Report for project2

Overall design

my\_memory.c can be divided into three parts:

1. queue related operations e.g. insert the address into the queue.
2. buddy allocation, including malloc and free
3. slab allocation, including malloc and free

The biggest work to do for the buddy allocation is to decide which chunk of memory should be sliced for a specific level of chunks to be used. Here we use a recursive method to do it. Although the recursive functions have some extra calling overhead, they have clearer concept. The idea is basically if there is no memory of the required level, then go to its higher level. If the higher level also does not have enough memory, still goes to much higher level until the top level (corresponds to 1MiB chunk). Then the recursive functions return, from higher level to the required level.

The entire my\_memory.c is written in a top down process. We firstly write the skeleton of my\_malloc and my\_free. When confronted with detailed operations e.g. insert an address into the free list, just leave an empty function call there. After all these have been finished, then move on to implement the subfunctions. During the implementation of the subfunctions, the exact behavior and details are interactively adjusted.

The free holes used to represent different level of chunks are implemented by an ordered linked list. These are several separate functions for this: get\_head, insert\_elem, remove\_elem, pop, find\_prev and find\_next.

Given the descriptions of the project2, there are actually not many decisions need to be chosen from.

Challenges

There are a variety of headers within the codes and each header should be stuffed with some kinds of byte value. It really took us a long time to debug to let it work correctly, or at least seemingly correct. The most common bugs for this are inconsistency header value, store a value A and retrieve but expect it to be value B.

Another challenge is that there are so many output lines for some test cases. Moreover, these lines are much dependent on each other. One mismatch in a specific line, will probably makes many subsequent lines all to be mismatched. This then makes it difficult to let the debugger stop the program at the input that corresponds to the mismatched output line.

As is similar in project1, the existed test case does not cover much the source code. And the test cases are hard to construct. We only construct another test case (pasted as below) to test this scenario: For slab allocation, when a total of 64 chunks have been freed, and these 64 chunks all belong to a middle slab, then this entire slab’s memory need to be freed. This needs the line 4 ~ line 67 in test\_7.



Trade-offs

We have used a dynamic and static combined method for allocation the slab’s descriptor table.

1. Initially allocate a slab descriptor table with 128 entry. Then use it as a large array.
2. If with the program progresses, there is no free entry within the table, then a double sized array is dynamically allocated and the content is copied from the old table to the new table.

By using this way, much easier static array like access patterns can be used.

Besides, although in input\_6 there are two lines” C 34 F 0” to test double free issue, this can be implemented in the codes to judge the header value, whether it is an invalid size value and leave it to be an invalid size value after each free. We tried to further extend the test case to be something call my\_free() with some invalid address input e.g. 0x12. Of course, the program crashed, since the address is too small to be a valid address. But we don’t find an easy way to avoid this, since the former judging the header method still needs to dereference the pointer, which crashes the program. Therefor, we give up judging this kind of invalid address.

partition of work between team members

Xiangyu Ren: the buddy allocator and queue related operations

Zihang Xu: the slab allocator and test

any specifics/quirks

None.