ECE 650 – Spring 2025
Project 2: Thread-Safe Malloc

Jingheng Huan
Email: jh730@duke.edu

February 5, 2025

1 Introduction

Dynamic memory management is important in system performance and reliability. In Project 2, I extend my Project 1 to support multi-threaded environments by implementing two thread-safe versions of `malloc` and `free`. One version uses a lock-based synchronization scheme while the other minimizes locking by employing thread-local storage, with the only exception being a lock for the `sbrk()` call. Both implementations use a best-fit allocation strategy. The primary goals of this project are to ensure the correctness of memory allocation and deallocation in concurrent and to compare the allocation efficiency between a fully lock-based implementation and a mostly non-locking implementation.

2 Implementation Description

The implementation is divided into two versions. The locking version, consisting of the functions `ts_malloc_lock` and `ts_free_lock`, uses a global mutex to protect shared data structures during memory operations. In contrast, the non-locking version, which comprises `ts_malloc_nolock` and `ts_free_nolock`, employs thread-local free lists so that each thread manages its own free memory, thereby reducing contention. The only exception is the `sbrk()` call, which is protected by a mutex since it is not thread-safe.

In my implementation each allocated memory block is preceded by a metadata structure that stores the block's size, a flag indicating whether the block is free, and pointers to the next and previous free blocks in a doubly-linked list. The locking version utilizes a global free list maintained by pointers such as `global_first_free` and `global_last_free`, and these are protected by the mutex `global_lock`. Meanwhile, the non-locking version uses thread-local free lists managed via thread-local storage, which minimizes the need for locks during allocation and deallocation operations.

Both versions implement a best-fit allocation strategy. Initially, the free list is traversed to identify the smallest block that can satisfy the allocation request. If a suitable block is found, it is reused; if the block is larger than necessary, it is split into an allocated block and a smaller free block. In cases where no suitable block exists, the heap is extended using `sbrk()`. The locking version acquires a global mutex before any access or modification of the global free list, ensuring that all critical sections that update shared variables and manipulate the free list are protected. On the other hand, the non-locking version relies on thread-local storage for most operations, with only the `sbrk()` call being protected by a dedicated mutex.

The function `ts_malloc_lock` takes a size parameter and begins by locking the global mutex before searching the global free list using the best-fit strategy. If a free block is found, it is

reused—possibly splitting the block if needed; otherwise, a new block is allocated via `sbrk()`. Once the allocation is complete, the mutex is unlocked and the pointer to the allocated memory is returned. The corresponding function `ts_free_lock` locks the global mutex, marks the block as free, reinserts it into the global free list, and coalesces adjacent free blocks to reduce fragmentation before unlocking the mutex. In contrast, the `ts_malloc_nolock` function operates by searching the thread-local free list for a suitable free block and, if none is found, calling `sbrk()` (under lock) to extend the heap. Similarly, `ts_free_nolock` marks the block as free, reinserts it into the thread-local free list, and coalesces adjacent free blocks as needed.

3 Experimental Results

Several test programs were run to verify both the correctness and performance of the locking and non-locking implementations. In both versions, all tests confirmed that there were no overlapping allocated regions, indicating that the memory allocation and deallocation mechanisms are thread-safe and free from race conditions. The measurement test for the locking version reported an execution time of 0.129794 seconds and a data segment size of 44,786,560 bytes. The measurement test for the non-locking version reported an execution time of 0.143708 seconds and a data segment size of 42,927,712 bytes.

Table 1: Performance Comparison Between Locking and Non-locking Versions

Version	Execution Time (s)	Data Segment Size (bytes)
Locking	0.129794	44,786,560
Non-locking	0.143708	42,927,712

4 Analysis and Trade-offs

The locking version offers the advantage of simplicity in design and strong consistency since all critical sections are serialized using a global mutex. This ensures correctness, but the global lock can become a bottleneck in highly concurrent environments, leading to increased overhead as all threads must wait for the lock even when their operations do not conflict. In contrast, the non-locking version reduces contention by allowing each thread to manage its own free list through thread-local storage. This design significantly improves scalability under high thread concurrency and minimizes lock overhead. However, it also introduces additional complexity in managing multiple free lists and may lead to increased fragmentation if the coalescing of free blocks is not optimal.

5 Conclusion

In this project, two thread-safe memory allocators were successfully implemented. The lock-based version guarantees correctness through comprehensive locking but can become a performance bottleneck in highly concurrent environments. The non-locking version leverages thread-local storage to reduce contention and improve scalability, although this approach may lead to higher fragmentation if free blocks are not optimally coalesced. Future work may explore further optimization of the coalescing strategies in the non-locking version to reduce fragmentation while maintaining high performance in multi-threaded environments.