Project Report

Implementation

For the thread-safe version of the malloc library, I implemented in two ways, one with lock-based synchronization using pthread_mutex_t, and the other with no-lock using thread-local storage. The total implementation is based on the previous non-thread-safe best-fit strategy.

Lock version:
This implementation is very easy and based on pthread_mutex_t. I first declared and initialized a pthread_mutex_t lock and then I put a pthread_mutex_lock before each malloc process and put a pthread_mutex_unlock after the process, and also similarly for the free implementation with one lock before and unlock after to prevent race conditions. The critical section is the section between the lock to prevent conditions where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events. The lock guarantees that we finish running the current malloc or free before anything else and allows concurrency at the malloc and free level. I also modify the parameters of each helper function to accommodate for the no-lock version.

```
// Thread safe malloc and free: locking version
void * ts_malloc_lock(size_t size){
    pthread_mutex_lock(&lock);
    void * result = bf_malloc(size, &head, &tail, 0);
    pthread_mutex_unlock(&lock);
    return result;
}

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

No-lock version:

This is mainly implemented through assigning unique thread-local storage for each thread. In this case, this means a separate linked list for free blocks for each thread. So I declared __thread block * head_thread and __thread block * tail_thread for this version's linked list head and tail and pass this type of head and tail to each helper function. This approach allows multi-threads to access the object without race conditions as each thread has its own unique object to assure thread-safe. I still used locks for the sbrk() by putting a lock before and unlock after thus allowing concurrency at the sbrk level as the function sbrk is not thread-safe.

```c
// Thread safe malloc and free: non-locking version
void * ts_malloc_nolock(size_t size){
    void * result = bf_malloc(size, &head_thread, &tail_thread, 1);
    return result;
}

void ts_free_nolock(void * ptr){
    bf_free(ptr, &head_thread, &tail_thread);
}
```

Performance Result Analysis:

I run the test several times to get the average results.

	Lock	No-Lock
Execution Time	0.210962 seconds	0.179930 seconds
Data Segment Size	42658904 bytes	42723240 bytes

We can see on average the execution time of the no-lock version is faster than that of the lock version. The reason for that is the no-lock version only puts one lock at the sbrk() function but the lock version puts locks for the entire malloc and free process. Thus, during no-lock version, other parts can run simultaneously to save runtime. For the data segment size, we can see the two versions are almost the same. That is because the two versions have similar processes to allocate new space, finding available free blocks, and free used blocks. Thus, for the execution time, the no-lock version may be better than the lock version but there is no strict better than for the data segment size.