Übungsblatt 3

Dr. Matthias Frank, Dr. Matthias Wübbeling

Ausgabe Mittwoch, 25. Oktober 2023
Abgabe bis **Freitag, 3. November 2023, 23:59 Uhr**
Vorführung vom 6. bis zum 10. 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

Hinweis: In dieser Woche werden die Aufgaben von Zettel 2, die letzten Mittwoch nicht besprochen werden konnten, nachgeholt. Bereitet euch also entsprechend darauf vor.

Aufgabe 1 Systemaufrufs-Wrapper (6 Punkte)

> Hinweis: Dies ist Teil eines Aufgabenzyklus zur Entwicklung einer simplen Shell ohne Verwendung der C-Standardbibliothek.

Um das Verwenden von Systemaufrufen (s. u.) zu erleichtern (insbesondere wenn der Code plattformunabhängig sein soll), bietet die C-Standardbibliothek Wrapper-Funktionen an, die beim Aufruf die ihnen jeweils entsprechenden Systemaufrufe auslösen. In dieser Aufgabe sollen ein paar davon in Assemblersprache implementiert werden.

Durch „Systemaufrufe“ (system calls) kommunizieren Anwendungsprogramme mit dem Betriebssystem; sie werden später im Verlauf der Vorlesung näher erläutert. Ein „Dateideskriptor“ (file descriptor) ist eine Zahl, die ein Objekt im Betriebssystemkern identifiziert; viele Systemaufrufe erhalten einen Dateideskriptor als Parameter und führen Operationen auf dem zugehörigen Objekt durch. Dateideskriptoren werden nochmals später im Verlauf der Vorlesung erläutert.

Systemaufrufe sind analog zu Funktionsaufrufen strukturiert – ein Systemaufruf erhält Parameter, gibt einen Rückgabewert zurück, und hat Seiteneffekte –, verwenden allerdings meist spezifische Anweisungen (anstatt von `CALL`) und u. U. abweichende Calling Conventions. Linux auf der AMD64-Architektur verwendet die SYSCALL-Anweisung, um Systemaufrufe tatsächlich auszulösen, erwartet die Systemaufrufsnummer im `RAX`-Register, erhält Parameter (die nur Ganzzahlen oder Zeiger sein können) in den Registern `RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9` (ein Systemaufruf mit `n` Parametern verwendet die ersten `n` Register dieser Liste in der gegebenen Reihenfolge; es gibt höchstens sechs Parameter), gibt einen Rückgabewert (der auch ein Fehlercode sein kann) im `RAX`-Register zurück, und zerstört die Register `RCX` und `R11` (alle Register außer diesen und dem Rückgaberegister bleiben also unverändert).

1. Vergleichen Sie die oben beschriebene Calling Convention für Systemaufrufe mit der solchen für reguläre Funktionen aus der System V AMD64 ABI. Was muss eine Assemblerroutine, die einen gegebenen Systemaufruf mit Nummer `k` und `n` Parametern auslöst, tun, um wie eine normale C-Funktion aufgerufen werden zu können? [1]

2. Um auf Funktionsprototypen und Dokumentation aus der Standardbibliothek verweisen zu können, müssen wir (erneut) einige Typen definieren. Ergänzen Sie dazu die Datei `mystddef.h` aus STRING UND BYTEBLOCK-FUNKTIONEN, um den Typ `ssize_t` als `long`, `mode_t` als `unsigned int`, sowie `pid_t` als `int` zu definieren.

   Erstellen Sie weiterhin eine Headerdatei **myunistd.h** und deklarieren Sie dort eine Datenstruktur `struct myrusage` (Sie müssen keinen Inhalt angeben).

   > Hinweis: Sie können Dateien aus vorigen Aufgaben „ergänzen“, ohne ihren Inhalt anzutasten, indem Sie die Vorgängerversion einer Datei mittels der Präprozessoranweisung `#include` einbinden. So könnten Sie etwa **strings/mystddef.h** zu **syscalls/mystddef.h** „ergänzen“, indem Sie in **syscalls/mystddef.h** die Anweisung `#include "../strings/mystddef.h"` einführen.

   Bonus: Schlagen Sie in Ihren Systemheaderdateien den genauen Inhalt von `struct rusage` nach [2] und ergänzen Sie Ihre Deklaration von struct myrusage zu einer entsprechenden Definition.

3. Implementieren Sie in einer Quelltext-Datei **myunistd.S** Wrapper-Prozeduren für die folgenden Systemaufrufe (ihre Nummern für Linux auf der AMD64-Architektur sind jeweils in eckigen Klammern nach den Deklarationen angegeben).

[1] Hinweis: Sie können davon ausgehen, dass alle Parameter genau ein Register weit sind; insbesondere können Sie die oberen Bits von etwa 32-Bit-Parametern unverändert weiterreichen.

[2] Sie können mittels `grep -r "struct rusage" /usr/include` nach einschlägigen Dateien suchen.

> Hinweis: Sie können die Beschreibungstexte nach den Gedankenstrichen für diese Aufgabe ignorieren.

- `ssize_t myread(int fd, void *buf, size_t count)` [0] – Liest bis zu `count` Bytes aus dem Dateideskriptor `fd`, schreibt sie in `buf`, und gibt die Anzahl der tatsächlich gelesenen Bytes (oder einen Fehlercode [3]) zurück.
- `ssize_t mywrite(int fd, const void *buf, size_t count)` [1] – Schreibt bis zu `count` Bytes aus `buf` in den Dateideskriptor `fd`, und gibt die Anzahl der tatsächlich geschriebenen Bytes (oder einen Fehlercode) zurück.
- `int myopen(const char *pathname, int flags, mode_t mode)` [2] – Öffnet die Datei `pathname` und gibt einen Dateideskriptor oder einen Fehlerwert zurück. In `flags` wird der Lese- bzw. Schreibzugriff bestimmt, darüberhinaus können optionale Funktionen zur Dateierstellung oder dem I/O-Zugriff gewählt werden.
- `int myclose(int fd)` [3] – Gibt den gegebenen Dateideskriptor frei und löscht ggf. das ihm zugrundeliegende Objekt. Gibt Null oder einen Fehlercode zurück. Nach einem `myclose` ist der Dateideskriptor stets unbelegt (entweder weil er vorher gar nicht belegt war oder weil er freigegeben wurde); Fehlerindikationen haben einen lediglich diagnostischen Wert.
- `int mypipe(int fds[2])` [22] – Erstellt eine neue Pipe (ein Paar aus Dateideskriptoren, bei der die in einen Dateideskriptor geschriebenen Daten aus dem anderen wieder herausgelesen werden können). Der Dateideskriptor des Lese-Endes wird in `fds[0]`, der des Schreib-Endes in `fds[1]` eingetragen. Gibt Null oder einen Fehlercode zurück.
- `int mydup2(int oldfd, int newfd)` [33] [4] – Sorgt dafür, dass der (von der Anwendung frei gewählte) Dateideskriptor `newfd` auf dasselbe Objekt wie der bereits existierende `oldfd` verweist. Falls `newfd` bereits ein gültiger Dateideskriptor ist, wird dieser zunächst geschlossen. Gibt `newfd` oder einen Fehlercode zurück.
- `pid_t myfork(void)` [57] – Verzweigt den aktuellen Prozess in zwei Kopien, indem ein Duplikat des den Systemaufruf auslösenden „Elternprozesses“ erzeugt wird. Gibt im „Elternprozess“ die Prozess-ID des „Kinderprozesses“ bei Erfolg oder sonst einen Fehlercode zurück; im „Kinderprozess“ wird u. A. der aktuelle Ausführungszustand dupliziert – sodass derselbe Code danach zweimal gleichzeitig ausgeführt wird –, aber der Systemaufruf gibt Null zurück.
- `int myexecve(const char *filename, char *const argv[], char *const envp[])` [59] – Ersetzt das aktuell laufende Programm durch eine neu geladene Instanz der ausführbaren Datei am Dateisystempfad `fn` [5], und übergibt dem neuen Programm die Kommandozeilenparameter `argv` und die Umgebungsvariablen `envp`. Wenn dieser Systemaufruf erfolgreich ist, kehrt er nicht zurück; ansonsten gibt er einen Fehlercode zurück [6].
- `pid_t mywait4(pid_t pid, int *st, int flags, struct myrusage *ru)` [61] – Wartet darauf, bis der Kinderprozess mit der Nummer `pid` beendet wird (falls `pid` = -1, wartet dies auf einen beliebigen Kinderprozess); nachdem er es tut, speichert dies grundlegende Statusinformationen in `*st` und erweiterte solche in `*ru`. `flags` ist das

[3] Hier und bei allen anderen Systemaufrufen sind Fehlercodes Zahlen im Bereich $\{-4095, \ldots, -1\}$; sie sind Negative von verschiedenen `errno`-Konstanten, die hier nicht definiert werden.
[4] Der verwandte Systemaufruf `dup`, der einen Dateideskriptor dupliziert, ist für uns wenig interessant.
[5] Der Systemaufruf ist Teil einer Familie verwandter Standardbibliotheksfunktionen, die alle das Ausführen eines neuen Programms anstelle des aktuellen gemeinsam haben.
[6] Oder terminiert in seltenen pathologischen Fällen den Prozess.

bitweise oder von verschiedenen Konstanten, die das Verhalten des Systemaufrufs ändern und hier ausgelassen seien. Falls `st` oder `ru` der Nullzeiger [7] sind, wird nichts durch sie geschriebenen (und die entsprechenden Informationen sind dem Aufrufer nicht verfügbar). Der Rückgabewert ist die Prozess-ID des Prozesses, der beendet wurde, oder ein Fehlercode [8].

- `int mychdir(const char *dirname)` [80] – Wechselt das aktuelle Arbeitsverzeichnis zu dem durch `dirname` bezeichneten solchen. Falls `dirname` ein relativer Dateisystempfad ist, wird er relativ zum (vor dem Systemaufruf) aktuellen Arbeitsverzeichnis aufgelöst. Gibt Null oder einen Fehlercode zurück.
- `int mymkdir(const char *pathname, mode_t mode)` [83] – Erstellt ein neues Verzeichnis `pathname` im aktuellen Arbeitsverzeichnis. Das Argument `mode` bestimmt die Zugriffsrechte auf das Verzeichnis und wird meist in Oktalnotation angegeben (z.B. `0777`). Gibt Null oder einen Fehlercode zurück.
- `void myexit(int status)` [231] – Beendet den aktuellen Prozess mit dem gegebenen Statuscode. Obwohl `status` ein `int` ist, werden nur die untersten acht Bits davon tatsächlich weitergereicht. Dieser Systemaufruf kehrt nie zurück[9].

> Hinweis: Dies könnte ein guter Anwendungsfall für Assembler-Makros sein.

4. Um die Wrapper-Funktionen aus C-Code verwenden zu können, müssen sie deklariert werden. Fügen Sie dazu C-Deklarationen der obigen Wrapperfunktionen in **myunistd.h** ein.

5. Testen Sie die soeben implementierten Wrapper mit einem C-Programm, welches im obersten Verzeichnis Ihres Repos für Übungsblätter 1-10 versucht, jeweils einen Ordner **blattXX** und (darin) einen Ordner **aufgabe01** zu erstellen und in einer Datei **log.txt** protokolliert, ob die jeweilige Operation geglückt ist. Achten Sie darauf, dass die txt-Dateien die Rechte 644 und auf die Ordner die Rechte 755 haben.

> Hinweis: `open("log.txt", 01101, 0644)` erstellt eine Datei **log.txt** (bzw. löscht ihren Inhalt, wenn sie bereits existiert) mit den Rechten 644 und öffnet sie anschließend mit Schreibzugriff.

[7] Wird in einer späteren Aufgabe definiert.

[8] Der Rückgabewert kann auch Null sein; dies tritt auf, falls die (hier nicht definierte) Konstante `WNOHANG` in `flags` angegeben wird und `pid` entsprechende Kinderprozesse existieren aber noch nicht beendet wurden.

[9] Der hier beschriebene Systemaufruf ist eigentlich `exit_group`; in einer echten C-Bibliothek ist `exit` eine Bibliotheksfunktion, die verschiedene Aufräumarbeiten durchführt und abschließend `exit_group` (als `_exit`) aufruft; die hiesige Wahl des Systemaufrufs erhält das für `exit` dokumentierte Verhalten, soweit es auf unseren Fall anwendbar ist.

Aufgabe 2 Der gdb Befehlssatz (3 Punkte)

Häufig funktionieren Programme nicht auf Anhieb, sondern liefern falsche oder gar keine Ausgaben. Zur Identifikation von Fehlern eignen sich Debugger, welche es ermöglichen Programmcode schrittweise auszuführen, Variablen oder Register einzusehen, oder Code während des Debuggins zu verändern um einen Patch ohne erneutes Kompilieren zu testen. Ein beliebter Debugger für C- und Assembler-Code ist der *GNU Debugger* (`gdb`), mit dem Sie sich in diesem Übungsblatt vertraut machen können.

> Hinweis: `(gdb) COMMAND` bedeutet, dass `COMMAND` innerhalb von `gdb` ausgeführt wird.

`gdb` bietet eine Vielzahl an Befehlen, die das Debugging vereinfachen können. Tabelle 1 enthält eine Übersicht über häufig verwendete Befehle; Tabelle 2 listet einige Beispiele für Datentypen, die von `(gdb) print` und `(gdb) x` unterstützt werden um Variablen bzw. Speicherbereiche auszugeben.

Verwenden Sie das gdb User Manual (`https://sourceware.org/gdb/current/onlinedocs/gdb`) oder die Hilfefunktion `(gdb) help` um beide Tabellen zu vervollständigen. Die PDF enthält hierzu die Tabellen als csv-Dateien.

Aktion	**Befehl**	**Kurzform**
		r
Ausführung unterbrechen wenn Zeile n erreicht wird		
Ausführung unterbrechen wenn Funktion fun() aufgerufen wird		
	watch X	
Alle Breakpoints anzeigen		
Alle Register anzeigen		
Breakpunkt Nummer N entfernen		
Breakpoint an Zeile oder Funktion X entfernen		
	next	
	step	
		ni
		si
Programm nach Breakpoint fortsetzen		
		fin
Die Ausführung der letzten Zeile rückgängig machen.		
	reverse-cont	
	print X	
	x X	–
Variable X Wert Y zuweisen		–
Datentyp von X ausgeben		
	backtrace	

Tabelle 1: Häufig verwendete gdb Befehle

Erwünschte Ausgabe	**Addresse**	**Befehl**
A 4-Byte signed decimal	r10	
A 2-Byte unsigned integer in decimal	r10	
A 8-Byte hexadecimal	r10	
A string	123456 (hex)	
An array of 5 characters	123456 (oct)	
An array of 5 characters printed as ASCII (numerical) values	123456 (dec)	
An array of 10 pointers	r10	
An array of 3 2-Byte octals	r10	
An array of 4 single-precision floating point numbers	r10	
An array of 8 half-precision floating point numbers	r10	

Tabelle 2: Formatierung für (gdb) print und (gdb) examine

Aufgabe 3 C Debugging mit gdb (3 Punkte)

Verwenden Sie nun die kürzlich erworbenen oder aufgefrischten gdb-Kenntnisse um ein einfaches Programm zu reparieren. Folgender Code soll ein Array an Daten anhand eines Schlüssel via Bubble Sort sortieren.

```c
#include <stdio.h>

typedef struct {
  char data[4096];
  int key;
} item;

item array[] = {
    {"bill", 3}, {"neil", 4}, {"john", 2}, {"rick", 5}, {"alex", 1},
};

void sort(item *a, int n){
  int i = 0, j = 0;
  int s = 1;

  for (; i < n && s != 0; i++) {
    s = 0;
    for (j = 0; j < n; j++) {
      if (a[j].key > a[j + 1].key) {
        item t = a[j];
        a[j] = a[j + 1];
        a[j + 1] = t;
        s++;
      }
    }
    n--;
  }
}

int main() {
  sort(array, 5);
  for(int i = 0; i < 5; i++) printf("array[%d] = {%s, %d}\n", i, array[i].data, array[i].key)
  return 0;
}
```

Das angegebene Programm ist als `bugblesort.c` in die PDF-Datei eingebettet. Kompilieren Sie das Program wie gewohnt mit

```
clang -g -o bugblesort bugblesort.c
```

Bei der Ausführung wird mit hoher Wahrscheinlichkeit auf einen ungültigen Speicherbereich zugegriffen, was in einem *Segmentation Fault* resultiert. Widerstehen Sie dem Impuls, den Fehler visuell zu identifizieren und verwenden Sie stattdessen gdb. Speichern Sie entweder den korrigierten Quellcode in einer neuen Datei `bubblesort.c` oder listen Sie in `patch.txt` (ausschließlich) gdb-Befehle, welche das Programm reparieren, so dass es anschließend korrekt ausgeführt werden kann.

Aufgabe 4 Compiler-Optimierungen (4 Bonuspunkte)

Disassembliert man Programme, deren Quellcode in einer Hochsprache wie C geschrieben und dann mit einem Compiler wie `gcc` compiliert wurde, so fällt auf, dass der Assembler- Code oft ganz anders aussieht, als wenn man ihn „straight-forward“selber geschrieben hätte. Ein wichtiger Grund dafür ist, dass sich gewisse Aufgabenstellungen mit zunächst unintuitiv erscheinenden Befehlen oder Befehlsfolgen oft schneller (das heißt in weniger CPU-Taktzyklen) oder mit weniger Maschinencode lösen lassen als mit einem intuitiven Ansatz. Unten sehen Sie einige Beispiele dafür. Geben Sie an, welche Befehle oder Befehlsfolgen der Programmierer wohl ursprünglich in seinem C-Quelltext verwendet hat!

a)

```
xor rax, rax
```

b)

```
lea rax, [rax+rax*8]
sal rax, 2
```

c)

```
xor rcx, rcx
cmp rax, 42
setnz cl
dec rcx
and rcx,0xfffffffffffffffb
add rcx, 0x0a
```

d)

```
mov rdx, 0xcccccccccccccccd
mul rdx
shr rdx, 3
```