A Custom Concurrent Malloc Library

Anran Chen (ac692)

Abstract

The aim of this study is to propose an implementation of C standard Malloc/Free library with the help of concurrency. The algorithm is built upon the previous version of the custom single-threaded Malloc/Free library, which is implementated using a modified doubly linked list data structure. The evaluations have shown this implementation a satisfactory replication of its C standart counterpart, matching in multithreading correctness, speed and data fragmentation.

Implementation Details

Two versions of the custom concurrent malloc library are implemented, namely the locking and non-locking. Both versions are based on the best-fit memory allocation policy from the previous project.

For the locking version, all threads operate on a shared, global free list. Mutex lock from the C standard `pthread` library is used to avoid race conditions. Before a malloc/free operation is requested, the free list is locked by a specific thread, and is prevented modification from other threads. And when the operation is over, the lock is released, ready to be acquired by another thread.

For the non-locking version, thread local storage(TLS) is leveraged, so each thread has its own free list. In this way, no mutex lock is require when the free list is modified, as there is not a shared list. However, the C standard `sbrk` function requires mutex lock when invoked, due to its thread-uncertain nature. Each list has two thread local variables, demarcated by keyword `__thread` indicating their heads and tails.

Performances

Experimentation

The library is implemented in C, and executed on a Windows 10 WSL with an Intel i7-7700HQ CPU and 8GB of RAM. Execution speed and data fragmentation are measured for both the locking and the non-locking versions. The average results of 20 iterations are taken to prevent performance fluctuation due to cold cache. The results are as follows in table 1.

	Speed (s)	Fragmentation (bytes)
Locking	0.245	48729536
Non-Locking	0.163	48729536

Table 1: Performance results

Speed

Both versions are able to finish execution in a reasonable time. The non-locking version slightly outperforms the locking version. It is expected that the overhead incurred from locking and unlocking mutexs by the locking version contributes to the time panelty significantly. The non-locking version does not require nutex locking when operating on the free list, but only during a `sbrk` request. However, `sbrk` is only called rarely when new segments are allocated, which does not have a prominent influence on its execution time.

Fragmentation

Both versions have the exact same fragmentation in bytes. This result, nevertheless, is not a true reflection of the algorithms' performance.

It is expected that the locking version would have a much lower fragmentation size, since that only a single free list exists, merging would happen more frequently resulting in a more optimized memory space.

On the contrary, the non-locking version sees each thread employing its own free list. Although two physically adjacent spaces are mergeable, they may belong to different threads and no subsequent merge can occur. As a result, it is expected to have a larger fragmentation size than the locking version.

Discussion and Conclusion

It is discovered that there is a tradeoff between speed and fragmentation for both versions of implementation. The locking version would preserve a more coherent memory space, but would suffer a significant mutex overhead. While the non-locking version surpasses the former in speed, but would not be ideal for memory integrity. It is recommended that the locking version should be used when memory coherence is prioritised, otherwise the non-locking version should be considered when speed is more crucial.

Possible Future Work

Although this implementation of the concurrent malloc libary has a major improvement in speed comparing to its single-threaded predecessor, certain issues could still be under future consideration.

For the locking version, due to the leverage of mutex, significant overhead is added to the execution time. This problem would be furthur aggravated when more threads are used. To eliminate the overhead while avoiding race conditions, atomic variables should be considered to implement the free list. It has two advantages over the current work: a great reduction in execution time, and a less bug-prone code base by taking away considerations from developers about where locks are needed.