Sockets

Lecture 04.05

Inter-process communication

- ✓ Wait for exit status (report when done)
- ✓ Pipe (always open for communication)
- ✓ Signals (send when you want, handle or ignore)
- ➡ Sockets (open connection with the world)

Need for a general way of inter-process communication

- We have seen IPC through:
  - files
  - pipes
  - wait
  - signals
- These are limited:
  - Pipes require a common ancestor process to set up the pipe
  - Signals are just a "poke" rather than a full data stream

General IPC through sockets

- We want two unrelated processes to talk with each other:
    - Created by different shells
    - Created by different users
    - Running on different machines

- Sockets are communication points on the same or different computers to exchange data
- Sockets are supported by Unix, Windows, Mac, and many other operating systems
- Now they are also supported by all modern browsers

Sockets use file descriptors to talk

- Every I/O action is done by writing or reading to/from a stream using *file descriptor*: an integer associated with an open stream – this can be a network connection, a text file, a terminal, or anything else
- To a programmer, a socket looks and behaves much like a low-level file descriptor: has read(), write(), close()
- Sockets are **full-duplex (2 way)** – as if opening a stream for both reading and writing
- The only difference – how we setup the socket

If 2 processes are unrelated – we need a protocol for communication

- To make 2 machines to understand each other we need a set of rules called a *protocol*
- There are several levels of protocols:
  - TCP protocol – how to transfer and receive byte streams
  - IP protocol – how to locate and connect to a machine on the internet
- There are other protocols:
  - For example, HTTP protocol establishes rules of communication between browser and web server
  - Many application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data

Protocol types

- There are four types of socket protocols

- The first two are most common:
  - TCP (Transmission Control Protocol)
  - UDP (User Datagram Protocol)

Socket types are based on underlying protocols

- Stream sockets – TCP
- Datagram sockets - UDP

Difference between stream and datagram sockets

- Stream Sockets (TCP)
  - Message delivery is guaranteed. If delivery is impossible, the sender receives an error indicator
  - If you send three items "A, B, C", they will arrive in the same order – "A, B, C"
  - Data records do not have any boundaries

- Datagram Sockets (UDP)
  - Delivery is not guaranteed
  - Connectionless: you don't need to have an open connection as in Stream Sockets – you build a packet with the destination information and send it out

Client-server applications

- Sockets are used in a client-server framework
- A server is a process that performs some functions on request from a client

- Creating socket on your machine is like installing a customer service phone line
- If clients know the number to call, they can connect and communicate with your service

For help – call 111.222.333.444 ext 555

Unix sockets

Data communications endpoints for exchanging data between processes executing **on the same host** system

Unix domain socket server

code in *server.c*

Define a socket prototype

```
int serv_socket_fd;
```

1 *Socket type – Unix socket*

2 *Protocol type – TCP*

3 *Protocol id – can be more than 1, but not here*

```
if ((serv_socket_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror ("socket");
        exit (1);
}
```

socket() call does not specify where data will be coming from, nor where it will be going to –it just creates the interface!

Assign address to socket (like a phone number)

- You got a socket descriptor prototype from the call to *socket*(), now you want to associate it with an address
- That address is a special file on disk – register a new name with Unix file system
- Different processes can access these “files” as file system inodes, so two processes can communicate by reading/writing from/to the same file

Address for Unix domain sockets

```
struct sockaddr_un server_addr;
```

Declare variable of type *sockaddr_un*

```
memset(&server_addr, '\0', sizeof (server_addr));
```

Clear all bytes to zero

```
server_addr.sun_family = AF_UNIX;
```

Setup address family

```
strcpy(server_addr.sun_path, "/tmp/something");
```

Setup file name (creates an inode)

```
unlink(server_addr.sun_path);
```

Delete file with this name if already exists

3 steps of socket server setup: BLA

1. Bind
2. Listen
3. Accept

1. Bind

- Binding involves creating the connection resource, in this case the socket inode (a new kind of "special file")

```
struct sockaddr_un serv_addr;  //all set up
```

1 *fd returned from socket()*

2 *Address to bind to and its size*

```
if (bind(serv_fd, (struct sockaddr *)&serv_addr, sizeof (serv_addr)))
{
        perror("bind");
        return(1);
}
```

Returns zero on success

Some sort of “polymorphism”

- Because there are many types of sockets and their addresses, second argument of *bind*() is of a general type *sockaddr*, so we can put there any address type, but we need to cast it to *sockaddr*
- Third parameter tells how much space to interpret for reading an actual address from a given memory location

```
struct sockaddr_un serv_addr;  //all set up

bind(serv_fd, (struct sockaddr *)&serv_addr, sizeof (serv_addr)) ;
```

2. Listen

- Listen — wait for incoming connections
- Also specifies the length of the queue for connections which have not yet been "accepted" - it is not a limit on the number of people you are talking to - it's just how many can do a connect() before you accept() them

1 *fd returned from socket()*

Backlog for incoming connections 2

```
if (listen(server_fd, 5)) {
     perror("listen");
     return(1);
}
```

3. Accept

- Accept processes client requests (usually in a loop)
- It returns a new socket file descriptor for talking to that particular client

```
struct sockaddr_un client _addr;
int len = sizeof (client_addr);
```

1 *fd returned from socket()*

2 *Address of a client and the length of this address*

```
if ((client_fd = accept(fd, (struct sockaddr *)&client_addr,
                                                    &len)) < 0) {
      perror("accept");
      return(1);
}
```

Client address is recorded into variable *client_addr*

- When accept() returns, the *client_addr* variable will be filled with the remote side's struct *sockaddr_un*, and len will be set to its length
- The new file descriptor client_fd is connected to the client, and is ready for sending and receiving data

```
struct sockaddr_un client _addr;

int len = sizeof (client_addr);


if ((client_fd = accept(fd, (struct sockaddr *)&client_addr, &len)) < 0) {
        perror("accept");
        return(1);
}
```

Read data from a client: example

```
char buf[BUF_SIZE];
if ((len = read(client_fd, buf, BUF_SIZE-1))) < 0) {
    perror("read");
    return(1);
}

// The read is raw bytes.  This turns it into a C string.
buf[len] = '\0';
printf("The other side said: %s\n", buf);
```

Write data to a client: example

```
//echo data back
if (write(client_fd, buf, strlen(buf)) != strlen(buf) ) {
    perror("write");
    return(1);
}
```

Close

- Closing the client_fd makes the other side see that the connection is dropped - server "hang up"

```
close(client_fd);
```

- Unix domain socket binding is reclaimed upon process exit, but the inode is not. You have to explicitly unlink (delete) it

```
close(server_fd);
unlink("/tmp/something");
```

Unix domain socket client

code in *client.c*

Client program: socket setup

- Create a socket interface of type Unix domain socket:

This is the only file descriptor
used by the client to connect to
a remote process

```
if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      return(1);
}
```

Connect to known address

- The client does connect(), the server does accept()
- Fill-in fields of server address:

```
struct sockaddr_un serv_addr;
memset(&serv_addr, '\0', sizeof (serv_addr));
serv_addr.sun_family = AF_UNIX;
strcpy (serv_addr.sun_path, "/tmp/something");
if (connect(fd, (struct sockaddr *)&serv_addr, sizeof (serv_addr))) {
	perror("connect");
	return(1);
}
//at this point we have connected to the server socket successfully
```

Now client can write, as usual

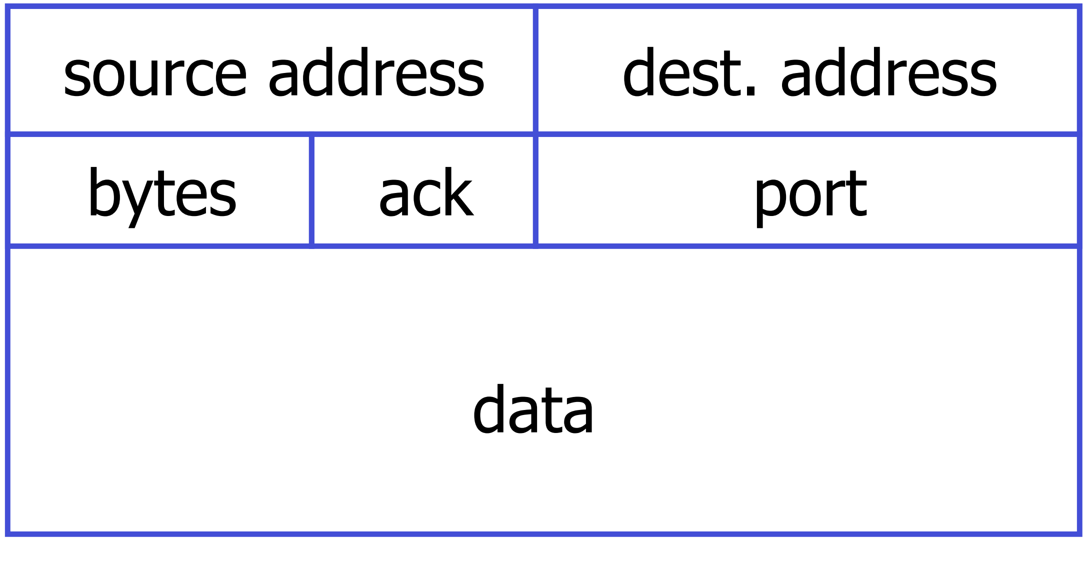
```
if ((len = write(fd, "Hello", 5)) != 5) {
      perror("write");
      return(1);
}
```

Internet sockets

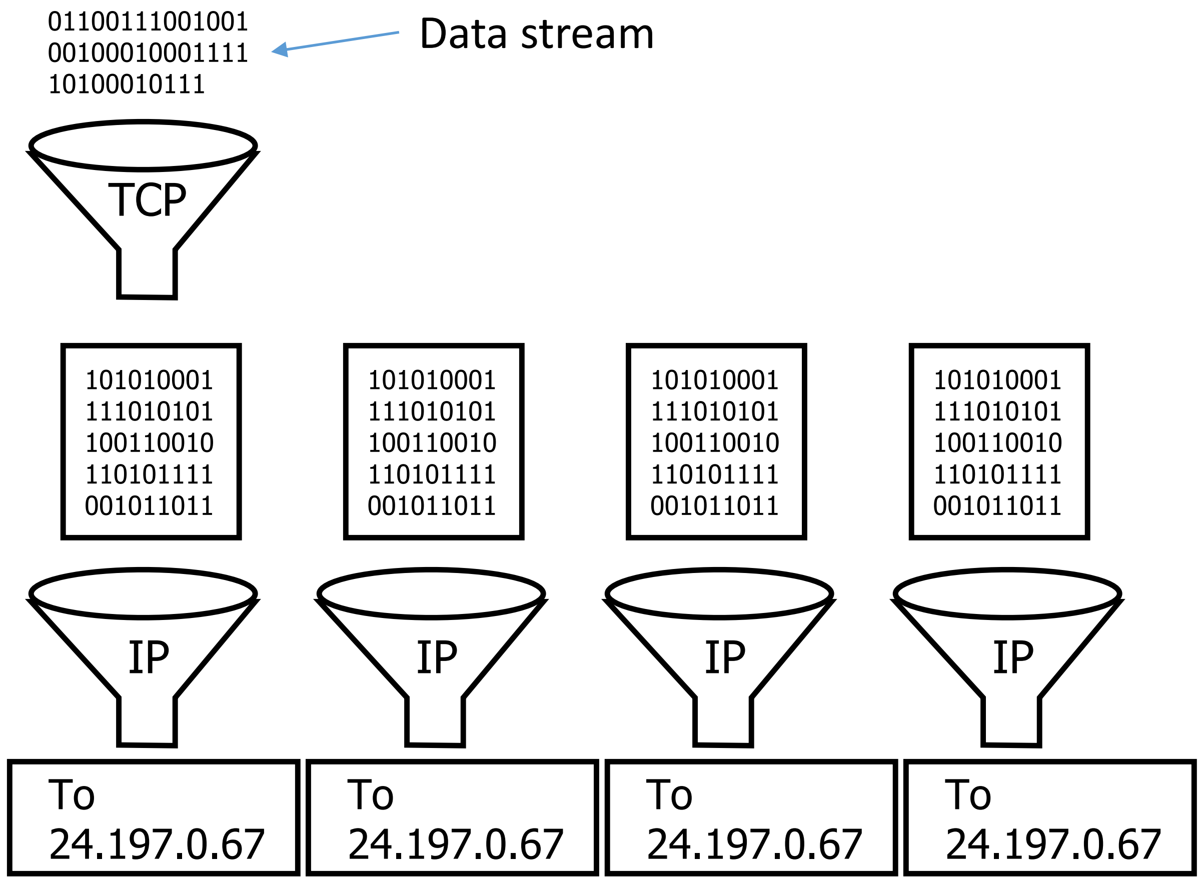
Code examples in server_inet.c and client_inet.c

Internet protocol: 2 layers – TCP/IP

- TCP breaks data into packets and give each packet a header:
  - sequence number
  - checksum
- IP – adds envelope with IP addresses



Preparing stream of data for transmission



TCP: make packets

put in an IP envelope with another header

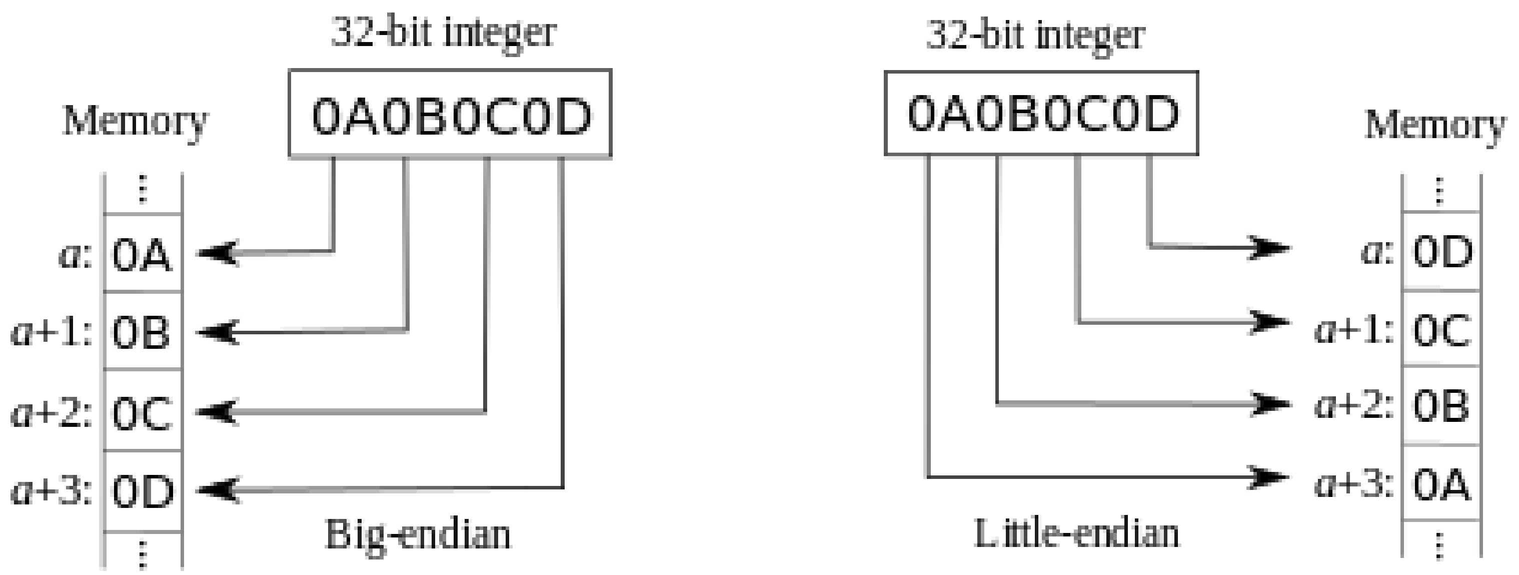
Challenges of Internet sockets

New challenge: Inter-operability

- Number representation
- If transmit as sequence of bytes – new line

- How to ensure proper network communication between heterogeneous operating systems?

Byte order for multi-byte numbers



- Intel is little-endian, and Sparc is big-endian
- The standard network byte order is **big-endian**; a "little-endian" machine must swap bytes in integers when copying them to and from network transmission buffers.

Finding endianness of your machine

Sample code in *my_endian.c*

Converting to network byte order

- To communicate between machines with unknown or different “endian-ness” we convert numbers to network byte order (big-endian) before we send them.
- There are functions provided to do this:

```
unsigned long htonl(unsigned long)
unsigned short htons(unsigned short)
unsigned long ntohl(unsigned long)
unsigned short ntohs(unsigned short)
```

Differences in data representation

- Different computer architectures use different conventions to represent data formats (byte order, size of integer and long, padding structures)
- To exchange data between heterogeneous systems over network – need to put data into agreed-upon format (marshalling protocols)
- A simpler approach: send data as text, as a sequence of bytes

Newline in different O/S

- A text is a sequence of zero or more "lines". A line of a text file is a sequence of zero or more non-newline characters followed by a newline
- Different operating systems have different newline "conventions":
  - The ASCII standard: use single byte number 10 ("control-J", or "line feed" or "LF“)
  - Unix: byte 10 as a "newline character", and we get it in C in Unix by typing "\n“
  - MS-DOS and successors: a two-byte sequence to separate lines: byte 13 and byte 10 ("control-M" and "control-J“) "Control-M" is also known as "carriage return" or "CR". Together, this two byte sequence is called "CRLF“
  - Some other operating systems have other newline conventions

Newline problem for sending data over the network

- In the case of transmitting text, the ASCII standard gives us standard byte values for just about everything except newlines
- So we need to adopt a newline standard for network text transmission

Network new line convention

- The network newline convention is CRLF. That is, a newline is represented by the two bytes (in order) which we could call CR and LF, or control-M and control-J, or 13 and 10, or \015 and \012.

Network new line: \r\n rather than just \n

Internet sockets: the same BLA

- Internet socket setup follows the same sequence of actions:
  - Socket
  - Bind
  - Listen
  - Accept

Before starting reading and writing

- There are only 2 differences:
  - Socket type is now **AF_INET**, not AF_UNIX
  - The **address** needs more details: we are connecting from a different machine

Socket()

- This is the same as before, except it says "INET" instead of "UNIX"

```
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return(1);
}
```

IP addresses

- The IP host address is used to uniquely identify machines connected to the Internet
- It is a 32-bit quantity interpreted as 4 8-bit numbers or octets
- An IP address is usually written in a dotted-decimal notation of the form N1.N2.N3.N4, where each Ni is a decimal number between 0 and 255 decimal

Host names are mapped to unique string names

- Host names in terms of numbers are difficult to remember and hence they are termed by ordinary names such as google.com or yahoo.com
- We need to find out the dotted IP address corresponding to a given name
- The process of finding out dotted IP address from host name is known as *hostname resolution*
- A hostname resolution is done by special software residing on Domain Name Servers (DNS): they keep the mapping of IP addresses and the corresponding ordinary names

In C we can get the real host address with *getaddrinfo*()

Sample code in *showip.c*

run with ./showip <str_name>

To find IP address of your machine (on Linux):

```
/sbin/ifconfig
```

Identifying the process on a host machine with port

- If the client knows the 32-bit Internet address of the host machine, it can contact that host
- To identify the particular server process running on that host we define a ***port number***
- New port number should be an integer between 1024 and 65535:
  - Port numbers smaller than 1024 are considered well-known (telnet uses port 23, http uses 80, ftp uses 21 etc.)
  - You can see port assignments in the file /etc/services
  - In your own application you need to make sure that your port is not assigned to any other service (Any port number more than 5000 is a good choice)

Defining address and port for INET server socket

```
struct sockaddr_in serv_addr;
memset(& serv_addr, '\0', sizeof (serv_addr));
serv_addr.sin_family = AF_INET;

serv_addr.sin_addr.s_addr = INADDR_ANY;
```

This means that if there are more than one IP address for this machine, use any of them

```
serv_addr.sin_port = htons(12345);
```

Note the use of *htons* (host-to-network-short) which converts a given port number to a network format

1. Bind

- For AF_INET sockets, there is no filesystem token representing them - only a list of bound ports/IPs kept track of by the kernel
- Looks exactly the same as with Unix domain sockets

```
struct sockaddr_in serv_addr; //all set up
```

Here we are casting again to struct sockaddr – so it can compile, but the address structure itself is quite different

```
if (bind(serv_fd, (struct sockaddr *)&serv_addr, sizeof (serv_addr)))
{
        perror("bind");
        return(1);
}
```

Returns zero on success

2. Listen

```
if (listen(server_fd, 5)) {
    perror("listen");
    return(1);
}
```

3. Accept

```
struct sockaddr_in client _addr;
int len = sizeof (client_addr);




if ((client_fd = accept(fd, (struct sockaddr *)&client_addr,
                                                &len)) < 0) {
    perror("accept");
    return(1);
}
```

Server ports are sticky

- If you terminate your server program, and then start it again, you may receive the following error message when calling bind():

```
Can't bind the port: Address already in use
```

- When you bind a socket to a port, the operating system will prevent anything else from rebinding to it for the next 30 seconds or so, and that includes the program that bound the port in the first place.
- To get around the problem, you just need to set an option on the socket before you bind it

```
int reuse = 1;
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR,
                                        (char *)&reuse, sizeof(int)) == -1)
        error("Can't set the 'reuse' option on the socket.");
```

Summary: Connection-Oriented (TCP) Stream sockets

Server

- Create a socket: `socket()`
- Assign an address to a socket: `bind()`
- Establish a queue for connections and start listening: `listen()`
- Get a connection from the queue: `accept()`

Client

- Create a socket: `socket()`
- Initiate a connection: `connect()`

Example: multi-client socket server

Code example in *server_inet_multi.c*

- You can connect and test it with a general socket client called *netcat*
- In its client mode, *netcat* will connect to the server specified with name and port, and will send everything that you type on the stdin (keybord)

IP address
of a server

```
netcat -C 128.100.31.200 12345
```

Send \r\n as
line ending

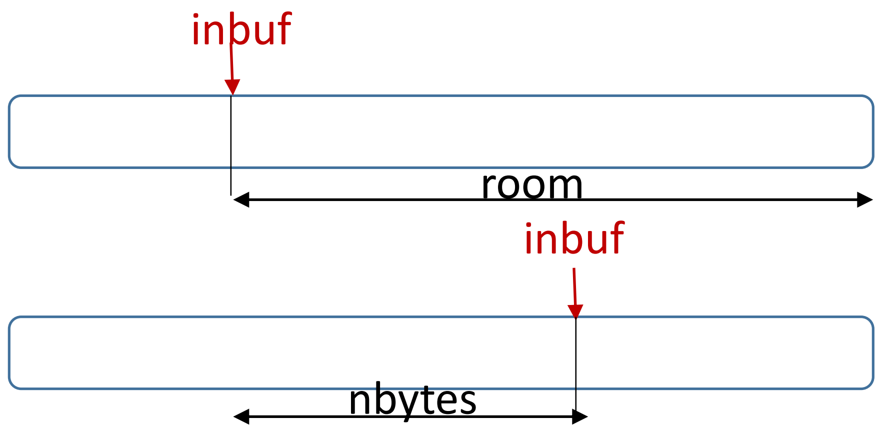
port

Message arrives as a sequence of packets

- In TCP protocol, a single message arrives as a sequence of packets
- If we want to reconstruct the original message lines, we need to parse one line of a message, and keep the beginning of the next line in buffer
- For this, we need keep one pointer for each buffer, to keep track of data length

```
char buf [BUFFER_SIZE];

int inbuf;
```

Parsing partial reads into lines of text: 1/3

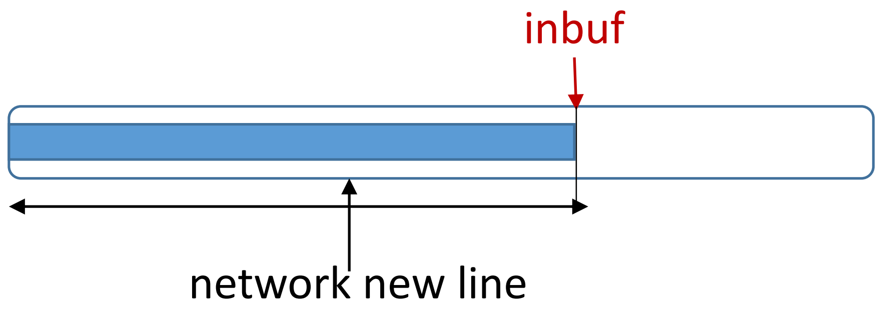


```
char *after = buf + inbuf;

int room = BUFFER_SIZE - inbuf;

int nbytes;
```

Read next piece of data (of size room) from fd into a computed place in buffer

```
if ((nbytes = read(fd, after, room)) > 0) {
      inbuf += nbytes;  //advance inbuf pointer
```

Parsing partial reads into lines of text: 2/3



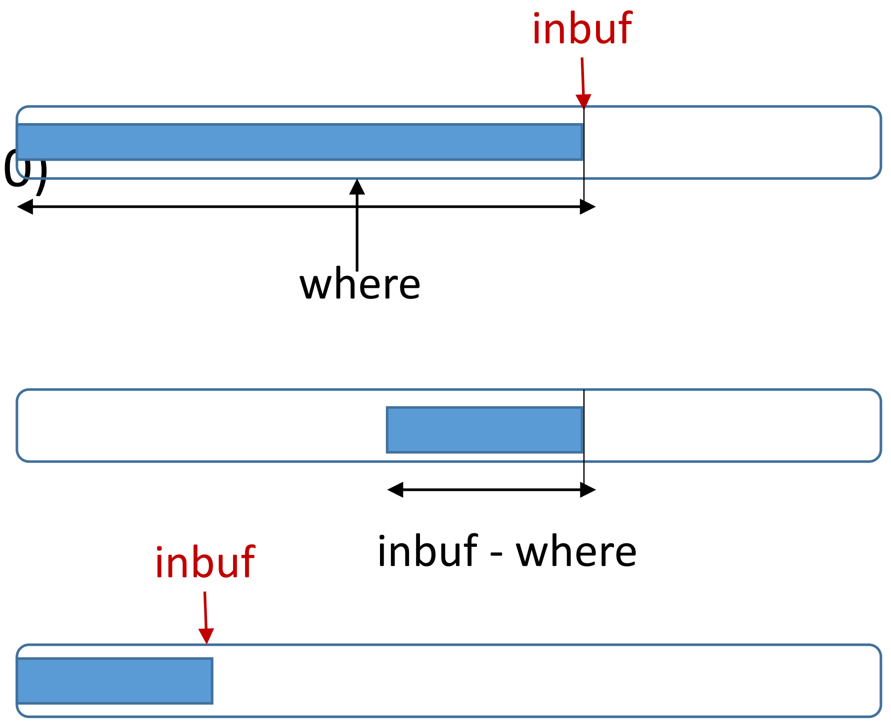
```
if ((nbytes = read(fd, after, room)) > 0)
{
        ...
        int where = find_network_newline (buf, inbuf);

        if (where >= 0) {
                buf[where] = '\0'; buf[where+1] = '\0';
                do_command(buf);


 }
```

Process data in buffer to find a new line

If data contains new line – make a C string and process it

Parsing partial reads into lines of text: 3/3

```
if ((nbytes = read(fd, after, room)) > 0)
{
    …



    if (where >= 0) {
        …
        where+=2;  // skip over \r\n
        inbuf -= where;
        memmove (buf, buf + where, inbuf);
    }
}
```



Move remaining data to the beginning of the buffer for next read

Select()

Sample code in file server_select.c

Blocking

- A *blocking* call does not return to your program until the event you requested has been completed
- Most of system calls in socket programming are blocking

Listening – non-blocking
int status = listen(sockid, queueLimit);

- status: 0 if listening, -1 if error

- Important: listen() is non-blocking: returns immediately.
- The listening socket (sockid) is never used for sending and receiving – it is used by the server only as a prototype for new sockets

Establish connection (in client) - blocking:
int status=connect(sockid, &foreignAddr, addrlen);

- The client establishes a connection with the server by calling connect()
- status: 0 if successful connect, -1 otherwise
- Important: connect() is **blocking**

Server - incoming connection - blocking: int s= accept(sockid, &clientAddr, &addrLen);

- The server gets a socket for an incoming client connection by calling accept()

- Important: accept() is **blocking**: waits for connection before returning

Exchanging data - blocking

```
int count = write (sockid, msg, msgLen);
int count = send(sockid, msg, msgLen, flags);
```

- msg: message to be transmitted
- msgLen: integer, length of message (in bytes) to transmit
- count: # bytes transmitted (-1 if error)

```
int count = read (sockid, recvBuf, bufLen);
int count = recv(sockid, recvBuf, bufLen, flags);
```

- **Calls are blocking - return only after data is sent / received**

Avoiding blocking in complex programs

- For simple programs, blocking is convenient
- What about more complex programs?
    - multiple connections
    - simultaneous sends and receives
    - simultaneously doing non-networking processing

Ways to handle multiple clients without blocking

- Forking a child process for each client
- Processing each client in a separate thread (not covered)
- Using poll() (not covered)

➡ Using *select*()

Select()

- Problem: from which socket the server should accept connections or receive messages?
- Solution: select()
  - specifies a list of descriptors to check for pending I/O operations
  - blocks until one of the descriptors is ready (or timeout)
  - returns which descriptors are ready

Preparing file descriptor sets

- Populate sets of socket descriptors you are interested in using macros
- Once you have the set, you pass it into the function as one of the following parameters:
  - *readfds* if you want to know when any of the sockets in the set is ready to read() data
  - *writefds* if any of the sockets is ready to write() data to
  - *exceptfds* if you need to know when an exception (error) occurs on any of the sockets
- Any of these parameters can be NULL if you're not interested in those types of events

Preparing fd sets: example

FD_SET(int fd, fd_set *set);	Add fd to the set.
FD_CLR(int fd, fd_set *set);	Remove fd from the set.
FD_ISSET(int fd, fd_set *set);	Return true if fd is in the set.
FD_ZERO(fd_set *set);	Clear all entries from the set.

```
fd_set readfds;
// pretend we've accepted two clients at this point: s1 and s2
FD_ZERO(&readfds);

FD_SET(s1, &readfds);
FD_SET(s2, &readfds)
```

Select parameters

int ***select*** (int **n**, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval ***timeout**);

- The first parameter, *n* is the highest-numbered file descriptor to check - plus one
- The last parameter *timeout* tell *select*() how long to check these sets for
- Select returns after the timeout, or when an event occurs, whichever is first

Parameters: example

- Suppose s2 > s1, so we use it for the n :

```
n = s2 + 1;
```

- Wait until either socket has data ready to be read (timeout 10.5 secs)

```
tv.tv_sec = 10;  //seconds
tv.tv_usec = 500000; //microseconds (1,000,000 microseconds in a second)
rv = select(n, &readfds, NULL, NULL, &tv);
```

Select return value

- Returns the number of ready descriptors in the set on success, 0 if the timeout was reached, or -1 on error
- After select() returns, the values in the sets will be changed to show which are ready for reading or writing, and which have exceptions

Return value: example

```
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
        perror("select"); // error occurred in select()
} else if (rv == 0) {
        printf("Timeout occurred!  No data after 10.5 seconds.\n");
} else {
        // one or both of the descriptors have data
        if (FD_ISSET(s1, &readfds))
                read(s1, buf1, sizeof buf1);
        if (FD_ISSET(s2, &readfds))
                recv(s2, buf2, sizeof buf2);
}
```