Lab 4: Network Chat
CMSC 242
Due: April 7, 2016 by 11:59:59pm

Network chat is a typical example of the use of sockets to implement a client/server application. The client connects to the server and relays everything the user types over the remote connection. It also listens for incoming network traffic and displays it on the screen. The server listens for incoming connections. It broadcasts any incoming traffic from a client to all the other clients.

Step 0. Setting Up

Create a folder named Lab4. In that folder, create the following Makefile:

```
all: server client

server: server.c
	gcc -g server.c -o Server

client: client.c
	gcc -g client.c -o Client

.phony: clean

clean:
	rm -f Server Client *.o
```

Note: You must use TAB characters to indent lines in a Makefile. Unfortunately, depending on your settings, vim may replace a TAB character with spaces (this is called "soft tabbing" and is often a very good thing). To get a TAB character in vim, type "i" to get into insert mode, then type CTRL-V followed by the TAB key. CTRL-V puts vim into "verbatim mode" where it embeds exactly what you type.

A Makefile is sort of a recipe for building a program. Once you've created a Makefile, you can just type "make" to build your project. The "make" program is smart – it knows that the program "Server" is built from the file "server.c" and so it won't bother recompiling it if you haven't made any changes to that file. This can save a lot of compile time.

If, for some reason, you need to force the Makefile to recompile everything, you can type "make clean" to delete your compiled programs (it will leave the source code alone) and then type "make" again.

Step 1. Server

Create a file named "server.c". In that file, type:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/types.h>
```

These includes pull in all the system calls and structs you will need to create a server. However, we are going to create a specific type of server: a chat room. Our chat room needs to keep track of some internal state, so add this to your file:

```
struct server_state {
        int listen_socket;
        int client_fds[40];
        int num_clients;
        int maxfd;
        fd_set monitor_fds;
};
```

The listen_socket is the socket on which we will listen for incoming connections. When we receive an incoming connection, we will accept it and create a new socket for it that we will store in the "client_fds" array and increment "num_clients" so we can keep track of how many users are connected. We will limit the number of allowed connections to forty.

The monitor_fds set (and the maxfd value) will allow us to perform I/O multiplexing on our sockets, responding to an incoming message from any of the clients.

One important software engineering principle is the use of "modularization". This means that we should split a complex program up into smaller pieces that can be more easily tested and understood. One of the tasks our server needs to perform is the initial setup of a listening socket. Add the following code to the bottom of the server.c file:

```
int setup_listen_socket() {
        int sockfd;
        struct sockaddr_in list_addr;
        int result;

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0) {
                perror("socket");
                exit(-1);
        }

        memset(&list_addr, 0, sizeof(struct sockaddr_in));
        list_addr.sin_family = AF_INET;
        inet_pton(AF_INET, "0.0.0.0", &list_addr.sin_addr.s_addr);
        list_addr.sin_port = htons(4000);


        printf("Binding socket.\n");
        result = bind(sockfd, (struct sockaddr *)&list_addr, sizeof(struct sockaddr_in));
        if (result < 0) {
                perror("bind");
                exit(-1);
        }

        printf("Listening.\n");
        result = listen(sockfd,1);

        if (result < 0) {
                perror("listen");
                exit(-1);
        }
        return sockfd;
}
```

To set up the listening socket, we first use the "socket" system call to create a TCP/IP socket (SOCK_STREAM means TCP, while AF_INET means IP) and check for errors.

We then create a "sockaddr_in" address structure that will tell our socket which IP address and TCP port number to listen on. We first zero out the memory for the address structure, then set it to be an IP address with value 0.0.0.0 on port 4000. We have to convert 4000 from host format to network format, because Intel systems are little-endian, but IP headers are big-endian.

Once we've set up the address structure, we bind the socket to that address using the "bind" system

call. We then use the “listen” system call to set up a queue for incoming connections and begin listening for incoming traffic.

When a new client connects, we will need to create a new socket we can use to communicate with just that client. This must be separate from our listening socket so that it doesn’t interfere with any new/incoming connections. For good modularity, let’s split that out into a separate function:

```
int connect_client(struct server_state* state) {
        int client_sock;
        int result;
        struct sockaddr_in client;
        socklen_t addrlen;

        //Too many clients
        if (state->num_clients >=40)
                return -1;

        client_sock = accept(state->listen_socket, (struct sockaddr *)&client, &addrlen);

        //Accept error
        if (client_sock < 0) {
                perror("accept");
                return -1;
        }

        state->client_fds[state->num_clients] = client_sock;
        state->num_clients++;

        if (client_sock > state->maxfd)
                state->maxfd=client_sock;

        FD_SET(client_sock, &state->monitor_fds);

        char buf[1024];
        strncpy(buf, "Welcome to network chat.\n", 25);
        buf[26]='\0';
        result = write(client_sock, buf, 1024);
        if (result < 0) {
                perror("write");
                exit(-1);
        }

        return 1;
}
```

We first check to see if we already have too many connections. If so, we don’t want to terminate the program, but we do want to return. We use -1 as a return value that indicates an error has occurred.

If we have room for a new connection, we use the “accept” system call to receive it. Accept will create a new socket for the incoming client, which we put in the file descriptor “client_sock”. After checking for errors, we update our server state by adding client_sock to the array of file descriptors, adding the new socket to the monitor_fds list, updating maxfd (if necessary), and incrementing the number of clients. We also send a short “welcome” message on the new socket. Note that, for simplicity, we always read and write in 1024-byte (1Kb) blocks.

We are now ready to put all these tools together to make a chat server. Add this main function to the bottom of your file:

```
int main(int argc, char* argv[])
{
        int result;
        char cmd;

        int i, j;

        struct server_state state;

        fd_set read_fds;
```

```
        fd_set except_fds;

        state.num_clients = 0;
        state.listen_socket = setup_listen_socket();

        FD_ZERO(&state.monitor_fds);
        FD_SET(0, &state.monitor_fds);
        FD_SET(state.listen_socket, &state.monitor_fds);
        state.maxfd = state.listen_socket;
        cmd = ' ';

        printf("Server ready.  Type 'q' to quit.\n");
        while (cmd != 'q') {
                read_fds = state.monitor_fds;
                except_fds = state.monitor_fds;
                result = select(state.maxfd + 1, &read_fds, NULL, &except_fds, NULL);
                if (FD_ISSET(state.listen_socket, &read_fds)) {
                        result = connect_client(&state);
                }
                if (FD_ISSET(0, &read_fds)) {
                        cmd = getc(stdin);
                }
                for (i=0;i<state.num_clients; ++i) {
                        if (FD_ISSET(state.client_fds[i], &read_fds)) {
                                char buf[1024];
                                result = read(state.client_fds[i], buf, 1024);
                                if (result <= 0) {
                                        printf("Client %d disconnected.\n", i);
                                        FD_CLR(state.client_fds[i], &state.monitor_fds);
                                        close(state.client_fds[i]);
                                        state.num_clients--;
                                        state.client_fds[i] = state.client_fds[state.num_clients];
                                }
                                else{
                                        printf("Client %d: %s\n", i, buf);
                                        for (j=0; j < state.num_clients; ++j) {
                                                if (i == j)
                                                        continue;
                                                result = write(state.client_fds[j], buf, 1024);
                                                if (result <= 0) {
                                                        perror("write");
                                                        exit(-1);
                                                }
                                        }
                                }
                        }
                        if (FD_ISSET(state.client_fds[i], &except_fds)) {
                                        printf("Client %d disconnected.\n", i);
                                        FD_CLR(state.client_fds[i], &state.monitor_fds);
                        }
                }
        }

        for (i=0; i < state.num_clients; ++i)
                close(state.client_fds[i]);
        close(state.listen_socket);
}
```

This is kind of a complicated program (we should probably split it up further for better modularity!), but essentially, it listens for either:

A. A keypress

B. An incoming message from a client

C. A “hangup” from a connected client

If it detects a keypress, it sets “cmd” to the value of that key. Pressing ‘q’ causes the while loop to end, terminating the server. Other commands are ignored (for now).

If it detects incoming messages, it reads the message into a buffer, then writes the message back out to ALL clients (except the one that sent the message).

If a client disconnects, we remove it from the set of monitored file descriptors, but not from the other data structures, so it still counts against our forty client limit. We could fix this, but it would involve some complicated data structures work, so let's leave it for now.

Step 2. Client

Now make a file named "client.c". In that file, type:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char* argv[])
{
        int sockfd;
        int result;
        fd_set listen_fds;
        fd_set read_fds;
        fd_set except_fds;
        struct sockaddr_in addr;
        char buf[1024];

        if (argc < 2) {
                printf("Syntax: Client <ip address>\n");
                exit(-1);
        }

        //Create a network socket
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0) {
                perror("socket");
                exit(-1);
        }

        addr.sin_family = AF_INET;
        addr.sin_port = htons(4000);
        inet_pton(AF_INET, argv[1], &addr.sin_addr.s_addr);

        printf("Connecting...\n");
        result = connect(sockfd, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));

        if (result < 0 ) {
                perror("connect");
                exit(-1);
        }

        printf("Connected.\n");

        FD_ZERO(&listen_fds);
        FD_SET(0, &listen_fds);
        FD_SET(sockfd, &listen_fds);

        buf[0] = '\0';
        while (strncmp(buf, "quit", 4) != 0) {
                read_fds = listen_fds;
                result = select(sockfd+1, &read_fds, NULL, &except_fds, NULL);
                if (FD_ISSET(0, &read_fds)){
                        fgets(buf, 1024, stdin);
                        write(sockfd, buf, 1024);
                }
                else {
                        result = read(sockfd, buf, 1024);
                        if (result <= 0) {
                                printf("Connection terminated.\n");
```

```
                                printf("%s", buf);
                                strncpy(buf, "quit", 4);
                        }
                        else {
                                printf("%s", buf);
                                buf[0] = '\0';
                        }
                }
        }
        close(sockfd);
}
```

Notice that the client is much simpler than the server. Test your code by typing "./Server" in one terminal and then typing "./Client 127.0.0.1" in another terminal. Open up additional clients in other terminals.

Step 3. Questions

In a file named "Questions.txt", answer the following questions:

1. What port does the network server run on?

2. Does the chat program use TCP or UDP? How do you know?

3. How many simultaneous clients are allowed connect to the server?

4. What does the "connect" function call in the client do?

5. The address passed to inet_pton in the server is "0.0.0.0". Why can we connect to "127.0.0.1"?

6. The client takes an IP address. What system call would we use to allow it to take a hostname?

7. This server runs "single-threaded" – there is no forking or spawning of new threads. Why is that probably not a good way to design the program?

Submitting
As usual, create a tarball of your entire Lab4 folder:

cd ..
tar czvf Lab4.tar.gz Lab4/

and submit through the course web site.