Aufgabe 4: halde (18.0 Punkte)

In dieser Aufgabe soll eine einfache Freispeicherverwaltung implementiert werden, welche die Funktionen **malloc(3)**, **calloc(3)**, **realloc(3)** und **free(3)** aus der Standard-C-Bibliothek ersetzt. Die Verwaltung des Speichers der Größe 1 MiB erfolgt mit Hilfe einer einfach verketteten Liste. Die einzelnen Listenelemente, die die Größe des verwalteten Speicherbereichs beinhalten, werden jeweils am Anfang des dazugehörigen Speicherbereiches abgelegt.

a) Makefile

Erstellen Sie ein Makefile, welches die Targets `test` und `test-ref` unterstützt. Das Target `test` erzeugt aus dem Testfall (`test.c`) und Ihrer Implementierung der Freispeicherverwaltung (`halde.c`) die ausführbare Datei `test`. Das Target `test-ref` erzeugt aus dem Testfall und der von uns bereitgestellten Freispeicherverwaltung (`halde-ref.o`) die ausführbare Datei `test-ref`. Greifen Sie dabei stets auf Zwischenprodukte (z. B. `halde.o`) zurück. Das Makefile soll ohne eingebaute Regeln funktionieren (**make(1)** mit den Optionen `-rR` starten). Nutzen Sie die im Moodle genannten Compilerflags und nutzen Sie die konventionskonforme Variablennamen (bspw. CFLAGS).

b) Testfall für `malloc()` und `free()`

Implementieren Sie einen Testfall für die Freispeicherverwaltung in der Datei `test.c`. Dieser soll **mindestens** aus vier aufeinanderfolgenden `malloc()`-Aufrufen, der Freigabe der angeforderten Speicherbereiche und weiteren vier `malloc()`-Aufrufen bestehen. Außerdem soll **mindestens einer** der in der Manpage (**malloc(3)**) beschriebenen Randfälle geprüft werden – bitte geben Sie in einem Textkommentar über dem Aufruf an, welchen Randfall Sie testen. Am Ende des Testfalles sollen alle angeforderten Speicherbereiche wieder mit `free()` freigegeben werden.
Nach jedem `malloc()`- und `free()`-Aufruf soll die Funktion vorgegebene `printList()` aufgerufen werden, die den internen Zustand der Freispeicherliste ausgibt. Sie dürfen jedoch nicht mit **printf(3)**/**fprintf(3)** auf `stdout` schreiben! Vergleichen Sie bereits während der Entwicklung Ihrer Freispeicherverwaltung die Ausgabe der Programme `test` und `test-ref`. Die Ausgabe muss nicht exakt übereinstimmen – es ist ausreichend, wenn die Anzahl der angezeigten Listenelemente genau und die Gesamtmenge des freien Speichers ungefähr übereinstimmt. Der Aufruf `make test test-ref` übersetzt beide Varianten Ihrer Freispeicherverwaltung.
Achtung: Ein funktionierender Testfall ist kein Garant dafür, dass die Freispeicherverwaltung vollständig korrekt funktioniert.

c) Funktionen `malloc()` und `free()`

Die Funktion `malloc()` sucht in der Freispeicherliste den ersten Speicherbereich, der für die angeforderte Speichermenge groß genug ist, und entfernt ihn aus der Freispeicherliste. Ist der Speicherbereich größer als benötigt und verbleibt **genügend** Rest, so wird dieser Speicherbereich geteilt und der Rest wird mit Hilfe eines neuen Listenelementes in die Freispeicherliste eingehängt. Im herausgenommenen Listenelement wird statt eines *next*-Zeigers eine *Magic Number* mit dem Wert `0xbaadf00d` eingetragen. Der von `malloc()` zurückgelieferte Zeiger zeigt auf die Nutzdaten hinter dem Listenelement. Wird `malloc()` mit der Größe `0` aufgerufen, dann liefert `malloc()` einen `NULL`-Pointer zurück.

Die Funktion `free()` hängt den freizugebenden Speicherbereich wieder vorne in die Freispeicherliste ein, **ohne** ihn mit gegebenenfalls vorhandenen benachbarten freien Bereichen zu verschmelzen. Vor dem Einhängen muss die *Magic Number* überprüft werden. Schlägt die Überprüfung fehl, so soll das Programm durch den Aufruf der Funktion **abort(3)** abgebrochen werden. Wird `free()` mit einem `NULL`-Pointer aufgerufen, dann kehrt `free()` ohne Fehler zurück.

d) Funktionen `realloc()` und `calloc()`

Die Funktion `realloc()` ist auf `malloc()` + `memcpy()` + `free()` abzubilden. Ein `realloc()` auf Größe 0 soll sich dabei wie ein Aufruf von `free()` verhalten. Ein `realloc()` auf den Zeiger `NULL` soll sich dabei wie ein Aufruf an `malloc()` verhalten. Falls die Allokation des neuen Speicherbereichs fehlschlägt, so bleibt der alte Speicherbereich unverändert und `realloc()` kehrt mit `NULL` und gesetzter `errno` zurück.

Die Funktion `calloc()` verwendet `malloc()` zur Anforderung eines Speicherbereichs in der passenden Größe und initialisiert ihn byteweise mit `0x0`.

Hinweise zur Aufgabe:

- Achten Sie darauf, dass die **errno(3)** korrekt auf `ENOMEM` gesetzt wird, falls kein ausreichend großer freier Speicherbereich verfügbar ist.
- Die Anforderung eines Speicherbereiches der Größe (*1 MiB - Größe eines Listenelementes*) ist erfolgreich.
- Hilfreiche *Manual-Pages*: **abort(3)**, **calloc(3)**, **free(3)**, **malloc(3)**, **memcpy(3)**, **memset(3)**, **realloc(3)**
- Die vorgegebenen Signaturen der Funktionen `malloc()`, `calloc()`, `realloc()` und `free()` dürfen nicht verändert werden.

- Im zip-Archiv befinden sich die Dateien `halde.{c,h}`, `halde-ref.o` und `test.c`. Kopieren Sie sich diese Dateien in Ihr Projektverzeichnis und implementieren Sie die fehlenden Funktionen und Definitionen in der Datei `halde.c` und `test.c`.
- Die Funktion `printList()` gibt für jedes Listenelement die Position im Adressraum (`addr`), den Offset innerhalb der 1 MiB (`offset`) und die eingetragene Größe (`size`) auf den Standardfehlerkanal aus.
- Ihr Programm muss mit den folgenden Compiler Flags übersetzen:
  `-std=c11 -pedantic -Wall -Werror -D_XOPEN_SOURCE=700`
  Diese Flags werden zur Bewertung herangezogen.

Hinweise zur Abgabe:

Bearbeitung:	Zweiergruppen
Abzugebende Dateien:	halde.c (14 Punkte), test.c (2 Punkte), Makefile (2 Punkte)
Abgabezeitpunkte nach Tafelübungsgruppe:	T01-T02: 18.06.2023, 17:30 Uhr T03-T06: 21.06.2023, 17:30 Uhr T07-T09: 22.06.2023, 17:30 Uhr