Vorlesungsbegleiter zu Systemnahe Software SS 2003

Andreas F. Borchert
Universität Ulm, SAI

Inhaltsverzeichnis

Tabellenverzeichnis

Verzeichnis der Programmbeispiele

Verzeichnis der Abbildungen

Kapitel 1

Sichere Programmiertechniken

1.1 Einführung in die Problematik

Systemnahe Software ist in vielen Fällen in Besitz von Privilegien und gleichzeitig im Kontakt mit potentiell gefährlichen Nutzern, denen diese Privilegien nicht zustehen. Daher muß bei der Entwicklung systemnaher Software nicht nur auf die korrekte Implementierung der gewünschten Funktionalitäten geachtet werden, sondern auch auf die umfassende Verhinderung nicht gewünschter Zugriffe. Dazu ist die Kenntnis der typischen Angriffstechniken notwendig und die konsequente Verwendung von Programmiertechniken, die diese zuverlässig abwehren.

Beispiel: Das Werkzeug `pubfile` (siehe Programm 1.1) soll dazu dienen, Dateien im Verzeichnis `pub` unterhalb meines nicht-öffentlichen Heimatkataloges zur Verfügung zu stellen.

Dieses Werkzeug könnte nach der Übersetzung beispielsweise im `tmp`-Verzeichnis für jeden öffentlich zugänglich gemacht werden:

```
cordelia$ id
uid=120(borchert) gid=200(sai)
cordelia$ gcc -Wall -o pubfile pubfile.c
cordelia$ cp pubfile /tmp
cordelia$ cat ~/pub/READ_ME
This is the READ_ME file within my pub directory.
cordelia$ /tmp/pubfile READ_ME
This is the READ_ME file within my pub directory.
cordelia$
```

Im Normalfall bringt das Programm, selbst wenn es öffentlich installiert ist, noch keine besonderen Privilegien für andere Benutzer; d.h. obwohl das Programm dem Benutzer `borchert` gehört, operiert es nicht notwendigerweise mit den Privilegien von `borchert`:

```
cordelia$ id
uid=6201(waborche) gid=230(student)
cordelia$ /tmp/pubfile READ_ME
/home/thales/borchert/pub/READ_ME: Permission denied
cordelia$ cat ~borchert/pub/READ_ME
cat: /home/thales/borchert/pub/READ_ME: Permission denied
cordelia$
```

pubfile.c

```
/*
 * Display files within my pub directory.
 * Usage: pubfile {file}
 * WARNING: This program has several security flaws.
 * afb 2/2003
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>

const int BUFFER_SIZE = 8192;
const char* pubdir = "/home/thales/borchert/pub";

int main(int argc, char** argv) {
   argv++; --argc; /* skip command name */

   while (argc-- > 0) {
      char pathname[BUFFER_SIZE];
      char buffer[BUFFER_SIZE];
      int fd;
      int count;

      strcpy(pathname, pubdir);
      strcat(pathname, "/");
      strcat(pathname, *argv++);

      if ((fd = open(pathname, O_RDONLY)) < 0) {
         perror(pathname); exit(1);
      }
      while ((count = read(fd, buffer, sizeof buffer)) > 0) {
         if (write(1, buffer, count) != count) {
            perror("write to stdout"); exit(1);
         }
      }
      if (count < 0) {
         perror(pathname); exit(1);
      }
      close(fd);
   }
   return 0;
}
```

Programmtext 1.1: Ausgabe nicht frei zugänglicher Dateien mit Sicherheitslücken

Das läßt sich aber ändern, wenn der Eigentümer des Programmes dem Programm das s-bit spendiert. Dabei steht "s" für **setuid**. Konkret bedeutet dies, daß das Programm mit den Privilegien des Programmeigentümers operiert und nicht mit denen des Aufrufers:

```
cordelia$ ls -l /tmp/pubfile
-rwxr-xr-x    1 borchert sai          7523 Feb 25 18:32 /tmp/pubfile
cordelia$ chmod u+s /tmp/pubfile
cordelia$ ls -l /tmp/pubfile
-rwsr-xr-x    1 borchert sai          7523 Feb 25 18:32 /tmp/pubfile
cordelia$
```

Nun klappt es für andere Benutzer:

```
cordelia$ id
uid=6201(waborche) gid=230(student)
cordelia$ /tmp/pubfile READ_ME
This is the READ_ME file within my pub directory.
cordelia$
```

Wir haben nun den Fall, daß das Programm Privilegien besitzt, die der Aufrufer normalerweise nicht hat. Natürlich sollte so ein Programm nicht all seine Privilegien (im Beispiel die Rechte von `borchert`) dem Aufrufer preisgeben. Stattdessen hatte der Autor von `pubfile` die Absicht, daß nur die Dateien aus dem Unterverzeichnis `pub` der Öffentlichkeit zur Verfügung stehen sollen. Wenn es möglich ist, auf andere Dateien zuzugreifen oder gar beliebige Privilegien des Programmeigentümers ausnutzen zu können, dann würden Sicherheitslücken vorliegen.

Die erste Sicherheitslücke ist leicht gefunden. Unter Angabe eines relativen Pfadnamens können beliebige Dateien mit den Rechten des Benutzers `borchert` betrachtet werden:

```
cordelia$ id
uid=6201(waborche) gid=230(student)
cordelia$ /tmp/pubfile ../.ssh/id_rsa
-----BEGIN RSA PRIVATE KEY-----
[...]
-----END RSA PRIVATE KEY-----
cordelia$
```

In diesem Beispiel wird der private RSA-Schlüssel ausgelesen, mit dessen Hilfe möglicherweise ein passwortloser Zugang auf andere Systeme mit den dortigen Privilegien von `borchert` eröffnet werden kann. Gelegentlich funktioniert das sogar auf dem gleichen System. Und hierfür genügte nur ein zu weitreichender Lesezugriff!

Das ist jedoch nicht die einzige Sicherheitslücke. Grundsätzlich können Sicherheitsprobleme dann auftreten, wenn der wohldefinierte Bereich einer Programmiersprache verlassen wird. Leider sind zahlreiche Fehler in C möglich, die weder vom Übersetzer festgestellt werden noch automatisch zur Laufzeit einen Abbruch des Programms bewirken. Zu dieser Kategorie von Fehlern gehört das Verlassen zulässiger Indexbereiche bei Feldern.

In den Zeilen 27 bis 29 des Programms 1.1 wird der lokale Puffer `pathname` gefüllt, ohne auf die Größe des Puffers zu achten. Zwar mag `BUFFER_SIZE` großzügig gewählt sein, aber ein Argument auf der Kommandozeile kann deutlich länger sein. Die Frage ist ganz einfach: Was kann passieren, wenn der Indexbereich verlassen wird? Die Sprachdefinition von C selbst gibt keine Antwort darauf, abgesehen davon, daß das Verhalten dann als "undefiniert" deklariert wird. Bei den gängigen Implementierungen mit einem rückwärts wachsenden Stack besteht die Möglichkeit, die Rücksprungadresse zu modifizieren und damit statt zum Aufrufer zu einem eingeschleusten Code springen zu lassen. Typischerweise kann der Code innerhalb des überlaufenden Puffers untergebracht werden.

1.2 Typische Schwachstellen bei C

In der Programmiersprache C hat es bereits erfolgreiche Einbrüche aufgrund folgender Programmierfehler gegeben:

- Unzureichende Überprüfung von Argumenten beim Eröffnen von Dateien, Ausführen von Kommandos oder anderen Systemaufrufen.
- Fehlende Einhaltung der Index-Grenzen eines Arrays. Gefahr besteht hier sowohl bei Arrays auf dem Stack als auch auf dem Heap (also per `malloc()` beschafft). Gefahr droht hier auch bei beliebten Funktionen der Bibliothek wie `strcpy`, `strcat`, `sprintf` und `gets`.
- Doppelte Freigabe eines Zeigers mit `free()`.
- Benutzung eines Zeigers, nachdem er bereits freigegeben worden ist.
- Weglassen des Formats bei `printf`. Statt `printf(s)` sollte besser `printf("%s", s)` verwendet werden.

Leider ist die Vermeidung dieser Fehler nicht einfach. Selbst bei sicherheitsrelevanter Software wie der ssh (*secure shell*) oder der SSL-Bibliothek (*secure socket layer*) wurden immer wieder neue Fehler bei aufwendigen Untersuchungen des Programmtexts gefunden. Deswegen ist es bei C sinnvoll, bei systemnaher Software auf die Standard-Bibliotheken von C teilweise zu verzichten und stattdessen auf Alternativen auszuweichen, die die Verwendung sicherer Techniken unterstützen.

1.3 Dynamische Zeichenketten in C

Die Unterstützung dynamischer Zeichenketten in C ist nicht sehr ausgeprägt. Zwar ist es leicht möglich, mit `malloc()` ein Array der gewünschten Länge zu erhalten, aber danach gibt es keine zuverlässige Längeninformation mehr. `strlen` ist nur sinnvoll im Falle wohldefinierter Zeichenketten, da es nach dem Nullbyte sucht. Entsprechend haben Standardfunktionen wie `strcpy` oder `sprintf` keine Möglichkeit zu überprüfen, ob genügend Platz für das Ergebnis vorhanden ist. Folglich muß die Abschätzung dem Programmierer im Vorfeld überlassen werden, die dann häufig unterlassen wird oder fehlerhaft ist.

Hinzu kommt, daß das Einlesen beliebig langer Zeichenketten in C relativ umständlich ist, wenn die Länge nicht zu Beginn bekannt ist. Der Programmtext 1.2 zeigt hier eine typische Lösung. Begonnen wird hier mit einem Puffer der Länge `INITIAL_LEN`. Bei jedem neu eingelesenen Zeichen wird in Zeile 21 überprüft, ob der zur Verfügung stehende Platz noch ausreicht. Falls dies nicht der Fall ist, wird in Zeile 23 mit `realloc()` ein vergrößerter Puffer allokiert und (implizit durch `realloc()`) die bisher eingelesenen Daten umkopiert.

Ein Ausweg besteht in der Schaffung einer alternativen Bibliothek für dynamische Zeichenketten in C, die folgende Voraussetzungen erfüllen muß:

readline.c

```
/*
 * Read a string of arbitrary length from a
 * given file pointer. LF is accepted as terminator.
 * 0 is returned in case of errors.
 * afb 3/2003
 */

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

static const int INITIAL_LEN = 8;

char* readline(FILE* fp) {
   size_t len = 0; /* current length of string */
   size_t alloc_len = INITIAL_LEN; /* allocated length */
   char* buf = malloc(alloc_len);
   int ch;

   if (buf == 0) return 0;
   while ((ch = getc(fp)) != EOF && ch != '\n') {
      if (len + 1 >= alloc_len) {
         alloc_len *= 2;
         char* newbuf = realloc(buf, alloc_len);
         if (newbuf == 0) {
            free(buf);
            return 0;
         }
         buf = newbuf;
      }
      buf[len++] = ch;
   }
   buf[len++] = '\0';
   return realloc(buf, len);
}
```

Programmtext 1.2: Einlesen beliebig langer Zeichenketten

- Neben der eigentlichen Zeichenkette muß auch eine Längenangabe vorliegen.
- Bibliotheksfunktionen analog zu `strcpy()` und `strcat()` müssen unterstützt werden. Diese Funktionen müssen entweder die Längenangabe einhalten oder automatisch die Zeichenketten in ihrer Größe anpassen.
- Hinzu kommen Funktionen für die Initialisierung und die Freigabe von Zeichenketten.

Bei der Semantik gibt es zwei grundsätzliche Ansätze:

- Jede Zeichenkette ist in ihrer Repräsentierung unabhängig von allen anderen Zeichenketten und kann daher auch jederzeit frei verändert werden. Dies entspricht der traditionellen Vorgehensweise in C und der `string`-Template-Klasse in C++.
- Jede Zeichenkette ist konstant. Daher kann bei einer Operation analog zu `strcpy()` auf das Kopieren verzichtet werden. Änderungen erfordern hingegen das vorherige Anfertigen von Kopien. Dies entspricht der Vorgehensweise von Java.

Eine C-Bibliothek, die dem ersten Ansatz folgt, wurde von Dan J. Bernstein entwickelt (u.a. für das Qmail-Paket) und später von Felix von Leitner nachprogrammiert, um die Bibliothek unter der GPL (GNU General Public License) zur Verfügung stellen zu können. Zu finden ist sie unter `http://www.fefe.de/libowfat/`.

/usr/local/diet/include/stralloc.h

```
typedef struct stralloc {
  char* s;
  unsigned int len;
  unsigned int a;
} stralloc;
```

Programmtext 1.3: Datenstruktur der **stralloc**-Bibliothek

Der Programmtext 1.3 zeigt die Datenstruktur für Zeichenketten in dieser Bibliothek: `s` verweist auf einen Buffer der Länge `a`, in dem eine Zeichenkette der Länge `len` untergebracht ist. Entsprechend gilt `len` $\leq$ `a`. Der Zeiger `s` darf gleich 0 sein, um eine leere Zeichenkette zu repräsentieren. Zu beachten ist, daß Zeichenketten auch Nullbytes enthalten dürfen und entsprechend auch weder eine Nullbyte-Terminierung sinnvoll wäre noch verwendet wird. Das Zulassen von Nullbytes hat viele Vorteile. Es ermöglicht die Aufnahme binärer Daten und die Verwendung des Nullbytes als geschickter Feldtrenner.

Wichtig ist die korrekte Initialisierung einer Variablen vom Typ `stralloc`. C sieht bei lokalen Variablen keine automatische Initialisierung vor, so daß hier die Initialisierung nicht vergessen werden darf:

```
stralloc sa = {0};
```

Damit wird übrigens nicht nur `sa.s` auf 0 initialisiert, sondern auch gleichzeitig `sa.len` und `sa.a` auf 0 gesetzt.

sareadline.c

```
1  /*
2   * Read a string of arbitrary length from a
3   * given file pointer. LF is accepted as terminator.
4   * 1 is returned in case of success, 0 in case of errors.
5   * afb 4/2003
6   */
7
8  #include <stralloc.h>
9  #include <stdio.h>
10
11 int readline(FILE* fp, stralloc* sa) {
12    if (!stralloc_copys(sa, "")) return 0;
13    for(;;) {
14       if (!stralloc_readyplus(sa, 1)) return 0;
15       if (fread(sa->s + sa->len, sizeof(char), 1, fp) <= 0) return 0;
16       if (sa->s[sa->len] == '\n') break;
17       ++sa->len;
18    }
19    return 1;
20 }
```

Programmtext 1.4: Einlesen beliebig langer Zeichenketten mit der **stralloc**-Bibliothek

Programmtext 1.4 zeigt, wie sich nun das Einlesen beliebig langer Zeilen vereinfachen läßt. In Zeile 12 wird zunächst `sa` mit Hilfe von `stralloc_copys` zu einer leeren Zeichenkette initialisiert. Generell dient `stralloc_copys` dazu, traditionelle nullbyte-terminierte Zeichenketten in C zu einem `stralloc`-Objekt zu kopieren. Nicht vergessen werden sollte die Überprüfung der Rückgabewerte. Bei 1 war die Operation erfolgreich, bei 0 konnte nicht genügend Speicher allokiert werden.

Die `for`-Schleife ab Zeile 13 behandelt das zeichenweise Einlesen, bis entweder das Zeilenende erkannt wird oder ein Fehler auftritt. In Zeile 14 wird mit `stralloc_readyplus` dafür gesorgt, daß in `sa->s` mindestens ein Byte mehr Platz vorhanden ist als die augenblickliche Länge `sa->len` beträgt. Wenn dies sichergestellt ist, kann in Zeile 15 genau an dieser Stelle versucht werden, ein weiteres eingelesenes Zeichen abzulegen. Handelt es sich dabei um einen Zeilentrenner, bricht Zeile 16 die `for`-Schleife ab. Ansonsten wird in Zeile 17 das Zeichen akzeptiert, indem die Länge der Zeichenkette um 1 erhöht wird.

Ähnlich wie die Lösung 1.2 wird auch bei der **stralloc**-Bibliothek darauf geachtet, daß Zeichenketten nicht etwa byteweise verlängert werden, sondern in etwas größeren Häppchen. Dan J. Bernstein verwendet hier folgende Heuristik: `stralloc_readyplus` bzw. `stralloc_ready` verändern nichts, wenn die aktuelle Länge ausreicht. Muß aber verlängert werden, dann wird die neue Länge mit der Formel `len+(len>>3)+30` bestimmt. Bei Verlängerungen von jeweils einem Byte wie in dem Beispiel von `readline()` ergibt dies die Folge 30, 64, 103, 147, 196, 251, 313, 383, 462, 550 usw.

Programmtext 1.5 zeigt, wie die **stralloc**-Bibliothek verwendet werden kann, um die Sicherheitslücken des einführenden Beispiels zu entfernen. Zu beachten ist hier, daß nach dem Zusammenfügen mit `stralloc_copys()` und `stralloc_cats()` am Ende noch ein Nullbyte mit spubfile.c

```
	stralloc pathname = {0};
	char buffer[BUFFER_SIZE];
	int fd;
	int count;

	if (**argv == '.' || strchr(*argv, '/')) {
	   fprintf(stderr, "invalid filename: %s\n", *argv);
	   exit(1);
	}

	stralloc_copys(&pathname, pubdir);
	stralloc_cats(&pathname, "/");
	stralloc_cats(&pathname, *argv++);
	stralloc_0(&pathname);
```

Programmtext 1.5: Sicheres Zusammenfügen von Zeichenketten

`stralloc_0` angehängt worden ist. Dies erlaubt es, `pathname.s` als traditionelle Zeichenkette in C zu verwenden — beispielsweise bei der Übergabe an die Funktion `open()`.

Darüber hinaus wird in der korrigierten Version jeder Dateiname dahingehend überprüft, ob er mit einem Punkt beginnt (um sich insbesondere gegen die Verwendung von "." und ".." zu schützen) und ob er einen Schrägstrich enthält, um sich gegen die Angabe relativer Pfadnamen zu schützen.

Hintergrund zur stralloc-Bibliothek

`stralloc sa = {0};`	Initialisierung einer Zeichenkette.
`stralloc_ready(sa, len)`	Bereitstellung von `len` Bytes.
`stralloc_readyplus(sa, len)`	Bereitstellung von `len` weiteren Bytes.
`stralloc_free(sa)`	Freigabe von `sa`.
`sa.s`	Direkter Zugriff auf den Zeiger.
`sa.len`	Länge der Zeichenkette.
`stralloc_copys(sa, s)`	Kopieren von `s` nach `sa`.
`stralloc_copy(sa1, sa2)`	Kopieren von `sa2` nach `sa1`.
`stralloc_cats(sa, s)`	Anhängen von `s` an `sa`.
`stralloc_cat(sa1, sa2)`	Anhängen von `sa2` an `sa1`.
`stralloc_0(sa)`	Anhängen eines Nullbytes an `sa`.
`stralloc_starts(sa, s)`	Findet sich `s` zu Beginn von `sa`?

Tabelle 1.1: Übersicht der **stralloc**-Bibliothek

Die **stralloc**-Bibliothek gehört zu **lowfat**-Bibliothek, die von `http://www.fefe.de/libowfat/` bezogen werden kann. Bei uns ist sie unter `/usr/local/diet` installiert. Die zugehörigen Manualseiten sind abrufbar, wenn der `MANPATH` passend gesetzt wird:

```
theseus$ MANPATH=/usr/local/diet/man man stralloc_ready
```

Eine Kurzübersicht der **stralloc**-Bibliothek findet sich in Tabelle 1.1.

Beim Übersetzen ist darauf zu achten, daß der C-Compiler die zugehörigen Header-Dateien findet und die Bibliothek. Dies geschieht mit den Optionen `-I/usr/local/diet/include` bzw. `-L/usr/local/diet/lib`.

1.4 Unsichere Aufrufe von Kommandos

pubfile2.c

```
/*
 * Display files within my pub directory.
 * Usage: pubfile {file}
 * WARNING: This program has several security flaws.
 * afb 2/2003
 */

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

const int BUFFER_SIZE = 8192;
const char* pubdir = "/home/thales/borchert/pub";

int main(int argc, char** argv) {
   argv++; --argc; /* skip command name */

   while (argc-- > 0) {
      char cmdname[BUFFER_SIZE];

      sprintf(cmdname, "cat %s/%s", pubdir, *argv++);
      system(cmdname);
   }
   return 0;
}
```

Programmtext 1.6: Aufruf des `cat`-Kommandos mit Sicherheitslücken

Zu den Kandidaten der C-Bibliothek, die ihre bequeme Benutzbarkeit mit gefährlichen Nebenwirkungen erkaufen, gehört `system()`. Dies gilt insbesondere dann, wenn die Zeichenkette, die an `system()` übergeben wird, vom Benutzer beeinflußbar ist wie sich am Beispiel des Programmtextes 1.6 zeigen läßt. Hier muß nicht lange darüber nachgedacht werden, wie der Pufferüberlauf bei `sprintf` ausgenutzt werden kann. Die Metazeichen der Shell machen es wirklich einfach:

```
thales$ id
uid=6201(waborche) gid=230(student)
thales$ ls -l /tmp/pubfile2
-rws--x--x   1 borchert sai        6717 Apr 23 15:58 /tmp/pubfile2
thales$ /tmp/pubfile2 'READ_ME; id'
This is the READ_ME file within my pub directory.
uid=6201(waborche) gid=230(student) euid=120(borchert)
thales$
```

Aus dem vorgeblichen Dateinamen "READ_ME; id" wurde hier die Kommandozeile "cat /home/thales/borchert/pub/READ_ME; id". Dank des Semikolons wurden so zwei Kommandos ausgeführt. Dabei ist nicht nur das Semikolon gefährlich: Weitere interessante Metazeichen sind die rückwärts gewandten Anführungszeichen, "&", "|" oder Ein- und Ausgabeumleitungen.

ssystem.h

```
#ifndef SSYSTEM_H
#define SSYSTEM_H
int ssystem(char** argv);
#endif
```

Programmtext 1.7: Sichere Alternative zu `system()`

Wie ist eine Abhilfe möglich? Die Standard-Bibliothek für C liefert leider keine sichere Alternative zu `system()`. Es bleiben natürlich die Systemaufrufe, wovon jedoch leider ein größerer Schwung auf einmal benötigt wird. Mehr dazu im nächsten Kapitel. Angenehm wäre eine Schnittstelle wie bei dem Programmtext 1.7, bei der die Kommandozeilenargumente genauso übernommen werden, wie sie später bei `main()` erwartet werden.

Programmtext 1.8 zeigt wie sich nun `ssystem()` verwenden läßt, um die Involvierung einer Shell und ihrer Metazeichen zu umgehen. Eine weitere Verbesserung besteht hier in der Verwendung von `snprintf()` anstelle von `sprintf()`. `snprintf()` erlaubt die Angabe der Puffergröße und kann somit einen Pufferüberlauf vermeiden.

Leider ist diese Fassung immer noch nicht sicher, da nicht fest angegeben wird, wo das `cat`-Kommando zu finden ist. Wenn kein absoluter Pfadname angegeben ist, läßt sich die Suche manipulieren, indem die Umgebungsvariable `PATH` verändert wird:

```
thales$ id
uid=6201(waborche) gid=230(student)
thales$ (echo '#!/bin/sh'; echo 'exec /bin/sh') >/tmp/cat
thales$ chmod +x /tmp/cat
thales$ PATH=/tmp /tmp/spubfile2 READ_ME
thales$ /bin/id
uid=6201(waborche) gid=230(student) euid=120(borchert)
thales$
```

Wenn `"cat"` durch `"/usr/bin/cat"` in `spubfile2.c` ersetzt wird, ist dieses Problem eliminiert.

Gibt es Umstände, unter denen die Verwendung von `system()` sicher ist? Selbst dann, wenn die Pfadnamen absolut sind und keinerlei Teile vom Benutzer beeinflußbar sind? Programmtext 1.9 zeigt so einen Kandidaten. Ein möglicher Angriffspunkt ist hier die Verwendung der Umgebungsvariablen `IFS`, die der Shell angibt, welche Zeichen als Separatoren zu verwenden spubfile2.c

```
/*
 * Display files within my pub directory.
 * Usage: spubfile2 {file}
 * afb 4/2003
 */

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

const int BUFFER_SIZE = 8192;
const char* pubdir = "/home/thales/borchert/pub";

int main(int argc, char** argv) {
   *argv++; --argc; /* skip command name */

   while (argc-- > 0) {
      char pathname[BUFFER_SIZE];
      char* args[3];

      snprintf(pathname, sizeof pathname, "%s/%s", pubdir, *argv++);
      args[0] = "cat";
      args[1] = pathname;
      args[2] = 0;
      ssystem(args);
   }
   return 0;
}
```

Programmtext 1.8: Etwas sicherer Aufruf des `cat`-Kommandos

pubfile3.c

```
#include <stdlib.h>

int main() {
   system("/usr/bin/cat /home/thales/borchert/pub/READ_ME");
}
```

Programmtext 1.9: Sicherer oder unsicherer Aufruf von `cat`?

sind. Dies sind normalerweise Leerzeichen und Tabs. Wenn es gelänge, den Schrägstrich darin aufzunehmen, würde der absolute Pfadname `/usr/bin/cat` in die drei Wörter `usr`, `bin` und `cat` zerlegt werden und danach eine Suche nach dem Kommando `usr` stattfinden, die wie zuvor mit dem geeigneten Setzen der Umgebungsvariablen `PATH` manipuliert werden kann:

```
thales$ (echo '#!/bin/sh'; echo 'exec /bin/sh') >/tmp/usr
thales$ chmod +x /tmp/usr
thales$ PATH=/tmp IFS=/ /tmp/pubfile3
```

Auf diesem Trick beruhte eine Sicherheitslücke bei dem `preserve`-Kommando, das dazu dient, Sicherheitskopien einer von `vi` editierten Datei in `/var/preserve` abzulegen, falls die `vi`-Sitzung aus irgendeinem Grund unerwartet abgebrochen werden sollte. Um das Verzeichnis `/var/preserve` vor fremden Zugang zu schützen, konnte nur `root` darauf zugreifen und das `preserve`-Kommando erhielt das **setuid**-Bit für `root`, um in dem Verzeichnis arbeiten zu können. `preserve` schickte u.a. auch eine E-Mail mit einem Aufruf von `system()` an den betroffenen Benutzer. Mit der Manipulation der Umgebungsvariablen `PATH` und `IFS` war es dann möglich, ein Skript namens `usr` unterzuschieben, das mit `root`-Privilegien ausgeführt wurde. Die Existenz dieser Sicherheitslücke führte nicht nur dazu, daß eine Lösung gefunden wurde, die ohne ein **setuid**-Bit bei `preserve` auskommt, sondern auch zum vorsichteren Umgang bei der Shell mit der Umgebungsvariablen `IFS`. Einige Versionen ignorieren sie völlig, andere zumindestens innerhalb eines **setuid**-Kontextes. Es ist jedoch nicht empfehlenswert, sich darauf zu verlassen, und daher ist von der Verwendung von `system()` generell abzuraten.

1.5 Richtlinien

Hier sind einige Richtlinien für das Schreiben sicherer systemnaher Software:

- Sicherheit sollte von Anfang an ein Kriterium sein. Es ist meistens ein hoffnungsloses Unterfangen, erst später Sicherheitsüberprüfungen einbauen zu wollen.
- Sicherheit sollte bei jedem Programm relevant sein, da sich sonst die Verwendung in einem sicherheitskritischen Kontext ausschließt. Nur bei temporären Wegwerf-Programmen können Sicherheitsbedenken wegfallen.
- Programme sollten nur ein Minimum an Privilegien erhalten. Häufig ist es ratsam, nicht nur auf `root`-Privilegien zu verzichten, sondern auch noch zusätzliche Restriktionen aufzunehmen wie die Limitierung des Ressourcen-Verbrauches und die Verwendung von `chroot`-Gefängnissen.
- Falls das Arbeiten mit Privilegien unverzichtbar ist, sollte das Aufteilen in mehrere Programme mit unterschiedlichen Privilegien in Betracht gezogen werden.
- Grundsätzlich sollte nichts und niemanden getraut werden, was von außen kommt.
- Bei der Überprüfung von Benutzereingaben sind Positivlisten (was ist erlaubt) besser als Negativlisten (was ist gefährlich).
- Sicherheit beruht auf Verantwortlichkeiten. Damit klar ist, welcher Programmteil für welche Überprüfungen verantwortlich ist, sollten entsprechende Vorgaben und Annahmen klar dokumentiert sein. So sollte beispielsweise innerhalb eines Programmes immer klar hervorgehen, wo mit ungeprüften Eingaben zu rechnen ist.

- Der wohldefinierte Bereich einer Programmiersprache sollte auf keinen Fall verlassen werden, unabhängig davon wie schwierig es sein mag, für Verletzungen passende Einbruchstechniken zu finden.

- Alle angebotenen automatischen Überprüfungen zur Übersetz- und Laufzeit sind zu verwenden.

- Wenn die Programmiersprache oder die Bibliothek nicht genügend automatische Überprüfungen mit sich bringen, ist es ratsam, Bibliotheken zu verwenden, die die Überprüfungen entweder durchführen oder überflüssig machen (Beispiel: **stralloc**-Bibliothek).

- Besser als das stille Abschneiden (Beispiel: `snprintf()`) ist die prinzipielle Unterstützung beliebig langer Eingaben. Der Speicherbedarf wird besser zentral limitiert als bei jeder einzelnen Eingabe.

- Die Grenzen aller Sicherheitsbemühungen sollten nicht vergessen werden. Das sicherste Programm nützt nichts, wenn die Bibliothek, der Compiler, das Betriebssystem oder die Hardware Sicherheitslücken aufweisen, die das Programm betreffen. Ebenso ist der korrekte Umgang mit einer sicherheitskritischen Anwendung relevant. Das schwächste Glied in der Kette ist allzu häufig der Mensch.

Kapitel 2

Prozesse unter UNIX

2.1 Virtueller Adreßraum

Bei einfachen Prozessoren gibt es keinen Unterschied zwischen physischen und virtuellen Speicheradressen und damit auch keine scharfe Trennlinie zwischen Betriebssystem und Anwendung. Dies macht nicht nur die Speicheraufteilung unübersichtlich, es führt auch dazu, daß ein Absturz einer Anwendung, beispielsweise hervorgerufen durch einen ungültigen Zeigerzugriff, das gesamte System beeinträchtigen kann.

Bessere Prozessoren besitzen eine Speicherverwaltung (*memory management unit*, abgekürzt MMU), die die Einrichtung virtueller Speicherumgebungen ermöglicht. Anwendungen verwenden dann virtuelle Speicheradressen, die mittels einer vom Betriebssystem konfigurierten Funktion in physische Speicheradressen abgebildet werden.

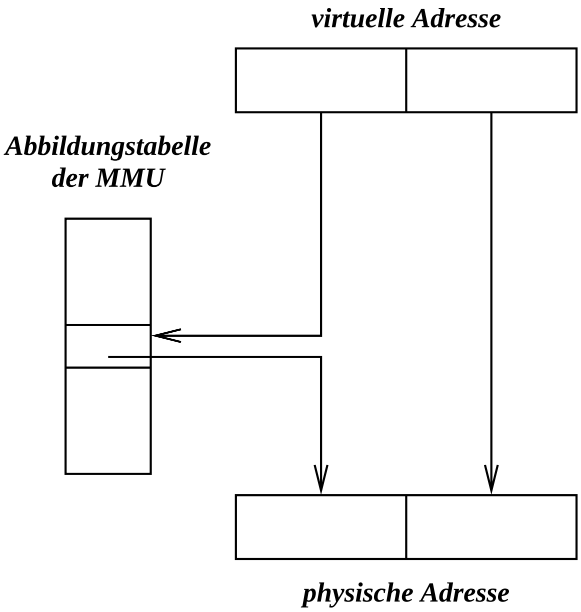

Abbildung 2.1: Virtuelle und physische Adressen

Abbildung 2.1 zeigt schematisch, wie typischerweise virtuelle Adressen in physische Adressen abgebildet werden. Um die Abbildung effizient (in der Hardware) durchführen zu können, wird der gesamte Speicher in Form von Seiten organisiert. Typische Seitengrößen liegen zwischen 4 und 64 Kilobyte. Entsprechend lassen sich virtuelle Adressen in zwei Teile zerlegen: Der niedrigwertige Teil spezifiziert nur die Position innerhalb einer Seite, der höherwertige gibt die Seitennummer an. In einer Tabelle der MMU können nun die Adressen physischer Seiten den Seitennummern virtueller Adressen zugeordnet werden.

Mehrere Abbildungstabellen können parallel nebeneinander existieren: Eine für das Betriebssystem selbst und eine für jede Anwendung. Ein Wechsel läßt sich effizient realisieren, indem ein spezielles Hardware-Register verwendet wird, das auf die derzeitig gültige Abbildungstabelle verweist.

Die Abbildungstabellen einer MMU bieten typischerweise noch Platz für Zugriffsrechte. So kann geregelt werden, welche Seiten aus der Sicht einer gegebenen Abbildungstabelle vorhanden, lesbar, schreibbar und ausführbar sind. Die Gesamtheit aus Abbildungstabelle und Zugriffsrechten wird als **virtueller Adreßraum** bezeichnet. Kommt es bei einem Zugriff zu einem Fehler durch die MMU, kann das Betriebssystem entscheiden, ob es sich dabei um einen Fehler der Anwendung handelt oder ob es den Zugriff nach einigen Manipulationen doch noch ermöglicht.

Viele Tricks sind auf diese Weise möglich:

- Zwei oder mehr Adreßräume können gemeinsam auf die gleiche Bibliothek zugreifen. Hierbei ist es sinnvoll, für den gemeinsamen Bereich nur Lese- und Ausführungsrechte einzuräumen.
- Teile eines belegten Adreßraumes können auf die Platte ausgelagert werden. Bei einem Zugriff kommt es zuerst zu einem Fehler der MMU, der dann vom Betriebssystem abgefangen wird, um den Bereich wieder von Platte einzulesen und zur Verfügung zu stellen.
- Kopien von Speicherbereichen werden verzögert angefertigt, indem der zunächst noch gemeinsam gehaltene Bereich mit einem Schreibschutz versehen wird. Sobald eine Schreiboperation erfolgt, wird die betroffene Seite dupliziert und bei beiden Kopien werden dann die Schreibrechte wieder zurückgegeben.

2.2 Ausführungskontext

Ein Programm wird in einem bestimmten Kontext ausgeführt. Zu diesem Kontext gehören

- der Adreßraum, in dem unter anderem der Programmtext (als Maschinencode) und die Daten untergebracht sind,
- ein Satz Maschinenregister einschließlich der Stackverwaltung (Stack-Zeiger, Frame-Zeiger) und dem PC (*program counter*, verweist auf die nächste auszuführende Instruktion) und
- weitere Statusinformationen, die vom Betriebssystem verwaltet werden wie beispielsweise Informationen über geöffnete Dateien.

Die Einrichtung eines Kontexts und der Wechsel zwischen verschiedenen Kontexten gehört zu den Aufgaben des Betriebssystems. Interessant ist die Frage, ob Ausführungskontexte nur isoliert existieren oder ob es mehrere Kontexte geben kann, die Teile gemeinsam haben.

Die Kombination aus Adreßraum und Statusinformationen des Betriebssystems wird unter UNIX als **Prozeß** bezeichnet. Zu einem Prozeß gehört mindestens ein Satz Maschinenregister einschließlich einem PC. Es können aber auch mehrere sein. In diesem Falle wird von **Threads** gesprochen, d.h. es existieren mehrere Ausführungsfäden, die parallel abgearbeitet werden. Allerdings verwaltet UNIX auch Statusinformationen für einzelne Threads, so daß bei einem Prozeß unter UNIX auch gelegentlich von einer Rechtegemeinschaft gesprochen wird.

Teile der Statusinformationen wie beispielsweise die Dateiverbindungen einschließlich der aktuellen Zugriffsposition können auf dem Wege der Vererbung zwischen UNIX-Prozessen geteilt oder auch explizit übergeben werden.

2.3 Die Prozeß-ID

printpid.c

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

int main() {
   printf("%d\n", (int) getpid());
}
```

Programmtext 2.1: Ausgabe der eigenen Prozeß-ID

Jeder Prozeß hat unter UNIX eine gleichbeibende identifizierende positive ganze Zahl, die mit `getpid()` abgefragt werden kann (siehe Programmtext 2.1). Bei der Mehrheit der UNIX-Systeme liegt die Prozeß-ID im Bereich von 1 bis 32767. Die Eindeutigkeit ist jedoch nur zu Lebzeiten garantiert. Sobald ein Prozeß beendet wird, kann die gleiche Prozeß-ID später einem neuen Prozeß zugeordnet werden. Alle gängigen UNIX-Systeme vergeben Prozeß-IDs reihum, wobei bereits vergebene Prozeß-IDs übersprungen werden.

2.4 Terminierung eines Prozesses

Ein Prozeß kann sich jederzeit mit `exit()` beenden und dabei einen Statuswert im Bereich von 0 bis 255 angeben. Die `exit`-Funktion kann in C-Programmen auch implizit aufgerufen werden: Ein `return` in der `main`-Funktion führt zu einem entsprechenden `exit` und wenn das Ende der `main`-Funktion erreicht wird, entspricht dies einem `exit(0)`. Ein Exit-Wert von 0 deutet dabei eine erfolgreiche Terminierung an, andere Werte, insbesondere `EXIT_FAILURE`, werden als Mißerfolg gewertet. Diese Konventionen orientieren sich zwar an UNIX, sind aber auch Bestandteil des ISO-Standards 9899-1999.

2.5 Das Erzeugen neuer Prozesse

Neue Prozesse können nur in Form eines Klon-Vorganges mit Hilfe des Systemaufrufs `fork()` erzeugt werden:

- Der Adreßraum, die Maschinenregister und fast der gesamte Status des Betriebssystems für den erzeugenden Prozeß werden dupliziert. Das bedeutet, daß beide Prozesse (der `fork()` aufrufende Prozeß und der neu erzeugte Prozeß) einen zu Beginn gleich aussehenden Adreßraum vorfinden. Änderungen werden jedoch nur bei jeweils einem der beiden Prozesse wirksam, so daß hier die Verzögerungstechnik beim Kopieren von Speicherbereichen zum Zuge kommt (*copy on write*).
- Einige Statusinformationen beim Betriebssystem betreffen beide Prozesse. So werden offene Dateiverbindungen vererbt und können gemeinsam genutzt werden. Dies bezieht sich aber nur auf Dateiverbindungen, die zum Zeitpunkt des `fork`-Aufrufs eröffnet waren und nicht auf Dateien, die später von einem der beiden Prozesse neu eröffnet werden.
- Einige Statusinformationen des Betriebssystems werden **nicht** weitergegeben. Dazu gehören beispielsweise Locks und anhängige Signale.

clones.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n");
   fork();
   printf("Hey, I am cloned!\n");
}
```

Programmtext 2.2: Aus einem Prozeß werden zwei

Das überraschende ist hier, daß ein neuer Prozeß nicht irgendwo mit einem neuen Programm bei `main()` beginnt, sondern wir nach dem `fork()` zwei weitgehend übereinstimmende Kopien eines Prozesses vorfinden, die alle das Programm hinter dem Aufruf von `fork()` fortsetzen. Der Programmtext 2.2 demonstriert bei der Ausführung, daß das zweite `printf` doppelt ausgeführt wird:

```
doolin$ clones
I am feeling lonely!
Hey, I am cloned!
Hey, I am cloned!
doolin$
```

Dieses einfache Beispiel ist gut geeignet, eine Falle von `fork()` zu demonstrieren:

```
doolin$ clones | cat
I am feeling lonely!
Hey, I am cloned!
I am feeling lonely!
Hey, I am cloned!
doolin$
```

Warum erhalten wir jetzt die Ausgabe "I am feeling lonely!" nun doppelt? Die Antwort ist in der Pufferung der **stdio**-Bibliothek zu suchen. Erfolgt die Ausgabe direkt auf ein Terminal, wird zeilenweise gepuffert. In diesem Falle erfolgt die Ausgabe des ersten `printf()` noch vor dem Aufruf von `fork()`. Falls jedoch voll gepuffert wird — dies ist bei der Ausgabe in eine Datei oder in eine Pipeline der Fall — dann erfolgt vor dem `fork()` noch keine Ausgabe.

Stattdessen wird der Puffer von `stdout` durch `fork()` dupliziert, womit die doppelte Ausgabe der ersten Zeile provoziert wird. Dieser Effekt läßt sich durch die rechtzeitige Leerung des Puffers mit Hilfe von `fflush()` vermeiden, wie Programmtext 2.3 zeigt.

clones2.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   printf("I am feeling lonely!\n"); fflush(stdout);
   fork();
   printf("Hey, I am cloned!\n");
}
```

Programmtext 2.3: Prozeßduplizierung mit vorheriger Pufferleerung

Es bleibt die Frage, ob die beiden Prozesse in Abhängigkeit davon, ob sie der alte oder der neue Prozeß sind, unterschiedliche Dinge tun können. Programmtext 2.4 zeigt, wie dies mit Hilfe von `getpid()` geschehen kann.

clones3.c

```
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t parent;

   printf("I am feeling lonely!\n"); fflush(stdout);
   parent = getpid();
   fork();
   if (getpid() == parent) {
      printf("I am the parent process!\n");
   } else {
      printf("I am the child process!\n");
   }
}
```

Programmtext 2.4: Unterscheidung von Vorfahre und Nachfahre

Die Unterscheidung kann aber auch mit Hilfe des Rückgabewertes von `fork()` selbst erfolgen. `fork()` liefert -1 im Falle von Fehlern, 0 für den neu erzeugten Prozeß und die Prozeß-ID des neu erzeugten Prozesses beim alten Prozeß. Programmtext 2.5 demonstriert nicht nur die Nutzung des Rückgabewertes, sondern auch die Vermeidung eines gemeinsamen Pfades nach dem `fork()` durch ein explizites `exit()`.

2.6 Synchronisierung bei der Prozeßterminierung

Es mag Fälle geben, bei denen neue Prozesse erzeugt und dann "vergessen" werden. Im Normalfall jedoch stößt das weitere Schicksal des neuen Prozesses auf Interesse und insbesondere ist es nicht unüblich, daß der erzeugende Prozeß auf das Ende der von ihm erzeugten Prozesse warten möchte.

fork.c

```c
#include <stdio.h>
#include <unistd.h>
int main() {
   pid_t pid;

   pid = fork();
   if (pid == -1) {
      perror("unable to fork"); exit(1);
   }
   if (pid == 0) {
      /* child process */
      printf("I am the child process: %d.\n", (int) getpid());
      exit(0);
   }
   /* parent process */
   printf("The pid of my child process is %d.\n", (int) pid);
}
```

Programmtext 2.5: Der Rückgabewert von `fork()`

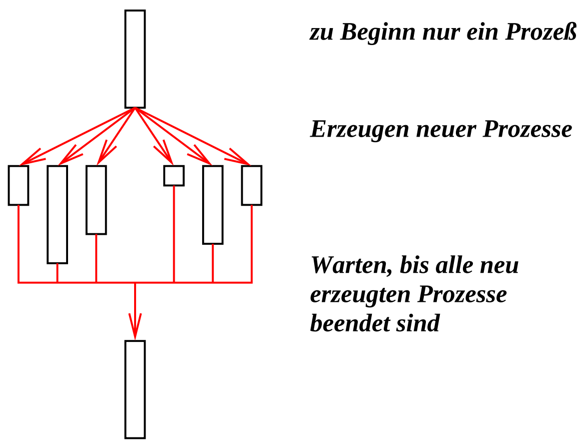

Abbildung 2.2: "Fork and Join"

Dies macht insbesondere dann Sinn, wenn mehrere Prozesse erzeugt werden, die parallel Teilprobleme des Gesamtproblems lösen. Dann wartet der erzeugende Prozeß nach Erzeugung all der Unterprozesse, bis sie alle ihre Teilaufgaben erledigt haben. Dieses Muster wird "Fork and Join" genannt (siehe Abbildung 2.2).

Der Systemaufruf, der es erlaubt, auf das Ende eines Prozesses zu warten, nennt sich `wait()`. Dabei ist zu beachten, daß nur das Warten auf Unterprozesse des aufrufenden Prozesses möglich ist. In der einfachsten Variante wartet `wait()`, bis der nächste Unterprozeß beendet ist und liefert die zugehörige Prozeß-ID zurück. Zusätzlich wird auch noch der bei `exit()` angegebene Wert geliefert.

forkandwait.c

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <unistd.h>
 4  #include <sys/wait.h>
 5
 6  int main() {
 7     pid_t child, pid;
 8     int stat;
 9
10     child = fork();
11     if (child == -1) {
12        perror("unable to fork"); exit(1);
13     }
14     if (child == 0) {
15        /* child process */
16        srand(getpid());
17        char randval = rand();
18        exit(randval);
19     }
20
21     /* parent process */
22     pid = wait(&stat);
23     if (pid == child) {
24        if (WIFEXITED(stat)) {
25           printf("exit code of child = %d\n", WEXITSTATUS(stat));
26        } else {
27           printf("child terminated abnormally\n");
28        }
29     } else {
30        perror("wait");
31     }
32  }
```

Programmtext 2.6: Warten auf die Terminierung

Programmtext 2.6 demonstriert `wait()` an einem einfachen Beispiel. In den Zeilen 16 bis 18 gibt der Sohn einen etwas abwechslungsreichen Rückgabewert bei `exit()` an. In Zeile 22 wartet der erzeugende Prozeß auf die Terminierung des Unterprozesses. `wait()` liefert die

Prozeß-ID des terminierten Prozesses oder -1, falls es keine Unterprozesse mehr gibt, auf die gewartet werden könnte. Der in `stat` abgelegte Status des Unterprozesses besteht aus mehreren Komponenten, die angeben,

- wie ein Prozeß sein Leben beendete[1] (durch `exit()` oder durch ein Signal (bei einem Crash oder Verwendung von `kill()`),
- welcher Wert bei `exit()` angegeben wurde, falls `exit()` benutzt wurde und
- welches Signal das Leben des Prozesses terminierte, falls der Prozeß nicht mit `exit()` endete.

Um das Zerpflücken des Status-Wertes zu vereinfachen, wurden im Rahmen des POSIX-Standards einige Makros definiert, die sich in `<sys/wait.h>` befinden und die Abfragen erleichtern. So ermöglicht `WIFEXITED` die Fallunterscheidung, wie der Prozeß terminierte und `WEXITSTATUS` liefert den Exit-Wert zurück.

Ein etwas trickreicheres Beispiel ist in Programmtext 2.7 zu finden. Beim n-Damen-Problem geht es darum, n Damen auf einem $n \times n$ Schachbrett so unterzubringen, daß sie sich gegenseitig nicht bedrohen. Typischerweise wird das Problem im Backtracking-Verfahren gelöst, bei dem rekursiv alle Varianten durchprobiert werden. Klappt es mit einem Weg nicht, werden die bereits gesetzten Damen wieder sukzessive abgebaut, bis sich andere, bislang noch nicht getestete Varianten eröffnen.

Diese sequentielle Vorgehensweise läßt sich auch parallelisieren. Auf der ersten Reihe auf dem Schachbrett gibt es n mögliche Positionen für die erste Dame. Entsprechend können n Prozesse erzeugt werden, die sich jeweils um einen Teilbaum kümmern. Das Verfahren kann auch weiter mit Parallelisierung fortgesetzt werden. Bei größeren Brettgrößen stößt dies jedoch rasch an die Grenze der Prozeßtabelle, so daß die vorgestellte Lösung nicht wirklich skalierbar ist.

Um das Problem zu vereinfachen, soll das Programm nur die Zahl der gefundenen Lösungen zählen. Eine Ausgabe der gefundenen Lösungen ist nicht trivial, da hier eine Synchronisierung stattfinden müßte. Sonst würden viele Prozesse konkurrierend versuchen, Ausgabe zu produzieren, was zu einer wilden Textmischung führt. Über den Exit-Wert kann für nicht zu große Werte von n bequem die Zahl der gefundenen Lösungen eines Teilbaumes zurückgeliefert werden, so daß der übergeordnete Prozeß die Gelegenheit hat, die Einzelresultate zusammenzuzählen.

Zur Lösung selbst: Der gesamte Stand auf dem Schachbrett wird in einer Reihe lokaler Variablen von `main()` verwaltet. Abgesehen von `col` wird keine dieser Variablen im ersten Prozeß modifiziert. Stattdessen erfolgen sämtliche Änderungen nur bei Unterprozessen, die dank der Magie des `fork()`-Aufrufes auf einer Kopie davon arbeiten.

Die entscheidende Schleife beginnt in Zeile 21. Auf den Reihen 0 bis `nofqueens` -1 sind die Damen bereits gesetzt. Als nächstes ist nun eine Dame auf Reihe `nofqueens` zu plazieren. Dazu geht die Schleife sämtliche Spaltenpositionen durch und überprüft in den Zeilen 22 bis 25, ob die einzelnen Varianten im Konflikt zu den bereits plazierten Damen stehen.

Falls eine weitere Dame gefahrlos gesetzt werden kann, dann wird diese Variante einem Prozeß überlassen, der in Zeile 26 neu erzeugt wird. Dieser setzt die Dame innerhalb der Datenstruktur in den Zeilen 31 bis 35. Wenn damit `size` Damen gesetzt sind, ist eine Lösung gefunden worden

[1] Theoretisch ist es auch möglich, daß der Prozeß nur gestoppt ist. Dies kommt jedoch nur im Kontext von Debuggern vor.

forkingqueens.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#define INSET(member,set) ((1<<(member))&set)
#define INCL(set,member) ((set) |= 1<<(member))
int main() {
   const int size = 6; /* square size of the board */
   struct { int row, col; } pos[size]; /* queen positions */
   /* a queen on (row, col) threatens a row, a column,
      and 2 diagonals;
      rows and columns are characterized by their number (0..n-1),
      the diagonals by row-col+n-1 and row+col,
      (n is a shorthand for the square size of the board)
   */
   int rows = 0, cols = 0; /* bitmaps of [0..n-1] */
   int diags1 = 0; /* bitmap of [0..2*(n-1)] used for row-col+n-1 */
   int diags2 = 0; /* bitmap of [0..2*(n-1)] used for row+col */
   int row = 0; int col;

   setnextqueen: for (col = 0; col < size; ++col) {
      if (INSET(row, rows)) continue;
      if (INSET(col, cols)) continue;
      if (INSET(row - col + size - 1, diags1)) continue;
      if (INSET(row + col, diags2)) continue;
      int child = fork();
      if (child == -1) {
         perror("fork"); exit(255);
      }
      if (child == 0) {
         INCL(rows, row); INCL(cols, col);
         INCL(diags1, row - col + size - 1);
         INCL(diags2, row + col);
         pos[row].row = row; pos[row].col = col;
         ++row; /* set next queen in next row */
         if (row == size) exit(1); /* solution found */
         goto setnextqueen;
      }
   }

   /* count the results of all children */
   int stat; pid_t child; int nofsolutions = 0;
   while ((child = wait(&stat)) > 0) {
      if (!WIFEXITED(stat)) continue;
      int exitval = WEXITSTATUS(stat);
      if (exitval == 255) exit(255);
      nofsolutions += exitval;
   }
   exit(nofsolutions);
}
```

Programmtext 2.7: Rekursion mit Unterprozessen und der Unterprozeß signalisiert dies mit einem `exit(1)` in Zeile 36. Ansonsten gibt es einen Sprung zur Schleife auf Zeile 21, die dann die nächste Zeile zu besetzen versucht.

Die Anweisungen ab Zeile 41 werden von allen Prozessen ausgeführt mit Ausnahme derjenigen, die entweder bei `fork()` scheitern oder höchstselbst eine Lösung gefunden haben. Die Aufgabe besteht hier darin, all die Zahlen der gefundenen Lösungen der einzelnen Unterprozesse zusammenzuzählen. Dies erledigt die `while`-Schleife auf Zeile 43, die `wait()` solange aufruft, bis -1 zurückgeliefert wird, d.h. alle Unterprozesse berücksichtigt worden sind. Die Exit-Werte werden in Zeile 47 einfach aufaddiert. Es findet nur eine zusätzliche Überprüfung statt, ob der Unterprozess normal terminierte und selbst keine Probleme mit der Erzeugung weiterer Unterprozesse hatte. So ein Problem wird mit einem Exit-Wert von 255 in Zeile 28 signalisiert.

Dieses Beispiel ist nicht zur Nachahmung geeignet. Schon ab einer Brettgröße von 7 scheiterte es auf meiner Workstation am Mangel zur Verfügung stehender Einträge in der Prozeßtabelle. Dennoch ist der Ansatz brauchbar, wenn es darum geht, auf einer Mehrprozessor-Maschine die vorhandenen Ressourcen auszunutzen. Allerdings macht es dann nicht Sinn, mehr Prozesse zu erzeugen als tatsächlich Prozessoren vorhanden sind. Ein pragmatischer Ansatz könnte es sein, die erste Stufe der Rekursion (also hier die erste Reihe auf dem Schachbrett) zu parallelisieren und ansonsten sequentiell weiterzuarbeiten. Elegant wird der Programmtext jedoch durch so einen Mischansatz nicht.

Ein weiterer Problempunkt ist die Rückgabe gefundener Lösungen. Dies geht entweder nur unter Verwendung von Dateien (leicht zu programmieren, jedoch nicht sehr effizient) oder der direkten Kommunikation zwischen den Prozessen (effizienter, jedoch leider nicht einfach zu programmieren).

Weitere Abzugspunkte ergeben sich aus der Verwendung einer `goto`-Anweisung und gemeinsamer Programmpfade nach dem `fork()`.

2.7 Sonderfälle: Zombies und der `init`-Prozeß

Was geschieht mit dem Rückgabewert bei `exit()` und dem sonstigen Endstatus eines Prozesses, wenn der übergeordnete Prozeß nicht zeitig `wait()` aufruft? Das UNIX-System läßt solche toten Prozesse noch in seiner Verwaltung weiterleben, so daß der Endstatus noch bewahrt wird, aber die nicht mehr benötigten Ressourcen freigegeben werden. Prozesse, die sich in diesem Stadium befinden, werden als Zombies bezeichnet.

Programmtext 2.8 demonstriert die Erzeugung eines Zombies. Der neu erzeugte Prozeß verabschiedet sich sofort mit `exit()`, während der übergeordnete Prozeß mit Hilfe eines `sleep()`-Aufrufes sich für 60 Sekunden zur Ruhe legt. Während dieser Zeit verbleibt der Unterprozeß im Zombie-Status, wie das `ps`-Kommando belegt:

```
doolin$ genzombie&
[1]     24489
doolin$ 24490

doolin$ ps -ylp 24489,24490
 S   UID   PID  PPID  C PRI NI   RSS     SZ    WCHAN TTY      TIME CMD
 S   120 24489 23591  0  64 28   616    936        ? pts/31   0:00 genzombi
 Z   120 24490 24489  0   0                                   0:00 <defunct>
doolin$
```

genzombie.c

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

int main() {
   pid_t child = fork();
   if (child == -1) {
      perror("fork"); exit(1);
   }
   if (child == 0) exit(0);
   printf("%d\n", child);
   sleep(60);
}
```

Programmtext 2.8: Erzeugung eines Zombies

In der ersten Spalte gibt `ps` bei dieser Aufrufvariante den Status eines Prozesses an. `Z` steht dabei für Zombie, `S` für schlafend. Weitere Varianten sind `O` für gerade arbeitend, `R` für arbeitsbereit und `T` für gestoppt.

Hält der Zombie-Status ewig an? Was geschieht, wenn der übergeordnete Prozeß — wie in diesem Beispiel — sich verabschiedet, ohne jemals mit `wait()` sich den Status abgeholt zu haben? Solange der übergeordnete Prozeß lebt, bleibt der Zombie-Status tatsächlich beliebig lange bestehen. Wenn jedoch ein Unterprozeß verwaist, weil sein übergeordneter Prozeß sich verabschiedet, dann wird ihm der Prozeß mit der Prozeß-ID 1 als neuer übergeordneter Prozeß zugewiesen. Dies geschieht unabhängig davon, ob der untergeordnete Prozeß noch aktiv ist oder bereits ein Zombie geworden ist.

Programmtext 2.9 demonstriert die Erzeugung eines Waisenkindes, indem es `getppid()` vor und nach dem Ende des übergeordneten Prozesses ausgibt. `getppid()` steht dabei für "*get parent process id*". So sieht eine beispielhafte Ausführung aus:

```
doolin$ orphan
Hi, my parent is 24583
doolin$ My parent is now 1

doolin$
```

Der Prozeß mit der Prozeß-ID 1 spielt eine besondere Rolle unter UNIX. Es ist der erste Prozeß, der vom Betriebssystem selbst erzeugt wird und den Programmtext, der unter `/etc/init` zu finden ist, ausführt. Dieser Prozeß startet weitere Prozesse anhand einer Konfigurationsdatei (bei uns unter `/etc/inittab`) und ruft ansonsten `wait()` auf, um den Status der von ihm selbst erzeugten Prozesse oder den von Waisenkindern entgegenzunehmen. Auf diese Weise wird dann auch der Zombie-Status eines Prozesses beendet, wenn es zum Waisenkind wird.

2.8 Der Wechsel zu einem anderen Programm

Mit `fork()` ist es möglich, neue Prozesse zu erzeugen. Allerdings teilten die neuen Prozesse sich das Programm mit ihrem Erzeuger. Wie ist nun der Wechsel zu einem anderen Programm orphan.c

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

int main() {
   pid_t child;
   child = fork();
   if (child == -1) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      printf("Hi, my parent is %d\n", (int) getppid());
      sleep(5);
      printf("My parent is now %d\n", (int) getppid());
   }
   sleep(3);
   exit(0);
}
```

Programmtext 2.9: Ein Prozeß wird zum Waisenkind

möglich? Die Lösung dafür ist der Systemaufruf `exec()`, der

- den gesamten virtuellen Adreßraum des aufrufenden Prozesses auflöst,
- an seiner Stelle einen neuen einrichtet mit einem angegebenen Programmtext,
- sämtliche Maschinenregister für den Prozess neu initialisiert und
- Statusinformationen des Betriebssystems weitgehend unverändert beläßt.

datum.c

```
#include <unistd.h>

int main() {
   execl(
      "/usr/bin/date",  /* path of the program */
      "/usr/bin/date",  /* name of the program, i.e. argv[0] */
      "+%d.%m.%Y",      /* first argument, i.e. argv[1] */
      0                 /* terminate list of arguments */
   );
   /* not reached except if execl failed */
   perror("/usr/bin/date");
}
```

Programmtext 2.10: Wechsel zum Programm `date`

Programmtext 2.10 zeigt ein erstes Beispiel, bei dem der laufende Prozeß seinen Programmtext durch den von `date` austauscht. Dabei ergibt sich die Möglichkeit, Kommandozeilenargumente zu übergeben, die dann vom neuen Programm in `argv[]` vorgefunden werden. Dabei ist zu beachten, daß der Pfadname des auszuführenden Programms (erster Parameter von `execl()` getrennt angegeben wird von dem Parameter, der später unter `argv[0]` zu finden ist. Normalerweise sind beide gleich — der Name kann aber auch vom Pfad beliebig abweichen. Bei `execl()` dürfen beliebig viele Argumente angegeben werden. Die Liste wird mit einem Nullzeiger beendet.

Normalerweise geht es im Programmtext nach einem Aufruf von `execl()` nicht weiter, weil im Erfolgsfalle das Programm ausgetauscht wurde. Nur bei einem Fehler (weil z.B. das `date`-Kommando nicht gefunden wurde) wird das Programm hinter dem Aufruf von `execl()` fortgesetzt.

2.9 Das Zusammenspiel von `fork`, `exec`, `exit` und `wait`

Auf den ersten Blick erscheinen diese vier Systemaufrufe seltsam. Warum ist eine Kombination aus `fork()` und `exec()` notwendig, um einen neuen Prozeß mit einem neuen Programmtext in Gang zu setzen? Wäre es nicht besser und einfacher, nur einen einzigen Systemaufruf dafür zu haben? Die Frage verschärft sich, wenn berücksichtigt wird, daß in der Zeit der frühen UNIX-Implementierungen die Technik des “*copy on write*” noch nicht zur Verfügung stand. Stattdessen war es bei `fork()` notwendig, den gesamten Speicher zu kopieren. Bei BSD wurde deswegen zeitweise `fork1()` eingeführt, das diesen Kopiervorgang unterdrückte, um die typische Kombination von `fork()` und `exec()` nicht zu teuer werden zu lassen.

UNIX ist keinesfalls das erste Betriebssystem, das Prozesse unterstützte. Die älteren Systeme boten in der Tat die Kombination aus `fork()` und `exec()` in einem Systemaufruf an. Überraschenderweise zeigt sich jedoch, daß dies in der Mehrheit der Fälle viel komplizierter ist. Der Haken liegt darin, daß Prozesse häufig eine Umgebung erwarten, die mehr umfaßt als eine Kommandozeile. Wichtiger Bestandteil der Umgebung sind bereits im Vorfeld eingerichtete Ein- und Ausgabeverbindungen und die Zuteilung von Ressourcen.

So sieht die traditionelle Erzeugung eines Prozesses aus:

- Erzeuge einen neuen Prozeß mit einem gegebenen Programmtext. (Die Kombination aus `fork()` und `exec()`.
- Einrichtung der Umgebung für den neuen Prozeß.
- Start des neuen Prozesses.

Entsprechend ist es notwendig, alle wichtigen Systemaufrufe für die Einrichtung einer Umgebung einschließlich dem Öffnen von Ein- und Ausgabeverbindungen in zwei Varianten zu unterstützen: Die eine Variante bezieht sich auf den eigenen Prozeß, die andere für einen untergeordneten Prozeß, der noch nicht gestartet wurde.

Die Trennung in `fork()` und `exec()` eröffnet die Möglichkeit, daß der Programmtext des übergeordneten Prozesses direkt im neu erzeugten Prozess die Umgebung vorbereitet, die dann bei `exec()` von dem gleichen Prozeß mit dem neuen Programmtext vorgefunden wird. Wie Abbildung 2.3 zeigt, wird genau dies von den UNIX-Shells genutzt, um die Umgebung für

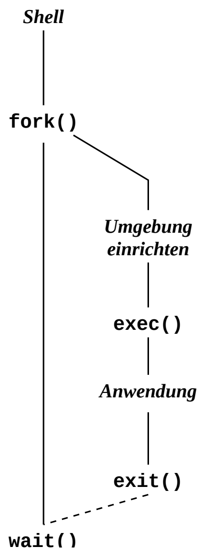

Abbildung 2.3: Start einer Anwendung von der Shell

Anwendungen einzurichten. Der übergeordnete Prozeß der Shell wartet dann normalerweise mit `wait()` auf das Ende des untergeordneten Prozesses.

Programmtext 2.11 zeigt die Grundstruktur einer einfachen Shell. Jede Eingabezeile ist entweder leer oder enthält genau ein Kommando, das innerhalb der `while`-Bedingung mit `readline()` eingelesen wird (siehe Programmtext 1.4). Die eingelesene Zeile wird dann mit der Hilfe des `tokenizer()` in einzelne Wörter zerlegt (siehe Programmtext 2.15), die dann als Kommandoname und als weitere Argumente des Kommandos interpretiert werden.

Wenn die Zahl der Kommandozeilenargumente variabel ist, empfiehlt sich die Verwendung von `execvp()` oder `execv()` anstelle von `execl()`. Sowohl `execvp()` als auch `execv()` akzeptieren eine mit einem 0-Zeiger terminierte Argumentliste in der Form `char**` wie sie auch an `main()` übergeben wird. `execvp()` durchsucht (anders als die einfachere Variante `execv()`) die Umgebungsvariable `PATH`, um den Programmtext zu finden.

Um eine obere Schranke in der Zahl der Kommandozeilenargumente zu vermeiden, ist es angemessen, analog zur **stralloc**-Bibliothek eine ähnliche Bibliothek für Listen von Zeichenketten einzurichten. Die Programmtexte 2.13 und 2.14 zeigen, wie dies geschehen kann.

Der Wortzerleger `tokenizer()` (siehe Programmtext 2.15) arbeitet auf einer als `stralloc` repräsentierten Zeichenkette und generiert eine Liste vom Typ `strlist`, die auf die einzelnen Wörter verweist. Dabei wird ein Umkopieren der Wörter vermieden. Stattdessen verweisen die Zeiger in die originale Zeichenkette und die Leerzeichen werden in Nullbytes verwandelt, um als Begrenzer dienen zu können. Abbildung 2.4 zeigt die resultierende Datenstruktur an einem Beispiel.

Damit die Einrichtung einer Umgebung für den neuen Prozeß emonstriert werden kann, soll `tinysh` einfache Ein- und Ausgabeumleitungen unterstützen. Vorgesehen sind dabei “`<`”, “`>`” und “`>>`”, wobei diese Symbole unmittelbar dem jeweiligen Dateinamen vorangehen müssen, so daß sie mit ihm zusammen als ein Wort betrachtet werden. Der Programmtext 2.12 wird vom neuen Prozeß ausgeführt. Die `for`-Schleife geht dabei durch die Liste aller Wörter aus der Eingabezeile und fügt dabei die normalen Kommandozeilenargumente zur Liste `argv` hinzu, während die Wörter, die Umleitungen spezifizieren, sofort interpretiert werden.

tinysh.c

```
int main() {
   stralloc line = {0};
   while (printf("%% "), readline(stdin, &line)) {
      strlist tokens = {0};
      stralloc_0(&line); /* required by tokenizer() */
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      pid_t child = fork();
      if (child == -1) {
         perror("fork"); continue;
      }
      if (child == 0) {
         strlist argv = {0}; /* list of arguments */
         char* cmdname = 0; /* first argument */
         char* path; /* of output files */
         int oflags;

         /*
            perform I/O redirections and
            collect all other arguments in argv

            ...
         */

         if (cmdname == 0) exit(0);
         strlist_push0(&argv);
         execvp(cmdname, argv.list);
         perror(cmdname);
         exit(255);
      }

      /* wait for termination of child */
      int stat;
      pid_t pid = wait(&stat);
      if (pid == child) {
         if (WIFEXITED(stat)) {
            int code = WEXITSTATUS(stat);
            if (code && code != 255) {
               printf("terminated with exit code %d\n", code);
            }
         } else {
            printf("terminated abnormally\n");
         }
      } else {
         perror("wait");
      }
   }
}
```

Programmtext 2.11: Grundstruktur einer einfachen Shell tinysh.c

```
if (child == 0) {
   strlist argv = {0}; /* list of arguments */
   char* cmdname = 0; /* first argument */
   char* path; /* of output files */
   int oflags;

   for (int i = 0; i < tokens.len; ++i) {
      switch (tokens.list[i][0]) {
         case '<':
            fassign(0, &tokens.list[i][1], O_RDONLY, 0);
            break;
         case '>':
            path = &tokens.list[i][1];
            oflags = O_WRONLY|O_CREAT;
            if (*path == '>') {
               ++path; oflags |= O_APPEND;
            } else {
               oflags |= O_TRUNC;
            }
            fassign(1, path, oflags, 0666);
            break;
         default:
            strlist_push(&argv, tokens.list[i]);
            if (cmdname == 0) cmdname = tokens.list[i];
      }
   }
   if (cmdname == 0) exit(0);
   strlist_push0(&argv);
   execvp(cmdname, argv.list);
   perror(cmdname);
   exit(255);
}
```

Programmtext 2.12: Einrichtung einer Umgebung für einen neuen Prozeß strlist.h

```
/*
 * Data structure for dynamic string lists that works
 * similar to the stralloc library.
 * Return values: 1 if successful, 0 in case of failures.
 * afb 4/2003
 */

#ifndef STRLIST_H
#define STRLIST_H

typedef struct strlist {
   char** list;
   unsigned int len; /* # of strings in list */
   unsigned int allocated; /* allocated length for list */
} strlist;

/* assure that there is at least room for len list entries */
int strlist_ready(strlist* list, unsigned int len);

/* assure that there is room for len additional list entries */
int strlist_readyplus(strlist* list, unsigned int len);

/* truncate the list to zero length */
int strlist_clear(strlist* list);

/* append the string pointer to the list */
int strlist_push(strlist* list, const char* string);
#define strlist_push0(list) strlist_push((list), 0)

/* free the strlist data structure but not the strings */
int strlist_free(strlist* list);

#endif
```

Programmtext 2.13: Datenstruktur und Schnittstelle für Listen von Zeichenketten strlist.c

```
/*
 * Data structure for dynamic string lists that works
 * similar to the stralloc library.
 * Return values: 1 if successful, 0 in case of failures.
 * afb 4/2003
 */
#include <stdlib.h>
#include "strlist.h"

/* assure that there is at least room for len list entries */
int strlist_ready(strlist* list, unsigned int len) {
   if (list->allocated < len) {
      unsigned int wanted = len + (len>>3) + 8;
      char** newlist = (char**) realloc(list->list,
         sizeof(char*) * wanted);
      if (newlist == 0) return 0;
      list->list = newlist;
      list->allocated = wanted;
   }
   return 1;
}

/* assure that there is room for len additional list entries */
int strlist_readyplus(strlist* list, unsigned int len) {
   return strlist_ready(list, list->len + len);
}

/* truncate the list to zero length */
int strlist_clear(strlist* list) {
   list->len = 0;
   return 1;
}

/* append the string pointer to the list */
int strlist_push(strlist* list, const char* string) {
   if (!strlist_ready(list, list->len + 1)) return 0;
   list->list[list->len++] = string;
   return 1;
}

/* free the strlist data structure but not the strings */
int strlist_free(strlist* list) {
   free(list->list); list->list = 0;
   list->allocated = 0;
   list->len = 0;
   return 1;
}
```

Programmtext 2.14: Verwaltung von Listen für Zeichenketten tokenizer.c

```c
/*
 * Simple tokenizer: Take a 0-terminated stralloc object and return a
 * list of pointers in tokens that point to the individual tokens.
 * Whitespace is taken as token-separator and all whitespaces within
 * the input are replaced by null bytes.
 * afb 4/2003
 */

#include <ctype.h>
#include <stdlib.h>
#include <stralloc.h>
#include "strlist.h"
#include "tokenizer.h"

int tokenizer(stralloc* input, strlist* tokens) {
   char* cp;
   int white = 1;

   strlist_clear(tokens);
   for (cp = input->s; *cp && cp < input->s + input->len; ++cp) {
      if (isspace(*cp)) {
         *cp = '\0'; white = 1; continue;
      }
      if (!white) continue;
      white = 0;
      if (!strlist_push(tokens, cp)) return 0;
   }
   return 1;
}
```

Programmtext 2.15: Zerlegung einer Zeile in Wörter

Die Feinarbeit übernimmt hier die Funktion `fassign` (siehe Programmtext 2.16), die die angegebene Datei mit den gewünschten Modi eröffnet und dafür den als ersten vorgegebenen Dateideskriptor verwendet. Hierbei kommt die Systemfunktion `dup2()` zum Einsatz, die `fd` zum Alias von `newfd` macht. Danach kann `newfd` geschlossen werden. Falls `fd` zuvor bereits auf eine geöffnete Datei verwies, wurde die Verbindung zuerst gekappt.

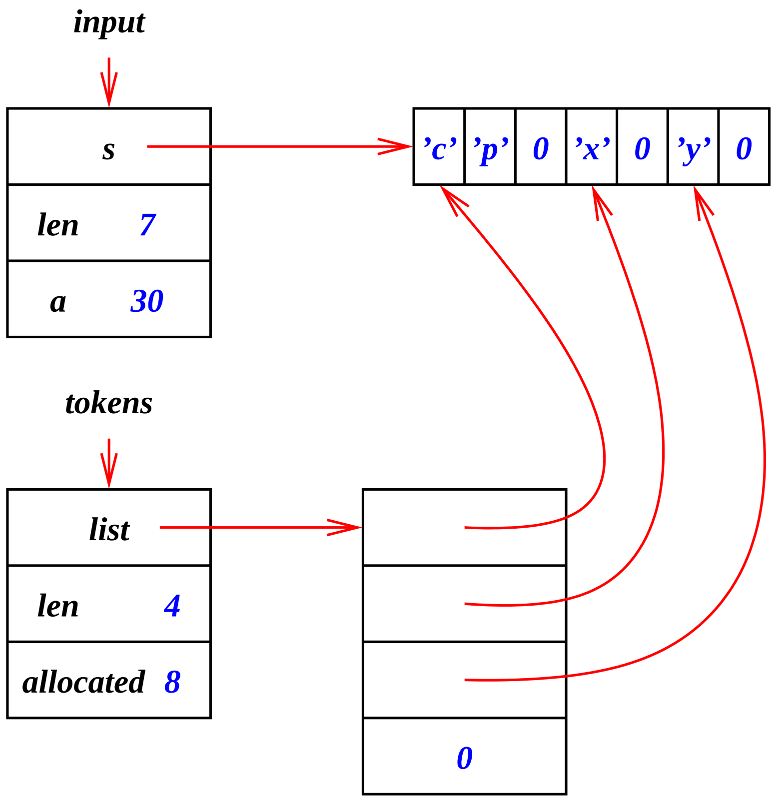

Abbildung 2.4: Resultierende Datenstruktur des Wortzerlegers am Beispiel “cp x y”

tinysh.c

```
/*
 * assign an opened file with the given flags and mode to fd
 */
void fassign(int fd, char* path, int oflags, mode_t mode) {
   int newfd = open(path, oflags, mode);
   if (newfd < 0) {
      perror(path); exit(255);
   }
   if (dup2(newfd, fd) < 0) {
      perror("dup2"); exit(255);
   }
   close(newfd);
}
```

Programmtext 2.16: Verknüpfung vorgegebener Dateideskriptoren mit neu eröffneten Dateien

Kapitel 3

Signale

3.1 Einführung

Signale werden für vielfältige Zwecke eingesetzt. Sie können verwendet werden,

- um den normalen Ablauf eines Prozesses für einen wichtigen Hinweis zu unterbrechen,
- um die Terminierung eines Prozesses zu erbitten oder zu erzwingen und
- um schwerwiegende Fehler bei der Ausführung zu behandeln wie z.B. den Verweis durch einen invaliden Zeiger.

Signale ersetzen keine Interprozeßkommunikation, da sie fast keine Informationen mit sich führen. In Abhängigkeit von der jeweiligen Systemumgebung gibt es mehr oder weniger fest definierte Signale, die über natürliche Zahlen identifiziert werden.

Der ISO-Standard 9899-1999 für die Programmiersprache C definiert eine einfache und damit recht portable Schnittstelle für die Behandlung von Signalen. Hier gibt es neben der Signalnummer selbst keine weiteren Informationen. Der IEEE Standard 1003.1 (POSIX) bietet eine Obermenge der Schnittstelle des ISO-Standards an, bei der wenige zusätzliche Informationen (wie z.B. die Angabe des invaliden Zeigers) dabei sein können und der insbesondere eine sehr viel feinere Kontrolle der Signalbehandlung erlaubt.

Signale können von verschiedenen Parteien ausgelöst werden: Von anderen Prozessen, die die dafür notwendige Berechtigung haben (entweder der gleiche Benutzer oder der Super-User), durch den Prozeß selbst entweder indirekt (durch einen schwerwiegenden Fehler) oder explizit oder auch durch das Betriebssystem.

Typisch für letzteres ist die Terminalschnittstelle unter UNIX. Diese wurde ursprünglich für ASCII-Terminals mit serieller Schnittstelle entwickelt, die nur folgende Eingabemöglichkeiten anboten:

- Einzelne ASCII-Zeichen, jeweils ein Byte (zusammen mit etwas Extra-Kodierung wie Prüf- und Stop-Bits).
- Ein BREAK, das als spezielles Signal repräsentiert wird, das länger als die Kodierung für ein ASCII-Zeichen währt.

- Ein HANGUP, bei dem ein Signal wegfällt, das zuvor die Existenz der Leitung bestätigt hat. Dies benötigt einen weiteren Draht in der seriellen Leitung.

Diese Eingaben werden auf der Seite des Betriebssystems vom Terminal-Treiber bearbeitet, der in Abhängigkeit von den getroffenen Einstellungen

- die eingegebenen Zeichen puffert und das Editieren der Eingabe ermöglicht (beispielsweise mittels BACKSPACE, CTRL-u und CTRL-w) und
- bei besonderen Eingaben Signale an alle Prozesse schickt, die mit diesem Terminal verbunden sind.

Ziel war es, daß im Normalfall ein BREAK zu dem Abbruch oder zumindest der Unterbrechung der gerade laufenden Anwendung führt. Und ein HANGUP sollte zu dem Abbruch der gesamten Sitzung führen, da bei einem Wegfall der Leitung keine Möglichkeit eines regulären Abmeldens besteht.

Heute sind serielle Terminals rar geworden, aber das Konzept wurde dennoch beibehalten. Zwischen einem virtuellen Terminal (beispielsweise einem xterm) und den Prozessen, die zur zugehörigen Sitzung gehören, ist ein sogenanntes Pseudo-Terminal im Betriebssystem geschaltet, das der Sitzung die Verwendung eines klassischen Terminals vorspielt. Da es BREAK in diesem Umfeld nicht mehr gibt, wird es durch ein beliebiges Zeichen ersetzt wie beispielsweise CTRL-c. Und wenn das virtuelle Terminal wegfällt (z.B. durch eine gewaltsame Beendigung der xterm-Anwendung), dann gibt es weiterhin ein HANGUP für die Sitzung.

Auf fast alle Signale können Prozesse, die sie erhalten, auf dreierlei Weise reagieren:

- Voreinstellung: Terminierung des Prozesses.
- Ignorieren.
- Bearbeitung durch einen Signalbehandler.

Es mag harsch erscheinen, daß die Voreinstellung zur Terminierung eines Prozesses führt. Aber genau dies führt bei normalen Anwendungen genau zu den gewünschten Effekten wie Abbruch des laufenden Programms bei BREAK (die Shell ignoriert das Signal) und Abbau der Sitzung bei HANGUP. Wenn ein Prozeß diese Signale ignoriert, sollte es genau wissen, was es tut, da der Nutzer auf diese Weise eine wichtige Kontrollmöglichkeit seiner Sitzung verliert. Sinnvoll ist es natürlich, eine Anwendung mit einem Signalbehandler zu ergänzen, wenn dadurch Datenverluste bei einer Terminierung vermieden werden können.

3.2 Signalbehandler

Programmtext 3.1 demonstriert die Behandlung des Signals `SIGINT`. Als Signalbehandler operiert die Funktion `signal_handler()`. Signalbehandler erhalten als Argument eine ganze Zahl, worüber sie die Nummer des Signals erhalten, das sie gerade bearbeiten. Einen Rückgabewert gibt es nicht.

Der Signalbehandler in diesem Beispiel setzt nur eine globale Variable auf den Wert des erhaltenen Signals (Zeile 8). Die Verwendung des Typs `volatile sig_atomic_t` wird später sigint.c

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  volatile sig_atomic_t signal_caught = 0;
6
7  void signal_handler(int signal) {
8     signal_caught = signal;
9  }
10
11 int main() {
12    if (signal(SIGINT, signal_handler) == SIG_ERR) {
13       perror("unable to setup signal handler for SIGINT");
14       exit(1);
15    }
16    printf("Try to send a SIGINT signal!\n");
17    int counter = 0;
18    while (!signal_caught) {
19       for (int i = 0; i < counter; ++i);
20       ++counter;
21    }
22    printf("Got signal %d after %d steps!\n", signal_caught, counter);
23 }
```

Programmtext 3.1: Behandlung eines `SIGINT`-Signals

diskutiert. `main()` richtet auf Zeile 12 die Funktion `signal_handler()` als Signalbehandler ein. Dies geschieht mit der Funktion `signal()` die eine Signalnummer (hier: `SIGINT`) und eine Reaktion (entweder `SIG_DFL` für die Prozeßterminierung, `SIG_IGN` für das Ignorieren oder einen Funktionszeiger) als Parameter erwartet. Im Erfolgsfalle liefert `signal()` die frühere Einstellung zurück, ansonsten `SIG_ERR`.

Weiter geht es auf Zeile 18 mit einer Schleife, die erst dann abbricht, wenn sich der Wert von `signal_caught` ändert. Dies kann in diesem Beispiel nur durch den Signalbehandler passieren. Wenn dies geschieht, wird auf Zeile 22 die Nummer des eingetroffenen Signals ausgegeben und das Programm beendet. So könnte ein Aufruf dieses Programms aussehen, wenn das Versenden des Signals `SIGINT` durch den Terminaltreiber durch die Eingabe von CTRL-c recht schnell geschieht:

```
doolin$ sigint
Try to send a SIGINT signal!
^CGot signal 2 after 2074 steps!
doolin$
```

Die 2 ist dabei die Nummer des Signals `SIGINT`:

```
doolin$ grep SIGINT /usr/include/sys/iso/signal_iso.h
#define SIGINT  2       /* interrupt (rubout) */
doolin$
```

Leider sind mit der Bearbeitung von Signalen große Probleme verbunden. Wenn ein optimierender Übersetzer den Programmtext 3.1 analysiert, könnten folgende Punkte auffallen:

- Die Schleife in den Zeilen 18 bis 21 ruft keine externen Funktionen auf.
- Innerhalb der Schleife wird `signal_caught` nirgends verändert.

Daraus könnte vom Übersetzer der Schluß gezogen werden, daß die Schleifenbedingung nur zu Beginn einmal überprüft werden muß. Findet der Eintritt in die Schleife statt, könnte der weitere Test der Bedingung ersatzlos wegfallen. Analysen wie diese sind für heutige optimierende Übersetzer Pflicht, um guten Maschinen-Code erzeugen zu können. Es wäre also fatal, wenn darauf nur wegen der Existenz von asynchron aufgerufenen Signalbehandlern verzichtet werden würde.

Um beides zu haben, die fortgeschrittenen Optimierungstechniken und die Möglichkeit, Variablen innerhalb von Signalbehandlern setzen zu können, wurde in C die Speicherklasse **volatile** eingeführt. Damit lassen sich Variablen kennzeichnen, deren Wert sich jederzeit ändern kann — selbst dann, wenn dies aus dem vorliegenden Programmtext nicht ersichtlich ist. Entsprechend gilt dann auch in C, daß alle anderen Variablen, die nicht als **volatile** klassifiziert sind, sich nicht durch "magische" Effekte verändern dürfen.

Daraus folgt, daß korrekte Signalbehandler in ihren Möglichkeiten stark eingeschränkt sind. So ist es nur zulässig,

- lokale Variablen zu verwenden,
- mit `volatile` deklarierte Variablen zu benutzen und
- Funktionen aufzurufen, die sich an die gleichen Spielregeln halten.

Letzteres schließt insbesondere die Verwendung von Ein- und Ausgabe innerhalb eines Signalbehandlers aus. Der ISO-Standard 9899-1999 nennt nur `abort()`, `_Exit()`[1] und `signal()` als zulässige Bibliotheksfunktionen. Beim POSIX-Standard werden noch zahlreiche weitere Systemaufrufe genannt. Auf den Manualseiten von Solaris wird dies dokumentiert durch die Angabe "Async-Signal-Safe" bei "MT-Level"[2]

Wozu der Datentyp `sig_atomic_t`? Dieser Datentyp ist ein ganzzahliger Typ, bei dem Lese- und Schreiboperationen garantiert atomar ablaufen. Es handelt sich also um einen Datentyp, den der verwendete Prozessor mit einer einzigen ununterbrechbaren Instruktion laden oder speichern kann. Bei allen anderen Datentypen ist es nicht ausgeschlossen, daß mitten in einer Zuweisung ein asynchrones Signal eintreffen kann und der zugehörige Signalbehandler dann eine partiell modifizierte Variable vorfindet.

3.3 Wecksignale mit `alarm`

Zu den Signalen, den der POSIX-Standard definiert, gehört auch `SIGALRM`, das sich als Wecksignal verwenden läßt. Der Wecker wird mit `alarm()` unter Angabe einer relativen Weckzeit in Sekunden gestellt. Am Ende dieser Frist kommt es zum Eintreffen des Signals `SIGALRM`.

[1] `_Exit()` unterläßt im Vergleich zu `exit()` sämtliche Aufräumarbeiten.

[2] "MT" steht hier für Multi-Threading, da dabei ähnliche Probleme auftreten, wenngleich in einem noch größeren Umfang.

tread.c

```
/*
 * Timed read operation. timed_read() works like read() but
 * returns 0 if no input was received within the given number
 * of seconds.
 */

#include <signal.h>
#include <unistd.h>
#include "tread.h"

static volatile sig_atomic_t time_exceeded = 0;

static void alarm_handler(int signal) {
   time_exceeded = 1;
}

int timed_read(int fd, void* buf, size_t nbytes, unsigned seconds) {
   if (seconds == 0) return 0;
   /*
    * setup signal handler and alarm clock but
    * remember the previous settings
    */
   void (*previous_handler)(int) = signal(SIGALRM, alarm_handler);
   if (previous_handler == SIG_ERR) return -1;
   time_exceeded = 0;
   int remaining_seconds = alarm(seconds);
   if (remaining_seconds > 0) {
      if (remaining_seconds <= seconds) {
         remaining_seconds = 1;
      } else {
         remaining_seconds -= seconds;
      }
   }

   int bytes_read = read(fd, buf, nbytes);

   /* restore previous settings */
   if (!time_exceeded) alarm(0);
   signal(SIGALRM, previous_handler);
   if (remaining_seconds) alarm(remaining_seconds);

   if (time_exceeded) return 0;
   return bytes_read;
}
```

Programmtext 3.2: `read`-Operation mit Zeitlimit

Programmtext 3.2 zeigt, wie `alarm()` verwendet werden kann, um eine Operation zeitlich zu befristen. Die Funktion `timed_read()` arbeitet genauso wie `read()`, wobei jedoch die Wartezeit auf die gegebene Zahl von Sekunden begrenzt wird. Wenn die Zeit verstreicht, ohne daß eine Eingabe erfolgte, wird 0 zurückgeliefert.

Was geschieht, wenn mehrere Wecksignale nebeneinander eingerichtet werden? Da `alarm()` nur die Verwaltung einer einzigen Weckzeit unterstützt, ist eine kooperative Vorgehensweise notwendig. Dies wird dadurch erleichtert, indem `alarm()` bei der Einrichtung einer neuen Weckzeit entweder 0 zurückgibt (vorher war keine Weckzeit aktiv) oder eine positive Zahl von Sekunden als Restzeit zum vorher eingestellten Wecksignal. Entsprechend wird auf Zeile 26 in Programmtext 3.2 in der Variablen `remaining_seconds` die alternative Weckzeit notiert und später auf Zeile 40 findet eine Wiedereinsetzung statt, nachdem zuvor auf Zeile 31 die verbleibende Restzeit neu berechnet worden ist. Analog wird in Zeile 23 der frühere Signalbehandler für `SIGALRM` in der Variablen `previous_handler` gesichert, so daß er in Zeile 39 wieder restauriert werden kann.

Bei der Restaurierung ist es wichtig, daß kein Fenster offenbleibt, das zum Verlust einer Signalbehandlung führt oder den alten Signalbehandler auf das noch nicht eingetretene Signal für das von `tread()` eingerichtete Zeitlimit reagieren läßt. Wenn der alte Signalbehandler `SIG_DFL` war, also die Voreinstellung, dann würde das sogar unerwartet zur Terminierung unseres Prozesses führen. Deswegen wird in Zeile 38 der Wecker zuerst deaktiviert, wenn er bislang sein Signal noch nicht von sich gegeben hat. Danach kann in Zeile 39 der alte Signalbehandler eingesetzt werden, ohne daß wir Gefahr laufen, daß er sofort verwendet wird. In Zeile 40 wird dann der Wecker neu aufgesetzt, falls er vor dem Aufruf von `tread()` eingeschaltet war.

Es bleibt hier noch anzumerken, daß es noch bessere Wege gibt, Leseoperationen mit Zeitbeschränkungen zu versehen. Es empfiehlt sich hier entweder die Verwendung der Systemaufrufe `poll()` oder `select()` oder der Einsatz asynchroner Lesetechniken. Dennoch ist die Verwendung von `alarm()` sinnvoll, da es genügend Operationen gibt, bei denen es keine Variante mit Zeitbeschränkung gibt.

3.4 Das Versenden von Signalen

Der ISO-Standard für C sieht nur eine Funktion `raise()` vor, die es erlaubt, ein Signal an den eigenen Prozeß zu versenden. Im POSIX-Standard kommt die Funktion `kill()` hinzu, die es erlaubt, ein Signal an einen anderen Prozeß zu verschicken, sofern die dafür notwendigen Privilegien vorliegen. Programmtext 3.3 zeigt ein Beispiel, bei dem ein neu erzeugter Prozeß ein Signal an den Erzeuger sendet.

Trotz ihres geringen Informationsgehalts dienen Signale gelegentlich zur Kommunikation. So gibt es eine weitverbreitete Konvention, daß bei langfristig laufenden Diensten `SIGHUP` das erneute Einlesen der Konfiguration veranlaßt und `SIGTERM` eine geordnete Terminierung einleiten soll. Gelegentlich sind für Dienste auch Reaktionen für `SIGUSR1` und `SIGUSR2` definiert. So wird der Apache-Webserver beispielsweise bei `SIGUSR1` veranlaßt, bei nächster Gelegenheit auf sanfte Weise neu zu starten. Anders als bei `SIGHUP` kommt es dann nicht zur Unterbrechung aktueller Netzwerkverbindungen.

Der Systemaufruf `kill()` erfüllt aber auch noch einen weiteren Zweck. Bei einer Signalnummer von 0 wird nur die Zulässigkeit des Signalversendens überprüft. Programmtext 3.4 demonstriert, wie dies dazu verwendet werden kann, um die Existenz eines Prozesses zu überprüfen.

killparent.c

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

void sigterm_handler(int signo) {
   const char msg[] = "Goodbye, cruel world!\n";
   write(1, msg, sizeof msg - 1);
   _Exit(1);
}

int main() {
   if (signal(SIGTERM, sigterm_handler) == SIG_ERR) {
      perror("signal"); exit(1);
   }

   pid_t child = fork();
   if (child == 0) {
      kill(getppid(), SIGTERM);
      exit(0);
   }
   int wstat;
   wait(&wstat);
   exit(0);
}
```

Programmtext 3.3: Versenden eines Signals an den übergeordneten Prozeß

Gelegentlich kommt es vor, daß Prozesse nur auf das Eintreffen eines Signals warten möchten und sonst nichts zu tun haben. Theoretisch könnte ein Prozess dann in eine Dauerschleife mit leerem Inhalt treten (auch *busy loop* bezeichnet), aber dies wäre nicht sehr fair auf einem System mit mehreren Prozessen, da dadurch Rechenzeit vergeudet würde. Abhilfe schafft hier der Systemaufruf `pause()`, der einen Prozeß schlafen legt, bis ein Signal eintrifft.

Programmtext 3.5 demonstriert das Warten auf ein Signal anhand eines virtuellen Ballspiels zweier Prozesse. Das Programm beginnt in Zeile 37 mit der Erzeugung eines weiteren Prozesses, um einen Spielpartner zu haben. Beide Prozesse rufen anschließend `playwith()` auf, um das Spiel durchzuführen. Der neu erzeugte Prozeß hat zuerst den virtuellen Ball und darf mit dem Spiel beginnen.

Der Besitzer des virtuellen Balles sendet in Zeile 25 den Ball mit Hilfe des Signals `SIGUSR1` an den Partner und legt sich anschließend in Zeile 22 mittels `pause()` schlafen, um auf das Wiedereintreffen des Balls zu warten, das wiederum durch `SIGUSR1` signalisiert wird.

Die Variable `sigcount` in Zeile 6 repräsentiert die Zahl der Bälle, die sich im Augenblick im Besitz des Prozesses sind. Diese Zahl wird von dem Signalbehandler `sighandler()` in Zeile 9 hochgezählt, wenn ein `SIGUSR1`-Signal eintrifft und in Zeile 30 wieder heruntergezählt, nachdem das `SIGUSR1`-Signal an den Partner verschickt wurde.

waitfor.c

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 1) {
      fprintf(stderr, "Usage: %s pid\n", cmdname);
      exit(1);
   }

   /* convert first argument to pid */
   char* endptr = argv[0];
   pid_t pid = strtol(argv[0], &endptr, 10);
   if (endptr == argv[0]) {
      fprintf(stderr, "%s: integer expected as argument\n",
         cmdname);
      exit(1);
   }

   while (kill(pid, 0) == 0) sleep(1);

   if (errno == ESRCH) exit(0);
   perror(cmdname); exit(1);
}
```

Programmtext 3.4: Verwendung von `kill()` zur Überprüfung der Existenz eines Prozesses

Zu Beginn setzen beide Spieler in Zeile 14 `sighandler()` als Signalbehandler ein. Derjenige, der mit dem Spiel beginnt, wartet dann in Zeile 18 noch eine Sekunde, um zu vermeiden, daß ein Signal geschickt wird, bevor der Spielpartner die Gelegenheit hatte, seinen Signalbehandler aufzusetzen.

Zu beachten ist dabei, daß der Signalbehandler in Zeile 10 sich selbst wieder einsetzt, da der POSIX-Standard nicht festlegt, ob diese Einstellung nach Eintreffen eines Signals erhalten bleibt. `signal()` gehört mit zu den vom POSIX-Standard genannten Funktionen, die auch innerhalb eines Signalbehandlers aufgerufen werden dürfen.

3.5 Die Zustellung von Signalen

Die vorangegangenen Beispiele werfen die Frage auf, wie UNIX bei der Zustellung von Signalen vorgeht, wenn

- der Prozeß zur Zeit nicht aktiv ist,

pingpong.c

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

static volatile sig_atomic_t sigcount = 0;

void sighandler(int sig) {
   ++sigcount;
   if (signal(sig, sighandler) == SIG_ERR) _Exit(1);
}

static void playwith(pid_t partner, int start) {
   if (signal(SIGUSR1, sighandler) == SIG_ERR) {
      perror("signal SIGUSR1"); exit(1);
   }
   /* give our partner some time for preparation */
   if (start) sleep(1);
   /* start the ping pong game */
   if (start) sigcount = 1;
   for(int i = 0; i < 10; ++i) {
      if (!sigcount) pause();
      printf("[%d] send signal to %d\n",
         (int) getpid(), (int) partner);
      if (kill(partner, SIGUSR1) < 0) {
         printf("[%d] %d is no longer alive\n",
            (int) getpid(), (int) partner);
         return;
      }
      --sigcount;
   }
   printf("[%d] finishes playing\n", (int) getpid());
}

int main() {
   pid_t parent = getpid();
   pid_t child = fork();

   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      playwith(parent, 1);
   } else {
      playwith(child, 0);
   }
}
```

Programmtext 3.5: Virtuelles Ballspiel zweier Prozesse

- gerade ein Systemaufruf für den Prozeß abgearbeitet wird oder
- gerade ein Signalbehandler bereits aktiv ist.

Vom ISO-Standard 9899-1999 für C wird in dieser Beziehung nichts festgelegt. Der POSIX-Standard geht jedoch genauer darauf ein:[3]

- Wenn ein Prozeß ein Signal erhält, wird dieses Signal zunächst in den zugehörigen Verwaltungsstrukturen des Betriebssystems vermerkt. Signale, die für einen Prozeß vermerkt sind, jedoch noch nicht zugestellt worden sind, werden als **anhängige** Signale bezeichnet.
- Wenn mehrere Signale mit der gleichen Nummer anhängig sind, ist nicht festgelegt, ob eine Mehrfachzustellung erfolgt. Es können also Signale wegfallen.
- Nur aktiv laufende Prozesse können Signale empfangen. Prozesse werden normalerweise durch die Existenz eines anhängigen Signals aktiv — aber dieses kann auch längere Zeit in Anspruch nehmen, wenn dem zwischenzeitlich mangelnde Ressourcen entgegenstehen.
- Für jeden Prozess gibt es eine Menge blockierter Signale, die im Augenblick nicht zugestellt werden sollen. Dies hat nichts mit dem Ignorieren von Signalen zu tun, da blockierte Signale anhängig bleiben, bis die Blockierung aufgehoben wird.
- Der POSIX-Standard legt nicht fest, was mit der Signalbehandlung geschieht, wenn ein Signalbehandler aufgerufen wird. Möglich ist das Zurückfallen auf `SIG_DFL` (Voreinstellung mit Prozeßterminierung) oder die temporäre automatische Blockierung des Signals bis zur Beendigung des Signalbehandlers. Alle modernen UNIX-Systeme wählen die zweite Variante. Dies läßt sich aber gemäß dem POSIX-Standard auch erzwingen, indem die umfangreichere Schnittstelle `sigaction()` anstelle von `signal()` verwendet wird. Allerdings ist `sigaction()` nicht mehr Bestandteil des ISO-Standards für C.
- UNIX unterscheidet zwischen unterbrechbaren und unterbrechungsfreien Systemaufrufen. Zur ersteren Kategorie gehören weitgehend alle Systemaufrufe, die zu einer längeren Blockierung eines Prozesses führen können. Ist ein nicht blockiertes Signal anhängig, kann ein unterbrechbarer Systemaufruf aufgrund des Signals mit einer Fehlerindikation beendet werden. `errno` wird dann auf `EINTR` gesetzt. Dabei ist zu beachten, daß der unterbrochene Systemaufruf nach Beendigung der Signalbehandlung **nicht** fortgesetzt wird, sondern manuell erneut gestartet werden muß. Dies kann leider zu unerwarteten Überraschungseffekten führen, weil insbesondere auch die `stdio`-Bibliothek keinerlei Vorkehrungen trifft, Systemaufrufe automatisch erneut aufzusetzen, falls es zu einer Unterbrechung kam. Dies ist eine wesentliche Schwäche sowohl des POSIX-Standards als auch der `stdio`-Bibliothek und ein Grund mehr dafür, auf die Verwendung der `stdio` in kritischen Anwendungen völlig zu verzichten.

Für die genauere Regulierung der Signalbehandlung bietet POSIX (jedoch nicht ISO-C) den Systemaufruf `sigaction()` an. Während bei `signal()` zur Spezifikation der Signalbehandlung nur ein Funktionszeiger genügte, kommen bei der `struct sigaction`, die `sigaction()` verwendet, die in Tabelle 3.1 genannten Felder zum Einsatz.

[3] Siehe "Signal Concepts", im Web unter
`http://www.opengroup.org/onlinepubs/007904975/functions/xsh_chap02_04.html`

Datentyp	Feldname	Beschreibung
`void(*) (int)`	`sa_handler`	Funktionszeiger (wie bisher)
`void(*) (int, siginfo_t*, void*)`	`sa_sigaction`	alternativer Zeiger auf einen Signalbehandler, der mehr Informationen zum Signal erhält
`sigset_t`	`sa_mask`	Menge von Signalen, die während der Signalbehandlung dieses Signals zu blockieren sind
`int`	`sa_flags`	Menge von Boolean-wertigen Optionen

Tabelle 3.1: Felder der `struct sigaction`

Ein wesentlicher Unterschied zwischen `sigaction()` und `signal()` besteht bereits darin, daß per Voreinstellung das Signal, das eine Signalbehandlung auslöst, während der Bearbeitung automatisch blockiert wird. Ferner findet (solange nichts gegenteiliges in den Optionen angegeben wurde) keine implizite Veränderung des Signalbehandlers auf `SIG_DFL` nach der Signalbehandlung statt. Bei `signal()` ist dies ebenfalls möglich, jedoch nicht garantiert.

Programmtext 3.6 demonstriert den möglichen Verlust von Signalen trotz umfangreicher Vorsichtsmaßnahmen. Das Experiment besteht hier im Versenden von 1000 `SIGUSR1`-Signalen, die beim Empfänger nachgezählt werden. In den Zeilen 26 bis 30 setzt der neu erzeugte Prozeß die Funktion `count_signals()` als Signalbehandler für `SIGUSR1` auf. Dabei wird hier `sigaction()` anstelle von `signal()` verwendet. Dies garantiert uns, daß während der Bearbeitung des Signals `SIGUSR1` weitere eintreffende `SIGUSR1`-Signale aufgeschoben und nicht sofort in verschachtelter Form bearbeitet werden.

Auf diese Weise ist die Atomizität des Hochzählens der Variable `received_signals` garantiert. Zu bedenken ist dabei, daß der Datentyp `sig_atomic_t` selbst nur die Atomizität einer einzelnen Lese- oder Schreiboperation gewährleistet. Bei einem Inkrement liegt jedoch eine Lese- **und** eine Schreib-Operation vor. Zwischen diesen Operationen wäre eine Unterbrechung wegen einer Signalbehandlung denkbar.

In diesem Beispiel versendet der übergeordnete Prozeß 1000 Signale und der neu erzeugte Prozeß zählt die eingetroffenen Signale. Wann darf der übergeordnete Prozeß davon ausgehen, daß der neu erzeugte Prozeß seinen Signalbehandler fertig aufgesetzt hat, so daß er mit dem Zählen beginnen kann? Wenn der übergeordnete Prozeß zu früh Signale versendet, während noch die voreingestellte Reaktion für `SIGUSR1` eingerichtet ist, würde dies nur zur vorzeitigen Terminierung des erzeugten Prozesses führen. Diese Problematik läßt sich vermeiden, indem die Signale, für die der erzeugte Prozeß Behandler aufsetzt, vor der Prozeßerzeugung blockiert werden. Das geht am einfachsten mit der Funktion `sighold()` (auf Zeile 19), die zum POSIX-Standard gehört. Zu jedem Prozeß unterhält das Betriebssystem eine Menge blockierter Signale. Mit `sighold()` wird das angegebene Signal zu der Menge hinzugefügt. Entscheidend ist hier, daß die Menge der blockierten Signale an den neu erzeugten Prozeß vererbt wird. So können nach dem `fork()` in aller Ruhe auf den Zeilen 26 bis 34 die Signalbehandler für `SIGUSR1` und `SIGTERM` aufgesetzt werden, bevor auf Zeile 35 diese Signale wieder mit Hilfe von `sigrelse()` (eine unglückliche Abkürzung von *signal release*) wieder aus der Menge der blockierten Signale entfernt werden. Auch der übergeordnete Prozeß nimmt die beiden Signale wieder heraus auf der Zeile 42.

sigfire.c

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

static const int NOF_SIGNALS = 1000;
static volatile sig_atomic_t received_signals = 0;
static volatile sig_atomic_t terminated = 0;

static void count_signals(int sig) {
   ++received_signals;
}

void termination_handler(int sig) {
   terminated = 1;
}

int main() {
   sighold(SIGUSR1); sighold(SIGTERM);

   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      struct sigaction action = {0};
      action.sa_handler = count_signals;
      if (sigaction(SIGUSR1, &action, 0) != 0) {
         perror("sigaction"); exit(1);
      }
      action.sa_handler = termination_handler;
      if (sigaction(SIGTERM, &action, 0) != 0) {
         perror("sigaction"); exit(1);
      }
      sigrelse(SIGUSR1); sigrelse(SIGTERM);
      while (!terminated) pause();
      printf("[%d] received %d signals\n",
         (int) getpid(), received_signals);
      exit(0);
   }

   sigrelse(SIGUSR1); sigrelse(SIGTERM);
   for (int i = 0; i < NOF_SIGNALS; ++i) {
      kill(child, SIGUSR1);
   }
   printf("[%d] sent %d signals\n",
      (int) getpid(), NOF_SIGNALS);
   kill(child, SIGTERM); wait(0);
}
```

Programmtext 3.6: Verlust von Signalen

Wie wird das Experiment beendet? Da, wie das Experiment zeigen soll, Signale verloren gehen können, sollte der erzeugte Prozeß nicht darauf warten, bis `NOF_SIGNALS` eingetroffen sind. Stattdessen wird `SIGTERM` vom übergeordneten Prozeß an den erzeugten Prozeß verwendet, um das Ende des Experiments zu signalisieren.

Hier sind einige Läufe des Experiments, die demonstrieren, wie sehr die Werte voneinander abweichen können:

```
doolin$ sigfire
[22073] sent 1000 signals
[22074] received 264 signals
doolin$ sigfire
[22075] sent 1000 signals
[22076] received 227 signals
doolin$ sigfire
[22077] sent 1000 signals
[22078] received 481 signals
doolin$ sigfire
[22079] sent 1000 signals
[22080] received 136 signals
doolin$
```

Wenn anstelle von nur `SIGUSR1` zwei Signale `SIGUSR1` und `SIGUSR2` abwechselnd verwendet werden, können höhere Werte erzielt werden, wobei der Erfolg keinesfalls garantiert ist:

```
doolin$ sigfire2
[22142] sent 1000 signals
[22143] received 495 signals
doolin$ sigfire2
[22144] sent 1000 signals
[22145] received 462 signals
doolin$ sigfire2
[22146] sent 1000 signals
[22147] received 468 signals
doolin$ sigfire2
[22151] sent 1000 signals
[22152] received 688 signals
doolin$
```

Dieses Experiment untermauert die Regel, daß einzelne Signale zuverlässig zugestellt werden, während es bei dem Mehrfachen Eintreffen des gleichen Signals zu Verlusten kommen kann. In der Praxis zeigen sich jedoch Signalverluste nur bei härteren Rahmenbedingungen, sei es durch ein explizites Dauerfeuer (wie in diesem Experiment) oder durch eine hohe Belastung der Maschine.

3.6 Signale als Indikatoren für terminierte Prozesse

Mit Hilfe der Funktionen `wait()` oder `waitpid()` wird die Terminierung erzeugter Prozesse **synchron** abgewickelt. Gelegentlich ist es auch sinnvoll, sich die Terminierung über Signale **asynchron** mitteilen zu lassen. Dies geht mit dem Signal `SIGCHLD`, das an den Erzeuger processlist.h

```
#ifndef PROCESSLIST_H
#define PROCESSLIST_H

typedef struct process {
   pid_t pid; int wstat;
   struct process* next;
} process;
typedef struct processlist {
   unsigned int size, length;
   process** bucket; /* hash table */
   unsigned int it_index;
   process* it_entry;
} processlist;

// All functions with the exception of pl_length, pl_next,
// and pl_pick return 1 on success, 0 in case of failures.

/* allocate a hash table for processes with the given bucket size */
int pl_alloc(processlist* pl, unsigned int size);

/* add tuple (pid,wstat) to the process list, pid must be unique */
int pl_add(processlist* pl, pid_t pid, int wstat);

/* modify wstat for a given pid */
int pl_modify(processlist* pl, pid_t pid, int wstat);

/* delete tuple by pid */
int pl_remove(processlist* pl, pid_t pid);

/* move entry for pid to another list */
int pl_move(processlist* from, processlist* to, pid_t pid);

/* return number of elements */
unsigned int pl_length(processlist* pl);

/* lookup wstat by pid */
int pl_lookup(processlist* pl, pid_t pid, int* wstat);

/* start iterator */
int pl_start(processlist *pl);

/* fetch next pid from iterator; returns 0 on end */
pid_t pl_next(processlist *pl);

/* pick and remove one element out of the list */
pid_t pl_pick(processlist *pl, int* wstat);

/* free allocated memory */
int pl_free(processlist* pl);
#endif
```

Programmtext 3.7: Schnittstelle der Bibliothek für Listen von Prozessen sigchld.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "processlist.h"

static processlist alive, dead;

void child_term_handler(int sig) {
   pid_t pid; int wstat;
   while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
      if (pl_move(&alive, &dead, pid)) {
         pl_modify(&dead, pid, wstat);
      }
   }
}

int main() {
   struct sigaction action = {0};
   action.sa_handler = child_term_handler;
   if (sigaction(SIGCHLD, &action, 0) != 0) {
      perror("sigaction");
   }
   pl_alloc(&alive, 4); pl_alloc(&dead, 4);
   sighold(SIGCHLD);
   for (int i = 0; i < 10; ++i) {
      fflush(0);
      pid_t child = fork();
      if (child < 0) {
         perror("fork"); exit(1);
      }
      if (child == 0) {
         srand(getpid()); sleep(rand() % 5);
         exit((char) rand());
      }
      pl_add(&alive, child, 0);
   }
   sigrelse(SIGCHLD);
   while (pl_length(&alive) > 0) {
      if (pl_length(&dead) == 0) pause();
      while (pl_length(&dead) > 0) {
         int wstat;
         sighold(SIGCHLD);
         pid_t pid = pl_pick(&dead, &wstat);
         sigrelse(SIGCHLD);
         printf("[%d] %d\n", (int) pid, WEXITSTATUS(wstat));
      }
   }
}
```

Programmtext 3.8: Demonstration von `SIGCHLD` versendet wird, sobald eine der von ihm erzeugten Prozesse terminiert. Per Voreinstellung wird dieses Signal ignoriert.

Programmtext 3.8 demonstriert die Verwendung von `SIGCHLD`. In den Zeilen 27 bis 38 werden zehn Prozesse erzeugt, die nach einem zufällig bestimmten Zeitintervall mit einem zufälligen `exit`-Wert enden. Die Prozesse werden in der Zeile 37 in die Liste der noch lebenden Prozesse `alive` eingetragen. Programmtext 3.7 zeigt die Schnittstelle der Prozeßverwaltung.

Der Signalbehandler `child_term_handler` auf den Zeilen 10 bis 17 verzeichnet alle Eingänge des `SIGCHLD`-Signals, indem mit Hilfe von `waitpid()` der Status abgeholt wird und der terminierte Prozess von der Liste `alive` nach `dead` verschoben wird. Das Problem ist, daß bei einer zeitgleichen Terminierung mehrerer Prozesse es wiederum zu Verlusten des `SIGCHLD`-Signals kommen kann. Somit können wir uns nicht darauf verlassen, daß für jeden terminierten Prozeß der Signalbehandler genau einmal aufgerufen wird. Deswegen empfiehlt es sich, den Status aller bereits terminierter Prozesse abzurufen. Dies wird mit `waitpid()` unter der Verwendung der Option `WNOHANG` erreicht. Diese verhindert ein Blockieren des Systemaufrufs `waitpid()` und somit kann `waitpid()` gefahrlos solange aufgerufen werden, bis `waitpid()` 0 oder einen negativen Wert zurückliefert.

Um die Prozesse und deren Status zu erfassen, sind umfangreichere Datenstrukturen erforderlich, die nicht mehr ohne weiteres mit dem Attribut `volatile` versehen werden können. Außerdem sollte es nicht dazu kommen, daß das Entfernen eines Eintrages aus der Liste der toten Prozesse in Zeile 45 von einer Signalbehandlung unterbrochen wird, die in Zeile 13 in genau die gleiche Liste, einen Eintrag hinzuzufügen versucht. Nur ein zuverlässiger gegenseitiger Ausschluß bewahrt uns hier vor Überraschungen.

Dies wird erreicht, indem gleich zu Beginn in Zeile 26 das Signal `SIGCHLD` blockiert wird. Das ist schon aus dem Grund wichtig, daß die Liste der noch lebenden Prozesse gefüllt werden kann, bevor diese Prozesse die Gelegenheit haben, sich allzu frühzeitig zu verabschieden. Nachdem alle zehn Prozesse erzeugt worden sind, wird in Zeile 39 die Blockierung wieder aufgehoben. Später, wenn die Liste `dead` modifiziert werden soll, wird die kritische Operation mit Hilfe von `sighold()` und `sigrelse()` wiederum geschützt.

Wenn auf gemeinsame Datenstrukturen von mehreren Seiten in asynchroner Form zugegriffen werden kann, dann sind die zugreifenden Programmbereiche sogenannte **kritische Regionen**. Nur durch den gegenseitigen Ausschluß wird verhindert, daß die betroffene Datenstruktur durch einen ungünstigen Unterbrechungszeitpunkt inkonsistent wird. Da `sigaction()` anstelle von `signal()` verwendet wird, ist bereits sichergestellt, daß `child_term_handler()` nicht mehrfach in verschachtelter Form aufgerufen wird. Somit müssen nur alle verbliebenen Programmbereiche, die auf die gleiche Datenstruktur zugreifen, in `sighold()` und `sigrelse()` geklammert werden.

3.7 Signalbehandlung in einer Shell

Die mit dem Programmtext 2.11 vorgestellte einfache Shell kümmerte sich nicht um die Signalbehandlung. Dies kann zu überraschenden Effekten führen, wenn der Versuch unternommen wird, aufgerufene Prozesse beispielsweise mit `SIGINT` zu unterbrechen:

```
doolin$ tinysh
% cat >OUT
Some input...
^Cdoolin$
```

Hier wurde zunächst `cat` aufgerufen und nach der ersten eingegebenen Zeile CTRL-c eingegeben, welches bei den aktuellen Einstellungen zu einem SIGINT an alle Prozesse der aktuellen Sitzung führte. Zur aktuellen Sitzung gehört jedoch nicht nur das gerade laufende `cat`-Kommando, sondern natürlich auch `tinysh`. Da `tinysh` keinerlei Vorkehrungen traf, wurde es genauso wie `cat` einfach terminiert, weil dies die voreingestellte Reaktion ist. Wäre `tinysh` die Login-Shell gewesen, wäre damit die gesamte Sitzung beendet. Hier in diesem Beispiel wurde `SIGINT` offensichtlich von der normalen Shell ignoriert.

Wie muß also die Signalbehandlung einer Shell aussehen?

- Wenn ein Kommando **im Vordergrund** läuft, muß die Shell die Signale `SIGINT` und `SIGQUIT` ignorieren.
- Wenn ein Kommando **im Hintergrund** läuft, müssen für diesen Prozeß `SIGINT` und `SIGQUIT` ignoriert werden.
- Wenn die Shell ein Kommando einliest, sollten `SIGINT` und `SIGQUIT` die Neu-Eingabe des Kommandos ermöglichen.
- Bezüglich `SIGHUP` muß nichts unternommen werden.

Programmtext 3.9 zeigt das Hauptprogramm einer einfachen Shell mit Signalbehandlung. Das zeilenweise Einlesen wurde in die Funktion `getline()` ausgelagert, die Unterbrechungen berücksichtigt (siehe Programmtext 3.10). Bei jeder sich bietenden Gelegenheit werden nicht-blockierend mit `waitpid()` die Statusinformationen der inzwischen beendeten Prozesse abgeholt (siehe Programmtext 3.11). Das Parsieren einer Kommandozeile wurde ausgelagert, wobei als weiteres unterstütztes Token noch "&" hinzugekommen ist, das für die Ausführung im Hintergrund steht (siehe Programmtext 3.12).

3.8 Überblick der Signale aus dem POSIX-Standard

Die Tabelle 3.2 liefert einen Überblick aller vom POSIX-Standard genannten Signale. Einzelne Implementierungen können noch weitere Signale unterstützen. Die Signale lassen sich dabei in mehrere Gruppen aufteilen:

- Programmierfehler: `SIGBUS`, `SIGFPE`, `SIGILL`, `SIGSEGV` und `SIGSYS`.
- Ressourcenverbrauch: `SIGVTALRM`, `SIGXCPU` und `SIGXFSZ`.
- Prozeßkontrolle: `SIGCONT`, `SIGKILL`, `SIGSTOP`, `SIGTERM` und `SIGTRAP`.
- Sitzungskontrolle: `SIGHUP`, `SIGINT`, `SIGQUIT` und `SIGTSTP`.
- Ereignis-Indikatoren: `SIGALRM`, `SIGCHLD`, `SIGPIPE`, `SIGPOLL`, `SIGPROF`, `SIGTTIN`, `SIGTTOU`, `SIGURG`, `SIGUSR1`, `SIGUSR2` und `SIGVTALRM`.

tinysh2.c

```
static sig_atomic_t interrupted = 0;

void interrupt_handler(int sig) {
   interrupted = 1;
}

int main() {
   struct sigaction action = {0};
   action.sa_handler = interrupt_handler;
   if (sigaction(SIGINT, &action, 0) != 0 ||
         sigaction(SIGQUIT, &action, 0) != 0) {
      perror("sigaction");
   }

   stralloc line = {0};
   while (getline(&line)) {
      strlist tokens = {0};
      stralloc_0(&line); /* required by tokenizer() */
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      command cmd = {0};
      if (!scan_command(&tokens, &cmd)) continue;

      sighold(SIGINT); sighold(SIGQUIT);
      pid_t child = fork();
      if (child == -1) {
         perror("fork"); continue;
      }
      if (child == 0) {
         sigrelse(SIGINT); sigrelse(SIGQUIT);
         if (cmd.background) {
            sigignore(SIGINT); sigignore(SIGQUIT);
         }
         exec_command(&cmd);
         perror(cmd.cmdname);
         exit(255);
      }

      if (cmd.background) {
         printf("%d\n", (int)child);
      } else {
         int wstat;
         pid_t pid = waitpid(child, &wstat, 0);
         if (!WIFEXITED(wstat) || WEXITSTATUS(wstat)) {
            print_child_status(pid, wstat);
         }
      }
      sigrelse(SIGINT); sigrelse(SIGQUIT);
   }
}
```

Programmtext 3.9: Hauptprogramm einer einfachen Shell mit Signalbehandlung tinysh2.c

```
int getline(stralloc* line) {
   int first = 1;
   interrupted = 0;
   for(;;) {
      if (interrupted) {
         interrupted = 0;
         printf("\n");
         first = 1;
      }
      if (first) {
         status_report();
         printf("%% ");
         first = 0;
      }
      errno = 0;
      if (readline(stdin, line)) return 1;
      if (errno != EINTR) return 0;
   }
}
```

Programmtext 3.10: Zeilenweisen Einlesen unter Berücksichtung von Unterbrechungen

tinysh2.c

```
void print_child_status(pid_t pid, int wstat) {
   printf("[%d] ", (int) pid);
   if (WIFEXITED(wstat)) {
      printf("exit %d", WEXITSTATUS(wstat));
   } else if (WIFSIGNALED(wstat)) {
      printf("terminated with signal %d", WTERMSIG(wstat));
      if (WCOREDUMP(wstat)) printf(" (core dump)");
   } else {
      printf("???");
   }
   printf("\n");
}

void status_report(void) {
   pid_t pid; int wstat;
   while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
      print_child_status(pid, wstat);
   }
}
```

Programmtext 3.11: Abholen und Ausgabe von Statusinformationen command.h

```
#ifndef COMMAND_H
#define COMMAND_H

#include <fcntl.h>
#include "strlist.h"

typedef struct fd_assignment {
   char* path;
   int oflags;
   mode_t mode;
} fd_assignment;

typedef struct command {
   char* cmdname;
   strlist argv;
   int background;
   /* for file descriptors 0 and 1 */
   fd_assignment assignments[2];
} command;

/* convert list of tokens into a command record */
int scan_command(strlist* tokens, command* cmd);

/*
 * open input and output files, if required, and
 * exec to the given command
 */
void exec_command(command* cmd);

#endif
```

Programmtext 3.12: Schnittstelle für die Behandlung von Kommandos

Signal	Voreinstellung	Beschreibung
SIGABRT	A	Process abort signal.
SIGALRM	T	Alarm clock.
SIGBUS	A	Access to an undefined portion of a memory object.
SIGCHLD	I	Child process terminated, stopped, or continued.
SIGCONT	C	Continue executing, if stopped.
SIGFPE	A	Erroneous arithmetic operation.
SIGHUP	T	Hangup.
SIGILL	A	Illegal instruction.
SIGINT	T	Terminal interrupt signal.
SIGKILL	T	Kill (cannot be caught or ignored).
SIGPIPE	T	Write on a pipe with no one to read it.
SIGQUIT	A	Terminal quit signal.
SIGSEGV	A	Invalid memory reference.
SIGSTOP	S	Stop executing (cannot be caught or ignored).
SIGTERM	T	Termination signal.
SIGTSTP	S	Terminal stop signal.
SIGTTIN	S	Background process attempting read.
SIGTTOU	S	Background process attempting write.
SIGUSR1	T	User-defined signal 1.
SIGUSR2	T	User-defined signal 2.
SIGPOLL	T	Pollable event.
SIGPROF	T	Profiling timer expired.
SIGSYS	A	Bad system call.
SIGTRAP	A	Trace/breakpoint trap.
SIGURG	I	High bandwidth data is available at a socket.
SIGVTALRM	T	Virtual timer expired.
SIGXCPU	A	CPU time limit exceeded.
SIGXFSZ	A	File size limit exceeded.

Voreinstellung	Beschreibung
T	Abbruch des Prozesses. Bei dem bei `wait()` zurückgelieferten Status ist `WIFSIGNALED` wahr und über `WTERMSIG` läßt sich das Signal ermitteln.
A	Analog zu T. Hinzu kommt möglicherweise noch die Erzeugung eines Speicherauszugs (in der Datei `core`). Letzteres läßt sich mit `WCOREDUMP` untersuchen.
I	Das Signal wird ignoriert.
S	Der Prozeß wird gestoppt.
C	Der Prozeß wird fortgesetzt.

Tabelle 3.2: Im POSIX-Standard genannte Signale (Quelle: www.opengroup.org)

Kapitel 4

Interprozeß-Kommunikation

4.1 Pipelines

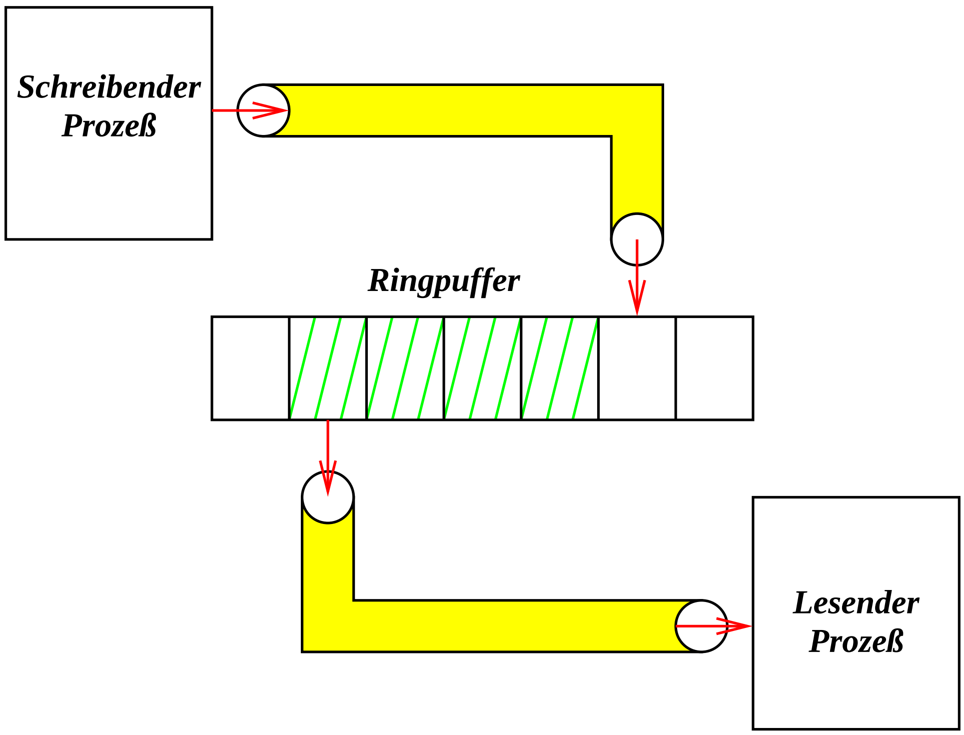

Abbildung 4.1: Aufbau einer Pipeline

Eine Pipeline ist ein unidirektionaler Kommunikationskanal (siehe Abbildung 4.1). Die schreibende und die lesende Seite einer Pipeline werden über verschiedene Dateiverbindungen angesprochen. Zur Pipeline gehört ein vom Betriebssystem verwalteter Ringpuffer, dessen Größenordnung typischerweise bei 4 Kilobyte liegt. Wenn der Puffer vollständig gefüllt ist, wird ein Prozeß, der ihn weiter zu füllen versucht, blockiert, bis wieder genügend Platz zur Verfügung steht. Wenn der Puffer leer ist, wird ein lesender Prozeß blockiert, bis der Puffer sich zumindest partiell füllt. Dies ist vergleichbar mit der Datenstruktur einer FIFO-Queue (*first in, first out*) mit explizit begrenzter Kapazität.

Der POSIX-Standard unterstützt sowohl benannte Pipelines als auch solche, die mit Hilfe des Systemaufrufs `pipe()` erzeugt werden. Die benannten Pipelines sind aber kaum noch in Gebrauch, da die bidirektionalen UNIX-Domain-Sockets (mehr dazu später) normalerweise bevorzugt werden.

pipehello.c

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  
5  const int PIPE_READ = 0;
6  const int PIPE_WRITE = 1;
7  
8  int main() {
9     int pipefds[2];
10    if (pipe(pipefds) < 0) {
11       perror("pipe"); exit(1);
12    }
13    pid_t child = fork();
14    if (child < 0) {
15       perror("fork"); exit(1);
16    }
17    if (child == 0) {
18       close(pipefds[PIPE_WRITE]);
19       char buf[32];
20       ssize_t nbytes;
21       while ((nbytes = read(pipefds[PIPE_READ],
22             buf, sizeof buf)) > 0) {
23          if (write(1, buf, nbytes) < nbytes) exit(1);
24       }
25       exit(0);
26    }
27    close(pipefds[PIPE_READ]);
28    const char message[] = "Hello!\n";
29    write(pipefds[PIPE_WRITE], message, sizeof message - 1);
30    close(pipefds[PIPE_WRITE]);
31    wait(0);
32 }
```

Programmtext 4.1: Aufbau und Verwendung einer Pipeline

Programmtext 4.1 zeigt den Aufbau und die Verwendung einer Pipeline an einem einfachen Beispiel. In Zeile 10 wird die Pipeline mit Hilfe des Systemaufrufs `pipe()` angelegt. Dabei werden zwei Dateideskriptoren zurückgegeben — einen Dateideskriptor für die lesende und einen für die schreibende Seite.

Nach dem `fork()` auf Zeile 13 sind beide Dateideskriptoren für die beiden Enden der Pipeline von beiden Prozessen aus zugänglich. Es empfiehlt sich, unmittelbar nach dem `fork()` das jeweils unbenützte Ende zu schließen. In diesem Beispiel soll das schreibende Ende beim erzeugenden Prozeß verbleiben und das lesende Ende beim Kind-Prozeß. Entsprechend schließt auf Zeile 18 der Kind-Prozeß das schreibende Ende und auf Zeile 27 der Erzeuger das lesende

Ende. Danach hat jeder der beiden Prozesse nur die eigene Seite zur Pipeline geöffnet. Dies ist sehr wichtig, da ein Eingabeende bei einer Pipeline nur dann erkannt werden kann, wenn alle Dateiverbindungen geschlossen sind, über die etwas in die Pipeline gefüllt werden kann. Sobald auf Zeile 30 der Erzeuger seine Schreib-Verbindung zur Pipeline schließt, kann die Schleife des Kind-Prozesses in Zeile 21 terminieren, die die komplette Eingabe aus der Pipeline zum Dateideskriptor 1 (Standard-Ausgabe) kopiert. Wenn die `close()`-Operation auf Zeile 18 fehlen würde, dann käme es zu einem endlosen Hänger bei der `read()`-Operation in Zeile 21.

Was geschieht im umgekehrten Falle, wenn versucht wird, in eine Pipeline zu schreiben, bei der alle lesenden Verbindungen geschlossen sind? Glücklicherweise wird der schreibende Prozeß nicht endlos blockiert, sondern erhält das Signal `SIGPIPE`. Zudem wird bei einer `write()`-Operation, die wegen einer geschlossenen Pipeline fehlschlägt, `EPIPE` als Fehlerindikation zurückgegeben. Aber auch hierfür ist es notwendig, daß alle überflüssigen Verbindungen zur Pipeline geschlossen worden sind. Programmtext 4.2 demonstriert die Berücksichtigung des `SIGPIPE`-Signals. Zu beachten ist auch hier die voreingestellte Reaktion: Wenn es keine Signalbehandlung für `SIGPIPE` gibt, wird der schreibende Prozess einfach terminiert.

Wozu können Pipelines sinnvoll eingesetzt werden? Praktisch kann die Verwendung in einem Fork-and-Join-Szenario sein (siehe Abbildung 2.2), wenn es darum geht, Ergebnisse von den Unterprozessen abzuholen. Das hätte beispielsweise bei dem Programmtext 2.7 genutzt werden können, um gefundene Lösungen an den übergeordneten Prozess wieder zurückzugeben.

Sehr viel häufiger ist allerdings der Einsatz von Pipelines, um die Standard-Eingabe oder Standard-Ausgabe eines Prozesses mit einem anderen Prozess zu verbinden. Hierfür bietet POSIX die Funktionen `popen()` und `pclose()`, die allerdings unvermeidbar die Shell mit all ihren Sicherheitsrisiken und die Verwendung der `stdio` mit sich bringen. Programmtext 4.3 zeigt eine alternative Schnittstelle, die analog zu `execvp()` mit Argumentlisten arbeitet. Nicht zu vergessen ist dabei eine Verwaltungsstruktur, die sich die Prozeß-ID merkt und die es ermöglicht, auf das Ende des Prozesses an der anderen Seite der Pipeline zu warten. `pconnect()` initialisiert im Erfolgsfalle eine entsprechende Struktur vom Typ `pipe_end`, und `phangup()` erlaubt es, die Pipeline zu schließen und auf das Ende der anderen Seite zu warten.

Programmtext 4.4 zeigt die zugehörige Implementierung. Nach der Erzeugung der Pipeline in Zeile 17 wird festgelegt, welche der beiden Dateideskriptoren welcher Seite gehört. In Zeile 19 werden alle Puffer der `stdio`-Bibliothek geleert. Der Kind-Prozeß, der die gegenüberliegende Seite vertritt, schließt in Zeile 26 das unbenutzte Ende der Pipeline und sorgt in Zeile 27 dafür, daß der entsprechende Standard-Deskriptor (0 für Standardeingabe und 1 für Standardausgabe) der Pipeline zugeordnet wird. Der zuvor verwendete Dateideskriptor für das Ende der Pipeline wird dann in Zeile 28 geschlossen.

Der übergeordnete Prozeß schließt nicht nur auf Zeile 31 das andere Ende, sondern setzt für das eigene Ende die Option `FD_CLOEXEC`, die bewirkt, daß der betreffende Dateideskriptor nicht über einen `exec()`-Systemaufruf hinweg vererbt, sondern geschlossen wird. Dies ist zwingend notwendig, wenn weitere Prozesse erzeugt werden, um zu verhindern, daß versehentlich Zugänge zu einer Pipeline offenbleiben, wenn die eigentlichen Beteiligten sie nicht mehr benötigen. Besser als ein *close-on-exec* wäre natürlich ein implizites `close()` bei `fork()`. Leider sieht dies der POSIX-Standard nicht vor. Entsprechend ist die Verwendung von Pipelines nur bei `fork()`/`exec()`-Kombinationen sicher, jedoch nicht bei der Erzeugung länger laufender Unterprozesse, die nicht mit `exec()` zu einem anderen Programmtext wechseln.

Programmtext 4.5 demonstriert, wie `pconnect()` verwendet werden kann, um die Ausgabe eines Kommandos einzulesen. In diesem Beispiel geht es um die Ausgabe des `rwho`-Kommandos, sigpipe.c

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

const int PIPE_READ = 0;
const int PIPE_WRITE = 1;

volatile sig_atomic_t sigpipe_received = 0;

void sigpipe_handler(int sig) {
   sigpipe_received = 1;
}

int main() {
   int pipefds[2];
   if (pipe(pipefds) < 0) {
      perror("pipe"); exit(1);
   }
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      close(pipefds[PIPE_WRITE]);
      char buf[32];
      ssize_t nbytes = read(pipefds[PIPE_READ],
            buf, sizeof buf);
      if (nbytes > 0) {
         if (write(1, buf, nbytes) < nbytes) exit(1);
      }
      exit(0);
   }
   close(pipefds[PIPE_READ]);
   struct sigaction action = {0};
   action.sa_handler = sigpipe_handler;
   if (sigaction(SIGPIPE, &action, 0) < 0) {
      perror("sigaction"); exit(1);
   }
   while (!sigpipe_received) {
      const char message[] = "Hello!\n";
      write(pipefds[PIPE_WRITE], message, sizeof message - 1);
   }
   close(pipefds[PIPE_WRITE]);
   wait(0);
}
```

Programmtext 4.2: Demonstration von `SIGPIPE` pconnect.h

```
/*
 * Create and manage pipelines to given commands.
 * In comparison to popen(), neither stdio nor the shell are used.
 * afb 6/2003
 */
#ifndef PCONNECT_H
#define PCONNECT_H

#include <unistd.h>

#define PIPE_READ 0
#define PIPE_WRITE 1

typedef struct pipe_end {
   int fd;
   pid_t pid;
   int wstat;
} pipe_end;

/*
 * create a pipeline to the given command;
 * mode should be either PIPE_READ or PIPE_WRITE;
 * return a filled pipe_end structure and 1 on success
 * and 0 in case of failures
 */
int pconnect(const char* path, char* const* argv,
      int mode, pipe_end* pipe_con);

/*
 * close pipeline and wait for the forked-off process to exit;
 * the wait status is returned in pipe->wstat;
 * 1 is returned if successful, 0 otherwise
 */
int phangup(pipe_end* pipe_end);

#endif
```

Programmtext 4.3: Schnittstelle für Pipelines zu Kommandos pconnect.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include "pconnect.h"

/*
 * create a pipeline to the given command;
 * mode should be either PIPE_READ or PIPE_WRITE;
 * return a filled pipe_end structure and 1 on success
 * and 0 in case of failures
 */
int pconnect(const char* path, char* const* argv,
      int mode, pipe_end* pipe_con) {
   int pipefds[2];
   if (pipe(pipefds) < 0) return 0;
   int myside = mode; int otherside = 1 - mode;
   fflush(0);
   pid_t child = fork();
   if (child < 0) {
      close(pipefds[0]); close(pipefds[1]);
      return 0;
   }
   if (child == 0) {
      close(pipefds[myside]);
      dup2(pipefds[otherside], otherside);
      close(pipefds[otherside]);
      execvp(path, argv); exit(255);
   }
   close(pipefds[otherside]);
   int flags = fcntl(pipefds[myside], F_GETFL);
   flags |= FD_CLOEXEC;
   fcntl(pipefds[myside], F_SETFL, flags);
   pipe_con->pid = child;
   pipe_con->fd = pipefds[myside];
   pipe_con->wstat = 0;
   return 1;
}

/*
 * close pipeline and wait for the forked-off process to exit;
 * the wait status is returned in wstat (if non-null);
 * 1 is returned if successful, 0 otherwise
 */
int phangup(pipe_end* pipe) {
   if (close(pipe->fd) < 0) return 0;
   if (waitpid(pipe->pid, &pipe->wstat, 0) < 0) return 0;
   return 1;
}
```

Programmtext 4.4: Pipelines zu Kommandos rwhousers.c

```
#include <stralloc.h>
#include <string.h>
#include <unistd.h>
#include "pconnect.h"
#include "rwhousers.h"
#include "strlist.h"

const char rwho_path[] = "/usr/bin/rwho";

/*
 * invoke rwho and get list of users that are currently logged in;
 * return 1 in case of success, otherwise 0
 */
int get_rwho_users(strlist* users) {
   strlist argv = {0};
   strlist_push(&argv, rwho_path);
   strlist_push0(&argv);
   pipe_end pipe;
   int ok = pconnect(rwho_path, argv.list, PIPE_READ, &pipe);
   strlist_free(&argv);
   if (!ok) return 0;

   stralloc rwho_output = {0};
   ssize_t nbytes;
   char buf[32];
   while ((nbytes = read(pipe.fd, buf, sizeof buf)) > 0) {
      stralloc_catb(&rwho_output, buf, nbytes);
   }
   phangup(&pipe);

   strlist_clear(users);
   char* user = rwho_output.s;
   for (int i = 0; i < rwho_output.len; ++i) {
      switch (rwho_output.s[i]) {
         case ' ':
            if (user != 0) {
               rwho_output.s[i] = 0;
               strlist_push(users, strdup(user));
               user = 0;
            }
            break;
         case '\n':
            user = rwho_output.s + i + 1;
            break;
      }
   }
   stralloc_free(&rwho_output);
   return 1;
}
```

Programmtext 4.5: Verarbeitung der Ausgabe eines Kommandos das angibt, welche Nutzer sich im gleichen Netzwerk angemeldet haben:[1]

```
turing# rwho
borchert theseus:pts/10  May  9 09:39 :19
borchert theseus:pts/13  Mar  9 13:39 :20
borchert theseus:pts/36  May  9 09:37 :06
borchert theseus:pts/38  Apr 26 13:36 :19
borchert turing:pts/31   Apr 26 13:37 :14
borchert xylopia:pts/4   Apr  8 10:38 :19
jens_k   theseus:pts/97  Jun 14 13:51 :02
rudib    centaurus:pts/2 Jun 14 14:57 :02
rudib    centaurus:pts/4 Jun 14 14:57 :24
francis  kleemann:pts/2  Jun 14 11:02 :17
francis  turing:pts/56   Jun 14 11:02 :18
hschmidt turing:pts/24   Jun 14 14:34
rmack    theseus:pts/49  Jun 14 15:15
fkraft   turing:pts/5    Jun 14 13:53 :52
prince   theseus:pts/84  Jun 14 15:01 :03
drtiger  turing:pts/75   Jun 14 11:51 :01
turing#
```

Die Funktion `get_rwho_users()` aus Programmtext 4.5 soll das Kommando `rwho` aufrufen und aus dessen Ausgabe nur die Namen der Benutzer extrahieren. In dem angezeigten Beispiel würden wir also 6 Mal `borchert` erhalten, gefolgt von `jens_k`, zwei Mal `rudib` und so fort. Zu Beginn eröffnet `get_rwho_users()` in den Zeilen 15 bis 21 die Pipeline. Dann wird in den Zeilen 23 bis 28 die gesamte Ausgabe des `rwho`-Kommandos in einen Puffer vom Typ `stralloc` eingelesen. In Zeile 29 wird die Pipeline geschlossen. Danach wird in den Zeilen 31 bis 46 das jeweils erste Wort jeder Zeile aus dem Puffer in die Liste `users` eingefügt.

Programmtext 4.6 zeigt den umgekehrten Fall, bei dem die Ausgabe für ein Kommando generiert wird. Bei diesem Beispiel geht es um die Erstellung einer E-Mail, wobei `sendmail()` den Aufbau einer Pipeline zu `/usr/lib/sendmail` übernimmt, womit lokal E-Mails versendet werden können. Obwohl `/usr/lib/sendmail` nicht zum POSIX-Standard gehört, ist es auf allen gängigen UNIX-Systemen zu finden, unabhängig von der eingesetzten Mail-Software. Die Option `-t` sorgt dafür, daß die Empfängeradresse der entsprechenden Kopfzeile der E-Mail entnommen wird und nicht auf der Kommandozeile angegeben werden muß. Die Funktion `sendmail()` übernimmt hier nur das Schreiben der Kopfzeilen. Diese werden zunächst auf den Zeilen 21 bis 27 in einen Puffer geschrieben, um sie danach in den Zeilen 29 bis 37 an die Pipeline weiterzugeben. Hier ist zu beachten, daß das `write()` keinesfalls verpflichtet ist, den Puffer in der gewünschten Gesamtlänge in einem Schwung zu übernehmen. Stattdessen steht es `write()` frei, nur einen Teil zu schreiben und die entsprechende Zahl der geschriebenen Bytes zurückzuliefern. Dies ist insbesondere bei Pipelines möglich, wenn die Zahl der zu schreibenden Bytes die zur Verfügung stehende Kapazität im Ringpuffer übersteigt.

Programmtext 4.7 verwendet `get_rwho_users()` und `sendmail()`, um per E-Mail einen Hinweis zu verschicken, wenn auf der Kommandozeile benannte Freunde angemeldet sind. Dabei kommt eine Hash-Organisation für Zeichenketten zum Einsatz (siehe Programmtext 4.8), um Duplikate zu eliminieren und das Abgleichen der Namenslisten zu erleichtern. Die Zeile 34 bei Programmtext 4.7 zeigt noch einen kleinen Trick. Wenn die Absicht besteht, die `stdio` zu verwenden, wenn nur ein Dateideskriptor vorliegt, besteht die Möglichkeit, `fdopen()` zu

[1] Die Benutzernamen sind hier mit der Ausnahme meines eigenen frei erfunden.

sendmail.c

```
#include <stralloc.h>
#include <unistd.h>
#include "pconnect.h"
#include "sendmail.h"
#include "strlist.h"

const char sendmail_path[] = "/usr/lib/sendmail";
/*
 * return a pipeline opened to /usr/lib/sendmail on the
 * local system; return the opened pipeline and 1 in
 * case of success; 0 in case of failures
 */
int sendmail(char* recipient, char* subject, pipe_end* pipe_con) {
   strlist argv = {0};
   strlist_push(&argv, sendmail_path);
   strlist_push(&argv, "-t");
   strlist_push0(&argv);
   int ok = pconnect(sendmail_path, argv.list, PIPE_WRITE, pipe_con);
   strlist_free(&argv);
   if (!ok) return 0;
   stralloc header = {0};
   stralloc_cats(&header, "To: ");
   stralloc_cats(&header, recipient);
   stralloc_cats(&header, "\n");
   stralloc_cats(&header, "Subject: ");
   stralloc_cats(&header, subject);
   stralloc_cats(&header, "\n\n");
   ssize_t written = 0; ssize_t left = header.len;
   while (left > 0) {
      ssize_t nbytes = write(pipe_con->fd, header.s + written, left);
      if (nbytes < 0) {
         stralloc_free(&header);
         phangup(pipe_con);
         return 0;
      }
      written += nbytes; left -= nbytes;
   }
   stralloc_free(&header);
   return 1;
}
```

Programmtext 4.6: Generierung der Eingabe für ein Kommando bigbrother.c

```
#include <stdio.h>
#include <stdlib.h>
#include "pconnect.h"
#include "sendmail.h"
#include "strhash.h"
#include "strlist.h"
#include "rwhousers.h"

int main(int argc, char** argv) {
   if (argc <= 2) {
      fprintf(stderr, "Usage: %s email login...\n", argv[0]);
      exit(1);
   }
   char* email = *++argv; --argc;
   strhash friends = {0};
   strhash_alloc(&friends, 4);
   while (--argc > 0) {
      if (!strhash_add(&friends, *++argv, 0)) exit(1);
   }

   strlist users = {0};
   if (!get_rwho_users(&users)) exit(1);
   strhash found = {0};
   strhash_alloc(&found, 4);
   for (int i = 0; i < users.len; ++i) {
      if (strhash_exists(&found, users.list[i])) continue;
      if (!strhash_exists(&friends, users.list[i])) continue;
      if (!strhash_add(&found, users.list[i], 0)) exit(1);
   }
   if (strhash_length(&found) == 0) exit(0);

   pipe_end pipe_con;
   if (!sendmail(email, "Your Friends Are Online!", &pipe_con))
      exit(1);
   if (dup2(pipe_con.fd, 1) < 0) exit(1);
   printf("Hi, ");
   if (strhash_length(&found) == 1) {
      printf("one of your friends is");
   } else {
      printf("some of your friends are");
   }
   printf(" online:\n");
   strhash_start(&found);
   char* key;
   while (strhash_next(&found, &key)) {
      printf("%s\n", key);
   }
   fclose(stdout);
   if (!phangup(&pipe_con)) exit(1);
}
```

Programmtext 4.7: Anwendung mit zwei Pipeline-Verbindungen strhash.h

```
/*
 * Hashes for strings, i.e. keys and values are of type char*
 * afb 6/2003
 */

#ifndef STRHASH_H
#define STRHASH_H

typedef struct strhash_entry {
   char* key;
   char* value;
   struct strhash_entry* next;
} strhash_entry;

typedef struct strhash {
   unsigned int size, length;
   strhash_entry** bucket; /* hash table */
   unsigned int it_index;
   strhash_entry* it_entry;
} strhash;

/* allocate a hash table with the given bucket size */
int strhash_alloc(strhash* hash, unsigned int size);

/* add tuple (key,value) to the hash, key must be unique */
int strhash_add(strhash* hash, char* key, char* value);

/* remove tuple with the given key from the hash */
int strhash_remove(strhash* hash, char* key);

/* return number of elements */
unsigned int strhash_length(strhash* hash);

/* check existance of a key */
int strhash_exists(strhash* hash, char* key);

/* lookup value by key */
int strhash_lookup(strhash* hash, char* key, char** value);

/* start iterator */
int strhash_start(strhash *hash);

/* fetch next key from iterator; returns 0 on end */
int strhash_next(strhash *hash, char** key);

/* free allocated memory */
int strhash_free(strhash* hash);

#endif
```

Programmtext 4.8: Schnittstelle einer Hash-Organisation für Zeichenketten verwenden oder — wie in diesem Beispiel — mit einem `dup2()` die bisherige Standardausgabe zu schließen und sie danach auf eine andere geöffnete Verbindung verweisen zu lassen. Danach kann `printf()` verwendet werden, um den Inhalt der zu versendenden E-Mail zu schreiben.

Interessant ist der Aufbau größerer Pipeline-Ketten, wie sie z.B. von der Shell unterstützt werden. Abbildung 4.2 zeigt die Struktur bei einer Kette von drei Kommandos, demonstriert am Beispiel "`who | grep borchert | wc -l`". Zu beachten ist dabei, daß die Kette mit dem letzten Kommando beginnt. Der unmittelbare Unterprozeß der Shell ruft also das Kommando "`wc -l`" auf. Bevor es dazu kommt, wird aber eine Pipeline erzeugt, die an einen neu erzeugten Prozeß als Standardausgabe weitervererbt wird, der dann die Ausführung von "`grep borchert`" übernimmt. Doch bevor es dazu kommt, erzeugt der Prozeß analog wie zuvor eine neue Pipeline und einen weiteren Prozeß, der für die Ausführung von "`who`" zuständig ist.

Programmtext 4.9 zeigt eine einfache Datenstruktur für Pipeline-Ketten. Die Felder `cmdname` und `argv` sind direkt an `execvp()` übergebbare Parameter. Das Feld `input` ist entweder 0 (Standard-Eingabe wird einfach übernommen) oder ein Verweis auf die Pipeline-Kette, die die Eingabe für dieses Kommando generiert. Abbildung 4.3 zeigt die zugehörige Datenstruktur für das genannte Beispiel. Die Datenstruktur entspricht dabei genau der benötigten Prozeßstruktur, bei der die Reihenfolge im Vergleich zur Kommandozeile reversiert ist. Die gestrichelten Linien, die den `exit()`-Aufrufen folgen, deuten die Synchronisierung durch das Erkennen des Eingabe-Endes an. Nur beim Kommando, das am Ende der Zeile stand, erfolgt eine Synchronisierung durch `wait()`.

Programmtext 4.10 demonstriert, wie eine Sequenz von Worten, die aus Kommandoargumenten und dem "`|`"-Zeichen bestehen kann, in eine Pipeline-Liste konvertiert werden kann. Diese Aufgabe wird von der Funktion `pipe_constructor()` erledigt, die in der `for`-Schleife in den Zeilen 25 bis 39 durch die Liste der Tokens geht. Wird das Ende eines Kommandos gefunden, sei es durch ein Pipeline-Symbol in Zeile 26 oder ganz am Ende der Token-Liste in Zeile 40, so wird mit der Funktion `new_command()` ein neues Element der Kette angelegt. Da das neueste Element der Kette jeweils auf alle früheren Elemente verweist, erhalten wir genau die gewünschte Reversion.

Programmtext 4.11 zeigt, wie sich die Datenstruktur unmittelbar in die zugehörige Prozeßstruktur umsetzen läßt. Im einfachen Falle, falls die Eingabe nicht von einer Pipeline erfolgt, haben wir nur den Aufruf von `execvp()` in Zeile 39. Wenn jedoch die Eingabe von einer Pipeline kommen soll, wird in Zeile 17 die Pipeline erzeugt, in Zeile 20 der zugehörige Unterprozeß gestartet und danach in den Zeilen 33 bis 37 dafür gesorgt, daß die Standard-Eingabe (und sonst nichts) mit dem lesenden Ende der Pipeline verbunden ist. Der erzeugte Prozeß verbindet in den Zeilen 25 bis 29 die schreibende Seite der Pipeline mit der Standard-Ausgabe. In Zeile 30 erfolgt ein rekursiver Aufruf von `exec_pipecmd()`, womit sichergestellt wird, daß die gesamte Pipeline-Kette abgearbeitet wird. Das Setzen von *close-on-exec* kann hier entfallen, weil die Pipelines hier grundsätzlich in Unterprozessen erzeugt werden, die nicht in Konflikt zu den sonstigen Aktivitäten des Hauptprozesses stehen.

Ein dazu passendes Hauptprogramm findet sich in Programmtext 4.12.

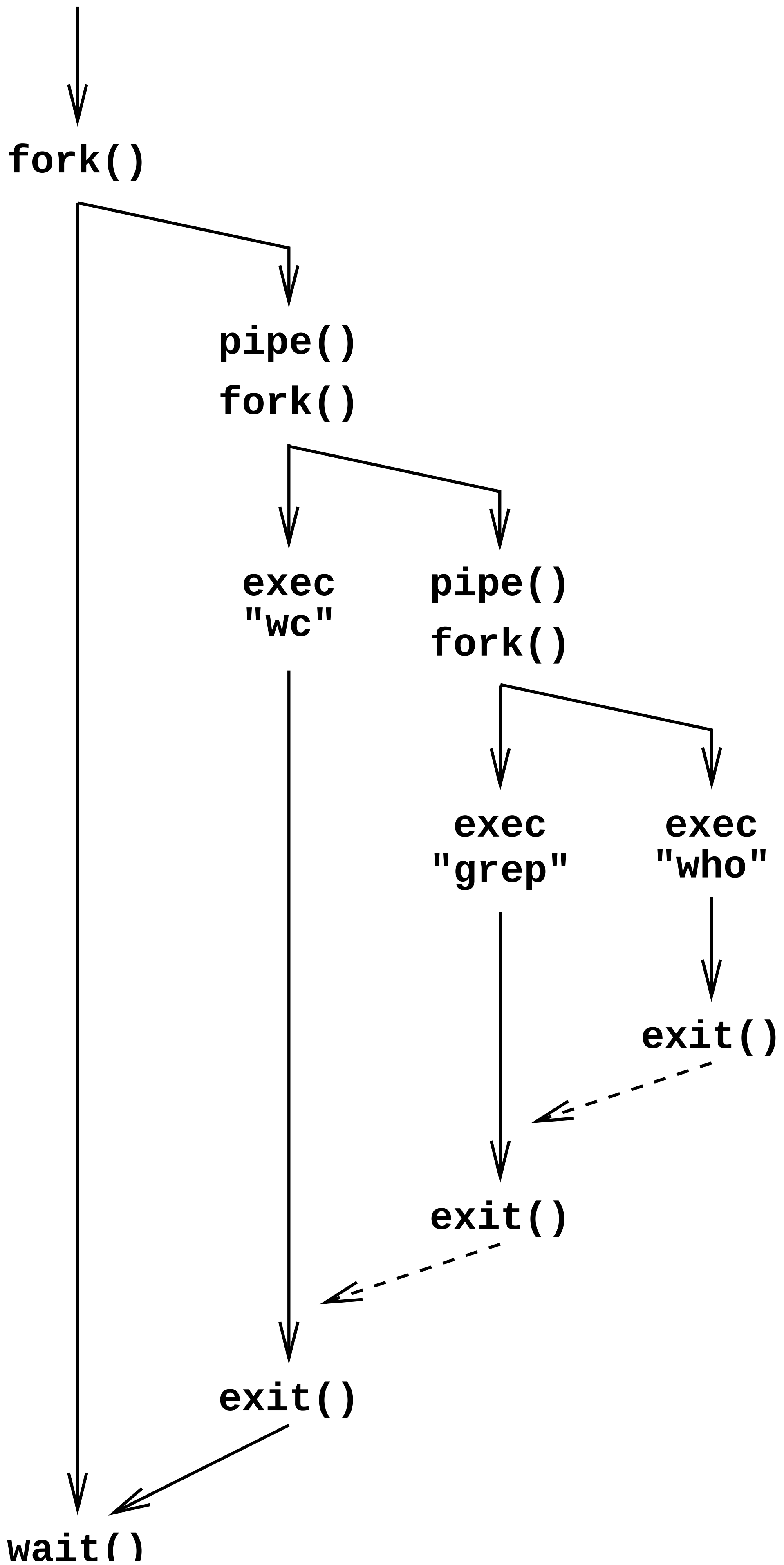

Abbildung 4.2: Prozeßstruktur bei einer Kette von Pipelines

pipecmd.h

```
#ifndef PIPECMD_H
#define PIPECMD_H

#include "strlist.h"

typedef struct pipecmd {
   char* cmdname;
   strlist* argv;
   struct pipecmd* input;
} pipecmd;

/* construct pipeline chain out of tokens */
pipecmd* pipe_constructor(strlist* tokens);

/* free space allocated by pipe_constructor() */
void pipe_free(pipecmd* cmd);

#endif
```

Programmtext 4.9: Datenstruktur für Kommandos einer Pipeline

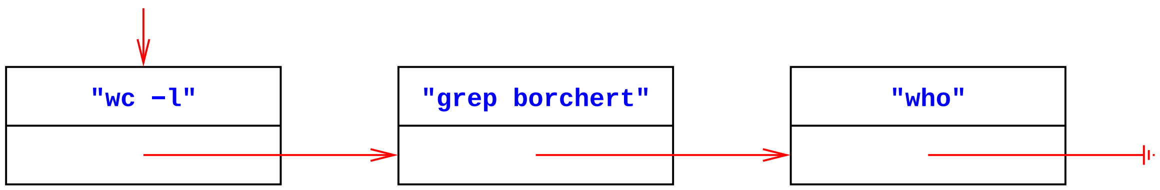

Abbildung 4.3: Datenstruktur für Kommandos einer Pipeline

pipecmd.c

```
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include "pipecmd.h"
#include "strlist.h"

/* generate new pipeline command node */
static pipecmd* new_command(char* cmdname,
      strlist* argv, pipecmd* input) {
   assert(cmdname != 0);
   pipecmd* cmd = (pipecmd*) malloc(sizeof(pipecmd));
   if (cmd == 0) return 0;
   cmd->cmdname = cmdname;
   strlist_push0(argv);
   cmd->argv = argv;
   cmd->input = input;
   return cmd;
}

/* construct pipeline chain out of tokens */
pipecmd* pipe_constructor(strlist* tokens) {
   pipecmd* input = 0;
   char* cmdname;
   strlist* argv = 0;
   for (int index = 0; index < tokens->len; ++index) {
      if (strcmp(tokens->list[index], "|") == 0) {
         if (argv == 0) return 0; /* syntax error */
         input = new_command(cmdname, argv, input);
         cmdname = 0; argv = 0;
      } else {
         if (argv == 0) {
            argv = (strlist*) malloc(sizeof(strlist));
            if (argv == 0) return 0;
            argv->list = 0; argv->len = 0; argv->allocated = 0;
         }
         if (cmdname == 0) cmdname = tokens->list[index];
         if (!strlist_push(argv, tokens->list[index])) return 0;
      }
   }
   if (cmdname != 0) input = new_command(cmdname, argv, input);
   return input;
}

/* free space allocated by pipe_constructor() */
void pipe_free(pipecmd* cmd) {
   if (cmd->input) pipe_free(cmd->input);
   strlist_free(cmd->argv); free(cmd->argv);
   free(cmd);
}
```

Programmtext 4.10: Aufbau einer Pipeline-Liste ausgehend von der Wortliste execpipe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "execpipe.h"
#include "pipecmd.h"

#define PIPE_READ 0
#define PIPE_WRITE 1

/*
 * exec() to the given pipeline command;
 * does not return in case of success
 */
void exec_pipecmd(pipecmd* command) {
   if (command->input != 0) {
      int pipefds[2];
      if (pipe(pipefds) < 0) {
         perror("pipe"); return;
      }
      pid_t child = fork();
      if (child < 0) {
         perror("fork"); return;
      }
      if (child == 0) {
         close(pipefds[PIPE_READ]);
         if (dup2(pipefds[PIPE_WRITE], 1) < 0) {
            perror("dup2"); exit(255);
         }
         close(pipefds[PIPE_WRITE]);
         exec_pipecmd(command->input);
         exit(255);
      }
      close(pipefds[PIPE_WRITE]);
      if (dup2(pipefds[PIPE_READ], 0) < 0) {
         perror("dup2"); return;
      }
      close(pipefds[PIPE_READ]);
   }
   execvp(command->cmdname, command->argv->list);
   perror(command->cmdname);
}
```

Programmtext 4.11: Ausführung einer Pipeline-Kette pipesh.c

```c
#include <stralloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "execpipe.h"
#include "pipecmd.h"
#include "sareadline.h"
#include "strlist.h"
#include "tokenizer.h"

int main() {
   stralloc line = {0};
   while (printf("%% "), readline(stdin, &line)) {
      strlist tokens = {0};
      stralloc_0(&line); /* required by tokenizer() */
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      pipecmd* command = pipe_constructor(&tokens);
      if (command == 0) continue;
      pid_t child = fork();
      if (child < 0) {
         perror("fork"); continue;
      }
      if (child == 0) {
         exec_pipecmd(command);
         exit(255);
      }
      pipe_free(command);

      /* wait for termination of child */
      int stat;
      pid_t pid = wait(&stat);
      if (pid == child) {
         if (WIFEXITED(stat)) {
            int code = WEXITSTATUS(stat);
            if (code && code != 255) {
               printf("terminated with exit code %d\n", code);
            }
         } else {
            printf("terminated abnormally\n");
         }
      } else {
         perror("wait");
      }
   }
}
```

Programmtext 4.12: Hauptprogramm einer einfachen Shell, die Pipelines unterstützt

4.2 Einführung in Netzwerkdienste und TCP/IP

Ein Netzwerkdienst ist ein Prozeß, der unter einer Netzwerkadresse einen Dienst anbietet. Ein Klient, der die Netzwerkadresse kennt, kann einen bidirektionalen Kommunikationskanal zu dem Netzwerkdienst eröffnen und über diesen mit dem Dienst kommunizieren, beispielsweise indem Kommandos auf dem Hinweg übermittelt werden und der Dienst auf dem Rückweg des Kommunikationskanals die Antwort überträgt.

Im Unterschied zu Pipelines müssen die beiden Kommunikationspartner nicht miteinander verwandt sein. Sie müssen nicht einmal auf dem gleichen Rechner laufen. Da der Kommunikationskanal bidirektional ist, wird ein echter Dialog zwischen den beiden Prozessen möglich. Der Aufbau einer Verbindung ist jedoch schwieriger, da die Netzwerkadresse des gewünschten Partners ermittelt werden muß.

Wenn Dienste über das Netzwerk angeboten und in Anspruch genommen werden, ergeben sich viele Vorteile:

- Der Dienst kann allen offenstehen, und ein direkter Zugang zu dem Rechner, auf dem der Dienst angeboten wird, ist nicht notwendig.
- Viele Parteien können in kooperativer Weise einen Dienst gleichzeitig nutzen.
- Der Dienste-Anbieter hat weniger Last, da die Benutzerschnittstelle auf anderen Rechnern laufen kann.

Andererseits bergen Netzwerkdienste eine Reihe von Risiken und Problematiken:

- Der Kreis derjenigen, die auf einen Netzwerkdienst zugreifen können, ist möglicherweise ziemlich umfangreich (normalerweise das gesamte Internet).
- Somit muß jeder Netzwerkdienst Zugriffsberechtigungen einführen und überprüfen und kann sich dabei nicht wie traditionelle Applikationen auf die des Betriebssystems verlassen.
- Dienste, die gleichzeitig von vielen genutzt werden können, haben vielerlei zusätzliche Konsistenz- und Synchronisierungsprobleme, für die nicht jede Art von Datenhaltung geeignet ist.
- Netzwerke bringen neue Arten von Ausfällen mit sich, wenn eine Netzwerkverbindung zusammenbricht oder es zu längeren “Hängern” kommt.

Es gibt eine unüberschaubare Vielfalt an Netzwerk-Hardware, Transport-Protokollen und Schnittstellen zu deren Benutzung. Einen exzellenten Überblick hierfür bietet das Standardwerk “Computer Networks” von Andrew S. Tanenbaum (Prentice Hall, Third Edition, 1996, ISBN 0-13-349945-6). Im weiteren konzentrieren wir uns jedoch auf die Transport-Protokolle des Internets (TCP/IP).

Hier ist ein triviales Beispiel für den Aufbau einer Netzwerkverbindung. Als Klient wird hier `telnet` verwendet, das eine interaktive Nutzung von Netzwerkdiensten ermöglicht. Als trivialer Netzwerkdienst kommt hier `daytime` zum Einsatz, das die Eingabe auf dem Kommunikationskanal ignoriert und nur die aktuelle Zeit analog zum `date`-Kommando zurückgibt:

```
theseus# telnet 134.60.166.2 13
Trying 134.60.166.2...
Connected to 134.60.166.2.
Escape character is '^]'.
Sat Jun 21 14:48:34 2003
Connection to 134.60.166.2 closed by foreign host.
theseus#
```

In diesem Beispiel besteht die Netzwerkadresse aus `134.60.166.2` (einer sogenannten IP-Adresse) und 13 (der Port-Nummer). IP-Adressen dienen dazu, einen Rechner zu adressieren, während Port-Nummern einem Dienst, der auf einem Rechner läuft, zugeordnet werden. In diesem Beispiel adressiert `134.60.166.2` unsere Theseus und die Port-Nummer 13 üblicherweise den `daytime`-Dienst.

Beim Internet gibt es für die Adressierung von Rechnern zwei Adreßräume. IPv4 arbeitet mit 32-Bit-Adressen, ist seit dem 1. Januar 1983 in Benutzung und dominiert bis heute. Daneben gibt es auch IPv6, das mit 128-Bit-Adressen arbeitet, um einen Ausweg zu bieten für die drohende Knappheit mit Adressen bei IPv4. Im weiteren beschäftigen wir uns hier allerdings nur mit IPv4-Adressen, die kurz als IP-Adressen bezeichnet werden.

Die sogenannte *dottet-decimal* Notation für IP-Adressen spezifiziert eine 32-Bit-Adresse als Folge von vier dezimalen Werten, die jeweils ein Byte angeben. `134.60.166.2` ist somit eine etwas übersichtlichere Schreibweise als 2252121602. Port-Nummern liegen im Bereich von 1 bis 65535, wobei der Bereich von 1 bis 1023 typischerweise für wohldefinierte Dienste reserviert ist. Diese Zuordnungen werden von der IANA (*Internet Assigned Numbers Authority*) vergeben und sind unter `http://www.iana.org/assignments/port-numbers` zu finden.

Beide Adreßräume werden in hierarchisierter Form verwaltet. Ganz oben steht IANA, das große Teile des Adreßraumes an regionale Institutionen weitergibt (ARIN für Amerika, RIPE für Europa, Afrika und Mittlerer Osten und APNIC für Asien, Australien und Ozeanien). Die Universität Ulm hat seit 1989 den Adreßbereich 134.60.0.0/16[2] (damals noch von ARIN) zugeteilt bekommen.

Die Abbildung von Rechnernamen wie theseus.mathematik.uni-ulm.de in IP-Adressen wie `134.60.166.2` erfolgt dabei durch sogenannte Domain-Server, die ebenfalls hierarchisch organisiert sind. Wenn jemand von außerhalb den Namen theseus.mathematik.uni-ulm.de angibt, wird zunächst untersucht, wer für die Domain "de" zuständig ist. Dies geschieht durch eine Anfrage an einen der sogenannten Root-Server, beispielsweise `198.41.0.4`, der selbst den Namen a.root-servers.net trägt. Dieser liefert unter anderem den Name-Server `81.91.161.5` mit dem Namen dns.denic.de. Wenn dns.denic.de nach uni-ulm.de gefragt wird, wird u.a. auf `134.60.66.21` mit dem Namen titania.mathematik.uni-ulm.de verwiesen. Letzterer Name-Server kann dann endlich auch theseus.mathematik.uni-ulm.de in `134.60.166.2` abbilden.

IP-Adressen wie `134.60.166.2` werden nur auf einer abstrakten Ebene zur Verfügung gestellt. IP-Adressen werden auf der darunterliegenen physischen Ebene und denen damit verbundenen Protokollen nicht verstanden. So wird beispielsweise beim Ethernet, das bei uns weitgehend an der Universität zum Einsatz kommt, mit 6-Byte-Adressen gearbeitet. Die Theseus hat beispielsweise die Ethernet-Adresse `8:0:20:b2:73:92` (Bytes werden hier in Form von Hexzahlen angegeben). Diese Adressen sind jedoch nur lokal auf einem Ethernet-Segment von Bedeutung.

[2]Diese Schreibweise wird als CIDR-Notation bezeichnet, wobei CIDR für *Classless Inter-Domain Routing* steht. Die Zahl hinter dem Schrägstrich gibt dabei die Zahl der führenden Bits an, die den Netzwerkteil ausmachen. Bei 16 sind das die ersten beiden Bytes, also `134.60`. Alle Adressen in der Form `134.60.*.*` gehören somit zur Universität Ulm.

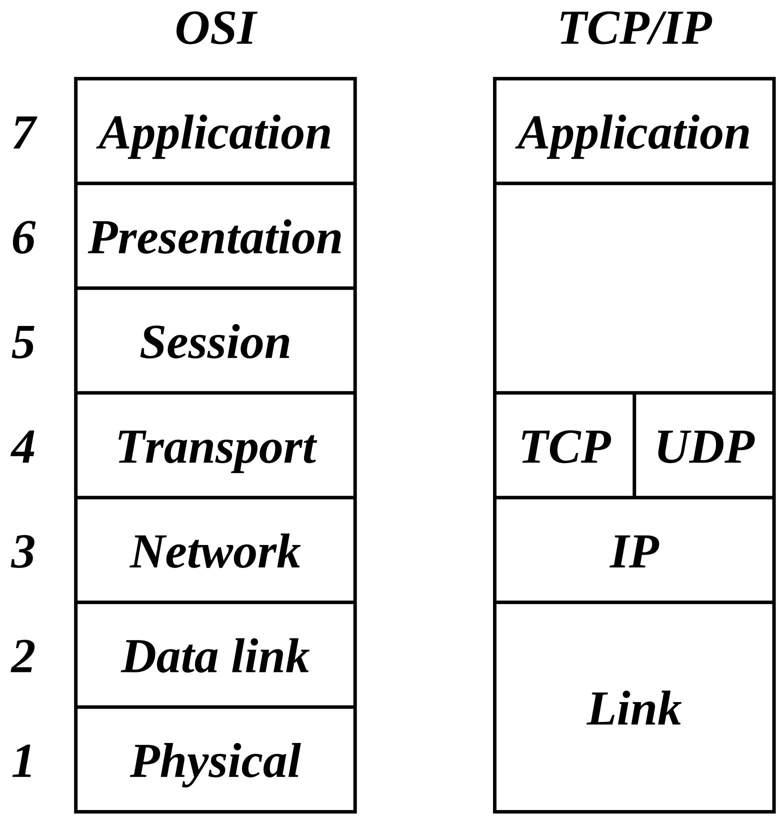

Abbildung 4.4: Schichtenmodell von OSI und TCP/IP

Wir haben also eine Schicht mit IP-Adressen und darunter eine Schicht, die von der verwendeten Netzwerk-Hardware abhängt. Aufbauend auf der Schicht mit IP-Adressen (IP-Protokoll) gibt es alternative Transport-Schichten, über die Pakete versendet werden können: Mittels UDP (*User Datagram Protocol*) können einzelne Pakete sehr effizient, aber unzuverlässig versendet werden, während TCP (*Transmission Control Protocol*) eine sichere Verbindung gewährleistet, die jedoch weniger effizient ist. Abbildung 4.4 zeigt auf der rechten Seite die vier Schichten, die beim Internet eine Rolle spielen: Anwendungen (wie etwa `telnet` und `daytime` im obigen Beispiel), Transport (UDP oder TCP), IP und die physische Ebene. Parallel dazu entstand 1983 das OSI-Referenz-Modell[3] der ISO (*International Standards Organization*), das eine etwas feinere Schichtung vorsieht. Die Präsentations- oder Sitzungs-Ebene fand jedoch nie ihren Weg in die Protokollhierarchie von TCP/IP.

4.3 Berkeley Sockets

Für TCP/IP gibt es zwei Schnittstellen, die beide zum POSIX-Standard gehören: Die Berkeley Sockets, die mit BSD 4.2 im Jahr 1983 eingeführt worden sind, und TLI (*Transport Layer Interface*), das auf Streams basiert und zuerst durch UNIX System V Release 3.0 im Jahr 1987 veröffentlicht wurde. Die Berkeley-Socket-Schnittstelle hat sich weitgehend durchgesetzt, da sie bei einigen UNIX-Systemen die einzige zur Verfügung stehende Schnittstelle ist.

Das Standardwerk über BSD 4.3 (*The Design and Implementation of the 4.3 BSD UNIX*

[3] OSI steht für *Open Systems Interconnection*.

Name	1	2	3	4	5	6
SOCK_STREAM	⋆	⋆	⋆		⋆	⋆
SOCK_DGRAM				⋆		
SOCK_SEQPACKET	⋆	⋆	⋆	⋆	⋆	⋆
SOCK_RDM	⋆	⋆	⋆	⋆		

Tabelle 4.1: Unterstützte semantische Kombinationen bei Berkeley Sockets

Operating System, Samuel J. Leffler et al, Addison Wesley, 1989, ISBN 0-201-06196-1) nennt folgende Ziele:

- **Transparenz**: Die Kommunikation zwischen zwei Prozessen soll nicht davon abhängen, ob sie auf dem gleichen Rechner laufen oder nicht.
- **Effizienz**: Zu Zeiten von BSD 4.2 (also 1983) war dies ein außerordentlich wichtiges Kriterium wegen der damals noch sehr geringen Rechenleistung. Aus diesem Grund werden insbesondere keine weiteren System-Prozesse zur Kommunikation eingesetzt, obwohl dies zu mehr Flexibilität und Modularität hätte führen können.
- **Kompatibilität**: Viele bestehende Applikationen und Bibliotheken wissen nichts von Netzwerken und sollen dennoch in einem verteilten Umfeld eingesetzt werden können. Dies wurde dadurch erreicht, daß nach einem erfolgten Verbindungsaufbau (der z.B. von einem anderen Prozeß durchgeführt werden kann) Ein- und Ausgabe in gewohnter Weise (wie bei Dateien, Pipelines oder Terminal-Verbindungen) erfolgen können.

Um den Anforderungen gerecht zu werden, wurden folgende Abstraktionen entwickelt:

- Die Abstraktion eines Kommunikationsbereiches macht es möglich, nicht nur TCP/IP zu unterstützen, sondern auch viele weitere Netzwerke (z.B. Appletalk, DECnet, IPX von Novell). Zu jedem Kommunikationsbereich gibt es unterschiedliche Namen (bzw. Adressen) für Kommunikationsendpunkte. Bei TCP/IP sind das die bekannten 32 Bit langen IP-Adressen (z.B. 134.60.166.2 für die Theseus) kombiniert mit der Port-Nummer des einzelnen Dienstes (z.B. 13 für `daytime`).
- Die Abstraktion eines Kommunikationsendpunktes (daher der Name "socket"), der mit der eines Dateideskriptors verbunden wird und über den eine bidirektionale Kommunikation möglich ist.
- Die Semantik einer Kommunikation.

Die Semantik einer Kommunikation umschließt bei jeder Verbindung eine Teilmenge der folgenden Punkte:

1. Daten werden in der Reihenfolge empfangen, in der sie abgeschickt worden sind.
2. Daten kommen nicht doppelt an.
3. Daten werden zuverlässig übermittelt.

4. Einzelne Pakete kommen in der originalen Form an (d.h. sie werden weder zerstückelt noch mit anderen Paketen kombiniert).

5. Nachrichten außerhalb des normalen Kommunikationsstromes (*out-of-band messages*) werden unterstützt.

6. Die Kommunikation erfolgt verbindungs-orientiert, womit die Notwendigkeit entfällt, sich bei jedem Paket identifizieren zu müssen.

Tabelle 4.1 zeigt die Varianten, die von der Berkeley-Socket-Schnittstelle unterstützt werden. Von diesen Varianten kommt `SOCK_STREAM` den Pipelines am nächsten, wenn davon abgesehen wird, daß die Verbindungen bei Pipelines nur unidirektional sind. `SOCK_STREAM` läßt sich ziemlich direkt auf TCP abbilden, während `SOCK_DGRAM` ziemlich genau UDP widerspiegelt. Die Varianten `SOCK_SEQPACKET` (TCP-basiert) und `SOCK_RDM` (UDP-basiert) fügen hier noch weitere Funktionalitäten hinzu. Allerdings fand `SOCK_RDM` nicht den Weg in den POSIX-Standard und wird auch von einigen Implementierungen nicht angeboten. Im weiteren Verlauf werden wir uns nur mit `SOCK_STREAM`-Sockets beschäftigen.

4.4 Aufbau einer Netzwerk-Verbindung

Bis zu einem gewissen Grad mag hier zur Einführung die Analogie mit unserem Telefonsystem hilfreich sein. Bevor Sie Telefonanrufe entgegennehmen oder selbst anrufen können, benötigen Sie einen Telefonanschluß. Dieser Anschluß wird mit dem Systemaufruf `socket()` erzeugt:

```
int sfd = socket(domain, type, protocol);
```

Bei `domain` wird hier normalerweise `PF_INET` angegeben, um IPv4 zu selektieren. `PF` steht dabei für *protocol family*. Bei `type` kann eine der unterstützten Semantiken ausgewählt werden, also beispielsweise `SOCK_STREAM`. Der dritte Parameter `protocol` erlaubt in einigen Fällen eine weitere Selektion. Normalerweise wird hier schlicht 0 angegeben.

Nachdem der Anschluß existiert, fehlt noch eine zugeordnete Telefonnummer. Um bei der Analogie zu bleiben, haben wir eine Vorwahl (IP-Adresse) und eine Durchwahl (Port-Nummer). Auf einem Rechner können mehrere IP-Adressen zur Verfügung stehen. Es ist dabei möglich, nur eine dieser IP-Adressen zu verwenden oder alle, die zur Verfügung stehen. Bei den Port-Nummern ist eine automatische Zuteilung durch das Betriebssystem möglich. Alternativ ist es auch möglich, sich selbst eine Port-Nummer auszuwählen. Diese darf aber noch nicht vergeben sein und muß bei nicht-privilegierten Prozessen eine Nummer jenseits des Bereiches der wohldefinierten Port-Nummern sein, also typischerweise mindestens 1024 betragen. Die Verknüpfung eines Anschlusses mit einer vollständigen Adresse erfolgt mit dem Systemaufruf `bind()`:

```
struct sockaddr_in address = {0};
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(port);
bind(sfd, (struct sockaddr *) &address, sizeof address);
```

Die Datenstruktur `struct sockaddr_in` repräsentiert Adressen für IPv4, die aus einer IP-Adresse und einer Port-Nummer bestehen. Das Feld `sin_family` legt den Adreßraum fest. Hier gibt es passend zur Protokollfamilie `PF_INET` nur `AF_INET` (`AF` steht hier für *address family*). Bei dem Feld `sin_addr.s_addr` läßt sich die IP-Adresse angeben. Mit `INADDR_ANY` übernehmen wir alle IP-Adressen, die zum eigenen Rechner gehören. Das Feld `sin_port` spezifiziert die Port-Nummer. Da Netzwerkadressen grundsätzlich nicht von der Byte-Anordnung eines Rechners abhängen dürfen, wird mit `htonl` (*host to network long*) der 32-Bit-Wert der IP-Adresse in die standardisierte Form konvertiert. Analog konvertiert `htons()` (*host to network short*) den 16-Bit-Wert `port` in die standardisierte Byte-Reihenfolge. Wenn die Port-Nummer vom Betriebssystem zugeteilt werden soll, kann bei `sin_port` auch einfach 0 angegeben werden. Der Datentyp `struct sockaddr_in` ist eine spezielle Variante des Datentyps `struct sockaddr`. Letzterer sieht nur ein Feld `sin_family` vor und ein generelles Datenfeld `sa_data`, das umfangreich genug ist, um alle unterstützten Adressen unterzubringen. Bei `bind()` wird der von `socket()` erhaltene Deskriptor angegeben (hier `sfd`), ein Zeiger, der auf eine Adresse vom Typ `struct sockaddr` verweist, und die tatsächliche Länge der Adresse, die normalerweise kürzer ist als die des Typs `struct sockaddr`. Schön sind diese Konstruktionen nicht, aber C bietet eben keine objekt-orientierten Konzepte, wenngleich die Berkeley-Socket-Schnittstelle sehr wohl objekt-orientiert ist.

Grundsätzlich kann auf `bind()` auch verzichtet werden. Wenn dies geschieht, entspricht dies der Angabe von `INADDR_ANY` bei dem Feld `sin_addr.s_addr` und einer 0 bei `sin_port`. Das bedeutet dann, daß die Adresse vollständig von dem Betriebssystem vergeben wird. Die auf diese Weise erhaltene Adresse läßt sich mit der Funktion `getsockname()` ermitteln, sobald eine Verbindung eröffnet wird oder eingehende Verbindungen angenommen werden können.

Damit eingehende Verbindungen (oder Anrufe in unserer Telefon-Analogie) entgegengenommen werden können, muß `listen()` aufgerufen werden:

```
listen(sfd, SOMAXCONN);
```

Wohlgemerkt, nach `listen()` kann der Anschluß “klingeln”, aber noch sind keine Vorbereitungen getroffen, das Klingeln zu hören oder den Hörer abzunehmen. Der zweite Parameter bei `listen()` gibt an, wieviele Kommunikationspartner es gleichzeitig klingeln lassen dürfen. `SOMAXCONN` ist hier das Maximum, das die jeweilige Implementierung erlaubt. Programmtext 4.13 zeigt, wie mit `socket()` und `listen()` ein Anschluß angelegt werden kann. Die zugeteilte Port-Nummer wird dann mit `getsockname()` ermittelt. Interessanterweise gibt es keine von POSIX unterstützte Methode, alle verfügbaren IP-Adressen zu ermitteln. Dies ist allerdings kein gravierendes Problem, da typischerweise `INADDR_ANY` oder eine explizit (z.B. über die Kommandozeile) angegebene IP-Adresse verwendet wird.

Die Entgegennahme eines Anrufes erfolgt mit `accept()`:

```
struct sockaddr client_addr;
socklen_t client_addr_len = sizeof client_addr;
int fd = accept(sfd, &client_addr, &client_addr_len);
```

Liegt noch kein Anruf vor, blockiert `accept()` bis zum nächsten Anruf. Wenn mit `accept()` ein Anruf eingeht, wird ein Dateideskriptor auf den bidirektionalen Verbindungskanal zurückgeliefert. Normalerweise speichert `accept()` die Adresse des Klienten beim angegebenen Zeiger ab. Wenn als Zeiger 0 angegeben wird, entfällt dies.

newsocket.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

void print_ip_addr(in_addr_t ipaddr) {
   if (ipaddr == INADDR_ANY) {
      printf("INADDR_ANY");
   } else {
      uint32_t addr = ntohl(ipaddr);
      printf("%d.%d.%d.%d",
         addr>>24, (addr>>16)&0xff,
         (addr>>8)&0xff, addr&0xff);
   }
}

int main() {
   int sfd = socket(PF_INET, SOCK_STREAM, 0);
   if (sfd < 0) exit(1);
   if (listen(sfd, SOMAXCONN) < 0) exit(2);
   struct sockaddr address;
   socklen_t len = sizeof address;
   if (getsockname(sfd, &address, &len) < 0) exit(3);
   struct sockaddr_in * inaddr = (struct sockaddr_in *) &address;
   printf("This is the address of my new socket:\n");
   printf("IP Address:  "); print_ip_addr(inaddr->sin_addr.s_addr);
   printf("\n");
   printf("Port Number: %d\n", (int) ntohs(inaddr->sin_port));
}
```

Programmtext 4.13: Einrichtung eines Anschlusses

Programmtext 4.14 zeigt einen einfachen Zeitdienst analog zu `daytime`, der die Port-Nummer 11011 verwendet. Zusätzlich kommt hier noch `setsockopt()` in den Zeilen 21 und 22 zum Einsatz, um die Option `SO_REUSEADDR` einzuschalten. Dies empfiehlt sich immer, wenn eine feste Port-Nummer verwendet wird. Fehlt diese Option, kann es passieren, daß bei einem Neustart des Dienstes die Port-Nummer nicht sofort wieder zur Verfügung steht, da noch alte Verbindungen nicht vollständig abgewickelt sind. In den Zeilen 31 bis 33 wird jeweils die aktuelle Zeitangabe in eine Zeichenkette konvertiert und mit Hilfe von `write()` an die zuvor geöffnete Verbindung verschickt. Mit `close()` wird die Verbindung dann auf der eigenen Seite aufgehängt.

Die Verbindungsaufnahme erfolgt mit `connect()`, wobei der bei `socket()` zurückgelieferte Deskriptor anzugeben ist und die Adresse des Kommunikationspartners:

```
connect(fd, (struct sockaddr *) &addr, sizeof addr);
```

timeserver.c

```
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

#define PORT 11011

int main () {
   struct sockaddr_in address = {0};
   address.sin_family = AF_INET;
   address.sin_addr.s_addr = htonl(INADDR_ANY);
   address.sin_port = htons(PORT);

   int sfd = socket(PF_INET, SOCK_STREAM, 0);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &address,
             sizeof address) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      perror("socket"); exit(1);
   }

   int fd;
   while ((fd = accept(sfd, 0, 0)) >= 0) {
      char timebuf[32]; time_t clock; time(&clock);
      ctime_r(&clock, timebuf);
      write(fd, timebuf, strlen(timebuf)); close(fd);
   }
}
```

Programmtext 4.14: Ein einfacher Zeitdienst timeclient.c

```
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define PORT 11011

int main (int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 1) {
      fprintf(stderr, "Usage: %s host\n", cmdname);
      exit(1);
   }
   char* hostname = *argv;
   struct hostent* hp;
   if ((hp = gethostbyname(hostname)) == 0) {
      fprintf(stderr, "unknown host: %s\n", hostname);
      exit(1);
   }
   char* hostaddr = hp->h_addr_list[0];
   struct sockaddr_in addr = {0};
   addr.sin_family = AF_INET;
   memmove((void *) &addr.sin_addr, (void *) hostaddr, hp->h_length);
   addr.sin_port = htons(PORT);

   int fd;
   if ((fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket"); exit(1);
   }
   if (connect(fd, (struct sockaddr *) &addr, sizeof addr) < 0) {
      perror("connect"); exit(1);
   }

   char buffer[BUFSIZ]; ssize_t nbytes;
   while((nbytes = read(fd, buffer, sizeof buffer)) > 0 &&
         write(1, buffer, nbytes) == nbytes);
}
```

Programmtext 4.15: Ein Klient des Zeitdienstes

Programmtext 4.15 zeigt einen Klienten, der den vorgestellten Zeitdienst kontaktiert, die Zeitangabe entgegennimmt und sie ausgibt. Der Klient erhält über die Kommandozeile den Namen des Rechners, auf dem der Zeitdienst zur Verfügung steht. Für die Abbildung eines Rechnernamens in eine IP-Adresse wird die Funktion `gethostbyname()` benötigt, die im Erfolgsfalle eine oder mehrere IP-Adressen liefert, unter denen sich der Rechner erreichen läßt. In Zeile 23 wird die erste IP-Adresse ausgewählt, die bereits in der richtigen Byte-Anordnung vorliegt. Mit Hilfe von `memmove()` wird sie zu `addr.sin_addr` in Zeile 26 kopiert. Danach kann in Zeile 33 mit der zuvor erzeugten Socket die gewünschte Verbindung mit Hilfe von `connect()` aufgebaut werden. Die `while`-Schleife in Zeile 38 gibt dann alles aus, was sich über die Netzwerkverbindung einlesen ließ.

4.5 Gepufferte Ein- und Ausgabe für Netzwerkverbindungen

Die Ein- und Ausgabe über Netzwerkverbindungen bringt in Vergleich zur Behandlungen von Dateien und interaktiven Benutzern einige Veränderungen mit sich. Wenn `SOCK_STREAM` zum Einsatz gelangt, so kommen die Daten zwar in der korrekten Reihenfolge an, jedoch nicht in der ursprünglichen Paketisierung. Als ursprüngliche Pakete werden hier die Daten betrachtet, die mit Hilfe eines einzigen Aufrufs von `write()` geschrieben werden:

```
const char greeting[] = "Hi, how are you?\r\n";
ssize_t nbytes = write(sfd, greeting, sizeof greeting);
```

Wenn beispielsweise bei einer Netzwerkverbindung immer vollständige Zeilen mit `write()` geschrieben werden, so ist es möglich, daß die korrespondierende `read()`-Operation nur einen Teil einer Zeile zurückliefert oder auch ein Fragment, das sich über mehr als eine Zeile erstreckt. Diese Problematik legt es nahe, nur zeichenweise einzulesen, wenn genau eine einzelne Zeile eingelesen werden soll:

```
char ch;
stralloc line = {0};
while (read(fd, &ch, sizeof ch) == 1 && ch != '\n') {
   stralloc_append(&line, &ch);
}
```

Diese Vorgehensweise ist jedoch außerordentlich ineffizient, weil Systemaufrufe wie `read()` zu einem Kontextwechsel zwischen dem aufrufenden Prozeß und dem Betriebssystem führen. Wenn ein Kontextwechsel für jedes einzulesende Byte initiiert wird, dann ist der betroffene Rechner mehr mit Kontextwechseln als mit sinnvollen Tätigkeiten beschäftigt. Wenn jedoch mit

```
char buf[512];
ssize_t nbytes = read(fd, buf, sizeof buf)
```

eingelesen wird, ist möglicherweise mehr als nur die gewünschte Zeile in `buf` zu finden. Oder auch nur ein Teil der Zeile.

Entsprechend ist eine gepufferte Eingabe notwendig, bei der die Eingabe-Operationen aus einem Puffer versorgt werden, der, wenn er leer wird, mit Hilfe einer `read()`-Operation aufzufüllen ist. Die Datenstruktur für einen Eingabe-Puffer benötigt entsprechend einen Dateideskriptor, einen Puffer und einen Positionszeiger innerhalb des Puffers:

```
typedef struct inbuf {
   int fd;
   stralloc buf;
   unsigned int pos;
} inbuf;
```

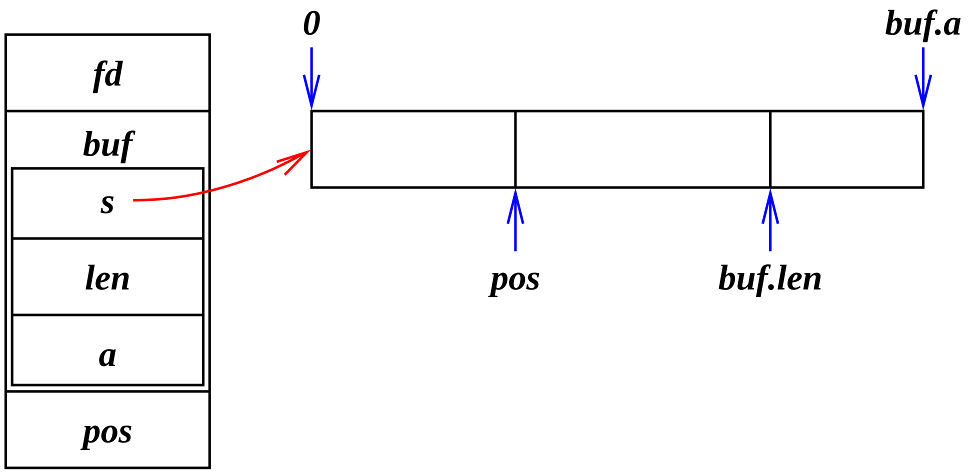

Abbildung 4.5: Struktur eines Eingabe-Puffers

Abbildung 4.5 dient der Illustrierung des Eingabe-Puffers. Als Puffer-Bereich dient `buf.s` mit einem zulässigen Indexbereich von 0 bis `buf.a-1`. Entsprechend dem letzten erfolgreichen Aufruf von `read()` ergibt sich ein Füllgrad des Puffers, der von `buf.len` repräsentiert wird. Der Positionszeiger `pos` begann unmittelbar nach der `read()`-Operation auf Position 0 und wandert bei jeder Einlese-Operation aus dem Puffer der Grenze von `buf.len` entgegen. Wird die Grenze erreicht, so ist die nächste `read()`-Operation fällig.

Programmtext 4.16 zeigt die entsprechende Schnittstelle. Die Funktion `inbuf_alloc()` dient dazu, die Größe des Puffers einzurichten, wobei eine sinnvolle Voreinstellung automatisch gewählt wird, wenn der Aufruf dieser Funktion unterbleibt. Als Einlese-Operationen vom Puffer dienen `inbuf_read()` und `inbuf_getchar()`, die sich in ihrer Aufrufsemantik an `read()` bzw. `fgetc()` orientieren. Ein zuviel gelesenes Zeichen kann mit `inbuf_back()` wieder zum erneuten Einlesen zur Verfügung gestellt werden. Mit `inbuf_free()` wird der Puffer dealloziert.

Programmtext 4.17 zeigt die zugehörige Implementierung, wobei hier insbesondere `inbuf_read()` interessant ist. In Zeile 14 wird untersucht, ob der Puffer bereits geleert ist, d.h. ob `pos` bereits `buf.len` erreicht hat. Falls ja, wird in den Zeilen 15 bis 24 der Puffer neu gefüllt. In Zeile 15 wird zunächst untersucht, ob der Puffer bereits allokiert worden ist. Falls nicht, wird dies mit der Standardgröße von 512 Bytes versucht. In Zeile 20 erfolgt die `read()`-Operation, bei der grundsätzlich versucht wird, den gesamten Puffer zu füllen. Allerdings ist damit zu rechnen, daß die Zahl der tatsächlich gelesenen Bytes `nbytes` niedriger als `buf.a` ist. Dies liegt daran, daß das Betriebssystem uns bereits vorhandene Daten sofort zur Verfügung stellt, inbuf.h

```
#ifndef INBUF_H
#define INBUF_H

#include <stralloc.h>
#include <unistd.h>

typedef struct inbuf {
   int fd;
   stralloc buf;
   unsigned int pos;
} inbuf;

/* set size of input buffer */
int inbuf_alloc(inbuf* ibuf, unsigned int size);

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size);

/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf);

/* move backward one position */
int inbuf_back(inbuf* ibuf);

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf);

#endif
```

Programmtext 4.16: Schnittstelle für den Eingabe-Puffer

selbst wenn es sich um eine geringere Quantität als angefordert handelt. Dies stellt sicher, daß effizientes Einlesen mit größen Puffergrößen ohne unnötiges Blockieren möglich ist. Die `read()`-Operation selbst ist in eine Schleife in den Zeilen 18 bis 21 eingebettet, die sicherstellt, daß es zu einem erneuten Versuch kommt, falls `read()` wegen einer Signalunterbrechung nicht erfolgreich sein konnte.

Sobald sichergestellt ist, daß mindestens ein Byte in dem Puffer verfügbar ist, wird `nbytes` in Zeile 26 auf die maximal mögliche Rückgabequantität gesetzt. Wurden weniger verlangt, so wird `nbytes` in Zeile 27 entsprechend zurückgesetzt. In Zeile 28 wird die zurückzuliefernde Quantität an Bytes aus dem Puffer in `buf` mit Hilfe von `memcpy()` kopiert. Der erste Parameter (`buf`) zeigt dabei auf das Ziel, der zweite Parameter (`buf.s + pos`) auf die Quelle und der dritte Parameter gibt die Zahl der zu kopierenden Bytes an (`nbytes`). Nach der Kopieraktion wird `pos` entsprechend aktualisiert.

Die Ausgabe sollte ebenfalls gepuffert erfolgen, um die Zahl der Systemaufrufe zu minimieren. Ein Positionszeiger ist nicht erforderlich, wenn Puffer grundsätzlich vollständig an `write()` übergeben werden. Hier ist das einzige Problem, daß die `write()`-Operation unter Umständen nicht den gesamten gewünschten Umfang akzeptiert und nur einen Teil der zu schreibenden inbuf.c

```
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include "inbuf.h"

/* set size of input buffer */
int inbuf_alloc(inbuf* ibuf, unsigned int size) {
   return stralloc_ready(&ibuf->buf, size);
}

/* works like read(2) but from ibuf */
ssize_t inbuf_read(inbuf* ibuf, void* buf, size_t size) {
   if (size == 0) return 0;
   if (ibuf->pos >= ibuf->buf.len) {
      if (ibuf->buf.a == 0 && !inbuf_alloc(ibuf, 512)) return -1;
      /* fill input buffer */
      ssize_t nbytes;
      do {
         errno = 0;
         nbytes = read(ibuf->fd, ibuf->buf.s, ibuf->buf.a);
      } while (nbytes < 0 && errno == EINTR);
      if (nbytes <= 0) return nbytes;
      ibuf->buf.len = nbytes;
      ibuf->pos = 0;
   }
   ssize_t nbytes = ibuf->buf.len - ibuf->pos;
   if (size < nbytes) nbytes = size;
   memcpy(buf, ibuf->buf.s + ibuf->pos, nbytes);
   ibuf->pos += nbytes;
   return nbytes;
}

/* works like fgetc but from ibuf */
int inbuf_getchar(inbuf* ibuf) {
   char ch;
   ssize_t nbytes = inbuf_read(ibuf, &ch, sizeof ch);
   if (nbytes <= 0) return -1;
   return ch;
}

/* move backward one position */
int inbuf_back(inbuf* ibuf) {
   if (ibuf->pos == 0) return 0;
   ibuf->pos--;
   return 1;
}

/* release storage associated with ibuf */
void inbuf_free(inbuf* ibuf) {
   stralloc_free(&ibuf->buf);
}
```

Programmtext 4.17: Implementierung des Eingabe-Puffers

Bytes akzeptiert und entsprechend eine geringere Quantität als Wert zurückgibt.

outbuf.h

```
#ifndef OUTBUF_H
#define OUTBUF_H

#include <stralloc.h>
#include <unistd.h>

typedef struct outbuf {
   int fd;
   stralloc buf;
} outbuf;

/* works like write(2) but to obuf */
ssize_t outbuf_write(outbuf* obuf, void* buf, size_t size);

/* works like fputc but to obuf */
int outbuf_putchar(outbuf* obuf, char ch);

/* write contents of obuf to the associated fd */
int outbuf_flush(outbuf* obuf);

/* release storage associated with obuf */
void outbuf_free(outbuf* obuf);

#endif
```

Programmtext 4.18: Schnittstelle für den Ausgabe-Puffer

Programmtext 4.18 zeigt die Schnittstelle für den Ausgabe-Puffer. Die Funktion `outbuf_write()` schreibt in den gegebenen Puffer und entspricht ansonsten dem Systemaufruf `write()`. Mit Hilfe von `outbuf_putchar()` können bequem einzelne Zeichen in den Puffer ausgegeben werden. Beide Schreiboperationen führen nur zur Verlängerung des Pufferinhalts, ohne daß dieser mit Hilfe einer `write()`-Operation geleert wird. Letzteres ist nur durch den Aufruf von `outbuf_flush()` möglich. Wenn der Puffer nicht mehr benötigt wird, kann er durch `outbuf_free()` freigegeben werden.

Die zugehörige Implementierung läßt sich dem Programmtext 4.19 entnehmen. In `outbuf_write()` wird in Zeile 9 darauf geachtet, daß der Puffer genügend Platz für den aufzunehmenden Inhalt aufweist, wonach in Zeile 10 der Kopiervorgang mit Hilfe von `memcpy()` durchgeführt werden kann. Danach muß nur noch `buf.len` in Zeile 11 angepaßt werden. In der Funktion `outbuf_flush()` gibt es zwei Schleifen. Die äußere Schleife in den Zeilen 24 bis 32 sorgt dafür, daß der gesamte Puffer-Inhalt geschrieben wird, da einzelne `write()`-Operationen die Freiheit haben, nur einen Teil umzusetzen. Mit Hilfe der Variablen `left` und `written` wird vermerkt, wieviel noch zu schreiben ist bzw. wieviel bereits geschrieben wurde. Die innere Schleife in den Zeilen 26 bis 29 wiederholt die `write()`-Operation im Falle von Unterbrechungen.

outbuf.c

```
#include <errno.h>
#include <stralloc.h>
#include <string.h>
#include "outbuf.h"

/* works like write(2) but to obuf */
ssize_t outbuf_write(outbuf* obuf, void* buf, size_t size) {
   if (size == 0) return 0;
   if (!stralloc_readyplus(&obuf->buf, size)) return -1;
   memcpy(obuf->buf.s + obuf->buf.len, buf, size);
   obuf->buf.len += size;
   return size;
}

/* works like fputc but to obuf */
int outbuf_putchar(outbuf* obuf, char ch) {
   if (outbuf_write(obuf, &ch, sizeof ch) <= 0) return -1;
   return ch;
}

/* write contents of obuf to the associated fd */
int outbuf_flush(outbuf* obuf) {
   ssize_t left = obuf->buf.len; ssize_t written = 0;
   while (left > 0) {
      ssize_t nbytes;
      do {
         errno = 0;
         nbytes = write(obuf->fd, obuf->buf.s + written, left);
      } while (nbytes < 0 && errno == EINTR);
      if (nbytes <= 0) return 0;
      left -= nbytes; written += nbytes;
   }
   obuf->buf.len = 0;
   return 1;
}

/* release storage associated with obuf */
void outbuf_free(outbuf* obuf) {
   stralloc_free(&obuf->buf);
}
```

Programmtext 4.19: Implementierung des Ausgabe-Puffers

4.6 Ein- und Ausgabe von Paketen für Netzwerkverbindungen

Zwischen Dienste-Anbietern und ihren Klienten auf dem Netzwerk besteht häufig ein ähnliches Verhältnis wie zwischen einer Shell und dem zugehörigen Benutzer: Der Klient gibt ein Kommando, das typischerweise mit dem Zeilentrenner CR LF[4] beendet wird, und der Dienste-Anbieter sendet darauf eine Antwort zurück,

- die zum Ausdruck bringt, ob das Kommando erfolgreich verlief oder fehlschlug, und
- einen Antworttext über eine oder mehrere Zeilen bringt.

Es gibt keine zwingende Notwendigkeit, bei einem Protokoll Zeilentrenner zu verwenden. Es gibt auch Alternativen wie die von Dan Bernstein vorgeschlagenen Net-Strings[5], jedoch erlaubt die Konvention mit dem Zeilentrenner CR LF die interaktive Benutzung eines Dienstes mit dem `telnet`-Kommando.

Bei jedem Protokoll lohnt es sich, eine gewisse Grundstruktur über die generelle Syntax von Kommandos und deren Antworten festzulegen. Hier ist ein Beispiel für eine (nicht sehr konstruktive) Sitzung mit einem Dienst, über den E-Mails zugestellt werden können. Zum Einsatz kommt hier das SMTP-Protokoll[6]:

```
doolin$ telnet mail.rz.uni-ulm.de smtp
Trying 134.60.246.1...
Connected to mail.rz.uni-ulm.de.
Escape character is '^]'.
220 mail.rz.uni-ulm.de ESMTP Sendmail 8.12.9/8.12.9
help
214-2.0.0 This is sendmail version 8.12.9
214-2.0.0 Topics:
214-2.0.0       HELO    EHLO    MAIL    RCPT    DATA
214-2.0.0       RSET    NOOP    QUIT    HELP    VRFY
214-2.0.0       EXPN    VERB    ETRN    DSN     AUTH
214-2.0.0       STARTTLS
214-2.0.0 For more info use "HELP <topic>".
214-2.0.0 To report bugs in the implementation send email to
214-2.0.0       sendmail-bugs@sendmail.org.
214-2.0.0 For local information send email to Postmaster at your site.
214 2.0.0 End of HELP info
huhu
500 5.5.1 Command unrecognized: "huhu"
helo doolin.andreas-borchert.de
250 mail.rz.uni-ulm.de Hello doolin.andreas-borchert.de
quit
221 2.0.0 mail.rz.uni-ulm.de closing connection
Connection to mail.rz.uni-ulm.de closed by foreign host.
doolin$
```

[4] *carriage return*, gefolgt von *line feed.*
[5] Siehe `http://cr.yp.to/proto/netstrings.txt`
[6] SMTP steht für *simple mail transfer protocol*, siehe RFC 2821

Beim SMTP-Protokoll erfolgt zunächst eine Begrüßung des Dienste-Anbieters. Die Begrüßung oder auch eine andere Antwort des Anbieters besteht aus einer dreistelligen Nummer, einem Leerzeichen oder einem Minus und beliebigem Text, der durch CR LF abgeschlossen wird. Die erste Ziffer der dreistelligen Nummer legt hier fest, ob ein Erfolg oder ein Problem vorliegt. Die beiden weiteren Ziffern werden zur feineren Unterscheidung der Rückmeldung verwendet. Eine führende 2 bedeutet Erfolg, eine 4 signalisiert ein temporäres Problem und eine 5 signalisiert einen permanenten Fehler.

In der Beispielsitzung ist das erste Kommando ein “help”, gefolgt von CR LF. Da die Antwort sich über mehrere Zeilen erstreckt, werden alle Zeilen, hinter der noch mindestens eine folgt, mit einem Minuszeichen hinter der dreistelligen Zahl gekennzeichnet. Danach wurde mit dem unbekannten Kommando “huhu” eine Fehlermeldung provoziert, die durch den Code 500 signalisiert wurde. Das SMTP-Protokoll erlaubt auch eine Fortsetzung des Dialogs nach Fehlern, so daß dann noch ein “helo”-Kommando akzeptiert wurde. Die Verbindung wurde mit dem “quit”-Befehl beendet.

mxprequest.h

```
#ifndef MXP_REQUEST_H
#define MXP_REQUEST_H

#include <stralloc.h>
#include "inbuf.h"
#include "outbuf.h"

typedef struct mxp_request {
   stralloc keyword;
   stralloc parameter;
} mxp_request;

/* read one request from the given input buffer */
int read_mxp_request(inbuf* ibuf, mxp_request* request);

/* write one request to the given outbuf buffer */
int write_mxp_request(outbuf* obuf, mxp_request* request);

/* release resources associated with request */
void free_mxp_request(mxp_request* request);

#endif
```

Programmtext 4.20: Schnittstelle für MXP-Anfrage-Pakete

Ungeachtet seines Namens ist das SMTP-Protokoll nicht mehr sehr einfach. Um ein überschaubareres Beispiel zu erhalten, wird im weiteren Text von folgenden Konventionen ausgegangen:

- Kommandos bestehen aus einem Kommandonamen (nur aus Kleinbuchstaben), einem Leerzeichen und genau einem Parameter. Abgeschlossen wird ein Kommando durch die Sequenz CR LF.

- Antworten beginnen mit “S”, “F” oder “C”, gefolgt von einem beliebigen Text, der durch mxpresponse.h

```
#ifndef MXP_RESPONSE_H
#define MXP_RESPONSE_H

#include "inbuf.h"
#include "outbuf.h"

typedef enum mxp_status {
   MXP_SUCCESS = 'S',
   MXP_FAILURE = 'F',
   MXP_CONTINUATION = 'C',
} mxp_status;

typedef struct mxp_response {
   mxp_status status;
   stralloc message;
} mxp_response;

/* write one (possibly partial) response to the given output buffer */
int write_mxp_response(outbuf* obuf, mxp_response* response);

/* read one (possibly partial) response from the given input buffer */
int read_mxp_response(inbuf* ibuf, mxp_response* response);

void free_mxp_response(mxp_response* response);

#endif
```

Programmtext 4.21: Schnittstelle für MXP-Antwort-Pakete

CR LF abgeschlossen wird. "S" deutet einen Erfolg an (*success*), "F" einen Fehlschlag (*failure*) und "C" gibt an, daß noch weitere Zeilen folgen.

Sobald dieser syntaktische Rahmen festliegt, ist es möglich, entsprechende Datenstrukturen für Anfragen und deren Antworten zu deklarieren und zugehörige Ein- und Ausgabefunktionen anzubieten wie Programmtext 4.20 und 4.21 zeigen.[7] Es ist dabei sinnvoll, jeweils die Ein- und Ausgabefunktionen zusammen zu belassen, da beide Funktionen jeweils genau zusammenpassen müssen.

Programmtext 4.22 demonstriert das Einlesen eines Anfrage-Pakets. Da eine Anfrage aus einem Kommandonamen, einem Leerzeichen, dem Parameter und den Zeilenterminator besteht, wurde das Einlesen entsprechend aufgeteilt. Die Funktionen `read_keyword()` und `read_parameter()` lesen jeweils ihren Teil in ein `stralloc`-Objekt ein und achten mit Hilfe von `inbuf_back()` darauf, daß das jeweilige terminierende Zeichen anschließend zum Einlesen wieder zur Verfügung steht. Die Funktion `expect_delimiter()` liest ein Zeichen ein und vergleicht es mit dem erwarteten Trennzeichen. Auf diese Weise lassen sich Leerzeichen und der Zeilentrenner überlesen. Übertriebene Toleranz bezüglich mehreren Leerzeichen oder verschiedenen Varianten bei Zeilentrennern ist hier fehl am Platze, da dies nur den Programmtext

[7] MXP steht hierbei für das *mutual exclusion protocol*, das im übernächsten Abschnitt vorgestellt wird.

mxprequest.c

```
#include "inbuf.h"
#include "outbuf.h"
#include "mxprequest.h"

static int read_keyword(inbuf* ibuf, stralloc* keyword) {
   int ch;
   if (!stralloc_copys(keyword, "")) return 0;
   while ((ch = inbuf_getchar(ibuf)) >= 0 &&
         ch >= 'a' && ch <= 'z') {
      if (!stralloc_readyplus(keyword, 1)) return 0;
      keyword->s[keyword->len++] = ch;
   }
   if (ch >= 0) inbuf_back(ibuf);
   return keyword->len > 0;
}

static int read_parameter(inbuf* ibuf, stralloc* parameter) {
   int ch;
   if (!stralloc_copys(parameter, "")) return 0;
   while ((ch = inbuf_getchar(ibuf)) >= 0 &&
         ch != '\r' && ch != '\n' && ch != '/' && ch != '\0') {
      if (!stralloc_readyplus(parameter, 1)) return 0;
      parameter->s[parameter->len++] = ch;
   }
   if (ch >= 0) inbuf_back(ibuf);
   return parameter->len > 0;
}

static int expect_delimiter(inbuf* ibuf, char delimiter) {
   int ch = inbuf_getchar(ibuf);
   if (ch < 0) return 0;
   return ch == delimiter;
}

/* read one request from the given input buffer */
int read_mxp_request(inbuf* ibuf, mxp_request* request) {
   return
      read_keyword(ibuf, &request->keyword) &&
      expect_delimiter(ibuf, ' ') &&
      read_parameter(ibuf, &request->parameter) &&
      expect_delimiter(ibuf, '\r') &&
      expect_delimiter(ibuf, '\n');
}
```

Programmtext 4.22: Das Einlesen eines MXP-Anfrage-Pakets mxprequest.c

```
/* write one request to the given outbuf buffer */
int write_mxp_request(outbuf* obuf, mxp_request* request) {
   return
      outbuf_write(obuf, request->keyword.s,
         request->keyword.len) == request->keyword.len &&
      outbuf_putchar(obuf, ' ') == ' ' &&
      outbuf_write(obuf, request->parameter.s,
         request->parameter.len) == request->parameter.len &&
      outbuf_putchar(obuf, '\r') == '\r' &&
      outbuf_putchar(obuf, '\n') == '\n';
}

/* release resources associated with request */
void free_mxp_request(mxp_request* request) {
   stralloc_free(&request->keyword);
   stralloc_free(&request->parameter);
}
```

Programmtext 4.23: Das Einlesen eines MXP-Anfrage-Pakets

verkompliziert ohne einen wahren Vorteil zu bringen, da primär nur Programme über dieses Protokoll kommunizieren. Programmtext 4.23 zeigt die zugehörige Schreib-Funktion und die Freigabe der mit der Datenstruktur verbundenen Speicher-Ressourcen. Zu beachten ist hier, daß `outbuf_flush()` nicht von `write_mxp_request()` aufgerufen wird. Die Kontrolle darüber, wann wirklich der Ausgabepuffer geleert wird, verbleibt so beim Kern des Programmtexts, der mit dem Protokoll umgeht.

Die Programmtexte 4.24 und 4.25 zeigen analog das Einlesen und Schreiben von Antwort-Paketen.

4.7 Parallele Sitzungen

Der zuvor mit dem Programmtext 4.14 (auf Seite 79) vorgestellte Zeitdienst wartete mit `accept()` auf den nächsten Aufruf und kümmerte sich dann ausschließlich um diesen. Da eines Sitzung des Zeitdienstes ohne Interaktionen auskommt und nur aus der Zeitansage besteht, führte dies nicht zu Problemen bei mehreren parallelen Anfragen. Entsprechend der bei `listen()` angegebenen Zahl (mit `SOMAXCONN` wird normalerweise das Maximum gewählt) ist es möglich, weitere Anrufe in eine Warteschlange einzureihen, bis die derzeitig laufende Sitzung beendet ist und `accept()` erneut aufgerufen wird. Dies ist jedoch bei längeren Sitzungen nicht mehr akzeptabel.

Es gibt drei Ansätze, um parallele Sitzungen zu ermöglichen:

- Für jede neue Sitzung wird mit Hilfe von `fork()` ein neuer Prozeß erzeugt, der sich um die Verbindung zu genau einem Klienten kümmert.
- Für jede neue Sitzung wird ein neuer Thread gestartet.

mxpresponse.c

```
#include "inbuf.h"
#include "mxpresponse.h"
#include "outbuf.h"

static int read_message(inbuf* ibuf, stralloc* message) {
   int ch;
   if (!stralloc_copys(message, "")) return 0;
   while ((ch = inbuf_getchar(ibuf)) >= 0 &&
         ch != '\r' && ch != '\n') {
      if (!stralloc_readyplus(message, 1)) return 0;
      message->s[message->len++] = ch;
   }
   if (ch >= 0) inbuf_back(ibuf);
   return 1;
}

static int expect_delimiter(inbuf* ibuf, char delimiter) {
   int ch = inbuf_getchar(ibuf);
   if (ch < 0) return 0;
   return ch == delimiter;
}

/* read one (possibly partial) response from the given input buffer */
int read_mxp_response(inbuf* ibuf, mxp_response* response) {
   int ch = inbuf_getchar(ibuf);
   switch (ch) {
      case MXP_SUCCESS:
      case MXP_FAILURE:
      case MXP_CONTINUATION:
         response->status = ch;
         break;
      default:
         return 0;
   }
   if (!read_message(ibuf, &response->message)) return 0;
   if (!expect_delimiter(ibuf, '\r')) return 0;
   if (!expect_delimiter(ibuf, '\n')) return 0;
   return 1;
}
```

Programmtext 4.24: Das Einlesen eines MXP-Antwort-Pakets mxpresponse.c

```
/* write one (possibly partial) response to the given output buffer */
int write_mxp_response(outbuf* obuf, mxp_response* response) {
   if (outbuf_putchar(obuf, response->status) < 0) return 0;
   if (response->message.len > 0) {
      ssize_t nbytes = outbuf_write(obuf, response->message.s,
         response->message.len);
      if (nbytes != response->message.len) return 0;
   }
   return
      outbuf_putchar(obuf, '\r') >= 0 &&
      outbuf_putchar(obuf, '\n') >= 0;
}

void free_mxp_response(mxp_response* response) {
   stralloc_free(&response->message);
}
```

Programmtext 4.25: Das Einlesen eines MXP-Antwort-Pakets

- Sämtliche Ein- und Ausgabe-Operationen werden asynchron abgewickelt mit Hilfe der `O_NONBLOCK`-Option und der Verwendung von `poll()` oder `select()`. Hierbei blockieren Ein- und Ausgabe-Operationen nicht mehr, sondern geben eine Fehlerindikation zurück, falls noch keine Eingabe vorliegt bzw. eine Ausgabe momentan noch nicht akzeptiert werden kann. Mit Hilfe von `poll()` oder `select()` ist es dann möglich, bei einer Vielzahl von Dateideskriptoren darauf zu waren, daß eine Lese- oder Schreib-Operation möglich ist.

Die erste Methode ist am einfachsten umzusetzen. Normalerweise wird nur dann eine der beiden anderen Varianten gewählt, wenn die parallel laufenden Sitzungen auf gemeinsame Datenstrukturen zugreifen möchten.

Die erste Variante ist so gängig, daß es sich lohnt, sie in verallgemeinerter Form zur Verfügung zu stellen. Von Dan J. Bernstein gibt es hierfür ein Werkzeug namens `tcpserver`[8], das die gesamte Sequenz von `socket()` bis `accept()` so verpackt, daß der eigentliche Dienst die gesamte Kommunikation über die Standard-Ein- und Ausgabe abwickeln kann. Auf diese Weise lassen sich sogar relativ einfach kleine Dienste auf Basis bestehender Kommandos einrichten. Hier ist ein Beispiel auf Basis des `factor`-Kommandos, das ganze Zahlen in ihre Primteiler zerlegt:

```
cordelia$ tcpserver 0 23456 /bin/sh -c \
> 'while read x; do echo $x | factor; done'
^Ccordelia$
```

Das Kommando `tcpserver` interpretiert die beiden ersten Argumente als Adresse und Portnummer, bei der der Dienst zu etablieren ist. Die 0 ist hier als Adreßangabe zulässig und entspricht dem `INADDR_ANY`. Für jede eingehende Verbindung wird ein neuer Prozeß erzeugt, bei dem Standard-Ein- und Ausgabe mit der Netzwerkverbindung verknüpft werden. Anschließend erfolgt ein `execvp()` zum angegebenen Kommando. Als Kommando kommt hier die Shell

[8] Siehe `http://cr.yp.to/ucspi-tcp/tcpserver.html`

zum Einsatz, bei der die Option "-c" es erlaubt, ein Shell-Kommando einschließlich der Verwendung von Kontrollstrukturen anzugeben. In diesem Beispiel handelt es sich dabei um eine `while`-Schleife, die jede eingelesene Zeile jeweils an das Kommando `factor` weiterleitet.

Prinzipiell wäre es natürlich auch möglich, `factor` direkt von `tcpserver` aufrufen zu lassen. Wegen der gepufferten Ausgabe von `factor` würden wir die Faktorisierung jedoch zu spät oder überhaupt nicht zu sehen bekommen. Wenn hingegen für jede Eingabezeile ein Kommandoaufruf erfolgt, ist damit auch ein implizites Leeren des Ausgabe-Puffers verbunden, sobald das Kommando beendet ist. Das Problem tritt bei einer normalen interaktiven Verwendung nicht auf, da die `stdio` implizit bei der Verwendung eines Terminals auf eine zeilenweise Pufferung umschaltet. Dies geschieht jedoch nicht bei Netzwerkverbindungen.

Hier ist eine Beispielsitzung des Faktorisierungsdienstes:

```
cordelia$ telnet cordelia 23456
Trying 134.60.66.8...
Connected to cordelia.
Escape character is '^]'.
8714
     2
     4357

9726
     2
     3
     1621

^]
telnet> quit
Connection to cordelia closed.
cordelia$
```

service.h

```
#ifndef SERVICE_H
#define SERVICE_H

#include <netinet/in.h>

typedef void (*session_handler)(int fd, int argc, char** argv);

/*
 * listen on the given port and invoke the handler for each
 * incoming connection
 */
void run_service(in_port_t port, session_handler handler,
   int argc, char** argv);

#endif
```

Programmtext 4.26: Schnittstelle für die Kontrollstruktur von Netzwerkdiensten service.c

```
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#include "service.h"

/*
 * listen on the given port and invoke the handler for each
 * incoming connection
 */
void run_service(in_port_t port, session_handler handler,
      int argc, char** argv) {
   struct sockaddr_in address = {0};
   address.sin_family = AF_INET;
   address.sin_addr.s_addr = htonl(INADDR_ANY);
   address.sin_port = htons(port);

   int sfd = socket(PF_INET, SOCK_STREAM, 0);
   int optval = 1;
   if (sfd < 0 ||
        setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR,
                   &optval, sizeof optval) < 0 ||
        bind(sfd, (struct sockaddr *) &address,
             sizeof address) < 0 ||
        listen(sfd, SOMAXCONN) < 0) {
      return;
   }

   /* our childs shall not become zombies */
   struct sigaction action = {0};
   action.sa_handler = SIG_IGN;
   action.sa_flags |= SA_NOCLDWAIT;
   if (sigaction(SIGCHLD, &action, 0) < 0) return;

   int fd;
   while ((fd = accept(sfd, 0, 0)) >= 0) {
      pid_t child = fork();
      if (child == 0) {
         handler(fd, argc, argv);
         exit(0);
      }
      close(fd);
   }
}
```

Programmtext 4.27: Implementierung für die Kontrollstruktur von Netzwerkdiensten

Die Kontrollstruktur, die `tcpserver` auf der Ebene der Kommandozeile liefert, läßt sich ebenso in genereller Form als Bibliotheksfunktion repräsentieren. Programmtext 4.26 zeigt eine entsprechende Schnittstelle und 4.27 die zugehörige Implementierung. Genauso wie `tcpserver` benötigt die Funktion `run_service()` die lokal zu verwendende Adresse. Der Einfachheit halber wird hier allerdings nur die Portnummer als Parameter übergeben und implizit `INADDR_ANY` verwendet. Für jeden eingehenden Anruf soll `run_service()` einen neuen Prozeß mit `fork()` erzeugen und in diesem die als Parameter übergebene Funktion `handler` aufrufen. Diese Funktion erhält als Parameter den Dateideskriptor der offenen Netzwerkverbindung und — analog zu `tcpserver` — die verbliebenen Kommandozeilenparameter. Genau wie `tcpserver` endet `run_service()` nur im Fehlerfalle und läuft ansonsten endlos.

In der zugehörigen Implementierung (siehe Programmtext 4.27) gleichen die Zeilen 18 bis 32 dem Aufbau des Anschlusses beim Zeitdienst (siehe Programmtext 4.14 auf Seite 79). Der entscheidende Kern der Kontrollstruktur ist in der Schleife in den Zeilen 41 bis 48 zu finden, bei der jeder erfolgreiche Aufruf von `accept()` zu einem neuen Prozeß führt, der `handler` aufruft. Der übergeordnete Prozeß schließt nur seinen Dateideskriptor zur Netzwerkverbindung und startet den nächsten Aufruf von `accept()`.

Im Rahmen dieses Verfahrens entstehen zahlreiche Kind-Prozesse, für die `wait()` nicht aufgerufen wird. Um nicht endlos Zombies zu akkumulieren, wird in den Zeilen 35 bis 38 das Signal `SIG_CHLD` ignoriert, wobei die Option `SA_NOCLDWAIT` gesetzt wird. Diese Option bewirkt, daß der Endstatus beendeter Kind-Prozesse sofort entsorgt wird, ohne daß diese zu Zombies werden oder später die Möglichkeit besteht, den Status noch mit `wait()` abzurufen. Alternativ wäre es natürlich auch möglich, `wait()` innerhalb eines Signalbehandlers aufzurufen oder regelmäßig in der `while`-Schleife `waitpid()` mit der `WNOHANG`-Option aufzurufen (siehe Programmtext 3.8 auf Seite 48).