Date
Page

1. Why UDP is faster than TCP? Which function is responsible for sending SYN segment during TCP connection establishment phase? Illustrate the TCP different states used in TCP connection and termination with suitable state ~~d~~ transition diagram.

=) ↳ UDP is faster than TCP because UDP provides a connectionless service. There is no need to have a long-term relationship between a UDP client and server whereas TCP is a connection-oriented protocol which makes it slower compare to UDP.

↳ The client issues an active open by calling connect. This causes the client TCP to send a "synchronize" (SYN) segment which tells the server the clients initial sequence number for the data that the client will send on the connection.

↳ The following scenario occurs when a TCP connection is established.

1. The server must be prepared to accept an incoming connection. This is normally done by calling socket, bind and listen and is called a passive open.
2. The client issues an active open by calling connect. This causes the client TCP to send a "synchronize" (SYN) segment, which tells the server the clients initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header and possible TCP options.
3. The server must acknowledge (ACK) the clients SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends its SYN and the ACK of the clients SYN is a single segment.

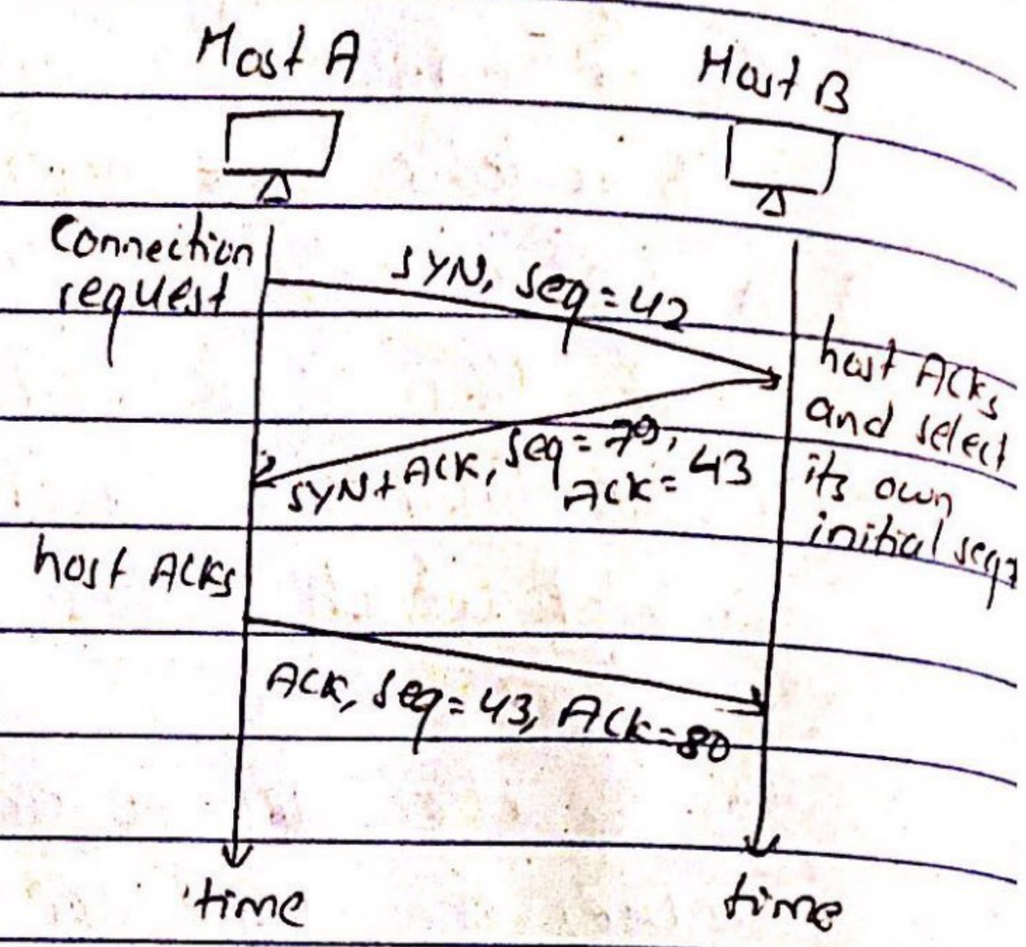
Scanned with CamScanner
Scanned with CamScanner

4. The client must acknowledge the server's SYN.

Three-way handshake:

Step-1: Client host sends TCP SYN segment to server
- specifies a random initial seq #
- no data

Step-2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #



∴ Three-way handshake

Step-3: Client receivers SYNACK, replies with ACK segment, which may contain data

Seq #'s: • byte stream "number" of first byte in segment's data

ACK's: • seq # of next byte expected from other side
- cumulative ACK

5) TCP Connection Termination:

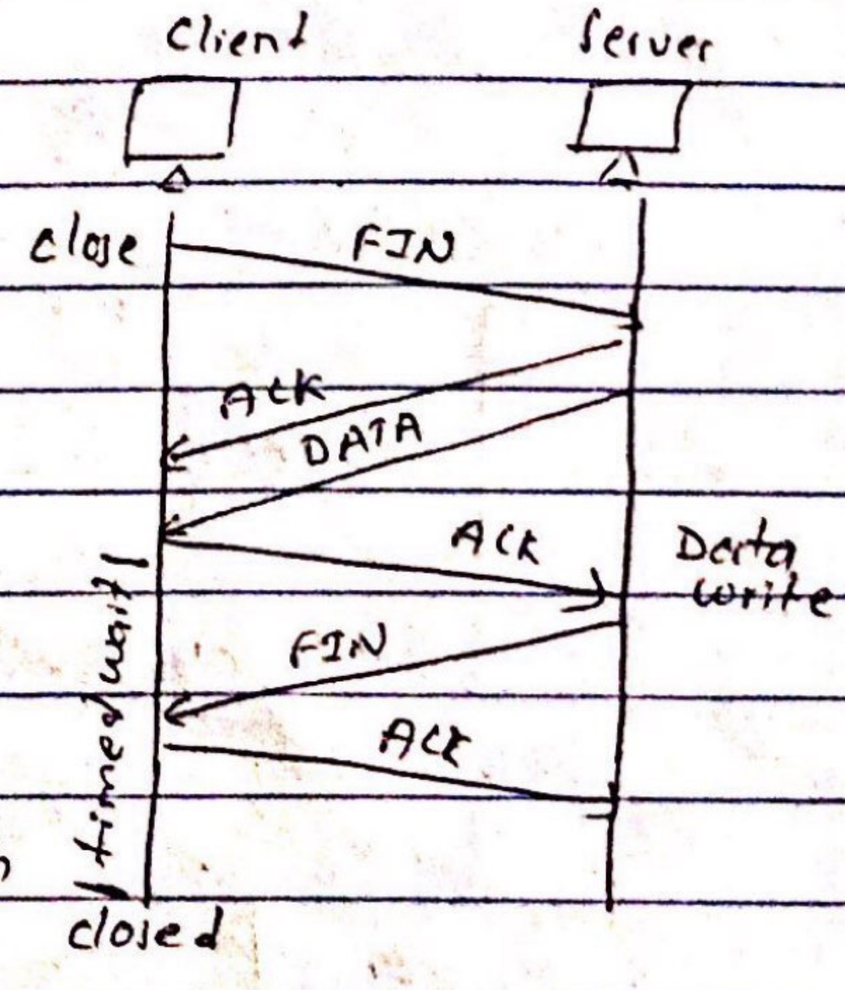
1. One application calls close first, and we say that this end performs the active close. This end's TCP sends a FIN segment, which means it is finished sending data.

2. The other end that receives the FIN performs the passive close. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file, since the receipt of the FIN means the application will not receive any additional data on the connection.

Scanned with CamScanner
Scanned with CamScanner

3. Sometime later, the application that received the end-of-file will close its socket. This causes its TCP to send a FIN.

4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.



<u>Closing a connection</u>:

client closes socket:

clientSocket.close();

<u>Step-1</u>: client end system sends TCP FIN control segment to server.

<u>Step-2</u>: server receives FIN, replies with Ack. Server might send some buffered but not sent data before closing the connection. Server then sends FIN and moves to closing state.

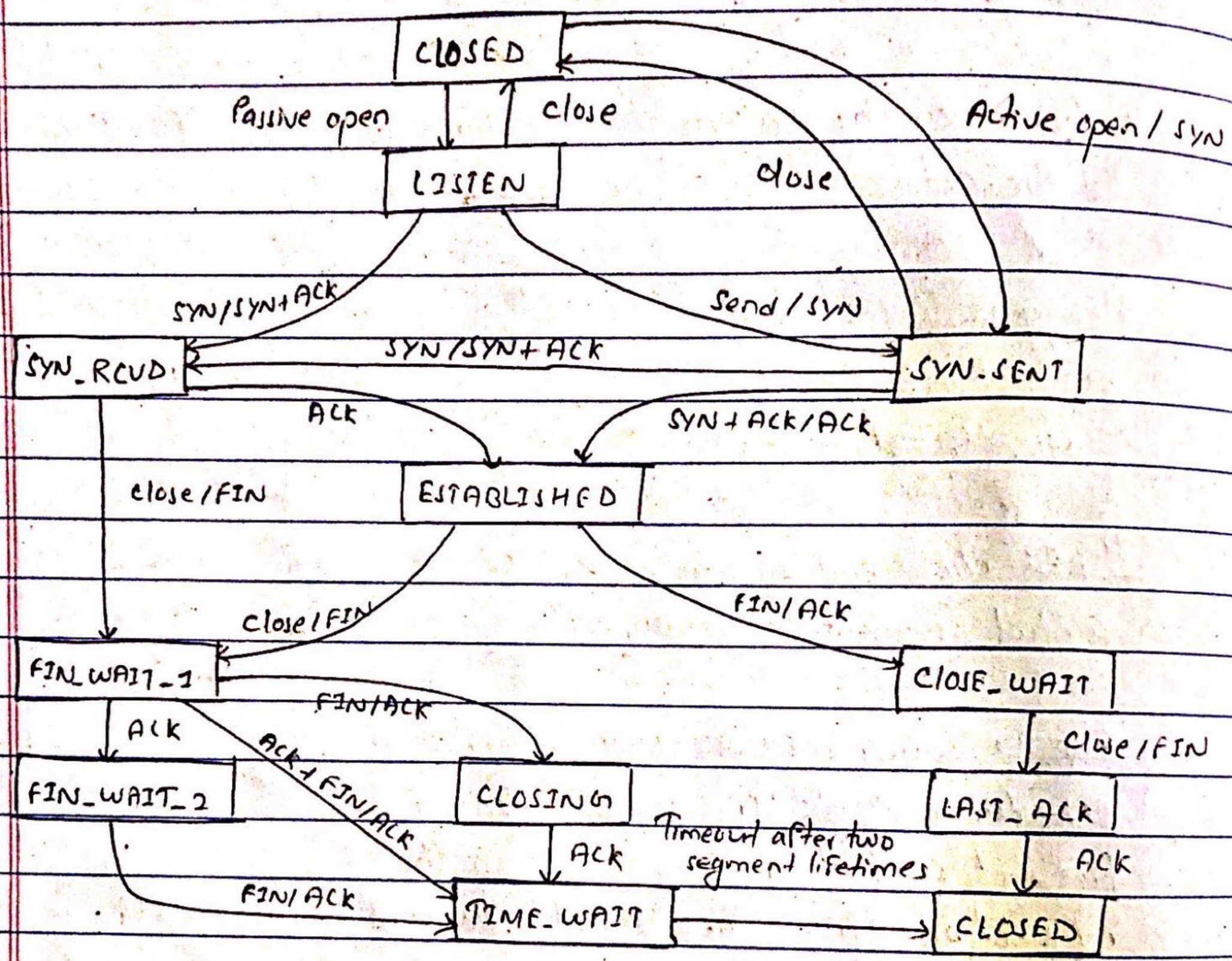
<u>Step-3</u>: client receives FIN, replies with Ack.

- Enters "timed wait" - will respond with Ack to receive FINs

<u>Step-4</u>: server, receives Ack. Connection closed.

Scanned with CamScanner
Scanned with CamScanner

TCP state- Transition Diagram:



2. How network programming is different from computer network? Explain different protocols (at least 3) in each layer of OSI reference model. Compare TCP, UDP and SCTP.

⇒ Computer Networking aims to study and analyze the communication process among various computing devices or computer systems that are linked or networked together to exchange information and share resources.

Network programming is about writing computer programs that talk to each other over a computer network.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

4) OSI reference model:

i) Physical layer : protocols are Bluetooth, RS-232, RS-449

a) Bluetooth: it is a short-range wireless technology standard that is used for exchanging data between fixed mobile device over short distance using UHF radio waves in the ISM bands from 2.402 GHz to 2.48 GHz and building personal area network.

b) RS-232: It is a standard protocol used for serial communication it is used for connecting computer and its peripheral devices to allow serial data exchange between them.

c) RS-449: It is a data standard that specifies the functional and mechanical characteristics of the interface between data terminal equipment, typically a computer and data communication equipment typically a modem or terminal server.

ii) Data Link Layer: Protocols are Frame Relay, ARP, SLIP

a) Frame Relay: It is a protocol that defines how frames are routed through a fast-packet network based on the address field in the frame.

b) ARP: It works between layer 2 and layer 3 of the OSI reference model which is used when a device wants to communicate with some other device on a local network.

c) SLIP: It is a simple protocol that works with TCP/IP for communication over serial ports and routers.

Scanned with CamScanner
Scanned with CamScanner

iii) Network layer : protocols are IP, ARP, ICMP.

a) IP : It is the network layer communications protocol in the internet protocol suite for relaying datagrams across network boundaries.

b) ARP : It is a network protocol used to find out the hardware (MAC) address of the device from an IP address.

c) ICMP : The internet Control message protocol is a supporting protocol in the internet protocol suite. It is used by network devices. including routers to send error messages and operation information indication success or failure when communicating with another IP address.

iv) Transport layer : protocols are TCP, UDP, SCTP.

a) TCP : It is designed to send packets across the internet and ensure the successful delivery of data and messages over networks.

b) UDP : It allows data to be transferred very quickly, but it can also cause packets to become lost in transmit and create opportunities for exploitation in the form of DDoS attacks.

c) SCTP : It provides multi homing support where one or both endpoints of a connection can consist of more than one IP address. This enables transparent failure between redundant network paths.

v) Session layer : protocols are RTP, PPTP, RPCP

Scanned with CamScanner
Scanned with CamScanner

a) RTP : It provide feedback on the quality of service (QoS) in media distribution periodically sending statistical information such as transmitted octet and packet counts or packet loss to the participants in the streaming multimedia session. (Real-time transport protocol).

b) PPTP (Point-to-point Tunneling protocol) : It provides a method for implementing virtual private networks. It provides security levels and remote access level comparable with typical VPN products.

c) RPCP (Remote Procedure Call Protocol): It is used when a computer program causes a procedure (or a sub-routine) to execute in a different address space without programmer explicitly coding the details for the remote interaction.

vi) Presentation layer: Protocols are SSL, MIME, NDR.

a) SSL (Secure Socket Layer): It is a protocol for web browser and servers that allows for the authentication, encryption and decryption of data sent over the internet.

b) MIME (Multipurpose Internet Mail Extension): It is an internet standard that provides scalable capable of email for attaching of image, sound and text in a message.

c) NDR (Network Data Representation) : It provides or define various primitive data types, constructed data types and also several types of data representations.

Scanned with CamScanner
Scanned with CamScanner

vii) Application layer: protocols are FTP, TFTP, Telnet

a) FTP (file Transfer Protocol): It promotes sharing of files via remote computers with reliable and efficient data transfer.

b) TFTP (Trivial file Transfer Protocol): It is the protocol of choice for transferring files between network devices and is simplified version of FTP.

c) Telnet: It helps in terminal Emulation. It allows telnet clients to access the resources of the telnet server. It is used for managing files on the internet.

Compairing TCP, UDP and SCTP.

TCP	SCTP
1. TCP doesn't supports multistreaming	1. SCTP supports multistreaming
2. TCP data transfer is less secure.	2. SCTP has more secure data transfer.
3. There is no partial data transfer in TCP.	3. There is partial data transfer in SCTP.
4. TCP doesn't supports a multi-haming.	4. Multi-haming is supported by SCTP.
5. TCP doesnot have unordered data delivery.	5. There is unordered data delivery in SCTP.
6. In TCP, the selective Acks are optional.	6. In SCTP, there are selective Acks.
7. There is less reliable data transfer in TCP.	7. There is more reliable data transfer in SCTP.

Scanned with CamScanner
Scanned with CamScanner

SCTP	UDP
1. SCTP supports multi-streaming.	1. UDP doesn't support multistreaming.
2. Data transfer is more reliable than SCTP.	2. There is no reliable data transfer in UDP.
3. Multi homing is supported by SCTP.	3. Multi homing is not supported by UDP.
4. SCTP protocol is connection-oriented.	4. UDP protocol is not connection-oriented.
5. There is partial data transfer in SCTP.	5. There is no partial data transfer in UDP.

3. While establishing TCP connection; the initial sequence number shouldnot start from 0. Why? What is 2MSL wait state?

⇒ - To protect against two incarnations of the same connection reusing the same sequence numbers too soon.

- That is, while there is still a chance that a segment from an earlier incarnation of a connection will interfere with a later incarnation of the connection.

How?

↳ Client machine seq#0, initiates connection to server with seq#0.

↳ Client sends one byte and client machine crashes.

↳ Client reboots and initiates connection again.

↳ Server thinks new incarnation is the same as old connection.

Scanned with CamScanner
Scanned with CamScanner

2MSL Wait State = TIME_WAIT

↳ When TCP does an active close, and sends the final ACK, the connection must stay in the TIME_WAIT state for twice the maximum segment lifetime.

2MSL = 2* Maximum Segment Lifetime

↳ Why?

TCP is given a chance to resent the final Ack. (Server will timeout after sending the FIN segment and resend the FIN).

↳ The MSL is set to 2 minutes or 1 minute or 30 seconds.

4. What is socket API? Explain different socket address structures available in unix network programming.

⇒ A socket API is an application-programming interface (API) usually provided by the operating system, that allows application programs to control and use network sockets.

↳ Generic socket address structure to hold socket information. It is passed in most of the socket function calls. Unix Socket Programming provides IP version specific socket address structures:

Generic socket address:

- for address arguments to connect, bind and accept.

```
struct sockaddr {
        unsigned short  sa_family ;   /* protocol family */
        char            sa_data [14];  /* address data */
};
```

Internet-specific socket address (IPv4 socket address structure):

- Must cast (sockaddr_in *) to (sockaddr *) for connect, bind and accept.

Scanned with CamScanner
Scanned with CamScanner

```
struct sockaddr_in {
    unsigned short  sin_family;  /* address family (always AF_INET) */
    unsigned short sin_port;  /* port num in network byte order */
    struct in_addr sin_addr;  /* IP addr in network byte order */
    unsigned char  sin_zero[8];  /* pad to sizeof (struct sockaddr) */
};

struct in_addr {
        unsigned long s_addr;     // this holds IPv4 address
};
```

↳ POSIX requires only three members in the structure: sin_family, sin_addr, and sin_port. Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size.

↳ The in_addr_t datatype must be an unsigned integer type of at least 32 bits, in_port_t must be an unsigned integer type of at least 16 bits, and sa_family_t can be any unsigned integer type. The latter is normally an 8-bit unsigned integer if the implementation supports the length field, or an unsigned 16-bit integer if the length field is not supported.

↳ Both the IPv4 address and the TCP or UDP port number are always stored in the structure in network byte order.

↳ The sin_zero member is unused. By convention, we always set the entire structure to 0 before filling it in.

Scanned with CamScanner
Scanned with CamScanner

↳ Socket address structures are used only on a given host. The structure itself is not communicated between different hosts.

IPv6 Socket Address structure:

```
struct sockaddr_in6 {
    u_int16_t       sin6_family;    // AF_INET6
    u_int16_t       sin6_port;      // this holds port number
    u_int32_t       sin6_flowinfo;  // this holds IPv6 flow informatic
    struct in6_addr sin6_addr;      // used to hold IPv6 address
    u_int32_t       sin6_scope_id;  // scope id
};
struct in6_addr {
    unsigned char s6_addr[16];  // this holds IPv6 address
};
```

↳ The IPv6 family is AF_INET6, whereas the IPv4 family is AF_INET

↳ The members in this structure are ordered so that if the sockaddr_in structure is 64-bit aligned, so is the 128-bit sin6_addr member.

↳ The sin6_flowinfo member is divided into two fields:

- The low-order 20 bits are the flow label.
- The high-order 12 bits are reserved.

↳ The sin6_scope_id identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address.

Scanned with CamScanner
Scanned with CamScanner

5. What are the ways to pass the length of socket structure for different socket API's argument? Explain them in detail with function prototype and argument detail.

=) ↳ Most socket functions require a pointer to a socket address structure as an argument.

↳ Generic socket address structure to hold socket information. It is passed in most of the socket function calls.

↳ Unix socket programming provides IP version specific socket address structures.

Generic Socket address:

- For address arguments to connect, bind and accept.

```
struct ~~at~~ sockaddr {
    unsigned short sa_family; /* protocol family */
    char sa_data [14]; /* address data */
};
```

Internet-specific socket address (IPv4 socket address structure):

- Must cast (sock addr_in *) to (sock addr *) for connect, bind and accept.

```
struct sockaddr_in {
    unsigned short sin_family; /*address family
                                (always AF_INET */

    unsigned short sin_port; /*port num in network byte
                              order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero [8]; /*pad to size of (struct sockaddr) */
};
```

Scanned with CamScanner
Scanned with CamScanner

```
struct in_addr {
        unsigned long s_addr; // this holds IPv4 address
};
```

IPv6 Socket Address Structure

```
struct sockaddr_in6 {
    u_int16_t     sin6_family ; // AF_INET 6.
    u_int16_t     sin6_port; // this holds port number.
    u_int32_t     sin6_flow info; // this holds IPv6 flow
                                      information.
    struct in6_addr   sin6_addr; // used to hold IPv6 address.
    u_int32_t     sin6_scope_id ; // scope id
};
  struct in6_addr {
        unsigned char s6_addr [16]; // this holds IPv6 address
```

- When a socket address structure is passed to any socket fun^n, it is always passed by reference. That is, a pointer to the structure is passed.
- The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed.

- from process to kernel

bind(), connect(), and sendto() functions pass a socket address structure from process to the kernel. Arguments to these functions:-

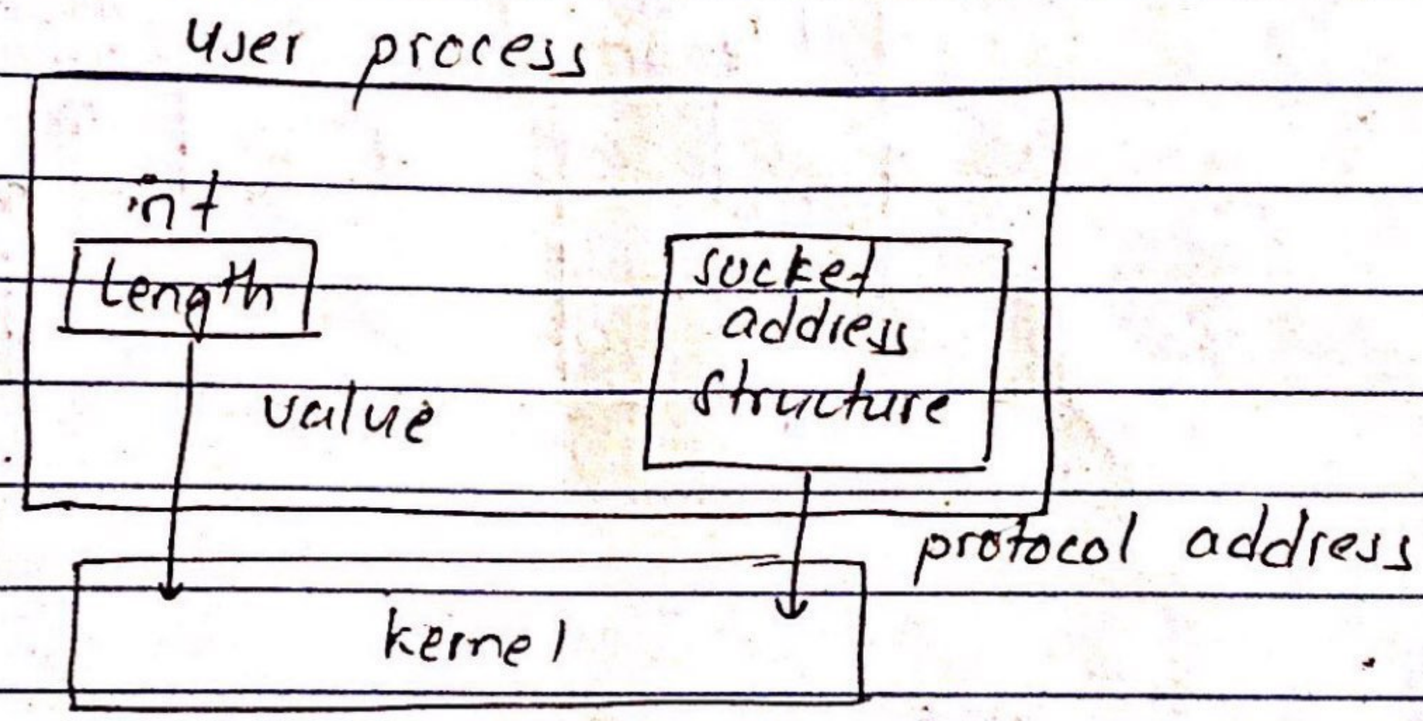
the pointer to the socket address structure

Scanned with CamScanner
Scanned with CamScanner

the integer size of the structure.

```
struct sockaddr_in serv.   /* fill in serv */
connect (sockfd, (SA *) &serv, sizeof (serv));
```

the datatype for the size of a socket address structure is actually socklen_t and not int, but the POSIX specification recommends that socklen_t be defined as uint32_t.



<u>from kernel to process:</u>

accept(), recvfrom(), getsockname(), and getpeername() functions pass a socket address structure from the kernel to the process.

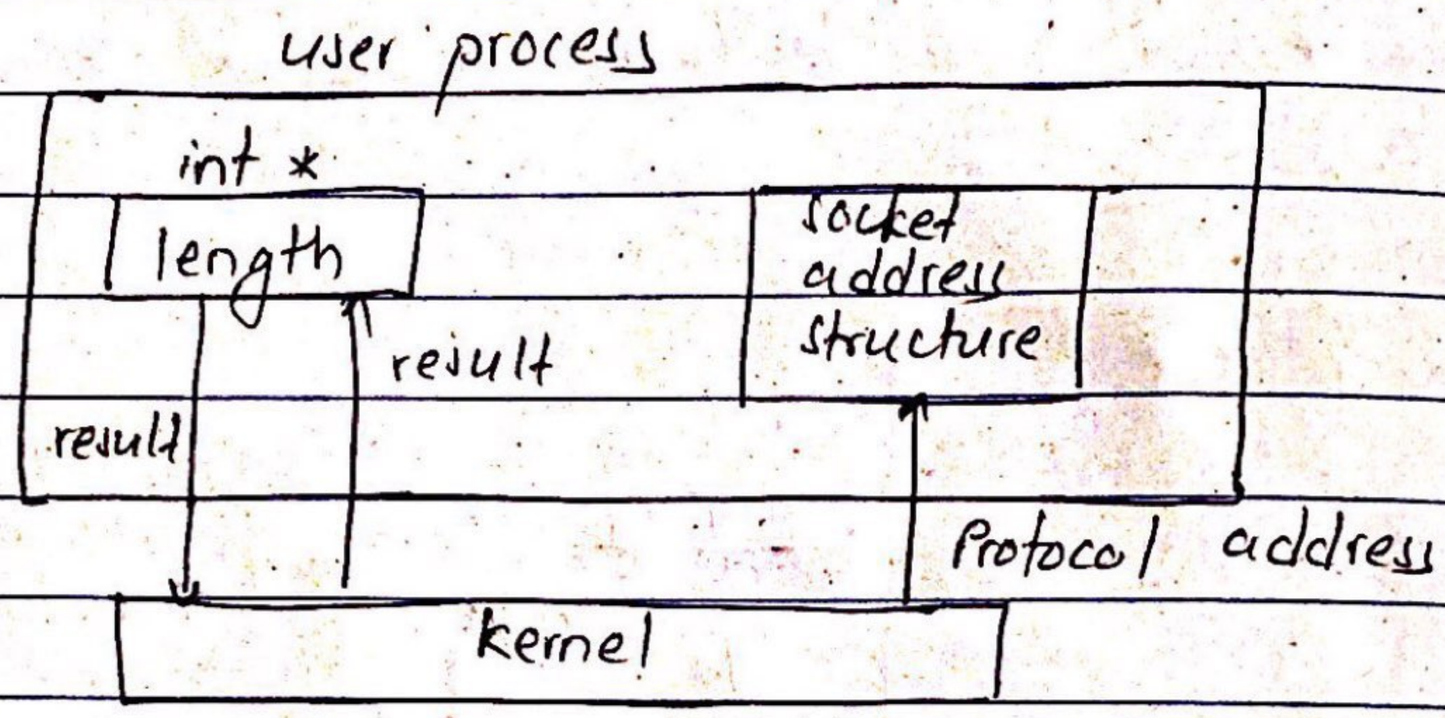
<u>Arguments to these functions</u>

The pointer to the socket address structure. The pointer to an integer containing the size of the structure.

```
struct sockaddr_un cli;   /* Unix domain */
socklen_t len;
len = sizeof (cli);          /* len is a value */
getpeername (unixfd, (SA *) &cli, &len);
                             /* len may have changed */
```

Scanned with CamScanner
Scanned with CamScanner

- Value-only = bind, connect, sendto (from process to kernel)
- Value-Result = accept, recvfrom, getsockname, getpeername (from kernel to process, pass a pointer to an integer containing size)
  - ↳ tells process how much information kernel actually stored



Q. Explain the statement : fork() is called once and it returns twice. What is the significance of calling exec() after fork() system call? What will happen when close() is called in TCP socket during the implementation of concurrent server?

:) A call to fork() will create a completely separate sub-process which will be exactly the same as the parent.

→ The process that initiates the call to fork is called the parent process.

→ The new process created by fork is called child process.

→ fork() system call return an integer to both the parent and child processes.

→ -1 indicates an error with no child process created.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

⇒ A value of zero indicates that the child process code is being executed.

⇒ positive integers indicates the child's process identifier (PID) and the code being executed is in the parent process.

```
if (pid = fork () == 0)
        printf ("I am child \n");
        else
        printf ("I am parent \n");
```

4) Significance of calling exec() after fork() system call is to terminate the currently running program and start executing a new one which is specified in the parameters of exec in the context of the existing process.

4) When close() is called on a TCP socket, it causes a FIN to be sent, followed by the normal TCP connection Termination Sequence. However, the close of connfd by the parent (in the outline) doesnot terminate its connection with the client. This is because every file or socket has a reference count.

⇒ outline of typical concurrent server.

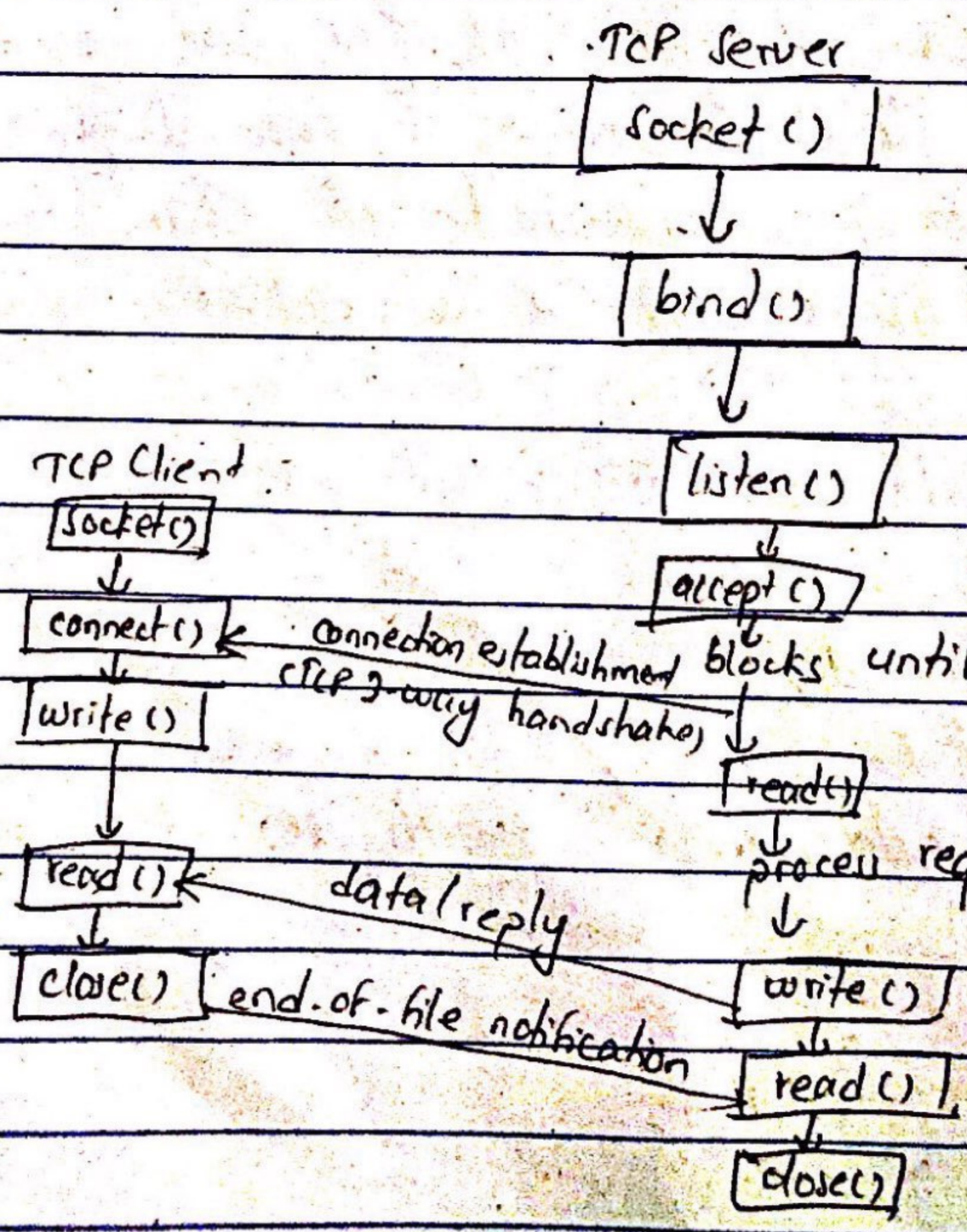
a) pid_t pid;
b) int listenfd, connfd;
c) listenfd = socket (---);
    /* fill in sockaddr_in {} with servers well-known port */
d) Bind (listenfd, ---);
e) listen (listenfd, LISTENQ);
f) for (;;) {

Scanned with CamScanner
Scanned with CamScanner

g) connfd = Accept (listenfd, ---); // probably blocks
h) if (pid = Fork () == 0) {
i) close (listenfd); // child closes listening socket
j) doit (connfd); // process the request
k) close (connfd); // done with the client
l) exit (0); // child terminates
}
m) close (connfd);

7. What is socket? Explain different system call in specific order, require to create a TCP client and TCP server in the Unix System. (Hint: Explain each function with prototypes, return types and sample code).

⇒ . Socket is an abstraction that is provided to an application programmer to send or receive data to another process.



Scanned with CamScanner
Scanned with CamScanner

i) Socket()

→ to perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc)

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

Returns: non-negative descriptor if ok, -1 on error.

family is one of

- AF_INET (IPv4), AF_INET6 (IPv6), AF_LOCAL (local Unix)
- AF_ROUTE (access to routing tables), AF_KEY (new, for encryption)

type is one of

- SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
- SOCK_RAW (for special IP packets; PING, etc. must be root)
- SOCK_SEQPACKET (sequenced packet socket)

protocol is one of

- IPPROTO_TCP
- IPPROTO_UDP
- IPPROTO_SCTP
- protocol is 0 (used for some raw socket options)

ii) Bind()

→ Bind function assigns a local protocol address to a socket. With the internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

Scanned with CamScanner
Scanned with CamScanner

int bind (int sockfd, const struct sockaddr * my addr, socklen_t addrlen);

-> returns: 0 if ok, -1 on error.

Sockfd is socket descriptor from socket()
myaddr is a pointer to address struct with:
- port number and IP address
- if port is 0, then host will pick ephemeral port
  - not usually for server. (exception RPC port-map).
- IP address != INADDR_ANY (useless multiple nics)

addrlen is length of structure.
returns 0 if ok, -1 on error
- EADDRINUSE ("Address already in use").

iii) LISTEN()

-> Connect function is used by a TCP client to establish a connection with TCP server.

Int listen (int sockfd, int backlog);
change socket state for TCP server.
- sockfd is socket descriptor from socket().
- backlog is maximum number of incomplete connections.
  - historically 5
  - rarely above 15 on a even moderate web server.

Sockets default to active (for a client)
- Change to passive so OS will accept connection.

-> An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of TCP 3-way handshake.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

→ A complete connection queue which contains an entry for each client with whom the TCP three-way handshake has completed.

- In term of the TCP state transition diagram, the call to the listen moves the socket from CLOSED state to LISTEN state.

iv) CONNECT ()

→ int connect (int sockfd, const struct sockaddr * servaddr, socklen_t addrlen);

Connect to Server.

→ sockfd is socket descriptor from socket ().

→ Servaddr is a pointer to a structure with:

- port number and ip address
- must be specified (unlike bind())

→ addrlen is length of structure.

→ returns socket descriptor if ok, -1 on error.

→ the client does not have to call bind before calling connect, the kernel will choose both an ephemeral port and the source IP address if necessary.

→ ETIMEDOUT : host doesn't exist (connect time out)

→ ECONNREFUSED : no process is waiting for connections on the server host at the port specified.

→ EHOSTUNREACH : no route to host.

v) ACCEPT() :

→ int accept (int sockfd, struct sock_addr * cli addr, socklen_t * addrlen);

Return next completed connection.

Scanned with CamScanner
Scanned with CamScanner

-> sockfd is socket descriptor from socket()
-> cliaddr and addrlen return protocol address from client.
-> returns brand new descriptor, created by the kernel. This new descriptor refers to the TCP connection with the client.
-> The listening socket is first argument (sockfd) to accept (the descriptor created by socket and used as the first argument to both bind and listen).

-> the connected socket is the return value from accept the connected socket.
-> A given server normally creates only one listening socket, which then exists for the lifetime of the server.
-> the kernel creates one connected socket for each client connection that is
-> when a server is finished serving a given client, the connected socket is closed.

vi) SENDING AND RECEIVING:-

-> int recv (int sockfd, void * buff, size_t mbytes, int flags)
int send (int sockfd, void * buff, size_t mbytes, int flags);

Same as read() and write() but for flags.
- MSG_DONTWAIT (this send non-blocking)
- MSG_OOB (out of band data, 1 byte sent ahead)
- MSG_PEEK (look, but don't remove)
- MSG_WAITALL (don't give me less than max)
- MSG_DONTROUTE (bypass routing table)

Scanned with CamScanner
Scanned with CamScanner

Date
Page

vii) CLOSE()

→ int close (int sockfd);
close socket for use.
sockfd is socket descriptor from socket()
closes socket for reading/writing.
- return (doesn't block)
- attempts to send any unsent data
- socket option SO_LINGER.
  - block until data sent
  - or discard any remaining data
- returns -1 if error.
- the default action of close with a TCP socket is to mark the socket as closed and return to the process immediately.
- The socket descriptor is no longer usuable by the process. It cannot be used as an argument to read or write.

Sample Code

```
// daytime tcp server
#include <stdio.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <std_lib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
```

Scanned with CamScanner
Scanned with CamScanner

```
# define MAXLINE 1024
# define LISTENQ 10
typedef struct sockaddr SA;
    int main (int argc, char ** argv)
{
        int listenfd, connfd;
        struct sockaddr_in servaddr, cliaddr;
        char buff [MAXLINE];
        time_t ticks;
        int port;
        socklen_t len;
      listenfd = socket (AF_INET, SOCK_STREAM, 0);
        port = atoi (argv [1]);
        bzero (&servaddr, sizeof (servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
        servaddr.sin_port = htons (port); /* daytime server */

        bind (listenfd, (SA *) &servaddr, sizeof (servaddr));
        printf ("Server is waiting connection at port %d\n");
        listen (listenfd, LISTENQ);
        for (;;) {
        //connfd = accept (listenfd, (SA *) NULL, NULL);
        //printf ("Client connected! \n");
        len = sizeof (cliaddr);
        connfd = accept (listenfd, (SA *) &cliaddr, &len);
        printf ("connection from %s, port %d\n",
      inet_ntop (AF_INET, &cliaddr.sin_addr, buff, sizeof (buff)),
        ntohs (cliaddr.sin_port));
```

Scanned with CamScanner
Scanned with CamScanner

```
ticks = time (NULL);
snprintf (buff, sizeof(buff), "%.24s\r\n", ctime (&ticks));
write (connfd, buff, strlen (buff));
        close (connfd);
    }
}

// daytime tcp client
int main (int argc, char ** argv)
{
    int sockfd, n;
    char recvline [MAXLINE + 1];
    struct sockaddr_in servaddr;
    int port;
    if (argc != 3)
        printf ("usage : a.out <IP address> <port>");

    if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
        printf ("socket error");
    port = atoi (argv [2]);
    bzero (&servaddr, sizeof (servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons (port); /*daytime server*/
    if (inet_pton (AF_INET, argv[1], &servaddr.sin_addr) <= 0)
        printf ("inet_pton error for %s", argv[1]);

    if (connect (sockfd, (SA *) &servaddr, sizeof (servaddr)) < 0)
        printf ("connect error");
```

Scanned with CamScanner
Scanned with CamScanner

```
while ((n= read (sockfd, recvline, MAXLINE)) > 0) {
        recvline[n]= 0;   /* null terminate */
        printf ("Server Time: ");
        if (fputs (recvline, stdout) == EOF).
        printf ("Fputs error");
}

if (n<0)
        printf ("read error");
exit (0);
}
```

8. What will happen if you call bind() in tcp client program? Explain the situation with relevant example code where you can use send() and recv() in UDP socket and sendto() and recvfrom() in tcp socket.

=> If we call bind() in tcp client program then the client domain will be limited i.e. it will connect to specific port of the server only.

-> We can ~~see~~ use send() and recv() in UDP socket by following:

i) <u>In server side</u>

=> if socket descriptor is bind to particular IP and port for e.g.

```
int sockfd, struct sockaddr_in address;
sockfd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
memset (&address, 0, sizeof (address));
address.sin.family = AF_INET.
address.sin_port = htons (PORT);
address.sin_addr.s_addr = htonl (INADDR_ANY);
bind (sockfd, (struct sockaddr *) &address, sizeof address);
```

Scanned with CamScanner
Scanned with CamScanner

Date
Page

* send (new_sockfd, buff, size of (buff));
* recv (new_sockfd, buff, size of (buff));

ii) In client side

→ if connect() is used in UDP socket we can use send() for sending and recv() for receiving.

for eg:

* sockfd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
* memset (&address, 0, sizeof (address));
* address.sin_family = AF_INET;
* address.sin_port = htons (PORT);
* address.sin_addr.s_addr = inet_addr (IPADDRESS);
* Connect (sockfd, (struct sockaddr *) & address, sizeof (address));
* send (sockfd, buff, size of (buff));
* recv (sockfd, buff, sizeof (buff));

→ We can use sendto() and recvfrom() in TCP socket by:

i) In server side

→ if we set NULL in argument of sendto and recvfrom it behaves like send and recv respectively.

for e.g:

* sockfd = SOCKET (AF_INET, SOCK_STREAM, IPPROTO_TCP);
* memset (&address, 0, size of (address));
* address.sin_family = AF_INET.
  address.sin_port = htons (PORT);
  address.sin_addr.s_addr = htonl (INADDR_ANY);
  bind (sockfd (struct sockaddr *) & address, size of (address));
  Listen (sockfd, BACKLOG);

Scanned with CamScanner
Scanned with CamScanner

```
while(1)?
        socklen_t addrlen = sizeof (client addr);
        newsockfd = accept (sockfd (struct sockaddr *)
        & clientaddr, & addrlen);
```

ii) <u>In Client side</u>

→ we can use sendto and recvfrom same as send and recv repectively, if we set NULL in the argument.

<u>Eg:</u>

```
sockfd = SOCKET (AF_INET, SOCK_STREAM, IPPROTO_TCP);
memset (&address, 0, sizeof (address));
address.sin_family = AF_INET.
address.sin_port = htons (PORT).
address.sin_addr.s_addr = inet_addr (IPADDRESS).
connect (sockfd, (struct sockaddr *) & address, sizeof (address)
sendto (sockfd, buff, sizeof (buff), 0, NULL, NULL);
recvfrom (sockfd, buff, sizeof (buff); 0, NULL, NULL);
```

Scanned with CamScanner
Scanned with CamScanner

Date
Page

9. Does getaddrinfo() is blocking or non-blocking? List out the different system call used to create TCP and UDP client server on the basis of blocking and non-blocking in nature. Also explain them in very short.

⇒ ↳ getaddrinfo() is a blocking call and there is no way to specify a timeout to make it return early, so it may block indefinitely.

Here are some system calls commonly used to create a TCP or UDP client or server, based on whether they are blocking or non-blocking in nature:

Blocking System Calls:

- Socket(): creates a new socket.
- bind(): binds a socket to a specific IP address and port
- listen(): sets a socket to listen for incoming connections

- accept(): accepts an incoming connection and creates a new socket for communication.

- read(): reads data from a socket
- write(): writes data to a socket
- close(): closes a socket

Non-blocking system calls:

- Fcntl(): sets a socket to be non-blocking.

- select(): waits for a socket to become ready for reading or writing, but doesn't block if the socket is not ready.

Scanned with CamScanner
Scanned with CamScanner

- poll(): similar to select(), but can handle multiple sockets at once.

- epoll(): similar to poll(), but more efficient and can handle a larger number of sockets

- sendto/recvfrom(): are used to send and receive data on a UDP socket.

11. What is the purpose of bind() function? What will be the outcomes if we donot specify IP address, port, both or neither during bind() system call?

⇒ The bind function assigns a local protocol address to a socket.

i) Calling bind lets us specify the IP address, the port, both or neither. The following table summarizes the values to which we set sin_addr and sin_port, or sin6_addr and sin6_port depending on the desired result.

IP address	Port	
* wildcard	0	kernel chooses IP address & port
* wildcard	non-zero	kernel chooses IP address, process specifies port.
* local IP address	0	process specifies IP address, kernel chooses port.
* local IP address	non-zero	process specifies IP address and port.

ii) If we specify a port number of 0, the kernel chooses an ephemeral port when bind is called.

Scanned with CamScanner
Scanned with CamScanner

iii) If we specify a wildcard IP address, the kernel doesnot choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

iv) With IPv4, the wildcard address is specified by the constant INADDR_ANY, whose value is normally 0. This tells the kernel to choose the IP address.

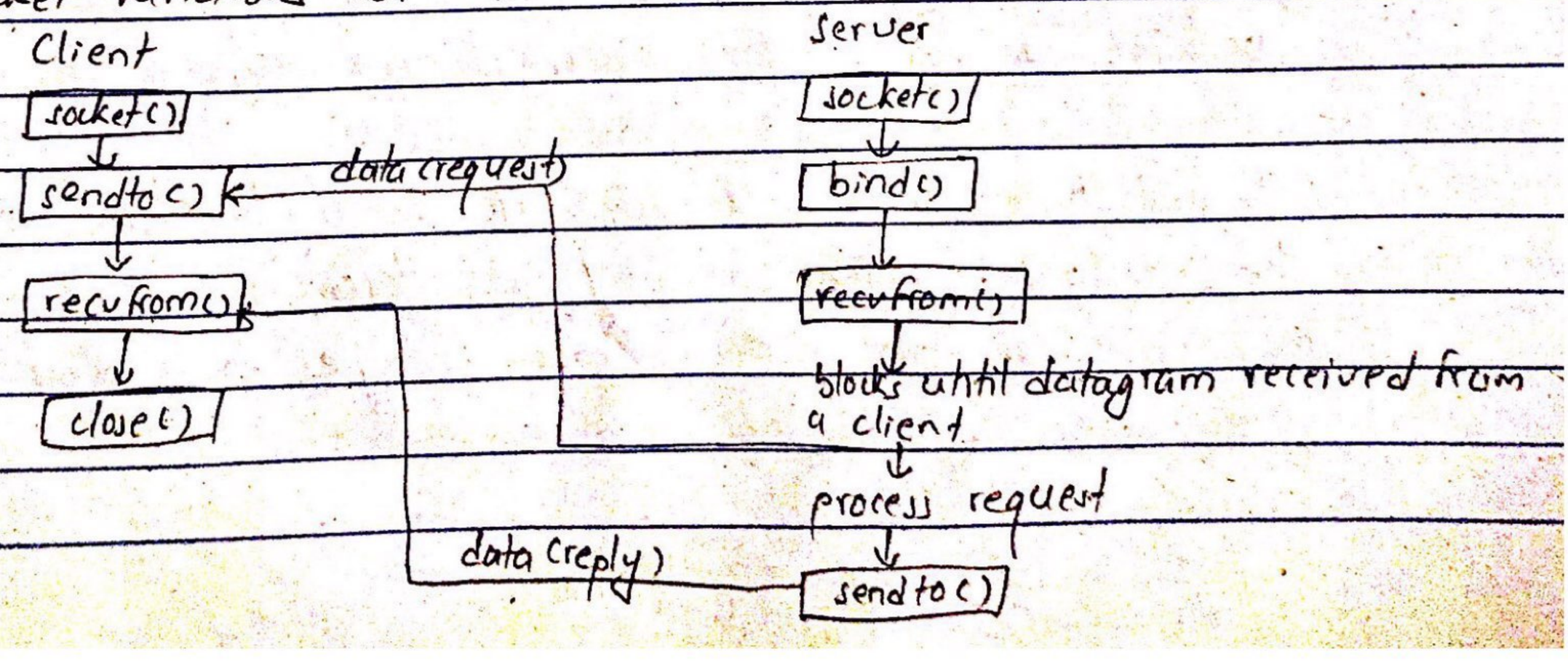
```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
```

12. What is completed connection queue and incomplete connection queue ()? Explain different system call in specific order, required to create a UDP client and UDP server in the Unix System.

=> A complete connection queue, which contains an entry for each client with whom the TCP three way handshake has completed. This sockets are in the ESTABLISHED state.

> An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state.

> Socket functions for UDP client/server:



Scanned with CamScanner
Scanned with CamScanner

i) Socket() : → To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, etc).

- int socket (int AF_INET, int SOCK_DGRAM, 0)
- Returns : non-negative descriptor if OK; -1 on error.

ii) BIND() :

→ the bind function assign a local protocol address to a socket with port number.

→ int bind (int sockfd, const struct sockaddr * myaddr, socklen_t addrlen);

→ Returns : 0 if ok; -1 on error.

iii) RECVFROM() and SENDTO() function

→ # include <sys/socket.h>

ssize_t recvfrom (int sockfd, void * buff, size_t nbytes, int flags, struct sockaddr * from, socklen_t * addrlen);

ssize_t sendto (int sockfd, const void * buff, size_t nbytes, int flags, const struct sockaddr * to, socklen_t addrlen);

// Both return : number of bytes read or written if OK, -1 on error.

- the first three arguments, sockfd, buff and nbytes are identical to the first three arguments for read and descriptor, pointer to buffer to read into a write from and number of bytes to read or write.

Scanned with CamScanner
Scanned with CamScanner

- the to argument for sendto is a socket address structure containing the protocol address (eg: IP address and port number) of where the data is to be sent.

- the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value_result argument).

iv) Close():

→ close socket for use

int close (int sockfd);

→ closes socket for reading / writing.

13. Can we use connect() system call while developing UDP socket? If yes, what will be the outcomes of using connect() in UDP socket? If no, how communicating processes know each other? Explain them in detail.

⇒ Yes, we can use connect() system call while developing UDP socket.

While calling connect for a UDP socket the kernel just checks for any immediate errors (eg: an obviously unreachable destination) records the IP address and port number of the peer and returns immediately to the calling process. Obviously, there is no 3-way handshake with this capability, it can distinguish between

i) An unconnected UDP socket, the default when we create a UDP socket.

ii) A connected UDP socket, the result of calling connect on a UDP socket.

Scanned with CamScanner
Scanned with CamScanner

→ With a connected UDP socket, three things change compared to the default unconnected UDP sockets:

i) we can no longer specify the destination IP address and port for an output operation. we don't use 'send to' but 'write' or 'send' instead. Anything written to a connected UDP socket is automatically sent to the protocol address (eg: IP address and port) specified by connect.

ii) We donot need to use recv from to learn the sender of a datagram, but read, recv or recvmsg instead. The only datagrams returned by the kernel for an i/p operation on a connected UDP socket are those arriving from the protocol address specified in connect.

iii) Asynchronous errors are returned to the process for connected UDP sockets.

→ figure given below summarizes point:

types of socket	write or send	send to that doesnot specify a destination	send to that specifies a destination
TCP socket	OK	OK	EISCONN
UDP socket	OK	OK	EISCONN
Connected	OK	OK	EISCONN
UDP Socket	EDESTA	EDESTADDR	OK
Unconnected	DDRREQ	REQ	OK

table: TCP and UDP sockets can be destination protocol address be specified

Scanned with CamScanner
Scanned with CamScanner

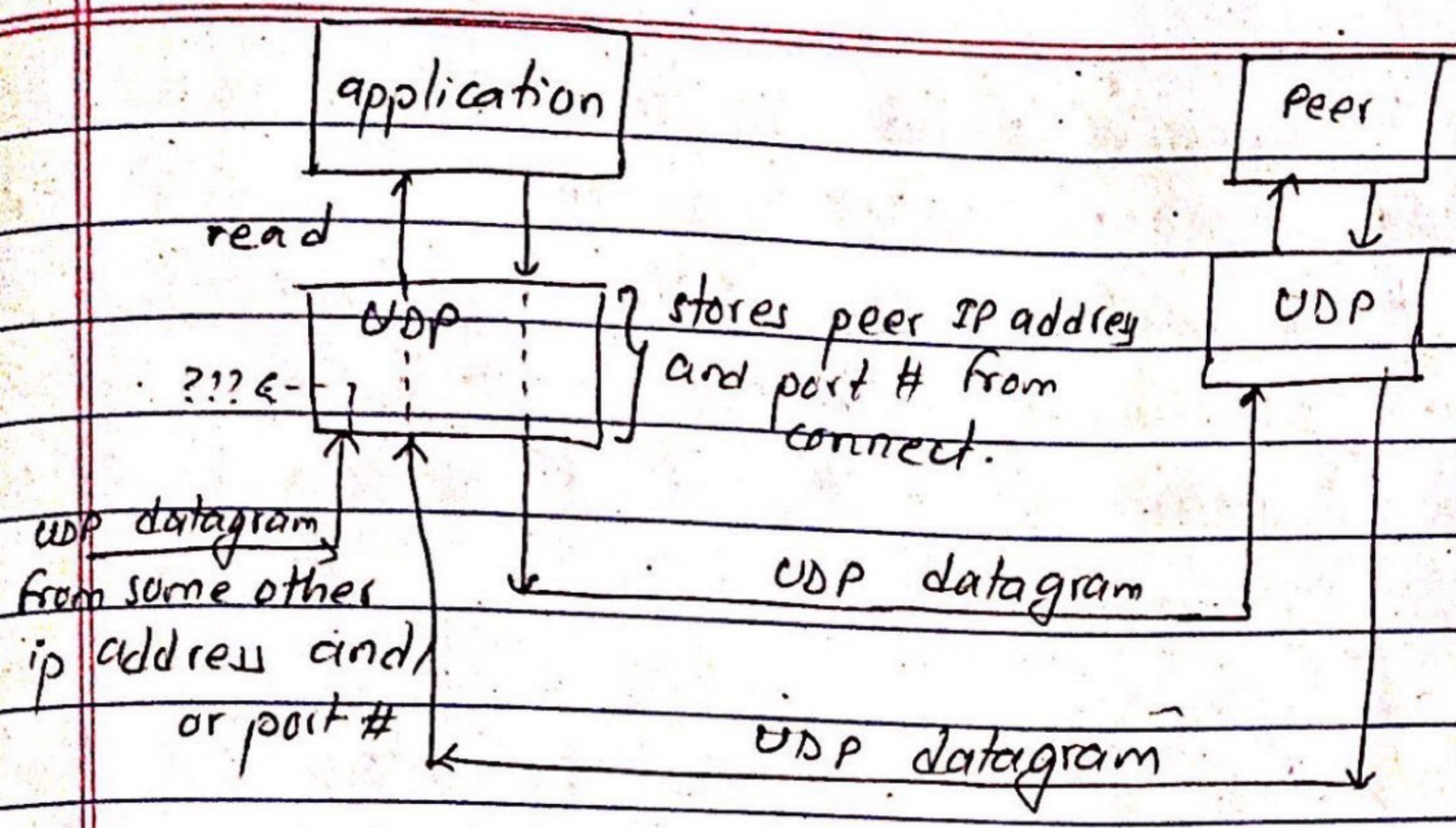


fig: Connected UDP Socket

→ In summary, UDP client or server can call 'connect' only if that process uses the UDP socket to communicate with exactly one peer. Normally, it is a UDP client that calls connect, but there application in which the UDP server communicates with a single client for a long duration (eg: TFTP) in this case, both the client and server can call connect.

14. The connect() function returns only when the connection is established or an error occurs, what are those errors and when these errors ~~occupied~~ occured? Explain in detail.

→ int connect (int sockfd, const struct sockaddr * servaddr, socklen_t addrlen);

the connect() returns only when the connection is established or an error occurs. Those possible errors are:-

i) If the client TCP receives no response to its SYN segment, ETIMEOUT is returned.

→ Connect moves from the CLOSED state (the state in which a socket begins when it is created by the socket function) to the SYN_SENT state, and then, on success, the ESTABLISHED state.

Scanned with CamScanner
Scanned with CamScanner

ii) If the server's response to the clients, SYN is a reset (RST), th indicates that no process is waiting for connections on the server, host at the port specified (the server process is probably not running). This is a hard error and the error ECONNREFUSED is returned to the client as soon as the RST is received.

iii) If the clients SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a soft error. The client kernel saves the messages but keeps sending SYN with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time 17.5 seconds for 4.4 BSD the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH.

15. What is listening socket and connected socket? Explain with function prototype and sample code: when and why getsockname() and getpeername() required?

=) Listening socket is the first argument (sockfd) to accept (the descriptor create by socket and used as the first argument to both bind and listen).

int accept (int sockfd, struct sockaddr * cliaddr, socklen_t * addrlen);

-> the connected socket is the return value from accept the connected socket. accept() returns up to three values:-

- an integer return code that is either a new socket descriptor or an error indication.
- protocol address of the client process (through the cliaddr pointer).

Scanned with CamScanner
Scanned with CamScanner

- size of this address (through the addrlen pointer).

```
#include <sys/socket.h>
    int getsockname (int sockfd, struct sockaddr. *local addr,
                     socklen_t *addrlen).
int getpeername (int sockfd, struct sockaddr *peeraddr,
                 socklen_t *addrlen)
```

getsockname() and getpeername() is required due to following reasons:

- After connect successfully returns in a TCP client that doesnot call bind, getsockname() returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number), get sockname returns the local port number that was assigned.

- get sockname can be called to obtain the address family of a socket.
- In a TCP server, that binds the wildcard. IP address once a connection is established with a client, the server can call getsockname to obtain the local IP address assigned to the connection.

- when the server is exceed by the process that calls accept, the only way the server can obtain the identify of the client is to call getpeername:

example :- obtaining the address family of a socket.

the sockfd_to_family function show in below returns the address family of a socket.

Scanned with CamScanner
Scanned with CamScanner

lib/sockfd_to_family.c

```
#include "Unp.h"
int sockfd_to_family (int sockfd)
{
    struct sockaddr_storage ss;
    socklen_t len;
    len = sizeof (ss);
    if (getsockname. (sockfd, (SA*) &ss, &(len) < 0 )
        return (-1);
        return (ss.ss_family);
}
```

16. Explain the functions send(), sendto(), recv() and recvfrom(). What are the major differences between wait() and waitpid()? Explain the possible option values that we can supply in waitpid() system call.

⇒ i) <u>Send()</u> : send function is used to send data over stream sockets or connected datagram sockets. If we want to send data over UNCONNECTED datagram sockets, we must use sendto() function.

```
int send (int sockfd, void * buff, size_t mbytes, int flags);
```

ii) <u>recv()</u> : is used to receive data over stream sockets or CONNECTED datagram sockets. recv() receive data on a socket with descriptor socket and stores it in a buffer.

```
int recv (int sockfd, void * buff, size_t mbytes, int flags);
```

Scanned with CamScanner
Scanned with CamScanner

iii) sendto() : is use to send data over UNCONNECTED datagram sockets

ssize_t sendto (int sockfd, const void * buff, size_t nbytes, int flags, const struct sockaddr * to, socklen_t addrlen);

iv) recvfrom() : is use to receive data from unconnected datagram sockets.

ssize_t recvfrom (int sockfd, void * buff, size_t nbytes, int flags, struct sockaddr *from, socklen_t * addrlen);

→ Either of wait and waitpid can be used to remove zombies. wait (and waitpid in its blocking form) temporarily suspends the execution of a parent process while a child process is running.
- Once the child has finished, the waiting parent is restarted.

	wait()	waitpid()
i)	wait blocks the caller until a child process terminates.	i) waitpid can be either blocking or non-blocking. * if option is 0, then it is blocking * if option is WNOHANG, then it is non-blocking.
ii)	If more than one child is running then wait() returns the first time of one the parents offspring exists.	ii) waitpid is more flexible according to different values of pid.

Scanned with CamScanner
Scanned with CamScanner

⇒ the possible option values that we can supply in waitpid() system call are;

i) if pid == -1, it waits for any child process in this respect waitpid is equivalent to wait.

ii) if pid > 0, it waits for the child whose process ID equals pid.

iii) if pid == 0, it waits for any child whose process group ID equals that of the calling process.

iv) If pid < -1 it waits for any child whose process group ID equals that absolute value of pid.

17. Differentiate unix domain socket and internet domain socket. Explain the mechanism of passing file descriptor in the Unix system.

⇒ Unix domain socket is used for same host while internet domain socket is used for different host as well as same host.

Mechanism of passing file descriptor in the Unix System.

i) Create a Unix domain socket either a stream socket or a datagram socket.

→ if the goal is to 'fork' a child and have the child open the descriptor and pass the descriptor back to the parent, the parent can call socket pair to create a stream pipe that can be used to exchange the descriptor.

ii) One process opens a descriptor by calling any of Unix functions that returns a descriptor. Any type of descriptor can be passed from one process to another.

Scanned with CamScanner
Scanned with CamScanner

iii) Sending process builds a "~~msg~~msghdr" structure containing the descriptor to be passed. The sending process calls 'sendmsg' to send the descriptor across the unix domain socket.

- At this point, the descriptor is "in flight". Even if the sending process closes the descriptor after calling sending but before the receiving process calls recvmsg. The descriptor remains open for the receiving process. Sending a descriptor increments the descriptors reference count by one.

iv) the receiving process calls '~~receiving~~ 'recvmsg' to receive the descriptor on the unix domain socket. It is normal for the descriptor number in the receiving process to differ from the descriptor number in the sending process passing a descriptor involves creating a new descriptor in the receiving process that refers to the same file table entry within the kernel as the descriptor that was sent by the sending process.

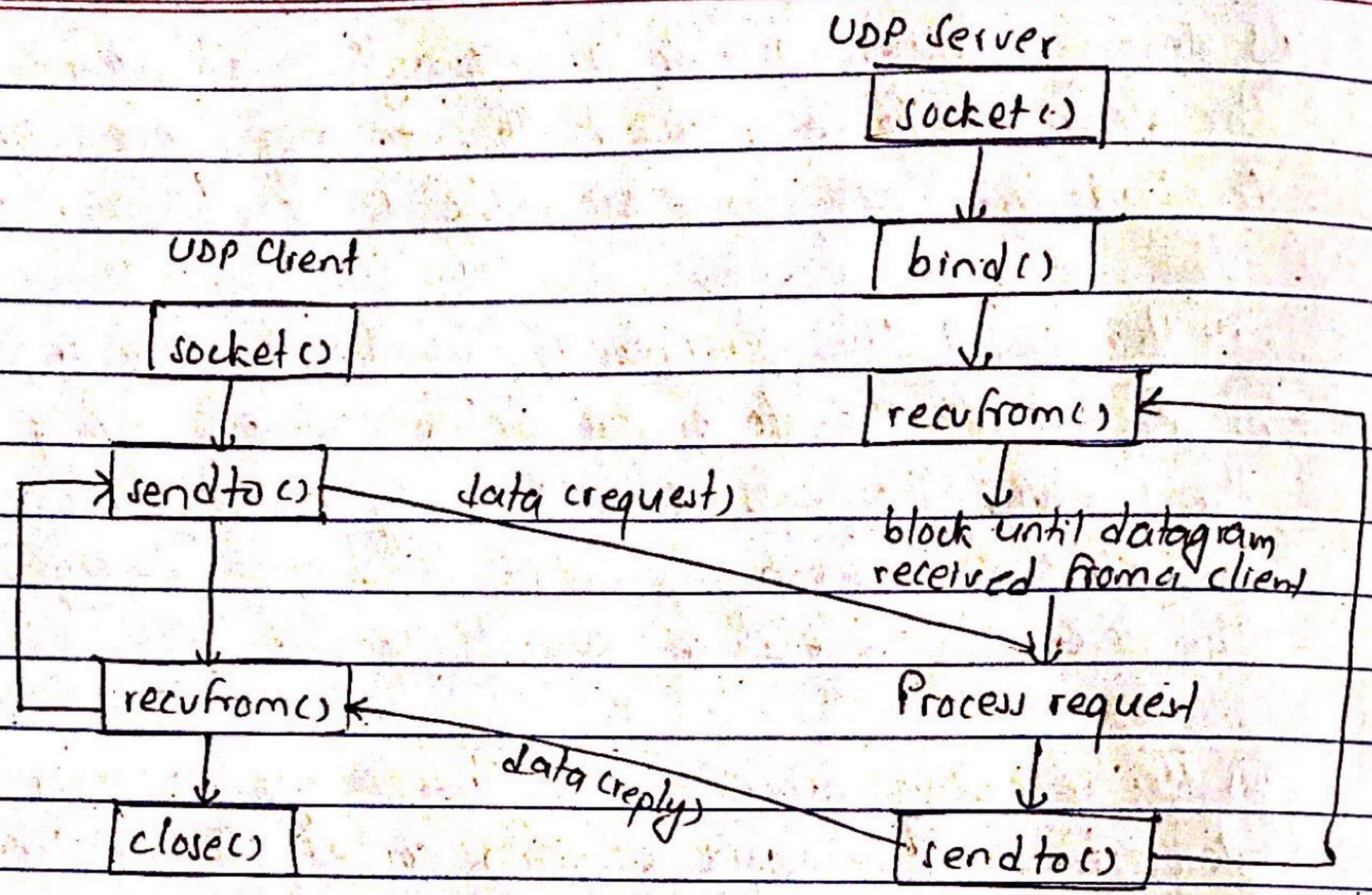
18. what is the role of socketpair() function? Explain different system call in specific order, required to create a UDP client and UDP server in the Unix System.

=) The socketpair() function creates two sockets that are connected together. This function applies only to Unix domain sockets.

```
# include <sys/socket.h>
int socketpair (int family, int type, int protocol, int
                sockfd [2]);
```

Returns: nonzero if OK, -1 on error.

Scanned with CamScanner
Scanned with CamScanner



i) <u>socket()</u>: To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, etc).

- int socket (int AF_INET, int SOCK_DGAM, 0)
- Returns: non-negative descriptor if ok; -1 on error.

ii) <u>bind()</u>: -> the bind function assign a local protocol address a/o a socket with port number.

> int bind (int sockfd, const struct sockaddr * myaddr, socklen_t addrlen);

-> Returns: 0 if ok, -1 on error.

iii) <u>recvfrom() and sendto() function</u>:

-> # include <sys/socket.h>

Scanned with CamScanner
Scanned with CamScanner

ssize_t recvfrom (int sockfd, void * buff, size_t nbytes, int flags, struct sockaddr * from, socklen_t * addrlen);

ssize_t sendto (int sockfd, const void * buff, size_t nbytes, int flags, const struct sockaddr * to, socklen_t addrlen);
// Both return: number of bytes read or written if ok, -1 on error.

- the first three arguments, sockfd, buff and nbytes are identical to the first three arguments for read and descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.

- the to argument for sendto is a socket address structure containing the protocol address (e.g: IP address and port number) of where the data is to be sent.

- the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).

iv) <u>close()</u>

→ close socket for use

int close (int sockfd);

→ closes socket for reading / writing.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

19. what is signal and why it is required? Explain POSIX signal handling mechanism. Also Explain few signal types.

=> Signal is a notification to a process that an event has occurred. Signals are sometimes called software interrupts. Signals usually occur synchronously.

-> Signal is required to notify a process that an event has occured..

POSIX Signal handling mechanism:

- Every signal has a disposition, which is also called the action associated with the signal we set the disposition of a signal by calling the 'sigaction' function. We have three choices for disposition.

i) we can provide a function that is called whenever a specific signal occurs. This function is called a signal handler and the action is called catching a signal. The two signals SIGKILL and SIGSTOP cannot be caught.

ii) we can ignore a signal by setting its disposition to SIG_IGN. The two signals SIGKILL and SIGSTOP cannot be ignored

iii) We can set the default disposition for a signal by setting its disposition to SIG_DFL.

There are few signals whose default disposition is to be ignored: SIGCHLD and SIGURG

Scanned with CamScanner
Scanned with CamScanner

Date
Page

→ few signal types are:

Name	Default Action	Description
i) SIGINT	terminate process	interrupt program.
ii) SIGQUIT	Create Card Image	quit program
iii) SIGILL	Create Core Image	illegal instruction
iv) SIGKILL	terminate process	kill program
v) SIGBUS	Create Core Image	bus error
vi) SIG TERM	terminate process	Software termination signal
vii) SIG URG	discard signal	urgent condition present on socket
viii) SIGSTOP	stop process	stop (cannot be caught or ignored).
ix) SIG CONT	discard signal	continues after stop.
x) SIG CHLD	discard signal	child status has changed.
xi) SIGIO	discard signal	I/O is possible on a descriptor
xii) SIG INFO	discard signal	status request from keyboard.

20. why IO and IO multiplexing is needed in socket programming? Explain various IO models in Unix system.

→ I/O is needed in socket programming because it allows bi-direction communication between client and server

→ I/O multiplexing is needed in socket programming to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e. inputs is ready to be ~~used~~ read or the descriptor is capable of taking more output). This capability is provided by the select and poll functions.

Scanned with CamScanner
Scanned with CamScanner

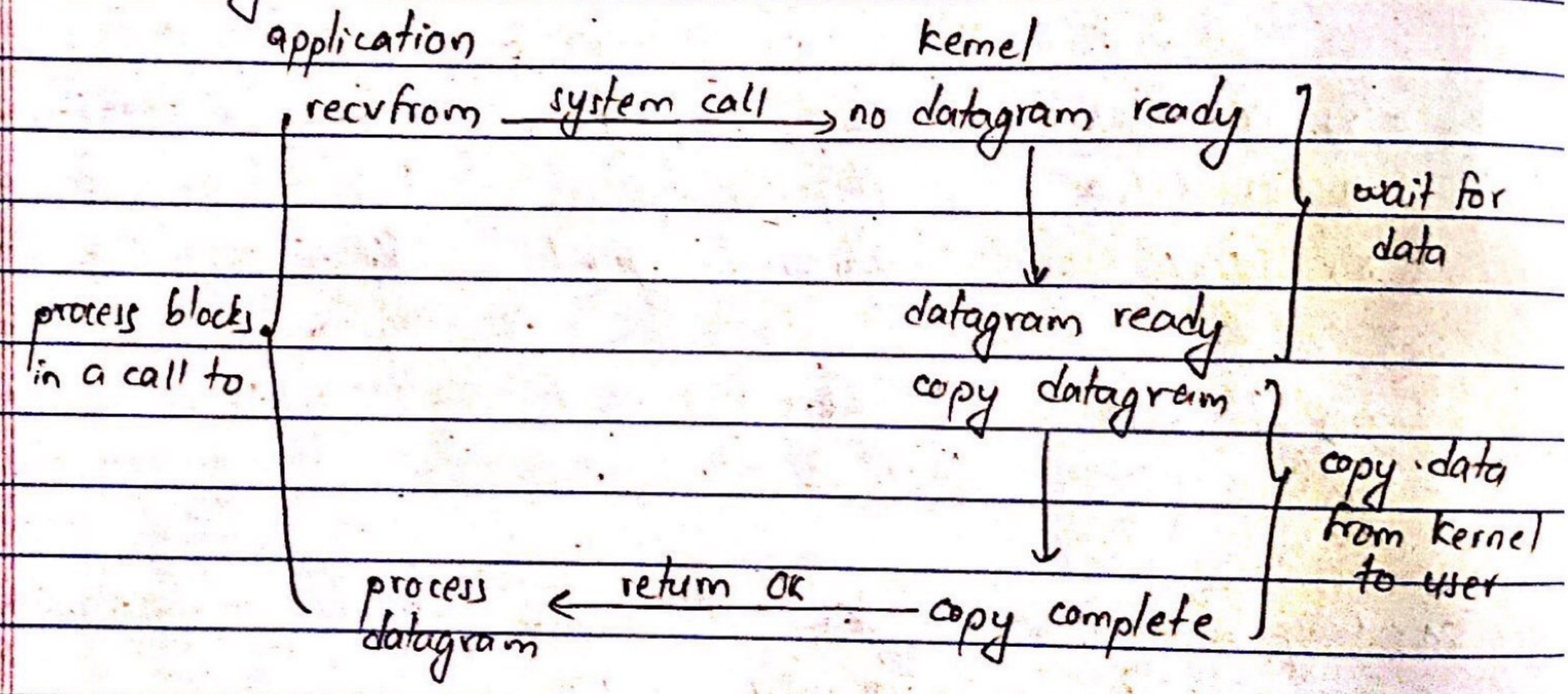
Date
Page

⇒ when a client is handling multiple descriptor I/O multiplexing is used.

⇒ Various I/O models are:

a) Blocking I/O
b) Non-blocking I/O
c) I/O multiplexing (select and poll)
d) Signal driven I/O (SIGIO)
e) Asynchronous (I/O)

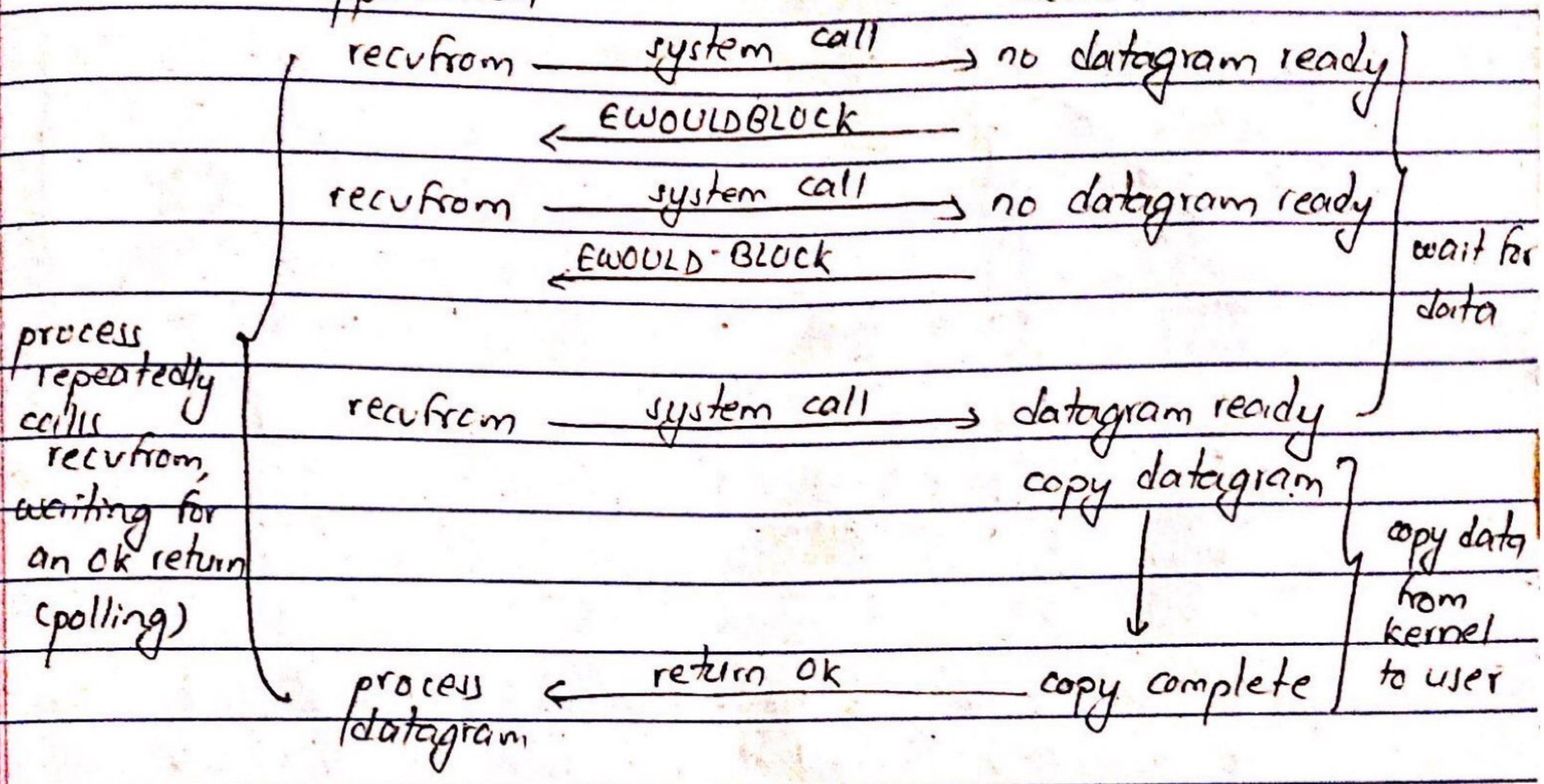
a) <u>Blocking I/O</u>



→ By default, all sockets are blocking.
→ The process calls recvfrom and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs.
→ We say that our process is blocked the entire time from when it calls recvfrom until it returns.
→ When recvfrom returns successfully, our application process the datagram.

Scanned with CamScanner
Scanned with CamScanner

b) Non-blocking I/O :



-> when a socket is non-blocking. It instruct the kernel as "when an I/O operation that the process requests cannot be completed without putting the process to sleep, donot put the process to sleep, but return an error instead."

-> The first three times that we call recvfrom, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead.

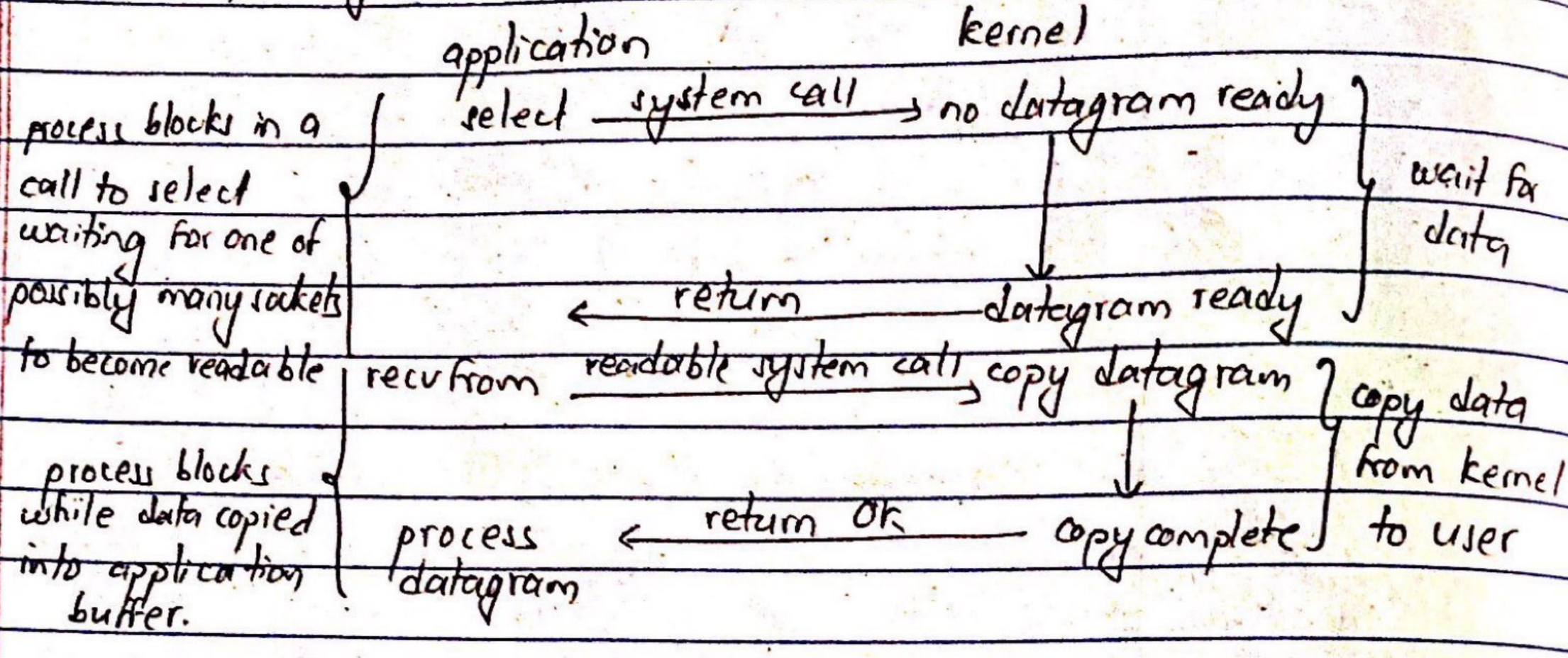
-> The fourth time we call recvfrom, a datagram is ready, it is copied into our application buffer, and recvfrom returns successfully.

-> We then process data. When an application sits in a loop calling recvfrom on a non-blocking descriptor like this, it is called polling.

-> The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time.

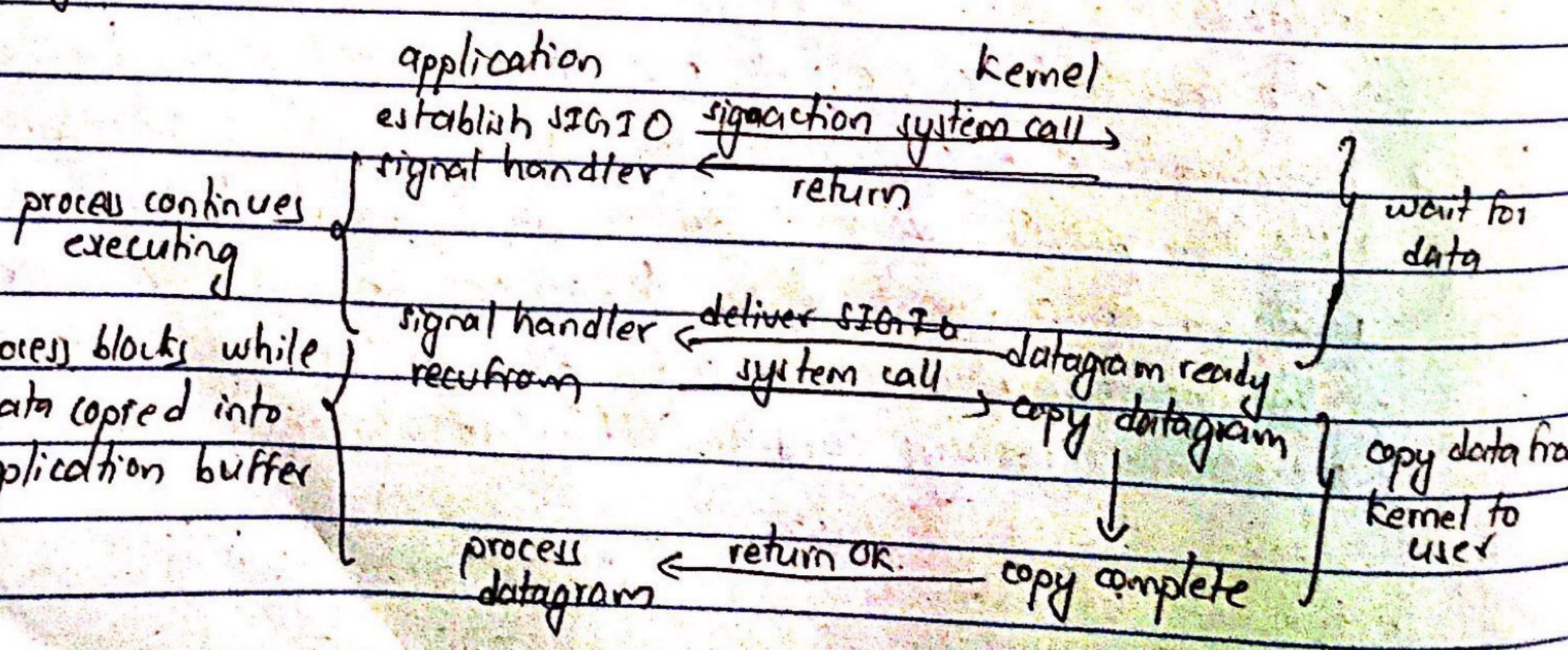
Scanned with CamScanner
Scanned with CamScanner

c) I/O Multiplexing:



→ with I/O multiplexing, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call.
→ we block in a call to select, waiting for the datagram socket to be readable.
→ when select returns that the socket is readable, we then call recvfrom to copy the datagram into our application buffer.
→ With select, we can wait for more than one descriptor to be ready.

d) Signal driven I/O:



Scanned with CamScanner
Scanned with CamScanner

Date
Page

→ A SIGIO signal is used to tell the kernel when the descriptor is ready. We call this signal-driven I/O.

→ We first enable the socket for the signal-driven I/O and install a signal handler using the sigaction system call.

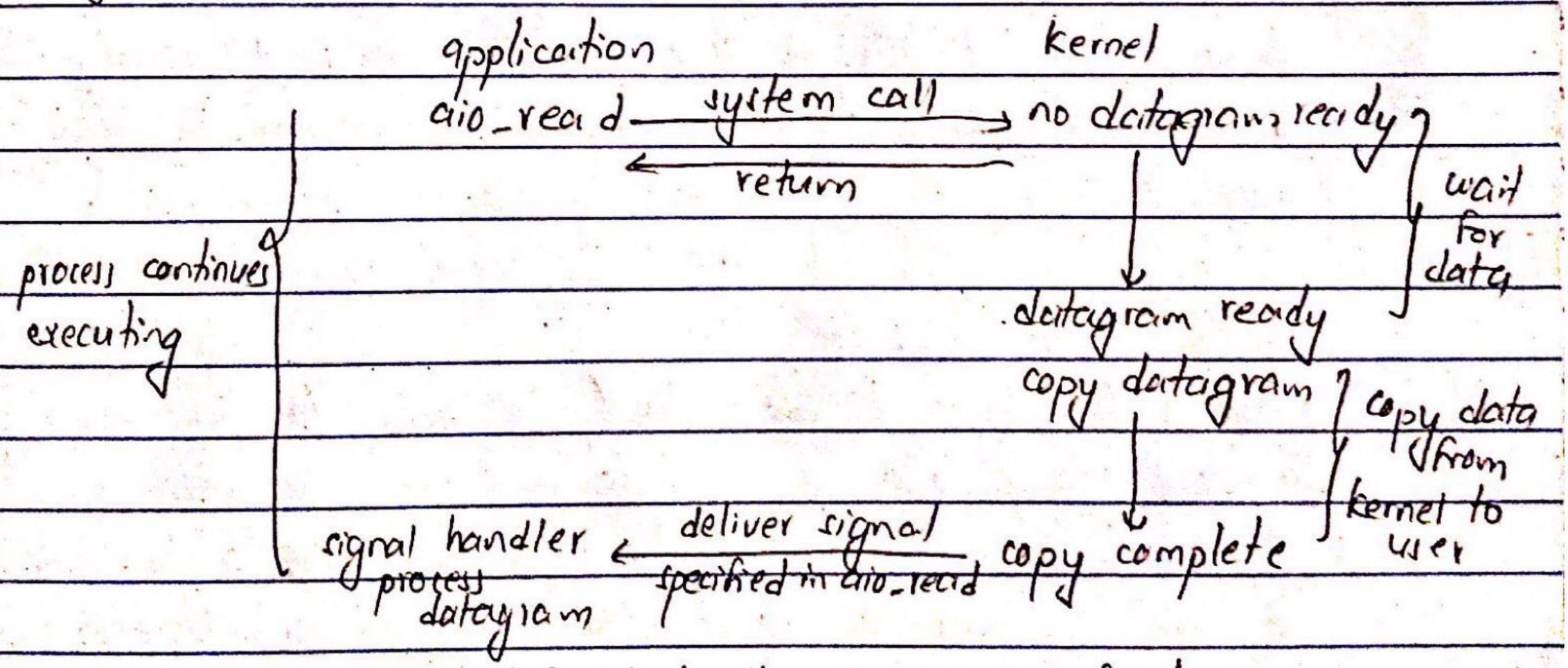
→ The return from this system call is immediate and our process continues; it is not blocked.

→ When the datagram is ready to be read, the SIGIO signal is generated for our process. We can then read the data.

→ The advantage of this model is that we arenot blocked while waiting for the datagram to arrive.

→ The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

e) Asynchronous I/O:



→ Asynchronous I/O is defined by the POSIX specification.

→ These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

→ The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/O operation is complete.

→ We call aio_read and pass the kernel the descriptor, buffer pointer, buffer size, file offset, and how to notify us when the entire operation is complete.

→ This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

21. What is I/O multiplexing? Explain the use of select function in the context of I/O multiplexing in detail. (Hint: Do not forget possibilities of wait options)

⇒ 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.

Select function

⇒ Allows the process to ~~inte~~ instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed. What descriptors we are interested in readable, writeable or exception condition) and how long to wait?

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfdp1, fd_set *readset, fd_set *writeset,
            fd_set *exceptset, const struct timeval *timeout);
```

Scanned with CamScanner
Scanned with CamScanner

/* returns : +ve count of ready descriptors, 0 on time out, -1 on error */

```
struct timeval {
        long tv_sec;   /* seconds */
        long tv_usec;  /* microseconds */
```

- the final argument timeout tells, the kernel how long to wait for one of the specified file descriptors to become ready. A time val structure specifies the number of second and microseconds.

→ possibilities for select function:-

i) <u>wait forever</u>: → return only when descriptor(s) is ready (specify timeout argument as NULL).

ii) wait upto a fixed amout of time:
→ Return when one of the specified descriptor is ready for I/O, but donot wait beyond the number of seconds and microseconds specified in the timeval structure pointed to by the timeout argument.

iii) Donot wait at all:
→ Return immediately after checking the descriptors (called polling) (specify timeout argument as pointing to a timeval structure where the time value is 0).

Scanned with CamScanner
Scanned with CamScanner

Date
Page

23. What is the basic differences between select() and poll()? Explain poll() and pselect() function.

⇒ The basic difference between select() and poll() is that select is a blocking system call, while poll() is non-blocking.

- 'select()' is used to monitor multiple file descriptors and wait until one or more of them become ready for I/O operations. It takes three sets of file descriptors as input (for reading, writing and errors), and the maximum file descriptor value and returns the number of file descriptors that are ready. However, select() has a limitation that the maximum numbers of file descriptors that it can monitor is limited by the operating system.
- 'poll()' is similar to 'select()', but it doesnot have the same limitation on the number of file descriptors that it can monitor. It takes an array of structures as inputs, each of which contain a file descriptor and events (e.g. POLLIN for input, POLLOUT for output) to be monitored and returns the number of file descriptors that are ready.

- In summary, 'select()' is a blocking call that can handle a limited number of file descriptor, while 'poll()' is a non-blocking call that can handle a larger number of file descriptor and it is more efficient.

<u>Poll()</u>: 'poll()' is a system call that allows a program to monitor multiple file descriptors, and wait for one or more of them to become ready for I/O operations. The call takes an array of structures as input, where each structure contains a file descriptor and events (e.g POLLIN for input, POLLOUT for output) to be monitored. The call returns the number of file descriptors that are ready for the specified events.

'poll()' function is designed to be more efficient than 'select()' as it

Scanned with CamScanner
Scanned with CamScanner

doesnot have the same limitations on the number of file descriptors that it can monitor. It also has the ability to handle events such as POLLERR, POLLHUP, and POLLNVAL in addition to the standard read and write events.

<u>pselect()</u>: 'pselect()' is similar to 'select()' but it has the ability to atomically block both the 'select()' and 'poll()' system calls. The 'pselect()' call takes three sets of file descriptors as input (for reading, writing, and errors), the maximum file descriptors value, a timeout value and a signal mask. The call returns the number of file descriptors that are ready, or -1 if an error occured.

'pselect()' can be useful when you want to wait for I/O events on multiple file descriptors, but also want to atomically block signals during the wait. This can help prevent race conditions where a signal is delivered and handled while 'select()' or 'poll()' is in progress, potentially leading to unexpected behavior.

24. Differentiate blocking IO model with non-blocking IO model. Can we use signal driven IO as asynchronous IO? If yes, how? If no, why we can not use signal IO as asynchronous IO?

:) - In a blocking IO model, when a process makes a call to a blocking IO system call, such as 'read()' or 'write()', the process is blocked (i.e. put to sleep) until the IO operation completes. This means that the process cannot perform any other tasks while it is waiting for the IO operation to complete.

- In a non-blocking IO model, when a process makes a call to a non-blocking IO system call, such as 'read()' or 'write()', the call

Scanned with CamScanner
Scanned with CamScanner

returns immediately with an error or with the data, if it is availabl. This means that the process can continue to perform other tasks while it is waiting for the IO operation to complete.

- The advantage of the non-blocking IO model is that it allows a process to perform multiple tasks concurrently, and it can respond to IO events in a timely manner. On the other hand, the blocking IO model can cause a process to become blocked and unresponsive, if it is waiting for a slow IO operation to complete.

- Additionally, the non-blocking IO model is typically more efficient in terms of resource usage, as it does not require the operating system to ~~handle~~ schedule the blocked process and it allows the program to handle the IO events as soon as they happen.

- The choice of IO model depends on the specific requirements of the application. In cases where responsiveness is important, non-blocking IO is often preferred. In cases where the application can afford to wait for IO operations to complete, blocking IO can be used.

Yes, signal driven IO can be used as asynchoronous IO.
In a signal-driven IO model, a process sets up a signal handler and uses the 'sigaction()' or 'signal()' system call to associate a signal with a file descriptor. When an IO events occurs on the file descriptor, the operating system sends a signal to the process, which causes the process's signal handler to be executed.
The process can then use the 'read()' or 'write()' system call within the signal handler to perform the IO operation and retrieve the

Scanned with CamScanner
Scanned with CamScanner

data.

One advantage of this method is that it allows a process to perform multiple tasks concurrently, and it can respond to IO events in a timely manner, similar to non-blocking IO. However, a major drawback of this method is that, since the process is doing IO in a signal handler, it can be dangerous and can lead to unpredictable behaviour if the process is not written carefully. Also the signal handlers are called in an asynchronous manner and the process might be in the middle of doing something important. This can disrupt the normal flow of execution and cause unexpected issues.

Therefore, it is generally not recommended to use signal driven IO as the primary method for asynchronous IO, as it can be complex and error-prone. Instead, it is better to use non-blocking IO or IO multiplexing mechanisms like 'select()', 'poll()' and 'epoll()' which are designed specifically for this purpose and are safer to use.

Scanned with CamScanner
Scanned with CamScanner

25. Differentiate fcntl() and ioctl() system call. Explain the meaning of following socket options: SO_BROADCAST, SO_KEEPALIVE, SO_RCVTIMEO, SO_SNDTIMEO, SO_RECUEADDR and SO_LINGER.

=>

fcntl()	ioctl()
i) fcntl() stands for file control and this function performs various descriptor control operations.	i) Common use of ioctl() by network programs is to obtain information on all the hosts interface when the program starts, the interface addresses whether interface supports broadcasting the interface supports multicasting and so on.
ii) int fcntl (int fd, int cmd, long arg);	ii) int ioctl (int fd, int request, ... /* void *arg */); - returns 0 if ok, -1 on error. - third argument is always a pointer, but the type of pointer depends on the request.
iii) To set socket for non-blocking I/O, we use F_SETFL, O_NONBLOCK.	iii) To set socket for non-blocking I/O, we use FIONBIO.
iv) we cannot get # bytes in socket receive buffer using fcntl().	iv) we use FIOREAD to get # bytes in socket received buffer.

a) SO_BROADCAST:

- Enables or disables the ability of a process to send broadcast message
- it is supported only for datagram sockets.
- its default value is off.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

b) SO_KEEPALIVE :

- purpose of this option is to detect if the peer host crashes. The SO_KEEP-ALIVE option will detect half open connection and terminate them.

c) SO_RCVTIMEO and SO_SNDTIMEO.

- These options place a timeout on socket receives and sends
- The timeout value is specified in a timeval structure.

```
struct timeval {
    long tv_sec;
    long tv_usec;
}
```

- To disable a timeout, the values in the timeval structure are set to 0.

d) SO_REUSEADDR:

- It allows a listening server to restart and bind its well known port even if previously established connections exist.

- it allows multiple instances of the same server to be started on the same port as long as each instance binds a different local ip address.

- it allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address.

e) SO_LINGER:

- specifies how close operates for a connection-oriented protocol.
- the following structure is used.

Scanned with CamScanner
Scanned with CamScanner

```
struct linger {
        int l_onoff;
        int l_linger;
}
// l_onoff - 0 = off; nonzero = on
// l_linger specifies seconds
```

26. What are the socket options ? Which functions are used to set and get a value of socket options ? Explain them in detail with sample code.

=) Socket options are settings that can be applied to sockets to change their behaviour. Some examples of socket options include the ability to enable or disable the Nagle algorithm, set the size of the receive buffer, or set the time-to-live (TTL) value for IP packets.

In C, the 'setsockets' function is used to set the value of a socket option, and the 'getsockets' function is used to retrieve the current value of a socket option.

The 'setsockets' function takes the following parameters:

- The socket descriptor
- The level of the option, which specifies the protocol for the option, for example, SOL_SOCKET for options that are applicable to the socket level, IPPROTO_IP for options that are applicable to the IP level, and so on.
- The option name, which specifies the specific option to set. For e.g. SO_REUSEADDR for the reuse address option, SO_RCVBUF for the receive buffer size option, and so on.
- A pointer to the buffer that holds the new value for the option
- The size of the buffer.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

Here, is an example of how to use the 'setsockopt' function to enable the reuse of the address on a socket:

```
int reuse_addr = 1;
if (setsockopt (sockfd, SOL_SOCKET, SO_REUSEADDR, &reuse_addr,
    sizeof (reuse_addr)) <0) {
    perror ("setsockopt");
    exit(1);
}
```

The 'getsockopt()' function takes the following parameters:

- The socket descriptor
- The level of the option
- The option name
- A pointer to the buffer that will hold the current value of the option.
- A pointer to an integer that will hold the size of the buffer.

Here is an example of how to use the 'getsockopt()' function to retrieve the current value of the receive buffer size option:

```
int bufsize;
socklen_t optlen = sizeof (bufsize);
if (getsockopt (sockfd, SOL_SOCKET, SO_RCVBUF, &bufsize, &optlen)<0) {
    perror ("getsockopt");
    exit(1);
}
print ("Current receive buffer size : %d\n", bufsize);
```

Scanned with CamScanner
Scanned with CamScanner

27. What are the meanings of SIOCATMARK, SIOCGIPGRP and SIOCSPGRP? How do you set socket as non blocking using fcntl() and ioctl()? Illustrate with section of code.

⇒ SIOCATMARK:

- return through the integer pointed to by the third argument a non-zero value. If the socket's read pointer is currently at the out of band mark or a zero value. If the read pointer is not at the out-of-band mark.

↳ SIOCGPGRP:

- return through the integer pointed to by the third argument either the process ID or the process group ID that is set to receive SIGIO or SIGURG signal for this socket. This request is identical to an fcntl of F_GETOWN note that POSIX standardizes the fcntl.

↳ SIOCSPGRP:

- set either the process ID or process group ID to receive the SIGIO or SIGURG signal for this socket from the integer pointed to by the third argument. This request is identical to an fctnl of F_SETOWN note that POSIX standardizes the fcntl.

⇒ Code to enable non-blocking I/O using fcntl()

```
int flags;
        /* set a socket as non-blocking */
if ((flags = fcntl (fd, F_GETFL, 0)) < 0)
        err_sys ("F_GETFL error");
        flags = O_NONBLOCK;
if (fcntl (fd, F_SETFL, flags) < 0)
        err_sys ("F_SETFL error");
```

Scanned with CamScanner
Scanned with CamScanner

→ FIONBIO is used for non-blocking I/O in ioctl()
FIONBIO → the non-blocking flag for the socket is cleared or turned on, depending on whether the third argument to ioctl points to zero or non-zero value respectively. This request has the same effect as the O_NONBLOCK file status flag, which can be set and cleared with the F_SETFL command to the fcntl function.

→ int ioctl (int sd, unsigned long cmd, void * data)
- if * data is 0, ioctl clears.
non-blocking I/O. if data is not 0,
sd is set for non-blocking I/O.
On = True
Statue = ioctl (sfd, FIONBIO,

28. What is daemon process and how does it started? How do you create the daemon process in Unix? Explain.

⇒ A daemon is a process that runs in the background and is not associated with a controlling terminal. Typically it is started when the system is bootstrapped.

→ We can create the Daemon process in Unix System in following ways:-

i) fork: At first, we call fork and parent terminates and child continues. If the process was started as a shell command in the foreground when the parent terminates the shell. This automatically runs child process in the background.

ii) Setsid: It is a POSIX signal that creates a new session. The process becomes the session leader of new session, becomes the process group leader of a new process group and has no controlling terminal.

Scanned with CamScanner
Scanned with CamScanner

iii) Ignore, SIGHUP and fork again

→ We ignore SIGHUP and call fork again. When this function returns the parent is really the first child and it terminates leaving the second child running. The second fork guarantees that daemon cannot automatically acquire a controlling terminal should it open a terminal device in the future.

iv) Change working directory

→ We change the working directory to root directory. Since file system can't be unmounted if the working directory is not changed.

v) Close any open descriptor:

→ We close any open descriptors inherited from the process that executes the daemon.

vi) Redirect stdin, stdout and stderr to /dev/null

→ We open /dev/null for standard input, standard output, and standard error. This guarantees that these common descriptors are open, and a read from any of these descriptors returns 0 (EOF) and the kernel just discards anything written to them.

vii) Use syslogd for errors:

→ The syslogd daemon is used to log errors.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

29. What is syslogd()? What is the technique for logging messages from a daemon process? Explain with sample code.

→ syslogd() is a daemons which runs in an infinite loop that call select waiting for any one of its three descriptors to be readable.

- a unix domain it reads the log messages and does what the configuration file says to do with that message.
- The technique for logging messages from a daemon process is to call the syslog function.

```
# include <syslog.h>
void syslog (int priority, const char * message, ...);
```

→ Here, the priority argument is a combination of a level and facility. The message like a format string to printf with an addition of %m which is replaced with an error message corresponding to the value of errno.

→ The following call would be issued by a daemon when a call to rename function unexpectedly fails.

```
syslog (LOG_INFO | LOG_LOCAL2, "rename (%s, %s): %m", file1,
  file2);
```

<u>sample code</u>

```
# include <stdio.h>
# include <unistd.h>
# include <syslog.h>
# include <fontl.h>
# include <signal.h>
```

Scanned with CamScanner
Scanned with CamScanner

```
# define MAXED 64
int main()
{
    int i;
    pid_t pid;
    if ((pid = fork()) < 0)
        return -1;
    signal (SIGHUP, SIG_IGN);
    if ((pid = fork()) < 0)
        return -1;
    else if (pid)
        exit(0);
    chdir ("/");
    for (i=0 ; i< MAXED ; i++)
        close (i);
    open ("/dev/null", O_RDONLY);
    open ("/dev/null", O_RDWR);
    open ("/dev/null", O_RDWR);
    openlog ("My Daemon :", LOG_PID, LOG_USR);
    syslog (LOG_ERR, "log message from process with %d",
            getpid());
    close log();
    return 0;
}
```

Scanned with CamScanner
Scanned with CamScanner

30. How close() is different from shutdown()? Which functions set h_errno global variable? Differentiates errno and h_errno as well.

=>

Close()	Shutdown()
i) Normal way to terminate connection is to call the close function. Closes the socket but the connection is still open for processes that shares this socket connection stays opened both for read and write.	i) shutdown() breaks the connection for all processes sharing the socket. A read will detect EOF and a write will receive SIGPIPE.
ii) Close() decrements the descriptor reference count and closes the socket only if the count reaches 0.	ii) With shutdown we can initiate TCPs normal connection termination sequence regards of the reference count.
iii) close terminate both directions of data transfer, reading and writing.	iii) It terminates one direction (half) of the connection.
iv) int close (int sockfd);	iv) int shutdown (int sockfd, int how to);

=> gethostbyname() and gethostbyaddr() set h_errno global variable.
gethostbyname() sets the global integer h_errno to one of the following constants defined by including <netdb.h>

- HOST_NOT_FOUND
- NO_RECOVERY
- TRY_AGAIN
- NO_DATA

Scanned with CamScanner
Scanned with CamScanner

Errno	h_errno
i) If a process is blocked in a call to 'select' the return from the signal handler causes the function to return with 'errno' set to 'EINTR'	i) It is a host error variable
ii) socket function like select(), poll() set errno when an error occurs.	ii) gethostbyname() and gethostbyaddr() sets the global integer h_errno to one of the following constants defined by including <netdb.h>: - HOST_NOT_FOUND - TRY_AGAIN - NO_RECOVERY - NO_DATA

31. How windows socket is different from berkeley socket? Explain the role of setup(), cleanup() and WSAStartup() function in winsock architecture.

=> Windows socket is different from berkeley socket in following ways:-

i) Winsock supports TCP/IP domain for IPC on the same computer as well as network communication. In addition to TCP/IP domain, socket is must Unix implementation supports the Unix domain for IPC on the same computer.

ii) Return values of certain Berkeley's functions are different. for e.g: Socket() function returns -1 on failure in Unix, in the winsock implementation returns INVALID_SOCKET.

Scanned with CamScanner
Scanned with CamScanner

iii) Certain Berkely functions have different names in winsock. For e.g in Unix, the close() system call is used to close a connection in winsock, the function is called closesocket().

→ The role of setup() and cleanup() are:-
↳ the winsock functions, the application needs are located in the dynamic library named WINSOCK.DLL or WSOCK32.DLL depending on whether the 16-bit or 32-bit version of windows is being targeted.
↳ the application is linked with either WINSOCK.LIB or WSOCK32.LIB as appropriate.

↳ this include file where the winsock function and structures are defined in named WINSOCK.H for both the 16-bit and 32-bit environments.

→ the following are the roles of WSAStartup() functions:
↳ initialize the underlying windows socket. Dynamic link library
↳ Gives the TCP/IP stack under a chance to do any application specific initialization.
↳ Also used to confirm that the version of winsock.DLL is compatible with requirements of the application.

→ Following are the roles of WSACLEANUP() function:
↳ Used to terminate an applications use of winsock.
↳ Usually called after the applications message loop has been terminated.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

32. Explain winsock architecture. What are the types of dynamic linking in winsock programming?

Ans) **Winsock Architecture**

Windows Sockets 16-bit 1.1 Application	Windows Sockets 32-bit 1.1 Application	Windows Sockets 32-bit 2.0 Application

Windows Sockets 1.1 API

Winsock.dll	Wsock 32.dll

Windows Sockets 2.0 API

Ws2_32.dll	Mswsock.dll Wshelp.dll

Windows Sockets 2.0 SPI

Layered Service Provider

Helper DLLs	Name Space DLLs
Wshtcpip.dll	Nwprovau.dll
Wshnetbs.dll	Rnr20.dll
Wshirda.dll	Winrnr.dll
Wshatm.dll	
Wshisn.dll	
Wshisotp.dll	
Sfmwshat.dll	

Msafd.dll	User Mode
Afd.sys	Kernel Mode

TDI Layer

Scanned with CamScanner
Scanned with CamScanner

- The top layer shows different network applications pertaining to different Winsock API versions and Windows architectures.
- The two layers below it specify DLLs used for Winsock API version 1.1 and 2.0 respectively.

- Winsock provides a Service Provider Interface for creating Winsock services, commonly referred to as the Winsock SPI.

- Afd.sys is a Windows driver. A driver is a small software program that allows your computer to communicate with hardware or connected devices.

→ There are two methods for calling a function in a DLL:
↳ In load-time dynamic linking, a module makes explicit calls to exported DLL functions as if they were local functions. This requires to link the module with the import library for the DLL that contains the functions. An import library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded.

↳ In run-time dynamic linking, a module uses the LoadLibrary or LoadLibraryEx function to load the DLL at run time. After the DLL is loaded, the module calls the GetProcAddress function to get the addresses of the exported DLL functions. The module calls the exported DLL functions using the function pointers returned by GetProcAddress. This eliminates the need for an import library.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

33. Differentiate blocked I/O with unblocked I/O. Explain all the functions in detail to handle blocked I/O in winsock architecture

=)

Blocked I/O	Unblocked I/O.
i) It means when socket functions take indeterminate amount of time to executes it is said to block; calling function blocks further execution.	i) We use single thread to handle multiple concurrent connections.
ii) We either need to accept that we are going to wait for every I/O request or we need to fire off thread per request.	ii) We can send off multiple requests but we need to keep in mind that the data will not be available until same latter point.

→ The functions that are needed to handle blocked I/O in winsock architecture are as follows:-

i) <u>WSAsyncselect()</u>: It works by sending a windows message to notify a window of a socket event.

int WSAAsyncselect (socket s, HWND hwnd, u_int wMsg, long lEvent);

Here,

→ s is a socket descriptor for which event notification is required.

→ hwnd is a windows handle that should receive a message when an event occurs in socket.

→ wMsg, is the message to be received by hwnd when a socket event occurs in socket.

Scanned with CamScanner
Scanned with CamScanner

Date
Page

→ lEvent is a bitmask that specifies the events in which the application is interested.

→ WSAAsynSelect() returns 0 on success and SOCKET_ERROR on failure. On failure WSAGetlastError should be called.

34. What is overlapped I/O? Which function is used to set windows socket in non-blocking mode? Explain non-blocking socket with connect() in the case of winsock architecture.

⇒ Overlapped I/O is also called asynchronous I/O in which form of input/output processing that permits other processing to continue before the transmission has finished.

→ ioctlsocket() function is used to set windows socket in non-blocking mode.

→ <u>non-blocking socket with connect()</u>

- When connect is called on a blocking socket, the function blocks i.e. the function doesn't return until the TCP's 3-way handshake is completed (actually until SYN, Ack is received from remote ends).

- the ioctlsocket function can be used to set socket in non-blocking mode, with a non-blocking socket, the connect returns immediately without completion.
- In this case, connect will return SOCKET_ERROR and WSAGETLastError will return WSAEWOULDBLOCK
- In this case, there are 3 possible scenarios:-

Scanned with CamScanner
Scanned with CamScanner

i) Use the socket function to determine the completion of the connection request by checking to see if the socket is writeable. If connect fails, failure of the socket attempt is indicated in exceptfd cappli cation must than call getsockopt SO_ERROR to determine the error value to describe why the failure occurred.

ii) If the application is using WSAAsyncselect to indicate in Connect events, then the application will receive an FD_CONNECT notification indicating that the connect operation is complete (successfully or not).

iii) If the application is using WSAEventselect to indicate in Connection events, then the associated event object will be signaled indicating that the connect operation is complete (successfully or not).

3s. Why WSAGetLastError() is required in winsock programming? Explain WSAGetOverlappedResult() function.

Scanned with CamScanner
Scanned with CamScanner