



Code-Dateien finden Sie in `swf-sp-ws2023-ue02.tgz`, Download mit:

```
wget swf.hgesser.de/bs-sp/ws2023/prakt/swf-sp-ws2023-ue02.tgz
```

2. Mini-Shell

In dieser Aufgabe geht es darum, eine kleine **Shell** zu entwickeln, also ein Programm, das Kommandos entgegennimmt, interpretiert und ausführt.

Sie bearbeiten diese Aufgabe in mehreren Einzelschritten.

a) Geben Sie im Editor das folgende Miniprogramm ein und speichern Sie es als `shell01.c`:

```
#include <stdio.h>
#include <string.h>

int main () {
    char command[255];

    while (1) {
        printf ("Eingabe: ");
        fgets (command, sizeof(command), stdin);
        printf ("Eingabe: '%s', Laenge: %d \n", command, strlen(command));
    }
}
```

Wenn Sie das Programm kompilieren (`gcc -o shell01 shell01.c`) und dann ausführen (`./shell01`), sehen Sie, dass es nicht wie gewünscht arbeitet. (Um das laufende Programm abzubrechen, drücken Sie [Strg-C].)

b) Warum erscheint in der Ausgabe ein (unerwünschter) Zeilenumbruch?

c) Modifizieren Sie das Programm so, dass es nach dem `fgets()`-Aufruf den String verändert – das folgende `printf()`-Kommando soll den Zeilenumbruch nicht mehr ausgeben. Hier hilft Ihnen auch die Funktion `strlen()`. Die neue Version speichern Sie als `shell02.c` und übersetzen sie mit `gcc -o shell02 shell02.c`.

d) Was würde passieren, wenn Sie einen sehr langen Text (z. B. mit 300 Zeichen) eingeben? Um diese Frage (ohne Testen) beantworten zu können, lesen Sie mit `man fgets` die Manpage zu `fgets()`. Bis zu wie vielen Zeichen funktioniert das Programm korrekt?

e) Ihr Programm soll nun die Eingabe in einzelne „Worte“ unterteilen, Worttrenner sind so genannte Whitespace-Zeichen (Leerzeichen und Tabulator `\t`). Lesen Sie zunächst die Manpage der Funktion `strtok()` (mit `man strtok`). Die Funktion muss mehrfach aufgerufen werden, um alle Teile der Eingabe zu verarbeiten. Beim ersten Aufruf übergeben Sie ihr (im ersten Argument) den zu bearbeitenden String, bei allen folgenden Aufrufen einen Nullzeiger (`NULL`). Außerdem müssen Sie die Worttrenner (als zweites Argument) angeben.

Kopieren Sie das Programm nach `shell03.c` und fügen Sie in dieser Version eine Variable `seps` (Separatoren) sowie einen Char-Pointer `part` hinzu:

```
char seps[] = " \t";
char *part;
```

Testen Sie zunächst die Funktion `strtok()`, indem Sie diese zweimal aufrufen (`part = strtok (...);`) und nach jedem Aufruf den String `part` ausgeben. Der eingegebene String sollte dabei mindestens zwei durch Leerzeichen getrennte Worte enthalten.

Ersetzen Sie nun die zwei Aufrufe durch eine While-Schleife. Wenn es keine Wortbestandteile

mehr gibt, gibt `strtok()` einen Nullzeiger zurück. Ihr Programm soll jetzt alle Teile der Eingabe separat ausgeben.

f) Im nächsten Schritt streichen Sie die Ausgaben und speichern stattdessen die einzelnen Teile in einem Array von Strings (`char *args[10]`). Damit schaffen Sie Platz für zehn Pointer. Erzeugen Sie dazu zunächst eine Kopie `shell04.c`, in der Sie dies implementieren.

Sie müssen für die einzelnen Bestandteile (im Folgenden „Argumente“ genannt) keinen Speicherplatz reservieren, weil `strtok()` die vorhandene Eingabe verändert und jeweils eines der Trennzeichen durch die String-Terminierung (`\0`) ersetzt. Platz brauchen Sie also nur für die Zeiger auf die einzelnen Argumente. Programmieren Sie die Schleife so, dass maximal neun Argumente akzeptiert werden. (Den 10. Pointer brauchen Sie für Teilaufgabe g.) Überzählige Argumente soll Ihr Programm verwerfen (ignorieren).

Wenn Sie die Zerlegung beendet haben, geben Sie in einer Schleife die Elemente von `args` aus, das soll so aussehen:

```
int i;
printf ("Zerlegung: \n");
for ( i=0; i<no_args; i++) {
    printf ("Argument %d: %s \n", i, args[i]);
}
```

g) Jetzt interpretieren wir die Eingabe als Kommando. In `args[0]` steht also der Name eines Programms, und in `args[1]`, `args[2]`, `args[3]` usw. stehen die Argumente. Sie sollen in dieser Teilaufgabe, für die Sie zunächst eine Kopie des Programms (`shell05.c`) erzeugen, die Funktion `execvp()` verwenden. Wenn Sie einen Blick in die Manpage werfen, finden Sie die folgende Beschreibung:

```
int
execvp(const char *file, char *const argv[]);
```

Die Funktion erwartet also zwei Argumente: einen String (`char*`) und ein Array von Strings (`char*[]`). Das erste Argument ist `args[0]`, das zweite einfach `args` – in den vorangehenden Teilaufgaben haben Sie darauf hingearbeitet, dass Ihre Datenstruktur genau zur Syntax von `execvp` passt. Allerdings muss die Liste der Pointer noch mit einem Nullzeiger abgeschlossen werden: Den müssen Sie ergänzen (und darum durften Sie in Teilaufgabe f nur neun der zehn Pointer für Argumente verwenden).

Probieren Sie es aus und testen Sie, was passiert, wenn Sie bei der Programmausführung `ls -l` eingeben. Testen Sie auch, was passiert, wenn Sie ein Kommando eingeben, das nicht gefunden werden kann.

h) Obwohl Sie in einer Schleife immer wieder eine neue Eingabe lesen, führt der erste (erfolgreiche!) Aufruf von `execvp()` dazu, dass Ihr Programm beendet wird. Das liegt daran, dass `execvp()` Ihr Programm durch das zu startende Programm ersetzt: Wenn letzteres fertig ist, endet der ganze Prozess. Das können Sie verhindern, indem Sie vorher mit der Funktion `fork()` einen neuen Prozess erzeugen.

Erstellen Sie eine Kopie (`shell06.c`) und ersetzen Sie darin den `execvp()`-Aufruf durch folgenden Block:

```
if ( fork() == 0 )
    // Kindprozess
    execvp ( /* ... Argumente aus Aufgabe g) ... */ );
else
    // Vaterprozess
    wait (NULL);
```

Testen Sie das Programm: Jetzt sollten Sie eine funktionierende Mini-Shell haben. Lesen Sie in den Manpages zu `fork()` (`man fork`) und `wait()` (`man waitpid`) nach, was die Kommandos bewirken.

3. System Calls: fork und exec

In Aufgabe 2 haben Sie fork und exec bereits verwendet. Betrachten Sie das folgende Programm forkexec.c (das Sie auch in swf-sp-ws2023-ue02.tgz 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
```

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
    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;
}
```

FreeBSD

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

_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
```

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=strlen(vater);
    char sohn[]="Ich bin der Sohn.\n";
    int slen=strlen(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.