CS2106: Operating Systems

Lab 4, part II – Building your own memory allocator (8 points + 5 stars)

Section 1. Overview

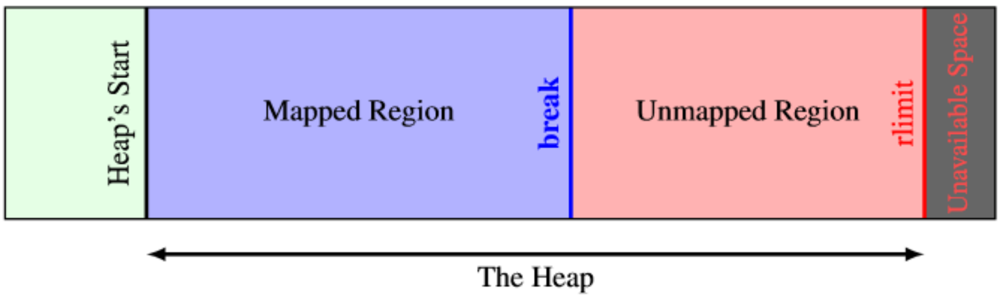
Recall that the *heap* memory region is used for dynamically allocated data. In C, library calls like **`malloc()`**, **`free()`**, **`realloc()`** etc., manipulate the heap region. Behind the scene, the heap region can be managed using **contiguous memory allocation scheme** as discussed in the lectures.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

There are five tasks remaining in this lab:

- Exercise 2: Understand the basic implementation and print simple usage statistics **[1 point]**
- Exercise 3: Change the **first-fit** algorithm to **worst-fit** algorithm in **`mymalloc()`** **[2 points]**.
- Exercise 4: Improve **`myfree()`** by automatic **merging** of adjacent free partitions and provide **compaction** functionality. **[2 points]**
- Exercise 5: Allow multiple user threads to concurrently use **`mymalloc()`** and **`myfree()`** functions in a safe manner while maximizing concurrency.**[3 points]**
- Exercise 6: Optimizing **`mymalloc()`** and **`myfree()`** for speed of allocation/deallocation. **[optional, 5 stars]**

Section 2. Heap Region Basics

On Unix 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.

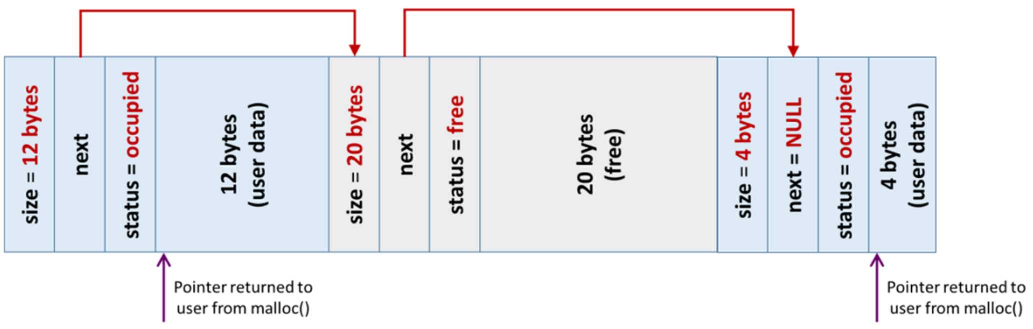
In C, we can use the "set break" **`sbrk(`*`size`*`)`** system call to increase the **break** boundary by ***`size`*** bytes. A special case of this system call is **`sbrk(0)`**, which returns the **current address of the break** boundary.

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 without involving the operating system, which is a big advantage of **mymalloc()** over conventional **malloc()**. As such, there is **no need to use sbrk()** for your own code. **setupHeap()** will set up a heap of 1024MB for exercises 2, 3, 4, and 5, and 1152MB for exercise 6.

In the dynamic allocation scheme, we try to provide partition of the **exact size** requested by the, which means we need to maintain information about variably sized partitions. 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 figure shows three sample partitions: **two occupied partitions** (i.e., allocated to the user through **mymalloc()** requests) and **one free partition** (possibly allocated and then deallocated by a **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. **ex*X*_runner.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. **ex*X*_mmalloc.c**: ***X*** refers to the exercise number. An implementation of the **mymalloc()** functionalities and your main focus in each of the exercises.

We have provided a **makefile** for each exercise. You can simply type "**make**" to build the **ex*X*** executable. Exercises 5 & 6 have an additional file, **ex*X*_extra.h**, which you can modify and submit.

3.1 Exercise 2 – Simple Usage Statistics [1 point]

For this exercise, calculate the following statistics in the **`printHeapStatistics`**() 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 (a.k.a. holes) and the aggregate size of the holes in bytes.
- Total size of all metadata information in bytes.

You should verify that the total occupied size, total hole size and total metadata information size should add up to the total size of the heap. For this exercise, you only need to change about 10 lines of code in function **`printHeapStatistics`**().

3.2 Exercise 3 – Worst-fit Algorithm [2 points]

The provided **`mymalloc()`** function implements the **first-fit** algorithm, i.e., we allocate the first large enough free partition for user requests. Your task is to change the **`mymalloc()`** function in **`ex3_mmalloc.c`** to implement the **worst-fit** algorithm, i.e., the largest free partition is chosen instead. If the allocation request cannot be satisfied, **`mymalloc()`** should return NULL.

You are not allowed to modify the existing functions and declarations, but you can implement helper function(s) if needed. **There is also no need to copy your solution for ex2 (`printHeapStatistics`()) over to ex3, because you don't need it**. If you need a step-by-step print out of the heap layout for debugging purposes, you can add a "**`-DDEBUG`**" flag during compilation (by changing the makefile), which will print out the layout of the heap after every call to **`mymalloc()`** or **`myfree()`**. I

3.3 Exercise 4 – Merging and Compaction [2 points]

The provided implementation of **`myfree()`** does not perform merging. Even if the newly freed partition has free adjacent partition(s), no coalescing is performed which leaves them as separate free holes. Your task is to modify the **`myfree()`** function in **`ex4_mmalloc.c`** so that merging is performed after a partition is freed.

Let's assume that we have a heap of 1024B and that 8 occupied partitions (40 bytes each) were allocated **initially**:

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

We then proceed to free:

a. Free partition 1: nothing special, just deallocate.
b. Free partition 3: nothing special, just deallocate.
c. **Free partition 4: should merge with the freed partition 3**
d. Free partition 7: nothing special, just deallocate.

e. **Free partition 6: should merge with the freed partition 7**
f. **Free partition 2: should merge with the free partition 1 and the merged free partition 3 and 4.**

At the end, the layout of the heap should look as follows:

```
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| 40bytes|1]

   [+  320 | 104 bytes | 0] //merged free partitions 6 and 7

   [+  448| 40bytes|1]

   [+  512 | 488 bytes | 0]  //the initial free partition 9
```

The second part of this exercise requires you to provide memory **compaction** functionality by modifying **`mymalloc()`** developed in exercise 3. Compaction moves all allocated partitions to the start of the heap region and consolidates all holes into one free partition at the end of the heap region. The compaction should be **invoked automatically** by **`mymalloc()`** when allocation fails to find a hole of an appropriate size due to external fragmentation. After the compaction process is completed, **`mymalloc()`** should proceed and fulfil the request that was impossible to fulfil prior to compaction. If even after compaction the allocation request still cannot be satisfied, **`mymalloc()`** should return NULL. Apart from **`ex4_mmalloc.c`**, no other file should be modified in this exercise.

Let's look at one example with the following layout before compaction is performed:

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

Assume partitions 1 to 8 are 40B large. The heap layout after the compaction should look as follows:

```
Heap Meta Info:
===============
Total Size = 1024 bytes
Start Address = 0x1c93000
Partition Meta Size = 24 bytes
Partition list:

[+ 0| 40bytes|1]

[+ 64| 40bytes|1]

[+ 128| 40bytes|1]

[+ 192| 40bytes|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.

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

- The compaction functionality must be implemented within the provided `compact()` skeleton.
- 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.
- There should be only 1 free partition after compaction. This hole should contains all remaining free space in the heap region.

3.2 Exercise 5 – Thread-safe `mymalloc()/myfree()` [3 points]

In this exercise, your task is to extend the code developed in exercise 4 to allow for multiple threads to concurrently call **`mymalloc()`** and **`myfree()`** functions, while ensuring correctness and avoiding any data races. For this exercise, the main function (runner) will be run with a high number of threads that simultaneously allocate and free memory using **`mymalloc()`** and **`myfree()`**. You should use your solution developed in Ex4 as a single-threaded baseline (if you use your Ex3 solution as a baseline, you can get maximum 1 point for this exercise), and provide the necessary synchronization support (of your choice) to allow for thread-safe allocation/deallocation. Your solutions will be graded based on correctness and the average speed of allocation achieved. You can assume that the maximum number of threads concurrently calling **`mymalloc()/myfree()`** will be **16**.

For this exercise, you are allowed to modify only **`ex5_mmalloc.c`** and **`ex5_extra.h`**, which is a header file provided for your convenience.

3.2 Exercise 6 – Free-style `mymalloc()/myfree()` [optional: 5 stars]

In contrast to previous exercises, in this exercise you are not required to use the provided linked-list implementation of heap management. You have the full freedom to explore various techniques of your choice that can lead to faster allocation. Therefore, you are allowed to use additional external data structures to help you manage the heap. You will notice that **`setupHeap()`** will set up a heap of 1024MB for exercises 2, 3, 4, and 5, and 1152MB for this exercise. You are allowed to store data allocated through **`mymalloc()`** only within the first 1024MB! The rest of the heap space is for your auxiliary data structures, if any.

In this exercise, your task is to implement **`mymalloc()/myfree()`** functions to:

1. Maximize the average allocation speed. The average allocation speed will be the main metric for evaluation. Your average allocation speed should be significantly higher compared to the speed you achieve in Ex5.
2. External fragmentatiown observed by the user should be minimal. In exercise 4 the allocation should always succeed if there is enough aggregate free space in the system to satisfy an allocation request. A reasonable relaxation of this requirement is tolerated if it leads to significantly higher allocation speed. This means that **`mymalloc(request_size)`** is allowed to return NULL if:
    a. aggregate size of the free memory is less than 10x `request_size`, or
    b. if the aggregate free memory size is less than 10% of the total heap size.
3. Ensure freedom from race conditions in **`mymalloc()/myfree()`** for multi-threaded programs, as in exercise 5.

You are allowed to modify only **`ex6_mmalloc.c`** and **`ex6_extra.h`**, which is a header file provided for your convenience. Note the following:

a) Your code will be tested in a multi-threaded setup.
b) **`mymalloc()/myfree()`** are allowed to be multi-threaded, if you need them to be.
c) You can assume that there will never be more than **1 million** occupied partitions in your heap.
d) You can also assume that the minimum allocation size will be **4B**, and maximum **4MB**, and that the allocation size approximately follows the Poisson's distribution with a mean of **1KB**.
e) You can assume that the maximum number of threads calling **`mymalloc()/myfree()`** will be **16**.
f) Your code is not allowed to use conventional `malloc()` and `free()`.

Section 4. Submission Instructions

The following files should be submitted to the appropriate LumiNUS folder:

- `ex2_mmalloc.c`
- `ex3_mmalloc.c`
- `ex4_mmalloc.c`
- `ex5_mmalloc.c` and `ex5_extra.h`
- `ex6_mmalloc.c` and `ex6_extra.h`

Please compress all the files (without any folder structures) into one file named `E0123456A.zip` (<your NUSNET id>.zip) and submit it through LumiNUS (Files→Lab Assignments → **lab4-rest-submissions**). The submission deadline is **April 11, 8pm**.