Lab Document
Semaphores and Shared Memory

For this lab, you will be creating two C/C++ programs (producer and consumer) that uses semaphores and shared memory in order to communicate with each other. When combined, these two programs will simply be an inter-process version of the cp (copy) command.

producer.[c | cpp]
Usage: ./producer <textfile> <shared memory size in bytes>

The producer program will read the contents of the given text file, and write it to the shared memory segment in chunks (based on the given size of the shared memory segment) that the consumer will be able to read from. The program will keep running until all the contents of the text file have been written to the shared memory segment.

Depending on your solution, you may have to run this first before running the consumer.

consumer.[c | cpp]
Usage: ./consumer <textfile> <shared memory size in bytes>

The consumer will read from the shared memory segment (the same one that the producer uses) in chunks (based on the given size of the shared memory segment), and write all the chunks to the given text file. In case the file already exists, the contents of that file will be overwritten. When the program ends, the given text file must contain the exact same copy of the text file that is given to the producer.

Notes

1. Have at least one semaphore and at least one shared memory segment (both programs should use the same semaphore/shared memory keys, obviously).

2. There are many solutions. Be creative. Mine used 1 semaphore and 2 shared memory segments.
    a. The semaphore handled access to the shared memory segment. That way, both programs wouldn't access the shared memory segment at the same time.
    b. I had to use sleep(1) in both program loops to give the other program a chance to gain control of the semaphore. I suggest you do the same.
    c. Shared memory segment A handled the actual data transfer. When testing your program, you may provide a small size for your shared memory segment to verify that the data is being written in chunks and not in one go. You can also use a very large text file.
    d. Shared memory segment B handled status updates between the 2 programs. Suggested size: 2B. Here, I passed 0-to-1 character length strings to indicate 1 of 3 following situations:

i. Shared memory segment A not touched yet OR producer is done and has exited.
    1. Producer behavior: Read from the text file and write to shared memory. If there is nothing more to write, indicate ‘done’ status and exit. Otherwise, indicate ‘written’ status.
    2. Consumer behavior: If the producer is already done, exit. Its text file must now be an exact copy of the text file supplied by the producer (assuming the producer program was run first). Otherwise, wait for the producer to write data to the shared memory segment.

ii. Shared memory segment A was last modified by the producer (which is still running).
    1. Producer behavior: Do nothing.
    2. Consumer behavior: Read from shared memory and write to text file. Indicate ‘read’ status.

iii. Shared memory segment A was last read by the consumer (which is still running).
    1. Producer behavior: Same as in (i).
    2. Consumer behavior: Do nothing.