User-level Thread Library with Synchronization

Spring 2021 CSci 5103 Project 2

Final Submission Due March 17
Intermediate Submission Due March 10

1. Overview

In this project, we will extend our user thread library (*uthread*) from Project 1 to support synchronization and asynchronous I/O. The Project 1 implementation of *uthread* allowed users to create and join on threads but did not provide any mechanisms for synchronizing thread execution. To support thread synchronization, we will introduce locks, condition variables, and spinlocks to the *uthread* library. We will also modify the *uthread* implementation to prevent unbounded priority inversion in the presence of these new synchronization mechanisms. After implementing and testing the synchronization mechanisms, we will also conduct a performance comparison between lock and spinlock performance within *uthread*.

Additionally, in the Project 1 implementation, a blocking call would result in blocking the entire process. We will add the ability to make asynchronous read and write calls to perform I/O in the background while allowing other threads to run. We will also conduct a performance comparison between asynchronous and synchronous I/O.

2. Project Details

2.1 uthread Starter Code

A slightly adapted solution for the *uthread* library from Project 1 has been provided to you as starter code. The *uthread* starter code has been separated into public and private interfaces. The public interface (uthread.h) provides the API for users to interact with the *uthread* library. The private interface (uthread_private.h) provides an API for new synchronization code to access internal thread library functionality (such as switching threads, enabling/disabling interrupts, etc.). The private interface provided should be sufficient for implementing the new functionality, but you can add to or modify the interface if you wish. If you do so, provide the necessary comments to explain the changes.

A test case (main.cpp) is also provided to you for testing your lock and condition variable implementations. The test case runs a bounded buffer with producers and consumers running until interrupted (CTRL-C). It will print a message for every PRINT_FREQUENCY (100000) items that are consumed. The test case also includes an extensive set of asserts on the bounded buffer invariants to ensure that your implementation is not violating the expected thread synchronization behavior. You will write your own set of tests as described below. An executable (uthread-sync-demo-solution) of the solution for the producer-consumer test has been provided to you so that you can see the expected output.

2.2 Mutex/Lock (Lock)

You should implement the Lock class to provide a basic mutex synchronization mechanism. The Lock class defines a simple interface of *lock()* and *unlock()* (refer to Lock.h for more details).

A collection of private member function declarations are provided to you in Lock.h. You do not have to follow the format provided but it should give you a good starting point for organizing your Lock code when dealing with lock requests from both user-code and from internal modules, such as CondVar. CondVar is declared a friend class of Lock so that CondVar can access the private *_unlock()* and *_signal()* functionality provided by Lock.

2.3 Condition Variable (CondVar)

You should implement the CondVar class to provide a condition variable mechanism. The CondVar class defines an interface that supports *wait()* as well as *signal()* and *broadcast()* (refer to CondVar.h for more details).

Your CondVar implementation should use **Hoare semantics**. Hoare semantics guarantee an atomic transfer from a *signal()*ing thread to a *wait()*ing thread, and then a return to the *signal()*ing thread after the *wait()*ing thread has released the lock. You will likely need to modify both Lock and CondVar to support Hoare semantics.

You may wish to start by implementing CondVar using Mesa semantics since it is a simpler design. After implementing and testing CondVar with Mesa semantics, modify your solution to use Hoare semantics. If testing with Mesa semantics, ensure that you are using *while* loops when checking condition variable conditions in user-code (the provided main.cpp assumes Hoare semantics and therefore uses *if* statements).

2.4 Spinlock (SpinLock)

You should also implement the SpinLock class to provide an alternate locking mechanism. The SpinLock class should implement the same *lock()* and *unlock()* interface as Lock (refer to SpinLock.h).

To implement a spinlock, we need the ability to run an atomic instruction. We will do this in C++ by using the std::atomic_flag type. std::atomic_flag provides the *test_and_set()* method to perform an atomic operation.

2.5 Priority Inversion

With the addition of synchronization mechanisms, the current *uthread* implementation is susceptible to priority inversion. You should implement a solution that addresses priority inversion.

There are many ways to address priority inversion. One method, that is apparently used by Microsoft Windows (https://docs.microsoft.com/en-us/windows/win32/procthread/priority-inversion), is called Priority Boosting. Priority Boosting addresses the problem by periodically increasing the priority of low priority threads that are currently holding locks so that they will eventually be scheduled and hopefully release the lock. This is one solution that you can implement but you are free to implement any solution you would like. Describe your final implementation in the README file.

A priority scheduler and some helper functions have been provided to you to assist in implementing your priority inversion solution. The priority scheduler has three priority levels: RED, ORANGE, and GREEN (RED is highest priority and GREEN is lowest). The scheduler strictly follows priorities by scheduling a thread with the highest priority currently in the ready queue. Functions for modifying the priority of a thread have also been provided through the *uthread_increase_priority()* and *uthread_decrease_priority()* public functions and *_uthread_increase_priority()* and *_uthread_decrease_priority()* private functions in uthread.cpp. Additionally, if following the Priority Boosting methodology, the *increaseLockCount()*, *decreaseLockCount()*, and *getLockCount()* methods have been added to the TCB class for keeping track of how many Locks a thread is currently holding.

2.6 Asynchronous I/O (async_io)

You should implement the *async_read()* and *async_write()* functions to provide a mechanism for supporting asynchronous I/O (refer to async_io.h).

These functions will initiate an I/O request with the operating system and then switch to another *uthread* thread until successfully polling for completion. This means that the asynchronous I/O functions will be synchronous with respect to the calling thread but asynchronous in terms of letting other threads complete work in the meantime. The *aio_read()*, *aio_write()*, *aio_error()*, and *aio_return()* system calls (refer to their man pages) will be helpful for carrying out asynchronous I/O requests.

3. Test Cases

You should also develop your own test cases for all the implementations and provide documentation that explains how to run each of your test cases, including the expected results and an explanation of why you think the results look as they do. For example, you could build multiple demo applications that call all of the public API functions provided by Lock, CondVar, SpinLock, and async_io in separate tests.

4. Performance Evaluation

4.1 Lock vs SpinLock

After implementing the Lock and SpinLock, conduct a performance evaluation of the two types of locks to see how they compare. Your evaluation should include a new user-code test file using both Lock and SpinLock with multiple *uthread* threads.

You should also include a performance evaluation write-up in the README file (if you wish to include any non-text content in your performance evaluation just include any additional files and reference them in the README). Your performance evaluation should answer the following questions:

- Which lock provides better performance in your testing? Why do you think that is?
- How does the size of a critical section affect the performance of each type of lock? Explain with results.
- *uthread* is a uniprocessor user-thread library. How might the performance of the lock types be affected if they could be used in parallel by a multi-core system?
- Are there any other interesting results from your testing?

To evaluate the performance of each lock type, you may want to use a timer within C/C++. There are many timer options in C/C++ that you can choose from. One option that you can use if you would like is the std::chrono::high_resolution_clock provided by <chrono>.

4.2 Synchronous vs Asynchronous I/O

After implementing the asynchronous I/O functionality, conduct a performance evaluation of synchronous *read()* and *write()* calls compared to *async_read()* and *async_write()* operations.

Your evaluation should include a new user-code test file and a write-up in the README file (similar to the Lock vs Spinlock evaluation). Your performance evaluation should answer the following questions:

- Which I/O type provides better performance in your testing? Why do you think that is?
- How does the amount of I/O affect the performance of each type of I/O? Explain with results.
- How does the amount of other available thread work affect the performance of each type of I/O? Explain with results.

4.3 Priority Inversion

After implementing your priority inversion solution, create a test that demonstrates that your implementation addresses the problem. Explain your test and how your implementation works in the README file.

5. Deliverables

You should submit all source code, test cases files, Makefile, README (with performance evaluation), and any other additional files necessary to run or describe your implementation.

All files should be submitted in a single tarball (.tar.gz). The following command can be used to generate the file:

```
$ tar -cvzf submission.tar.gz project_folder
```

6. Grading

1. (Basic Rule: only verified code earn credits) Project owners are responsible for proving that their code could work. How? By calling all the functions in user-defined test cases. In real-life, the ratio between test code and functional code is 8:1 (anecdotally). We don't require students' code to achieve production-level quality. **But the minimum requirement is that every function should be called at least once.** Uncalled functions do not count, since they are not verified.
2. (Tentative grade breakdown)
    a. (+10) Intermediate Submission
    b. (+40) Lock, CondVar, SpinLock, Priority Inversion Handling, and Async I/O
    c. (+20) Test cases
    d. (+30) Performance evaluations/tests