Project 2 Report

Jingyi Xie (NetID: jx95)

1. Implementation

1.1 Lock Version

In the lock version, I use a mutex primitive as a global variable and initialize it with `PTHREAD_MUTEX_INITIALIZER`. The idea is that whenever we need to manipulate the linked list or call `sbrk`, a lock is needed. To make it straightforward, my implementation is to acquire a lock immediately before calling `my_malloc` / `my_free` (implemented in project 1) and release it immediately after `my_malloc` / `my_free`. Therefore, the critical section is the code between `lock` and `unlock`.

```
meta_t *head_lock = NULL;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
//.....
void *ts_malloc_lock(size_t size) {
    pthread_mutex_lock(&lock);
    void * res = my_malloc(size, &head_lock, 1); //critical section
    pthread_mutex_unlock(&lock);
    return res;
}
void ts_free_lock(void *ptr) {
    pthread_mutex_lock(&lock);
    my_free(ptr, &head_lock); //critical section
    pthread_mutex_unlock(&lock);
}
```

Note: The third argument of `my_malloc` is the flag to distinguish between the lock and no-lock version. The only difference is when calling `sbrk`. If the flag is 1 (lock version), there is no need to use lock for `sbrk`. However, if the flag is 0 (no-lock version), we must use lock for calling `sbrk` because as mentioned in the project requirement, `sbrk` is not thread-safe.

1.2 No-lock Version

In this part, I used TLS (thread local storage) to implement the no-lock version. The idea is that it creates an independent linked list for each thread. As a result, each thread is totally independent from each other, so there will be no overlap in the memory. To implement this idea, first I declared another head `__thread meta_t *head_tls = NULL;` (which is different from `meta_t *head_lock = NULL;` used in 1.1). Then I modified all the function in project 1 which involve handling the linked list so that now they all take an argument to identify the head of the list to be changed.

```
void *ts_malloc_nolock(size_t size) {
    return my_malloc(size, &head_tls, 0);
}
void ts_free_nolock(void *ptr) {
    my_free(ptr, &head_tls);
}
```

As mentioned in 1.1, here I pass `0` as the third argument of `my_malloc` so that it will use lock to make `sbrk` thread-safe.

2. Performance Experiments

	Lock Time	Lock Seg Size	No-lock Time	No-lock Seg Size
1	0.135381 s	44000240	0.127264 s	44045344
2	0.113710 s	43351264	0.118295 s	43038064
3	0.111451 s	43499888	0.130829 s	44146064
4	0.108387 s	43225440	0.116654 s	43858160
5	0.142689 s	45112640	0.118015 s	42804096
Avg.	0.1223236 s	43837894.4	0.1222114 s	43578345.6

3. Result Analysis

- As is shown in the result, the run-time of the no-lock version is slightly faster than that of the lock version. The reason is that no-lock version only requires a lock for `sbrk`, so that other functions can execute at the same time. However, in the lock version, we put a lock on the entire `my_malloc` and `my_free` function, so it is actually one single thread with one really long linked list, which will no doubt increase the runtime.
- For segment size, I thought the no-lock version will have larger segment size because every thread has its independent linked list. However, the lock version turns out to have larger size. I think it has something to do with the fact that in the lock version, we reuse the same linked list. The length of this list keeps growing very fast, and the segment size will keep cumulating in the process.
- In conclusion, local thread storage improves the run-time of the no-lock version, while the lock version has larger segment size.