Project Report

Implementation Description

We use first fit as an example to describe. We first create our own struct block as a linked list data structure to store the free blocks and each block has three fields, one for the size of the block, one for the next block to this block, and one for the previous block to this block.

When we call ff_malloc, we iterate through the current free linked list to find the first free block with a size bigger than the size we want to malloc. If we find that block, we check whether the block is bigger than sizeof(block) + size. If the block is bigger than that size, we split the block. If not, we directly remove that block and return the address. If we don't find such a block, we call the new_space() function to allocate new free memory using sbrk().

new_space(): We increase the heap to get new free memory for our malloc and return the newly created memory address. We also need to add this value to the data_segment_size.

ff_split(): We create two blocks, one for malloc and one for the remaining free block. We assign the splitting block's address to the malloc block and assign the splitting block's address + sizeof(block) + size to the new free block. We then change the sizes of these two blocks. After changing the sizes, we remove the splitting block and add the new free block to the whole free linked list with pointers changing for next, prev, head, and tail.

remove_malloc(): This is just a helper function to remove one block from the free linked list, which needs to consider about the case of head, tail, or other cases.

We then need to finish the free function. We first minus the sizeof(block) to get the address of the free block and then we need to add this free block back to the free linked list according to the address order in the linked list. We call the add_free() function, which includes adding the free block back and merging the free block if satisfied. We first check the head case. Then we loop through the current free linked list to find the appropriate position to add the free block and change pointers accordingly. After adding back, we check whether the free block adding block size and its size is equal to the free block's next or the free block's prev adding block size and its size equal to the free block. We merge the block if the case is satisfied and change pointers.

We then need to implement the bf_malloc and bf_free. Bf_free has the same logic as the ff_free using the same helper functions. Bf_malloc has almost the same logic as ff_malloc just we need to loop the linked list to find the smallest block which is bigger than the size we want to malloc with using the same helper functions.

Performance Result

FF	small_range_rand_allocs	large_range_rand_allocs	equal_size_allocs
Execution Time	14.902732 seconds	44.244891 seconds	21.634543 seconds
Fragmentation	0.073883	0.093421	0.450000

BF	small_range_rand_allocs	large_range_rand_allocs	equal_size_allocs
Execution Time	6.426427 seconds	58.176599 seconds	21.496540 seconds
Fragmentation	0.026958	0.041318	0.450000

Result Analysis

For small_range_rand_allocs, we can see that the execution time of BF is much faster than that of FF and the fragmentation ratio is also smaller. The reason for that is BF always finds the smallest satisfied block to do allocation, which is always the best fit. In contrast, FF only finds the first fit. Then though BF may take a longer time in finding the best fit, it can efficiently use its free blocks without calling the new_space() function with sbrk(). Instead, FF may continuously call sbrk() to request new memory space. This results in a faster runtime and a smaller ratio in BF.

For large_range_rand_allocs, we can see that the execution of FF is a little faster than that of BF and the BF still has a smaller fragmentation ratio. The reason behind this is pretty similar to the small_range_rand_allocs case. The difference is that the sizes of blocks in the large case vary a lot as compared to those in the small case, which may cause each iteration of the whole free linked list to find the best-fit block to take longer time than usual and this may even exceed the time used for sbrk() in the FF case and result in a longer runtime.

For equal_size_allocs, we can see that FF and BF have almost the same execution time and the fragmentation ratios for these are also the same. The reason for that is the program uses the same number of bytes (128) in all of its malloc calls, which results in almost the same time for FF to find the first fit and BF to find the best fit. This makes the runtime almost the same and the fragmentation ratios close to 0.5.