Report of Thread-Safe Malloc

Student name: Suo Chen
NETID:sc831

1. Description of thread-safe model

The best fit allocation policy is used to implement thread-safe malloc and free functions of lock-based version and no-lock version. If more than one thread works on a global doubly linked list, it might lead to race condition and cause the unexpected result. A race condition happens when the result of program relies on the timing or sequential order of occurrences. When many threads access shared resources at the same time, it will result in unpredictable and inconsistent outcomes. To prevent race condition from happening, two methods are used in my program. The first method is to ensure only one thread can modify the linked list at a time, which utilizes lock-based synchronization. The second method is to ensure every thread has its own linked list to modify, which involves no lock except for locks for sbrk().

To provide necessary synchronization for the locking version, a mutex lock supported from the pthread library is needed. pthread_mutex_lock() is placed before malloc and free functions, and pthread_mutex_unlock() is placed after malloc and free functions as shown in figure.1 . The mutex lock ensures that only one thread can modify the linked list at a time. This method provides a concurrency of malloc and free level.

```
void *ts_malloc_lock(size_t size){
    pthread_mutex_lock(&lock);
    void * res = bf_malloc_withlock(size);
    pthread_mutex_unlock(&lock);
    return res;
}

void ts_free_lock(void *ptr){
    pthread_mutex_lock(&lock);
    bf_free(ptr, free_head, free_tail);
    pthread_mutex_unlock(&lock);
}
```

Figure1. lock-based version

For no-lock version, as indicated by figure.2, Thread-Local Storage is used to make sure every thread has its own doubly linked list. Additionally, as sbrk() is not thread-safe, pthread_mutex_lock() is placed before sbrk(), and pthread_mutex_unlock() is placed after sbrk() as shown in figure.3. This method provides a concurrency of sbrk level.

```
__thread Blk * nolock_free_head = NULL;
__thread Blk * nolock_free_tail = NULL;
```

Figure2. use Thread-Local Storage

```
pthread_mutex_lock(&lock);
Blk * nb = sbrk(BLK_SIZE + size);
pthread_mutex_unlock(&lock);
nb->next = NULL;
nb->prev = NULL;
nb->size = size;
return (void *)((char *) nb + BLK_SIZE);
```

Figure3. thread-safe sbrk()

2. Performance result presentation & Analysis

Method	Average execution time /s	Average data segment size/bytes
Lock-based	0.16	48089500
No lock	0.12	48089536

It can be seen from the table that no-lock version runs faster than lock-based version. It is because that no-lock version allows a concurrency of sbrk level while lock-based version only allows a concurrency of malloc/free functions level. As for no-lock version, since every thread has its own linked list, operations like adding, removing and searching can be conducted simultaneously, only sbrk() is considered as the critical section. However, the overhead of using lock-based version is significant that a lock must be obtained by each thread before each malloc/free function can be executed, and before the lock is released, other threads must wait.

As for the results of data segment size, the average data segment size of no lock version is larger than that of lock-based version. The merging operation of no-lock version is conducted less times than that of lock-based version since even the adjacent block is freed as well, they will not be merged if they belong to different threads. Therefore, no-lock version is expected to have larger data segment size.

In conclusion, lock-based version could obtain a more coherent memory space at the cost of the overhead of mutex lock, while no-lock version has higher speed but larger data segment size.