1.Implementation

Data Structure:

Text

Description automatically generated with medium confidence

Use a doubly linked list to store those free blocks. All these blocks are defined as a data structure called meta data which has the attributes: the memory it occupies, if this memory is free to use, and the pointer to the previous meta data, the pointer to the post meta data.

Details:

As to allocate a memory space, we should first consider whether there is a free space meeting the requirement. So, I try to find a block (meta data) through the free block list(the doubly linked list used to store those blocks which have been freed). Then, if there is a block found, we can use that block to allocate memory for the target. Otherwise, we should expand the heap to allocate new memory.

Text

Description automatically generated

As to find the appropriate block, I use an int parameter first\_fit to decide whether to return the first appropriate block or the best block. It is just a flag to choose a policy. The implementation is quite eazy. Always, I use the fhead and ftail which is defined as the head and the tail of the free list to main the list.

Text

Description automatically generated

Text

Description automatically generated

As to split the block, I split the block into two blocks. I create a new meta data at the corresponding position (the second part). The new created block has the size which equals to the original block size minus target size. Then the first split part is used for storage and is not free anymore. So, I remove it from the free list. Next, I check if the new block (second part) size is bigger than the meta data struct size. This step is to see whether the left part can at least store a meta data struct. If so, we set the new created block and add it to the free list.

Text

Description automatically generated

As to allocate new memory, I create a meta data at the current break position using the function sbrk(0). Then update meta.

Text

Description automatically generated

As to free a block, first is to find the head meta data position. Then set the attribute isfree to 1 and add the block to the free list. In the end, to see if we can merge the block according to the position at the list. Merge Function is not complex, just check the previous block and the post block to see whether the prev block address + sizeof(meta) + prev block size is equal to current block address.

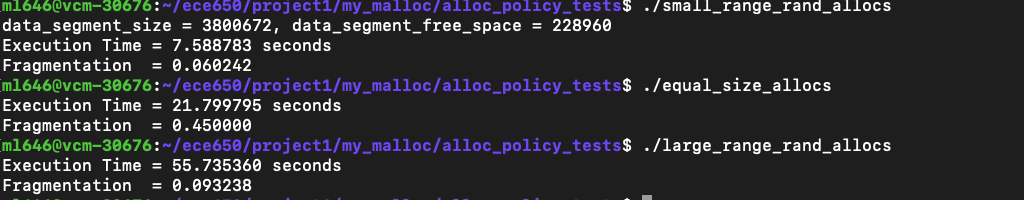
Text

Description automatically generatedText

Description automatically generated

2.Results

FF:



BF:

A screenshot of a computer

Description automatically generated with medium confidence

The results are interesting. In common sense, from the logic of the two policies, first fit policy tends to have shorter execution time while best fit policy tends to have better memory usage (lower fragmentation). However, the result of small\_range\_rand\_allocs is not as expected. BF has lower fragmentaion, which is reasonable, but takes shorter to execute. I think this is because in this case the required memory size varies in a small range. Imagine three free blocks have the size 5, 3, 7, respectively. With BF policy, when we require the size 3, 5, 7, it takes all the three buckets. However, with FF policy, it takes 5 and 7 and must require a piece of new memory. This may account for the result that BF runs faster than FF.

For the large\_range\_rand\_allocs and equal\_size\_allocs, BF takes longer to run while has lower or same fragmentaion, which is what we expected.

Generally, I would recommend first fit policy because the large\_range\_rand\_allocs setting is the closest to real cases. Most of the time the required sizes vary in a great scope. Consequently, trying to traverse and find a best fit block is difficult and meaningless. If we know ahead that the required sizes are similar, I will recommend best fit policy instead according to the experiment results.