University of Washington Bothell
CSS503: System Programming
Program 4
Socket Programming

1. Purpose

This assignment is intended for three purposes: (1) to utilize various socket-related system calls, (2) to create a multi-threaded server and (3) to evaluate the throughput of different mechanisms when using TCP/IP do to do point-to-point communication over a network.

2. Client-Server Model

In this program you will use the client-server model where a client process establishes a connection to a server, sends data or requests, and closes the connection. The server will accept the connection and create a thread to service the request and then wait for another connection on the main thread. Servicing the request consists of (1) reading the number of iterations the client will perform, (2) reading the data sent by the client, and (3) sending the number of reads which the server performed.

3. Program Specification

Write a Client.cpp and Server.cpp that establishes a TCP connection and sends buffers of data from the client to server. To start, the client sends a message to the server which contains the number of iterations it will perform (each iteration sends 1500 bytes of data). When the server has read the full set of data from the client it will send an acknowledgment message back which includes the number of socket **read()** calls performed.

The client will send the data over in three possible ways depending on the type of test being performed (see below for details of the three tests).

Client.cpp

Your client program will take the following six arguments:

1. **serverName**: the name of the server
2. **port**: the IP port number used by server (use the last 5 digits of your student id)
3. **repetition**: the repetition of sending a set of data buffers
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

For example,

client csslab6 12345 20000 10 150 1

Will have the client connect to server uw1-320-07 at port 12345. It will perform 20000 iterations of test 1. For test 1, each iteration sends 10 buffers of 150 bytes.

Each iteration will send over 1500 bytes of data. The test type will determine how the 1500 bytes are transferred. For the sake of explanation assume we have the following structure:

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

The three test types are defined as follows:

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(clientSd, databuf[j], bufsize);
}
```

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(clientSd, vector, nbufs);
```

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(ClientSd, databuf, nbufs * bufsize);
```

The client program should execute the following sequence of code:

1. Establish a connection to a server using **getaddrinfo**() and **socket**() calls
2. Send a message to the server letting it know the number of iterations of the test it will perform
3. Perform the appropriate number of tests with the server (measure the time this takes; I would recommend the **chrono** time library)
4. Receive from the server a message with the number **read()** system calls it performed
5. Print information about the test (use **chrono** library to time the

```
Test (1,2, or 3): time = xx usec, #reads = yy, throughput zz Gbps
```

6. Close the socket.

server.cpp

Your server program will take only one argument:

1. **port**: a server IP port

Example:

Server 12345

The main thread of a server will run in a dispatch loop which:

- Accepts a new connection.
- Creates a thread to service that request

The servicing thread will do the following:

1. Allocate **dataBuf[BUFSIZE]**, where BUFSIZE = 1500 to read in data being sent by client
2. Receive a message by the client with the number of iterations to perform
3. Read from the client the appropriate number of iterations of BUFSIZE amounts of data

   Note: the **read** system call may return without reading the entire data buffer. You must repeat calling **read** until you have read a BUFSIZE amount of data. This should be done for each repetition on the server.

4. Send the number of read() calls made as an acknowledgment to the client
5. Close this connection.
6. Terminate the thread

4. Hints on TCP Communication

This assignment focuses on basic connection-oriented socket (SOCK_STREAM) communication between a client and a server process. To establish such communication, those processes should perform the following sequence of operations:

Client

To connect to a server using TCP utilize the **getaddrinfo**() system call to get information about the server being used: https://linux.die.net/man/3/getaddrinfo. This information can be used to create a socket (**socket**() system call) and then connecting to the server (**connect**()) .

Server

The main server thread will create a socket (**socket()** system call) using the SOCK_STREAM protocol. You will bind this socket to a local address (**bind()** system call). The use the **listen()** system call to tell the server how many concurrent connections to listen for (5 should be plenty). Finally, the **accept()** system call will accept the call from the client and return a socket from which to read and write.

Use the Set the SO_REUSEADDR option. This option is useful to prompt OS to release the server port as soon as your server process is terminated.

```
const int on = 1;

setsockopt(serverSd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(int));
```

4. Performance tests

Conduct the following performance evaluation using your program in the UWB Linux lab. You will be measuring the throughput (bps) you achieve with the different test types and buffer size permutations. Have your client and server run on any two nodes of uw1-320-00 ~ uw1-320-15. For these tests set your repetition count to get a steady result (generally 20000 is good enough but please validate). You will want to run the tests multiple times and take averages.

Perform the following 12 tests:

1. Four combinations of **nbufs * bufsize**: 15 * 100, 30 * 50, 60 * 25, and 100 * 15
2. Three test scenarios such as **type** = 1, 2, and 3 (note: test type 3 is the same for each variation)

Create a report as described in the rubric below highlighting your major conclusions.

5. What to Turn in

Please turn in your program to Canvas as a .zip file.

	Criteria	Grade
Code	**Source code** that adheres good modularization, correct system call usage, coding style, and an appropriate amount of comments. **Build Script** which builds your code	5 pts
	Correctness **Execution output** Submit examples of the output your client emits by redirecting to a file. This is just the line which reports the test type, time, throughput and # reads. You don't have to print out all data.	10 pts
Report	**Documentation** of your implementation including explanations and illustration in one or two pages.	
	Performance evaluation showing round trip times and number of reads in an understandable manner. Also clearly show throughput. **Discussion** should cover the following topics: (1) comparing your actual throughputs to the underlying bandwidth, (2) comparisons of the performance of multi-writes, writev, and single-write performance, (3) comparison of the different buffer size / number buffers combinations.	15 pts
	Total	30 pts