Contents

1 Design

When calling `sbrk` to allocate new heap memory to the program, I abstract the allocated part to be a block region of bytes. Each block contains its metadata (used by the block structure) and the bytes used for memory storage.

I use a doubly linked list to store only the free block regions. My block structure contains the following fields:

- `size_t size`: number of bytes this block contains for memory storage (not including metadata bytes)
- `block_t * prev`: pointer to the previous free block region
- `block_t * next`: pointer to the next free block region

1.1 Implementation of `malloc`

I think the only difference between the implementation of `ff_malloc` and `bf_malloc` is the way of linked list traversal, so I implemented these `malloc` functions with the same helper functions and different list traversal method.

Specifically, the helper functions are:

- `void * expandMemory(size_t size)`
  - When no free block region is larger than the required size, call sbrk() to create one, and return a pointer to the start of the newly allocated memory, like sbrk()
- `updateBlock(block_t * cur, size_t size)`
  - Given a free block region of size larger than we need, mark the first `size` byte to be occupied memory region, and update the free block region linked list with shrinked block.
  - We distinct two cases in this function.
    1. If the block's available free size is only slightly larger than what we need (a.k.a the remaining free bytes are not enough for metadata header bytes), then we throw away those unused bytes, and delete the block entirely from the linked list.
    2. If the block is much larger (a.k.a the remaining bytes can make a new free block), then we slice the block into two parts, use the first part to store data, and put back the shrinked free block region into list.

After finding the suitable free block region to allocate memory, I use these two helper functions to actually implement malloc. If no suitable free block is found, then I create one and use it with `expandMemory`. Else, I update (delete or slice) the chosen free block region with `updateBlock`.

The different algorithms for *First Fit* and *Best Fit* to find the suitable block are:

- First Fit: traverse the free block region linked list **up to the point** we find the first block that is large enough to use the memory.
- Best Fit: traverse the **whole** free block region linked list to find the suitable block with the smallest size to enhance data usage efficiency.

1.2 Implementation of `free`

I think that `free` function would do the same thing no matter what fit policy it is, so I implemented a `freeHelper` function with:

- `void addBlock(block_t * ptr)`
  - Add a free block region into the doubly linked list.
- `void mergeBlocks(block_t * lhs, block_t * rhs)`
  - Merge two free block region in list if adjacent to each other.

Specifically, every time I free a block with `freeHelper`, I insert the new free block region into the linked list using `addBlock`, do two merges (both left and right side) with `mergeBlock` upon insertion.

2 Performance

2.1 Performance result presentation

I run the tests with `-O3` optimization flag on a 2-processor, 4 GB base memory, ubuntu20 virtual machine on Duke VM server.

Figure 1 shows the results for First Fit:

```
sl846@vcm-38454:~/650/my_malloc/alloc_policy_tests$ ./small_range_rand_allocs
data_segment_size = 3705640, data_segment_free_space = 273784
Execution Time = 11.403441 seconds
Fragmentation  = 0.073883
sl846@vcm-38454:~/650/my_malloc/alloc_policy_tests$ ./equal_size_allocs
Execution Time = 15.088542 seconds
Fragmentation  = 0.450000
sl846@vcm-38454:~/650/my_malloc/alloc_policy_tests$ ./large_range_rand_allocs
Execution Time = 37.522322 seconds
Fragmentation  = 0.093421
sl846@vcm-38454:~/650/my_malloc/alloc_policy_tests$
```

Figure 1: First Fit Results

Figure 2 shows results for Best Fit.

To make data more intuitive, I made a table for comparison of results.

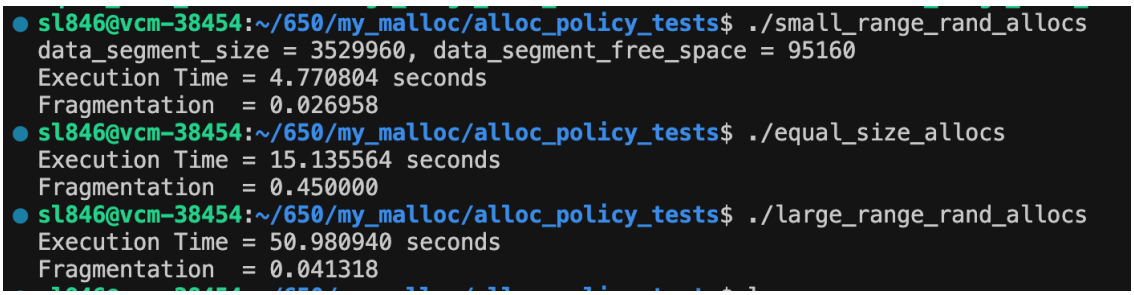

Figure 2: Best Fit Results

Table 1: First Fit and Best Fit comparison

	FF run time (s)	BF run time (s)	FF fragmentation	BF fragmentation
small rand alloc	11.40	4.77	0.074	0.027
equal size alloc	15.09	15.14	0.450	0.450
large rand alloc	37.52	50.98	0.093	0.041

2.2 Result analysis

2.2.1 Analysis of execution time

With `large_range_rand_allocs`, the Best Fit policy executes with more time than First Fit, as is expected. This is because First Fit policy would just naively take the first available free block region no matter how big it is, yet Best Fit would sacrifice more time to traverse the whole list to find the best fit to improve storage efficiency.

With `equal_size_allocs`, Best Fit and First Fit take about the same time, because the program uses the same number of bytes (128) in all its malloc calls. This is reasonable because I designed my Best Fit to end as soon as it found the block of the exact same size of requirement without having to traverse the whole list, which in this specific case acts the same as First Fit policy.

With `small_range_rand_allocs`, Best Fit runs surprisingly faster than First Fit, which is not very reasonable. My expectation is that Best Fit should always run slower than First Fit, because it has more blocks to traverse and check. After looking things up, I assume that it's because my Best Fit policy somehow gets automatically optimized by the compiler, which used the real library function `malloc` behind the scene.

2.2.2 Analysis of fragmentation

Fragmentation is calculated by the amount of unallocated data segment space divided by total data segment space. Except for the special case of `equal_size_allocs` where Best Fit and First Fit reasonably makes no difference, in all other cases, Best Fit has much lower fragmentation than First Fit. This is to my expectation because Best Fit policy chooses to sacrifice execution time in exchange for a better memory usage efficiency, achieved by always `malloc`-ing at the smallest suitable free block region. Moreover, Best Fit policy used less data segment size than First Fit, because it efficiently used its freed memory and avoided naively `sbrk`-ing too much new heap memory.

3 Things I learned along the project

- The most important thing I learned from this project is to use abundant `printf`s to debug. Sometimes, with too many iterations, even GDB cannot really point out where the true problem lies. For example, I got a lot of segmentation faults (core dumped) with GDB flagging my slice block logic in `updateBlock` to be problematic, while the true problem that influenced this to dysfunction is because my `mergeBlock` is not good. I did wrong merging with free block regions, leading to unexpected behaviors in block updates later on. I could not have spotted this problem without printing a lot of information and checking them carefully and patiently one after another.
- Always check `NULL` input corner cases. Strange `NULL`s that seem to have come out of nowhere have been the main cause of my segmentation faults.
- Debugging can be emotional, and it's totally OK. When feeling down, take a deep breath, go out and stretch a little, talk to friends, take a nap. Debugging needs a sharp brain and a lot of patience.
- When doing pointer arithmetics, remember to cast to the same reasonable type. For example, instead of adding `sizeof(block_t)` (number of bytes of metadata) directly to a pointer of type `block_t *`, I need to first cast the pointer into `char *` to make sure I'm really adding by Bytes.
- I first tried using `void * start_address = sbrk(0);` as a global variable to mark the beginning of the program, but the IDE complains at `sbrk` that "initializer element is not a compile-time constant". After looking it up, I realized that C global variables must be initialized with constant expressions, so the right way to achieve my goal is to set `start_address` as `NULL` at compile time, then set it to the correct value at runtime by calling `sbrk` at the beginning of main. However, I then realized that I cannot control what is in the main function (as this will be done by the auto-grader), so I proposed an easier and more intuitive way to keep track of the whole program heap memory.