CST8234 – C Programming
Week13: Introduction to C programming

Sockets System Calls
Used for interprocess communication over a network. They allow a process on one computer to communicate with another process on another computer.

Each process makes some socket system call to set itself up for sending and receiving data. Some other system call are made to do the actual send / receive data and finally, each process makes a system call to terminate the communication.

Three simple steps:

(1) Connect
(2) Send / receive
(3) Terminate
Similar to the basic file I/O, open, read/write, close.

Basic Network Concepts and System Commands
IP
A network interface is identified by its Internet Protocol (IP) addresses. IPv4 uses a 32-bit identifier and IPv6 uses a 128-bit identifier. Using the Domain Name System (DNS) an IP address is given a human readable name.

Ports
A port is an identifier on a network device (computer in this case) thought which network process communication takes place. If a computer has multiple processes involved in separate network communications at the same time, each is assigned a different port number. A port is identified by a 16-bit, nonnegative integer. Port values between 0-1023 are generally reserved for traditional services. For example, telnet uses port 23, web server uses 80. The port values in the range 1024-49151 are generally assigned to applications. The port values in the range 49152-65535 are unreserved.

The ports used on each of the two devices do not need to match. When establishing the connection, the calling device must know both the IP address of the device it wishes to call and the port number of the process to which it wishes to communicate.

To find the IP address in a Linux box:

```
root@luna:CST8234# ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:428588 errors:0 dropped:0 overruns:0 frame:0
          TX packets:428588 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:30108515 (30.1 MB)  TX bytes:30108515 (30.1 MB)
wlan0     Link encap:Ethernet  HWaddr 7c:7a:91:27:1d:20
          inet addr:192.168.0.19  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::7e7a:91ff:fe27:1d20/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:17176634 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8814993 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:24083143393 (24.0 GB)  TX bytes:1329471983 (1.3 GB)
```

`nslookup` can be used to look up the DNS name of an IP address, or vice versa.

```
root@luna:CST8234# nslookup 8.8.8.8
Server:      127.0.0.1
Address:     127.0.0.1#53

Non-authoritative answer:
8.8.8.8.in-addr.arpa      name = google-public-dns-a.google.com.
```

netstat can be used to display all the ports on a system.

```
root@luna:CST8234# netstat -ta
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 luna.local:51817        198.183.167.120:http    ESTABLISHED
tcp        0      0 localhost:54896         localhost:37392         ESTABLISHED
tcp        0    508 luna.local:58439        ipv4_1.lagg0.c034.:http ESTABLISHED
tcp        0      0 luna.local:33496        sc-in-f113.1e100.n:http ESTABLISHED
tcp        1      0 luna.local:36693        mulberry.canonical:http CLOSE_WAIT
```

Client-Server Model

A process in one computer acts as a server and a process on the second computer acts as a client.

The server opens up a port for listening and waits for a client to attempt to establish a connection. The client calls the server by connecting to the port on which the server is listening. When establishing a connection, a process must identify the IP and the port to which it wishes to communicate.

socket( )

```
NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

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

DESCRIPTION
       socket() creates an endpoint for communication and returns a descriptor.

       [ Partial man page – output omitted ]
```

First step in establishing a network communication is to create a socket. It provides an integer identifier through which a network communication is going to take place.

```
void error(char *msg) {
   perror(msg);
    exit( EXIT_FAILURE );
}

int sockfd;
if ( ( sockfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 )
            error( "Open Socket" );
```

`socket( )` requires 3 arguments:

domain: AF_UNIX --> Unix internal protocol -- same file system
AF_INET --> Internet, any two host

type_socket:
SOCK_STREAM --> sequence two way connection based on bytes
SOCK_DGRAM --> datagrams, info read in chunks

protocol:
if 0 --> OS choose

The system call returns a valid file descriptor for the new socket on success or -1 on error.

bind( )

The second step depends on which the process will act as a server or a client. A server will bind the socket, defining the IP and port on which it will listen for connections.

```
 struct sockaddr_in serv_addr;

/*
 * Init serv_add to zeros
 */
bzero( (char *) &serv_addr, sizeof(serv_addr) );

portno = 55555;

/*
 * struct sockaddr_in {
 *        short   sin_family;
 *        u_short sin_port;
 *        struct  in_addr sin_addr;
 *        char    sin_zero[8];
 * };
 *
 * s_add --> ANY IP address of the host
 */
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port        = htons(portno);

/*
 * Step 2:  Bind
 * bind( ) binds a socket to an address
 */

if ( bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
       error("ERROR on binding");
```

The `struct sockaddr_in` holds information about the connection, including the IP and the port number. The structure must be zeroed out (all bytes in the structure must be set to zero). The structure is then filled with information about how the socket will be used. The `htons( )` function ensures that bytes are in the correct order for network transport.

The value of `INADD_ANY` indicates that the socket should be bound to the IP of the machine on which the process is currently running.

listen( )
Use to wait for communication. The second parameter of the function describes how many connections can be queued while the server is handling another communication.

```
listen( sockfd, NCLIENTS );
```

If the `sockfd` is a valid socket, this function can not fail!

connect( )
A client actively makes a call, establishing a connection.

```
struct sockaddr_in serv_addr;
struct hostent *server;

bzero((char *) &serv_addr, sizeof(serv_addr));

serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
      (char *)&serv_addr.sin_addr.s_addr, server->h_length);
serv_addr.sin_port = htons(portno);

if (connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0)
             error("ERROR connecting");
```

The structure `sockadd_in` is filled with information about the server the client wishes to connect.

accept( )
Once a server has received an incoming connect attempt, it can accept the connection.

```
if ( (newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen)) < 0 )
             error("ERROR on accept");
```

The `accept( )` function returns a second socket on which the data is to be transmitted. This allow the original socket to continue to listen for additional connections.

send( ) / recv()
After the connection has been established between the server and the client, data can be transmitted and received.

`send()` requires four arguments, the socket id, and address point to data, the number of bytes to send and a flag setting.

`secv()` arguments are similar to `send()`, except the third parameter indicates the maximum number of bytes that can be received.

This functions are similar to the `fread( )` and `fwrite()` functions in that the arguments define an address and a number of bytes.

Alternative, you can do a low level I/O using `read( )` and `write()` instead.

close()
Once communication is finished, both server and client should close their respective sockets.