Aufgabe 1: Fachbereichsaccount

In den meisten Veranstaltungen hier benötigen Sie einen Fachbereichsaccount. Diesen erstellen Sie, indem Sie auf `https://portal.mi.fu-berlin.de` gehen, sich dort mit Ihren ZEDAT-Logindaten anmelden, und unter „Account am Fachbereich registrieren“ die Bedingungen akzeptieren und auf „registrieren“ klicken.

Weitere Informationen gibt es unter `http://www.mi.fu-berlin.de/w/IT/Computeraccess`.

Aufgabe 2: Zugriff auf Linux-Umgebung

In diesem Kurs werden unsere Programmieraufgaben eine Linux-Umgebung voraussetzen. Da wir euren Code auf dem Linux-Sever „andorra“ der Uni testen, empfehlen wir euch, auch selber direkt auf diesem zu arbeiten. Hierzu werden wir uns von eurem System (Client) auf den andorra (Server) verbinden.

1. Öffnen sie eine Kommandozeile (`cmd.exe` oder Powershell auf Windows, Terminal auf macOS oder Linux).

2. Geben sie nun folgendes nach dem `$` ein und drücken sie Enter:

   ```
   $ ssh ZEDAT_USER_NAME@andorra.imp.fu-berlin.de
   ```

3. Wenn Sie sich zum ersten Mal verbinden, müssen sie noch diese Verbindung einmalig mit `yes` bestätigen, das sieht bspw. so aus:

   ```
   $ ssh koenigl@andorra.imp.fu-berlin.de
   The authenticity of host 'andorra.imp.fu-berlin.de ...
   ECDSA key fingerprint is ...
   Are you sure ... (yes/no/[fingerprint])? yes
   Warning: Permanently added 'andorra ...
   ```

4. Danach geben Sie ihr Passwort ein. Achtung, hierbei gibt es keinerlei „visuelle Rückmeldung“, Sie müssen quasi blind eingeben. Danach landen Sie wieder auf einer Konsole – nur liegt diese auf dem Server:

   ```
   koenigl@andorra.imp.fu-berlin.de's password:
   koenigl@andorra:~$
   ```

Als nächstes möchten wir auf der Linux Konsole navigieren. Nach dem Login sind Sie in ihrem Heimatverzeichnes („Eigene Dateien“).

1. Zeigen Sie sich alle Dateien und Ordner an indem Sie den Befehl „List“ aufrufen:

   ```
   $ ls
   ```

   Bei manchen von Ihnen existieren hier bereits Dateien und Ordner, bei manchen ist die Ausgabe „leer“.

2. Erstellen die Ordner „Bilder“ und „Dokumente“ mittels „Make Directory“:

   ```
   $ mkdir Bilder
   $ mkdir Dokumente
   ```

Nun können Sie das Ergebnis mit `ls` sich anzeigen lassen, jetzt sollten diese Ordner erstellt sein.

3. Wechseln Sie mittels des „Change Directory“ Befehls in Ihren „Bilder“ Ordner:

```
$ cd Bilder
```

4. Wechseln Sie nun wieder in der Hierarchie einen Schritt „nach oben“ indem Sie `cd` das Argument `..` übergeben:

```
$ cd ..
```

5. Wechseln Sie nun in den „Dokumente“ Ordner:

```
$ cd Dokumente
```

Übrigens, Sie hätten auch statt der letzten beiden Befehle schreiben können:

```
$ cd ../Dokumente
```

Der `/` ist ein sog. Pfad-Separator.

6. Erstellen Sie in diesem Ordner einen weiteren Ordner „Rechnerarchitektur“ mittels des „Make Directory“ Befehls und bewegen Sie sich in den Ordner:

```
$ mkdir Rechnerarchitektur
$ cd Rechnerarchitektur
```

7. Öffnen Sie den Konsolenbasierten Editor `nano` und erstellen Sie damit eine Testdatei `test` im aktuellen Ordner Sie können auch einen anderen Dateinamen wählen oder eine Dateiendung wie `.txt` benutzen – aber Dateiendungen sind für Linux weitestgehend egal.

```
$ nano test
```

Unten in dem Terminalfenster sehen Sie eine kleine Kurzanleitung wie Sie mit dem Editor umgehen. Hierbei steht das Symbol `^` für die Steuerung-Taste (Ctrl), sprich um die Datei zu speichern, drücken Sie Ctrl+O und bestätigen den Dateinamen mit Enter.

8. Probieren Sie sich ein wenig aus mit dem Editor. Wenn Sie möchten, können Sie auch andere Editoren ausprobieren, weiterhin sind installiert `vi`/`vim` und `emacs`.

9. Legen Sie nun mittels „Copy“ eine Kopie Ihrer Datei an:

```
$ cp test test_kopie
```

10. Hilfreich ist es auch, Dateien zu verschieben bzw. umzubennen zu können. Dazu nutzen Sie den „Move“ Befehl:

```
$ mv test_kopie neuer_name
```

Es ist immer Hilfreich mittels `ls` sich anzugucken, welche Dateien in ihrem aktuellen Ordner sind.

Schaffen Sie es, die Datei `neuer_name` eine Ordnerhierarchie nach oben zu verschieben?

11. Zuguterletzt, löschen Sie alle Ihre angelegten Dateien, mittels „Remove“ – Achtung, hierbei wird kein „Papierkorb“ oder ähnliches genutzt!

```
$ rm test
$ rm neuer_name    # falls nicht nach "oben" verschoben
$ rm ../neuer_name # falls nach oben verschoben
```

    Um Ordner zu löschen können Sie `rmdir` benutzen, das funktioniert jedoch nur mit leeren Ordnern. Abhilfe schafft `rm -r` welches *rekursiv* alle Unterordner und Dateien löscht und dann den Ordner selbst. Passen Sie dabei gut auf, `rm -r` hat schon zu vielen Datenverlusten geführt!

Es wird manchmal nötig sein, von ihrem persönlichen Rechner Dokumente auf den Uni-Server zu kopieren bzw. andersherum, von diesem Dateien auf Ihren eigenen Rechner zu übertragen. Wärend `cp` uns erlaubt innerhalb eines Systems Dateien zu kopieren, bietet das auf SSH aufbauende `scp` uns an, Dateien von einem System auf ein anderes zu kopieren.

1. Laden Sie hierzu erst die Datei `sample_wrapper.c` aus dem KVV (Anhang dieser Übung) herunter.

2. Öffnen Sie nun wieder eine Kommandozeile und verbinden sich innnerhalb dieses Fensters *nicht* via SSH. Navigieren Sie stattdessen zu dem Ort, wo die heruntergeladene Datei liegt. Mit meinem Nutzernamen wäre das bspw.:

```
$ cd C:\Users\koenigl\Downloads
```

3. Kopieren Sie nun die Datei in den Ordner `Dokumente/Rechnerarchitektur/` auf dem Server (die ... durch die komplette Domain ersetzen, und alles in eine Zeile, den `\` nicht dazu schreiben):

```
$ scp sample_wrapper.c                                    \
  ZEDAT_USER_NAME@and...de:Dokumente/Rechnerarchitektur
```

    Wenn sie „Permission denied“ bekommen, stellen Sie sicher, dass sie den Ordner auf dem Server korrekt erstellt haben.

    Sie hätten auch an Stelle dieser zwei Befehle das wieder in einem machen können, indem Sie anstelle von `sample_wrapper.c` den gesamten Dateipfad angeben:

```
C:\Users\koenigl\Downloads\sample_wrapper.c
```

    Enthält dieser Leerzeichen, ist es am Einfachsten, den Pfad in Anführungszeichen zu setzen.

4. Um von Andorra auf Ihren Rechner zu kopieren, führen Sie auf Ihrem Rechner den gleichen Befehl aus – jedoch mit den Argumenten in der vertauschten Reihenfolge.

Aufgabe 3: Erstes Assembly-Programm

Wir haben uns nun ein wenig mit dem Terminal uns seinem Interface, genannt „Shell“, auseinandergesetzt. Nun schreiben wir aber unser erstes, *sehr kleines*, Assembly-Programm.

1. Erstellen Sie eine Datei, bspw. mit dem Namen `sample.asm`, in dem üblichen Ordner `Dokumente/Rechnerarchitektur` mit ihrem präferierten Editor, und tippen folgenden Code ein:

   ```
                global tri_area

   tri_area:    mov rax, 42
                ret
   ```

   Speichern Sie die Datei.

2. Wir werden nun diesen Code von der menschenlesbaren Assemblysprache in Maschinensprache übersetzen (kompilieren, genauer assemblieren). Hierzu rufen wir den Netwide Assembler auf:

   ```
   $ nasm -f elf64 -o sample.o sample.asm
   ```

   Das `-o sample.o` gibt an, dass der Maschinencode „output“ in die Datei `sample.o` geschrieben werden soll.

   Es gibt auch andere Assembler mit anderen Assemblersprachen für unser System (d.i., Linux AMD64) wie den GNU/as (`gas`), diese nutzen wir jedoch nicht.

3. Unser Programm ist bisher jedoch unvollständig, da das Betriebssytem noch ein bisschen Logik „drumherum“ benötigt. Diese haben wir euch in unserem, in der Sprache C geschriebenen, Wrapper bereitgestellt – und ihr habt ihn bereits auf das System kopiert. Wir werden nun auch diesen kompilieren:

   ```
   $ c99 -O2 -c -o sample_wrapper.o sample_wrapper.c
   ```

4. Jetzt liegen beide unsere Programmteile `sample.o` und `sample_wrapper.o` in maschinenlesbarer Form vor, wir müssen sie nurnoch zusammen „linken“:

   ```
   $ c99 -o sample sample_wrapper.o sample.o
   ```

   Diesmal rufen wir wieder den C-Compiler auf, jedoch ohne die Flag `-c` (`-O2` hat hier keine Auswirkungen).

5. Zu guter letzt können wir unser Programm ausführen:

   ```
   $ ./sample
   Provide exactly two arguments!
   ```

   Wie wir sehen, beschwert sich unser C-Wrapper darüber, dass wir keine Argumente übergeben haben. Da ich das Programm geschrieben habe, weiß ich, dass wir zwei Ganzzahlen übergeben müssen:

   ```
   $ ./sample 3 4
   tri_area(3, 4) = 42
   ```

Unser Programm berechnet uns, dass ein Dreieck mit der Länge der Grundseite von 3 und der Höhe von 4 eine Fläche von 42 hat. Hm. Nicht ganz korrekt, aber das wird wohl daran liegen, dass unsere in Assembly geschriebene Funktion `tri_area` immer den Wert 42 zurückgibt.

Schreiben wir nun die Funktion so, dass sie korrekt funktioniert. Wir erinnern uns, die Fläche eines Dreiecks berechnet sich aus der Grundseite g und der Höhe h wiefolgt:

$$A = \frac{g \cdot h}{2}$$

Wie können wir nun diese Formel in Assembly schreiben? In Assembly haben wir sog. Register, das sind bei uns 64 bit breite „Speicher“, wovon wir eine Handvoll haben. Unsere Programmargumente liegen in genau solchen Registern, und zwar die Grundseite in dem Register mit dem Name `rdi`, die Höhe in `rsi`.

1. Wir möchten nun `rdi` mit `rsi` multiplizieren. Hierfür gibt es tatsächlich den Befehl `mul`, jedoch kann dieser leider nicht beliebige Register multiplizieren – ein Faktor *muss* das Register `rax` sein! Kopieren wir also den Wert der Grundseite in `rdi` nach `rax` mit „move“. Unser Code sieht jetzt so aus:

```
            global tri_area

tri_area:   mov rax, rdi     ; rax := rdi
            ret
```

   Es ist *essentiell* unseren Code regelmäßig zu testen, wir assemblieren also unteren Assembly-Code `sample.asm` zu einer neuen `sample.o`. Dadurch müssen wir unser Programm `sample` natürlich auch neu Linken. Führen wir unseren Code jetzt aus, merken wir, dass unser Programm immer die Grundseite als Ergebnis zurückgibt!

   Tipp: Um nicht jedes Mal den gleichen Befehl erneut einzugeben, könnt ihr die Pfeiltaste hoch nutzen um durch die letzten Befehle zu scrollen. Zudem könnt ihr auf der Konsole Dateinamen mit der Tab-Taste vervollständigen.

2. Multiplizieren wir nun die Grundseite in `rax` mit der Höhe `rsi`. Da der Operand `rax` ja festgeschrieben ist, geben wir lediglich den anderen Faktor an:

```
            global tri_area

tri_area:   mov rax, rdi     ; rax := rdi
            mul rsi          ; rdx:rax := rax * rsi
            ret
```

   Wieder testen wir unseren Code und wir werden sehen, es kommt tatsächlich $g \cdot h$ heraus! Das bedeutet, `mul x` weißt dem Register `rax` das Ergebnis aus $x \cdot$ rax zurück.

   Eine Sache muss aber noch angemerkt werden: Da eine Multiplikation zweier 64 bit Register einen Wert der Größe 128 bit produzieren kann, wird dieser aufgeteilt. Lediglich die untere Hälfte wird in das Register `rax` geschrieben, die obere Hälfte in das Register `rdx`. Solange unsere Zahl also klein genug ist, sodass sie mit 64 bit darstellbar ist, werden in `rdx` nur führende Nullen produziert.

3. Fehlt noch die Division durch 2. Analog zur Multiplikation wird immer das Register `rax` dividiert und wir können nur den Divisor angeben. Aber Achtung: Wieder wird eigentlich nicht `rax` dividiert, sondern eine 128 bit Zahl die sich zusammensetzt aus den Registern `rdx` für die obere Hälfte und `rax` für die untere. Falls wir nur eine 64 bit Zahl dividieren möchten, müssen wir also das Register `rdx` auf den Wert 0 setzen!

   Führen wir nun die Division aus:

```
            global tri_area

tri_area:   mov rax, rdi     ; rax := rdi
            mul rsi          ; rdx:rax := rsi * rax

            ; not neccessarily needed here:
            ;mov rdx, 0

            mov rcx, 2       ; rcx := 2
            div rcx          ; rax := rcx/2, rdx := rcx % 2

            ret
```

   Wir haben nun ein funktionierendes Programm!

   Aber woher weiß unser C-Wrapper, dass er die Programmargumente in die Register `rdi` und `rsi` kopieren muss und wir unser Ergebnis in `rax` ablegen? Das bestimmt die sogenannte Calling-Convention. In unserem Fall ist das die System V AMD64 ABI – mehr dazu ein anderes Mal.