Kaleab Belete – G00918609

Hamza Mughal – G00903050

Prof. Rangwala

Lab 3 Report

Overall Design and Data Structures

The overall design of the project was broken into multiple phases for us. First, in mdriver.c in the get_seglist_info() function, we get input from the user which is used solely to configure the segregated list(s). Directly after the input is entered, we consider a series of checks on it (such as making sure numbers are in a non-decreasing order). We were given the choice of picking our own return type and parameters – and we chose not to have either. We chose this because it was easier to declare global variables (that would be used throughout multiple files and functions). We declared additional global variables segListSize and segList which are used in the mm_malloc, mm_free, and mm_realloc.

After getting input from the user, the main function in the mdriver file is the mm_init function. Within the function, we initialize the segregated list(s), and set fields as necessary based off of the input. In order for us to create a dynamic array of pointers that pointed to its own mem_ptr, we had to declare a data structure and edit the existing data structure (struct m).

The data structure that we created was struct b. This new data structure contains a next pointer which points to a mem_ptr, and also start and end int variables. The start and end variables are set based off of the user input, and are primarily used to check what the size (bounds) of the segregated list is. Edits that were made to the existing data structure m were additions of a nextBlock pointer, previousBlock pointer, int isFree, and int index. The nextBlock and previousBlock pointers are used to connect all mem_ptrs, weather allocated or not, across multiple segregated lists in address order. The next and previous pointers are for mem_ptrs within the segregated list. isFree is used to check if a block is free or not, and index is used to check what segList a mem_ptr is in.

For the mm_malloc file, the design was to loop through the segregated list(s) and check if there is a free block from which we can allocate memory from. The blocks are found using a first-fit heuristic. When an appropriate block is found, we allocated memory from it, and set the appropriate links/variables. After memory is taken from the block, any excess memory is freed by calling mm_free. We took this approach because by calling mm_free the excess blocks can be linked into the appropriate position and be coalesced if necessary. It also allowed for better error handling due to a reduction in code reuse.

For the mm_free file, the design was to look at the previousBlock pointer (if it is not NULL), and check if the previous block is free. If it is free, the block (that was passed in) is updated and the old previousBlock pointer is freed. The same occurs with the nextBlock pointer. This is where the previousBlock and nextBlock pointers come into great use, these additions allow for constant time coalescing rather than looping through segregated list(s) to see whether they are adjacent in memory or not. Afterwards, we link the block into its proper spot by first using binary search to find the correct segregated list, and then looping through the appropriate segregated List and putting the block in its correct spot based on low to high address ordering.

For the mm_realloc file, the design is broken up into four different cases. The first case is to check if the blocks size is equal to the size that is requested and return the same block. The second case checks if the block has a greater size than the updated requested size. If so, we split the excess space into a new excess block, necessary variables are updated, and then the excessBlock is freed by calling mm_free. In the third case the requested size is greater than the current size of block, so we check if the next block contiguous in memory is free and has enough memory to give. If the next block has enough space, we combine the blocks and update the variable and links as needed making sure to free the excess memory and any unused mem_ptrs. Finally, if all previous cases have failed we simply call mm_free and mm_malloc to get a new block to return.

Self Assessment

The strengths to our implementation of the segregated list(s) is that coalescing of blocks occurs in constant time. There is no need to loop through segregated list(s), and check if they are adjacent in memory. We also have the advantage of using binary search to find the appropriate segregated list due to the segregated lists being size ordered, making our search time ~O(log(n)).

At the time of submission, we do not have any known clear limitations to our project implementation. We made sure to follow the specifications and we were able to pass all provided test cases. We also made our own traces and tested the program over multiple edge cases. We made sure to test different numbers of segregated lists, in the end we did not find any bugs or limitations.

At the time of submission, from our testing we could not find any known major problems with our project implementation. The only known issue we had was some left over reachable allocated blocks that we did not free because we were not allowed to modify the main function.

A few extensions we would have liked to implement was:

1. A best-fit instead of first-fit heuristic. This is another possible strength of our design, because we have address ordered pointers, we do not need the segregated list to be in address order to expand or coalesce blocks. We could have shifted our design to implement a best-fit heuristic to reduce fragmentation and increase memory efficiency without sacrificing throughput.

2. Another big extension we could implement is a faster design. This one is a bit tricky, for the most part, any major speed-up we could think of was introduced into our program, so as of now we do not know exactly how we would make it faster but given more time I believe we can find some tradeoff to increase the throughput of the program.