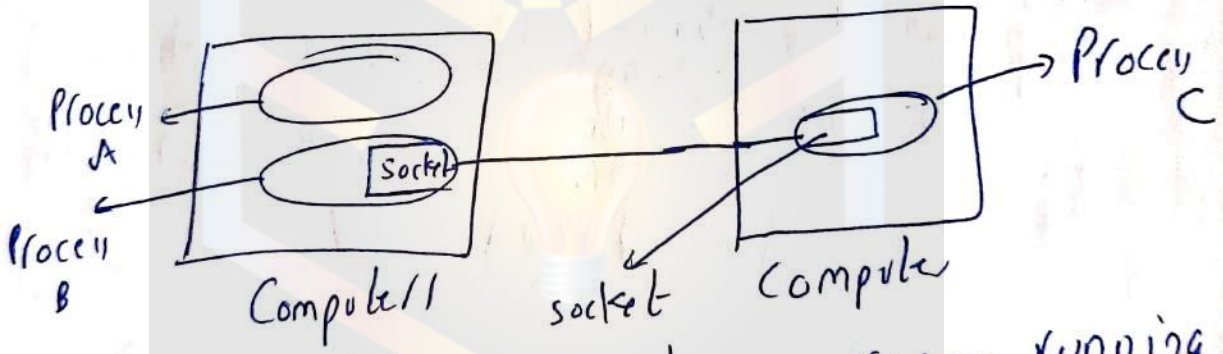
Unit V Socket programming

→ **Socket:** (1)

→ A socket acts as an end point in connection b/n a client & a server present in a network

or

→ A socket is one endpoint of a two way communication link b/n two progtams running on the network.



→ Using a socket, a process running one one computer can communicate with another process running on distinct other computer.

ex: real world example of a telephone (call)

→ Both parties have a telephone installed
→ A phone number is assigned to each telephone
→ Turn on ringer to listen for the caller.
→ Caller lifts telephone & dials a number
→ Telephone rings & the receiver of the call picks it up.
→ Both parties talk & exchange data
→ After conversation is over, they hang up the phone

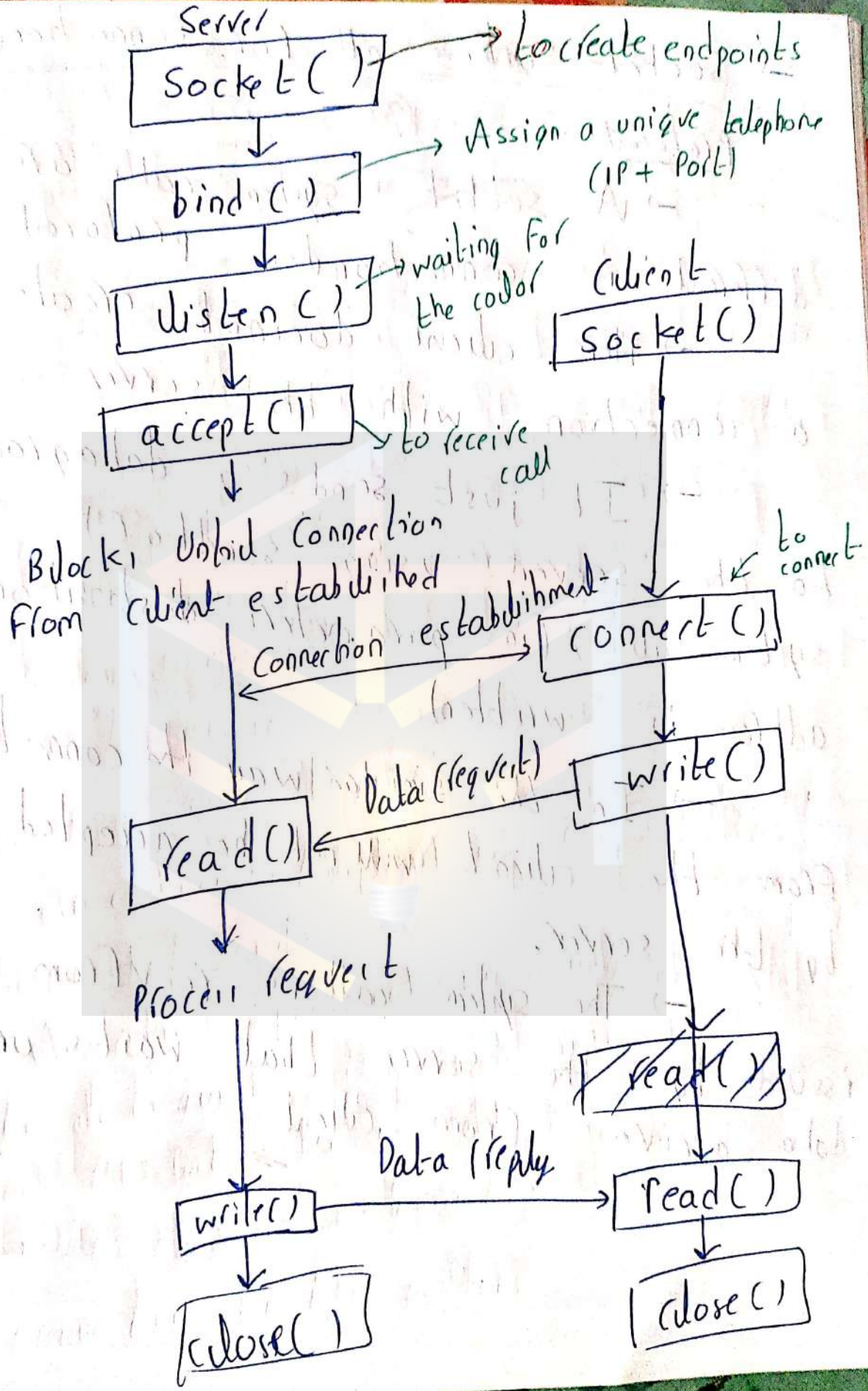
Similarly in networking:-

→ An endpoint (telephone) for communication is created on both ends
→ An address (phone no) is assigned to both ends to distinguish them from rest of the network (IP + Port number is used)
→ One of the endpoint (caller) initiate a connection to the other.

→ The otherend (receiver) point waits for communication to start.
→ Once a connection has been made, data is exchanged (talk)
→ Once data has been exchanged the endpoints are closed (hang up).

→ **Socket System call for Connection oriented protocol:-** (2)

In connection oriented systemcall transfer first the server is started, then after some time client is started that connects to the server.



→ Socket System call for Connectionless Protocol

(3)

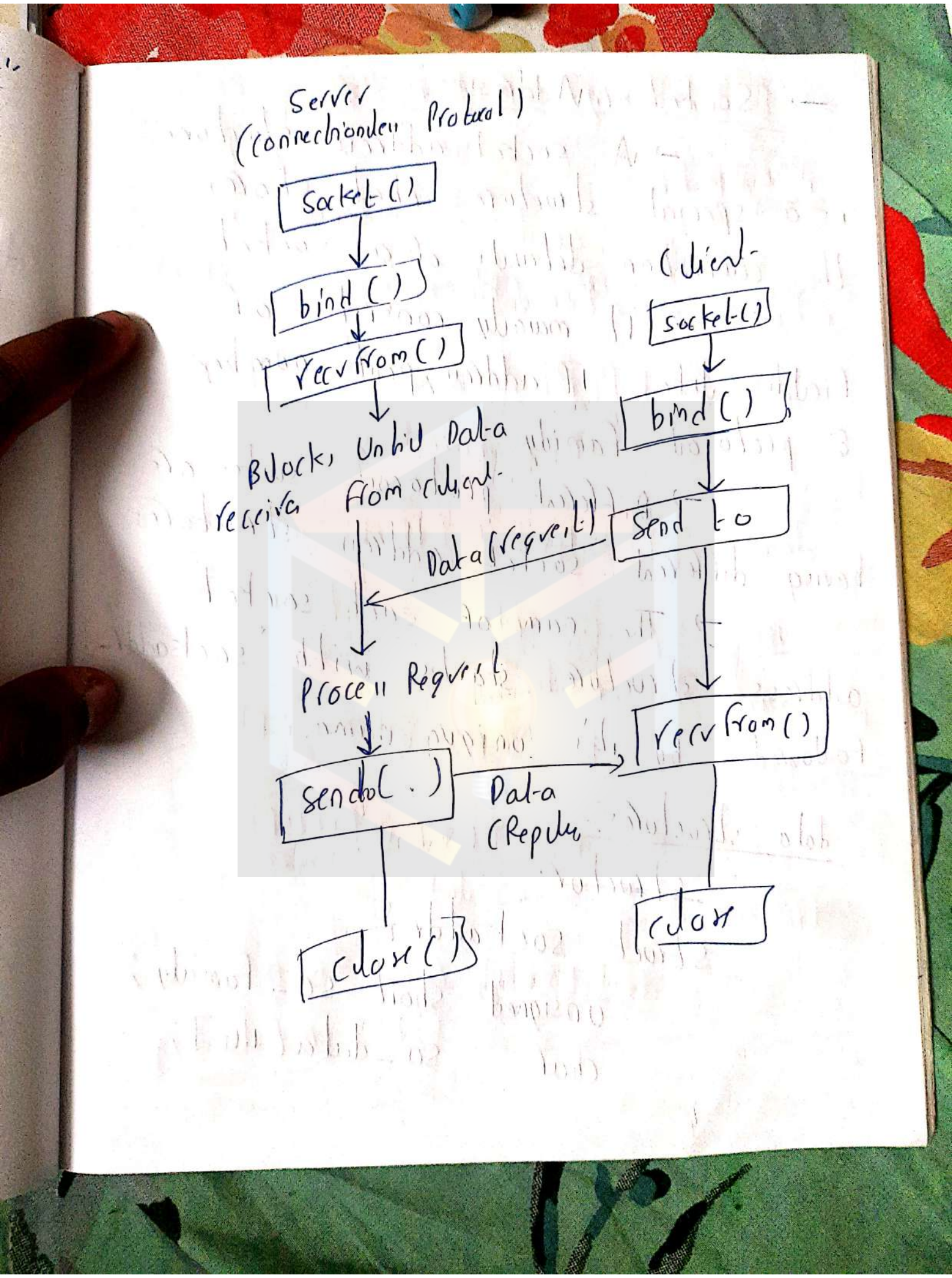
→ A ~~socket~~ system calls are different in connectionless protocol.

→ Here client doesn't create a connection with the server.

→ It just sends a datagram to the server using sendto() system call, (in parameters destination address is written).

→ In the similar way the connection from the client will not be accepted by the server.

→ The system call is recvfrom is issued by the server that waits until data arrives from client.



→ Socket Addresses:- (4)

→ A socket address structure is a special structure that stores the connection details of a socket.

→ It mainly consists of fields like IP address, port number & protocol family.

→ Different protocol suites are having different socket address structures

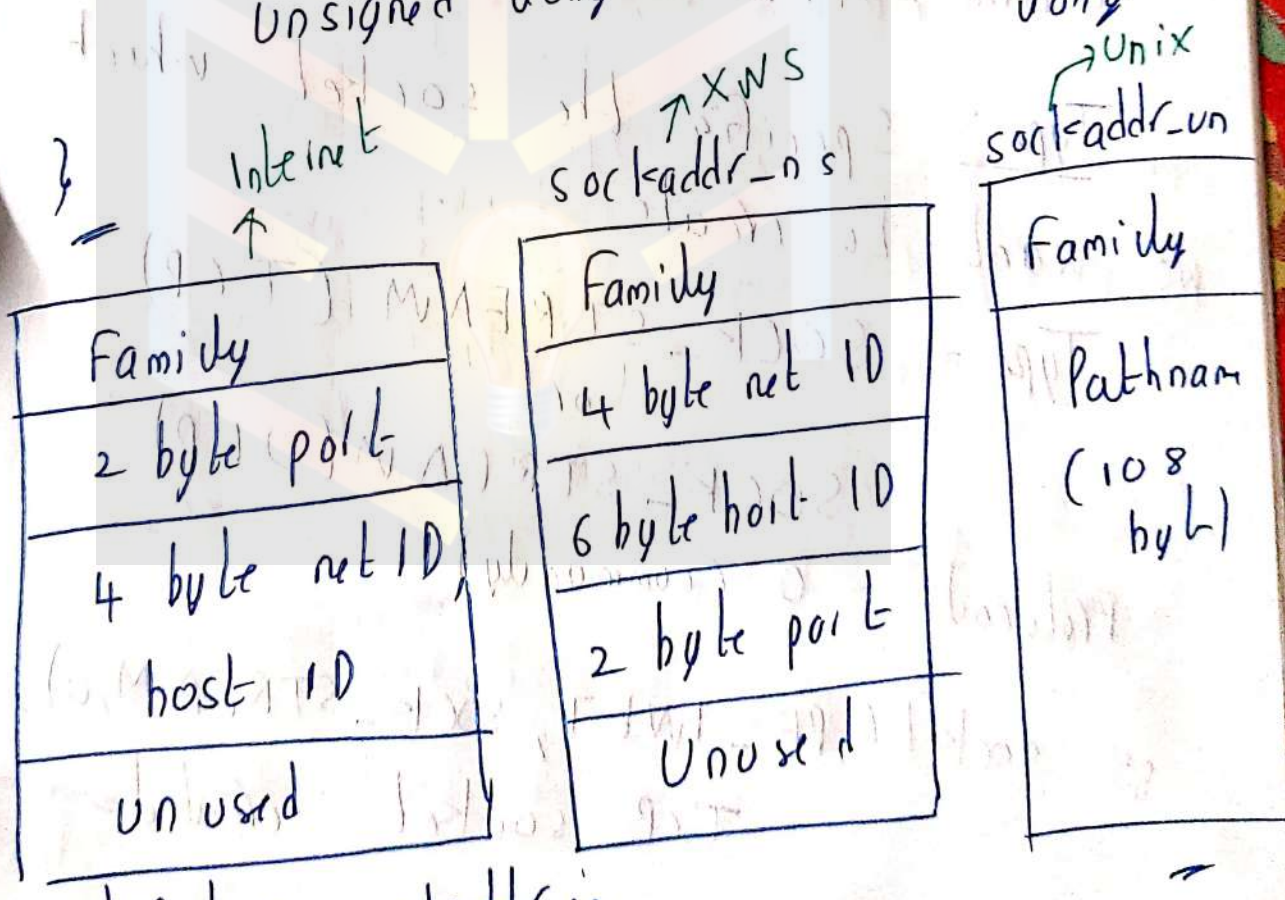
→ The name of each socket address structure starts with "sockaddr" followed by it's unique name.

data structures:-

~~Structure~~

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

```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port; // Port number.
    struct in_addr sin_addr; // IP address
    char sin_zero[8];
};
struct in_addr {
    unsigned long s_addr; // 4 byte long
};
```



structure sockaddr_in

→ Elementary socket system calls:-

i) Socket() : (a connection endpoint)

This creates an endpoint for a network connection. (b/n client & server).

int Socket(int domain, int type,
int protocol)

domain = PF_INET (IPv4 communication)

Type : Specifies the socket which we want to create

Type = SOCK_STREAM (TCP)
or
SOCK_STREAM (UDP)

Protocol : 0 (commonly)

eg: socket(PF_INET, SOCK_STREAM, 0);
TCP socket created

ii) Bind() : (IP + Port)

→ A bind() function assigns a name to an unamed socket

→ which is usually a combination of IP address & port no.

ex:

```
struct sockaddr_in my;
my.sin_family = PF_INET;
my.sin_port = htons(80);
my.sin_add.s_addr = INADDR_ANY;
bzero(&my, 8)
bind(sock, (struct sockaddr *) &my,
        size of(my));
```

synt:

```
bind(int sockfd, struct sockaddr
    * my_addr, socklen_t addrlen)
```

iii) listen() : (wait for connection)

The execution of this function indicates that a connection oriented server is ready to receive connection

int listen (int sock, int backlog)

↑ max length of pending connection

e.g. Listen (sock, 10)

↑ max 10 connection in waiting/pending state

iv) Accept() :- (A new connection)

Accept is called by a server process to accept new connection from new client try to connect the server

int Accept (int socket, (struct sockaddr *) &client, socklen_t *client_len)

v) Connect() :- (connect to a server)

Connect is called by a client to connect to a server port.

int connect(int sock, (struct sockaddr *) & server_addr, socklen_t len);

vi) Send / Recv. (Finally Data)

Send(), Recv(), Read(), write() etc. calls are used to send the data & receive the data

int send(int sock, void *msg, size_t len, int flags)

int recv(int sock, void *msg, size_t len, int flags)

vii) close() (Bye Bye)

close signals the end of communication b/n server - client.

int close(int sock);

→ Advanced socket system calls

(6) In addition to the elementary socket system calls, unix provides the following advanced socket system calls

i) readv & writev :

→ These are the variants of the standard read & write system calls and are called scatter read & gather write respectively

→ These system calls are used to read into & write from the buffers that are non contiguous.

→ They are defined in <sys/types.h> & <sys/uio.h>

Syntax

```
int writev(int fd, struct iovec
    iov[], int iovcount);
int readv(      "
                "      );
```

ii) sendmsg & recvmsg :-

→ These are the most general system calls used for reading & writing purposes → in <sys/types.h> & <sys/socket.h>

```
int sendmsg(int sock_Fd, struct
        msg_hdr msg[], int Flags);
int recvmsg(  "
              "  );
```

iii) getpeername :-

→ This function determines the foreign protocol address associated with socket.

→ The address include foreign IP address & port number

→ in <sys/socket.h>

iv) getsockname :

This function determines the local protocol address associated with a socket.

It includes local IP + port no. (defined in <sys/socket.h>

v) getsockopt & setsockopt :

reserved for sockets.

```
int getsockopt(sock_fd, level,
    optname, void * optval, int
    * optlen);
int    setsockopt(  "
                    "
                    " );
```

vi) shutdown :

The shutdown function was introduced in order to overcome the limitation of close

→ close Function when invoked completely terminates the connection & there will be no further data transfer [shutdow Function on other hand allows one side to terminate the connection while the other side might continue with its data transfer]

→ close function works by decrementing the reference count of the descriptor it closes the socket only when the reference count become zero [shutdown Function however follow the termination process (using FIN)]

→ Reserved ports :- (1) → 16 bit = 0 to 65535

0 - 1023 (well defined ports)	1024 - 4915 registered port.	4.9153 - 65535 client program can used
80 - HTTP 21 - FTP etc.		

→ Socket options: → not found popular/easy answer

→ Asynchronous I/O →

→ Input/Output multiplexing: (8)

(printed)

→ Out of Band data: (9)

→ Out of band data is an urgent data which, if occurs at one end of the network connection causes that end to tell it's peer immediately, irrespective of any flow control or blocking issues.

→ The word immediately indicates that this data is sent before the normal data even though the normal data is already queue.

(out of bound data has high priority)

→ The concept of out of band data mainly arise in the transport layer

→ Protocols SPP & TCP supports this data in different ways where as UDP does not support it.

(SPP - Sequenced Packet protocol)

i) SPP out of Band data:-

It is simple method

ii) TCP out of Band data:-

It is complex when compared to SPP

→ no proper explanation found.

→ Internet Super Server:-

→ no proper theory / explanation found.

NETWORK PROGRAMMING: I/O MULTIPLEXING

I/O multiplexing is the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready, like input is ready to be read, or descriptor is capable of taking more output.

Scenarios is which I/O multiplexing is used –

1. When client is handling multiple descriptors (like standard input and network socket).
2. When client handles multiple sockets at the same time, example - Web client.
3. When TCP server handles both listening and its connected sockets.
4. When server handles both TCP and UDP.
5. When server handles multiple services and perhaps multiple protocols.

I/O Models

There are 5 I/O models -

1. Blocking I/O
2. Non-blocking I/O
3. I/O multiplexing

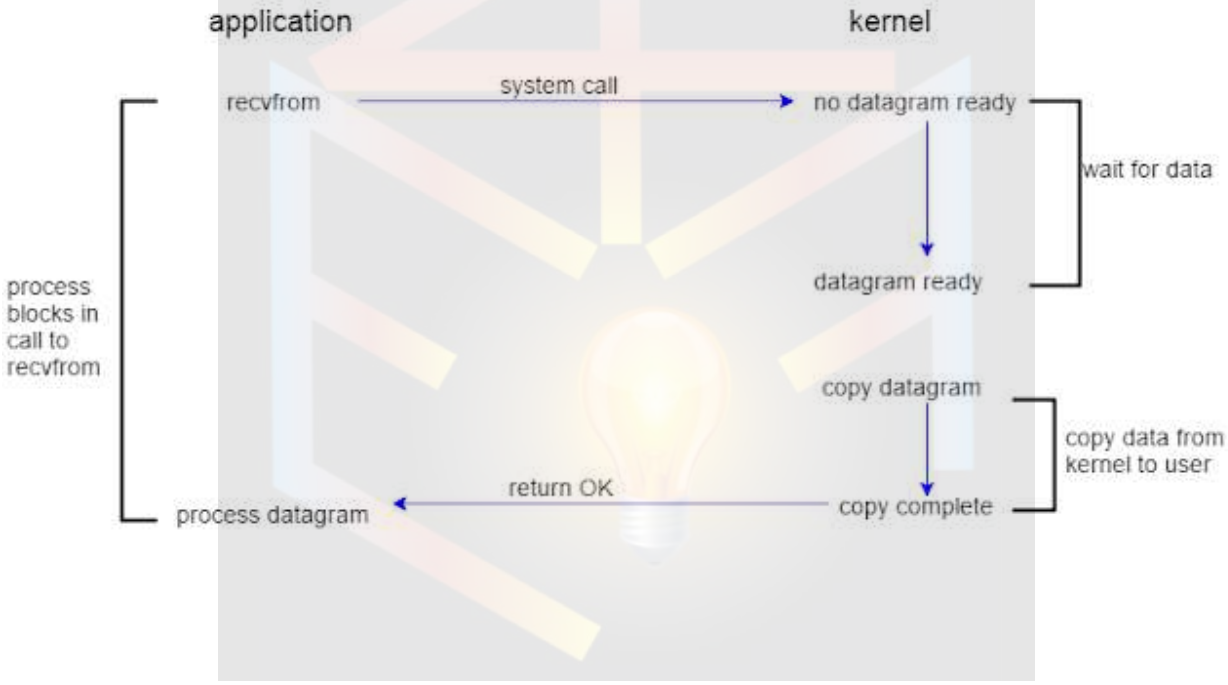
4. Signal driven I/O
5. Asynchronous I/O

There are two phases for any input operation -

1. Waiting for data to be ready
2. Copying data from kernel to process

Blocking I/O Model

By default, all sockets are blocking.



We use UDP in this diagram because it's easier, as we only have to deal with sending and receiving datagrams.

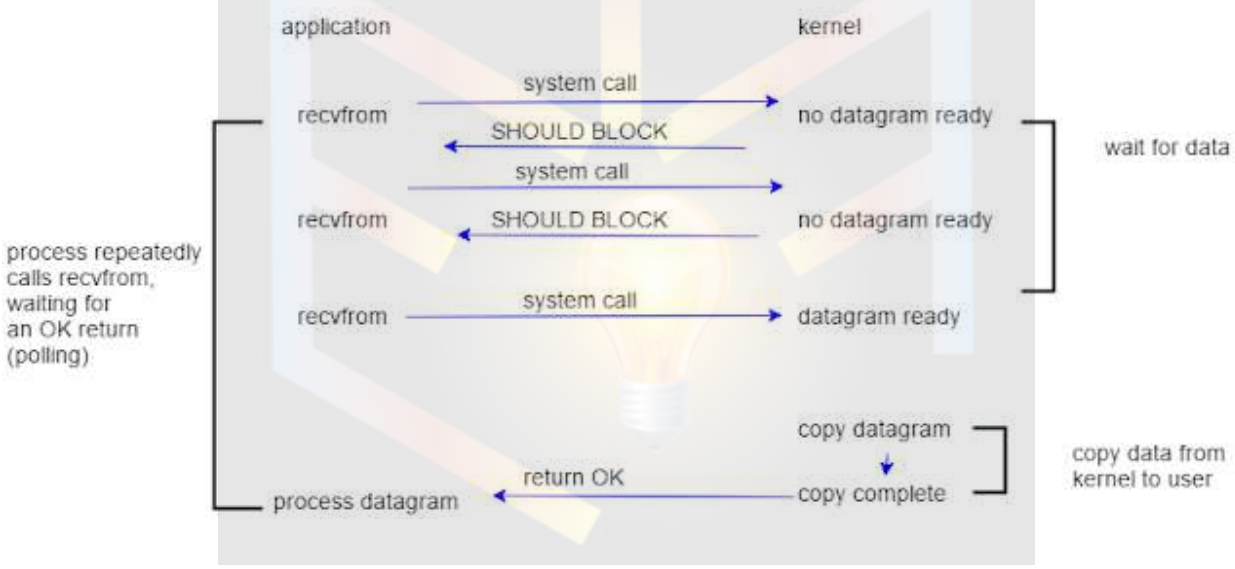
There is a switch from running in the application to running in the kernel, and returning back to application.

Most common error occurring is when system call is interrupted by a signal.

Non-blocking I/O Model

When we set a socket to nonblocking, we tell the kernel that if an I/O request from me will put the process to sleep, return an error instead of blocking the process (putting the process to sleep).

This method is called polling. It is often a waste of CPU time, but sometimes used on system dedicated to one function.

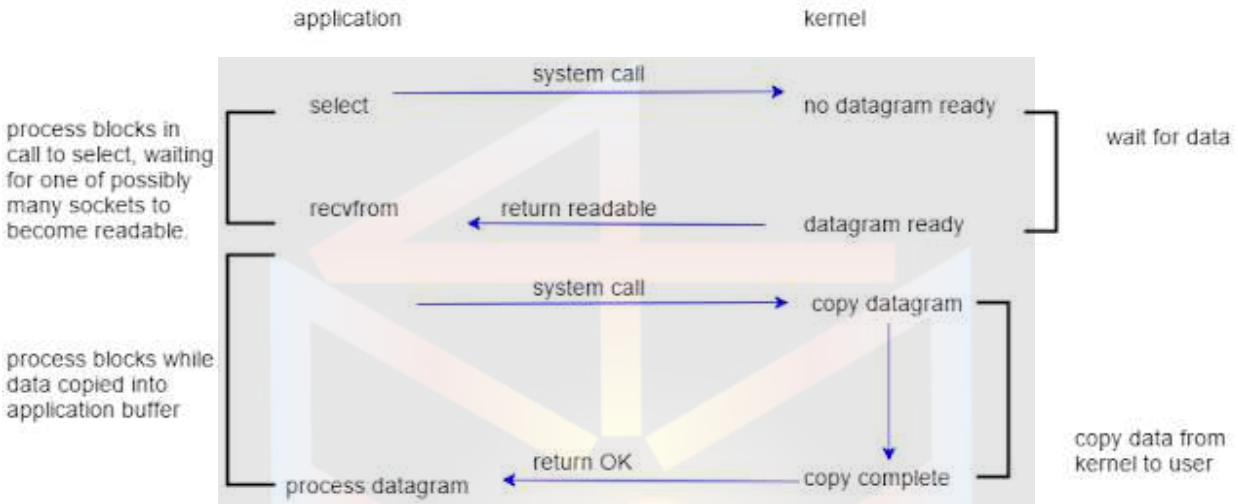


I/O Multiplexing Model

Disadvantage - *select* requires two system calls instead of one.

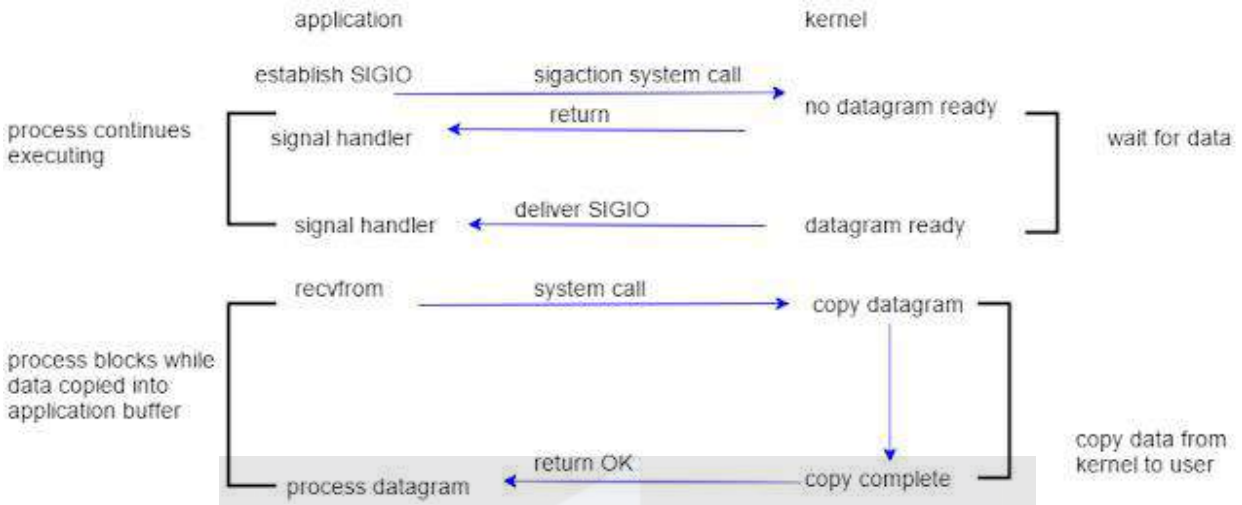
Advantage - We can wait for more than one descriptor to be ready.

Another alternative is using multithreading. Instead of using *select* to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking system calls like *recvfrom*.



Signal Driven I/O Model

Signals can tell the kernel to notify user process with SIGIO signal when descriptor is ready.



First enable the socket for signal driven I/O and install signal handler using *sigaction* system call. Return from this system call is immediate and our process continues, it is not blocked. When datagram is ready to be read, SIGIO signal is generated for our process. We can either read the datagram from the signal handler by calling *recvfrom* and then notify the main loop that data is ready to be processed, or we can notify the main loop and let it read the datagram.
Advantage - process is not blocked while waiting for data to arrive, can continue executing.

Asynchronous I/O Model

It works by telling the kernel to start operation and notify the process when it's complete, unlike signal driven I/O where kernel notifies when process is initiated.