Assignment Four: Simple IPC - Pipes

EECE 4029: Operating Systems and Systems Programming
ELTN 4022: Operating Systems

Department of Electrical Engineering and Computer Science
University of Cincinnati, Cincinnati OH
Revision Date: Aug 12, 2020

Student Name: **Brian Culberson**

1.1: In the above sample code, you'll see a for loop of the form "`for (i=0; i < 2; i++)`". Change that "2" to a "3", recompile, and run the program. What do you see? Is this the expected behavior? Why or why not? (10 points)

If you change the 2 to a 3 it kind of gets stuck and does not print anything. This is because since it is read more than twice but there are only two messages with nothing else in the pipe. If you added a third message and added it to the pipe it would execute normally with that added part into the message.

1.2: In the code I gave above, the parent is the producer (it writes things into the pipe) and the child is the consumer (it pulls things out of the pipe and prints them). What would happen if, through some mischance, the kernel process scheduler sent the child process to a CPU before it the parent process had any chance to write to the pipe? Be specific. (10 points)

The child would check to see if anything is in the pipe. After seeing nothing is in the pipe it would wait until something is in the pipe for it to get. This would result in no change in the program operation because of the wait in the child.

1.3: In the code I gave above, what happens if the parent tries to write to the pipe when it is already full (on Linux, it has 64K of data in it already). Be specific. Is this a good thing? (10 points)

After some quick research, writing something to a full pipe will make the write wait until there is space to write it to the pipe. This is a good thing because there is no data loss to the pipe

1.4: `read()` and `write()` are guaranteed by the kernel to be "atomic". This means that while a read() or write() is happening, NOTHING will interrupt it. Is this a good thing? Is this atomic condition necessary? Explain your answer. (10 points)

This is a good thing because pipes are important to send info to other processes. If this does not happen the application will not work properly or another error will happen.

1.5: In example 2 code above, the first line of the child code closes `p[1]`. Comment out or remove that call to `close()`, recompile, and run the code. What behavior do you see. Is this

expected? Why or why not? If I restored that the initial close() in the child code, but removed or commented out the initial close() in the parent code, what behavior would I see? Is this consistent with *how blocking works with a pipe*? Why or why not? (10 points)

For the first comment out in the child there is no output, but it does not exit. For the second comment out in the parent, it worked as expected. When there is a connection to the pipe from both sides, the child and both do input and output there is a problem on what can be sent through the pipe, or it is blocking the pipe. I don't think it should work this way and a pipe should be able to send or receive data whenever it can. This would leave out unnecessary waiting and blocking.

IN ADDITION TO YOUR ANSWERS TO THE ABOVE QUESTIONS TURNED IN AS A SINGLE PDF. YOU SHOULD ALSO TURN IN A C SOURCE CODE FILE AS A TEXT FILE THAT CONTAINS YOUR IMPLEMENTATION OF THE TEE COMMAND. BE SURE TO COMMENT THIS FILE EXTENSIVELY. (50 POINTS + POTENTIAL EXTRA CREDIT IF BASIC FUNCTIONS WORK + EXTRA FUNCTIONS AS DESCRIBED IN ON THE ASSIGNMENT CANVAS PAGE)