Section 1: Problem & Setup

This homework assignment involves the development and analysis of thread-safe malloc and free functions. Two versions of malloc and free are implemented in this assignment. One version of malloc and free is implemented with locks from the pthread library (pthread_mutex_t's) while the other is implemented without locks. In the non-locking version of malloc and free, there is one exception. Since the sbrk() system call is used to extend the heap segment for memory allocations, and it is not thread-safe, a single mutual exclusion lock is used around that sbrk() call.

The motivation for implementing these thread-safe C library functions is to gain a better understanding of memory management for processes running multiple threads and thread-safe functions. Both locking and non-locking versions of malloc and free are implemented with a best-fit policy. In each malloc implementation, previously freed blocks are split if they are larger than the needed allocation size. In each free implementation, adjacent free blocks are coalesced to form larger contiguous blocks. These features enforce a minimization policy of the processes heap segment.

Code for this assignment was written in C and developed in the Duke Linux environment at login.oit.duke.edu. Various tests are included in this homework assignment to verify the correctness of each malloc and free implementation. A performance analysis is also conducted. The performance analysis measures execution time and the size of a created data segment and is followed by a discussion of the results. The subsequent sections of this report are structured as follows: Section 2 details how each malloc and free function was implemented and describes any helper functions used; and Section 3 discusses the performance metrics of each malloc and free implementation and analyzes the results of benchmark testing.

Section 2: Implementation

Memory management data structure:

The data structure used for managing heap memory is a doubly-linked list. The nodes in the list are block_node structs which contain the following meta data: size (size of data payload plus the block_node struct), next (address of next block_node), and prev (address of the previous block_node):

```
typedef struct block_node_t {
        size_t size;
        struct block_node_t * next;
        struct block_node_t * prev;
} block_node;
```

A state variable for the blocks used/free status is not included. This is because the list consists of only free blocks. Used block addresses are returned by malloc and are the argument to free calls, so there is no need to track used blocks (that falls on the user). Tracking only free blocks greatly improves performance as it reduces the number of blocks searched on a malloc call.

The doubly linked list of block_nodes is sorted by ascending address. This ensures that the next and previous pointers in a free block's block_node struct point to the closest neighboring free blocks in memory. This makes for easy free block coalescing, since blocks can compare their next and previous pointers to their adjacent memory blocks. For example, if you have a block_node * current_block, and [(block_node *)((char *)current_block + current_block->size)] == [current_block->next] evaluates to true, then you know the current block and the next free block are adjacent in memory. These blocks are then coalesced to form one larger, contiguous block.

Malloc & Free: Locking version

The locking version of the thread-safe malloc function (ts_malloc_lock) was implemented using mutual exclusion locks (pthread_mutex_t's) from the pthreads library. These locks are placed within the code such that at any given time only a single thread can perform one of two actions. A single thread may either select and remove a block from the free list (the block may be split in the process) or add a block to the free list and attempt to coalesce that block. These operations must be serialized as they modify the linked list data structure used for tracking free blocks which is shared by all threads. The locking version of the thread-safe malloc function was implemented with the help of several helper functions. These remain largely unchanged from the previous assignment:

block_node * try_block_reuse_bf(size_t size);

- This function searches the entire list of free blocks and returns a block_node * to a block whose size is greater than the size argument of the function and has the smallest difference in size relative to that size argument. For example, if the free list consists of blocks with sizes 16, 8, 24, 12, 32, and a request is made for a size totaling in 10 bytes, then try_block_reuse_bf would return a pointer to the block_node with size 12, as 12-10 = 2 is the smallest difference.
- If a block is found for reuse, the attempt_split function is called to attempt to split the block, and a pointer to that block is returned. If no block is found for reuse, NULL is returned.

void attempt_split(block_node * to_split, size_t size_needed);

- This function splits the block found for re-use (for an allocation request of size_needed bytes) and creates a new block_node from the remaining size (to_split->size – size_needed) if the remaining size of the block is greater than or equal to a MIN_SIZE constant defined in my_malloc.c
- The MIN_SIZE constant can be tuned based on the workload for malloc but must be at least the size of the block_node struct (i.e. 24 bytes) to avoid error (otherwise the

attempt_split function could create a new block_node in a memory location which could over-write another block or data). Increasing the MIN_SIZE parameter will also increase the over-all size of the running process's heap. This is because the size blocks must be in order to split increases, and blocks with larger sizes may be selected for smaller allocations, resulting in unused space. Complete minimization of heap is done when MIN_SIZE == sizeof(block_node).

void remove_from_free_list(block_node * to_remove);

- This function is called if the malloc function successfully finds a block for re-use. This function is responsible for removing that block from the list of free blocks.

block_node * grow_heap(size_t size);

- This function increases the size of the processes heap by size bytes with the sbrk system call. Grow_heap is only called when no free blocks exist which can satisfy the request. A pointer to the newly allocated block is returned. If the sbrk call fails, NULL is returned. A mutual exclusion lock is used within this function around the sbrk system call since it is not thread-safe.

Together these helper functions form ts_malloc_lock. A size is passed in as the single argument to ts_malloc_lock. The size of a block_node struct is added to the size argument, and a block_node * called target_block is initialized to NULL. The function then checks if it's the first time it's been called. If true, the grow_heap function is called, and its result is assigned to target_block which is then returned with an offset to the data payload (the address at the end of the block_node struct). If false, the calling thread must acquire a lock to call the proceeding functions which access the free list. Once the lock is acquired, try_block_reuse_bf is called. The try_block_reuse_bf function searches the free list for a block of adequate size. If a block is found, the attempt_split function is called from try_block_reuse_bf, which will split the block if possible based on the MIN_SIZE parameter. The result of try_block_reuse is assigned to target_block back in the ts_malloc_lock function body. If target_block == NULL at this point, then no blocks have been found for reuse. The lock is released and target_block is assigned to the result of a grow_heap call and returned. If target_block is non-NULL, a valid block has been found for re-use, and remove_from_free_list is called to remove that block from the list of free blocks before releasing the lock and returning target_block with an offset to the data payload.

The locking version of the thread-safe free function (ts_free_lock) was implemented with the following helper functions:

void add_to_free_list(block_node * to_add);

- This adds a block to the list of free blocks using sorted insert by the block's address (ascending order).

void coalesce(block_node * free_block);

- This function checks if the next and prev pointers in free_block point to free blocks which are adjacent in memory and combines the blocks into a single contiguous block if true. This function may perform up to 2 coalesces per call (one for each block_node * in the block_node struct of free_block), and calls remove_from_free_list for each block it is able to coalesce.

These functions comprise ts_free_lock. A void * is passed in as the single argument. The block_node* is found via pointer offset. A lock for the free list is then acquired (the same lock ts_malloc_lock uses) and add_to_free_list is called to add the block to the free list, making it available for reuse. The coalesce function is then called to coalesce the freed block if possible. Once the calling thread returns from the coalesce function into the body of ts_free_lock the lock for the free list is released and the free function has finished.

Malloc & Free: Non-locking version

The non-locking versions of the thread-safe malloc and free functions are implemented with the help of thread local storage. While this simplifies the implementation, it limits other features such as the ability for blocks to be coalesced if they belong to different threads.

The non-locking version of the thread-safe malloc function (ts_malloc_nolock) was implemented in a similar manner to the locking version. The primary modification is the omission of locks in the body of the ts_malloc_nolock function and the functions which are called. The called functions are the same functions that the locking version of malloc calls but with “thread_” appended to their declaration. This is to distinguish the fact that the free list head and tail pointers which are modified in these functions are found in thread local storage. Each thread has its own list of free blocks, eliminating the need to synchronize access to a shared global free list (threads can perform ts_malloc_nolock and ts_free_nolock at the same time). The grow_heap function is the only unmodified function which uses a lock (as specified in the homework assignment) around the sbrk system call since it is not thread safe.

block_node * thread_try_block_reuse_bf(size_t size);

- Implemented identically to try_block_reuse but searches the free list in thread local storage.

void thread_attempt_split(block_node * to_split, size_t size_needed);

- Implemented identically to attempt_split but modifies the free list in thread local storage.

void thread_remove_from_free_list(block_node * to_remove);

- Implemented identically to remove_from_free_list but removes from the free list in thread local storage.

block_node * grow_heap(size_t size);

- Same function used by both ts_malloc_nolock and ts_malloc_lock.

The non-locking version of the thread-safe free function (ts_free_nolock) is similar to the locking version but does not contain any locks. The use of a free list in thread local storage eliminates the need for synchronization primitives in the ts_free_nolock function. The non-locking version instead uses the modified functions "thread_add_to_free_list" and "thread_coalesce" which are identical to the add_to_free_list and coalesce functions but modify free list pointers in thread local storage.

void thread_add_to_free_list(block_node * to_add);

- Same as add_to_free_list but uses free list in thread local storage.

void thread_coalesce(block_node * free_block);

- Same as coalesce but modifies free list in thread local storage upon successfully coalescing blocks.

Section 3: Performance Results & Analysis

To analyze the performance of the thread-safe malloc and free functions the thread_test_measurement benchmark/test was used. This benchmark was ran using 4 threads and measures execution time and the total size of the created data segment from the benchmark.

A bash script was written to run the benchmark 100 times and record an average execution time of the results. This was done to get a better idea of the average performance of the thread-safe malloc functions, since multi-threaded execution of programs can yield differing results. The thread_test_measurement.c file was modified to print the results of each run to a .csv file which can be opened with Microsoft Excel to easily view stats and calculate an average.

Results for the locking versions of the thread-safe malloc and free functions as well as the non-locking versions are depicted below:

Malloc & Free: Locking version

Execution Time (seconds)	Data Segment Size (bytes)
0.1076	43464255.44

The locking version of the thread-safe malloc and free functions performed as expected based on the outcomes of the best-fit malloc/free tests conducted in Homework 1. Perhaps additional performance could be gained by allowing for more concurrent operations to a globally shared free list. However, I was not able to allow any concurrent access or use a combination of reader/writer locks that didn't introduce the possibility of race conditions or other problems.

Malloc & Free: Non-locking version

Execution Time (seconds)	Data Segment Size (bytes)
0.08499	44522848.4

Based on the results shown above, two conclusions can clearly be drawn. First, the locking version of malloc and free better utilizes memory and has a smaller heap data segment as a result. Second, the non-locking version of malloc and free has a faster average execution time. This is consistent with the expected behavior.

The locking version of malloc and free has a global free list which allows for blocks freed by different threads to be coalesced together. This well help reclaim small free blocks into larger contiguous free blocks which can be better utilized by malloc when searching the free list. Sbrk() calls are therefore reduced in the locking version, since the likelihood of finding a free block large enough to satisfy a request increases. However, the need for locks to serialize access to the thread-global free list has the result of introducing delay to execution time.

While having a free list in thread local storage simplified the implementation of the non-locking version of malloc and free, there are some drawbacks. Having the free list in thread local storage limits the ability to coalesce freed blocks. Even if two freed blocks are adjacent in memory, they will not be coalesced if they were freed by two different threads. This is due to the fact that each thread is unaware of the blocks in another's free list. This is necessary to avoid race conditions and improper results since no locks are used. However, based on the above results, it is also detrimental an efficient use of memory. The non-locking version of malloc and free does however have an improved execution time relative to the locking version. Again, this is consistent with the expected behavior, since the non-locking version does not block any threads which need to access or modify a free list, which allows for more concurrency.

As in the prior homework assignment. The MIN_SIZE parameter which is used to split blocks can be tuned based on the target workload for the thread-safe malloc and free functions. Test runs which varied this parameter generally showed a pattern of sacrificing efficient use of memory for improvements to execution time (at larger values for MIN_SIZE), or sacrificing execution time for a more efficient use of memory (smaller values for MIN_SIZE). Definitively saying which version of the thread-safe malloc and free functions is "best" is a difficult thing to do. The locking version of malloc and free clearly makes a better use of memory, but the non-locking version has better execution time. As is so often the case in the field of ECE, the answer is "it depends" and can largely be reached by an analysis of the target workload allocation size, the frequency of allocations and the prioritization of efficiently using memory.