Project 3: Reader/Writer Locks Report

Param Chokshi

University of South Florida

Department of Computer Science and Engineering

The objective of the project is to build an implementation of Readers/Writers lock using semaphores that do not starve either of the writing or reading threads. Reader/writer locks help in implementing a program in which critical sections can be accessed by multiple reader threads concurrently, but only by one writer thread at a time.

The implementation provided here uses three semaphores, each initialized to one. Functions to acquire and release locks for writer threads and reader threads are defined separately. One of the semaphores is used to provide mutual exclusion (only one thread can access the section at a time) within these functions. Mutual exclusion is used to protect the increment and decrement operations on the variable used to track the number of readers. The second semaphore is used to make sure that either multiple readers access the critical section or a writer thread access the critical section. Lastly, a third semaphore is used to make sure that a waiting writer gets a fair chance to be executed. Both the reader and the writer thread wait on this semaphore before executing. This helps in scheduling a waiting writer with a fair chance. The fairness of the algorithm depends more on the scheduling policies.

```
Psuedo-Code:

Struct rwlock_t{

Lock // lock used for mutual exclusion

writelock // lock used for scheduling either the readers or the writer

readlock // lock used to make sure that the writer doesn’t starve

Readers // stores the number of readers.

}

rwlock_init(rwlock_t *rw){

Initialize lock semaphore with value 1

Initialize writelock semaphore with value 1

Initialize readlock semaphore with value 1

Readers <- 0

}

rwlock_acquire_readlock(rwlock_t * rw) {

  readlock.wait() // acquires the readlock
```

```
  lock.wait() //provides mutex
  rw -> readers++
  if (rw -> readers == 1)
       writelock.wait() // first reader acquires the writelock
  readlock.post() // releases the readlock
  lock.post()
}
rwlock_release_readlock(rwlock_t * rw) {
  lock.wait() // provides mutex
  rw -> readers--
  if (rw -> readers == 0)
       writelock.post() // last reader already in the critical releases writelock
  lock.post()
}
rwlock_acquire_writelock(rwlock_t * rw) {
  readlock.wait() // acquires readlock, gets added to the readlock queue. In the
//queue any of the reader/writer thread can be scheduled implementing a fair policy
  writelock.wait() //acquires writelock
  readlock.post() //releases the readlock
}
rwlock_release_writelock(rwlock_t * rw) {
  writelock.post() //releases writelock
}
```

To test the implementation a driver function is used which creates reader and writer threads by reading from a file. Here each thread executes a dummy function which executes for some time. The implementation provides adequate proof of a non-starving implementation using multiple scenarios with 10-15 concurrent threads.

Estimated Time spent: 5-10 hours