CS407/CS591 – Adv. Linux Programming – Fall 2017

Lab #3
(Due: 10/13/17)

With Lab #2 completed, you have a (largely) correctly functioning client and server for the remote Bash session client-server application. In Lab #3, we are going to modify the server so that it can handle many more simultaneous clients (no changes will be required for the client). We are also going to start addressing the risks from malicious/broken clients.

Program Submission:
You are to again submit both a server program and a client program, in *two separate files*, to be named: (1) `lab3-server.c` and (2) `lab3-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. `lab3-client.c` can be identical to your `lab2-client.c`, or can be improved if that client had some bugs that you want to correct. Submit your two *C source files* using the Lab #3 Dropbox on the course D2L website. Do not submit any additional files!

As with all labs in this class, 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!!**

Lab #3 Requirements:
Your Lab #3 must meet these additional requirements as compared with Lab #2:

1. each client is to require only a single additional subprocess (for running `bash`)
2. data transfers (PTY`<-->`socket) for *all clients* are to be done by a *single Pthread*
3. that single Pthread is to use a *multiplexed I/O* approach with the `epoll` API
4. server must not be vulnerable to *denial of service attacks* by malicious/broken clients

Data Transfer via Multiplexed I/O:
For Lab #2, you were to use a *pair of processes* with each client to carry out the PTY`<-->`socket data transfer for the client. Since you also had to have another process in which to run `bash`, that meant you were using *three processes for each client*. This will severely limit how many simultaneous clients your server will be able to handle. (Of course the one subprocess per client for running separate instances of `bash` is required.)

The *first major improvement* you are to make for Lab #3 is to convert your server to handle the data transfers among large numbers of file descriptors the way modern high-capacity Linux-based servers do: with *multiplexed I/O* using the `epoll` API. Multiplexed I/O means using a single thread/process that loops, monitoring client file descriptors for `read()/write()` readiness, and then carrying out the data transfers for those that are currently ready. This approach uses many fewer system resources than would be required using separate processes (or separate Pthreads) for each client's data transfers.

The one potential drawback of multiplexed I/O as we will do it in Lab #3 is that we will be using a *single thread* for all data transfers. This *might* end up being slower handling data transfers for many clients than it would be if we used separate processes or Pthreads that could run in parallel on a multi-core machine. Whether a single data transfer thread actually ends up being slower than multiple processes/Pthreads depends on many factors (e.g., the cost of context switches with processes/Pthreads may eliminate potential gains from parallelism). In any case, in Labs #4 and #5 we will see how to remove the single I/O thread limitation through the use of *thread pools*.

You are to use the `epoll` API for multiplexed I/O because—as our textbook shows—this *Linux-specific syscall* is much faster than the original multiplexed I/O syscalls: `select()` and `poll()`. The only reason to use `select()`/`poll()` any longer is if one needs to produce portable code (which we do not). In that case, though, it would be better to use `libevent` (or similar `libev` or `libuv`), which would provide both portability and speed.

With the addition of an `epoll` loop, your server now requires *two permanent threads of execution:*

1. the main server loop that `accept()`'s new client connections and starts them being "handled";
2. the `epoll_wait()` loop that waits for readiness to `read()`/`write()` on client FDs and then transfers the data to the corresponding client FDs.

Because information must be shared between these two threads of execution, the `epoll` loop is to be implemented as a *Pthread* within the same process as the main server loop.

See the **epoll for Data Transfer** section below for further info.

Handling the Initial Protocol Exchanges:

While your server must handle all data transfer activity for all clients within the single "`epoll` Pthread," what about the initial protocol exchange for new clients? There are two main ways that this can be done for Lab #3. In both cases, your program must meet the requirement of preventing DoS attacks against the server by "clients" that connect but then do nothing (tying up server resources indefinitely).

One possible approach for running the initial protocol exchange code is to use a *separate, temporary Pthread for each new client.* Creating a new Pthread is quite fast and takes few resources, so should not be a problem even with large numbers of clients. Such a thread would be respnsible for sending the protocol ID, and then getting and validating the client's shared secret. Once that has been done, the thread would create the required PTY and `bash` process for the new client, and add the client's FDs to the `epoll` unit. It would then terminate (the Pthread only would terminate).

The problem with this approach in the long term is that it will not integrate well with our eventual use of a *thread pool.* One of the ideas with a thread pool is to match the worker threads to the number of machine cores. If other Pthreads are being created and running, those threads will effectively compete with the worker threads for computation resources. On the other hand, it is unlikely with our particular client-server application that new clients are going to be frequently starting, so should not be major issue.

Nonetheless, for the ultimate version of this server (Lab #5), you are going to be required to have *all operations* (including `accept()`'s) *carried out by the thread pool worker threads!*

This is going to require that you be able to represent the *state* of each client, so that when `epoll` says an FD is ready for I/O, it is clear what should be done with it. For example, we have might have just two states: `new` and `established`. If client socket FD is ready for I/O and state is `new`, the operation to carry out is the initial protocol exchange. If client socket FD is ready and state is `established`, the operation to carry out is to transfer data to corresponding PTY.

It is up to you which approach you use for Lab #3. Creating temporary Pthreads for new clients is simpler to implement, but since your Lab #5 must ultimately use a state-based model of clients, you could get a jump on Lab #5 by starting to use such an approach here.

Avoiding DoS Attacks:
A key new requirement for Lab #3 is to make certain that malicious/broken clients cannot tie up server resources indefinitely, leading to a Denial of Service attack. The specs above already state that you cannot have the initial protocol exchange handled as part of the main server (`accept()`) loop thread. Simply having the exchange be handled by a separate Pthread is not enough to prevent problems, though. An attacker could make many client connections (using a *botnet*) and ultimately exhaust the server's ability to create new Pthreads (preventing valid clients from connecting).

Properly dealing with this requires that your server allow a new client only a short amount of time to prove it is a valid client by supplying the correct shared secret. This can only be done (in an efficient manner) by using the kernel's *timer* mechanisms. Basically, when the initial protocol exchange begins, a timer is set, and if that timer *expires* before the client has sent its shared secret, the server will close the client connection. So the timer limits the length of time that the server will wait for the client during the initial protocol exchange. See the **Timers** section below for further information.

Please note that once a client has had its shared secret validated so it is known to be a valid client, the question of what should be done if it is inactive for a period of time is very different. One might want to eventually disconnect inactive clients, or one may not. Even if one does want to disconnect them, the time scale for disconnection should be vastly longer than for the initial exchange. (Inactive network connections may also be closed by various entities—see "`man 7 tcp`" for TCP's "keep-alive" options.)

Pthreads:
You will make significant use of Pthreads in this assignment. If you are not familiar with them, see the following resources:

- instructor's Threads slides (available through the course webpage)
- textbook chapter 29 (Threads: Introduction)
- CS306 textbook chapters on threads (e.g., Matthew-Stones chapter 12)

Remember that when you start a Pthread, you need to pass it the function to be run. If you need to pass data to that function, you need to figure out how to do that in a manner that is guaranteed to always(!) work. Unfortunately this may not be obvious or simple.

For example, let's suppose that you need to pass a file descriptor to the thread function (e.g., the new client's connection FD). C will happily allow you to do something like:

```
  pthread_create(&threadid,NULL,handle_client,(void*)connect_fd);
```

and then in the thread function `handle_client(void*arg)` "decode" the argument like:

```
int client_fd =  (int)arg;
```

While C will always allow these casts, it is by no means guaranteed that such code will work properly, since C does not guarantee that you can cast an `int` to a pointer and back, and properly recover the `int`. Here is what the C99 standard has to say on these conversions: *"An integer may be converted to any pointer type. The result is implementation-defined, might not be properly aligned, and might not point to an entity of the referenced type. Any pointer type may be converted to an integer type; the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type."*

The phrase "implementation-defined" means that different implementations are allowed to do whatever they want (i.e., find convenient) to handle casts of pointers`<-->`integers. While it is the case that with appropriate size integer types, many C implementations will allow you to do integer`-->void*-->`integer and recover the integer, there are no guarantees this will always work. Furthermore, you will generally need to use an integer type that is the same size (in bytes) as pointers to do this for any integer value.

The errors and security issues that arise from casting integer and pointer types to one another are many. For more info, google "`CERT C Coding Standard`" and look at Rule 04. Stackoverflow has some discussions about why pointers may not be simple integers on all CPU architectures

C99 added the types `intptr_t` and `uintptr_t` which might at first seem to solve the above problem. However, these types guarantee that you can recover a pointer that has been cast into these integer types—they do *not* guarantee that you can recover these integer types after casting to a pointer. I.e., this is fine:

```
char *str1;
...
intptr_t num = (intptr_t)str1;
...
char *str2 = (char*)(void*)num;
```

This is *not:*

```
int fd1;
...
void *ptr = (void*)(intptr_t)fd1;
...
int fd2 = (int)(intptr_t)ptr;
```

Another issue discussed in the Threads slides is the potential for a *race condition* where the memory pointed to by the passed pointer gets changed before a new thread can make use of what was stored there.

The Threads slides show a correct way to deal with both of these problems: pass a pointer to `malloc`'d memory that holds the data you want to pass.

For example:

```
int connect_fd;
...
int *fdptr = malloc(sizeof(int));
*fdptr = connect_fd;
pthread_create(&threadid,NULL,handle_client,fdptr);
```

and then in the thread function `handle_client(void*arg)` "decode" the argument like:

```
int client_fd =  *(int*)arg;
free(arg);
```

Of course now we have to do a `malloc()` and `free()` with each `pthread_create()`, but it is better to bear that cost than to have potentially incorrect code. (How we could eliminate the memory allocation costs?)

Important notes in the `pthread_create()` man page:

- `Compile and link with -pthread.`

epoll for Data Transfer:
Please read the textbook section on `epoll` in Chapter 63. Remember that the `epoll` API consists of three syscalls: `epoll_create()`, `epoll_ctl()`, and `epoll_wait()`. There is also an `epoll` man page ("`man 7 epoll`").

Once created with `epoll_create()`, an epoll unit is accessed via a *file descriptor* handle. The epoll unit must be accessible in the epoll data transfer loop thread and in the threads that setup handling of each new client (so they can add the new clients FDs to the epoll unit). Sharing such data among Pthreads is typically done with *global variables*.

Global variables are variables that are declared outside of `main` or any function. They are typically defined at the top of a file, right before `main` and the functions. Storage for global variables is in the *initialized data segment* or *uninitialized data segment* of the executable created by the compiler—i.e., the same memory as for `static` variables. Thus, global variables can be initialized only to constants in their declarations. This means that the `epoll_create()` call to truly initialize the epoll FD global must occur inside of `main` or another function—not in the global's declaration.

Another use for globals is probably going to be for mapping between the PTY and client socket FDs. Each client effectively has a *pair* of FDs that will be required in the data transfer loop: its PTY master FD and its client socket FD. The `epoll_wait()` loop is going to give a list of FDs that are ready to `read()`/`write()` from/to, PTY master FDs and/or client socket FDs. To be able to do the data transfer, you will need to know the corresponding FD to `write()`/`read()` the read data to/from. This mapping must be setup by the thread that initially handles the client and creates the PTY, and it must be available to the epoll loop. So, again, a global variable (or multiple GVs).

You must use a data structure that allows you to take a PTY FD or socket FD returned by `epoll_wait()` (you won't know which), and find the corresponding FD for that client. This sounds like a job for a hash table perhaps, but you can probably get away with something simpler if you consider the characteristics of FDs. New FDs are always assigned the lowest possible unused integer values. 0, 1, 2 are already used, plus the listening socket and the epoll unit. So the client FDs should start at 5 and go up: 5 & 6, 7 & 8, etc., with these pairs being reused as clients close. One possible way to store mappings then is with a simple (single dimension) array: use a FD as an array index, and what is stored at that location is its pair FD. Initialize this array to be the size `MAX_NUM_CLIENTS * 2 + 5`, and you would be able to handle `MAX_NUM_CLIENTS` clients. Note that even if this was set to 100,000, we are talking about needing just about 800 kB of memory—not much for a modern computer (especially a server).

As part of writing the epoll data transfer loop, you need to make sure that you are able to handle clients/connections that suddenly close (i.e., with*out* the client sending `exit` to `bash`). You can simulate this by terminating a client program with `^-c`. What happens if you are waiting to be able to `read()` on the socket FD? Read the epoll man pages and do some testing to make certain your server doesn't continue to maintain a client that is no longer active.

Timers:
There are many times when a program needs to be able to do something for a limited period of time. For example, a standard blocking `read()` can cause a program to wait *indefinitely* for data to become available, but waiting longer than some finite amount of time may not be reasonable. Another common example is when a program needs to be able to handle multiple events, and it cannot get tied up handling a single event for too long or it won't be responsive enough.

A program can determine that some period of time has elapsed through the use of *timers*. Timers are covered in chapter 23 of the course textbook. Timers allow a program to setup a timer that counts down in real time (or other units) starting from some value, until the timer reaches zero. When that happens—the timer is said to *expire*. This causes a *signal* to be sent to the program to notify it of the timer expiration. The signal can interrupt blocked syscalls (e.g., `read()`) and/or may invoke a *handler* function to deal with the event. The standard timer signal is `SIGALRM`.

The simplest timer interface is `alarm()`. It takes a single argument: the number of (real time) seconds before a `SIGALRM` signal is generated. Calling it with an argument of 0 (zero) cancels a pending `alarm()` timer. Note that the default disposition for `SIGALRM` is to terminate the process, so a signal handler function will have to be registered to avoid termination. Sometimes such handler functions will do nothing (simply allow the signal to interrupt `read()` or another syscall without terminating the process).

Section 23.3 in the course textbook explains and has example code of how to use `alarm()` to keep from blocking in `read()` indefinitely. The book's example `alarm()` call is rather complicated, but one can set a timer for one minute by simply doing: `alarm(60)`. Key elements of the example are (1) the `SIGALRM` handler setup, (2) the timer setup with `alarm()`, (3) cancelling the timer if the `read()` happens before the timer expires, and checking `errno` to see if it is `EINTR` in case `read()` returns -1 (to check if `read()` was simply interrupted rather than errored).

Dealing with malicious/broken clients or broken network connections as part of the initial client setup will require a timer be set prior to the server sending the initial protocol ID, and cancelled if the shared secret is not received from the client in a reasonable amount of time. Should the timer expire and a signal be sent, we will want the server to quit engaging in the protocol change with the client, and close the connection. Note that you need to deal only with clients who fail to carry out the initial protocol exchange (i.e., fail to send the shared secret); we will leave consideration of how to deal with clients that already have a `bash` process (but send no commands or fail to read sent data) for Lab #5.

Unfortunately, the use of timers with Pthreads is somewhat complicated due to how signals interact with Pthreads. With `alarm()`, the `SIGALRM` signal will get delivered to the entire process, which means one randomly selected Pthread (assuming no special setup). This is not going to be desirable if we have multiple Pthreads in the server, only one of which is handling the new client. While there are some methods to get around this problem, they are not simple. Luckily, the newer *POSIX timers* (Section 23.6 in the textbook) have a Linux-specific option that will do just what we need: deliver the timer signal to only the new client Pthread. Using these timers requires understanding the syscalls `timer_create()` and `timer_settime()`. Pay particular attention to the `SIGEV_THREAD_ID` option.

If the timer expires and interrupts `read()`, the client handling thread needs to make certain to `close()` the socket FD before it terminates (both to conserve FD resources in the server and to close the client's connection). There are a couple of ways to do this when using POSIX timers: (1) check the error from `read()` to see if it was interrupted so done with client (similar to what the book shows doing with `alarm()`), or (2) use the POSIX timer to pass the client's FD to the handler to close (this requires using a "`siginfo_t`" handler function).

Another difference between `alarm()` and the POSIX timers is that you can create multiple timers with `timer_create()`. This means that it is critical to call `timer_delete()` to delete each client's timer (or figure out how to reuse them) to avoid the server having a "timer leak." Note that deleting an armed timer also disarms it. Make certain you delete all timers: both those that expire (and interrupt `read()`) and those that did not expire (so need to be disarmed after `read()`).

Important notes in the `timer_create()` man page:

- `Link with -lrt.`
- `Feature Test Macro Requirements: _POSIX_C_SOURCE >= 199309L`

Pthreads and _CLOEXEC file descriptor options

As we know, the key difference between using multiple threads vs. multiple processes is in the inherent sharing vs. not sharing of data. While using threads in Lab #3 makes it easier to share data (FDs) between threads setting up a new client and the thread running `epoll_wait()`, the fact that all FDs are shared among all the server threads also causes FD inheritance problems.

We discussed this issue in Lab #2, and you were to correct it, but problems may arise again in Lab #3 due to the use of Pthreads. You should be able to see the problem once you have your basic server working with threads. Start the server, connect multiple clients (three is better than two), then run the `show-server` script that was provided. Pay attention to the

FDs that are open in the `bash` subprocesses. Do you see a problem? What you will probably find is that these processes not only have the required FDs for the PTY slave open (0, 1, 2), they also has open FDs for the server's listening socket, the epoll unit, one or more of the PTY masters, and one or more of the client sockets. Not only are none of these FDs required by the `bash` processes, they really should *not* be open in those processes.

Why? For one thing, it gives each client the ability to manipulate server FDs that the clients should not be able to manipulate. E.g., by redirecting to the appropriate FD, one client could send bogus commands or output to another. So we have a security/protection problem. To some extent, this not is a major issue for our server since all clients run as the same user, so could manipulate the same files, etc. However, we are using a simplified (single user) protocol to avoid having to deal with passwords. A real-world version of a remote shell app (e.g., Telnet) would have each client being required to login, and then run bash as that user. For that situation, it should be clear to you that having the server's and other clients' (i.e., other users) FDs be accessible from the `bash` process would be a serious security problem!

It turns out that the extra FDs in the bash processes cause significant problems even for our simplified rembash server. Once you have created two or three clients, try exiting from them in the same order you created them (i.e., `exit` from client 1 first). What happens? Does the first client appear to hang when you try to exit? Look back at the output from `show-server` when all clients were running. Can you see what is causing the "hang?" Hint: how many processes have an open FD for the first client socket? Run `show-server` again to see why Client 1 still considers the connection open.

The source of the `bash` process' extra FDs problem is that we are `fork()`'ing off the new process for `bash` from a thread now, and that thread has all the server's open FDs of course. So, the new process by default inherits *all* the open server FDs—not just those for the one client it is concerned with.

What can we do about this? One potential solution that might come to mind is to `close()` every unwanted FD after we call `fork()` but before we call `exec`. We could keep a global list of all the currently used FDs and iterate through it, calling `close()`. In fact, the kernel provides the necessary info in the directory: `/proc/self/fd`. This seems like a lot of bother, though: get directory listing from `/proc`, iterate through filenames 0, 1, ..., convert the names to integers, and call `close()` on each. Ugh!

Turns out that there is a mechanism that can be used to provide exactly the functionality that we want: *close-on-exec*. Needing automatically inherited FDs from `fork()` to get closed if `exec` is called (so a different program starts running) is quite common. Various syscalls have options with names of the form `*_CLOEXEC` that allow you to end up with inherited FDs that get automatically closed when an `exec` call is made. This is precisely what we want for nearly every server FD. The basic call to set any FD into close-on-exec mode is:

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

If you search the textbook and various man pages, you will see that a number of syscalls that create FDs, provide the ability to specify the close-on-exec option (eliminating the need for a separate `fcntl()` syscall). All of these options are of the form: `*_CLOEXEC`. Some of the calls that can make use of the options are Linux-specific extensions of standard syscalls. They are fine to use, since we are already using the Linux-specific epoll.

NonBlocking Mode:
If you carefully read the man pages for `epoll` and related calls, you will see that there can be "spurious" activations reported for FDs. A similar issue is that client connections could close between `epoll` reporting an FD ready and the time your code runs to handle that FD. This means that even when `epoll`, etc. reports an FD as ready for the operation, it is still possible for the operation to block! With a single loop carrying out operations, this can potentially stall the server. (Even with a thread pool, worker threads can be stalled.)

The correct solution to this problem is to use *nonblocking mode* for I/O (and even other FDs). When an FD is set into nonblocking mode, `read()/write()` calls always return "immediately" (note that *regular file* I/O calls can still block very briefly—but the key is that they cannot block *indefinitely*).

To learn more about nonblocking I/O, look at the man pages (particularly ERRORS sections) for `read()`, `write()`, `send()`, `recv()`, `open()`, `accept()`, and `fcntl()`. The textbook covers nonblocking mode in several places, including Sections 5.9 and 44.9, and Chapter 63.

The key idea is that instead of blocking, calls like `read()/write()` return -1 (error), and set `errno` to a special value: `EAGAIN` or `EWOULDBLOCK`. Rather than indicating true errors—i.e., errors that should terminate execution or similar—these errors simply indicate that one needs to try the operation again later. In the case of the `epoll` data transfer loop, receiving these errors when trying to `read()/write()` simply means that the FD was not ready (even though `epoll` indicated it was), so the operation should be postponed, by having `epoll` continue to monitor the FD, so the operation can be tried again later.

Use of nonblocking mode for the listening socket, client sockets, and PTY master FDs will be required for Lab #5, but is optional for this lab. If you are interested and have time, however, you may want to get a jump on the Lab #5 requirements. Note that in addition to using `fcntl()` to set nonblocking mode, you may want to consider the `*_NONBLOCK` options for `socket()` and `accept4()`.

Client State and Client Objects:
Ultimately (Lab #5), you are going to have to do all operations via thread pool workers so will require state info about clients. You will also have to handle partial `write()`'s (`write()` calls that do not write all requested bytes), so will have to keep track of that (and store unwritten data). This means that you will have to maintain several pieces of info about each client (beyond FD pair). In other words, you will need *objects* (`struct`'s) for each client, which can maintain the required information. Doing this is *optional* for Lab #3, but you may want to at least keep it in mind while you are designing your implementation. To also keep in mind: allocating/deallocating such objects should be fast and efficient, so using `malloc()`, etc. is not ideal!

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