFD    - holen des close-on-exec-Flags
     F_SETFD    - setzen des close-on-exec-Flags ( arq=1 )
     F_GETPL    - holen der Status-Flags für filedes (Rückkehrwert)
     F_SETPL    - setzen der Status-Flags für filedes ( arg )
                     0_RDONLY, 0_WRONLY, 0_RDWR,
                     0_APPEND, 0_NONBLOCK, O_SYNC, O_ASYNC
     F GETLK     holen des 1.Lockdescriptors ( Adresse in arg)
                     struct flock {
                         short l_type; /* F_RDLCK,F_WRLCK,F _UNLCK*/
                         short l_whence; /* flag for starting offset*/
                         off_t  l_start; /* relative start offset */
                         off_t  l_len;    /* length in byte*/
                         pid_t  l_pid;    /* pid which lock F_GETLK*/
```



```
cmd:

      F_SETLK    - lock ein Filesegment  (ohne warten)
      F_SETLKW   - wie F-SETLK mit Warten, wenn lock nicht ausgeführt,
                   werden kann.
      F_RSETLK   - NFS
      F_RSERLKW  - NFS
      F_RGETLK   - NFS

Rückkehrwert:

    >=0 - ok, eventuell Wert:
                F_DUPFD - neuer Filedescriptor
                F_GETFD - close-on-exec-Flag
                F_GETFL - Status-Flag
     <0 - Fehler
             EACCES, EBADF, EDEADLK
             EFAULT, EINTR, EINVAL,
             EMFILE, ENOLCK
```

Beispiel:

Zugriffsrechte mit fcntl bestimmen

Quelle:

```
fileflags.c
```

Ausführung:

```
fileflags 0
./fileflags 0
./fileflags 1
  ./fileflags 2
./fileflags 0 <fileflags.c
./fileflags 0 </dev/tty
```



```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

stat() liefert Informationen über das durch den Pfadname *pathname spezifizierte File.

fstat() liefert Informationen über das eröffnete File mit dem Filedescriptor filedes.

lstat(0 liefert Informationen über das durch den Pfadnamen *pathname spezifiziert File. Eventuelle symbolische Links werden nicht verfolgt.

Die Informationen werden immer in einem Puffer mit der Struktur stat abgelegt.

Rückkehrwert:

```
    0  -  ok
   -1  - Fehler
          stat(), lstat():
               EACCES, EFAULT, EIO, ELOOP,
               ENAMETOOLONG, ENOENT, ENOTDIR
          fstat:
               EBADF, EFAULT, EIO
```

```
struct  stat {   /* SUNOS  4.1.3 */
        dev_t   st_dev;   /* devicenumber (filesystem)*/
        ino_t   st_ino;   /* i-node number */
        mode_t  st_mode;  /* file-type, permissions */
        short   st_nlink; /* number of links */
        uid_t   st_uid;   /* UID of owner */
        gid_t   st_gid;   /* GID of owner */
        dev_t   st_rdev;  /* device number of special files */
        off_t   st_size;  /* size in bytes of regulare files */
        time_t  st_atime; /* time of last access */
        time_t  st_mtime; /* time of last modify */
        time_t  st_ctime; /* time of last file status change */
        long    st_blksize;/* best I/O blocksize */
        long    st_blocks;  /*number of allocated blocks */
};
```



```
Beispiele:

  Tatsächlie Filegröße bestimmen mit lstat

    Quelle
      filesize.c
                        mit lstat
    Ausführung:
         ./filesize file.hole
         ./hole
         ./filesize file.hole

  Filetype bestimmen mit lstat

    Quelle:
      filetype.c

    Ausführung:
         ./filetype .
         ./filetype .
         ./filetype filetype.c
         ./filetype filetype
         ./filetype /dev/tty
         ./filetype /dev/hda
```

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

access() prüft für das durch Pfadnamen *pathname spezifizierte File ob die durch mode geforderten Zugriffsrechte vorhanden sind.

```
mode:
      R_OK    -  Test auf Lesen
      W_OK    -  Test auf Schreiben
      X_OK    -  Test auf Ausführen
      F_OK    -  Test auf Existenz

Rückkehrwert:
    0  - ok, Zugriffsrechte vorhanden
   -1  - Fehler
         EACCES  - keine Zugriffsrechte
         EFAULT, EINVAL, EIO, ELOOP,
         ENAMETOOLONG, ENOENT, ENOTDIR, EROFS
```


Beispiel:

Zugriffsrecht bestimmen mit access

Quelle

access.c

Ausführung:

```
./access access
./access /etc/passwd
./access /etc/shadow
diff access access1
./access1 /etc/shadow
```

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

umask() setzt die Filecreation-Maske. In cmask werden die Bits gesetzt, für die später bei der Filecreation kein Zugriff erlaubt werden soll. Folgende Bits sind für cmask zulässig:

```
S_IRWXU    /* read, write, execute: owner */
S_IRUSR    /* read permission: owner */
S_IWUSR    /* write permission: owner */
S_IXUSR    /* execute permission: owner */
S_IRWXG    /* read, write, execute: group */
S_IRGRP    /* read permission: group */
S_IWGRP    /* write permission: group */
S_IXGRP    /* execute permission: group */
S_IRWXO    /* read, write, execute: other */
S_IROTH    /* read permission: other */
S_IWOTH    /* write permission: other */
S_IXOTH    /* execute permission: other */
```

Rückkehrwert:
    alte Filecreation-Maske

Beispiel:

  Wirkung von umask

    Quelle
        umask.c

    Ausführung:
```
./umask
ls -lisa bar foo
```

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
int fchmod(int filedes, mode_t mode);
```

chmod() und fchmod erlauben es die Zugriffsrechte mode für das durch *pathname bzw. filedes spezifizierte File zu setzen. Dies darf nur der Eigentümer des Files bzw. der SU tun. Für mode sind die gleichen Werte wie bei umask() zulässig, zusätzlich die Werte:

```
S_ISUID  -  set-user-ID on execution
S_ISGID  -  set-group-ID on execution
S_ISVTX  -  saved-text
```

Rückkehrwerte:

```
 0 - ok
-1 - Fehler
     EACCES, EFAULT, EINVAL, EIO, ELOOP, ENOENT,
     ENOTDIR, EPERM, EBADF
```


Beispiel:

Wirkung von chmod

Quelle:

```
changemod.c
```

Ausführung:

```
./umast
ls -lisa bar foo
./changemod
ls -lisa bar foo
```

```
#include <sys/types.h>
#include <unistd.h>

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

Die Systemrufe chown, fchown(), lchown() erlauben es den Eigentümer-ID und den Gruppen-ID eines Files zu ändern.
Das File kann mittels Filename (chown,lchown) oder Filedescriptor (fchown - eröffnetes File) spezifiziert werden.
Ist das spezifizierte File ein symbolischer Link, so wird durch die Funktion lchown() lediglich der UID und der GID des symbolischen Links modifiziert.

Diese Systemrufe können nur vom Eigentümer des Files bzw. dem Superuser ausgeführt werden.

Rückkehrwert:
- 0 - ok
- -1 - Fehler
  EACCESS, EFAULT, EIO, ELOOP,
  ENAMETOOLONG, ENOENT, ENOTDIR,
  EPERM, EROFS, EBADF, EINVAL,


```
#include <sys/stat.h>
#include <unistd.h>

int truncate(const char *pathname, off_t length);
int ftruncate(int filedes, off_t length);
```

Die Systemrufe ftruncate() und truncate() können die Länge eines Files festlegen. Das File kann dabei verkürzt oder verlängert werden. Bei einer Verlängerung werden beim Lesen des Files für die Daten des verlängerten Bereiches 0-Bytes geliefert. Das File kann mittels Filename oder Filedescriptor spezifiziert werden. Bei ftruncate muss das File zum Schreiben eröffnet sein.

Rückkehrwert:
- 0 - ok
- -1 - Fehler
  EACCESS, EFAULT, EIO, EISDIR,
  ELOOP, ENAMETOOLONG, ENOENT,
  ENOTDIR, EROFS, EINVAL

```
#include <unistd.h>

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

Der Systemruf link() erzeugt für das bestehende File *existingpath einen neuen Directory-Eintrag *newpath. Wenn das File *newpath bereits existiert, wird ein Fehler angezeigt. link() ist nur zulässig, wenn *newpath und *existingpath auf das gleiche Filesystem verweisen.

Rückkehrwert:

```
      0 - ok
     -1 - Fehler
          EACCESS, EDQUOT, EEXIST,
          EFAULT, EIO, ELOOP, EMLINK,
          ENAMETOOLONG, ENOENT, ENOSPC,
          ENOTDIR, EPERM, EROFS, EXDEV
```



```
#include <unistd.h>

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

Der Systemruf unlink() löscht den Directory-Eintrag für das File *pathname. Wenn der Directory-Eintrag der letzte für diese File war, so werden ebenfalls die zugehörigen Daten gelöscht. Der Löschvorgang wird erst dann ausgelöst, wenn kein Prozess mehr das File eröffnet hat.
Der Systemruf unlink() kann nur von Nutzern ausgeführt werden, die Schreibzugriff und Execution-Zugriff zum Directory haben. unlink() für einen symbolischen Link bewirkt das Löschen des symbolischen Links selbst.

Rückkehrwert:

```
   0  -  ok
  -1  -  Fehler
         EACESS, EBUSY, EFAULT, EINVAL,
         EIO, ELOOP, ENAMETOOLONG,
         ENOENT, ENOTDIR, EPERM, EROFS
```

Beispiel:

Demonstration von unlink, Speicherfreigabe erst nach close

Quelle:

```
unlink.c
```

Ausführung:

```
# großes File big1 erzeugen
ls -lisa big1
df -k .
./unlink big1 &
ls -lisa big1
df -k .
```



```
#include <unistd.h>

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

Die Systemfunktion rmdir() realisiert das Streichen einer Directory mit dem Namen *pathname. Die Directory muss leer sein und darf z.Z. nicht von anderen Prozessen benutzt werden. Die zugehörigen Datenblöcke werden freigegeben.

Rückkehrwert:
- 0 – ok
- -1 – Fehler
  EACCES, EBUSY, EFAULT, EINVAL,
  EIO, ELOOP, ENAMETOOLONG, ENOENT,
  ENOTDIR, ENOTEMPTY, EROFS

```
#include <unistd.h>

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

Streichen von Files und Directories (ANSI-C). Funktion wie unlink und

Rückkehrwert:
- 0 – ok
- -1 – siehe oben

```
#include <stdio.h>

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

Der Systemruf rename() erlaubt das Umbenennen von Files und Director
Ist *oldpath ein File, so muss *newname nicht existieren bzw. kann a
File verweisen, das dann gelöscht wird. Ist *oldpath eine Directory,
so muss *newname nicht existieren bzw. kann auf eine leere Directory v
die dann gelöscht wird. Der Prozess muss Execution- und Schreibzugriff
zu den Directories haben, in denen das File/Directory momentan steht
bzw. stehen wird.

Rückkehrwert:
 0 - ok
 -1 - Fehler
 EACCES, EBUSY, EDQUOT, EFAULT,
 EINVAL, EIO, EISDIR, ELOOP,
 ENAMETOOLONG, ENOENT, ENOSPC,
 ENOTDIR, ENOTEMPTY, EROFS, EXDEV


```
#include <unistd.h>

int readlink(const char *pathname, char *buff, int buffsize);
```

Die Systemfunktion readlink() erlaubt das Lesen eines symbolischen
Links *pathname in den Puffer *buff mit er Länge buffsize.
Dieser Ruf vereinigt open(), read() und close().

Rückkehrwert:
 >=0 - Anzahl der in den Puffer gelesenen Zeichen.
 -1 - Fehler
 EACCES, EFAULT, ELOOP, EINVAL, EIO,
 ENAMETOOLONG, ENOENT

Folgende Systemrufe folgen symbolischen Links:

access, chdir, chmod, chown, creat, exec, link, mkdir, mkfifo,
mknod, open, opendir, pathconf, stat, truncate

Folgende Systemrufe wirken direkt auf symbolische Links:
lchown, lstat, readlink, remove, rename, unlink

Beispiele:

```
$ mkdir bell1
$ touch bell1/xxxx
$ ln -s ../bell1 bell1/dir
$ ls -l bell1
total 1
lrwxrwxrwx  1 bell  8 Jan 25 17:43 dir -> ../bell1
-rw-r--r--  1 bell  0 Jan 25 17:42 xxxx
$ cd bell1/dir
$ pwd
/tmp/bell/bell1
$ ls
dir     xxxx
$ cd dir
$ ls
dir     xxxx
$ ls -l
total 1
lrwxrwxrwx  1 bell            8 Jan 25 17:43 dir -> ../bell1
-rw-r--r--  1 bell            0 Jan 25 17:42 xxxx
$
$ ln -s /tmp/no/file file
$ ls file
file
$ cat file
cat: file: No such file or directory
$ ls -lisa file
 11377  1 lrwxrwxrwx  1 bell  12 Jan 25 17:45 file -> /tmp/no/file
$
```



```
#include <sys/types.h>
#include <utime.h>

int utime(const char *pathname, const struct utimebuf *times);

struct utimebuf {
     time_t  actime;  /* access time */
     time_t  modtime; /* modification time */
}

#include <sys/time.h>

int utimes(const char *pathname, const struct timeval times[2]);

struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* and microseconds */
}
```

Die Systemrufe utime() und utime() ermöglichen es Zugriffszeit und Modifikationszeit eines Files zu ändern. Wenn *times der NULL-Pointer ist wird die aktuelle Zeit als Zugriffs- und Modifikationszeit eingetragen. Der effektive UID des Prozesses muss zum Ändern identisch mit dem Eigentümer UID sein oder 0 (Superuser).
utimes() hat eine Genauigkeit von einer Mikrosekunde.

Rückkehrwert:

- 0 – ok
- -1 – Fehler
  EACCES, EFAULT, EIO, ELOOP, ENOENT, ENOTDIR, EPERM, EROFS

Beispiel:

Umsetzen der Modifikationszeit mittels utime()

Quelle:

```
zap.c
```

Ausführung:

```
cat yyy.c
ls -l yyy.c
./zap yyy.c
ls -l yyy.c
ls -lc yyy.c
```



```
#include <sys/types.h>
#include <sys/stat.h>

int mknod(const char *pathname, int mode, int dev);
int mkfifo(const char *pathname, int mode);
```

Der Systemruf mknod() dient zum erzeugen von Specialfiles (Block- und Characterdevices), FIFO's (named pipes), Directories und gewönlichen Files. Der Name des zu erzeugenden Objekts wird durch *pathname festgelegt. Die Art des Objektes durch mode :

```
S_IFDIR  0040000 /* directory */
S_IFCHR  0020000 /* character special */
S_IFBLK  0060000 /* block special */
S_IFREG  0100000 /* regular */
S_IFIFO  0010000 /* fifo */
```

festgelegt. dev dient zur Spezifikation der Geräte(Major- und Minornummer). Die Zugriffsrechte werden durch die Filecreationmaske bestimmt. mknode() kann nur durch den Superuser ausgeführt werden. Der normale Nutzer darf lediglich FIFO's und reguläre Files erzeugen. Der Systemruf mkfifo() ist eine abgerüstete Variante von mknod(). Er dient lediglich zum Erzeugen von FIFO's.

Rückkehrwert:

- 0 – ok
- -1 – Fehler
  EACCES, EDQUOT, EEXIST, EFAULT, EIO, EISDIR, ELOOP, ENAMETOOLONG, ENOENT, ENOSPC, ENOTDIR, EPERM, EROFS

```
Beispiel:

   Major und Minor-Nummber mittels lstat()

     Quelle:
          devrdev.c

     Ausführung:
          ./devrdev / /usr/ /dev/ttyS[01]
          ls -ld /dev/hda7 /dev/ttyS[01]
```



```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

Der Systemruf `mkdir()` erzeugt eine neue Directory mit den in `mode` spezifizierten Zugriffsrechten. Filecreationmaske wird berücksichtigt. Achtung: Execution-Bits setzen für Suchen.

Rückkehrwert:

```
   0  -  ok
  -1  -  Fehler
         EACCES, EDQUOT, EEXIST,
         EFAULT, EIO, ELOOP,
         ENAMETOOLONG, ENOENT,
         ENOSPC, ENOTDIR, EOFS
```

Funktionen zur Unterstützung des Directoryzugriffs

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *pathname);

struct dirent *readdir(DIR *dp);

void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

Die Funktionen opendir(), rewinddir(), readdir() und closedir() diene zum systemunabhängigen Lesen von Directories. open erröffnet ein Directory. readdir() liest einen Directoryeintrag in die standardisierte Struktur dirent. rewinddir() setzt die den Lesezeiger von readdir() auf den Anfang der Directory. closedir() schliesst die Directory.

```
   struct dirent {
       ino_t d_ino;              /* i-node number */
       char  d_name[NAME_MAX];   /* null-terminated filename */
   }
```


Beispiel:

Anzeigen einer Directory mit opendir, readdir
ls1.c

Rekursives anzeigen von Directories
ftw2.c

Rekursives anzeigen von Directories + Statistik
ftw3.c

```
#include <unistd.h>

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

Jeder Prozess hat ein aktuelles Arbeitsdirectory, das benutzt wird, wenn ein Filename nicht mit einem "/" anfängt. Die Systemrufe chdir() und fchdir() erlauben das Setzen des aktuellen Arbeitsdirectories. chdir() verlangt die Spezifikation des Directories in Form des Pfadname *pathname und fchdir verlangt die Spezifikation des Directories als Filedescriptorpointer fieldes. Für die Directory muss der Prozess Lese- und Ausführungsrechte besitzen.

Rückkehrwert:

```
   0  -  ok
  -1  -  Fehler
         EACCES, ENAMETOOLONG, ENOENT, ENOTDIR
```


Beispiel:

Wirkung von Change Directory demonstrieren

```
    Quelle:
         mycd.c
         cdpwd.c
    Ausführung:
         pwd
         ./mycd
         pwd
         ./cdpwd
         pwd
```

```
int chroot(char *dirname);
int fchroot(int filedes);
```

Die Systemrufe chroot() und fchroot() dienen dem Superuser zum Ändern des Root-Verzeichnisses für den aktuellen Prozess. Die Lage des neuen Root-Verzeichnisses wird durch die Parameter *dirname bzw. filedes bestimmt. Nach Ausführung der Systemrufe hat der aktuelle Prozess nur noch Zugriff auf Files und Directories, die unterhalb des als neues Root-Directory liegen.

Rückkehrwert:

```
 0 - ok
-1 - Fehler
     EACCES, EBADF, EFAULT, EINVAL, EIO, ELOOP,
     ENAMETOOLONG, ENOENT, ENOTDIR, EPERM
```



```
#include <sys/mount.h>

int mount(char *type, char *dir, int flags, caddr_t data); /*BSD*/
int mount(char *spec, char *dir, int rdonly);  /* System V */
int mount(const char *source, const char *target, const char *filesyst
          unsigned long mountflags, const void *data);  /*linux*/
```

Der Systemruf mount() dient für das Eingliedern eines blockorientierten Gerätes in das bestehende Filesytem. Auf dem Gerät muss vorher ein Filesystem installiert sein. Der Mountpoint wird durch den Parameter *dir festgelegt. data bzw. *spec spezifizier das blockorientierte Gerät das "eingemountet" werden soll. Durch *typ wird die Art des Filesystems spezifiziert, das "eingebunden werden sol z.B. 4.2, nfs, rfs, hsfs. Für die verschiedenen Filesystemtypen muss eine entsprechende Datenstruktur data bereitgestellt werden. Der Parameter flags spezifiziert Zugriffsrechte:

```
M_RDONLY   - nur lesen      M_NOSUID  -  ignoriere Set-UID-Bit
M_NEWTYPE  - immer gesetzt  M_GRPID   -  BSD-Filecreation
M_REMOUNT  - ändern         M_NOSUB   -  keine Submounts
```

Rückkehrwerte:

```
 0  -  ok
-1  -  Fehler
       EACCES, EBUSY, EFAULT, ELOOP, ENAMETOOLONG, ENODEV, ENOENT,
       ENOTDIR, EPERM, EINVAL, EIO, EMFILE, ENOMEM, ENOTBLK, ENXIO
```

(nach Volker G

Mounten der Festplatte 2 in das Filesystem der Festplatte 1
(Mountpunkt-Direktory /mnt/p3 ist leer)

```
        Festplatte 1
 ____________________________
|                            |
|          /                 |
|          :                 |
|    .....:......            |
|    :     :     :           |
|   usr   etc   mnt          |                Festplatte 2
|                :           |            _____________________
|          .....:.....       |   mount   |                     |
|           :    :    :      |           |                     |
|           p1   p2   p3 <---|-----------|---------- /         |
|____________________________|           |           :         |
                                         |   ......:.......    |
                                         |   :      :      :   |
                                         | data1  data2  data3 |
                                         |_____________________|
```


Mounten der Festplatte 2 in das Filesystem der Festplatte 1
(Mountpunkt-Direktory /mnt ist nicht leer)

```
        Festplatte 1
 ____________________________
|                            |
|          /                 |                Festplatte 2
|          :                 |            _____________________
|    .....:......            |   mount   |                     |
|    :     :     :           |           |                     |
|   usr   etc   mnt <--------|-----------|---------- /         |
|                            |           |           :         |
|          .....*.....       |           |   ......:.......    |
|           :    :    :      |           |   :      :      :   |
|           p1   p2   p3     |           | data1  data2  data3 |
|____________________________|           |_____________________|
```

Achtung!! nach dem Mounten sind die Directories/Files p1, p2, p3 nicht mehr sichtbar. Nach dem Entmounten sind sie wieder sichtbar.

```
int unmount(char *dir);  /* BSD */
int unmount(char *special);  /* System V */
int umount2(const char *target, int flags); /*Linux, Solaris*/
```

Der Systemruf unmount() gliedert ein zuvor mit mount() eingegliederte blockorientiertes Gerät aus dem Filesystem aus. Als Parameter muss der Name des Special-Files in *special bzw. die Directory in *dir angegeben werden. Der Systemruf erfodert Superuserrechte. Das auszugliedernde Filesystem darf von keinem anderen Prozess z.Z. genutzt werden.

Rückkehrwert:

```
   0 -  ok
  -1 -  Fehler
        EACCES, EBUSY, EFAULT, EIO, ELOOP, ENAMETOOLONG,
        ENOENT, ENOTDIR, ENOTBLK, ENXIO
```



```
int sync(void);
```

Der Systemruf sync() synchronisiert das Filesystem. Alle modifiziert Blöcke, die bisher noch nicht auf die Platte gebracht wurden, werden auf die Platte geschrieben, einschliesslich Superblock.

Rückkehrwert:

```
  0  -  ok
```

```
int fsync(int filedes);
```

Der Systemruf fsync() synchronisiert alle Blöcke, die zu dem File gehören, das durch den Filedescriptor filedes spezifizierten wird. Der Systemruf fsync() realisiert damit sync() für ein File.

Rückkehrwert:

```
   0  -  ok
  -1  -  Fehler
         EBADF, EINVAL, EIO
```

```
#include <sys/file.h>

int flock(int filedes, int operation);
```

Der Systemruf `flock()` erlaubt es ein eröffnetes File gegen Zugriff zu schützen. `filedes` spezifiziert den Filedescriptor des Files und `operation` gibt die Art des Lockzugriffs an:

```
     LOCK_SH  1   - shared lock
     LOCK_EX  2   - exclusive lock
     LOCK_NB  4   - don't block when locking
     LOCK_UN  8   - unlock
```

Rückkehrwert:

```
   0  -  ok
  -1  -  Fehler
         EBADF, EOPNOTSUPP, EWOULDBLOCK
```


Beispiel :
"nonblocking" write

```
     Quelle:
         nonblockw.c
     Ausführung:
         ls -lisa /etc/termcap
         ./nonblockw < /etc/termcap
         ./nonblockw < /etc/termcap 2>xxx
         ls -l xxx
         more xxx

     Quelle:
         flock.c
     Ausführung:
        ./flock r xxx &
        ./flock r xxx &
        ./flock w xxx &
        ./flock w xxx &

     Quelle:
        flock1.c
     Ausführung:
        ./flock1 w xxx &
        ./flock1 w xxx &
        ./flock1 r xxx &
        ./flock1 r xxx &
```

```
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int select(int maxfdpl, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *tvptr);
```

Der Systemruf select() erlaubt es auf mehrere E/A-Bedingungen gleichzeitig zu warten bzw. zu testen ob E/A-Anforderungen ausgeführt würden. maxfdl spezifiziert den Wert des höchsten File-descriptors. Durch die Mengen fd_set werden die Filedescriptoren und die Art der E/A-Operationen festgelegt, die berücksichtigt werden sollen. Die einzelnen Mengen legen dabei die Art der E/A-Operation fest, die mit dem entsprechenden Filedescriptor ausgeführt wird:

- readfs - Descriptoren für Leseoperationen
- writefds - Descriptoren für Schreiboperationen
- exceptfds - Descriptoren für Ausnahmebedingungen

Die Mengen werden mit Hilfe folgender Funktionen manipuliert:

```
FD_ZERO(fd_set *fdset);             /* clear fdset */
FD_SET(int fd, fd_set *fdset);      /* set bit for fd in fdset */
FD_CLR(int fd, fd_set *fdset);      /* clear bit for fd in fdset */
int FD_ISSET(int fd, fd_set *fdset);/* test bit for fd in fdset */
```


Ausserdem kann mittels *tvptr festgelegt werden, wie lange gewartet werden soll. *tvptr hat folgenden Aufbau:

```
struct timeval {
    long tv_sec;  /* seconds */
    long tv_usec */
}
```

Ist tvptr==NULL, so wird gewartet, bis ein Filedescriptor das Ende einer E/A-Operation meldet. Ist tvptr->tv_sec==0 und tvptr->tv_usec==0, wird nicht gewartet. Ansonsten wird die spezifizierte Zahl von Sekunden gewartet.
Wenn select() zurückkehrt, sind in den einzelnen Descriptormengen die Filedescriptoren eingetragen (Bit gesetzt), für die eine E/A-Operation beendet wurde.

Rückkehrwert:

- >0 - Anzahl der Filedescriptoren, die das Ende einer E/A-Operation gemeldet haben
- =0 - Timeout, keine E/A-Operation wurde beendet
- <0 - Fehler
  EBADF, EFAULT, EINTR, EINVAL

Beispiel:

Bestimmen der Blockgröße einer Pipe mittels select

Quelle:

selectpipe.c

Ausführung:

```
$
$ ./selectpipe
pipe capacity = 61441
$
```



```
#include <stropts.h>
#include <poll.h>

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

Der Systemruf poll() erlaubt es auf mehrer E/A-Ereignisse gleichzeitig zu warten. Jedes Ereignis muss einzeln in fdarray spezifiziert werden. Die Zahl der Ereignisse wird durch nfds angegeben. Mittels timeout kann eine Wartezeit spezifiziert werden.

Die Struktur pollfd hat folgenden Aufbau:

```
struct pollfd {
        int    fd;      /* filedescriptor to check */
        short  events   /* events of interested on fd */
        short  revents; /* events that occurred on fd */
}
```

value	events	revents	Description
POLLIN	x	x	normal data can be read
POLLPRI	x	x	priority data can be read
POLLOUT	x	x	normal data can be write
POLLERR		x	an error has occurred
POLLHUP		x	a hangup has occurred
POLLNVAL		x	filedescriptor not open

timeout spezifiziert die Wartezeit in Millisekunden.

```
    timeout==0     -   nicht warten
    timeout==-1    -   warten bis zum
    timeout> 0     -   Wartezeit in Millisekunden
```

Rückkehrwert:

```
    >0   -  Zahl der Filedescriptoren, für die ein Ereignis
            eingetreten ist
    =0   -  Timeout, kein Ereignis eingetreten
    <0   -  Fehler
            EAGAIN, EFAULT, EINTR, EINVAL
```


Beispiel:

Bestimmung der Größe einer Pipe mittels poll()

Quelle:
pollpipe.c

Abarbeitung:

```
$
$ ./pollpipe
pipe capacity = 61441
$
$
```

```
#include <sys/types.h>
#include <sys/uio.h>

size_t readv(int filedes, const struct iovec iov[], int iovcnt);
size_t writev(int filedes, const struct iovec iov[], int iovcnt);
```

Die Systemrufe readv() bzw. writev() ermöglichen das Lesen bzw. Schreiben in/aus verschiedenen Puffern mit jeweils verschienen Längen aus/in ein File. Das File wird durch den Filedescriptor fildes bestimmt. iov adressiert ein Feld von Strukturen des Types iovec. iovcnt gibt die Anzahl der Strukturen (Anzahl der Puffer) an.

```
      struct iovec {
         void   *iov_base; /* starting address of buffer */
         size_t  iov_len;  /* size of buffer */
      }
```

Rückkehrwert:

```
   >=0  -  Anzahl der gelesenen bzw. geschriebenen Bytes
    <0  -  Fehler
           EAGAIN, EBADF,  EBADMSG, EFAULT, EINTR, EINVAL,
           EIO, EISDIR, EWOULDBLOCK, EDQUOT, EFBIG, ENOSPC,
           ENXIO, EPIPE, ERANGE
```



```
#include <sys/types.h>
#include <sys/mman.h>

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

Der Systemruf mmap() dient zum Einblenden eines Teils eines eröffneten Files in den Adressraum des Prozesses. addr spezifiziert die Adresse innerhalb des Adressraums des Prozesses, an der die Einblendung beginnen soll. Ist addr==NULL, wird die nächste frei Adresse benutzt. len gibt die Länge des Speicherbereiches an. filedes spezifiziert das eröffnete File, das eingeblendet werden soll off gibt die Stelle im File an, ab der die Daten des Files eingeblendet werden sollen. prot spezifiziert die Zugriffsrechte:

```
    PROT_READ  -  Lesen
    PROT_WRITE -  Schreiben
    PROT_EXEC  -  Ausführen
```

flag gibt zusätliche Eigenschaften an:

```
    MAP_FIXED   -  Einblendung muss an Adresse addr erfolgen
    MAP_SHARED  -  Änderung im Speicherbereich sind ebenfalls im
                   File zu realisieren (write)
    MAP_PRIVATE-  Änderungen sind nur im Speicher durchzuführen
```

Rückkehrwert:

```
   !=NULL - Adresse, an der die Einblendung beginnt
   =NULL  - Fehler
            EACCES, EAGAIN, EINVAL, ENODEV, ENOMEN, ENXIO
```

```
#include <sys/mman.h>

int munmap(caddr_t addr, int len);
```

Der Systemruf munmap() hebt eine zuvor mit mmap() gemachte Einblendu eines Fileabschnittes in den Speicher wieder auf. addr spezifiziert d Adresse (Rückkehrwert von mmap) und len gibt die Länge des Bereiches an.

Rückkehrwert:

```
   0  -  ok
  -1  -  Fehler
         EINVAL
```


Beispiel:
Kopieren eines Files ohne read und write ueber mmap.

Quelle:
mcopy.c

Ausführung:

```
$
$ ./mcopy mcopy test
$ ls -lisa mcopy test
7607   24 -rwxr-xr-x  1 bell        24576 Feb  1 20:59 mcopy
7605   24 -rwxr-xr-x  1 bell        24576 Feb  1 21:00 test
$ ./test mcopy test1
$ ls -lisa mcopy test1
7607   24 -rwxr-xr-x  1 bell        24576 Feb  1 20:59 mcopy
7608   24 -rwxr-xr-x  1 bell        24576 Feb  1 21:00 test1
$
```