I - INTRODUCTION

In this project, we will write a C library which implements a better version of the system's dynamic memory functions malloc() and free(). Dealing with dynamic memory is cumbersome because the system only outputs segment faults without further explanation when users make mistake. It can create lots of frustration for users since they have no insight about what is wrong with their applications. The new malloc() and free() functions will not only handle the essential functionality of the system malloc library, but will also detect common programming and usage errors that may occur when working with dynamic memory.

II - WHAT IS MALLOC AND FREE ?

Malloc is a C function which allows users to allocate a block of memory from the system memory heap. The data inside this block will persist until the block is freed by users. Malloc returns a void pointer that points to the first address of a block of memory of at least the requested size.

The free function allows users to free a dynamically-allocated memory block, which is created by the malloc function. Freeing malloced memory before terminating the application is crucial to avoid memory leaks from occurring. When there are enough memory leaks, the application can consume more memory than is physically available and can cause programs to crash.

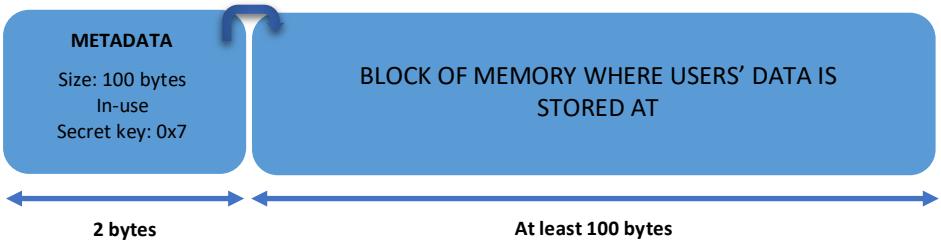
IV - MYMALLOC IMPLEMENTATION

First of all, we need to create a virtual memory, which is a static array of size 4096 bytes to simulate the heap (assuming that the heap size is 4096 bytes). The mymalloc function performs similarly to the system malloc(). Indeed, it returns a void pointer which holds the address of the first byte of a block of memory that has as least as many bytes as requested; and

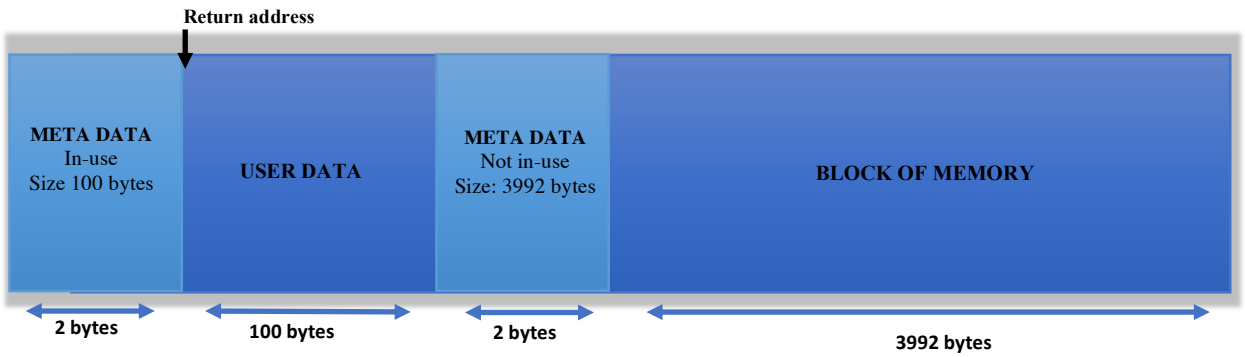
returns NULL if there is not enough space available or there is no such block of memory in the virtual memory.

To implement mymalloc, we need to keep track of how much and what other memory users have allocated before. We use a structure called metadata that stores useful information when mymalloc is called. Please read section VI to learn more about how metadata works.

Generally, we have a metadata to store information of a block of memory, which is used to store user data. For example, when users want to allocate a memory of size 100 bytes using mymalloc, the metadata and the memory block would look like this:

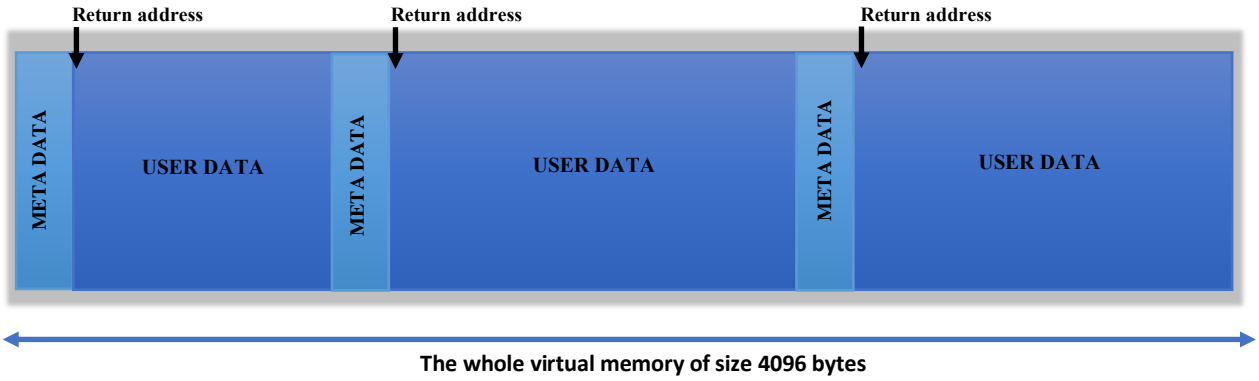


For the first call of mymalloc, the virtual memory only consists of a single metadata and a single memory block of size 4094 bytes, the other 2 bytes is consumed by the metadata. Assuming the user ask for 100 bytes of memory, since we have 4094 bytes available, the memory block of size 4094 bytes will be spliced into 2 blocks, one of size 100 bytes, and one of size 3994 bytes, because we do not need to give the user 4094 bytes block of memory to store 100 bytes of data. Then, the first metadata is now updated to in-use. Since we need a metadata to store information of the second block of memory, another metadata is created. Now, the second memory block has size of 3992 bytes because its metadata consumes 2 bytes.



The address at the first byte of a user data block is the return value of mymalloc.

Because we can store multiple memory blocks of different sizes on the virtual memory, the virtual memory would look like a flattened linked list, whose nodes are all separated metadata and blocks of memory:



In addition, we keep track of the first and last metadata to check for out of bounds when traversing the virtual memory. Since we implement First Fit algorithm, we also want keep track of the first not-in-use memory block to minimize the time traversing the virtual memory when calling mymalloc.

V - MYFREE IMPLEMENTATION

A call of myfree function will first check for all possible errors before it frees a block of memory. If an error occurs, myfree will print out an informative message then return.

The first and second errors to check for are NULL input and out of bounds input. Then myfree will check the secrete key of the block of memory's metadata, as explained in the VI section.

After clarifying that the block of memory in front of the input address may be a metadata. We will traverse through the virtual memory to find a match to the address of the potential metadata. If there is a match, it is assured that the block of memory in front of our input address is a metadata. We then check if that metadata is in-use or not. If it is in-use, we are ready to free the input block of memory.

The freeing process first sets in-use to 0. Then we need to decide on whether we can combine the current block of memory with its adjacent blocks of memory. The purpose of joining adjacent blocks of memory together is to optimize the amount of space for future mymalloc calls. When combining two blocks of memory, the right block of memory will be merged with the left block of memory. The right block of memory's metadata will be zeroed out, and the size of the left block of memory is updated to the sum of the size of both the left and right blocks of memory plus the metadata size.

VI - METADATA EXPLANATION

Metadata is a structure that stores useful information about its following block of memory on the virtual memory. By reading information off a metadata, we can learn about the size of the following memory block and whether that memory block is in-use or not. Metadata can also be used to determine whether an address is a metadata address. Despite its significant aplication, the metadata structure only contains a single element of type unsigned short named data:

```
struct metadata
{
        unsigned short data;
}
```

Unsigned short takes 2 bytes, equivalent to 16 bits. We use the first bit of the data of a metadata to determine whether a memory block is in-use or not in-use. If a memory block is in-use, we set the first bit to 1, otherwise the default is 0.

The next three bits contain a secrete key for fast checking a metadata address. This secrete key is set to 7 (0b111) whenever a metadata is created, and zeroed out when a metadata is freed. When myfree is called, we first obtain the metadata of the block of memory requested to free. Then we abstract the next three bits after the first bit of data, if this number equals to 7, the current address may be a metadata address. However, if this number is not 7, we immediately

know that the current address is not a metadata address, so that the input block of memory is also not dynamically allocated.

Finally, the last 12 bits of data stores the size of a block of memory, the maximum size is 1111 1111 1111 (0xFFF), which infers 4095 bytes. It is sufficient enough to store the largest size possible when call mymalloc, which is 4094 bytes, since the first metadata consumes 2 bytes out of 4096 bytes.