Report of HW1 (malloc)

Name: Tongbo Liu Net id : tl259

1. Overview of the implementation

1.1 The implementation of malloc & free

First, there're few steps to realize the malloc:

1. Call the system call *sbrk()* (**void sbrk(intptr_t increment);*) to malloc a specific size of space.
2. Call the function of *memset()* to initialize the space to free a specific size of space.

There're few challenges during these steps above:

1. During malloc, when the free space is big enough, we need to split into two spaces, one is used for malloc, the other one will be still free.
2. During free, when the free space is next to each other, we need to merge the two spaces into one whole free space.
3. There're two ways to find the free space when mallocing:

   (1) Best Fit: find the best free space for allocating. In my implementation, I use **two ways** to realize.

   (2) First Fit: find the first free space available for allocating.

Considering the frist two challenges, I create a struct which can be best used for my implementation.

```
// create a meta information of a free block
typedef struct Meta_Free {
  size_t size;
  // for creating a list of blocks (random order)
  struct Meta_Free *next;
  struct Meta_Free *prev;
  // set the address of the neighbor free block (physically consecutive)
  struct Meta_Free *next_add;
  struct Meta_Free *prev_add;
  // set if it's free
  int is_free;
} meta_free;
```

The struct is called Meta_free, which means that the meta can show us the detail of the free block. The *size* is the size of the space which is available for allocating data. The *is_free* will be set 1 if the block is free and is n't allocated data. And there're two couples of points:

1. The first couple of pointer is *next* and *prev*, *next* is pointed to the next free space, and *prev* is pointed to the previous free space. So now we can see that, through implementing the pointers, the free space can be connected via the pointers and become a double direction list. The purpose of this is to let the process of finding free space in both FF and BF easier. We can just traverse the list and find the appropriate space to allocate the data. The highlight of my implementation is that the space free is **not** listed in a specific order. The list is only used for finding fitted space when executing malloc. So when we free a allocated space, we just change the meta data of the space and add it in the front or the tail or the list. (In my implementation I use the front). The couple of the pointers is also used for splitting. I set a threshold to decide when to split the block into two.

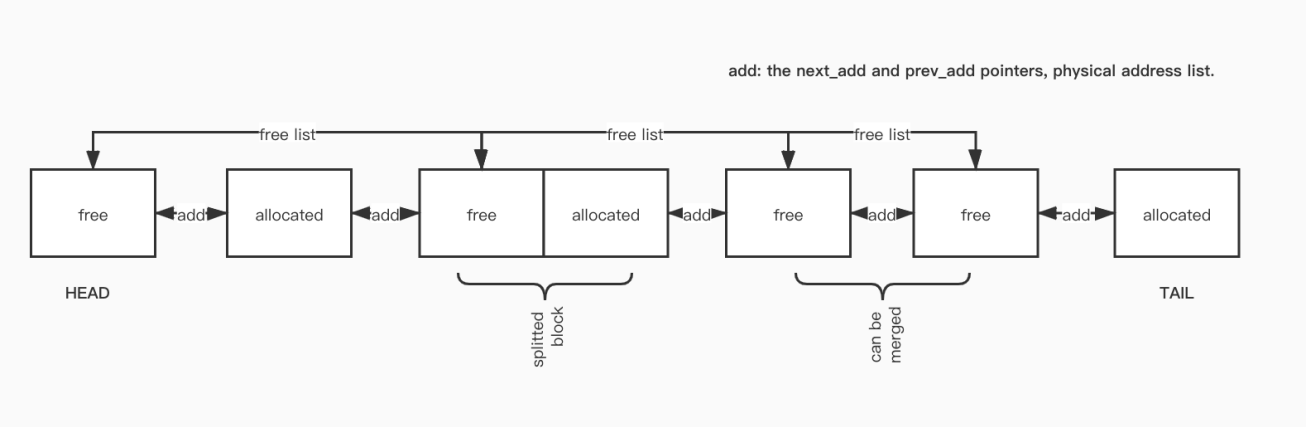
   ```
   #ifndef split_threshold
   #define split_threshold sizeof(meta_free)
   #endif
   if (meta->size <= malloc.size + split_threshold) {
     // no need to split
   }
   else {
     // split
   }
   ```

   I set the threshold be the size of the struct since if the remain free space is bigger than the size of meta, we can split the space into two spaces, and the other space can both have the space to allocated data and set meta data. When executing the split operation, the frist thing need to be considered is to choose which space to allocate. In my implementation, I choose the latter one to allocate data and the ahead to let it still be free. Becuase by choosing in this way, we will only need to change the meta data (size) of the original free space and new a meta data of the allocated space, we will **not** need to change free list. This way is beneficial for coding easier and speed up the execution time.

2. The second couple of pointer is used for merging two free space into one whole space. The *next_add* is the address of the meta data of next space of cuurent space. The *prev_add* is the address of the meta data of previous space of cuurent space. During the implementation of this couple of pointers, we don't need to consider if the space is freed. All the blocks of space is connected via this pointer no matter what kind of the block is. So when executing the operation of merge, through the pointers, we can find the blocks next to or previous to the current block **on the disk**, which means that the block is physically next or previous to the current block. Then we can check the meta data of the two blocks do decide if it can be merged. There are two conditions, see the code below.

```
    // check the next block can be merged
    // 1. exist 2. is free
    if (free_blk->next_add && free_blk->next_add->is_free == 1){
        // the next block exists and it is free.
    }
    // check the previous block can be merged
    if (free_blk->prev_add && free_blk->prev_add->is_free == 1){
        // the previous block exists and it is free.
    }
```

The following figure is a overall implementation of the struct I use.



The head means the head of the free list, and the tail means the tail of all the blocks (***sbrk(0)***).

1.2 The implementation of Best Fit & First Fit

1.2.1 First Fit

First Fit means that when executing malloc, we will traversing the free list and find the first block that the size of which is bigger than the size we need plus the size of meta data. Then we can check if it exists and if it can be splitted. Then remove the allocated block from the free list.

```
// find first match free block
    meta_free *find_res = head;
    while (find_res != NULL){
        if (find_res->size >= size){
            break;
        }
        find_res = find_res->next;
    }
```

We can see that the FF is easier.

1.2.2 Best Fit

Best Fit means find the best suitable block for the malloc. The ***suitable*** can be explained by many ways, In my implementation I used two ways to implement the BF.

1. First way is that I defined *suitable* as the smallest block which can be chose as allocated block. This means that we need to traverse all along the free list and find the smallest one. When I test on the large range allocated datas, the execution time of the program is very slow. So I think the solution is too slow when facing big amount of malloc demand. But the fragmentation is smaller. So if it's worthy of sacrifice the speed for the better disk distribution.
2. Second way is that I defined a threshold, if I find a block whose size is smaller than the threshold and bigger than the size of malloc demand, I'll just choose the block and break out of the while loop. This solution works on all the test datasets. So the process of finding the best fit block will define if the program can run within limited time.
3. Thrid way is to combine the 1st and 2nd solution, each time we'll check the size of the block, at the same time save the smallest suitable block. This solution is also worked. But I think that unless the threshold is small enough, or each time we'll just break out of the loop. And the size of the block we choose is just the threshold size.

```
while (find != NULL){
        // speed up the match
        // since sizeof(meta)=48
        // make a threshold = split_threshold//3 = 16
        //slotution 2
        if (find->size >= size && find->size <= size + 16) {
            find_res = find;
            // find a better one, sacrifice the optimal for the speed.
            break;
        }

        // solution 1.
        if(find->size > size + split_threshold){
            if (find_res == NULL || find->size < find_res->size){
                find_res = find;
            }
        }
        //
        find = find->next;
    }
```

2 Result and Performance Analysis

2.1 First Fit

Range of malloc	Run time (seconds)	Fragmentation
Small	1.300549	0.080939
Equal	0.895861	0.450000
Large	6.387061	0.114686

We can see that the runtime is fast and the fragmentation is small. Comparing small test data to big test data, both of runtime and fragmentation is smaller. The result is reasonable, as the big test data has bigger range of mallocs space, so the run time will be longer, considering the longer free list, and the more split and merge operations. Meanwhile, the big test data has bigger range of mallloc space which means that there'll be more free space, so the fragmentation will be bigger. The analysis above is also suitable for the results of Best Fit.

2.2 Best Fit

I use three solutions to BF, I list two results of both the solution 1 and solution 3 above. The solution 1 is to find the smallest block. The solution 3 is to set a threshold which is a *suitable* size of Best Fit.

Range of malloc	Solution 1: Run time (seconds)	Solution 3: Run time (seconds)	Solution 1: Fragmentation	Solution 3: Fragmentation
Small	1.818012	0.438971	0.031487	0.043699
Equal	0.994846	0.928301	0.450000	0.450000
Large	60.574932	18.704469	0.040703	0.046823

We can see that the run time of solution 2 is faster than solution 1, and the fragmentation of solution 2 almost the same as or we can say a little bigger than solution 1. So we can conclude that, set a threshold is better. Considering the result of First Fit, the run time is slower, and the fragmentation is smaller. So we can sacrifice the speed to get smaller fragmentation via BF, and we can scacrifice the fragmentation to run faster via FF. By the way, under the equal test data, both of the BF and FF has the same fragmentation (0.45), tha's because all the block of the equal test data's allocated block has the same size. BF and FF only works different on different size, so when running on equal datasets, the fragmentation is the same.

3 Conclusion

Through the implementation of malloc, we can use a meta data for saving the information of adjuncant blocks and free blocks, and we can also use 2 ways to find the free space for malloc. Through the anlysis of the result, we can conclude that BF works better on the fragmentation but run longer time. FF run shorter time but the fragmentation not do as well as BF. Due to the result, I prefer to choose FF when the data range is big, although the fragmentaion of BF is less than FF, they're at the same magnitude. But the runtime of BF is much slower than FF, so I'll choose FF when facing big

range of data.