Project #2: Thread-Safe Malloc Report
ECE 650 – Spring 2023
Yu Xiong (yx236)

1. Overview of Implementation of Thread-safe Model
   a. Simple Design Idea
      Based on the goals of this project we can see that we need to create two versions of the thread-safe malloc/free functions. Version 1 can use locks while version 2 cannot use locks except for sbrk function. For thread-safety reason, my initial thought was that version 1 would require bf_malloc and bf_free to be fully locked. As of version 2, it will only use locks if the code uses sbrk function. Then, I let the threads have their own linked list. So I kept single_node_t * free_head in project1 and created __thread single_node_t * nolock_free_head to work with both versions.

```
// global variables
single_node_t * free_head = NULL;
__thread single_node_t * nolock_free_head = NULL;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

   b. pthread_mutex_t lock
      First we use the concept of pthread_mutex_t. In the lecture, the pthread_mutex_t, or lock, is used to block race conditions. It prevents logical errors or failures of the program caused by multiple threads competing for resources. When a thread obtains a resource, the presence of a lock prevents other threads from using the resource. When the thread finishes using the resource, the lock releases this resource. The threads can now try to obtain this resource.
   c. single_node_t * free_head for version 1
      The struct single_node represents a single node of the double linked list. Each node contains the start address, the size of the actual data, the free state, and two pointers of the same struct type to the prev and next nodes. We'll use it in version 1. Both malloc and free are protected by locks, so all threads will share the same linked list free_head. Version 1 will manage the malloc and free functions directly with locks. For example, when a thread needs to use malloc, it gets the entire resource of malloc. No other thread can use malloc when it uses malloc. Our use of locks in this way will completely avoid the problem of resource contention.
   d. __thread single_node_t * nolock_free_head for version 2
      In version 2, we cannot use locks to protect resources. So in addition to using the double-linked list, we add __thread. __thread causes each thread to have its own linked list. In other words, their resources are not shared. Under the situation that we are not allowed to use lock, managing each thread's resources separately can avoid resource contention. Because of sbrk function's nature, sbrk function is the only function in version 2 that needs to be protected with locks. The rest of the time, threads interact with their own lists, such as using and free memory.
   e. Implementation

To achieve my goal, I added two new parameters to the related malloc and free functions. These are size_t lockdecision and single_node_t ** head. The size_t lockdecision will help us distinguish whether we are in version 1 or version 2. single_node_t ** head will help us use different versions of the list without having to distinguish between them. The implementation steps are also very clear. For version 1, we will add locks to ts_malloc_lock and ts_free_lock as a whole. As we pass data to subsequent functions, we add additional information to help subsequent functions operate. For version 2, we just need to lock the sbrk function.

f. Conclusion

We implement two versions of the thread-safe model through different ways of using locks.

2. Performance Result Presentation

For Locking Version:

```
yx236@vcm-30604:~/ece650/project2/homework2/thread_tests$ ./thread_test
No overlapping allocated regions found!
Test passed
yx236@vcm-30604:~/ece650/project2/homework2/thread_tests$ ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed
yx236@vcm-30604:~/ece650/project2/homework2/thread_tests$ ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed
yx236@vcm-30604:~/ece650/project2/homework2/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.222214 seconds
Data Segment Size = 43892368 bytes
```

For Non-locking Version:

```
yx236@vcm-30604:~/ece650/project2/homework2/thread_tests$ ./thread_test
No overlapping allocated regions found!
Test passed
yx236@vcm-30604:~/ece650/project2/homework2/thread_tests$ ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed
yx236@vcm-30604:~/ece650/project2/homework2/thread_tests$ ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed
yx236@vcm-30604:~/ece650/project2/homework2/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.175399 seconds
Data Segment Size = 44259736 bytes
```

So we get this graph:

	Locking Version	Non-locking Version
Execution Time	0.222214 seconds	0.175399 seconds
Data Segment Size	43892368 bytes	44259736 bytes

3. Comparison of Locking vs. Non-locking Version

According to the data we obtained and the graph we made, we could see that the performance of locking version and non-locking version was very different. First, let's focus on the Execution Time. We can see that locking version takes longer to execute than non-locking version. We know that in locking version, all threads share

the same linked list. For thread-safe reasons, we use locks for malloc and free functions of locking version to prevent resource contention. When a resource is used, other threads that also want to use that resource must wait. This results in an overall increase in time. The non-locking version of malloc and free does not have this problem. Since we don't have a lock on malloc and free functions as a whole, threads can work simultaneously. It saves time. Since they have separate linked lists, simultaneous work does not cause thread insecurity.

Second, let's look at the Data Segment Size. We can see that locking version uses less size than non-locking version. This is also the result of their different characteristics. For locking version, all threads share the same resource. This results in a more concentrated use and free of resources. It's easy to get a more complete memory after free. Under the non-locking version, all threads have their own linked list. This causes memory to become more fragmented, that is, segment size to increase as a result.

Locking version and non-locking version have their own advantages and disadvantages. We need to choose which one to use according to the actual situation. If the industry is focused on execution time, the non-locking version is a good choice. If the industry pays more attention to data segment size, then locking version can be better served.