Project 3: Reader/Writer Locks

Miguelangel Rodriguez

Miguelangel Rodriguez

Introduction

In project 3 the Reader-writer problem arises. The main concerns generated by this problem is that there has to be a flexible locking system in order to implement the use of semaphores and handle cases such as many readers accessing a data path meanwhile a writer operation want to perform a modification to the data that is being accessed. In the Reader-writer problem, it is known that many readers can access he data at the same time, what is significantly important is to keep track of how many readers are inside the critical section. At this point is where semaphores come handy because we can use them to lock as the readers access the data so that no writes come in until the lock is removed. It is important to understand that it is extremely thanks to this locking system many concurrency issues can be avoid. On the other hand, there is a disadvantage to this system, and it is that if a write wants to come in and access the critical section while readers are executing. What this will cause is for the writes to have to wait until the very end, this behavior is defined as writer starvation. In order to avoid, a better approach to this issue will be demonstrated thought this project.

Pseudo-code

Main.c

Create struct rw

Init rw

loop

    Load scenarios to array

End loop

If File == null

New pthread = threads[macsizethread]

Loop threads in array

    If threadtype = r

        Create pthread passing threat, calling emulator reader function, and pasing rw.

    Else if = w

        Create pthread passing threat, calling emulator writes function, and pasing rw.

End loop

New loop (joing threads)

    Joint threads from 0 to maxsizethreads.

End loop

Close file

End progam.

Readerwriter.c

Reader non-starve solution:

```
rwlock_acquire_readlock(&rw);
wastetime()
rwlock_release_readlock(&rw);
```

Writer non-starve solution:

```
rwlock_acquire_writelock(&rw);
wastetime()
rwlock_release_writelock(&rw);
```

```
typedef struct {
  sem_t lock;
  sem_t writelock;
  sem_t avoidStarve;
  int readers; //
} rwlock_t;


void rwlock_init(rwlock_t *rw)
  rw->readers = 0;
  sem_init(lock);
  sem_init(writelock);
  sem_init(avoidStarve);

void rwlock_acquire_readlock(rwlock_t *rw)
  sem_waitavoidStarve);
  sem_postavoidStarve);
  sem_wait(lock);
  rw->readers++;
  if (rw->readers == 1)
     sem_wait(writelock);
  sem_postlock);

void rwlock_release_readlock(rw)
  sem_wait(lock);
  rw->readers--;
```

```
  if (readers == 0)
    sem_post(writelock);
  sem_postlock);

void rwlock_acquire_writelock(rw)
  sem_wait(avoidStarve);
  sem_wait(writelock)


void rwlock_release_writelock(rw)
  sem_post(avoidStarve);
  sem_post(writelock);

Void emulatReader
   rwlock_acquire_readlock
   wastetime()
   rwlock_release_readlock
   return NULL;


void emulatWriter
   rwlock_acquire_writelock
   wastetime
   rwlock_release_writelock
   return NULL;
```

Approach

In order to solve this project, the resources provided by the professor were the most helpful. The lecture in canvas helped to understand the main problem and clarify several questions that I had regarding the code provided in chapter 31. In addition, the book and the link to GitHub provided a strong background for testing purposes that enriched the understanding of the problem. Moreover, when it came to the improvement of this problem, I started by creating a new semaphore which main purpose was to prevent the writers to starve.

When a reader comes in the counter for readers increments to keep track of reads inside the critical section, is a writer wants to come in a lock is place and when the counter. Of readers in the critical path have been decreased to zero the writer comes in an places a lock to avoid readers or writers to come in. as soon as it is done and exist the data path the remaining threats can keep coming in. the main difference to the other solution is that in this case the writes will not starve since it is places in the quote and will not allow readers to keep accessing.

Conclusion

Reader-writers problem improved my understanding on how semaphores and locking systems works. In addition, it demonstrates the importance of this systems to avoid concurrency issues and misleading used of data. Without them, it would be risky trusting a database that can have multiple writers at the same time or reader obtaining information that was supposed to be updated. Finally, it was a pretty interesting topic and a better way to learn material for our next exam. I was more encouraged to read the book and research about the topic. Overall this project took me about 14 hours, which were divided in about 8 hours reading the book and the little book of semaphores, and 6 hours debugging, testing, and developing the code.