CSCI 432 Lab 1 Report

Angus Li, Chris Brown

12-10-2023

1 Introduction

The goal of this assignment was to learn how to implement simple multi-threaded programs correctly, and also to understand how to implement locking/unlocking, waiting/signalling, and threads.

We first implemented a mock disk scheduler, which was given a fictional list of disks that made a series of requests and serviced them according to a shortest seek time first (SSFO) order. This was tested using a correct thread library binary provided by Jeannie.

We then began work on our own thread library, implementing the same basic interface. Programs initialize our thread library at the user-level, allowing them to create threads, manipulate locks, and wait, signal and broadcast using condition variables. We evaluated our work by writing a collection of small test programs, some of which were written from the ground up and a small number of which were based on our disk scheduler code.

2 Design

We focus on the design of the thread library. On the type level, we decided to implement thread queues, locks and condition variables as simple aliases for deques of `ucontext_t` pointers, unsigned ints, and signed ints[1] respectively. This allowed us to take advantage of implicit casting of user-passed parameters and iterate work on the library quickly. We globally track 4 important kinds of state: variables to track what thread is currently running and what thread to delete next, ready and blocked queues, a map between locks and the thread currently owning them,[2] and a simple boolean flag to check whether the library has been initialized.

Our ready queue is implemented using the C++ `deque` type, to allow both conventional queue operations and the ability to iterate through its elements. Our blocked queue is a map between locks and a inner map associating condition variables and threads. In this inner map, -1 represents the queue of threads that are just waiting on the lock. This is acceptable as unsigned ints passed by the

[1]We require the ability to represent -1 as a condition variable, for reasons explained later.

[2]If a lock is free, the map points to `NULL`.

user are implicitly cast and there is a bijection between signed and unsigned ints, meaning we lose no functionality from this change.

Helper functions are instrumental in our implementation, and much of our time was initially spent deciding what helper functions would be useful to include before we began to work on the public interface. These helpers fall into two main categories: internal implementations of public functions written for code reuse, and functions that provide two broad internal ”interfaces” for thread manipulation and blocking respectively. With regards to code reuse, `thread_libinit` and `thread_create` are almost entirely wrappers around a helper function that creates a thread, handles a potential malloc error and adds it to the ready queue. Similarly, we avoid a significant amount of duplication between `thread_signal` and `thread_broadcast` using this method. Most of our monitor functions, including signal and wait, use internal implementations of lock and unlock both to avoid repetition and to ensure important procedures happen safely.

To abstract away thread manipulation, we wrote helpers to initialize an empty thread context block, delete a thread, start a thread's function while handling cleanup, and run the next ready thread. These are used by libinit, create and yield to handle thread creation and cleanup, as well as to handle the ready queue in a safe way. Given the complicated structure of our blocked queue, we implemented internal functions to get the queue of waiters, block a thread and add the next waiting thread to the ready queue respectively for a given lock and a given condition variable (if any). This was particularly useful, as it abstracted away a process of unpacking the inner map which would have otherwise added a great deal of unnecessary complexity to the code.

3 Evaluation

A major requirement for this lab was to provide a suite of tests to verify the correctness of the thread library provided. We designed our test cases according to the broad use-cases of locking/unlocking, signalling/waiting and thread initialization/cleanup. Not all of our test cases triggered faulty libraries used by the autograder, but they were an important part of our development process and we've maintained them in our repository even though they were not a part of our final submission.

Throughout the initial phases of development, we used the `simple.cc` program provided in the helper code as a broad heuristic of correctness. This program turned out to be especially useful because it uses the majority of the public interface and prints output relating to each library function call, making it easier to pinpoint errors. After this process, we began running our disk scheduler with pre-emptions enabled on various outputs. Combined with `gdb`, this was useful in catching various subtle mistakes in the library's core loop of running ready threads. We submitted a number of tests based on `disk.cc` with pre-emptions enabled, which helped a great deal in increasing the percentage of faulty libraries used by the autograder that we were able to uncover.

A lot of productive debugging was initiated by our discovery of errors in our monitor functions, particularly with regards to locking/unlocking twice in a row and trying to publicly manipulate locks and condition variables that don't exist, or in the case of locks are not held by the caller. This led to major changes in our code, including the addition of helper functions to internally implement `thread_lock` and `thread_unlock` for use by the monitor functions. In the same manner, we discovered a major memory leak in the helper function `thread_run_next_ready`, the fix to which significantly improved our performance on Valgrind.

4 Conclusion

Our thread library was one of the most substantial single programs we have written to date. One key lesson is that the choice of data structure at the start of development can dramatically impact the experience of coding. Even though the structures themselves are likely to change as you progress, the choice to initially use a map to represent lock state instead of a vector, for example, influences the implicit assumptions you use in your helper functions and the rest of your code and can make it much harder to make changes if refactoring becomes necessary. A closely related observation is that powerful, *safe* helper functions are the key to success in a project of this nature, as they make it much easier to do the conceptual reasoning required to implement the public functions successfully. We've learnt that good abstractions are still relevant even at this relatively lower level of programming, and arguably even more so than in non-system code.

We particularly enjoyed the process of iterative design, where problems that confused us and seemed to suggest that added complexity was required turned out to have more elegant solutions that in many cases allowed us to simplify our library and re-use more code. For example, a problem with `thread_yield` led us to the realization that much of its code could be replaced with a call to `thread_run_next_ready`, making the library both conceptually clearer and more safe. However, debugging memory issues was challenging, especially since Valgrind output is often very difficult to parse. We hope that we learn more about this process as the semester continues. It was also nice to have the early effort we put into the disk scheduler rewarded, as it played an important role in our debugging of the library.