# Praktikum Betriebssysteme – Übung 3

Die folgenden Aufgaben sollen einen ersten Einstieg in die systemnahe Entwicklung auf UNIXoiden Betriebssystemen bieten. Dazu werden sehr einfache C-Programme geschrieben und mit Hilfe der GNU Compiler Collection (gcc) übersetzt. Vor allem die in der Vorlesung besprochenen *Systemaufrufe* sollen dabei eine Rolle spielen.

## Aufgabe 1 – Verständnisfragen

**a)** Programme, die auf UNIXoiden Systemen gestartet werden, haben standardmäßig drei Dateien geöffnet. Welche Dateien sind das und wofür werden diese benötigt/genutzt?
(*Hinweis: Im virtuellen Dateisystem* `/proc`*, das Informationen des Betriebssystemkerns exportiert, können Sie diese offenen „Dateien" eines jeden Prozesses (z.B. mit* `ls -al /proc/self/fd` *für den aktuellen Prozess) anzeigen lassen. Wie in der Vorlesung gesehen, verweisen diese auf das virtuelle Terminal (*`/dev/pts`*). Weitere Infos zu virtuellen Terminals finde Sie z.B. hier: https://www.baeldung.com/linux/dev-pts*).

**b)** Ist die folgende Aussage korrekt? → „Eine Funktion mit einem Parameter vom Typ void* kann jeden Pointer mit beliebigem Datentyp aufnehmen"

**c)** Welche der folgenden Funktionen können nur mit Unterstützung des Betriebssystems (vollständig oder teilweise im Kern mittels Systemaufruf) implementiert werden?

- `quicksort()` - Sortieren einer Eingabe mittels des Quicksort Algorithmus
- `fork()` - Erzeugen eines neuen Prozesses
- `open()` - Öffnen einer Datei
- `memcpy()` - Kopieren eines Speicherbereichs

## Hinweise zu Compilern (gcc, clang/llvm)

Ab hier beginnen wir mit der Erstellung von C-Programmen, daher einige Hinweise zu Compilern.

Die ***GNU Compiler Collection*** **(gcc)** war eines der ersten fertigen Programme des GNU-Projekts. Sie enthält verschiedene Programme zur Übersetzung von Quellcode. Während das Programm `gcc` den eigentlichen C/C++ Compiler darstellt, sind zum Beispiel mit `cpp` der C-Präprozessor, mit `as` der Assemblierer und mit `ld` der Linker einzeln nutzbar (→ jeweils `manpages` verfügbar).

Die einfachste Syntax lautet `gcc <Eingabedatei.c>` und führt dazu, dass `gcc` den Quellcode der Eingabedatei in ein ausführbares Binärprogramm mit dem Namen `a.out` übersetzt. Der Name a.out (kurz für Assembler Output) bezieht sich auf das ursprüngliche Dateiformat ausführbarer Programme und Objektdateien auf UNIXoiden Systemen. Zwar nutzen UNIX- und Linux Systeme schon lange die moderneren Formate COFF (Common Object File Format, aus dem sich auch das Portable Executable (PE) von Microsoft Windows ableitet) bzw. ELF (Executable and Linkable Format), aber der Standard-Ausgabe-Dateiname von GCC ist geblieben.

Dem Compiler können hunderte von Optionen übergeben werden, von denen vor allem der Name der Ausgabedatei (`-o <Dateiname>`), Optimierungen (`-O`, z.B. `-Os`, `-O1`, `-O3`), das Hinzufügen von Debugger-Informationen (`-g`) und Warnungen (`-W`, z.B. `-Wall`) für Sie interessant sein dürften.

Mit ***LLVM (Low Level Virtual Machine)*** steht uns ein weiteres moderneres Compiler-Framework zur Verfügung, das in Gegensatz zu GCC sehr viel modularer aufgebaut ist. Es kann als ein Baukasten für Compiler betrachtet werden, das die Infrastruktur bereitstellt um verschiedene Programmiersprachen in Maschinencode zu übersetzen. Dazu erzeugt LLVM zunächst Programmcode in einer plattformun-abhängigen Zwischensprache, der anschließend optimiert und in Maschinencode übersetzt wird.

Das Frontend für die Übersetzung von C-Programmen (C, C++, Objctive-C) nennt sich *Clang*, allerdings nutzten zum Beispiel auch *Rust* und *Swift* das LLVM Framework (dann aber eben unter Verwendung anderer Frontends).

Mit Hilfe des **Compiler Explorer** (https://godbolt.org) können Sie Quellcode und den daraus erzeugten Maschinencode unter Verwendung verschiedener Architekturen und Compiler vergleichen.

Welchen Compiler Sie einsetzen bleibt Ihnen überlasen, wobei meine Beispiele in der Regel Anweisungen für den Compiler *gcc* enthalten.

## Aufgabe 2 – Einstieg: “Hello World” Programm in C

In dieser Aufgabe erstellen Sie als Aufwärmübung ein kleines „Hello World“-Programm in C, das beim Aufruf einen Integer-Parameter erhält und den Text „Hello World!“ (mittels Schleife) so oft ausgibt, wie durch den Parameter angegeben. Fehlt der Parameter, soll eine Fehlermeldung angezeigt werden. Nutzen Sie einen Editor Ihrer Wahl (z.B. `nano` oder `vim` auf der Kommandozeile oder auch VSCode).

**1)** Erstellen Sie die Datei `HelloWorld.c`. Sie finden diese vorbereitet **in Git**.

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

int main (int argc, char **argv) {
	if (argc < 2) {
		fprintf (stderr, "Usage: %s <# loops>\n", argv[0]);
		exit (EXIT_FAILURE);
	}

	int loops = atoi (argv[1]);
	for (int i = 0; i < loops; i++) {
		printf ("%d of %d: Hello World!\n");
	}
	exit (EXIT_SUCCESS);
}
```

**2)** Kompilieren Sie das Programm (`gcc -o HelloWorld HelloWorld.c` ) und führen Sie es anschießend aus (z.B.: `./HelloWorld 5`). Was passiert? Sind die angezeigten Werte korrekt?

**3)** Beheben Sie den Fehler und kompilieren Sie das Programm erneut. Falls darin Funktionen verwendet werden, die Sie nicht kennen, werfen Sie einen Blick in die entsprechende `man`-Page.

Versuchen Sie die folgenden Fragen zu beantworten:

- Warum benötigen Sie `./` vor dem Programmnamen um das Programm zu starten?
- Was bedeutet `EXIT_SUCCESS` bzw. `EXIT_FAILURE`?
- Wie können Sie sich den Rückgabewert eines Programms in der Shell anzeigen lassen?
- Warum hat sich das Programm trotz des „Fehlers“ (im Schritt 2) compilieren und starten lassen? Woher kamen die Werte, die im `printf`-Statement ausgegeben wurden?

## Aufgabe 3 – Tutorial: Minimales C-Programm unterschiedlich beenden

In dieser Aufgabe erstellen wir ein sehr kleines C-Programm und beenden dieses auf verschiedene Arten: 1) Durch ein einfaches `return`-Statement, 2) durch den Bibliotheksaufruf `exit()`, und 3) durch den Systemaufruf `exit`. Für alle drei Arten sehen wir uns den Assembler-Quelltext an, um darin die unterschiedlichen CPU-Instruktionen zu entdecken – Sie müssen den Assembler-Code natürlich nicht im Detail verstehen.

**1)** Erstellen Sie die Datei `minimal_return.c` und schreiben Sie ein minimalistisches C-Programm.

```
int main (void) {
      return 42;
}
```

Kompilieren Sie das Programm mit Hilfe des C-Compiler und führen Sie es aus. Sie werden feststellen, dass nicht sonderlich viel passiert, da sich das Programm unmittelbar nach dem Start wieder beendet. Lassen Sie sich den Rückgabewert des Programms anzeigen.

Mit `gcc` können Sie das Programm zum Beispiel auch in den Assembler Quelltext (Option `-S`) oder Maschinencode (Option `-c`) übersetzen lassen. Wir wählen den Weg über den Maschinencode, da dieser sich im Anschluss mit Hilfe des Programms `objdump` (→ `manpage`) übersichtlich darstellen lässt.

Übersetzen Sie das Programm zunächst in Maschinencode (i.d.R. Dateiendung .o für Objectfile):

```
gcc -c minimal_return.o minimal_return.c
```

Nun können Sie mit `objdump` den relevanten Teil der Maschineninstruktionen anzeigen lassen:

```
objdump -d -M intel -S minimal_return.o
```

Die Ausgabe sollte relativ überschaubar sein. Relevant ist vor allem der im folgenden dargestellte Teil (der Assemblercode unserer Funktion `main()` innerhalb des `.text`-Segment der ausführbaren Datei).

```
int main (void) {                          ← Optische Verschönerung (unser C-Statement)
   0:  55                 push rbp         ← Vorarbeit: Sichern des Base Pointer (rbp)
   1:  48 89 e5           mov  rbp,rsp     ← Vorarbeit: Neuer Base Pointer ist Stackpointer
    return 42;                             ← Optische Verschönerung (unser C-Statement)
   4:  b8 2a 00 00 00     mov  eax,0x2a    ← 42 in Register eax
}                                          ← Optische Verschönerung (unser C-Statement)
   9:  5d                 pop  rbp         ← Nacharbeit: rbp wiederherstellen (für return)
   a:  c3                 ret              ← Return (an die Rücksprungadresse)
```

Maschinencode — Assembler-Quelltext

Sie sollten erkennen, dass (nach kurzer Vorarbeit bzgl. Base-Pointer und Stack-Pointer (mehr dazu hier)) der Wert 42 (HEX: 0x2a) in das CPU-Register `eax` geschrieben und anschließend die Instruktion `ret` aufgerufen wird, um aus der Funktion `main()` zu „returnen“. Bevor Sie weiter machen, überlegen Sie einmal, wohin dieses Return eigentlich führt ...

**2)** In der Regel wird die Bibliotheksfunktion `exit()` verwendet, um ein Programm zu beenden. Im vorherigen Beispiel führte das `return` aus `main()` zurück zu einem Wrapper namens `_start`, den der Compiler automatisch hinzugefügt hat und der das Programm für uns sauber beendet.

Schreiben Sie ein weiteres Programm `minimal_exit.c`, das mit `exit()` beendet wird.

```
#include <stdlib.h>
int main (void) {
      exit(42);
}
```

Sehen Sie sich auch hier den Maschinencode an. Sie sollten erkennen können, dass wieder der Wert 42 in ein (diesmal anderes) Register kopiert wird und anschließend ein `call` an eine Offset-Adresse stattfindet. An dieser Offset-Adresse (relativer Sprung von der aktuellen Programmcode-Position) befindet sich die Bibliotheksfunktion `exit()`, die sich um die Beendigung des Programms (per Systemaufruf `exit`) kümmert.

**3)** Um das Programm statt mit der Bibliotheksfunktion `exit()` nun mit Hilfe des `exit`-Systemaufrufs zu beenden, müssen wir Assemblercode schreiben. Dies nimmt uns die Bibliothek normalerweise ab.

Das Code-Beispiel funktioniert wie folgt: Zuerst wird die Syntax festgelegt, anschließend werden zwei Segmente (`.data` und `.text`) definiert. Im `.text`-Segment gibt es genau einen Einstiegspunkt `_start`, der unseren Programmcode enthält (wir schreiben den Code einfach direkt in die `_start`-Routine, statt uns um `main()` zu kümmern). Das Programm lädt die Nummer des Systemaufrufs (exit = 60) in das Register `rax` und den Rückgabewert (42) in das Register `rdi`. Anschließend wird die CPU-Instruktion `syscall` aufgerufen um den Systemaufruf zu initiieren.

```
.intel_syntax noprefix                ← Syntax für den nachfolgenden Code festlegen
.section .data                        ← .data-Section (in unserem Fall leer)

.section .text                        ← .text-Section (hier steht der Programmcode)
.globl _start                         ← Defnition eines Symbols/Einstiegspunktes namens _start
_start:                               ← Beginn des Einstiegspunkts _start

mov rax, 0x3c                         ← Wert für Systemcall exit in Register rax
mov rdi, 0x2a                         ← Returnwert in Register rdi
syscall                               ← Ausführen des Systemaufrufs
```

Schreiben Sie den Programmcode (natürlich ohne die Erklärungen in der rechten Spalte) in eine Datei namens `minimal_syscall.s`. Das Programm können Sie anschließend übersetzen und mit Hilfe des Linker zu einer ausführbaren Datei machen lassen.

```
gcc -c -o minimal_syscall.o minimal_syscall.s          ← Maschinencode erzeugen
ld -o exit_syscall exit_syscall.o                      ← Ausführbare Datei erzeugen
```

Auch hier können Sie sich nun den Inhalt der Objektdatei mit Hilfe von `objdump` anzeigen lassen, jedoch sehen Sie dann nur den Assemblercode, den Sie zuvor selbst geschrieben haben. Auch ausführen können Sie das finale Programm. Es sollte noch immer 42 zurückgegeben werden :)

**Abschließende Hinweise**

Im Assembler-Quelltext der drei Programme sehen Sie die unterschiedlichen CPU-Instruktionen, die angewendet werden. Meine Beispiele beziehen sich auf die 64-Bit Intel-Architektur. Wenn Sie auf einer anderen Architektur (z.B. macOS mit ARM-CPU, RISC-V, 32-Bit) arbeiten, wird der Assembler-Quelltext anders aussehen. Der Compiler ist es, der sich darum kümmert, dass der für die jeweilige Architektur benötigte Maschinencode heraus kommt.

Wenn Sie `objdump` auf die Binärdatei statt auf die Objektdatei anwenden (z.B. `minimal_return` statt `mimimal_return.o`), sehen Sie auch den zusätzlich generierten Programmcode (z.B. die Routine `_start`).

Falls Sie sich dafür interessieren, wie das Laden von Programmen unter Linux funktioniert – wie man beim Programmstart also überhaupt zur `main()`-Funktion gelangt – dann sehen Sie sich diesen Artikel an: http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

## Aufgabe 4 – Bibliotheksaufrufe und Systemaufrufe verfolgen

Das Programm **`ltrace`** erlaubt es, Bibliotheksaufrufe (z.B. den Aufruf der Funktion `exit()` in unserem Beispiel aus Aufgabe 3.2) zu visualisieren.

**ltrace**

Verwenden Sie `ltrace` um die beiden Programme (`minimal_return` und `minimal_exit`) zu vergleichen. Rufen Sie die Programme wie folgt auf:

```
ltrace ./minimal_return
ltrace ./minimal_exit
```

Sie sollten erkennen, dass in dem zweiten Programm ein Aufruf der Bibliotheksfunktion `exit()` angezeigt wird, der im Falle des anderen Programms nicht auftaucht.

*Hinweis: Das dritte Programm können Sie mit `ltrace` nicht verfolgen, da wir dieses viel zu minimalistisch (aka nicht Standardkonform) gebaut haben.*

**strace**

Das Programm **`strace`** erlaubt es Systemaufrufe zu verfolgen.

Verwenden Sie `strace` um die drei Programme (`minimal_return`, `minimal_exit` und `minimal_syscall`) zu vergleichen. Rufen Sie die Programme wie folgt auf:

```
strace ./minimal_return
strace ./minimal_exit
strace ./minimal_syscall
```

Hier kommt eine Menge Ausgabe heraus, da schon für dieses minimalistische Programm einige Systemaufrufe durchgeführt werden müssen (u.a. für die Allokation von Speicher, dem Prüfen von Berechtigungen, usw.). Beim Programm `minimal_syscall` sieht es hingegen überschaubar aus, da wir hier ja die Startroutine `_start` mit nur drei Zeilen Assembler ausgestaltet haben.

Sie können `ltrace` und `strace` natürlich auch einmal mit etwas aufwendigeren Programmen testen.