ECE650 - HW2 Report - tc233 - Ting Chen

Part 1 Implementation

In the first assignment I used the two linked lists implementation for both malloc functions. Upon that, I built the two required malloc functions in this assignment.

1. ts_malloc_lock()

For the lock version of malloc, it's intuitive to put some mutex locks around the critical section, which manipulates the linked lists. When the blocks in either linked list are being accessed, we do not want other threads to access them simultaneously. Therefore, the memory management won't run into chaos and results in segfault.

The two linked lists version from previous homework can be described in short as follow. I used a full linked list to record all the blocks that I've allocated, and another "free linked list" to record the free blocks. When calling sbrk() for more memory blocks, it always adds to the end of the full list. When free function reset a block to be usable, it is added to the free list for further use. When the required size is smaller than the best fit block, I do split the block for better fragmentation ratio.

To make this implementation thread safe, I added two mutex locks. One is for the malloc and free function, and the other is for sbrk() system call. The mutex lock is acquired when the malloc function starts to allocate new space or to find space from the free list, and is released when the block is found and ready to be returned.

2. ts_malloc_nolock()

This function seems to be tricky, and my solution might be naïve and intuitive. Since we need to prevent race conditions and multiple threads accessing the same region of memory simultaneously, an easy idea is to separate each thread from the others and give them their own scope of memory accessibility. In other words, we create two virtual linked lists for each thread, and make them invisible to other threads. Although we still use a linear allocated memory region by sbrk() call, we actually have multiple linked lists that work just as a single threaded implementation.

The only problem is that - how can we create a pair of linked lists for each thread? Luckily, with the hint provided in the instructions, I got some basic knowledge of Thread-Local Storage variables. In short, they are mutual excluded between threads, meaning that for each thread created, it can only see its own TLS variable. Therefore, I created 3 TLS variables corresponding with my previous implementation, which are

```
1.  __thread metadata_block* head_TLS = NULL;
2.  __thread metadata_block* tail_TLS = NULL;
3.  __thread metadata_block* free_head_TLS = NULL;
```

In the no lock version, I just use them instead of the global variables for lock version. Some shortcomings come with this implementation, and I'll discuss them later in this document.

Part 2 Performance and results

The lock version passed all given test cases without error.
The no lock version did come with some struggles. See in next part.

	Lock version		No lock version	
	Runtime/s	fragmentation	Runtime/s	fragmentation
1	0.233799	45023776	0.137799	47512928
2	0.129331	45717696	0.156365	47054976
3	0.208080	47135712	0.181348	47748928
4	0.235957	46282432	0.140523	47335168
5	0.158405	45346560	0.175030	47054560
Average	0.193114	45901235	0.158213	47341312

Part 3 Discussion

1. Shortcomings for data structure

This data structure inherited from last assignment has some flaws. First, it need to maintain 3 pointers for its linked lists rather than just 1. Second, the full list consumes a lot of space, though we do not require to recycle the blocks allocated by sbrk() call. Third, this structure introduce some difficulties for splitting and merging blocks in a multithreaded fashion.

As for the merge function, due to the design of separated lists, we cannot merge two blocks with neighboring addresses as long as they are not within the same thread. Therefore, the full list become less useful to maintain. Maybe later on we can learn some better ideas on how to improve this implementation.

2. Performance analysis

As expected, the no lock version runs faster than the lock version. It's easy to interpret, as the lock version is actually just "single-threaded". Because we did not allow multithreads to access the memory region at the same time, threads can only standby and access one by one with the control of mutex locks. In this way, the fastest runtime would be more or equal to the single thread version in HW1. And with the time consumed by acquiring the status of mutex lock, it'll always be slower than the single threaded version.

The no lock version, on the other hand, runs faster. This is because multiple threads are running in parallel without blocking each other. The only block comes from the sbrk() call, which is identical to the lock version. Again, if creating and looping through linked list is faster than waiting for locks and run one by one, then no lock version will always be faster.

The fragmentation result tells some other facts. No lock version has a slightly larger fragmentation than lock version. Since the lock version only have one universal full linked list and one free list, it can manage all the blocks in a more effective way, as adjacent blocks can be merged and space used more efficiently. In contrast, no lock version has to

maintain 2 linked list for each thread, and among each thread the blocks might not be merged and split effectively as there are less blocks on each list. Therefore, the no lock version would result in a higher fragmentation size than its peer.

3. Other thoughts

- Refactoring code?

At first I thought about refactoring my code, since the 2 linked lists implementation seems not to be concise and easy to understand as the 1 free list implementation suggested by others. However, part of being lazy and part of thinking that changing it to 1 free list won't improve overall performance, I did not refactor my code.

- When and where to initialize the mutex lock?

Most sample code about multithreading I've seen initializes the lock in main function. Since my code does not have the main function to do that, I just create the lock as a global variable. Also, we might initialize that in the header file.

- Either using RWLOCK or MUTEX LOCK?

According to some reference, rwlock has worse performance than mutex lock. But to my expectation, properly written rwlock should be faster than mutex, since we can read simultaneously without exclude other threads. Investigate later.

- Thread-Local Storage

What is TLS variable? To keep it simple as I've described above, it's just like a replica for each thread created automatically.