Systemnahe Software

Prof. Dr. Franz Schweiggert und Mitarbeiter

1. Mai 2004

Fakultät Mathematik u. Wirtschaftswissenschaften
Abteilung Angewandte Informationsverarbeitung
Vorlesungsbegleiter (gültig ab SS 2000)

Anmerkungen:

- Auf eine systematische Unterscheidung zwischen *BSD Unix* und *System V Unix (Linux, Solaris)* wird hier verzichtet.
- Die enthaltenen Beispiel-Programme wurden zum großen Teil unter Linux entwickelt und sind weitestgehend unter Solaris getestet (fur konstruktive Hinweise sind die Autoren dankbar).
- Die Beispiele sollen jeweils gewisse Aspekte verdeutlichen und erheben nicht den Anspruch von Robustheit und Zuverlässigkeit. Man kann alles anders und besser machen.
- Details zu den behandelten / verwendeten System Calls sollten jeweils im Manual bzw. den entsprechenden Header-Files nachgelesen werden.
- Dieses Skript ersetzt kein Lehrbuch und erhebt auch nicht den Anspruch eines zu sein. Die Erklärungen der Vorlesung sind zum Verständnis unverzichtbar.

Literatur: (jeweils neueste Ausgabe beachten!)

- [Bach86] M. J. Bach: The Design of the UNIX Operating System. Prentice Hall 1986
- [Comer91] D. E. Comer: Internetworking with TCP/IP. Volume I; Principles, Protocols, and Architecture. Prentice Hall 1991
- [Comer91] D. E. Comer, David L. Stevens: Internetworking with TCP/IP. Volume II; Design, Implementation and Internals. Prentice Hall 1991
- [Etter99] M. Etter: Zur Qualitätsverbesserung in der Entwicklung und Pflege verteilter Anwendungen - Konzeption und Realisierung einer Bibliothek auf der Basis von TCP/IP. Dissertation an der Universität Ulm, Abt. Angewandte Informationsverarbeitung, 1999.
- [Rochkind85] M. Rochkind: UNIX-Programmierung für Fortgeschrittene. Hanser Verlag 1985
- [Santifaller] M. Santifaller: TCP/IP and ONC/NFS. Internetworking in a UNIX Environment. Addison Wesley 1994
- [Stevens92] W. R. Stevens: Advanced Programming in the UNIX Environment. Addison-Wesley 1992
- [Stevens95] W. R. Stevens: TCP/IP Illustrated, Volume 2 - The Implementation. Addison-Wesley 1995
- [Stevens96] W.R. Stevens: TCP/IP Illustrated, Volume 3 - TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols. Addison-Wesley 1998
- [Stevens98] W.R. Stevens: UNIX Network Programming, Volume 1 - Networking APIs: Sockets and XTI. Prentice Hall 1998.
- [Stevens99] W.R. Stevens: UNIX Network Programming, Volume 2 - Interprocess Communications. Prentice Hall 1999.
- [Tanenbaum87] A. S. Tanenbaum: Operating Systems - Design and Implementation. Prentice Hall 1987

Inhaltsverzeichnis

Kapitel 1

Unix Process Subsystem

1.1 "Theorie" der Prozesse

- Die Ausführung eines Programms heißt **Prozess**.
- Der **Kontext** eines Prozesses ist seine Ausführungsumgebung. Dazu gehören: Anweisungen, Daten, Stack und Heap - Register - Betriebssystem-Ressourcen (offene I/O-Verbindungen, IPC-Elemente, ...)
(IPC: *InterProcess Communication*)
- Ausführbare Programme sind z.B.:
  - vom Compiler generierter Objektcode ("a.out")
  - Datei mit einer Folge von Shell-Kommando's → **chmod+x**)
- Das Betriebssystem verwaltet eine endliche Menge von Prozessen und versucht, die vorhandenen Resourcen (Speicher, Rechenzeit, I/O-Operationen) fair auf die einzelnen Prozesse zu verteilen.
Ein Prozess folgt bei der Ausführung einer genau festgelegten Folge von Anweisungen, die in sich abgeschlossen sind. Schutzmechanismen des Betriebssystems und der Hardware verhindern, dass ein Prozess auf Anweisungen oder Daten außerhalb seines Adreßraums zugreift. Die gesamte Kommunikation mit anderen Prozessen (*IPC*) oder mit dem Kernel muß über System Calls erfolgen.
- (Fast) alle Prozesse entstehen aus einem bereits existierenden Prozess durch Aufruf von **int fork()**.
- Jeder Prozess besitzt eine Prozessnummer (**PID**). als eindeutige Identifikation, die einen Index in die Prozesstabelle darstellt. In dieser Tabelle verwaltet das Betriebssystem die wichtigsten Daten aller Prozesse.
Mit der Funktion **int getpid()** kann ein Prozess seine eigene *PID* abfragen:.
- (Fast) jeder Prozess hat einen "Erzeuger" (siehe oben: *fork()*), der sog. **parent process**
Dessen *PID* kann ein Prozess mit der Funktion **int getppid()** erhalten.
- Jeder Prozess gehört zu einer **Prozessgruppe**, die ebenfalls durch eine kleine positive Integerzahl identifiziert wird (*process group ID*).
Ist die *process ID* eines Prozesses gleich der *process group ID*, so wird dieser Prozess als **process group leader** bezeichnet.

Mit **kill** (Shell-Kommando und auch System Call) können Signale an alle Prozesse einer *process group* gesandt werden.
Bestimmung der *process group ID*: Funktion **int getpgrp**();
in BSD4.3: *int getpgrp(int pid);* ist das Argument *pid* 0, liefert diese Funktion die *process group ID* des aktuellen Prozessen, ansonsten die der Prozessgruppe des über *pid* angegebenen Prozesses.

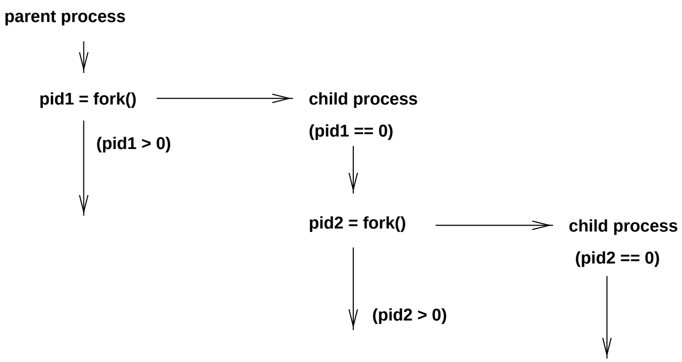
Die Zuordnung zu einer Prozessgruppe kann geändert werden:
unter System V:
int setpgrp();
damit wird der aufrufende Prozess zum *process group leader*; als *process group ID* wird die *process ID* geliefert.
unter 4.3.BSD:
int setpgrp(int pid, int pgrp);
ist *pid* 0, so ist der aktuelle Prozess gemeint, ansonsten muß der angesprochene Prozess dieselbe effektive *user ID* wie der aktuelle Prozess haben oder der aktuelle Prozess muß Superuser Privilegien haben.

Beispiel:

- Jeder Prozess kann Mitglied einer **terminal group** sein, die selbst wieder über eine positive ganze Zahl (*terminal group ID*) identifiziert wird. Diese ist die *process ID* des *process group leader*, der das Terminal "geöffnet" hat - typischerweise die *login shell*. Dieser Prozess wird **control process** für dieses Terminal genannt. Jedes Terminal hat nur einen *control process*.
Die *terminal group ID* identifiziert das Kontroll-Terminal für jede Prozessgruppe. Das Kontroll-Terminal wird benutzt, um Signale abzuarbeiten (von der Tastatur aus erzeugt, oder wenn die *login shell* terminiert). Wenn der Prozess, der gleichzeitig Kontroll-Prozess eines Terminals und Prozessgruppenführer ist (typischerweise die *login shell*), ein *exit* zum Beenden aufruft, so wird das **hangup** Signal an alle Prozesse dieser Gruppe gesandt. Das Kontrollterminal wird automatisch referenziert durch */dev/tty*. Ein **Dämon-Prozess** hingegen ist ein Hintergrundprozess ohne kontrollierendes Terminal. (Z.B. *httpd*)

1.2 Prozess-Baum

- Hierarchie und Vererbung



Vererbung: Prozesse vererben Teile ihrer Umgebung an ihre Kindprozesse:

- die Standard-Dateiverbindungen
- den aktuellen Katalog
- Umgebungsvariablen

Auch die Regeln für die Weitergabe von Informationen folgen streng der hierarchischen Vererbungslehre. Elternprozesse können ihren Kindern Teile ihrer Umgebung als Kopie mitgeben. Die umgekehrte Richtung ist nicht möglich. Prozesse können Daten untereinander nur über **IPC**-Mechanismen (*interprocess communication*) austauschen.
Eine Ausnahme bildet der **Beendigungsstatus** (kleine ganze Zahl), den ein Prozess bei seiner Beendigung an seinen Erzeugerprozeß zurückgeben kann (**exit()** → **wait()**). Der Beendigungsstatus 0 signalisiert einen erfolgreichen Ablauf des Prozesses (UNIX -Konvention); ein von Null verschiedener Wert ist das Zeichen für eine nicht erfolgreiche Beendigung und wird in der Regel mit einer Fehlermeldung auf der Standard-Fehlerausgabe *stderr* verbunden (Exit-Status > 0: nicht notwendig Fehlersituation, s. z.B. egrep()).

1.3 Prozessmanagement

- Kommando **ps** - Ausgabe der Prozesstabelle

  Das Kommando *ps* gibt alle wesentlichen Informationen über die aktuelle Prozesshierarchie (→ Prozesstabelle) aus.

  Je nach Option (und Betriebssystemversion) besteht die Ausgabe von *ps* aus mehr oder weniger Informationen zu den einzelnen Prozessen. Die Ausgabe ist, wie bei *ls*, in tabellarischer Form.

  Bedeutung:

PID	die eindeutige Prozessnummer (process id)
TTY	Name des Terminals (Windows), an dem der Prozess gestartet wurde
STAT	Prozeßstatus - R für Running, S für Sleeping T für Stopped, Z für **Zombie** (tot, aber nicht vom Vater beerdigt) (siehe man ps)
TIME	verbrauchte Rechenzeit
COMMAND	der zum Prozess gehörende Kommandoname

- Kommando **top** - Darstellung der CPU-intensiven Prozesse → **man top**

- Kommando **kill** - Prozesse abbrechen

  Mit dem Kommando **kill nummer** kann man den Prozess mit der Prozessnummer *nummer* gewaltsam beenden (mit *kill* werden an einen Prozess Signale verschickt; siehe *signal.h*, *man kill* oder *kill -l*).

  Auf diesem Weg lassen sich Hintergrundprozesse abbrechen, falls sie blockiert sind oder ungewöhnlich viel CPU -Zeit verbrauchen (Endlosschleife?).

  Ein blockiertes Terminal läßt sich durch ein *kill*-Kommando für die betreffende Login-Shell, von einem anderen Terminal aus, im Normalfall wieder in einen benutzbaren Zustand zurückbringen. Die Login-Shell ist in der Spalte *COMMAND* als *-sh* oder schlicht als - aufgeführt.

  Als gewöhnlicher Benutzer darf man nur seine eigenen Prozesse mit *kill* beenden. Als Super-User darf man alle Prozesse beenden; von dieser Möglichkeit sollte man aber nur dann Gebrauch machen, wenn man sich über die Konsequenzen im klaren ist...

1.4 Synchrone und asynchrone Prozesse

- Vordergrundverarbeitung

  In der Regel verarbeitet die Shell Kommandos *synchron*. Sie startet ein Kommando (→ erzeugt Prozess) und erst nach dessen Beendigung meldet sie sich mit dem *Prompt* (beginnt sie mit dem nächsten Kommando). Der Erzeugerprozeß wartet auf das Ende seines Kindprozesses.

- Hintergrundverarbeitung

  Kommandos, deren Abarbeitung länger dauert (etwa in der Statistik oder Numerik), blockieren bei synchroner Abarbeitung den interaktiven Betrieb. Schließt ein **&**-Zeichen das Kommando ab, wartet die Shell (Erzeugerprozeß) nicht auf das Ende des gerade gestarteten Prozesses. Sie ist statt dessen sofort bereit, das nächste Kommando entgegenzunehmen.

- Fußangeln

  Hintergrundkommandos, die von der Standardeingabe lesen wollen, erhalten das Signal **SIGTTIN** - der Prozeß wird angehalten, existiert aber weiterhin! Damit wird verhindert, dass sich ein Vordergrundprozeß (meistens die Shell selbst) und mehrere aktive Hintergrundprozesse um die Eingabe vom Terminal streiten.

  Mit dem Kommando **fg** kann ein via SIGTTIN angehaltener Prozeß in den Vordergrund geholt werden.

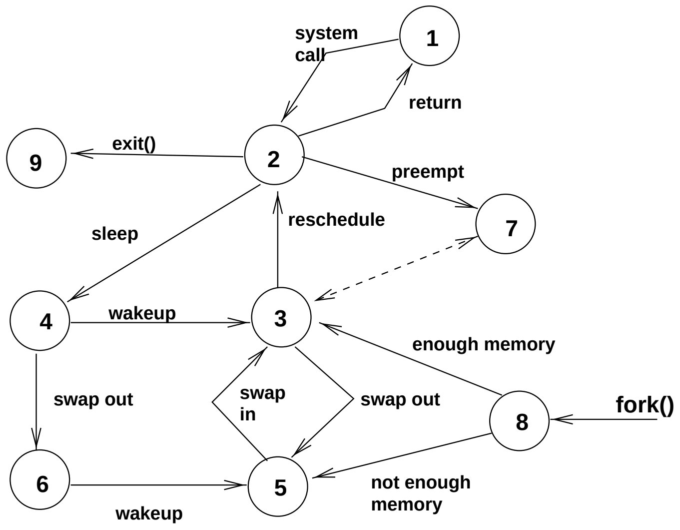
  Ein beliebiges Hintergrundkommando terminiert automatisch mit der Terminalsitzung, aus der es gestartet wurde; dies läßt sich für Programme mit extrem langer Laufzeit verhindern:

  nohup kommando &

1.5 CPU-intensive Prozesse

Mit dem Kommando **nice** kann die Priorität eines Prozesses verändert (herabgesetzt) werden!

1.6 Prozesszustände und Zustandsübergänge



1 Ausführung im **User Mode** (Prozess arbeitet in seinem Adreßraum)

2 Ausführung im **Kernel Mode** (Prozess arbeitet im Kernel-Adreßraum)

3 Prozess wartet auf Zuteilung der CPU durch Scheduler

4 Prozess "schläft" im Hauptspeicher, wartet auf bestimmtes Ereignis, das ihn "aufweckt"

5 Prozess ist "ready to run", aber ausgelagert - muß vom "Swapper" (Prozess 0) in Hauptspeicher geholt werden, bevor der Kern ihn zuteilen kann

6 Prozess "schläft" im Sekundärspeicher (ausgelagert)

7 Prozess geht vom *Kernel Mode* in *User Mode*, aber der Kern supendiert ihn und vollzieht einen Kontext-Switch (anderer Prozess kommt zur Ausführung (bis auf Kleinigkeiten identisch mit 3)

8 Prozess neu erzeugt (Startzustand für jeden Prozess - ausgenommen Prozess 0)

9 Prozess hat **exit** -Aufruf ausgeführt, existiert nicht weiter, hinterläßt aber Datensatz (Exit-Code, Statistik-Angaben) für Vater-Prozess - Endzustand

- Übergänge

  System Calls und Kernel-Serviceroutinen, sowie äußere Ereignisse (Interrupts durch Uhr, HW, SW, ...) können Zustandswechsel bei den Prozessen bewirken.

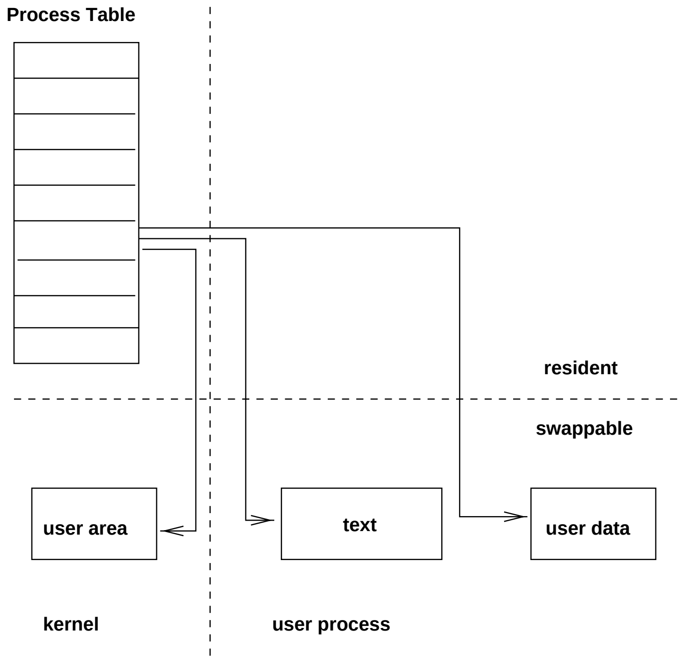
Ein *"Context Switch"* zwischen Prozessen ist nur beim Zustandswechsel des aktiven Prozesses von Zustand (2) *"Running in Kernel Mode"* nach (4) *"Asleep"* möglich. Beim *"Context Switch"* wählt der Scheduler den Prozess mit der größten Priorität (Zur Erinnerung: Kommando **nice**!) aus der Ready-Liste (alle Prozesse im Zustand (3) *"Ready to Run"*) aus und befördert ihn in den Zustand (2). Der Scheduler teilt ihm die CPU zu und bringt ihn damit zur Ausführung.

1.7 Regions

Der *Kernel* unterteilt den **virtuellen** Adreßraum eines Prozesses in logische (abstrakte) Bereiche (**regions**) zusammenhängender Adressen; diese werden als verschiedene Objekte behandelt, die als Ganzes geschützt (*protected*) bzw. geteilt (*shared*) werden können
Text, Daten, Stack bilden typischerweise separate Bereiche
Der *Kernel* unterhält eine Tabelle mit Verweisen auf die aktiven Bereiche (u.a. Information, wo diese im physischen Speicher liegen)
Jeder Prozess unterhält eine private *per process region table* (z.B. in der **u area**)

- Context
  Der *UNIX*-Jargon unterteilt den *Context* in drei disjunkte Bereiche: *"Text"*, *"User Data"* und *"System Data"*. Davon hält der Kernel die für ihn wichtigen Daten *("System-Daten")* an zwei definerten Plätzen in seinem Adreßraum - in der **Prozesstabelle** und in der **"User Area"** *("per process data region")*

  Text und User-Daten Bereich bilden den *"User Adreßraum"*.



- Context Switch

  Ein *"Context Switch"* findet statt, wenn der Scheduler beschließt, dass ein anderer Prozess zur Ausführung kommen soll. Der Kernel hält den laufenden Prozess an, sichert dessen *Context* und lädt den *Context* des nächsten Prozesses.

  Bei jedem Wechsel sichert der Kernel soviel Informationen, dass er später zu dem unterbrochenen Prozess zurückkehren und dessen Ausführung fortsetzen kann. Für den Prozess bleibt die Unterbrechung völlig transparent.

  Ein *"Context Switch"* kann nur stattfinden, während der Prozess im Kernel Mode arbeitet. Ein Wechsel des *"Execution Mode"* ist kein *"Context Switch"*.

Einträge in der Prozesstabelle:

- Prozesszustand (siehe obiges Beispiel *bg.c*: Zustand T)
- Information, wo der Prozess und die *u area* im Haupt-/Sekundärspeicher liegen sowie über Speicherbedarf (☞ *context switch*)
- "Verwandtschaftsverhältnisse"

- *scheduling* Parameter
- diverse "Uhren" (→ *time*)
- u.a.m.

Mehr dazu: **man ps** !

Einträge in der "u area"

- Reale und effektive UserID
- Vektor zur Signalverarbeitung
- Hinweis auf *login terminal*
- Fehlereintrag für Fehler bei einem Systemaufruf
- Rückgabewert von Systemaufrufen
- Aktueller Katalog, Aktuelle Wurzel
- *User File Descriptor Table*
- u.a.m.

Kapitel 2

Grundlegende System Call's

2.1 System Call "fork"

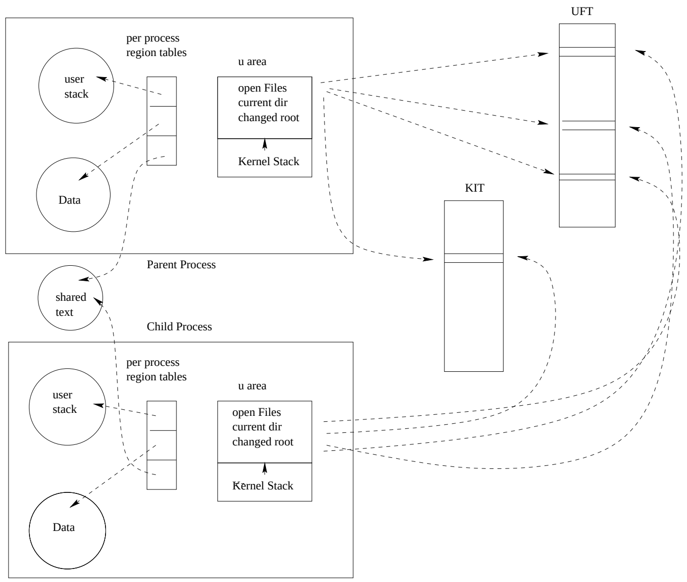
Syntax

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

pid_t fork (void) /* create new processes */
   /* returns PID and 0 on success or −1 on error */
```

Beschreibung
Der System Call *fork* erzeugt einen neuen Prozess, indem er den Context des ausführenden Prozesses dupliziert.

- Der neu erzeugte Prozess wird als Kind-Prozess (**child process**), der aufrufende als Vater-Prozess (**parent process**) bezeichnet.
- *fork* wird einmal aufgerufen und kehrt im Erfolgsfall zweimal zurück: der Aufrufer erhält als Rückgabewert die PID des erzeugten Prozesses, der Kindprozeß erhält 0; im Fehlerfall (z.B. bereits zuviele Prozesse erzeugt) kehrt er einmal mit -1 zurück.

Beispiel




```
/* fork1.c */

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

int main()
{
     int pid;

     if ( (pid = fork()) > 0 ) {
        printf("Parent: created process %d, my pid is %d\n",
                 pid, (int) getpid());
     }
     else if ( pid == 0 ) {
         printf("Child: after fork, my pid is %d\n", (int) getpid());
         printf("Child: my parent is %d\n", (int) getppid());
          }
     else
         perror( "fork() - can't fork a child" );
```

```
     printf("fertig\n");
     exit( 0 );
}
```

Ausgabe:

```
Parent: created process 319, my pid is 318
Child: after fork, my pid is 319
Child: my parent is 1
```

Der Vater-Prozess terminiert hier vor dem Sohn, der Init-Prozess "erbt" den "verlorenen Sohn". (Das Geschehen ist **nicht** deterministisch!)
Oder so (Erzeuger "schläft" ein bißchen, bevor er terminiert):

```
/* fork2.c */

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

int main()
{
   int pid;
   switch ( pid = fork())  {
      case -1:
         perror("fork() - can't fork a child");
         break;
      case 0:
         printf("Child: after fork, my pid is %d\n", (int) getpid());
         printf("Child: my parent is %d\n", (int) getppid());
         break;
      default:
         sleep(5);
         printf("Parent: created process %d, my pid is %d\n",
                     pid, (int) getpid());
           break;
   }
   printf("Who am I?\n");
   exit(0);
}
```

Ausgabe:

```
Child: after fork, my pid is 347
Child: my parent is 346
Who am I?
Parent: created process 347, my pid is 346
Who am I?
```


Vererbt werden:

- *real user ID*
- *real group ID*
- *effective user ID*
- *effective group ID*
- *process group ID*
- *terminal group ID*
- *root directory*
- *current working directory*
- *signal handling settings*
- *file mode creation mask*

Unterschiede:

- *process ID*
- *parent process ID*
- eigene File Deskriptoren (Kopie)
- Zeit bis zu einem ggf. gesetzten *alarm* ist beim Kind auf 0 gesetzt

Gebrauch:

1. Ein Prozess erzeugt von sich selbst eine Kopie, so dass diese "die eine oder andere" Operationen ausführt (→ Server).
2. Ein Prozess will ein anderes Programm zur Ausführung bringen - der Kind-Prozess führt via *exec* ein neues Programm aus (überlagert sich selbst) (→Shell)

2.2 System Call "exit"

Der eigentliche SystemCall ist `_exit()`, wir verwenden eine entsprechende Bibliotheksfunktion (die auch noch etwas aufräumt):
Syntax

```
#include <stdlib.h>
void exit( int status ) /* terminate process */
                        /* does NOT return   */
                        /* 0 <= status < 256 */
```

Beschreibung

- Mit dem System Call *exit* beendet ein Prozess aktiv seine Existenz. Von diesem System Call gibt es keine Rückkehr in den User Mode.
- Unterscheide: System Call *_exit* und C-Bibliotheks-Funktion *exit*
C-Funktion leert erst alle Puffer und ruft dann dann den System Call auf.
- Der Kernel gibt den User Adreßraum (Text und User-Daten) des Prozesses frei, sowie auch einen Teil des System-Daten Bereichs (User Area). Übrig bleibt nur der Eintrag in der Prozesstabelle, der den Beendigungsstatus und die Markierung des Prozesses als **Zombie** enthält, bis ihn der init-Prozess erbt und abräumt.

2.3 System Call "exec"

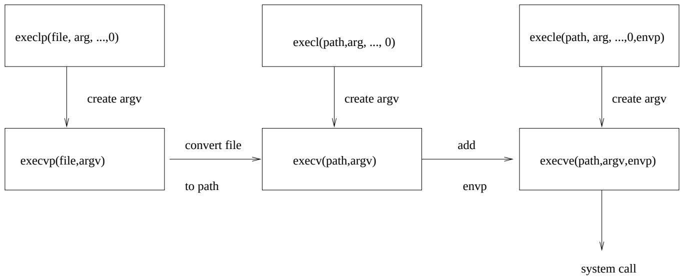
Syntax

```
#include <unistd.h>
extern char ** environ;

int execl( char *path,       /* path of program file */
           char *arg0,       /* 1st argument (cmd name) */
           char *arg1, ...,  /* 2nd, ... argument */
           char *argn,       /* last argument */
           (char *) 0
         );

int execlp( char *filename,    /* name of program file */
            char *arg0, char *arg1, ... char *argn,
            (char *) 0
          );

int execle( char *path,   /*path of program file */
            char *arg0, char *arg1, ... char *argn,
            (char *) 0,
            char **envp)  /* pointer to environment */
          );

int execv( char *path,    /* path of program file */
           char *argv[];  /* pointer to array of argument */
         );

int execvp( char *filename,    /* name of program file */
            char *argv[];      /* pointer to array of argument */
          );

int execve( char *path,    /* path of program file */
            char *argv[],  /* pointer to array of argument */
            char *envp[],  /* pointer to environment */
         );
 /* all return with -1 on error only */
```

Beschreibung

Die **e**xec - System Calls überlagern im Context des ausführenden Prozesses den Text und den User-Daten Bereich mit dem Inhalt der Image-Datei. Anschließend bringen sie die neuen Instruktionen zur Ausführung.

Zusammenhang



1. Die drei Funktionen in der oberen Reihe enthalten jedes Kommando-Argument als separaten Parameter, der NULL-Zeiger ((char *) 0 - !!!) schließt die variable Anzahl ab (kein *argc*!). Die drei Funktionen in der unteren Reihe fassen die Kommando-Argumente in einen Parameter **argv** zusammen, das Ende wird entsprechend wieder durch den NULL-Zeiger definiert.

2. Die zwei Funktionen in der linken Spalte definieren die Programm-Datei nur durch den Datei-Namen; diese wird über die Einträge in der Umgebungsvariable **PATH** gesucht (konvertiert in vollen Pfadnamen). Falls diese nicht gesetzt ist, wird als *default*-Suchpfad ":/bin:/usr/bin" genommen. Enthält das Argument *path* einen *slash*, so wird die Variable *PATH* nicht verwendet.

3. Bei den vier Funktion in den beiden linken Spalten wird das Environment über die externe Variable *environ* an das neue Programm übergeben. Die beiden Funktionen in der rechten Spalte spezifizieren explizit eine Environment-Liste (muß ebenfalls mit einem NULL-Zeiger abgeschlossen sein).

Vererbung bei *exec()*:

- *process ID*
- *parent process ID*
- *process group ID*
- *terminal group ID*
- *time left until an alarm clock signal*
- *root directory*
- *current working directory*
- *file mode creation mask*
- *real user ID*

- *real group ID*
- *file locks*

Mögliche Anderungen mit *exec()*:
set user ID / set group ID - bit der neuen Datei gesetzt

- *effective user ID* auf *user ID* des Besitzers der Programm-Datei
- *effective group ID* auf *group ID* des Besitzers der Programm-Datei

Signale:

- Terminieren bleibt Terminieren
- Ignorieren bleibt Ignorieren
- Speziell abgefangene Signale werden wegen des Überlagerns auf Terminieren gesetzt

Beispiel:

```
/* ------------- exec1.c ------------------------*/

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


void exec_test()              /* execl version */ {
   printf( "The quick brown fox jumped over " );
   fflush( stdout );
   execl( "/bin/echo", "echo", "the", "lazy", "dogs.",
                                         (char *) 0 );
   perror("execl - can't exec /bin/echo" );
}

int main() {
   int pid;

   if( (pid = fork() ) > 0)
      printf("PP: Parent-PID: %d / Child-PID: %d\n",
                                      (int) getpid(), pid);
   else if (pid == 0) {
      printf("CP: Child-PID: %d\n", (int) getpid() );
      exec_test();
      }
   else
      perror("fork() - can't fork");
   exit(0);
}
```

Ausgabe:

```
PP: Parent-PID: 662 / Child-PID: 663
CP: Child-PID: 663
The quick brown fox jumped over the lazy dogs.
```

Läßt man in obigem Beispiel den Funktionsaufruf **fflush()** weg, so passiert folgendes:

```
/* -------------- exec2.c -----------------*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

void exec_test()             /* execl version */ {
   printf( "The quick brown fox jumped over " );
   execl( "/bin/echo", "echo", "the", "lazy", "dogs.",
                                          (char *) 0 );
   perror("execl - can't exec /bin/echo" );
}

int main() {
   int pid;

   if( (pid = fork() ) > 0)
     printf("PP: Parent-PID: %d / Child-PID: %d\n",
                               (int) getpid(), pid);
   else if (pid == 0) {
     printf("CP: Child-PID: %d\n", (int) getpid() );
     exec_test();
   }
   else
     perror("fork() - can't fork");
   exit(0);
}
```

Ausgabe:

```
PP: Parent-PID: 671 / Child-PID: 672
the lazy dogs.
```

printf() erledigt die Ausgabe gepuffert (in Blöcken zu 512 Bytes), wenn in eine Datei oder in eine Pipe geschrieben wird. Bei der Ausgabe ans Terminal (s.o.) kann die Ausgabe auch dann bruchstückhaft (oder gar nicht) erscheinen, wenn Zeilenpufferung stattfindet. Normalerweise führt dies zu keinen Problemen; der letzte Puffer wird automatisch entleert, wenn der Prozess endet. In obigem Beispiel war der Prozess noch nicht beendet, als der **exec** -Aufruf erfolgte - der Benutzerdatensegment liegende Puffer wurde überlagert, bevor er entleert werden konnte.

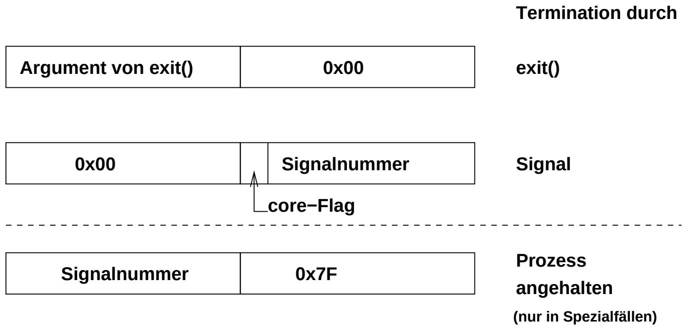
2.4 System Call "wait"

Syntax

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

pid_t wait( int * status );
/* returns PID of child or -1 on error */
```

Beschreibung von wait() (zu waitpid() siehe Manuals)

- Falls der Aufrufer von *wait()* keinen Kind-Prozess erzeugt hat, kehrt *wait()* mit -1 zurück.
- Kehrt zurück, falls ein Kind bereits vorher terminierte
- Ansonsten blockiert *wait()* (wird vom Kernel suspendiert), bis einer der erzeugten Prozesse terminiert. In diesem Fall liefert *wait()* die PID des terminierten Kind-Prozesses. Über das Argment von *wait()* erhält der Prozess Informationen über den terminierten Kind-Prozess:



- Signal-Nummern sind größer 0!

Ein erstes Beispiel mit wait():

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
  pid_t pid;

  if ( (pid = fork() ) < 0 )
    perror("fork"), exit(1);

  if (pid == 0) {
    printf("Child -- my pid is %d\n", (int) getpid());
    sleep(10);
    return 0;
  } else {
    // Parent
    int status, signal_nr, exit_st;
    pid_t child_pid = wait(&status);
    if (child_pid == -1)
       perror("wait"), exit(2);
    signal_nr = status & 0x3f;
    exit_st = (status >> 8) & 0xff;
    printf("Parent -- child with pid %d finished\n", (int) child_pid);
    printf("SignalNr: %d || ExitSt: %d\n", signal_nr, exit_st);
    return 0;
  }
}
```


Makros erleichtern die Dekodierung:
aus `/usr/include/sys/wait.h`

- **WIFEXITED(status)???**
- **WIFSIGNALED(status)???**
- **WIFSTOPPED(status)???**
- **WEXITSTATUS(status**
- **WTERMSIG(status**
- **WSTOPSIG(status**

waitpid():

```
pid_t waitpid(pid_t pid, int * status, int options);
```

Parameter **pid**:

< -1 auf alle Kind-Prozesse, warten, deren Prozessgruppen-ID gleich dem Betrag von `pid` ist

-1 auf alle Kind-Prozesse warten (wie `wait()`)

0 auf alle Kind-Prozesse warten, deren Prozessgruppen-ID gleich der der Ayfrufers ist

> 0 auf den Kindprozess mit der angegebenen `pid` warten

Parameter **options**: bitweises ODER folgender Konstanten

- **WNOHANG**: nicht blockieren, falls kein Kindprozess terminierte ist
- **WUNTRACED**: auch bei gestoppten Kindprozessen zurückkehren

Ansonsten wird der dritte Parameter einfach auf 0 gesetzt!

Beispiel mit **waitpid()**

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
  pid_t pid;

  if ( (pid = fork() ) < 0 )
    perror("fork"), exit(1);

  if (pid == 0) {
    printf("Child -- my pid is %d\n", (int) getpid());
    sleep(10);
    printf("Child -- going to terminate!\n");
    return 0;
  } else {
    // Parent
    pid_t child_pid;
    int status;
    if ((child_pid = waitpid(pid, & status, WUNTRACED )) < 0)
        perror("wait"), exit(2);
    // WUNTRACED: return for children which are stopped, and
    // whose status has not been reported

    printf("Parent -- child with pid %d finished, ", (int) child_pid);
    if WIFEXITED(status)
      printf("with exit value %d\n", WEXITSTATUS(status) );
    if WIFSIGNALED(status)
      printf("caused by signal %d\n", WTERMSIG(status) );
    if WIFSTOPPED(status)
      printf("not really, is stopped by %d and now dead!\n",
          WSTOPSIG(status) );
    printf("Parent -- going to terminate!\n");
    return 0;
  }
}
```


Aufgaben des Kernel, wenn ein Prozess **exit()** aufruft:

- Wartet der Erzeuger-Prozess mit **wait()**, so wird er von der Termination des Kind-Prozesses benachrichtigt (*wait()* kehrt zurück). Im Argument von *wait()* wird der *exit*-Status (das Argument in *exit()*) geliefert, als Rückgabewert die PID des Kind-Prozesses.
- Wartet der Erzeuger-Prozess nicht, so wird der Kind-Prozess als **Zombie**-Prozess markiert. Der Kernel gibt dessen Ressourcen frei, bewahrt aber den *exit*-Status (Beendigungsstatus) in der Prozesstabelle auf.
- Sind *process ID, process group ID, terminal group ID* des terminierenden Prozesses alle gleich, so wird das Signal **SIGHUP** an jeden Prozess mit gleicher *process group ID* gesandt.

Wenn der Erzeuger vor dem Kind-Prozess terminiert:

- Die PPID der Kind-Prozesse wird auf 1 gesetzt, der *init*-Prozess ”erbt” die ”Waisen”.
  Der *init*-Prozess terminiert nie, und wenn doch, so wäre es im Multiuser-Betrieb nicht mehr möglich, dass sich weitere Benutzer anmelden. In 4.3BSD würde es zu einem automatischen *reboot* kommen.

Wenn ein Prozess mit **exit()** terminiert, so wird dem Erzeuger das Signal **SIGCLD** (**SIGCHLD**) zugeleitet (default-Reaktion: ignorieren). In System V ist es möglich, dass ein Prozess verhindert, dass seine Kind-Prozesse zu *Zombies* werden; dazu hat er nur den Aufruf

```
signal(SIGCLD, SIG_IGN)
```

abzusetzen; der *exit*-Status der Kindprozesse wird bei deren Termination nicht weiter aufbewahrt.

2.5 Bootstrapping

- Henne oder Ei?

  Alle Aktionen in einem *UNIX* System erfolgen durch **Prozesse**. Für alle Prozesse besteht eine Erzeuger-Kindprozeß-Beziehung.

  Bleibt die Frage, woher kommt der erste Prozess, die Wurzel dieses Prozessbaums?
- **Bootstrapping**

  Das Starten eines Betriebssystems heißt *Bootstrapping*.

  Für *UNIX* fallen darunter alle Aktionen vom Stromeinschalten bis zum Erreichen des stabilen Zustands in dem Prozess 1 läuft. Anschließend können alle weiteren Aktionen mit Prozessen und nach den durch die System Calls festgelegten Regeln erfolgen.
- Phase 0 Hardware

  Das Einschalten der Stromversorgung bewirkt verschiedene Aktionen, die sehr von der Hardware und der Architektur des Rechners abhängen. Alle haben aber das gleiche Ziel: Selbsttest und Grundinitialisierung der einzelnen Komponenten. Die Hardware befindet sich danach in einem definierten Startzustand.

- Phase 1: First Stage Boot

  Die Hardware verfügt über einige "festeingebaute" Routinen, die in der Lage sind, einen Boot-Block von der Platte zu lesen und zur Ausführung zu bringen.

  Der Boot-Block enthält ein **loader** - Programm. Dieses Programm muß mit der Console kommunizieren können und bereits über das *UNIX* File System Bescheid wissen. Falls der Platz im Boot-Block für ein Programm mit diesen Fähigkeiten nicht ausreicht, muß das minimale *Boot Loader* Programm ein weiteres, größeres *Loader* Programm laden und ausführen können.

- Phase 2: Second Stage Boot

  Aufgabe des **loader** - Programms ist es, den Kernel, meist die Datei */unix*, von der Platte in den Hauptspeicher zu laden und zu starten.

  Anschließend beginnt *UNIX* zu "leben".

  Loader Programme sind meist trickreiche Assemblerprogramme. Im Boot-Block können nur wenige Anweisungen untergebracht werden, die müssen aber komplexe, Hardware-nahe Aufgaben erledigen.

- Phase 3: Standalone

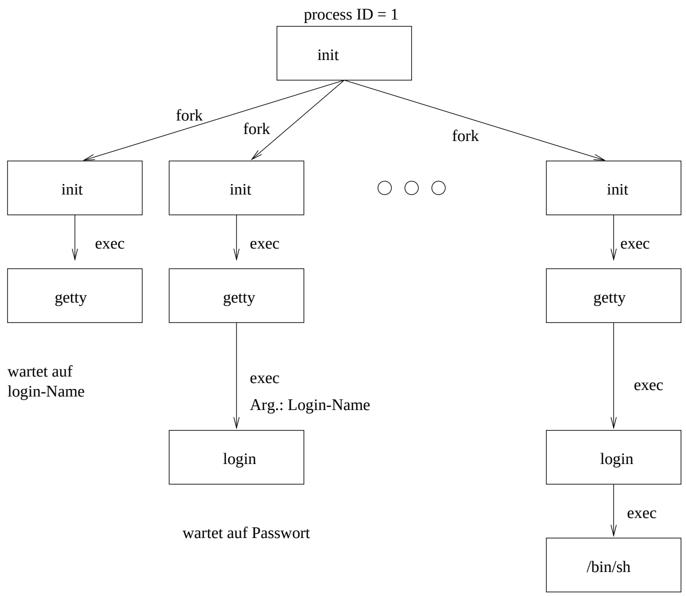
Sobald der *UNIX* Kernel gestartet wurde, übernimmt er die gesamte Kontrolle über den Rechner. Der Kernel kann jetzt alles weitere aus eigener Kraft erledigen *(Standalone)*.
In seiner Startphase
☞ setzt er die Interruptvektoren im *low mem*,
☞ initialisiert die Speicherverwaltungs-Hardware,
☞ baut seine Tabellen (Prozess-, Open File-, Inode-, usw.) auf
☞ führt eine *mount*-ähnliche Operation für das *root* File System aus...
Nun fehlt noch etwas "Magie", um den ersten Prozess zu erschaffen. Die Prozessverwaltung generiert in ihrer Startphase den **Prozess 0** "von Hand". Dieser Prozess, meist **swapper** genannt, besteht nur aus dem System-Daten Bereich: Slot 0 in der Prozesstabelle plus *per process data region*. Er besitzt keinen Text oder User-Daten Bereich. Dafür existiert er während der gesamten Systemlaufzeit und ist für das *Scheduling* zuständig. Er benötigt hierfür nur Instruktionen und Daten aus dem Kernel-Adreßraum.
Da Prozess 0 eigentlich kein echter Prozess ist, erschafft der Kernel auch **Prozess 1** manuell. Soweit als möglich benutzt oder imitiert der Kernel hier bereits den *fork* Mechanismus, um den Prozess 1 vom Prozess 0 abzuspalten. Prozess 1 erhält jedoch einen ganz regulären Context und kann anschließend vom Scheduler als normaler Prozess zur Ausführung gebracht werden. Sein "hand crafted" Text Bereich enthält einzig die Anweisung

```
execl( ''/etc/init'', ''init'', 0 )
```

Nach dem *exec* System Call läuft im Context des Prozesses 1 das Programm */etc/init* und Prozess 1 heißt nun **init - Prozess**.

2.6 Der init-Prozess

- *user Id* 0
- Operationen:
  1. 4.3BSD
     init führt das Shell-Skript */etc/rc* aus. Dabei werden u.a. einige Daemon-Prozesse gestartet. *init* entnimmt der Datei */etc/ttys*, welche Terminals für den Multiuser-Betrieb aktiviert werden müssen.
  2. System V
     init liest die Datei */etc/inittab*, in der spezifiziert ist, was zu tun ist. Zum normalen Multiuser-Betrieb wird die Datei */etc/rc* ausgeführt; dieses Programm startet die meisten Daemon-Prozesse. Nach dessen Beendigung werden wie in */etc/inittab* definiert die angeschlossenen Terminals aktiviert. (mehr dazu in einem *Administrator Reference Manual*).



- *getty* setzt die Übertragungsgeschwindigkeit des Terminals, gibt irgendeine Begrüßungsformel aus und wartet auf Eingabe eines login-Namens.
- Login-Name eingegeben → *exec(/bin/login)*
- *login* sucht den eingegebenen Login-Namen in der Paßwortdatei und fordert die Eingabe eines Paßwortes.

Alle bis hier ausgeführten Programme (*init, getty, login*) laufen als Prozesse mit *user ID* und *effective user ID* 0 (*superuser*) - mit dem System Call *exec* ändert sich die *process ID* nicht.
Danach wird das *current working directory* auf den entsprechenden Eintrag für das *login directory* aus der Paßwortdatei gesetzt.
Group ID und *user ID* (in dieser Reihenfolge) werden via *setgid* und *setuid* wie in der Paßwortdatei definiert gesetzt.
Über *exec* wird das in der Paßwortdatei spezifizierte Programm gestartet (falls keine Angabe: */ bin/ sh*).

Falls der sich anmeldende Benutzer nicht der Superuser ist (*login*-Name meist *root*):

- *setgid* und *setuid* reduzieren Prozessprivilegien
- beide System Calls sind dem Superuser vorbehalten (daher diese Reihenfolge)

Kapitel 3

Signale

3.1 Grundlegendes

- Nachricht an einen Prozess bzgl. eines eingetretenen Ereignisses
- auch als "Software-Interrupt bezeichnet
- tritt i.a. asynchron auf

Signale werden geschickt

- von einem Prozess an einen anderen Prozess (oder auch an sich selbst)
- vom Kernel an einen Prozess

Signale werden identifiziert

- intern über Integers (**Signalnummer**)
- "extern" über Namen → **signal.h**

Reaktion auf Signale:

- *default* → signal.h
- kann mit dem System Call **signal()** und der Bchereifunktion **sigaction()** eingestellt werden (s.u.)

Bedingungen, die Signale erzeugen:

1. System Call **kill**: damit kann ein Prozess sich oder einem anderen Prozess ein Signal senden (führt nicht notwendig zur Termination)
   int kill(int pid, int sig);
   Der sendende Prozess muss entweder ein superuser-Prozess sein oder der sendende und empfangende Prozess müssen **dieselbe** effektive userId haben.
   - Ist das pid-Argument 0, so geht das Signal an alle Prozesse in der Prozess-Gruppe des Senders
   - Ist das pid-Argument -1 und der Sender nicht der Superuser, so geht das Signal an alle Prozesse, deren *real user ID* gleich der effektive user ID des Senders ist

   - Ist das pid-Argument -1 und der Sender der Superuser, so geht das Signal an alle Prozesse ausgenommen die Superuser-Prozesse (i.a. PID's 0 oder 1)
   - Ist das pid-Argument negativ, aber ungleich -1, so geht das Signal an alle Prozesse, deren Prozess-Gruppen-Nummer gleich dem Absolutbetrag von pid ist
   - Ist das sig-Argument 0, so wird nur eine Fehler-Prüfung durchgeführt, aber kein Signal geschickt (z.B. zur Prüfung der Gültigkeit des pid-Arguments)
2. Kommando **kill**: nutzt den System Call **kill()** (s.o. und **man**)
3. Tastatureingaben, z.B.:
   - **ctrl-c** (oder **delete**) beendet einen im Vordergrund laufenden Prozess (→ **SIGINT**), genauer alle Prozesse in der Kontrollgruppe dieses Terminals - vom Kernel verschickt
   - **c**trl-\\) erzeugt **SIGQUIT**
   - **ctrl-z** hält einen im Vordergrund laufenden Prozess an
     (→ **SIGSTOP**)
     kann mit **SIGCONT** fortgesetzt werden
4. Hardware-Bedingungen, z.B.:
   Gleitpunkt-Arithmetik-Fehler (*Floating Point Exception*) (**S**IGFPE)
   Adreßraum-Verletzungen (*Segmentation Violation*) (**S**IGSEGV)
5. Software-Bedingungen, z.B.:
   Schreiben in eine Pipe, an der kein Prozess zum Lesen "hängt" (**S**IGPIPE).

3.2 Reaktion auf Signale: signal()

1. Ein Prozess kann eine Funktion definieren, die bei Eintreten eines bestimmten Signals ausgeführt werden soll (**s**ignal handler).
2. Signale (außer **S**IGKILL und **S**IGSTOP) können ignoriert werden.

System Call *signal*:

```
#include <signal.h>

int (*signal (int sig, void (*func) (int))) (int);
/* ANSI C signal handling */
```

Also:

- *signal* ist eine Funktion, die einen Zeiger auf eine Funktion zurückliefert, die selbst eine *int* zurückliefert (die bisherige Reaktion auf das Signal oder bei Fehler **S**IG_ERR)
- Das erste Argument ist die Signalnummer (Makro aus *signal.h*), das zweite (*func*) ist Zeiger auf eine Funktion, die *void* liefert (die neue Reaktion auf das Signal)

Wegen der detaillierteren Reaktionsmöglichkeit und einiger anderen Fußangeln ist die Funktion **sigaction()** vorzuziehen!

3.3 Reaktion auf Signale: sigaction()

```
NAME
     sigaction - detailed signal management

SYNOPSIS
     #include <signal.h>

     int sigaction(int sig, const struct sigaction *newact,
          struct sigaction *oldact);

DESCRIPTION
     The sigaction() function allows the calling process to
     examine or specify the action to be taken on delivery of a
     specific signal.  (See signal(5) for an explanation of  gen-
     eral signal concepts.)

     The sig argument specifies the signal and  can  be  assigned
     any of the signals specified in signal(5) except SIGKILL and
     SIGSTOP.

     If the argument newact is not NULL, it points  to  a  structure
     specifying  the  new action to be taken when delivering sig.
     If the argument oldact is not NULL, it points to  a  structure
     where  the  action  previously  associated with sig is to be
     stored on return from sigaction().

     The sigaction structure includes the following members:

          void        (*sa_handler)();
          void        (*sa_sigaction)(int, siginfo_t *, void *);
          sigset_t    sa_mask;
          int         sa_flags;

     The sa_handler member identifies the action to be associated
     with  the  specified  signal,  if  the  SA_SIGINFO flag (see
     below) is cleared in the sa_flags  field  of  the  sigaction
     structure. It may take any of the values specified in sig-
     nal(5) or that of a user specified signal handler. If the
     SA_SIGINFO   flag   is   set  in  the  sa_flags  field,  the
     sa_sigaction field specifies a signal-catching function.

     The sa_mask member specifies a set of signals to be  blocked
     while  the signal handler is active.  On entry to the signal
     handler, that set of signals is added to the set of  signals
     already  being  blocked  when  the  signal is delivered.  In
     addition, the signal that caused the handler to be  executed
     will  also  be  blocked, unless the SA_NODEFER flag has been
     specified.  SIGSTOP and SIGKILL cannot be blocked
     (the system silently enforces this restriction).

     The sa_flags member specifies a set of flags used to  modify
     the delivery of the signal.  It is formed by a logical OR of
     any of the following values (more: see man page):
```



```
          SA_RESTART          If set and the  signal  is  caught,
                              certain  functions  that are inter-
                              rupted by  the  execution  of  this
                              signal's  handler are transparently
                              restarted by  the  system;  namely,
                              read(2) or write(2) on slow devices
                              like terminals, ioctl(2), fcntl(2),
                              wait(2), and waitid(2).  Otherwise,
                              that  function  returns  an   EINTR
                              error.

          SA_NOCLDWAIT        If set and sig equals SIGCHLD,  the
                              system   will   not  create  zombie
                              processes when children of the cal-
                              ling  process exit.  If the calling
                              process   subsequently   issues   a
                              wait(2), it blocks until all of the
                              calling process's  child  processes
                              terminate, and then returns -1 with
                              errno set to ECHILD.

RETURN VALUES
     On success, sigaction() returns 0.  On failure,  it  returns
     -1  and  sets  errno  to indicate the error.  If sigaction()
     fails, no new signal handler is installed.

ERRORS
     see man page

SEE ALSO
     kill(1), intro(2),  exit(2),  fcntl(2),  ioctl(2),  kill(2),
     pause(2),   read(2),  sigaltstack(2),  sigprocmask(2),  sig-
     send(2),  sigsuspend(2),   wait(2),   waitid(2),   write(2),
     signal(3C),  sigsetops(3C),  thr_create(3T),  attributes(5),
     siginfo(5), signal(5), ucontext(5)
```

Beispiel:

```
/* -------------------- sign.h ------------------------ */
/* Signal Handler                                       */

#include <signal.h>

typedef void (*Sigfunc)(int);

#ifndef SIGN_H
#define SIGN_H
/* avoid multiple includes */

Sigfunc ignoresig(int);
/* interrupt und quit ignorieren */
Sigfunc entrysig(int);
/* interrupt und quit restaurieren */
#endif



/* ------------------------ sign.c ------------------------ */

# include "sign.h"
# include <stdio.h>

void myignore(int sig){
  printf("signal handler: SIGNAL = >%d<\n",sig);
  return;
}

struct sigaction newact, oldact;

Sigfunc ignoresig(int sig) {
   static int first = 1;
   newact.sa_handler = myignore;
   if (first) {
      first = 0;
      if (sigemptyset(&newact.sa_mask) < 0)
          return SIG_ERR;
      /* Durch diese Initialisierung der Komponente sa_mask mit
       * der leeren Menge wird bewirkt, dass waehrend der Aus-
       * fuehrung der installierten Signalbehandlungsfunktion
       * mit Ausnahme des im Argument sig angegebenen Signals
       * keine weiteren Signale von der Zustellung durch den
       * Systemkern zurueckgehalten werden
       */

       newact.sa_flags = 0;
       newact.sa_flags |= SA_RESTART;
       if (sigaction(sig, &newact, &oldact) < 0)
              return SIG_ERR;
       else
              return oldact.sa_handler;
```



```
   } else {
       if (sigaction(sig, &newact, NULL) < 0)
              return SIG_ERR;
       else
              return NULL;
   }
}

Sigfunc entrysig(int sig)   {
   if (sigaction(sig, &oldact, NULL) < 0 )
      return SIG_ERR;
   else
      return NULL;
}



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

int main() {
   int sleep_time;
   /*install reaction*/
   if ( (ignoresig(SIGINT) == SIG_ERR) ||
         (ignoresig(SIGQUIT) == SIG_ERR) ) {
      perror("ignoresig");
      exit(1);
   }
   printf("Break?\n");
   while(1) {   /*loop forever*/
      sleep_time = 5;
      do {
         printf("go asleep for %d sec\n", sleep_time);
         sleep_time = sleep(sleep_time);
      } while(sleep_time != 0);
                        ;
      printf("And now?\n");
      /*restore reaction */
      if ( (entrysig(SIGINT) == SIG_ERR) ||
           (entrysig(SIGQUIT) == SIG_ERR) ) {
         perror("entrysig");
         exit(2);
      }
   }
   exit(0);
}
```

Ausführung:

```
hypatia$ show
```

```
Break?
go asleep for 5 sec
#ctrl-c
signal handler: SIGNAL = >2<
go asleep for 4 sec
#ctrl-\
Quit
hypatia$ show
Break?
go asleep for 5 sec
# ctrl-c
signal handler: SIGNAL = >2<
go asleep for 4 sec
#ctrl-c
signal handler: SIGNAL = >2<
go asleep for 2 sec
And now?
go asleep for 5 sec
#ctrl-c

hypatia$
```

Modifikation wie oben:

```
/* ------------------------ main1.c --------------------- */
# include <stdio.h>
# include <unistd.h>
# include "sign.h"

int main() {
   int sleep_time;
   if (ignoresig(SIGINT) == SIG_ERR) /*set new reaction*/ {
      perror("ignoresig");
      exit(1);
   }
   printf("Break ?\n");
   while(1) {   /*do forever*/
      sleep_time = 5;
      do {
          printf("go asleep for %d s\n", sleep_time);
          sleep_time = sleep(sleep_time);
          printf("sleep_time: %d\n", sleep_time);
      } while( (sleep_time != 0) );

      printf("and now?\n");
   }
   exit(0);
}
```

Ausführung:

```
hypatia$ show1
```



```
Break ?
go asleep for 5 s
# ctrl-c
signal handler: SIGNAL = >2<
sleep_time: 4
go asleep for 4 s
# ctrl-c
signal handler: SIGNAL = >2<
sleep_time: 1
go asleep for 1 s
sleep_time: 0
and now?
go asleep for 5 s
# ctrl-c
signal handler: SIGNAL = >2<
sleep_time: 5
go asleep for 5 s
# ctrl-c
signal handler: SIGNAL = >2<
sleep_time: 3
go asleep for 3 s
# ctrl-\
Quit
hypatia$
```

3.4 Anwendungsbeispiele

Im folgenden soll ein Prozess Kind-Prozesse erzeugen, die vor dem Erzeuger-Prozess terminieren und auf die der Erzeuger auch nicht wartet - er wird allerdings über ein Signal **S**IGCHLD / **S**IGCLD benachrichtigt. Die Kind-Prozesse werden zu Zombie-Prozessen.

```
/* -------------------- sign.h ------------------------ */

#include <signal.h>

typedef void (*Sigfunc)(int);

#ifndef SIGN_H
#define SIGN_H
/* avoid multiple includes */

Sigfunc ignoresig(int);
Sigfunc entrysig(int);
#endif



/* ------------------------ sign.c ------------------------ */

# include "sign.h"
# include <stdio.h>

void myignore(int sig){
  printf("signal handler: SIGNAL = >%d<\n",sig);
  return;
}

struct sigaction newact, oldact;

Sigfunc ignoresig(int sig) {
   static int first;
   newact.sa_handler = myignore;
   first = 1;
   if (first) {
      first = 0;
      if (sigemptyset(&newact.sa_mask) < 0)
          return SIG_ERR;

      newact.sa_flags = 0;
      newact.sa_flags |= SA_RESTART;

      if (sigaction(sig, &newact, &oldact) < 0)
              return SIG_ERR;
      else
              return oldact.sa_handler;
   } else {
      if (sigaction(sig, &newact, NULL) < 0)
              return SIG_ERR;
```



```
      else
              return NULL;
   }
}

Sigfunc entrysig(int sig)   {
   if (sigaction(sig, &oldact, NULL) < 0 )
      return SIG_ERR;
   else
      return NULL;
}



/* ------------------------ main.c ------------------------ */
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <signal.h>
# include "sign.h"

int main() {
   char cmdstr[128];
   int pid[3], i = 3;
   if (ignoresig(SIGCHLD) == SIG_ERR) {
      perror("ignoresig");
      exit(1);
   }

   printf("Create child processes:\n");
   while(i>0) {
   switch(pid[3-i]=fork()) {
     case -1: perror("fork");
              exit(1);
     case 0 : printf("I'm child %d with PID = %ld!\n", 4-i, getpid());
              sleep(5);
              exit(0);
     default:
              sleep(1);
              if (kill(pid[3-i],2) < 0 ) {
                  perror("kill");
                  exit(1) ;
              }
   }
   i--;
  }
  sleep(5);
  printf("Process table (extract child processes above): \n");
  sprintf(cmdstr,"ps -lp %d,%d,%d", pid[0], pid[1], pid[2]);
  system(cmdstr);
  sleep(1);
  printf("Parent: going to exit!\n");
  exit(0);
}
```

Ausführung:

```
hypatia$ show
Create children:
I'm child 1 with PID = 18542!
I'm child 2 with PID = 18543!
signal handler: SIGNAL = >17<
I'm child 3 with PID = 18544!
signal handler: SIGNAL = >17<
signal handler: SIGNAL = >17<
Process table (extract):
18541  p2 S    0:00 show
18542  p2 Z    0:00 (show <zombie>)
18543  p2 Z    0:00 (show <zombie>)
18544  p2 Z    0:00 (show <zombie>)
18545  p2 S    0:00 sh -c sleep 1; ps | grep show
18548  p2 R    0:00 sh -c sleep 1; ps | grep show
signal handler: SIGNAL = >17<
Parent: going to exit!
hypatia$
```

Vermeidung der Zombies

```
/* -------------------- sign1.h ------------------------ */

#include <signal.h>

typedef void (*Sigfunc)(int);

#ifndef SIGN_H
#define SIGN_H
/* avoid multiple includes */

Sigfunc ignoresig(int);
Sigfunc entrysig(int);
#endif



/* ------------------------ sign1.c ------------------------ */

# include "sign.h"
# include <stdio.h>
# include <sys/wait.h>

void myignore(int sig){
  int status;
  printf("signal handler: SIGNAL = >%d<\n",sig);
  waitpid(0, &status, WNOHANG);
  return;
}


struct sigaction newact, oldact;
```



```
Sigfunc ignoresig(int sig) {
   static int first;
   newact.sa_handler = myignore;
   first = 1;
   if (first) {
      first = 0;
      if (sigemptyset(&newact.sa_mask) < 0)
          return SIG_ERR;

      newact.sa_flags = 0;
      newact.sa_flags |= SA_RESTART;

      if (sigaction(sig, &newact, &oldact) < 0)
              return SIG_ERR;
      else
              return oldact.sa_handler;
   } else {
      if (sigaction(sig, &newact, NULL) < 0)
              return SIG_ERR;
      else
              return NULL;
   }
}

Sigfunc entrysig(int sig)   {
   if (sigaction(sig, &oldact, NULL) < 0 )
      return SIG_ERR;
   else
      return NULL;
}



/* ------------------------ main1.c ------------------------ */
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include "sign1.h"

int main() {
   char cmdstr[128];
   int pid[3], i = 3;
   if (ignoresig(SIGCLD) == SIG_ERR) {
      perror("ignoresig");
      exit(1);
   }

   printf("Create child processes:\n");
   while(i>0) {
   switch(pid[3-i]=fork()) {
     case -1: perror("fork");
              exit(1);
     case 0 : printf("I'm child %d with PID = %ld!\n", 4-i, getpid());
```

```
              sleep(5);
              exit(0);
     default: sleep(1);
              kill(pid[3-i],9);
   }
   i--;
  }
  sleep(5);
  printf("Process table (extract child processes above): \n");
  sprintf(cmdstr, "ps -lp %d,%d,%d", pid[0], pid[1],pid[2]);
  system(cmdstr);
  sleep(1);
  printf("Parent: going to exit!\n");
  exit(0);
}
```

Ausführung:

```
hypatia$ show1
Create children:
I'm child 1 with PID = 334!
I'm child 2 with PID = 335!
signal handler: SIGNAL = >17<
I'm child 3 with PID = 336!
signal handler: SIGNAL = >17<
signal handler: SIGNAL = >17<
Process table (extract):
  333  p2 S    0:00 show1
  337  p2 S    0:00 sh -c sleep 1; ps | grep show
  340  p2 S    0:00 grep show
signal handler: SIGNAL = >17<
Parent: going to exit!
hypatia$
```


Weitere Anwendung: Signale zur "Kommunikation" – SIGUSR1 und SIGUSR2

```
/* -------------------- sign.h ------------------------ */

#include <signal.h>

typedef void (*Sigfunc)(int);

#ifndef SIGN_H
#define SIGN_H
/* avoid multiple includes */

Sigfunc ignoresig(int);
Sigfunc entrysig(int);
#endif



/* ------------------------ sign.c ------------------------ */

# include "sign.h"
# include <stdio.h>
# include <siginfo.h>

struct sigaction newact, oldact;
siginfo_t info;

void my_handler(int sig){
  printf("signal handler: SIGNAL = >%d<\n",sig);
  return;
}


Sigfunc ignoresig(int sig) {
   static int first = 1;
   newact.sa_handler = my_handler;
   if (first) {
      first = 0;
      if (sigemptyset(&newact.sa_mask) < 0)
          return SIG_ERR;

      newact.sa_flags = 0;
      newact.sa_flags |= SA_RESTART;
      /*SA_SIGINFO -> man sigaction */

      if (sigaction(sig, &newact, &oldact) < 0)
              return SIG_ERR;
      else
              return oldact.sa_handler;
   } else {
      if (sigaction(sig, &newact, NULL) < 0)
              return SIG_ERR;
      else
              return NULL;
```

```
   }
}

Sigfunc entrysig(int sig)   {
   if (sigaction(sig, &oldact, NULL) < 0 )
      return SIG_ERR;
   else
      return NULL;
}



/* ----------------------- main.c ----------------------- */
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <signal.h>
# include "sign.h"

int main() {
   int pid, ppid;
   if (ignoresig(SIGUSR1) == SIG_ERR) {
      perror("ignoresig");
      exit(1);
   }

   switch(pid=fork()) {
     case -1: perror("fork");
              exit(1);
     case 0 :
      printf("I'm child with PID = %ld!\n", getpid());
              ppid=getppid();
              sleep(1);
              if (kill(ppid,SIGUSR1) < 0 ) {
                 perror("kill");
                 exit(1);
              }
              sleep(10);
              exit(0);
     default: printf("I'm parent with PID = %ld\n", getpid());
              sleep(1);
   }
  sleep(1);
  printf("Parent: going to exit!\n");
  exit(0);
}
```


Erläuterungen zu einigen Signalen:
Generell mit Vorsicht zu betrachten, da system-abhängig System V $\longleftrightarrow$ BSD) und oft nicht sehr zuverlässig zu behandeln!

SIGALRM Ein Prozess kann mit **u**nsigned int alarm(unsigned int sec); eine Alarmuhr stellen. Der Parameter *sec* bestimmt, nach wievielen Sekunden (*real time*) der Kernel dem Prozess das *SIGALRM* Signal sendet (*software timeout*); ist der Wert 0, werden alle früher gestellten Alarmuhren ge*cancel*t. Der Rückgabewert ist die verbleibende Zeit des letzten Aufrufs (falls einer) von *alarm*.

Die Funktion *u*nsigned int sleep(unsigned int sec); (Prozess für *sec* Sekunden "schlafen" legen) nutzt *SIGALRM*.

SIGCLD Wird dem Erzeuger-Prozess gesandt, wenn ein von ihm erzeugter Prozess terminiert. In 4.3BSD besagt dieses Signal, dass sich der Status des Kind-Prozesses geändert hat (Termination, aber auch angehalten durch Signale).

SIGHUP Beim Schließen eines Terminals gesandt an alle Prozesse, für die dieses Terminal das Kontroll-Terminal ist. Terminiert ein **p**rocess group leader, so wird dieses Signal an alle Prozesse dieser Gruppe gesandt.

SIGPIPE Erzeugt, wenn ein Prozess versucht in eine FIFO-Datei oder Pipe zu schreiben, an der kein Prozess zum Lesen hängt.

SIGSTOP Zum gezielten Anhalten eines Prozesses, kann nicht ignoriert oder abgefangen werden; Fortsetzen des Prozesses mit Signal **S**IGCONT.

SIGUSR1 SIGUSR2 Zur "Kommunikation" zwischen Prozessen; die einzige Information für den Empfänger ist der Signaltyp, die Prozess ID des Senders kann der Empfänger nicht feststellen.

Zuverlässige Signale
In früheren UNIX-Versionen konnten Laufzeitbedingungen entstehen, die dazu führten, dass Signale verlorengingen, ein Ereignis konnte eintreten und ein Signal hervorrufen, das ein Prozess nicht registrierte. In System VR3 und 4.3BSD wurden Erweiterungen - leider nicht kompatibel - durchgeführt.

- Signal-Handler bleiben installiert, nachdem ein Signal eintrifft. In früheren Versionen wurde eine prozeß-definierte Aktion auf **S**IG_DFL zurückgesetzt, bevor der Signal-Handler aufgerufen wurde; dies bedeutet, dass das erneute Eintreffen desselben Signals verloren geht, bevor der Prozess in der Lage ist, den **s**ignal() Aufruf erneut abzusetzen.
- Ein Prozess muss Signale abfangen können - aber sie sollen nicht einfach verschwinden (*discarded*); wenn das gewünscht ist, dann explizit über die **S**IG_IGN Aktion. Statt dessen sollte das Signal aufbewahrt werden und dann ausgeliefert werden, wenn der Prozess bereit ist. Dazu gibt es in 4.3BSD das Konzept des *blocking*, in System V des *holding* von Signalen.
- Während ein Signal einem Prozess zugestellt wird, wird dieses "gehalten" (blockiert); d.h., dass wenn ein Signal ein zweites Mal erzeugt wird, während der Prozess noch das erste Auftreten bearbeitet, der Signal Handler kein zweites Mal aufgerufen wird. Statt dessen wird das zweite Auftreten aufbewahrt; nur wenn der Signal Handler von der Behandlung des ersten Auftretens normal zurückkehrt, wird er ein zweites Mal zur Behandlung des zweiten Auftretens aufgerufen.

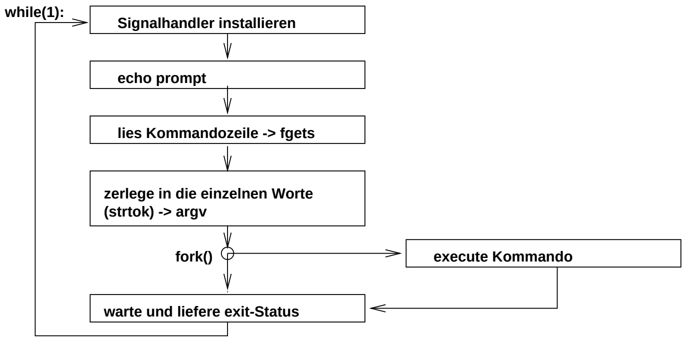
Mehr zu Signalen: /Stevens90/

3.5 Implementierung einer Mini-Shell

Leistungsmerkmale:

- Ausführung eines einfachen Kommandos im Vordergrund, das von *stdin* eingebenen wird (keine I/O-Umlenkung, Quotierung, Substitution u.dgl.)
- Eine Kommandozeile besteht aus Worten, die durch Blank oder Tab getrennt sind
- Möglichkeiten, Kommandos mit *ctrl-c* bzw. *ctrl-* abzubrechen (dazu muss die *minishell* gegen diese Signale immun gemacht werden

Aufbau:



Das Aufsammeln der Worte geschieht in der Funktion **getargs**. Die Standardfunktion **s**trtok() (bei dieser Funktion ist Vorsicht angebracht: siehe Manual) holt das nächste Symbol aus einer Zeichenkette. Beim ersten Aufruf wird ein Zeiger auf die Zeichenkette zur Verfügung gestellt, die weiteren Aufruf werden durch **NULL** als erstes Argument unterschieden. Das zweite Argument gibt die Folge der Zeichen (hier: Blank, tab und newline) an, die als Begrenzer dienen.


```
/* -------------------- sign.h ------------------------ */
/* Signal Handler                                       */

#include <signal.h>

typedef void (*Sigfunc)(int);

#ifndef SIGN_H
#define SIGN_H
/* avoid multiple includes */

Sigfunc ignoresig(int);
/* interrupt und quit ignorieren */
Sigfunc entrysig(int);
/* interrupt und quit restaurieren */
#endif



/* ------------------------ sign.c ------------------------ */

# include "sign.h"
# include <stdio.h>

void myignore(int sig){ return; }

struct sigaction newact, oldact;

Sigfunc ignoresig(int sig) {
   static int first = 1;
   newact.sa_handler = myignore;
   if (first) { first = 0;
      if (sigemptyset(&newact.sa_mask) < 0)
          return SIG_ERR;
      newact.sa_flags = 0;
      newact.sa_flags |= SA_RESTART;
      if (sigaction(sig, &newact, &oldact) < 0)
              return SIG_ERR;
      else
              return oldact.sa_handler;
   } else {
      if (sigaction(sig, &newact, NULL) < 0)
              return SIG_ERR;
      else return NULL;
   }
}

Sigfunc entrysig(int sig)   {
   if (sigaction(sig, &oldact, NULL) < 0 )
      return SIG_ERR;
   else return NULL;
}
```

```
/* ------------------------ args.h ------------------------ */
#ifndef ARGS_H
#define ARGS_H
#define MAXARG 20
#define ARG_LENGTH 20
#define LINE_LENGTH 256
int getargs(int *argcp, char **argv, int max);
#endif
```

```
/* ------------------------ args.c ------------------------ */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "args.h"

int getargs(int * argcp, char **argv, int max) {
   /*Argumente lesen und trennen */
   static char cmd[LINE_LENGTH];
   char * cmdp, *strtok();   /*strtok: naechstes Symbol */
   int i;
   if (fgets(cmd,LINE_LENGTH,stdin) == NULL) {
       printf("terminated\n\n"); exit(0);
   };
   cmdp = cmd;
   for ( i=0; i <= max; i++) {
      if ( (argv[i] = strtok(cmdp, " \t\n") ) == NULL)
         break;
      cmdp = NULL; /*strtok soll weitergehen*/
   }
   if (i > max) {
      fprintf(stderr,"Too many args\n");
      return 0;   /*False*/
   }
   * argcp = i;
   return 1;   /* TRUE*/
}
```



```
/* ------------------------ run.h ------------------------ */
#ifndef RUN_H
#define RUN_H
#define lowbyte(w) ( (w) & 0xff)
#define highbyte(w) lowbyte( (w) >> 8 & 0xff)
int run_cmd(int argc, char **argv);
#endif
```

```
/* ------------------------ run.c ------------------------ */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "run.h"
#include "sign.h"

int run_cmd(int argc, char **argv) {
   /*Kommando ausf. u. warten*/
   int status, done, pid;

   switch ( pid=fork() ) {
     case -1:
         perror("fork"); return -1;
     case 0:
/*        if ( (entrysig(SIGINT) == SIG_ERR) |
             (entrysig(SIGQUIT) == SIG_ERR) ) {
               perror("entrysig");
               return -1;
          }
        nicht noetig an dieser Stelle, da die
        Signaleinstellung nicht vererbt wird!
*/
        execvp(argv[0], argv);
        perror("execvp"); exit(2);

     default: do {
                 if ( (done = wait(&status)) < 0) {
                    perror("wait"); exit(2);
                 }
              } while (done != pid);

      if ( lowbyte(status) == 0) /*normal termination*/
        return highbyte(status);
      else
        return lowbyte(status);
   }
}
```

```
/* ------------------------ main.c ------------------------ */

/* Eine Mini-Shell: minishell.c */
#include <stdio.h>
#include <stdlib.h>
#include "sign.h"
#include "args.h"
#include "run.h"

int main() {
   char * argv[MAXARG+1], * prompt;
   int argc, status;
   if ( (ignoresig(SIGINT) == SIG_ERR) ||
        (ignoresig(SIGQUIT) == SIG_ERR) ) {
        perror("ignoresig"); exit(1);
   };

   prompt = "minish> ";
   while (1) {
      printf("%s", prompt);
      if (!getargs(&argc, argv, MAXARG) || argc == 0)
         continue;
      status = run_cmd(argc, argv);
      printf("status: %d\n", status);
   }
   exit(0);
}
```

```
# ------------ makefile for minishell ----------------

minish:    main.o run.o args.o sign.o
    gcc -o minish -Wall main.o run.o args.o sign.o
    # ausfuehrbares Programm: minish
    gcc -o sleepwell sleepwell.c
    # sleepwell: Testprogramm fuer Signale


main.o:    main.c sign.h run.h args.h
    gcc -Wall -c main.c

sign.o:    sign.h sign.c
    gcc -Wall -c sign.c

run.o:    run.h sign.h run.c
    gcc -Wall -c run.c

args.o:    args.h args.c
    gcc -Wall -c args.c

.PHONY: clean realclean
clean:
    rm -f *.o core
```



```
realclean:
    rm -f *.o minish sleepwell core
```

Ausführung:

```
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -c run.c
gcc -Wall -c args.c
gcc -Wall -c sign.c
gcc -o minish -Wall main.o run.o args.o sign.o
# ausfuehrbares Programm: minish
gcc -o sleepwell sleepwell.c
# sleepwell: Testprogramm fuer Signale

hypatia$ minish
minish> ls
args.c      main.c   minish       run.o     sign.o
args.h      main.o   run.c        sign.c     sleepwell
args.o      makefile   run.h        sign.h     sleepwell.c
status: 0
minish> pwd
/home/swg/soft/soft2.00/kap14dir/14/bsp/minishell
status: 0
minish> s;ee
execvp: No such file or directory
status: 2
minish> sleepwell
# ctrl-c
status: 2
minish> sleepwell
# ctrl-\
status: 3
minish> terminated

hypatia$
```

3.6 Implementierung einer Midi-Shell

Obige "Mini"-Shell soll um folgende Leistungsmerkmale erweitert werden:

- Ausführung eines einfachen Kommandos auch im **H**intergrund (dann aber immun gegen ctrl-c)
- **I**/O-Umlenkung der Standardverbindungen (0, 1, 2)

Die Worte auf der Kommandozeile müssen jetzt differenzierter betrachtet werden. Zulässige Symbole (*token*) der Kommandozeilen-Sprache sind:

- `> (T_GT)`

  Ausgabeumlenkung - danach muss ein Dateiname kommen
- `>> (T_GTGT)`

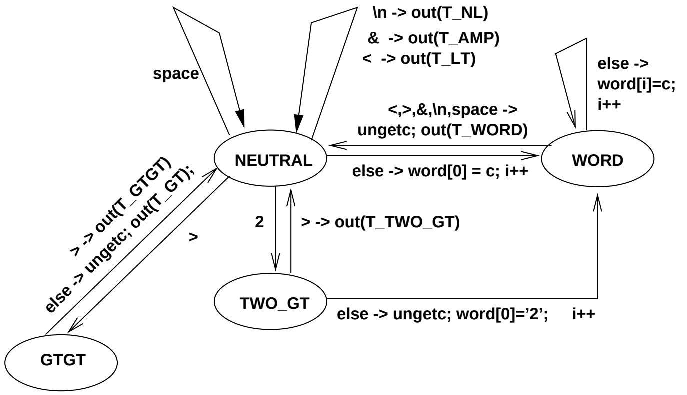
  Ausgabeumlenkung zum Anfügen - danach muss ein Dateiname kommen
- `< (T_LT)`

  Eingabeumlenkung - danach muss ein Dateiname kommen
- `2> (T_TWO_GT)`

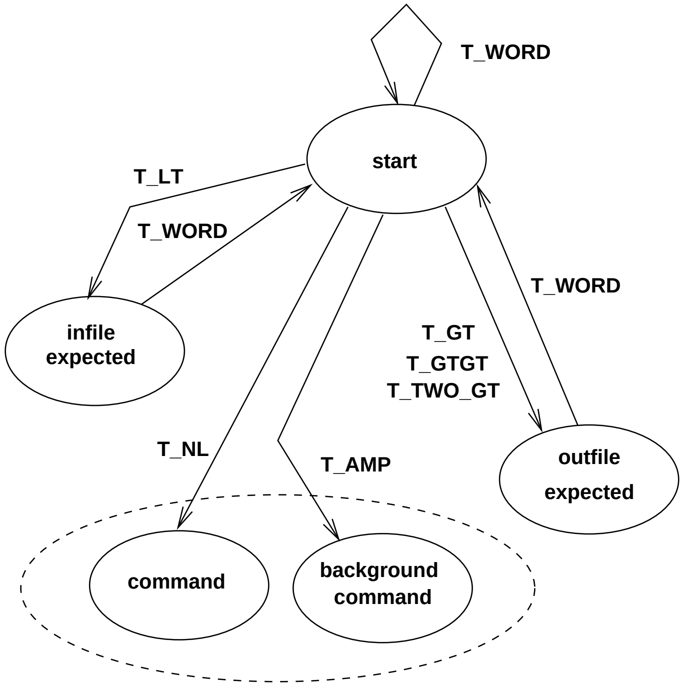
  Umlenkung der Diagnose-Ausgabe - danach muss ein Dateiname kommen
- `& (T_AMP)`

  Ausführung des Kommandos im Hintergrund - die darauf folgenden Zeichen bis zum Zeilenende werden hier ignoriert

Die Erkennung der verschiedenen Symbole (*token*) beschreibt folgender Automat:



Die Syntax-Analyse der Kommandozeile wird wie folgt durchgeführt:



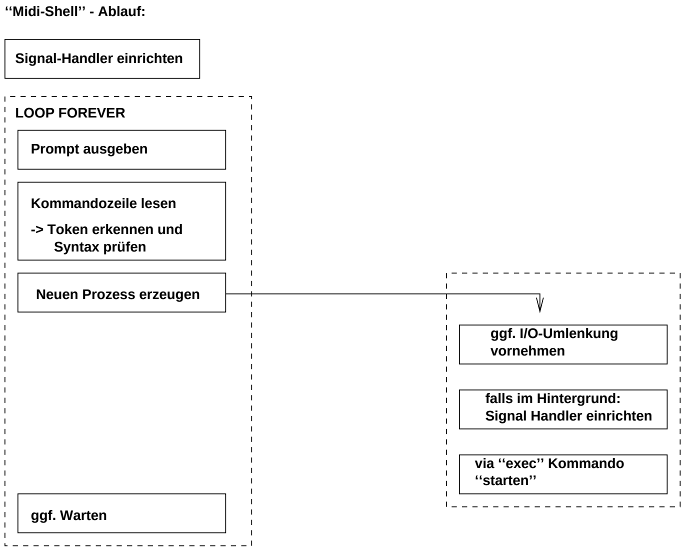
Anm.: Fehlerzustände sind hier nicht dargestellt - diese sind:

- Von den vier Arten der I/O-Umlenkung

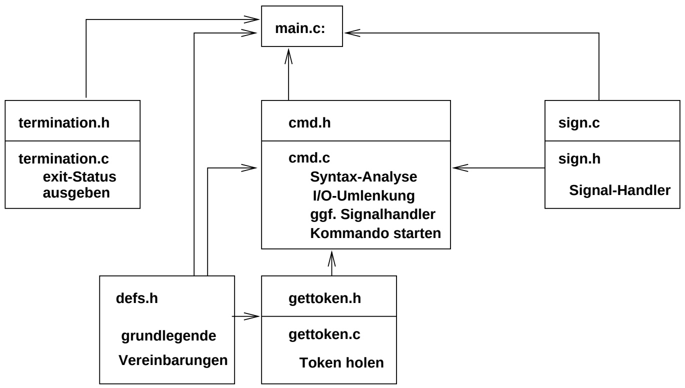
  `>, >>, 2>, <`

  ist jeweils maximal eine zulässig!

Ablauf-Schema:



Modularisierung:



Die Programme im Einzelnen:

```
/* ----------------- defs.h -----------------------*/
/* Grundlegende Vereinbarungen: */

typedef enum {FALSE, TRUE} BOOLEAN;

/* Token-Symbole: */
typedef enum {T_WORD,  T_GT, T_GTGT, T_TWO_GT, T_AMP,
T_LT, T_NL, T_EOF, T_ERR} TOKEN;

#define BADFD -2
#define MAXARG 20
#define MAXWORD 200
#define MAXFNAME 14



/*------------------ gettoken.h -----------*/
#ifdef GET_H
#include <stdio.h>
#include "defs.h"
#else
/* lexikalische Analyse der Kommandozeile */
extern TOKEN gettoken(char * word);

extern void skip_line();

#endif
```

```
/* ---------------------- gettoken.c ----------- */

#define GET_H
#include "gettoken.h"

void skip_line() {
  int c;
  while( (c = getchar()) != '\n');
}

/* lexikalische Analyse der Kommandozeile */

TOKEN gettoken(char * word) {
   int c;
   char * w;
   enum {NEUTRAL, TWO_GT, GTGT, INWORD } state = NEUTRAL;

   w = word;
   while ( (c=getchar() ) != EOF ) {
      switch (state) {

         case NEUTRAL:
            switch (c) {
               case '&':
                  /* read rest from line: */
                  skip_line();
                  return (T_AMP);
               case '<':
                  return (T_LT);
               case '\n':
                  return (T_NL);
               case ' ':
               case '\t':
                  continue;
               case '>':
                  state = GTGT;
                  continue;
               case '2':
                   state = TWO_GT;
                   continue;
               default:
                  state = INWORD;
                  * w++ = c;
                  continue;
            }
         case GTGT:
            if (c == '>')
               return (T_GTGT);
            ungetc(c,stdin);
            return (T_GT);
         case TWO_GT:
            if (c == '>')
              return (T_TWO_GT);
            *w++ = '2';
```



```
            ungetc(c,stdin);
            state = INWORD;
            continue;
         case INWORD:
            switch (c) {
               case '&':
               case '<':
               case '>':
               case '\n':
               case ' ':
               case '\t':
                  ungetc(c,stdin);
                  * w = '\0';
                  return (T_WORD);
               default:
                  * w++ =c;
                  continue;
            }
      }
   }
   return (T_EOF);
}
```

```
/* -------------------- sign.h ------------------------ */
/* Signal Handler                                       */

#ifdef SIGN_H
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>

typedef void (*Sigfunc)(int);

#else

#define SIGN_H
/* avoid multiple includes */
#include <signal.h>

typedef void (*Sigfunc)(int);

Sigfunc ignoresig(int);
/* ignore interrupt and avoid zombies
         * just for midishell (parent)
         */
Sigfunc ignoresig_bg(int);
/* ignore interrupt  -
         * just for execution of background commands
         */
Sigfunc entrysig(int);
/* restore reaction on interrupt */
#endif



/* ------------------------ sign.c ------------------------ */

#define SIGN_H

# include <stdio.h>
# include "sign.h"

void shell_handler(int sig){
  if( (sig == SIGCHLD) || (sig == SIGCLD)) {
     int status; long gone;
     gone = waitpid(0, &status, WNOHANG);
     printf("Terminated: %ld - ", gone);
     if (status & 0177)
printf("Signal %d\n", status & 0177);
     else
printf("Exit-Status: %d\n", (status >> 8)&0xff);
  }
  return;
}

struct sigaction newact, oldact;
```



```
Sigfunc ignoresig(int sig) {
   static int first = 1;
   newact.sa_handler = shell_handler;
   if (first) {
      first = 0;
      if (sigemptyset(&newact.sa_mask) < 0)
          return SIG_ERR;
      newact.sa_flags = 0;
      newact.sa_flags |= SA_RESTART;
      if (sigaction(sig, &newact, &oldact) < 0)
              return SIG_ERR;
      else
              return oldact.sa_handler;
   } else {
      if (sigaction(sig, &newact, NULL) < 0)
              return SIG_ERR;
      else
              return NULL;
   }
}


struct sigaction newact_bg, oldact_bg;

Sigfunc ignoresig_bg(int sig) {
   newact_bg.sa_handler = SIG_IGN;
   if (sigemptyset(&newact_bg.sa_mask) < 0)
          return SIG_ERR;
   newact_bg.sa_flags = 0;
   newact_bg.sa_flags |= SA_RESTART;
   if (sigaction(sig, &newact_bg, &oldact_bg) < 0)
              return SIG_ERR;
   else
              return oldact_bg.sa_handler;
}

Sigfunc entrysig(int sig)   {
   if (sigaction(sig, &oldact, NULL) < 0 )
      return SIG_ERR;
   else
      return NULL;
}
```

```
/* ------------------- cmd.h ----------------*/

#ifdef CMD_H
#include <stdio.h>
#include <strings.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include "defs.h"
#include "gettoken.h"
#include "sign.h"
#else

extern void redirect(int srcfd, char *srcfile,
                     int dstfd, char *dstfile,
                     int errfd, char * errfile,
                     BOOLEAN append, BOOLEAN bckgrnd);
/* I/O - redirection */

extern int invoke(int argc, char *argv[],
                  int srcfd, char *srcfile,
                  int dstfd, char *dstfile,
                  int errfd, char *errfile,
                  BOOLEAN append, BOOLEAN bckgrnd);
/* invoke() - execute simple command */

extern TOKEN command(int *waitpid);
/* collect a simple command from stdin
 * by calling gettoken() - do redirection
 * if necessary by redirect() and execute
 * command by invoke()
 */

#endif



/* --------------- cmd.c ------------------*/

#include <string.h>
#define CMD_H
#include "cmd.h"

/* Redirection of I/O: */
/* after redirect the caller has file descriptor 0
 * for input, descriptor 1 for output, and 2 for stderr !
 */
static void redirect(int srcfd, char * srcfile,
                     int dstfd, char * dstfile,
                     int errfd, char * errfile,
                     BOOLEAN append,
                     BOOLEAN bckgrnd) {
  int flags, fd;
```



```
  /* we expect for srcfd:
   *    0: nothing to do
   *   -2: redirect 0 (stdin) to file
   * (-1 indicates error)
   *
   * we expect for dstfd:
   *    1: nothing to do
   *   -2: redirect 1 (stdout) to file
   *   (with respect to parameter 'append'
   *
   * we expect for errfd:
   *    2: nothing to do
   *   -2: redirect 2 (stderr) to file
   */

  if (srcfd == 0 && bckgrnd) {
    strcpy(srcfile, "/dev/null");
    /* /dev/null ->
     * there is nothing to read, only EOF
     * a background command couldn't get any
     * input from stdin;
     */
    srcfd = BADFD;
    /* so redirect 0 to srcfile
     * set to /dev/null above
     */
  }

  if (srcfd != 0) {
  /* 0 should point to file for input
   */
    if (close(0) == -1)
      perror("close");
    else if (open(srcfile, O_RDONLY, 0) == -1) {
      fprintf(stderr, "can't open %s\n", srcfile);
      exit(1);
    }
  }

  /*now file is referenced by file descriptor 0
   */

  /* now the same for std_output */
  if (dstfd != 1) {
    /* output to  file (>,>> (dstfd==-2)
     */
    if (close(1) == -1)
      perror("close");
    else {
      flags = O_WRONLY | O_CREAT;
      if (!append)  /* > file */
        flags |= O_TRUNC;
      else
```

```
        flags |= O_APPEND;
      if (open(dstfile, flags, 0666) == -1) {
        /* open returns the smallest
         * free file descriptor
         */
        fprintf(stderr, "can't create %s\n", dstfile);
        exit(1);
      }
    }
  }
  /* now the same for std_error*/
  if (errfd != 2) {
    /* output to  file
     */
    if (close(2) == -1)
      perror("close");
    else {
      flags = O_WRONLY | O_CREAT | O_TRUNC;

      if (open(errfile, flags, 0664) == -1) {
        /* open returns the smallest
         * free file descriptor
         */
        fprintf(stderr, "can't create %s\n", errfile);
        exit(1);
      }
    }
  }


  for (fd =3; fd < 20; fd++)
    (void) close(fd);
  /* the caller now only needs 0,1,2 !!!*/
}

/* invoke() - execute simple command
 * in a new process
 */

static int invoke(int argc, char * argv[],
                  int srcfd, char * srcfile,
                  int dstfd, char * dstfile,
                  int errfd, char * errfile,
                  BOOLEAN append, BOOLEAN bckgrnd) {
  /* uses redirect() */

  int pid;

  /* empty commandline??? */
  if (argc == 0 )
    return (0);
  switch (pid = fork() ) {
    case -1:
      fprintf(stderr, "Can't create new process\n");
```



```
      return (0);
    case 0:
      /* CHILD */

      redirect(srcfd, srcfile, dstfd, dstfile, errfd, errfile,
                  append, bckgrnd);

      /* restore reaction on interrupt and quit???
       * not necessary, but could be as follows:
       * if (!bckgrnd)
       *    entrysig(SIGINT);
       */

      /* install signal handler for background: */
      if (bckgrnd) {
         if ( ignoresig_bg(SIGINT) == SIG_ERR) {
            perror("ignorsig_bg - SIGINT");
            exit(1);
         }
      }


      execvp(argv[0], argv);
      /* this shouldn't be reached */
      fprintf(stderr, "can't execute %s\n", argv[0]);
      exit(1);
    default:
      /* PARENT */
      if (srcfd > 0 && close(srcfd) == -1)
        perror("close src");
      if (dstfd > 1 && close(dstfd) == -1)
        perror("close dst");
      if (errfd > 2 && close(errfd) == -1)
        perror("close error");

      if (bckgrnd)
        printf("%d\n", pid);
      return (pid);
    }
}

TOKEN command(int * waitpid) {
  /* int * waitpid: perhaps we have to wait
   * for the command
   *
   * return value: T_NL or T_AMP on success, T_ERR on error
   *
   * uses: gettoken(), invoke()
   */

  TOKEN token, term;
  int argc, srcfd, dstfd, errfd, pid;
  char * argv[MAXARG+1];
  char  srcfile[MAXFNAME+1];
```

```
char  dstfile[MAXFNAME+1];
char  errfile[MAXFNAME+1];
char word[MAXWORD], *malloc();
BOOLEAN append;

argc = 0; srcfd = 0; dstfd = 1; errfd = 2;
/* defaults */

while (1) {
  switch (token = gettoken(word)) {
  case T_WORD:
    if (argc == MAXARG) {
      fprintf(stderr, "Too many args\n");
      break;
    }
    if ((argv[argc]=malloc(strlen(word)+1))==NULL) {
      fprintf(stderr,"Out of arg memory\n");
      break;
    }
    strcpy(argv[argc],word);
    argc++;
    continue;
  case T_LT:
    if (srcfd != 0) {
      fprintf(stderr, "syntax error: EXTRA <\n");
      skip_line();
      return T_ERR;
    }
    if (gettoken(srcfile) != T_WORD) {
      fprintf(stderr, "syntax error: Illegal <\n");
      skip_line();
      return T_ERR;
    }
    /* we have to redirect 0 to a file */
    srcfd = BADFD;
    continue;
  case T_GT:
  case T_GTGT:
    if (dstfd != 1) {
      fprintf(stderr, "syntax error: EXTRA > or >>\n");
      skip_line();
      return T_ERR;
    }
    if (gettoken(dstfile) != T_WORD) {
      fprintf(stderr, "syntax error: Illegal > or >>\n");
      skip_line();
      return T_ERR;
    }
    dstfd = BADFD;
    append = (token == T_GTGT);
    continue;
  case T_TWO_GT:
    if (errfd != 2) {
      fprintf(stderr, "syntax error: EXTRA 2>\n");
```



```
      skip_line();
      return T_ERR;
    }
    if (gettoken(errfile) != T_WORD) {
      fprintf(stderr, "syntax error: Illegal 2>\n");
      skip_line();
      return T_ERR;
    }
    errfd = BADFD;
    continue;

  case T_AMP:
  case T_NL:
         term = token;

    /* one simple command is read */
    argv[argc] = NULL;

    /* Eingabe von '> file' allein: loeschen / anlegen
     *  einer Datei:
     */
    if( (argc == 0 ) && (dstfd == BADFD) && (!append)) {
       dstfd = open(dstfile, O_WRONLY | O_CREAT | O_TRUNC, 0664);
       close(dstfd);
       return T_NL;
    }

    pid = invoke(argc,argv,srcfd, srcfile,dstfd,
                 dstfile, errfd, errfile,
                 append, term == T_AMP);
    * waitpid = pid;

    while (--argc >= 0)
      free(argv[argc]);
    return (term);
  case T_EOF:
    printf("\n\n");
    exit(0);
  default: exit(1); /* not reached! */
  }
}; /*end of while(1)*/
/*if reached, then error: */
return T_ERR;
}
```

```
/* ------------ termination.h ------------ */

#ifdef TERM_H

#include <stdio.h>
#define lowbyte(w) ((w) & 0377)
#define highbyte(w) lowbyte((w) >> 8)
#define MAXSIG 19

#else
void statusprt(int status);
#endif



/* ------------ termination.c -----------*/
#define TERM_H
#include "termination.h"

static char * sigmsg[] = {
      "",
      "Hangup", "Interrupt", "Quit",
      "Illegal instruction", "Trace trap",
      "IOT instruction", "EMT instruction",
      "Floating point exception",
      "Kill", "Bus error",
      "Segmentation violation",
      "Bad arg to system call",
      "Write on pipe", "Alarm clock",
      "Terminate signal",
      "User signal 1", "User signal 2",
      "Death of child", "Power fail"
   };

void statusprt(int status) {
   int code;

   if (lowbyte(status) == 0) {
      if ((code = highbyte(status)) != 0)
         printf("Exit code %d\n", code);
   }
   else {
      if ((code=status & 0177) <= MAXSIG)
         printf("%s", sigmsg[code]);
   else
      printf("Signal# %d", code);
   if ((status & 0200) == 0200)
      printf(" - core dumped");
   printf("\n");
   }
}
```



```
/*----- main.c -------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#include "defs.h"
#include "sign.h"
#include "cmd.h"
#include "termination.h"

int main() {
   char * prompt;
   int pid, status;
   TOKEN term;

   if( ( ignoresig(SIGINT) == SIG_ERR) ||
       ( ignoresig(SIGCHLD) == SIG_ERR)) {
       perror("ignoresig");
       exit(1);
   }

   prompt = "midish> ";

   while (1) {
      printf("%s", prompt);
      term = command(&pid);
      if (term == T_ERR) {
         continue;
      }
      if ( (term != T_AMP) && (pid != 0) ) {
 /* wait for foreground process -
  * if fg process terminates the
  * signal handler will handle the exit status
  * (will do his wait!)
  * the following waitpid will return with -1!
  */
         waitpid(pid,&status,0);
      }
   }
   printf("\n\n");
   exit(0);
}
```

Das Makefile:

```
# eine kleine Shell - die midishell

midish:    main.o cmd.o sign.o termination.o gettoken.o
    gcc -Wall -o midish main.o cmd.o sign.o termination.o gettoken.o
    # ausfuehrbares Programm: midish
    gcc -Wall -o sleepwell sleepwell.c
    # sleepwell: ein Dauerlaeufer
    gcc -Wall -o read_something read_something.c
    # read_something: lies von stdin

main.o:    main.c defs.h sign.h cmd.h termination.h
    gcc -Wall -c main.c

cmd.o:    cmd.c cmd.h gettoken.h defs.h sign.h
    gcc -Wall -c cmd.c

gettoken.o:    gettoken.c gettoken.h defs.h
    gcc -Wall -c gettoken.c

sign.o:    sign.c sign.h
    gcc -Wall -c sign.c

termination.o:    termination.c termination.h
    gcc -Wall -c termination.c

.PHONY: clean realclean
clean:
    rm -f *.o core

realclean:
    rm -f *.o core midish sleepwell read_something
```

Ausführung:

```
hypatia$ make
gcc -Wall -c main.c
gcc -Wall -c cmd.c
gcc -Wall -c sign.c
gcc -Wall -c termination.c
gcc -Wall -c gettoken.c
gcc -Wall -o midish main.o cmd.o sign.o termination.o gettoken.o
# ausfuehrbares Programm: midish
gcc -Wall -o sleepwell sleepwell.c
# sleepwell: ein Dauerlaeufer
gcc -Wall -o read_something read_something.c
# read_something: lies von stdin

hypatia$ midish
midish>ps
  PID TTY          TIME CMD
  212 tty1     00:00:00 bash
```



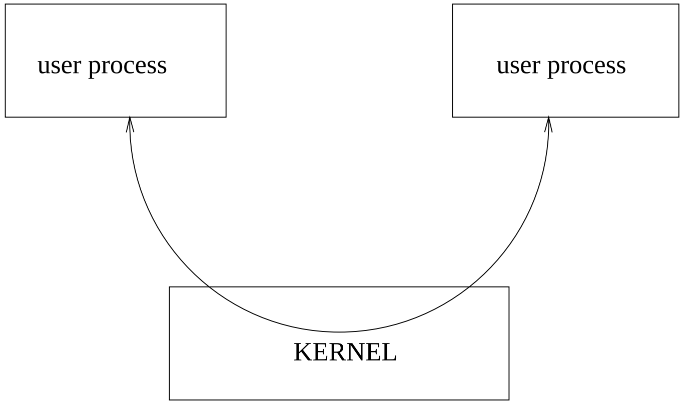
```
  846 pts/0    00:00:00 bash
  895 pts/0    00:00:00 midish
  896 pts/0    00:00:00 ps
midish>sleepwell
# Eingabe von ctrl-c
Interrupt
midish>sleepwell &
898
midish>ps
  PID TTY          TIME CMD
  212 tty1     00:00:00 bash
  846 pts/0    00:00:00 bash
  895 pts/0    00:00:00 midish
  898 pts/0    00:00:00 sleepwell
  899 pts/0    00:00:00 ps
midish> # ctrl-c
midish>ps
  PID TTY          TIME CMD
  212 tty1     00:00:00 bash
  846 pts/0    00:00:00 bash
  895 pts/0    00:00:00 midish
  898 pts/0    00:00:00 sleepwell
  900 pts/0    00:00:00 ps
midish>read_something
Give number:5
got: 5
midish>read_something &
902
Give number:midish>5
can't execute 5
Exit code 1
midish>cat < text
dies ist
ein Text
midish>cat < text > new > very_new
syntax error: EXTRA > or >>
midish>cat -?
cat: invalid option -- ?
Try `cat --help' for more information.
Exit code 1
midish>cat -? 2> err
Exit code 1
midish>cat err
cat: invalid option -- ?
Try `cat --help' for more information.
midish> # ctrl-d

hypatia$
```

Kapitel 4

Inter-Prozess-Kommunikation (IPC)

4.1 Einführung

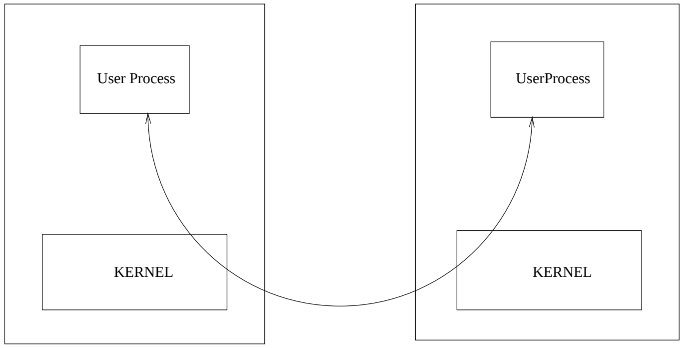
Jeder *UNIX*-Prozess besitzt seinen eigenen Context. Innerhalb eines Prozesses können die verschiedenen Moduln über Parameter und Rückgabewerte bei Funktionsaufrufen oder über globale Variablen Daten austauschen. Wollen jedoch zwei eigenständige Prozesse Daten miteinander austauschen, so kann dies nur über den Kernel via System Calls erfolgen. Denn der Kernel verhindert unkontrollierte Übergriffe eines Prozesses in den Adreßraum eines anderen Prozesses.



Bei diesem Konzept müssen beide Prozesse explizit der Kommunikation zustimmen. Der *UNIX*-Kernel bietet mit seinen *Interprocess Communication Facilities* nur die Möglichkeit zur Kommunikation an.

Netzwerk-Kommunikation

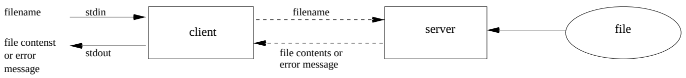
Das UNIX-IPC-Konzept läßt sich orthogonal erweitern von der lokalen Kommunikation zwischen Prozessen innerhalb eines Systems auf Netzwerk-Kommunikation zwischen Prozessen, die auf verschiedenen System laufen. Vor allem die Entwicklungsarbeiten der University of California at Berkeley brachte hier einige neue Ansätze zur Interprocess Communication in *UNIX* ein.



Für die Prozesse kann es völlig transparent sein, wo der jeweilige Partnerprozeß abläuft.

4.2 IPC - Client-Server Beispiel

Der *Client* liest einen Dateinamen von *stdin* ein und schreibt ihn in den *IPC*-Kanal. Anschliessend wartet er auf die Reaktion des Servers.
Der *Server* liest einen Dateinamen von dem *IPC*-Kanal und versucht die Datei zu öffnen. Gelingt es dem Server, die Datei zu öffnen, kopiert er ihren Inhalt in den *IPC*-Kanal. Läßt sich die Datei nicht öffnen, schickt der Server eine Fehlermeldung über den *IPC*-Kanal.
Der Client wartet auf Daten am *IPC*-Kanal, er liest sie von dort und schreibt sie nach *stdout*. Konnte der Server die Datei öffnen, zeigt der Client so den Dateiinhalt an, sonst kopiert der Client die Fehlermeldung durch.



Die beiden gestrichelten Pfeile zwischen dem Server und dem Client entsprechen dem jeweiligen *"Interprocess Communication"* Kanal.

4.3 System Calls "dup()", "dup2()"

Im Zusammenhang mit **u**nnamed pipes ist der Systemaufruf **dup** nützlich:

```
#include <unistd.h>
int dup(int oldfd)   /*duplicate file descriptor*/
int dup2(int oldfd, int newfd)   /*newfd is closed before (if open)*/
/* returns new file descriptor or -1 on error */
```

dup verdoppelt einen bestehenden Filedeskriptor und liefert als Resultat einen neuen Filedeskriptor (mit der **kleinsten** verfügbaren Nummer), der mit der gleichen Datei oder der gleichen Pipe verbunden ist. Beide File Deskriptoren haben denselben Positionszeiger. Damit kann z.B. ein Filedeskriptor mit der Nummer **0** erhalten werden, falls dieser vorher geschlossen wurde. Falls so mit **exec** ein Programm ausgeführt wird, das von Filedeskriptor **0** liest, kann es so dazu gebracht werden, aus einer Pipe zu lesen. Ähnlich kann so der Filedeskriptor **1** "manipuliert" werden.

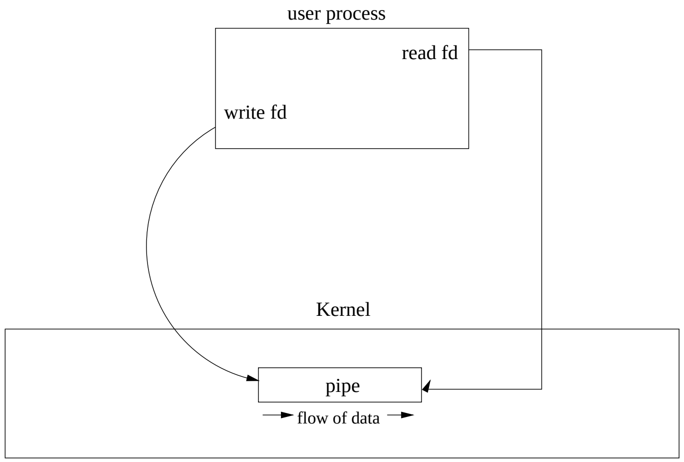
4.4 Unnamed Pipes

• System Call **p**ipe()

```
#include <unistd.h>
int pipe( int pipefd[2] )  /* create a pipe */

/* pipefd[2]: file descriptors */
/* returns 0 on success or -1 on error */
```

• Beschreibung

Pipes sind der älteste *IPC*-Mechanismus. Seit Mitte der 70er Jahre existieren sie auf allen Versionen und Arten von *UNIX*.
Eine Pipe besteht aus einem *unidirektionalen* Datenkanal. Zwei File Deskriptoren repräsentieren die Pipe im User Prozess. Der System Call *pipe()* kreiert die Pipe, er liefert die beiden Enden als File Deskriptoren über sein Vektorargument an den Prozess. Dabei ist **p**ipefd[1] das "Ende" zum Schreiben, **p**ipefd[0] das "Ende" zum Lesen.

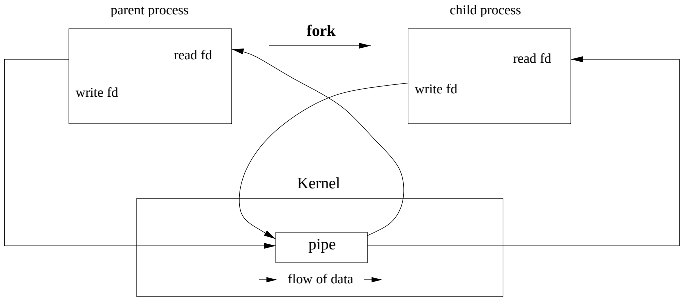


• Deadlock

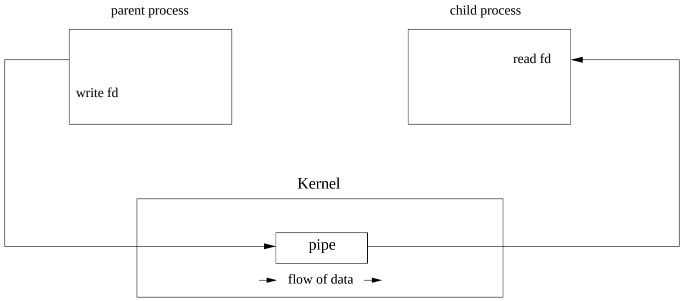
In dieser Konstellation läßt sich die Pipe nur als "Zwischenspeicher" für Daten außerhalb des User Adreßraums benutzen.
VORSICHT! Der Kernel synchronisiert Prozesse, die in Pipes schreiben oder aus Pipes lesen – *Producer / Consumer* Modell. Sollte hier in dem Ein-Prozess-Beispiel ein **r**ead() oder **w**rite System Call blockieren, entsteht ein *Deadlock*, denn der blockierte Prozess kann den befreienden, komplementären *write()* oder *read()* System Call nicht absetzen.

• Kommunikation zwischen verwandten Prozessen

Durch einen **f**ork() System Call entstehen zwei eigenständige, aber verwandte Prozesse, die insbesondere die gleichen I/O-Verbindungen besitzen.



Schließt nun ein Prozess sein Lese-Ende und der andere Prozess sein Schreib-Ende, so entsteht ein unidirektionaler Kommunikationspfad zwischen den beiden Prozessen.



Wiederholen die Prozesse die **f**ork, **p**ipe() und **c**lose System Calls, entstehen längere Pipelines. Dieses Datenverarbeitungs-Prinzip ist untrennbar mit *UNIX* verbunden.

• Beispiel: Pipe zwischen zwei Prozessen

Das Programm kreiert eine *pipe* und einen zweiten Prozess, dem diese beiden *pipe*-Deskriptoren verebt werden. Der Erzeugerprozeß schreibt Text in die Pipe und wartet auf das Ableben des Kindprozesses. Der Kindprozeß liest Text aus der Pipe, schreibt ihn nach *stdout* und beendet seine Ausführung. Folgende Schritte sind der Reihe nach auszuführen:

- ☞ Parent führt *pipe()* aus
- ☞ Parent führt *fork()* aus
- ☞ Child schließt sein Schreib-Ende der Pipe
  und wartet an seinem Lese-Ende der Pipe.
- ☞ Parent schließt sein Lese-Ende der Pipe,
  schreibt Text in sein Schreib-Ende der Pipe,
  und führt *wait()* für sein Kind aus.
- ☞ Child liest von seinem Lese-Ende, gibt gelesenen Text aus
  und terminiert.
- ☞ Parent hat auf Child gewartet und kann jetzt auch terminieren.

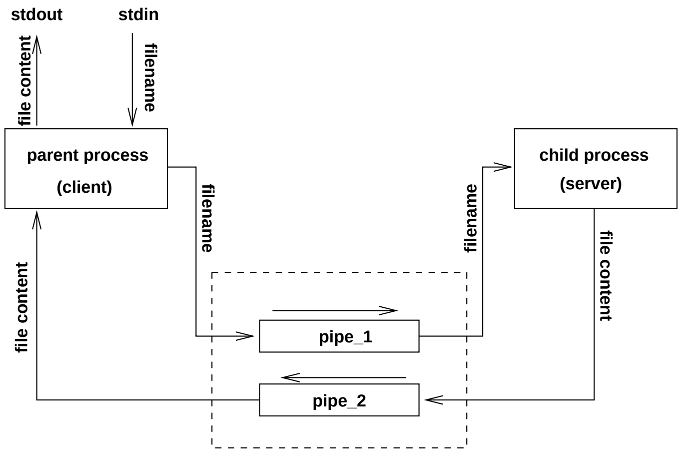
Realisierung

```
/*---------------- ./pipe1/pipe.c -------------------*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define RD_FD   0
#define WR_FD   1

int main() {
int childpid, gone, pipefd[2];

if ( pipe( pipefd ) < 0 ) { perror("pipe" ); exit(1); }

switch( childpid = fork() ){
   case -1:
       perror("fork"); exit(1);
   case  0: /* child: */ { /* <-- */
       char buf[ 128 ];   int nread;

       /* close WRITE end of pipe */
       close( pipefd[ WR_FD ] );

       /* read from pipe and copy to stdout */
       nread = read( pipefd[ RD_FD ], buf, sizeof(buf));

       printf( "Child: read `%.*s', going to exit\n",
                                          nread, buf );
       break;
       }

   default: /* parent: */
       /* close READ end of pipe */
       close( pipefd[ RD_FD ] );

       /* write something into pipe */
       write( pipefd[ WR_FD ], "hello world", 11 );

       printf("parent: wrote `hello world' into pipe\n");

       do /* wait for child */ {
          if ( (gone = wait( (int *) 0 )) < 0 )  {
             /*no interest for exit_status*/
             perror("wait"); exit(2);
          }
       } while ( gone != childpid );
   }
exit(0);
}
```

4.5 Client-Server mit "Unnamed Pipes"

• Bidirektional

Durch eine Pipe fließen Daten nur in genau eine Richtung. Zur Realisierung unseres Client-Server Beispiels benötigen wir aber einen *bidirektionalen* Kommunikationskanal. Wir müssen dazu zwei Pipes kreieren und eine Pipe für jede Richtung konfigurieren.



- Vorgehen
  - ☞ Pipe1 und Pipe2 kreieren
  - ☞ *fork()* ausführen
  - ☞ Linker Prozess (Erzeuger) schließt
    - ∗ Lese-Ende von Pipe1 und
    - ∗ Schreib-Ende von Pipe2
  - ☞ Rechter Prozess (Kind) schließt
    - ∗ Schreib-Ende von Pipe1 und
    - ∗ Lese-Ende von Pipe2

Realisation mit zwei Pipes

- Hauptprogramm:

```
/*
 * main.c: two (unnamed) pipes to realize the IPC-channel
 *   two pipes between two processes
 * main - uses two functions for client / server functionality
 */

#include  <stdio.h>
#include  <unistd.h>
#include  <sys/wait.h>

# include "client.h"
# include "server.h"

#define RD_FD   0
#define WR_FD   1

int main() {
int childpid, gone, pipe_1[2], pipe_2[2];

if ( pipe( pipe_1 ) < 0  ||  pipe( pipe_2 ) < 0 ) {
   perror("pipe(): can't creat pipes");
   exit(1);
}

if ( (childpid = fork()) > 0 ) {
   /* -------- client <- parent ------------------------*/
   printf( "Client/Parent: my pid is %ld\n", getpid() );

   close( pipe_1[ RD_FD ] );/* close READ  end of pipe1 */
   close( pipe_2[ WR_FD ] );/* close WRITE end of pipe2 */

   /*--------------------------------------------------*/
   client( pipe_2[ RD_FD ], pipe_1[ WR_FD ] );
   /*--------------------------------------------------*/

   /* wait for child */
   do {
            if ( (gone = wait( (int *) 0 )) < 0 ) {
               perror("wait");
               exit(2);
            }
   } while ( gone != childpid );

   printf( "Cli/Par: server/child %d terminated\n", gone );
   printf( "Cli/Par: going to exit\n" );

} else if ( childpid == 0 ) {
/* ---------------- server/child -----------------------*/
   printf( "S/Ch: after fork, my pid is %ld\n", getpid() );
```

```
   close( pipe_1[ WR_FD ] );/* close WRITE end of pipe1 */
   close( pipe_2[ RD_FD ] );/* close READ  end of pipe2 */

   /*-----------------------------------------------------*/
   server( pipe_1[ RD_FD ], pipe_2[ WR_FD ] );
   /*-----------------------------------------------------*/

   printf( "Server/Child: going to exit\n" );

} else {
   perror("fork" );
   exit(3);
  }

   exit( 0 );
}
```

- Realisation des Client-Teils

```
/*
 *   client.c: realize the client part
 *   read line from stdin,
 *   write this text to IPC-channel,
 *   copy text from IPC-channel to stdout
 */

# define CLIENT_H
# include "client.h"

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

# define BUFSIZE 256

void client( int readfd, int writefd ) {
   char buf[ BUFSIZ ];
   int  n;

   /*
    * read filename from stdin,
    * write it to IPC-channel
    */
    printf("give filename: ");

   if ( fgets(buf, BUFSIZ, stdin) == (char *) 0 ) {
      fprintf(stderr, "Client: filename read error" );
      exit(1);
   }

   n = strlen( buf );
   if ( buf[ n-1 ] == '\n' )
      n--;   /* zap NL */
```



```
   if ( write( writefd, buf, n ) != n ) {
       fprintf(stderr, "write(): Client: can't write to IPC-channel" );
       exit(2);
   }

   /*
    * read data from IPC-channel
    * write it to stdout
    */

   while ( (n = read( readfd, buf, BUFSIZ )) > 0 )
     if ( write( 1, buf, n ) != n )  /* fd 1 == stdout */ {
      fprintf(stderr, "write(): Client: can't write to stdout" );
      exit(3);
     }

   if ( n < 0 ) {
     fprintf(stderr, "read(): Client: can't read from IPC-channel" );
     exit(4);
   }
}
```

- Realisation des Server-Teils

```
/*
 * server.c: realize the server part
 *   read filename from IPC-channel,
 *   open this file,
 *   copy data from file to IPC-channel.
 */

# define SERVER_H
# include "server.h"

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

#define BUFSIZE 512

void server( int readfd, int writefd ) {
   char buf[ BUFSIZ ];
   int n, fd;

   /* read filename from IPC-channel */

   if ( (n = read( readfd, buf, BUFSIZ )) < 0 ) {
      fprintf(stderr, "Server: filename read error" );
      exit(1);
   }
   buf[ n ] = '\0';
```

```
    /* try to open file */

    if ( (fd = open( buf, O_RDONLY )) < 0 ) {
       /* Format and send error mesg to client */
       char errmesg[ BUFSIZ ];

       (void) sprintf( errmesg, "Server: can't open infile (%.*s)\n",
          BUFSIZ/2, buf );
       n = strlen( errmesg );
       if ( write( writefd, errmesg, n ) != n ) {
         fprintf(stderr, "S: can't write errmesg to IPC-channel" );
         exit(2);
       }
       return;
    }

 /*
  * read data from file and
  * write to IPC-channel
  */

   while ( (n = read( fd, buf, BUFSIZ )) > 0 )
     if ( write( writefd, buf, n ) != n ) {
       fprintf(stderr, "Server: can't write to IPC-channel" );
       exit(3);
     }
   if ( n < 0 ) {
     fprintf(stderr, "Server: can't read file" );
     exit(3);
   }
 }
```


4.6 Standard I/O Bibliotheksfunktion

popen() und **p**close()

```
#include <stdio.h>

FILE *popen( char * cmd, char * mode )
/* create a pipe to a cmd:
     cmd:  cmd to be executed
     mode: read from or write to pipe/cmd
   returns file pointer on success or NULL on error
*/

int pclose( FILE * fp )
/* close pipe with cmd */
/* returns exit status of cmd or -1 on error */
```

• Beschreibung

Die Funktion *popen()* aus der Standard I/O Bibliothek kreiert eine **unidirektionale (!)** Pipe und einen neuen Prozess, der von der Pipe liest oder in die Pipe schreibt. In dem neuen Prozess startet eine Shell und führt die mitgegebene Kommandozeile **cmd** (unter Berücksichtigung von **PATH**) aus. Die Pipe wird abhängig vom Argument **mode** (entweder ”r” oder ”w”!) so konfiguriert, dass sie *stdout* oder *stdin* des erzeugten Kommandos mit dem aufrufenden Prozess verbindet. Der aufrufende Prozess erhält sein Pipe-Ende als *”File Pointer”* von *popen*.
pclose schließt eine mit *popen* geöffnete I/O-Verbindung ab (NICHT: fclose()). Die Funktion blockiert bis das Kommando terminiert und liefert den Exit-Status des Kommandos zurück.

• Beispiel:

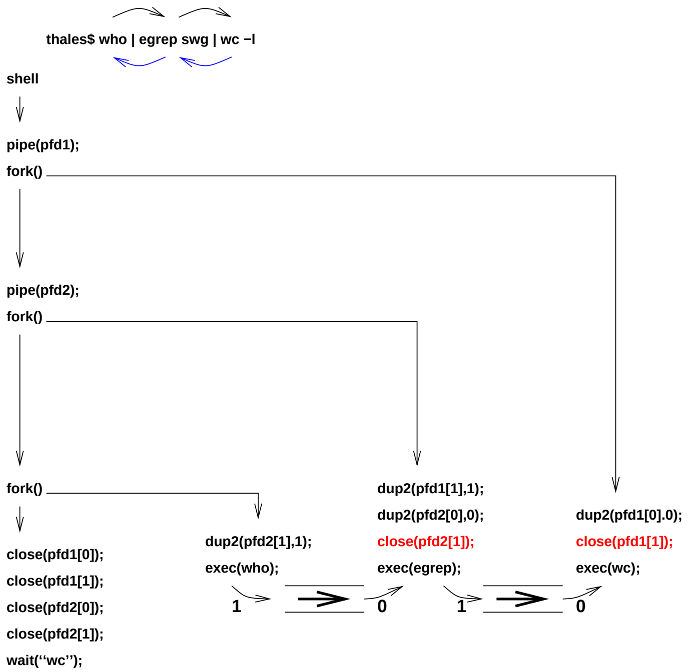
```
/* -------------- popen.c: popen() --------------*/

#include <stdio.h>

#define BUFFER_SIZE 1024

int main() {

   char buf[ BUFFER_SIZE ];
   FILE * fp;

   if ( (fp = popen( "/bin/pwd", "r" )) == (FILE *) 0 ) {
      perror("popen()");
      exit(1);
   }

   if ( fgets( buf, BUFFER_SIZE, fp ) == (char *) 0 ) {
      perror("fgetsr");
      exit(2);
   }

   printf( "The current working directory is:\n" );
   printf( "\t%s", buf );   /* pwd inserts newline */

   pclose( fp );
   exit(0);
}
```

4.7 Pipes in der Shell

In der Midi-Shellimplementierung fehlen noch die Pipes. Das folgende Beispiel zeigt den prinzipiellen Ablauf in der Shell für die Kommandozeile

`egrep muster datei | od -bc:`
Sool die Shell "beliebig" viele aneinandergereihte Pipes ermöglichen, stellt sich die Frage, wie man die Kommandos aufbaut und miteinander verbindet. Da man immer das Kommando **vor** und das **nach** der Pipeline braucht, ist die rekursive Lösung naheliegend:

- man sammelt die Kommandos von vorne her rekursiv auf
- und startet sie von hinter her



4.8 Nicht behandelte IPC-Mechanismen

- **n**amed pipes (**F**IFO-Dateien)

  ähnlich den *unnamed pipes*, aber Eintrag in *directory*
  Erzeugung:
  int mknod(char * pathname, int mode, int dev);
  /etc/mknod *pathname* **p**
  für FIFO's entfällt das Argument *dev*; siehe *man mknod*

- **M**essage Queues

  siehe *<sys/ ipc.h>, <sys/ msg.h>*

- **S**emaphore

  Mechanismus zur Synchronisation, weniger zum Datenaustausch
  siehe *<sys/ ipc.h>, <sys/ sem.h>*

- **S**hared Memory

  siehe *<sys/ ipc.h>, <sys/ shm.h>*

Index