UNIT – 5
Chapter - 1

System Protection

Protection refers to a mechanism which controls the access of programs, processes, or users to the resources defined by a computer system. We can take protection as a helper to multi programming operating system, so that many users might safely share a common logical name space such as directory or files.

Need of Protection:

- To prevent the access of unauthorized users and
- To ensure that each active programs or processes in the system uses resources only as the stated policy,
- To improve reliability by detecting latent errors.

Role of Protection:

The role of protection is to provide a mechanism that implement policies which defines the uses of resources in the computer system. Some policies are defined at the time of design of the system, some are designed by management of the system and some are defined by the users of the system to protect their own files and programs.
Every application has different policies for use of the resources and they may change over time so protection of the system is not only concern of the designer of the operating system. Application programmer should also design the protection mechanism to protect their system against misuse.

Policy is different from mechanism.

- Mechanisms determine how something will be done and policies determine what will be done.
- Policies are changed over time and place to place. Separation of mechanism and policy is important for the flexibility of the system.

Goals of Protection

- Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system *policies*, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

Principles of Protection

- The ***principle of least privilege*** dictates that programs, users, and systems be given just enough privileges to perform their tasks.
- This ensures that failures do the least amount of harm and allow the least of harm to be done.
- For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.
- Typically each user is given their own account, and has only enough privilege to modify their own files.
- The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

Domain of Protection

Various domains of protection in operating system are as follows:

1. The protection policies restrict each process's access to its resource handling. A process is obligated to use only the resources necessary to fulfill its task within the time constraints and in the mode in which it is required. It is a process's protected domain.
2. Processes and objects are abstract data types in a computer system, and these objects have operations that are unique to them.

3. Each domain comprises a collection of objects and the operations that may be implemented on them. A domain could be made up of only one process, procedure, or user. If a domain is linked with a procedure, changing the domain would mean changing the procedure ID. Objects may share one or more common operations.

Domain Structure

- A ***protection domain*** specifies the resources that a process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- An ***access right*** is the ability to execute an operation on an object.
- A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.

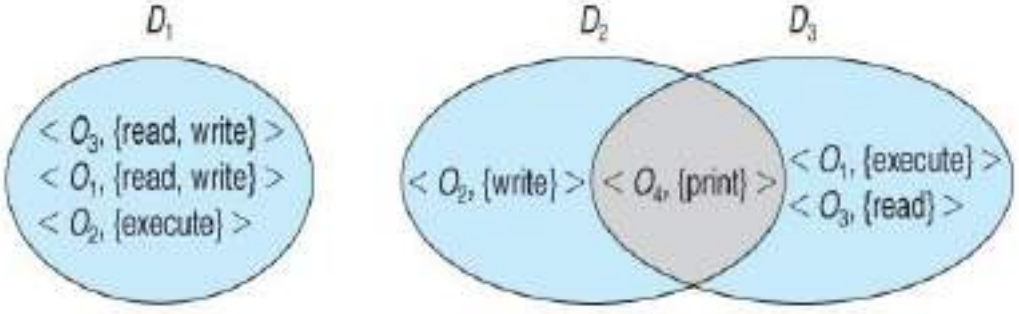


Figure 14.1 - System with three protection domains.

- The association between a process and a domain may be *static* or *dynamic*.
    - If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically.
    - If the association is dynamic, then there needs to be a mechanism for ***domain switching.***
- Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID.

An Example: UNIX

- UNIX associates domains with users.
- Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running. (and similarly for the SGID bit.) Unfortunately this has some potential for abuse.
- An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.
- Yet another alternative is to not allow the changing of ID at all. Instead, special privileged daemons are launched at boot time, and user processes send messages to these daemons when they need special tasks performed.

Access Matrix

The Access Matrix is a security model for a computer system's protection state. It is described as a matrix. An access matrix is used to specify the permissions of each process running in the domain for each object.
In which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.

- View protection as a matrix (*access matrix*)
- Rows represent domains
- Columns represent objects
- *Access(i, j)* is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix
- Can be expanded to dynamic protection
- Operations to add, delete access rights
- Special access rights:
  - *owner of* O_i
  - *copy op from* O_i *to* O_j
  - *control* – D_i *can modify* D_j *access rights*
  - *transfer – switch from domain* D_i *to* D_j

Access matrix design separates mechanism from policy

- Mechanism
  - Operating system provides access-matrix + rules
  - If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced
- Policy
  - User dictates policy
  - Who can access what object and in what mode

- Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 14.4 - Access matrix of Figure 14.3 with domains as objects.

- The ability to ***copy*** rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:

- If the asterisk is removed from the original access right, then the right is ***transferred,*** rather than being copied. This may be termed a ***transfer*** right as opposed to a ***copy*** right.
- If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a ***limited copy*** right, as shown in Figure 14.5 below:

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure 14.5 - Access matrix with *copy* rights.

- The ***owner*** right adds the privilege of adding new rights or removing existing ones:

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Figure 14.6 - Access matrix with *owner* rights.

- Copy and owner rights only allow the modification of rights within a column. The addition of ***control rights***, which only apply to domain objects, allow a process operating in one domain to affect the rights available in other domains. For example in the table below, a process operating in domain D2 has the right to control any of the rights in domain D4.

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figure 14.7 - Modified access matrix of Figure 14.4

Implementation of Access Matrix

1. Global Table

- The simplest approach is one big global table with < domain, object, rights > entries.
- Unfortunately this table is very large (even if sparse) and so cannot be kept in memory (without invoking virtual memory techniques.)
- There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

2. Access Lists for Objects

- Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.
- For efficiency a separate list of default access rights can also be kept, and checked first.

3. Capability Lists for Domains

- In a similar fashion, each row of the table can be kept as a list of the capabilities of that domain.
- Capability lists are associated with each domain, but not directly accessible by the domain or any user process.
- Capability lists are themselves protected resources, distinguished from other data in one of two ways:
    - A ***tag,*** possibly hardware implemented, distinguishing this special type of data. (other types may be floats, pointers, booleans, etc.)
    - The address space for a program may be split into multiple segments, at least one of which is inaccessible by the program itself, and used by the operating system for maintaining the process's access right capability list.

4. A Lock-Key Mechanism

- Each resource has a list of unique bit patterns, termed locks.
- Each domain has its own list of unique bit patterns, termed keys.
- Access is granted if one of the domain's keys fits one of the resource's locks.
- Again, a process is not allowed to modify its own keys.

Sockets

Sockets are a fundamental communication mechanism that allows processes (programs) running on different devices or computers to establish a network connection and exchange data. Sockets are widely used in network programming to enable communication between a client application and a server application over the internet or a local network.

A socket is a software endpoint that represents one side of a communication channel. A server socket is used by a server application to listen for incoming client connections, while a client socket is used by a client application to initiate a connection to the server.

Here's how the socket communication process typically works:

1. **Server Socket Setup:** The server creates a socket and binds it to a specific IP address and port number on the host machine. This is done to specify the interface through which the server will listen for incoming connections.
2. **Client Socket Setup:** The client creates a socket and specifies the IP address and port number of the server to which it wants to connect.
3. **Connection Establishment:** The client attempts to connect to the server by sending a connection request to the server's IP address and port number.
4. **Server Connection Acceptance:** The server's socket, which is actively listening for incoming connections, accepts the client's connection request and establishes a connection with the client socket. This creates a dedicated communication channel between the client and server.
5. **Data Exchange:** Once the connection is established, both the client and server can send and receive data through their respective sockets. Data can be transmitted in both directions simultaneously, allowing for real-time communication.
6. **Connection Termination:** When the communication is complete or either party wants to end the connection, they can close their sockets, terminating the connection.

Sockets can operate using different communication protocols, such as TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). TCP ensures reliable, ordered, and error-checked delivery of data, while UDP provides a faster, connectionless, and less reliable communication method.

Socket programming involves using specific libraries or APIs provided by the programming language to work with sockets. For example, in Python, you can use the socket module to create and manage sockets, while other programming languages have their own socket libraries.

Socket Connections

Socket connections are a fundamental concept in computer networking, allowing processes running on different devices or computers to communicate and exchange data over a network. Sockets provide a bi-directional communication channel that enables applications to send and receive data reliably and efficiently.

The key components of a socket connection are the client socket and the server socket. Here's a breakdown of how socket connections work:

1. Client Socket:
    - The client socket is created by a client application when it wants to communicate with a server.
    - The client socket specifies the IP address and port number of the server it wants to connect to.

- Once the connection is established, the client socket can send data to and receive data from the server socket.

2. Server Socket:
    - The server socket is created by a server application to listen for incoming client connections.
    - The server socket is bound to a specific IP address and port number on the server machine, specifying the interface through which the server will accept connections.
    - When a client attempts to connect, the server socket accepts the connection and establishes a dedicated communication channel with the client socket.
3. Establishing a Connection:
    - To initiate a connection, the client sends a connection request to the server's IP address and port number.
    - The server's socket, which is actively listening for incoming connections, accepts the client's request, creating a connection.
4. Data Exchange:
    - Once the connection is established, both the client and server can send and receive data through their sockets.
    - Data can be transmitted in both directions simultaneously, enabling real-time communication.
5. Protocol and Data Handling:
    - Sockets can use different communication protocols, such as TCP or UDP, depending on the requirements of the application.
    - TCP (Transmission Control Protocol) ensures reliable, ordered, and error-checked delivery of data. It is commonly used for applications that require data integrity and guaranteed delivery.

- UDP (User Datagram Protocol) is connectionless and provides faster, lower overhead communication. It is suitable for applications where speed is crucial, and some data loss is acceptable.

6. Connection Termination:
    - When the communication is complete or either party wants to end the connection, they can close their sockets, terminating the connection.
    - Properly closing sockets is essential to release network resources and ensure that the connection is cleanly terminated.

Socket connections are extensively used in various networked applications, including web browsing, email communication, file transfers, online gaming, video streaming, and more. Socket programming is supported by most programming languages, and each language has its own set of libraries or APIs to handle socket communication.

Socket Attributes

In the context of socket programming, socket attributes refer to the properties or settings associated with a socket object that can be used to control its behavior and characteristics. These attributes provide developers with the flexibility to customize socket communication according to their specific requirements. The exact set of attributes may vary depending on the programming language or library being used, but some common attributes include:

1. **IP Address and Port Number:** Every socket is associated with an IP address and a port number. For server sockets, the IP address and port number represent the local address the server is bound to. For client sockets, they define the destination server's address.
2. **Socket Type:** Sockets can be of different types, such as stream sockets (e.g., TCP) or datagram sockets (e.g., UDP). The type determines the communication mode and reliability of the socket.
3. **Protocol:** The communication protocol used by the socket, such as TCP, UDP, or others. The choice of the protocol affects how data is transmitted and received.
4. **Blocking/Non-blocking Mode:** In blocking mode, socket operations (e.g., sending or receiving data) will wait until the operation is completed. In non-blocking mode, these operations return immediately, and the application needs to handle any potential delays or errors.
5. **Timeout:** A timeout value specifies the maximum time a socket operation can wait before returning an error if no data is available or the connection cannot be established.
6. **Buffer Size:** The buffer size refers to the amount of memory allocated for data transmission and reception. Larger buffer sizes can increase performance but may also consume more memory.
7. **Socket Options:** Socket options are settings that control various aspects of socket behavior. Examples include setting socket-level timeouts, enabling or disabling certain features, and specifying the size of the receive and send buffers.
8. **Multicast Options (for UDP):** For UDP sockets, multicast options allow a socket to participate in multicast group communication. This allows a single datagram to be sent to multiple recipients.
9. **Keep-Alive:** The keep-alive attribute, if enabled, allows the socket to send periodic keep-alive messages to check if the connection is still active.
10. **Socket Family:** The socket family determines the address format used for the socket. Common families include IPv4, IPv6, and Unix domain sockets.

These attributes can be set and modified using specific functions or methods provided by the socket library or programming language. It's essential for developers to understand and properly configure these attributes to establish reliable and efficient communication between client and server applications.

Socket Addresses

In socket programming, a socket address, also known as a network address, is used to identify and locate a networked device or application within a network. It consists of two main components: an IP address and a port number. Together, these components uniquely define the endpoint of a communication channel, allowing data to be sent to and received from a specific process on a specific device.

1. IP Address:
    - An IP (Internet Protocol) address is a numerical label assigned to each device connected to a computer network that uses the Internet Protocol for communication. It serves as the device's identifier on the network.
    - IPv4 addresses are written in the format of four decimal numbers separated by dots (e.g., 192.168.0.1).
    - IPv6 addresses are longer and written in a hexadecimal format (e.g., 2001:0db8:85a3:0000:0000:8a2e:0370:7334).

2. Port Number:
    - A port number is a 16-bit unsigned integer (ranging from 0 to 65535) used to identify a specific process or service running on a device. It allows multiple services to operate on the same device simultaneously.
    - Well-known port numbers (0 to 1023) are reserved for standard services (e.g., HTTP uses port 80, HTTPS uses port 443), while registered (1024 to 49151) and dynamic (49152 to 65535) ports are used for various applications.

When combined, the IP address and port number form a socket address, uniquely identifying a specific network endpoint. In the case of client-server communication:

- The client needs the server's socket address to connect to it. The server socket address consists of the server's IP address and the port number it is listening on for incoming connections.
- The server uses its own IP address and the port number assigned to the client's connection to send data back to the specific client.

For example, a server socket address could be represented as "192.168.0.1:8080," where "192.168.0.1" is the server's IP address, and "8080" is the port number. The client uses this socket address to connect to the server and initiate communication.

Socket addresses are essential for establishing connections in client-server architectures and enabling data exchange across different devices on a network. Socket programming libraries and APIs provide functions to work with socket addresses and handle communication between processes on different devices using these addresses.

Socket – connect( )

In socket programming, the connect() function is used by a client to establish a network connection to a server. This function is typically employed in the context of TCP (Transmission Control Protocol) sockets, which provide reliable, stream-oriented communication.

The connect() function performs the following steps:

1. Create a Socket:

Before attempting to connect to a server, the client creates a socket using the socket() function. This creates a communication endpoint that will be used to send and receive data over the network.

2. Set Up Server Socket Address:

The client needs to know the server's socket address (IP address and port number) to establish a connection. The server socket address specifies where the server is listening for incoming client connections.

3. Call the connect() Function:

Once the client has the server's socket address, it calls the connect() function, passing the socket descriptor (a unique identifier for the client's socket) and the server's socket address as arguments.

The connect() function tries to establish a connection to the server using the specified address.

4. Three-Way Handshake:

The connect() function initiates a three-way handshake with the server to establish a TCP connection. This involves a series of messages exchanged between the client and the server to ensure both parties agree to establish the connection.

The three steps of the handshake are: SYN (synchronize), SYN-ACK (synchronize-acknowledge), and ACK (acknowledge).

5. Connection Established:

If the server accepts the client's connection request, and the three-way handshake completes successfully, the connect() function returns, indicating that the connection is established.

The client can now start sending data to the server or receiving data from it through the established connection.

6. Handle Connection Errors:

The connect() function may return an error if it cannot establish a connection to the server. Common reasons for connection failure include the server not being reachable, the server socket being unavailable, or the network being down.

Here's a simplified example of how the connect() function might be used in C:

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- sockfd -
  - This is the file descriptor of the socket that you want to use for the connection.
  - The sockfd should be a valid socket descriptor obtained from a previous call to the socket() function.
- addr -
  - A pointer to a struct sockaddr representing the address of the server to which you want to connect.
  - The struct sockaddr is a generic structure used to hold various types of socket addresses (IPv4, IPv6, Unix domain, etc.).
  - To use the struct sockaddr_in structure for IPv4 addresses, you'll need to cast the addr pointer appropriately (e.g., (struct sockaddr*)&server_address).
- addrlen -
  - The size of the addr structure in bytes.
  - It is essential to provide the correct size of the addr structure to prevent buffer overflows and ensure proper address resolution.
  - For IPv4 addresses, the size would be sizeof(struct sockaddr_in).

The connect() function is used to establish a TCP connection between the client and the server. Before calling this function, you should have already created a socket using the socket() function, set up the server's address details in a struct sockaddr, and filled in the appropriate socket address family, IP address, and port number.

Socket - bind( )

In C socket programming, the bind() function is used to associate a socket with a specific network address, including an IP address and a port number. This is typically done on the server side to set up the server's endpoint, allowing clients to connect to it.

Here's the syntax of the bind() function:

```
#include <sys/socket.h>

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

let's explain each parameter:

- sockfd -
  - This is the file descriptor of the socket you want to bind.
  - The sockfd should be a valid socket descriptor obtained from a previous call to the socket() function.
- addr -
  - A pointer to a struct sockaddr representing the network address you want to bind the socket to.
  - The struct sockaddr is a generic structure used to hold various types of socket addresses (IPv4, IPv6, Unix domain, etc.).
  - To use the struct sockaddr_in structure for IPv4 addresses, you need to cast the addr pointer appropriately (e.g., (struct sockaddr*)&server_address).
- addrlen -
  - The size of the addr structure in bytes.
  - It is essential to provide the correct size of the addr structure to prevent buffer overflows and ensure proper address resolution.
  - For IPv4 addresses, the size would be sizeof(struct sockaddr_in).

The purpose of the bind() function is to set the local address of the socket, allowing the operating system to direct incoming connection requests to the specified IP address and port number.

Socket - listen( )

In C socket programming, the listen() function is used on the server side to make a bound socket actively listen for incoming client connections. After calling the bind() function to associate a socket with a specific network address (IP address and port number), the listen() function prepares the socket to accept incoming connections from clients.

Here's the syntax of the listen() function:

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Let's explain each parameter:

- sockfd -
    - This is the file descriptor of the socket that you want to set in the listening mode.
    - The sockfd should be a valid socket descriptor obtained from a previous call to the socket() function and a subsequent call to the bind() function.
- backlog -
    - The backlog parameter specifies the maximum number of pending client connections that can be queued up before the server starts rejecting new connections.
    - The backlog parameter allows you to control the size of the connection queue when the server is busy and cannot handle new incoming connections immediately.
    - A typical value for the backlog parameter is often set to 5 or more, depending on the expected load and the nature of the server application.

After calling listen(), the socket becomes a "passive" socket, waiting for incoming connection requests. It does not mean that the server is actively handling clients yet; instead, it's ready to accept connections and queue them up in the connection backlog until the server has the capacity to process them.

Socket – accept( )

In C socket programming, the accept() function is used by a server to accept incoming client connections. After calling the listen() function to put the server's bound socket into a listening state, the server can use the accept() function to establish a new socket connection with a client when the client tries to connect.

Here's the syntax of the accept() function:

```
#include <sys/types.h>

#include <sys/socket.h>

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

let's explain each parameter:

- sockfd -
    - This is the file descriptor of the listening socket.
    - The sockfd should be a valid socket descriptor that has been created using the socket() function and bound to a local address using the bind() function, and then put in the listening state using the listen() function.
- addr -
    - A pointer to a struct sockaddr that will be filled in by the accept() function with the client's address information after a successful connection.
    - The struct sockaddr is a generic structure used to hold various types of socket addresses (IPv4, IPv6, Unix domain, etc.).
    - To receive the client's address in a specific format (e.g., IPv4 with struct sockaddr_in), you need to cast the addr pointer appropriately (e.g., (struct sockaddr*)&client_address).
- addrlen -
    - A pointer to a socklen_t variable that specifies the size of the addr structure.
    - On input, it indicates the size of the buffer pointed to by addr. On output, it is

updated with the actual size of the client's address information.
- The value of addrlen must be initialized to the size of the buffer pointed to by addr.

The accept() function blocks until an incoming connection request is received. When a client attempts to connect to the server, the accept() function creates a new socket specifically for that client and establishes a dedicated communication channel between the server and the client. The new socket descriptor returned by accept() is the one used for communication with that specific client.

Socket Communications

Socket communication is a fundamental concept in computer networking, allowing processes running on different devices or computers to exchange data over a network. Sockets provide a bidirectional communication channel between a client and a server, enabling them to send and receive data in real-time.

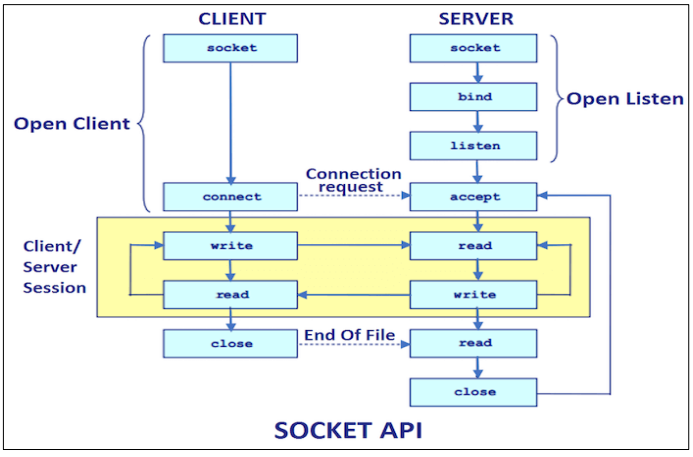
Socket communication typically involves two main protocols: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). Each protocol serves different communication needs:

1. TCP (Transmission Control Protocol):
   - TCP is a connection-oriented, reliable, and stream-oriented protocol.
   - It ensures that data is delivered accurately and in the correct order. If any data packets are lost during transmission, TCP automatically retransmits them.
   - TCP performs flow control and congestion control to manage the data flow and prevent network congestion.
   - It is ideal for applications where data integrity and reliable delivery are essential, such as web browsing, email, file transfer (FTP), and database communication.
2. UDP (User Datagram Protocol):
   - UDP is a connectionless, unreliable, and datagram-oriented protocol.
   - It does not guarantee the delivery or ordering of data packets. If any packets are lost, there is no automatic retransmission.
   - UDP is faster and has lower overhead compared to TCP due to its simplicity and lack of connection setup.
   - It is suitable for applications that prioritize speed and real-time responsiveness over data reliability, such as video streaming, online gaming, and Voice over IP (VoIP).

The typical steps involved in socket communication are as follows:

1. Socket Creation:

Both the client and server create sockets using the socket() function. The client creates a client socket, while the server creates a server socket.



2. Binding (Server-side only):

The server binds the server socket to a specific IP address and port number using the bind() function. This allows

clients to know where to connect.

3. Listening (Server-side only):

The server uses the listen() function to make the server socket ready to accept incoming client connections.

4. Connection Establishment (Client-side only):

The client uses the connect() function to establish a connection to the server's IP address and port number.

5. Accepting Connections (Server-side only):

The server uses the accept() function to accept incoming client connections after the clients attempt to connect.

6. Data Exchange:

Once the connection is established, both the client and server can use the socket to send and receive data using send() and recv() functions (for TCP) or sendto() and recvfrom() functions (for UDP).

7. Connection Termination:

When the communication is complete or either party wants to end the connection, they can close their sockets using the close() function.

Socket communication is widely used for various networked applications, including web browsing, email, video streaming, online gaming, instant messaging, and more. It provides the backbone for communication in today's interconnected world.

//server.c (Client –Server communication using Sockets)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCKET_PATH "/tmp/mysocket.sock"
#define BUFFER_SIZE 256

int main() {
    int serverSocket, clientSocket;
    struct sockaddr_un serverAddress, clientAddress;
    char buffer[BUFFER_SIZE];
    int bytesReceived;

        // Create a server socket
    serverSocket = socket(AF_UNIX, SOCK_STREAM, 0);
    if (serverSocket == -1) {
```

```
        perror("socket");
        exit(EXIT_FAILURE);
    }

        // Set the server address structure
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sun_family = AF_UNIX;
    strncpy(serverAddress.sun_path, SOCKET_PATH, sizeof(serverAddress.sun_path) - 1);

// Bind the server socket to a file
    if (bind(serverSocket, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) == -1) {
        perror("bind");
        exit(EXIT_FAILURE);
    }

// Listen for client connections
    if (listen(serverSocket, 1) == -1) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    printf("Server is running and listening for connections...\n");

        // Accept client connections
    socklen_t clientAddressLength = sizeof(clientAddress);
    clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddress, &clientAddressLength);

    if (clientSocket == -1) {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    printf("Client connected.\n");

// Receive messages from the client
    while ((bytesReceived = recv(clientSocket, buffer, BUFFER_SIZE, 0)) > 0) {
        buffer[bytesReceived] = '\0';
        printf("Received from client: %s\n", buffer);
    }
    if (bytesReceived == -1) {
        perror("recv");
        exit(EXIT_FAILURE);
    }

// Close the client socket
    close(clientSocket);

        // Close the server socket and remove the socket file
    close(serverSocket);
    unlink(SOCKET_PATH);
    return 0;
 }
```

Explanation:

Server Program (server.c) Flow:

1. Create a server socket using the socket() function:
    - Arguments:
        - AF_UNIX: This specifies that we are creating a Unix domain socket.
        - SOCK_STREAM: This specifies that we are creating a stream socket for reliable, two-way, connection-based communication.
        - 0: The protocol, which is 0 for the default protocol based on the socket types.
    - Returns:
        - The socket file descriptor (an integer) representing the server socket.
2. Set up the server address structure using memset() and strncpy() functions:
    - memset(): This clears the serverAddress structure to ensure there are no garbage values.
    - strncpy(): This copies the SOCKET_PATH constant (which contains the socket file path) into serverAddress.sun_path.
3. Bind the server socket to a file using the bind() function:
    - Arguments:
        - serverSocket: The server socket file descriptor.
        - (struct sockaddr*)&serverAddress: A pointer to the server address structure.
        - sizeof(serverAddress): The size of the server address structure.
    - The bind() function associates the socket with the specified address (in this case, the Unix domain socket file path).
4. Listen for incoming client connections using the listen() function:
    - Arguments:
        - serverSocket: The server socket file descriptor.
        - 1: The maximum number of queued connections that can be waiting for acceptance.
    - The listen() function prepares the socket to accept incoming client connections.
5. Accept client connections using the accept() function:
    - Arguments:
        - serverSocket: The server socket file descriptor.
        - (struct sockaddr*)&clientAddress: A pointer to the client address structure.
        - &clientAddressLength: A pointer to the length of the client address structure.
        - The accept() function blocks until a client connection is established.
        - When a client connects, the function returns a new socket file descriptor (clientSocket) representing the connection to that client.
6. Receive messages from the client using the recv() function:
    - Arguments:
        - clientSocket: The client socket file descriptor.
        - buffer: A character buffer to store the received data.
        - BUFFER_SIZE: The maximum number of bytes to receive at a time.
        - 0: Flags, which in this case, is set to 0 (no special flags).
    - The recv() function blocks until data is received from the client. It then stores the received data in the buffer and returns the number of bytes received.
7. Close the client socket and the server socket using the close() function:

- Arguments:
    - clientSocket: The client socket file descriptor to be closed.
    - serverSocket: The server socket file descriptor to be closed.
- The close() function releases the resources associated with the respective sockets.

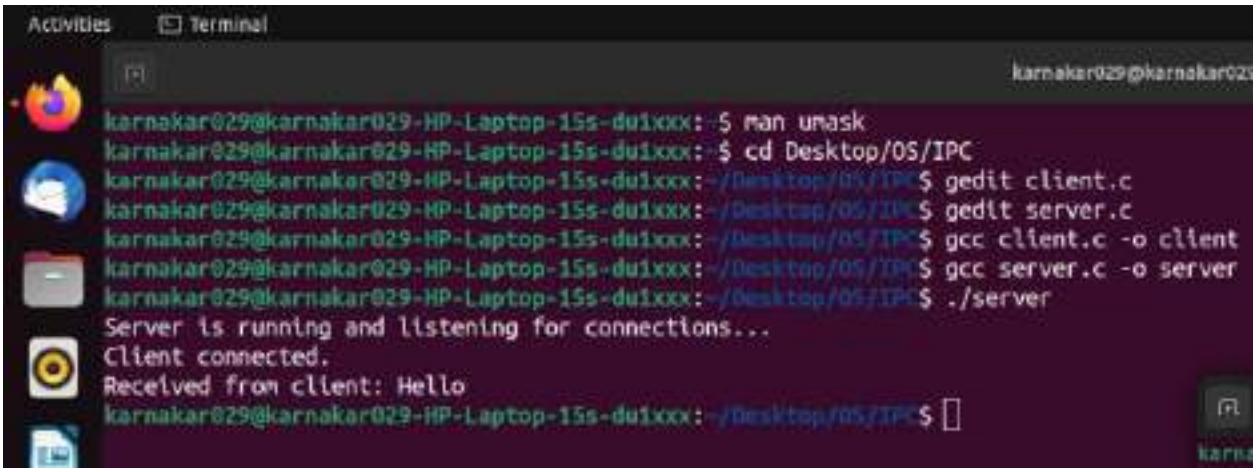
8. Unlink (remove) the socket file using the unlink() function:
    - Argument:
        - SOCKET_PATH: The socket file path.
        - The unlink() function removes the socket file from the file system.

// Server Program (server.c):

1. Include necessary header files:
    - stdio.h: Standard I/O functions (e.g., printf).
    - stdlib.h: Standard library functions (e.g., exit).
    - string.h: String manipulation functions (e.g., memset, strncpy).
    - unistd.h: UNIX standard functions (e.g., close).
    - sys/socket.h: Definitions for socket-related functions and structures.
    - sys/un.h: Definitions for UNIX domain sockets.
2. Define constants and buffer size:
    - SOCKET_PATH: A constant representing the file path for the UNIX domain socket.
    - BUFFER_SIZE: The size of the buffer used for sending and receiving messages.
3. main function:
    - Create necessary variables: serverSocket, clientSocket, serverAddress, clientAddress, buffer, and bytesReceived.
    - Create a server socket using socket() and handle errors if the socket creation fails.
    - Set up the server address structure using memset() and strncpy().
    - Bind the server socket to the specified file using bind().
    - Listen for incoming client connections using listen().
    - Accept a client connection using accept() and handle errors if the connection fails.
    - Receive messages from the client using recv() in a loop until the connection is closed or an error occurs.
    - Display the received messages on the server's console.
    - Close the client socket and the server socket using close().
    - Unlink (remove) the socket file from the file system using unlink().

//Client Program (client.c):

1. Include necessary header files (same as in the server program).
    - Define constants and buffer size (same as in the server program).
2. main function:
    - Create necessary variables: clientSocket, serverAddress, buffer, and bytesSent.
    - Create a client socket using socket() and handle errors if the socket creation fails.
    - Set up the server address structure using memset() and strncpy().
    - Connect to the server using connect() and handle errors if the connection fails.
    - In a loop, prompt the user to enter a message, send it to the server using send(), and handle errors if the sending fails.
    - If the user enters 'q', break the loop and close the client socket using close().


Activities
Terminal
karnakar029@karnakar029-HP-Laptop-15s-du1xxx: $ man umask
karnakar029@karnakar029-HP-Laptop-15s-du1xxx: $ cd Desktop/OS/IPC
$ gedit client.c
$ gedit server.c
$ gcc client.c -o client
$ gcc server.c -o server
$ ./server
Server is running and listening for connections...
Client connected.
Received from client: Hello


karnakar029@karnakar029-HP-Laptop-15s-du1xxx: ~/Desktop...
karnakar029@karnakar029-HP-Laptop-15s-du1xxx:~$ cd Desktop/OS/IPC
karnakar029@karnakar029-HP-Laptop-15s-du1xxx:~/Desktop/OS/IPC$ ./client
Connected to the server.
Enter a message (or 'q' to quit): Hello
Enter a message (or 'q' to quit): q
karnakar029@karnakar029-HP-Laptop-15s-du1xxx:~/Desktop/OS/IPC$

Introduction of System function

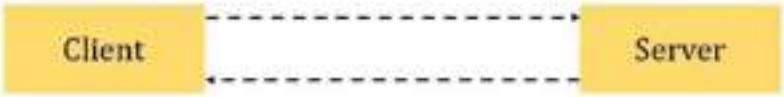
Socket

- What is Socket?
- What is Socket Programming?
- Which functions are used for socket programming?

- Socket is an endpoint of a 2-way communication between programs running on the network.
- Socket programming is a way of connecting two nodes on a network to communicate with each other.
- The main functions in <sys/socket.h> are:
  socket( )
  bind( )
  listen( )
  connect( )
  accept( )
  send( )/recv( )/read( )/write ( )/sendto( )/recvfrom( )
  close( )

Socket

- What is Socket?
- What is Socket Programming?
- Which functions are used for socket programming?
- Categorized function between Server and Client.



Client Function	Server Function
socket( )	
connect( )	bind ( )
	listen ( )
	accept ( )
send( ) / recv( ) / sendto ( ) / recvfrom ( )	
close( )	

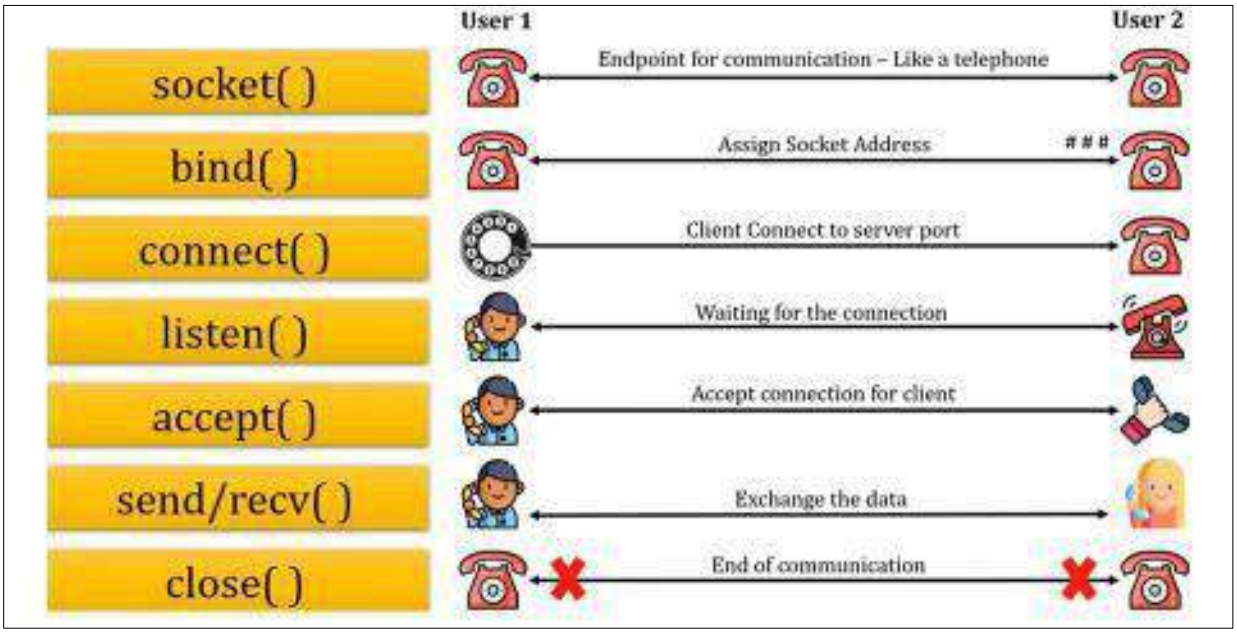
Socket

- What is Socket?
- What is Socket Programming?
- Which functions are used for socket programming?
- Categorized function between Server and Client.

Understanding System Function

NaaaaH...!!!!!

Client = User 1
Server = User 2



System function for Socket Programming

socket( )

socket( ) – A Connection Endpoint

- **Purpose:** It creates socket.
- **Syntax:** int socket(int family, int type, int protocol)

Family	Description
AF_INET	IPV4
AF_INET6	IPV6

Type	Description
SOCK_STREAM	Stream Socket
SOCK_DGRAM	Datagram Socket

Type	Description
IPPROTO_TCP	TCP Protocol
IPPROTO_UDP	UDP Protocol

- **Example:**

```
int socket(AF_INET, SOCK_STREAM, 0);
```

socket( )

bind( )

bind( ) – Attaching to an IP and Port

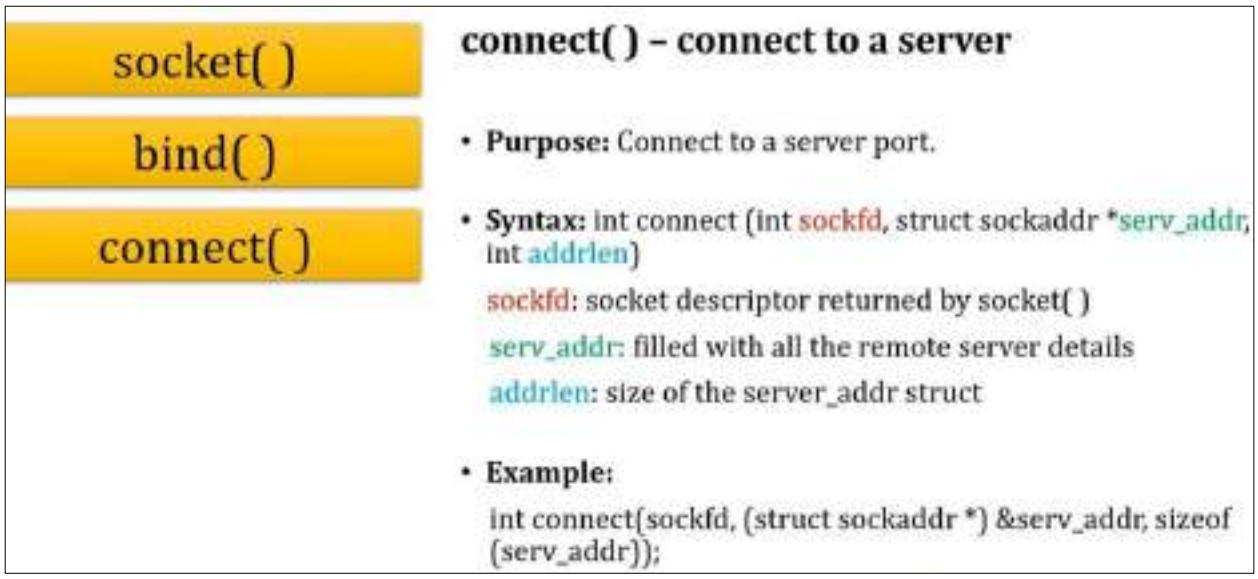
- **Purpose:** Attach itself to a specific port and IP address.
- **Syntax:**

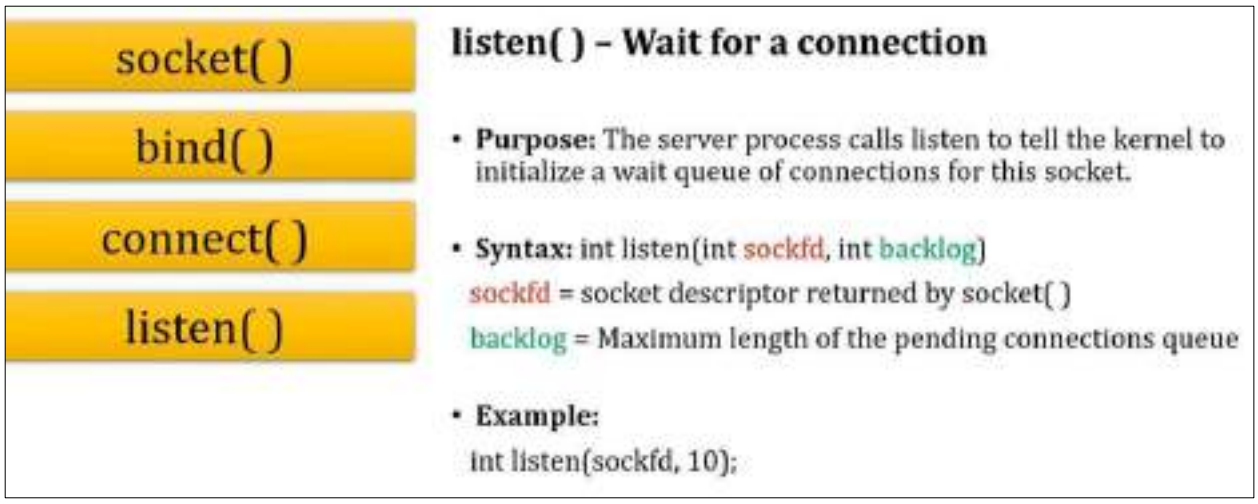
```
int bind (int sockfd, struct sockaddr *serv_addr, int addrlen)
sockfd = socket descriptor returned by socket( )
serv_addr = It contains server IP address and port.
addrlen = length of the address in bytes.
```

- **Example:**

```
struct sockaddr_in serv_addr;
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);// define port number assigned

int bind (sockfd, (struct sockaddr *) &serv_addr,
sizeof(serv_addr));
```


socket( )
bind( )
connect( )
connect( ) – connect to a server
• Purpose: Connect to a server port.
• Syntax: int connect (int sockfd, struct sockaddr *serv_addr, int addrlen)
sockfd: socket descriptor returned by socket( )
serv_addr: filled with all the remote server details
addrlen: size of the server_addr struct
• Example:
int connect(sockfd, (struct sockaddr *) &serv_addr, sizeof (serv_addr));


socket( )
bind( )
connect( )
listen( )
listen( ) – Wait for a connection
• Purpose: The server process calls listen to tell the kernel to initialize a wait queue of connections for this socket.
• Syntax: int listen(int sockfd, int backlog)
sockfd = socket descriptor returned by socket( )
backlog = Maximum length of the pending connections queue
• Example:
int listen(sockfd, 10);

socket( )

bind( )

connect( )

listen( )

accept( )

accept( ) – A new connection !

- **Purpose:** accept new connections from new clients
- **Syntax:** int accept (int sockfd, struct sockaddr *cli_addr, int addrlen)

  sockfd = socket descriptor returned by socket( )

  cli_addr = will hold the new client's information when accept returns

  addrlen= size of client address
- **Example:**

```
struct sockaddr_in cli_addr;
int len = sizeof(client);
int accept(sockfd, (struct sockaddr *) &cli_addr, &len);
```

socket( )

bind( )

connect( )

listen( )

accept( )

send( )/recv( )

send( ) / recv( ) – Stream Socket

- **Purpose:** Send, Recv functions are used to send and receive data over stream socket.
- **Syntax:** int send(int sockfd, void *msg, int len, int flags)

  int recv(int sockfd, void *msg, int len, int flags)

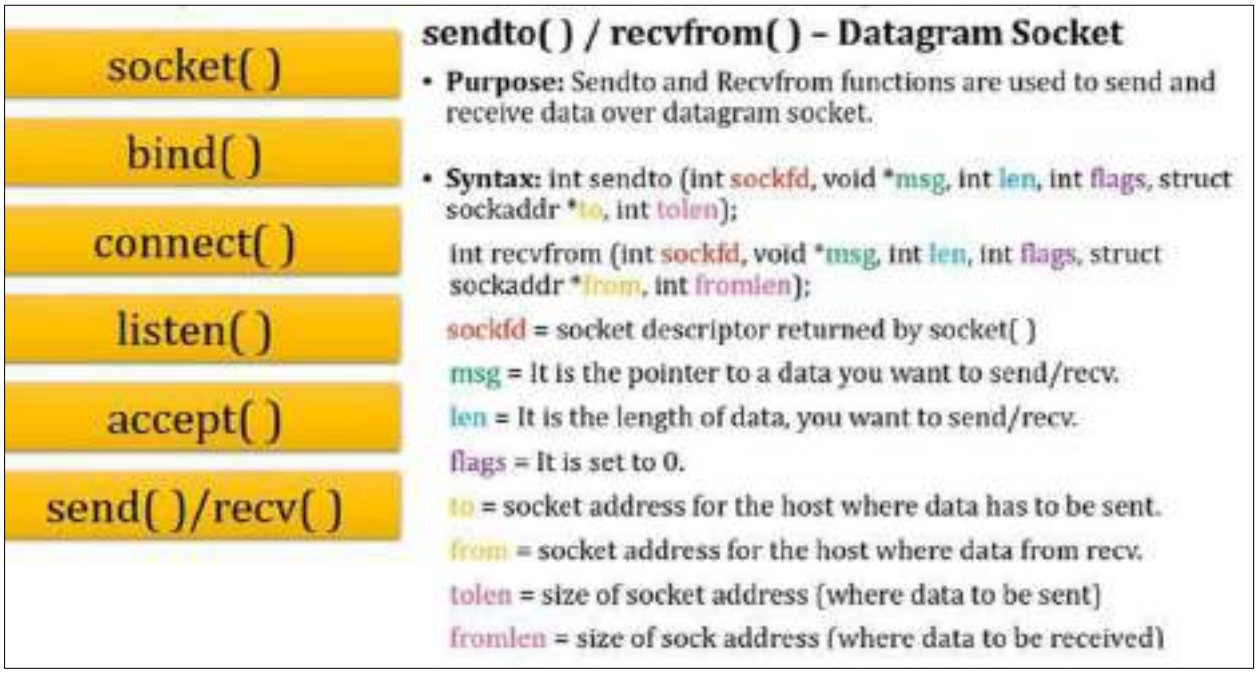
  sockfd = socket descriptor returned by socket( )

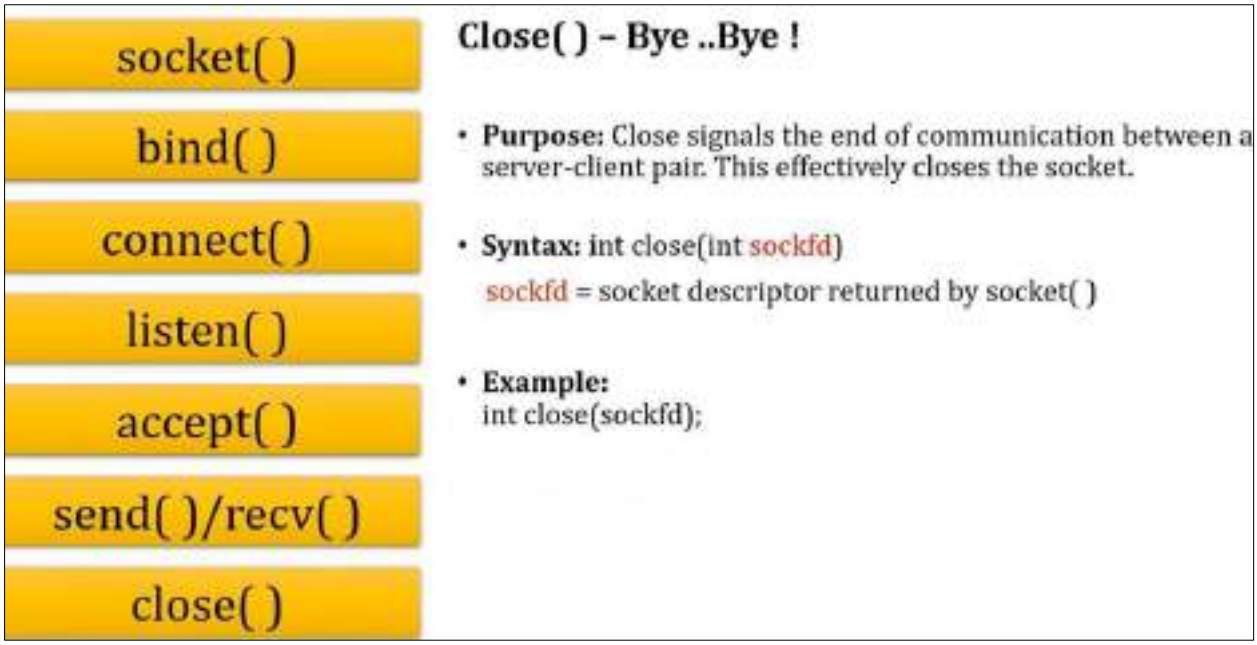
  msg = It is the pointer to a data you want to send/recv.

  len = It is the length of data, you want to send/recv.

  flags = It is set to 0.
- **Example:**

```
char send_msg[1024], recv_msg[1024];
int sent_bytes, recv_bytes;
sent_bytes = int send(sockfd, send_msg, 1024, 0);
recv_bytes= int recv (sockfd, recv_msg, 1024, 0);
```


socket( )
bind( )
connect( )
listen( )
accept( )
send( )/recv( )
sendto( ) / recvfrom( ) – Datagram Socket
• Purpose: Sendto and Recvfrom functions are used to send and receive data over datagram socket.
• Syntax: int sendto (int sockfd, void *msg, int len, int flags, struct sockaddr *to, int tolen);
int recvfrom (int sockfd, void *msg, int len, int flags, struct sockaddr *from, int fromlen);
sockfd = socket descriptor returned by socket( )
msg = It is the pointer to a data you want to send/recv.
len = It is the length of data, you want to send/recv.
flags = It is set to 0.
to = socket address for the host where data has to be sent.
from = socket address for the host where data from recv.
tolen = size of socket address (where data to be sent)
fromlen = size of sock address (where data to be received)


socket( )
bind( )
connect( )
listen( )
accept( )
send( )/recv( )
close( )
Close( ) – Bye ..Bye !
• Purpose: Close signals the end of communication between a server-client pair. This effectively closes the socket.
• Syntax: int close(int sockfd)
sockfd = socket descriptor returned by socket( )
• Example:
int close(sockfd);