

# Network Fundamentals

Benjamin Brewster

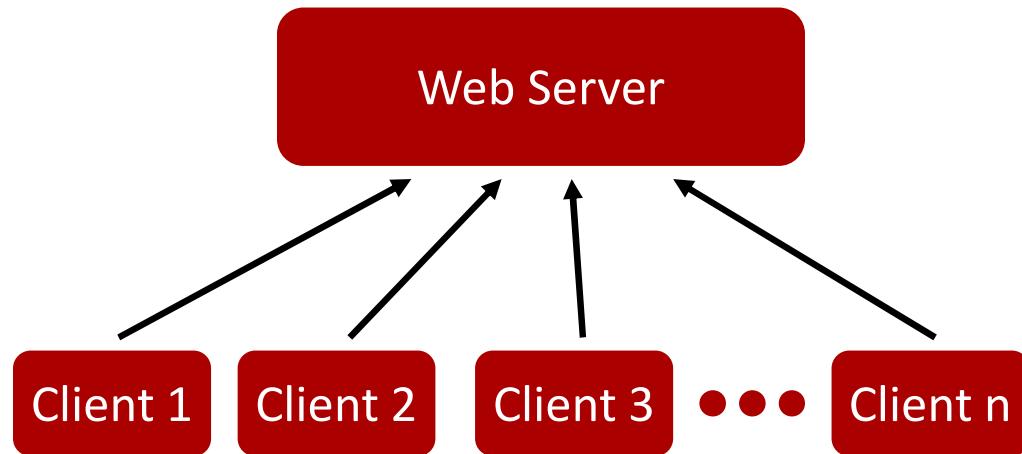
Except as noted, all images copyrighted with Creative Commons licenses,  
with attributions given whenever available

# IPC via Network Communication



- Terminology:
  - **Client/Server architecture** :: a networking arrangement such that one process (server) is continuously waiting for new connections from other processes (clients)
  - **Client** :: process that initiates the connection, requesting a service
  - **Server** :: process that is always running, waiting for new connections from client processes

# Client/Server Example: Web Server



- The web server is always running and looking for new connections
- Potentially unknown number of clients may connect at any time
  - Chrome, Edge, Safari, IE, Firefox, Opera, Lynx, etc.

# UNIX Daemons - a Type of Server

- In UNIX, a daemon is a process running in the background, ready to provide a service to the programs that need it
  - `syslogd` (system log daemon) maintains the system log
  - `lpd` (line printer daemon) manages print spooling
  - `ntpd` (network time protocol daemon) manages clock sync on a network
  - `dhcpd` (dynamic host control protocol daemon) assigns TCP/IP configuration data to network clients that request it
- In UNIX, daemons all have the `init` process (`pid == 1`) as their parent
  - As of 2016, many Linux distributions have replaced `init` with `systemd`



# Example Protocol: HTTP

- **HTTP** - Hyper Text Transfer Protocol
- Relatively simple: primarily used to support one feature, the requesting of a file to be downloaded
- Standardized protocol defining:
  - How client requests are formatted
  - How server responses are formatted
- The protocol is text-based, including all requests, responses, and errors



The world's most widely used web server is Apache. Don't confuse this software with the HTTP protocol that the server primarily deals with! Apache's UNIX daemon name is `httpd`

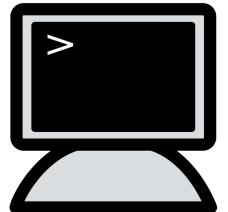
# Example Protocol: HTTP

- Simple request mode
  - Client connects to server then sends:
    - GET abc.html
  - Server receives and parses the GET command, then returns the file requested
- Enhanced request mode
  - Client connects to the server then sends:
    - GET index.html HTTP/1.1
  - Server responds with some header information, such as server type and version, then a blank line, and *then* the data file



# Text Protocol Debugging Tool

- telnet helps debug text-based protocols, but was originally designed for interacting with text-based protocol network sessions
- Almost without exception now, servers do not use telnet for shell access, as all text is passed in plain text - it can be read by every node in between the client and server
  - SSH is the current standard, which encrypts transmitted data
- You can pass telnet a second parameter that specifies the port you want to connect to



# Demo of telnet and HTTP

1. \$ telnet eecs.oregonstate.edu 80
2. <web server> GET / HTTP/1.1
3. <web server> Host: eecs.oregonstate.edu
4. <web server> (Enter again)

To connect to a web server,  
we typically use port 80



# Non-Text-Based Application Protocols

- Not all application protocols are text-based; TCP/IP has no problem transferring binary data!
- Advantages of text-based protocols
  - Easy to debug
  - Easy to communicate and understand (and teach!)
- Disadvantages of text-based protocols
  - Not very compact or efficient
  - Server can spend a lot of time just parsing text
    - Very important for text protocols to be simple!



# Network Layer Model - Application

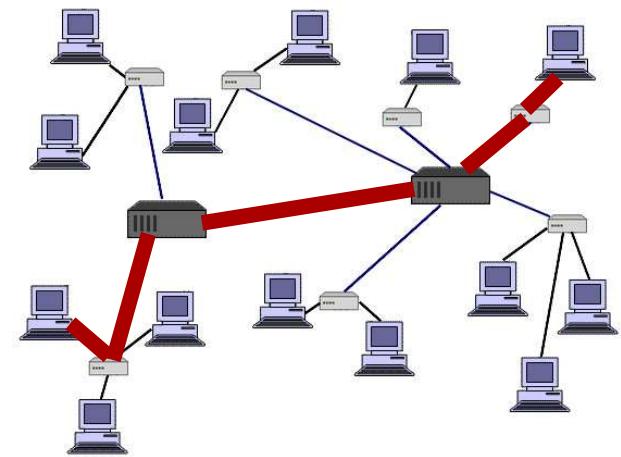
Layer	Common Protocols
Application	HTTP
Transport	TCP and UDP
Network	IP
Link	Ethernet, 802.11
Physical	Twisted pair copper, radio, fiber



- **Application:** Your software, utilizing the network
  - Agnostic to the underlying methods that make the data go from one host to the other
  - Web browsers, games, IM clients, video chat, email, etc.
  - Uses `send()` and `recv()`, for example

# Network Layer Model - Transport

Layer	Common Protocols
Application	HTTP
Transport	TCP and UDP
Network	IP
Link	Ethernet, 802.11
Physical	Twisted pair copper, radio, fiber



- **Transport:** Protocols that control how data is sent from one host all the way to the other, irrespective of the number of nodes, hops, or networks the data passes through
- TCP: Transmission Control Protocol - connection-oriented, guaranteed, in-order data transport
- UDP: Universal Datagram Protocol - connectionless, not guaranteed

# Network Layer Model - Network

192.168.1.1, for example

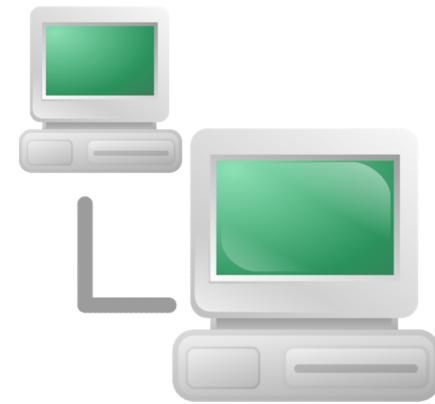
Layer	Common Protocols
Application	HTTP
Transport	TCP and UDP
<b>Network</b>	IP
Link	Ethernet, 802.11
Physical	Twisted pair copper, radio, fiber



- **Network:** Addressing and organization of a particular set of connected hosts (called a network), defines addressing between networks
- IP: Internet Protocol - naming, addressing and routing from host to host and across networks
- Still independent of physical connection type

# Network Layer Model - Link

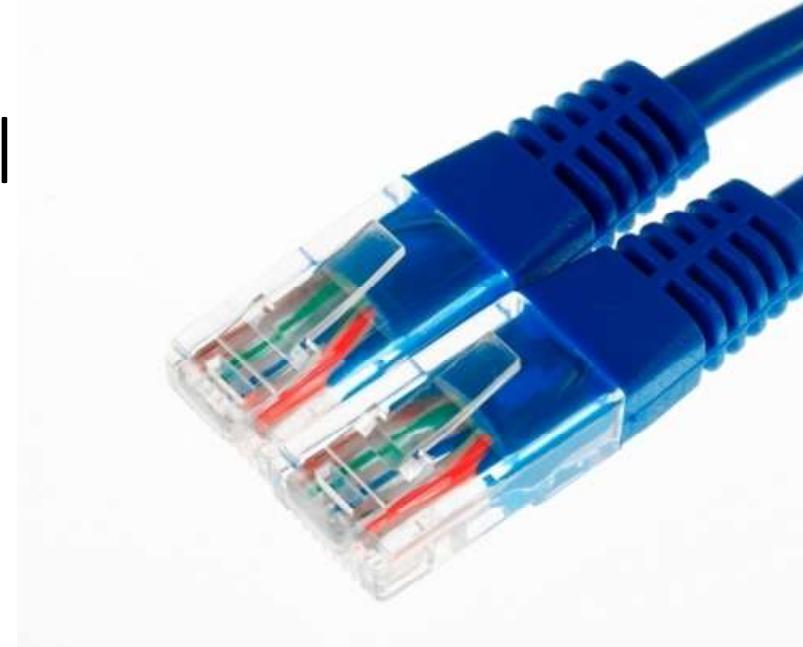
Layer	Common Protocols
Application	HTTP
Transport	TCP and UDP
Network	IP
<b>Link</b>	Ethernet, 802.11
Physical	Twisted pair copper, radio, fiber



- **Link:** Concerned with getting data from just one node to the next neighboring node; does no planning of routes
- Ethernet: The de facto addressing and signaling protocol currently in use in modern networks; uses Media Access Control addresses to communicate, together with IP
- 802.11: aka Wi-fi, the de facto protocol for wireless communication; controls sharing of the congested transmission space and connection speeds based on quality

# Network Layer Model - Physical

Layer	Common Protocols
Application	HTTP
Transport	TCP and UDP
Network	IP
Link	Ethernet, 802.11
<b>Physical</b>	Twisted pair copper, radio, fiber



- **Physical:** The actual hardware used to enable two hosts to talk
- Copper: Standard category 5 and 6 network cables (4 pairs of twisted copper strands) connect most of the world
- Radio: Enables wireless devices to communicate



# TCP/IP High-Level Functionality

- The combination of TCP and IP provides communication between two processes, potentially separated by a network
- TCP/IP stands for **Transmission Control Protocol / Internet Protocol**
- TCP is the protocol that your application interacts with, while IP provides the addressing system for routing network packets



# TCP Details

- TCP is the most commonly used protocol for transferring information across a network; second-most common Transport protocol is UDP
- Provides a byte stream interface (like stdio)
- Connection oriented - each side of the connection maintains resources to keep the connection open until it is explicitly closed
- A TCP connection is bi-directional - traffic can be sent across the connection in either direction
- Provides controls to slow down the sender if the nodes in the path to the receiver are burdened with traffic or otherwise lossy

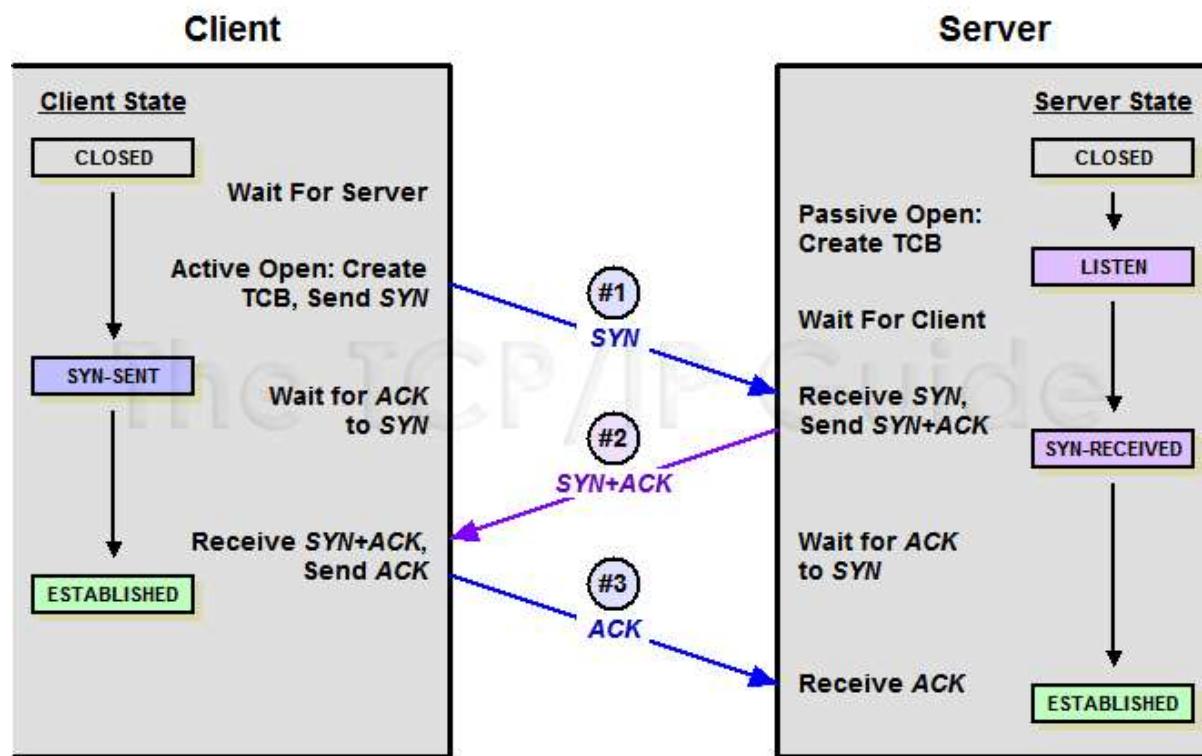


# Applications On Top of TCP/IP

- TCP/IP only passes bytes between processes - it does not interpret those bytes
- You write an "application protocol" on top of TCP/IP which defines what the bytes mean: this is what your program does with the bytes it sends & receives
- TCP/IP is like a phone connection
  - A phone transfers sound between two people
  - A phone does not interpret the meaning of the sound
- The telephone application protocol (Bob calling Alice):
  1. Bob calls Alice's phone
  2. Alice answers and says "Hello"
  3. Bob says "Hi" back, and then they both exchange information



# TCP Handshaking Starts the Connection



From [tcpipguide.com](http://tcpipguide.com), fetched 2/18/2015

# TCP Comparison with IP

- TCP sends out network traffic organized into bundles called packets, using the addresses specified and organized by IP
- Problem: IP does not guarantee
  - Data integrity
  - Packet order
  - Prevention of duplicates
  - Packet will actually arrive
- TCP can detect if the integrity of the stream of packets encounters issues and will take steps such as these to keep things running:
  - Re-order packets
  - Request packet re-transmission
  - Drop duplicate packets



# UDP - User Datagram Protocol

- Provides a very different interface:
  - Connectionless (no handshaking, etc.)
  - Data is broken into packets called datagrams
  - Server does not remember clients between datagrams
  - Datagrams may be dropped by the network
  - Datagrams may arrive out of order
- UDP has *much* less overhead than TCP, and is much faster to transfer data
- When to use UDP
  - Streaming video/audio
  - Mass broadcasting
  - Asynchronous communication like that used in games



# Internet Protocol Details

- IP specifies:
  - How we address machines on the network
  - If the machine we are addressing is not on the same local network, IP dictates how the data can be routed to it
- Each network interface (network card) has an **IP address**, which must be unique on the network
- IP(v4) address are 32 bit (binary) numbers, but are usually *represented* as four one-byte decimal numbers, separated by periods:

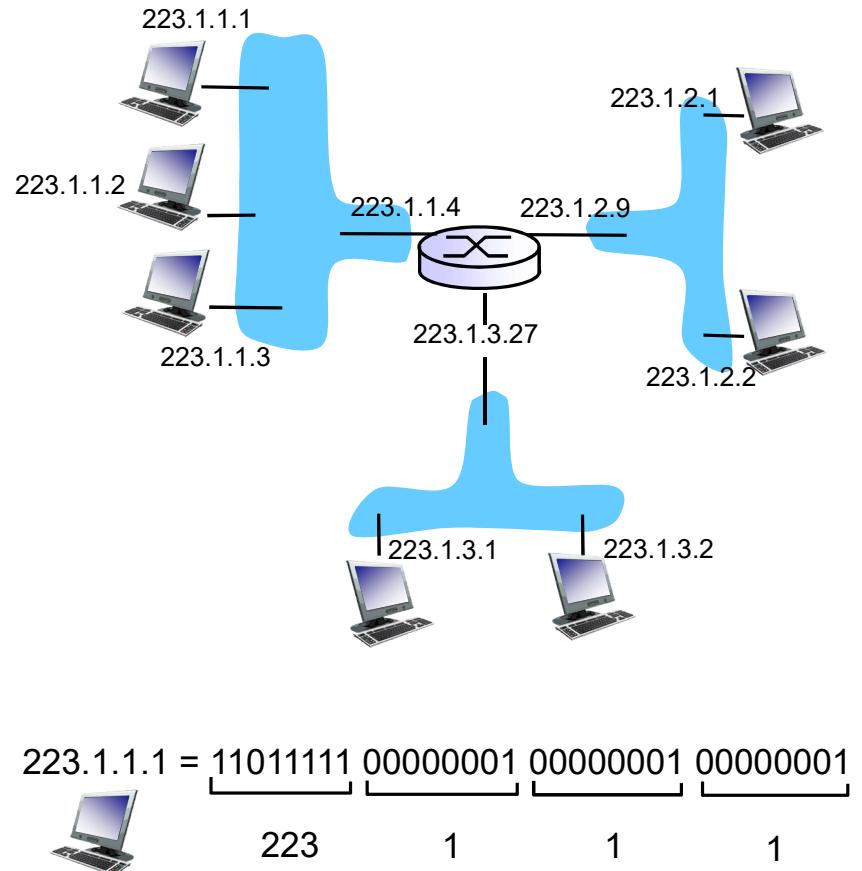
223.1.1.1 = 11011111 00000001 00000001 00000001



223      1      1      1

# IP Addressing: Introduction

- **Interface:** connection between host/router and physical link
  - Routers have multiple interfaces
  - A host typically has one or maybe two interfaces  
(e.g., wired Ethernet, wireless 802.11)
- **Each interface has its own IP address;** thus devices with more than one interface control multiple addresses



## IP 4 vs. 6



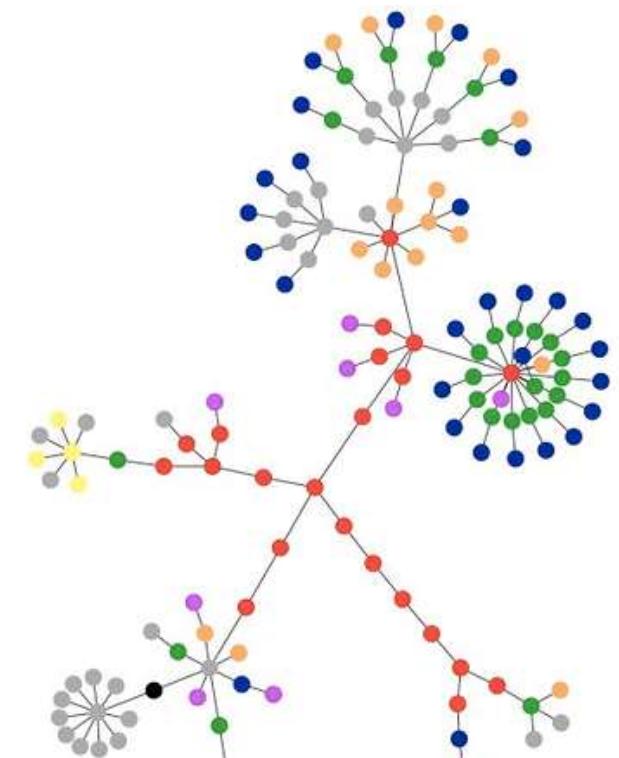
- IP version 4 (IPv4) uses 32-bit addresses, and therefore has  $2^{32} = 4,294,967,296 = 4 * 10^9$  unique addresses
- This is not enough for the world; in my house alone:
  - Xbox One, Xbox 360, five computers, 3 cell phones, amplifier, Blu-ray player, printer
- A lot of this can be handled by routers, which creates your own private LAN with addresses hidden from the outside networks, but this only shines light on the problem
- There are no more IPv4 addresses to give out for the Internet!
- Internet of Things (IoT), always-on connections, and inefficient address allocation all contributed to usage

## Companies/orgs with IPv4 /8 blocks from Internet Assigned Numbers Authority (IANA, a dept. of ICANN)

Owner	/8 Blocks	~IP addresses
US Military (Department of Defense etc.)	12	201 million
Level 3 Communications, Inc.	2	33 million
Hewlett-Packard	2	33 million
AT&T Bell Laboratories (Alcatel-Lucent)	1	16 million
AT&T Global Network Services	1	16 million
Bell-Northern Research (Nortel Networks)	1	16 million
Amateur Radio Digital Communications	1	16 million
Apple Computer Inc.	1	16 million
Cap Debis CCS (Mercedes-Benz)	1	16 million
Computer Sciences Corporation	1	16 million
Department of Social Security of UK	1	16 million
E.I. duPont de Nemours and Co., Inc.	1	16 million
Eli Lily and Company	1	16 million
Ford Motor Company	1	16 million
General Electric Company	1	16 million
Halliburton Company	1	16 million
IBM	1	16 million
Interop Show Network	1	16 million
Merck and Co., Inc.	1	16 million
MERIT Computer Network	1	16 million
Massachusetts Institute of Technology	1	16 million
Performance Systems International (Cogent)	1	16 million
Prudential Equity Group, LLC	1	16 million
Société Internationale De Telecommunications Aero.	1	16 million
U.S. Postal Service	1	16 million
UK Ministry of Defence	1	16 million
Xerox Corporation	1	16 million

<http://royal.pingdom.com/2008/02/13/where-did-all-the-ip-numbers-go-the-us-department-of-defense-has-them/>

# Who's Got Those Addresses?



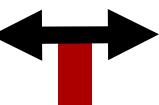
## IP 4 vs. 6

- IPv6 uses 128-bit addresses
  - $2^{128} > 2^{32}$
  - $3.4 * 10^{38} > 4.2 * 10^9$
  - This is **50 octillion** addresses for **each** of 6.5 billion people on earth

# 50 octillion addys\*:

50,000,000,000,000,000,000,000,000,000

\*Theoretically: a lot of them are reserved for special purposes, so the number is lower



# Back to Just Our Process

- A process can use just one interface to communicate with itself and other processes on the same machine
- Address network transmissions to your interface's own IP address, or the special hostname "localhost"
- Indicate which process you're talking to by specifying the "port" (more on this next lecture)



# Network Clients

Benjamin Brewster

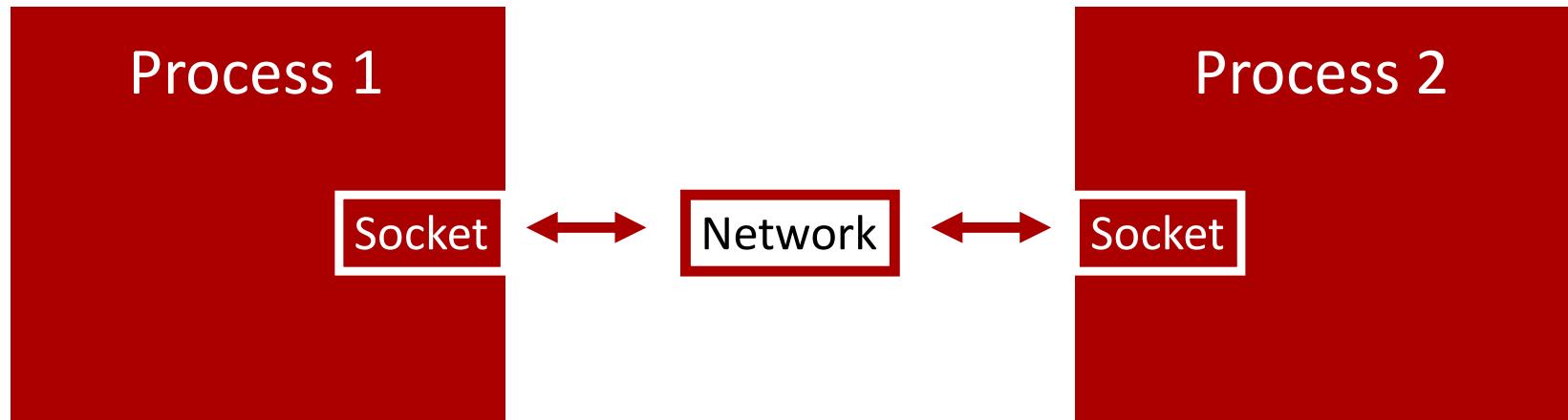
Except as noted, all images copyrighted with Creative Commons licenses,  
with attributions given whenever available

# Berkeley Sockets

- Developed in the early 1980s for BSD Unix under a grant from DARPA
- The *de facto* network communication method across local area networks (LAN) and the internet.
  - Other transmission methods exist, but they require different transport protocols.
    - i.e., you'd have to write your own version of TCP for a different network protocol
      - Seriously, don't do this. Just use sockets
        - Decades of research! Thousands of scientists, academics, engineers, and hobbyists!
        - Think of the children!

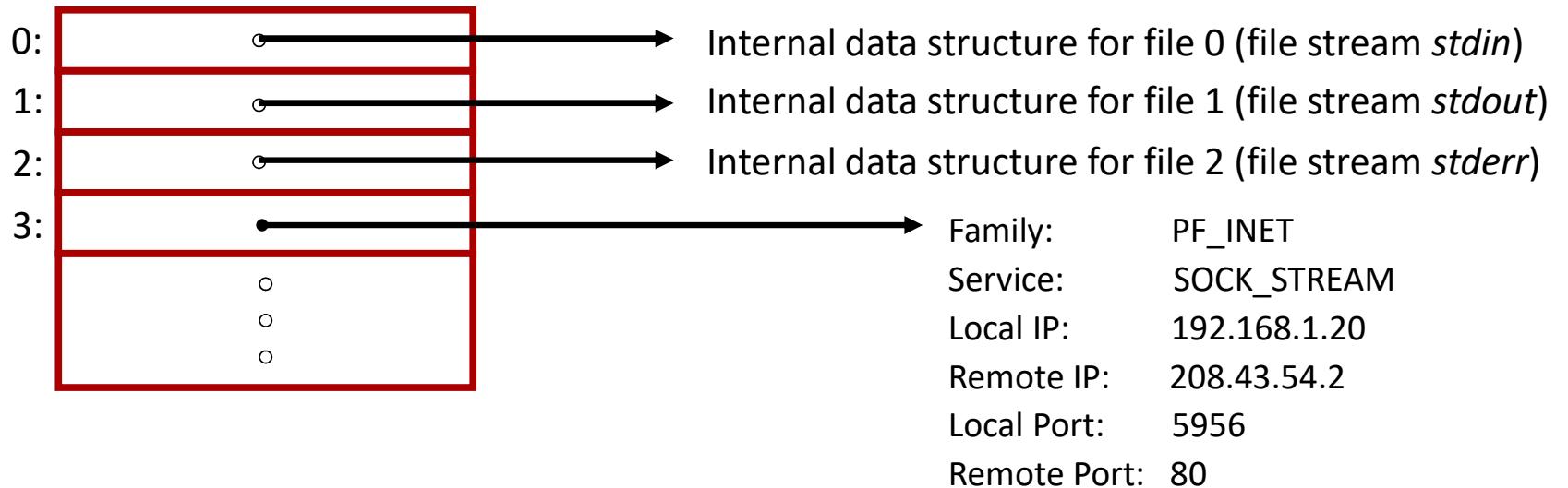


# Network Sockets



- Berkeley Socket API
  - A "socket" is the endpoint of a communication link between two processes
  - The socket API treats network connections like *files* as much as possible

# File Descriptor Table - Sockets Show Up as Files



- The file descriptor number returned by `open()` is an index into an array of pointers to internal OS data structures
- Sockets are added to this table of descriptors in the same way



# Multiple Process Communication

- Many different processes can be running on one computer
- However, an IP address only identifies the interface on the computer, not the process
- How do we know which process is communicating at that particular interface's IP address?



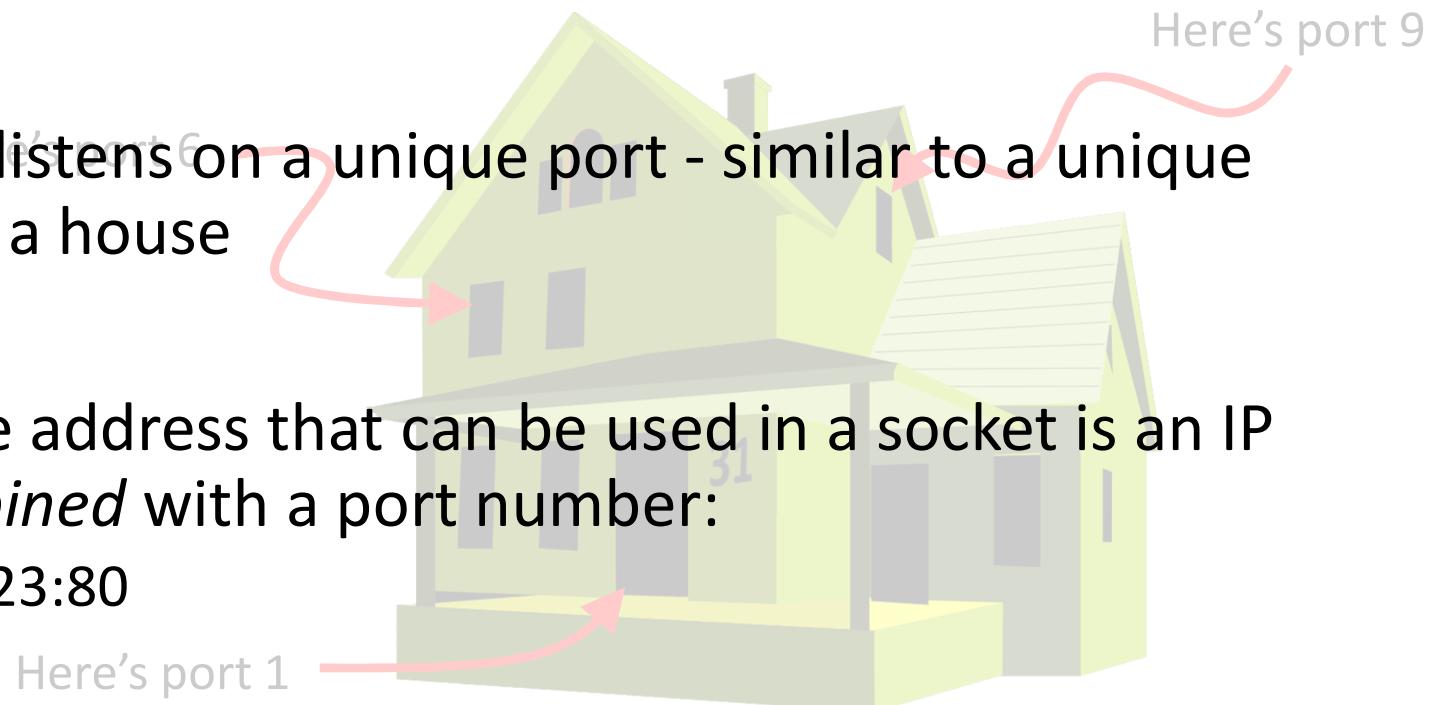
# Ports

- This house has address 31



# Ports

- *Ports* are used to reach a specific process on a machine
- Each process listens on a unique port - similar to a unique entrance into a house
- So a complete address that can be used in a socket is an IP address *combined* with a port number:
  - 43.144.31.223:80



# Socket Documentation

- Most socket related man pages are in the "3n" section
  - **man -s 3n socket**
  - **man -k socket**
- All the info you need to use the network library is scattered across different man pages
- It's definitely best to work from a known good network program starting point! Stay tuned!



# Creating and Connecting a Socket on the Client

- Process:
  - 1) Create the socket endpoint with `socket()`
  - 2) Connect the socket to the server with `connect()`
  - 3) Use `read()` and `write()`, or `send()` and `recv()`, to transfer data to and from the socket (which is sent automatically to and from the socket on the server)
- Sockets act like files, in that you can `read()` and `write()` to them
- `send()` and `recv()` are specialized, and can take special flags



# Creating the Socket

```
int socket(int domain, int type, int protocol);
```

Returns file descriptor or -1

For general-purpose sockets that can connect across a network, use AF\_INET

For sockets that are used ONLY for same-machine IPC, use AF\_UNIX

For TCP, use SOCK\_STREAM  
For UDP, use SOCK\_DGRAM

Use 0 for normal behavior

```
int socketFD = socket(AF_INET, SOCK_STREAM, 0);
if (socketFD == -1) {
    perror("Hull breach: socket()");
    exit(1);
}
```

# Connecting the Socket to an Address

```
int connect(int sockfd, struct sockaddr* address, size_t address_size);
```

Returns 0 on success, -1 on failure

Socket you want to connect

The size of the address struct

A struct that holds the *address* of where  
you're connecting, plus other settings;  
More on this coming up

```
if (connect(socketFD, (struct sockaddr*)&serverAddress, sizeof(serverAddress)))
{
    perror("Hull breach: connect()");
    exit(1);
}
```

# Filling the Address Struct: IP Address

- Getting the actual address into a form `connect()` can use it is tricky:

```
struct sockaddr_in serverAddress;  
  
serverAddress.sin_family = AF_INET;  
serverAddress.sin_port = htons(7000);  
serverAddress.sin_addr.s_addr = inet_addr("192.168.1.1");
```

`htons()`: **host-to-network-short**

Converts from *host/PC* byte order (LSB)  
to *network* byte order (MSB)

PCs store bytes with smallest digit first,  
but networks expect largest digit first

`inet_addr()` converts a standard dotted IP address  
string into an *integer* format that `sockaddr_in` requires

# Filling the Address Struct: Domain Name

- Client connecting to server:

```
struct sockaddr_in serverAddress;
struct hostent* serverHostInfo;

serverHostInfo = gethostbyname("www.oregonstate.edu");
if (serverHostInfo == NULL) {
    fprintf(stderr, "could not resolve server host name\n");
    exit(1);
}

serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(80);

memcpy((char*)&serverAddress.sin_addr.s_addr,
       (char*)serverHostInfo->h_addr, serverHostInfo->h_length);
```

Do a DNS lookup and return address information

This will be used to connect to a server on port 80

Preserve the special arrangement of the bytes in these variables by copying the bytes in the given order, regardless of format, structure, or meaning

Destination, copying to

Source, copying from

# Sending Data

Socket file descriptor

Pointer to data that  
should be sent

```
ssize_t send(int sockfd, void *message, size_t message_size, int flags);
```

Returns number  
of bytes sent

Number of bytes to send, starting  
at address in message

Configuration flags

```
char msg[1024];
```

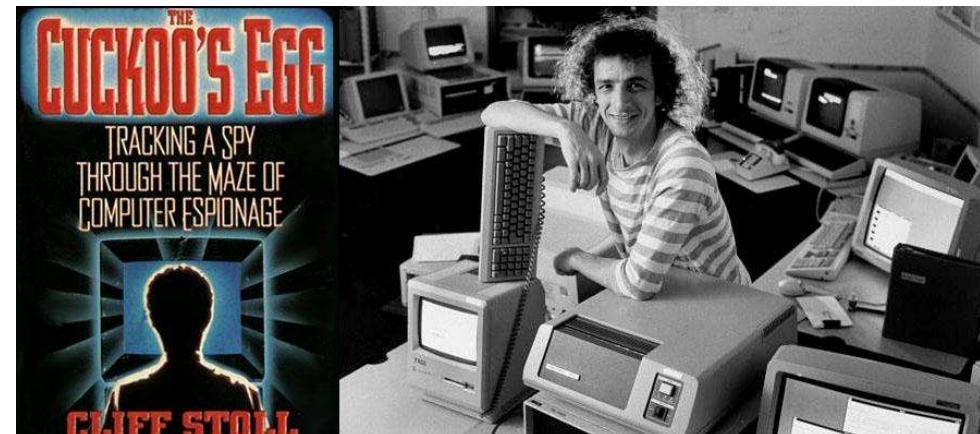
```
...
```

```
r = send(socketFD, msg, 1024, 0);  
if (r < 1024)  
    {} // handle possible error
```

If this happens, you'll have to  
call send() again to send  
what didn't get sent previously

# Sending Data

- Send will block until all the data has been sent, the connection goes away, or a signal handler interrupts the write () system call
- Remember that internet connections fail all the time
  - Client intentionally disconnects (STOP button in a web browser)
  - Network failure
  - etc.



# Receiving Data

Socket file descriptor

Pointer to where received data should be written

```
ssize_t recv(int sockfd, void *buffer, size_t buffer_size, int flags);
```

Returns number of bytes read

Maximum number of bytes to receive

Configuration flags

```
char buffer[1024];
memset(buffer, '\0', sizeof(buffer));
r = recv(socketFD, buffer, sizeof(buffer) - 1, 0);
if (r < sizeof(buffer) - 1)
    {} // handle possible error
```

Gray is really, really unlikely

- if  $r == -1$ , ERROR
- if  $0 < r < \text{sizeof(buffer)} - 1$ , there may be more data
- if  $r == 0$ , sender shut down OR sent a 0-length packet OR 0 bytes were requested

# Receiving Data

- Data may arrive in odd size bundles!
- `recv()` or `read()` will return exactly the amount of data that has already arrived
- `recv()` and `read()` will block if the connection is open but *no data is available*
  - So be careful to match what you send with what you receive, or use:

```
fcntl(socketFD, F_SETFL, O_NONBLOCK) ;
```

...to set the socket to not block if there's no data, but that means you're polling the socket, waiting for data; `select()` would be better (see next lecture!)



# Receiving Data - Using Control Codes

- Similar to controlling data being sent through pipes, you can watch for the amount of data coming through `recv()` if you know how much there should be, or use codes:

```
...
char completeMessage[512], readBuffer[10];
memset(completeMessage, '\0', sizeof(completeMessage)); // Clear the buffer

while (strstr(completeMessage, "@@") == NULL) // As long as we haven't found the terminal...
{
    memset(readBuffer, '\0', sizeof(readBuffer)); // Clear the buffer
    r = recv(socketFD, readBuffer, sizeof(readBuffer) - 1, 0); // Get the next chunk
    strcat(completeMessage, readBuffer); // Add that chunk to what we have so far
    printf("PARENT: Message received from child: \"%s\", total: \"%s\"\n", readBuffer, completeMessage);
    if (r == -1) { printf("r == -1\n"); break; } // Check for errors
    if (r == 0) { printf("r == 0\n"); break; }
}

int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
printf("PARENT: Complete string: \"%s\"\n", completeMessage);
...
```

# Debugging the Contents of Buffers

- Often, when writing send and receive functions, you'll get garbage. Here's an easy way to actually check what's in a buffer:

```
int x = 0;  
printf("CHAR INT\n");  
for (x = 0; x < strlen(buffer); x++)  
    printf(" %c    %d\n", buffer[x], buffer[x]);
```

Show all chars up to  
the first newline

- Or:

```
int x = 0;  
printf("CHAR INT\n");  
for (x = 0; x < sizeof(buffer); x++)  
    printf(" %c    %d\n", buffer[x], buffer[x]);
```

Show all chars in the  
entire array

# Debugging the Contents of Buffers: Results

CHAR	INT
o	111
O	79
o	48
	32
l	108
	10
L	76
1	49
	0

Annotations:

- Annotation for INT 32: "Space" with a red arrow pointing to the row.
- Annotation for INT 10: "New line" with a red arrow pointing to the row.
- Annotation for INT 0: "Null terminator" with a red arrow pointing to the row.

- Look up these ints in a good ASCII table, like this one:  
<http://www.asciitable.com>



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg) { perror(msg); exit(0); } // Error function used for reporting issues

int main(int argc, char *argv[])
{
    int socketFD, portNumber, charsWritten, charsRead;
    struct sockaddr_in serverAddress;
    struct hostent* serverHostInfo;
    char buffer[256];

    if (argc < 3) { fprintf(stderr,"USAGE: %s hostname port\n", argv[0]); exit(0); } // Check usage & args

    // Set up the server address struct
    memset((char*)&serverAddress, '\0', sizeof(serverAddress)); // Clear out the address struct
    portNumber = atoi(argv[2]); // Get the port number, convert to an integer from a string
    serverAddress.sin_family = AF_INET; // Create a network-capable socket
    serverAddress.sin_port = htons(portNumber); // Store the port number
    serverHostInfo = gethostbyname(argv[1]); // Convert the machine name into a special form of address
    if (serverHostInfo == NULL) { fprintf(stderr, "CLIENT: ERROR, no such host\n"); exit(0); }
    memcpy((char*)&serverAddress.sin_addr.s_addr, (char*)serverHostInfo->h_addr, serverHostInfo->h_length);
                                            // Copy in the address
```

This is a basic client program that can send and receive.  
It is intended to pair with server.c

# client.c 1 of 2

# client.c 1 of 2

# client.c 2 of 2

```
// Set up the socket
socketFD = socket(AF_INET, SOCK_STREAM, 0); // Create the socket
if (socketFD < 0) error("CLIENT: ERROR opening socket");

// Connect to server
if (connect(socketFD, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) < 0) // Connect socket to addy
    error("CLIENT: ERROR connecting");

// Get input message from user
printf("CLIENT: Enter text to send to the server, and then hit enter: ");
memset(buffer, '\0', sizeof(buffer)); // Clear out the buffer array
fgets(buffer, sizeof(buffer) - 1, stdin); // Get input from the user, trunc to buffer - 1 chars, leaving \0
buffer[strcspn(buffer, "\n")] = '\0'; // Remove the trailing \n that fgets adds

// Send message to server
charsWritten = send(socketFD, buffer, strlen(buffer), 0); // Write to the server
if (charsWritten < 0) error("CLIENT: ERROR writing to socket");
if (charsWritten < strlen(buffer)) printf("CLIENT: WARNING: Not all data written to socket!\n");

// Get return message from server
memset(buffer, '\0', sizeof(buffer)); // Clear out the buffer again for reuse
charsRead = recv(socketFD, buffer, sizeof(buffer) - 1, 0); // Read data from the socket, leaving \0 at end
if (charsRead < 0) error("CLIENT: ERROR reading from socket");
printf("CLIENT: I received this from the server: \"%s\"\n", buffer);

close(socketFD); // Close the socket
return 0;
}
```

# Client/Server Results

```
$ gcc -o client client.c
$ gcc -o server server.c
$ ./server 51717 &
[1] 21094
$ ./client localhost 51717
CLIENT: Enter text to send to the server, and then hit enter: AWESOMESAUCE
SERVER: I received this from the client: "AWESOMESAUCE"
CLIENT: I received this from the server: "I am the server, and I got your message"
[1]+  Done                  ./server 51717
$
```



# Network Servers

Benjamin Brewster

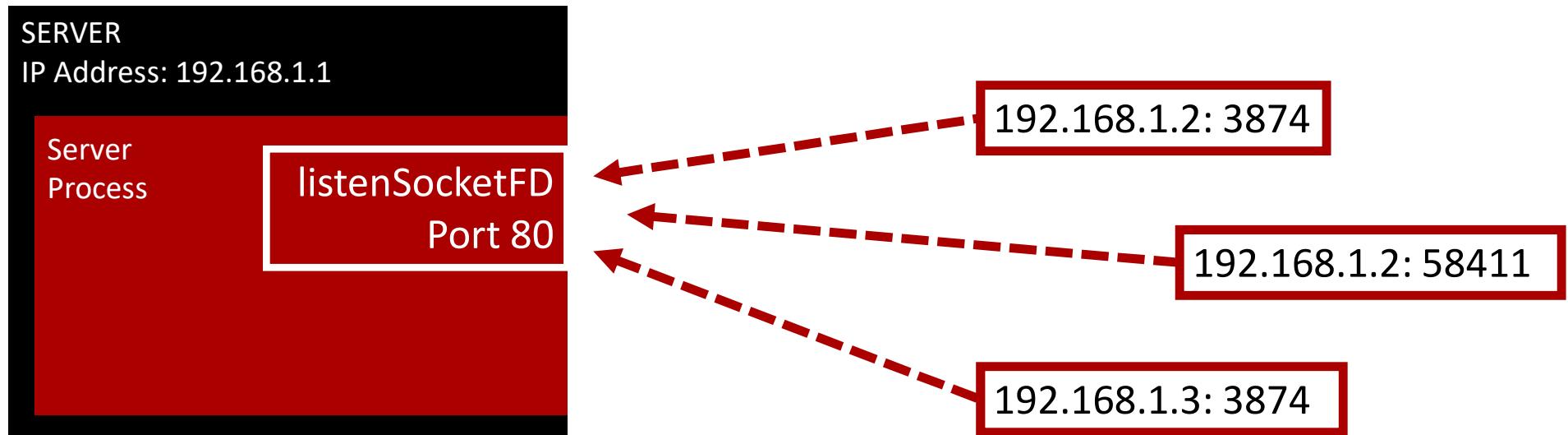
Except as noted, all images copyrighted with Creative Commons licenses,  
with attributions given whenever available

# Network Servers

- This lecture covers
  - Setting up network sockets and connecting clients to them
  - Demo working server code
  - Server concurrency methodologies
  - Knowing when data is available

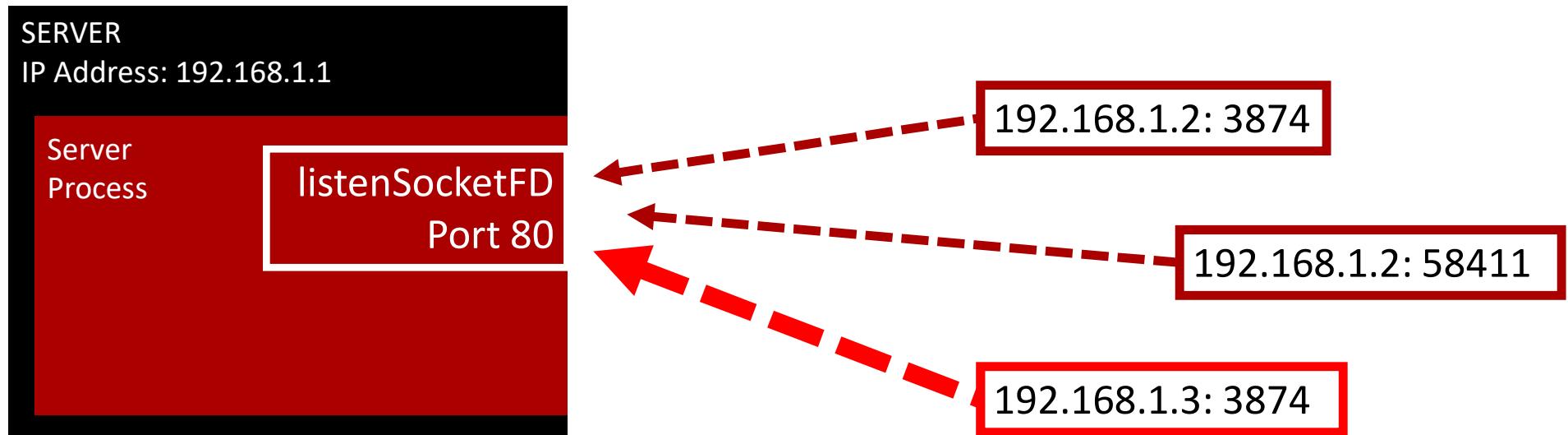


# Non-Concurrent Server Connection Overview



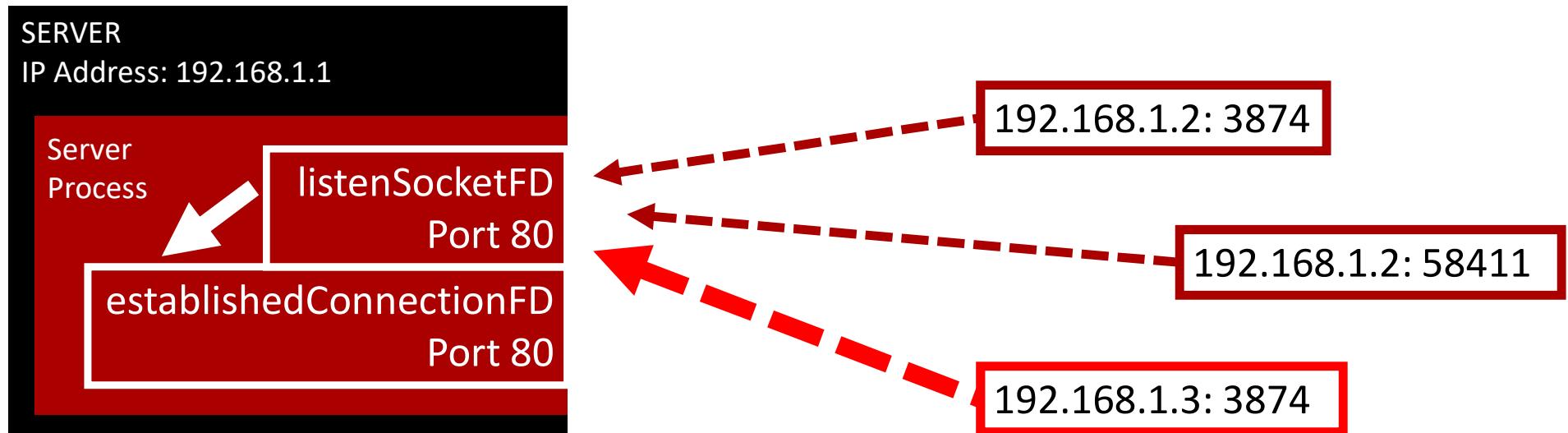
- A server socket listens on a specified port
- Many different clients may be connecting to that port
- The server needs to *differentiate* between and *communicate* with each client

# Non-Concurrent Server Connection Overview



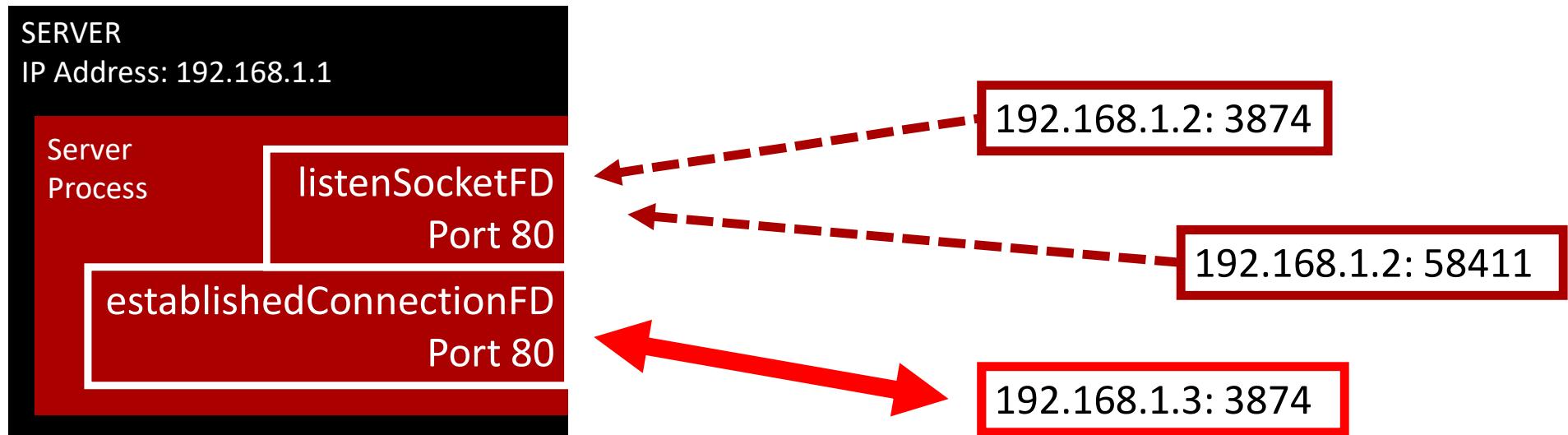
- Step 1: The server chooses the next connection to deal with

# Non-Concurrent Server Connection Overview



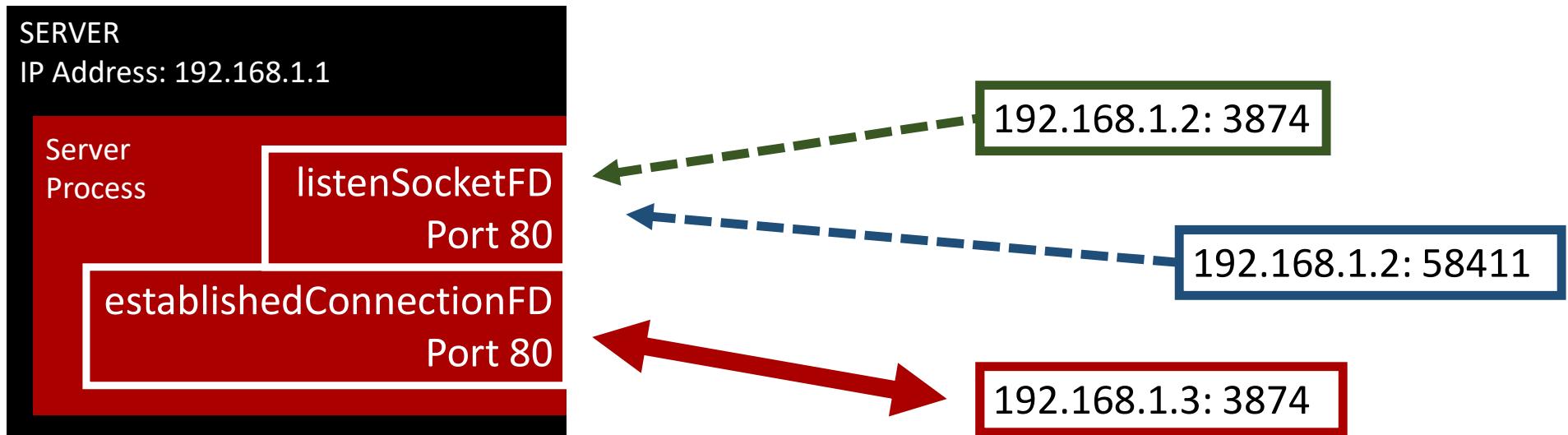
- Step 2: The server creates a new file descriptor for the chosen connection to exclusively use
- Note that the port doesn't change!

# Non-Concurrent Server Connection Overview



- Step 3: The chosen client begins communication with the new file descriptor created by the server
- New connections can now be handled again by listenSocketFD

# Connection Differentiation



- The OS network layer on the server *differentiates each connection* by using four pieces of the addresses, routing the network packets to the correct socket/FD based on:
  - Server IP, Server Port, Client IP, Client Port

# Server Sockets API

- Server Procedure:
  1. Create a network socket/FD with `socket()`
  2. Bind the socket to a port number with `bind()`
  3. Start listening for connections on that socket/port with `listen()`
  4. Loop and accept connections on that socket/port with `accept()`, connecting them to *new* sockets for each connection's exclusive use
  5. Read and write data to and from the *newly* created sockets for connected and accepted clients using `send()` & `recv()` or `read()` & `write()`



# Creating the Socket - Same Method as Client

```
int socket(int domain, int type, int protocol);
```

Returns file descriptor or -1

For general-purpose sockets that can connect across a network, use AF\_INET

For sockets that are used ONLY for same-machine IPC, use AF\_UNIX

For TCP, use SOCK\_STREAM  
For UDP, use SOCK\_DGRAM

Use 0 for normal behavior

```
int socketFD = socket(AF_INET, SOCK_STREAM, 0);
if (socketFD < 0) {
    perror("Hull breach: socket()");
    exit(1);
}
```

# Filling the Address Struct for the Server

- Set the address struct so that it accepts connections from any IP address, or just one, and which specific port it will be available on:

```
struct sockaddr_in serverAddress;  
  
serverAddress.sin_family = AF_INET;  
serverAddress.sin_port = htons(80);  
serverAddress.sin_addr.s_addr = INADDR_ANY;
```

htons () : **host-to-network-short**  
Converts from *host/PC* byte order (LSB)  
to *network* byte order (MSB)  
PCs store bytes with smallest digit first,  
but networks expect largest digit first

Allows connections from any IP address

# Bind the Socket to a Port

- Ports allow multiple processes running on a single machine to communicate across the network from only a single IP address
- A server process has to choose a port where clients can contact it on
- `bind()` associates the chosen port with a socket already created with the `socket()` command
- Subsequent calls to `bind()` using an already-bound socket will fail
- Even after the sockets are all closed, the OS does not immediately release the port; you'll need to wait many seconds for it to be available again for reuse



# Binding the Socket

```
int bind(int sockfd, struct sockaddr *address, size_t add_len);
```

Returns 0 on success or -1 on error

The socket we're binding to the port

The network address struct, which identifies which port this socket will use

The size of the address struct

```
if (bind(listenSocketFD, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) < 0)
{
    perror("Hull breach: bind()");
    exit(1);
}
```

# Listening for Connections

- The server will ignore any connection attempts until you tell the socket to start listening for connections with `listen()`
- Once this has been done, the socket will begin queuing up connection requests until it reaches the connection queue limit

```
int listen(int sockfd, int queue_size);
```

Returns 0 on success or -1 on error

The socket we're enabling for connections

Maximum number of connections to queue

```
if (listen(sockfd, 5) < 0) {
    perror("Hull breach: listen()");
    exit(1);
}
```

# Loop and Accept

- Servers generally run continually, waiting for clients to contact them
- Thus a server has an "infinite loop" that continually processes connections from clients
- The `accept()` function takes the next connection off of the listen queue for a socket, or blocks the process until a connection request arrives



# Accepting Connections

```
int accept(int sockfd, struct sockaddr* address, size_t &add_len);
```

Returns file descriptor for new connection or -1 on error

The socket we're going to get a connection from

Network address struct, into which connecting *client* information will be written

The size of the address struct

```
int establishedConnectionFD = accept(listenSocketFD,
                                      (struct sockaddr*)&clientAddress,
                                      &sizeOfClientInfo);

if (establishedConnectionFD < 0) { perror("Hull breach: accept()"); exit(1); }
```

# server.c 1 of 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg) { perror(msg); exit(1); } // Error function used for reporting issues

int main(int argc, char *argv[])
{
    int listenSocketFD, establishedConnectionFD, portNumber, charsRead;
    socklen_t sizeOfClientInfo;
    char buffer[256];
    struct sockaddr_in serverAddress, clientAddress;

    if (argc < 2) { fprintf(stderr,"USAGE: %s port\n", argv[0]); exit(1); } // Check usage & args

    // Set up the address struct for this process (the server)
    memset((char *)&serverAddress, '\0', sizeof(serverAddress)); // Clear out the address struct
    portNumber = atoi(argv[1]); // Get the port number, convert to an integer from a string
    serverAddress.sin_family = AF_INET; // Create a network-capable socket
    serverAddress.sin_port = htons(portNumber); // Store the port number
    serverAddress.sin_addr.s_addr = INADDR_ANY; // Any address is allowed for connection to this process

    // Set up the socket
    listenSocketFD = socket(AF_INET, SOCK_STREAM, 0); // Create the socket
    if (listenSocketFD < 0) error("ERROR opening socket");

    This is a basic server program
    that can send and receive.

    It is intended to pair with client.c
```

# server.c 2 of 2

```
// Enable the socket to begin listening
if (bind(listenSocketFD, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0) // Connect socket to port
    error("ERROR on binding");
listen(listenSocketFD, 5); // Flip the socket on - it can now receive up to 5 connections

// Accept a connection, blocking if one is not available until one connects
sizeOfClientInfo = sizeof(clientAddress); // Get the size of the address for the client that will connect
establishedConnectionFD = accept(listenSocketFD, (struct sockaddr *)&clientAddress, &sizeOfClientInfo); // Accept
if (establishedConnectionFD < 0) error("ERROR on accept");

// Get the message from the client and display it
memset(buffer, '\0', 256);
charsRead = recv(establishedConnectionFD, buffer, 255, 0); // Read the client's message from the socket
if (charsRead < 0) error("ERROR reading from socket");
printf("SERVER: I received this from the client: \"%s\"\n", buffer);

// Send a Success message back to the client
charsRead = send(establishedConnectionFD, "I am the server, and I got your message", 39, 0); // Send success back
if (charsRead < 0) error("ERROR writing to socket");
close(establishedConnectionFD); // Close the existing socket which is connected to the client

close(listenSocketFD); // Close the listening socket
return 0;
}
```

# Client/Server Results

```
$ gcc -o client client.c
$ gcc -o server server.c
$ ./server 51717 &
[1] 21094
$ ./client localhost 51717
CLIENT: Enter text to send to the server, and then hit enter: AWESOME SAUCE
SERVER: I received this from the client: "AWESOME SAUCE"
CLIENT: I received this from the server: "I am the server, and I got your message"
[1]+  Done                      ./server 51717
$
```



# multiserver.c 1 of 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg) { perror(msg); exit(1); } // Error function used for reporting issues

int main(int argc, char *argv[])
{
    int listenSocketFD, establishedConnectionFD, portNumber, charsRead;
    socklen_t sizeOfClientInfo;
    char buffer[256];
    struct sockaddr_in serverAddress, clientAddress;

    if (argc < 2) { fprintf(stderr,"USAGE: %s port\n", argv[0]); exit(1); } // Check usage & args

    // Set up the address struct for this process (the server)
    memset((char *)&serverAddress, '\0', sizeof(serverAddress)); // Clear out the address struct
    portNumber = atoi(argv[1]); // Get the port number, convert to an integer from a string
    serverAddress.sin_family = AF_INET; // Create a network-capable socket
    serverAddress.sin_port = htons(portNumber); // Store the port number
    serverAddress.sin_addr.s_addr = INADDR_ANY; // Any address is allowed for connection to this process

    // Set up the socket
    listenSocketFD = socket(AF_INET, SOCK_STREAM, 0); // Create the socket
    if (listenSocketFD < 0) error("ERROR opening socket");

    This server lives forever by wrapping
    the accept() section in a while loop
    It is also intended to pair with client.c
```

# multiserver.c 2 of 2

```
// Enable the socket to begin listening
if (bind(listenSocketFD, (struct sockaddr *)&serverAddress, sizeof(serverAddress)) < 0) // Connect socket to port
    error("ERROR on binding");
listen(listenSocketFD, 5); // Flip the socket on - it can now receive up to 5 connections
while (1) {
    // Accept a connection, blocking if one is not available until one connects
    sizeOfClientInfo = sizeof(clientAddress); // Get the size of the address for the client that will connect
    establishedConnectionFD = accept(listenSocketFD, (struct sockaddr *)&clientAddress, &sizeOfClientInfo); // Accept
    if (establishedConnectionFD < 0) error("ERROR on accept");
    printf("SERVER: Connected Client at port %d\n", ntohs(clientAddress.sin_port));
    // Get the message from the client and display it
    memset(buffer, '\0', 256);
    charsRead = recv(establishedConnectionFD, buffer, 255, 0); // Read the client's message from the socket
    if (charsRead < 0) error("ERROR reading from socket");
    printf("SERVER: I received this from the client: \"%s\"\n", buffer);

    // Send a Success message back to the client
    charsRead = send(establishedConnectionFD, "I am the server, and I got your message", 39, 0); // Send success back
    if (charsRead < 0) error("ERROR writing to socket");
    close(establishedConnectionFD); // Close the existing socket which is connected to the client
}
close(listenSocketFD); // Close the listening socket
return 0;
}
```

# Client/Multiserver Results

Note that the order of the client and server sending text to the terminal is hard to control!

```
$ multiserver 55556 &
[1] 26889

$ client localhost 55556
CLIENT: Enter text to send to the server, and then hit enter: SERVER: Connected Client at port 38422
My Test!
SERVER: I received this from the client: "My Test!"
CLIENT: I received this from the server: "I am the server, and I got your message"

$ client localhost 55556
CLIENT: Enter text to send to the server, and then hit enter: SERVER: Connected Client at port 38424
So much text!!
SERVER: I received this from the client: "So much text!!"
CLIENT: I received this from the server: "I am the server, and I got your message"

$ kill -TERM 26889
[1]+  Terminated                  multiserver 55556
```



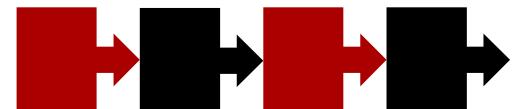
# Doing it All at Once

- Many clients may need to both connect and perform tasks at the same time
- We want to minimize:
  - Response time
  - Complexity
- Want to maximize:
  - Throughput (connections serviced / second)
  - Hardware utilization (%CPU usage)
- These are all tradeoffs of each other!
- Let's look at two methods...



# Iterative Servers

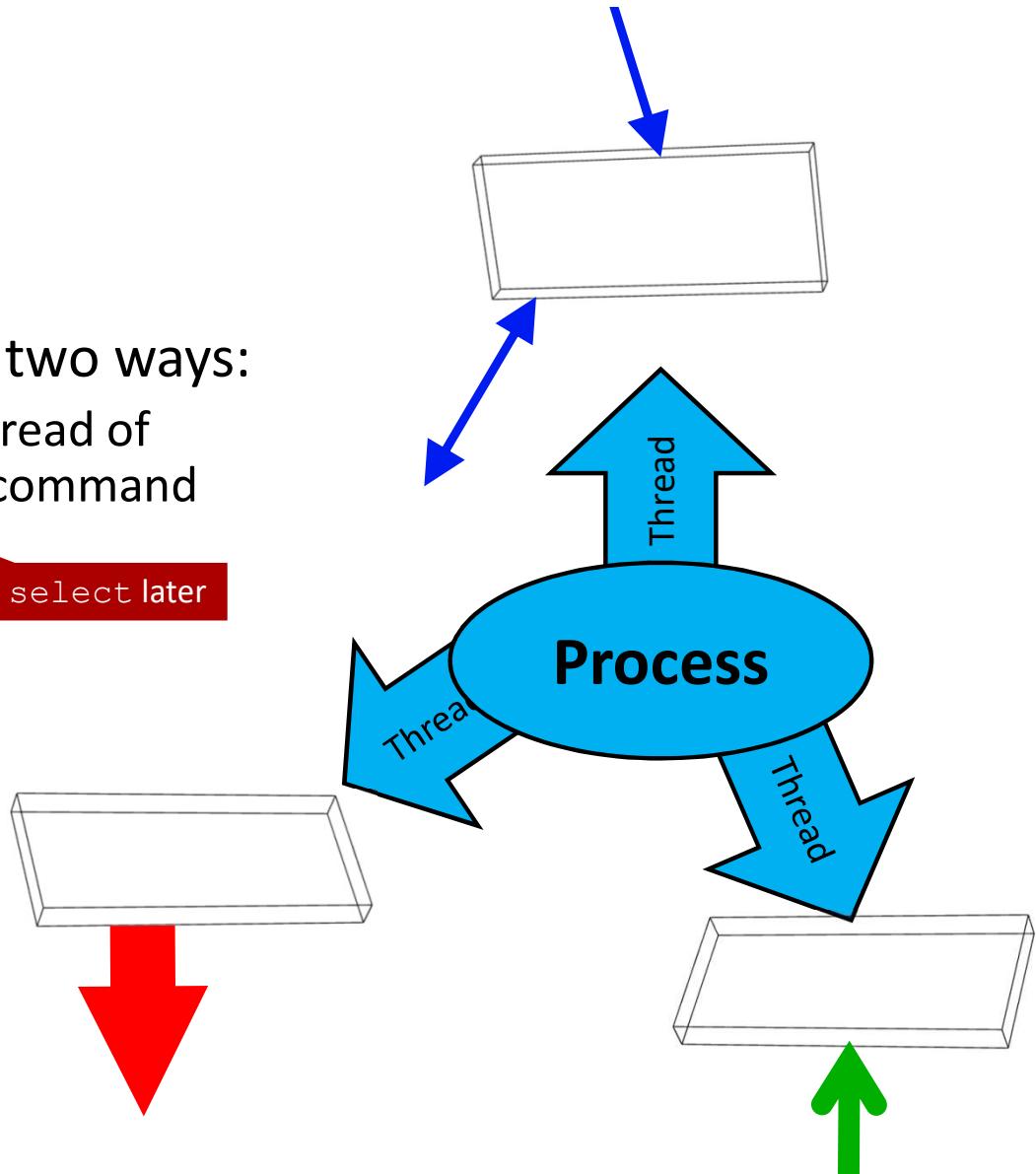
- Iterative
  - Handles only one client at a time
  - Non-preemptive: additional client must wait for all previous requests to complete
- Easy to design, implement, and maintain
- Best when:
  - Request processing time is short
  - No I/O is needed by server
  - Order matters



# Concurrent Servers

- Concurrency can be provided in two ways:
  - Apparent concurrency: a single thread of execution, using the `select()` command and non-blocking I/O
  - Real concurrency: multiple threads of execution, or multiple processes, each with one thread

More on select later



# Apparent Concurrency: Details

- Only one thread, no preemption, using non-blocking I/O
- Whenever an I/O request would block, switch to another connection
- Up to a certain number of clients being served:
  - Maximizes CPU utilization
  - Increases throughput
- Complexity involves tracking connections, choosing the next to run, detecting blocking calls, etc.
- Works well if requests are short



# Real Concurrency: Details

- Preemptive
  - Clients can connect anytime to the server, which uses multiple threads or processes to service connections
- Up to a certain number of connections:
  - Maximizes CPU utilization
  - Maximizes response time
  - Increases throughput
- Harder to design, implement, and *Maintain*:
  - After too many concurrent connections:
    - Everything gets worse -> server eventually hangs
    - Need to put limits on concurrent connections



# More Real Concurrency

- Four different methods
  - Create one process per client connection
  - Create a pool of available processes before clients connect
  - Use only one process, but create one thread per client connection
  - Use only one process, but create a pool of available threads before clients connect



# Fork Solution #1



- One process per client connection
- Fork a new process to handle every connection
- Advantages:
  - Simple: minimal shared state to worry about
- Disadvantages:
  - Process creation via `fork()` is slow
  - Context-switching between processes is also slow, but minor compared to `fork()`

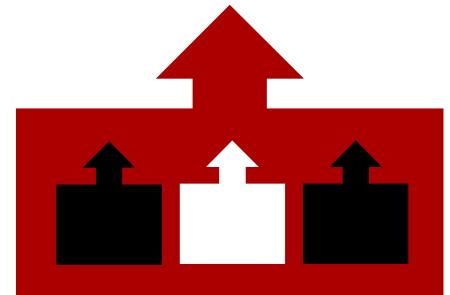
# Fork Solution #2



- Create a pool of available processes for clients to use
- Advantages:
  - No longer have to fork
  - Have rapid response as long as there is an idle process available
  - Can set the pool size, so that you don't overload the hardware
- Disadvantages:
  - Still have process context switching
  - Managing the pool of processes can be complex

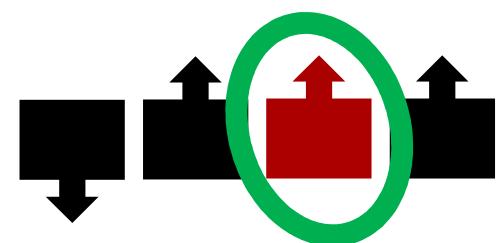
# Threads Solutions 1 & 2

- Threads allow multiple concurrent execution contexts within a single process
- Can implement a server as a single process with multiple threads
  - Either one thread per connection, or a pool of threads
- Advantages:
  - Trades process context switches (slow) for thread switches (fast)
  - Shared address space, shared code, shared data, etc.
- Disadvantages:
  - Code must be thread-safe
  - Must always worry about inadvertent data-sharing



# select ()

- select () is designed for server-like applications that have many communication channels open at once (like a pool of threads)
  - Data or space may become available at any time on *any* of the channels
  - You want to minimize the delay between when data/space becomes available and your process takes action
- Overview: you call select () with a list of read and/or write file descriptors, and it *returns* when any one of those descriptors becomes readable or writable



# select()

```
int select(
    int nfds,                                // Highest numbered FD + 1
    fd_set* readfds,                          // Input FDs of interest
    fd_set* writefds,                         // Output FDs of interest
    fd_set* errorfds,                         // FDs where exception has occurred
    struct timeval* timeout                  // when to time out if nothing happens
)
```

- The three parameters `readfds`, `writefds`, and `errorfds` are bit masks
  - Each bit of the number refers to one file descriptor
  - Bit 0 is file descriptor 0, bit 1 is file descriptor 1, etc.
- UNIX provides you with macros to manipulate bit masks:
  - `FD_ZERO()` :: Set all bits to 0
  - `FD_SET()` :: Set one specific bit to 1
  - `FD_ISSET()` :: Determine if a specific bit is set to 1
  - `FD_CLR()` :: Set one specific bit to 0

## select() Return Values

- -1 if error
- 0 if time out: nothing ready
- Else, the return value is the *number* of file descriptors ready for reading, writing, or have had errors occur

# selectDemo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    fd_set readFDs;
    struct timeval timeToWait;
    int retval;

    // Watch stdin (FD 0) to see when it has input
    FD_ZERO(&readFDs); // Zero out the set of possible read file descriptors
    FD_SET(0, &readFDs); // Mark only FD 0 as the one we want to pay attention to

    // Wait up to 50 seconds
    timeToWait.tv_sec = 50;
    timeToWait.tv_usec = 0;

    retval = select(1, &readFDs, NULL, NULL, &timeToWait); // Check to see whether any read FDs have data!
                                                        // After select returns, timeToWait is undefined
    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now!\n"); // FD_ISSET(0, &readFDs) will return true
    else
        printf("No data within 50 seconds\n");

    return(0);
}
```

This example comes from the  
select() man page

Together with returning an int, select() also overwrites your bit masks to show you *which* bits are interesting; you'll have to iterate through them to see which ones are set, though

# selectDemo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    fd_set ready;
    struct timeval timeout;
    int retval;
    :%s/\//\//g
    // Watch for readable file descriptor
    FD_ZERO(&ready);
    FD_SET(0, &ready);

    // Wait up to 1 second
    timeToWait = 1.0;
    timeToWait = 0.0;

    retval = select(0, &ready, NULL, NULL, &timeout);

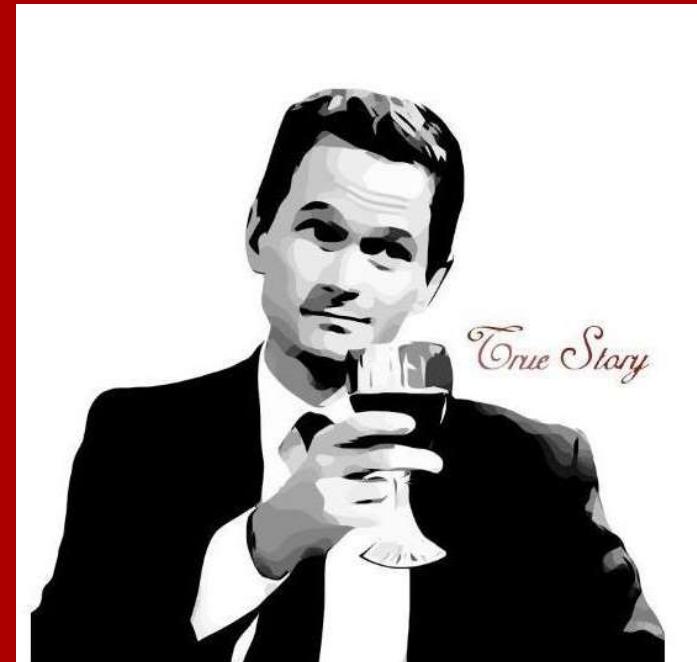
    if (retval < 0)
        perror("select error");
    else if (FD_ISSET(0, &ready))
        /* Readable */
    else
        /* No readable */

    return(0)
}
```

In marking up this code and testing it, the double slash comments // were confusing the copy/paste functions in vim. So, I changed them all to single slashes, pasted it all into vim, and then entered this command:

```
:%s/\//\//g
```

... and it worked, first try.



read FDs have data!  
oWait is undefined

# selectDemo.c Results

```
$ mkfifo myfifo  
$ selectDemo < myfifo &  
[1] 17013  
  
$ echo "text" > myfifo  
Data is available now!  
[1]+ Done
```

```
selectDemo < myfifo
```

This hooks the output of the echo command to the input of the selectDemo program through a named pipe!

As soon as both ends are opened, and data is written, the pipe transfers the data



# UNIX & Security

Benjamin Brewster

Except as noted, all images copyrighted with Creative Commons licenses,  
with attributions given whenever available

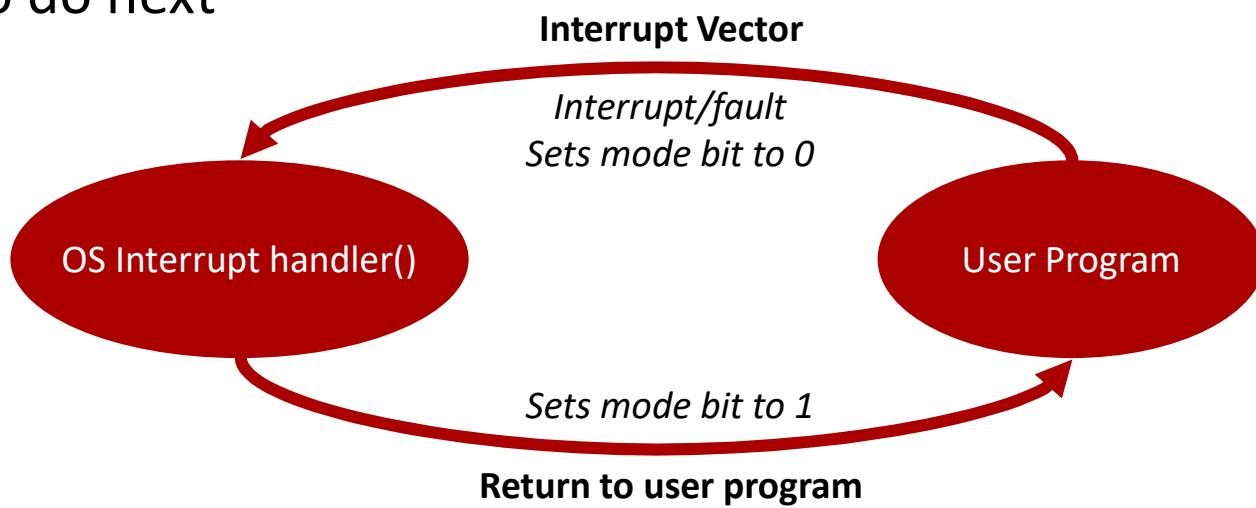
# Dual-Mode Operation

- Sharing system resources requires the operating system to ensure that a program cannot arbitrarily interfere with other programs
- The hardware itself provides support to differentiate between at least two modes of operations:
  - User mode: execution done on behalf of a user
  - Monitor mode (also supervisor mode or system mode): execution done on behalf of the operating system
- Privileged instructions can be issued only in monitor mode



# Dual-Mode Operation

- The *mode bit* is added to computer hardware to indicate the current mode: monitor (0) or user (1)
- When an interrupt or fault occurs, the hardware switches to monitor mode by following the address, stored in the *interrupt vector*, to the *interrupt handler* function in the OS; this handler will let the OS decide what to do next



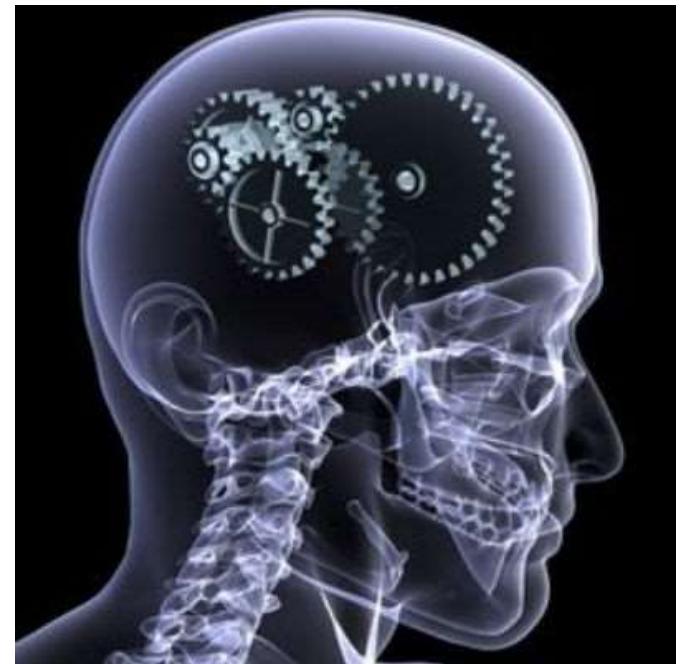
# I/O Protection

- All I/O instructions (`read()`, `write()`, `send()`, `recv()`, `fgets()`, `putc()`, etc.) are privileged instructions
- Because: the OS must ensure that a user program could never gain control of the computer in monitor mode by storing a new address in the interrupt vector



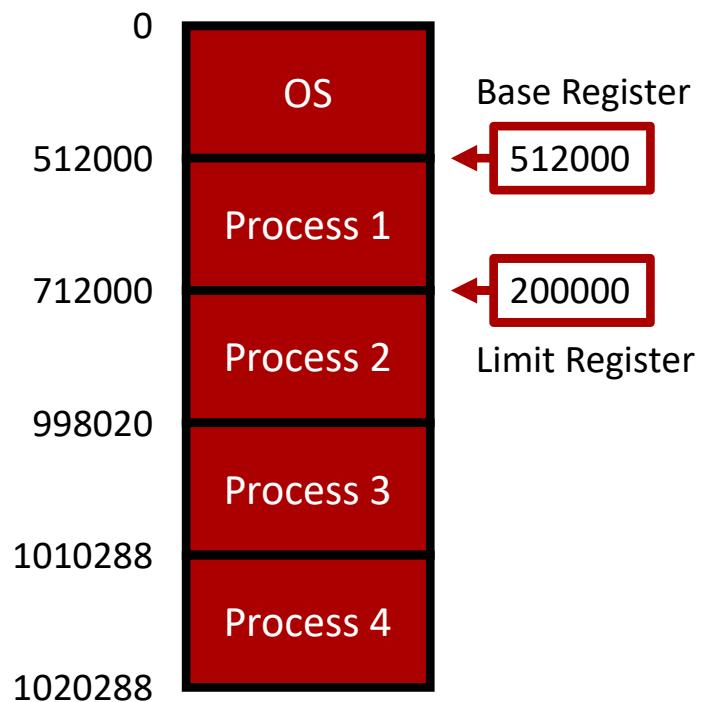
# Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt handler function
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
  - **Base register** – holds the smallest legal physical memory address.
  - **Limit register** – contains the size of the range
- Memory outside the defined range is protected



# Memory Protection

- The base and limit registers define a logical address space, which is virtualized for the process to start at address 0
- When executing in monitor mode, the operating system has unrestricted access to both monitor and user's memory
- Obviously, the load instructions for the base and limit registers are privileged instructions



# CPU Protection

- If the CPU is executing program instructions one after the next, how does the OS retain control?
- A timer interrupts the control flow after a specified period to ensure that the operating system has a chance to determine what to do
  - Timer is decremented every clock tick
  - When timer reaches the value 0, the interrupt vector is followed to the interrupt handler
- Timer commonly used to implement time sharing
- Also used to compute the current time
- “Load-timer” is a privileged instruction



# General-System Architecture

- Given the I/O instructions are privileged, how does the user program perform I/O?
- With a *system call*: the method used by a process to request action by the operating system
  - Control passes through the interrupt vector to a service routine in the OS, and the mode bit is set to monitor mode
  - The monitor verifies that the parameters are correct and legal, executes the request, and returns control to the program instruction immediately following the system call



# User Account Rights Protect Files

- User files are protected from other users by defining access based on user accounts
- If you are logged in as an account with access (e.g., you're the owner, or a group owner), you can manipulate the file:
  - chmod
  - vim
  - touch
  - rm
  - etc.



# Acting as a Different User - Pretexting

- If you want to temporarily act as a different user (but stay logged on as yourself), you can use the `su` command:
  - `su yoog`
- Obviously, you'll need to know yoog's login credentials
- You can also execute just one action with the `sudo` command:
  - `sudo -u yoog rm -rf ~/yoogFiles/*`
- These commands change your effective user and/or group IDs, all of which can be displayed with the `id` command

# The root User Account

- Most UNIX systems have a super-user account, typically called root, which has permissions to do anything
  - su root
  - sudo -u root pkill -u brewsteb
- As root, you can change file ownership, change limits on how many processes users can run at once, add and delete user accounts, and many other things
- It is generally considered bad form to stay logged-in to root itself - it's preferred that you make use of sudo to make changes



# SUID, SGID

- Each executable has two security bits associated with it: SUID, and SGID
  - If SUID is set, the executable runs with effective user ID of the *owner* of the file
  - If SGID is set, the executable runs with effective user ID of the *group owner* of the file
- This is different from before – we're now talking about specific executables that have bits that enable them to run as different users
  - As opposed to *being* a different user, and then running programs, as `su` and `sudo` allow



# Changing SUID Example

```
$ which ping  
/bin/ping  
  
$ ls -pla /bin/ping  
-rwsr-xr-x. 1 root root 38264 May 10 2016 /bin/ping  
  
$ chmod u-s /bin/ping  
chmod: changing permissions of `/bin/ping': Operation not permitted  
  
$ ls -pla junk.test  
-rw-rw----. 1 brewsteb upg57541 332 Nov 17 09:47 junk.test  
  
$ chmod u+s junk.test  
  
$ ls -pla junk.test  
-rwSr-----. 1 brewsteb upg57541 332 Nov 17 09:47 junk.test  
  
$ chmod u+x junk.test  
  
$ ls -pla junk.test  
-rwsrw----. 1 brewsteb upg57541 332 Nov 17 09:47 junk.test
```

Can't change the permissions of a file I don't own

Capital 'S' means that the SUID bit is set, but user execute is not

Changes to lower case 's' now that SUID is set

# chmod Revisited

- It turns out that there are twelve mode bits:
  - 4000 - Setuid on execution
  - 2000 - setgid on execution
  - 1000 - set sticky bit
  - 0400 - read by owner
  - 0200 - write by owner
  - 0100 - execute by owner
  - 0040 - read by group
  - 0020 - wr
  - 0010 - execute by group
  - 0004 - read by others
  - 0002 - write by others
  - 0001 - execute by others



# Why SUID Matters

- What if you replace the contents of the real ping, which has SUID set and is owned by root, with your own code?
- It would have the same permissions (owned by root), but could do anything you want to the system



# Why SUID Matters

- What happens when you set the SUID bit on your own executables?
- They would still be owned by you, and thus would run as you
  - Since you're not root this isn't very interesting
- Can you give your custom executable to root?
- No – this is specifically why you have to *be* logged in as root to change file ownership!
  - chown doesn't work unless you're root
  - chgrp don't work unless you are a member of that group



# Strongest Forms of Security

- The strongest forms of security involve network and physical isolation, but these seriously limit utility
- If you do grant physical access to your computer - even disabling local login access - you still have to worry about:
  - Bootable devices (live CDs, flash drives, etc.) can boot a different OS that can access the hard drive of your computer
  - Hard drive could be stolen and read
  - Reading link-level NIC lights, keyboard EM
- With local logins, passwords = pain



# Actual Password Security... is a Pain in the Neck

- Don't let users write them down
- Age the passwords
- Enforce stronger (but more annoying) passwords
  - 1337: @nt3@t3|2
  - random: Z1#3s8u\*h
  - long: Ho\|\doYouTypeMeF@st
- Restrict use of previous passwords
- Password dictionary check



# Password Security

- Longer is better than more complicated
  - Lower case letters = 26 possibilities per character
  - Upper case letters = 26 possibilities per character
  - Numbers = 10 possibilities per character
  - Special Characters = 30 possibilities per character
  - Any given character could be 1 of 92 choices
  - There are then  $92^8$  8-character passwords:
    - $92 \times 92 = 92^8$
    - $92^8 = 5.1 \times 10^{15} = 5,132,188,731,375,616$

# Password Security

- Longer is better than more complicated
  - $92^8 = 5.1 \times 10^{15} = 5,132,188,731,375,616$
  - Using just lower case letters:
    - $26^8 = 2.0 \times 10^{11} = 208,827,064,576$
    - A 12 character, lower-case password:
      - $26^{12} = 9.5 \times 10^{16} = 95,428,956,661,682,176$

# Password Security

- Which is easier to remember:
  - TR0m&on3
  - ihavetwoarms
- Which are you more likely to write down?
- FYI, 4 common words are important in the example above
  - See xkcd's excellent correct horse battery staple comic:  
<https://xkcd.com/936/>



Most common  
passwords  
recovered from  
hacked data  
dumps

# WORST PASSWORDS OF 2014



- 1 123456
- 2 password
- 3 12345
- 4 12345678
- 5 qwerty
- 6 123456789
- 7 1234
- 8 baseball
- 9 dragon
- 10 football



# Login Failures

- What happens if you don't lock a user account if too many failures happen?
  - A account can be brute forced by guessing possibilities
- Passwords are generated with the sausage model (one-way):
  - username: UserBob
  - password: 123456 -> *hashes to* -> a3R7nito5fo%r
- Store the pair UserBob / a3R7nito5fo%r
- This encrypted pair is public knowledge, but the encryption method is one-way



# Password Encryption

- If anyone knew how to reverse the password method, then:
  - a3R7nito5fo%r -> *comes from* -> 123456
- Fortunately it is very hard to crack the one-way encryption
- Problem: why is storing the password file publicly dangerous, and why is having a large encrypted password file stolen a problem?
  - A dictionary can be built of encryptions by turning the crank sequentially:
    - 123454 = JoF9#\$94(4k9!
    - 123455 = fj49#mc903#0Q
    - **123456 = a3R7nito5fo%**
    - 123457 = h9^wehf9\*3xd9



# Monitoring and Logs



- With all of the insecure protocols still in use (telnet, FTP), keep a tight eye on everything with log files:
  - Network
  - Account login/logout
  - Program usage
  - File access
  - Security checks
  - etc.

# Getting Root Access when you're not supposed to have it...

- Try the front door first:

*ACCOUNT: PASSWORD*

- **root: root**
- sys: sys / system / bin
- bin: sys / bin
- **mountfsys: mountfsys**
- adm: adm
- uucp: uucp
- nuucp: anon
- anon: anon
- user: user
- games: games
- **install: install**
- demo: demo
- **umountfsys: umountfsys**
- **sync: sync**
- admin: admin
- guest: guest
- daemon: daemon



# Getting Root Access when you're not supposed to have it...

- Assuming social engineering didn't work, you'll have to use fancy stuff:
  - Port scans + port/program insecurities
  - Buffer overflows (with system access)
  - Boot hacking (with physical access)
- Why are we talking about this stuff?
  - So you can protect yourself against it



# Breaking Into Windows

- Boot off of the installation media (Windows Server 2008 DVD, here)



<http://www.howtogeek.com/106333/how-to-reset-your-forgotten-domain-admin-password-on-server-2008-r2/>

# Breaking Into Windows

- Click here...



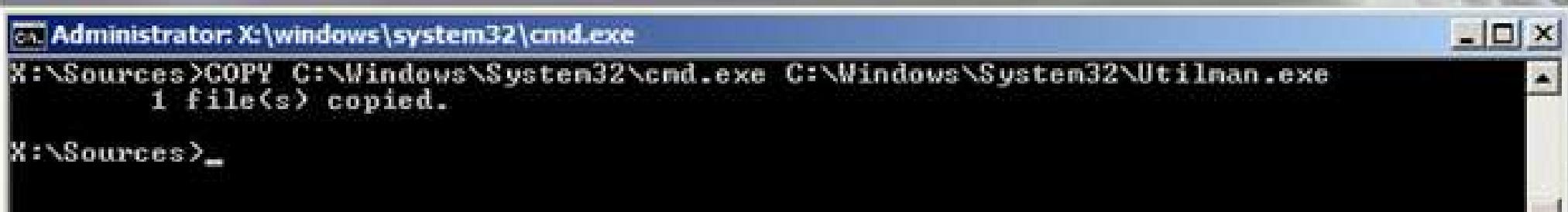
# Breaking Into Windows

- Enter these two commands...



```
Administrator: X:\windows\system32\cmd.exe
X:\Sources>MOVE C:\Windows\System32\Utilman.exe C:\Windows\System32\Utilman.exe.bak
 1 file(s) moved.

X:\Sources>_
```



```
Administrator: X:\windows\system32\cmd.exe
X:\Sources>COPY C:\Windows\System32\cmd.exe C:\Windows\System32\Utilman.exe
 1 file(s) copied.

X:\Sources>_
```

# Breaking Into Windows

- Reboot without CD, booting normally into Windows on the hard drive, then click here:



- Instead of accessibility, you get a privileged prompt! Change the password like this...

A screenshot of a Windows command prompt window titled "Administrator: C:\Windows\system32\utilman.exe". The window shows the command "C:\Windows\system32>net user administrator \*" being entered, followed by prompts for a password and confirmation. The command is completed successfully. The window has a standard Windows title bar and a close button.

```
C:\Windows\system32>net user administrator *
Type a password for the user:
Retype the password to confirm:
The command completed successfully.

C:\Windows\system32>
```

# Breaking Into Windows

- Log in using the new password!
- Remember to put the files back where you got them from
- Works in Windows 7, 8, 8.1, 10, and Server 2012, too!
- Why not create a few new local user accounts of your own, while you're in there?
- Q: Why can't we create domain accounts?



# Password Annihilator - with Physical Access

- Reset, change, or blank out any Windows password by booting from a flash drive or CD:
  - <http://pogostick.net/~pnh/ntpasswd/>

```
*****
*   Windows NT/2k/XP/Vista Change Password / Registry Editor / Boot CD
*
*   (c) 1998-2007 Petter Nordahl-Hagen. Distributed under GNU GPL v2
*
* DISCLAIMER: THIS SOFTWARE COMES WITH ABSOLUTELY NO WARRANTIES!
* THE AUTHOR CAN NOT BE HELD RESPONSIBLE FOR ANY DAMAGE
* CAUSED BY THE (MIS)USE OF THIS SOFTWARE
*
* More info at: http://home.eunet.no/~pnordahl/ntpasswd/
* Email      : pnordahl@eunet.no
*
* CD build date: Mon Apr  9 15:33:39 CEST 2007
*****
Press enter to boot, or give linux kernel boot options first if needed.
Some that I have to use once in a while:
boot nousb      - to turn off USB if not used and it causes problems
boot irqpoll     - if some drivers hang with irq problem messages
boot nodrivers   - skip automatic disk driver loading
boot: _
```



<http://www.techrepublic.com/blog/windows-and-office/reset-lost-windows-passwords-with-offline-registry-editor/>

# Apple and You?

- In August of 2012, Wired Magazine editor Mat Honan had his Apple account penetrated
- The perpetrators used Mat's Apple account to remotely erase all data on his iPhone, iPad, and MacBook
- This was accomplished by using what multiple companies knew about Mat to put together a complete profile

-Ariel Zambelich, Wired

# Who ARE you?

- The perps proved they were Mat, which let them reset Mat's Apple password, and then reset his equipment
- How can you prove you're Mat?
  - Apple says that Mat is the last four digits of his credit card
  - How do we get these last four digits?



# How to become Mat

1. Call Amazon, tell them you are the Account Holder
  - You'll need Name, Email address, Billing address
2. Add a credit card over the phone
3. Hang Up
4. Call Back, tell them you've lost access to your account
  - You'll need Name, Email address, Billing address, and a credit card number
  - They let you add a new email address: use yours
5. From the web, reset the password, using your new email
6. View the last four digits of the credit cards in the account



# Security Isn't Easy

- That's literally all this slide says



# OS Comparisons & Beyond

Benjamin Brewster

Except as noted, all images copyrighted with Creative Commons licenses,  
with attributions given whenever available

# What We'll Cover

- A brief history of some of the major Operating Systems that make up our current landscape
- Compare and contrast a few Operating Systems
- Even wax a bit philosophical about the past and future



# Short History of DOS - Business-Driven

- The setup: Bill Gates and Paul Allen were writing and selling Microsoft BASIC, a language for development, to early computer hobbyists
- IBM was just finishing up prototypes for their new PC and approached Microsoft to see if they had an OS suitable for the hardware in ~1980
- Microsoft didn't have an OS, so they quickly licensed 86-DOS from Seattle Computer Products (written by 24-year old Tim Paterson)
- Microsoft presented 86-DOS as “Microsoft DOS” to IBM, who accepted it; after a re-write, IBM and MS jointly retained copyright
- Microsoft purchases the rights to 86-DOS in 1981 from SCP; SCP later sues, claiming Microsoft hid their deal with IBM to get it cheap

# Short History of DOS - Business Driven

- IBM begins sales of its PCs with PC-DOS in 8/81
- Microsoft, however, begins sales of MS-DOS (the same thing as PC-DOS without IBM's specific drivers for its own PC) to other OEMs
- Eventually, Microsoft gains exclusive rights to DOS, and sells to all OEMS, securing their place in the market
- The last version of MS-DOS was 6.0, released 3/93 to huge sales and success

# Short History of Windows - To The Future

- Windows 1.0 was released November 20, 1985, and was not well received - it was mainly an overlay over DOS
  - Licensing requirements for Microsoft, which had created applications for Apple, enforced limits: windows could not overlap on screen!
  - Included Paint, Write, a terminal, MS-DOS, and Reversi, among others
- Apple and Microsoft began legal battles with Windows 2.03 and 3.0; the judge dropped all but 10 of Apple's 189 claims of infringement, and most of the 10 left were over uncopyrightable ideas

# Short History of Windows - To The Future

- Windows 3 (released May 1990) was wildly successful, selling around 10 million copies even before 3.1 came out; support ended in 2001
- Support for 32-bit software began with Windows 95, released 8/24/95
- Followed by: OS/2, NT, 98, 2000, ME, XP, Server 2003, XP 64, Home Server, Vista, Server 2008, 7, Server 2008 R2, Home Server 2011, Thin PC, 8, Server 2012, 10, Server 2016

# Short History of Windows - The Wealthiest



- Forbes list of billionaires in 2016:
  1. Bill Gates, 75 B
  2. Amancio Ortega, 67 B
  3. Warren Buffett, 60 B
  4. Carlos Slim, 50 B
  5. Jeff Bezos, 45 B
  6. Mark Zuckerberg, 44 B
  7. Larry Ellison, 43 B
  8. Michael Bloomberg, 40 B
  9. Charles Koch, 39 B
  10. David Koch, 39 B

# Short History of macOS - Inspired

- Steve's Wozniak and Jobs found Apple Computer Inc. on 1/3/77, immediately beginning to make computers of Wozniak's design
- Apple I sells 200 units, before Apple is founded, but the Apple II, launched in April 1977 goes on to sell millions well into the 1980s
- Apple IPO on 12/12/80 generates more capital than any IPO since Ford in 1956, instantly creates more millionaires than any company in history
- Apple visits Xerox PARC, comes back with ideas for Apple's first GUI-based computer, the Lisa, which introduces the words mouse, desktop, and icon; Lisa costs \$10K in 1983 (\$23.8K in 2016) and fails

# Short History of macOS - Inspired

- Subsequent computers are successful running early versions of the Mac OS, though Jobs and Wozniak both leave in 1985
- Jobs founds NeXT, which produces the UNIX-based NeXTstep OS on the computers it sells; this OS uses the Mach 2.5 kernel and subsystems from BSD 4.3
- In 1996, Apple buys NeXT, as a result of Jobs influence, beating out rival BeOS, which sees Jobs return as CEO to Apple!
- Microsoft donates \$150 million to Apple to call an end to the decades-long legal war between the companies
- NeXTstep becomes the foundation of OSX, which launches in 2001 as version 10.0 “Cheetah”
- OSX morphs into macOS version 10.12 “Sierra” in 2016

# Short History of iOS - Revolutionary

- When Jobs returns to Apple, he brings with him the WebObjects Application server from NeXT, which turns into the App Store
- Internally at Apple, OS X is forked into iOS: it's still Mach and BSD underneath it all
- Capacitive sensors are added to the screen, and an entirely new user paradigm is created: touch, hiding the file system, and a simple web browser
- iOS version 1 (initially called “iPhone OS”) launches with the iPhone in January 2007; App Store doesn't launch until July 2008
- Apple unit sales in 2016 across their three major divisions:
  - iPhone: 211.88 million
  - iPad: 45.59 million
  - Mac: 18.48 million



Mobile devices (iPhone, iPad) make up 93% of Apple's unit sales in 2016

# Short History of Android - Playing Catch-Up

- Android, Inc. was founded in Palo Alto in October 2003
- Initial smartphone plans were to compete with Symbian and Windows Mobile using a custom fork of Linux
- In June 2005, Google purchased Android, Inc. for at least \$50 million
- The earliest prototypes of Android resembled BlackBerry's OS with a full QWERTY keyboard layout; after Apple's announcement and rollout of the iPhone in 2007, those visual layouts and keyboard plans were scrapped
- The first phone with Android was the HTC Dream, released on 10/22/2008

# Short History of Android - Victory

- Fast forward to today, Android is not just the dominant mobile OS, it's the dominant OS in all sales of devices worldwide
- In 2015 sales, according to Gartner Research, Android's market share is unrivaled:
  - **Android:** 1.3 billion devices (54%)
  - **iOS/OS X combined:** 297 million devices (12.3%)
  - **Windows:** 283 million (11.7%)
  - **All others:** ~520 million (21.6%)



# OS Internal Architecture - Booting with BIOS

- Before we can talk about the procedures being followed in the kernel, we need to understand the first stages of how modern OSs boot
- In ye olden days (pre-2014) a switched-on computer would first load the **BIOS** (Basic Input and Output System), which:
  - Performed hardware initialization, including testing
  - Loaded a full-featured boot loader, which may simply load an OS itself from mass memory device, or provide the user a selection of OSs to boot from
  - Provided an abstracted, consistent method of getting data to and from input and output devices

# OS Internal Architecture - Booting with UEFI

- Modern PCs, when switched on, now first load the **UEFI** (Unified Extensible Firmware Interface) firmware
- The UEFI is a mini-OS that provides many more features:
  - Boot from large disks (up to 8 ZiB) with a modern GUID Partition Table (GPT)
  - Full-featured pre-boot environment, including a mouse and keyboard-driven GUI, audio, network access, hardware testing, and IT management
  - Booting the OS of your choice from all registered systems
  - Optionally, preventing unsigned drivers from being used in the booting of an OS, which prevents firmware-level rootkits (and possibly locking hardware to a specific OS)

# OS Internal Architecture - Booting with UEFI

- Modern PCs, when switched on, now first load the **UEFI** (Unified Extensible Firmware Interface) firmware
- The UEFI is a mini-OS that provides many more features:
  - Boot from large disks (up to 8 ZiB) with a modern GUID Partition Table (GPT)
  - Full-featured pre-boot environment, including a mouse and keyboard-driven menu

One megabyte	= 1,000,000 bytes	= $1000^2$ bytes
One gigabyte	= 1,000,000,000 bytes	= $1000^3$ bytes
One terabyte	= 1,000,000,000,000 bytes	= $1000^4$ bytes
One petabyte	= 1,000,000,000,000,000 bytes	= $1000^5$ bytes
One exabyte	= 1,000,000,000,000,000,000 bytes	= $1000^6$ bytes
One zettabyte	= 1,000,000,000,000,000,000,000 bytes = 1,000,000,000 gigabytes	= $1000^7$ bytes
One zebibyte	= 1,180,591,620,717,411,303,424 bytes	= $1024^7$ bytes

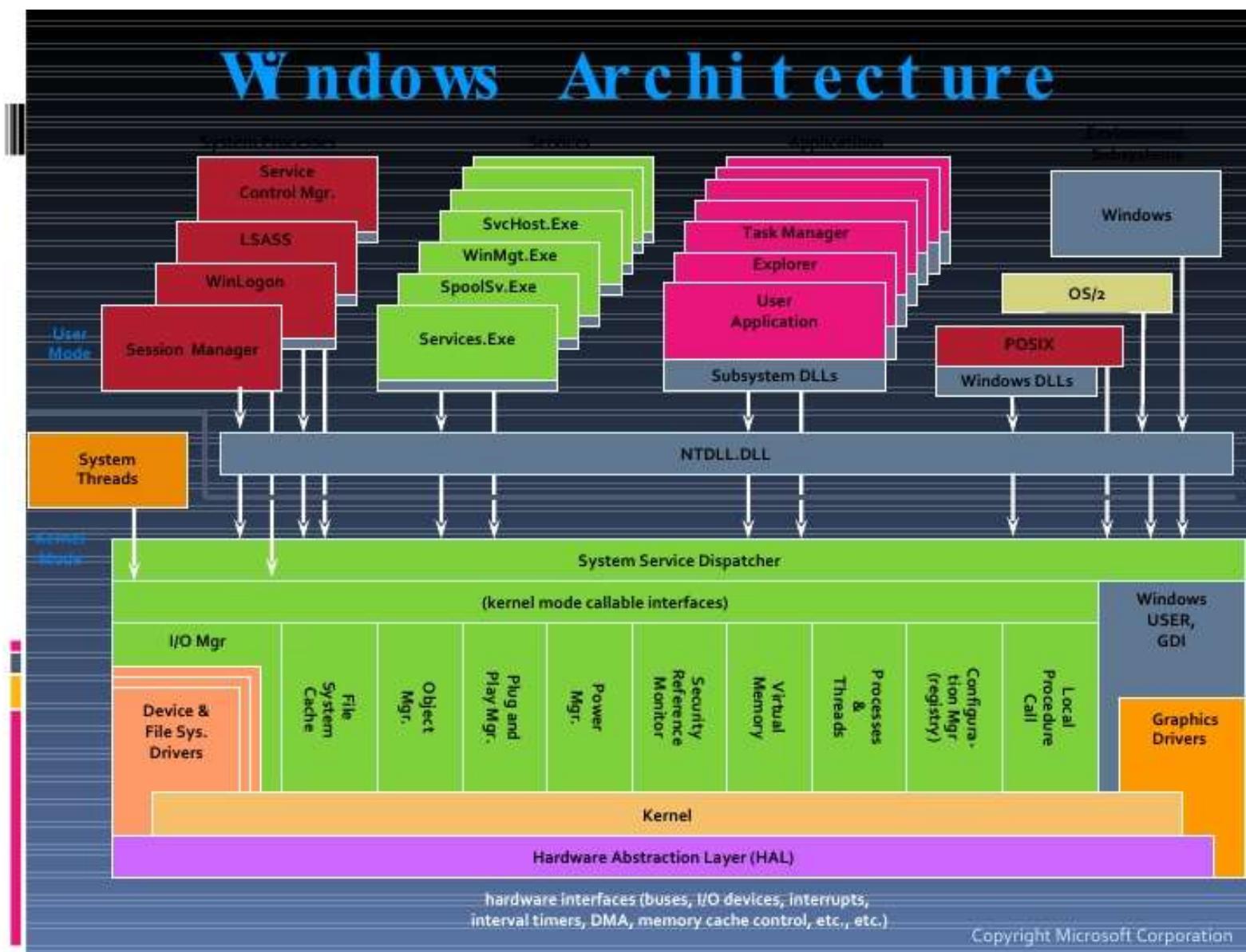
# Windows Internal Architecture

- User programs can access the OS by using the Windows API, of which there are many versions: Win32, Win64, etc.; backwards compatibility is a major focus, allowing many of these APIs to be accessible at the same time
- The user has deep access to low-level aspects of the GUI, which can cause simple programs to be large if API libraries such as .NET or DirectX are not used
- A large set of services run on top of the kernel providing access to various features of Windows to user programs

# Windows Internal Architecture

- The Registry: a hierarchical database of configuration settings for Windows and Applications
  - Used by the OS itself (kernel), drivers, services (what Windows calls daemons), the security system (SAM), and UI
  - Applications can use the registry if they wish, but could also use standard configuration files in the file system
- Security Account Manager
  - An encrypted database file that stores the hashed passwords and local accounts
  - Cannot be messed with while Windows is running (though in-memory passwords can be dumped); can be edited when Windows is not running (during a Linux boot off a live CD, for example);

# Windows Internal Architecture



# Windows Boot Procedure

- Power On
- UEFI program executed
- The Windows bootloader Winload.exe loads basic drivers required to read data from disk
- Kernel is initialized
- Registry and non-boot drivers are loaded and started
- Winlogon.exe starts, requiring the user to login
- Upon successful login, explorer.exe is started
- Desktop window manager (DWM) is started

# macOS Internal Architecture - Kernel Dev Origins

- The UNIX concept and implementation of **pipes**, which allows data to be moved between many small interacting programs, also causes/implements **blocking system calls**, which causes data to be moved in staggered pieces around the system
- Thus, the implementation of pipes as memory buffers, which copy so much data around, does not scale well when fast speeds and low latency are desired
- So the natural response in the 1980s was to write the **kernel** and core functionality as a single large, unwieldy program to prevent copying
- These large kernels are bug-prone; small interacting pieces are *much* easier to test and verify independently!

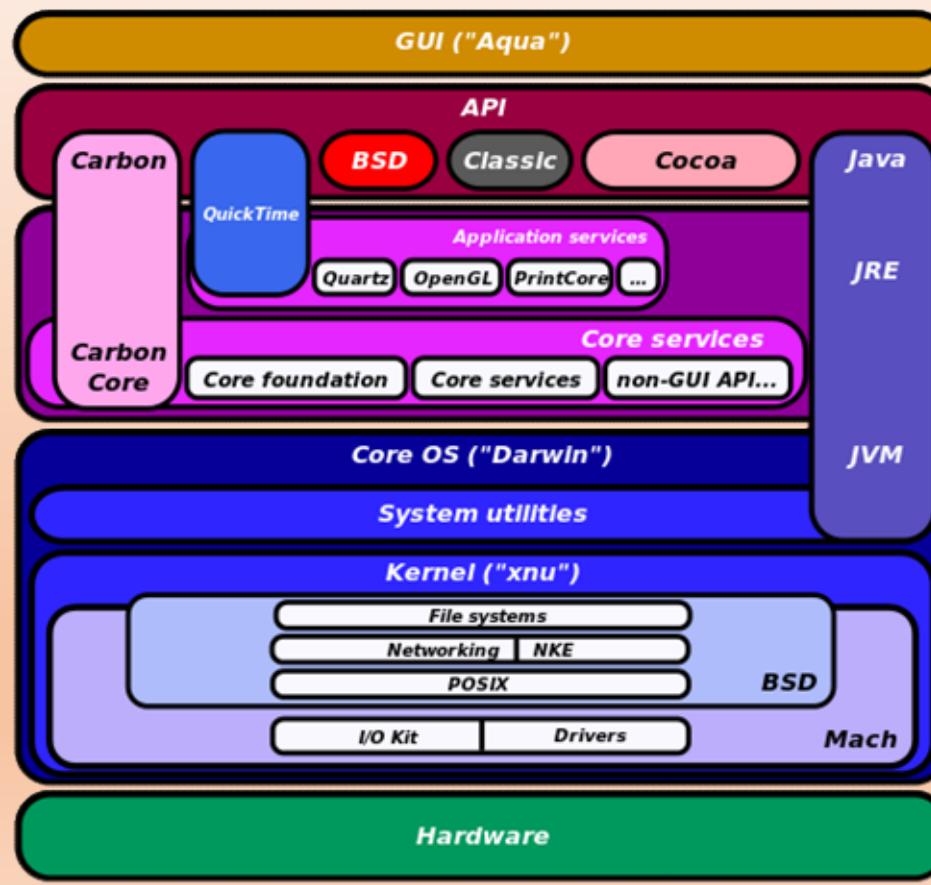
# macOS Internal Architecture - Kernel Dev Outcome

- New theories of kernel design were tested out in the Aleph kernel developed at University of Rochester in 1975, where the kernel was shrunk to handle only access to hardware, including memory: thus, it uses a form of shared memory to copy information around between easily tested subsystem programs
- New CPUs developed in the early 80s offer support for a **Memory Management Unit** (MMU), which allows for virtual memory to be implemented, tracking the pages of memory in use by processes
- This allows memory that is told to be “copied” to instead be transparently *referenced virtually*, dramatically reducing the amount of data being copied by the kernel; this is called **copy-on-write** (*i.e., copy only if you’re going to make a modification*)
- These new, smaller kernels are called **microkernels**

# macOS Internal Architecture

- macOS uses the Mach microkernel paired with BSD programs to provide system call access
- User programs can access macOS by using the standard, tried-and-true UNIX API system calls we've been studying, in addition to other API libraries that access additional functionality of the OS
- Drivers are abstracted as one form of “kernel extensions”, which are all loaded when the OS boots - thus trusted, low-level code can be added to the kernel by third-parties

# macOS Internal Architecture



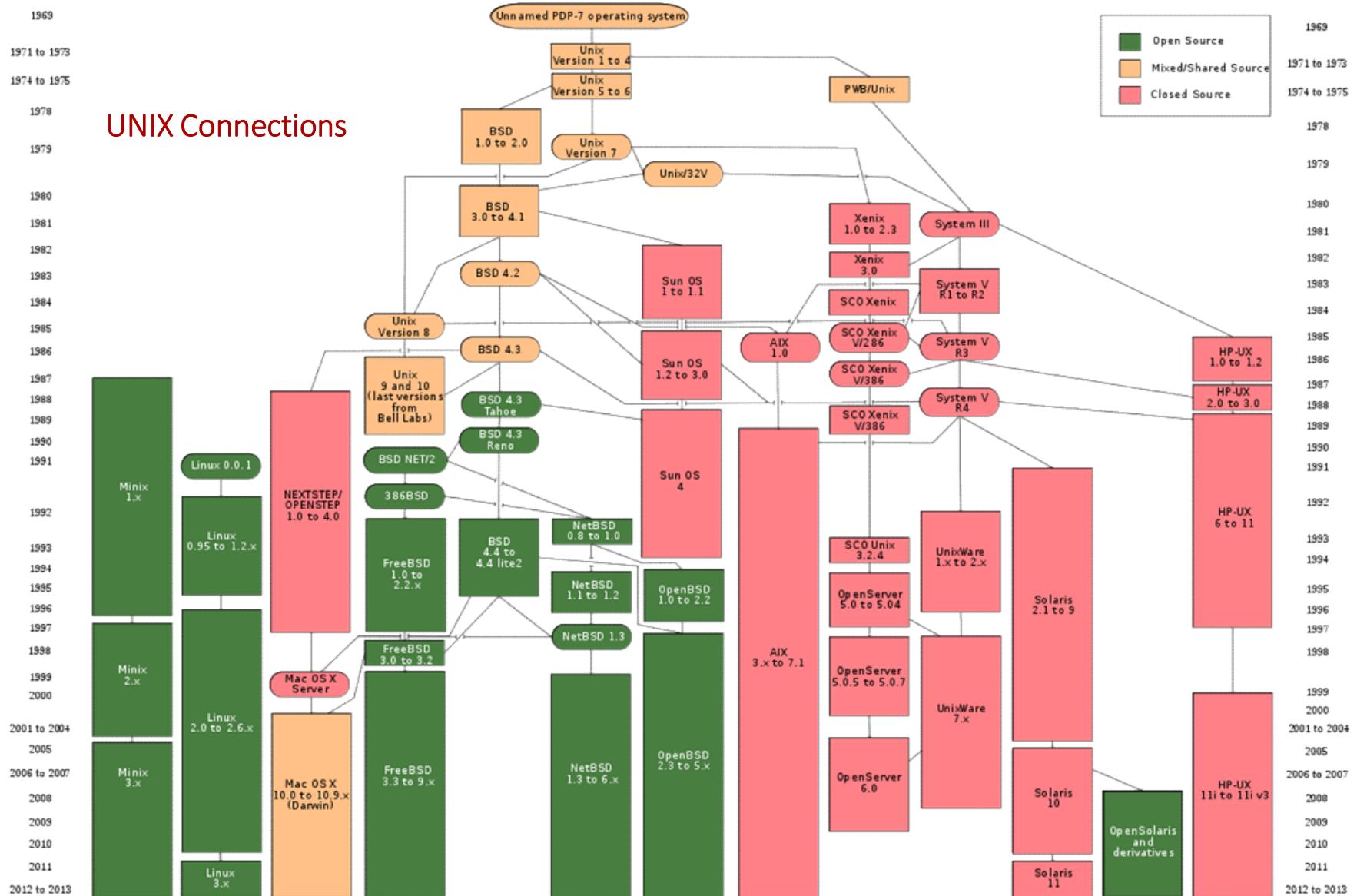
CC BY-SA 3.0  
[https://upload.wikimedia.org/wikipedia/commons/f/f2/Diagram\\_of\\_Mac\\_OS\\_X\\_architecture.svg](https://upload.wikimedia.org/wikipedia/commons/f/f2/Diagram_of_Mac_OS_X_architecture.svg)

# macOS Boot Process

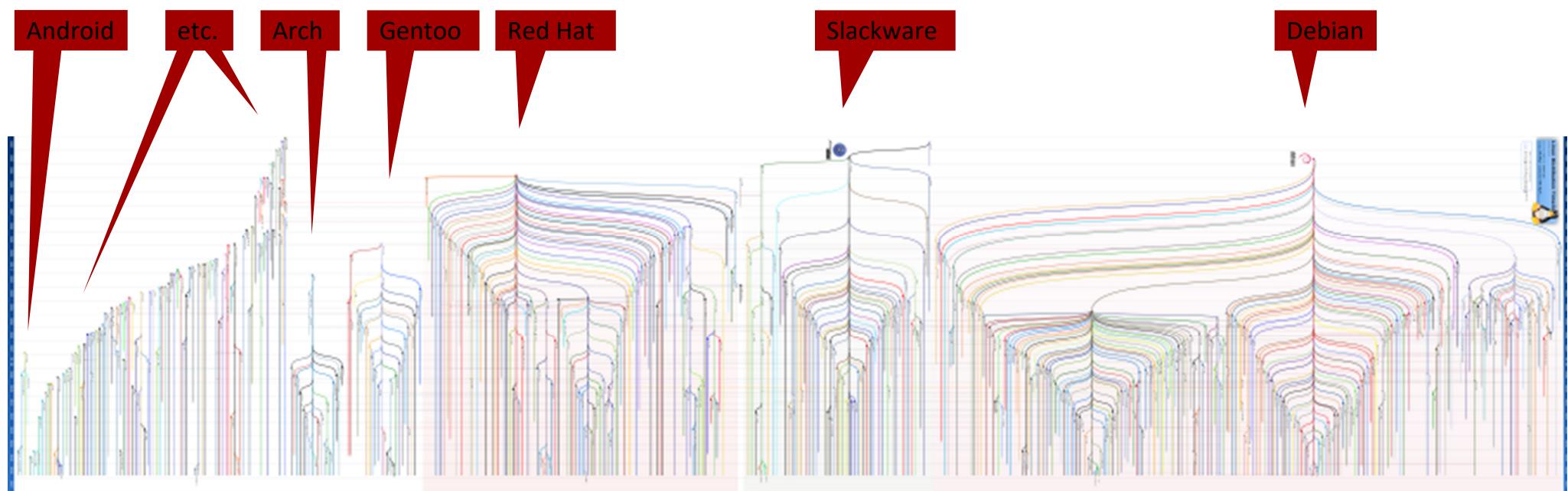
- Power On
- UEFI program executed
- Control passed to the BootX bootloader
- BootX loads previously cached list of kernel extensions, including drivers
- init process in Mach microkernel is started; control has left the firmware
- Mach and BSD data structures are initialized
- I/O starts up
- Kernel starts virtual memory management tracking routines
- Kernel starts GUI and other daemons/services

# The Importance Of UNIX

- UNIX is run on a vast majority of devices around the world
  - It has generated fortunes, saved lives, and been an agent for positive change for millions and millions of people
  - Today, you can even run a bash shell in Windows 10
- 
- Let's take a look back at the historical connections between UNIX, Linux, and all the variants



# Linux Connections - Go Check This Out



CC BY-SA 3.0 - [https://en.wikipedia.org/wiki/List\\_of\\_Linux\\_distributions#/media/File:Linux\\_Distribution\\_Timeline.svg](https://en.wikipedia.org/wiki/List_of_Linux_distributions#/media/File:Linux_Distribution_Timeline.svg)

# A Few Early Pioneers

- Tommy Flowers designed Colossus (1944), the world's first programmable electronic computer to help decrypt German wartime messages
- J. Presper Eckert & John Mauchly designed and built the ENIAC (1946), the first all electronic, Turing-complete computer, and the UNIVAC I (1951), the first commercially available computer
- Margaret Hamilton coined the phrase “software engineering”, instrumental in testing and timing-critical human interaction with computers; led dev of on-board software used in Apollo missions
- One of the first OSs with a software-based paged virtual memory was THE, designed by a team led by Edsger Dijkstra way back in 1962

# Edsger Dijkstra (1930 – 2002)

## Contributions

- “Dijkstra’s Algorithm”
- Semaphores, including the Dining Philosopher Problem, deadlock, and other synchronization issues
- Operating system design, including abstraction layers
- Compiler design (he wouldn’t shave his beard until he had created the first ALGOL 60 compiler, then decided to keep it)
- Software engineering, including the paper, “A Case against the GO TO Statement”, helping shape programming as a discipline, not an ad hoc craft
- Distributed computing
- Formal specs and verification, and how to simplify them; CS cast as math



Picture by Hamilton Richards - manuscripts of Edsger W. Dijkstra, University Texas at Austin.  
(Mirrored), CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=46866934>

# THE's layers

- Dijkstra's THE system had these layers all the way back in 1968:
  - Layer 0: Scheduler
  - Layer 1: Memory pager
  - Layer 2: Communication between OS and terminal
  - Layer 3: Managed IO between attached devices
  - Layer 4: User Programs
  - Layer 5: The User (Dijkstra says, “not implemented by us”)
- What will you, dear student, do with the implementation of yourself?

# Edsger Dijkstra Quotes

- "Brainpower is by far our scarcest resource."
- "The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague."
- "Simplicity is prerequisite for reliability."
- "Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians."

# Edsger Dijkstra Quotes

There may also be political impediments [to becoming an exceptional programmer]. Even if we know how to educate tomorrow's professional programmer, it is not certain that the society we are living in will allow us to do so. The first effect of teaching a methodology — rather than disseminating knowledge — is that of enhancing the capacities of the already capable, thus magnifying the difference in intelligence. In a society in which the educational system is used as an instrument for the establishment of a homogenized culture, in which the cream is prevented from rising to the top, the education of competent programmers could be politically impalatable.

Let me conclude. Automatic computers have now been with us for a quarter of a century. They have had a great impact on our society in their capacity of tools, but in that capacity their influence will be but a ripple on the surface of our culture, compared with the much more profound influence they will have in their capacity of intellectual challenge without precedent in the cultural history of mankind.

ACM Turing Lecture 1972

# Edsger Dijkstra Quotes

There may also be political impediments [to becoming an exceptional programmer]. Even if we know how to educate tomorrow's professional programmer, it is not certain that the society we are living in will allow us to do so. **The first effect of teaching a methodology — rather than disseminating knowledge** — is that of enhancing the capacities of the already capable, thus magnifying the difference in intelligence. In a society in which the educational system is used as an instrument for the establishment of a homogenized culture, in which the cream is prevented from rising to the top, the education of competent programmers could be politically impalatable.

Let me conclude. Automatic computers have now been with us for a quarter of a century. They have had a great impact on our society in their capacity of tools, but in that capacity their influence will be but a ripple on the surface of our culture, compared with the much more profound influence they will have in their capacity of intellectual challenge without precedent in the cultural history of mankind.

ACM Turing Lecture 1972

We are teaching you the theories so that you can acquire any skill

# Edsger Dijkstra Quotes

There may also be political impediments [to becoming an exceptional programmer]. Even if we know how to educate tomorrow's professional programmer, it is not certain that the society we are living in will allow us to do so. The first effect of teaching a methodology — rather than disseminating knowledge — is that of enhancing the capacities of the already capable, thus magnifying the difference in intelligence. **In a society in which the educational system is used as an instrument for the establishment of a homogenized culture, in which the cream is prevented from rising to the top, the education of competent programmers could be politically impalatable.**

Let me conclude. Automatic computers have now been with us for a quarter of a century. They have had a great impact on our society in their capacity of tools, but in that capacity their influence will be but a ripple on the surface of our culture, compared with the much more profound influence they will have in their capacity of intellectual challenge without precedent in the cultural history of mankind.

ACM Turing Lecture 1972

We will reward winners and victors; life doesn't hand out participation trophies

# Edsger Dijkstra Quotes

There may also be political impediments [to becoming an exceptional programmer]. Even if we know how to educate tomorrow's professional programmer, it is not certain that the society we are living in will allow us to do so. The first effect of teaching a methodology — rather than disseminating knowledge — is that of enhancing the capacities of the already capable, thus magnifying the difference in intelligence. In a society in which the educational system is used as an instrument for the establishment of a homogenized culture, in which the cream is prevented from rising to the top, the education of competent programmers could be politically impalatable.

Let me conclude. Automatic computers have now been with us for a quarter of a century. **They have had a great impact on our society in their capacity of tools, but** in that capacity their influence will be but a ripple on the surface of our culture, compared with the **much more profound influence they will have in their capacity of intellectual challenge** without precedent in the cultural history of mankind.

ACM Turing Lecture 1972

Go do something useful!

Improve life, don't just live it!