1. Thread-safe Malloc Implementation

In my malloc implementation, I use a doubly linked list to store all the free blocks. And all free blocks in list are sorted by the address of them. In this project, I use both lock way and no lock way to make the malloc function thread safe.

- Lock Version

For the lock version, I use pthread_mutex_lock(&lock) and pthread_mutex_unlock() to implement. I add lock and unlock operation before and after the malloc and free function. As the below function shows:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
void * ts_malloc_lock(size_t size) {
    pthread_mutex_lock(&lock);
    void * ans = bf_malloc(size, &freeList, 1);
    pthread_mutex_unlock(&lock);
    return ans;
    }
void ts_free_lock(void * ptr) {
    thread_mutex_lock(&lock);
    real_free((node *)(ptr - meta), &freeList);
    pthread_mutex_unlock(&lock);
}
```

- Nolock Version

For non-lock version, I used Thread-Local Storage to ensure that thread has its own linked list. Keyword __thread enables that each thread has its own list. Besides, the sbrk() is still locked in critical section, for the function is not thread safe. The code is as below:

```
__thread list tls_freeList = {NULL, NULL};
…
pthread_mutex_lock(&lock);
newSpace = (node *)sbrk(size + meta);
pthread_mutex_unlock(&lock);
…
```

2. Performance Analysis

Executation Time (s)

Version	Average Execution Time/s	Average Data Segment Size/bytes
Lock	0.88	42799648
Non-Lock	0.29	42901760

From the results of execution time, we see that non-lock version code run faster than lock version code. This is because non-lock version code only put sbrk() into lock section while lock version code put whole malloc/free code into lock section.

For the data segment size, these two ways are similar to each other, which shows that they reuse a similar ratio of allocated space.