Grundlagenpraktikum Rechnerarchitektur (GRA)

Tutorübung

Moritz Beckel

02.05.2022 16:00 / 06.05.2022 15:00

Hausaufgaben

- Mul8
- Gauss
- Fakultät
- Quiz

T2.1 Setup und erste Schritte

Um ein einheitliches Setup zu gewährleisten, entwickeln wir auf dem Uni-Server der Rechnerhalle. Mittels einer SSH-Verbindung können Sie auch von Ihrem Laptop aus darauf zuzugreifen. Ansonsten benötigen Sie keine weiteren Tools.

Account Zur Anmeldung benötigen Sie Ihre Informatik-Kennung der Rechnerbetriebsgruppe (RBG) (dies ist *nicht* Ihre TUM-Kennung!). Sollten Sie Ihr Passwort vergessen haben, wenden Sie sich bitte an den RBG-Helpdesk. Weitere Informationen zur `lxhalle` finden Sie im RBG-Wiki[1]

https://wiki.rbg.tum.de/Informatik/Helpdesk/Passwort

T2.2 Von Java zu C

Ziel dieser Aufgabe soll es sein, die Formel des *kleinen Gauß* in C zu implementieren. Dabei soll mit Hilfe einer Schleife die Zahl $z = 1 + 2 + 3 + \cdots + n = \sum_{k=1}^{n} k$ für $n = 100$ berechnet, und anschließend auf die Konsole ausgegeben werden.

1. Überlegen Sie sich, wie Sie dieses Programm in Java implementieren würden.

T2.2 Von Java zu C

```
1  public class Gauss {
2
3    public static void main(String [] args) {
4      int sum = 0;
5
6      for (int i = 1; i <= 100; i++) {
7        sum += i;
8      }
9
10     System.out.println("Die Summe aller natürlichen Zahlen von 1 bis
           100 beträgt " + sum + ".");
11   }
12 }
```

T2.2 Von Java zu C

2. Was passiert beim Kompilieren und Ausführen eines Java-Programms? Was macht im Gegensatz dazu der C-Compiler? Wo liegen die Vor- und Nachteile?

T2.2 Von Java zu C

2. Was passiert beim Kompilieren und Ausführen eines Java-Programms? Was macht im Gegensatz dazu der C-Compiler? Wo liegen die Vor- und Nachteile?

- Java-Compiler erstellt aus dem Programmcode Java Bytecode
- Bytecode wird von JVM ausgeführt und zur Laufzeit in Maschinencode übersetzt

- C-Compiler übersetzt Programmcode bereits zur Kompilierzeit direkt in Maschinencode
- Nach Kompilierung keine Plattformunabhängigkeit

T2.2 Von Java zu C

3. Betrachten Sie nun die folgende C-Implementierung des Programms. Was ist anders, als Sie es in Java kennen? Wie funktioniert die Funktion *printf*?

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int sum = 0;

  for (int i = 1; i <= 100; i++) {
    sum += i;
  }

  printf("Die Summe aller natürlichen Zahlen" \
         "von 1 bis 100 beträgt %d.\n", sum);

  return 0;
}
```

T2.2 Von Java zu C

3. Betrachten Sie nun die folgende C-Implementierung des Programms. Was ist anders, als Sie es in Java kennen? Wie funktioniert die Funktion *printf*?

```c
#include <stdio.h>

int main(int argc, char **argv) {
   int sum = 0;

   for (int i = 1; i <= 100; i++) {
      sum += i;
   }

   printf("Die Summe aller natürlichen Zahlen" \
          "von 1 bis 100 beträgt %d.\n", sum);

   return 0;
}
```

Präprozessoranweisung:
Kopiert Inhalt von stdio.h
Enthält Deklaration von printf

Nicht objektorientiert

Andere Signatur
argc Anzahl Argumente
argv Argumentarray

Parameterübergabe:
Mit %d als Platzhalter
\n als Zeilenumbruch

Andere Rückgabewert

T2.2 Von Java zu C

Implementieren Sie dieses C-Programm nun auf der Rechnerhalle.

1. Loggen Sie sich auf der Rechnerhalle ein.
2. Erstellen Sie einen neuen Ordner *gauss*: `mkdir gauss`
3. Wechseln Sie in das neu angelegte Verzeichnis und schreiben Sie diese Implementierung mithilfe eines Texteditors Ihrer Wahl in die Datei `gauss.c`.
4. Kompilieren Sie Ihr Programm mithilfe des *Gnu-C-Compilers*: `gcc -o gauss gauss.c`

   Führen Sie Ihr Progamm auf der Kommandozeile aus: `./gauss`
5. Kompilieren Sie das Programm nun wie folgt: `gcc -o gauss.i -E gauss.c`

   Betrachten Sie die Ausgabedatei `gauss.i` mit einem Texteditor. Was ist passiert?
6. Verwenden Sie nun: `gcc -o gauss.S -S gauss.i -masm=intel`

   Können Sie die Ausgabe des Compilers nachvollziehen?

T2.3 Analyse des kompilierten Programms

Im Folgenden werden wir den Maschinencode betrachten, den der Compiler aus einem C-Programm erzeugt hat.

1. Kompilieren Sie das Programm wie folgt: `gcc -o gauss gauss.c`
2. Verwenden Sie nun den Befehl *objdump*, um den Maschinencode der kompilierten Datei in lesbarer Form anzuzeigen: `objdump -d -M intel gauss | less`
3. Suchen Sie in der Ausgabe von *objdump* (der sogenannten Disassembly) nach der Funktion `main`. Können Sie die Schleife aus der Hochsprache im Assemblercode lokalisieren?
4. Kompilieren Sie Ihr Programm erneut unter der Verwendung der Optionen `-O0`[4], `-O1` oder `-O2`. Wie verändert sich die Disassembly?

https://godbolt.org/

Hausaufgaben

P2.1 Collatz [3 Pkt.]

Die Collatz-Vermutung besagt, dass für eine beliebige natürlich Zahl n folgende Transformation immer bei der Zahl 1 herauskommt: wenn n gerade ist, wird $n \leftarrow \frac{n}{2}$ ausgeführt, andernfalls $n \leftarrow 3 \cdot n + 1$. Schreiben Sie in C die Funktion `collatz` mit folgender Signatur, welche die Anzahl der notwendigen Schritte bestimmt, um die Zahl $n = 1$ zu erreichen. Falls dies nie der Fall ist (z.B. bei $n = 0$) oder irgendein n im Verlauf der Berechnung die Größe von 64 Bit überschreitet, soll das Ergebnis 0 sein.

```
uint64_t collatz(uint64_t n)
```

Hausaufgaben

P2.2 EAN-13 Verifier [3 Pkt.]

Implementieren Sie folgende Funktion, welche für eine gegebene EAN genau dann den Wert 1 zurück gibt, wenn die EAN gültig ist, andernfalls den Wert 0. Die EAN wird direkt als Zahl übergeben, d.h. für die EAN 3213213213229 wird `ean=3213213213229` gesetzt. Eine EAN-13 besteht aus 12 Ziffern zur Produktidentifikation und einer Prüfziffer an letzter Stelle. Zur Bestimmung der Gültigkeit werden die 13 Ziffern aufaddiert, wobei Ziffern an ungerader Stelle mit 3 multipliziert werden. Eine EAN-13 ist gültig, wenn diese Summe ein Vielfaches von 10 ist.

```
int ean13(uint64_t ean)
```