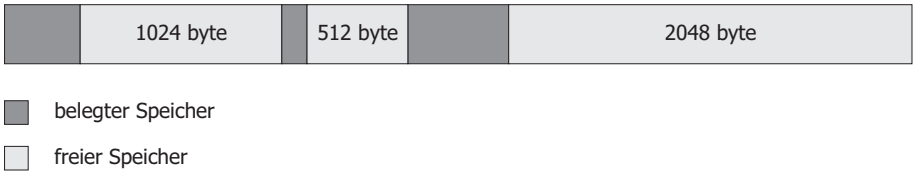
Übung 04

Betriebs- und Kommunikationssysteme
ProInf IV 2020
Simon Schmitt, Heiko Will

Abgabe freiwillig

Aufgabe 1: Memory Management

Gegeben sei folgende Belegung eines Speichers:



Weiterhin seien drei Belegungsmethoden für Speicherplatzanforderungen gegeben:

First-Fit: Belege von vorne beginnend den ersten freien Speicherbereich, der groß genug ist, die Anforderung zu erfüllen.

Rotating-First-Fit: Wie First-Fit, jedoch wird von der Position der vorherigen Platzierung ausgehend ein passender Bereich gesucht.

Wird das Ende des Speichers erreicht, so wird die Suche am Anfang des Speichers fortgesetzt (maximal bis zur Position der vorherigen Platzierung). Bei der ersten Anforderung beginnt die Suche am Anfang des Speichers.

Best-Fit: Belege den kleinsten freien Speicherbereich, in den die Anforderung passt.

Wie sieht der obige Speicher aus, wenn nacheinander vier Anforderungen der Größe 300 B, 512 B, 2048 B und 624 B ankommen? Notieren Sie für jede Strategie die freien Speicherbereiche nach jeder Anforderung, und geben Sie an, für welche Methoden die Anforderungen erfüllt werden können!

Beachten Sie, dass die Daten jeweils linksbündig in einer Lücke abgelegt werden und einmal belegte Speicherbereiche nicht wieder freigegeben werden!

Aufgabe 2: Speicherverwaltung programmieren

Entwickeln Sie eine *eigene* Speicherverwaltung. Funktionen zur Speicherverwaltung aus Bibliotheken sollen nicht genutzt werden (kein `malloc()` o. Ä.).

1. Simulieren Sie den Hauptspeicher durch ein char-Array `memory` der Größe `MEM_SIZE`.
2. Hierzu sollen Sie die folgenden Funktionen implementieren:
   a) `void memory_init()`: Initialisiert den zur Verfügung stehenden Speicherbereich und etwaige Verwaltungsdaten.
   b) `void *memory_allocate(size_t byte_count)`: Gibt einen Zeiger auf einen zusammenhängenden Speicherbereich der Größe `byte_count` zurück.
   c) `void memory_free(void *pointer)`: Gibt einen von `memory_allocate` reservierten Speicherbereich wieder frei.
   d) `void memory_print()`: Visualisiert den aktuellen Zustand des Speichers bzw. gibt Informationen über den Zustand aus.
   e) `void *memory_by_index(size_t index)`: Gibt einen Pointer auf das n-te Element in der Liste zurück, `NULL`, falls nicht existent.
3. Nutzen Sie eine geeignete Struktur für Ihre Daten (Hinweis: verkettete Liste[1]).
4. Die Daten zur Verwaltung des Speichers müssen in demselben Speicher liegen. Es hilft, sich die Struktur ggf. vorher auf einem Blatt Papier aufzumalen.
5. Überlegen Sie sich Testfälle für Ihre Funktionen und testen Sie diese mit unserem Wrapper.
6. Berücksichtigen Sie eventuell auftretende Sonderfälle.

Der Wrapper nutzt intern die GNU/Readline Bibliothek, weshalb zusätzlich die Flag `-lreadline` benutzt werden muss, d. h. die Kompilation ist wiefolgt:

```
$ c99 -o memory -Wall -Wextra -pedantic -O2
  memory.c memory_wrapper.c -lreadline
```

Hinweis: Unser Wrapper bietet Ihnen ein „Prompt“ um Ihr Programm *interaktiv* zu testen.

```
$ ./memory
Initializing Memory
Usage:
 a <bytes>      allocate memory
 f <index>      free block at index
 p              print block list
 q              quit
> a 100
Allocated 100 bytes
```

[1] Eine kreisweise (doppelt) verkettete Liste (linked list) ist eine dynamische Datenstruktur und besteht aus einer Menge von Knoten, die untereinander verkettet sind. Jeder Knoten besteht aus dem zu speichernden Objekt und einem Zeiger auf das nächste sowie vorherige Element. Das letzte Element der Liste zeigt wieder auf den ersten Knoten. Mittels eines Zeigers head wird auf den ersten Knoten in der Liste gezeigt.

```
> a 20
Allocated 20 bytes
> p
Block 0 of size 100 (USED)
Block 1 of size 20 (USED)
Block 2 of size 808 (FREE)
> q
```

Sie können alternativ die Eingabe automatisieren, indem Sie eine Textdatei anlegen, die die Eingabe enthält und diese mittels der Shell an den Wrapper übergeben. Bspw. könnte eine solche Datei wiefolgt aussehen:

```
a 100
a 20
p
q
```

Sie rufen dann das Programm wiefolgt auf:

```
$ ./memory < testfile.txt
```

Die Ausgabe ist dieselbe.