ECE650 ASSIGNMENT 2 REPORT

1. THREAD-SAFE MALLOC IMPLEMENTATION

In the previous assignments, we were asked to implement our own malloc libraries that achieve the same functionality as the stdlib in C. My previous implementation makes use of a doubly linked list to store the meta-information of each block allocated and freed. Although the implementation works as excepted, the runtime of this implementation is drastically slowed down since whenever we need to find an available free block for a new requested block, the implementation has to iterate through the entire list that contains much unhelpful information. Now for this specific assignment, we are no longer interested in computing segment size. So, to improve the performance, we could take advantage of this and achieve the malloc functionality by manipulating a doubly linked list that stores information of free blocks.

A. Locking Version Implementation

To make sure that the program is thread-safe, we could introduce mutex lock from pthread library to synchronize threads and prevent race conditions. The mutex variable is declared as a global variable. The algorithms are pretty much the same as the previous assignment. A thread must acquire the mutex before starting running the main algorithm, which is also the critical section for the multithreaded implementation, of the malloc/free function, and that thread must release the lock before the malloc/free function returns. The head node of the linked list is also declared as a global variable in this case.

The malloc function first uses best-fit policy to find an available free region that fits the newly requested memory block. Now, the function searches for the available block by iterating through the linked list of free blocks, so runtime could be vastly reduced. After iteration, if a proper block is found, it is popped out of the linked list. And we then try to split the block if the block size exceeds the sum of the requested memory size plus the size of meta-information structure. This action also needs to manipulate the linked list since the remaining portion of the original block is now free, and we have to update the linked list. If an available block is found but its size is not large enough to be split, we just remove the node from the linked list. In case that no such proper free space is available, we call sbrk() to get a new block and directly return the starting address of newly sbrk'd memory block.

The free function basically also has the same layout as the non-TS version. Given an arbitrary address, we first find the corresponding meta-information structure, and then we are able to add this block to the linked list to mark it as a free region. Next, we try to merge the adjacent free blocks. In order to do that, we need to check if the blocks are actually adjacent to each other. We could use some pointer manipulations to accomplish that. Meanwhile, a little trick that was used is maintaining the order of the linked list. It helps to easily tell whether the end of previous block is exactly the beginning address of the latter block and if multiple blocks are physically adjacent.

Given a block to be freed, we check its previous block and its next block and try to merge them. The free process is completed after merging.

B. Non-locking Version Implementation

The non-locking version implementation prohibits the usage of mutex lock. But the implementation virtually has the same algorithms as locking version, with one major exception -- the global variable used before now becomes invalid for non-locking implementation. To solve this, we make use of the Thread-local storage to replace the head of our linked list which was previously shared globally. By declaring the head as TLS variable, we allocate the variable such that there is 1 per thread locally so that each thread sees its own list.

2. PERFORMANCE EVALUATION

Both implementations were tested for multiple times using the provided test program. The some of results were shown below.

```
yl506@vcm-2850:~/ece650/draft/draft/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.192617 seconds
Data Segment Size = 42773072 bytes
yl506@vcm-2850:~/ece650/draft/draft/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.267733 seconds
Data Segment Size = 43387128 bytes
yl506@vcm-2850:~/ece650/draft/draft/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.198963 seconds
Data Segment Size = 42722528 bytes
```

Fig.1 Lock version output

```
No overlapping allocated regions found!
Test passed
Execution Time = 0.192696 seconds
Data Segment Size = 42854296 bytes
yl506@vcm-2850:~/ece650/draft/draft/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.146053 seconds
Data Segment Size = 43411376 bytes
yl506@vcm-2850:~/ece650/draft/draft/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.200586 seconds
Data Segment Size = 43773304 bytes
```

Fig.2 Non-locking version output

Here I listed 3 test results for each implementation, due to the space limitation. But we could still see the general trends of the two implementations: Lock version has longer runtime but smaller data segment size, while non-locking version has faster runtime but larger data segment size. This conclusion is in agreement with the mechanisms of multithreaded programming. On the one hand, Mutex lock is easy to use, but it introduces performance overhead and makes our programs more serial/sequential since other threads are blocked when the lock has been taken, which opposes the

objective of multithreaded programming and results in slower runtime. On the other hand, non-locking version makes good use of multithreaded computing that results in faster runtime. But since each thread maintains its local scope, the memory utilization efficiency decreases when free is called, since some of the real adjacent free regions are not merged together due to the fact that they may be visible to different threads. Also, the fact that there are multiple linked lists with smaller size also reduces the probability of finding an available free block relative to a single large list with many nodes. So the tradeoff is apparent, and I am really interested in how the real thread-safe malloc library is implemented.