



CSCI-UA.0201 – 007 Computer Systems Organization FALL 2022

Assignment 5

Due date: 12.17.2022, 11.59pm

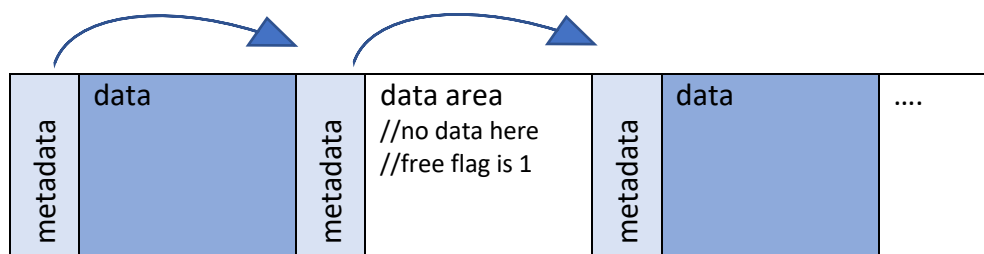
Late submissions due date: 12.20.2022, 11.59pm

This is an individual work. No team work is allowed.

Your Own Dynamic Memory Allocator and Deallocator

In this project, you will write your own dynamic memory allocator in C. Please check the given source code carefully and fill in the `//TO DO` areas to run your program. This project will give you an idea in an abstract level about how memory is dynamically allocated and deallocated in our computer system. We can start by checking our heap structure.

heap:



data area to be pointed and
returned by malloc

What we need for the metadata:

A block that contains:

- a pointer to the next block metadata area,
- a flag to show if the block is free or not,
- data size of the block.

This part is shared with you in the helper source code. In this code, you will also find that memory for 8000 bytes is declared and will be initialized by you. Then, we will allocate and deallocate our chunks inside it.

```
char mymemory[8000];

struct block
{
    size_t size;
    int free;
    struct block *next;
};
typedef struct block block;
```

Initially, we will point to the main block of memory which is completely free:

```
block *list=(void*)mymemory;
```

Here, we use void type casting in order to use different data types easily.

Now, you are asked to write five functions in your implementation:

- ◆ initMemory
- ◆ NewMalloc
- ◆ NewFree
- ◆ split
- ◆ coalesce

Let's take a closer look at these special functions:

1-initMemory

In this function, the size of the list should be defined as:

the total memory size – size of your metadata block

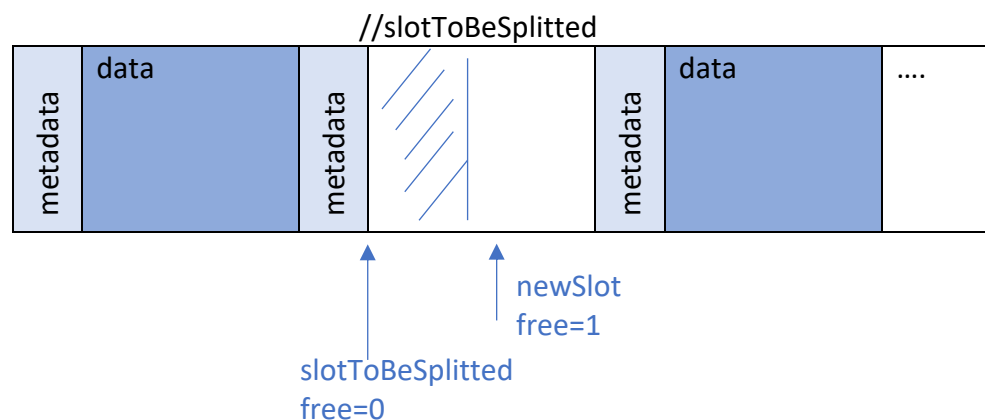
List should be initialized as free and the next pointer should point to NULL.

2-split

This function splits the memory after allocation. As function arguments, we send a pointer to the metadata block of the chunk we will use (let me name it as **slotToBeSplitted**) and the required size to be allocated. Now, we need a new block (**newslot** in our code). The size of the **newslot** can be computed as:

$\text{slotToBeSplitted} \rightarrow \text{size} - \text{size} - \text{sizeof}(\text{struct metadata block})$

Here, do not forget to properly assign next pointers as newslot's next pointer is now slotToBeSplitted's next pointer and slotToBeSplitted's next pointer is now the newslot itself.



3-NewMalloc

This function is our malloc function. Here is a simple case to understand the idea better: Assume that we get a request to allocate a block of memory of size 1000 bytes. Starting from the first metadata block, we look for the first block which has enough space to allocate 1000 bytes (First-Fit approach). Now, as we traverse, we can encounter many free blocks with size, e.g. 600, 300, 500, 1200, etc.. We will pass the first three of them and allocate in the block of size 1200.

If we can find a free block which exactly fits the required size, we don't need splitting. Otherwise, we need to call the split method.

This method should start by pointing to the start of our metadata block list and you should check whether the pointed chunk has enough space for allocation. Initially, you will have one block. After each splitting, you will have additional blocks. If you have an exact match (size you want to allocate == chunk size), you do not need to split. If the size of the chunk size is small, you need to check for the next available slot. If you find a large enough chunk, stop there and you have to call the split method. If there is no enough memory, print an error message. After a successful allocation, return a pointer to the allocated slot.

4-coalesce

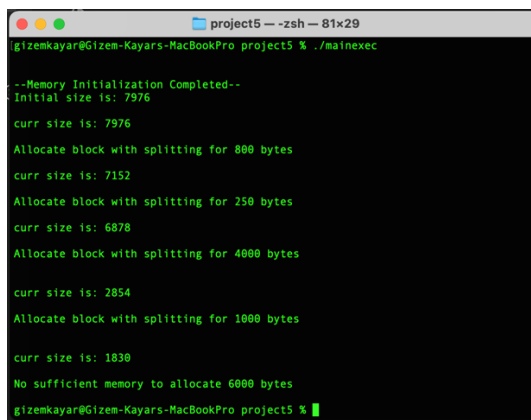
Remember the class slides. After the deallocation, we have consecutive blocks that are set free and it causes extra fragmentation. Chances are high that our NewMalloc() function can return a NULL pointer although we have enough memory to allocate due to this unorganized fragments. Therefore, we need to implement coalescing to combine these consecutive free blocks by removing the metadata blocks in between them.

5-NewFree

This is the part where we deallocate the unused memory chunks. Our function will take the pointer to a block of memory previously allocated. First of all, we need to check whether the given pointer is pointing to a valid location. If yes, set the free flag as 1 and search for consecutive free blocks to coalesce.

main() function is shared with you. Please check the given output and be sure that your output is consistent with the given one. Do not forget that we will also try other sizes for your full memory chunk and requested block sizes.

Output for the shared main function:



```
project5 -- -zsh -- 81x29
gizemkayar@Gizem-Kayars-MacBookPro project5 % ./mainexec
--Memory Initialization Completed--
Initial size is: 7976
curr size is: 7976
Allocate block with splitting for 800 bytes
curr size is: 7152
Allocate block with splitting for 250 bytes
curr size is: 6878
Allocate block with splitting for 4000 bytes
curr size is: 2854
Allocate block with splitting for 1000 bytes
curr size is: 1830
No sufficient memory to allocate 6000 bytes
gizemkayar@Gizem-Kayars-MacBookPro project5 %
```

How to submit: submit your source code: myallocator.c. Do not submit any other project file!!