CS 407/591 – Adv. Linux Programming – Fall 2017

Lab #5
(Due: 11/15/17)

Lab #5 will complete the rembash protocol server, leaving you with a largely complete server that utilizes the techniques used in modern high load servers (multiplexed I/O with epoll, thread pools for concurrency on multicore systems, etc.). You are to start with your server from Lab #3 and your thread pool implementation from Lab #4, and modify the Lab #3 server to make use of the thread pool to carry out all the operations required for the clients (concurrently). Some changes/additions will be *optional*.

The Lab #3 server had these key characteristics:

- A single Pthread was used to do data transfers for all clients, using multiplexed I/O via *epoll*.
- The server was not vulnerable to DoS attacks because:
  - each new client was handled by a separate temporary Pthread;
  - *timers* were used to deal with lack of response from clients;
- Only the single (sub)process for `bash` was required for each client.

Lab #5 is required to meet the following requirements:

- Thread pool must be used:
  - *all operations* for clients must be carried out by the thread pool worker threads;
  - this includes `accept`'ing new clients, new client protocol exchange and initialization, and data transfer;
- The server is to be running *at most one* non-thread pool Pthread after initial startup:
  - thread is to run `epoll_wait()` loop to identify FDs to become thread pool tasks;
  - this thread then calls `tpool_add_task()` to get the necessary work done;
  - no additional temporary Pthreads are to be used;
- The server must use *nonblocking I/O* for data transfers:
  - allowing blocking operations with thread pools can lead to poor performance;
  - once *nonblocking mode* is used for I/O, there is the potential for *partial writes* (`write()` returns a byte count less than the request);
  - simply sitting in a `while` loop repeatedly calling `write()` is no better than using blocking I/O, so is not an option;
  - instead, the server must be able to store unwritten data and try `write()`'ing it again, but only when writing to the target FD is again possible;
  - no guarantee unwritten data will get written in *single* subsequent `write()`;
- Broken/malicious clients must still be prevented from causing DoS attacks:
  - unauthenticated clients must be prevented from indefinitely tying up resources;
  - this again requires using *timers*;
  - now, however, timers must be integrated into the `epoll_wait()` loop;

Using the Thread Pool:
Remember that the external interface to your thread pool consists of just two functions:

- `int tpool_init(void (*process_task)(int))`
- `int tpool_add_task(int newtask)`

The prototypes for these functions are to be provided to your server code by `#include`'ing the header file `tpool.h`. You should be able to compile/run your code by using your thread pool libraries. However, if you need to turn on debugging code within your thread pool, you will need to compile with your `tpool.c` file directly.

The most important aspect of using your thread pool with your server is the `process_task()` function. The thread pool design you were to use was to be very simple: tasks are represented by `int` file descriptors that are then given to a single function to perform all tasks.

This means that the `process_task()` function must be able to take a FD that could be for (1) the listening socket, (2) a client socket, or (3) a PTY master, and figure out what to do with it. I.e., the `process_task()` must take an FD and *dispatch* on appropriate additional information to figure out what it should do with the FD. There are only a small, *fixed* set of possible actions:

1. `accept()` a new client;
2. carry out protocol exchange and initialize client;
3. transfer data to corresponding FD (of client-socket–PTY-master pair);
4. continue partial `write()` during previous transfer.

Note that an alternative design for the thread pool could have allowed for tasks to include both an FD and a *function* that "knew" what should be done when that FD was ready. While this design makes sense when functions can be "anything," the fact that we really have only four possible actions means that it shouldn't be too difficult to figure out what to do with an FD. Furthermore, we will be figuring out what to do with an FD when it is ready, as opposed to when the FD was placed onto the task queue. E.g., consider that both the socket and PTY FDs for each client will generally be placed simultaneously onto the queue, but one will typically be processed first. Suppose the run of that FD causes the client to terminate. With our design, `process_task()` will be able to know this if/when the second FD should come to be processed. That would not be the case if task functions had been queued. In highly concurrent programs, there can definitely be an advantage to figuring out what to do when you are going to do it rather than relying on a decision that was made in a potentially different state.

The question is, though, *how* can `process_task()` figure out how to *dispatch* to the proper (sub)function to handle a ready FD?

Client Objects:
In Lab #3, the only information about clients that had to be maintained (outside of code) was the mapping between a client's socket FD and its PTY-master FD. As we discussed, this can simply be done by maintaining an fixed-size array as large as all the FDs your server is able to use. It could also be done with a *hash table* if one is worried about saving space,

though that will end up being slower if there are collisions and if initial table is too small and ends up having to be expanded. A classic time-space tradeoff—and we are focusing on time (making server as fast as possible to handle very high number of clients).

In this lab, you will still need to maintain socket-PTY FD mapping information, but there is additional information about clients that will also be needed. Most important is an indication of the *"state"* of each client. This will be a key item in `process_task()` determining how to dispatch to the appropriate functionality to process a ready FD. Another key item that will need to be able to be associated with a client is a buffer containing unwritten data for the socket.

So how do we maintain a diverse set of data as a unit? A struct of course. I.e., we will need to maintain an *object* (struct instance) for each client. These objects will need to maintain at least: (1) client state, (2) socket and PTY FDs, and (3) a (pointer to) a buffer containing unwritten socket data. (We should not need to handle partial writes to PTY if we write correct size!)

What sort of states might we want? We need to think about what `process_task()` will have to know to figure out what to do with a ready FD. At minimum you probably want something like:

- *new:* so carry out protocol exchange
- *established:* so do data transfer
- *unwritten:* continue unwritten data transfer
- *terminated:* do nothing

What is the best way to maintain something like a state in C? Clearly you can just use an `int`, but C provides something that makes your code a bit more clear: *enums* (enumerations). Enums are not convered in my C slides, so a bit of googling will be required, but please used them where appropriate.

What about creating and maintaining the client objects? Obviously one could `malloc()` up (heap) memory every time a new client is accepted and `free()` it when the client terminates. Remember, though, that we are aiming for a design that can handle very large numbers of clients, and `malloc()`/`free()` are relative slow operations. Operating systems have to deal with similar issues for processes (and other objects). One technique that they use to avoid `malloc()`/`free()` is what is commonly referred to as *slab allocation*. At its most basic, this is a simple idea: preallocate memory for the maximum number of objects you will need to store, and then include a very simple/fast version of `malloc()` and `free()` for this memory. While you are allowed to use `malloc()`/`free()` for client objects, you are strongly urged to use slab allocation (it will count for more points). Hint: think about how you can rapidly find a free object in this memory, and how you could rapidly return an object once the client terminates.

Of course `process_task()` will need to be able to locate the relevant client object from its task FD. How do you think you could do that sort of mapping quickly? (Think about Lab #3.)

Restructuring Your Server:
The fact that you must handle the protocol steps for new clients in the thread pool workers will cause you to have to restructure your code from Lab #3 (unless you implemented some

of the options). First, you will have to include the listening socket FD in the epoll unit, and when it is returned as readable (meaning there is a new client available) you must add the listening socket's FD as a thread pool task. Then, your `process_task()` function run in the workers must be able to identify the FD as the listening socket's FD and proceed to `accept()` the new client and start the protocol exchange.

Another restructuring that you may want to consider as compared to your Lab #3 server is to break up the protocol exchange into two stages: (1) initial server write of protocol ID; (2) reading, checking, and responding to shared secret. This will require that you can keep track of what stage the interactions with a client are in: new vs. protocol exchange vs. data transfer. The advantage of doing this is that you don't tie up a thread pool worker thread as long handling the protocol exchange: in one cycle a worker can `accept()` a client and send the protocol ID, and then in another cycle the shared secret can be read and checked, etc. Note that restructuring the server's interactions with clients into two stages will affect how the server might deal with malicious/broken clients.

Nonblocking Mode:
The default mode for I/O operations like `read()`/`write()` is *blocking mode:* a call *blocks* (process is suspended) until the call can be completed. For sockets and some other devices, the FD can be put into *nonblocking mode:* a call always returns immedaiately, but may indicates an error if operation cannot immediately be complete (i.e., it would block if FD was in blocking mode). When this situation causes an error to be indicated, `errno` will be set to `EAGAIN` (the POSIX/SUS also allows `EWOULDBLOCK`, but in Linux this has the same value as `EAGAIN`). By checking `errno` then, one can determine if `read()`/`write()`/etc. threw an error due to an actual error, or simply because the operation would have blocked.

Nonblocking mode is very useful when we do not want to have a process/thread block because other work could be getting done instead of sitting idle (while blocked). One common reason for using a multithreaded approach for some program is so that useful work can continue to get done by the program overall, even if one thread blocks. Since a thread pool has multiple workers, having one or two workers block temporarily may not have a major impact on overall performance (NGINX added thread pools just to allow this), but in our server it is quite easy to avoid any blocking due to I/O by simply using nonblocking mode.

What FDs should be set to nonblocking? First, recall the following from the Lab #3 discussion: 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_wait()` reporting an FD as ready and the time your code runs to handle that FD. This means that even when `epoll_wait()` reports an FD as ready for some I/O operation, it is still possible for that operation to *block!*

To avoid this possibility, *all FDs being monitored by epoll* need to be set to *nonblocking mode.* This means the listening socket FD, all client-socket FDs, and all PTY-master FDs.

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.

An FD may be able to be set into nonblocking mode in several ways. The basic method for an arbitrary FD is to call `fcntl()` (see the man page). However, nonblocking mode can also

be set when using some syscalls that create FDs: e.g., `open()`, `socket()`, and `accept4()` (latter is Linux specific). See these syscalls man pages to read about the `*_NONBLOCK` options.

Once an FD is set into nonblocking mode, I/O operations like `read()`/`write()` must account for the possibility that these calls return an error result, but that the "error" is simply `EAGAIN` so the call simply needs to be repeated. Another critial difference is that *writes can now be partial!* (Remember that `read()` calls can always be "partial" whether blocking or nonblocking mode.) I.e., `write()` (or similar) can return a byte count that is less than what was requested in the call.

Here is example code that works for nonblocking mode data transfer with `read()-write()`:

```
char buff[4096];
ssize_t nread, nwritten, total;

nwritten=0;
while (nwritten != -1 && (nread = read(source_fd,buff,4096)) > 0) {
  total = 0;
  do {
    if ((nwritten = write(target_fd,buff+total,nread-total)) == -1) break;
    total += nwritten;
  } while (total < nread);
}
```

The problem with the above code is the the `write()` loop effectively performs a *busy wait*. In this server, that is not a very good approach (though other workers can continue). Instead, you are to handle failure to completely write data that has been read, by temporarily storing the unwritten data (with your client object), and then determining (via `epoll_wait()`) when further data can be written to the target FD. Keep in mind, though, that even if `epoll_wait()` says writing is possible, there are no guarantees about the amount of data that can be written. Thus, it could take several more attempts to get all the transfer data written out.

Note that nonblocking mode has *no effect* on *regular files* or *block devices*. I.e., no effect when reading from disk drives and the like. In such cases, I/O requests will typically block briefly (meaning cause a *context switch*). The key difference is that I/O calls cannot *block indefinitely*. Another term used for these calls is that they are *"slow system calls"*. The issue that nonblocking mode is intended to address is *indefinite blocking*.

Timers:
To create a robust server that can deal with broken/malicious clients, it must be possible to free all server resources if clients do not successfully pass the protocol exchange in a reasonable time. Otherwise, it would be trivial to carry out a denial of service (DoS) attack on the server by simply opening new connections and then never following through with the protocol exchange. The only way (or at least the only efficient way) for the server to notice that a client is not responding is by using a *timer* so that if the protocol exchange is not completed within a reasonable amount of time, the connection/socket gets closed and resources are freed.

In Lab #3, you could have a separate (temporary) thread for each new client's protocol exchange, and could set a thread-specific timer. Something similar could be done in this lab: when a thread pool worker gets the task to carry out the initial protocol exchange for new client, it could carry it out to completion, and set a timer for the thread so that the thread could not be blocked too long if cient is broken/malicious. This is probably not ideal, however, since carrying out the entire protocol exchange in a single worker execution cycle is going to tie that worker up for longer than it needs to be (think about network time for client that is on other side of world).

Instead, you are suggested to use a different strategy for timers, to avoid tying up workers for the entire protocol exchange period. This requires that you break the protocol exchange into two conceptual stages for the server (with two different clien states). When the protocol ID is sent, a (POSIX) timer would be set. If it expires before the client FD becomes active again to deal with the shared secret, you will have to notice this and remove the client. While this could potentially be dealt with through the standard *signal-based* timer mechanism (using `epoll_pwait()`), it is easier to use Linux' *timerfd* mechanism. The timerfd mechanism allows a *file descriptor to be associated with a timer*. That FD can now be monitored for timer expiration using epoll. See `timerfd_create()` and related calls to see about setting up a timerfd. The timerfd mechanism is covered in Section 23.7 of the textbook.

There are some complexities involved in using timerfds with your server. One is that the server must be able to connect an expired timerfd's FD with the client (i.e., its socket FD). Unlike POSIX timers where this information can be passed via the signal handler, there is no mechanism for passing information for timerfds. Thus you will have to be able to identify the associate client from a timerfd. In fact, when epoll tella you an FD is ready, you will now have to determine if the FD is a socket or PTY, or if it is a timer.

One way to make the latter simpler is to use a *second epoll unit* that just monitors timers, and have the first/main epoll unit monitor the second, timer epoll unit. When the primary epoll unit says that the timer epoll is ready for reading, this means that one or more timers have expired. Note that testing appears to show that the main epoll loop will continue to identify the timer epoll unit as ready for reading until the timers are closed, even if the timer epoll FD is added to the main epoll to be monitored as edge triggered. You may thus have monitor the timer FD as *oneshot* to avoid multiple timer epoll FDs tasks being added to the thread pool queue each time a timer expires. Be sure to test this functionality using a client that does not respond with the shared secret!

EPOLLONESHOT:
The high degree of concurrency that comes from having an `epoll_wait()` loop thread and a set of tpool worker threads can cause problems if one does not set epoll up properly. Once the `epoll_wait()` loop Pthread is running, it has the potential to run many times adding the same FD to the tpool queue before a worker Pthreads gets awakened to start processing the first instance of the FD added to the tpool queue. While your server should obviously be able to handle workers dequeueing an FD that is already being processed by another worker and/or has already been closed, enqueueing the same FD multiple times in a row is not going to lead to efficient use of the thread pool.

While it seems like setting the FDs to be *edge-triggered* in epoll would solve the above problem, it does not. Because our clients are (supposed to be) in *noncanonical mode*, a

command line can arrive as a sequence of packets containing individual characters. Even with edige-triggered mode, this will likely cause the same socket FD to be identified as ready multiple times. So while edge-triggering may reduce the above problem a bit as opposed to level-triggered, FDs can still end up being enqueued multiple times before any are processed by tpool workers.

The best solution is to set the FDs up in *oneshot* mode, by including `EPOLLONESHOT` in your events. This way, each FD will be enqueued only once prior to it being processed by a tpool worker Pthread. Note: as the final step in processing an FD, you must re-enable the FD in the epoll unit (call `epoll_ctl()` using `EPOLL_CTL_MOD` for the `op` (operation) argument).

EPOLLRDHUP:
In addition to monitoring for `EPOLLIN`/`EPOLLOUT` events, you probably also want to monitor for the `EPOLLRDHUP` event. Here is what the man pages say about `EPOLLRDHUP`: "Stream socket peer closed connection, or shut down writing half of connection. (This flag is especially useful for writing simple code to detect peer shutdown when using Edge Triggered monitoring.)"

Note that the other two "error" condition events, `EPOLLERR` and `EPOLLHUP`, are automatically monitored for by `epoll_wait()`, so do not need to be explicitly set in the events for the epoll unit. `EPOLLRDHUP` is not however, so you need to explicitly include it in the events to monitor for (and test for it with the ready FDs obviously).

Program Submission:
For this lab, you are to submit several files:

1. `lab5-server.c` – your server *source code*
2. `tpool.h` – *header file* for using your thread pool
3. `tpool.so` – a *dynamic library* containing your thread pool
4. `tpool.c` – your thread pool *source code* (if changed from Lab #4)

Because you are submitting several files, you are to submit them as *gzipped tar archive*, in a single file named `lab5.tgz`, to the Lab #5 Dropbox on the course D2L website.

The following command will tar up a directory:

```
tar czf lab5.tgz DIRECTORY_CONTAINING_REQUIRED_FILES
```

The `DIRECTORY_CONTAINING_REQUIRED_FILES` should be your name, formatted analagously to: `Norman_Carver`

Improperly packaged/submitted files will not be accepted! Note also, that your *libraries must be compiled for 64-bit Linux systems!* 32-bit libraries will not be able to be tested, so will not be accepted.

Make sure you have *only the desired files in the directory* when you create the tar archive!

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!!**