ECE 420 Lab 2 Report

February 25, 2021

Description of Implementation

Single Mutex

Main1 utilizes the simplest form of critical section protection used in this lab; a single mutex protecting the entire array. Protection of the array is accomplished by calling pthread_mutex_lock and pthread_mutex_unlock to obtain and release the lock respectively. A lock is required whether reading or writing to the array. This simple form of protection offers the guarantee of safety and data consistency though can lead to slower execution times as threads wait to obtain the single lock.

Multiple Mutexes

Main2 builds on Main1 however instead of a single mutex an array of mutexes of the same size as the array of strings is now used. This allows each string in the array to be protected by its own mutex. The same phtread_mutex_lock and pthread_mutex_unlock are used to obtain or release a lock on the mutex though now the position of the string is now needed to index the proper mutex for the string the thread wishes to operate on. This type of critical section protection allows upto N threads to execute in parallel, with the condition that they must be accessing unique indexes of the array, reducing the amount of time spent waiting to acquire locks.

Single Read/Write Lock

The Main3 program utilizes a single read/write lock to protect the entire array. Protection is achieved by calling pthread_rwlock_rdlock to obtain a read lock, pthread_rwlock_wrlock to obtain a write lock and pthread_rwlock_unlock to relinquish the obtained lock. Read/Write locks allow multiple threads to obtain a read lock simultaneously while a write lock can only be obtained when no other threads have an existing lock. This type of protection is well suited to applications where reads dominate the access requests as it requires less waiting then a mutex since reads can safely occur simultaneously.

Multiple Read/Write Locks

On the Main4.c program, this focuses on the implementation of multiple read/write locks, each protecting a different string. This implementation is one of many to protect the critical section. This is done by a specific pthread_rwlock_wrlock to obtain a write lock and pthread_rwlock_unlock to relinquish the obtained lock. But since we did have to focus each on protecting a different string, we needed to treat the locks as an array and used for loops when initiating, and destroying the threads.

Performance Discussion

Average Memory Access Latency

Implementation	Average Memory Access Latency Over 100 runs (s)		
	N = 10	N = 100	N = 1000
Single Mutex	6.913462e-310	6.923087e-310	6.948619e-310
Multiple Mutexes	6.897923e-310	6.907520e-310	6.922539e-310
Single Read/Write Lock	6.942035e-310	6.927068e-310	6.922237e-310
Multiple Read/Write Locks	6.898698e-310	6.920859e-310	6.950186e-310

Table 1: Average Latency times of the four protection schemes

Discussion

As can be seen in table 1 above when the array is small, N = 10, the multiple mutexes and multiple read/write locks perform better than their single lock implementations. This is likely due to the increased number of collisions that will occur when using a single lock on a small array and the relatively low added overhead of maintaining 10 locks versus 1 lock. Having each individual array entry protected by its own lock will significantly reduce this number of times threads contend for the same lock and in turn reduce idle thread time.

However as the size of the array grows the single read write lock performs better than multiple read/write locks because the added overhead of multiple locks grows as the array size increases and read/write locks allow an unbound number of threads to read at once with the only limitation being a single write will block all other operations. As the client is executing 70% reads and 30% writes far fewer writes will exist which allows the single read/write lock to outperform multiple read/write locks when the array is sufficiently large. If the fraction of writes were higher the multiple read/write locks should regain the performance edge over a single lock.

In all of the tests performed multiple mutexes were superior to a single mutex. This can be explained as a single mutex will only allow a single thread to operate on the array while multiple mutexes could result in as many as N threads operating concurrently. Given that our threads do little other than operate on the shared array, a single mutex is inefficient and results in threads waiting.