Übungsblatt 6

Dr. Matthias Frank, Dr. Matthias Wübbeling

Ausgabe Mittwoch, 15. November 2023
Abgabe bis **Freitag, 24. November 2023, 23:59 Uhr**
Vorführung vom 27. November bis zum 1. Dezember 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:	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 Speicherverwaltung (6 Punkte + 2 Bonuspunkte)

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

Eine wichtige Funktion der C-Standardbibliothek ist die Bereitstellung einer dynamischen Speicherverwaltung; ebendiese soll in dieser Aufgabe implementiert werden. Dafür sollen Sie einen 1 MiB (2^{20} Bytes) großen fixen und statisch allokierten Speicherblock verwenden, den Sie z. B. durch folgende (globale) Deklaration anlegen können:

```
char memory[1048576];
```

Implementieren Sie die in der Vorlesung vorgestellte Free-Liste, um diesen Speicher zu verwalten:

1. Um auf Funktionsprototypen und Dokumentation aus der Standardbibliothek verweisen zu können, müssen wir den Null-Pointer definieren. Ergänzen Sie dazu die Datei **mystddef.h** aus STRING- UND BYTEBLOCK-FUNKTIONEN (oder aus SYSTEMAUFRUFS-WRAPPER) um eine Definition des Präprozessormakros `NULL` als `((void*)0)`.

2. Deklarieren Sie in einer Headerdatei **mystdlib.h** und implementieren Sie in einer Quelltextdatei **mystdlib.c** die folgenden Funktionen:

   - `void *mymalloc(size_t size)` – Reserviert `size` Bytes Speicher und gibt einen Zeiger an den Anfang des reservierten Blocks zurück. Falls `size = 0`, soll `NULL` zurückgeben werden. Falls nicht genügend Speicher verfügbar ist, gibt dies `NULL` zurück.

     Falls der Aufruf erfolgreich war (d. h. nicht `NULL` zurückgab), müssen sämtliche Bytes des Speicherblocks erfolgreich lesbar und schreibbar sein; der initiale Inhalt des Speicherblocks ist nicht spezifiziert, und der neue Speicherblock darf keinen bereits reservierten, aber noch nicht freigegebenen Speicherblock überlappen.
   - `void myfree(void *ptr)` – Gibt den Speicherblock beginnend bei `ptr` frei, sodass er wiederverwendet werden kann. Falls `ptr` der Nullzeiger `NULL` ist, passiert nichts. Falls `ptr` nicht `NULL` ist, muss `ptr` vorher von `mymalloc` zurückgegeben und seitdem nicht freigegeben worden sein, ansonsten ist das Verhalten undefiniert (d. h. Sie müssen diesen Fall nicht gesondert behandeln).
   - `void *mycalloc(size_t nmemb, size_t size)` – Reserviert `nmemb` · `size` Bytes an Speicher, initialisiert sie mit Nullen, und gibt einen Zeiger auf den Anfang des Speicherblocks zurück. Siehe `mymalloc` für weitergehende Anforderungen an den zurückgegebenen Speicherblock.

   > Hinweis: **mystdlib.c** ist auch der richtige Ort, um den zu verwaltenden statischen Speicherblock anzulegen (s. o.).

Auf der Vorlesungs-Website sowie eingebunden in der PDF-Datei finden Sie ein Programm **test.c**, welches Sie benutzen können, um ihre Implementierung zu testen.

> Hinweis: Ihre Lösung muss nicht die effizienteste sein. Achten Sie mehr auf lesbaren Code und auf die korrekte Behandlung aller Sonderfälle, die bei der Speicherverwaltung auftreten können.

Bonus (2 Bonuspunkte)

Zusätzlich zu den obigen Funktionen können Sie folgende Funktion implementieren, die besonders für dynamische Speicherverwaltung bei Programmen hilfreich ist.

`void *myrealloc(void *ptr, size_t size)` – Verändert die Größe des bei `ptr` startenden Speicherblocks zu `size` und verschiebt ihn gegebenenfalls an eine neue Adresse, falls nicht genug Speicher nach hinten verfügbar ist. Gibt die neue Addresse des Speicherblocks zurück (falls sie unterschiedlich von `ptr` ist, zählt `ptr` als freigegeben und darf nicht an `myfree` etc. übergeben werden).

Nach dem Aufruf ist das Minimum von der alten Größe des Speicherblocks und `size` Bytes am neuen Zeiger startend gültig; der Inhalt von evtl. neu entstandenen Bytes ist nicht spezifiziert. Falls das Reservieren von neuem Speicher nicht gelingt, gibt dies `NULL` zurück und lässt den Speicher an `ptr` unberührt. Siehe `mymalloc` für weitergehende Anforderungen an den zurückgegebenen Speicherblock. Falls `ptr` der Nullzeiger `NULL` ist, verhält sich dies wie `mymalloc(size)`. Falls `size = 0` und `ptr != NULL`, verhält sich dies wie `myfree(ptr)` und gibt `NULL` zurück.

Aufgabe 2 Aufruf von glibc-Funktionen in Assembler (4 Punkte)

In der vorangegangenen Aufgabe haben Sie eine Assembler-Funktion aus einem C-Programm heraus aufgerufen. In dieser Aufgabe sollen Sie — praktisch umgekehrt — Funktionen der C-Standardbibliothek **glibc** aus einem Assembler-Programm heraus aufrufen.

Schreiben Sie ein Assembler-Programm, das folgendes leistet:

- Ein maximal 20 Zeichen langer String wird von der Standardeingabe `stdin` mithilfe der glibc-Funktion `fgets` eingelesen.
- Der String wird komplett in Großbuchstaben umgewandelt (das heißt kleine Buchstaben werden in große umgewandelt, alle anderen Zeichen bleiben unverändert).
- Falls der String einen Zeilenumbruch enthält, soll dieser nicht mit ausgegeben werden.
- Der umgewandelte String wird mithilfe der glibc-Funktion puts auf der Standardausgabe `stdout` ausgegeben.

Kompilieren Sie ihr Programm in 64-Bit und linken Sie es mit der glibc. Wenn Sie mit clang linken, brauchen Sie das Flag `-no-pie`. Sie finden die glibc unter Ubuntu 22.04 im Verzeichnis **/usr/lib/x86_64-linux-gnu/**.

> Hinweis: Die Folien zum Stack Alignment können bei dieser Aufgabe hilfreich sein. Ihr findet diese in Kapitel 1 ab Folie 62.

Aufgabe 3 Fork und Exec (4 Punkte)

In dieser Aufgabe sollen Sie einen Prozess-Launcher implementieren. Dies soll ein relativ einfaches Programm sein, welchem Sie per Kommandozeilenparameter eine Zahl `n` und den Namen eines Programmes übergeben. Der Launcher soll dann `n` Instanzen des angegebenen Programmes starten. Soll ein Programm mit Parametern ausgeführt werden, sollen diese ebenfalls an den Launcher übergegen werden und von dort aus an das auszuführende Programm weiter gereicht werden. Um einen Prozess des Ziel-Programmes zu starten, sollen `fork` und `execvp` verwendet werden. So soll Ihr Programm bei einem Beispielaufruf wie

```
./prozess-launcher 10 firefox --browser
```

zehn neue Prozesse forken und in jedem neuen Prozess das Programm `firefox` mit dem Argument `--browser` ausführen.

> Hinweis: Da zum Ausführen der Programme execvp verwendet werden soll, können Sie für alle Programme, die im System-`PATH` liegen, einfach den Namen des Programmes an den Prozess-Launcher übergegeben. Für Programme, die nicht im `PATH` liegen, muss entsprechend ein absoluter Pfad zum Ziel-Programm an den Prozess-Launcher übergegeben werden. Ihr Launcher soll sich also nicht selbst darum kümmern, für einen gegebenen Programmnamen den absoluten Pfad zur entsprechenden ausführbaren Datei zu finden und diesen an `execvp` zu übergeben.