Report 2: Thread-Safe Malloc

Student name: Shujie Yang

Student netid: sy289

- Implementations

This assignment, it is required to implement two versions of Thread-Safe best fit malloc/free functions. Specifically, one should use locks, the other one does not. For the lock version, the idea is basically firstly lock the mutex when malloc, then unlock the mutex before the malloc returns. For the free function, it also first locks the mutex and then unlocks it at the end. The purpose of using locks is to prevent race conditions in a multi-threading environment, where simultaneous access and modification of shared data by multiple threads can lead to data inconsistencies or unpredictable behavior. By ensuring that only one thread can operate on shared resources at any given time, mutex locks help maintain the consistency and integrity of the data.

For the nolock version, there is one exception. When using sbrk, in order to ensure the concurrency, we need to add lock and unlock before and after the function. Then, the idea of this version is to create a free list for every thread independently, so that the code will be executed sequentially for each thread. In this way can the concurrency be ensured.

- Result & Analysis

Here is a table comparing the performance of implementations with and without locks in terms of execution time and data segment size:

Scenario	Average Execution Time (s)	Average Data Segment Size (b)
lock	0.107541	44072960
noLock	0.110258	43852896

Based on the test results, we can see that the average execution time of the no locks version is slightly longer than the one with locks, and the average data segment size is smaller than that of it. The slightly longer execution time of the lock-free version may be due to the fact that each thread maintains its own free list, which avoids the overhead of locks but may add complexity to managing multiple free lists. Each time a malloc or free operation is performed, the thread may need to perform more search and management operations in its own free list.

On the other hand, versions that use locks may have less search and management overhead because of a global free list, but this typically increases lock contention,

especially in environments where memory allocation and freeing is used intensively by multiple threads. However, in this test, the version using locks has a shorter execution time, which may mean that lock contention is not a significant issue.

The smaller average data segment size of the nolock version may be due to the ability of each thread's free list to match and reuse memory more efficiently. This reduces memory fragmentation because each thread may optimize its own memory usage, thus reducing the overall memory footprint.

The slightly larger data segment for the version that uses locks may be because global free lists are not managed as efficiently as thread-localized free lists, which may result in more memory fragmentation.