COP4600 Wanwan Li
Uliana Ozerova, U17965448
Project 2
Date: 10/15/2022
Total Time Spent: ~ 9-10 hours

Project 3
Concurrency
Non-starving Reader-Writer Problem

Introduction

Reader-Writer problem is an example of a common computing problem in concurrency, which describes a situation where some data or file system is read and updated or written by multiple threads. For the program it is important to prevent reading when the data is being updated, and not let the program modify it simultaneously. With this, we define two main constraints:

1) Writers have exclusive access(no other writer or reader are working at the same time) to the critical sections.
2) Multiple readers are allowed to read, but have to finish reading before writer starts writing.

The original first and second solution can starve either writers or readers, so our goal is to create a starve-free Reader Writer solution and avoid deadlocks.
So, additionalluThe problem requires understanding of locks, semaphores, logic behind concurrency and threads. The program will be checked using “scenario.txt” files, that consists of the order in which readers and writers arrive such as: rrrrwwwrwrwrrrr.

Solution

In the immediate original solution using two semaphores(one “writerlock” that is accessed by both readers and writers and “readerlock” that is only used by readers), depending on the implementation, it is possible for writers to starve. It happens because there is nothing stopping readers from blocking the writers and making them wait indefinitely. One of the most common solutions is to use an additional semaphore-turnstile that can be locked by writer when they arrive will help to achieve fairness among readers and writers(Little Book of Semaphores).
So here is in total three semaphores: readerlock, writerlock, and turnstile. All initialized to 1 in the beginning.

Turnstile wraps around both readers and writers functions, guaranteeing the entry of each process to operate within the critical section. In other words, it remembers the order of arrival "to the gate" and ensures that no process is inherently prioritized, allowing writers to lock and unlock it once done. Once unlocked, readers will be able to proceed as queued until another writer arrives and lock the turnstile again, at which point it will wait until the process that is currently accessing the critical section is done.

To ensure that my program works as expected, that is – it meets the requirements and constraints, I have created a shared variable that is updated every time with the write operation, and just read with the reading operation. That helps to follow what each thread is doing.

Pseudocode

Writer:	Reader:

wait(turnstil);
wait(writerlock);
--critical section--
[update s*hared]*
---------- 	
signal(writerlock);
signal(turnstile);
	wait(turnstil);
	wait(readerlock);
	if (readingCount++)==1 then wait(writerlock)
	signal(readerlock);
	signal(turnstil);
	--critical section--
	[read shared variable]
	---------- 	
	wait(readerlock);
	if (readingCount--)==0 then signal(writerlock)
	signal(readerlock);

Conclusion/Discussion

Using three semaphores instead of two, which is offered by the most simple solution and might lead to deadlock and starvation, we were able to achieve the goal of fairness among the two: readers and writers. The algorithm does not prioritize neither. However, there could be seen some downsides such as the reader having to lock two semaphores to enter the area of operations.

Although this solution is starvation free and has no prioritization, it might not be the best solution for every case. Sometimes it is useful to prioritize readers or writers depending on the goal of the project.

I spent the total number of 9-10 hours on this project, including researching, studying, developing, testing, experimenting with different solution, and the report.

Throughout the project, I have referenced "*The Little Book Of Semaphores",* our professor COP4600 Wanwan Li presentation and slides, as well as our class textbook "*Three Easy Pieces".*