– Lösung zur Praktikumsaufgabe 9 –

Thema: *Pipes*

2. Zur Beendigung des Programmes sind verschiedene Ansätze denkbar. Wenn explizit auf eine bestimmtes Symbol (z. B. „quit“, Leerzeile) getestet wird, dann darf dieses Symbol im Klartext natürlich nicht vorkommen. Eine Beendigung per Signal ist ebenfalls möglich, aber unelegant und außerdem noch nicht behandelt.

   Am elegantesten ist es, mit der Funktion `feof()` auf EOF („End Of File“) zu prüfen. Wird das Programm direkt aufgerufen, so muss man EOF mit der Konsole generieren (CTRL-D), beim Aufruf innerhalb einer Pipe erledigt dies das vorangehende Kommando.

Listing 1: Lösung von Aufgabe 2)

```
/*
   reads a line from stdin, ROT-13 it and writes it to stdout
   ends, when EOF is read (^D)
*/


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

#define LINESIZE 200

void rot13(char *buf)
{
  int c;
  char x;

  for (c=0; c < strlen(buf); c++) {
    x = buf[c];
    if (isalpha(x)) {
      if ( ((x >= 'a') && (x <= 'm')) || ((x >= 'A') && (x <= 'M') ) ) {
  x += 13;
      }
      else {
  x -= 13;
      }
    }
    buf[c] = x;
  }
}

int main(int argc, char* argv[])
{
  char buf[LINESIZE];

  fgets(buf, LINESIZE, stdin);
```

```
  while(!feof(stdin)){
    rot13(buf);
    printf("%s", buf);
    fgets(buf, LINESIZE, stdin);
  }
  return EXIT_SUCCESS;
}
```

Eine typische Aufrufsequenz könnte folgendermaßen aussehen:

```
robge@hadrian:~> ./aufgabe-09-02a
Wenn ist das Nunstück git und Slotermeyer?
Jraa vfg qnf Ahafgüpx tvg haq Fybgrezrlre?
Beiherhund
Orvureuhaq
Orvureuhaq
Beiherhund
robge@hadrian:~> echo Jehova | ./aufgabe-09-02a
Wrubin
robge@hadrian:~> echo Jehova | ./aufgabe-09-02a | ./aufgabe-09-02a
Jehova
```

Zweimalige Verschlüsselung mittels ROT-13 liefert wieder den Originaltext. Überzeugen Sie sich auch, dass das Programm z. B. auch mit einer leeren Datei zurechtkommt.

Die Umkehrung der eingegebenen Zeichenkette ist programmiertechnisch auch nicht schwieriger. Die Lösung zeigt die Beendigung mittels Kommando „quit“.

Listing 2: Lösung von Aufgabe 2) mit zeichenweiser Umkehrung der Zeile

```
/*
   reads a line from stdin, inverts it and writes it to stdout
*/


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

#define LINESIZE 200

void invert(char *buf)
{
  int c, l;
  char x;

  l = strlen(buf)-1;  /* do not exchange trailing '\n' */

  for (c=0; c < l/2; c++) {
    x = buf[c];
    buf[c] = buf[l-c-1]; // Why 'l-c-1' ?
    buf[l-c-1] = x;
```

```
  }
}

int main(int argc, char* argv[])
{
  char buf[LINESIZE];

  fgets(buf, LINESIZE, stdin);
  /* Abbruch bei 'quit' */
  while( strcmp(buf, "quit\n") ) {
    invert(buf);
    printf("%s", buf);
    fgets(buf, LINESIZE, stdin);
  }

  return EXIT_SUCCESS;
}
```

Ein Shellskript ist zur Realisierung der ROT-13-Verschlüsselung nicht nötig; es reicht ein einfaches Kommando:

```
robge@hadrian~> echo Jehova! | tr A-MN-Za-mn-z N-ZA-Mn-za-m
Wrubin!
```

3. a) Es ist günstig, bei mehreren Prozessen und Ressourcen sich zunächst klarzumachen, welche Aktivitäten und Ressourcen und welche Beziehungen zwischen diesen existieren (Abb. 1).

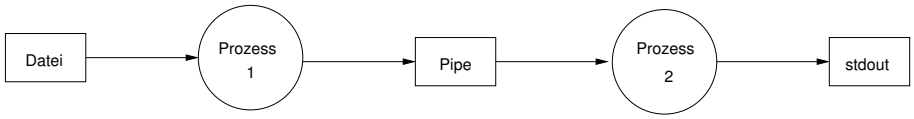

Abbildung 1: Kommunikationsbeziehungen der Prozesse aus Aufgabe 3

Es handelt sich um eine typische *Verarbeitungskette*. Bei der Abarbeitung eines solchen Konstrukts kann man grob drei Phasen unterscheiden:

1. Etablierung und Initialisierung der Infrastruktur (Prozesse erzeugen, IPC-Mechanismen erzeugen und initialisieren, Dateien eröffnen),
2. die eigentliche Arbeit (im wesentlichen liest jeder Prozess Daten aus einer Quelle, verarbeitet diese und schreibt sie an ein Ziel),
3. Abbau der Infrastruktur: Beendigung der Kindprozesse inklusive Testung der Rückgabewerte, Zerstörung/Rückgabe nicht mehr benötigter Ressourcen, Schließen von Dateien.

Im ersten Schritt müssen konkret

- die Lesedatei eröffnet,

- ein neuer Prozess erzeugt,
- die unbenutzten Deskriptoren der Pipe geschlosssen,
- die Pipe erzeugt,

werden. Die Reihenfolge dieser Aktionen ist nicht beliebig, sondern hängt u. a. davon ab, welche Ressourcen der Sohnprozess mit erben muss (hier: die Pipe, aber nicht die eröffnete Datei). In dieser Aufgabe ist eine zu obiger Aufzählung genau umgekehrte Reihenfolge dieser Initialisierungsoperationen sinnvoll.

Der Abbau der Infrastruktur wird durch den Vaterprozess initiiert. Wenn die Datei vollständig eingelesen wurde, schließt er diese, schreibt die Restdaten in die Pipe und schließt auch deren Deskriptor. Der Sohn erkennt, dass keine weiteren Daten aus der Pipe zu erwarten sind und schließt daher diesen Deskriptor. Da nun alle Deskriptoren der Pipe geschlossen sind, vernichtet das Betriebssystem diese automatisch (ein erneutes Öffnen ist unmöglich). Der Sohn schreibt die restlichen Daten nach `stdout` und beendet sich dann. Der Vater sollte auf das Ende des Sohnes per `wait()` warten, den Rückgabewert auswerten und sich anschließend ebenfalls beenden.

Listing 3: Beispielhafte Lösung von Aufgabe 3a)

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

#define LINESIZE 200

int main(int argc, char* argv[])
{
  int ret;
  int fd[2], rfd;
  pid_t pid, wpid;
  ssize_t in_father, in_son;
  char buf_father, buf_son[LINESIZE];

  if (argc != 2) {
    printf("Usage %s <file>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  rfd = open(argv[1], O_RDONLY);
  if (rfd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
  }
  ret = pipe(fd);
  if (ret == -1) {
    perror("pipe");
```

```
  exit(EXIT_FAILURE);
}
pid = fork();
if (pid == -1) {
  perror("fork");
  exit(EXIT_FAILURE);
}

if (pid == 0) { /* Son */
  ret = close(fd[1]);
  if (ret == -1) {
    perror("close (son, pipe)");
    exit(EXIT_FAILURE);
  }

  /* read file from pipe and print it to stdout */

  in_son = read(fd[0], &buf_son, LINESIZE-1);
  if (in_son == -1) {
    perror("read (son, pipe)");
    close(fd[0]);
    exit(EXIT_FAILURE);
  }
  while (in_son != 0) {
    ret = write(0, buf_son, in_son);
    if (ret == -1) {
perror("write (son, stdout)");
break;
    }
    in_son = read(fd[0], &buf_son, LINESIZE-1);
    if (in_son == -1) {
perror("write (son, stdout)");
break;
    }
  }

  /* close file descriptor */
  ret = close(fd[0]);
  if (ret == -1) {
    perror("close (son, pipe)");
    exit(EXIT_FAILURE);
  }
  printf("Son exits.\n");
  exit(EXIT_SUCCESS);
} /* End of son */

else { /* Father */
  ret = close(fd[0]);
  if (ret == -1) {
    perror("close (father, pipe)");
    exit(EXIT_FAILURE);
  }

  /* read in file and put it into pipe */
  in_father = read(rfd, &buf_father, 1);
```

```
      if (in_father == -1) {
        perror("read (father, file)");
        close(fd[1]);
        close(rfd);
        exit(EXIT_FAILURE);
      }
      while (in_father != 0) {
        ret = write(fd[1], &buf_father, 1);
        if (ret == -1) {
    perror("write (father, pipe)");
    break;
        }
        in_father = read(rfd, &buf_father, 1);
        if (in_father == -1) {
    perror("read (father, file)");
    break;
        }
      }

      /* close used descriptors */
      ret = close(fd[1]);
      if (ret == -1) {
        perror("close (father, pipe)");
        exit(EXIT_FAILURE);
      }
      ret = close(rfd);
      if (ret == -1) {
        perror("close (father, file)");
        exit(EXIT_FAILURE);
      }
      wpid = wait(NULL);
      if (wpid == -1) {
        perror("wait (father)");
        exit(EXIT_FAILURE);
      }
      printf("Father exits.\n");
      exit(EXIT_SUCCESS);
    } /* End of Father */
  }
```

b)*

Einige Überlegungen:

- Der Vater benötigt nun zwei unabhängige Puffer, da er zweimal liest und zweimal schreibt.
- Es ist günstig, nun mit einer einheitlichen Blockgröße für alle Lese- und Schreiboperationen zu operieren.
- Es ist vom Standpunkt der Parallelisierung ungünstig, dass der Vater zwei *verschiedene* Datenströme bedienen muss; besser wäre es, diese auf verschiedene Prozesse zu verteilen.

- Kommunikationsbeziehungen skizziert Abb. 2

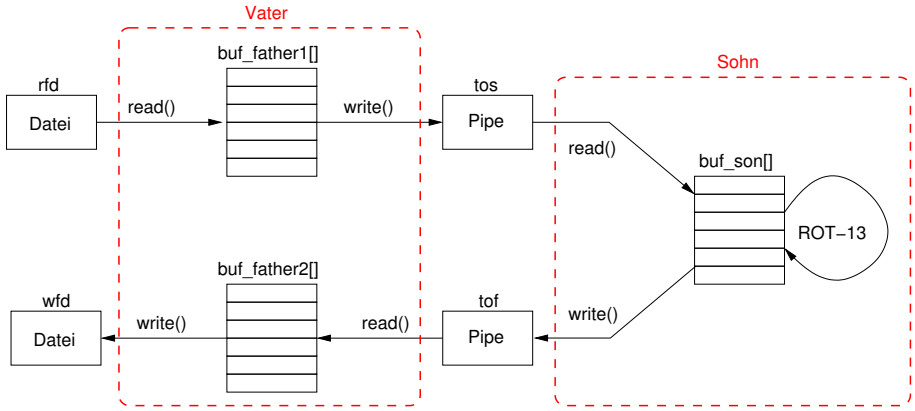

Abbildung 2: Kommunikationsbeziehungen der Prozesse aus Aufgabe 3b

Listing 4: Beispielhafte Lösung von Aufgabe 3b)

```
/*
  o Father reads data from file (argv[0]), sends data via pipe (
     tos) to son
  o son reads data from pipe (tos), rot13s data, sends data via
     pipe (tof)
    to father
  o father reads data from pipe (tof) and writes it to file (argv
     [1])

  Bugs:
  - testing return value of close() omitted for clarity of code
*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>

#define READ  0
#define WRITE 1

#define READ  0
#define WRITE 1

#define BLOCKSIZE 512

/*
  rotates-13 all characters of buf with size buflen
*/
void rot13(char *buf, int buflen)
{
  int c;
```

```
  char x;

  for (c=0; c < buflen; c++) {
    x = buf[c];
    if (isalpha(x)) {
      if ( ((x >= 'a') && (x <= 'm')) ||
     ((x >= 'A') && (x <= 'M')) ) {
        x += 13;
      }
      else {
        x -= 13;
      }
    }
    buf[c] = x;
  }
}


int main(int argc, char* argv[])
{
  int ret;
  int tos[2], tof[2], rfd, wfd;
  pid_t pid, wpid;
  ssize_t in_father, in_father2, in_son;
  char buf_father1[BLOCKSIZE], buf_father2[BLOCKSIZE],
    buf_son[BLOCKSIZE];

  if (argc != 3) {
    printf("Usage %s <infile> <outfile>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  /* open files for reading and writing, respectively */
  rfd = open(argv[1], O_RDONLY);
  if (rfd == -1) {
    perror("open() infile");
    exit(EXIT_FAILURE);
  }
  wfd = creat(argv[2], S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
  if (wfd == -1) {
    perror("open() outfile");
    close(rfd);
    exit(EXIT_FAILURE);
  }

  /* open two pipes */
  ret = pipe(tos); /* from father TO Son */
  if (ret == -1) {
    perror("pipe() to son");
    close(wfd);
    close(rfd);
    exit(EXIT_FAILURE);
  }
  ret = pipe(tof); /* from son TO Father */
  if (ret == -1) {
```

```
    perror("pipe() to father");
    close(tos[READ]);
    close(tos[WRITE]);
    close(wfd);
    close(rfd);
    exit(EXIT_FAILURE);
  }

  pid = fork();
  if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }

  if (pid == 0) {
    /* code of son */
    close(tos[WRITE]);
    close(tof[READ]);

    /* read file from pipe1, rot13 buffer and write to pipe2 */
    in_son = read(tos[READ], buf_son, BLOCKSIZE);
    if (in_son == -1) {
      perror("read (son, pipe to son)");
      close(tos[READ]);
      close(tof[WRITE]);
      exit(EXIT_FAILURE);
    }
    while (in_son != 0) {
      rot13(buf_son, in_son);

      ret = write(tof[WRITE], buf_son, in_son);
      if (ret == -1) {
  perror("write (son, pipe to father)");
  break;
      }
      in_son = read(tos[READ], buf_son, BLOCKSIZE);
      if (in_son == -1) {
  perror("read (son, pipe to son)");
  break;
      }
    }

    /* close unneeded file descriptors */
    close(tos[READ]);
    close(tof[WRITE]);
    printf("Son exits.\n");
    exit(EXIT_SUCCESS);
  } /* End of son */

  else {
    /* father's code */
    close(tos[READ]);
    close(tof[WRITE]);

    /* read infile,
```

```
        write to tos pipe,
        read from tof pipe,
        write to outfile
      */
      in_father = read(rfd, buf_father1, BLOCKSIZE);
      if (in_father == -1) {
        perror("read (father, in file)");
        close(tos[WRITE]);
        close(tof[READ]);
        close(rfd);
        close(wfd);
        exit(EXIT_FAILURE);
      }
      while (in_father != 0) {
        ret = write(tos[WRITE], buf_father1, in_father);
        if (ret == -1) {
  perror("write (father, tos pipe)");
  break;
        }
        in_father2 = read(tof[READ], buf_father2, BLOCKSIZE);
        if (in_father2 == -1) {
  perror("read (father, tof pipe)");
  break;
        }
        ret = write(wfd, buf_father2, in_father2);
        if (ret == -1) {
  perror("write (father, outfile)");
  break;
        }
        in_father = read(rfd, buf_father1, BLOCKSIZE);
        if (in_father == -1) {
  perror("read (father, file)");
  break;
        }
      }

      /* close used descriptors */
      close(tos[WRITE]);
      close(tof[READ]);
      close(rfd);
      close(wfd);

      wpid = wait(NULL);
      if (wpid == -1) {
        perror("wait (father)");
        exit(EXIT_FAILURE);
      }
      printf("Father exits.\n");
      exit(EXIT_SUCCESS);
    } /* End of Father */
}
```