Assignment #2: Thread-Safe Malloc
ECE 650 – Spring 2018

Liang Zhang/lz139

I. Implementation Statement

locking version

1. Locks
I use two locks in the implementation:
pthread_mutex_t lock_sbrk;
pthread_mutex_t lock_update;
lock_sbrk is used to guard the heap extend actions, which can guarantee that only one thread can execute the sbrk() action in one time. lock_update is used to guard the existing blocks status update actions, like block merge, split and status(free or occupied) update actions. It can guarantee that there will be only one thread changing the status of one unique block at one time from both malloc and free functions, and no memory leaks will happen during these actions.

2. Critical Sections Analysis
There are 3 critical sections in meta_t merge_lock(meta_t block); and 1 critical section in void ts_free_lock(void *ptr);

In meta_t merge_lock(meta_t block);
critical section 1:
At the beginning, the linked list is empty. So after calling malloc, multiple threads will call sbrk() to extend the heap and update the head of the linked list, which will cause race condition. I use lock_sbrk to lock this critical area to guarantee that only one thread can call sbrk() to extend the heap and update the linked list head. To achieve that, after the thread acquire the lock and enter the critical area, the function will recheck whether the head is NULL. If yes, the thread can go on with the heap extend action. If no, it means that other thread has already updated the linked list head. Then the current thread will call malloc again to allocate the required block.
critical section 2:
If the linked list is not empty, the threads will use best-fit policy to find the required blocks concurrently. If there are no suitable ones found in the existing free blocks, threads will call sbrk() to acquire blocks at the end of the list at the same time, which will cause another race condition. I use lock_sbrk to lock this critical area to guarantee that only one thread can call sbrk() to extend the heap at one time. Also, after one thread acquire the lock, it will enter a while loop to check whether the curr pointer is pointing to the tail of the linked list(as other thread might have

extended the heap and changed the tail address). If not, it will be updated to the correct tail position to do the sbrk() action to extend the heap. The reason I use same lock in critical section 1 and 2 is that they both apply sbrk() action, which is not thread-safe.
critical section 3:
After best-fit search, multiple threads might find the same empty block suitable and want to split and return it at the same time, which will cause the third race condition. I use lock_update to guard this section to make sure that only one thread can mark block as occupied and split it if needed at one time. After one thread enters this critical area, the function will recheck whether the acquired block is free or not(it might have been occupied by other thread). If it is free, the current thread can split and return it. If not, the current thread will do another malloc operation by calling void *ts_malloc_lock(size_t size) again.

In ts_free_lock(void *ptr);
critical section 1:
After one thread free one block, it will check its previous and next blocks. If the adjacent blocks are also free, merge actions will be applied. There are two kinds of race conditions in this area. The first one is that while the current thread is trying to merge the current block and the next block, the next block might have be chosen as a suitable block by another thread doing malloc action. So the merge action and malloc return action might happen at the same time. However, the merge action will combine the current and next blocks into the current block, and remove the next block from the linked list. But the other thread is not aware of this and will still return the next block after malloc. In this situation, the next block data area is actually occupied, but it is marked as free in the linked list, and other threads can reoccupy this area, which will cause overlapping allocated regions. Also, the occupied next block is lost in the linked list now, which will cause memory leak. The second race condition is that after one thread mark the current block as free, it might be occupied by another thread instantly, while the current thread is doing merge action on current block and previous free block at the same time, which will cause same problems as stated above. To solve this issue, I add lock_update to guard the block status updating and merge actions in the free function. This will guarantee that only one thread can update the unique block status from both malloc and free functions at one time. There is also one problem after adding this lock. If the thread form malloc acquire the lock first, everything is fine. But if the thread from free function acquire the lock first, after merge action, the merged next block can still be occupied by the thread from malloc function. To solve this issue, I rewrite the meta_t merge_lock(meta_t block); function to mark the next block as occupied after the merge action. Then after the thread from the malloc function acquire the lock, it will recheck the block information. After it find out that the block has been occupied, it will give up the block and find a new one.

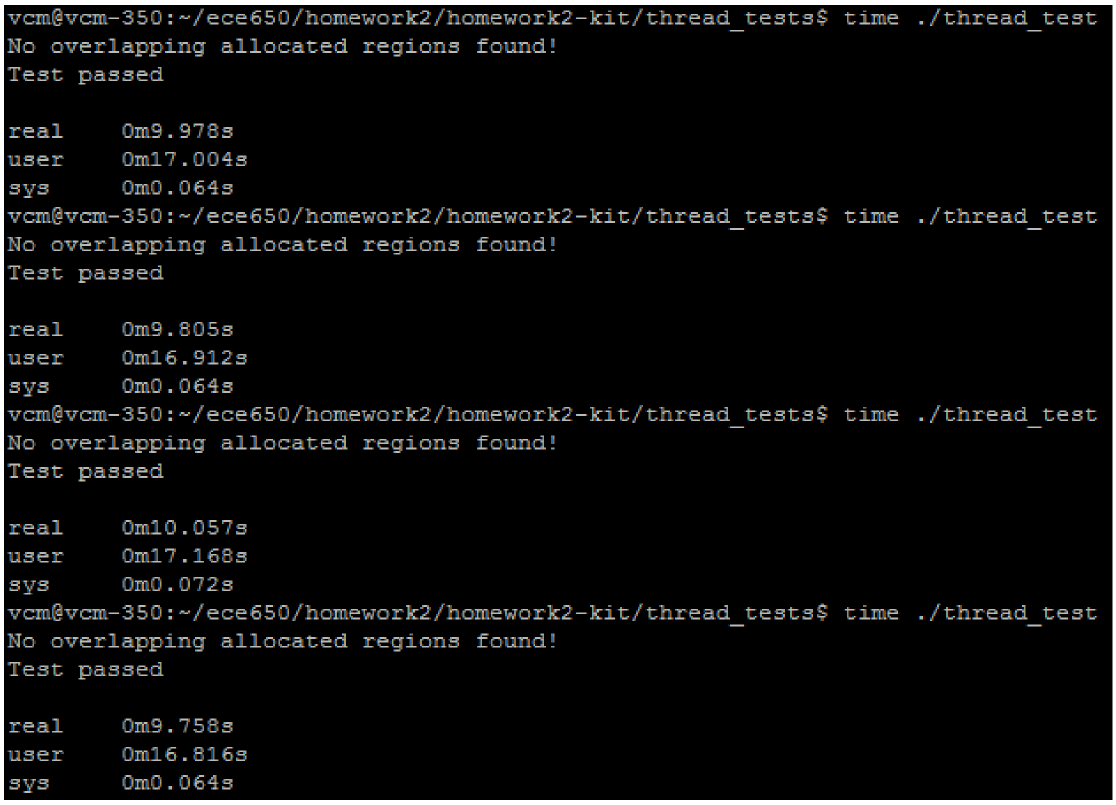
3. Concurrency Analysis
This thread-safe implementation allows concurrency in two ways:
1. In void *ts_malloc_lock(size_t size);, multiple threads can enter the linked list to do best-fit search to find the suitable blocks at the same time, which will save lots of time comparing to sequential execution.
2. Thread which finds the suitable block and thread which not after best-fit search can do their

following actions at the same time. For example, thread 1 finds the right block after search and will update the block and return it next. Thread 2 does not find the right block and will extend the heap to acquire the block next. These two actions can be executed concurrently, which will save some time comparing to sequential execution.

4. Execution Results

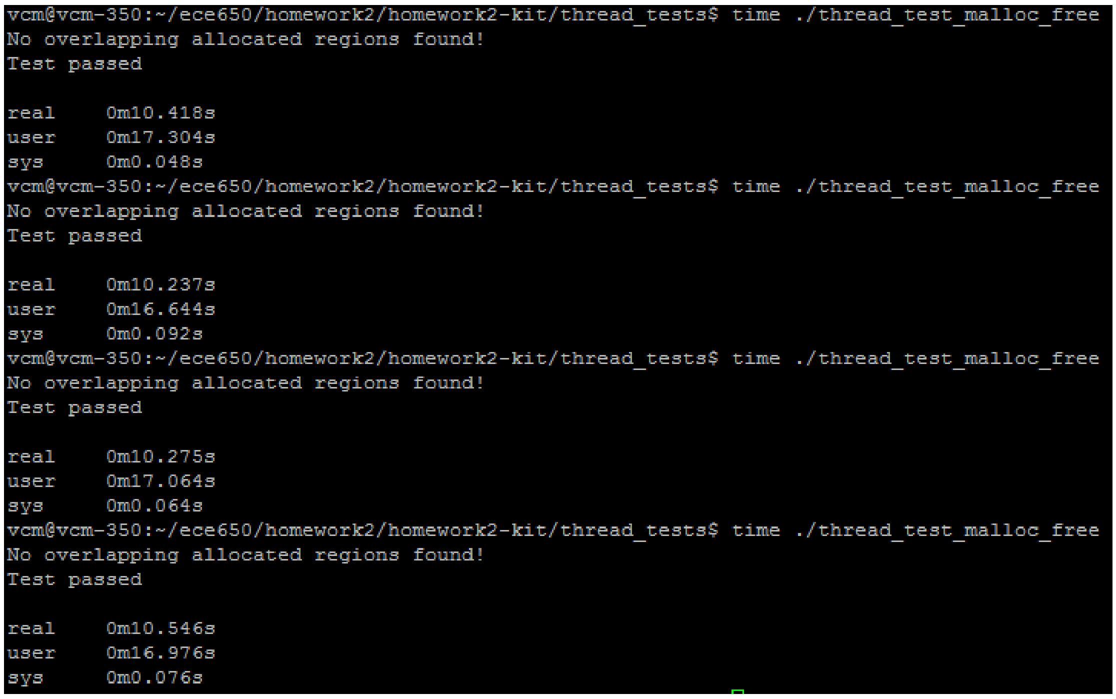
I ran each test with original parameters 4 times

thread_test



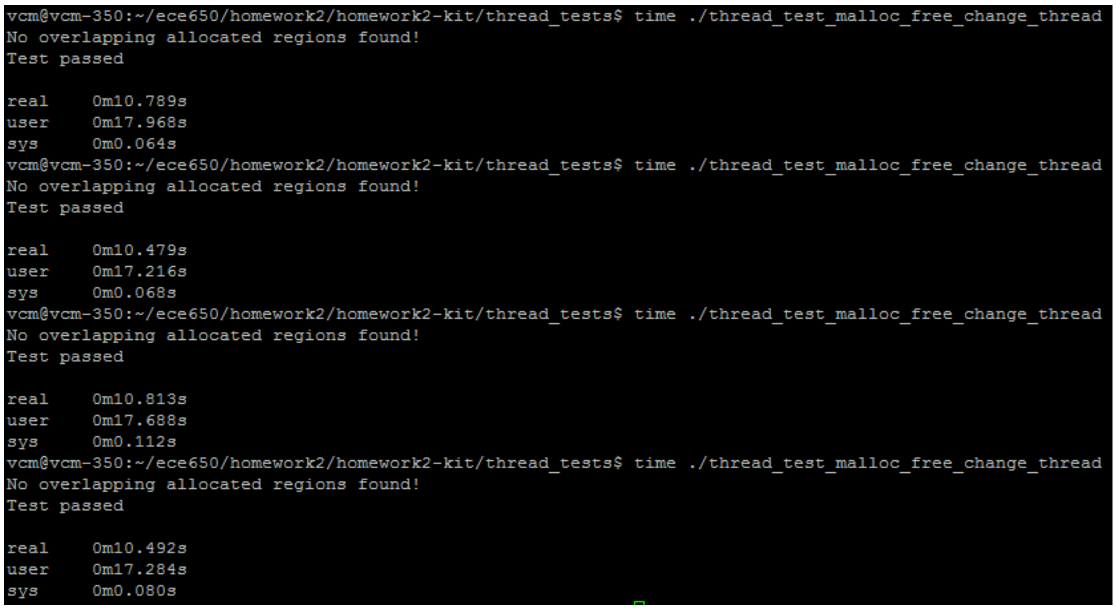
```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test
No overlapping allocated regions found!
Test passed

real    0m9.978s
user    0m17.004s
sys     0m0.064s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test
No overlapping allocated regions found!
Test passed

real    0m9.805s
user    0m16.912s
sys     0m0.064s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test
No overlapping allocated regions found!
Test passed

real    0m10.057s
user    0m17.168s
sys     0m0.072s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test
No overlapping allocated regions found!
Test passed

real    0m9.758s
user    0m16.816s
sys     0m0.064s
```

thread_test_malloc_free



```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed

real    0m10.418s
user    0m17.304s
sys     0m0.048s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed

real    0m10.237s
user    0m16.644s
sys     0m0.092s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed

real    0m10.275s
user    0m17.064s
sys     0m0.064s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed

real    0m10.546s
user    0m16.976s
sys     0m0.076s
```

thread_test_malloc_free_change_thread



```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed

real    0m10.789s
user    0m17.968s
sys     0m0.064s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed

real    0m10.479s
user    0m17.216s
sys     0m0.068s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed

real    0m10.813s
user    0m17.688s
sys     0m0.112s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed

real    0m10.492s
user    0m17.284s
sys     0m0.080s
```

Non-locking Version

1. TLS and Locks

TLS:

__thread void * block_head_nolock=NULL;

I create a TLS block_head_nolock for each thread, so each tread will have its own linked list with a unique list head. As each thread has its own linked list, so they can do block search, block split, and block status update(occupy and free) concurrently. As blocks from each thread list are intersected with each other in the heap, so free blocks adjacent to each other in one linked list

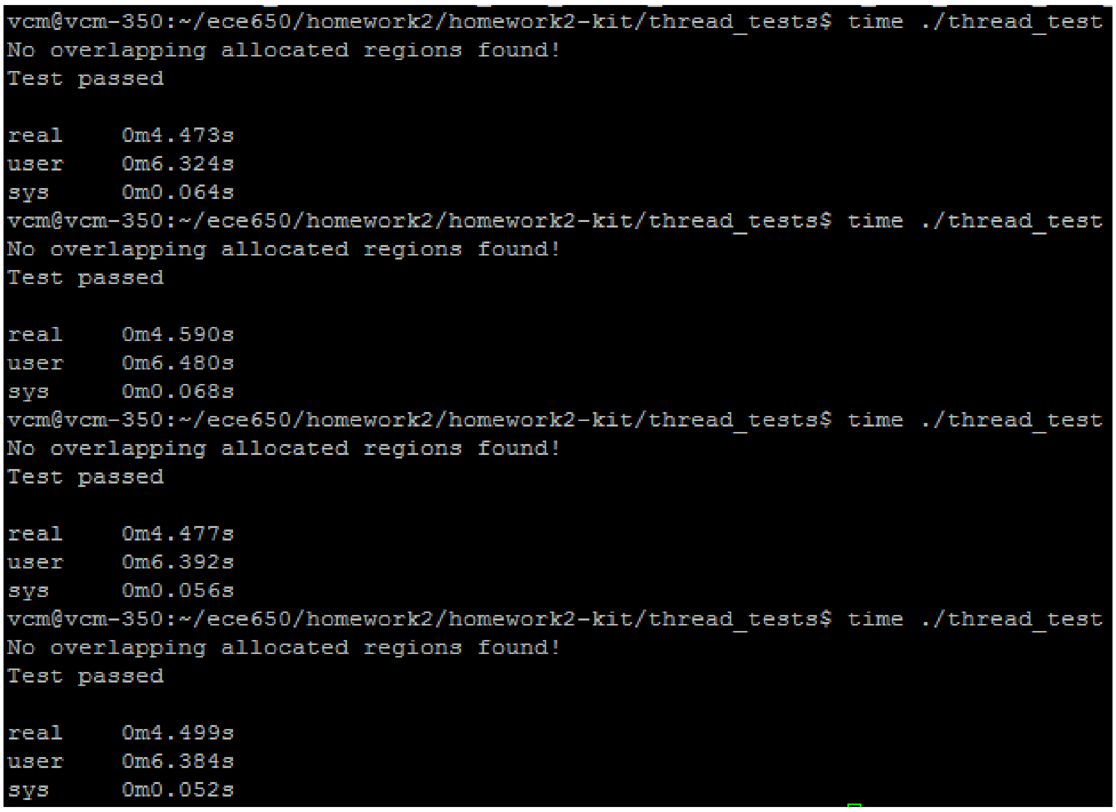
may not be adjacent in the heap. Physical adjacent blocks merge action is only applied when they are in the same linked list in this implementation due to this reason. Before merge action in each list, I will check whether these two blocks are physically adjacent in the heap. If no, no merge action will be taken.
Lock:
pthread_mutex_t lock_nolock=PTHREAD_MUTEX_INITIALIZER;
I use a mutex lock to guard the sbrk() action as sbrk() is not thread-safe.

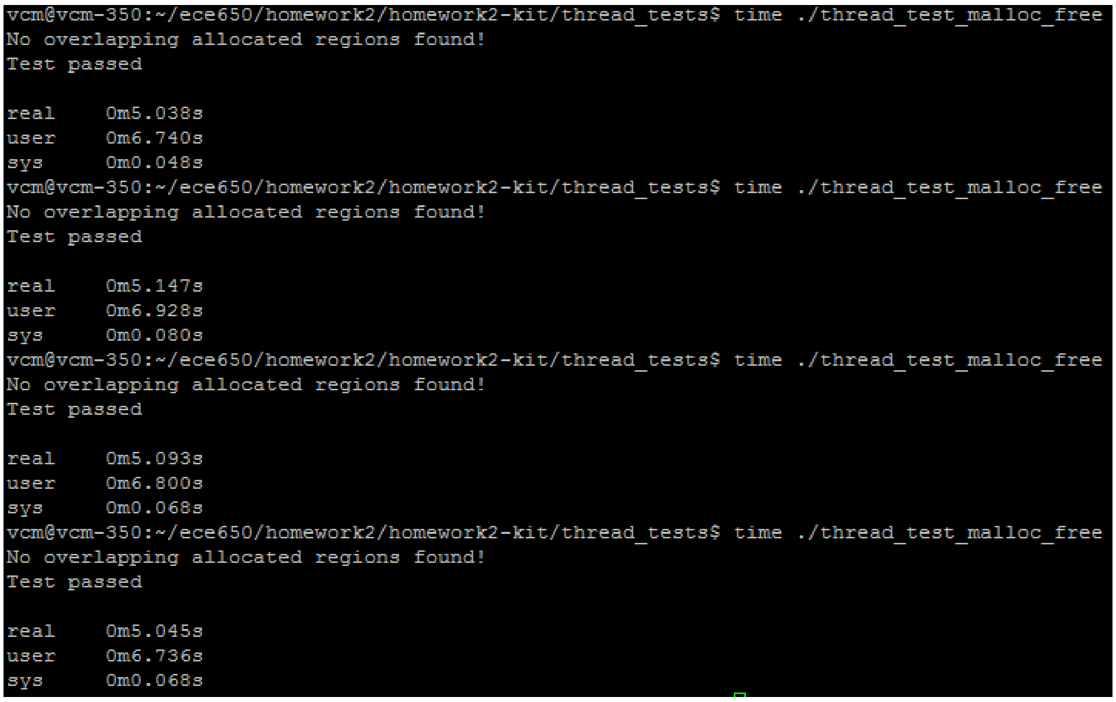
2. Critical Sections Analysis
As each thread has its own linked list in this implementation, so there are only two race conditions in this case:
1. Multiple threads call the sbrk() function when extending the heap at the same time. As sbrk() is not thread-safe, multiple threads need to call sbrk() sequentially. So I add a mutex lock to guard the sbrk() area in meta_t grow_heap_nolock(meta_t curr, size_t size);.
Also in this case, as there are several linked lists, so the tail of each list is not the real break point of the heap. So when each thread extends the heap, it needs to call sbrk(0) first to update its tail to the break pint of the heap first. This can guarantee each thread will extend non-overlapping blocks at the tail.

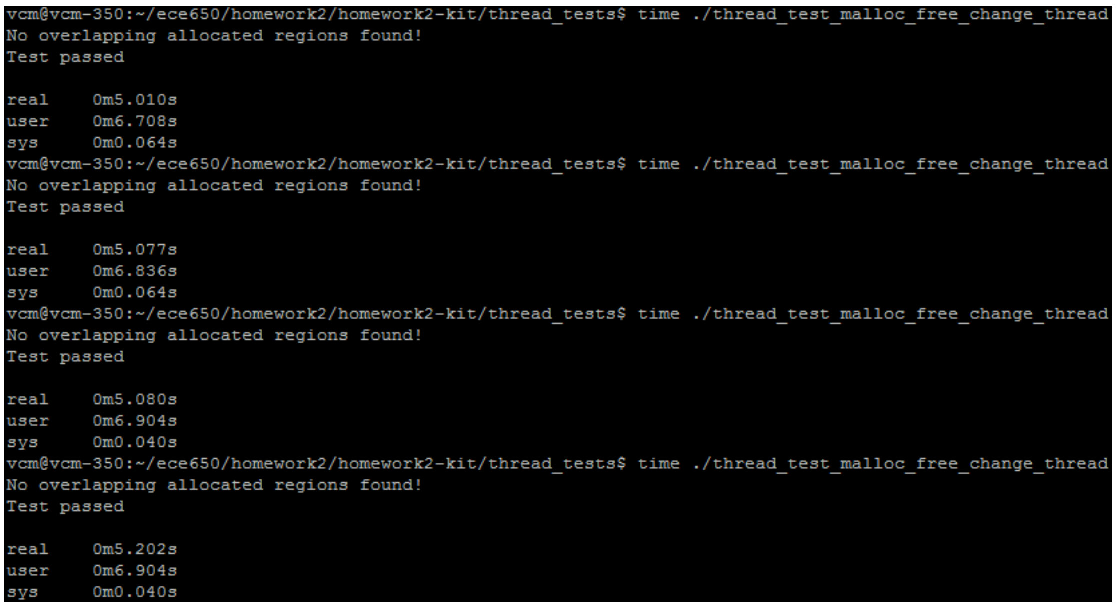
2. After thread 2 freeing the block which is malloced by thread 1, it will try to do the merge action if the next block in the list is free and physically adjacent to the current block. However, thread 1 may want to occupy the same next block at the same time, which will cause race condition. To avoid this, I add a thread_id part in the meta data of each block, which records the thread(linked list) it belongs to. Before merge action, I will check whether the current block's thread_id matches the current thread. If yes, no race condition will occur and merge action can carry on. If no, race condition will occur, thus I will cancel the merge action.

3. Concurrency Analysis
As each thread has its own linked list, so they can do the following actions concurrently:
1. Use best-fit policy to search for the suitable blocks.
2. Update the blocks as occupied or free.
3. Split the blocks when needed.
4. Merge the adjacent blocks in each list when requirements are met.
Only one action needs to be done sequentially in this implementation, the sbrk() call. When multiple threads want to extend the heap at the same time, they need to call sbrk() sequentially to ensure the thread-safe action.

4. Execution Results
I ran each test with original parameters 4 times
thread_test


```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test
No overlapping allocated regions found!
Test passed

real    0m4.473s
user    0m6.324s
sys     0m0.064s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test
No overlapping allocated regions found!
Test passed

real    0m4.590s
user    0m6.480s
sys     0m0.068s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test
No overlapping allocated regions found!
Test passed

real    0m4.477s
user    0m6.392s
sys     0m0.056s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test
No overlapping allocated regions found!
Test passed

real    0m4.499s
user    0m6.384s
sys     0m0.052s
```
thread_test_malloc_free


```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed

real    0m5.038s
user    0m6.740s
sys     0m0.048s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed

real    0m5.147s
user    0m6.928s
sys     0m0.080s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed

real    0m5.093s
user    0m6.800s
sys     0m0.068s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed

real    0m5.045s
user    0m6.736s
sys     0m0.068s
```
thread_test_malloc_free_change_thread

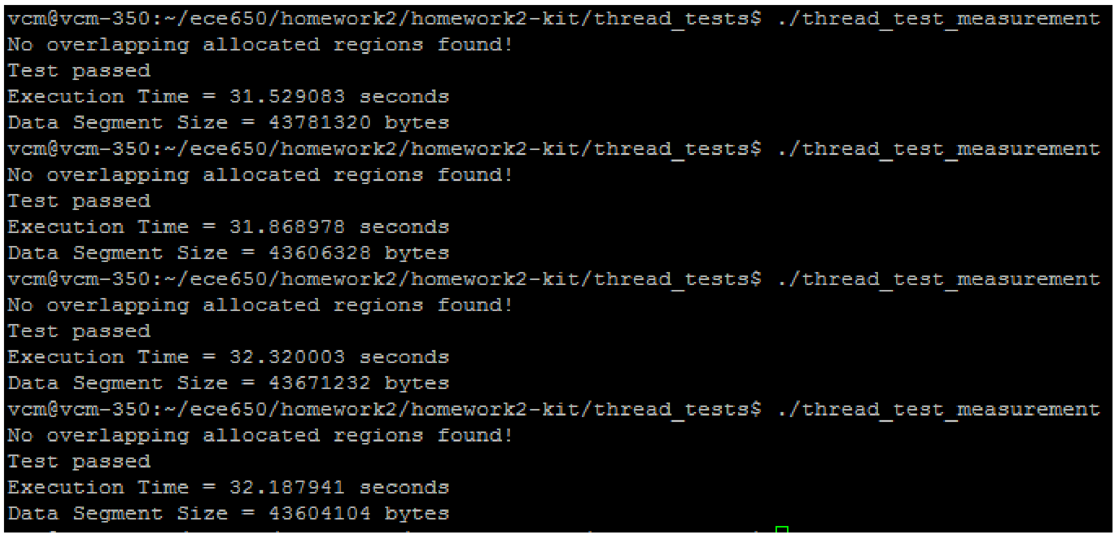


```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed

real    0m5.010s
user    0m6.708s
sys     0m0.064s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed

real    0m5.077s
user    0m6.836s
sys     0m0.064s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed

real    0m5.080s
user    0m6.904s
sys     0m0.040s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed

real    0m5.202s
user    0m6.904s
sys     0m0.040s
```

II. Performance Analysis on two Versions

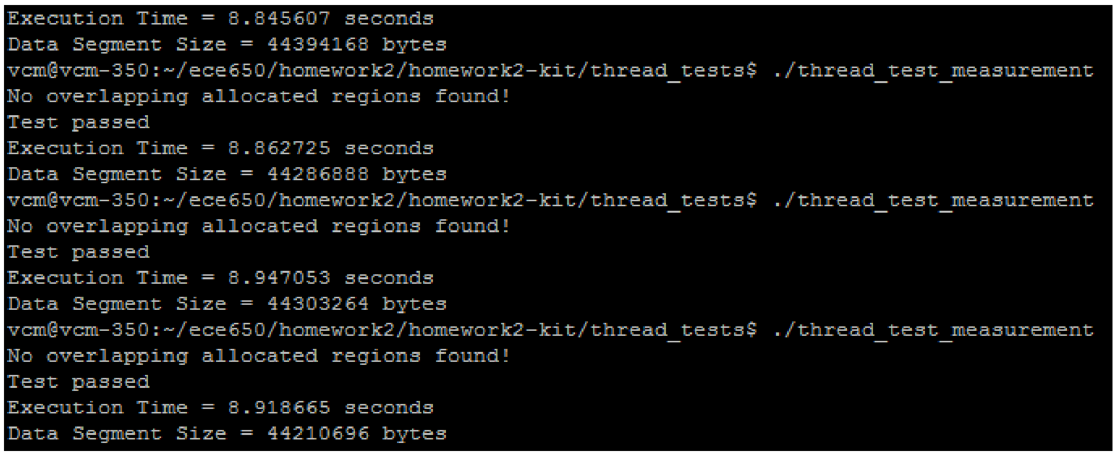
thread_test_measurement results(I left the parameters unchanged):

locking version



```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 31.529083 seconds
Data Segment Size = 43781320 bytes
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 31.868978 seconds
Data Segment Size = 43606328 bytes
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 32.320003 seconds
Data Segment Size = 43671232 bytes
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 32.187941 seconds
Data Segment Size = 43604104 bytes
```

non-locking version



```
Execution Time = 8.845607 seconds
Data Segment Size = 44394168 bytes
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 8.862725 seconds
Data Segment Size = 44286888 bytes
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 8.947053 seconds
Data Segment Size = 44303264 bytes
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 8.918665 seconds
Data Segment Size = 44210696 bytes
```

1. Allocation Efficiency Analysis

We can see that the average data segment size of the lock version is 43600000 bytes. And the average data segment size of the non lock version is 44300000 bytes. Lock version is a little more efficient in this test via non lock version. The reason is that after freeing action, adjacent free blocks will be merged to create one larger combined block in lock version, while due to multiple linked lists, non lock version will do merge action only when the two adjacent blocks are in the same linked list. Thus the non lock version is more likely to have small free block segments while lock version doesn't. As a result, best-fit search is more likely to find suitable blocks in the lock version, while non lock version is more likely to do the heap extend action as small free segments can not meet the demand. Thus non lock version will occupy more data segment size after the test than the lock version.

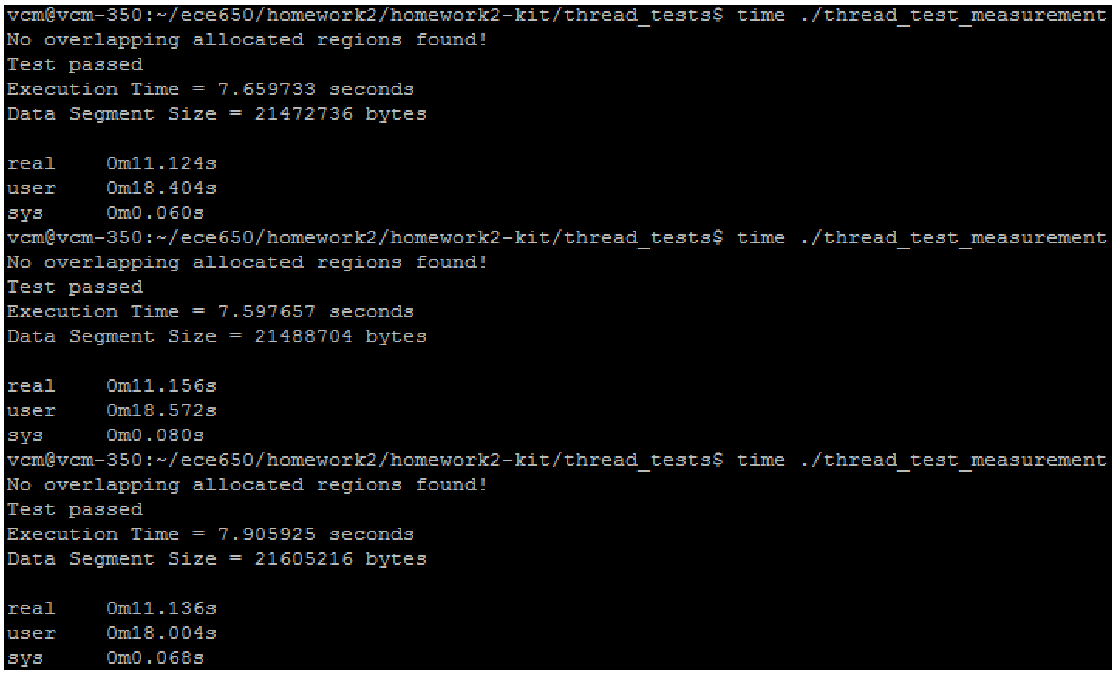
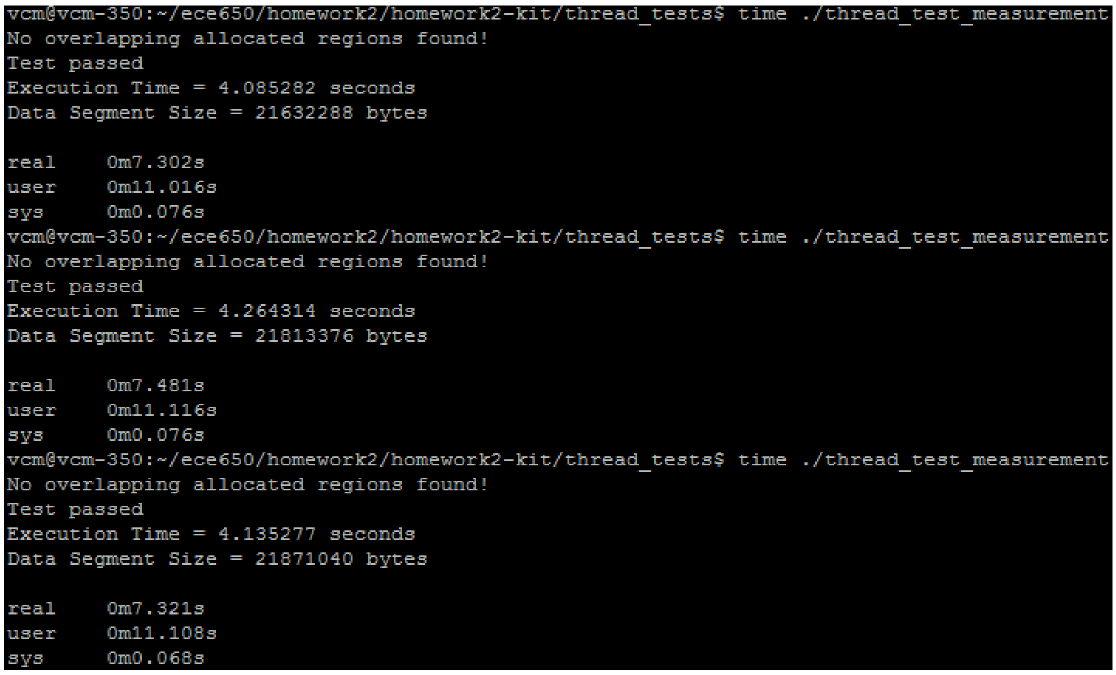
However, the difference between these two version is not that obvious. That's because in thread_test_measurement, only some of the threads (the threads with an even ID) occasionally free allocations from all the threads. If the free action happens more frequently, the merge function will also be called more times, which will lead to bigger difference in the results.

2. Execution Time Analysis

Most of the execution time is spent on the best-fit block search action. As best-fit will traverse through the whole linked list to find the right block, the list length is the crucial factor in deciding the execution time.

We can see that the average execution time of the lock version is 32s. And the average execution time of the non lock version is 8.8s. 32/8.8=3.6. That's approximately 4. This result comes from the fact that the length of the linked list in the lock version is approximately 4 times of the non lock version.

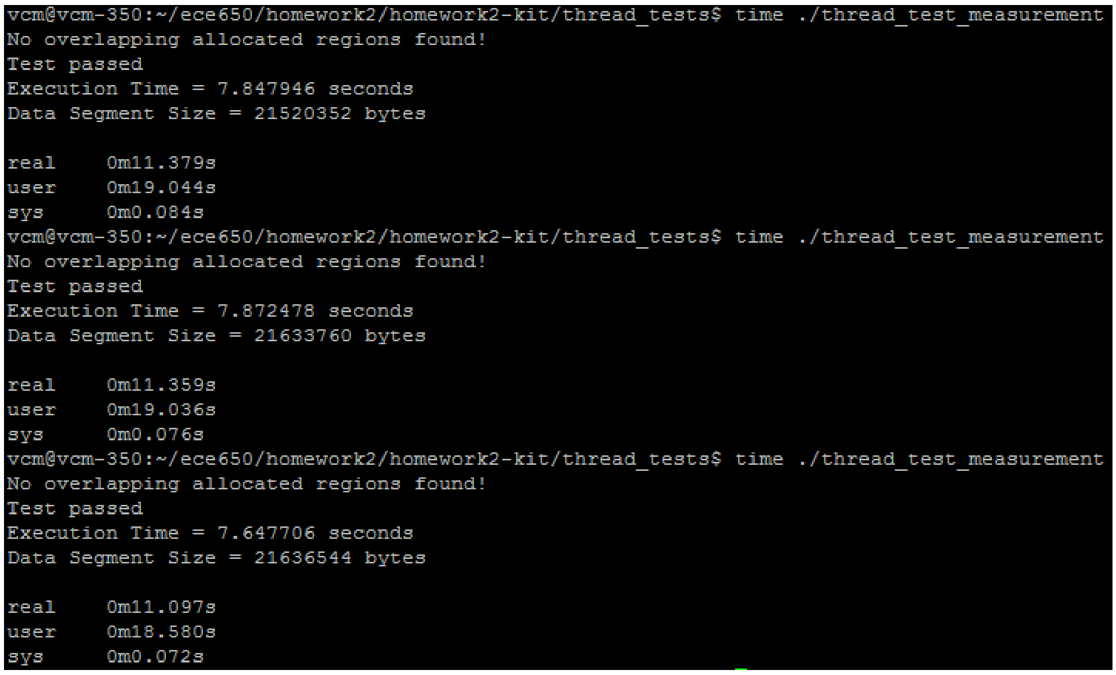
As from above, we can see that the data segment sizes after execution are almost the same in the two versions. In this test case, there are 4 threads executing the code in both two versions. In the lock version, 4 threads will traverse though the same linked list to find the suitable blocks. However, in the non lock version, the 4 threads have their own distinct linked lists, they only need to traverse through their own list to find the suitable blocks. As the total length of these four lists is approximately the same as the single list of the lock version, each thread in the non lock version only needs to traverse through 1/4 length of the lock one. This list length difference leads to the different best-fit search time, and affects the execution time.

So as a result, if there are n threads executing at the same time, the execution of the lock version is approximately n times of the non lock version.
To test this theory, I changed the NUM_THREADS form 4 to 2 in thread_test_measurement.c, and other parameters are unchanged. The execution results are:
Lock version

```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 7.659733 seconds
Data Segment Size = 21472736 bytes

real    0m11.124s
user    0m18.404s
sys     0m0.060s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 7.597657 seconds
Data Segment Size = 21488704 bytes

real    0m11.156s
user    0m18.572s
sys     0m0.080s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 7.905925 seconds
Data Segment Size = 21605216 bytes

real    0m11.136s
user    0m18.004s
sys     0m0.068s
```
non lock version

```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 4.085282 seconds
Data Segment Size = 21632288 bytes

real    0m7.302s
user    0m11.016s
sys     0m0.076s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 4.264314 seconds
Data Segment Size = 21813376 bytes

real    0m7.481s
user    0m11.116s
sys     0m0.076s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 4.135277 seconds
Data Segment Size = 21871040 bytes

real    0m7.321s
user    0m11.108s
sys     0m0.068s
```

We can see that the average execution time of the lock version is 7.7s. And the average execution time of the non lock version is 4.1s. 7.7/4.1=1.9. Which is approximately 2. This meets the above theory.

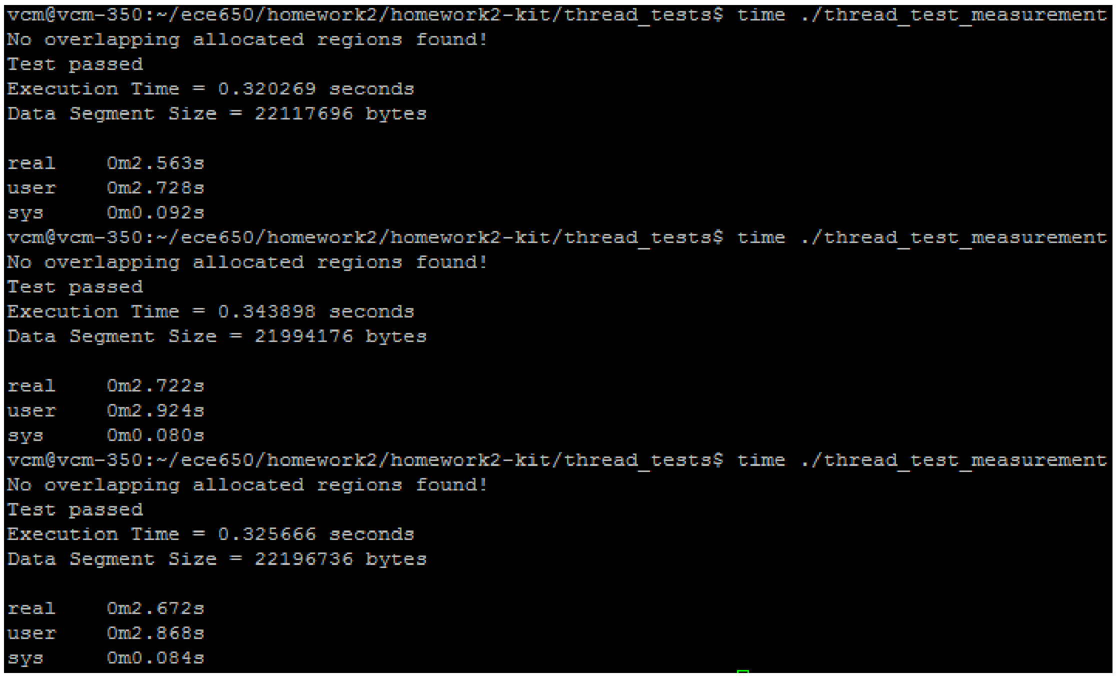
And then I changed the NUM_THREADS form 4 to 20 in thread_test_measurement.c(I also changed the NUM_ITEMS from 20000 to 2000 to save time). The results are:

lock version



```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 7.847946 seconds
Data Segment Size = 21520352 bytes

real    0m11.379s
user    0m19.044s
sys     0m0.084s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 7.872478 seconds
Data Segment Size = 21633760 bytes

real    0m11.359s
user    0m19.036s
sys     0m0.076s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 7.647706 seconds
Data Segment Size = 21636544 bytes

real    0m11.097s
user    0m18.580s
sys     0m0.072s
```

non lock version



```
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.320269 seconds
Data Segment Size = 22117696 bytes

real    0m2.563s
user    0m2.728s
sys     0m0.092s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.343898 seconds
Data Segment Size = 21994176 bytes

real    0m2.722s
user    0m2.924s
sys     0m0.080s
vcm@vcm-350:~/ece650/homework2/homework2-kit/thread_tests$ time ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 0.325666 seconds
Data Segment Size = 22196736 bytes

real    0m2.672s
user    0m2.868s
sys     0m0.084s
```

The average execution time of the lock version is 7.8s. And the average execution time of the non lock version is 0.34s. 7.8/0.34=23, which is approximately 20. This meets our theory too.

There are other factors may influence the execution time. They are:

1. Threads in the lock version can not do blocks update actions like update the blocks status(mark as free or occupied), split blocks and merge blocks concurrently. However, threads in the non lock version can do these actions concurrently. This factor will make the execution time of the lock

version longer than the non lock version.
2. The lock version will do adjacent blocks merge action when needed while the non lock version will only do merge action when the adjacent blocks belong to the same lined list. This will make the lock version's utilization ratio of the blocks higher than the non lock version. As a result, threads in lock version will do less heap extend action than the non lock version. And the list length of the lock version will be a little shorter than the total lists length of the non lock version. All these will make the execution time of the lock version shorter comparing to the non lock version.
However, we can observe from above that the majority of the execution time is spent in the block search action. That's why the execution time of the lock version will be even longer than the non lock version if more threads are added into the program execution.