Übungsblatt 8

Dr. Matthias Frank, Dr. Matthias Wübbeling

Ausgabe Mittwoch, 29. November 2023
Abgabe bis
Freitag, 8. Dezember 2023, 23:59 Uhr
Vorführung vom 11. bis zum 15. Dezember 2022

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

Hinweis:	Manche Browser sind nicht dazu in der Lage die in die PDFs eingebetteten Dateien anzuzeigen. Mittels eines geeigneten Readers, wie dem Adobe Reader, lassen sich diese jedoch anzeigen und verwenden.

Aufgabe 1. Thread-Safe – Skalarprodukt (4 Punkte)

Hier ein (komplexeres) C-Programm, welches das Skalarprodukt von zwei Vektoren nicht thread-safe berechnet und daher häufig ein falsches Ergebnis liefert. Führen Sie das Programm mehrfach hintereinander aus und überprüfen Sie diesen Zusammenhang. Identifizieren Sie den kritischen Abschnitt und modifizieren das Programm so, dass es thread-safe ist. Die Performance darf dabei nicht massiv leiden. Unten ist ein Ausschnitt der Threadfunktion aus dem Programm **dotprod.c** Das vollständige und kommentierte Programm ist in die PDF-Datei eingebettet.

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

#define NUMTHRDS 8
#define VECLEN 100000
int *a, *b;
long sum=0;

void *dotprod(void *arg) {
        int i, start, end, offset, len;
        long tid = (long)arg;
        offset = tid;
        len = VECLEN;
        start = offset * len;
        end = start + len;
        printf("thread:%ld starting. start=%d end=%d\n", tid, start, end - 1);
        for (i = start; i < end; i++) sum += (a[i] * b[i]);
        printf("thread: %ld done. Global sum now is=%li\n",tid,sum);
        pthread_exit((void*) 0);
}
```

Aufgabe 2. Fast Bogo Sort mit pthread (5 Punkte)

Schreiben sie ein C-Programm, welches das neue und intuitive Verfahren *Divide & Bogo* mit `pthread` implentiert. Dazu sollen 2^n mit $n > 4$ zufällige Zahlen (im Bereich $[0, 10\,000]$) generiert werden. Anschließend werden die Zahlen in 4er-Päckchen aufgesplittet und pro Päckchen ein Thread gestartet, der das 4er-Array mit BogoSort[1] sortiert. Dies soll natürlich mit allen Päckchen parallel geschehen, wir nennen dies dann Fast-BogoSort. Nachdem alle BogoSort-Threads gejoint wurden, sollen solange Merger-Threads gestartet werden (welche jeweils zwei gleich lange Listen zusammenführen – analog zum Merge-Schritt in MergeSort), bis alles zu einer Liste zusammengeführt wurde. Die Ergebnisliste soll am Ende ausgegeben werden.

[1] `https://de.wikipedia.org/wiki/Bogosort`

Aufgabe 3. Klausurvorbereitung Assembler (4 Punkte)

Die folgende Aufgabe war in ähnlicher Form Teil einer Assembler- Aufgabe einer der Klausuren der SysProg im Durchlauf WS 2015/2016. Die originale Klausur-Aufgabe bezog sich noch auf 32-Bit und den GNU Assembler (GAS) und wurde hier für NASM angepasst.

a)

Bearbeiten Sie die Aufgabe der folgenden 2 Seiten und geben die Lösung als PDF (gescannt, elektronisch bearbeitet, ...) im Git Repository ab.

Hinweis zur Übung: Im ersten Anlauf können Sie zur eigenen Übung versuchen, die Aufgabe *OHNE* das aus der Vorlesung bekannte Assembler-Hilfsblatt zu lösen. In der Klausur war das Hilfsblatt natürlich dabei (und steht Ihnen auch jetzt zur Verfügung). Verwenden Sie nach Möglichkeit keine Hilfsmittel, sondern versuchen Sie, die Aufgabe wie in der Klausur zu lösen.

Klausur-Aufgabe 3: (Assembler programmieren + ….)

Gegeben ist folgender Assembler-Rahmen für eine Funktion `Max2()`, die das **zweitgrößte** Element aus einem Array von gegebenen positiven Integer-Werten (je 16 Bit) berechnen soll.

In C-Notation wäre die Signatur dieser Funktion:

```
uint16_t Max2(uint16_t *basisadresse, int anzahl);
```

Hierbei ist:

`basisadresse`	Zeiger auf die Speicherstelle, an der das Array von positiven 16-Bit-Integer-Werten beginnt
`anzahl`	Anzahl der Werte im gegebenen Array

Am Ende dieser Klausur finden Sie als Hilfestellung den bekannten Auszug aus der x86-Befehlsrefenz.

a) Ergänzen Sie den vorgegebenen Funktionsrahmen (im Formular auf der folgenden Seite) derart, dass die Funktion das zweitgrößte Element der gegebenen positiven 16-Bit-Integer-Werte ermittelt und den entsprechenden Wert im Register `eax` an den Aufrufer zurückgibt. Die beiden Parameter werden per cdecl Calling Conventions übergeben.

Sie dürfen Ihre Ergänzungen nur im vorgegebenen Lösungs-Rahmen eintragen (außerhalb sind für eine korrekte, vollständige Lösung keine Änderungen nötig). Bitte kommentieren Sie Ihre Lösung aussagekräftig.

Sie dürfen beliebige (der bekannten) Register benutzen (außer Segment-Register). Bei Bedarf können Sie lokale Variablen benutzen (ebenfalls gemäß den bekannten cdecl Calling Conventions).

Vereinfachende Annahme: Sie können annehmen, dass im Array mehr als 2 Zahlen gegeben sind, und dass alle Zahlenwerte unterschiedlich sind (d.h. kein Wert kommt mehrfach vor).

```
.intel_syntax noprefix

.text

Max2:
                                # set up stack frame
          push  ebp
          mov   ebp, esp

          mov   ebx, [ebp+8]    # hole Basisadresse in ebx
          mov   ecx, [ebp+12]   # hole Parameter „Anzahl“ in ecx


# Lösung auf Folge-Seite !!!


                    # Ergebnis soll jetzt in eax liegen

                    # reconstruct stack- and base-pointer
          leave

          ret
```

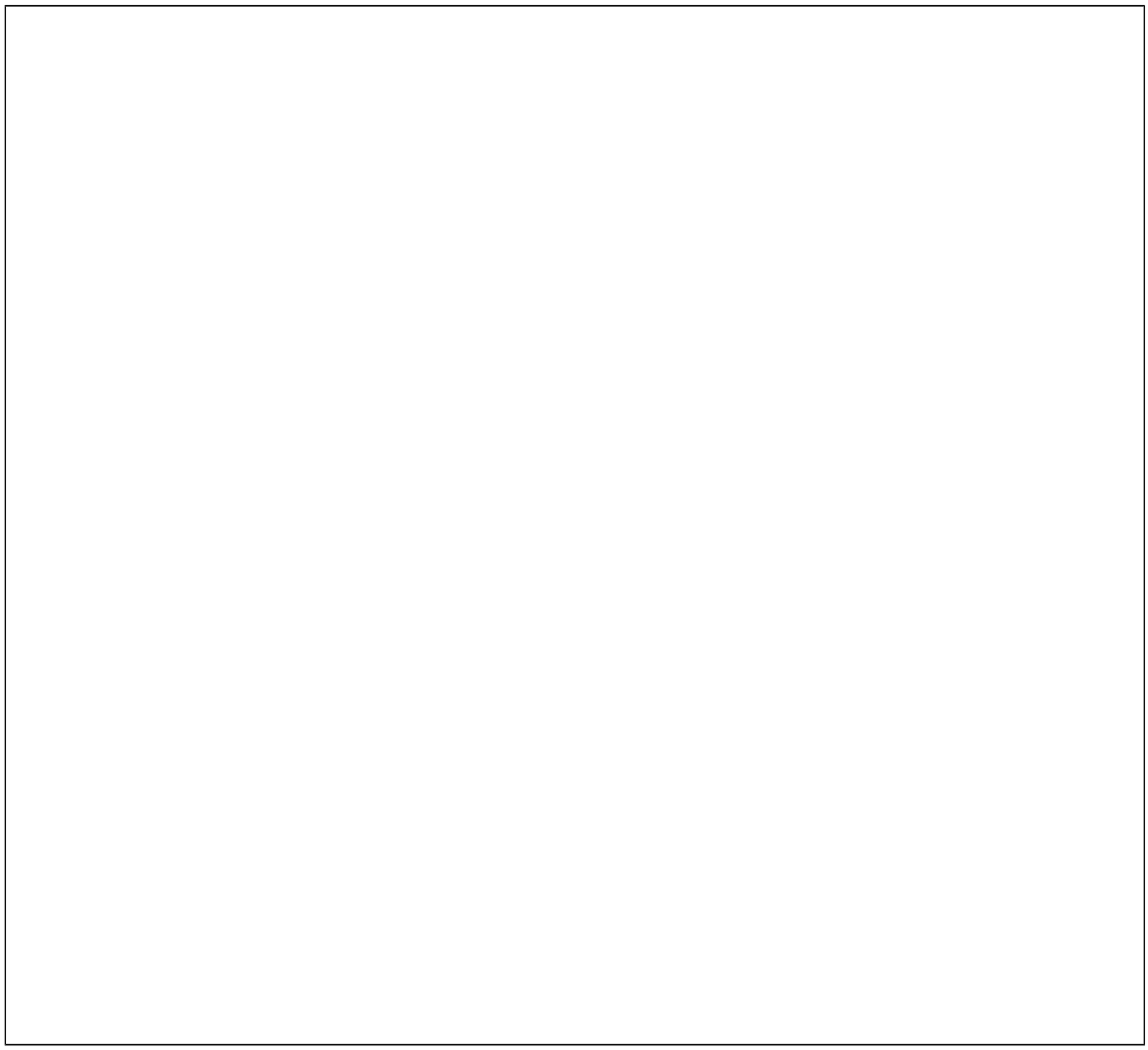
Platz für Ihre Lösung für Teil a)

```
.intel_syntax noprefix

.text

Max2:
                                # set up stack frame
          push  ebp
          mov   ebp, esp

          mov   ebx, [ebp+8]    # hole Basisadresse in ebx
          mov   ecx, [ebp+12]   # hole Parameter „Anzahl“ in ecx
```



```
                    # Ergebnis soll jetzt in eax liegen


                    # reconstruct stack- and base-pointer
          leave

          ret
```

Bonusaufgabe IV. Reverse Engineering von Shellcode (5 Punkte)

Der Administrator eines Webhosting-Unternehmens wendet sich an Sie, weil Sie sich mit x86-Assembler auskennen und ihm bei der Untersuchung eines Sicherheitsvorfalls helfen sollen. Das Network Intrusion Detection System (NIDS) des Unternehmens hat die Entdeckung von potenziellem Shellcode in einem Paket gemeldet. Ziel des potenziellen Angriffs war ein 32-Bit x86 Linux-Server, der in einem vom Internet aus erreichbaren Dienst eine Verwundbarkeit für einen Buffer-Overflow-Exploit aufwies. Der Administrator konnte aus dem geloggten Paket bereits den Teil extrahieren, der potenziellen Maschinencode enthält (diesen finden Sie auch auf der Vorlesungshomepage zum Download):

```
00000000 31 db f7 e3 53 43 53 6a 02 89 e1 b0 66 cd 80 5b
00000010 5e 52 68 ff 02 04 d2 6a 10 51 50 89 e1 6a 66 58
00000020 cd 80 89 41 04 b3 04 b0 66 cd 80 43 b0 66 cd 80
00000030 93 59 6a 3f 58 cd 80 49 79 f8 68 2f 2f 73 68 68
00000040 2f 62 69 6e 89 e3 50 53 89 e1 b0 0b cd 80
```

1. Disassemblieren Sie den Maschinencode! Sie können dazu z.B. einen Online-Disassembler[2] verwenden. Zur Kontrolle: Der korrekte Code beginnt mit `xor ebx, ebx`. Fügen Sie dann sinnvolle Kommentare hinzu! Informieren Sie sich über die verwendeten Syscalls[3].
2. Was tut der Code? Geben Sie möglichst genau an, welche Funktionen mit welchen Parametern aufgerufen werden und was dadurch passiert. Sie können statt einer reinen Beschreibung auch ein äquivalentes, gut kommentiertes C-Programm schreiben.
3. Welche spezielle Einschränkung gilt für Shellcode von Exploits, die Buffer Overflows von in C geschriebenen Programmen ausnutzen?

> Hinweis: Diese Aufgabe können Sie bis zur Abgabefrist von **Übungsblatt 9** am **15. Dezember 2023 um 23:59 Uhr** abgeben.

[2] Beispielsweise `https://www.onlinedisassembler.com/odaweb/`

[3] Beispielsweise `http://www.informatik.htw-dresden.de/~beck/ASM/syscall_list.html` oder `http://www2.ift.ulaval.ca/~marchand/ift17584/Documentation/syscall_list.pdf`