Einführung

Ablaufbeschleunigung

- Cache — Beschleunigter Zugriff auf zwischengespeicherte Daten
- Pipeline — Beschleunigte Ausführung durch gestaffelte Verarbeitung

Arbeitsentlastung

- IC — **I**nterrupt **C**ontroller — Vermitteln von Interrupts
- DMA — **D**irect **M**emory **A**ccess — Daten kopieren ohne CPU-Interaktion
- FPU — **F**loating **P**oint **U**nit — Recheneinheit für Gleitkommazahlen
- DSP — **D**igital **S**ignal **P**rocessor — spezielle Daten-Recheneinheit
- GPU — **G**raphics **P**rocessing **U**nit — spezielle Graphik-Recheneinheit
- MPU — **M**emory **P**rotection **U**nit — Überwachung von Adresszugriffen

PC-HW: Zentrale Elemente

- CPU — **C**entral **P**rocessing **U**nit
- Memory — Speichert Daten und Instruktionen
- Input / Output — Interface zu externen Devices
- System-Bus — elektrische Verbindung der Komponenten

CPU

- Programmausführung
- Datenverarbeitung
- Master am Systembus

Memory

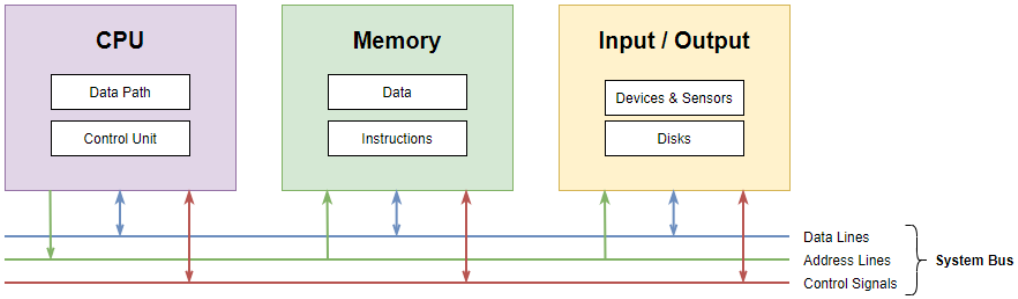
- RAM: Random Access Memory, behält die gespeicherten Daten nur solange es durch Strom gespiesen wird.
- ROM: Read-Only Memory, Daten definiert zur Produktionszeit, behält die Daten unabhängig von der Stromversorgung

Systembus

Verbindet die Komponenten des Computersystems. Die CPU signalisiert via. Systembus die gewünschten Zugriffe: Wer liest/schreibt wann und welche Daten?

I/O

- Anbindung des Computersystems an die Aussenwelt
- Lese-/Schreib-Schnittstellen für externe Hardware



Control-Unit

- IR — **I**nstruction-**R**egister, die aktuell ausgeführte Instruktion
- PC — **P**rogram-**C**ounter, gibt an, wo im Memory die nächste Instruktion liegt

Linda Riesen (rieselin)

C Programm Elemente

Datentypen

- Typen *char, int, float, double*
- Modifiers *signed, unsigned, short, long, long long*

Ersatz der Datentypen im String:

%d, %i (int), %u (unsigned int), %c (char), %s (char *), %f (float)

Literale

- Dezimal 1234
- Oktal **0**555 Unsigned!
- Hexadezimal **0x**3A Unsigned!
- ASCII **'**ASC**'**
- Konstanten *const*
- Symb. Konstanten *#define* String-Replace

Operatoren (*Left to Right* / *Right to Left*)

- Arithmetisch *+ - * / %*
- Relational *> >= < <=*
- Logische *&& ||*
- Gleichheit *== !=*
- Negation *!*
- Zähler *++ --*
- Inverse *~*
- Bit-Operatoren *& | ^ << >>*
- Zuweisung *= += -= *= /= %= &= ^= |= <<= >>=*
- Conditional *? :*
- Adress / Referenz *& **

Strukturen

- Eine Struktur ist ein neuer «Datentyp»

```c
//Struct mit Alias
typedef struct {
    double x;
    double y;
} Point2D;

Point2D point2D = { 2.0, 4.0 };
```

```c
//Struct ohne Alias
struct point2D {
    double x;
    double y;
};

struct point2D point2D = { 2.0, 4.0 };
```

Aufzählungstyp

- Erlauben die Definition einer konstanten Liste mit int-Werten
- Die konstanten Werte können in Ausdrücken verwendet werden

```c
enum weekday {
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3
};

printf("%i\n", Monday);
enum weekday mon = Monday;
```

```c
enum weekday {
    Monday,    // = 0
    Tuesday,   // = 1
    Wednesday  // = 2
};

printf("%i\n", Monday);
enum weekday mon = Monday;
```

```c
typedef enum  {
    Monday,    // = 0
    Tuesday,   // = 1
    Wednesday  // = 2
} weekday;

printf("%i\n", Monday);
weekday mon = Monday;
```

C ist Fehleranfälliger als bsp Java da

- -kein Automatischer Garbage Collection
- -keine Überprüfung der Array Grenzen
- -Pointers können falsch angewendet werden
- -Variabeln in C können alle wahlweise auf HEAP oder Stash alloziert werden
- String, boolean fehlt

Linda Riesen (rieselin)

C Funktionen (Prozedurale Programmiersprache = nicht objektorientiert)

Funktionen «Parameter by-value»

In C werden Parameter immer «by value» übergeben. Die Werte der Variablen, werden in die Funktion hineinkopiert.

- **D**eclare-**B**efore-**U**ser (DBU) Eine Funktion muss deklariert sein, bevor sie verwendet wird
- **O**ne-**D**efinition-**R**ule (ODR) Jeder Name darf nur eine Definition im gesamten Programm haben (except. Identische Typdefinitionen
- Deklaration und Definition müssen die gleiche Form haben.

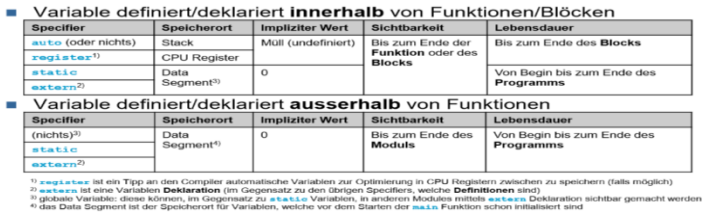
	Parameter	Rückgabewert
Basis-Datentypen	Gültig	Gültig
Strukturen und Aufzählungstypen	Gültig	Gültig
Arrays	Gültig	Ungültig
Pointer	Gültig	Gültig

```c
//Funktions-Kopf
int max(int a, int b);

//Funktions-Körper
int max(int a, int b) {
    if (a > b) return a;
    else return b;
}

int main(){
    //Funktions-Aufruf
    int x = max(3, 5);
}
```

Sichtbarkeit von Variablen

Typ	Sichtbarkeit	Bemerkung
Lokale Variablen	Block / Funktion	
Lokal-statische Variablen	Block / Funktion	Der Wert bleibt gespeichert
Globale Variablen	Source-File	
Global-statische Variablen	Programm	Der Wert bleibt gespeichert

Attribute von Variablen:



- Variable definiert/deklariert **innerhalb** von Funktionen/Blöcken

Specifier	Speicherort	impliziter Wert	Sichtbarkeit	Lebensdauer
auto (oder nichts)	Stack	Müll (undefiniert)	Bis zum Ende der **Funktion** oder des **Blocks**	Bis zum Ende des **Blocks**
register[1]	CPU Register			
static	Data Segment[3]	0		Von Begin bis zum Ende des **Programms**
extern[2]				

- Variable definiert/deklariert **ausserhalb** von Funktionen

Specifier	Speicherort	impliziter Wert	Sichtbarkeit	Lebensdauer
(nichts)[3]	Data Segment[4]	0	Bis zum Ende des **Moduls**	Von Begin bis zum Ende des **Programms**
static				
extern[2]				

[1] register ist ein Tipp an den Compiler automatische Variablen zur Optimierung in CPU Registern zwischen zu speichern (falls möglich)
[2] extern ist eine Variablen **Deklaration** (im Gegensatz zu den übrigen Specifiers, welche **Definitionen** sind)
[3] globale Variable: diese können, im Gegensatz zu static Variablen, in anderen Modules mittels extern Deklaration sichtbar gemacht werden
[4] das Data Segment ist der Speicherort für Variablen, welche vor dem Starten der main Funktion schon initialisiert sind

Funktionsparameter

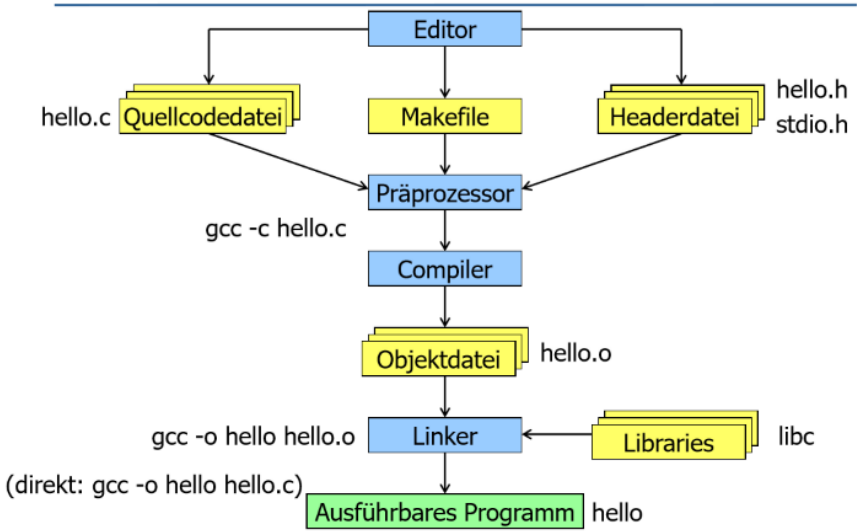
- Konstanter Parameter (*const*) Gibt an, dass ein Parameter innerhalb einer Funktion nicht verändert wird.
- Arrays Können nur «by Reference» übergeben werden
- Mehrdimensionale Arrays Alle Dimensionen ausser der ersten müssen angegeben werden
- Structs Können entweder «by Reference» oder «by Value» übergeben werden.
- Funktionen Können «by Reference» übergeben
- Variable Anzahl Parameter Mit der Ellipse «…» können beliebig viele Argumente übergeben werden (Letztes Argument)
- Void als Parameter Dann wird überprüft dass wirklich keine Parameter drin sind, (clean code!)

Linda Riesen (rieselin)

C Modulare Programmierung

Vom Source-Code zum lauffähigen Programm

1. **Präprozessor**
   - Präprozessor-Befehle beginnen mit #
   - Text-Einbindung aus anderen Dateien (*#include*)
   - Text-Ersetzungen im Quellcode (*#define*)
   - Text einbinden/ausschliessen (#ifdef, #elif, #else, #endif, #if, #ifndef)
2. **Compiler**
   - Wandelt den Quellcode in Objektdateien um
   - Der Objektcode enthält Maschineninstruktionen (nicht ausführbar)
   - Syntax-Check -> Ausgabe von Errors und Warnungen
   - Produziert eine Objekt-Datei pro Modul
3. **Linker**
   - Verbindet die offenen Aufrufe (check if used functions from extern do exists at place given)
   - Generiert ein ausführbares Programm
   - Funktionsaufrufe und Funktionen werden zusammengesetzt



Aufteilung des Quellcodes

- Ein Header-File pro Modul (*file.c*)

Header

- Verwendung
  - ✓ *#include «header.h»*
- Mehrfache Includes verhindern
  - ✓ «Include Guard»
- Enthält
  - ✓ Konstanten
  - ✓ Funktionsdeklarationen
  - ✓ User-Definierte Typen

```c
/* Header output.h */

//Include Guards-Start
#ifndef OUTPUT_H
#define OUTPUT_H

#include <stdlib.h>

//Andere Header Files
#include "data.h"

//Funktions-Kopf
void output_dot(data_t data);

//Include Guard-End
#endif
```

Header Datei wird ausgeführt und eingebunden

Gleiches Direct.: gcc -o name main.c header.c / Anderes Direct.: gcc -Idirect. -o etc.

Nützliche Libraries

- *<stdio.h>* Input / Output (scanf, printf)
- *<stdint.h>* Integer-Typen u Grössen OS-Unabhängig!
- *<stddef.h>* Pointer Subtraktion
- *<stdbool.h>* Boolean
- *<stdlib.h>* Standard-Bibliothek

Linda Riesen (rieselin)

Tests

```c
#include <CUnit/Basic.h>
#include "test_utils.h"
// setup & cleanup
static int setup(void)
{
    remove_file_if_exists(OUTFILE);
    remove_file_if_exists(ERRFILE);
    return 0; // success
}

static int teardown(void)
{
    // Do nothing.
    // Especially: do not remove result files - they are removed in
int setup(void) *before* running a test.
    return 0; // success
}

// tests
static void test_person_compare(void)
{
    // BEGIN-STUDENTS-TO-ADD-CODE
    // arrange
    CU_FAIL("missing test");
}



CU_ASSERT_TRUE(person_compare(&b, &a) > 0);
CU_ASSERT_PTR_NOT_EQUAL(anchor, anchor->next->next->next);
CU_ASSERT_PTR_EQUAL(anchor, anchor->next->next->next->next);
```

Linda Riesen (rieselin)

Make, Makefile

Make Utility

- Tool dient zum inkrementellen erzeugen von Programmen (= nur die out-of-dae Teile werden neu erzeugt)
- Ein Objekt ist «Out Of Date» wenn mind. eines seiner Bestandteile von neuerem Datum ist.
- Bei Aufruf von make werden die Regeln von Makefile (aus aktuellem Verzeichnis) abgearbeitet

Makefile

- Enthält Regeln was, wann, wie auszuführen ist (Zeilenorientiert)
  - Kommentare: #
  - Variabelndefinitionen
  - Explizite Regeln
- Regel besteht aus:
  - Target: was zu erstellen ist
  - Dependencies wovon das Target abhngig ist
  - Commands: 1-* abzuarbeitende Kommandos die immer dann ausgefgührt werden wenn eine deer dependencies ein jüngeres Mod. Datum hat als das Target

Make Aufrufe Optionen

- make -n (Dry Run, jede Regel wird abgeleitet u angegeben, aber keine Dateien werden verändert)
- -p alle Regeln u Variabeln werden aufgelistet
- | grep CFLAGS: überall werden CFLAGS gesetzt und verwendet

Make File: Eingebaute Regeln

- Abarbeitung der Regeln rekursiv

Make File: user defined, spezielle Variabeln

- make clean (Bsp spezielle Variable + Aufruf), wird definiert in makefile
- andere spezielle Targets: default, test, all, clean
  - damit dies sicher richtig läuft wird .PHONY als target angegeben
- Variable Definition: var := value;

Make File: Variabeln Substitution

- $(VAR) : Substitution mit unverändertem Inhalt
- $(VAR:%.c=%.o) wird mit dem Inhalt substituiert, bei jedem Wort ein terminierendes .c durch .o ersetzt.

Linda Riesen (rieselin)

C Pointers and Arrays

Aufbau eines Arrays

- Datentyp
- Name
- Anzahl Elemente

```
//Define and Initialize
int data[10] = {0, 1, 2};
//Assign values
data[3] = 3;
//data = 0, 1, 2, 3, 0, 0...
```

Eigenheiten von Arrays

- Können weder direkt verglichen **noch zugewiesen werden**
- Keine Default-Werte
- Keine Exceptions
- Keine Funktion zur Abfrage der Länge
- Bei der Übergabe eines Arrays wird nur der Pointer übergeben

Sizeof Operator -> nicht verwenden bei Structs für Pointer
Incrementation (+= 1 reicht dort)

- Speichergrösse des Datentyps in Byte an
- Verwendung mit Variable / Typ

```
sizeof(char);       // = 1
sizeof(char_var);   // = 1
sizeof(int);        // = 4
sizeof(int_var);    // = 4
```

Char-Array / Strings

- Letztes Zeichen «\0»
- Deklaration mit String-Literal
- Länge ermitteln mit *strlen()*
- Wichtige Funktionen *<string.h>*
  - ✓ Vergleichen *strcmp*
  - ✓ Kopieren *strcpy*
  - ✓ Zusammenhängen *strcat*

```
char array[] = "Hello World";
```

0	1	2	3	4	5	6	7	8	9	10	11
H	e	l	l	o		W	o	r	l	d	\0

Linda Riesen (rieselin)

char* meinezeile;

Aufbau eines Pointers

- Datentyp des Pointers
- Zeichen für Pointer *
- Name des Pointers

```
int var;    //Variable vom Typ int
int * pt;   //Pointer vom Typ int
pt = &var;  //Adresse zuweisen
```

Eigenschaften von Pointer

- Ein Pointer ist eine eigene Variable, die eine Adresse enthält.
- Ein Pointer hat einen Typ, damit er weiss bis zu welcher Speicherzelle der referenzierte Wert reicht.
- Können als Parameter übergeben werden (Arrays nicht)

Sizeof Pointer

- Pointer haben immer die gleiche Grösse (32 Bit-OS = 4, 64 Bit-OS = 8)

Operatoren

- *** Dereferenz-Operator
- *&* Adress-Operator

```
int x = 1;
int y = 2;      //x = 1, y = 2
int * pt = &x; //x = 1, y = 2, pt = 1
y = *pt;        //x = 1, y = 1, pt = 1
```

Typen von Pointern

- *Void*-Pointer Zeigt auf eine «nackte» Adresse
  Kann einem beliebigen Pointer zugewiesen werden
- *NULL*-Pointer Steht für die Adresse «0»
  Wird verwendet um anzugeben, dass es einen Fehler gab

Strukturen und Pointer

- -> Zugriff auf Strukturen,
  die als Pointer angegeben sind.

```
struct student {
    char name[30];
    char vorname[30];
};
struct student *sp;
sp->vorname;
sp->name;
```

Pointer können const sein und/oder auf const Objekte zeigen

Pointer Arithmetik

- *== !=* Pointer (Adressen) können verglichen werden
- *+ -* Mit Pointern (Adressen) kann gerechnet werden -> mult / division nicht erlaubt

Regel

Ist p ein Pointer auf das erste Element eines Arrays, so zeigt der Ausdruck $(p + i)$ auf das *i*-te Element.

- Umwandlung des Compilers $x[n] \rightarrow *(x + n)$

Beispiele$int\ array[5] = \{2, 4, 6, 8, 10\};$

- $int * pointer;$
- $pointer = array + 3;$ $pointer = \&array[3]$
- $*(pointer + 1) = 17$ $äquivalent\ zu\ p[1] = 17,\ a[4] = 17$

Mehrdimensionale Arrays

Wird ein Array in einem Ausdruck verwendet, so wird er implizit in den Pointer auf das erste Elemente (der ersten Dimension) konvertiert!

Müssen jeweils in Paramtern alles bis 1. (makmaPointer ausreichend) die Dimensionen Angegeben werden

- $a[2]$ $*(a + 2)$
- $a[2][3]$ $(a[2])[3]$ $*(*(a + 2) + 3)$

Jagged Arrays

- *Jagged Arrays* können unterschiedlich viele Elemente (gleiche Dimension) aufweisen.
- Dargestellt als eindimensionale Arrays von Pointern
- Die Elemente können unterschiedlich lang sein

```
// Jagged array (zweidimensionaler Array, der aber unterschiedliche Array-Längen erlaubt)
char *str[] = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};
```

*(p+1)[2] = 3. Char von Tuesday

Linda Riesen (rieselin)

Beispiele

- *int* p;*
- *char *d[20];* *// Array von Pointern*
- *double (*d) [20];* *// Pointer auf ein Array*
- *char **ppc;* *// Pointer auf Pointer*

Eigenheiten von Arrays

Beispiel 1

- *int a[5] = { 1, 2, 3, 4, 5 };*
- *a[3] = 4;* *// In Ordnung*
- *a[8] = 8;* *// Achtung Kein Fehler!!!*
- *a[-3] = -3;* *// Achtung Kein Fehler!!!*

Beispiel 2

- *const int b[5];* *//Sinnfrei, aber funktioniert*
- *b[0] = 33;* *//Kompilierfehler*

Beispiel 3

- *void *vp;*
- *double *dp = vp;* *//Kein Fehler!!!*

Pointer to Function

- *void logger (char *msg)*
- *void (*out) (char *)* *// Pointer auf Funktion*
- *out = &logger;* *// & - Operator optional*
- *(out) («Hello»);* *// * - Operator optional*
- *out («Hello»);*

C Dynamische Allozierung

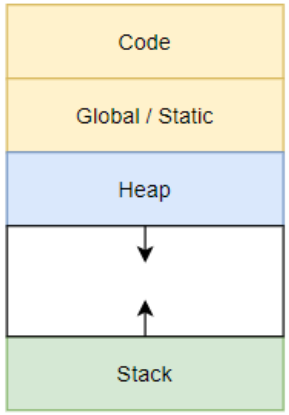
Heap – Dynamischer Speicher

- Speicherplatz kann dynamisch alloziert werden
- Allozierung «*malloc*», «*calloc*», «*realloc*»
- Freigabe «*free*»

```c
//Allocate memory (Heap)
node_t * node_ptr = malloc(sizeof(node_t));
//...
//Free memory (Heap)
free(node_ptr);
```

Stack – Automatischer Speicher

- Speicherplatz wird per default automatisch alloziert
- Bei jedem Funktionsaufruf wird Speicherplatz alloziert
- Der Stack Speicherbedarf verändert sich dauernd



Heap-Overflow

Der Heap ist zu klein oder zu fragmentiert, um ein genügend grosses Stück von zusammenhängendem Speicher zu reservieren.

Verhindern von Heap-Overflow

- *Ablauf anpassen* damit nicht gleichzeitig zu viel Speicher benötigt wird
- *Fragmentierung* des Speichers reduzieren
- *Konsequentes Fehler Handling* (Jede Anfrage muss geprüft werden)
- *Anwender-Eingaben konsequent prüfen*
- *Keine Unsicheren Funktionen verwenden (bsp fgets statt gets, free nur 1x)*

Heap Overflow:

Absturz durch Dereferenzieren eines NULL-Pointers

Heap Buffer Overflow:

Problem: Kontrollstrukturen für malloc()/free() überschrieben

und/oder Memory von anderen malloc() Aufrufen überschrieben

Stack-Overflow

Es hat nicht mehr genügend Speicherplatz auf dem Stack.

Verhindern von Stack-Overflow

- Rekursionen verbieten
- Rekursionen in der Tiefe limitieren
- Umfang von lokalen Daten limitieren
- Verwendung von Heap führt zu keiner Überschreibung von Systemdaten

Stack: Buffer-Overflow

Daten auf dem Stack werden überschrieben. (auch Systemdaten, bsp Ursprungsadresse, Programmcode)

Verhindern von Buffer-Overflow

- Sichere Funktionen verwenden
- Vorbedingungen prüfen, bevor Arrays beschrieben werden
- Anwender-Eingaben immer prüfen

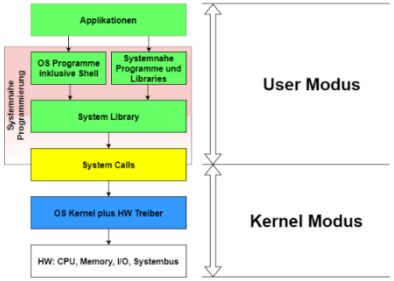
Linda Riesen (rieselin)

System Calls / System Libraries

Isolation

- Applikationen und Betriebssysteme haben einen «privaten» Speicher

User- und Kernel-Modus



- Kernel-Operation Kernel-Modul (alles erlaubt)
- Andere Operationen User-Modus (eingeschränkt)

System-Calls

- Wrapper – Funktion
- *syscall()* – Funktion Fehlerfall: Return -1 und setzt die Variable errno

Virtuelles Memory

- Alle Prozesse haben denselben *virtuellen Memory* Bereich
- Das virtuelle Memory hat *physikalischen Speicher* hinterlegt

MMU und MPU

- HW-Support: **M**emory **M**anagement **U**nit
  - Übersetzt logische Adresse in physikalische Adresse
  - Ein MMU beinhaltet auch die MPU Funktionalität
- HW-Support: **M**emory **P**rotection **U**nit
  - Überwacht den Adress-Bus auf unerlaubte Speicherzugriffe
  - Löst im Konfliktfall eine Exception aus

Standards

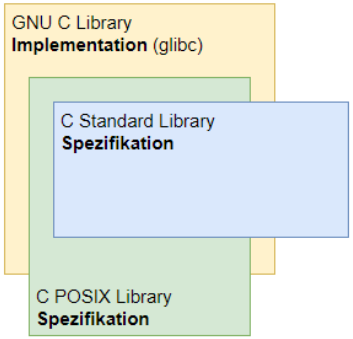
- Standard C-Library Teil des C-Standards
- Linux C Compiler (GCC = **G**NU **C**ompiler **C**ollection)

POSIX

- Definiert das C API zu UNIX-ähnlichen Betriebssystemen

Filesystem **H**ierarchy **S**tandard (FHS)

- Definiert für Unix-ähnliche Systeme (*/bin*, */dev*, */etc*, …)



Linda Riesen (rieselin)

Filesystem / IO

Reguläre Files

Ein zusammenhängender, unstrukturierter Array von Bytes, auch Byte-Strom genannt. Files können mehrfach geöffnet sein. Das OS stellt keine Synchronisation zur Verfügung.

Spezielle Files

Die speziellen Files liegen unter */dev.*

- Character Devices — Zugriff in Sequenz von Bytes (Tastatur, Maus, etc.)
- Block Devices — Zugriff in Arrays in Bytes (Massenspeicher)
- Named Pipes
- Sockets

File Länge

- Gemessen in Bytes
- Die Grösse kann manuell geändert werden

Inode

Verwaltungseinheit eines Files (Meta-Daten).

- Eindeutige *i-Nummer*
- Wird vom Kernel verwaltet
- Enthält: «Owner, Länge, Pfad, Grösse, usw.»

Der Filename ist nicht in der *Inode*.

Verzeichnis

Ein Directory ist ein File, welches eine «Map» von Namen (Pfad und *i-Nummer*).

File Deskriptoren

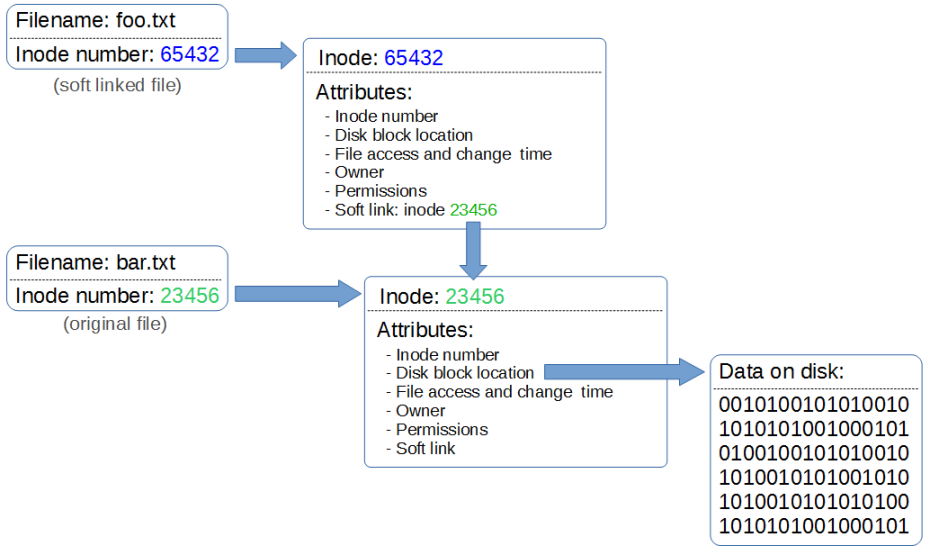
Geöffnete Files werden anhand einer Integer-ID verwaltet.

Hard-Link – ein Directory Eintrag

- Verschiedene Links können auf dieselbe *ino* verweisen.
- Die *Inode* eines Files enthält die Anzahl Links.

Symbolischer Link / Soft Link

Verweist nur auf ein File (*Inode)*. Entspricht einem Link in Windows.



Error Handling

Jeder I/O Zugriff kann fehlschlagen. Daher muss nach jedem Zugriff der Erfolg geprüft werden.

Stream-Buffering

- Unbuffered — Direkt gesendet
- Fully-Buffered — Gesammelt und gesendet sobald Buffer voll
- Line-Buffered — Gesammelt und nach einer Zeile gesendet

Linda Riesen (rieselin)

Task / Prozess / Thread

Tasks

- Task — Eine Aufgabe, die von der CPU abgearbeitet wird
- Batch-Ausführung — Sequenzielle Ausführung von Tasks
- Multi-Tasking — Parallele Ausführung von Task (max. CPU-Cores)

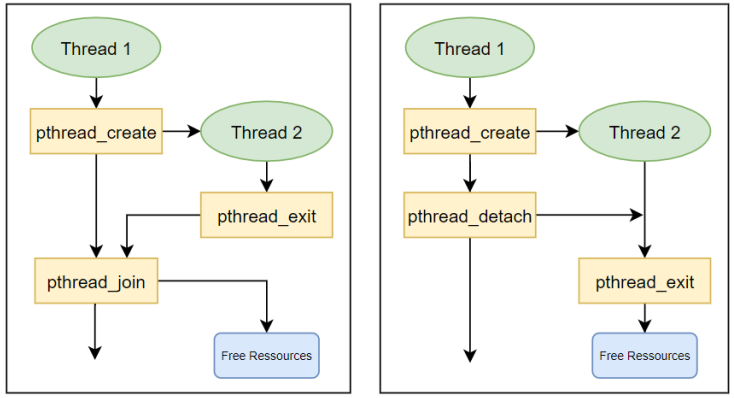
Kontext Switch

- CPU wechselt Task
- Jeder Task erhält die Illusion, er hätte die Kontrolle

Threads

Separater Kontrollfluss/Stack innerhalb eines Prozesses, teilt sich das Memory mit dem Eltern-Prozess.

- *pthread_create* — Erzeugt und startet einen Thread
- *pThread_join* — Wartet bis der angegebene Thread terminiert
- *pThread_detach* — Ressourcen werden beim Terminieren, freigegeben
- *pThread_exit* — Beendet einen Thread
- *pThread_cancel* — Unterbricht einen Thread von aussen



Scheduling

- Kooperativ — Jeder Task entscheidet, wann er die Kontrolle abgibt
- Präemptiv — Kontrollabgabe wird erzwungen

Der Scheduler unterbricht Tasks präemptiv und entscheidet, welcher Tasks als nächstes an er Reihe ist (priority-driven / round-robin). Gibt jedem Task die Illusion die alleinige Kontrolle über System zu haben.

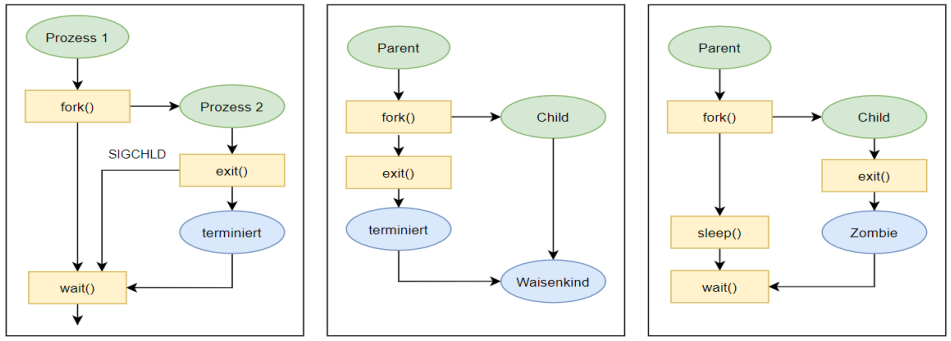
Prozesse

Ein Kontrollfluss/Stack, eigenes virtuelles Memory.

- *fork* — Erzeugt ein Child-Prozess (0 = Child, 1+ = Parent)
- *wait* — Wartet bis ein Child-Prozess terminiert
- *exit* — Terminiert den Prozess
- *exec* — Ersetzt ausführendes Programm (nach *fork*)
- *execv* — Führt Programm in neuem Thread aus
- *waitpid* — Nimmt den Exitcode des Child-Prozesses entgegen
- *WEXITSTATUS* — Exitcode aus return Status vom wait()-Call

Spezialfälle

- Waisenkind — Parent-Prozess existiert nicht mehr
- Zombie — *Wait* wird nach der Beendung des Childs aufgerufen



Linda Riesen (rieselin)

Interprozess Kommunikation

IPC

Die Fähigkeit des Kernels, Benachrichtigungen und Daten zwischen parallel ausgeführten Prozessen auszutauschen.

POSIX Signals (<*signal.h*>)

- Ein Prozess kann Signale senden
- Ein Prozess kann pro Signal definieren, was passieren soll

Default Aktionen

- *SIGINT* Interrupt-Signal von Tastatur (Ctrl + C)
- *SIGQUIT* Quit-Signal von der Tastatur (Ctrl + \)
- *SIGABRT*
- *SIGSTOP*

Signal-Handling

- *kill()* Sendet einen Signal-Code an einen Prozess
- *raise()* Analog zu *kill(getpid(), sig)*
- *sigaction()* Registriert den Signal-Handler
- *struct sigaction* Parametrisiert den *sigaction()* Aufruf
- *sigfillset()* Signale die blockiert werden sollen

```
// set action handler
struct sigaction a = { 0 };
a.sa_flags = SA_SIGINFO;
a.sa_sigaction = handler;
sigfillset(&a.sa_mask);
sigaction(sig, &a, NULL);
```

```
// set default action
struct sigaction a = { 0 };
a.sa_flags = 0;
a.sa_handler = SIG_DFL;
sigfillset(&a.sa_mask);
sigaction(sig, &a, NULL);
```

```
// set signal to be ignored
struct sigaction a = { 0 };
a.sa_flags = 0;
a.sa_handler = SIG_IGN;
sigfillset(&a.sa_mask);
sigaction(sig, &a, NULL);
```

POSIX Pipe

- Nur in einer Richtung (FIFO)
- Lesen und schreiben ist implizit synchronisiert

POSIX Message Queues

- Jede Message hat eine Priorität
- Bidirektional (Mehrere Schreiber und Leser)
- Strukturiert

POSIX Socket

- Verschiedene Protokolle
- Synchronisiert
- Bidirektional
- Unstrukturiert

Blockierend / Nicht blockierend

- I/O Zugriffe können blockierend oder nicht-blockierend ausgeführt werden.

Strukturiert / Unstrukturiert

Im Allgemeinen sind Daten in Linux unstrukturiert. Das heisst der Inhalt wird in Einheiten von Bytes bearbeitet.

- Shared Memory, Socket, Shared File

Strukturierte Daten sind dann vorhanden, wenn Zugriffe in grösseren bzw. abstrakteren Einheiten ablaufen. Messages beispielsweise werden nur als ganzes und nicht in Byte-Häppchen von Teilen der Message bearbeitet.

- Message Queue

Linda Riesen (rieselin)

Random Stuff

Bubble Sort:

```
for (i=0; i<n-1; i++){
    for (j=i+1; j<n; j++){
      if (a[i] > a[j]){
        t = a[i];
        a[i] = a[j];
        a[j] = t;
      }
    }
  }
```

What is a PID in C?

Every process on the system has a unique process ID number, known as the pid. This is simply an integer. You can get the pid for a process via the getpid system call.

Linda Riesen (rieselin)

Semaphore – Signalisierung

School of Engineering, InES Institute of Embedded Systems

- **Anwendung**
  - Tasks können über Semaphoren Synchronisationspunkte vereinbaren
- **Konzept: Ampel**
  - anlegen einer Semaphor-Instanz pro Synchronisationspunkt
  - Zähler, wie viele Tasks durchgelassen werden sollen
- **Operationen auf einer Semaphor-Instanz**
  - **Init**
    - Zähler = 0: ankommende Tasks müssen warten (von Beginn weg)
    - Zähler > 0: so viele Tasks können die Ampel passieren (von Beginn weg)
  - **Wait** (auch Down, P)
    - falls Zähler = 0: die aufrufende Task blockiert, wird in die Warteliste eingereiht
    - sonst wird der Zähler um eins vermindert, die aufrufende Task läuft weiter
  - **Signal** (auch Post, Up, V)
    - falls es Tasks in der Warteschlange hat, eine davon aus der Warteschlange entfernen und aktivieren (d.h. diese läuft wieder weiter)
    - ansonsten wird der Zähler um eins erhöht (d.h. Tasks werden durchgelassen)

21 ZHAW, Systemnahe Programmierung 25.04.2021

POSIX Semaphoren

School of Engineering, InES Institute of Embedded Systems

- **Unnamed Semaphoren**
  - In-Memory Semaphoren (typischerweise zwischen Threads)

```
#include <semaphore.h>              // declarations
sem_t sem;                          // semaphore instance
...
sem_init(&sem, 0, initial_value);   // init memory semaphore with initial count
...
sem_wait(&sem);                     // wait opertation
...
sem_post(&sem);                     // signal operation
...
sem_destroy(&sem);                  // cleanup
```

- **Named Semaphoren**
  - über Semaphoren File (typischerweise zwischen Prozessen)

```
#include <semaphore.h>              // declarations
sem_t *sem;                         // semaphore instance address
...
sem = sem_open("/name", O_CREAT);   // init IPC semaphore
...
sem_wait(sem);                      // wait opertation
...
sem_post(sem);                      // signal operation
...
sem_close(sem);                     // cleanup
...
sem_unlink("/name");                // remove file
```

25 ZHAW, Systemnahe Programmierung 25.04.2021

= Wie atomic variables, kleiner als Mutex da dieser noch blockieren kann

Default Values

In C, the default values of variables depend on their storage duration. There are three main types of storage duration: automatic, static, and dynamic.

1. Automatic variables (local variables):
    - If an automatic variable is declared without an initializer, its value is indeterminate, meaning it contains garbage data.
    - However, if an automatic variable is defined with an initializer, it will be initialized with the specified value.
2. Static variables (global variables or variables declared with the **static** keyword):
    - If a static variable is declared at the global scope or outside any function, it is initialized to zero by default.
    - If a static variable is declared inside a function, it retains its value between different invocations of the function. If not explicitly initialized, it is also initialized to zero.
3. Dynamic variables (allocated using dynamic memory allocation functions like **malloc**):
    - Dynamic variables do not have default values. They contain whatever value was previously stored in the memory location they occupy, and it's important to initialize them before use.

It is good practice to always initialize variables explicitly to avoid relying on their default values, as they may vary depending on the compiler or platform.

String.Split at «»

```
// Extract the first token
   char * token = strtok(string, " ");
   // loop through the string to extract all other tokens
   while( token != NULL ) {
      printf( " %s\n", token ); //printing each token
      token = strtok(NULL, " ");
   }
```

Reverse Array

```
void reverseArray(int arr[], int n) {
    int start = 0;
    int end = n - 1;
    while (start < end) {
        // Swap elements at start and end
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        // Move start and end towards the center
        start++;
        end--;
    }
}
```

Linux Befehle

Befehl	Hilfe	Beschreibung
echo		Anzeige
cd	**C**hange **D**irectory	
mkdir	**M**ake **Dir**ectory	Verzeichnis anlegen
nl	**N**umber **L**ines	Nummerierte Anzeige
ls	**l**ist	Auflisten von Verzeichnissen und Files
find	**find**	Suchen und anzeigen
wc	**W**ord **C**ount	Word Count
chmod	**Ch**ange **Mod**ification	Berechtigungen ändern
man	**Man**ual	
pwd	**P**rint **W**orking **D**irectory	
code	VS**Code**	Öffnet VSCode
gedit		Öffnet gedit
grep		Filtern / Suchen
apt	**P**ackage Manager **T**ool	
make	Build Utility	Default, clean, test, install und doc
gcc	**G**nu **C** **C**ompiler	
rm	**R**emo**v**e	Delete File
du	**D**isk **U**sage	
which		Locate command
Ln	**Link node**	
touch		File erstellen
findmnt		Listet die aktuell eingebundenen Filesysteme
mount		Bindet ein neues Filesystem ein
unmount		Entfernt ein Filesystem
ps		Prozess Zustände
pstree		Prozesshierarchie
top		Prozess Zustände
htop		Top mit CPU-Auslastung
lscpu		Auflistung der CPU's
cat /proc/cpuinfo		Ähnlich wie Lscpu

Standard I/O Umleitung

Eingabe aus Datei (anstelle von Tastatur)

- *… < file* Umleitung auf stdin

Ausgabe in Datei (anstelle von Tastatur)

- *… > new-file* Erstellt File mit stdout
- *… 1> new-file* Erstellt File mit stdout
- *… >> append-to-file* Hängt stdout an File an
- *2> new-error-file* Erstellt File mit stderr
- *>& new-combi-file* Kombiniert stdout / stderr

Pipe speist den stdout eines Kommandos in den stdin des nächsten.

- *Kommando1 … | Kommando2*

Bash

Zusammenfassung Shell Variablen

Setzen:	var=value	`usage="usage: cmd arg1 arg2"`
Anwenden:	$var oder ${var}	`wer=$USER`
Transformation:	${name//Text/Ersatz}	`var=${PATH//:/ }`
Kommando Output:	$(Kommando)	`n=$(cat myfile.txt | wc -l)`
Rechnen:	$((Ausdruck))	`i=$((i+1))`

```
for p in $path
do
    i=$((i+1))
    [ -n "$p" ] || p="."
    if [ -d "$p" ] && [ -x "$p" ]
    then
        find -L "$p" -maxdepth 1 -type f -executable -printf "$i:%h:%f\n" 2>/dev/null
    fi
done
```

`[ -f "$path" ]`	Existiert das File $path?
`[ -d "$path" ]`	Existiert das Directory $path?
`[ -x "$path" ]`	Execute Permission auf dem File oder Directory?
`[ -n "$var" ]`	Ist die Länge des Wertes **nicht** Null?
`[ -z "$var" ]`	Ist die Länge des Wertes Null

Linda Riesen (rieselin)