



Vorbereitung

- Die Dateien zum heutigen Praktikumstermin finden Sie auf der Vorlesungsseite:
wget <http://hm.hgesser.de/bs-ss2009/prakt03.tgz>
- Entpacken Sie das Archiv mit tar (tar xzf prakt03.tgz) und wechseln Sie mit cd prakt03 in das neue Unterverzeichnis.

1. System Calls in Assembler und C

In Assembler-Programmen rufen Sie System Calls über den Software-Interrupt `int 0x80` auf:¹

| Linux | FreeBSD |
|---|--|
| <pre>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. (sys_write) int 0x80 ; Syscall ausfuehren mov eax, 1 ; Syscall-Nr. (sys_exit) int 0x80 ; Syscall ausfuehren section .data msg db 'Hallo Welt!', 0xa ; Text len equ \$ - msg ; Laenge</pre> | <pre>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</pre> |

Die allgemein übliche Unix-Variante ist die von FreeBSD: Argumente in umgekehrter Reihenfolge auf den Stack pushen, dann die Syscall-Nummer in Register EAX schreiben und einen Syscall-Interrupt auslösen (hier: `int 0x80`). Linux verwendet stattdessen die Register EBX, ECX, EDX, ESI, EDI und EBP für bis zu sechs Argumente (und ebenfalls den Software-Interrupt `0x80`). Der Rückgabewert des Syscalls steht in EAX.

Die Linux-Variante können Sie auch in C-Programme übernehmen und definieren dafür folgende Inline-Assembler-Funktion:

```
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;
}
```

¹ Quelle dieser Listings: <http://asm.sourceforge.net/intro/hello.html> (übersetzt) - Syntax für Assembler nasm geeignet



Diese Funktion `syscall()` erwartet dann als erstes Argument die Syscall-Nummer (wie Sie sie in der Datei `/usr/include/asm/unistd_32.h` finden, eine Kopie der Datei 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`).

[ **Lösungsblatt**] 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.

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“.

[ **Lösungsblatt**] Geben Sie hier nur die wesentlichen Aufrufe an (keine Funktionsdefinition für `main()` etc.).



2. Signal-Ping-Pong (Signal-Handler statt Interrupt-Handler)

Treiberprogrammierung und eigene Interrupt-Behandlungsroutinen würden für das Praktikum etwas zu weit führen, auf Benutzerebene können Sie aber mit Signal-Handlern ein ähnliches Konzept umsetzen.

Wie Sie bereits wissen, können Sie mit `kill` Signale an Prozesse schicken; eine Liste der verfügbaren Signale liefert `kill -l`. Dort finden Sie auch die Signale `SIGUSR1` (10) und `SIGUSR2` (12), die keine vordefinierte Funktion haben: Über einen Signal-Handler können Sie selbst bestimmen, was Ihr Programm tun soll, wenn es diese Signale erhält.

Die Funktion `signal()` erwartet eine Signalnummer (dafür gibt es vordefinierte Konstanten, z. B. `SIGINT` oder `SIGUSR1`) als erstes und den Namen einer Signal-Handler-Funktion als zweites Argument. Ein Minimalprogramm `signal0.c` (Archiv!), das dieses Feature nutzt, sieht so aus:

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

// Signal-Handler
void handle_SIGUSR1(int sig_num) {
    printf("\n--SIGUSR1 erhalten!--\n");
    fflush(stdout);
}

// Hauptprogramm
int main() {
    // Signal-Handler fuer SIGUSR1 anmelden
    signal (SIGUSR1, handle_SIGUSR1);

    // Hauptprogramm: Punkte ausgeben...
    for ( ;; ) {
        sleep(1);
        printf("."); fflush(stdout);
    }
}
```

- Kompilieren Sie das Programm (`signal0.c`), lassen Sie es laufen, und schicken Sie ihm von einem anderen Terminalfenster aus über den Befehl `killall -USR1 signal0` das USR1-Signal.
[ Lösungsblatt] Was passiert dann? Was passiert, wenn Sie das Signal mehrmals schicken?
- Es gibt zwei Standard-Signal-Handler, die Sie im `signal()`-Aufruf (anstelle einer von Ihnen definierten Funktion) angeben können, `SIG_IGN` (ignorieren) und `SIG_DFL` (Default-Verhalten). Ändern Sie das Programm aus Aufgabe a) so ab, dass es die Tastenkombination [Strg-C] abfängt, die das Signal `SIGINT` auslöst. Dabei soll der Signal-Handler nur zweimal anspringen und eine Warnung ausgeben. Nach dem zweiten Aufruf soll er sich selbst deaktivieren. Ein drittes Drücken von [Strg-C] bricht das Programm also wieder wie gewohnt ab. Sie werden dazu eine globale Variable benötigen, in der Sie mitzählen, wie oft der Signal-Handler schon aufgerufen wurde.
[ Lösungsblatt] Schreiben Sie die wichtigsten Teile des Programms auf.
- (Zusatzaufgabe) Schreiben Sie ein Programm, das zunächst via `fork()` einen Kindprozess erzeugt. Die beiden Prozesse sollen einander (abwechselnd) gegenseitig Signale schicken. Erhält der Vater ein Signal, schreibt er „Pong“ und schickt ein Signal zurück; erhält der Sohn eines, schreibt er „Ping“ (und schickt wieder ein Signal zurück). Ansonsten geben beide Prozesse einfach Punkte aus, wie die bisherigen Beispielprogramme.
[ Lösungsblatt] Dieses Programm schicken Sie mir bitte per E-Mail (an die Adresse `hans-georg.esser@hm.edu`) zu.



Lösungsblatt – Übung 3

Punkte:

| | Name | Matrikelnummer |
|---------------------|------|----------------|
| Teilnehmer 1 | | |
| Teilnehmer 2 | | |