University of Houston

Introduction to Computer Networks

COSC 6377

Midterm Review

Author
K.M. Hourani

Based on Notes By
Dr. Omprakash Gnawali

March 3, 2021

Contents

1 Intro **5**
- 1.1 The Internet . . . 5
- 1.2 Packet vs. Circuit Switching . . . 5
  - 1.2.1 Circuit Switching . . . 5
  - 1.2.2 Some Circuit Switching Techniques . . . 6
  - 1.2.3 Packet Switching . . . 6
  - 1.2.4 Summary . . . 6
- 1.3 Describing a Network . . . 6
  - 1.3.1 Throughput . . . 7
  - 1.3.2 Latency . . . 7
    - 1.3.2.1 Relation between Latency and Throughput . . . 7
  - 1.3.3 Reliability . . . 7
- 1.4 Protocols . . . 7
- 1.5 Network Protocols . . . 8
  - 1.5.1 Protocols and Standards . . . 8
    - 1.5.1.1 Protocol Layers . . . 8
- 1.6 Encapsulation . . . 8

2 Network Applications and Socket Programming **11**
- 2.1 Network Applications . . . 11
  - 2.1.1 Inter-Application Communication . . . 11
  - 2.1.2 Application Protocols . . . 11
  - 2.1.3 Network Time Service . . . 12
    - 2.1.3.1 Protocol Timing Diagram . . . 12
    - 2.1.3.2 Cloud-based File Backup Application . . . 12
- 2.2 Socket Programming . . . 13
  - 2.2.1 Using TCP/IP . . . 13
  - 2.2.2 System Calls . . . 13
  - 2.2.3 File Descriptors . . . 13
  - 2.2.4 Error Returns . . . 13
  - 2.2.5 Some operations on File Descriptors . . . 14
  - 2.2.6 Sockets: Communincation Between Machines . . . 14
  - 2.2.7 System calls for using TCP . . . 14
  - 2.2.8 Socket Naming . . . 14
  - 2.2.9 Socket Address Structures . . . 15
  - 2.2.10 Dealing with Address Types . . . 15
  - 2.2.11 Client Skeleton (IPv4) . . . 15
  - 2.2.12 Server Skeleton (IPv4) . . . 15
  - 2.2.13 Looking up socket address with `getaddrinfo` . . . 16
  - 2.2.14 `getaddrinfo()`[RFC3493] . . . 16
  - 2.2.15 EOF in more detail . . . 16
  - 2.2.16 Using UDP . . . 17
  - 2.2.17 Serving Multiple Clients . . . 17

2.2.18 Threads 17
2.2.19 Non-blocking I/O 17
2.2.20 How do you know when to read/write? 18
2.2.21 Event-driven servers 18

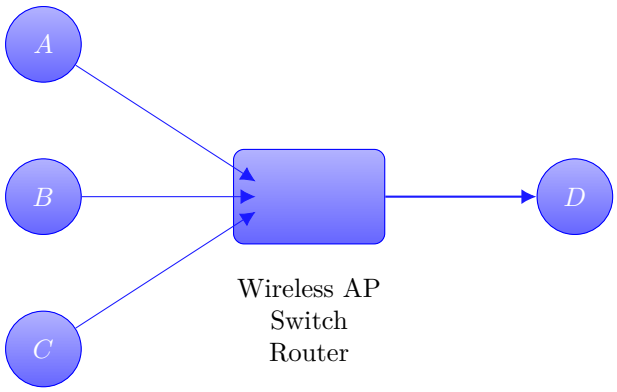
Chapter 1

Intro

1.1 The Internet

- Collection of nodes, wired and wireless technology connecting these nodes, applications and services
- Types of nodes
  - Desktops and Laptop
  - Servers
  - TV/Refrigerator
  - Cellphones
- Goal: Connect all the nodes to each other
- Solutions
  - $\binom{n}{2} = \mathcal{O}(n^2)$ cables
  - Sharing the links
    - Circuit Switching
    - Packet Switching
- Packet
  - Collection of bits to transfer across a network
  - Think: envelope and its contents
- Circuit
  - Pre-allocated path/resource

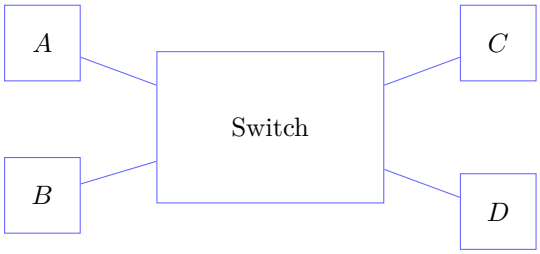
1.2 Packet vs. Circuit Switching



1.2.1 Circuit Switching

- Setup the connection or resource

  - Schedule (e.g., TDMA)
  - State in the network



Time	Circuit
T, $3T$, $5T$, ...	$A-D$
$2T$, $4T$, $6T$, ...	$B-C$

- Natural for predictable data races
- Can guarantee certain level of services
- Can be inefficient for many applications

1.2.2 Some Circuit Switching Techniques

- Time
  - Reserve to use the link at a given schedule
  - Read: https://en.wikipedia.org/wiki/Time-division_multiplexing
- Frequency
  - Reserve to use certain frequencies (channel)
  - Read: https://en.wikipedia.org/wiki/Frequency-division_multiplexing

1.2.3 Packet Switching

- Wire is selected for each packet
- No network **state**
- Supports unpredictable/bursty traffic pattern
- Higher link utilization
- No guarantees but good enough for most applications

https://en.wikipedia.org/wiki/Packet_switching

1.2.4 Summary

- Packet Switching
  - Plus: more sharing (more efficient)
  - Minus: no service guarantee
- Circuit Switching
  - Plus: service guarantee
  - Minus: less sharing (less efficient)
- Every day examples
  - Road network

1.3 Describing a Network

- How to describe how well a network is working?
  - Metrics
- Performance metrics
  - Throughput
  - Latency
  - Reliability

1.3.1 Throughput

- How many bytes can we send through in a given time?
  - Bytes per second
  - How many bits/s in kbps?
  - Read: https://en.wikipedia.org/wiki/Data-rate_units
- Useful bytes transferred vs. overhead
  - Goodput
  - Everyday example: car vs. passenger

https://en.wikipedia.org/wiki/Throughput

1.3.2 Latency

- How long does it take for one bit to travel from one end to the other end?
  - ms, s, minutes, etc.
- Typical latencies
  - Speed of light
  - Why is web browsing latency in seconds?

1.3.2.1 Relation between Latency and Throughput

- Characterize the latency and throughput of
  - Oil Tanker –
  - Aircraft –
  - Car –
  - Tractor Trailer –
- Which metrics matter most for these applications?
  - Netflix
  - Skype
  - Amazon
  - Facebook

1.3.3 Reliability

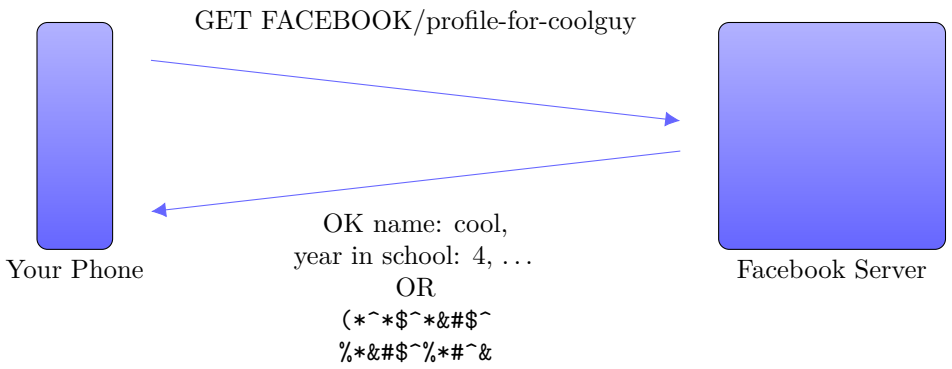
- How often does a network fail?
- How often do packets drop?
  - Damage (corruption)
  - Drops in the queues
- How persistent are failures?
- Typical metrics
  - uptime percentage
  - packet or bit loss rate

1.4 Protocols

- Agreed-upon rules, format, and meaning for message exchange
- Let's examine this sequence:
  - Hellow
  - How are you?
  - Fine.

https://en.wikipedia.org/wiki/Communication_protocol

1.5 Network Protocols



What are the rules, format, and meaning in this message exchange?

1.5.1 Protocols and Standards

- How can your phone (HTC running Android) access Facebook (runs on UNIX-like OS on big servers)?
- Using standard protocol enables interoperation
- Who standardizes the protocols?

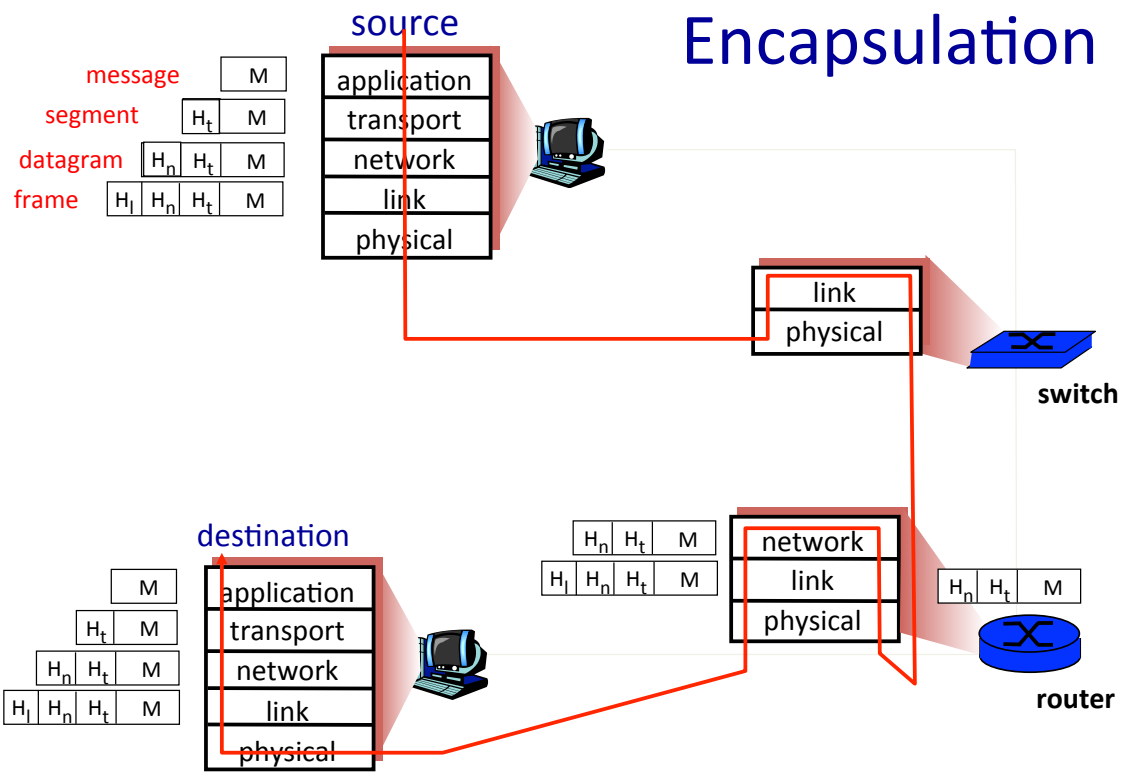
1.5.1.1 Protocol Layers

- Lower level to higher level message exchange
  - Organize the functionalities
  - Abstractions in services used and provided
- 5-7 layers depending on who you talk to
  - Physical, Link, Network, Transport, Application
- Should a smartphone app developer worry about
  - Voltages being applied on the wire
  - If the underlying media uses packet or circuit switching

https://en.wikipedia.org/wiki/Protocol_stack

1.6 Encapsulation

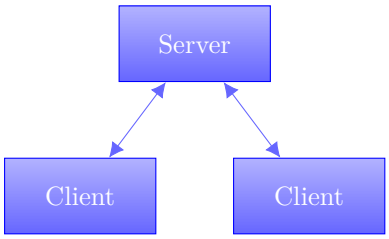
- Think of how paperwork is processed in a university
  - Each person processes and adds some information to it and passes it along
- On the transmitter, the lower layers include the message from upper layers, add their own information, and send it along
- On the receiver: reverse


Encapsulation
source
message
M
segment
H_t M
datagram
H_n H_t M
frame
H_l H_n H_t M
application
transport
network
link
physical
link
physical
switch
destination
M
H_t M
H_n H_t M
H_l H_n H_t M
application
transport
network
link
physical
H_n H_t M
H_l H_n H_t M
network
link
physical
H_n H_t M
router

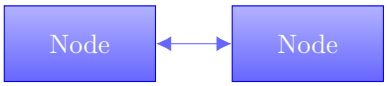
Chapter 2

Network Applications and Socket Programming

2.1 Network Applications



Client-Server



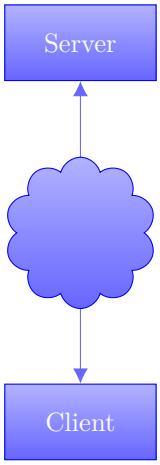
Peer-to-Peer

Hybrid

https://en.wikipedia.org/wiki/Peer-to-peer

2.1.1 Inter-Application Communication

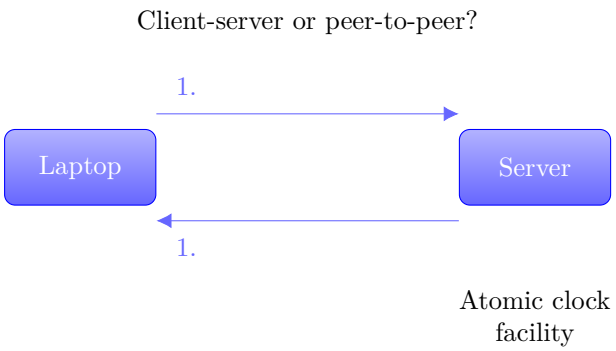
- Need a way to send and receive messages
- Inter-process communication
- Need naming, routing, transport
- Transport using TCP and UDP
  - On top of IP



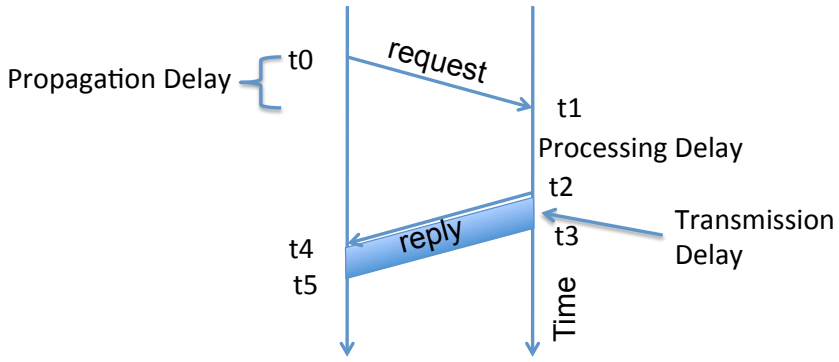
2.1.2 Application Protocols

- Messages between processes, typically encapsulated within TCP or UDP
- Need agreement between
  - Sending process
  - Receiving process

2.1.3 Network Time Service

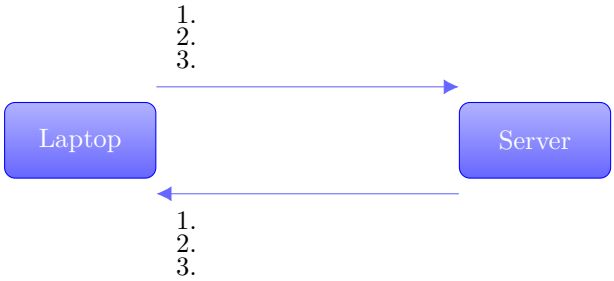


2.1.3.1 Protocol Timing Diagram



2.1.3.2 Cloud-based File Backup Application

- Client-server or peer-to-peer?
- Where do the applications run?
- Who/how to run these applications?
- What messages are exchanged?

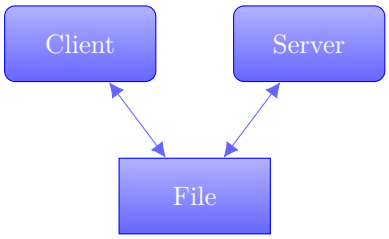


2.2 Socket Programming

2.2.1 Using TCP/IP

- How can applications use the network?
- *Sockets* API
  - Originall from BS, widely implemented (*BSD, Linux, Mac OS X, Windows, . . .)
  - Higher-level APIs build on them
- After basic setup, much like files

One could test network protocols with read/write on a file



2.2.2 System Calls

- Problem: how to access resources other than the CPU
  - Disk, netowrk, terminal, other processes
  - CPU prohibits instructions that would access devices
  - Only privileged OS kernel can access devices
- Kernel supplies well-defined system call interface
  - Applications request I/O oeprations through syscalls
  - Set up syscall arguments and trap to kernel
  - Kernel performs operation and returns result
- Higher-level functions built on syscall interface
  - `printf`, `scanf`, `gets`, all user-level code

2.2.3 File Descriptors

- Most I/O in Unix done through *file descriptors*
  - Integer *handles* to per-process table in kernel
- `int open(char *path, int flags, ...);`
- Returns file descriptor, used for all I/O to file

https://en.wikipedia.org/wiki/File_descriptor

2.2.4 Error Returns

- What if `open` fails? Return `-1` (invalid file descriptor)
- Most system calls return `-1` on failure
  - Specific type of error in gobal `int errno`

- `#include <sys/errno.h>` for possible values
  - `2` = `ENOENT` "no such file or directory"
  - `13` = `EACCES` "permission denied"

2.2.5 Some operations on File Descriptors

- `ssize_t read(int fd, void* buf, int nbytes);`
  - Returns number of bytes read
  - Returns `0` bytes at end of file, or `-1` on error
- `ssize_t write(int fd, void* buf, int nbytes);`
  - Returns number of bytes written, `-1` on error
- `off_t lseek(int fd, off_t offset, int whences);`
  - `whence`: `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
  - returns new offset, or `-1` on error
- `int close(int fd);`

2.2.6 Sockets: Communincation Between Machines

- Network sockets are file descriptors too
- 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 another
  - Reads may not return full amount requested, must reread

2.2.7 System calls for using TCP

	Client	Server
1.		`socket` – make socket
2.		`bind` – assign address, port
3.		`listen` – listen for clients
4.	`socket` – make socket	
5.	`bind` – assign address[1]	
6.	`connect` – connect to listening socket	
7.		`accept` – accept connection

2.2.8 Socket Naming

- Naming of TCP and UDP communication endpoints
  - IP address specifies host (129.7.240.18)
  - 16-bit port number demultiplexes within host
  - Well-known services listen on standard ports (e.g. ssh – 22, http – 8, see `/etc/services` for list)
  - Clients connect from arbitrary ports to well-known ports
- A connection is named by 5 components
  - Protocol, local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP

[1] This call to bind is optional, connect can choose address and port

2.2.9 Socket Address Structures

- Socket interface supports multiple network types
- Most calls take a generic `sockaddr`:

```
struct sockaddr {
    uint16_t sa_family;   /* address family */
    char     sa_data[14]; /* protocol-specific addr */
};
```

- e.g. `int connect(int s, struct sockaddr* srv, socklen_t addrlen);`
- Cast `sockaddr*` from protocol-specific struct, e.g.

```
struct addr_in {
    short   sin_family;       /* = AF_INET */
    u_short sin_port;         /* = htons (PORT) */
    struct  in_addr sin_addr; /*32-bit IPV4 addr */
    char    in_zero[8];
};
```

2.2.10 Dealing with Address Types

- All values in network byte order (Big Endian)
  - `htonl()`, `htons()`: host to network, 32 and 16 bits
  - `ntohl()`, `ntohs()`: network to host, 32 and 16 bits
  - Remember to always convert!
- All address types begin with family
  - `sa_family` in `sockaddr` tells you the actual type
- Not all addresses are the same size
  - e.g. `struct sockaddr_in6` is typically 28 bytes, yet generic `struct sockaddr` is only 16 bytes
  - so most calls require passing around socket length
  - new `sockaddr_storage` is big enough

2.2.11 Client Skeleton (IPv4)

```
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);
memset(&sin, sizeof(sin), 0);
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));
while ((n = read(s, buf, sizeof(buf))) > 0) {
    write(1, buf, n);
}
```

2.2.12 Server Skeleton (IPv4)

```
int s = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin;
```

```
    memset(&sin, sizeof(sin), 0);
    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);
    while (true) {
        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);
    }
```

2.2.13 Looking up socket address with `getaddrinfo`

```
struct addrinfo hints, *ai;
int err;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;     /* or AF_INET or AF_INET6 */
hints.ai_socktype = SOCK_STREAM; /* or SOCK_DGRAM for UDP */

err = getaddrinfo("www.brown.edu", "http", &hints, &ai);
if (err) {
    fprintf (stderr, "%s\n", gia_strerror (err));
} else {
    /* ai->ai_family = address type (AF_INET or AF_INET6) */
    /* ai->ai_addr = actual address cast to (sockaddr *) */
    /* ai->ai_addrlen = length of actual address */
    freeaddrinfo (ai); /* must free when done! */
}
```

2.2.14 `getaddrinfo()`[RFC3493]

- Protocol-independent node name to address translation
  - Can specify port as a service name or number
  - May return multiple addresses
  - You must free the structure with `freeaddrinfo`
- Other useful functions to know about
  - `getnameinfo` – lookup hostname based on address
  - `inet_ntop` – convert IPv4 or 6 address to printable
  - `inet_prton` – convert string to IPv4 or 6 address

2.2.15 EOF in more detail

- What happens at the end of store?
  - Server receives EOF, renames file, responds OK
  - Client reads OK, *after* sending EOF: didn't close `fd`
- `int shutdown(int fd, int how);`
  - Shuts down a socket without closing the file descriptor
  - how: `0` = read, `1` = write, `2` = both
  - Note 1: applies to *socket*, not descriptor, so copies of descriptor (through fork or dup) affected
  - Note 2: with TCP, can't detect if other side shuts down for reading

2.2.16 Using UDP

- Call socket with `SOCK_DGRAM`, bind as before
- New calls for sending/receiving individual packets
  - `sendto(int s, const void* msg, int len, int flags, const struct sockaddr* to, socklen_t tolen);`
  - `recvfrom(int s, void* buf, int len, int flags, struct sockaddr *from, socklen_t* fromlen);`
  - Must send/get peer address with each packet
- Can use UDP in connected mode (why?)
  - connect assigns remote address
  - `send`/`recv` syscalls, like `sendto`/`recvfrom`, without last two arguments

2.2.17 Serving Multiple Clients

- A server may block when talking to a client
  - Read or write of a socket connected to a slow client can block
  - Server may be busy with CPU
  - Server might be blocked waiting for disk I/O
- Concurrency through multiple processes
  - Accept, fork, close in parent; child services request
- Advantages of one process per client
  - Doesn't block on slow clients
  - May use multiple cores
  - Can keep disk queues full for disk-heavy workloads

2.2.18 Threads

- One process per client has disadvantages:
  - High overhead – fork + exit $\approx 100\mu$sec
  - Hard to share state across clients
  - Maximum number of processes limited
- Can use threads for concurrency
  - Data races and deadlocks 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.

2.2.19 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 a real error like `ECONNREFUSED`)
  - accept may fail with `EAGAIN` or `EWOULDBLOCK` if no connections present to be accepted

2.2.20 How do you know when to read/write?

```
struct timeval {
    long tv_sec;  /* seconds */
    long tv_usec; /* and microseconds */
};
int select(int nfds, 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*

2.2.21 Event-driven servers

- Quite different from processes/threads
  - Race conditions, deadlocks rare
  - Often more efficient
- But...
  - Unusual programming model
  - Sometimes difficult to avoid blocking
  - Scaling to more CPUs is more complex