ECE 435 – Network Engineering
Lecture 2

Vince Weaver

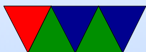
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

28 January 2021

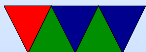
Announcements

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



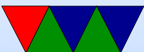
Homework #1

- 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?



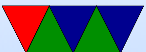
Homework #1 – Hardware Notes

- 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.



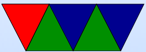
Socket Programming

- BSD sockets – Berkeley UNIX, 1983
- Sort of at the transport layer, we are skipping ahead here
- We'll use these for Homework #1
- Will reuse the code throughout the semester



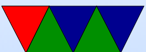
Client and Server

- Can you be both?



Low level C programming

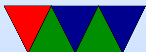
- Why C code?
  - Close to hardware.
  - Always know what's going on.
  - Performance.
  - I like it.
- Why not C-code?
  - Hard to code
  - **Security**



Small C Program

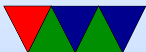
What do all the parts do?
argc/argv handle command line arguments.
what are sycalls?
How does printf work?

```
#include <stdio.h>

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



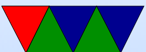
File descriptors and system calls

- 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

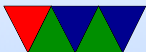
- Once you have a file descriptor, use syscalls such as `read()`, `write()`, `ioctl()` to do I/O
- You can `close()` when you are done
- Magic of Linux/UNIX is not just disk files, but all devices act as files and same syscalls work on them.
- Just to be difficult though the socket interface does things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



Socket Syscalls

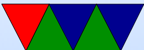
remember: for docs run `man socket` etc..

- 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



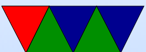
Opening a socket for listening

```
/* 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));
}
```



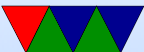
Address and Port

- More layer violations
- While in theory generic, we are coding to TCP/IP here
- Address is typically the global IP address
  can run on same machine with localhost
- 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)

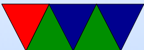


Setting up Address

- `htons()` has to do with endianess (network [big-endian] vs host [probably little])
- `memset()` to clear memory to zero, be sure to get order of arguments right!
- C structures and how they work
- Casting, lets use fake pointer type for all types of connections and cast to right one.



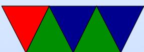
```c
/* for reference, these live in header file */
struct sockaddr_in {
   sa_family_t    sin_family;
   in_port_t   sin_port;
   struct in_addr sin_addr;
};
struct in_addr {
   uint32_t s_addr;
};

struct sockaddr_in server_addr;
/* Set up the server address to listen on */
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.
- Since we're a server and listening we don't have to give an address
- We use 0.0.0.0 (set by memset) which means to listen on all networks

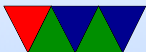
```
/* Bind to the port */
if (bind(socket_fd, (struct sockaddr *) &server_addr,
    sizeof(server_addr)) <0) {
    fprintf(stderr,"Error binding! %s\n", strerror(errno));
}
```



`listen()` system call

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

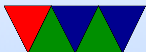
```
/* Tell the server we want to listen on the port */
/* Second argument is backlog, how many pending connections can */
/* build up */
listen(socket_fd,5);
```



accept() system call

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

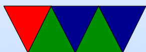
```
/* Call accept to create a new file descriptor for an incoming */
/* connection.  It takes the oldest one off the queue */
/* We're blocking so it waits here until a connection happens */
client_len=sizeof(client_addr);
new_socket_fd = accept(socket_fd,
   (struct sockaddr *)&client_addr,&client_len);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



read() system call

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

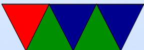
```
/* Someone connected!  Let's try to read BUFFER_SIZE-1 bytes */
memset(buffer,0,BUFFER_SIZE);
n = read(new_socket_fd,buffer,(BUFFER_SIZE-1));
if (n==0) {
   fprintf(stderr,"Connection to client lost\n\n");
}
else if (n<0) {
   fprintf(stderr,"Error reading from socket %s\n",
      strerror(errno));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

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



write() system call

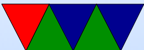
Can also use `send()` if need extra options.

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

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



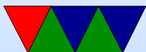
close() system call

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

         /* Try to avoid TIME_WAIT */
//       sleep(1);

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



Server Notes

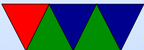
- What if you don't want to exit after, but instead loop?
- What happens if you have more than one incoming connection?
- poll() vs busy wait?
- What if you want to handle multiple connections at once?



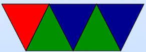
TIME_WAIT

- 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;
setsockopt(s,SOL_SOCKET,SO_REUSEADDR,
   (char *)&on,sizeof(on));
```

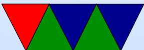


Client Code



socket() again

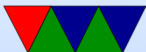
```
/* Open a socket file descriptor */
/* AF_INET means an IP network socket, not a local (AF_UNIX) one */
/* There are other types you can open too */
/* SOCK_STREAM means reliable two-way byte stream (TCP) */
/* last argument is protocol subset.  We leave at zero */
socket_fd = socket(AF_INET, SOCK_STREAM, 0);
if (socket_fd<0) {
   fprintf(stderr,"Error socket: %s\n",
      strerror(errno));
}
```



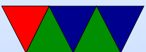
get host address / port

- Note for this example using "localhost"
- This is a special case, 127.0.0.1 on IPv4.
- Could put in a host name, this gets looked up via DNS. Or manually put in an IP address.

```
/* Look up the server info based on its name */
server=gethostbyname(DEFAULT_HOSTNAME);
if (server==NULL) {
   fprintf(stderr,"ERROR!  No such host!\n");
   exit(0);
}

/* clear out the server_addr structure and set some fields */
/* Set it to connect to the address and port of our server */
```

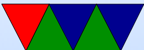


```
memset(&server_addr,0,sizeof(server_addr));
server_addr.sin_family=AF_INET;
memcpy(server->h_addr,&server_addr.sin_addr.s_addr,
   server->h_length);
/* port should be in "network byte order" (big-endian) so convert *
/* htons = host to network [byte order] short */
server_addr.sin_port=htons(port);
```



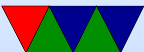
connect system call

```
/* Call the connect system call to actually connect to server */
if (connect(socket_fd,(struct sockaddr *) &server_addr,
   sizeof(server_addr)) < 0) {
   fprintf(stderr,"Error connecting! %s\n",
      strerror(errno));
}
```



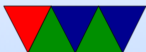
wait for response with read()

```
/* Prompt for a message */
printf("Please enter a message to send: ");
memset(buffer,0,BUFFER_SIZE);

/* Read message */
fgets(buffer,BUFFER_SIZE-1,stdin);

/* Write to socket using the "write" system call */
n = write(socket_fd,buffer,strlen(buffer));
if (n<0) {
   fprintf(stderr,"Error writing socket! %s\n",
      strerror(errno));
}
```



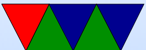
wait for response with read()

```
/* Clear buffer and read the response from the server */
memset(buffer,0,BUFFER_SIZE);
n = read(socket_fd,buffer,BUFFER_SIZE-1);
if (n<0) {
   fprintf(stderr,"Error reading socket! %s\n",
      strerror(errno));
}

/* Print the response we got */
printf("Received back from server: %s\n\n",buffer);
```



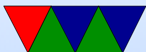
close again

```
/* All finished, close the socket/file descriptor */
close(socket_fd);
```

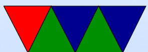


Notes on Homework

- Make the server loop forever until a string comes in.
- How do you loop forever?
- How do you compare with a string? Can you use ==
- Be careful with strcmp()
- You might even want to use strncmp()
- Comment your code!



- Try to fix all compiler warnings!



Other Languages

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