Project Report #2: Thread-Safe Malloc

Name: Xushan Qing NetID: xq47

1. Thread-safe model

1.1 Goal

For this assignment, the goal is to implement two different thread-safe versions of the malloc() and free() functions.
In version 1, I use lock-based synchronization to prevent race conditions that would lead to incorrect results: *void *ts_malloc_lock(size_t size);void ts_free_lock(void *ptr);*. In version 2, I not use locks with one exception. Because the *sbrk* function is not thread-safe, I acquire a lock immediately before calling *sbrk* and release a lock immediately after calling *sbrk, void *ts_malloc_nolock(size_t size);void ts_free_nolock(void *ptr);*.
I also use support from pthread library, needed sysynchronization primitives (*pthread_mutex_t*). and Thread-Local Storage*(__thread*).

1.2 Model for lock version

To distinguish the lock and non-lock version, I use an integer, flag, and two different linked list with different head. When flag equals to 0, it is the lock version, when flag equals to 1, it is the non-lock version.

```
block_h * head_lock = NULL;
__thread block_h * head_nonlock = NULL;
```

For both version, we need to initialize a lock at the beginning as follows.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

For lock version, every time I call *malloc* or *free* function, I lock the corresponding part to prevent race conditions (*pthread_mutex_lock(&lock), pthread_mutex_unlock(&lock)*). The section between lock and unlock, which is the *bf_malloc* and *bf_free* function, is called critical sections. Every time one thread call malloc or free, the section would be locked, allows only one thread can access at the same time. Other thread needs to wait until the lock has been released. The thread can run simultaneously, allowing concurrency outside of malloc and free. Other part of code all allows concurrency. Here is what the model is like, and everything else is the same as what I have done in project1 for best-fit strategy:

```
block_h * head_lock = NULL;

void *ts_malloc_lock(size_t size){
        pthread_mutex_lock(&lock);
```

```
        void * p = bf_malloc(size, &head_lock, 0);
        pthread_mutex_unlock(&lock);
        return p;
}

void ts_free_lock(void *ptr){
        pthread_mutex_lock(&lock);
        bf_free(ptr, &head_lock);
        pthread_mutex_unlock(&lock);
}
```

1.3 Model for non-lock version

For non-lock version, I only lock the *sbrk* function because the *sbrk* functions is not thread safe to prevent race conditions (*pthread_mutex_lock(&lock), pthread_mutex_unlock(&lock)*). The thread can run simultaneously, allowing concurrency outside of sbrk. The section between lock and unlock, which is the *sbrk* function, is called critical sections. corresponding Here is what the model is like, and everything else is the same as what I have done in project1 for best-fit strategy:

```
__thread block_h * head_nonlock = NULL;

void *ts_malloc_nolock(size_t size){
   void * p = bf_malloc(size, &head_nonlock, 1);
   return p;
}
void ts_free_nolock(void *ptr){
   bf_free(ptr, &head_nonlock);
}
void *allocate_new_block(size_t size, int flag){
   ……
    if (flag == 0){
//Thread Safe malloc/free: locking version
      head_block = sbrk(size + sizeof(block_h));
   }
   else{
//Thread Safe malloc/free: non-locking version
      pthread_mutex_lock(&lock);
      head_block = sbrk(size + sizeof(block_h));
      pthread_mutex_unlock(&lock);
   }
   ……
}
```

2. Performance Results

nolock:

```
vcm@vcm-24033:~/ece650proj2/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.137874 seconds
Data Segment Size = 44509352 bytes
vcm@vcm-24033:~/ece650proj2/thread_tests$ ./thread_test
No overlapping allocated regions found!
Test passed
vcm@vcm-24033:~/ece650proj2/thread_tests$ ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed
vcm@vcm-24033:~/ece650proj2/thread_tests$ ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed
```

lock:

```
vcm@vcm-24033:~/ece650proj2/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.151762 seconds
Data Segment Size = 42145568 bytes
vcm@vcm-24033:~/ece650proj2/thread_tests$ ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed
vcm@vcm-24033:~/ece650proj2/thread_tests$ ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed
vcm@vcm-24033:~/ece650proj2/thread_tests$ ./thread_test
No overlapping allocated regions found!
Test passed
```

	Execution time	Data segment size
Nonlock method	0.138s	44509352bytes
Lock method	0.152s	42145568bytes

3. Comparison and Analysis

In the *thread_test_measurement* test, we can see that for the execution time, non-lock method is faster than the lock method. The reason is that the non-lock method only lock the sbrk function, whose range is smaller than the lock method's range which lock both the malloc and free function. This means that threads can run simultaneously in a larger range for non-lock method, which is more time efficiently.

For the data segment size, non-lock method needs more space to allocate than the lock method. The reason is that for non-lock version, each thread maintains its own free list independently, so the adjacent memory blocks in different threads cannot merge, and cannot be reused as a larger block, which cause more memory. However, for lock version, threads can share one free list, which allows merging adjacent memory blocks and enhance the memory usage.
In summary, I prefer to use non-lock method, since in most cases in real world, time matters more than space.