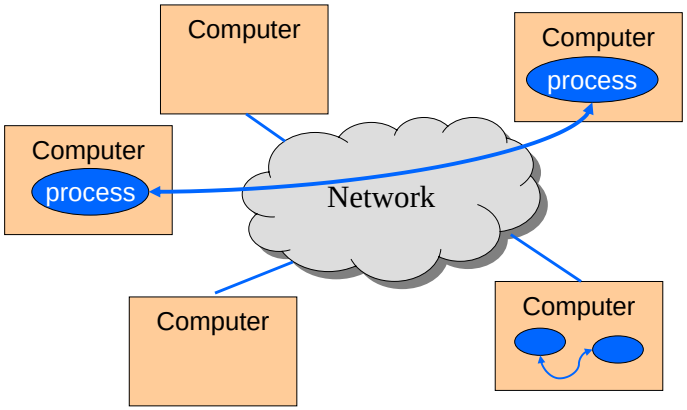
Inter-process Communication

CS 217

Networks

- Mechanism by which two processes exchange information and coordinate activities

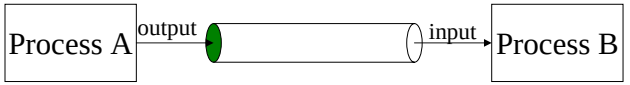


Inter-process Communication

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

Pipes

- Provides an interprocess communication channel



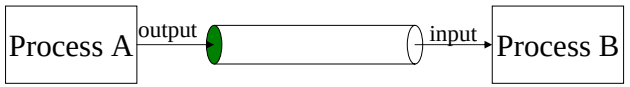
- A filter is a process that reads from **`stdin`** and writes to **`stdout`**

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



- Pipe is a communication channel abstraction
  - Process A can write to one end using “write” system call
  - Process B can read from the other end using “read” system call
- System call

```
int pipe( int fd[2] );
return 0 upon success –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];
...
if (pipe(p) == -1)
   exit(1);
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
- Example: redirect stdin to “foo”

```
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(new)`** to duplicate **`fd1 (old)`**

  closes **`fd2`** if it was in use

- Example: redirect stdin to “foo”

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


Pipes and Standard I/O

```
int pid, p[2];
if (pipe(p) == -1)
   exit(1);
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);
}
```


Pipes and Exec()

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


K&P’s Example

```
#include <signal.h>
#include <stdio.h>

system( char *s) {
   int status, pid, w, tty;

   fflush(stdout);
   tty = open(“/dev/tty”,
         O_RDWR);
   if (tty == -1) {
      fprintf(stderr, “...” );
      return -1;
   }
   if ((pid = fork()) == 0 ) {
      close(0); dup(tty);
      close(1); dup(tty);
      close(2); dup(tty);
      execlp(“sh”, “sh”, “-c”, s,
  NULL);
      exit(127);
   }
   close(tty);
   istat =
      signal(SIGINT, SIG_IGN);
   qstat =
      signal(SIGQUIT, SIG_IGN);
   while (
      (w = wait(&status)) != pid
      && (w != -1)
      ;
   if (w == -1) status = -1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   return status;
}
```

Unix shell (sh, csh, bash, ...)

- Loop
  - Read command line from stdin
  - Expand wildcards
  - Interpret redirections < > |
  - pipe (as necessary), fork, dup, exec, wait
- Start from code on previous slides, edit it until it's a Unix shell!

Client-Server Model

Network

Server
(file server,
mail server,
web server)

Client

Web browser
Emailer
Applications

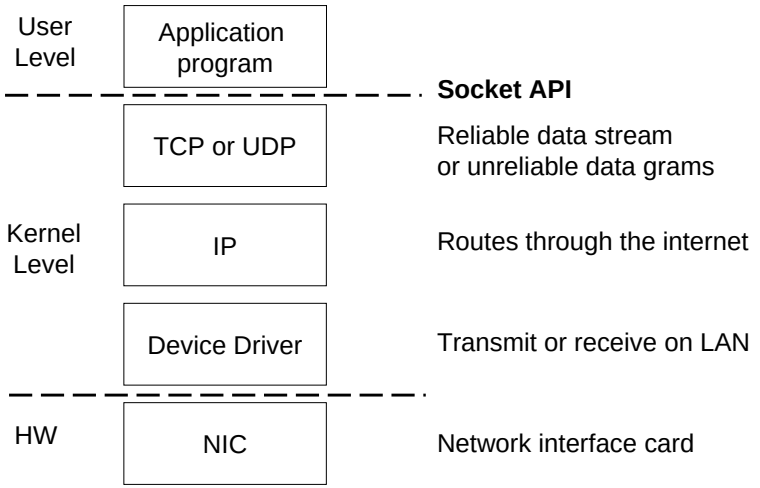
Active participant
Initiate contacts

Passive participant
Waiting to be contacted

Message Passing

- Mechanism to pass data between two processes
  - Sender sends a message from its memory
  - Receiver receives the message and places it into its memory
- Message passing is like using a telephone
  - Caller
  - Receiver

Network Subsystem

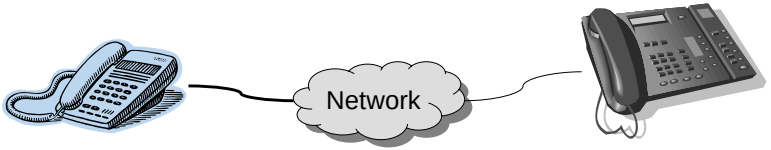


Names and Addresses

- 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

- Socket abstraction
  - An end-point of network connection
  - Treat like a file descriptor
- Conceptually like a telephone
  - Connect to the end of a phone plug
  - You can speak to it and listen to it



Steps for Client and Server

Client

- Create a socket with the socket() system call
- Connect the socket to the address of the server using the connect() system call
- Send and receive data, using write() and read() system calls or send() and recv() system calls

Server

- Create a socket with the socket() system call
- Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the listen() system call
- Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
- Send and receive data

Creating A Socket (Install A Phone)

- Creating a socket

```
#include <sys/types.h>
#include <sys/socket.h>

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

  - Domain: PF_INET (Internet), PF_UNIX (local)
  - Type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
  - Protocol: 0 usually for IP (see /etc/protocols for details)
- Like installing a phone
  - Need to what services you want
    - Local or long distance
    - Voice or data
    - Which company do you want to use

Connecting To A Socket

- Active open a socket (like dialing a phone number)

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


Binding A Socket

- Need to give the created socket an address to listen to (like getting a phone number)

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

  - Passive open on a server

Specifying Queued Connections

- Queue connection requests (like “call waiting”)

```
int listen(int socket, int backlog)
```

  - Set up the maximum number of requests that will be queued before being denied (usually the max is 5)

Accepting A Socket

- Wait for a call to a socket (picking up a phone when it rings)

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

  - Return a socket which is connected to the caller
  - Typically blocks until the client connects to the socket

Sending and Receiving Data

- Sending a message

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

- Receiving a message

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


Close A Socket

- Done with a socket (like hanging up the phone)

```
close(int socket)
```

- Treat it just like a file descriptor

Summary

- Pipes
  - Process communication on the same machine
  - Connecting processes with stdin and stdout
- Messages
  - Process communication across machines
  - Socket is a common communication channels
  - They are built on top of basic communication mechanisms