CPE 464 Lab 3 – Socket API and using poll()

Due:

- **Lab worksheet (pdf) this Friday at 11:59 pm via Canvas**
- **Lab code – you will demo your code in lab next week. (also, handin your code by end of lab next week)**

Name: _____

Each student must type up their own answers. While you should work as a group, you are required to type up your worksheet and turn in a copy of your work.

I. Since it's a new group introduce yourself to your group

Please turn your cameras on during the group work part of the lab.

Have each person introduce themselves to the group.

a. Name
b. Current Location
c. Major, Year
d. Something fun you have done in the last 3-weeks

First Name	Major and Year

II. Socket API (Use my sockets lecture and Google to find answers to these questions)

1. Write the C language command to create a TCP socket for IPv4/IPv6 family?

2. Using the man pages (Google: man socket), what are the **#include** files needed in order to use the socket() function call:

3. Regarding the listen function()

    a. What does the listen() function do?
    b. What is the purpose (what does it do) of the second parameter to the listen() function?
    c. Do you call the listen() function in the client, server or both?
    d. Is the listen() function blocking or non-blocking?

4. Regarding the bind() function
    a. What does the bind() function do?

b. What information is needed to **name** a socket?
c. Why do you only need to call bind() on the server? (so why do you need to call it on the server but not on the client)
d. If you want to print out a port number related to a socket, how do you do this?
e. How does the client get the server's port number?
f. If we do not need to call bind() on the client, how does the client get assigned a port number? (this question is NOT about the server's port number but about the client getting its own port number)

5. Regarding the accept() function
   a. When will the accept() function return (it's a blocking function)?
   b. What does the accept() function return?
   c. Besides checking it for an error (< 0), how/why/when do you use the return value of the accept() function?

6. When using TCP on the **server** to communicate with clients there are at least two different sockets involved on the **server**. Explain:

7. Regarding the poll() function (Google man poll):
   a. What does the poll() function do?
   b. Explain why we might use the poll() function.
   c. Name/explain the parameters passed into the poll() function.
   d. Regarding the timeout parameter passed to the poll() function
      i. What value will cause the function to block indefinitely?
      ii. What value will cause the function to not block but just look at the socket(s) and return immediately?
      iii. How can the function be made to block for 3.5 seconds?

8. What is the purpose of the MSG_WAITALL flag on the recv() function? If you have a user level packet in the format: 2 byte packet length and then data, give the recv() code to process this packet.

9. How does the server (or client) know that the client (or server) has terminated (e.g. ^C, segfault) when using the recv() call?

10. Regarding the code provided with the lab (it is the same code provided with programming assignment #2). The purpose of these questions is to help you understand the code I have provided.

    i. In networks.c,

       a. Write the line of code that creates the server_socket:

       b. What are the line numbers for the code that is used to name the server's socket?

       c. In the call to listen() what is the value for the backlog?

       d. If the accept() call fails, what is part of the error message that will be printed?

       e. For the client to connect() to the server, what information must be loaded into the sockaddr_in6 (called server in the code) structure?

III. Regarding TCP:

There are several function calls that must be made on a client and server in order to utilize sockets for communications (these are all in networks.c).

1. List the function names in the order which are needed to setup and execute communications using Stream Sockets (get this from the **lecture slides** on TCP).
2. Connect the functions that are part of the connection oriented part of TCP using arrows. (Try to line up related calls e.g. connect() and accept())
3. Looking at the network.c file provided with this lab put the line number of the call to that function in the table.

Client Line #	**Functions called on the Client**	**Functions called on the Server**	Server Line #

IV. Program #2 questions (see program spec):

1. What is the format for the "chat-header" that is attached to every packet?
2. For every application packet that is sent you must do two recv()s. Explain why this is:
3. List with a short description the application commands (e.g. %m) that your client must process from STDIN.
4. Draw a packet flow diagram of the flow using packets with flag 1 and 2.

V. Programming:

1. Compile the myclient/myserver code provide and verify that the client talks with the server.
    a. Modify the Makefile to produce executables called **cclient and server** (these are the executable names needed for program #2)

2. Application Level PDU Discussion

In the next part of the lab, you are going to implement the process for sending and receiving an application PDU in the following format. This is an application level PDU which includes a two byte header and then the payload.

Discussion of this application PDU:

- PDU: Two-byte PDU length in **network order**, then a non-null terminated text message

PDU Length (2 bytes) (header) (in network order)	Non-Null terminated text message (payload) (so no null is sent!!!)

- The PDU length is for the entire PDU (length of header+payload)
- You are creating a simple application level PDU (header + payload)
- As discussed below, the entire PDU must be sent in one send() call
- As discussed below, all messages (both on the client and server) must be received in **two recv() calls** using the **MSG_WAITALL**

3. Implement (in a new .c and .h file you create):

Now implement two functions in new .c/.h files that handle sending and receiving an application level PDU with the format. These functions, described below, are:

- int recvBuf(int clientSocket, uint8_t * dataBuffer, int bufferLen);
- int sendBuf(int socketNumber, uint8_t * dataBuffer, int lengthOfData);

These functions must be in files (.c and .h) separate from my code and your main code.

- **recvBuf**: Implement a receiving function that recv()s an application level PDU. This function includes checking for recv() errors (<0), checking for closed connections (==0) and does the two step recv() process (e.g. using MSG_WAITALL). This function must be in a separate .c file from your client and server code (and must be declared in a .h file). You MUST use this function for all receives on **both the client and the server**.

  Function prototype:

  int recvBuf(int clientSocket, uint8_t * dataBuffer, int bufferLen);[1]

  - The return value is the number of bytes receive/0 if closed/-1 on error

- **sendBuf**: Implement a sending function that send()s an application level PDU in one send(). This function should create the PDU, send() the PDU in one send() call, and check for errors on the send() (<0). This function must be in a separate .c file from your client and server code

[1] The bufferLen should be used for error checking. If the length of the PDU as defined by the first two bytes of the PDU is greater than bufferLen then you wrote a bug in your program (so printf an error message and exit(-1)).

(and must be declared in a .h file). You MUST use this function for all send()s on **both the client and the server**.

Function prototype:

int sendBuf(int socketNumber, uint8_t * dataBuffer, int lengthOfData) ;

- The return value is the number of bytes sent, -1 on error

a) Testing your new recvBuf/sendBuf functions

i. Starting with my server code, modify the server code to use your new sending and receiving functions (code for handling the application level PDUs)

ii. Starting with my client code, modify the client code to use your new sending and receiving functions.

iii. Make sure your new recvBuf/sendBuf functions work before continuing.

b) Modify the Server

i. Make the server receive messages (loop) from the client until the client sends the message with the text **exit** or the client terminates (^C).

ii. After the client ends (either by sending exit or client terminates), server should go back to accept() a new client. The server does not end until it is killed (^C).

iii. Modify the server to handle the case where the client ends without sending an **exit** but instead terminates (e.g. ^C)

iv. Modify the output of the server to include the message length and message text:

e.g.: recv() Len 7, PDU Len: 7, Message: hello

(note the length is 7 because there is no NULL sent as part of the message)

c) **More server modifications - Modify the Server to send back two messages to the client**

When the server receives a message, it should send back (in a proper application PDU) the received text message and then immediately (but in a separate send()) send back a PDU with the message that says: Number of bytes received by the server was: XXX (where XXX is the number of bytes in the application PDU just received by the server.) The 2nd PDU must be sent immediately after sending the first PDU:

```
sendBuf (clientSocket, receivedPDU, pduLength);
sendBuf(clientSocket, length, pduLengthMessage);
```

The purpose of sending the second PDU is to test the client code's use of the 2-recv() process.

d) Modify the Client

i. Modify the server to use your new sending and receiving functions (for handling PDUs)

ii. Make the client loop until the **exit** is entered. If exit is entered, the client should send the **exit** message before terminating.

iii. Modify the client to receive the two messages sent back by the server and to print out these messages with the words "Recv() from Server: " before each message. The client should send its message (PDU) and then wait for this two message reply. Before prompting for more user input.

Pseudo code:

```
Prompt for user input
Send PDU with user input
Receive first server response and print
Receive second server response and print
Go back to prompt for user input
```

Verify that your client and server work as discussed above. Note – at this point the server can only process one client at a time. The next section fixes this.

e) Modify the server to use poll() or select() to handle multiple clients

i. This step builds off of the code you finished in the steps above

ii. <u>You will demo this during lab next week (lab week 4)</u>

iii. Modify the server so that it accept()s new clients while at the same time receiving packets from currently active clients.

iv. This must be done in a single process (so no threads, no fork()). You must use poll()/select() on the server to process the multiple sockets. Feel free to use my poll() library. The poll() library is part of the tar file you downloaded for this program.

Save this code – you will demo it during lab next week.
This code should also help you start program #2.

You need to handin in your all of your code including the Makefile, client and server code and the network code (and any other code we need to compile these two programs).

Handin using the **lab3_sockets** directory e.g.: **handin csc-cpe464 lab3_sockets** *.c *.h Makefile README