CS407/CS591 – Adv. Linux Programming – Fall 2017

Lab #1
(Due: Friday 9/1/17)

The subject of the first lab assignment is writing a network *client-server* application that allows users to run arbitrary *Bash shell command lines* on a (remote) server machine. I.e., it is to be similar to basic *Telnet* functionality (or *SSH* shell sessions but without the encryption). The server is to allow *multiple clients* to *simultaneously* connect and run *Bash sessions* on the server. The client is to connect to the server, then read shell command lines from the user on its local keyboard, pass them on to the server, and print all output that comes back from the server to the local display.

One goal of the assignment is simply to get everyone back in practice with C system call programming. The assignment is at the level of typical CS306/CS491 final lab assignments (in fact, it has been used as the final lab assignment in some semesters). It will require that students be able to work with *sockets* and implement *multi-process programs*. The relevant CS306/CS491 lecture videos/slides are available from the course webpage for review.

Another goal of this assignment is to develop the code that will serve as the basis for several future lab assignments. There is a vast difference between a CS306-level client-server, and what is required in production-quality versions, where the server can handle many thousands of simultaneous clients. Over the next few labs, the basic client and server developed for this assignment will be transformed into increasingly robust and capable versions.

The rembash Application Protocol:
A critical aspect of any client-server application is the *protocol*: the specific ordering and data for exchanges between the client and server. We will refer to the particular *protocol* to be followed by your server and client as the *rembash application protocol*. It is critical that your programs precisely follow the rembash protocol so that they can interoperate with other rembash clients and servers.

Let us summarize the server part of the protocol: When a rembash client first connects to the server, some data is initially exchanged between the client and server, including a *"secret word"* to provide some security. If all is good, the server will `fork` off a new subprocess in which to run Bash, and continue accepting additional client connections. In the newly created subprocess, *standard input, output, and error* must be *redirected* to the client socket, so that when Bash reads/writes, it will be doing so from/to the socket (and thus the client). This will allow the remote client's user to interact with Bash just as if s/he were logged into the server machine. After this, Bash (`/bin/bash`) will be `exec`'d in the subprocess (it will run until the user types `exit` or the client connection closes).

Now the client part of the protocol: When the client program first connects to the server, it is to wait for the server to start the protocol exchange. Once initiated, the client exchanges protocol data with the server (including the "secret word"). After this exchange completes successfully, the client is to `fork` off a subprocess. One client process will read commands from the terminal/user and send them via the socket to the Bash process on the server. The other process will read any output coming back from the socket and print it out on the local terminal. The client should continue until the user types the `exit` command, causing the remote Bash session to terminate and close the connection.

Here is the *server* side of the *rembash application protocol* in detail:
(`\n` indicates a newline char, all other chars should be taken literally)

1. Upon establishment of a new TCP connection from the client, the server sends to the client: `<rembash>\n`
2. It then waits to read the *"shared secret"* sent by the client as: `<`*shared secret text*`>\n`
3. If the client secret does not match server's secret, the server should send: `<error>\n` and then close the connection
4. If the client secret does match, a Bash subprocess must be created for the new client:
   (a) create (`fork`) a new *subprocess*
   (b) redirect *standard input, output, and error* in that process to the client socket
   (c) `exec /bin/bash` (redirections remain in effect)
   (d) let client know shell is ready by sending: `<ok>\n`
   (e) when the Bash subprocess eventually terminates, the client socket is to be closed
5. Simultaneously with creating the new Bash subprocess, continue looping accepting further client connections

Here is the *client* side of the *rembash application protocol* in detail:

1. Upon establishement of a TCP connection to the server, the client waits to read the protocol ID message: `<rembash>\n`
2. If the correct ID is received, the client sends: `<`*shared secret text*`>\n`
3. The client now waits for the server response: `<ok>\n`
4. Once this is received, the client is to simultaneously:
   (a) loop, repeatedly reading a (command) line from standard input (the terminal/user) and writing the line to the server socket; and
   (b) loop, reading output from the server socket and writing it (just as received) to standard output (the terminal)
5. The client is to continue both loops until the server closes the connection. This should happen only after the `exit` command is typed in the client, and the related Bash process on the server, terminates

For ease in parsing responses, the rembash protocol specifies that all messages are *terminated by newlines* (remember that TCP is a *stream-oriented protocol*). A function called `readline()` will be made available to you via the course website. This function will be able to read and return (arbitrary length) *lines* from an open socket (actually, from any file descriptor). You are free to use this function to read the messages from the sockets, or you can write your own comparable function. If you write your own function, it must be capable of handling arbitrary length lines. If you use `readline()`, leave it in the separate file `readline.c` and simply `#include` it. *You are absolutely not to cut and paste provided readline code into your own code!* Also, be certain to read all the comments in the `readline.c` file so you understand the limitations/requirements of the function.

To validate that your programs properly follow the rembash protocol, your client and server programs will not only be tested with each other, your client will be tested against the server solution and your server will be tested against the client solution. Thus, it is critical that you precisely follow the rembash application protocol spec. If you fail to precisely follow the

rembash protocol, your programs may fail to function with the solution programs. This will represent a bug in your programs, and will cause you to lose points. Having clear protocols for applications and having programs that precisely adhere to those protocols is critical for client-server applications.

Program Submission:
You are to produce both a *server* program and a *client* program, so this lab is to be implemented in *two separate files*. These files are to be named: (1) `lab1-server.c` and (2) `lab1-client.c`. Each file must be able to be compiled into an executable, which means they must contain a `main` along with any needed header includes, etc. You are to submit your two *C source files* using the Lab #1 Dropbox on the course D2L website. Do not submit any additional files! In particular, do not submit "project" files created by an IDE you may be using. Submit only the required two files, properly named!

Server Program Details:
The server program is not to take any command line arguments. You are to implement what is known as a *parallel server*, meaning it can *handle multiple simultaneous clients*. For this lab, you are to meet that goal by `fork`'ing off a *new subprocess* in which to run Bash for each client. (In later labs we will talk about other issues that may need to be dealt with.)

In addition to a `main`, your server file must contain at least one function:

- `void handle_client(int connect_fd)`
  - This function should be called by the main server loop to do all the work involved handling each client/connection.
  - `connect_fd` is the file descriptor of the socket associated with a connection to a particular client.

Structuring your program with the function `handle_client()` will facilitate extensions that we will make in future labs. It will also make your server code much cleaner than it would be if protocol handling were mixed into the main server loop code.

When run, the main server process is to do the following:

1. Create and set up a *listening TCP socket*.
2. Go into an *infinite loop* `accept`'ing new connections from clients.
3. When a new connection is received, the function `handle_client()` is to be called and passed the FD for the new client/connection.

`handle_client()` is to implement the rembash application protocol as specified above. This requires an initial exchange of data, then the creation of a subprocess in which Bash (`/bin/bash`) will be `exec`'d. For Bash to be able to work with a remote client, *standard input, output, and error* for the Bash subprocess must be associated with the client/connection *socket* (rather than the terminal). This requires *redirection*: changing what file descriptors 0, 1, and 2 are associated with, from the terminal to the connection socket. Redirection is accomplished using the `dup2()` system call.

Note that it may not be obvious that we can even change standard in/out/error for Bash, since we cannot insert code to be run between the `exec` call and Bash starting. What makes

it possible to have the redirections for Bash is that *file descriptors are inherited across an exec call.* By redirecting the standard in/out/error file descriptors prior to `exec`'ing Bash, the redirections remain in effect once Bash is started by `exec`.

Remember that good practice is for child processes to eventually be *collected*. This is particularly important with a long-running program such as a server, where failure to collect children could lead to the inability to `fork` off new subprocesses. Your server code must ensure that all children are eventually collected. The problem with a server is that you do not know whether a child process will have terminated at a particular point in your main server loop. This means that you cannot use `wait()` to collect children, as this could cause your server to hang until some child/client terminates (not allowing you to handle further clients). You do not need to know the exit status of children here, so the easiest way to handle the collection of children is to simply tell the kernel to automatically remove children's results when they terminate. You can do this by setting the *signal* `SIGCHLD` to be *ignored*:
`signal(SIGCHLD,SIG_IGN)`.

Running Bash the "Easy" (but Not Correct) Way:
For this lab, we are going to "cheat" when running the Bash subprocesses, to simplify our implementation. Bash expects to interact with a *terminal* (TTY) via standard in/out/error, not a *socket*. Properly interacting with a remote client requires the use of a *pseudo TTY* (PTY): a software device that simulates a TTY. Since setting up a PTY is somewhat complicated, we will leave this for the next lab.

Becuse we are not using a PTY, there are a few "tricks" you need to know to `exec /bin/bash` and get it to interact properly with the client program via the socket. The reason you have to redirect standard in/out/error prior to `exec`'ing `bash` is because `bash` reads/writes via standard in/out/error. It turns out, however, that getting `bash` to run in *interactive mode* when `exec`'ing it (especially without a PTY) requires passing some options. Here is the `exec` call you should use:
`execlp("bash","bash","--noediting","-i",NULL)`.

In fact, we will see that there is still a serious problem that occurs when trying to run multiple Bash sessions simultaneously in multiple subprocesses! However, rather than just tell you what the problem is and how to fix it, I want you to encounter it and research it yourselves. (We will eventually then discuss the problem in class, before the lab is due.)

Client Program Details:
Your client program is to take *one command-line argument*: the "IP address" of the server machine to connect to, in *dotted decimal/quad* format (recall that this is not truly an IP address, since IP addresses are 32-bit binary numbers). An example call could be:
`./client 131.230.133.20`

Using IP addresses rather than *hostnames* avoids having to deal with DNS calls. The IP addresses of lab machines is printed on them, and can be found using commands like `host` or `dig`. The two remotely accessible lab machines have these IP addresses: 131.230.133.20 (pc00) and 131.230.133.21 (pc01).

When your client program is executed, it should:

1. Initiate a TCP connection to the server on the specified machine.

2. Follow the initial exchange in the rembash application protocol.
3. `fork` off a new subprocess.
4. In one process, enter a loop and repeatedly read output from the server socket and write it to standard output.
5. In the other process, enter a loop and repeatedly read (command) lines from standard input and write them to the server socket.
6. The client should terminate when a file-end return is gotten reading from the socket or from the terminal (or any errors occur).
7. Collect the subprocess before terminating (though this is not so critical).

The client must simultaneously: (1) read commands being typed by the user (at the terminal) and send the commands to Bash, plus (2) read Bash output coming back from the socket and print it out on the user's terminal. While this is somewhat complicated to do with a single thread of execution, it is straightforward if we use multiple processes: (1) one process reads command lines from standard input and writes them to the socket; (2) another process reads all data from the socket and prints it to standard output.

It is slightly simpler if the *parent process* reads from the socket and the *child process* reads from the terminal. The reason for this is that the usual termination situation will be the socket reading process encountering a file-end return when reading from the socket, and because parents are generally supposed to *collect* their children, it is easier for the client parent process to terminate and collect its child, then exit, than it is for the child to properly clear-nup. Still, because errors can occur in reading from the terminal and writing to the socket, *your client code is to be prepared to cleanly terminate if the child process encounters errors!*

Shared Secret:
Because the server process will be running as you, it is important to provide some security. Otherwise, anyone could potentially connect to your server and issue the command "`rm -r *`" and delete all of your files. In order to provide some basic security, your client and sever will use a *shared secret*. The client will send the secret to the server in order to authenticate itself and be allowed to connect. The shared secret can be any text string (of any length). See below for the secret that your submitted code must use—but use a different secret during your development and testing of course!

Additional requirements and details:

- You must use Linux/UNIX system calls *for all socket I/O*, but you can use C library I/O functions rather than system calls *for terminal I/O*.
- To simplify getting commands from the terminal, you may use `fgets()` and assume that the length of any command will not exceed 512 characters.
- Remember to check the man pages for the socket-related calls to determine header includes and other socket setup (e.g., `<sys/socket.h>` and `<arpa/inet.h>`).
- As the *port* for your service, use 4070 (a high, normally unused port number). Use `#define` syntax to create a symbol `PORT` to define this port number in both programs. Note that when testing this on the Dept. Linux machines, you should use a different, random high port to avoid conflicts with other students' programs. Be sure to set the port back to 4070 before submission.

- Your *shared secret* is effectively a kind of password/passphrase, which must be common to both the server and client. To make it easy to change, use `#define` syntax in both the client and server to create a symbol `SECRET` to hold your secret word/phrase. Remember to set your secrets to "`cs407rembash`" before submitting your programs!
- Normally, when you terminate a server abnormally, the port remains "in use" for some period of time, which can make testing annoying. You can tell the kernel to allow immediate reuse of a port by setting the `SO_REUSEADDR` option on your listening socket. This can be done by including the following two lines:

```
int i=1;
setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
```

- Remember that you must use the `htons()` and/or `htonl()` functions to convert ports into *network byte order* (since X86 machines are little endian).
- Remember that a dotted quad input address (as a string) must be converted to a *true* (32-bit binary) IP address before it can be used in a socket address. You should use the function `inet_aton()` (`inet_addr()` is the older, but now deprecated function to do this).
- To allow the server to easily be run on different machines, use the "address" `INADDR_ANY` to specify that the server should listen on all local interfaces.
- For socket I/O, you can use `read()`/`write()` or the special socket I/O calls `recv()`/`send()`; the two sets are equivalent without `flags` arguments with `recv()`/`send()`.
- Once you redirect standard input, output, and standard error in the subprocess, any output that process tries to do—including debugging statements—will end up going to the socket. You can use `dup` to "save" file descriptors prior to redirection if you need to restore them.
- Your program must error check *all appropriate* system and library calls. In case of errors, it must print appropriate error messages to *standard error*.
- Any errors in the client should cause the client program to terminate.
- Errors setting up the socket (or `fork`'ing off Bash processes) should cause the server to terminate with an appropriate status. However, errors while handling a connection should simply cause the connection to be closed—they should not affect the main server process (it should continue accepting new connections).
- Try to write *robust code* as much as is practical—i.e., code that can deal with all possible conditions and errors. Doing so will be a major topic in this course. In particular, try to make certain that your code cannot leave subprocesses running or leave socket FDs in use in case of errors.
- The server should run *forever*—i.e, until it is terminated (via *control-c* or `kill`). Most programs that are intended to run "forever" are *daemons*, but being a daemon program involves more than just an infinite loop, they must be able to run "in the background" (i.e., without a *controlling terminal*). We will look at how to properly "daemonize" the server in a future lab. For now, simply use one terminal window for your server, and separate ones for each client when testing.

Cheating:
Every single line of code in your submissions is to have been written by you alone (or given to you by the instructor with permission to include it in your own code). Working in groups and/or submitting code written by someone else (including code from the Internet) will be considered **cheating**. If found to have cheated, you will receive a zero on this assignment, and be subject to very stringent conditions for remaining in the course. Graduate students who hold CS Dept. TA positions may be dismissed from those positions and will absolutely be barred from further TA positions if caught cheating.

You are allowed to discuss *general approaches* with other students in the class, just as you are free to use the Internet as a source for learning about how to accomplish particular tasks required of your programs. Using similar approaches as other students is not cheating, as long as you implement them independently in your own programs. Even if two programs use the exact same approach to accomplish some task, there are so many ways most methods can be implemented, that the programs should share very little similar code—unless that code originated from a common source. If you are cutting/pasting code from someone else into your program, you are almost certainly cheating.

Testing:
For testing, you can run the server and client(s) on the same machine. Simply use the *loopback IP address* `127.0.0.1` with the client. The easiest way to test is to open multiple shell windows, and run the server in one, and client(s) in others. Be sure to test your server with *multiple simultaneous clients*, as this is necessary to uncover the problem mentioned above with respect to running multiple simultaneous Bash sessions.

It can sometimes be confusing to tell whether what you are seeing in a shell window is Bash running via your server or Bash running locally, particularly when the client and server are running on the same machine. If you have your Bash prompt showing the CWD, a good approach is to start the server running from a different directory from the clients, so the prompt's CWD will change when you start a shell session via your server. Running `ps` should also give you very different results between server Bash session and normal shell session.

We will pay particular attention to learning how to write robust code in this class. Your client and server code should be able to function reasonably despite an error occuring at any point in either program, and despite loss of the network connection. Test your code for unusual situations by using `^c` (or `kill`) to terminate various processes at random points during execution (do not conduct only tests that end with you typing `exit` in the client). Neither your server nor client should hang, and you should not end up with unwanted processes or open sockets being left.

To see whether you have unwanted processes, you can use `ps` (e.g., "`ps aux`") along with `grep` for `server`, etc. You can also use the `ss` command (or the older `netstat`) to see processes using sockets. E.g., "`ss -atp | grep 4070`" will show you information about the listening server process as well as both server and client processes for connections (obviously replace `4070` with whatever port you are using in your testing). See if you can decode all of the valuable info that this command shows. Note: you will probably have to invoke `ss` as "`/sbin/ss`" when running it as a regular user.

With servers (which run indefinitely), it is very important to reclaim system resources when clients are finished. While it might not be a problem for a short program to leave some

unused processes or open sockets, if this happens with a server, these things can eventually hang the server or even the system. So, it is important to use commands like `ss` and `ps` to make certain that subprocesses and their sockets get removed as clients are finished.

If you end up with *stopped* processes, remember that regular `kill`/`killall` will not terminate them—you must use the "`-9`" or "`-SIGKILL`" options.

If you wish to eventually try running the client and server on separate machines, be aware that SIUC has a border firewall that will prevent you from connecting to your server running on pc00 or pc01 from outside of the university VPN. The best place to make such tests is within the CS Dept. Linux Lab.

Debugging and A Deeper Understanding:
One of the goals of this course is for students to become better at debugging C programs involving Linux system calls. This will also help students gain a deeper understanding of how Linux and C work.

An important starting point is to use `strace` to trace the syscalls that your client and server make when running. Since both your client and server use multiple processes, you will have to use the "`-f`" option to `strace` to follow all processes.

Use of `strace` is often critical to being able to understand why client-server programs fail or hang, particularly when problems happen only occasionally. E.g., if you see the client or server sitting in `read()` when the system hangs, this tells you it is waiting for data to be sent by the other side, but that data is not being sent. It is important that students become accustomed to routinely using `strace` (and `ltrace`) when doing Linux syscall programming.

Using `strace` is particularly valuable to understand the problem that occurs when trying to run multiple simultaneous Bash sessions, as mentioned above.

The best way to see a snapshot of the listening/connected sockets and associated processes is with the `ss` command and its "`-p`" option (which was discussed above under testing).

Development Advice:
As with all of your programming assignments, you are strongly urged to develop this assignment *in stages*. One of the most frustrating aspects of client-server systems is that failure to follow protocols exactly (or program errors) can lead to the overall system hanging or connections being terminated unexpectedly. Thus, it is beneficial to *incrementally* build up the client and the server code, adding just a bit of new functionality to both the client and the server on each interation.

While the calls to set up the socket-related syscalls are somewhat complicated, remember that they are nearly always very similar for, say, all TCP client-server systems. This means that it makes a lot of sense to start with a simple working TCP client-server system, and modify it to have the required functionality. (This is not cheating because you will end up with only the lines that are the same in every TCP client-server left in your final code.)

Just such a simple client-server system code is available from the CS306 course textbook, *Beginning Linux Programming* (4th ed) by Matthew and Stones. The textbook's code is all available online. Chapter 15 is the Sockets chapter, so that is the code required. The text goes through several versions of both their client and server. The best ones for you to

start with are `client3.c` and `server4.c`. This is a very simple system: the client sends a char to the server, which increments it (next ASCII char), and sends it back to the client (which prints it out). `server4.c` is a parallel server, creating a new process to handle each client—as you are to do.

To get a start on this lab, you can download and compile both files. To get `server4.c` to compile without warnings, you may have to change line 16 to use `socklen_t` instead of `int`:

```
socklen_t server_len, client_len;
```

You probably also want to remove the `sleep()` call. These programs are designed to run on the same machine. It is best to open two terminal/shell windows, one to run each. Start the server running in one, e.g., `./server`. It should print the message "`server waiting`" and then sit there, as it is waiting for a client to connect. Now switch to the other terminal window and run the client: e.g., `./client`. It should print the message "`char from server = B`" and then exit. Run the client again and the same thing should happen. Switch back to the server window, and you should see new "`server waiting`" messages. Type `^-c` to terminate the server (or use the `kill` command).

Now that you have a very basic working TCP client and server, with the correct socket setup, start making incremental modifications to build up to Lab #1 functionality. For example, start implementing the initial rembash application protocol exhcnage. The server and client will have to send and verify several fixed messages. One way to simplify this is to define several string constants, e.g.:

```
char *server1 = "<rembash>\n";
```

or to ensure it is a constant:

```
const char * const server1 = "<rembash>\n";
```

Now you can send this message to the client by doing:

```
write(client_sockfd,server1,strlen(server1));
```

(note that we do not send null char's across the socket typically).

Work your way through getting the initial protocol exchange to work properly. You can simply have the server and client print out what they read from the other to help you with debugging (and don't forget `strace`). Simply the server reach the point of sending `<ok>\n` to the client, then have it send one more random message and close the connection. After you get the client so it successfully reads `<ok>\n` from the server, have the client go into a loop reading whatever the server sends and printing it out (to standard output). That loop should terminate when `read()` indicates file-end/error, and the client should then terminate.

At this point, you should have a client and server that successfully exchange the shared secret and other protocol elements, and the client prints out the final random message sent by the server (after `<ok>\n` was sent). Only after you are certain that your server and client and properly interact through the initial stages of the rembash protocol, should you continue!!

What is left to do? In the client, you must add code to `fork()` a new subprocess that will loop reading lines from the terminal and sending them to the server. In the server, you must add code to `fork()` a new subprocess that will be used to run Bash, redirect standard input, output, and error to the socket in it, then `exec` Bash.

The reading loop in the client is probably easier to implement, but I would suggest that you get the server Bash code working first. Even if you do not have the client read loop working, you can simply have the client send a simple Bash command to the server before going into the socket read-print loop you already have. A reasonable command to send is `ls` and you

will also have to send `exit` to get the Bash process to terminate. You can do this in one line as: `write(sockfd,"ls -l; exit\n",12);`

In the server, after sending `<ok>\n`, you now want to first `fork()` off a new subprocess. In that new subprocess, you need Bash to use the client socket for its standard input, output, and error. This is done by redirecting file descriptors 0, 1, 2 to the client socket using the `dup2()` syscall (read the man page carefully to get argument order). One you have completed the redirections in the new subprocess, all that remains is `exec`'ing Bash as shown above:

```
execlp("bash","bash","-i","--noediting",NULL)
```

If you have done this right and changed your client to send say the `ls` command before going into the socket read loop, you should see a directory listing of the directory where the server is running get printed out by the client before the client terminates.

Do not continue until this is happening correctly!!

At this point, all you have left to do it get the client to read commands typed by the user and send them to the server. This is to be done in a subprocess of the client, so before going into the socket read loop you already have, insert code to `fork()` off a new subprocess. In that subprocess, simply run a loop calling `fgets()` to read from the terminal and then sending each line (command) to the server running Bash. This process can keep running, until the parent detects file-end/error and uses `kill()` to terminate the child before it terminates.

You should now have a functioning client and server.

Oops, but not quite done! The original `client3.c` code from the text has the *loopback address* builtin as the server IP address. Your final client must take the IP address from the command line. Make that simple change and now you really are done!!

Adjust the shared secrets and ports to what is required before submitting!!!