Alle Lösungsdateien finden Sie auch im Archiv **uebung02-loesung.tar.gz** auf der Webseite.

2. System Calls: fork und exec

In Aufgabe 1 (letzte Übung) haben Sie `fork` und `exec` bereits verwendet. Betrachten Sie das folgende Programm (das Sie auch im Archiv *uebung2.tar.gz* auf der Webseite finden):

`[...]`

a) Jeder Aufruf von `fork()` verdoppelt die Anzahl der Prozesse; jeder Aufruf von `exec()` ersetzt den aufrufenden Prozess durch das angegebene Programm. Wie viele Ausgaben der Zeile „Prozessende ...“ erwarten Sie?

Es gibt vier solche Ausgabezeilen. Zwar wird der Prozess 3x verdoppelt (so dass es acht Prozesse gibt), aber nach dem ersten `fork()` fällt einer der beiden Prozesse durch `exec()` weg. Damit bleibt nur einer, der das Programm fortsetzt.

3. System Calls in Assembler und C

```
int syscall (int eax, int ebx, int ecx, int edx) {
  int result;
  asm (
    "int $0x80"
    : "=a" (result)
    : "a" (eax), "b" (ebx), "c" (ecx), "d" (edx)
  );
  return result;
}
```

Dieser Code funktioniert auch unter 64-Bit-Linux, wenn

- Sie `gcc` mit der Option `-m32` aufrufen, was nur möglich ist, wenn die 32-Bit-Umgebung des C-Compilers installiert,
- wozu Sie unter Ubuntu `sudo apt-get install g++-multilib` eingeben müssen (und das wird wohl auf den Praktikumsrechnern nicht klappen).

Vielleicht ist das Paket aber schon installiert – probieren Sie mal, ob `gcc -m32 …` funktioniert.

a) Die Lösung zu a) ist dann:

```
// fork+write-syscall-32bit.c
int main() {
  char vater[]="Ich bin der Vater.\n"; int vlen=sizeof(vater);
  char sohn[]="Ich bin der Sohn.\n";   int slen=sizeof(sohn);
  // int pid=fork();
  int pid=syscall (2,0,0,0);
  if (pid) {
    // write(1,&vater,vlen);
    syscall (4,1, (long)&vater, vlen);
  }
  else {
    // write(1,&sohn,slen);
    syscall (4,1, (long)&sohn, slen);
  }
  return 0;
}
```

Wenn Sie die 32-Bit-Variante nicht verwenden können/wollen, geht es auch direkt 64-bittig:

```
// fork+write-syscall-64bit.c
#include <asm/unistd.h>

int syscall (long rax, long rdi, long rsi, long rdx) {
  int result;
  asm (
    "syscall"
    : "=a" (result)
    : "a" (rax), "D" (rdi), "S" (rsi), "d" (rdx)
  );
  return result;
}

int main() {
  char vater[]="Ich bin der Vater.\n";
  int vlen=sizeof(vater);
  char sohn[]="Ich bin der Sohn.\n";
  int slen=sizeof(sohn);

  // andere Syscall-Nummern für 64-Bit-Code; verwende __NR_*
  int pid=syscall(__NR_fork,0,0,0);

  if (pid) {
    // write(1,&vater,vlen);
    syscall (__NR_write, 1, (long)&vater, vlen);
  }
  else {
    // write(1,&sohn,slen);
    syscall (__NR_write, 1, (long)&sohn, slen);
  }
  return 0;
}
```

Zum Verständnis der Unterschiede:

- 32-Bit-Code erwartet die Argumente in den 32-Bit-Registern `eax`, `ebx`, `ecx` und `edx` (die im `asm`-Aufruf über `a`, `b`, `c`, `d` anzusprechen sind),
- 64-Bit-Code erwartet die Argumente in den 64-Bit-Registern `rax`, `rdi`, `rsi` und `rdx` (die im `asm`-Aufruf über `a`, `D`, `S`, `d` anzusprechen sind).
- Außerdem wird statt `int 0x80` (32 Bit) bei 64-Bit-Code die Instruktion `syscall` verwendet. Die Syscall-Nummern sind verschieden, wenn Sie `__NR_write` und `__NR_fork` verwenden, bekommen Sie aus der Include-Datei direkt die richtigen Nummern.

b) Schreiben Sie ein C-Programm, das
- mit `creat()` eine neue Datei (mit im Programm vorgegebenen Namen) erzeugt und öffnet,
- mit `write()` das Wort „Hallo\n“ in diese Datei schreibt,
- mit `close()` die neue Datei schließt.

```
// creat-write.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main () {
  char* filename = "output.txt";
  char* text = "Hallo\n";
  int fd = creat (filename, S_IRUSR | S_IWUSR);
  write (fd, text, strlen(text));
  // wir benutzen strlen und nicht sizeof, weil bei
  // sizeof noch das abschliessende Null-Byte mit
  // geschrieben wird.
  close (fd);
}
```

Verwenden Sie dafür zunächst die angegebenen Systemaufrufe und ersetzen Sie diese anschließend durch Aufrufe von `syscall()`. Welche Parameter Sie `creat()` übergeben müssen, verrät wieder die Manpage (`man 2 creat`). Tipp: `syscall()` erwartet immer genau vier Argumente. Benötigt Ihr Syscall weniger Argumente, dann „füllen Sie mit Nullen auf“.

```
// creat-write-syscall.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <asm/unistd.h>

int syscall (long rax, long rdi, long rsi, long rdx) {
  int result;
  asm (
    "syscall"
    : "=a" (result)
    : "a" (rax), "D" (rdi), "S" (rsi), "d" (rdx)
  );
  return result;
}

int main () {
  char* filename = "output.txt";
  char* text = "Hallo\n";
  int fd = syscall (__NR_creat, (long)filename, S_IRUSR | S_IWUSR, 0);
  syscall (__NR_write, fd, (long)text, strlen(text));
  syscall (__NR_close, fd, 0, 0);
}
```

4. Kopierprogramm

Schreiben Sie ein Programm `copy.c`, das zwei Dateinamen `quelle` und `ziel` definiert. Es soll die über `quelle` erreichbare Datei öffnen, eine Datei `ziel` erzeugen und dann byteweise den Inhalt von `quelle` lesen und nach `ziel` schreiben (im Ergebnis also die Datei kopieren).

Prüfen Sie bei allen Schritten auf mögliche Fehler, geben Sie – bei Auftreten eines Fehlers – eine passende Meldung aus und brechen Sie das Programm dann ab.

Zunächst die Lösung ohne Fehlerprüfungen:

```
// copy.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main () {
  char* source = "source.txt";
  char* target = "target.txt";
  char c;

  int fd1 = open (source, O_RDONLY);
  int fd2 = creat (target, S_IRUSR | S_IWUSR);

  while ( read (fd1, &c, 1) != 0 )  write (fd2, &c, 1);

  close (fd1);
  close (fd2);

  return 0;
};
```

Mit Prüfungen wird es etwas komplexer:

```
// copy-errorcheck.c
#include <sys/stat.h>  // fuer S_IRUSR usw.
#include <fcntl.h>     // fuer O_RDONLY usw.
#include <errno.h>     // fuer perror()
#include <stdlib.h>    // fuer exit()
#include <stdio.h>     // fuer printf()
#define true 1
```

```
int main () {
  char* source = "source.txt";
  char* target = "target.txt";
  char c;
  int retval;   // benutzen wir mit close()
  int fd1, fd2;

  fd1 = open (source, O_RDONLY);
  if (fd1==-1) {
    perror ("copy: while opening source file");
    printf ("errno = %d\n", errno);
    exit (-1);
  }

  fd2 = creat (target, S_IRUSR | S_IWUSR);
  if (fd2==-1) {
    perror ("copy: while opening target file");
    printf ("errno = %d\n", errno);
    exit (-1);
  }

  while ( true ) {
    retval = read (fd1, &c, 1);
    if (retval==0) break;  // EOF Eingabedatei, Schleife verlassen
    if (retval==-1) {
      perror ("copy: while reading from source file");
      printf ("errno = %d\n", errno);
      exit (-1);
    };

    retval = write (fd2, &c, 1);
    if (retval==-1) {
      perror ("copy: while writing to target file");
      printf ("errno = %d\n", errno);
      exit (-1);
    };
  };
  retval = close (fd1);
  if (retval==-1) {
    perror ("copy: while closing source file");
    printf ("errno = %d\n", errno);
    exit (-1);
  }

  retval = close (fd2);
  if (retval==-1) {
    perror ("copy: while closing source file");
    printf ("errno = %d\n", errno);
    exit (-1);
  }

  return 0;
};
```