Thread-safe Malloc Implementation Report

I. Background

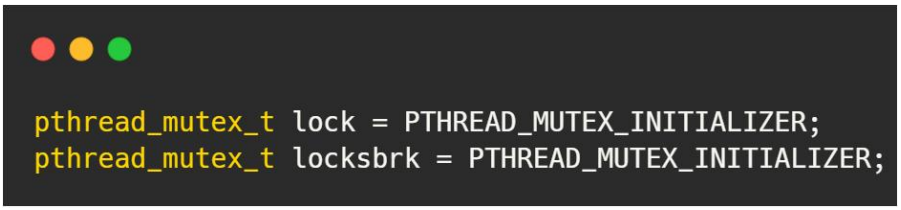
In this assignment, two different thread-safe versions of the malloc() and free() functions must be implemented. The only allocation policy used in this assignment is the best-fit allocation.

The first thread-safe version is required to use locks in the pthread library, and no locks should be used in the second thread-safe version, except for the sbrk function. Since the sbrk function is not thread-safe, it requires a lock before calling the sbrk(), and it is unlocked immediately after finishing the call.

II. Implementation

i. The First Version with Locks

Two mutexes are utilized to perform the malloc function and free function in the thread-safe version. The first one 'lock' is used to lock the malloc() and free(), and the second one is implemented on the sbrk function.



```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t locksbrk = PTHREAD_MUTEX_INITIALIZER;
```

As for the 'lock', it is added at the beginning of the malloc() and the free(), and it is unlocked before the function ended. We expect a mutex here because if there are multiple threads, each of them may modify the data we allocated on the heap and lead to errors or exceptions that we can not obtain the data we want.
As for the 'locksbrk', it is used only for keeping the sbrk() thread-safe.

ii. The Second Version with No Lock

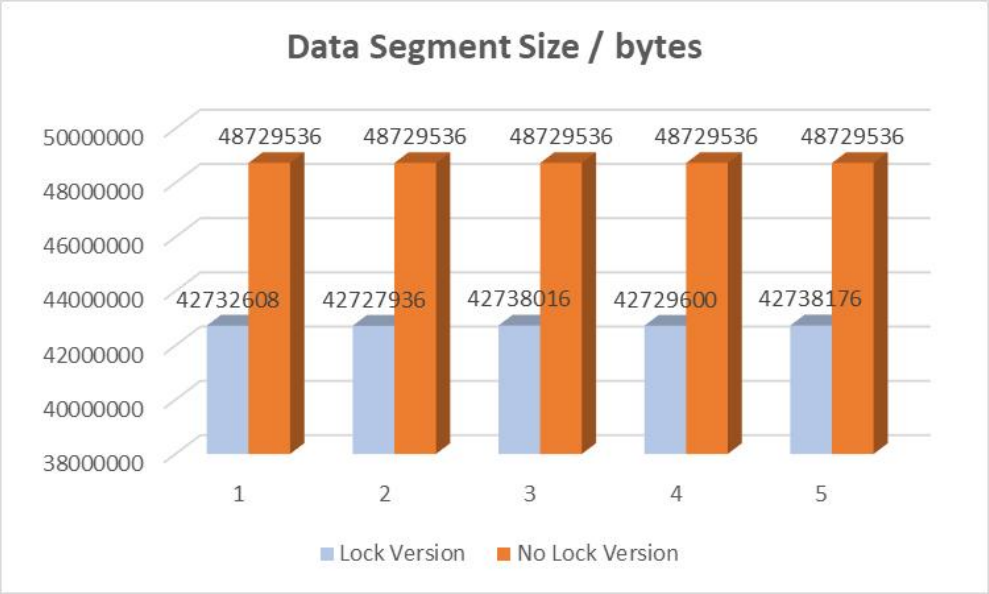
In this version, the thread local variable is introduced by assigning the '_Thread_local' keyword. The thread storage class marks a static variable as having thread-local storage duration. In this way, each thread creates a unique instance of the variable with the '_Thread_local' keyword and this variable will be destroyed when the thread terminates. Using such a variable can provide a convenient way to ensure thread safety without avoiding the need for thread synchronization.

```
_Thread_local mData* ts_basePtr =NULL;
```

III. Results

The execution time of the lock version and the no-lock version is shown below.

The data segment size of the lock version and the no-lock version is illustrated as follow:



IV. Analysis

The reason why the version using locks takes more time is that each thread needs to lock its critical region at the beginning. However, there is no available lock when if another thread does not unlock the mutex. Therefore, other threads need to wait for the available thread and compared with the no-lock version, it does not need to wait for others since each thread has its own copies of the variable. Consequently, the locked version takes a longer execution time. According to the data-segment size of the two versions, the version with locks takes a small data-segment size since each thread in the no-lock version maintains its own linked list while the version with locks does not.