Unix Interprocess Communication

Named Pipes (fifos)

TCP Sockets

UDP Sockets

FIFO

- Like a pipe:
  - whatever is written is stored in a kernel buffer until someone reads from the fifo
  - bytes come out (are read) in the same order they were written (FIFO means "*first-in-first-out"*).
- Unlike a pipe:
  - a fifo has a *name*, which is a pathname (a fifo has a location in the file system.
  - Any process with the appropriate file permissions can access the fifo.
    - Two processes don't have to have a common ancestor.

Creating a fifo

- From the unix command line:

```
> mkfifo blah
> ls -al blah
prw-r--r--  1 hollingd 0 2005-04-06 17:36 blah|
```

- From within a C program:

```
int mkfifo(const char *pathname,
           mode_t mode);
```

mode specifies the fifo (file) permissions

Opening a fifo

- Use open()
  - just like opening a regular file.
  - to open for reading: `open(fifoname,O_RDONLY)`
  - to open for writing: `open(fifoname,O_WRONLY)`
    - fifo must already exist.
- Issues:
  - when opening for reading, the call to open will block until some process has the fifo open for writing.

reading and writing a fifo

- read() and write() work just like it was a file or pipe.
  - read will block until there is at least one byte available.
  - read can return as soon as there is some data, not necessarily the amount you requested.
    - read returns the number of bytes read.
    - write may block if the kernel buffer for the fifo is full.

Some code (reading from a fifo)

```
if (mkfifo(name, FIFO_PERMS)<0) { /*error */ }

if ((fd = open(fifo,O_RDONLY))<0) { /* error */ }

while ((cnt = read(fd,buff,100))>0) {
     write(STDOUT_FILENO,buff,cnt);
}
```

Some code (writing to a fifo)

```
if ((fd = open(fifo,O_WRONLY))<0) { /* error */ }

write(fd,"Hello There\n",12);

write(fd,"Goodbye\n",8);

close(fd);
```

Unix I/O Generality

- Once you open fifo:
  - syntax for reading and writing is the same as for any other Unix I/O (for example – file I/O).
  - semantics of calls to read and write are a little different than when reading/writing files.
    - read might return less than you asked for.
    - if read returns 0, the other end of the fifo has closed it's end (just like a pipe).
    - write can block (just like a pipe).

Quiz – what does this do?

```
void foo(int fd)  {
    int cnt;
    char *line;
    while (line = readline("> ")) {
        write(fd,line,strlen(line));
        write(fd,"\n",1);
    }
}
```

Answer: sends user input somewhere - we don't know where. You can't tell whether `fd` refers to a file, pipe, fifo or even a network connection.

Sample FIFO Client/Server

- Program looks at the command line to decide whether to be a client or server.
- Server creates fifo opens for reading (waits for client to connect for writing to fifo).
  - sends everything received from the client to STDOUT.
- Client opens fifo (pathname specified on command line).
  - readline -> send to fifo.

TCP

- TCP Network programming involves creation of socket descriptors, which can be used just like a file/pipe/fifo descriptor.
- Creating a socket descriptor is obviously more complex – we need to specify a communication endpoint that includes the IP address of another machine and a port number.
- Servers and clients create socket descriptors very differently.

Sockets API

- Sockets is a network programming API
  - not the only API available.
  - sockets is not a “TCP/IP” API, it can support many network protocols.
  - makes use of existing Unix I/O services where it makes sense to do so.
- Sockets can be supported by the kernel, or as a library that translates calls to the native network api.

Socket types

- Two basic kinds of sockets:
  - stream oriented (connection oriented)
    - a connection is formally established before any data flows.
    - server needs to create a special kind of socket (passive mode socket) used to establish connections (but never used to send/receive data).
    - When using TCP/IP – TCP socket.
  - datagram (message oriented)
    - When using TCP/IP – UDP socket.

Basic Sockets functions

- `socket()`: create a socket descriptor
  - address is not yet assigned.
  - need to specify what kind of socket.
- `bind()`: establish local address
  - typically only used by servers
  - clients usually don't care what their local address is.

TCP Server Sockets functions (really any connected mode sockets)

- `listen()`: put a socket in passive mode
  - connected mode server socket
  - tells OS to accept incoming connections and queue them up.
- `accept()`: wait for incoming connection
  - connected mode server socket
  - tell OS to give us the next incoming connection.

TCP Client Sockets functions (really any connected mode sockets)

- `connect()`: create a connection to a remote server.
  - need to already have a connected mode socket.
  - need to specify the address of the server:
    - IP address
    - port number

socket()

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

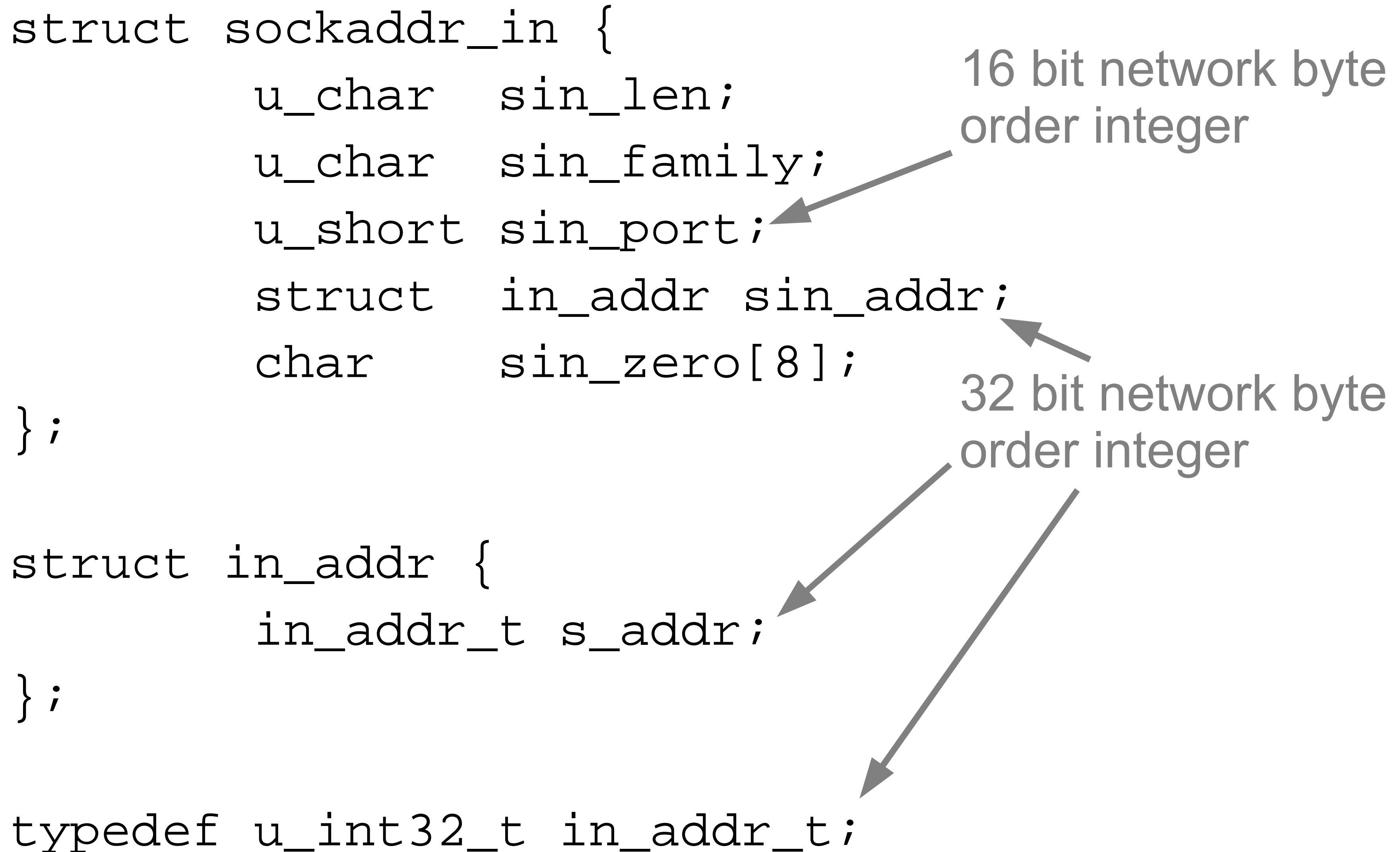
- `domain` specifies the protocol family, for TCP/IP use the constant `PF_INET`
- `type` specifies the type of socket:
  - TCP: `SOCK_STREAM`
  - UDP: `SOCK_DGRAM`
- `protocol` specifies a specific protocol
  - for TCP/IP just use `0`

Sockets data structure

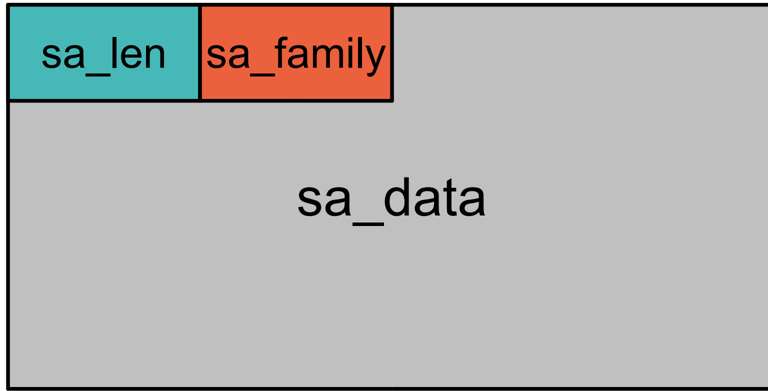
```
struct sockaddr {
  u_char sa_len;               /* total length  */
  sa_family_t  sa_family; /* address family */
  char sa_data[14];          /* address value  */
};
```

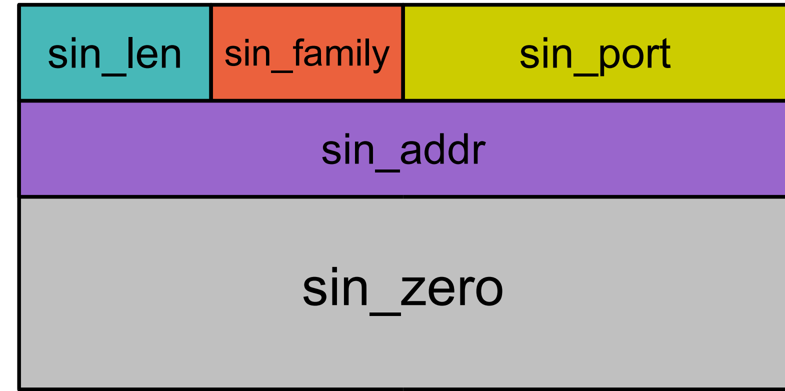
- generic *holder of a network address*
- actual address byes go in `sa_data`

TCP/IP specific sockaddr



sockaddr vs. sockaddr_in






```
struct sockaddr {
  u_char sa_len;
  sa_family_t  sa_family;
  char sa_data[14];
}
```

```
struct sockaddr_in {
    u_char   sin_len;
    u_char   sin_family;
    u_short sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

Using sockaddr_in

When using TCP/IP, we always use sockaddr_in.

- sockets library routines expect a pointer to a sockaddr
- we need to do lots of casting.
- This is expected (this was the idea when the sockets library was designed).

bind()

```
int bind(int s,
         const struct sockaddr *addr,
         socklen_t addrlen);
```

- `s` is a socket descriptor.
- addr is a pointer to a `sockaddr` (or `sockaddr_in`!)
- `addrlen` is the size of the `sockaddr`
  - to support addresses that might be too large for a `sockaddr`.

Typical `bind` usage

```
struct sockaddr_in skaddr;


skaddr.sin_family = AF_INET;
skaddr.sin_addr.s_addr = htonl(INADDR_ANY);
skaddr.sin_port = htons(port);
if (bind(sd, (struct sockaddr *) &skaddr,
         sizeof(skaddr))<0) {
       /* Error - bind failed... */
}
```

When to use `bind`

- Typically only server cares about it's address.
  - generally only cares about the port number
  - `INADDR_ANY` tells the OS “whatever IP we have is fine with me”
- Sometimes a server needs to bind to a specific IP address (if there are more than one network interface or IP address on the host).
- Clients rarely care what their local address is...
  - they don't need to call bind, the local address can be assigned other ways.

`listen()`

```
int listen(int s, int backlog);
```

- `s` is a stream socket descriptor
  - already bound to an address
- `backlog` is the number of incoming requests to queue up (a lower bound).
- Only a server calls listen!
  - `s` becomes a *passive* mode socket.

accept()

```
int accept(int s,
              struct sockaddr *addr,
              socklen_t *addrlen);
```

- `s` is a passive mode socket descriptor
- `addr` is the address of a sockaddr where accept will place the client's address.
- `addrlen` is a value/result parameter
  - initially **must** have the size of `addr`,
  - upon return will have the size of the client address

Return value from `accept()`

- `accept()` returns -1 if there is an error.
- If everything is ok, accept returns an *active mode* socket descriptor.
  - if your `write` to this descriptor, it goes to the client.
  - anything you `read` from this descriptor was sent by the client.

Typical `accept` usage

```
struct sockaddr_in from;
int len;
len=sizeof(from);
client = accept( ld, (struct sockaddr*) &from,
                 &addrlen));
if (client < 0) {
      perror("Problem with accept call");
      exit(1);
}
```

connect()

```
int connect(int s,
            const struct sockaddr *svr,
            socklen_t addrlen);
```

- `s` is a stream socket descriptor (no need to be bound to a local address).
- `svr` is the address of a sockaddr that contains the address of the server (IP and port).
- `addrlen` is the size of the `sockaddr` passed.

Return value from connect

- connect returns -1 if there is an error
- 0 means everything is OK (there is now a connection).
  - a TCP connection has been established, control information has been exchanged with the server.
  - the socket descriptor is now ready for writing and reading (to/from the server).

Reading and Writing TCP sockets

- Once the connection is established, there is no difference between the client and server (as far as the network is concerned).
  - read() will block until:
    - there is something (at least 1 byte) from the peer.
    - the peer has closed the connection – read will return 0.
  - write() will typically not block. A call to write on a TCP socket does not wait for the peer to receive the data (it queues the outgoing data, and sends it later).

I/O and TCP sockets

- If the peer has closed it's end and your OS knows this, a call to `write()` will generate a `SIGPIPE`.
  - by default this kills your process.
- Just because you tell read() to read 20 bytes, does not mean it will return 20 bytes! It can return a single byte (it depends on how the data actually arrives at your machine).
  - read knows nothing about “lines of text”, or even about “text”. Just raw bytes.

Sample TCP Client/Server

- Program looks at the command line to decide whether to be a client or server.
- Server creates passive mode TCP socket and waits for connections.
  - sends everything received from each client to STDOUT.
- Client connects to server (hostname,port specified on command line).
  - readline -> send to TCP server.

UDP Programming

- Not stream based, instead *message based.*
- `recvfrom()`: wait for incoming datagram.
  - can read datagram from *anyone*, or specify a specific peer address (IP,port).
- `sendto()`: send a datagram to a peer (must specify peer IP and port number).

Sending a datagram means "please try your best to deliver this". No guarantee, no way to know if the message was delivered.

recvfrom()

```
ssize_t recvfrom(int s,
                 void *buf, size_t len,
                 int flags,
                 struct sockaddr *from,
                 socklen_t *fromlen);
```

- `s` is UDP socket descriptor
- `buf` is the address to put the incoming message
  - maximum # bytes is `len`
- The address of the sender is placed in `from`
- `fromlen` is a value/result parameter – the size of the address `sockaddr`.

sendto()

```
ssize_t sendto(int s,
               const void *msg, size_t len,
               int flags,
               const struct sockaddr *to,
               socklen_t tolen);
```

- `s` is socket descriptor
- `msg`, `len` are the address of the payload and it's length (bytes)
- `to` holds address of peer, `tolen` is size of the `sockaddr`
- `flags` allows you to specify some options.
- return the number of bytes sent , or -1 on error.
- you can send a 0 byte message!

Sample UDP client/server

- Program looks at the command line to decide whether to be a client or server.
- Server creates UDP socket and waits for incoming messages (datagrams).
  - sends everything received from to STDOUT.
- Client calls readline, sends each line to server as a single message (datagram).