System-Programmierung 2: Funktionen

Slides: tmb.gr/syspr-2

Source auf GDocs

Überblick

Diese Lektion behandelt *Funktionen* in C.

Und zeigt, wie ein *System-Call* funktioniert.

So kommen wir zur *System-Programmierung*.

Funktionen in C

Definition einer Funktion:

```
return-type function-name(parameter-decl's) {
    declarations and statements
}
```

```
int max(int a, int b) {
    int m;
    if (a > b) { m = a; } else { m = b; }
    return m;
}
```

Deklaration einer Funktion:

```
return-type function-name(parameter-decl's);
```

```
int max(int a, int b);
```

Aufruf einer Funktion:

```
function-name(arguments);
```

```
int m = max(5, 7);
printf("%d", max(3, 4));
max(5, 7); // ignoriert Resultat
```

Argumentübergabe "by value"

power.c

Parameter *b* und *n* sind Kopien der Argumente *a*, *m*:

```
int power(int b, int n) { // Parameter b, n
    … // Änderung von n beeinflusst m nicht
}
```

Aufruf mit Argumenten *a* und *m*:

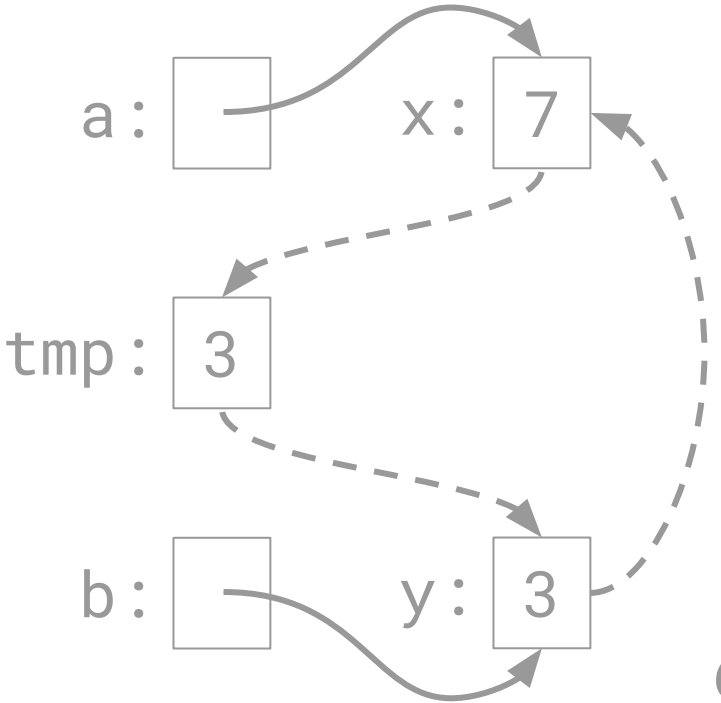
```
int a = 2;
int m = 5;
int q = power(a, m); // Argumente a, m
```

Argumentübergabe "by reference" swap.c

Parameter zeigen auf die übergebenen Argumente:

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int x = 3, y = 7;
swap(&x, &y);
```



Funktion in Datei auslagern main.c, f.c

Deklaration der Funktion *f* in *main.c* (Impl. in *f.c*):

```
// main.c              // f.c
void f(void);          void f(void) { … }
```

C Dateien einzeln mit *gcc -c* kompilieren:

```
$ gcc -c f.c       erzeugt f.o
$ gcc -c main.c    erzeugt main.o
```

Objektdateien zu einem Programm linken:

```
$ gcc -o my_program main.o f.o
```

Basis-Typen zurückgeben

z.B. Return-Wert vom Typ *int*:

```
int atoi(char *s) {
    int i;
    … // parsing
    return i;
}
```

Funktionsaufruf evaluiert zu *int*:

```
int value = atoi("2");
```

Struct-Typen zurückgeben

struct.c

z.B. Return-Wert vom Typ *struct point*:

```
struct point { int x; int y; };
struct point create_point(int x, int y) {
    struct point p = {x, y};
    return p;
}
```

Funktionsaufruf evaluiert zu Typ *struct point*:

```
struct point origin = create_point(0, 0);
```

Pointer zurückgeben

z.B. Return-Wert vom Typ *struct point* *:

```
struct point { int x; int y; };
struct point *create_point(int x, int y) {
    struct point *p = …;
    return p;
}
```

Funktionsaufruf evaluiert zu Typ *struct point* *:

```
struct point *origin = create_point(0, 0);
```

Hands-on, 15': Heap Struct

struct_v2.!c

In struct.c wird ein Struct auf dem Stack alloziert, mit return zurückgegeben und dabei "by value" kopiert.

Ändere das Programm so, dass *create_struct()* malloc() verwendet und einen Pointer zurück gibt:

```
struct point *create_point(int x, int y);
```

Passe den restlichen Code entsprechend an, der Compiler gibt Ihnen dabei nützliche Hinweise.

Globale Variablen

count.c

Globale, "externe" Variable bleibt erhalten:

```
int count; // global

void f() { count++; }

int main() {
    f(); f(); f();
    // count = 3
}
```

Sichtbarkeit von Variablen

Der *Scope* einer Variable beginnt mit der Deklaration:

```
int a = b; // error: b undeclared
int b = 0;
int c = b; // ok: b was declared
```

Auch globale Variable muss zuerst deklariert werden:

```
void f() { int i = j; } // error: j undeclared
int j;
void g() { int k = j; } // ok: j was declared
```

Sichtbarkeit von Variablen

scope.c

Jeder Block *{ }* spannt einen eigenen Scope auf:

```
int i = 0; // "extern", globaler Scope
void f() { // nicht geschachtelt
    int i = 1;
    { // freistehender Block
        int i = 2;
    }
    if (…) { int i = 3; … } else { … };
}
```

Funktionen sollten im Voraus definiert werden:

```
void f() { g(); } // warning: implicit decl.
void g() { … }    // (error, falls gcc -Wall)
```

Falls Reihenfolge fix*, hilft "Vorwärts-Deklaration":

```
void g(void); // forward declaration
void f() { g(); }
void g() { … }
```

*Oder falls Definition in einer anderen Datei.

Um eine Variable in mehreren Dateien zu nutzen, wird sie in ihrer Ursprungsdatei ganz normal definiert:

```
int i; // Definition von i und (unten) Array a,
int a[32]; // Speicher für a wird alloziert
```

Und in jeder weiteren Datei als *extern* deklariert:

```
extern int i; // Deklaration von i und a[],
extern int a[]; // kein Speicher alloziert

extern int i = 0; // Fehler, nicht erlaubt
extern int a[32]; // Dimension ist optional
```

Header Dateien /heater

Eine Header Datei erlaubt, Deklarationen zu teilen:

```
// heater.h
#define MIN_TEMP 5
void heater_up(void);
void heater_down(void);
int heater_temp(void);
```

```
// home.c
#include "heater.h"
void home_leave() {
    heater_down(); …
}
```

```
// heater.c
#include "heater.h"
int temp; …
```

Statische Variablen

static.c

Variablen sind über Dateigrenzen hinweg sichtbar:

```
int temp; // in heater.c, sichtbar in home.c
```

Modifier *static* begrenzt Sichtbarkeit auf die Datei:

```
static int temp; // nur sichtbar in heater.c
```

In Funktionen beschränkt *static* den Scope auf diese. Der Zustand bleibt über Funktionsaufrufe hinweg da:

```
void f() { static int count = 0; count++; }
```

Initialisierung

garbage.c

Globale, "externe" und *static* Variablen sind Null:

```
int i; // per Default mit 0 initialisiert
char c = '0' + 3; // konstante Expression
```

Lokale, "automatische" Variablen sind undefiniert:

```
void f() {
    int i; // nicht initialisiert, Garbage
}
```

Compiler Flags können hier helfen, siehe makefile.

Rekursion

fib.c

Eine Funktion kann sich selbst aufrufen:

```
int f(int n) {
    if (n < 2) { // "Abbruchbedingung"
        return n;
    } else {
        return (f(n-1) + f(n-2)); // Rekursion
    }
}
```

Pointers auf Funktionen

map.c

Funktion *map*, die Funktionen auf Arrays anwendet:

```
void map(int a[], int len, int (*f)(int));
int inc(int i); // Beispiel-Funktion
```

Implementierung wendet *f* auf die Elemente von *a* an:

```
for (int i=0; i<len; i++) { a[i] = f(a[i]); }
```

Aufruf mit *f* = *inc* Funktion, die ein *int* inkrementiert:

```
map({0, 0, 7}, 3, inc); // => {1, 1, 8}
```

Präprozessor

max.c

Jedes *#include* wird mit dem Datei-Inhalt ersetzt:

```
#include "file-name" // sucht im Source Dir.
#include <file-name> // folgt Such-Heuristik
```

Jedes Auftreten des Tokens wird textuell ersetzt:

```
#define token-name replacement-text
#define PI 3.14159
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Präprozessor #if:

```
#if int-expression
#elif int-expression
#else
#endif
```

Bedingte #defines:

```
#ifndef token-name (oder #ifdef token-name)
#define token-name
#endif
```

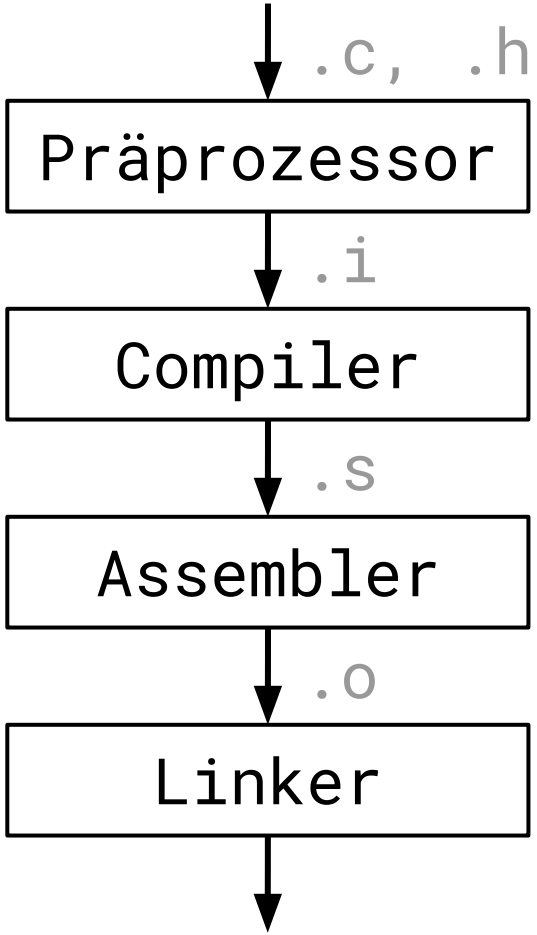
Kompilationsprozess

Schritt für Schritt:

```
$ echo "int main() {}" > my.c
$ cpp my.c > my.i          => my.i
$ gcc -S my.i              => my.s
$ as -o my.o my.s          => my.o
$ ld -o my my.o …          => my
```

Was *gcc* wirklich macht:

```
$ gcc -v -o my my.c
```



Libraries

Eine Library (Programmbibliothek) besteht aus vorkompilierten Objektdateien die mit einem Linker in ein Programm gelinkt werden können.

Statische Libraries *.a* werden ins Programm kopiert.

Dynamische Libraries *.so* werden zur Laufzeit in das Programm gelinkt, mit "dynamic linking". Der Code kann von mehreren Programmen genutzt werden.

System-Programmierung

Neben der Programmiersprache C brauchen wir für System-Programmierung ein Verständnis des UNIX/ Linux Betriebssystems, das in den Modulen *bsys* und *sysad* ausführlich behandelt wurde.

Hands-on, 5': Linux Betriebssystem

Aus welchen Teilen besteht das Linux Betriebssystem?

Suche online nach schematischen Darstellungen.

Welche Darstellungsweise findest Du besonders klar?

Was sind die jeweiligen Aufgaben einzelner Teile?

Betriebssystem-Kern

Betriebssystem kann auch Tools bedeuten, hier eher Core OS, *Kernel*; verwaltet Linux System-Resourcen.

Prozess-Scheduling; Memory-Management; Datei-System; Prozesse starten / beenden; Device-Zugang verwalten (USB etc.); Networking; *System Call API*.

Kernel- und User-Mode

Die CPU läuft im *Kernel-Mode* oder im *User-Mode*.

Teile des virtuellen Speichers können als User- bzw. Kernel-Space markiert werden; User dürfen weniger.

Manche Operationen sind nur dem Kernel erlaubt: z.B. der Zugang zur Speicherverwaltungs-Hardware, die Instruktion *halt* und Operationen für Geräte-I/O.

Kernel- und Prozess-Sicht

Für ein Prozess passieren Dinge asynchron, er weiss nicht, wann und wie lange er die CPU für sich hat, ob er im RAM oder ausgelagert ist, und wo auf der Disk Dateien physisch abgelegt sind; wie Device I/O geht.

Mit einem *System-Call* bittet der Prozess den Kernel, eine Aufgabe zu erledigen, die nur dieser kann/darf.

System Calls

Ein System Call ist ein kontrollierter Eintrittspunkt in den Kernel, der seine Dienste via API bereitstellt.

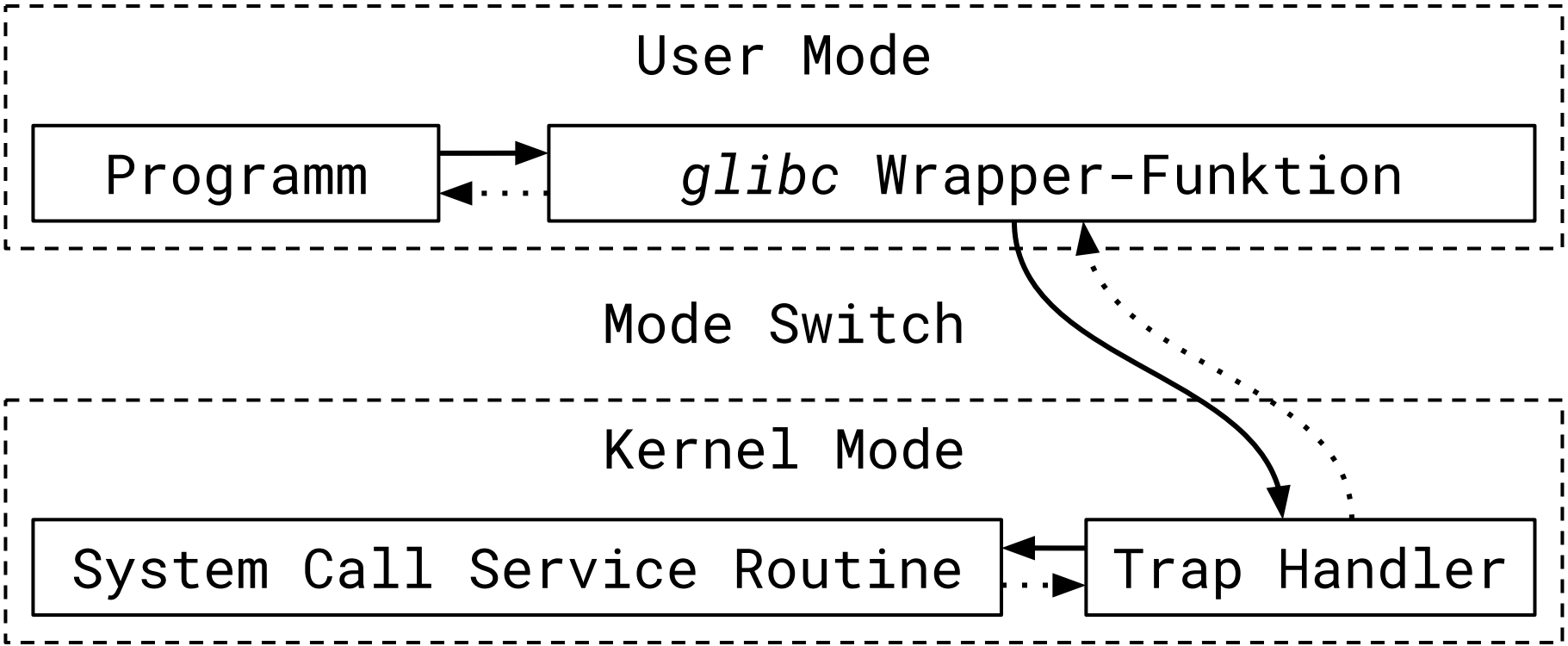
Bei System Calls geht die CPU in den Kernel-Mode.

Argumente werden kopiert v. User- zu Kernel-Space.

Jeder System Call hat einen Namen und eine Nr./ID.

Siehe syscalls.h in Linux, syscallent.h in *strace*.

System Call


User Mode
Programm
glibc Wrapper-Funktion
Mode Switch
Kernel Mode
System Call Service Routine
Trap Handler

Standard / GNU C Library

Die *Standard C Library*, kurz libc, ist die Standard-bibliothek der Sprache C und die Schnittstelle bzw. das *API* zwischen Anwendung und Betriebssystem.

Die *GNU C Library* glibc ist eine Implementierung der *libc* für GNU/Linux Systeme.

Ein Designziel von *glibc* ist Plattformunabhängigkeit, die Bibliothek ist in C (und Assembler) geschrieben.

System-Datentypen

Hardware-unabhängige Datentypen mittels *typedef*:

```
// User Code nutzt System-Datentyp, portabel
#include <sys/types.h>
pid_t pid = …; // pid_t ist immer gross genug

// sys/types.h, Hardware-abhängig, Hardware A
typedef int pid_t; // ein int hat genug Platz

// sys/types.h, Hardware-abhängig, Hardware B
typedef long pid_t; // hier braucht es long
```

Error Handling

error.c

Viele System Calls geben im Fehlerfall -1 zurück, der Fehlercode steht dann in der globalen Variable *errno*:

```
#include <errno.h> …

int fd = open(pathname, flags, mode);
if (fd == -1) { // Fehlerbehandlung
    if (errno == EINTR) { … } else { … }
}
```

Aber: Erfolgreiche Calls setzen *errno* nicht auf Null.

Fehlermeldung ausgeben mit *perror()*: errno.c

```
#include <stdio.h>
perror("open"); // liest errno
```

Oder Meldung mit *strerror()*:

```
#include <string.h>
char *msg = strerror(errno);
```

Manche System Calls geben im *Erfolgs*fall -1 zurück; dort setzt man *errno* vor dem Aufruf auf `0`.

Selbststudium: Kilo.c

Guten Code zu lesen hilft, besser zu programmieren.

Analysiere den Source Code dieses Text Editors:
https://github.com/antirez/kilo/blob/master/kilo.c

Kompiliere das Programm, starte und benutze es.

Wie verwaltet das Programm eingegebenen Text?

Wozu wird hier das *goto* Statement verwendet?

Feedback oder Fragen?

Gerne in Teams, oder per Email an

thomas.amberg@fhnw.ch

Danke für Deine Zeit.

Max Bougie on Twitter: "... "

https://twitter.com/monkchips/status/1040006321318375424

Home

Max Bougie

@monkchips

Following

Replying to @sarah_edo

Tweet

@selfsame@tiny.tilde.website

https://twitter.com/jplur_/status/1039592541656633344

Home

@selfsame@tiny.tilde.website

@jplur_

Following

The recursive centaur: half horse, half recursive centaur

9:12 PM - 11 Sep 2018

7,842 Retweets **32,467** Likes

236 7.8K 32K

Tweet

@selfsame
site
@jplur_
computers were a
Delray Beach,