ECE361 Text Conferencing Lab Tutorial

Getting Started

Organizing Your Development

The text conferencing lab is broken down into 2 sections, each due in a different practical. This lab is much longer than the file transfer lab and is more open-ended; plan your time accordingly and remember to test frequently. If you are working on the client and your partner is working on the server (or vice versa), remember to try integrating your code early. In particular, make sure you agree on message formats.

At the server, one of the first problems you will run into is how to deal with requests from multiple clients simultaneously. Similarly, at the client, you will quickly encounter a problem where you need to wait for user input on the command line and messages from the server simultaneously. Both are *I/O multiplexing* problems, and there are (at least!) two common approaches for dealing with them:

1. **Synchronous I/O multiplexing**. With this approach, we use special functions to block on multiple I/O sources *simultaneously*—the key point being that as soon as *any of them* have a message, our program unblocks. Blocking refers to a state where our program is stuck waiting for messages.
2. **Multithreading**. With this approach, we use different *threads* to block on each I/O source at the same time. Multithreading is a powerful programming technique, but can be difficult to debug if you're not aware of the issues that can arise (e.g., synchronization).

In this document, we will go over the basics of these approaches so that you can choose which is best for you. Note that your client and server can use different approaches.

Caution: you may find sample code (e.g., in Beej's guide) that uses multiple *processes* to handle requests at the server. **We do not recommend that approach**. Processes are much harder to use for this lab than threads, since threads share the same address space (and hence variables) but processes don't. This makes it difficult to communicate (e.g., share variables) between processes.

Testing

You are required to run your server on a different UG machine than your client(s) when testing your code. To ensure proper handling of edge cases, we suggest you test your code with at least 3 clients connected to the server; it is fine if these clients are all running on the same machine as each other.

In the lab sessions, we will ask you to demonstrate the correct functionality of the commands given in the lab handout. We have also thought of some scenarios that constitute edge cases!

Readings

This lab assumes familiarity with the sockets library used in the File Transfer lab. However, **in this lab you must use TCP** instead of UDP; accordingly, **the programming techniques used will be different**. The following sections will provide reference material on TCP, synchronous I/O multiplexing, and multithreading.

FAQ

Section 1

Q: When you send a message in a conference, does that message need to become visible to everyone in that session?

A: Yes, if a user has joined a session then any messages they type (which are not commands) should go to everyone in that same session. This text conferencing lab is basically having a server set up a group chat between a set of people.

Q: Is it okay to hardcode a maximum number of sessions at the server?

A: Yes. Have a graceful way of handling the edge case though.

Q: Why am I getting an undefined reference to pthread_x?

A: You are probably not linking with the pthread library. Add `LDFLAGS=-pthread` to your makefile.

Q: Why does recv read 0 bytes?

A: Receiving a message of length 0 when using TCP on Linux means the connection was closed by the other end.

Section 2

Q: How many additional features do we need to implement?

A: Two! The marks are split between them evenly.

Q: What should go in the text file? How long does it need to be?

A: You should explain what your new functionalities are, how they work, and the rationale behind your design (there are no length requirements, so it doesn't have to be long). Imagine you are reading through your code a year from now, trying to remember what new features you chose for this lab and how they worked. Now imagine you open your explanation file. What would be helpful to have in there?

Function Reference: TCP

In the File Transfer lab, we saw that UDP sockets were message-oriented. Among other things, this meant that one call to `sendto` at the transmitter corresponded to one call to `recvfrom` at the receiver. **This is not the case for TCP**.

TCP sockets are stream-oriented (or byte-oriented), meaning that message boundaries are not preserved. To put this another way, **TCP sockets do not tell us where one message ends and another begins**—all that TCP sees is a sequence of bytes. **The *restructuring* of received data must be done at the application layer, by us**. Take a minute to let this sink in!

As an example, suppose I am the transmitter and you are the receiver. At some point, I send you the byte sequence "hello". A little while later, I send you "world".

- Case 1: If you call the receive function with a buffer of size 10 (big enough to fit everything I sent you) then you will read out "helloworld".
- Case 2: If instead you call receive with a size-7 buffer, you will read out "hellowo". If you call receive *again* you will read out "rld".

This example illustrates two major differences from UDP. First, if we transmit two separate byte sequences, the receiver cannot distinguish them—instead, it just sees one long sequence. Second, if you call the receive function with a buffer that is not sufficiently large to empty the receive queue, then the leftover bytes will be returned on subsequent receive calls (rather than being discarded).

Suggestion: A common technique for dealing with the stream-oriented nature of TCP is to transmit the length of the message before transmitting the message itself (you need to figure out an unambiguous way to do that). Once the receiver reads out this length, it will know how many more bytes it needs to read before it reaches the message boundary. If the receiver reads exactly this many bytes (no more and no less), we will effectively have message-oriented TCP. This is not difficult, but it is more nuanced than it appears at first glance. For example, suppose we find out that the message is 900 bytes long. We then make a call to our receive function with a buffer size of 900 bytes. If 500 bytes are already available, then our receive function will return immediately after populating our buffer with those 500 bytes. We then need to be careful that next time we call our receive function, we pass in a buffer of *only 400 bytes*. If we passed in a buffer of 900 bytes again, we might accidentally receive part of the next message, which could be complicated to deal with!

A few exercises related to this suggestion (not for marks):

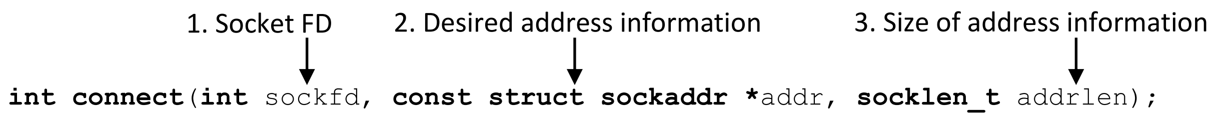
- How can we transmit the message length unambiguously? That is, how does the receiver know which bytes contain the message length?
- How can we ensure that the receiver reads the message length from the socket without reading any of the message contents?
- What unintended consequences might arise from passing in, e.g., a 2000 byte receive buffer when trying to read the message length from the socket?

socket

We are already familiar with `socket`, thanks to the File Transfer lab! The one thing worth pointing out is that since we **must** use TCP for this lab, we will need to pass in `SOCK_STREAM` as the socket type.

connect

`connect` is used to try to establish a TCP connection. Usually, we use this as a client when trying to establish a connection with a server. If `connect` is successful, then we can start sending and receiving data in a reliable, stream-oriented manner.


1. Socket FD
2. Desired address information
3. Size of address information

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Arguments

1. The socket FD we want to use to establish the connection.
2. **(POINTER)** Address we want to connect with. Fill this in with the server's address information.
3. Size of the `struct sockaddr` passed in for argument 2.

Return Value

Returns 0 on success and -1 on failure. If `connect` fails, it is likely because your server is not running, or you entered its address information incorrectly.

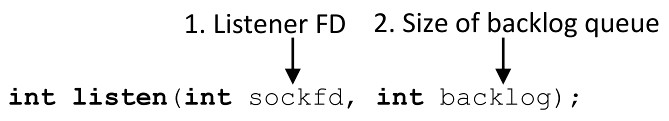
References

- https://man7.org/linux/man-pages/man2/connect.2.html

listen

Typically, a server will have one special socket called a *listener*, which does nothing except wait for incoming connection requests. When a connection request arrives, the listener reaches under the hood in the socket library, indicates that there is a pending connection, *and then continues listening*. This is because the listener's only job is to listen—it is not responsible for dealing with the connection requests (that is what `accept` is for).

To mark a socket as a listener, we pass it into the `listen` function. This function will return immediately, and (assuming success) the socket will automatically start listening for connection requests in the background.



```
int listen(int sockfd, int backlog);
```

Arguments

1. The socket we wish to mark as a listener.
2. The maximum number of pending connection requests. This value matters if connection requests arrive faster than the server can respond to them.

Return Value

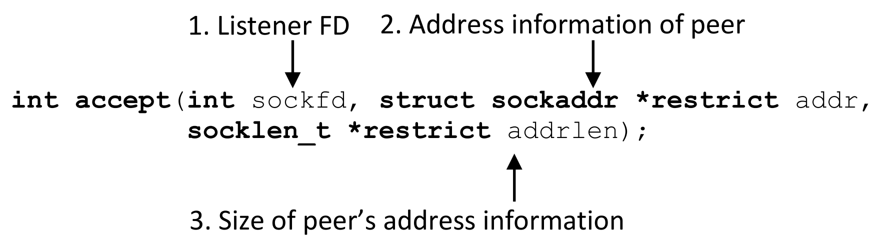
Returns 0 on success and -1 on failure.

References

- https://man7.org/linux/man-pages/man2/listen.2.html

accept

Whenever the listener indicates that there is a pending connection, the server can complete that connection by calling `accept`. This function does *not* make the listener part of the connection; instead, it creates a *new* socket to complete the connection. This way, the listener can keep on listening.


1. Listener FD
2. Address information of peer
3. Size of peer's address information

```
int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict addrlen);
```

Arguments (inputs)

1. Socket FD of the listener. This socket must be bound and listening.

Arguments (outputs)

2. **(POINTER)** Like `recvfrom`, you need to create an empty `struct sockaddr_storage` and pass it into `accept` via pointer. The address information of peer (connection initiator) will be automatically filled in.
3. **(POINTER)** Same purpose as the `addrlen` argument for `recvfrom`. Initialize it the same way.

Return Value

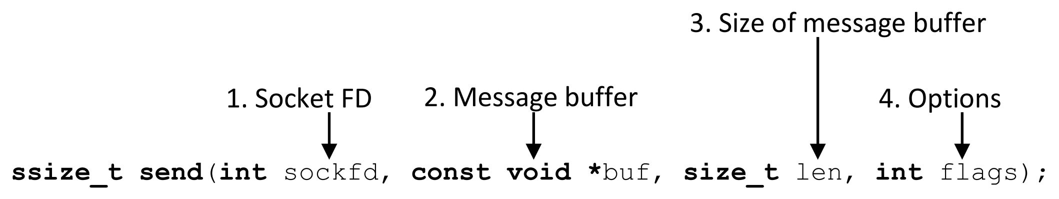
Returns the FD for the newly created socket upon success and -1 on failure. To communicate with the peer who requested the connection, you will use this new socket.

References

- https://man7.org/linux/man-pages/man2/accept.2.html

send

`send` is used to send a message through a connected socket. As mentioned earlier, be cautious about the fact that TCP does not preserve message boundaries. One nice thing about being connected is that we don't need to pass in the destination address every time we send or receive; this is remembered by the socket library under the hood. Alternatively, the standard file I/O function `write` can be used instead of `send`.



```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Arguments

1. The socket FD tells `send` which socket you want to use to send messages. This socket must be connected.
2. **(POINTER)** Pointer to the message (i.e., byte sequence) you want to send.
3. Length of the message you want to send (i.e., number of bytes).

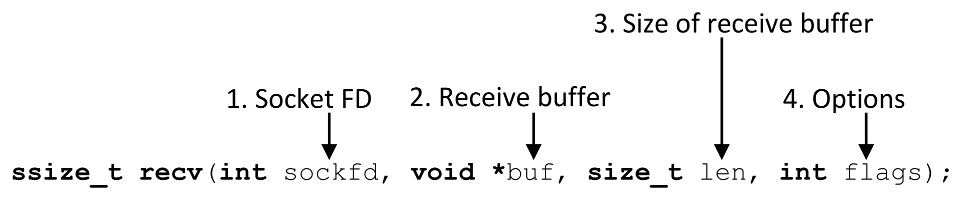
4. This argument allows you to specify transmit options, if needed.

Return Value
Returns the number of bytes sent on success and -1 on failure.

References
- https://man7.org/linux/man-pages/man2/send.2.html

recv
`recv` is used to receive bytes from a socket. Alternatively, the standard file I/O function `read` can be used instead of `recv`.

As has been emphasized throughout this document, when we use TCP the connection becomes stream-oriented. What this means for us, as programmers, is we need to keep track of where one message ends and another begins (one strategy for this was suggested in at the start of the Function Reference: TCP section). In any case, messages are not given to us atomically as with UDP, so we need to exercise more caution with the `recv` function to make sure that the logic for receiving messages is correct.



```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Arguments
1. The socket FD tells `recv` which socket you want to receive bytes from.
2. **(POINTER)** Address of buffer for `recv` to copy received bytes into.
3. Length of your receive buffer. This can be thought of as the *maximum* number of bytes you wish to receive. This is a bit different than the story for UDP, and again it comes back to TCP not preserving message boundaries (sometimes we might not *want* to receive more than X bytes, because we know that the message currently being received is X bytes long).
4. This argument allows you to specify some options, if needed.

Return Value
Returns the number of bytes received on success, -1 on failure, and 0 if the connection was closed from the other side (i.e., by the peer).

References
- https://man7.org/linux/man-pages/man2/recv.2.html

Function Reference: Synchronous I/O Multiplexing

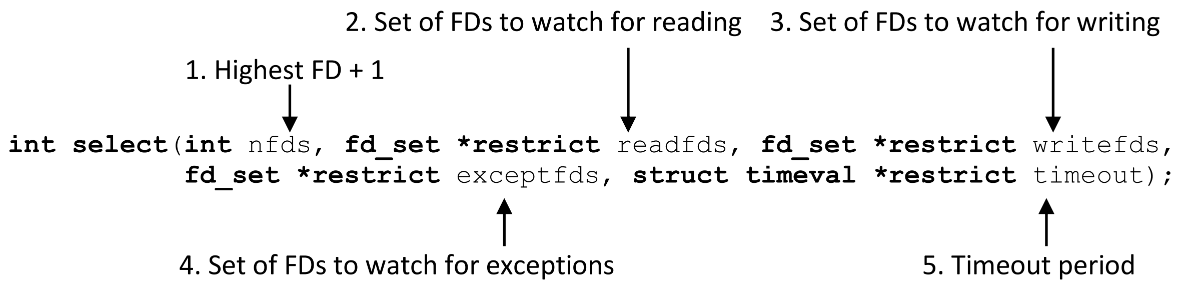
This is a particularly powerful technique for dealing with the I/O multiplexing problem at the server (it can also be used at the client though). Some references to help get started include:

- Section 7.3 in Beej's Guide: https://beej.us/guide/bgnet/html/#select. **Highly recommended!**
- Linux manual page on select: https://man7.org/linux/man-pages/man2/select.2.html

If you plan on using synchronous I/O multiplexing, you must read section 7.3 from Beej's Guide because it contains an excellent introduction to the topic as well as sample code. There is not much we can add that hasn't already been said, so this section will be shorter than the others.

select

Synchronous I/O multiplexing is possible because the `select` function allows us to monitor sets of FDs simultaneously. This is summarized well in Beej's guide, "select() gives you the power to monitor several sockets at the same time. It'll tell you which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions, if you really want to know that".


2. Set of FDs to watch for reading
3. Set of FDs to watch for writing
1. Highest FD + 1
4. Set of FDs to watch for exceptions
5. Timeout period

```
int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,
           fd_set *restrict exceptfds, struct timeval *restrict timeout);
```

Arguments

1. You need to pass in the value of the largest FD in your set, plus one. Make sure you keep this value up to date when you add and remove FDs from the set.
2. **(POINTER)** Here, you specify the set of FDs that you want `select` to monitor for *readability*. From the manual, "A file descriptor is ready for reading if a read operation will not block…After select() has returned, `readfds` will be cleared of all file descriptors except for those that are ready for reading". This last part means that prior to calling `select`, you might want to make a copy of your `readfds` if you plan to use it again.
3. **(POINTER)** Set of FDs that you want `select` to monitor for writability. You can pass in `NULL` if you don't need this.
4. **(POINTER)** Set of FDs that you want `select` to monitor for exceptions. You can pass in `NULL` if you don't need this.
5. **(POINTER)** Specifies the maximum amount of time that `select` should wait before returning. You can pass in `NULL` if you don't need this.

Return Value

Returns the total number of FDs contained in the modified FD sets (read, write, except) on success, -1 on failure, and 0 on timeout.

References

Section 7.3 in Beej's Guide and the Linux manual page on select, linked at the start of this section.

Function Reference: Multithreading

Multithreading can be achieved using the pthreads library. Before using this library, you will need to add the following line to your makefile: `LDFLAGS=-pthread`. Some guides for getting started include:

1. POSIX Threads Programming: https://hpc-tutorials.llnl.gov/posix/
2. Linux Tutorial: POSIX Threads: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
3. Multithreaded Programming (POSIX pthreads tutorial): https://randu.org/tutorials/threads/

The key thing to understand is that multiple threads can be used to run tasks in your program *independently*. When programming with multiple threads, we often think of the threads as executing concurrently. This can be helpful for identifying scenarios in which threads may conflict with each other, and thus need to communicate among themselves or *synchronize*. In reality, threads do not always execute concurrently since their execution is determined by the scheduler within the operating system (OS)—this is a detail we will revisit soon.

When you create a thread, you pass it an *entry point*, which is a function for it to execute. All programs consist of at least one thread (called the main thread) which is given the `main` function as its entry point. With pthreads, to create a thread which runs the function `my_func` and passes argument `my_arg` into `my_func`, you can run the following code in your main thread:

```
pthread_t new_thread;
pthread_create(new_thread, NULL, my_func, (void*)&my_arg);
```

A few things to note:

1. When the new thread returns from `my_func`, the new thread shuts down. As such, if `my_func` is an infinite loop, then the new thread will continue running forever.
2. It may be the case that you want your main thread to wait for the new thread to shut down, in this case you can call the function `pthread_join` from your main thread. **Note that this will not *cause* the thread to shut down**, you need to figure out some other way of doing that.

Before going further, let's return to the I/O multiplexing problem for the text conferencing lab, where you wish to receive messages from multiple sources simultaneously. A multithreaded approach to solving this issue would begin by using separate threads to listen to each I/O source independently. For example, you could use your main thread to listen to one I/O source and create threads (along with functions for them to run) that listen to the other I/O sources. When one of the sources produces a message, the thread that's listening to it will then be able to respond to it while the other threads continue listening (or responding) to their own I/O sources. Note that you can call `send` from one thread and `recv` from another thread on the same socket without issues (source).

When designing a multithreaded program, remember that threads *may* execute concurrently (but not always). Whether this happens depends on how your OS chooses to schedule the threads, and what the state of each thread is. It is possible for a thread to be *blocked*/*waiting*, which means it is waiting for a *resource* that is currently being used by a different thread. An example of such a resource is a *mutex*, which we will discuss next.

An important thing to know about threads (of the same process) is that they share the same address space, i.e., they can all access and modify global variables in your program. Without going into too much detail, let's just say that multiple threads accessing shared variables at the same time can lead to unexpected behaviour. To deal with such scenarios, we can make use of a *synchronization primitive* called a mutex. A nice summary can be found on this site: "a Mutex is a lock that we set before using a shared resource and release after using it. When the lock is set, no other thread can access the locked region of code". When the mutex lock is set, another thread which tries to acquire it will be put into the *blocked* state; this thread will be *unblocked* when it is its turn to access the shared resource. The functions you'll need to know about for mutexes are:

1. pthread_mutex_init
2. pthread_mutex_lock, pthread_mutex_unlock

You are not required to use mutexes in your code, but they could be helpful. **Remember: if you have multiple threads reading or writing the same resource (e.g., variable or data structure), it is almost certainly the case that you need to use a mutex to control access to that resource**. This site shows an example of a mutex in action.