Report of Thread-Safe Malloc

Name: Yue Yang | NetID: yy258

In this project, I implemented thread-safe versions of the malloc() and free() functions.

1 Overview of Implementation

To implement our thread-safe malloc() and free() functions which allow concurrency, we need to pay attention to race condition. Race condition is the result of computation by concurrent threads which depends on precise timing of an instruction's execution sequence. Because of race condition, we sometimes get incorrect or non-deterministic results. To avoid that, we need to control possible execution interleaving of threads.

In my code, I use LinkedList to track free blocks and LinkedList is what may cause race condition. To avoid that, we need to ensure that different thread has its own LinkedList and can only edit its own LinkedList.

1.1 Idea on Lock Version

For lock version, I use pthread_mutex_lock(&lock) and pthread_mutex_unlock() to avoid race condition. I add lock and unlock operation before and after the original malloc/free function. The code is as below:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
void * ts_malloc_lock(size_t size) {
    pthread_mutex_lock(&lock);
    int sbrk_lock = 0;
    void * p = bf_malloc(size, &first_free_lock, &last_free_lock, sbrk_lock);
    pthread_mutex_unlock(&lock);
    return p;
}
void ts_free_lock(void * ptr) {
    pthread_mutex_lock(&lock);
    bf_free(ptr, &first_free_lock, &last_free_lock);
    pthread_mutex_unlock(&lock);
}
```

The section between pthread_mutex_lock and pthread_mutex_unlock is called critical section. Lock operation enforces that only one thread at a time in the critical section. The lock version code allows concurrency in malloc and free level.

1.2 Idea on Nolock Version

For non-lock version, I used Thread-Local Storage to ensure that every thread has its own

LinkedList. _thread enables that each thread has its own head pointer and tail pointer of LinkedList. Also, since sbrk is not thread-safe, I add lock before calling sbrk and release lock after calling sbrk. The lock version code allows concurrency in sbrk level. The code is as below:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
__thread Metadata * first_free_nolock = NULL;
__thread Metadata * last_free_nolock = NULL;

void * ts_malloc_nolock(size_t size) {
    int sbrk_lock = 1;
    void * p = bf_malloc(size, &first_free_nolock, &last_free_nolock, sbrk_lock);
    return p;
}
void ts_free_nolock(void * ptr) {
    bf_free(ptr, &first_free_nolock, &last_free_nolock);
}
……
pthread_mutex_lock(&lock);
new_block = sbrk(size + sizeof(Metadata));
pthread_mutex_unlock(&lock);
……
```

2 Results and Analysis

My results of performance experiments(using default value) are as below:

Version	Average Execution Time /s	Average Data Segment Size /bytes
Lock	0.08	43038784
Non-Lock	0.06	43203840

From the results of execution time, we can see that non-lock version code run faster than lock version code. This is because non-lock version code only put sbrk into lock operation while lock version code put whole original malloc/free code into lock operation. Since non-lock version code allows more code to run simultaneously outside the lock operation, it will run faster.

From the results of data segment size, we can see that the two version codes have similar data segment size. This shows that the two version of code work similarly in finding free blocks that can be reused and calling sbrk.