Malloc Library Implementation

Xueqian Hu | Netid:xh110

ECE650:HW1

1 Malloc and Free Implementation

1.1 Basic understanding of heap and the function sbrk()

In a running program, if we use the malloc function to allocate a sequential memory, this memory will be in the heap. At the beginning of a process, the heap space is 0, which will increase when using malloc. sbrk() is a function to move the top of the heap. For example, when we use sbrk(size), it will move the top of heap "size" steps, meaning the space of the heap increase size byte. And there is a special case sbrk(0) which will return the address of the top of the heap. In the homework, sbrk(0) and sbrk(size) will be useful for us.

1.2 malloc and free

There two parts of the homework I need to implement, one is the malloc() function and the other one is the free() function.
For the malloc() function: void *malloc(size_t size), the input parameter is the number of bytes needed to be allocated and the return value is a pointer that points to the beginning address of the allocated memory.
For the free() function: void ff_free(void *ptr), the input parameter is the beginning address of the allocated memory and there is no return value of the function.

1.3 Implementation Detail of malloc

In my implementation, when allocating a size of memory, I will add extra memory at the begin of the memory to describe that memory. I use a struct called Node, and the Node has 4 members, 1.size_t size(which use to describe the memory I allocate), 2.int isFree(which use to know if the memory is free or not), 3.Node* prev, 4.Node* next. Member 3 and 4 will be used in free list which I will explain later. In my implementation I keep a free list to trace the free memory. When allocating a size of memory. I will first check my free list to find if there is a free memory large than the memory the memory required. There are two policies to find such free block, one is first fit, meaning choosing the first block in heap that meets the requirement. The other policy is best fit, meaning if there are multiples blocks meets the requirements, we should choose the smallest one. If we cannot find such a free block, we can just move the top of the heap by using sbrk(size). But remember, I use extra space to store Node before that data, so actually we should call sbrk(sizeof(Node + size)). If the free block can be found, there will be two situations. The first one is that the free space is too large then we

need to split it. I will cut a part of the space to return and keep the rest part of the free space in the free list. The second situation is that the free space is slightly bigger than require. Then I just allocate all the space but remove the block from the free list. Remember, I have Node* prev and Node* next in the Node struct, so I can remove a block using these pointers.
When I need to free a memory block, I need to insert the block into the free list. The blocks in free list will maintain their relative location. After insertion, work has not done yet. If the insertion block is actually sequenced with its prev block or next block in heap, the those block needs to be merge to one block. So each time after insertion, I will check the prev block and next block to see if any merging will happen.

1.4 Problems I met

I met 2 bug in my work which led to segment fault. The first one is that in malloc function I forgot to return a pointer in the situation that I need to split a block. However, the compiler did not tell me error and the .c file can be compiled. And the second bug is also in this part, I forgot to modify the rest size of the block after split. I solved these bugs by using gdb debbugger, running the program myself and comparing the addresses and pointers with the graph I drew. Then I spent several hours locating my bugs and solved them. And One problem but not bug I met is that when I run the equal size test cases, I found the BF policy spent too much time. Then I checked my code there, I found that when I met a free block whose size is equal to the size I need, I did not break and return but keep searching until the tail of the free list which did useless job and really slow my program down! After I fixed it, all problems solved.

2 Result and Analysis

My result is shown below:

Pattern	First-Fit		Best-Fit	
	Execution time /s	Fragmentation	Execution time/s	Fragmentation
Small	15.548	0.075	6.56	0.027
Equal	21.115	0.45	21.25	0.45
Large	44.09	0.095	57.73	0.04

Figure 1: result

We can see from the result, when all the malloc blocks' size are equal, the First Fit and the Best Fit show the same performance. When the blocks' sizes are in small range, Best Fit will perform better. When the blocks' sizes are in large range, First Fit will perform better. I considered some reasons to explain the difference. When the blocks' sizes are in large range, whether I found the best place are not important, but it took much more time to search a better choice. However when the blocks' sizes are in small ranges, the policy will find a better choice, which means the rest blocks' space is more available, because it will not split a big block if there is a small block that meets the requirement.