ECE650 Malloc Library Part 1

1 Requirements & Summary of Developments

1.1 Project requirements

1.1.1 `malloc` and `free` functions

The template of `malloc` function is: `void * malloc (size_t size);`

`malloc` takes a parameter of type `size_t`, which indicates the number of bytes users need. It returns a pointer of type `void *` that points to the allocated memory.

When the `malloc` function is called, it returns the address of an allocated block. On failure, it returns `NULL`.

The template of `free` function is: `void free (void * ptr);`

`free` takes a parameter of type `void *`, which indicates the memory address that needs to be released.

When the `free` function is called, it checks whether the pointer is within the heap. If not, return directly. If so, mark the block as free and release it back to the memory.

1.1.2 Environment & tools

We developed the `malloc` library in C language under Linux system. Using the `Makefile` provided, we built the source files into a library that can be called outside the current directory.

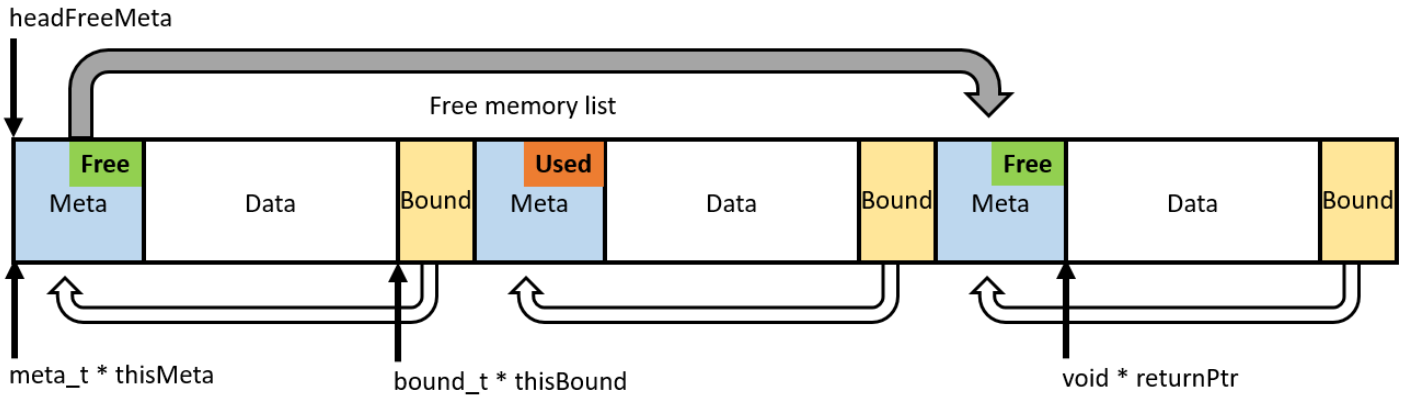
1.2 Report organization

In **Section 2**, we discussed the design and implementation, including data structures and algorithms; testing is also discussed here. In **Section 3**, we discussed the results and performance analyses, including operation time and fragmentation. **Section 4** is the reference.

2 Design, Implementation & Testing

2.1 Design choices

In this project, we used extra memory space to store the block information.



2.1.1 Memory block structure

The memory block consists of three parts: metadata, data space and bound.

The metadata (of type `meta_t`) stores the size of the data space (which is always aligned to a multiple of 16), a flag to indicate the free status and a pointer to the metadata of the next free memory block (`NULL` if the block itself is not in the free list).

The bound contains 1 field: `thisMeta`, which is a pointer (of type `meta_t`) to the metadata of the current block. It helps us to locate the current block from backward.

When `malloc` allocates a block, the address of the data space is returned to the user.

2.1.2 Free list structure

A free list is designed to store the address of free memory blocks.

The free list is a linked list. In `my_malloc.h`, a static variable `headFreeMeta` is defined. It indicates the head of the free list. Every pointer in the list is of type `meta_t`, pointing to the meta address of the next free block. The addresses in the list are in ascending order.

2.2 Implementation logic

2.2.1 Fit block searching

Two policies are supported in this project: first-fit and best-fit. In both policies, `malloc` traverses the free list and checks the size of each block. On first-fit, once `malloc` finds a block of enough space, it makes use of the block without further checking. On best-fit, `malloc` examines all free blocks and uses the one with enough but smallest space. It stops traversing in advance only if it founds a block which has just enough space.

Both `ff_malloc` and `bf_malloc` calls the function `perform_malloc` with template

```
void * perform_malloc(size_t inSize, void * (* get_fit)(size_t));
```

It will calls either `get_ff_fit` or `get_bf_fit` to search for free blocks.

2.2.2 Function logic

`malloc` working flow

When either `ff_malloc` or `bf_malloc` is called, it will call the function `perform_malloc`. `perform_malloc` will then use the function pointer (which points to either `get_ff_fit` or `get_bf_fit`) to get the suitable memory block. Both `get_ff_fit` and `get_bf_fit` check the size of the block: if there is enough space for a new block, it will split the current block (function `split_block`) into two parts and return the pointer of the latter one; otherwise, it simply returns the found block and remove the pointer from the free list (function `delete_node`).

`malloc` uses the function `renew_info_malloc` to mark the block as allocated and then return the pointer to the data space.

`free` working flow

When either `ff_free` or `bf_free` is called, it will call the function `perform_free`. `perform_free` will use the get to the meta part of the block and mark it as free and try to merge the block with other blocks (function `merge_back` and `merge_front`).

`merge_back` checks whether the next block is free. If so, it deletes the next block from free list (function `delete_node`) and changes the settings of the newly created block. `merge_front` is used after `merge_back`, which checks whether the previous block is free. If so, it changes the settings of blocks; if not, it adds the current block to the free list (function `add_node`).

2.3 Testing

The code was tested on the ECE551 VM with both the provided kit and some other cases.

When developing, I used `general_tests` to check whether the pointer arithmetic was done correctly. After that, I used the `alloc_policy_tests` to check whether the codes worked efficiently. At first, `malloc` worked unusually fast, so I added extra codes to print messages about how functions allocated space, freed pointers, merged and split blocks. By checking the printed free list, I noticed that the list was not successfully built, so I fixed the code. After debugging, I wrote an extra test file to allocate space for different types of variables (`int`, `char`, `char*`, etc.). The cases are designed to get different results for FF and BF. By comparing the printed messageand hand-calculated results, I double-checked that the codes could work.

3 Performance Results & Analysis

Our `malloc` and `free` got stable results on intensive tests. For equal size, we ran 3 same tests and got the average; for small and large allocation, we set the random seed to 0, 1 and 2 and got averages after 3 tests respectively. The results are as follows:

	FF		BF	
	Time	Fragmentation	Time	Fragmentation
Equal	20.14838	0.999889	20.16704	0.999889
Small	9.215181	0.624184	2.623498	0.904712
Large	51.66187	0.843796	61.26451	0.960573

Both policies got almost the same results for equal-size, because both returns the first free block. For small-range allocation, FF works slower than BF and gets a lower fragmentation. For large-range allocation, FF works faster than BF and gets a lower fragmentation. Both fragmentations rise when range goes larger, because the rising range raises the uncertainties of block sizes and creates more split blocks.

In both situations, BF got a higher fragmentation because it always chooses the block with smallest allowable size. When blocks are just a little larger than requested, they will be split into two part: one block for user and another block that has such a small size that it is nearly impossible to be reused. A possible explanation for BF'S faster run time on small-range is that BF traverses through the list and is less likely to need splitting and merging; but when range becomes larger, FF overtakes BF because it needs less time for list traversal.

Which policy to choose depends on user requirements. When users care more about time than fragmentation and only need small-range allocation at most time, they may choose BF. However, if users either want a higher efficiency on memory management or need to do many large-range allocations, FF is a better choice.

4 References

[1] Malloc. *CS241: System Programming Coursebook*. University of Illinois.

[2] Saelee, Michael. Mallow Slides. Illinois Institute of Technology.