CSS 432: HW1

1. Overall Requirement

The first assignment will require you to create a multi-threaded client/server and evaluate the reads and writes made by the system. The focus of this assignment is not on the threads, so we will not be covering them in detail like you will in an OS course.

Your program should consist of two parts. Please create two files accordingly, one is for the **client** (e.g., client.cpp) and one is for the **server** (e.g., server.cpp).

- **Server**. The server will create a TCP socket that listens on a port (the last 4 digits of your ID number unless it is < 1024, in which case, add 1024 to your ID number). The server will accept an incoming connection and then create a new thread (use the pthreads library) that will handle the connection. The new thread will read all the data from the client and respond back to it (acknowledgement). The response detail will be provided in Server.cpp.
- **Client**. The client will create a new socket, connect to the server and send data using 3 different ways of writing data (data transferring). It will then wait for a response and output the response.
- Perform this task between two computers in the remote Linux lab (Any two from **csslab1.uwb.edu** to **csslab12.uwb.edu**) or the in-person Linux lab (UW1-320). For more information about CSS Linux labs, please check http://depts.washington.edu/cssuwb/wiki/connecting_to_the_linux_lab.

See Beej's programming guide https://beej.us/guide/bgnet/html and the lecture slides for information on using sockets.

See http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html (Links to an external site.) or https://computing.llnl.gov/tutorials/pthreads/ (Links to an external site.) for information on using pthreads.

2. Detailed Requirement

Client.cpp

Your client program must receive the following six arguments:

1. **serverPort**: server's port number
2. **serverName**: server's IP address or host name
3. **repetition**: the number of iterations a client performs on data transmission using "single write", "writev" or "multiple writes".
4. **nbufs**: the number of data buffers
5. **bufsize**: the size of each data buffer (in bytes)
6. **type**: the type of transfer scenario: 1, 2, or 3 (see below)

From the above parameters, you need to allocate data buffers as below:

```
char databuf[nbufs][bufsize]; // where nbufs * bufsize = 1500
```

The three transfer scenarios are:

1. **multiple writes**: invokes the write( ) system call for each data buffer, thus resulting in calling as many write( )s as the number of data buffers, (i.e., **nbufs**).

   ```
   for ( int j = 0; j < nbufs; j++ )
      write( sd, databuf[j], bufsize );    // sd: socket descriptor
   ```

2. **writev**: allocates an array of **iovec** data structures, each having its ***iov_base** field point to a different data buffer as well as storing the buffer size in its **iov_len** field; and thereafter calls writev( ) to send all data buffers at once.

   ```
   struct iovec vector[nbufs];
   for ( int j = 0; j < nbufs; j++ ) {
     vector[j].iov_base = databuf[j];
     vector[j].iov_len = bufsize;
   }
   writev( sd, vector, nbufs );           // sd: socket descriptor
   ```

3. **single write**: allocates an **nbufs**-sized array of data buffers, and thereafter calls write( ) to send this array, (i.e., all data buffers) at once.

   ```
   write( sd, databuf, nbufs * bufsize ); // sd: socket descriptor
   ```

The client program should execute the following sequence of code:

1. Open a new socket and establish a connection to a server.
2. Allocate **databuf[nbufs][bufsize]**.
3. Start a timer by calling **gettimeofday**.
4. Repeat the **repetition** times of data transfers, each iteration is performed based on the specified **type** such as **1: multiple writes**, **2: writev**, or **3: single write**
5. Lap the timer by calling **gettimeofday**, where lap - start = data-transmission time.
6. Receive from the server ***an acknowledgement*** that shows how many times the server called read( ).
7. Stop the timer by calling **gettimeofday**, where stop - start = round-trip time.
8. Print out the statistics as shown below:

   ```
   Test 1: data-transmission time = xxx usec, round-trip time = yyy
   usec, #reads = zzz
   ```

9. Close the socket.

Server.cpp

Your server program must receive the following two arguments:

1. **port**: server's port number
2. **repetition**: the repetition of client's data transmission activities. **This value should be the same as the "repetition" value used by the client.** It means the number of iterations a server shall perform on "read" from the socket. If a client performs "repetition of 100" data transmissions, then the server shall perform at least "repetition of 100" reads from the socket to completely receive all data sent by the client.
   Again, the value "repetition" shall be the same for both client and server to use.

The main function should be:

1. Accept a new connection
2. Create a new thread
3. Loop back to the accept command and wait for a new connection

The server must include **your_function** (whatever the name is) which is the function called by the new thread. This function should:

1. Allocate **databuf[BUFSIZE]**, where BUFSIZE = 1500.
2. Start a timer by calling **gettimeofday**.
3. Repeat reading data from the client into databuf[BUFSIZE]. Note that the **read** system call may return without reading the entire data if the network is slow. You must repeat calling **read** like:

   ```
   for ( int nRead = 0;
        ( nRead += read( sd, buf, BUFSIZE - nRead ) ) < BUFSIZE;
        ++count );
   ```

   Check the manual page for **read** carefully.

4. Stop the timer by calling **gettimeofday**, where stop - start = data-receiving time.
5. Send the **number of read( ) calls made**, (i.e., **count** in the above) as an acknowledgement.
6. Print out the statistics as shown below:

   ```
   data-receiving time = xxx usec
   ```

7. Close this connection.
8. Optionally, terminate the server process by calling **exit(0)**. (This might make it easier for debugging at first. You would not want to do this on a multi-threaded server).

Your performance evaluation should cover **9 test cases with the following parameters**:

1. **repetition** = 20000
2. Three combinations of **nbufs * bufsize** = 15 * 100, 30 * 50, and 60 * 25
3. Three test scenarios such as **type** = 1, 2, and 3

You may have to repeat your performance evaluation and to average elapsed times.

3. What to submit

You must submit the following deliverables in a zip file. If your code does not work in such a way that it could have possibly generated your results, you will not receive credit for your results. Some points in the documentation, evaluation, and discussion sections will be based on the overall professionalism of the document you turn in. You should make it look like something you are giving to your boss and not just a large block of unorganized text.

Criteria	**Percentage**
Documentation of your algorithm including explanations and illustrations in one or two pages	10
Source code that adheres good modularization, coding style, and an appropriate amount of comments. The source code is graded in terms of (1) correct tcp socket establishment, (2) three different data-sending scenarios at the client, (3) instantiation of the thread on the server, (4) use of gettimeofday and correct performance evaluating code, and (5) comments, etc.	35
Execution output such as a snapshot of your display/windows. Or, submit partial contents of standard output redirected to a file. You don't have to print out all data. Just one-page evidence is enough.	5
Performance evaluation that summarizes performance data in a table. For your reference, the table may have the following columns: test#, type, nbufs, bufsize, repetition, # server reads, server-recv time, client-send time, client rtt	10
Discussion should be given comparing multi-writes, writev, and single-write performance. Discuss how the results would be different (other than just being slower) if you ran this on a slower network (say, 1 Mbps). Additionally, discuss why you want to use a thread to serve the connection rather than serving it in the main function.	40
Total	100

Some general help --

Client

1. Receive a server's port (**serverPort**) and name (**serverName**) as Linux shell command arguments.
2. Retrieve an **addrinfo** structure **servInfo** corresponding to this IP name by calling **getaddrinfo( )**. See https://beej.us/guide/bgnet/html/#getaddrinfoprepare-to-launch (Links to an external site.)
3. Declare an **addrinfo** structure **hints**, zero-initialize it, and set its data members as follows:

```
struct addrinfo hints;
struct addrinfo *servInfo;
memset(&hints, 0, sizeof(hints) );
hints.ai_family      = AF_UNSPEC; // Address Family Internet
hints.ai_socktype    = SOCK_STREAM; // TCP
getaddrinfo(serverName, serverPort, &hints, &servInfo );
```

4. Open a stream-oriented socket with the Internet address family.

```
int clientSd = socket( servInfo->ai_family,
servInfo->ai_socktype, servInfo->ai_protocol );
```

   Note: **servInfo** contains “next” pointer, and you may want to check each available entry, please check https://beej.us/guide/bgnet/html/ for example.

5. Connect this socket to the server by calling **connect** as passing the following arguments: the socket descriptor, the **servInfo** structure and its size.

```
connect( clientSd, servInfo->ai_addr, servInfo->ai_addrlen);
```

6. Use the **write** or **writev** system call to send data.
7. Use the **read** system call to receive a response from the server.
8. Close the socket by calling **close**.

Server

1. Declare an **addrinfo** structure, zero-initialize it, and set its data members as follows:

```
int port = YOUR_ID;  // the last 4 digits of your student id
struct addrinfo hints, *res;
memset( &hints, 0, sizeof (hints) );
hints.ai_family      = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, port, &hints, &res);
```

2. Open a stream-oriented socket with the Internet address family.

```
int serverSd = socket(res->ai_family, res->ai_socktype,
res->ai_protocol );
```

3. Set the SO_REUSEADDR option. **(Note this option is useful to prompt OS to release the server port as soon as your server process is terminated.)**

```
const int yes = 1;
setsockopt( serverSd, SOL_SOCKET, SO_REUSEADDR, (char *)&yes,
sizeof( yes ) );
```

4. Bind this socket to its local address by calling **bind** as passing the following arguments: the socket descriptor, the sockaddr_in structure defined above, and its data size.

```
bind( serverSd, res->ai_addr, res->ai_addrlen );
```

5. Instruct the operating system to listen to **up to n connection requests** from clients at a time by calling **listen**.

```
listen( serverSd, n );
```

6. Receive a request from a client by calling **accept** that will return a new socket specific to this connection request.

```
struct sockaddr_storage newSockAddr;
socklen_t newSockAddrSize = sizeof( newSockAddr );
int newSd = accept( serverSd, (struct sockaddr *)&newSockAddr,
&newSockAddrSize );
```

7. Use the **read** system call to receive data from the client. (Use newSd but not serverSd in the above code example.)
8. Use the **write** system call to send back a response to the client. (Use newSd but not serverSd in the above code example.)
9. Close the socket by calling **close**.

```
close(newSd);
```

You need to include the following header files so as to call these OS functions:

```
#include <sys/types.h>    // socket, bind
#include <sys/socket.h>   // socket, bind, listen, inet_ntoa
#include <netinet/in.h>   // htonl, htons, inet_ntoa
#include <arpa/inet.h>    // inet_ntoa
#include <netdb.h>        // gethostbyname
#include <unistd.h>       // read, write, close
#include <strings.h>      // bzero
#include <netinet/tcp.h>  // SO_REUSEADDR
#include <sys/uio.h>      // writev
```