WEEK 2: Task

Date: 19/08/2019

2	System Calls: Build simple Client-Server program to transfer file from server to client using system calls open (), read (), write () and close ().

GOAL:

Familiarizing with basic system calls open(), read(), write() and close() to do System Programming using C language. (Use man command to know the definition and usage of each of the system calls)

WEEK 2 IOS Program Execution Steps for

Simple file transfer between Client and Server

1. We have 2 C programs the **client.c** and **server.c** with socket skeleton code already shared with students.
2. Client sends request 'list' to know the files available at the server. **./client list**
3. Server responds to client by displaying all files at its end.
4. Client sends request 'get' to get the contents of the file specified after get argument as third argument. **./client get 'filename'**
5. Server responds to client by displaying the contents of the file by using basic system calls.
    - open() - Open the file requested.
    - read() - Read the contents of the file.
    - write() - Write the contents of the file to the new file created on the server side with same name as given for the third argument on the client request get (**./client get 'filename'**).
    - close() - Closes the file opened.

6. Skeleton socket program is shared with proper explanation. Students are supposed to use the basic system calls to complete the client server file transfer program. Check for error conditions for all system calls used.
    - open()
    - read()
    - write()
    - close()

7. **TODO:** lines are left for the students to complete the solution to the problem.

8. If both the programs client.c and server.c are run on the same system we use local host 127.0.0.1 as IP address.

9. If the client and server programs are run on different systems then we need to change the IP address of the client and server system accordingly.

10. Terminal 1: Run Server first to accept request from Client:

```
$gcc server.c -o server && ./server
```

(Compile server.c and redirect the output to server. Later run/execute server. Both compilation and execution happens in the same command using &&)

11. Terminal 2: Run Client second to send request to Server:

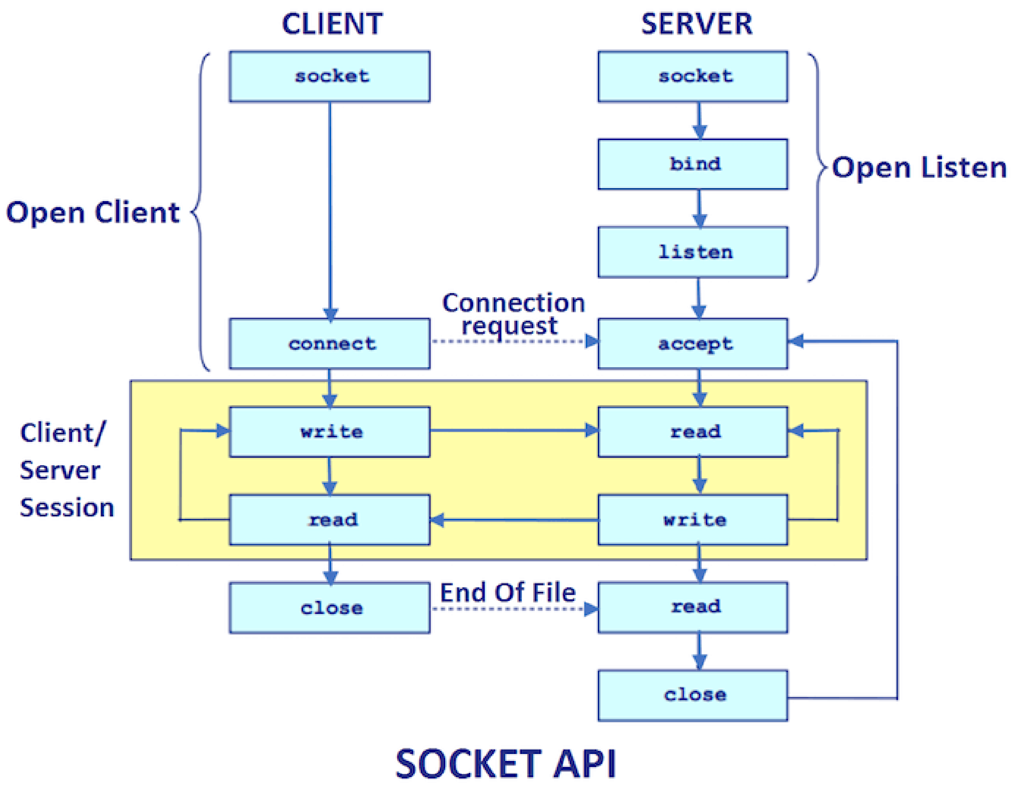
```
$gcc client.c -o client
$./client list
$./client get 'filename'
```

(list' and 'get' are command line arguments that can be passed to the client)

11. If completed with assignment on the same system, try the same with different systems and show the execution.

Sockets for Interprocess Communication:

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a *socket*. A socket is one end of an inter process communication channel. The two processes each establish their own socket.



The steps involved in establishing a socket on the *server* side are as follows:

1. Create a socket with the socket() system call.
2. Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the listen() system call .
4. Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
5. Send and receive data.

Step 1: Creating a socket:

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

Creating a socket is in some ways similar to opening a file. This function creates a file descriptor and returns it from the function call. You later use this file descriptor for reading, writing and using with other socket functions

Parameters:
`family`: AF_INET or PF_INET (These are the IP4 family)
`type`: SOCK_STREAM (for TCP) or SOCK_DGRAM (for UDP)

Step 2: Binding an address and port number

```
int bind(int socket_file_descriptor, const struct sockaddr * LocalAddress, socklen_t
AddressLength);
```

We need to associate an IP address and port number to our application. A client that wants to connect to our server needs both of these details in order to connect to our server. Notice the difference between this function and the connect() function of the client. The connect function specifies a remote address that the client wants to connect to, while here, the server is specifying to the bind function a local IP address of one of its Network Interfaces and a local port number.

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *bind()* is 0 for success and –1 for failure.

Step 3: Listen for incoming connections

Binding is like waiting by a specific phone in your house, and Listening is waiting for it to ring.

```
int listen(int socket_file_descriptor, int backlog);
```

The parameter *socket_file_descriptor* is the socket file descriptor returned by a call to *socket()* function. The return value of *listen()* is 0 for success and –1 for failure.

It is important in determining how many connections the server will connect with. Typical values for backlog are 5 – 10.

Step 4: Accepting a connection.

```
int accept (int socket_file_descriptor, struct sockaddr * ClientAddress, socklen_t
*addrlen);
```

accept() returns a new socket file descriptor for the purpose of reading and writing to the client. The original file descriptor is used usually used for listening for new incoming connections.

The steps involved in establishing a socket on the *client* side are as follows:

1. Create a socket with the socket() system call.
2. Connect the socket to the address of the server using the connect() system call .
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

Step 1: Create a socket : Same as in the server.

Step 2: Binding a socket: This is unnecessary for a client, what bind does is (and will be discussed in detail in the server section) is associate a port number to the application. If you skip this step with a TCP client, a temporary port number is automatically assigned, so it is just better to skip this step with the client.

Step 3: Connecting to a Server:

```
 int connect(int socket_file_descriptor, const struct sockaddr *ServerAddress,
socklen_t AddressLength);
```

Once you have created a socket and have filled in the address structure of the server you want to connect to, the next thing to do is to connect to that server. This is done with the connect function listed above.

Connect performs the three-way handshake with the server and returns when the connection is established or an error occurs.

Once the connection is established you can begin reading and writing to the socket.

Step 4: read/write: These are the same functions you use with files but you can use them with sockets as well.

Writing to a socket:

```
int write(int file_descriptor, const void * buf, size_t message_length);
```

Reading from a socket:

```
int read(int file_descriptor, char *buffer, size_t buffer_length);
```

The value returned is the number of bytes read which may not be buffer_length! It returns –1 for failure. As with write(), read() only transfers data from a buffer in the kernel to your application , you are not directly reading the byte stream from the remote host, but rather TCP is in control and buffers the data for your application.

<u>Shutting down sockets:</u>

After you are finished reading and writing to your socket you most call the close system call on the socket file descriptor just as you do on a normal file descriptor otherwise you waste system resources.

The close() function: `int close(int filedescriptor);`

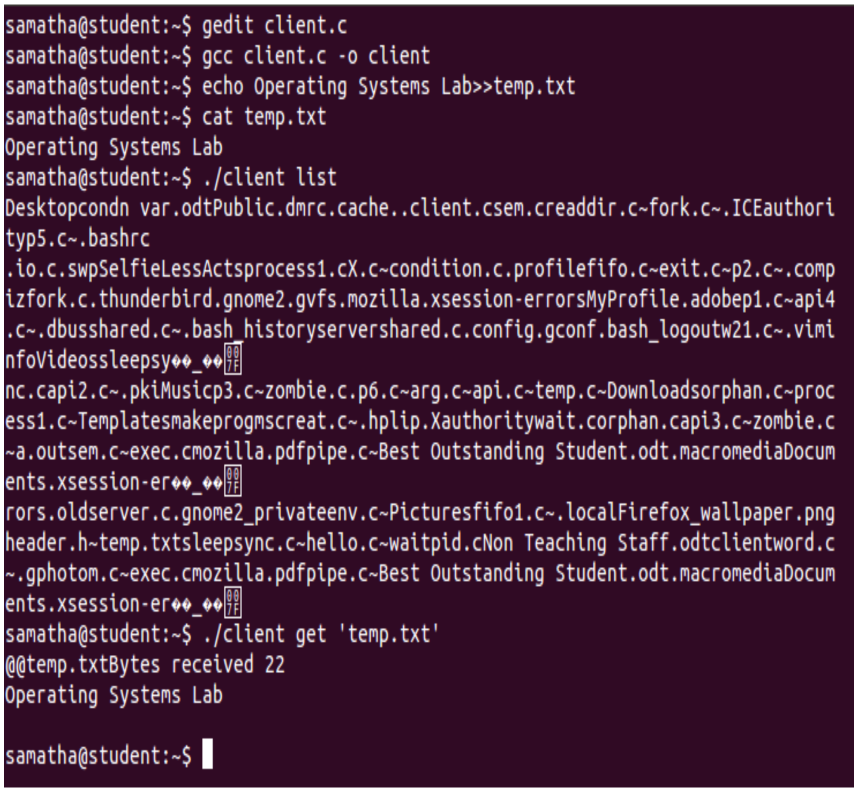
Error Handling:

- **Connect Failed Error Occurred**

  Kill the server and client process by ctrl+C and run once again.
  or
  Change the port numbers and run.

Week 2 IOS Program Execution Snapshots:

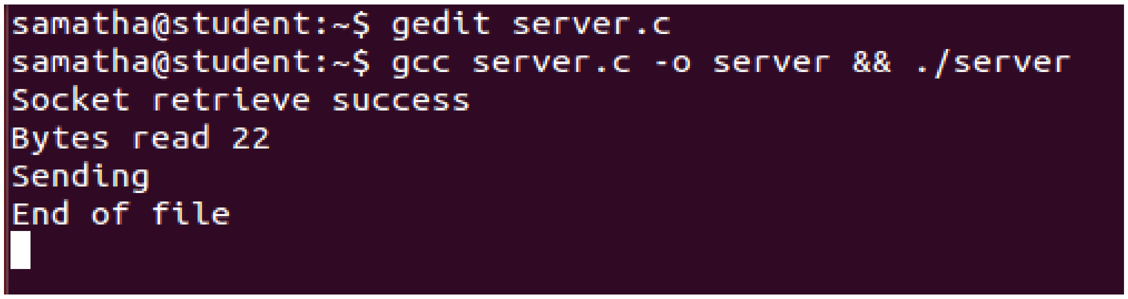
TERMINAL 1: CLIENT PROGRAM



```
samatha@student:~$ gedit client.c
samatha@student:~$ gcc client.c -o client
samatha@student:~$ echo Operating Systems Lab>>temp.txt
samatha@student:~$ cat temp.txt
Operating Systems Lab
samatha@student:~$ ./client list
Desktopcondn var.odtPublic.dmrc.cache..client.csem.creaddir.c~fork.c~.ICEauthori
typ5.c~.bashrc
.io.c.swpSelfieLessActsprocess1.cX.c~condition.c.profilefifo.c~exit.c~p2.c~.comp
izfork.c.thunderbird.gnome2.gvfs.mozilla.xsession-errorsMyProfile.adobep1.c~api4
.c~.dbusshared.c~.bash_historyservershared.c.config.gconf.bash_logoutw21.c~.vimi
nfoVideossleepsy��_��␡
nc.capi2.c~.pkiMusicp3.c~zombie.c.p6.c~arg.c~api.c~temp.c~Downloadsorphan.c~proc
ess1.c~Templatesmakeprogmscreat.c~.hplip.Xauthoritywait.corphan.capi3.c~zombie.c
~a.outsem.c~exec.cmozilla.pdfpipe.c~Best Outstanding Student.odt.macromediaDocum
ents.xsession-er��_��␡
rors.oldserver.c.gnome2_privateenv.c~Picturesfifo1.c~.localFirefox_wallpaper.png
header.h~temp.txtsleepsync.c~hello.c~waitpid.cNon Teaching Staff.odtclientword.c
~.gphotom.c~exec.cmozilla.pdfpipe.c~Best Outstanding Student.odt.macromediaDocum
ents.xsession-er��_��␡
samatha@student:~$ ./client get 'temp.txt'
@@temp.txtBytes received 22
Operating Systems Lab

samatha@student:~$ █
```

TERMINAL 2: SERVER PROGRAM



```
samatha@student:~$ gedit server.c
samatha@student:~$ gcc server.c -o server && ./server
Socket retrieve success
Bytes read 22
Sending
End of file
```