Week 13 - Wednesday

CS222

Last time

- What did we talk about last time?
- Networking

Questions?

Project 6

Quotes

Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer.

Edsger Dijkstra

OSI seven layer model

- Seven layers is a lot to remember
- Mnemonics have been developed to help

Application	All	All	A	Away
Presentation	Pros	People	Powered-Down	Pretzels
Session	Search	Seem	System	Salty
Transport	Top	To	Transmits	Throw
Network	Notch	Need	No	Not
Data Link	Donut	Data	Data	Dare
Physical	Places	Processing	Packets	Programmers

Netcat

- Netcat (**`nc`**) is a very useful tool for testing networking
- It allows you to interact with network communications through stdin and stdout
- You can run **`nc`** as either a client or a server

nc as a client

- We can run **nc** as a client, connecting to some waiting server:

```
nc google.com 80
```

- Then, we can type in a command that server is expecting

```
GET / HTTP/1.0
```

- We should see the webpage response from Google

`nc` as a server

- Alternatively, we can use **`nc`** as a server to see what a client does when it tries to connect
  - Which can be useful when trying to understand HTTP

```
nc -l 30000
```

- Now, we can type **`127.0.0.1:30000`** into the address bar of a web browser
  - **`127.0.0.1`** is a special IP address that means "the host I'm on"
  - **`30000`** is the port that **`nc`** is listening on (in this case)

nc as both!

- We can even use **nc** as both a client and a server just for the hell of it
- In one terminal, start **nc** as a server:

```
nc -l 50000
```

- In another terminal, connect **nc** as a client to that server:

```
nc 127.0.0.1 50000
```

- Now, send stuff back and forth!

Sockets

Includes

- There are a lot of includes you'll need to get your socket programming code working correctly
- You should always add the following:
    - #include <netinet/in.h>
    - #include <netdb.h>
    - #include <sys/socket.h>
    - #include <sys/types.h>
    - #include <arpa/inet.h>
    - #include <unistd.h>

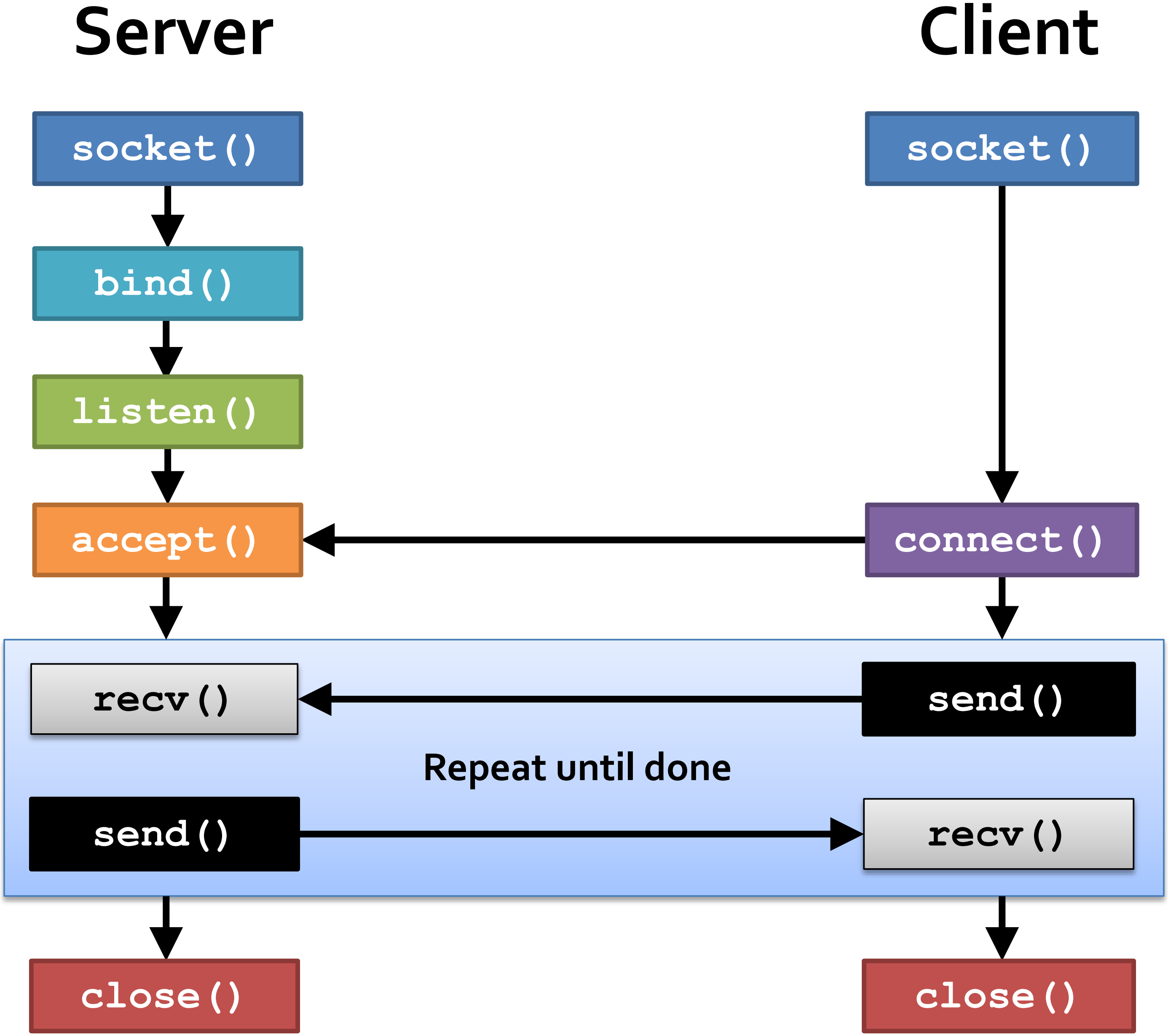
socket()

- If you want to create a socket, you can call the **socket()** function
- The function takes a communication domain
  - Will always be **AF_INET** for IPv4 Internet communication
- It takes a type
  - **SOCK_STREAM** usually means TCP
  - **SOCK_DGRAM** usually means UDP
- It takes a protocol
  - Which will always be **0** for us
- It returns a file descriptor (an **int**)

```
int sockFD = -1;
sockFD = socket(AF_INET, SOCK_STREAM, 0);
```

Now you've got a socket…

- What are you going to do with it?
- By themselves, they aren't useful
- You need to connect them together
- We're going to be interested in the following functions to work with sockets
  - `bind()`
  - `listen()`
  - `accept()`
  - `connect()`
- And also functions that are similar to the ones you know from low-level file I/O
  - `recv()`
  - `send()`
  - `close()`



Server

Client

socket()

bind()

listen()

accept()

socket()

connect()

recv()

send()

Repeat until done

send()

recv()

close()

close()

Making an address for a client

- We fill a **`sockaddr_in`** structure with
  - The communication domain
  - The correct endian port
  - The translated IP address
- We fill it with zeroes first, just in case

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = htons(80);
inet_pton(AF_INET, "173.194.43.0",
&(address.sin_addr));
```

Sending

- Once you've created your socket, set up your port and address, and called **connect()**, you can send data
  - Assuming there were no errors
  - Sending is very similar to writing to a file
- The **send()** function takes
  - The socket file descriptor
  - A pointer to the data you want to send
  - The number of bytes you want to send
  - Flags, which can be **0** for us
- It returns the number of bytes sent

```
char* message = "Flip mode is the squad!";
send(socketFD, message, strlen(message)+1, 0);
```

Receiving

- Or, once you're connected, you can also receive data
  - Receiving is very similar to reading from a file
- The **`recv()`** function takes
  - The socket file descriptor
  - A pointer to the data you want to receive
  - The size of your buffer
  - Flags, which can be **`0`** for us
- It returns the number of bytes received, or **`0`** if the connection is closed, or **`-1`** if there was an error

```
char message[100];
recv(socketFD, message, 100, 0);
```

Servers

- Sending and receiving are the same on servers, but setting up the socket is more complex
- Steps:
  1. Create a socket in the same way as a client
  2. Bind the socket to a port
  3. Set up the socket to listen for incoming connections
  4. Accept a connection

Bind

- Binding attaches a socket to a particular port at a particular IP address
  - You can give it a flag that automatically uses your local IP address, but it could be an issue if you have multiple IPs that refer to the same host
- Use the bind() function, which takes
  - A socket file descriptor
  - A sockaddr pointer (which will be a sockaddr_in pointer for us) giving the IP address and port
  - The length of the address

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = htons(80);
address.sin_addr.s_addr = INADDR_ANY;
bind(socketFD, (struct sockaddr*)&address,
sizeof(address));
```

Listening

- After a server has bound a socket to an IP address and a port, it can listen on that port for incoming connections
- To set up listening, call the **`listen()`** function
- It takes
  - A socket file descriptor
  - The size of the queue that can be waiting to connect
- You can have many computers waiting to connect and handle them one at a time
- For our purpose, a queue of size 1 often makes sense

```
listen( socketFD, 1);
```

Accept

- Listening only sets up the socket for listening
- To actually make a connection with a client, the server has to call **accept()**
- It is a blocking call, so the server will wait until a client tries to connect
- It takes
  - A socket file descriptor
  - A pointer to a **sockaddr** structure that will be filled in with the address of the person connecting to you
  - A pointer to the length of the structure
- It returns a file descriptor for the client socket
- We will usually use a **sockaddr_storage** structure

```
struct sockaddr_storage otherAddress;
socklen_t otherSize = sizeof(otherAddress);
int otherSocket;
otherSocket = accept( socketFD, (struct
sockaddr *) &otherAddress, &otherSize);
```

setsockopt()

- The **setsockopt()** function allows us to set a few options on a socket
- The only one we care about is the **SO_REUSEADDR** option
- If a server crashes, it will have to wait for a timeout (a minute or so) to reconnect on the same port unless this option is set
  - A dead socket is taking up the port
- When working on Project 6, it's a good idea to use this so you don't get stuck waiting around

```
int value = 1; //1 to turn on port reuse
setsockopt(socketFD, SOL_SOCKET,
SO_REUSEADDR, &value, sizeof(value));
```

`send()` and `recv()`

- Last time, we suggested that you use **`send()`** and **`recv()`** for writing to and reading from sockets
- You can actually use **`write()`** and **`read()`**
- The difference is that **`send()`** and **`recv()`** have an extra parameter for flags
  - We provided **`0`** as the argument (no value)
- These flags control how the **`send()`** or **`recv()`** acts

Flags

Flag	Meaning for `send()`	Meaning for `recv()`
`MSG_DONTWAIT`	Nonblocking send. If the buffer is full, return **`EAGAIN`**.	Nonblocking receive. If no message is available, return **`EAGAIN`**.
`MSG_OOB`	Send a single out of band (high priority) byte	Receive a single out of band (high priority) byte
`MSG_PEEK`	**Invalid flag**	Read the data from the buffer, but don't remove it
`MSG_WAITALL`		Keep reading until you have received the maximum bytes you can hold
`MSG_MORE`	A series of messages will be packed into a single TCP packet	**Invalid flag**
`MSG_NOSIGNAL`	A send on a closed socket will not generate a signal	

Why do we cast to sockaddr*?

- This is the basic **sockaddr** used by socket functions:

```
struct sockaddr {
    unsigned short sa_family; //address family
    char sa_data[14];  //14 bytes of address
};
```

- We often need **sockaddr_in**:

```
struct sockaddr_in {
    short               sin_family;   // AF_INET
    unsigned short      sin_port;     // e.g. htons(3490)
    struct in_addr      sin_addr;     // 4 bytes
    char                sin_zero[8];  // zero this
};
```

- They start with the same bytes for family, we can cast without a problem
  - C has no inheritance, we can't use a child class

Example 1

- Let's make a client and connect it to **`nc`** acting as a server
- We'll just print everything we get to the screen

Example 2

- Let's make a server and connect to it with **`nc`**
- We'll just print everything we get to the screen

Quiz

Upcoming

Next time…

- Function pointers
- Lab 13

Reminders

- Keep working on Project 6
  - It's tough!
- Read Section 5.11 of K&R for information on function pointers