Report-650 Project 1

Name: Kaidi Lyu

NetID: kl347

1. Implementation

First, I design a double linked LinkList as my data structure as follows:

```
typedef struct _LinkList{
    struct _LinkList* prevNode;
    struct _LinkList* nextNode;
    size_t size;
    int isFree;
    void* address;
}LinkList;
```

prevNode is the pointer which points to the current node's previous free node.

nextNode is the pointer which points to the current node's next free node.

size is current node's size in memory.

isFree is used to mark if current node is free node or not.

address is current node's address in memory.

(1). ff_malloc() implementation

For the first fit malloc function, my design is to read from the head of the whole LinkList to find the first free node which has enough size to allocate. If there exists such a node, we divide it into two nodes. One is the required node, another one is a new free node. Then we need to change the old node's adjacent nodes' prevNode and nextNode pointer to make sure they all point to the new free node. Also, we need to change all these nodes' isFree value to make sure we mark these nodes correctly. There is one special corner case we need to pay attention to. When we allocate a new node, we need to add metadata, sizeof(LinkList). The free node we find may have enough space for the required size but not enough space left for metadata. In this case, we shouldn't divide this node. Instead, we directly use the whole node as the required node and use its original metadata.

If such nodes don't exist, we need to call sbrk() to create a new node that has the required size. During this whole process, we should update all the nodes' address no matter if they are free or not to make sure they are physically adjacent since the prevNode and nextNode only point to the free node and we need to use the address value to access to all nodes even they are not free.

(2). free() implementation

Since ff_free() and bf_free has the same implementation, here I introduce them together. When free() is called, we first set the target node's isFree value as true. Then according to the address passed in, we traverse the whole free list, find the node which is closest to this target node physically and insert this target node after the node we find. Reset the prevNode and nextNode to make sure the target node is added correctly into the free node list. Then we use the target node's address to read its physically adjacent nodes' isFree value, if these two nodes are free, we conquer them together and reset the prevNode and nextNode.

(3). bf_malloc() implementation

The only difference between first fit malloc function and best fit malloc function is how we find the node to allocate. In first fit malloc function, we traverse the whole list and find the first appropriate node to allocate. But in best fit malloc, we need to compare all possible nodes and find the nodes with smallest size which means it will produce least fragments if it needs to be divided. We implement the finding process by a while loop: Read every node in the list, compare current node with the minimize node and update the current node to its next free node by nextNode. There also exists a special corner case. During this comparation, if current node's size equals target size, we need to break and use this node. On the one hand, this will help speed up the program; on the other hand, if we don't break here, in this specific equal size test, our program has to traverse the whole list every time we call malloc(). Because there are thousands of test, our program will get stuck here.

(4). get_data_segment_size() and get_data_segment_free_space_size()

For data segment size, every time we call sbrk() to allocate a new memory, we add the required size and metadata to data segment size.

For data segment free space size, every time we use free() to free memory, we add the target node's size and metadata to it. When we call malloc and don't need to call sbrk() to allocate a new memory, we need to subtract it. If we need to divide the node to two nodes, we need to subtract the required size and metadata. If we don't divide it and use it directly, we need to subtract the node's size and metadata.

2. performance experiments

Here is the result of my performance experiments. First picture is the result of first fit implementation and second picture is the result of best fit implementation. The table is a summary:

```
kl347@vcm-12510:~/650-project1/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3908328, data_segment_free_space = 253552
Execution Time = 11.059005 seconds
Fragmentation  = 0.064875
kl347@vcm-12510:~/650-project1/alloc_policy_tests$ ./large_range_rand_allocs
data_segment_size = 359826504, data_segment_free_space = 33674720
Execution Time = 87.905080 seconds
Fragmentation  = 0.093586
kl347@vcm-12510:~/650-project1/alloc_policy_tests$ ./equal_size_allocs
data_segment_size = 3360000, data_segment_free_space = 1680000
Execution Time = 28.115431 seconds
Fragmentation  = 0.500000
```

test result of first fit implementation

```
kl347@vcm-12510:~/650-project1/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3712456, data_segment_free_space = 85592
Execution Time = 4.586406 seconds
Fragmentation  = 0.023055
kl347@vcm-12510:~/650-project1/alloc_policy_tests$ ./large_range_rand_allocs
data_segment_size = 340029936, data_segment_free_space = 13855872
Execution Time = 148.340628 seconds
Fragmentation  = 0.040749
kl347@vcm-12510:~/650-project1/alloc_policy_tests$ ./equal_size_allocs
data_segment_size = 3360000, data_segment_free_space = 1680000
Execution Time = 34.519351 seconds
Fragmentation  = 0.500000
```

test result of best fit implementation

		First Fit	Best Fit
Small range test	Time	11.059 s	4.586 s
	Fragmentation	0.064	0.023
Equal size test	Time	28.115 s	34.519 s
	Fragmentation	0.5	0.5
Large range test	Time	87.905 s	148.346 s
	Fragmentation	0.093	0.04

summary

3. analysis of the results

(1). Fragmentation

As we can see from the result, except for equal size test, best fit has a smaller fragmentation than first fit which means best fit is more efficient in space utilization. For equal size test, because we use nodes which all have the same size to test BF and FF, there is no difference in space utilization between BF and FF.

(2). Time

In theory, FF should take less time. Because FF does not need to completely traverse the entire list, it starts allocating memory when it encounters the first appropriate node. But the BF must traverse the entire list to determine that it has found the most appropriate node. My equal size test and large range test results prove it. But my small range test results don't fit the theoretical expectations. According to my analysis, because the data for the small range test is generally small, when using the FF implementation, these small data will be highly possible to fit to a larger size node. Then my program will divide the node. If the next required size is large, it is possible that the originally appropriate nodes in the list have been divided, and we can only call sbrk() to allocate a new memory. The process of calling sbrk() can be time-consuming. But for BF, every time the nodes that need to be processed are the closest to the size, there is no need to worry about a similar situation. That’s why my small range test results don't fit the theoretical expectations.