ECE 650 Assignment 2: Thread-Safe Malloc

Yuchen Yang(yy227)

This report is a study of the implementation of thread-safe “malloc” function in C library. There are mainly three parts contained, function implementation, experimental results, comparison and analysis.

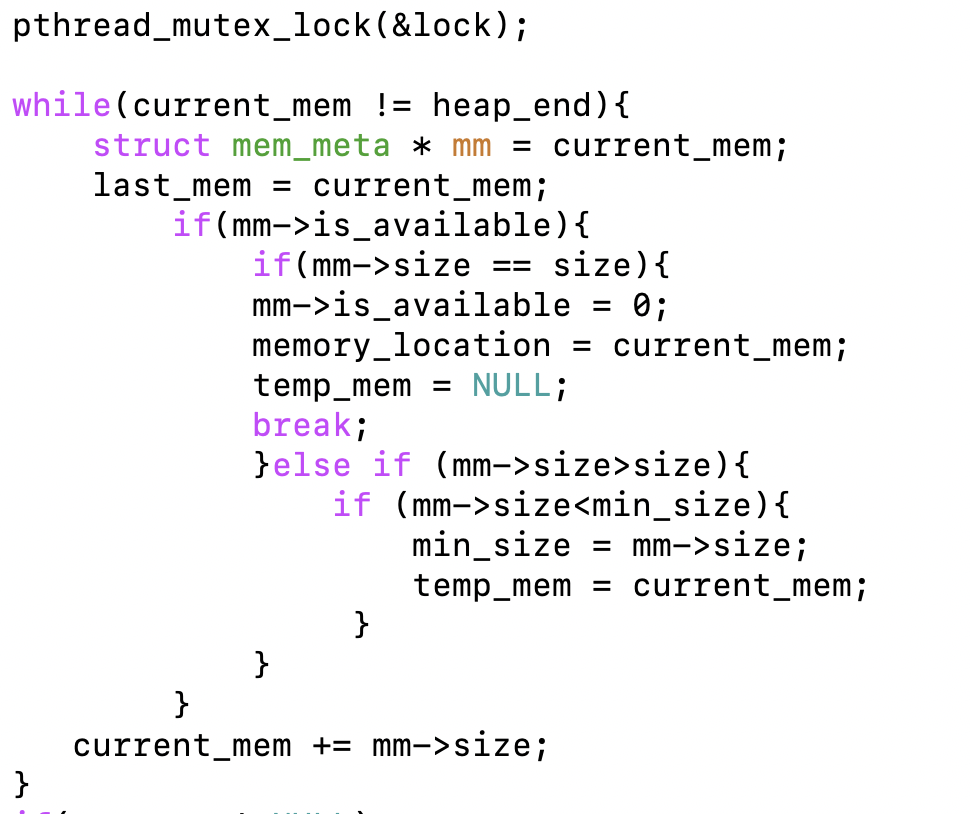
1. **Function Implementation**

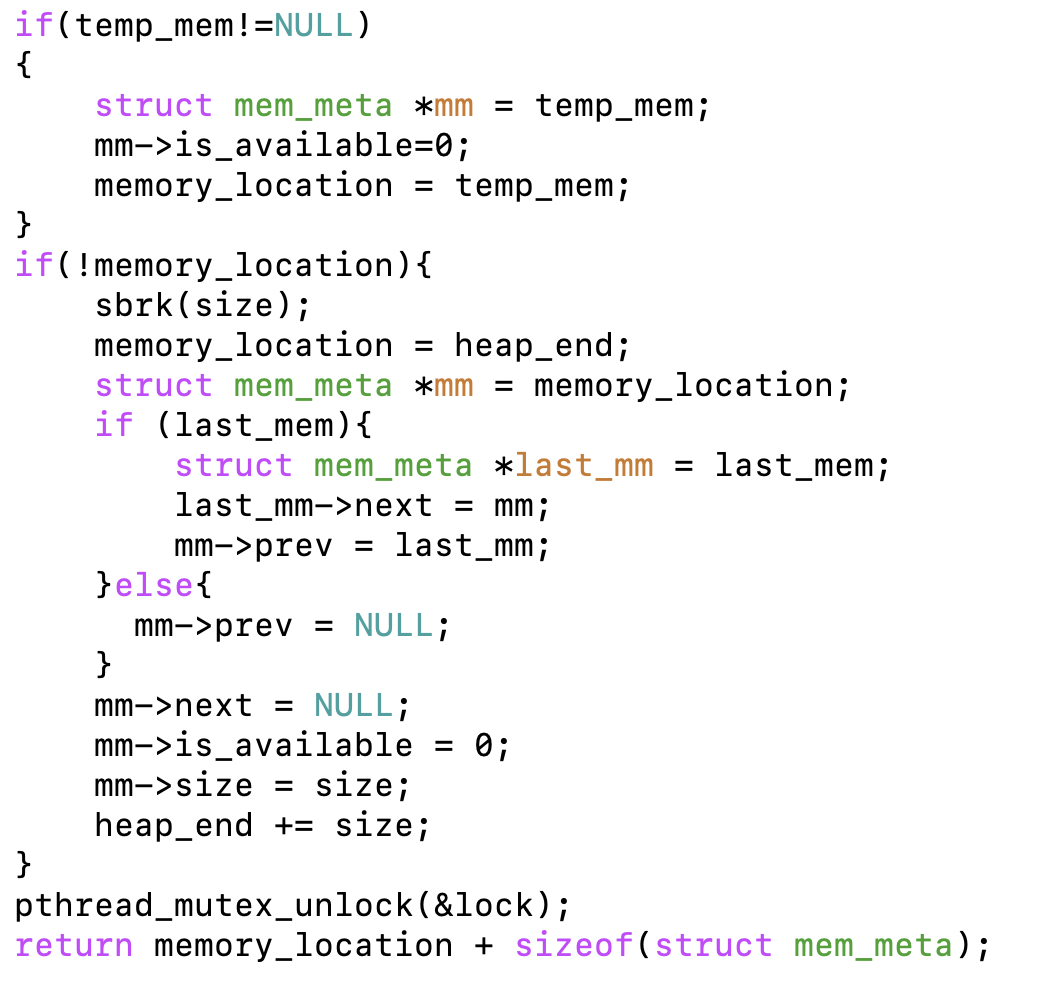
In real life, the malloc function is used in multiple threads in one process, and this leads to the race condition. There are multiple ways to deal with it, in this report, I used two of them to solve this problem.

1. Lock Version Malloc



Based on last assignment, the data structure I implemented is just a huge linked list. When all threads try to access the same space, I let the first arrived thread to take up the space and locked the rest, which is shown in the figure above. This can be achieved by adding a “mutex” to where the race condition happens, which is the trunk is available.

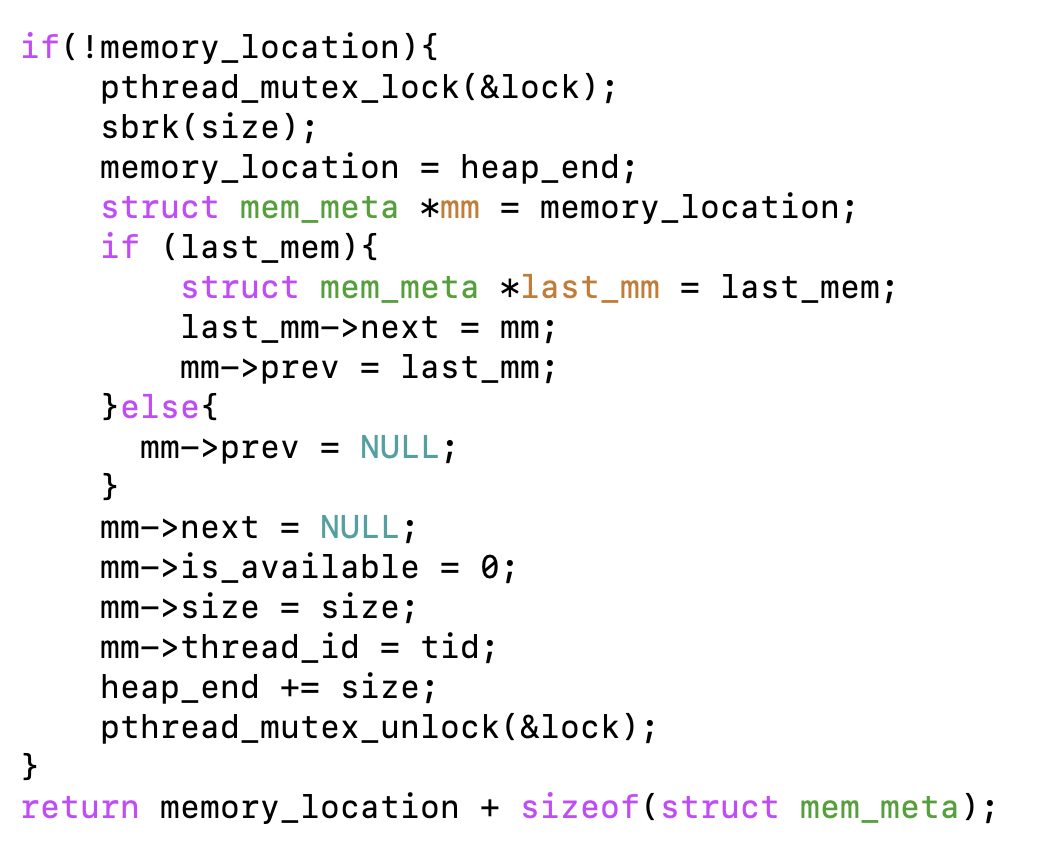




1. No-lock Version Malloc

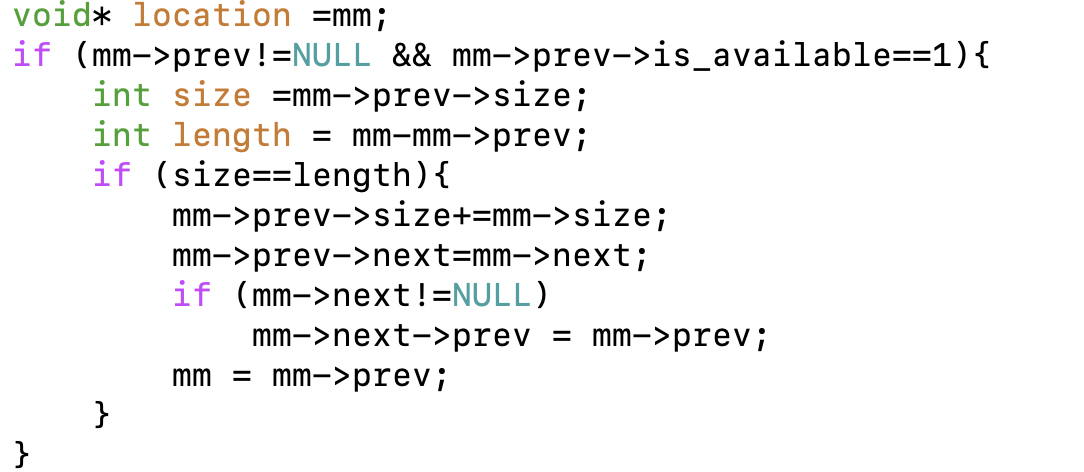


In this case, I separated the data by their thread id. The original linked list which everything was related now breaks into several lists grouped by thread id. When malloc and free, threads can only access the list with same thread id.



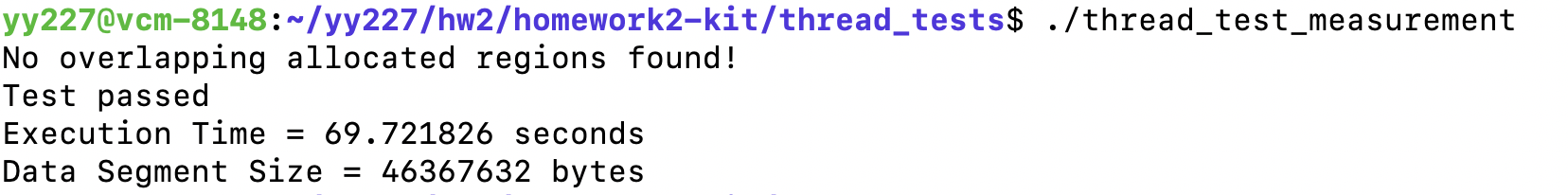
The reason why we have to keep the lock on sbrk() is that there is a non-avoidable situation when multiple threads run out of space and ask for new space.

Also note that when free, we need to merge the adjacent free chunks with the same thread id.

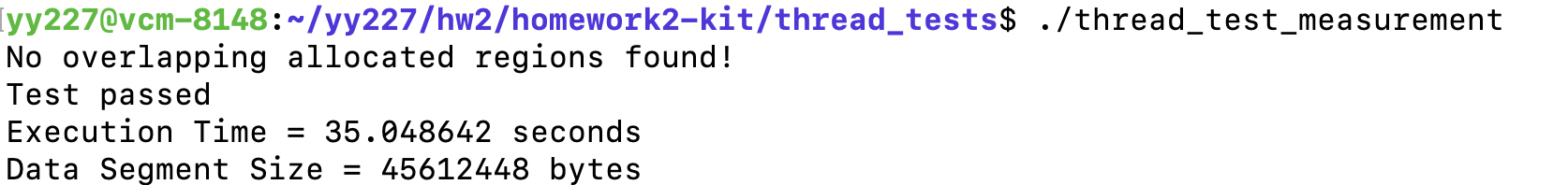


1. **Experimental Results**

**Result of lock version:**



**Result of no lock version:**



|  |  |  |
| --- | --- | --- |
|  | Execution Time | Data Segment Size |
| LOCK | 69.721826 s | 46367632 bytes |
| NOLOCK | 35.048642 s | 45612448 bytes |

1. **Analysis and Conclusion**
2. From the table above, it can be observed that the execution time of no\_lock\_version malloc is half the time of lock version. This is because when multiple threads trying to malloc the same space, thread\_lock blocks all others but one, and the waiting time is really expensive. Contrary to the lock version, in my no\_lock\_version malloc, it only searches for the linked list with the same thread id, and the execution time is saved.
3. In both malloc, the data segment size of no\_lock malloc is slightly smaller than the lock version. However, in theory, the lock version should have done better in space allocation because the no\_lock malloc can only allocate data in the linked list with same thread id while lock malloc does not have that constraint. Through digging into the test files, I found that one possible reason is that in the test file the process is “malloc-free”, which means that both version allocated data trunks and then freed all of them and did not go through the “malloc-free-malloc-free” process. In this case, the advantage of lock version cannot be observed easily.

To support my hypothesis, I went through the formula below:





The computation result is 46 080 000, which is the maximum value of possible segmentation size. Through examining the test code, the problem may lay in the lock inside test code. Due to this additional lock, the free processes of lock version malloc are blocked, and leading to a much larger segmentation size.