CS 214 / 2021-04-26
==========

Final Exam: May 10, starts at 4 PM
    between 1 and 3 hours
    24-hour window
    done on-line via Sakai
        primarily multiple-choice
            -> look for words like "not" in the questions
        open notes
        no writing code
        some reading code

I mentioned telnet in the writeup
    -> use netcat (nc) instead

    $ nc <hostname> <port>

    stuff you type in gets sent to remote end, once you hit return

Sample code note

    If you are specifying -std=c99 or -std=c89,
        you need to explicitly enable getaddrinfo() using a
        feature test macro to get POSIX 2001+

    Put this at the top of your file (before any #include)

    #define _POSIX_C_SOURCE 200112L

    or

    #define _POSIX_C_SOURCE 200809L

    if you want to use strdup()

    Or just don't specify a standard or use -std=gnu99

    The sample code has been updated to include these macros

GNU is a project that makes open source software
    GCC is the GNU Compiler Collection
    most of the commands in a Linux environment are from GNU
    GNU = "GNU's Not Unix"

One other point:
    ilab.cs.rutgers.edu is a name for four different machines
    -> not a good choice for hosting a server, because you can't control
        which machine you connect to
    -> just use the command center/cave/meltdown clusters
    -> or specify ilab1.cs.rutgers.edu, etc.

Opening a connection:
    getaddrinfo() to get a list of addrinfo structs

    for each struct,
        create a socket using the provided fields
            ai_family, ai_socktype, ai_protocol
            socket()
        attempt to connect to the remote host/service
            ai_addr, ai_addrlen
            connect()
                - returns 0 for success

    if connect() succeeds, then we can use the socket as a file descriptor

        read()
        write()

    more powerful variants
        recv()
        send()

Note: TCP connections are "full duplex"
    two streams:
        client sends bytes to server
        server sends bytes to client

To open a socket and wait for incoming connection requests

```
char *port = ...;
struct addrinfo hints, *info_list, *info;
memset(&hints, 0, sizeof(struct addrinfo)); // set all bytes to 0
hints.ai_family = AF_UNSPEC;     // we want IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // we want a TCP connection
hints.ai_flags = AI_PASSIVE;     // we will want to listen

error = getaddrinfo(NULL, port, &hints, &info_list);
    // NULL because we want a port on this host
    // port specifies what port we want (e.g., "5050")
    // info_list will point to the head of the linked list of results

for each addrinfo struct
    listener = socket(info->ai_family, info->ai_socktype, info->ai_protocol);
        // create our listening socket

    error = bind(listener, info->ai_addr, info->ai_addrlen);
        // associates the socket with the specified port
        // will fail if port is unavailable or in use

    error = listen(listener, queue_length);
        // set up socket to accept incoming connections
        // queue_length is mostly arbitrary (e.g., using 5 or 8 is usually okay)

if we succeeded in binding and listening to the socket, we can wait
for incoming connection requests

    connection = accept(listener, NULL, NULL);
        // blocks until a remote host (client) tries to connect to our port (using TCP)
        // connection is a new file descriptor/socket
        //    it is specific to this connection

    once we have accepted a connection, we use read() and write() and
        eventually close()

    to get the next incoming connection request, we have to call accept() again


read() and write() work with sockets similarly to how they work with files
-> you need to be a little more careful about blocking

    recall:
        we give read a buffer and a requested (maximum) number of bytes

            bytes = read(connection, buffer, BUFFER_SIZE);

        bytes will contain the actual number of bytes read from the socket
        or  0 if the socket has closed
        or -1 if something went wrong

    read blocks if no data is currently available

    the network stack maintains a buffer of bytes that have arrived but
    have not been read

    when we call read, we get as much data as is available (up to our maximum)
        if we request 10 bytes, but only 2 bytes have arrived,
        read gives us two bytes

    if no bytes are available, read blocks until data arrives or the connection
        is closed (by the remote host)

NOTE WELL:
    TCP gives us a stream of bytes, not messages
    there is no guarantee that I will get a complete message in a single read
    there is no guarantee that a single read will contain only one message

From TCP's perspective, a session with our client is two uninterrupted streams

stream from client to server is
    "GET\n3\nday\nSET\n11\nday\nSunday\nGET\n6\na\nb c\n"

challenge:
    server and client need to respond to messages as they arrive
    * server won't send response until it has a complete request
    * client may not send next request until it gets a response

if we are not careful, the client or server may block on a call to read()
while the other party is waiting for more information

This is why the Project III protocol specifies the message length and
includes an end-of-message sequence (the final \n)

    -> server must not try to read past the end of a message
        -> may wait forever, because client might not send more data until
           it gets a response


You can use read()/write() for your server

or, we can use C's formatted I/O

FILE *fp = fdopen(file_descriptor, mode);
    // fdopen() creates a FILE struct for an existing file descriptor
    //    works for files, pipes, sockets, etc.
    // fp refers to the same file as file_descriptor
    // now we use fprintf(), fscanf(), getc(), fputc(), etc.
    // fclose() will close the underlying file descriptor

but, we need to read and write this socket
    using files in read-write mode is tricky

solution: use dup() to create a second file descriptor for the socket

FILE *fin = fdopen(dup(connection), "r");  // copy socket & open in read mode
FILE *fout = fdopen(connection "w")  // open in write mode

We can use getc() to get individual bytes from the socket

    int c = getc(fin);
    if (c == EOF) ....

We can use fscanf() to read and parse integers

    fields = fscanf(fin, "%d", &len);
    if (fields != 1) ...

We can use fprintf() to write to our socket

    fprintf(fout, "GET\n%d\n%s\n", strlen(key)+1, key);


When we are done, we fclose() the FILEs

    fclose(fin);
    fclose(fout);



If you don't want to mess around with fdopen(),
    it is also okay to just call read() and request 1 byte at a time

    bytes = read(connection, &some_char, 1);
```