650 HW2 Thread-safe malloc report

Nibo Ying ny38 02/05/2019

1. Thread-safe malloc implementation

General:

Implemented two versions of multithread-safe malloc. Used pthred_mutex_lock for lock version and thread local storage for no-lock version. For lock version, used mutex lock to lock the critical section so that threads must wait in a line to access critical section one by one. For no-lock version, used thread local storage to create a unique freeList for each thread. This method ensures no race condition exists because each thread finds a fit block and links a freed block in its unique linked list.

Lock version:

Synchronization strategy:

Use pthread_mutex_lock and pthread_mutex_unlock to protect critical section.

Critical section:

findBlock() :

this is critical section because two threads might compete for the same block.

Dislink()

This is critical section because two threads might dislink two adjacent blocks , thus causing chaos.

Sbrk()

This is critical section because two threads may extend the heap at the same time and return one address to two malloc.

Link()

This is critical section because link() manipulates the linked list's head and tail. When two threads try to change head/tail at the same time, error occurs.

Conclusion:

For lock version, the fact that all threads manipulate a common linked list leads to the result that almost all functions have critical section.

Where concurrency is allowed:
All the code except Malloc() and free().

Unlock version:

Synchronization strategy:
Use thread local storage so that each thread has its own freeList head and tail. Since the linked list's head and tail are unique, the linked list is unique. Therefore, the operation of one thread would not affect the function of another thread. Because of this special feature, no critical section exists.

Critical section:
No critical section exists except sbrk().

Where concurrency is allowed:
All of the code except sbrk().

2. Result observations

Used this command:

```
For run in {1..50}; do ./thread_test_measurement ; done
```

To run two versions of test cases 50 times each. Average runtime and fragment size recorded in the table below

Runtime:

Version	Lock	No-lock	Result difference
Average time(s)	0.229208	0.161258	42.14%
Fragment size(byte)	46442761	44682120	3.94%

According to the result, lock version malloc has longer execution time than no-lock version. The average runtime differs by 42.14%. Apparently, no-lock version runs faster.

The data fragment size for two versions are almost the same. Before the experiment, I thought no-lock version malloc will waste a lot of space because it has several linked lists while lock version has only one linked list. The result contradicts my anticipation. This is because the test case code runs malloc for thousands of hundreds of times so that all linked lists in no-lock version have a good chance to be fully used. As a conclusion, when the number of items to malloc is very large, two versions of malloc have little difference in fragment size.

Combine the above observations, I recommend use no-lock version malloc because it runs faster and does not waste too much space.