Aufgabe 5: halde (14.0 Punkte)

In dieser Aufgabe soll eine einfache feingranulare Freispeicherverwaltung implementiert werden, welche die Funktionen **malloc(3p)**, **calloc(3p)**, **realloc(3p)** und **free(3p)** aus der Standard-C-Bibliothek ersetzt. Die Verwaltung des Speichers der Größe 1 MiB, welcher bei der Initialisierung einmalig (i.e., grobgranular) vom Betriebssystem mittels **mmap(2)** Systemaufruf angefordert wird, 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 zur Aufgabe passendes Makefile, welches die Targets `all`, `clean`, `test` und `test-ref` unterstützt sowie die Makros `CC` und `CFLAGS` nutzt. 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 (deshalb **make(1)** mit den Optionen `-rR` starten).
Nutzen Sie in dieser Aufgabe *ausnahmsweise* die Compilerflags
`-std=c11 -pedantic -Wall -Werror -fsanitize=undefined -fno-sanitize-recover -static-libubsan -g -D_GNU_SOURCE`
anstatt der sonst üblichen Flags von der Webseite. Dies ist für die Nutzung von `MAP_ANONYMOUS` unbedingt notwendig.

b) Testen der Freispeicherverwaltung

Implementieren Sie einen Testfall für die Freispeicherverwaltung in der Datei `test.c`. Dieser soll zuerst die Grundfunktionalität mithilfe von **mindestens vier** aufeinanderfolgenden `malloc()`-Aufrufen, der Freigabe der angeforderten Speicherbereiche und weiteren vier `malloc()`-Aufrufen testen. Zusätzlich sollen **mindestens sechs** Randfälle geprüft werden. Denken Sie darüber nach, welche Tests sinnvoll sind (Hinweis: ein Blick in die Manpages der zu implementierenden Funktionen kann helfen) und geben Sie in einem Textkommentar über jedem Aufruf an, welcher Randfall getestet wird. Am Ende des Testfalles sollen alle angeforderten Speicherbereiche wieder mit `free()` freigegeben werden.
Nach jedem `malloc()`- und `free()`-Aufruf soll die Funktion `printList()` aufgerufen werden, die den internen Zustand der Freispeicherliste ausgibt. Sie dürfen jedoch nicht mit **printf(3p)**/**fprintf(3p)** 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. Achten Sie darauf, dass die zurückgegebenen Speicherbereiche Vielfaches von `alignof(max_align_t)` groß sind. Dafür wird im vorgegebenen Code die Hilfsfunktion `round_to_alignment(size_t)` bereitgestellt.

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(3p)** abgebrochen werden.

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

e) Unvollständige Checkliste

Die folgenden Punkte der **unvollständigen** Checkliste sollten Sie vor der finalen Abgabe zwingend abarbeiten:

- ☐ Die Funktionen **malloc(3p)**, **calloc(3p)**, **realloc(3p)** und **free(3p)** weisen das in den Manpages beschriebene Verhalten auf, auch in den genannten Grenzfällen (z. B. `free(NULL)`).
- ☐ Die **errno(3p)** wird im Fehlerfall korrekt gesetzt.
- ☐ Die Allokation eines Speicherbereiches der Größe (*1 MiB - Größe eines Listenelementes*) ist erfolgreich.

Hinweise zur Aufgabe:

- Erforderliche Dateien: `halde.c` (10 Punkte), `Makefile` (2 Punkte), `test.c` (2 Punkte)
- Hilfreiche *Manual-Pages*: **abort(3p)**, **calloc(3p)**, **free(3p)**, **malloc(3p)**, **memcpy(3p)**, **memset(3p)**, **realloc(3p)**, **mmap(2)**
- In Ihrem Git-Verzeichnis befinden sich die Dateien `halde.{c,h}`, `halde-ref.o` und `test.c`. 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.

Hinweise zur Abgabe:

Bearbeitung: Dreiergruppen
Bearbeitungszeit: 10 Werktage (ohne Wochenenden und Feiertage)
Abgabezeit: 17:30 Uhr