Report

- ECE 650 Project 1
- Name: Rui Cao
- NetID: rc384

1. My Implementation

1.1. Overall Design

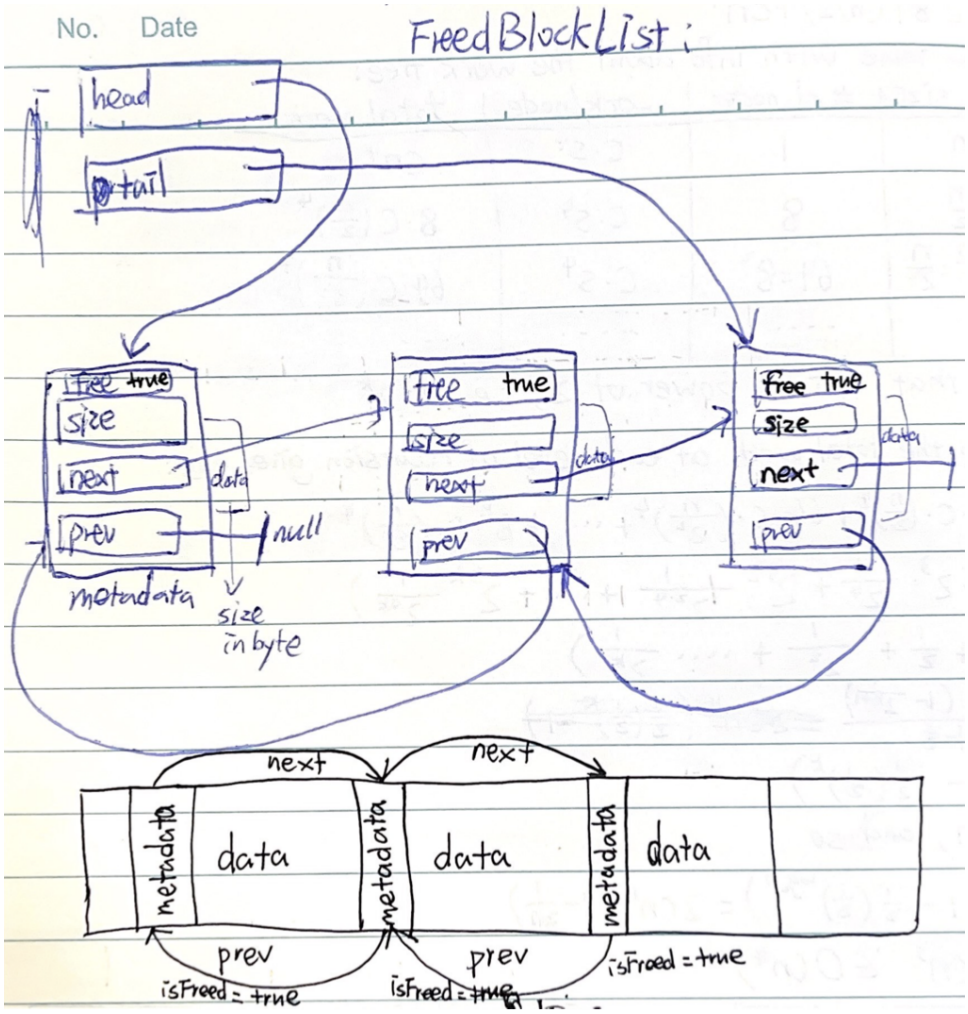
- metadata (struct) is used to describes information about data block (in my implementation, it is for free blocks) in heap. The two pointers next and prev respectively point to the metadata block of the next and previous free block in the linked list. As shown in the figure in Section 2.1.
    - ```
      struct metadata_tag {
          size_t size;
          struct metadata_tag * next;
          struct metadata_tag * prev;
          bool isFreed;
      };
      typedef struct metadata_tag metadata;
      ```
- ff_malloc using:
    - twoKindsMalloc
    - ffFindBlock
    - allocNewSpace
    - splitBlock
    - removeFromFreeList
- bf_malloc using:
    - twoKindsMalloc
    - bfFindBlock
    - allocNewSpace
    - splitBlock
    - removeFromFreeList
- ff_free using:
    - addToFreeList
    - mergeFreedBlocks
- bf_free using:
    - ff_free
- get_data_segment_size:
    - Using global variable: dataSegmentSize.
    - In allocNewSpace, the dataSegmentSize is increased every time new heap space is allocated: dataSegmentSize += size + sizeof(metadata);
- get_data_segment_free_space_size:
      ```

- Definition: size of the "free list" = (actual usable free space + space occupied by metadata) of the blocks in your free list
- So I iterate over my linked list of free blocks, and add the size of every free blocks: freeListSize += (curr->size + sizeof(metadata));

2. Detailed Description

2.1. the linked list of free blocks:

- There is a hint in our README: keep a data structure that represents a list of free memory regions. => I use a linked list that represents a list of free memory regions. As shown in the figure below.
  - Note: the three blocks in the lower part of the figure below need to be merged, because they are three adjacent free blocks.
  - Global variables: headOfFreeList and tailOfFreeList. They represent the head and tail in the figure below, respectively.



2.2. ff_malloc/bf_malloc:

- There are many situations:
  - First, check whether the linked list of free blocks has elements or not.

        - If there is element in the linked list of free blocks, then, check whether there is suitable free block or not (ff_malloc using ffFindBlock; bf_malloc using bfFindBlock): If there is free block that fits an allocation request, need to split free regions if the ideal free region is larger than requested size (> size + sizeof(metadata)) and no need to split if the ideal free region is lower than requested size (because the remaining free space is too small to keep track of). If there is no free space that fits an allocation request, then sbrk() will be used to create that space (using splitBlock).
        - If there is no element in the linked list of free blocks, sbrk() will also be used to create that space (using splitBlock).

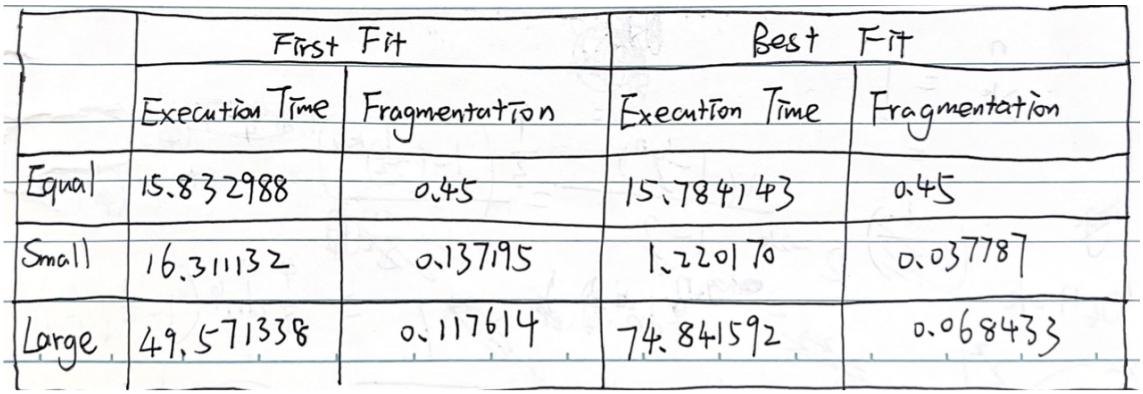
- allocNewSpace: Use sbrk() to create new heap space.
    - Note: For sbrk(), on error, (void *) -1 is returned
- splitBlock: This function is used to split free blocks if the ideal free block is larger than requested size (size + sizeof(metadata)).
    - There are four situations: when there is only one element in free blocks list; when there is more than one element in free blocks list and the input block is the head of the list; when there is more than one element in free blocks list and the input block is the tail of the list; and others. Diffierent situations have diffierent operations for splitting the input block.
- removeFromFreeList: This function is used to remove block from free blocks linked list.
    - There are four situations: when there is only one element in free blocks list; when there is more than one element and remove from front; when there is more than one element and remove from back; other removals. Diffierent situations have diffierent operations for removing the input block from the linked list of free blocks.

2.3. ff_free/bf_free:

- ff_free and bf_free have the same implementation logic.
- First, we need to avoid that the input is a NULL pointer. free() does nothing on NULL.
- After avoiding this coner case, I will use addToFreeList function to insert this freed block into the linked list of free blocks. And then I will merge adjacent freed blocks using mergeFreedBlocks function.
- addToFreeList: This function is used to add block into free blocks linked list.
    - There are four situations: when there is no element in free blocks list; when the block is added to the front of the linked list; when the block is added to the back of the linked list; other additions. Diffierent situations have diffierent operations for adding the input block into the linked list of free blocks.
    - Note: each free block in the linked list is sorted according to the order in the heap. Their order is not random.
- mergeFreedBlocks: This function is used to merge adjacent freed blocks.
    - First, this function will merge the next adjacent free block. Second, this function will merge the previous adjacent free block.

3. Results & Analysis of Performance Experiments

My results of the performance experiments are shown in the figure below.



	First Fit		Best Fit	
	Execution Time	Fragmentation	Execution Time	Fragmentation
Equal	15.832988	0.45	15.784143	0.45
Small	16.311132	0.137195	1.220170	0.037787
Large	49.571338	0.117614	74.841592	0.068433

- equal_size_allocs: This program uses the same number of bytes (128) in all of its malloc calls. And then later free'ing the blocks (in the same order as they were malloc'ed). My results show that the two policies have close execution time and the same fragmentation. Because in the case of allocating blocks of the same size, both policies will find the first free block in the linked list when searching for free block that fits an allocation request, so they have the close execution time. The two policies have the same fragmentation (about 0.5), because the program record fragmentation halfway through (try to repsresent steady state).
- small_range_rand_allocs: This program works with allocations of random size, ranging from 128 - 512 bytes (in 32B increments) and frees a random selection of these allocated regions. My results show that the execution time and fragmentation of the best fit policy are lower than those of the first fit policy. The reasons are as follows. Although the best fit policy will spend more time traversing the linked list of free blocks and looking for free blocks that fits an allocation request, this also leads to the higher utilization of free blocks by the best fit policy than the first fit policy. For example, there are three free blocks in the linked list. Their sizes are 30, 20, 10 respectively in the order in the linked list. Next, the program will allocate memory of size 10, 20, 30 in order. For the best fit policy, the program will not allocate new space in the heap (not using sbrk()), because the three free blocks will be fully utilized. However, for the first fit policy, when the memory of size 30 is allocated, the program needs to use sbrk() to allocate new space in the heap, so the three free blocks are not fully utilized. In the situation of small_range_rand_allocs, the execution of sbrk () will take more time than traversing the linked list and finding the suitable block. Therefore, the execution time of the best fit policy are lower than that of the first fit policy.
- large_range_rand_allocs: This program works with allocations of random size, ranging from 32 - 64K bytes (in 32B increments), which does the same thing as small_range_rand_allocs, except that the size of allocated blocks fluctuates more than small_range_rand_allocs. In this case, it is less likely to find a free block that fits an allocation request, which results in a longer linked list than small_range_rand_allocs. Therefore, in this case, the time spent traversing the linked list and finding a suitable free block is longer than the execution of sbrk (). Therefore, the execution time of the first fit policy are lower than that of the best fit policy.

- Which policy would you like to choose?
  - I think it depends on whether the caller of the function is more pursuing speed or memory space utilization.
  - For speed, when the size range of randomly allocated blocks is large (as in the case of large_range_rand_allocs), I recommend using the first fit policy. But when the size range of randomly allocated blocks is small (as in the case of small_range_rand_allocs), I think it is better to use the best fit policy. Moreover, both policies can be used when only allocating blocks of the same size (as in the case of equal_size_allocs).
  - For memory space utilization, the best fit policy is better. In addition to the best fit policy allowing more free blocks to be used and reducing the allocation of new space (execution of sbrk ()), this policy can also reduce splitting some bigger free regions, leaving some small blocks that cannot be used.