

Kapitel 3: Systemwerkzeuge

3: Systemwerkzeuge

In diesem Kapitel wollen wir uns kurz die folgenden Werkzeuge anschauen

- ▶ `gcc` und `clang` – C Compiler
- ▶ `[l|s]trace` – A [library|system] call tracer
- ▶ `make` – Ein GNU-Tool zum Bauen von Programmen

3.1: GNU Compiler Collection (gcc)

Wichtige Compilerflags

- `-O3` Optimierung
- `-W` `-Wextra` `-Wall` Aktivierung alle Warnungen
- `-Werror` Behandelt Warnungen wie Fehler
- `-g` `-ggdb3` Generiert Debug-Symbole
 - `-E` Führe nur den Präprozessor aus
 - `-S` Generiere Assembler-Code
 - `-c` Nur kompilieren nicht linkern
- `-o file` Ausgabedatei
- `-static` Generiere statische Binärdatei

Hallo Welt – Normales Kompilieren

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     puts("Hello_World!");
6     return EXIT_SUCCESS;
7 }
```

```
$ gcc -O3 -W -Wextra -Wall -Werror hw.c -o hw
```

```
$ ls -lah hw | cut -d " " -f 5,9
```

```
6,6K hw
```

```
$ file hw | cut -d "," -f 2,4
```

```
x86-64, dynamically linked
```

```
$ objdump -p hw | grep NEEDED
```

```
NEEDED          libc.so.6
```

Hallo Welt – Kompilieren mit Debug-Symbolen

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main() {
5      puts("Hello_World!");
6      return EXIT_SUCCESS;
7  }
```

```
$ gcc -ggdb3 -W -Wextra -Wall -Werror hw.c -o hw
$ ls -lah hw | cut -d " " -f 5,9
7,6K hw
$ file hw | cut -d ", " -f 2,4
x86-64, dynamically linked
$ objdump -p hw | grep NEEDED
NEEDED          libc.so.6
```

Hallo Welt – Statisch kompiliert

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main() {
5     puts("Hello_World!");
6     return EXIT_SUCCESS;
7 }
```

```
$ gcc -O3 --static -W -Wall -Werror hw.c -o hw
$ ls -lah hw | cut -d " " -f 5,9
796K hw
$ file hw | cut -d "," -f 2,4
x86-64, statically linked
$ objdump -p hw | grep NEEDED
```

Anmerkungen

`-W -Wall -Wextra -Werror`: Verwenden Sie immer diese Compilerflags

`objdump`: Display information from object files.

`file`: Bestimmt den Dateitypen

- ▶ Verwenden Sie in der Regel `-ggdb3` oder `-O3`
- ▶ Statisch kompilierte Programme funktionieren auch ohne die `libc`
- ▶ Mittels `dietlibc` oder `ulibc` lassen sich auch schlanke statische Programme bauen
- ▶ Das Kommando `gcc` kann meist mit `clang` ausgetauscht werden

AddressSanitizer (ASan)

- ▶ Werkzeug von Google um fehlerhafte Speicherzugriffe zu erkennen.
- ▶ Ausführungszeit des Programms wird um ca. 70% verringert.
- ▶ Speicherbedarf des Programms ist ca. 3,5 mal so hoch.
- ▶ Sollte bei der Entwicklung immer aktiviert sein.
- ▶ Integration in clang und gcc: `-fsanitize=address`
- ▶ Hat in in dem Chromium Webbrowser über 300 Fehler gefunden.
- ▶ Nutzen Sie den ASan um Ihre Programmierfehler zu finden.

AddressSanitizer: Demo

```

1  #include <stdlib.h>
2
3  int main() {
4      int *a = malloc(100);
5      a[0] = 0;
6      int res = a[100];
7      return res;
8  }

```

```

make
cc -ggdb3 -W -Wall -Wextra -Werror -fsanitize=address
    asdemo.c -o asdemo
./asdemo
...
SUMMARY: AddressSanitizer: heap-buffer-overflow
/home/cforler/git/beuth/it-sec/vorlesung/examples/insecure
    /asdemo.c:6
in main
...

```

3.2: Das Dynamische Tracer-Duo: ltrace und strace

▶ ltrace

- ▶ Zeigt alle Library-Calls (Bibliotheksaufrufe) an
- ▶ Ausgabe: Funktionsnamen, Parameterlisten und Rückgabewerte
- ▶ **Library-Call**: Aufruf einer Funktion welche Teil einer Programmbibliothek ist

▶ strace

- ▶ Zeigt alle System-Calls (Systemaufrufe oder Syscall) an
- ▶ Ausgabe: Funktionsnamen, Parameterlisten und Rückgabewerte
- ▶ **Syscall**: Aufruf einer Funktion welche vom Kernel ausgeführt wird

Systemcall

- ▶ Viele Programme benötigen Funktionalität die nur der Kernel (Ring 0) bereitstellen kann
- ▶ Direkter Zugriff auf Hardware ist dem Ring 0 vorbehalten
- ▶ Glücklicherweise hat jeder Kernel eine API
- ▶ Der POSIX-Standard definiert diese Systemschnittstelle (**Achtung:** Windows unterstützt (noch?) kein POSIX)
- ▶ Bitte lesen Sie sich die folgenden Wikipedia Eintrag durch:
https://de.wikipedia.org/wiki/Portable_Operating_System_Interface
- ▶ Zu den Systemcalls gibt es auch Manpages (Abschnitt 2)
Beispiel: `$ man 2 write`
- ▶ Bitte lesen Sie sich die folgenden Wikipedia Eintrag durch:
<https://de.wikipedia.org/wiki/Systemaufruf>

Systemcall unter Linux Ausführen

▶ x86

- ▶ Assemblerbefehl: `int 0x80`
- ▶ Registerbelegung
 - ▶ Systemcall ID: `eax`
 - ▶ Parameter 1-6: `ebx, ecx, edx, esi, edi, ebp`

▶ x86_64

- ▶ Assemblerbefehl: `syscall`
- ▶ Registerbelegungen
 - ▶ Systemcall ID: `rax`
 - ▶ Parameter 1-6: `rdi, rsi, rdx, r10, r8, r9`

▶ Weiter Architekturen: `$ man 2 syscall`

Prozesseüberwachung mittels ptrace

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request r, pid_t Pid,
            void *addr, void *data);
```

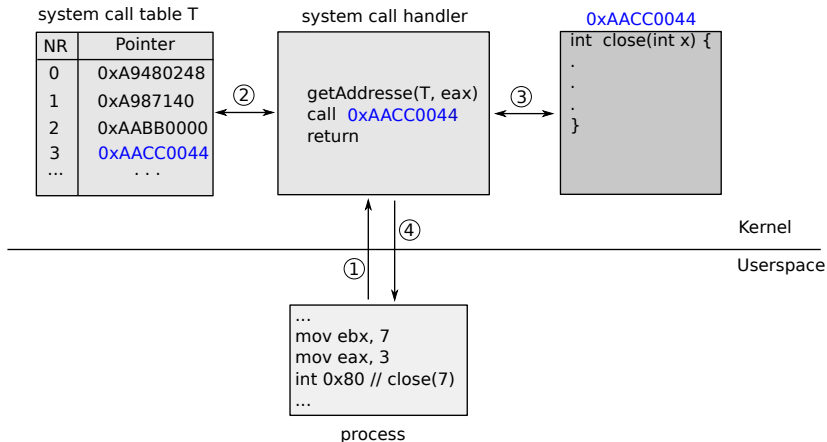
- ▶ Mit dem Systemcall `ptrace` lassen sich Prozesse überwachen.
- ▶ Es lassen sich Haltepunkte (engl. *break points*) setzen. Bei Erreichen eines solchen wird der überwachte Prozess angehalten.
- ▶ Mittels `ptrace` können auch Register- und Speicherinhalte gelesen oder beschrieben werden.
- ▶ Profiler und Debugger wie `ltrace`, `strace`, `gdb` und `valgrind` verwenden `ptrace` zur Prozessüberwachung.
- ▶ Weiter Informationen: `$ man 2 ptrace`

Systemcall unter Linux Ausführen

Die Systemcall ID ist in dem Headerfile `unistd_64.h` bzw. `unistd_32.h` definiert.

```
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
.
.
.
#define __NR_copy_file_range 326
#define __NR_preadv2 327
#define __NR_pwritev2 328
```

Systemcall Ausführen



strace

```
$ strace ./hw
execve("./hw", [ "./hw" ], [ /* 33 vars */ ]) = 0
...
open("/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, ..., 832) = 832
...
close(3)
...
write(1, "Hello World!\n", 13Hello World!) = 13
exit_group(0)                                = ?
+++ exited with 0 +++
```

Die `strace`-Ausgabe beinhaltet auch das Laden des Prozesses und der benötigten Programmbibliotheken.

Bibliotheksaufrufe

- ▶ Implementation der Funktion befindet sich in einer anderen Binärdatei.
- ▶ Beim Starten des Programmes werden die benötigten Funktionen aus den entsprechenden Bibliotheken nachgeladen
- ▶ Programme sind in der Regel von Bibliotheken abhängig
- ▶ Dynamisch gelinkte Programme (Default) hängen von der Standard-C-Bibliothek `libc` ab.
- ▶ Der Befehl `$ objdump -p hw | grep NEEDED` zeigt alle Abhängigkeiten des Programms `hw` an.
- ▶ Zu den Standard-Bibliotheksaufrufen gibt es auch Manpages (Abschnitt 3). **Beispiel:** `$ man 3 puts`
- ▶ Bitte lesen Sie sich den folgenden Wikipedia-Eintrag durch:
<https://de.wikipedia.org/wiki/Programmbibliothek>

ltrace

```
$ ltrace ./hw
__libc_start_main(0x400520, 1, ...
puts("Hello World!"Hello World!)  = 13
+++ exited (status 0) +++
```

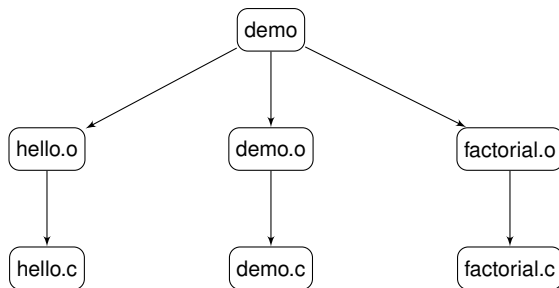
Die ltrace-Ausgabe verrät:

Die Funktion main ruft die Bibliotheksfunktion puts mit dem Parameter "Hello World!" auf. Der Rückgabewert ist 13. Anschließend wird das Programm erfolgreich beendet.

Merke: Mit der Option `-l library_pattern` werden nur Bibliotheksaufrufe von ausgewählten Bibliotheken angezeigt.

Beispiel: `$ ltrace -l libc* ./hw`

3.3: make – Bauanleitung für Programme



- ▶ Das Compilieren von Programmen ist oftmals kompliziert
- ▶ Von Hand bauen ist oft sehr mühselig
- ▶ Daher gibt es Programme wie `make` mit denen sich der Build-Prozess automatisieren lässt
- ▶ Einige Alternativen zu `make`: `SCons`, `CMake` und `Gradle`

Eine Demoanwendung – Teil 1

demo.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "functions.h"
4
5  int main(){
6      print_hello();
7      printf("The_factorial_of_5_is_%u\n", factorial(5));
8      return EXIT_SUCCESS;
9  }
```

functions.h

```
1  # pragma once
2  void print_hello();
3  unsigned int factorial(unsigned int a);
```

Eine Demoanwendung – Teil 2

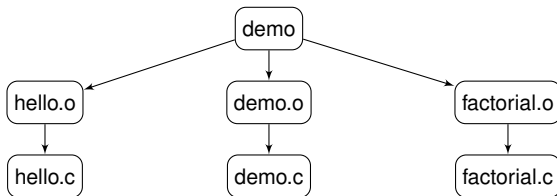
hello.c

```
1  #include <stdio.h>
2  #include "functions.h"
3  void print_hello(){
4      puts("Hello_World.");
5  }
```

factorial.c

```
1  #include "functions.h"
2
3  unsigned int factorial(unsigned int a) {
4      if(a<2) return a;
5      else   return(a * factorial(a-1));
6  }
```

Manuelles Bauen




- ▶ **Annahme:** Alle Quellendatei der Demoanwendung befinden sich in einem Verzeichnis (z.B. `src`).
- ▶ Bei dieser Demo ist der manuelle Buildvorgang noch einfach

```
$ gcc -W -Wextra -Wall -Werror demo.c hello.c factorial.c -o demo
```
- ▶ Das manuelle Bauen skaliert aber nicht.
- ▶ Stellen Sie sich ein größeres Softwareprojekt wie z.B. Mozilla-Firefox mit hunderten bzw. tausenden von C-Dateien, vor.

Makefile Regeln

Syntax einer Makefile-Regel

target: Zutaten

 Rezept

 ...

 ...

Variablen

- ▶ Zuweisung: `FOO = bar`
- ▶ Lesender Zugriff (Referenzierung von Variablen)
 - ▶ `$(FOO)` oder
 - ▶ `${FOO}`
- ▶ Funktionsweise des Referenzieren: strikte Textersetzung.
- ▶ Referenz wird durch Variableninhalt ersetzt.

Demoanwendung – Einfaches Makefile

```
1 CFLAGS += -W -Wall -Wextra -Werror -O3
2
3 demo: hello.c factorial.c
4
5 clean:
6     $(RM) *~ *.o demo
```

- ▶ **RM:** Steht für `rm -f`.
- ▶ **CFLAGS:** Compiler-Flags für den C-Compiler
- ▶ Der Befehl `$ make` baut die Demoanwendung
- ▶ Der Befehl `$ make clean` löscht die Demoanwendung
- ▶ Das Target `demo` wird mit einem Default-Rezept gebaut

Demoanwendung – Einfaches Makefile mit Objekten

```
1 CFLAGS += -W -Wall -Wextra -Werror -std=c11
2
3 demo: hello.o factorial.o
4
5 clean:
6     $(RM) demo *.o
```

Die Objektdateien `hello.o` und `factorial.o` werden mit einem Default-Rezept gebaut.

```
%.o: %.c
# Auszuführende Regel (eingebaut):
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

```
%.o: %.s
# Auszuführende Regel (eingebaut):
    $(COMPILE.s) -o $@ $<
```

Demoanwendung – Makefile mit Objekten

```
1 CFLAGS += -W -Wall -Wextra -Werror -std=c11
2
3 all: demo
4
5 hello.o: functions.h
6
7 factorial.o: functions.h
8
9 demo: hello.o factorial.o
10
11
12 clean:
13     $(RM) demo *.o
```

Bei Änderungen in der Headerdatei `functions.h` werden die Objekte `hello.o` und `factorial.o` neu gebaut.

Pseudo-Targets

- ▶ Targets werden nur gebaut, falls diese noch nicht existieren.
- ▶ Pseudo-Targets sollen immer gebaut werden.
- ▶ Alle Zutaten des Spezial-Targets `.PHONY` sind Pseudo-Targets.
- ▶ **Beispiel:** `.PHONY: clean`

Demoanwendung Makefile mit PHONY-Targets

```
1 CFLAGS += -W -Wall -Wextra -Werror -std=c11
2
3 .PHONY: clean all
4
5 all: demo
6
7 hello.o: functions.h
8
9 factorial.o: functions.h
10
11 demo: hello.o factorial.o
12
13 clean:
14     $(RM) demo *.o
```

Demoanwendung – Release-Debug Makefile

```
1  WARNFLAGS = -W -Wall -Werror -Wextra
2  OPTFLAGS = -O3
3  DEBUGFLAGS = -DDEBUG -ggdb3 -fsanitize=address
4  CFLAGS += $(WARNFLAGS)
5
6  .PHONY: clean all
7
8  all: release
9
10 hello.o: functions.h
11
12 factorial.o: functions.h
13
14 debug: CFLAGS += $(DEBUGFLAGS)
15 debug: demo
16
17 release: CFLAGS += $(OPTFLAGS)
18 release: demo
19
20 demo: hello.o factorial.o
21
22 clean:
23     $(RM) *~ *.o demo
```

Zusammenfassung

Sie sollten in der Lage sein...

- ▶ ...ein C-Programm zu kompilieren.
- ▶ ...immer die Compilerflags `-W -Wall -Wextra -Werror` verwende.
- ▶ ...wissen das die Compilerflags `-ggdb3 -fsanitize=address` ihnen helfen Programmierfehler zu finden.
- ▶ ...mittels `ltrace` und `strace` die Funktionsweise eines einfachen Programmes zu rekonstruieren.
- ▶ ...einfache Makefiles zu erstellen.