Project 3: Reader/Writer Locks

Due date: Thursday, Apr 6th 11:59 pm, on Canvas.

In this project, you are to:

a. Design and implement a readers/writers lock using semaphores that do not starve the readers and do not starve the writers;
b. Write the main C program that uses Reader/Writer locks;
c. Come up with a set of input scenarios that shows the behavior of your non-starving lock compared to the starving lock.

The readers/writers problem is presented in section 31.5 of the textbook along with the readers/writers lock that starves writers. There may be solutions published in articles over the years – feel free to consult and **cite** them. It is ***not acceptable*** to use code that implements this solution from the Internet or other sources other than your own work. It is acceptable to use the code from the textbook as your starter code, if useful.

This is an **individual** project.

The output of this project includes:

- A report (in PDF) that includes:
  - Your name.
  - A short description of the problem you address (hopefully not cut-and-pasted from this project but written in your own words).
  - A description in plain English and pseudocode of your solution.
  - An estimation of the time you spent working on the project.

  This is likely to be a very short report – do not try to make it longer than it needs. But do make it neater than usual.

- A `tar` file that includes:
  - A `makefile` for easy compilation. The target executable should be `rwmain`.
  - A `README` file that describes how to run your project (arguments, etc).
  - A C program, named `readerwriter.c,` that implements the nonstarving locks.
  - A C program, named `main.c,` that uses the locks and shows their functionality. Note that the reading/writing parts of the code are only simulated: you do not have to read or write a particular data structure; instead, you might want to pretend to do it, and take some time such as:

```
reading_writing(){
    int x=0, T = rand()%10000;
     for(i = 0; i < T; i++)
                for(j = 0; j < T; j++)x=i*j;
}
```

  This function is only meant to waste time for a variable amount of time. Feel free to adjust this code as you see fit (or ignore all together if not useful in your solution).

There are no required inputs to this program. You may use inputs if useful and explain use in the readme file. Output messages useful for testing and debugging.

- An input file, named `scenarios.txt`, that proves that your lock:
  - Is a correct readers/writers lock.
  - Does not starve the writers.
  - Does not starve the readers.
  - Each scenario takes one line (as the traces file in last project) ▪ An example and interpretation of this file is below:
    rwrrrrwrr wwrrrrwr
    This file contains two scenarios:
    1. One in which one reader arrives first, then a writer, then four more readers, another writer, then two more readers.
    2. And the second in which two writers arrive first, then four readers, one more writer, and one more reader.
  - You want these scenarios to test corner cases that are relevant for the point of your design: specifically, that writers will not starve. Thus, you design these test scenarios to make it possible for writers to starve. You do not need lots of readers/writers to make the case. You might want to limit each scenario to 10--15 readers and writers at the very most.

To ease the task of grading, please name your files as requested and put all these files in the **same** folder. In addition, please hardcode the file name for scenarios.txt with the **relative path** (not the absolute path) in your main.c code, e.g.:

FILE ptr = fopen("scenarios.txt ","r");*

Suggestions on how to approach this project:

1) Understand the Readers/Writers problem and the solution provided:
   a. Follow slides and video lecture on the topic (scheduled for Tuesday, March 16 2021 lecture)
   b. Check textbook (Chapter 31.5)
   c. Look at the code (provided in the text but also on GitHub, linked from the web version of the textbook).
   d. (optional) Experiment with the code. If you use the GitHub version, you will realize that it is more complicated than it needs to be because it is written to work on both Mac and Linux systems. It thus obscures the API you will be using on Linux machines. It is, however, a very good

2 opportunity to learn a bit more about programming. Alternatively, you can use the code from the textbook and add what is needed (e.g., a main function to call the functions presented in Figure 31.13) to test it and become comfortable with it.

2) Understand what the problem is with starving writers.
3) Design a solution to this problem (on "paper" first. Don't code and hope you'll fix your understanding by debugging concurrency issues. Think first, then implement).
4) Ask questions:
   a. During office hours (instructor's or/and TA's)
   b. By email directly to us (cc instructor and both TAs).

5) Implement, debug, test on C4 lab machines, etc.
6) Make copies of your work. It is a stressful time, save your work with meaningful names such that you reduce risks of overwriting your code, losing work due to disk crashes, etc.
7) Write report, edit, edit, edit, spell check, ….
8) Submit, of course.