Abgabefrist Übungsaufgabe 4

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

- **W1** – eigene Übung U4 in KW 21 (06.06.-08.06.): Abgabe bis **Donnerstag, 16.06.2022 12:00 Uhr**
- **W2** – eigene Übung U4 in KW 22 (13.06.-15.06.): Abgabe bis **Montag, 20.06.2022 12:00 Uhr**

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

Allgemeine Hinweise zu den BS-Übungen

- Ab jetzt ist es *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://sys-sideshow.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

Aufgabe 4: Speicherverwaltung (10 Punkte)

ACHTUNG: Zu den Programmieraufgaben existieren Vorgaben in Form von C-Dateien mit einem vorimplementierten Code-Rumpf, die ihr in den Aufgaben erweitern sollt. Diese Vorgaben könnt ihr im Moodle[4] herunterzuladen, entpacken und vervollständigen. Die Datei `vorgaben-A4.tar.gz` lässt sich unter Linux/UNIX mittels `tar xf vorgaben-A4.tar.gz` entpacken.

Theoriefragen (4 Punkte)

1. Best Fit (2 Punkte)

Ein 16 MB großer Arbeitsspeicher wird in Blöcken der Größe 1 MB mit der Best Fit-Strategie verwaltet. Wenn für eine Anfrage nicht genügend Speicher zur Verfügung steht, wird diese abgelehnt und es werden keine Änderungen an der Speicherbelegung vorgenommen.

Zu Beginn ist der Speicher leer. Führt nacheinander die folgenden Anfragen der Prozesse A, B, C und D aus.

1. A fordert 4 MB an
2. B fordert 7 MB an
3. A fordert 1 MB an
4. B gibt seinen gesamten Speicher frei
5. C fordert 3 MB an
6. C fordert 7 MB an
7. D fordert 2 MB an.
8. A gibt seinen gesamten Speicher frei
9. D fordert 6 MB an

Gebt nach jeder Anfrage die aktuelle Speicherbelegung der einzelnen Prozesse an. Das folgende Format mit einer Zeile je Anfrage bietet sich dafür an:

```
|---- ---- ---- ----| 16 MB freier Speicher
|AAAA ---- ---- ----| A belegt 4 MB
...
```

⇒ `antworten.txt`

2. Verschnitt (1 Punkt)

Erklären Sie anhand der Best Fit-Strategie aus Aufgabe 1, um welche Art von Verschnitt es sich handelt und wie er zustandekommt.

In den folgenden Aufgaben wird das Buddy-Verfahren betrachtet. Kann es bei diesem Verfahren auch zu Verschnitt kommen? Begründen Sie Ihre Antwort.

⇒ `antworten.txt`

[4] `https://sys-sideshow.cs.tu-dortmund.de/Teaching/SS2022/BS/Material/vorgabe-A4.tar.gz`

3. Baumstruktur und Buddy-Verfahren (1 Punkt)

In der Programmieraufgabe wird als Datenstruktur zur Speicherverwaltung ein Binärbaum eingesetzt. Worin liegt der Vorteil, bei Einsatz des Buddy-Verfahrens, diese Datenstruktur (und keine lineare Datenstruktur wie z.B. eine Bitliste) zu wählen?

In der Programmieraufgabe wird ein Baum der Höhe 7 verwendet, d.h. unterhalb der Wurzel befinden sich sechs Ebenen mit Knoten. Was würde bei gleichbleibender Speichergröße eine abweichende Wahl der Baumhöhe bewirken?

⇒ `antworten.txt`

Programmierung: Speicherverwaltung mit dem Buddy-Verfahren (6 Punkte)

Die Speicherverwaltung moderner Linux-Systeme basiert auf dem Buddy-Verfahren. In dieser Übung sollt ihr die grundlegenden Operationen zur Bereitstellung und Freigabe von Speicher mit diesem Verfahren implementieren.

Zur Verwaltung der Speicherbelegungsdaten nutzen wir einen vollständigen Binärbaum[5]. Jedem Knoten dieses Baums ist ein Speicherbereich zugeordnet. Die Wurzel beschreibt den gesamten Arbeitsspeicher (2^n Bytes); bei jedem Knoten ist das linke Kind für die linke Hälfte und das rechte Kind für die rechte Hälfte (d.h. jeweils 2^{n-1}Bytes) des jeweiligen Speicherbereichs zuständig.

Wir verwalten Speicher in Blöcken von 64kB, so dass jedes Blatt des Binärbaums 2^0 Blöcken bzw. einem 64kB großen Speicherbereich entspricht. Die Knoten der nächsthöheren Baumebene verwalten je 2^1 Blöcke (also 128kB) je Knoten, die darüber je 2^2 (256kB) und so weiter.

Zusätzlich wird in jedem Knoten ein Zeichen (`char`) gespeichert, welches den Belegungszustand des zugeordneten Speicherbereiches angibt. Als Invariante der Baumstruktur gilt, dass nach jeder Speicherbelegung oder -freigabe jeder Knoten eines der folgenden Zeichen enthält:

- `NODE_FREE` bzw. `0` für „Der Speicherbereich ist frei oder Teil eines größeren belegten Speicherabschnitts“,
- `NODE_SPLIT` bzw. `-1` für „Der Speicherbereich ist teilweise belegt“ (d.h. der Speicherbereich wurde aufgeteilt und mindestens ein Speicherbereich im Teilbaum unterhalb dieses Knotens ist belegt) oder
- ein Zeichen $X > 0$ für „Der gesamte Speicherbereich dieses Knotens ist von Anfrage X belegt“.

Ein Beispiel für so verwaltete Speicherbereiche findet ihr in Abbildung 1. Hier sind insgesamt 4 MB Arbeitsspeicher vorhanden, von denen derzeit 1 MB von Anfrage A belegt ist.

Den Binärbaum und die Methoden zu dessen Traversierung geben wir bereits in der Datei `bst.c` vor. Diese enthält ebenfalls ein 4 MB großes Array als Hauptspeicher und Methoden, um die einem Knoten zugeordneten Speicheradressen zu erhalten. Dazu definieren wir den Datentypen **`node_t`** als Zeiger auf einen **`char`**, welcher die im Baum vermerkten Knotendaten (`NODE_FREE` / `NODE_SPLIT` / X) speichert. Hier findet ihr u.a. die folgenden Funktionen.

- `bst_root()` gibt die Wurzel des Baums zurück.

[5] Auch in der Praxis werden dafür häufig Bäume oder baumartige Strukturen verwendet. Der Linux-Kernel nutzt beispielsweise hierarchische Bitlisten, bei der jede Liste einer Baum-Ebene entspricht

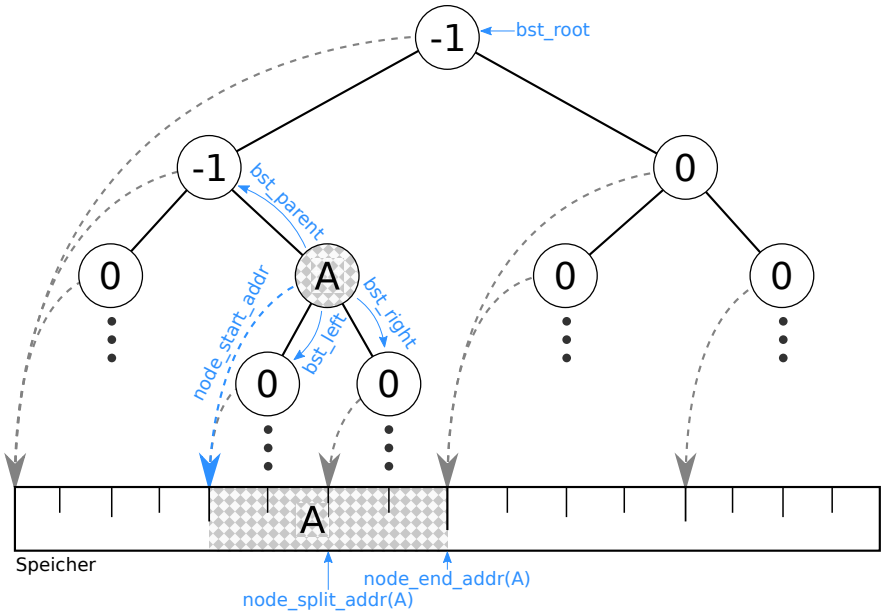

Abbildung 1:

- Für einen Knoten *node* liefern `bst_left`(*node*), `bst_right`(*node*) und `bst_parent`(*node*) das linke / rechte Kind bzw. den Elternknoten. Falls kein Kind oder kein Elternknoten existiert, geben diese Funktionen `NULL` zurück.
- `get_node_content`(*node*) gibt den im Knoten gespeicherten Wert (d.h. `NODE_FREE` / `NODE_SPLIT` / *X*) zurück, `set_node_content`(*node*, *value*) verändert ihn. Als Alternative zur Nutzung dieser Funktionen könnt ihr auch direkt `*node` lesen und schreiben.
- Die kleinste und größte Adresse des zugeordneten Speicherbereichs lassen sich mittels `node_start_addr`(*node*) und `node_end_addr`(*node*) abfragen. Die Hilfsfunktion `node_split_addr`(*node*) gibt zudem die Startadresse des Speicherbereichs des rechten Kindes (d.h. die „Mitte“ des Speicherbereichs des aktuellen Knotens) zurück – oder `NULL`, falls der Knoten kein Kind hat. Mit `node_width_chunks`(*node*) könnt ihr die Größe des Speicherbereichs in 64 kB-Blöcken abfragen.
- Zur Umrechnung von einer angeforderten Speichermenge in eine entsprechende Menge an Blöcken könnt ihr die Methode `size_to_chunks`(*bytes*) verwenden. Diese rundet die angefragte Speichermenge auf eine Zweierpotenz der Blockgröße auf und gibt die Anzahl an benötigten 64 kB-Blöcken zurück.
- Die Funktion `bst_dump()` gibt die im Baum gespeicherten Werte ebenenweise (beginnend bei der Wurzel) auf der Standardausgabe aus.
- Die Funktion `memory_dump()` gibt die Speicherbelegung auf der Standardausgabe aus.

Der Knoten für die Anfrage A in Abbildung 1 ist beispielsweise per `node_t node = bst_right(bst_left(bst_root()));` erreichbar.

a) Speicherfreigabe (3 Punkte)

Implementiert die Funktion `buddy_free(void *addr)` in der Datei `4a.c`. Diese Funktion wird aufgerufen, um einen zuvor reservierten Speicherbereich wieder freizugeben, und erhält die Startadresse des Speicherbereichs als Argument. Ihre Aufgabe ist, den zugehörigen Knoten im Baum zu finden und als frei zu markieren. Zusätzlich muss sie sicherstellen, dass

die Invariante der Baumstruktur weiterhin gilt. Hierzu sollt ihr die von uns bereitgestellte Funktion `bst_housekeeping(`*node*`)` nutzen, die alle mit `NODE_SPLIT` markierten Knoten unterhalb von *node* besucht und sie, sofern ihre Teilbäume beide frei sind, als `NODE_FREE` vermerkt (d.h. die beiden Buddies zusammenfasst).

Falls die Adresse `NULL` übergeben wird, soll nichts passieren. Bei anderweitig ungültigen Adressen, die z.B. nicht der Startadresse eines belegten Speicherbereichs entsprechen, soll eine Fehlermeldung ausgegeben und das gesamte Programm mit dem Rückgabewert 255 beendet werden.

Hinweis:

- Die Funktion hat kein Wissen über die Größe des Speicherbereichs oder die Bezeichnung der zugehörigen Anfrage. Diese Informationen sind zur Speicherfreigabe nicht notwendig.
- Ob eine Startadresse `void *addr` von einem Node belegt wird kann mit `node_start_addr(node) == addr` mit geprüft werden.

Zum Testen eurer Implementierung geben wir in der Datei `test_4a.c` einen teilweise belegten Speicher und eine Folge von `buddy_free`-Aufrufen vor. Dieses Testprogram könnt ihr mit dem Befehl `make 4a` übersetzen und ausführen. Ergänzt es um eigene Testfälle , da unsere Vorgabe nicht notwendigerweise alle Randfälle abdeckt.

⇒ `4a.c`

b) Speicherallokation (3 Punkte)

Implementiert die Funktion `buddy_alloc(char request_id, size_t size)` in der Datei `4b.c`. Diese soll für die Speicheranfrage mit dem Bezeichner `request_id` einen Speicherbereich reservieren, der mindestens `size` Bytes groß ist, und die Startadresse des reservierten Speicherbereichs zurückgeben. Dazu müsst ihr einen unbelegten Knoten mit einer passenden Blockgröße suchen und für diese Anfrage belegen. Ebenso müsst ihr sicherstellen, dass nach Abschluss der Funktion die Invariante weiterhin gilt. Hierzu geben wir keine Hilfsfunktion vor.

Falls 0 Bytes anfgefragt werden oder kein genügend großer freier Speicherbereich mehr vorhanden ist, soll die Funktion NULL zurückgeben.

Hinweis:

- Die Implementierung einer rekursiven Hilfsfunktion kann sich hier als nützlich erweisen.
- Das Buddy verfahren aus der Vorlesung präferiert beim allozieren bereits geteilte Nodes vor ungeteilten. Ihr müsst das nicht berücksichtigen!

Zum Testen könnt ihr die Datei `test_4b.c` und den Befehl `make 4b` verwenden. Im Gegensatz zu Teil a) haben wir hier keine Assertions eingebaut, so dass ihr selbst entscheiden müsst, ob die einzelnen Speicherbelegungen richtig sind. Beachtet zudem, dass unser vorgegebenes Testprogramm auch die Funktion `buddy_free` aus Aufgabenteil a) aufruft.

⇒ `4b.c`

c) Effiziente Speicherfreigabe (Bonusaufgabe, 1 Punkt)

Die in Aufgabenteil a) implementierte Speicherfreigabe ist nicht sehr effizient, da sie nach jeder Freigabe zur Aufrechterhaltung der Invariante in `bst_housekeeping()` den kompletten Baum traversieren muss. Bei geschickter Umsetzung ist dies nicht notwendig. Implementiert daher hier die Funktion `buddy_fast_free(void *addr)` in der Datei `4c.c`. Diese soll sich wie `buddy_free(void *addr)` verhalten, darf zur Aufrechterhaltung der Invariante aber weder `bst_housekeeping()`, noch einen Ansatz, der den kompletten Baum besucht, benutzen. Zudem erlauben wir hier keine rekursiven Funktionen.

Auch hier könnt ihr mittels `test_4c.c` und `make 4c` eure Implementierung testen.

⇒ `4c.c`

- Baut Testausgaben in euer Programm ein, um Programmierfehlern leichter auf die Schliche zu kommen.
- 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 und beendet euer Programm danach ordnungsgemäß.
- Die Programme sollen sich mit dem gcc auf den Linux-Rechnern im IRB-Pool übersetzen lassen. Zum Compilieren könnt ihr das mitgelieferte Makefile nutzten
- Achtet darauf, dass sich der Programmcode ohne Warnungen übersetzen lässt.
- Alternativ kann auch der GNU C++-Compiler (*g++*) verwendet werden.
- Bei Fragen oder Problemen könnt ihr euch gerne an unseren **HelpDesk** wenden.