

Systemaufrufe

Inhalt

1. Systemaufrufe	2
2. Prozessverwaltung.....	5
3. Signale	12
4. Dateiverwaltung	19
5. Katalogverwaltung.....	32
6. Schutzmechanismen	37
7. Zeitverwaltung.....	41

1. Systemaufrufe

Die vorgestellten Systemaufrufe sind an der **Implementierung in Linux** orientiert. In anderen Betriebssystemen sind die Umsetzungen zwar teilweise anders, das Prinzip ist aber das gleiche.

Systemaufrufe bilden die Schnittstelle zur Hardware, auf dem das Betriebssystem abläuft. Deshalb sind große Teile eines Systemaufrufs in Assembler programmiert. Um sie für Programmierer nutzbar zu machen, wird oft eine **C-Bibliothek** bereitgestellt.

Ein einfacher Systemaufruf zum Lesen einer Datei ist „**read**“. Mit drei Parametern von „read“ wird beschrieben, welche Datei gelesen werden soll, wohin die Leseoperation das Ergebnis ablegen soll und wie viele Bytes aus der Datei gelesen werden sollen. Der Aufruf in einem C-Programm hat folgendes Aussehen:

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

Durch den Systemaufruf werden wird die durch „file“ angegebene Datei gelesen und „count“ Bytes in die Variable „buffer“ gespeichert. Normalerweise ist „count“=„nbytes“, aber wenn das Dateiende erreicht wird, sind u.U. weniger als „nbytes“ Bytes zum Lesen da. Wenn der Systemaufruf nicht ausgeführt werden kann (Plattenfehler oder Datei nicht lesbar), wird „count“ auf -1 gesetzt und die Fehlernummer wird in einer globalen Variablen „errno“ abgelegt.

Ein vollständiges C-Programm (simpleCat.c), das eine Datei liest und auf dem Bildschirm ausgibt, ist nachfolgend gezeigt:

```
#define BUFSIZE 512
main()
{
    char buf[BUFSIZE];
    int n;
    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
}
```

Das Programm wird von Shellebene aus mit dem Kommando

```
$ cc simpleCat.c -o simpleCat
$
```

übersetzt. Die resultierende ausführbare Datei „simpleCat“ kann nun verwendet werden, um Dateien zu lesen, etwa durch das Kommando:

```
$ simpleCut < simpleCut.c
#define BUFSIZE 512
main()
..
$
```

Im Folgenden werden nun die wichtigsten Systemaufrufe kurz vorgestellt. Dabei werden sie gemäß folgender Gruppen diskutiert:

- Prozessverwaltung

- Signale
- Dateiverwaltung
- Katalog- und Dateisystemverwaltung
- Schutzmechanismen
- Zeitverwaltung

2. Prozessverwaltung

Ein Prozess wird erzeugt, indem ein Eltern Prozess durch den Systemaufruf „fork“ ein Kind Prozess erzeugt. Der Aufruf erzeugt eine exakte Kopie des Originalprozesses (Kind=Clone des Vaters), einschließlich aller Dateideskriptoren, Register usw. Nach dem fork wird der Vater und das Kind unterschiedliche Aktivitäten übernehmen. Zum Zeitpunkt des fork haben alle Variable die gleichen Werte, nach dem fork wirken sich Änderungen der Variablen nur noch im jeweiligen Prozess aus.

Der **fork** Aufruf gibt einen Wert zurück, durch den im Programm unterschieden werden kann, ob der Kode des Kindes oder des Vaters gemeint ist: NULL ist der Kindprozess, Wert größer 0 ist die Prozessidentifikation (pid) des Kindprozesses. Ein Rückgabewert von fork, der kleiner als 0 ist, zeigt an, dass kein neuer Prozess erzeugt werden konnte.

```

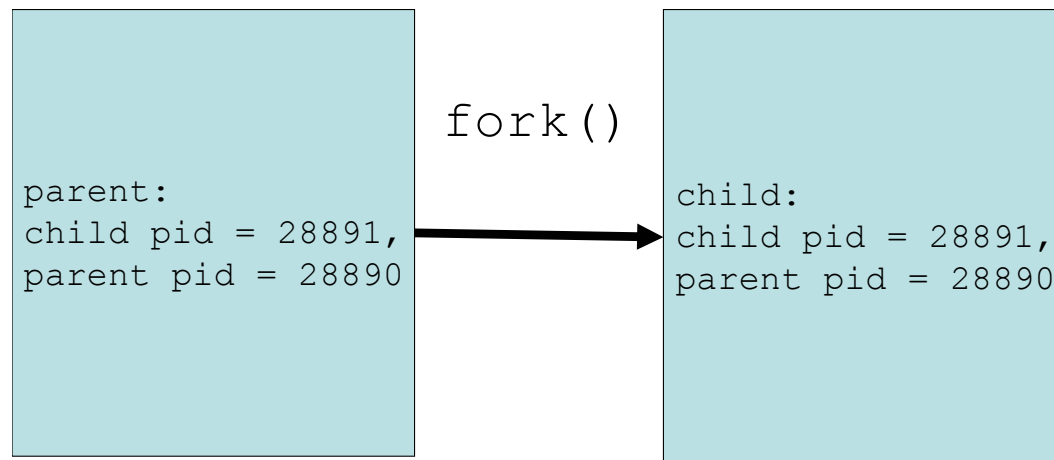
#include <stdio.h>
main()
{
    int childPid;
    if ((childPid = fork()) == -1) {
        fprintf(stderr, "can't fork\n");
        exit(1);
    } else if (childPid == 0) {
        /* child process */
        fprintf(stdout, "child: child pid = %d,
                        parent pid = %d\n",
                        getpid(), getppid());

        exit(0);
    } else {
        /* parent process */
        fprintf(stdout, "parent: child pid = %d,
                        parent pid = %d\n",
                        childPid, getpid());

        exit(0);
    }
}

```

Das C-Programm produziert folgende Ausgabe.

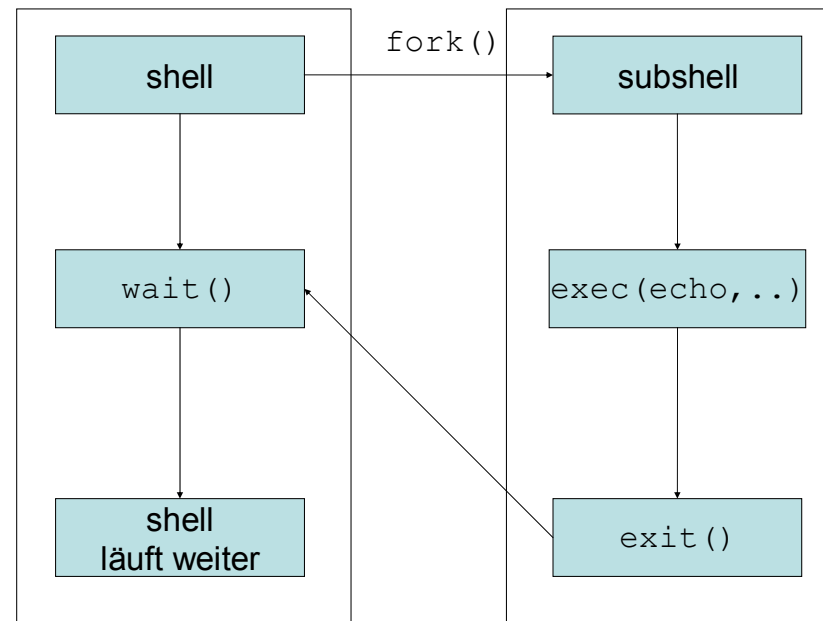


Ausgabe der beiden Prozesse,
die durch Aufruf des C-Programms erzeugt werden

Der „exit“ Systemaufruf beendet die Prozessaufführung und gibt den Exitstatus zurück. Die Systemaufrufe „getpid“ und „getppid“ liefern die PID von einem Prozess resp. die PID des eigenen Erzeugers.

Ein reales Beispiel, bei dem ein Prozess erzeugt wird, ist die Shell. Für jedes Kommando, das aus der Shell heraus ausgeführt wird, wird von der Shell ein eigener Prozess erzeugt. Dabei dupliziert sich die Shell, überlagert den eigenen Code mit dem Code des auszuführenden Kommandos und wartet bis der so erzeugte Prozess terminiert.

Dazu sind die Systemaufrufe „exec“ und „wait“ wie im nachfolgenden Schaubild verdeutlicht, verwendet. Dabei von der Shell das Kommando echo aufgerufen.



Ausführung „echo“-Kommando durch Shell

Die Shell erzeugt durch fork einen Prozess und führt den Systemaufruf „wait“ aus. Dadurch wartet sie, bis sie ein Signal erhält. Dieses Signal wird vom „exit“ Aufruf des Kindprozesses erzeugt. Der Kindprozess ruft des Systemaufruf „exec“ auf. Er nimmt den Code des ersten Parameters (hier das echo Kommando) und überlagert den eigenen Code damit. Somit kann in der Umgebung des Kindprozesses, das echo Kommando ausgeführt werden. Es existieren mehrere Varianten des exec Systemaufrufes. Wir werden sie später genauer kennen lernen.

Das folgende Programm stellt eine Minishell dar, die nach dem o.a. Prinzip funktioniert.

```

#include <stdio.h>
void read_command(char *com, char **par) {
    fprintf(stdout, "$ ");
    .....
    return;
}
main()
{
    int childPid;
    int status;
    char command[20];
    char *parameters[60];
    while (1) {
        read_command(command, parameters);
        if ((childPid = fork()) == -1) {
            fprintf(stderr, "can't fork\n");
            exit(1);
        } else if (childPid == 0) {
            /* child */
            execv(command, parameters);
            exit(0);
        } else { /* parent process */
            wait(&status);
        }
    }
}

```

„exit“ hat einen Parameter, den so genannten Exitstatus; dies ist ein Integer zwischen 0 und 255. Konvention in Unix ist, dass ein Exitstatus von Null bedeutet, dass die Aktion erfolgreich ausgeführt werden konnte. Jeder andere Wert wird als Fehler angesehen. Dieser Status wird dem Elternprozess in der Variablen „status“ des wait Aufrufs mitgegeben.

Das niederwertige Byte von „status“ enthält 0 für normale Terminierung des Kindes, einen Fehlercode, wenn Fehler aufgetaucht sind. Im höherwertigen Byte ist der Exitstatus des Kindes gespeichert. Der Returncode von „wait“ gibt ist die PID des Kindes.

Soll z.B. eine Kommunikation zwischen Kind und Eltern stattfinden, so kann das Kind z.B. durch `exit(4);`

dem Elternprozess die Nachricht „4“ übergeben. Der Elternprozess wird durch `n = wait(status);`

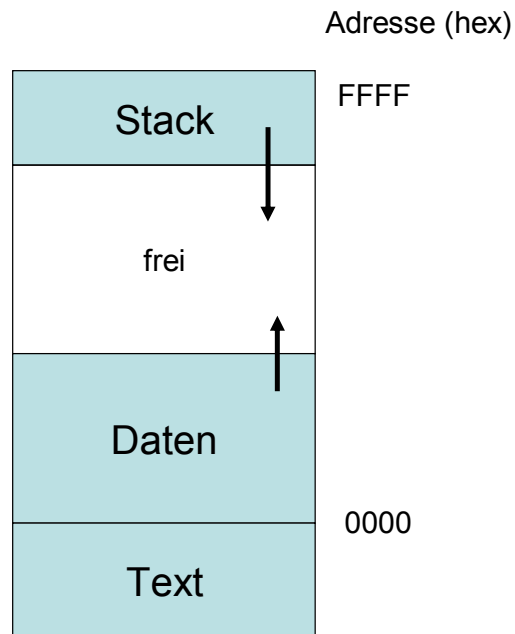
dann die Information im der Variablen „status“ sehen:

niederwertige Byte	höherwertige Byte
0000000000000000	0000000000000010

Der **Prozessraum** in Linux ist in drei Teile gegliedert:

- das Textsegment (text segment) für den Programmcode,
- das Datensegment (data segment) zur Speicherung von Daten und
- das Stacksegment (stack segment) um z.B. Aufrufe von Funktionen abzulegen.

Der Prozessraum ist wie folgt organisiert:



Daten- und Stacksegmente laufen von unterschiedlichen Seiten in die freie Lücke des Prozessraums.

Durch den Systemaufruf „brk“ kann man die Expansion des Datensegments veranlassen. Der Aufruf hat einen Parameter, der die Adresse des Prozessraums angibt. Ist sie größer als die momentane Grenze, so wächst das Datensegment, ist sie kleiner, so schrumpft es. Überlappungen von Stack- und Datensegment sind nicht möglich.

3. Signale

Signale sind das Äquivalent im Bereich Software zu Interrupts im Bereich Hardware. Programme, die als Prozess innerhalb des Betriebssystems ablaufen, müssen unterbrechbar sein, damit z.B. ungewollte Aktionen nicht ausgeführt werden. So kann die Ausgabe (durch das Kommando `cat`) einer großen Datei mit „CTR-C“ abgebrochen werden. Durch „CTR-C“ wird dem Ausgabeprozess ein Signal gesendet. Als Reaktion beendet der Prozess die Ausgabe.

Wenn ein **Signal** zu einem Prozess gesendet wird und der Prozess das Signal nicht annimmt, dann wird der Prozess vom Betriebssystem **automatisch entfernt**. Um sich vor diesem Automatismus schützen zu können, kann sich ein Prozess durch den Systemaufruf „`signal`“ auf das Eintreffen von Signalen vorbereiten. Dazu muss er eine **Signalbehandlungs-Routine** bereitstellen.

Ein Prozess, der eine Signalbehandlungs-Routine bereitgestellt hat, wird bei Eintreffen eines Signals angehalten, der Prozesszustand auf des Stack geschrieben und die Signalaroutine wird aufgerufen. Diese Routine darf selbst beliebige Systemaufrufe veranlassen. Ist sie beendet, wird der Prozess da weiter ablaufen, wo er vorher unterbrochen wurde (der Zustand wird vom Stack restauriert).

In einem Betriebssystem gibt es mehrere Signalarten. Die meisten Signale werden durch Ereignisse, die von der Hardware ausgelöst werden erzeugt. Die nachfolgende Tabelle listet die wichtigsten Signalarten:

Nummer	Bezeichnung	Bedeutung
1	SIGHUP	Hang up, Modemunterbrechung
2	SIGINT	DEL Taste
3	SIGQUIT	Quit Signal von Tastatur
4	SIGILL	Nicht erlaubte Instruktion
5	SIGTRAP	Unterbrechung für Testzwecke
8	SIGFPE	Gleitkommaüberlauf
9	SIGKILL	Abbruch
10	SIBBUS	Busfehler
11	SIGSEGV	Segmentfehler
12	SIGSYS	Ungültige Argumente bei Systemaufruf
13	SIGPIPE	Ausgabe auf Pipe ohne Leser
14	SIGALARM	Alarm
15	SIGTERM	Softwareerzeugtes Endesignal
16	frei	

Die Signalbehandlungs-Routine sollte als erstes selbst wieder den Mechanismus zur Signalbehandlung aufrufen, damit ein erneut eintretendes Signal ebenfalls abgefangen wird.

Das folgende Beispiel zeigt, wie in C eine Signalbehandlungs-Routine realisiert wird. Dabei wird das Programm (eine Unendlichschleife) auf das Eintreffen vom Signal „SIGINT“, ausgelöst durch „CTR-C“ reagieren, indem es einen Text auf die Ausgabe schreibt.

```

$ cat signal.c
#include <stdio.h>
#include <signal.h>
void handler()
{
    signal(SIGINT, handler);
    printf("handler\n");
    return;
}

void main()
{
    signal(SIGINT, handler); /* CTR-C handled */
    while (1) {
        printf("main\n");
        sleep(2);
    }
}
$

```

Anstelle eines selbst programmierten Handlers, kann man vordefinierte Handler verwenden. Sie werden durch die Konstante `SIG_IGN` (ignoriere Signal) und `SIG_DFL` (reagiere per Default Aktion) beim Systemaufruf „signal“ verwendet. So ist in der Implementierung der Shell vor dem „fork“ ein Systemaufruf zum ignorieren des Signals `SIGINT`, wenn der Prozess im Hintergrund gestartet werden soll (`signal(SIGINT, SIG_IGN);`).

In Unix gibt es das Kommando „**kill**“ zum Beenden von Prozessen, die im Hintergrund laufen. Wenn ein Programm so geschrieben ist, dass es alle Signale ignoriert, könnte es nie abgebrochen werden. Deshalb gibt es das Signal „SIGKILL“. Dieses Signal kann **nicht** per Signalhandler abgefangen werden.

```
$ ps
2864 pts/1      00:00:18 bash
4423 pts/1      00:00:01 signal
4525 pts/1      00:00:00 ps
$
$ kill -9 4423
killed signal
$
```

Im Bereich Echtzeitanwendungen muss ein Betriebssystem in der Lage sein, Prozesse nach einer gewissen Zeit zu informieren, dass bestimmte Dinge zu erledigen sind. In Kernkraftwerken muss die Temperatur des Reaktors regelmäßig überprüft werden. Deshalb wird dem Prozess, der die Überwachung bewerkstelligt regelmäßig ein Signal gesendet, wodurch er die Temperatur prüft. Der Systemaufruf „**alarm**“ hat einen Parameter, der die Anzahl Sekunden angibt, nach denen das Signal „SIGALARM“ erzeugt werden soll. Im Umfeld der Netzprogrammierung wird der Systemaufruf z.B. verwendet, um das ping Kommando zu realisieren. Dabei wird jede Sekunde ein Datenpaket zu einem Rechner gesendet und gewartet, ob es zurück gesendet wird.

Wenn Programme ablaufen, bei denen eine gewisse Zeit gewartet werden soll, kann dies programmtechnisch realisiert werden, indem man eine Schleife verwendet, die stets nichts tut, als testet, ob die Zeit schon um ist. Diese Lösung ist CPU intensiv. Besser wäre es, man könnte per Systemaufruf sagen, dass eine gewisse Zeit pausiert werden soll. Dazu ist der Systemaufruf

„pause“ verfügbar. Der Systemaufruf „**pause**“ veranlasst den aufrufenden Prozess zu warten, bis er das entsprechende Signal zum Weitermachen erhält.

Das folgende Programm „timer.c“ verwendet den Systemaufruf „alarm“, um einen Timer zu setzen. Wenn innerhalb von 5 Sekunden keine Benutzereingabe erfolgt, wird das Programm beendet.

```
$ cat timer.c
#include <stdio.h>
#include <signal.h>
void handler()
{
    printf("By\n");
    exit(0);
}

main()
{
    char str[80];
    signal(SIGALRM, handler);

    while (1) {
        alarm(5);          /* start timer */
        printf("> ");
        gets(str);
        printf("%s\n", str);
    }
}
$
```

Ein Aufruf von „timer“ bewirkt etwa:

```
$ timer  
> 12345  
12345          5 Sekunden warten  
> By  
$
```

4. Dateiverwaltung

Das nachfolgende Beispielprogramm „simpeTouch.c“ demonstriert den Systemaufruf „creat“. Das Programm legt die als Aufrufparameter anzugebende Datei mit den Zugriffsrechten „0640“ an.

```
#include <stdio.h>
main(int argc, char **argv)
{
    int fd;
    if (argc != 2) {
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    if ((fd=creat(argv[1],0640)) < 0) {
        fprintf(stderr, "create error\n");
        exit(2);
    }
}
```

Bevor eine Datei bearbeitet werden kann, muss sie geöffnet werden. Dazu existiert der Systemaufruf „open“, der neben dem Namen noch die Art des Zugriffs (Lesen, Schreiben oder beides) benötigt.

Das folgende Programm („cat.c“) liest die als Parameter anzugebende Datei und schreibt den Inhalt auf die Standardausgabe.

```

#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE 512
main(int argc, char **argv)
{
    int fd;
    int n;
    char buf[BUFSIZE];

    /* check usage */
    if (argc != 2) {
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    /* open file */
    if ((fd=open(argv[1], O_RDWR)) < 0) {
        fprintf(stderr, "open error\n");
        exit(2);
    }
    /* read and write file */
    while ((n = read(fd, buf, BUFSIZE)) > 0)
        write(1, buf, n);
}

```

Der **wahlfreie Zugriff** auf Dateien wird durch den Systemaufruf „lseek“ realisiert. lseek hat drei Parameter:

1. einen Filedescriptor, der die zu bearbeitende Datei definiert,
2. den Offset , der die Position des Lese/Schreibkopfes in Byte relativ zum Ausgangspunkt definiert und
3. den Ausgangspunkt (SEEK_SET=Dateianfang, SEEK_END=Dateiende, SEEK_CUR=aktuelle Position) für die Positionierung des Lese/Schreibkopfes

Damit kann man ein Programm („revCat.c“), das eine Datei von hinten nach vorne liest einfach schreiben:

```

#include <stdio.h>
#include <fcntl.h>
main(int argc, char **argv)
{
    int fd;
    int pos;
    char buf[1];
    if (argc != 2) { /* check usage */
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    if ((fd=open(argv[1], O_RDWR)) < 0) { /* open file */
        fprintf(stderr, "open error\n");
        exit(2);
    }
    /* set position for reading at end of file */
    if ((pos=lseek(fd, -1, SEEK_END)) == -1) {
        fprintf(stderr, "lssek error\n");
        exit(1);
    }
    while (pos>=0) { /* read and write file */
        read(fd, buf, 1);
        write(1, buf, 1);
        lseek(fd, --pos, SEEK_SET);
    }
    printf("\n");
}

```

Im folgenden Programm „hole.c“ wird der Systemaufruf „lseek“ verwendet, um ein Loch in einer Datei zu erzeugen.

```

#include <stdio.h>
int main()
{
    int fd;
    char buf1[] = "abcdefghijklmnop";
    char buf2[] = "ABCDEFGHIJKLMNOP";

    if ((fd=creat("file.hole",0640)) < 0) {
        fprintf(stderr, "create error\n");
        exit(1);
    }
    if ((write(fd, buf1, 16)) != 16) { /* offset now 16 */
        fprintf(stderr, "buf1 write error\n");
        exit(2);
    }
    if ((lseek(fd, 32, SEEK_SET)) == -1) { /* offset now 32 */
        fprintf(stderr, "lseek error\n");
        exit(3);
    }
    if ((write(fd, buf2, 16)) != 16) { /* offset now 48 */
        fprintf(stderr, "buf2 write error\n");
        exit(4);
    }
    exit(0);
}

```

Ein Aufruf von „hole“ bewirkt etwa:

```

$ ls -l fil*
-rw-r----- 1 as users 48 Nov 1 19:11 file.hole
$ od -c file.hole
0000000  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 Loch
0000040  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P
0000060
$

```

Spezialdateien werden durch den Systemaufruf „mknod“ erzeugt (nicht durch „create“). Ein neues Terminal wird z.B. durch das C-Fragment

```
fd = mknod(„/dev/tty18“, 020744, 0x0402);
```

erzeugt. Dabei ist der erste Parameter der Name des Device, der zweite die Zugriffsmaske (rwxr—r--). Der dritte Parameter definiert die Hauptgerätenummer (4) und die Nebengerätenummer (2).

Die Informationen über eine Datei, der so genannte Dateistatus kann durch Systemaufrufe „stat“, fstat“ und „lstat“ abgefragt werden. „stat“ und „fstat“ liefern Information über Dateien, „lstat“ über Links. Alle Aufrufe geben eine Struktur vom Typ „stat“ zurück, die folgendermaßen aufgebaut ist:

```

struct stat
{
    dev_t      st_dev;      /* Device */
    ino_t      st_ino;     /* INode */
    umode_t    st_mode;    /* Typ und Protection */
    nlink_t    st_nlink;   /* Anzahl d. Hard_Links */
    uid_t      st_uid;     /* UID des Besitzers */
    gid_t      st_gid;     /* GID des Besitzers */
    dev_t      st_rdev;    /* Typ (wenn INode-Device) */
    off_t      st_size;    /* Grösse in Bytes*/
    unsigned long st_blksize; /* Blockgrösse */
    unsigned long st_blocks; /* Allozierte Blocks */
    time_t     st_atime;   /* Letzter Zugriff */
    time_t     st_mtime;   /* Letzte Modifikation */
    time_t     st_ctime;   /* Letzte Aenderung */
};

```

Das folgende Programm („simpleFile.c“ verwendet diese Struktur, um Informationen über eine Datei auszugeben:

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
main(int argc, char **argv)
{
    int i;
    struct stat buf;
    char *ptr;
    /* check usage */
    if (argc != 2) {
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    if (lstat(argv[1], &buf) < 0) {
        fprintf(stderr, "lstat error\n");
        exit(2);
    }
    /* print file info */
    if (S_ISREG(buf.st_mode)) ptr = "regular";
    else if (S_ISDIR(buf.st_mode)) ptr = "directory";
    else if (S_ISCHR(buf.st_mode)) ptr = "character special";
    else if (S_ISBLK(buf.st_mode)) ptr = "block special";
    else if (S_ISFIFO(buf.st_mode)) ptr = "fifo";
    else if (S_ISLNK(buf.st_mode)) ptr = "link";
    else ptr = "unknown mode!";
    printf("%s: %s\n", argv[1], ptr);
}

```

Pipes sind ein Kommunikationsmedium, das es erlaubt, dass Prozesse in FIFO Manier kommunizieren. Eine Pipe ist dabei eine Pseudodatei, die die Kommunikationsdaten temporär beinhaltet. In Unix könne zwei Dateien sortiert in einer zusammengeführt werden durch folgende Kommandosfolge:

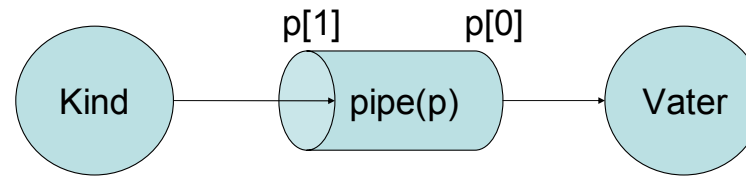
```
$ cat file2 file2 | sort
```

Hierbei werden zwei Prozesse erzeugt: der erste (cat) schreibt seine Ausgabe in die Pipe, der zweite (sort) liest die Standardeingabe aus der Pipe.

Der Systemaufruf „pipe“ erzeugt eine Pipe und gibt zwei Dateideskriptoren zurück, einen zum Lesen und einen zum Schreiben.

```
int p[2];  
pipe(p); /* p[1]=Schreibende, p[0]=Leseende */
```

Der Mechanismus wird nun am Beispiel eines Programmes gezeigt, bei dem ein Vaterprozess einem Kind Informationen über eine Pipe sendet, d.h. es wird folgende Kommunikation realisiert:



1. Sohn schließt Leseende
2. Sohn schreibt

1. Vater erzeugt pipe
2. Vater erzeugt Sohn
3. Vater schließt Schreibende
4. Vater liest

Als C-Programm sieht die Implementierung wie folgt aus:

```

$ cat pipe.c
/* Example: Child | Father */
#include <stdio.h>
#define BUFSIZE 20
main()
{ int pid, status;
  int p[2];
  char buf[BUFSIZE];
  if (pipe(p) != 0) {
      fprintf(stderr, "pipe error\n");
      exit(1);
  }
  switch (pid = fork()) {
      case -1: /* fork failed */
          fprintf(stderr, "cannot fork\n");
          exit(2);
      case 0: /* child: write into pipe */
          {int i;
            close(p[0]); /* close read end of pipe */
            /* create example data */
            for (i=getpid(); i>0; i--) {
                sprintf(buf, "%d\n", i);
                write(p[1], buf, BUFSIZE); /* write into pipe */
                sleep(2); /* just to see it with ps -el" */
            }
            close(p[1]);
            exit(0);
          }
  }
}

```

```

    }
default: /* father: read from pipe */
    {int length;
      close(p[1]); /* close write end of pipe */
      do {
          length=read(p[0], buf, BUFSIZE);
                                     /* read from pipe */
          fprintf(stdout, "%s", buf);
      } while (length>0);
      close(p[0]);
      while (wait(&status) !=pid);
      exit(0);
    }
} /* switch */
}
$

```

5. Katalogverwaltung

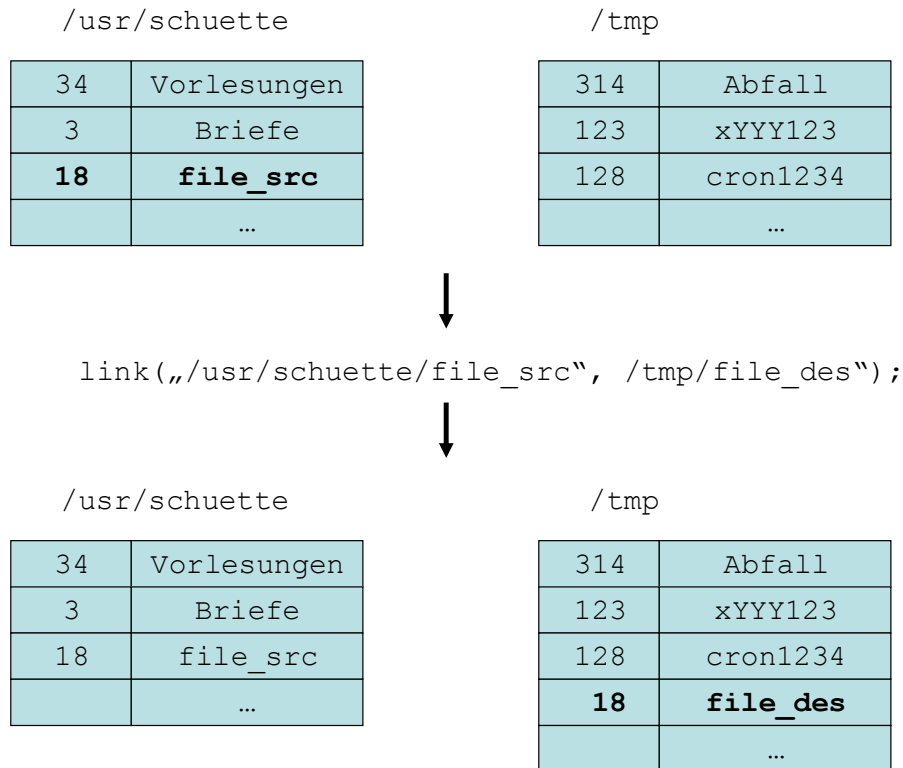
Hier werden Systemaufrufe vorgestellt, die auf Verzeichnisse wirken.

Durch den Systemaufruf „**link**“ kann eine physische Datei unter mehreren Namen, auch in unterschiedlichen Verzeichnissen erscheinen. Somit ist der Zugriff auf die Datei mit unterschiedlichen Namen möglich. Eine Änderung über den Zugriff mit einem Namen wirkt sich immer auf das reale Dateiojekt aus. Um dies zu realisieren, wird eine Datei in Unix eindeutig identifiziert durch ihre **i-Zahl** (*inumber*). Ein Verzeichnis ist eine Datei, die eine Anzahl von Paaren (*inumber*, Name im ASCII Code) enthält. Durch einen Link wird dann einfach ein neuer Eintrag im Verzeichnis erzeugt, der zur selben *inumber* einen zusätzlichen Namen einführt.

Der Systemaufruf

```
link(„/usr/schuette/file_src“, „/tmp/file_des);
```

bewirkt folgende Veränderung im tmp-Verzeichnis:



Durch den Systemaufruf „**unlink**“ kann man Links wieder Löschen. Dabei wird nur der Eintrag aus dem Verzeichnis entfernt; die Datei bleibt erhalten. Wenn es keinen weiteren Link auf die Datei gibt (das ist ja im i-Node abgespeichert), so wird die Datei physisch gelöscht.

Das Ein- und Aushängen von Verzeichnissen wird durch die Systemaufrufe „**mount**“ und „**umount**“ bewerkstelligt.

In Unix wird in einem Cache (Zwischenspeicher) werden die zuletzt benutzten Blöcke abgespeichert, um ein erneutes Zugreifen schneller zu machen, da dann nicht mehr auf die Festplatte

zugegriffen werden muss. Wenn ein Fehler auftaucht, bevor die Platte wirklich beschrieben (durch write Systemaufruf) wird, so gehen Daten verloren und ein inkonsistentes Dateisystem wäre die Folge. Deshalb wird periodisch vom Betriebssystem der „**sync**“ Systemaufruf ausgeführt; er schreibt die Blöcke im Cache auf die Platte. Beim Hochfahren eines Unix Systems wird ein Programm „update“ als Hintergrundprozess gestartet, der alle 30 Sekunden einen „sync“ Aufruf durchführt.

Während „sync“ den gesamten Cache synchronisiert, kann man mit dem Systemaufruf „fsync“ die Blöcke einer Datei im Cache synchronisieren. Dies wird von Datenbanksystemen, wie Oracle verwendet, um selbst die gepufferten Daten sicher auf die Platte zu schreiben, wenn ein z.B. „commit“ durchgeführt wird.

Dies ist im folgenden Programm („fsync.c“) gezeigt:

```

#include <stdio.h>
#include <fcntl.h>
main(int argc, char **argv)
{
    int fd;
    if (argc != 2) {
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    if ((fd=open(argv[1], O_RDWR)) < 0) {
        fprintf(stderr, "open error\n");
        exit(2);
    }

    /* manipulation of file
    ... */

    if (fsync(fd) < 0) {
        fprintf(stderr, "sync error\n");
        exit(3);
    }
    if (close(fd) < 0) {
        fprintf(stderr, "close error\n");
        exit(4);
    }
    exit(0);
}

```

Um vom aktuellen Verzeichnis in ein anderes Verzeichnis zu wechseln, existiert der Systemaufruf „**chdir**“. Nachdem das aktuelle Verzeichnis mit dem Systemaufruf

```
chdir („/usr/schuetzte/tmp“)
```

geändert wurde, werden die folgenden Aufrufe (create, open), bei denen kein voller Pfadname angegeben ist, immer im tmp-Verzeichnis Dateien anlegen, bzw. öffnen.

Ein Systemaufruf, mit dem das Root-Verzeichnis bestimmt werden kann ist „**chroot**“. Er wird verwendet, um etwa auf Web Servern virtuelle Root-Verzeichnisse für unterschiedliche Benutzergruppen definieren zu können.

6. Schutzmechanismen

In Linux besitzt eine Datei 12 Schutzbits in 4 Gruppen:

Ausführungsmodi	Benutzer	Gruppe	alle anderen
sgd	rwX	rwX	rwX
s=setuid g=setgid d=directory	r=read w=write x=execute		

Durch den Systemaufruf „chmod“ kann die Zugriffsmaske für eine Datei gesetzt werden. Die Maske wird dabei oktal angegeben. So bedeutet 0644, dass der Benutzer den Zugriff „6=110=rw-“ hat, die Gruppe „4=100=r-“.

Das folgende Programm (simpleChmod.c) setzt die Zugriffsrechte einer Datei (2.Parameter) so wie es die Maske (1. Parameter) angibt.

```

#include <stdio.h>
main(int argc, char **argv)
{
    int mask;
    if (argc != 3) {
        fprintf(stderr, "usage %s mask file\n", argv[0]);
        exit(1);
    }
    sscanf(argv[1], "%o", &mask);
    if ((chmod(argv[2], mask)) < 0) {
        fprintf(stderr, "chmod error\n");
        exit(2);
    }
}

```

Wenn ein Programm gestartet wird, so hat es die Rechte, die der Aufrufer des Programms hat. Deshalb kann ein Programm dem Superuser gehören (etwa das Programm `rm=remove file`), aber der Aufrufer kann nur die Dateien löschen, für die er die Rechte hat. Mit dem „**setuid**“ („setgid“) Bits kann man erreichen, dass ein Programm mit den Rechten des Besitzers (Gruppe) abläuft und nicht wie normal mit den Rechten des Aufrufers. Dies wird benötigt, um zum Beispiel die Passwort-Datei durch die Ausführung des Kommandos „`passwd`“ abzuändern. Dazu hat das Kommando (=Datei) das `setuid`-Bit gesetzt.

Der Systemaufruf „**umask**“ setzt eine Linux-interne Bitmaske, die die Schutzbits beim Anlegen einer Datei maskiert. Wird nach „umask(022) eine Datei angelegt, etwa durch „create(„date“, 0777) so hat die neue Datei nicht die Schutzbits 0777, sondern 0755:

file	111	111	111
umask	000	010	010
Resultat (file-umask)	111	101	101

Wird ein Programm mit dem setuid-Bit ausgestattet, so kann es nicht prüfen, ob der Aufrufer die Rechte auf eine Datei hat, da das Programm selbst ja durch das setuid-Bit alle Rechte hat. Um per Programm auf die Zugriffsrechte des realen Benutzers zugreifen zu können, existiert der Systemaufruf „**access**“.

Der erste Parameter von „access“ ist die zu prüfende Datei, der zweite der Zugriffsmodus, der geprüft werden soll.

Das folgende setuid-Programm prüft, ob eine Datei durch den Benutzer geöffnet werden könnte (access); danach wird die Datei geöffnet.

```
#include <stdio.h>
#include <fcntl.h>
main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "usage %s file\n", argv[0]);
        exit(1);
    }
    if (access(argv[1], R_OK) < 0) {
        fprintf(stderr, "access error\n"); exit(2);
    }
    if (open(argv[1], O_RDONLY) < 0) {
        fprintf(stderr, "open error\n"); exit(3);
    }
    exit (0);
}
```

7. Zeitverwaltung

In Betriebssystemen muss die Verwaltung von Datum und Uhrzeit durch Systemaufrufe unterstützt werden. Neben dem Abfragen der Systemzeit muss es Aufrufe für das Setzen der Zeit und für die Umstellung von Sommerzeit zu Winterzeit und umgekehrt geben.

In Unix ist Datum und Uhrzeit als Anzahl Sekunden, die seit dem „Unix Urknall“ (1.1.1970 00:00:00) vergangen sind, abgelegt. Alle Datumsangaben, werden so gespeichert und erst in der Anzeige in lesbare Form gebracht. Dazu existiert der Systemaufruf „time“, der die Sekundenanzahl liefert und Routinen, um diese Intergerzahl in ein lesbares Format zu konvertieren.

Das folgende C-Programm („now.c“) liefert das aktuelle Datum und die Uhrzeit.

```
#include <stdio.h>
#include <time.h>
main()
{
    time_t now;
    now = time(NULL); /* now as no. secs since ZERO */
    printf("%s\n", ctime(&now)); /* convert into readable form */
    exit(0);
}
```

```
$ now
```

```
Thu Nov 15 16:07:55 2001
```