Lecture Notes on Operating Systems

Lab: Reader-Writer Locks

Goal

In this lab, we will implement reader-writer locks, a special cases of regular locks or mutexes. We will implement these reader-writer locks using simple mutexes and condition variables.

Before you begin

Familiarize yourself with the `pthreads` or POSIX threads library, particularly with the synchronization primitives like mutexes and condition variables.

Implementing reader-writer locks

Consider an application where multiple threads of a process wish to read and write data shared between them. Some threads only want to read (let's call them "readers"), while others want to update the shared data ("writers"). In this scenario, it is perfectly safe for multiple readers to concurrently access the shared data, as long as no other writer is updating it. However, a writer must still require mutual exclusion, and must not access the data concurrently with any other thread, whether a reader or a writer. A reader-writer lock is a special kind of a lock, where the acquiring thread can specify whether it wishes to acquire the lock for reading or writing. That is, this lock will expose two separate locking functions, say, *ReaderLock()* and *WriterLock()*, and analogous unlock functions. If a lock is acquired for reading, other threads that wish to read may also be permitted to acquire the lock at the same time, leading to improved concurrency over traditional locks.

There are two flavors of the reader-writer lock, which we will illustrate with an example. Suppose a reader thread R1 has acquired a reader-writer lock for reading. While R1 holds this lock, a writer thread W and another reader thread R2 have both requested the lock. Now, it is fine to allow R2 also to simultaneously acquire the lock with R1, because both are only reading shared data. However, allowing R2 to acquire the lock may prolong the waiting time of the writer thread W, because W has to now wait for both R1 and R2 to release the lock. So, whether we wish to permit more readers to acquire the lock when a writer is waiting is a design decision in the implementation of the lock. When a reader-writer lock is implemented with *reader preference*, additional readers are allowed to concurrently hold the lock with previous readers, even if a writer thread is waiting for the lock. In contrast, when a reader-writer lock is implemented with *writer preference*, additional readers are not granted the lock when a writer is already waiting. That is, we do not prolong the waiting time of a writer any more than required.

In this lab, you must implement these two flavors of the reader-writer lock. You must complete the definition of the structure that captures the reader-writer lock in `rwlock.h`. The following functions to

be supported by this lock are also defined in the header file:

- The function `InitalizeReadWriteLock()` must initialize the lock suitably.
- The function `ReaderLock()` is invoked by a reader thread before entering a read-only critical section, and the function `ReaderUnlock()` is invoked by the reader when exiting the critical section.
- The function `WriterLock()` is invoked by a writer thread before entering a critical section with shared data updates, and the function `WriterUnlock()` is invoked by the writer when exiting the critical section.

Next, you must write the code to implement these functions. You must write code that implements reader-writer locks with reader preference in `rwlock-reader-pref.cpp`. Similarly, you must implement reader-writer locks with writer preference in `rwlock-writer-pref.cpp`. Note that both implementations of the lock must share the same header file, including the definition of the reader-writer lock structure. Therefore, your lock structure may have some fields that are only used in one version of the code and not the other.

Testing your code

You have been provided with a C++ program `tester.cpp` to test your reader-writer lock implementation. You can compile and link this tester with both flavors of the lock separately to create two different executables of the tester, as follows.

```
$g++ tester.cpp rwlock-reader-pref.cpp -o rwlock-reader-pref -lpthread
$g++ tester.cpp rwlock-writer-pref.cpp -o rwlock-writer-pref -lpthread
```

The tester programs takes two command line arguments, say R and W. The program then spawns R reader threads, followed by W writer threads, followed by R *additional* reader threads. Each thread, upon creation, tries to acquire the same reader-writer lock as a reader or writer (as the case may be), holds the lock for a long period of time, and finally releases the lock. The tester program prints a message to the screen when it acquires and releases the lock, and the relative ordering of these print messages will help us verify the correctness of your implementation. By invoking this tester with different values of M and N, one can test the reader-writer reasonably thoroughly. Of course, you are encouraged to write your own test programs that use the reader-writer lock as well.

We have also provided you with an autograder script `autograder.sh`, which does the following. This script compiles and tester program with both flavors of the reader-writer lock, runs the two tester executables for different values of M and N, and stores the print output. It then runs the autograder scripts `autograder-reader-pref.py` and `autograder-writer-pref.py` on these outputs. These Python scripts parse the output printed from the tester program, and ensure that all the threads have successfully acquired and released locks in the expected order.

Submission instructions

- You must submit the files `rwlock.h`, `rwlock-reader-pref.cpp`, and `rwlock-writer-pref.cpp`.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.

Grading

We will check correctness of your code using the test scripts provided (with additional test cases beyond those provided to you). In addition to using the test scripts, we will also read through your code to ensure that you have adhered to the problem specification in your implementation.