CS407/CS591 – Adv. Linux Programming – Fall 2017

Lab #2
(Due: 9/25/17)

In the second lab, we are going to continue to develop our remote Bash client-server application, addressing key limitations of the Lab #1 version. The primary limitations we will address are the fact that Lab #1 did *not* have Bash interacting with a *terminal.* The `bash` program is a *terminal oriented program* as described in the text: it expects to interact with a *terminal* (TTY) on standard input, output, error. `bash` verifies that it is connected to TTY devices when it starts up, and complains if it finds it is not (as you should have seen with your Lab #1: "`bash: cannot set terminal process group (-1)`...."

Luckily, it is easy to solve this problem: use *pseudo terminals* (PTYs). Instead of connecting `bash` directly to the client-socket, we need to have the server create a PTY *master-slave pair* for each `bash` process, and connect `bash` to the PTY. This can be done by redirecting standard input, output, error to the PTY slave instead of the socket. Doing this will cause `bash` to believe that it is connected to a hardware terminal (TTY), since the PTY behaves just as a TTY would.

Program Submission:
You are to again produce both a server program and a client program, so this lab is to be implemented in *two separate files*, to be named: (1) `lab2-server.c` and (2) `lab2-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 #2 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!

Lab #2 Requirements:
Your server and client for Lab #2 must meet all the requirements from Lab #1, along with these *additional requirements:*

1. the server must connect each `bash` process to its own PTY, so `bash` functions properly
2. the server must not be able to be blocked from handling further clients by a malicious client that does not carry out the initial protocol
3. the various server processes must maintain only the open file descriptors that they need—i.e., they must not accidentally inherit unneeded/unwanted FDs
4. the client must function properly with the remote `bash` using a PTY, which means supporting the use of commands such as `top` and `vi` that require the "TTY" be able to be put into *noncanonical mode*
5. the client must properly cleanup for any possible *normal termination* sequence (it must not leave orphans or only a single process of the pair running)
6. remove any "artifical limit" you have on the size of command lines that you client accepts

Please read Chapter 64 of the course textbook (by Kerrisk) to learn the details of PTYs. You will also need to read up on TTYs as well. They are covered in Chapter 62 of the course text and Chapter 5 of the CS306 textbook by Matthew and Stones.

You are to continue to use *multiple processes* whenever concurrency is required. While using *threads* (Pthreads) would be slightly faster and less resource intensive, neither the (straightforward) use of processes or threads will allow one to build servers that can handle very large numbers of simultaneous clients (the *C10k problem*). We will start exploring better alternatives (chiefly *multiplexed I/O*) in Lab #3. Thus, there is little point in worrying about whether processes or threads are better for this lab. (You will get a chance to use Pthreads eventually for this server, when we implement a *thread pool*.)

You do not have to worry about your server being able to completely handle malicious/broken clients yet. That will require the use of *timers*, again, for Lab #3. However, you must make certain that a bad client cannot block *further* clients from being handled. This is easily accomplished by simply `fork()`'ing a subprocess at the start of handling a new client (right after `accept()` returns) instead of waiting until you start `bash`. so that each client is handled by a separate subprocess. (Some of you already did that in your Lab #1.)

Using PTYs in the Server:
Figure 64-3 on pg. 1376 of the textbook shows the layout for using a PTY with SSH, which is similar to our rembash application (with the addition of encryption, etc.). The *"ssh client"* corresponds to our rembash client process and the *"login shell"* corresponds to the `bash` (sub)process started by our server for the client. So what does the *"ssh server"* process correspond to? It corresponds to a process—process*es* really—that are part of our server, and handle the transfer of data back-and-forth between the PTY and the client-socket. So data movement within the server processes handling each client ends up looking like:
`bash`⇔PTY-slave⇔PTY-master⇔client-socket

Let us consider the ⇔ data transfers from left to right:

- Data transfer between `bash` and the PTY (slave) happens "automatically" once you *redirect* standard in/out/error to the PTY *slave* before exec'ing `bash`.
- Data transfer between the two components of the PTY is handled by the kernel (so automatic).
- Data transfer between the PTY (master) and the client socket must be handled by your server. You are to use two subprocesses to accomplish this.

Your server will have to now create *three subprocesses* for each client, rather than the single subprocess that you used in Lab #1. These three subprocesses will (eventually) do the following:

- run `bash` (just as in Lab #1)
- transfer data from PTY master⇒client-socket
- transfer data from client-socket⇒PTY master

Transferring data back-and-forth between the PTY master and client-socket is exactly analogous to what was done in the client in Lab #1 transferring data back-and-forth between stdin/TTY and the server-socket: TTY⇔server-socket. You are to again use *two concurrent* *processes* to do this: one to `read` from the PTY master and `write` what was read to the client-socket, and a second process to `read` from the client-socket and `write` to the PTY master.

The textbook discusses the two alternative PTY syscall APIs:

1. the UNIX 98 API: `posix_openpt()`, `unlockpt()`, `ptsname()`
2. the BSD API: `openpty()` or `forkpty()`

You must use the UNIX 98 API for this lab! The BSD API is *not* standardized in POSIX/SUS, so not guaranteed to be available or to always work the same across different Unixes. While students seem to like `forkpty()` because it does multiple steps for you, one of the goals in this course is to fully understand what is going on in our programs, so `forkpty()`'s relative ease of use is not actually a positive.

Using `posix_openpt()`, etc. is not really too difficult. Look at the example code in the textbook (Sections 64.3 and 84.4) to see how to use the UNIX 98 API PTY calls. **Please note, though, that you are not allowed to simply use the text's functions** (`ptyMasterOpen()` and `ptyFork()`). You are to use these functions only for guidance on how to use `posix_openpt()`, etc. in your own server code!

In addition to reading Chapter 64 of the textbook, be sure you read the man pages for `posix_openpt()`), `unlockpt()`, and `ptsname()`, and also have a look at "`man 7 pty`." Pay particular attention to *feature test macro* requirements and/or libraries that must be linked with!! (If you do not know about feature test macros, see the **Development Tools** lecture #6.)

Pay attention in the textbook to the fact that we must make the PTY the *controlling terminal* for the `bash` process. This requires we continue to use `setsid()`, just as in Lab #1. The difference in Lab #2 will be that each `bash` process ends up with a PTY as its controlling terminal, instead of lacking of controlling terminal. `bash` will now be happy, and those initial nasty messages will not appear.

Avoiding Inheriting File Descriptors:
Another requirement for this lab vs. Lab #1 is to pay attention to the file descriptors that are open in your various processes. Good (secure) programming style means that processes should have open only those file descriptors that they need access to. Here that means:

- listening socket: open only in main server process
- each client socket: open only in the two processes doing PTY-master⇔client-socket
- PTY master: open only in the two processes doing PTY-master⇔client-socket
- PTY slave: open only in the related `bash` process

Your server code needs to make certain that other file descriptors get closed as soon as possible in each process. E.g., in the main server loop, each new client socket FD returned by `accept()` needs to be `close()`'d immediately after a new subprocess is created to handle that client. Likewise, the listening FD that will be inherited by that new subprocess must be `close()`'d as the first step in that new client-handling subprocess.

Please note that the `show-server` script made available to you via the course D2L page with Lab #1 allows you to determine what FDs are open in your server processes! (Remember the script assumes you have compiled your server as an executable named `server`.)

Some students are confused about what `close()` does with an FD representing a socket, thinking it will close the network connection. This is not the case! Calling `close()` on a socket FD closes the network connection only if that is the last FD referring to the socket. So, as long as an FD for a socket remains open in some process, `close()` will not affect the connection. (The syscall `shutdown()` can be used to immediately terminate a network connection.)

Besides explicit calls to `close()`, there is another approach that can be used to close FDs that are not to be open in subprocesses: setting FDs *close-on-exec* flag. See the texbook and the `O_CLOEXEC` option in "`man 2 open`" for more information. While some calls (like `open()` or non-standard Linux versions of certain syscalls) allow this flag to be set when an FD is created, it can be set on an arbitrary FD by doing:

```
  fcntl(fd,F_SETFD,FD_CLOEXEC);
```

The Client Program and TTY Settings:
As noted earlier, in order to complete Lab #2, you will need to understand TTYs as well as PTYs. With Lab #1, only one TTY was involved: the TTY for client input/output. Once the PTY is setup, however, `bash` will be directly connected to a PTY (simulating a TTY) and the client will also continue to have an associated TTY. To get our remote shell session to appear as if it is running locally, we are going to have to adjust the client TTY's settings.

If one follows the development suggestions (below) and first gets the server converted to use a PTY for each `bash` process, it may appear that the client from Lab #1 is pretty much fine to also work with the new server. Just one little problem: each command you type gets echoed *twice*. Can you figure out why? Hint: two "TTYs" are involved.

TTYs normally run in what is known as *canonical mode*. In canonical mode, the terminal driver immediately echoes each character typed back to the TTY output, but does not send input on to a program reading from standard output until *return* (representing a *newline*) is typed. Think now about having two "TTYs" both in canonical mode. Does it make sense why you see each command you type appearing twice?

To eliminate the command output duplication problem, at least one of the "TTYs" must be put into what is known as *noncanonical mode*. In noncanonical mode, characters are not echoed by the terminal driver, and each character is passed to a reading program as soon as it is typed. Programs like `top` and `vi` set terminals into noncanonical mode so that they can read each keystroke as it is typed, and programs that require passwords (that should not be echoed out) also do temporarily.

So if we need to put one of the "TTYs" into noncanonical mode, the question you may ask yourself is, which one should it be? The client's TTY or the `bash` process' PTY? To help answer this question, let's first consider how a program like `top` would work with our setup. When you type the `top` command in the client, it doesn't get executed on the client. Rather, the command gets sent to the remote `bash` process, which executes it there. Thus, when `top` sets *its* "TTY" into noncanonical mode, it will be the PTY that gets set—not the client's TTY. So while we have sole control over the client's TTY, the server PTY can be controlled by commands. This sugests that if we are going to change one of the "TTYs" mode, it should be the client's TTY that we change.

Let's think further about what is necessary to get commands like `top`/`vi` to work properly. Suppose we type `top` in the client and let it run briefly, then type "`q`" to terminate it. While this should work, we know that with the client's TTY in canonical mode, the "`q`" won't get `read()` by our TTY⇒server-socket process until we hit *return*. That is not how `top` is supposed to work—i.e., not how it works if we run it in a local terminal. To fix this problem, we clearly need the client's TTY to let each character be `read()` as it is typed. Again, this means that we need the client's TTY to be running in noncanonical mode!

Please look at the course textbook and/or the CS306 textbook to see how to use `tcsetattr()` to *set the client's TTY into noncanonical mode.* Unfortunately, it isn't too pretty, but this is a fairly common thing to have to do, so very useful to have practice with.

Be aware that by setting the "client's TTY" into noncanonical mode, what you are actually going to be doing is setting the PTY in your GUI shell program (e.g., `konsole`) into noncanonical mode. The PTY does not get automagically set back into canonical mode just because your client terminates. Thus, it is up to your program to restore its TTY (actually a PTY) back to canonical mode (or really whatever mode it was set to when the client started) before terminating. This is what well behaved programs should do. Normally one does this by calling `tcgetattr()` before `tcsetattr()`, to get and save the TTY's settings, and then again calling `tcsetattr()` right before termination to restore settings. Ideally, your code must ensure the terminal settings get restored no matter how the program terminates (as long as termination is not abnormal as when killed by OS).

Note: the command `reset` can be used to fix the terminal settings if your program terminates abnormally or you have bugs in your code. Of course if echo'ing is disabled, you need to be good at typing r-e-s-e-t-return without any feedback!

Properly Terminating the Client:
Your Lab #1 client program was to end up with two concurrent processes tranferring data back and forth between stdin/TTY and the server-socket. The typical scenario for client shutdown would have the user type `exit`, causing `bash` to terminate, causing the server to terminate the network connection, leading to the process `read()`'ing from the server socket to get EOF/error return. Thus, you were suggested to have the parent process of the pair doing server-socket⇒TTY because that made it easy to terminate (`kill()`) and collect the child process (or just set `SIGCHLD` to be ignored and skip the collection).

The problem with this is that the standard sequence shutdown scenario is not the only one that can occur. E.g., if the user types `^-d` in the client, this will cause the child process `read()`'ing from stdin to get an EOF return, potentially terminating, and thus leaving the parent process (and network connection) continuing to run—even though the user can never enter another command.

We want to improve the Lab #2 client so that it always shuts down "cleanly" with all normal termination scenarios (we cannot do anything about *abnormal* termination that result from kernel signals due to hardware errors, etc.). The key issue is that if the child process terminates first, we need to get the following to happen: (1) the parent recognizes that termination, (2) the parent collects the child (if necessary), and (3) the parent terminates.

How can we get the parent process to recognize that the child process has terminated unexpectedly? A child's termination will cause a `SIGCHLD` signal to be sent to the parent (unless we have set `SIGCHLD` to be ignored in the parent). Getting the parent to interrupt what it is doing, collect the child, and terminate, will require that code be run when the `SIGCHLD` is received by the parent. This is exactly the purpose of a *signal handler*. Signal handlers are covered in the **Syscalls: Signals** lectures and the textbook.

To make certain that your client cleanly shuts down under any normal termination scenario will require that you setup a signal handler for `SIGCHLD` in the parent process. That handler will have to collect the child and then terminate the parent. Of course the parent will continue to have to deal with the standard shutdown as well (getting EOF/error return from `read()` and then terminating the child, etc.).

Cheating:
As with Lab #1, all students are expected to work completely independently on this assignment! The code that you submit is to have been written by you and you alone. Working in groups and/or submitting code written by someone else (including code from the Internet) will be considered **cheating!!**.

You are encouraged to examine the instructor's code, and make improvements to your own Lab #1 code. You are not, however, to cut-and-paste any of the instructor's solution code directly into your own programs. The proper way to use the solution code is to identify different approaches to doing things, and adapt those approaches to the particulars of your own programs and programming style.

Testing:
Obviously, you need to make certain that your final code does not cause `bash` to produce the warning messages it did in Lab #1, so you are sure the `bash`'s have a PTY as a controlling terminal.

Be sure to again test your server with *multiple simultaneous clients*, to be certain you have a true parallel server, and are properly creating subprocesses to handle each client.

The best way to determine whether only required file descriptors are open in your various server processes is by using the `show-server` script supplied with Lab #1. Make certain to run this with multiple simultaneous clients running.

Checking that you have TTYs setup properly will involve running commands in your client and making sure you see the commands echo'd only once. It will also require that you run programs such as `top` and `vi` that place TTYs into noncanonical mode, making certain they work properly—including getting them to respond immediately to individual characters you type in the client.

The easiest way to force the child (TTY⇒server-socket) process to terminate first is by typing `^-d` (control-d). You will need to check that this causes the parent process and thus the network connection and remote `bash` to terminate.

If you read Chapter 62 of the textbook in full, you may notice that there are terminal issues that we have not discussed, such as what happens when you *resize* your client's window. (E.g., try running `top` and resize the terminal window the client is running in.) **You will**

not be required to deal with window resizing, but you should at least be aware of it. See Section 62.9 in the course textbook for further info.

Development Advice:
Because both your server and client will become substantially more complex in Lab #2, you are strongly urged to consider decomposing their functionality into a set of functions. These functions can then be called from the `main`'s and `handle_client()`, making those elements cleaner and easier to understand. If you try to develop Lab #2 by putting all functionality into the `main`'s and `handle_client()`, you are going to end up with long, complex, and difficult to understand code. This will not make your programming task easy, so please consider *modularizing* your code for Lab #2!

As with all of your programming assignments, you are strongly urged to develop this assignment *in stages*. Since the client from Lab #1 will function (though somewhat incorrectly) with the completed Lab #2 server, it makes sense to get the server modified before you attempt making any changes to the Lab #1 client.

The first thing it makes sense to do is to make certain that your server cannot be blocked by clients that do not carry out the initial protocl exchange. This simply requires that you call `fork()` right after `accept()` returns with a new client. This way, all of `handle_client()` will be run in a separate process for each client, so clients cannot block one another from running.

In addition to carrying out the initial protocol exchange and then creating a subprocess in which to `exec bash`, `handle_client()` must now create a PTY in between these two actions. See the course textbook for example code. Since the PTY will get created in the subprocess running `handle_client()`, it is easiest if that process continues to interact with the PTY master, to transfer data back-and-forth with the client-socket. Before it begins to deal with the back-and-forth data transfer, however, the process must `fork()` off a new subprocess in which `bash` will be `exec`'d.

Prior to that new subprocess making the `exec` call, it must set up its session and redirect standard in/out/error to the *PTY slave*. Because `bash` will now have the PTY as its controlling terminal, you should start it in interactive mode simply by doing:

```
  execlp("bash","bash",NULL)
```

As noted earlier, you are to use two processes (per client) to carry out the back-and-forth data transfer between the PTY master and client-socket. This way, your code will be very similar to what you have in the client. (Having similar code suggests why breaking this out into a function is worthwhile.) So, after `fork()`'ing off the subprocess for `bash`, the process handling the client will have to `fork()` off another process to do the other half of the PTY master⇔client-socket data transfer.

Pay attention to what will normally happen to end a client: the user types `exit` in the client, this gets sent to `bash`, which terminates. You must make certain that both of the PTY master⇔client-socket transfer processes notice this and also terminate appropriately. (Hint: use `strace` to see what happens with its PTY when `bash` terminates.) Your server code is also to handle "unexpected" situations, such as the client terminating without sending `exit`, the network connection being lost, etc.

Once your new server is "working" with your Lab #1 client, it is time to modify the client.

Before you do that, though, you might want to modularize the client code, as this will ultimately make your job easier.

The main changes to the client involve having it set its TTY into noncanonical mode, and then restoring terminal settings on termination. You must also ensure clean termination of the client's processes for any normal termination sequence (such as user typing `^-d`). Finally, you should remove any limitation on the length of command lines that you had in Lab #1.

Questions to Consider:
While working on this lab, here are a few questions to think about:

- Having `handle_client()` immediately start running in a separate process may help server performance when dealing with clients on poor network connections (so slow responding due to need to repeat packets), since main server loop gets right back to `accept()`. It does nothing to deal with malicious/broken clients hogging server resources, however. What is needed to keep malicious/broken clients from being able to carry out DoS attacks on your server?
- We now have three (sub)processes being created for each client. This will drastically limit how many clients can be simultaneously be handled, because each process takes up kernel space (PCB, etc.) and has its own separate address space. Even if we have enough RAM for PCBs, eventually we would get to point where most processes will be *swapped out* when they able to run, leading to *trashing* and poor performance. We cannot get away from having one (sub)process for each `bash` instance, but we could have used two threads to do PTY$\Leftrightarrow$socket data transfers instead. Would that solve our problems?
- When writing to a socket (whether using `write()`, `send()`, etc.), the write can be *partial* (less data is written than was requested). This turns out to be a rather major issue for many servers. Think about what is required in this lab to deal with partial writes. Hint: your `write()/send()` calls need to be in a *loop*. While looping on socket writes can potentially cause blocking issues in many servers, because we are using separate processes for each client in this lab, it is not an issue.
- Having client TTY in noncanonical mode can significantly affect network traffic. How? Hint: look into *Nagle's algorithm* and the `TCP_NODELAY` and `TCP_QUICKACK` socket options.

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