UNIT-IV

1. **Differentiate pipe and named pipe concepts in IPC process.**
   Ans:

PIPE	NAMED PIPE
1. A named type has a specific name which can be given to it by the user. Named pipe if referred through this name only by the reader and writer. All instances of a named pipe share the same pipe name.	1. Unnamed pipes is not given a name. It is accessible through two file descriptors that are created through the function pipe(fd[2]), where fd[1] signifies the write file descriptor, and fd[0] describes the read file descriptor.
2. A named pipe can be used for communication between two unnamed processes as well. Processes of different ancestry can share data through a named pipe.	2. An unnamed pipe is only used for communication between a child and its parent process
3. Named pipe exists in the file system. After input-output has been performed by the sharing processes, the pipe still exists in the file system independently of the process, and can be used for communication between some other processes.	3. An unnamed pipe vanishes as soon as it is closed, or one of the process (parent or child) completes execution.
4. Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network, as in case of a distributed system.	4. Unnamed pipes are always local; they cannot be used for communication over a network.
5. A Named pipe can have multiple process communicating through it, like multiple clients connected to one server.	5. An unnamed pipe is a one-way pipe that typically transfers data between a parent process and a child process.
6. A named pipe provides us with greater functionalities.	6. An unnamed type is simple to use and incurs less overheads

2. **Illustrate pipes? Explain their limitations. Explain how named pipes are replaced to overcome the drawback of pipe in IPC with an Examples.**
   Ans:
   - ✓ Pipe is one-way communication only i.e., we can use a pipe such that One process writes to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a “virtual file”.

- ✓ The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this "virtual file" or pipe and another related process can read from it.
- ✓ If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- ✓ The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.

3. **Create a FIFO to build the communication channel between two different processes.**

   Ans:

   Creation of FIFO:

   We can create a FIFO from the command line and within a program.
   To create from command line we can use either mknod or mkfifo commands.

   $ mknod filename p

   $ mkfifo filename

   (Note: The mknod command is available only n older versions, you can make use of mkfifo in
   new versions.)

   - ✓ To create FIFO within the program we can use two system calls. They are,

     #include<sys/types.h>
     #include<sys/stat.h>
     int mkfifo(const char *filename, mode_t mode);
     int mknod(const char *filename,
     mode_t mode|S_IFIFO,(dev_t) 0);

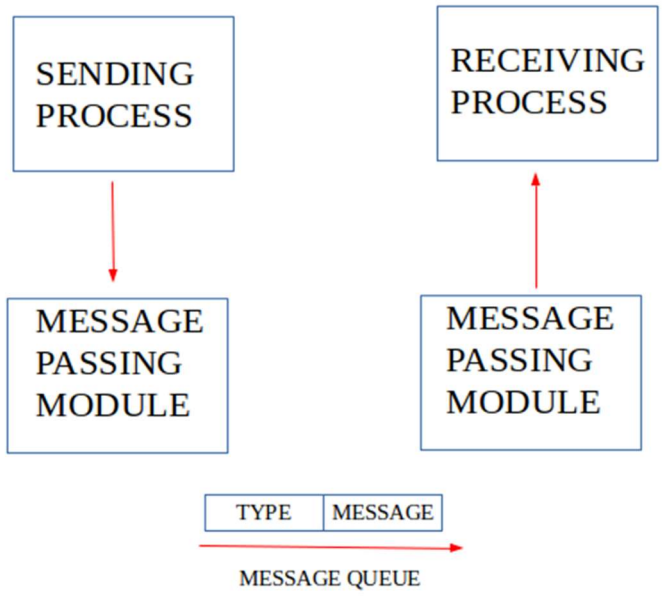
   If we want to use the mknod function we have to use ORing process of fileaccess mode with S_IFIFO and the dev_t value of 0. Instead of using this we can use the simple mkfifo function.

4. **Describe message queue API with syntax and example?**

   Ans:

   - ✓ A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**.
   - ✓ New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue.
   - ✓ Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

- ✓ All processes can exchange information through access to a common system message queue.
- ✓ The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process.
- ✓ Each message is given an identification or type so that processes can select the appropriate message.
- ✓ Process must share a common key in order to gain access to the queue in the first place.



System calls used for message queues:

- **ftok()**: is use to generate a unique key.
- **msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
- **msgsnd()**: Data is placed on to a message queue by calling msgsnd().
- **msgrcv()**: messages are retrieved from a queue.
- **msgctl()**: It performs various operations on a queue. Generally, it is use to destroy message queue.

5. Illustrate about V IPC semaphore mechanism with example.

Ans:

- ✓ System V IPC Mechanisms Linux supports three types of interprocess communication mechanisms which first appeared in Unix System V (1983).
- ✓ These are message queues, semaphores, and shared memory. These System V IPC mechanisms all share common authentication methods.
- ✓ Processes may access these resources only by passing a unique reference identifier to the kernel via system calls.
- ✓ Access to these System V IPC objects is checked using access permissions, much like accesses to files are checked.

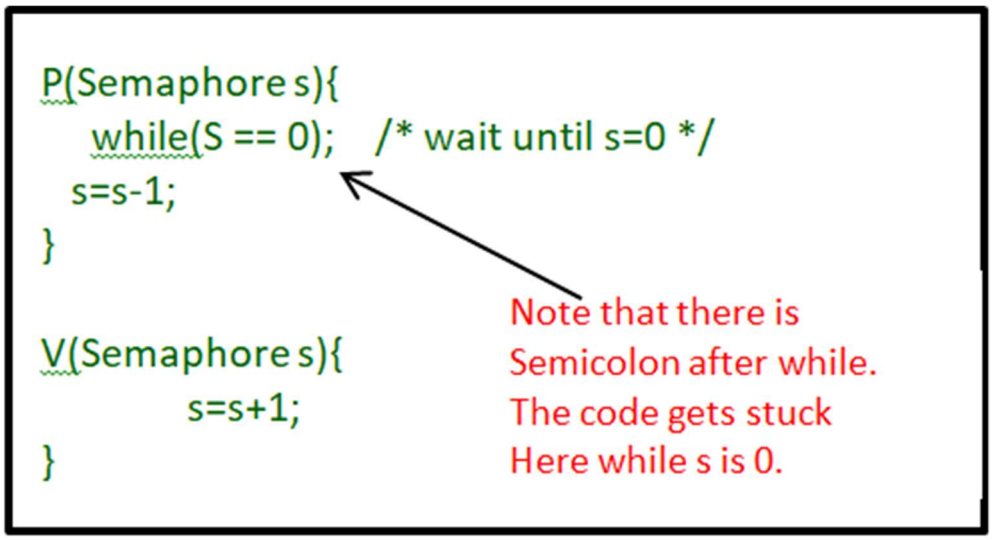
- ✓ The access rights to the System V IPC object is set by the creator of the object via system calls.
- ✓ The object's reference identifier is used by each mechanism as an index into a table of resources. It is not a straightforward index but requires some manipulation to generate the index.
- ✓ All Linux data structures representing System V IPC objects in the system include an ipc_perm Structure which contains the owner and creator processes user and group identifiers.
- ✓ The access mode for this object (owner, group and other) and the IPC object's key. The key is used as a way of locating the System V IPC object's reference identifier.
- ✓ Two sets of key are supported: public and private. If the key is public then any process in the system, subject to rights checking, can find the reference identifier for the System V IPC object.
- ✓ System V IPC objects can never be referenced with a key, only by their reference identifier

6. **Describe about synchronization and how synchronization is achieved with Semaphores?**

   Ans:

   - ✓ Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.
   - ✓ Semaphore is simply a variable that is non-negative and shared between threads.
   - ✓ This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment. Semaphores are of two types:

   1. **Binary Semaphore –**
      This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
   2. **Counting Semaphore –**
      Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.



Some point regarding P and V operation

1. P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in-between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization. See the below image. The critical section of Process P is in between P and V operation.

Limitations :

1. One of the biggest limitations of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and to signal the semaphore.

Problem in this implementation of semaphore:

- ✓ The main problem with semaphores is that they require busy waiting, If a process is in the critical section, then other processes trying to enter critical section will be waiting until the critical section is not occupied by any process.
- ✓ Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle.
- ✓ There is also a chance of “spinlock” as the processes keep on spins while waiting for the lock.

PART-C

1. **Illustrate to redirect the standard input (stdin) and the standard output (stdout) of a process, so that scanf () reads from the pipe and printf () writes into the pipe?**

   Ans:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
main()
{
int fd[2]; int n=0, i;
pipe(fd);
if (fork() == 0) { /* create Child process */
close(1) ;
dup(fd[1]) ;
close(fd[0]);
/* try not read from the pipe in this example. So close fd[0]. */
for(i=0; i < 10; i++) {
printf("%d\n",n);
/* Now that stdout has been redirected, printf automatically writes into the
pipe. */
n++; }
} else {
/* Parent process */
close(0) ;
dup(fd[0]) ;
/* Redirect the stdin of this process to the pipe*/
close(fd[1]);
/* will not write into the pipe.So we close fd[1]. */
for (i=0; i < 10; i++) {
scanf("%d",&n);
/* Now that stdin has been redirected, scanf automatically reads from the
pipe. */
printf("n = %d\n",n);
/* try stdout of this has not changed . So this will be shown in the terminal.
*/ sleep(1);
}
}
}
```

2. **Write a c program to send and receive message using pipes. Implement two-way communication using pipes.**
   Ans:

3. **Demonstrate the priority message queues with example using Message Queue API.**
   Ans:

4. **Illustrate to displays no of messages in queue, last message sends, last message read time in a given message queue.**
   Ans:

5. **Write a C program to create a message queue with read and write permissions to write 3 messages to it with different priority numbers.**
   Ans:

6. **Write a C program that receives 3 messages from the sender using message queues system calls and displays messages to output stream based on priority.**
   Ans:

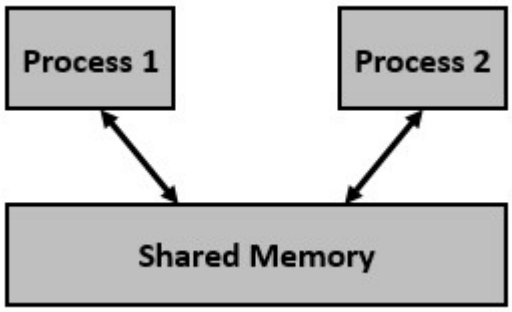
7. **Write thread synchronization with semaphores with example.**
   Ans:

8. **Illustrate about Semaphores with examples.**
   Ans:

UNIT-5
Part- B

1. **Demonstrate shared-memory segment to overcome the Drawback of message queue with example.**
   Ans:

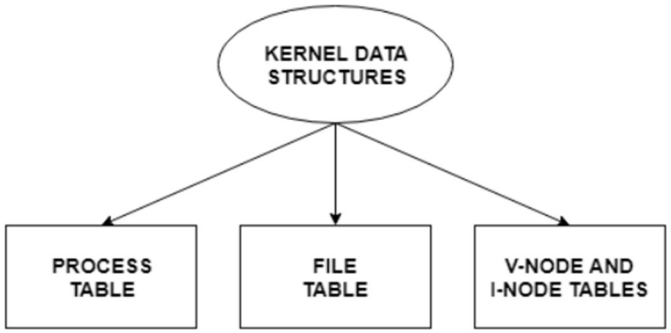
2. **Discuss the structure of a shared memory and kernel data structure with a neat diagram?**
   Ans:
   Shared Memory:
   - ✓ Shared memory is a memory shared between two or more processes.
   - ✓ To reiterate, each process has its own address space, if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter process communication) techniques.
   - ✓ As we are already aware, communication can be between related or unrelated processes.
   - ✓ Usually, inter-related process communication is performed using Pipes or Named Pipes. Unrelated processes (say one process running in one terminal and another process in another terminal) communication can be performed using Named Pipes or through popular IPC techniques of Shared Memory and Message Queues.
   - ✓ We have seen the IPC techniques of Pipes and Named pipes and now it is time to know the remaining IPC techniques viz., Shared Memory, Message Queues, Semaphores, Signals, and Memory Mapping.



   SharedMemory:
   https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_shared_memory.htm

   Kernel Data Structure:
   - ✓ The kernel data structures are very important as they store data about the current state of the system.

- ✓ For example, if a new process is created in the system, a kernel data structure is created that contains the details about the process.
- ✓ Most of the kernel data structures are only accessible by the kernel and its subsystems.
- ✓ They may contain data as well as pointers to other data structures.



Details about these are as follows:

1. **Process Table**
   - ✓ The process table stores information about all the processes running in the system. These include the storage information, execution status, file information etc.
   - ✓ When a process forks a child, its entry in the process table is duplicated including the file information and file pointers. So the parent and the child process share a file.
2. **File Table**
   - ✓ The file table contains entries about all the files in the system. If two or more processes use the same file, then they contain the same file information and the file descriptor number.
   - ✓ Each file table entry contains information about the file such as file status (file read or file write), file offset etc. The file offset specifies the position for next read or write into the file.
   - ✓ The file table also contains v-node and i-node pointers which point to the virtual node and index node respectively. These nodes contain information on how to read a file.
3. **V-Node and I-Node Tables**
   - ✓ Both the v-node and i-node are references to the storage system of the file and the storage mechanisms. They connect the hardware to the software.
   - ✓ The v-node is an abstract concept that defines the method to access file data without worrying about the actual structure of the system.
   - ✓ The i-node specifies file access information like file storage device, read/write procedures etc.

3. **Illustrate TCP socket connection establishment with a neat diagram?**
   Ans:
   TCP is a connection-oriented protocol and every connection-oriented protocol needs to establish connection in order to reserve resources at both the communicating ends.
   Connection Establishment –

1. Sender starts the process with following:
   - **Sequence number (Seq=521):** contains the random initial sequence number which generated at sender side.
   - **Syn flag (Syn=1):** request receiver to synchronize its sequence number with the above provided sequence number.
   - **Maximum segment size (MSS=1460 B):** sender tells its maximum segment size, so that receiver sends datagram which won't require any fragmentation. MSS field is present inside **Option** field in TCP header.
   - **Window size (window=14600 B):** sender talks about his buffer capacity in which he has to store messages from receiver.

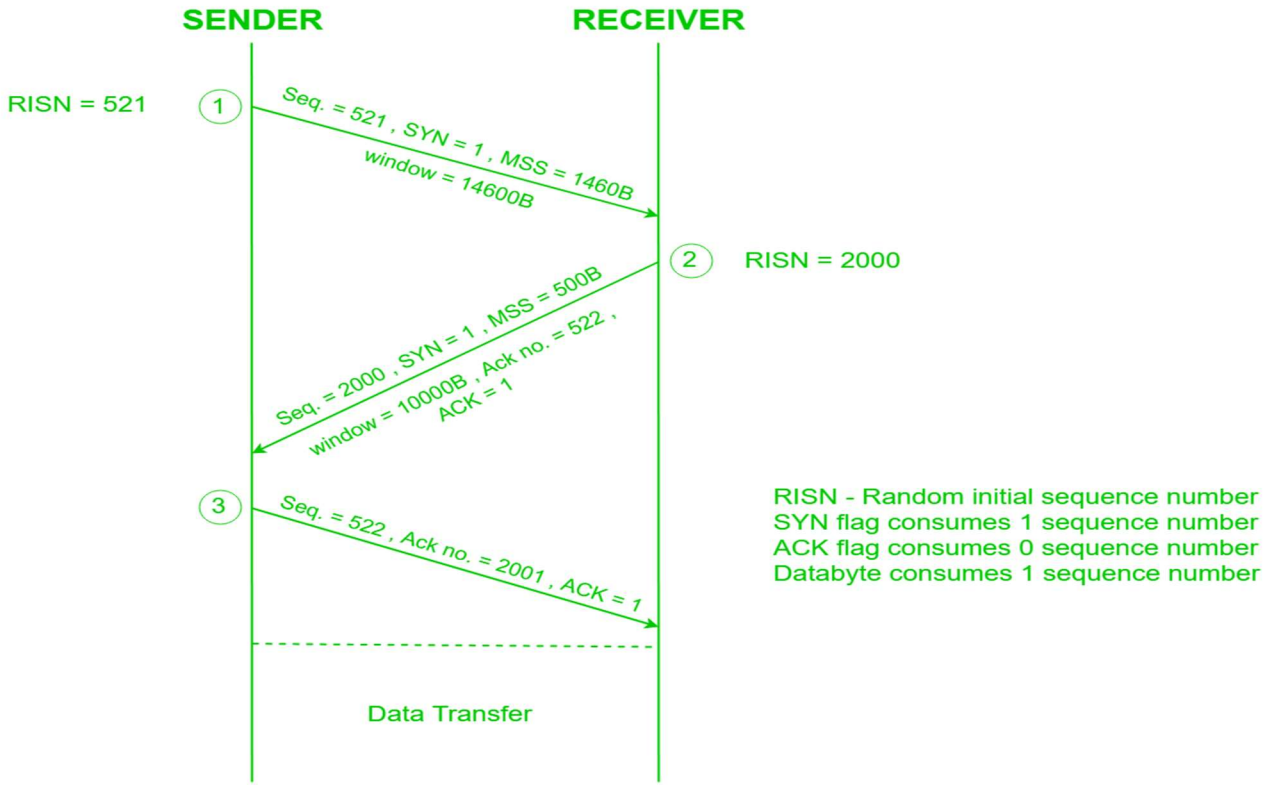
2. TCP is a full duplex protocol so both sender and receiver require a window for receiving messages from one another.
   - **Sequence number (Seq=2000):** contains the random initial sequence number which generated at receiver side.
   - **Syn flag (Syn=1):** request sender to synchronize its sequence number with the above provided sequence number.
   - **Maximum segment size (MSS=500 B):** sender tells its maximum segment size, so that receiver sends datagram which won't require any fragmentation. MSS field is present inside **Option** field in TCP header.
     Since $MSS_{receiver} < MSS_{sender}$, both parties agree for minimum MSS i.e., 500 B to avoid fragmentation of packets at both ends.
   - Therefore, receiver can send maximum of 14600/500 = 29 packets.

   This is the receiver's sending window size.
   - **Window size (window=10000 B):** receiver tells about his buffer capacity in which he has to store messages from sender.
   - Therefore, sender can send a maximum of 10000/500 = 20 packets.

   This is the sender's sending window size.
   - **Acknowledgement Number (Ack no.=522):** Since sequence number 521 is received by receiver so, it makes a request of next sequence number with Ack no.=522 which is the next packet expected by receiver since Syn flag consumes 1 sequence no.

- **ACK flag (ACk=1):** tells that acknowledgement number field contains the next sequence expected by receiver.

3. Sender makes the final reply for connection establishment in following way:
   - **Sequence number (Seq=522):** since sequence number = 521 in 1st step and SYN flag consumes one sequence number hence, next sequence number will be 522.
   - **Acknowledgement Number (Ack no.=2001):** since sender is acknowledging SYN=1 packet from the receiver with sequence number 2000 so, the next sequence number expected is 2001.
   - **ACK flag (ACK=1):** tells that acknowledgement number field contains the next sequence expected by sender.



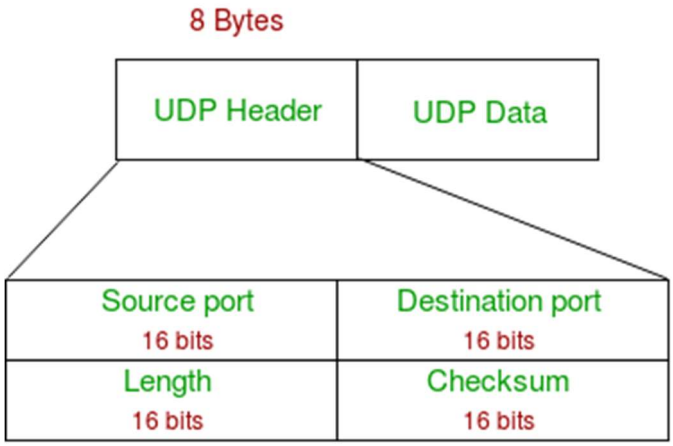
4. **Illustrate UDP data transfer with a neat diagram.**

   Ans:

   - ✓ **User Datagram Protocol (UDP)** is a Transport Layer protocol. UDP is a part of Internet Protocol suite, referred as UDP/IP suite. Unlike TCP, it is **unreliable and connectionless protocol.** So, there is no need to establish connection prior to data transfer.
   - ✓ Though Transmission Control Protocol (TCP) is the dominant transport layer protocol used with most of Internet services; provides assured delivery, reliability and much more but all these services cost us with additional overhead and latency. Here, UDP comes into picture.

- ✓ For the realtime services like computer gaming, voice or video communication, live conferences; we need UDP. Since high performance is needed, UDP permits packets to be dropped instead of processing delayed packets. There is no error checking in UDP, so it also save bandwidth. User Datagram Protocol (UDP) is more efficient in terms of both latency and bandwidth.

  UDP Header –
- ✓ UDP header is **8-bytes** fixed and simple header, while for TCP it may vary from 20 bytes to 60 bytes. First 8 Bytes contains all necessary header information and remaining part consist of data.
- ✓ UDP port number fields are each 16 bits long, therefore range for port numbers defined from 0 to 65535; port number 0 is reserved. Port numbers help to distinguish different user requests or process.



1. **Source Port:** Source Port is 2 Byte long field used to identify port number of sources.
2. **Destination Port:** It is 2 Byte long field, used to identify the port of destined packet.
3. **Length:** Length is the length of UDP including header and the data. It is 16-bits field.
4. **Checksum:** Checksum is 2 Bytes long field. It is the 16-bit one's complement of the one's complement sum of the UDP header, pseudo header of information from the IP header and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.

Notes – Unlike TCP, Checksum calculation is not mandatory in UDP. No Error control or flow control is provided by UDP. Hence UDP depends on IP and ICMP for error reporting.

5. **Illustrate about bind (), read(), write() functions in Linux**

   Ans:

1. **bind** command is Bash shell built-in command. It is used to set Read line key bindings and variables. The key bindings are the keyboard actions that are bound to a function. So it can be used to change how the bash will react to keys or combinations of keys, being pressed on the keyboard.
   Syntax:
   bind [-lpsvPSVX] [-m keymap] [-q name] [-f filename] [-u name] [-r keyseq]
   [-x keyseq:shell-command] [keyseq:readline-function or readline-command]
2. **write()** writes up to *count* bytes to the file referenced by the file descriptor *fd* from the buffer starting at *buf*. POSIX requires that a **read**() which can be proved to occur after a **write**() has returned returns the new data. Note that not all file systems are POSIX conforming.
   write()- writes to a file descriptor
   #include <unistd.h>

   ssize_t write(int *fd***, const void ****buf***, size_t** *count***);**

3. **read()** attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. If count is zero, read() returns zero and has no other results. If count is greater than SSIZE_MAX, the result is unspecified.
   read- read from a file descriptor
   #include <unistd.h>

   ssize_t read(int fd, void *buf, size_t count);

6. **Demonstrate about sendto() and recvfrom() functions in Linux.**
   Ans:
   The *sendto* Function
   - ✓ The *sendto* function is used to send data over UNCONNECTED datagram sockets. Its signature is as follows −
   - ✓ int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
   - ✓ This call returns the number of bytes sent, otherwise it returns -1 on error.

   Parameters
   - **sockfd** − It is a socket descriptor returned by the socket function.
   - **msg** − It is a pointer to the data you want to send.
   - **len** − It is the length of the data you want to send (in bytes).

- **flags** − It is set to 0.
- **to** − It is a pointer to struct sockaddr for the host where data has to be sent.
- **tolen** − It is set it to sizeof(struct sockaddr).

The *recvfrom* Function

- ✓ The *recvfrom* function is used to receive data from UNCONNECTED datagram sockets.
- ✓ int recvfrom(int sockfd, void *buf, int len, unsigned int flags struct sockaddr *from, int *fromlen);
- ✓ This call returns the number of bytes read into the buffer, otherwise it returns -1 on error.

Parameters

- **sockfd** − It is a socket descriptor returned by the socket function.
- **buf** − It is the buffer to read the information into.
- **len** − It is the maximum length of the buffer.
- **flags** − It is set to 0.
- **from** − It is a pointer to struct sockaddr for the host where data has to be read.
- **fromlen** − It is set it to sizeof(struct sockaddr).

7. **Illustrate about TCP NODELAY syntax with a small program.**

Ans:

- ✓ The TCPNODELAY option specifies whether the server disables the delay of sending successive small packets on the network.
- ✓ Change the value from the default of YES only under one of these conditions:
- You are directed to change the option by your service representative.
- You fully understand the effects of the TCP Nagle algorithm on network transmissions. Setting the option to NO enables the Nagle algorithm, which delays sending small successive packets.

Syntax

```
>>-TCPNodelay--+-Yes-+---------------------------------------><
               '-No--'
```

Parameters

1. Yes

Specifies that the server allows successive small packets to be sent immediately over the network. Setting this option to YES might improve performance in some high-speed networks. The default is YES.

2. **No**
   Specifies that the server does not allow successive small packets to be sent immediately over the network.

8. **Demonstrate all byte ordering and manipulation functions with Examples?**

Ans:

- ✓ There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string.
- ✓ We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings.
- ✓ The functions beginning with str (for string), defined by including the header, deal with null-terminated C character strings.
- ✓ The first group of functions, whose names begin with b (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions.
- ✓ The second group of functions, whose names begin with mem (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library.
- ✓ We first show the Berkeley-derived functions, although the only one we use in this text is bzero. (We use it because it has only two arguments and is easier to remember than the three-argument memset function, as explained on p. 8.)
- ✓ You may encounter the other two functions, bcopy and bcmp, in existing applications.

9. **Demonstrate about socket () , listen(), accept( )system calls in Linux?**

Ans:

Socket()
https://www.tutorialspoint.com/unix_system_calls/socket.htm

Listen() https://www.tutorialspoint.com/unix_system_calls/listen.htm

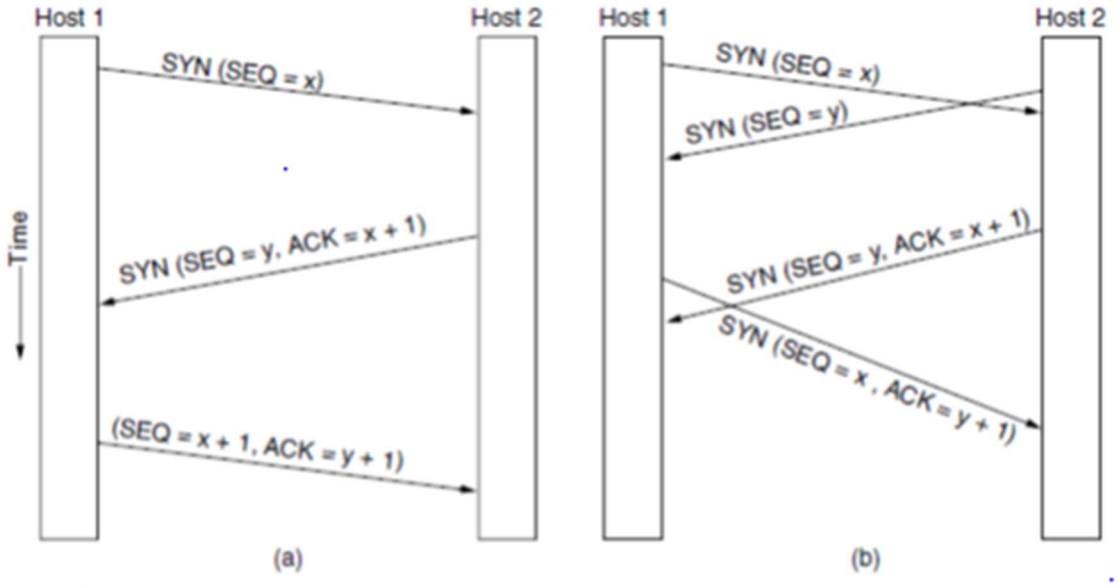
Accept()
https://www.tutorialspoint.com/unix_system_calls/accept.htm

10. **Illustrate about how TCP connections are established and terminated.**

Ans:

TCP Connection Establishment:

- ✓ To establish a connection, one side, say the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives in that order, either specifying a specific source or nobody in particular.
- ✓ The other side, say, the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password).
- ✓ The CONNECT primitive sends a TCP segment with the SYN bit on and ACK bit off and waits for a response.
- ✓ When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the Destination port field. If not, it sends a reply with the RST bit on to reject the connection.
- ✓ If some process is listening to the port, that process is given the incoming TCP segment. It can either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig below. Note that a SYN segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously.



a) TCP connection establishment in the normal case. (b) Simultaneous connection establishment on both sides.

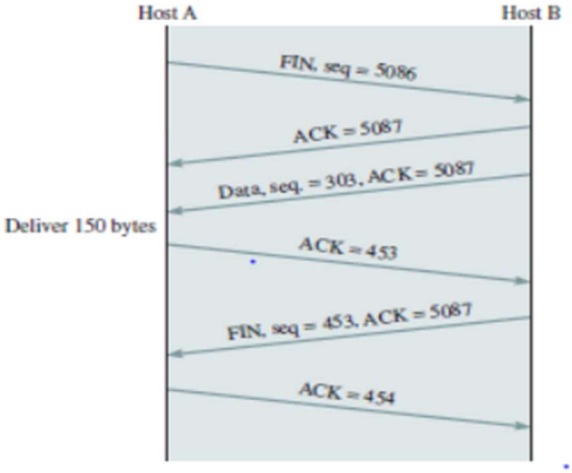
- ✓ In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Fig.(b). The result of these events is that just one

connection is established, not two, because connections are identified by their end points.

- ✓ If the first setup results in a connection identified by (x, y) and the second one does too, only one table entry is made, namely, for (x, y).

TCP Connection Termination:

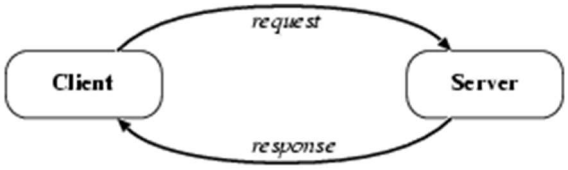
- ✓ TCP provides for a graceful close that involves independent termination of each direction of the connection. A termination is initiated when an application tells TCP that it has no more data to send.
- ✓ The TCP entity completes transmission of its data and upon receiving acknowledgement from the receiver, issues a segment with the FIN bit set.
- ✓ Upon receiving a FIN segment, A TCP entity informs its application that the other entity has terminated its transmission of data.
- ✓ For e.g. as shown in fig below the TCP entity in host A terminates its transmission first by issuing a FIN segment. Host B sends an ACK segment to acknowledge receipt of the FIN segment from A. The FIN segment uses one byte, so the ACK is 5087 as shown in the example.
- ✓ After B receives the FIN segment, the direction of the flow from B to A is still open. In fig below host B sends 150 bytes in one segment, followed by a FIN segment. Host A then send s an acknowledgment. The TCP in host A then enters the TIME_WAIT state and starts the TIME_WAIT timer with an initial value set to twice the maximum segment lifetime (2MSL).
- ✓ The only valid segment that can arrive while host A is in the TIME_WAIT state is a retransmission of the FIN segment from host B(if host A's ACK was lost, and host B's retransmission time-out has expired).
- ✓ If such a FIN segment arrives while host A is the TIME_WAIT state, then the ACK segment is retransmitted and the TIME_WAIT timer is restarted at 2MSL. When the TIME_WAIT timer expires, host A closes the connection and them deletes the record of the connection.



11. **Demonstrate echo server and echo client using 6666 port in TCP style?**
Ans:
TCP Echo Client

- ✓ In the TCP Echo client a socket is created.
- ✓ Using the socket a connection is made to the server using the connect() function.
- ✓ After a connection is established , we send messages input from the user and display the data received from the server using send() and read() functions.

TCP Echo Server



- ✓ In the TCP Echo server , we create a socket and bind to a advertised port number.
- ✓ After binding , the process listens for incoming connections.
- ✓ Then an infinite loop is started to process the client requests for connections.
- ✓ After a connection is requested , it accepts the connection from the client machine and forks a new process.
- ✓ The new process receives data from the client using recv() function and echoes the same data using the send() function.
- ✓ Please note that this server is capable of handling multiple clients as it forks a new process for every client trying to connect to the server.

Part-C

1. **Develop a program to implement UDP chat client server?**
Ans:
UDP SERVER:

```
// Server side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

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

#define PORT     8080
#define MAXLINE 1024

// Driver code
int main() {
	int sockfd;
	char buffer[MAXLINE];
	char *hello = "Hello from server";
	struct sockaddr_in servaddr, cliaddr;

	// Creating socket file descriptor
	if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
		perror("socket creation failed");
		exit(EXIT_FAILURE);
	}

	memset(&servaddr, 0, sizeof(servaddr));
	memset(&cliaddr, 0, sizeof(cliaddr));

	// Filling server information
	servaddr.sin_family = AF_INET; // IPv4
	servaddr.sin_addr.s_addr = INADDR_ANY;
	servaddr.sin_port = htons(PORT);

	// Bind the socket with the server address
	if ( bind(sockfd, (const struct sockaddr *)&servaddr,
			sizeof(servaddr)) < 0 )
	{
		perror("bind failed");
		exit(EXIT_FAILURE);
	}

	int len, n;

	len = sizeof(cliaddr); //len is value/resuslt

	n = recvfrom(sockfd, (char *)buffer, MAXLINE,
```

```
                        MSG_WAITALL, ( struct sockaddr *) &cliaddr,
                        &len);
        buffer[n] = '\0';
        printf("Client : %s\n", buffer);
        sendto(sockfd, (const char *)hello, strlen(hello),
                MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
                        len);
        printf("Hello message sent.\n");

        return 0;
}
```

UDP CLIENT:

```
// Client side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT         8080
#define MAXLINE 1024

// Driver code
int main() {
        int sockfd;
        char buffer[MAXLINE];
        char *hello = "Hello from client";
        struct sockaddr_in  servaddr;

        // Creating socket file descriptor
        if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
                perror("socket creation failed");
                exit(EXIT_FAILURE);
        }

        memset(&servaddr, 0, sizeof(servaddr));

        // Filling server information
```

```
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;

    int n, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof(servaddr));
    printf("Hello message sent.\n");

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                    MSG_WAITALL, (struct sockaddr *) &servaddr,
                    &len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

    close(sockfd);
    return 0;
}
```

2. **Demonstrate client and server programming using UDP protocol with neat diagram?**

Ans:

- ✓ In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram.
- ✓ Similarly, the server need not accept a connection and just waits for datagrams to arrive.
- ✓ Datagrams upon arrival contain the address of sender which the server uses to send data to the correct client.
- ✓ The entire process can be broken down into following steps:

UDP Server:

1. Create UDP socket.
2. Bind the socket to the server address.
3. Wait until the datagram packet arrives from the client.
4. Process the datagram packet and send a reply to the client.
5. Go back to Step 3.

UDP Client:

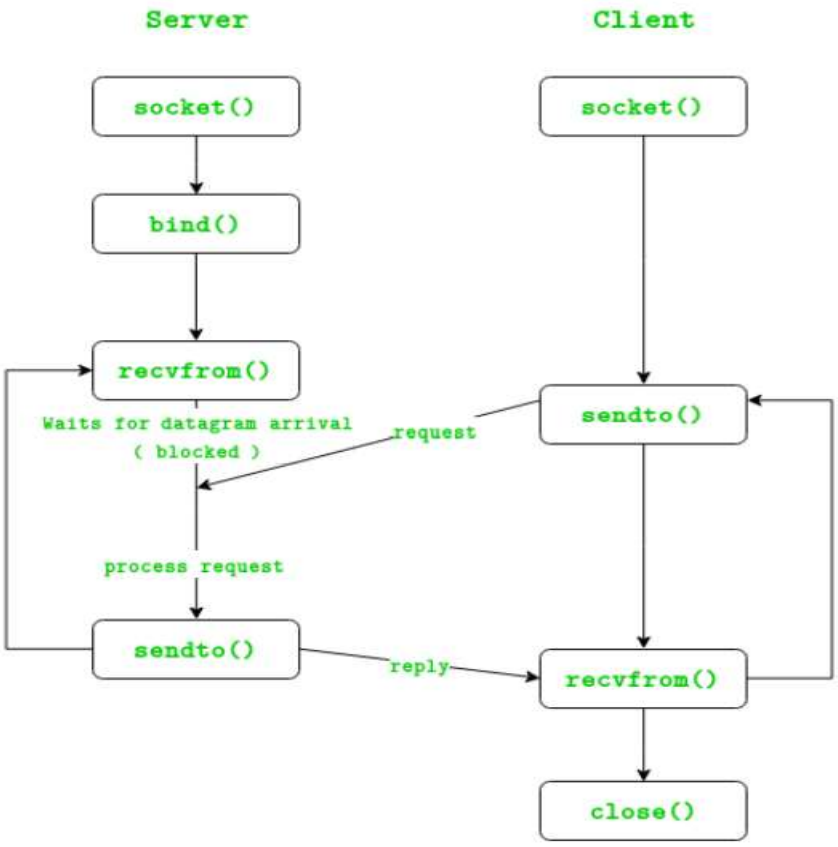
1. Create UDP socket.

2. Send message to server.

3. Wait until a response from the server is received.

4. Process reply and go back to step 2, if necessary.

5. Close socket descriptor and exit.



3. **Differentiate between TCP and UDP protocols**

Ans:

	Transmission control protocol (TCP)	**User datagram protocol (UDP)**
	TCP is a connection-oriented protocol. Connection-orientation means that the communicating devices should establish a connection before transmitting data and should close the connection after transmitting the data.	UDP is the Datagram oriented protocol. This is because there is no overhead for opening a connection, maintaining a connection, and terminating a connection. UDP is efficient for broadcast and multicast type of network transmission.
	TCP is reliable as it guarantees the delivery of data to the destination router.	The delivery of data to the destination cannot be guaranteed in UDP.

TCP provides extensive error checking mechanisms. It is because it provides flow control and acknowledgment of data.	UDP has only the basic error checking mechanism using checksums.
Sequencing of data is a feature of Transmission Control Protocol (TCP). this means that packets arrive in-order at the receiver.	There is no sequencing of data in UDP. If the order is required, it has to be managed by the application layer.
TCP is comparatively slower than UDP.	UDP is faster, simpler, and more efficient than TCP.
Retransmission of lost packets is possible in TCP, but not in UDP.	There is no retransmission of lost packets in the User Datagram Protocol (UDP).
TCP has a (20-60) bytes variable length header.	UDP has an 8 bytes fixed-length header.
TCP is heavy-weight.	UDP is lightweight.
TCP doesn't support Broadcasting.	UDP supports Broadcasting.
TCP is used by HTTP, HTTPs, FTP, SMTP and Telnet.	UDP is used by DNS, DHCP, TFTP, SNMP, RIP, and VoIP.

4. **Illustrate by writing a c program to implement TCP chat client server?**

Ans:

TCP SERVER:

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr

// Function designed for chat between client and server.
void func(int sockfd)
```

```
{
	char buff[MAX];
	int n;
	// infinite loop for chat
	for (;;) {
		bzero(buff, MAX);

		// read the message from client and copy it in buffer
		read(sockfd, buff, sizeof(buff));
		// print buffer which contains the client contents
		printf("From client: %s\t To client : ", buff);
		bzero(buff, MAX);
		n = 0;
		// copy server message in the buffer
		while ((buff[n++] = getchar()) != '\n')
			;

		// and send that buffer to client
		write(sockfd, buff, sizeof(buff));

		// if msg contains "Exit" then server exit and chat ended.
		if (strncmp("exit", buff, 4) == 0) {
			printf("Server Exit...\n");
			break;
		}
	}
}

// Driver function
int main()
{
	int sockfd, connfd, len;
	struct sockaddr_in servaddr, cli;

	// socket create and verification
	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	if (sockfd == -1) {
		printf("socket creation failed...\n");
		exit(0);
	}
	else
		printf("Socket successfully created..\n");
	bzero(&servaddr, sizeof(servaddr));
```

```
	// assign IP, PORT
	servaddr.sin_family = AF_INET;
	servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
	servaddr.sin_port = htons(PORT);

	// Binding newly created socket to given IP and verification
	if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
		printf("socket bind failed...\n");
		exit(0);
	}
	else
		printf("Socket successfully binded..\n");

	// Now server is ready to listen and verification
	if ((listen(sockfd, 5)) != 0) {
		printf("Listen failed...\n");
		exit(0);
	}
	else
		printf("Server listening..\n");
	len = sizeof(cli);

	// Accept the data packet from client and verification
	connfd = accept(sockfd, (SA*)&cli, &len);
	if (connfd < 0) {
		printf("server acccept failed...\n");
		exit(0);
	}
	else
		printf("server acccept the client...\n");

	// Function for chatting between client and server
	func(connfd);

	// After chatting close the socket
	close(sockfd);
}
```

TCP CLIENT:

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

```
#include <string.h>
#include <sys/socket.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
void func(int sockfd)
{
	char buff[MAX];
	int n;
	for (;;) {
		bzero(buff, sizeof(buff));
		printf("Enter the string : ");
		n = 0;
		while ((buff[n++] = getchar()) != '\n')
			;
		write(sockfd, buff, sizeof(buff));
		bzero(buff, sizeof(buff));
		read(sockfd, buff, sizeof(buff));
		printf("From Server : %s", buff);
		if ((strncmp(buff, "exit", 4)) == 0) {
			printf("Client Exit...\n");
			break;
		}
	}
}

int main()
{
	int sockfd, connfd;
	struct sockaddr_in servaddr, cli;

	// socket create and varification
	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	if (sockfd == -1) {
		printf("socket creation failed...\n");
		exit(0);
	}
	else
		printf("Socket successfully created..\n");
	bzero(&servaddr, sizeof(servaddr));

	// assign IP, PORT
	servaddr.sin_family = AF_INET;
```

```
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);

    // connect the client socket to server socket
    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
        printf("connection with the server failed...\n");
        exit(0);
    }
    else
        printf("connected to the server..\n");

    // function for chat
    func(sockfd);

    // close the socket
    close(sockfd);
}
```

5. **Differentiate stream sockets and raw sockets and related system calls?**
   Ans:
   Stream Sockets –
   - ✓ Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C".
   - ✓ These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator.
   - ✓ Data records do not have any boundaries.

Tag	Description
SOCK_STREAM	
	Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmissio mechanism may be supported.
SOCK_DGRAM	
	Supports datagrams (connectionless, unreliable messages of a fixed maximum length).
SOCK_SEQPACKET	
	Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each read system call.
SOCK_RAW	
	Provides raw network protocol access.
SOCK_RDM	
	Provides a reliable datagram layer that does not guarantee ordering.
SOCK_PACKET	
	Obsolete and should not be used in new programs; see **packet**(7).

Raw Sockets –

- ✓ These provide users access to the underlying communication protocols, which support socket abstractions.
- ✓ These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol.
- ✓ Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

Constant	Description
AF_LOCAL	Local communication
AF_UNIX	Unix domain sockets
AF_INET	IP version 4
AF_INET6	IP version 6
AF_IPX	Novell IPX
AF_NETLINK	Kernel user interface device
AF_X25	Reserved for X.25 project
AF_AX25	Amateur Radio AX.25
AF_APPLETALK	Appletalk DDP
AF_PACKET	Low level packet interface
AF_ALG	Interface to kernel crypto API

6. **Over the socket? is there a way to have a dynamic buffer? What does one do when one does not know how much information is coming?**
   Ans:
   - ✓ When the size of the incoming data is unknown, you can either make the size of the buffer as big as the largest possible (or likely) buffer, or you can re-size the buffer on the fly during your read.
   - ✓ When you malloc() a large buffer, most (if not all) varients of unix will only allocate address space, but not physical pages of ram.
   - ✓ As more and more of the buffer is used, the kernel allocates physical memory.
   - ✓ This means that malloc'ing a large buffer will not waste resources unless that memory is used, and so it is perfectly acceptable to ask for a meg of ram when you expect only a few K.
   - ✓ On the other hand, a more elegant solution that does not depend on the inner workings of the kernel is to use realloc() to expand the buffer as required in say 4K chunks (since 4K is the size of a page of ram on most systems).
   - ✓ I may add something like this to sockhelp.c in the example code one day.
7. **Explain address structure of IPV4 and IPV6 in sockets**
   Ans:
   An IPv4 address has the following format:
   - ✓ x . x . x . x where *x* is called an *octet* and must be a decimal value between 0 and 255. Octets are separated by periods. An IPv4 address must contain three periods and four octets. The following examples are valid IPv4 addresses:
     - 1 . 2 . 3 . 4
     - 01 . 102 . 103 . 104

   The following example shows a screen that uses IPv4 addresses.

```
Ethernet IPv4                Panel 0175


Current Settings Frame 1:


IP Address (IPv4):  19.117.63.126

MAC Address: 18:36:F3:98:4F:9A

Gateway (IPv4):     19.117.63.253

Subnet Mask (IPv4): 255.255.253.0


[BACK]  [ UP ]  [DOWN]   [ENTER]

Ethernet Mode:  Manual IP Entry

Press ENTER to Change Settings
```

An IPv6 address can have either of the following two formats:

- Normal - Pure IPv6 format
- Dual - IPv6 plus IPv4 formats

An IPv6 (Normal) address has the following format:

- ✓ y : y : y : y : y : y : y : y where *y* is called a *segment* and can be any hexadecimal value between 0 and FFFF. The segments are separated by colons - not periods.
- ✓ An IPv6 normal address must have eight segments, however a short form notation can be used in the Tape Library Specialist Web interface for segments that are zero, or those that have leading zeros.
- ✓ The short form notation can not be used from the operator panel. The following list shows examples of valid IPv6 (Normal) addresses:

- 2001 : db8: 3333 : 4444 : 5555 : 6666 : 7777 : 8888
- 2001 : db8 : 3333 : 4444 : CCCC : DDDD : EEEE : FFFF
- : : (implies all 8 segments are zero)
- 2001: db8: : (implies that the last six segments are zero)
- : : 1234 : 5678 (implies that the first six segments are zero)
- 2001 : db8: : 1234 : 5678 (implies that the middle four segments are zero)
- 2001:0db8:0001:0000:0000:0ab9:C0A8:0102 (This can be compressed to eliminate leading zeros, as follows: 2001:db8:1::ab9:C0A8:102)

8. **Write a program to implement TCP client server application in which client takes an Integer value from the command line and sends to the server. Server returns the Factorial of the received integer value to the client.**
   Ans:

9. **Write a program to implement UDP client server application in which client take a file name from the command line and sends to the server. Server returns the content of received file to the client.**
   Ans:

10. **What is the difference between connected and unconnected sockets?**
    Ans:
    - ✓ If a UDP socket is unconnected, which is the normal state after a bind() call, then send() or write() are not allowed, since no destination address is available; only sendto() can be used to send data.
    - ✓ Calling connect() on the socket simply records the specified address and port number as being the desired communications partner. That means that send() or write() are now allowed; they use the destination address and port given on the connect call as the destination of the packet.