1. System Calls (Assembler)

Auf aktuellen 64-bittigen PCs (`x86_64`) finden Sie die System-Call-Nummern für die Syscalls `write` und `exit` wie folgt heraus:

```
$ egrep -w '__NR_(write|exit)' /usr/include/x86_64-linux-gnu/asm/unistd_64.h
#define __NR_write 1
#define __NR_exit 60
```

Sie können einen System Call durchführen, indem Sie die Register `RAX`, `RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9`[1] mit der System-Call-Nummer und den Argumenten für den System Call füllen (in dieser Reihenfolge – und nur so viele, wie Sie brauchen). (Auf 32-Bit-PCs verwendet Linux die Register `EAX`, `EBX`, `ECX`, `EDX`, `EDI`, `ESI` und `EBP`.)

Eine (etwas ältere) Liste der `x86_64`-System-Calls mit Beschreibungen der zu füllenden Register finden Sie online.[2]

Das folgende Assembler-Programm gibt über den `write`-Syscall den Text „Hallo Welt!“ aus und beendet sich dann über den `exit`-Syscall:

(hello.asm)

```
global _start
_start:
       mov rax, 1               ; Syscall-Nummer: 1 = __NR_write

       ; C: write (1, msg, len);
       mov rdi, 1               ; file descriptor, 1 = stdout
       mov rsi, msg             ; Adresse des Strings
       mov rdx, len             ; Laenge des Strings
       syscall

       mov rax, 60              ; Syscall-Nummer: 60 = exit
       mov rdi, 0               ; Exit-Code: 0
       syscall

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

Register	Inhalt	Beispiel, `write`
`RAX`	Syscall-Nummer	`__NR_write = 1`
`RDI`	1. Argument	1 (stdout, file descriptor)
`RSI`	2. Argument	Adresse des Strings
`RDX`	3. Argument	Länge des Strings
`R10`	4. Argument	–
`R8`	5. Argument	–
`R9`	6. Argument	–

1 siehe https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf, Anhang A.2.1, S. 124.
2 https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

a) Laden Sie im Ubuntu-Docker-Container das Quellcode-Paket `bs1-ue04.zip` herunter, entpacken Sie es, und wechseln Sie in den Ordner `ue04`:

```
cd /realworld
wget swf.hgesser.de/bs-b1/prakt/bs1-ue04.zip
unzip bs1-ue04.zip
cd ue04
```

Das Beispielprogramm von oben finden Sie in `hello.asm`. Übersetzen Sie es in eine Objektdatei (`hello.o`), erstellen Sie daraus ein ausführbares Programm (`hello`), und starten Sie es:

```
nasm -f elf64 hello.asm
ld -o hello hello.o
./hello
```

(`./hello` mit vorangestelltem `./` ohne Leerzeichen). Das Programm sollte „Hallo Welt!“ ausgeben.

b) Verwenden Sie das Hilfsprogramm `strace`, um die System-Call-Aufrufe des Programms zu protokollieren:

```
strace -o hello.log ./hello
cat hello.log
```

c) Vergleichen Sie `hello.asm` mit der 32-Bit-Version `hello32.asm`. Sie können aus dieser Quellcode-Datei mit den folgenden Befehlen ein 32-Bit-Binary erzeugen und testen:

```
nasm -f elf32 hello32.asm
ld -m elf_i386 -o hello32 hello32.o
strace -o hello32.log ./hello32
```

(Das funktioniert, weil das mit Docker verwendete 64-Bit-Ubuntu in der Lage ist, 32-Bit-Anwendungen auszuführen.) Prüfen Sie mit `file`, dass es sich um unterschiedliche Programmformate handelt:

```
file hello hello32
```

2. System Calls (C)

Es ist nicht nötig, immer separate Assembler-Quellcode-Dateien zu erstellen, wenn Sie Syscalls direkt ausführen wollen; Sie können stattdessen Inline-Assembler und generische Syscall-Wrapper in C-Programmen verwenden. Die Funktion

```
int syscall3 (long syscallno, long arg1, long arg2, long arg3) {
  int result;
  asm (
    "syscall"
    : "=a" (result)
    : "a" (syscallno), "D" (arg1), "S" (arg2), "d" (arg3)
  );
  return result;
}
```

nutzt das Inline-Assembler-Feature des GNU-C-Compilers, dessen Syntax gewöhnungsbedürftig ist. Die drei Zeilen hinter `asm(` sind am besten in der Reihenfolge 3. Zeile; 1. Zeile; 2. Zeile zu lesen und haben die folgenden Bedeutungen:

- 3. Zeile:

  `"a" (syscallno), "D" (arg1), "S" (arg2), "d" (arg3)`

  Vorbereitung; schreibe die vier Werte `syscallno`, `arg1`, `arg2` und `arg3` in die vier Register `RAX` (`"a"`), `RDI` (`"D"`), `RSI` (`"S"`) und `RDX` (`"d"`).

- 1. Zeile:

  `"syscall"`

  Befehle; führe die Assembler-Befehle aus, die im String stehen; hier einfach `syscall`, also den Spezialbefehl, der in den Kernel Mode springt und den generischen System Call Handler im Kernel aufruft

- 2. Zeile:

  `"=a" (result)`

  Nachbereitung: sichere den Inhalt des Registers `RAX` (`"=a"`) in der Variablen `result`.

In den Strings aus der zweiten und dritten Zeile stehen also `a`, `D`, `S`, `d` immer für die Register `RAX`, `RDI`, `RSI` und `RDX`, wobei ein `=`-Präfix anzeigt, dass der Registerinhalt in eine Variable geschrieben wird.

Ein C-Programm, das (wie oben im Assembler-Programm) „Hallo Welt!“ nur mit Hilfe der System Calls ausführt, könnte damit so aussehen:

```
#define __NR_exit  60
#define __NR_write 1

#define stdout     1
#define ENOERR     0

int main () {
  char msg[] = "Hallo Welt!\n";
  int len = sizeof (msg);
  syscall3 (__NR_write, stdout, (long)msg, len);
  syscall3 (__NR_exit,  ENOERR, 0,         0);
}
```

(Oberhalb der `main`-Funktion muss außerdem die Implementierung von `syscall3` eingefügt werden. Mit `(long)msg` wird der Pointer `msg` in einen Integer-Wert „gecastet“, also typ-konvertiert.)

Für alle bekannten System Calls finden Sie in `/usr/include/x86_64-linux-gnu/asm/unistd_64.h` die Definitionen passender `__NR_`*name*-Makros, so dass Sie die beiden Definitionen von `__NR_exit` und `__NR_write` auch weglassen können, wenn Sie stattdessen die Zeile

```
#include <asm/unistd_64.h>
```

ergänzen.

a) Im Ordner mit den Assembler-Quellcode-Dateien (aus Aufgabe 1) finden Sie auch das C-Programm `hello-world.c`. Übersetzen Sie es mit

```
gcc -o hello-world hello-world.c
```

und starten Sie das erzeugte Programm mit

```
./hello-world
```

(mit vorangestelltem `./` ohne Leerzeichen).

b) Beim zweiten `syscall3`-Aufruf werden mehr Parameter als nötig übergeben, der Syscall erwartet nur in *EBX* den Rückgabewert. `syscall3` braucht aber vier Argumente (die Syscall-Nummer sowie drei Parameter für den Syscall; daher der Name `syscall3`). Ergänzen Sie Funktionen `syscall1()` und `syscall2()`, die weniger Parameter erwarten und auch im `asm`-Aufruf weniger Register mit Inhalt füllen – wie Sie `syscall3()` dafür anpassen müssen, sollte sich leicht ergeben.

Ersetzen Sie dann den zweiten Aufruf von `syscall3` durch einen geeigneteren Aufruf, der auf sinnlose Parameter verzichtet.

c) Anstelle der `syscall?`-Funktionen würde man beim Programmieren normal direkt die spezifischen Syscall-Wrapper `write()` und `exit()` aus den Standardbibliotheken (`unistd.h` und `stdlib.h`) verwenden, im Beispiel also

```
int main () {
  char msg[] = "Hallo Welt!\n";
  int len = sizeof (msg);
  write (stdout, msg, len);
  exit  (ENOERR);
}
```

Hier sehen Sie die direkte Gegenüberstellung der beiden Varianten in vollständigen, kompilierbaren Quellcode-Dateien:

```
// Version mit syscall3

#define __NR_exit  1
#define __NR_write 4

int syscall3 (long syscallno, long arg1, long arg2,
              long arg3) {
  int result;
  asm (
    "syscall"
    : "=a" (result)
    : "a" (syscallno), "D" (arg1), "S" (arg2),
      "d" (arg3)
  );
  return result;
}

#define stdout     1
#define ENOERR     0

int main () {
  char msg[] = "Hallo Welt!\n";
  int len = sizeof (msg);
  syscall3 (__NR_write, stdout, (long)msg, len);
  syscall3 (__NR_exit,  ENOERR, 0,         0);
}
```

```
// Version mit write und exit

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

#define stdout     1
#define ENOERR     0

int main () {
  char msg[] = "Hallo Welt!\n";
  int len = sizeof (msg);
  write (stdout, msg, len);
  exit  (ENOERR);
}
```

d) Eine solche Umwandlung ist für alle Bibliotheksfunktionen möglich, die direkte Wrapper für System Calls sind. In vielen Fällen finden Sie zu einem Syscall-Namen `__NR_`*name* eine passende Funktion *name*`()` und können aus der zugehörigen Man-Page (Handbuchseite) mit `man 2` *name* die Aufrufparameter entnehmen. (Beispiel: `__NR_read` / `read()` / `man 2 read`).

Auf der folgenden Seite sehen Sie das Programm `copy.c`, das zwei Argumente (Quelle und Ziel) erwartet und dann versucht, die Quelldatei auf die Zieldatei zu kopieren. Alle aufgerufenen Funktionen (mit Ausnahme von `sprintf` und `strlen`) sind Syscall-Wrapper.

Erstellen Sie eine Kopie `copy-syscall.c` und ersetzen Sie darin alle Wrapper-Aufrufe durch Aufrufe von `syscall3`, `syscall2` und `syscall1`. Beachten Sie hier, dass `open()` einmal mit zwei und einmal mit drei Parametern aufgerufen wird!

Testen Sie, dass Ihr angepasstes Programm funktioniert. Achten Sie auch auf angemahnte implizite Typ-Konvertierungen und ergänzen Sie ggf. Cast-Operatoren, z. B. `(long)buf` statt `buf`.

Erläuterung zu Bestandteilen des Programms:

- `buf[1024]`: 1024 Byte langer Puffer, der zum Formatieren der Strings (in den `sprintf`-Aufrufen) und als Datenpuffer für die Kopieroperation verwendet wird
- `sprintf`: spezielle Variante der `printf`-Funktion, die einen zusätzlichen ersten Parameter `buf` erhält und die formatierte Ausgabe nicht auf die Standardausgabe, sondern in diesen Buffer `buf` schreibt.
- `argc`: Anzahl der Aufrufparameter des Programms. Der Programmname zählt mit.
- `argv[]`: Array, das die Parameter enthält.
  `argv[0]` = Programmname
  `argv[1]` = 1. Parameter usw.
- `O_RDONLY`: Modus zum Öffnen; hier: nur-lesen
- `O_WRONLY | O_CREAT | O_TRUNC`: Modus zum Öffnen; hier: nur-schreiben, erzeugen (falls nicht vorhanden), auf 0 Bytes zurücksetzen (falls nicht leer)
- `0644`: Zugriffsrechte für die erstellte Datei (führende `0` → Oktalzahl; 644 = `rw-r--r--`, also les- und schreibbar für Dateibesitzer:in, lesbar für alle anderen)
- `strlen`: gibt Länge eines Strings zurück; wird benötigt, weil man `write` die Zahl zu schreibender Bytes übergeben muss
- `r_fd`, `w_fd`: File Descriptors für die beiden Dateien. Werden beim Öffnen erzeugt und dann für alle Operationen wie Lesen, Schreiben, Schließen benötigt.

(copy.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main (int argc, char **argv) {
  char buf[1024];

  // Anzahl Parameter pruefen
  if (argc != 3) {
    sprintf (buf, "%s: Datei kopieren mit %s quelle ziel\n",
             argv[0], argv[0]);
    write (1, buf, strlen(buf));
    exit (1);
  }

  // Quelle zum Lesen oeffnen
  int r_fd = open (argv[1], O_RDONLY);
  if (r_fd < 0) {
    sprintf (buf, "%s: Kann Datei %s nicht zum Lesen oeffnen.\n",
             argv[0], argv[1]);
    write (1, buf, strlen(buf));
    exit (1);
  }

  // Ziel zum Schreiben oeffnen
  int w_fd = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (w_fd < 0) {
    sprintf (buf, "%s: Kann Datei %s nicht zum Schreiben oeffnen.\n",
             argv[0], argv[2]);
    write (1, buf, strlen(buf));
    exit (1);
  }

  // Kopieren
  int bytes = 0; int res;
  for (;;) {
    res = read (r_fd, buf, 1024);     // Hinweis: Fehlerbehandlung zu
    write (w_fd, buf, res);           // read/write fehlt hier...
    bytes = bytes + res;
    if (res < 1024) break;
  }

  // Dateien schliessen
  close (r_fd);
  close (w_fd);

  // Ergebnis
  sprintf (buf, "%s: %d Bytes kopiert.\n",
           argv[0], bytes);
  write (1, buf, strlen(buf));
  exit (0);   // Error Code 0 = alles gut.
}
```