Report

- ECE 650 project 2 (Implementation of thread-safe malloc library)
- Name: Rui Cao, netID: rc384

1. My Implementation

In this project, I implement two different thread-safe versions (i.e. safe for concurrent access by different threads of a process) of the malloc() and free() functions besed on based on my code of project 1. Besides, both of my thread-safe malloc and free functions use the best fit allocation policy. We know that concurrent execution means that multiple threads will call the malloc and free functions, and thus may be concurrently reading and updating any shared data structures used by the malloc routines (e.g. free list information). Here are two solutions: lock-based synchronization and not useing locks.

1.1. Locking Version:

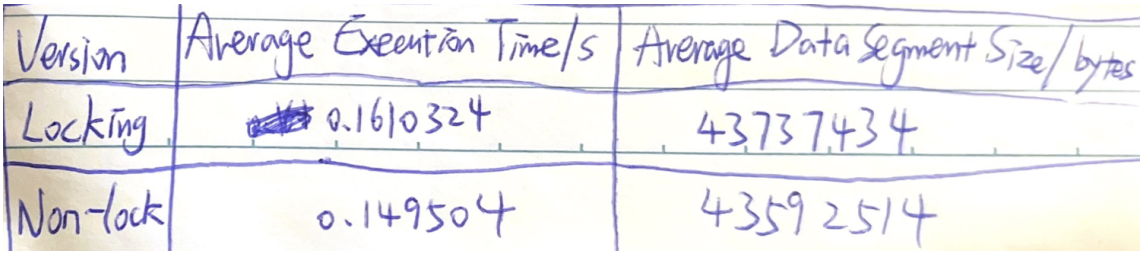
For the locking version, I use pthread_mutex_lock and pthread_mutex_unlock functions to prevent race conditions. I acquire a lock immediately before calling the original malloc and free function and release a lock immediately after calling them, which guarantees that only one thread is making changes to the free list and avoids other threads to change the free list. After one thread finishes, the lock is released and another thread reacquires the lock. The critical section is the section between locking (pthread_mutex_lock function) and releasing the lock (pthread_mutex_unlock function).

1.2. Non-locking Version:

For the non-locking version, I use Thread Local Storage which allows each thread to have its own head and tail (of free list) variables. So each thread has its own free blocks linked list and there is no free blocks linked list shared. Besides, because the sbrk function is not thread-safe, I acquire a lock immediately before calling sbrk and release a lock immediately after calling sbrk.

2. 3. Results & Analysis of Experiments

Because the test case contains some randomized behavior, so I run the test with each version 20 times to identify the typical behavior. The results of my experiments are as follows. For both versions, I took the average of 20 runs.



Version	Average Execution Time/s	Average Data Segment Size/bytes
Locking	0.1610324	43737434
Non-lock	0.149504	43592514

- Execution time results show that the non-locking version executes faster than the locking version. I think the reason for this result is that the non-locking version only locks when sbrk is called, while other runtimes (most of the

time), multiple threads run simultaneously. On the contrary, all threads of the locking version share a free list, and the program performs locking operation while the entire original malloc/free function is running, which means that multiple threads cannot run at the same time. Because non-locking version allows most of the code to run simultaneously, it runs faster than the locking version.

- The results of data segment size show that there is no significant difference in allocation efficiency between the two versions. This shows that the two versions have no significant difference in the number of times they can find free blocks for reuse and call the sbrk function.
- In general, based on the experimental results, if the execution speed is more pursued, the non-locking version is better. However, the non-locking version has no obvious improvement in allocation efficiency compared to the locking version.