SYSTEMPROGRAMMIERUNG: 04b. Aufgabenblatt

WS 2013/2014
Prof. Dr.-Ing. C. Gips, Dipl.-Inf. Birgit C. George

FH Bielefeld
University of
Applied Sciences

Speicherverwaltung selbst gemacht (Teil 2)

Ausgabe:	**05.11.2013**
Abgabe:	**KW47 (19./22.11.2013)**
Punkte:	**20+0**

Teil 2/2

Auf diesem Blatt soll eine einfache Freispeicherverwaltung wie in der Vorlesung besprochen implementiert werden.

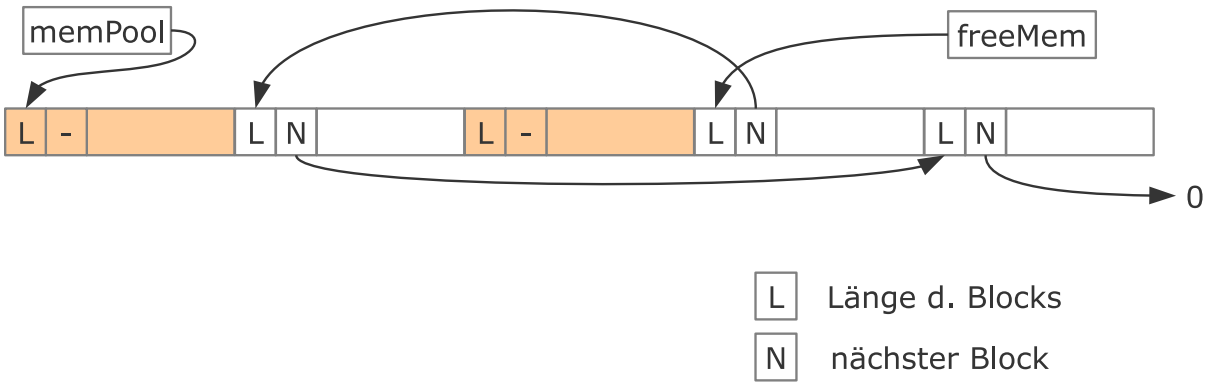
- Der Heap soll durch ein ausreichend großes Byte-Array `memPool` simuliert werden:

```
1  char memPool[MEM_POOL_SIZE];
```

- Eine Vergrößerung/Verkleinerung des simulierten Heaps während der Laufzeit des Programmes ist nicht vorgesehen.
- Die freien Blöcke sollen über eine einfach verlinkte Liste verwaltet werden. Dazu enthält jeder Block im simulierten Heap am Anfang eine Verwaltungsstruktur, die die Gesamtlänge des Blocks (in Bytes) und einen Zeiger auf den Anfang des nächsten freien Blocks enhält:

```
1  struct MemBlock {
2      size_t size;
3      struct MemBlock *next;
4  };
```

- Der `next`-Pointer des letzten Blocks ist ein Null-Pointer.
- Der Pointer `freeMem` (Typ: `struct MemBlock*`) zeigt auf den Anfang der Freispeicherliste.
- Belegte Blöcke sollen nicht in der Freispeicherliste verwaltet werden, sie behalten aber ihre Verwaltungsstruktur am Anfang des Blockes. Um belegte Blöcke extra kenntlich zu machen, soll der `next`-Pointer belegter Blöcke den „magischen“ Integerwert `0xdeaddead` enthalten.



Achten Sie darauf, *alle* von Ihnen belegten Ressourcen auf dem Heap-Speicher wieder freizugeben, sobald diese nicht mehr benötigt werden!

Schreiben Sie eine `main()`-Funktion, in der die einzelnen Funktionen aufgerufen werden (können).

1. Aufgabe: Eigenes Malloc — 8 Punkte

Implementieren Sie die Funktion `void *fbtMalloc(size_t size)`, die sich ähnlich zur `malloc`-Funktion aus der C-Standardbibliothek verhält.

Es wird mit dem *first-fit*-Verfahren nach dem ersten freien Speicherblock gesucht, der mindestens die Größe X Bytes (`size` Bytes, vgl. Funktionsparameter) hat. Im Erfolgsfall findet der Suchalgorithmus einen Block der Gesamtgröße Y Bytes, wobei gelten muss

$$Y \geq X + |\text{Verwaltungsstruktur [Bytes]}|$$

(sonst wäre der Block zu klein).

- Falls nun sogar gilt
$$Y > X + 2x|\text{Verwaltungsstruktur [Bytes]}| + 32\text{Bytes},$$
dann soll der Block gesplittet werden und der Rest als neuer Block in die Freispeicherliste eingehängt werden.
- Falls die Bedingung nicht erfüllt ist (der gefundene Block also nur „ein bisschen zu groß“ ist), dann wird der gesamte gefundene Block alloziert.

Anschliessend wird ein Pointer auf den Beginn des *Nutzdatenbereichs* des Blocks (also ein Pointer auf das erste Byte **hinter** der Verwaltungsstruktur) zurückgeliefert.

Falls kein ausreichend großer Block gefunden wird, soll ein Null-Pointer zurückgeliefert werden.

Hinweis: Vergessen Sie nicht, beim ersten Aufruf von `fbtMalloc` den simulierten Heap zu initialisieren! Rufen Sie in `fbtMalloc` die Initialisierungs-Funktion `void initHeap(void)` vom ersten Teil dieses Blattes auf.

Bemerkung für Linux-Fans: Linux verwaltet nur die Größe des freien Speichers (ohne Verwaltungsstruktur).

Ziel: Verständnis für Malloc, Pointerarithmetik, Iteration durch verkettete Liste

2. Aufgabe: Eigenes Free — 6 Punkte

Implementieren Sie die Funktion `void fbtFree(void *ptr)`, die sich ähnlich zur `free`-Funktion aus der C-Standardbibliothek verhält.

Die Funktion soll den Block wieder als freien Block markieren und ihn dazu **vorn** in die Freispeicherliste einhängen. Vor dem Freigeben ist der `next`-Pointer der Blockes zu prüfen: Er muss als Wert die „magische“ Zahl haben. Anderenfalls soll die Funktion mit einer Fehlermeldung abbrechen.

Hinweis: Durch diese einfache Art des Einhängens freier Blöcke an den Anfang der Liste wird der `freeMem`-Zeiger irgendwann nicht mehr auf den physikalisch ersten freien Block zeigen.

Ziel: Verständnis für Free, Pointerarithmetik

3. Aufgabe: Weitere Ausgaben — 3 Punkte

(a) Implementieren Sie die Funktion `void printAllocatedBlock(void *p)`, welche die Verwaltungsdaten **eines** belegten Blockes ausgeben soll.

(b) Implementieren Sie die Funktion `void printHeap(void)`, welche **alle** Blöcke des Heaps in der Reihefolge, in der sie sich im Heap befinden, ausgeben soll.

Nutzen Sie dazu jeweils die Ausgabefunktion `void printBlock(struct MemBlock *p)` vom ersten Teil dieses Blattes.

Hinweis: Beachten Sie, daß `printAllocatedBlock` als Parameter den Pointer auf den Nutzdatenbereich eines belegten Blockes erhält, aber dessen Verwaltungsstruktur ausgeben muss! Es sollen keine Nutzdaten ausgegeben werden.

Ziel: Pointerarithmetik

4. Aufgabe: Defragmentieren — 3 Punkte

Implementieren Sie die Funktion `void fbtDefragHeap(void)`, die benachbarte freie Speicherbereiche zu einem einzigen Speicherbereich zusammenfügt.

Ziel: Iteration durch verkettete Liste, Pointerarithmetik