CS311/CS350 –Producer/Consumer with Threads

You have decided to begin a new telephone app: tweet-extractor. The problem is this: there are millions of tweets using thousands of hashtags. You want to create an app that will find all the tweets for a specific hash-tag and group them into a cohesive output file. The tweets come at irregular intervals. You need to be able to grab them as they appear and save them at whatever speed your device permits.

Part I. Description

As discussed in class, the objective of this part of the course is to learn the use of several concepts: mutual exclusion, multi-threaded programming, semaphores, mutexes and their effect on program throughput by allowing overlapping of I/O with CPU processing.

As part of this assignment, you will create and use a circular (bounded) buffer with semaphores. (Why can't you use mutexes?) See page 228 in the book for the general solution.

For file input and output specs, get the 2 files "setpath_defs.h" and"setpath_fn.h" on my server: insert #include <setpath.defs> immediately after your usual #includes (stdio, etc.), insert #include <setpath_fn.h> at the very end of your program file and change the file names inside setpath_defs.h to *p2-in.txt* and *p2-out.txt*, but keep the paths (as in project 1). Add this line at the beginning of *main*: setpath();

The Assignment

A. Create and run (on a Windows system), a single program in C or C++, with two threads that does the following:
   1 Opens a simple text file containing AT LEAST 500 KB of data (500 "lines" of 1KB each.). File naming rules are the same as project 1.
   2 Starts all timers with "start_timing()" function.
   3 Has two threads:
      a Thread 1 (the producer):
         (1) reads a line of the file into a "buffer-slot" (the "buffer" is a "circular" array with fixed size = 10 slots). Each subscript element of the buffer will hold one "line" of the file.
         (2) when "end of file" is detected, puts a "special code" in the buffer (you can define that code as a single hexadecimal byte that is NOT a keyboard character). The special code is the final entry in the buffer (i.e.; it is not added to an existing line of the file.)
         (3) exits after inserting the special code in the buffer.
         (4) continues reading until the buffer is full or waits for an indication that there is space in the buffer, then continues reading
      b Thread 2 (the consumer):
         (1) gets a line from a buffer slot and writes the line to a 2nd file
         (2) exits when the special code is in the buffer.
         (3) repeats the previous steps (a, b) as long as there is data in the buffer, waits if the buffer is empty

         Design: notice that this is different from Project 1. In Project 1 you had 1 loop that read 1 item, then wrote it, before looping back to get another item. In this program there are 2 loops (each is a separate thread), 1 reads until nothing is left, the other writes until there is nothing left.
   4 Stops timing using the "stop_timing() " function.
   5 Closes the files
   6 Computes & displays the time it took to read and write the total file.
      a wall clock difference (e.g.; double x=get_wall_clock_diff();)
      b CPU clock difference (e.g.; int y= get_nanodiff();)
   7 Compare the input and output files to ensure they are the same content and of the same size.

B. Compare the results of this lab with the results of Lab 1 and explain the differences, if any. If none, explain why. Hand in your answers. Send your source code & Design Document via FTP.

Part II. Run this program and the program from Project1 (UNCHANGED including handling line-end characters, if required) on the Linux system[1]. Be sure that the output file, for Linux is in the ~/fileio/ folder (which is a separate hard drive). Again, compare the times and explain the differences if any. If none, explain why. Use a word processing program to enter your answers and print them.

The rules:

A. You may use the standard template library and the pthreads library. You may **not** use system-specific or vendor-specific interfaces[2] .
B. The files will be in the same directories as in Project1. The input file contains any of the printable characters (A-Z, a-z, 0-9, blank, # and any of the other punctuation symbols). It is named p2-in.txt. The output file will be named p2-out.txt. Each "line" of the file ends with a "system dependent" end-of-line signal. Some systems (Windows vs. Linux) use LF+CR, others use only LF, some use only CR, so you need to use a system-independent line-reading function.
C. You cannot use >> for the I/O[3] because >> will read words and numbers into variables, but cannot read a "line". It also ignores blanks (spaces) in the input. There are several functions that will do this in both of the acceptable languages.
D. Don't start your timings until you are ready to start reading the file. Your file must include spaces as part of the text, so it "looks like" words. You may use any tools available to you to create the input file. You SHOULD NOT COPY the file from one system to another unless your code knows how to handle line ends (CR & LF) which may not act the same way. You should use another tool to verify that the output file is identical to the input. On *X, you can use "cmp", "diff" or any other tool. On Windows, you can use "comp" in a command prompt window.

Design info: You MUST use semaphores (3 of them), not mutexes. Your "reading" thread is supposed to use the semaphores to determine when it is safe to read for EACH buffer slot it needs to fill, BEFORE trying to read anything. Your "writing" thread uses a different semaphore to determine when there is a buffer slot to read (for EACH slot it reads). The 3rd semaphore is used to manage the number of open buffer slots.

A buffer **slot** is either a single character (for character-by-character operation, if required for the assignment) or a whole line (for line by line operation).

DO NOT use a character input loop to read the characters in whole-line mode. Use a function such as *getline* (there are others), which avoids the looping for I/O on each char. Note that *getline* deletes the end-of-line character, so you have to replace it in your buffer.

See the sample solution in the book.

[1] The VS compiler will use #define to set the existence of WIN32, or _WIN32 so my time_functions can generate Windows code.
[2] Vendor-specific and system-specific interfaces are those that only work on the specified system. Remember that you must run the program UNCHANGED across platforms.
[3] >> ignores whitespace (which includes all kinds of "end-of-line" characters).