LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Arbeitsblatt 3
09.05.2022 - 15.05.2022

T3.1 Verwendung von Materialien

In dieser und den kommenden Wochen werden wir Ihnen für einige der Programmieraufgaben Materialvorlagen bereitstellen, die Ihnen das bearbeiten der Aufgaben erleichtern. Diese enthalten üblicherweise ein Rahmenprogramm, welches grundlegende I/O-Funktionalitäten bereitstellt, und ein `Makefile`, welches Anweisungen zum Kompilieren des Programms enthält.

1. Laden Sie die Vorlage herunter:

   ```
   wget https://gra.caps.in.tum.de/m/gcd.tar
   ```

2. Extrahieren Sie die Dateien aus dem Tar-Archiv:

   ```
   tar xf gcd.tar
   ```

3. Wechseln Sie in den neu angelegten Ordner `gcd`: `cd gcd`

4. Führen Sie den Befehl `ls` aus und vergewissern Sie sich, dass sich in diesem Ordner die Dateien `gcd.c` und `Makefile` befinden.

5. Öffnen Sie nun die Datei `gcd.c` mit einem Texteditor Ihrer Wahl.

   Diskutieren Sie mit Ihrem Tutor die Bedeutung folgender Zeilen:

   - `#include <stdio.h>`

     Diese Präprozessoranweisung bewirkt lediglich, dass der gesamte Inhalt der Datei `stdio.h` an dieser Stelle eingefügt wird. Diese sog. Header-Datei enthält Deklarationen von Funktionen wie z.B. `printf`, welche in der C-Standardbibliothek definiert sind.

   - `void test(int, int);`

     Diese* Deklaration *der Funktion gibt an, dass die genannte Funktion später oder in einer anderen Datei definiert wird. Zudem wird der Prototyp der Funktion (d.h. Rückgabetyp und Parametertypen) definiert.

   - `int main(int argc, char** argv)`

     Ein C-Programm beginnt üblicherweise bei der Funktion `main`. Die beiden Parameter geben die Anzahl und die Werte der Programmargumente an.

   - `printf(...);` – Nutzen Sie hierzu auch die man-Page `man 3 printf`.

6. Kompilieren Sie das Programm mittels `make` und führen Sie es aus. Verhält sich das Programm wie erwartet?

   Das Programm stürzt mit einem* Segmentation Fault *ab.

7. Bevor wir das Programm debuggen werden, lassen Sie sich die Disassembly des Programms anzeigen (`objdump -d gcd | less`). Können Sie die rekursiven Funktionsaufrufe identifizieren?

8. Öffnen Sie die Datei `Makefile`[1], passen Sie das Compiler-Flag `-O0` zu `-O2` an (was verändert dies?) und speichern Sie Datei. Löschen Sie die alte Binary mit `make clean` und kompilieren Sie das Programm erneut mit `make`. Wie verhält sich das Programm nun?

   Das Programm stürzt nicht mehr ab, sondern geht in eine Endlosschleife.

9. Lassen Sie sich erneut die Disassembly des Programms anzeigen (`objdump -d gcd | less`). Können Sie die rekursiven Funktionsaufrufe identifizieren?

   Die Rekursion ist vom Compiler optimiert worden. Diese sog.* Tail Call Optimization *ist möglich, da der Rückgabewert der Rekursion nicht mehr verändert wird. Damit ist es möglich Tail-Recursion durch eine Schleife zu ersetzen. Dies passiert natürlich nur bei aktivierter Compiler-Optimierung.

10. Ändern Sie das Compiler-Flags im Makefile erneut zu `-O0` und fügen Sie zusätzlich das Flag `-g`[2] hinzu. Kompilieren Sie das Programm erneut (`make clean gcd`).

11. Starten Sie den Debugger GDB (`gdb ./gcd`) und starten Sie das Programm (`r`). Lassen Sie sich den Backtrace anzeigen (`bt`). Erkennen Sie das Problem?

    Hinweis: Sie können auch ein ausgeführtes GDB-Kommando jederzeit mit *Ctrl-C* abbrechen.

    Es handelt sich um einen Stack Overflow.

12. Überlegen Sie, wie Sie das Problem lösen könnten.

    Man könnte eine andere Formulierung des Algorithmus nehmen, die kein Problem mit negativen Zahlen hat.

```
1 int gcd(int a, int b) {
2   return b ? gcd(b, a % b) : a;
3 }
```

[1] Wir werden Makefiles auf einem zukünftigen Blatt noch genauer betrachten.

[2] Nutzen Sie `man gcc` um herauszufinden, was dieses Flag macht. Es empfiehlt sich, die Suchfunktion durch Eingabe von `/-g\b` zu nutzen.

S3.1 Speicherzugriff und Pointerarithmetik in C

Im Gegensatz zu Java erlaubt C einen direkten Zugriff auf den Speicher. Eine Adresse entspricht in C einem sogenannten *Pointer*. In diesem Zusammenhang ist folgende Syntax relevant:

C Source	Erläuterung
`int v = 5;`	Definition einer normalen Variable
`int* a;`	Deklaration eines Pointers der auf ein (mehrere) `int` zeigen kann
`a = &v;`	Der Pointer wird auf die Adresse von `v` gesetzt
`int q = *a;`	Der Pointer wird *dereferenziert*, d.h. der Wert aus dem Speicher geladen
`*a = 2;`	Der Wert im Speicher auf den `a` zeigt wird auf 2 gesetzt

Zum Speichern mehrerer Werte gleichen Typs gibt es auch in C Arrays. Durch die Deklaration eines Arrays wird zu Ausführungsbeginn ein entsprechend großer Bereich im Speicher reserviert. Der Array-Name entspricht dabei der Adresse dieses Speicherbereichs.

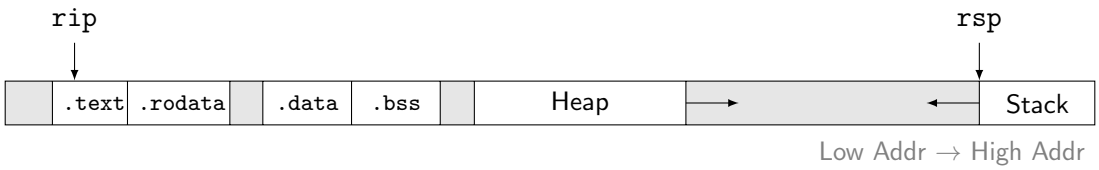
Welche Werte nehmen die Variablen in folgendem Code-Ausschnitt an?

C Source	Wert der Variable
`int array[4] = { 10, 20, 30, 40 };`	`0xfffe1000` *(beispielhaft!)*
`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*
`int val7 = *ptr2;`	*30*
`int* ptr3 = array + 3;`	*0xfffe100c*
`int val8 = *ptr3;`	*40*

Ein Array ist lediglich ein Pointer auf einen Speicherbereich – die Länge des Speicherbereiches muss bekannt sein, es gibt keinen automatischen Schutz vor out-of-bounds Zugriffen (sog.* Buffer Overflows*).

S3.2 Speicherbereiche

Der (virtuelle) Programmspeicher, also der Teil des Arbeitsspeichers, der für ein Programm zur Verfügung gestellt wird, ist wie folgt aufgeteilt:



1. Verwenden Sie folgenden Befehl, um sich die in der Binärdatei Ihrer Wahl gespeicherten *Sections* anzusehen:[3]

   `objdump -wh prog`

   Betrachten Sie dort den Abschnitt *Sections* und lokalisieren die vier oben abgebildeten Abschnitte. Können Sie aus den gezeigten Informationen Rückschlüsse auf den Inhalt ziehen?

   Relevant für diese Aufgabe ist lediglich die Spalte* Flags. *Dort ist (hier auszugsweise, die anderen Sections sind im Praktikum nicht relevant) Folgendes zu finden:

```
1 Idx Name                  Flags
2  12 .text                 CONTENTS, ALLOC, LOAD, READONLY, CODE
3  14 .rodata               CONTENTS, ALLOC, LOAD, READONLY, DATA
4  22 .data                 CONTENTS, ALLOC, LOAD, DATA
5  23 .bss                  ALLOC
```

   Die einzelnen Flags bedeuten:

   - **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.*

   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`.*

[3] Bei Interesse können Sie auch `readelf -aW prog` und `objdump -wx prog` probieren.

In Assembler muss die gewünschte Sektion mit `.section` `name` ***markiert werden werden. Es gibt Abkürzungen für*** `.text` ***(alias für*** `.section .text`***) und*** `.data` ***(alias für*** `.section .data`***).***

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

   - **Heap:** ***Hier können jederzeit Speicherbereiche alloziiert werden und zu jedem Zeitpunkt wieder freigegeben werden. Jede Allokation geschieht (in C) nur bei einem Aufruf von*** `malloc`***/***`calloc`***.***
   - **Stack:** ***Der Stack nimmt lokale Variablen auf und arbeitet nach dem*** **LIFO-Prinzip*****; Allokationen werden mit Ende der Funktion wieder freigegeben. (Wieso nämlich?) Die Größe des Stack ist begrenzt (i.d.R 8–16 MiB) und daher nur für kleine Allokationen (wenige kiB) geeignet. Das Ende vom Stack wird mit dem Stack-Pointer*** `rsp` ***markiert und muss vor einem Funktionsaufruf auf 16-Byte ausgerichtet sein.***

3. Betrachten Sie folgendes C-Programm. In welchen der Speicherbereichen werden die einzelnen Variablen alloziiert? Verwenden Sie auch `man malloc` und `man 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));
  // ...
}
```

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

Wichtige Anmerkung: ***Es gibt keine Garantie, dass eine dynamische Speicherallokation erfolgreich ist, weder auf dem Stack noch auf dem Heap. Wenn eine Heap-Allokation fehlschlägt, wird `NULL` zurückgegeben — dies muss zwingend überprüft werden. Bei einer Stack-Allokation lässt sich der Erfolg nicht überprüfen. Daher ist der Stack nur für kleine Allokationen geeignet.***

P3.1 Summe einer Linked List [2 Pkt.]

Eine *Linked List* ist eine Datenstruktur, wo jedes Element der Liste zusätzlich einen Pointer auf das nachfolgende Element enthält; oder `NULL`, falls es kein nachfolgendes Element gibt. Die leere Liste ist daher einfach ein `NULL`-Pointer. Schreiben Sie in C eine Funktion, die für eine Linked List mit Elementen vom Datentyp `uint64_t` die Summe der enthaltenen Elemente berechnet:

```
struct node { struct node* next; uint64_t val; };
uint64_t listsum(const struct node* list);
```

Aufgabentester: Sie können den Aufgabentester nutzen, um diese Funktion zu entwickeln und abzugeben. Wir empfehlen, dass Sie die Aufgabe dennoch zunächst lokal mit einem selbst erstellten Rahmenprogramm bearbeiten.

Referenzlösung:

```c
uint64_t listsum(const struct node* list) {
    uint64_t sum = 0;
    for (; list; list = list->next)
        sum += list->val;
    return sum;
}
```

P3.2 Brainfuck-Interpreter [4 Pkt.]

Brainfuck[4] ist eine esoterische, aber dennoch Turing-vollständige Programmiersprache. Als Zustand gibt es ein (theoretisch) unendlich großes Byte-Array und einen verschiebbaren Pointer auf diesen. Sie besteht aus lediglich acht Befehlen (alle anderen Zeichen, mit Ausnahme des Null-Bytes zum Anzeigen des Programmendes, werden als Kommentare behandelt und ignoriert):

- `>` — Inkrementiert Pointer
- `<` — Dekrementiert Pointer
- `+` — Erhöht Byte an der Stelle vom Pointer um 1
- `-` — Verringert Byte an der Stelle vom Pointer um 1

[4] `https://de.wikipedia.org/wiki/Brainfuck`

- `.` — Gibt Byte an der Stelle vom Pointer auf *stdout* aus
- `,` — Liest Byte von *stdin* und schreibt dies an die Stelle vom Pointer
- `[` — Springt zum *passenden* `]` nach vorn, wenn Wert am Pointer 0 ist; andernfalls wird der nächste Befehl ausgeführt.
- `]` — Springt zum *passenden* `[` zurück, wenn Wert am Pointer ungleich 0 ist; andernfalls wird der nächste Befehl ausgeführt.

Aufgabe: Schreiben Sie einen Brainfuck-Interpreter.

Vorlage: `https://gra.caps.in.tum.de/m/brainfuck.tar`

Empfehlung: Die folgenden Schritte können Ihnen als Hilfestellung zur Aufgabe dienen. Es kann hilfreich sein, die Ausgabe des Programms mit `./brainfuck <prog> | hexdump -C` anzusehen.

1. Machen Sie sich mit der Funktionsweise von C Structures (`struct`) vertraut. Wie können Sie auf die Felder eines `struct` zugreifen? Wie unterscheidet sich ein `struct` von einem Pointer auf ein `struct`? Wir empfehlen die Lektüre der Abschnitte 6.3 und 11.2 aus dem Buch *Modern C*.
2. Initialisieren Sie eine Variable für den Program Counter des Eingabeprogramms. Implementieren Sie nun Ihre Fallunterscheidung für die acht Instruktionen – Sie können hierfür das `switch`-Konstrukt benutzen.
3. Beginnen Sie nun mit der Instruktion `.` — dies wird Ihnen das Debugging deutlich erleichtern. Nutzen Sie hierzu die Funktion `putchar` (siehe auch `man putchar`).

   Hinweis: Sie können in C auf Felder eines `struct`-Pointers mit dem Operator `->` zugreifen.
4. Behandeln Sie nun die Instruktionen `+/-/>/<`. Achten Sie beim Verschieben des Pointers darauf, dass dieser sich nicht außerhalb des alloziierten Arrays befinden kann und behandeln Sie den Fehlerfall, indem Sie die Ausführung des Brainfuck-Programms abbrechen.
5. Für die Instruktionen `[/]` müssen Sie (sofern die Sprungbedingung zutrifft) eine Suche implementieren, die im String die passende Klammer findet. Sie können einfach in der entsprechenden Richtung über den String iterieren und die Verschachtelungstiefe von Klammern zählen. Wenn Sie wieder bei Tiefe 0 angekommen sind, haben Sie das Sprungziel erreicht.
6. Implementieren Sie zuletzt die Operation `,` mithilfe der Funktion `getchar`.

Referenzlösung:

```c
int brainfuck(struct BFState* state, const char* program) {
  size_t pc = 0;
  while (true) {
    switch (program[pc++]) {
    case '.': putchar(*state->cur); break;
    case ',': *state->cur = getchar(); break;
    case '+': (*state->cur)++; break;
    case '-': (*state->cur)--; break;
    case '>':
      if (++state->cur >= state->array + state->array_len)
        return -1;
      break;
    case '<':
      if (--state->cur < state->array)
        return -1;
      break;
    case '[':
      if (!*state->cur) {
        size_t nestdepth = 1;
        while (nestdepth) {
          switch (program[pc++]) {
          case '\0': return -1;
          case '[': nestdepth++; break;
          case ']': nestdepth--; break;
          }
        }
      }
      break;
    case ']':
      if (*state->cur) {
        size_t nestdepth = 0;
        do {
          if (!pc) return -1;
          switch (program[--pc]) {
          case '[': nestdepth--; break;
          case ']': nestdepth++; break;
          }
        } while (nestdepth);
      }
      break;
    case '\0': return 0; // End-of-program
    default: break; // Ignore
    }
  }

  return -1;
}
```

Q3.1 Quiz [4 Pkt.] (siehe Praktikumswebsite)