Select for responsiveness

In this lab, you will use `select` to make a version of the echo server that can handle multiple simultaneous connections.

Let’s begin by verifying the problem. The code for the echo program is still available in the directory `/home/courses/cs226/echo`. Copy this code:

```bash
cp -R /home/courses/cs226/echo .
```

(The `-R` stands for recursive, meaning it will copy the entire directory.) Go into the `echo` directory. Compile the code with `make`; this is set to use the `-g` option to enable debugging. Then run the server in `gdb` (give a port number as the command line argument to the `r` command) and verify that it blocks on the call to `accept`. (Start it with a breakpoint on `main` (b main) and step through the code (using `n` to advance by a line) until it reaches this line.) Then use `telnet` to connect to it from two different windows (`telnet 127.0.0.1 port` assuming you’re on the same machine as the server). The program will unblock when you make the first connection. Keep letting the program run (using `n` as necessary) and interact with it a bit over the first `telnet` connection. Verify that it blocks on the `recv` call and that the second client receives no responses after the connection is established (which is done by the operating system without entering your code).

Now to look at `select`. The book’s example code is in the directory `/home/courses/cs226/echo-select`. This code uses `select` to multiplex between standard input and a list of sockets listening on ports specified on the command line. It uses `FD_ZERO` to zero out the array of flags and `FD_SET` to set the flags corresponding to the sockets it is interested in. `maxDescriptor` is the largest-numbered socket descriptor. Note that there are three sets of descriptors; these are being checked for readiness to read, readiness to write, and the presence of exceptions (errors), respectively. The sample code (and our application) just uses the first set and leaves the others `NULL`. Once `select` returns, the program knows that one of the sockets is ready to read and identifies this socket by checking each with `FD_ISSET`. Refer to the text, online sources, or ask me if there are other aspects of the code that you do not understand.

Finally, you’re ready to put these programs together. The goal is to create a program that can handle multiple simultaneous connections, behaving like the echo server with respect to each one. You will need to keep a list (using an array is fine) of the currently active sockets. You’ll have one “server socket” on which you’re waiting for new clients to connect. The other sockets will be the ones that you get from `accept`, i.e. one for each client connected to the server. The basic structure of the program is that you’ll block until at least one of these sockets is ready. Whenever the server socket is ready, you’ll accept a new connection and add it to the list. Whenever a connection socket is ready, you’ll read data from it and echo the data back. (You may also discover that the connection is closed, in which case you remove that socket from the list of active connections.) Use `NULL` as the argument for the timeout amount; this will cause `select` to wait until one of the sockets is ready. See yesterday’s slides on the course webpage (under Lectures) for pseudocode for this program.