Trent University
COIS3380H
Winter 2017
Lab 6
Due: Friday, April 7th, 2017 at NOON

Lab #6: Client Server Processes

Choose ***one*** of these two options:

A. IPC using FIFO
B. Sockets and Client-Server

In this lab, you will be required to write **two** C routines which talk to each other through a communications channel. You can Choose to either implement the code using FIFOs or through the use of sockets. In the description below these will be referred to generically as "the communications channel" or just "the channel".

The objective is to create a client-server environment to copy the contents of a file from the server to the client. Your client code will run as a single process on the computer. Your server code must also be run as a separate process on the same computer. No *fork()* or *exec()* calls are to be used.

Your client code, once connected, should send a message to the server. The message should contain a filename. The filename should be supplied to the client as a command line parameter. The server should then open the file and copy it through the channel to the client. Each "packet" of information sent should use a structure that contains both the number of bytes being transferred as well as the data stream.

The client code should then accept the data structure through the channel and write the identified number of bytes to a local filename. The data exchange should happen in fixed block sizes, say 1024 bytes. When the server sends a packet to the client containing fewer than the expected packet size, your code can assume the server has hit the end of file. Remember how you did something similar to this in Lab 4.

Both the server and client should close their copy of the file. The client should then send a disconnect message to the server and exit.

For testing and your assignment submission, use the same file as lab4 (/home/common/lab_sourcefile). Your destination filename should be something like: lab_sourcefile_local_clone.

A) IPC using FIFO

Your client code should use two FIFOs. Once for client-to-server transfers and one for server-to-client messages. The client must check for the presence of both FIFOs before it accepts user input. You can use the access() function from unistd.h to check for the presence of the FIFOs. If either of the FIFOs is missing, a message should be displayed as to which is missing, then the code should exit. You can't run a client if the FIFOs don't exist to act as communication channels to the server.

Your server code, which you should run first, should check for the existence of the two FIFOs. If any of them is missing, the server code should create them. The server code should then wait for a message to arrive from the client, process it and return content of the requested file back to the client. The server should remain running waiting for a different client to connect.

Ensure that once you are done running your code that you kill the server process. Either stop it using the kill command line verb (if you created a background process using &) or by pressing CTRL-C in the terminal window it is running under. The server should remain running just in case a different client connects at a later time (in a real world scenario).

B) Internet Sockets

This lab is essentially a socket implementation of the FIFO process described above. In this lab, you will be required to write **two** C routines which talk to each other through a proper TCP/IP sockets. The objective is to create a real client-server environment to copy the contents of a file from the server to the client. Your client code will run as a single process on the computer. You server code must be run as a separate process on the same computer.

> **Note:** The fact that the client and server processes are running on the same computer is a limitation of our lab environment. Your code should be able to run on two separate systems. In the test phase, I will attempt to provide for you a system where you can test your code across devices.

Your client code should check for the presence of the server and your code's ability to connect. If the client cannot connect, it should exit gracefully.

Your server code should create and listen on a proper TCP/IP socket. The server code should then wait for a message to arrive from the client, process it and return content of the requested file back to the client. Once the file is transferred, your server code should gracefully hang up the connection. The server should remain running waiting for a different client to connect. **You need not code your server to allow for multiple simultaneous connections.**

Ensure that one you are done running your code that you kill the server process. Either stop it using the kill command line verb (if you created a background process using &) or by pressing CTRL-C in the terminal window it is running under.

NOTE: On the server side, you will need to supply a PORT number for your server to listen on. Please use the last 4 digits of your student number and ass 60000 to it. So if your student number ends in 1701, code your server (and hence the client) to use port 61701. That way we don't interfere with each other and we don't use a port already allocated on the system (your server won't start as the O/S won't let it).

Submissions:

To confirm your file transfer has performed correctly, you MUST submit a proof with your ZIP file. The easiest verification is to run the md5sum program against both the source and target file to show that their md5 hashes are identical.

You should test your code with more than just one file. It is not expected that your code transfer binary files but you should try those as well as they "should" transfer equally well.

You are required to hand in **well document** and modular programs plus sample output showing the functionality of your. Zip all of your code, using the proper naming convention and directory paths, testing and sample results into a single submission for Blackboard.