Lesson 7

TCP/IP:

TCP/IP enables communication between processes on different computers/networks.

Every computer that is connected to the internet has an IP address that is anchored in the TCP/IP (for example: 132.70.1.4). Every computer has communication entries called *ports*.
TCP/IP enables to set up a stream between a process on computer A and a process on computer B. This stream enables the transfer of data (like pipe). The communication is ***bi-directional*** (can read and write to the stream). Usually, one end is the ***server*** and the other end is the ***client***.

Sockets:
The sockets enable the communication between processes on the same computer or on different computers that are connected to the Internet. Basically, sockets are a way to speak to other programs using Unix file descriptors.

Definition: A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

Using Sockets:
Data Structures:

```
int sd;    // socket descriptor

struct in_addr
{
        unsigned long     s_addr; // s_addr contains the IP of the computer (32 bit long, or 4 bytes)
}

sturct sockaddr
{
        unsigned short    sa_family;        // address family: AF_xxx
        char              sa_data[14];      // computer's address and port number
}
```

sa_family: there are two ways to communicate: AF_INET (communication across the net - what will be used in class) and AF_UNIX (local communication on the same computer).

We can use the following struct instead:

```
struct sockaddr_in
{
   short int              sin_family; // like sa_family above. We'll use AF_INET
   unsigned short int     sin_port; // port number - Network Byte Order!
   struct in_addr         sin_addr; // computer's address (internet address) - Network Byte Order!
   unsigned char          sin_zero[8]; // for alignments - same size as struct sockaddr
}
```

sin_port: If 0 - chooses in unused port at random.
sin_addr: If INADDR_ANY it uses the local address of the computer.
sin_zero: Used to enable conversion between *struct sockaddr_in** and *struct sockaddr**. *sin_zero* should be set to all zeros.

Note: there are two byte orderings: most significant byte ("Network Byte Order"), or least significant byte. Some machines store their numbers internally in Network Byte Order, and some don't. Some of the numbers must be in Network Byte Order and not in Host Byte Order. Then, it must be converted.

To do this we can use:
unsigned long htonl(unsigned long hostlong); // host to network long (32 bits)
unsigned short htons(unsigned short hostshort); // host to network short (16 bits)
unsigned long ntohl(unsigned long netlong); // network to host long
unsigned short ntohs(unsigned short netshort); // network to host short

Reamrk:
Unix doesn't work with an IP address in the form of 4 decimal number (for example: 132.70.1.4). The IP addresses are saved in a *in_addr_t* structure. There's a function that converts IP address in the form of 4 decimal numbers to *in_addr_t* structure:

#include <arpa/inet.h>

in_addr_t inet_addr(const char *ip_address); /* return -1 in failure. Upon success - returns the address in Network Byte Order */

For example:

```
sockaddr_in in;
in.sin_addr.s_addr = inet_addr("132.70.1.4");
```

Client-Server model:
One process which job is to give the service and waits for connections, and other clients that connect to the server and ask for services. The client is the initiator of the connection.

If a computer runs several servers (ftp, http, …), when a client wants to connect to the ftp server on that computer it first needs to connect to the computer's IP and then to connect to the desired server. How?
Each address in TCP/IP model contains internal address as well - ***port***. Every port is an integer number.
Ports 1-1024 are system ports and should not be used by programs.
IP address is a 32 bit unsigned integer. The *port* is a 16 bit unsigned integer.
So, when a client wants to connect to a server it needs to specify IP address and a port number.

The Server:
The server has a socket that is associated to a certain port. The server process sends and receives the data from the socket.
1. ***socket:*** defines the socket from which read/write will be done.
2. ***bind:*** associates the socket to a specific port.
3. ***listen:*** makes it possible to start accepting connections to the server. This command also defines how many "call waiting" is possible (usually it is defined to 5, i.e., the first request will be accepted and if another 4 are requesting in the same type - they will be blocked until we finish the connection process of the first client. However, if a 6 client requests connection it will get a "connection refused" message).
4. ***accept:*** accepts a connection. This command blocks the process until some client asks for services.
5. ***read/write.***
6. ***close:*** closes the socket.

How the server can handle several clients?
The server will play the role of an operation: the clients will connect to it. The *accept* commands return a new socket that the server doesn't use. So the server will use *fork* and the "child" server will use the socket and will handle all the communication with this client.
In general, only the connection operations are done with the main socket, and all the other read/write actions are done using other sockets that are returned by *accept*.

The Client:
1. ***socket:*** defines the socket.
2. ***connect:*** the command receives IP address and port number and it initiates a connection from the socket to the process located at (IP, port). We don't need to define the port number from which the socket of the client connects since we don't need it. The OS decides from which port to connect.
3. ***read/write***
4. ***close.***

After the connection, both sides can read (receive) or write (send) data. This is done by ***read*** or ***recv***, and ***write*** or ***send***:

ret = read(sd, buf, len); // read data from the socket to the buffer
ret = recv(sd, buf, len, flag); // if flag == 0 it's just like read
ret = write(sd, buf, len); // write data from the buffer to the socket

I/O Multiplexing:
Using *select()* enables multiplexing several I/O channels. It enables to serve many clients using one process, even if each request takes a lot of time. It also enables server to listen for incoming connections as well as keep reading from the connections it already has.
You have to use *select()* since *accept()* and *recv()* block until data is ready…

Detailed Information:

SYNOPSIS
#include <sys/socket.h>

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

DESCRIPTION
socket() allocates a new socket.
format is either AF_INET or AF_UNIX. We'll always use AF_INET.
type is either SOCK_STREAM (for connection oriented communication, for example: TCP. Think of connection-oriented communication as a *telephone line*) or SOCK_DGRAM (for connection-less oriented communication, for example: UDP. Think of connectionless-oriented communication as *sending mail*). We'll always use SOCK_STREAM.
protocol is the protocol the socket will use. It's usually always 0 (and then it is by the default definitions: SOCK_STREAM uses TCP, and SOCK_DGRAM uses UDP);

RETURN VALUES
Upon successful return the *socket descriptor* is returned. Otherwise, -1 is returned.

Some explanation about SOCK_STREAM and SOCK_DGRAM: *(taken from Beej's guide)*
Stream sockets are *reliable* 2-way connected communication streams (the system takes care of errors). If you output 2 items into the socket in the order "1, 2", they will arrive in the order "1, 2" at the opposite end. They will also be error free. This is achieved using the TCP protocol. Stream sockets are used in ***telnet*** programs, ***web browsers***,
Datagram sockets are unreliable: if you send a datagram, it may arrive. It may arrive out-of-order. It may not arrive at all. If it arrivers, the data within the packet will be error free. Datagram sockets use UDP protocol. As opposed to stream sockets, you do not need to maintain an open connection. You just build a packet, slap an IP header on it with destination information, and send it out.

SYNOPSIS
int bind(int sd, struct sockaddr *sock_addr, int addr_len);

DESCRIPTION
bind() connects to the socket *sd* the port in *sock_addr*. *addr_len* is the size of *struct sockaddr*. *sd* is the socket descriptor *socket* returns and the *sock_addr* struct needs to be initialized before this call.

RETURN VALUES
0 upon success, -1 upon failure.

Example: *(taken from Beej's guide)*
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT 3490

```
main()
{
        int sockfd;
        struct sockaddr_in my_addr;

        sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!

        my_addr.sin_family = AF_INET; // host byte order
        my_addr.sin_port = htons(MYPORT); // short, network byte order
        my_addr.sin_addr.s_addr = inet_addr("132.70.1.4");
        memset( &(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

        // don't forget your error checking for bind
        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
        …
}
```

SYNOPSIS
int listen(int sd, int backlog);

DESCRIPTION
listen() defines the size *backlog* of the waiting queue of processes that want to open a connection with the server. That is, the number of connections allowed on the incoming queue (incoming connections are going to wait on the queue until you *accept()* them).
sd is the socket descriptor returned by *socket()*.

RETURN VALUES
0 upon success, -1 upon failure.

SYNOPSIS
int accept(int sd, void *client_addr, int *addr_len);

DESCRIPTION
accept() is done by the server. *accept()* causes to wait on the socket for incoming connection. The address of the process who sent the connection request is saved in *client_addr*. *addr_len* is a pointer to the size of *client_addr*.
sd is the listening socket descriptor.
In connection oriented communication the server doesn't usually need to know that address of the client and we can write NULL in the last 2 fields.

RETURN VALUES
Upon successful return, **a new socket descriptor is returned**. This sd will be used for communication only with the process who initiated the connection. Otherwise, -1 is returned.

Note: After *accept()* you have 2 sd's. The original one is still listening on the port and the newly created one is ready to *send()* and *recv()*.

Example: *(taken from Beej's guide)*

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

#define MYPORT 3490 // the port users will be connecting to
#define BACKLOG 10 // how many pending connections queue will hold

main()
{
        int sockfd, new_fd; // listen on sockfd, new connection on new_fd
        struct sockadd_in my_addr; // my address information
```

```
        struct sockadd_in their_addr; // connector's address information
        int sin_size;

        sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!

        my_addr.sin_family = AF_INET; // host byte order
        my_addr.sin_port = htons(MYPORT); // short, network byte order
        my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
        memset( &(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

        // don't forget your error checking for connect
        bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
        listen(sockfd, BACKLOG);

        sin_size = sizeof(struct sockaddr_in);
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        …
}
```

SYNOPSIS
int connect(int sd, struct sockaddr *serv_addr, int addr_len);

DESCRIPTION
connect() is done by the client. *connect()* tries to connect to the server *serv_addr* which contains the destination port and IP address. *addr_len* is the size of *struct sockaddr*. *sd* is the socket descriptor returned from *socket()*.

RETURN VALUES
Upon successful return, 0 is returned. Otherwise, -1 is returned.

Example: *(taken from Beej's guide)*

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

#define DEST_IP "132.70.1.4"
#define DEST_PORT 3490

main()
{
        int sockfd;
        struct sockadd_in dest_addr;

        sockfd = socket(AF_INET, SOCK_STREAM, 0); // do some error checking!

        dest_addr.sin_family = AF_INET; // host byte order
        dest _addr.sin_port = htons(DEST_PORT); // short, network byte order
        dest _addr.sin_addr.s_addr = inet_addr(DEST_IP);
        memset( &( dest _addr.sin_zero), '\0', 8); // zero the rest of the struct

        // don't forget your error checking for connect
        connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
        …
}
```

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

DESCRIPTION
select() is given 3 arguments of type *fd_set*. Each one represents a group of file descriptors which *select()* monitors. The fourth argument represents time duration. After this time the function must return. If it's NULL the call isn't limited in time.
When calling *select()* the *readfds* group must contain file descriptors from which we want to read; the *writefds* group must contain file descriptors to which we want to write; the *execptfds* need to contain file descriptors that we want to know of exceptions or errors in.
If you don't care waiting for a certain set, you can pass an empty group using NULL.
The first argument, *max_fd_plus_one*, needs to be the highest number of descriptor in any of the groups, plus 1.

For example:
If you want to see if you can read from stdin and some socket descriptor, sockfd, just add the file descriptors ***0*** and ***sockfd*** to the set *readfds*. In this example will set *max_fd_plus_one* to be equals to ***sockfd + 1*** (which is definitely higher than stdin - 0).
When *select()* returns, *readfds* will be modified to reflect which of the file descriptors you selected is ready for reading. You can test it using FD_ISSET.

To manage the file descriptors group we can use 4 macros:

FD_CLR(int fd, fd_set *set);	// remove a descriptor from the group
FD_ISSET(int fd, fd_set *set);	// check if a descriptor is in the group
FD_SET(int fd, fd_set *set);	// add a descriptor to the group
FD_ZERO(fd_set *set);	// nullifying (clear) the file descriptor group.

RETRUN VALUES
When the function returns every one of the groups contain a subgroup of descriptors that we can read from, write to, or had errors.
The return value of the function represents the number of descriptors that are ready in the 3 groups.
0 is returned if the function returned due to *timeout* and if no file is ready.
-1 is returned upon error.

Example: *(taken from Beej's guide)*

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0 // fd for stdin

int main() // the following code waits 2.5 seconds for something to appear on stdin
{
        struct timeval tv;
        fd_set readfds;
        tv.tv_sec = 2;
        tv.tv_usec = 500000; // micro-secs

        FD_ZERO(&readfds);
        FD_SET(STDIN, &readfds);

        // don't care about writefds and exceptfds
        select(STDIN + 1, &readfds, NULL, NULL, &tv);

        if (FD_ISSET(STDIN, &readfds))
                printf("A key was pressed!\n");
        else
                printf("Timed out\n");
}
```

SYNOPSIS
struct hostent *gethostbyname(const char *hostname);

```
struct hostent {
        char    *h_name; // official name of the host
        char    **h_aliases; // a NULL-terminated array of alternate names for the host
        int     h_addrtype; // the type of address being returned; usually AF_INET
        int     h_length; // the length of the address in bytes
        char    **h_addr_list; /* a zero-terminated array of network addresses for host. Host
                                  addresses are in Network Byte Order. */
}
#define h_addr h_addr_list[0]       // the first address in h_addr_list.
```

DESCRIPTION
gethostbyname() is used to find the address of the computer by its name.

RETURN VALUES
Upon successful return, the function returns a pointer to a structure that describes the desired computer. Otherwise, NULL is returned and the global variable *h_errno* is set to indicate the error.

Example:

```
if ( ( hp = gethostbyname(argv[1])) == NULL)
{
      herror("gethostbyname");
      exit(1);
}
printf("Connection information: \n");
printf("host name: %s\n", hp->h_name);
printf("IP address: %s\n", inet_ntoa(*((struct in_addr *)hp->h_addr)));
```

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

size_t recv(int sd, void *buf, size_t length, int flags);

DESCRIPTION
sd is the socket from which we read the information. *buf* is to where to read the information. *length* is how many bytes to read.
flags influence on the way the data is read. It is formed by *or*'ing one or more of the values:
1. MSG_PEEK: the process can "peek" at the information without really receiving it.
2. MSG_OOB: ignores regular information and only receives "out of band" information, for example: interrupt signals.
3. MSG_WAITALL: *recv()* will end only when the whole requested data is available (blocked).

If *flag* is **0** it behaves like *read()*.

RETURN VALUES
Upon successful return, the function returns the number of bytes read. Otherwise, -1 is returned.
If *recv()* returns 0 it means that the remote side has closed the connection.

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

size_t send(int sd, const void *msg, size_t length, int flags);

DESCRIPTION
sd is the socket to which we send the information. *msg* is from where to send the information. *length* is how many bytes to send.
flags influence on the way the data is sent. It is formed by *or*'ing one or more of the values:

1. MSG_OOB: send only "out of band" information.
2. MSG_DONTROUTE: the message will be sent ignoring routing conditions of the protocol. It usually means that the message will be sent using the most direct route and not necessarily the fastest one.

If *flag* is **0** it behaves like *write()*.

RETURN VALUES
Upon successful return, the function returns the number of bytes read. Otherwise, -1 is returned.

Important:

- If a process tries to write or send information to a closed socket a SIGPIPE signal will be returned.
- If *read* or *recv* returns 0 it means we got to EOF and thus to the end of the connection. That's why it's important to check the return value of those functions.

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

size_t sendto(int sd, const void *msg, size_t length, int flags, const struct sockaddr *to, int tolen);

DESCRIPTION
Like *send()* with the following additions:
to is a pointer to a *struct sockaddr* which contains the destination IP address and port.
tolen is set to *sizeof(struct sockaddr)*.

RETURN VALUES
Upon successful return, the function returns the number of bytes actually sent. Otherwise, -1 is returned.

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

size_t recvfrom(int sd, const void *buf, size_t length, int flags, struct sockaddr *from, int *fromlen);

DESCRIPTION
Like *recv()* with the following additions:
from is a pointer to a local *struct sockaddr* that will be filled with the IP address and port of the origination machine.
fromlen is pointer to local int that should be initialized to *sizeof(struct sockaddr)*. When the function returns, *fromlen* will contain the length of the address actually stored in *from*.

RETURN VALUES
Upon successful return, the function returns the number of bytes actually received. Otherwise, -1 is returned.

SYNOPSIS
int close(int sd);

DESCRIPTION
closes the connection.

RETURN VALUES
Upon successful return, 0 is returned. Otherwise, -1 is returned.

compilation: gcc -lsocket -lnsl -o my_server server.c (same for client)

Note: bind() might fail and return "Address already in use". That's because a socket that was connected is still hanging around in the kernel, and it's hogging the port. You can either wait for it to clear, or allow your program to reuse it by writing:

```
char yes = '1';
if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))) == -1)
{
        perror("setsockopt");
        exit(1);
}
```

server.c

```
 1  /* server process */
 2  #include <ctype.h>
 3  #include <sys/types.h>
 4  #include <sys/socket.h>
 5  #include <netinet/in.h>
 6  #include <signal.h>
 7
 8  #define PORTNUM 2000 // the port clients will be connecting to
 9  #define BACKLOG 10 // how many pending connections queue will hold
10
11  void catcher(int sig);
12  int ns; // new socket from accept
13
14  main(int argc, char* argv[])
15  {
16          int s; // the main socket
17          char c;
18          struct sockaddr_in server = {AF_INET, PORTNUM, INADDR_ANY};
19
20          /* another option:
21          server.sin_family = AF_INET;
22          server.sin_port = htons(PORTNUM);
23          server.sin_addr.s_addr = htonl(INADDR_ANY); // my IP
24          bzero( &(server.sin_zero), 8); // the function bzero zeroes out a buffer of the specified length
25          */
26
27          signal(SIGPIPE, catcher); // to catch a death of a client
28          signal(SIGCHLD, catcher); // to wait for finished child (avoiding zombies)
29          signal(SIGTERM, catcher); // to free the ports that we've used when program terminates
30
31          // create a socket
32          if ( ( s = socket(AF_INET, SOCK_STREAM, 0) ) < 0)
33          {
34                  perror("socket");
35                  exit(1);
36          }
37
38          // bind the socket to an address
39          if ( bind(s, (struct sockaddr*)&server, sizeof(server) ) < 0)
40          {
41                  perror("bind");
42                  exit(1);
43          }
44
45          // start listening for incoming connections
46          if ( listen(s, BACKLOG) < 0)
47          {
48                  perror("listen");
49                  exit(1);
50          }
51
52          for ( ; ; )
53          {
54                  // accept a connection
55                  if ( ( ns = accept(s, NULL, NULL) ) < 0)
56                  {
57                          perror("accept");
58                          continue;
```

```
59                      }
60
61                      switch ( fork() ) // spawn a child to deal with the connection
62                      {
63                              case -1:
64                                      perror("fork");
65                                      break;
66                              case 0: // child: send and receive information to/from the client
67                                      while ( recv(ns, &c, 1, 0) > 0)
68                                      {
69                                              c = toupper(c);
70
71                                              if ( send(ns, &c, 1, 0) < 0)
72                                              {
73                                                      perror("send");
74                                                      exit(1);
75                                              }
76                                      }
77                                      close(ns);
78                                      exit(0);
79                              default: // parent
80                                      printf("got connection\n");
81                      }
82                      // at parent: close ns and continue in the loop
83                      close(ns);
84              }
85              exit(0);
86      }
87
88      void catcher(int sig)
89      {
90              switch (sig)
91              {
92                      case SIGCHLD:
93                              signal(SIGCHLD, catcher); // to catch future children
94                              printf("client is done\n");
95                              wait(NULL);
96                              break;
97                      case SIGPIPE:
98                              printf("client died\n");
99                              close(ns);
100                             exit(0);
101                     case SIGTERM:
102                             exit(0);
103             }
104     }
```

line 83: We want that when the child closes *ns* the client will receive EOF. If there is still a process that didn't close *ns* this won't happen. That's why the parent needs to close *ns* as well.

line 92: When trying to write to a socket that the client closed → broken pipe.

client.c

```
 1  /* client process */
 2  #include <stdio.h>
 3  #include <ctype.h>
 4  #include <sys/types.h>
 5  #include <sys/socket.h>
 6  #include <netinet/in.h>
 7  #include <arpa/inet.h>
 8  #include <netdb.h>
 9
10  main(int argc, cha *argv[])
11  {
12          int s;
13          char c;
14          struct sockaddr_in sa;
15          struct hostent *hp;
16          int port;
17
18          if (argc != 3)
19          {
20                  fprintf(stderr, "Usage: %s server port\n", argv[0]);
21                  exit(-1);
22          }
23
24          if ( ( hp = gethostbyname(argv[1])) == NULL)
25          {
26                  fprintf(stderr, "%s: Invalid host name\n", argv[1]);
                    /* better use:
                    herror("gethostbyname");
                    */
27                  exit(-1);
28          }
29
30          port = atoi(argv[2]);
31
32          if (port < 1024)
33          {
34                  fprintf(stderr, "%s: Invalid port number\n", argv[1]);
35                  exit(1);
36          }
37
38          bzero( &(sa.sin_zero), 8);
39          sa.sin_family = AF_INET;
40          bcopy(hp -> h_addr, &sa.sin_addr.s_addr, hp -> h_length);
41          /* another option: if the IP of the server is known we can use:
42          sa.sin_addr.s_addr = inet_addr("1.2.3.4");
43          */
44
45          sa.sin_port = port;
46
47          // create a socket
48          if ( (s = socket(AF_INET, SOCK_STREAM, 0) ) < 0)
49          {
50                  perror("socket");
51                  exit(1);
52          }
53
54          // connect socket to servers'
55          if (connect(s, (struct sockaddr *)&sa, sizeof(sa)) < 0)
```

```
56          {
57                  perror("connect");
58                  exit(1);
59          }
60
61          while ( (c = getchar()) != EOF) // send and receive information to/from the server
62          {
63                  if (  send(s, &c, 1, 0) < 0)
64                  {
65                          perror("send");
66                          exit(1);
67                  }
68
69                  switch( recv(s, &c, 1, 0) )
70                  {
71                          case -1:
72                                  perror("recv");
73                                  exit(1);
74                          case 0:
75                                  fprintf(stderr, "server died\n");
76                                  close(s);
77                                  exit(1);
78                          default:
79                                  putchar(c);
80                  }
81          }
82          close(s);
83          exit(0);
84  }
```

line 40: using *bcopy()* to copy the IP address to the appropriate place in the *sa* strcuture
hp -> h_addr: the field in *hp* where the IP address resides.
&sa.sin_addr.s_addr: the copy destination (the address field in *sa*).
hp->h_length: how many bytes to copy.
bcopy copies a specified number of bytes from a source to a target buffer.
The *bcopy* and *bzero* functions are functions dealing with arrays of bytes.

Running Example:

```
1 > gcc server.c -lxnet -o server
2 > gcc client.c -lxnet -o client
3 > server &                    // running the server in background
[1] 28525
4 > client sunlight 2000            //  running  the  client  with  the
computer name and port number
got connection
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Have a nice day
HAVE A NICE DAY
^D    // sending EOF
client done

5 > client sunlight 2000
got connection
server still working
SERVER STILL WORKING
^D
client done
6 > jobs
[1] + Running     server
7 > kill -9 %1
[1] Killed  server
```

Example: *(taken from Beej's guide)*

This program acts like a simple multi-user chat server. Start it running in one window, then telnet to it (telnet hostname 9034) from multiple other windows. When you type something in one telnet session, it should appear in all the others.

selectserver.c

```
1   #include <stdio.h>
2   #include <stdilb.h>
3   #include <string.h>
4   #include <unistd.h>
5   #include <sys/types.h>
6   #include <sys/socket.h>
7   #include <netinet/in.h>
8   #include <arpa/inet.h>
9
10  #define PORT 9034 // port we're listening on
11
12  int main()
13  {
14          fd_set master;      // master file descriptor list
15          fd_set read_fds;   // temp file descriptor list for select()
16          struct sockaddr_in myaddr;              // server address
17          struct sockaddr_in remoteaddr;       // client address
18          int fdmax;          // maximum file descriptor number
19          int listener;       // listening socket descriptor
20          int newfd;          // newly accept()ed socket descriptor
21          char buf[256];      // buffer for client data
22          char yes = '1';     // for setsockopt() SO_REUSEADDR, below
23          int addrlen;
24          int i, j;
25
26          FD_ZERO(&master);       // clear master and temp sets
27          FD_ZERO(&read_fds);
28
29          // get the listener
30          if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
31                  perror("socket");
32                  exit(1);
33          }
34
35          // lose the pesky "address already in use" error message
36          if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
37                  perror("setsockopt");
38                  exit(1);
39          }
40
41          // bind
42          myaddr.sin_family = AF_INET;
43          myaddr.sin_addr.s_addr = INADDR_ANY;
44          myaddr.sin_port = htons(PORT);
45          memset(&(myaddr.sin_zero), '\0', 8);
46          if (bind(listener, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1) {
47                  perror("bind");
48                  exit(1);
49          }
50
51          // listen
52          if (listen(listener, 10) == -1) {
```

```
53                  perror("listen");
54                  exit(1);
55          }
56
57          // add the listener to the master set
58          FD_SET(listener, &master);
59
60          // keep track of the biggest file descriptor
61          fdmax = listener; // so far, it's this one
62
63          // main loop
64          for (; ;) {
65             read_fds = master; // copy it
66             if (select(fdmax + 1, &read_fds, NULL, NULL, NULL) == -1) {
67                perror("select");
68                exit(1);
69             }
70
71             // run through the existing connections looking for data to read
72             for (i = 0; i <= fdmax; i++) {
73               if (FD_ISSET(i, read_fds)) { // we got one!!
74                  if (i == listener) {
75                     // handle new connections
76                     addrlen = sizeof(remoteaddr);
77                     if ((newfd = accept(listener, (struct sockaddr *) &remoteaddr, &addrlen)) == -1) {
78                              perror("accept");
79                     } else {
80                              FD_SET(newfd, &master); // add to master set
81                              if (newfd > fdmax) // keep track of the maximum
82                                       fdmax = newfd;
83                              printf("selectserver: new connection from %s on socket %d\n",
84                                       inet_ntoa(remotaddr.sin_addr), newfd);
85                     }
86                  } else {
87                     // handle data from a client
88                     if (nbytes = recv(i, buf, sizeof(buf, 0)) <= 0) {
89                              // got error or connection closed by client
90                              if (nbytes == 0) {
91                                       // connection closed
92                                       printf("selectserver: socket %d hung up\n", i);
93                              } else {
94                                       perror("recv");
95                              }
96                              close(i); // bye!
97                              FD_CLR(i, &master); / remover from master set
98                     } else {
99                              // we got some data from a client
100                             for (j = 0; j <= fdmax; j++) {
101                                      // send to everyone!
102                                      if (FD_ISSET(j, &master)) {
103                                               // except the listener and ourselves
104                                               if (j != listener && j!= i) {
105                                                        if (send(j, buf, nbytes, 0) == -1) {
106                                                                 perror("send")
107                                                        }
108                                               }
109                                      } // end if - FD_ISSET
110                             } // end for - j
111                    } // end else - recv
112                 } // end else - i == listener
```

```
113                     } // end if - FD_ISSET
114                 } // end for - i <= fdmax
115             } // end for (; ;)
116     } // end main
```

Explanation:

There are 2 file descriptor sets: *master* and *read_fds*. *master* holds all the socket descriptors that are currently connected, as well as the socket descriptor that is listening for new connections.

We need 2 sets because *select()* actually **changes** the set you pass into it to reflect which sockets are ready to read. Since there is a need to keep track of the connections from one call of *select()* to the next, it must be stored somewhere. At the last minute, we copy the *master* into the *read_fds* and then call *select()*.

Note that every time a new connection is made you need to add it to the *master* set, and every time a connection closes, you have to remove it from the *master* set.

We check to see when the *listener* socket is ready to read. When it is, it means we have a new connection pending, and we *accept()* it and add it to the *master* set. Similarly, when a client connection is ready to read, and *recv()* returns 0, we know the client has closed the connection, and we must remove it from the *master* set.

If the client *recv()* returns non-zero, then some data has been received. So we get it and then go through the *master* list and send that data to all the rest of the connected clients.

Client-Server - Using Threads instead of fork() - Exampels:

A standard socket server should listen on a socket port and, when a message arrives, fork a process to service the request. Since a *fork()* system call would be used in a nonthreaded program, any communication between the parent and child would have to be done through some sort of interprocess communication.
We can replace the *fork()* call with a *thr_create()* call. Doing so offers a few advantages: *thr_create()* can create a thread much faster than a *fork()* could create a new process, and any communication between the "server" and the new thread can be done with **common variables**. This technique makes the implementation of the socket server much easier to understand and should also make it respond much faster to incoming requests.
The server program first sets up all the needed socket information. This is the basic setup for most socket servers. The server then enters an endless loop, waiting to service a socket port. When a message is sent to the socket port, the server wakes up and creates a new thread to handle the request. Notice that the server creates the new thread as a **detached thread** and also passes the socket descriptor as an argument to the new thread.
The newly created thread can then read or write, in any fashion it wants, to the socket descriptor that was passed to it. At this point the server could be creating a new thread or waiting for the next message to arrive. The key is that the server thread does not care what happens to the new thread after it creates it.

Example:
In this example (taken from the *Threaded Primer* book by Daniel Berg and Bill Lewis), the created thread reads from the socket descriptor and then increments a global variable. This global variable keeps track of the number of requests that were made to the server. Notice that a **mutex lock** is used to protect access to the shared global variable. The lock is needed because many threads might try to increment the same variable at the same time. The mutex lock provides serial access to the shared variable. See how easy it is to share information among the new threads! If each of the threads were a process, then a significant effort would have to be made to share this information among the processes.
The client piece of the example sends a given number of messages to the server. This client code could also be run from different machines by multiple users, thus increasing the need for concurrency in the server process.

soc_server.c:

```
 1  #define _REENTRANT
 2  #include <stdio.h>
 3  #include <sys/types.h>
 4  #include <sys/socket.h>
 5  #include <netinet/in.h>
 6  #include <string.h>
 7  #include <sys/uio.h>
 8  #include <unistd.h>
 9  #include <pthread.h>
10
11  /* the TCP port that is used for this example */
12  #define TCP_PORT   6500
13
14  /* function prototypes and global variables */
15  void *do_chld(void *);
16
17  mutex_t lock;
18  int         service_count;
19
20  main()
21  {
22              int         sockfd, newsockfd, clilen;
23              struct sockaddr_in cli_addr, serv_addr;
24              pthread_t chld_thr;
25
```

```
26          if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
27          {
28                  fprintf(stderr,"server: can't open stream socket\n");
29                  exit(1);
30          }
31
32          memset((char *) &serv_addr, 0, sizeof(serv_addr));
33          serv_addr.sin_family = AF_INET;
34          serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
35          serv_addr.sin_port = htons(TCP_PORT);
36
37          if(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
38          {
39                  fprintf(stderr,"server: can't bind local address\n");
40                  exit(0);
41          }
42
43          /* set the level of thread concurrency we desire */
44          thr_setconcurrency(5);
45
46          listen(sockfd, 5);
47
48          for(;;){
49                  clilen = sizeof(cli_addr);
50                  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
51
52                  if(newsockfd < 0)
53                  {
54                          fprintf(stderr,"server: accept error\n");
55                          exit(0);
56                  }
57
58                  /* create a new thread to process the incomming request */
59                  thr_create(NULL, 0, do_chld, (void *) newsockfd, THR_DETACHED, &chld_thr);
60
61                  /* the server is now free to accept another socket request */
62          }
63          return(0);
64  }
65
66  /*
67          This is the routine that is executed from a new thread
68  */
69
70  void *do_chld(void *arg)
71  {
72          int     mysocfd = (int) arg;
73          char    data[100];
74          int     i;
75
76          printf("Child thread [%d]: Socket number = %d\n", thr_self(), mysocfd);
77
78          /* read from the given socket */
79          read(mysocfd, data, 40);
80
81          printf("Child thread [%d]: My data = %s\n", thr_self(), data);
82
83          /* simulate some processing */
84          for (i=0;i<1000000*thr_self();i++);
85
```

```
86              printf("Child [%d]: Done Processing...\n", thr_self());
87
88              /* use a mutex to update the global service counter */
89              mutex_lock(&lock);
90                      service_count++;
91              mutex_unlock(&lock);
92
93              printf("Child thread [%d]: The total sockets served = %d\n", thr_self(), service_count);
94
95              /* close the socket and exit this thread */
96              close(mysocfd);
97              thr_exit((void *)0);
98      }
```

soc_client.c:

```
 1  #include <stdio.h>
 2  #include <sys/types.h>
 3  #include <sys/socket.h>
 4  #include <netinet/in.h>
 5  #include <arpa/inet.h>
 6  #include <string.h>
 7  #include <unistd.h>
 8  #include <stdlib.h>
 9
10  #define TCP_PORT   6500
11  #define SERV_HOST_ADDR   "11.22.33.44"
12
13  main(int argc, char **argv)
14  {
15              int         i, sockfd, ntimes = 1;
16              struct sockaddr_in serv_addr;
17              char buf[40];
18
19              memset((char *) &serv_addr, 0, sizeof(serv_addr));
20              serv_addr.sin_family = AF_INET;
21              serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
22              serv_addr.sin_port = htons(TCP_PORT);
23
24              if (argc == 2)
25                      ntimes = atoi(argv[2]);
26
27              for (i=0; i < ntimes; i++) {
28                      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
29                      {
30                              perror("clientsoc: can't open stream socket");
31                              exit(0);
32                      }
33
34                      if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
35                      {
36                              perror("clientsoc: can't connect to server");
37                              exit(0);
38                      }
39
40                      printf("sending segment %d\n", i);
41                      sprintf(buf, "DATA SEGMENT %d", i);
42                      write(sockfd, buf, strlen(buf));
43                      close(sockfd);
44              }
```

```
45              return(0);
46      }
```

select-example.c: (http://joda.cis.temple.edu/~ingargio/old/cis307s96/readings/unix3.html)
The third is a simple program where a process reads fixed size messages from two pipes. You should terminate the program from the terminal with a CONTROL-C.

```
 1  #include <sys/types.h>
 2  #include <sys/time.h>
 3  #include <unistd.h>
 4  #include <stdio.h>
 5  #define MAXLINE 12
 6
 7  int main(void)
 8  {
 9              int     i, m, n;
10              int     fd1[2];   /* pipe for communications from child1 to parent */
11              int     fd2[2];   /* pipe for communications from child2 to parent */
12              pid_t   pid;
13              char    line[MAXLINE];
14              fd_set   readset;
15              static struct timeval timeout;
16
17              if (pipe(fd1) < 0) {
18                      perror("pipe1");
19                      exit(1);
20              }
21              if (pipe(fd2) < 0) {
22                      perror("pipe2");
23                      exit(1);
24              }
25
26              /* child1 */
27              if ( (pid = fork()) < 0) {
28                       perror("fork1");
29                       exit(1);
30              }
31              else if (pid == 0) {
32                       close(fd1[0]);
33                       while (1) {
34                                write(fd1[1], "from child1\n", MAXLINE);
35                                sleep(1);
36                       }
37                       exit(0);
38              }
39
40              /* child2 */
41              if ( (pid = fork()) < 0) {
42                      perror("fork2");
43                      exit(1);
44              }
45              else if (pid == 0) {
46                       close(fd2[0]);
47                       while (1) {
48                        write(fd2[1], "from child2\n", MAXLINE);
49                       sleep(1.3);
50                       }
51                       exit(0);
52              }
```

```
53              /* parent */
54              close(fd1[1]);
55              close(fd2[1]);
56              m = 1 + ((fd1[0]<fd2[0])?fd2[0]:fd1[0]); /*Max # of objs to wait for*/
57              FD_ZERO(&readset);
58              while (1) {
59                 FD_SET(fd1[0], &readset);
60                 FD_SET(fd2[0], &readset);
61                 if (select(m,&readset, NULL,NULL,NULL) < 0) {
62                   perror("select");
63                   exit(1);
64                 }
65
66                 if (FD_ISSET(fd1[0], &readset)) {
67                           n = read(fd1[0], line, MAXLINE);
68                           write(STDOUT_FILENO, line, n);
69                 }
70
71                 if (FD_ISSET(fd2[0], &readset)) {
72                           n = read(fd2[0], line, MAXLINE);
73                           write(STDOUT_FILENO, line, n);
74                 }
75              }
76              exit(0);
77   }
```

Appendix: Connection Oriented Sequence: