THREAD_SAFE_MALLOC

Section 1: Overview

Recall in the first assignment in which we don't need to consider thread safety, I kept two doubly linked list in my implementation. One is a linked list which tracks all of the nodes I had through `sbrk()` system call while the other one is a free linked list which only tracks the nodes that are currently free. This implementation becomes very tricky when it needs to consider thread safety since manipulating `prev` and `next` pointers inside the node will almost inevitably cause race condition. This problem can be solved by simply putting a global `mutex_lock` between the `malloc` and `free` function, which maybe inefficient, but certainly can solve the problem. The largest issue is that it's really tricky, time-consuming and error-prone when you consider implementing lock free version by this data structure.

So what I did is to rewrite my whole data structure. In my second implementation, I didn't keep a large linked list to track all nodes I've `sbrk`'d. Instead, I only kept a doubly free linked list to track all the free nodes. The general thoughts is like the following:

> 1. When someone want to allocate a chunk of memory, `malloc` will first search through the free linked list to see whether there is available free node with proper size the user asks for. If there is, then simply use that node and kick it out from the free linked list.
> 2. If there is no available node inside of the free linked list, `malloc` will just call `sbrk()` to allocate a new chunk of memory and immediately return this chunk of memory to user.
> 3. When one wants to free a chunk of memory, `free` will add the "to be freed" block into free Linked list. To merge adjacent nodes, it needs to check whether it's physically previous node and next node are free or not. If any of them is free, merge needs to happen.

While this seems legit, the only problem is that how are we going to find the physically previous and next nodes since we do not have that "friendly" linked list to keep track of all the nodes?

To solve this problem, we just need to do a little modification to our free list. We make our linked list sorted by address in ascending order. That is the address of each node keeps increasing from head to tail. If we do so, we know the current "to be free" node's previous node exists, this previous node's address must be less than the current node. Since we can have access to previous node's meta head, we can reach the end of this previous node by doing pointer arithmetic. The next thing we need to do is to check whether the end of this previous node is the beginning of the current node. If it is, these two nodes are physically adjacent and we can do merge, otherwise we do nothing. The similar mechanism works for its next node. The data structure and code snippet of merging nodes looks like this:

```
typedef struct _Node_t Node_t;
struct _Node_t{
  Node_t * prev_free;
  Node_t * next_free;
  size_t blk_size;
};

void merge(Node_t * first, Node_t * second){
  if(first == NULL || second == NULL){return;}
  Node_t * check = (Node_t *)(((int8_t *)(first + 1)) + first->blk_size);
  if(check == second){ //is adjacent
    if(first->next_free->next_free != NULL){
      first->next_free->next_free->prev_free = first;
    }
    first->blk_size = first->blk_size + second->blk_size + metaSize;
    first->next_free = first->next_free->next_free;
  }
}
```

This data structure will make the lock free version very easy to implement. What we need is to make the head of the free linked list a local variable to each thread so that nodes in one thread's free linked list will be invisible to any other threads. Fortunately, `pthread` library does offer this functionality called `TLS`. However, this certainly will have tradeoff. One major tradeoff is that we are unable to merge nodes that belong to two different threads since every node can't see the other node even if they are physically adjacent to each other.

Speaking of lock version, if we use sorted singly linked list, there are actually multiple ways to implement this. We can use node level lock. That is we add one lock in each node, preventing race condition by locking the node that we are currently operating on and its adjacent nodes. Or we can also use read/write locks to increase performance. However, the latter two implementations are not very easy to implement and certainly take more time to debug. Thus, I chose the easiest version to implement considering the given time limit.

Section2: Performance

Version	real(s)	user(s)	sys(s)	execution(s)	data segment (bytes)
lock_free	31.29	31.208	0.08	0.164460	43327528
lock	30.363	30.224	0.144	0.251747	43598688

You can see from the data given above that lock_free version runs faster than lock version by looking at the execution time which is the actual time spent in allocating memory and free them. You may wonder why the user and real time are much larger than the execution time. This is because that most of the time is spent in checking the correctness of implementation by detecting overlapped allocated areas. The execution result is reasonable because for lock version it doesn't actually take advantage of multithread benefit. Since we lock the entire function, every time only one thread can `malloc` or `free`. Other threads will be block, no matter what they are trying to do. This sounds like what sequential programming will do. The time complexity exists at two places:

> 1. Since we are using best fit algorithm, search an available block in the free Linked List certainly takes O(n) to finish
> 2. When we free one node, since we need to add this node into a sorted free linked list, this will also take O(n) to finish.

Since we gain no benefit by using multithread and increase an extra O(n) operation when invoking free function, the performance will be worse than the implementation I use in the first assignment.

Let's take a look at the lock_free version. Since it also uses sorted free Linked List, it will increase an extra O(n) operation when invoking free function. The good news is that multithread works well in this version because every thread won't block each other. They can improve the performance by dividing the task. The tradeoff is apparent, However, since it is impossible for free blocks in different threads to be merged. This will increase the length of linked list and the times to invoke `sbrk()` system call. You can infer this by looking at the different data segment size of two versions. The `lock_free` version is larger than a `lock` version. The overall behavior of `lock_free` is still better than the `dumb lock` version.

It will really be a good practice to think about `read/write lock` and `node lock` and I heard that some of my classmates are talented enough to implement them in the given time limit. From my perspective, `node lock` is feasible and I can find related reference on the internet but I think there will be inevitable race condition in `read/write lock`. I also heard that the instruction solution used `read/write lock`. So I do hope that instructor can post the idea of using different lock strategy(**not the code**) after the deadline.