B4 - Network Programming

B-NWP-400

Bootstrap

My FTP

Bootstrap

language: C

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

This bootstrap aims to introduce you to the client-server model.
It will hinge on 3 main steps:

- creating a server
- creating a client
- Handling multiple clients
- First step to File Transfer Protocol

Step 1 - Server

We will, in several steps, initialize a "server" *socket* whose main feature will be to accept new connections.

Creating the socket

socket, as the name suggests, allows to create a communication channel.
Start by creating a socket with the following properties:

- IPv4 addressing
- TCP communication (connected mode)

Read the *socket(2)* man page.

Configuring the socket

bind allows to “name” the socket, that is assigning a network address (and possibliy a port) to the socket, on which it will “listen”.
Initialize a *sockaddr_in* structure with the following properties:

- address family: Internet
- communication port: any
- network interface: all local interfaces

And configure the socket created in the previous step with the provided informations.

Read the *bind(2), htons(3) , ip(7)* man pages.

Initializing the queue

In order to communicate with your server, you must initialize a queue for clients to wait until their connection request is taken into account.
Initialize a queue with a sufficient length.

This highly evolved process was shamefully inspired from post offices and child benefit offices treatment processes...
However, we will ask you to be more effective than them in dealing with your clients' requests...

Read the *listen(2)* man page

Accepting incoming connections

When a client attempts to communicate with a server, it first sends a connection request that the server must “accept”. For that purpose, the *accept* system call will come in handy.
Accept the first incoming connection and display the client's connection information (IP address, port).
Then, write “Hello World!!!” to the socket and close it.

Read the *accept(2), inet_ntoa(3) and ntohs(3)* man pages.

Testing

Start your server and connect with a *telnet* client.
You should go with something like this (in the example, the argument is the port on which we listen).
Both terminals are really separate terminals, and thus the message displayed by the server appears only after the telnet client was connected.

Terminal

```
∼/B-NWP-400> ./server 4242
Connection from 127.0.0.1:1024
```

Terminal

```
∼/B-NWP-400> telnet 127.0.0.1 4242
Trying 127.0.0.1...
Connected to localhost
Escape character is '^]'.
Hello World!!!
Connection closed.
```

If you do not get a similar behavior, you must have done something wrong.
Start over and take the time to read the man pages, they are full of useful informations.

Step 2 - Client

Now that we have made our first server, let us see how to program a "simple" client.

Creating a socket

Create socket the same way and with the same properties as the one in the server.

Connecting to the server

Unlike a server, a network client does not need to "name" the socket, nor to create a queue. Thus, *bind* and *listen* system calls will not be of any use.
In the same way, a client will not accept incoming connections (it is specific to servers), the *accept* system call will not be of any use either.

Instead, a client shall request a new connection from the server. For that purpose, you will have to use the *connect* system call.
Fill a *sockaddr_in* struct with the following information:

- the server's connection port
- the server's IP address
- address family: Internet

And connect to the previously created server.

> Read the *connect(2), htons(3) and inet_addr(3)* man pages.

Reading the socket

It is now time to read the information sent by the server on the socket using a simple call to *read* and then display it.
You should get the following behavior:

Terminal

```
~/B-NWP-400> ./server 4242
Connection from 127.0.0.1:1024
```

Terminal

```
~/B-NWP-400> ./client 127.0.0.1 4242
Server said: Hello World!!!
```

> Simple *read/write* calls on the client-side and server-side socket will allow you to exchange messages, commands, results...

Step 3 - Multiple clients

So far, your server is able to accept one incoming connection before quitting. This is somewhat useless... We will now ensure that the server is capable of handling all incoming connections.

Loop it!

In the code of your server, add an infinite loop that will:

- Call *accept*
- Display the connection information
- Write to the socket
- Close the socket

Start your server and try successive client connections... It should work.

Fork it!

Our server is starting to look like something. However, it is only able to process incoming connections one after the other.
If a treatment happens to last for a while, the other clients would have to wait until it is finished.
So we need to find a solution to deal with our clients in the fastest possible way (so it does not look like the line in a post office on Saturday morning).
For that purpose, we will handle each request in a new process.
Call *fork* after accepting the incoming connection.
In the child process, do the same as in the previous exercise.
In the parent process, close the in-bound socket (Indeed, it is useless in the parent process) and return to the call to *accept*.

> To easily test it, you can add a call to *sleep* in the son process to simulate a long operation.
> Then connect multiple clients to your server. All of them should be processed in parallel.

Step 4 - Using select

Handling only 1 client per process is easy to implement, but extremely resource-consuming !
If the client management is not time-consuming, a better way is often to use *select(2)*.

> Let's have a look at the man page to understand its usage.

As you have read in the man page, *select(2)* indicates that a data is available for reading on the client socket, *read(2)* is then nonblocking.
But what happens if we use *select(2)* on the server's socket ?
As you might guess, we never need to read any data on a server socket, we just use *accept(2)*. That's precisely to avoid a blocking call to this function that *select(2)* will be useful.
Indeed, when a new connection comes in, *select(2)* flags the server's filedescriptor as having data to read, and then gives the control back to the program.
From then, we can call *accept(2)* because we know that it won't be blocking.
Look at the example on the man page to understand how to use *select(2)*.

> In this example, *writefds* is not used but is however very useful for not being CPU time-consuming. Try to understand how to use it judiciously.

> Is a timeout always needed ? Try to write a code as much optimised as possible !!!

You can now patch your previous multiclient server to avoid the use of *fork(2)* and implement a *select(2)* version.
You should probably keep a list of used filedescriptors (socket's one) and identify for each the use you will have (server, client, free).

Step 5 - File Transfer Protocol

Well, your server is now able to handle all incoming connections. We will now implement the "passive" mode for data transfer of the FTP.
These are the steps to do that:

- The client sends the **PASV** command to the server using the connected socket (called *CONTROL socket*)
- When receiving the **PASV** command your server :
  - Creates a new socket (called *DATA socket*)
  - Opens it using the same network interface as the *CONTROL socket*
  - Binds a free port
- The server sends to the client; using the *CONTROL socket* a **227** reply code, followed by an informative message, ended by the *(IP1, IP2, IP3, IP4, PORT1, PORT2)* pattern.

> *IP1, IP2, IP3, IP4* are each byte of the server's IP address, in the right order.
> *PORT1* and *PORT2* are used to calculate the *DATA socket* port to bind, using the following formula: *PORT1 * 256 + PORT2*

- The client decodes the server's answer and opens a new connection to the given IP binding the given port.
- The client is now able to send a **RETR /pathto/file** comand to the server using the *CONTROL socket*.
- When receiving the **RETR** command, your server:
  - Opens and reads the file
  - Sends the tight reply code on the *CONTROL socket*
  - Sends data over the *DATA socket*
  - Closes the *DATA socket* when finished transfering data
- The client reads data over the *DATA socket* and writes it to the destination file.
- The server sends the right reply code over the *CONTROL socket*.
- That's all!

> Please refer to the subject's RFC extract to discover some of the reply codes.

It works? Great! Now that you know how to transfer a file over the network using FTP, try to implement the "active" mode of it, which is quite of the same thing.