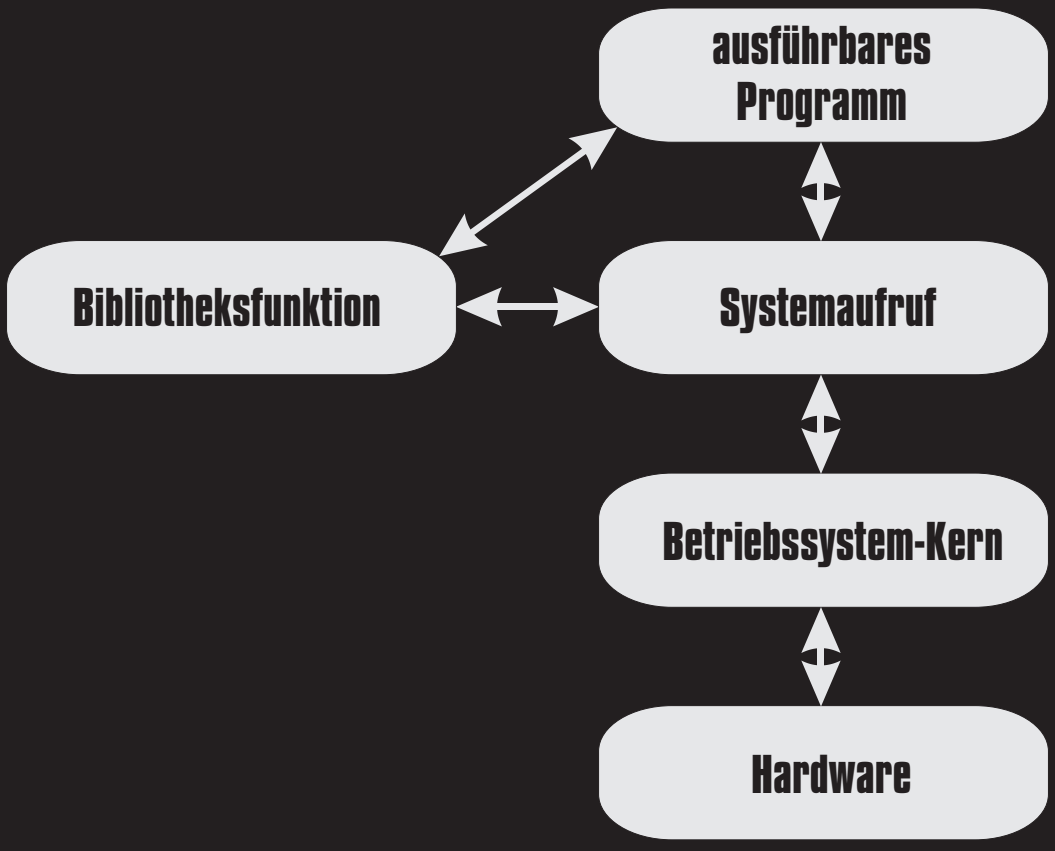
Lutz Wegner

Ausgewählte Themen zu Rechnernetzen

Universität
Kassel



SKRIPTEN DER
PRAKTISCHEN INFORMATIK

Lutz Wegner

Universität Gh Kassel

FB Mathematik/Informatik

D-34109 Kassel

2. durchgesehene Folienvorlage zur Vorlesung im Sommersemester 1999

Vorwort

Es gibt mindestens zwei grundsätzlich verschiedene Arten, das Thema Rechnernetze anzugehen:

- das Netzwerk, seine Topologie, seine Übertragungstechnik, die Protokolle, usw. studieren;
- die Programmierung in einem Rechnernetz, die Prozeßkommunikation, usw. studieren.

Vertreter der ersten Richtung sind Tanenbaum, Peterson&Davie, Proebster. Die zweite Richtung vertritt z.B. Stevens[1] und Gray.

Im Prinzip braucht man beides. Beim ersten Thema ändern sich die Techniken recht schnell, man denke an ATM, usw. Grundlagen, wie z.B. graphentheoretische Betrachtungen zur Definition der Ausfallssicherheit und wie z.B. Leistungsabschätzungen für Protokolle mit statistischer Zugangskontrolle, (was landläufig als Ethernet bezeichnet wird), sind dagegen unverändert gültig.

Das zweite Thema lehnt sich an Betriebssysteme an, spez. die Prozeßsynchronisierung. Große Bedeutung kommt dabei den UNIX-Techniken *Threads* und *Sockets* zu, die sich für Mehrbenutzer- und Client-Server-Anwendungen anbieten. Diese programmtechnische Seite scheint auch relativ stabil zu sein. Wir werden ihr hier auf der Grundlage des Buches von John Shapley Gray nachgehen.

---

1. Richard Stevens verstarb am 1. September 1999

Sein Inhalt wird wie folgt beschrieben:

> Extensive coverage of named and unnamed pipes, message queues, semaphores, and shared memory. Practical, detailed explanations of RPC and sockets. Thorough coverage of communication in multi-threaded applications using POSIX threads.

Weitere wichtige Quellen:

Den Quelltext zu dem Buch von Stevens: UNIX Network Programming, Vol. I, und weitere nützliche Verweise finden sich auf der „Nachlass-Seite“ von Stevens:

```
http://www.kohala.com/start/
```

Die Quellen zu dem Buch von Gray, der an der University of Hartford in Conneticut lehrt, finden sich unter

```
http://www.cs.hartford.edu/~gray/
```

oder bei uns auf der Web-Seite zur Vorlesung.

```
http://www.db.informatik.uni-kassel.de/
```

Inhaltsverzeichnis

1 Programme und Prozesse

1.1 Einleitung

In allen Betriebssystemen ist die grundlegende Einheit der Programmausführung der *Prozeß*. Ein Prozeß ist ein sich in Ausführung befindendes Programm. Neben dem Programmcode und den Datenteilen gehört dazu sein momentaner Status (laufend, bereit, wartend), die zugehörigen Ressourcen, die Register- und Stapelinhalte und Zeiger, der Programmzähler.

Ein Prozeß ist demnach etwas dynamisches. In UNIX *scheinen* zu jedem Zeitpunkt auch mehrere Prozesse gleichzeitig zu laufen. Ferner *sieht es so aus*, als könnten sie auf alle Betriebsmittel zugreifen, wie wenn sie alleine (isoliert) liefen.

Beide Sichten sind eine Illusion. Die meisten UNIX-Systeme laufen auf Plattformen mit nur einer CPU, die auch zu jedem Zeitpunkt nur einen Prozeß aktiv abarbeiten kann. Durch schnelles Hin- und Herschalten der CPU können jedoch mehrere Prozesse quasi-gleichzeitig (nebenläufig) mit Rechenzeit versorgt werden.

Die Fähigkeit, die Betriebsmittel zwischem mehreren Prozessen zu multiplexen, nennt man **multiprogramming** oder **multitasking**.

Geschieht die Verteilung auf Zeitscheibenbasis, spricht man auch gern von time-sharing (berühmter Spruch: J.H.: If this is time-sharing, give me my share right now! D.R.: It’s not time yet.) Bei Mehrprozessorsystemen ist dann echte Nebenläufigkeit möglich, man spricht von **multiprocessing**.

Ein Prozeß ist also die Ausführung eines Programms. Wird ein Programm mehrfach aufgerufen, kann es mehrfache Prozesse erzeugen. Demnach existieren Programme in zwei Formen:

- als Quelltext (**source program**) , d.h. als (Folge von ASCII-Zeichen im Wertebereich 32-127), z.B. als C- oder Java-Programm , das anschaubar ist und meist in UNIX das Ausführungsrecht nicht gesetzt hat (x-flag ist -). Ein Beispiel wäre das C-Programm für dreimal „Hello World“ unten.
- als ausführbares Programm (**executable program**), das mittels Compiler und Assembler/Linker in maschinenlesbare und abarbeitbare Form gebracht wurde und das man sich in der Regel nicht mehr anschauen oder ausdrucken kann.

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

/*  Display Hello World 3 times  */
char *cptr = „Hello World\n“; /* static by placement */
char buffer1[25];
void
main(void){
 void showit(char *);
 int i = 0; /* automatic variable */
 strcpy(buffer1, „A demonstration\n“); /* lib function*/
 write(1, buffer1, strlen(buffer1)+1); /* system call */
 for (; i < 3; ++i)
  showit(cptr); /* function call */
}

void
showit(char *p){
 char *buffer2;
 if ((buffer2=(char *) malloc((unsigned)
     (strlen(p)+1)))!= NULL){
  strcpy(buffer2, p);
  printf(„%s“, buffer2);
  free(buffer2);
 } else {
```

```
    printf(„Allocation error.\n“);
    exit(1);
  }
}
```

1.2 Bibliotheksfunktionen

Größere Programme werden üblicherweise aus Funktionen aufgebaut. Eine Funktion ist eine Sammlung von Vereinbarungen und Anweisungen zur Erledigung einer speziellen, abgeschlossenen Aufgabe und liefert in der Regel einen Wert ab.

Funktionen (**functions**) können vom Benutzer selbst definiert oder vorab definiert und dem Benutzer zur Verfügung gestellt sein. Solche vorab definierten Funktionen, die von ihrer Funktionalität her verwandt sind, werden im sog. Objectcodeformat (**object code format)** in Bibliotheksdateien - **library** (archive) **files** - gespeichert.

Sie sind damit in einem Zwischenformat, das noch das Binden mit anderen library files und mit Quellcode erlaubt. Andererseits kann man sie auch nicht mehr anschauen. Derartige Funktionen in Bibliotheken heißen oft Bibliotheksfunktionen (**library functions**) oder Laufzeitbibliotheksroutinen (**run-time library routines**).

In den meisten UNIX-Systemen heißt das Verzeichnis für die Bibliotheksdateien `/usr/lib`. Weitere Hilfsdateien befinden sind in `/usr/local/lib`. Gewöhnlich beginnen Bibliotheksdateien mit dem dreibuchstabigen Präfix **`lib`** und enden mit der Erweiterung **`.a`**.

Mit dem Archivkommando **`ar`**, das Archive pflegt, kann man ihre Inhalte ansehen (Achtung: Optionen in Kommandozeile ohne **`-`**). Beispiel:

```
ar t /usr/lib/libc.a | pr -4 -t
```

Die Option **`t`** liefert das Inhaltsverzeichnis (**t**able of contents).

Der Objektcode der Dateien wird beim Übersetzen von C-Quellprogrammen automatisch dazugebunden. Eine Referenz auf `printf` (`strcpy`)

in einem C-Programm führt daher zum Dazubinden des Objectcodes für die `printf` (`strcpy`) Funktion aus `/usr/lib/libc.a`.

Mit dem Kommando `man -k '(3)'` kann man sich eine einzeilige Zusammenfassung (Synopsis) aller Bibliotheksfunktionen anzeigen lassen, sofern die (Gray: `windex`, AIX: `whatis`) Datenbasis aufgebaut wurde.

Auszug aus dem `man`-Manual:

```
-k Displays each line in the keyword database that
contains a string of characters matching the title given
as the final parameter. You can enter more than one
title, each separated by a space. To use the -k flag, a
root user must have previously entered catman -w to
create the /usr/share/man/whatis file.
```

Der Aufruf von `man -k` ist äquivalent zum `apropos`-Befehl in AIX.

1.3 Systemaufrufe (system calls)

Einige der vorher genannten Bibliotheksfunktionen sind in Wirklichkeit **Systemaufrufe** (**system calls**), die der Betriebssystemkern (**kernel**) selbst ausführt. Die Aufrufe ähneln im Format den Funktionen. Da es in Systemaufrufen um den Zugriff auf kritische Resourcen geht, z.B. eine Schreib-/Leseoperation auf eine Platte, das Löschen eines Dateiknotens, etc.) geht das Betriebssystem für die Dauer der Ausführung eines Systemaufrufs in den priviligierten Zustand (**system mode** statt **user mode**) über. Dies hat einen sog. Kontextwechsel (**context switch**) zur Folge, der relativ teuer (aufwendig) ist.

Section 2 des Handbuchs enthält die Systemaufrufe, wobei - wie gesagt - viele Aufrufe in Funktionen verpackt sind, z.B. **`read`** und **`write`** in den Funktionen **`scanf`** und **`printf`**.

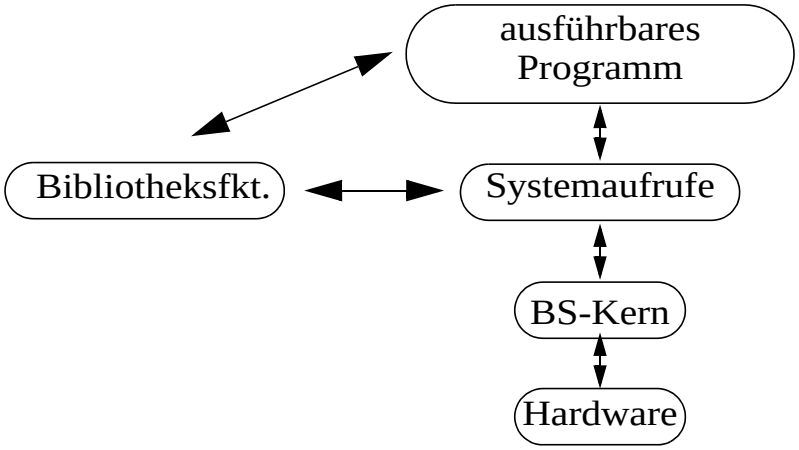


Abb. 1–1

1.4 Das Binden von Objektcode

Code aus den Bibliotheksdateien wird bei der Übersetzung nach Bedarf dazugebunden. Andere, zusätzliche Dateien mit Objektcode für Systemaufrufe und Bibliotheksaufrufe, die nicht in der Standard C-Bibliothek enthalten sind, müssen zur Übersetzungszeit angegeben werden.

Dies geschieht mit der **`-l`** Option, gefolgt von dem Bibliotheksnamen ohne Präfix `lib` und ohne Erweiterung **`.a`**. Ein C-Compileraufruf könnte lauten

```
cc prgm.c -lm
```

und würde den Objektcode aus der Mathematikbibliothek `libm.a` zum Quellprogramm `prgm.c` binden.

Häufig werden zusätzliche Spezifikationsdateien (**header files**) benötigt, die Funktionsköpfe, Makrodefinitionen, definierte Konstanten, usw. enthalten. Ohne diese Angaben ist eine korrekte Übersetzung nicht möglich. Umgekehrt funktioniert die Übersetzung auch nicht richtig, wenn zwar die header files angegeben werden, der Verweis auf die passenden Objektbibliothek fehlt.

In den Manual-Seiten stehen auch immer die benötigten headerfiles. Stehen mehrere drin, ist deren Reihenfolge ggf. relevant, da eine Datei die

Einbindung einer anderen voraussetzt, usw. Dieser Abhängigkeitsgraph ist auch der Grund für die Einbindung des `<sys/types.h>` Header-Datei vor allen anderen Headers. Die Notation `<sys/types.h>` mit spitzen Klammern deutet an, daß die Datei `types.h` am gewöhnlich dafür vereinbarten Ort, d.h. bei UNIX-Systemen meistens in `/usr/include` im Unterverzeichnis `sys`, zu finden ist.

1.5 Fehlerbehandlung

Gewöhnlich gibt ein Systemaufruf oder eine Bibliotheksfunktion im Fehlerfall den Wert -1 zurück und setzt eine globale Variable namens **`errno`** auf einen Wert, der den Fehler anzeigt. Funktionen, die einen Zeiger (z.B. auf eine Zeichenkette) zurückgeben, liefern im Fehlerfall den NULL-Zeigerwert zurück.

Die Headerdatei `<sys/errno.h>` enthält die Konstanten aller Fehlercodes.

Man sollte es sich angewöhnen, grundsätzlich nach einem Funktionsaufruf den Rückgabewert zu testen, um festzustellen, ob er erfolgreich war. Bei einem Fehler, sollte man die notwendigen Schritte unternehmen, z.B. eine kurze Fehlermeldung auszugeben und das Programm mit `exit` zu verlassen.

Die Biblitheksfunktion `perror` liefert eine solche Fehlermeldung.

Gray gibt für alle erwähnten Systemaufrufe und Funktionen eine Zusammenfassungstabelle an, die ein Kondensat der Manualseite ist. Für `perror` sieht sie z.B. so aus.

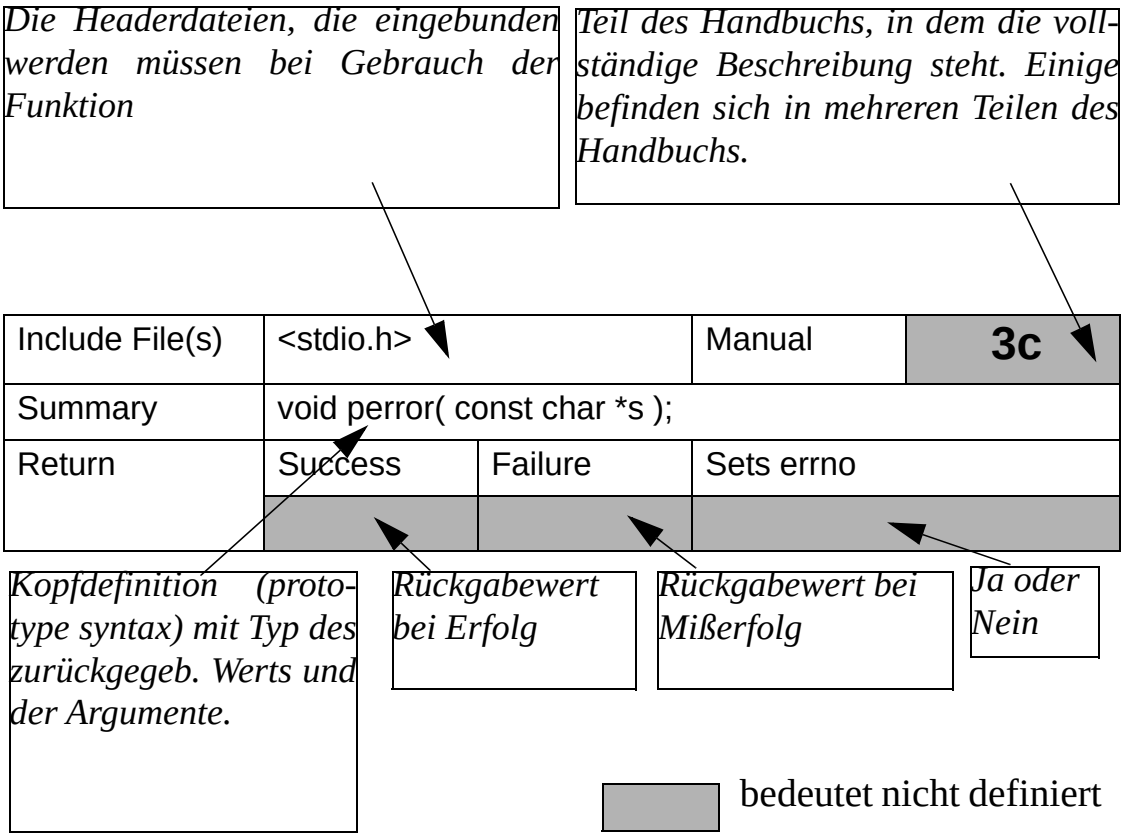


Abb. 1–2 *Beispiel für perror*

Demnach müssen wir `<stdio.h>` einbinden, wenn wir `perror` benutzen wollen. Dagegen brauchen wir die vorher erwähnte Headerdatei `<sys/errno.h>` nicht, es sei denn, wir wollten auf spezielle Errorcodes zugreifen.

Das Argument ist ein einzelner Pointer auf eine Zeichenkettenkonstante und `perror` liefert nichts zurück und setzt `errno` nicht bei Mißerfolg.

Hier ein Beispiel für den Gebrauch von `errno` und `perror`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
/* Checking errno and using perror */
extern int errno;
void
main(int argc, char *argv[]) {
 int n_char = 0, buffer[10];
 /* Initially these should be both 0 */
 printf(„n_char = %d \t errno = %d \n“, n_char, errno);
 /* Display a prompt to stdout */
 n_char = write(1, „Enter a word „, 14);
 /* Use the read system call to obtain up to 10
  characters from stdin */
 n_char = read(0, buffer, 10);
 printf(„\nn_char = %d \t errno = %d \n“, n_char, errno);
 /* If the read has failed ... */
 if (n_char == -1) {
  perror(argv[0]);
  exit(1);
 }
 /* Display the characters read */
 n_char = write(1, buffer, n_char);
}
```

Man beachte, daß `errno` als externe Integervariable deklariert sein muß, um sie im Programm benutzen zu können.

Läßt man das Programm laufen, zeigt die erste Ausgabe an, daß `n_char` und `errno` beide den Wert 0 besitzen. Gibt der Aufrufer das Wort *testing* bei Eingabeaufforderung an, sieht die Ausgabe wie folgt aus:

```
wegner@elsie(tests)$ ./a.out
n_char = 0 errno = 0
Enter a word testing

n_char = 8 errno = 0
testing
wegner@elsie(tests)$ ./a.out
n_char = 0 errno = 0
Enter a word testing further

n_char = 10 errno = 0
testing fuwegner@elsie(tests)$ rther
bash: rther: command not found
wegner@elsie(tests)$
```

Wie man sieht, springt die Fehlerroutine auch bei einer zu langen Eingabe nicht an, es verbleibt einfach der Rest der Eingabe im Puffer, hier „rther“, wodurch die Shell, hier die bash, eine Fehlermeldung „unbekanntes Kommando“ erzeugt.

Erst wenn wir z.B. für den read-Aufruf eine unbekannte Dateinummer (z.B. file id 3) eintragen, liefert `read` den Wert -1 ab und die Fehlervariable `errno` wird auf 9 gesetzt.

Die Ausgabe lautet jetzt:

```
wegner@elsie(tests)$ cc p12.c
wegner@elsie(tests)$ ./a.out
n_char = 0 errno = 0
Enter a word
n_char = -1 errno = 9
./a.out: Bad file number
wegner@elsie(tests)$
```

Wenn man will, kann man dem Aufruf von `perror` noch einen eigenen Text mitgeben, der dann - getrennt durch den Doppelpunkt `:` - vor der Systemfehlermeldung erscheint.

Ferner gibt es zwei weitere externe Variablen

```
extern char *sys_errlist[]
```

und

```
int sys_nerr.
```

Erstere ist ein Zeiger auf einen externen Zeichenkettenvektor, die andere die höchste Fehlermeldungsnummer. Mitteln `errno` kann man demnach den Vektor `sys_errlist[]` indizieren.

Das macht im Prinzip auch `strerror`, dem man die Fehlernummer, also `errno`, als Argument übergibt und das einen Pointer auf die Fehlermeldung zurückliefert.

Include File(s)	<string.h>		Manual	**3c**
Summary	char *strerror(int errnum);			
Return	Success	Failure	Sets errno	
	Zeiger auf Fehlerbotschaft			

Tab. 1–1 *strerror*

Man beachte auch, daß der `errno` Wert nicht zurückgesetzt wird, auch nicht bei einem anschließenden erfolgreichen Aufruf einer Funktion. Die Variable enthält also weiterhin den Wert des letzten fehlerhaften Aufrufs.

Übung 1–1

Schreibe ein Programm, das alle verfügbaren Fehlermeldungen in einer durchnumerierten, zweispaltigen Ausgabe auflistet. Gibt es eine Ausgabe für Fehlernummer 0? Sollte es eine geben?

Übung 1–2

Das erste Argument im Systemaufruf `read`/`write` ist ein Integerwert, der den Dateideskriptor darstellt. Jedes ausgeführte Programm hat bekanntlich drei Dateideskriptoren automatisch zugeteilt und geöffnet: `stdin` (Standardeingabe, fid 0, üblicherweise die Tastatur), `stdout` (Standardausgabe, fid 1, üblicherweise der Bildschirm) und `stderr` (Standardfehlerausgabe, fid 2, auch Bildschirm). Würde im letzten Programm 1.2 write nach 0 schreiben (der Standard**eingabe**), würde das Programm weiterhin übersetzt und ausgeführt werden, eine Ausgabe produzieren und **keine** Fehlermeldung - Warum?

Übung 1–3

Man schreibe eine freundlichere Fehlerroutine für Dateimanipulierungsfehler, als `perror` dies darstellt. Dazu schaue man in die Datei `<sys/errno.h>` und beachte den Handbucheintrag für `Intro` im Teil 2 (d.h. `man -s2 Intro`).

1.6 Ausführbares Dateiformat

In UNIX werden Dateien, die übersetzt wurden, in einem ausführbaren Format, genannt `a.out` Format, abgespeichert. Dateien im `a.out`-Format enthalten einen speziellen Vorspann (für Hardware-/Programmeigenschaften), Programmtext, Daten, Verschiebbarkeitsinformation, Symboltabelle und Zeichenkettentabelleninformation.

Dateien im `a.out`-Format werden vom Betriebssystem als ausführbar markiert und können durch Eingabe des Dateinamens von der Kommandozeile aus aufgerufen werden.

Um den Leser ganz zu verwirren sei auch erwähnt, daß übersetzte C-Programme vom Compiler standardmäßig den Namen `a.out` erhalten.

Übung 1–4

Die Struktur des Vorspanns im `a.out`-Format wird von der `exec`-Struktur bestimmt. Diese findet sich, je nach UNIX-System, in einer der Header-Dateien `<sys/a.out.h>` oder `<sys/exec.h>`. Man erstelle ein Programm, daß ausführbare Dateien auf ihre Ausführbarkeitsvoraussetzungen prüft. Man beachte auch die Information in `/etc/magic` zur Identifikation von `a.out` Dateien.

1.7 Speicheraufteilung des Systems

Unter UNIX wird ein vom BS-Kern eingelesenes und zur Ausführung gebrachtes Programm zum Prozeß. Den Hauptspeicher können wir uns als in zwei Teile getrennt vorstellen: Erstens den Benutzerraum (**user space**), in dem Anwenderprogramme laufen. Einzelne Programme darin werden voneinander getrennt. Man nennt sie Benutzerprozesse (**user processes**), die im Benutzermodus (**user mode**) laufen.

Zweitens gibt es den Systembereich (**kernel space**), den Benutzerprozesse nur über Systemaufrufe betreten können. In diesem Fall ist ein Prozeß temporär im Systemmodus (**kernel mode**) und hat spezielle Rechte ähnlich wie root. Den Wechsel vom Benutzer- in den Systemstatus bezeichnet Gray als **context switch** (Hinweis: Oft versteht man unter context switch auch allgemein einen Prozeßwechsel).

1.8 Prozeßspeicher

Ein Benutzerprozess im Hauptspeicher ist in drei Teile (Segmente, Bereiche; segments, regions) aufgeteilt: **Text**, **Daten** und **Stapel**.

- Das Textsegment enthält die ausführbaren Instruktionen und konstanten Daten. Es ist als nicht-beschreibbar (read-only) markiert und kann vom Prozeß nicht verändert werden. Üblicherweise teilen sich mehrere ablaufende gleiche Programme ein Textsegment, es sei denn, es wäre mit der Übersetzeroption `-N` übersetzt worden.
- Das Datensegment, daß sich fortlaufend (im Sinne fortlaufender virtueller Speicher) an das Textsegment anschließt, enthält

zunächst die initialisierten Daten (in C Variablen, die als static vereinbart wurden oder die statisch durch ihre Plazierung sind). Danach kommen die nichtinitialisierten Daten. Im Beispielprogramm 1.1 wäre `cptr` im initialisierten Teil, `buffer1` im nichtinitialisierten Teil zu finden. Wärend der Laufzeit kann mit `malloc` zusätzlicher dynamischer Speicherplatz angefordert werden. `malloc` oder z.B. `calloc` rufen ihrerseits system calls `brk` und `sbrk` auf, um das Datensegment (nach unten, vgl. Abbildung) zu erweitern. Diesen Bereich nennt man auch den Heap.

- Das Stapelsegment (stack segment) wird vom Prozeß für sog. automatische Bezeichner, Registervariablen und Funktionsaufrufinformation benutzt, In der Funktion `main` wäre dies der Bezeichner `i`, `buffer2` in der Funktion `showit`, sowie die Aufrufinformation (stack frame) für den Aufruf von `showit` innerhalb der `for`-Schleife. Üblicherweise wächst der Stapel gegen den Heap.

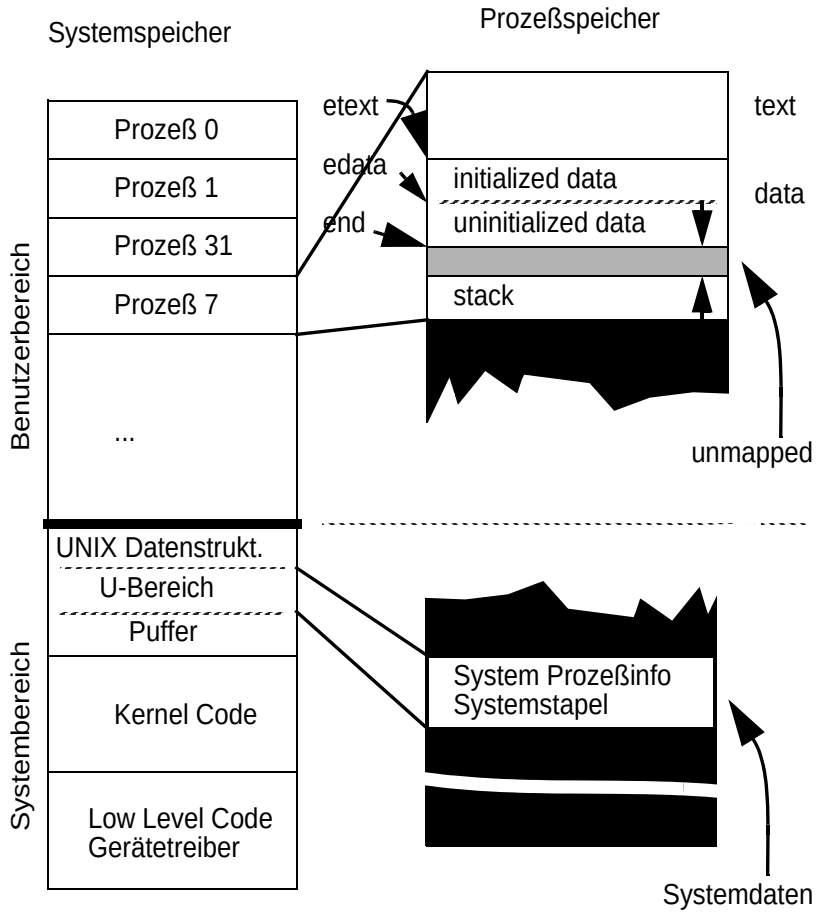


Abb. 1–3

1.9 Der U-Bereich

Zusätzlich zu den drei Segmenten hält das Betriebssystem für jeden Prozeß noch einen geschützten Bereich vor, den U-Bereich (**u area**, user area). Dieser Bereich enthält Kontrollinformation für den Prozeß, z.B. die offenen Dateien, das Arbeitsverzeichnis, Signale, Abrechnungsinformation) sowie Platz für einen Systemstapel.

Wenn der Prozeß einen Systemaufruf macht, z.B. den Aufruf `write` in `main` in Programm 1.1, wird der Aufrufsrahmen (**stack frame information**, manchmal auch Funktionstemplate genannt) auf dem Systemstapel abgelegt. Auf diese Daten hat der Anwenderprozeß keinen direkten Zugriff, sondern nur über spezielle Systemaufrufe. Die Struktur des U-Bereichs wird durch das Header-File `<sys/user.h>` definiert.

1.10 Prozeßadressen

Das System weiß natürlich, wo jedes der drei Segmente im virtuellen Speicher liegt und bietet die Information dem Anwenderprozeß über die externen Variablen `etext`, `edata` und `end` an. Die Adressen dieser drei Variablen (nicht ihr Inhalt!) entsprechen der ersten gültigen Adresse über dem Text, initialierten Daten und nicht-initialisierten Daten. Programm 1.3 zeigt, wie man diese Information ausgeben kann.

```
#include <stdio.h>
/*
 Programm 1.3 Displaying process segment addresses,
 angepasst für RS6000/AIX (Stephan Wilke 27.4.98)
*/
extern int _text, _etext, _data, _edata, _end;
void
main(void){
  printf(„\
  _text: %X [Specifies the first location of the
program]\n\
  _etext:%X [Specifies the first location after the
program]\n\
  _data: %X [Specifies the first location of the data]\n\
  _edata:%X [Specifies the first location after the
initialized data]\n\
  _end: %X [Specifies the first location after all
data]\n“, &_text,&_etext,&_data,&_edata,&_end);

}
```

Wenn wir Programm 1.1 entsprechend modifizieren, können wir die Adressen einzelner Bezeichner in unserem Programm bestimmen. Das Programm 1.4 verwendet ein Makro namens SHW_ADR( ), um die Adressen anzuzeigen.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#define SHW_ADR(ID,I) printf(„The id %s  is at
adr:%8X\n“,ID,&I)
extern _etext, _edata, _end;
/* Prog 1.1 modified to display identifier addresses */
char *cptr = „Hello World\n“; /* static by placement */
char buffer1[25];
void
main(void){
 void showit(char *);
 int i = 0; /* automatic variable */
       /* display segment adr */
 printf(„Adr _etext: %8X  Adr _edata: %8X  Adr _end: %8X
\n\n“,
 &_etext, &_edata, &_end);
```

```
 SHW_ADR(„main“, main); /* display addresses */
 SHW_ADR(„showit“, showit);
 SHW_ADR(„cptr“, cptr);
 SHW_ADR(„buffer1“, buffer1);
 SHW_ADR(„i“, i);
 strcpy(buffer1, „A demonstration\n“); /* library fct */
 write(1, buffer1, strlen(buffer1) + 1); /* system call
*/
 for (; i < 1; ++i)
 showit(cptr); /* function call */
}
void
showit(char *p){
 char *buffer2;
 SHW_ADR(„buffer2“, buffer2);
 if ((buffer2=(char *)malloc((unsigned)(strlen(p)+1)))
!= NULL){
  strcpy(buffer2, p);
  printf(„%s“, buffer2);
  free(buffer2);
 } else {
  printf(„Allocation error.\n“);
  exit(1);
 }
}
```

Die Ausgabe des Programms bestätigt unsere Behauptungen zu den einzelnen Speicherbereichen für verschiedene Variablen. Die tatsächlich gezeigten Adressen variieren natürlich von System zu System und Aufruf zu Aufruf.

```
wegner@elsie(chpt1)$ ./p14
Adr _etext: 1000087F Adr _edata: 2000089C Adr _end:
20000989
The id main is at adr:200008B8
The id showit is at adr:200008C4
The id cptr is at adr:2000089C
The id buffer1 is at adr:20000970
The id i is at adr:2FF21CD0
A demonstration
The id buffer2 is at adr:2FF21C88
Hello World
wegner@elsie(chpt1)$
```

Der Zusammenhang wird in Abbildung 1.8 nochmals bildlich dargestellt.

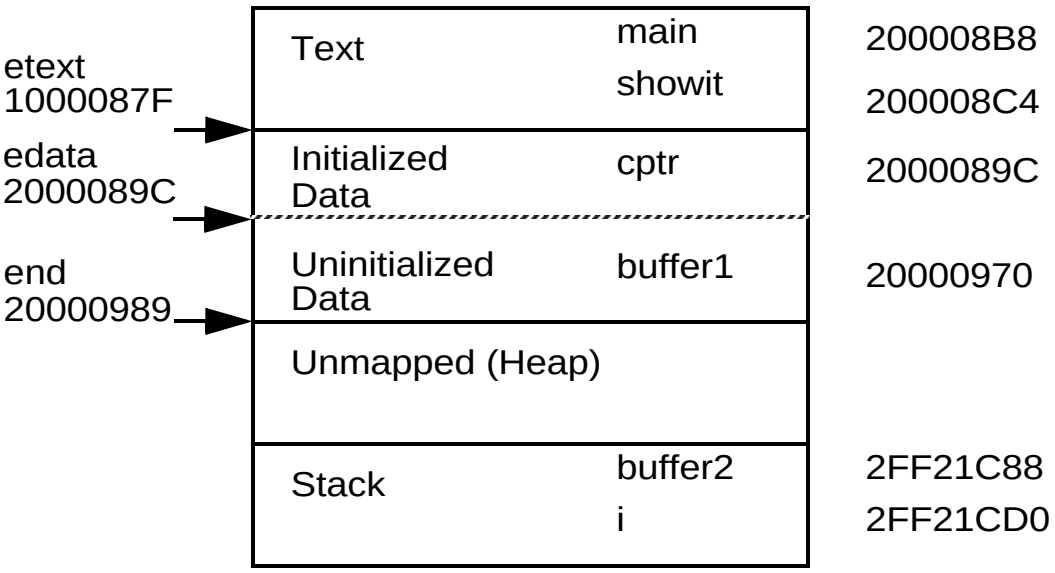


Abb. 1–4 *Prozeßspeicherabbild*

Übung 1–5

Man untersuche die UNIX-Kommandos `limit` und `size`. Wie steht diese Information in Zusammenhang mit `etext`, `edata`, `end`.

Übung 1–6

Alle Programme haben auch einen `break`-Wert (die erste gültige Adresse hinter dem Datensegment des Programms). Dieser wird zunächst auf den Wert von `end` gesetzt. Er müßte sich per Definition bei jeder Speicher-zuteilung `malloc` und Freigabe `free` ändern. Stimmt das unter AIX (unter Linux). Man lese dazu das Handbuch mit dem Hinweis zum Systemaufruf `sbrk` sorgfältig. Ist die Warnung berechtigt?

1.11 Prozeßerzeugung

Offensichtlich muß es einen Mechanismus geben, mit dem sich neue Prozesse erzeugen lassen. Mit Ausnahme einiger Startprozesse, die der Kern beim Hochfahren erzeugt (z.B. `swapper`, `init` und `pagedaemon`), werden in UNIX alle Prozesse mittels des `fork`-Systemaufrufs erzeugt.

Der aufrufende Prozeß ist der Vater (**parent**), der neu erzeugte Prozeß der Sohn (**child**).

Include File(s)	<sys/types.h> <unistd.h>		Manual	**2**
Summary	pid_t fork(void);			
Return	Success	Failure	Sets `errno`	
	0 im Sohn, Sohn PID im Vater-prozeß	-1	Yes	

Tab. 1–2 *fork*

#	Konstante	`perror` Meldung	Erläuterung
11	EAGAIN	Resource temporarily unavailable	Systemgrenze für Anzahl der Prozesse je Benutzer überschritten Systemspeicher für raw I/O nicht ausreichend
12	ENOMEM	Not enough space	Ungenügend swap space verfügbar um weiteren Prozeß zu erzeugen

Tab. 1–3 *fork Fehlermeldungen*

`fork` wird ohne Argumente aufgerufen und liefert im Fehlerfall `-1` zurück. Sonst liefert `fork` dem Aufrufer (Vater) die Prozeßnummer (**PID**) des Sohnprozesses zurück, dem Sohnprozeß eine Null. Durch Überprüfen des Rückgabewerts von `fork` kann ein Prozeß demnach feststellen, ob er Vater oder Sohnprozeß ist. Ein Prozeß kann mehrere Sohnprozesse erzeugen, aber jeder Prozeß kann nur einen Vater haben.

Übung 1–7

Mit `ps` schaue man sich einige der ständig präsenten Kernprozesse an (BSD: `swapper`, `init`, `pagedaemon`; Solaris: `sched`, `init`, `pageout`). Gray weist darauf hin, daß `init` zwar eine entsprechende Datei (meist `/sbin/init`) hat, die anderen aber keine. Warum?

Durch den Aufruf von fork erzeugt das Betriebssystem eine identische Kopie des laufenden Prozesses (einen Klon). An den neu erzeugten Prozeß wird die Prozeßumgebung des Vaters (z.B. Dateideskriptoren, Umgebungsvektor) vererbt mit einigen Ausnahmen:

- der Sohn hat seine eigene Prozeßnummer (PID)
- der Sohn wird eine andere Vaterprozeßnummer haben (parent process ID, PPID)
- die Prozeßbegrenzungen werden auf null zurückgesetzt (z.B. verbrauchte CPU-Zeit)
- Satzsperren auf Dateien werden gelöscht
- Die Maske für Signalreaktionen ist anders.

Ein Programm, das einen `fork`-Aufruf enthält, wird unten gezeigt.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
/* First example of a fork system call (no error check)*/
void
main(void) {
 printf(„Hello\n“);
 fork( );
 printf(„bye\n“);
}
```

Die Ausgabe lautet:

```
wegner@elsie(chpt1)$ ./p15
Hello
bye
bye
wegner@elsie(chpt1)$
```

Gray weist darauf hin, daß `printf` nur einmal im Programm auftaucht. Das zweite `bye` kommt demnach vom zweiten Prozeß.

Übung 1–8

Man erweitere 1.5 um eine Abfrage, so daß die Ausgabe einmal „`bye vom Vater`“, andernfalls „`bye vom Sohn`“ lautet. Welcher kommt zuerst?

Zusammenfassung

- Prozesse und Programme
- vordefinierte Funktionen
- Systemaufrufe, Ausführung durch BS-Kern
- Bibliotheksfunktionen
- Objektcodeformat
- errno, perror, strerror
- Speicherabbild: text, data, stack segment
- u area
- fork Systemaufruf und Vererbung

2 Die Prozeßumgebung

2.1 Einleitung

Alle Prozesse besitzen eine Prozeßumgebung, nicht zu verwechseln mit den Umgebungsvariablen (environment variables), die Teil dieser Umgebung sind. Die Umgebung besteht aus einer eindeutigen Menge von Informationen und Bedingungen, die vom gegenwärtigen Systemzustand und dem Vaterprozeß bestimmt wird. Ein Prozeß kann auf diese Information zugreifen und in einigen Fällen sie auch ändern.

2.2 Prozeßnummer (process ID)

Mit jedem Prozeß ist eine Prozeßnummer, der process ID (**PID**), verbunden, z.B. 0 für den `swapper`, 1 für `init`, den Prozeß, der Wurzel des Vererbungsprozesses ist und auch alle Waisen adoptiert (Prozesse, deren Väter „gestorben“ sind, bevor deren Kinder `exit` machen konnten). Andere PIDs werden frei, meist aufsteigend, vergeben. Die Datei `<sys/param.h>` enthält hierzu einige interessante Parameter.

Include File(s)	<sys/types.h> <unistd.h>		Manual	**2**
Summary	pid_t getpid(void);			
Return	Success	Failure	Sets `errno`	
	Die Prozeß-nummer	-1	Yes	

Tab. 2–1 *getpid*

Der Systemaufruf `getpid` liefert einem Prozeß die eigene Prozeßnummer. In einem Shellskript geht dies im übrigen mit $$ (Wert der Shell-Variablen $).

2.3 Prozeßnummer des Vaterprozesses

Analog geht dies mit `getppid` für den Vaterprozeß.

2.4 Prozeßgruppen

Jeder Prozeß gehört zu einer *Prozeßgruppe*, die durch eine ganzzahlige **Prozeßgruppennummer (process group ID)** bestimmt ist. Wichtig: das ist etwas anderes als der vielleicht bekannte effektive und tatsächliche Gruppenid (Besitzgruppe, **group id**) eines Prozesses.

Wenn ein Prozeß andere Prozesse erzeugt, bildet das BS automatisch eine Prozeßgruppe, deren ursprünglicher Vater der Prozeßführer (**process leader**) ist. Dessen Prozeßnummer PID wird die **Prozeßgruppennummer**. Weitere Abkömmlinge, die der selbe Prozeßtext erzeugt, erben die Gruppennummer.

Wichtigste Aufgabe dieser Gruppennummern ist die Verteilung von Signalen an alle Abkömmlinge eines Prozesses, etwa wenn der Prozeßführer das kill-Signal erhält, das dann auch an alle abhängigen Prozesse geschickt wird.

Ein Prozeß kann seine Prozeßgruppennummer mittels des Systemaufrufs `getpgid` (mit eigener PID als Argument) herausfinden. In manchen Systemen heißt die Funktion `getpgrd`, dann ohne Argument. Ein Aufruf von `getpgid` mit Argument `0` setzt als Argument automatisch die eigene PID ein, man spart sich den Aufruf von `getpid` an dieser Stelle.

Include File(s)	<sys/types.h> <unistd.h>		Manual	**2**
Summary	pid_t getpgid(void pid);			
Return	Success	Failure	Sets `errno`	
	Die Prozeß-gruppennum-mer	-1	Yes	

Tab. 2–2 *getpgid*

#	Konstante	`perror` Meldung	Erläuterung
1	EPERM	Not owner	Ungültige Zugriffsrechte des aufrufenden Prozesses
3	ESRCH	No such process	Falsche Prozeßnummer `pid`

Tab. 2–3 *getpgid Fehlermeldungen*

Das folgende Programm 2.1 erzeugt einige Prozesse und meldet deren process group IDs.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
/* Displaying process group ID information */
void
main( void ){
 int i;
 printf(„\n\nInitial process  PID %6d  PPID %6d  GID
%6d\n\n“,
 getpid(), getppid(), getpgid(0));
 for (i = 0; i < 3; ++i)
 if (fork( ) == 0) /* Generate some processes */
 printf(„New process  PID %6d  PPID %6d  GID %6d\n“,
 getpid(), getppid(), getpgid(0));
}
```

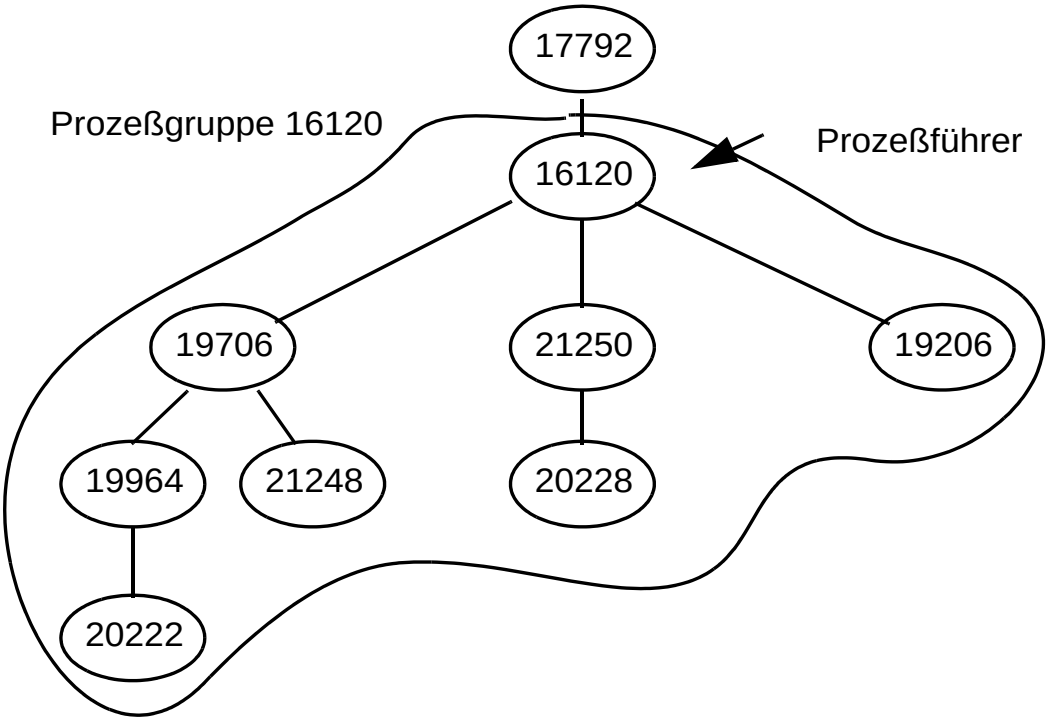
Die Ausgabe (laufzeitabhängig) lautet:

```
wegner@elsie(chpt2)$ ./p21


Initial process PID 16120 PPID 17792 GID 16120

New process PID 19706 PPID 16120 GID 16120
New process PID 19964 PPID 19706 GID 16120
New process PID 20222 PPID 19964 GID 16120
New process PID 21248 PPID 19706 GID 16120
New process PID 21250 PPID 16120 GID 16120
New process PID 20228 PPID 21250 GID 16120
New process PID 19206 PPID 16120 GID 16120
wegner@elsie(chpt2)$ echo $$
17792
```

Der Vererbungsbaum hat das folgende Aussehen.



In Gray sieht die Reihenfolge der Prozeßerzeugungen etwas anders aus, außerdem erzeugt er einen PPID 1 durch Tod eines Vaterprozesses.

Übung 2–1

Steuern Sie mit eingebauten Wartezwängen (sleep) die Reihenfolge und erzeugen Sie dabei mindestens einen Waisen.

Mittels Aufruf von `setpgid` setzt man die Prozeßgruppennummer.

Include File(s)	<sys/types.h> <unistd.h>		Manual	**2**
Summary	pid_t setpgid(pid_t pid, pid_t pgrp);			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Tab. 2–4 *setpgid*

Der Aufruf setzt die Prozeßnummer des Prozesses `pid` auf den neuen Wert `pgrp`. Ist das Argument `pid` 0, bedeutet dies „gegenwärtiger Prozeß“. Ist das Argument `pgrp` 0, wird der Prozeß `pid` Prozeßführer, sofern er dazu berechtigt ist. Die möglichen Fehlermeldungen sind ganz aufschlußreich.

#	Konstante	`perror` Meldung	Erläuterung
1	EPERM	Not owner	- Prozeß pid bereits Prozeßführer - Prozeß pid nicht in selber Sitzung wie aufrufender Prozeß - ungültige Prozeßgruppe genannt
3	ESRCH	No such process	Falsche Prozeßnummer `pid`
13	EACCES	Permission denied	Prozeß pid hat exec ausgeführt
22	EINVAL	Invalid argument	pgrp Wert kleiner 0 oder größer als PID_MAX-1

Tab. 2–5 *getpgid Fehlermeldungen*

Übung 2–2

Man modifiziere das Programm 2.1 so, daß jeder neue Prozeß sein eigener Prozeßführer wird.

2.5 Rechte

Gray geht auf die mit `ls -l` anschaubaren Dateirechte für Besitzer (**owner**), Gruppe (**group**) und andere (**other**) ein, sowie die Rolle des `umask` Wertes für die Vorbesetzung der Dateirechte.

Wir gehen davon aus, daß diese Dinge bekannt sind.

2.6 Wirklicher und eigentlicher Benutzer und Gruppe.

Das Konzept der gespaltenen Benutzerkennung (**real and effective user and group id**) aus UNIX ist nicht ganz trivial. Kurz gesagt, kann der wirkliche Besitzer sich temporär für die Ausführung kritischer Aufgaben die Rechte einer anderen Person leihen, wenn diese andere Person (häufig `root` oder `bin`) für sein Programm das Ausführungsrecht auf `s` (statt `x`) setzt, das sog. set-user-ID (**SUID**) Bit.

Mit dem Kommando `id` lassen sich die gegenwärtigen Benutzer- und Gruppenkennung und Gruppenzugehörigkeiten ausgeben. Für ein Programm kann man mit den Aufrufen `getuid`, `geteuid`, `getgid`, `getegid` wirkliche und effektive Kennungen ermitteln.

2.7 Die Dateiumgebung

Zur Prozeßumgebung gehört auch die Information über geöffnete Dateien. In Unix werden diese durch Indexwerte in Tabellen hinein realisiert, die sog. Dateideskriptoren (**file descriptors**). Eine kleine Tabelle (früher traditionell max. 20, jetzt ca. 64 oder mehr Einträge) je Prozeß befindet sich in der `u area`. Diese Tabelle enthält den Index in die systemweite Dateitabelle im Systembereich. Diese wiederum indiziert die sog. physische **i-node** Tabelle, die einmal je physischem Datenträger (**volume**) existiert (früher 64K Einträge).

Die kleine Tabelle aus der `u area` vererbt sich vom Vater mit jeder Prozeßerzeugung. Insbesondere erben die Söhne die Vorbesetzungen für `stdin`, `stdout`, `stderr`, z.B. Kommandos die von der Shell analysierte und eingestellte Umlenkung mit „<„ und „>“ bzw. „>>“.

Ferner können die Filedeskriptoren der Prozesse auf den selben Eintrag der Systemdateitabelle zeigen, wenn sie gemeinsam auf eine Datei zugreifen. Ist die Datei als gemeinsam benutzbar markiert (**shareable**), hat jeder Prozeß seinen eigenen Schreib-/Lesezeiger (file offset pointer).

Übung 2–3

Man schreibe ein Programm, das eine Textdatei anlegt und dann einen Sohnprozeß erzeugt. Der Sohn soll aus der Datei lesen und anzeigen, was er liest. Nach Beendigung des Sohns soll der Vater weiterlesen und die gelesenen Daten ausgeben. Man zeige damit die gemeinsame Nutzung des Dateizeigers. Damit die Steuerung von Sohn- und Vaterprozeß funktioniert, wird man wohl den `sleep` Systemaufruf brauchen.

2.8 Dateiinformation

Mit den `stat` Systemaufrufen kann man sich Informationen über Dateien (analog zu den von `ls -l` gelieferten) besorgen.

Include File(s)	<sys/types.h> <sys/stat.h>		Manual	**2**
Summary	int stat (const char *path, struct stat *buf); int lstat (const char *path, struct stat *buf); int fstat (int fildes, struct stat *buf)			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Tab. 2–6 *stat, lstat, fstat*

Das erste Argument ist der Pfadname, bzw. im dritten Fall der Dateideskriptor. Der Aufruf von `lstat` ist für symbolische Verweise auf Dateien (**symbolic links**). Alle drei Aufrufe liefern das Resultat mit einem Zeiger auf eine Struktur `stat` ab, die in <`sys/stat.h`> definiert ist.

```
struct stat
{
 dev_t st_dev; /* ID of device containing a directory
entry
 * for this file. File serial no + device
 * ID uniquely identify the file within the
 * system
 */
 ino_t st_ino; /* File serial number */
#ifdef _NONSTD_TYPES
 ushort st_mode_ext;
 ushort st_mode;
#else
 mode_t st_mode; /* File mode; see #define's below */
#endif
 nlink_t st_nlink; /* Number of links */
 ushort_t st_flag; /* Flag word */
#ifdef _NONSTD_TYPES
 ushort st_uid_ext;
 ushort st_uid;
 ushort st_gid_ext;
 ushort st_gid;
#else
 uid_t st_uid; /* User ID of the file's owner */
 gid_t st_gid; /* Group ID of the file's group */
#endif

 dev_t st_rdev; /* ID of device. This entry is defined
only for
 * character or block special files
 */
 off_t st_size; /* 32 bit file size */
 time_t st_atime; /* Time of last access */
 int st_spare1;
 time_t st_mtime; /* Time of last data modification */
 int st_spare2;
 time_t st_ctime; /* Time of last file status change */
 int st_spare3;
 long st_blksize; /* Optimal blocksize for file system
```

```
i/o ops */
 blkcnt_t st_blocks; /* Actual number of blocks
allocated
 * in DEV_BSIZE blocks
 */
 int st_vfstype; /* Type of fs (see vnode.h) */
 ulong_t st_vfs; /* Vfs number */
 ulong_t st_type; /* Vnode type */
 ulong_t st_gen; /* Inode generation number */

 /* end of information returned by non-large file enabled
stat */

 ulong_t st_reserved[9];
#ifdef _KERNEL
 ulong_t st_padto_ll;
 offset_t st_llsize; /* 64 bit file size in bytes */
#endif
};
```

Das folgende Programm 2.2 verwendet einen stat-Aufruf.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#define N_BITS 3
/* Using the stat system call */
void
main(int argc, char *argv[ ]){
 unsigned int i, mask = 0700;
 struct stat buff;
 static char *perm[] = {„---“, „--x“, „-w-“, „-wx“,
 „r--“, „r-x“, „rw-“, „rwx“};
 if (argc > 1) {
  if ((stat(argv[1], &buff) != -1)) {
     printf(„Permissions for %s „, argv[1]);
     for (i = 3; i; --i) {
        printf(„%3s“, perm[(buff.st_mode &
           mask) >> (i-1) * N_BITS]);
        mask >>= N_BITS;
     }
     putchar('\n');
  } else {
     perror(argv[1]);
```

```
      exit(1);
    }
   } else {
   fprintf(stderr, „Usage: %s file_name\n“, argv[0]);
   }
  }
```

Wenn angewandt auf sich selbst (Programmname als Kommandozeilenargument), erscheint die folgende Ausgabe.

```
wegner@elsie(chpt2)$ ./p22 p22
Permissions for p22 rwxr-xr-x
wegner@elsie(chpt2)$
```

Übung 2–4

Man modifiziere Programm 2.2 dahingehend, daß auch der Besitzer ausgegeben wird. Den Namen des Besitzers kann man über `getpwuid` ermitteln, nachdem man den UID hat.

Auf das Ändern der Rechte und der Besitzverhältnisse mittels des Systemaufrufe `chmod` und `fchmod` sei hier nicht weiter eingegangen. Ebenso überspringen wir die Funktionen `umask`, `getcwd` (**get current working directory**), `chdir` und `fchdir`.

2.9 Prozeßgrenzen

Prozesse können Betriebsmittelressourcen nur in begrenztem Umfang in Anspruch nehmen. Mit dem `limit` Kommando in der C-Shell, ggf. mit Option `-h`, können die (*weichen* = prozeßabhängigen und *harten* = systemabhängigen) Grenzen angezeigt werden. Die harten können nur vom Superuser *erhöht* werden.

```
wegner@elsie(chpt2)$ csh
% limit
cputime unlimited
filesize 1048575 kbytes
datasize 131072 kbytes
stacksize 32768 kbytes
coredumpsize 1024 kbytes
memoryuse 32768 kbytes
% limit -h
```

```
cputime unlimited
filesize 1048575 kbytes
datasize unlimited
stacksize unlimited
coredumpsize unlimited
memoryuse unlimited
%
```

Auf weitere Systemaufrufe, wie `ulimit`, `getrlimit`, `setrlimit`, wird hier nicht eingegangen. Die `rlimit` Struktur findet sich in `<sys/resource.h>`. Programm 2.3 (p23 in chpt2) nutzt `getrlimit` und produziert (nicht ganz korrekt für AIX) die folgende Ausgabe.

```
wegner@elsie(chpt2)$ ./p23

CPU time Current: 2147483647 Max: 2147483647
File size Current: 1073741312 Max: 1073741312
Data segment Current:  134217728 Max: 2147483647
Stack segment Current:   33554432 Max: 2147483647
Core size Current:    1048576 Max: 2147483647
Resident set size Current:   33554432 Max: 2147483647
File descriptors Current: 2147483647 Max: 2147483647
       Current: 2147483647 Max: 2147483647
wegner@elsie(chpt2)$
```

Weitere Information liefert `sysconf`, meist aus `<unistd.h>`. Programm 2.4 (in der für AIX verbesserten Form p24b) liefert damit einige ganz nette Informationen.

```
Max size of argv + envp 24576
Max # of child processes    40
Ticks / second   100
Max # of groups    32
Max # of open files  2000
Max # of streams  2000
Timezone name?   255
Job control supported?     1
Saved IDs supported?     1
Version of POSIX supported 199009
```

2.10 Signale

Der Eintritt eines Ereignisses kann einem Prozeß durch ein Signal mitgeteilt werden. Es ist die Softwareform der Hardwareunterbrechung. Verursacher kann sein:

- die Hardware - z.B. Adreßverletzung, Division durch Null
- der BS-Kern - E/A-Wartezwang aufgehoben
- andere Prozesse - Sohnprozeß teilt Vater mit, daß seine Abarbeitung beendigt ist
- Anwender - Tastaturunterbrechung für quit, interrupt oder stop.

Signale sind durchnummeriert und in `<signal.h>` definiert, Erweiterungen dazu meist in `<sys/signal.h>`.

Prozesse können auf drei Arten auf Signale reagieren.

- Die systemseitige Voreinstellung befolgen, d.h. dem Vaterprozeß den Abbruch melden, eine `core` Datei mit dem Speicherauszug erzeugen und abbrechen.
- Das Signal ignorieren; nicht möglich für SIGSTOP (signal 23) und SIGKILL (9).
- Das Signal abfangen und selbst darauf mit eigener Unterbrechungsroutine reagieren (**catch the signal**); nicht möglich für SIGSTOP und SIGKILL

Sohnprozesse erben die Signalvorbesetzung von ihren Vätern, setzen sie aber auf den Standardwert, wenn sie sich mit `exec` ein neues Prozeßbild geben. Der Grund ist klar: im neuen Programm wären die alten Signalbehandlungsadressen ungültig!

Bei Gray folgen dann (S.43) einige Anmerkungen zum Problem von Signalen während der Ausführung von Systemaufrufen (die eigentlich ununterbrechbar gehalten sein sollten). Offensichtlich ist es zu aufwendig, bei jedem Systemaufruf (`read`, `write`, `link`, `chmod`, ...) zu prüfen, ob ein Signal eingetroffen ist, welches es ist und wie aufrufspezifisch darauf zu reagieren ist. Deshalb behauptet Gray, daß in der Regel ein zurückge-

liefertes Resultat -1 des Systemaufrufs den Aufruferprozeß dazu veranlaßt, aufzuräumen (Dateien schließen) und danach auszusteigen, statt einen erneuten Systemaufruf zu versuchen.

Übung 2–5

Man schaue sich die für `signal` im Handbuch angegebenen Standardreaktionen (core dump, exit, ignore) an.

2.11 Kommandozeilenparameter

In der UNIX-Vorlesung haben wir ausführlich über die Parameterweitergabe aus der Kommandozeile an den aufgerufenen Prozeß gesprochen. Aus programmtechnischer Sicht sind die durch **white spaces** getrennten Argumente per Konvention in einem Zeichenkettenvektor `argv` gespeichert. Die Anzahl der Argumente wird üblicherweise durch die Integervariable `argc` verfügbar gemacht.

Das Programm 2.5 listet seine Argumente aus dem Aufruf untereinander auf und nutzt die Vereinbarung aus, daß das letzte Vektorelement `argv[argc]` ein Nullpointer ist.

```
#include <stdio.h>
void
main(int argc, char *argv[ ]){
 for ( ; *argv; ++argv )
 printf(„%s\n“, *argv);
}
```

Die Ausgabe bei einem entsprechenden Aufruf lautet:

```
wegner@elsie(chpt2)$ ./p25 null und nichtig
./p25
null
und
nichtig
wegner@elsie(chpt2)$
```

Insbesondere enthält `argv[0]` den Kommandonamen. Gray schildert dann den Einsatz der Bibliotheksfunktion `getopt` zum Parsen der Kom-

mandozeilenoptionen. Wir übergehen hier die sehr aufwendigen Konventionen, interessierte Studenten mögen sich Programm 2.6 anschauen.

2.12 Umgebungsvariablen

Neben den Kommandozeilenargumenten erbt ein Sohnprozeß Umgebungsvariablen, die üblicherweise über einen externen Zeiger

```
extern char **environ;
```

zugänglich gemacht werden. Programm 2.7 zeigt die Werte an.

```
#include <stdio.h>
extern char **environ;
void
main( void ){
 for ( ; *environ; ++environ)
 printf(„%s\n“, *environ);
}
```

Der Anfang der Ausgabe lautet:

```
no_proxy=www.db.informatik.uni-kassel.de
TERMINAL_EMULATOR=dtterm
LOCPATH=/usr/lib/nls/loc
ignoreeof=10
MAILMSG=[you have new mail]
HISTSIZE=100
LOGNAME=wegner
DTSCREENSAVERLIST=StartDtscreenSwarm StartDtscreenQix
StartDtscreenFlame StartDtscreenHop StartDtscreenImage
StartDtscreenLife StartDtscreenRotor StartDtscreenPyro
StartDtscreenWorm StartDtscreenBlank
HISTFILESIZE=100
LOGIN=wegner
...
```

Einen einzelnen Wert kann man sich mit `gentenv` zurückgeben lassen, das Setzen mit `putenv` ist „tricky“ und wird hier übergangen.

Include File(s)	<stdlib.h>		Manual	**3c**
Summary	char *getenv(char *name);			
Return	Success	Failure	Sets `errno`	
	Zeiger auf Wert in der Umgebung	NULL-Zeiger		

Tab. 2–7 *getenv*

Programm 2.8 verwendet den Aufruf für die Anzeige der TERM-Variable. Wichtig ist: im Aufruf muß Argument *name* das Gleichheitszeichen enthalten („TERM=“), das Resultat enthält es nicht.

```
#include <stdio.h>
#include <stdlib.h>
void
main( void ){
 char *c_ptr;
 c_ptr = (char *) getenv(„TERM=“);
 printf(„The variable TERM is %s\n“, c_ptr==NULL ? „NOT
found“:c_ptr);
}
```

Die entsprechende Ausgabe lautet:

```
wegner@elsie(chpt2)$ echo $TERM
dtterm
wegner@elsie(chpt2)$ ./p28
The variable TERM is dtterm
wegner@elsie(chpt2)$
```

Zusammenfassung

Wir haben versucht, die Umgebung, in der ein Prozeß arbeitet, darzustellen. Dazu gehören die Kennungen PID und PPID, die Prozeßgruppenkennung, die Ressourcen (Dateien) und allgemeinen Beschränkungen, die Möglichkeiten auf Signale zu reagieren und die Parameter, die vom Vaterprozeß vererbt werden. Alle können von laufenden Prozessen über

Systemaufrufe oder Bibliotheksfunktionen abgefragt, manche auch neu gesetzt werden.

3 Mehr zu Prozessen

3.1 Einleitung

Das Prozeßkonzept in UNIX ist die Grundlage aller Betrachtungen zur Kommunikation in Rechnernetzen. In diesem Kapitel betrachten wir die Systemaufrufe `fork` und `exec` nochmals im Detail.

3.2 Der `fork` Systemaufruf

Der fork-Aufruf ist etwas besonderes dadurch, daß ein Aufruf zwei getrennte Rückkehrvorgänge auslöst. Sowohl der Vater- wie auch der Sohnprozeß machen mit der Anweisung, die dem fork-Aufruf folgt, weiter, wobei bekanntlich der Sohn 0 und der Vater den PID des Sohns zurückgeliefert bekommt.

Im folgenden Programm wird dies auf einfachste Art demonstriert.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
/*
 Generating a child process (no error check)
*/
void
main( void ){
 if (fork( ) == 0)
 printf(„In the CHILD process\n“);
 else
 printf(„In the PARENT process\n“);
}
```

Welche der beiden Ausgaben `In the CHILD process`, bzw. `In the PARENT process` zuerst kommt, hängt vom scheduler („Prozeß-zuteiler“) ab. Im nächsten Programm wird dies bereits bewußt beeinflußt, indem mit `sleep` ein Prozeß eine Sekunde in den Wartezustand versetzt wird.

Damit die Ausgabe auch richtig versetzt ausgegeben wird, arbeitet das Programm mit dem `write`-Systemaufruf und nicht mit dem gepufferten `printf`.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
/*
 Multiple PARENT - CHILD processes (no error check)
*/
void
main(void) {
 int i;
 static char buffer[10];
 if (fork( ) == 0) { /* In the child process */
  strcpy(buffer, „CHILD\n“);
 } else { /* In the parent process */
  strcpy(buffer, „PARENT\n“);
 }
 for (i = 0; i < 5; ++i) { /* Both processes do this */
  sleep(1); /* 5 times each. */
  write(1, buffer, sizeof(buffer));
 }
}
```

Die Ausgabe lautet:

```
wegner@elsie(chpt3)$ ./p32
CHILD
PARENT
CHILD
PARENT
CHILD
PARENT
CHILD
PARENT
CHILD
```

```
PARENT
wegner@elsie(chpt3)$
```

Übung 3–1

Für das untenstehende Programm (liegt als `ex31.c` vor) überlege man sich vor dem Ausprobieren, wie die Ausgabe aussieht. Gray fragt, was sich ändert, wenn man die Newlines in den `printf`-Aufrufen wegläßt.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
/*
 A funny C program ...
*/
void
main( void ) {
 fork( ); printf(„hee\n“);
 fork( ); printf(„ha\n“);
 fork( ); printf(„ho\n“);
}
```

3.3 Überlagern mit `exec`

Neue Prozesse werden aus einer Reihe von Gründen erzeugt. Im Hintergrund laufende `daemon` Prozesse, wie z.B. `lpd`, `inetd` (Internet services daemon), `routed` (network routing daemon) produzieren Unterprozesse zur Erledigung von Hausaufgaben. Meist wird dabei nicht ein identischer Unterprozeß erzeugt, sondern einer, der zwar die Umgebung (**u area**) erbt, sich aber mit einem neuen Prozeßabbild (text, data, stack) durch Aufruf von `exec` (in einer der 6 Geschmacksrichtungen) überlagert.

Weil jetzt das ursprüngliche Textsegment weg ist, kann `exec` auch nichts abliefern, d.h. es ist ein Aufruf ohne Rückkehr.

Die Wirkungsweise von `fork` + `exec` kann man sich an einem Kommandoaufruf von Shellebene aus klarmachen (vgl. UNIX-Vorlesung):

- Aufruf z.B. `$cat file.txt >file2.txt`

- Die Shell analysiert die Zeile und bildet Tokens, z.B. `cat`, `file.txt`, usw.
- Die Shell erzeugt einen Sohnprozeß mittels `fork`
- Die Shell (als Sohn) schließt die Standardausgabe und öffnet die Datei `file2.txt`.
- Die Shell (als Sohn) überlagert sich durch Aufruf von `execve` mit dem Programmcode von `cat`.
- Die Shell (der Vater) wartet (Systemaufruf `wait`) auf die Beendigung des Sohnprozeßes `cat` und gibt danach das Promptzeichen für die nächste Eingabe aus.
- Prozeß `cat` macht `exit` oder `return`. Er gibt dabei einen Integer - den **Status** - an den Vater zurück, der ihn als Parameter in `wait` empfängt (kann man mit `echo $status` in der C-Shell ausgeben).

Wir sollten auch erwähnen, daß man auch auf Shellebene ein `exec`-Kommando absetzen kann, das dann die aufrufende Shell überlagert (und damit ggf. ein **logout** nach sich zieht).

Die Prototypen für die `exec` Systemaufrufe (aus der AIX `man` Seite lauten:

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

int execl (Path, Argument0 [, Argument1, ...], 0)
const char *Path, *Argument0, *Argument1, ...;

int execle (Path, Argument0 [, Argument1, ...], 0,
   EnvironmentPointer)
const char *Path, *Argument0, *Argument1, ...;
char *const EnvironmentPointer[ ];

int execlp (File, Argument0 [, Argument1, ...], 0)
const char *File, *Argument0, *Argument1, ...;

int execv (Path, ArgumentV)
const char *Path;
char *const ArgumentV[ ];
```

```
int execve (Path, ArgumentV,
   EnvironmentPointer)
const char *Path;
char *const ArgumentV[ ], *EnvironmentPointer[ ];

int execvp (File, ArgumentV)
char *const ArgumentV[ ];

int exect (Path, ArgumentV, EnvironmentPointer)
char *Path, *ArgumentV, *EnvironmentPointer [ ];
```

Die Argumentstruktur spiegelt sich im Namen wieder: Nach den Buchstaben `exec` folgt als erster Buchstabe der Hinweis, ob die Argumente in Listenform (Buchstabe `l`) oder als Pointer auf einen Vektor (analog zu `argv` oben) übergeben werden. Danach folgt ggf. e oder p mit der Bedeutung, daß bei `e` der Aufrufer die Environmentliste selbst übergibt (Gray: nicht zu gebrauchen), bei `p`, daß die gegenwärtige Besetzung von PATH bei der Suche nach dem auszuführenden Programm zu verwenden ist, sonst muß der ganze Pfadname genannt werden.

Die folgende Tabelle faßt die 6 von Gray genannten Varianten zusammen, wobei `execlp` und `execvp` die geläufigsten sind.

Library Call Name	Argumente Format	Environment Variable übergeben?	Automatische Suche in PATH?
execl	list	ja	nein
execv	array	ja	nein
execle	list	nein	nein
execve	array	nein	nein
execlp	list	ja	ja
execvp	array	ja	ja

Tab. 3–1 *exec system calls*

Übung 3–2

Interpretieren Sie `exect` aus der AIX `man` Seite.

Die Aufrufform für `execlp` lautet:

Include File(s)	<unistd.h>		Manual	**2**
Summary	int execlp (const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/);			
Return	Success	Failure	Sets `errno`	
	keine Rück-kehr	-1	Yes	

Tab. 3–2 *execlp*

Das erste Argument, `file`, ist ein Zeiger auf die Zeichenkette mit dem Namen der Programmcodedatei. Es gelten die allgemeinen Regeln über absolute und relative Pfadnamen. Auch bei einem absoluten Pfadnamen ist die Erweiterung `p` relevant, falls eines der Argumente ein relativer Pfadname wäre.

Damit der Aufruf klappt, muß die Datei gefunden werden und sie muß als ausführbar markiert sein. Im Fall eines Shellskripts wird die Bourne-Shell aufgerufen und führt das Skript aus.

Die Liste der möglichen Fehler ist groß und wird separat ausgeteilt.

Die Argumente 0 bis *n* sind Zeiger auf auf die Argumente, wobei nach Konvention `arg0` der Programmname (ohne Pfadanteile), `arg1` dann das erste Argument in `argv`-Notation, demnach `argv[1]`), `arg2` das 2. Argument, usw. bis zum letzten, das aus Portabilitätsgründen mit einem NULL-Zeiger besetzt ist.

Programm 3.3 verwendet `execlp` für ein `cat`-Ausgabeprogramm. Das Programm und seine Ausgabe sehen wie folgt aus.

```
$ ./p33 p33.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int
main(int argc, char *argv[ ]){
 if (argc > 1) {
   execlp(„/bin/cat“, „cat“, argv[1], (char *) NULL);
   perror(„exec failure „);
   exit(1);
 }
 fprintf(stderr, „Usage: %s text_file\n“, *argv);
 exit(1);
}$
```

Man beachte, daß `perror` nur erreicht wird, wenn die Überlagerung fehlschlägt.

Übung 3–3

Was passiert, wenn man `execlp` mit der leeren Zeichenkette für `arg0` aufruft oder falls sofort nach `file` der NULL-Pointer folgt. Hat dann `argc` den Wert 0? Man ändere p33.c ab, um die Vermutungen zu testen!

Es folgt eine kurze Besprechung der `execvp`-Variante. Diese wird mit zwei Argumenten aufgerufen, dem Dateinamen für den Code und einen Zeiger auf einen Zeichenkettenvektor mit einem Format wie für `argv` (und meist tatsächlich auch `argv`). Der Abschluß dieses Vektors sollte als letztes Element wieder ein NULL-Pointer bilden.

Include File(s)	<unistd.h>		Manual	**2**
Summary	`int execvp (const char *file, const *char argv[] );`			
Return	Success	Failure	Sets `errno`	
	keine Rück-kehr	-1	Yes	

Tab. 3–3 *execvp*

Offensichtlich eignet sich diese Aufrufform für variabel lange Argumentlisten.

Programm 3.4 zeigt eine Anwendung, in der das 1. Argument des Aufrufs auf Shellebene das eigentlich auszuführende Kommando ist (vgl. UNIX-Vorlesung `RUN $*`).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int
main(int argc, char *argv[ ]) {
 execvp(argv[1], &argv[1]);
 perror(„exec failure“);
 exit(1);
}
```

Die Ausführung mit der vorhandenen Textdatei `test.txt` bzw. mit `echo` ergibt:

```
wegner@elsie(chpt3)$ ./p34 cat test.txt
This is a sample text
file for a program to
display!
wegner@elsie(chpt3)$ ./p34 echo Hallo Leute!
Hallo Leute!
wegner@elsie(chpt3)$ ./p34 cat -n test.txt
 1 This is a sample text
 2 file for a program to
 3 display!
```

Falls wir uns neue Argumente schnitzen wollen, müssen wir das in einem eigenen `argv`-Vektor tun, wie unten gezeigt für das Programm 3.5, das die selbe Ausgabe wie 3.4 liefert.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int
main( void ){
 static char *new_argv[ ] = {„cat“,
         „test.txt“,
         (char *) 0
         };
```

```
 execvp(„/bin/cat“, new_argv );
 perror(„exec failure „);
 exit(1);
}
```

3.4 Der Gebrauch von `fork` und exec zusammen

Meist wird ja `fork` zusammen mit `exec` aufgerufen; in einigen BS-Varianten gibt es die Kombination beider Aufrufe als `spawn` Systemaufruf (Gray).

Das Programm 3.6 demonstriert den Gebrauch.

```
#include <stdio.h>
#include <unistd.h>
void
main(void){
 static char *mesg[ ] = {„Fie“, „Foh“, „Fum“};
 int display_msg(char *), i;
 for (i = 0; i < 3; ++i)
  (void) display_msg(mesg[i]);
 sleep(2);
}
int
display_msg(char *m) {
 char err_msg[25];
 switch (fork( )) {
 case 0:
  execlp(„/usr/bin/echo“, „echo“, m, (char *) NULL);
  sprintf(err_msg, „%s Exec failure“, m);
  perror(err_msg);
  return (1);
 case -1:
  perror(„Fork failure“);
  return (2);
 default:
  return (0);
 }
}

wegner@elsie(chpt3)$ ./p36
Fie
Foh
Fum
```

Hinweis: Gray erhält immer sehr interessante Ausführungsfolgen, z.B. im vorliegenden Fall `Foh, Fie, Fum`. Ob er den scheduler besticht, damit er ihm so „zufällige“ Prozeßfolgen liefert, können wir nur ahnen.

Sehr interessant wird es, wenn wir dafür sorgen, daß der Aufruf von `execlp` schiefgeht, etwa durch

```
execlp(“usr/bin/no_echo“, “echo“, m, (char *) NULL );
```

Dabei schreibt `sprintf` in einen angegebenen Puffer, den man dann `perror` übergibt. Stevens warnt übrigens davor, daß `sprintf` gnadenlos über den angegebenen Puffer hinausschreibt, wenn das zweite Argument zu lang ist.

```
wegner@elsie(chpt3)$ ./p36falsch
Fie Exec failure: No such file or directory
Foh Exec failure: No such file or directory
Foh Exec failure: No such file or directory
Fum Exec failure: No such file or directory
Fum Exec failure: No such file or directory
Fum Exec failure: No such file or directory
Fum Exec failure: No such file or directory
```

Damit beenden wir den fork + exec Abschnitt.

Übung 3–4

Betrachten Sie das längere Programm 3.7, die sog. **huh?** Shell, und kommentieren Sie seinen Zweck und Aufbau. Dabei liest `gets` eine Eingabezeile (inkl. Leerzeichen) bis zum ersten **Newline**, `strtok` ist eine Parsing-Funktion die den zuerst übergebenen Zeichenstring erneut ab der zuletzt gelesenen Stelle weiteranalysiert, sofern es beim 2. und allen weiteren Aufrufen das Argument NULL erhält.

Übung 3–5

Ruft man in der **huh?**-Shell z.B. `df -t` auf, wird die Option richtig erkannt. Eine Umlenkung, z.B.
`>/tmp/ps_out` aber nicht. Warum?

3.5 Beendigung eines Prozesses

Alles kommt einmal zum Ende, auch Prozesse. Es gibt drei Arten für einen Prozeß freiwillig aus dem Leben zu scheiden (ohne signal und ohne Systemcrash):

- Systemaufruf von `exit` oder `_exit` an beliebiger Stelle im Code.
- Ausführung einer `return`-Anweisung in der Funktion **main**.
- Programmabarbeitung erreicht das Ende der **main**-Funktion

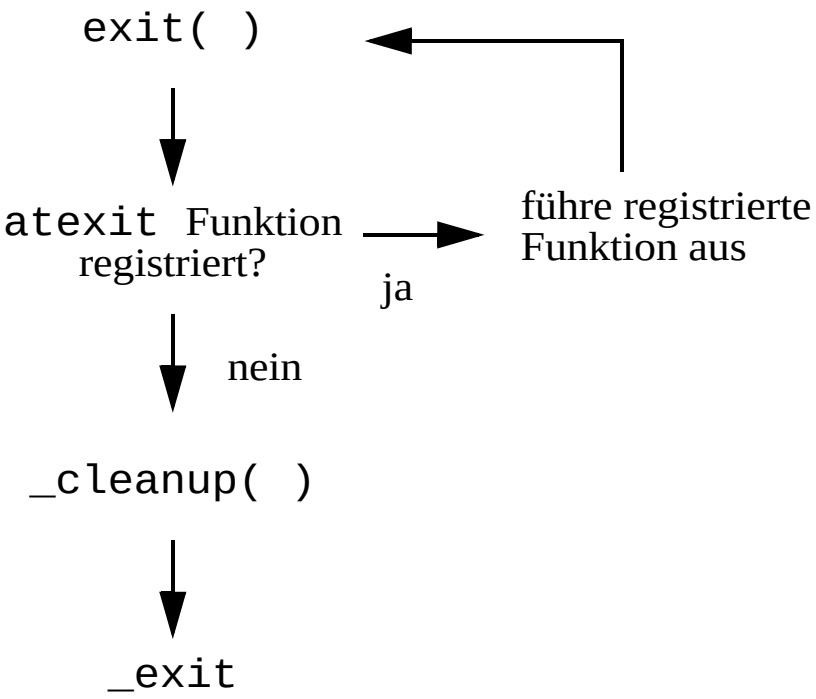
Die vorzugsweise Beendigung ist mittels exit, das nichts zurückliefert und als einzigen Paramenter einen Statuswert an den Vaterprozeß abliefert, per Konvention 0 bei Erfolg und ≠0 bei Fehler[1].

Include File(s)	<stdlib.h>		Manual	**2**
Summary	void exit (int status);			
Return	Success	Failure	Sets `errno`	
	keine Rück-kehr	keine Rück-kehr		

Tab. 3–4 *exit Bibliotheksfunktion*

Der Ablauf des Ausstiegs läßt sich an der folgenden Graphik ablesen.

1. Gray: es werden nur die unteren 8 bits zurückgeliefert im Bereich 0..255. Was macht exit(-1)?



Zunächst werden in umgekehrter Registrierungsreihenfolge alle die benutzerdefinierten Funktionen aufgerufen, die vorher bei `atexit` angemeldet wurden (es werden deren Startadressen hinterlegt, Parameter sind nicht zugelassen).

Sind diese alle abgearbeitet, kommt die systemeigene Aufräumroutine `_cleanup( )`zum Zug, danach die „finale“ Ausstiegsroutine `_exit()`, die den Status als Argument erhält und ihn an den Vater zurückgibt. Man kann auch `_exit( )` direkt aufrufen und umgeht dann die o.g. Aufräumarbeiten.

Einige Aufräumarbeiten macht aber auch `_exit( )` als Teil der folgenden Schritte (Details siehe `man exit`).

- Alle offenen Dateien schließen (Puffer rausschreiben, fid freigeben).
- Vaterprozeß benachrichtigen (SIGCHLD signal). Hat Vater noch kein `wait` gemacht, Signal aufheben, Sohn wird Zombie, d.h. Eintrag in Prozeßtabelle bleibt, aber Code und Daten abgeräumt.
- Status an Vater zurückliefern.

- Falls noch Kinderprozesse da, deren PPID auf 1 setzen (`init` adoptiert alle Waisen).
- war der Prozeß ein Prozeßführer, erhalten diese SIGHUP/ SIGCONT Signale
- shared memory und Semaphore werden adjustiert
- Abrechnungssatz (accounting record) wird geschrieben

Übung 3–6

Man experimentiere mit Programm **p38.c**, das den Gebrauch von `atexit` erklärt. Insbesondere untersuche man den eigenständigen Aufruf von Aufräumroutinen, verglichen mit dem `atexit`-gesteuerten.

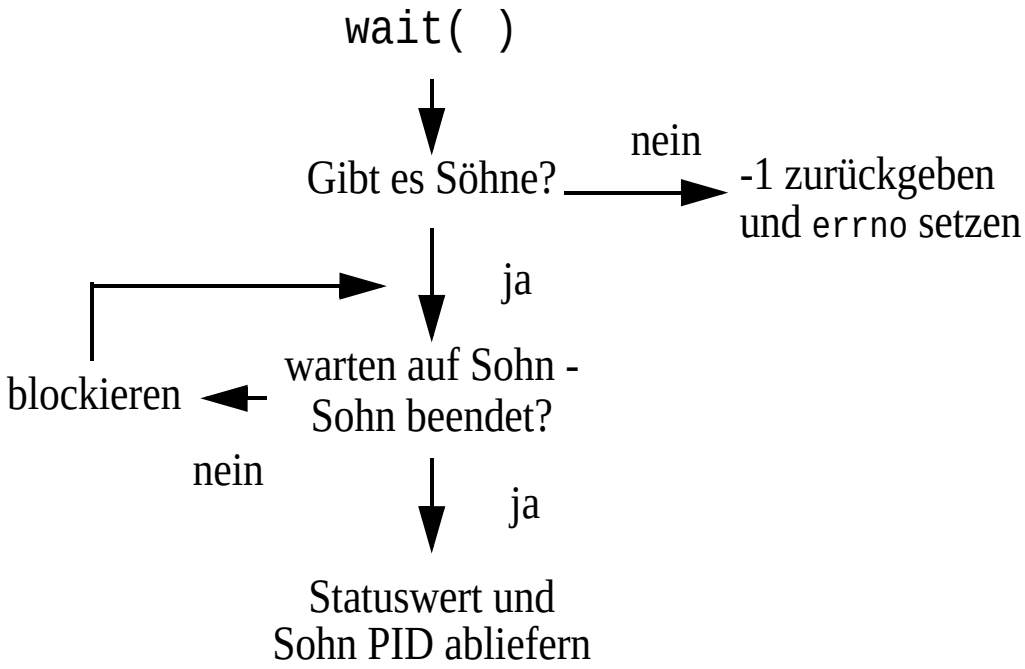
3.6 Warten auf Prozesse

Für die Organisation von Arbeitsabläufen ist die Erzeugung eines oder mehrerer Unterprozesse, an die Aufgaben delegiert werden, sowie das Warten auf deren Beendigung, eine natürliche Angelegenheit, vergleichbar der Strukturierung von Programmen in Unterprogramme.

Include File(s)	<sys/types.h> <sys/wait.h>		Manual	**2**
Summary	pid_t wait(int *stat_loc);			
Return	Success	Failure	Sets `errno`	
	Sohn PID	-1	Yes	

Tab. 3–5 *wait Systemaufruf*

Die Aktivitäten als Folge eines `wait( )` Systemaufrufs zeigt die nächste Abbildung.



Der Aufruf von `wait` erwartet als Argument einen Zeiger auf einen Integer. Die unteren zwei Bytes dieses Integers enthalten nach Rückkehr Statusinformationen (siehe unten).

Der zurückgelieferte Funktionswert ist durch `pid_t` definiert, üblicherweise ein `long int`, für -1 oder der Sohn PID bei Erfolg.

Gibt es bei Aufruf von `wait` keine Sohnprozesse, liefert `wait` sofort einen Fehler (siehe Übung). Gibt es Sohnprozesse, die noch nicht fertig sind, wartet der Vaterprozeß auf die Terminierung des ersten Sohnes, d.h. er stellt seine Aktivitäten ein (Kontextumschaltung).

Übung 3–7

Auf welchen Wert wird `errno` gesetzt, wenn `wait` aufgerufen wird, obwohl keine Sohnprozesse existieren?

Hat ein Sohnprozeß seine Abarbeitung eingestellt, z.B. durch ein `exit( )`, wird der Vater wieder aktiviert und weiß dann aus der Funktionswertrückgabe, welcher Unterprozeß verschieden ist. Falls es „der fal-

sche“ ist, muß er die gelieferte Information (Status aus Argument) zwischenspeichern und weiterwarten.

Die Statusinformation lautet (die alten PDP-10 Zeiten lassen grüßen!):

1. Bei normalem Prozeßende enthält byte 0 den Wert 0 und byte 1 den Exitcode (0-255).

byte 3	byte 2	byte 1	byte 0
		exit code	0

2. Bei Prozeßende aufgrund eines nicht abgefangenen Signals ist das obere Byte 0 und das untere enthält die Signalnummer.

byte 3	byte 2	byte 1	byte 0
		0	signal #

Im zweiten Fall ist bei einem produzierten core dump das oberste Bit in byte 0 auf 1 gesetzt. Damit bleiben theoretisch nur noch 128 Signalnummern übrig; wieviele sind es tatsächlich?

Das folgende Programm `p39` in Zusammenarbeit mit `p310` zeigt die Arbeitsweise von `wait`. Es werden drei Unterprozesse gestartet und danach wird auf ihre Beendigung gewartet. In den Söhnen wird über einen Zufallszahlengenerator (`rand( )`) eine zufällige Zeit gewartet und dann, je nach PID mit Signal 9 oder mit `exit` abgebrochen.

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int
main( void ){
 pid_t pid, w;
 int i, status;
 char value[3]; /* place to store index as string */
 for (i = 0; i < 3; ++i) { /* generate 3 child processes
 */
 if ((pid = fork( )) == 0) {
```

```
 sprintf(value, „%d“, i);
 execl(„child“, „child“, value, (char *) 0);
 } else /* assuming no failures here ... */
 printf(„Forked child %d\n“, pid);
 }
/* Wait for the children
*/
 while ((w = wait(&status)) && w != -1) {
 if (w != -1)
 printf(„Wait on PID: %d returns status of : %04X\n“, w,
status);
 }
 exit(0);
}
```

Programm 3.10, “child“, sieht wie folgt aus.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
int
main(int argc, char *argv[ ]){
 pid_t pid;
 int ret_value;
 pid = getpid( );
 ret_value = (int) (pid % 256);
 srand((unsigned) pid);
 sleep(rand( ) % 5);
 if (atoi(*(argv + 1)) % 2) { /* assuming argv[1] exists!
*/
 printf(„Child %d is terminating with signal 0009\n“,
pid);
 kill(pid, 9); /* commit hara-kiri */
 } else {
 printf(„Child %d is terminating with exit(%04X)\n“,
pid, ret_value);
 exit(ret_value);
 }
}
```

Die Ausgabe für einen Lauf lautet z.B. (siehe auch Übung).

```
wegner@elsie(chpt3)$ ./p39
Forked child 17454
Forked child 22832
Forked child 19250
Child 17454 is terminating with exit(002E)
Wait on PID: 17454 returns status of : 2E00
Child 22832 is terminating with signal 0009
Wait on PID: 22832 returns status of : 0009
Child 19250 is terminating with exit(0032)
Wait on PID: 19250 returns status of : 3200
```

Übung 3–8

In unserem Quellverzeichnis stehen die ausführbaren Programme `p39` und `p310`. Der Aufruf `./p39` liefert aber eine ganz andere Ausgabe. Welche Kleinigkeit wurde vergessen?

Wie bereits erwähnt, geht ein beendeter Prozeß, auf den der Vater noch kein `wait` gemacht hat, in den **Zombie**-Zustand über, wie man aus `ps -alx` ersehen kann. Einen solchen, bereits toten Prozeß, kann man auch nicht nochmal mit `kill -9` abschießen (vielleicht mit einer silbernen Kugel?) um ihn aus der Prozeßtabelle zu vertreiben. Stirbt der Vater, ist `init` für die Beseitigung zuständig und macht deshalb ständig `wait`.

Übung 3–9

Man schreibe ein kleines Programm, das 3 Zombies erzeugt und vergewissere sich mit `ps`, daß dies auch tatsächlich geschieht und daß sie letztendlich dann doch verschwinden.

Für das Entschlüsseln der Statusinformation nach `wait` gibt es einige Makros unter dem `wstat` Eintrag, trotzdem bleibt die ganze Handhabung umständlich.

Speziell auch liefert `wait` des Resultat des ersten zu Ende kommenden Unterprozesses zurück und es blockiert immer, wenn es einen noch laufenden Unterprozess gibt, selbst wenn der gesuchte vielleicht bereits beendet wurde. Deshalb gibt es (noch komplexere) Formen, z.B. `wait-pid`, die gezielter warten können. Wir übergehen die Besprechung hier,

Programm 3.11 entält ein gegenüber 3.10 verändertes Beispielprogramm mit `waitpid`.

Man beachte auch, daß es auf Shellebene ebenfalls ein `wait(pid)` gibt.

Zusammenfassung

Prozesse werden mit `fork` erzeugt. Dadurch entstehen zwei Prozeßbilder mit gleichem Programmcode. Insbesondere erbt der Sohn die Umgebung des Vaters. Meist überlagert sich der Sohn anschließend mit `exec`. Nach Beendigung seiner Aufgaben führt er `return` in **main** aus oder `exit`. Der Vaterprozeß wird üblicherweise auf die Rückkehr mit `wait` warten und erhält Statusinformation.

4 Primitive Kommunikation

4.1 Einleitung

In Gray folgen im 4. Kapitel auf ca. 35 Seiten eine Besprechung einfacher Prozeßkommunikation. Wir fassen dieses Kapitel nur zusammen, da die Techniken (*Lock Files*, *Signals*) im Prinzip unbefriedigend sind.

4.2 Sperrdateien (Lock Files)

Eine Sperrdatei ist eine „gewöhnliche“ Datei, über die sich zwei oder mehr Prozesse synchronisieren können. Sperrdateien sind etwas anderes als Dateisperren (siehe Abschnitt 4.3). Kurz gefaßt gilt:

- zwei oder mehr Prozesse wollen sich synchronisieren, z.B. um den Zugriff auf eine gemeinsam zu nutzende Ressource (Druckspooler, etc.) regeln zu können.
- die beteiligten Prozesse müssen sich auf Namenskonventionen einigen und die Datei in ein Verzeichnis legen, auf das sie Schreibrecht haben (häufig `/tmp`).
- der exklusive Zugriff erfolgt durch erfolgreiches Anlegen der Sperrdatei; dies gelingt nur, wenn die Datei noch nicht existiert, da sie für alle schreibgesperrt ist.
- die Anfrage, ob eine Datei existiert und, wenn nicht, das Anlegen geschieht atomar durch den Systemaufruf `creat`.

- diese Zusicherung des atomaren `creat` (ohne e) ist eine uralte UNIX-Geschichte und war lange Zeit die einzig zuverlässige Synchronisationsmöglichkeit.
- ist die Datei bereits vorhanden, sollte der „abgewiesene“ Prozeß warten und nach einer gewissen Zeit erneut einen Versuch starten.
- Freigeben der Sperrdatei durch `unlink`-Aufruf.

Nachteile

- funktioniert nicht für Synchronisationsaufgaben des *super users*, da dieser immer schreiben darf.
- erzwingt eine Art *busy waiting* (aktives Warten durch Abfackeln von Prozessorzyklen), ggf. mit `sleep` dazwischen, dann keine Garantie, daß nicht anderer Prozeß wiederholt zuerst Zugriff hat.

Gray bespricht die Technik ausführlich mit einem Beispielprogramm 4.1. Uns interessiert momentan nur die `sleep`-Funktion zum temporären Verzögern eines Prozesses.

Include File(s)	<unistd.h>		Manual	**3c**
Summary	unsigned sleep(unsigned seconds);			
Return	Success	Failure	Sets `errno`	
	Rest der nichtverschlafenen Zeit			

Tab. 4–1 *sleep Bibliotheksfunktion*

Wird `sleep` durch ein Signal unterbrochen, liefert es den Restwert aus der beim Aufruf eingestellten Schlafzeit minus der „verschlafenen Zeit“ (in Sekunden) zurück, in der Regel also 0.

4.3 Dateisperren

Der englische Begriff **record locking** ist in UNIX eigentlich fehl am Platz, weil UNIX kein Satzkonzept hat. Trotzdem hat sich ein Konzept für

das bereichsweise, bzw. totale Sperren von Textdateien entwickelt, allerdings unter AIX nicht so auf Shellebene, wie dies Gray beschreibt.

Generell unterscheidet man Pflichtsperren (**mandatory locks**) und Kooperationssperren (**advisory locks**). Im ersteren Fall werden die Zugriffsrechte einer Datei so gesetzt, daß jeder `read` und `write` Systemaufruf prüfen muß, ob er unter den gegenwärtig gesetzten Sperren seinen lesenden oder schreibenden Zugriff durchführen kann. Dies verlangsamt natürlich den Zugriff.

Im zweiten Fall geht man von einer Funktionsbibliothek von Zugriffsfunktionen aus, z.B. einer Datenbank, die Satzsperrinformationen verwenden, um kooperierend den Zugriff auf gemeinsame Datenbestände zu regeln. Ein „disziplinloser“ Prozeß (rogue process), der das Schreibrecht auf diese Datei hat, kann die Kooperationsvereinbarung allerdings ignorieren und den Inhalt korrumpieren.

Sperren werden im wesentlichen mit dem Systemaufruf `fcntl` (**file control**) oder der Bibliotheksroutine `lockf` geregelt.

Das erste Argument ist der Dateideskriptor einer geöffneten Datei, die man mit `fcntl` befragen oder verändern will. Das 2. Argument betrifft die gewünschte Operation, definiert über eine Liste von Konstanten (siehe Liste unten) aus dem `fcntl`-Header.

<table>
<tr><td>Include File(s)</td><td colspan="2"><sys/types.h>
<fcntl.h></td><td>Manual</td><td>2</td></tr>
<tr><td>Summary</td><td colspan="4">int fcntl(int fildes,
int cmd,
/* arg */ ...);</td></tr>
<tr><td rowspan="2">Return</td><td>Success</td><td>Failure</td><td colspan="2">Sets <code>errno</code></td></tr>
<tr><td>Wert abhängig von <code>cmd</code> Argument</td><td>-1</td><td colspan="2">Yes</td></tr>
</table>

Tab. 4–2 *fcntl Systemaufruf*

F_GETLK	Return information about an existing lock.
F_GETLK64	Return information about an existing lock.
F_SETLK	Set or clear a file lock.
F_SETLK64	Set or clear a file lock.
F_SETLKW	Set or clear a file lock and wait if blocked.
F_SETLKW64	Set or clear a file lock and wait if blocked.

Das dritte, optionale Argument enthält bei Aufrufen in Zusammenhang mit Sperren einen Zeiger auf eine `flock`-Struktur aus `sys/flock.h`, die in AIX ungefähr wie folgt aussieht (mit eigenen Kommentaren)

```
struct flock {
 short l_type;/* Sperrtyp */
 short l_whence;/* Sperre ab Position ... */
#ifndef _LARGE_FILES
 off_t l_start;/* relativer Offset */
 off_t l_len; /* len = 0 means until end of file */
#endif
 unsigned long l_sysid;/* distributed process id */
#ifdef _NONSTD_TYPES
 ushort l_pid_ext;
 ushort l_pid;/* process id assoc. with file */
#else
 pid_t l_pid;
#endif
 int l_vfs;
#ifdef _LARGE_FILES
 /* If LARGE FILES then off_t is a long long and struct*/
  /* flock must be laid out as struct flock64 */
 off_t l_start;
 off_t l_len;
#endif
```

Satzsperren haben eine Reihe von Nachteilen.

- Die Sperrarten sind wenig genormt, z.B. kennt POSIX nicht mandatory locks
- Sperren funktionieren nicht bei `open` und `unlink`-Aufrufen.

- die Verwendung der `flock`-Struktur ist mühsam, der `flock`-Aufruf etwas komfortabler, aber nicht so flexibel wie `fcntl`.
- Ein böswilliger Anwender kann durch Setzen und Halten einer Lesesperre auf die ganze Datei den schreibenden Zugriff aller anderen Anwender blockieren.

Im Programm 4.2 gibt Gray ein ausführliches Anwendungsbeispiel.

4.4 Signale

Wie oben angedeutet, wollen wir Signale nicht ausführlicher behandeln. Signale sind nicht sehr portabel und die nützlichste Variante, bei der man eigene Signalbehandlungsroutinen schreibt und an Ereignisse bindet, ist trickreich und nicht sehr sicher.

Stevens [Stev92] beschreibt ganz anschaulich die Probleme, wenn im signal handler sog. **non-reentrant** Systemaufrufe auftreten. Wird z.B. ein `malloc`-Aufruf durch ein Signal unterbrochen, dessen Behandlungsroutine erneut `malloc` aufruft, dürfte der Absturz vorprogrammiert sein.

Für unsere Zwecke ganz nützlich ist der `alarm` Systemaufruf, der wie ein Wecker funktioniert und bei Auslösen des Alarms das Signal SIGALRM erzeugt.

Include File(s)	<unistd.h>		Manual	**2**
Summary	`unsigned alarm( unsigned seconds );`			
Return	Success	Failure	Sets `errno`	
	restliche Zeit auf dem Wecker		No	

Tab. 4–3 *alarm Systemaufruf*

Jeder Prozeß kann höchstens einen Wecker haben. Dieser wird durch den Aufruf von `alarm` geschaffen, bzw. neu gesetzt. Speziell löscht ein Auf-

ruf mit dem Wert 0 (Sekunden) den Wecker, liefert aber die vor der Löschung verbliebene Restzeit zurück.

Gray gibt eine Reihe netter Programme 4.3 - 4.7 im Zusammenhang mit Signalen an. Wer gezwungen ist, mit Signalen zu arbeiten, z.B. in einer speziellen Anwendung ^C „fangen“ muß, sollte hier reinschauen.

5 Pipes

5.1 Einleitung

Das Konzept der Pipelineverarbeitung, bei der die Ausgabe eines Programms in ein anderes umgeleitet wird, das als Filter fungiert und seine Ausgabe wiederum in eine **pipe** steckt, ist eines der großartigsten Konzepte in UNIX.

Generell ist eine Pipe ein begrenzter FIFO-Puffer (POSIX 512 bytes, AIX 32 KB, siehe PIPE_BUF Konstante in `limits.h`), in den ein Prozeß hineinschreibt und aus dem ein anderer liest. Versucht ein Prozeß, in eine volle Pipe zu schreiben, wird er blockiert. Gleiches gilt für einen Prozeß, der aus einer leeren Pipe lesen will.

Ferner hält ein Prozeß beim Öffnen einer Pipe an, wenn noch kein anderer Prozeß sie zum Schreiben geöffnet hat. Letzterer Punkt ist der normale Vorgang, wie wir sehen werden, kann aber ggf. eine Verklemmung verursachen.

Durch eine pipe entsteht also eine natürliche Synchronisation und es werden Informationen (Daten) vom Erzeugerprozeß an den Verbraucherprozeß weitergegeben. Das Schreiben in die Pipe verwendet den ungepufferten I/O Systemaufruf `write`.

Dieser versucht, `nbyte` viele Bytes aus dem Puffer, auf den `buf` zeigt, in die Pipe, gegeben durch den Dateideskriptor `fildes`, zu schreiben. Gelingt dies, wird die Anzahl der tatsächlich geschriebenen Bytes zurückgeliefert, sonst -1 und es wird eine der zahlreichen Fehlermöglichkeiten angezeigt.

Include File(s)	<unistd.h>		Manual	**2**
Summary	ssize_t write(int fildes, const void *buf, size_t nbyte);			
Return	Success	Failure	Sets `errno`	
	Anzahl der geschriebenen Bytes	-1	Yes	

Tab. 5–1 *write Systemaufruf*

Für `write` in Zusammenhang mit Pipes gilt:

- Es wird immer ans Ende der FIFO-Pipe geschrieben.
- Ein Schreiben innerhalb der PIPE_BUF Grenzen ist **atomar**, d.h. kein anderer Prozeß kann dieser Schreiben unterbrechen und selbst in die Pipe schreiben.
- beim Öffnen und später durch Aufruf von `fcntl` kann das O_NONBLOCK Flag (auch O_NDELAY) gesetzt werden; für das angegebene „Gerät“ (file descriptor) sind damit alle Operationen (`open`, `read`, `write`) nichtblockierend; d.h. Operationen, die nicht sofort ausgeführt werden können, werden abgebrochen und melden sich sofort zurück. Pipes werden in der Regel aber **blockierend** betrieben.
- Versucht ein Prozeß in eine Pipe zu schreiben, für die noch kein lesender Prozeß ein `open` gemacht hat, wird dem Prozeß das „**broken pipe**“-Signal (SIGPIPE) geschickt, das unabgefangen den Abbruch des schreibwilligen Prozesses auslöst.

Aus der Pipe wird mit dem ungepufferten `read` Systemaufruf gelesen.

Include File(s)	<sys/types.h> <sys/uio.h> <unistd.h>	Manual	**2**

Summary	ssize_t read(int fildes, const void *buf, size_t nbyte);		
Return	Success	Failure	Sets `errno`
	Anzahl der gelesenen Bytes	-1	Yes

Tab. 5–2 *read Systemaufruf*

Durch den Aufruf werden `nbyte` viele Bytes (oder weniger) in den mit `buf` angegebenen Bereich aus der durch `fildes` angegebenen Datei (Pipe, Gerät) gelesen. Die tatsächlich gelieferte Anzahl der Zeichen wird zurückgegeben, speziell 0 bei Dateiende.

Für `read` in Verbindung mit pipes gilt:

- es wird von der gegenwärtigen Leseposition im FIFO-Puffer gelesen (kein seek)[1].
- ist O-NONBLOCK und O_NDELAY nicht gesetzt, blockiert der `read` Aufruf (Standardvorbelegung), bis wieder Daten vom schreibenden Ende kommen oder der Erzeuger die Pipe schließt.
- ist O_NDELAY gesetzt und ist die Pipe fürs Schreiben geöffnet worden, aber momentan leer, liefert read 0 zurück.
- Wurde die Pipe nicht von einem anderen Prozeß fürs Schreiben geöffnet, liefert read 0 (end-of-file) zurück. Diese Situation ist (systemabhängig) ggf. nicht unterscheidbar von read mit nichtgesetztem O_NDELAY und offener, aber leerer Pipe.

Generell gibt es „benamte“ (FIFO’s) und „unbenamte“ Pipes, letztere sind aus der Shell mit `cmd1 | cmd2` her bekannt. Die „normalen“ (**unnamed**) Pipes können nur einen Datenfluß zwischen Vater/Sohn und Sohn/Sohn herstellen und funktionieren generell nur in eine Richtung (half duplex).

1. in der Regel vom Anfang (hier stand früher im Skript „vom Ende“, was sicher falsch ist)

5.2 NamenlosePipes

Die namenlosen Pipes werden mit dem `pipe` Systemaufruf konstruiert.

Include File(s)	<unistd.h>		Manual	**2**
Summary	int pipe(int filedes[2]);			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Tab. 5–3 *pipe Systemaufruf*

Bei Erfolg liefert der Aufruf zwei Dateideskriptoren `filedes[0]` und `filedes[1]` zurück, die sich auf zwei Datenströme beziehen. Bach [8] ist etwas genauer: es werden **zwei Einträge** der globalen Dateitabelle belegt, deren Indexwerte an `filedes[0]` und `filedes[1]` zurückgegeben werden. Die beiden Einträge in dieser globalen Systemdateitabelle zeigen aber auf den gleichen i-Knoten, dessen Referenzzähler damit auf 2 hochgeht. Dies spielt in Abb. 5–1 eine Rolle.

Obwohl die meisten UNIX-Systeme heute eine Duplexpipe erzeugen, d.h die Enden für Lesen und Schreiben öffnen, ist die normale Pipe per Konvention nur halb-duplex in der Form, daß in das `filedes[1]`-Ende geschrieben wird und aus dem `filedes[0]`-Ende gelesen wird.

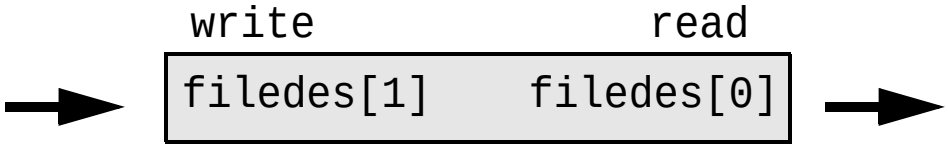
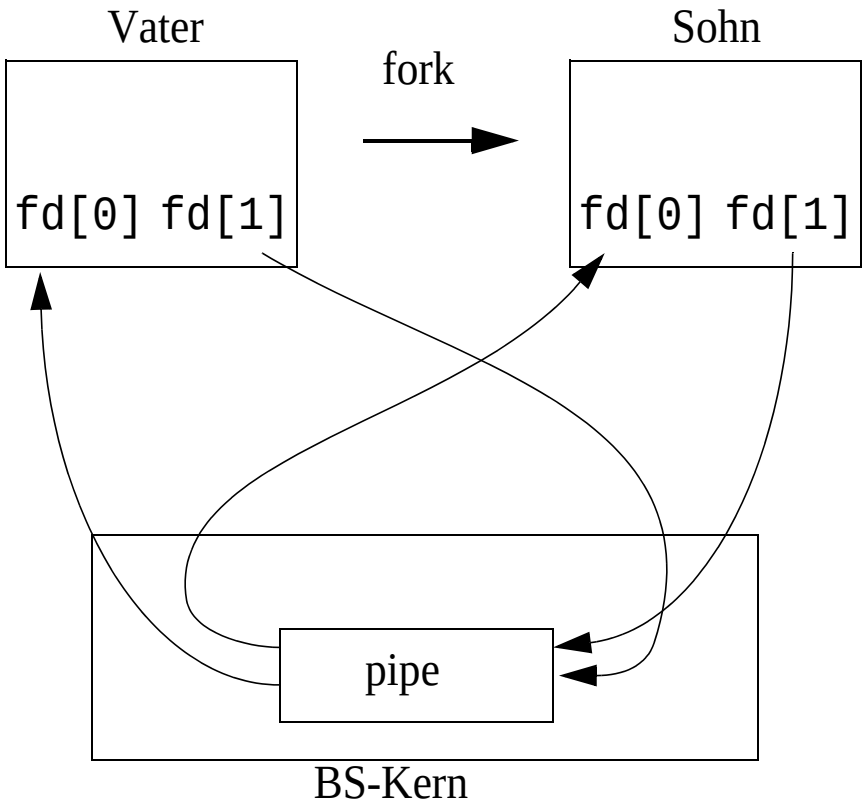


Abb. 5–1

An den Aufruf von `pipe` schließt sich in der Regel sofort ein `fork` an. Danach wird die Pipe in eine Richtung „dicht“ gemacht. Möchte man Datenfluß vom Vater zum Sohn, schließt der Vaterprozeß `filedes[0]` und der Sohn `filedes[1]`.



Das Schließen der überzähligen Deskriptoren wäre zwar nicht unbedingt notwendig, allerdings funktioniert ein `read` mit einer ungenügenden Anzahl zu lesender Bytes, bzw. einer leeren Pipe, nur dann und liefert EOF, wenn alle schreibenden Filedeskriptoren geschlossen wurden. Das folgende Programm 5.1 zeigt den Gebrauch für die Übertragung einer Botschaft, die als Kommandozeilenargument übergeben wird.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
main(int argc, char *argv[ ]) {
 int f_des[2];
 static char message[BUFSIZ];
 if (argc != 2) {
  fprintf(stderr, „Usage: %s message\n“, *argv);
  exit(1);
 }
 if (pipe(f_des) == -1) { // generate the pipe
  perror(„Pipe“);
```

```
    exit(2);
   }
   switch (fork( )) {
   case -1:
    perror(„Fork“);
    exit(3);
   case 0: // In the child
    close(f_des[1]);
    if (read(f_des[0], message, BUFSIZ) != -1) {
       printf(„Message received by child: \
          [%s]\n“, message);
       fflush(stdout);
    } else {
       perror(„Read“);
       exit(4);
    }
    break;
   default: // In the Parent
    close(f_des[0]);
    if (write(f_des[1], argv[1], strlen(argv[1])) != -1) {
       printf(„Message sent by parent : [%s]\n“,
          argv[1]);
    fflush(stdout);
    } else {
       perror(„Write“);
       exit(5);
    }
   }
   exit(0);
  }
```

Die Ausgabe lautet:

```
wegner@elsie(chpt5)$ ./p51 “In die Röhre geschaut“
Message received by child: [In die Röhre geschaut]
Message sent by parent : [In die Röhre geschaut]
```

Übung 5–1

Modifizieren Sie Programm 5.1 so, daß der Sohnprozeß nach Erhalt der Botschaft diese zurück an den Vater schickt. Sofern ihr UNIX-System nicht Duplexpipes unterstützt, müßte eine zweite Pipe aufgemacht werden.

Meist werden Pipes ja auf Kommandoebene in der Form

```
sort anmeldung | uniq | lpr
```

oder wie bei Gray mit dem `last`-Kommando (login/logout Info)

```
last | sort
```

aufgerufen. Damit brauchen wir einen Mechanismus, der auch für Sohn/Sohn-Paare funktioniert. Diese Aufgabe kommt dem `dup`-Systemaufruf, bzw. dem Nachfolger, der heute üblichen `dup2`-Bibliotheksroutine zu.

Include File(s)	<unistd.h>		Manual	**2**
Summary	int dup(int fildes);			
Return	Success	Failure	Sets `errno`	
	nächster verfügbarer nicht-negativer Dateideskriptor	-1	Yes	

Tab. 5–4 *dup Systemaufruf*

Der Systemaufruf `dup` dupliziert den als Argument übergebenen offenen Dateideskriptor und liefert als Resultat den Index des kleinsten freien Eintrags der Dateitabelle des Prozesses, also einen Dateideskriptor, zurück. Damit zeigen `fildes` und der zurückgelieferte **fd** auf den selben Eintrag der globalen Dateitabelle, teilen sich demnach die Zugriffsrechte, den Offset-Zeiger, und der neue Deskriptor erhält Flags, die bewirken, daß die Dateideskriptoren über einen `exec`-Aufruf hinaus offen bleiben.

Der übliche Gebrauch für die Umlenkung der Ausgabe, z.B. in eine Pipe, ist

```
int f_des[2];
pipe(f_des)
close( fileno(stdout) );
dup(f_des[1]);
close(f_des[0]);
```

```
close(f_des[1]);
...
```

Dabei wird eine Pipe angelegt. Bei der heute üblichen Duplexpipe wohl mit den Deskriptoren 3 und 4 wie unten gezeigt, wobei 4 als Schreibende genutzt werden soll. Danach wird `stdout` geschlossen, üblicherweise also Deskriptor 1. Jetzt müsste `dup` diesen kleinsten freien Eintrag verwenden und f_des[1] dorthin kopieren. Demnach würden 1 und 4 die Pipe referenzieren. Anschließend können wir 3 und 4 wieder schließen.

Der Vorgang hat einen kleinen Haken. Sollte zwischen dem Schließen von `stdout` und dem Aufruf von `dup` ein Prozessorwechsel erfolgen, z.B. als Folge eines Signals, und würden in dieser Zeit Veränderungen an der Dateitabelle des Prozesses erfolgen, z.B. ein Schließen aller Einträge 0 - 2 durch eine Abräumroutine, liefert anschließend `dup` den falschen Eintrag.

Dies vermeidet `dup2` indem es beide Vorgänge (Schließen von `fildes2` und Kopieren von `filedes` nach `fildes2`) **atomar** vornimmt. Es gilt also: ist `fildes2` vor dem Duplizieren offen, wird der (alte `fildes2`) Deskriptor vorher geschlossen.

Include File(s)	<unistd.h>		Manual	**3c**
Summary	int dup2(int fildes, int fildes2);			
Return	Success	Failure	Sets `errno`	
	nächster verfügbarer nicht-negativer Dateideskriptor	-1	Yes	

Tab. 5–5 *dup2 Bibliotheksroutine*

Das Programm unten bildet das Kommando `last | sort` nach. Man beachte, daß vom Sohn zum Vater hin „gepiped“ wird.

```
main(void) {
 int f_des[2];
```

```
    if (pipe(f_des) == -1) {
     perror(„Pipe“);
     exit(1);
    }
    switch (fork()) {
    case -1:
     perror(„Fork“);
     exit(2);
    case 0: // In the child
     dup2(f_des[1], fileno(stdout));
     close(f_des[0]);
     close(f_des[1]);
     execl(„/usr/bin/last“, „last“, (char *) 0);
     exit(3);
    default: // In the parent
     dup2(f_des[0], fileno(stdin));
     close(f_des[0]);
     close(f_des[1]);
     execl(„/bin/sort“, „sort“, (char *) 0);
     exit(4);
    }
   }
```

Übung 5–2

Man schreibe drei kleine Programme: einen Erzeuger `Tic`, einen Verbraucher `Tac` und ein Pipe-Programm `Uhr`, das `Tic` mit `Tac` durch `pipe` und `dup2` verbindet, analog zu Programm 5.2 oben. Dabei liefert `Tic` im Sekundentakt einen Text ab, z.B. die Uhrzeit analog zu `date`. `Tac` macht daraus eine Bildschirmausgabe: „Beim nächsten Ton ist es ...“.

Weil die Folge von Aufrufen `pipe`, `fork`, `dup2`, `close` so häufig vorkommt, hat man in UNIX zwei I/O-Funktionen `popen` und `pclose`, wobei `popen` einen **Dateizeiger** (pointer auf FILE), keinen Integer Dateideskriptor, zurückliefert und als Argument ein Shell-Kommando und eine Modusangabe (`r` = read, `w`= write) verlangt. Der Aufruf `popen` erzeugt einen Sohn, der durch eine Bourne-Shell überlagert wird (`exec`), die wiederum das Kommando ausführt, in das man per Pipe schreibt, bzw. das einem Daten per Pipe zum Lesen liefert. Wir übergehen die Details, Programm 5.3 arbeitet mit `popen` und `pclose`.

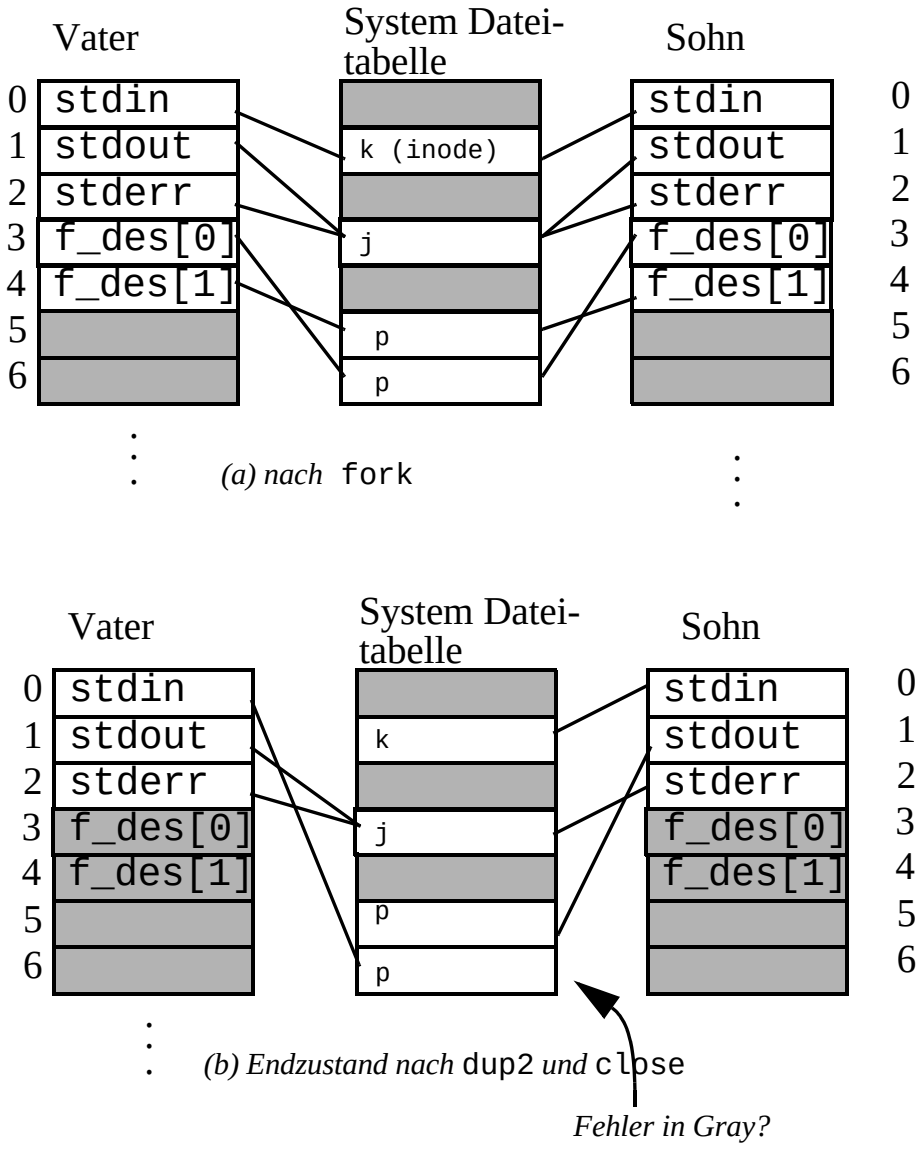


Abb. 5–2

Übung 5–3

Wenn man Programm 5.3 wie unten gezeigt aufruft, werden Daten vom ersten zum zweiten Argument weitergeleitet.

```
wegner@elsie(chpt5)$ ./p53 date cat
Wed May 13 18:21:28 MES 1998
```

Man modifiziere `p53` zu `p53b` so, daß analog zu dem `tee`-Kommando eine Kopie in eine Datei geht, wenn es mit der Option -c aufgerufen wird, also z.B. mit

```
./p53b -c text.txt date cat
```

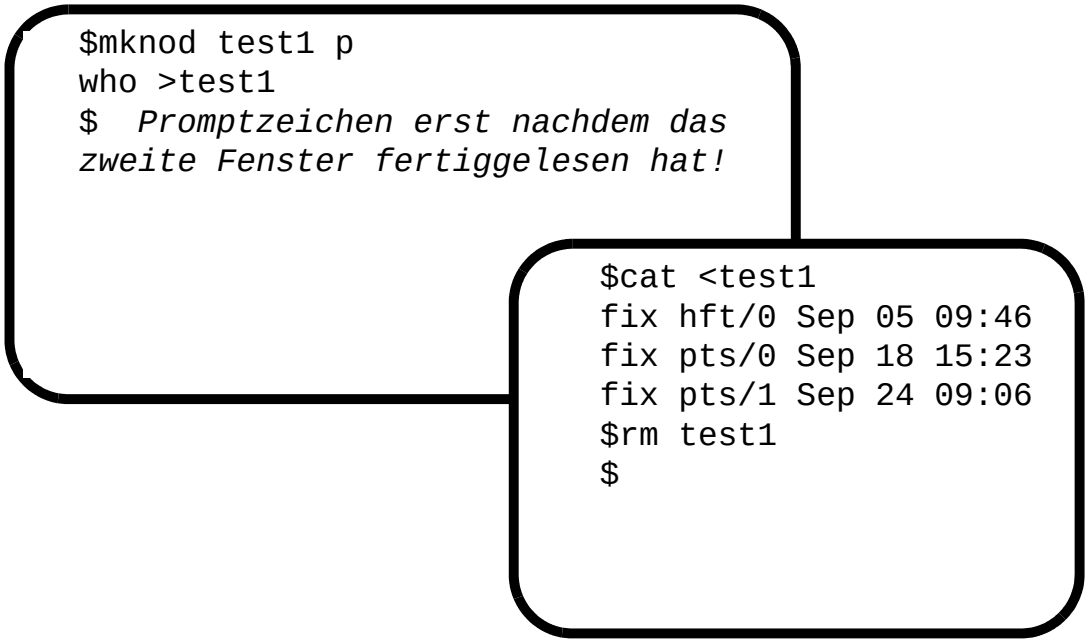
aufgerufen wird.

5.3 FIFO Pipes

Der Nachteil, daß man nur direkt verwandte Prozesse mit der normalen Pipe verbinden kann, läßt sich mit der benannten (FIFO, named pipe) vermeiden. Im UNIX-Kurs machten wir dazu die folgenden Aussagen:

> Behandelt man dagegen die Pipe wie eine temporäre Datei, kann man einen uni-direktionalen Datenstrom zwischen nahezu beliebigen Kommandos aufbauen. Dies ist die Grundidee der benannten Pipe (*named pipe*, *FIFO*). Mit dem Kommando `mknod` *name type* läßt sich eine Spezialdatei mit Namen *name* anlegen, in unserem Fall für `test1` durch Angabe des Typs `p` eine benannte Pipe.

In einem Mehrfenstersystem kann man sich jetzt Daten von einem Fenster in ein anderes schicken, im Beispiel die Ausgabe von `who`.



> Ohne mehrere Fenster muß man die Umlenkung als Hintergrundprogramm starten (siehe folgende Abschnitte), da der Absender blockiert wird und auf die Aktivierung des Empfängers wartet.
>
> Named pipes oder FIFOs (first in first out Pipes), wie sie z. B. im POSIX-Standard heißen, werden im Dateisystem abgelegt. Daraus ergeben sich schon einige Einschränkungen, etwa für die Nutzung über Rechnergrenzen hinweg. Daneben ist ein korrektes Verhalten in einer Client-Server-Umgebung, also mit mehreren lesenden oder schreibenden Prozessen (Kunden) und einem bedienenden Prozeß (dem Auftragnehmer) schwierig herzustellen, da named pipes prinzipiell wie die normale Pipe uni-direktional sind und keine eigene Mechanismen für die Prozeßsynchronisierung (siehe nebenläufige Prozesse im Abschnitt 3.3 unten) bereitstellen.

Dem ist zunächst nichts hinzuzufügen. Auf die Verwendung der FIFOs aus einem Programm heraus gehen wir unten ein.

Übung 5–4

Gray erzeugt eine named pipe namens PIPE und zeigt sie im Arbeitsverzeichnis mit `ls -l` an. In der Regel wird die Längenangabe 0 sein. Gray weist aber darauf hin, daß die pipe die Daten hält, solange noch ein („unbefriedigter“) Leser existiert, so daß sie dann mit einem Wert >0 angezeigt werden. Wie kann das funktionieren?

> **Hinweis:** Mit viel Mühe ist es mir gelungen, die Situation wie folgt zu konstruieren. Ich habe ein Shellskript `tester` mit folgenden Inhalt:

```
while true
do
 sleep 30
 cat -n
done
```

Das rufe ich mit `./tester <PIPE >myfile` auf.

Die Eingabe produziere ich mit `cat >PIPE` und tippe dann Zeilen ein, bis ich mit *Ctrl-D* die Eingabe beende. In einem dritten Fenster schaue ich mit `ls -l` die momentane Situation an. Zu einem Zeitpunkt lautete sie:

```
wegner@elsie(chpt5)$ ls -l
total 184
prw-r--r-- 1 wegner mitarb        12 May 14 PIPE
-rw-r--r-- 1 wilke mitarb        462 Apr 23 gcc.error
-rw-r--r-- 1 wilke mitarb        339 Sep 18 local.h
-rw-r--r-- 1 wegner mitarb         0 May 14 myfile
-rwxr-xr-x 1 wilke mitarb      11685 Apr 27 p51
-rw-r--r-- 1 wilke mitarb        984 Sep 18 p51.c
...
-rw-r--r-- 1 wilke mitarb       1472 Sep 18 p55.c
-rwxrwxrwx 1 wegner mitarb        40 May 14 tester
```

Versuchen Sie, die Situation zu rekonstruieren. Können Sie eine einfachere Lösung finden?

Aus einem Programm heraus lautet der Aufruf zur Erstellung einer FIFO-Pipe ebenfalls `mknod`.

Include File(s)	<sys/types.h> <sys/stat.h>		Manual	**2**
Summary	int mknod(const char *path, mode_t mode, dev_t dev);			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Tab. 5–6 *mknod Systemaufruf*

Der Systemaufruf `mknod` legt die durch `path` definierte Datei an. Der Typ und die Zugriffsrechte bestimmen sich aus dem `mode` Argument, meist durch ODER-Verknüpfung entsprechender Konstanten.

Symbolische Konstante	Dateityp
S_IFIFO	FIFO special
S_IFCHR	character special

Symbolische Konstante	Dateityp
S_IFDIR	Verzeichnis
S_IFBLK	block special
S_IFREG	normale Datei

Tab. 5–7 *Spezifikation des Dateityps*

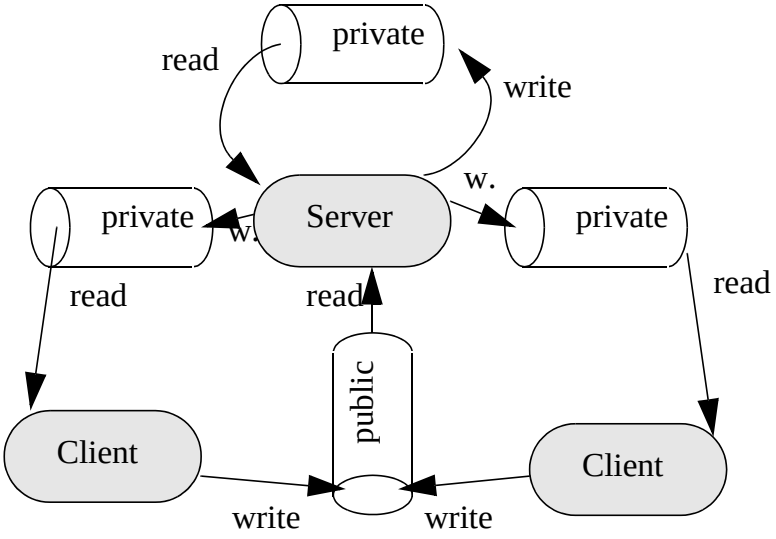
Das `dev`-Argument wird für *character* und *block special devices* gebraucht. Benutzer, die nicht super user sind, dürfen nur FIFOs anlegen. In diesem Fall sollte `dev` auf 0 gesetzt werden.

In vielen Systemen gibt es für den Spezialfall der named pipe den Bibliotheksaufruf `mkfifo`, der aber selbst wieder `mknod` aufruft.

Include File(s)	<sys/types.h> <sys/stat.h>		Manual	**3c**
Summary	int mkfifo(const char *path, mode_t mode);			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Tab. 5–8 *mkfifo Systemaufruf*

Das folgende größere Beispiel erstellt eine Client-Server-Umgebung. In dieser werden Shellkommandos von den Kundenprozessen über eine öffentliche named pipe an den Server geschickt, der diese via popen und pclose abarbeitet und das Ergebnis über private named pipes an die Kunden zurückschickt.



Die Schritte im einzelnen sind:

- Server erzeugt öffentlichen FIFO, der allen beteiligten Clients zugänglich ist.
- Client-Prozeß eröffnet seinen privaten FIFO.
- Client verlangt Kommandoeingabe und erhält sie.
- Client schreibt Namen der privaten FIFO und Shellkommando in öffentliche FIFO
- Server liest öffentliche FIFO mit Namen der privaten FIFO und dem Kommando.
- Server macht `popen` und `pclose` für Ausführung des Shellkommandos. Die Ausgabe geht über die private FIFO zurück an den Kunden.
- Client gibt die Ausgabe aus.

Für die Namenskoordinierung wird ein gemeinsames Headerfile `local.h` angelegt.

```
wegner@elsie(chpt5)$ less local.h
/* local.h */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>

#define PUBLIC „/tmp/PUBLIC“
#define B_SIZ (PIPE_BUF / 2)

struct message {
 char fifo_name[B_SIZ];
 char cmd_line[B_SIZ];
};
```

Speziell wird eine **öffentliche named pipe** `/tmp/PUBLIC` definiert und `message` als eine Struktur mit zwei Zeichenkettenfeldern festgelegt.

Das folgende Programm 5.4 zeigt den Client.

```
/* Client */

#include „local.h“

void
main( void ){
 int n, privatefifo, publicfifo;
 static char buffer[PIPE_BUF];
 struct message msg;

 // Make the name for the private FIFO
 sprintf(msg.fifo_name, „/tmp/fifo%d“, getpid( ));

 // Generate the private FIFO
 if (mknod(msg.fifo_name, S_IFIFO | 0666, 0) < 0) {
  perror(msg.fifo_name);
  exit(1);
 }

 // OPEN the public FIFO for writing
```

```
 if ((publicfifo = open(PUBLIC, O_WRONLY)) == -1) {
  perror(PUBLIC);
  exit(2);
 }

 while ( 1 ) { // FOREVER
   write(fileno(stdout), „\ncmd>“, 6); // prompt
   memset(msg.cmd_line, 0x0, B_SIZ); // clear first
   n = read(fileno(stdin), msg.cmd_line, B_SIZ);
              // Get command
   if (!strncmp(„quit“, msg.cmd_line, n - 1)) // EXIT ?
   break;
   write(publicfifo, (char *) &msg, sizeof(msg));
    // to PUBLIC

   // OPEN private FIFO to read returned command output
   if ((privatefifo = open(msg.fifo_name, O_RDONLY)) ==
-1){
    perror(msg.fifo_name);
    exit(3);
   }

   // READ private FIFO and display on standard error
   while ((n = read(privatefifo, buffer, PIPE_BUF)) > 0)
{
    write(fileno(stderr), buffer, n);
   }
   close(privatefifo);
 }       // WHILE
 close(publicfifo);
 unlink(msg.fifo_name); // REMOVE
}
```

Die beachtenswerten Punkte beim Kundenprozeß sind die Namensgenerierung für die private FIFO-Pipe mittels `sprintf`. Problematisch ist das `open` der öffenlichen Pipe. Sollte der Server diese noch nicht angelegt haben, folgt ein Abbruch des Client. Beim Lesen des Kommandos aus der Eingabezeile schneiden wir durch `n-1` das *newline* der Eingabe ab. Diese geben wir über die gesamte `message`-Struktur an den Server und versuchen anschließend über `read` aus der privaten FIFO zu lesen. Bei langen Kommandos könnte der Server noch kein Resultat haben, weshalb `read` den Prozeß standardgemäß in den Wartezustand versetzt. Kommt die Eingabe dann, wird sie auf der Standardfehlerausgabe ausgegeben.

Es folgt das Programm für den Server.

```
/* Server */

#include „local.h“
main( void ){
 int n, done, dummyfifo, publicfifo, privatefifo;
 struct message msg;
 FILE *fin;
 static char buffer[PIPE_BUF];

 // Generate the public FIFO
 mknod(PUBLIC, S_IFIFO | 0666, 0);

 // OPEN public FIFO for reading and writing
 if ((publicfifo = open(PUBLIC, O_RDONLY)) == -1 ||
 (dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY)) == -1) {
  perror(PUBLIC);
  exit(1);
 }

 // message can be read from the PUBLIC pipe
 while (read(publicfifo, (char *) &msg, sizeof(msg)) >
0) {
  n = done = 0; // clear counters / flags
 do { // try OPEN of private FIFO
  if ((privatefifo = open(msg.fifo_name, O_WRONLY |
         O_NDELAY)) == -1)
     sleep(3); // sleep a while
  else { // OPEN successful
     fin = popen(msg.cmd_line, „r“);
        // execute the command
     write(privatefifo, „\n“, 1);
        // keep output pretty
     while ((n = read(fileno(fin), buffer,
           PIPE_BUF)) > 0) {
       write(privatefifo, buffer, n);
        // to private FIFO
       memset(buffer, 0x0, PIPE_BUF);
        // clear between times
     }
     pclose(fin);
     close(privatefifo);
     done = 1; // record success
  }
```

```
 } while (++n < 5 && !done);
if (!done) // Indicate failure
  write(fileno(stderr),
 „\nNOTE: SERVER ** NEVER ** accessed private FIFO\n“,
48);

 }
}
```

Man beachte insbesondere einen Trick, nämlich das Öffnen der public FIFO für Lesen **und** Schreiben, obwohl aus ihr nur gelesen wird. Damit gibt es aber mindestens einen schreibenden Prozeß, wodurch bei einem end-of-line die Pipe nicht geschlossen wird. Das erspart das Schließen und Neuöffnen der Pipe beim Ausstieg eines Clients.

Ferner verwendet der Server beim Öffnen der privaten pipe zum Client das O_NDELAY Flag. Damit wird eine Blockade des Servers verhindert, falls ein Client kein `open` auf der privaten Pipe gemacht hat. Vielmehr versucht der Server es noch 5 weitere Male mit jeweils 3 Sekunden Pause dazwischen, bevor er aufgibt und die private Pipe freigibt.

Im Erfolgsfall wird durch `popen` das Kommando als Unterprozeß ausgeführt und sein Ergebnis mittels `read` von der namenlosen Pipe in den Server geleitet und von dort über `write` auf der privaten FIFO an den Client weitergereicht.

Ein Probelauf mit den Kommandos `ps` und `who` ergibt das folgende Resultat, wobei der im Hintergrund laufende Server durch `kill -9` explizit umgebracht werden muß.

```
wegner@elsie(chpt5)$ ./p55 &
[1] 16044
wegner@elsie(chpt5)$ ./p54

cmd>ps

 PID TTY TIME CMD
 16044 pts/3 0:00 ./p55
 19044 pts/3 0:00 /usr/bin/bash
 19374 pts/3 0:00 ./p54
 20146 pts/3 0:00 ps

cmd>who
```

```
wegner . May 18 09:54
wegner pts/0 May 18 09:56 (:0.0)
wegner pts/1 May 18 09:56
wegner pts/2 May 18 10:09 (:0.0)
wegner pts/3 May 18 13:57 (:0.0)

cmd>quit
wegner@elsie(chpt5)$ kill -9 16044
wegner@elsie(chpt5)$
```

Übung 5–5

Startet man den Client, ohne daß der Server läuft, gibt es eine Fehlermeldung (`/tmp/PUBLIC: No such file or directory`). Man verbessere das Beispiel dahingehend, dass der Client in diesem Fall den Server startet.

Zusammenfassung

Pipes und speziell die benannten Pipes, die im Dateisystem abgelegt werden, erlauben eine einfache Kommunikation. Allerdings gibt es keine explizite Synchronisation, Anwendungen sind auf einen einzelnen Rechner beschränkt, und es kann zu Verklemmungen kommen.

6 Botschaften (Message Queues)

6.1 Einleitung

In [Gray97] folgen drei Kapitel zu den Themen *Message Queues*, *Semaphores* und *Shared Memory.* Diese Interprocess Communications (IPCs) wurden als Teil von System V zu UNIX hinzugefügt.

- Botschaftenwarteschlangen (message queues) erlauben es, Informationen in vordefinierte Botschaftsstrukturen abzulegen. Der Absender kann deren Typ bestimmen. Lesende Prozesse können selektiv nach Typ Botschaften in FIFO-Manier auswählen. Zugleich können mehrere Produzenten in die Warteschlangen Botschaften ablegen.
- Semaphore sind die klassischen Prozeßsynchronisierungsmittel. Sie übertragen allerdings keine Information. Wegen der Notwendigkeit, Sperren vom System freigeben zu müssen, wenn der sperrende Prozeß abgestürzt ist, ist die Realisierung in UNIX eher unhandlich. Wir werden auf die Besprechung im Detail verzichten.
- Zugriff auf gemeinsamen Speicher (shared memory) ist die sicherlich schnellste Interprozeßkommunikation. Häufig wird er über Semaphore synchronisiert. Auch shared memory werden wir übergehen.

IPC Einrichtungen müssen angelegt werden vor dem ersten Gebrauch. Jede Einrichtung hat einen Erzeuger, Besitzer und Zugriffsrechte. Mittels des `ipcs`-Kommandos kann man sich die momentan existierenden Einrichtungen ansehen, die Option `-b` liefert Größeninfo. (max. Botschaften-

größen, max. Segmentgröße in Bytes, Anzahl Semaphore in Semaphormenge).

```
wegner@elsie(wegner)$ ipcs -b
IPC status from /dev/mem as of Mon May 18 16:54:01 MES
1998
T ID KEY MODEOWNER GROUP QBYTES
Message Queues:
q 0 0x4107001c -Rrw-rw---- root printq 65535
T ID KEY MODE OWNER GROUP SEGSZ
Shared Memory:
m 0 0x58065165 --rw-rw-rw- root system 268435456
T ID KEY MODE OWNER GROUP NSEMS
Semaphores:
s    0 0x58065165 --ra-ra-ra- root system 1
s    1 0x44065165 --ra-ra-ra- root system 2
s 4098 0x0106322b --ra------- root system 1
s    3 0x620630da --ra-r--r-- root system 1
```

Der Buchstabe `R` im Modus bei den Queues zeigt an, daß die Schlange auf ein `msgrcv` wartet (`S` = warten auf `msgsnd`, `D` = shared memory removed, disappears when last process detached, `C` = shared memory will be cleared when first attached, `-` = flag not set).)

IPCs bleiben über die Lebenszeit des erzeugenden Prozesses hinaus mit ihrem momentanen Inhalt bestehen. Der Besitzer kann eine IPC-Einrichtung mittels `ipcrm` entfernen.

6.2 IPC Aufrufe - Eine Übersicht

Die System V IPC Aufrufe lassen sich wie folgt zusammenfassen.

Funktionalität	Message Queue	System Call Semaphore	Shared Memory
IPC anlegen, Zugriff auf IPC erlangen	msgget	semget	shmget
IPC kontrollieren, Status abfragen/modifizieren, IPC entfernen	msgctl	semctl	shmctl

Funktionalität	Message Queue	System Call Semaphore	Shared Memory
IPC Operation: send/receive messages, sem. operat., attach/free shared memory segment	msgsnd msgrcv	semop	shmat shmdt

Tab. 6–1 *Übersicht System V IPCs*

Der generische Begriff *get* (nicht zu verwechseln mit sccs `get`) bezeichnet dabei den Schritt der IPC-Erzeugung. Soll dies für mehrere, nichtverwandte Prozesse möglich sein, z.B. wenn mehrere Erzeuger Aufträge in die Warteschlange stellen, die mehrere Verbraucher abholen, dann muß

- die Warteschlange genau einmal erzeugt werden;
- welcher Prozeß sie erzeugt, soll beliebig sein;
- die Warteschlange muß eindeutig innerhalb des Systems sein;
- es muß dafür einen eindeutigen Bezeichner geben, den alle verwenden, wenn sie Aufträge an die Schlange schicken oder welche holen.

Das Problem ist, daß diese Vorabverständigung zwischen den Prozessen eine Art Synchronisation ist, die ja gerade durch die Message Queues, Semaphores, Shared Memory erreicht werden soll. Entsprechend schlecht erklären Stevens und Gray auch das Problem.

Grundsätzlich wird ein IPC-Konstrukt durch einen eindeutigen Integer bezeichnet, der einen Index in eine große IPC-Tabelle ist. Dieser Wert (Stevens: „slot usage sequence number“) wird vom System bestimmt und kontinuierlich hochgezählt, bis er irgendwann wieder auf Null springt.

6.3 Botschaftenwarteschlange anlegen

Der Botschaftenidentifier wird bei einem *get*-Aufruf zurückgeliefert, z.B. bei `msgget` wie unten gezeigt, erzeugt aber nur vom ersten Aufrufer.

<table>
<tr><td>Include File(s)</td><td colspan="2"><sys/types.h>
<sys/ipc.h>
<sys/msg.h></td><td>Manual</td><td>2</td></tr>
<tr><td>Summary</td><td colspan="4">int msgget(key_t key,
int msgflg);</td></tr>
<tr><td rowspan="2">Return</td><td>Success</td><td>Failure</td><td colspan="2">Sets <code>errno</code></td></tr>
<tr><td>Non-negative message queue identifier associated with key</td><td>-1</td><td colspan="2">Yes</td></tr>
</table>

Tab. 6–2 *msgget Systemaufruf*

Nach dem *get* können ihn alle Prozesse für Senden und Empfangen verwenden. Vor dem *get* kann er aber schlecht dafür verwandt werden, sich darüber zu verständigen, daß man eine bestimmtes IPC-Konstrukt zur Kommunikation verwenden will. Dafür verwendet man einen Schlüssel (key), über den man sich verständigt. Dieser Schlüssel ist wiederum eine ganze Zahl und kann im Prinzip beliebig gewählt werden. Eine Anwendung kann ihn z.B. in einem Headerfile ablegen.

Alternativ geht es etwas geordneter mit der `ftok` (file-to-key)Bibliotheksfunktion. Dabei wird aus einem Dateinamen und einem weiteren Zeichen (ein Integer im Bereich 0 .. 255, auch Projekt-ID genannt) auf eindeutige Weise ein Integerschlüssel produziert. Diesen selben Schlüssel verwenden dann alle Prozesse für ihren *get*-Aufruf.

Include File(s)	<sys/types.h> <sys/ipc.h>		Manual	**3c**
Summary	key_t ftok(const char *path, int id);			
Return	Success	Failure	Sets `errno`	
	key value for an IPC get system call	-1		

Tab. 6–3 *ftok Bibliotheksfunktion*

Zusätzlich gibt es noch einen speziellen Schlüssel IPC_PRIVATE (in Wirklichkeit 0), der immer das erstmalige Anlegen erzwingt. Dies kann man bei Vater/Sohn-Beziehungen verwenden und den Warteschlangen-ID vererben.

Generell kann eigentlich nur einer den IPC-Konstrukt anlegen. Je nach Anwendung muß dies ein bestimmter Prozeß sein (Fall **a**), dann müßten alle anderen Prozesse, die *get* aufrufen und die Warteschlange nicht vorfinden, dies als Fehler registrieren.

In anderen Anwendungen kann es egal sein, wer zuerst die Queue anlegt (Fall **b**). Diese Alternativen steuert man mit den Flags im `msgget`-Aufruf, z.B. IPC_CREAT und IPC_EXCL. Aus den Fehlermeldungen für `msgget` kann man die möglichen Kombinationen erkennen.

Speziell für Fall **b** wird man also *get* aufrufen mit IPC_CREAT (anlegen wenn nicht schon da) aber **ohne** IPC_EXCL (egal wenn schon da), damit `msgget` in jedem Fall den Warteschlangen-ID liefert und nicht Mißerfolg mit -1 signalisiert.

#	Konstante	Meldung	Erläuterung
2	EOENT	No such file or directory	Warteschlangenid existiert nicht für diesen Schlüssel und IPC_CREAT nicht gesetzt
13	EACCES	Permission denied	W-id existiert für diesen Schlüssel, aber gewünschte Operation nicht möglich bei gegenw. Rechten
17	EEXIST	File exists	W.-id existiert für diesen Schlüssel, aber IPC_CREAT und IPC_EXCL sind beide gesetzt
28	ENOSPC	No space left on device	Systemgrenze für Anzahl von Botschaftenwarteschl. überschritten

Tab. 6–4 *msgget Fehlermeldungen*

Das folgende Programm `p62` legt fünf message queues an mit Lese-/Schreibberechtigung und verwendet das `ipcs`-Kommando zur Statusanzeige via `popen`/`pclose`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX 5

main(void)
{
 FILE *fin;
 char buffer[PIPE_BUF], u_char = 'A';
 int i, n, mid[MAX];
 key_t key;
 for (i = 0; i < MAX; ++i, ++u_char) {
  key = ftok(„.“, u_char);
  if ((mid[i] = msgget(key, IPC_CREAT | 0770)) == -1) {
     perror(„Queue create“);
```

```
      exit(1);
    }
   }
   fin = popen(„ipcs“, „r“);
   while ((n = read(fileno(fin), buffer, PIPE_BUF)) > 0)
    write(fileno(stdout), buffer, n);
   pclose(fin);
   for (i = 0; i < MAX; ++i)
    msgctl(mid[i], IPC_RMID,(struct msqid_ds *) 0);
   exit(0);
  }
```

Die Ausgabe bei uns sieht wie folgt aus.

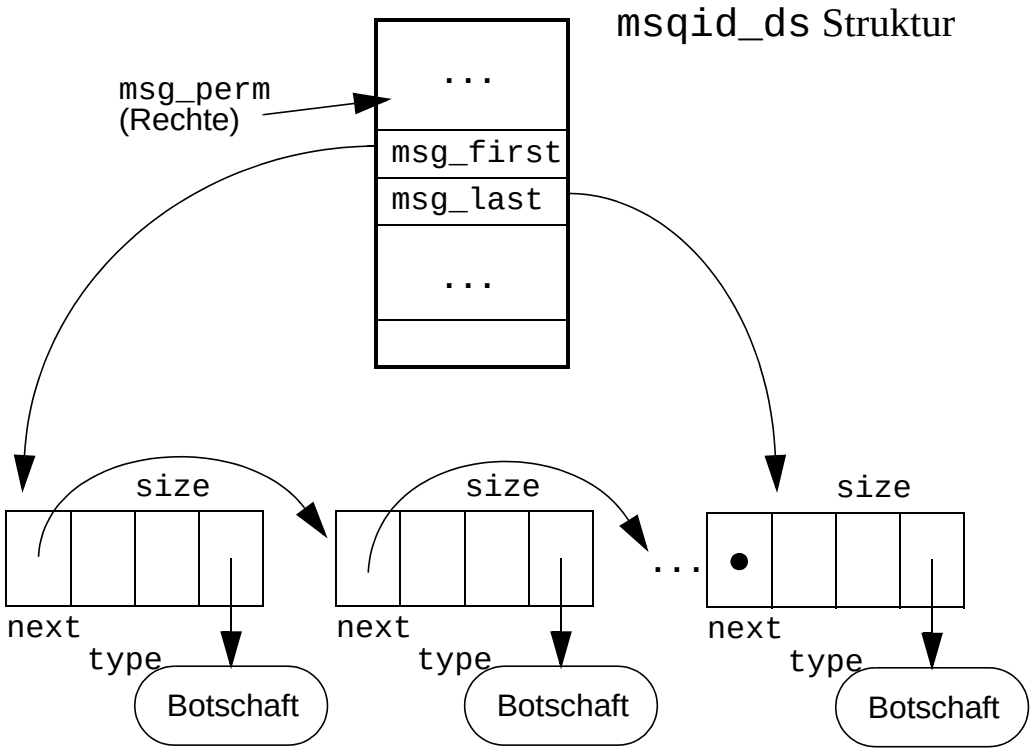
```
wegner@elsie(chpt6)$ ./p62
IPC status from /dev/mem as of Mon May 25 21:59:51 MES
1998
T ID KEY MODE OWNER GROUP
Message Queues:
q 0 0x4107001c -Rrw-rw---- root    printq
q 1 0x410b706d --rw-rw---- wegner mitarb
q 2 0x420b706d --rw-rw---- wegner mitarb
q 3 0x430b706d --rw-rw---- wegner mitarb
q 4 0x440b706d --rw-rw---- wegner mitarb
q 5 0x450b706d --rw-rw---- wegner mitarb
Shared Memory:
m 0 0x58065165 --rw-rw-rw- root system
Semaphores:
s 0 0x58065165 --ra-ra-ra- root system
s 1 0x44065165 --ra-ra-ra- root system
s 4098 0x0106322b --ra------- root system
s 3 0x620630da --ra-r--r-- root system
```

Nach noch zweimaligem Aufruf sind die IDs im übrigen wie folgt hochgezählt worden.

```
wegner@elsie(chpt6)$ ./p62
IPC status from /dev/mem as of Mon May 25 22:05:19 MES
1998
T ID KEY MODE OWNER GROUP
Message Queues:
q 0    0x4107001c -Rrw-rw---- root printq
q 8193 0x410b706d --rw-rw---- wegner mitarb
q 8194 0x420b706d --rw-rw---- wegner mitarb
q 8195 0x430b706d --rw-rw---- wegner mitarb
q 8196 0x440b706d --rw-rw---- wegner mitarb
```

```
q 8197 0x450b706d --rw-rw---- wegner mitarb
Shared Memory:
m 0 0x58065165 --rw-rw-rw- root system
Semaphores:
s 0 0x58065165 --ra-ra-ra- root system
s 1 0x44065165 --ra-ra-ra- root system
s 4098 0x0106322b --ra------- root system
s 3 0x620630da --ra-r--r-- root system
```

Die Warteschlangen sehen schematisch wie unten abgebildet aus.



Demnach gibt es je Warteschlange eine Struktur `msqid_ds` mit Rechten, Besitzer, Zeiten, etc., sowie einem Anfangs- und Endezeiger auf eine einfach verkettete Liste von Knoten (Queue Items). Diese Items enthalten Vorwärtszeiger, Botschaftentyp, Textgröße der Botschaft und einen Verweis (Gray: Typ `short` ?) auf den abgelegten Text. In AIX sieht das wie folgt aus:

```
struct msg {
 struct msg *msg_next; /* ptr to next message on q */
 struct msg_hdr msg_attr; /* message attributes */
 unsigned short msg_ts; /* message text size */
 char *msg_spot; /* pointer to message text */
};
```

6.4 Warteschlangenkontrolle

Mittels des `msgctl`-Aufrufs lassen sich einige Parameter der Queues abfragen und einstellen, sowie die Queues löschen. Die Syntax ist ziemlich scheußlich und verlangt Zeiger auf die Queue-Strukturen, die in `<sys/msg.h>` definiert sind. Programm 6.3 in Gray gibt einige Informationen dazu aus, wir übergehen diesen Teil hier.

6.5 Botschaften senden und empfangen

Die Botschaften bestehen aus einem Integerwert für den Typ gefolgt von einer Folge von Bytes.

```
struct msgbuf {
 mtyp_t mtype; /* message type */
 char mtext[1]; /* message text */
};
```

wobei `types.h` festlegt, daß für `mtyp_t` gilt

```
typedef long mtyp_t; /* ipc message type */
```

Über den Typ kann der Sender steuern, wer die Botschaft empfängt, bzw. wann eine Botschaft gelesen wird. Dies setzt ein informelles Protokoll voraus, damit der Empfänger weiß, nach welchem Typ von Botschaft er suchen soll. Gray und Bach geben Beispiele, bei denen ein Client PID als Typ genommen wird und Clients nur Botschaften lesen, deren Typ gleich ihrem PID ist.

Die Botschaften selbst werden beim Senden in den **Kernel Space** kopiert mittels `msgsnd(msqid, buf, msgsize, msgflag)`, wobei `msqid` der Warteschlangen-ID, `buf` ein Zeiger auf die Botschaft, `msgsize` die Botschaftenlänge, und `msgflag` Flags sind, die u.a. ange-

ben, wie zu verfahren ist, wenn keine Botschaften mehr in die Warteschlange aufgenommen werden können.

Analog sieht `msgrcv` (message receive) aus, das aus dem Kernel Space in den Anwenderbereich kopiert. Dabei ist msgtyp die Angabe des Typs von Botschaft, die man empfangen möchte. Speziell bedeutet 0 beliebigen Typ. Über die Flags kann bestimmt werden, ob gewartet werden soll, wenn keine Botschaft des gewünschten Typs vorliegt und wie mit zu langen Botschaften zu verfahren ist.

<table>
<tr><td>Include File(s)</td><td colspan="2"><sys/types.h>
<sys/ipc.h>
<sys/msg.h></td><td>Manual</td><td>2</td></tr>
<tr><td>Summary</td><td colspan="4">int msgrcv(int msqid,
void *msgp,
size_t msgsz,
long msgtyp,
int msgflg);</td></tr>
<tr><td rowspan="2">Return</td><td>Success</td><td>Failure</td><td colspan="2">Sets <code>errno</code></td></tr>
<tr><td>Anzahl der tatsächlich gelesenen Bytes</td><td>-1</td><td colspan="2">Yes</td></tr>
</table>

Tab. 6–5 *msgrcv Systemaufruf*

Gray gibt mit den Programmen 6.4 (client) und 6.5 (server) ein größeres Beispiel für die Verwendung, bei dem Texte vom Client an den Server gehen, der diese von Klein- in Großbuchstaben „übersetzt“ und wieder zurückschickt. Die Rückadresse wird per PID des Kundenprozesses bestimmt. In einer Mehrfensterumgebung kann man das probieren. Das Eingabeende signalisiert man mit *CTRL-D*.

Übung 6–1

Im Verzeichnis `chpt6` liegen die Programme `client` und `server`. Der Aufruf muß mit `./client &` erfolgen, es funktioniert aber nicht. In einem zweiten Fenster sieht man mit `ps -a`, daß zwei clients laufen, aber kein Server. Es kann nur eine Kleinigkeit sein (der letzte Bug, wie immer). Beheben Sie den Fehler!

Stevens beurteilt die Nützlichkeit der Message Queues kritisch. Insbesondere sind sie heute nicht schneller als andere IPC-Konstrukte, z.B. **stream pipes**. Wir schließen deshalb hier ihre Behandlung ab und übergehen im folgenden auch Semaphore und Shared Memory.

7 Semaphore

Wir wollen auf Semaphore nicht eingehen. Diese Seite(n) dienen nur der Zusammenfassung und als eine Art Platzhalter, damit die Numerierung synchron zu dem Buch von Gray erfolgt.

Die von E.W. Dijkstra 1965 eingeführten Prozeßsynchronisierungsmechanismen dienen der Zugriffskontrolle auf kritische Abschnitte. Eine durch ein Semaphor *S* geschützter Abschnitt wird betreten, indem die **P(S)**-Operation aufgerufen wird, die einen in *S* vorhandenen Zähler testet und, falls er größer 0 ist, herunterzählt, andernfalls den aufrufenden Prozeß in den Wartezustand versetzt und ihn in eine Warteschlange des Semaphors *S* einreiht.

Prozesse, die den kritischen Abschnitt verlassen, rufen **V(S)** auf. Dadurch wird der Zähler hochgezählt, bzw. es wird ohne Hochzählen ein wartender Prozeß aus der Warteschlange von *S* freigesetzt und in den kritischen Abschnitt gelassen.

Unter UNIX gibt es für Semaphore eine `semget` Operation, mit der ein Semaphor angelegt wird, bzw. mit dem man sich Zugang zum Semaphor verschafft (ähnlich einem `creat`). Mit `semctl` kann man Semaphore initialisieren, Werte abfragen und Semaphore entfernen. Der Wert des Semaphores ist im übrigen ein Integer der im BS-Kern gehalten wird. Zuletzt kann man mit `semop` die oben genannten *P* und *V* Operationen ausführen, die *obtain* (herunterzählen), *release* (hochzählen) und *test* (abfragen auf 0) als Flags erhalten können und atomar arbeiten.

Die Semaphorprogrammierung sieht ferner vor, daß sich alle Operationen auf Mengen von Semaphoren beziehen, wodurch die Kontrolle von Betriebsmittelgruppen erleichtert wird.

Semaphore unter UNIX setzen voraus, daß nur gutwillige Prozesse beteiligt sind. Den eigentlichen Zugriff auf eine Ressource schützen sie nicht.

Gray gibt ein übersichtliches Erzeuger-Verbraucher Beispiel (Programm 7.4) an.

Neben der sehr komplizierten Syntax gilt nach Stevens das Anlegen und Initialisieren der Semaphorwerte in zwei getrennten Operationen (`semget` und `semctl`) als Entwurfsfehler. Zusätzlich kann es bei einem Prozeßausstieg vorkommen, daß Semaphore übrigbleiben, da diese nicht automatisch abgeräumt werden.

Da letzteres nicht bei Satzsperren passieren kann, empfiehlt Stevens **record locking** als einfachere und genauso schnelle Alternative zu Semaphoren.

8 Shared Memory

Die gemeinsame Nutzung von Hauptspeicher ist sicherlich eine der schnellsten Formen der Prozeßkommunikation innerhalb einer Rechenanlage. Mittels der UNIX *shared memory* IPC ist dies möglich, allerdings wird keine explizite Synchronisation zur Verfügung gestellt, d.h. man muß die Kontrolle für den Zugriff auf den gemeinsamen Speicher noch über Semaphore oder Satzsperren herstellen.

Anlegen, kontrollieren und nutzen erfolgt ähnlich zu den vorherigen Mechanismen; es gibt `shmget` zum Anlegen, bzw. Verbinden zu einem bereits existierenden shared memory. Der Aufruf liefert einen *shared memory identifier* zurück. Ferner gibt es wieder ein `shmctl` zum Abfragen und Setzen gewisser Werte, sowie zum Löschen.

Der globale Speicher wird vom Anwender dann in die eigene Prozeßumgebung mittels `shmat` (*shared memory attach*) eingebunden, bzw. durch `shmdt` (~ *detach*) wieder aus der eigenen Speicherbelegung gestrichen. Einbinden heißt, daß der Bereich in den eigenen virtuellen Prozeßadressraum zwischen Heap und Stack gelegt wird, wobei „ablegen“ ein Setzen von Seitenreferenzen aus der prozeßeigenen Seitentabelle in eine globale Seitentabelle, bzw. auf Kacheladressen, ist.

Der zurückgelieferte HS-Pointer gibt die Startadresse an, ab der beliebige Datenstrukturen definiert sein können, wodurch beliebiger Zugriff auf die dort gespeicherten Daten möglich ist. Mit *detach* wird die Bindung wieder gelöst, wobei der Kernel einen Zähler herunterzählt, der angibt, wieviele Prozesse einen shared memory Bereich nutzen. Ist der Zähler 0, kann er durch `shmctl` abgeräumt werden.

Die interessanteste Variante ist die, bei der durch den `mmap`-Aufruf der gemeinsame Speicher auf eine (persistente) Datei abgebildet wird. Sog. *memory mapped I/O* stellt die schnellste Form der Ein-/Ausgabe dar und erlaubt mit dauerhaft gespeicherten Daten (einer Datenbasis) so umzugehen, wie wenn sie als normale HS-Daten in Arrays und Records abgelegt wäre. Die aus manchen objekt-orientierten Datenbanksystemen bekannte Technik des sog. *Pointer-Swizzlings* (Wandlung von HS-Zeigern in permanente DB-Zeiger und umgekehrt) beruht auf den `mmap`-Fähigkeiten. Änderungen der Werte müssen allerdings explizit durchgeschrieben werden. Eine copy-on-write Semantik steht zur Verfügung.

Interessant ist wieder, daß die Zuordnung über die Seitentabelle der MMU erfolgt. Insbesondere kann `mmap` und `mprotect` für Bereiche erfolgen, die noch nicht angelegt wurden. Über ein Schutzbit kann ein Zugriff auf eine solche Seite in dem „gemappeten“ Bereich abgefangen werden (Segment Violation -`SIGSEV` Signal). Gewöhnlich erfolgt dann durch das Anwendungsprogramm die Aufbereitung der Seite, z.B. das Wandeln von Pointern.

9 Remote Procedure Calls

9.1 Einleitung

Die bisherigen Vorschläge zur Prozeßsynchronisation und ~kommunikation setzten einen gemeinsamen Rechner voraus. Häufig sollen Anwendungen aber innerhalb eines lokalen oder weitgespannten Netzes laufen.

Eine hierfür häufig gebrauchte Methode sind die *remote procedure calls* (RPC). Sie setzen auf der Überlegung auf, daß der normale Prozeduraufruf die klassische Form der synchronen Kommunikation darstellt: der Aufrufer gibt einen Auftrag an einen Unterprozeß ab und wartet auf die Rückkehr.

Die aufrufende Prozedur heißt deshalb auch **client stub** und muß insbesondere den Adressaten nennen. Die aufgerufene Prozedur heißt **server stub**. In den Aufrufen verbergen sich die Details der verwendeten Kommunikationsdienste (Übertragungsprotokolle), die auf niedrigeren Schichten definiert sind.

Da die Details der Übertragung trickreich sind, verwendet man einen Protokoll-Compiler, z.B. `rpcgen` und gibt diesem ein in einer C-artigen Sprache - der sog. RPC Sprache - geschriebenes Programm, das insbesondere die zu übertragenden und als Ergebnis erwarteten Parametertypen enthält.

Kritische Punkte sind:

- Regeln über das Verhalten bei Verlust der Botschaft oder Serverabsturz; üblich sind time-outs.

- Regeln über sichere Kommunikation, die Paketverlust und Paketverdopplung registriert.
- Behandlung von Pointern, Speicheranforderungen, Datentypen; wegen unterschiedlicher Rechnerarchitekturen ist deshalb eine Übersetzung in ein linearisiertes Format, gewöhnlich **XDR** (e**X**ternal **D**ata **R**epresentation), bzw. die Rückübersetzung aus XDR, nötig.

Intern wird für RPC gewöhnlich eine Kommunikation verwandt, die auf *Sockets* (vgl. Kapitel 10) beruht. Die verfügbaren Dienste kann man sich mit `rpcinfo` und einigen Optionen (AIX: `rpcinfo -s` *host*) ansehen. Den Namen des eigenen Rechners bekommt man mit `hostname` heraus.

9.2 Die Ausführung von entfernten Kommandos auf Systemebene

Ehe man sich die Feinheiten des RPC-Mechanismus ansieht, lohnt es, die Ausführung eigener Kommandos auf fremden Rechnern zu studieren. Dies wird in UNIX durch das `rsh`-Kommando (**r**emote **sh**ell command) ermöglicht. Die Syntax lautet

```
rsh host [ -l login ] [ -n ] command
```

Durch einen Aufruf auf Shellebene des eigenen (lokalen) Rechners wird auf dem entfernten Rechner *host* das angegebene Kommando ausgeführt, wobei die Standardeingabe auf die Aufrufzeile beschränkt ist. Nur wenn das Kommando fehlt, wird `rsh` ersetzt durch `rlogin` (also das Anmelden auf dem entfernten Rechner) und auf dem entfernten Rechner kommt die Standardeingabe der `login`-Shell von der lokalen Standardeingabe. In jedem Fall (`rsh` mit oder ohne Kommando) wird die Standard- und Standardfehlerausgabe vom entfernten auf den eigenen lokalen Rechner geleitet.

Durch die Option `-n` erreicht man, daß die Eingabe sofort von `/dev/null` kommt, da entfernt ausgeführte Kommandos wie batch-Jobs nicht interaktiv arbeiten können. Der Aufruf

```
$rsh haensel ps -e
...
```

```
 22444 - 7:45 oracle
 22954 - 12:52 oracle
 23390 - 0:00 rshd
 24234 - 0:34 dtgreet
 26978 - 0:00 e45Server
 27354 - 0:42 nmbd
 27744 - 0:00 ps
 28374 - 0:00 smbd
 28650 - 0:00 mathlm
 29112 - 0:00 dtexec
 29594 - 0:00 e45Server
$
```

zeigt u.a. `ps` und `rshd` (den remote shell daemon) in der Ausführung auf unserem Server `haensel`.

Natürlich bedeutet die Ausführung fremder Aufrufe auf einem eigenen Rechner ein Sicherheitsrisiko. Deshalb wird der Zugriff auf Dateien durch die Definition zugelassener fremder Rechner und Benutzer geregelt. Üblicherweise sind die Zulassungen und Sperren in der `/etc/hosts.equiv` Datei und der `.rhosts` Datei festgehalten.

Vorsicht ist auch geboten, was Umlenkungen angeht. Die lokale Shell interpretiert Umlenkmetazeichen als lokal. Möchte man auf dem entfernten Rechner in eine Datei umlenken, dann muß die Umlenkung mit Anführungszeichen geschützt werden:

```
rsh df “>“ remotefile
```

9.3 Entfernte Ausführung eines Kommandos in einem C-Programm

Die Bibliotheksfunktion `rexec` erlaubt es, ein Systemkommando aus einem C-Programm heraus auf einem fremden Rechner zur Ausführung zu bringen. Dies ist ähnlich zum Systemaufruf `system`, der ein Shellkommando in einem C-Programm ausführen läßt (vgl. UNIX-Kurs, dort Einsatz für ein Shell-Skript, damit man dem das suid-Bit verpassen kann, was nur für ausführbaren Maschinencode geht).

Die Syntax ist der folgenden Tabelle zu entnehmen. Der Aufruf verlangt 6 Argumente. Das erste ist der Name des entfernten Rechners. Das Argu-

ment wird an `gethostbyname` weitergeleitet, das die Zugriffsregelung vornimmt (siehe Kapitel 10). Laut Manual wird in der `/etc/hosts` oder der `/etc/resolv.config` Datei nachgesehen.

<table>
<tr><td>Include File(s)</td><td colspan="2"><netdb.h></td><td>Manual</td><td>3N</td></tr>
<tr><td>Summary</td><td colspan="4">int
rexec(char **ahost,
unsigned short inport,,
const char *user,
const char *passwd,
const char *cmd,
int *fd2p);</td></tr>
<tr><td rowspan="2">Return</td><td>Success</td><td>Failure</td><td colspan="2">Sets <code>errno</code></td></tr>
<tr><td>Ein stream socket Datei-deskriptor</td><td>-1</td><td colspan="2"></td></tr>
</table>

Tab. 9–1 *rexec*

Das zweite Argument ist die Portnummer für die Kommunikation. Häufig wird Port 512 benutzt, der für *remote execution* zuständig ist und TCP verwendet. Laut `man rexec` läßt sich der Port mit dem Bibliotheksaufruf

```
getservbyname("exec","tcp")
```

ermitteln.

Die folgenden zwei Argumente sind der Benutzername und sein Passwort, ggf. sind die Werte NULL, worauf laut Gray in der Datei `.netrc` für das Passwort des Benutzers *user* auf dem Rechner *ahost* nachgesehen wird. Gibt es eine solche Datei nicht, wird der Aufrufer zur Eingabe seines Passwort aufgefordert.

Nach dem aufzurufenden Kommando folgt ein Zeiger auf eine ganze Zahl, die für Werte gleich 0 angibt, daß die Standardfehlerausgabe in die Standardausgabe gehen soll und daß keine Signale an den entfernten Prozeß gesendet werden können.

Bei Werten ≠ 0 wird durch den Wert ein Fehlerausgabekanal angegeben und die Prozeßgruppe akzeptiert in der umgekehrten Richtung einzelne Bytes als Signalnummern mit denen sich ggf. das Verhalten des Kommandos beeinflussen läßt.

Der erfolgreiche Aufruf liefert eine *stream socket* Verbindung zurück, mit der die Standardein- und Standardausgabe des entfernten Prozesses auf die lokale Ein-/Ausgabe abgebildet wird.

In Gray folgt ein kleines Programm 9.1, das von der Kommandozeile aus mittels Rechnername und Kommando — ähnlich zu `rsh`, aber mittels `rexec` — das Kommando auf dem entfernten Rechner ausführt.

```
/* using rexec */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
void main (int argc, char *argv[])
{
 int fd, count;
 char buffer[BUFSIZ], *command, *host;
 if (argc != 3){
  fprintf(stderr, “Usage %s host command\n“, argv[0]);
  exit(1);
 }
 host = argv[1];
 command = argv[2];
 if ((fd = rexec(&host, htons(512), 0, 0 , command, 0))\
               == -1) {
  fprintf(stderr, “rexec failed\n“);
  exit(2);
 }
 while ((count = read(fd, buffer, BUFSIZ)) > 0)
  fwrite(buffer, count, 1, stdout);
}
```

Die Beispielausgabe lautet:

```
wegner@elsie(chpt9)$ ./p91a haensel who
Name (haensel.db.informatik.uni-kassel.de:wegner):
wegner
Password (haensel.db.informatik.uni-kassel.de:wegner):
ueb17 lft0 Jun 09 09:53
injt pts/4 Jun 09 09:23 (snow-white.db.in)
ueb17 pts/5 Jun 09 09:53
ueb17 pts/6 Jun 09 10:05
ueb17 pts/9 Jun 09 10:21 (:0.0)
ueb17 pts/10 Jun 09 10:33 (:0.0)
wilke pts/11 Jun 09 11:07 (rapunzel.db.info)
wegner@elsie(chpt9)$
```

Hiermit schließen wir die vorläufigen Schritte zur entfernten Kommandoausführung ab. Das UNIX-Handbuch warnt im übrigen davor, `rexec` in einer Multithread-Umgebung einzusetzen.

9.4 Wandlung eines lokalen Funktionsaufrufs in eine Remote Procedure

Wir wandeln einen Aufruf von `printf` zur Ausgabe von „`Hello, world`“ so um, daß er auf einem entfernten Rechner zur Ausführung kommt und an den lokalen Aufrufer sein Funktionsergebnis (Gray: Ja, auch `printf` liefert ein Ergebnis ab!) zurücksendet.

> **Hinweis:** im Kapitel 9 gehen die Sources von Gray ziemlich durcheinander. Obiges Programm fehlte und wurde von Stephan als `p91a.c` abgeschrieben und eingefügt. Das folgende Programm heißt im Buch 9.2, firmiert aber unter `p91.c` in den Quellen. In einem Unterverzeichnis `hello` finden sich dann die für RPC gewandelten Programme.

```
/* A C program with a local function */
#include <stdio.h>
void
main(void){
 int print_hello(void);
 if (print_hello())
 printf(„Mission accomplished\n“);
 else
 printf(„Unable to display message“);
}
```

```
int
print_hello(void) {
 return printf(„Hello, world.\n“);
}
```

Lokal aufgerufen gibt `p91` die erwartete Ausgabe.

```
wegner@elsie(chpt9)$ ./p91
Hello, world.
Mission accomplished
wegner@elsie(chpt9)$
```

Um das Programm von `rpcgen` so umwandeln zu lassen, daß es auf einem anderen Rechner läuft und Ein-/Ausgabe über das XDR-Format serialisiert wird, brauchen wir zuerst eine in der RPC-Sprache geschriebene Protokolldatei, die traditionell mit dem Suffix `.x` bezeichnet wird, hier also als `hello.x`.

```
/*
 * This is the protocol definition file written in RPC
 * language that will be passed to protocol generator
 * rpcgen. Every remote procedure is part of a remote
 * program. Each procedure has a name and number.
 * A version number is also supplied so different
versions
 * of the same procedure may be generated.
 */

program DISPLAY_PRG {
 version DISPLAY_VER {
 int print_hello( void ) = 1;
 } = 1;
} = 0x20000001;
```

Die RPC-Sprache ist eine Mischung aus C und Pascal. In Gray wird die Syntax im Anhang C beschrieben, wir verzichten hier auf die Angabe.

Im Einzelnen wird in der ersten Zeile mit `program DISPLAY_PRG {...}` in Pascal-Manier ein Programmname definiert, der anders lauten kann als die Datei, in der das Programm gespeichert ist. Zeile 2 definiert eine Versionsnummer, hier in Zeile 4 als 1 angegeben. Innerhalb des Versionsblocks erfolgt in Zeile 3 die Angabe der Funktion, die wiederum mit einer Nummer versehen ist (Funktion Nr. 1).

Zuletzt folgt eine achtstellige Hexadezimalzahl, die sog. RPC Programmnummer, die einer von SUN vorgegebenen Systematik folgt und der Registrierung von Programmen dient. Nummern im Bereich `20 00 00 00` bis `3F FF FF FF` sind für benutzerdefinierte RPC Programme vorgesehen, daher unsere Wahl.

Durch Aufruf von

```
rpcgen -C hello.x
```

werden drei Zieldateien erzeugt:

- eine header Datei, hier `hello.h`
- einen client stub, hier `hello_clnt.c`
- einen server stub, hier `hello_svc.c`

Die Option `-c` verlangt nach ANSI-C Ausgabe. Andere Optionen sind laut AIX und verschieden zu den Angaben von [Gray] und SUN

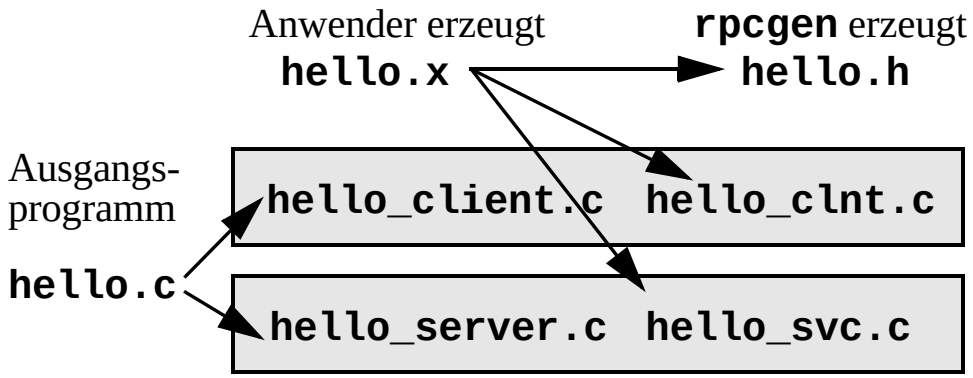
```
usage:
 rpcgen infile
 rpcgen [-c | -h | -l | -m] [-o outfile] [infile]
 rpcgen [-s udp|tcp]* [-o outfile] [infile]
Flags
-cCompiles into XDR routines.
-hCompiles into C-data definitions (a header file).
-lCompiles into client-side stubs.
-mCompiles into server-side stubs, but does not generate
a main routine. This option is useful for doing call-back
routines and for writing a main routine to do
initialization.
-o OutputFileSpecifies the name of the output file. If
none is specified, standard output is used.
-s TransportCompiles into server-side stubs, using given
transport. The supported transports are udp and tcp.
This flag can be run more than once to compile a server
that serves multiple transports.
```

Die Ausgabe `hello.h` hat das folgende Aussehen.

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
```

```
 */
#ifndef _HELLO_H_RPCGEN
#define _HELLO_H_RPCGEN
#include <rpc/rpc.h>
#ifdef __cplusplus
extern „C“ {
#endif
#define DISPLAY_PRG ((unsigned long)(0x20000001))
#define DISPLAY_VER ((unsigned long)(1))
#if defined(__STDC__) || defined(__cplusplus)
#define print_hello ((unsigned long)(1))
extern int * print_hello_1(void *, CLIENT *);
extern int * print_hello_1_svc(void *, struct svc_req
*);
extern int display_prg_1_freeresult(SVCXPRT *,
xdrproc_t, caddr_t);
#else /* K&R C */
#define print_hello ((unsigned long)(1))
extern int * print_hello_1();
extern int * print_hello_1_svc();
extern int display_prg_1_freeresult();
#endif /* K&R C */
#ifdef __cplusplus
}
#endif
#endif /* !_HELLO_H_RPCGEN */
```

Insgesamt kommen wir jetzt auf 6 Programme, die in der folgenden Beziehung stehen.



Aus dem ursprünglichen „Hallo, Welt“ Programm müssen wir zwei Versionen, ein Client und einen Server produzieren, die wir durch Einbinden der `hello.h` Headerdatei über die Existenz der externen Prozeduren `print_hello_1`, bzw. `print_hello_1_svc` informieren. Diese

wurden von `rpcgen` erzeugt und müssen auf dem Client-Rechner und dem Serverrechner zur Verfügung stehen. Die von `rpcgen` erzeugte Komponente für den Client sieht wie folgt aus:

```
wegner@elsie(hello)$ less hello_clnt.c
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include „hello.h“

/* Default timeout can be changed using clnt_control()
*/
static struct timeval TIMEOUT = { 25, 0 };

int *
print_hello_1(void *argp, CLIENT *clnt)
{
 static int clnt_res;

 memset((char *)&clnt_res, 0, sizeof (clnt_res));
 if (clnt_call(clnt, print_hello,
  (xdrproc_t) xdr_void, (caddr_t) argp,
  (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
  TIMEOUT) != RPC_SUCCESS) {
    return (NULL);
 }
 return (&clnt_res);
}
```

Das vom Anwender selbst zu modifizierende `hello_client.c` Programm sieht wie folgt aus und enthält als wesentliche Komponenten den `clnt_create` (Bibliotheksfunktion für und den `print_hello_1` Aufruf.

```
/*
 * The CLIENT program: hello_client.c
 * This will be the client code executed by the local
client process.
 */
#include <stdio.h>
#include <stdlib.h>
#include „hello.h“ /*** Generated by rpcgen from hello.x
*/

void
main(int argc, char *argv[]) {
 CLIENT *client;
 int *return_value, filler;
 char *server;
/*
 * We must specify a host on which to run. We will get
the
 * host name from the command line as argument 1.
 */
 if (argc != 2) {
  fprintf(stderr, „Usage: %s host_name\n“, *argv);
  exit(1);
 }
 server = argv[1];
/*
 * Generate the client handle to call the server
 */
 if ((client = clnt_create(server, DISPLAY_PRG,
    DISPLAY_VER, „visible“)) == (CLIENT *) NULL) {
      clnt_pcreateerror(server);
      exit(2);
 }
 return_value = print_hello_1((void *) &filler, client);
 if (*return_value)
  printf(„Mission accomplished\n“);
 else
  printf(„Unable to display message\n“);
}
```

In der von Gray gewählten Ausprägung erhält es den Servernamen als Kommandozeilenargument. Die Verbindung zu den von `rpcgen` erzeugten Dateien `hello_clnt.c` wird über den Compileraufruf zur Erzeugung des lauffähigen Programms, hier einfach `client`, hergestellt. Der Compileraufruf lautet z.B.

```
CC hello_client.c hello_clnt.c -o client -lnsl
```

und enthält die `-lnsl` Option zur Verwendung der Networking Bibliothek `libnsl`.

Das Serverprogramm ist dagegen viel einfacher:

```
/*
 * The SERVER program: hello_server.c
 * This will be the server code executed by the
 * „remote“ process
 */

#include <stdio.h>
#include „hello.h“ /** generated by rpcgen from hello.x
*/

int *
print_hello_1_svc(void * filler, struct svc_req * req)
{
 static int ok;
 ok = printf(„Hello, world.\n“);
 return (&ok);
}
```

Zum ersten Testen können wir Client und Server auf der selben Maschine laufen lassen. Den Server müssen wir nicht als Hintergrundprozeß starten, allerdings anschließend wieder killen. Mit `ps -ef` kann man feststellen, daß er existiert. Sein Vaterprozeß ist `init` mit PID 1 und er besitzt keine Kontrollkonsole.

Ruft man jetzt den Client mit

```
./client elsie
Mission accomplished
```

auf, kommt nur die Bestätigungsbotschaft des Client. Die „`Hello, World`“ Ausgabe des Servers erscheint nicht. Das liegt daran, daß der Server kein Ausgabeterminal besitzt und daher die Ausgabe in den großen Biteimer umlenkt.

Übung 9–1

Man rufe mit dem Kommando `rsh` auf einem entfernten Rechner `./client` *anderer-rechner* auf, damit Client und Server tatsächlich auf unterschiedlichen Rechnern laufen.

Übergangen haben wir die in `hello_client.c` deklarierte `CLIENT`-Struktur, die in `<rpc/clnt.h>` definiert ist. Ferner wollen wir nicht auf die Bibliotheksfunktion `clnt_create` und Fehlerroutine `clnt_pcreateerror` eingehen. Erstere enthält in Gray für den Transporttyp den Eintrag „`visible`“, der eine Suche in `/etc/netconfig` nach einem Eintrag mit Flag „`v`“ auslöst - was es unter AIX nicht gibt! Im wesentlichen geht es hier um die Wahl des Protokolls: UDP oder TCP.

Wie man sieht, ist die ganze RPC-Geschichte aufwendig. In Gray folgen weitere Abschnitte zu RPC-Debugging und Gebrauch des `rcpgen` Werkzeugs, sowie eine Besprechung des `rpc_broadcast` Aufrufs, um Dienstleistungen zu suchen. Wir übergehen diesen Teil um direkt zu den Sockets zu kommen.

10 Sockets

10.1 Einführung

Einer der großen Vorteile von UNIX ist das Konzept der unstrukturierten Dateien, hinter denen sich auch Geräte, etwa der Bildschirm oder die Tastatur, verbergen können und die auch durch lesende, bzw. im Fall der Eingabe, andere schreibende Programme ersetzt werden können, die einen Bytestrom liefern oder abnehmen.

Die einheitliche Schnittstelle ist der Dateideskriptor. Er ist das Ziel der `read` und `write` Systemaufrufe.

Mit den Pipes (unbenamt und FIFO's) ist dies am elegantesten für die Prozeßkommunikation übernommen worden, allerdings beschränkt auf verwandte Prozesse, bzw. solche, die Zugriff auf ein gemeinsames Dateisystem haben. Ferner kann es zu Blockaden und Verklemmungen kommen.

Bei Message Queues, Semaphoren und globalem Speicher ist die Einheitlichkeit verschüttet worden, sie arbeiten nicht mit Dateideskriptoren und verwenden spezialisierte Aufrufe für die Kommunikation.

RPC hätte vom Konzept her die Chance gehabt, wieder Einheitlichkeit herzustellen. Durch die Verpackung der Routinen, die Hilfsdateien, den Namensservice, usw. kommt es allerdings zu einer solchen Verkomplizierung, daß die ursprüngliche Idee des transparenten Prozeduraufrufs nicht mehr erkennbar wird.

Mit den **Sockets**, die auch als **Berkeley Sockets** bekannt sind und aus den frühen achzigern stammen, gibt es aber ein solches Konzept, das dem Dateigebrauch stark angenähert ist, jedoch Kommunikation über Rechnergrenzen hinweg erlaubt.

Nachteil hier ist die Abhängigkeit vom Transportprotokoll (was das **TLI** - Transport Level Interface - von AT&T aus den Mitte der achziger Jahren vermeidet) und die Tatsache, daß sie in Zusammenhang mit *threads* nur mit Vorsicht zu gebrauchen sind.

Sockets sind prinzipiell Datenstrukturen, an denen Verbindungskanäle zum Senden und Empfangen von Daten zwischen verschiedenen Prozessen angebunden sind. Sockets müssen vom Server und vom Client angelegt werden, wobei der Client die Verbindung zum Server (*host name* und *port*) benennen muß. Ist dieser Verbindungsvorgang (`connect`, `listen`, ...) hergestellt, kann mit `read` und `write` darauf empfangen und gesendet werden.

10.2 Grundlagen der Kommunikation

Um Sockets zu verstehen, muß man gewisse Grundkenntnisse der Netzwerkkommunikation und deren Terminologie haben.

10.2.1 Netzwerkadressen

Jeder Rechner in einem Netzwerk hat wenigstens zwei eindeutige Adressen. Zum einen die 48-bit Ethernetadresse, die vom Hersteller der Ethernetkarte bestimmt wird.

Man gibt sie als Hexadezimalzahl mit einem Punkt zwischen jeweils zwei Hexziffern an. Die Datei `/etc/ethers` gibt häufig die vertretenen Ethernetadressen eines Systems an.

Eine ähnliche Datei bei uns sieht wie folgt (partiell) aus:

```
 rapunzel.db.informatik.uni-kassel.de (141.51.180.4) at
8:0:5a:1:b7:ef [ethernet]
```

```
 hrz-ro-avz2.hrz.uni-kassel.de (141.51.168.1) at
0:0:d:c0:28:e0 [ethernet]

 elsie.db.informatik.uni-kassel.de (141.51.180.6) at
8:0:5a:1:b9:60 [ethernet]

 inf-pc4.db.informatik.uni-kassel.de (141.51.180.8) at
0:20:af:2d:ea:88 [ethernet]

...

 gretel.db.informatik.uni-kassel.de (141.51.180.2) at
8:0:5a:c7:1e:93 [ethernet]
```

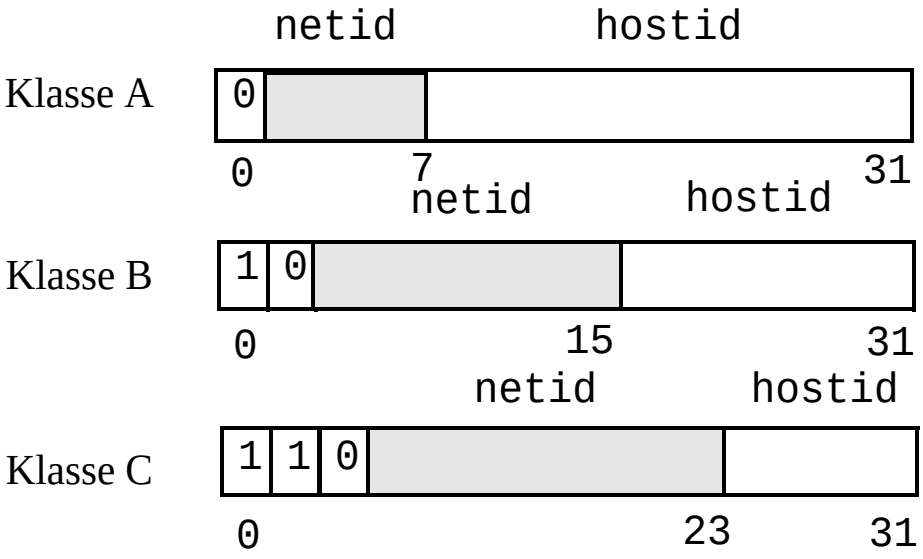
Die zweite eindeutige Adresse ist die 32-bit Internet (IP) Adresse. Internet Adressen werden (wurden?) vom NIC (Internet Information Center) vergeben, daß eine Registrierung dieser Adressen betreibt. Sie werden in vier je 8-bit Gruppen (dezimal 0 - 255) eingeteilt und in Dezimalschreibweise, mit einem Punkt zwischen den Gruppen, angegeben.

Die Adresse zerfällt in einen Netzwerkteil und einen lokalen Teil. Der Netzwerkteil, `netid`, nimmt den linken Teil der Adresse ein und kann in drei Klassen eingeteilt werden. Die Zugehörigkeit zu einer Klasse läßt sich aus den führenden bits ableiten:

- Klasse A mit bit 0 gesetzt auf 0
- Klasse B mit den ersten zwei Bits 10
- Klasse C mit den ersten drei Bits 110.

Die Zuordnung zu einer Klasse nimmt das NIC vor und gibt damit die Netzwerkzugehörigkeit (association) an.

Den lokalen Teil bestimmt der örtliche Netzadministrator, der damit die genaue Rechnernummer des lokalen Servers oder einer Workstation angibt. Aus den Klassen bestimmt sich, wie groß dieser Kreis an vergebbaren Nummern ist.



Es liegt im Verantwortungsbereich des Systemverwalters, Internet-Adressen auf Ethernet-Adressen abzubilden. Diesen Prozess nennt man Adressauflösung (**address resolution**), das Protokoll dazu ARP (address resolution protocoll). Mit
`arp -a` läßt sich eine solche Zuordnung auflisten.

Wenn wir im folgenden in Zusammenhang mit Sockets auf Adressen zu sprechen kommen, meinen wir immer IP-Adressen.

10.2.2 Domänen - Netzwerk und Kommunikation

Die IP-Adresse ist zwar eine präzise und platzsparende Methode, Rechner zu adressieren. Für den menschlichen Gebrauch sind aber sprechende Namen besser, die mit Bereichsnamen (**domain names**) arbeiten. Diese werden mit Punkt getrennt zu einem Bereichsnamen zusammengesetzt. Bekannt sind die Domänenbezeichner `edu`, `gov`, `mil`, `com` und die Länderbezeichner `de`, `fi`, `ca`, usw. Von rechts nach links gelesen werden die Namen immer spezifischer, z.B.
`gretel.db.informatik.fb17.uni-kassel.de`. Ganz links steht im vollen Namen dann der Rechnerbezeichner.

Auf allen Rechnern läuft in der Regel Software, die eine Umsetzung der domain names in IP-Adressen vornimmt, z.B. DNS (domain name service) oder NIS+ (network information service).

Übung 10–1

Schauen Sie sich `man nslookup` an und rufen Sie `nslookup` z.B. für `uni-kassel.de` auf.

Der Begriff Domäne taucht auch bei Sockets wieder auf, da wir für ein Socket seine Kommunikationsdomäne festlegen.

Zwei Arten werden hier besprochen, die

- **UNIX domain** - Sockets, die nur innerhalb eines Rechners gebraucht werden können und ihre Kommunikation über Pfadnamen (Dateinamen) herstellen; aufgrund ihrer Lokalität gestaltet sich die Fehlersuche mit ihnen leichter.
- **Internet domain** - Sockets, die beliebige Prozesse auf fremden Rechnern verbinden können.

Wenn wir also von domains sprechen, werden wir vorzugsweise die Kommunikationsdomäne von Sockets, nicht Internet Bereichsnamen, meinen.

10.2.3 Protokollfamilien

Protokolle legen Regeln, Kodierungen, technische Details, usw. für Datenkommunikation fest. Wegen ihrer Komplexität sind sie in Schichten aufgeteilt.

Sehr bekannt ist das 7-Schichten ISO-OSI (Open Systems Interconnect) Referenzmodell.

	Ebene	Funktionalität
obere Schichten	Anwendung (application)	Prozesse haben Zugriff auf Prozeßkommunikationsmöglichkeiten
	Präsentation (presentation)	Datenkonvertierung, Textkompression, Verschlüsselung
	Sitzung (session)	Synchronisation des Prozeßdialogs
Protokollfamilie {	Transport	Transportservice, Sicherheitsniveau, Datenfluß
	Netzwerk	Routing, Aufrechterhaltung der Verbindung
	Datenverbindung (data link)	stellt fehlerfreie Übertragung sicher
untere S.	Physische Schicht	hardwarenahe Übertragung von Signalen

Tab. 10–1 *ISO-OSI Schichtenmodell*

Einige ältere, aber weitverbreitete Protokolle, wie z.B. TCP, stammen aus der Zeit vor OSI und lassen sich nur schlecht auf das Referenzmodell abbilden.

Die als Protokollfamilie zusammengefaßten zwei Schichten Transport und Network (Netzwerk) beinhalten die wichtigen Aufgaben der Adreßauflösung, Fehlerbehandlung, Routenwahl, Flußkontrolle. Hierfür gibt es eine Reihe von Familien:

- **SNA** - IBM’s System Network Architecture
- **UUCP** - UNIX-to-UNIX copy
- **XNS** - Xerox Network System
- **NETBIOS** - IBM’s Network Basic Input/Output System
- **TCP/IP** - DARPA Internet

Hier werden wir uns auf die TCP/IP Protokollfamilie stützen, die sich aus den folgenden Komponenten zusammensetzt.

- **TCP** - Transmission Control Protocol. TCP bietet verlustfreie Übertragung, ist voll-duplex und verbindungsorientiert. Die Daten werden als Bytestrom übertragen.
- **IP** - Internet Protocol. Bietet Lieferung von Datenpaketen. Wird von TCP, UDP und ICMP aufgerufen.
- **ARP/RARP** - Adress/Reverse Adress Resolution Protocol. Internet/Hardware Adressumsetzung
- **UDP** - User Datagram Protocol. Nicht verlustfreie, voll-duplex, verbindungslose Übertragung von Datenpaketen.
- **ICMP** - Internet Control Message Protocol. Fehlerbehandlung und Datenflußkontrolle.

Innerhalb der TCP/IP Familie konzentrieren wir uns auf TCP und UDP. Von uns angelegte Sockets bekommen als Protokollfamilie entweder PF_UNIX (UNIX - eigentlich kein echtes Komm.-protokoll) oder PF_INET (TCP/IP).

10.2.4 Sockettypen

Es gibt zwei grundlegende Arten der Datenübertragung. Einerseits die verbindungsorientierte Art, die die Daten als einen Bytestrom sieht (stream format), oder die verbindungslose Art, bei der Daten in eine Folge kleiner Pakete, jeweils mit Adresse, Inhalt, Prüfsummenabschluß etc., verpackt werden (datagram format).

> **Hinweis**: So klar ist die Trennung leider auch wieder nicht. Auf Paketbasis kann z.B. eine virtuelle Leitung hergestellt werden, man vergleiche etwa die ATM-Diskussion z.Zt.

Damit zwei Sockets kommunizieren können, müssen sie sich auf die selbe Übertragungsart geeinigt haben. Angeboten für den Anwender werden

- **Stream sockets**. Diese bieten zuverlässige Kommunikation ohne Duplizierung oder Verlust von Daten, ohne Vertauschung von Reihenfolgen von Daten. Die Verbindung ist bi-direktional und leitungsorientiert (die Analogie zum Telefonieren ist erlaubt), d.h. die Information über Adressaten und Absender, Verbindungsweg, usw. wird während der gesamten Übertragung aufrechterhalten.
- **Datagram sockets**. Diese sind potentiell unzuverlässig. Pakete können in anderer Reihenfolge beim Empfänger ankommen, als sie abgesandt wurden. Pakete können fehlen oder dupliziert worden sein. Die Route kann potentiell für jedes Paket wechseln. Man spricht von verbindungsloser Kommunikation (Analogie zum Postkartenversand).

10.3 IPC mit `socketpair`

Zum Aufwärmen erzeugen wir ein Paar von UNIX domain sockets. Dies geht mit dem `socketpair` Netzwerkaufruf, der ein Paar von Sockets anlegt.

<table>
<tr><td>Include File(s)</td><td colspan="2"><sys/types.h>
<sys/socket.h></td><td>Manual</td><td>3N</td></tr>
<tr><td>Summary</td><td colspan="4">int socketpair (int family,
int type,
int protocol,
int sv[2]);</td></tr>
<tr><td rowspan="2">Return</td><td>Success</td><td>Failure</td><td colspan="2">Sets <code>errno</code></td></tr>
<tr><td>0 und zwei offene Socketdeskriptoren</td><td>-1</td><td colspan="2">Yes</td></tr>
</table>

Tab. 10–2 *socketpair Netzwerkaufruf*

Der Aufruf verlangt vier Argumente. Das erste bestimmt die Protokollfamilie, auch Adreßfamilie genannt. Die symbolischen Konstanten dafür finden sich im `socket.h`, wobei dort sowohl die PF-Konstanten als

auch die mit gleichen Werten belegten AF-Konstanten genannt sind. Verwenden sollte man PF-Konstanten.

```
#define AF_UNSPEC 0 /* unspecified */
#define AF_UNIX 1 /* local to host (pipes, portals) */
#define AF_INET 2 /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK 3 /* arpanet imp addresses */
#define AF_PUP 4 /* pup protocols: e.g. BSP */
#define AF_CHAOS 5 /* mit CHAOS protocols */
#define AF_NS 6 /* XEROX NS protocols */
#define AF_ISO 7 /* ISO protocols */
#define AF_OSI AF_ISO
#define AF_ECMA 8 /* european computer manufacturers */
#define AF_DATAKIT 9 /* datakit protocols */
#define AF_CCITT 10 /* CCITT protocols, X.25 etc */
#define AF_SNA 11 /* IBM SNA */
#define AF_DECnet 12 /* DECnet */
#define AF_DLI 13 /* DEC Direct data link interface */
#define AF_LAT 14 /* LAT */
#define AF_HYLINK 15 /* NSC Hyperchannel */
#define AF_APPLETALK 16 /* Apple Talk */
#define AF_ROUTE 17 /* Internal Routing Protocol */
#define AF_LINK 18 /* Link layer interface */
    /* Moved to _ALL_SOURCE section since it doesn't
    begin with AF_ */
    /* efine pseudo_AF_XTP 19
    eXpress Transfer Protocol (no AF) */
#ifdef _AIX
#define AF_INTF 20 /* Debugging use only */
#define AF_RIF 21 /* raw interface */
#define AF_NETWARE 22
#define AF_NDD 23
#define AF_MAX 30
#else
#define AF_MAX 20
#endif
```

Der zunächst von uns verwendete Aufruf `socketpair` kann nur mit der PF_UNIX Familie arbeiten.

Das zweite Argument, `type`, bestimmt den Sockettypen. Hier sind die Angaben SOCK_STREAM und SOCK_DGRAM möglich.

Das dritte Argument ist das Protokoll. Meist setzt man es auf 0 und überläßt es dem System, das richtige Protokoll zu wählen. Bei verbindungslosen wird dann UDP genommen, bei verbindungsorientierten TCP.

Das vierte Argument ist ein Integerarray für die beiden Socketdeskriptoren. Jeder ist bi-direktional und kann für Lesen und Schreiben verwendet werden.

Die möglichen Fehlermeldungen übergehen wir und betrachten sofort Programm 10.1. Es erzeugt ein Socketpaar und verzweigt sich dann mit `fork` um miteinander zu kommunizieren.

Gray weist darauf hin, daß zum Übersetzen die `-lsocket` Option angegeben werden muß.

```
/*
 * Program 10.1 Creating a socket pair
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#define BUF_SZ 10

main(void) {
  int sock[2], /* The socket pair */
  cpid, i;
  static char buf[BUF_SZ]; /* Temporary buffer for
message */
  if (socketpair(PF_UNIX, SOCK_STREAM, 0, sock) < 0) {
    perror(„Generation error“);
    exit(1);
  }
  switch (cpid = (int) fork()) {
  case -1:
    perror(„Bad fork“);
    exit(2);
  case 0: /* The child process */
    close(sock[1]);
    for (i = 0; i < 10; i += 2) {
      sleep(1);
      sprintf(buf, „c: %d\n“, i);
```

```
        write(sock[0], buf, sizeof(buf));
        read(sock[0], buf, BUF_SZ);
        printf(„c-> %s“, buf); /* Msg. from parent */
      }
      close(sock[0]);
      break;
    default: /* The parent process */
      close(sock[0]);
      for (i = 1; i < 10; i += 2) {
        sleep(1);
        read(sock[1], buf, BUF_SZ);
        printf(„p-> %s“, buf); /* Message from child */
        sprintf(buf, „p: %d\n“, i);
        write(sock[1], buf, sizeof(buf));
      }
      close(sock[1]);
    }
    return 0;
}
```

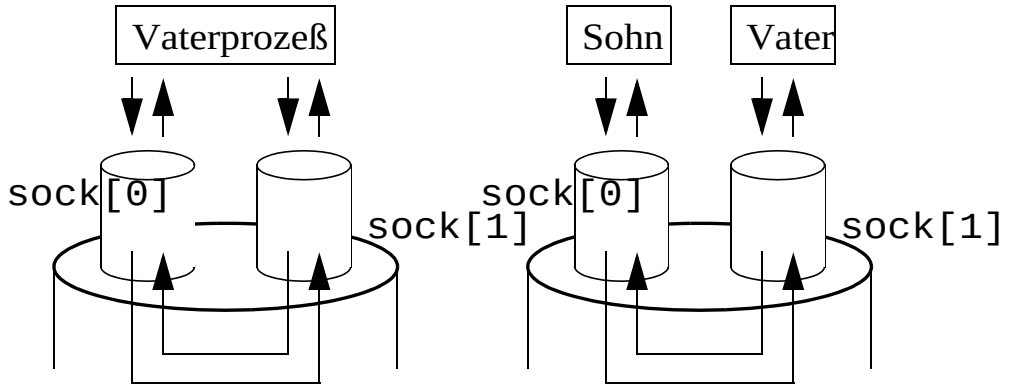
Der erzeugte Sohnprozeß schließt den Socketdeskriptor `sock[1]`. Er beginnt dann eine Schleife von 0 bis 9 in Schritten von 2 und erzeugt eine Botschaft, die mit `c:` anfängt (vom Kind an den Vater). Nachdem er jeweils eine Botschaft weggeschickt hat (`write` nach `sock[0]`), liest der Sohnprozeß aus dem selben Deskriptor `sock[0]` die Antwortbotschaft des Vaterprozesses.

Der Vaterprozeß arbeitet ganz analog. Zusammen erzeugen sie die folgende Ausgabe.

```
wegner@elsie(chpt10)$ ./p10.1
p-> c: 0
c-> p: 1
p-> c: 2
c-> p: 3
p-> c: 4
c-> p: 5
p-> c: 6
c-> p: 7
p-> c: 8
c-> p: 9
```

Man beachte, daß vor dem `fork` sowohl `sock[0]` als auch `sock[1]` zur Verfügung stehen. Nach dem `fork` schließt der Sohnprozeß

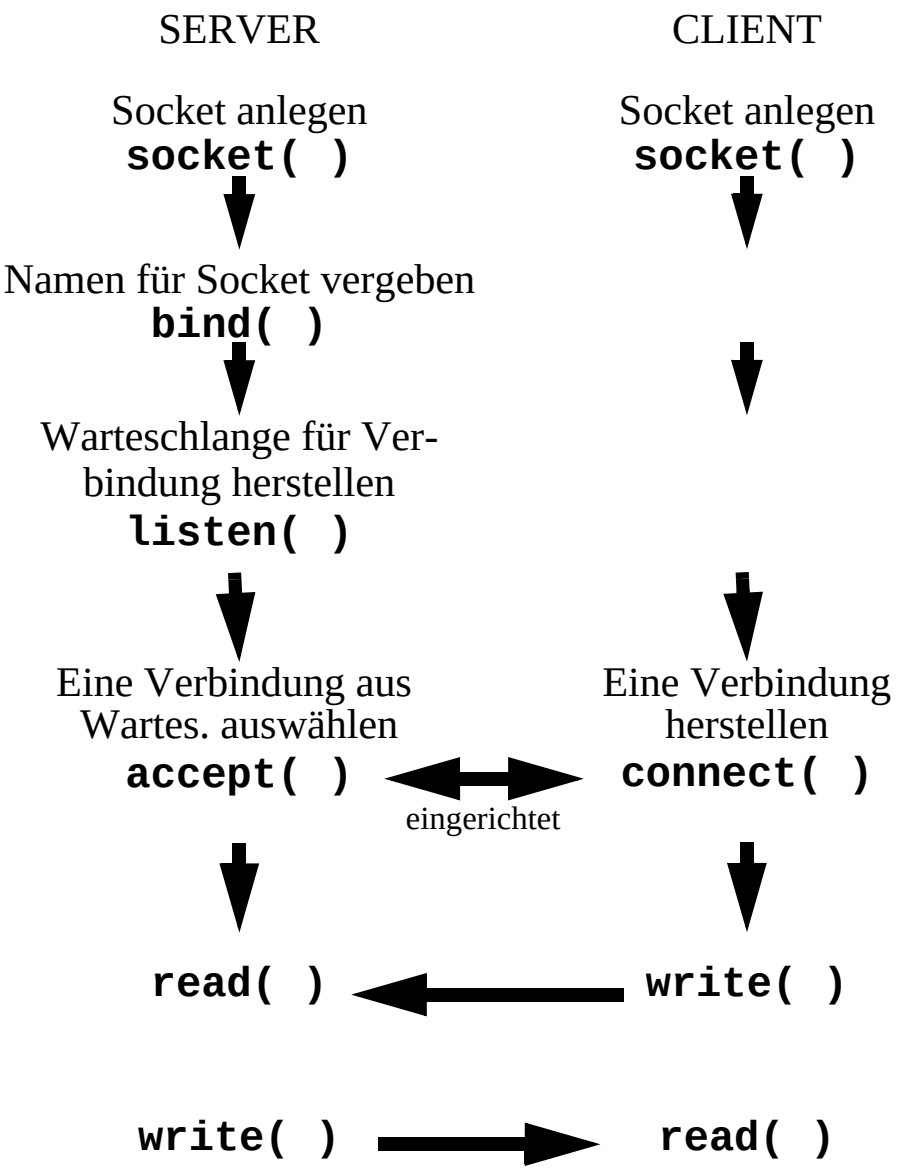
`sock[1]` und liest und schreibt mit `sock[0]`. Beim Vaterprozeß wird `sock[0]` geschlossen und mit `sock[1]` gelesen und geschrieben. Für den BS-Kern sind die Sockets aber weiterhin ein und dasselbe. Was der Sohn in `sock[0]` schreibt, liest der Vater aus `sock[1]` und umgekehrt.



vor dem `fork` nach dem `fork`

10.4 Sockets - das verbindungsorientierte Paradigma

Werden Sockets für die Interprozeßkommunikation verwendet, kann man entweder einen verbindungsorientierten Typ (SOCK_STREAM) oder einen verbindungslosen Typ (SOCK_DGRAM) angeben. Die Abfolge der Schritte für die verbindungsorientierte Variante ist in der Abbildung unten gegeben. Der Auslöser der Verbindung ist dabei der Client. Der Prozeß der die Verbindung empfängt (**accept**) ist der Server.



Der Verbindungsaufbau beginnt mit dem beiderseitigen Aufruf der Netzwerkfunktion `socket`.

Include File(s)	<sys/types.h> <sys/socket.h>		Manual	**3N**
Summary	int socket (int family, int type, int protocol);			
Return	Success	Failure	Sets `errno`	
	einen offenen Socketdeskriptor	-1	Yes	

Tab. 10–3 *socket Netzwerkaufruf*

Die drei verlangten Argumente sind die gleichen wie für `socketpair`. Für `family` setzen wir PF_UNIX oder PF_INET (TCP/IP-Familie), für `type` SOCK_STREAM oder SOCK_DGRAM), das Protokoll wird häufig wieder 0 sein.

Gray gibt noch den Hinweis, daß Programme, die einen `socket`-Aufruf enthalten, mit der `-lsocket` Option im Compileraufruf auch die *socket* Bibliothek zubinden müssen.

Wird der `socket`-Aufruf von Serverseite aus gemacht, ist das Socket zunächst noch nicht mit einer Adresse und einer Portnummer versehen. Dies geschieht erst mit dem `bind`-Aufruf. Der Schritt ist vergleichbar mit der Zuordnung einer Telefonnummer zu einem neu installierten Telefon. Wird das Socket in einer UNIX-Domäne eingerichtet, ist ein Dateiname anzugeben, in der Internet-Domäne ist es ein Adresse/Portnummernpaar. Die folgende Aufstellung für bind gibt die Details an.

Include File(s)	<sys/types.h> <sys/socket.h>		Manual	**3N**
Summary	int bind (int socket, const struct sockaddr *name, int namelen);			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Tab. 10–4 *bind Netzwerkaufruf*

Das erste Argument ist der Socketdeskriptor, den der `socket`-Aufruf zurückgeliefert hat. Das zweite Argument ist eine generische Adreßstruktur, die in `<sys/socket.h>` definiert ist als

```
struct sockaddr {
 ushort_t sa_family; /* address family */
 char sa_data[14]; /* up to 14 bytes of direct address */
};
```

Für die Socket-Variante, die in der UNIX-Domäne arbeitet, wird hierfür auf `<sys/un.h>` die tatsächliche Ausprägung

```
#if defined(COMPAT_43) && !defined(_KERNEL)
struct sockaddr_un {
 ushort_t sun_family; /* AF_UNIX */
 char sun_path[104]; /* path name (gag) */
};
#else
struct sockaddr_un {
 uchar_t sun_len; /* sockaddr len including null */
 sa_family_t sun_family; /* AF_UNIX */
 char sun_path[104]; /* path name (gag) */
};
#endif /* COMPAT_43 && !_KERNEL */
```

eingesetzt, wobei `sun_familiy` auf AF_UNIX gesetzt wird. Die Komponente `sun_path` ist dann der relative oder absolute Pfadname für die Datei. Diese wird im übrigen bei `ls -l` mit dem Typindikator `s` oder `p` angezeigt (pipe bzw. socket).

Aus `man 3 bind` kann man ein Beispiel entnehmen.

```
The following program fragment illustrates the use of
the bind subroutine

to bind the name „/tmp/zan/“ to a UNIX domain socket.

#include <sys/un.h>

.
.
.
struct sockaddr_un addr;

.
.
.
strcpy(addr.sun_path, „/tmp/zan/“);
addr.sun_len = strlen(addr.sun_path);
addr.sun_family = AF_UNIX;
bind(s,(struct sockaddr*)&addr, SUN_LEN(&addr));
```

Bei dem Einsatz in der Internet-Domäne benötigen wir die Headerdatei `<netinet/in.h>` (AIX: `/usr/include/netinet/in.h`), die in der Programmliste enthalten sein muß. Sie definiert die Struktur als

```
/*
 * Socket address, internet style.
 */
struct sockaddr_in {
 uchar_t sin_len;
 sa_family_t sin_family;/* AF_INET*/
 in_port_t sin_port;/* 16 bit port */
 struct in_addr sin_addr;/* Zeiger Adr.struktur */
 uchar_t sin_zero[8];/* zukünft. Erweiterung */
};
```

Die erste Komponente dieser Struktur erhält die symbolische Konstante AF_INET.

Die zweite Komponente erhält die 16-bit Portnummer. Diese stellt einen Offset zu der angegebenen Internetadresse dar und gibt den tatsächlichen Endpunkt für die Kommunikation an. Sie wirkt daher wie die Anschlußerweiterung (Durchwahlerweiterung) einer Telefonnummer.

Zugeordnete Portnummern kann man sich aus der Datei `/etc/services` auflisten lassen, die auf unseren Rechnern fast 700 Einträge hat. Hier ist das Anfangsstück dieser Liste aus unseren Rechnern.

```
tcpmux          1/tcp  # TCP Port Service Multiplexer
tcpmux          1/udp  # TCP Port Service Multiplexer
compressnet     2/tcp  # Management Utility
compressnet     2/udp  # Management Utility
compressnet     3/tcp  # Compression Process
compressnet     3/udp  # Compression Process
echo            7/tcp
echo            7/udp
discard         9/tcp sink null
discard         9/udp sink null
systat          11/tcp users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp quote
msp             18/tcp # Message Send Protocol
msp             18/udp # Message Send Protocol
chargen         19/tcp ttytst source
chargen         19/udp ttytst source
ftp-data        20/tcp
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp mail
nsw-fe          27/tcp # NSW User System FE
nsw-fe          27/udp # NSW User System E
msg-icp         29/tcp # MSG ICP
msg-icp         29/udp # MSG ICP
msg-auth        31/tcp # MSG Authentication
msg-auth        31/udp # MSG Authentication
dsp             33/tcp # Display Support Protocol
dsp             33/udp # Display Support Protocol
time            37/tcp timserver
time            37/udp timserver
```

Jeder Service hat einen Namen, wie z.B. `telnet`, und eine Portnummer, verbunden mit einem speziellen Protokoll, z.B. 23/tcp. Portnummern unterhalb von 1024 sind reserviert für Prozesse, die mit `root` (oder suid-root) Berechtigung laufen.

Die dritte Komponente der `sockaddr_in` Struktur ist die 32 bit Internet Adresse, ggf. adjustiert für Byteanordnungen (little endian, big endian).

Die vierte Komponente ist momentan unbenutzt.

Die Sache mit der generischen Struktur hat einen ganz einfachen Hintergrund. Je nach Adreßfamilie füllen wir zunächst die richtige Struktur mit den Angaben. Die aufgerufene Funktion `bind` kann später aus dem ersten `u_short`-Argument `sa_family` die Strukturart des Rests (UNIX Dateiname oder Internet Port/Adressenpaar) erkennen. Im Aufruf selbst wird auf den Zeiger auf die Struktur eine cast-Operation gemacht, um die `bind`-Funktion davon zu überzeugen, sie erhalte die Adresse der generischen Struktur `struct sockaddr`.

Dies läßt sich aus dem kurzen Beispiel aus der `man`-Seite oben gut erkennen. Weitere Beispiele folgen.

Fehlercodes, die `bind` in beiden Fällen (UNIX vs Internet) zurückliefert, werden hier übergangen.

Nach dem `bind` muß der Server ein `listen` absetzen. Er legt dadurch eine Warteschlange für hereinkommende Verbindungswünsche an. Sollte diese „überlaufen“, erhält ein Kundenprozeß den ECONNREFUSED Fehler vom Server.

Include File(s)	<sys/types.h> <sys/socket.h>		Manual	**3N**
Summary	int listen (int socket, int backlog);			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Tab. 10–5 *listen Netzwerkaufruf*

Das erste Argument ist ein gültiger Socketdeskriptor. Das zweite Argument, `backlog`, gibt die maximale Größe der Warteschlange an.

Anmerkung 1: In [Stev98, S. 35], wird die Analogie zum Telefon nochmals aufgegriffen. Der `socket`-Aufruf entspricht der Beschaffung eines Telefonapparats. Mit `bind` wird die Nummer allen bekanntgegeben. Mit `listen` wird die Glocke angestellt, damit man Anrufer hört. Der Anrufer macht `connect` (siehe unten), was voraussetzt, daß er die Nummer kennt und wählt. Wird der Server angewählt, nimmt er mit `accept` den Anruf entgegen, wobei er, wie bei ISDN, die Identität des Anrufers erfährt (allerdings - anders als bei ISDN - erst beim Abheben).

Anmerkung 2: Was den zweiten Parameter, `backlog`, angeht, sind Werte um 5 bis 10 sind üblich. In AIX gibt es aber auch die SOMAXCONN Konstante in `/usr/include/sys/socket.h`, die 1024 als gültigen Wert akzeptiert. Im Zusammenhang mit Web-Servern, die HTTP-Sockets enthalten, sind Warteschlangen mit vielen tausend Verbindungswünschen keine Seltenheit. Überhaupt ist die Rolle dieses Wertes sehr diffus, was auch damit zusammenhängt, daß es unter TCP in Wirklichkeit zwei Warteschlangen gibt, bedingt durch das TCP 3-Wege-Handshakeprotokoll.

Eine vernünftige Interpretation, die auch vom AIX Manual unterstützt wird, ist zu sagen, daß `backlog` die Anzahl der simultanen Verbindungen, die ein Server unterhalten will, beschränkt. Damit wären auch für einen Web-Server kleine Werte um die 10 bis 100 sinnvoll, da nie mehr als diese Zahl an Seiten „gleichzeitig“ abgeschickt werden.

Stevens [Stev98, S. 99] erwähnt auch einen Hacker-Angriff durch getürkte `connect`-Versuche mit zufälligen (falschen) Client-IP-Adressen, die einen Server lahmlegen, weil dessen Warteschlange überläuft. (Ende Anmerkung 2)

Jetzt kann der Server `connect`-Versuche der Clients annehmen. Dazu führt er `accept` aus und blockiert, wenn keine Verbindungswünsche vorliegen. Wichtig ist dabei, daß `accept` mit zwei Socketdeskriptoren arbeitet. Der erste geht als Argument in den Aufruf hinein und stammt aus dem vorherigen `bind` und `listen`. Der zweite ist brandneu und kommt aus dem Aufruf zurück.

Include File(s)	<sys/types.h> <sys/socket.h>		Manual	**3N**
Summary	int accept(int socket, struct sockaddr *addr, int *addrlen);			
Return	Success	Failure	Sets `errno`	
	positive ganze Zahl für einen **neuen** Sok-ketdeskriptor	-1	Yes	

Tab. 10–6 *accept Netzwerkaufruf*

Dieser neue Socketdeskriptor wird für `read` und `write` genutzt. Der alte Deskriptor bleibt bestehen und kann für weitere, nebenläufige Verbindungen durch weitere `accept`-Aufrufe genutzt werden, was Gray aber nur mit einer lauwarmen Bemerkung zuläßt: „... remains as it was and can, in some settings, still continue to **`accept`** additional connections [Gray98, S. 285].“)

Das zweite Argument im `accept`-Aufruf ist wieder die generische Adresse, hinter der sich, je nach domain, ein UNIX-Pfadname oder eine IP-Adresse verbirgt. Das letzte Argument gibt die Größe der Adreßangabe an.

Auf Clientseite steht im verbindungsorientierten Paradigma jetzt der `connect`-Aufruf an. Dieser verlangt den Socketdeskriptor aus dem vorangegangenen `socket`-Aufruf (`bind` wird auf Kundenseite nicht gebraucht). Danach kommt als 2. Argument wieder die Adreßstruktur mit der Angabe der Serveradresse und als 3. die Adreßgröße.

Include File(s)	<sys/types.h> <sys/socket.h>		Manual	**3N**
Summary	int connect(int socket, struct sockaddr *name, int namelength);			
Return	Success	Failure	Sets `errno`	
	0	-1	Yes	

Tab. 10–7 *connect Netzwerkaufruf*

Die Anzahl der Fehlermeldungen ist riesig. Bekannt sind bei TCP-Umgebung besonders ETIMEOUT (Connection timed out), wenn es nicht gelingt, eine Verbindung herzustellen, z.B. weil zwar die IP-domain korrekt ist, es den angegebenen Rechner in diesem Subnetz aber nicht gibt (weicher Fehler). Ein sog. harter Fehler tritt auf, wenn z.B. auf dem angegebenen Server kein Prozeß läuft, der für das Socket zuständig ist. Die Fehlermeldung ist dann z.B. ECONNREFUSED (Connection refused), die sofort - also nicht erst nach ca. 3 Minuten mit diversen Neuversuchen durch TCP - zurückkommt.

Im jedem Fall muß man bei verbindungsorientiertem `connect` den Deskriptor mit `close` schließen und sich über `socket` einen neuen generieren. Dies ist zu beachten, wenn z.B. in einer Schleife für eine bekannte **domain** ein ansprechbarer **host** gesucht wird und dazu die host-id hochgezählt wird.

Ein `close`-Aufruf wird auch erwartet nach Beendigung der Verbindung, die natürlich auch auf Kundenseite mittels `read` und `write` betrieben wird.

10.4.1 Beispiel eines verbindungsorientierten Sockets

Wir starten mit einem Beispiel, das einen Server mit einem Client innerhalb der UNIX-Domäne über ein SOCK_STREAM-Socket verbindet. Der Server wird eine Botschaft des Kundenprozesses empfangen und sie

ausdrucken. Das Serverprogramm liegt als `p10.2` vor und muß im Hintergrund gestartet werden. Der Client ist `p10.3`. Zusammen aufgerufen produzieren sie den folgenden Output.

```
 ./p10.2 &
[1] 22322
wegner@elsie(chpt10)$ ls -l my_soc
srwxrwxrwx 1 wegner mitarb 0 Jun 17 13:08 my_sock
wegner@elsie(chpt10)$ ./p10.3
wegner@elsie(chpt10)$ s-> c: 1
s-> c: 2
s-> c: 3
s-> c: 4
s-> c: 5
s-> c: 6
s-> c: 7
s-> c: 8
s-> c: 9
s-> c: 10

[1]+ Done ./p10.2
```

Interessant ist, daß man das Socket im aktuellen Verzeichnis sehen kann. Es wird anschließend entfernt. Unter AIX liegt allerdings eine andere Adreßstruktur vor, die interessanterweise den letzten Buchstaben abschnitt. Dadurch lief zwar das Beispiel, das Socket blieb aber unter dem Namen my_soc bestehen und wurde auch nicht von der eigens angegebenen Abräumroutine entfernt, weshalb ein neuer Serveraufruf auch einen `bind`-Fehler produziert: `bind error: Address already in use`. In AIX sollte man deshalb das eigens vorgesehene Macro `SUN_LEN()` verwenden.

```
/*
 Server - UNIX domain, connection-oriented
*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h> /* as we are using UNIX protocol */
#define NAME „my_sock“
```

```
main( void ) {
 int orig_sock,
     /* Original socket descriptor in server */
 new_sock, /* New socket descriptor from connect */
 clnt_len, /* Length of client address */
 i; /* Loop counter */
 static struct sockaddr_un
     clnt_adr, /* UNIX addresses of client and server
*/
     serv_adr;
 static char buf[10]; /* Buffer for messages */
 void clean_up( int, char *);
     /* Close socket and remove it routine */

 if ((orig_sock = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
{
     /* SOCKET */
     perror(„generate error“);
     exit(1);
 }
 serv_adr.sun_family = AF_UNIX;
     /* Set tag appropriately */
 strcpy(serv_adr.sun_path,NAME);
     /* Assign name (108 chars max) */
 unlink(NAME); /* Remove old copy if present */
 if (bind( orig_sock, (struct sockaddr *) &serv_adr,
     /* BIND - alte Gray Version, in AIX verwende
     SUN_LEN */

sizeof(serv_adr.sun_family)+strlen(serv_adr.sun_path)
) < 0)
     { perror(„bind error“);
       clean_up(orig_sock, NAME);
       exit(2);
 }
 listen(orig_sock, 1); /* LISTEN */
 clnt_len = sizeof(clnt_adr);
 if ((new_sock = accept( orig_sock,
     (struct sockaddr *) &clnt_adr,
     &clnt_len)) < 0) { /* ACCEPT */
       perror(„accept error“);
       clean_up(orig_sock, NAME);
       exit(3);
 }
```

```
    for (i = 1; i <= 10; i++) { /* Process */
        sleep(1);
        read(new_sock, buf, sizeof(buf));
        printf(„s-> %s“, buf);
    }
    close(new_sock);
    clean_up(orig_sock, NAME);
    exit(0);
   }
   void
   clean_up( int sd, char *the_file ){
        close( sd ); /* close it */
        unlink( the_file ); /* rm it */
   }
```

Das Client-Programm sieht wie folgt aus.

```
/*
 Client - UNIX domain, connection-oriented
*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#define NAME „my_sock“

main( void ) {
 int orig_sock,
  /* Original socket descriptor in client */
 i; /* Loop counter */
 static struct sockaddr_un
 serv_adr; /* UNIX address of the server process */
 static char buf[10]; /* Buffer for messages */
 if ((orig_sock = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
   /* SOCKET */
  perror(„generate error“);
  exit(1);
 }
 serv_adr.sun_family = AF_UNIX;
  /* Set tag appropriately */
 strcpy(serv_adr.sun_path, NAME);/* Assign name */
 if (connect( orig_sock, (struct sockaddr *) &serv_adr,
   /* CONNECT - hier noch alte Gray Version !!*/
```

```
    sizeof(serv_adr.sun_family)+
    strlen(serv_adr.sun_path)) < 0) {
      perror(„connect error“);
      exit(1);
   }
   for (i = 1; i <= 10; i++) { /* Send msgs */
    sprintf(buf, „c: %d\n“, i);
    write(orig_sock, buf, sizeof(buf));
   }
   close(orig_sock);
   exit(0);
  }
```

Gray schlägt auch vor, sich mit `netstat` die Anwesenheit von Sockets anzeigen zu lassen. Nach Start des Servers im Hintergrund wird im obigen Fall angezeigt:

```
...
tcp 0 0 elsie.db.informa.cppbr *.* LISTEN

Active UNIX domain sockets
SADR/PCB Type Recv-Q Send-Q Inode ...Addr
5abe900  stream   0    0    6aa7600  /tmp/.X11-unix/X0
5aaa600  stream   0    0    6acb2b8  /tmp/.X11-unix/XIM
5abffc0
5ab1d00  stream   0    0    6abf518  /tmp/.info-help
5ae5040
5ab1c00  dgram    0    0  0 5abff00  5abfe00
5ae5000
5ad1900  stream   0    0  6dde60c    my_sock
```

Übung 10–2

Gray schlägt als Übung vor, den Server so zu verändern, daß er drei Clients bedienen kann, die man dann mit

```
server%
client &; client &; client
```

aufruft. In der jetzigen Form kommen laut Gray zwei Fehlermeldungen heraus (connect error), in AIX jedoch nicht, wohl aber nur eine Ausgabefolge. Überprüfen sie die Verhältnisse und verändern Sie das Programm, damit es mit bis zu drei Clients funktioniert.

10.4.2 Ein Beispiel für Internet Stream Sockets

Um mit einem Rechner im Internet zu kommunizieren, muß man dessen Adresse, z.B. seine Internet-Nr., kennen. Hierfür kann man den Netzaufruf `gethostbyname` verwenden, der einen Zeiger auf eine `hostent` Struktur abliefert. Diese ist in der Headerdatei `<netdb.h>` definiert.

```
struct hostent {
    char *h_name; /* official name of host */
    char **h_aliases; /* alias list */
    int h_addrtype; /* host address type */
    int h_length; /* length of address */
    char **h_addr_list; /* list of addresses from */
        /* name server */
    #define h_addr h_addr_list[0]
      /* address, for backward compatiblity */
};
```

Die Funktion bezieht ihre Information aus der Datei `/etc/hosts` oder ähnlichen Diensten. Die Funktion `gethostbyname` hat die folgende Definition.

Include File(s)	<sys/types.h> <sys/socket.h> <netdb.h> <netinet/in.h>		Manual	**3N**
Summary	struct hostent *gethostbyname(const char *name);			
Return	Success	Failure	Sets `errno`	
	Zeiger auf *`hostent`* Struktur	NULL	Yes	

Tab. 10–8 *gethostbyname Netzwerkaufruf*

Ihr Aufrufargument ist der Rechnername. Im Programm 10.4 wird dies verwandt, wobei der Rechnername vom Anwender interaktiv einzugeben ist. Die Ausgabe lautet:

```
./p10.4
Enter host name to look up: elsie
Here is what I found about elsie :
Official name : elsie.db.informatik.uni-kassel.de
Aliases :
Address type : 2
Address length: 43´] = 141.51.180.6
```

Das Programm selbst liegt wie üblich als Quelltext `p10.4.c` vor.

```
/*
 Checking host entries
*/
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

main( void ) {
 struct hostent *host;
 static char who[10];
 printf(„Enter host name to look up: „);
 scanf(„%10s“, who);
 host = gethostbyname( who );
 if ( host != (struct hostent *) NULL ) {
  printf(„Here is what I found about %s :\n“, who);
  printf(„Official name : %s\n“, host->h_name);
  printf(„Aliases : „);
  while ( *host->h_aliases ) {
  printf(„%s „, *host->h_aliases );
  ++host->h_aliases;
 }
 printf(„\nAddress type : %i\n“, host->h_addrtype);
 printf(„Address length: %i\n“, host->h_length);
 printf(„Address list : „);
 while ( *host->h_addr_list ) {
  struct in_addr in;
  memcpy( &in.s_addr, *host->h_addr_list, sizeof
     (in.s_addr));
  printf(„[%s] = %s „, *host->h_addr_list,
     inet_ntoa(in));
  ++host->h_addr_list;
 }
```

```
    printf(„\n“);
    }
  }
```

Neben der Internetadresse brauchen wir noch den Port und Dienst. Hierzu gibt es die Funktion `getservbyname`, die uns einen Zeiger auf eine Struktur mit entsprechenden Werten liefert. Man beachte, daß Ports 0 bis 1023 reserviert sind für den Super-user, ab 1024 sind sie für jeden Anwender frei.

Include File(s)	<netdb.h>		Manual	**3N**
Summary	struct servent *getservbyname(const char *name, char *proto);			
Return	Success	Failure	Sets `errno`	
	Zeiger auf `servent` Struktur	NULL		

Tab. 10–9 *getservbyname Netzwerkaufruf*

Die zurückgelieferte Struktur ist wieder in `<netdb.h>` definiert. Ein Programm, das den Aufruf demonstriert ist `p10.5`, wobei besonders der Aufruf `ntohs`, der die Byteordnung einer Rechnerfamilie für die Angabe der Portnummer richtigstellt.

```
struct servent {
 char *s_name; /* official service name */
 char **s_aliases; /* alias list */
 int s_port; /* port # */
 char *s_proto; /* protocol to use */
};
```

Übliche Angaben für ein Protokoll wäre z.B. `tcp`.

```
/* Checking service -- port entries for a host */
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>

main( void ) {
 struct servent *serv;
 static char protocol[10], service[10];
 printf(„Enter service to look up : „);
 scanf(„%9s“, service);
 printf(„Enter protocol to look up: „);
 scanf(„%9s“, protocol);
 serv = getservbyname( service, protocol );
 if ( serv != (struct servent *)NULL ) {
 printf(„Here is what I found \n“);
 printf(„Official name : %s\n“, serv->s_name);
 printf(„Aliases : „);
 while ( *serv->s_aliases ) {
 printf(„%s „, *serv->s_aliases );
 ++serv->s_aliases;
 }
 printf(„\nPort number : %i\n“, htons(serv->s_port));
 printf(„Protocol Family: %s\n\n“, serv->s_proto);
 } else printf(„Service %s for protocol %s not
found\n“,service,protocol);
}
```

Die Ausgabe lautet:

```
wegner@elsie(chpt10)$ ./p10.5
Enter service to look up : mail
Enter protocol to look up: tcp
Here is what I found
Official name : smtp
Aliases : mail
Port number : 25
Protocol Family: tcp
```

Jetzt sind wir bereit, ein Beispiel für eine Client-Server-Verbindung über Internet Stream Sockets aufzubauen. Als Port sollte man eine nicht verwendete Nummer aussuchen.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
/*SW, #include <sys/sockio.h>*/

#define PORT 6969
static char buf[BUFSIZ]; /* Buffer for messages */
```

Server und Client verwenden die gemeinsame, oben aufgelistete Datei `local.h` für den Austausch der Portnummer und anderer Angaben.

Als Aufgabe werden Botschaften vom Kundenprozess an einen Server geschickt, der sie in Großbuchstaben übersetzt und wieder zurückschickt. Der Botschaftenaustausch endet, wenn eine Zeile mit nur einem Punkt als erstem Zeichen übermittelt wird.

Für jede aufgebaute Verbindung erzeugt der Server mit `fork` einen Sohnprozeß, der parallel läuft und die Kommunikation übernimmt. Das Serverprogramm aus `p10.6.c` folgt hier.

```
/*
 * Internet domain, connection-oriented SERVER
 */
#include „local.h“
main( void ) {
 int orig_sock,
  /* Original socket descriptor in server */
 new_sock, /* New socket descriptor from connect */
 clnt_len; /* Length of client address */
 static struct sockaddr_in
  clnt_adr, /* Internet address of client & server */
  serv_adr;
 static char buf[BUFSIZ]; /* Buffer for messages */
 int len, i;
  /* Misc counters, etc. */
```

```
if ((orig_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
 /* SOCKET */
 perror(„generate error“);
 exit(1);
}
memset( &serv_adr, 0, sizeof(serv_adr) );
 /* Clear out */
serv_adr.sin_family = AF_INET; /* Set address type */
serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
                /* Any interface */
serv_adr.sin_port = htons(PORT); /* Use our fake port */

/* BIND */
if (bind( orig_sock, (struct sockaddr *) &serv_adr,
                sizeof(serv_adr)) < 0) {
 perror(„bind error“);
 close(orig_sock);
 exit(2);
}
if (listen(orig_sock, 5) < 0 ) { /* LISTEN */
 perror(„listen error“);
 exit(3);
}
do {
 clnt_len = sizeof(clnt_adr);
 if ((new_sock = accept( orig_sock, (struct sockaddr *)
        &clnt_adr, &clnt_len)) < 0) { /* ACCEPT */
   perror(„accept error“);
   close(orig_sock);
   exit(4);
 }
 if ( fork( ) == 0 ) { /* In CHILD process */
   while ( (len=read(new_sock, buf, BUFSIZ)) > 0 ){
     for (i=0; i < len; ++i) /* Change the case */
       buf[i] = toupper(buf[i]);
     write(new_sock, buf, len); /* write it back */
     if ( buf[0] == '.' ) break; /* are we done? */
   }
   close(new_sock); /* In CHILD process */
   exit( 0 );
 } else close(new_sock); /* In PARENT process */
} while( 1 ); /* FOREVER */
}
```

Das Programm enthält einige unbekannte Elemente.

- `memset` wird verwandt um die Adreßstruktur mit Nullwerten zu löschen.
- beim Setzen der Interface und Port-Werte für die Serveradresse verwenden wir die Funktionen `htonl` und `htons` (host-to-network-long, ~-short)
- durch die Konstante INADDR_ANY aus <`netinet/in.h`>, die dort als 0 deklariert ist, wird angezeigt, daß jede Adresse vom Sokkettyp (SOCK_STREAM) akzeptiert wird.

Der Client verlangt als Kommandozeilenargument den Namen eines Servers. Die spezielle Rechneradresse wird dann mit `gethostbyname` ermittelt. Diese wird in der `hostent`-Struktur gespeichert, auf die man über den Zeiger `*host` zugreift.

```
/*
 * Internet domain, connection-oriented CLIENT
*/
#include „local.h“
main( int argc, char *argv[] ) {
 int        orig_sock,
            len;
 static struct sockaddr_in
            serv_adr;
 struct hostent *host;

 if ( argc != 2 ) {
  fprintf(stderr, „usage: %s server\n“, argv[0]);
  exit(1);
 }
 host = gethostbyname(argv[1]); /* GET INFO */
 if (host == (struct hostent *) NULL ) {
  perror(„gethostbyname „);
  exit(2);
 }
 memset(&serv_adr, 0, sizeof( serv_adr));
  /* Clear it out */
 serv_adr.sin_family = AF_INET;
 memcpy(&serv_adr.sin_addr, host->h_addr, host-
>h_length);
 serv_adr.sin_port = htons( PORT );
```

```
if ((orig_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    /* SOCKET */
 perror(„generate error“);
 exit(3);
} /* CONNECT */
if (connect(orig_sock, (struct sockaddr *)&serv_adr,
         sizeof(serv_adr)) < 0) {
 perror(„connect error“);
 exit(4);
}
do {
 write(fileno(stdout),“> „, 3); /* Prompt the user */
 if ((len=read(fileno(stdin), buf, BUFSIZ)) > 0) {
    /* Get user input */
   write(orig_sock, buf, len); /* Write to socket */
   if ((len=read(orig_sock, buf, len)) > 0 )
     /* If returned */
     write(fileno(stdout), buf, len); /* display */
 }
} while( buf[0] != '.' );
close(orig_sock);
exit(0);
}
```

An Kleinigkeiten weist Gray darauf hin, daß mit `memcpy` die Adresse kopiert wird, damit zusätzliche Nullbytes oder fehlende `\0` keine Probleme bereitet.

Mit dem Anlegen des Sockets mit Adressfamilie AF_INET und als SOCK_STREAM wird die Verbindung mit `connect` hergestellt und der Client fordert vom Anwender mit dem Promptzeichen „>“ eine Eingabe, die er in den Socketdeskriptor `orig_sock` schreibt. Aus diesem liest er dann die Antwort und zeigt sie auf der lokalen Standardausgabe an.

Die folgende Abbildung zeigt den Gebrauch, wobei wir den Server auf `haensel` gestartet haben (dorthin kopiert unter dem Namen `sockserver`). Auf `haensel` müssen wir ihn später auch explizit killen. Auf `elsie` läuft der Client als `p10.7`. Während der Kommunikation kann man ggf. mit `ps` noch die Existenz mehrerer Serverkopien feststellen.

haensel

```
wegner@haensel(wegner)$ ./sockserver &
[1] 2886
wegner@haensel(wegner)$ ps a
 PID TTY STAT TIME COMMAND
 2886 pts/7 A 0:00 ./sockserver
 3392 pts/7 A 0:02 -bash
 4262 Z 0:00 <defunct>
 14312 lft0 A 0:01 /usr/sbin/getty
/dev/console
 15702 Z 0:00 <defunct>
 17748 pts/5 A 0:39 /usr/local/bin/netscape-
3.01
 18248 pts/5 A 0:00 -bash
 24834 pts/4 A 0:00 -bash
 26040 pts/6 A 0:00 /usr/bin/bash -i
 27728 pts/7 A 0:00 ps a
 30298 pts/5 A 4:49 emacs etfoc.c
 30792 Z 0:00 <defunct>
wegner@haensel(wegner)$ kill 2886
[1]+ Terminated ./sockserver
wegner@haensel(wegner)$
```

elsie

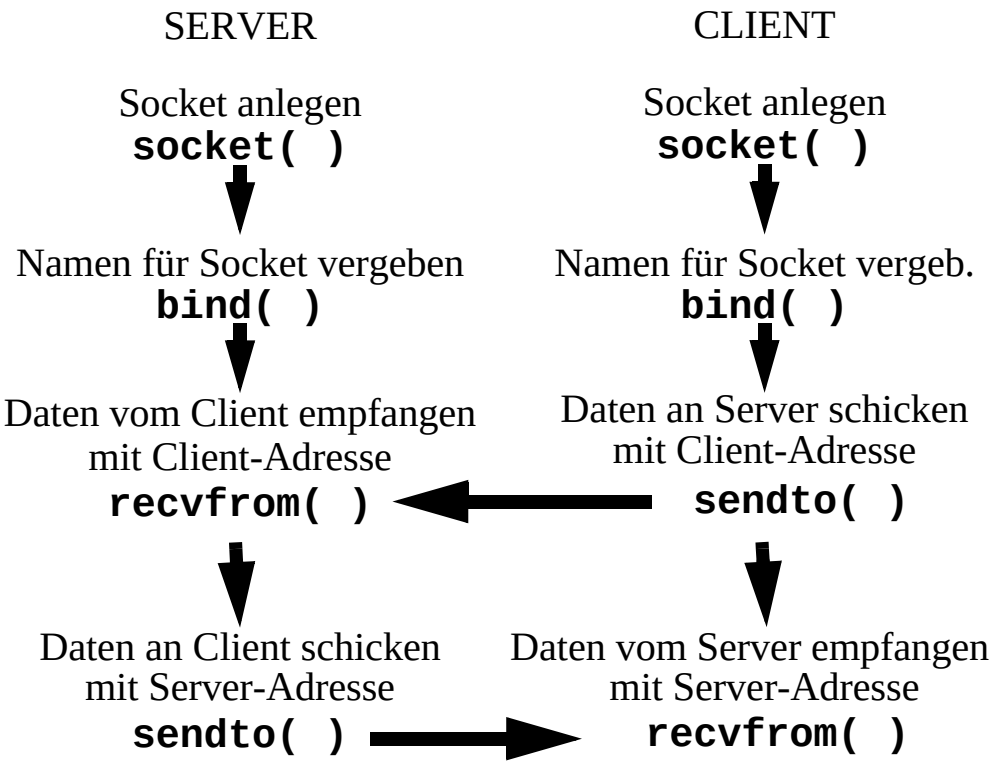
```
wegner@elsie(chpt10)$ ./p10.7
usage: ./p10.7 server
wegner@elsie(chpt10)$ ./p10.7 haensel
> hallo Leute
HALLO LEUTE
> Verbindung steht
VERBINDUNG STEHT
> .
.
wegner@elsie(chpt10)$
```

Wir beenden damit die Behandlung der verbindungsorientierten Sockets.

10.5 Verbindungslose Sockets

Wir wollen im Datagrammodus von Sockets nur das Kommunikationsprinzip mit Senden (*send*) und Empfangen *(receive*) und die Funktionsdefinitionen kennenlernen. Auf ausführliche Beispiele verzichten wir, der Leser findet sie in 10.8, 10.9 für die UNIX-Domäne und in 10.10, 10.11 für die Internetdomäne.

Generell erzeugen in der verbindungslosen Variante Server und Client wieder ein Socket mit dem `socket` Netzwerkaufruf. Anders als beim verbindungsorientierten Ansatz müssen jetzt aber beide Prozesse ein `bind` machen, um das Socket mit der Adresse zu verbinden. Entsprechend kann der Client auf `connect` verzichten.



Stattdessen senden beide Pakete über `sendto` an den jeweiligen Adressaten, bzw. empfangen welche mittels `recvfrom`. Der Aufruf `sendto` erstetzt demnach das `connect-write` Pärchen, während `recvfrom` für `accept-read` beim Server zu sehen ist (vgl. Abbildung oben).

Der `sendto`-Aufruf ist dabei nur eine von mehreren möglichen Alternativen um Daten in ein Socket zu schreiben. Die folgende Tabelle zeigt drei mögliche Netzwerkaufrufe.

<table>
<tr><td>Include File(s)</td><td colspan="2"><sys/types.h>
<sys/socket.h>
<sys/uio.h></td><td>Manual Section</td><td>3N</td></tr>
<tr><td>Summary</td><td colspan="4">int send(int socket, const char *msg,
int len, int flags);
int sendto(int socket, const char *msg,
int len, int flags,
const struct sockaddr *to,
int tolen);
int sendmsg(int socket,
const struct msghdr *msg,
int flags);</td></tr>
<tr><td rowspan="2">Return</td><td>Success</td><td>Failure</td><td colspan="2">Sets <code>errno</code></td></tr>
<tr><td>Anzahl der gesendeten Bytes</td><td>-1</td><td colspan="2">Yes</td></tr>
</table>

Tab. 10–10 *send, sendto, sendmsg Netzwerkaufrufe*

Die Form `send` kann nur mit verbindungsorientierten Sockets (SOCK_STREAM) verwendet werden. Dagegen funktionieren `sendto` und `sendmsg` mit beiden Arten, werden aber gewöhnlich nur für SOCK_STREAM eingesetzt. Die ersten beiden versenden einen Bytestrom, die letzte Form den Inhalt einer Struktur, bei der die Daten verteilt im Hauptspeicher stehen dürfen. Wir verzichten auf die Behandlung dieser Form.

Als Flags kann man MSG_OOB (Message Out Of Band = für Internet stream based Sockets zum Versand wichtiger Botschaften) und MSG_DONTROUTE (Routentabellen umgehen und versuchen, direkt zu senden) angeben.

Bei Erfolg liefern die Aufrufe im Stil von write die Anzahl der gesendeten Bytes ab. Im Falle eines ungebundenen Sockets auf Empfängerseite werden die Bytes weggeschmissen.

Auf der Empfängerseite gibt es drei *receive* Formen. Alle drei Funktionen blockieren den Prozeß, solange keine Botschaften am Socket eingetroffen sind.

<table>
<tr><td>Include File(s)</td><td colspan="2"><sys/types.h>
<sys/socket.h>
<sys/uio.h></td><td>Manual Section</td><td>3N</td></tr>
<tr><td>Summary</td><td colspan="4">int recv(int socket, char *buffer,
int len, int flags);
int recvfrom(int socket, char *buffer,
int len, int flags
const struct sockaddr *from,
int fromlen);
int recvmsg(int socket,
const struct msghdr *msg,
int flags);</td></tr>
<tr><td rowspan="2">Return</td><td>Success</td><td>Failure</td><td colspan="2">Sets <code>errno</code></td></tr>
<tr><td>Anzahl der empfange-nen Bytes</td><td>-1</td><td colspan="2">Yes</td></tr>
</table>

Tab. 10–11 *recv, recvfrom, recvmsg Netzwerkaufrufe*

Die erste Form sollte wegen der fehlenden Absenderidentifikation nur mit einem Stream-Socket benutzt werden. Die anderen sind sowohl verbindungsorientiert als auch verbindungslos zu benutzen. Üblicherweise wird man eine mit `sendto` (`sendmsg`) auf Absenderseite aufgebaute Netzwerkverbindung symmetrisch mit `recvfrom` (`recvmsg`) auf Empfängerseite koppeln.

Nützlich sind die beiden möglichen Flags: MSG_OOB (Message Out Of Band) erlaubt den Empfang wichtiger Botschaften außerhalb der durch das Internet-Stream-Socket-Protokoll bestimmten Reihenfolge; MSG_PEEK erlaubt es, Botschaften anzuschauen (to peek at) ohne sie zu verbrauchen, d.h. anschließende `read`-receive-Aufrufe können diese Botschaften noch empfangen.

10.5.1 UNIX Domain Datagram Socket Beispiel

Gray bietet jetzt mit den Programmen 10.8 und 10.9 ein Beispiel für Datagramsockets innerhalb eines Rechners an. Die Aufgabe ist die schon oben gesehene Schleife mit 10 Botschaften, die vom Server an den Client zurückgespiegelt werden. Interessant ist der Aufruf mehrerer Clients für einen Server.

Wir übergehen dieses Beispiel hier und geben stattdessen noch das folgende Internet Domain Beispiel an.

10.5.2 Internet Domain Datagram Socket Beispiel

Die folgenden Programme arbeiten wie ein einfaches `chat`-Programm. Der Server liest Botschaften von den Clients und schreibt ihnen wieder. Er läuft im Vordergrund und gibt beim Start seinen Port an, an den ein Kunde sein `bind` machen sollte.

Dieser Client, der in einem anderen Fenster des selben Rechners oder auf einem anderen Rechner laufen sollte, wird gestartet mit den Kommandozeilenargumenten Hostname und Portnummer des anderen Rechners, mit dem er kommunizieren soll.

Danach kann der Anwender auf Clientseite Nachrichten eingeben, die auf Serverseite ausgegeben werden, und umgekehrt. Der Kundenprozeß beendet sich, wenn der Benutzer ^D eingibt. Den Server, der in einer Enlosschleife läuft, muß mit `kill` explizit gestoppt werden.

```
/*
 *   Program 10.10 - SERVER -
     Internet Domain - connectionless
 */
#include „local.h“
main( void ) {
 int        sock, n;
 size_t     server_len, client_len;
 struct sockaddr_in server, /* Address structures */
            client; /* create the SOCKET */
 if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror(„SERVER socket „);
    exit(1);
```

```
 }                /* set svr adr info */
 server.sin_family = AF_INET; /* Address family */
 server_len=server.sin_len = sizeof(server);
  /* SW corr */
 server.sin_addr.s_addr = htonl(INADDR_ANY);
                  /* use any address */
 server.sin_port = htons(0); /* pick a free port */
                  /* BIND the socket */
 if (bind(sock, (struct sockaddr *) &server,
              sizeof(server) ) < 0) {
    perror(„SERVER bind „); exit(2);
 }                /* obtain chosen adr */
 if (getsockname(sock, (struct sockaddr *) &server,
                &server_len) < 0) {
    perror(„SERVER getsocketname „); exit(3);
 }                /* display port # */
 printf(„Server using port %d\n“,
ntohs(server.sin_port));
 while ( 1 ) {
  client_len = sizeof(client); /* estimate length */
  memset(buf, 0, BUFSIZ); /* clear the buffer */
  if ((n=recvfrom(sock, buf, BUFSIZ, 0,
                  /* the clnt message */
    (struct sockaddr *) &client, &client_len)) < 0){
      perror(„SERVER recvfrom „);
      close(sock); exit(4);
  }
  write(fileno(stdout), buf, n); /* show msg to server */
  memset(buf, 0, BUFSIZ); /* clear the buffer */
  if (fgets(buf, BUFSIZ, stdin) != NULL ){
                  /* get server’s msg */
    if ((sendto(sock, buf, strlen(buf) ,0,
                  /* send it to client */
      (struct sockaddr *) &client, client_len)) <0){
        perror(„SERVER sendto „);
        close(sock); exit(5);
    }
  }
 }
}
```

Zur Erinnerung: in der Internetdomain wird eine Verbindung durch ein Fünftupel dargestellt:

> Protokoll, lokale Adresse, lokaler Port,
> entfernte Adresse, entfernter Port

Im Server wird ein Socket angelegt mit Adressfamilie
AF_INET und - wegen Protokoll 0 - Standardvorbelegung UDP. Es folgt die Adressangabe des Servers.

INADDR_ANY ist ein Jokersymbol und bestimmt, daß der Server an jeder gültigen Adresse Botschaften empfangen kann. Mit Portnummer 0 sucht sich das System die erste freie Portnummer mit Wert größer IPPORT_USERRESERVED aus. Mit getsockname wird dieser zur Anzeige gebracht, damit Clients ihn wählen können.

Der Server startet dann seine Endlosschleife, die den Puffer löscht und einen `recvfrom`-Aufruf absetzt, der den Server blockiert, bis eine Botschaft eintrifft. Wird eine Botschaft empfangen, ist darin die Absenderangabe enthalten.

Mit `fgets` holt sich der Server die Anwenderantwort und, wenn eine nichtleere Antwort vorliegt, sendet er diese mit `sendto` an den Client.

Das Programm des Clients ist ganz analog und wird unten gezeigt.

```
/* Program 10.11 - CLIENT-Internet Domain -
connectionless */
#include „local.h“
main(int argc, char *argv[]){
 int          sock, n;
 size_t       server_len;
 static struct sockaddr_in /* Address structures */
              server,
              client;
 struct hostent *host; /* For host info */
 if ( argc < 3 ) { /* need server & port */
  fprintf(stderr, „usage: %s server port_#\n“, argv[0]);
  exit(1);
 }
 if (!(host=gethostbyname(argv[1]))){
  /* get server info */
```

```
 perror(„CLIENT gethostname „); exit(2);
}                    /* set svr adr info */
server.sin_family = AF_INET; /* address family */
memcpy(&server.sin_addr, host->h_addr, host->h_length);
                     /* act adr */
server.sin_port = htons(atoi(argv[2]));
                     /* @ passed in port # */
                     /* create a SOCKET */
if ((sock=socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
 perror(„CLIENT socket „); exit(3);
}                    /* set clnt adr info */
client.sin_family = AF_INET; /* address family */
client.sin_len = sizeof(client); /* SW corr */
client.sin_addr.s_addr = htonl(INADDR_ANY);
                     /* use any address */
client.sin_port = htons( 0 ); /* pick a free port */
                     /* BIND the socket */
if (bind(sock, (struct sockaddr *) &client,
                  sizeof(client)) < 0) {
 perror(„CLIENT bind „); exit(4);
}
while( fgets(buf, BUFSIZ, stdin) != NULL ){
                    /* get clnt's msg */
 server_len=sizeof(server); /* guess at length */
                    /* send msg to server */
 if (sendto( sock, buf, strlen(buf), 0,
      (struct sockaddr *) &server, server_len) < 0 ){
   perror(„CLIENT sendto „);
   close(sock); exit(5);
 }
 memset(buf,0,BUFSIZ); /* clear the buffer */
                     /* server's message */
 if ((n=recvfrom(sock, buf, BUFSIZ, 0,
   (struct sockaddr *) &server, &server_len)) < 0){
   perror(„CLIENT recvfrom „);
   close(sock); exit(6);
 }
 write(fileno(stdout),buf,n); /* show msg to clnt */
 memset(buf,0,BUFSIZ); /* clear the buffer */
}
close(sock);
exit(0);
}
```

Die folgende Abbildung zeigt einen Dialog mittels zweier Fenster auf dem selben Rechner.

Übung 10–3

Gray schlägt vor, den Dialog mit zwei Clients in zwei weiteren Fenstern zu führen. Dazu müßte der Code aber modifiziert werden, damit jeder Client einen ID (handle) mitschickt, der seine Botschaften identifiziert.

Gray stellt zurecht heraus, daß die oben vorgestellte Anwendung zu starr ist. Insbesondere kann man einen Server nicht blockieren, nur weil er auf die „falsche“ Nachricht eines vielleicht abgestürtzten Clients wartet. Gray stellt deshalb I/O-Multiplexing mit dem `select`-Kommando vor und bespricht den Gebrauch von Prioritätsbotschaften und der Voraus-

schau auf Botschaften, die noch in der Warteschlange stehen. Wir übergehen diese Themen und schließen Sockets hiermit ab.

11 Threads

11.1 Einleitung

Das letzte Kapitel in Gray umfaßt ca. 100 Seiten über Threads. Threads (Fäden der Ausführung) ähneln Prozessen, d.h. sie sind in Abarbeitung befindliche Programmstücke, jedoch arbeiten mehrere Threads innerhalb des selben Adressraums und innerhalb einer Prozeßumgebung. Sie teilen sich demnach die Ressourcen und empfangen „alle“ die gleichen Signale.

Sie eignen sich besonders für Anwendungen, in denen nebenläufige Aktivitäten innerhalb **einer** Anwendung stattfinden, z.B. die Bildschirmgestaltung mit mehreren Fenstern, das Aufsammeln von Eingaben mehrerer Kassenautomaten, die parallele Ausführung von Datenbankabfragen, usw.

Da es sich um „befreundete“ nebenläufige Aktivitäten handelt, kann die Synchronisierung und Prioritätssteuerung vereinfacht werden. Speziell kann auf die teure Prozessorumschaltung zwischen mehreren Prozessen (context switch) mit Laden/Speichern der Registerinhalte und des Prozeßkontrollblocks verzichtet werden. Auch kann relativ billig ein neuer Thread erzeugt werden und auf seine Beendigung reagiert werden, d.h. das teure `fork`/`exec` und `exit`/`wait` in UNIX entfällt.

Speziell für das Client-Server-Modell, etwa in einer Streamsocket Anwendung mit mehreren Clients (mehrfaches `accept` beim Server), kann man sich gut vorstellen, daß jede der Verbindungen von einem Thread betreut wird. Klar ist allerdings auch, daß der Zwang, sich zu synchronisieren, nicht entfällt. Als Synchronisierungsmodell dient häufig der sog. Hoaresche **Monitor** mit *Entryprozeduren*, *wait*/*signal* und *condition*

variables, der in einer Art von Objektorientierung Daten und Methoden in einem Objekt (dem Monitor) kapselt und den wechselseitigen Ausschluß regelt.

Zwei offensichtliche Probleme mit Threads seien auch gleich erwähnt. Ein Thread einer größeren Anwendung wird auch immer Funktionen einer Bibliothek verwenden. Bei der quasi-gleichzeitigen Abarbeitung von Threads kann es dann auch zu Mehrfachaufrufen von Bibliotheksfunktionen kommen, was zu Fehlern führen kann, wenn diese globale Variablen verwenden. Bibliotheken, die diese Verwendung ausschließen, heißen demnach **thread-safe**. Das ist prinzipiell stärker als die Erklärung, die Bibliothek sei reentrant, da wir das problemlose Arbeiten mehrerer Inkarnationen innerhalb **eines** Prozesses garantieren.

Zweitens muß das Blockieren eines Threads, bzw. des Prozesses, neu betrachtet werden, etwa wenn ein Thread eine `read`-Anweisung ausführt und auf die Eingabe wartet. Offensichtlich wäre ein Blockieren des umgebenden Prozesses wenig sinnvoll, da andere Threads durchaus weiterarbeiten könnten.

Für die Realisierung gibt es zwei Modellausrichtungen und Mischformen. Zum einen können Threads in einem **user-level** Modell laufen, bei dem die Threads vollständig in einer Bibliothek implementiert sind. Das BS selbst unterstützt Threads nicht. Die gerade genannten `read`-Routinen und ähnliche blockierende Aufrufe werden in umhüllende Funktionen verpackt, die blockierende Systemaufrufe und Kontextwechsel vermeiden. Der zusätzliche Systemaufwand bei dieser Realisierung ist gering, die Funktionalität kann gut erweitert werden. Nachteilig ist, daß diese Form der Threads reale Parallelität mit Mehrprozessorsystemen nicht ausnutzen kann.

Dies ist möglich beim **kernel-level** Modell, bei dem das Betriebssystem von der Existenz der Threads weiß. Diese Realisierung ist aufwendiger, wenngleich immer noch billiger als mehrfache Prozesse. Threadfunktionalität wird über Systemaufrufe angeboten, die im BS-Kern implementiert ist.

Bei Mischformen werden user-level Threads auf kernel-level Threads entweder 1:1 (Gray: Windows NT und OS/2) oder *n*:1 oder *n*:*m* (Sun Solaris) abgebildet. Bei den *n*:*m* Ansätzen bilden die kernel-level Threads einen Pool, aus dem sich die user-level Threads bedienen. Diese kernel-level Threads nennt man auch **Lightweight Processes** (LWPs, leichtgewichtige Prozesse). Die wichtigste und bekannteste Threadbibliothek ist die für den POSIX Standard, entsprechend heißen sie POSIX Threads und ihre Funktionen beginnen mit dem Präfix `pthread`. Ihre Beschreibung findet sich auf den **3t** Handbuchseiten. Wir wollen im folgenden einige dieser Threadfunktionen an Beispielen kennenlernen.

11.2 Einen Thread anlegen

Jeder Prozeß besitzt einen Haupt- oder Startthread, der vom BS angelegt wird, wenn der Prozeß startet. Mit der Bibliotheksfunktion `pthread_create` (man beachte das „e“!!) können weitere Threads hinzugefügt werden, die dann mit den innerhalb des Prozesses bereits existierenden Threads kooperieren müssen.

Include File(s)	<pthread.h>		Manual	**3t**
Summary	int pthread_create (pthread_t *new_thread_ID, const pthread_attr_t *attr, void * (*start_func) (void *), void *arg);			
Return	Success	Failure	Sets `errno`	
	0	non-zero		

Tab. 11–1 *pthread_create Bibliotheksfunktion*

Die Funktion hat vier Pointerargumente. Das erste ist ein `pthread_t` getypter Pointer, der eigentlich auf eine vorzeichenlose ganze Zahl zeigt (unsigned integer), den Thread-identifikator, der **innerhalb des Prozesses** eindeutig ist.

Das zweite Argument `*attr` ist ein Zeiger auf eine dynamisch angelegte Struktur, die wiederum einen `void`-Zeiger enthält In AIX sieht diese struct wie folgt aus.

```
/* ------------------------------------------------
 * Pthread Attributes
 */
typedef struct __pt_attr *pthread_attr_t;
extern pthread_attr_t pthread_attr_default;
```

Die Attribute bestimmen das Verhalten der Threads, ihre Stapelgrößen, Ablaufpolitik (scheduling policy), usw. Wird der Zeiger mit NULL vorbesetzt, verwendet das System die Standardvorbelegung. Will man selbst andere Werte setzen, muß dies vorab mit der Funktion `pthread_attr_init` vornehmen.

Das dritte Argument ist ein Zeiger auf eine Funktion, die den Code enthält, der als Thread abgearbeitet werden soll. Die Funktion sollte einen `void`-Zeiger abliefern und einen `void`-Zeiger als Argument aufnehmen.

Die eigentlichen Argumente der Funktion, sofern nötig, werden als 4. Argument übergeben, wobei im Falle mehrerer aktueller Parameter, diese vorab in eine **statisch angelegte** Struktur zu packen sind, auf die ein Zeiger als Argument übergeben wird.

Der Grund für diese Trennung von Code und Parametern ist, daß Parameter einer Funktion lokale Variablen sind, die auf dem Laufzeitstapel abgelegt werden und nach Verlassen der Prozedur und außerhalb der Prozedur undefiniert sind.[1]

Der erzeugte Thread hat seine eigenen Attribute und seinen eigenen Laufzeitstapel. Er erbt seine Signalmaske und Priorität von dem aufrufenden Programm. Er erbt keine noch anhängigen Signale und kann ggf. eigenen Speicher anfordern.

---

1. Gray drückt sich wenig genau aus: „As would be anticipated, locally allocated objects reside on the stack, and their value is undefined when we leave their scope.“ Das Variablen ungültig werden, wenn man ihren Gültigkeitsbereich verläßt, ist tautologisch.

Bei Erfolg wird `pthreat_create` Null zurückliefern, sonst einen Fehlercode, z.B. EINVAL (22) für ungültige Attribute. Threads setzen in der Regel **nicht** die `errno` Variable mit Ausnahme der Fälle, in denen in Threads aufgerufene Systemfunktionen bei aufgetretenen Fehlern `errno` mit einem Wert belegt wird.

Ein neugeschaffener Thread beginnt seine Arbeit mit der ersten Anweisung der als Argument angegebenen Funktion und führt diese fort bis

- die Funktion implizit oder explizit fertig ist
- ein Aufruf von `pthread_exit` erfolgt
- der Thread mit einem Aufruf von `pthread_cancel` abgebrochen wird
- der Prozeß, der den Threat implizit oder explizit angelegt hat, aufhört
- einer der Threads ein `exec` ausführt.

11.3 Einen Thread verlassen

Ein `pthread_exit` Bibliotheksaufruf beendet einen Thread ganz ähnlich wie ein `exit` einen Prozeß terminiert.

Include File(s)	<pthread.h>		Manual	**3t**
Summary	void pthread_exit (void *status);			
Return	Success	Failure	Sets `errno`	

***Tab. 11–2** pthread_exit Bibliotheksfunktion*

Das einzige Argument ist ein Zeiger auf einen Statuswert. Dieser Verweis wird zurückgeliefert, wenn ein **non-detached thread** verlassen wird. Der Statuswert und der Thread-ID werden aufgehoben für den Thread, der auf

den beendeten warten will (einen Join machen will, siehe `pthread_join()` unten).

Der Begriff **non-detached** (=**joinable**) für einen Thread bezieht sich auf die Möglichkeit, durch Aufruf von `pthread_detach` einen Thread (meist sich selbst) „abzuhängen“, d.h. zu terminieren und seine Ressourcen freizugeben. Dies entspricht der Beendigung eines Daemonprozesses; es ist nicht möglich, auf einen Thread zu warten (`pthread_join()`), der bereits *detached* ist.

Wenn die Funktion, die innerhalb eines Threads ausgeführt wird, zurückkehrt (durch Ausführung der letzten Anweisung oder per `return`), ruft das System implizit auch `pthread_exit` auf.

11.4 Einfaches Thread Management

Wurde ein Thread angelegt, kann man den erzeugenden Prozeß anweisen, zu warten bis der Thread fertig ist. Dies wird mit der `pthread_join` Bibliotheksfunktion erreicht.

Das erste Argument ist ein gültiger, von `pthread_create` zurückgelieferter Thread-ID. Der angegebene Thread muß mit dem aufrufenden Prozeß assoziiert sein und sollte nicht als *detached* spezifiziert sein. Das zweite Argument, `**status`, zeigt auf eine statische Speicherzelle, in die das Abschlußergebnis gespeichert werden soll, das `pthread_exit` übergeben bekommt oder das geliefert wird, wenn der Funktioncode return ausführt. Ist das Argument NULL, wird die Statusinformation weggeschmissen.

Include File(s)	<pthread.h>		Manual	**3t**
Summary	int pthread_join (pthread_t target_thread, void **status);			
Return	Success	Failure	Sets `errno`	
	0	non-zero		

Tab. 11–3 *pthread_join Bibliotheksfunktion*

Einige Hinweise sind im Zusammenhang mit `pthread_join` angebracht. Auf einen Thread sollte nur ein einziger anderer Thread warten. Der wartende Thread muß nicht der erzeugende sein. Warten mehrere Threads auf einen Thread, erhält nur einer die korrekte Statusinformation zurück. Die anderen „joins“ liefern einen Fehler ab.

Wird der Thread, der join aufgerufen hat, entfernt (cancelled), kann ein anderer Thread mit join den ausstehenden Thread empfangen. Wird der in `pthread_join` genannte Thread beendet bevor der Aufruf selbst erfolgt ist, kehrt der Aufruf sofort zurück.

Zuletzt gilt, daß ein non-detached Thread (also die Standardvorgabe), auf den kein anderer Thread mit `pthread_join` wartet, bei Beendigung seine Ressourcen nicht freigibt; dies geschieht dann erst mit Beendigung des Prozesses, in dem der Thread vereinbart war.

Zurückgelieferte Fehlercodes ESRCH (3) deuten darauf hin, daß kein undetached Thread für den angegebenen Thread-ID gefunden werden konnte. Ein Wert EDEADLK (45) zeigt eine Deadlocksituation an.

Der Join-Vorgang ähnelt dem Warten auf einen mit `fork` abgezweigten Prozeß. Andererseits kann sich ein Thread - anders als ein Prozeß - mit einem einzigen Bibliotheksaufruf abhängen. Beendigt sich ein abgehängter Thread, werden seine Ressourcen automatisch an das System zurückgegeben.

Include File(s)	<pthread.h>		Manual	**3t**
Summary	int pthread_detach (pthread_t thread-ID);			
Return	Success	Failure	Sets `errno`	
	0	non-zero		

Tab. 11–4 *pthread_detach Bibliotheksfunktion*

Einziges Argument ist ein Thread_ID. Bei Erfolg wird der Thread abgehangen und der Aufruf liefert 0 zurück. Scheitert der Aufruf, wird er nicht

abgehangen und liefert z.B. EINVAL (22) für einen bereits abgehangenen Thread oder ESRCH (3) für einen unbekannten Thread-ID zurück.

Stevens [Stev98, S. 604] erwähnt, daß der häufigste Aufruf die Form `pthread_detach(pthread_self())` hat.

Das folgende, von Stephan Wilke für AIX angepaßte Programm 11.1 zeigt `pthread_create` und `pthread_join`.

```
/***
 * P11.1.c: Generating and joining threads
 * Compile: % cc_r p11.1.c -o p11.1 -lpthreads
 ***/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#define MAX 5
#define TRAND(limit,n) {struct timeval t;\
   gettimeofday(&t,(void *)NULL);\
   (n)=rand_r((unsigned *) &t.tv_usec) % (limit)+1;}

extern char * sys_errlist[];

void * say_it( void *);
int main(int argc, char *argv[]) {
 int   i,rc;
 pthread_t thread_id[MAX];
 int   status, *p_status = &status;
 pthread_attr_t attr; /* SW, Anpassung an AIX 4.2 */

 if ( argc > MAX+1 ) /* check arg list */
  fprintf(stderr,“%s arg1, arg2, ... arg%d\n“, *argv,
MAX ),
   exit( 1 );

 printf(„Displaying\n“);
 /* SW, Anpassung an AIX 4.2,
 das Attribut muß mit nachstehenden Parameter
 PTHREAD_CREATE_UNDETACHED initialisiert werden, um das
 gewüschte Verhalten in pthread_join zu erhalten
 */
```

```
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_UNDETACHED);

 for (i = 0; i < argc - 1; ++i) { /* generate threads */
    rc=pthread_create(&thread_id[i],
    &attr, /* SW, im Orginal hier NULL */
    say_it,
    (void *)argv[i+1]);
  if (rc != 0) {
    fprintf(stderr, „pthread_create failure: \
    %d [%s]\n“, rc, sys_errlist[rc]);
    exit( 2 );
  }
 }
 pthread_attr_destroy(&attr);
  /* SW, Anpassung an AIX 4.2 */

 for (i = 0; i < argc - 1; ++i){
  /* wait for each thread */
  if ( pthread_join(thread_id[i],
            (void **) p_status) != 0 )
    fprintf(stderr, „pthread_join failure\n“),
              exit( 3 );
  printf(„\nThread %X returns %d“, thread_id[i],
status);
 }
 printf(„\nDone\n“);
 pthread_exit (NULL);
}

/***
 * Display the word passed in a random # of times
 ***/

void * say_it(void *word) {
 int i, numb;
 TRAND(MAX, numb);
 printf(„\n%d %s „, numb, word);
 for (i = 0; i < numb; ++i) {
  sleep(1);
  printf(„%s „, word);
 }
 return (void *) numb;
}
```

Das Programm möchte bis zu 5 Argumente und erzeugt für jedes der Argumente einen Thread, dessen ID in dem Vektor thread_id gespeichert wird. Abweichend von Gray darf als 2. Argument nicht NULL angegeben werden. Die im Thread auszuführende Funktion ist `say_it`, die ein Wort aus der Argumentliste als aktuellen Parameter bekommt.

Diese Funktion `say_it` gibt das Argument in einer zufälligen Anzahl von Ausgaben aus und kehrt dann zurück, wo der erzeugende Prozeß mit `pthread_join` wartet.

Die Ausgabe eines Laufs des so angepaßten Programms ergibt die folgende Ausgabe:

```
wegner@elsie(chpt11)$ ./p11.1 Ich und Du Muellers Kuh
Displaying

1 Muellers
2 Kuh
2 und
4 Du
2 Ich und Muellers Kuh Du Ich und Kuh Du Ich
Thread 2001C6FC returns 2
Thread 2001C5FC returns 2Du Du
Thread 2001C4FC returns 4
Thread 2001C3FC returns 1
Thread 2001C2FC returns 2
Done
```

Hinweis: Die zufällige Anzahl der Wortwiederholung ändert sich natürlich bei jedem Aufruf. Die Scheduling-Politik beim zweiten und den folgenden Aufrufen ist aber anders wie oben und erscheint fest:

```
wegner@elsie(chpt11)$ ./p11.1 Ich und Du Muellers Kuh
Displaying

3 Ich
2 und
4 Du
4 Muellers
2 Kuh Ich und Du Muellers Kuh Ich und Du Muellers Kuh
Ich
Thread 2001C6FC returns 3
```

```
Thread 2001C5FC returns 2Du Muellers Du
Thread 2001C4FC returns 4Muellers
Thread 2001C3FC returns 4
Thread 2001C2FC returns 2
Done
```

11.5 Thread Attribute

Im Beispiel oben haben wir bereits `pthread_attr_init` benutzt, um ein Attributsobjekt zu initialisieren, das wir in `pthread_create` übergeben können. Der Aufruf besetzt das als Argument übergebene Objekt mit den Standardvorgaben.

Include File(s)	<pthread.h>		Manual	**3t**
Summary	int pthread_attr_init (pthread_attr_t *attr);			
Return	Success	Failure	Sets `errno`	
	0	non-zero		

Tab. 11–5 *pthread_attr_init Bibliotheksfunktion*

Nach Anlegen des Attributobjekts können einzelne Attribute mit den entsprechenden set-/get-Funktionen abgerufen und neugesetzt werden. Die Belegung bestimmt das Threadverhalten beim Anlegen des Threads. Nachträgliche Veränderungen an dem Objekt beeinflußen den Thread nicht mehr. Entsprechend kann ein Objekt die Initialisierung mehrerer Threads steuern. Die Attribute und ihre Vorbesetzung sind in der Tabelle unten festgehalten.

Attribut	Vorbesetzung	Kommentar
contentions-cope	PTHREAD_SCOPE_PROCESS	Der Thread streitet uml Ressourcen innerhalb des Prozesses . Er gilt als unbound = nicht an speziellen LWP gebunden
detachstate	PTHREAD_CREATE_JOINABLE	Ein non-detached Thread

Attribut	Vorbesetzung	Kommentar
stackaddr	NULL	System legt Stapel an
stacksize	NULL	System gibt Größe vor (Solaris 1 MB); Thread Stapel wachsen nicht dynamisch
priority	-	Hat Priorität des aufrufenden Threads
policy	SCHED_OTHER	System bestimmt Scheduling
inheritsched	PTHREAD_EXPLICIT_SCHED	Scheduling explizit, bestimmt durch Attributobjekt

***Tab. 11–6** Thread Attribute*

Hinweis: Im Beispielprogramm oben hat Stephan den `detachstate` auf PTHREAD_CREATE_UNDETACHED gesetzt, weil sonst die Rückkehr der Threads nicht gemeldet werden kann. Entgegen den Angaben in Gray (Vorbesetzung ist **joinable**) ist in AIX (POSIX-konform?) die Vorbesetzung PTHREAD_CREATE_DETACHED, was laut `pthread.h` auf unserem System gleichzusetzen ist mit dem Wert 1:

```
/* detach state
*/
#define PTHREAD_CREATE_DETACHED 1
#define PTHREAD_CREATE_UNDETACHED 0
#define DEFAULT_DETACHSTATE PTHREAD_CREATE_DETACHED
```

Übung 11–1

Lassen Sie Programm 11.1 mit PTHREAD_CREATE_DETACHED laufen!

11.6 Thread Ablaufsteuerung (Scheduling)

Die Ablaufsteuerung haben wir oben ausgeklammert. Mit ihr läßt sich die Reihenfolge der Threadabläufe steuern. Wie aus der Betriebssystemvorlesung bekannt, gibt es verschiedene Strategien hierfür:

- FCFS (first come first serve), auch als FIFO bekannt
- SJF (shortest job first)
- prioritätsgesteuert
- RR (round robin)

Generell muß man noch zwischen unterbrechbarem und nicht-unterbrechbarem Mehrprogrammbetrieb unterscheiden (**preemptive** vs **non-preemptive**). Ein BS mit einer non-preemptive Strategie kann einem Prozeß nur dann die Kontrolle (den Prozessor) entziehen, wenn der Prozeß sich beendet oder auf eine blockierende Anweisung, z.B. ein `read`, gestoßen ist.

Wie für Prozesse gibt Gray das magische Dreieck der Threadumschaltung an, wobei wir uns erlaubt haben, die Pfeilrichtungen für *preempt* und *dispatch* richtigzustellen.

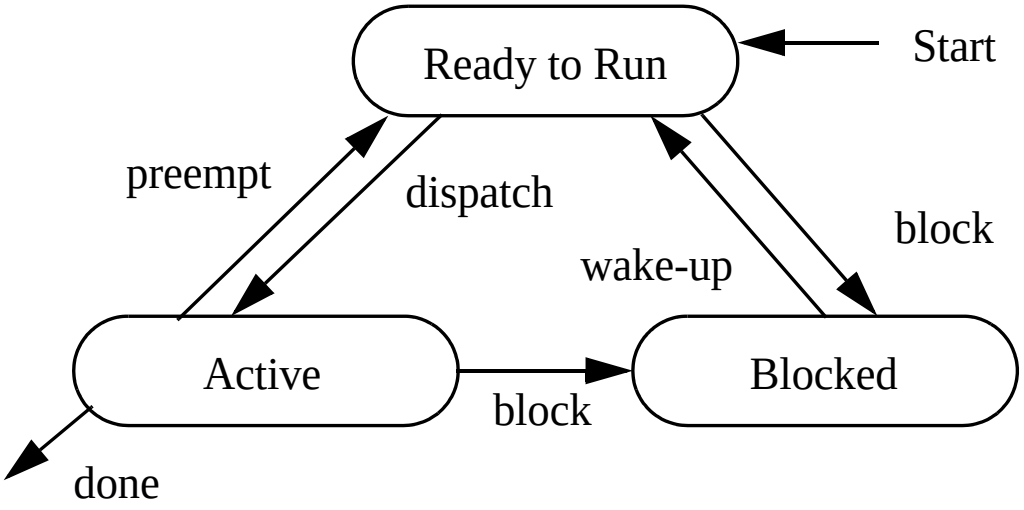


Abb. 11–1 Magisches Dreieck der Threadumschaltung

Statt „active“ ist auch „running“ und statt „blocked“ ist auch „waiting“ oder „sleeping“ üblich.

Scheduling von Threads wird zusätzlich bestimmt durch die Zuordnung zu den LWPs: ein Thread heißt gebunden (**bound**), wenn er 1:1 auf einen LWP abgebildet ist, der natürlich mit seiner Zuteilungsstrategie die des Threads bestimmt. Ist der Thread **unbound**, bedient er sich mit einer

der oben genannten Strategien aus dem Pool an leichtgewichtigen Prozessen.

Ist PTHREAD_SCOPE_SYSTEM gesetzt, ist der Thread 1:1 gebunden. Ist dagegen PTHREAD_SCOPE_PROCESS gesetzt, konkurriert der Thread im Pool um Ressourcen. In letzterem Fall kann man jetzt noch die Politik für die Prozessorzuteilung angeben:

- SCHED_OTHER - die Voreinstellung; meist eine Zeitscheibensteuerung
- SCHED_FIFO - first in first out; nimmt als nächsten den Prozeß mit der höchsten Priorität und darunter den am längsten wartenden Prozeß.
- SCED_RR - ähnlich wie FIFO mit einem zusätzlichen Zeitscheibenfaktor .

Die Besprechung der `pthread_setschedparam` Funktion schenken wir uns.

11.7 Die Verwendung von Signalen in Threads

Signale gehen an Prozesse. Bei sog. synchronen Signalen, z.B. SIGSEGV (Adreßverletzung) oder SIGFPE (Division durch Null) geht das Signal an den Thread, der den Fehler verursachte, was Sinn macht.

In asynchronen Fällen, z.B. bei einem SIGINT (Interrupt etwa durch kill), ist dies nicht gegeben. Welcher Thread das Signal empfängt, läßt sich nicht voraussagen und hängt davon ab, ob ein Threads das Signal maskiert (ausgeschaltet) hat.

Wir übergehen die Signalverarbeitung für Threads mit `pthread_kill`, `pthread_sigmask` und `sigwait`.

11.8 Thread Synchronisation

Programme, die auf Threads aufbauen, benötigen in aller Regel Synchronisationsmechanismen, um kritische Abschnitte zu schützen. Dazu wer-

den zwei bekannte Methoden angeboten, die den Semaphoren ähnlichen **Mutexvariablen** (**Mutex-Sperren**) und die aus den Hoarschen Monitoren bekannten **Bedingungsvariablen** (**condition variables**).

11.8.1 Mutex Variablen

Die Methode, auch einfach Mutex oder **mutex lock** genannt, wirkt wie ein Semaphor und verhindert bei Werten ungleich 0 den Zugang (locked) und erlaubt bei Werten gleich 0 den Zutritt in den kritischen Abschnitt. Diese Semantik ist im übrigen gegenüber Semaphoren gerade verdreht (P(S) für S.condition = 0 blockiert).

Die Implementierung dieser Mutex-Operationen sichert, wie bei Semaphoren, atomares Testen-und-Setzen zu, verhindert aber **nicht**, daß ein Thread eine Sperre aufhebt, die ein anderer gesetzt hat.

Das übliche Verhalten ist, daß Threads, die eine Sperre nicht erlangen konnten, warten.

Fischli und Goetz in den Offenen Systemen (1998) 7:4-12, beschreiben sehr kurz und gut lesbar, wie der Gebrauch aussieht:

- Vor Gebrauch muß mit `pthread_mutex_init` das Mutex initialisiert werden, wobei wieder, wie bei `pthread_create`, ein Zeiger auf ein Attributobjekt übergeben werden kann;
- Mit `pthread_mutex_lock` wird eine Mutexsperre erlangt und mit `pthread_mutex_unlock` wieder freigegeben. Mit `pthread_mutex_trylock` kann man die Sperre eines Mutex prüfen, ohne bei bereits gesetzter Sperre blockiert zu werden.
- Wird ein Mutex nicht mehr benötigt, kann es mit `pthread_mutex_destroy` zerstört werden.

Günstig ist auch, daß die Bibliothek einen Fehler signalisiert, wenn ein Thread rekursiv erneut versucht, eine bereits von diesem Thread gehaltene Sperre zu erlangen, was eine Verklemmung verursachen würde. Hierzu führt die Datenstruktur, die das Mutex implementiert, ein Feld für den **Mutexbesitzer** mit.

Die Vorstellung ist, daß der Thread, der die Sperre erlangt hat, Besitzer der Sperre wird und nur er diese wieder freigibt. Dieses Verhalten wird aber nicht erzwungen und macht das Mutex auch im Verhalten anders als das Semaphore, bei dem durchaus verschiedene Prozesse **P**- und **V**-Operationen auf demselben Semaphor ausführen.

Es folgen in Gray zwei Produzent-Konsument-Beispiele, kompiliert zu INTRA und INTER, die Threads und Mutexsperren verwenden, um einmal innerhalb eines Prozesses sich zu koordinieren und zum anderen dies zwischen Prozessen erledigen.

Der Quellcode dazu steht in 11.3A - 11.3D, ist länglich und wird hier nicht aufgelistet. Statt dessen geben wir den kürzeren Code aus Fischli und Goetz für eine Client-Server Bankenautomatanwendung an.

```
#include <pthread.h>

pthread_mutex_t db_mutex;

int main( void )
{
  connection_t *connection;
  pthread_t thread;

  acct_init();
  comm_init();
  pthread_mutex_init( &db_mutex, NULL );

  while ( 1 ) {
    connection = comm_accept();
    pthread_create( &thread, NULL, handle_client,
                connection );
    pthread_detach( thread );
  }
}


void *handle_client( void *connection )
{
  request_t request;
  response_t response;

  while ( comm_receive( connection, &request ) != NULL )
```

```
{
    pthread_mutex_lock( &db_mutex );
    process( &request, &response );
    pthread_mutex_unlock( &db_mutex );
    comm_send( connection, &response );
  }
  comm_close( connection );
  return NULL;
}
```

Im Beispiel wird der Zugriff von mehreren Bankautomaten auf die Kontendatenbank über ein Mutex-Lock synchronisiert.

In einer Erweiterung dieses Beispiels wird die Anzahl der Threads, die der Server gleichzeitig bedienen kann, begrenzt. Dazu wird der Konstrukt der Condition-Variablen eingeführt.

11.8.2 Condition Variablen

In der Definition des Hoarschen Monitors ist eine Condition Variable *c* eine Warteschlangen, in die sich Prozesse mit **wait(c)** einreihen und aus der sie andere Prozesse mit **signal(c)** wieder befreien.

Fischli/Goetz schreiben:
„Die Funktion `pthread_cond_init()` initialisiert eine Condition-Variable, wobei ein Zeiger auf ein Attributobjekt übergeben werden kann. Mit `pthread_cond_wait()` kann dann ein Thread auf der Condition-Variablen warten. Eine Condition-Variable wird immer zusammen mit einem Mutex-Lock verwendet, der die Zugriffe auf die Daten schützt, von denen die entsprechende Bedingung abhängt. Dieser Mutex-Lock, den der wartende Thread vorher erworben haben muß, wird von `pthread_cond_wait()` implizit freigegeben. Dadurch kann ihn ein anderer Thread erwerben, die Daten verändern und dies dem wartenden Thread mit `pthread_cond_signal()` signalisieren. Der wartende Thread wird geweckt, kehrt aber erst aus der Funktion `pthread_cond_wait()` zurück, nachdem er den Mutex-Lock implizit wieder erwerben konnte.

Warten mehrere Threads auf derselben Condition-Variablen, so entscheidet die Schedulingpolitik, welcher Thread geweckt wird. Mit

`pthread_cond_broadcast()` ist es aber möglich, alle Threads zu wecken, die auf derselben Condition-Variablen warten. Wird eine Condition-Variable nicht mehr benötigt, so kann sie mit `pthread_cond_destroy()` zerstört werden.“ [Ende Zitat]

Include File(s)	<pthread.h>		Manual	**3t**
Summary	int pthread_cond_signal (pthread_cond_t *cond); int pthread_cond_broadcast (pthread_cond_t *cond);			
Return	Success	Failure	Sets `errno`	
	0	non-zero		

Tab. 11–7 *Bibliotheksfunktionen zur Benachrichtigung von Condition-Variablen*

Wichtig bei Broadcast ist, daß zwar alle geweckt werden, aber nur einer die Mutex-Sperre erlangen kann. Wie für signal so üblich, verpuffen Aufrufe auf Condition-Variablen, bei denen keine Threads warten. Dies ist verschieden von einer falschen Adreßangabe für eine Condition-Variable. Ein solcher Aufruf mit einer ungültigen Variablenadresse produziert eine Fehlermeldung. Condition-Variablen können mit entsprechender Initialisierung auch prozeßübergreifend angelegt werden.

Das modifizierte Programm von Fischli/Goetz sieht wie folgt aus.

```
#include <pthread.h>

pthread_mutex_t db_mutex;
pthread_mutex_t count_mutex;
pthread_cond_t count_cond;
int      count;

int main( void )
{
  connection_t *connection;
  pthread_t thread;

  acct_init();
```

```
  comm_init();
  pthread_mutex_init( &db_mutex, NULL );
  pthread_mutex_init( &count_mutex, NULL );
  pthread_cond_init( &count_cond, NULL );
  count = 0;

  while ( 1 ) {
    connection = comm_accept();
    pthread_create( &thread, NULL, handle_client,
                connection );
    pthread_detach( thread );
    increment_count();
  }
}


void *handle_client( void *connection )
{
  request_t request;
  response_t response;

  while ( comm_receive( connection, &request ) != NULL )
{
    pthread_mutex_lock( &db_mutex );
    process( &request, &response );
    pthread_mutex_unlock( &db_mutex );
    comm_send( connection, &response );
  }
  comm_close( connection );
  decrement_count();
  return NULL;
}

void increment_count()
{
  pthread_mutex_lock( &count_mutex );
  count++;
  while ( count == MAX_THREADS )
    pthread_cond_wait( &count_cond, &count_mutex );
  pthread_mutex_unlock( &count_mutex );
}

void decrement_count()
{
  pthread_mutex_lock( &count_mutex );
  count--;
  pthread_cond_signal( &count_cond );
```

```
    pthread_mutex_unlock( &count_mutex );
  }
```

Gray gibt das klassische Readers-Writers Problem mit Condition Variablen an. Dabei wird in einen begrenzten Puffer geschrieben und schreibende Prozesse, die den Puffer voll vorfinden, signalisieren die lesenden Prozesse und machen selbst ein wait. Umgekehrt signalisieren Leser, wenn sie einen leeren Puffer vorfinden.

Das Programm 11.4 hat die folgende Form:

```
#define _REENTRANT
#include <stdio.h>
#include <ctype.h>
#include <pthread.h>
#define MAXx 5
/***
 * These are global
 ***/
pthread_mutex_t lock_it = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t write_it = PTHREAD_COND_INITIALIZER;
typedef enum _bool { false, true } boolean;
typedef struct { /* A small data buffer */
  char   buffer[MAXx];
  int  how_many; /* # of chars in buffer */
} BUFFER;
BUFFER   share = {„“, 0}; /* start empty */
void   *read_some(void *), *write_some(void *);
boolean  finished = false;
void main(void) {
 pthread_t  t_read,
         t_write;
 pthread_attr_t attr; /* SW, Anpassung an AIX 4.2 */

 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr,
         PTHREAD_CREATE_UNDETACHED);
 /* create threads */
 pthread_create(&t_read, &attr, read_some,
  (void *) NULL);
 pthread_create(&t_write, &attr, write_some,
  (void *) NULL);
 pthread_join(t_write, (void **) NULL);
  /* wait for writer */
 pthread_mutex_destroy( &lock_it ); /* clean up */
```

```
 pthread_cond_destroy( &write_it );
 pthread_attr_destroy(&attr); /* SW, Anpassung an AIX
4.2 */
}
/***
 * Code to fill the buffer
 ***/
void *
read_some(void * junk) {
 int   ch;
 printf(„R %X: Starting\n“, pthread_self());
 while (!finished) {
  pthread_mutex_lock(&lock_it);
  if (share.how_many != MAXx) { /* buffer is not full */
     ch = getc(stdin);
     if ( ch == EOF ) { /* end-of-file */
        share.buffer[share.how_many] = NULL;
        share.how_many = MAXx;
        finished = true; /* we are all done */
        printf(„R %X: Signaling done\n“,
               pthread_self());
        pthread_cond_signal(&write_it);
                 /* signal condition var */
        pthread_mutex_unlock(&lock_it);
        break;
     } else {
        share.buffer[share.how_many++] = ch;
        printf(„R %X: Got char [%c]\n“,
                      pthread_self( ),
        isalnum(ch) ? ch : ’#’); /* output pretty */
        if ( share.how_many == MAXx ) { /* if full */
           printf(„R %X: Signaling full\n“,
                pthread_self());
           pthread_cond_signal(&write_it);
        }
     }
  }
  pthread_mutex_unlock(&lock_it);
 }
 printf(„R %X: Exiting\n“, pthread_self());
 return NULL;
}
/***
 * Code to write (display) buffer
 ***/
```

```
void *
write_some(void * junk) {
 int   i;
 printf(„W %X: Starting\n“, pthread_self());
 while (!finished ) {
  pthread_mutex_lock(&lock_it);
  printf(„\nW %X: Waiting\n“, pthread_self());
  while (share.how_many != MAXx) /* while not full */
    pthread_cond_wait(&write_it, &lock_it);
                  /* wait for notify */
  printf(„W %X: Writing buffer\n“, pthread_self( ));
  for( i=0; share.buffer[i] && share.how_many; ++i,
                share.how_many--)
    putchar(share.buffer[i]);
  pthread_mutex_unlock(&lock_it);
 }
 printf(„W %X: Exiting\n“, pthread_self( ));
 return NULL;
}
```

Ein Ablauf dieses Programms sieht wie folgt aus.

```
wegner@elsie(chpt11)$ ./p11.4
W 2001C8BC: Starting

W 2001C8BC: Waiting
R 2001C9BC: Starting
ABCDE            Benutzereingabe 5 Zeichen + CR
R 2001C9BC: Got char [A]
R 2001C9BC: Got char [B]
R 2001C9BC: Got char [C]
R 2001C9BC: Got char [D]
R 2001C9BC: Got char [E]
R 2001C9BC: Signaling full  Reader signalisiert voll
W 2001C8BC: Writing buffer
ABCDE
W 2001C8BC: Waiting
R 2001C9BC: Got char [#]     CR aus Eingabe angezeigt als #
ab
R 2001C9BC: Got char [a]
R 2001C9BC: Got char [b]
R 2001C9BC: Got char [#]
CTRL+D           Benutzer gibt ^D ein für EOF
R 2001C9BC: Signaling done
R 2001C9BC: Exiting
W 2001C8BC: Writing buffer
```

Rest wird ausgegeben

```
ab
W 2001C8BC: Exiting
```

Damit beenden wir die detaillierte Besprechung der Threads.

Weitere Themen dazu in Gray sind

- Read/Write Locks
- Multithread Semaphore
- threadspezifische Daten (Daten, die über Keys mit den Threads verbunden werden können; braucht man besonders in Bibliotheken, wenn man interne Zustände der Threads speichern möchte).
- Fehlersuche in Programmen mit Threads (`gdb` und `dbx` sind **thread aware**)

Fischli/Goetz erwähnen noch

- Scheduling
- den Once-Mechanismus zur einmaligen Initialisierung von Programmteilen unabhängig von der Anzahl der aufrufenden Threads
- Cancellation zum Abbruch eines Threads durch einen anderen
- Signal-Handling (synchron/asynchron).

Interessant ist auch die Kombination von Threads mit Java. Eher abschreckend dagegen die Diskussion zu Multithreading mit Visual Basic 5.0 Enterprise Edition mit dem VB Servicepack 3 wie in c’t 9/98 beschrieben ...

Index

S

T

U

W

X

Literatur

[1] John Shapley Gray, Interprocess Communications in UNIX, Prentice Hall, Upper Saddle River, NJ, 1997, 364 S., 2nd Edition 1998

[2] W. Richard Stevens, UNIX Network Programming, Vol. I: Networking APIs: Sockets and XTI, 2nd Edition, Prentice Hall, Upper Saddle River, NJ, 1998, 1009 S.

[3] Larry L. Peterson and Bruce S. Davie, Computer Networks: A Systems Approach, Morgan Kaufmann, San Francisco, CA, 1996, 552 S.

[4] W. Richard Stevens, UNIX Network Programming, Prentice Hall, Englewood Cliffs, Nj, 1990, 772 S.

[5] W. Richard Stevens, Advanced Programming in the Unix Environment, Addison-Wesley (Professional Computing Series), 1992, 744 S.

[6] Andrew S. Tanenbaum, Computer Networks, 3rd Edition, Prentice Hall, 1996, 848 S.

[7] Marc J. Rochkind, Advanced Unix Programming, Prentice Hall, 1986, 265 S.

[8] Maurice, J. Bach, Design of the Unix Operating System, Prentice Hall, 1986, 471 S.

[9] William Stallings. High-Speed Networks : Tcp/Ip and Atm Design Principles, Prentice Hall (William Stallings Books

on Computer and Data Communications Technology.), 1997, 632 S.

[10] Walter Proebster. Rechnernetze Technik, Protokolle, Systeme, Anwendungen, Oldenbourg, München-Wien, 1998. 432 Seiten, DM 68,--, ISBN 3-486-24540-6 (Folienausarbeitung siehe: `http://www.informatik.tu-muenchen.de/~heinzman/skripten.html`)

[11] K.A.Robbins and S. Robbins, Practical UNIX Programming - A Guide to Concurrency, Communication, and Multithreading , Prentice Hall PTR, NJ, 1996