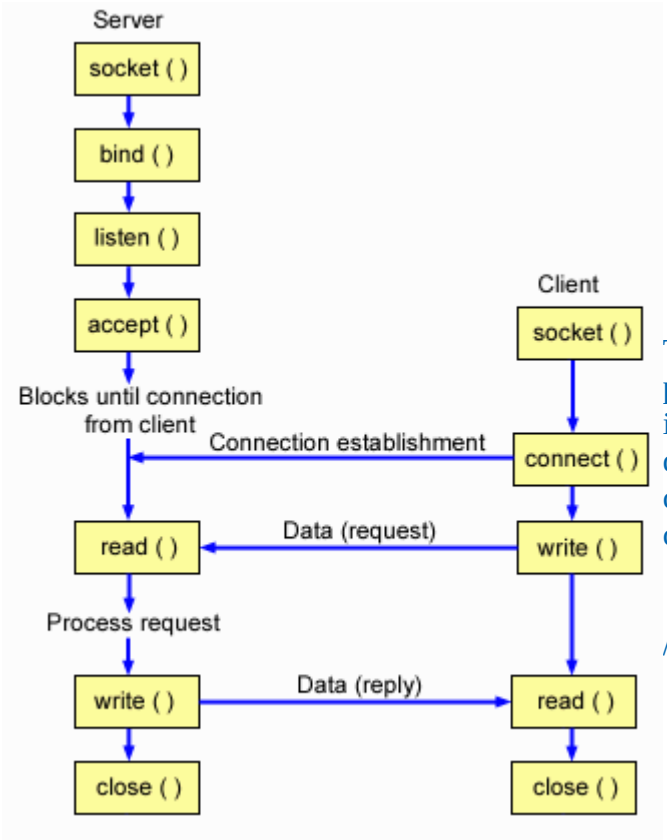
Note:

- If you get this while executing the compiled code-
  error in binding tcp socket: Address already in use
  Error while binding udp socket
  then close that instance of terminal open a new one and compile again

- Note use port numbers which are not reserved
  port numbers 0 to 1023 are reserved.
- The port numbers should be same in both the client and the server
- The address specified in client_address.sin_addr.s_addr = INADDR_ANY;
  should also be same for both the client and the server in order for the client to discover the server socket.
- In place of INADDR_ANY you can also provide the IP as
  inet_addr("127.0.0.1") or
  inet_addr("172.##.18.XXX"); (in order to connect to a specified IP)
  but again do this in both the client and the server.

QUESTION_1
Write a client-server program that provides text and voice chat facility using datagram socket. Your application allows a user on one machine to talk to a user on another machine. Your application should provide non blocking chat facility to the users. This means, the user can send its message at any time without waiting for a reply from the other side.
(Hint: Use select() system call).
DID NOT PROVIDE VOICE CHAT FACILITY
Ans
Header files included and their requirements

- #include<stdio.h>//Defines core input and output functions
- #include<stdlib.h>//Defines numeric conversion functions, pseudo-random numbers generation functions, memory allocation, process control functions
- #include<sys/types.h>//The <sys/types.h> header contains a number of basic derived types that should be used whenever appropriate, including select() functionality
- #include<sys/socket.h>//The <sys/socket.h> header shall define the sockaddr structure
- #include<netinet/in.h>//The <netinet/in.h> header shall define the sockaddr_in structure
- #include<unistd.h>//for read write and close operations, also select() implementation
- #include<string.h>//Defines string-handling functions



This is a basic roadmap , but just before performing the read syscall on either side we introduce select() syscall to select the descriptor for either the stdin or the connection_socket , which ever is the one we can read from.

//socket() is for socket creation

//bind() is to bind the socket with an address

//listen() listens for available connections

//accept() accepts the connection

//connect() connects the client socket to the server's address

(read from either the connection_socket from the server or the standard input stream using select())

Submit two files: dns_server.c and dns_client.c

select() allows a program to monitor multiple file (or socket) descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation (e.g., input possible). A file descriptor is considered ready if it is possible to perform a corresponding I/O operation without blocking.

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                  fd_set *exceptfds, struct timeval *timeout);
```

The principal arguments of **select**() are three "sets" of file descriptors for ready to read ones , ready to write ones and error occurance ones, (declared using the type *fd_set*), which allow the caller to wait for three classes of events on the specified set of file descriptors. Each of the *fd_set* arguments may be specified as NULL if no file descriptors are to be watched for the corresponding class of events.

Note: Upon return of the select() call, each of the file descriptor sets is modified in place to indicate which file descriptors are currently "ready". **Thus, if using select() within a loop, the sets *must be reinitialized* before each call.**

We use select(5, descp_set ,0,0,0) as our syscall because we need to check out of which of the descriptors in the set descp_set which is the one ready to be read.(5 specifies the max size of the descp_set).

- We use these two operations in the beginning of **every iteration.**

FD_ZERO()
This macro clears (removes all file descriptors from) set. It should be employed as the first step in initializing a file descriptor set.
FD_SET()
This macro adds the file descriptor fd to set. Adding a file descriptor that is already present in the set is a no-op, and does not produce an error.

- After that we use the select syscall, and using....

FD_ISSET()
select() modifies the contents of the sets according to the rules described below. After calling **select**(), the **FD_ISSET**() macro can be used to test if a file descriptor is still present in a set. **FD_ISSET**() returns nonzero if the file descriptor *fd* is present in *set*, and zero if it is not.

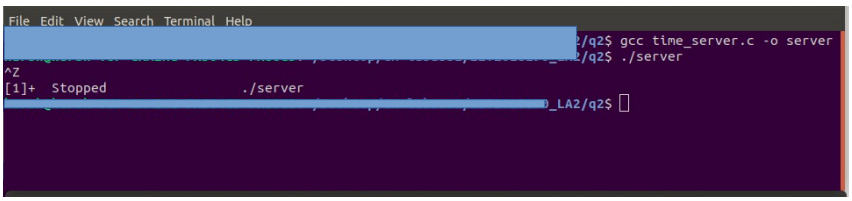
... we check if the bit for a specified descriptor is set or not , if yes we read from that descriptor.

In either case we either read from the terminal through fgets with the third argument as stdin or read from the connection socket using recv syscall , we could also use read() , just remove the last value of recv which is a flag and is set to 0.
In case a “bye” message is sent we terminate the connection by closing (using close syscall in unistd.h) the sockets.

Note first execute the server and then the client,(though compilation can be performed in any order).

Here is a glimpse of execution.

QUESTION_2
Write a TCP client-server system to allow client programs to get the system date and time from the server. When a client connects to the server, the server gets the local time on the machine and sends it to the client. The client displays the date and time on the screen, and terminates. The server should be an iterative server.
Submit two C files: time_server.c and time_client.c



Note-

- **Note first execute the server and then the client**
- **Every time you execute the client, the server sends the date and time as reply**
- **The server has to be stopped by pressing Ctrl+Z**
- **You can optimize this by executing the client too in endless loop (as in the server) and using bye to quit both the client and the server.**

- **The requirement was only the server should be iterative, no condition on the client.**
- **You can even have different executables for different clients, with essentially the same code for requesting the date and time info**

Similar roadmap as previous question , here also we implement the tcp client server connection.
For the server,

1. socket creation
2. binding socket and address
3. listening for connections
4. accepting a connection
5. find current local time
6. close the connection

   Steps 4,5,6 happen in infinite repetition

In this server side code we use time.h additionally as it contains definitions of functions to get and manipulate date and time information. It describes three time related **datatypes**.

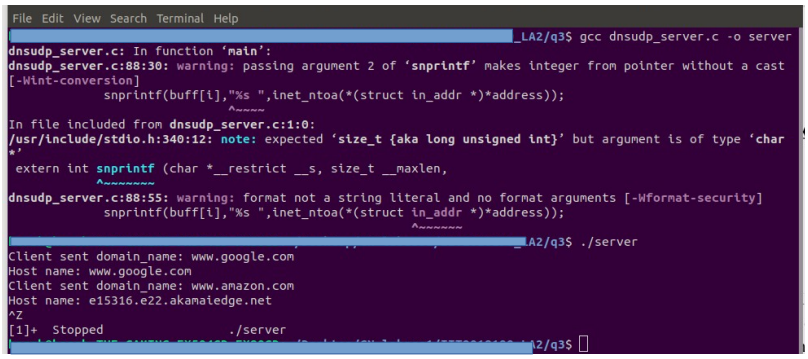
1. **clock_t**: clock_t represents the date as integer which is a part of the calendar time.
2. **time_t**: time_t represents the clock time as integer which is a part of the calendar time.
3. **struct tm**: struct tm holds the date and time which contains:

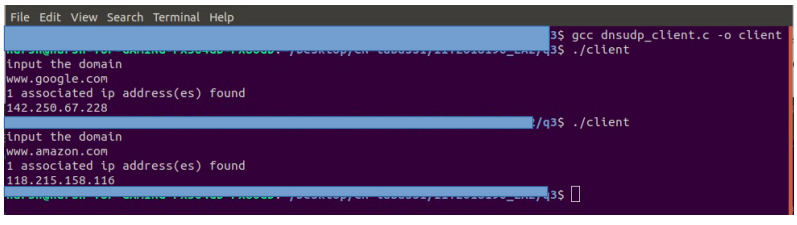
We used time() function which returns the calendar-time equivalent using data-type time_t.
We then use the function ctime() which returns the date and time in the format day month hours:minutes:seconds year
Eg: Sat Jul 27 11:26:03 2019
The time is printed based on the pointer returned by Calendar Time.

For the client,
after establishing connection, we receive the date and time in a string format and print it.

QUESTION_3

Write a simple UDP iterative server and client to convert a given DNS name (for example, www.google.com) into its IP address(es). The client will read the DNS name as a string from the user and send it to the server. The server will convert it to one or more IP addresses and return it back to the client. The client will then print ALL the addresses returned, and exit. For basic UDP socket communication, see the sample program given. To get the IP address corresponding to a DNS name, use the function gethostbyname(). Look up the description of the function from the man page and the tutorial on the webpage.
Submit two files: dns_server.c and dns_client.c





Note-

- **Note first execute the server and then the client**
- **Every time you execute the client, the server sends the date and time as reply**
- **The server has to be stopped by pressing Ctrl+Z**
- **You can optimize this by executing the client too in endless loop (as in the server) and using bye to quit both the client and the server.**
- **The requirement was only the server should be iterative, no condition on the client.**
- **You can even have different executables for different clients, with essentially the same code for requesting the HostName**

To begin with , here apart from the usual libraries required for the socket programming in C, we also use netdb.h which provides the hostent structure and also the function gethostbyname().
The structure hostent contains the necessary data associated with the allotment of an ip address, viz., domain name, ip address,host name, number of aliases , number of addresses etc.
Here we specificallly use the list of addresses and the hostname apart from the domain name and ip address.

For the server,

- socket creation(this time using SOCK_DGRAM) for udp
- binding socket and address
- Here instead of using the listen and accept operations , we use recvfrom instead of recv and sendto for send , recvfrom also takes into argument a structure for client address and its length and stores the information in these variables upon finding a client , this data is then used in sendto operations where data is to be sent to the client.
- Get the domain name from the client
- check whether the host information is available for the given domain
- print the corresponding host name
- traverse the list of address and convert them to string format using inet_ntoa() function which is part of arpa/inet.h library and then store them in an array of strings called buff.
- send the number of addresses and also the addresses to the client to display
- iterate infinitely

For the client,

- socket creation and binding
- input domain name from stdin
- sendto() the server
- receive the number and the ip_addresses
- print them out

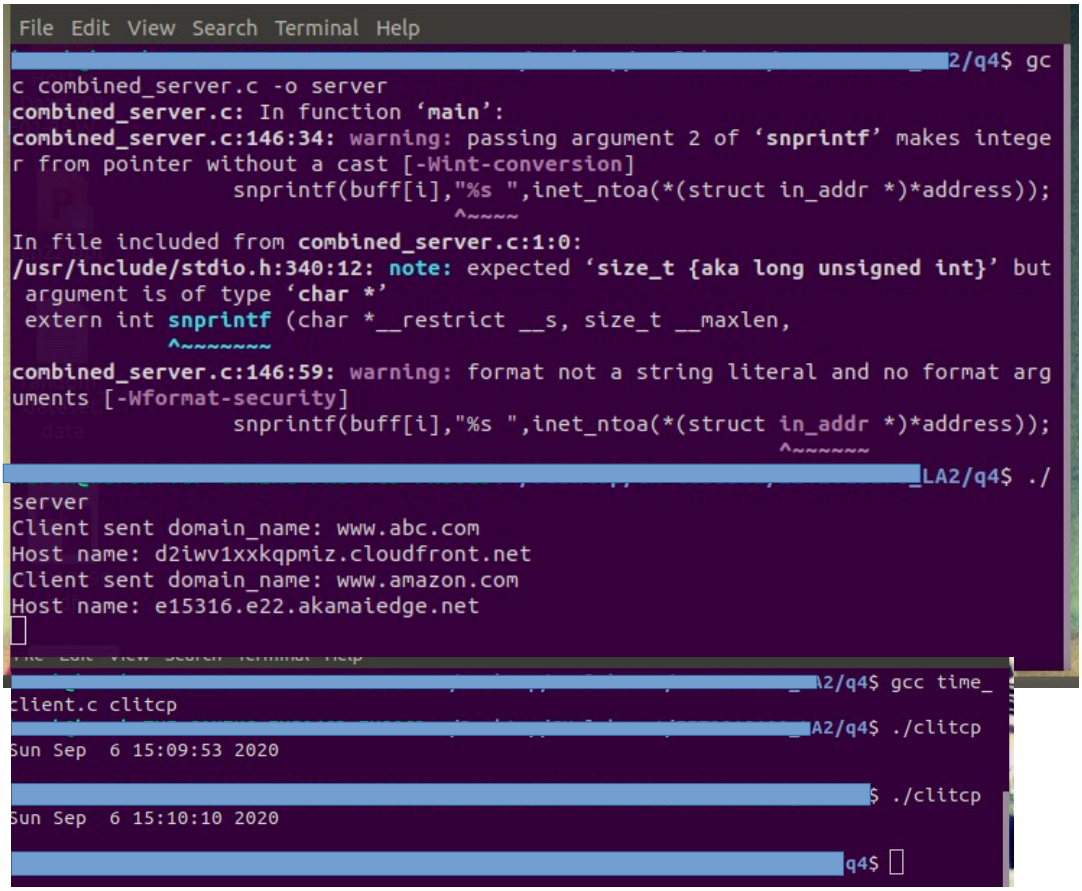
QUESTION_4
Now suppose that the same server will act both as the time server in Problem 1 and the DNS server in Problem 2. Thus, some clients will request over the UDP socket for name-to-IP conversion, and some will connect over a TCP socket for the time. Thus, the server now needs to open both a TCP socket and a UDP socket, and accept request from any one (using the accept() + read()/send() call for TCP, and recvfrom() call for UDP), whichever comes first. Use the select() call to make the server wait for any one of the two connections, and handle whichever comes first. All handlings are iterative.
Submit one C file: combined_server.c

Note-

- **Note first execute the server and then the client**

- **Every time you execute the client, the server sends the date and time as reply**
- **The server has to be stopped by pressing Ctrl+Z**

I used the infromation from all three of the above questions and using select call, assigned a file descriptor to both the connections in the server code and responded accordingly.



aaa