Abgabefrist Übungsaufgabe 4

Die Lösung muss abhängig von der Tafelübung abgegeben werden, an der ihr teilnehmt:

- **W1** – eigene Übung U4 am 15./16.06.: Abgabe bis **Donnerstag, den 24.06.2021 08:00**
- **W2** – eigene Übung U4 am 22./23.06.: Abgabe bis **Dienstag, den 29.06.2021 08:00**

In darauffolgenden Tafelübungen werden teilweise einzelne abgegebene Lösungen besprochen, teilweise auch ein Lösungsvorschlag aus dem Tutorenteam.

Allgemeine Hinweise zu den BS-Übungen

- Es ist *nicht* mehr möglich, Einzelabgaben im AsSESS zu tätigen. Falls ihr (statt in einer Dreiergruppe) zu zweit oder zu viert abgeben möchtet, klärt dies bitte *vorher* mit eurem Übungsleiter! Der Lösungsweg und die Programmierung sind gemeinsam zu erarbeiten.
- Die abgegebenen Antworten/Programme werden automatisch auf Ähnlichkeit mit anderen Abgaben überprüft. Wer beim Abschreiben[1] erwischt wird, verliert ohne weitere Vorwarnung die Möglichkeit zum Erwerb der Studienleistung in diesem Semester!
- Die Zusatzaufgaben sind ein Stück schwerer als die „normalen“ Aufgaben und geben zusätzliche Punkte.
- Die Aufgaben sind über AsSESS (`https://ess.cs.tu-dortmund.de/ASSESS/`) abzugeben. Dort gibt **ein** Gruppenmitglied die erforderlichen Dateien ab und nennt dabei die anderen beteiligten Gruppenmitglieder (Matrikelnummer, Vor- und Nachname erforderlich!). Namen und Anzahl der abzugebenden C-Quellcodedateien[2] variieren und stehen in der jeweiligen Aufgabenstellung; Theoriefragen sind grundsätzlich in der Datei `antworten.txt`[3] zu beantworten. Bis zum Abgabetermin kann eine Aufgabe beliebig oft abgegeben werden – es gilt die letzte, vor dem Abgabetermin vorgenommene Abgabe.
- Sobald eine Abgabe korrigiert wurde, kann das Ergebnis ebenfalls im AsSESS eingesehen werden.

[1] Da wir im Regelfall nicht unterscheiden können, wer von wem abgeschrieben hat, gilt das für Original **und** Plagiat.
[2] codiert in UTF-8
[3] reine Textdatei, codiert in UTF-8

Betriebssysteme Übungsaufgabe 4 Speicherverwaltung

Aufgabe 4: Speicherverwaltung (10 Punkte)

Ziel dieser Aufgabe ist die Implementierung einer kleinen, einfachen Speicherverwaltung (angelehnt an das in C zur Verfügung stehende **malloc(3)** und **free(3)**).

ACHTUNG: Zu dieser Aufgabe existiert eine Vorgabe in Form von C-Dateien mit einem vorimplementierten Code-Rumpf, die ihr in der Programmieraufgabe erweitern sollt. Diese Vorgaben sind von der Veranstaltungswebseite herunterzuladen, zu entpacken und zu vervollständigen! Die Datei `vorgaben-A4.tar.gz` lässt sich unter Linux/UNIX mittels `tar -xzvf vorgaben-A4.tar.gz` auspacken.

Theoriefragen (3 Punkte)

⇒ antworten.txt

1. Buddy-Verfahren

Ein Computersystem verfügt über **16 MBytes Hauptspeicher**, die **kleinst mögliche Blockgröße** beträgt **1 MByte** und es kommt das Buddy-Verfahren zum Einsatz.

Zu Beginn ist der Hauptspeicher leer. Anschließend treffen die folgenden Speicheranforderungen bzw. -freigaben ein:

(1) Anforderung A von 2 MBytes
(2) Anforderung B von 6,5 MBytes
(3) Anforderung C von 1,3 MBytes
(4) Anforderung D von 42 KBytes
(5) Anforderung E von 1,8 MBytes
(6) Freigabe von Anforderung C
(7) Freigabe von Anforderung D
(8) Anforderung F von 3,5 MBytes
(9) Anforderung G von 512 KBytes

Wenn für eine Anforderung nicht genügend Speicher zur Verfügung steht, wird diese abgelehnt und es werden keine Änderungen an der Speicherbelegung vorgenommen.

Stellt die Speicherbelegung nach jedem Anforderungs- bzw. Freigabeschritt dar. Das folgende Format mit einer Zeile je Anforderung bzw. Freigabe bietet sich dafür an.

Beispiel: 8 MBytes Hauptspeicher und minimale Blockgröße 1 MByte

```
--------        Initialer Zustand ohne Belegung(en)
AAAA|----       Anforderung A von 4 MBytes
AAAA|B|-|--     Anforderung B von 1 MByte
AAAA|B|-|--     Anforderung C von 4 MBytes: abgelehnt
----|B|-|--     Freigabe von Anforderung A
```

2. Verschnitt

Beantwortet die folgenden Fragen in *eigenen* Worten.

a) Was versteht man unter Verschnitt? Welche Arten von Verschnitt können auftreten und wodurch unterscheiden sich diese?

b) In der Programmieraufgabe sollt ihr euch mit der Platzierungsstrategie Best Fit beschäftigen. Kommt es hier zu Verschnitt? Warum oder warum nicht?

© 2021 TU Dortmund, Informatik 12, Arbeitsgruppe Systemsoftware

Programmierung: Dynamische Speicherverwaltung mit der Best Fit Strategie (7 Punkte)

⇒ bestfit.c und bestfit.h

In der Vorgabe befinden sich drei Dateien:

malloctest.c Ein vorgegebenes Testprogramm, das eure selbstgebaute dynamische Speicherverwaltung testet.

bestfit.c, bestfit.h Hier sollt ihr die Funktionen **bf_alloc** und **bf_free** mit Hilfe einer ganzen Reihe von vorgegebenen Hilfsfunktionen implementieren.

Makefile Eine Steuerdatei für Make, mit der ihr das Testprogramm **malloctest** erstellen könnt. Das Programm kann mit einem optionalen Parameter aufgerufen werden, der die Anzahl der *sequentiellen* Speicherbelegungen steuert (z. B. `malloctest 42`).

Die Vorgabe lässt sich mit dem Kommando „make“ übersetzen.

Abzugeben sind nur die Dateien **bestfit.c** und **bestfit.h**. Dementsprechend dürft ihr nicht die Schnittstelle der beiden Funktionen (**bf_alloc**, **bf_free**) ändern, da sonst unser Testprogramm nicht mehr damit funktioniert. Abgaben mit geänderter Schnittstelle werden mit 0 Punkten bewertet.

Die zu entwickelnde Speicherverwaltung ist an die C-Funktionen **malloc(3)** und **free(3)** angelehnt. Der Speicher soll nach der **Best Fit** Strategie aus der Vorlesung belegt werden.

Die Speicherverwaltung verwaltet, der Einfachheit halber, den Speicher in einem 12 KiB großen (die Größe ist mittels `#define MEM_POOL_SIZE` festgelegt), globalen char-Array `char mem_pool[MEM_POOL_SIZE]`. Speicher aus diesem Array wird nur in Stücken (*„Chunks“*) der Größe 32 Bytes (als `CHUNK_SIZE` definiert) belegt. Nicht passende Speicheranforderungen sollen auf das nächste Vielfache der *Chunk*-Größe (z. B. von 37 auf 64 Bytes) aufgerundet werden. Die vorgegebene Hilfsfunktion **size_to_chunks** berechnet aus einer gegebenen, noch nicht aufgerundeten Größe die Anzahl der zu belegenden *Chunks*.

Die Information, ob ein Speicherstück der Größe `CHUNK_SIZE` belegt ist, wird in der Bitliste `char free_list[MEM_POOL_SIZE / CHUNK_SIZE / 8]` abgelegt: Eine 0 entspricht einem freien Speicherstück (*Chunk*), wohingegen bei einer 1 die entsprechende Stelle belegt ist (siehe Abbildung 1). Diese Liste wird im Folgenden als *Freispeicher-Bitliste* bezeichnet. In der Liste gibt es `FREELIST_BIT_COUNT` viele Bits.

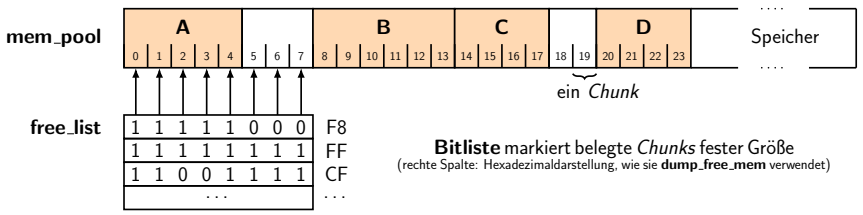

Abbildung 1: Freispeicher-Bitliste mit zugehörigen *Chunks* fester Größe (z.B. 32 Bytes) im Speicher.

Im Modul **bestfit** (**bestfit.c**, **bestfit.h**) soll die einfache Freispeicherverwaltung implementiert werden, die vom Testprogramm über die Funktionen `void *bf_alloc(size_t size)` und `void bf_free(void *ptr, size_t size)` verwendet wird. **bf_alloc** soll dabei einen Speicherbereich der Größe **size** Bytes belegen und dessen Startadresse zurückliefern; **bf_free** soll diesen (unter Angabe seiner Startadresse und seiner Länge) wieder freigeben.

a) bf_alloc (5 Punkte)

In dieser Teilaufgabe sollt ihr **bf_alloc** implementieren. Geht dabei schrittweise vor:

- Rechnet zunächst aus, wie viele *Chunks* ihr belegen müsst; das ist mit der gegebenen Hilfsfunktion **size_to_chunks** möglich.
- Nun müsst ihr, unter Verwendung der vorgegebenen Hilfsfunktionen **bit_is_set**, eine freie Stelle (frei = genügend 0-Bits hintereinander) in der Freispeicher-Bitliste finden. Die Lücke muss groß genug sein, um die Speicheranforderung aufzunehmen. Die „Lückensuche“ soll nach der Platzierungsstrategie **Best Fit** erfolgen.
- Wenn keine ausreichend große Lücke vorhanden ist oder wenn `size` gleich 0 ist, gebt `NULL` an den Aufrufer zurück und signalisiert ihm damit, dass die Anforderung fehlgeschlagen ist.
- Ansonsten gebt dem Aufrufer die Speicheradresse innerhalb des `mem_pool` zurück, die der gefundenen Stelle in der Freispeicher-Bitliste entspricht. Beachtet, dass die *absolute Speicheradresse* zurückgegeben werden muss, nicht die Adresse relativ zum Beginn des `mem_pool`. In der Freispeicher-Bitliste muss natürlich, mit **set_bit** (Funktion ist vorgegeben), gespeichert werden, dass der Speicherplatz nun belegt ist.
- Hinweis: Beachtet auch den Spezialfall, bei dem die passende Lücke bis ganz ans Ende des Speichers reicht.

b) bf_free (2 Punkte)

Um zuvor reservierten Speicher freigeben zu können, sollt ihr in dieser Teilaufgabe **bf_free** implementieren. Das heißt, ihr müsst die entsprechenden Bits in der Freispeicher-Bitliste wieder löschen.

- Die *Position* dieser Bits könnt ihr leicht anhand der Differenz des übergebenen Zeigers `ptr` und der Startadresse des Speicherpools `mem_pool` ausrechnen. Denkt daran, dass ein Bit immer für einen ganzen *Chunk* steht!
- Die *Anzahl* der zu löschenden Bits lässt sich, wie in **bf_alloc**, mit Hilfe des gegebenen Parameters `size` und der Hilfsfunktion **size_to_chunks** berechnen.
- Die vorgegebene Hilfsfunktion **clear_bit** löscht ein Bit im übergebenen Bitfeld.

c) Zusatzaufgabe: fmalloc, ffree (2 Sonderpunkte)

Implementiert in einem separaten Modul (**fmalloc.c, fmalloc.h**) die Funktionen **fmalloc** und **ffree**, die (unter Verwendung des zuvor erstellten **bf_alloc** und **bf_free**) die gleiche Schnittstelle zur Verfügung stellen sollen, wie die C-Funktionen **malloc(3)** und **free(3)**. **ffree** soll also nicht, wie zuvor **bf_free**, die Größe des freizugebenden Speichers übergeben bekommen!

Um euch die Größe des reservierten Speicherbereichs zu „merken“, sollt ihr ein bisschen mehr Speicher als notwendig allozieren und die Größe am Anfang des Speicherbereichs ablegen. So könnt ihr in **ffree** zunächst die Speichergröße auslesen und anschließend den zuvor allozierten Speicherbereich freigeben.

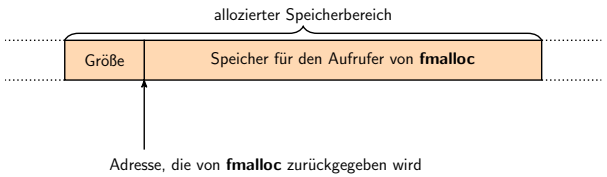

Abbildung 2: Reservierter Speicherbereich mit dazugehöriger Größe.

Tipps zu den Programmieraufgaben:

- Baut Testausgaben in euer Programm ein, um Programmierfehlern leichter auf die Schliche zu kommen. Die Hilfsfunktion **dump_free_mem** könnt ihr dazu verwenden, an geeigneter Stelle den momentanen Zustand der Freispeicher-Bitliste auszugeben.
- Zu Testzwecken empfiehlt es sich evtl. auch, die Speicherpool-Größe (`#define MEM_POOL_SIZE`) zu verkleinern. Achtet dabei darauf, dass sie durch die `CHUNK_SIZE` teilbar bleibt.
- Kommentiert euren Quellcode ausführlich, so dass wir auch bei Programmierfehlern im Zweifelsfall noch Punkte vergeben können!
- Denkt daran, dass viele Systemaufrufe fehlschlagen können! Fangt diese Fehlerfälle ab (die Aufrufe melden dies über bestimmte Rückgabewerte, siehe die jeweiligen man-Pages), gebt geeignete Fehlermeldungen aus (z.B. unter Zuhilfenahme von **perror(3)**), und beendet euer Programm danach ordnungsgemäß.