Assignment 1: Malloc

Junming Xing

September 15, 2022

1. **Implementation**
   The metadata structure is shown below.

Listing 1: C implementing metadata structure

```
typedef struct Header {
    size_t size;
    size_t left_size;
    struct Header *next;
    struct Header *prev;
} Header;
```

   (a) **Multiple explicit free lists**
   I used a global variable array to store the heads of each free list. Each free list is a doubly linked list with a head and a tail. Each time when an allocation is happened, we calulate the free list index according to the size it requests.
   The list index $idx = min(\frac{aligned\_Request\_Size}{unit\_Size}, \text{NLISTS} - 1)$
   After finding the free list using the *idx*, we call function *fit_list()* to find the fit block using first-fit strategy. If the fit block is found, we remove it from the free list and try to split it using *split_block()*(if the block is large enough).

   (b) **Fence posts**
   For each consistent chunk, I used a dummy block at the end of the chunk to mark the end of the chunk. And, in the metadata, there is a field called *left_size*. It records the size of neighboring block in a consistent chunk.
   In a consistent chunk, the *left_size* field in the first block is always 0; and the last block is always dummy(*size* field is always 0); while any other block in the same chunk has non-zero *size* and *left_size*. So, we can easily detect the boundary of each consistent chunk, this can avoid going out of boundary in coalescing free blocks operation; coalescing without boundary may cause a segmentation fault.

   (c) **Constant time coalescing with boundary tags**
   After calling the *myfree()* function and one block is set free, we can using the *size* and *left_size* fields to get the header pointers of left and right blocks. If the neighboring block is free, we update the *size* or the *left_size* fields and merge the consistent free blocks.

2. **Optimizations**

   (a) **Metadata reduction**
   As the size of each block is aligned to one word size, we can use the least significant bit of the *size* field to record whether the block is free or not; Also, when we allocate the block our actual allocated space for each block takes over the two pointers place.

(b) **Requesting additional chunks from the OS**
If no block fits in this free list, we call function *init_list()* to request a new chunk from the OS. After requesting a new chunk we initialize it as a new block and add it to free list.

(c) **Optimize large allocation requests**
If the request allocation size is less than *kLargeAllocationSize*, then *mymalloc()* will use the free list to handle this allocation request; while if the request size is equal to or bigger than *kLargeAllocationSize*, then *mymalloc()* directly use *mmap()* for this allocation.
Different allocation strategys need different deallocation methods, so I need to record which allocation strategy is used for each block. I used the *left_size* field in metadata to record block's type. If a block is allocated by *mmap()* directly, the *left_size* in it's metadata is -1. As none of blocks in the free lists has a *left_size* field value that is -1. We can distinguish different type of blocks
Also, the *myfree()* function need to detect different types of the block to be free. If it used a free list method we set it free, coalesce and append it to the free list where it is used to be; if it is allocated by large allocation method, we call *munmap()* to free this large block.

3. **Two Implementation Challenges**

(a) One thing is that I was using vscode to edit my code and have some bugs that mess up with pointers which caused the segmentation fault. And, the vscode's debug modecan't locate the line where the error happens instantly. Finally, I switched to CLion to debug this program, in which it can automatically jump to the line that cause an exception and keep all the status and data in memory at that time.

(b) How to detect if the point we want to free is in range of our *mymalloc()* is challenging, especially when implementing the multiple free lists and the optimization of requesting new chunk from OS. I created a struct named *Range_Mark* to record the range of each free list. It is actually a single linked list without head.

4. **Two Key Observations**
CPU: AMD Ryzen 5 4600H, RAM: 16GB, OS: Windows 10, WSL: Ubuntu 20.04 LTS

(a) In the testing *random_ptr0-expect*, I came across segmentation fault which is because I didn't check if the pointer to be free is in the range allocation using *mymalloc()*. I finally used *Range_Mark* that I mentioned above to record the range of each free list. The time complexity of this check operation is $O(t)$ where t is the number of times this list request chunks from OS.

(b) To detect the pointer is in allocated range when we want to free it, at first, I was trying to install a signal handler to detect the SIGSEGV. But I find that it takes on average 1.5s in default benchmark; Then I used *Range_Mark* that I mentioned above. It takes on average 0.5s in default benchmark. The result wasn't as I excepted because in the first implementation, it don't need to initialize *Range_Mark* but takes more time. I googled it and found that transmitting the signal and invoking the handler take time and that may cause this result. Meanwhile, I found that the signal handler method can miss something that should be detected.