ECE 650 Assignment 2: Thread-Safe Malloc

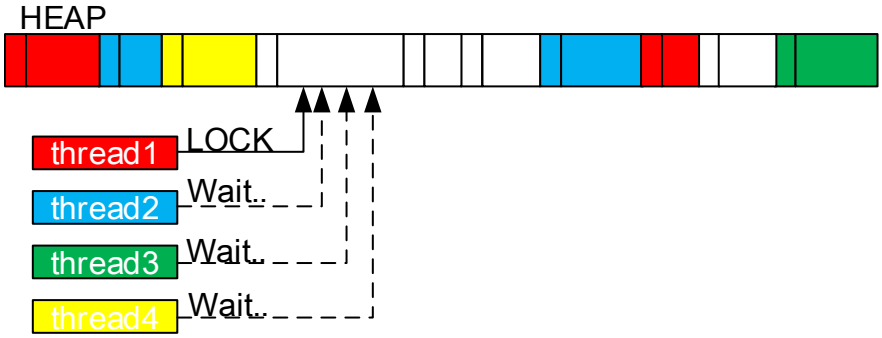
Yuchen Yang(yy227)

This report is a study of the implementation of thread-safe “malloc” function in C library. There are mainly three parts contained, function implementation, experimental results, comparison and analysis.

I. Function Implementation

In real life, the malloc function is used in multiple threads in one process, and this leads to the race condition. There are multiple ways to deal with it, in this report, I used two of them to solve this problem.

1. Lock Version Malloc



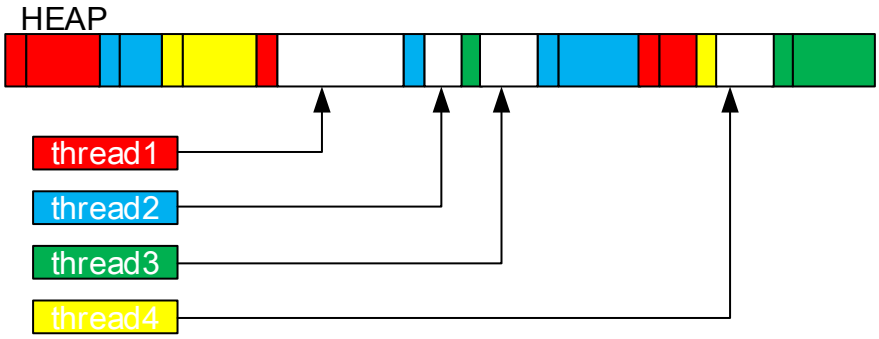
Based on last assignment, the data structure I implemented is just a huge linked list. When all threads try to access the same space, I let the first arrived thread to take up the space and locked the rest, which is shown in the figure above. This can be achieved by adding a “mutex” to where the race condition happens, which is the trunk is available.

```
pthread_mutex_lock(&lock);

while(current_mem != heap_end){
    struct mem_meta * mm = current_mem;
    last_mem = current_mem;
        if(mm->is_available){
            if(mm->size == size){
            mm->is_available = 0;
            memory_location = current_mem;
            temp_mem = NULL;
            break;
            }else if (mm->size>size){
                if (mm->size<min_size){
                    min_size = mm->size;
                    temp_mem = current_mem;
                }
            }
        }
   current_mem += mm->size;
}
```

```
if(temp_mem!=NULL)
{
    struct mem_meta *mm = temp_mem;
    mm->is_available=0;
    memory_location = temp_mem;
}
if(!memory_location){
    sbrk(size);
    memory_location = heap_end;
    struct mem_meta *mm = memory_location;
    if (last_mem){
        struct mem_meta *last_mm = last_mem;
        last_mm->next = mm;
        mm->prev = last_mm;
    }else{
      mm->prev = NULL;
    }
    mm->next = NULL;
    mm->is_available = 0;
    mm->size = size;
    heap_end += size;
}
pthread_mutex_unlock(&lock);
return memory_location + sizeof(struct mem_meta);
```

2. No-lock Version Malloc



In this case, I separated the data by their thread id. The original linked list which everything was related now breaks into several lists grouped by thread id. When malloc and free, threads can only access the list with same thread id.

```
if(!memory_location){
    pthread_mutex_lock(&lock);
    sbrk(size);
    memory_location = heap_end;
    struct mem_meta *mm = memory_location;
    if (last_mem){
        struct mem_meta *last_mm = last_mem;
        last_mm->next = mm;
        mm->prev = last_mm;
    }else{
      mm->prev = NULL;
    }
    mm->next = NULL;
    mm->is_available = 0;
    mm->size = size;
    mm->thread_id = tid;
    heap_end += size;
    pthread_mutex_unlock(&lock);
}
return memory_location + sizeof(struct mem_meta);
```

The reason why we have to keep the lock on sbrk() is that there is a non-avoidable situation when multiple threads run out of space and ask for new space.

Also note that when free, we need to merge the adjacent free chunks with the same thread id.

```
void* location =mm;
if (mm->prev!=NULL && mm->prev->is_available==1){
    int size =mm->prev->size;
    int length = mm-mm->prev;
    if (size==length){
        mm->prev->size+=mm->size;
        mm->prev->next=mm->next;
        if (mm->next!=NULL)
            mm->next->prev = mm->prev;
        mm = mm->prev;
    }
}
```

II. Experimental Results

Result of lock version:

```
yy227@vcm-8148:~/yy227/hw2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 69.721826 seconds
Data Segment Size = 46367632 bytes
```

Result of no lock version:

```
yy227@vcm-8148:~/yy227/hw2/homework2-kit/thread_tests$ ./thread_test_measurement
No overlapping allocated regions found!
Test passed
Execution Time = 35.048642 seconds
Data Segment Size = 45612448 bytes
```

	Execution Time	Data Segment Size
LOCK	69.721826 s	46367632 bytes
NOLOCK	35.048642 s	45612448 bytes

III. Analysis and Conclusion

1. From the table above, it can be observed that the execution time of no_lock_version malloc is half the time of lock version. This is because when multiple threads trying to malloc the same space, thread_lock blocks all others but one, and the waiting time is really expensive. Contrary to the lock version, in my no_lock_version malloc, it only searches for the linked list with the same thread id, and the execution time is saved.
2. In both malloc, the data segment size of no_lock malloc is slightly smaller than the lock version. However, in theory, the lock version should have done better in space allocation because the no_lock malloc can only allocate data in the linked list with same thread id while lock malloc does not have that constraint. Through digging into the test files, I found that one possible reason is that in the test file the process is "malloc-free", which means that both version allocated data trunks and then freed all of them and did not go through the "malloc-free-malloc-free" process. In this case, the advantage of lock version cannot be observed easily.

   To support my hypothesis, I went through the formula below:

$$chunks \approx \left(\frac{\min\_chunks + \max\_chunks}{2} \right)$$

$$Bytes = NUM\_THREADS \times NUM\_ITEMS \times chunk\_size \times chunks$$

The computation result is 46 080 000, which is the maximum value of possible segmentation size. Through examining the test code, the problem may lay in the lock inside test code. Due to this additional lock, the free processes of lock version malloc are blocked, and leading to a much larger segmentation size.