Operating Systems

HOCHSCHULE DER MEDIEN

Exercise E: Network chat

One of the many responsibilities of an operating system is to provide access to network communication. In this exercise we are going to write a little chat program using a TCP connection.

1. Find a partner

For this exercise you'll need a partner. One of you will create the server program, the other one the client program.

- To chat you need someone to chat with. Find someone to do the exercise with.
- Agree on who will create the server and who will create the client program. The server is a bit more code to write (5 system calls instead of 2).
- Help each other out. Ask your partner if you run into a problem. Sometimes it helps just to explain the problem to someone else to actually understand what's wrong. ;)

2. Get what you need

- Create a new c file "chat.c" for your code.
- Download network_helper_functions.c from the personal schedule and put the code in it into chat.c.

3. Basic terminal echo program

We'll start out with the basic parts, one thing at a time. For a chat program we'll need a way to enter a message and print messages. So we'll do just that first, write a program that will read a line from the terminal and print it back on the terminal (often called an "echo" program). Later on we'll make it read and write data from a network connection instead of the terminal.

- In your main() function create a buffer that can hold 1024 bytes.
- Use the read() syscall with STDIN_FILENO as first argument to read a line from the standard input. read() waits until you've entered a line and put it in your buffer. Remember to track how many bytes read() put into your buffer.
- Use the write() syscall with STDOUT_FILENO as first argument to write the data in your buffer back to the standard output.

Operating Systems

- T Read and write in a loop until the standard input is closed. That is read() from STDIN_FILENO returns 0. When your program runs you can press ctrl + d (strg + d) to close the running programs standard input.
- T Test it. Each line you enter into your program should be printed again as output until you press ctrl + d.

Hints

These functions may be helpful to use. Use the manpage to get further information:

- read()
- write()

Includes

```
#include <unistd.h>
```

Operating Systems

4. Get a socket address from command line options

The next step in our networking crusade is to actually get a socket address into our program. The server program will listen on that address for incoming connection. The client program will connect to that address.

We'll use the sockaddr_from_str() function to convert strings like “127.0.0.1” and “12345” into a binary socket address with a host and port in it. We'll take the strings from our programs command line options (remember, the stuff our main function gets in these argc and argv parameters). They're nice strings given to us by the operating system just waiting to be used. And we can change them each time we start our program without needing to recompile anything.

T Change your main functions signature into:

```
int main(int argc, char* argv[])
```

Now you can use the argc and argv variables to access the command line arguments of your program.

T Print the error message "usage: ./chat address port\n" if your program is executed with less than 3 arguments (argv < 3). The first argument will always be the name of the program itself ("./chat", the first word of the command you typed). We'll use the 2nd argument as the network address we want to bind or connect to. The 3rd argument is the TCP port used by our chat program. If you printed the error message also exit the program (e.g. with "return 1"). No point executing something if we don't have the arguments we need.

T Create a variable to store a socket address (type struct sockaddr_storage) and another one to store the actual size of the address (type size_t).

T Use the sockaddr_from_str() function to convert the network address string (argv[1]) and port number string (argv[2]) into a binary socket address. Tell it to store the socket address and size of the address in the variables from above.

5. For the server: The wait_for_first_connection() function

Skip this exercise if you write the client program. Continue with the next one instead.

For the server program the time has come to tell the system what it wants: It should open a server socket at the specified address and establish a connection with the first one who wants to talk to us. Then it should close the server and return the file descriptor of the established connection.

We'll put all of that into our very own wait_for_first_connection() function and call it later on from main(). This is just another building block for our crusade… so unfortunately we can't test our server just yet.

- Create the following function right before your main() function:

  ```
  int wait_for_first_connection(struct sockaddr_storage
  *bind_addr_ptr, size_t bind_addr_size) {

      // add your server code here

  }
  ```

  We'll put all the code our server needs in here.
- Create a new socket with the socket() system call. Take the value for the first parameter out of the "ss_family" member of the bind address. It tells the operating system if we want an IPv4 or IPv6 socket (depending on which kind of address we entered in the command line). Use SOCK_STREAM for the second parameter. This will give us a network connection that transmits a stream of bytes we can read and write from (just like a file). You can set the "protocol" parameter to 0. We don't care how we get what we want.
- Bind the socket to the address. You'll have to cast the "addr" parameter to "struct sockaddr *". This is one of the few cases where we actually know it better than the compiler who tries to enforce stuff written in 1983...
- Bring the socket into listen mode. A typical value for "backlog" is 3. From now your server is open for incoming connections.
- Use accept to wait for and accept the first incoming connection into a new socket. Use NULL for the 2nd and 3rd parameter.
- Close the server socket (we don't want to accept any other connections but the first one).

T Return the file descriptor of the established connection.

Hints

These functions may be helpful to use. Use the manpage to get further information:

- socket()
- bind()
- listen()
- accept()
- close()

Includes

```
#include <sys/socket.h>
```

6. For the client: The establish_connection_to() function

Skip this if you write the server program.

For the client program the time has come to tell the system what it wants: It should open a socket and connect it to the target address. This should give our program a network connection to the server listening on that target address.

We'll put all of that into our very own establish_connection_to() function and call it later on from main(). This is just another building block for our crusade… so unfortunately we can't test our client just yet.

T Create the following function right before your main() function:

```
int establish_connection_to(struct sockaddr_storage
*target_addr_ptr, size_t target_addr_size) {

    // add your client code here

}
```

T Create a new socket with the socket() system call. Take the value for the first parameter out of the "ss_family" member of the target address. It tells the operating system if we want an IPv4 or IPv6 socket (depending on which kind of address we entered in the command line). Use SOCK_STREAM for the second parameter. This will give us a network connection that transmits a stream of bytes we can read and write from (just like a file). You can set the "protocol" parameter to 0. We don't care how we get what we want.

T Connect the socket to the target address. You'll have to cast the "addr" parameter to "struct sockaddr *". This is one of the few cases where we actually know it better than the compiler who tries to enforce stuff written in 1983...

T Return the socket file descriptor. It represents our network connection to the server.

Hints

These functions may be helpful to use:

- socket()
- connect()

Includes

```
#include <sys/socket.h>
```

7. Put in the network

Now we're going to take your read() / write() loop from the start of the exercise and extend it so we get a network chat.

- Create an “int connection_fd” variable in your main function (right after you converted the command line arguments to a binary socket address)
- If you're programming a server call your wait_for_first_connection() function and put the returned value into connection_fd.
- If you're programming a client call your establish_connection_to() function and put the returned value in connection_fd.
- Within the loop duplicate your read() / write() pair so you have the same code twice. Just doing the same thing twice in a row.
- Change the first piece of code to write to connection_fd instead of STDOUT_FILENO. Now this piece of code reads a line from your terminal and sends it over the network connection.
- In the second piece of code read from connection_fd instead of STDIN_FILENO. This piece now receives a message from the network connection and prints it on your terminal.

8. Do some chatting with your partner

Now we have everything in place for a basic network chat. Time to go for a test drive.

- One of you has to run the server. Use the command "ip address list" to get the IP addresses of your computer. Run your server program with a link-local or organization IP as a parameter. Use 12345 as a port number. Here's an example (but remember to put in *your* IP address!):

  ```
  ./chat aaa.bbb.ccc.ddd 12345
  ```

- The other one has to use the client program to connect to the IP the server runs on. You also have to use the port number 12345. The command is the same as above.
- Now you should be connected. Send some messages! Be aware: You have to send a message before you can receive one (just press enter to send a message with just a line break in it).

9. Optional: Merge server and client into the same program

You should now have two programs: One server program with an wait_for_first_connection() and one client program with a establish_connection_to(). As the final part of our conquest we'll put both function into the same program and add a command line switch to decide if we want to run a server or client.

With that each of you can run both, a server or a client.

- Copy your partners wait_for_first_connection() or establish_connection_to() function into your program.
- Add a 3rd command line argument to your program: If it is the string "server" use wait_for_first_connection() to get the file descriptor of the connection (the stuff we put into the connection_fd variable). If it's "client" use establish_connection_to() instead.
- You can use the function strcmp() to compare two strings like this:

```
if ( strcmp(argv[3], "server") == 0 ) {
    // 3rd command line argument is "server"
}
```

- Remember to modify the argc check at the start of your program and update the usage error message.
- Tip: You can also chat with yourself by running the server and client in two different terminals on the same computer.

Hints

These functions may be helpful to use. Use the manpage to get further information:

- strcmp()

Includes

```
#include <string.h>
```