

## **Office Hours (also posted)**

- **Aaron: Mon 6pm-8pm in the Bird Cage - 2nd floor**
- **Chris: Tue 4pm-6pm CIT423**
- **Me: CIT379. Would you like set times?**

## Using TCP/IP

- We learned what TCP & IP do. How can your programs use them?
- The *sockets* API. POSIX specific, but similar everywhere.
- After basic setup, I/O much like files.
- How can you handle multiple connections?

# System calls

- **Problem: How to access resources other than CPU**
  - Disk, **network**, terminal, other processes
  - CPU prohibits instructions that would access devices
  - Only privileged OS “kernel” can access devices
- **Applications request I/O operations from kernel**
- **Kernel supplies well-defined *system call* interface**
  - Applications set up syscall arguments and *trap* to kernel
  - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `gets`, etc. all user-level code

## I/O through the file descriptors

- Applications “open” files/devices/**sockets** by name
  - I/O happens through integer handles to open resources.
- `int open(char *path, int flags, ...);`
- Returns file descriptor—used for all I/O to file

## Error returns

- **What if open fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int errno
- **#include <sys/errno.h> for possible values**
  - 2 = ENOENT “No such file or directory”
  - 13 = EACCES “Permission Denied”
- **perror function prints human-readable message**
  - perror ("initfile");  
→ “initfile: No such file or directory”

# Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, void *buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - whence: 0 – start, 1 – current, 2 – end
  - Returns previous file offset, or -1 on error
- `int close (int fd);`
- `int fsync (int fd);`
  - Guarantee that file contents is stably on disk

# Creating/monitoring processes

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns twice!
    - In parent: *process ID* of new process
    - In child: 0
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – process to wait for, or -1 for any
  - `stat` – will contain exit value, or signal
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error

# Deleting processes

- `void exit (int status);`
  - Current process ceases to exist
  - status shows up in `waitpid` (shifted)
  - By convention, status of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM` most common value, kills process by default (but application can catch it for “cleanup”)
  - `SIGKILL` stronger, kills process always

# Pipes

- `int pipe (int fds [2] );`
  - Returns two file descriptors in `fds [0]` and `fds [1]`
  - Writes to `fds [1]` will be read on `fds [0]`
  - When last copy of `fds [1]` closed, `fds [0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read/write/close` – as with files
  - When `fds [1]` closed, `read (fds [0] )` returns 0 bytes
  - When `fds [0]` closed, `write (fds [1] )`:
    - Kills process with SIGPIPE, or if blocked
    - Fails with EPIPE
- **For example code, see `pipesh.c` on web site**

# Sockets: Communication between machines

- **Datagram sockets: Unreliable message delivery**
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: send/recv
- **Stream sockets: Bi-directional pipes**
  - With IP, gives you TCP
  - Bytes written on one end read on the other
  - Reads may not return full amount requested—must re-read

# Socket naming

- **Recall how TCP & UDP name communication endpoints**
  - 32-bit IP address specifies machine
  - 16-bit TCP/UDP port number demultiplexes within host
  - Well-known services “listen” on standard ports: finger—79, HTTP—80, mail—25, ssh—22
  - Clients connect from arbitrary ports to well known ports
- **A *connection* can be named by 5 components**
  - Protocol (TCP), local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP

# System calls for using TCP

## Client

socket – make socket

bind\* – assign address

connect – connect to listening socket

## Server

socket – make socket

bind – assign address

listen – listen for clients

accept – accept connection

\*This call to bind is optional; connect can choose address & port.

# Client interface

```
struct sockaddr_in {
    short    sin_family;   /* = AF_INET */
    u_short  sin_port;     /* = htons (PORT) */
    struct   in_addr sin_addr;
    char     sin_zero[8];
} sin;
```

```
int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (13); /* daytime port */
sin.sin_addr.s_addr = htonl (IP_ADDRESS);
connect (s, (sockaddr *) &sin, sizeof (sin));
```

## Byte order

- When sending/receiving multibyte integers, which byte is sent first?
- Big-endian: first byte is the most significant
- Little-endian: you can guess, right?
- Processors differ, so protocols must pick.
- TCP/IP is Big-endian. x86 is not.
- Remember to convert! `htons()`, `ntohl()`, *etc*

# Server interface

```
int s = socket (AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin;
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (9999);
sin.sin_addr.s_addr = htonl (INADDR_ANY);
bind (s, (struct sockaddr *) &sin, sizeof (sin));
listen (s, 5);

for (;;) {
    socklen_t len = sizeof (sin);
    int cfd = accept (s, (struct sockaddr *) &sin, &len);
    /* cfd is new connection; you never read/write s */
    do_something_with (cfd);
    close (cfd);
}
```

# A fetch-store server

- **Clients sends commands, gets responses over TCP**
- **Fetch command**
  - Command consists of string "fetch\n"
  - Response contains last contents of file stored there
- **Store command**
  - Command consists of "store\n" followed by file
  - Response is "OK" or "ERROR"
- **What if server or network goes down during store?**
  - Don't say "OK" until data safely on disk (c.f. email)
- **Example: fetch\_store.c**

# EOF in more detail

- **What happens at end of store?**
  - Server receives EOF, renames file, responds OK
  - Client reads OK, *after* sending EOF—so didn't close fd!
- `int shutdown (int fd, int how);`
  - Shuts down a socket w/o closing file descriptor
  - how: 0 = reading, 1 = writing, 2 = both
  - Note: Applies to *socket*, not descriptor—so copies of descriptor (through `dup` or `fork` affected)
  - Note 2: With TCP, can't detect if other side shuts for reading
- **Many network applications detect & use EOF**
  - Common error: “leaking” file descriptor via `fork`, so not closed (and no EOF) when you exit

# Using UDP

- **Call socket with SOCK\_DGRAM, bind as before**
- **New system calls for sending individual packets**
  - `int sendto(int s, const void *msg, int len, int flags, const struct sockaddr *to, socklen_t tolen);`
  - `int recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, socklen_t *fromlen);`
  - Must send/get peer address with each packet
- **Example: udpecho.c**
- **Can use UDP in connected mode (Why?)**
  - connect assigns remote address
  - send/recv syscalls, like sendto/recvfrom w/o last 2 args

# Uses of connected UDP sockets

- **Kernel demultiplexes packets based on port**
  - So can have different processes getting UDP packets from different peers
  - For “security”, ports  $< 1024$  usually can't be bound
  - But can safely inherit UDP port below that connected to one particular peer
- **Feedback based on ICMP messages (future lecture)**
  - Suppose no process has bound the UDP port you sent a packet to...
  - With `sendto`, you might think network dropping packets
  - Server sends port unreachable message, but only detect it when using connected sockets

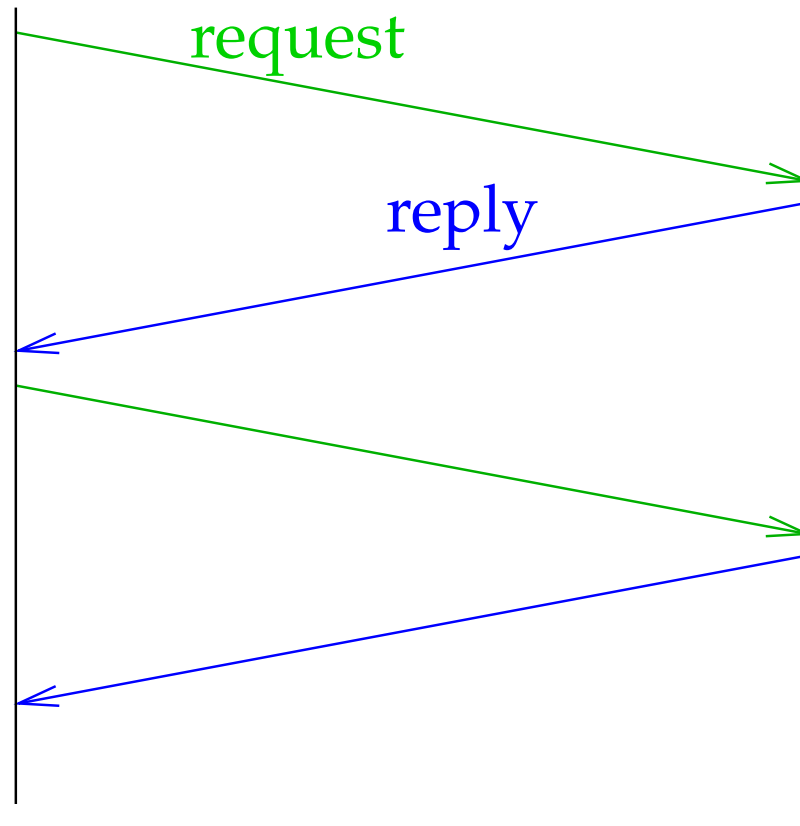
# Performance definitions

- **Bandwidth** – Number of bits/time you can transmit
  - Improves with technology
- **Latency** – How long for message to cross network
  - Propagation + Transmit + Queue
  - We are stuck with speed of light...  
16ms to cross country (3000mi)
- **Throughput** –  $\text{TransferSize} / \text{Latency}$
- **Jitter** – Variation in latency
- What matters most for your application?

# Small request/reply protocol

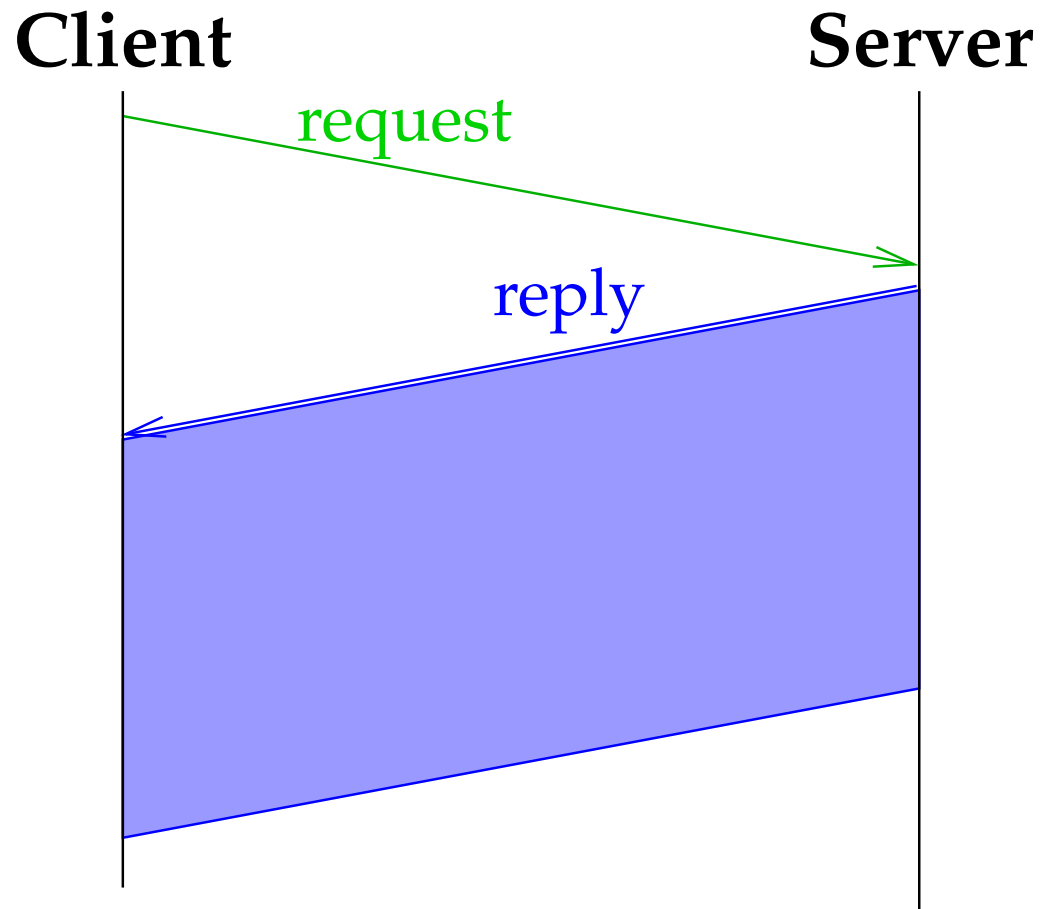
Client

Server



- Small message protocols typically dominated by latency

# Large reply protocol



- For bulk transfer, throughput is most important

# Bandwidth-delay



- **Can view network as a pipe**
  - For full utilization want bytes in flight  $\geq$  bandwidth  $\times$  delay
  - But don't want to **overload** the network (future lectures)
- **What if protocol doesn't involve bulk transfer?**
  - Get throughput through concurrency—service multiple clients simultaneously

# Traditional fork-based servers

- **When is a server not transmitting data**
  - Read or write of a socket connected to slow client can block
  - Server may be busy with CPU (*e.g.*, computing response)
  - Server might be blocked waiting for disk I/O
- **Can gain concurrency through multiple processes**
  - Accept, fork, close in parent; child services request
- **Advantages of one process per client**
  - Don't block on slow clients
  - May scale to multiprocessors if CPU intensive
  - For disk-heavy servers, keeps disk queues full  
(similarly get better scheduling & utilization of disk)

# Threads

- **One process per client has disadvantages:**
  - High overhead – fork+exit  $\sim 100 \mu\text{sec}$
  - Hard to share state across clients
  - Maximum number of processes limited
- **Can use threads for concurrency**
  - Data races and deadlock make programming tricky
  - Must allocate one stack per request
  - Many thread implementations block on some I/O or have heavy thread-switch overhead

Rough equivalents to `fork()`, `waitpid()`, `exit()`, `kill()`, plus locking primitives. Attend tonight's tutorial to learn more.

# Non-blocking I/O

- **fcntl sets O\_NONBLOCK flag on descriptor**

```
int n;  
if ((n = fcntl (s, F_GETFL)) >= 0)  
    fcntl (s, F_SETFL, n | O_NONBLOCK);
```

- **Non-blocking semantics of system calls:**

- read immediately returns -1 with errno EAGAIN if no data
- write may not write all data, or may return EAGAIN
- connect may “fail” with EINPROGRESS (or may succeed, or may fail with real error like ECONNREFUSED)
- accept may fail with EAGAIN if no pending connections

# How do you know when to read/write?

```
struct timeval {
    long    tv_sec;          /* seconds */
    long    tv_usec;        /* and microseconds */
};

int select (int nfd, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);
```

Entire program runs in an *event loop*.

# Event driven servers

Quite different from processes/threads.

- **Race conditions, deadlock are rare.**
- **Often more efficient.**

but...

- **Unusual programming model.**
- **Sometimes difficult to avoid blocking.**
- **Scaling to multiple CPUs is more complex.**

## Coming up

- **Tonight, 5pm: Snowcast help session [368]**
- **Tonight, 7pm: cs167's Pthreads intro 1 [368]**
- **Thu 13: Link layer**
- **Thu 13, 7pm: cs167's Pthreads intro 2 [368]**