Trent University
COIS3380H
Lab 6
Due: April 6th, 2023
(I cannot accept any submissions after 11:59pm. No extensions)

Lab #6: Client Server Processes

Choose *one* of these three options:

A. IPC using FIFO
B. Shared memory
C. 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, a shared memory segment 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 client to the server. This will mimic a file UPLOAD process. 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. This means having two terminal windows running. One for the server, one for the client.

No *fork()* or *exec()* calls are to be used for this lab.

The filename should be supplied to the client as a command line parameter.

Your client code, once connected, should send a message to the server. The message should contain a filename.

The server should then receive the filename and prepare to receive data. The client should open the file and copy it through *the channel* to the server. You must only use read() and write() to transfer data through the socket.

Each "packet" of information sent must use a structure that contains both the number of bytes being transferred as well as the data stream. Something akin to:

```
struct student_record
{
  int    nbytes;
  char   payload[1024];
};
```

The server code should then accept the data buffer through the channel and writes the identified number of bytes to the local filename. The data exchange should happen in fixed block sizes, say 1024 bytes. When the client sends a packet to the server containing fewer than the expected packet size, your code can assume the client has hit the end of file. You should expect the test files I will use will be of arbitrary length. Some will be smaller, some much larger than 1024 bytes.

Restrictions:

- When you are retrieving data from the source file, you must use open() and read(). You are not allowed to use fopen() as it will not read binary files properly.
- **You are <u>not</u> allowed to use the sendfile() system call or any other bulk transfer utilities.**
- Depending on the approach you have chosen, you are **<u>required</u>** to read()/write() or send()/recv() the data and then push it through the channel you created.

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/COIS/3380/lab4_sourcefile). Your destination filename should be something like: lab6_sourcefile_local_clone. Don't forget to test your code with non-text files such as those found in /home/COIS/3380/lab3

A) IPC using FIFO

Your client code should use two FIFOs. One for client-to-server transfers and one for server-to-client messages. You are allowed to assume that these FIFOs have static well known names and that only one client will ever be talking to the server at a time.

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 client should exit. You can't run a client if the FIFOs don't exist to act as communication channels to the server. Your testing logs should show both scenarios.

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. It should contain the filename being uploaded. The server must rename the file if it is going to place it in the same directory as the source. The server should then read "structure" size chunks from the client->server FIFO and write the appropriate data to the output file. When the number of bytes in the payload portion of the structure is less than the buffer size, you can assume that's the last block of the file. The server should send an "All DONE" message to the client. The client should wait to receive this message, print out an OK message then exit. 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). You could, should you want to be fancy, create a signal handler to help terminate the server process.

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 proper TCP/IP sockets. The objective is to create a real client-server environment to upload the contents of a file from the client to the server. 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. Because of the firewall protecting Loki, you cannot run the server on Loki and your client on your laptop. The socket port will be blocked.

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. The first message will be the name of the file being uploaded. Next, the server should expect "structure" size chunks of data through the socket and write the payloads to a local copy of the file. Once the file is transferred, your server code should send an OK message to the client, then, 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 (that would/should require a fork() or threads).

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 (or a signal).

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 add 50000 to it. So if your student number ends in 1701, code your server (and hence the client) to use port 51701. Or if it is 17, your port number would be 50017. 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).

C) Shared memory segment.

Since a shared memory segment is...shared, there is only a requirement for one such channel for your code. This is a slightly more complex piece of code since it will require synchronization between the two parties sharing the memory region.

The server process should start up and create the shared memory region. The client should then connect to the shared memory region and pass the filename through it. The server should read the filename and respond telling the client it can start transferring data. This would probably involve some chunk of memory used to exchange "status".

Since access to the shared area is uncontrolled, you'll need to create your own handshaking routine to manage the flow control. Maybe reserve the first few bytes of the shared region as a set of "traffic lights". As an example, your structure might contain an extra flag field called something like: "flow". When that field is a ZERO, the client has permission to update the data block in the shared area. Once the file chunk has been written memory, the client changes the flag value in the "flow" (Say to a 1). The server polls the "flow" variable (memory area) waiting for it to change to a 1. The "1" would indicate that it is the server's turn to write the data block to disk, then set the flag to a zero. And so on until all of the file has been transferred. You could also use a simple counter for the number of payloads transferred. When the number changes, it means the server has written the payload to disk. Again, not a trivial exercise but a rewarding one.

Again, the shared memory region should reserve room for both the number of bytes transferred and the data buffer. Remember that if you don't keep track of how much data is in the region, and you don't zero it out, then you may wind up with more data in the uploaded file than you would want. Hence, the number of bytes field in the structure to limit what you read and write.

Of course, this implementation would cause a CPU SPIN situation waiting for the flag to change. This could be alleviated with a sleep(1) or a pause() and a alarm(1) type signal inside each routine. I'm OK if it is a bit slow. It will help you "watch" the program work and see any debug statements you might have in your code as you test 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 and more than one file type. Test your program with flat ASCII/text files, an image or a PDF.

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