Administrative Details

- Web server is due Wed, Feb 22 at 11:59pm
  - Milestone 1 due Feb 13 at 11:59pm…START SOON
  - Paper due Feb 23 at 11:59pm
  - Firewall prevents non-CS access to lab machines
  - Develop anywhere you want, but test on lab machines
  - Use ports 8000+

- First paper eval is due Thur (on GLOW)
  - Classic systems paper describes the end-to-end argument
  - Summarize and make observations (might be hard to be critical at this point…that's ok)
  - Remember that it was written in 1984(!)…
Last Time

- Course overview
- Learned a bit about the history of the Internet
Today's Outline

• Brief overview of sockets in C/C++
  • Most other languages have sockets too
• Threads vs. processes
  • Very brief discussion of their tradeoffs (more in OS!)
• Learn basic terminology used in network characterization
  • Bandwidth, latency, bw-delay product, error rate
• Discuss OSI layer model for conceptualizing network design
Project 1 Overview
Distributed Systems Architectures

- Popular dist system design models:
  - Client-server
    - Most common, and arguably the simplest
  - Multi-tier client-server
    - Variation on the simple client-server architecture
    - Multiple levels of communication, separation of functionality
  - Peer-to-peer
    - All processes involved in a task or activity play similar roles
    - All "peers" are created equal
    - All peers are both clients and servers
Clients and Servers

• Server tasks
  • Listen, accept, receive req, send resp, repeat indefinitely

• Client tasks
  • Connect, send req, receive resp, close
Implementing a Server

• How to create network connections?
  • Sockets!
  • Java uses ServerSockets (and Sockets for clients)
  • Python and C/C++ make no distinction between client and server connections
Review: File Descriptors

• File descriptors
  • Most I/O on Unix-based systems take place through **system calls** using **file descriptors**
    • Non-negative integers created using system calls (like `open()`, `socket()`, `accept()`)
    • FDs remain bound to files/sockets even when renamed or deleted
  • Read/write and send/recv ops are subsequently performed on these file descriptors
• Popular file descriptors
  • 0 = stdin, 1 = stdout, 2 = stderr
File I/O

- System calls are essentially function calls that request something from the OS

- System calls for file I/O:
  - int open(char *path, int flags, ...)
  - int read(int fd, void *buf, int nbytes)
  - int write(int fd, void *buf, int nbytes)

- Example using files
  - Read from "a.txt" and write to stdout
  - lecture2.c
int main(int argc, char **argv) {

    int fd, nread;
    char buf[1024];     //C has no "string" class

    fd = open("a.txt", O_RDONLY);

    if (fd == -1) {
        printf("Error opening file.\n");
        return -1;           //error opening file
    }

    while ((nread = read(fd, buf, sizeof(buf))) > 0) {
        //recall that stdout has fd=1
        write(1, buf, nread);
    }

    close(fd);
    return (0);
}
Introduction to Sockets

- Sockets are abstractions for network connections
- Intended to make network I/O look like file I/O
- Create new sockets using `socket()` sys call
  - Just like `open()` for files
  - Returns a file descriptor
    - `int socket(int domain, int type, int protocol)`
- For TCP over IP (common combo):
  - `domain = AF_INET`
  - `type = SOCK_STREAM`
  - `protocol = 0, or IPPROTO_TCP`
- Unlike files, sockets require more info to be functional
  - They need an IP address and port!
**Bind**

- **The `bind()`** system call sets local IP address and port information.
  - `int bind(int s, struct sockaddr *addr, int addrlen)`
  - `s` = socket file descriptor (returned from `socket()`)
  - `addr` = sockaddr_in structure (see below)
  - `addrlen` = size of `addr` (just use `sizeof(addr)`)

- **struct sockaddr_in comes with `netinet/in.h`**
  ```c
  struct sockaddr_in {
      short sin_family;    ------>AF_INET
      u_short sin_port;    ------>htons(8888)
      struct in_addr sin_addr;    ------>htonl(INADDR_ANY)
  }
  ```

- **Example:**
bind example

```c
struct sockaddr_in myaddr;
int sock;

myaddr.sin_port = htons(8888);
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(sock, (struct sockaddr*)&myaddr, sizeof(myaddr))<0) exit(1);
```
Listen

- **bind()** “binds” the socket to a specific port
  - Like setting up your store at a specific address
- Also need to **listen** for clients
  - Opens store for business
- **Use listen() to tell the OS to accept client network connections**
  - `int listen(int s, int backlog)`
    - `s` = socket file descriptor (returned from socket())
    - `backlog` = max # of unaccepted connections (use 10)
- Listening sockets have no “remote end”
- The OS creates new (separate) sockets for each incoming client
Accept

We use the accept system call to *block* and wait for new clients (why block?)

- int accept(int s, struct sockaddr *addr, int *addrlenp)
  - s = socket file descriptor (returned from socket())
  - addr is a struct sockaddr_in* that gets "filled in" automatically with client info
  - addrlenp is a pointer to an integer containing the value sizeof(struct sockaddr_in)

- `accept()` returns (after client connects) with a new file descriptor that represents the new network connection to the client

- Clients use `connect()` instead of `accept()` because they are *initiating* the connection

For reading and writing from sockets:

- int `send(int socket, char *msg, int msglen, int flags)`
- int `recv(int socket, char *buf, int buflen, int flags)`
Clients and Servers

• How do servers handle multiple clients simultaneously?
  • We don't want to consume the server's resources with just one client…

• Three choices for multi-tasking
  • Multiple threads
  • Multiple processes
  • Event queue
Threads and Processes

- In OS, we discuss threads and processes for weeks
  - The textbook discusses them briefly
  - Also mention them in 237

- A process is a single activity that a processor can execute (see `fork()`)
  - "Heavyweight," independent tasks, consist of a private address space, only interact with other processes through inter-process communication

- A thread is a "thread of execution" (pthreads)
  - "Lightweight," shares state information (like address space) with other threads within a single process, easily interacts with other threads since memory is shared
Threads and Processes

- Execution environments
  - Consist of an address space, thread synchronization and communication resources (sockets), higher-level resources like open files
- Each process has its own separate execution environment
- Threads share a single execution environment
- Tradeoffs?
Threads and Processes

• Benefits of using threads instead of processes
  • Low communication/context switching overhead since execution environment/address is shared
  • Easy to take advantage of parallelism in multi-processor computers

• Disadvantages of threads
  • Shared resources can lead to synchronization problems (classic bank acct example)
  • Need to be careful to avoid deadlocks
  • Need to provide atomic operations
  • Architecture specific (not as "portable" as processes)
Creating Processes

• In C/C++, processes are created using the fork() system call
• fork() makes a duplicate of the current process' execution environment
• Creates a new (child) process and returns twice (once in child, once in parent)!

```c
int fork(void)
pid = fork();
if (pid==0) {
    //child process
} else {
    //parent process
}
```
Creating Threads

- pthreads (POSIX threads) are created using `pthread_create()`
  ```c
  int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg)
  ```
- `const pthread_attr_t *attr` is usually NULL
- `void *(*start_routine)` is a function pointer to the function that the new thread will execute when it starts
- `void *arg` are any arguments that the functions needs
- Successful thread creation returns 0
Event Driven Architectures

- There are several options
- Event-driven architectures are usually single threaded (sometimes there are ~2 threads)
- Maintain a list of active clients/socket connections
- As clients arrive/leave, update list
- Loop through active connections servicing each one equally
- Benefits: No serious synchronization issues, no context switching
- Check out `select()` system call
Things to Watch Out For

- Sockets/ports are "already in use"
  - Just pick a new port for a few minutes…
- Leaving ports open indefinitely
  - This is really bad!
- Compiling/linking with threads usually requires `-lpthread`
Helpful Hints

- (telnet demo)
- Use Google (but don't search for solution!)
- Check out the links on the assignment page
- Start small and test often
- Learn how to use a Makefile
- Use "man" and "man 2" for understanding system calls
- Use telnet or nc for preliminary testing and experimentation
Moving on…

Networks Crash Course Part I:

How is the Network Characterized?
Some Definitions

- **Host** – computer, iPhone, smart watch, sensor, ...
- **Packet** – unit of transmission across a network
- **Link** – used to transmit bits, aka "pipe"
  - Wired or wireless, broadcast or switched (or both)
- **Switch** – used to move bits between links
  - Packet switching: stateless, store & forward
  - Circuit switching: stateful, cut through
Simple Network Model

- Network is a “virtual pipe” connecting two hosts

- Basic Metrics
  - Bandwidth, latency, overhead, error rate, and message size
Performance Metrics

- Bandwidth: number of bits transmitted per unit of time
- Total latency (time) = Propagation + Transmit + Queue
  - Propagation delay = Distance/SpeedOfLight
  - Transmit time = Size/Bandwidth
  - Queue delay = Time packets spend in router queues (often negligible)
- Overhead
  - # secs for CPU to put message on wire (usually negligible)
- Error rate
  - Probability P that message will not arrive intact
- Message size
  - Avg size of packets (in bits or bytes)
Bandwidth vs. Latency

1 Byte Object

<table>
<thead>
<tr>
<th></th>
<th>Prop delay: 1 ms</th>
<th>Prop delay: 100 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth: 1 Mbps</td>
<td>1,008 µs</td>
<td>100,008 µs</td>
</tr>
<tr>
<td>Bandwidth: 100 Mbps</td>
<td>1,000 µs</td>
<td>100,000 µs</td>
</tr>
</tbody>
</table>

10 MB Object

<table>
<thead>
<tr>
<th></th>
<th>Prop delay: 1 ms</th>
<th>Prop delay: 100 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth: 1 Mbps</td>
<td>80.001 s</td>
<td>80.1 s</td>
</tr>
<tr>
<td>Bandwidth: 100 Mbps</td>
<td>.801 s</td>
<td>.9 s</td>
</tr>
</tbody>
</table>

- Desirable network qualities based on application semantics
Types of Connections

- Point-to-point

- Multiple access
Types of Connections

- Switched
Types of Connections

• Interconnection of networks
Internet = Network of Networks

- Network delivers packets (and locates hosts)
- Router (gateway) moves packets between networks
- Software layer: IP interoperability on top of any potential network or link layer
  - Wifi (802.11), Ethernet, token ring, cell phone, ADSL, cable modem, smoke signals, ...
- IP imposes minimum possible requirements on underlying networks
Network Layering
Layering in Network Design

• Computer system design is often about defining the right set of abstractions
  • What services do apps require from the bare hardware?
    Don't force each app to re-implement the same functionality
  • Operating systems are all about abstractions
    What set of services do you put on top of the hardware?
  • Network cards/links hand a frame to the operating system

• But what abstraction does the application desire?
• Do all applications need the same abstraction?
• What abstractions do intermediate hosts (routers) in the network need?
Layering in Network Design

- Encapsulate each layer in lower-layer format
- Ethernet's view of the world
  - Header | Payload
  - IP View
    - Header | Payload
  - TCP View
    - Header | Payload
  - Reality
    - Eth | IP | TCP | HTTP | Payload