ECE 435 – Network Engineering
Lecture 2

Vince Weaver

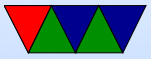
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

24 January 2025

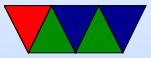
Announcements

- Homework 1 will be posted.
  Will be on website, will announce via mainestreet e-mail
  Due next Friday (via e-mail)



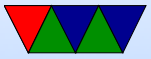
Client and Server

- Client: makes requests
- Server: listens for requests, and responds
- Can you be both?



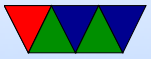
Homework #1

- Write a client and a server
- Server waits for incoming network connection.
  When one comes in it is opened and it listens for text.
  It takes that text back, interprets it, sends a response.
- Client opens a connection to server. Takes input from the keyboard and sends it to server, waits for response, and prints response.
- How would you code this up?



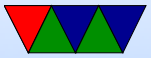
Homework #1 – Code Notes

- I provide a lot of the code for you as writing socket code completely from scratch is a huge pain
- If you took ECE471 this might seem straightforward
- I might not be able to cover all this before the assignment is assigned. We should be able to cover it all by Monday
- Sorry this involves throwing a lot of C at you right at the start of the semester



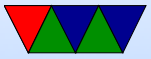
Homework #1 – Hardware Notes

- Assume you have a Linux machine
- Can also do this on OSX if you have compiler/etc installed
- Also in windows, maybe if you install the new Linux subsystem for it? Or run Linux in a VM?
- If you can't do any of those things, I can provide an account you can ssh into to do the homework.



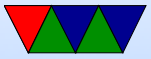
Homework #1 – Something Cool

- Last point is for something cool
- As described, do this in a separate copy of the code to make grading easier



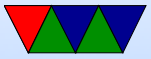
Socket Programming

- BSD sockets – Berkeley UNIX, 1983
  Why the standard? Right place at right time, also "free" and open-source
- Sort of at the transport layer, we are skipping ahead here
- Will reuse the code throughout the semester
- Can use for things other than TCP/IP (AF_UNIX, netlink, bluetooth, IPX, appletalk, etc)



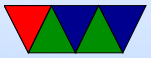
Low level C programming

- Why C code?
  - Close to hardware.
  - Always know what's going on.
  - Performance.
  - I like it.
- Why not C-code?
  - Hard to code
  - **Security**



Other Languages

- Python
  - Low-level interface a lot like C one
  - Higher level sockserver interface
- Java
  - More abstraction
  - java.net, socket=newsocket(addr,port);
- Rust
  - std::net



Small C Program

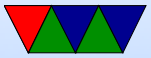
What do all the parts do?

argc/argv handle command line arguments.

what are sycalls?

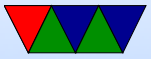
How does printf work?

```c
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello world\n");
    return 0;
}
```



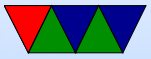
File descriptors and system calls

- At the lowest level, everything on UNIX/Linux is a "file" (or is supposed to be)
- Files are tracked per-process, with an integer value *file descriptor* acting as a sort of reference.
- Your process starts out with three open files, STDIN (0), STDOUT (1), STDERR (2)
- You can create more file descriptors with various system calls. `open()` is a common one. Returns -1 on error.



More File descriptors and system calls

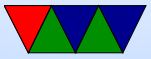
- Once you have a file descriptor, use syscalls such as `read()`, `write()`, `ioctl()` to do I/O
- You can `close()` when you are done
- Magic of Linux/UNIX is not just disk files, but all devices act as files and same syscalls work on them.
- Just to be difficult though the socket interface does things slightly differently (you don't use `open()` on `/dev/network`, some people are still angry about this



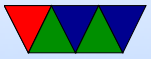
Socket Syscalls

remember: use `man` for documents, e.g. `man socket`

- SOCKET – create a new endpoint
- BIND – associate an address with a socket
- LISTEN – announce willing to accept connections
- ACCEPT – passively establish incoming connection
- CONNECT – actively attempt to establish connection
- SEND – send data
- RECEIVE – receive data
- CLOSE – close connection

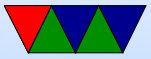


Writing a Simple Server



Opening a socket for listening

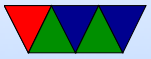
```
/* Open a socket to listen on */
/* AF_INET means an IPv4 connection (others are possible) */
/* SOCK_STREAM means reliable two-way connection (TCP) */
/* last argument is protocol subset.  We leave at zero */

int socket_fd = socket(AF_INET, SOCK_STREAM, 0);

if (socket_fd<0) {
   fprintf(stderr,"Error opening socket! %s\n",
      strerror(errno));
}
```



Setting an Address

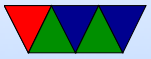
- More layer violations
- While in theory generic, we are coding to TCP/IP here
- Address is a 32-bit number that uniquely identifies system

  IP Address, often written 127.0.0.1 but it's actually just a 32-bit integer



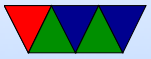
Setting the Port

- Port is how you handle multiple applications on same machine, based on the "port" it can map back to which application (the OS has a table)
- On TCP/IP limited to a 16-bit port number (65536)



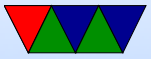
Network Byte Order

- The address and port are in network byte order, which is big-endian (stored biggest byte first)
- Most modern machines are little-endian (stored smallest byte first)
- You will need to convert the address and port to the proper endianess
- Aside/example on endianess



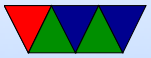
Network Byte Order Conversion

- `htonl()` will convert a long (32-bit)
  host **to** **n**etwork order **l**ong
- `htons()` will convert a short (16-bit)
  host **to** **n**etwork order **s**hort
- Can you convert the other way? Yes, there's also `ntohl()` and `ntohs()`

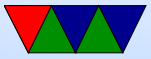


Setting up Address

- `memset()` to clear memory to zero, be sure to get order of arguments right!
- C structures and how they work
- Casting, lets us fake pointer type for all types of connections and cast to right one.
- We're a server and listening for any address
- We use 0.0.0.0 which means to listen on all networks
- No need to explicitly set 0.0.0.0, the zeroing by `memset` does it for us



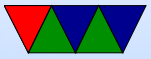
```
/* for reference, these live in header file */
/* /usr/include/x86_64-linux-gnu/sys/socket.h etc */
struct in_addr { uint32_t s_addr; };
struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t   sin_port;
    struct in_addr sin_addr;
};
/* Set up the server address to listen on */
struct sockaddr_in server_addr;
/* Clear struct, also sets the address to 0.0.0.0 */
memset(&server_addr,0,sizeof(struct sockaddr_in));
server_addr.sin_family=AF_INET;
/* Convert the port we want to network byte order (short) */
server_addr.sin_port=htons(port);
```



`bind()` system call

- `bind()` gives the socket an address, in this case 0.0.0.0 from above

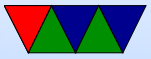
```
/* Bind to the port */

if (bind(socket_fd, (struct sockaddr *) &server_addr,
    sizeof(server_addr)) <0) {
    fprintf(stderr,"Error binding! %s\n", strerror(errno));
}
```



`listen()` system call

Sets up a data structure to hold pending incoming connections in case more than one come in at once.

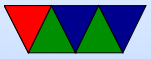
```
/* Tell the server we want to listen on the port */
/* Second argument is backlog, how many pending connections can */
/* build up */

listen(socket_fd,5);
```



`accept()` system call

- Blocks waiting for incoming connection
- When comes in, gets *new* file descriptor (careful)
- You can take this and fork a new thread to handle it (why?)

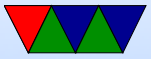
```
/* Call accept to create a new file descriptor for an incoming */
/* connection.  It takes the oldest one off the queue */
/* We're blocking so it waits here until a connection happens */
client_len=sizeof(client_addr);
new_socket_fd = accept(socket_fd,
    (struct sockaddr *)&client_addr,&client_len);
if (new_socket_fd<0) {
    fprintf(stderr,"Error accepting! %s\n",strerror(errno));
}
```



read() system call

Can also use `recv()` if need extra options.

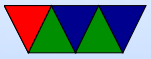
```
#define BUFFER_SIZE 1024
char buffer[BUFFER_SIZE];

/* Someone connected!  Let's try to read BUFFER_SIZE-1 bytes */
memset(buffer,0,BUFFER_SIZE);
n = read(new_socket_fd,buffer,(BUFFER_SIZE-1));
if (n==0) fprintf(stderr,"Connection to client lost\n\n");
else if (n<0) {
   fprintf(stderr,"Error reading from socket %s\n",
      strerror(errno));
}

/* Print the message we received */
printf("Message from client: %s\n",buffer);
```



write() system call

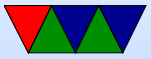
Can also use `send()` if need extra options.

```
/* Send a response */
n = write(new_socket_fd,"Got your message, thanks!",25);
if (n<0) {
                fprintf(stderr,"Error writing. %s\n",
                        strerror(errno));
}
```



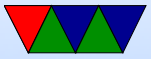
close() system call

```
printf("Exiting server\n\n");

/* Close the sockets */
close(new_socket_fd);
close(socket_fd);
```

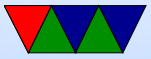


Server Summary

- socket() – tell kernel to create socket
- set up address/port
- bind() – assign address to socket
- listen() – start listening on socket
- accept() – wait for incoming connection, assign file descriptor
- recv()/read() – get data
- send()/write() – send response
- close() – close connection

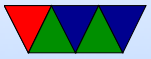


- close() – close socket



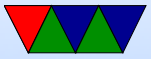
HW#1 – Keeping Server Open

- I provide code that does a simple, once-through server transaction
- What if you want to keep the server open and listening for multiple transactions of same connection?
- You will need to loop. Where should we loop to?
- After write, loop back to just before the read. Don't loop to before the accept or else you'll just continually start new connections, not re-use the current

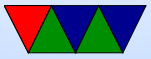


Server Handling Multiple Connections

- High-end servers (like webservers) can handle multiple active connections at once.
- You can use accept to get file descriptors for multiple connections
- How can you handle this? Rapidly query each fd over and over to see if any data has come in? Inefficient.
- A few ways to handle
  - poll()/select() let you set up an array of fds and get notified if any see activity



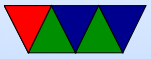
- alternately, fork() or spawn a thread for each fd and handle separately



TIME_WAIT

- If you quit and immediately try to restart server might get error saying socket busy.
- Spec says you should wait a minute for all packets to clear out. You can wait, or can force with

```
int on=1;    /* we want to turn the feature on */
setsockopt(s,SOL_SOCKET,SO_REUSEADDR,
   (char *)&on,sizeof(on));
```



Client Code

- Ran out of time, see Monday's lecture notes