CSCI 4210 — Operating Systems
Homework 4 (document version 1.0)
Network Programming and TCP/IP

- This homework is due in Submitty by 11:59PM EST on Thursday, April 21, 2022
- You can use at most three late days on this assignment
- This homework is to be done individually, so **do not share your code with anyone else**
- You **must** use C for this assignment, and all submitted 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
- All submitted code **must** successfully compile and run on Submitty, which uses Ubuntu v20.04.3 LTS and `gcc` version 9.3.0 (`Ubuntu 9.3.0-17ubuntu1~20.04`)

Hints and reminders

To succeed in this course, do **not** rely on program output to show whether your code is correct. Consistently allocate exactly the number of bytes you need regardless of whether you use static or dynamic memory allocation. Further, deallocate dynamically allocated memory via `free()` at the earliest possible point in your code. And as covered in initial class videos, make use of `valgrind` to check for errors with dynamic memory allocation and use. Also close any open file descriptors or `FILE` pointers as soon as you are done using them.

Another key to success in this course is to always read (and re-read!) the corresponding `man` pages for library functions, system calls, etc.

Homework specifications

In this final homework assignment, you will use C to implement a TCP client based on the `tcp-client.c` example that is given. 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.

Note that the TCP server code will not be provided to you. Use the `tcp-server-iterative.c` and `tcp-server-fork.c` examples to test your TCP client code.

Application protocol

For the application protocol, your client process sends a packet containing `n+1` four-byte `int` values. The first `int` value specifies `n`, with `n` specifying the number of additional `int` values to follow. Note that all `n+1` values are sent as one TCP packet to the server.

In response, your client process will receive at least two separate TCP packets. The first packet contains a single two-byte `short` value; call this value `short-result`. The second and subsequent packets contain character strings; call these `secret-message` strings and assume there is an arbitrary number (`x`) of such messages. Display all received data as output in your client.

Note that all integer values must be transmitted in network byte order. Further, 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 with server <<

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

>> RESPONSE: short-result >>>>>>>>>>>>>>>>>

>> RESPONSE: secret-message #1 >>>>>>>>>>>>
>> RESPONSE: secret-message #2 >>>>>>>>>>>>
     .         .         .
     .         .         .
     .         .         .
>> RESPONSE: secret-message #x >>>>>>>>>>>>

>> close( newsd ) >>>>> TCP-connection-closed >>>>>>>>>>>>>>>>>>>>>>>>>
-----------------------------          -----------------------------
```

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 8123 3 895 110 942
```

This will attempt to connect to the server process running on port `8123` 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:

```
bash$ ./a.out linux02.cs.rpi.edu 8123 3 895 110 942
CLIENT: Successfully connected to server
CLIENT: Sending 3 integer values
CLIENT: Rcvd result: 649
CLIENT: Rcvd secret message #1: "ABCEGHIJ"
CLIENT: Rcvd secret message #2: "LMNOPQRS"
CLIENT: Rcvd secret message #3: "TUVWXYZ"
CLIENT: Disconnected from server
```

As another example:

```
bash$ ./a.out linux02.cs.rpi.edu 8123 1 -334
CLIENT: Successfully connected to server
CLIENT: Sending 1 integer value
CLIENT: Rcvd result: 5876
CLIENT: Rcvd secret message #1: "GHIJ"
CLIENT: Rcvd secret message #2: "LMNS"
CLIENT: Rcvd secret message #3: "!"
CLIENT: Rcvd secret message #4: "TUVYZ"
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()`. Only use `perror()` if the given library function or system call sets the global `errno` variable.

Error messages must be one line only and use the following format:

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

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( "aaaaaaaaaaaaagggggggghhhh!\n" );
#endif
```

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

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

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 good results on Submitty, this is a good technique to use.