Experiment 2

To familiarize and understand the use and functioning of System Calls used for Operating system and network programming in Linux.

Some system calls of Linux operating systems

1. Ps

This command tells which all processes are running on the system when ps runs.

ps -ef

UID	PID	PPID	C	STIME TTY	TIME	CMD
root	1	0	0	13:55 ?	00:00:01	/sbin/init
root	2	0	0	13:55 ?	00:00:00	[kthreadd]
root	3	2	0	13:55 ?	00:00:00	[ksoftirqd/0]
root	4	2	0	13:55 ?	00:00:01	[kworker/0:0]
Root	5	2	0	13:55 ?	00:00:00	[kworker/0:0H]
root	7	2	0	13:55 ?	00:00:00	[rcu_sched]
root	8	2	0	13:55 ?	00:00:00	[rcuos/0]

This command gives processes running on the system, the owners of the processes and the names of the processes. The above result is an abridged version of the output.

2. fork

This system call is used to create a new process. When a process makes a fork system call, a new a process is created which is identical to the process creating it. The process which calls fork is called the parent process and the process that is created is called the child process. The child and parent processes are identical, i.e, the child gets a copy of the parent's data space, heap and stack, but have different physical address spaces. Both processes start execution from the line next to the fork. Fork returns the process id of the child in the parent process and returns 0 in the child process.

```
#include<stdio.h>
void main()
{
int pid;
pid = fork();
if(pid > 0)
{
printf (“ Iam parent\n”);
}
else
{
printf(“Iam child\n”);
}
}
```

The parent process prints the first statement and the child prints the next statement.

3. exec

New programs can be run using exec system calls. When a process calls exec, the process is completely replaced by the new program. The new program starts executing from its main function.

A new process is not created, process id remains the same, and the current process's text, data, heap, and stack segments are replaced by the new program. exec has many flavors one of which is execv.

execv takes two parameters. The first is the pathname of the program that is going to be executed. The second is a pointer to an array of pointers that hold the addresses of arguments. These arguments are the command line arguments for the new program.

4. wait

When a process terminates, its parent should receive some information regarding the process like the process id, the termination status, amount of CPU time taken etc. This is possible only if the parent process waits for the termination of the child process. This waiting is done by calling the wait system call. When the child process is running, the parent blocks when wait is called. If the child terminates normally or abnormally, wait immediately returns with the termination status of

the child. The wait system call takes a parameter which is a pointer to a location in which the termination status is stored.

5. Exit

When exit function is called, the process undergoes a normal termination.

6. open

This system call is used to open a file whose pathname is given as the first parameter of the function. The second parameter gives the options that tell the way in which the file can be used.

open(filepathname , O_RDWR);

This causes the file to be read or written. The function returns the file descriptor of the file.

7. read

This system call is used to read data from an open file.

read(fd, buffer, sizeof(buffer));

The above function reads sizeof(buffer) bytes into the array named buffer. If the end of file is encountered, 0 is returned, else the number of bytes read is returned.

8. write

Data is written to an open file using write function.

write(fd, buffer, sizeof(buffer));

System calls for network programming in Linux

1. Creating a socket

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

This system call creates a socket and returns a socket descriptor. The domain parameter specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <sys/socket.h>. In this program the AF_INET family is used. The type parameter indicates the communication semantics. SOCK_STREAM is used for tcp

connection while SOCK_DGRAM is used for udp connection. The protocol parameter specifies the protocol used and is always 0. The header files used are <sys/types.h> and <sys/socket.h>.

Experiment 3

Implementation of Client-Server communication using Socket Programming and TCP as transport layer protocol

Aim: Client sends a string to the server using tcp protocol. The server reverses the string and returns it to the client, which then displays the reversed string.

Description:

Steps for creating a TCP connection by a client are:

1. Creation of client socket

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

This function call creates a socket and returns a socket descriptor. The domain parameter specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <sys/socket.h>. In this program, the domain AF_INET is used. The socket has the indicated type, which specifies the communication semantics. SOCK_STREAM type provides sequenced, reliable, two-way, connection based byte streams. The protocol field specifies the protocol used. We always use 0. If the system call is a failure, a -1 is returned. The header files used are sys/types.h and sys/socket.h.

2. Filling the fields of the server address structure.

The socket address structure is of type struct sockaddr_in.

```
struct sockaddr_in {

u_short sin_family;
u_short sin_port;
struct in_addr sin_addr;
char sin_zero[8]; /*unused, always zero*/
};
struct in_addr {

u_long s_addr;

};
```

The fields of the socket address structure are
sin_family which in our case is AF_INET
sin_port which is the port number where socket binds
sin_addr which is the IP address of the server machine

The header file that is to be used is **netinet/in.h**

```
struct sockaddr_in servaddr;
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(port_number);
```

Why htons is used ?. Numbers on different machines may be represented differently (big-endian machines and little-endian machines). In a little-endian machine the low order byte of an integer appears at the lower address; in a big-endian machine instead the low order byte appears at the higher address. Network order, the order in which numbers are sent on the internet is big-endian.

It is necessary to ensure that the right representation is used on each machine. Functions are used to convert from host to network form before transmission- htons for short integers and htonl for long integers.
The value for servaddr.sin_addr is assigned using the following function

inet_pton(AF_INET, "IP_Address", &servaddr.sin_addr);

The binary value of the dotted decimal IP address is stored in the field when the function returns.

3. Binding of the client socket to a local port

This is optional in the case of client and we usually do not use the bind function on the client side.

4. Connection of client to the server

A server is identified by an IP address and a port number. The connection operation is used on the client side to identify and start the connection to the server.

int connect(int sd, struct sockaddr * addr, int addrlen);

sd – file descriptor of local socket
addr – pointer to protocol address of other socket
addrlen – length in bytes of address structure

The header files to be used are sys/types.h and sys/socket.h

It returns 0 on sucess and -1 in case of failure.

5. Reading from socket

In the case of TCP connection reading from a socket can be done using the read system call

int read(int sd, char * buf, int length);

6. writing to a socket

In the case of TCP connection writing to a socket can be done using the write system call

int write(int sd, char * buf, int length);

7. closing the connection

The connection can be closed using the close system call

int close(int sd);

Steps for TCP Connection for server

1. Creating a listening socket

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

This system call creates a socket and returns a socket descriptor. The domain field used is **AF_INET**. The socket type is **SOCK_STREAM**. The protocol field is 0. If the system is a failure, a -1 is returned. Header files used are sys/types.h and sys/socket.h.

2. Binding to a local port

int bind(int sd, struct sockaddr * addr, int addrlen);

This call is used to specify for a socket the protocol port number where it will wait for messages. A call to bind is optional on the client side, but required on the server side. The first field is the socket descriptor of the local socket. Second is a pointer to the protocol address structure of this socket. The third is the length in bytes of the structure referenced by **addr**. This system call returns an integer. It is 0 for success and -1 for failure. The header files are s**ys/types.h** and s**ys/socket.h.**

3. Listening on the port

The listen function is used on the server in connection oriented communication to prepare a **socke** to accept messages from clients.

int listen(int fd, int qlen);

fd – file descriptor of a socket that has already been bound
qlen – specifies the maximum number of messages that can wait to be processed by the server while the server is busy servicing another request. Usually it is taken as 5. The header files used are sys/types.h and sys/socket.h. This function returns 0 on success and -1 on failure.

4. Accepting a connection from the client

The accept function is used on the server in the case of connection oriented communication to accept a connection request from a client.

int accept(int fd, struct sockaddr * addressp, int * addrlen);

The first field is the descriptor of the server socket that is listening. The second parameter **addressp** points to a socket address structure that will be filled by the address of calling client when the function returns. The third parameter **addrlen** is an integer that will contain the actual length of address structure of the client. It returns an integer that is a descriptor of a new socket called the connection socket. Server sockets send data and read data from this socket. The header files used are sys/types.h and s**ys/socket.h.**

Algorithm

Client

1. Create socket
2. Connect the socket to the server
3. Read the string to be reversed from the standard input and send it to the server
Read the matrices from the standard input and send it to server using socket
4. Read the reversed string from the socket and display it on the standard output
Read product matrix from the socket and display it on the standard output
5. Close the socket

Server

1. Create listening socket
2. bind IP address and port number to the socket
3. listen for incoming requests on the listening socket
4. accept the incoming request
5. connection socket is created when accept returns
6. Read the string using the connection socket from the client
7. Reverse the string
8. Send the string to the client using the connection socket
9. close the connection socket
10. close the listening socket

Client Program

#include<stdio.h>

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>

int main( int argc, char *argv[])
{
struct sockaddr_in server;
int sd ;
char buffer[200];
if((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
perror("Socket failed:");
exit(1);
}
// server socket address structure initialisation
bzero(&server, sizeof(server) );
server.sin_family = AF_INET;
server.sin_port = htons(atoi(argv[2]));
inet_pton(AF_INET, argv[1], &server.sin_addr);
if(connect(sd, (struct sockaddr *)&server, sizeof(server))< 0)
{
perror("Connection failed:");
exit(1);
}
fgets(buffer, sizeof(buffer), stdin);
buffer[strlen(buffer) - 1] = '\0';
write (sd,buffer, sizeof(buffer));
read(sd,buffer, sizeof(buffer));
printf("%s\n", buffer);
close(fd);
}
```

Server Program

```
#include<stdio.h>
```

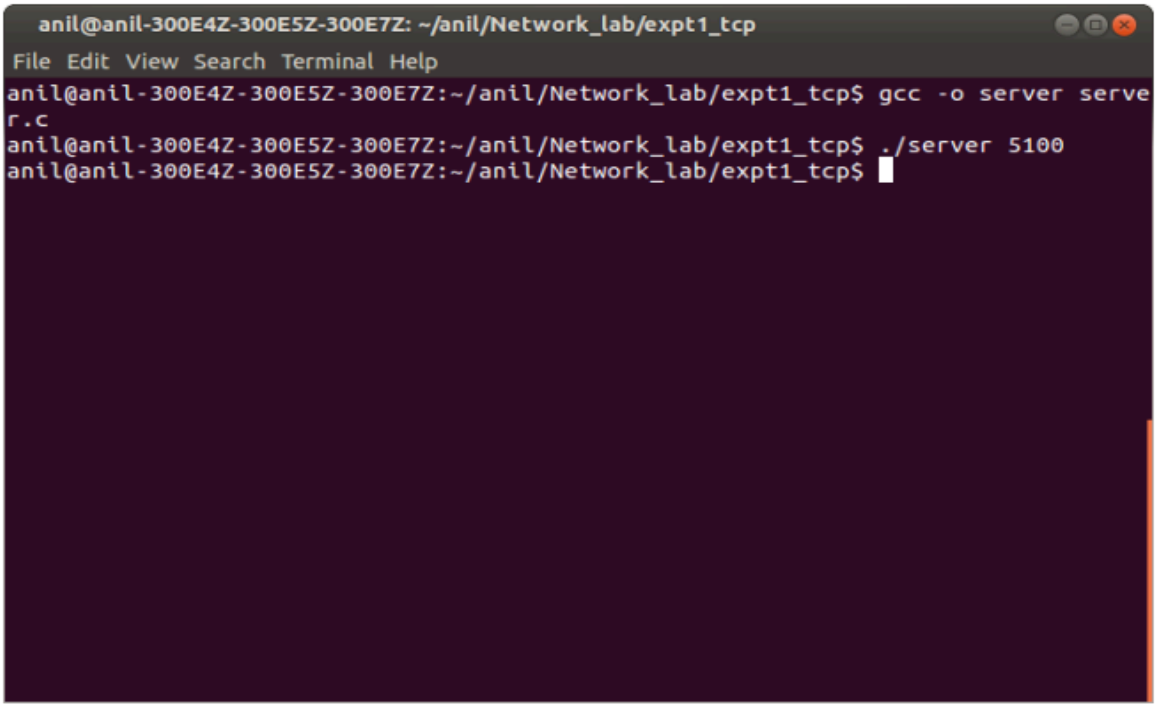
```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<fcntl.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>

int main( int argc, char *argv[])
{
struct sockaddr_in server, cli;
int cli_len;
int sd, n, i, len;
int data, temp;
char buffer[100];
if((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
perror("Socket failed:");
exit(1);
}

// server socket address structure initialisation
bzero(&server, sizeof(server) );
server.sin_family = AF_INET;
server.sin_port = htons(atoi(argv[1]));
server.sin_addr.s_addr = htonl(INADDR_ANY);
if(bind(sd, (struct sockaddr*)&server, sizeof(server)) < 0)
{
perror("bind failed:");
exit(1);
}
listen(sd,5);
if((data = accept(sd , (struct sockaddr *) &cli, &cli_len)) < 0)
{
perror("accept failed:");
exit(1);
}
read(data,buffer, sizeof(buffer));
len = strlen(buffer);
```

```
for( i =0; i<= len/2; i++)
{
temp = buffer[i];
buffer[i] = buffer[len - 1-i];
buffer[len-1-i] = temp;
}
write (data,buffer, sizeof(buffer));
close(data);
close(sd);
}
```

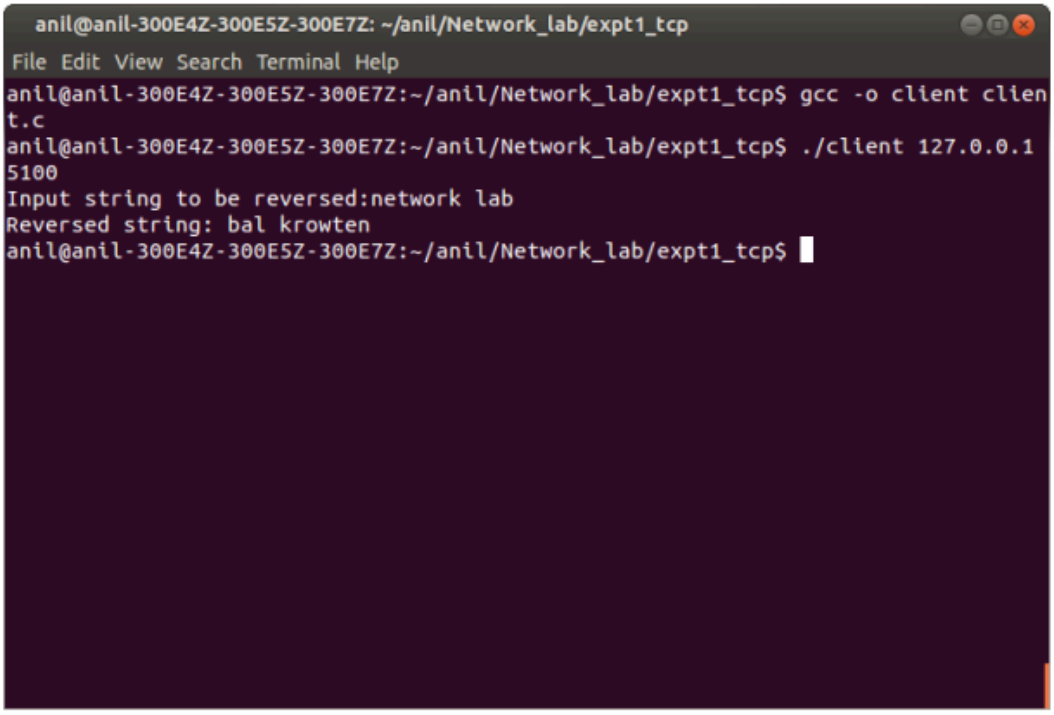
Output

Open with

Server



Client



Experiment 8

Implementation of Client-Server communication using Socket Programming and UDP as transport layer protocol

Aim: Client sends two matrices to the server using udp protocol. The server multiplies the matrices and sends the product to the client, which then displays the product matrix.

Description:

Steps for transfer of data using UDP

1. Creation of UDP socket

The function call for creating a UDP socket is

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

The domain parameter specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <sys/socket.h>. In this program,