

Weitere Hinweise zu den Übungen Betriebssysteme

- Die abgegebenen Antworten/Programme werden automatisch auf Ähnlichkeit mit anderen Abgaben überprüft. Wer beim Abschreiben¹ 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.

Aufgabe 4: Speicherverwaltung (10 Punkte)

In diesem Übungsblatt sollt ihr euch mit der Speicherverwaltung auf dem Heap beschäftigen. Um die Feinheiten und Schwierigkeiten entsprechender Algorithmen besser zu verstehen, sollt ihr im Programmier Teil eine einfache Speicherverwaltung nach dem First-Fit-Verfahren selbst implementieren.

Theoriefragen (5 Punkte)

1. Verschnitt (1 Punkt)

Beschreibt in eigenen Worten, was man unter Verschnitt versteht. Beschreibt dabei insbesondere den Unterschied zwischen internem und externem Verschnitt und warum man häufig internen Verschnitt beim Allokieren von Speicher riskiert.

⇒ antworten.txt

2. Seitenadressierung (2 Punkte)

Beantwortet in eigenen Worten folgende Fragen zum Thema Seitenadressierung (*paging*):

- Bei der Einlagerung einer Seite kann sich der ihr zugeordnete physikalische Adressbereich ändern. Muss das Betriebssystem diesen Fall gesondert behandeln, damit Programme weiterhin auf Daten in der Seite zugreifen können? Begründet eure Antwort.
- Warum muss bei einem Kontextwechsel der TLB geleert werden?

⇒ antworten.txt

3. First Fit (2 Punkte)

Verwendet die First-Fit-Strategie, um die nachfolgende Reihe von Speicheranfragen umzusetzen, und notiert eure Ergebnisse wie in der angegebenen Tabelle. Ein Feld entspricht dabei einem MiB und es stehen insgesamt 32 MiB zur Verfügung:

A freigeben, F belegt 4 MiB, A belegt 2 MiB, B freigeben,
E freigeben, E belegt 7 MiB, E freigeben, E belegt 4 MiB

+-----+			
Start	AAABBB	CCCCC	DDDDDEEEEE
A: Freigeben	BBB	CCCCC	DDDDDEEEEE
F: Belegt 4 MiB	BBB	CCCCCFFFF	DDDDDEEEEE
...	...		
+-----+			

⇒ antworten.txt

¹Da wir im Regelfall nicht unterscheiden können, wer von wem abgeschrieben hat, gilt das für Original **und** Plagiat.

Programmierung: Speicherverwaltung nach dem First-Fit-Verfahren (5 Punkte)

In dieser Aufgabe sollt ihr eine kleine Speicherverwaltung (angelehnt an die Funktionalität von `malloc(3)` und `free(3)`) selbst implementieren. Dazu geben wir euch einen **256 KiB großen Speicherbereich** vor, der in **Blöcke (Chunks) von 4 KiB** aufgeteilt ist und von euch verwaltet werden soll. **Zur Verwaltung** verwenden wir zunächst **ein Feld**, welches für jeden Block **eine Metadatenstruktur vom Typ `mem_info`** enthält. Diese gibt an, ob am jeweiligen Block ein zusammenhängender Speicherbereich einer bestimmten Länge beginnt (`length`) und wenn ja, ob dieser frei oder belegt ist (`status`).

Beispiel:

Für vier belegte Bereiche in einem 128 KiB großen Speicher können die Verwaltungsinformationen folgendermaßen aussehen. Hier steht **A** für belegt (*allocated*), **F** für frei (*free*) und **Sternchen** dafür, dass der status ignoriert werden kann, da `length == 0`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
heap {	A								frei				B				C				frei				D							
status {	A	*								F	*				A	*				A	*				F	*				A	*	
length {	9	0	0	0	0	0	0	0	0	5	0	0	0	0	6	0	0	0	0	0	4	0	0	0	5	0	0	0	0	3	0	0

Unter <https://ess.cs.tu-dortmund.de/Teaching/SS2019/BS/Downloads/vorgabe-A4.tar.gz> findet ihr ein Makefile und eine vorgegebene Modulstruktur, in die ihr eure Lösungen integrieren sollt. Das Archiv kann wie gewohnt mit `tar -xzf vorgabe-A4.tar.gz` entpackt werden.

Den simulierten Hauptspeicher geben wir bereits als **Array `heap`** in der Datei `firstfit.c` vor. Diese enthält ebenfalls die **Liste `allocation_list` vom Typ `mem_info[]`**, in welcher ihr den Belegungsstatus für jeden Speicherblock ablegen sollt. Im Detail stehen euch die folgenden Hilfsmittel zur Verfügung.

- Die **`mem_info`-Struktur** enthält die **Member `status` und `length`**. Ist `length == 0`, so beginnt am zugehörigen Speicherblock kein zusammenhängender Speicherbereich und die `status`-Angaben werden nicht berücksichtigt. Andernfalls beginnt an dieser Stelle ein `length` Blöcke umfassender Speicherbereich, der entweder frei (`status == CHUNK_FREE`) oder belegt (`status == CHUNK_ALLOCATED`) ist.
- Auf die Blockgröße in Bytes könnt ihr über das **Makro `CHUNK_SIZE`** zugreifen, die **Speichergröße** (in Chunks) ist über `NUM_CHUNKS` verfügbar.
- `size_to_chunks(bytes)` gibt die zur Speichergröße `bytes` korrespondierende Anzahl an Speicherblöcken zurück. Falls nötig, wird aufgerundet.
- Für **Größenangaben und Array-Indizes** verwenden wir den plattformunabhängigen Typ `size_t`. Im Regelfall entspricht dieser einem `unsigned long int`.
- `dump_memory()` gibt den aktuell in `allocation_list` vermerkten Belegungsstatus auf der Standardausgabe aus und führt einige einfache **Konsistenztests** durch. Eine Raute (#) steht für einen belegten Speicherblock, ein Punkt (.) für einen freien.

a) Speicherallokation (2,5 Punkte)

Implementiert die Funktion `ff_alloc(size_t size)` in der Datei `4a.c`. Diese soll nach dem First-Fit-Verfahren einen mindestens `size` Bytes großen Speicherbereich belegen und die Startadresse des belegten Speicherbereichs zurückgeben.

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

Verwendet zum Testen den Befehl `make 4a`. Dieser führt einige in `test_4a.c` definierte Speicherbelegungen durch und gibt dabei jeweils die zurückgegebene Speicheradresse und die aktuelle Speicherbelegung aus.

Ob die Ausgaben richtig sind oder nicht, müsst ihr selbst entscheiden. Zudem garantieren wir nicht, dass die von uns vorgegebene `test_4a.c` alle möglichen Randfälle abdeckt – es könnte sich also lohnen, sie um zusätzliche Randfälle zu erweitern. Die Tests sollen von euch allerdings nicht mit abgegeben werden und sind somit nicht Teil der bewerteten Aufgabenstellung.

⇒ `4a.c`

b) Speicherfreigabe (2,5 Punkte)

Implementiert die Funktion `ff_free(void *addr)` in der Datei `4b.c`. Diese soll einen zuvor belegten Speicherbereich wieder freigeben und erhält die Startadresse des Speicherbereichs als Argument. Dazu muss die `allocation_list` entsprechend aktualisiert werden. Wenn die Speicherbereiche vor und/oder nach dem freigegebenen Speicherbereich ebenfalls frei sind, müsst ihr diese zu einem neuen (großen) Freispeicherbereich vereinigen.

Falls die Adresse `NULL` übergeben wird, soll nichts passieren. Bei anderweitig ungültigen Adressen, die nicht der Startadresse eines belegten Speicherbereichs entsprechen oder gar nicht erst im von euch verwalteten Arbeitsspeicher liegen, soll eine Fehlermeldung ausgegeben und das gesamte Programm mit dem Rückgabewert 255 beendet werden.

Verwendet zum Testen den Befehl `make 4b`. Dieser führt die in `test_4b.c` definierten Speicherbelegungen und -freigaben durch und gibt dabei jeweils die aktuelle Speicherbelegung aus. Auch hier müsst ihr selbst entscheiden, ob die Ausgaben richtig sind, und solltet ggf. weitere Testfälle hinzufügen.

⇒ `4b.c`

Zusatzaufgabe: Speicherallokation nach dem Best-Fit-Verfahren (2 Punkte)

Nun soll die Funktion `bf_alloc(size_t size)` in der Datei `4extended.c` implementiert werden. Diese soll nach dem Best-Fit-Verfahren einen Speicherbereich belegen, der mindestens `size` Bytes groß ist und wie bei a) die Startadresse des belegten Speicherbereichs zurückgeben.

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

Verwendet zum Testen den Befehl `make 4extended`. Dieser führt einige in `test_4extended.c` definierte Speicherbelegungen durch und gibt dabei jeweils die zurückgegebene Speicheradresse und die aktuelle Speicherbelegung aus.

⇒ `4extended.c`

Tipps zu den Programmieraufgaben:

- 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.
- Die Programme sollen sich mit dem gcc auf den Linux-Rechnern im IRB-Pool übersetzen lassen. Es ist das mitgelieferte Makefile (Kommando `make`) zu verwenden, oder der Compiler mit den folgenden Parametern aufzurufen::
`gcc -std=c11 -Wall -o ziel datei.c`
Weitere (nicht zwingend zu verwendende) nützliche Compilerflags sind: `-Wpedantic`
`-Werror` `-D_POSIX_SOURCE`
- Achtet darauf, dass sich der Programmcode ohne Warnungen übersetzen lässt; z.B. durch Nutzung von `-Werror`.
- Alternativ kann auch der GNU C++-Compiler (`g++`) verwendet werden.

Abgabe: bis spätestens Donnerstag, den 13. Juni 10:00 (Übungen in ungerader Kalenderwoche) bzw. Dienstag, den 18. Juni 10:00 (Übungen in gerader Kalenderwoche).