Part 1

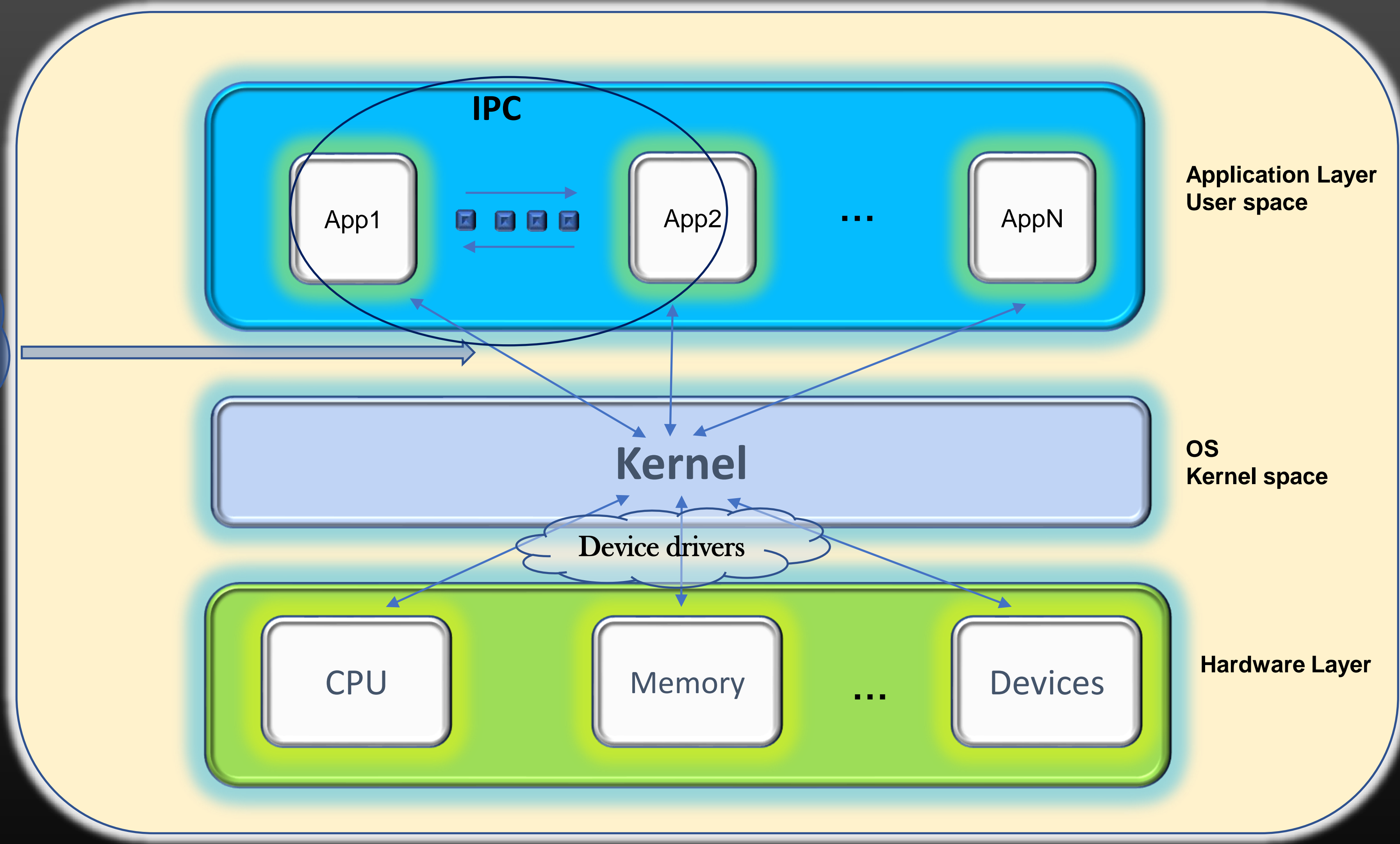
Inter-process Communication in Linux (IPC)

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 :

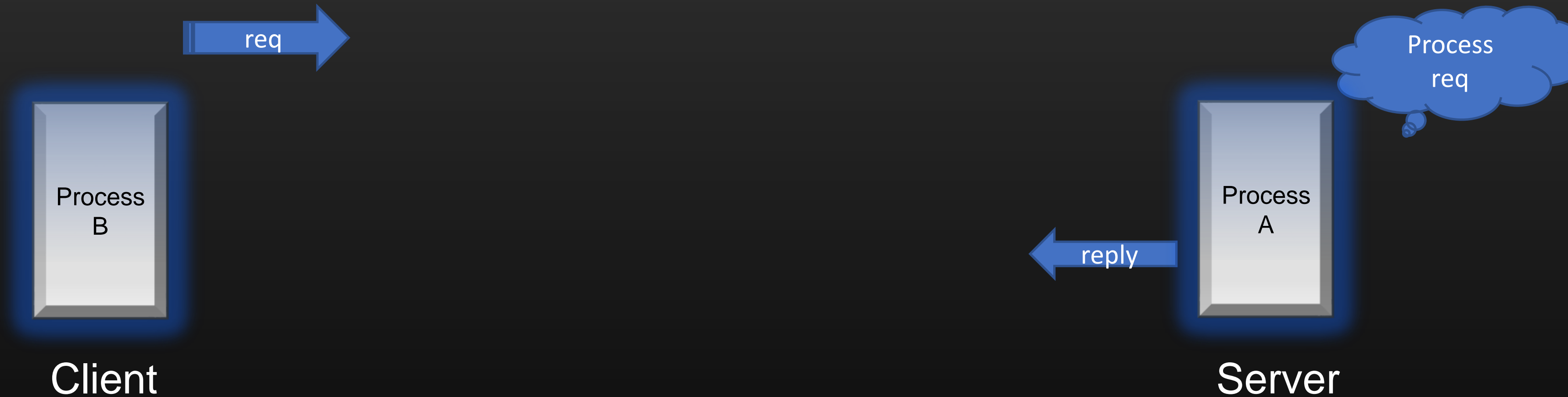
  1. Using Unix Domain sockets
  2. Using Network Sockets (Not covered in this course)
  3. Message Queues
  4. Shared Memory
  5. Pipes
  6. Signals

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

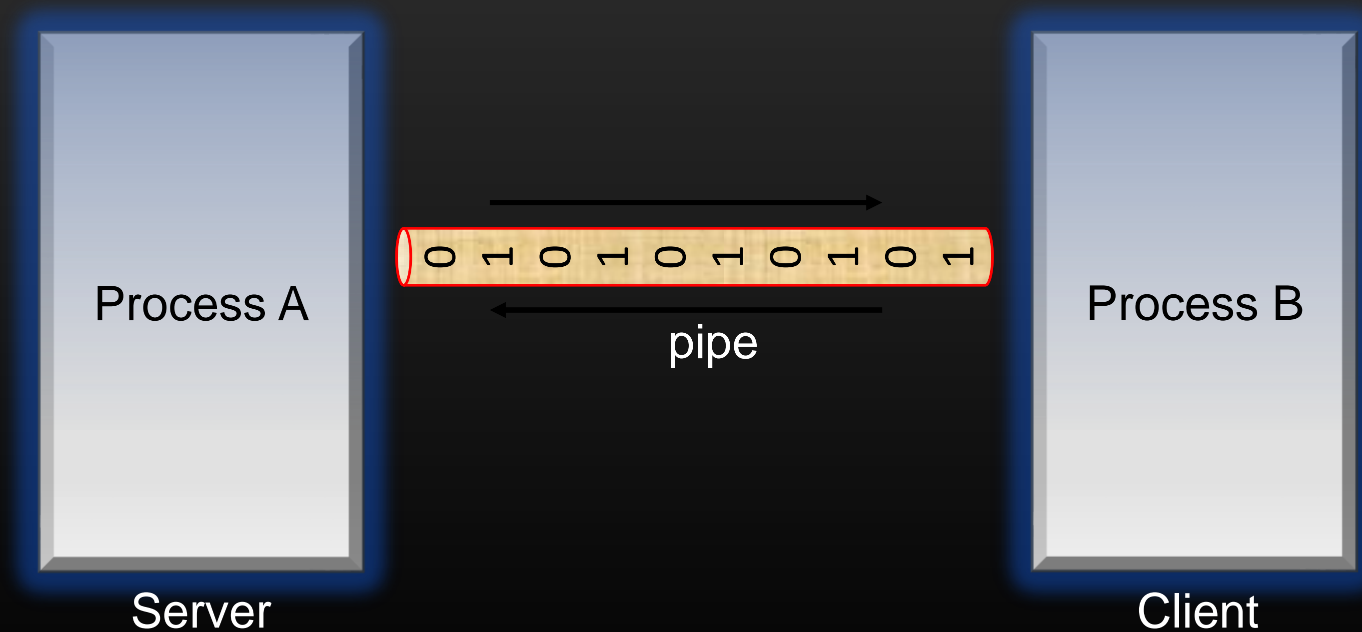


Communication types

- 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



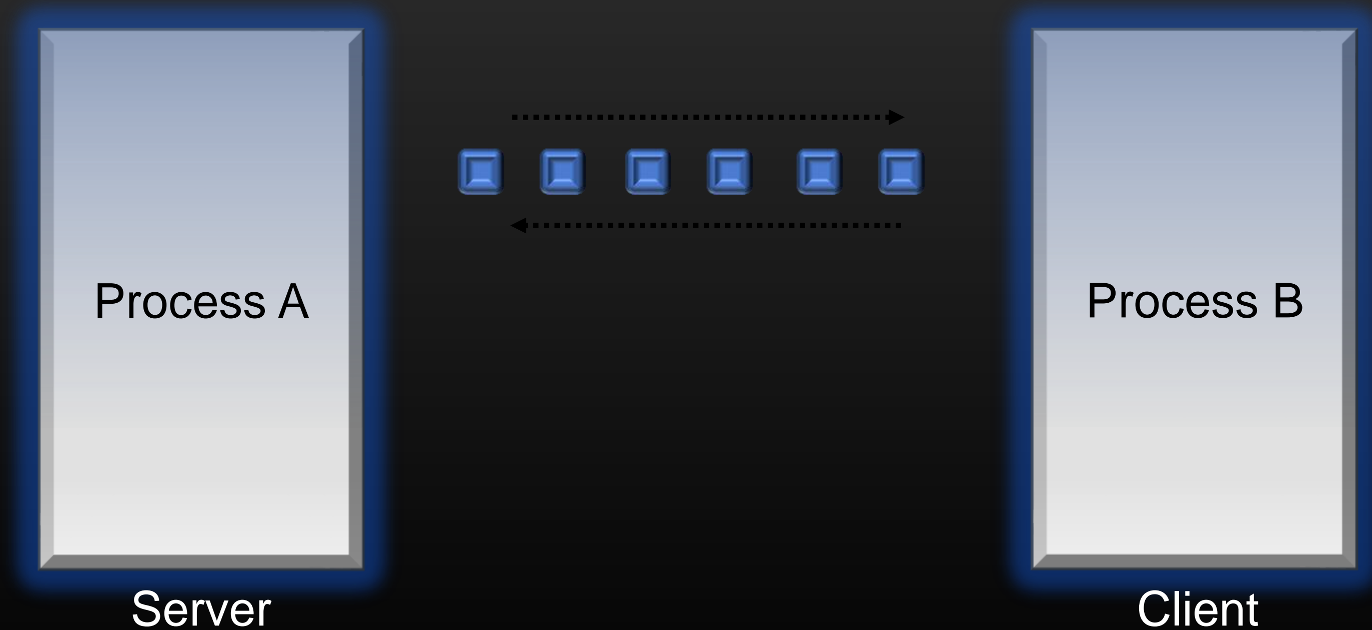
Communication types

- 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 . . .

IPC Techniques – Sockets

- Socket programming Steps and related APIs
- Steps :

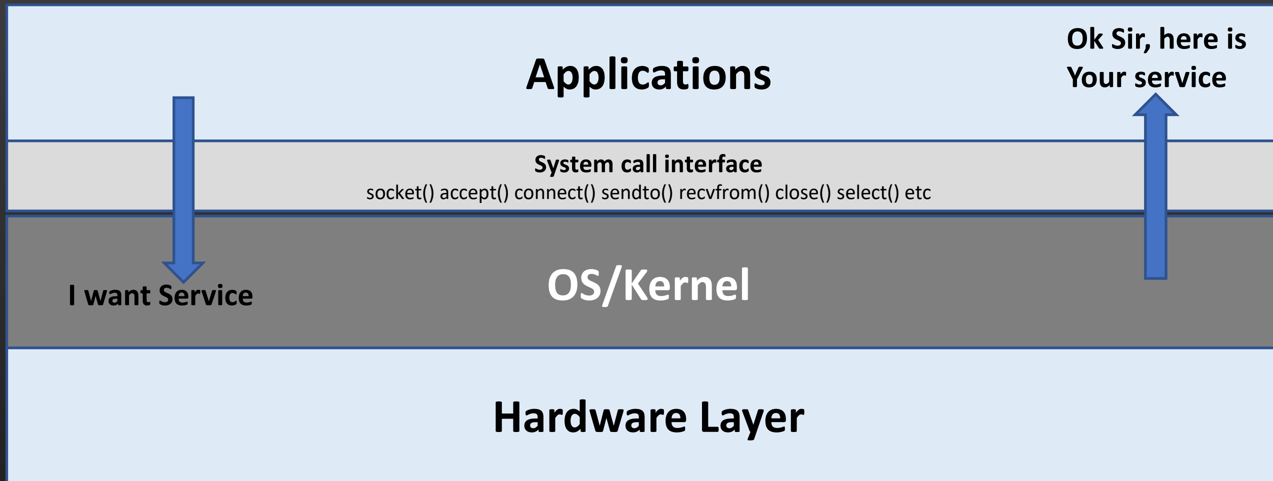
1. Remove the socket, if already exists
2. Create a Unix socket using *socket()*
3. Specify the socket name
4. Bind the socket using *bind()*
5. listen()
6. accept()
7. Read the data recvd on socket using recvfrom()
8. Send back the result using *sendto()*
9. *close* the data socket
10. *close* the connection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

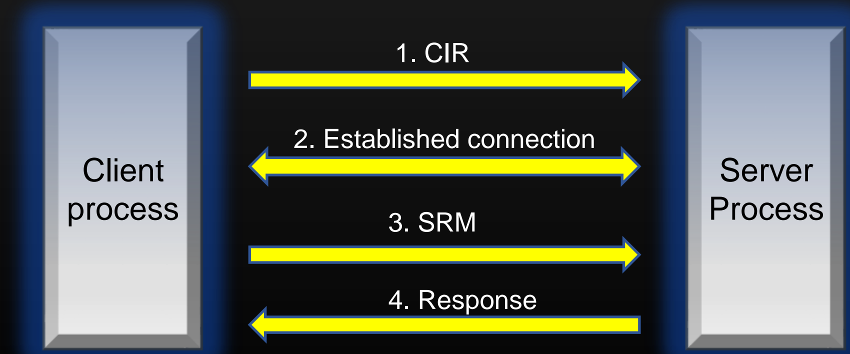
Socket Message types

- 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

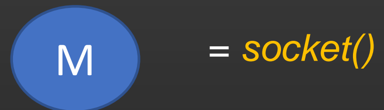
  - Service Request Messages
    - Client can send these msg to server once the connection is fully established.
    - Through these messages, Client requests server to provide a service

- Servers identifies and process both the type of messages very differently



State machine of Socket based Client Server Communication

1. When Server boots up, it creates a *connection socket* (also called *"master socket file descriptor"* using *socket()*)



= *socket()*

2. M is the mother of all Client Handles. M gives birth to all Client handles. Client handles are also called *"data_sockets"*

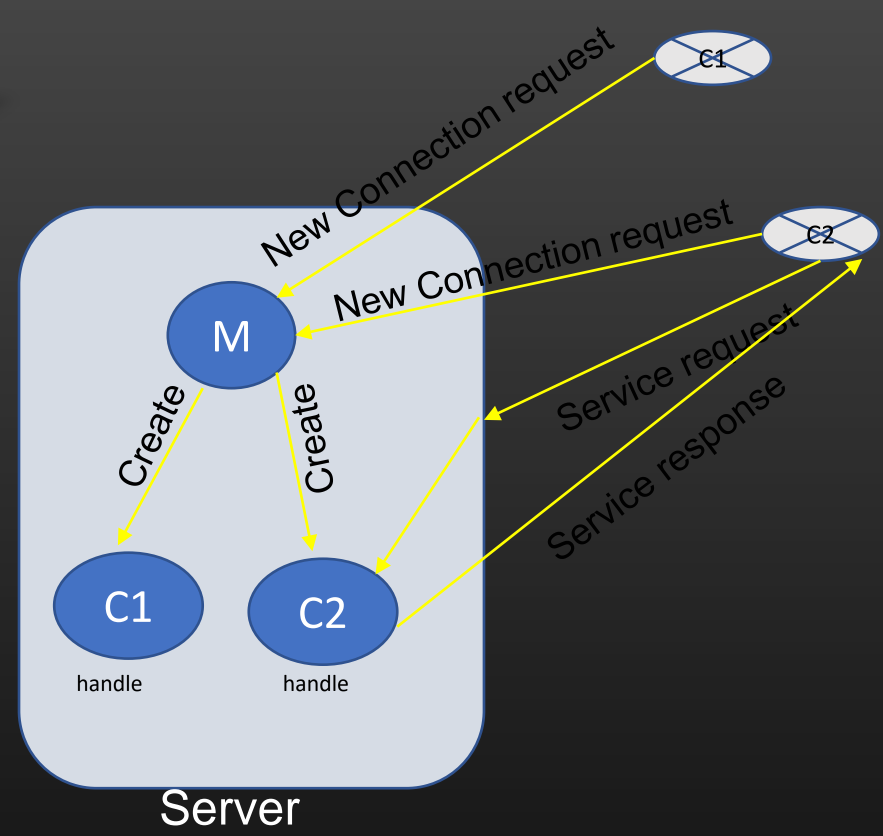
3. Once Client handles are created for each client, Server carries out Communication *(actual* **data** *exchange)* with the client using Client handle (and not M).

4. Server Has to maintain the database of connected client handles Or data sockets

5. M is only used to create new client handles. M is not used for data exchange with already connected clients.

6. *accept()* is the system call used on server side to create client handles.

7. In Linux Terminology, handles are called as *"file descriptors"* which are just positive integer numbers. Client handles are called *"communication file descriptors"* Or *"Data Sockets"* and M is called *"Master socket file descriptor"* Or *"Connection socket"*

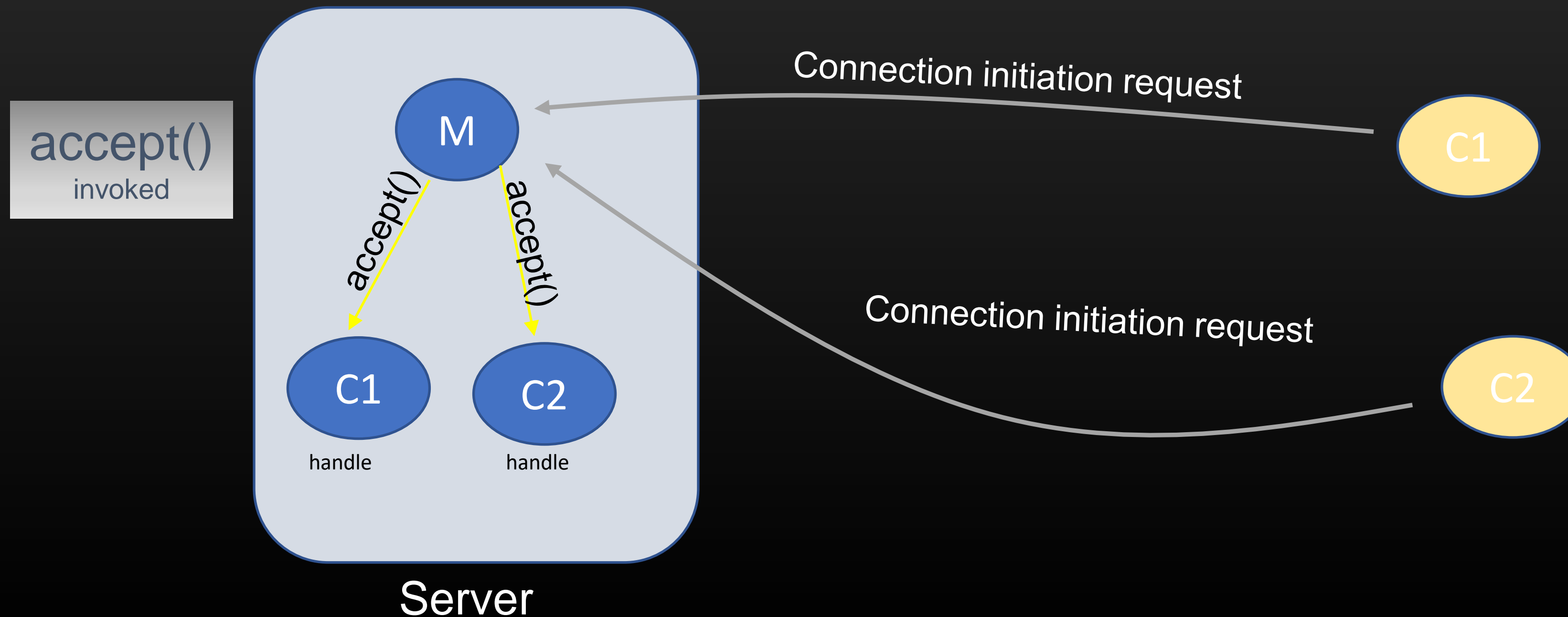


accept()

- 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 types

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

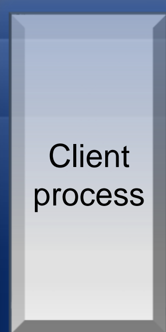
How data is transmitted

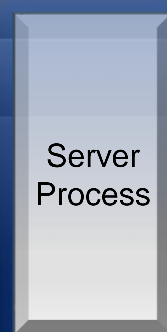
- Communication between two processes can also be broadly classified as
  - *Connection Oriented communication*
  - *Connection-less Communication*

Nature of connection

Communication types : Connection oriented Vs Connection less

	Connection oriented Communication	Connection-less 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. Eg : Audio, Video calls, Live broadcast, File downloads etc	Suitable for bursty Transmission. 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





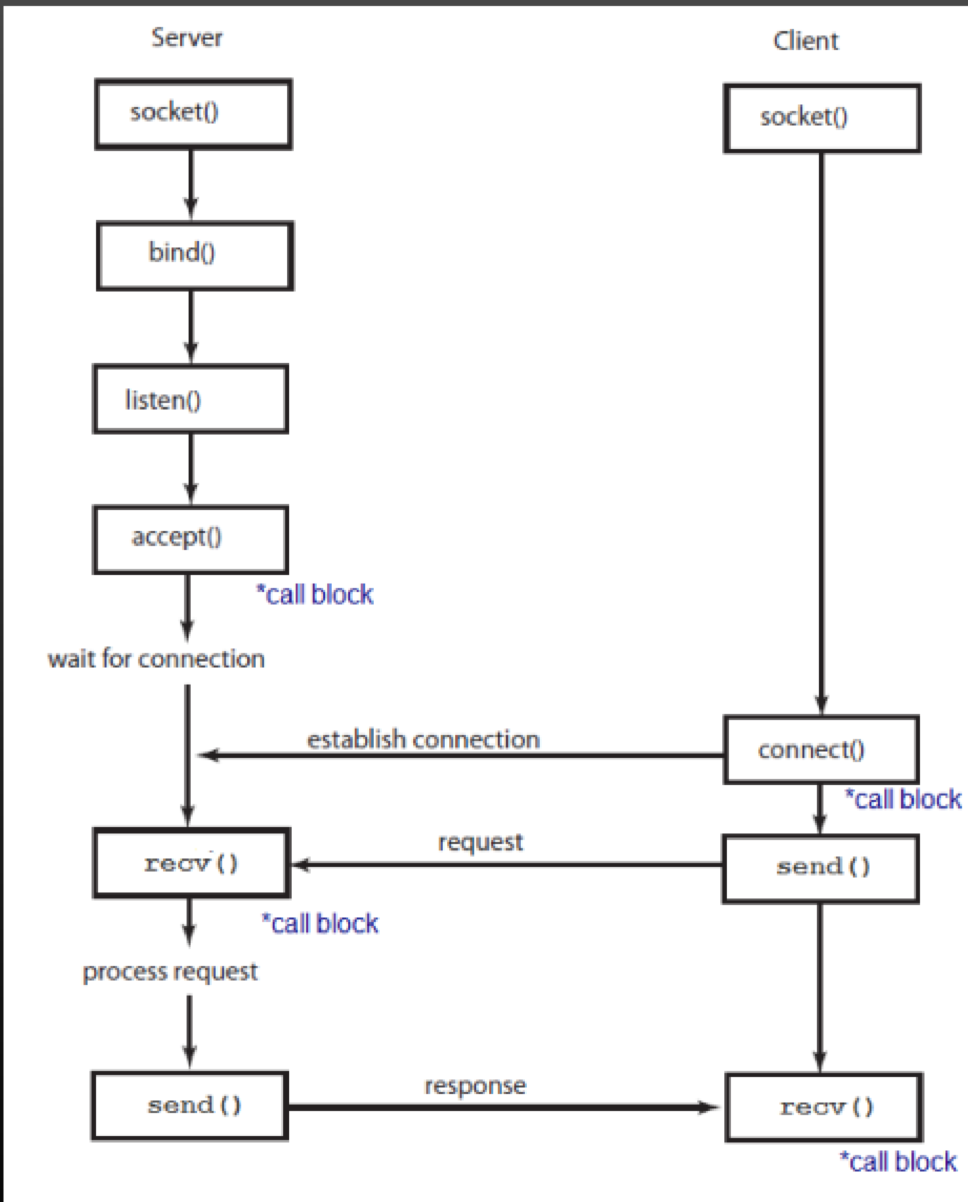
UNIX DOMAIN SOCKETS

- 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

Unix Domain sockets -> code Walk

- Code Walk for Process B (Client)
- Steps :
    1. …

Unix Domain sockets -> Demonstration

- Demonstration . . .

Socket Communication APIs

- High level Socket Communication Design

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

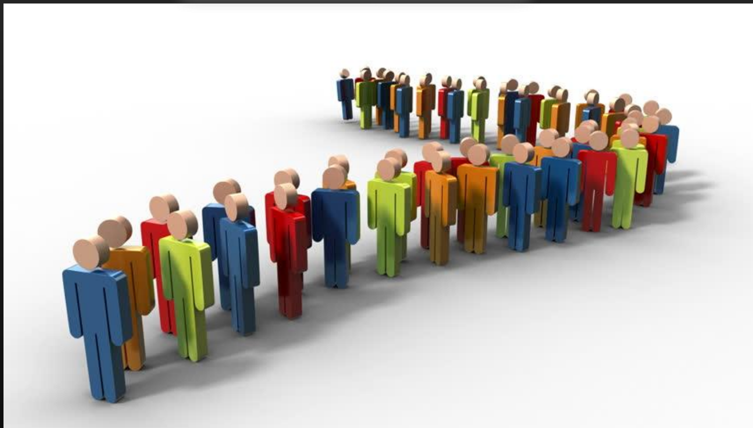
Unix Domain sockets -> Observation

- 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()

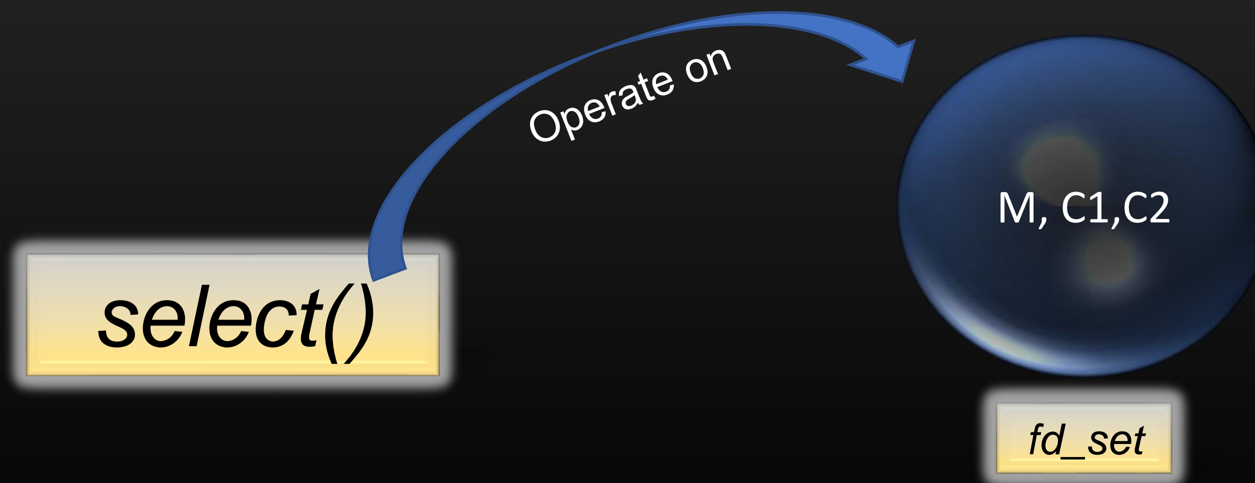
Monitor all Clients activity at the same time

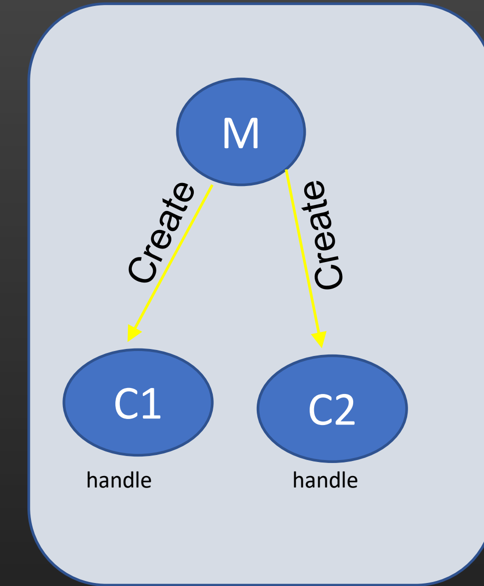
Multiplexing -> select

- 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*

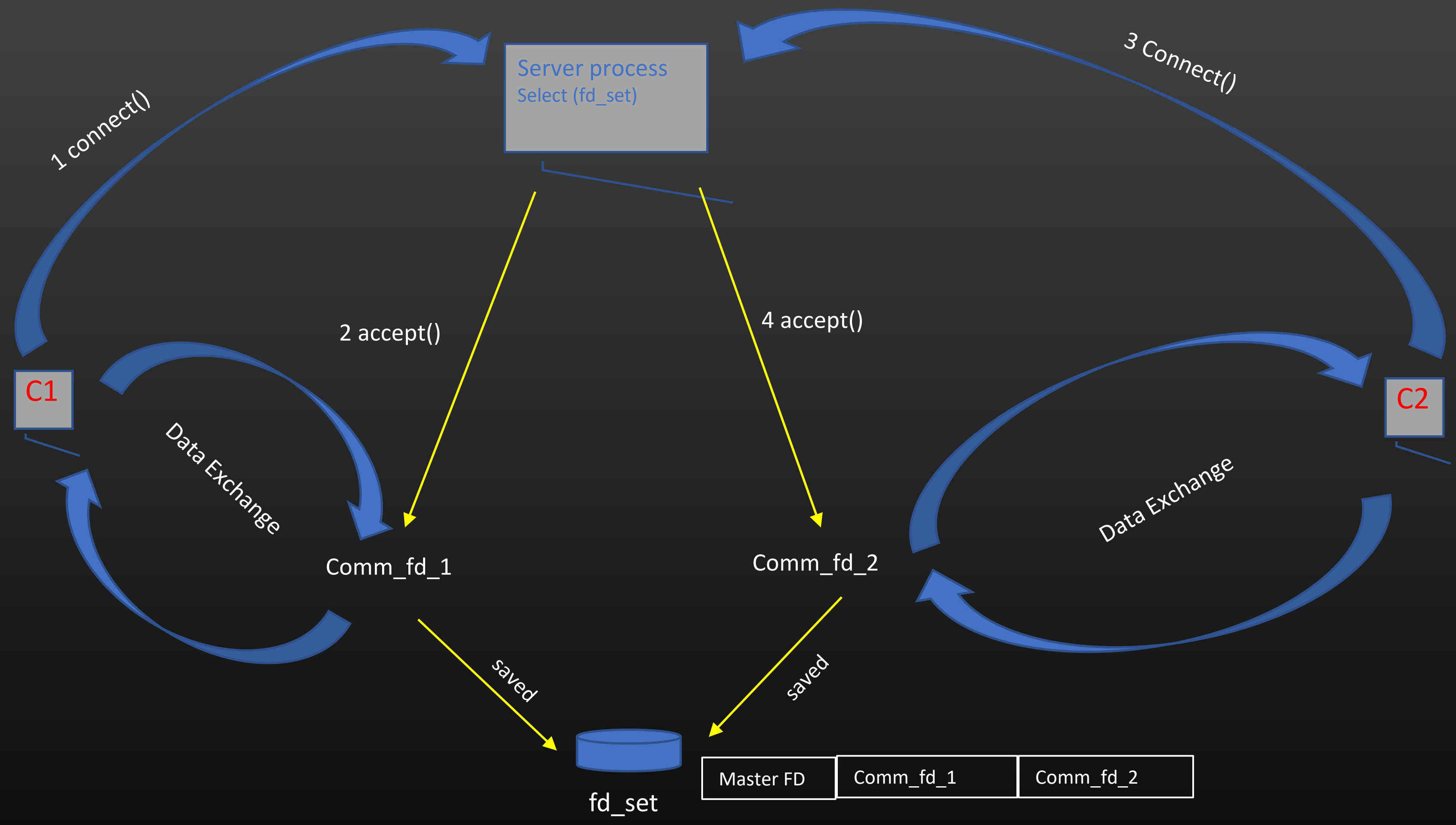




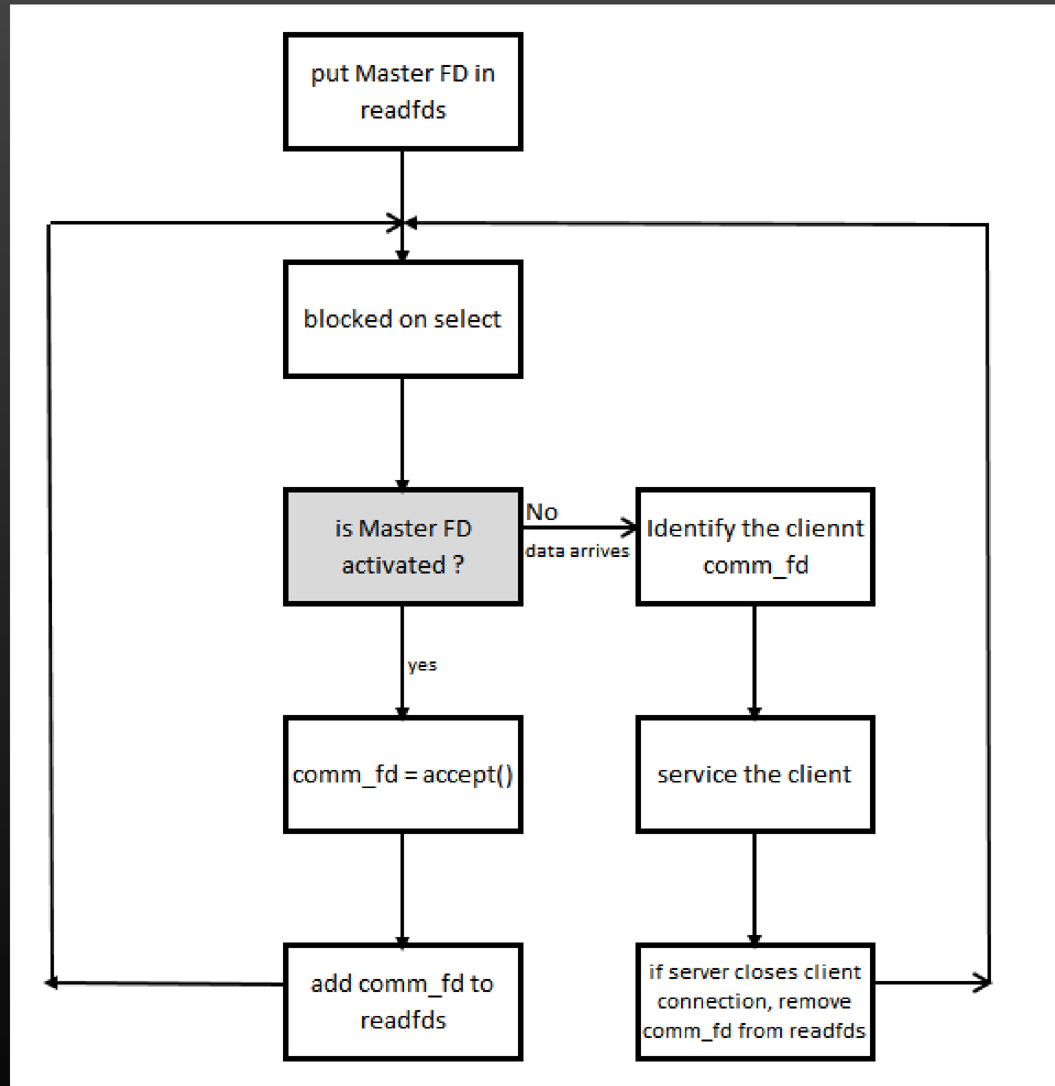
Server

- 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 …

Unix domain socket Project -> Data Synchronization

Let us Create a routing table Manager (RTM) process

- A RTM is in-charge of a Layer 3 routing table
- Its responsibility is to maintain the L3 routing table, and sends notification of any change in the routing table contents to connected clients
- State of routing table needs to be synchronized across all clients at any point of time

Unix domain socket Project -> Data Synchronization

A big picture

Client process

CUD

Route Manager process

CUD

Client process

CUD

Client process

Route Manager Server process sends CUD (Create, Update And Delete) notification to all connected Client process)

Let us discuss the project step by step

Unix domain socket Project -> Data Synchronization

- Route Manager process maintains a routing table
- This table is managed by Admin
- Sample of table is shown below.
- You are free to choose the Data structure to represent and manage this table

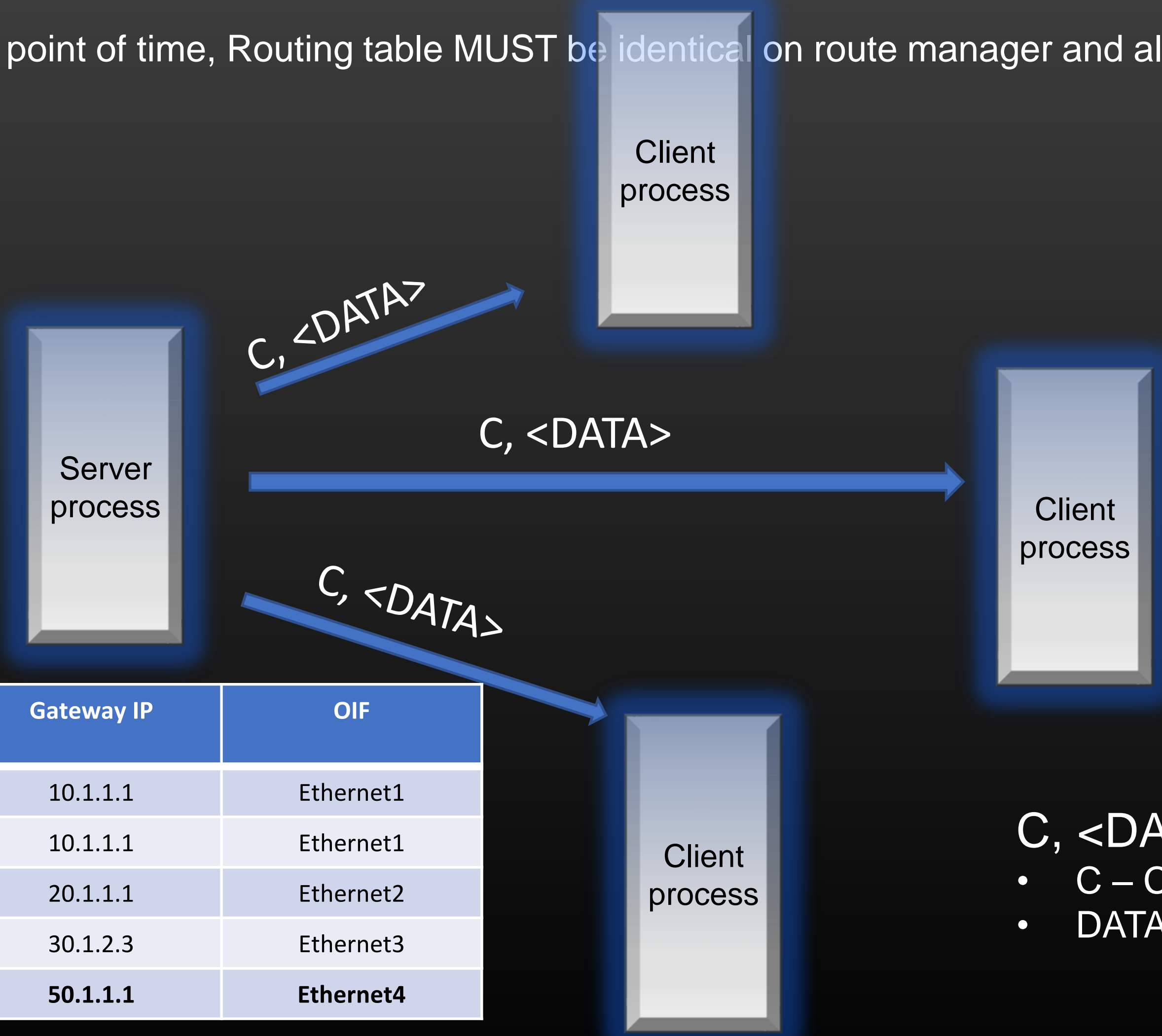
Note :
Mask – [0,32]

Destination Subnet (Key)	Gateway IP	OIF
122.1.1.1/32	10.1.1.1	Ethernet1
130.1.1.0/24	10.1.1.1	Ethernet1
126.30.34.0/24	20.1.1.1	Ethernet2
220.1.0.0/16	30.1.2.3	Ethernet3

- Operation supported on the table :
  - Insert - <Destination/mask> <Gateway IP> <OIF>
  - Update - <Destination/mask> <new Gateway IP> <new OIF>
  - Delete - <Destination/mask>

Unix domain socket Project -> Data Synchronization

- Whenever the User perform any CUD operation on the routing table, Route Manager Server process sync that particular operation to all connected clients
- When new client connects to the server, Server sends the entire table state to this newly connected client
- At any given point of time, Routing table MUST be identical on route manager and all connected clients



Destination Subnet (Key)	Gateway IP	OIF
122.1.1.1/32	10.1.1.1	Ethernet1
130.1.1.0/24	10.1.1.1	Ethernet1
126.30.34.0/24	20.1.1.1	Ethernet2
220.1.0.0/16	30.1.2.3	Ethernet3
100.100.100.0/24	**50.1.1.1**	**Ethernet4**

Destination Subnet (Key)	Gateway IP	OIF
122.1.1.1/32	10.1.1.1	Ethernet1
130.1.1.0/24	10.1.1.1	Ethernet1
126.30.34.0/24	20.1.1.1	Ethernet2
220.1.0.0/16	30.1.2.3	Ethernet3
100.100.100.0/24	**50.1.1.1**	**Ethernet4**

C, <DATA>

- C – CREATE (operation code)
- DATA - <Dest/mask> <Gw IP> <OIF>

Provide a menu-driven approach to show routing table Contents on Server and Client processes

Unix domain socket Project -> Data Synchronization

Data Structures Suggestions :

The operation code could be Enums :

```
typedef enum{

        CREATE,
        UPDATE,
        DELETE
} OPCPDE;
```

The structure which contains the data to be synchronized can be :

```
typedef struct _msg_body {

        char destination[16];
        char mask;
        char gateway_ip[16];
        char oif[32];
} msg_body_t;
```

The final msg that needs to be synced from routing table manager process to all clients
Should have Operation code and Body

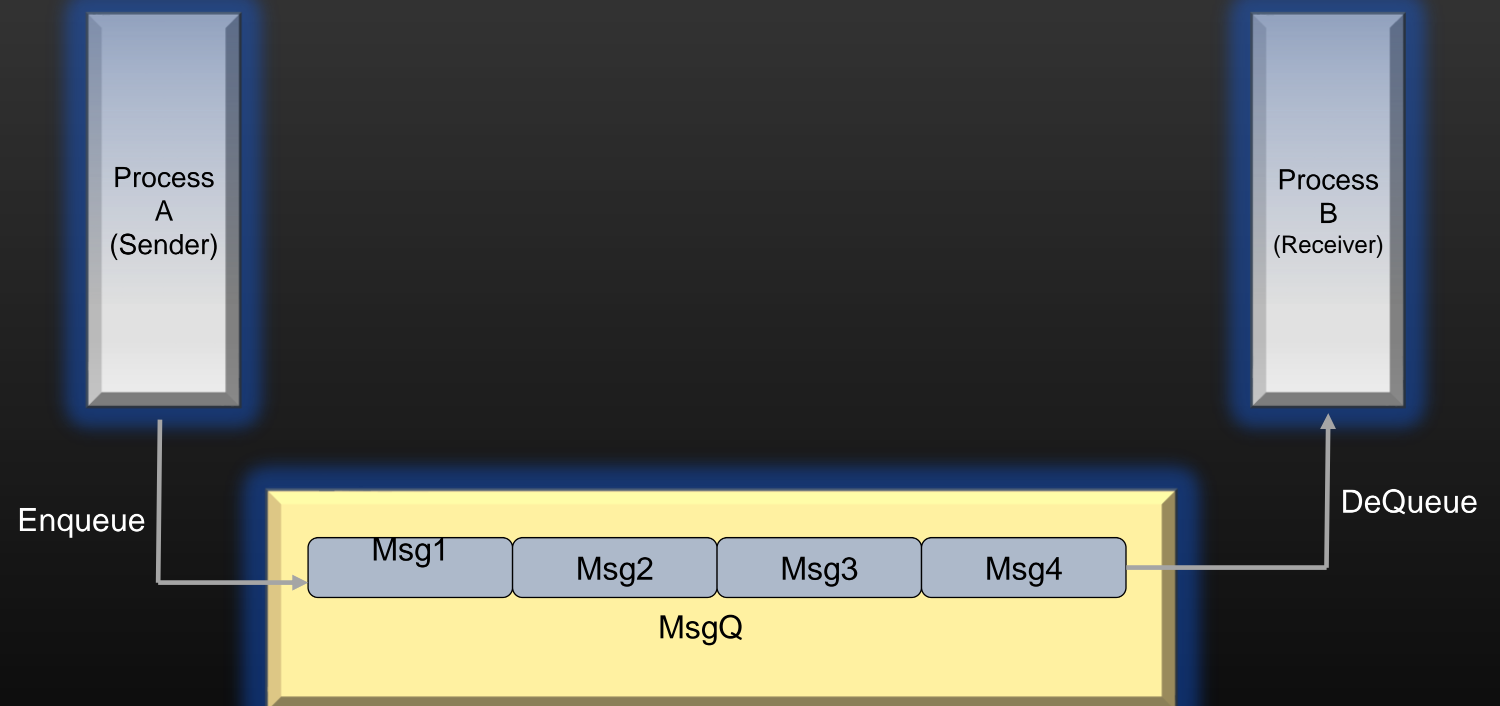
```
typedef  struct _sync_msg {

        OPCODE op_code;
        msg_body_t msg_body;
} sync_msg_t;
```

Socket programming APIs

1. socket() – Used to create a connection/master socket on server side. Used to create data socket on Client side
2. select() – Used for monitoring multiple file descriptors. Can be used on both client and server sides. Blocking system call. Blocks until new connection request Or data arrived on FDs present in fd_set.
3. accept() – Used on server side. Used to accept the client new connection request and establish connection with client
4. bind() – Used on server side. Used by the Server application process to inform operating system the criteria of packets of interests
5. listen() – Used on server side. Used by the Server application process to inform operating system the length of Queue of incoming connection request/data request from clients
6. read() – Used on the server and client side. Used by the Server/client process to read the data arrived on communication file descriptors. This call is blocking call by default. Process block if data hasn’t arrived yet.
7. write() – Used to send data to client/server. Used on Server and client sides.
8. close() – Used to close the connection. Used by both – clients and Server

Message Queues

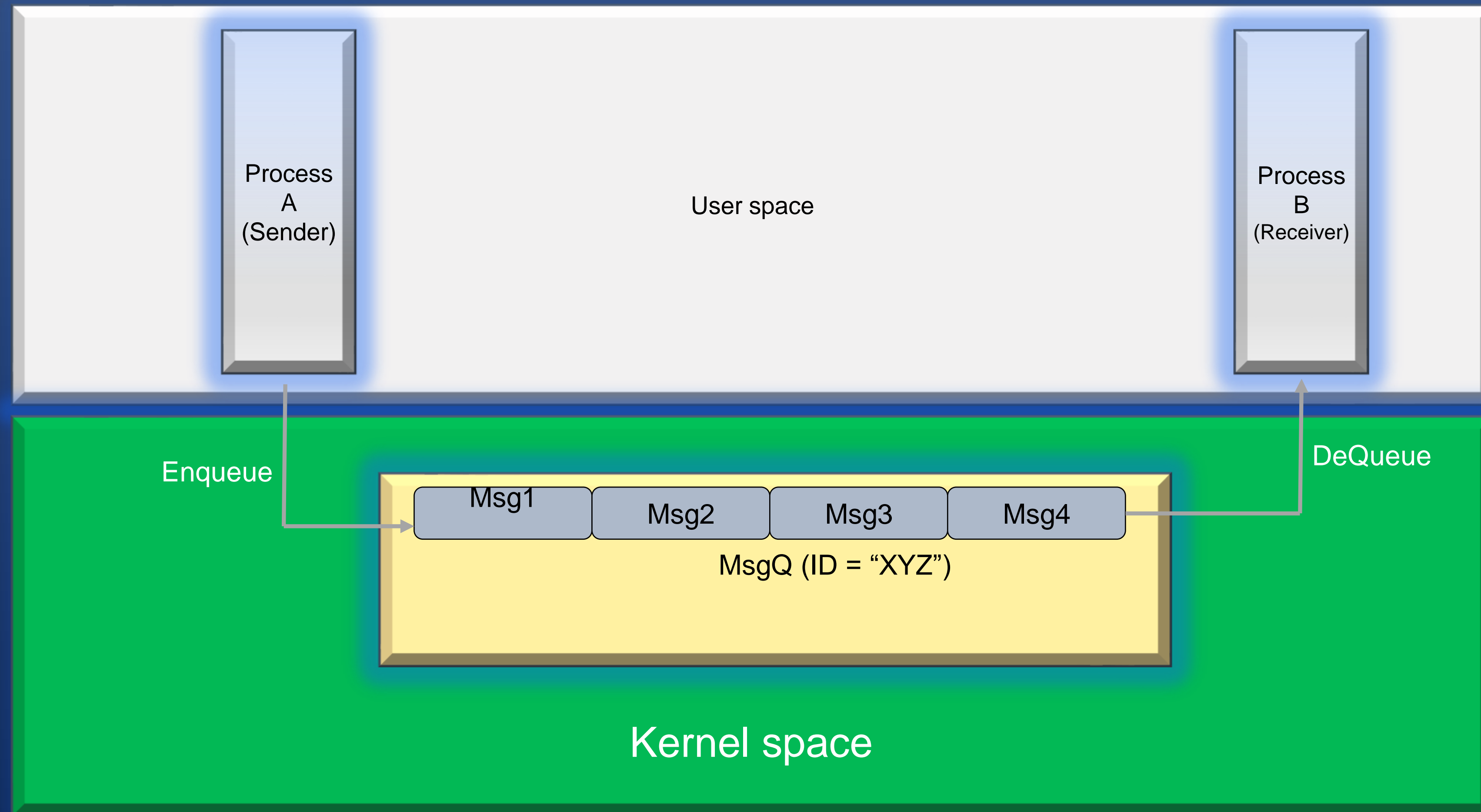
1. IPC using Message Queues
2. Msgq Concepts
3. MsgQ management APIs
4. Using a MsgQ
5. Code Walk – Step by Step



IPC – Techniques -> Message Queue

- Linux/Unix OS provides another mechanism called *Message Queue* for carrying out IPC
- Process(s) running on same machine can exchange any data using Message queues
- Process can create a new message Queue Or can use an existing msgQ which was created by another process
- A Message Queue is identified uniquely by the ID, No two msgQ can have same ID
- Message Queue resides and manage by the Kernel/OS
- Sending process A can post the data to the message Queue, Receiving process B reads the data from msg Q
- Process which creates a msgQ is termed as owner Or creator of msgQ

IPC – Techniques -> Message Queue



- There can be multiple message Queues created by several processes in the system
- Each message Queue is identified by a unique name called msgQ ID which is a string

1. Message Queue Creation

A Process can create a new msgQ or use an existing msgQ using below API :

```
mqd_t
mq_open (const char *name,
                int oflag);

mqd_t
mq_open (const char *name,
                int oflag, mode_t mode,
                struct mq_attr *attr);
```

- Two flavors of the API
- *name* – Name of msg Q , eg "/server-msg-q"
- *oflag* – Operational flags
  - O_RDONLY : process can only read msgs from msgQ but cannot write into it
  - O_WRONLY : process can only write msgs into the msgQ but cannot read from it
  - O_RDWR : process can write and read msgs to and from msgQ
  - O_CREAT : The queue is created if not exist already
  - O_EXCL : mq_open() fails if process tries to open an existing queue. This flag has no meaning when used alone. Always used by OR-ing with O_CREAT flag
  - *mode* - Permissions set by the owning process on the queue, usually 0660
  - *attr* – Specify various attributes of the msgQ being created
    - Like maximum size the msgQ can grow, should be less than or equal to *\/proc/sys/fs/mqueue/msg_max*
    - Maximum size of the msg which msgQ can hold , should be less than or equal to *\/proc/sys/fs/mqueue/msgsize_max*

```
struct mq_attr {
        long mq_flags;          /* Flags: 0 */
        long mq_maxmsg;     /* Max. # of messages on queue */
        long mq_msgsize;     /* Max. message size (bytes) */
        long mq_curmsgs;     /* # of messages currently in queue */
};
```

If mq_open() succeeds, it returns a file descriptor (a handle) to msgQ. Using this handle, we perform All msgQ operations on msgQ throughout the program

1. Message Queue Creation

A Process can create a new msgQ or use an existing msgQ using below API :

Example 1: Without attributes

```
mqd_t msgq;

if ((msgq = mq_open (“/server-msg-q”, O_RDONLY | O_CREAT | O_EXCL, 0660, 0)) == -1) {
        perror ("Server: mq_open (server)");
        exit (1);
}
```

Example 2: With attributes

```
mqd_t msgq;

struct mq_attr attr;

attr.mq_flags = 0;
attr.mq_maxmsg = 10;
attr.mq_msgsize = 4096;
attr.mq_curmsgs = 0;

if ((msgq = mq_open (“/server-msg-q”, O_RDONLY | O_CREAT | O_EXCL, 0660, &attr)) == -1) {
        perror ("Server: mq_open (server)");
        exit (1);
}
```

2. Message Queue Closing

A Process can close a msgQ using below API :

```
int
mq_close (mqd_t msgQ);
```

Return value :
0 – sucess
-1 - failure

- After closing the msgQ, the process cannot use the msgQ unless it open it again using mq_open()
- Operating system removes and destroy the msgQ if all processes using the msgQ have closed it
- OS maintains information regarding how many process are using same msgQ (invoked mq_open()). This concept is called *reference counting*
- msgQ is a kernel resource which is being used by application process. For kernel resources, kernel keeps track how many user space processes are using that particular resource
- When a kernel resource (msgQ in our example) is created for the first time by appln process, *reference_count = 1*
- if other process also invoke open() on existing kernel resource (mq_open() in our case) , kernel increments *reference_count by 1*
- When a process invoke close() in existing kernel resource (mq_close() in our case), kernel decrements *reference_count by 1*
- When *reference_count = 0*, kernel cleanups/destroys that kernel resource

Remember, Kernel resource could be anything, it could be socket FD, msgQ FD etc

3. Enqueue a Message

A Sending Process can place a message in a message Queue using below API :

```
int
mq_send (mqd_t msgQ,
                    char *msg_ptr,
                    size_t msg_len,
                    unsigned int msg_prio);
```

Return value :
0 – success
-1 - failure

- ***mq_send*** is for sending a message to the queue referred by the descriptor msgQ.
- The ***msg_ptr*** points to the message buffer. ***msg_len*** is the size of the message, which should be less than or equal to the message size for the queue.
- ***msg_prio*** is the message priority, which is a non-negative number specifying the priority of the message.
  - Messages are placed in the queue in the decreasing order of message priority, with the older messages for a priority coming before newer messages.
- If the queue is full, ***mq_send*** blocks till there is space on the queue, unless the O_NONBLOCK flag is enabled for the message queue, in which case *mq_send* returns immediately with errno set to EAGAIN.

4. Dequeue a Message

A Receiving Process can dequeue a message from a message Queue using below API :

```
int
mq_receive (mqd_t msgQ,
            char *msg_ptr,
            size_t msg_len,
            unsigned int *msg_prio);
```

Return value
n_bytes of recvd msg – success
-1 - failure

- ***mq_receive*** is for retrieving a message from the queue referred by the descriptor msgQ.
- The ***msg_ptr*** points to the empty message buffer. ***msg_len*** is the size of the buffer in bytes.
- The oldest msg of the highest priority is deleted from the queue and passed to the process in the buffer pointed by ***msg_ptr.***
- If the pointer ***msg_prio*** is not null, the priority of the received message is stored in the integer pointed by it
- The default behavior of ***mq_receive*** is to block if there is no message in the queue. However, if the O_NONBLOCK flag is enabled for the queue, and the queue is empty, ***mq_receive*** returns immediately with errno set to EAGAIN.
- On success, ***mq_receive*** returns the number of bytes received in the buffer pointed by msg_ptr.

5. Destroying a Message Queue

A creating Process can destroy a message queue using below API :

```
int
mq_unlink (const char *msgQ_name);
```

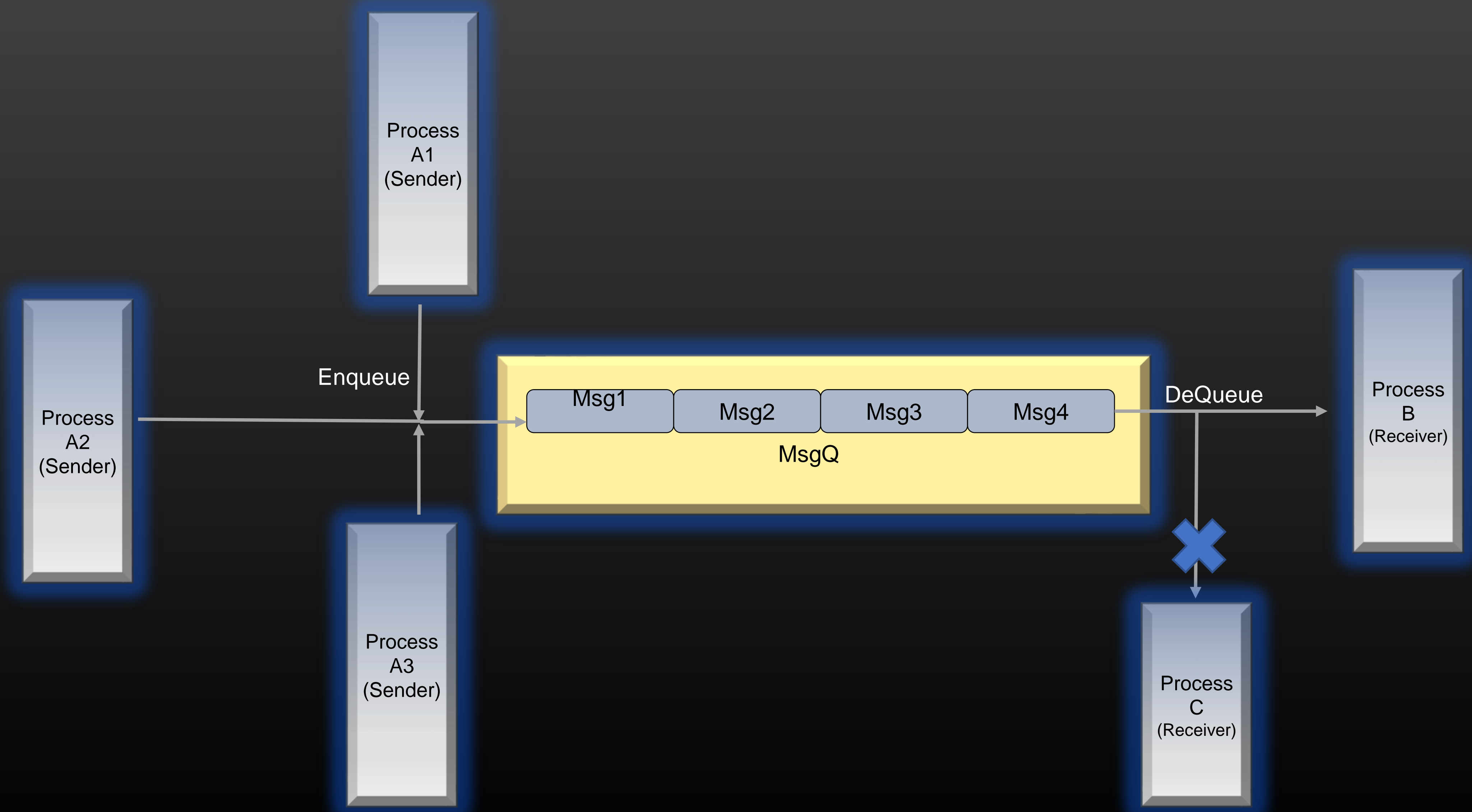
- ***mq_unlink*** destroys the msgQ (release the kernel resource)
- Should be called when the process has invoked mq_close() on msgQ
- return -1 if it fails, 0 on success
- Postpone if other processes are using msgQ

6. Using a Message Queue

- A Message Queue IPC mechanism usually supports N:1 communication paradigm, meaning there can be N senders but 1 receiver per message Queue
- Multiple senders can open the same msgQ using msgQ name, and enque their msgs in the same Queue
- Receiver process can dequeue the messages from the message Queue that was placed by different sender processes
- However, Receiving process can dequeue messages from different message Queues at the same time (Multiplexing using select())

  - A msg Queue can support only one client process
  - Client can dequeue msgs from more than one msg Queues
  - No limitation on Server processes

6. Using a Message Queue



IPC – Techniques -> Message Queue -> Illustration

Download code :

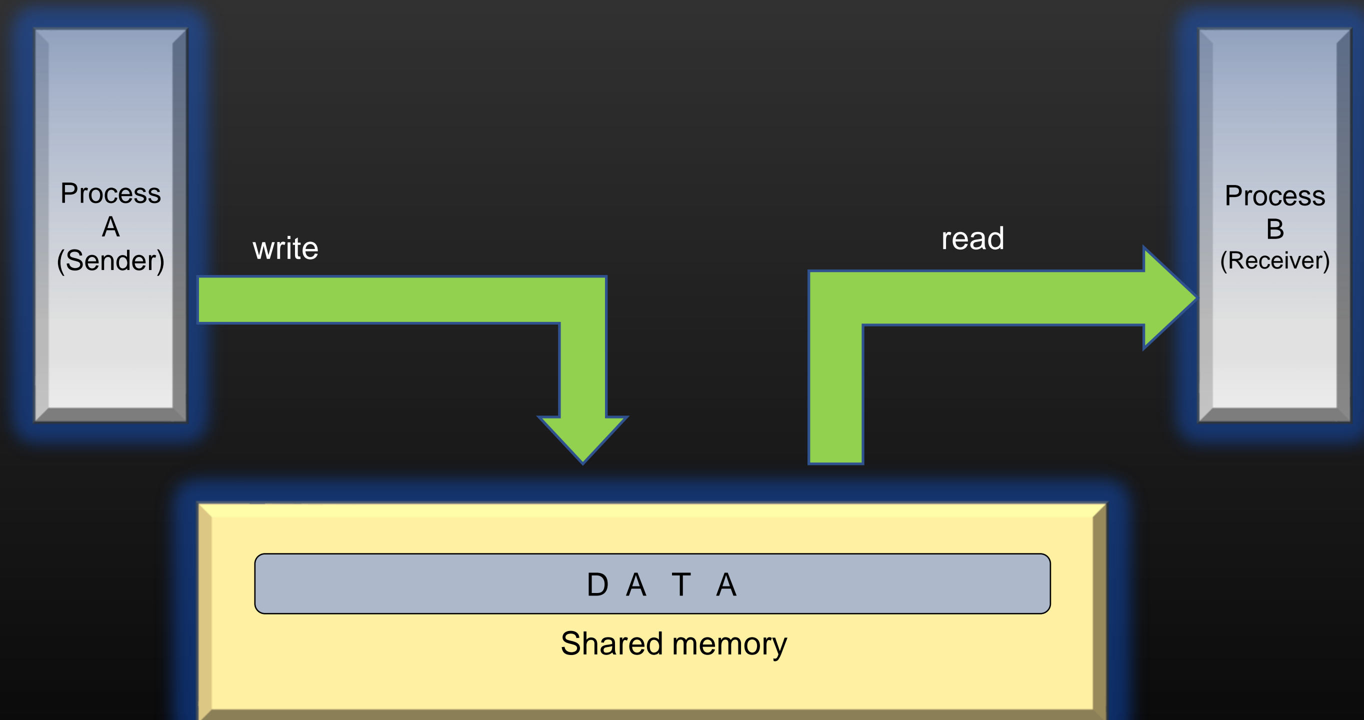
git clone https://github.com/sachinites/IPC

Dir : IPC/IPC/msgQ

Files : sender.c
recvr.c

Shared Memory

1. IPC using Shared Memory
2. Shared Memory Concepts
3. Shared memory management APIs
4. Using a shared memory as an IPC
5. Project



IPC – Techniques -> shared Memory

Pre-requisite knowledge

- To understand the concept of Shared Memory, we first need to understand some basics :
  - Virtual Address Space
  - Virtual Memory
  - Program Control Block

Virtual Address space & Virtual Memory

What exactly is Virtual Memory and Virtual Address space ?

- Virtual Memory is the total amount of memory your system has for a process.
- It is different from physical memory and is computer architecture dependent

For example :

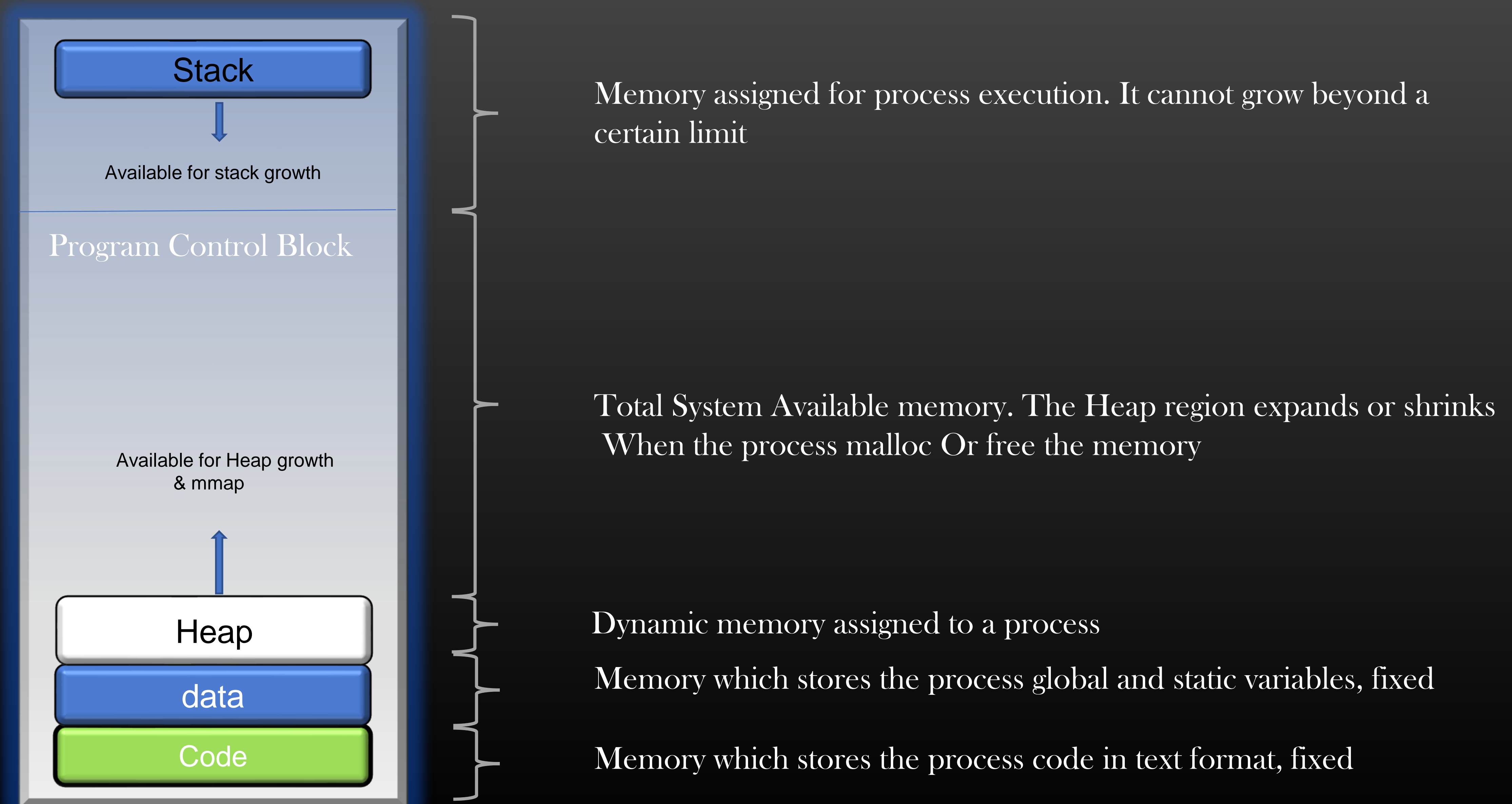
For a 32 bit system, Total virtual memory is 2^32 bytes (Fixed), and it is per process (Not for all processes combined !)
Whereas You can extend physical memory to 4GB, 8GB or 16GB (Variable) (Upper limit)

- 2^32 bytes !! Every byte has an address. Therefore, there are 2^32 Virtual addresses in a 32 bit system.
- Computer Programs works with Virtual Memory only, means, your program access only virtual addresses
- Your Program never knows anything about physical memory , all it knows that it has 2 ^ 32 bytes of virtual memory to store its data
- Each process in execution is allotted virtual memory for its usage, which can grow to maximum of 2^32 bytes
- The Memory assigned to a process is called process's virtual address space
- Every process running on a 32 bit system, can use maximum of 2 ^ 32 bytes of Virtual memory

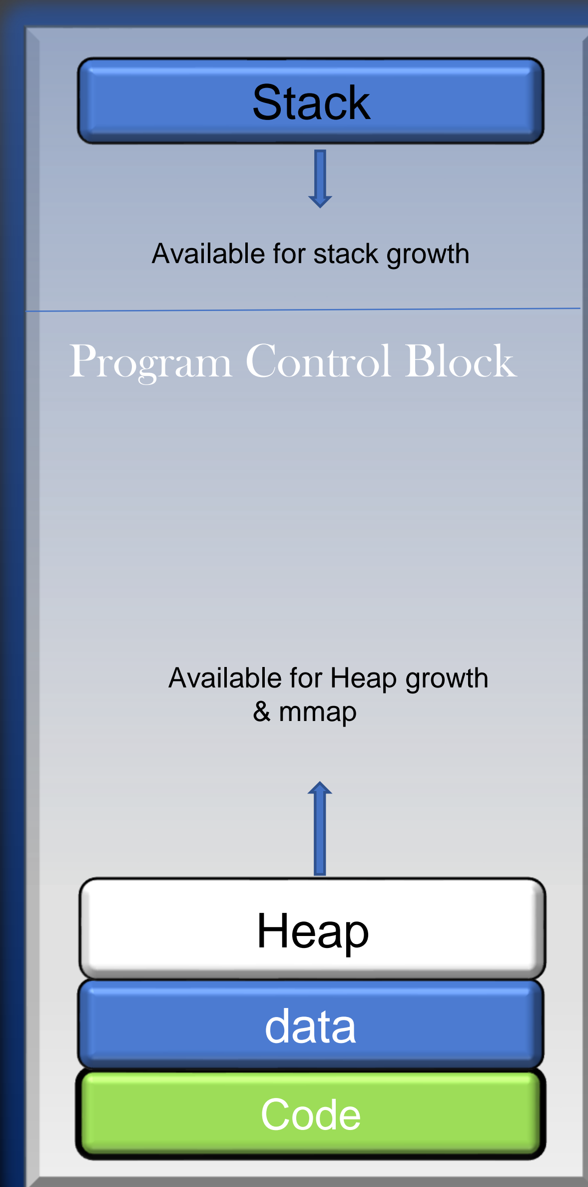
Program Control Block

- It is a diagrammatic logical representation of the of the process and its memory layout in Linux/Unix OS
- PCB helps us to understand how process virtual memory works during the course of execution of a process
- Let us see how PCB looks like for a linux process . . .

Program control block



Program control block




```
int global_var = 10;

int
main(int argc, char **argv){
        int i = 0;
        char *my_name = "Abhishek";
        static int j = 5;
        int *k;
        k =  malloc(100);
        foo(j);
        return 0;
}
```

VAS = data + Heap + Stack

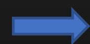
IPC – Techniques -> Shared Memory -> Basics

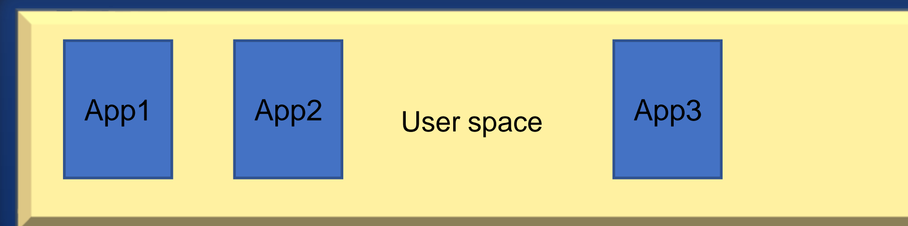
- Shared Memory is a chunk of Kernel Memory. No process is the owner of this memory.
- In linux, When a process executes, it is assigned a chunk of virtual memory of the system. This chunk of Memory is called process's virtual address space
- A process can access only the memory addresses which lies within its virtual address space
- If a process tries to access the address which lies outside its virtual address space, Linux OS kills the process (Segmentation fault)
- So, take it as thumb rule – A process is not allowed to read/write into any memory which is outside its virtual memory region
- A process Virtual memory region grows and shrinks as the process request more memory from OS or free the memory
- A given address which lies in virtual memory region of process P1 has no meaning for any other process in the system
- Let us picturize the above stated points ..

Kernel Memory

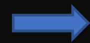
- Kernel has its own memory called kernel memory. Kernel uses its own memory for its own work – Schedulers, Timers, User process's memory Management, Interrupts and what not !
- No user space process is the owner of this kernel memory
- A user process can request kernel to create a kernel memory region
- This kernel memory region is outside the virtual address space of any process, hence no process can access this memory directly
- Through Shared memory approach, Application(s) can access kernel memory

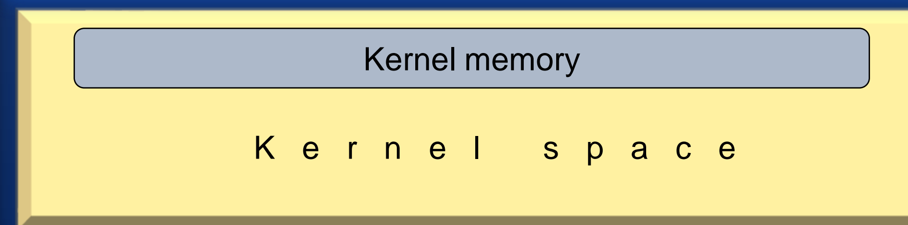
User Process's running on the system



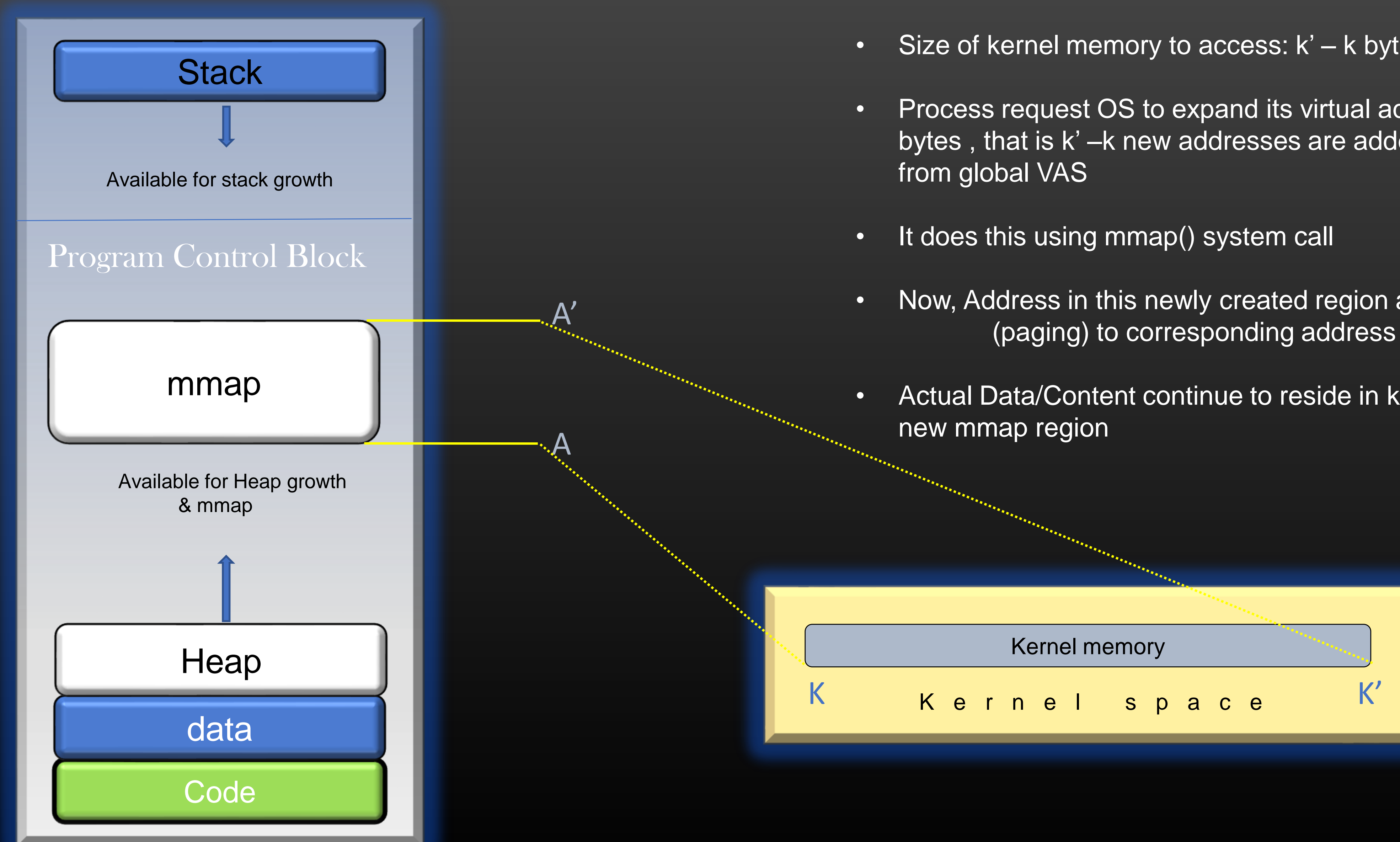


Underlying OS



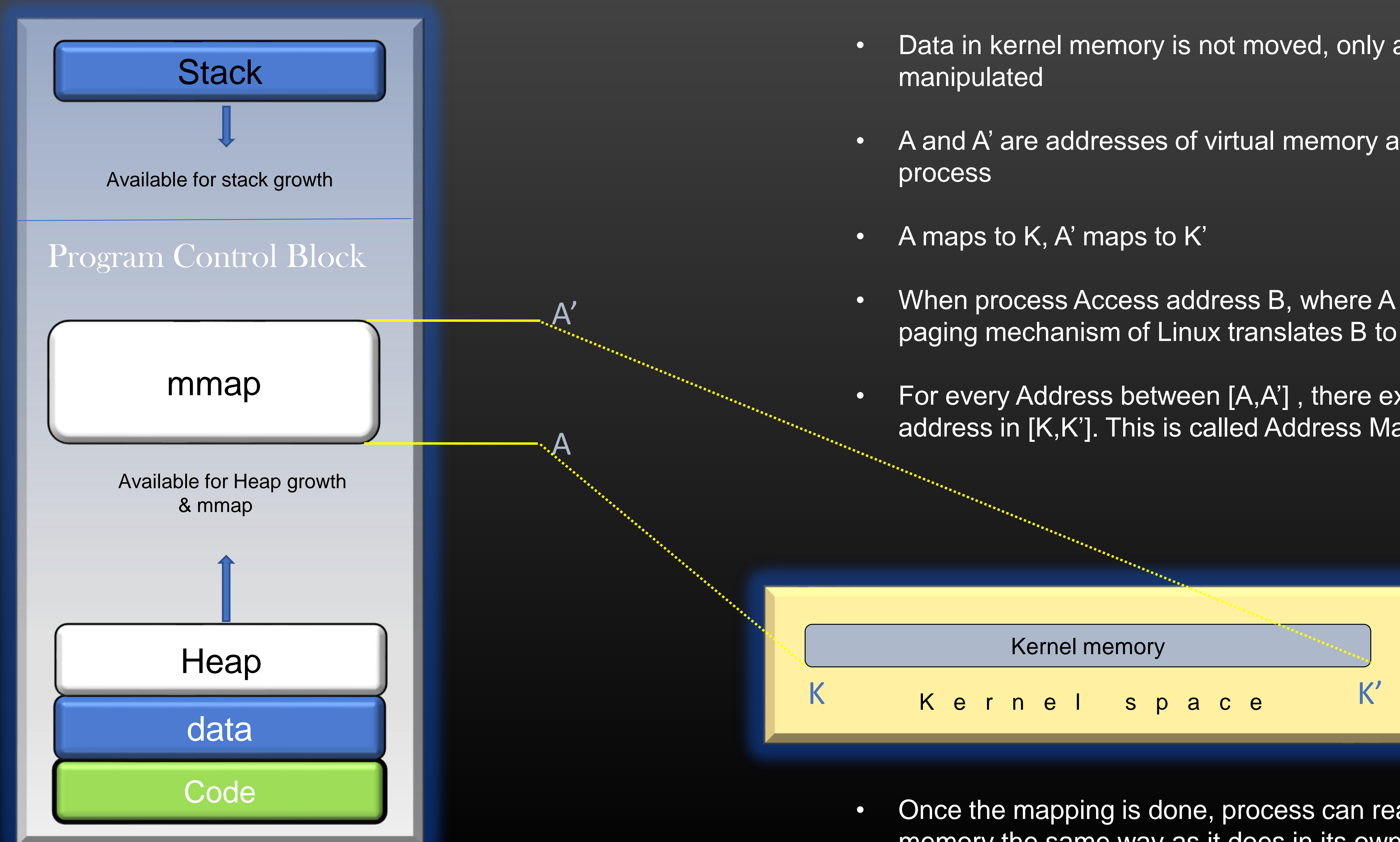


Accessing the Kernel Memory by a user space process



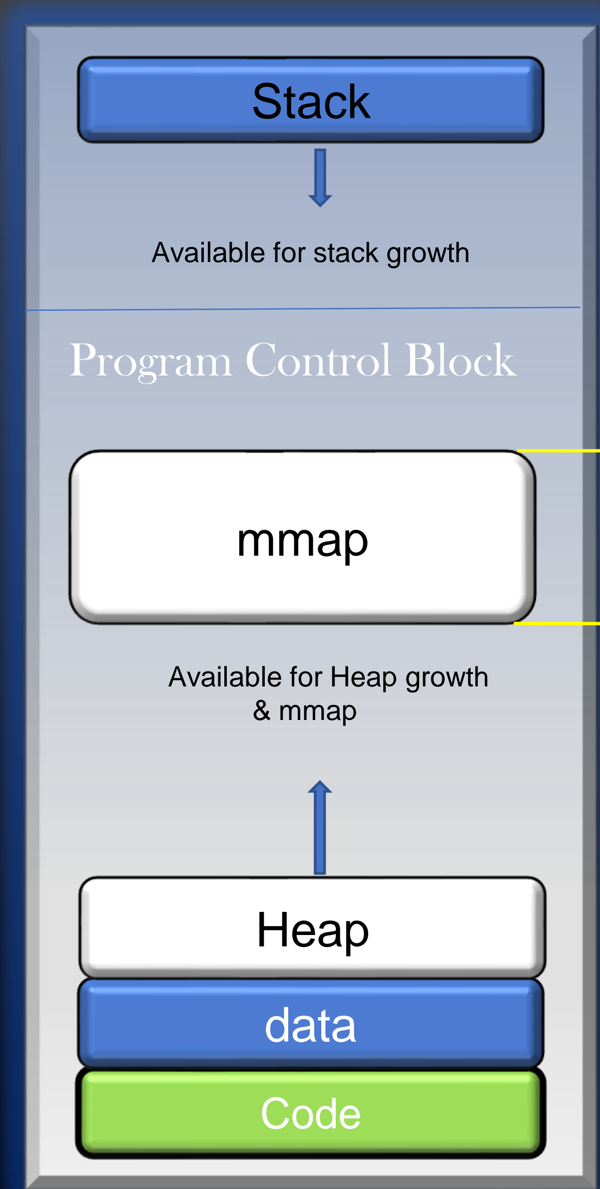
- Kernel memory staring address : k, end address k'
- Size of kernel memory to access: k' – k bytes
- Process request OS to expand its virtual address space by k' –k bytes , that is k' –k new addresses are added to process VAS from global VAS
- It does this using mmap() system call
- Now, Address in this newly created region are mapped by OS (paging) to corresponding address of Kernel memory
- Actual Data/Content continue to reside in kernel memory, not in new mmap region

Accessing the Kernel Memory by a user space process

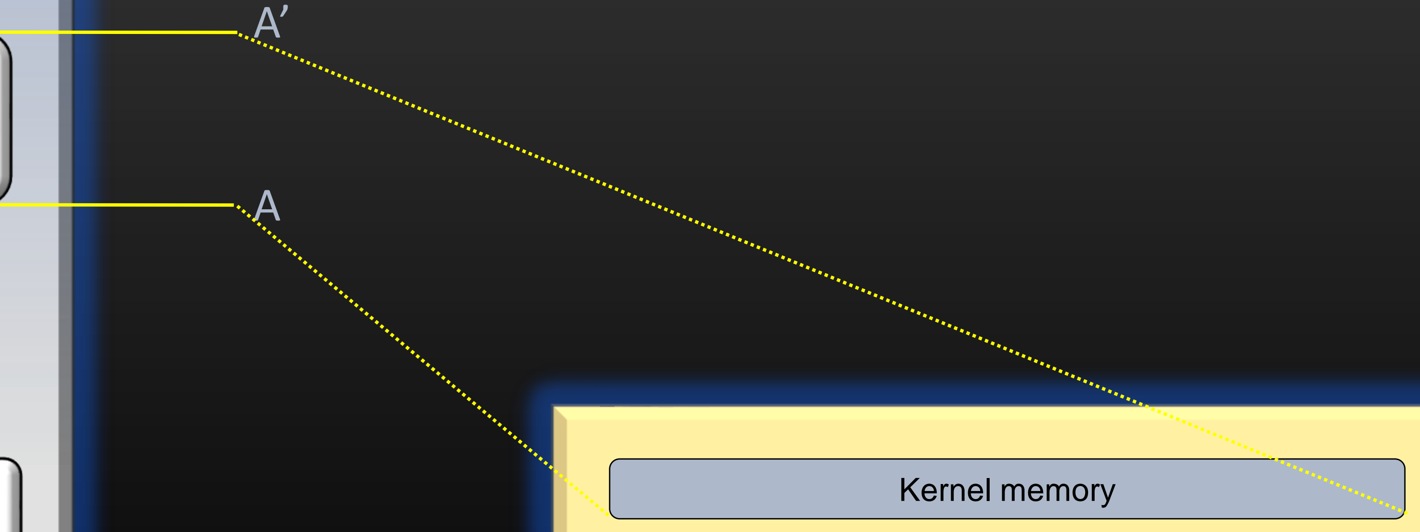


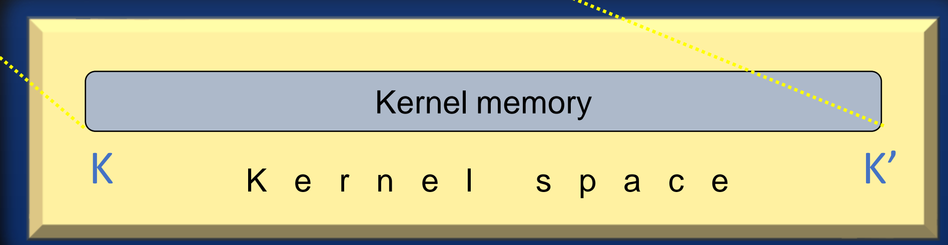
- A' – A = K' – K
- Data in kernel memory is not moved, only address mappings is manipulated
- A and A' are addresses of virtual memory address space of process
- A maps to K, A' maps to K'
- When process Access address B, where A <= B < A' , then paging mechanism of Linux translates B to B' where K <= B' < K'
- For every Address between [A,A'] , there exists corresponding address in [K,K']. This is called Address Mapping
- Once the mapping is done, process can read/write into kernel memory the same way as it does in its own virtual memory

Accessing the Kernel Memory by a user space process

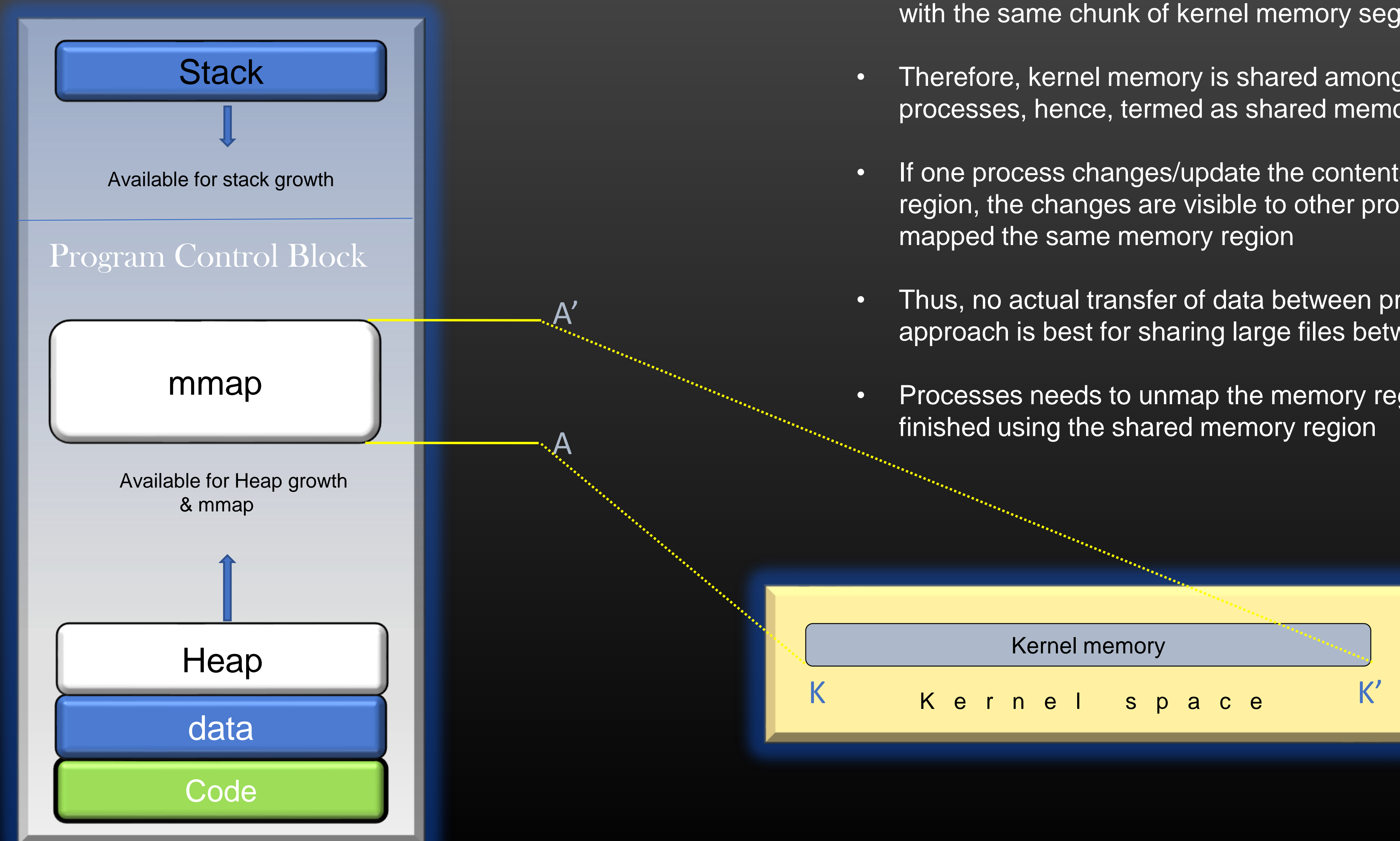


VAS = Data + Heap + Stack + mmap



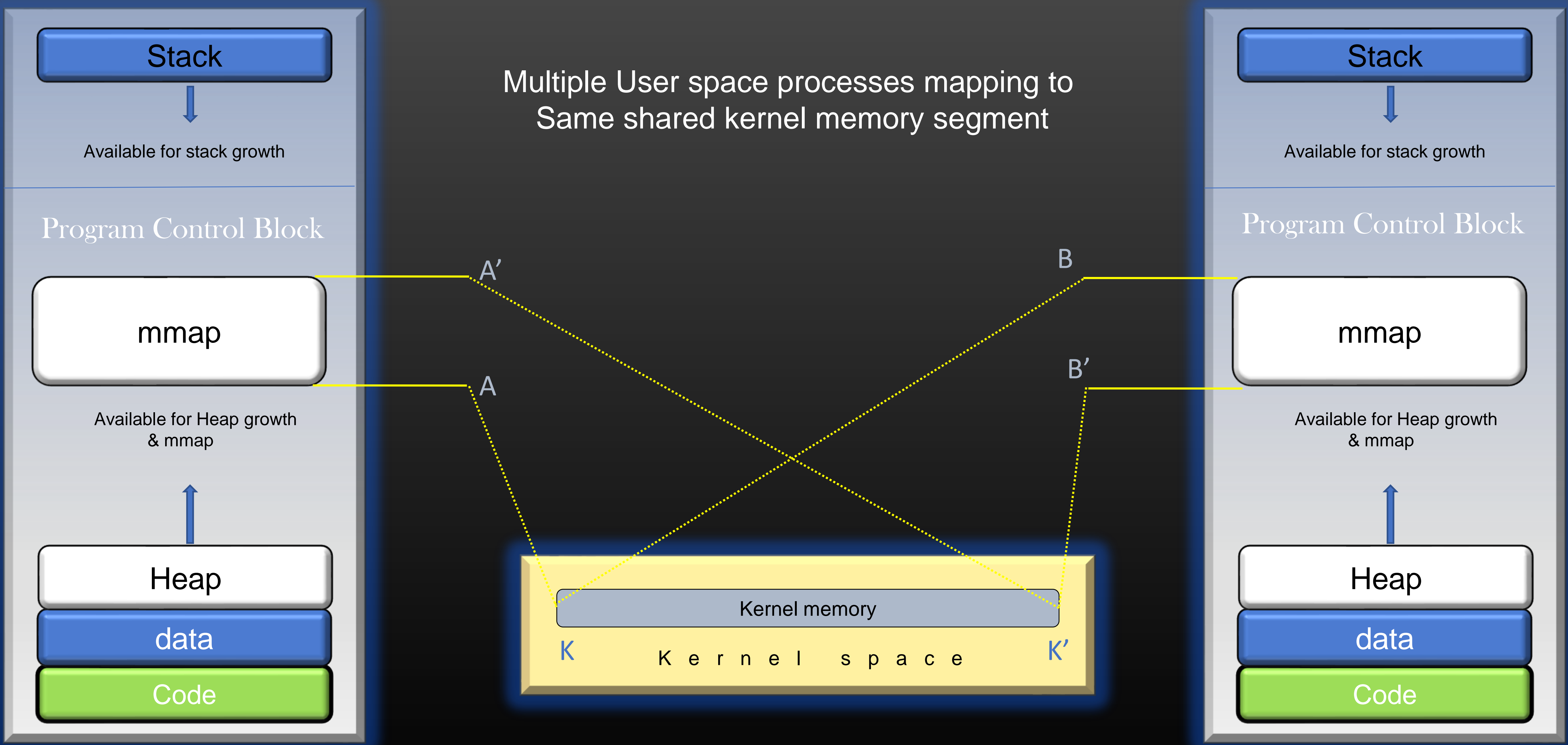


Accessing the Kernel Memory by a user space process



- Two or more processes can do this mapping at the same time with the same chunk of kernel memory segment
- Therefore, kernel memory is shared among multiple user space processes, hence, termed as shared memory
- If one process changes/update the contents of shared memory region, the changes are visible to other processes which have mapped the same memory region
- Thus, no actual transfer of data between processes, hence, this approach is best for sharing large files between processes
- Processes needs to unmap the memory region once it is finished using the shared memory region

Accessing the Kernel Memory by a user space process



Design Constraints for using Shared Memory as IPC

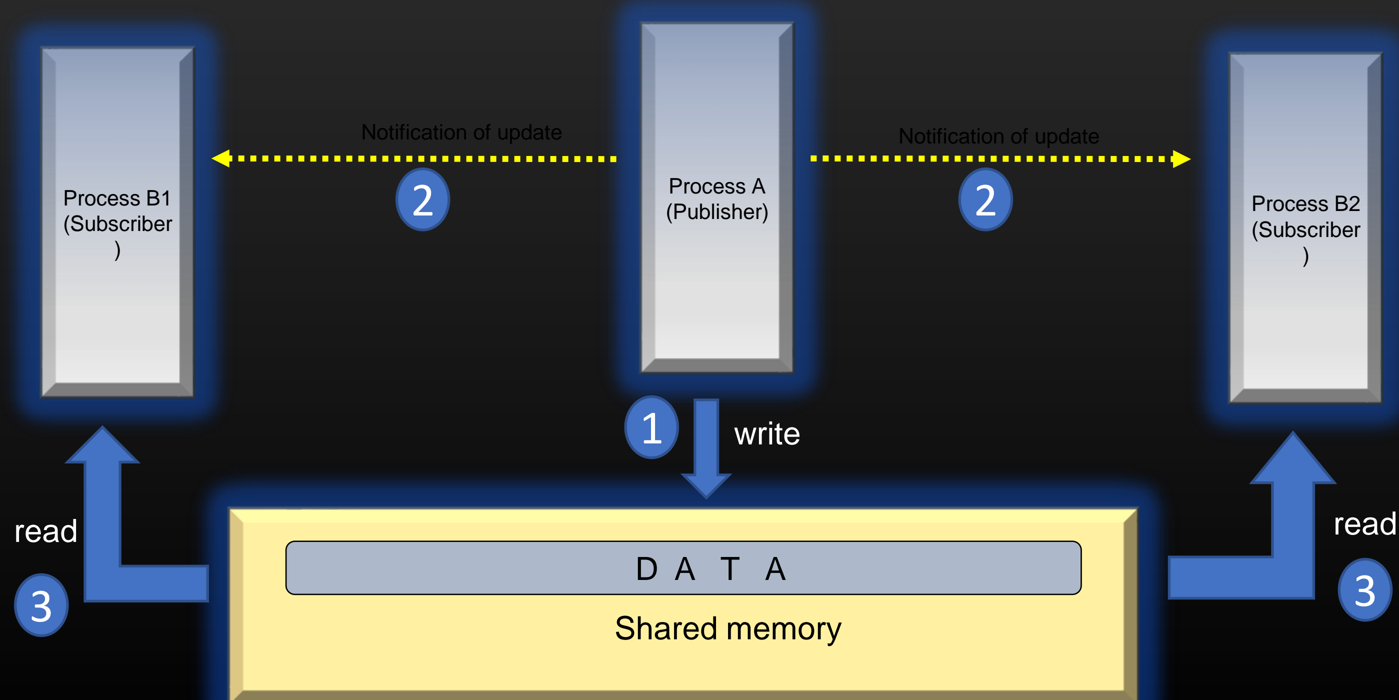
- **Recommendation when it is good to use Shared Memory IPC :**
  - Shared Memory approach of IPC is used in a scenario where :
    - Exactly one processes is responsible to update the shared memory (publisher process)
    - Rest of the processes only read the shared memory (Subscriber processes)
    - The frequency of updating the shared memory by publisher process should not be very high
      - For example, publisher process update the shared memory when user configure something on the software
- If multiple processes attempts to update the shared memory at the same time, then it leads to write-write conflict :

➢ We need to handle this situation using Mutual Exclusion based Synchronization

➢ Synchronization comes at the cost of performance
Because we put the threads to sleep (in addition to their natural CPU preemption) in order to prevent concurrent access to critical section

Design Constraints for using Shared Memory as IPC

- **When publisher process update the shared memory :**
  - The subscribers would not know about this update
  - Therefore, After updating the shared memory, publisher needs to send a notification to all subscribers which states that "shared memory has been updated"
  - After receiving this notification, Subscribers can read the updated shared memory and update their internal data structures, if any
  - The notification is just a small message, and can be sent out using other IPC mechanisms, such as Unix domain sockets Or Msg Queues
  - Thus IPC using shared memory has to be backed/supported by other type of IPCs



Shared Memory APIs

Shared Memory APIs

1. Creating Or opening a Shared memory region

```
int
shm_open (const char *name, int oflag, mode_t mode);
```

Name of shared memory
Eg "/Data_segment"

- Create a new shared memory object inside a kernel
- returns an integer which represents a file descriptor to this shared memory object
- returns -1 if call fails

- *oflag* – Following flags , OR-ing of one or more flags is supported
  - O_CREAT -- create a shared memory if it don't exist already
  - O_RDONLY – open a shared memory object in read only mode
  - O_WRONLY - open a shared memory object in write only mode
  - O_RDWR - open a shared memory object in read-write mode
  - O_APPEND – New data shall be appended to the existing content of shared memory
  - O_TRUNC – If specified, it truncates the shared memory object, if already exists, to zero bytes again
  - If O_EXCL flag is used together with the O_CREAT flag, then *shm_open* fails if shared shared memory object already exists with the same name

- *mode* – if the shared memory is newly created by the process, then mode is the file permissions applied on this memory object

- A newly created shared memory object is of size zero bytes
- It can be made of the desired size by using the *ftruncate* system call

Eg :

int shmfd =
shm_open("/datasegment", O_CREAT | O_RDWR | O_TRUNC, 0660);

Shared Memory APIs

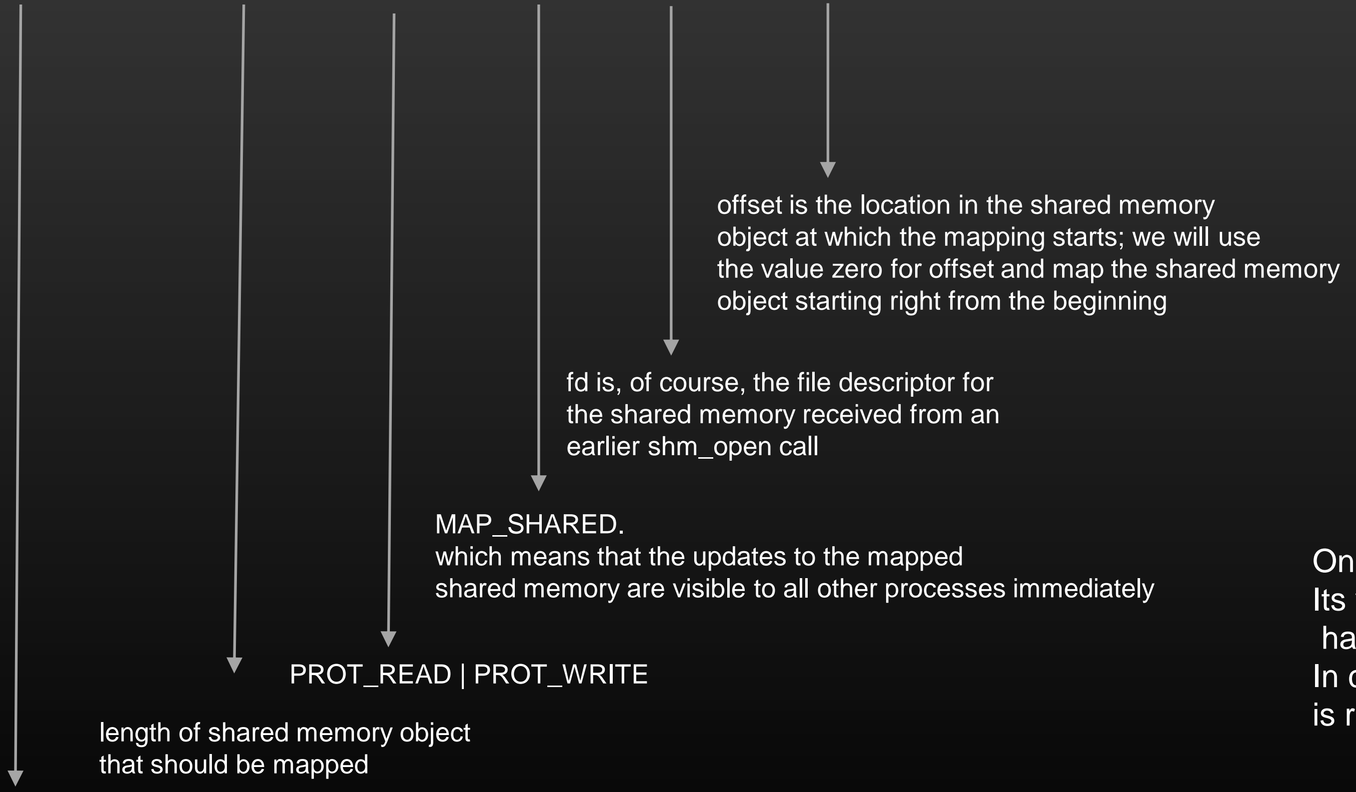
2. Setting the size of shared memory object

```
int
ftruncate (int shm_fd, size_t new_size);
```

- Resize the shared memory object to new size
- Returns -1 on failure
- Should be called after *shm_open* if required.

3. Mapping the shared memory into process's Virtual Address Space

```
#include <sys/mman.h>
void *
mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```



On success, mmap returns the pointer to the location in Its virtual address space where the shared memory region has been mapped.
In case of error, MAP_FAILED, which is, (void *) -1, is returned and *errno* is set to the cause of the error.

4. Un-Mapping the shared Memory

```
int
munmap (void *addr, size_t length);
```

- addr – Address in process virtual address space where Shared memory has been mapped (starting end of mapping) (Return value of mmap())
- length – length of mapping which we want to unmap
- unmapps the shared memory object at location pointed by *addr* and having size, *length*
- Returns -1 on failure, 0 on success
- It do not destroys the shared memory, it only destroy the mapping between shared memory in kernel space and virtual memory of the process

5. shm_unlink

```
int
shm_unlink (const char *name);
```

- De-associate shared memory with its name
- It is equivalent to like no shared memory with the same name exists anymore. The name shall be available to associate with new instance of shared memory
- return -1 on error, 0 on success

6. closing the shared memory file descriptor

```
int
close (int shm_fd)
```

- decrease the reference count of the shared memory object maintained in kernel space
- If reference count decreases to zero, shared memory and its content is destroyed (on some systems continue to exists until system is rebooted)
- return -1 on error, 0 on success

Points to remember

- Shared memory objects open and/or mapped by a process are both closed and unmapped when that process terminates prematurely, but they are not automatically unlinked.
- They will persist at least until manually unlinked or the system is rebooted.

Shared Memory IPC Demonstration

Src code : git clone http://github.com/sachinites/IPC
Dir path : cd IPC/IPC/SHM/Demo

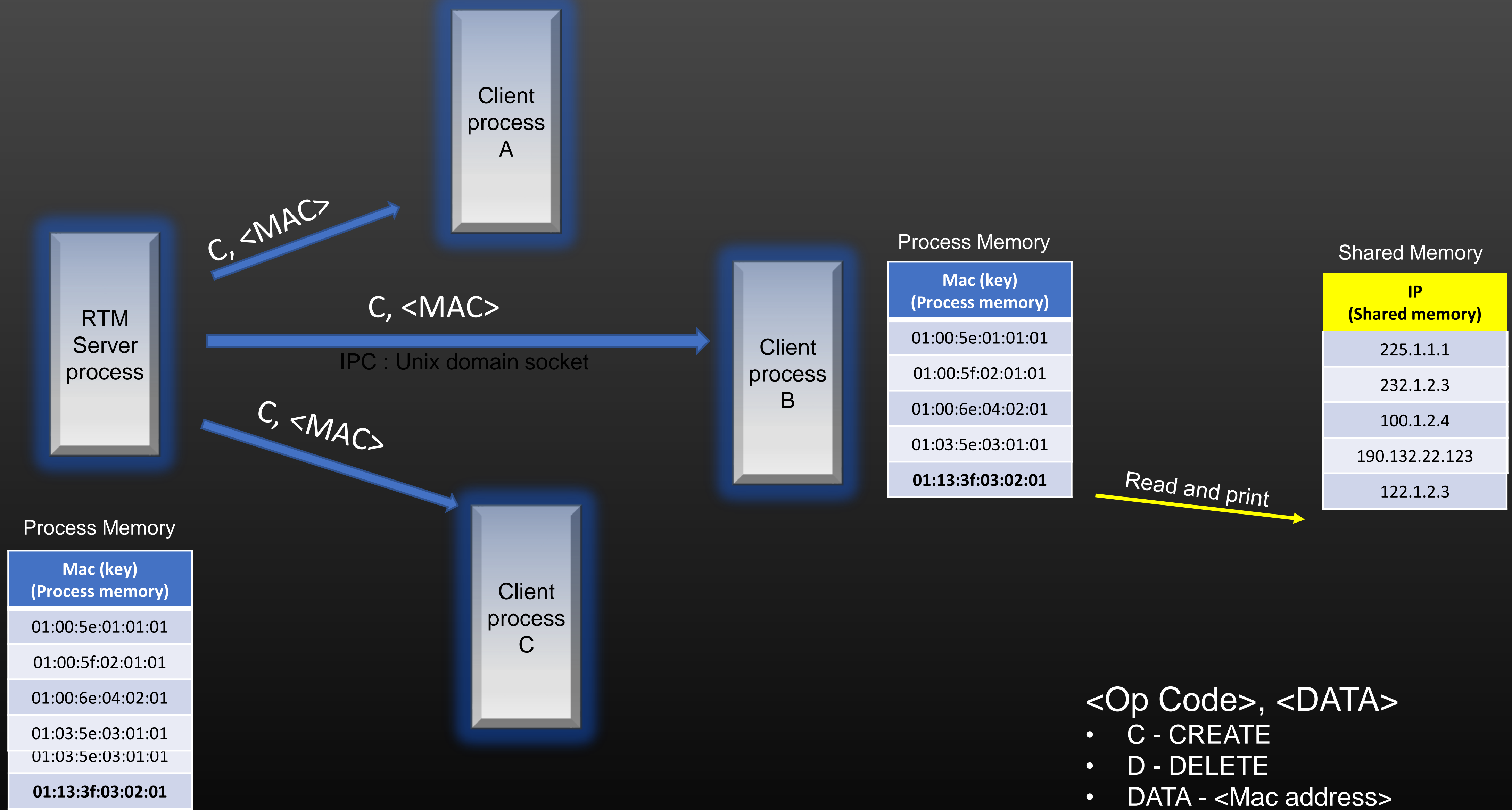
Files : writer.c - create and add contents to shared memory
reader.c – Read contents from shared memory created by writer.c
shm_demo.c – consists of functions and routines to manipulate shared memory

Code Walk . . .

Project on shared Memory IPC

- We shall extend the project that you have done on Unix domain sockets
- In Unix domain socket project, the routing table manager server process synchronized routing table to all connected clients using Unix Domain sockets IPC
- In this project, the routing table manager server process also maintains another table called ARP table and it synchronizes it to all subscribed clients using Shared Memory IPC
- Let us discuss the project step by step. The functionality of Previous project will stay as it is, no changes

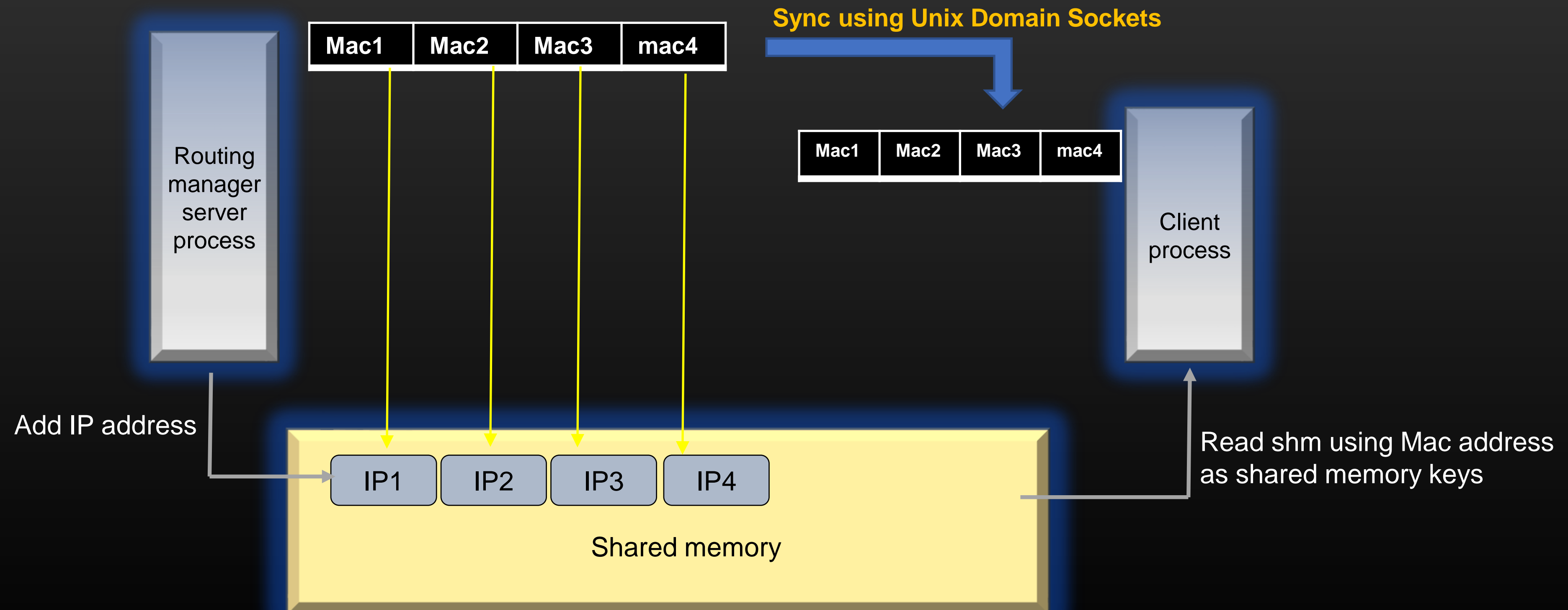
Project on shared Memory IPC



Provide a menu-driven approach to show Mac table Contents on Server and Client processes

Project on shared Memory IPC

- All the processes – Server and clients stored only the shared memory key (the Mac address) in its internal data structure
- Server process adds the Data – the ip address in the shared memory corresponding to the mac address (key)
- Server process then syncs only the mac address (shm key) to rest of the connected clients using Unix domain sockets
- Client having received this new key (the mac address), stores the mac address in their private list. Using this mac address as key they can also access the corresponding ip address which was added by Server process in shared memory
- When the fresh clients get connected to server process, besides routing table, server also syncs the entire local mac address list to new client using unix domain socket



Signals

1. What are Signals ?
2. Types of Signals
3. Capturing Signals
4. Using Signals as IPC

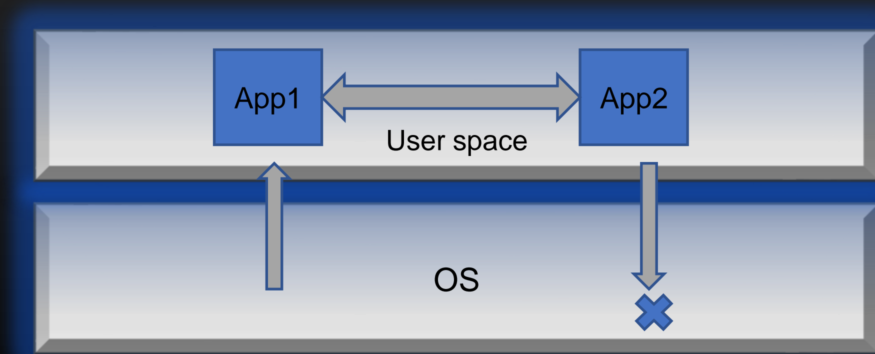
IPC – Techniques -> Signals

Definition :

A system message sent from one process to another, not usually used to transfer data but instead used to remotely command the partnered process.

Eg :

The Cricket Umpire *signaling* the batsman is out. It is an example of Signal, not a data transfer
Signals are usually small msg (1 or 2 byte). Recipient needs to interpret its meaning.

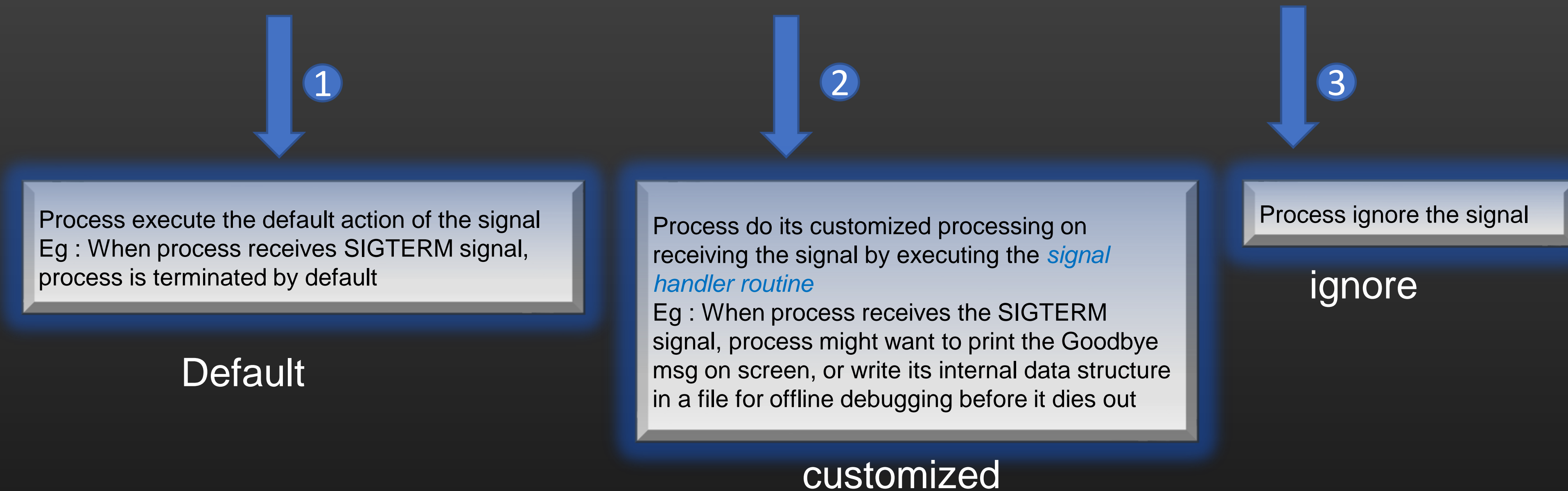


Signals Works from :

OS -> Application
Between Applications

From Application to OS - No

When a process receives a Signal, Either of the three things can happen :



Few points on Signal handler routines :

- A signal handler routine is a special function which is invoked when the process receives a signal.
- We need to register the routine against the type of signal (that is, process needs to invoke the function F when signal S is received)
- Signal handler routine are executed at the highest priority, preempting the normal execution flow of the process

Well known Signals in Linux

- **SIGINT** – interrupt (i.e., Ctrl-C)
- **SIGUSR1 and SIGUSR2** – user defined signals
- **SIGKILL** – sent to process from kernel when kill -9 is invoked on pid.
  This signal cannot be caught by the process.
- **SIGABRT** – raised by abort() by the process itself. Cannot be blocked.
  The process is terminated.
- **SIGTERM** – raised when *kill* is invoked. Can be caught by the process to execute user defined action.
- **SIGSEGV** – segmentation fault, raised by the kernel to the process when illegal memory is referenced.
- **SIGCHILD** – whenever a child terminates, this signal is sent to the parent.
  Upon receiving this signal, parent should execute wait() system call to read child status.
  Need to understand fork() to understand this signal

Three ways of generating Signals in Linux

1. Raising a signal from OS to a process (ctrl – C etc)
2. Sending a signal from process A to itself (using *raise()*)
3. Sending signal from process A to process B (using *kill()*)

We shall go through examples of each . . .

1. A Process can Trap the signal received from OS and perform user defined function when signal is Received

```
#include <signal.h>

/*Register User defined function ctrlC_signal_handler
which will be invoked When ctrl + C is pressed*/

signal (SIGINT, ctrlC_signal_handler);

static void
ctrlC_signal_handler(int sig){
    printf("Ctrl-C pressed\n");
    printf("Bye Bye\n");
    exit(0);
}
```

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static void
ctrlC_signal_handler(int sig){
    printf("Ctrl-C pressed\n");
    printf("Bye Bye\n");
    exit(0);
}

static void
abort_signal_handler(int sig){
    printf("process is aborted\n");
    printf("Bye Bye\n");
    exit(0);
}

int
main(int argc, char **argv){

    signal(SIGINT, ctrlC_signal_handler);
    signal(SIGABRT, abort_signal_handler);
    char ch;
    printf("Abort process (y/n) ?\n");
    scanf("%c", &ch);
    if(ch == 'y')
        abort();
    return 0;
}
```

2. A Process can send a signal to itself using *raise()*

```
int
raise (int signo);
```

- The signal denoted by *signo*, when raised using raise() is delivered to process itself

3. A Process can send a signal to another process using *kill()*

```
int
kill (int process_id, int signo);
```

- The sending process needs to know the recipient process id to send a signal

```c
#include <stdio.h>
#include <signal.h>

int
main(int argc, char **argv){


    kill(5939, SIGUSR1);
    scanf("\n");
    return 0;
}
```

Kill_sender.c

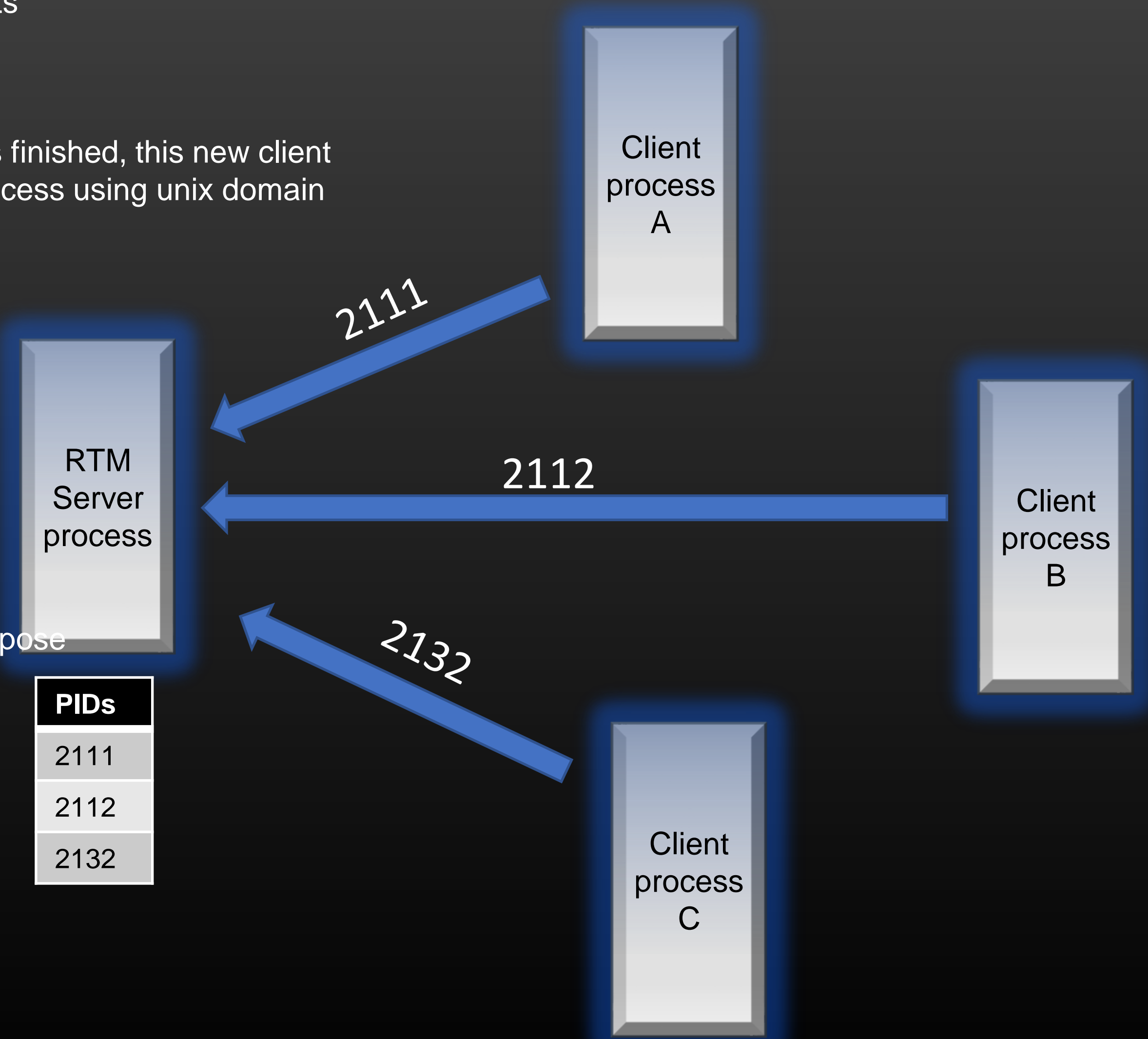
```c
#include <stdio.h>
#include <signal.h>

static void
signal_handler(int sig){

    printf("Signal %d recieved\n", sig);
}

int
main(int argc, char **argv){


    signal(SIGUSR1, signal_handler);
    scanf("\n");
    return 0;
}
```

Kill_recv.c

- 5939 is the pid of the kill_recv.c process
- SIGUSR1, SIGUSR2 are reserved for user defined signals

Project on Signals

- We shall continue to extend our same project
- Till now, when new client connects to Server process, Server syncs entire Routing table and ARP table to this new connected clients using Unix Domain Sockets and Shared memory respectively
- Now, When a client gets connected to the server process, and Synchronization is finished, this new client sends its own process id (*getpid()*) to the routing manager server process using unix domain sockets
- Server stores the process id of all clients in a list or something
- Now, add a choice in Menu on Server side choosing which Server flushes off its entire routing table and ARP table in shared memory
- After flushing , the server sends SIGUSR1 signal to all connected clients using their pids with *kill()*. Having received this signals, clients are suppose to flush off their own routing table and ARP table copies
- Also, whenever the client terminate the unix domain connection with the server, Client should clean all its tables it is maintaining. Before closing the connection, client should send one last msg to server, Server having received this last msg should remove pid of this client from its list



Multiplexing

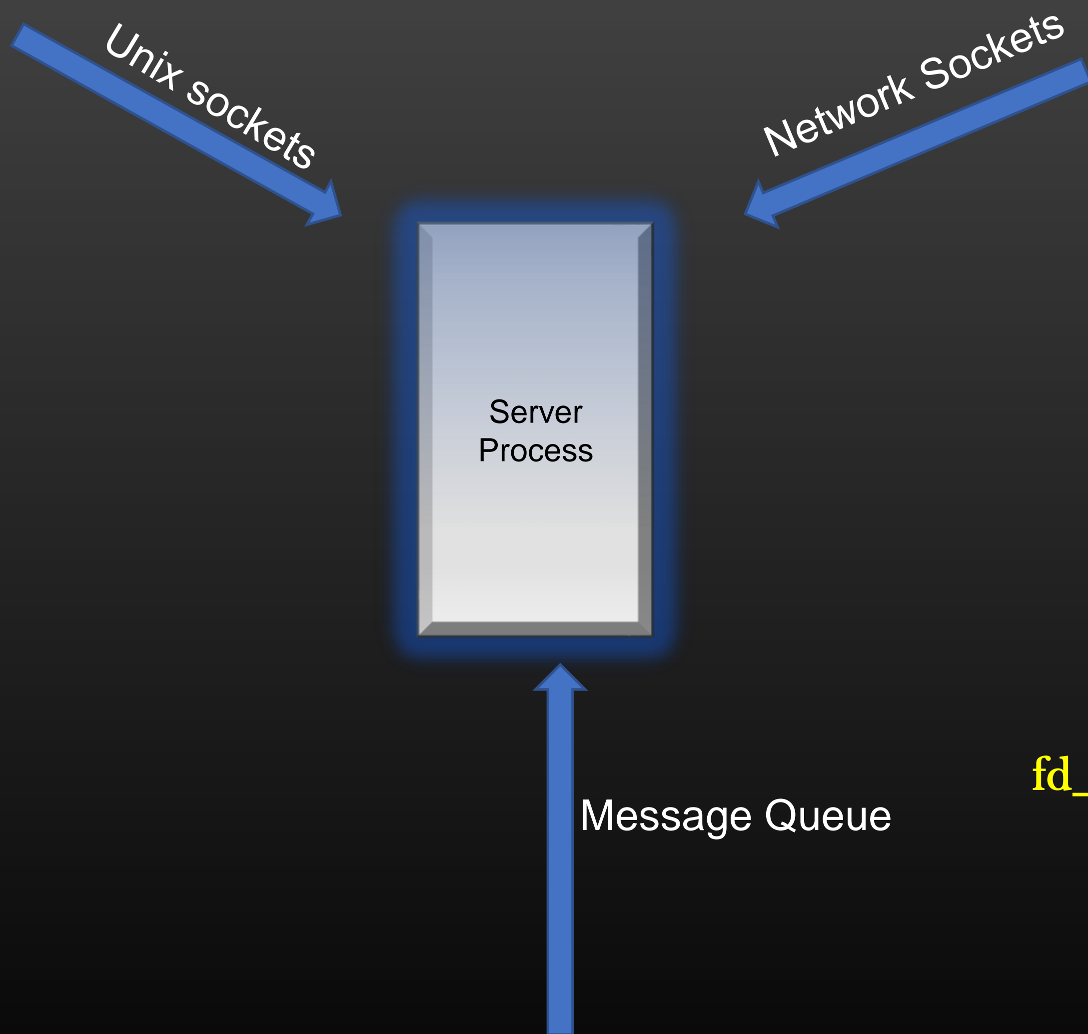
How can a single process can receive the data from multiple IPC mechanism at the same time ?

We have a process which can receive data Asynchronously from :

- Network Socket
- Unix Domain Socket
- Console
- Message Queue

How to design such a process which would pick up the data sent from any IPC mechanism and process it ?

IPC – Techniques -> Multiplexing



- If you notice, Network Sockets, Unix Sockets & Message Queues have File descriptors
- We have already learned multiplexing in the context of sockets
- In this case, simply add all file descriptors to *fd_set* and invoke *select()* on this set
- *select()* will unblock if server receives data on any FD via any IPC mechanism
- Simple !!

fd_set = data sockets of network clients + Master Network skt
\+
data socket of unix domain clients + connection skt
\+
FDs of all message Queues
\+
0 (console)