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 output the SEGMENTATION FAULT error without further explanation when users make mistake. It can create lots of frustrations 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 and friendly resolve common programming and usage errors occur when working with dynamic memory.

II - WHAT IS MALLOC AND FREE ?

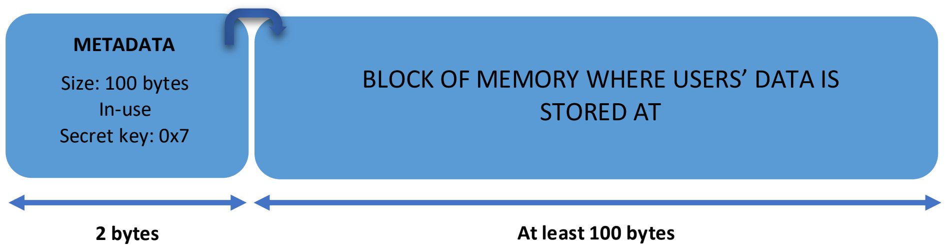
Malloc is a C function which allows users to allocate a block of memory from the system memory heap. The data inside this block will persist until the block is freed by users. Malloc returns a void pointer that points to the first address of a block of memory of at least the requested size.

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 to occur. When there are enough memory leaks, the application can consume more memory than is physically available and can cause programs to crash.

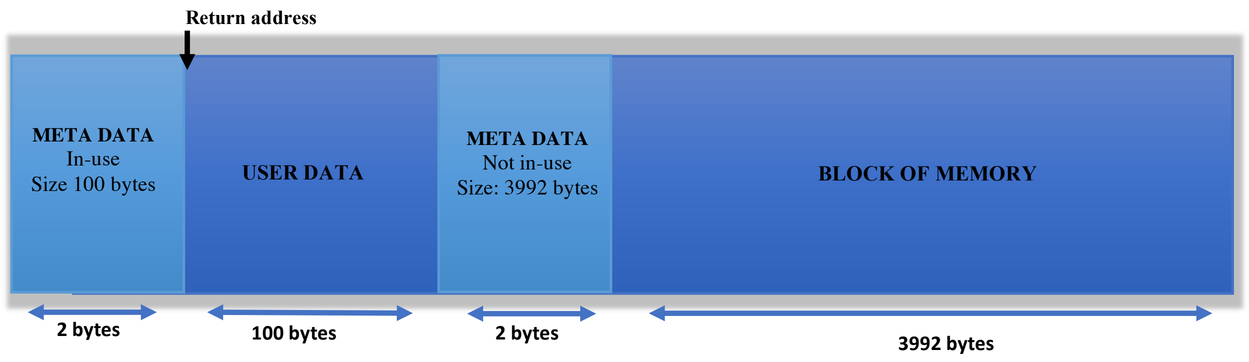
IV - MYMALLOC IMPLEMENTATION

For the purpose of this project, we need to create a virtual memory, which is a static array of size 4096 bytes to simulate the heap. The mymalloc function performs similarly to malloc(). Indeed, it returns a pointer (to an address from the virtual memory) to at least as many bytes as requested, and returns NULL if there is not enough space available on the virtual memory. To implement it, mymalloc need to keep track of how much and what other memory users have allocated before. To keep track of these information, we need to implement a metadata that stores important information when mymalloc is called. Please read section VI to learn more about how we structure a metadata.

In short, we have a metadata to store information of its following block of memory, follows right after the metadata is the actual block of memory where users' data is stored. A memory block must have the size as least as big as the memory requested by users. 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:

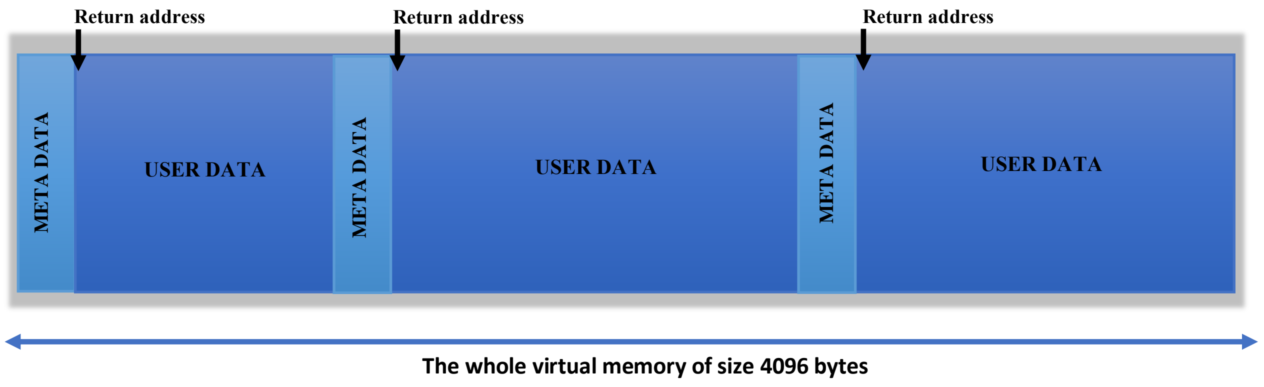


For the first mymalloc call, the virtual memory only consists of a single metadata and a single memory block of size 4094 bytes, the other 2 bytes is consumed by the metadata. Assuming the user ask for 100 bytes of memory, since we have 4094 bytes available, the memory block of size 4094 bytes will be spliced into 2 blocks, one of size 100 bytes, and one of size 3994 bytes. The first metadata is now updated to in-use. Since we need a metadata in front of every memory block, the second metadata is created. The second memory block has 3992 bytes available now 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:



V - MYFREE IMPLEMENTATION

The myfree function will fist clarify that the input address is valid. Valid as in if it was allocated by mymalloc within the virtual memory and has not been freed yet. If the input address is valid, the metadata of that memory block will be updated to not in-use, indicating that the memory block is available for other uses. Then myfree will look for the two adjacent blocks of memory, if either or both of them is/are also not in-use, myfree will merge these blocks of memory and update the new status of the consolidated block of memory.

VI - METADATA EXPLANATION

Metadata is a structure that stores useful information about its following memory block 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 we can free an address. Despite its significant role, the metadata structure only contains a single element – data:

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

The size of unsigned short is 2 bytes, equivalent to 16 bits. We use the first bit of the data element to determine whether a memory block is in-use or not. If a memory block is in-use, we set the first bit to 1, otherwise its default is always 0.

The next three bits contain a secrete key that determines whether an address is a valid address to free. We already know that when users call mymalloc, it returns a pointer to an address where the memory block start. To free that memory block, we need to go to the same address and perform the myfree function. But how do we know the address where we are at is one of the address returned by mymalloc? The secrete number help us overcome this problem. We set the secrete key to 0x7 to indicate that the current memory block is created with mymalloc, if the secrete number found in an address that does not match 0x7, we know that this address has never been returned by mymalloc and it cannot be free.

Finally, the last 12 bits of data stores the size of the requested memory block, it can be as big as 1111 1111 1111 (0xFFF), which infers 4095 bytes. It is sufficient enough to store the largest size of a requested memory block, which is 4094 bytes, since the first metadata consumes 2 bytes out of 4096 bytes.

Overall, the total size of metadata is 2 bytes.