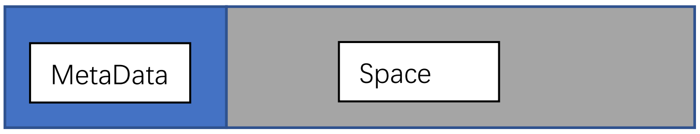
1. Implementation

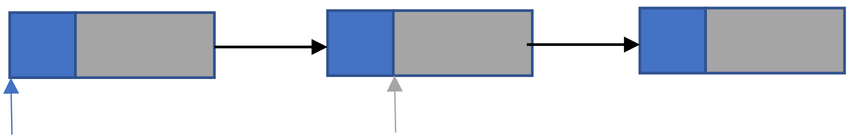
1.1. High Level

In the high level design, I use a singly LinkedList to implement the FreeList and Use sbrk() to increase the heap size. In the FreeList Node, I name it as node_t, which is composed of a MetaData and the useable Space.



1.2. FreeList

In the FreeList, the List will be sorted by the blue pointer, which replace the address in the heap. And the grey pointer is what we will return to the user.



When I want to free a block, I firstly, add it to the very front of my Free List, And Iterate over the whole List to find a suitable position. Once I find the position, I will return the pointer to the former node. Then I will check if I can merge the newly added node with the former one or the next one.

1.3. Malloc_lock & Free_Lock

For these two functions. The strategy is easier to come up with. Actually, we can just lock the Malloc process and Free process to finish the task. In my realization, I just reuse the work of my last homework and add a clock to realize the Malloc and Free function (lock version). And the process is not very complex.

1.4. Malloc unlock & Free_unlock

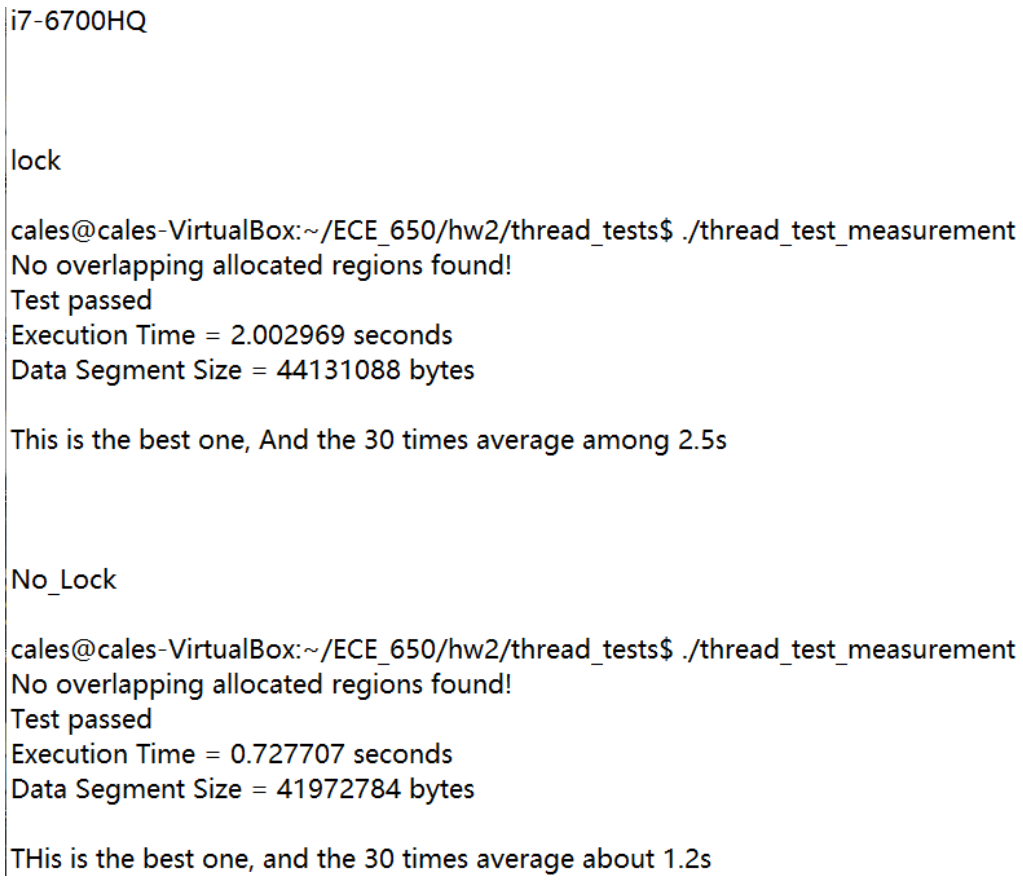
For these two functions, the strategy is a little complext. As we need to have an unlock version of Malloc and Free Function.

For the Malloc Function, we need to consider the process. In fact, it have two process. One is just calling sbrk() and the other will use The Block in the Free_List. So the first thing I would like to do is to add a lock to the sbrk() function so that the first process will be thread safe. And for Using the Block in the Free_List, it will be a little complex. So We can just move to discussion about the Free_Function.

For the Free Function, what we need to do is separate the process related to the Free_List, that means every time we will not manipulate on the same list. So we need to use TLS technology to make different process have different FreeList to access. So, in malloc.h I add two lines code to realize this.

2. **Result**

   The Test Result is just like the following Figure. (you can find the Res.txt in the GitHub Repository).
   As the figure has displayed. you can observe that the lock_version is faster than no_lock version and the range between them is a little big. In my opinion, I believe this difference arise from that no_lock version will have less code be locked. actually. all of the lock version code will be locked, which means that different isolated thread will run respectively. in fact, it will equal to running different thread in sequence. So, we need to improve this process by making the code more isolatedly.
   So we will have a better choice，which is no_lock version。No_lock version doesn't means that there is no lock. In fact, it means we only lock the required sbrk() function and let different thread have different free list head. After applying this, I have saw that the speed is faster than before. And I believe this arise from that the code can be more parallel than before, which save a lot of time.



i7-6700HQ

lock

```
cales@cales-VirtualBox:~/ECE_650/hw2/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 2.002969 seconds
Data Segment Size = 44131088 bytes
```

This is the best one, And the 30 times average among 2.5s

No_Lock

```
cales@cales-VirtualBox:~/ECE_650/hw2/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.727707 seconds
Data Segment Size = 41972784 bytes
```

THis is the best one, and the 30 times average about 1.2s

And the Test Environment is i7-6700HQ, in the Virtual Box virtual machine. The Operating System is Ubuntu 20.04, using gcc, make to compile the code. The editor is Emacs, which has the same configure with ECE 651 VM.