NP UNIT 2

1. Illustrate the significance of socket functions for elementary TCP client/server with a neat block diagram.

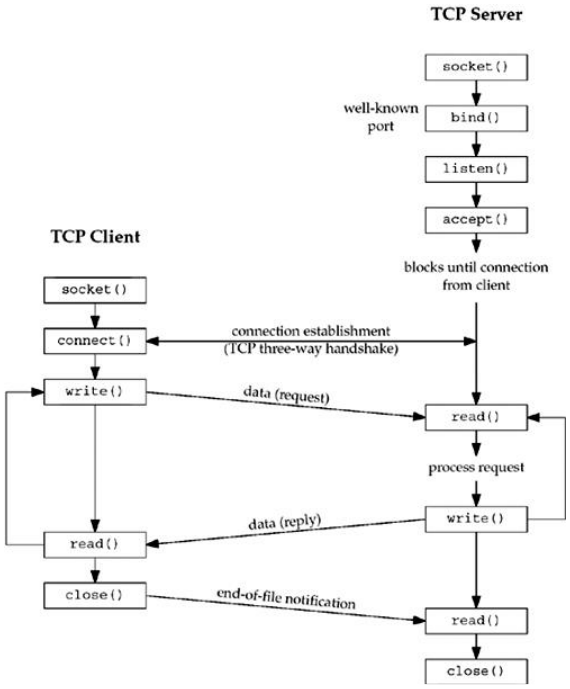
OR

What are Socket Functions? With neat figure explain Socket functions for elementary TCP client/Server. 10 mark

Ans: Socket functions –

- Socket functions are a set of programming functions or APIs (Application Programming Interfaces) provided by the operating system to enable network communication between processes over a computer network.
- These functions are commonly used in network programming to create, configure, and manage network sockets.
- Sockets provide a standard mechanism for processes on different devices to communicate over a network.

- Figure shows a timeline of the typical scenario that takes place between a TCP client and server.
- First, the server is started, then sometime later, a client is started that connects to the server.
- We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client.
- This continues until the client closes its end of the connection, which sends an end-of-file notification to the server.
- The server then closes its end of the connection and either terminates or waits for a new client connection.



2. Develop a C Program to demonstrate the TCP echo server: main function.

Ans:

```
#include "unp.h"
Int main(int argc, char **argv)
{
        int listenfd, connfd;
        pid_t childpid;
        socklen_t clilen;
        struct sockaddr_in cliaddr, servaddr;
        listenfd = Socket (AF_INET, SOCK_STREAM, 0);
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
        servaddr.sin_port = htons (SERV_PORT);
        Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
        Listen(listenfd, LISTENQ);
        for ( ; ; ) {
                clilen = sizeof(cliaddr);
                connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
                if ( (childpid = Fork()) == 0) { /* child process */
                        Close(listenfd); /* close listening socket */
                        str_echo(connfd); /* process the request */
                        exit (0);
                }
                Close(connfd); /* parent closes connected socket */
        }
}
```

3. Explain the following arguments of the socket function a) family b) Type c) Protocol

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol.

```
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

Returns: non-negative descriptor if OK, -1 on error

Arguments:

a) family

family specifies the protocol family and is one of the constants in the table below. This argument is often referred to as domain instead of family.

Protocol *family* constants for socket function.

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

b) type

The socket *type* is one of the constants in the table below.

***type* of socket for socket function.**

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

c) protocol

The *protocol* argument to the socket function should be set to the specific protocol type found in the following table, or 0 to select the system's default for the given combination of family and type.

***protocol* of sockets for AF_INET or AF_INET6.**

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

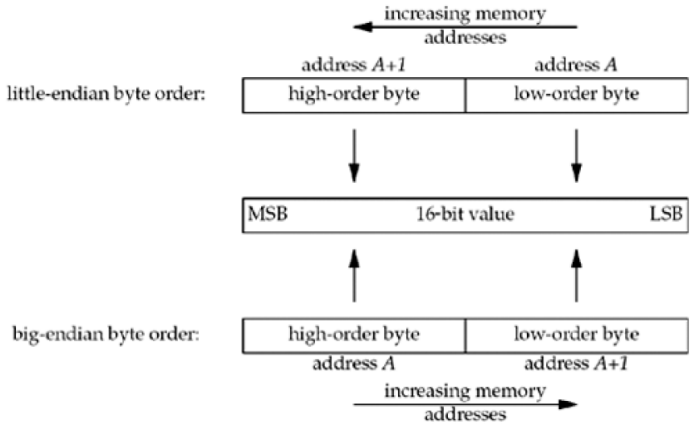
Not all combinations of socket *family* and *type* are valid. Following table shows the valid combinations, along with the actual protocols that are valid for each pair. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.

Combinations of *family* and *type* for the socket function.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP | SCTP	TCP | SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

4. Compare the little-endian and big-endian byte ordering functions with a neat diagram. Develop a C Program to determine host byte order.

little-endian byte order	*big-endian* byte order
Consider a 16-bit integer that is made up of 2 bytes. The two bytes in memory are stored with the low-order byte at the starting address. This is known as *little-endian* byte order	Consider a 16-bit integer that is made up of 2 bytes. The two bytes in memory are stored with the high-order byte at the starting address. This is known as *big-endian* byte order



- The terms "little-endian" and "big-endian" indicate which end of the multi byte value, the little end or the big end, is stored at the starting address of the value.
- We must deal with these byte ordering differences as network programmers because networking protocols must specify a network byte order.
- For example, in a TCP segment, there is a 16-bit port number and a 32-bit IPv4 address. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields will be transmitted. The Internet protocols use big-endian byte ordering for these multibyte integers.

```
#include "unp.h"
Int main(int argc, char **argv)
{
        union {
                short s;
                char c[sizeof(short)];
        } un;
        un.s = 0x0102;
        printf("%s: ", CPU_VENDOR_OS);
        if (sizeof(short) == 2) {
                if (un.c[0] == 1 && un.c[1] == 2)
                        printf("big-endian\n");
                else if (un.c[0] == 2 && un.c[1] == 1)
                        printf("little-endian\n");
                else
                        printf("unknown\n");
```

```
        } else
                printf("sizeof(short) = %d\n", sizeof(short));
        exit(0);
}
```

5. Discuss value-result arguments passed from process to kernel and kernel to process. 10 mark

OR

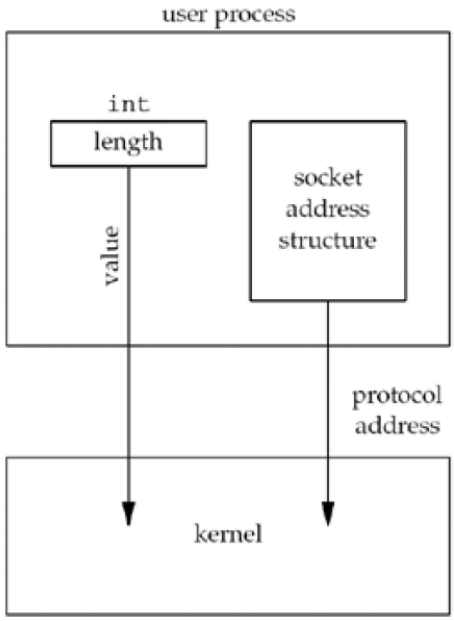
Explain Value-Result Arguments.

- When a socket address structure is passed to any socket function, 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 the process to the kernel, or vice versa.
- Three functions, bind, connect, and sendto, pass a socket address structure from the process to the kernel.
- One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in:

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

- Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel.

Socket address structure passed from process to kernel.



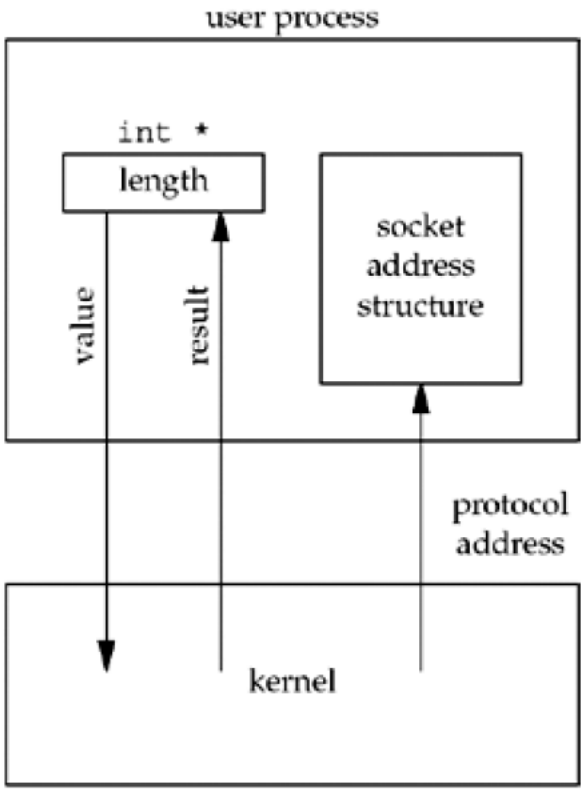
- Four functions, accept, recvfrom, getsockname, and getpeername, pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario.
- Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure, as in

```
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 */

- The reason that the size changes from an integer to be a pointer to an integer is because the size is both a *value* when the function is called (it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in) and a *result* when the function returns (it tells the process how much information the kernel actually stored in the structure).

Socket address structure passed from kernel to process.



6. With a neat diagram, explain the sub-parts of Sockaddr_in structure and justify why it must be typecast to sockaddr while passing Sockaddr_in variable to bind API as an argument 10 mark.

OR

Explain IPV4 socket address structure and its fields

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named sockaddr_in and is defined by including the <netinet/in.h> header.

```
struct sockaddr_in {
        uint8_t         sin_len;        /* length of structure (16) */
        sa_family_t sin_family;         /* AF_INET */
        in_port_t       sin_port;       /* 16-bit TCP or UDP port number */
                                        /* network byte ordered */

        struct in_addr sin_addr;        /* 32-bit IPv4 address */
                                        /* network byte ordered */
        char sin_zero[8];               /* unused */
};
```

- **Sin_len**: This function copies the socket address structure from the process and explicitly sets its sin_len member to the size of the structure that was passed as an argument to socket functions.
- **sin_family**: A 16-bit field that specifies the address family (e.g., AF_INET for IPv4). This field is crucial for the system to interpret the structure correctly.
- **sin_port**: A 16-bit field that represents the port number in network byte order. Network byte order is big-endian.
- **sin_addr**: A structure of type struct in_addr that holds the 32-bit IPv4 address.
- **sin_zero:** An 8-byte array that pads the structure to the size of struct sockaddr. It is not used in practice and is included for historical reasons.

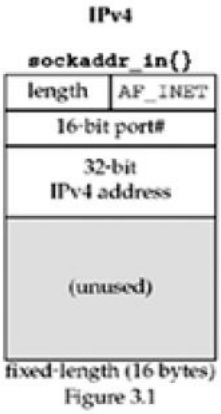


Figure 3.1

- The socket address structures contain a one-byte length field, that the family field also occupies one byte, and that any field that must be at least some number of bits is exactly that number of bits. The socket address structures are fixed-length.

- When passing a struct sockaddr_in to the bind API, it must be typecast to struct sockaddr because the bind function expects a pointer to a generic socket address structure (struct sockaddr). This typecasting is necessary to maintain compatibility with the generic interface provided by the Berkeley Sockets API.
- The struct sockaddr is a generic structure that can be used to represent socket addresses for various address families, and typecasting ensures that the function receiving the address can correctly interpret the specific type of address being passed.

7. Demonstrate with appropriate code, the application of fork () and exec() APIs in Concurrent Server implementation 10 mark.
Or
Outline the typical concurrent server with the help of pseudocode. 8 marks

The simplest way to write a *concurrent server* under Unix is to fork a child process to handle each client.

```
pid_t pid;
int listenfd, connfd;
listenfd = Socket( ... );
/* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);
for ( ; ; ) {
        connfd = Accept (listenfd, ... ); /* probably blocks */
        if( (pid = Fork()) == 0) {
                Close(listenfd); /* child closes listening socket */
                doit(connfd); /* process the request */
                Close(connfd); /* done with this client */
                exit(0); /* child terminates */
        }

        Close(connfd); /* parent closes connected socket */

}
```

When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket).
The parent closes the connected socket since the child handles the new client.

8. What are Concurrent Servers? Explain how does Concurrent Servers handle multiple clients at the same time. (explain with the necessary code and the diagrams) 10 marks

- When a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time. Special socket calls are available for that purpose they are called concurrent servers.
- The simplest way to write a *concurrent server* under Unix is to fork a child process to handle each client.

```
pid_t pid;
int listenfd, connfd;
listenfd = Socket( ... );
/* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);
for ( ; ; ) {
        connfd = Accept (listenfd, ... ); /* probably blocks */
        if( (pid = Fork()) == 0) {
                Close(listenfd); /* child closes listening socket */
                doit(connfd); /* process the request */
                Close(connfd); /* done with this client */
                exit(0); /* child terminates */
        }

        Close(connfd); /* parent closes connected socket */

}
```

9. Defend the use of htons/htonl and inet_pton functions in network programming. Write a sample program to demonstrate the use of above functions. 10 marks

The htons/htonl functions are used for converting between host byte order and network byte order.

- **htons** - host to network short — This function converts 16-bit quantities from host byte order to network byte order.
- **htonl**: host to network long — This function converts 32-bit quantities from host byte order to network byte order.

```
#include <netinet/in.h>
uint16_t htons(uint16_t host16bitvalue) ;
uint32_t htonl(uint32_t host32bitvalue) ;
```

Both return: value in network byte order.

inet_pton : This function tries to convert the string pointed to by *strptr*, storing the binary result through the pointer *addrptr*. If successful, the return value is 1.
If the input string is not a valid presentation format for the specified *family*, 0 is returned.

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);
```

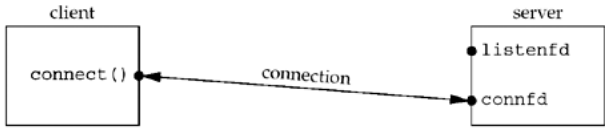
Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error

10. Develop the C program to demonstrate the TCP echo client:str_cli function. 12 marks

```
#include "unp.h"
void str_cli(FILE *fp, int sockfd)
{
        char sendline[MAXLINE], recvline[MAXLINE];
        while (Fgets(sendline, MAXLINE, fp) != NULL) {
                writen(sockfd, sendline, strlen (sendline));
                if (Readline(sockfd, recvline, MAXLINE) == 0)
                        err_quit("str_cli: server terminated prematurely");
                Fputs(recvline, stdout);
        }
}
```

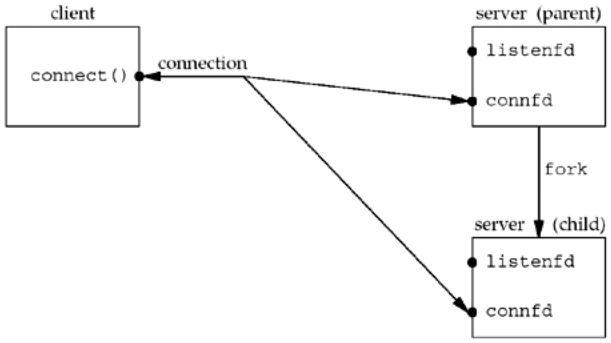
8. Demonstrate the status of client/server after fork returns with a neat block diagram. 8 marks

- Immediately after accept returns, we have the scenario shown in following Figure.



- The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

- The next step in the concurrent server is to call fork. Following shows the status after fork returns.



- Both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.
- After fork returns, *both descriptors are shared (i.e., duplicated) between the parent and child,* so the file table entries associated with both sockets now have a reference count of 2.
- Therefore, when the parent closes connfd, it just decrements the reference count from 2 to 1 and that is all.
- The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0.
- This will occur at some time later when the child closes connfd.

9. Develop the pseudocode that returns the address family of a Socket 12 marks

```
#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);
}
```

10. What are Sockets? Explain Socket address structures

- A socket is a software endpoint that establishes a communication link between two processes across a network.
- It's a fundamental concept in network programming and is commonly used in client-server applications.
- A socket is created using the following code –

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

  where, sockid – a socket descriptor, an integer; family – integer, communication domain; type – communication type; protocol – specifies the protocol.

Socket address structure:

Most socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begin with `sockaddr_` and end with a unique suffix for each protocol suite.

IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named `sockaddr_in` and is defined by including the `<netinet/in.h>`

```
struct sockaddr_in {
        uint8_t         sin_len;        /* length of structure (16) */
        sa_family_t sin_family;         /* AF_INET */
        in_port_t       sin_port;       /* 16-bit TCP or UDP port number */
                                        /* network byte ordered */

        struct in_addr sin_addr;        /* 32-bit IPv4 address */
                                        /* network byte ordered */
        char sin_zero[8];               /* unused */
};
```

IPv6 Socket Address Structure

The IPv6 socket address is defined by including the `<netinet/in.h>` header.

```
#define SIN6_LEN /* required for compile-time tests */
struct sockaddr_in6 {
        uint8_t sin6_len;               /* length of this struct (28) */
        sa_family_t sin6_family;        /* AF_INET6 */
        in_port_t sin6_port;            /* transport layer port# */
        /* network byte ordered */
        uint32_t sin6_flowinfo;         /* flow information, undefined */
        struct in6_addr sin6_addr;      /* IPv6 address */
        /* network byte ordered */
        uint32_t sin6_scope_id; /* set of interfaces for a scope */
};
```

11. Explain Generic socket address structure and its fields

- A socket address structures is *always* passed by reference when passed as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from *any* of the supported protocol families.
- A *generic* socket address structure is defined in the <sys/socket.h> header.

```
struct sockaddr {
uint8_t sa_len;
sa_family_t sa_family;          /* address family: AF_xxx value */
char sa_data[14];                /* protocol-specific address */
};
```

- Where,
  - → sa_family – 16 bit integer value identifying the protocol family being used; eg – TCP/IP -> AF_INET.
  - → Sa_data – address information used in the protocol family; eg - TCP/IP -> IP address and port number.

- The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the bind function:

```
int bind(int, struct sockaddr *, socklen_t);
```

- This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure.
- For example,

```
struct sockaddr_in serv; /* IPv4 socket address structure */
/* fill in serv{} */
bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

- From an application programmer's point of view, the *only* use of these generic socket address structures is to cast pointers to protocol-specific structures.

12. Explain IPV6 address structure and its fields

- The IPv6 socket address is defined by including the <netinet/in.h> header.

```
struct in6_addr {
        uint8_t s6_addr[16]; /* 128-bit IPv6 address */
                             /* network byte ordered */
};
#define SIN6_LEN /* required for compile-time tests */
struct sockaddr_in6 {
        uint8_t sin6_len;          /* length of this struct (28) */
        sa_family_t sin6_family;   /* AF_INET6 */
        in_port_t sin6_port;       /* transport layer port# */
                                   /* network byte ordered */
        uint32_t sin6_flowinfo;    /* flow information, undefined */
        struct in6_addr sin6_addr; /* IPv6 address */
                                   /* network byte ordered */
        uint32_t sin6_scope_id;    /* set of interfaces for a scope */
};
```

- The SIN6_LEN constant must be defined if the system supports the length member for socket address structures.
- 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_in6 structure is 64-bit aligned, so is the 128-bit sin6_addr member. On some 64-bit processors, data accesses of 64-bit values are optimized if stored on a 64-bit boundary.
- 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

13. **Implement client server communication using socket programming that uses connection oriented protocol at transport layer 10 Marks**

14. Explain New Generic socket address structure and its fields

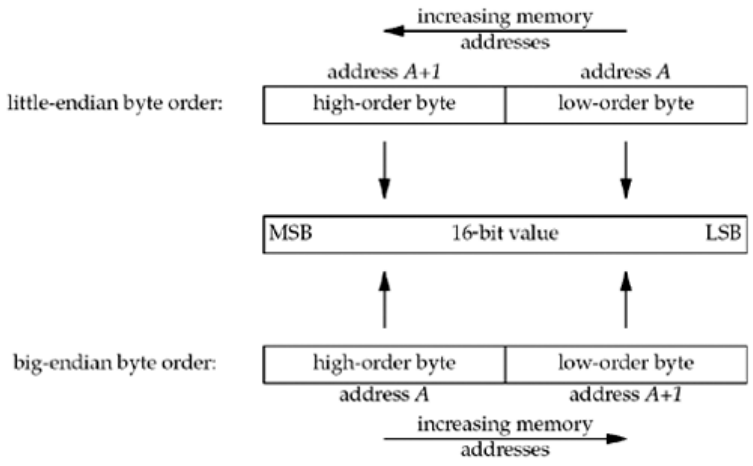
- A new generic socket address structure was defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing struct sockaddr.
- Unlike the struct sockaddr, the new struct sockaddr_storage is large enough to hold any socket address type supported by the system.
- The sockaddr_storage structure is defined by including the <netinet/in.h> header.

```
struct sockaddr_storage {
        uint8_t ss_len;  /*length of this struct (implementation dependent)*/
        sa_family_t ss_family; /* address family: AF_xxx value */
};
```

- The sockaddr_storage type provides a generic socket address structure that is different from struct sockaddr in two ways:
  - → If any socket address structures that the system supports have alignment requirements, the sockaddr_storage provides the strictest alignment requirement.
  - → The sockaddr_storage is large enough to contain any socket address structure that the system supports.
- The fields of the sockaddr_storage structure are opaque to the user, except for ss_family and ss_len (if present).
- The sockaddr_storage must be cast or copied to the appropriate socket address structure for the address given in ss_family to access any other fields.

15. Explain Byte Ordering Functions

- Consider a 16-bit integer that is made up of 2 bytes.
- There are two ways to store the two bytes in memory: with the low-order byte at the starting address, known as *little-endian* byte order, or with the high-order byte at the starting address, known as *big-endian* byte order.



- In this figure, we show increasing memory addresses going from right to left in the top, and from left to right in the bottom. We also show the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.
- The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.
- There is no standard between these two byte orderings and we encounter systems that use both formats. We refer to the byte ordering used by a given system as the *host byte order*.

16. Explain Byte Manipulation Functions.

- There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string.
- We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings.

- The first group of functions, whose names begin with **b** (for byte).
- The finctions are – bzero, bcopy, bcmp.

```
#include <strings.h>
void bzero(void *dest, size_t nbytes);
void bcopy(const void *src, void *dest, size_t nbytes);
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
Returns: 0 if equal, nonzero if unequal
```

→ **bzero** sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0.
→ **bcopy** moves the specified number of bytes from the source to the destination.
→ **bcmp** compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero.

- The second group of functions, whose names begin with **mem** (for memory).
- The functions are – memset, memcpt, memcmp.

```
#include <string.h>
void *memset(void *dest, int c, size_t len);
void *memcpy(void *dest, const void *src, size_t nbytes);
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);
Returns: 0 if equal, <0 or >0 if unequal (see text)
```

→ **memset** sets the specified number of bytes to the value c in the destination.
→ **memcpy** is similar to bcopy, but the order of the two pointer arguments is swapped.
→ **memcmp** compares two arbitrary byte strings and returns 0 if they are identical.

17. Explain fork and exec Functions

Fork:

- This function is the only way in Unix to create a new process.

```
#include <unistd.h>
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, -1 on error

- It returns once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child). It also returns once in the child, with a return value of 0. Hence, the return value tells the process whether it is the parent or the child.
- The reason fork returns 0 in the child, instead of the parent's process ID, is because a child has only one parent and it can always obtain the parent's process ID by calling getppid.
- There are two typical uses of fork:
  1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is typical for network servers. We will see many examples of this later in the text.
  2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies (typically the child process) calls exec (described next) to replace itself with the new program. This is typical for programs such as shells.

Exec:

- The only way in which an executable program file on disk can be executed by Unix is for an existing process to call one of the six **exec** functions.
- **exec** replaces the current process image with the new program file, and this new program normally starts at the main function.

```
#include <unistd.h>
#include <unistd.h>
int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );
int execv (const char *pathname, char *const argv[]);
int execle (const char *pathname, const char *arg0, ...
/* (char *) 0, char *const envp[] */ );
int execve (const char *pathname, char *const argv[], char *const envp[]);
int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );
int execvp (const char *filename, char *const argv[]);
All six return: -1 on error, no return on success
```

- These functions return to the caller only if an error occurs. Otherwise, control passes to the start of the new program, normally the main function.
- Normally, only execve is a system call within the kernel and the other five are library functions that call execve.

18. Explain in detail the socket Function. Explain the two queues maintained by TCP for a listening socket . How can you increase the maximum queue size to hold more number of connections.

Socket functions –

- Socket functions are a set of programming functions or APIs (Application Programming Interfaces) provided by the operating system to enable network communication between processes over a computer network.
- These functions are commonly used in network programming to create, configure, and manage network sockets.
- Sockets provide a standard mechanism for processes on different devices to communicate over a network.

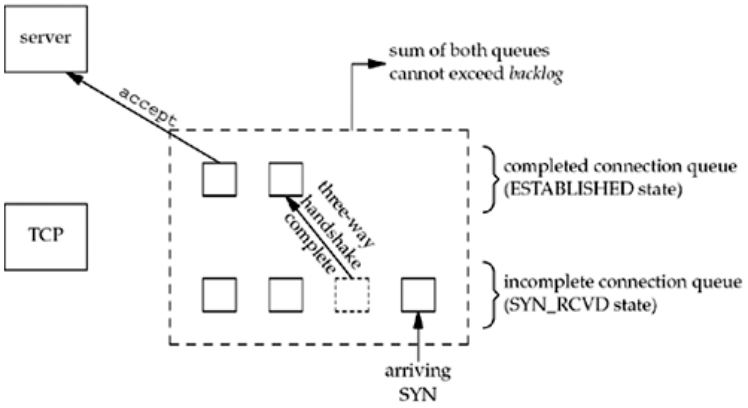
(refer next question to explain socket functions)

For a given listening socket, the kernel maintains two queues:

→ 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

→ A completed connection queue, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.

Following figure depicts these two queues for a given listening socket.



19. Discuss the following Socket functions: a) Connect Function b) bind Function
listen Function
Accept Function
close Function

a. **connect()** :
This function is used by a TCP client to establish a connection with a server. It takes three arguments: the socket descriptor, a pointer to the server's address (IP and port), and the size of the address structure. The function returns 0 on success, and -1 on failure.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```
Returns: 0 if OK, -1 on error

b. **bind()** :
This function is used by a TCP server to bind a socket to a specific address and port. It takes three arguments: the socket descriptor, a pointer to the server's address (IP and port), and the size of the address structure. The function returns 0 on success, and -1 on failure.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```
Returns: 0 if OK,-1 on error

c. **listen()** :
This function is used by a TCP server to listen for incoming connections. It takes two arguments: the socket descriptor, and the maximum number of connections that can be queued. The function returns 0 on success, and -1 on failure.

```
#include <sys/socket.h>
#int listen (int sockfd, int backlog);
```
Returns: 0 if OK, -1 on error

d. **accept()** :
This function is used by a TCP server to accept an incoming connection. It takes three arguments: the socket descriptor, a pointer to the client's address (IP and port), and the size of the address structure. The function returns a new socket descriptor on success, and -1 on failure.

```
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```
Returns: non-negative descriptor if OK, -1 on error

e. **close()** :
This function is used to close a socket and terminate a connection. It takes a single argument, the socket descriptor, and returns 0 on success and -1 on failure.

```
#include <unistd.h>
int close (int sockfd);
```
Returns: 0 if OK, -1 on error