Nisarg Dabhi
Assignment I: Malloc Library Part II
ECE 650- Spring 2018

Introduction

The files that can be found in my zip file consist of *my_malloc.c*, *test.c, test.h, my_malloc.h*, *Makefile* and this report. Each implementation of malloc and free was tested with custom tests as well as with the allocation test files provided. The customs tests are shown in *test.c* and *test.h*. This report will provide an overview of how allocation policies were executed, display results from performance experiments for each policy and offer an analysis of results.

Implementation Overview: Best Fit Allocation

To implement malloc using a best fit allocation policy, I first elected to represent each memory block with a struct called "mem_block". This struct served as a header for each memory block, storing information on what block came before it, what block was after it, if it was free or not, and finally the size of the block. I then built a Doubly Linked List to represent my free list: a list of memory blocks that are free. I used a global mem_block * variable to serve as the head of the list.

I built a function "sortedInsert" to insert memory blocks into the Doubly Linked List and maintain the list's chronological order. The list was organized in ascending order from lower addresses to higher ones. This order was maintained to allow for easy merging of blocks later in the free list. I also built a function to remove items from the list, "RemoveNode", when memory blocks could be used to complete allocation requests. I created a function, "BestFitFind", that would traverse the linked list starting from its head node to find the best node that was large enough to complete a call to malloc. This function returned NULL if no block was large enough to fulfill a request.

Next, I built a function, "addblock" which would push back the program break by calling sbrk() in order to create space for a new memory block. This was called when no free nodes in the free list were large enough to complete a malloc request. I created a function "partition" that would split up memory blocks. If a malloc request only needed half of a memory block, partition would create a new block out of the remaining space in the memory block, mark it as free, and add it to the freelist represented by the Linked List. To merge blocks that are next to one another in the freelist and are next to one another in memory, I created "merge_block". This function added the size of a Header and the size of the memory block to the size of the starting address of a memory block and compared this value to the starting address of the next block. By doing so you can determine if these free blocks were actually next to one another and should be combined. This function iterates through the whole free list and coalesce any blocks that are directly adjacent to one another.

The function "BestFitFind" iterates through each node in the free list starting from the head. It calculates the difference between the size of the current node it is on and the size of the memory request and compares it to a minimum difference value. If it is smaller than the minimum difference value and positive, the minimum value is replaced by that calculated difference. This process is repeated for every node in the free list, until the smallest positive difference in the size of node and the size of a memory request is found along with the index of that node. The index of the node is then used to get that specific memory block from the free list.

The malloc functions works by looking for the best available block (through BestFitFind) in the free list. If one is found, it attempts to split it (through partition) and after doing so removes it from the free list. If no block is found on the free list, a new one is created and used to fulfill a malloc call. The free functions work by using the pointer inputted to the function to get to the exact address of the start of the memory block that will be freed. The function then will add that block to the free list, and finally attempt to merge the block that was just added to the free list to any blocks it may be adjacent to.

Implementation Overview: Thread Safe Lock Implementation

To make the above Best Fit implementation thread safe with lock, I had threads acquire a global mutex or lock right before finding an available block through BestFitFind in my malloc function. The function would release the lock either immediately if no block was found, or only after calling RemoveNode to remove the found block on the free list. The calls to BestFitFind and RemoveNode form a critical section as no two threads should be searching the free list for available blocks simultaneously. If they do, they may navigate to the same block and lead to data being overwritten by two separate threads. In order to ensure no threads also end up finding the same block after searching the free list, the lock is only released after the removal of a found block.

If no block is found, a call to addblock is made to extend the program break. Since the sbrk system call is not thread safe, the call to sbrk forms a critical section. As a result, a lock is acquired immediately before the sbrk call and released immediately after in the addblock function.

For the free implementation with locks, a lock is acquired right before the call to sortedInsert, that inserts blocks into the free list. This stops race conditions between threads trying to add blocks at the same time. If threads try to simultaneously add a block in the same spot in the free list, one of the thread's blocks being added may not successfully be added to the chain. As a result, you would lose data.

The lock is also released only after two calls to merge_block to merge the added free block to the block before it and after. If two threads have blocks next to one another in the free list and both try coalescing blocks, the size field of each block may not actually be correct. As a result, a block may think it has more space available than it is actually allocated, leading to the possibility of data erroneously being overwritten. Thus, the sortedInsert call and the merge_block calls in the free implementation with locks form a critical section.

The main place where concurrency is still available in this implementation is in the calls to partition. The partition function is used to split a block so that only the necessary amount of memory is used and the rest of the memory in the block will remain in the free list. In order to successfully split a block, pointer arithmetic is needed to get to the start of the remainder of a free block that can go back into the free list. This pointer arithmetic can be large as a pointer must move "size" bits over in the address space where "size" is the amount of space specified by a malloc call.

Malloc calls the partition function after successfully releasing the lock right after a call to RemoveNode. As a result, the block that was just removed from the free list is then split up. Multiple threads can then call partition without spawning race conditions. as no threads are simultaneously accessing the free list. Each thread instead is just splitting up a block that is not on the free list. The one caveat to this implementation is the partition function has its own critical

section. A lock is acquired right before partition makes a call to sortedInsert to insert the split portion of the original block into the freelist. The lock is immediately released on the next line.

The one other spot for concurrency in the implementation is in the beginning of free. In every free call pointer arithmetic must be done to successfully get to the start of the metadata of each block. This pointer arithmetic can still occur concurrently as it is not in a critical section.

Implementation Overview: Thread Safe Lockless Implementation

The Lockless implementation has code laid out exactly the way specified in the Best Fit Allocation section. The main distinction is that Thread Level Storage(TLS) is used to hide what one thread sees from what another thread sees. A global head node that uses thread level storage named thread_base is created in the my_malloc header file. This head node pointer for the free list is thread local meaning a different copy of this node pointer is used for each individual thread. As a result, each thread will be manipulating its own free list.

This implementation works by having all of its function calls that involve the free list use thread local versions of head nodes, instead of a shared global head node. As a result, each thread can concurrently execute on their own free lists. The only critical section is the call to sbrk as this function is not thread safe. Each thread must acquire the global lock to call sbrk and then will immediately release it after the call. Additionally, this implementation uses a function called "partition_no_lock" to split up blocks as this is a lockless version of the partition function explained in the lock implementation.

Performance Experiments and Analysis

Table I: Execution Time and Data Segment Size for Lock Thread Safe Implementation

Run	Execution Time (seconds)	Data Segment Size
0	.192298	42918816
1	.120302	45613376
2	.189500	46416384
3	.154451	45720928
4	.124728	47634688
5	.133759	44343936
6	.124569	45029856
7	.119176	43779744

Table II: Average Execution Time and Data Segment Size for Lock Thread Safe Implementation

	Execution Time (seconds)	Data Segment Size
Average Value	.14484788	45182216

Table III: Execution Time and Data Segment Size for No Lock Thread Safe Implementation

Run	Execution Time (seconds)	Data Segment Size
0	.096041	47580640
1	.101614	46864864
2	.188714	45564832
3	.113183	45987808
4	.106700	46675776
5	.149695	45658688
6	.099713	45617536
7	.099444	46264064

Table IV: Average Execution Time and Data Segment Size for No Lock Thread Safe Implementation

	Execution Time (seconds)	Data Segment Size
Average Value	.119388	46276776

We expect the locked thread safe version of malloc and free to have a greater execution time than in the lockless thread safe version. Since each thread has to acquire the global lock before executing the critical section, threads are barred from fully concurrently executing. This leads to a greater delay in execution as each thread must wait until another thread releases the lock in order to acquire it. Additionally, we should expect the data segment size to be smaller for the locked version than in the lockless version. Each thread in the lockless version has its own free list since it has its own thread local version of a head node. The lock version on the other hand shares one free list between all threads, which should lead it to have a smaller overall data segment size.

Tables I-IV show the execution time and data segment size results for both sets of implementations. The test case measuring execution time and data segment size had randomized behavior so multiple test runs were executed. The execution time and data segment size data for each implementation was averaged to give a clearer picture of the normal expected behavior for each implementation.

We can see that the results above support our expectations for each implementation. The locked thread safe implementation had an average execution time of .14484788 seconds while the lockless version had an average execution time of .119388. We can see that the lockless version seemed to execute quicker. The average data segment size on the locked version was 45182216 while the average data segment size was 46276776 for the lockless version. Again, our results are consistent with our expectation as the lockless version seemed to have a larger data segment size on average when compared to the locked version. These numbers are extremely close however, so it is difficult to say whether the data truly supports our expectations.