CIS*3210
Assignment 3
Deadline: Monday, November 18, 11:59pm
Weight: 12.5%

Description

This assignment can be done individually or **in teams of two**. You are welcome to use any of the code examples that I have provided for this course. You are also free to use Beej's Guide to Network Programming Using Internet Sockets: https://beej.us/guide/bgnet/html/split/.

Overview

In this assignment, you need to write a simple server/client system for transferring files over TCP:
- A client would send a text file to the server. For the sake of simplicity, we will only handle text files.
- The server would be capable of receiving text data (files) from clients over TCP sockets and creating new files on the disk.

The server should be storing the files in the same directory as the server executable. Each complete message (file) is large and will be broken by the client into multiple chunks for transfer, and each chunk might be further broken up by the TCP protocol implementation. The chunks will be assembled into a single file by the server before being written to the disk.

You will also do some tests on your code and report your results.

The client should read the file from the disk, transmit it, and exit. You should execute the client is executed as follows:

```
$./sendFile fileName IP-address:port-number bufSize
```

where
- `fileName` is the name of the file you are sending
- `IP-address` is the IP address of the server
- `port-number` is the TCP port the server listens on
- `bufSize` is the optional size of the buffer used by the client to handle large files/sends

The server should be listening for messages on a specific port using the TCP protocol. It should handle the client connections sequentially and accept connections from multiple clients. It would service clients one at a time - in this case, servicing a client means receiving the entire file from this client.

After receiving the entire file from one client, the server would save it to disk, display the necessary transfer summary, and move on to the next client. If multiple clients try to simultaneously send text messages to the server, the server should handle them one at a time (in any order).

The server is executed as

```
$./server port-number bufSize
```

When a server receives a file, it should display:
- The name and total size of the file it received
- The IP address of the host it received the file from
- The size of the read buffer on the server. Remember, the TCP stream will deliver data to the server in chunks of its own choosing, and you will want to avoid buffer overruns when reading data into a temporary buffer for the file.

The server should be storing the files on the disk, in the same directory as the server executable. Duplicate files should be resolved and how you do this up to you. The easiest option is to do what your main browser or operating system does - create a new file with a slightly different name, e.g. with "(2)" appended after the original file name and before the extension. Please clearly state how you resolve the duplicate files in the PDF report discussed below.

Assume that the file contents can be arbitrarily large, but the buffers you use to read the file / write to the socket must be small and of fixed size. There are some limitations, often very system-specific, on how much data can be written into a socket at once, and efficient handling of large files is outside the scope of this course.

To avoid worrying about these details, you will decide on the sizes of read/write buffers yourself Your client and the server should have an optional command-line argument that specifies the length of the buffer. If an optional buffer is not provided, set the default buffer size to 4096 bytes.

Also, please remember that even though you write N fixed-size chunks into a socket, TCP on the sender might further fragment these, and TCP on the receiver may merge these chunks into blocks larger than the original chucks sent from the client. As a result, be careful when receiving data to avoid buffer overruns.

The application-layer protocol for this file transfer application is up to you. For example, you can break up a file transfer into at least two messages - the file name and file contents, where the file contents may themselves be broken up. Alternatively, you could embed the file name into the message, with the first N bytes representing the file name, and the rest of the message - the file contents.

Also, remember the stream nature of the TCP protocol. The data will be fragmented by the TCP protocol, so the server will need to know how many bytes to expect for each file.

Error handling

Make sure you do basic error handling. This is harder to do with blocking calls, since we cannot do time-outs, but make sure you handle invalid files, invalid/missing command line arguments, failure to open a socket or bind to a port number, etc..

Make sure you handle the following correctly:
- **Local and remote operation**: Your program must be able to operate when connection over both `localhost` (`127.0.0.1`) or between separate machines on a network. For a small

(2%) bonus grade, you can let the user host name instead of the IP address, and then use `getaddrinfo()` to get the IP address of the target server.

- **Handling return values**: By default, sockets are blocking, and for this assignment we will use only blocking sockets. "Blocking" means that when we issue a socket call that cannot be done immediately (including not being able to read or write), our process is blocked - i.e. waits - until it can perform the action. This includes the case when
  - a socket's internal buffer is full and therefore, no data can be written, or
  - a socket's buffer is empty, and no data is available to be read.

However, if there is some data available to be read or some can be written, the call will return the number of bytes read or written respectively.

NOTE: This returned value (e.g. number of bytes read or written) can be less than the length specified in the function call. You need to handle this and determine whether this indicates an error.

Development and testing

Since we will end up with multiple servers running on the same SoCS hosts, each of you will need a unique port number. We will email you unique port numbers that you can use. Initially, you should develop and test everything on localhost, so you can use a port number of your choice. Please do not deploy your code on the school servers until you have received your unique port number.

As mentioned in class, debugging your code might be easier if you work on campus, because you will be able to examine your TCP traffic using Wireshark. Developing from home is definitely possible, but you will not be able to see the contents of the TCP traffic with Wireshark because of all the security/encryption measures.

Make sure that your server can handle more than one client. Create a script (e.g. in Python or bash) that spawns several clients that try to write to the server simultaneously.

You will need to test your client and server in two different environments:
- The client and server are running on the same machine. This can be done at home, or with the client and server running on the same SoCS Linux server.
- The client and the server are running on separate SoCS Linux servers.
  - Remember that `linux.socs.uoguelph.ca` is an alias for several different hosts: `linux-01.socs.uoguelph.ca` to `linux-08.socs.uoguelph.ca`.
  - Make sure you know which host you are connecting to. For this case, SSH into a specific server to run the server, then run the client on a different one. For example, you could run the server on `linux-01.socs.uoguelph.ca` and the client - on `linux-02.socs.uoguelph.ca`.
  - For two-person teams, you are welcome to run the server using one user account, and the client using the other.

For each environment, record the following:
- Ping time - let the ping utility do at least 20 pings, record the stats (round-trip min/avg/max/stddev, packet loss rate).
- The average, minimum, and maximum time that your code took to do the transfer. Do at least 20 file transfers, use the same file each time. A test file will be provided to you.
- The min/max/average transfer rates.

Required files

- All the `.c` and .h files for your client and server
- A `Makefile` that compiles the server and the client
  - `make all` creates executables server and client
  - `make clean` deletes all executables and intermediate files
- A script for spawning and running multiple clients simultaneously
- A `README` file that describes how to run the script(s) and provides all other instructions necessary to run your assignment
- A PDF report that shows the test results from your two test environments. If you work as a team, clearly state this in the PDF file. Discuss your approach for dealing with duplicate files sent to the server here.

Submission

Create a zip archive with all your deliverables and submit it on Moodle. The filename must be `FirstnameLastnameA3.zip`. Do not include any binary files in your submission. If you are completing A3 as a team, only one team member needs to submit the file. The file name should match the submitter.