Project #1: Malloc Library Part 1 Report
ECE 650 – Spring 2023
Yu Xiong (yx236)

1. Overview of Implementation of Allocation Policies
   a. Simple Design Idea
      My initial idea was to get the memory address and store free memory in a list. Since we not only needed to allocate memory but also needed to keep track of the memory and related information, I found that storing only addresses was not enough for this project. So I came up with the idea of using struct to store the information and address of each memory. After that, I found that the data structure linked list was suitable for connecting the relationship between each free memory. However, the C language does not support the direct use of linked list. So I wrote a linked list.
   b. Struct single_node
      The struct single_node represents a single node of the double linked list. Each node contains the start address, the size of the actual data, the free state, and two pointers of the same struct type to the prev and next nodes.
   c. Allocation Memory
      We use the linked list to connect each node that is idle. Their order is based on their address. Addresses will be sorted from smallest to largest. When we get the size of the actual data requirement from the user, the first thing we do is check whether the linked list has a suitable node. A suitable node means that it can hold a size greater than or equal to the actual data size given by the user. If we don't find a suitable node, we create a pointer to single_node_t. It will point to the address where sbrk allocates a sizeof(metadata)+size of memory. The reason we need to consider the size of metadata is that each allocated memory does not contain only the size of the actual information it stores. We need additional information, namely metadata, to interpret the real information and facilitate subsequent actions. We then change the node's free state to 0, which means occupied. And set the nodes connected before and after the node to NULL.
   d. Allocation Policies
      If there are multiple suitable nodes in the free list. Then we have to think about which candidate we should use. At this time, we have two policies: First Fit and Best Fit. First Fit picks the first candidate that comes along. And Best Fit is going to iterate through the free list and try to find the best match.
   e. Split Memory
      After selected the suitable node, we need to consider a problem. Whether the size provided by this node will be enough for another node? That is, should we split the memory space and divide it into two parts. Sometimes we need only a fraction of the amount of memory. The rest of free memory can be used later. If the suitable node can be split, then required part of it will be removed from the list. The other part will remain in the list waiting to be used.
   f. Free Memory

When we do not need to use an allocated memory, we mark it as free memory and add it in the linked list mentioned earlier. Doing so can help us avoid misuse of resources. Because maybe this part of the freed memory can be used to meet future memory requirements.

g. Merge Memory

To avoid fragmentation, we always try to make available free memory in a more complete and continuous form. For example, 5 separate 2 bytes memories are harder to utilize than 1 10 bytes memory. We need to merge adjacent free memory. Merging memories usually occurs after we free a memory. To do this, we first add the newly freed node to the linked list. Then, we need to check whether the addresses of this node and the nodes before and after it are adjacent. Two adjacent nodes mean that they can be combined into one node with contiguous memory. There are three main scenarios. First, it is adjacent to the previous node. Second, adjacent to the following node. Third, adjacent to the nodes before and after. Depending on the situation, we do different fusions to get more continuous nodes.

h. Conclusion

We achieve a second use of allocated memory by constantly updating a linked list of free memory. This helps us use less memory and mentions overall efficiency.

2. Results from the Performance Experiments

For First Fit:

```
yx236@vcm-30604:~/ece650/project1/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 19.137318 seconds
Fragmentation  = 0.450000
yx236@vcm-30604:~/ece650/project1/my_malloc/alloc_policy_tests$ ./small_range_rand_a
llocs
data_segment_size = 3910152, data_segment_free_space = 248008
Execution Time = 7.453213 seconds
Fragmentation  = 0.063427
yx236@vcm-30604:~/ece650/project1/my_malloc/alloc_policy_tests$ ./large_range_rand_a
llocs
Execution Time = 40.165644 seconds
Fragmentation  = 0.093578
```

For Best Fit:

```
yx236@vcm-30604:~/ece650/project1/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 19.075085 seconds
Fragmentation  = 0.450000
yx236@vcm-30604:~/ece650/project1/my_malloc/alloc_policy_tests$ ./small_range_rand_a
llocs
data_segment_size = 3723624, data_segment_free_space = 84264
Execution Time = 2.617682 seconds
Fragmentation  = 0.022630
yx236@vcm-30604:~/ece650/project1/my_malloc/alloc_policy_tests$ ./large_range_rand_a
llocs
Execution Time = 52.794324 seconds
Fragmentation  = 0.041497
```

So we get this graph:

	First Fit			Best Fit		
	Equal	Small	Large	Equal	Small	Large
Execution Time	19.137318 seconds	7.453213 seconds	40.165644 seconds	19.075085 seconds	2.617682 seconds	52.794324 seconds
Fragmentation	0.450000	0.063427	0.093578	0.450000	0.022630	0.041497
data_segment_size	\	3910152	\	\	3723624	\
data_segment_free_space	\	248008	\	\	84264	\

3. Analysis of the Results

   When we execute ./equal_size_allocs, the results obtained by the two policies are not much different.

   If we execute ./small_range_rand_allocs, Best Fit is recommended. In this situation, you can see that the data_segment_size and data_segment_free_space of Best Fit are smaller than those obtained in First Fit. According to the lecture, it is better to have a block of memory as complete and contiguous as possible. For example, 5 separate 2 bytes memories are harder to utilize than 1 10 bytes memory. We want to avoid fragmentation. The less fragmentation, the better. The Best Fit logic for using free memory allows it to handle fragmentation easily. Because we're trying to make the best use of the free memory in the free list. The free memory in the free list used should differ from the required size as little as possible. So Best Fit is doing better than First Fit.

   Now we execute ./large_range_rand_allocs. Accroding to the data we explored, Best Fit takes more time than First Fit. If we have a strict time requirement, Best Fit now is not preferred. Best Fit requires iterating the free list until it finds the best match. When we find the first free memory we can use, we are not sure if it is the best one. We need to keep looking later. But the free list in the real world is generally large enough. If we spend a long time iterating to find the best suitable memory, it might takes a lot of extra time. According to the data shown above, we find that the First Fit does perform better than Best Fit when the size is larger. First Fit stops exploring when it finds the first available free memory. So in reality, it's more normal to use First Fit and it could save time.