LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Arbeitsblatt 2
02.05.2022 - 08.05.2022

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]

Verbindung unter Windows

Seit dem April-Update 2018[2] unterstützt auch Windows *OpenSSH*[3]. Stellen Sie hierzu sicher, dass der *OpenSSH*-Client aktiviert ist.

1. Öffnen Sie die Windows-Einstellungen und navigieren Sie zu *Programme / Optionale Features verwalten*.
2. Falls Sie in der Liste den Punkt *OpenSSH-Client* nicht sehen, klicken Sie auf *Features hinzufügen*. Wählen Sie dann den *OpenSSH-Client* aus und installieren Sie ihn.

Verbindung unter Windows (OpenSSH), macOS, Linux, BSD, ...

1. Öffnen Sie das Terminal Ihres Vertrauens.
2. Verbinden Sie sich per `ssh username@halle.in.tum.de`

   Username: *Ihre Informatik-Kennung, ohne `@in.tum.de`*
   Password: *Ihr Passwort zu der Informatik-Kennung*
3. Beim ersten Verbinden werden Sie gefragt, ob Sie der Verbindung vertrauen möchten. Überprüfen Sie den angezeigten Fingerabdruck (siehe RBG-Wiki); stimmen diese überein, akzeptieren Sie den Key durch das Eintippen von *yes*.
4. Sie sind nun mit der Rechnerhalle verbunden.

[1] `https://wiki.in.tum.de/Informatik/Helpdesk/Ssh`
[2] Bei früheren Windows-Versionen können sie auf das Tool *PuTTY* zurückgreifen.
[3] `https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh_install_firstuse`

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.

```java
public class Gauss {

  public static void main(String [] args) {
    int sum = 0;

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

    System.out.println("Die Summe aller natürlichen Zahlen von 1 bis
        100 beträgt " + sum + ".");
  }
}
```

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

   Der Java-Compiler erstellt aus dem Programmcode Java Bytecode. Dieser kann auf der Java Virtual Machine ausgeführt werden und wird von dieser zur Laufzeit in Maschinencode übersetzt. Der C-Compiler übersetzt den Programcode bereits zur Kompilierzeit direkt in Maschinencode. Während der Maschinencode speziell für den jeweiligen Prozessor (bzw. dessen Architektur) erstellt wurde, ist der Java Bytecode plattformunabhängig. Zur Ausführung wird jedoch immer eine Java Runtime Edition benötigt.

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;
}
```

Zeile 1: include – kopiere Inhalt von Header `stdio.h`, enthält die Deklaration von `printf`
Zeile 2: Keine Klasse, C ist nicht objektorientiert
Zeile 3: Andere Signatur der main-Methode, keine Zugriffsrechte
Zeile 11/12: printf - Konsolenausgabe: Text enthält Platzhalter `%d` für ein Integer, Variable wird als zusätzliches Argument übergeben. `\n` bewirkt einen Zeilenumbruch.
Zeile 14: Rückgabewert: 0 für kein Fehler

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?
   Es wurde lediglich der Präprozessor ausgeführt, welcher Makros und `#include`-Direktiven bearbeitet.
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`
   Sie können die Ansicht von `less` mit der Taste `q` beenden. Die Repräsentation der Ausgabe von *objdump* ist:

   ```
   <address>: instruction_bytes instruction_mnemonic
   ```

   Die lesbare Repräsentation des Programms findet sich in `instruction_mnemonic`.

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?

   Die `main`-Funktion sieht wie folgt aus; die Schleife befindet sich zwischen den Adressen `0x670` und `0x67e`. Zu erkennen ist diese am Sprung zum Compare und dem bedingten Sprung zurück. Das Programm ist einfacher zu verstehen, wenn man sich klar macht, dass die Variable `i` an der Adresse `rbp-0x4` steht und die Variable `sum` an der Adresse `rbp-0x8`.

```
000000000000064a <main>:
// ...
659: c745f800000000  mov DWORD PTR [rbp-0x8],0x0  //sum = 0
660: c745fc00000000  mov DWORD PTR [rbp-0x4],0x0  //i = 0
667: c745fc01000000  mov DWORD PTR [rbp-0x4],0x1  //i = 1
66e: eb0a            jmp 67a <main+0x30>          //Sprung zu Vgl.
670: 8b45fc          mov eax,DWORD PTR [rbp-0x4]  //eax = i
673: 0145f8          add DWORD PTR [rbp-0x8],eax  //sum = sum + eax
676: 8345fc01        add DWORD PTR [rbp-0x4],0x1  //i = i + i
67a: 837dfc64        cmp DWORD PTR [rbp-0x4],0x64 //Vergleich i,100
67e: 7ef0            jle 670 <main+0x26>      //Sprung, wenn i<=100
// ...
69c: c3              ret
```

4. Kompilieren Sie Ihr Programm erneut unter der Verwendung der Optionen `-O0`[4], `-O1` oder `-O2`. Wie verändert sich die Disassembly?

   Bei höheren Optimierungsstufen (GCC unterstüzt die Stufen bis `-O3`) fällt auf, dass die Schleife nicht mehr existert; sie wurde zur Compile-Zeit berechnet und im Programm durch eine Konstante ersetzt.

   Als Vorgriff: Zudem werden nicht alle Variablen ständig auf dem Stack gespeichert und wieder geladen.

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)
```

[4] Minus, großer Buchstabe O, Null

Aufgabentester: Nutzen Sie den Aufgabentester, um diese Funktion zu entwickeln; oder entwickeln Sie lokal mit einer selbst erstellten Vorlage.

Referenzlösung:

```
uint64_t collatz(uint64_t n) {
  long r=0;
  while (n > 1) {
    r++;
    if((n&1) && n >= UINT64_MAX/3)
      return 0;
    n = n&1 ? 3*n+1 : n>>1;
  }
  return r;
}
```

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)
```

Aufgabentester: Nutzen Sie den Aufgabentester, um diese Funktion zu entwickeln; oder entwickeln Sie lokal mit einer selbst erstellten Vorlage.

Referenzlösung:

```
int ean13(uint64_t ean) {
  int sum = 0;
  for (int i = 0; i < 13; i++) {
    sum += (ean % 10) * (i & 1 ? 3 : 1);
    ean /= 10;
  }
  return ean == 0 && sum % 10 == 0;
}
```

Q2.1 Quiz [4 Pkt.] (siehe Praktikumswebsite)

X2.1 Analyse von Disassembly

Betrachten Sie die folgende Disassembly einer Funktion, die einen Parameter in `rdi` entgegen nimmt. Was wird hier berechnet? Versuchen Sie zunächst, die Funktion händisch auf einigen (bevorzugt binär notierten) Beispieleingaben zu berechnen.

```
00000000000016f0 <op0>:
    16f0: 48 ba 55 55 55 55 55    movabs rdx,0x5555555555555555
    16f7: 55 55 55
    16fa: 48 89 f8                mov    rax,rdi
    16fd: 48 d1 e8                shr    rax,1
    1700: 48 21 d0                and    rax,rdx
    1703: 48 ba 33 33 33 33 33    movabs rdx,0x3333333333333333
    170a: 33 33 33
    170d: 48 29 c7                sub    rdi,rax
    1710: 48 89 f8                mov    rax,rdi
    1713: 48 c1 ef 02             shr    rdi,0x2
    1717: 48 21 d0                and    rax,rdx
    171a: 48 21 d7                and    rdi,rdx
    171d: 48 01 c7                add    rdi,rax
    1720: 48 89 f8                mov    rax,rdi
    1723: 48 c1 e8 04             shr    rax,0x4
    1727: 48 01 f8                add    rax,rdi
    172a: 48 bf 0f 0f 0f 0f 0f    movabs rdi,0xf0f0f0f0f0f0f0f
    1731: 0f 0f 0f
    1734: 48 21 f8                and    rax,rdi
    1737: 48 bf 01 01 01 01 01    movabs rdi,0x101010101010101
    173e: 01 01 01
    1741: 48 0f af c7             imul   rax,rdi
    1745: 48 c1 e8 38             shr    rax,0x38
    1749: c3                      ret
```

Diese Funktion zählt die Anzahl der gesetzten Bits in `rdi`. Dieser und weitere* Bit Twiddling Hacks *finden sich auf der Website von Sean Anderson.[5]

[5] `https://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel`