Kapitel 3

Pipelines

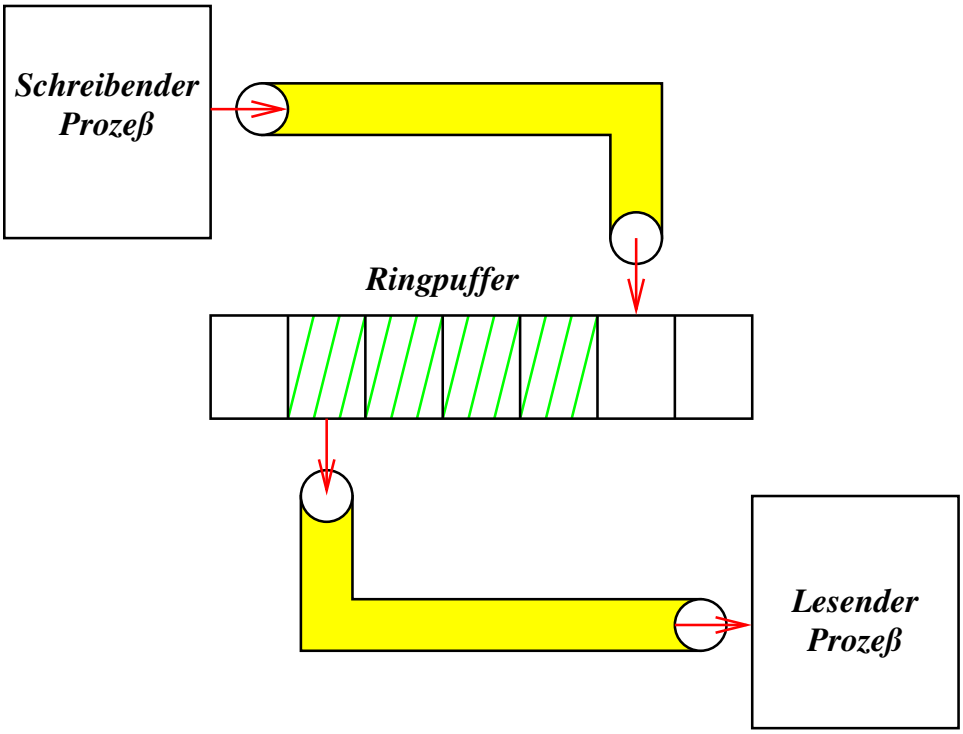

Abbildung 3.1: Aufbau einer Pipeline

Eine Pipeline ist ein unidirektionaler Kommunikationskanal (siehe Abbildung 3.1). Die schreibende und die lesende Seite einer Pipeline werden über verschiedene Dateiverbindungen angesprochen. Zur Pipeline gehört ein vom Betriebssystem verwalteter Ringpuffer, dessen Größenordnung bei mindestens 4 KiB liegt.[1] 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

[1] Bei Linux und Darwin habe ich 64 KiB gemessen, bei Solaris 20 KiB.

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.

Der folgende Programmtext 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.

Programm 3.1: Aufbau und Verwendung einer Pipeline (*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 }
```

Nach dem Aufruf von *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 geschrieben 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. Der folgende Programmtext 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.

Programm 3.2: Demonstration von *SIGPIPE* (*sigpipe.c*)

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  const int PIPE_READ = 0;
7  const int PIPE_WRITE = 1;
8
9  volatile sig_atomic_t sigpipe_received = 0;
10
11 void sigpipe_handler(int sig) {
12     sigpipe_received = 1;
13 }
14
15 int main() {
16     int pipefds[2];
17     if (pipe(pipefds) < 0) {
18         perror("pipe"); exit(1);
19     }
20     pid_t child = fork();
21     if (child < 0) {
22         perror("fork"); exit(1);
23     }
24     if (child == 0) {
25         close(pipefds[PIPE_WRITE]);
26         char buf[32];
27         ssize_t nbytes = read(pipefds[PIPE_READ],
28                 buf, sizeof buf);
29         if (nbytes > 0) {
30             if (write(1, buf, nbytes) < nbytes) exit(1);
31         }
32         exit(0);
33     }
34     close(pipefds[PIPE_READ]);
35     struct sigaction action = {0};
36     action.sa_handler = sigpipe_handler;
37     if (sigaction(SIGPIPE, &action, 0) < 0) {
38         perror("sigaction"); exit(1);
39     }
40     while (!sigpipe_received) {
```

```
41          const char message[] = "Hello!\n";
42          write(pipefds[PIPE_WRITE], message, sizeof message − 1);
43      }
44      close(pipefds[PIPE_WRITE]);
45      wait(0);
46  }
```

Wozu können Pipelines sinnvoll eingesetzt werden? Praktisch kann die Verwendung in einem Fork-and-Join-Szenario sein (siehe Abbildung 1.2), wenn es darum geht, Ergebnisse von den Unterprozessen abzuholen. Das hätte beispielsweise bei dem Programmtext 1.7 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. Das folgende Beispiel zeigt eine alternative Schnittstelle, die analog zu *execvp*() mit Argumentlisten arbeitet:

Programm 3.3: Schnittstelle für Pipelines zu Kommandos (*pconnect.h*)

```
1   /*
2    * Create and manage pipelines to given commands.
3    * In comparison to popen(), neither stdio nor the shell are used.
4    * afb 6/2003
5    */
6   #ifndef PCONNECT_H
7   #define PCONNECT_H
8
9   #include <unistd.h>
10
11  enum {PIPE_READ = 0, PIPE_WRITE = 1};
12
13  typedef struct pipe_end {
14      int fd;
15      pid_t pid;
16      int wstat;
17  } pipe_end;
18
19  /*
20   * create a pipeline to the given command;
21   * mode should be either PIPE_READ or PIPE_WRITE;
22   * return a filled pipe_end structure and 1 on success
23   * and 0 in case of failures
24   */
25  int pconnect(const char* path, char* const* argv,
26          int mode, pipe_end* pipe_con);
27
28  /*
29   * like pconnect() but connect fd to the standard input
30   * or output file descriptor that is not connected to the pipe
31   */
32  int pconnect2(const char* path, char* const* argv,
33          int mode, int fd, pipe_end* pipe_con);
```

```
34
35 /*
36  * close pipeline and wait for the forked-off process to exit;
37  * the wait status is returned in pipe->wstat;
38  * 1 is returned if successful, 0 otherwise
39  */
40 int phangup(pipe_end* pipe_end);
41
42 #endif
```

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. So könnte eine zugehörige Implementierung aussehen:

Programm 3.4: Pipelines zu Kommandos (*pconnect.c*)

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6  #include "pconnect.h"
7
8  /*
9   * create a pipeline to the given command;
10  * mode should be either PIPE_READ or PIPE_WRITE;
11  * return a filled pipe_end structure and 1 on success
12  * and 0 in case of failures
13  */
14 int pconnect(const char* path, char* const* argv,
15         int mode, pipe_end* pipe_con) {
16     return pconnect2(path, argv, mode, mode, pipe_con);
17 }
18
19 /*
20  * like pconnect() but connect fd to the standard input
21  * or output file descriptor that is not connected to the pipe
22  */
23 int pconnect2(const char* path, char* const* argv,
24         int mode, int fd, pipe_end* pipe_con) {
25     int pipefds[2];
26     if (pipe(pipefds) < 0) return 0;
27     int myside = mode; int otherside = 1 - mode;
28     fflush(0);
29     pid_t child = fork();
30     if (child < 0) {
31         close(pipefds[0]); close(pipefds[1]);
32         return 0;
33     }
34     if (child == 0) {
35         close(pipefds[myside]);
36         dup2(pipefds[otherside], otherside);
```

```
37          close(pipefds[otherside]);
38          if (fd != myside) {
39              dup2(fd, myside); close(fd);
40          }
41          execvp(path, argv); exit(255);
42      }
43      close(pipefds[otherside]);
44      int flags = fcntl(pipefds[myside], F_GETFD);
45      flags |= FD_CLOEXEC;
46      fcntl(pipefds[myside], F_SETFD, flags);
47      pipe_con->pid = child;
48      pipe_con->fd = pipefds[myside];
49      pipe_con->wstat = 0;
50      return 1;
51  }
52
53  /*
54   * close pipeline and wait for the forked-off process to exit;
55   * the wait status is returned in wstat (if non-null);
56   * 1 is returned if successful, 0 otherwise
57   */
58  int phangup(pipe_end* pipe) {
59      if (close(pipe->fd) < 0) return 0;
60      if (waitpid(pipe->pid, &pipe->wstat, 0) < 0) return 0;
61      return 1;
62  }
```

Nach der Erzeugung der Pipeline in Zeile 26 wird festgelegt, welche der beiden Dateideskriptoren welcher Seite gehört. In Zeile 28 werden alle Puffer der *stdio*-Bibliothek geleert. Der Kind-Prozess, der die gegenüberliegende Seite vertritt, schließt in Zeile 35 das unbenutzte Ende der Pipeline und sorgt in Zeile 39 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 39 geschlossen.

Der übergeordnete Prozess schließt nicht nur auf Zeile 43 das andere Ende, sondern setzt in Zeile 45 für das eigene Ende die Option *FD_CLOEXEC*, die bewirkt, dass der betreffende Dateideskriptor nicht über einen *exec*()-Systemaufruf hinweg vererbt, sondern geschlossen wird.[2] 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.[3] 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.

Programm 3.5: Verarbeitung der Ausgabe eines Kommandos (*rwhousers.c*)

```
1  #include <stralloc.h>
2  #include <string.h>
```

[2] Unter Linux gibt es auch den Systemaufruf *pipe2*, der die Angabe von Flags erlaubt, zu denen auch *O_CLOEXEC* gehört. Diese Erweiterung fand aber bislang nicht Eingang in den POSIX-Standard.

[3] Bei Linux wäre dies denkbar über die Registrierung von Funktionen mit Hilfe von *__register_atfork*, die unmittelbar nach dem Aufruf von *fork* wahlweise auf der Seite des erzeugenden und/oder des Unterprozesses aufgerufen werden können.

```
3  #include <unistd.h>
4  #include "pconnect.h"
5  #include "rwhousers.h"
6  #include "strlist.h"
7
8  const char rwho_path[] = "/usr/bin/rwho";
9
10 /*
11  * invoke rwho and get list of users that are currently logged in;
12  * return 1 in case of success, otherwise 0
13  */
14 int get_rwho_users(strlist* users) {
15     strlist argv = {0};
16     strlist_push(&argv, rwho_path);
17     strlist_push0(&argv);
18     pipe_end pipe;
19     int ok = pconnect(rwho_path, argv.list, PIPE_READ, &pipe);
20     strlist_free(&argv);
21     if (!ok) return 0;
22
23     stralloc rwho_output = {0};
24     ssize_t nbytes;
25     char buf[32];
26     while ((nbytes = read(pipe.fd, buf, sizeof buf)) > 0) {
27         stralloc_catb(&rwho_output, buf, nbytes);
28     }
29     phangup(&pipe);
30
31     strlist_clear(users);
32     char* user = rwho_output.s;
33     for (int i = 0; i < rwho_output.len; ++i) {
34         switch (rwho_output.s[i]) {
35             case '␣':
36                 if (user != 0) {
37                     rwho_output.s[i] = 0;
38                     strlist_push(users, strdup(user));
39                     user = 0;
40                 }
41                 break;
42             case '\n':
43                 user = rwho_output.s + i + 1;
44                 break;
45         }
46     }
47     stralloc_free(&rwho_output);
48     return 1;
49 }
```

Der vorstehende Programmtext demonstriert, wie *pconnect*() verwendet werden kann, um die Ausgabe eines Kommandos einzulesen. In diesem Beispiel geht es um die Ausgabe des *rwho*-Kommandos, das angibt, welche Nutzer sich im gleichen Netzwerk angemeldet haben:[4]

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

```
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 dem Programmtext *rwhousers.c* 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.

Der folgende Programmtext zeigt den umgekehrten Fall, bei dem die Ausgabe für ein Kommando generiert wird:

Programm 3.6: Generierung der Eingabe für ein Kommando (*sendmail.c*)

```
1  #include <stralloc.h>
2  #include <unistd.h>
3  #include "pconnect.h"
4  #include "sendmail.h"
5  #include "strlist.h"
6
7  const char sendmail_path[] = "/usr/lib/sendmail";
8  /*
9   * return a pipeline opened to /usr/lib/sendmail on the
10  * local system; return the opened pipeline and 1 in
11  * case of success; 0 in case of failures
12  */
13 int sendmail(char* recipient, char* subject, pipe_end* pipe_con) {
14     strlist argv = {0};
15     strlist_push(&argv, sendmail_path);
16     strlist_push(&argv, "-t");
17     strlist_push0(&argv);
18     int ok = pconnect(sendmail_path, argv.list, PIPE_WRITE, pipe_con);
19     strlist_free(&argv);
20     if (!ok) return 0;
21     stralloc header = {0};
22     stralloc_cats(&header, "To: ");
23     stralloc_cats(&header, recipient);
```

```
24      stralloc_cats(&header, "\n");
25      stralloc_cats(&header, "Subject:␣");
26      stralloc_cats(&header, subject);
27      stralloc_cats(&header, "\n\n");
28      ssize_t written = 0; ssize_t left = header.len;
29      while (left > 0) {
30          ssize_t nbytes = write(pipe_con->fd, header.s + written, left);
31          if (nbytes < 0) {
32              stralloc_free(&header);
33              phangup(pipe_con);
34              return 0;
35          }
36          written += nbytes; left -= nbytes;
37      }
38      stralloc_free(&header);
39      return 1;
40  }
```

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.

Programm 3.7: Anwendung mit zwei Pipeline-Verbindungen (*bigbrother.c*)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "pconnect.h"
4   #include "sendmail.h"
5   #include "strhash.h"
6   #include "strlist.h"
7   #include "rwhousers.h"
8
9   int main(int argc, char** argv) {
10      if (argc <= 2) {
11          fprintf(stderr, "Usage:␣%s␣email␣login...\n", argv[0]);
12          exit(1);
13      }
14      char* email = *++argv; --argc;
15      strhash friends = {0};
16      strhash_alloc(&friends, 4);
17      while (--argc > 0) {
18          if (!strhash_add(&friends, *++argv, 0)) exit(1);
19      }
```

```
20
21      strlist users = {0};
22      if (!get_rwho_users(&users)) exit(1);
23      strhash found = {0};
24      strhash_alloc(&found, 4);
25      for (int i = 0; i < users.len; ++i) {
26          if (strhash_exists(&found, users.list[i])) continue;
27          if (!strhash_exists(&friends, users.list[i])) continue;
28          if (!strhash_add(&found, users.list[i], 0)) exit(1);
29      }
30      if (strhash_length(&found) == 0) exit(0);
31
32      pipe_end pipe_con;
33      if (!sendmail(email, "Your␣Friends␣Are␣Online!", &pipe_con))
34          exit(1);
35      if (dup2(pipe_con.fd, 1) < 0) exit(1);
36      printf("Hi,␣");
37      if (strhash_length(&found) == 1) {
38          printf("one␣of␣your␣friends␣is");
39      } else {
40          printf("some␣of␣your␣friends␣are");
41      }
42      printf("␣online:\n");
43      strhash_start(&found);
44      char* key;
45      while (strhash_next(&found, &key)) {
46          printf("%s\n", key);
47      }
48      fclose(stdout);
49      if (!phangup(&pipe_con)) exit(1);
50  }
```

Der vorstehende Programmtext 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, um Duplikate zu eliminieren und das Abgleichen der Namenslisten zu erleichtern:

Programm 3.8: Schnittstelle einer Hash-Organisation für Zeichenketten (*strhash.h*)

```
1   /*
2    * Hashes for strings, i.e. keys and values are of type char*
3    * afb 6/2003
4    */
5
6   #ifndef STRHASH_H
7   #define STRHASH_H
8
9   typedef struct strhash_entry {
10      char* key;
11      char* value;
12      struct strhash_entry* next;
13  } strhash_entry;
14
15  typedef struct strhash {
```

```
16      unsigned int size, length;
17      strhash_entry** bucket; /* hash table */
18      unsigned int it_index;
19      strhash_entry* it_entry;
20  } strhash;
21
22  /* allocate a hash table with the given bucket size */
23  int strhash_alloc(strhash* hash, unsigned int size);
24
25  /* add tuple (key,value) to the hash, key must be unique */
26  int strhash_add(strhash* hash, char* key, char* value);
27
28  /* remove tuple with the given key from the hash */
29  int strhash_remove(strhash* hash, char* key);
30
31  /* return number of elements */
32  unsigned int strhash_length(strhash* hash);
33
34  /* check existance of a key */
35  int strhash_exists(strhash* hash, char* key);
36
37  /* lookup value by key */
38  int strhash_lookup(strhash* hash, char* key, char** value);
39
40  /* start iterator */
41  int strhash_start(strhash* hash);
42
43  /* fetch next key from iterator; returns 0 on end */
44  int strhash_next(strhash* hash, char** key);
45
46  /* free allocated memory */
47  int strhash_free(strhash* hash);
48
49  #endif
```

Die Zeile 34 bei *bigbrother.c* zeigt noch einen kleinen Trick. Wenn die Absicht besteht, die *stdio* zu verwenden, obwohl zunächst nur ein Dateideskriptor vorliegt, dann besteht die Möglichkeit, *fdopen*() zu 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 3.2 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.

Der folgende Programmtext 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

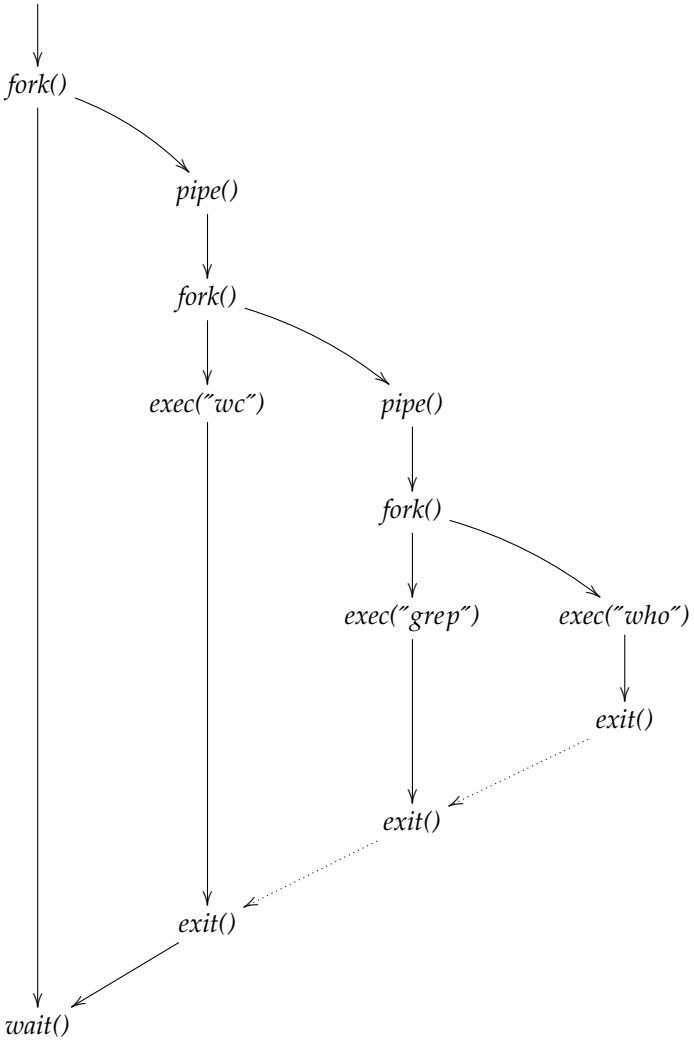

Abbildung 3.2: Prozessstruktur bei einer Kette von Pipelines

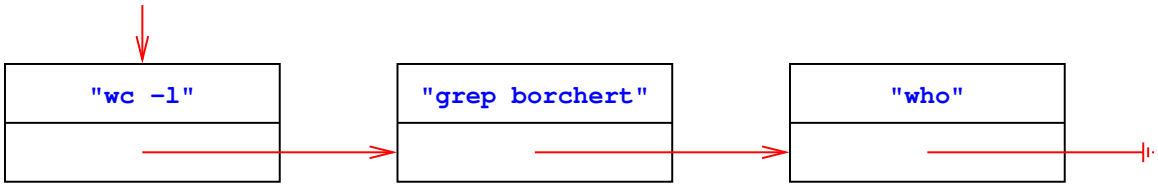

Abbildung 3.3: Datenstruktur für Kommandos einer Pipeline

Pipeline-Kette, die die Eingabe für dieses Kommando generiert.

Programm 3.9: Datenstruktur für Kommandos einer Pipeline (*pipecmd.h*)

```
1  #ifndef PIPECMD_H
2  #define PIPECMD_H
3
4  #include "strlist.h"
5
6  typedef struct pipecmd {
7      char* cmdname;
8      strlist* argv;
9      struct pipecmd* input;
10 } pipecmd;
11
12 /* construct pipeline chain out of tokens */
13 pipecmd* pipe_constructor(strlist* tokens);
14
15 /* free space allocated by pipe_constructor() */
16 void pipe_free(pipecmd* cmd);
17
18 #endif
```

Abbildung 3.3 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*().

Programm 3.10: Aufbau einer Pipeline-Liste ausgehend von der Wortliste (*pipecmd.c*)

```
1  #include <assert.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "pipecmd.h"
5  #include "strlist.h"
6
7  /* generate new pipeline command node */
8  static pipecmd* new_command(char* cmdname,
9          strlist* argv, pipecmd* input) {
10     assert(cmdname != 0);
11     pipecmd* cmd = (pipecmd*) malloc(sizeof(pipecmd));
12     if (cmd == 0) return 0;
13     cmd->cmdname = cmdname;
14     strlist_push0(argv);
15     cmd->argv = argv;
16     cmd->input = input;
17     return cmd;
18 }
19
20 /* construct pipeline chain out of tokens */
21 pipecmd* pipe_constructor(strlist* tokens) {
22     pipecmd* input = 0;
23     char* cmdname;
```

```
24      strlist* argv = 0;
25      for (int index = 0; index < tokens->len; ++index) {
26          if (strcmp(tokens->list[index], "|") == 0) {
27              if (argv == 0) return 0; /* syntax error */
28              input = new_command(cmdname, argv, input);
29              cmdname = 0; argv = 0;
30          } else {
31              if (argv == 0) {
32                  argv = (strlist*) malloc(sizeof(strlist));
33                  if (argv == 0) return 0;
34                  argv->list = 0; argv->len = 0; argv->allocated = 0;
35              }
36              if (cmdname == 0) cmdname = tokens->list[index];
37              if (!strlist_push(argv, tokens->list[index])) return 0;
38          }
39      }
40      if (cmdname != 0) input = new_command(cmdname, argv, input);
41      return input;
42  }
43
44  /* free space allocated by pipe_constructor() */
45  void pipe_free(pipecmd* cmd) {
46      if (cmd->input) pipe_free(cmd->input);
47      strlist_free(cmd->argv); free(cmd->argv);
48      free(cmd);
49  }
```

Vorstehender Programmtext demonstriert, wie eine Sequenz von Worten, die aus Kommandoargumenten und möglichen „|“-Zeichen besteht, 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.

Der folgende Programmtext zeigt, wie sich die Datenstruktur unmittelbar in die zugehörige Prozessstruktur umsetzen lässt.

Programm 3.11: Ausführung einer Pipeline-Kette (*execpipe.c*)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include "execpipe.h"
5   #include "pipecmd.h"
6
7   #define PIPE_READ 0
8   #define PIPE_WRITE 1
9
10  /*
11   * exec() to the given pipeline command;
12   * does not return in case of success
13   */
14  void exec_pipecmd(pipecmd* command) {
```

```
15      if (command->input != 0) {
16          int pipefds[2];
17          if (pipe(pipefds) < 0) {
18              perror("pipe"); return;
19          }
20          pid_t child = fork();
21          if (child < 0) {
22              perror("fork"); return;
23          }
24          if (child == 0) {
25              close(pipefds[PIPE_READ]);
26              if (dup2(pipefds[PIPE_WRITE], 1) < 0) {
27                  perror("dup2"); exit(255);
28              }
29              close(pipefds[PIPE_WRITE]);
30              exec_pipecmd(command->input);
31              exit(255);
32          }
33          close(pipefds[PIPE_WRITE]);
34          if (dup2(pipefds[PIPE_READ], 0) < 0) {
35              perror("dup2"); return;
36          }
37          close(pipefds[PIPE_READ]);
38      }
39      execvp(command->cmdname, command->argv->list);
40      perror(command->cmdname);
41  }
```

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.

Hierzu passt folgendes Hauptprogramm:

Programm 3.12: Hauptprogramm einer einfachen Shell, die Pipelines unterstützt (*pipesh.c*)

```
1   #include <stralloc.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <unistd.h>
5   #include <sys/wait.h>
6   #include "execpipe.h"
7   #include "pipecmd.h"
8   #include "sareadline.h"
9   #include "strlist.h"
10  #include "tokenizer.h"
11
12  int main() {
```

```
13      stralloc line = {0};
14      while (printf("%% "), readline(stdin, &line)) {
15          strlist tokens = {0};
16          stralloc_0(&line); /* required by tokenizer() */
17          if (!tokenizer(&line, &tokens)) break;
18          if (tokens.len == 0) continue;
19          pipecmd* command = pipe_constructor(&tokens);
20          if (command == 0) continue;
21          pid_t child = fork();
22          if (child < 0) {
23              perror("fork"); continue;
24          }
25          if (child == 0) {
26              exec_pipecmd(command);
27              exit(255);
28          }
29          pipe_free(command);
30
31          /* wait for termination of child */
32          int stat;
33          pid_t pid = wait(&stat);
34          if (pid == child) {
35              if (WIFEXITED(stat)) {
36                  int code = WEXITSTATUS(stat);
37                  if (code && code != 255) {
38                      printf("terminated with exit code %d\n", code);
39                  }
40              } else {
41                  printf("terminated abnormally\n");
42              }
43          } else {
44              perror("wait");
45          }
46      }
47  }
```