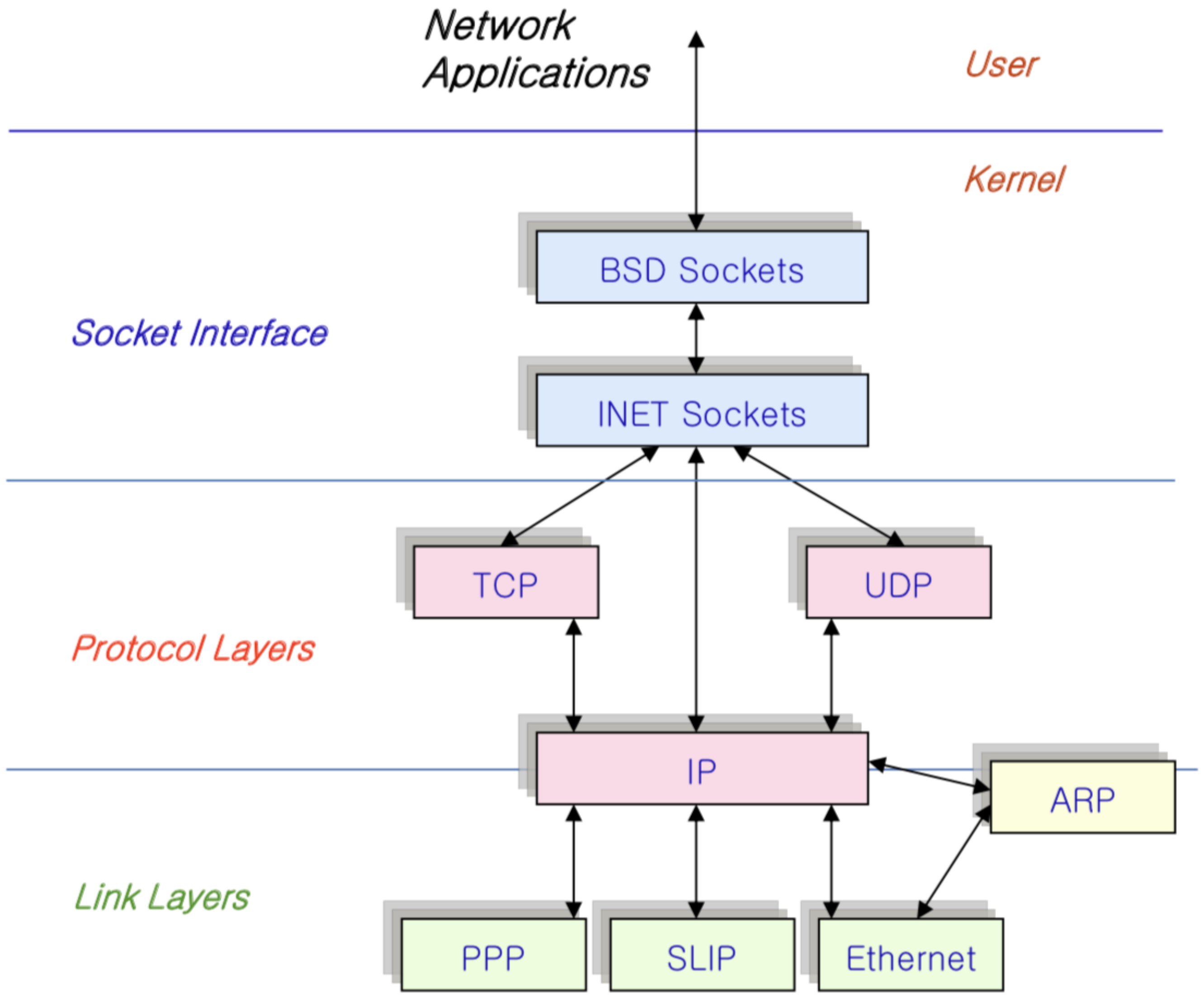
Linux Kernel & Device Drivers

Chapter 7 - Network Device Drivers

»log in to knowledge

The Linux Network Protocol Stack – TCP/IP



Sockets

- The `socket` is a dual layer entity.
- The upper layer implements the network related system calls (*e.g.* `socket`, `bind`, `connect`, etc.).
- The bottom layer is protocol family dependent (*e.g.* `AF_INET`, `AF_UNIX`, etc).
- For each protocol in a protocol family, the bottom layer provides a set of protocol specific backends of the network related system calls.

Sockets

- Each protocol family implements a `create` function. The role of this function is to associate the socket the the appropriate bottom layer.
- Kernel modules that implement protocol families use `sock_register` to register a `net_proto_family` structure with the protocol families table.
- When a new socket is created, the *`domain`* argument of the `socket` system call is used to select the `create` function suitable for the protocol family.

Sockets

- The `create` function infers from the *`type`* and *`protocol`* arguments of the `socket` system call the exact protocol that the socket should use.
- `create` register a `proto_ops` structure with the socket. This structure provides a set of call back functions that implement protocol specific handling of network related system calls.
- `create` also initializes protocol specific fields of the `socket` structure.

proto_ops

```
struct proto_ops {
  int              family;
  struct module    *owner;
  int              (*release)   (struct socket *sock);
  int              (*bind)      (struct socket *sock,
                                 struct sockaddr *myaddr,
                                 int sockaddr_len);
  int              (*connect)   (struct socket *sock,
                                 struct sockaddr *vaddr,
                                 int sockaddr_len, int flags);
  int              (*socketpair)(struct socket *sock1,
                                 struct socket *sock2);
  int              (*accept)    (struct socket *sock,
                                 struct socket *newsock,
                                 int flags, bool kern);
  int              (*getname)   (struct socket *sock,
                                 struct sockaddr *addr,
                                 int peer);
  __poll_t         (*poll)      (struct file *file, struct
socket *sock,
                                        struct
poll_table struct *wait);
```

proto_ops

```
  __poll_t          (*poll)       (struct file *file,
                                   struct socket *sock,
                                   struct poll_table_struct
*wait);
  int               (*ioctl)      (struct socket *sock,
                                   unsigned int cmd,
                                   unsigned long arg);
   .    .    .    .
};
```

Handling of Protocols

- The protocol operations that handle sending and receipt of data create and remove the protocol envelopes.

- Transport layer protocols (*e.g.* TCP & UDP) communicate with their respective network layer protocols (*e.g.* IP), by calling callbacks defined by the **`proto_ops`** structure of the network layer.

- Network layer protocols communicate with link layer interfaces by propagation **`of sk_buff`**’s.

Handling the Link Layer

- The handling of link layer depends on the underlaying hardware (*e.g.* Ethernet, serial port, etc.).

- The link layer adds / removes link layer envelopes.

- The link layer enqueues / dequeues **`sk_buff`**'s to / from their respective device queues.

Netfilter Hooks

- Enqueueing and dequeuing of `sk_buff`'s on / from specific devices is handled by callback functions assigned as netfilter hooks.
- A netfilter hook is an ordered list of functions that operate on `sk_buff`'s.
- Netfilter hooks allow additional packet processing such as tunneling, bridging, firewalls, etc.

Network Devices

- Network devices resemble character devices, in terms of data transfers.
  - Ethernet NICs packet size is 64-1524 bytes.
  - Modern Gigabit ethernet hardware support a payload of up to 9000 bytes (jumbo frames)†.
- Yet, the data is never raw, it is enveloped within protocol headers and footers.

†Jumbo frames are supported by linux starting at kernel 2.6.17

Network Device Drivers

- Network device drivers have no corresponding special device files.

- Network device drivers are accessed via the socket API.

- Network device drivers implement entry points that are suitable for communicating with network protocols.

The net_device Structure

- `net_device` is the structure that stores network device information.

- The structure is defined in `<linux/netdevice.h>`, and stores a mixture of:
  - Low level information that corresponds to the network device.
  - High level information that corresponds to network protocols that can utilize the device

net_device Fields of Interest

- `dev` - Associates the device with a `sysfs` entry (`struct device`).
- `name` - Used in conjunction with device `ioctl`'s, and within `sysfs` (`/sys/class/net/`*name*).
- `netdev_ops` - a structure of pointers to callback functions that the rest of the kernel uses to access the driver's entry points. This field must be assigned by the driver during driver initialization.

net_device Fields of Interest

- **ethtool_ops** - a structure of callback functions that are used by **ethtool**†.
- **watchdog_timeo** - the watchdog timer for handling hung transmittals (clock ticks).
- **features** - a bit mask of network features that are currently active (vlan, scatter/gather,...).
  See **<linux/netdev_features.h>** for full list, and description.

†ethtool is a utility for querying and controlling network drivers and hardware settings.

net_device Fields of Interest

- `perm_addr` - The **MAC** address that is burnt of the device's eeprom.

- `dev_addr` - The **MAC** address to use. Defaults to `perm_addr`, and may be overridden by an `ioctl` command.

- There are many more fields in the `net_device` structure, that relate to specific device types, specific network protocols, link layers, etc..

alloc_netdev

```
#include <linux/netdevice.h>

struct net_device *alloc_netdev(
               int            sizeof_priv,
               const char    *name,
               void          (*setup)(struct net_device *));
```

- `alloc_netdev` is a generic function that allocates a `net_device` structure.
- The `name` field of the allocated structure is set to `name`.
- Any other fields of the allocated structure may be initialized by the `setup` callback function.

alloc_netdev

- **`alloc_netdev`** allocates **`sizeof_priv`** bytes to store driver private data.

- **`alloc_netdev`** returns `0`, if it fails.

Registering A Network Device alloc_etherdev

```
#include <linux/etherdevice.h>

struct net_device *alloc_etherdev(int sizeof_priv);
```

- `alloc_etherdev` allocates a `net_device` structure, and initializes it as an ethernet device.

- `alloc_etherdev` is a wrapper function that calls `alloc_netdevice`, and uses a built-in `setup` function written to handle ethernet devices.

- The name field is initialized to `eth%d`, where `%d` is replaced by the number of the next ethernet device.

Network Device Driver Private Data

- The **`net_device`** structure has no field dedicated for private data. In fact it is a trailer that follows its end.

- **`alloc_netdevice`** / **`alloc_etherdev`** allocate **`sizeof_priv`** extra bytes at the end of the **`net_device`** structure.

```
#include <linux/netdevice.h>

void *netdev_priv(const struct net_device *dev)
```

- Use **`netdev_priv`** to access the area reserved for the driver's private data.

net_device_ops

- The structure `net_device_ops`, defined in `<linux/netdevice.h>`, holds pointers to the network device driver's entry points.

- There is a total of more than 30 supported network device operations.

net_device_ops

- The list below is the bare minimum of entry points that are required to implement a functional network device driver:
  - **ndo_open** - Called when the device if brought to the "up" state.
  - **ndo_stop** - Called when the device is brought to the "down" state.
  - **ndo_do_ioctl** - Used to implement network device **ioctl** commands.
  - **ndo_start_xmit** - Starts the transmission of a packet.
- The **netdev_ops** field of the **net_device** structure should be assigned with valid **net_device_ops** prior to device registration.

register_netdev

```
#include <linux/netdevice.h>

int register_netdev (struct net_device *netdev);
```

- `register_netdev` registers a network device, after completing the initialization of the `net_device` structure.

- Returns `0` on success, or a negated error code otherwise.

unregister_netdev

```
#include <linux/netdevice.h>

void unregister_netdev (struct net_device *netdev);
```

- **`unregister_netdev`** unregisters a network device that was previously registered.

- It must be called from the module's **`exit`** function, or when device removal has been detected.

free_netdev

```
#include <linux/netdevice.h>

void free_netdev (struct net_device *netdev);
```

- `free_netdev` frees a `net_device` structure.
- `unregister_netdev` must be called prior to calling `free_netdev`.
- Should be called from the module's `exit` function, or upon detecting device removal.

sk_buff

- The `sk_buff` structure is the focal point of network activity in LINUX.
- The `sk_buff` structure is defined in `<linux/skbuff.h>`.
- Each packet has a corresponding `sk_buff` structure .
- First to use allocates an `sk_buff` structure.
- Last to use frees the `sk_buff` structure.

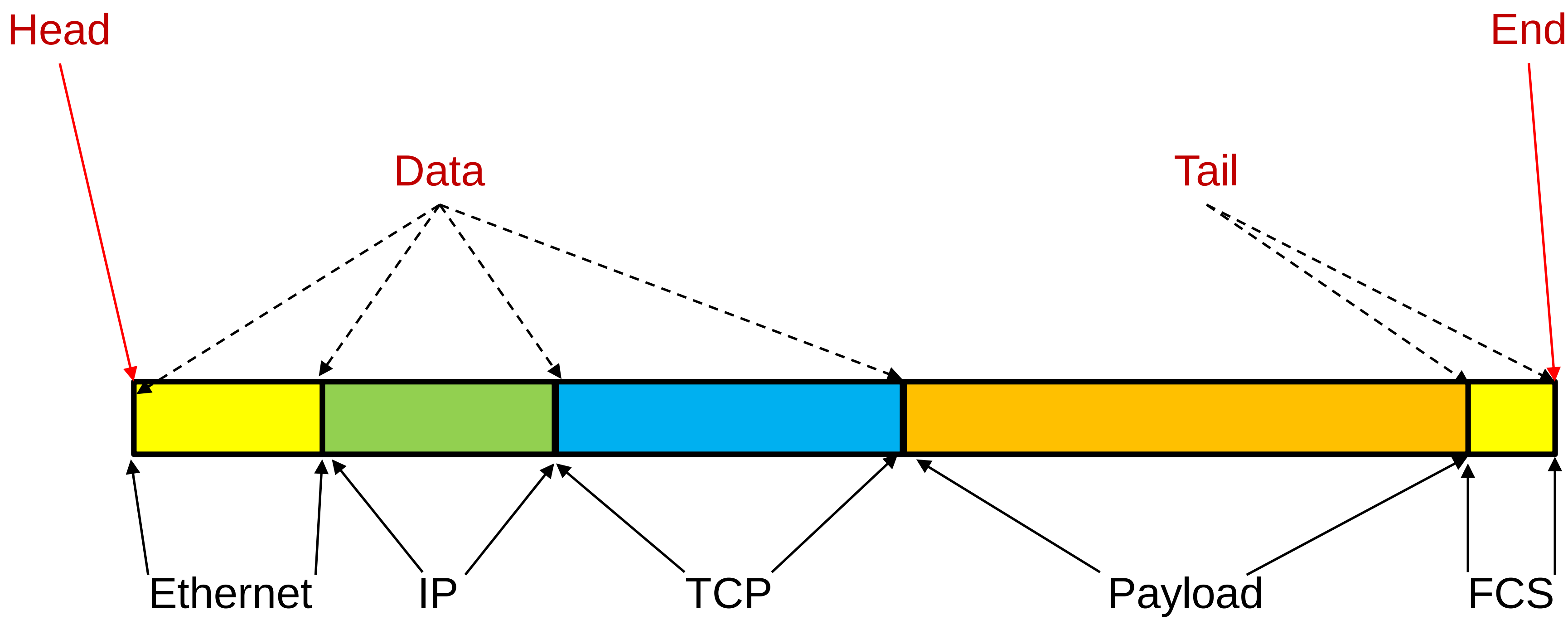
sk_buff Fields

- `next`, `prev` – pointers to the next and previous (respectively) `sk_buff'`s in the list:
  - Upon arrival packets are enqueued on a per-CPU queue.
  - Later stages of processing move the packet to per protocol, and per socket queues.
- `sock` – the socket with which an `sk_buff` is associated.
- `dev` – the `net_device` structure of the device with which this `sk_buff` is associated.

sk_buff Fields

- `head` – Pointer to the beginning of the raw packet data buffer.
- `data` – Pointer to the beginning of meaningful data within the buffer.
- `tail` – Pointer to the end of meaningful data within the buffer
- `end` – Pointer to the end of the buffer.

SK_BUFF Data Pointers & Packet Data



sk_buff Fields

- `transport_header` – A pointer to the location in data where the transport layer's header begins (layer 4).

- `network_header` – A pointer to the location in data where the network layer's header begins (layer 3).

- `mac_header` – A pointer to the location in data where the link layer's header begins.

sk_buff Fields

- `len` – The length of the actual data.
- `data_len` – The length of the buffer.
- `mac_len` – The length of the link layer header.

alloc_skb

```
#include <linux/skbuff.h>

struct sk_buff *alloc_skb (unsigned int size,
                           int          gfp_mask);
```

- **alloc_skb** allocates an **sk_buff** suitable to accommodate size bytes.
- **gfp_mask** is a mask of allocation flags, as described in respect of **kmalloc**.
- **alloc_skb** returns a pointer to an **sk_buff**, or a **NULL** in case of a failure.

alloc_skb

- `alloc_skb` uses a dedicated cache to allocate the memory for `sk_buff`’s.
- It is recommended to use the value of the device’s mtu field (`net_device->mtu`) for `size`.
- Typical ethernet mtu values are 1536, for standard frames, and 9000 for jumbo frames.

kfree_skb

```
#include <linux/skbuff.h>

void kfree_skb (struct sk_buff *skb);
```

- `kfree_skb` frees an `sk_buff`.
- This function should be only used when an skb is not associated with a socket, and / or a list of skb's
- upon removal of a network device driver module, it is responsible to free all the `sk_buff`'s that were pre-allocated for input, and are still hanging around.

consume_skb

```
#include <linux/skbuff.h>

void consume_skb (struct sk_buff *skb);
```

- `consume_skb` frees an `sk_buff`.
- It is the responsibility of the last user of an `sk_buff` to free it:
    - The driver will free an `sk_buff` upon completion of it's transmit.
    - The socket layer frees `sk_buff`'s allocated by the driver.

skb_pull

```
#include <linux/skbuff.h>

unsigned char *skb_pull (struct sk_buff *skb,
                         unsigned int    len);
```

- `skb_pull` removes `len` bytes from the beginning of `skb`.
- A pointer to the new `skb->data` is returned.

skb_push

```
#include <linux/skbuff.h>

unsigned char *skb_push (struct sk_buff  *skb,
                         unsigned int     len);
```

- `skb_push` adds `len` bytes at the beginning of `skb`.
- A pointer to the new `skb->data` field is returned.

skb_put

```
#include <linux/skbuff.h>

unsigned char *skb_put (struct sk_buff  *skb,
                        unsigned int     len);
```

- **skb_put** adds **len** bytes at the end of **skb**.
- A pointer to the previous **skb->tail** is returned.

skb_headroom

```
#include <linux/skbuff.h>

int skb_headroom (const struct sk_buff *skb);
```

- `skb_headroom` returns the space available at the beginning of an `skb`:
  - `skb->data - skb->head`

skb_tailroom

```
#include <linux/skbuff.h>

int skb_tailroom (const struct sk_buff *skb);
```

- `skb_tailroom` returns the space available at the end of `skb`:
  - `skb->end - skb->tail`

ndo_open

```
static int myopen (struct net_device *dev);
```

- The `open` entry point brings a network device up.
- It is responsible to enable device interrupts.
- It powers up a device that is powered down.
- `open` does not correspond directly to a system call.

ndo_open

- The `ifconfig` system utility, makes an `ioctl` command that calls `open` when the CLI option `up` is used.
- The `ioctl` command that calls `open` is implemented at the socket layer, and has to be applied to a socket.
- `open` returns `0` upon success, or a negated error code upon failure.

ndo_open

- User space code sample that causes `do_open` to be called:

```
int sockfd;
struct ifreq ifr;

sockfd = socket(AF_INET, SOCK_DGRAM, 0);

if (sockfd < 0)
    return;

memset(&ifr, 0, sizeof ifr);

strncpy(ifr.ifr_name, "eth0", IFNAMSIZ);

ifr.ifr_flags |= IFF_UP;
ioctl(sockfd, SIOCSIFFLAGS, &ifr);
```

ndo_stop

```
static int mystop (struct net_device *dev);
```

- The **stop** entry point, an **ioctl** command at socket layer, when the system utility **ifconfig** is ran with the CLI option **down**.
- The **stop** entry point should disable interrupts, deactivate the hardware and free resources that are not used when the device is inactive.
- The **stop** entry point returns **0** upon success, or a negated error code upon failure.

start_xmit

```
static int start_xmit (struct sk_buff     *skb,
                       struct net_device *dev);
```

- `start_xmit` transmits the data between `skb->head` and `skb->tail` via the device described by `dev`.

- Transmission is considered complete, once the data has been copied to the device.

- Returns `0` if transmit has started successfully, otherwise `1`.

Interrupts

```
#include <linux/interrupt.h>

irqreturn_t handler (int irq, void *dev);
```

- Registration, and un-registration of network interrupt handlers is identical to the way any other interrupt handler is registered or unregistered.
- The interrupt handler is responsible for checking completion of transmittals.
- Whenever a packet transmission is complete, the interrupt handler should call `consume_skb`, to free the corresponding `sk_buff`.

Interrupts

- Whenever packets arrive, the interrupt handler is responsible to propagate them to the upper layers, that handle their content.

- Newly arrived packets should be:
  - Inserted into an **`sk_buff`** (the actual mechanism depends on how the hardware implements receipt of packets).
  - Network protocol should be identified.
  - **`sk_buff`** should be enqueued for protocol handling.

eth_type_trans

```
#include <linux/etherdevice.h>

unsigned short eth_type_trans (struct sk_buff    *skb,
                                struct net_device *dev);
```

- **`eth_type_trans`** extracts the protocol id from **`skb`**, and returns it.
- It assigns the pointer to the link layer header to **`skb->mac_header`**, and calls **`skb_pull`**.
- The value returned should be assigned to **`skb->protocol`**:

```
skb->protocol = eth_type_trans (skb, netdev);
```

netif_rx

```
#include <linux/netdevice.h>

void netif_rx (struct sk_buff *skb);
```

- `netif_rx` enqueues `skb` for handling by the network layer of `skb->protocol`.

- Actual handling is done by the `softIRQ` that handles network traffic receipt (priority 3).

do_ioctl

```
static int do_ioctl (struct net_device    *dev,
                     struct ifreq         *rq,
                     int                   cmd);
```

- At the user's program `ioctl` commands are applied to sockets.
- The socket layer identifies the device to use by `rq->ifr_name`.
- Request arguments are passed via `rq`.
- Non standard `ioctl` requests can use `rq->ifr_data` for passing arguments.

do_ioctl

- The pointer `rq`, points to a user space buffer and should be handled accordingly.

- The `default` clause in `ioctl`'s switch should return `-EOPNOTSUPP`.

- Notice, that network device drivers have generic `ioctl` commands. See definitions in `<sys/ioctl.h>`.

Exercise

1. Create a module and register it as a network device.
2. Add `open`, `stop`, and `do_ioctl` entry points to your network device driver module.
3. The entry points should be minimalistic.
4. Testing:
   - Try only `ifconfig -a` - this driver is far from providing a working solution!

Socket - Protocol Association

- When a user application creates a socket, the options used determine:
  - The protocol to use in conjunction with the socket.
  - The addressing format to use for data sent over the socket.
  - How data sent / received over the socket is delivered to the application.

Socket - Protocol Association

- Example - creating a TCP/IP socket:

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

  - Internet addressing format (4 octets specifying the host + an unsigned short specifying service / port), will be used.
  - Message ordering is preserved, message boundaries are not preserved.
  - Default protocol selected (TCP/IP).

- Example - create a UDP/IP socket:

```
socket (AF_INET, SOCK_DGRAM, 0);
```

  - Internet addressing format, will be used.
  - Message ordering is not preserved, message boundaries are preserved.
  - Default protocol selected (UDP/IP).

Socket - Protocol Association

- The examples in the previous slide provide the highest possible level of abstraction. The only protocol level detail that the application deals with is network addressing.
- This level of abstraction is suitable for protocols that are implemented by the kernel, and are exposed to user applications.
- But what about protocols that are either:
  - Not implemented.
  - Are implemented, but not exposed at the socket level (**`icmp`**, **`arp`**, ...).

Packet Sockets

- Packet sockets provide the lowest level of abstraction.
- Network traffic may be accessed even at the level of the link layer.
- Protocols may be fully implemented in user mode.
- User applications should be capable of:
  - understanding protocol envelopes when reading.
  - Generating protocol envelopes when writing.

AF_PACKET

- When a socket is created, with address family **`AF_PACKET`**:
  - Data read includes the protocol headers.
  - Application code generates protocol headers when writing data.
- Packet sockets support the following socket types:
  - **`SOCK_RAW`** - Direct access to the link layer. Raw packet data is returned when reading from the socket. Data written to the socket is dispatched to the driver as-is.
  - **`SOCK_DGRAM`** - Only link layer headers are removed.

Creating Packet Sockets

```
socket (AF_PACKET, SOCK_RAW, ETH_P_ALL);
```

- Create a socket that returns raw network data including the Ethernet headers.

```
socket (AF_PACKET, SOCK_DGRAM, ETH_P_ALL);
```

- Create a socket that returns raw protocol data, with the link layer header stripped.

```
socket (AF_PACKET, SOCK_DGRAM, ETH_P_ARP);
```

- Create a socket that returns raw **ARP** packets without the link layer header.

AF_PACKET - Summary

- Application code has to handcraft protocol envelopes for the data sent via these **`AF_PACKET`** sockets.
- **`AF_PACKET`** sockets provide access to protocols that are not normally exposed to the user.
- **`AF_PACKET`** sockets may be used to implement in user mode, protocols that are not implemented in the kernel.
- **`AF_PACKET`** sockets are commonly used to implement network traffic monitoring software (sniffers, etc.).

Socket - Protocol Association

- The kernel associates a `proto_ops` structure with the `socket` structure, when the socket is created.
- Each valid combination of family, type, protocol, defines a unique `proto_ops` structure.
- The callbacks in the `proto_ops` structure correspond to socket related system calls.
- The protocol operations use kernel internal routing tables to determine the network device to use.

What is Net Filter

- Net filter is the Linux infrastructure for special handling of network traffic, beyond simple protocol processing.

- It defines several stages during the network protocol processing, at which specialized functions may be hooked at.

- Net filter hooks are used in Linux to implement the following and more:
  - Drop incoming traffic that is not allowed.
  - Masquerade network addresses.
  - Log network traffic information.

Implementing a Net Filter

- A net filter is a kernel module, that implements hook functions, per the net filter prototype definitions, and registers them with the desired “hooking points”.
- The return value of a hook function determines if and how to proceed processing a packet.
- Support of net filters is limited to bridges, decent, ipv4, ipv6, and arp, as of Linux 5.4.
- Hook functions are prioritized to set the order of multiple functions assigned to the same hook.

The Net Filter API

- The generic netfilter API is defined in **`<linux/netfilter.h>`**.

- The generic API provides:
  - A structure to describe a hook.
  - Functions for registering / unregistering hooks.
  - Hook function prototype definition.
  - Additional definitions.

The Net Filter API

- Per protocol net filter related definitions reside in protocol specific files:
  - `<linux/netfilter_ipv4.h>` - ipv4 related definitions.
  - `<linux/netfilter_ipv6.h>` - ipv4 related definitions.
  - `<uapi/linux/netfilter_arp.h>` - arp related definitions.
  - . . . .

ipv4 Hooking Points

Hook	Description
`NF_IP_PRE_ROUTING`	handle incoming packets before the routing code handles them.
`NF_IP_LOCAL_IN`	handle incoming packets destined to the local host.
`NF_IP_FORWARD`	handle incoming packets not destined to the local host.
`NF_IP_LOCAL_OUT`	handle packets that originate from the local host, before the routing code handles them.
`NF_IP_POST_ROUTING`	handle outgoing packets before they leave the computer via the network device.

arp Hooking Points

Hook	Description
`NF_ARP_IN`	Handle inbound packets before any protocol code handles them.
`NF_ARP_OUT`	Handle outbound packets before handing them over to the driver.
`NF_ARP_FORWARD`	Handle inbound packets that are destined to another host.

The Hook Function

```
unsigned int nf_hookfn(void                        *priv,
                       struct sk_buff              *skb,
                       const struct nf_hook_state *state);
```

- The hook function analyzes **`skb`**, and determines its verdict.

- **`priv`** – Module internal private data. The layout of **`priv`** is arbitrarily decided by the module's author, along with its content and whatever it represents.

- **`skb`** - the **`sk_buff`** that holds the packet that has to be processed by this hook.

The Hook Function

- `state` – This structure combine fields from the `nf_hook_ops` structure with which this hook was registered, along fields that describe the source / target device of `skb`, and what to do if `skb` is accepted.

- The fields below were formerly parameters of the hook function:
  - `in` - the `net_device` structure of the device from where `skb` originated†.
  - `out` - the `net_device` structure of the device to which `skb` is destined†.
  - `okfn` - a function to invoke when all the functions that are related to this hook complete successfully (return `NF_ACCEPT`).

†May be null if the packet flow is in the opposite direction.

Hook Function Return Values

The Verdict	
Value	Description
`NF_DROP`	the packet should be dropped without any further processing.
`NF_ACCEPT`	the packet should be passed to the next hook function in the list. If this is the last hook function okfn() should be invoked.
`NF_STOLEN`	processing by the net filter will be stopped, and the packet will be further processed per the decision taken by the current hook.
`NF_QUEUE`	the packet should be put on a queue for further processing by user code.
`NF_REPEAT`	the current function should be invoked again on this packet.

The Legacy Hook Function Prototype

```
unsigned int nf_hook_func (unsigned int            hooknum,
                           struct sk_buff          *skb,
                           const struct net_device *in,
                           const struct net_device *out,
                           int (*okfn)(struct sk_buff *))
```

- This prototype was used up to Linux 3.12 (inclusive).
- Yet, RHEL 7.x, and RHEL 7.x based distro’s, using Linux 3.10 have applied patches that incorporate the new API.

Registering a Hook With Net Filter

```
struct nf_hook_ops {
  /* User fills in from here down. */
  nf_hookfn *hook;
  struct net_device *dev;
  void *priv;
  u_int8_t pf;              /* protocol family */
  unsigned int hooknum;     /* The hooking point */
  /* Hooks are ordered in ascending priority. */
  int priority;
};
```

- The `nf_hook_ops` structure has to be filled prior to registering the network hook.

Hook Registration

```
#include <linux/netfilter.h>

int nf_register_net_hook (struct net          *net,
                          struct nf_hook_ops *nfho)
```

- Where `nfho`, is a pre-initialized `nf_hook_ops` structure.
- `net` describes the network namespace to which this hook should correspond. The systems default network namespace is described by the global structure `init_net`.
- Typically the registration will take place in the modules initialization function.

Removing a Hook

```
#include <linux/netfilter.h>

void nf_unregister_net_hook (struct net         *net,
                             struct nf_hook_ops *nfho)
```

- Where `nfho` is the `nf_hook_ops` structure with which it was registered.
- `net` is the network namespace within which the hook was registered
- This function is typically called in the modules `exit` function.

Exercise

1. Write a hook that "catches" `arp` replies (`NF_ARP_IN`) and prints on the console the MAC addresses of remote systems (Bytes 6-11 of the link layer header).

2. Generate some outgoing traffic (*e.g.* `ping`) and inspect the output of `dmesg` for messages generated by your net filter.