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);

…

1. **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.