# Project #2: Thread-Safe Malloc

# ECE 650 – Spring 2025

**Due on 2/6 11:59 pm**

## General Instructions

1. You will work individually on this 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 or https://vcm.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.

6. You have a chance to preview your program results tested by our auto-grader if you submit your homework on Gradescope 2 days before the due date. Please see Auto-grader policy under Grading section for more information.

## 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. You may implement it based on your code of project 1.

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 (and you may not use things like semaphore), 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 in the lecture slides. 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_test_measurement.c test. Based on these results, discuss any tradeoffs you identify between the two versions.

Tips: Valgrind cannot be used to test the multi-threaded program. If you have successfully completed the single-threaded version in project 1 and have confirmed that there are no race conditions when running project 2 tests without Valgrind, then you should be in good shape.

## Grading

### Rubrics

- code correctness(60'), 20 test cases total
- code check(10'), in non-locking version, no lock except the lock for sbrk()
- report(30'), description of thread-safe model, performance result presentation and comparison of locking vs. non-locking version

This project does not require better performance but correctness under multi-threads environments.

### Auto-grader Policy

When you submit your code, you won't be able to see the autograder grade immediately. However, we will release a pre-grade of the autograder two days before the due date (on Feb 4 at 11:59 pm). If you'd like to view your autograder result, please make sure to submit your code before this pre-release date.

## Detailed Submission Instructions

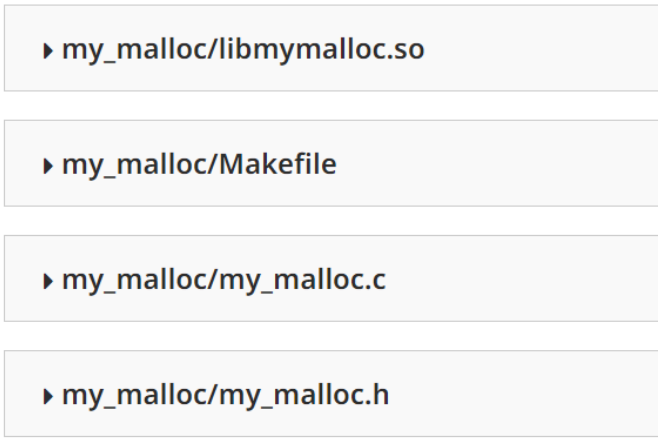
You will submit this project as a single zip file named **proj2_netID.zip** to **Gradescope**. The zip file should contain exactly the following:

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.

Ensure that the file structure you upload for Project 2 is exactly the same as the screenshot below; otherwise, it might cause the auto-grader to run incorrectly. Additionally, please submit your "libmymalloc.so" file!




## FAQ

1. Q: If my program works when the NUM_ITEMS is set 1K but doesn't work at 10k, do I have to optimize my code for proj2?
A: First, you need to make sure that you code work well under single thread model (lab1), not under valgrind env but your vm env. If you executed program under valgrind, please test your code directly on your vm. If it crushed under your lab1 env, you should check your code and find errors instead of adding lock and running in multi-threads env.

If your code crushed under multi-threads env (lab 2), please check your critical sections, your lock positions, address operations.
And please post your error details on Ed therefore we can locate your mistake sources.
Project 2 will not require better performance but correctness under multi-threads environment. So if your code works well in project 1, you can implement it based on your previous work.

2. Q: I'm getting a brk() segment overflow when running my ts_malloc_lock function. Immediately after that error, sbrk() returns 0xffffffffffffffff which causes a segfault. I've googled the error but haven't found anything particularly helpful. Any ideas on what I should investigate to resolve this issue?
A: Valgrind seems not workable with the multithread test program. If you've passed the single-thread version in project 1 and have no race condition when you run the project 2 test without Valgrind, then I think your code could pass the final autograder.

3. Q: I am struggling with project 2. I have implemented a locking structure that locks the head of the data structure and locks a node as the thread iterates through it. Ideally, it lets one thread into the data structure at a time, and when following threads are in, they cannot pass the thread ahead of it.
To ensure it is working, I started with making it wait a large number of steps before the next thread could enter the data structure, effectively locking the whole data structure for one thread at a time, and then reduced the number of steps behind a following thread would wait. For very small data structures, this would lock the whole data structure, but for large data structures multiple threads would be traversing simultaneously. Nevertheless, counterintuitively, as I reduce the distance between threads, the performance goes down (and the execution time goes up substantially).
I am also struggling with the no locking algorithm. The algorithm I have tried so far is having threads hop over other threads as they are working, but nevertheless, I can't seem to implement this effectively without using locks. Particularly, if a data structure is very small, there does not seem to be any benefit to having multiple threads in the data structure simultaneously, and I feel like the most effective way it to lock the thread out.
A: A mutex is utilized at the thread level to ensure serialized access to resources that are shared among multiple threads. When a thread needs to access shared data, it must first acquire (lock) the mutex before executing the critical section of code. And once done, the thread should release (unlock) the mutex, so that other threads can acquire it.
To correctly implement mutex, You should not rely on hard-coded timings for locking and unlocking. Instead, mutex operations should be integrated into your logic flow, ensuring that the mutex is locked before the critical section and unlocked after. This approach ensures consistent behavior regardless of the data structure's size.
For the no locking version, the idea is to make sure the rest of your code won't cause race conditions(threads do not access a shared resource at the same time) other than the 'sbrk' function.