Agenda

- How to Do this Course ?
- Getting Started with Writing Linux Kernel Modules (LKM)
- Introducing - The Netlink Sockets
  - Theory, Architecture and Design
- Basics - The Netlink Message Structures and Types
- User Space to Kernel Space Interaction
- Kernel Space to User Space Interaction
- Kernel to Multiple User Space Processes Interaction
- Event Notification from Kernel Space To User Space

Pre-requisites

- You are good with C programming
- Good with basic DS such Linked List, Trees etc
- Audience is not novice or beginner,
  therefore fast paced course, without loss of informati
- Must have basic experience with socket programming

Assumptions

- Zero Kernel Programming knowledge
- Able to Google right questions and right doubts
- Mature Audience
- Zeal to Learn and Excel

Src Codes : https://github.com/sachinites/UdemyCourseOnNetlink

How to do kernel Programming Based Courses ?

- Install Ubuntu as a Virtual Machine using any Virtualization software (VMWare Workstation / Virtual Box)
  - You VM will freeze if kernel crash
  - Solution : Reboot VM
  - Debugging : kprintf
- Install exactly same version of ubuntu as I am running (Mandatory)
  - No compilation errors
  - No unnecessary head scratching and time waste
  - We all are in sync and on same page
  - My codes will run on your machine and your codes will run on mine
  - Lot of help present on internet especially for ubuntu
  - We all shall be looking same kernel source code
- Tools :
  - GCC compiler
  - Use Github

➢ Setting up the Development Environment

- Image to use : http://releases.ubuntu.com/19.04/ubuntu-19.04-desktop-amd64.iso
  - AMD or Intel Don't matter, Image is compatible with both !
  - 64 bit system

- Browsing Kernel Source Code – Locally Or from Web
  - Run uname –r on terminal, it will show kernel version

    vm@ubuntu:/usr/src$ uname -r

    5.0.0-36-generic [5 kernel version, 0 – Major number, 0 – Minor Number, 36 – Bug fix number]

    Web link : https://elixir.bootlin.com/linux/v5.0/source/include/linux

    Download : https://launchpad.net/ubuntu/+archive/primary/+sourcefiles/linux/5.0.0-32.34/linux_5.0.0.orig.tar.gz

    gzip -d linux_5.0.0.orig.tar.gz

    tar -xvf linux_5.0.0.orig.tar (Browse the code using cscope tool)

- Make your own notes – whatever you feel like
  - Github is the best place for note making and preserving all your source codes
  - You should be able to revise after couple of years if you happen to work in system programming domain

➢ Installing Linux Kernel Headers

➢ Kernel Header File is required to compile Kernel Code that we shall be going to write in this course

```
vm@ubuntu:/usr/src$ uname -r
5.0.0-36-generic
vm@ubuntu:/usr/src$ ls -l
total 160
-rw-r--r--  1 root root 145342 Oct 24 21:07 'download_script.php?src_id=9679'
drwxr-xr-x 25 root root   4096 Oct 22 06:58  linux-headers-5.0.0-32
drwxr-xr-x  8 root root   4096 Oct 22 06:58  linux-headers-5.0.0-32-generic
drwxr-xr-x 25 root root   4096 Nov 14 06:27  linux-headers-5.0.0-36
drwxr-xr-x  8 root root   4096 Nov 14 06:27  linux-headers-5.0.0-36-generic   << required !!
```

If not present, run the below cmds :

```
sudo apt update
sudo apt install linux-headers-$(uname -r)
```

What is Linux Kernel Module (LKM)

- Modules are pieces of code that can be loaded and unloaded into the kernel upon demand
- They extend the functionality of the kernel without the need to reboot the system Or recompiling the Linux kernel
- Eg : Almost all Device Drivers are written as Linux Kernel Module
- Linux Kernel is shipped with hundreds of kernel modules with them
- The LKM, when loaded, becomes a part of linux kernel
  - Load kernel module : sudo insmod <lkm.ko>
  - Unload LKM : rmmod <lkm>
  - Check loaded LKMs : lsmod
- Lets Code . . .
- Download :

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

Summary

- This is Quick short Demo on how to write LKMs, Compile, load and run !

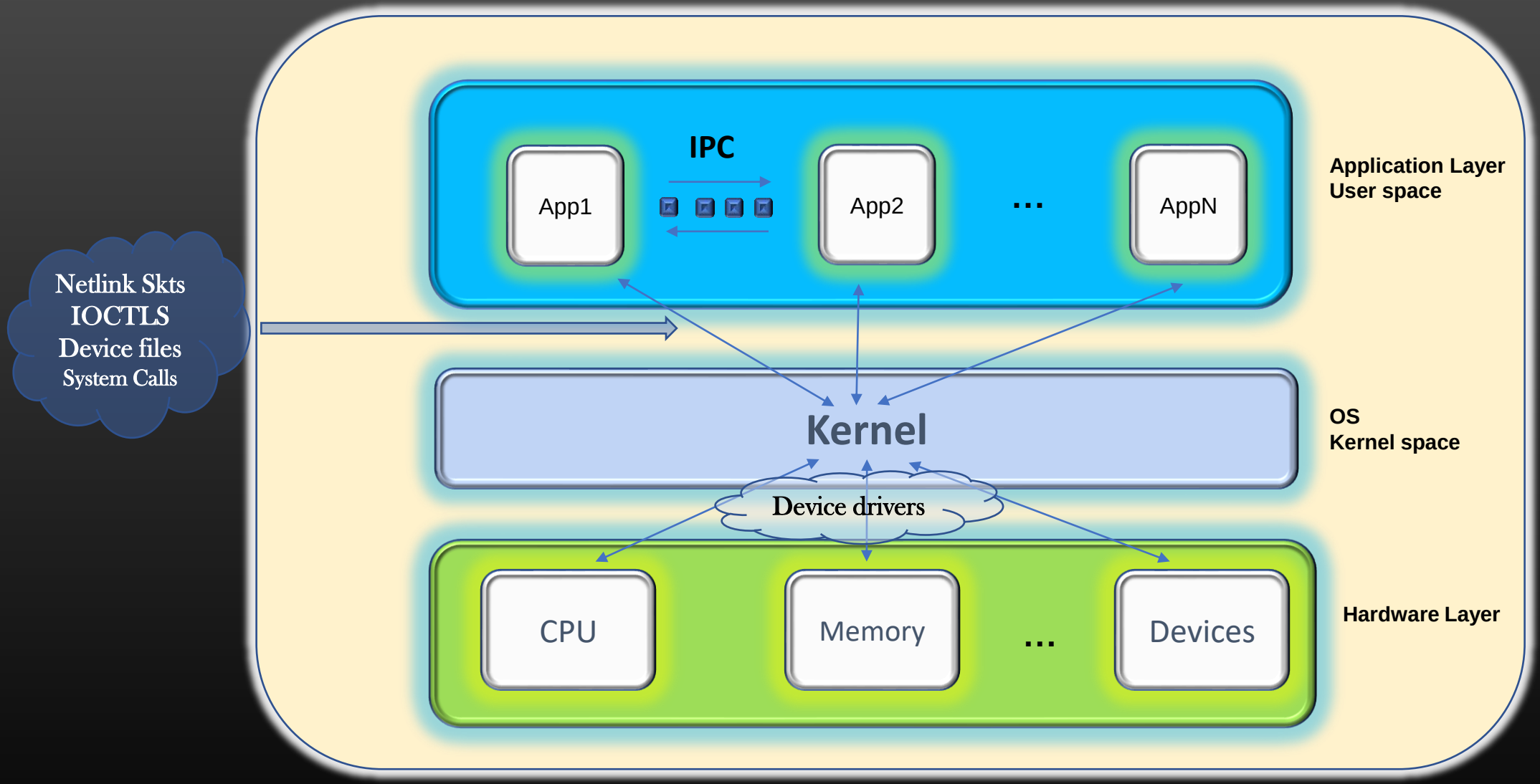
Linux Kernel Programming

IPC between Userspace and kernel space

Linux Kernel Programming

Understanding Netlink Sockets

- IPC between Userspace and kernel space



Computer Architecture

➢ Introducing Netlink Sockets

- Netlink Sockets are especially created to facilitate clean bidirectional communication between user space and kernel space

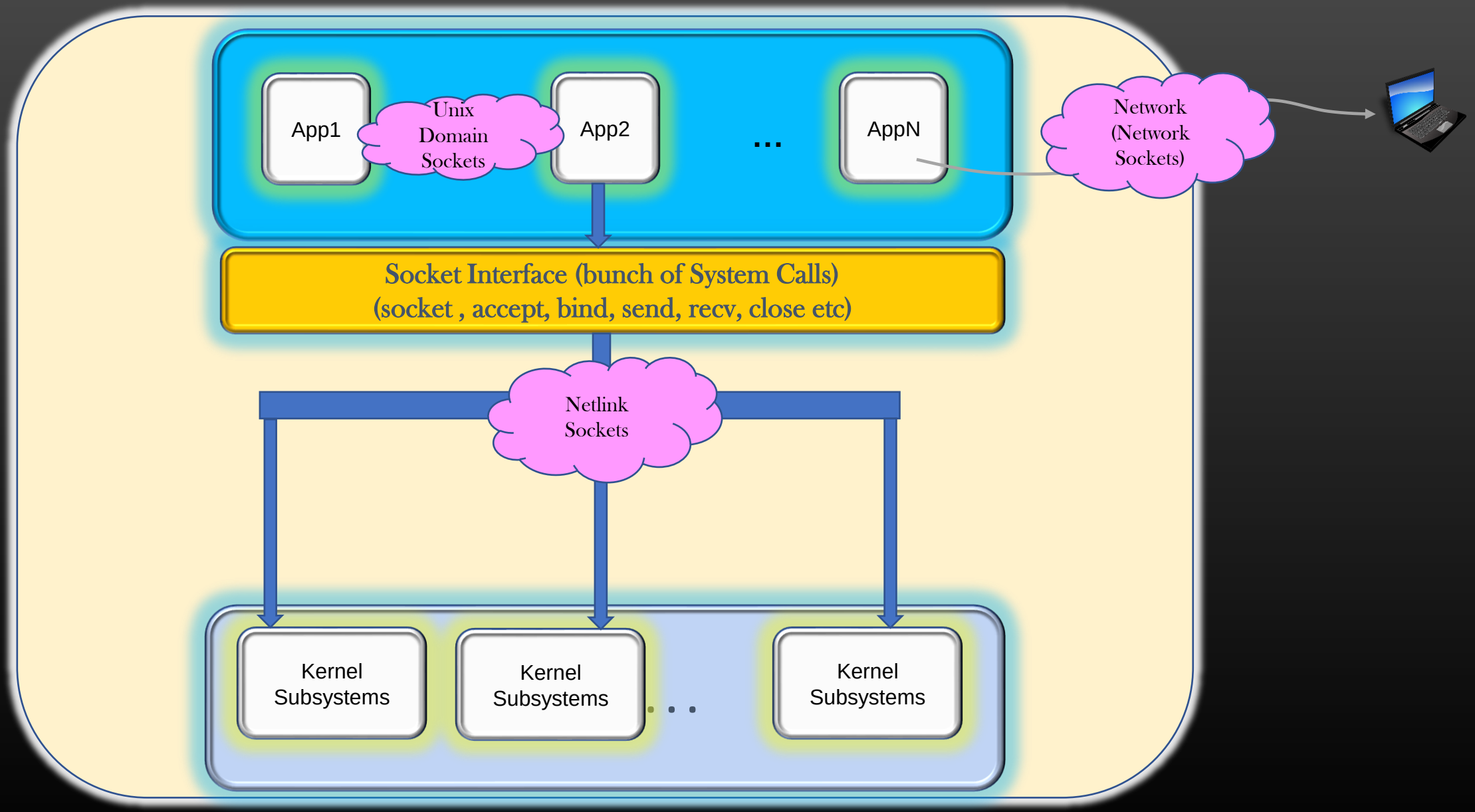
- Other Techniques can also be used for US <-> KS communication, but they were not invented for this purpose
  - Eg : Ioctls, device files, System calls

- A Socket based technique was developed to build the unified interface using which user space applications (USA) can interact with various kernel subsystems

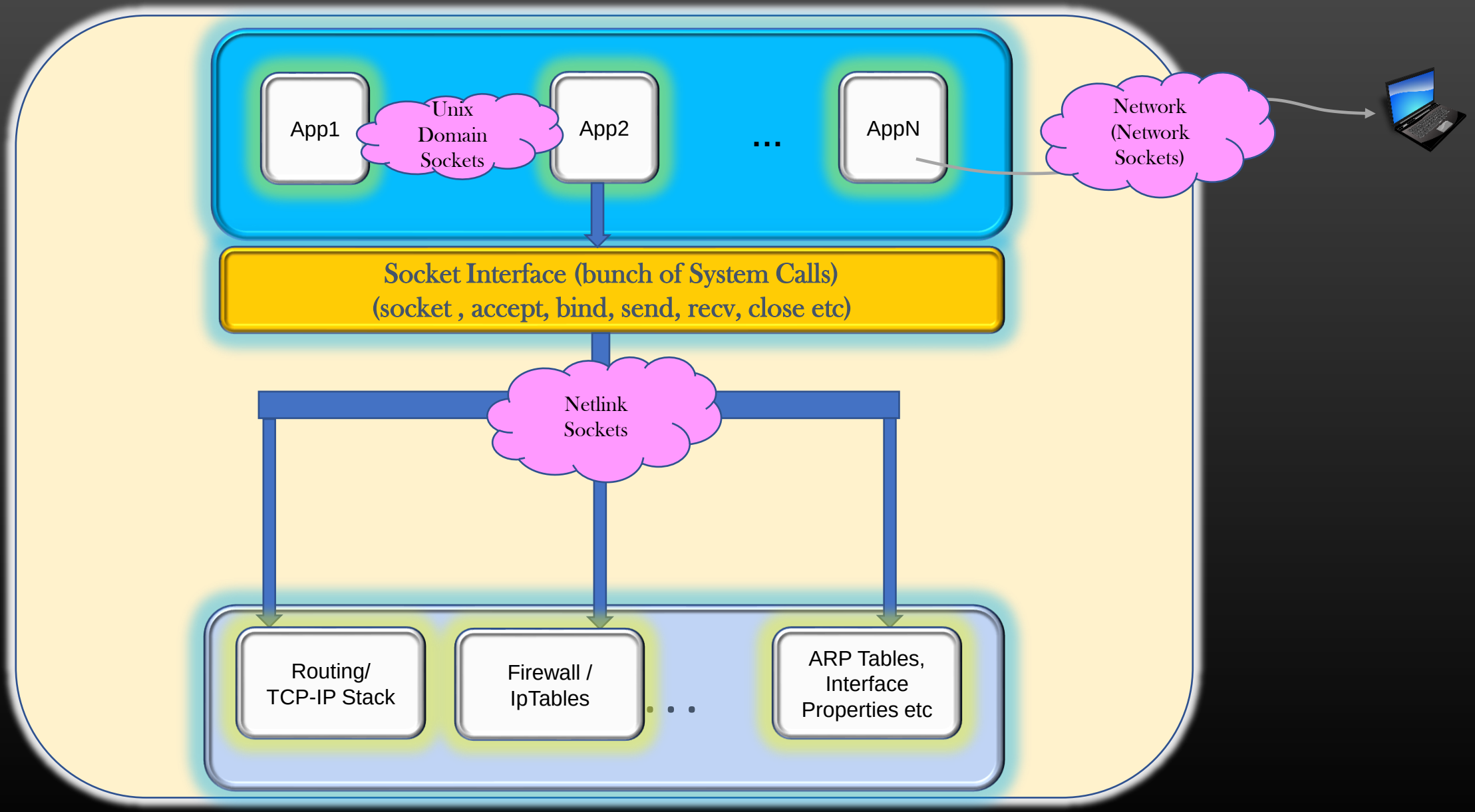
int skt_fd = socket (AF, Socket Type, Protocol ID)

These 3 arguments determine :
1. Socket Address Family
2. Communication Type : Datagram based Or Stream Based
3. Protocol used for communication

- Thus, socket interface is *unified* – depending on arguments passed, we set up communication properties – whom to communicate, what to communicate, how to communicate

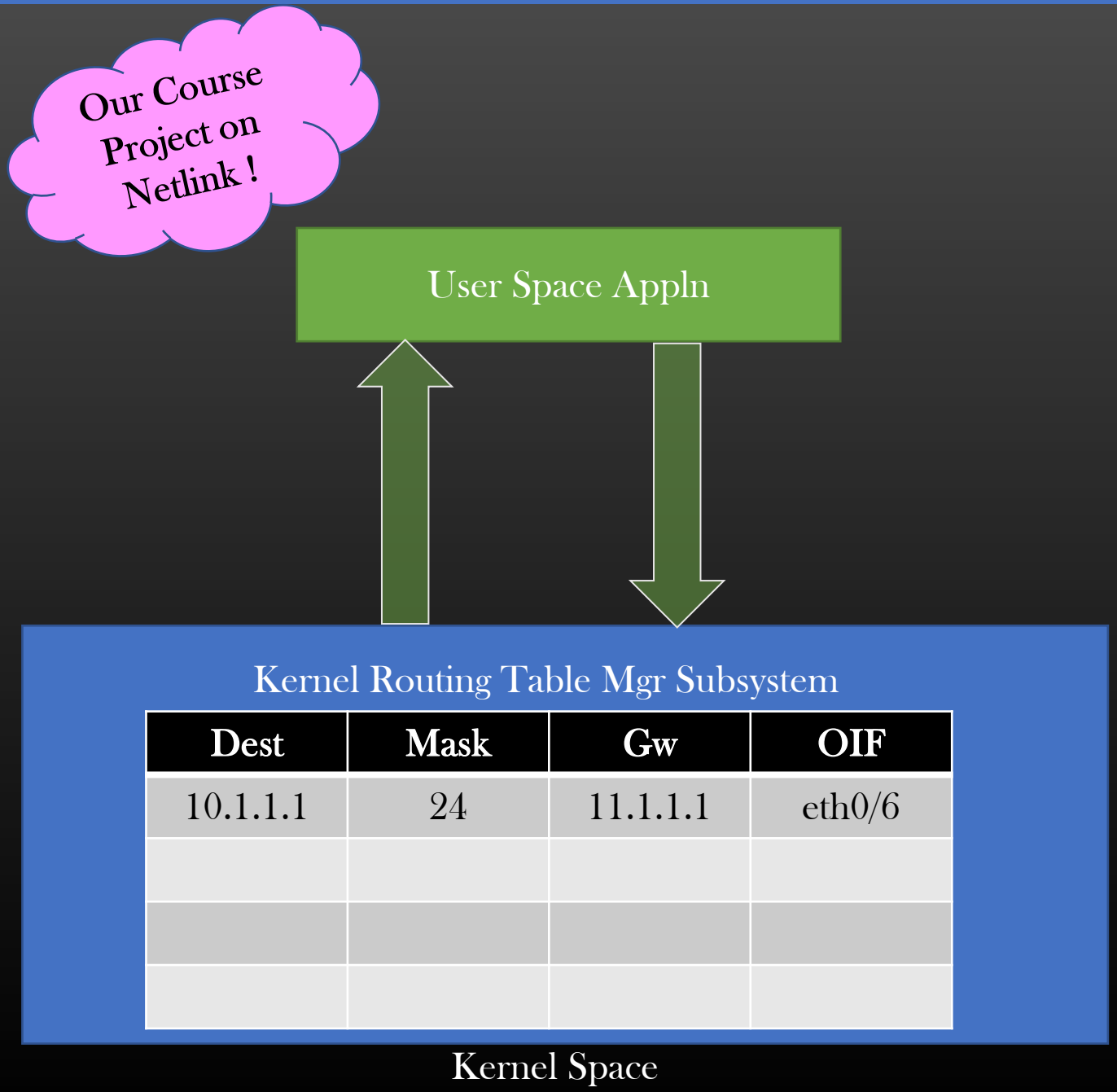

App1
Unix Domain Sockets
App2
...
AppN
Network (Network Sockets)
Socket Interface (bunch of System Calls)
(socket , accept, bind, send, recv, close etc)
Netlink Sockets
Kernel Subsystems
Kernel Subsystems
Kernel Subsystems

Linux Kernel Programming -> The Netlink Sockets UseCases



Linux Kernel Programming -> Other Netlink UseCases

- Configure Run time Linux Kernel Configuration parameters from User-space
- Configure kernel Routing Sub-system
  - L3 Routing Table, ARP, Fire-walls, IP-Tables etc
- Fetch or configure NIC (Network Interface Cards) information such as MAC, IP addresses, MTU etc
- Extracting Memory usage, Process related information



- In this project, we shall be discussing the communication between USA and Routing Table Mgr Kernel Subsystem as an example
- USA Actions :
  - Perform CRUD Operations
- KS Actions
  - We will write LKM which will behave as Routing Table Mgr residing in kernel space.
  - It shall process CRUD orders coming from application
- Same Netlink Communication semantics applies to other kernel subsystems

- In this Course, We shall explore Netlink Socket Based Communication between US & KS by developing USA which interact with our LKM which is in-charge of our *Routing Subsystem of Kernel*

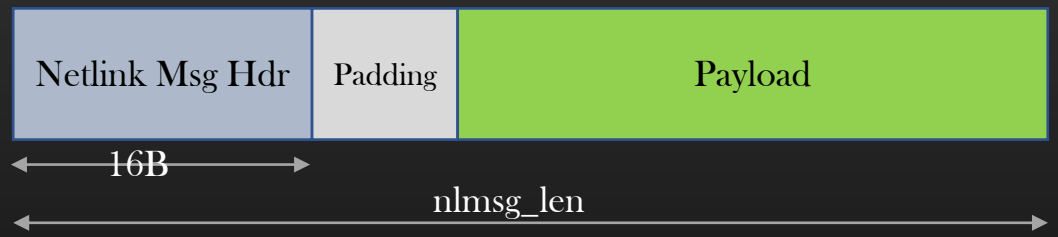
- Once we understand how Netlink Based communication work using one sub-system as example, we can use our knowledge to develop USA which can interact with any kernel subsystems – The principles and rules of communications are same, what changes are only type of Netlink Messages exchanged

- Along the way, we shall also learn TLV based communication, event-based notifications

- Time to write our first Netlink Program ... ! ☺

➢ Netlink Msg Format

- USA and KS exchange Netlink msgs in well defined format
- Any Netlink msg going from USA to KS or from KS to USA must be as per Netlink standard msg format
- A Typical Netlink Msg is laid out in Memory as below :

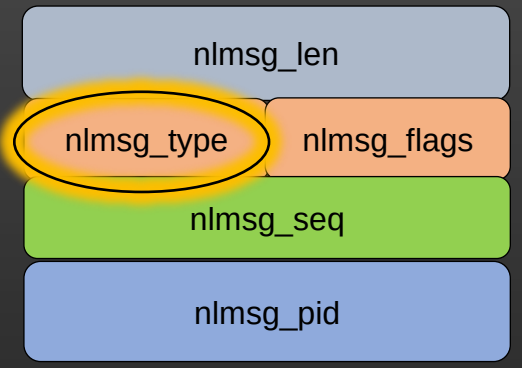



```
struct nlmsghdr {
        u32 nlmsg_len;      /*Total length of msg = nlhdr + padding + payload*/
        u16 nlmsg_type;
        u16 nlmsg_flags;
        u32 nlmsg_seq;
        u32 nlmsg_pid;
};
```

➢ Netlink Msg Format

- Both parties, can exchange Multiple Netlink msg units cascaded one after the other

➢ Netlink Msg Hdr -> Netlink Msg Types



nlmsg_type

4 standard types are defined in /usr/include/linux/netlink.h

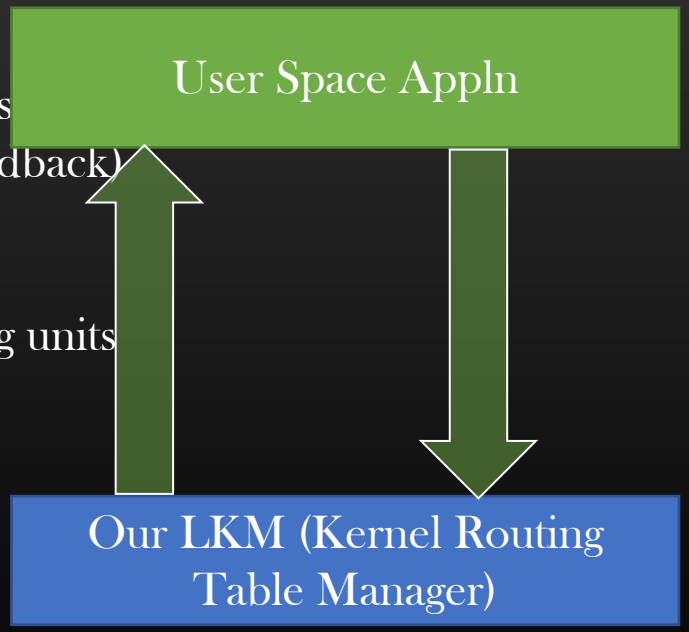
NLMSG_NOOP : When the other party recvs this msg, it does nothing except it replies with NLMSG_DONE telling the sender that all is fine (= Is all ok ?)

NLMSG_ERROR : When the party recvs this msg as a reply to the msg sent previous it means that other party failed to perform requested action (= negative feedback)

NLMSG_DONE : This is the last Netlink msg in the cascade of multiple Netlink msg units

NLMSG_OVERRUN : Currently not used in linux kernel anywhere

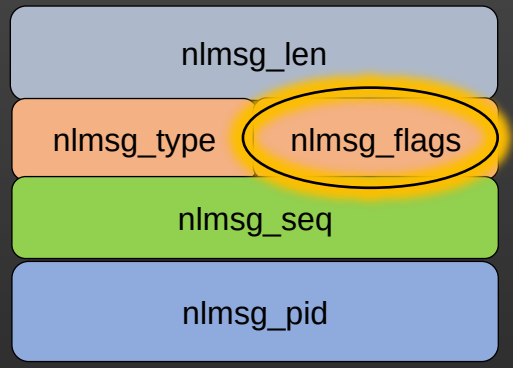
Note : Besides above, User can define his own msg types which should be >= 16



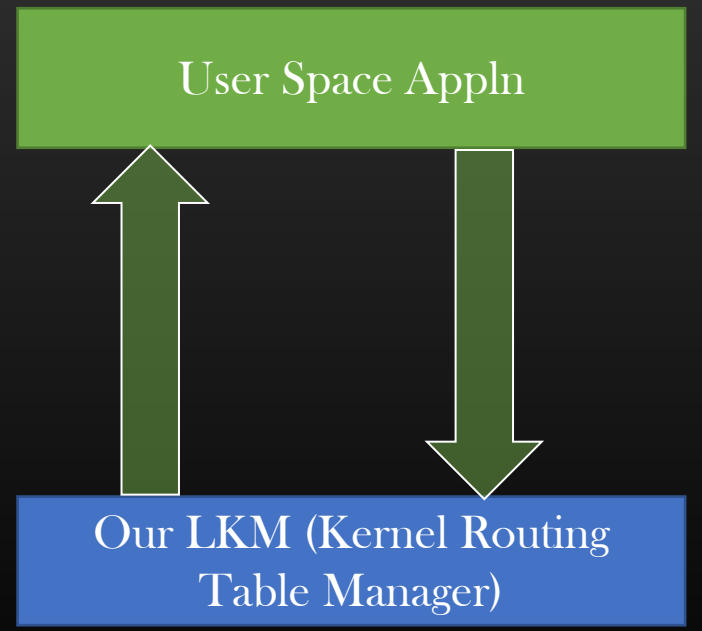
➢ Netlink Msg Hdr -> Netlink Msg flags

nlmsg_flags

> These flags are set in Netlink msg to convey additional information to the recipient
> Multiple flags could be set using bitwise AND/OR operators

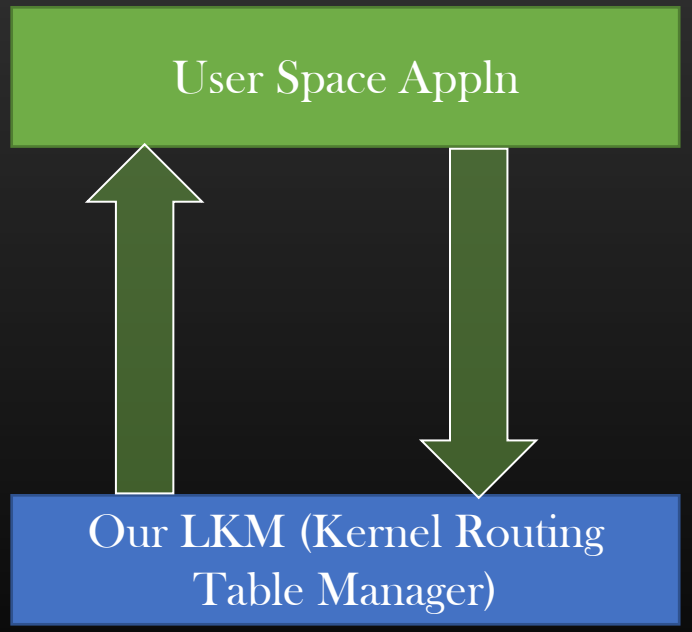


flags	Use
NLM_F_REQUEST	The Netlink msg contains a request. Should be set for each Netlink Msg going from USA to KS, if not kernel replies back with invalid argument EINVAL error. This US as Master, and kernel as Slave.
NLM_F_CREATE	USA asking kernel subsystem to create a resource or configuration
NLM_F_EXCL	Used together with NLM_F_CREATE, USA asking kernel to return an error with if the configuration/resource already exists
NLM_F_REPLACE	USA wants to replace an existing configuration in the kernel-space subsystem
NLM_F_APPEND	USA application requesting KS to add more data to existing configuration, for example adding some data to existing linked list
NLM_F_DUMP	USA requesting KS to send itself all the data of particular type. KS replies with multipart cascaded Netlink msgs to such request from USA
NLM_F_MULTI	This flag is set to tell the recipient that there is NEXT Netlink msg following to this one !
NLM_F_ACK	If set, USA is requesting the KS to reply back with the confirmation msg of the USA's request. KS replies with NLMSG_NOOP Or NLMSG_ERROR msg type

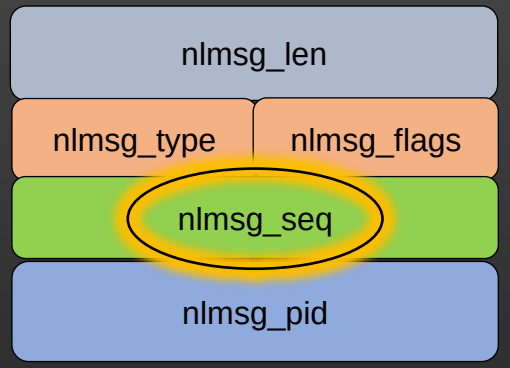


➢ Netlink Communication Model

- From Netlink Flags, you should get an idea that Netlink based communication :
  - USA application is generally the requester – the master who is placing order
  - Kernel is generally the request Entertainer – the slave who acts on application's order/request
  - Most of the time, it is USA which initiate the communication with KS
  - In case of event-based notification, it is kernel which initiate the communication



➢ Netlink Msg Hdr -> Sequence Number

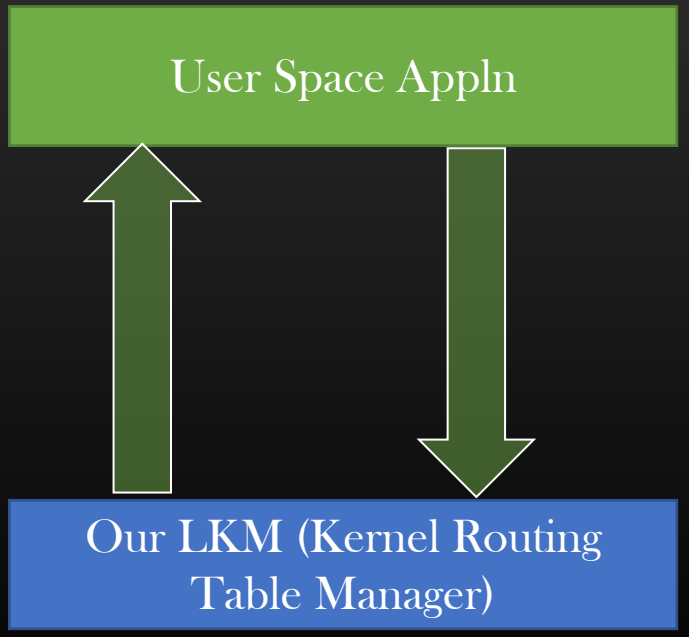


Sequence number

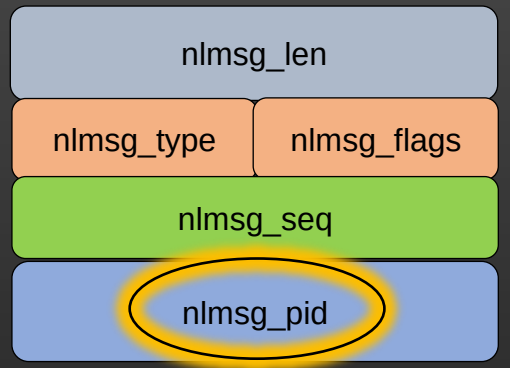
- When USA sends a Netlink request msg to KS, it must set a unique number to this request if USA sets NLM_F_ACK flag

- When KS replies back with confirmation msg to USA, it sets the same sequence no which was specified in the request msg sent from USA

- This helps the USA to correlate which Netlink reply is for which Netlink request in case USA has issues multiple Netlink Requests to Kernel and awaiting reply

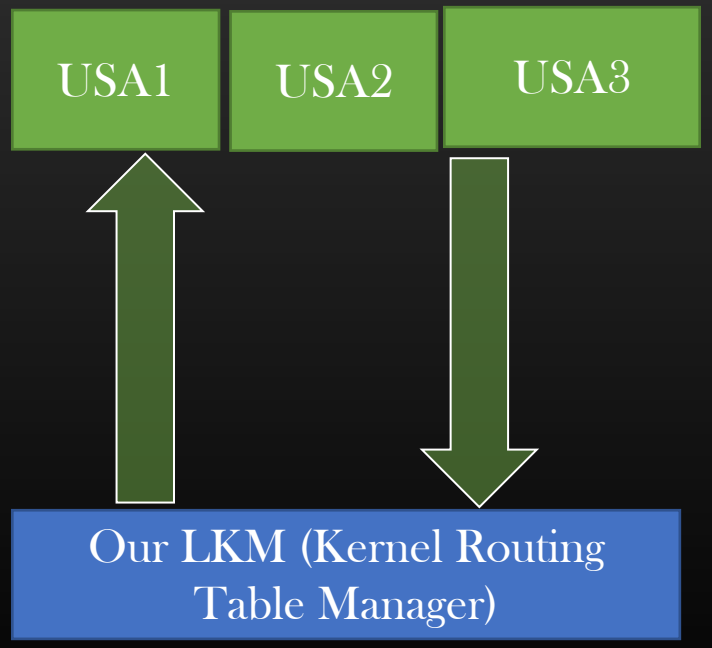


➢ Netlink Msg Hdr -> Port Id



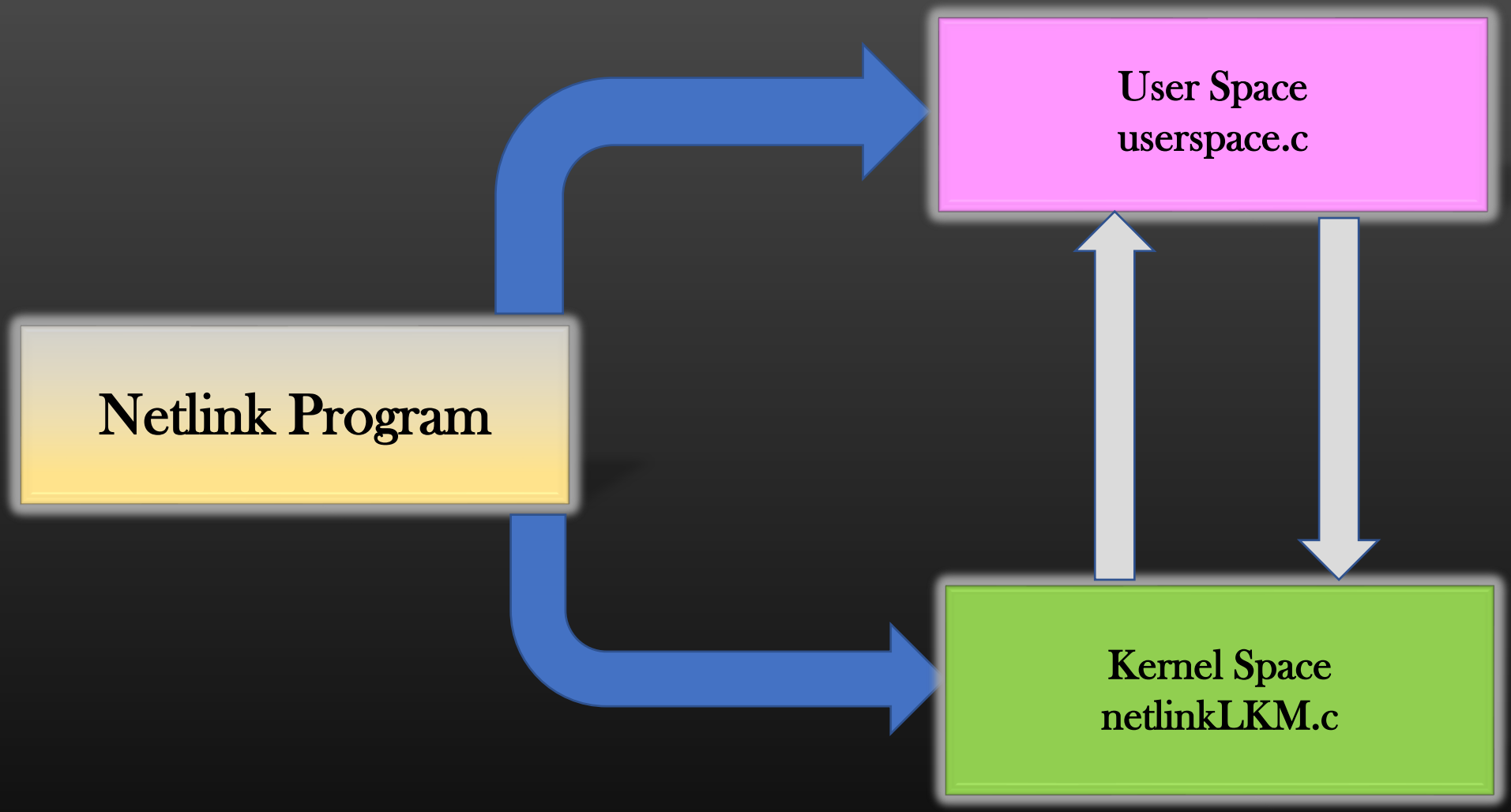
Port ID

- Set by the USA while sending Netlink msg to KS
- It must be unique to the USA, therefore good practice to use process id
- Kernel Use this info to reply back to the correct application in user space
- This value is set to zero for Netlink msgs Originating from KS to USA

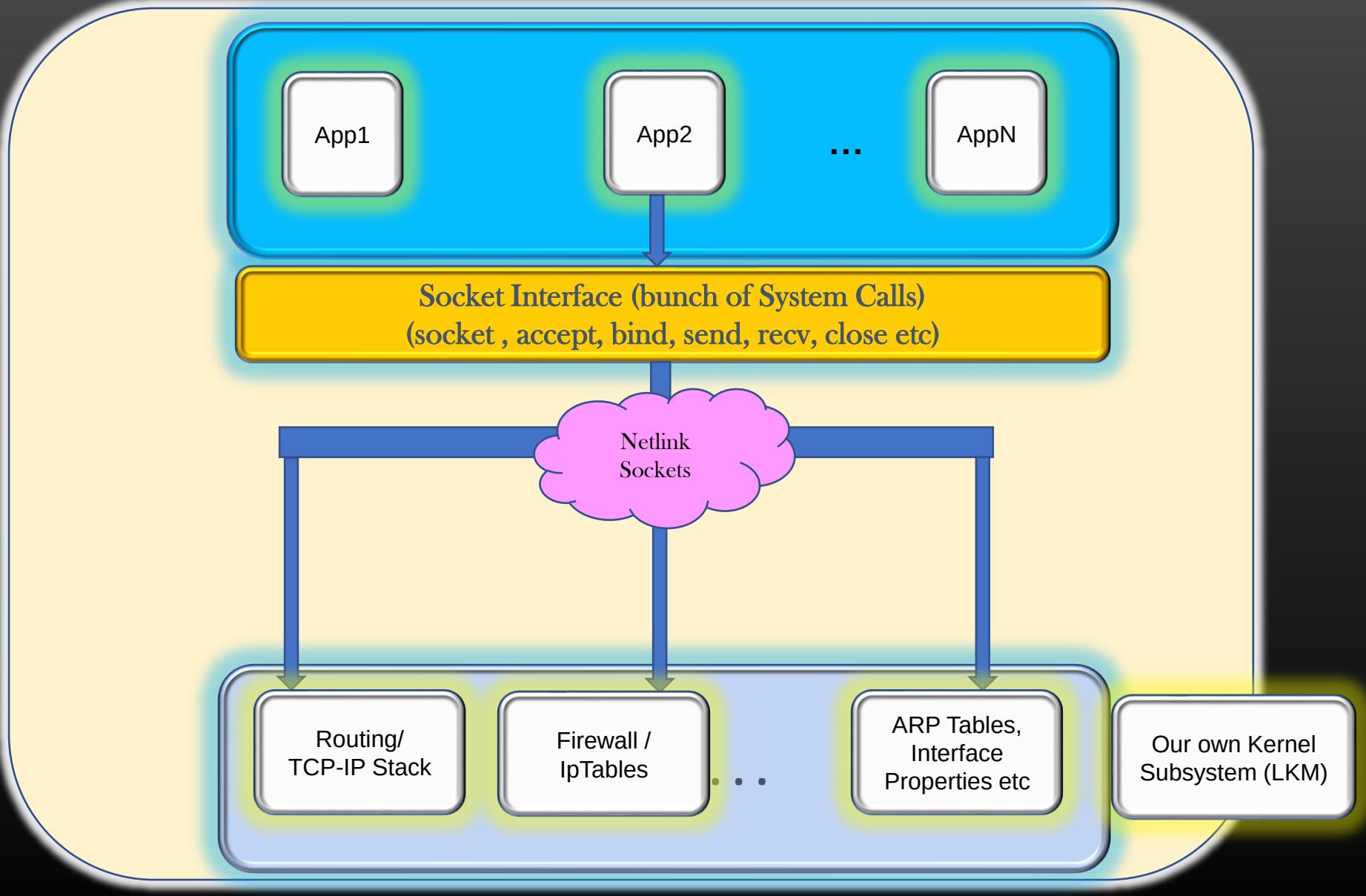


Netlink Hello World

- We shall write a Netlink Hello World User Space and Kernel LKM to demonstrate the Netlink Socket based communication
- We shall design our USA which forks a separate thread to receive data, and send data to KS in main thread only
- We will still use same typical steps involved in socket programming :
  - Creating socket
  - specifying src and dst address
  - bind()
  - sendmsg and recvmsg
  - Close
- I shall explain the new APIs that we shall encounter right in the code walk
- In the next Section, we shall begin a new project and incrementally add new code as we learn more new concepts , So Let us start with building our first very basic Netlink program . . .



- Compile separately
- We shall write a GREET Example, in which USA and LKM simply exchange “Hello How are you !” kind of message
- Once we setup the basic communication infrastructure in place, we shall begin our Netlink Project



- Netlink Protocol Number

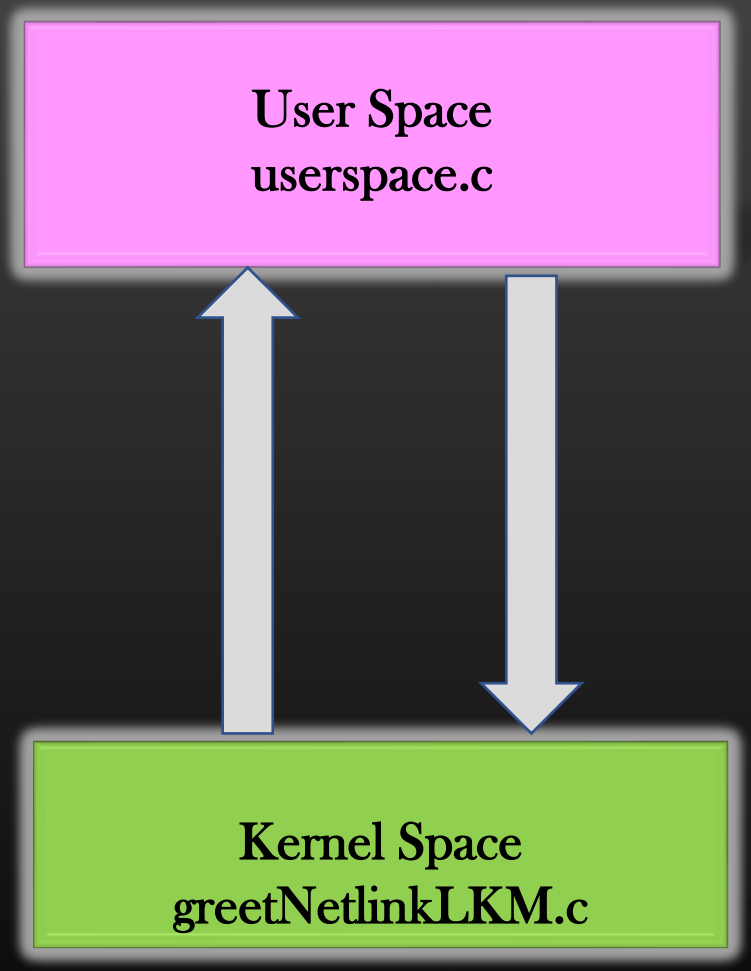
- A unique ID called Netlink Protocol number is assigned to Each Netlink capable kernel Subsystem

- For example, see linux/netlink.h

☜ We need to choose a Unused unreserved Netlink Protocol number

- Getting Started with Writing our Netlink LKM
  - Steps :
    - Register init and cleanup functions
    - Initialization struct netlink_kernel_cfg
    - Netlink Socket Creation
    - Netlink Socket Destruction
    - Receiving User space message
    - Processing User space msg
    - Replying to User space

APIs :



- When Kernel Space receives data from USA via Netlink, data is received in a data structure called socket buffer

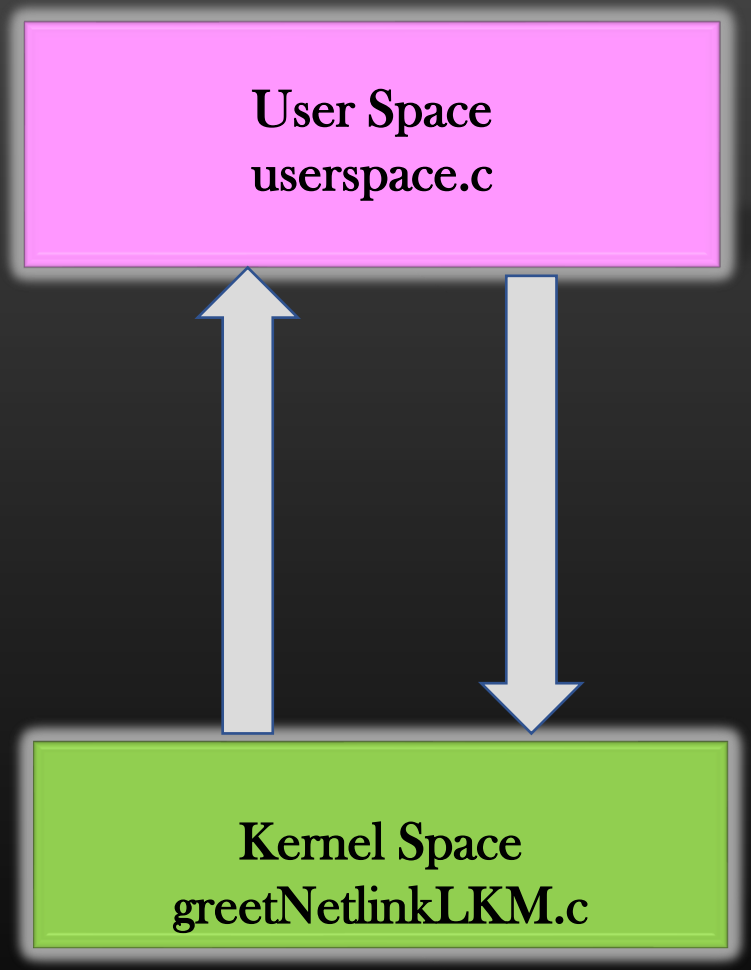
  struct sk_buff; defined in include/linux/skbuff.h

- Kernel Uses this data structure extensively for multiple purposes:
  - For transferring messages from one kernel Subsystem to another
  - For receiving Network Packet
  - Packet movement upwards and downwards in the layers of TCP/IP Stack (Linux Implementation of OSI Model)
  - Etc

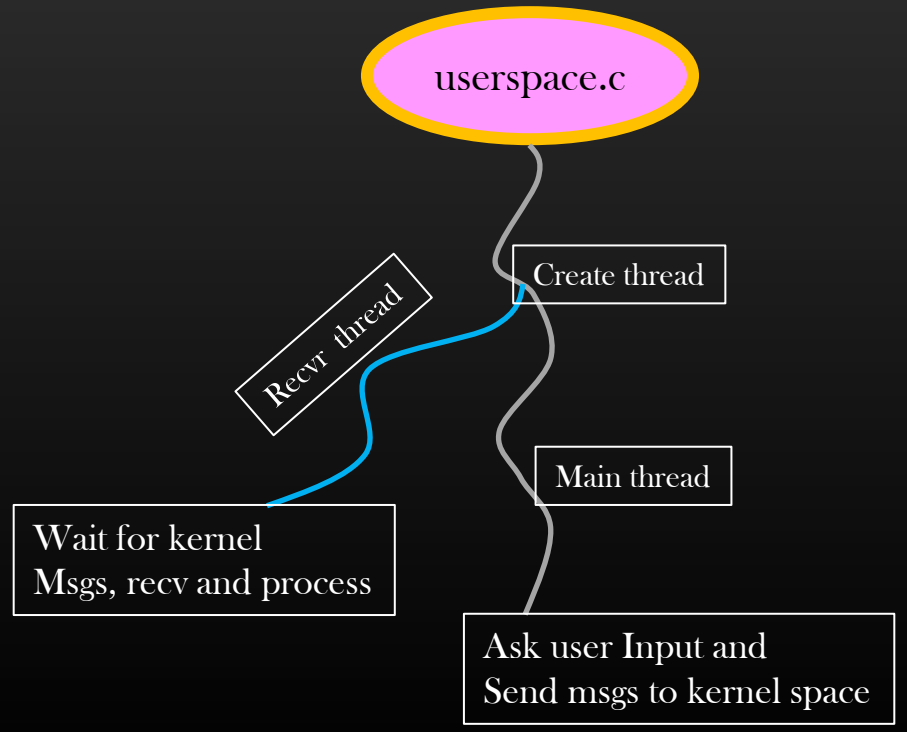
- This is large data structure; we shall be discussing it only in the context of Netlink Socket Communication

- struct sk_buff *skb_in;
  User space data is received in skb_in->data;
  length of data : skb_in->len

Writing Netlink Userspace Program



- Now we shall begin writing Userspace Netlink program
- This Userspace program will do the following :
  - Create a Userspace Netlink Socket
  - Start a separate Netlink msg receiver thread
  - Ask the user for input to send GREET msg to kernel
  - Send GREET Netlink msg to kernel LKM using protocol no 31
  - recv kernel reply GREET_REPLY msg via receiver thread



API to send Msg to LKM from Userspace

```
int
send_netlink_msg_to_kernel (int sock_fd,     /*Netlink Socket FD*/
                            char *nl_payload,                    /*msg to be se
                            uint32_t msg_size,                 /*msg size in Bytes*/
                            int nlmsg_type                       /*msg type = G
                            uint16_t flags);                     /*NLM_F_*
```

Step 1 : Prepare *struct nlmsghdr* with payload msg to be sent

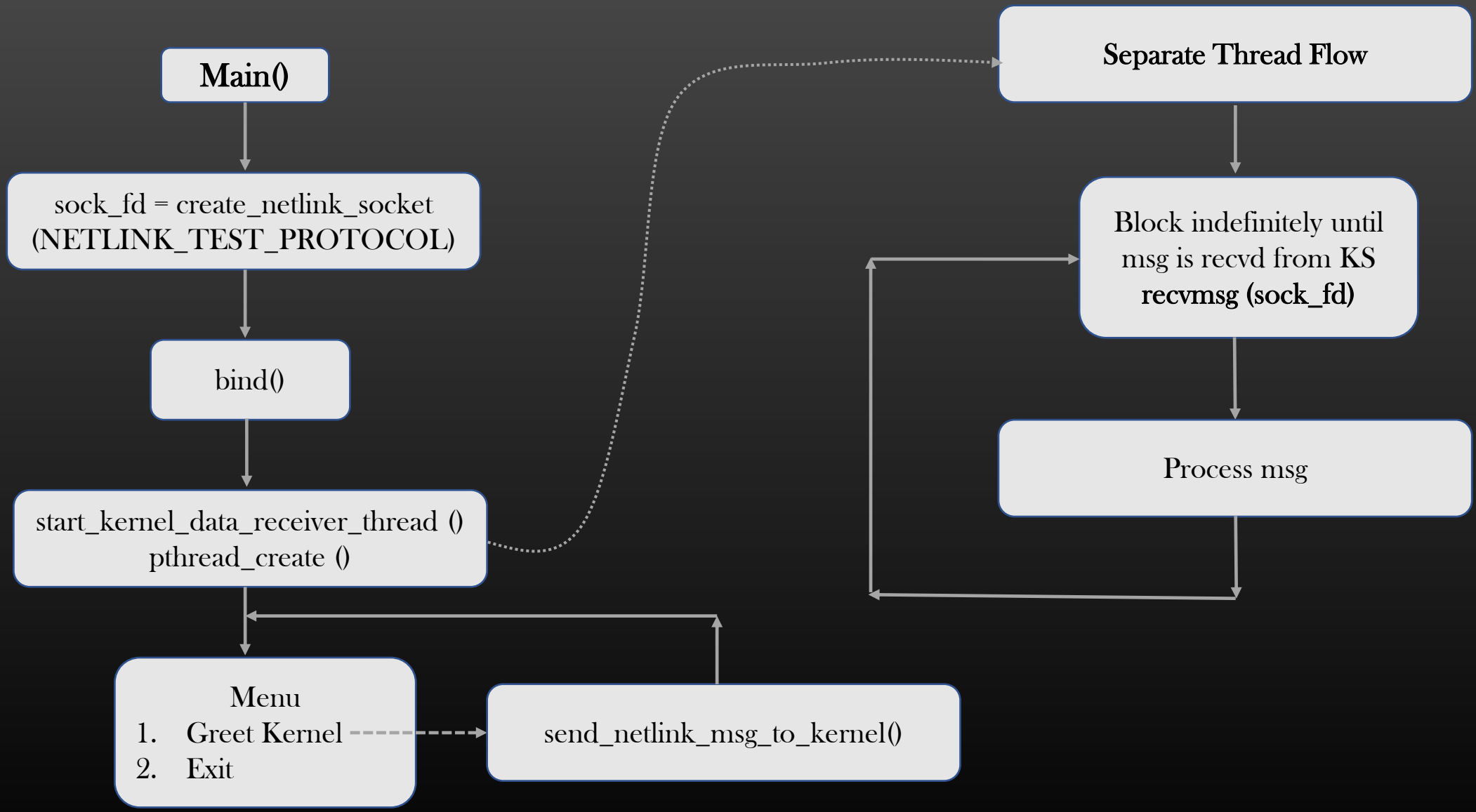
Step 2 : Wrap the message inside *struct iovec iov*


```
struct sockaddr_nl dst_addr;

dest_addr.nl_family = AF_NETLINK;

dest_addr.nl_pid = 0;  /*If kernel is Dest*/
```

Step 3 : Wrap the iovec inside *struct msghdr outermsghdr;*

```
memset (&outermsghdr, 0, sizeof(struct msghdr));
outermsghdr.msg_name = (void *)&dest_addr;   /*Whom you are sending this msg to*/
outermsghdr.msg_namelen = sizeof(dest_addr);
outermsghdr.msg_iov = &iov;
outermsghdr.msg_iovlen = 1;
```

Step 4 : sendmsg (sock_fd, &outermsghdr, 0);


Main()
sock_fd = create_netlink_socket (NETLINK_TEST_PROTOCOL)
bind()
start_kernel_data_receiver_thread () pthread_create ()
Menu
1. Greet Kernel
2. Exit
send_netlink_msg_to_kernel()
Separate Thread Flow
Block indefinitely until msg is recvd from KS recvmsg (sock_fd)
Process msg

Remaining Section of The course ...

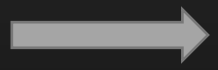
Netlink Attributes - The concept of TLVs

The Netlink Project

Multicast with Netlink Sockets

The Concept of TLVs

- TLV – Type Length Value
- TLV is a mechanism of packaging the data in a TYPE LENGTH VALUE order
- Benefits :
  - Flexible, easy to add and remove data
  - Not tied to pre-defined Structure
  - Ignore the Data which is not recognized
  - Process the data which is recognized
- If you are already familiar with TLV, Jump to assignment section straightaway
- Example :
- Data which shows 3 TLVs are packed together

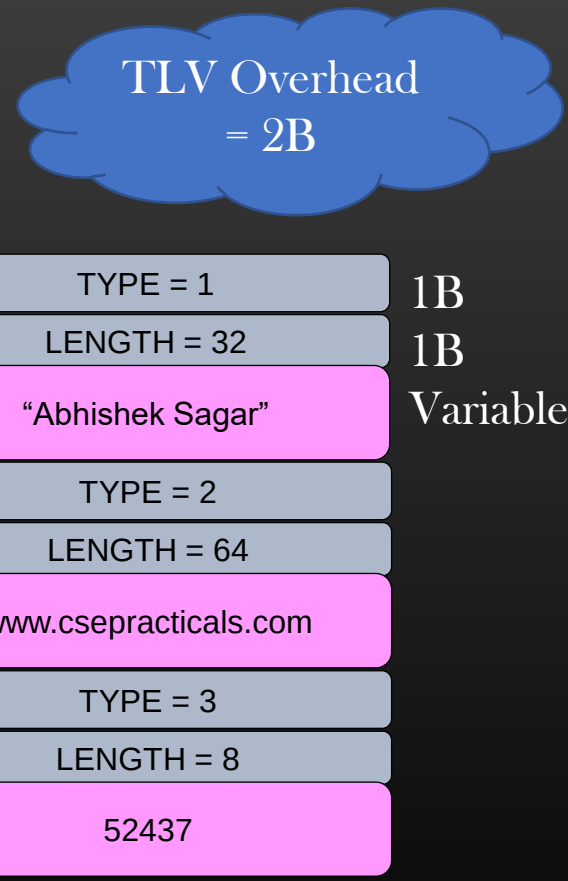


```
#define NAME_TLV_CODE          1  >> String format , Unit Data length 32B
#define WEBSITE_TLV_CODE       2  >> String format,  Unit Data length 64B
#define EMP_ID_TLV_CODE        3  >> Integer format, Unit Data length 8B
```

TLV Code point gives two information :

> Data type of Value (String, integer , float, ip-address . . .)

> Length of 1 Unit of Data



The Concept of TLVs

- Parsing TLV Buffer

```
#define NAME_TLV_CODE          1  >> String format , length 32B
#define WEBSITE_TLV_CODE       2  >> String format, length 64B
#define EMP_ID_TLV_CODE        3  >> Integer format, length 8B

char *tlv_buffer = <pointer to the start of TLV buffer>
uint32_t total_size = 222B

uint8_t type = *tlv_buffer;
uint8_t len =  *(tlv_buffer + 1);
int units = len/get_unit_data(type);
char *name = (tlv_buffer + 2);
for ( int i = 0; i < units; i++){
        char *name = (tlv_buffer + i * get_unit_data(type));
        printf ( "Name = %s\n", name);
}
tlv_buffer += len;
tlv_buffer += TLV_OVERHEAD;
/*Read next TLV and go on*/
```


```
int
get_unit_data_size(uint8_t tlv_type )
{
 switch(tlv_type){
  case NAME_CODE_TLV:
            return 32;
   case WEBSITE_TLV_CODE:
            return 64;
   case EMP_ID_TLV_CODE:
            return 8;
   default:
            return 0;
}
}
```

Caution : Take care that you don't overshoot TLV buffer, track how much buffer you have scanned

The Concept of TLVs

Assignment : Write a TLV Iterative macros, and make your life easy

```
#define ITERATE_TLV_BEGIN(start_ptr, type, length, tlv_ptr, buffer_size)
#define ITERATE_TLV_END(start_ptr, type, length, tlv_ptr, buffer_size)
```

How to use :

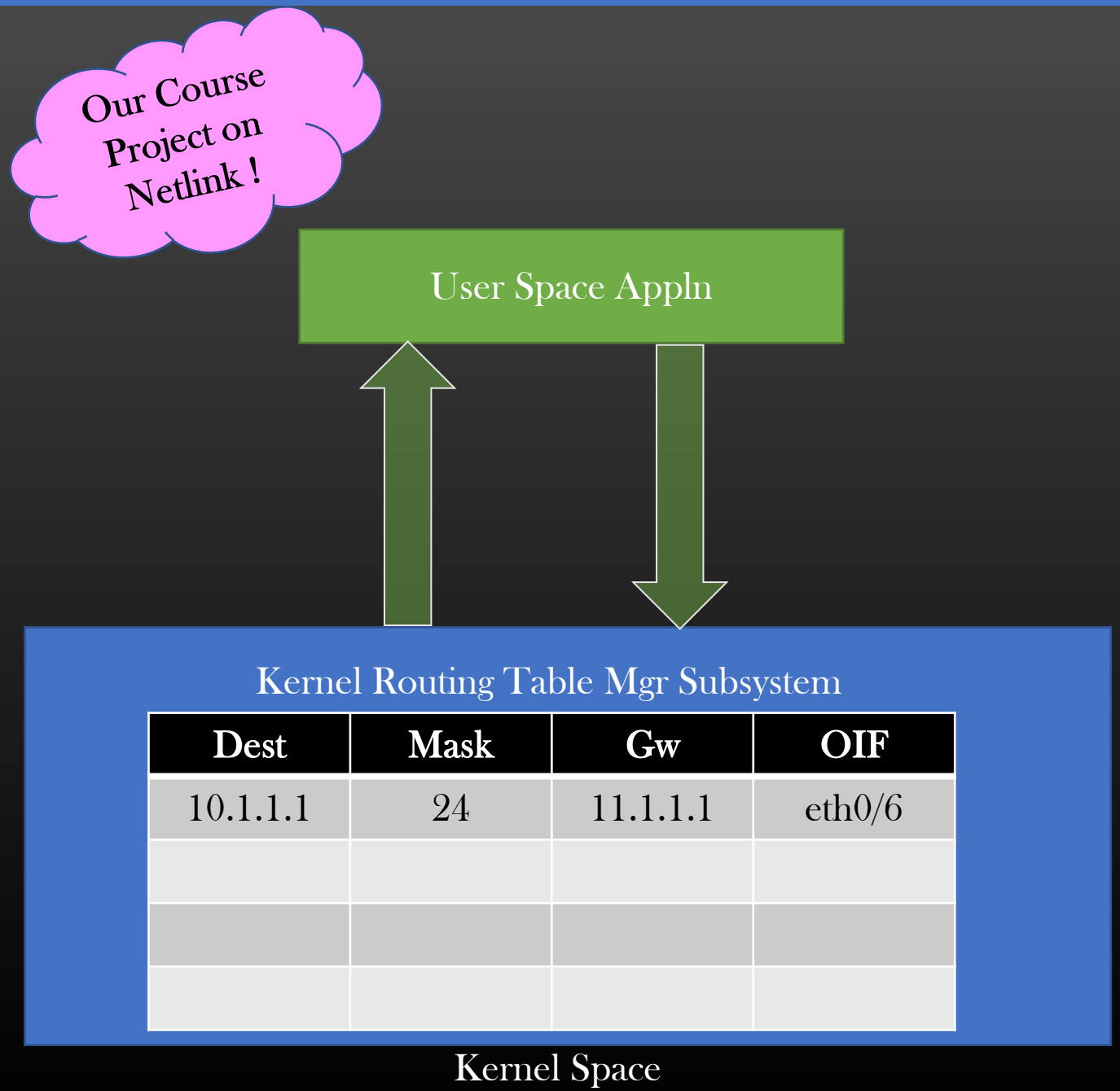
```
char *tlv_buffer = <pointer to the start of TLV buffer>
uint32_t total_size = 222

uint8_t type, uint8_t len;
char *val;
ITERATE_TLV_BEGIN( tlv_buffer , type, len, val, total_size){
        switch(type){
                case NAME_TLV_CODE:
                        process 'val';
                        break;
                case WEBSITE_TLV_CODE:
                        process 'val';
                        break;
                . . .
                default: /*Do nothing */
        }
} ITERATE_TLV_END(start_ptr, type, length, tlv_ptr, buffer_size)
```

Macro :
Read all TLVs in a buffer Sequentially

Gives the T L V for each TLV in buffer to programmer For processing

Take care to not overrun TLV buffer

Programmer don't have to Worry about adjusting and Jumping pointers with-in a TLV buffer



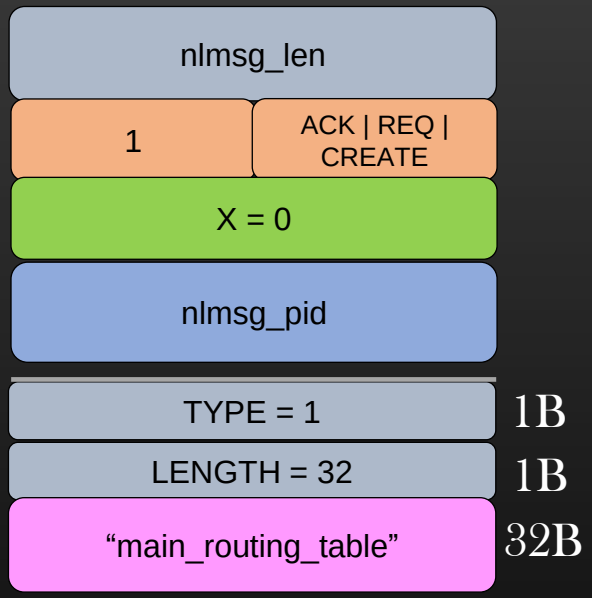
- We will create a Netlink User space and Kernel space LKM as a project
- We shall Implement a Routing Table Manager, the routing table resides in kernel space
- User space Program send instruction to RTM LKM, RTM in turn perform actions on Routing Table
- Lets kick start the project straightaway, We shall add more complexity to it as We progress
- us_rtm.c – user space program
- ks_rtm_lkm.c – kernel module

- Operation 1 Specification
  - Synopsis : Creating a New Routing Table in KS

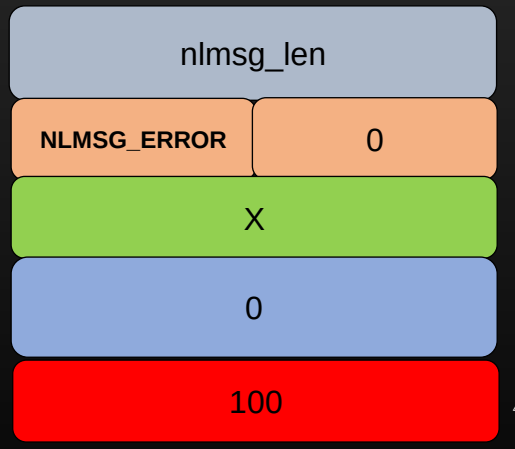
- Functionality :
  - User Space Instructs the KS LKM to create a new Routing Table with Name *name*

- Netlink Msg US -> KS
  - Nlmsg_type = NLMSG_RT_NEW_CREATE = 21
  - Flags = NLM_F_ACK | NLM_F_REQ | NLM_F_CREATE
  - Attribute :
    - TYPE : 1
    - LENGTH : 32B
    - Value : *String name*

US -> KS Instr.



ACK MSG



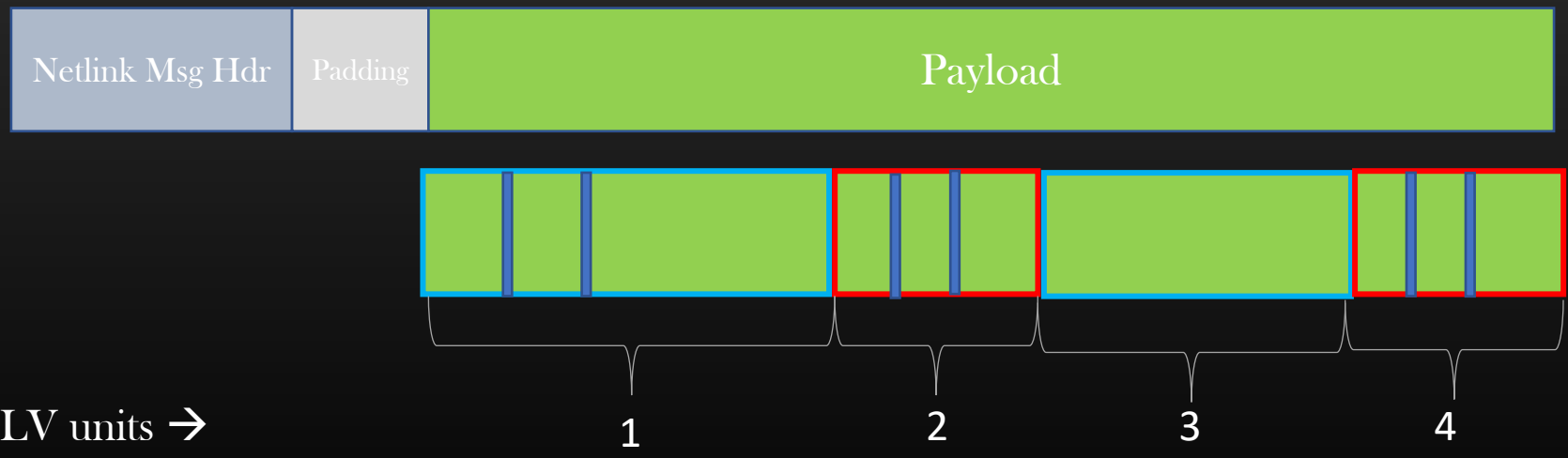
ERROR MSG

- KS Action
  - If msg recvd of type NLMSG_RT_NEW_CREATE
    - If NLM_F_REQ flag not set, ignore the message silently
    - If Table do not exist && NLM_F_CREATE flag set then create a new table
      - If NLM_F_ACK flag is set, send ACK to US otherwise stay silent
    - If Table already exists, Send Error Code -100 to US

- US on Receiving an ACK or ERROR code, notify the user by printing
  Success
  Failure, error code -100

Netlink Attribute

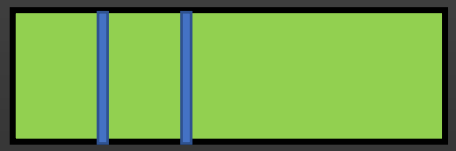
- We have setup a basic Netlink Socket Based Communication between our USA and KS
- Netlink protocol require the communication parties to make use of TLV format in order to exchange data
- These TLVs are appended after the Netlink msg header



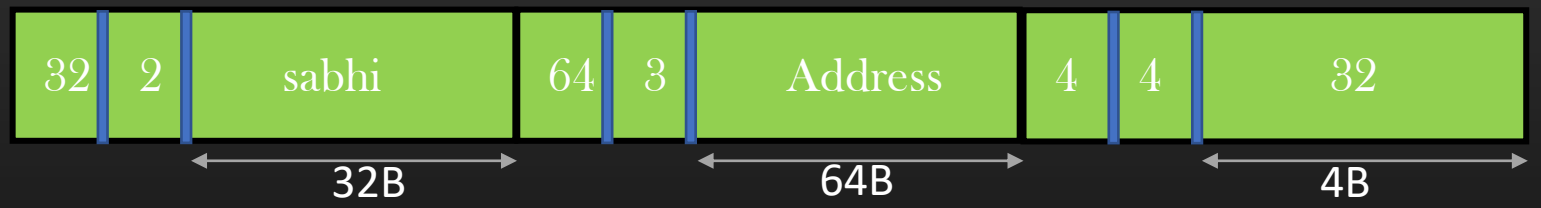
4 TLV units →
Variable Sizes
Each TLV unit has three parts : Type (2B) , Length (2B), Value (Variable data)

➢ Netlink Attribute

```
struct nlattr {
    __u16           nla_len;
    __u16           nla_type;
};
```



Example :



The recipient need to be programmed to how to interpret the TLV Codes, else skip the unknown TLV Code point

TLV Code	Meaning
2	Name, so use char [32]
3	Address, so Use char[64]
4	Age of the person, so Use int

➢ Netlink Attribute

➢ Thus, USA and KS exchange Netlink Data strictly in the below format :

Linux Kernel APIs Provides us various macros to work with Netlink Headers, and TLVs

We shall discuss those macros directly in Programs !