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

Wechsel zum Programm *date*

datum.c

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

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

- Dieses Programm ersetzt seinen eigenen Programmtext durch den von *date*.

datum.c

```
execl(
   "/usr/bin/date",  /* path of the program */
   "/usr/bin/date",  /* name of the program, i.e. argv[0] */
   "+%d.%m.%Y",      /* first argument, i.e. argv[1] */
   0                 /* terminate list of arguments */
);
```

- *execl* erlaubt die Angabe beliebig vieler Kommandozeilenargumente in der Form einzelner Funktionsparameter. Mit einem Nullzeiger wird die Liste der Parameter beendet.
- Dabei ist zu beachten, dass der Pfadname des auszuführenden Programms und der später unter *argv*[0] zu findende Kommandoname getrennt angegeben werden. Normalerweise sind beide gleich, es gibt aber auch Ausnahmen.

datum.c

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

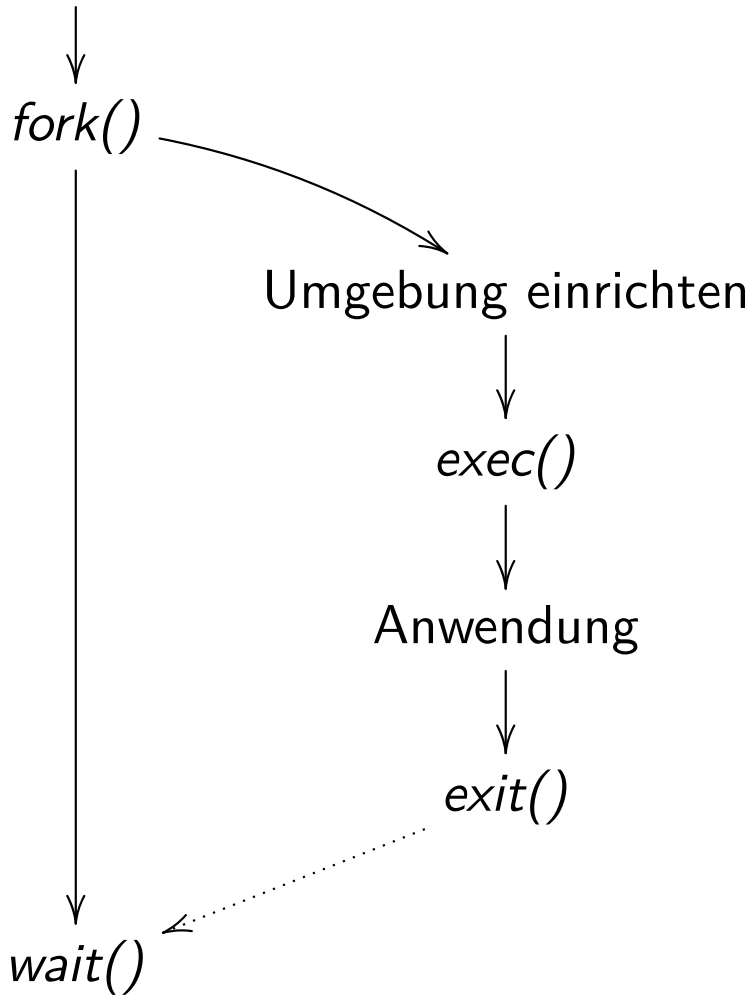
- Normalerweise geht es im Programmtext nach einem Aufruf von *execl*() nicht weiter, weil im Erfolgsfalle das Programm ausgetauscht wurde. Nur bei einem Fehler (weil z.B. das *date*-Kommando nicht gefunden wurde) wird das Programm hinter dem Aufruf von *execl*() fortgesetzt.

- 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.

```
//IS198CPY JOB (IS198T30500),'COPY JOB',CLASS=L,MSGCLASS=X
//COPY01   EXEC PGM=IEBGENER
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DSN=OLDFILE,DISP=SHR
//SYSUT2   DD  DSN=NEWFILE,
//           DISP=(NEW,CATLG,DELETE),
//           SPACE=(CYL,(40,5),RLSE),
//           DCB=(LRECL=115,BLKSIZE=1150)
//SYSIN    DD  DUMMY
```

- 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.
- Das Beispiel zeigt ein Kopierkommando in der JCL (Job Command Language) aus der IBM-Mainframe-Welt (von der Wikipedia übernommen). Hieran zeigt sich, dass dies die Kommandosprache deutlich verkompliziert. 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*() erlaubt die Konfiguration der Umgebung des aufzurufenden Programms innerhalb der Shell mit ganz normalen Systemaufrufen, die sich auf den eigenen Prozess beziehen.

```
theon$ tinysh
% date
Mon Apr 29 11:09:30 CEST 2019
% date >out
% cat out
Mon Apr 29 11:09:33 CEST 2019
% awk {print$4} <out
11:09:33
% theon$
theon$
```

- Die kleine Shell *tinysh* erlaubt
  - den Aufruf von Kommandos mit beliebig vielen Parametern, die durch Leerzeichen getrennt werden,
  - die Umlenkung der Standard-Ein- und Ausgabe, wobei auch das Anhängen unterstützt wird und
  - die Auswertung des *wait*-Systemaufrufs.
- Die Konfiguration des aufzurufenden Programms erfolgt hier zwischen *fork* und *exec*.

tinysh.c

```
int main() {
   inbuf in = {0}; outbuf out = {1};
   stralloc line = {0}; strlist tokens = {0};
   while (outbuf_printf(&out, "%% ") >= 0 && outbuf_flush(&out) &&
         inbuf_sareadline(&in, &line)) {
      stralloc_0(&line); /* required by tokenizer() */
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      pid_t child = fork();
      if (child == -1) {
         print_error("fork"); continue;
      }
      if (child == 0) {
         // setup child and argv.list ...
         execvp(cmdname, argv.list);
         print_error(cmdname); exit(255);
      }

      /* wait for termination of child */
      // ...
   }
} // main
```

Zeilenweises Einlesen mit der stralloc-Bibliothek 60

inbuf_sareadline.c

```
bool inbuf_sareadline(inbuf* ibuf, stralloc* sa) {
   sa->len = 0;
   for(;;) {
      int ch;
      if ((ch = inbuf_getchar(ibuf)) < 0) return false;
      if (ch == '\n') break;
      if (!stralloc_readyplus(sa, 1)) return false;
      sa->s[sa->len++] = ch;
   }
   return true;
}
```

- Diese Funktion erlaubt das Einlesen beliebig langer Zeilen auf Basis der *inbuf*-Bibliothek.
- Mit *stralloc_readyplus* wird jeweils Platz für mindestens ein weiteres Zeichen geschaffen.
- Die resultierende Zeichenkette ist *nicht* durch ein Nullbyte terminiert.

Erzeugung der Liste mit Kommandozeilenparametern 61

- Die Funktion *execl* ist für die *tinysh* ungeeignet, da die Zahl der Kommandozeilenparameter nicht feststeht. Diese soll auch nicht durch das Programm künstlich begrenzt werden.
- Alternativ zu *execl* gibt es *execv*, das einen Zeiger auf eine Liste mit Zeigern auf Zeichenketten erwartet, die am Ende mit einem Null-Zeiger abzuschliessen ist.
- Die in der *tinysh* verwendete Funktion *execvp* (mit zusätzlichem p) sucht im Gegensatz zu *execv* nach dem Programm in allen Verzeichnissen, die die Umgebungsvariable *PATH* aufzählt.

Erzeugung einer Liste mit Zeigern auf Zeichenketten 62

strlist.h

```
#ifndef STRLIST_H
#define STRLIST_H

#include <stddef.h>
#include <stdbool.h>

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

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

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

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

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

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

#endif
```

Erzeugung einer Liste mit Zeigern auf Zeichenketten 63

strlist.h

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

bool strlist_ready(strlist* list, size_t len);
bool strlist_readyplus(strlist* list, size_t len);
void strlist_clear(strlist* list);
bool strlist_push(strlist* list, char* string);
void strlist_free(strlist* list);
```

- Die *strlist*-Bibliothek folgt weitgehend dem Vorbild der *stralloc*-Bibliothek.

Erzeugung einer Liste mit Zeigern auf Zeichenketten 64

strlist.c

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

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

Erzeugung einer Liste mit Zeigern auf Zeichenketten 65

strlist.c

```
void strlist_clear(strlist* list) {
   list->len = 0;
}

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

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

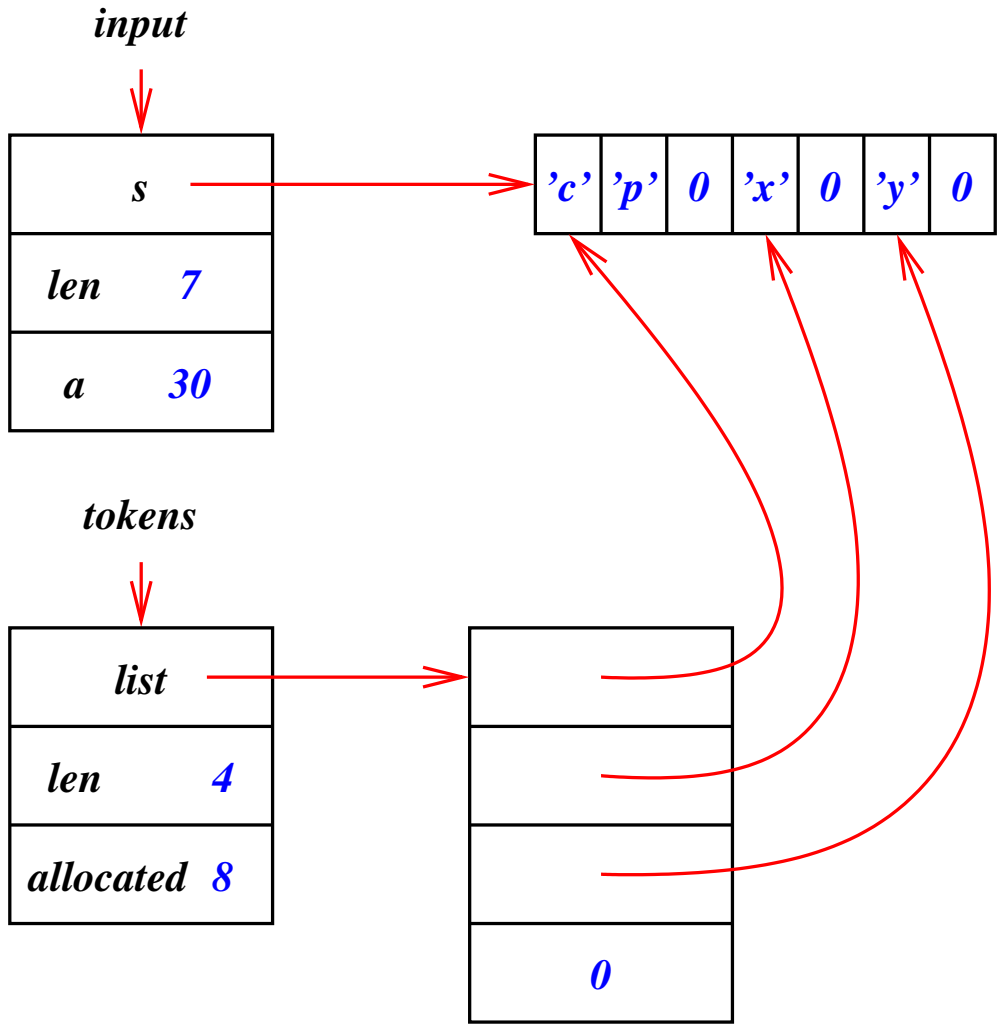
tokenizer.h

```
#ifndef AFBLIB_TOKENIZER_H
#define AFBLIB_TOKENIZER_H

#include <stdbool.h>
#include <stralloc.h>
#include <afblib/strlist.h>
bool tokenizer(stralloc* input, strlist* tokens);

#endif
```

- Die Funktion *tokenizer* zerlegt die Eingabezeile in *input* in einzelne (durch Leerzeichen getrennte) Wörter und fügt diese in die Liste *tokens*.
- Wesentlich ist hier, dass die einzelnen Zeichenketten nicht dupliziert werden, sondern innerhalb der Eingabezeile verbleiben. Zu diesem Zweck werden Leerzeichen durch Nullbytes ersetzt.



- Das Diagramm zeigt die resultierende Datenstruktur des Wortzerlegers am Beispiel „cp x y“.

Zerlegung einer Eingabezeile

tokenizer.c

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

#include <ctype.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stralloc.h>
#include <afblib/strlist.h>
#include <afblib/tokenizer.h>

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

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

tinysh.c

```
while (outbuf_printf(&out, "%% ") >= 0 && outbuf_flush(&out) &&
       inbuf_sareadline(&in, &line)) {
   stralloc_0(&line); /* required by tokenizer() */
   if (!tokenizer(&line, &tokens)) break;
   if (tokens.len == 0) continue;
   // ...
}
```

- Da der Wortzerleger nullbyte-terminierte Zeichenketten liefert, muss mit *stralloc_0* noch ein Nullbyte angehängt werden.
- Falls keine Wörter zu finden sind, wird sofort die nächste Zeile eingelesen.
- Die Erzeugung der Kommandozeilenparameterliste wird dem neu zu erzeugenden Prozess überlassen.

Konfiguration des neuen Prozesses

tinysh.c

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

tinysh.c

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

- Mit dem Systemaufruf *dup2* lässt sich ein Dateideskriptor auf einen gegebenen anderen Deskriptor duplizieren, die dann beide auf den gleichen Eintrag in der *Open File Table* verweisen.
- So lassen sich neu eröffnete Datei-Verbindungen mit vorgegebenen Dateideskriptoren wie etwa 0 (stdin) oder 1 (stdout) verknüpfen.