Assignment 5

cpe 357 Winter 2024

`Rarely do people communicate; they just take turns talking.`

— /usr/games/fortune

`That's cool! It looks hard...`

— TLD, my 12 y.o., when I showed him mytalk

Due by 11:59:59pm, Wednesday, March 6th.
This assignment is to be done individually (but see if you can interoperate with your classmates).

Program: `mytalk`

`talk(1)` is a program that allows a user on one machine to send and receive messages from another user on another machine. To do this, it splits the screen into two parts, displaying the remote user's text in the top half and the local user's in the lower half. `talk(1)'s` connections are managed by a talk daemon, a process that exists in the background, waits for network connections, then asks the targeted user if they want to talk.

Your task is to implement `mytalk`, a simplified version of `talk(1)` that has a server and a client but no daemon and a simpler protocol.

In the process you will gain:

- experience with networking,
- experience with concurrency,
- experience with asynchronous IO (more concurrency),
- experience linking against a given library (and visibility into libraries at all), and
- tangential experience dealing with ncurses and terminals.

Good news: I've abstracted away most of the difficult IO processing into a library leaving you mostly just the networking.

Running mytalk

`Mytalk` is one program, with two modes, *client* and *server.* In server mode, mytalk opens a network socket and listens for connections from a client elsewhere in the world. In client mode, the program opens a socket and attempts to connect to a server on a remote host.

Usage:

```
mytalk [ -v ] [ -a ] [ -N ] [ hostname ] port
```

Options supported:

`-v`	increases verbosity. May be repeated.
`-a`	(server) accept a connection without asking (useful while debugging)
`-N`	do not start ncurses windowing (useful while debugging)

If the *hostname* option is present, `mytalk` acts as a client. *hostname* is the name of the remote host to which to connect at the given port.

If *hostname* is not present, `mytalk` acts as a server, opening a listening socket on the local machine at the given port.

The `-v` option increases verbosity. What exactly it does is up to you, but it can be very helpful while debugging to be able to turn on and off code that narrates what your program is doing.

The `-a` option tells the server to accept all connections without asking. This is not really useful in the final version of the program, but it really speeds up testing since you don't have to keep switching windows just to type "y".

The `-N` option tells mytalk not to start the ncurses windowing. This is also not terribly useful in the final version of the program, but it can make debugging easier since it will interact better with, say, `gdb`.

General rules

- Client and server will use TCP (`SOCK_STREAM`) for communications.
- When the server starts it opens a socket on the given port and listens for connection attempts from a client. (See TCP/IP Programming, below.)
- When a client starts, it attempts to open a connection to the server on the given host at the given port. If the connection is established, the client sends a packet containing the user's username and waits for a response.

  In the meantime it displays a message of the form:

  `Waiting for response from` *hostname*`.`
- When the connection is established, the server displays a message of the form:

  user`@`*hostname*`.  Accept (y/n)?`

  If the user answers with "yes" or "y" (case insensitive), the server responds with "ok". If not, it responds with anything else.
- If the client receives any answer other than "ok", it terminates its connection and exits. (This will cause the server's connection to close so both will exit.) It's final message will be:

  hostname `declined connection.`
- Once this introduction is completed, both server and client switch to a graphical mode where text from the remote machine is displayed in the top half of the screen and local text is displayed in the lower half.

  This is done using a library known as ncurses. Good news: I have abstracted away the complexity of ncurses and dealing with ncurses IO into a library. (See below.)
- Both server and client wait for IO from either the local or remote user and display it when it becomes available. If you have two file descriptors and you don't know which one will become ready first, `poll(2)` is your friend for determining this in a nonblocking fashion.
- A message is passed when either a complete line of input is available or if EOF is detected on an incomplete line. (The library more or less takes care of this.)

  Note: The `talk(1)` protocol requires that erase and kill characters be supported as well as a few other things. The library takes care of this.

- Message passing stops when either user types either the EOF character (^D) or generates a SIGINT (^C) causing that end to terminate. Note: `ncurses` installs its own SIGINT handler, so if you must install your own handler, be sure to install it after starting windowing.
- When the other end detects that its peer has terminated, it displays a message of the form:

  ```
  Connection closed.  ^C to terminate.
  ```

  and will perform no further IO.

Example

Figure 1 shows what the system looks like on both server and client machines while trying to establish a connection, and Figure 2 shows the system during a chat. Figure 3 shows the system after the client has exited. (The server exiting is symmetric.)

TCP/IP Programming

This is a quick description of the steps in establishing and using a TCP connection to refresh what we talked about in class.

Server side

1. Create the socket with `socket(2)`
2. Attach an address to it with `bind(2)`
3. Wait for a connection with `listen(2)`
4. Accept a connection with `accept(2)`
5. send and receive with `send(2)` and `recv(2)` until done
6. `close(2)` any remaining sockets

Client side

1. Look up peer address with `getaddrinfo(3)` (or `gethostbyname(3)` which is old-fashioned, but simpler)
2. Create the socket with `socket(2)`
3. Connect to the server using `connect(2)`
4. Send and receive with `send(2)` and `recv(2)` until done
5. `close(2)` any remaining sockets

Both sides

1. Note that many of these functions specify that arguments or results are in *network byte order.*
2. Host-to-network (and vice-versa) conversions can be done sith `htonl(3)`, `htons(3)`, `ntohl(3)`, and `ntohs(3)` as appropriate.

Server

```
unix6% mytalk 1234
Mytalk request from pn-cs357@unix1.csc.calpoly.edu.  Accept (y/n)?
```

Client

```
unix1% ./mytalk unix6.csc.calpoly.edu 1234
Waiting for response from unix6.csc.calpoly.edu.
```

Figure 1: Before a connection is established

Server

```
Hi, server
What's new?




========================================
Hi, Client!
Stuff typed on the server machine
More of the same
```

Client

```
Hi, Client!
Stuff typed on the server machine
More of the same




========================================
Hi, server
What's new?
```

Figure 2: During a chat session.

Server

```
Hi, server
What's new?
bye

Connection closed.  ^C to terminate.

========================================
Hi, Client!
Stuff typed on the server machine
More of the same
bye
```

Client

```
unix1% ./mytalk unix1.csc.calpoly.edu 1234
Waiting for response from unix6.csc.calpoly.edu.
unix1%
```

Figure 3: After the client terminates the a connection.

The Talk library

Because dealing with `ncurses` in particular—and IO processing in general—is a pain, I have provided a library for you with some useful functions to handle the windowing and most of the IO difficulties. The functions in the library are given in Table 1.

The library itself is available on the CSL machines in `~pn-cs357/Given/Talk/`. The 32-bit version is in the `lib` subdirectory, and the 64-bit version is in `lib64`. The header file is `include/talk.h`.

To use such a library file, lib*name*.a, you can do one of two things. First, you can simply include it on the link line like any other object file:

```
% gcc -o prog prog.o thing.o libname.a
```

Second, you can use the compiler's library finding mechanism. The `-L` option gives a directory in which to look for libraries and the `-l`*name* flag tells it to include the archive file `lib`*name*`.a`:

```
% gcc -o prog prog.o thing.o -L. -lname
```

Because the library uses ncurses for windowing, you need to include the ncurses library, too, so to build, you'll do something like:

Compile: gcc -c -g -Wall -I ~pn-cs357/Given/Talk/include foo.c

Link: gcc -g -Wall -L ~pn-cs357/Given/Talk/lib64 -o prog foo.o -ltalk -lncurses

For the 64-bit version, you'd use `lib64`, or you can put both on the link line and let the linker choose.

Note: When linking a program, the linker resolves symbols (finds missing names) in the order in which it encounters them. This means that **the order in which libraries appear on the link line matters.** If `-lncurses` appears before anything that needs ncurses, none of those symbols will be included.

Why are computers so $#%$ complicated? (a note on asynchronous IO)

If `read_from_input()` and `write_to_output()` read and write lines, respectively, and `poll(2)` tells when a file descriptor is ready to be read from, what do we need with those other functions?

As with many things computer-related, the reality is more complicated than it first appears (and even the below isn't totally true).

Because of the way it processes IO, ncurses has to shift the terminal into what is known as noncanonical mode. In normal, canonical, mode, the terminal line discipline gathers input up into lines, processing special characters like erase and kill in the process. In noncanonical mode, characters become available to the application as soon as they're typed.

Who cares?, you might say. Turns out, you do: Being in non-canonical mode (technically cbreak mode, for old timers) means that the local input file descriptor will become ready for reading the moment the user types the first character of a line. If the program calls `read_from_input()` then, it will block (wait) until the entire line has been typed before returning. While it is blocked it will not be listening to the network socket so it will neither see nor relay any messages from the other user. If the local user types a key and never hits <enter> it will remain forever deaf.

This is where the other functions come in. `update_input_buffer()` and `has_whole_line()` allow the program to read as much input is available into an internal buffer and to check if it has read an entire line without blocking. This means that the program knows whether it can call `read_from_input()` without blocking or if it should go back to waiting on `poll(2)` for further input from anywhere.

Where does `has_hit_eof()` come in then?

A whole line means a sequence of characters ending in newline, or any sequence of characters having been read before encountering EOF. `read_from_input()` returns the line if there exists one, or 0 if all that's left is EOF. If the program received the string "this is a stringˆD", this means

`void start_windowing(void);`	Turns on the ncurses windows if stdin and stdout are ttys. Does nothing if they are not.
`void stop_windowing(void);`	Turns off the ncurses windows if they are on. Nothing otherwise.
`int read_from_input(char *buf, size_t len);`	If windowing is enabled, read up to `len-1` bytes from the input window, otherwise read from stdin using `fgets(3)`. The buffer will be nul-terminated and will include the final newline if it exists and fits. Returns the number of bytes read on success, 0 on EOF, and `ERR` on error. (ncurses doesn't actually define any error conditions.) If the internal buffer does not include a full line, blocks until there is one or EOF.
`int write_to_output(const char *buf, size_t len);`	If windowing is enabled, write `len` bytes to the output, otherwise write to stdout (using `fputs(3)`). Returns `OK` on success and `ERR` on failure.
`int fprint_to_output(const char *fmt, ...);`	uses `write_to_output()` to write formatted data to the appropriate output in the manner of `printf(3)`.
`void update_input_buffer(void);`	If windowing is enabled, update the library's internal buffer as far as the next newline or EOF in a nonblocking fashion (see below). Does nothing if windowing is disabled.
`int has_whole_line(void);`	If windowing is enabled, return TRUE if the library's buffer holds a whole line ready to read, FALSE otherwise. Returns TRUE if windowing is disabled.
`int has_hit_eof(void);`	If windowing is enabled, return TRUE if the library has encountered EOF while reading, FALSE otherwise. Calls `feof(stdin)` if windowing is disabled.
`int set_verbosity(int level);`	set the verbosity level within the library. It doesn't say much, but it can be made more chatty.

Table 1: The contents of libtalk.a

that `read_from_input()` would first return 16 for the string, then 0 for the EOF on the next call. But why would that next call ever be made? So far as `poll(2)` is concerned that file descriptor will never be ready to read since it's already seen EOF. This is what `has_hit_eof()` is for. Calling `read_from_input()` again risks blocking, but `has_hit_eof()` can check for this condition without that concern.

While it's at it, the library does various input processing required by the `talk(1)` specification. It processes erase and kill characters, encodes any non-printable characters in a way that will be harmless to the remote terminal (octal strings), and passes through bel characters in such a way as to make the remote terminal beep. If the internal buffer—which is the same length as a row of the screen—fills up, it discards further characters with a cheerful beep until either newline or EOF is encountered.

In conclusion, `read_from_input()` and `write_to_output()` do almost what you want, in a slightly inaesthetic way, but to really do it right is a little more complicated.

Endianness

Different machines order the bytes of multi-byte integers differently With a single byte, the meaning of an address is clear, but with multi-byte data, such as integers, the question arises, "Which end of the data does the address really point to?" The two obvious possibilities are the most significant byte or the least significant byte. Each is quite valid, but unfortunately, they are incompatable.

Consider the number $0xAABBCCDD$. If represented as a little-endian number at address A, the least significant byte, DD, comes at location A, then the more significant bytes follow at locations $A+1$, $A+2$ and $A+3$. For big-endian, the most significant byte, AA comes at address A, and the less significant bytes follow:

Little Endian

Address	+0	+1	+2	+3
	DD	*CC*	*BB*	*AA*

Big Endian

Address	+0	+1	+2	+3
	AA	*BB*	*CC*	*DD*

One can easily be mapped to the other by reversing the order of the bytes.

The reason you care is that not all machines have the same endianness, and a small number written on a little-endian machine suddently becomes huge if read on a big-endian one. For portability, we must decide on a standard for interchange. That is network byte order.

Intel x86 machines are little-endian. Network byte-order is big-endian.

Tricks and Tools

Things to consider

- Messages in network packets are not necessarily nul-terminated. If you want your messages to be nul-terminated, be sure to send the nul or add one on arrival (a safer choice).
- Valid port numbers run from 1 to 65535, but ports below 1024 are typically reserved for root.
- As mentioned above, you do not know whether the next message will be coming from the local or remote user. `poll(2)` can help you with this.
- Be wary of firewalls. If your connections aren't working, be sure that it's not because your computer isn't allowing it. *Usually* they don't block connections from localhost to itself, but the key is *usually...*

Library functions, etc.

Some (potentially) useful functions, man pages, and programs are listed in Table 2.

`snprintf(3)`	write a limited amount of formatted data to a string
`poll(2)`	check a set of file descriptors for readiness
`getopt(3)`	Your best friend for option parsing; well worth the scary-looking learning curve
`perror(3)` `strerror(3)`	Don't think; know what's wrong.
`socket(2)`	creates a socket
`bind(2)`	attaches an address to a socket
`listen(2)`	tells a socket to listen for connections
`accept(2)`	accept a connection request on a listening socket
`connect(2)`	attempt to connect to a remote socket
`send(2)`	send a message, via a socket, to a remote socket
`recv(2)`	receive a message on a local socket
`close(2)`	close a file descriptor (socket or otherwise)
`getaddrinfo(3)` `gethostbyname(3)`	look up a host address from its name
`inet_ntop(3)`	Render a printable version of an IP address
`getnameinfo(3)` `gethostbyaddr(3)`	look up a host's name from its address
`getuid(2)`	return the calling process's user id
`getpwuid(3)`	look up the password entry for the given uid (to get username)
`strcmp(3)` `strcasecmp(3)`	string comparison routines
`ncurses(3)`	way more than you ever wanted to know about ncurses
`reset(1)`	reset the terminal to a reasonable state after a crash of a program that changes it. Similarly, "stty sane".
`pause(2)`	block until a signal is delivered, possibly forever
`htonl(3) htons(3)` `ntohl(3) ntohs(3)`	Convert from host to network byte order and vice-versa

Table 2: Some potentially useful system calls, library functions, and man pages

Coding Standards and Make

See the pages on coding standards and make on the cpe 357 class web page.

What to turn in

Submit via `handin` in the CSL to the `asgn5` directory of the `pn-cs357` account:

- Your well-documented source files.
- A makefile (called `Makefile`) that will build your program when given the target `mytalk` or no target at all.

- A README file that contains:
  - Your name.
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

  The README file should be **plain text,** i.e, **not a Word document**, and should be named "README", all capitals with no extension.

Sample runs

This is really an interactive program. I will place an executable version of `mytalk` on the CSL machines in `~pn-cs357/demos` so you can run it yourself.