Chapter 1

In this chapter, I will briefly describe the journey of an incoming data packet, from its arrival to the Network Interface Card (NIC, from now on) to the moment it is delivered to the kernel networking subsystem. Specific kernel and driver mechanisms, implemented in Linux to achieve that goal, will be explained and documented, offering the reader a clear overview of the whole process.

The interrupt argument, one of the toughest in the fine art of the Operating system design, will be treated only when it comes to masking/unmasking operations, and the relative consequences, as well as the triggering of servicing routines. Specific subjects, like level-triggered vs. edge-triggered interrupts, for example, are unnecessary for the present understanding and, like many others, will not be mentioned.

For the same reasons explained above, the DMA (Direct Memory Access) argument will also be treated only conceptually.

DMA from NIC to RAM, and IRQ signaling

At the beginning, as a first step of the journey, the NIC receives a packet. The data contained therein must be transferred to RAM for the CPU access, and this happens usually by DMA (Direct Memory Access).

Accessing RAM through DMA means bypassing memory access controls operated by the kernel via CPU's MMU (Memory Management Unit). To avoid security problems, modern architectures interpose an IOMMU to confine the DMA I/O to well-controlled memory zones. For the sake of simplicity, the presence of an IOMMU will not be explicitly considered, but the DMA, from now on, will be intended as the "DMA safeguarded by the IOMMU".

The NIC sends (TX) and receives (RX) packets using vectors of I/O descriptors. Those descriptors, highly hardware dependent, are composed of structures containing at least physical RAM addresses, where incoming data is written and outgoing data is read, the size of those data areas, plus other custom fields and flags.
Some NIC specialized registers - the descriptors pointers - keep track of the first and the last available descriptor for receiving and sending the data. A vector containing descriptors is usually indicated as a ring, because each descriptor pointer has to jump back to the beginning of the vector itself when it reaches the end.
In many cases, TX-Rings and RX-Rings are also physically separated hardware structures, indexed by different descriptor pointers.

A few examples of NIC data I/0, based on the Intel E1000 network card design, are shown in the figure 1.1:

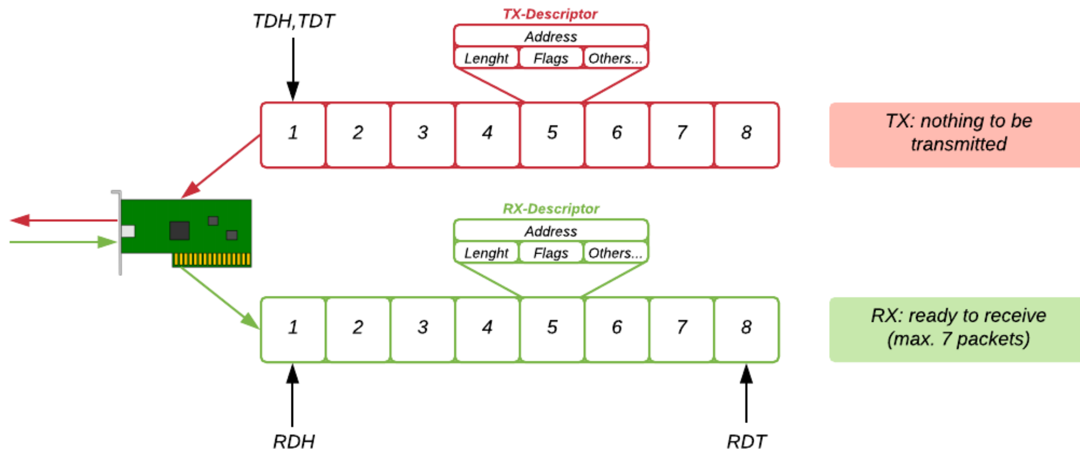

Figure 1-1

The NIC is ready for receiving a whole ring of data, while at the same time there are no data ready to be transmitted. Therefore, RDH points to the first available RX-Descriptor, and RDT to the first unavailable one. In other words, the RX-Descriptors starting at RDH and stopping before RDT ([1…7]) are usable for receiving data. Remember that RDH=RDT represents the stop position for the receiver, while TDH=TDT represents the stop position for the transmitter.

Note: The RX-Ring descriptors between RDH and RDT (excluded), and the TX-Ring descriptors between TDH and TDT (excluded), are indicated as 'owned by the hardware', whereas the remaining ones are indicated as 'owned by the software'. The OS 'produces' the descriptors, and the NIC 'consumes' them. Therefore, data transmitting/receiving, in essence, represents a typical producer/consumer problem.

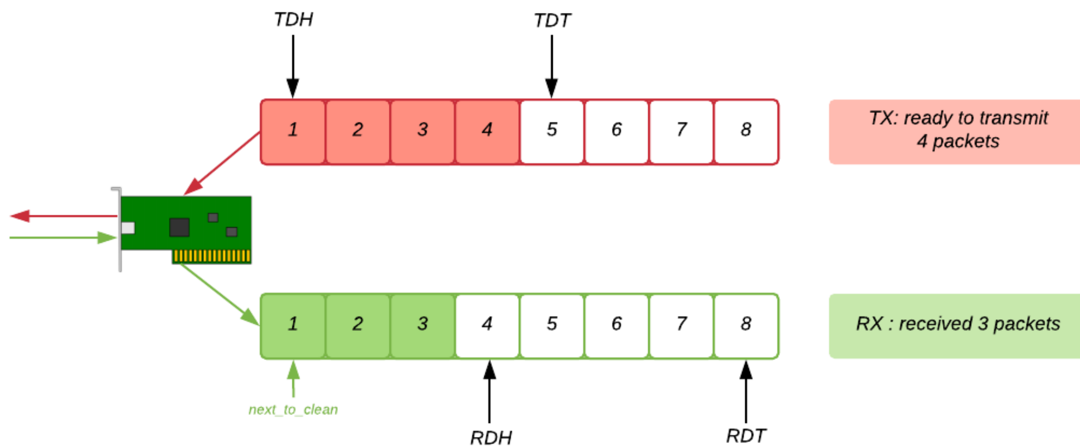

Figure 1-2

The NIC has received three data packets ([1..3]) and it still has enough free RX-Descriptors to receive up to other four packets ([4..7]). Nevertheless, it is ready to transmit four data packets ([1..4]), and while TDH points to the first next available TX-Descriptor, TDT points to the first unavailable one. In other words, the TX-Descriptors starting at TDH and stopping before TDT ([1..4]) are used for transmitting data.

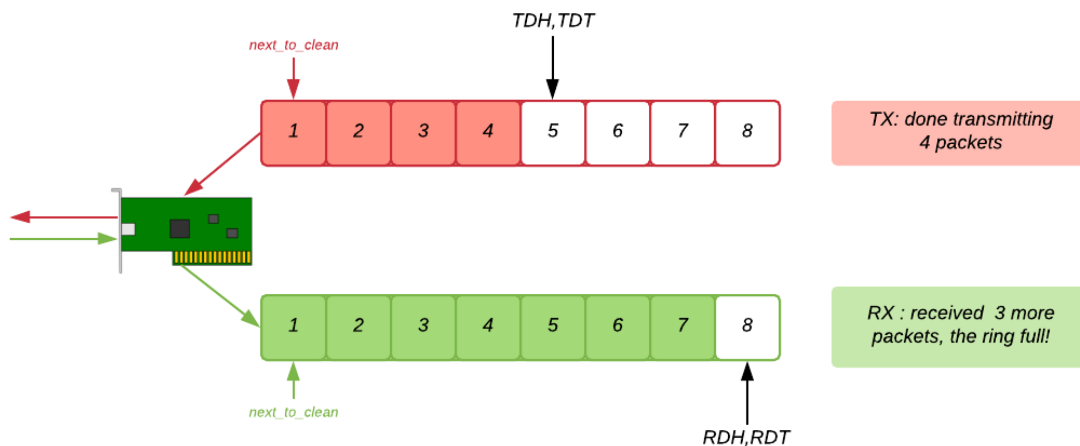

Figure 1-3

The NIC has received four more data packets ([4..7]) and it has exhausted the available RX-Descriptors. RDH=RDT represents the stop position for the receiver.
The NIC has also transmitted four data packets and it has no more data to transmit. TDH=TDT represents the stop position for the transmitter.

Note: In both cases, next_to_clean points to the first descriptor where the OS is supposed to begin cleaning up the ring.

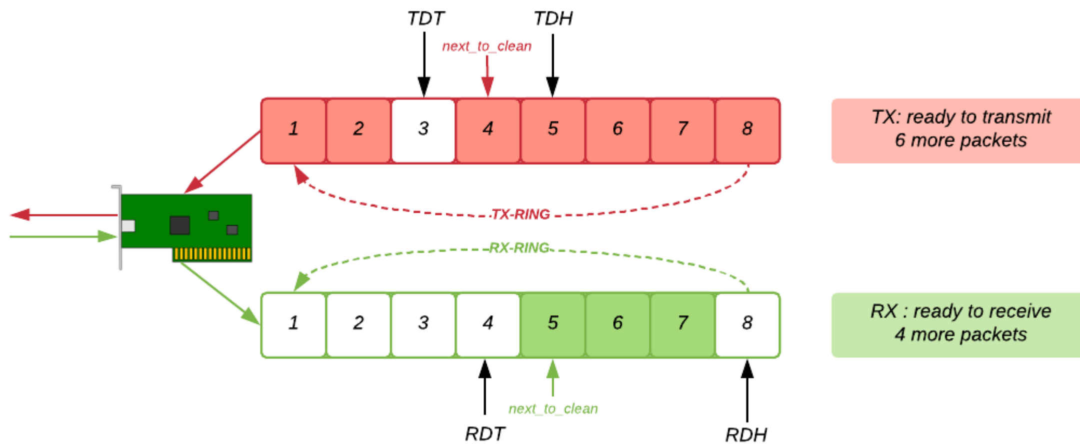

Figure 1-4

After the removal of four data packets ([1..4]) from the RX-Ring by the OS, the NIC is ready to receive four more data packets (8, [1..3]); note that RDT passed through the beginning, along the RX-ring.
After the removal of three data packets ([1..3]) from the TX-Ring by the OS, the NIC is ready to transmit six more data packets ([5..8], [1..2]); note that TDT passed through the beginning, along the TX-Ring.

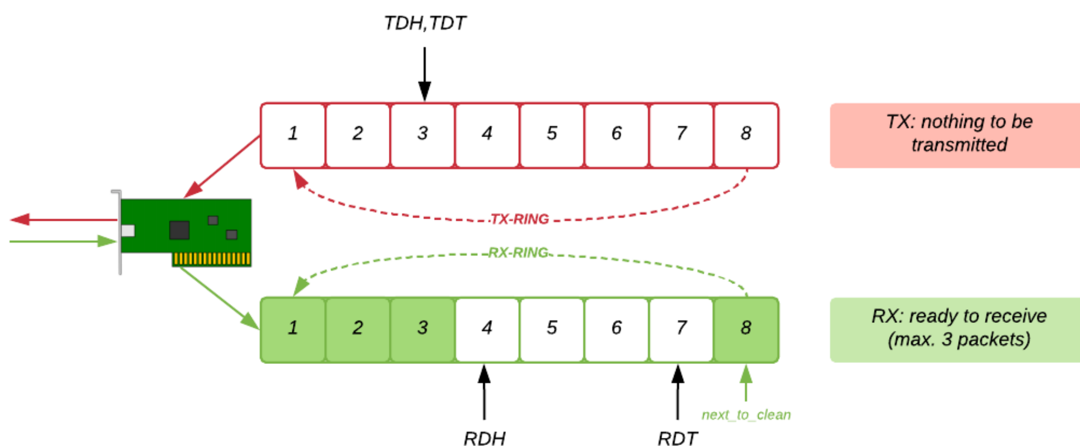

Figure 1-5

The NIC has just received four more data packets (8, [1..3]) and it still has enough free RX-Descriptors to receive up to other three packets ([4..6]), while there are no data ready to be transmitted.

The NIC informs the CPU of the reception of data (RX), or about the completion of a data transmission (TX), generating an interrupt, a signal that 'interrupts' what the CPU is doing, imposing a jump to an Interrupt Service Routine (ISR, from now on).

This is possible because every interrupt is associated with an Interrupt Request Line (IRQ, from now on), a hardware signal sent to the processor, coupled with a 'line number' used by the CPU as an index into a vector of pointers to the corresponding ISRs.

In Linux, when an ISR is executing, the corresponding IRQ is usually masked by the kernel, for every CPU, and put in a pending state until the ISR completes.

Masking an IRQ for a CPU means preventing the reception of that specific signal by the CPU itself. A CPU can usually mask its reception of every interrupt directly, for example via the '*cli'* instruction on Intel architecture, or indirectly, by programming a Peripheral Interrupt Controller (PIC, from now on). This second option is more flexible, because it allows us to choose which IRQs have to be blocked and which ones not.

Disabling the IRQ generation by a device means programming the device to prevent the generation of the corresponding signal when the triggering event happens, in our examples when a data packet is received by the NIC.

If multiple devices are sharing the same IRQ, disabling the IRQ generation on one of them, but not masking the IRQ for the CPU, means allowing all the devices but the one with the IRQ generation disabled to trigger the ISR that corresponds to the IRQ, when needed.

Who wants an IRQ?

A question that arises with a multiprocessor system is "which processor an IRQ has to be sent to?"

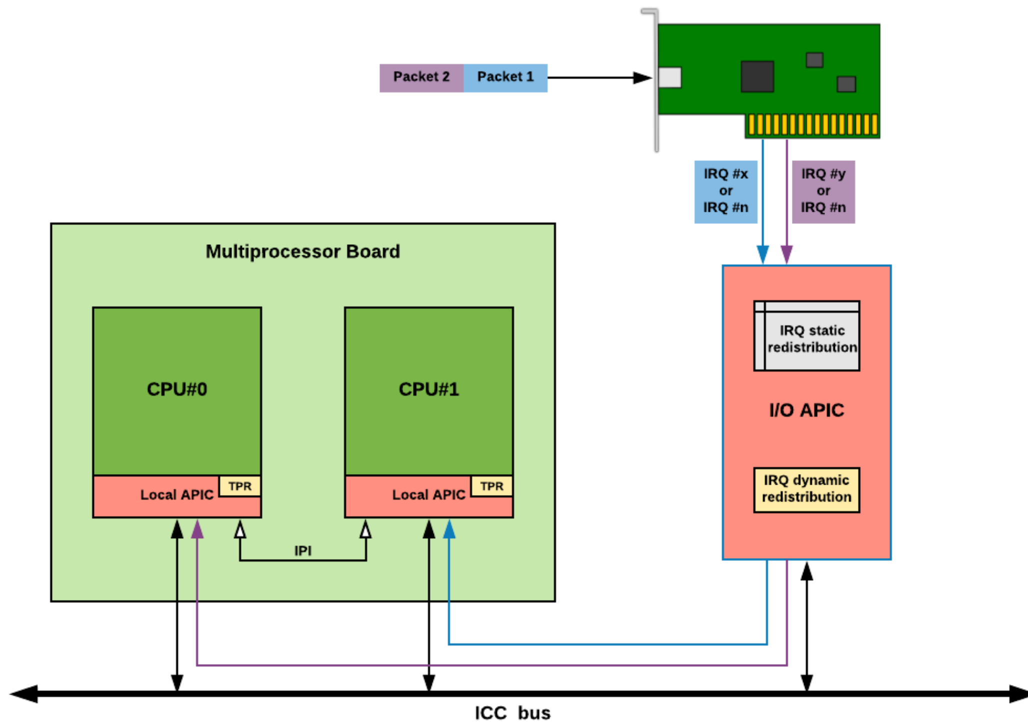

Figure 1-6

Note: Some modern network NICs are smart enough to split the data traffic into multiple hardware queues and generate different IRQs for every one of them. Other, less modern and performant, have only one hardware queue, and thus generate a single interrupt. The above figure is a synthesis of both cases.

Imagine having two packets in a row received by the RX ring of an NIC that generates, therefore, two IRQ signals. The question is which processor each IRQ, IRQ #x and IRQ #y, or multiple instances of the same IRQ #n, should be delivered to.
The optimal case would be sending IRQ#x to one CPU and IRQ#y to the other, and to do the same for every instance of IRQ#n, obtaining a highly parallelized processing.

Leveraging the presence of an Advanced Programmable Interrupt Controller (APIC, from now on), modern architectures are able to redistribute IRQ signals to different CPUs in two ways:

- *Static redistribution*: the OS programs a specific redistribution table that dictates to which CPU's local APIC (or a group of them) the IRQ signal must be delivered by the I/O APIC
- *Dynamic redistribution*: the OS keeps track of the actual running process priority via a special CPU's local APIC Task Priority Register (TPR). The I/O APIC delivers the IRQ to the local APIC of the CPU with the TPR containing the lowest priority. If two or more CPUs share the lowest priority, a Round-Robin like arbitration is used.

The CPU's local APIC and the I/O APIC are connected via an Interrupt Controller Communication (ICC) bus.

Furthermore, in a multiprocessor system, the OS can leverage the Inter-Processor Interrupts (IPIs) capability to exchange messages among CPUs. This characteristic is useful for balancing the IRQ servicing and, consequently, to increase the performance of modern operating systems.

Note: The present I/O APIC topics are based on the Intel CPU and chipset design.

Top-Half/Bottom-Half strategy for IRQ management

Up to now we are aware of the following:

- When a packet arrives to a NIC, it is DMA transferred to a RAM location via an OS programmed RX-ring descriptor pointed by a specialized RX-register
- The NIC then generates an IRQ, delivered to a CPU chosen by the I/O APIC
- The chosen CPU will be forced to execute a specific ISR routine, which is a part of the kernel
- The triggered IRQ is masked by the kernel, on every CPU during the ISR execution.

A question that comes to the mind is: what happens if multiple different devices share the same interrupt line? Specifically, what happens when the same IRQ is generated by different devices managed by different device drivers?

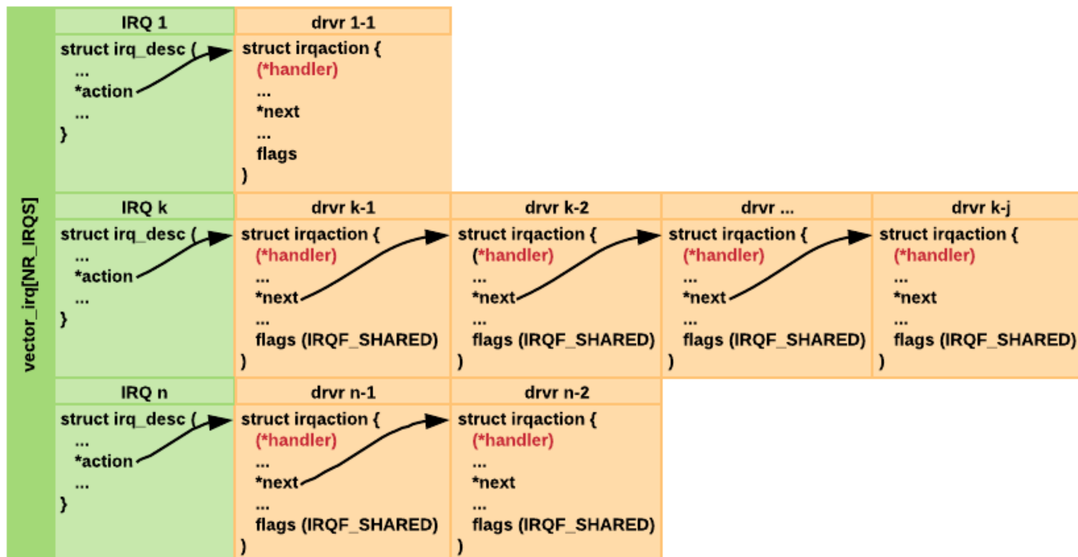

Figure 1-7

For every available IRQ number the kernel maintains an interrupt descriptor pointing to a list of associated interrupt actions containing, in turn, a handler pointing to the driver's ISR.

When only one physical device is associated with an IRQ number, the corresponding interrupt descriptor points to a list of interrupt actions containing a single element, and a single handler pointing to the driver's ISR is called, when the IRQ is received.

When multiple physical devices are sharing the same IRQ number, the corresponding interrupt descriptor points to a list of interrupt actions containing multiple elements, and every associated handler pointing to a driver ISR is called, in sequence, when the IRQ is received. In this case, it is up to every single driver ISR to understand, usually by looking at the specific HW resources (registers, memory location, I/O ports…), if data are available from the device, to collect them in case they are, and notify the kernel about the data availability or unavailability.

Executing the driver ISR could be a time-consuming operation, and during this time the serviced IRQ remains masked for every CPU. To avoid data loss, it is very important to be fast in servicing the IRQ and unmasking it as soon as possible.

Note: In this chapter, from now on, we will focus on the NIC data reception instead of data sending; the most important reason being the asynchronous nature of the data reception, which makes some operations more interesting and challenging.

A practical way of processing the network traffic could be when the NIC generates, and the CPU services, one IRQ for every data packet received. Under heavy load, perhaps, this strategy is highly unscalable and inefficient, because of the outstanding number of the received interrupts. On the other hand, totally coalescing the generation of interrupts through a polling strategy could be equally dangerous because, under variable load, the CPU can be both wasting time doing essentially nothing but waiting for the data that have yet to arrive, or doing something else and losing the data due to the RX-ring overflow when they arrive in an unexpected moment.
A more suitable way of mitigating the number of interrupts generated by the arrival of big storms of packets, and optimizing the CPU usage for collecting them, is combining together the IRQ servicing and data polling strategies.

The Linux solution, refined through the so called NAPI (New API), is to divide the device driver ISR in two parts, the top-half handler, trigged by hardware via NIC generated IRQs, and the bottom-half handler, triggered by software via a kernel interrupt deferring mechanism named SOFTIRQ, that will be explained later in more detail.

The NAPI represents, in essence, a kernel bottom-half handling mechanism to which devices are assigned to be scheduled for running the deferred driver's data polling routines.
A key point to keep in mind is that those driver's data polling routines will be run, when the data are available, with IRQs unmasked and are, therefore, interruptible by IRQs.

The Linux kernel ensures however that an instance of a device driver ISR and an instance of the driver polling function will never be active on more than one processor at a time. This strategy greatly reduces the complexity in writing the device drivers code.

NET_DEVICE and SK_BUFF, the networking pillars

A net_device structure contains information about a network device, either real or virtual.

A sk_buff structure contains pointers to network data, metadata and feature-specific fields.

Together, net_device and sk_buff are the pillars of Linux networking.

For the sake of simplicity, regarding both the net_device and the sk_buff structures, I will comment only on the fields that are relevant for the topics I intend to present.

struct net_device

- **name** – name of the associated interface (e.g. lo0, ppp0, eth0, ens33…)
- **dev_list** – global list of network devices
- **napi_list** – global list of napi_struct object
- **adj_list** – direct linked devices
  - the **upper** field points to a list of upper level devices (e.g. bridge for a bridge port)
  - the **lower** field points to a list of lower level devices (e.g. bridge ports for a bridge)
- **mem_start** – shared memory start
- **mem_end** – shared memory end
- **base_addr** – device I/O address
- **irq** – device IRQ number
- **ifindex** – interface index
- **group** – group the device belongs to; useful for bonding, load balancing...
- **flags** – flags visible from userspace (e.g. IFF_UP, IFF_PROMISC, IFF_MACVLAN, IFF_POINTTOPOINT…)
- **mtu** – interface MTU value
- **netdev_ops** – structure containing pointers to several callback methods to be defined for changing the default behavior of a network device
- **(*rx_handler)** – function called from inside the __netif_receive_skb() kernel function, to do special processing of the skb prior to its delivery to protocol handlers; <u>it is a cornerstone function for writing the bridging software</u>
- **rx_handler_area** – pointer to a memory area allocable to support the rx_handler functionalities
- **nd_net** – network namespace this network device is inside

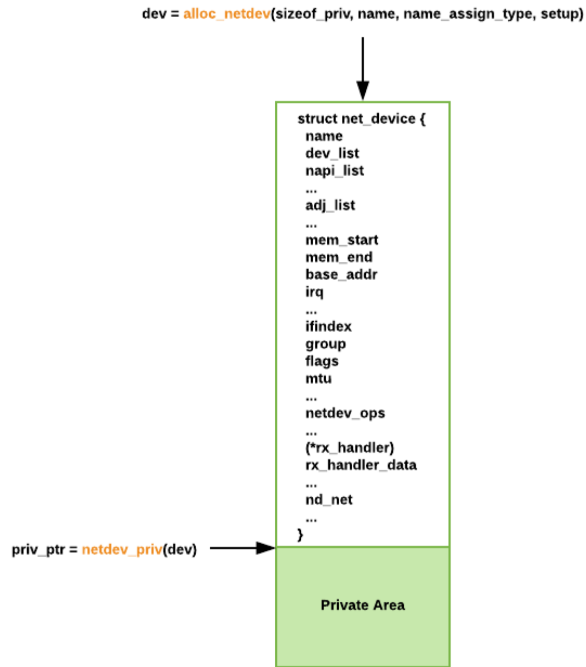

Figure 1-8

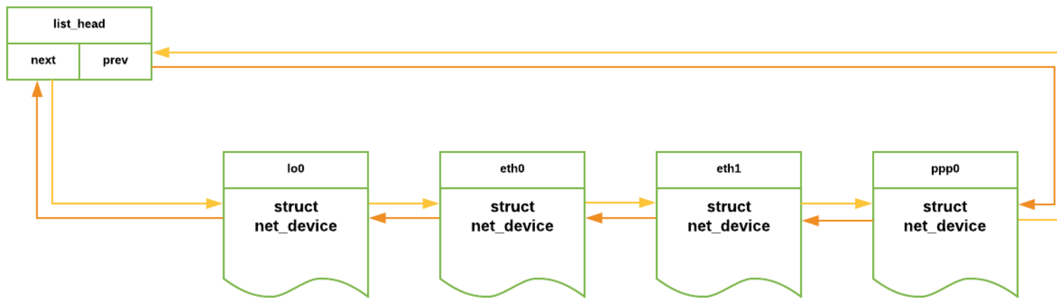

Figure 1-9

struct sk_buff

- **next** – next buffer in list
- **prev** – previous buffer in list
- **dev** – device the data arrived on/are leaving by
- **sk** – socket the skb is owned by
- **cb** – control buffer, free for use by every layer; put private vars here
- **len** – size of the data in the buffer (when skb is linear, that means not fragmented)
- **skb_iif** – ifindex of the device the packet arrived on
- **protocol** – the next-level protocol from the driver's perspective
- **transport_header** – transport layer header (L4)
- **network_header** – network layer header (L3)
- **mac_header** – link layer header (L2)
- **tail** – end of data pointer
- **end** – end of buffer pointer
- **head** – start of buffer pointer
- **data** – start of data pointer

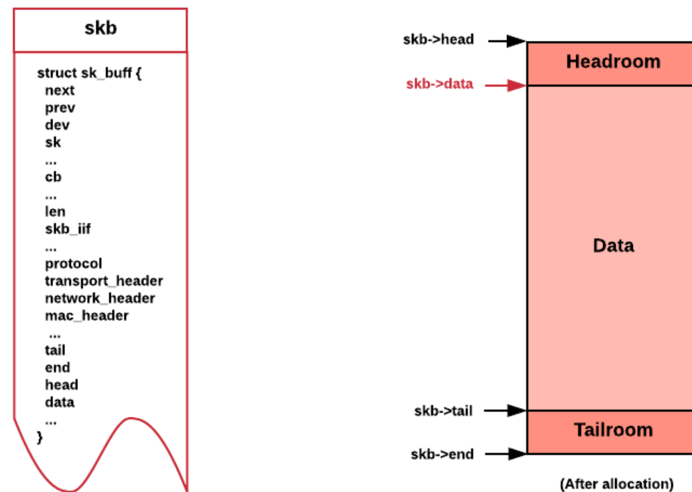

Figure 1-10

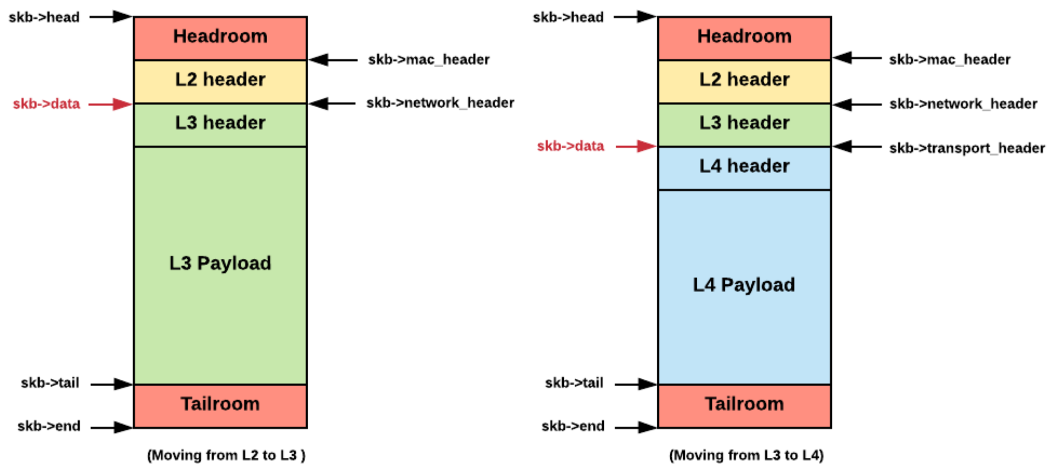

Figure 1-11

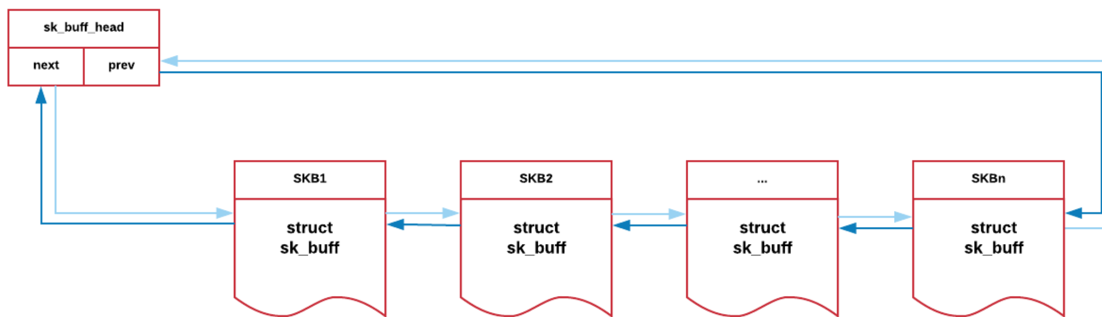

Figure 1-12

Some considerations on NET_DEVICE and SK_BUFF

As seen before, the net_device structure represents a network device inside the kernel, while the sk_buff structure represents the data packets traversing the networking layers.

Some considerations about both of them:

- The kernel maintains lists of net_device and sk_buff structures for internal use
- The net_device permits, during its allocation, to define a private area where device driver developers can store driver's custom structures
- The net_device knows about its hierarchy, in simple words, if the associated device is a master, like a bridge, or a slave, like a bridge port
- The net device maintains info about the namespace it is part of
- <u>The net_device, through the function pointer rx_handler and the data pointer rx_handler_data, permits to a bridge module developer to intercept and manage packets just after they are polled out from the device ring</u>
- The sk_buff knows which network devices it is coming from or it is destined to, and even which socket it is owned by
- The sk_buff data pointers architecture allows moving the packets through the networking layers without copying data
- The sk_buff data pointers architecture allows cloning the packets simply by creating new sk_buff structures pointing to the same data
- The sk_buff pointed Headroom and Tailroom data regions must be pre-allocated and are reserved for special purpose, like the packet encapsulation, avoiding copying data in bigger areas in such cases
- <u>The sk_buff contains a private area, cb (control buffer), for storing private information maintained by each networking layer for internal use.</u>

Top-Half/Bottom-Half IRQ Handlers operations

In Linux, the top-half IRQ handlers, sometimes indicated as hard-IRQ handlers, and the bottom-half IRQ handlers, sometimes indicated as deferred-IRQ handlers, are both running (the bottom-half possibly running, to be precise, as we will see later) in the interrupt context. The interrupt context is very limitative, it is not associated with a process, it cannot be rescheduled and, consequently, no function running there can sleep. Everything running in this context has to be fast and bugs free, device driver code at first.

- **Top-Half driver's IRQ Handler**
  The driver's IRQ servicing routine, irq_handler() in my examples, which is the top-half IRQ handler and therefore it runs with NIC IRQ masked on the CPU(s), accomplishes the following macro-steps:
  - In a standard NAPI-driver implementation, the most usual ones and interesting for this document:
    - Disables the NIC's IRQ generation
    - Defers the execution of the driver's poll() function, the one described in the bottom-half IRQ handles section, invoking the system call **__napi_schedule()**
    - Gives the control back to the kernel ISR, completing the top-half IRQ handler execution
  - In a mixed NAPI-driver implementation, where standard interrupt coalescing techniques are used together with the NAPI ones:
    - Disables the NIC's IRQ generation
    - Executes immediately the driver's poll() function; if the data burst is small enough not to exceed the *budget* (see bottom-half handler), a single IRQ generation and a one-run execution of the polling function are sufficient to collect all the available data, without the need of a bottom-half handler execution. This approach reduces the latency during small data bursts, still coalescing the NIC's IRQ generation.
    - If the budget has been exhausted by the previous driver's poll() function triggering, but data are still present in the NIC's RX-ring, a call to **__napi_schedule()** will defer the bottom-half handler execution
    - If no more data are present in the NIC, then it re-enables the NIC's IRQ generation
    - Gives the control back to the kernel ISR, completing the top-half IRQ handler execution.
      Note: Some NICs permit to program special IRQ generation methods, such as:
      - Generate interrupts after receiving n frames
      - Generate interrupts if no more frames are received after n µsecs.

Both of them can be leveraged for interrupt coalescing, at the cost of a higher latency.

- **Bottom-Half driver's IRQ Handler**
  The driver's polling function poll(), which is the driver's bottom-half IRQ handler and therefore runs with NIC's IRQ unmasked for the CPU(s) but with NIC IRQ generation disabled, accomplishes the following macro-steps:
  - Polls the RX-ring up to a predefined maximum number of RX-descriptors (*the budget*) and, for every one of them:
    - Unmaps from DMA the received data area, addressed by the RX-descriptor, and replaces it with a newly allocated one
    - Maps for DMA the newly allocated data area and addresses it by updating the RX-descriptor, completing in this way the actual descriptor replacement
    - Allocates a new skb for the received data
    - Initializes the skb from an L2 perspective (removing L1 data, setting the upper protocol metadata…)
    - Execs a system call, e.g. **napi_gro_receive()**, for passing the skb up to the network stack; this system call triggers **__netif_receive_skb()** that's the ingress frame processing function of the kernel's network management code
  - If all the available data have been collected, enables again the NIC's IRQ generation.

SOFTIRQs and Bottom-Half IRQ Handlers scheduling:

The Linux SOFTIRQs are a set of statically defined system bottom-halves that can run, even simultaneously, on multiple processors.
Every SOFTIRQ is associated with a priority and a corresponding action where, the lower is the number representing the priority, the higher will be the priority itself.
For sending packets, the NET_TX_SOFTIRQ (priority 2) has been defined, whereas for receiving packets, the NET_RX_SOFTIRQ (priority 3) has been defined.
The SOFTIRQ scheduling is based on priority, the higher priority SOFTIRQs are scheduled before the lower priority ones.
The SOFTIRQs never preempt eachother, the IRQ-triggered ISRs instead preempt the SOFTIRQs.

To manage the registered SOFTIRQs, Linux maintains an ordered by priority vector of softirq_action structures, softirq_vec.

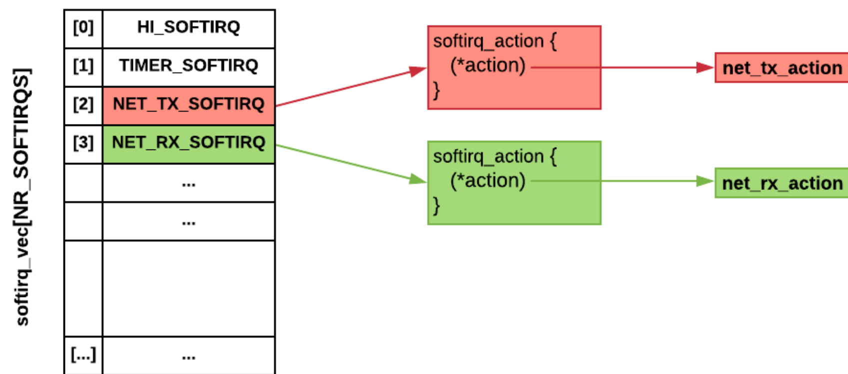

Figure 1-13

NET_TX_SOFTIRQ is managed by the kernel function **net_tx_action**, pointed by the corresponding softirq_action's action field of softirq_vec[NET_TX_SOFTIRQ]. NET_RX_SOFTIRQ is managed by the kernel function **net_rx_action**, pointed by the corresponding softirq_action's action field of softirq_vec[NET_RX_SOFTIRQ].

Linux maintains a per-CPU data structure, **softnet_data**, to support the SOFTIRQs NET_TX_SOFTIRQ and the NET_RX_SOFTIRQ. Since softnet_data is maintained per-CPU, the kernel doesn't need to use expensive inter-CPU locking mechanisms to manage its content.

The softnet_data structure contains a pointer to a queueing discipline, **output_queue**, representing devices with data ready to be sent by **net_tx_action**.

The softnet_data structure also contains the list **poll_list**, a list of napi_struct structures representing devices with data waiting to be read by the kernel function **net_rx_action**.

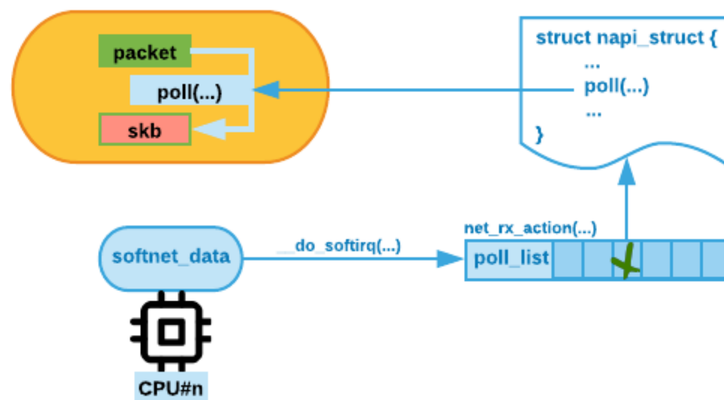

Figure 1-14

Every one of the napi_struct structures linked in the poll_list contains a pointer to the respective NAPI-registered driver's **poll()** function that is used by the kernel function net_rx_action to poll the NIC's RX-Ring, as we will see later in more detail.

At this point almost every ingredient needed to run our bottom-half IRQ handler, and consequently read the packets present into the NIC's RX-ring, is available:

- We know the SOFTIRQ to use - NET_RX_SOFTIRQ - and the associated action function - net_rx_action
- We have the per-CPU softnet_data structure that contains the list of devices with data waiting to be read - the poll_list
- We have, inside the poll_list, the napi_struct corresponding to the device, whose pointer to its driver's NAPI poll() function will be used by net_rx_action to read the available data.

So, what is missing?

Only the most important thing, that is, the spark that fires the SOFTIRQ scheduling.

As you certainly remember, the top-half IRQ handler is fired by an IRQ, and therefore it runs inside an IRQ context with the respective IRQ line disabled. However, it is exactly because we do not want to read every packet from the NIC in such a context, moreover with disabled IRQs, that we have split the IRQ management routine in two parts.

Therefore, to trigger the bottom-half IRQ handler via the SOFTIRQ mechanism, we need something different than an IRQ, and that something is a call to the kernel function **__do_softirq()**.

But who calls __do_softirq, when and in which context?

The kernel function __do_softirq is called, by the kernel itself, essentially in the following circumstances:

- When the kernel function **irq_exit()** is called, in the exiting part of the function do_IRQ(), that is the kernel's generic IRQ handler. That means, in our examples, every time an IRQ's top-half handler has been executed. In this case __do_softirq runs in the Interrupt Context.
  Note: To be more precise, the Linux kernel invokes __do_softirq() only in the exiting part of a non-nested IRQ. In simple words, a nested IRQ, not supported by every platform, is an IRQ that interrupts the ISR of another IRQ.
- When the kernel schedules the kernel thread ksoftirqd/#cpu. In this case __do_softirq runs in the Kernel Context.

Note: SOFTIRQs are evaluated even in some other parts of the kernel code, such as when the kernel function local_bh_enable is called, when the IO/APIC's local timer interrupt servicing routine smp_apic_timer_interrupt is about to exit, during some Inter Process Interrupts (IPI) management, and in other different places that sometimes change due to the continuous kernel optimization process.

The __do_softirq() kernel function, essentially, executes the actions associated to every pending SOFTIRQ in order of priority. Since those actions are interruptible by IRQs, and an IRQ can reschedule a running action related to itself, under heavy load it is possible that the running actions could stay running for a very long time, delaying unacceptably the user-mode processes. For that reason __do_softirq() returns after a fixed amount of interactions or time, even if there is still some unfinished work. In this case, however, to complete the SOFTIRQ's remaining work, before exiting __do_softirq() schedules, on the running CPU, the execution of the ksoftirqd/#cpu kernel thread. The ksoftirqd thread, if any SOFTIRQs are pending, calls do_softirq() to serve them. In this case the duration of the ksoftirqd thread responds to the logic of the scheduler, and the advantage of having a per-CPU ksoftirqd process is that, in this way, an idle processor can service SOFTIQs very often.

Graphically supported recap

If we consider our NIC device driver a NAPI-compliant one, to sum up, the following steps are taking place while receiving data:

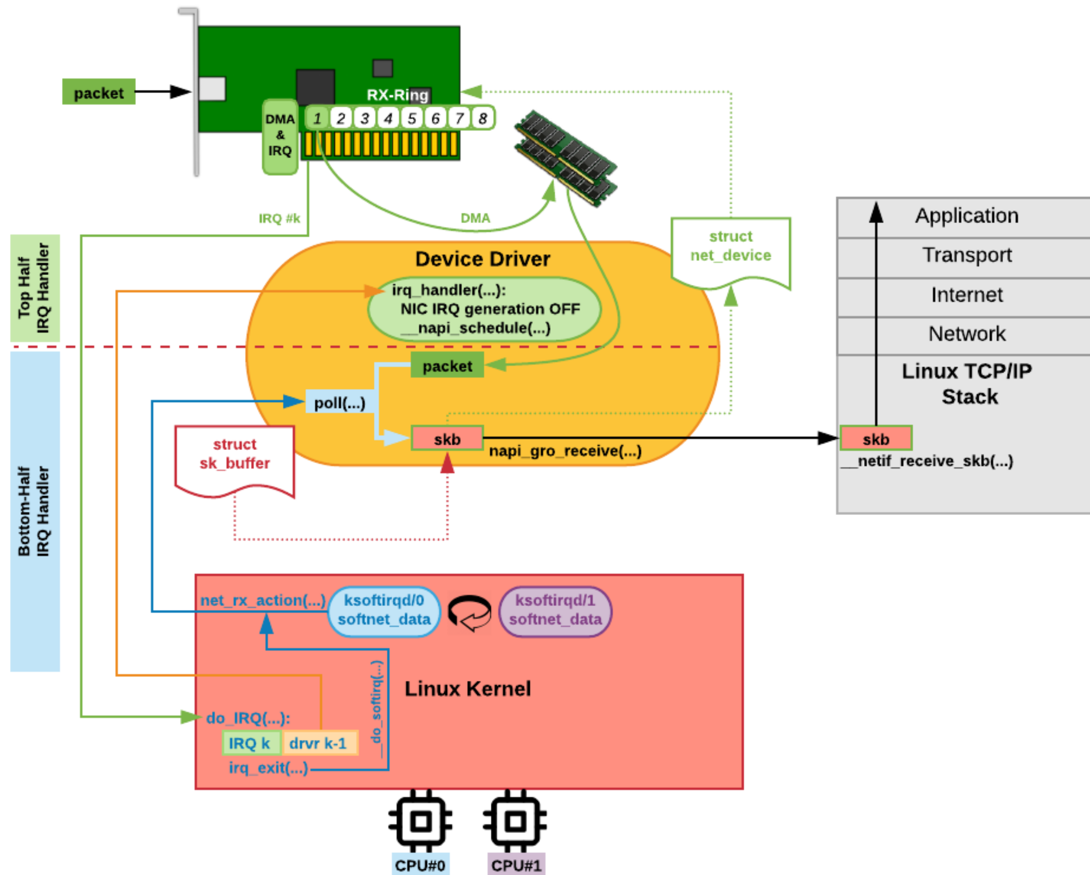

Figure 1-15

- The incoming packet enters the NIC
- An RX-Ring Descriptor is used to DMA-copy the packet to RAM
- An IRQ is generated by the NIC
- The kernel's generic IRQ handler do_IRQ() is called
- **do_IRQ()** triggers the top-half IRQ handler calling the driver's IRQ management routine, **irq_handler** in our example, using the handler referred by the action which is in turn pointed by the interrupt descriptor associated with the NIC's IRQ
- The exiting part of the kernel function do_IRQ calls the kernel function irq_exit
- The kernel function **irq_exit**, if the kernel is leaving the interrupt context and there are pending SOFTIRQs for the actual CPU, triggers the kernel function __do_softirq

- The kernel function **__do_softirq** execs, when higher priority SOFTIRQS are done, the network bottom-half IRQ handler via a SOFTIRQ of type NET_RX_SOFTIRQ, using the kernel function net_rx_action
- The kernel function **net_rx_action** by leveraging the list of drivers with pending data present in the per-CPU softnet_data - the poll_list - calls the driver's NAPI-registered polling function poll().
  The driver's poll() function removes the packets from the NIC, associates them with the respective skbs, as described in more detail in the next section, and calls the NAPI function **napi_gro_receive** that, in turn, calls __netif_receive_skb.
  Note: The Linux Generic Receive Offload (GRO, from now on), together with the counterpart Generic Send Offload (GSO, from now on), are two mechanisms which, essentially, aggregate many packets having strict commonalities but small MTU (i.e. 1500 bytes) in fewer larger packets having bigger MTU (i.e. 9000 bytes). The advantages are a better usage of the wire, in the case of GSO, and fewer packets to be managed by the kernel's networking stack in the case of GRO. The NAPI function napi_gro_receive, called by poll() in the above description, is a part of the GRO optimized family.

- The kernel function **__netif_receive_skb** represents the joint between the device driver and the Linux networking code; from this moment on, the life of our packet continues outside device driver's realm.

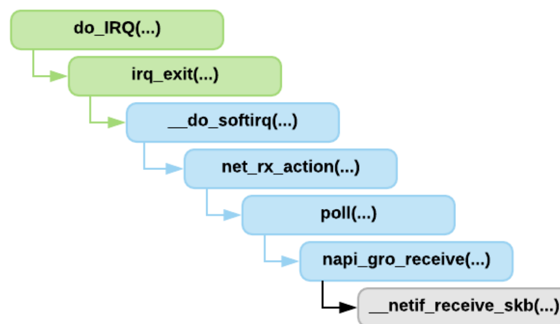

Figure 1-16

NAPI polling function in more detail

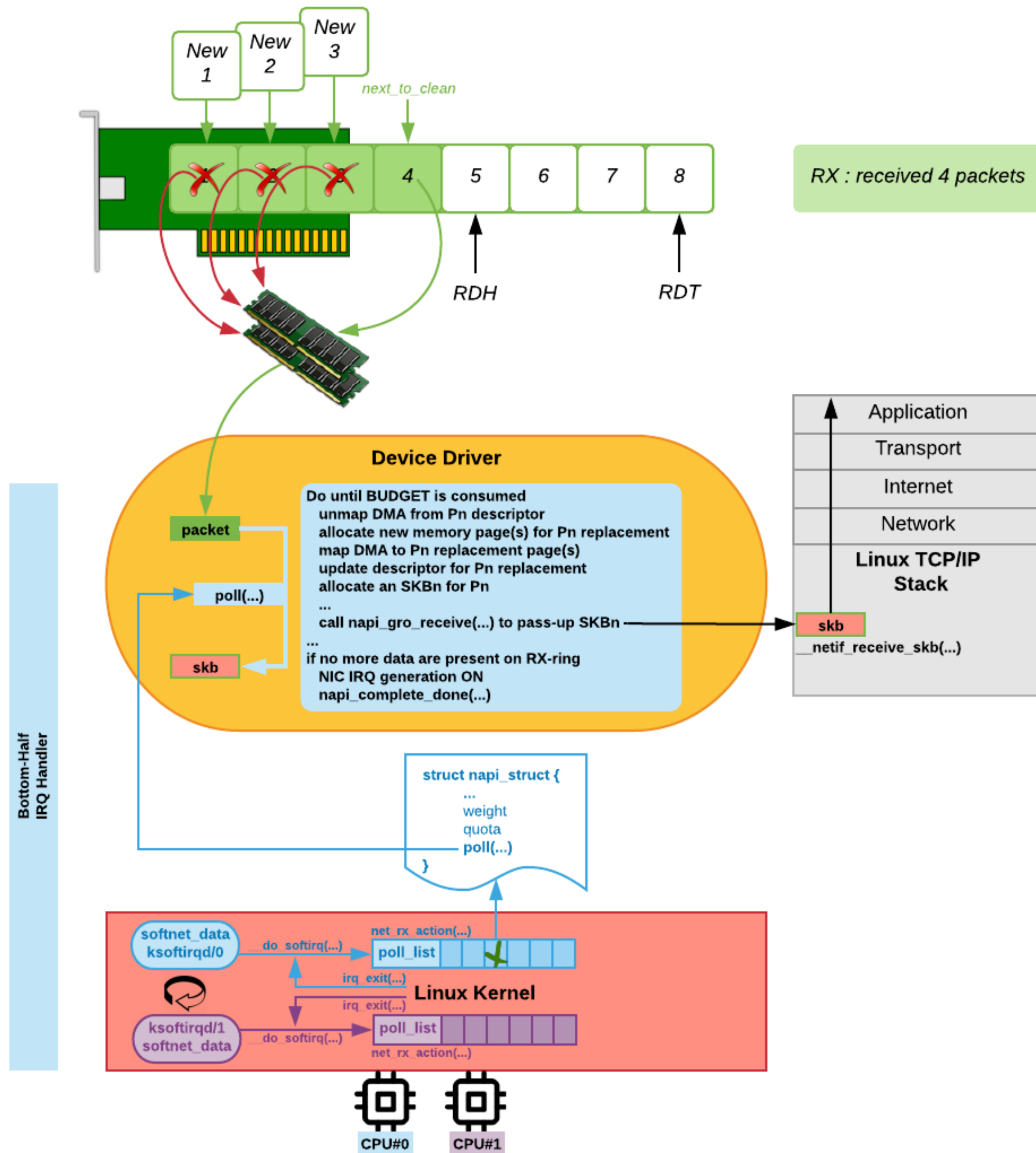

Figure 1-17

The driver's poll() function is the engine of the NET_RX_SOFTIRQ's operations. It's up to the device driver's designer to implement it in the optimal way. This function receives two parameters, a pointer to a structure of type napi_struct and a budget. The first parameter is mainly used for accessing a driver's private structure, containing all

the specific fields needed for managing the NIC. The budget, on the other hand, represents a maximum number of packets that a single run of poll() is allowed to manage.

There are many complex situations that could occur when mapping data packets to RAM and associating SKBs to them. Therefore, to simplify the description, as it is reasonable for our purpose, we assume that:

- The physical memory page size is 4KB
- The incoming packets are not fragmented
- The kernel does not split a packet among physical memory pages.

This simplification allows us not to deal with topics like scatter-gather lists and non-linear SKBs, nevertheless, the following explanation is detailed enough to describe, conceptually and practically, a real situation, even if optimal.

Under these conditions, the poll() function does the following:

- For each RX-Descriptor containing data, until the budget is reached or the RX-descriptors containing data have been used up:
    - Unmaps the actual RX-Descriptor's RAM area from DMA
    - If necessary, allocates a new RAM page as RX-Descriptor replacement new RAM area
    - Maps the RX-Descriptor replacement to the new RAM area
    - Allocates a new SKB for the actual descriptor's DMA-unmapped RAM area, and initializes it from an L2 perspective (removing L1 data, setting the upper protocol metadata…)
    - Passes the SKB up to the network stack via a kernel function call, e.g. napi_gro_receive(), as explained in a previous paragraph
    - Substitutes the actual RX-Descriptor with the replacement, mapping its RAM area for DMA
- If there are no more RX-Descriptors containing data:
    - Enables again the NIC's IRQ generation
    - Calls the NAPI function napi_complete_done() to remove the driver's associated napi_struct from the CPU's softnet_data contained poll_list.