



3. System Calls: fork und exec

In Aufgabe 2 (letzte Übung) haben Sie `fork` und `exec` bereits verwendet. Betrachten Sie das folgende Programm `forkexec.c` (das Sie auch in `sp-ss2015-ue03.tar.gz` auf der Webseite finden):

```
/* forkexec.c, Hans-Georg Esser, Systemprogrammierung */

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

int main () {
    pid_t pid1, pid2, pid3;

    printf ("Start\n");
    pid1 = fork();          /* fork 1 */
    printf ("Nach fork 1\n");
    if (pid1==0) {
        execlp ("echo","echo","echo-test",NULL);
        printf ("Nach exec\n");
    } else {
        pid2 = fork();     /* fork 2 */
        printf ("Nach fork 2\n");
        pid3 = fork();     /* fork 3 */
        if ((pid2 == 0) && (pid3 ==0)) {
            printf ("pid1 != 0, pid2=pid3=0\n");
        };
    };
    printf ("Prozessende; pid1=%d, pid2=%d, pid3=%d\n", pid1, pid2, pid3);
    if ((pid1 != 0) && (pid2 != 0) && (pid3 != 0)) {
        /* urspruenglicher Prozess */
        sleep(1);
        printf ("Programmende\n");
    };
    return 0;
}
```

- 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?
- b) Übersetzen Sie das Programm (`gcc -o forkexec forkexec.c`) und starten Sie es; überprüfen Sie Ihre Vermutung aus Aufgabe a).

4. System Calls in Assembler und C

In Assembler-Programmen rufen Sie (auf älteren 32-Bit-Linux-Systemen) System Calls über den Software-Interrupt `int 0x80` auf und füllen vorher die Register EAX bis EDX mit passenden Werten. Da kann man sich die Argumentregister leicht merken (EAX: Syscallnummer, dann in EBX, ECX, EDX die Argumente); bei 64-Bit-Linux ist das leider etwas unübersichtlicher (RAX, RDI, RSI, RDX; dafür heißt der Befehl zum Auslösen des System Call `syscall`).

FreeBSD und viele andere Unix-Systeme verwenden (bis auf die Syscallnummer) den Stack für Argumente:

Linux (32 bit, klassisch)

```
section .text
global _start          ; fuer den Linker (ld)

_start:                ; fuer Linker (wo gehts los)

    mov     edx, len    ; Nachrichtenlaenge
    mov     ecx, msg    ; Adresse der Nachricht
    mov     ebx, 1     ; file descriptor (1=stdout)
    mov     eax, 4     ; Syscall-Nr. (__NR_write)
    int     0x80      ; Syscall ausfuehren

    mov     eax, 1     ; Syscall-Nr. (__NR_exit)
    int     0x80      ; Syscall ausfuehren
```

```
section .data
msg     db     'Hallo Welt!', 0xa    ; Text
len     equ   $ - msg                ; Laenge
```

FreeBSD

```
section .text
global _start          ; fuer den Linker (ld)

_syscall:
    int     0x80      ; system call
    ret

_start:                ; fuer Linker (wo gehts los)

    push   dword len  ; Nachrichtenlaenge
    push   dword msg  ; Adresse der Nachricht
    push   dword 1    ; file descriptor (1=stdout)
    mov    eax, 0x4   ; Syscall-Nr. (sys_write)
    call   _syscall  ; Syscall ausfuehren
    add   esp, 12    ; Stack aufraeumen
                    ; (3 Argumente, Laenge 4)

    push   dword 0    ; exit code
    mov    eax, 0x1   ; Syscall-Nr. (sys_exit)
    call   _syscall  ; Syscall ausfuehren
                    ; nach exit() nicht aufr.
```

```
section .data
msg     db     "Hallo Welt!", 0xa    ; Text
len     equ   $ - msg                ; Laenge
```

Linux (64 bit)

```
BITS 64
section .text
global _start          ; fuer den Linker (ld)

_start:                ; fuer Linker (wo gehts los)

    mov     rdx, len    ; Nachrichtenlaenge
    mov     rsi, msg    ; Adresse der Nachricht
    mov     rdi, 1     ; file descriptor (1=stdout)
    mov     rax, 1     ; Syscall-Nr. (__NR_write: 1)
    syscall                    ; Syscall ausfuehren

    mov     rax, 60    ; Syscall-Nr. (__NR_exit: 60)
    syscall                    ; Syscall ausfuehren
```

```
section .data
msg     db     'Hallo Welt!', 0xa    ; Text
len     equ   $ - msg                ; Laenge
```

Übersetzung und Test auf 64-Bit-Linux (mit installiertem Assembler nasm):

```
$ nasm -f elf64 test.asm; ld -s -o test test.o
$ ./test
Hallo Welt!
```

(Die Systemcall-Nummern stehen auf einem 64-Bit-Linux in der Header-Datei `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`.) Die allgemein übliche Unix-Variante ist die von FreeBSD. Der Rückgabewert des Syscalls steht bei Linux in EAX (32 bit) bzw. RAX (64 bit). Die Linux-Variante können Sie auch in C-Programme übernehmen und definieren dafür folgende Inline-Assembler-Funktion:

```
int syscall (long a, long b, long c, long d) { // 64 Bit, z.B. im Praktikum
    int result;
    asm ( "syscall"      : "=a" (result)
          : "a" (a), "D" (b), "S" (c), "d" (d) );
    return result;
}
```

bzw.

```
int syscall (int a, int b, int c, int d) { // 32 Bit
    int result;
    asm ( "int $0x80"   : "=a" (result)
          : "a" (a), "b" (b), "c" (c), "d" (d) );
    return result;
}
```

Diese Funktion `syscall()` erwartet dann als erstes Argument die Syscall-Nummer (wie Sie sie in der Datei `/usr/include/asm/unistd_64.h` finden, eine Kopie liegt im Archiv). Anstelle von `exit(0)`; können Sie mit obiger Definition also auch `syscall(1,0,0,0)`; schreiben, um den aktuellen Prozess zu beenden.

a) Betrachten Sie das folgende Programm (`fork+write.c` im Aufgabenarchiv):

```
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();

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

Es verwendet die Systemaufrufe `fork()` und `write()`. Die Bedeutung der Argumente in `write()` entnehmen Sie der Manpage (`man 2 write`); das Argument 1 ist der File-Deskriptor (`fd`) für die Standardausgabe `stdout` (0: Standardeingabe `stdin`, 2: Standardfehlerausgabe `stderr`).

Ersetzen Sie im Programm die drei Aufrufe (`fork`, `write`, `write`) durch Aufrufe von `syscall()`. Die benötigten Syscall-Nummern finden Sie in der Datei `unistd_32.h`. Schreiben Sie nur auf, wodurch Sie die drei rot markierten Zeilen ersetzt haben. Prüfen Sie, ob Ihr verändertes Programm funktioniert. (Die Datei enthält bereits die Definition von `syscall()`.) Überprüfen Sie, dass Ihr Programm nach der Änderung noch genauso funktioniert wie vorher.

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.

In der Manpage zu `creat` finden Sie Hinweise auf die einzubindenden Header-Dateien und die nötigen Parameter.

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“.

5. 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.