Week 12 - Monday

COMP 2400

Last time

- What did we talk about last time?
- More networking
- Sockets

Questions?

Project 5

Quotes

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

Edsger Dijkstra

Sockets

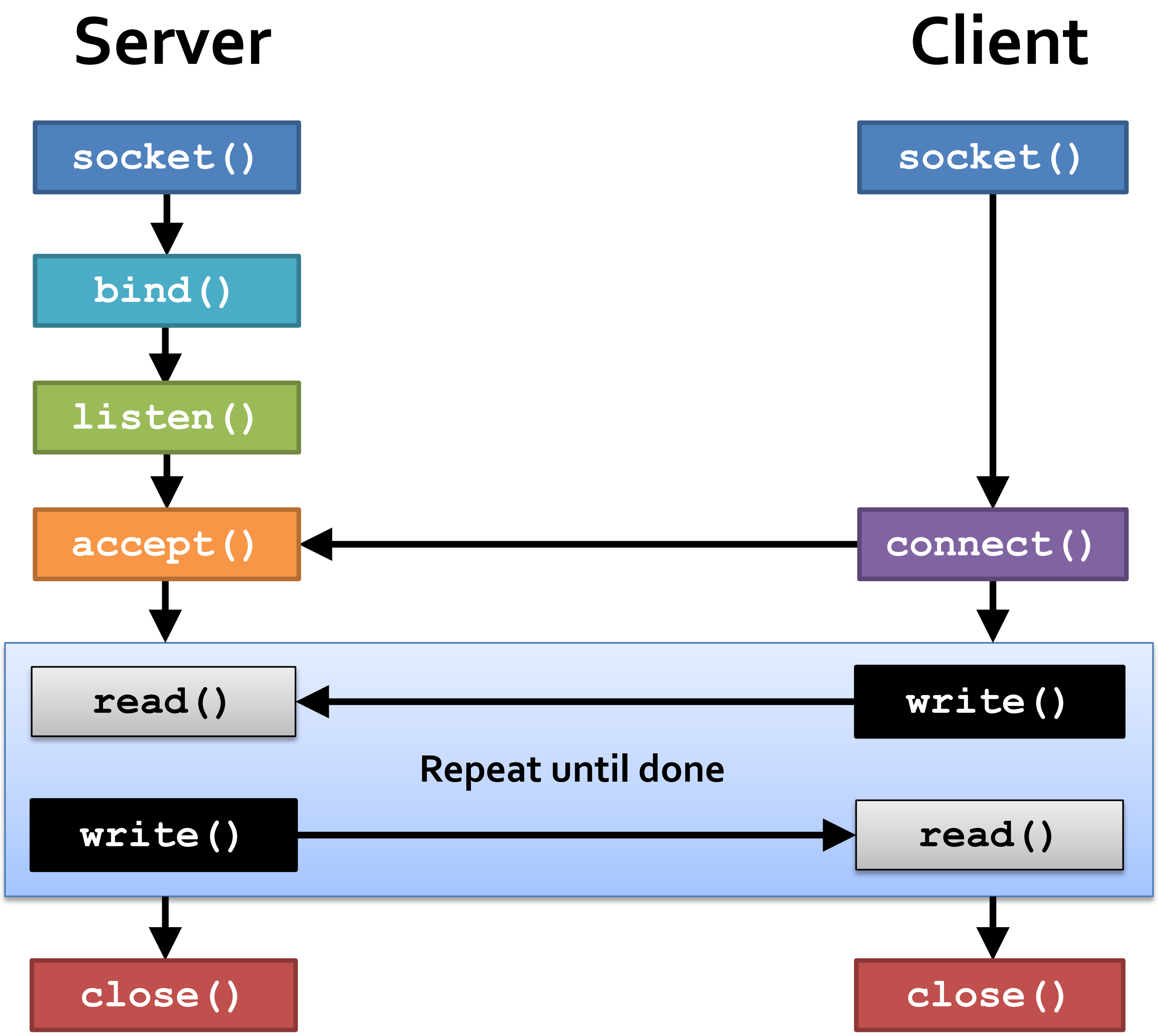
Includes

- 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);
```


Server
Client
socket()
socket()
bind()
listen()
accept()
connect()
read()
write()
Repeat until done
write()
read()
close()
close()

Client

- We'll start with the client, since the code is simpler
- Assuming that a server is waiting for us to connect to it, we can do so with the **`connect()`** function
- It takes
  - A socket file descriptor
  - A pointer to a **`sockaddr`** structure
  - The size of the **`sockaddr`** structure
- It returns -1 if it fails

```
connect(sockFD, (struct sockaddr *) &address,
    sizeof(address));
```

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 just like writing to a file
- The **`write()`** function takes
  - The socket file descriptor
  - A pointer to the data you want to send
  - The number of bytes you want to send
- It returns the number of bytes sent

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

Receiving

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

```
char message[100];
read(socketFD, message, 100);
```

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 = 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

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

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 read things and send them across the network

File Systems

Disks and partitions

- Until SSDs completely take over, many physical hard drives are electronically controlled spinning platters with magnetic coatings
  - Disks have circular **tracks** divided into **sectors** which contain **blocks**
  - A block is the smallest amount of information a disk can read or write at a time
- Physical disks are partitioned into logical disks
- Each partition is treated like a separate device in Linux
  - And a separate drive (**`C:`**, **`D:`**, **`E:`**, etc.) in Windows
  - Each partition can have its own file system

Popular file systems

- Linux supports a lot of file systems
  - ext2, the traditional Linux file system
  - Unix ones like the Minix, System V, and BSD file systems
  - Microsoft's FAT, FAT32, and NTFS file systems
  - The ISO 9660 CD-ROM file system
  - Apple's HFS
  - Network file systems, including Sun's widely used NFS
  - A range of journaling file systems, including ext3, ext4, Reiserfs, JFS, XFS, and Btrfs
  - And more!

Partition layout

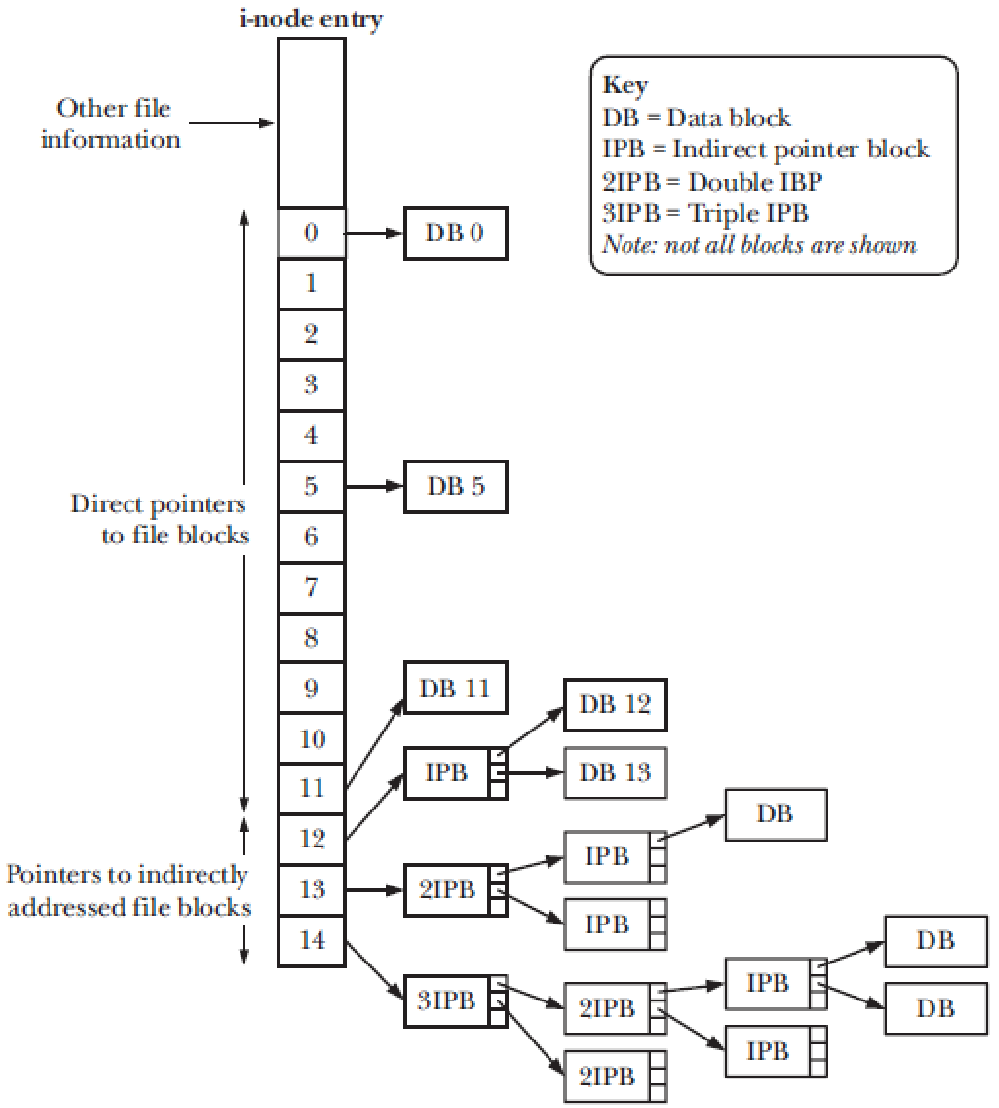
- Virtually all file systems have each partition laid out something like this

Boot block	Superblock	i-node Table	Data blocks

- The boot block is the first block and has information needed to boot the OS
- The superblock has information about the size of the i-node table and logical blocks
- The i-node table has entries for every file in the system
- Data blocks are the actual data in the files and take up almost all the space

i-nodes

- Every file has an i-node in the i-node table
- Each i-node has information about the file like type (directory or not), owner, group, permissions, and size
- More importantly, each i-node has pointers to the data blocks of the file on disk
- In ext2, i-nodes have 15 pointers
  - The first 12 point to blocks of data
  - The next points to a block of pointers to blocks of data
  - The next points to a block of pointers to pointers to blocks of data
  - The last points to a block of pointers to pointers to pointers to blocks of data



Upcoming

Next time…

- Function pointers

Reminders

- Finish Project 5
  - **Due Wednesday!**
- Read Section 5.11 of K&R for information on function pointers