CSCI 4210 — Operating Systems
Homework 4 (document version 1.2)
Network Programming using TCP/IP

- This homework is due by 11:59PM ET on Monday, May 3, 2021
- This homework is to be done individually, so **do not share your code with anyone else**
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v18.04.5 LTS and `gcc` version 7.5.0 (`Ubuntu 7.5.0-3ubuntu1~18.04`)

Homework specifications

In this final homework assignment, you will use C to implement a TCP client similar to the given `tcp-client.c` example. The specifications that follow focus on the application protocol to implement in your client process such that it successfully communicates with the server process that is running on the Submitty machines.

Application protocol

For the application protocol, your client process first sends the four-byte `int` value `n` to the server; here, `n` specifies the number of additional `int` values to follow. So, your server also sends these subsequent four-byte `int` values, i.e., `int-value-1`, `int-value-2`, etc. All values are sent as one TCP packet to the server.

In response, your client process receives two separate TCP packets. The first packet contains a single four-byte `int` value; call this value `result`. The second packet contains a character string; call this `output-char-string`. Both values are to be displayed as output in your client code.

Note that all integer values are to be sent as four-byte integers in network byte order. And remember that character strings do not have a terminating `'\0'` character when sent across the network.

Below is a summary of the required exchange:

```
          SERVER PROCESS                    CLIENT PROCESS
        ----------------------------     ----------------------------
                <<<<<<<< TCP-connection-established <<<<<<<<

                        <<<<< REQUEST: n int-value-1 ... int-value-n <<

        >> RESPONSE: result >>>>>

        >> RESPONSE: output-char-string >>>>>

(v1.2)  >> close( newsd ) >>>> TCP-connection-closed >>>>>>>>>>
  ^     ----------------------------     ----------------------------
  |
 see 04/29 lecture for details
```

Command-line arguments

There are at minimum four required command-line arguments. The first two command-line arguments are the server hostname and port number to connect to. The third command-line argument specifies `n`, while the remaining command-line arguments are the actual `int` values.

If invalid arguments are given, display the following to `stderr` and exit with `EXIT_FAILURE`:

```
ERROR: Invalid argument(s)
USAGE: a.out <server-hostname> <server-port> <n> <int-value-1> ... <int-value-n>
```

Program execution and required output

To illustrate via an example, you could execute your program as follows:

```
bash$ ./a.out linux02.cs.rpi.edu 8192 3 895 110 942
```

This will attempt to connect to the server process running on port `8192` of the `linux02.cs.rpi.edu` machine using TCP. If successful, the client process sends a packet containing the `int` values using network byte order. In response, the client process receives two packets. Sample client process output is shown below (**(v1.1)** each line of output starts with `CLIENT` and other minor changes):

```
bash$ ./a.out linux02.cs.rpi.edu 8192 3 895 110 942
CLIENT: Connected to server
CLIENT: Sending 3 integer values
CLIENT: Rcvd response 649
CLIENT: Rcvd response "ABCEGHIJLMNOPQRSTUVWXYZ"
CLIENT: Disconnected from server
```

Match the above output format **exactly as shown above**.

Error handling

In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`. Remember to only use `perror()` if the given library function or system call sets the global `errno` variable. Otherwise, use `fprintf()`.

In general, error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

If an error does occur, have your client process close the TCP connection and exit with `EXIT_FAILURE`. Otherwise, exit with `EXIT_SUCCESS`.

Submission instructions

To submit your assignment (and also perform final testing of your code), please use Submitty.

Note that this assignment will be available on Submitty a minimum of three days before the due date. Please do not ask when Submitty will be available, as you should first perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, make use of the `DEBUG_MODE` technique to make sure that Submitty does not execute any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
  printf( "the value of q is %d\n", q );
  printf( "here12\n" );
  printf( "why is my program crashing here?!\n" );
  printf( "aaaaaaaaaaaaaggggggggghhhh!\n" );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw4.c
```

Second, output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure output is not buffered, this is a good technique to use.