Tutorium

Grundlagenpraktikum: Rechenarchitektur (IN0005)

Lukas Ketzer

Lehrstuhl für Rechnerarchitektur & Parallele Systeme (Prof. Schulz)
TUM School of Computation, Information and Technology
Technische Universität München

22. April 2024

Wer bin ich?

- Lukas Ketzer
- 4. Semester Informatik Bachelor
- Mein zweites Mal GRA-Tutor
- System-Design-Zweig
- Meine Tutorien:
  - Gruppe 18: Do. 10:00 - 12:00 (00.08.059)
  - Gruppe 25: Do. 16:00 - 18:00 (01.06.011)
- Kommunikation
  - Zulip: öffentliche GRA-Streams
  - Zulip DMs: Lukas Ketzer (nur in dringenden Fällen)
  - lukas.a.ketzer@gmail.com (nur in dringenden Fällen)

Tutoriums-Webseite

Abbildung 1 home.in.tum.de/ketz/

Zulip

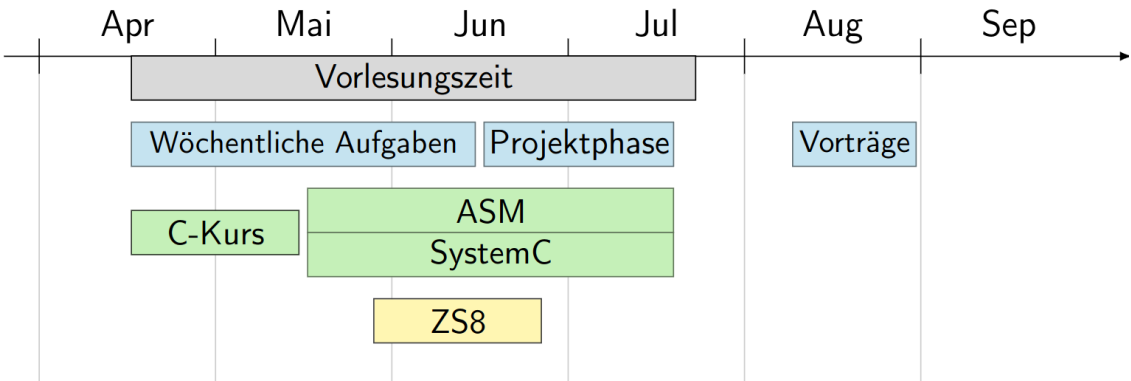
Abbildung 2 Zulip Gruppe 18 (Do. 10:00 - 12:00

Abbildung 3 Zulip Gruppe 25 (Do. 16:00 - 18:00

Ablauf / Wichtige Infos

- Ersten 4. Wochen: C Programmiersprache
- Letzen 4. Wochen:
  - Systemdesign in System C
  - ASM Optimierung und SIMD (x86-64 Assembly)
- Keine Ausarbeitung mehr (nur noch README)
- Wer noch 8 ECTS hat, muss weiterhin eine **ASM** Aufgabe bearbeiten.
- Anmeldung zu der Gruppenphase über Artemis (nicht TUMonline)

Ablauf


Apr
Mai
Jun
Jul
Aug
Sep
Vorlesungszeit
Wöchentliche Aufgaben
Projektphase
Vorträge
C-Kurs
ASM
SystemC
ZS8

C-Setup

- Windows
  - MinGW
  - GCC (WSL)
  - Visual Studio
- macOS
  - clang (Default)
  - gcc (homebrew)
- Linux
  - gcc (Default)
  - clang
- Rechnerhalle
  - `ssh <ito-username>@lxhalle.in.tum.de` (gcc)

SSH-Setup

- `ssh <ito-username>@lxhalle.in.tum.de` (gcc)
- SSH-Guide (Tutoriums-Webseite)
- ITO-Helpdesk

Frage zu letzten Woche?

- C- und SSH-Setup
- Von Java zu C
- Einführung in C
- Speicherbereiche in C

Aufgabe 1: Speicherbereiche

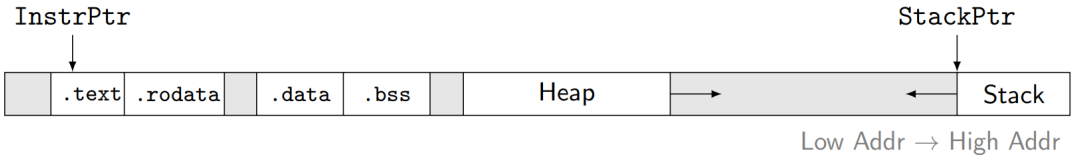


Abbildung 4 Virtueller Programmspeicher eines Programms im Arbeitsspeicher

Aufgabe 1: Speicherbereiche

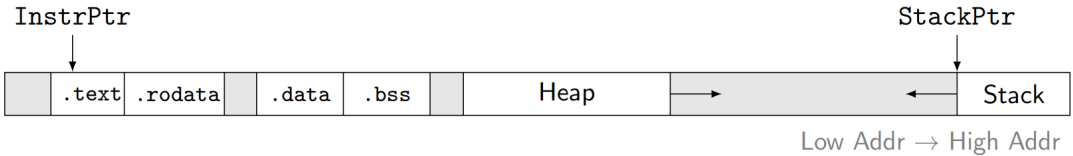


Abbildung 4 Virtueller Programmspeicher eines Programms im Arbeitsspeicher

Objdump einer **Binary**

```
objdump -wh <progname>
```

Aufgabe 1: Speicherbereiche

1. Betrachten Sie den Abschnitt *Sections* und lokalisieren die vier oben abgebildeten Abschnitte (`.text`, `.rodata`, `.data`, `.bss`). Können Sie aus den gezeigten Informationen Rückschlüsse auf den Inhalt ziehen?

Aufgabe 1: Speicherbereiche

1. Betrachten Sie den Abschnitt *Sections* und lokalisieren die vier oben abgebildeten Abschnitte (`.text`, `.rodata`, `.data`, `.bss`). Können Sie aus den gezeigten Informationen Rückschlüsse auf den Inhalt ziehen?

- CONTENTS: Die Section hat Inhalte in der Datei (sonst leer).
- ALLOC: Die Section benötigt Speicher.
- LOAD: Die Section wird in den Speicher geladen (wenn CONTENTS vorhanden sind, diese, ansonsten wird mit 0 initialisiert).
- READONLY: Die Section ist nicht beschreibbar.
- CODE: Die Section enthält Code und muss ausführbar sein.
- DATA: Die Section enthält Daten.

Aufgabe 1: Speicherbereiche

Hieraus ergibt sich dann die Bedeutung der Sections:

- `.text`*: Programmcode, der Bereich ist* **read-only** *und* **executable** *und kommt aus der Binary.*
- `.rodata`*: Konstante initialisierte globale Variablen, Strings, usw..*
- `.data`*: Initialisierte globale Variablen, i.d.R.* **read-write** *(aber nicht* **executable***) und kommt ebenfalls aus der Binary.*
- `.bss`*: Globale Variablen, die mit 0 initialisiert werden und daher keinen Speicherplatz in der kompilierten Datei benötigen. Ansonsten wie* `.data`*.*

Aufgabe 1: Speicherbereiche

2. Was ist der Unterschied zwischen den Speicherbereichen *Heap* und *Stack* und welchen Verwendungszweck haben diese?

Aufgabe 1: Speicherbereiche

2. Was ist der Unterschied zwischen den Speicherbereichen *Heap* und *Stack* und welchen Verwendungszweck haben diese?

- Stack: Lokale Variable, LIFO-Prinzip, **Variablen werden nach Beendigung der Funktion wieder freigegeben**
- Heap: Globale Variablen, speicher muss alloziert werden mit `malloc / calloc`, Speicher muss **wieder freigegeben werden**.

Aufgabe 1: Speicherbereiche

3. Betrachten Sie folgendes C-Programm. In welchen der Speicherbereichen werden die einzelnen Variablen alloziiert (`.text`, `.rodata`, `.data`, `.bss`, Heap, Stack)? Verwenden Sie auch die man-Pages von `malloc` und `alloca`.

```c
#include <alloca.h>
#include <stdlib.h>

int v0 = 6;
int v1;
const int v2[4] = {1, 3, 3, 7};

int main(int argc, char** argv) {
  int v3 = 5;
  int v4[v3];
  int* v5 = malloc(v3 * sizeof(int));
  if (v5 == NULL) abort(); // Wichtig!!
  int* v6 = alloca(v3 * sizeof(int));
  // ...
}
```

Aufgabe 1: Speicherbereiche

v0	v1	v2	v3	v4	v5	v6
.data	*.bss*	*.rodata*	***Stack***	***Stack***	***Heap***	***Stack***

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

- Pointer in C sind **Adressen auf Variablen im Speicher** (Stack oder Heap)
- * : **Dereferenzieren** eines Pointers oder deklarieren einer **Pointervariable**
- & : Adresse einer Variable bekommen;

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

- Pointer in C sind **Adressen auf Variablen im Speicher** (Stack oder Heap)
- * : **Dereferenzieren** eines Pointers oder deklarieren einer **Pointervariable**
- & : Adresse einer Variable bekommen;

```
int v = 5;     // Deklarieren einer Variable
int *a;        // Initalisieren einer Pointer-Variable

a = &v;        // Addresse von v bekommen und a zuweisen

int q = *a;    // Dereferenzieren von a und Wert in q
                  Schreiben

*a = 2;        // Wert hinter der Pointervaribale
                  a auf 2 setzen
```

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	11
`int val5 = *(array + 1)`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	11
`int val5 = *(array + 1)`	20
`int *ptr1 = array`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	11
`int val5 = *(array + 1)`	20
`int *ptr1 = array`	`0xfffe1000`
`int val6 = *ptr1`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	11
`int val5 = *(array + 1)`	20
`int *ptr1 = array`	`0xfffe1000`
`int val6 = *ptr1`	10
`int *ptr2 = &array[2]`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	11
`int val5 = *(array + 1)`	20
`int *ptr1 = array`	`0xfffe1000`
`int val6 = *ptr1`	10
`int *ptr2 = &array[2]`	`0xfffe1008` (`2 * sizeof(int) = 8`)
`int val7 = *ptr2`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	11
`int val5 = *(array + 1)`	20
`int *ptr1 = array`	`0xfffe1000`
`int val6 = *ptr1`	10
`int *ptr2 = &array[2]`	`0xfffe1008` (`2 * sizeof(int) = 8`)
`int val7 = *ptr2`	30
`int *ptr3 = array + 3`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	11
`int val5 = *(array + 1)`	20
`int *ptr1 = array`	`0xfffe1000`
`int val6 = *ptr1`	10
`int *ptr2 = &array[2]`	`0xfffe1008` (`2 * sizeof(int) = 8`)
`int val7 = *ptr2`	30
`int *ptr3 = array + 3`	`0xfffe100c` (`3 * sizeof(int) = 12 = 0xc`)
`int val8 = *ptr3`	

Aufgabe 2: Speicherzugriffe und Pointerarithmetik

C Source	Wert der Variable
`int array[4] = {10, 20, 30, 40}`	Beispielsweise irgendeine Adresse `0xfffe1000`
`int val1 = array[0]`	10
`int val2 = array[1]`	20
`int val3 = *array`	10
`int val4 = (*array) + 1`	11
`int val5 = *(array + 1)`	20
`int *ptr1 = array`	`0xfffe1000`
`int val6 = *ptr1`	10
`int *ptr2 = &array[2]`	`0xfffe1008` (`2 * sizeof(int) = 8`)
`int val7 = *ptr2`	30
`int *ptr3 = array + 3`	`0xfffe100c` (`3 * sizeof(int) = 12 = 0xc`)
`int val8 = *ptr3`	40

Exkurs: man-Pages

- Auf Linux-Systemen zum nachschlagen von Befehlen oder C-Funktionen
- Die wichtigsten man-Pages
  - 1: Programme oder Shell-Befehle (z.B. `objdump, grep, cd`)
  - 2: System- und Kernelfunktionen (z.B. `read, open, sigaction`)
  - 3: C-Bibliotheksfunktionen (z.B. `printf, fopen, sqrt`)
- Beispiele:
  - `man objdump`
  - `man 2 read`
  - `man 3 printf`
  - `whatis printf` (alle verfügbaren man-pages für einen Befehl sehen)

Aufgabe 3: Kopieren eines Strings

	`strcpy`	`strncpy`	`stpncpy`	`strlcpy`	`memccpy`
Standardisiert?					
Buffer-Länge spezifizierbar?					
Rückgabe von String-Ende?					
Schreibt immer NUL-Byte?					
Schreibt nur notwendige Bytes?					

Aufgabe 3: Kopieren eines Strings (Lösung)

	`strcpy`	`strncpy`	`stpncpy`	`strlcpy`	`memccpy`
Standardisiert?	*C*	*C*	*POSIX*	*– (BSD)*	*POSIX*
Buffer-Länge spezifizierbar?	—	*Ja*	*Ja*	*Ja*	*Ja*
Rückgabe von String-Ende?	—	—	*Ja*	*Ja*	*Ja*
Schreibt immer NUL-Byte?	*Ja*	—	—	*Ja*	—
Schreibt nur notwendige Bytes?	*Ja*	—	—	*Ja*	*Ja*

Vielen Dank für euer Aufmerksamkeit!