

IPC

# Inter-process Communication in Linux (IPC)

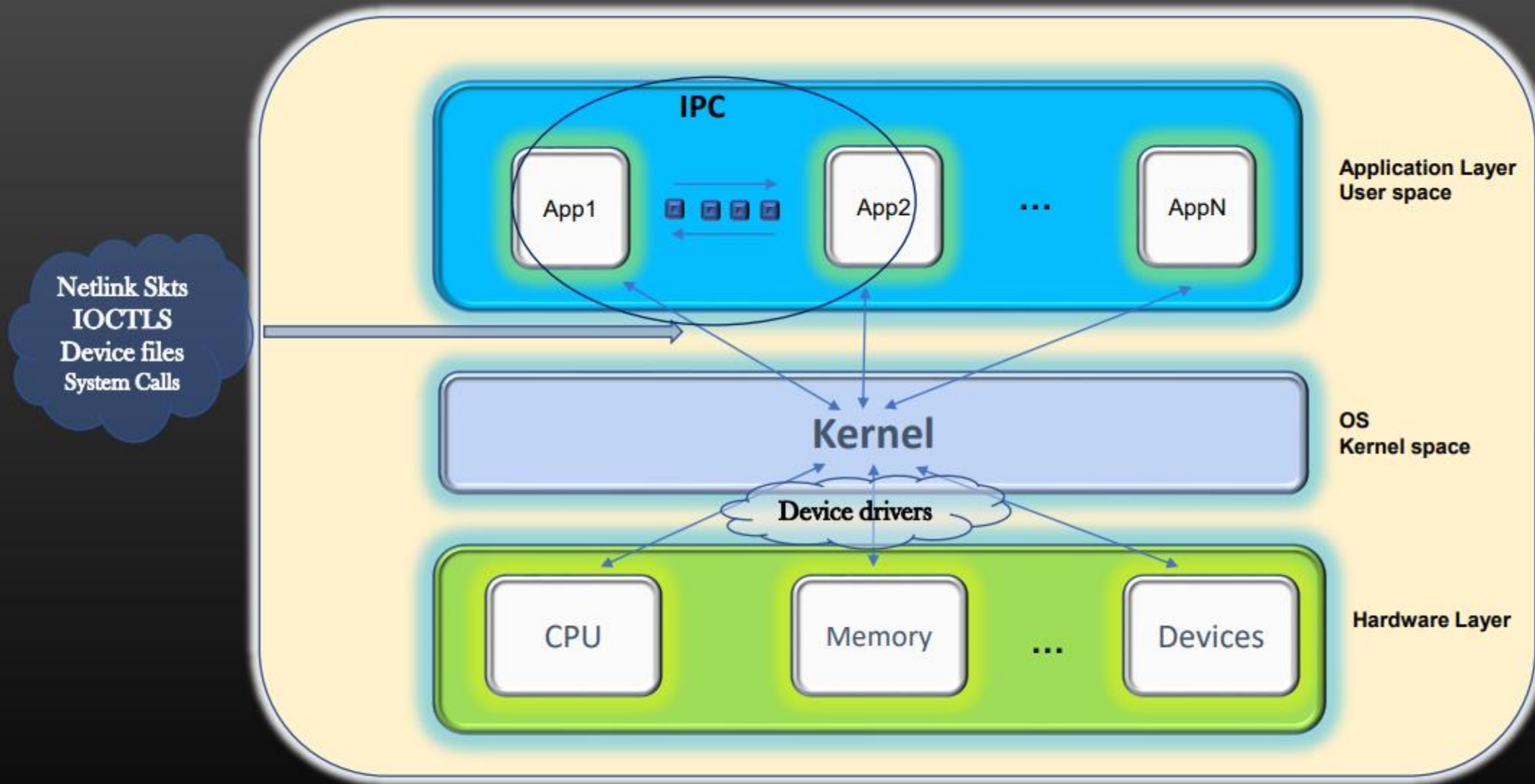
1. What is IPC ?
2. Why we need IPC ?
3. Various ways to implement IPC in Linux
  - Unix Sockets
  - Message Queues
  - Shared Memory
  - Pipes
  - Signals
4. Demonstration & Code Walk
5. Design discussions on IPC
6. Project on IPC



## What is IPC and Why we need it ?

- IPC is a mechanism using which two or more process running on the SAME machine exchange their personal data with each other
- Communication between processes running on different machines are not termed as IPC
- Processes running on same machine, often need to exchange data with each other in order to implement some functionality
- Linux OS provides several mechanisms using which user space processes can carry out communication with each other, each mechanism has its own pros and cons
- In this course, we will explore what are the various ways of carrying out IPC on Linux OS
- The IPC techniques maps well to other platforms such as windows/MAC OS etc, conceptually same.

## What is IPC and Why we need it ?



*Computer Architecture*

## IPC techniques

- There are several ways to carry out IPC
- We shall explore each one of it, and try to analyze the pros and cons of each
- Expect several Questions on IPC in technical interviews
- IPC Techniques :
  1. Using Unix Domain sockets
  2. Using Network Sockets (Not covered in this course)
  3. Message Queues
  4. Shared Memory
  5. Pipes
  6. Signals

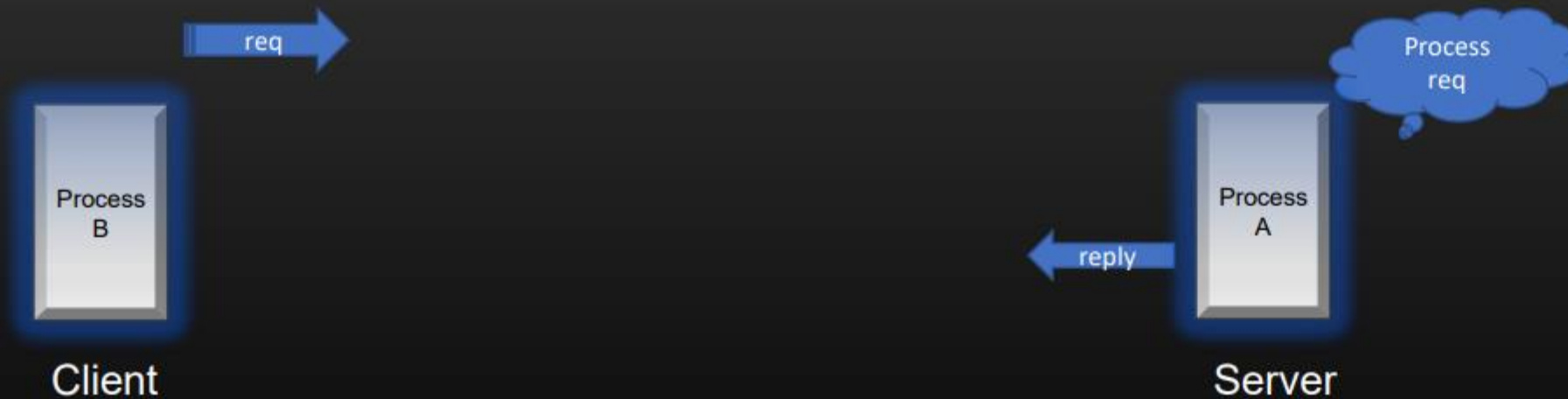


# What is a Process ?

- Whenever you write an application and execute it on a machine, it runs as a Process
- Even your “Hello-World” program runs as a process
- Machine runs several processes – *User process* Or *System Processes* at the same time
- Eg:
  - *User Process*
    - Browsers, MS office, IMs, Games, Video editing tools etc , Or any software you run on your OS
  - *System Processes*
    - Memory Manager, Disk Manager, OS Services , Schedulers etc

# Terminology

- *Communicating Processes can be classified into two categories :*
  - *Server Process*
  - *Client process*
- *Server* is any process which receives the communication request, process it and returns back the result
- *Client* is any process which initiates the communication request
- *Server* never initiates the communication



# Communication types

- Communication between two processes can be broadly classified as
  - *STREAM Based communication*
  - *DATAGRAM based communication*

## *STREAM Based communication*

1. Connection Oriented : Dedicated connection between A & B
2. Dedicated connection needs to establish first before any actual data exchange
3. Duplex communication and Reliable
4. B can send continuous stream of bytes of data to A, analogous to flow of water through a pipe
5. Data arrives in same sequence
6. Should be used where, Receiver can tolerate a LAG but not packet loss
7. Eg : Downloading an Audio Or a Movie Or software stored on a disk on a server, Emails, Chat messages etc
8. Famous Standard Network Protocol which provides stream based communication is TCP Protocol





# Communication types

- Communication between two processes can be broadly classified as

- STREAM Based communication*
- DATAGRAM based communication*

## *DATAGRAM based communication*

1. No Dedicated connection between A & B
2. Data is sent in small chunks Or units, No continuous Stream of bytes
3. Duplex Communication and Unreliable
4. Data may arrive out of sequence
5. Should be used where recipient can tolerate a packet loss, but not a log
6. LIVE Video/Audio conferences
7. Famous Standard Network Datagram based protocol - UDP

Check Resource section for a good stack overflow discussion !



## IPC Techniques – Sockets

- Unix/Linux like OS provide Socket Interface to carry out communication between various types of entities
- The Socket Interface are a bunch of Socket programming related APIs
- We shall be using these APIs to implement the Sockets of Various types
- In this course We shall learn how to implement Two types of :
  - **Unix Domain Sockets**
    - IPC between processes running on the same System
  - **Network Sockets**
    - Communication between processes running on different physical machines over the network
- Let us First cover how Sockets Works in General, and then we will see how socket APIs can be used to implement a specific type of Communication. Let us first Build some background . . .

- Socket programming Steps and related APIs

- Steps :

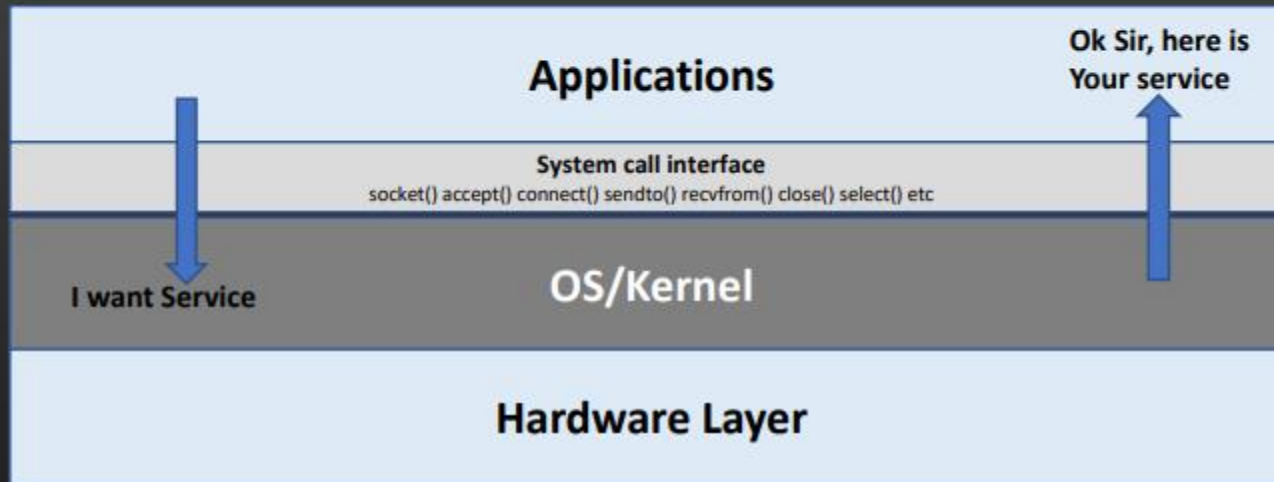
1. Remove the socket, if already exists
2. Create a Unix socket using `socket()`
3. Specify the socket name
4. Bind the socket using `bind()`
5. `listen()`
6. `accept()`
7. Read the data recvd on socket using `recvfrom()`
8. Send back the result using `sendto()`
9. `close` the data socket
10. `close` the connection socket
11. Remove the socket
12. `exit`

Before diving into these steps, let's study *how the Socket based communication state machine works*

and

*various socket APIs provided by Linux OS*

# Linux System call Interface



## Computer Layer Architecture

- Linux Provides a set of APIs called System calls which application can invoke to interact with the underlying OS
- Socket APIs are the interface between application and OS
- Using these APIs, application instructs the OS to provide its services
- Familiar Example :
  - `malloc()`, `free()`

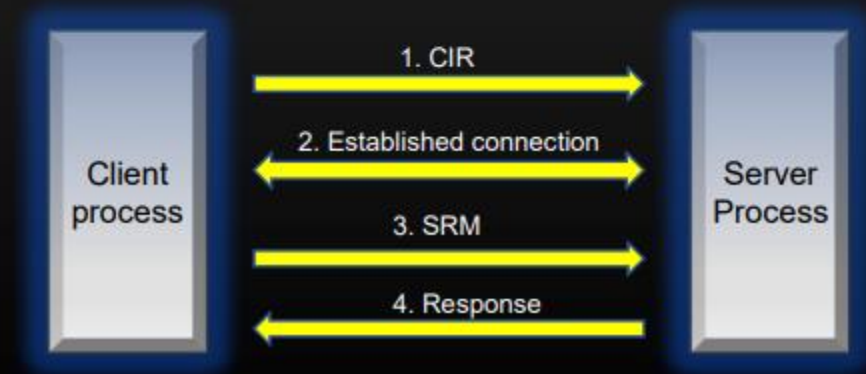


## Socket Message types

- Messages (Or requests) exchanged between the client and the server processes can be categorized into two types :
  - Connection initiation request messages
    - This msg is used by the client process to request the server process to establish a *dedicated* connection.
    - Only after the connection has been established, then only client can send Service request messages to server.

**And**

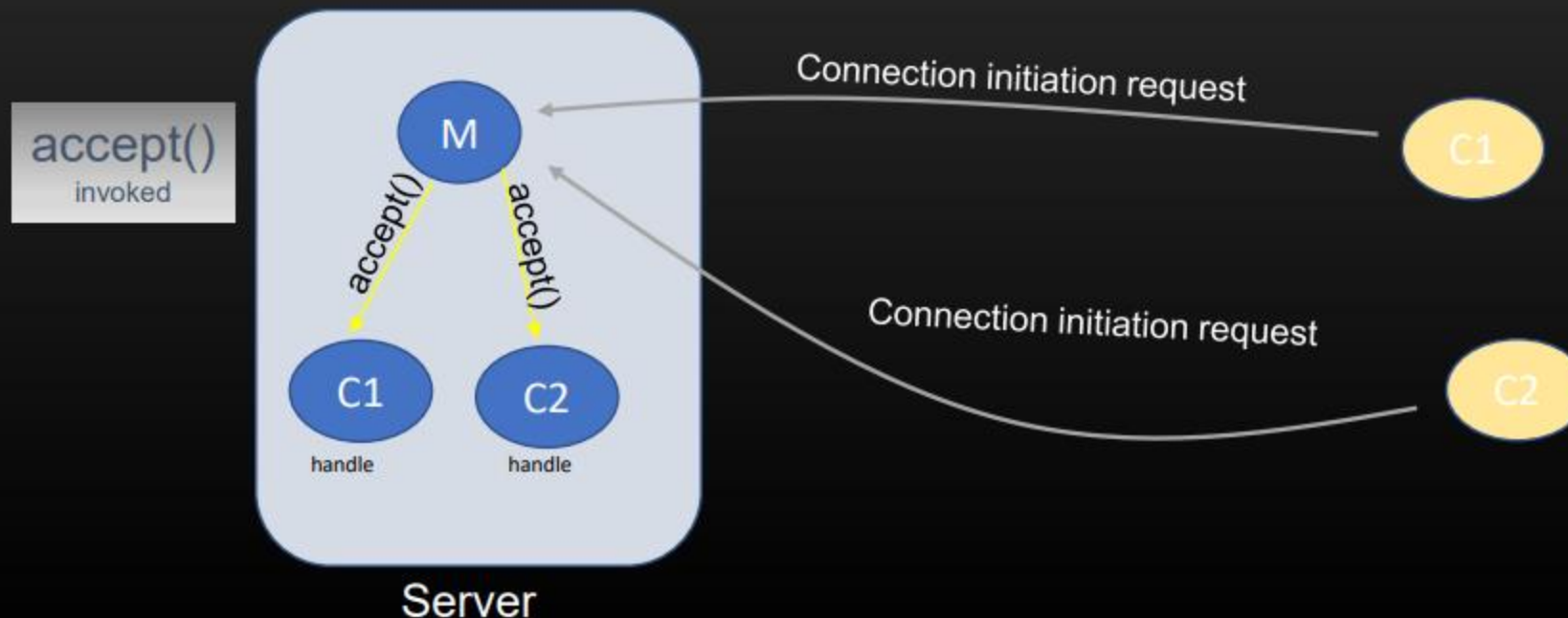
- Service Request Messages
  - Client can send these msg to server once the connection is fully established.
  - Through these messages, Client requests server to provide a service
- Servers identifies and process both the type of messages very differently





### *accept()*

- When the server receives new connection initiation request msg from new client, this request goes on Master socket maintained by server. Master socket is activated.
- When Server receives connection initiation request from client, Server invokes the `accept()` to establish the bidirectional communication
- Return value of `accept` System call is Client handle Or communication file descriptor
- `accept()` is used only for connection oriented communication, not for connection less communication



- Communication between two processes can be broadly classified as

- *STREAM Based communication*
- *DATAGRAM based communication*



*How data is transmitted*

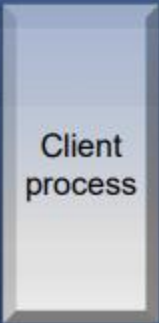
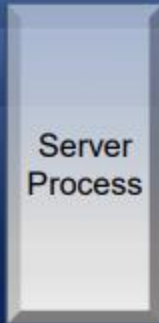
- Communication between two processes can also be broadly classified as

- *Connection Oriented communication*
- *Connection-less Communication*



*Nature of connection*

Communication types : Connection oriented Vs Connection less

|                              | Connection oriented<br>Communication   | Connection-less<br>Communication  |
|------------------------------|--|---|
| Prior Connection Requirement | Required   | Not Required  |
| Reliability                  | Ensures reliable transfer of data.   | Not guaranteed. If data is lost in the way, it is lost forever                        |
| Congestion                   | Can control congestion   | Cant control congestion   |
| Lost data retransmission     | Feasible, eg TCP communication   | Not feasible  |
| Suitability                  | Suitable for long and steady communication.<br>Eg : Audio, Video calls, Live broadcast, File downloads etc | Suitable for bursty Transmission.<br>Eg : Sending chat msgs, Emails etc               |
| Data forwarding              | Bytes are received in the same sequence, through same route  | Bytes are received in the random sequence, through different routes                   |
| Communication type           | STREAM based Communication   | Datagram based Communication  |
|                              |                         |  |

# *UNIX DOMAIN SOCKETS*

- Unix Domain Sockets are used for carrying out IPC between two processes running on **SAME** machine
- We shall discuss the implementation of Unix Domain Sockets wrt to Server and Client processes
- Using UNIX Domain sockets, we can setup STREAM Or DATAGRAM based communication
  - **STREAM** - When large files need to be moved or copied from one location to another, eg : copying a movie  
ex : continuous flow of bytes, like water flow
  - **DATAGRAM** - When small units of Data needs to be moved from one process to another within a system

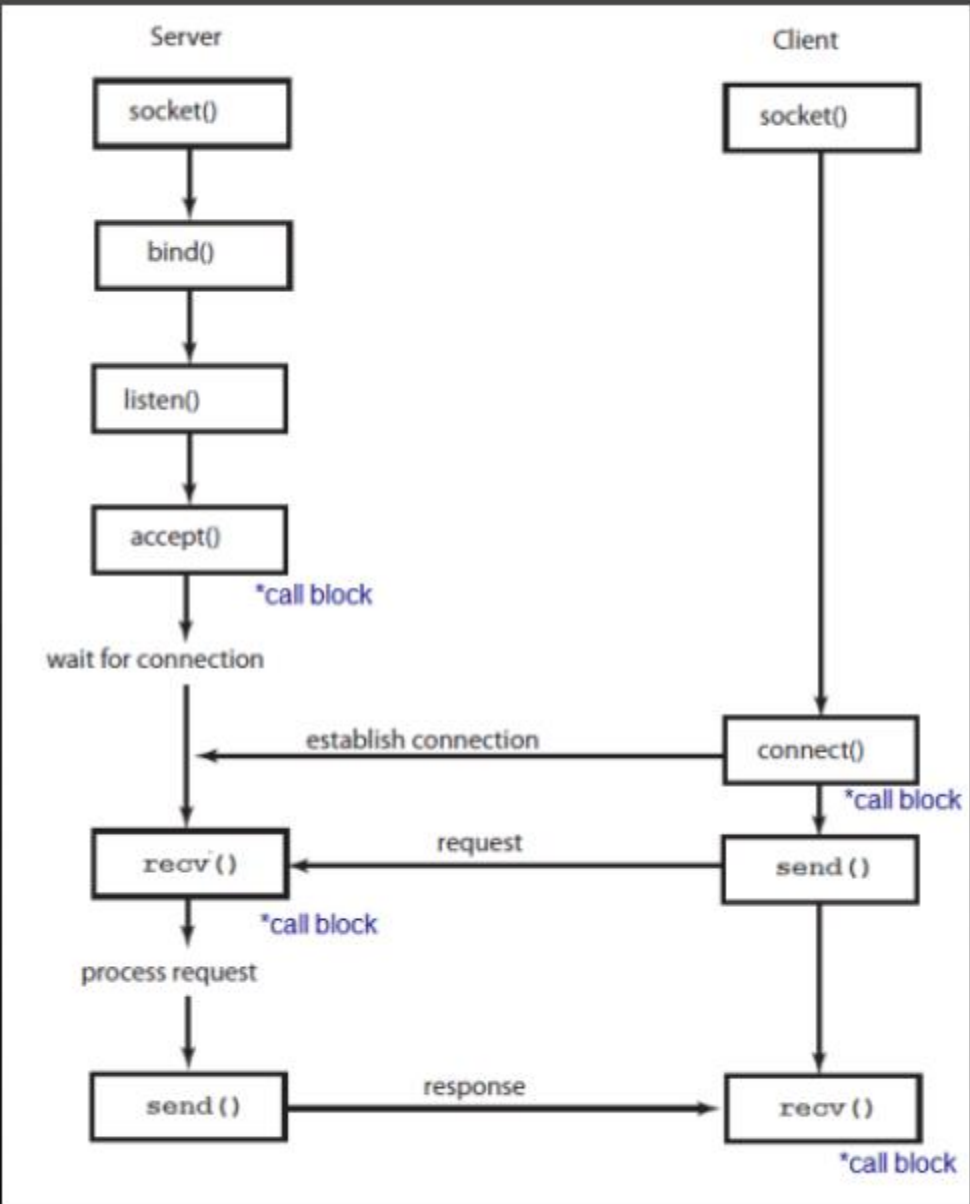


# Unix Domain sockets -> code Walk

- Code Walk for Process A (Server)

- Steps :

1. Remove the socket, if already exists
2. Create a Unix socket using `socket()`
3. Specify the socket name
4. Bind the socket using `bind()`
5. `listen()`
6. `accept()`
7. Read the data recvd on socket using `read()`
8. Send back the result using `write()`
9. `close` the data socket
10. `close` the connection socket
11. Remove the socket
12. exit



Generic steps for socket programming



- High level Socket Communication Design

| Msg type                           | Client side API                    | Server side API                                  |
|------------------------------------|------------------------------------|--|
|                                    |                                    |  |
| Connection initiation request msgs | connect()                          | accept()<br>(blocking)                           |
| Service request msgs               | sendmsg(),<br>sendto(),<br>write() | recvmsg(),<br>recvfrom()<br>read()<br>(blocking) |

## Unix Domain sockets -> Observation

- While Server is servicing the current client, it cannot entertain new client
- This is a drawback of this server design, and we need to alleviate this limitation
- A server can be re-designed to server multiple clients at the same time using the concept of *Multiplexing*

## Multiplexing

- Multiplexing is a mechanism through which the Server process can monitor multiple clients at the same time
- Without Multiplexing, server process can entertain only one client at a time, and cannot entertain other client's requests until it finishes with the current client
- With Multiplexing, Server can entertain multiple connected clients simultaneously

### No Multiplexing



Once the current client is serviced by the server, Client has to join the queue right from the last (has to send a fresh connection request) to get another service from the server

### Multiplexing



Server can service multiple clients at the same time

*select()*

**Monitor all Clients activity at the same time**



## Multiplexing -> select

- Server-process has to maintain client handles (communication FDs) to carry out communication (data exchange) with connected clients
- In Addition, Server-process has to maintain connection socket Or Master socket FD as well (M) to process new connection req from new clients
- Linux provides an inbuilt Data structure to maintain the set of sockets file descriptors

`fd_set`

- `select()` system call monitor all socket FDs present in `fd_set`

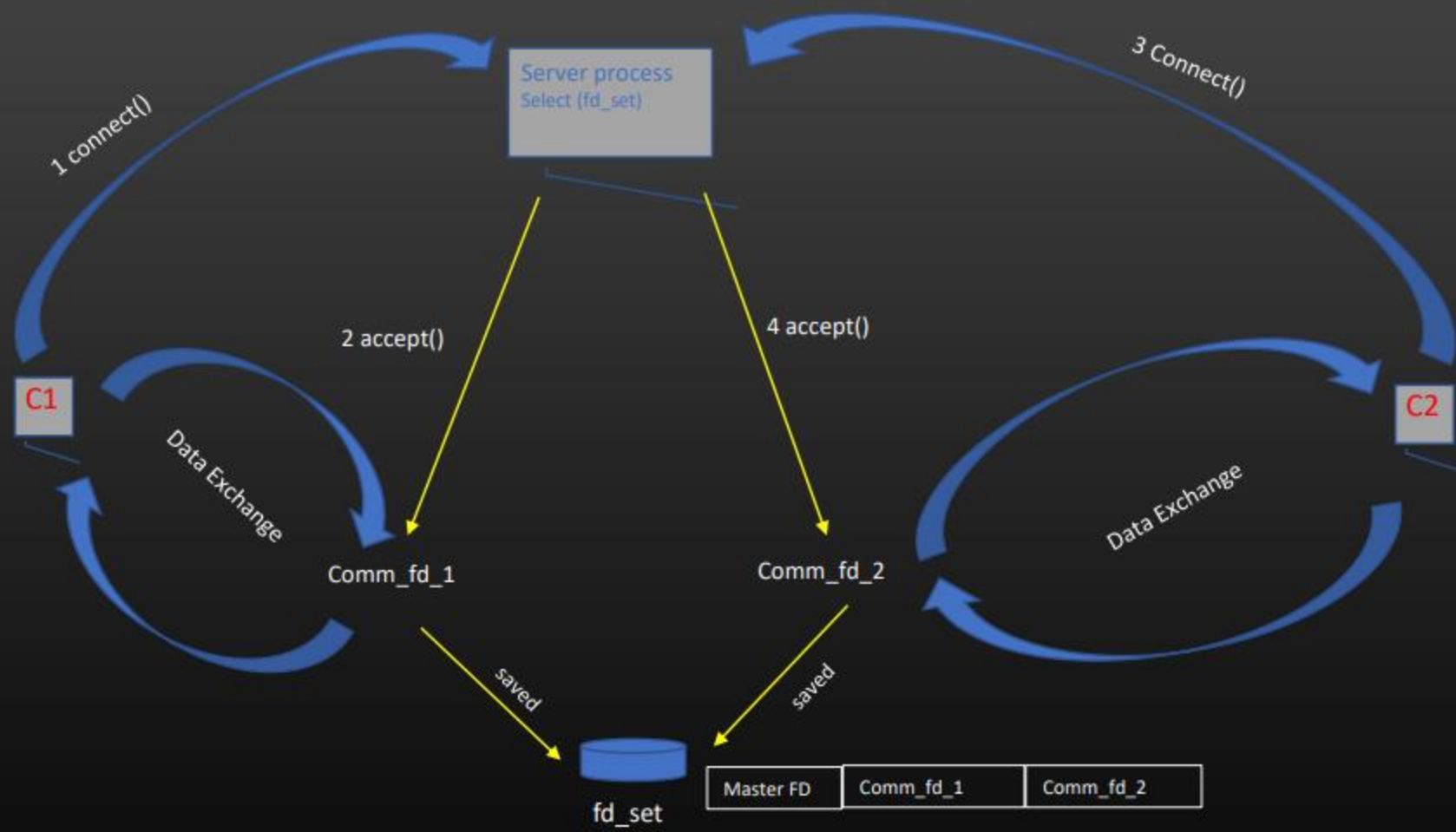


Server

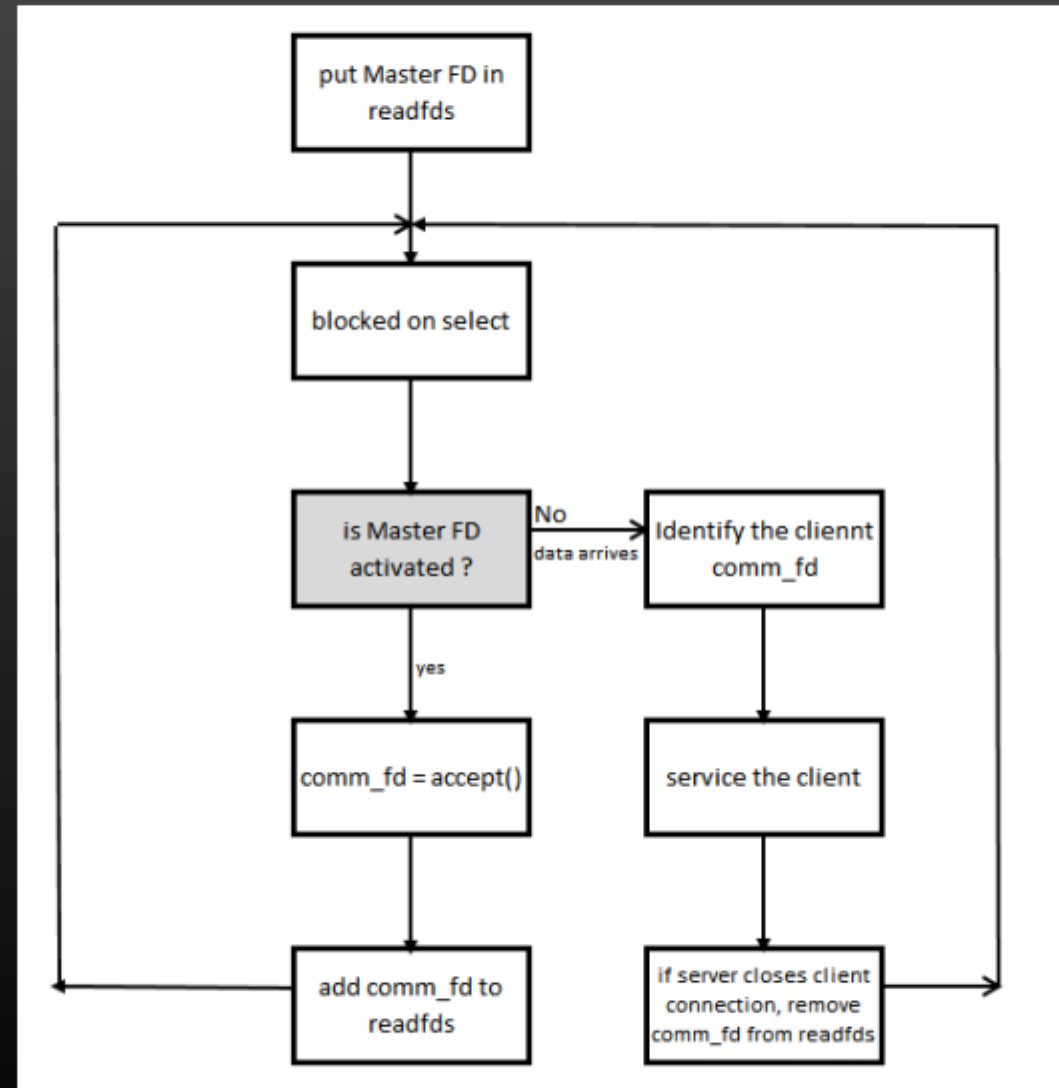
- Let us have a detailed discussion on `select()` system call



# Select and accept together in action



## Multiplexed Server process state machine diagram



*Time for a Code walk ...*

Thank You!!