Alexander Cook

Operating Systems

4/2/2020

Project 3 Report

Description of problem:

Consider multiple threads accessing the same memory location over time. Some threads must read the data, and others must write. The simple solution is to implement total mutual exclusion i.e. only 1 thread may access the memory at once. This is non-ideal as reads do not change the data. Instead allowing either many reading threads or a single writing thread in the critical section at once takes advantage of concurrency.

In practice the simple implementation of this solution requires all reading threads to exit the critical section before a writing thread is allowed in. Thus, it is possible for writers to "starve" and be blocked from writing indefinitely.

Example pattern of reads and writes:

rwrr

This leads to a possible read write sequence of rrrw. The write may have to wait for the first read to finish, and in that window additional reads arrive further delaying the write.

A possible solution to this problem is to set up a "gate" that reading threads must pass through before reading. If a writer needs to write, it can simply close the gate and wait for the remaining reading threads still inside the critical section to finish.

Pseudocode:
4 semaphores each initialized to 1

```
rmutex, wmutex, readTry, resource
```

2 variables to count the number of reading/writing threads active in their functions both initialized to 0

```
readers, writers
```

4 functions

```
acquireReadlock
     wait(readTry)
     wait(rmutex)
     readers++
     if first reader, wait(resource)
     post(rmutex)
     post(readTry)
```

```
releaseReadlock
     wait(rmutex)
     readers--
     if reader is last one to stop reading, post(resource)
     post(rmutex)

acquireWritelock
     wait(wmutex)
     writers++
//next is the key line, this writer function can control the
flow of new readers by controlling the readTry semaphore used in
acquireReadlock
     if first writer, wait(readTry)
     post(wmutex)
     wait(resource)

releaseWritelock
     post(resource)
     wait(wmutex)
     writers—
//the last writer allows readers to continue
     if writer is last one, post(readTry)
     post(wmutex)
```

Using this new method, the thread read-write pattern rwrrrrrr will go differently.

Once the writing thread calls acquireWritelock() it is guaranteed no new reads can skip ahead of it. It is possible that in the time w is calling acquireWritelock() one or more reading threads can pass the readTry block before it can block them resulting in an access pattern like:

rrrwrrrr

This is non-deterministic, though it ensures a finite time to write rather than an indefinite time before.

Another important point here is that a series of writes can now starve reads. Considering writes are likely less common than reads this is less concerning.

Estimation of time spent: 8 hours

Sources: https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem