Übungsblatt 5

Dr. Matthias Frank, Dr. Matthias Wübbeling

Ausgabe Mittwoch, 8. November 2023
Abgabe bis **Freitag, 17. November 2023, 23:59 Uhr**
Vorführung vom 20. bis zum 24. November 2023

Alle Programme müssen unter **Ubuntu 22.04** kompilierbar bzw. lauffähig und ausreichend kommentiert und mit **Makefile** (C, Assembler) versehen sein, um Punkte zu erhalten. Als Compiler sollen **clang** (C) und **nasm** (Assembler) verwendet werden. Die Lösungen sind bei Ihrem Tutor während Ihrer Übungsgruppen vorzuführen. Alle Gruppenmitglieder sollten die Abgabe erklären können. Die Abgabe erfolgt mittels Ihres Git-Repositories und der vorgegebenen Ordnerstruktur.

Die Punkte der Aufgaben sind relevant für die Zulassung. Die Punkte der Bonusaufgaben werden auf Ihren Punktestand addiert, werden aber nicht auf die für die Zulassung benötigten Punkte addiert.

Abgabestruktur: Jede Abgabe, die die folgende Struktur nicht umsetzt, wird **NICHT** bepunktet bzw. mit 0 Punkten bewertet

- die zu korrigierenden Lösungen müssen bis zur Deadline auf dem `master`-Branch liegen. Lösungen auf anderen Branches werden nicht gewertet
- alle Lösungen müssen in der vorgegebenen Ordnerstruktur (**blattXX/aufgabeYY**) abgelegt werden, wobei `XX` und `YY` durch die jeweiligen Nummern des Zettels und der Aufgaben ersetzt werden sollen. Der Name soll exakt nur aus diesen Zeichen bestehen und achtet auf Kleinschreibung
- sämtliche Aufgaben, mit Ausnahme der theoretischen Aufgaben, die eine PDF erfordern, benötigen zwingend ein **Makefile**. Abgaben ohne dieses werden nicht gewertet

Aufgabe 1 Fibonacci-Zahlen und 64-Bit-Assembler – Rekursion (6 Punkte)

In dieser Aufgabe sollen Sie eine *rekursive* Funktion zur Berechnung von Fibonacci-Zahlen in 64-Bit-Assembler schreiben. Eine Fibonacci-Zahl $f_n, n \in \mathbb{N}_0$, ist definiert durch

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-2} + f_{n-1}, & \text{sonst.} \end{cases} \tag{1}$$

a)

Gegeben ist folgender Quellcode in 64-Bit-Assembler (auch eingebettet in die PDF-Datei):

```
SECTION .data

; define the fibonacci number that should be calculated
n:        dq        10

SECTION .text

global _start

_start:
        ; call Fibonacci function f(n)
        mov rdi, [n] ; parameter: fibonacci number to calulculate
        call f ; call function

        ; print calculated Fibonacci number on stdout
        call printnumber

        ; exit process with exit code 0
        mov rax, 60
        mov rdi, 0
        syscall

; f: Calculates a Fibonacci number
; f(n) = {n, if n <= 1; f(n - 1) + f(n - 2), else}
; Parameter: Integer n >= 0, passed using AMD64 ABI convention
; Returns: Fibonacci number f(n), returned in rax
f:
        ; ...
        ret
```

Ergänzen Sie die Funktion `f` so, dass sie die Fibonacci-Zahl $f(n)$ rekursiv berechnet. Der Parameter n soll gemäß AMD64 ABI Calling Convention an die Funktion übergeben werden. Das Ergebnis soll im Register `rax` zurückgegeben werden. Zur Ausgabe verwenden Sie idealerweise die Funktion `printnumber`, die Sie auf einem vorherigen Zettel implementiert haben.

Hinweis:

- Sollten Sie die Funktion `printnumber` nicht geschafft haben, so können Sie eine alternative Ausgabe über den Return-Wert des Programms nutzen. Dazu muss kurz vor Programmende statt einer 0 das Ergebnis der Rechnung in das Register `rdi` geschrieben werden. Es kann dann auf der Konsole nach Programmende mit

  ```
  echo $?
  ```

  der Return-Wert ausgelesen werden.
- Sie können zur Verwaltung von Stackframes die Befehle `enter` und `leave` benutzen.

b)

Betrachten Sie allgemein die Programmierung von rekursiven Funktionen in Assembler. Was müssen Sie bezüglich der Verwendung von Registern und dem Stack grundsätzlich beachten? Was genau tun die Befehle `enter` und `leave`?

c)

Skizzieren Sie den Stack für $n = 3$ direkt vor dem Aufruf der ersten Rekursion (also $f(2)$ oder $f(1)$. Zeichnen Sie auch den Basepointer `rbp` und den Stackpointer `rsp` ein.

Aufgabe 2 Adressierung in x86-Assembler (4 Punkte)

Häufig müssen Sie in Assembler Daten aus dem Speicher lesen oder in den Speicher schreiben. Während RISC-Architekturen wie MIPS oder ARM oder auch die aus der Systemnahen Informatik bekannte Alpha-Notation dafür nur direkte LOAD- und STORE-Operationen zur Verfügung stellen, bei denen die Adresse fertig berechnet in einem Register liegen muss, bietet x86 sehr komplexe Möglichkeiten, den Speicher zu adressieren. Diese Möglichkeiten wurden Ihnen in Kapitel 1.0 ab Folie 38 vorgestellt.

Im Folgenden sollen Sie stets eine Funktion implementieren, die ein gegebenes Problem lösen soll. Achten Sie darauf, die AMD64-ABI einzuhalten, und beachten Sie die zusätzlichen Hinweise und Einschränkungen zu jeder Aufgabe. Für jedes Problem erhalten Sie außerdem ein Testprogramm auf der Webseite oder eingebettet in der PDF-Datei, mit dem Sie Ihre Funktion linken und testen sollen.

Aufgabe 2.1

`int64_t is_sorted(int64_t* array, uint64_t num_elements)` – Sie erhalten ein Array `array`, das aus vorzeichenbehafteten 64-Bit-Zahlen besteht. Die Anzahl der Elemente des Arrays bekommen Sie über den zweiten Parameter `num_elements` gegeben.

Sofern die Zahlen aufsteigend sortiert sind (nicht streng monoton, also jede Zahl größer oder gleich der vorangegangenen ist), geben Sie `1` zurück; ansonsten geben Sie `0` zurück.

Laden Sie in jedem Schleifendurchlauf das aktuelle Element mit einer einzelnen Instruktion in ein Register. Verwenden Sie die Instruktion `CMP` für den Vergleich. Erklären Sie insbesondere, wie die `CMP`-Instruktion funktioniert.

Aufgabe 2.2

`int64_t dot_product(int64_t* a, int64_t* b, uint64_t num_elements)` – Sie erhalten zwei Vektoren als Arrays von vorzeichenbehafteten 64-Bit-Zahlen `a` und `b`. Die Anzahl der Elemente der Arrays (also die Dimension n der Vektoren) erhalten Sie über den zweiten Parameter `num_elements`. Berechnen Sie das Skalarprodukt der beiden Vektoren und geben Sie es zurück.

Das Skalarprodukt zweier Vektoren a und b ist definiert als die Summe der Produkte der Einträge beider Vektoren, also

$$a \cdot b := \sum_{i=1}^{n} a_i \cdot b_i \text{ für } a, b \in \mathbb{R}^n.$$

Verwenden Sie für die Schleife die `LOOP`-Instruktion. Den Wert des Eintrags aus `a` laden Sie mit der `MOV`-Instruktion; den Wert aus `b` laden Sie direkt im `IMUL`-Befehl. Berechnen Sie dafür die Adresse vorher mit der `LEA`-Instruktion.

Erklären Sie die Funktionsweise der `LOOP`- und `LEA`-Instruktion.

> Hinweis: Bei der Adressierung kann es nötig sein, alle Teile der Adressierung zu verwenden, also
>
> ```
> cmd foo, [base + index * scale + disp] .
> ```

Sie dürfen annehmen, dass die Produkte alle klein genug sind, um in eine vorzeichenbehaftete 64-Bit-Zahl zu passen. Das selbe dürfen Sie für das Ergebnis annehmen, das heißt, Sie müssen keine Overflows und ähnliche Sonderfälle beachten.

Aufgabe 3 Analyse von `fork()` (2 Punkte)

In dieser Aufgabe sollen Sie sich mit der Verwendung der Systemfunktion `fork()` vertraut machen. Betrachten Sie dazu das folgende C-Programm, das Sie auch auf der Website zur Vorlesung herunterladen oder aus der PDF-Datei entnehmen können. Führen Sie es mehrmals hintereinander aus.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
        int cnt = 0;
        int fork_pid = getppid();
        printf("cnt = %d, fork_pid = %5d, pid = %5d\n", cnt, fork_pid, getpid());
        fork_pid = fork();
        cnt++;
        printf("cnt = %d, fork_pid = %5d, pid = %5d\n", cnt, fork_pid, getpid());
        fork_pid = fork();
        cnt++;
        printf("cnt = %d, fork_pid = %5d, pid = %5d\n", cnt, fork_pid, getpid());
}
```

1. Zeichnen Sie den Prozessbaum für das C-Programm.
2. Wie viele Ausgabezeilen generiert der Aufruf des Programms? Erläutern Sie anhand des Programmtexts für jede Zeile der bei Ihnen erzeugten Ausgabe, wie sich die Werte für die ausgegebenen Variablen ergeben. Geben Sie dazu die bei Ihnen erzeugte Ausgabe an.
3. Erklären Sie, warum sich die Ausgabe bei wiederholtem Aufruf des Programms manchmal verändert.