Fakultät für Informatik und Mathematik
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN MÜNCHEN

Betriebssysteme, SoSe 2020

Praktische Übung 1: C Grundlagen, Tools und System Calls

Ziele des Labortermins

- Grundlagen von Betriebssystemen
- Benutzung der Virtuellen Maschine
- C-Grundlagen wiederholen, Übersetzungs-/Build-Prozess wiederholen
- System Calls verwenden und nachvollziehen

Im Rahmen der praktischen Übungen werden Sie in einer Linux-basierten virtuellen Maschine. Es wird vorrausgesetzt, dass Sie die Grundlagen im Umgang mit einer Linux Shell kennen. Darüber hinaus basieren die praktischen Übungen auf C Quelltexten, die teilweise oder vollständig erstellt werden müssen. Grundlagen von C und dem Übersetzungs- bzw. Build-Prozess sind daher ebenfalls notwendig.

Beide Kenntnisse können sie auch "on-the-fly" erlernen. Zur Unterstützung empfehlen sich folgende Online-Tutorials (die Links finden Sie auch in Moodle):

- `https://de.wikibooks.org/wiki/C-Programmierung`
- `https://www.learn-c.org/`
- `https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html`

Aufgabe 1.1: Grundlagen von Betriebssystemen

a) Nennen Sie zwei Funktionen die ein Betriebssystem erfüllt in Hinsicht auf seine Rolle als Mittler zwischen Programmen und Resourcen.

b) Benennen Sie die zwei zentralen CPU Modi. Wie unterscheiden Sie sich in Hinblick auf die Möglichkeiten auf Hardware zuzugreifen?

c) Benennen und beschreiben Sie die üblichen Komponenten eines Betriebssystems.

d) Was ist der zentrale Unterschied zwischen einem monolithischen Kernel und einem Microkernel?

e) Mit welcher Methode kann ein Programm im Userspace den Kernel aufrufen?

Aufgabe 1.2: Virtuelle Maschine starten und benutzen (praktisch)

Im Rahmen der Übung verwenden Sie eine Virtuelle Maschine (VM) mit Linux. Es handelt sich dabei um die Variante "Ubuntu", die eine beliebte Linux-*Distribution* ist. Mehr Details finden Sie in Moodle. Natürlich können Sie auch ihr eigenes System verwenden.

Aufgabe 1.3: Einfaches C-Programm übersetzen (praktisch)

Als erstes Übersetzen Sie ein einfaches "Hello World"-Programm indem Sie den Compiler verwenden um aus dem Quelltext ein Programm zu erstellen.

a) Navigieren Sie in den Ordner zum ersten Versuch und von da zu dieser Aufgabe:

```
cd 01
cd hello
```

In der Virtuellen Maschine ist Visual Studio Code als empfohlener Editor installiert. Insofern Sie sich auskennen und eine Präferenz haben, können Sie selbstverständlich jeden Editor Ihrer Wahl installieren und verwenden.

b) Schauen Sie den Quelltext in VS Code an:

```
code hello.c
```

Der Quelltext sollte Ihnen bekannt vorkommen und leicht nachvollziehbar sein.

c) Übersetzen Sie den Quelltext mit dem gcc-Compiler in ein lauffähiges Programm und führen es aus:

```
gcc -o hello hello.c
./hello
```

Aufgabe 1.4: Programm aus mehreren C Dateien übersetzen (praktisch)

Im Normalfall bestehen Programme aus sehr vielen verschiedenen Dateien. Neben C-Quelltexten spielen dabei sogenannte Header-Dateien eine wichtige Rolle. Im folgenden übersetzen Sie ein Programm aus mehreren Dateien.

a) Schauen Sie sich die Dateien in `01/hellomulti` an.

b) Sie können ein lauffähiges Programm erstellen, indem Sie wie oben den Compiler aufrufen, aber diesmal mit mehreren Eingabedateien:

```
gcc -o hello hello.c main.c
./hello
```

c) Mit steigender Zahl der Quelltext-Dateien ist das weniger praktikabel und dauert außerdem sehr lang – selbst wenn nur eine Datei geändert wurde, werden alle Dateien neu übersetzt. Daher übersetzt man die einzelnen Quelldateien separat in "Objektdateien" und "linkt" diese am Ende zusammen:

```
gcc -o hello.o -c hello.c
gcc -o main.o -c main.c
gcc -o hello hello.o main.o
./hello
```

Sie brauchen nun nur noch diejenigen Quelldateien zu übersetzen, die sich auch geändert haben. Auch das wird schnell unpraktikabel, weshalb es *Makefiles* gibt. Das sind Dateien die die Anweisungen enthalten um ein Programm zu bauen. Diese Makefiles bilden die Abhängigkeiten zwischen Quelldateien, Objektateien und ihrem Programm ab. Sie finden ein Beispiel in dem Ordner, dass kurz erklärt werden soll:

```
hello: main.o hello.o
    gcc -o main.o main.o hello.o

main.o: main.c hello.h
    gcc -o main.o -c main.c

hello.o: hello.c hello.h
    gcc -o hello.o -c hello.c
```

In Zeile 1 finde Sie eine sogenannte Regel. Jede Regel definiert eine Ausgabedatei (vor dem Doppelpunkt) und von welchen Dateien diese abhängt (nach dem Doppelpunkt). Diese Abhängigkeit hat zur folge, dass wenn sich eine Datei rechts vom Doppelpunkt ändert, die Ausgabedatei neu erstellt wird. Die nachfolgende(n) Zeile(n), hier Zeile 2 definieren dann wie die Ausgabedatei in diesem Fall neu erstellt wird. Sie finden hier den Befehl von oben. Beachten Sie, dass die Kommandos eingerückt sind. Die weiteren Zeilen erklären sich von selbst.

d) Übersetzen Sie die Dateien mit Hilfe des Makefiles:

```
make
./hello
```

e) (optional) Vereinfachen und generalisieren Sie das Makefile, insbesondere mithilfe von `makedepend` und den Makros `$@`, `$^` und `$<`.

Aufgabe 1.5: System Calls (praktisch)

In dieser Aufgabe erstellen Sie selbständig ein kleines Programm, das eine Datei erstellt und öffnet, und schreiben "Hello World!" in diese Datei. Im Anschluß schauen Sie sich mit Unterstützung von Hilfsprogrammen die Ausführung der System Calls an. Erstellen Sie das Programm `hello` in dem Ordner `01/syscall`.

Hinweise:

- Zur Lösung der Aufgabe verwenden Sie bitte die Funktionen `open`, `write` und `close`, die direkt auf System Calls mappen.
- Die effizienteste Methode (neben Google bzw. die Suchmaschine Ihrer Wahl) um die Funktionen zu verstehen, sind die sogenannten "man pages", die Sie in der Konsole lesen können, zum Beispiel:

  ```
  man 2 open
  man 2 write
  man 2 close
  man 2 syscalls
  ```

  Hinweis: Die Zahl 2 bezeichnet als Parameter dabei die Sektion in der die Funktion auftaucht im Manual (open, write und close tauchen öfter auf). Sektion 2 sind System Calls.

a) Schreiben Sie das C Programm, das die Datei `hello.txt` öffnet (und anlegt falls sie nicht existiert) und darin den String "Hello World!
n" (vom Typ `char*`) abspeichert und die Datei wieder schließt. Hinweis: `write` kann den String direkt als Parameter verarbeiten, die Größe können Sie manuell ablesen. Übersetzen und testen Sie das Programm. *Tipp: open benötigt zwei flag parameter, die Sie miteinander "ver-oder-n" (`|`) und Sie sollten den mode zum Beispiel als `S_IRUSR | S_IWUSR` setzen.*

b) Das Programm `strace` hilft Ihnen einen System Call Trace anzeigen zu lassen. Es zeigt alle Aufrufe des Kernels von Ihrem Programm aus an. Sie rufen das strace mit Ihrem Programm als Parameter auf, zum Beispiel:

```
strace -e trace=openat,write,close ./hello
```

Interpretieren Sie die Ausgabe.

c) Im letzten Schritt wollen wir das *Linux Tracing Tool Next Generation* (*LTTng*) Framework verwenden um uns den zeitlichen Ablauf im Kernel anzuschauen. Die Befehle zur Durchführung der Tracing-Sitzung sind in dem Skript `lttng.sh` zu finden, in späteren Versuchen schauen wir uns diese Befehle genauer an.

Führen Sie das Skript aus:

```
./lttng.sh ./hello
```

Das Ergebnis können Sie mit dem Tool *lttng-scope* betrachten. Starten Sie es mit dem folgenden Icon:

Im Tool wählen Sie "File" und "New Project from Existing Trace(s)...". Im Dateidialog wählen Sie `01/syscalls/trace/kernel` aus. Bestätigen Sie den nächsten Dialog ("Trace Project Setup"). Vollziehen Sie die angezeigte Auswertung nach.

d) Im letzten Schritt rufen Sie den System Call direkt mit dem Maschinenbefehle `syscall` auf. Sie brauchen keine eigene Assembler-Datei dafür anzulegen, sondern können das sogenannte "Inline-Assembly" verwenden um direkt in C Assembler-Befehle auszuführen.

Wie in der Vorlesung vorgestellt werden neben dem Befehl zum System Call selbst die richtigen Register mit den Parametern und Rückgabewerten befüllt. Im Inline-Assembly müssen Sie die Befehle um die Daten in die richten Register zu bekommen nicht ausführen, sondern können die Zielregister und die Daten einfach angeben.

Ersetzen Sie die `write` API-Funktion durch Inline-Assembly. Im folgenden finden Sie das Template dafür. Sie müssen noch die richtigen Parameter und Rückgabewerte angeben:

```
asm volatile
(
    "syscall"
    : "=a" (..)
    : "a"(..), "D"(..), "S"(..), "d"(..)
    : "cc", "rcx", "r11", "memory"
);
```

Das erste Element ist der Assembler-Befehl den Sie ausführen. Das zweite Element (`=a`) gibt an, dass der Rückgabewert des System Calls über das `rax` Register übergeben wird. In der Klammer müssen Sie Ihre Variable dafür angeben. Das nächste Element gibt die System Call ID und die Parameter an. Dabei landet die System Call ID in dem register `rax` und die Parameter in den Registern `rdi` ("D"),

`rsi` (“S”) und `rdx` (“d”). Das letzte Element sind die sogenannten “clobber”-Hinweise. Sie besagen welche Register (und Speicher) während des System Calls überschrieben werden und deshalb vom Compiler gesichert werden müssen.

Finden Sie für Linux die System Call ID für write und die Reihenfolge der Parameter heraus und füllen Sie die Lücken entsprechend.