Übungsblatt 4

Dr. Matthias Frank, Dr. Matthias Wübbeling

Ausgabe Mittwoch, 1. November 2023
Abgabe bis **Freitag, 10. November 2023, 23:59 Uhr**
Vorführung vom 13. bis zum 17. 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 Assembler Debugging mit gdb (3 Punkte)

gdb eignet sich nicht nur zum Debugging von C-Programmen, sondern bietet auch einige interessante Features zur Analyse von Assembler-Code. Hierzu eignet sich insbesondere das `text user interface` oder `tui`. Dieses ermöglicht gleichzeitigen Einblick in den Quellcode, Register oder Maschinenbefehle. Der tui-Modus lässt sich auf verschiedene Weisen aktivieren, z.B. durch Aufruf von `gdb -tui <file>` oder den Befehl `(gdb) layout <type>`. Eine Übersicht über die verfügbaren Layouts befindet sich im gdb User Manual unter TUI/Commands. Wir empfehlen `(gdb) layout reg` für das Debugging von SysProg-Übungsaufgaben. Beachten Sie, dass das ASM-Fenster standardmäßig AT&T-Syntax verwendet und mit `(gdb) set disassembly intel` auf Intel-Syntax umgeschaltet werden kann.

Korrigieren Sie nun mithilfe gdb das Programm `average.asm`, welches den (abgerundeten) Durchschnitt einiger Zahlen berechnen soll:

```
        global   main
        section  .text
main:
        lea      rsi, [array]           ; load start of array
        mov      rcx, [count]           ; load number of elements
        dec      rcx                    ; n-th element = array[n-1]
accumulate:
        mov      rax, [rsi+rcx*4]       ; load i-th element
        add      [sum], rax             ; accumulate sum as we go
        loop     accumulate             ; more numbers?
average:
        mov      rax, [sum]             ; prepare dividend
        div      qword [count]          ; rax is integer quotient, rdx is remainder
        ret

        section  .data
array:  dq       89, 10, 67, 1, 4, 27, 12, 34, 85, 1
count:  dq       10
sum:    dq       0
```

Tipp: Sie können das Programm folgendermaßen kompilieren/assemblieren und linken:

```
nasm -f elf64 -g -F dwarf -o average.o average.asm
ld -e main -o average average.o
```

> Hinweis: Üblicherweise wird in NASM das Symbol `_start` für den Einstiegspunkt in ein Programm verwendet. Der Linker (ld) erlaubt aber die Spezifizierung beliebiger Labels mit dem Parameter `-e`. Überlegen Sie, für welche Anwendungsfälle dies nützlich sein könnte.

Aufgabe 2 Matrixaddition (4 Punkte)

Betrachten Sie das folgende C-Programm, welches Matrizen addiert:

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

void print_matrix(int n, int m, int* M);
void m_add(int n, int m, int* M1, int* M2, int* M);

int main() {
        int A[] = { /* 3x4-Matrix */
                12,2,-3,5,
                0,19,5,9,
                2,-1,13,-3
        };
        print_matrix(3, 4, A);
        int B[] = { /* 3x4-Matrix */
                30,2,-1,-47,
                0,23,2,-2,
                7,-5,-55,9
        };
        print_matrix(3, 4, B);
        int C[12]; /* 3x4-Matrix */
        m_add(3, 4, A, B, C); /* C = A + B */
        print_matrix(3, 4, C);
        return 0;
}

void print_matrix(int n, int m, int* M) {
        int x;
        int y;
        printf("\n");
        for (y=0; y < n; y++) {
                for (x = 0; x < m; x++) {
                        printf("%5d", M[y*m+x]);
                }
                printf("\n");
        }
        printf("\n");
}

void m_add(int n, int m, int* M1, int* M2, int* R) {
        int x;
        int y;
        for (y = 0; y < n; y++) {
                for (x = 0; x < m; x++) {
                        R[y*m+x] = M1[y*m+x] + M2[y*m+x];
                }
        }
}
```

Das Programm finden Sie auch auf der Veranstaltungs-Homepage zum Download oder einge-

bettet in die PDF-Datei.

Entfernen Sie aus diesem Programm die Funktion `void m_add(...)` und schreiben Sie eine entsprechende Funktion in einer separaten Datei in Assembler. Linken Sie anschließend die Assembler-Funktion mit dem verbleibenden C-Programm zu einer Binary und überprüfen Sie damit die korrekte Funktionsweise Ihrer Assembler-Routine. Angenommen, Ihre C-Quellcodedatei heißt **matrix.c** und Ihre Assembler-Quellcodedatei heißt **m_add.S**, so können Sie die beiden Dateien folgendermaßen kompilieren/assemblieren und linken:

```
nasm -f elf64 -o m_add.o m_add.S
clang -o matrix m_add.o matrix.c
```

Verwenden Sie die Aufrufkonvention AMD64 ABI.

Hinweis: Beachten Sie, dass neben Werten auch Pointer beim Funktionsaufruf übergeben werden. Bei diesen Pointern auf die Matrizen handelt es sich um Werte, die Sie direkt als Basis für eine indirekte Adressierung verwenden können. Bedenken Sie dabei auch, dass die Adressierung auf Grundlage von Bytes erfolgt.

Aufgabe 3 Fehlersuche in Programmen (2 Punkte)

In das folgende Programm haben sich einige syntaktische und semantische Fehler eingeschlichen, die Sie erkennen und korrigieren sollen.

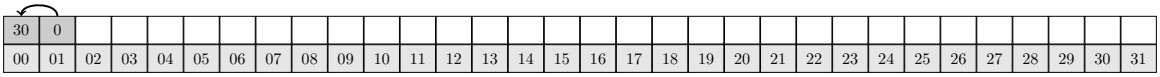
Finden und korrigieren Sie die vorhandenen Fehler. Beseitigen Sie alle Warnungen (`-Wall -Wextra`), die beim Kompilieren erzeugt werden. Protokollieren Sie die Fehler in einer Datei namens **README.txt**. Das folgende Programm ist auch in die PDF-Datei eingebettet:

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

int palindrom_check1(const char *st1) {
        int st1_len = strlen(st1), i;
        for(i = 0; i < st1_len / 2; i++)
                if (toupper(st1[i]) != toupper(st1[st1_len - i])) return 0;
        }
        return 1;
}

int main(void) {
        char *st1, *st2;
        int st1_len, i;
        char *exit = 'exit';
        st1 = malloc(100);
        st2 = malloc(100);
        printf("Zum Beenden exit als Eingabe angeben. \n");
        while (1) {
                printf("Eingabe: ");
                scanf("%s", st1);
                if (strcmp(st1, exit) == 0) return 0;
                st1_len = strlen(st1);
                for (i=0; i<st1_len; i++) {
                        st2[i] = toupper(st1[st1_len - 1 - i]);
                }
                if (palindrom_check1(st1)) printf("1. Test: Palindrom \n");
                else printf("1. Test: Kein Palindrom \n");
                if (st1 == st2) printf("2. Test: Palindrom\n");
                else printf("2. Test: Kein Palindrom\n");
        }
        return 0;
}
```

Aufgabe 4 Speicherverwaltung (3 Punkte)

In dieser Aufgabe sollen Sie die Verwaltung des Arbeitsspeichers mit `malloc` und `free` übernehmen. Nehmen Sie dafür an, dass der betreffende Teil des Arbeitsspeichers 32 Byte umfasst und byteweise mit Adressen von 0–31 adressiert wird. Programminstruktionen und die unten verwendeten Zeiger (`p`, `q`, ...) werden nicht in diesem Speicherbereich abgelegt, so dass er ganz für `malloc` zur Verfügung steht. Adressen (also auch Zeiger) sind hier 8 Bit (1 Byte) lang. Die Methoden `malloc` und `free` arbeiten wie in der Beispielumsetzung in der Vorlesung vorgestellt. Insbesondere liegt vor jedem von `malloc`/`free` verwalteten Speicherblock ein Header. Dieser Header soll hier aus zwei Byte bestehen: Das erste Byte beschreibt die Größe des jeweiligen Speicherblocks (Header nicht mitgerechnet), das zweite Byte enthält im Falle eines freien Speicherblocks einen Zeiger auf den Header des nächsten Speicherblocks in der Free-Liste.

Zu Beginn besteht der gesamte Speicher aus einem einzigen freien Block, und sieht deshalb folgendermaßen aus:

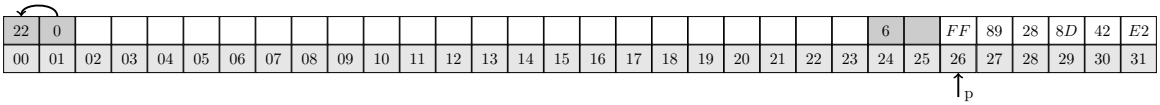


Jedes Kästchen steht für ein Speicher-Byte; bei Kästchen ohne Eintrag ist der Inhalt unbestimmt. Der Übersichtlichkeit halber werden hier die Zeigerfelder freier Speicherblöcke zusätzlich durch Pfeile veranschaulicht.

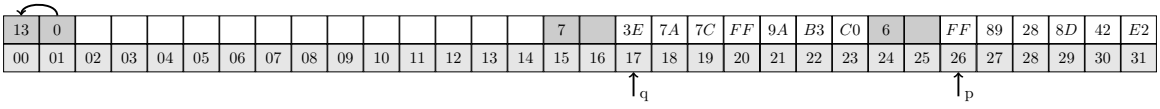
Im Folgenden wird eine Reihe von Anweisungen auf diesem System ausgeführt. Wir geben die Veränderungen, die die ersten drei Schritte im Speicher zur Folge haben, zur Veranschaulichung vor. Setzen Sie dies analog für die darauffolgenden sechs Schritte (d)–(i) fort (0,5 Punkt je Schritt).

> Hinweis: Sie können die Lösung zu dieser Aufgabe in einem (sinnvollen) Dateiformat Ihrer Wahl – Grafikdatei, gescannte handschriftliche Lösung, ASCII-Art, ... – ins Git hochladen. Achten Sie aber bitte darauf, dass die Dateigröße pro Seite unter 5 MB bleibt.

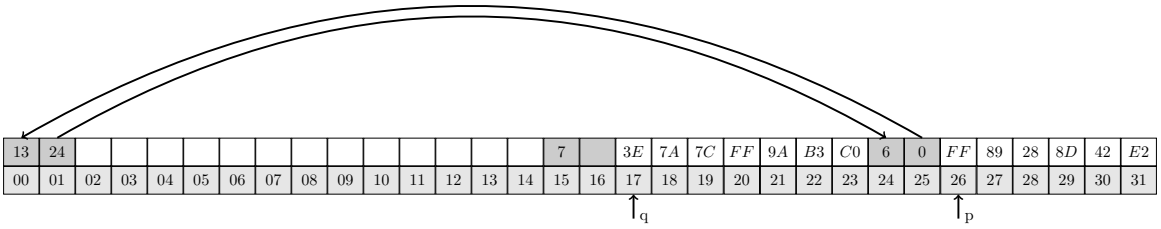
1. `p = malloc(6)` und schreibe `FF 89 28 8D 42 E2` an die Stelle, auf die `p` zeigt



2. `q = malloc(7)` und schreibe `3E 7A 7C FF 9A B3 C0` an die Stelle, auf die `q` zeigt



3. `free(p)`



4. `r = malloc(2)` und schreibe `BE EF` an die Stelle, auf die `r` zeigt
5. `free(q)`
6. `s = malloc(12)` und schreibe `A7 AF C4 7F 03 90 CD 13 A0 F5 49 00` an die Stelle, auf die `s` zeigt
7. `free(p)` – Ja, hier hat der Programmierer wohl einen Fehler gemacht... Machen Sie trotzdem weiter, ein echtes System würde sich davon auch nicht aufhalten lassen!
8. schreibe `8A C3 DF 56 0F 05 FF 10 6E B7 C4 00` an die Stelle, auf die `s` zeigt
9. `t = malloc(15)` und schreibe `4B 61 74 61 73 74 72 6F 70 68 65 21 21 21 00` an die Stelle, auf die `t` zeigt

Hinweis: Sollten Ihnen die Abbildungen in der PDF-Datei zu klein sein, sind die Grafiken als Vektor-Grafiken ebenfalls noch einmal in der PDF-Datei eingebunden.