Network Drivers

What to Expect?

- Understanding Network Subsystem
- Network Drivers: A different category
- Writing Network Drivers

Enhancing the Character Vertical

- Packet Parsing logic
  - Add the header/footer during the transmission & remove it during reception
  - Can be added as library and reuse as and when needed
- From implementation
  - Every layer has its own header/footer
  - How do we add the header/footer?
    - Make a copy at each layer
    - Have the buffer allocated with maximum possible capacity to account for the header/footer
      - Hence, we would make structure with required header/footer & data
        - We call this as a socket buffer

Packet Routing

- Multiple processes would be doing ‘cat /dev/nw’
  - How to know which process to give what packet?
    - Where to parse the packet
    - User space?
    - In driver space?
    - Better to have the library (generic code) to do this parsing
  - Where to place the library?
    - Peer to the driver?
    - Just above the character vertical
      - This is called as network library or networking stack

Initialization

- Done at 3 places
  - Init
    - Software related initialization such as driver registration, driver related, buffer related initialization
  - Probe
    - We do horizontal specific or controller specific initialization such as getting the physical to virtual memory
    - Registering the vertical
    - Protocol related initialization
  - Open
    - Device specific initialization
    - Enabling the interrupt

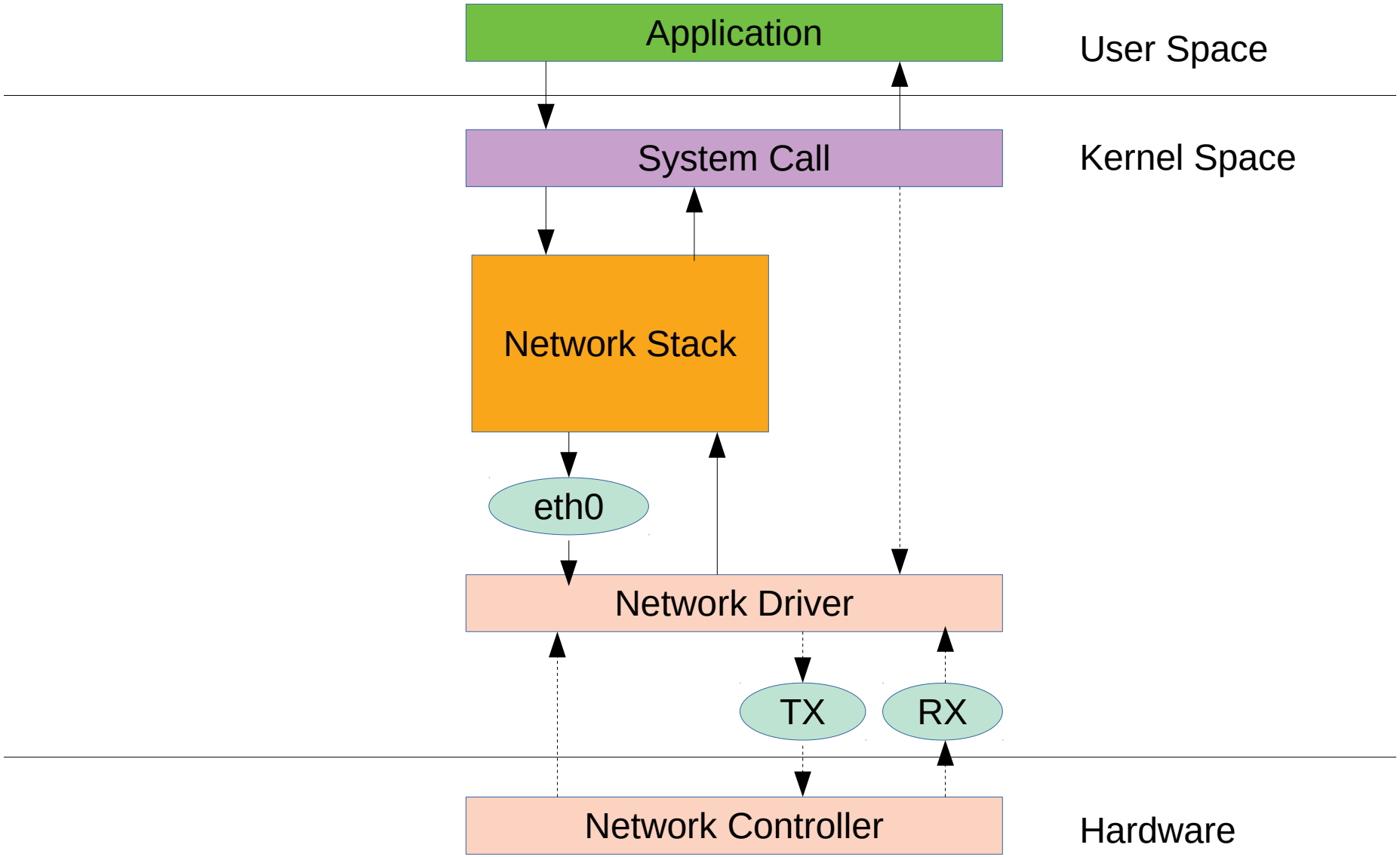
Asynchronous Data

- Data comes from anywhere & anytime
  - Even outside the system
  - For other drivers, its under the purview of the system
- Another thing is that there is no clue to whom does the packet belongs to

Using the open for initialization

- Whenever, any of the process calls the open, initialization is done
  - Buffer allocation is done & descriptor is being setup
  - And this can’t be done multiple times
  - Maintaining the reference count
  - Again, it doesn’t solve the purpose of whom does the packet belongs to?
- That’s why no device file & no open to the user space
  - So, VFS is replaced with network stack
  - The device interface is internally maintained by network stack
  - File operations are not provided to user space, instead n/w stack deals with the user space

Network Subsystem



Network Subsystem

- (Network) Protocol Stack
  - Typically, the TCP/IP Stack
  - Interfaces with the User Space through network interfaces, instead of device files
  - Unlike other drivers, does not provide any /sys or /dev entries
  - Interface Examples: eth0, wlan1, …
  - Resides in the <kernel_source>/net folder
- Network Interface Card (NIC) Drivers
  - Driver for the Physical Network Cards
  - Provides uniform hardware independent interface for the network protocol layer to access the card
  - Typically, resides in the <kernel_source>/drivers/net folder

Network Interface Structure

- Represented in the kernel as an instance of the struct net_device
- #include <linux/netdevice.h>
- Fields
  - Interface name
  - netdev_ops
  - features
  - min_mtu, max_mtu
  - stats

Net Device Registration

- Header: <linux/netdevice.h>
- Net Device Storage
  - struct net_device *alloc_etherdev(sizeof_priv);
  - struct net_device *alloc_ieee80211dev(sizeof_priv);
  - struct net_device *alloc_irdadev(sizeof_priv);
  - struct net_device *alloc_netdev(sizeof_priv, name, name_assign_type, setup_fn);
  - void free_netdev(struct net_device *);
- Registering the Net Device
  - int register_netdev(struct net_device *);
  - void unregister_netdev(struct net_device *);

struct net_device

- Header: <linux/netdevice.h>
- Driver relevant Operation Fields
  - Activation: open, stop, ioctl
  - Data Transfer: start_xmit, poll (NAPI: new API)
  - WatchDog: tx_timeout, int watchdog_timeo
  - Statistics: get_stats, get_wireless_stats
    - Typically uses struct net_device_stats, populated earlier
  - Configuration: struct *ethtool_ops, change_mtu
    - Structure defined in Header: <linux/ethtool.h>
  - Bus Specific: mem_start, mem_end
- Latest kernels have all the operations moved under
  - struct net_dev_ops
  - And typically, a prefix ndo_ added and some name changes

Network device Operations

- struct net_device_ops
  - Represents the operations that can be performed on the network interfaces
  - The structure used by network driver to register the operations
  - One of the field in struct net_device
  - Fields
    - ndo_init
    - ndo_uninit
    - ndo_open
    - ndo_stop
    - ndo_start_xmit
    - ndo_do_ioctl
    - ndo_tx_timeout

Related Data Structures

- Writing a NIC or Network Driver involves
  - Interacting with the underlying Card over its I/O Bus
  - Providing the standard APIs to the Protocol Stack
- This needs three kind of Data Structures
  - Core of Protocol Stack
    - struct sk_buff
  - NIC & Protocol Stack Interface
    - struct net_device
  - NIC I/O bus, e.g. PCI, USB, …
    - I/O bus specific data structure

struct sk_buff

- Network Packet Descriptor
- Header: <linux/skbuff.h>
- Driver relevant Fields
  - head – points to the start of the packet
  - tail – points to the end of the packet
  - data – points to the start of the pkt payload
  - end – points to the end of the packet payload
  - len – amount of data that the packet contains
- Refer http://vger.kernel.org/~davem/skb_data.html

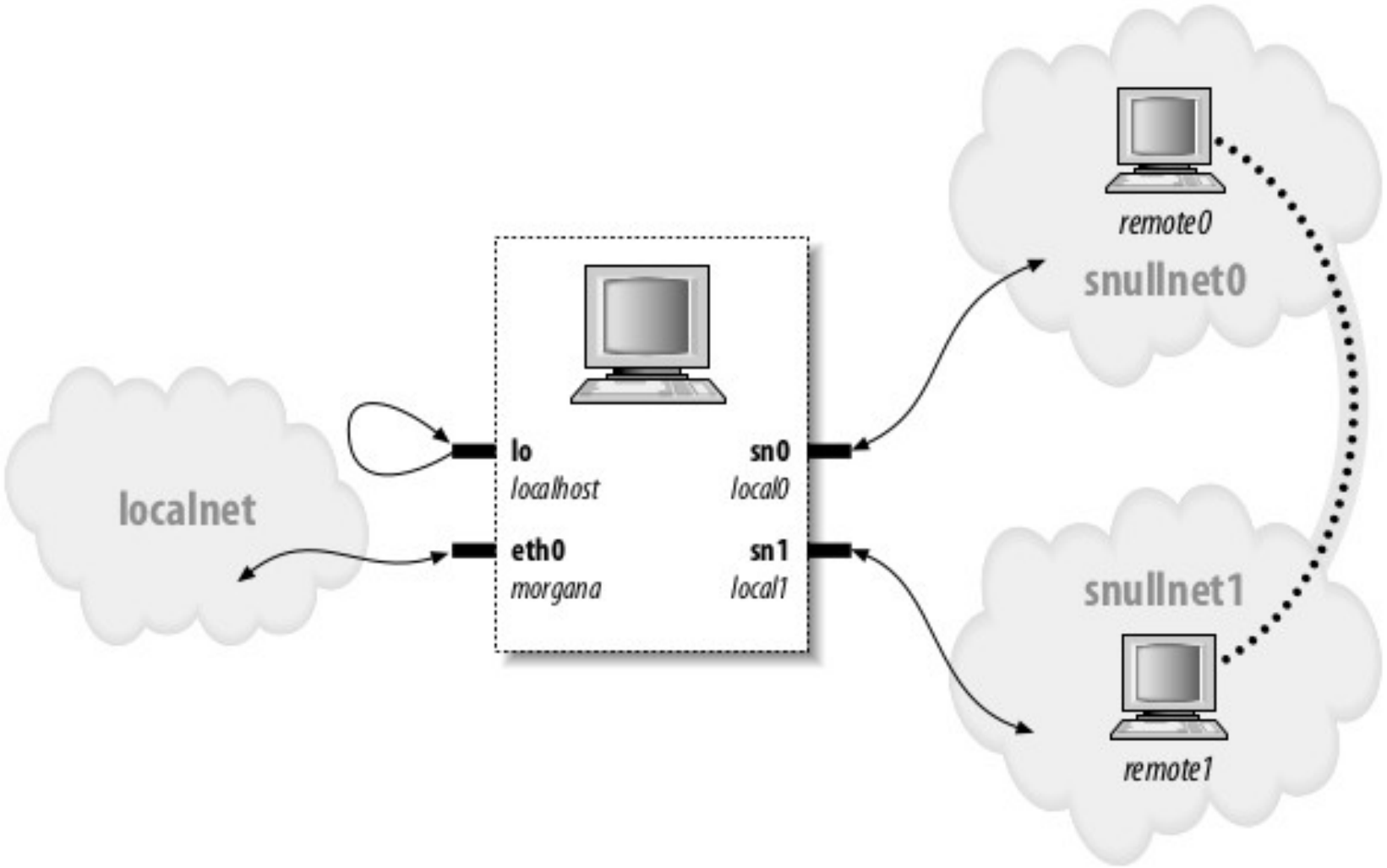
Socket Buffer APIs

- Header: <linux/skbuff.h>
- SK Buffer Storage
  - struct sk_buff *dev_alloc_skb(len);
  - dev_kfree_skb(skb);
- SK Buffer Operations
  - void skb_reserve(struct sk_buff *, int len);
  - struct sk_buff *skb_clone(struct sk_buff *, gfp_t);
  - unsigned char *skb_put(struct sk_buff *, int len);

Simple Network Utility

- Dummy network driver that doesn’t talk to the actual devices
- Simulates actual operations
- Produces 2 interfaces to simulate 2 external links
- Works as a kind of hidden loopback

Snull Interfaces



IP Addresses for snull

- snullnet0 - 192.168.0.0
- snullnet1 - 192.168.1.0
- local0 – 192.168.0.1
- remote0 – 192.168.0.2
- local1 – 192.168.1.2
- remote1 – 192.168.1.1

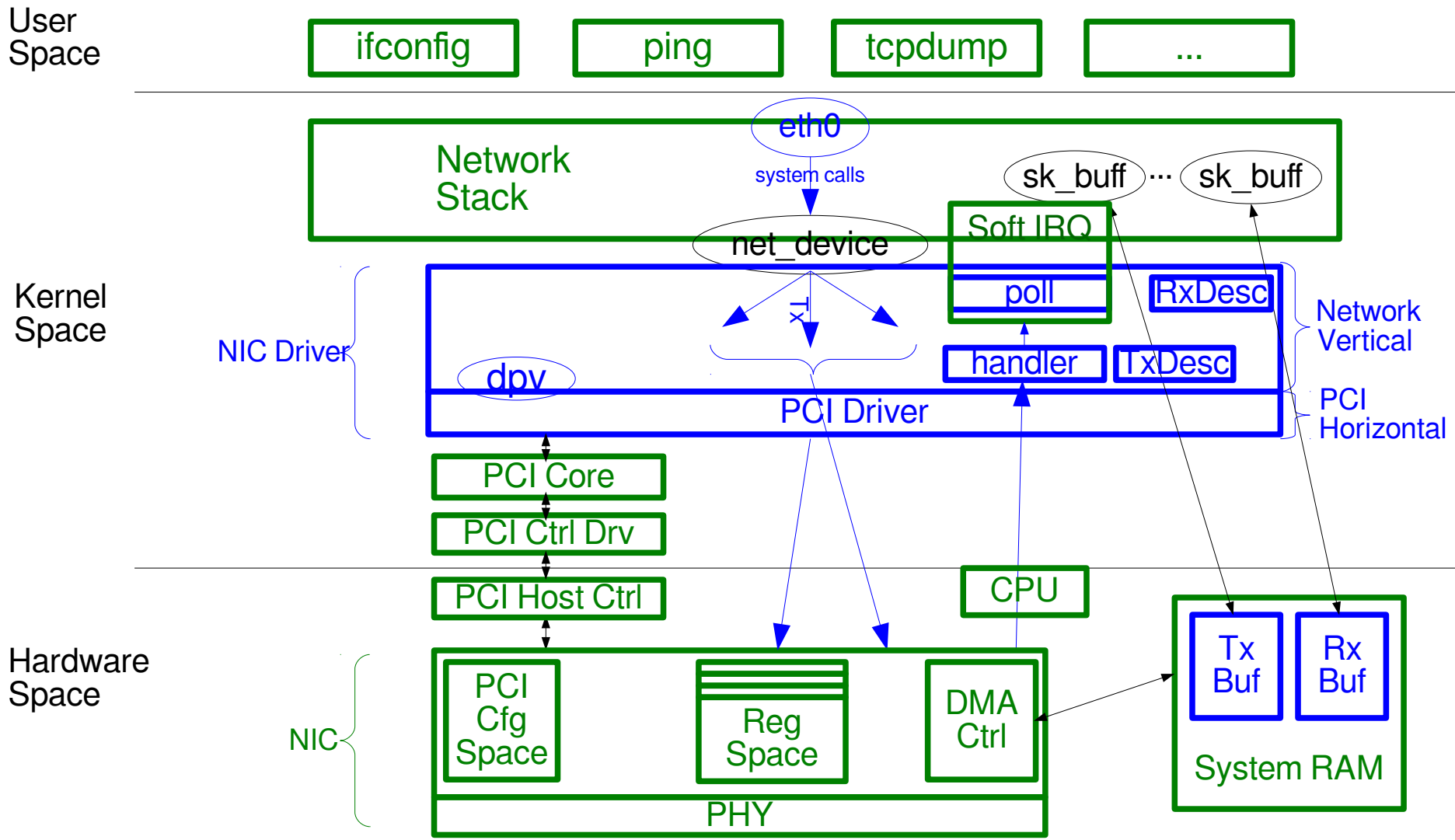
New API (NAPI)

- High bandwidth interfaces may receive thousands of packets per second
  - This in turn means thousands of thousands of interrupts per second
  - This may severely degrade the system performance
- To overcome this, there exists an alternative interface called NAPI based on polling
- NAPI involves disabling the receive interrupts once the first packet is received
- The subsequent packets are received by polling

NAPI APIs

- netif_napi_add
  - Initialize the NAPI context
  - Paramters
    - Network device
    - NAPI context
    - Polling function
    - Weight
      - Defines the relative importance of the interface
      - How much traffic should be accepted from the interface, when the resources are tight
      - By convention, 10Mbps Ethernet interface set weight to 16 and faster interfaces set it to 64
- napi_schedule
  - Schedule a NAPI poll routine
- netif_receive_skb
  - Push the packet to the nw layer
- netif_complete_done
  - Returns true, if the interrupts should be enabled

The Big Picture (with a PCI NIC)



RT8169 Network Driver

- Networking Vertical (net_stk.c)
- PCI Horizontal (pci_net.c)
- Mac Level Operations (mac.c)
- Phy Level Operations (phy.c)

Network Device Open & Close

- A typical Network Device Open
  - Allocates ring buffers and associated sk_buffs
  - Initializes the Device
  - Gets the MAC, … from device EEPROM
  - Requests firmware download, if needed
    - int request_firmware(fw, name, device);
  - Register the interrupt handler(s)
- A typical Network Device Close
  - Would do the reverse in chronology reverse order

Packet Receive Interrupt Handler

- A typical receive interrupt handler
  - Minimally handles the packet received
    - Sanity checks
  - Puts back equal number of sk_buffs for re-use
  - Passes the associated sk_buffs (& ring buffers) to the protocol layer by the NET_RX_SOFTIRQ
  - On higher load, switch to poll mode, if supported, which then passes the associated sk_buffs (& ring buffers) to the protocol layer by the NET_RX_SOFTIRQ

Flow Control related APIs

- For interaction with the protocol layer
- Header: <linux/netdevice.h>
- APIs
  - void netif_start_queue(struct net_device *);
  - void netif_stop_queue(struct net_device *);
  - void netif_wake_queue(struct net_device *);
  - int netif_queue_stopped(struct net_device *);

Network Driver Examples

- Driver: snull
- Browse & Discuss

- Driver for Realtek NIC 8136
- Browse & Hack

What all have we learnt?

- Linux Network Subsystem
- How are Network Drivers different?
- Writing Network Drivers
  - Key Data Structures & their APIs
    - sk_buff & net_device
  - Network Device Registration
  - Network Device Operations
  - Interrupt Handling for Packets received
  - Flow Control related APIs