Assignment2 Report: Thread Safe Malloc

Name: Yinchao Shi
NetID: ys322

1 Requirements and Summary of Development

1.1 Objectives and Requirements

For this assignment, the objectives is to implement two different thread-safe versions of the malloc() and free() functions. Both of the thread-safe malloc and free functions use the best fit allocation policy. In the first version, lock-based synchronization is used to prevent race condition. In the second version, lock is only used before sbrk function and released after it.

We are also required to conduct a performance study in which we tested the execution time and the allocation efficiency of two different versions. The tradeoffs are evaluated.

1.2 Organization of the Report

This assignment report consists of four main parts. In the first part, the overview of the assignment is introduced. Detailed design, implementation of the algorithm and tests for the code are presented in the second part. The third part is mainly about the performance study, in which data results are shown and analysis and conclusion are drawn. In the last part, I put forward my thoughts and conclusions, experiences and lessons from this assignment.

2 Design, Implementation and Test

2.1 Design

2.1.1 Lock-based Version

For this lock version, the goal is to acquire lock before the critical section and release it after the critical section. Suppose that there are multiple threads want to alloc the memory simultaneously, they call the bf_malloc function and may edit the free block list at the same time, leading to the use of overlapped memory. Therefore, I just lock the mutex before calling the malloc or free function and unlock it after. Then when one thread execute this function, others cannot acquire the lock.

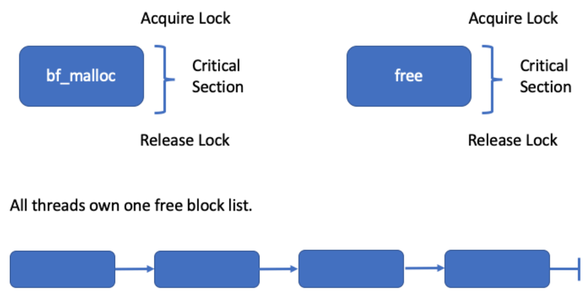

Figure 1: Schamatic Diagram for Lock Version

2.1.2 No-lock Version

For no-lock version, we should only acquire lock before the sbrk function. However, only having lock on getting new memory is not enough. Also suppose that there are multiple threads want to alloc the memory simultaneously, they possibly get the same block from the list, the memory they use will still be overlapped. If each thread can have their own free block list, the memory will not be overlapped, because each time getting new memory is not concurrent. Each thread can only edit their own free block list. Therefore, I am going to use Thread Local Storage, which is the method that each thread can allocate thread-specific data.

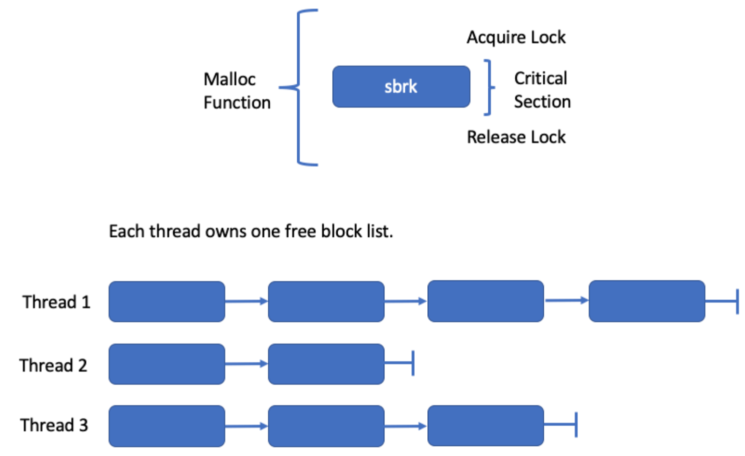

Figure 2: Schametic Diagram for No-Lock Version

2.2 Implementation

I mainly used the code from the assignment1 and had some improvement on it to have two thread-safe versions. Because I want to use thread local storage to allocate the head block for free block list, I declared another variable in header file.

tlsHeadBlock Variable is declared thread-local using the ___thread keyword.

```
    ___thread block * tlsHeadBlock = NULL;
```

Because the two versions are using different headBlock variables and getNewMemory functions, to decrease the duplication of the code, I passed these two variables into bf_malloc function.

```
    void * bf_malloc(size_t size, block ** head, void* (*func)(size_t));
```

2.2.1 Implementation of Lock Vesion

Always acquire the lock before malloc or free, and release the lock after the function.

```
        void *ts_malloc_lock(size_t size) {
          pthread_mutex_lock(&lock);
          void * bestBlock = bf_malloc(size, &headBlock, getNewMemory);
          pthread_mutex_unlock(&lock);
            return bestBlock;
        }

        void ts_free_lock(void * ptr) {
          pthread_mutex_lock(&lock);
          my_free(ptr, &headBlock);
          pthread_mutex_unlock(&lock);
          return;
        }
```

2.2.2 Implementation of No-lock Vesion

Acquire the lock before sbrk function, and release it after the sbrk. Use thread local storage to declare the tlsHeadBlock variable.

```
        void * tlsGetNewMemory(size_t size) {
          // compare the size with pagesize
          pthread_mutex_lock(&lock);
          void * new = sbrk(size + BLOCKSIZE);
          // sbrk failed
          if(new == (void*) - 1) {
            pthread_mutex_unlock(&lock);
            return NULL;
          }
          block * newBlock = new;
          newBlock->size = size;
          newBlock->next = NULL;
          pthread_mutex_unlock(&lock);
          return (void*)(newBlock + 1);
        }

        void *ts_malloc_nolock(size_t size) {
          void * bestBlock = bf_malloc(size, &tlsHeadBlock, tlsGetNewMemory);
          return bestBlock;
        }

        void ts_free_nolock(void *ptr) {
          my_free(ptr, &tlsHeadBlock);
          return;
        }
```

2.3 Test

For both lock version and no-lock version, the code has passed all the tests provided.

3 Performance Results and Analysis

3.1 Performance Results

Because the test is random, I have run the thread_test_measurement test for each version for 10 times, and calculated the average execution time and data segment size.

	Lock Version	No-lock Version
Execution Time	0.7064206 s	0.1784289 s
Data Segment Size	41680165 bytes	41699480 bytes

3.2 Analysis

3.2.1 Execution Time

From the results of execution time, we can see that no-lock version is much faster than the lock-version. The reason for the shorter execution time is that lock is only used when calling sbrk function, then most other parts for different threads can still run concurrently. On the contrary, most execution time of lock version is waiting for the lock to be released. In addition, it takes much longer time for the lock version to traverse in the only shared free block list.

3.2.2 Data Segment Size

From the results of data segment size, we cannot see obvious difference between two versions.

3.2.3 Summary

In conclusion, two versions of thread-safe malloc function mainly differ in the aspect of execution time. And I think no-lock version is better, because it has more concurrent parts, and each thread has its own free block list, leading to its smaller execution time and high efficiency.

4 Conclusion

From this assignment, I got a lot of thoughts and lessons.

At the first sight of this assignment, I thought it was easy. I just used malloc function in the last assignment, and added the lock between the codes. However, there were a bunch of same code in two versions. I found that these two versions had most same parts except for the sbrk function and the headBlock pointer. To decrease the duplication, I abstracted out those functions, and passed parameters that I wanted.

Secondly, I've learnt the concept of thread local storage, to allocate thread-specific Data.