Fachhochschule
Ravensburg-Weingarten

Hochschule für Technik und Sozialwesen

Studiengang Angewandte Informatik

Systemprogrammierung unter UNIX System V / Linux

Systemsoftware Praktikum

Prof. Dr. S. Keller

Ausgabe: 19.03.2004

Anzahl der Seiten: 27

INHALTSVERZEICHNIS

1 EINFÜHRUNG IN DIE SYSTEMPROGRAMMIERUNG IN C

In diesem Dokument werden die wichtigsten Systemfunktionen, die für das Praktikum benötigt werden in einer kurzen Einführung vorgestellt. Weiterführende Informationen können unter der man-page oder /11/ nachgelesen werden.

Unter UNIX wird ein Anwendungsprogramm von der **SHELL** aus durch Eingabe des Namens der Programmdatei gestartet und als eigenständiger UNIX-Prozess ausgeführt. Der **shell-Prozess** erzeugt mit Hilfe der Systemfunktion **fork()** einen Sohnprozess, in dem das C-Programm ausgeführt wird. Jedes C-Programm beginnt dabei mit der Funktion main(). Soll dem C-Programm von der **SHELL** aus Parameter übergeben werden, muß der Programmierer die Funktion main() mit zwei Parametern definieren:

```
int main( int argc, char *argv[] );
```

In **argc** wird die Anzahl der Worte in der Kommandozeile (Einträge in der Liste argv) übergeben. Da jedem Programm mindestens ein Argument, der Programmname selbst, übergeben wird ist der Wert von argc immer größer oder gleich 1.

argv ist eine Liste mit Zeigern, die auf die einzelnen Worte der Kommandozeile verweisen. Über argv[0] (Zeiger auf den ersten string in der Liste) erhält man das erste Wort der Kommandozeile, also den Namen des Programms, über argv[1] erhält man das zweite Wort der Kommandozeile und damit den ersten Parameter des aufgerufenen Programms, über argv[n] den n-ten Parameter. Die Anzahl der Parameter ist beliebig. Das Ende der Argumentliste argv[argc] wird daher mit einer besonderen Kennung, dem Wert (char*) 0 (NULL) abgeschlossen.

Das folgende Beispiel zeigt ein C-Programm, welches von der **SHELL** aus mit dem Programmname "shellprog" und zwei Parametern aufgerufen wird

B e i s p i e l :

Kommandozeile: $ shellprog "erster Parameter" 3

C-Programm:

```
int main( int argc, char *argv[] )
{   int  anzpara;
     printf("Programmname : %s\n",argv[0]);
     printf("Anzahl der Parameter : %i\n", argc-1);
     for ( anzpara=argc-1; anzpara; anzpara--)
          printf("%i.ter Parameter : %s\n", anzpara, argv[anzpara]);
     return 0;
}
```

Ausgabe des Programms als Sohnprozeß der SHELL:

Programmname : shellprog

Anzahl der Parameter : 2

2. Parameter : 3

1. Parameter : erster Parameter

1.1 FEHLERABFRAGE VON SYSTEMAUFRUFE

Für Systemaufrufe wird in UNIX eine C-Funktions-Bibliothek zu Verfügung gestellt, so daß dem Programmierer die Systemaufrufe wie normale Funktionen erscheinen.

Die meisten Systemaufrufe liefern im Fehlerfall den Wert -1 zurück, ansonsten ein Wert größer oder gleich 0. Im Fehlerfall setzten die meisten (leider nicht alle) Systemfunktionen den Wert einer globalen Fehlervariablen **errnno** auf eine vom System festgelegten Fehlernummer.

In der Headerdatei **<errno.h>** ist der Name sowie die Werte aller möglichen Fehlernummern festgelegt und muss mit #include eingebunden werden, soll die Fehlervariable abgefragt werden.

Da die Fehlervariable bei korrektem Ablauf eines Systemaufrufes nicht zurückgesetzt wird, sollte nur im Fehlerfall die Variable abgefragt werden.

Mit der C-Funktion perror() kann die Fehlervariable in eine lesbare Fehlermeldung übersetzt und auf der Konsole ausgegeben werden.

```
Syntax:   #include <stdio.h>
          void perror( const char *EigeneMeldung)
```

Der Parameter EigeneMeldung gibt einen Text an, der vor dem Fehlertext für errno ausgegeben wird

2 PROZESSVERWALTUNG

2.1 PROZESS ERZEUGEN

In UNIX werden Prozesse durch den Systemaufruf `fork()` e rzeugt. Der Systemaufruf bewirkt, daß eine Kopie des aufrufenden Prozesses im Arbeitsspeicher angelegt wird. Der aufrufenden Prozeß wird als **PARENT** bezeichnet, die Kopie als **CHILD**.

Durch diese Technik erbt der erzeugte Sohnprozeß vom Vater alle Zustände, Umgebungsvariablen, Programmdaten und Programmcode. Um PARENT von CHILD unterscheiden zu können erhält der neu erzeugte Prozeß eine eindeutige Prozeßnummer (PID), so daß nach Ausführung des Systemaufrufes fork() die beiden Prozesse eindeutig unterscheidbar sind. Da nach dem Systemaufruf zwei identische Prozesse existieren erhält jeder der beiden Prozesse einen Returnwert. Im Vaterprozeß ist der Returnwert die PID des Sohnes. Im neu erzeugten Sohnprozeß ist der Returnwert immer 0. Benötigt der CHILD die PID seines PARENT, so kann er dies durch den Systemaufruf **`getppid()`** erfragen. Möchte der PARENT seine eigene PID wissen, so kann er diese durch den Systemaufruf **`getpid()`** erfragen.

Syntax:

```
int fork(void)
```

Normalerweise wird der Vaterprozeß während der Ausführung des Sohnprozesses auf dessen Ergebnis warten. Der Vater blockiert sich daher i.a. selbst bis der Sohnprozeß terminiert. Die Funktion wait()blockiert das aufrufende Programm solange, bis ein beliebiger Sohnprozeß terminiert. Die Funktion waitpid() wartet auf das terminieren eines bestimmten Sohnprozesses.

Syntax:

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

pid_t wait( int *status)
```

Als Returnwert beider Funktionen erhält der aufrufende Vaterprozess die Prozessnummer des Sohnes. Im Fehlerfall ist der Returnwert -1. Ein solcher Fall tritt z.B. auf, wenn kein Sohnprozess existiert, auf dessen Ende gewartet werden kann.

Die Variable *status* enthält in Bit 8-15 den Ausführungsstatus des Sohnes (siehe dazu Prozess beenden). Die anderen Bits der Variable *status* sind für die Übungen ohne Interesse.

Ist der Sohnprozess im Zombiestatus, so kehren beide Funktionen sofort zurück und wirken daher nicht blockiernd. Sind die Kindprozesse noch aktiv so blockieren die Funktionen, bis zum Ende der Kindprozesse.

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

pid_t waitpid( pid_t pidSohn, int *status, int optionen )
```

Mit der Funktion waitpid() kann der Vaterprozess auf einen bestimmten Sohnprozess warten. Über den Parameter optionen kann eine Blockierung des Vaterprozesses untersagt werden (Konstante WNOHANG). In diesem Fall liefert die Funktion als Returnwert 0.

Sind keine Optionen erwünscht so wird der Wert 0 angegeben.

2.2 PROZESS BEENDEN

Das Ende einer Programmausführung und der aktuelle Ausführungszustand (Status) wird dem Vaterprozeß durch den Systemaufruf `exit()` mitgeteilt

Syntax:

```
void exit ( int status )                Prototyp : stdlib.h
```

Die Anweisung " **return status** " in der Funktion main() hat die selbe Wirkung.

Zombie Prozesse

Terminiert ein Sohnprozess, muss das Betriebsystem den return-status zur Auswertung für den Vatreprozess weiterhin bereit halten, obwohl der Prozess nicht mehr existiert. Dazu muss das betriebssystem die PID des terminierten Sohnprozesses und den returnwert weiterhin aufbewahren. Der Sohnprozess wird damit in den Zustand „Zombie“ versetzt. Dieser Zustand wird erst dann wieder verlassen, wenn der Vaterprozess durch wait() oder waitpid() den Status des Sohnes abgefragt hat und damit den zombie erlöst, der jetzt endgültig aus dem System entfernt wird.

→ es stellt sich dabei folgende Frage

Was passiert mit einem aktiven Sohnprozess, dessen Vater terminiert, so dass dieser des Status später nicht mehr abfragen kann ? Nach Terminierung des Sohnes würde ja dann der Sohnprozess ewig in den Zombie-Zustand versetzt.

→ Antwort:

Verliert ein Sohnprozess seinen Vater, so erhält er als neuen Vater den INIT-Prozess (PPID=1) Beendet sich ein solcher Kindprozess ruft INIT automatisch wait() auf und erlöst damit den Zombie. Kinder von INIT werden somit niemals Zombies.

2.3 Ändern der Ausführungsumgebung eines laufenden Prozesses

Nach der Erzeugung eines Sohnprozesses soll der CHILD im Normalfall ein bestimmtes Programm ausführen.Das neue Programm sowie die Programmdaten müssen zur Ausführung durch einen exec-Systemaufruf von der Festplatte in den Arbeitsspeicher geladen werden.

Syntax:

```
int execl( char *file, char *arg0, .... )
int execlp( char *file, char *arg0, ... )
```

Der erste Parameter *file* beschreibt den Dateinamen des auszuführenden Programms. Alle weiteren Parameter enthalten die Argumente des Programms, die über die Funktion main() an das Programm übergeben werden.

Da die Anzahl der Parameter variabel ist, benötigt das Betriebssystem eine Kennung, die das Ende der Liste kennzeichnet. Der letzte Parameter in der Parameterliste muß daher den Wert (char*) 0 haben (Symbolischer Name NULL).

Die beiden Systemfunktionen unterscheiden sich nur hinsichtlich des Suchpfades der Programmdatei.

excel sucht nur im direkt angegebenen Verzeichnis (bei relativem Pfad ist dies das aktuelle Arbeitsverzeichnis, bei Absolutem Pfad das in der Funktion angebenden Verzeichnis).

execlp dehnt die Suche bei Angabe eines relativen Pfades auf alle Verzeichnisse aus, die in der Systemvariablen $PATH angegeben sind.

Kann ein Programm nicht ausgeführt werden, so wird der Fehlerwert -1 als Returnwert geliefert. Ein Returnwert 0 kann niemals auftreten, da bei erfolgreicher Ausführung ein neues Programm geladen und sofort durch Aufrufen der Funktion main() gestartet wird ; das ursprüngliche Programm existiert damit nicht mehr.

2.4 Beispiel

Das folgende C-Programm demonstriert die Verwendung der Systemfunktionen fork(), wait(), exit() und excel() zur Erzeugung von UNIX-Prozessen. *CreateProzess* erzeugt durch fork einen Sohnprozeß in dem das Programm *HalloWorld* ausgeführt werden soll.Danach terminiert der Sohnprozeß. Während der Ausführung des Sohnes wartet der Vater auf dessen Terminierung.

Programm CreateProzess

```
int main( int argc, char *argv[])
{    int statusSohn,exitStatus,PIDSohn;
      printf("Ich bin der Vaterprozess\n");
      statusSohn=fork();     /*  Sohnprozess  wird  als  Kopie  des  Vaters
erzeugt */
      /* Programm des Sohnprozesses */
      /****************************/
      if (statusSohn == 0)
      {   printf("Ich bin der Sohn\n");
          execl("halloworld",NULL);
          exit(0);
       }
           /* Programm des Vaterprozesses */
      /****************************/

       else
       {    PIDSohn=wait(&exitStatus);
            printf(" Sohnprozess gestartet PID = %i, Sohnprozess terminiert
PID = %i\n",statusSohn,PIDSohn);
        }
exit(0);
}
```

Programm HalloWorld

```
int main()
{    printf("Hallo World\n");
     exit(0);
}
```

2.5 Ein Prozess kann seine eigene PID und die seines Vaterprozesses erfragen.

2.5.1 Eigene PID erfragen

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

pidnr getpid(void)          pidnr ist die eigene PID .
```

2.5.2 Erfragen der PID des Vater-Prozesses

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

ppidnr getppid(void)     ppidnr ist die PID des Vaterprozesses
```

2.6 THREADS

Im Juni 1995 wurde vom IEEE der definitive Standard POSIX.1c anerkannt, auch kurz Pthreads genannt.

POSIX ist eine Familie von Standards, die vom IEEE (Institute of Electrical and Electronics Engineers) entwickelt werden. POSIX steht für "Portable Operating System Interface" und definiert die Schnittstelle zu einem Betriebssystem, nicht aber deren Implementation. Die Spezifikation dieser Schnittstelle erfolgt vorwiegend in der Sprache C, obschon auch andere Sprachanbindungen vorgesehen sind.

Obwohl POSIX ursprünglich für UNIX-Systeme entwickelt wurde, ist der Standard nicht auf solche beschränkt. So gibt es z.B. auch in Windows NT ein POSIX-Subsystem. Zudem wurden einige der Teilstandards (wie POSIX.1) auch von der ISO (International Standardization Organisation) als internationale Standards anerkannt

Es gibt über 15 Teilstandards in POSIX, die noch nicht alle abgeschlossen sind. Der für die systemnahe Programmierung relevante Standard ist POSIX.1: System Application Programming Interface (API) [4]. Er garantiert die Portierbarkeit von Programmen auf Sourcecode-Stufe.

Bezüglich der threads schreibt der Standard nicht vor, auf welche Art (User-Level oder Kernel-Level) die Thread-Bibliothek implementiert sein muss. Deshalb können sich die Möglichkeiten der Bibliothek von System zu System stark unterscheiden, vor allem hinsichtlich Scheduling von Threads.

2.6.3 POSIX -Threads API für C/C++ Bibliotheken

THREAD-EIGENSCHAFTEN:

- Ein thread unter linux kennt weder seinen erzeugenden thread noch die threads, die von ihm erzeugt wurden.
- Alle threads eines Prozesses teilen sich einen gemeinsamen Adressraum, der Adressraum des Prozesses und alle seinen Ressourcen. Dazu gehören:
    - Process instructions
    - Globale Daten
    - Offene Dateien (descriptors)
    - Signale und Sgnal-Handler
    - working directory
    - User und group id
- Jeder Thread hat eine eigene eindeutige Thread ID, einen eigenen Stack, eine private Priorität und Signalmaske
- Alle Funktionen liefern einen returnwert von 0, falls kein Fehler auftritt.

THREAD-SYNCHRONISATION

Posix unterstützt drei Mechanismen

- **mutexes**
    - Realisiert mutual exclusion bei Zugriffen auf gemeinsam genutzte globale Variablen.
- **joins**
    - realisiert passives Warten auf des Terminieren anderer threads
- **condition variables**

2.6.3 Threads erzeugen

Ein Thread wird erzeugt durch die Funktion **pthread_create().**

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *),
                   void *arg);
```

Parameter:

- thread – rückgabe der thread id. (unsigned long int definiert in bits/pthreadtypes.h)
- attr - NULL falls voreingestelle Attribute verwendet werden

  else define members of the struct pthread_attr_t defined in bits/pthreadtypes.h) Attributes include:

  - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
  - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
  - scheduling parameter
  - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
  - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
  - guard size
  - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
  - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
- void * (*start_routine) – Zeiger auf die Funbktion, die der thread abarbeiten soll.

  Function has a single argument: pointer to void.
- *arg – Zeiger auf die Parameter der Thread-Funktion. To pass multiple arguments, send a pointer to a structure.

2.6.3 Threads terminieren

Ein thread terminiert beim Erreichen der return-Anweisung, bei Aufruf der Funktion pthread_exit(), oder durch dir Funktion exit(), mit der allerdings der gesamte Prozess terminiert, zu dem der Thread gehört und auch alle weiteren threads des Prozesses.

Funktion:

void pthread_exit(void *retval);

Parameter: retval – Return-Wert des thread.

3 INTERPROZEßKOMMUNIKATION (IPC)

Die ursprünglichen mechanismen zur Interprozesskommunikation unter Unix sind Pipes und Signale.

Als Erweiterung dieser sehr alten IPC-Mechanismen sind unter UNIX System V modernere Mechanismen implementiert, das sind **Semaphore**, **Shared Memory** und **Nachrichtenwarteschlangen**.

Diese drei neuen Mechanismen werden im Betriebssystem identisch verwaltet und sind daher in Ihrer Verwendung sehr ähnlich. Zwischen Prozessen werden Kommunikationskanäle (IPC-Kanal) erzeugt. Jeder IPC-Kanal wird durch eine Datenstruktur im Betriebssystem beschrieben. Diese Datenstruktur ist in der Headerdatei `<sys/ipc.h>` deklariert.

```
struct   ipc_perm {
     unsigned short uid;      /* owner user ID */
     unsigned short gid;      /* owners group ID */
     unsigned short cuid;     /* creator user ID */
     unsigned short cgid;     /* creator grpup ID */
     unsigned short mode      /* Zugriffmodus */
     key_t key;               /* Eindeutiger Key des Kanals */
     };
```

Mit dieser Datenstruktur wird in **key eine eindeutige numerische Kennung für den Kanal** eingetragen.

Über einen **get-Systemaufruf** wird ein Kanal erzeugt bzw. geöffnet. Ein Flag gibt dabei sowohl den Zugriffsmodus auf den Kanal an, als auch ob der Kanal neu erzeugt oder ein bestehender Kanal nur geöffnet werden soll.

Ein key wird unter UNIX **systemglobal** verwendet. d.h. Ressourcen sind nicht auf einen Prozeß oder ein Projekt bezogen. Um bei der Auswahl eines key Konflikte zwischen den Übungsgruppen zu vermeiden müssen sie eine feste Konvention einhalten, um eindeutig unterscheidbare Keys zu erzeugen. Hierfür gibt es die Möglichkeit mit der Funktion `ftok()` eindeutige keys generieren zu lassen.

Syntax:

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

key_t   ftok(char *pathname, char kanalnummer);
```

pathname ist die Pfadbezeichnung einer Datei, die für alle Prozesse einer Anwendung zugänglich ist.

kanalnummer ist die Nummer des Kommunikationskanals in der Anwendung bzw. im im Projekt .

Aus pathname und proj erzeugt die Funktion einen eindeutigen key.

Der Datentyp key_t wird in der Headerdatei `<sys/types.h>` deklariert.

Eine andere Möglichkeit einen eindeutigen key zu benutzen besteht darin Ihren key aus Ihrer User-ID abzuleiten.

Sie definieren Ihren key dann über eine define-Anweisung in einer Headerdatei. Für die keys kann folgende Konvention vereinbart werden: **key = EUID + 1000*n**

Die drei rechten Dezimalziffern eines Key sind mit der User-ID identisch, die übrigen Ziffern sind beliebig wählbar.

3.1 KOOPERATION ÜBER GEMEINSAME DATEN

3.1.1 Shared Memory

ERZEUGEN / ÖFFNEN EINES SHARED MEMORY SEGMENTES

Ein Prozeß besorgt sich vom Betriebssystem einen Speicherbereich , der nicht nur von dem Prozeß privat sondern auch von anderen Prozessen global genutzt werden kann durch die Funktion "get shared memory segment identifier" `(shmget() )`.

Syntax:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

int  shmget(key_t key, int size, int flag)
```

Diese Operation ist vergleichbar mit dem Erzeugen einer Datei. Mit der Funktion open() erhält das Programm eine numerische Kennung (Filedeskriptor), über die alle folgenden Operationen auf der Datei ausgeführt werden kann. Entsprechend liefert die Funktion shmget() einen int Wert "shared-memory-identifikation" (SHMID), über die der Zugriff auf den Speicherbereich erfolgen kann. Vergleichbar mit dem Namen einer neu erzeugten Datei, der als logischer Bezeichner fungiert, dient ein numerischer "key" (Datentyp key_t) als eindeutiger Bezeichner für ein Shared-Memory-Segment. Alle Prozesse, die das Shared-Memory-Segment arbeiten wollen müssen den gleichen eindeutigen key verwenden. Der erste Aufruf der Funktion bewirkt das Erzeugen eines Speicherbereiches mit der in *size* angegebenen Größe in Bytes. Alle folgenden Aufrufe kommen von Prozessen, die einen existierenden Shared Memory benutzen wollen. Im Fehlerfall (z.B. Mangel an Speicherplatz) wird ein Returnwert -1 zurückgeliefert.

Mit *flag* kann die Funktionsweise des Aufrufes gesteuert werden

Erzeugen:

flag = IPC_CREAT | IPC_EXCL | 0600

Erzeugen eines Shared-Memory-Sementes (IPC_CREAT). Dem Segment werden Schreib- und Leserechte für alle Prozesse desselben USER gegeben (Oktalzahl 0600)

Existiert ein Segment mit dem gleichen key so wird dies als Fehler gemeldet (IPX_EXCL).

Benutzen:

flag = 0

Der aufrufende Prozeß will ein Shared-Memory-Segment nur benutzen. Der Aufruf dient daher lediglich zur Ermittlung des SHMID. Existiert das Segment nicht, wird der Returnwert -1 für Fehler zurückgeliefert. size ist ohne Bedeutung.

Die vordefinierten Konstanten z.B. IPC_CREAT, IPC_EXCL sind in den Headerdateien `sys/ipc.h` und `sys/types.h` , der Datentyp key_t ist in `sys/types.h` definiert.

ARBEITEN MIT EINEM SHARED MEMORY SEGMENT

Ein durch die SHMID spezifiziertes Shared-Memory-Segment muß zum Adreßraum eines Benutzers hinzugefügt werden, will der Prozeß auf Adressen in diesem Segment zugreifen. Auf die Daten in diesem Segment kann der Prozeß wie über ein array mit Komponenten vom Typ char mit der Größe size zugreifen.

Syntax:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>
```

void *shmat(int shmid, void *addr, int flag);

Wieder abhängen: ***int shmdt (void *shmaddr);***

Die Funktion shmat() fügt den globalen Speicherbereich zum Datenbereich des aufrufenden Prozesses hinzu. Als Ergebnis liefert die Funktion einen Zeiger auf das erste Zeichen des char array zurück. Über diesen Zeiger kann man auf die Daten im shared memory zugreifen.

Im Fehlerfall ist der Returnwert (char *) -1 ansonsten ein Zeiger auf den hinzugefügten Speicherbereich..

SHMID	ist der shared-memory-identifikation (returnwert von shmget())
addr 0 :	das UNIX-System legt den Speicherbereich auf den ersten freien Speicherbereich im Datenbereich des Prozesses
flag	SHM_RND
	SHM_RDONLY

FREIGABE UND LÖSCHEN VON SHARED-MEMORY-SEGMENTEN

Die Freigabe des Speicherbereiches eines Shared-Memory-Segmentes, wenn dieses nicht mehr benötigt wird, geschieht über die Funktion shared memory control operations (**shmctl**)

Syntax:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>

int shmctl( int shmid, int flag, struct shmid_ds   *buffer)
```

Diese Funktion erlaubt die Beeinflussung des Shared-Memory-Segmentes über die Angabe von Flags. Zum Löschen eines Segmentes muß flag auf den Wert IPC_RMID gesetzt werden. In diesem Fall sollte

der dritte Parameter den Wert 0 haben.. Im Fehlerfall wird der Wert -1 zurückgeliefert (z.B. Segment existiert nicht).

Die Datenstruktur shmid_ds ist in der Headerdatei `sys/shm.h` definiert.

3.2 KOMMUNIKATION ÜBER NACHRICHTEN

3.2.1 PIPES

Eine PIPE ist ein unidirektionaler, byteorientierter Datenkanal zwischen zwei UNIX-Prozessen. Eine PIPE ist damit eine FIFO, in die ein Prozeß Daten schreiben und aus der ein anderer Prozeß Daten auslesen kann. Eine PIPE kann maximal 5120 Byte speichern.

Im Normalfall wird die PIPE von einem Prozeß erzeugt und danach durch den Systemaufruf fork() an weitere Sohnprozesse übergeben, die dann automatisch mit der PIPE verbunden sind.. Da durch den fork()-Aufruf eine Kopie erzeugt wird, besitzen beide Prozesse sowohl den Lesezeiger, als auch den Schreibzeiger auf die PIPE. Um einen unidirektionalen Kanal zu erhalten müssen daher beide Prozesse einen von beiden Zugriffszeigern schließen. Im Beispiel Bild 2.1 erzeugt PARENT eine PIPE und danach den Prozeß CHILD. Der PARENT schließt den Lesezeiger, der CHILD den Schreibzeiger, so daß ein Datenfluß vom PARENT zum CHILD entsteht.

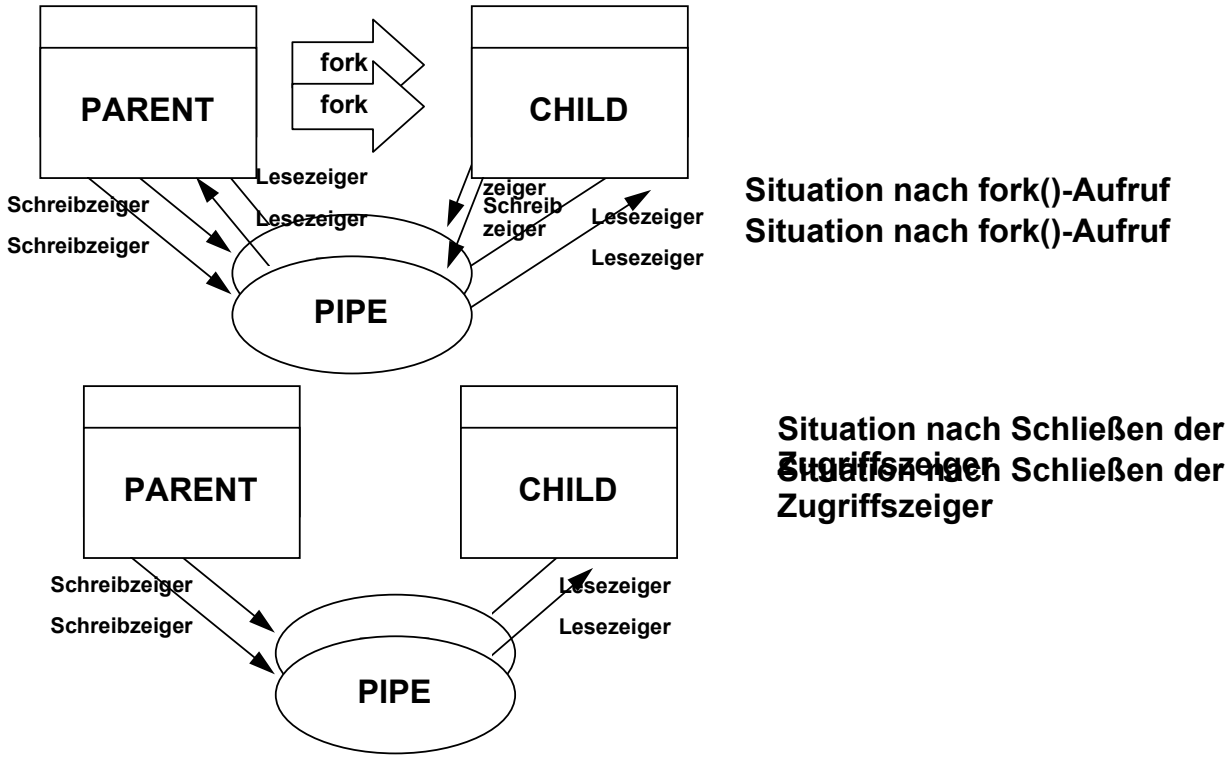

Bild 2.1

Benötigt man einen bidirektionalen Kanal, so müssen zwei PIPE‘s erzeugt werden.

Will ein Prozeß Daten in eine volle PIPE schreiben so wird der Prozeß blockiert, bis durch ein Auslesen von Daten genügend Platz in der PIPE geschaffen wurde.

Eine PIPE wird erzeugt durch die Funktion: `int PIPE(int fildeskriptor[2])`

Durch diese Funktion wird eine PIPE (Datei auf der Festplatte) angelegt und zwei Dateideskriptoren *filedeskrtiptor* (lesen, schreiben) zurückgeliefert. Über *filedeskriptor[0]* kann ein Prozeß Daten aus der PIPE auslesen, mit *filedeskriptor[1]* kann ein Prozeß Daten in die PIPE schreiben.

Im Fehlerfall liefert die Funktion den Wert -1, ansonsten 0.

SCHLIEßEN EINES ZUGRIFFSZEIGERS

Der Systemaufruf

```
void close( int fzeiger);
```

schließt den Dateideskriptor *fzeiger*.

LESEN UND SCHREIBEN

Über die Funktionen write to a file (**write**) und read from file (**read**) können Daten in die PIPE geschrieben und wieder ausgelesen werden.

```
int read(int filedeskript, char *puffer, unsigned anzahl)
```

filedeskriptor ist der Lesezeiger auf die PIPE. Die Funktion liest *anzahl* von Bytes aus der PIPE und schreibt diese in Puffer. Der aufrufende Prozeß wird solange blockiert bis in der PIPE *anzahl* Bytes vorhanden sind oder EndOfFile, dh. kein geöffneter Schreibzeiger existiert, erkannt wird.. In Falle von EndOfFile liefert die Funktion den Wert 0 zurück, andernfalls die Anzahl der gelesenen Daten..

```
int write(int filedeskript, char *puffer, unsigned anzahl)
```

filedeskriptor ist der Schreibzeiger auf die PIPE. Die Funktion schreibt *anzahl* von Bytes aus *puffer* in die PIPE.(siehe ANSI-C Funktion write()). Ist die PIPE voll, so wird der aufrufende Prozeß solange blockiert, bis mindestens anzahl Bytes in die PIPE geschrieben werden können.

2.2.2.1 NAMED PIPES (FIFO)

Bei der Verwendung von PIPE's zur Interprozeßkommunikation gibt es unter UNIX einen großen Nachteil:

Pipes können nur zwischen Prozessen exitieren, die in einer Vater-Sohn-Beziehung zueinander stehen oder die einen gemeinsamen Vaterprozeß besitzen.

Damit zwei beliebige Prozesse Interprozeßkommunikation (IPC) über den PIPE-Mechanismus betreiben können, wurde in System V die " named pipes " , oder auch FIFO genannt , eingeführt. Im Unterschied zu der herkömmlichen PIPE unter UNIX besitzt eine " named pipe " einen Namen über den sich beliebige Prozesse mit der PIPE verbinden lassen.

Eine named pipe wird im Normalfall vom Superuser oder von einen ausgewählten Prozeß durch den Systemaufruf `mknod()` erzeugt. Bevor man aus der PIPE lesen (mit der Systemfunktion read()) oder in die PIPE schreiben kann (Systemaufruf write()) muß die PIPE durch den Systemaufruf open() geöffnet worden sein Ansonsten verhält sich eine named pipe entsprechend der herkömmlichen PIPE unter UNIX.

SYSTEMFUNKTIONEN:

Erzeugen einer named pipe

```
int mknod( char * pathname, int mode, int dev )
```

pathname	ist ein Unix-Verzeichnis und der Name der pipe.
mode	kennzeichnet die Dateizugriffsrechte für den Eigentümer, die Gruppe und alle Anderen. Dieser Wert wird mit dem Flag S_IFIFO aus der Datei <sys/stat.h> logisch oder verknüpft.
dev	wird ignoriert

Öffnen einer pipe: Systemaufruf `open()`

Lesen/schreiben: Systemaufruf `read(), write()`

3.2.2 Nachrichten

Erzeugen einer Nachrichtenwarteschlage mit FIFO-Charakter als Briefkasten für Nachrichten. Jeder Briefkasten wird über einen eindeutigen Bezeichner den **key** identifiziert. Nach Einrichten eines Briefkastens durch das Betriebssystem wird als Returnwert eine Identifikationsadresse zurückgeliefert. Über diese Adresse kann der Prozeß auf Nachrichten in der Nachrichtenwarteschlage zugreifen.

Syntax:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/msg.h>

int msgget( key_t key, int flag)
```

Im Fehlerfall liefert die Funktion den Wert –1 sonst die ID der Messege-Queue.

Mögliche Flagwerte:

flag = IPC_CREAT | IPC_EXCL | 0600

Erzeugen eines neuen Briefkastens (IPC_CREAT)

Dem Briefkasten (FIFO) werden Schreib- und Leserechte für alle Prozesse desselben USER gegeben (0600)

Existiert ein Briefkasten mit dem gleichen key so wird dies als Fehler gemeldet (IPC_EXCL).

flag = 0

Der aufrufende Prozeß will einen schon vorhandenen Briefkasten mit dem Bezeichner KEY benutzen. Die Funktion soll nur die Identifikationsadresse zurückliefern

Existiert die Nachrichtenwarteschlange nicht, wird der Returnwert -1 für Fehler zurückgeliefert.

VERSENDEN UND EMPFANGEN VON NACHRICHTEN

Zum versenden einer Nachricht verwendet man die Systemfunktion :

Syntax:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/msg.h>

int msgsnd( int msgqid, struct msgbuf *buffer, int size, int flag)
```

msgqid gibt den Briefkasten an, in dem man die zu versenden Nachricht ablegen möchte.

Die zu sendende Nachricht steht in einer Datenstruktur vom Typ `msgbuf`. Diese Struktur besteht aus zwei Komponenten:

`long` **`messagetype`**	`Typ der Nachricht. Diese Komponente kann von dem Empfänger der Nachricht in der Funktion msgrcv() angegeben werden, um so nur bestimmte Typen von Nachrichten zu empfangen`
`char` **`messagetext[size]`**	`Puffer, in dem der Text der zu sendenden Nachricht liegt`

Mit flag kann man das Verhalten der Funktion steuern:

flag=IPC_NOWAIT	Der Prozeß wird nicht blockiert.
flag=0	Der Prozeß wird solange blockiert bis die Nachricht in der Nachrichtenwarteschlange abgelegt wurde. Ist die Nachrichtenwarteschlage voll, blockert der Prozeß, bis eine Nachricht ausgelesen wurde.

Zum empfangen einer Nachricht verwendet man die Systemfunktion

```
int msgrcv( int msgqid, struct msgbuf *buffer, int size, long typ, int flag)
```

msgqid gibt den Briefkasten an, in dem man eine Nachricht erwartet.

Ist eine Nachricht angekommen, so wird der Text der Nachricht in der Datenstruktur buffer abgelegt. In size ist die Größe in Bytes der erwarteten Nachricht anzugeben. Ist die Nachricht größer als size, so wird der Text der Nachricht abgeschnitten auf eine Größe von size Bytes.

Mit dem Parameter typ kann man angeben, daß man nur eine Nachricht von einem bestimmten Typ erwartet. Folgende Möglichkeiten sind gegeben:

type=0	Die erste / nächste Nachricht der FIFO (Briefkasten) soll geholt werden
type>1	Die erste Nachricht, die im Briefkasten durch den Nachrichtentyp type gekennzeichnet ist soll empfangen werden
type <0	Die erste Nachricht mit dem kleinsten Typ kleiner oder gleich dem Wert von type soll empfangen werden

Mit FLAG kann man den Modus festlegen:

flag= IPC_NOWAIT	Der Prozeß soll nicht blockiert werden, wenn keine Nachricht im Briefkasten abgelegt ist. Ist keine Nachricht vorhaden ist der Returnwert -1 sonst 0.
flag=0	Der Prozeß wird solange blockiert bis eine Nachricht vom angebenen Typ aus dem Briefkasten genommen werden kann.

3.3 PROZESSSYNCHRONISATION

3.3.1 Signale

Signale unter Unix sind Ereignisse, die asynchron einen Prozess unterbrechen können. Bei eintreffen eines Signals wird der laufende Prozess unterbrochen und ein Signalhandler abgearbeitet. Enthält der Signalhandler die Funktion exit() beendet der prozess. Ruft der Signalhadler nicht die Funktion exit() auf, wirkt der Signalhandler wie ein Unterprogramm, d.h. nach Abarbeitung des Signals arbeitet das unterbrochene Programm ab der unterbrochenen Stelle wieder weiter.
Unix kennt einige vordefinierte Signale, auf die das Betriebssystem mit einer vorgegebenen Aktion (vordefinierter Signalhandler) reagiert.

Für viele dieser Signal kann ein Prozess einen eigenen Signalhandler angeben, wie der Prozess auf das Signal reagieren möchte und somit die vordefinierte Aktionen überschreiben.
Die Signale werden normal über symbolische Konstanten vom Programm angesprochen z.B. SIGTERM ist ein von der Software gesendetes Signal zum beenden des Prozesses. SIGUSR1 und SIGUSR2 sind Signale, die von einem Anwenderprogramm frei verwendet werden können.
In älteren UNIX System V wurden 15 Signale zu Verfügung gestellt. Ab Unix system V R4 stehen 30 Signale zu Verfügung.

Beispiele für die wichtigsten Signalnummern und deren vordefinierte Aktionen:

SIGCHLD	Dieses Signal wird an den Elternprozess gesendet, wenn ein Sohn-Prozess terminiert. Wird ignoriert.
SIGINT	Signalnummer für Eingabe von STRG+C. Beendet den Prozess.
SIGKILL	Dieses Signal beendet den Prozess auf jeden Fall, da das Signal nicht mit einem eigenen Signalhandler abgefangen werden kann.
SIGTERM	Signalnummer, die vom kill –Befehl versendet wird zum beenden eines Prozesses.
	Dieses Signal kann im Unterschied zu SIGKILL durch einen eigenen Signalhandler abgefangen werden.
SIGUSR1	Benutzerdefiniertes Signal. Ist für die Verwendung von eigenen Systemprogrammen gedacht. Beendet den Prozess.
SIGUSR2	Benutzerdefiniertes Signal. Ist für die Verwendung von eigenen Systemprogrammen gedacht. Beendet den Prozess.

Nähere Informationen zu Signalen sind aus gängigen UNIX-Büchern zu entnehmen.

SYSTEMFUNKTIONEN

Verwendete Headerdatei:

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

Senden eines Signals an einen Prozess

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

Sendet ein Signal an einen anderen Prozess.

pid ist die PID des Prozesses, an den das Signal gesendet werden soll.

signal ist die Signalnummer des Signals das gesendet werden soll.

Warten auf das Eintreffen eines Signals

Mit der Systempunktion `int pause(void )`

kann ein Prozess blockieren, bis ein Signal eintrifft. Nach Eintreffen des Signals wird der Signalhandler abgearbeitet und der prozess ist danach wieder deblockiert.

Einrichten eines einfachen Signal-Handlers

Die Systemfubktion signal() richtet einen eigenen Signalhandler ein. Sie erwartet zwei Parametr, einmal die Nummer des Signals, auf das der Handler reagieren soll und zweitens die C-Funktion, die aufzurufen ist, wenn das Signal auftritt. Die Funktion gibt einen zeiger auf den Signalhandler, der überschrieben wird, zurück.

```
#include <signal.h>


void (*signal(int signal, void(*function(int))) (int)
```

3.3.2 Zeitliches Blockieren eines Prozesses

Um einen Prozess für bestimmte Zeit schalfen zu legen gibt es die Systemfunktion sleep().

Syntax: #include <unistd.h>

Unsigned int sleep(unsigned int sekunden);

Die Funktion blockiert den aufrufenden prozess solange bis die in sekunden angegene Zeit verstrichen ist oder aber der prozess durch ein Signal unterbrochen wurde.

Returnwert. 0 falld er prozess seine angegebenen sekunden blockiert war
>0 falls der prozess durch ein Signal deblockiert wurde wird im Rückgabewert die Anzahl der nicht geschfanen Sekunden angegeben.

3.3.3 Semaphore

Anlegen von allgemeinen Semaphoren

Durch die Funktion semget() (get semaphor) können sowohl allgemeine Semaphore als auch einen Menge von Semaphoren (Mengensemaphore) angelegt werden.

```
int semget (key_t key, int nsems, int flag)
```

nsems gibt die Anzahl der Semaphore an. Hat nsem den Wert 1 so wird nur ein Semaphor angelegt. Die Bedeutung von key und flag ist dieselbe wie bei shmget().

Initialisieren und Löschen von semaphoren

Nach dem Erzeugen des Semaphors hat dieser automatisch den Wert 0. In den meisten Anwendungen müssen jedoch Semaphore mit bestimmten Werten vorbelegt werden um eine korrekte Synchronisation zu ermöglichen. Mit der Funktion semctl() kann ein angelegter Semaphor sowohl initialisiert als auch wieder gelöscht werden.

```
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (int semid, int semnum, int flag, union semun value)
```

flag = SETVAL

Setzt Semaphor auf einen gewünschten Wert, der in *value* anzugeben ist.

*In diesem Fall wird value alsUnionkomponente **val** (int) interpretiert.*
semnum ist der Index des zu beeinflussenden Semaphors im Array.
Wenn man nur einen Semaphor erzeugt hat, ist der Index natürlich 0.

Die Konstante SETVAL ist in sys/sem.h definiert.

```
union semun {    int               val;          /* für flag SETVAL
*/
                 struct semid_ds   *buff;
                 ushort            *array;
             }
```

flag = IPC_RMID

Löschen des angegebenen Semaphors bzw. des gesamten Arrays. *semnum und value* sind ohne Bedeutung und können zu 0 gesetzt werden.

Im Fehlerfall ist der Returnwert -1, sonst 0.

SYNCHRONISATIONSFUNKTIONEN -ANFORDERN(P) UND ABGEBEN(V) SEMAPHOR

Mit der Funktion **semop**() (semaphor operations) lassen sich die Synchronisationsoperationen P (wait) und V (signal) realisieren,. Da System V Mengensemaphore unterstützt, können mit einem Aufruf gleichzeitig mehrere Semaphore einer angelegten Menge von Semaphoren als unteilbare Operation beeinflußt werden. Hierbei ist es möglich auf einen Semaphor der Menge eine P-Operation und auf einen anderen Semaphor gleichzeitig eine V-Operation auszuführen. Die Anwendung dieser Funktion ist daher recht kompliziert.

```
int semop (int semid, struct sembuf buffer[], unsigned nops)
```

Die Anzahl der gleichzeitigen Semaphore/Operationen gibt der Parameter nops an.

Zur Spezifikation der Operation für einen einzelnen Semaphor dient eine Struktur *sembuf.*

Um alle nops Operationen für die Semaphore zu spezifizieren gibt man ein Array mit nops Komponenten vom Typ struct sembuf an.

Im Normalfall hat nops den Wert 1 und das Array nur eine Komponente.

Eine sembuf-Komponente besitzt in `sys/sem.h` folgende Strukturdefinition:

```
struct    sembuf {
                    short sem_num; /* Index des Semaphors, i.a. 0 */
                    short sem_op;  /* Operation auf den Semaphor */
                    short sem_flg; /* Flag zur Steuerung der Operation, i.a.
0 */
                    }
```

Die Komponente sem_op gibt den Zahlenwert an, um den der Semaphor verändert werden soll

Eine P-Operation wird durch negative Zahlenwerte realisiert eine V-Operation durch positive Zahlenwerte. Eine P-Operation deren Ergebnis auf negative Werte des Semaphors führen blockieren den aufrufenden Prozeß. Eine V-Operation kann blockierte Prozesse wieder rechenbereit machen, wenn der Wert des Semaphors positiv wird.

Der Returnwert ist normalerweise 0, im Fehlerfall -1 (z.B. bei fehlendem Schreibrecht des Aufrufers).

4 Wichtige Shell-Kommandos zur Abfrage von Prozessen und IPC-Objekten

PS	Prozessstatus abfragen
KILL	Terminiert einen laufenden Prozess (auch wenn dieser im Zustand blockiert ist).
IPCS	Status von IPC-Objekten (Semaphore, Shared Memory, Message Queues) abfragen
IPCRM	Löschen vorhandener IPC-Objekte

Die Syntax und die Optionen können sie sich über die man-page erfragen.

LITERATURVERZEICHNIS

/1/ Banahan & Rutter,: UNIX lernen, verstehen, anwenden, Carl Hanser Verlag, 1987, ISBN 3-446-13975-3

/2/ Bach, Maurice J.: UNIX - Wie funktioniert das Betriebssystem, Carl Hanser Verlag, 1991, ISBN 3-446-15693-3

/3/ Bourne, Stephen R., Das UNIX System V, Addison Wesley, 1987, ISBN 3-925118-23-2

/4/ Curry, David A. : Using C on the UNIX System. A guide to system programming. O`Reilly & Associates, Inc., 1989, ISBN 0-937175-23-4

/5/ Gulbins, Jürgen : UNIX, 3. Auflage, Springer Verlag, 1988, ISBN3-540-13242-2

/6/ Rochkind, M.: UNIX Programmierung für Fortgeschrittene, Hanser Verlag, 1991

/7/ W. Richard Stevens: Programmierung von UNIX-Netzen, Grundlagen, Programmierung, Anwendung, Carl Hanser Verlag,, 1992, ISBN 3-446-16318-2

/8/ W. Richard Stevens: Programmierung in der Unix Umgebung, Addison Wesley, ISBN 3-89319-814-8

/9/ Herold, Helmut: UNIX und seine Werkzeuge, UNIX-Shells, Addison-Wesley,1993, ISBN 3-89319-381-2

/10/ Herold, Helmut: UNIX und seine Werkzeuge, Shared Libraries und Interprozeß-Kommunikation, Addison-Wesley

/11/ H. Herold: Linux - Unix Systemprogrammierung, Addison Wesley ISBN 3-8273-1512-3

/12/ Güting, Ralf Hartmut : Datenstrukturen und Algorithmen, Teubner Verlag, 1992, ISBN 3-519-02121-8

/13/ Wirth, Niklaus : Algorithmen und Datenstrukturen, Teubner Verlag, 1983, ISBN 3-519-02250-8

Der ASTA verkauft eine einführende Broschüredes Regionalen Rechnezentrums Niedersachsen zum Thema UNIX.