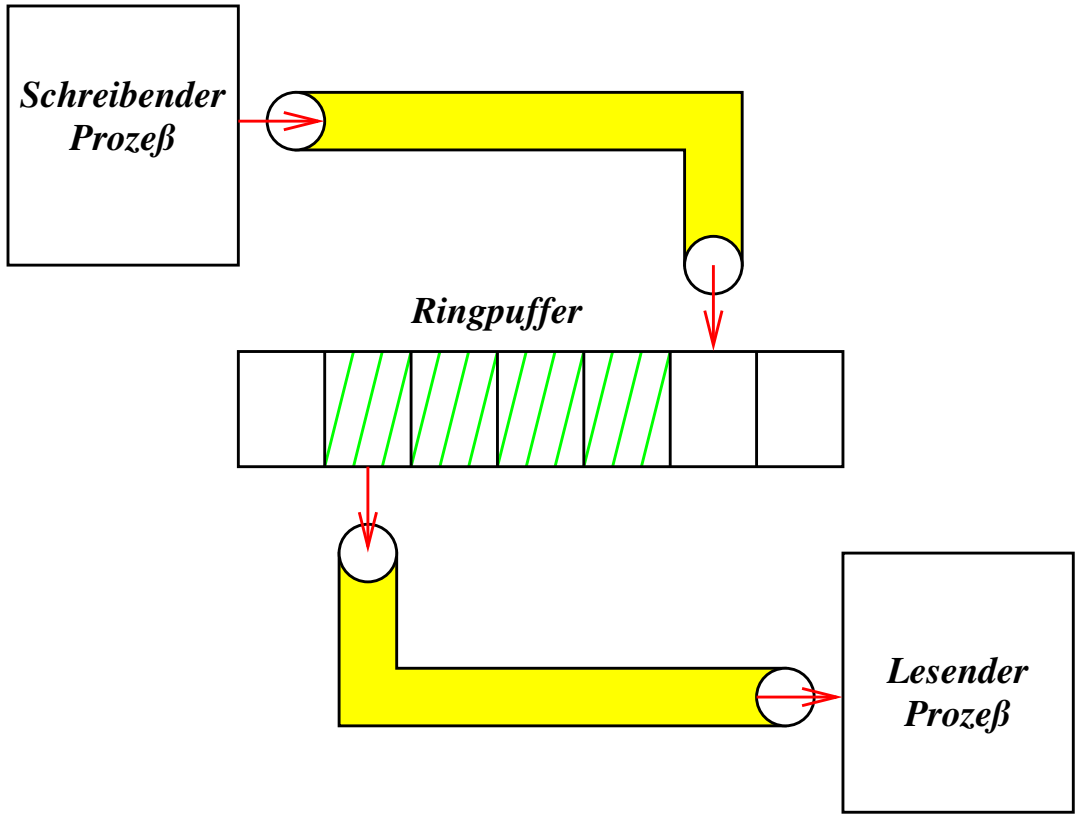
Pipelines

```
theseus$ ypcat passwd | cut -d: -f5 | cut -d:' ' -f1 |
> sort | uniq -c | sort -rn | head
     93 Michael
     92 Christian
     84 Florian
     83 Stefan
     81 Andreas
     69 Martin
     67 Thomas
     63 Matthias
     61 Daniel
     57 Markus
theseus$
```

- Welches sind die 10 häufigsten Vornamen unserer Benutzer?
- Dank Pipelines und dem Unix-Werkzeugkasten lässt sich diese Frage schnell beantworten.
- Die Notation und die zugehörige Art der Interprozesskommunikation wurde von einem der Mitautoren der ersten Unix-Shell Douglas McIlroy in den 70er-Jahren entwickelt und hat sehr zur Popularität von Unix beigetragen.

Aufbau einer Pipeline



- Pipelines sind unidirektionale Kommunikationskanäle. Die beiden Enden einer Pipeline werden über verschiedene Dateiverbindungen angesprochen.
- Sie werden innerhalb des Unix-Betriebssystems mit Hilfe eines festdimensionierten Ringpuffers implementiert.

Aufbau einer Pipeline

- Der festdimensionierte Ringpuffer hat typischerweise eine Größe von 4 Kilobyte.
- 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.

Konstruktion einer Pipeline

pipehello.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
enum {PIPE_READ = 0, PIPE_WRITE = 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;
      while ((nbytes = read(pipefds[PIPE_READ],
            buf, sizeof buf)) > 0) {
         if (write(1, buf, nbytes) < nbytes) exit(1);
      }
      exit(0);
   }
   close(pipefds[PIPE_READ]);
   const char message[] = "Hello!\n";
   write(pipefds[PIPE_WRITE], message, sizeof message - 1);
   close(pipefds[PIPE_WRITE]);
   wait(0);
}
```

Konstruktion einer Pipeline

pipehello.c

```
enum {PIPE_READ = 0, PIPE_WRITE = 1};
int main() {
   int pipefds[2];
   if (pipe(pipefds) < 0) {
      perror("pipe"); exit(1);
   }
   /* ... */
}
```

- Mit dem Systemaufruf *pipe* wird eine Pipeline erzeugt.
- Zurückgegeben wird dabei ein Array mit zwei Dateiverbindungen, die auf das lesende (Index 0) und das schreibende (Index 1) Ende verweisen.
- Eine Interprozesskommunikation auf Basis von *pipe* lässt sich nur über *fork* aufbauen, indem das entsprechende andere Ende der Pipeline an einen neu erzeugten Prozess vererbt wird.
- Solche Pipelines können also nur zwischen verwandten Prozessen existieren, bei denen ein gemeinsamer Urahn diese mit *pipe* angelegt hat.

Schreibende Seite einer Pipeline

pipehello.c

```
pid_t child = fork();
if (child < 0) {
   perror("fork"); exit(1);
}
if (child == 0) {
   /* ... */
}
close(pipefds[PIPE_READ]);
const char message[] = "Hello!\n";
write(pipefds[PIPE_WRITE], message, sizeof message - 1);
close(pipefds[PIPE_WRITE]);
wait(0);
```

- Der in eine Pipeline schreibende Prozess sollte das nicht genutzte Ende der Pipeline (hier das lesende) schließen. (Mehr dazu später.)
- Danach kann auf das schreibende Ende ganz normal mit *write* (oder auch darauf aufbauend der *stdio*) geschrieben werden.
- Sobald dies abgeschlossen ist, sollte das schreibende Ende geschlossen werden, damit ein Eingabe-Ende auf der anderen Seite der Pipeline erkannt werden kann.

Lesende Seite einer Pipeline

pipehello.c

```
if (child == 0) {
   close(pipefds[PIPE_WRITE]);
   char buf[32];
   ssize_t nbytes;
   while ((nbytes = read(pipefds[PIPE_READ],
         buf, sizeof buf)) > 0) {
      if (write(1, buf, nbytes) < nbytes) exit(1);
   }
   exit(0);
}
```

- Der von einer Pipeline lesende Prozess sollte das nicht genutzte Ende der Pipeline (hier das schreibende) schließen. (Mehr dazu später.)
- Danach kann auf das lesende Ende ganz normal mit *read* (oder auch darauf aufbauend der *stdio*) geschrieben werden.
- Die Schleife kopiert einfach alle Eingaben aus der Pipeline zur Dateiverbindung 1 (Standard-Ausgabe).
- Sobald alle schreibenden Enden geschlossen und der Ringpuffer geleert sind, wird ein Eingabe-Ende erkannt.

Gefahr von Hängern auf der lesenden Seite

- Nach *pipe* und *fork* haben zwei Prozesse jeweils beide Enden der Pipeline.
- Ein Eingabe-Ende auf der lesenden Seite wird genau dann (und nur dann!) erkannt, wenn **alle** schreibenden Enden geschlossen sind.
- Wenn also die lesende Seite es versäumt, die schreibende Seite zu schließen, wird sie kein Eingabe-Ende erkennen, wenn der andere Prozess seine schreibende Seite schließt.
- Stattdessen käme es zu einem endlosen Hänger.

Gefahr von Hängern auf der schreibenden Seite

- Genau dann (und nur dann!) wenn es kein Ende der Pipeline zum Lesen mehr gibt, führt das Schreiben auf das Ende zum Schreiben zur Zustellung des *SIGPIPE*-Signals bzw. dem Fehler *EPIPE*.
- Wenn die schreibende Seite es versäumt, ihr Ende zum Lesen zu schließen und der lesende Prozess aus irgendwelchen Gründen terminiert, ohne die Pipeline auslesen zu können, dann füllt sich zunächst der Ringpuffer und danach wird die schreibende Seite endlos blockiert.
- Entsprechend gäbe es wieder einen endlosen Hänger.
- Deswegen ist es von kritischer Bedeutung, dass die nicht benötigten Enden nach *fork* bei beiden Prozessen sofort geschlossen werden, um diese Probleme zu vermeiden.

Behandlung von *SIGPIPE*

sigpipe.c

```
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);
}
```

Behandlung von *SIGPIPE*

sigpipe.c

```
volatile sig_atomic_t sigpipe_received = 0;

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

- Der Signalbehandler für *SIGPIPE* setzt hier nur eine globale Variable, so dass entsprechend getestet werden kann.
- Alternativ könnte als Signalbehandler auch *SIG_IGN* eingetragen werden. Das würde keine Funktion benötigt werden und es müsste dann explizit jede *write*-Operation überprüft werden. Wenn niemand mehr das andere Ende lesen kann, würde *errno* auf *EPIPE* gesetzt werden.

Behandlung von *SIGPIPE*

sigpipe.c

```
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);
}
```

- Anders als zuvor ruft der neu erzeugte Prozess *read* nur ein einziges Mal auf und endet dann.
- Sobald sich dieser Prozess mit *exit* verabschiedet, bleibt kein lesendes Ende der Pipeline mehr offen, so dass damit dann die schreibende Seite das Signal *SIGPIPE* erhält, sobald sie in die Pipeline weiterhin schreibt.

Behandlung von *SIGPIPE*

sigpipe.c

```
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);
```

- Beim übergeordneten Prozess wird zunächst der Signalbehandler für *SIGPIPE* eingesetzt.
- Danach wird solange in die Pipeline geschrieben, bis das Signal endlich eintrifft.

Behandlung von *SIGPIPE*

sigpipe2.c

```
close(pipefds[PIPE_READ]);
sigignore(SIGPIPE);
ssize_t nbytes;
do {
   const char message[] = "Hello!\n";
   nbytes = write(pipefds[PIPE_WRITE],
      message, sizeof message - 1);
} while (nbytes > 0);
if (errno != EPIPE) perror("write");
close(pipefds[PIPE_WRITE]);
wait(0);
```

- Alternativ könnte *SIGPIPE* ignoriert werden.
- Dann ist die Überprüfung der *write*-Operationen zwingend notwendig.

Pipelines zu Kommandos

- Pipelines werden sehr gerne eingesetzt, um die Ausgabe eines Kommandos auszulesen und/oder die zugehörige Eingabe zu generieren.
- POSIX bietet für diese Funktionalität auf Basis der *stdio* die Funktionen *popen*() und *pclose*() an.
- Da *popen* in jedem Falle das erste Argument mitsamt Sonderzeichen an die Shell weiterreicht, ist dies nicht ohne Sicherheitsrisiken, die sich bei dieser Schnittstelle leider nicht vermeiden lassen.
- Das Sicherheitsrisiko ist beispielsweise gegeben, wenn Teile des ersten Arguments durch Benutzereingaben beeinflussbar sind.
- Deswegen ist von dieser Schnittstelle abzuraten.
- Besser ist es, direkt mit *pipe*, *fork* und *execvp* zu arbeiten, so dass keine Gefahr besteht, dass Kommandozeilenargumente als Programmieranweisung in der Shell mißverstanden werden.

Schnittstelle für Pipelines zu Kommandos

pconnect.h

```
#include <unistd.h>

enum {PIPE_READ = 0, 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);
```

Schnittstelle für Pipelines zu Kommandos

pconnect.h

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

- In der Verwaltungsstruktur wird von *pconnect* die Prozess-ID des neu erzeugten Prozesses und der Dateideskriptor zur Pipeline notiert.
- Wenn *phangup* aufgerufen wird, kann auf das Ende dieser Prozess-ID mit *waitpid* gewartet werden.
- Der zurückgelieferte Status wird dann in *wstat* abgelegt.

Schnittstelle für Pipelines zu Kommandos

pconnect.c

```
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_GETFD);
   flags |= FD_CLOEXEC;
   fcntl(pipefds[myside], F_SETFD, flags);
   pipe_con->pid = child;
   pipe_con->fd = pipefds[myside];
   pipe_con->wstat = 0;
   return 1;
}
```

Schnittstelle für Pipelines zu Kommandos

pconnect.c

```
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;
   }
   /* ... */
}
```

- Der Index *myside* wird auf zu benutzende Ende des übergeordneten Prozesses gesetzt, *otherside* auf das Ende des neu erzeugten Prozesses.
- Mit *fflush*(0) werden alle Puffer der *stdio* geleert, damit eine Duplizierung von Pufferinhalten durch *fork* vermieden wird.

Schnittstelle für Pipelines zu Kommandos

pconnect.c

```
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_GETFD);
flags |= FD_CLOEXEC;
fcntl(pipefds[myside], F_SETFD, flags);
pipe_con->pid = child;
pipe_con->fd = pipefds[myside];
pipe_con->wstat = 0;
return 1;
```

- Beim Kindprozess wird zunächst das nicht benötigte Ende der Pipeline geschlossen. Dann wird mit *dup2* das verbliebene Ende als Standardeingabe bzw. -ausgabe zur Verfügung gestellt. Nach dem *dup2* kann die dann überflüssig gewordene Dateiverbindung geschlossen werden.
- Die Option *FD_CLOEXEC* sorgt dafür, dass diese Dateiverbindung automatisch beim Aufruf einer der *exec*-Varianten geschlossen wird.

Schnittstelle für Pipelines zu Kommandos

pconnect.c

```
int phangup(pipe_end* pipe) {
   if (close(pipe->fd) < 0) return 0;
   if (waitpid(pipe->pid, &pipe->wstat, 0) < 0) return 0;
   return 1;
}
```

- *phangup* schließt die Verbindung zur Pipeline und wartet darauf, dass der entsprechende Kindprozess terminiert.

Verarbeitung der Ausgabe eines Kommandos

rwhousers.c

```
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);

   /* ... */
}
```

Verarbeitung der Ausgabe eines Kommandos

rwhousers.c

```
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;
```

- Mit der bereits vorgestellten *strlist*-Datenstruktur wird hier eine Kommandozeile zusammengestellt, die von *pconnect* akzeptiert wird. In diesem Beispiel ist sie besonders einfach, weil sie nur aus dem Namen des aufzurufenden Programms */usr/bin/rwho* besteht.
- Aus Sicherheitsgründen werden in so einem Kontext immer gerne absolute Pfade bei Kommandonamen angegeben, damit eine Manipulation durch das Setzen der Umgebungsvariable *PATH* ausgeschlossen bleibt.

Verarbeitung der Ausgabe eines Kommandos

rwhousers.c

```
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);
```

- In dieser Schleife wird die gesamte Ausgabe des aufgerufenen Kommandos eingelesen und in dem *stralloc*-Objekt *rwho_output* abgelegt.
- In der Praxis sind größere Puffergrößen üblich. Im Falle von Pipelines ist es sinnvoll, die Größe des Ringpuffers zu nehmen, falls diese bekannt ist.

Verarbeitung der Ausgabe eines Kommandos

rwhousers.c

```
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;
```

Generierung der Eingabe für ein Kommando

sendmail.c

```
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;
}
```

Generierung der Eingabe für ein Kommando

sendmail.c

```
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;
```

- Hier wird */usr/lib/sendmail* (unter Linux bei */usr/bin/sendmail* zu finden) aufgerufen mit der Option „-t“. Diese besagt, dass die Liste der Empfänger der E-Mail (beim „To“-Header) zu entnehmen ist.

Generierung der Eingabe für ein Kommando

sendmail.c

```
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;
}
```

- Hier wird zunächst der Kopf der E-Mail generiert und dann mit Hilfe einer Schleife geschrieben, da nicht garantiert ist, dass eine *write*-Operation alles schafft.

Arbeitet gerade einer meiner Freunde?

- Ziel einer kleinen Anwendung ist es, festzustellen, ob einer der Freunde, die alle auf der Kommandozeile aufzuzählen sind, gerade angemeldet ist. (Dies erfolgt durch die Auswertung der Ausgabe von *rwho*.)
- Wenn einer oder mehrere Freunde gefunden wurden, dann wird diese freudige Nachricht per E-Mail versandt.
- Um den Abgleich effizient durchführen zu können, wird eine Hash-Tabelle verwendet, in der die Freunde alle eingetragen werden.

Schnittstelle für eine Hash-Tabelle für Zeichenketten

strhash.c

```
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);
```

Big Brother is Watching You

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);
   }

   /* ... */
}
```

- Alle genannten Freunde werden in die Tabelle *friends* eingefügt.

Big Brother is Watching You

bigbrother.c

```
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);
```

- In der Tabelle *found* werden alle Benutzer notiert, die *rwho* zurücklieferte und die gleichzeitig in der Tabelle *friends* enthalten sind.
- Wenn keine der Freunde gefunden wird, terminiert das Programm danach schlicht mit einem Exit-Code von 0.

Big Brother is Watching You

bigbrother.c

```
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);
```