Lehrstuhl Connected Mobility
Fakultät für Informatik
Technische Universität München

Grundlagen von Betriebssystemen und Systemsoftware

WS 2018/19

Übung 9: Speicherverwaltung

zum 17. Dezember 2018

- Die **Hausaufgaben** für dieses Übungsblatt müssen **spätestens** am **Sonntag, den 06.01.2019 um 23:59** in Moodle hochgeladen worden sein.
- Alle C-Programme müssen mit den folgenden Flags kompiliert werden:
  `gcc -Wall -o <progname> <prog.c>`.
- Hierzu stellen wir für jedes Übungsblatt jeweils ein Makefile bereit, das **nicht** verändert werden darf, um sicherzustellen, dass Ihre Abgabe auch korrekt von uns getestet und bewertet werden kann. **Ihr Programmcode muss zwingend mit dem Makefile kompilierbar sein. Anderenfalls kann die Abgabe nicht bewertet werden.** Wenn Sie zwischenzeitlich Änderungen vornehmen wollen, um etwa bestimmte Teile mittels `#ifdef` einzubinden und zu testen, dann kopieren Sie am besten das Makefile und modifizieren Ihre Kopie. Mit `make -f <Makefile>` können Sie dann eine andere Datei zum Übersetzen verwenden.
- Die Abgabe der Programme erfolgt als Archivdatei, die die verschiedenen Quelldateien (.{c|h}) umfasst und **nicht** in Binärform. Nicht kompilierfähiger Quellcode wird **nicht gewertet**.
- Um die Abgabe zu standardisieren enthält das Makefile ein Target "submit" (`make submit`), was dann eine Datei blatt09.tgz zum Hochladen erzeugt.
- Damit die richtigen Dateien hochgeladen und ausgeführt werden, geben wir bei allen Übungen die jeweils zu verwendenden Dateinamen für den Quellcode und auch für das Executable an.
- Die Tests werden auf diesem Aufgabenblatt mittels Python-Programmen durchgeführt. Zum Ausführen der Tests geben Sie `python xyz_test.pyc` auf der Konsole ein. Ihre ausführbaren Programme müssen sich im selben Verzeichnis wie die Testprogramme befinden.

1 Vorbereitung auf das Tutorium

Was ist die Rolle der folgenden POSIX- (Portable Operating System Interface) bzw. Bibliotheksfunktionen? Erläutern Sie kurz die Parameter und den Rückgabewert jeder Funktion. [1]

- `void *calloc(size_t nmemb, size_t size);`
- `void *alloca(size_t size);`
- Machen Sie sich mit der Verwendung des Datentyps `enum` in C vertraut.

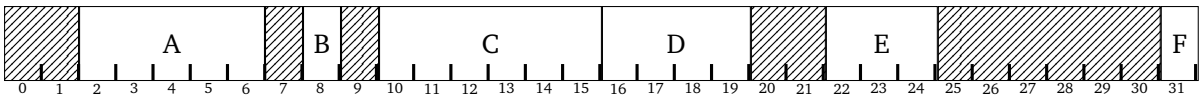
2 Tutoraufgaben

2.1 Verwaltung freien Speichers

In dieser Aufgabe werden die in der Vorlesung vorgestellten Verfahren zur Verwaltung freien Speichers vertieft.

2.1.1 Bitmaps

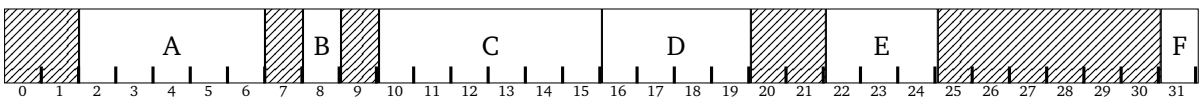
Gegeben Sei die folgende Speicherbelegung (32 Blöcke):



Fertigen Sie eine Bitmap an, welche den Belegungszustand korrekt widergibt.

2.1.2 Verkettete Listen

Gegeben Sei die folgende Speicherbelegung (32 Blöcke):



Fertigen Sie eine verkettete Liste an, welche den Belegungszustand korrekt widergibt.

2.1.3 Buddy-Algorithmus

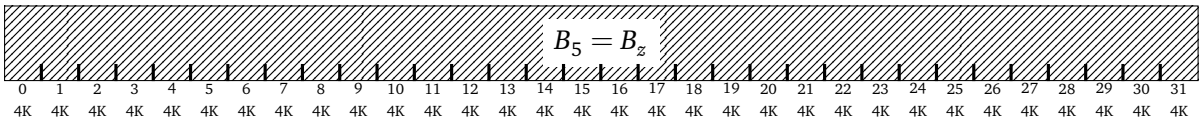
Gegeben sei der Buddy-Allocator-Algorithmus gemäß Vorlesung, sowie der Pseudocode des folgenden Programm. Gehen Sie von einer Blockgröße von 4096 Bytes (4K) und einer Gesamtspeichergröße von 131072 Bytes (128K) aus.

[1] Erläuterungen und Beispiele der einzelnen Funktionen finden Sie auch in den **man**-Pages auf Ihrer Linux-VM.

```
A = allocate(13337);
B = allocate(24242);
C = allocate(8193);
D = allocate(13)
free(A);
E = allocate(32768);
free(B);
F = allocate(10000);
G = allocate(12345);
H = allocate(11111);
free(G);
free(D);
free(H);
free(C);
free(F);
free(E);
```

Zeichnen Sie die Belegung des Speichers nach der Ausführung jedes Statements aus dem obigen Pseudocode auf. Legen Sie besonderen Wert darauf, Blockgrößen sowie deren Belegungszustand genau zu kennzeichnen. Berechnen Sie außerdem nach jedem Schritt die Größe des Verschnitts aller Blöcke.

Gehen Sie von der folgenden Initialbelegung des Speichers aus:



2.2 Strategien zur Speicherallokation I

Es seien folgende Speicherverwaltungsstrategien gegeben:

- First Fit
- Next Fit
- Best Fit
- Worst Fit

Für die Bewertung und den Vergleich dieser Strategien zum Belegen und Freigeben von zusammenhängenden Speicherbereichen seien die folgenden Eigenschaften von Interesse:

a) die Ausnutzung des Speichers,

b) die Art der Zerstückelung des Speichers (Fragmentierung), die damit verbundene Anzahl der Freibereiche und der Suchaufwand sowie

c) der Aufwand zur Erstellung und Führung der Verwaltungsstrukturen.

Erklären Sie kurz die Funktionsweise der vier gegebenen Strategien und vergleichen Sie diese im Hinblick auf die vorgenannten Eigenschaften.

2.3 Funktionen zur Speicherallokation

Diskutieren Sie die Unterschiede zwischen den verschiedenen Funktionen zur Allokation von Speicher:

- `void *malloc (size_t size);`
- `void *realloc(void *ptr, size_t size);`
- `void *calloc(size_t nmemb, size_t size);`
- `void *alloca(size_t size);`
- `int brk (void *addr);`
- `int sbrk (int *increment);`

3 Hausaufgabe (10 Punkte)

Realisieren Sie Ihre eigene (Frei-)Speicherverwaltung für einen Prozess. Diese soll die beiden Strategien *First Fit* und *Worst Fit* realisieren. Schreiben Sie hierzu eine Bibliothek zur Speicherverwaltung, die – unter Nutzung Ihrer Listenverwaltung – die folgenden Funktionen bereitstellen soll:

- `int mem_init (unsigned int size, unsigned int blocksize, enum mem_algo strategy);`

  Diese Funktion alloziert einen zusammenhängende Speicherbereich von `size` Bytes zur internen Vergabe und merkt sich für die interne Verwaltung vor, dass die Vergabe in der Granularität in Einheiten von `blocksize` Bytes passieren soll. Als enum können Sie folgendes als Vorlage nutzen.

```
enum mem_algo { none, first, worst };
```

- `void *mem_alloc (unsigned int size);`

  Sucht nach einem vorgegebenen Freispeicherverwaltungs das nächste passende Stück Speicher, markiert diesen als belegt und einen Pointer darauf zurück. Dabei wird ggf. ein vorhandener größerer Speicherblock so geteilt, dass der Verschnitt kleiner ist als `blocksize`.

- `void mem_free (void *addr);`

  Prüft, ob der freizugebende Speicher wirklich von `mem_alloc` alloziert wurde und gibt dann den allozierten Speicher wieder frei. Dabei werden nach der Freigabe benachbarte Speicherblöcke zusammengeführt.

- `void mem_dump ();`

  Gibt den Zustand aller Speicherblöcke in folgendem Format aus:

  `[Adresse P|F Genutzt/Laenge]`

  Beispiel:

  `[0x7ffeed0b5abc F 0/4096] [0x7ffeed0aeabc P 4096/4096]`

  Verwenden Sie konkrete folgenden Formatstring für die Ausgabe mit `printf(3)`:

  ```
  "[%p %c %d/%d] "
  ```

  Kombinieren Sie diese Ausgabe mit Ihrem `list_print()` und geben Sie alle Speicherblöcke in einer Zeile, durch Leerzeichen getrennt aus.

  ```
  1:[0x7fbb1c801000 P 1000/1000] 2:[0x7fbb1c8013e8 F 0/9000]
  ```

  Wichtig: Rufen Sie `mem_dump()` einmalig nach `mem_init()` auf, um den initialen Zustand Ihrer Speicherbelegung anzuzeigen und dann nach jedem Aufruf von `mem_alloc()` bzw. `mem_free()`, so dass Ihr Programm schrittweise die Belegung und Freigabe des Speichers visualisiert. Siehe die Beispielausgaben unten.

Nutzen Sie Ihre Listenverwaltung, um den Speicher zu verwalten. Verwenden Sie nur eine einzige Liste, die nach den Anfangsaddressen der Speicherblöcke sortiert ist; dann ist auch das spätere Zusammenführen von Speicherbereichen leicht realisierbar. Bedenken Sie, dass beim Allozieren eines Speicherblocks einen großen Block potentiell in zwei kleine teilen müssen. Hierzu können Sie Ihrer Listenverwaltung beispielsweise eine weitere Funktion mit folgender Signatur hinzufügen:

```
struct list_elem list_put (list_t *list, struct list_elem *current, char *data);
```

Diese Funktion fügt ein neues Datenelement **nach** dem Element `current` ein, wofür sie natürlich ein Listenelement allozieren und dann einfügen muss.

Als Basis für Ihre Speicherverwaltung können Sie die folgende Datenstruktur verwenden, diese bei Bedarf mit Erweiterungen versehen oder einnfach nur als Inspiration nutzen.

```
struct memblock {
    enum { free, used }  status;  /* ist der Speicherblock frei   */
    char                 *addr;   /* Adresse des Speicherblocks   */
    unsigned int         size;    /* Laenge des Speicherblocks    */
    unsigned int         in_use;  /* Anteil zugewiesener Speicher; 0 wenn status == free */
};
```

Ihr Programm `mem` soll die folgenden Parameter von der Kommandozeile akzeptieren:

- `-m <total memory>`

  zeigt an, wieviel Speicher zur internen Verwaltung vergeben werden soll. Dieser Speicherbereich soll einmalig alloziiert werden und initial vom Programm ausgegeben werden.
  (Default: 1048576 = 256 * 4096)

- `-b <blocksize>`

  gibt an, wie groß die zur Speicherverwaltung genutzten Blöcke sein sollen.
  (Default: 4096)

- `-1|-w`

  wählt den Speicherverwaltungsalgorithmus aus: `-1` → First Fit, `-w` → Worst Fit.

- `-a <size>`

  alloziert einen Speicherblock der Größ `size` aus dem Speicher. Dieser Parameter kann wiederholt angegeben werden. Die Allozierungen werden in der Reihenfolge des Auftretens durchgführt.

- `-f <n>`

  gibt den *n*-ten allozierte Speicherblock wieder frei. Dieser Parameter kann wiederholt angegeben werden. Die Allozierungen werden in der Reihenfolge des Auftretens durchgführt.

  Achtung: *n* soll ab 1 zählen, d.h., `-f 1` refenziert den ersten mit `-a` allozierten Speicherblock.

Wichtig: Die Parameter `-a` und `-f` können gemischt angeben werden. So führt beispielsweise die folgende Angabe

-a 1000 -a 2000 -f 1 -a 500 -f 2 -a 4000 -f 4 -f 3

dazu, dass erst 1000 und dann 2000 Bytes alloziert werden, dann wird der zuerst allozierte Block (1000 Bytes) wieder freigegeben, dann werden 500 Bytes alloziert, dann wird der zweite Block (2000 Bytes) freigegeben, dann 4000 Bytes alloziert und schließlich das zuletzt (4000 Bytes) und dann das 3. (500 Bytes) allozierte Stück Speicher wieder freigegeben.

Beispiele:

Achtung: Hier sind künstliche Zeilenumbrüche eingefügt, damit die Zeilen ganz sichtbar sind (angedeutet durch eine leichte Indentierung). In Ihrer Ausgabe sollen keine Zeilenumbrüche innerhalb eines `mem_dump()` vorhanden sein.

```
$ ./mem -m 10000 -b 1000 -l -a 1000 -f 1
1:[0x7ffdc0801000 F 0/10000]
1:[0x7ffdc0801000 P 1000/1000] 2:[0x7ffdc08013e8 F 0/9000]
1:[0x7ffdc0801000 F 0/10000]

$ ./mem -m 10000 -b 1000 -l -a 1000 -a 1000 -f 1 -f 2
1:[0x7fcea0001000 F 0/10000]
1:[0x7fcea0001000 P 1000/1000] 2:[0x7fcea00013e8 F 0/9000]
1:[0x7fcea0001000 P 1000/1000] 2:[0x7fcea00013e8 P 1000/1000] 3:[0x7fcea00017d0 F 0/8000]
1:[0x7fcea0001000 F 0/1000] 2:[0x7fcea00013e8 P 1000/1000] 3:[0x7fcea00017d0 F 0/8000]
1:[0x7fcea0001000 F 0/10000]

$ ./mem -m 10000 -b 1000 -l -a 1000 -a 1000 -a 1000 -f 2 -a 500
1:[0x7fee79001000 F 0/10000]
1:[0x7fee79001000 P 1000/1000] 2:[0x7fee790013e8 F 0/9000]
1:[0x7fee79001000 P 1000/1000] 2:[0x7fee790013e8 P 1000/1000] 3:[0x7fee790017d0 F 0/8000]
1:[0x7fee79001000 P 1000/1000] 2:[0x7fee790013e8 P 1000/1000] 3:[0x7fee790017d0 P 1000/1000]
 4:[0x7fee79001bb8 F 0/7000]
1:[0x7fee79001000 P 1000/1000] 2:[0x7fee790013e8 F 0/1000] 3:[0x7fee790017d0 P 1000/1000]
 4:[0x7fee79001bb8 F 0/7000]
1:[0x7fee79001000 P 1000/1000] 2:[0x7fee790013e8 P 500/1000] 3:[0x7fee790017d0 P 1000/1000]
 4:[0x7fee79001bb8 F 0/7000]

$ ./mem -m 10000 -b 1000 -w -a 1000 -a 1000 -a 1000 -f 2 -a 500
1:[0x7f8b55800000 F 0/10000]
1:[0x7f8b55800000 P 1000/1000] 2:[0x7f8b558003e8 F 0/9000]
1:[0x7f8b55800000 P 1000/1000] 2:[0x7f8b558003e8 P 1000/1000] 3:[0x7f8b558007d0 F 0/8000]
1:[0x7f8b55800000 P 1000/1000] 2:[0x7f8b558003e8 P 1000/1000] 3:[0x7f8b558007d0 P 1000/1000]
 4:[0x7f8b55800bb8 F 0/7000]
1:[0x7f8b55800000 P 1000/1000] 2:[0x7f8b558003e8 F 0/1000] 3:[0x7f8b558007d0 P 1000/1000]
 4:[0x7f8b55800bb8 F 0/7000]
1:[0x7f8b55800000 P 1000/1000] 2:[0x7f8b558003e8 F 0/1000] 3:[0x7f8b558007d0 P 1000/1000]
 4:[0x7f8b55800bb8 P 500/1000] 5:[0x7f8b55800fa0 F 0/6000]
```

Syntax: `mem [-m <total-memory>] [-b <blocksize>] (-1|-w) {-a <alloc mem>|-f <alloc#>}*`

Quellen: mem.c memory.c memory.h list.c list.h

Executable: mem