

INHALT

Teil 1: Unix-Systemaufrufe.....	1
1. Grundlegende Begriffe	1
2 Einfache Datei-Eingabe und -Ausgabe	7
open Systemaufruf	7
close Systemaufruf	8
read Systemaufruf	8
write Systemaufruf	9
3 Signale	9
Signale und ihre Zuverlässigkeit	17
4 Pipes	19
pipe Systemaufruf	19
5 Prozeß-Steuerung	21
fork Systemaufruf	21
exit Systemaufruf	22
exec Systemaufruf	22
wait Systemaufruf	24
Teil 2: Aufgaben	1
1. Programmargumente und Ein/Ausgabe	1
2. Prozesse	2
3. Pipes	2

Teil 1: Unix-Systemaufrufe

1. Grundlegende Begriffe

Unix-Prozesse

Ein Prozeß ist eine Instanz eines Programmes, das vom Betriebssystem ausgeführt wird. Die einzige Möglichkeit unter Unix einen neuen Prozeß zu erzeugen, ist über die Systemfunktion `fork()`. Siehe auch Versuch 1: "Das Unix Betriebssystem".

Prozeß-ID

Jeder Prozeß enthält seine eigene Prozeß-Identität (process-ID), kurz *PID* genannt. Die PID ist eine Integergröße. Wird ein neuer Prozeß erstellt, so vergibt der Kernel die PID. Ein Prozeß kann auf die PID über den Systemaufruf `getpid()` zugreifen [1].

```
int getpid();
```

Unter **Digital Unix**TM ist der Prozeß mit der Prozeß-ID 1 der sogenannte *init-Prozeß*, die PID 0 ist dem Kernelprozeß zugewiesen.

Parent-Prozeß-ID

Zu jedem Prozeß gehört auch ein Parent Prozeß und eine zugehörige *Parent-Prozeß-ID*. Der Kernel weist die Parent-Prozeß-ID zu, wenn ein neuer Prozeß gestartet wird. Der neue Prozeß kann auf diesen Wert über den Systemaufruf `getppid()` zugreifen.

```
int getppid();
```

Das folgende C-Programm gibt die PID und die Parent-Prozeß-ID eines Prozesses aus.

```
main()
{
    printf("Prozeß-ID=%d, Parent-Prozeß-ID=%d\n", getpid(),
        getppid());
}
```

Reale User-ID

Jedem Benutzer wird eine positive ganzzahlige User-Identifikation (*User-ID* oder *UID*) zugewiesen. Ein Prozeß kann auf diesen Wert über den Systemaufruf `getuid` verfügen.

```
unsigned short getuid();
```

Die Datei `/etc/passwd` legt die Zuordnung zwischen Login-Namen und den numerischen User-ID-Werten fest.

Der numerische Wert der UID wird zum Beispiel im Dateisystem dazu verwendet, um zu registrieren, wer eine Datei erstellt hat.

Reale Gruppen-ID

Zusammen mit der User-ID wird jedem Benutzer eine positiv ganzzahlige *Gruppen-ID* vergeben. Ein Prozeß kann seine Gruppen-ID über den Systemaufruf `getgid` feststellen.

```
unsigned short getgid();
```

Die *GID* wird normalerweise dazu verwendet, um die User in projektbezogene Gruppen einzuteilen.

Die Datei `/etc/group` enthält die Zuordnung zwischen Gruppennamen und den numerischen Gruppen-ID's.

Jede Datei enthält auch die Gruppen-ID desjenigen, der die Datei erstellt hat.

Effektive User-ID

Jeder Prozeß besitzt auch noch eine *effektive User-ID*. Auf diesen Wert kann ein Prozeß über den Systemaufruf `geteuid` zugreifen.

```
unsigned short geteuid();
```

Normalerweise entspricht der `geteuid`-Wert der tatsächlichen User-ID. Jedoch kann für eine Programm-Datei ein spezielles Flag gesetzt werden, welches bewirkt, daß, wenn dieses Programm ausgeführt wird, die UID des Prozesses in die UID des Eigentümers dieser (Programm-)Datei geändert wird (*Set-User-ID-Programm*). Diese Eigenschaft ermöglicht den Anwendern besondere zusätzliche Zugriffsrechte während des Ablaufs des *Set-User-ID-Programms*.

Effektive Gruppen-ID

Jeder Prozeß besitzt auch eine *effektive Gruppen-ID*. Auf diesen Wert kann ein Prozeß über den Systemaufruf `getegid` zugreifen.

```
unsigned short getegid();
```

Es gilt das für die effektive UID gesagte sinngemäß.

Superuser

Die User-ID mit dem Wert 0 ist dem *Superuser* vorbehalten. Der Login-Name des Superusers ist üblicherweise *root*. Dem Superuser ist jeglicher Zugriff auf Dateien erlaubt, und er hat im Vergleich mit anderen Benutzern besondere Rechte.

Dateinamen

Jede Unix-Datei, jedes Verzeichnis oder jede spezielle Datei besitzt einen *Dateinamen*. Einige Unix-Systeme begrenzen diesen auf 14 Buchstaben. Die einzigen ASCII-Zeichen, die in einen Dateinamen *nicht* vorkommen dürfen sind das ASCII-NUL-Zeichen (`\0`) und der Schrägstrich (`/`).

Pfadnamen

Der *Pfadname* (pathname) ist ein durch das Nullbyte abgeschlossener String, der aus einem oder mehreren Dateinamen besteht. Die Dateinamen innerhalb eines Pfadnamens sind voneinander durch Schrägstriche (`/`) getrennt. Ein Pfadname kann optional mit dem Schrägstrich beginnen, womit festgelegt wird, daß der Pfad im Stammverzeichnis (root -> oberste Stufe der Dateihierarchie. Ein Pfad, der nicht mit einem Schrägstrich beginnt, wird als *relativer Pfad* bezeichnet, und dieser Pfad beginnt dann im aktuellen Verzeichnis.

Beispiele:

```
/                # root  
/usr/users_1/praktikum  # absoluter Pfad  
lib/math          # relativer Pfad
```

Passendes Bild?

Datei-Deskriptoren

Ein *Datei-Deskriptor* ist eine kleine Integergröße. Sie wird verwendet, um zu kennzeichnen, daß eine Datei zu Ein-/Ausgabezwecken geöffnet wurde.

Viele Unix-Programme, auch die Shells, verbinden die Datei-Deskriptoren 0, 1 und 2 mit dem Standardein-, Standardaus- und dem Standardausgabegerät für

Fehlermeldungen. Datei-Deskriptoren werden vom Kernel zugewiesen, wenn z.B. einer der folgenden Systemaufrufe erfolgreich war: `open`, `creat`, `pipe`.

Socket-Deskriptoren

4.3BSD verwendet auch für den Zugriff auf die Sockets kleine Integergrößen (siehe letzter Teil des Basispraktikums über das Netzwerk-API). Diese heißen *Socket-Deskriptoren* (`socket descriptors`). Bei 4.3BSD spricht man ganz allgemein von Deskriptoren, womit dann Datei-Deskriptoren und Socket-Deskriptoren gemeint sind. Nur wenn es unbedingt notwendig ist, wird der spezielle Deskriptortyp angegeben. Für manche Systemaufrufe ist ein Socket-Deskriptor zu einem Datei-Deskriptor equivalent. Aber es gibt einige 4.3BSD-Systemaufrufe, die nur mit Socket-Deskriptoren funktionieren, wie zum Beispiel `connect`. Socket-Deskriptoren werden vom Kernel zugewiesen, wenn z.B. einer der folgenden Systemaufrufe erfolgreich war: `accept`, `pipe`, `socket`.

Dateizugriffsrechte

Wie bereits erwähnt wurde, sind jedem UNIX-Prozeß 4 ID's zugewiesen:

- Reale User-ID
- Reale Gruppen-ID
- Effektive User-ID
- Effektive Gruppen-ID

Zusätzlich besitzt jede Datei die folgenden Attribute (es gibt noch weitere Attribute, sie sind aber hier uninteressant):

- User-ID des Eigentümers (16-Bit Integerwert)
- Gruppen-ID des Eigentümers (16-Bit Integerwert)
- Leseberechtigung des Users (ein 1-Bit Flag)
- Schreibberechtigung des Users (ein 1-Bit Flag)
- Ausführungsberechtigung des Users (ein 1-Bit Flag)
- Leseberechtigung der Gruppe (ein 1-Bit Flag)

- Schreibberechtigung der Gruppe (ein 1-Bit Flag)
- Ausführungsberechtigung der Gruppe (ein 1-Bit Flag)
- Lesezugriffsberechtigung für Jedermann (ein 1-Bit Flag)
- Schreibzugriffsberechtigung für Jedermann (ein 1-Bit Flag)
- Ausführungsberechtigung für Jedermann (ein 1-Bit Flag)
- Setze-User-ID (ein 1-Bit Flag)
- Setze-Gruppen-ID (ein 1-Bit Flag)

Der Kernel eines UNIX-Systems durchläuft die folgenden Schritte, um festzustellen, ob ein Prozeß auf eine Datei zugreifen darf. Dies ist der Fall, wenn eine der nachstehenden Bedingungen erfüllt ist.

1. Wenn die effektive User-ID des Prozesses 0 ist. Dann handelt es sich um den Superuser, und der Zugriff ist erlaubt.
2. Wenn die effektive User-ID des Prozesses mit derjenigen des Eigentümers der Datei übereinstimmt *und* wenn dieser das Bit für den erlaubten Zugriff auf diese Datei gesetzt hat, dann ist der Zugriff erlaubt. Der nach dem *und* durchgeführte Test bedeutet, daß das Erlaubnisbit für lesende Zugriffe eines Anwenders gesetzt sein muß, wenn der Prozeß die Datei lesen will. Wenn der Prozeß die Datei beschreiben will, so muß das Erlaubnisbit für schreibende Zugriffe eines Anwenders gesetzt sein. Will der Prozeß die Programmdatei ausführen, so muß das Bit für die Ausführungsberechtigung des Anwenders gesetzt sein.
3. Wenn die effektive User-ID des Prozesses nicht mit der User-ID des Eigentümers der Datei übereinstimmt und wenn die effektive Gruppen-ID des Prozesses mit derjenigen des Eigentümers übereinstimmt, und wenn eines der Gruppenzugriffsberechtigungsbits (Lesen, Schreiben oder Ausführen) gesetzt ist, so handelt es sich um einen erlaubten Zugriff.
4. Wenn die effektive User-ID des Prozesses nicht mit der User-ID des Dateieigentümers übereinstimmt, und wenn die effektive Gruppen-ID des Prozesses nicht mit der Gruppen-ID des Dateieigentümers übereinstimmt und wenn eines der 3 Zugriffserlaubnisbits (Lesen, Schreiben und Ausführen) für *andere* gesetzt ist, so handelt es sich um einen erlaubten Zugriff.

Zu beachten ist, daß nur wenn die User-ID's nicht übereinstimmen, die Übereinstimmung bei den Gruppen-ID's geprüft wird. Und nur wenn auch die Gruppen-ID's nicht übereinstimmen, wird das Zugriffsbit für andere geprüft.

Statuswort für Dateizugriffe

Die oben beschriebenen Bits haben in dem sogenannten Statuswort für Dateizugriffe (file access mode word) eine bestimmte Position. Die in den im Statuswort enthaltenen Bits verschlüsselte Information wird u.A. von den folgenden Systemaufrufen ausgewertet: `access`, `chmod`, `creat`, `open`, `semctl`, `shmctl`, `stat`, `fstat` und `unmask`. Im Bild x.x werden die Oktalwerte der einzelnen Bits des Statuswortes angegeben. (Oktalwerte bieten sich für die Statusangabe an, da alle Felder jeweils 3 Bit belegen.)

oktaler Wert	Bedeutung
04000	Setze User-ID bei der Ausführung
02000	Setze Gruppen-ID bei der Ausführung
01000	Speichere Textdaten nach der Erstausführung ("Sticky-Bit")
00400	Lesbar von User
00200	Beschreibbar von User
00100	Ausführbar von User
00040	Lesbar von der Gruppe
00020	Beschreibbar von Gruppe
00010	Ausführbar von der Gruppe
00004	Lesbar von Jedermann
00002	Beschreibbar von Jedermann
00001	Ausführbar von Jedermann

Bild x.x: Die Bedeutung der Bits im Statuswort Dateizugriffe.

Man beachte, daß man die Userzugriffsbits vom Zugriffsstatus durch die Maske 0700 erhalten kann. Die Gruppenzugriffsbits erhält man mit der Maske 070 und die Zugriffsbits für den Rest mit der Maske 07.

Maske für den Dateistatus

Jedem Prozeß ist eine Maske für den Dateistatus (file mode creation mask) zugeordnet. Sie wird mit dem Systemaufruf `umask` gesetzt.

```
int umask (int cmask);
```

Die unteren 9 Bits von `cmask` geben die Maske für den Dateistatus des Prozesses an. Diese 9 Bits stehen dabei für die Zugriffsbits des Users, der Gruppe und für die anderen Anwender. Dabei wird das bereits beschriebene Verfahren zugrundegelegt. Dieser Systemaufruf liefert stets die letzte dem Prozeß zugeordnete Maske zum Aufbau des Dateistatus zurück.

Die Dateistatusaufbaumaske wird dann benötigt, wenn ein neues Verzeichnis oder eine neue Datei erstellt wird. Durch die Maske wird festgelegt, welche Statusbits der neuen Datei *gelöscht* werden müssen. Wenn zum Beispiel eine neue Datei mit dem Status von oktal 0664 (schreib- und lesbar von User und Gruppe und von anderen Usern nur lesbar) erstellt werden soll und die Maske zum Aufbau des Dateistatus ist oktal 022, so wird das Gruppen-Lese-Bit gelöscht, und die Datei erhält den neuen Status von oktal 0644.

2 Einfache Datei-Eingabe und -Ausgabe

Unter Unix stehen 2 Ein- und Ausgabenverfahren zur Verfügung:

- die Unix-Systemaufrufe für Ein- und Ausgaben,
- die Standard-Ein-/Ausgabebibliothek

Die Unix Systemaufrufe für Ein-Ausgabe (`open`, `read`, `write`, `close` usw...) werden vom Kernel direkt bearbeitet. Andererseits stellt die Standard-Ein-/Ausgabebibliothek eine Sammlung von Funktionen dar, die eine höherwertige Schnittstelle (Pufferung, formatierte Ausgabe usw...) zwischen einem Prozeß und dem Kernel darstellen.

open Systemaufruf

Eine Datei wird geöffnet durch:

```
include <fcntl.h>
int open(char *Pfadnamen, int openAnzeige [, int Modus]);
```

In den früheren Unix-Versionen gab es keinen optionalen dritten Parameter und die `openAnzeige` war als binäre Konstante angegeben:

1. => geöffnet zum Lesen (01₂)
2. => geöffnet zum Schreiben (10₂)

3. => geöffnet zum Lesen und Schreiben (112)

Die neueren Unix-Versionen erlauben die Zuweisung symbolischer Werte an openAnzeige. Diese können durch ODER-Verknüpfung miteinander kombiniert werden.

- O_RDONLY Geöffnet zum Lesen
- O_WRONLY Geöffnet zum Schreiben
- O_RDWR Geöffnet zum Lesen und Schreiben
- O_NDELAY Erlaube das Öffnen, Lesen und Schreiben
- O_APPEND Schreibende Zugriffe am Dateiende anfügen
- O_CREAT Erstelle die Datei, wenn sie nicht existiert
- O_TRUNC Existiert die Datei, so setze die Größe auf 0
- O_EXCL Fehler bei O_CREAT, da die Datei existiert

close Systemaufruf

Eine geöffnete Datei wird wie folgt geschlossen:

```
int close(int DateiDeskriptor);
```

Wird ein Prozeß beendet, so werden vom Kernel alle noch offene Dateien automatisch geschlossen.

read Systemaufruf

Aus einer geöffneten Datei werden Daten wie folgt gelesen:

```
int read(int DateiDeskriptor,  
        char *Puffer,  
        unsigned int nZeichen);
```

Wenn read erfolgreich ausgeführt wurde, dann wird als (Funktions-)Ergebnis die Anzahl der gelesenen Zeichen übermittelt. Die Anzahl kann dabei geringer sein als die angeforderte Anzahl von nZeichen. Wird das Dateiende erreicht, so wird der Wert 0 zurückgeliefert. Tritt ein Fehler auf, wird der Wert -1 übermittelt.

`write` Systemaufruf

In eine geöffnete Datei werden Daten wie folgt geschrieben:

```
int write(int DateiDeskriptor,  
         char *Puffer,  
         unsigned int nZeichen );
```

Dieser Systemaufruf liefert die Anzahl der in die Datei geschriebenen Bytes zurück. Normalerweise entspricht dieser Wert dem Wert des Argument `nbytes`. Tritt ein Fehler auf, so wird der Wert `-1` zurückgeliefert.

Beim Schreiben von Netzanwendungen muß man darauf achten, daß von `read` und `write` Werte zurückgeliefert werden können, die kleiner sind als der durch `nZeichen` angegebenen Wert. Dies deutet nicht auf einen Fehler hin, sondern es kann durch die im Netz aufgetretene Pufferung der Daten auftreten.

3 Signale

Ein Signal ist die Mitteilung an einen Prozeß, daß ein bestimmtes Ereignis eingetreten ist. Man bezeichnet Signale manchmal auch als "Software-Interrupts". Signale treten üblicherweise asynchron auf. Das bedeutet, daß ein Prozeß nicht die geringste Ahnung hat, wann ein Signal auftreten wird. Signale können ausgetauscht werden zwischen:

- Einem Prozeß und einem zweiten Prozeß (oder dem ersten),
- Zwischen dem Kernel und einem Prozeß.

ACHTUNG: Die folgenden Abschnitte zum Thema Signale können die Teilnehmer am Basispraktikum ignorieren.

3.1 Signalnamen

Jedes Signal hat einen Namen. Die Namen sind in der Headerdatei `<signal.h>` festgelegt. In Tabelle 3.1 wird ein Überblick über einige Namen, die Bedeutung und die Standardaktionen der Signale gegeben. Die Spalte für die Bemerkungen gibt an, ob das betreffende Signal nur bei BSD oder System V verfügbar ist. Fehlt ein entsprechender Hinweis in dieser Spalte, so ist das Signal bei beiden Systemen verfügbar.

Name	Bemerkung	Beschreibung	Aktion
SIGALRM		Alarmzeit	Beenden
SIGBUS		Busfehler	Beenden mit Speicherabbild
SIGCLD		Ende eines Child-Prozesses	Verwerfen
SIGCONT	4.3 BSD	Fortsetzung nach SIGTSTP	Verwerfen
SIGFPE		FPE-Befehl	Beenden mit Speicherabbild
SIGHUP		Hangup	Beenden
SIGINT		Interrupt	Beenden
SIGKILL		Beenden	Beenden
SIGQUIT		Beendigungszeichen	Beenden mit Speicherabbild
SIGSEGV		Segmentverletzung	Beenden mit Speicherabbild
SIGSTOP	4.3 BSD	Stopp	Prozeß stoppen
SIGTERM		Software Beendigungssignal	Beenden
SIGTSTP	4.3 BSD	Stopp-Signal von der Tastatur	Prozeß stoppen
SIGURG	4.3 BSD	Außergewöhnliches Ereignis an einem Socket	Verwerfen
SIGUSR1		Anwenderdefiniertes Signal 1	Beenden
SIGUSR2		Anwenderdefiniertes Signal 2	Beenden

Tabelle 3.1 Einige Unix-Signale

3.2 Signalerzeugung

Wann treten Signale auf, und wie werden sie erzeugt? Es gibt Bedingungen, die erfüllt sein müssen, wenn ein Signal erzeugt werden soll.

1. Der *Systemaufruf* `kill`¹ ermöglicht es einem Prozeß, ein Signal zu senden. Dieses Signal kann zu einem anderen Prozeß oder zu dem Prozeß selbst gesendet werden.

```
int kill (int pid, int sig);
```

Ein Prozeß ist nicht in der Lage, einfach an *irgendeinen anderen Prozeß* ein Signal zu senden. Um an einen Prozeß ein Signal zu senden, muß der sendende Prozeß und der das Signal empfangende Prozeß dieselbe effektive User-ID besitzen. Die einzige Ausnahme ist, wenn der sendende Prozeß der Superuser ist.

Der Systemaufruf `kill` bewirkt in Abhängigkeit der Werte seiner Parameter viele verschiedene Dinge.

- Wenn der Parameter `pid` 0 ist, dann wird das Signal an alle Prozesse, die der Prozeßgruppe des Senders angehören, gesendet.
- Wenn der Parameter `pid` den Wert -1 besitzt, so wird, nur falls der Sender nicht der Superuser ist, wie folgt verfahren: das Signal wird an alle Prozesse gesendet, deren reale User-ID mit der effektiven User-ID des Senders übereinstimmt.
- Wenn der Parameter `pid` den Wert -1 besitzt und der Sender ist der Superuser, so wird das Signal an alle anderen Prozesse gesendet, mit Ausnahme der Systemprozesse. (Diese haben normalerweise die PID's 0 und 1.)
- Ist der Wert von `pid` negativ, aber nicht -1, so wird das Signal an alle Prozesse übertragen, deren Prozeßgruppen-ID mit dem absoluten Wert von `pid` übereinstimmt.
- Ist der Wert des Parameters `sig` 0, so wird kein Signal gesendet, aber es erfolgt eine Fehlerüberprüfung. Dieses kann zur Überprüfung der Gültigkeit von `pid` dienen. Damit das funktioniert, müssen alle Signale, die im Bild 3.1 aufgeführt sind, einen Wert größer oder gleich 1 haben.

¹Der Name `kill` ist etwas unglücklich gewählt, da ein Signal selten einen Prozeß beendet. Aber es gibt doch Signale, die einen Prozeß abbrechen, während andere Signale dazu dienen, einem Prozeß eine Nachricht zu überbringen, auf die dieser Prozeß dann reagieren soll.

-
2. Auch das *kill-Kommando* (siehe Versuch 1) von UNIX dient zum Senden von Signalen. Dieses Kommando nimmt seine Kommandozeilenparameter auf und führt einen Systemaufruf `kill` durch. Ab diesem Zeitpunkt gelten alle oben genannten Punkte.
 3. Bestimmte Terminalzeichen erzeugen Signale. Zum Beispiel ist jedem interaktiven Terminal ein Interrupt- und ein Quitzeichen zugewiesen. Das Interruptzeichen (üblich sind Delete oder Control-C) beendet einen ablaufenden Prozeß. Dabei wird das SIGINT-Signal erzeugt. Das Quitzeichen (meist Control-\) beendet einen Prozeß. Dann wird ein Speicherabbild (core image) dieses Prozesses erzeugt und das SIGQUIT-Signal generiert. Das Speicherabbild des Prozesses kann dann von einem Debugger verwendet werden, um ein nachträgliches Fehlersuchen zu ermöglichen. Als Interruptzeichen und als Quitzeichen kann fast jedes gewünschte Terminalzeichen gewählt werden.

Diese Signale, die von einem Terminal erzeugt werden, werden nicht nur zu dem ablaufenden Prozeß gesendet, sondern zu allen Prozessen der Kontrollgruppe des Terminals. Und diese Terminal-Signale werden normalerweise vom Kernel erzeugt und an den Prozeß gesendet.

4. Signale werden auch durch verschiedene Hardwareereignisse erzeugt. So wird zum Beispiel durch einen Fehler bei der Verarbeitung von Floating-Point-Werten ein SIGFPE-Fehlersignal erzeugt. Oder der Zugriff auf eine Adresse außerhalb des Adreßbereiches eines Prozesses erzeugt das SIGSEGV-Signal. Die besonderen Hardwarezustände, die zum Erzeugen von bestimmten Signalen führen, sind von der jeweiligen UNIX-Implementation abhängig. Solche Signale werden normalerweise vom Kernel an einen Prozeß gesendet.
5. Der Kernel erkennt bestimmte Softwarebedingungen und kann daraufhin bestimmte Signale erzeugen. So wird zum Beispiel bei einem Socket, wenn ankommende Daten die Bandgrenze überschreiten, das SIGURG-Signal erzeugt.

3.3 Signalbehandlung

Nachdem die Signale und ihre Entstehung verdeutlicht worden sind, bleibt noch die Frage offen, wie ein Prozeß ein Signal behandelt.

1. Der Prozeß kann eine bestimmte Funktion aufrufen. Diese Funktion soll immer dann aufgerufen werden, wenn ein bestimmtes Signal auftritt. Man spricht deshaß

von einem sogenannten Signalbehandlungsfunktion (signal handler). Der Prozeß kann bestimmen, auf welche Art die Signalbehandlungsfunktion auf das Signal reagiert. Dies ist das sogenannte Auffangen eines Signales (signal-catching).

2. Ein Prozeß kann ein Signal auch ignorieren. Dies gilt für alle Signale, mit Ausnahme des SIGKILL-Signals. Das SIGKILL-Signal darf nicht ignoriert werden, da der Systemadministrator damit jeden Prozeß zu jedem Zeitpunkt abbrechen können muß.
3. Ein Prozeß kann auf ein Signal mit der voreingestellten Aktion (default action) reagieren. Dies entspricht der Spalte "Aktion" in Tabelle 3.1 . Normalerweise wird ein Prozeß beim Empfang eines Signales abgebrochen, wobei bei einigen Signalen noch ein Speicherabbild des Prozesses erstellt wird (im aktuellen Arbeitsverzeichnis). Unter 4.3BSD jedoch ist Ignorieren die voreingestellte Reaktion auf die Signale SIGURG, SIGIO, SIGCONT und SIGWINCH. Außerdem ist bei 4.3BSD die Vorgabeaktivität, bei Auftreten der Signale SIGSTOP, SIGTSTP, SIGTTIN und SIGTTOU den Prozeß zu beenden.

Ein Prozeß kann die Art und Weise, wie er auf ein Signal reagieren will, mit dem Systemaufruf `signal` festlegen.

```
#include <signal.h>

int (*signal (int sig, void (*func) (int))) (int);
```

Diese Deklaration bewirkt, daß `signal` eine Funktion ist, die einen Zeiger auf eine Funktion liefert, die vom Ergebnistyp Integer ist. Der Parameter `func` gibt die Adresse einer Funktion an, die kein Ergebnis zurückliefert, d.h. sie ist vom Typ `void`. Es gibt 2 Spezialwerte für `func`, `SIG_DFL` und `SIG_IGN`. `SIG_DFL` bewirkt, daß auf das Signal mit der Vorgabereaktion reagiert werden soll, und `SIG_IGN` bewirkt, daß es ignoriert werden soll. Der Systemaufruf `signal` liefert stets den vorhergehenden Wert von `func` für das angegebene Signal zurück.

Um zum Beispiel zu bewirken, daß das Signal `SIGUSR1` ignoriert wird, ist wie folgt zu verfahren:

```
signal(SIGUSR1, SIG_IGN);
```

Soll beim Auftreten des Signales `SIGINT` die Funktion `myintr` nur aufgerufen werden, wenn dieses Signal zur Zeit nicht ignoriert wird, so ist wie folgt zu verfahren:

```
include<signal.h>
extern void    myintr(); /*Signal-Handler des Benutzers */
...
if signal(SIGINT, SIG_IGN) != SIG_IGN)
signal(SIGINT, myintr);
```

Wird eine Funktion aufgerufen, um auf ein Signal zu reagieren, so wird als Parameter die Signalnummer an die Funktion übergeben. Die Signalnummer ist eine Integergröße, und sie ist der erste übergebene Parameter. Dadurch ist es einer Funktion möglich, auf verschiedene Signale zu reagieren. Sie kann dann zur Laufzeit feststellen, um welches Signal es sich jetzt handelt. Einige der durch Hardwareereignisse erzeugten Signale übergeben zusätzliche Parameter an die Signalbehandlungsfunktion. Darauf soll hier aber nicht eingegangen werden. Ist die Signalbehandlungsfunktion abgearbeitet, so wird der aufgerufene Prozeß an der Stelle fortgesetzt, an der er durch das Signal unterbrochen wurde.

3.4 Signalbeschreibung

Im folgenden werden die Signale der Tabelle 3.1 besprochen und auf einige Besonderheiten näher eingegangen. Einige der Signale dienen zur Anpassung abweichender Hardwareeigenschaften.

SIGALRM Ein Prozeß kann einer bestimmten Uhrzeit eine Alarmwirkung zuweisen. Dazu dient der Systemaufruf `alarm`.

```
unsigned int alarm(unsigned int sec);
```

Der Parameter `sec` gibt die Anzahl der Sekunden an, die vergehen müssen, bis der Kernel dem Prozeß das SIGALRM-Signal sendet. Der Parameter bezieht sich dabei auf die "tatsächliche Zeit" (real time) und nicht auf die abgelaufene Rechnerzeit (CPU time). Ist der Wert von `sec` 0, so wird eine zuvor zugeordnete Alarmzeit wieder auf 0 zurückgesetzt. Dieser Systemaufruf wird verwendet, um die Bedingungen für Software-Timeouts zu definieren. Der Funktionswert der von `alarm` zurückgeliefert wird, ist die bis zum nächsten Alarm ggf. noch verbleibende Zeit, gemessen seit dem letzten Aufruf der Funktion.

In den folgenden Kapiteln wird auch die Funktion `sleep` verwendet. Mit dieser legt man eine Pause für die angegebene Zahl von Sekunden fest.

```
unsigned int sleep (unsigned int sec);
```

Die Funktion `sleep` benutzt üblicherweise das `SIGALRM`-Signal, und sie fangt dieses Signal mit einer Signalbehandlungsfunktion auf. Es gibt mehrere Möglichkeiten, wie die `sleep`-Funktion mit dem `SIGALRM`-Signal kommuniziert, um festzustellen, daß der Prozeß bereits installiert ist. Darauf soll hier aber nicht näher eingegangen werden. Die 4.3BSD-Version dieser Funktion liefert keinen Funktionswert zurück, während die Variante des System V die Anzahl der nicht im Sleep-Zustand verbrachten Sekunden zurückliefert. Dies tritt dann ein, wenn die "Schlafpause" vorzeitig beendet wird, wenn der Prozeß ein anderes Signal mit einer Signalbehandlungsfunktion empfangen hat.

- SIGBUS** Abhängig von der Systemimplementation tritt dieses Signal bei einem Hardwarefehler auf.
- SIGCLD** Dieses Signal wird an einen Parent-Prozeß gesendet, wenn ein Child-Prozeß beendet wird. Im Gegensatz zu den meisten im Bild 3.1 aufgeführten Signale bleibt dieses Signal unberücksichtigt, wenn der Prozeß dieses Signal nicht mit einer Signalbehandlungsfunktion auffängt. Unter dem 4.3BSD-System gibt dieses Signal auch an, daß sich der Status eines Child-Prozesses geändert hat. Die 4.3BSD-Version ist damit allgemeiner gehalten. Die System V-Version bezieht sich dagegen nur auf das Ende eines Child-Prozesses. Eine Statusänderung eines Child-Prozesses kann die Beendigung desselben sein. Es kann aber auch der Abbruch des Child Prozesses sein durch eines der folgenden Signale: `SIGSTOP`, `SIGTTIN`, `SIGTTOU`, oder `SIGTSTP`.
- SIGCONT** (Nur 4.3BSD) Wird unter 4.3BSD ein Prozeß angehalten (siehe die unten beschriebenen Signale `SIGSTOP` und `SIGTSTP`) und wieder fortgesetzt, wird dieses Signal gesendet. Wird zum Beispiel ein Editor kann er daraufhin seinen Bildschirm wieder neu aufbauen, wenn er fortgesetzt wird.
- SIGFPE** Abhängig von der Systemimplementation wird dieses Signal bei einem bestimmten Hardwareereignis erzeugt. Es wird zum Beispiel auf der VAX unter 4.3BSD verwendet, um bestimmte Ereignisse bei der Verarbeitung von Gleitkommazahlen anzuzeigen (z.B. Gleitkomma-

zahlenbereichsunterschreitung) oder bestimmte Bedingungen bei der Verarbeitung von Integergrößen (z.B. Division durch 0).

- SIGHUP** Wird ein Terminal abgemeldet, so wird an jeden Prozeß, für den es das Kontrollterminal war, das Signal hangup gesendet. Genauso wird dieses Signal an alle Mitglieder einer Prozeßgruppe gesendet, wenn der Gruppenleiter der Prozeßgruppe beendet wird.
- SIGINT** Dieses Interrupt-Signal wird für gewöhnlich erzeugt, wenn man am Terminal den Interruptschlüssel eingibt.
- SIGKILL** Dieses Signal ist die einzig sichere Art, einen Prozeß zu beenden. Es ist dem dieses Signal empfangenden Prozeß nämlich nicht möglich, dieses Signal zu ignorieren oder Signal-catching auf es anzuwenden.
- SIGQUIT** Dieses Signal wird üblicherweise beim Drücken der Quit-Taste am Terminal erzeugt. Dies ist ähnlich zum SIGINT-Signal, es wird aber zusätzlich ein Speicherauszug erzeugt.
- SIGSEGV** In Abhängigkeit von der Systemimplementation tritt dieses Signal bei einem Hardwarefehler auf. Eine Verletzung der Segmentbestimmungen erfolgt zum Beispiel, wenn ein Prozeß auf eine Adresse zugreifen will, auf die er nicht zugreifen darf.
- SIGSTOP** (Nur 4.3BSD) Durch dieses Signal wird ein Prozeß unterbrochen. Wie beim SIGKILL-Signal, so ist es auch hier nicht möglich, das Signal zu ignorieren oder es durch eine Signalbehandlungsfunktion aufzufangen. Dadurch hat der Systemadministrator eine Möglichkeit, einen Prozeß auf diesem Weg zu unterbrechen. Ein unterbrochener Prozeß kann mit dem SIGCONT-Signal fortgesetzt werden.
- SIGTERM** Signal für Abbruch durch Software. Dieses Signal wird als Vorgabe verwendet, wenn ein kill-Befehl ausgeführt wird. (Dabei darf man den kill-Befehl nicht mit dem gleichnamigen Systemaufruf kill verwechseln.)
- SIGTSTP** (Nur 4.3BSD) Wird die direkte (meist Control-Z) oder die verzögerte (meist Control-Y) Abbruchtaste (suspend key) auf der Tastatur gedrückt, dann wird dieses Signal an den Prozeß gesendet. Ein Prozeß, der unterbrochen wurde, kann mit dem SIGCONT-Signal fortgesetzt werden.

-
- SIGURG** (Nur 4.3BSD) Ein außergewöhnliches Ereignis ist eingetreten (wie z.B. Out-of-Band-Daten). Dieses Signal und das Konzept der Out-of-Band Daten werden im 3. Versuch beschrieben.
- SIGUSR1** Es gibt 2 vom User definierbare Signale. Diese können zur Kommunikation zwischen 2 Prozessen dienen. Dabei ist folgendes zu beachten: Dient ein Signal der Kommunikation zwischen 2 oder mehreren Prozessen, kann der Empfangsprozess nicht auf die Prozeß-ID des Senderprozesses zugreifen. Die einzige übertragene Information, die dabei mit übertragen wird, ist der Signaltyp. Der Empfangsprozess kann also nur sie auswerten und keine weiteren Informationen mit dem Signal erhalten. Durch diese Einschränkungen werden diese Signale nicht sehr oft zur Kommunikation zwischen Prozessen verwendet.

3.5 Signale und ihre Zuverlässigkeit

Signale waren in früheren UNIX-Versionen unzuverlässig. Damit ist gemeint, daß bestimmte Bedingungen existierten, unter denen *der Verlust von Signalen* möglich war. Es konnte ein Ereignis auftreten, welches ein Signal erzeugte, aber das Signal erreichte den zuständigen Prozeß nicht. Inzwischen wurden bei 4.3BSD und bei System V Änderungen vorgenommen, um die Zuverlässigkeit der Signale sicherzustellen. (Leider sind die von Berkeley und AT&T gemachten Änderungen aber nicht kompatibel.) Um zuverlässigere Signale zu erhalten, wurden die folgenden Änderungen getroffen:

- Signalbehandlungsfunktionen bleiben installiert, auch *nachdem* das Signal aufgetreten ist. Die früheren UNIX-Versionen setzten die mit dem Signal verbundene Aktion auf SIG_DFL zurück, bevor die vom Anwender installierte Signalbehandlungsfunktion überhaupt aufgerufen wurde. Dies hatte zur Folge, daß jedes Auftreten dieses Signales, bevor die Signalbehandlungsfunktion des Anwenders den Systemaufruf `signal` nochmals durchführen kann, verloren gehen würde.
- Ein Prozeß muß in der Lage sein, das Auftreten bestimmter Signale zu verhindern. Aber man will keine weitere Ignoriermöglichkeit schaffen. (Dies kann mit dem SIG_IGN-Aufruf geschehen.) Stattdessen soll das Signal notiert werden, um es, wenn der Prozeß dazu bereit ist, erneut zu übermitteln. Unter 4.3BSD wird dies als Blockieren (blocking) und unter System V als Halten (holding) von Signalen bezeichnet.
- Währenddem ein Signal an einen Prozeß übermittelt wird, ist es blockiert bzw. wird es gehalten. Damit ist gemeint, daß, wenn das Signal ein zweites Mal auftritt,

währenddem der Prozeß auf das erste Auftreten reagiert, die Signalbehandlungsfunktion kein zweites Mal aufgerufen wird. Stattdessen wird er nur dann aufgerufen, wenn er die Reaktion auf das erste Auftreten des Signales auf normale Art (ohne Fehler) beendet hat.

Das Prinzip der *maskierbaren* Signale wird nur von 4.3BSD unterstützt. Die Maske ist eine Integergröße, die einen oder mehrere Signalwerte angibt. Der Makro `sigmask`, der in der Headerdatei `<signal.h>` definiert ist, erzeugt die Signalmasken. Wenn zum Beispiel eine Maske für die Signale `SIGQUIT` **und** `SIGINT` erzeugt werden soll, so kann man schreiben:

```
int mask;  
mask = sigmask(SIGQUIT) | sigmask(SIGINT);
```

Die 4.3BSD-Version dieses Aufrufes verwendet für jedes Signal ein Bit eines 32-Bit Integerwertes. (Dadurch ist die Anzahl der Signale auf 32 begrenzt.) Um eines oder auch mehrere Signale zu blockieren, verwendet man den Systemaufruf `sigblock`.

```
int sigblock(int mask);
```

Die durch den Wert von `mask` angegebenen Signale werden zusätzlich zu den bereits blockierten Signalen blockiert. Der Funktionswert, der zurückgeliefert wird, ist die vor dem Aufruf aktive Maske. Die Blockierung eines Signales wird aufgehoben mit folgen dem Aufruf:

```
int sigsetmask(int mask);
```

Hierbei ist der Wert von `mask` so zu wählen, daß das Signal, welches nicht mehr blockiert werden soll, *nicht* maskiert wird. Oft gibt es in einem Programm kritische Abschnitte, die nicht durch das Auftreten bestimmter Signale unterbrochen werden sollen. Angenommen, die Signale `SIGQUIT` und `SIGINT` sollen blockiert werden. Dies läßt sich mit folgendem Programmstück erreichen:

```
int oldmask;  
oldmask = sigblock(sigmask(SIGQUIT) | sigmask(SIGINT));  
/* critical region */  
sigsetmask(oldmask); /* reset the mask to what it was */
```

4 Pipes

pipe Systemaufruf

Ein Pipe definiert eine Einbahnstraße für Daten. Über sie können Programme auf einfache Art Daten austauschen. Beispiel:

```
who | sort
```

das Shell-Kommando läßt die Shell zwei Prozesse erzeugen und eine Pipe, das `who` Programm schreibt seine Ausgabe in die Pipe, die anschließend vom Programm `sort` ausgelesen wird.

Der entsprechende pipe-Systemaufruf hat folgende Syntax:

```
int pipe(int *FileDes);
```

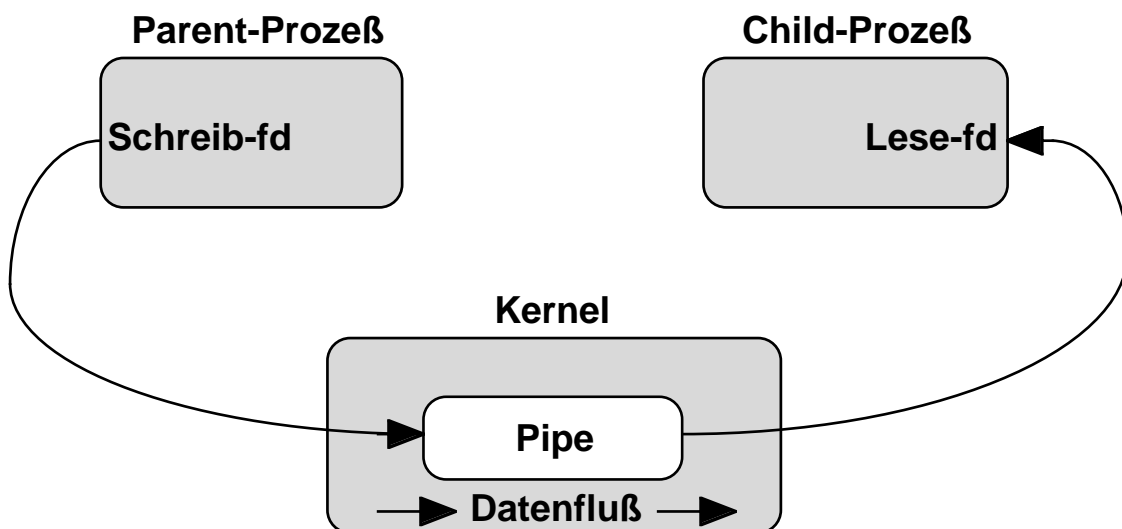


Bild 1.1 Eine Pipe zwischen 2 Prozessen

Dabei können als Ergebnis **2** Datei-Deskriptoren zurückgeliefert werden. `FileDes[0]` wird geöffnet zum Lesen und `FileDes[1]` wird geöffnet zum Schreiben. Geschrieben und gelesen wird mit den einfachen Systemaufrufen `read` und `write`.

Um nun einen Datenaustausch zwischen zwei Prozessen über Pipes ablaufen zu lassen, müssen beide einen gemeinsamen Parent-Prozeß aufweisen. Denn nur über den `fork`-Systemaufruf können die Datei-Deskriptoren der Pipe weitergegeben werden. Zuerst

erzeugt der Parent die Pipe und verwendet dann den fork-Aufruf, um von sich selbst eine Kopie zu erzeugen. Als nächstes schließt der Parent den Pipe-Ausgang, über den er von der Pipe liest. Der Child-Prozeß schließt den Eingang, über den er in die Pipe schreibt.

Man beachte, daß Pipes nur einen Datenfluß in einer Richtung ermöglichen, sie sind also unidirektional. Wenn man einen bidirektionalen Datenfluß wünscht, so muß man 2 Pipes erzeugen. Dann ist jede Pipe für eine Richtung zuständig. Die hierbei auszuführende Schritte sind:

- Erzeuge Pipe1 und Pipe2,
- fork-Aufruf,
- der Parent-Prozeß schließt den Leseausgang von **Pipe1**,
- der Parent-Prozeß schließt den Schreibeingang von **Pipe2**,
- der Child-Prozeß schließt den Leseausgang von **Pipe2**,
- der Child-Prozeß schließt den Schreibeingang von **Pipe1**.

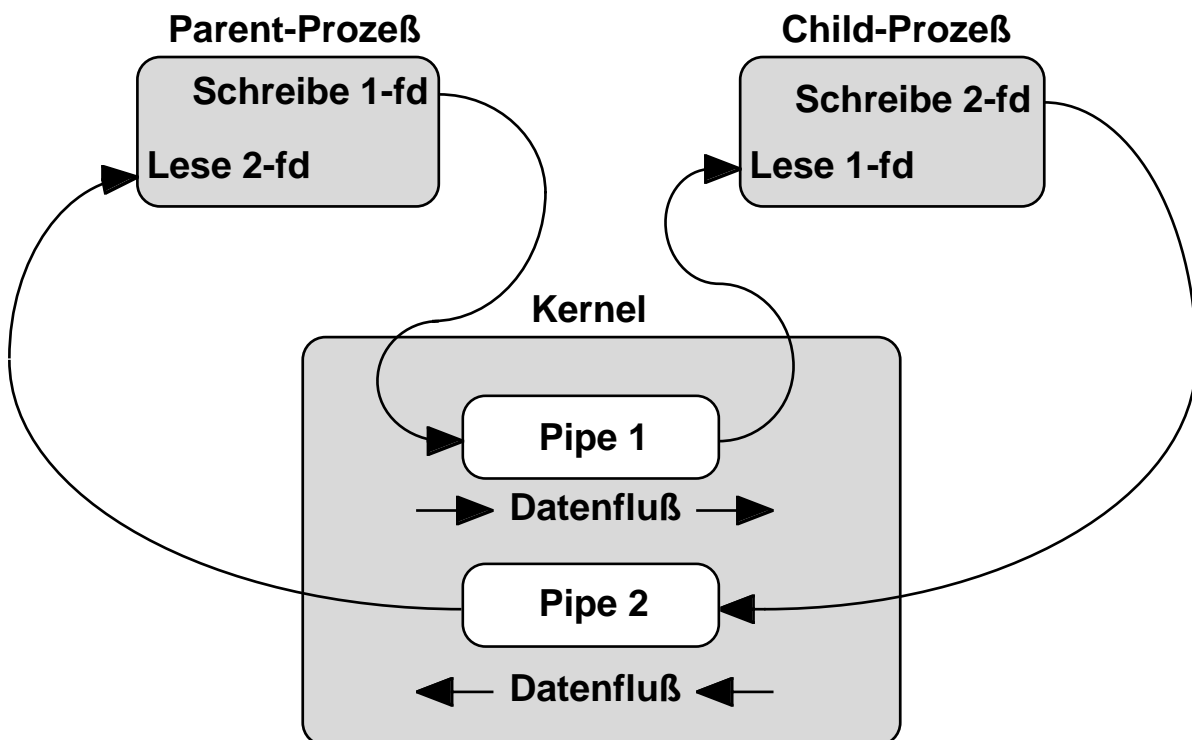


Bild 1.2 " Pipes, die einen bidirektionalen Datenfluß ermöglichen

5 Prozeß-Steuerung

In Netzen kommunizieren stets 2 oder mehr Prozesse miteinander. Nun wird betrachtet wie Programme ausgeführt werden, wie Prozesse aufgerufen und beendet werden.

`fork` Systemaufruf

Wie bereits erwähnt, gibt es unter Unix nur eine Möglichkeit, einen Prozeß zu erzeugen. Diese besteht darin, daß ein bereits bestehender Prozeß den Systemaufruf `fork` verwendet, um einen Prozeß zu erzeugen.²

```
int fork();
```

Der `fork`-Systemaufruf erstellt eine Kopie des ihn aufrufenden Prozesses. Der Prozeß, der den `fork`-Aufruf durchführte, wird als *Parent-Prozeß* bezeichnet. Der neue Prozeß wird *Child-Prozeß* genannt. Der `fork`-Systemaufruf wird einmal aufgerufen, aber zweimal beendet. Aufgerufen wird er vom Parent-Prozeß, beendet wird er von diesem und vom neuen Child-Prozeß. Der einzige Unterschied in der Beendigung der beiden Prozesse liegt darin, daß der Parent-Prozeß als Resultat die ProzeßID des neu erzeugten Child-Prozesses zurückliefert. Der Child-Prozeß liefert hingegen eine 0 zurück. Wird der `fork`-Systemaufruf nicht erfolgreich ausgeführt, dann wird der Wert -1 zurückgeliefert. Wenn der Child-Prozeß die ProzeßID seines Parent-Prozesses benötigt, dann kann er diese über den Systemaufruf `getppid` erhalten.

Zu beachten ist, daß der Datenbereich des Child-Prozesses direkt nach dem `fork`-Aufruf eine Kopie des Datenbereiches des Parent-Prozesses ist. Der Datenbereich des Child-Prozesses wird also nicht von der jeweiligen Programmdatei entnommen.

`exit` Systemaufruf

```
exit(int Status);
```

Ein Prozeß beendet sich durch den `exit`-Systemaufruf. Bei diesem Systemaufruf erfolgt kein Rücksprung zum aufrufenden Prozeß. Stattdessen wird von dem Prozeß eine Integergröße an den Kernel weitergegeben. Dieser Exitstatus kann dann vom Parent-

² Dies betrifft alle Prozesse, mit Ausnahme des `init`-Prozesses. Dieser hat die Prozeß-ID 0 oder 1 und wird vom Kernel gestartet, wenn das Unix-System gestartet wird.

Prozeß des beendeten Prozesses verarbeitet werden. Der Zugriff auf den Exitstatus erfolgt über den Systemaufruf `wait`.

`exec` Systemaufruf

Unter Unix kann ein Programm von einem bestehenden Prozeß nur durch Aufruf des `exec`-Systemaufrufes gestartet werden. Durch den Systemaufruf `exec` wird der gerade ausgeführte Prozeß durch das neu auszuführende Programm ersetzt. Dabei wird die Prozeß-ID nicht verändert. Der Prozeß, der den `exec`-Aufruf durchführt, wird als *aufrufender Prozeß* bezeichnet.

Es gibt von der `exec`-Funktion verschiedene Varianten:

```
int execlp(char *filename, char *arg0, char *arg1, ...,
           char *argn, (char *) 0);

int execl(char *pathname, char *arg0, char *arg1, ..., char
          *argn, (char *) 0);

int execlp(char *pathname, char *arg0, char *arg1, ...,
           char *argn, (char *) 0, char **envp);

int execvp(char *filename, char **argv,);

int execv(char *pathname, char **argv);

int execve(char *pathname, char **argv, char **envp);
```

Die `exec`-Funktion liefert nur im Fehlerfall einen Funktionswert an den aufrufenden Prozeß zurück. Ansonsten wird die Kontrolle an das neu gestartete Programm übergeben.

Das durch den `exec`-Aufruf gestartete neue Programm erbt folgende Eigenschaften vom Prozeß, der den Aufruf durchführt:

- Prozeß-ID
- Parent-Prozeß-ID
- Prozeßgruppen-ID
- Terminalgruppen-ID

- Stammverzeichnis
- aktuelles Arbeitsverzeichnis
- Maske für den Dateistatus
- reale User-ID
- reale Gruppen-ID
- Dateizugriffssperren (noch erläutern)
- bis zum auftreten eines Alarmsignals noch zur Verfügung stehende Rechenzeit

2 Attribute können sich (falls das Set-User-ID-Bit und oder das Set-Gruppen-ID-Bit gesetzt sind) beim neuen Programm ändern:

- effektive User-ID
- effektive Gruppen-ID

`wait` **Systemaufruf**

Ein Prozeß kann auf die Beendigung eines seiner Child-Prozesse warten, indem er den Systemaufruf `wait` verwendet.

```
int wait(int *status);
```

Der von diesem Aufruf zurückgelieferte Wert, ist die ProzeßID des beendeten Child-Prozesses. Wenn der Prozeß, der `wait` aufruft, gar keine Child-Prozeß besitzt, dann wird sofort der Wert -1 zurückgeliefert.

Teil 2: Aufgaben

Es sollen kompakte, kommentierte C-Programme geschrieben werden, die die unten stehende Aufgaben erbringen. Dabei soll auf möglichst viel bereits benutzten C-Code zurückgegriffen werden.

1. Programmargumente und Ein/Ausgabe

1.a Das Programm gibt alle Programmargumente mittels printf wieder aus. Dabei sollen die Argumente numeriert werden.

Beispiel einer möglichen Formatierung der Ausgabe:

```
Arg1 => prog_1a
Arg2 => hello
```


1.b Das zu erstellende Programm erwartet die Eingabe von zwei Parameter: der erste wird als *Eingabedatei* interpretiert, der zweite als *Ausgabedatei*. Es soll geprüft werden ob die Eingabedatei vorhanden ist, die Ausgabedatei darf nicht vorhanden sein. Eine falsche Parameteranzahl führt auch zum Programmende mit Fehlermeldung.

Der Inhalt der ersten Datei soll dann in die zweite kopiert werden, mittels Standard-E/A Aufrufe.

1.c.1 Eine Variante des Programms 1.b: wenn die Ein- und Ausgabedatei nicht genannt werden, wird von stdin gelesen und auf stdout geschrieben. Wird nur ein Parameter angegeben, wird er als Eingabedatei aufgefaßt (und als Ausgabedatei stdout gewählt).

1.c.2 Eine weitere Variante des Programms 1.b: wenn die Ein- und/oder Ausgabedatei nicht genannt werden, werden sie interaktiv erfragt.

1.d Dieses Programm ist eine Erweiterung des 1.c.2, und erlaubt die Angabe eines dritten optionalen Parameters (-x). Ist er vorhanden, werden alle Kleinbuchstaben der Eingabe in Großbuchstaben umgewandelt.

 In der C-Bibliothek (Definition in ctype.h) gibt es die Funktion "toupper", die Kleinbuchstaben in Großbuchstaben wandelt.

2. Prozesse

2.a.1 Schreibe ein C-Programm daß die Prozeß-ID des laufenden Prozesses ausgibt, in folgendem Format:

```
Prozeß-ID = xxxx  
xxxx Fertig!
```

2.a.2 Dieses Programm gibt neben seiner eigenen noch die Prozeß-ID seines Parent-Prozesses aus, im Format:

```
PID = xxxx   Parent-ID = yyyy  
xxxx Fertig!
```

2.b Ein Programm soll ein Kind-Prozeß erzeugen. Beide Prozesse (Parent und Child) geben neben ihre PID auch die PID *ihres* Parent aus.

2.c Über das zu erstellende Programm (`run`) sollen 2 beliebige Programme gestartet werden (Namensangabe über Programmargumente). Mit Fehlerbehandlung falls der Start nicht funktioniert.

Beispiel :

```
run a b           # startet die Programme a und b
```

2.d (optionale Aufgabe)

Das zu erstellende Programm startet ein Programm (als Child), dessen Namen als Parameter übergeben wird. Der Parent-Prozeß wartet auf die beendigung des Child-Prozesses.

3. Pipes (optionale Aufgabe)

3.a Schreiben Sie ein Programm, das Daten aus einer (als Parameter genannten) Datei liest, und diese Daten einem Child-Prozess mittels einer Pipe überträgt. Zur einfacheren Fehlersuche und Dokumentation des Prozeßablaufs sollen möglichs regelmäßig die Programmschritte über `printf` gemeldet werden.