Programmieren in C

SS 2019

Vorlesung 6, Dienstag 4. Juni 2019

(Speicher, Debugger)

Prof. Dr. Peter Thiemann
Professur für Programmiersprachen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü4/5 — Babylon, Adjazenzmatrix
  - Ankündigungen
- Inhalt
  - Speicherorganisation
  - Lesen von C-Typen — const
  - Debugging — gdb
  - **Übungsblatt 6: Wumpus, Datenstrukturen, Graphik**

Erfahrungen mit dem Ü4/5 1/1

UNI FREIBURG

- Zusammenfassung / Auszüge
  - Ü4: Schwierige erste Aufgabe, Hinweise zu spät bemerkt

    Generell zuerst alle Aufgaben und Hinweise durchlesen (Klausur!). Dann mit der Aufgabe anfangen, wo die Lösung naheliegt. Zeitlimit setzen und zur nächsten Aufgabe übergehen, wenn die Zeit ohne weiterzukommen abgelaufen ist.
  - Zu viele Schwierigkeiten in einer Aufgabe

    Werden wir in Zukunft besser im Blick behalten.
  - Ü5: gut zurechtgekommen mit Adjazenzmatrix
  - Mehr Zeit für Fehlersuche → Heute Debugger
  - Zeitbedarf 8-10 Stunden → siehe erste Vorlesung

Ankündigungen 1/1

UNI FREIBURG

- Nächstes Blatt
  - Zwei Wochen Bearbeitungszeit
  - Etwas umfangreicher
  - Bonusaufgaben für die Opfer der babylonischen Gefangenschaft
- Weiter Sprechstunden im Juni (Raum 079-00-013)
  - 12.6. von 11-12
  - 17.6. von 12-13
  - 24.6. von 12-13

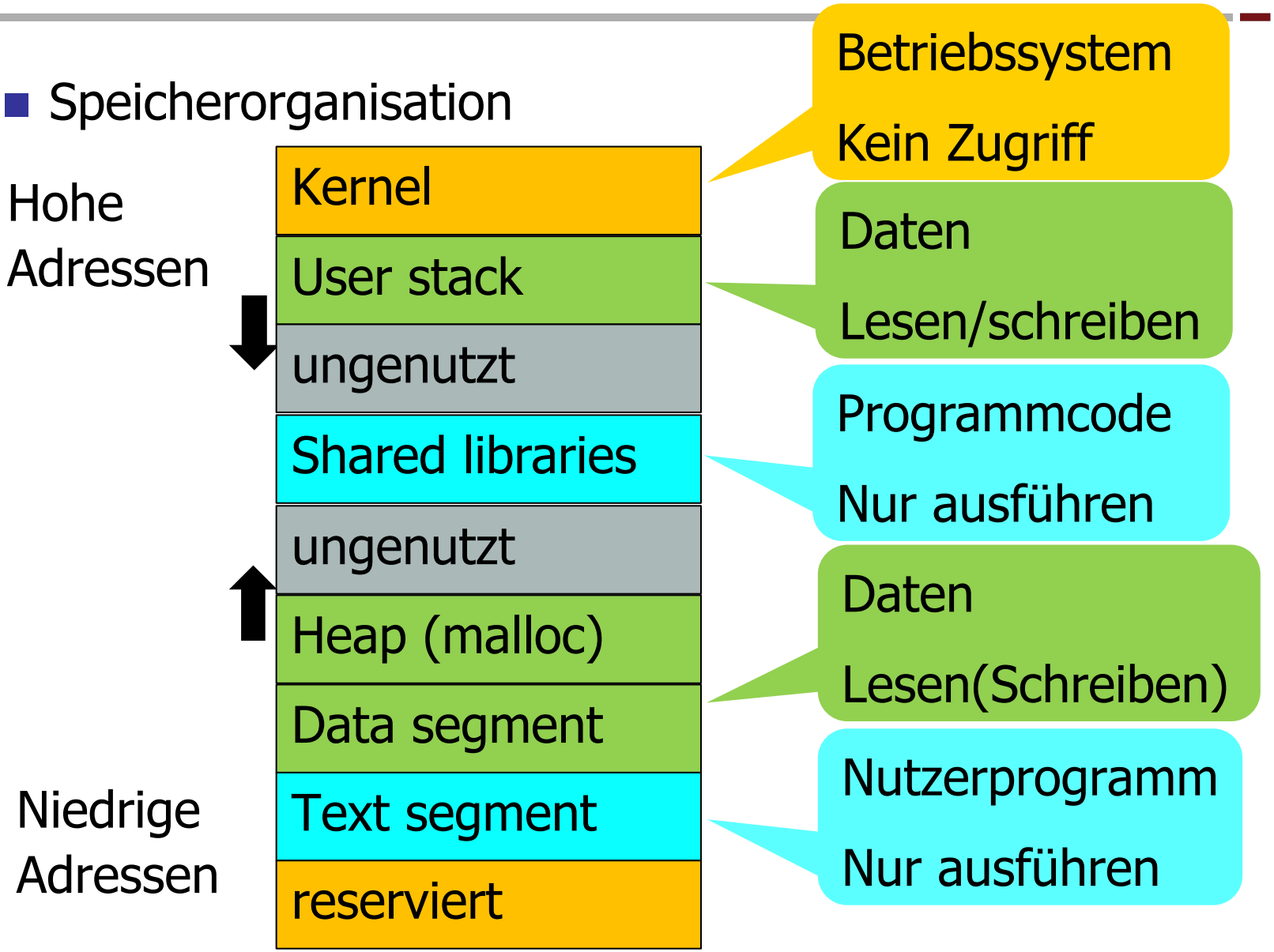
Speicher 1/11

UNI FREIBURG

- Physikalischer Hauptspeicher
  - Mehrere GB
  - Verwaltet vom Betriebssystem
  - Geteilt zwischen Betriebssysten und laufenden Prozessen
- Virtueller Speicher
  - Jeder Prozess hat die Illusion sich „allein“ im Adressraum des Prozessors zu befinden
  - Adressraum 2^{64} Bytes
  - Der Prozess darf nur auf kleine Bereiche davon zugreifen
  - Der Fehler segmentation violation zeigt einen unberechtigten Zugriff an

Speicher 2/11

- Speicherorganisation



Speicher 3/11

- Zugriffsrechte
  - Grüne Bereiche

    Lesen und Schreiben ☑ 😀

    Ausführen verboten ❌ 🤮

  - Blaue Bereiche
    - Lesen und Schreiben verboten ❌
    - Ausführen ok ☑

  - Beige Bereiche
    - Jeder Zugriff gibt segmentation violation 🤦

  - Graue Bereiche
    - Werden nach Bedarf dem Prozess zugeteilt

Speicher 4/11

UNI FREIBURG

- Das Datensegment

Data segment

  - Enthält (Platz für) alle Variablen, die global definiert sind
  - Beispiel

```
char *message = "Hello world!";

int counter = 0;

int myarray[42] = {0};
int main (void) {... }
```

  - Wieviel Platz brauchen wir hierfür im Datensegment?

Speicher 5/11

UNI FREIBURG

- Heap (Haufen)

Heap (malloc)

  - Wird durch stdlib verwaltet
  - Malloc() & Freunde vergrößern den Heap
  - Fordern notfalls neue Speicherbereiche vom Betriebssystem an
  - Leer bei Programmstart
  - Bleibt leer, falls ein Programm kein malloc() verwendet

Speicher 6/11

UNI FREIBURG

- Stack (Stapel)

User stack

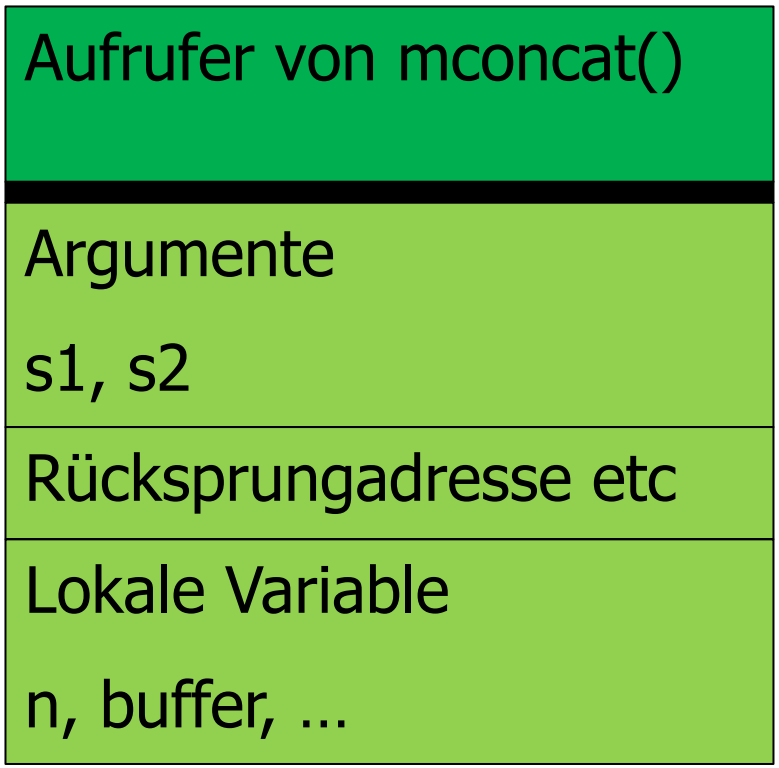
  - Speicherbereich zur Verwaltung von Funktionsaufrufen und Funktions-lokalen Daten
  - Jeder aktive Funktionsaufruf belegt ein **Stackframe** im Stacksegment
  - Stack wächst mit jedem Aufruf; schrumpf bei return
  - Beispiel aus V04:

```
char * mconcat (const char *s1, const char *s2) {
    size_t n = mstrlen (s1) + mstrlen (s2) + 1;
    char * buffer = …;
```

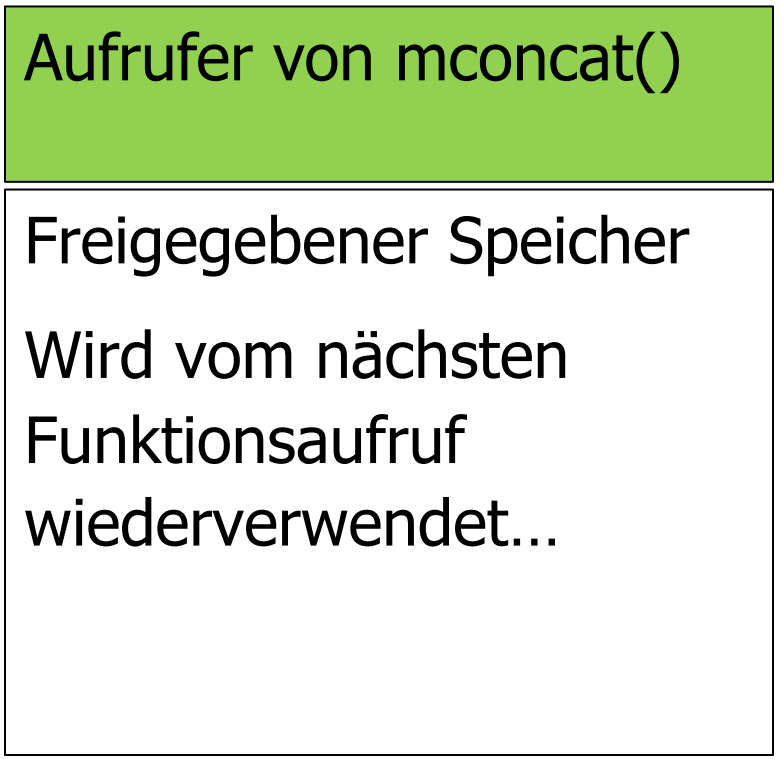
  - Die Variablen s1, s2, n und buffer befinden sich im Stackframe eines Aufrufs von mconcat()

Speicher 7/11

- Stackframe
  - Direkt nach Aufruf von.
  - mconcat("hello ", "world");



Nach return:



Speicher 8/11

UNI FREIBURG

- Stackframes
  - Stackframes werden wiederverwendet
  - Der nächste Funktionsaufruf verwendet den gleichen Speicherbereich
  - Lokale Variablen des früheren Aufrufs können eingesehen werden
  - Adressen von lokalen Variablen niemals zurückgeben
- Stacktrace
  - Liste der aktuell aktiven Stackframes
  - Alle offenen Funktionsaufrufe mit Parametern etc
  - Kann durch den Debugger angezeigt werden

Speicher 9/11

UNI FREIBURG

- Nullpointer

reserviert

  - Der unterste Speicherbereich ab Adresse 0 ist reserviert
  - Zeigervariablen dürfen den Wert Null annehmen (Nullpointer), aber ein Zugriff darüber ist nicht erlaubt und liefert segmentation violation
  - Daher muss vor jedem Zugriff über einen Zeiger sichergestellt werden, dass der Zeiger nicht Null ist.
  - Das wird oft vergessen und führt zu Fehlern

Speicher 10/11

UNI FREIBURG

- Der Billion-Dollar Mistake
  - C.A.R. Hoare ist Erfinder von QuickSort, Hoare Logic, CSP, Monitoren und … vom Nullpointer!
  - Er hat sich 2009 dafür entschuldigt:
  - I call it my **billion-dollar mistake**. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). **My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler**. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr [CC BY-SA 2.0 fr (https://creativecommons.org/licenses/by-sa/2.0/fr/deed.en)]

Speicher 11/11

UNI FREIBURG

- Speicherschutz mit const Typen
  - Das Schlüsselwort const kann in Typen verwendet werden um anzuzeigen, dass der Wert eines Speicherbereichs nicht geändert werden darf
  - Beispiele

```
size_t const buffersize = 2048; // don‘t assign to buffersize
char const *splash = “Hello, World!“;
                                  // must not assign into the string
char * const splosh = “Hi!“;    // must not assign to splosh
char const * const rigid = “Ultimate protection“;
                        // must not assign to rigid nor into the string
```

Tipp 1/2

UNI FREIBURG

- Lese Typen mit der Spiralregel
  - Beginne beim Namen der Variablen
  - Gehe weiter in einer Spirale im Uhrzeigersinn

    [X] – Feld (der Größe X) von …

    * -- Zeiger auf …

    Const -- Konstante

Tipp 2/2

UNI FREIBURG

- Quiz
  - int const buffer[20] = {0,1,2,3};
  - char (*ap)[20];
  - int * const * p;

Debugging 1/6

UNI FREIBURG

- Fehler im Programm kommen vor
  - Mit Feldern, Zeigern und malloc() lassen sich unangenehme **seg**mentation **faults** produzieren
  - Das passiert beim versuchten Zugriff auf Speicher, der dem Programm nicht gehört, zum Beispiel

```
int* p = NULL;    // Pointer to address 0.
*p = 42;          // Will produce a segmentation fault.
```

  - Schwer zu debuggen, es kommt dann einfach etwas wie:

```
Segmentation fault (core dumped)
```

    Ohne Hinweis auf die Fehlerstelle im Code (gemein)
  - Manche Fehler sind zudem nicht deterministisch, weil sie von nicht-initialisiertem Speicherinhalt abhängen

Debugging 2/6

UNI FREIBURG

- Methode 1: printf
  - printf statements einbauen

    an Stellen, wo der Fehler vermutlich auftritt

    von Variablen, die falsch gesetzt sein könnten
  - **Vorteil:** geht ohne zusätzliches Hintergrundwissen
  - **Nachteil 1:** nach jeder Änderung neu kompilieren, das kann bei größeren Programmen lange dauern
  - **Nachteil 2:** printf schreibt nur in einen Puffer, dessen Inhalt bei segmentation fault nicht ausgedruckt wird, wenn die Ausgabe in Datei umgeleitet wird. Abhilfe: nach jedem printf

    ```
    fflush(stdout);
    ```

Debugging 3/6

UNI FREIBURG

- Methode 2: gdb, der **GNU debugger**
  - Gbd Features
    - Anweisung für Anweisung durch das Programm gehen
    - Sogenannte breakpoints im Programm setzen und zum nächsten breakpoint springen
    - Werte von Variablen ausgeben (und ändern)
  - **Vorteil:** beschleunigte Fehlersuche im Vgl zu printf
  - **Nachteil:** ein paar gdb Kommandos merken

Debugging 4/6

UNI FREIBURG

- Grundlegende gdb Kommandos
  - **Wichtig:** Programm kompilieren mit der –g Option!
  - gdb aufrufen, z.B. gdb ./ArraysAndPointersMain
  - Programm starten mit run <command line arguments>
  - stack trace (nach seg fault) mit backtrace oder bt
  - breakpoint setzen, z.B. break Number.c:47
  - breakpoints löschen mit delete oder d
  - Weiterlaufen lassen mit continue oder c
  - Wert einer Variablen ausgeben, z.B. print x oder p i

Debugging 5/6

UNI FREIBURG

- Weitere gdb Kommandos
  - Nächste Zeile im Code ausführen step bzw. next

    step folgt Funktionsaufrufen, next führt sie ganz aus
  - Aktuelle Funktion bis zum return ausführen finish
  - Aus dem gdb heraus make ausführen make
  - Kommandoübersicht / Hilfe help oder help all
  - gdb verlassen mit quit oder q
  - Wie in der bash command history mit Pfeil hoch / runter

    Es geht auch Strg+L zum Löschen des Bildschirmes

Debugging 6/6

UNI FREIBURG

- Methode 3: valgrind
  - Mit Zeigern kann es schnell passieren, dass man über ein Feld hinaus liest / schreibt … oder sonst wie unerlaubt auf Speicher zugreift
  - Solche Fehler findet man gut mit **valgrind**

    Machen wir später

Hinweise zu Ü6-A5

UNI FREIBURG

- Folgende Pakete müssen auf Debian bzw Ubuntu installiert werden
  - libglfw3-dev
  - Libgles1
- ZB mit Befehl

sudo apt-get install libglfw3-dev libgles1

Literatur / Links

- Speicher
  - https://manybutfinite.com/post/anatomy-of-a-program-in-memory/
  - https://en.wikipedia.org/wiki/Call_stack
  - http://c-faq.com/decl/spiral.anderson.html
- Debugger / gdb
  - http://sourceware.org/gdb/current/onlinedocs/gdb