Project #2: Thread-Safe Malloc

ECE 650 – Spring 2020

See course site for due date

General Instructions

1. You will work individually on this the project.
2. The code for this assignment should be developed and tested in a UNIX-based environment, specifically the Duke Linux environment available at login.oit.duke.edu.
3. You must follow this assignment spec carefully, and turn in everything that is asked (and in the proper formats, as described). Due to the large class size, this is required to make grading more efficient.
4. You should plan to start early on this project and make steady progress over the next two weeks. It will take time and careful thought to work through the assignment.
5. This assignment deals with multi-threaded execution. Note that **bugs in multi-threaded code are often timing dependent (i.e. race conditions),** This means that incorrect parallel code may (sometimes often) result in correct execution due to the absence of certain timing conditions in which the bugs can manifest.

Implementation of thread-safe malloc library

In this assignment, you are to implement two different thread-safe versions (i.e. safe for concurrent access by different threads of a process) of the malloc() and free() functions. Both of your thread-safe malloc and free functions should **use the best fit allocation** policy.

In version 1 of the thread-safe malloc/free functions, you may use lock-based synchronization to prevent race conditions that would lead to incorrect results. These functions are to be named:

```
//Thread Safe malloc/free: locking version
void *ts_malloc_lock(size_t size);
void ts_free_lock(void *ptr);
```

In version 2 of the thread-safe malloc/free functions, you may **not** use locks, with one exception. Because the `sbrk` function is not thread-safe, you may acquire a lock immediately before calling `sbrk` and release a lock immediately after calling `sbrk`. These functions are to be named:

```
//Thread Safe malloc/free: non-locking version
void *ts_malloc_nolock(size_t size);
void ts_free_nolock(void *ptr);
```

To provide necessary synchronization for the locking version, you may use support from the pthread library and needed synchronization primitives (e.g. pthread_mutex_t, etc.). Also, don't forget about Thread-Local Storage -- you may find that useful.

In order to exercise and test the thread-safe malloc routines, pthread-based multi-threaded sample programs will be provided. These programs will create threads which will make concurrent calls to the thread-safe malloc and free functions. You are also strongly encouraged to create your own multi-threaded test programs to test your library code. Recall that concurrent execution means that multiple threads will call the malloc and free functions, and thus may be concurrently reading and updating any shared data structures used by the malloc routines (e.g. free list information).

These test cases (and a sample Makefile for your thread-safe library code) are provided as homework2-kit.tgz. The Makefile included in the test directory allows you to compile the tests to use either one of the two thread-safe versions (see the README.txt file for more details).

Note there is one test case that you will use for running experiments with the two different versions. This test case self-reports (1) the execution time of the test and (2) the allocation efficiency (i.e. the total size of the data segment after running the test). You should experiment with this test in order to assess the tradeoffs in terms of performance and allocation efficiency between your locking and non-locking thread-safe versions. This test case contains some randomized behavior, so you may need to run the test with each version several times to identify the typical behavior.

Your submitted code for the thread-safe malloc implementation will be tested against the provided sample test programs, as well as potentially additional tests. So you are encouraged to: 1) reason thoroughly through your implementation to ensure there are no race conditions, and 2) create your own additional test cases to further stress your thread-safe implementation.

As described below, you should include the source code in your submission. Additionally, you will submit a report. This first part of this report should describe your two thread-safe malloc implementations. This should include a description of where your thread-safe implementation allows concurrency. It should also describe any critical sections you identified, and your synchronization strategy to prevent race conditions. The second part of this report should show and discuss results from your experiments with the thread_safe_measurement.c test. Based on these results, discuss any tradeoffs you identify between the two versions.

Detailed Submission Instructions

You will submit this project as a single zip file named **proj2_netID.zip** to **Gradescope**. The zip file should contain exactly the following:

1) The report writeup called **report.pdf** addressing all items described above.

2) All source code in a directory named “**my_malloc**”
    - There should be a header file name “**my_malloc.h**” with the function definitions for the ts_malloc_lock(), ts_free_lock(), ts_malloc_nolock(), and ts_free_nolock() functions.
    - You may implement these functions in “**my_malloc.c**”. If you would like to use different C source files, please describe what those are in the report.
    - There should be a “**Makefile**” which contains at least two targets: 1) “all” should build your code into a shared library named “libmymalloc.so”, and 2) “clean” should remove all files except for the source code files. If you have not compiled code into a shared library before, you should be able to find plenty of information online, or just talk to the instructor or TA for guidance!
    - With this “Makefile” infrastructure, the test programs will be able to: 1) #include “my_malloc.h” and 2) link against libmymalloc.so (-lmymalloc), and then have access to the new malloc functions.