ulm university universität
uulm

Vorlesungsbegleiter zu Systemnahe Software II SS 2017

Andreas F. Borchert

Fakultät für Mathematik und Wirtschaftswissenschaften
Institut für Numerische Mathematik

Hinweise:

- Auf eine detaillierte Unterscheidung zwischen *BSD-Unix*, *System-V-Unix* oder *Linux* wird hier verzichtet. Stattdessen dient der IEEE Standard 1003.1 (POSIX) weitgehend als Grundlage.
- Die enthaltenen Beispiel-Programme wurden zum großen Teil unter Linux entwickelt und sind weitestgehend unter Solaris getestet (für konstruktive Hinweise sind die Autoren dankbar).
- Die Beispiele sollen jeweils gewisse Aspekte verdeutlichen und erheben nicht den Anspruch von Robustheit und Zuverlässigkeit. Man kann alles anders und besser machen.
- Details zu den behandelten bzw. verwendeten Systemaufrufen sollten jeweils im **Manual** bzw. den entsprechenden Header-Files nachgelesen werden.
- Die Sprache C dient in erster Linie als *Werkzeug* zur Darstellung systemnaher Konzepte.
- Einige der Beispiele verwenden die Vorlesungsbibliothek (zu erkennen an **#include** <afblib/...>), die unter
  `http://www.mathematik.uni-ulm.de/sai/ss17/soft2/afblib/`
  zur Verfügung steht.

Inhaltsverzeichnis

Kapitel 1

Prozesse unter UNIX

1.1 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.

Besser ausgestattete 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.

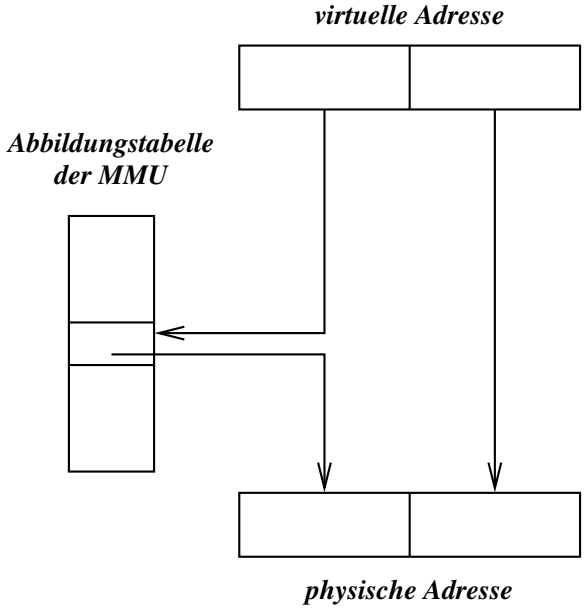

Abbildung 1.1: Virtuelle und physische Adressen

Abbildung 1.1 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 (auch Kacheln genannt, englisch: *page*). 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 Techniken sind auf dieser Grundlage 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 (*copy on write*), 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.

1.2 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 für jeden Thread einschließlich der Stackverwaltung (Stack-Zeiger, Frame-Zeiger) und dem PC (*program counter* oder *instruction pointer*, verweist auf die nächste auszuführende Instruktion) und
- weitere Statusinformationen, die durch das 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.

1.3 Die Prozess-ID

Jeder Prozess hat unter UNIX eine gleichbeibende identifizierende positive ganze Zahl, die mit *getpid*() abgefragt werden kann:

Programm 1.1: Ausgabe der eigenen Prozess-ID (*printpid.c*)

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5     printf("%d\n", (int) getpid());
6 }
```

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. Einige niedrige Prozess-IDs werden für System-Prozesse reserviert. Das gilt insbesondere für den Prozess mit der Prozess-ID 1, der für den *init*-Prozess reserviert ist.

1.4 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) (ab C99). 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 der ISO-Standards 9899-1999 und 9899-2011.

1.5 Das Erzeugen neuer Prozesse

Neue Prozesse können nur durch das Klonen eines bestehenden Prozesses 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.
- Der neue Prozess beginnt mit nur einem Thread. Das gilt auch dann wenn der erzeugende Prozess mehrere Threads hatte. (Die Kombination von *fork* und *multi threading* ist jedoch nicht unproblematisch.)

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 folgende Programmtext demonstriert bei der Ausführung, dass das zweite *printf* doppelt ausgeführt wird:

Programm 1.2: Aus einem Prozess werden zwei (*clones.c*)

```
1 #include <stdio.h>
2 #include <unistd.h>
3 int main() {
4     printf("I␣am␣feeling␣lonely!\n");
5     fork();
6     printf("Hey,␣I␣am␣cloned!\n");
7 }
```

```
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 folgender Programmtext zeigt:

Programm 1.3: Prozessduplizierung mit vorheriger Pufferleerung (*clones2.c*)

```
1  #include <stdio.h>
2  #include <unistd.h>
3  int main() {
4      printf("I␣am␣feeling␣lonely!\n"); fflush(stdout);
5      fork();
6      printf("Hey,␣I␣am␣cloned!\n");
7  }
```

Es bleibt die Frage, ob die beiden Prozesse in Abhängigkeit davon, ob sie der alte oder der neue Prozess sind, unterschiedliche Dinge tun können. Folgendes Beispiel zeigt, wie dies mit Hilfe von *getpid*() geschehen kann:

Programm 1.4: Unterscheidung von Vorfahre und Nachfahre (*clones3.c*)

```
1  #include <stdio.h>
2  #include <unistd.h>
3  int main() {
4      pid_t parent;
5
6      printf("I␣am␣feeling␣lonely!\n"); fflush(stdout);
7      parent = getpid();
8      fork();
9      if (getpid() == parent) {
10         printf("I␣am␣the␣parent␣process!\n");
11     } else {
12         printf("I␣am␣the␣child␣process!\n");
13     }
14 }
```

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. Folgende Variante demonstriert nicht nur die Nutzung des Rückgabewertes, sondern auch die Vermeidung eines gemeinsamen Pfades nach dem *fork*() durch ein explizites *exit*().

Programm 1.5: Der Rückgabewert von *fork*() (*fork.c*)

```
1  #include <stdio.h>
2  #include <unistd.h>
3  int main() {
4      pid_t pid;
5
6      pid = fork();
7      if (pid == −1) {
8          perror("unable␣to␣fork"); exit(1);
9      }
10     if (pid == 0) {
11         /* child process */
12         printf("I␣am␣the␣child␣process:␣%d.\n", (int) getpid());
13         exit(0);
14     }
15     /* parent process */
16     printf("The␣pid␣of␣my␣child␣process␣is␣%d.\n", (int) pid);
17 }
```

1.6 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.

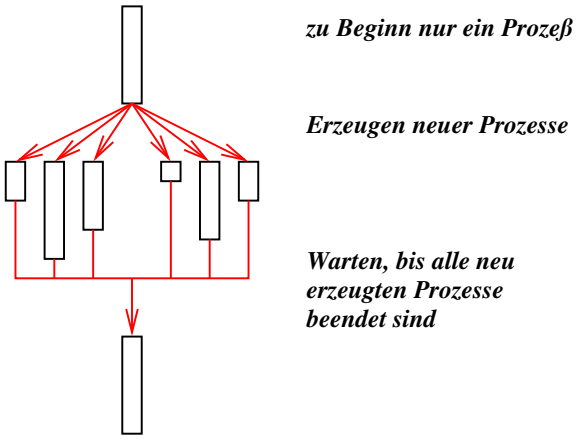

Abbildung 1.2: „Fork and Join“

Dies ist insbesondere sinnvoll, 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 1.2).

Der Systemaufruf, der es erlaubt, auf das Ende eines Prozesses zu warten, nennt sich *wait*(). Dabei ist zu beachten, dass nur das Warten auf unmittelbare 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.

Der folgende Programmtext demonstriert *wait*() an einem einfachen Beispiel:

Programm 1.6: Warten auf die Terminierung (*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  }
```

In den Zeilen 16 bis 18 gibt der neu erzeugte Prozess 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. (Prinzipiell kann ein Prozess auch gestoppt werden. Gestoppte Prozesse sind ebenfalls bei *wait* zu sehen, aber das beschränkt sich typischerweise auf Szenarien im Kontext eines Debuggers.) Der in *stat* abgelegte Status des Unterprozesses besteht aus mehreren Komponenten, die angeben,

- wie ein Prozess sein Leben beendete (durch *exit*() oder durch ein Signal (bei einem Crash oder Verwendung von *kill*()) oder ob der Prozess nur gestoppt wurde,
- welcher Wert bei *exit*() angegeben wurde, falls *exit*() benutzt wurde und
- welches Signal das Leben des Prozesses terminierte bzw. stoppte, 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, das den Exit-Wert zur nicht-trivialen Informationsübermittlung nutzt, ist im folgenden Programmtext zu finden:

Programm 1.7: Rekursion mit Unterprozessen (*forkingqueens.c*)

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/wait.h>
4   #include <unistd.h>
5   #define INSET(member,set) ((1<<(member))&set)
6   #define INCL(set,member) ((set) |= 1<<(member))
7   int main() {
8       const int size = 6; /* square size of the board */
9       struct { int row, col; } pos[size]; /* queen positions */
10      /* a queen on (row, col) threatens a row, a column,
11          and 2 diagonals;
12          rows and columns are characterized by their number (0..n−1),
13          the diagonals by row−col+n−1 and row+col,
```

```
14          (n is a shorthand for the square size of the board)
15      */
16      int rows = 0, cols = 0; /* bitmaps of [0..n−1] */
17      int diags1 = 0; /* bitmap of [0..2*(n−1)] used for row−col+n−1 */
18      int diags2 = 0; /* bitmap of [0..2*(n−1)] used for row+col */
19      int row = 0; int col;
20
21      setnextqueen: for (col = 0; col < size; ++col) {
22          if (INSET(row, rows)) continue;
23          if (INSET(col, cols)) continue;
24          if (INSET(row − col + size − 1, diags1)) continue;
25          if (INSET(row + col, diags2)) continue;
26          int child = fork();
27          if (child == −1) {
28              perror("fork"); exit(255);
29          }
30          if (child == 0) {
31              INCL(rows, row); INCL(cols, col);
32              INCL(diags1, row − col + size − 1);
33              INCL(diags2, row + col);
34              pos[row].row = row; pos[row].col = col;
35              ++row; /* set next queen in next row */
36              if (row == size) exit(1); /* solution found */
37              goto setnextqueen;
38          }
39      }
40
41      /* count the results of all children */
42      int stat; pid_t child; int nofsolutions = 0;
43      while ((child = wait(&stat)) > 0) {
44          if (!WIFEXITED(stat)) continue;
45          int exitval = WEXITSTATUS(stat);
46          if (exitval == 255) exit(255);
47          nofsolutions += exitval;
48      }
49      exit(nofsolutions);
50  }
```

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 mit dem Resultat einer wilden Textmischung. 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 Kopien der Variablen 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 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 ist es dann nicht sinnvoll, mehr Prozesse zu erzeugen als tatsächlich Prozessoren bzw. Prozessorkerne vorhanden sind. Ein pragmatischer Ansatz könnte darin bestehen, 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*().

1.7 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 aufbewahrt wird, aber die nicht mehr benötigten Ressourcen freigegeben werden. Prozesse, die sich in diesem Stadium befinden, werden als Zombies bezeichnet.

Das folgende Beispiel demonstriert die Erzeugung eines Zombies:

Programm 1.8: Erzeugung eines Zombies (*genzombie.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main() {
6      pid_t child = fork();
7      if (child == -1) {
8          perror("fork"); exit(1);
9      }
10     if (child == 0) exit(0);
11     printf("%d\n", child);
12     sleep(60);
13 }
```

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$
```

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. Folgender Programmtext 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:

Programm 1.9: Ein Prozess wird zum Waisenkind (*orphan.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main() {
6      pid_t child;
7      child = fork();
8      if (child == -1) {
9          perror("fork"); exit(1);
```

```
10      }
11      if (child == 0) {
12          printf("Hi, my parent is %d\n", (int) getppid());
13          sleep(5);
14          printf("My parent is now %d\n", (int) getppid());
15      }
16      sleep(3);
17      exit(0);
18  }
```

```
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* bzw. */sbin/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.

1.8 Der Wechsel zu einem anderen Programm

Mit *fork*() ist es möglich, neue Prozesse zu erzeugen. Allerdings teilen die neuen Prozesse sich den Programmtext mit ihrem Erzeuger. Wie ist nun der Wechsel zu einem anderen Programmtext 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

Der folgende Programmtext dient als erstes Beispiel, bei dem der laufende Prozess seinen Programmtext durch den von *date* austauscht:

Programm 1.10: Wechsel zum Programm *date* (*datum.c*)

```
1   #include <unistd.h>
2
3   int main() {
4       execl(
5           "/usr/bin/date", /* path of the program */
6           "/usr/bin/date", /* name of the program, i.e. argv[0] */
7           "+%d.%m.%Y", /* first argument, i.e. argv[1] */
8           0 /* terminate list of arguments */
9       );
10      /* not reached except if execl failed */
11      perror("/usr/bin/date");
12  }
```

Dabei ergibt sich die Möglichkeit, Kommandozeilenargumente zu übergeben, die dann vom neuen Programm in *argv*[] vorgefunden werden. Hierbei 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.

1.9 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 mit einem Systemaufruf, der *fork*() und *exec*() kombiniert.
- 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 1.3 zeigt, wird genau dies von den UNIX-Shells genutzt, um die Umgebung für Anwendungen einzurichten. Der übergeordnete Prozess der Shell wartet dann normalerweise mit *wait*() auf das Ende des untergeordneten Prozesses.

Das folgende Beispiel 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. Die eingelesene Zeile wird dann mit der Hilfe des *tokenizer* () in einzelne Wörter zerlegt, die dann als Kommandoname und als weitere Argumente des Kommandos interpretiert werden.

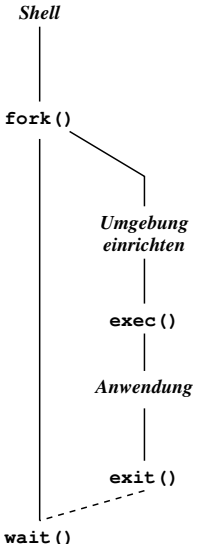

Abbildung 1.3: Start einer Anwendung von der Shell

Programm 1.11: Hauptprogramm einer einfachen Shell (*tinysh.c*)

```
1  /*
2
3  =head1 NAME
4
5  tinysh −− a tiny shell with a minimal set of features
6
7  =head1 SYNOPSIS
8
9  B<tinysh>
10
11 =head1 DESCRIPTION
12
13 B<tinysh> reads command lines from its standard input
14 and executes them. % '' is given as prompt.
15
16 Each command line consists of space−separated tokens. Special tokens
17 begin with <'', indicating the input file, >'', indicating the output
18 file, and >>'', specifying an output file that is to be extended. The
19 first non−special token specifies the command name which must be either
20 an absolute path of an executable file or locatable within the list of
21 directories provided by the environment variable B<PATH>.
22
23 =head1 DIAGNOSTICS
24
25 B<tinysh> prints the exit code of a terminated program if it is
26 non−zero. It tells that a program terminated abnormally'' if it
27 did not exit. An exit code of 255 is used by subprocesses if they
28 are unable to start the command for some reason.
29
30 =head1 BUGS
```

```
31
32  No signal handling, no support of backgrounded commands, no pipelines,
33  no builtins, no shell variables.
34
35  =head1 AUTHOR
36
37  Andreas Borchert
38
39  =cut
40
41  */
42
43  #include <fcntl.h>
44  #include <stdio.h>
45  #include <stdlib.h>
46  #include <unistd.h>
47  #include <sys/wait.h>
48  #include <afblib/strlist.h>
49  #include <afblib/tokenizer.h>
50  #include "sareadline.h"
51
52  /*
53   * assign an opened file with the given flags and mode to fd
54   */
55  void fassign(int fd, char* path, int oflags, mode_t mode) {
56      int newfd = open(path, oflags, mode);
57      if (newfd < 0) {
58          perror(path); exit(255);
59      }
60      if (dup2(newfd, fd) < 0) {
61          perror("dup2"); exit(255);
62      }
63      close(newfd);
64  } // fassign
65
66  int main() {
67      stralloc line = {0};
68      while (printf("%% "), readline(stdin, &line)) {
69          strlist tokens = {0};
70          stralloc_0(&line); /* required by tokenizer() */
71          if (!tokenizer(&line, &tokens)) break;
72          if (tokens.len == 0) continue;
73          pid_t child = fork();
74          if (child == -1) {
75              perror("fork"); continue;
76          }
77          if (child == 0) {
78              strlist argv = {0}; /* list of arguments */
79              char* cmdname = 0; /* first argument */
80              char* path; /* of output files */
81              int oflags;
82
83              for (int i = 0; i < tokens.len; ++i) {
```

```
84                  switch (tokens.list[i][0]) {
85                      case '<':
86                          fassign(0, &tokens.list[i][1], O_RDONLY, 0);
87                          break;
88                      case '>':
89                          path = &tokens.list[i][1];
90                          oflags = O_WRONLY|O_CREAT;
91                          if (*path == '>') {
92                              ++path; oflags |= O_APPEND;
93                          } else {
94                              oflags |= O_TRUNC;
95                          }
96                          fassign(1, path, oflags, 0666);
97                          break;
98                      default:
99                          strlist_push(&argv, tokens.list[i]);
100                         if (cmdname == 0) cmdname = tokens.list[i];
101                 }
102             }
103             if (cmdname == 0) exit(0);
104             strlist_push0(&argv);
105             execvp(cmdname, argv.list);
106             perror(cmdname);
107             exit(255);
108         }
109
110         /* wait for termination of child */
111         int stat;
112         pid_t pid = wait(&stat);
113         if (pid == child) {
114             if (WIFEXITED(stat)) {
115                 int code = WEXITSTATUS(stat);
116                 if (code && code != 255) {
117                     printf("terminated␣with␣exit␣code␣%d\n", code);
118                 }
119             } else {
120                 printf("terminated␣abnormally\n");
121             }
122         } else {
123             perror("wait");
124         }
125     }
126 } // main
```

Die folgende Implementierung der Funktion *readline*() basiert auf der *stralloc*-Bibliothek:

Programm 1.12: Zeilenweises Einlesen mit der *stralloc*-Bibliothek (*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 #include "sareadline.h"
11
12 bool readline(FILE* fp, stralloc* sa) {
13     sa->len = 0;
14     for(;;) {
15         int ch = getc(fp);
16         if (ch == EOF) return sa->len > 0;
17         if (ch == '\n') break;
18         if (!stralloc_readyplus(sa, 1)) return false;
19         sa->s[sa->len++] = ch;
20     }
21     return true;
22 } // readline
```

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. Der folgende Programmtext zeigt, wie dies erfolgen kann:

Programm 1.13: Datenstruktur und Schnittstelle für Listen von Zeichenketten (*strlist.h*)

```
1  /*
2   * Data structure for dynamic string lists that works
3   * similar to the stralloc library.
4   * Return values: 1 if successful, 0 in case of failures.
5   * afb 4/2003
6   */
7
8  #ifndef AFBLIB_STRLIST_H
9  #define AFBLIB_STRLIST_H
10
11 #include <stddef.h>
12 #include <stdbool.h>
13
14 typedef struct strlist {
15     char** list;
16     size_t len; /* # of strings in list */
17     size_t allocated; /* allocated length for list */
18 } strlist;
19
20 /* assure that there is at least room for len list entries */
21 bool strlist_ready(strlist* list, size_t len);
22
23 /* assure that there is room for len additional list entries */
24 bool strlist_readyplus(strlist* list, size_t len);
25
26 /* truncate the list to zero length */
```

```
27 void strlist_clear(strlist* list);
28
29 /* append the string pointer to the list */
30 bool strlist_push(strlist* list, char* string);
31 #define strlist_push0(list) strlist_push((list), 0)
32
33 /* free the strlist data structure but not the strings */
34 void strlist_free(strlist* list);
35
36 #endif
```

Programm 1.14: Verwaltung von Listen für Zeichenketten (*strlist.c*)

```
1  /*
2   * Data structure for dynamic string lists that works
3   * similar to the stralloc library.
4   * Return values: 1 if successful, 0 in case of failures.
5   * afb 4/2003
6   */
7  #include <stdlib.h>
8  #include <afblib/strlist.h>
9
10 /* assure that there is at least room for len list entries */
11 bool strlist_ready(strlist* list, size_t len) {
12     if (list->allocated < len) {
13         size_t wanted = len + (len>>3) + 8;
14         char** newlist = (char**) realloc(list->list,
15             sizeof(char*) * wanted);
16         if (newlist == 0) return false;
17         list->list = newlist;
18         list->allocated = wanted;
19     }
20     return true;
21 }
22
23 /* assure that there is room for len additional list entries */
24 bool strlist_readyplus(strlist* list, size_t len) {
25     return strlist_ready(list, list->len + len);
26 }
27
28 /* truncate the list to zero length */
29 void strlist_clear(strlist* list) {
30     list->len = 0;
31 }
32
33 /* append the string pointer to the list */
34 bool strlist_push(strlist* list, char* string) {
35     if (!strlist_ready(list, list->len + 1)) return false;
36     list->list[list->len++] = string;
37     return true;
38 }
39
40 /* free the strlist data structure but not the strings */
```

```
41 void strlist_free(strlist* list) {
42     free(list->list); list->list = 0;
43     list->allocated = 0;
44     list->len = 0;
45 }
```

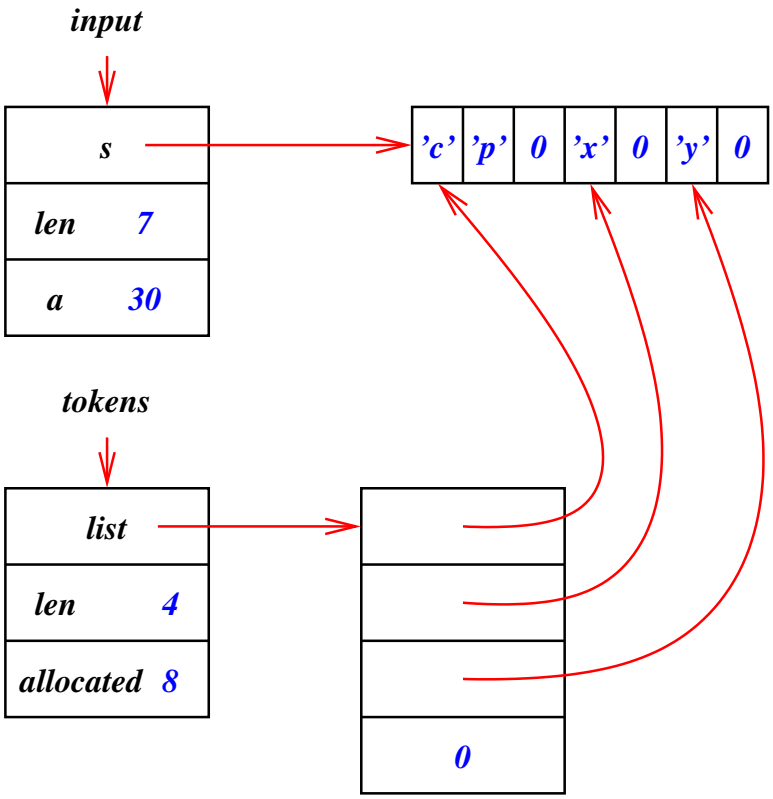

Abbildung 1.4: Resultierende Datenstruktur des Wortzerlegers am Beispiel „cp x y“

Der Wortzerleger *tokenizer*() 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 1.4 zeigt die resultierende Datenstruktur an einem Beispiel.

Programm 1.15: Zerlegung einer Zeile in Wörter (*tokenizer.c*)

```
1  /*
2   * Simple tokenizer: Take a 0-terminated stralloc object and return a
3   * list of pointers in tokens that point to the individual tokens.
4   * Whitespace is taken as token-separator and all whitespaces within
5   * the input are replaced by null bytes.
6   * afb 4/2003
7   */
8
9  #include <ctype.h>
10 #include <stdlib.h>
```

```
11 #include <stralloc.h>
12 #include <stdbool.h>
13 #include <afblib/strlist.h>
14 #include <afblib/tokenizer.h>
15
16 bool tokenizer(stralloc* input, strlist* tokens) {
17     char* cp;
18     int white = 1;
19
20     strlist_clear(tokens);
21     for (cp = input->s; *cp && cp < input->s + input->len; ++cp) {
22         if (isspace((int) *cp)) {
23             *cp = '\0'; white = 1; continue;
24         }
25         if (!white) continue;
26         white = 0;
27         if (!strlist_push(tokens, cp)) return false;
28     }
29     return true;
30 }
```

Damit die Einrichtung einer Umgebung für den neuen Prozess demonstriert 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 innerhalb der **if**-Anweisung **if** (*child* == 0) 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.

Die Feinarbeit übernimmt hier die Funktion *fassign*, 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 diese alte Verbindung zuerst gekappt.