ECE 650 Project #2: Thread-Safe Malloc Report

Xh114, Xiao Han

1. Implementation Overview

1) Lock Version

a) Malloc

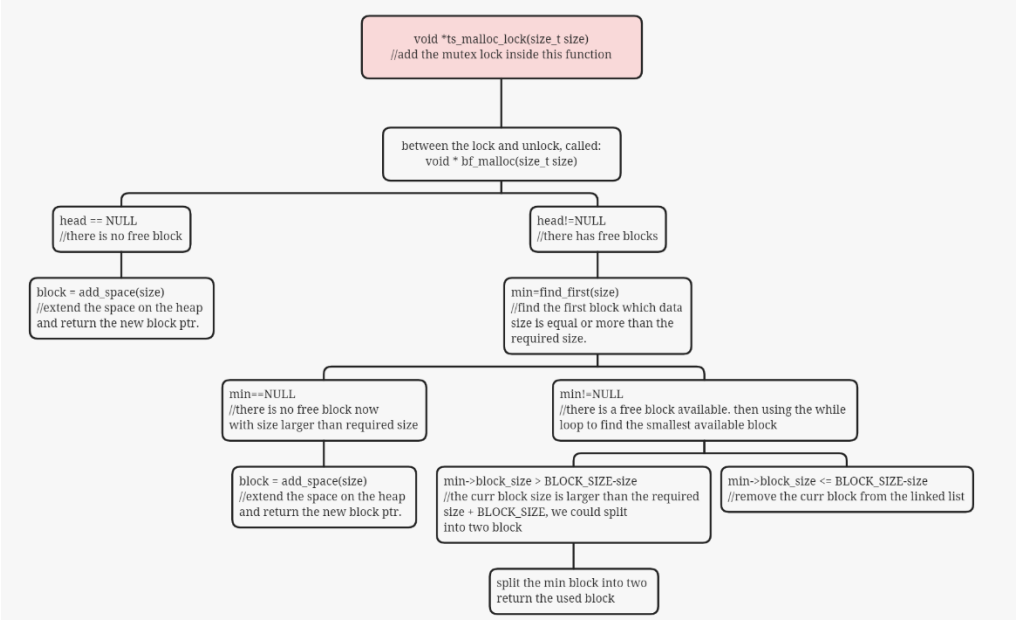
In order to keep the thread-safe for the best fit malloc and free function, in the lock version, I simply use the mutex lock from the beginning of the function call.

a. Initiate a mutex lock as a global parameter; therefore, when a thread is called the function, it will also initiate a lock for future use.

```
pthread_mutex_t mutex_malloc =PTHREAD_MUTEX_INITIALIZER;
```

b. Lock and unlock this lock in malloc and free function. Therefore, whenever a thread is using malloc or free, the other thread calling the malloc or free must wait until the previous thread is finished using malloc or free.

```
void *ts_malloc_lock(size_t size){
        pthread_mutex_lock(&mutex_malloc);
        void * ans = bf_malloc(size);
        pthread_mutex_unlock(&mutex_malloc);
        return ans;
    }
```



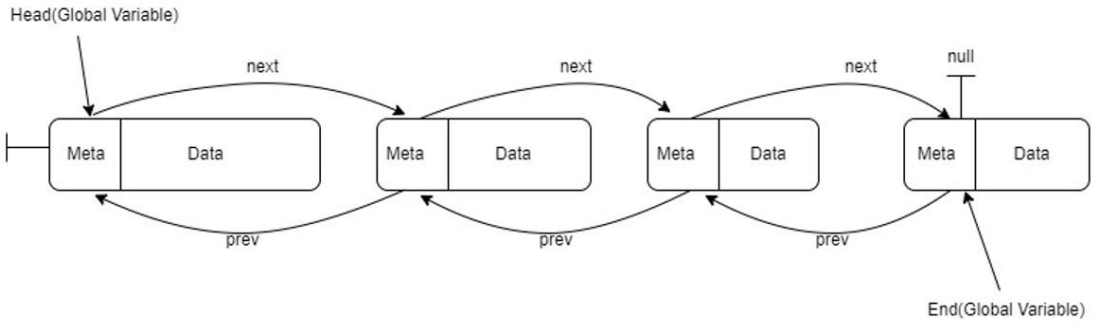
From the diagram, we can see the mutex clock add at the beginning of the whole implementation therefore the other operation after that will keep the same.

b) Free

For the lock version, to keep the thread-safe for free() function is the same way as malloc, add the lock and unlock in the font and back of call the free_block(*ptr) function:

```
void ts_free_lock(void *ptr){
    pthread_mutex_lock(&mutex_malloc);
    free_block(ptr);
    pthread_mutex_unlock(&mutex_malloc);
}
```

Based on this implementation, the free block listed list data structure only has one list on the heap; multi-thread will add/delete the free block on the same list.



2) No-Lock Version

a) Malloc

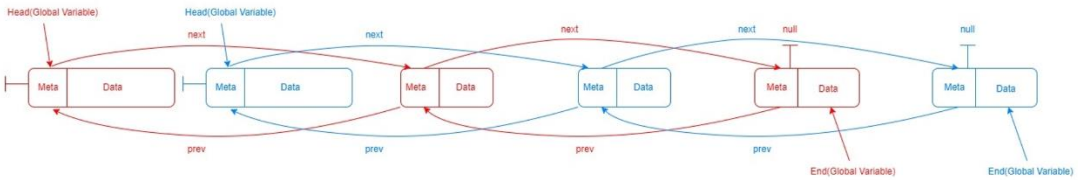
For the no-lock version, after carefully checking my previous design, the only parts that need to take care of is at add_space(sizt_t size) function, where will using sbrk() to allocate more space on the heap, where may cause the race condition: when one thread sbrk(size) for new space, at the same time, the other thread calling sbrk(size) at the same space.

Based on the explore, the no-lock version will be updated in the steps as below:

a. Instead of adding the global parameter for the head and end, adding

```
__thread block_ptr head_thread = NULL;
__thread block_ptr end_thread =NULL;
```

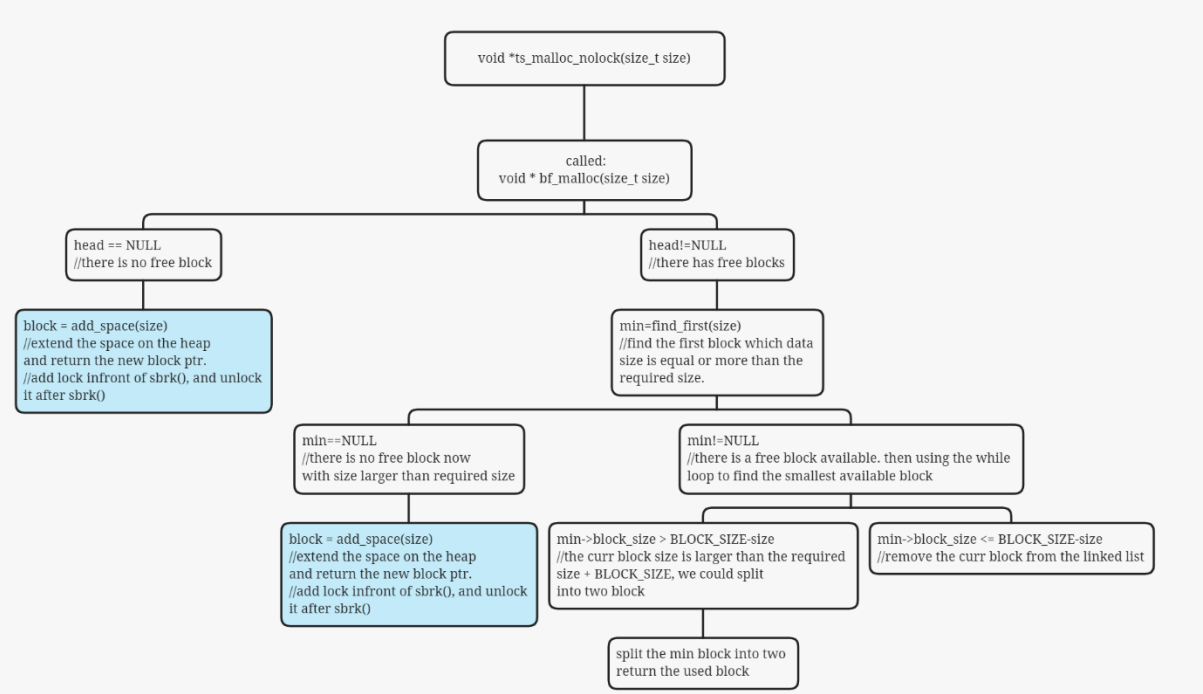
This will create a head, and an end block_ptr for each thread called the function, different than the first version with lock, this will create a list for each thread. Shows as the diagram below, here I use two threads as an example.



b. Adding the lock and unlock before and after sbrk(size) in add_space function:

```
pthread_mutex_lock(&mutex_thread);
block_ptr new = sbrk(BLOCK_SIZE+size);
pthread_mutex_unlock(&mutex_thread);
```

The diagram below shows where to add the lock in the no-lock version,



In this diagram, we can see the differences between the two versions, the lock_version with the lock at the front, therefore the functions below will only operate one time at the same time; In the no-lock version, only the add_space function will operate one time at the same time, the other functions could keep changing the linked list of the own thread list at the same time.

2. Result Analysis

For each version, I run the test for 20 times and take the average result as below:

1) Lock Version

Execution Time: 1.472

Data segment size = 43481024

2) No-Lock Version

Execution Time: 0.267

Data segment size = 43354304

For both versions, the data segment size is almost the same, this comes from two reasons:

a) The function which increases the data segment size will be add_space function; in the lock_version, the function is locked and unlocked following the whole malloc operation, therefore the total data size increase will be the similar as the no-lock version, where we only close the add_function.

b) Since they both use the best fit approach, the differences between the two data segment sizes because: in some cases, the choice of use which block in different lists are different, since the lock version only has one list for all threads but the no-lock version has multiple lists for multiple thread.

The execution time makes a big difference between the two versions.

a) In the lock version, since there is only one list for the multiple threads, it takes longer than the no-lock version every time it loops from the free blocks.

b) In the no-lock version, each thread has its own list, and multiple threads could run while loop

to find the min block in their thread at the same time, therefore it will take way more less time than the lock_version.

3) Tradeoff

Even though the data segment size does not make significant differences in these test cases. But in a specific condition, where the lock version could provide more efficiency in area use because it contains more blocks, which offer more choices when finding the min free block, if the area usage is critical for us, the lock version should be a better choice.

But in general, the no-lock version provides way more better execution time, when the runtime is critical, the no-lock version should be a better choice.