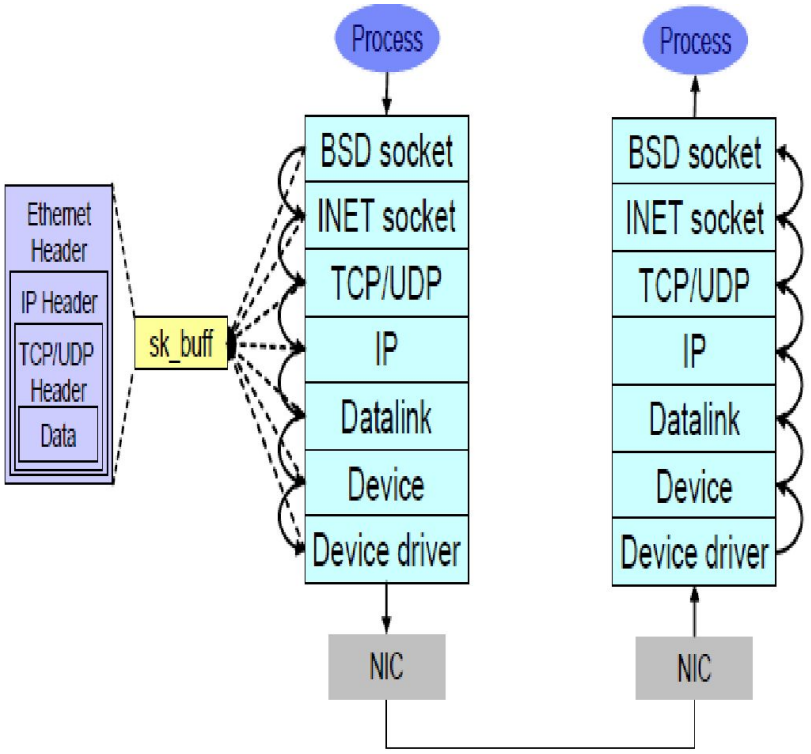
Module 2

Network Driver

Linux Network Stack

- An Ethernet frame arrives at the wire is moved into the NIC buffer if the MAC address matches the MAC address of the NIC.
- NIC eventually moves the packet into the network buffer of the Operating System kernel and then issues a hardware interrupt to the CPU. The CPU then process the packet and moves it up the network stack until it arrives at TCP or UDP port of an application
- All data in Linux network stack is maintained by a special control structure called ***sk_buff***
- When the buffer passes up/down through the network layers: MAC, IP, TCP, UDP, headers from the previous layer are no longer needed.
- Instead of removing the headers, the pointers in the ***sk_buff*** are adjusted to the beginning of the payload for the next layer header. This operation requires fewer cpu cycles and thus more efficient.



Building Block of Linux Networking

- **Device or Interface:** A network interface represents a thing which sends and receives packets. This is normally interface code for a physical device like an ethernet card. However some devices are software only such as the loopback device which is used for sending data to the same host.
- **Protocol:** Each protocol is effectively a different language of networking. Within the Linux kernel each protocol is a separate module of code which provides services to the socket layer. Example: TCP/IP Network Stack
- **Socket:** A socket is a logical connection in the networking that offers unix file I/O semantics and exists to the user program as a file descriptor. In the kernel each socket is a pair of structures that represent the high level socket interface and low level protocol interface.
- **sk_buff:** Buffers used by Linux networking stack are of type *sk_buff*. sk_buff structure is used to manage payload in the protocol stack. Control for these buffers is provided by core low-level library routines. sk_buffs supports buffering and flow control facilities required by Linux network stack.

NIC Driver

- Unlike other classes, applications interact with a NIC drivers via a network interface
  - Example: eth0 for the first Ethernet interface
- Ethernet Interface abstracts an underlying protocol stack, typically, the TCP/IP stack
- NIC drivers do not rely on /dev or /sys to communicate with user space. Means no file system representation!
- TCP/IP stack in Linux makes it easy to slide a NIC driver without knowing much about the upper layer protocols

You can set up networking in Linux using netconfig where IP address, netmask, gateway, primary nameserver are applied at boot time. One can use ifconfig or ip commands to bring up/down the interface card

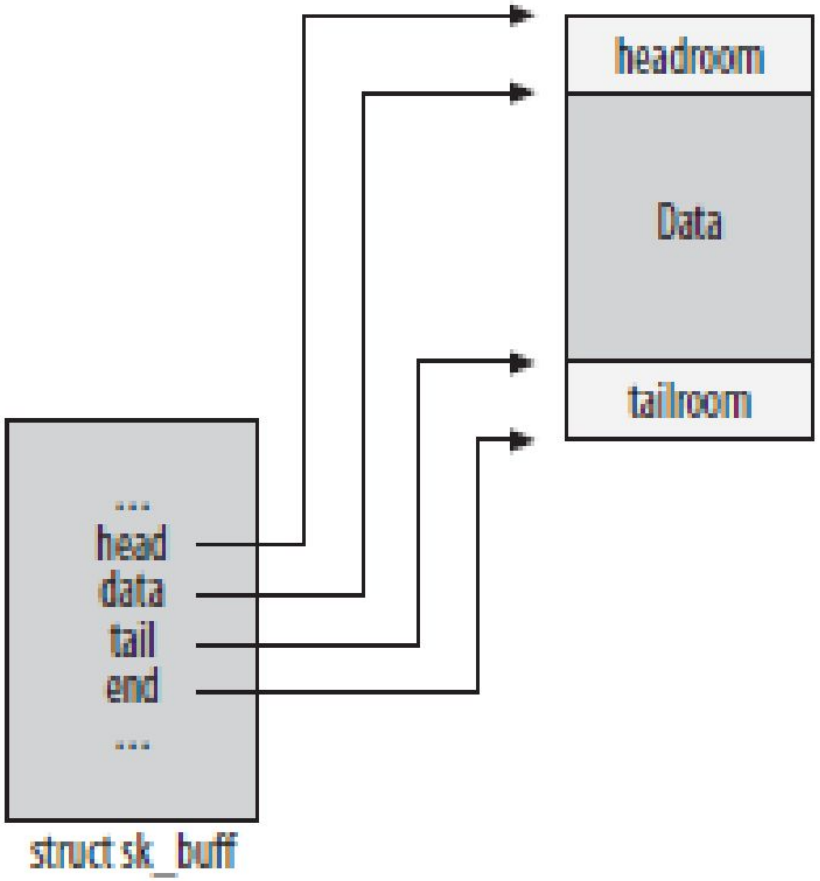
NIC Data Structures

- **struct sk_buff:** Structure that forms the building blocks of the network protocol stack. It is defined in **sk_buff.h.** It is the key structure used by the kernel's TCP/IP stack. The structure is used by Linux network stack for storing header information about the user data (payload) and other information needed internally.
- **struct net_device:** Defines the interface between the NIC driver and the protocol stack. It is defined in **netdevice.h.** It is the core structure that constitutes the network interface. There is one for each network device. It contains both hardware and software information about the NIC
- Structures specific to **I/O buses** used by NIC cards. PCI and USB are common buses used by today's NICs.

sk_buff

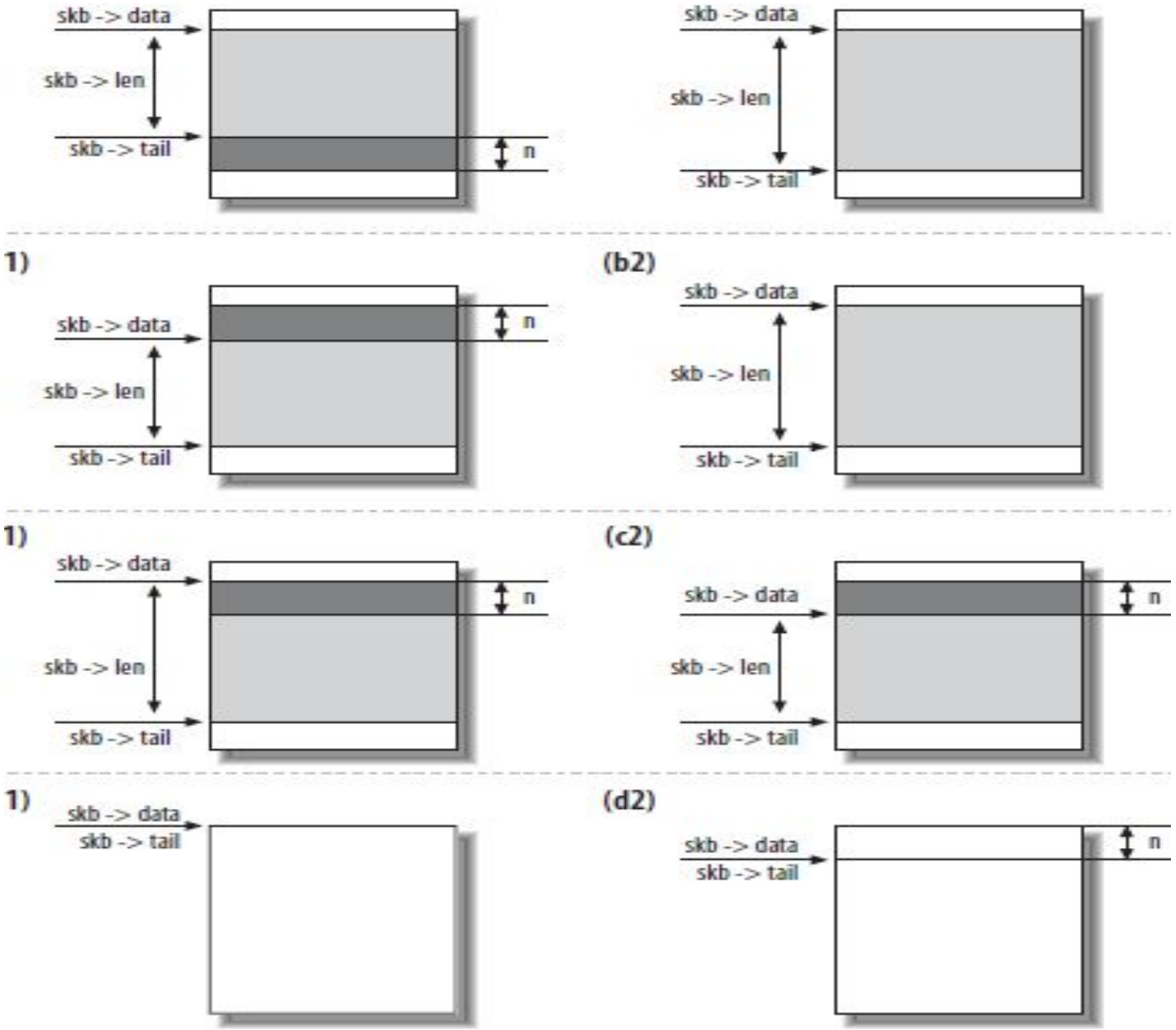
- sk_buff is a control structure with a block of memory attached
- sk_buff links itself to its associated packet (payload) using five main fields:
  - ***head**, which points to the start of the packet*
  - ***data**, which points to the start of packet payload*
  - ***tail**, which points to the end of packet payload*
  - ***end**, which points to the end of the packet*
  - ***len**, the amount of data that the packet contains*
- These fields slide over the associated packet buffer as the packet traverses through the protocol stack in either direction.
  - When the buffer passes up/down through the network layers: MAC, IP, TCP, UDP, headers from the previous layer is no longer needed. Instead of removing the headers, the pointer to the beginning of the payload is moved ahead to the beginning of the next layer header. This operation requires fewer cpu cycles and thus more efficient.
- All these fields are updated whenever *skb->len* field is incremented at every stage. *skb->data* field points to the header of protocol currently processing the packet.

Socket Buffers - sk_buffs



sk_buff library functions

- These are list functions to update doubly linked lists of *sk_buffs*. Buffers are appended to the end and are removed from the start. These routines are written to be efficient, bug free and atomic considering some of them are called in interrupt context.
- See **net/core/skbuff.c** for the full list of *sk_buff* library functions. These functions don't really add any data to the buffer, they simply move the pointers to its head or tail:
- **skb_reserve** - This function increments both data and tail fields of the sk_buff. This function is commonly used for adding space to the beginning of a buffer. This is the first thing done by network stack is to reserve space for protocol headers as the buffer passes down through layers. It is also used by the NIC device drivers to align the IP header of incoming frame
- **skb_put** – This function adds data to the end of the buffer and updates the tail and len fields of the sk_buff.
- **skb_push** - This function adds data to the beginning of the buffer and decrements data and increment len fields
- **skb_pull** - This function removes data from the head of the packet by moving the head pointer
- **skb_headroom** - This function returns the amount of space in front of data
- **skb_tailroom -** This function returns amount of space available in skb



4. Before and after: (a)skb_put, (b)skb_push, (c)skb_pull, and (d)skb_reserve

Net Device Interface

- Network devices deal with transmission of network buffers from the protocol stack (TCP/IP) to the physical media
- Network device also decodes the frame that hardware generates.
- Incoming frames are then turned into network buffers (sk_buff), and delivered to the protocol stack for further processing.
- Each device provides a set of additional methods (called *net_device methods*) for the network packet handling. These method and other control information are kept in net_device structure and are used for managing devices

NOTE: Receive part of network packet processing is dealt by the Interrupt Service Routine (ISR) that copies data from DMA buffer into sk_buff and pass it to network stack.

net_device structure

- The *net_device* structure stores all information for handling a network device. There is one such structure for each device.
- *sk_buff* structure contains a pointer to struct *net_device*.
  - This field is updated by NIC driver before passing a packet to the protocol layer representing a receiving interface. Protocol layer also updates this field representing transmitting interface.
- Like *sk_buf, net_device* is a large structure, Some fields of this structure that applies to NIC driver are listed below:
  - **name**[]: name of the device, default eth0 for first NIC
  - **mem_start, mem_end, base_addr**: Applies to devices that supports memory mapped IO (PCI) to device memory. NIC driver update these field so that *ifconfig* can display the memory mapped IO information for device.
  - **irq**: Interrupt number assigned to device. NIC driver updates this field so that *ifconfig* can display device irq number.
  - **mtu**: Maximum Transmission Unit. Ethernet frame size is 1500
  - **hard_header_len**: Device header in octets. For Ethernet, it is 14 octets
  - **broadcast[]**: Link layer broadcast address
  - **dev_addr[]**: MAC address. For Ethernet, it is 6 octets
  - **tx_queue_len:** maximum number of frames that can be queued for the device by the protocol stack

net_device methods

- Similar to char file operation methods, network driver defines methods to control NIC hardware.
- net_device methods are invoked by Linux network stack.
- net_device methods are used to open, close, initialize, cleanup, start and stop the device.

Driver Method	Detail
open	Opens the device. Device is opened when ifconfig is invoked. It registers the associated net_device structure, initialize hardware and sets up transmit and receive DMA and interrupt handler.
stop	Clean up, reverse of open(), when network interface is brought down.
hard_start_xmit	network stack invokes this method and pass pointer to sk_buff. Full packet (protocol headers and all) is contained in the *sk_buff*. Driver loads this sk_buff into DMA buffers for transmission to NIC.
tx_timeout	Transmission timeout. This method is called to reset the device after watchdog_timeo (jiffies) value in the net_device structure is expired.
get_stats	Returns *net_device_stat* struct containing device stats to ifconfig
poll	Method provided by NAPI compliant drivers to operate the device in polled mode with interrupt disabled

kernel version 2.6.27 and above have netdevice methods moved under separate structure: net_dev_ops

Network Receive Path

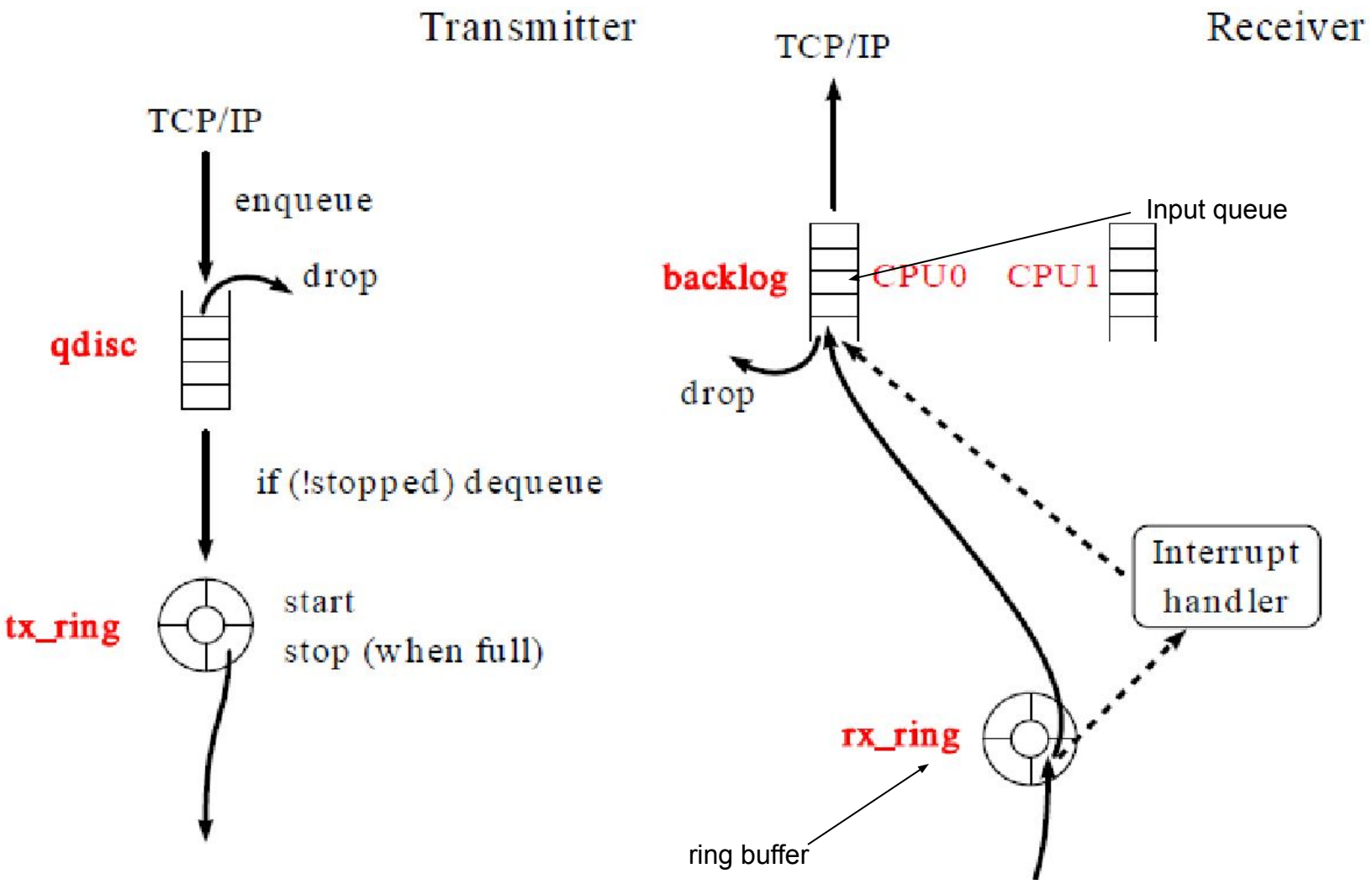
- Network Interface Cards are DMA capable. Once NIC receives the frame, it DMA the frame into kernel memory used as a ring buffer and then notify the kernel by sending interrupt (IRQ #, or interrupt vector) to CPU.
- Kernel invokes Network Driver Interrupt Service Routine (ISR) that copies the frame from the ring buffer into the CPU input queue.
- Network Drivers that are not NAPI aware, calls *netif_rx()* to queue the input frame into input_pkt_queue (part of cpu softnet_data structure)
- Linux network stack uses softirqs of types: *NET_RX_SOFT_IRQ* and *NET_TX_SOFTIRQ* for processing incoming and outgoing packets.
- When the packet is queued into CPU input queue, NET_RX_SOFT_IRQ softirq is invoked that process the frames in the input queue and pass it to the protocol stack, for example IP. There is a one softirq per CPU.
- If CPU input queue has space, frame will be copied. Otherwise, it will be silently dropped by the network driver and may result in retransmission at the upper layer.

Network Transmit Path

- Protocol stacks (IP) calculates the route for outgoing packet and selects one of the NIC (if there are multiple NICs) in the system for packet transmission.
- Each device maintains a queue of packets to be transmitted.
- Queue disciple is named ***qdisc***. Default qdisc is FIFO (First In First Out), but other sophisticated schemes such as RED (Random Early Detection), CBQ and other can be selected using:

  # tc qdisc add dev eth0 root <algorithm><alg options>
- Network driver copies the packet to one of the NIC transmit descriptor.
- Device reads the packet from kernel memory into device memory via DMA.
- Depending on the type of NIC, there can be several transmit descriptor per device.
- When device does not have any free transmit descriptor, Network driver flow controls the protocol stack that pauses the transmission until device catches up and transmit descriptors becomes available.

NIC Transmit and Receive



NIC Poll Method (NAPI)

- NAPI improves network performance by allowing drivers to run with interrupts disabled during high network activity and use polling to process packets received on the network adapter. This feature is called *"Interrupt mitigation"*.
  - As long as there are packets in the DMA ring buffer of NIC, no new interrupt will be generated
- NAPI aware driver uses mix of interrupts and polling. Instead of using an interrupt per frame, driver disables interrupts and request protocol stack to process multiple frames by registering ***poll()*** net_device method.
- *softnet_data* structure contains ***poll_list*** to keep track of devices that have interrupt disabled and frames waiting in the device memory to process.
  - sotftnet_data structure also contains "*quota*" and "*weight*" to control maximum number of buffer that can be processed by *poll*() in one shot and to enforce fairness among multiple devices.
- Devices ***poll()*** function is invoked in a round robin fashion. If the queue cannot be emptied in one time slot, they have to wait for their next iteration.
- Another feature of NAPI is "Packet throttling", which prevents system to be overwhelmed by packet storms. NAPI-compliant drivers can drop packets queued in network adapter before kernel sees it

NIC Poll Method (NAPI)

- NAPI aware drivers calls ***netif_rx_schedule*** instead of *netif_rx*(). Device is added to the *poll_list* by calling ***netif_rx_schedule*()**.
- For NAPI aware drivers, softirq *NET_RX_SOFT_IRQ* browses the *poll_list*() of devices that have frame sitting in the device memory.
- *poll()* net_device of driver is invoked to receive the frame. driver poll routine calls *netif_receive_skb()* to enqueue packet into local cpu backlog. If *RPS/RFS* feature is enabled, *get_rps_cpu()* routine is called by *netif_receive_skb()* to locate the cpu associated with TCP flow and then enqueue packet (skb) into remote cpu backlog.
- *poll()* can temporarily disable and re-enable the polling with ***netif_poll_disable*()** and ***netif_poll_enable*()**.
- When there is no more frame left in the device memory, poll function exits by calling ***netif_rx_complete*()**, that re-enable interrupts.
- Softirq *NET_RX_SOFT_IRQ* stops processing when there is no more devices in the *poll_list*; it ran out of time (ran for too long and thus scheduled again for execution to be fair with other tasks); or reached upper bound limit (budget). ***b***

Typical Network Driver Initialization

- Allocates DMA buffers and associated *sk_buffs*
- Allocates and initialize *net_device struct*
- Register the *net_device* structure with protocol stack
- Read the MAC address from the device. Request firmware download, if needed.
- Register interrupt handler for interrupt processing. However, postponing till device open is recommended. A typical interrupt handler performs the following operation:
  - Minimally handles the packet received
  - Allocates *sk_buffs. Copy packet into sk_buff managed buffer*
  - Passes the associated *sk_buffs* to the protocol stack using *netif_rx, if not NAPI*
  - Register poll() net_device method with protocol stack, if NAPI. This allows packets processing to happen in context of poll()

Device Registration and Initialization

- Each network device is represented by *net_device* structure. This structure is initialized by both core kernel routines and device driver. ***alloc_etherdev***() allocates net device structure for wired ethernet NIC.
- Device driver requests IRQ for the device and register the Interrupt Service Routine (ISR) via *request_irq*(). PCI devices are assigned interrupts automatically by BIOS.
  - Status of device IRQ binding can be extracted by reading **/proc/interrupts** file.
- Request access to device IO memory regions (PCI). IO memory is a region of PCI device memory (registers) that can be mapped into RAM.
- Populate the net_device struct with NIC net_device methods such as: ***open, stop, hard_start_xmit, getstats etc..***
- Register the net device structure via register_netdev(). An unused ethernet interface is allocated. This links the device structure into the kernel network device tables.

Net Device Method – open

- Requests an IRQ via request_irq()
- Allocates DMA buffers for packet transmission and reception.
- Initialize the hardware to make sure it is ready to process packets
- Notify the protocol layer so that it can start transmitting packets to us. ***netif_start_queue***() is used for this purpose.

Net Device Method – Transmit

- All net devices must provide a transmit function
- Protocol stack invokes net device ***hard_start_xmit()*** method when it needs to transmit a packet.
- If there is a room then the buffer should be processed by copying into DMA buffer. The buffer handed down already contains all the headers, including link layer headers, and need only be loaded into the hardware for transmission.
- The driver gets the packet out of the door by DMA-ing packet data into the NIC hardware. Once the data is copied into DMA buffer, sk_buff can be freed by ***dev_kfree_skb***()
- If hardware is unable to accept the packet (no transmit descriptor) it should call ***netif_stop_queue***(). This causes the protocol layer to stop sending packets.
- Status of packet transmission (failure or success) is sent to driver by hardware via interrupt.

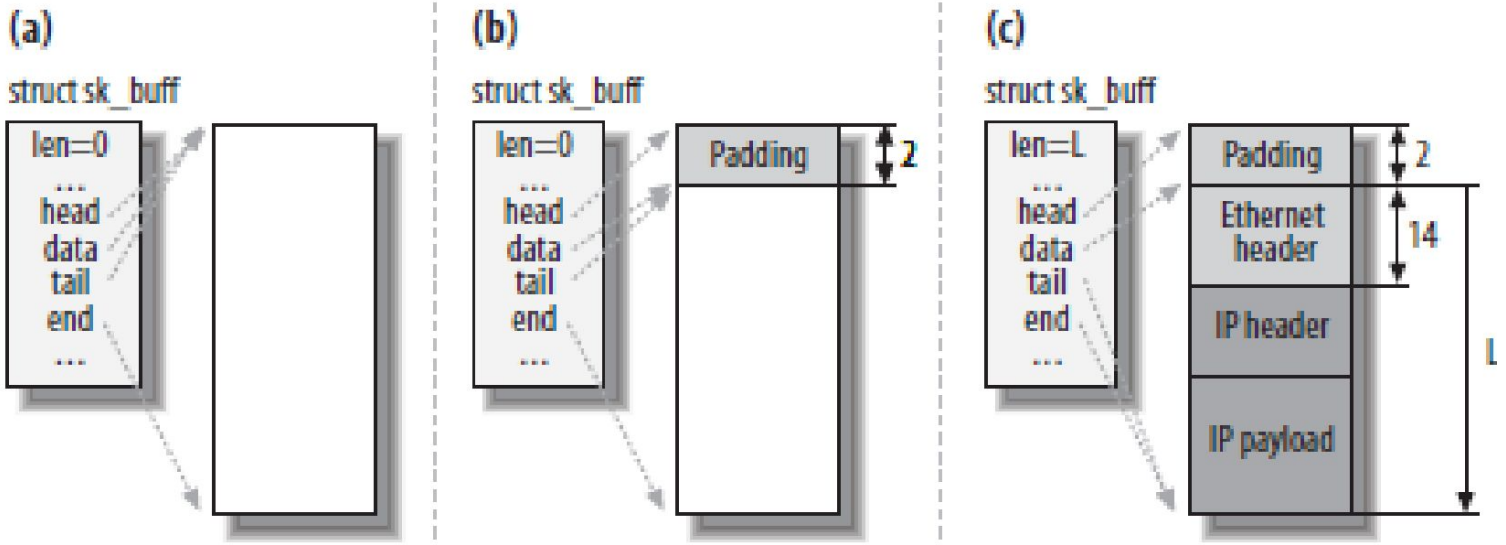
Net Device Method - Receive

- There is no receive method in a network device, because it is the hardware interrupt that invokes processing of such events.
- With a typical device, an interrupt notifies the kernel that a completed packet is ready for reception. Frame is written into kernel memory by the device via DMA
- Driver ISR routine copies the packet into *sk_buff*, that is allocated via **dev_alloc_skb()**.
  - *dev_alloc_skb()* allocates a buffer and calls **skb_reserve(skb, 2)** to add 2-byte padding at the start of *sk_buf*.
  - 2-byte padding is to allows packet to land into the *sk_buf* with IP header aligned at 16-byte boundary.
- After copying frame into skb, device driver analyzes the frame to find the packet and protocol type. **eth_type_trans(skb, dev)** is called to extract the protocol type (IP in case of TCP/IP stack) and then stored in *skb_protocol* field.
  - driver updates the *net_device* fields in *sk_buff* .. that now point to the net device that has received the frame
  - Finally, the device driver invokes ***netif_rx()*** to pass the buffer up to the protocol layer, if not NAPI. Otherwise, register poll() method with protocol stack
- The buffer is queued for processing by the networking protocols after the interrupt handler returns.

NIC driver receive path routines

- **dev_alloc_skb(length+NET_IP_ALIGN)** [pkt_size +2]: It is the buffer allocation function called by NIC driver in interrupt context. It is a wrapper around *alloc_skb*() that allocates 16 bytes of buffer and associate *sk_buff* to it. Additional two bytes are allocated for optimization reasons. These two bytes are then reserved by calling *skb_reserve(skb, NET_IP_ALIGN)*. 14 bytes for Ethernet frame and 2 bytes of padding. This keeps the IP header aligned at 16-byte boundary from the beginning of the buffer

***Figure showing how Ethernet frame is copied into the buffer**: a) before skb_reserve, b) after skb_reserve, c) ethernet frame copied into the buffer*



Net Device Method - Flow Control

- Flow control on received packets is applied at two levels by the protocols.
  - Maximum amount of data may be outstanding for *netif_rx()* to process.
  - Each socket on the system has a queue which limits the amount of pending data. Thus flow control is applied by the protocol layers
- Flow control on transmit packets is applied by network driver when the hardware has no transmit descriptors (part of device IO memory) available to process outgoing packets.
- Flow control routines:
  - **netif_start_queue()**: It is driver readiness to accept protocol data. It is called during device open() to ask the protocol layer to start transmit packets to the queue
  - **netif_stop_queue()**: This can be called when closing the device. Downstream flow control is accomplished by this funtion
  - **netif_wake_queue()**: When there is sufficient buffers, NIC driver call this routine to request protocol stack to restart queuing
  - **netif_queue_stopped()**: This is to check if the flow-control state.

You should always re-enabled the flow control using netif_wake_queue() instead of netif_start_queue. netif_wake_queue() do little more work. It checks whether anything in the queue is waiting to be transmitted considering there could be attempts to send the frame while the queue was disabled. netif_start_queue() is to enable the transmit queue when the device is first time activated. It can, however, be called at other times, when needed to restart a stopped device.

Net Device Method - stop

- Notify the protocol layer not to send any more packet to this interface via ***netif_stop_queue***().
- Disable device interrupts, transmission and receive DMA by writing to device IO memory region
- Free irq
- Free transmit and receive DMA buffers

Net Device Method – statistics

- net_device structure includes a pointer to "**priv**" field that is set by the driver to point to *net_device_stats* structure to collect network statistics common to all the network devices.
- The NIC driver populates a ***net_device_stats*** structure and provides a ***get_stats***() method to retrieve it.
- Driver Updates different types of statistics from relevant entry points such as interrupt handler.

```
#include <linux/netdevice.h>
struct net_device_stats mycard_stats;
static irqreturn_t mycard_interrupt(int irq, void *dev_id) { /* ... */
 if (packet_received_without_errors) {
        mycard_stats.rx_packets++;        /* update statistics */
  }
/* ... */
}
```

Net Device Method - statistics

- Implements the get_stats() method to retrieve the NIC statistics:
- To collect statistics from your NIC, trigger invocation of mycard_get_stats() by executing an appropriate user mode command.
- For example, to find the number of packets received through the eth0 interface, do this:

  bash> cat /sys/class/net/eth0/statistics/rx_packets

```
static struct net_device_stats *mycard_get_stats(struct net_device *netdev)
{
/* ... */
return(&mycard_stats);
 }
```