Project Report: Malloc Library Part1

Kaifeng Yu | ky99

1. Overview of Implementation

In this assignment, I implemented my own version of two memory allocation functions from the C standard library: *malloc()* and *free()*. For each of the functions, I compared two different strategies: First Fit and Best Fit.

1.1. Data Structure

One of the key points of memory allocation functions is a data structure to store allocated memory blocks. In my implementation, an allocated memory block comes with a metadata block in the front. The metadata block stores basic information of the allocated block, including its size, whether it is free and information about its adjacent free blocks. The data structure of a metadata block is defined as below:

```
typedef struct meta_data {
   size_t size;
   size_t is_used;
   struct meta_data * prev_free_block;
   struct meta_data * next_free_block;
} meta_data_t;
```

Another key point of managing allocated data is to keep a free list to keep track of all allocated blocks that are not currently used. In my implementation, I construct a doubly linked list using the above data structure to function as a free list. The blocks in the free list are sorted in ascending order by their address.

In addition, I also have two global variables, *meta_data_t * free_list_head* and *meta_data_t * free_list_head*. These two variables are used to keep track of the first block and the last block in the free list, respectively. They are very useful when adding or removing blocks from the free list.

1.2. Implementation of functions

For *malloc()* function, we first call a *try_exist_block()* function to try to find an appropriate block(for first fit strategy, it is the first large enough block; for best fit strategy, it is the smallest large enough block) in the free list, by traversing through the free list. In the *try_exist_block()* function, if we do find an appropriate block, we will decide whether to split it into two blocks base on its size. To be specific, if the size of the block is larger than the size we desired plus the size of our meta data, we will split it into two blocks, use the first block to return back to caller (remove it from the free list) and still keep the second block in the free list; if not, we will return the whole block without splitting it, even if the size of the block is larger than what the caller desired. On the contrary, if the *try_exist_block()* function do not find an appropriate block, it will return a NULL pointer to *malloc()*, and then *malloc()* will *call add_new_block()* which will use *sbrk()* to allocate a new block.

For *free()* function, we first use *add_to_free_list()* function to add the target block into the free list by traversing through the free list. Then, we use *try_coalesce()* function to determine if the previous or next free block in the free list is adjacent to the target block. If so, we will merge them into one free block.

2. Experiment Results and Analysis

2.1. Results

The experiment results are shown below:

Pattern	First Fit		Best Fit	
	Execution Time /s	Fragmentation	Execution Time /s	Fragmentation
Equal	16.06	0.450	16.09	0.450
Small	5.28	0.060	1.28	0.022
Large	36.22	0.093	42.36	0.041

2.2. Analysis

For equal_size_allocs pattern, the program uses the same number of bytes in all its malloc calls. As you can see, both execution time and fragmentation for first fit strategy and best fit strategy are almost the same. This is because that when allocating and freeing same number of bytes, both strategies are looking for the first block in the free list when allocating and they are reusing the same blocks in the free list. In another word, both strategies behave the same in this situation thus they have similar runtime and fragmentation results. Another thing to notice is that the fragmentation for both strategies is 0.45, which seems very high at the first glance. This is because the program pre-allocate half of the blocks before the experiment begins and only allocate and free a small fraction of blocks from the other half of blocks once at a time after the experiment begins, so the fragmentation is around 0.5.

For small_size_allocs pattern, the program works with allocations of random size, ranging from 128 - 512 bytes. The results shows that the best fit strategy has both shorter execution time and smaller fragmentation comparing to the first fit strategy. This seems a little bit strange at the first glance, because we typically believe that best fit strategy will spend more time searching in the free list to use the free blocks more efficiently. This result might be caused by the following two facts: 1. The free list is relatively short 2. It is relatively likely for best fit strategy to find the best block in a few steps instead of traversing through the whole free list, since the range of the allocated blocks' size is relatively small (from 128 – 512 bytes, in 32B increment). In this case, the best fit strategy does not spend too much time on searching in free list. On the other hand, since the first fit strategy does not use the free blocks efficiently, it will need to call *sbrk()* to allocate space more frequently, which might play a bigger role in the runtime.

For large_range_allocs pattern, the program works with allocations of random size, ranging from 32 – 62k bytes. This experiment is very similar to small_range_allocs, except that the range of the size of the allocated blocks is much larger. In this case, it is much harder for the best fit strategy to find the best block in the free list, since there are much more possible block sizes. It might need to traverse though the whole free list in many times, which will lead to a much longer execution time than the first fit strategy. In this case, the time spend on calling sbrk() is not that important anymore. As the results show, the first fit strategy has a slightly shorter execution time but a larger fragmentation than the best fit strategy, which matches our theory.

As we have discussed above, the best fit strategy is more efficient in the small_size_allocs experiment and the first fit strategy is slightly more efficient in the large_size_allocs experiment, since the difference in runtime will become larger when we allocate more spaces (which means more free blocks). Therefore, if you are only allocating a few blocks or your desired blocks have similar sizes, I recommend best fit strategy. Otherwise, first fit strategy might be a better choice.