CS2106: Operating Systems
Lab 4 – Building your own `malloc()`

Important:
- The deadline of submission on IVLE is **26th October 5pm**
- The total weightage is 6%:
  - Exercise 1: 1% [**Demo Exercise**]
  - Exercise 2: 1%
  - Exercise 3: 4%

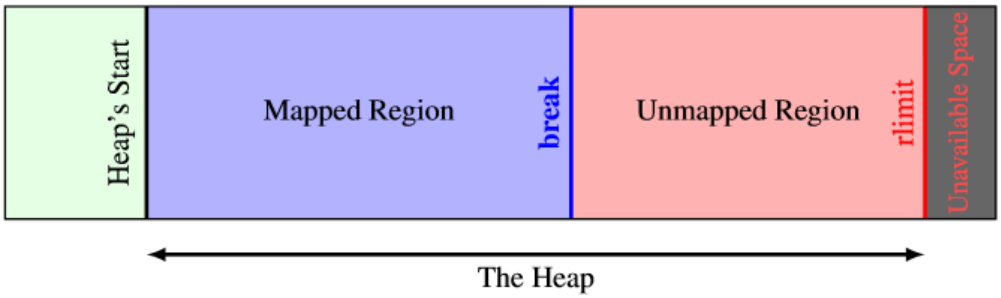
Section 1. Overview

Heap memory region is used for dynamically allocated data. In C, library calls like **`malloc()`**, **`free()`**, **`realloc()`** etc manipulate the heap region. Behind the scene, the heap region can be managed using **contiguous memory allocation scheme** as discussed in lecture 7. For this lab, we are going to implement our very own **`malloc()`** and **`free()`** functions. For "mysterious reasons" (ask Uncle Soo), we are going to call our own version ***`mymalloc()`*** and ***`myfree()`*** respectively. Don't panic! Substantial amount of code has already been written, your tasks are to understand, expand and improve the current implementation.

There are three tasks in this lab
- Exercise 1: Understand the basic implementation and print simple usage statistics.
- Exercise 2: Change the **first-fit** algorithm to **worst-fit** algorithm in **`mymalloc()`**.
- Exercise 3: Improve **`myfree()`** by automatic merging of adjacent free partitions and provide **compaction** functionality.

Section 2. Heap Region Basics

On *nix systems, the heap region is defined by three important parameters:



1. **Start:** Starting address of the heap region.

2. **Break:** The boundary of currently **usable** heap region.
3. **rLimit:** The maximum boundary of heap region, i.e. **break** can only grows up to **rLimit**. Once **break == rLimit**, we have run out of heap memory.

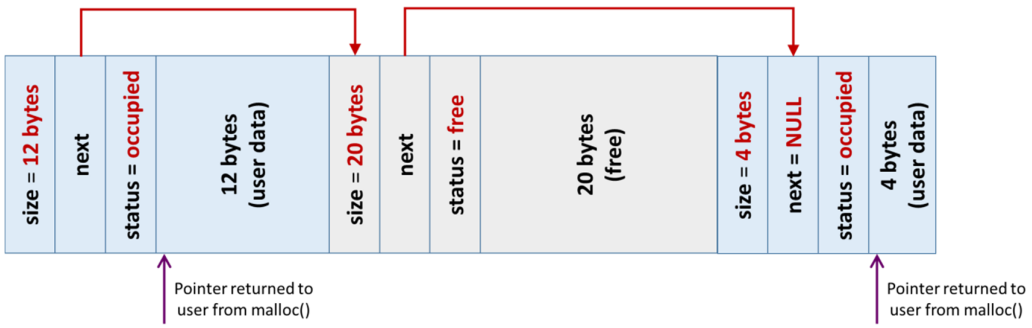
In C, we can use the "set break" **`sbrk(size)`** system call to increase the **break** boundary by ***`size`*** bytes. The **`sbrk(0)`** is a special usage which returns the **current address of the break** boundary.

For our usage, we will acquire a contiguous piece of heap region at the program start via **`setupHeap()`** function. This piece of memory is then used to fulfil all user **`mymalloc()`** requests. So, there is **no need to use `sbrk()`** for your own code.

In the dynamic allocation scheme, we try to provide partition of the **exact size** requested by the users. Instead of using a *separate* linked list to maintain the partition information, we will use an alternative design in this lab. At the head of each partition, there is a "hidden" **meta data**, which stores:

- Size of the partition
- Pointer to the next partition
- Status of the partition

Graphically, the layout of our "heap" looks like:



The above show three sample partitions: **2 occupied partitions** (i.e. allocated to user **`mymalloc()`** requests) and **1 free partition** (possibly allocated and then deallocated by user **`myfree()`** request). Essentially, you can imagine the linked list of the partition information is **embedded** in the partitions themselves. Also, note that the pointer returned by **`mymalloc()`** points to **the start of the user data portion**, i.e. **[Starting address of partition] + offset [Size of the meta data]**. The partition meta data is defined as a **`partMetaInfo`** structure in the given code.

In addition, we have a single **`heapMetaInfo`** structure **`hmi`**, which keep tracks of a) the address of the first partition; b) the total size of the whole "heap" and c) the size (in bytes) of the meta data information. (c) is frequently used in calculation, setup and teardown of partitions. The above information should help to shed some lights on the given sample code.

Section 3. Exercises in Lab 4

In lab 4, we are going to improve / modify the library calls **`mymalloc()`**, **`myfree()`** and a few utility functions. The given skeleton code has the same structure for all exercises:

1. **`main.c`**: A sample user program that utilize our **`mymalloc()`** and **`myfree()`**. It is used as a test driver for your exercises. **No need to modify**.
2. **`mmalloc.h`**: A header file for function declarations. User program should include this header file to use our malloc functionalities. **No need to modify.**
3. **`exX_mmalloc.c`**: ***X*** refers to the exercise number. An implementation of the **`mymalloc()`** functionalities and your main focus in each of the exercises.

For your convenience, we have provided a **makefile** for each exercise. You can simply type "**`make`**" to build the **`a.out`** executable.

3.1 Exercise 1 – Simple Usage Statistic [Demo Exercise]

For this exercise, calculate the following statistics in the **`printHeapStatistic`**() function:

- Number of occupied partition and the total size of occupied partition in bytes. Note that the meta data of each partition is not counted.
- Number of free partition (aka hole) and the total size of holes in bytes.
- Total size of all meta information block in bytes.

Sample Output (using **test1.in**, information in **bold** needs to be calculated):

```
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x1dd5000
Partition Meta Size = 24 bytes
Partition list:
[+    0 |    40 bytes | 1]
[+   64 |    80 bytes | 0]
[+  168 |   120 bytes | 0]
[+  312 |   160 bytes | 1]
[+  496 |   504 bytes | 0]

Heap Usage Statistics:
======================
Total Space: 1024 bytes
Total Occupied Partitions: 2
    Total Occupied Size: 200 bytes
Total Number of Holes: 3
    Total Hole Size: 704 bytes
Total Meta Information Size: 120 bytes
```

You can see that the total occupied size, total hole size and total meta information size should add up to the total size of the heap (**1024** bytes in this test).

Again, don't panic ☺. You will spend more time to understand the given code rather than coding yourself. (hint: you need about 10 lines of code for this exercise).

3.2 Exercise 2 – Worst-fit Algorithm

The given **`mymalloc()`** implements the **first-fit** algorithm, i.e. we allocate the first large enough free partition for user requests. Change the **`mymalloc()`** function in **`ex2_mmalloc.c`** so that we use **worst-fit** algorithm, i.e. the largest free partition is chosen instead.

You are not allowed to modify the existing functions and declarations, but can implement helper function(s) if needed. **There is also no need to copy your solution for ex1 over**.

Sample Output (using **test1.in**, just before the last **mymalloc(88)** request)

```
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0xa30000
Partition Meta Size = 24 bytes
Partition list:
[+    0 |    40 bytes | 1]
[+   64 |    80 bytes | 0]
[+  168 |   120 bytes | 1]
[+  312 |   160 bytes | 0]
[+  496 |   200 bytes | 1]
[+  720 |   240 bytes | 0]
[+  984 |    16 bytes | 0]
```

Sample Output (using **test1.in**, after the last **mymalloc(88)** request)

```
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0xbc6000
Partition Meta Size = 24 bytes
Partition list:
[+    0 |    40 bytes | 1]
[+   64 |    80 bytes | 0]
[+  168 |   120 bytes | 1]
[+  312 |   160 bytes | 0]
[+  496 |   200 bytes | 1]
[+  720 |    88 bytes | 1]
[+  832 |   128 bytes | 0]
[+  984 |    16 bytes | 0]
```

The row in red shows that worst fit algorithm is used because the largest free partition at offset 720, size 240 bytes was used to satisfy the request **`mymalloc(88)`**. If you need a step by step print out of the heap layout for debugging purpose, you can add a **`"-DDEBUG"`** flag during compilation.

Please note that the skeleton code for this exercise will result in a segmentation fault before your code is added as the **`mymalloc()`** functionfrom exercise 1 has been removed.

3.3 Exercise 3 – Merging and Compaction

The given implementation of **`myfree()`** does not perform merging. Even if the newly freed partition has free neighbouring partition(s), no coalescing is performed which leave them as separate free holes. One example can be seen from the sample output in exercise 2, the last two partitions are both free, but not merged.

Modify the **`myfree()`** function in **`ex3_mmalloc.c`** so that merging is performed after a partition is freed. The given test case ***test1.in*** test the following scenario:

1	2	3	4	5	6	7	8	9
Occ	Occ	Occ	Occ	Occ	Occ	Occ	Occ	Free

8 partitions (40 bytes each) were allocated **initially**. We then procced to free:

a. Partition 1: nothing special, just deallocate.
b. Partition 3: nothing special, just deallocate.
c. **Partition 4: should merge with the freed partition 3**
d. Partition 7: nothing special, just deallocate.
e. **Partition 6: should merge with the freed partition 7**
f. **Partition 2: should merge with the free partition 1 and the merged free partition 3 and 4.**

The end result is shown in the test output **test1.out**:

Sample Output (using **test1.in**)

```
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x111a000
Partition Meta Size = 24 bytes
Partition list:
[+    0 |   232 bytes | 0]   //merged free partitions 1, 2, 3 and 4
[+  256 |    40 bytes | 1]
[+  320 |   104 bytes | 0]   //merged free partitions 6 and 7
[+  448 |    40 bytes | 1]
[+  512 |   488 bytes | 0]   //the initial free partition 9
```

Finally, you need to provide **compaction** functionality via the **`compact()`** function call. Compaction moves all allocated partitions to the start of the heap region and consolidate all holes into one free partition at the end of the heap region.

In the given test case **test2.in**, we have the following layout before compaction is performed:

1	2	3	4	5	6	7	8	9
Free	Occ	Free	Occ	Free	Occ	Free	Occ	Free

After the compaction, the end result is shown in the test output:

Sample Output (using **test1.in**)

```
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x1c93000
Partition Meta Size = 24 bytes
Partition list:
[+    0 |    40 bytes | 1]
[+   64 |    40 bytes | 1]
[+  128 |    40 bytes | 1]
[+  192 |    40 bytes | 1]
[+  256 |   744 bytes | 0]
```

You can see that all four occupied partitions were moved to the start of the region and the holes were merged into a big free partition at the end. In the given code, we will perform a "compact verification" process as a check on your **`compact()`** implementation.

Several **key criteria** for a correct implementation:

- The relative ordering of all occupied partitions should be maintained after compaction.

- The **user data** of each partition needs to be copied to new location. Remember: There are actual user data in the occupied partitions! So, you need to relocate the user data instead of just changing the partition meta information. We simulate user data by placing **magic numbers** at the start and end of each of the occupied partitions (you can find out more from the **`main.c`**).

- There should be only 1 free partition after compaction. This hole should contains all remaining free space in the heap region.

Section 4. Level UP! [Just for your pondering, no need to submit]

This exercise attempts to shed lights on some "weird" behaviours you may have encountered before with the standard **`malloc()`**, **`myfree()`**. For example:

1. Why data in recently freed memory space seems intact (for a while)?

2. Why exceeding the range of allocated memory space **sometimes** does not cause segmentation fault?

3. Why dynamically allocated memory space contains "random" garbage values?

You should be able to give a pretty good guess / answer to the above questions after this lab.

Now, one last surprise. Once you have the complete code, you can try to link any code that uses dynamic memory with the **`mmalloc.c`**, e.g. linked list code (e.g. from lab 1) would be a good showcase. The linked list code should work perfectly with your very own **`mymalloc()`** and **`myfree()`**, fun eh? ☺
[Note: you need to remove the "**`<stdlib.h>`**" library, as it provides the standard malloc/free and can conflict with your own implementations.] [Note2: Of course, you need to rename all **`malloc()`** to **`mymalloc()`** and **`free()`** to **`myfree()`** in the user code to utilize your own version.]

Section 5. Submission

Zip the following files as A/E0123456.zip (**use your NUSNET user id!**):

`a. ex2_mmalloc.c` (Remember to remove all debug messages)
`b. ex3_mmalloc.c` (Remember to remove all debug messages)

Upload the zip file to the "Student Submission→Lab 4" workbin folder on IVLE. Note the deadline for the submission is **26th October, 5pm**.

Again, please ensure you follow the instructions carefully (**output format**, how to zip the files etc). **Deviations will be penalized.**

Reference:

1. "A Malloc Tutorial" by Marwan Burelle, 2009, http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf
   - Good exploration material. Some parts of the lab code were adapted from this tutorial.