```
CS 214 / 2021-04-28
===================

In POSIX C, we have two families of IO operations

Posix operations / syscalls / non-buffered IO

    read() and write() (also send() and recv())

    - low-level calls
    - provide the same interface for files, sockets, pipes, etc
    - precise control
        - the data I send using write() is sent to the OS / no longer my responsibility
        - when I receive data using read(), I can set a specific maximum

    -> no built-in formatted IO
        printing a decimal integer requires allocating a local buffer
        and either writing a function or using sprintf()

C operations / buffered IO

    fread(), fwrite(), getc(), putc()
    fprintf(), fscanf()

    these maintain a local buffer
        data written using putc() and fprintf() may not get sent immediately
            to the OS

        when we call getc() or fscanf(), the library may prefetch data and
            store it in the buffer

        -> less control about when data is sent and received
        -> potentially fewer system calls (= better performance)
            -> calling getc() in a loop is far more efficient than calling
                read(fd, &ch, 1)

    If you want to use buffered IO with a socket, you will probably want
        separate read and write buffers

    FILE *fp = fdopen(socket, "r+")
        // create a FILE for a socket in read/write mode
        // only has one buffer, shared by reading and writing operations
            // after we write, we must fflush() before reading
            // after we read, must reset the file pointer before writing

    Instead, when using buffered IO on a socket, duplicate the socket and
        create two separate FILEs

    int sock2 = dup(sock);
    // should confirm sock2 != -1
    FILE *fin = fdopen(sock, "r");
    FILE *fout = fdopen(sock2, "w");
    // should confirm fin != NULL and fout != NULL

    when you call fclose(), it calls close() on the file descriptor

        fclose(fin);
        fclose(fout);

    fscanf(fin, "%d" &len);
        // reads bytes until the first non-digit

    Note: sockets do not have all the features of files
        -> we can't skip around; no way to skip forward or back
        -> we don't necessarily know how many bytes we have written

    recall: writing using buffered IO does not necessarily send data to the OS
        immediately

        fflush(fout);   // sends anything in the buffer to the OS

    fprintf(fout, "OKG\n%d\n%s\n", strlen(value)+1, value);
        // puts data into the buffer
    fflush(fout);
        // sends buffer contents to OS -> to client


Sockets do not behave exactly like files
    read() with files usually gives us all the data we want
    read() with sockets gives us all the data currently available
        -> may be less than we asked for
        -> we have no control over how quickly data arrives from the other
            party
        -> if the connection is bad, we could get data 1-byte at a time!

    write() with files pretty much always writes all the bytes given
    write() with sockets usually does, but might not
        -> other party may have closed the connection
        -> we could have been interrupted by a signal

    -> for maximum safety, any time we call write() we need to check how
        much was actually written, and possibly call write() again to rewrite
        the remainder


    char *buf;
    int buflen, bytes, written = 0;

    while (written < buflen) {
        bytes = write(fd, buf + written, buflen - written);
        if (bytes < 1) { some sort of error handling }

        written += bytes;
    }

    -> we don't usually need to do this, because write() usually writes
        everything
    -> we should still check the return value in case we were interrupted
        by a signal, or the connection closed/file became unwritable


Fuzzy thinking about sockets
* our protocol is described in terms of messages
    clients sends a request
    server sends a response

* the TCP model gives us two streams of bytes
    TCP does not guarantee that we will get a whole message at once
    TCP does not guarantee a break between messages

When we read, we don't want to read too much
    - if the client sent another request without waiting for our response,
        we could read part of the second message
        -> now we have to hold onto it until we finish dealing with the first
        one

    - if we ask for more bytes than the client sent, but the client is waiting
        for our response before it sends any more, then we deadlock
            - server is waiting for the client
            - client is waiting for server
        - buggy client sends message that is too short
        - buggy server asks for more bytes than it should

        - we defend against server bugs by writing good code
        - we defend against short messages from the client by looking for the
            terminating newline

            GET\n5\n\day\n

            terminating \n comes after 4 characters, not 5
            server should immediately detect this, and not try to read further
                -> send ERR\nLEN\n and close connection

            GET\n3\day\n

            server should not read past the 'y'
                -> send ERR\nLEN\n and close connection


            GET\n4\nday

            no terminating \n
                -> maybe not even a problem; client could still send the \n
                    later
                -> defending against this requires more advanced features
                -> we won't consider this scenario in this class
                -> you can't fix this with the tools I have given you,
                    so don't try

Note about netcat (nc)
    - we can use nc to pretend to be a client for Project III
    - nc is one possible interpretation of the protocol
        - nc may not interleave input and output precisely
        - if data arrives while nc is waiting for input, it may not show us
            the data immediately

    - if you want to know the exact sequence of events, write a client


Multithreading in servers
-------------------------


echos.c, and your Project III, start a thread for each connection request

loop {

    struct arg_t args = malloc(sizeof(struct arg_t));

    connection_fd = accept(listening_fd, NULL, NULL);
        // block until a remote host tries to connect
    if (connection_fd < 0) ...

    args->fd = connection_fd;


    err = pthread_create(&worker_id, NULL, worker_function, args);
        // worker_fun will eventually close connection_fd and free args
    if (err) ...

    pthread_detach(worker_id);
        // thread will clean up after itself; no return value
        // the main thread doesn't need to remember how many threads are
        // running, and does not need to join them
}


As written, the only way to stop this server is to terminate it with
    SIGINT, SIGTERM, SIGKILL, etc
    -> server shuts down immediately, closes all sockets, terminates all
        threads

How can we make this cleaner?
- we can use a signal handler to catch SIGINT and/or SIGKILL, etc.
- many blocking system calls will return early if you receive a signal
    while you are blocked
    - they may also be set to auto-resume by default


One possible strategy
    - install a signal handler
    - have some flag that indicates whether the signal has been received
    - accept() will return -1 if it is interrupted by a signal
    - have the main loop exit if the signal was received


    while (running) {
        fd = accept(listener_fd, &addr, &addrlen);
        if (fd < 0) continue;

        .. start up thread to handle connection
    }

    // we don't get here until after the signal arrives

A few points to consider
- when we return from main(), the whole process stops and all threads are
    terminated
- if we call pthread_exit(), only that thread ends
    - if the main thread exits, the process won't terminate until every
        thread is finished

    - any exit handlers will get called after the last thread finishes
        e.g.,
        atexit(cleanup_data);

- signals are sent to any thread that is not blocking the signal
    - each thread has its own signal mask (= which signals are blocked)
    - all threads have the same signal dispositions (SIG_IGN, SIG_DEF, functions)
    - accept() will only break out of the block if the main thread gets the
        signal
    - we need to arrange things so that only the main thread receives the
        signals of interest
    - when a thread is created, it inherits its parent's signal mask
    - so:
        block SIGINT
        spawn child thread
        unblock SIGINT
```