ECE 650 Project #2 - Thread-Safe Malloc

1. Project Overview

This project aims to apply thread-safe strategies on malloc() and free() functions. Both of the thread-safe malloc and free functions are using the best fit allocation policy. Thread-safe strategies are useful in preventing race conditions while applying thread concurrency. Race conditions are a phenomenon that describes the outputs of a system or a process depends on uncontrolled sequence or timing of events. Two signals compete with each other to decide which one outputs first. As a coping strategy, thread-safe techniques ensure that one thread can only access a part of the codes by using locks. In this project, thread-safe malloc and free functions are implemented to make sure that multiple threads will not compete with each other to malloc or free a chunk in heap. Two versions of thread-safe malloc/free functions are implemented and compared to see their tradeoffs.

2. Malloc/Free Functions with Locks

In this version, mutual exclusion locks are used to avoid two threads from simultaneous reading or writing to the same chunk of resource in heap. One pointer points to the first free chunk. Another pointer points to the last free chunk. They are created as global variables that are shared between threads. To prevent two threads concurrently access these global variables, I use mutual exclusion locks that lock the malloc and free functions at entrance and release them after the critical malloc and free functions are executed. By doing that, I can ensure that only one thread can access these parts at a time. The specific implementation steps are as follows.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; //initialize mutual exclusion locks
//The thread-safe malloc function of version 1
void * ts_malloc_lock(size_t size) {
    pthread_mutex_lock(&lock); //create the lock
    void * p = bf_malloc(size, &first_free_block_lock, &last_free_block_lock, 0); //normally perform the malloc function, pass in a variable 0 to cancel the requirement of applying thread-safe lock on sbrk function as only one thread can use sbrk function at a time in this version.
    pthread_mutex_unlock(&lock); //release the lock
    return p;
}
//The thread-safe free function of version 1
void ts_free_lock(void * ptr) {
    pthread_mutex_lock(&lock); //create the lock
    bf_free(ptr, &first_free_block_lock, &last_free_block_lock); //normally perform the free function
    pthread_mutex_unlock(&lock); //release the lock
}
```

3. Malloc/Free Functions without Locks

In this version, locks will not be used. Instead, threads will have their own pointers that maintain a linked list of free chunks. It can be implemented by using Thread-Local Storage technique. I use __thread to embellish global variables as individual entities of each thread. In that case, each thread will have their own pointers that respectively points to the first free block and last free block. The values of pointers of each thread do not interfere with each other. In that case, each thread can maintain its own linked list. Linked lists do not have overlapped regions. The specific implementation steps are as follows.

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; **//initialize mutual exclusion locks to lock the region that applies the sbrk function**
//use Thread-Local Storage technique to embellish global pointers
__thread Metadata * first_free_nolock = NULL;
__thread Metadata * last_free_nolock = NULL;

//The thread-safe malloc function of version 2
void * ts_malloc_nolock(size_t size) {
    return bf_malloc(size, &first_free_nolock, &last_free_nolock, 1); // **pass in a variable 1 to require applying thread-safe lock on sbrk function as multiple threads can use sbrk function simultaneously in this version.**
}
//The thread-safe free function of version 2
void ts_free_nolock(void * ptr) {
    bf_free(ptr, &first_free_nolock, &last_free_nolock);
}

4. Experimental Results and Analysis

The results were obtained under the Linux system and recorded in the following table. Each result was calculated based on the average of ten testcases.

Version	Average Execution Time (s)	Average Data Segment Size (bytes)
With Locks	0.467	43023200
Without Locks	0.117	44925024

It can be viewed that the average execution time of version with locks is way greater than which of version without locks. The reason is that, if one thread locks the malloc function, other threads who want to access the malloc function are required to wait until the owner releases the lock, which can be time-consuming. While for version without locks, all threads can synchronously perform malloc and free functions on their own non-interference memory chunks. Only when two threads want to

apply for a new region in heap (using sbrk function), one thread need to queue for the previous thread to complete its task and release the lock.

For average data segment size, it can be observed that version with locks has relatively high allocation efficiency than version without locks. The reason is that threads of version with locks share the same free-chunk linked list, while threads of version without locks have different linked lists of nonoverlapping free chunks, making it hard to merge adjacent free chunks in non-locking version if adjacent chunks are distributed to different threads. Sometimes, a thread has to apply for a new region in heap, even if other threads have abundant free chunks to use.

As a result, two versions of thread-safe malloc and free functions have their own advantages and disadvantages. If we pay more attention on execution time, we'd better use the version without locks. Alternatively, if we have low memory heap, we may consider to use the version with locks. After all, these two versions are proved to be useful in preventing race conditions.