Parker Hague

Professor Name

CS5323

27 March 2022

Reader Writer Problem

I came up with a fairly simple solution to the reader writer problem that also handles fairness. First, the base of my solution consists of a typical reader writer solution where we have a read\_count and we lock and unlock a semaphore based on the value of the read\_count. The read\_count variable has to be adjusted inside a critical section otherwise a race condition could occur on the value. In a typical reader writer problem when the read\_count is 1, we block the writer from writing. We would usually allow n readers to enter as long as the writer is still locked. When the last reader is done reading, we usually signal the writer to start its work. My solution uses a lot of these ideas, but now I’ll discuss where several of my differences are.

Handling fairness was the trickiest part of this assignment. If the program uses the default reader writer solution, then the writer will almost always finish first and the readers starve. To fix this, I had to regulate how many readers were allowed to read before I forced them to stop. I decided to let my number of readers be, k + 1, where k is half the value of the total readers. So if a user wants there to be 10 readers, then my program will allow 6 readers to enter and do work before they’re forced to stop for the writer. To me, this solution seemed fair as more than half the readers are immediately being serviced and the writer is constantly getting attention in between the readers. There is usually a point where the writer finishes before some of the readers, but I think this is okay because the previous work was split up and the writer was never starved for at least more than half of the reader’s work. Also, the vast difference in work size between the readers and writer makes it completely normal and expected that the writer may finish before some of the readers.

I had to make similar changes to handle fairness for the writer. The writer is prone to want to finish its work super fast (especially compared to the readers). So to solve this, I had to not allow the writer to call post on it’s semaphore if a certain amount of work had been achieved. This is one of the most important parts of the program as the writer always finished first until I implemented this check.

To implement the extra blocking of the readers I had to add another semaphore. I set the initial value of this semaphore to be k + 1, where k is the half value of the number of readers. I then call wait on the semaphore and I put it at the beginning of the reader function. This means it will only let k + 1 number of readers begin to execute the function before the rest are blocked. At this point, the reader and writer then alternate taking turns and new readers are only able to enter as other readers finish their execution. Overall, my solution used 1 mutex lock and 2 semaphores. The reader semaphore is a counting semaphore and the writer one is a binary semaphore.