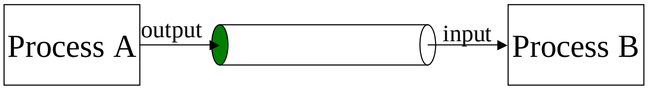
Inter-Process Communication

CS 217

Pipes

- Provides an interprocess communication channel



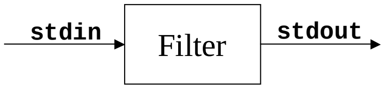
Process A

output

input

Process B

- A filter is a process that reads from **`stdin`** and writes to **`stdout`**



`stdin`

Filter

`stdout`

Pipes (cont)

- Many Unix tools are written as filters
  - `grep, sort, sed, cat, wc, awk ...`
- Shells support pipes

```
ls -l | more
who | grep mary | wc
ls *.[ch] | sort
cat < foo | grep bar | sort > save
```

Creating a Pipe

- System call

```
int pipe( int fd[2] );
return 0 upon success and -1 upon failure
fd[0] is open for reading
fd[1] is open for writing
```

- Two coordinated processes created by **`fork`** can pass data to each other using a pipe.

Pipe Example

```
int pid, p[2];
...
pipe(p);
pid = fork();
if (pid == 0) {
   close(p[1]);
   ... read using p[0] as fd until EOF ...
}
else {
    close(p[0]);
    ... write using p[1] as fd ...
    close(p[1]); /* sends EOF to reader */
    wait(&status);
}
```

Dup

- Duplicate a file descriptor (system call)

  ```
  int dup( int fd );
  ```

  duplicates **`fd`** as the lowest unallocated descriptor

- Commonly used to redirect stdin/stdout

  ```
  int fd;
  fd = open(“foo”, O_RDONLY, 0);
  close(0);
  dup(fd);
  close(fd);
  ```

Dup (cont)

- For convenience…

```
  dup2( int fd1, int fd2 );
```

use **fd2** to duplicate **fd1**
closes **fd2** if it was in use

```
fd = open(“foo”, O_RDONLY, 0);
dup2(fd,0);
close(fd);
```

Pipes and Standard I/O

```
int pid, p[2];
pipe(p);
pid = fork();
if (pid == 0) {
   close(p[1]);
   dup2(p[0],0);
   close(p[0]);
   ... read from stdin ...
}
else {
   close(p[0]);
   dup2(p[1],1);
   close(p[1]);
   ... write to stdout ...
   wait(&status);
}
```

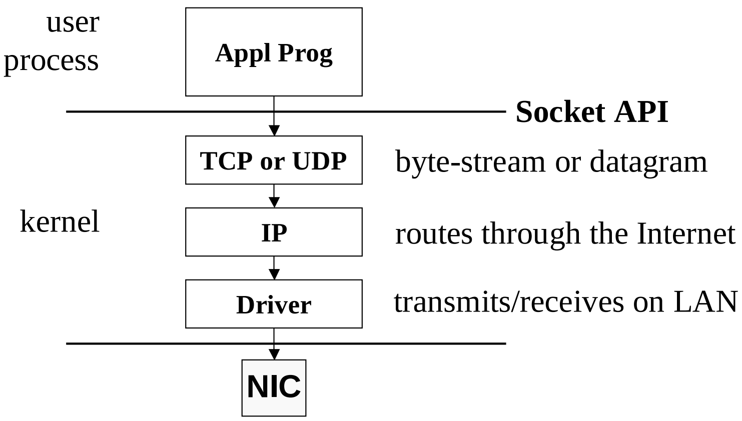
Inter-Process Communication

- Pipes
  - Processes must be on same machine
  - One process spawns the other
  - Used mostly for filters
- Messages
  - Processes can be on any machine
  - Processes can be created independently
  - Used for clients/servers, distributed systems, etc.

Messaging Example: Client/Server

- Server: process that provides a service
  - e.g., file server, web server, mail server
  - called a passive participant: waits to be contacted
- Client: process that requests a service
  - e.g., desktop machine, web browser, mail reader
  - called an active participant: initiates communication

Network Subsystem



user process

Appl Prog

Socket API

TCP or UDP

byte-stream or datagram

kernel

IP

routes through the Internet

Driver

transmits/receives on LAN

NIC

Communication Semantics

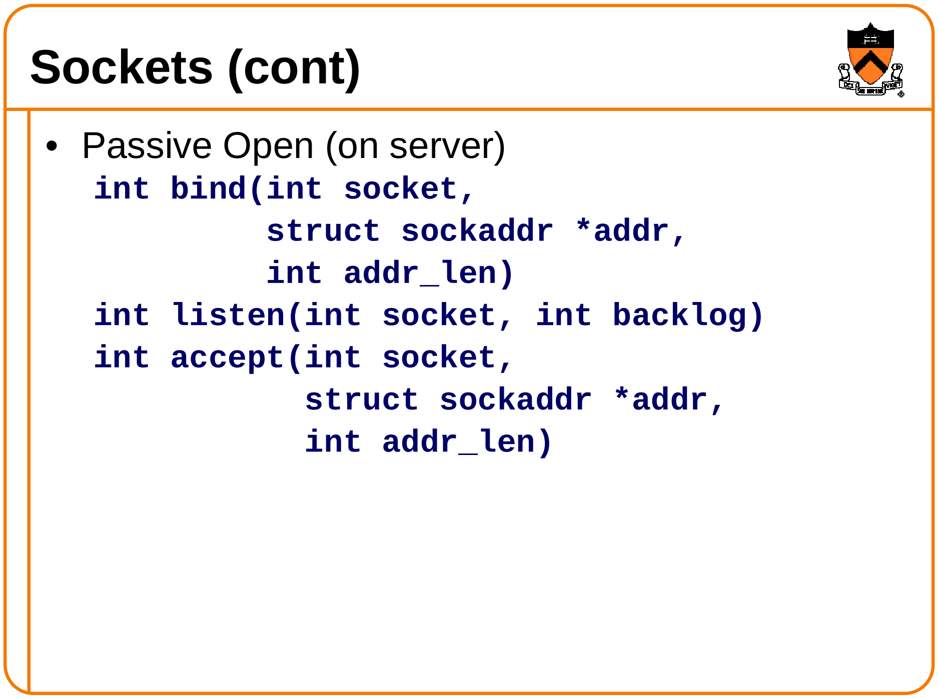
- Reliable Byte-Stream (like a pipe):
  - TCP
- Unreliable Datagram:
  - UDP

Names and Addresses

- Host name
  - like a post office name; e.g., www.cs.princeton.edu
- Host address
  - like a zip code; e.g., 128.112.92.191
- Port number
  - like a mailbox; e.g., 0-64k

Socket API

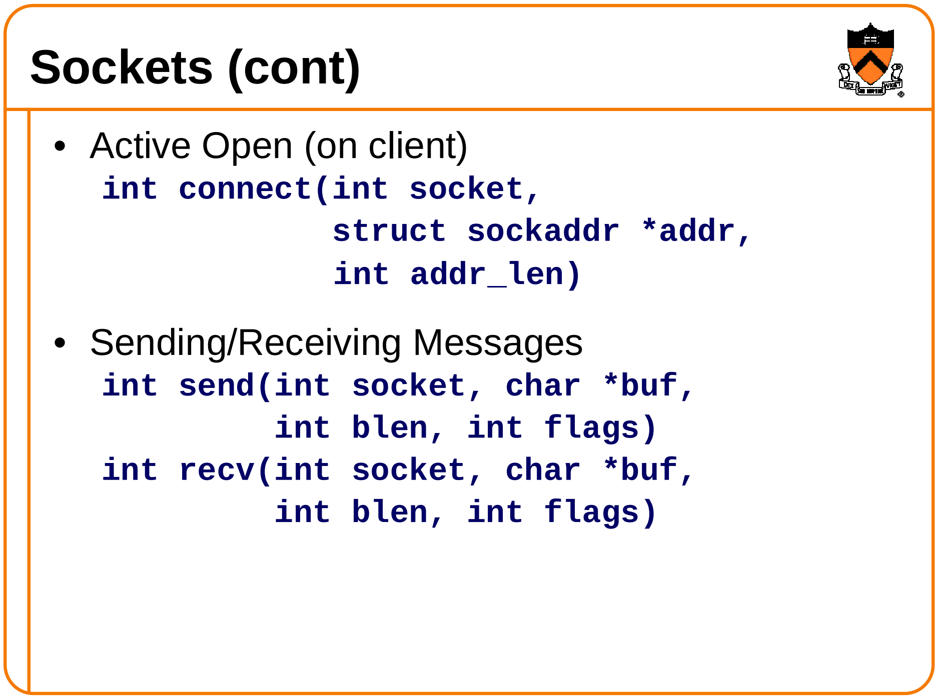
- Socket Abstraction
  - end-point of a network connection
  - treated like a file descriptor
- Creating a socket
  - `int socket(int domain, int type, int protocol)`
  - domain = PF_INET, PF_UNIX
  - type = SOCK_STREAM, SOCK_DGRAM, SOCK_RAW



Sockets (cont)

- Passive Open (on server)

```
int bind(int socket,
         struct sockaddr *addr,
         int addr_len)
int listen(int socket, int backlog)
int accept(int socket,
           struct sockaddr *addr,
           int addr_len)
```



Sockets (cont)

- Active Open (on client)

```
int connect(int socket,
            struct sockaddr *addr,
            int addr_len)
```

- Sending/Receiving Messages

```
int send(int socket, char *buf,
         int blen, int flags)
int recv(int socket, char *buf,
         int blen, int flags)
```

Trivia Question

- How many messages traverse the Internet when you click on a link?