CS 374 Lab 7: Shared Memory Programming

Student Names: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Today you will examine a program from John Gray which shares memory among its processes and implements a producer-consumer scenario using a 6-slot shared memory buffer. Instead of creating multiple producers, the relative speeds of producer and consumer are variable and controlled by the user. The code for this can be found in four files on Canvas:

* producer.cxx
* consumer.cxx
* parent.cxx
* local.h

The files worked fine for me, but there may be some startup time required to get them going. The header file references tools which all the programs need. Everything should work under g++ on either the lab Linux machines or nrs-labs machine with minimal fuss but the code is over ten years old and uses some deprecated features or syntax, so one of these days an update will cause a minor break. Note that “.cxx” is an older, less common extension for C++ files (worked fine for me with g++), so change to .cpp if the local compiler expresses disdain. The IO is done using the iostream class, hence the need for the C++ compiler. You can usually safely ignore compilation warnings, but not compilation errors; this code does generate a bunch of warnings.

Note that when running someone else’s program, you need to take time to learn how it works. *READ THE CODE*, it has the answers. A few key points here:

* In what order are the files compiled? (Note .h file)
* How do you execute the code? (Look at the source code, and think logically.)
* Some well-written programs will give you directions on how to run them when invoked incorrectly: “*parent producer\_time consumer\_time*.”

This code is elegantly written, and this lab is in essence an examination of the brevity and beauty of John’s code. In small groups of 2-3 students, compile and run this code and then answer the following questions, handing in a copy for your group (*with your names on it!*):

1. What does the producer actually produce? (Run it to be sure you understand what is happening. Try slightly different speeds. Think about what happens to the buffer when either the producer or consumer is faster than the other. Think what the number printed out on each line is referring to.)
2. How is data being passed between the producer and consumer? Which process owns the shared memory segment?
3. How does the producer find out how to access the memory of its parent? Reference a line or two of code to back up your claim!
4. Find the code pieces which display the messages to the screen. Are these pieces done inside of critical regions? Think about why this might be necessary, and *give a clear example* of how the code could go wrong without semaphore protections.
5. If both the producer and consumer know there are six buffer slots, are two semaphores actually needed? *Explain your answer clearly*. Note: an enumerated type (enum) is just a mapping between ints and strings, so AVAIL\_SLOTS and TO\_CONSUME are effectively the semaphore names (instead of 0 and 1, which aren’t good names).
6. Find the code which adjusts the wait times in the producer and consumer, and make adjustments to force the consumer to run slower than the producer on one hand, and at the same speed on the other. What happens to observed output and how does this connect to the message queue? Again, *explain clearly*!
7. How does John’s code attempt to ensure that all processes die if one of them exits? *Does it work?* Note: the easiest way to test this is to use **ps** and **kill** commands; try starting a long run, killing producer or consumer, then see if the others were killed.