ECE650 HW1 Report

Student Name: Chen Gong

NetID: cg387

Implementation Overview

In this homework, two versions of the thread-safe malloc() and free() functions are implemented. They are lock-based synchronization and no-lock based synchronization separately. Both the versions are implemented under best-fit memory allocation policy.

In this assignment, the following sbrk function was used to allocate new memory for the program. However, the sbrk function is not thread-safe, it can cause race condition when two threads are simultaneously requesting memory using sbrk function and thus result in incorrect address for the allocated memory.

Lock-based Synchronization

In this version of thread safe implementation, mutexes lock is used to allow only one thread can access a certain piece of code at a time. As shown in the following code, pthread_mutex_lock function is used to obtain a lock when a thread calls the function. If there is no another thread locking the mutex, the lock can be obtained, and the code after the locking function will be executed. In this program, it is the best-fit memory malloc function. After the memory allocation is completed, the locked will be released, therefore, other threads can execute this malloc function. Furthermore, in this version, a sigle global shared free list is used to store the free memory spaces.

```
pthread_mutex_lock(&lock);
// critical section begin
void* address = bf_malloc(size, true);
// critical section end
pthread_mutex_unlock(&lock);
```

Non-locking based Synchronization

In version 2, the thread-safe implementation is conducted using the Thread-Local Storage. Lock is only used for sbrk function in this version, instead, a thread-local storage is used to provide a local free space list for each thread. Thus, race condiction can be avoided in this case. In this version, a thread local free list head is allocated for each thread using the following variable declaretion statement. Each thread would be able to have its private data that will not be shared by other threads, so the same variables and memory address will also not be shared among other threads.

```
__thread block *threadHead = NULL;
```

Performance Results

Results of the execution time and data segment size which is the total size of created data segment after the text program execution. The performance results are all obtained from running the thread_test_measurement.c random times with different number of threads and number of items for each thread to malloc.

Test 1

Number of thread: 4

Number of Items (each thread will malloc): 20000

Implementation Version	Execution Time (s)	Data Segment Size (byte)
Lock-based	0.110912	45454336
Nonlock-based	0.121781	45017920

Test 2

Number of thread: 10

Number of Items (each thread will malloc): 20000

Implementation Version	Execution Time (s)	Data Segment Size (byte)
Lock-based	0.233591	110801984
Nonlock-based	0.303104	108727072

Test 3

Number of thread: 4

Number of Items (each thread will malloc): 5000

Implementation Version	Execution Time (s)	Data Segment Size (byte)
Lock-based	0.023756	11609312
Nonlock-based	0.034961	10848800

From the above three tables, it can be observed that nonlock based situations always have an execution time larger than lock-based versions. This can be because the strategy of lock-based version locks the whole malloc and free function each time a thread requests memory while nonlock-based version only locks the sbrk new memory creating function. However, the thread-local storage requires more memory to store data for each thread, therefore, more overhead in initialization for private memories for each thread will be needed. Private memory initialization overhead can accumulate, thus, the execution time for non-lock based version is longer.

At the same time, data segment size for non-lock version is smaller than lock version. This can be because during lock, more data segment will be created while for private memory creation the created data is smaller.