Socket System Calls

Section 7.5

Introduction

- **Socket system calls** are used for programs to communicate with each other over a network
- Code-wise, similar to working with files
  - Open, read / write, close
  - Connect, receive / send, close

Network Concepts

- A **network** connects computers together so they can communicate with each other
- Each computer on the network has a unique identifier called an **IP address** (**Internet Protocol**)
- An IP address looks like four numbers, each between 0 and 255, inclusive, separated by dots
  - 192.168.0.100

- Might be difficult to remember IP address of website
- Have **domain names**
  - google.com
  - ucmo.edu
- **DNS server** (**Domain Name System**) translates domain names to the correct IP address

- A **socket** is like an outlet or a slot for a connection
  - Logical end point of a connection
- Connections also involve a **port**
- A **port** is an integer that identifies what service is being connected for

- Ports in the range 0 – 1023 are reserved for standard services
  - 80 is web / HTTP
  - 22 is secure shell
  - 23 is telnet
- Custom programs we make should use ports above 1023

- Consider a call center analogy
  - Phone number is the IP address
  - Phones (yours and the answering one at the call center) are the sockets
  - Port is the number you enter to the automated system for why you called or what you want to do

Networking Commands

- Some common Unix networking commands include:
  - **ifconfig** – display information about networking devices
  - **netstat** – display ports currently being used
  - **iptables** – firewall

Client Server Model

- Common paradigm in networking to designate two different roles to networked computers
- A **server** provides some service and receives connections
- A **client** connects to the server
- Typically, one server might have many clients connecting to it

- When writing networking programs, generally going to have two separate programs
  - Server program
  - Client program
- We can run them on the same computer, but generally they would be run on different computers

socket()

- Before a connection can be made, a **socket** must be created
- socket() takes in a few parameters for the type of connection the socket is for
- socket() gives back an integer indicating the socket that was created (or negative number if failed)

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

int sock;
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

if (sock < 0)
{
    printf("socket() failed\n");
}
```

bind()

- After a socket is created, a server will usually bind the socket to prepare to listen for incoming connections
  - Generally, client does not do this step
- Basically, associates the server socket with which port to listen on
- **bind()**'s input includes the socket number and a structure containing the IP address of the server and the port number to listen on

```
int i;
int sock;
struct sockaddr_in my_addr;
unsigned short int listen_port = 60000;

/* Create the socket */

memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(listen_port);

i = bind(sock, (struct sockaddr *)&my_addr, sizeof(my_addr));

if (i < 0)
{
    printf("bind() failed\n");
}
```

memset()

- **memset()** is a function for setting a chunk of memory to a particular value
- Commonly used to zero-out the memory of a structure
  - **memset(*ptr*, *value*, *size*)**
  - *ptr* is the address to start setting
  - *value* is what to set in that memory
  - *size* is how much memory from the address to set

struct sockaddr_in

- **struct sockaddr_in** is a structure for representing an Internet address
- Has a member for the address family, called **sin_family**, which is essentially always set to **AF_INET**
- Also has an integer member for the port, called **sin_port**
- And has a **struct in_addr** member for the IP address, called **sin_addr**

htons(), htonl(), etc.

- Variables of many data types are comprised of multiple bytes
- If we have to send those bytes one by one, do we start at the most significant byte or the least significant byte?
- This is referred to as **endianness**
- Storing the most significant byte first is called **big endian**, where as storing the least significant byte first is called **little endian**
  - Diagram: https://en.wikipedia.org/wiki/Endianness

- Sometimes the order on the local computer is different than the network, so we have to convert to the order the network expects
  - If not, our data might be out of order
- **htons()** stands for host to network short and **htonl()** stands for host to network long

listen()

- Once the socket has been created and bound to a port, the server can start listening to incoming connections
- **listen()**'s input is the socket to listen on and the number of connection that can be queued

```
int i;
int sock;

/* Create the socket */

/* Bind the socket */

i = listen(sock, 5);

if (i < 0)
{
    printf("listen() failed\n");
}
```

accept()

- The server will essentially pause on **accept()** until a client tries to connect
- Once a client tries to connect, the server will return a socket number from accept()
  - This socket represents the client that just connected

```
int i;
int sock;
int sock_recv;
int addr_size;
struct sockaddr_in recv_addr;

/* Create the socket */
/* Bind the socket */
/* Listen for a connection */

addr_size = sizeof(recv_addr);

sock_recv = accept(sock, (struct sockaddr *)&recv_addr, &addr_size);
```

connect()

- So far been mostly looking at code from the server
- On the client side, client is still going to create a socket, but instead of binding and listening, the client is going to **connect**
- **connect()**'s input includes the socket to connect from and the IP address and port to connect to

```
int i;
int sock;
struct sockaddr_in addr_send;
char *server_ip = "130.127.24.92";
unsigned short int server_port = 60000;

/* Create the socket */
memset(&addr_send, 0, sizeof(addr_send));
addr_send.sin_family = AF_INET;
addr_send.sin_addr.s_addr = inet_addr(server_ip);
addr_send.sin_port = htons(server_port);

i = connect(sock, (struct sockaddr *)&addr_send, sizeof(addr_send));

if (i < 0)
{
    printf("connect() failed\n");
}
```

send() and recv()

- Once a connection exists between the server and client, either of the two can use **send()** and **recv()** to transmit data across the network
  - **send(*socket*, *data*, *sizeOfData*, *optionFlag*)**
    - Usually the *optionFlag* is left as 0
  - Returns the number of bytes sent

```
int sock;
int bytes_sent;
char text[80];

/* Create a socket and connect */

printf("Send? ");
scanf("%s", text);
bytes_sent = send(sock, text, strlen(text), 0);
```

- recv() works similarly to send(), but instead receives data
  - recv(*socket*, *data*, *sizeOfData*, *optionFlag*);
- Server and client program have to be on the same page about who is sending and receiving what

```
int sock_recv;
int bytes_received;
char text[80];

/* Create and bind socket, accept connection */

bytes_received = recv(sock_recv, text, 80, 0);
text[bytes_received] = 0;
printf("Received: %s\n", text);
```

close()

- When done with communication, should close the connection
- **close()**'s input is a socket number

```
int i;
int sock_recv;
int bytes_received;
char text[80];

/* Done with communication*/

i = close(sock_recv);

if (i < 0)
{
    printf("close() failed\n");
}
```

- **server.c** and **client.c**
  - Example client and server
  - Client sends text to server
  - Client can tell server to shutdown