McGill

Programming Assignment #2: Readers-Writers Problem

Due date: Check My Courses`

1. Why this assignment?

Synchronization is an important part of modern computer applications that have multiple threads of execution within them. For example, an Internet browser tool such as Mozilla Firefox would have multiple threads to manage various user and system tasks simultaneously. Most operating systems and programming languages provide some primitives for synchronization. Java, for instance, provides several high-level constructs for synchronization. C, on the other hand, does not provide any constructs by itself. However, there are various library-based or OS-based solutions for synchronization that can be invoked from a C program. In this assignment, you learn how to build high-level constructs using some basic primitives supported by Pthreads/UNIX within C.

2. What is required as part of this assignment?

In this assignment, you are expected to solve a readers-writers problem, where we have threads representing readers and writers. The readers want to read a shared data object while the writers want to update the shared data. Two readers can access the data concurrently while a writer needs exclusive access. That is a writer cannot concurrently access the data with another writer or reader. The readers-writers problem has several variations involving priorities. The first readers-writers problem requires that no reader be kept waiting unless a writer has already obtained the permission to write and is in the process of doing so. As soon as the writer completes the update process, the reader should be allowed to proceed. A solution to the readers-writers problem with readers having priority can have the writers suffering from starvation. That is, a writer could be waiting indefinitely to perform an update.

Here is a solution for the readers-writers problem given by the textbook. You need to implement this pseudo-code using pthreads and POSIX semaphores for shared memory. The example code given with the assignment shows how to use pthreads and semaphores.

In the proposed solution, the reader and writer threads share the following data structures.

```
semaphore rw_mutex = 1;

semaphore mutex = 1;

int read_count = 0;
```

The semaphore **`rw_mutex`** is common to both reader and writer threads. That is both threads use them. The **`mutex`** semaphore is used to ensure mutual exclusion when the variable **`read_count`** is updated. The **`read_count`** variable keeps track of how many threads are currently reading the object. The semaphore **`rw_mutex`** functions as a mutual exclusion semaphore for the writers. It is also used by the first and last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections. The code for the writer thread is shown below.

```
do {

     wait(rw_mutex);

           ..

     // write is performed
```

```
            ...
            signal(rw_mutex);
            } while (true);
```

The code for the reader thread is shown below. Note that if a writer is in the critical section and *n* readers are waiting, then one reader is queued on **`rw_mutex`** and *n*-1 readers are queued on **`mutex`**. Also observe that, when a writer executes **`signal(rw_mutex)`**, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the underlying scheduler.

```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1)
          wait(rw_mutex);
     signal(mutex);

     // read is performed

     wait(mutex);
     read_count--;
     if (read_count == 0)
          signal(rw_mutex);
     signal(mutex);
} while (true);
```

In this assignment, you create a program where the readers and writers are represented by threads. Create 10 writer threads and 500 reader threads. There is a shared variable (just an integer) that is the target of the readers and writers. A writer would increment the variable by 10 every time it gets access to it. The reader would simply read the value. When a reader or writer thread gains access to the shared variable, it is going to sleep for a random amount of time between 0 to 100 milliseconds and then try again. The number of iterations a reader or writer thread would attempt to access the shared variable should be given as an input parameter for the program. That is, a common parameter for all threads.

1. Write the program implementing the above pseudo-code for the readers-writers problem. Run the program. Reader repeat count is set at 60 and writer repeat count is 30. Measure the waiting times of the readers and writers. Find the minimum, maximum, and average of the waiting times. Does your implementation have a starvation problem? Briefly explain your answer. That is, explain why you concluded that there is starvation or no starvation.
2. If there is no starvation, rerun the tests with different parameters so that starvation shows up. If it is not possible to run into starvation, explain why it is not possible to run into starvation.
3. Modify the program so that starvation problem is solved. Remember in this version of the readers-writers problem we can have writers starving. So, you create modification to the pseudo-code provided here to solve the starvation problem.
4. Repeat the above tests and show that the starvation problem is solved.

See attached example program for pthread and semaphore usage in Linux. This may not work in OS X or other variations of UNIX.