ECE650 - HW1 Report - tc233 - Ting Chen

Requirements

The report should include an overview of how you implemented the allocation policies, results from your performance experiments, and an analysis of the results (e.g. why do you believe you observed the results that you did for different policies with different malloc/free patterns, do you have recommendations for which policy seems most effective, etc.).

Allocation policies

I use 2 linked lists to keep track of memory allocated by my malloc function. One to keep track of all memory allocated, no matter it is free or not. The other one only keeps track of all the memory that are free to allocate, which makes it faster to search for a free block when calling malloc.

Before that, I only used 1 linked list to keep track of all the memory allocated with my malloc function. The runtime for that practice is really long. It takes around 550 seconds to pass the equal size test, which is unacceptable. See ref #1.

Since I migrated from my previous design to the current, the code is sort of convoluted. I'm running out of time before submission, hence I just leave it as is. After tonight, I'll add more comments and clean up a little bit.

When splitting a large block, I just test to see if it can hold current size + METADATA size. If it does, I split this block into 2 pieces and add the second piece to the free blocks list.

When merging blocks, I traverse to the first free neighbor block, i.e.

```
while(b -> prev && b -> prev -> is_free) {
    b = b -> prev;
}
```

And from that block, I merge all the following free blocks and add the merged one to the free blocks list. Be careful - when merging the blocks, the tail pointer might need to be moved to the new merged block. Also think about changing the tail pointer when splitting the last block. Hell no! Why in first place did I design like that?

I also included an alignment function which align all the malloc'd blocks to 8 bytes. The reason is describe in appendix.

Finally, I added a validation function to check if the block of memory is allocated with my malloc function. This function might be redundant. I'll discuss with TAs later on to see.

Performance experiments

Below attaches the results from my current design.

### First fit

```
./equal_size_allocs
Execution Time = 22.478081 seconds
Fragmentation    = 0.450000

./large_range_rand_allocs
Execution Time = 90.789174 seconds
Fragmentation    = 0.093517

./small_range_rand_allocs
data_segment_size = 4191040, data_segment_free_space = 218624
Execution Time = 3.604416 seconds
Fragmentation    = 0.052165
```

### Best fit

```
./equal_size_allocs
Execution Time = 23.274010 seconds
Fragmentation    = 0.450000

./large_range_rand_allocs
Execution Time = 123.631368 seconds
Fragmentation    = 0.041092

./small_range_rand_allocs
data_segment_size = 3954432, data_segment_free_space = 71168
Execution Time = 1.074224 seconds
Fragmentation    = 0.017997
```

Analysis of the results

First Fit seems to be the better method of allocating memory. It have better efficiency in finding free blocks, and do not sacrifice too much in terms of fragmentation.

Comparing with First Fit, Best Fit is expected to have lower fragmentation ratio and longer runtime. The results confirmed this prediction. Since Best Fit need to traverse the whole free blocks list to find the best fit, it would always take no less than First Fit to find a block. However, due to the intrinsic limitation, First Fit would have larger fragmentation ratio since it might be the best fit to requested size of memory.

According to my result, First Fit seems to have twice as much in fragmentation ratio in random tests. They both run around 20 seconds for the equal test, since First Fit and Best Fit should both pick the first free block they encounter in the free list, as that block is already the Best Fit.

The only odd comes from FF's small comparing to BF's small. Despite the normal prediction of fragmentation ratio that FF should be worse than BF, it's weird to find that BF is actually faster than FF in execution time. To my guess (running out of time to test systematically), it should be related with some cost balancing between searching the linked list and opening extra space with sbrk() system call. Since the small random tests can have rather shorter free blocks list, it might be better just searching to find the Best Fit rather than always hast and waste some nice big blocks.

There are possibly some bugs for best fit in my code, due to the clumsy implementation of head and tail pointers to the linked lists, though I passed all the tests without segfault or memleak. I'll try to fix them during the second HW and probably refactor the whole program.

I've heard that some students can make the runtime under 1 second. To my guess, if I switch totally to the 1 linked list version, I might catch up with their speed, since currently I still need to traverse the full list which takes some time.

Below are some notes taken during development. Feel free to skim or skip.

## malloc referred resources

- Official reference and definition of malloc:
http://man7.org/linux/man-pages/man3/malloc.3.html

- Reference that I read but did not copy from:
https://danluu.com/malloc-tutorial/

- size_t maximum size:
https://en.wikipedia.org/wiki/C_data_types

## testing

### General test

```
valgrind --leak-check=full --show-leak-kinds=all ./mymalloc_test
```

- new + METADATA_SIZE

When moving pointers, +1 means advancing by 1 block, not 1 byte...
Haven't done C for a while, and forgot this one. Took me 1 hour to find it out.

https://stackoverflow.com/questions/6449935/increment-void-pointer-by-one-byte-by-two

### alloc policy test

valgrind --leak-check=full --show-leak-kinds=all ./equal_size_allocs

## Either to track malloc'd blocks

In my previous implementation, I tracked all the blocks malloc'd and free'd from my funcs. However, after discussing with TAs, I found it might not need to check that. Thus, the minimal items or the METADATA should be something like :

```c
struct _metadata_linked_list {
    size_t size;
    bool is_free;
    struct _metadata_linked_list* next;
};
```

## Alignment issues

I read some materials about how alignment of memory would affect the performance of it. It suggests using 8 bytes aligned blocks for better performance.

In terms of splitting blocks, we can always round up blocks to 8n so that it would be better.

## More to add

- A single linked list to record all the free blocks

- Efficiency improvement thoughts

to improve best fit method of searching, pointers to METADATA blocks could be saved in a binary search tree ordered by size.

to improve overall performance, one can maintain several different lists that fits different sizes. e.g. 8b, 32b, 64b, etc. so that malloc and free can directly go to corresponding lists.

one can add all the addresses allocated to a hashset, so that the validation process could be faster. (C++ unordered set), or even do not need to validate.