Project #2: Thread-Safe Malloc

Name: Aiden Miao

NetID: km480

1. Implementation

In this assignment, I'm required to implement the lock and nolock version of thread-safe malloc () and free () basing on the best fit allocation policy.

Before we proceed, I want to mention that I've made some changes to my previous homework.

I didn't get to pass all the test on homework 1, and based on TA's response, some of my malloced space has been overwritten. I double checked my code, but I couldn't find where it went wrong. I'm guessing it's because of my design it's too complicated since I used 2 doubly linked lists and 4 pointers when implementing and it's almost impossible to find where it went wrong. So, I redesigned my code, instead of using too many lists, I only used one free list and it's singly linked. After implementing it again, I'm able to pass all the test at TA's. And I base my project 2 on this new version of malloc.

1.1 Lock version

The idea of lock version is very direct: we put the mutex lock around our critical section to prevent other thread from accessing it. So that we can avoid incorrect result and even segmentation fault.

In here, to ensure the thread safety of my malloc and free, I add mutex locks at 2 places. One is for the malloc function, one is for the free function. The first mutex lock is acquired when we malloc space and released after we finish allocating new space. The second mutex lock is acquired when we free space and add space to freelist, merge space, then it's released.

Overall, the locked version of malloc and free is very intuitive. We just need to make sure we lock before every critical section and release it afterwards.

1.2 No_Lock version

In no lock version, we are only allowed to lock when using sbrk function, other than that we cannot use lock. In order to solve this problem, basically we need to create a

dependent free list for each thread. According to the hint in instruction, I did some research about the thread-local storage. To all the thread, they can only see their own TLS variable. So, in order to implement the no lock version, we add thread-local storage variable for our global variable:

```
__thread block * tls_fhead;
__thread int tls_isempty = 1;
```

In here, we add 2 tls variables, one is the head of our free list, the other is the variable isempty used to check if our list is empty. Also, I write a TLS version of all the helper function like tls_add_freelist, tls_rmv_freelist, in which the variable they use are all TLS variable. Also, my TLS version of malloc and free all use these TLS functions. So basically, I have 2 versions of all my function. One is the normal, the other is the TLS version.

2. Performance results & analysis

- Results:

	LOCK		NO_LOCK	
	Time	fragmentation	Time	fragmentation
Test1	0.95	41755600	0.18	42111856
Test2	0.36	41562736	0.13	41562016
Test3	0.54	41569072	0.13	41565328
Test4	0.49	41584256	0.49	41857888
Test5	0.61	42006384	0.16	42135008
MAX	0.95	42006384	0.49	42135008

MIN	0.36	41562736	0.13	41562016
Avg	0.59	41698646	0.21	41846419

- Analysis:
    - Runtime: According to the result, we can see that the NO_LOCK version is much faster than the LOCK version. This result is correct because in LOCK version, whenever a thread is trying to malloc or free, it will lock this function and other threads cannot operate in parallel. Also, since all of the threads are using one long free list, it will take more time for them to search through the list. So actually, the LOCK version is more like serialization instead of parallelization. The NO_LOCK version, on the other hand, only locks when sbrk () new space, and all the threads have their own local free list so it will take them much less time to go through.
    - Fragmentation: The fragmentation of NO_LOCK is slightly bigger than the LOCK version, my guess is that since in NO_LOCK, all threads operate on their own free list, so the free block between each thread may not be merged efficiently. LOCK version, on the other hand, only need to maintain one long free list for all threads, so it can manage the space in a more efficient way.

3. Summary:

Overall, the NO_LOCK version has better performance at run time, which shows that the main factor of run time here it's the operation on our free list. In respect of segmentation size, the difference between these 2 methods is actually trivia. As I talked before, LOCK version may have more efficient use of the space since it only maintains one free list. So, if we are operating on a big amount data, I would recommend using NO_LOCK version. But if the data amount is not huge, and we have no strict requirement on runtime, for the sake of code simplicity, we can use LOCK version.