Experiment -2

Familiarity of Socket system calls

1.1 Aim: To identify and demonstrate the use of socket system calls

1.2. Description:

IPC mechanisms: Allows the processes to communicate. Linux supports the following IIPC mechanisms:

- PIPES
- FIFOS
- Message Queues
- Shared Memory
- Sockets and
- RPC

Sockets: Provide a standard interface between the network and application. Allow communication between processes on the same machine or different machines. A socket is a communication end point. It basically contains an IP address and Port number. socket in Unix/Linux domain provide communication between the processes in the same machine.

Types of sockets: There are four types of sockets, they are

1. **Stream Sockets**: Provide reliable byte stream transport service. These sockets guarantee the delivery of packets and the order in a network environment. Uses TCP (Transmission Control Protocol). Data records do not have any boundaries.
2. **Datagram Sockets**: Delivery in a network environment is not guaranteed. Uses UDP(User Datagram Protocol)
3. **Raw Sockets**: Provide access to the underlying communication protocols. These sockets normally datagram-oriented, though their exact characteristics are dependent on the interface provided by the protocol. These are not intended for general user, but used for the development of new communication protocols or gaining access to some of the more cryptic facilities of an existing protocol.
4. **Sequenced Packet Sockets:** these are similar to stream sockets, with the exception that record boundaries are preserved. These sockets allow the user to manipulate the Sequenced Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with the data.

Client Process: Typically a process which makes a request for information. After getting the response, a client process may terminate or may do some other processing. Ex. Web browser

Server Process: Is a process which takes a request from the clients and serves it. After getting a request from the client, this process will perform the required processing, gather the requested information and send it to the client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests. Ex: HTTP server, SMTP server, DNS, mail server etc.

Types of Servers: Servers can be:

1. **Iterative Server**: Serves its clients one after other. They cannot serve clients simultaneously and may be implemented when the service time s finite and short time then they can be implemented as iterative servers. Ex: Time server, DNS etc.
2. **Concurrent Server**: Is a server that can serve multiple client requests simultaneously. Simplest way to create concurrent servers, we use *fork ()* system call.

Socket system calls: In a client server environment, both the client and server have to create sockets for communication. The standard API for network programming in C is Berkely Sockets. This API was first introduced in 4.3BSD UNIX and now available on all Unix-like platforms including Linux, MacOS X, Free BSD and Solaris. A very similar network API is available on Windows.

Server side: socket(), bind(), listen(), accept(), close()
Client side: socket(), connect() and close()
Both sides: recv()/read(), send()/write()

Socket addresses structure in internet domain:

```
struct sockaddr{
     unsigned short sa_family;
     char           sa_data[14];
};

struct in_addr{                        //32-bit network and host ID
         unsigned long s_addr;
};
struct sockaddr_in{
  {
      short int      sin_family; //family
      unsigned short sin_port;   //16-bit port number in network byte order
      struct in_addr sin_addr;   //32-bit IP address in network byte order
      unsigned char  sin_zero[8];
};
```

Include files: <netinet/in.h> *//socket address structure*
#include<sys/types.h> *//user defined types*
#include<sys/socket.h> *//socket related*

socket(): Used to creates an unnamed socket. Used by both server an client

(src: https://www.tutorialspoint.com/unix_sockets/socket_core_functions.htm)

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

returns socket file descriptor(positive), otherwise returns –1 on error.

Family	: AF_INET/AF_INET6/AF_UNIX/AF_NS/AF_IMPLINK/AF_ROUTE
Type	: SOCK_STREAM/SOCK_DGRAM/SOCK_RAW/SOCK_SEQPACKET
Protocol	: IPPROTO_TCP/IPPROTO_UDP/IPPROTO_SCTP

bind(): Used to assign an address to an unnamed socket. For a server on the internet, the address contains IP address and port number. Used by server and connection-less client

```
int bind(int sfd, struct sockaddr *addr, int addrlen);
```

listen(): Listens for the connections. Used by server only.

```
int bind(int sfd, struct sockaddr *addr, int addrlen);
```

accept(): Used by a connection-oriented client to accept a connection request. This system call blocks the server process until a client connects with the server.

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

listen(): Listens for the connections. Used by server.

```
int bind(int sfd, int backlog);
```

read() and write(): Used by client and servers to exchange data

```
int read(int sfd, char*buff, int size);
int write(int sfd, char*buff, int size);
```

close(): Close a socket.

```
int close(int sfd);
```

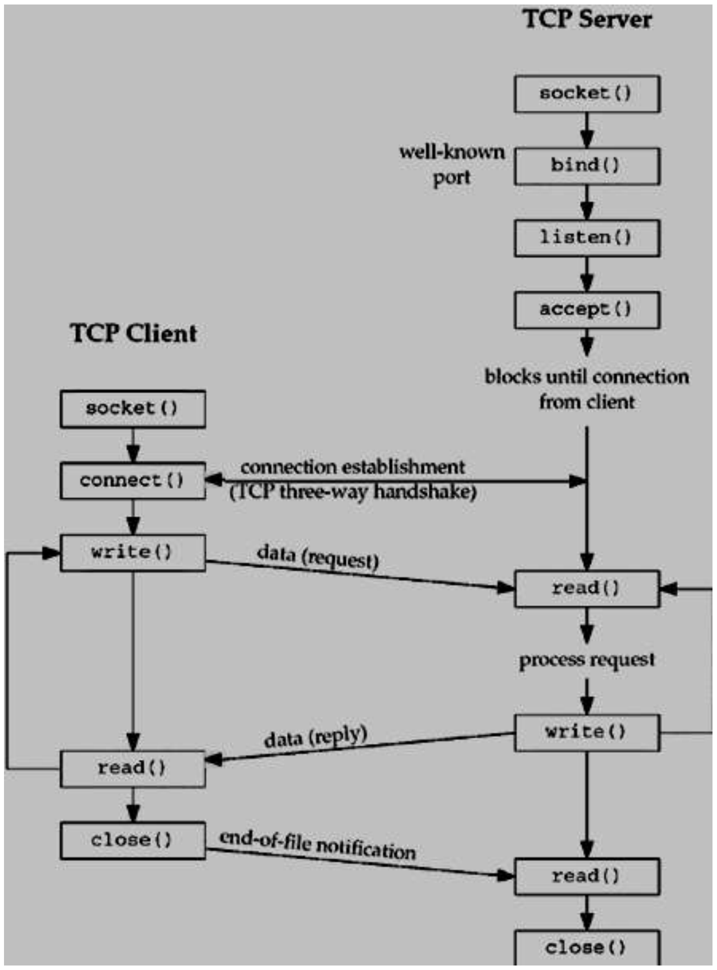

Figure 1: Connection-oriented Client_Server interaction

recv() and send(): Used by client and servers to exchange data

```
int recv(int sfd, char*buff, int size, int flag);
int send(int sfd, char*buff, int size, int flag);
```

connect(): Used by a connection-oriented client to request a connection.

```
int connect(int sfd, struct sockaddr *addr, int addrlen);
```

address conversion functions:

```
#include<arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr)
in_addr_t inet_addr(const char *strptr)
char *inet_ntoa(struct in_addr inaddr)
```

Sample code fragments to demonstrate elementary socket system calls

//Program to create a socket. File name: *socket.c*

```
.........
int main()
{
   . . . . . .
   sfd=socket(AF_INET, SOCK_STREAM,0);      //Create a TCP socket
   if(sfd<0)                                //check for error
    {
      perror("socket() error:");
      exit(-1);
    }
   printf("Socket created and its ID=%d\n",sfd);
   return 0;
}
```

```
$gcc socket.c –o sock
$./sock
```

//Program to demonstrate bind() system call

```
int main(int argc, char *argv[])
{
   int sfd,s;
   struct sockaddr_in servaddr;
   sfd=socket(AF_INET, SOCK_STREAM,0);      //creates a socket
   if(sfd<0)
    {
      perror("socket() error:");
      exit(-1);
    }
    servaddr.sin_family=AF_INET;            //Internet family
    printf("\nBefore bind:\n");
    printf("IP=%s/t Port No.= %d\n ", inet_ntoa(servaddr.sin_addr), ntohs(servaddr.sin_port));
    servaddr.sin_port=htons(atoi(argv[1])); //convert the port number into network byte order
    servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
                                 //extracts the machine IP address and convert it into network byte order
    //servadr.sin_addr.s_addr=inet_addr("172.16.0.100");
                                 //converts decimal dotted address 172.16.0.100 to network byte order
    printf("\nAfter inet_Addr: \n");
    printf("IP=%s/t Port No.=%d\n",inet_ntoa(servaddr.sin_addr),ntohs(servaddr.sin_port));
    s=bind(sfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
                                        //binds the server with the IP address and  port number
    if(sfd<0)                                  //check the return status of bind()
     {
      perror("bind() error:");
      exit(-1);
     }
    printf("Server is binded with the specified address.............:\n");
    printf("\nAfter bind(): \n");
    printf("IP=%s/t Port No.=%d\n",inet_ntoa(servaddr.sin_addr),ntohs(servaddr.sin_port));
    return 0;
}
```

//Program to implement a simple echo server. File name: echo_server.c

```
. . . . . . . .
int main(int argc, char *argv[])
{
   int sfd,newsfd,s,len,n,i; char buff[100];
   struct sockaddr_in servaddr, ca,client;

   sfd=socket(AF_INET, SOCK_STREAM,0);                        //creates a socket
   if(sfd<0)
    {
      perror("socket() error:");
      exit(-1);
    }

    servaddr.sin_family=AF_INET;
    servaddr.sin_port=htons(atoi(argv[1]));
                                   //convert port address given from command line into network byte order
    servaddr.sin_addr.s_addr=htonl(INADDR_ANY);      //extracts the machine IP address
    //servaddr.sin_addr.s_addr=inet_addr("172.16.0.100");

    s=bind(sfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
                                                  //bind the server with the specified address

    if(s<0)
     {
      perror("bind() error:");
      exit(-1);
     }
     s=listen(sfd,5);                             //tells its readiness to TCP
     if(s<0)
     {
      perror("listen() error:");
      exit(-1);
     }
     printf("Server is ready and waiting for client requests..........\n");

     newsfd=accept(sfd,(struct sockaddr*)&ca,&len); //server blocks fr connection request
     if(newsfd<0)
     {
       perror("accept() error:");
       exit(-1);
     }
     printf("Connected to client........");
     n=recv(newsfd,&buff,100,0);                  //reads data into 'buff' from the scoket
     buff[n]='\0';
     printf("\nMessage from client: %s",buff);    //displays the data on to the terminal
     for(i=0;buff[i]!='\0';++i)
          buff[i]=toupper(buff[i]);                //converts the data in the 'buff' to upper case
     send(newsfd,&buff, n,0);                     //writes the data from 'buff' over to  socket.
     close(sfd); close(newsfd);                   //close the connection
return 0;
}
```

```
//Program to implement a simple echo-client. File name:echo_client.c
. . . . . . .
int main(int argc, char *argv[])
{
   int sfd,newsfd,s,len,n; char buff[MAX+1];
   struct sockaddr_in servaddr,sa;
   sfd=socket(AF_INET, SOCK_STREAM,0);              //creates an unnamed socket
   if(sfd<0)
    {
      perror("socket() error:");
      exit(-1);
    }
    servaddr.sin_family=AF_INET;
    servaddr.sin_port=htons(atoi(argv[1]));
    servaddr.sin_addr.s_addr=inet_addr("172.16.0.100");
    s=connect(sfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
                                                    //requests for connection with the server (172.16.0.100)
    if(s<0)
     {
      perror("connect() error:");
      exit(-1);
     }
     write(1,"Enter a message: ",17);
     n=read(0,&buff,100);                           //read a message from keyboard (i.e. From user)
     send(sfd,&buff,n,0);                           //writes the message over to the socket from the buffer
     n=recv(sfd,&buff,MAX,0);                       //reads a message from the socket into the buffer
     write(1,"Received from server: ",22);
     write(1,&buff,n);                              //display the message on the terminal
     close(sfd);
     return 0;
}
```

```
$gcc echo_server.c –o eserv          //compile echo-server program and store in eserv.
$gcc echo_client.c –o  ecli          //compile echo-client program and store in ecli.
```

- On one terminal run server (**$./eserv 5050**) by passing the port number and another terminal run the client program (**$./ecli 5050**).

Task:

1. Modify the above programs to exchange 'n' messages
2. Take the snapshots of the results and put them in the report
3. Write the conclusions and submit the record.

Conclusions:

References:

1. http://www.tutorialspoint.com/unix_system_calls/socket.htm socket tutorials
2. Richard Stevens, "UNIX Network Programming", PHI