Report for Homework 2 – ECE650

Introduction

To have a comprehensive understanding of Thread-safety in Clang, this project required students to study the synchronization technique in multithreading and practice follow it to implement a thread-safe Malloc Library. There are three main requirements in this project, which are lock version malloc() / free() implementations and no-lock version of malloc() / free() implementations, and Performance study report.

In this project, CLion 2021.2.3, and Ubuntu 18 LTS have been used to implements Malloc Library, and MS Words has been used to write this report.

This report mainly consists of two sections, the first section is the design, implementation, and test of the Lock and No-lock version Malloc Library, and the second section is Performance Results and Analysis of the Lock and No-lock version Malloc Library Implemented.

Design and implementation of Malloc() Library

```
-------------------------------------------------------------------------------------------------------------------
| LastTail   || head 1 |           | tail 1 || head 2 |           | tail 2 || head 3 |           | tail 3 || LastHead   |
|            ||        |           |        ||        |           |        ||        |           |        ||            |
|            || size 1 | content 1 | size 1 || size 2 | content 2 | size 2 || size 3 | content 3 | size 3 ||            |
-------------------------------------------------------------------------------------------------------------------
```

Code Block 1 the structure of Boundary Tags, dynamic allocation memory data structures

At the beginning, the Boundary Tags method has been chosen as the dynamic allocation memory data structures in this implementation, which is a Bidirectional Coalescing of implicit list, raised by Prof. Donald Knuth in 1973, in which structure, two tags, a head and a tail, are used to enclose the allocated memory space. These two tags save the total size of the memory spaces used (including themselves), which can be used as an offset in finding the adjacent segments. Meanwhile, the last bit of each tag is used to represent if this segment has been freed, due to the last two bits (LSB) are always 0, because of the 4-byte aligned in the memory allocation.

Figure 1 A Free block

Figure 2 An Allocated Block

At the same time, this program maintained a Free Block List which would use the free blocks' to store the prev_node_ptr and next_node_ptr (as shown in Figure 1), which would generate a doubly linked list for free blocks and this method would greatly improve the performance of malloc because the program would not go through the whole memory in the heap just need to find a free block.

Both First Fit and Best Fit implementations shared most of the codes, such as splitting the free space, increase the total allocated space by calling sbrk() when not enough free space, merging the

adjacent free regions into a single free region of memory, and also in the garbage collection, to shrink the total allocated size.

BF has O(n) and Ω(n) in each search because it would always go through all segments to find the min one which also meet the requirements. But there is an optimization way it can stop searching once it got the well fitted segment (the free segment has the same size of the space the allocation required).

For free() implementation, this function would set the both tags' last bit as 0 to implies that segment has been freed and insert the freed block into the Free Block List.

Finally, all functions have been tested step by step and using CLion's Mem View window in the debug mode to monitor if the memory behaviors as expected.

Design and implementation of Read-Write Lock Version

In conclusion, in the read-write lock version, read lock and write lock were implemented using binary semaphore to ensure the manipulation safety of Free List, and mutex surrounded the sbrk() system call to ensure the thread-safe because of the not re-entrance design of sbrk();

```
void freeList_reader_lock(){
    sem_wait(&freeList_r);
    readcnt ++;
    if (readcnt == 1) sem_wait(&freeList_w);
    sem_post(&freeList_r);
}
```

Code Block 2 freeList_reader_lock

```
void freeList_reader_unlock(){
    sem_wait(&freeList_r);
    readcnt --;
    if (readcnt == 0) sem_post(&freeList_w);
    sem_post(&freeList_r);
}
```

Code Block 3 freeList_reader_unlock

The read locks were assigned before each best-fit searching and unlock after each searching. Considering the searching in the Free List wound not update the list, the read lock design can improve the searching performance because it allows multiple threads searching at the same time.

In each the first read lock assignment, it would maintain a reader counter. If it is the first reader to hold the read lock, it would also hold a write lock to prevent manipulation of the Free List (Code Block 2). When the last read lock was released, it would also release the write lock (Code Block 3).

The write lock would hold before any manipulation of the Free List, and release after the manipulation finished.

Design and implementation of No-lock Version

In conclusion, in the no-lock version, Thread-Local Storage was to ensure the manipulation safety of Free List, and mutex surrounded the sbrk() system call to ensure the thread-safe because of the not re-entrance design of sbrk();

Each thread in no-lock version would use Thread-Local Storage to store its own Free List. Therefore, the manipulation of each Free List would not affect other's thread and ensure the thread safety.

```
static size_t** freeListRoot = NULL;
__thread size_t** localFreeListRoot = NULL;
```

Code Block 4 Declaration of Free_List_Root and Local_Free_List_Root

As shown in Code Block 4, the freeListRoot was declared with *"static",* but localFreeListRoot was declared with *"__thread"*. The *"static"* modifications (or no modification) would allow all thread using this variable, however, the *"__thread"* modification make a copy of this variable for each thread.

Performance Results and Analysis

Table 1 The Performance of lock version and nolock version

Thread Number		Read-Write Lock Version	Simple Lock Version	No-lock Version
4	Time (s)	3.779776	0.239584	0.187680
	Segment Size (bytes)	41992752	47449552	47449552
8	Time (s)	11.731621	0.525797	0.322593
	Segment Size (bytes)	83867376	94880528	94880528

The Simple Lock Version is just holding a mutex before each malloc and free and release it later. As shown in the Table 1, the Read-Write Lock version has the worst performance in the measurement. But the Simple Lock version and No-lock version do not have significant difference.

This might be caused by the different threads in BF searching (reading) return the same free block in the Free List, however, only one thread can write this free block, and other threads should rollback and search again or malloc a new space.
Furthermore, the read-write lock realization above is a read first design, which means that the reader has a higher priority than the writer, and writer should wait until all reader finished then

the writer can write. These two might be the main reasons for the bad performance in read-write lock.

The difference between the Simple Lock version and No-lock version might result from the cost of operations the mutex.

In conclusion, the No-lock version has a better performance than the two lock versions.