Report 2

Keyu Chen
kc487

1.Implementation

This project is an extension of project 1 by adding thread-safe features. Both of my thread-safe malloc and free functions used the best fit allocation policy. This project deals with multi-threaded execution and it is implemented in two different ways, lock and unlock.

1.1 Lock Version

For the malloc/free function, most work is same as project 1. In the lock version, I added two lines of code to my previous code: pthread_mutex_lock(&lock) and pthread_mutex_unlock(&lock); I defined a free list which contains the free node and it can only be accessed by one thread one time by applying above 2 lines. A thread must acquire the mutex lock before entering the critical section. Meanwhile, the same thread must release the lock before malloc or free return. In ts_malloc_lock, I first used bf to find a free space in free list. Then I determined that if it needs to be split. If I failed to find a free node, I need to generate a new node at the end of heap and remove it.

```
void *ts_malloc_lock(size_t size){//this function is used to implement malloc
    pthread_mutex_lock(&lock);
    Node *node =NULL;
    node = find_free_node_bf(node,size,&head);

    if(node != NULL){//If find a free node
        if (node->datasize > NodeSize + size){
            return split(node,size,&head,&tail,0);
        }else{
            return removeNode(node,size,&head,&tail,0);
        }


    }else{//unable to find a free node => node=NULL
        node = (Node*)sbrk(size+NodeSize);
        node->next = NULL;
        node->datasize = size;
        node->prev = NULL;
    }
    pthread_mutex_unlock(&lock);
    return node + 1;
}
```

figure 1

```
void ts_free_lock(void* ptr){
    pthread_mutex_lock(&lock);
    if(ptr == NULL){
        return;
    }
    Node *node = (Node*)(ptr-NodeSize);
    InsertToFreeList(node,&head,&tail);
    Node* prevN = node->prev;
    Node* nextN = node->next;
    if(prevN==NULL){
        merge(node,nextN);
    }else{
        merge(prevN,node);
        if(prevN->next != NULL){
            merge(prevN,nextN);
        }
    }
    pthread_mutex_unlock(&lock);
}
```

figure 2

As we can see from above pictures, the section between pthread_mutex_lock(&lock) and pthread_mutex_unlock(&lock) is critical section. In the ts_free_lock, I need to insert the node into the freelist firstly. Then doing the merge operation.

1.2 Non-lock Version

In this version, I only used lock when I called sbrk(). The original global head and tail variables used before became useless now.In order to avoid two threads accessing the same linked list at the same time, I used Thread-local storage to replace the head and tail before. And defining the head and tail of the linked list for each thread.

__thread Node * head_nolock = NULL;

__thread Node * tail_nolock = NULL;

Meanwhile, we need to notice that we still need to use lock when calling sbrk() like following,

pthread_mutex_lock(&lock);

node = (Node*)sbrk(size+NodeSize);

pthread_mutex_unlock(&lock);

2. Result Analysis

I run both of these two implementations for multiple times. And results are shown below.

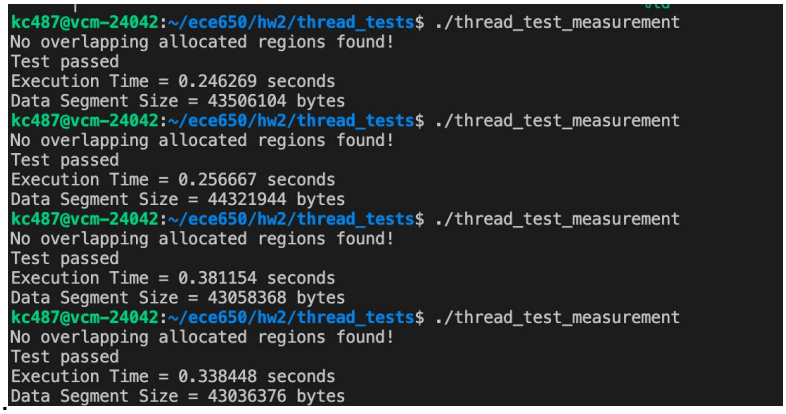

Figure3.LockVersion

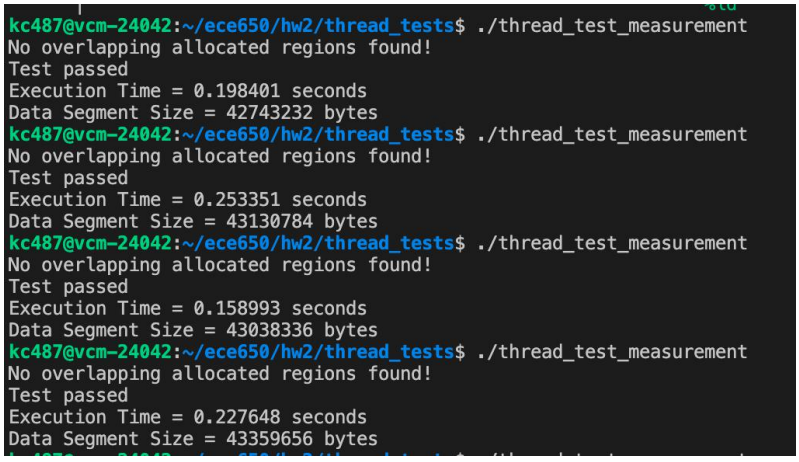

Figure 4. Non-Lock Version

From the above 2 pictures, we can learn that Lock Version has longer execution time than Non-Lock Version, and Non-Lock Version has larger data segment size than Lock Version. But the difference is slight. Maybe the difference is caused by the difference of concurrency level. For the Lock Version, the whole program become more sequential because other threads are blocked when using lock. However, Non-Lock Version makes good use of multiple thread but decreases memory utilization efficiency. So in response to trade-off, people who cares more about running time will choose Non-Lock Version. And people who care more about memory utilization efficiency would choose Lock Version.