Virtio networking: A case study of I/O paravirtualization

Ing. Vincenzo Maffione

1/12/2016

Outline

1. How NICs work
2. How Linux NIC drivers work
3. How NICs are emulated
4. Performance analysis of emulated e1000
5. I/O paravirtualization ideas
6. The VirtIO standard
7. The VirtIO network adapter (virtio-net)
8. Performance analysis of virtio-net

How NICs work (1)

Ethernet Network Interface Cards (NICs) are used to attach hosts to Ethernet Local Area Networks (LANs). NICs are deployed everywhere - laptops, PCs, high-end machines in data centers - and many vendors and models are available - e.g. Intel, Broadcom, Realtek, Qualcomm, Mellanox.

But how does a NIC work? How does the Operating System (OS) control it?

How NICs work (2)

All modern NICs are DMA-capable PCI devices, exposing a model-specific set of registers.

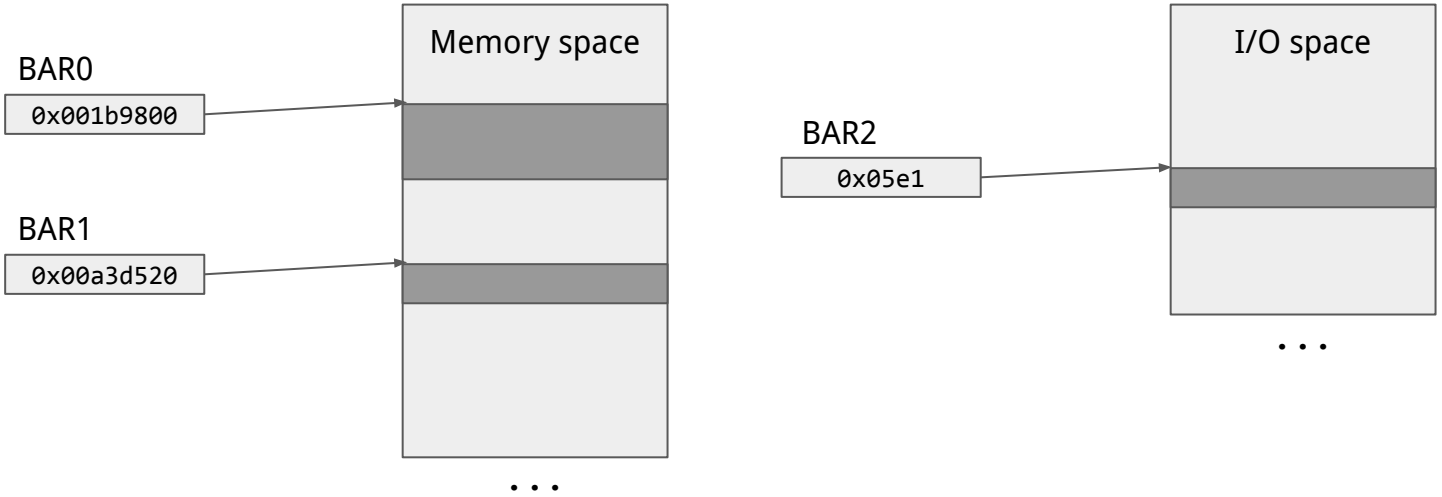
Most of the registers are memory-mapped, which means that x86 CPUs can access them with regular MOV instructions. Some registers can be mapped in the CPU I/O space, which means that they can be accessed with IN/OUT instructions on x86 CPUs. In x86, memory-mapped I/O is preferred over port I/O, because it is more flexible. IN/OUT instructions can only use DL or immediate operand as I/O port, and EAX/AX/AL as value.

Direct Memory Access (DMA)* allows PCI devices to read (write) data from (to) memory without CPU intervention. This is a fundamental requirement for high performance devices.

(Very) Old devices - e.g. ISA, non-DMA capable - are not considered here.

(*) In the PCI standard DMA is also known as *Bus Mastering*

How NICs work (3)



A PCI device exposes one or more Base Address Registers (BARs) in its PCI configuration space. BARs are programmed by the OS with base addresses where the corresponding memory or I/O regions - containing PCI registers in the NIC case - will be available. OS can read the sizes of the device’s address regions.

e1000 (1)

The device's configuration space also exposes the Device ID, Vendor ID and other registers that are used by the Operating System (OS) to identify the device model.

We will take the Intel e1000 family of Gigabit network adapter as a NIC case study. In particular, we will consider the 82540EM model, commonly emulated by hypervisors (VMWare, VirtualBox, QEMU), which has Device ID 0x100E and Vendor ID 0x8086.

The 82540EM exposes a 128KB memory-mapped region (through `BAR0`) and a 64 bytes I/O mapped region (through `BAR1`). More than 300 32-bit registers are visible to the NIC programmer. Regular PCI interrupts are used by the NIC to inform the OS about asynchronous events.

Complete reference available at
http://www.intel.com/content/dam/doc/manual/pci-pci-x-family-gbe-controllers-software-dev-manual.pdf

e1000 (2)

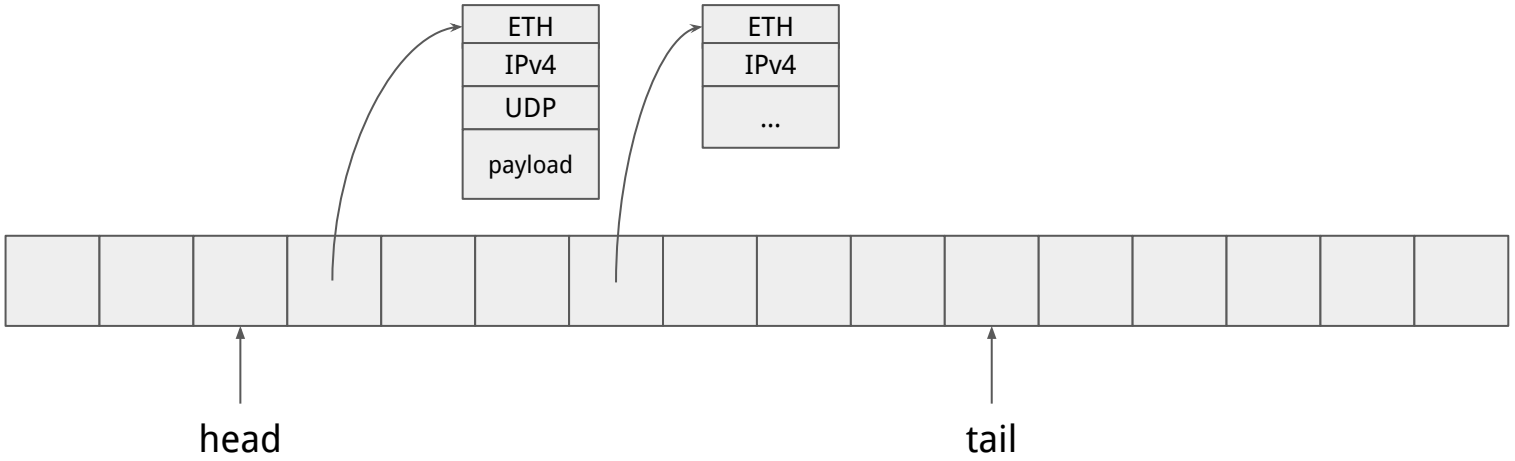
A lot of complexity of NICs comes from configuration: power management, PHY module, EEPROM, FLASH, and other advanced features and components. This explains the huge manuals (500-1000 pages).

Nevertheless, we will focus on the NIC *datapath*, which refers the software interface that the OS uses in order to transmit and receive Ethernet frames (packets, in the following). Datapath involves only a few registers and DMA-mapped memory.

While reported here on a specific example, these datapath basic concepts are actually common to all modern NICs, even if register names and other details are different.

e1000 (3)

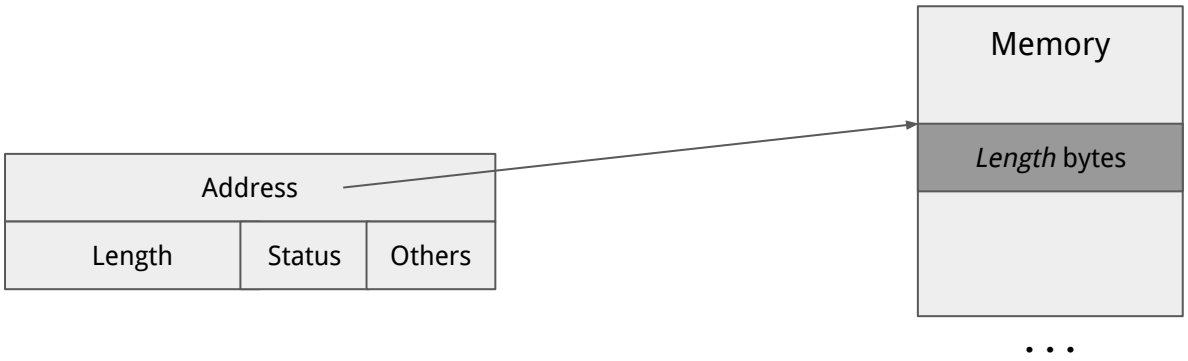
The OS exchanges packets with the NIC through the so called *rings*. A ring is a circular array of *descriptors* allocated by the OS in the system memory (RAM). Each descriptor contains information about a packet that has been received or that is going to be transmitted. The e1000 NIC uses a ring for transmission (*TX ring*) and another ring for reception (*RX ring*). The descriptor format is different, but (almost) all descriptors have at least the physical address and length of a (memory) buffer containing a packet. Note that physical addresses are used, not virtual ones, since the NIC accesses rings and packets through DMA.



e1000 transmission (1)

Each TX descriptor is 16 bytes long and contains

- The physical address of a buffer with the contents of a packet that is going to be sent or that has already been sent
- The length of the packet
- A status bit (Descriptor Done) indicating whether the packet is going to be sent or has already been sent
- Flags and other fields that can be used by the OS to specify TX options to the NIC

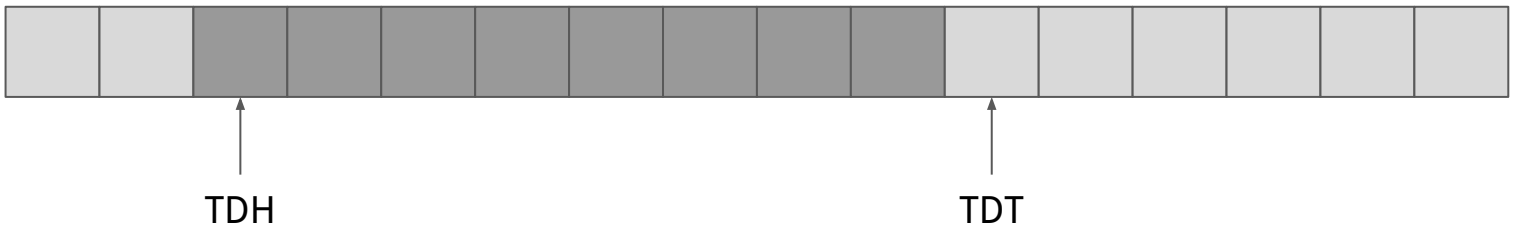


e1000 transmission (2)

Synchronization between the OS and the NIC happens through two registers, whose content is interpreted as an index in the TX ring:

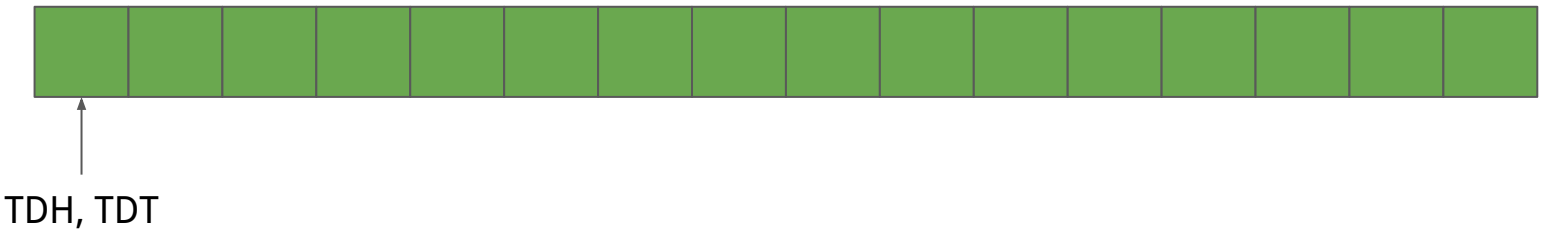
- Transmit Descriptor Head (TDH): indicates the first descriptor that has been prepared by the OS and has to be transmitted on the wire.
- Transmit Descriptor Tail (TDT): indicates the position to stop transmission, i.e. the first descriptor that is not ready to be transmitted, and that will be the next to be prepared.

Descriptors between TDH (incl.) and TDT (excl.) are owned by the hardware, ready to be transmitted by the NIC. The remaining ones are owned by the software, ready to be used by the OS to request new transmission.



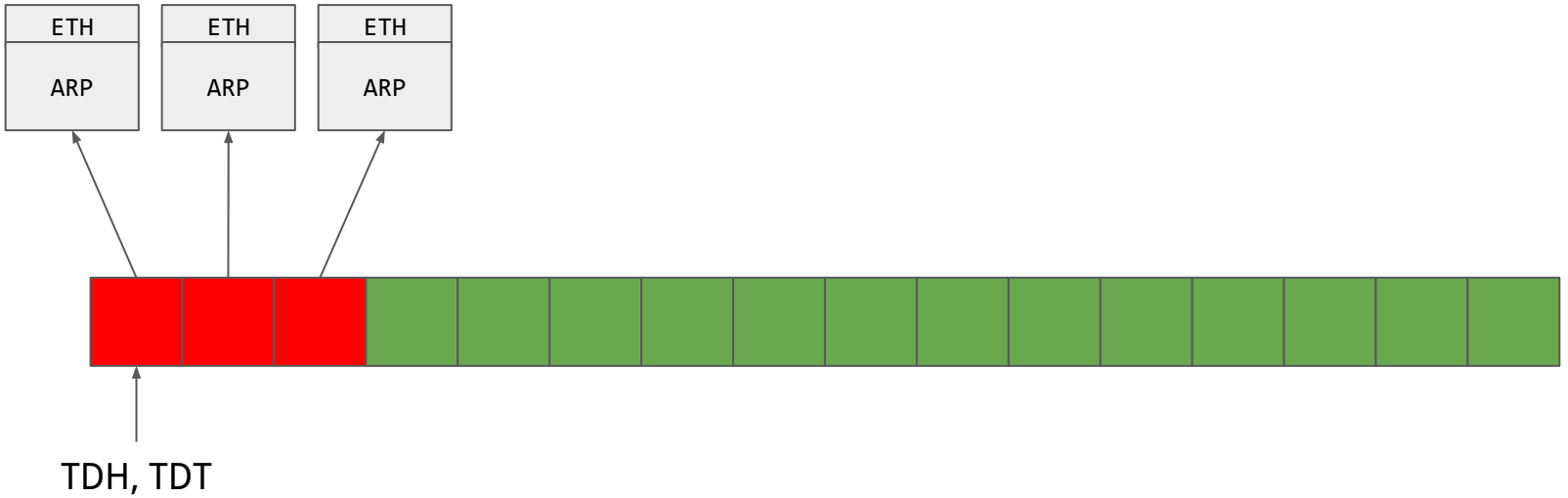
e1000 transmission (3)

At the start-up, the OS initializes TDT and TDH at zero. In this situation, the TX ring is completely free, since all the descriptors are available for the OS to be used for transmission. Consequently, there are no pending descriptors to be processed (i.e. no packets waiting in the ring to be transmitted by the NIC).



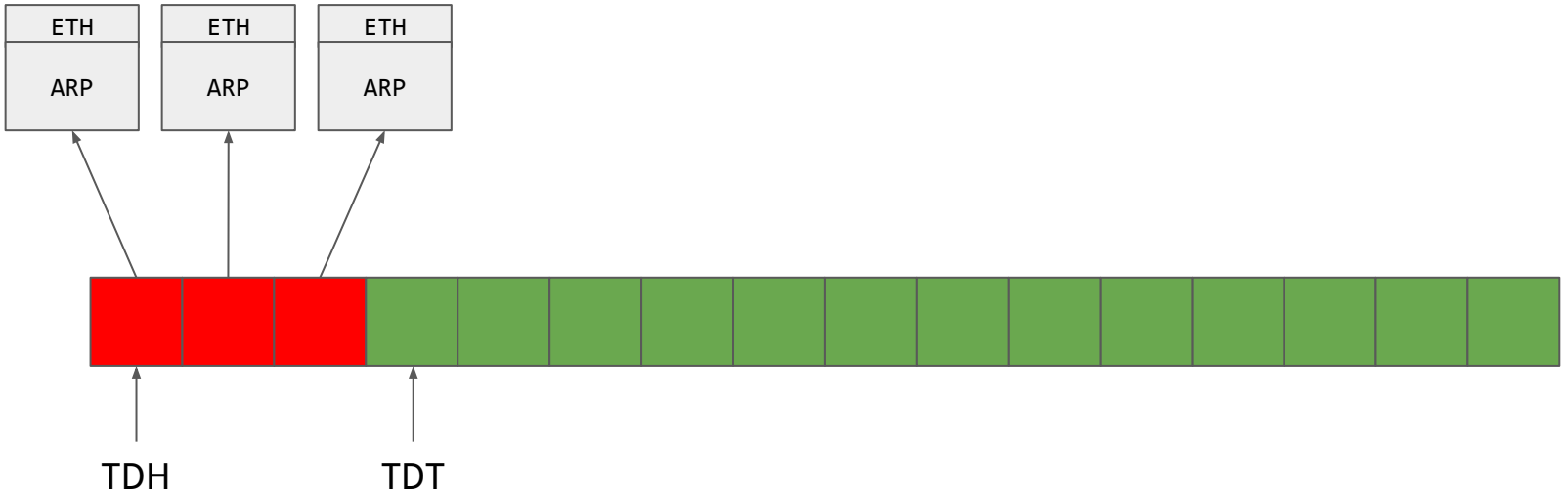
e1000 transmission (4)

Let’s assume the OS wants to transmit 3 packets at once (e.g. three ARP requests). The OS will first prepare (*produce*) three consecutive descriptors, starting from the first available one, which corresponds to the current TDT value. The prepared descriptors will contain packet physical addresses and packet lengths, with the status bit set to zero.



e1000 transmission (5)

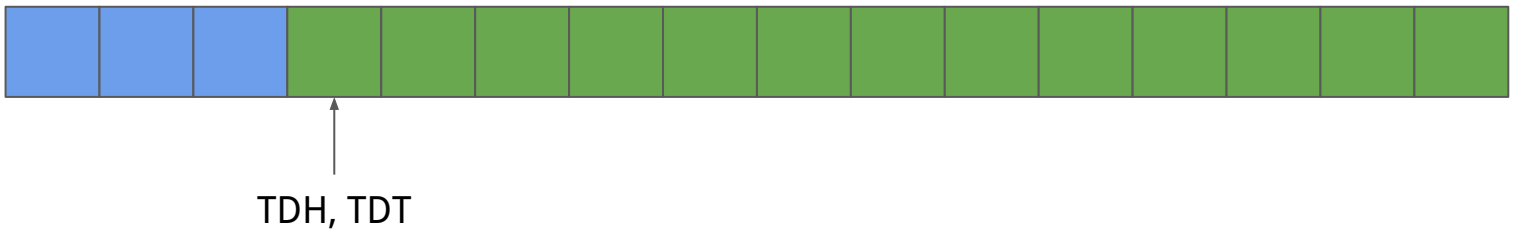
Once the three descriptors are ready, the OS will update the TDT register (with the value of 3 in the example), to make the NIC aware of them. A write to the TDT acts as a *notification* from the OS to the NIC, so that the NIC can start transmitting the prepared descriptors.



e1000 transmission (6)

Once notified, the NIC will start *consuming* the prepared descriptor, starting from the one pointed by TDH. Consuming a descriptor involves reading (DMA) the descriptor from the ring, reading (DMA) the buffer referenced by the descriptor and sending it on the wire. Once a descriptor has been consumed, the NIC advances the TDH to the next position. Moreover, if required by the descriptor flags, the NIC will set the Descriptor Done (DD) bit in the status flag (descriptor *write-back*) and raise a TX interrupt.

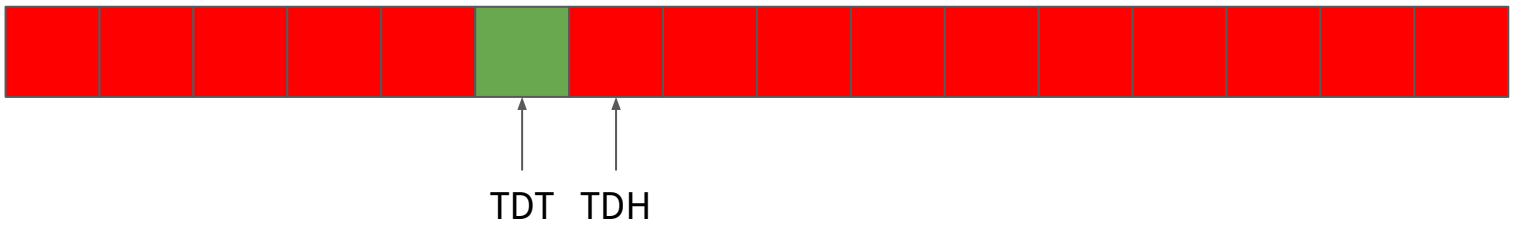
TX processing only stops when TDH reaches TDT.



e1000 transmission (7)

Note that the OS may prepare new descriptors and advance TDT concurrently to the NIC processing descriptors and advancing TDH. Therefore, the TX datapath is a producer-consumer parallel system where the OS produces descriptors and NIC consumes them. OS uses TDT writes to notify (*kick*) NIC that more descriptors are ready for transmission, whereas NIC uses TX interrupts to notify OS that more descriptors have been consumed (and can then be reused for new transmission).

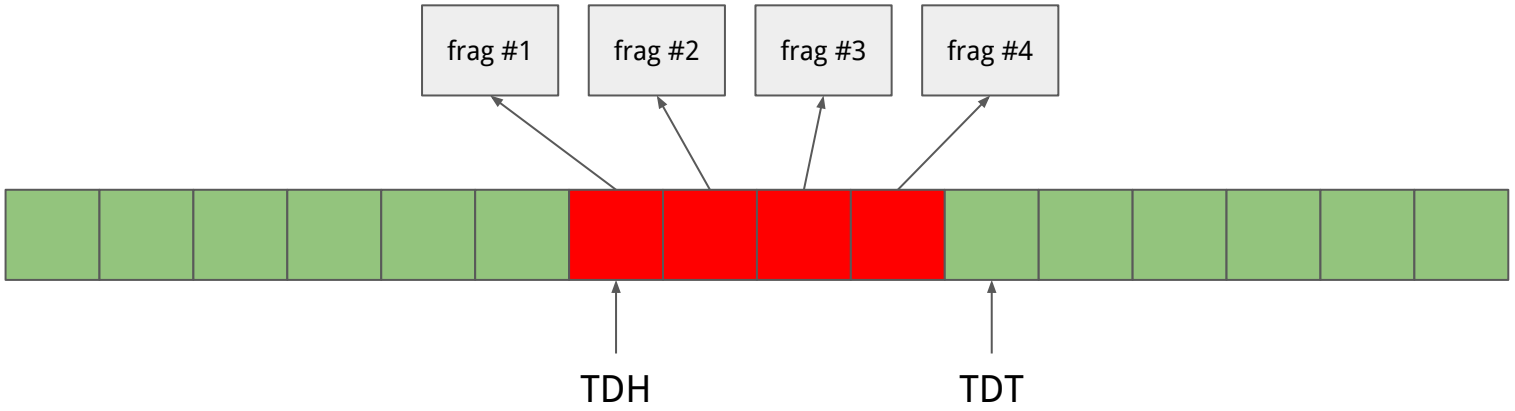
To prevent the index registers to wrap around, the OS should always leave one TX descriptor unused, in such a way that the condition `TDT == TDH` indicates the stop condition for the consumer (no pending descriptors), while `(TDT+1)%N==TDH` (with N number of descriptors in the ring) indicates the stop condition for the producer (no more descriptors available).



e1000 transmission (8)

e1000 also supports Scatter-Gather (SG) transmission, that allows the NIC to transmit a multi-fragment packet (i.e. stored non-contiguous memory chunks). In this case, many consecutive TX descriptors must be used for a single packet (one for each fragment), with descriptor flags indicating that packet continues in the next descriptor.

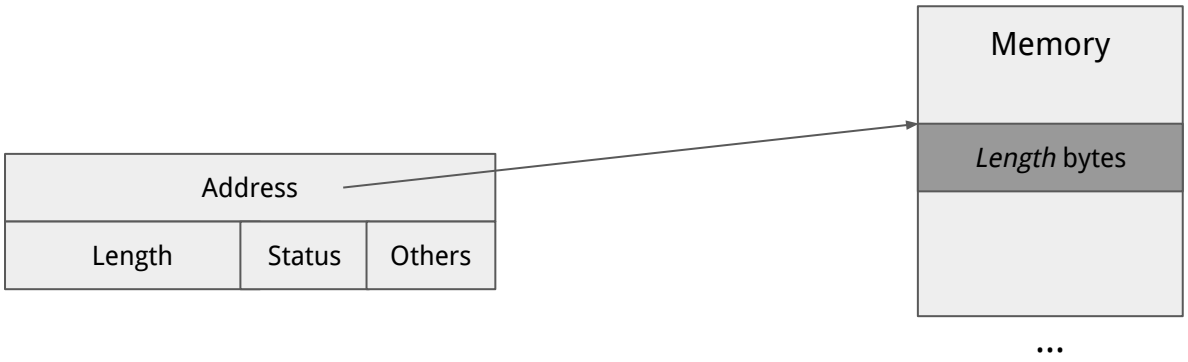
SG transmission is particularly useful with packets that are bigger than page size. This happens with jumbo frames or when TCP Segmentation Offloading (TSO) is used. With TSO, the NIC can process TCP segments up to 64KB, doing the TCP segmentation - to build MTU-sized frames - in hardware.



e1000 reception (1)

Each RX descriptor is 16 bytes long and contains

- The physical address of a buffer containing a packet that has been received from the wire or a buffer that is ready to be used for a next-coming packet.
- The length of the receive buffer.
- A status bit that indicates whether the associated buffer has already been filled by a received packet or is still available.
- Flags and other fields that can be used by the OS to specify RX options to the NIC

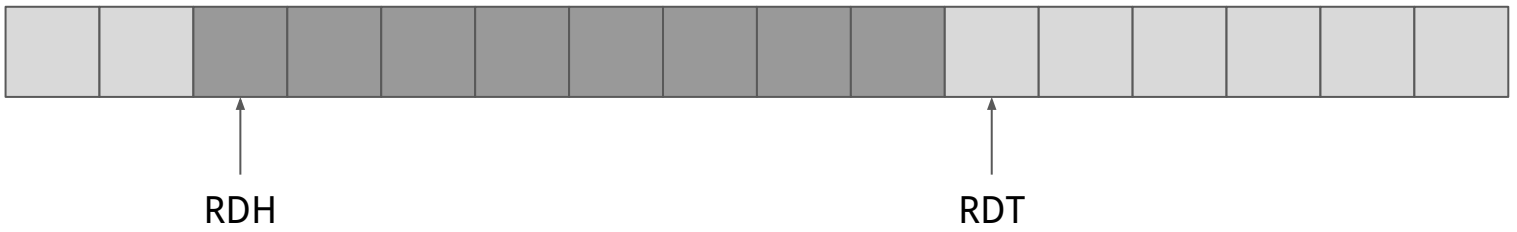


e1000 reception (2)

Synchronization between the OS and the NIC happens through two registers, whose content is interpreted as an index in the RX ring:

- Receive Descriptor Head (RDH): indicates the first descriptor prepared by the OS that can be used by the NIC to store the next incoming packet.
- Receive Descriptor Tail (RDT): indicates the position to stop reception, i.e. the first descriptor that is not ready to be used by the NIC.

Descriptors between RDH (incl.) and RDT (excl.) are owned by the hardware, ready to be used for packets received from the wire. The remaining ones are owned by the software, ready to be prepared by the OS with new buffers to give to the RX hardware.

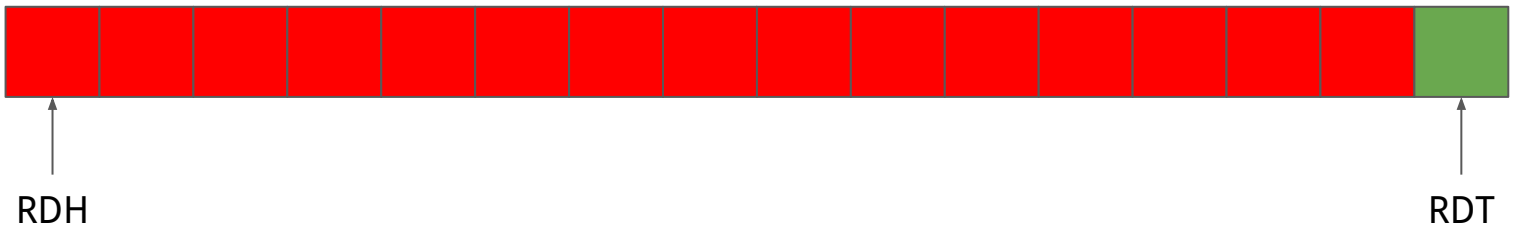


e1000 reception (3)

At the start-up, the OS initializes RDH at zero, and prepares (*produces*) as many descriptors as possible for reception, setting RDT accordingly. Preparing a descriptor involves filling it with a buffer physical address and buffer length, and zeroing the status bit.

To prevent the index registers to wrap around, the OS always leaves one RX descriptor unused, in such a way that the condition `RDT == RDH` indicates the stop condition for the NIC consumer (no descriptors available for reception), while `(RDT+1)%N==RDH` (with `N` number of descriptors in the ring) indicates the stop condition for the OS producer (no more descriptors to prepare).

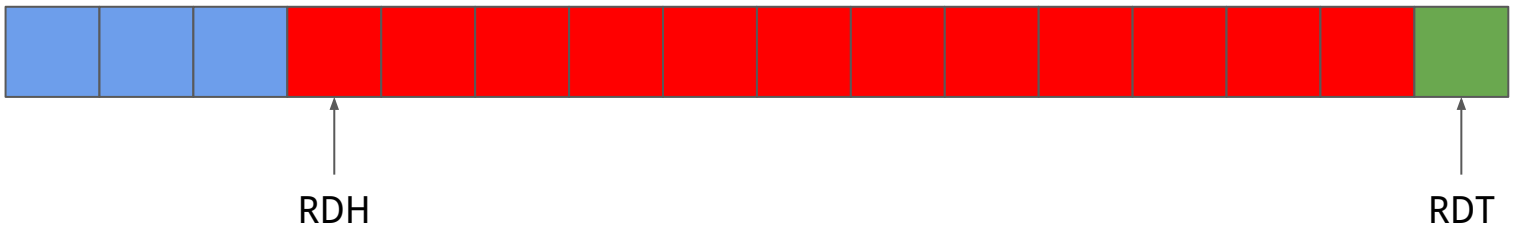
In this situation, almost all the RX descriptors are pending, which means that are available for the NIC to be used for reception. Consequently, no packets have been received yet.



e1000 reception (4)

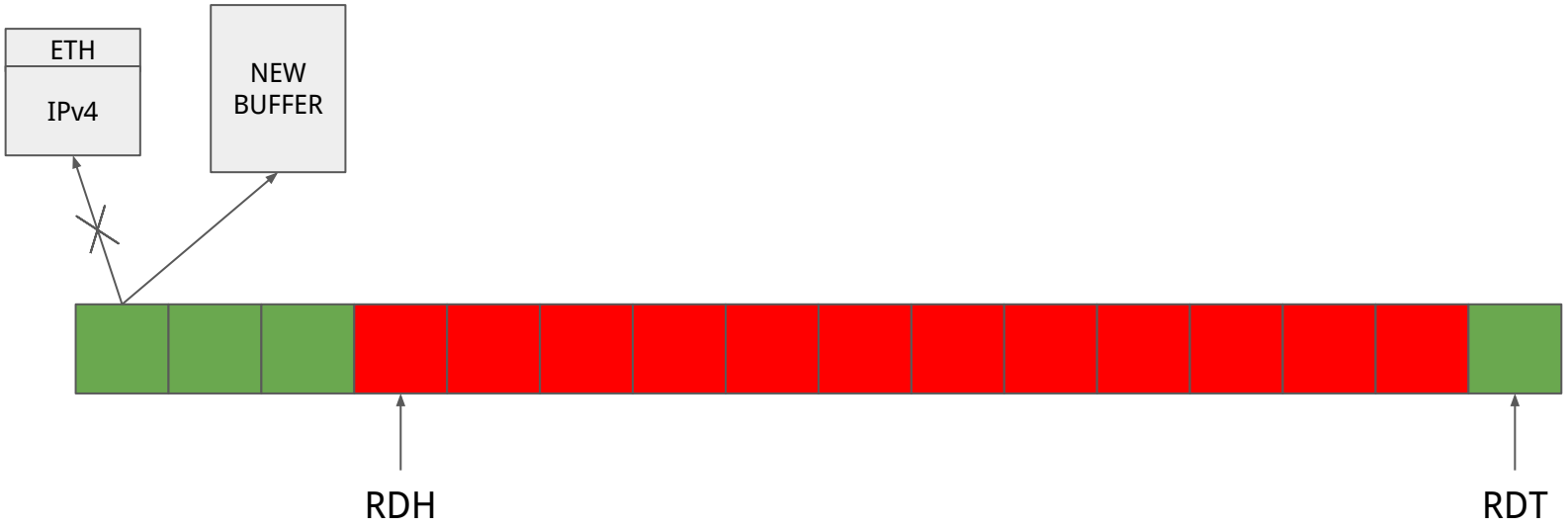
Let’s assume 3 packets are received on the wire, one immediately after the other. Since descriptors are available for reception, the NIC will *consume* three consecutive descriptors, starting from the first available one, which corresponds to the current RDH value.

Consuming a descriptor involves reading (DMA) the descriptor from the ring and writing (DMA) the packet received from the wire to the buffer referenced by the descriptor. Once a descriptor has been consumed, the NIC advances the RDH to the next position. Moreover, the NIC will set the Descriptor Done (DD) bit in the status flag, properly set the length field (descriptor *write-back*) and raise an RX interrupt.



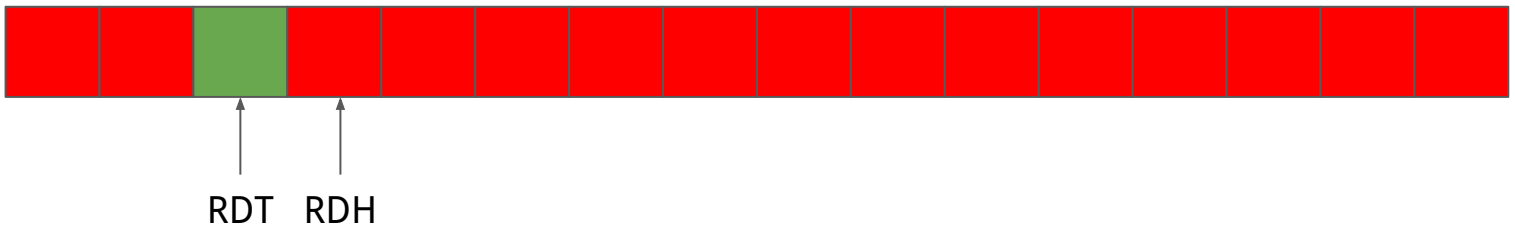
e1000 reception (5)

When the OS processes the RX interrupt, it will go over the received packets and use them (typically putting them in a queue where they can be read by an user-space application). Each consumed descriptor is also replenished, i.e. prepared again, with a (typically) new buffer.



e1000 reception (6)

Once the replenished descriptors are ready, the OS will update the RDT register (with the value of 2 in the example), to make the NIC aware of them. A write to the RDT acts as a *notification* from the OS to the NIC, so that the NIC is aware of the new buffers being available.



e1000 reception (7)

Once again, OS may prepare new descriptors and advance RDT concurrently to the NIC processing descriptors and advancing RDH. Therefore, the RX datapath is a producer-consumer parallel system where the OS produces descriptors and NIC consumes them. OS uses RDT writes to notify (*kick*) NIC that more descriptors are available for reception, whereas NIC uses RX interrupts to notify OS that more descriptors have been consumed (and are then available for applications).

Note that preparing an RX descriptors involves both pushing the received packets to applications and make the descriptor point to a new buffer.

How Linux NIC drivers work (1)

How is the e1000 driver implemented (regarding the datapath parts)?

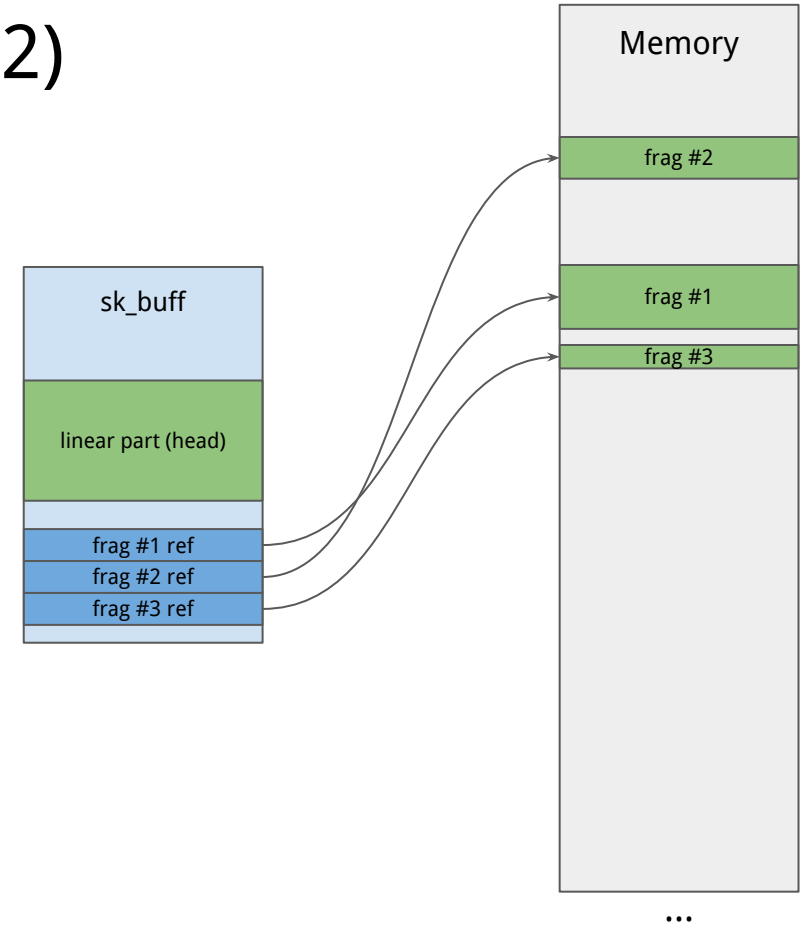
NIC drivers in Linux are usually implemented by loadable kernel modules. The kernel exports an API that NIC drivers use to exchange network packets with the rest of the kernel. By means of this API, the NIC driver can register a new network interface, associating a name to it (e.g. `eth0`, `enp2s0f1`). Registered interfaces can be listed by using the `ifconfig` or `ip link` command-line tools.

Linux extensively uses Object Oriented techniques to manage complexity. The interaction between kernel and driver happens through an abstract class, represented by the `struct net_device`. When registering a new NIC, the driver specifies a set of function pointers (contained in a `struct net_device_ops` object) to provide the implementation of the abstract methods to be invoked by the kernel. About 70 methods are defined, but here we will focus on basic datapath-related ones.

How Linux NIC drivers work (2)

Linux stores each network packet using the `struct sk_buff`, which contains data and metadata. During normal TX and RX operation, these objects (`skbs`) are continuously allocated, deallocated, and passed back and forth between the driver and kernel, possibly on behalf of an user-space application.

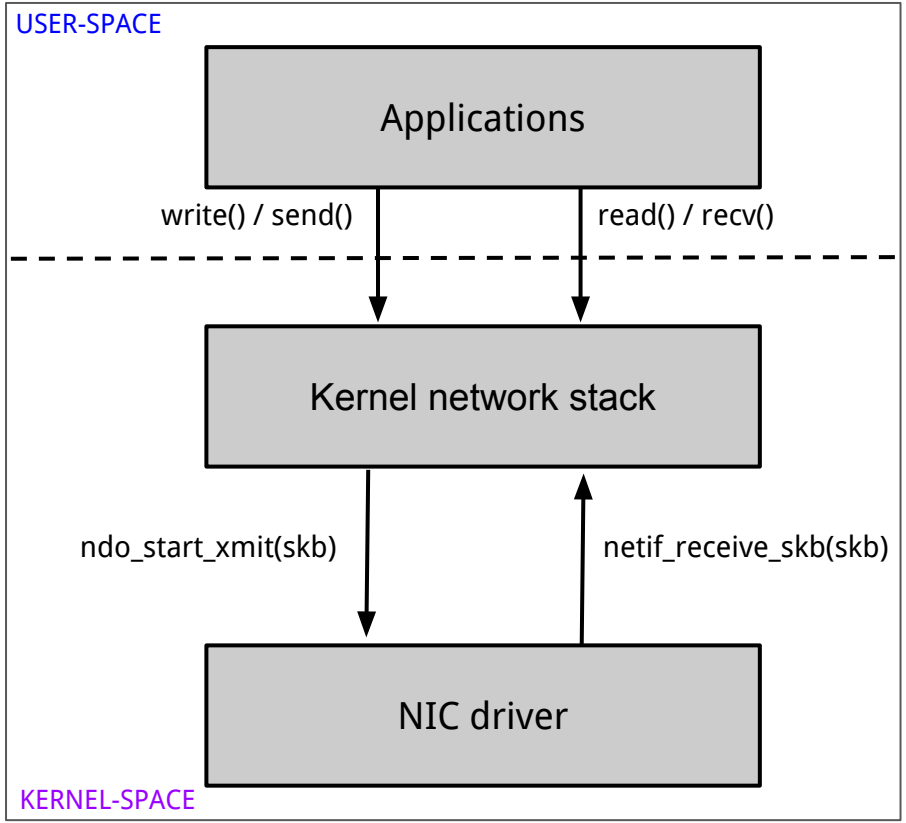
The packet contents are stored partly in the skb object itself (linear part) and partly in pages scattered throughout system memory (paged/indirect part). The linear part always contain the beginning of the packet. The paged part is optional, and usually used for very big packets (e.g. > 1.5KB). Maximum packet size is 64KB.



How Linux NIC drivers work (3)

Some driver methods:

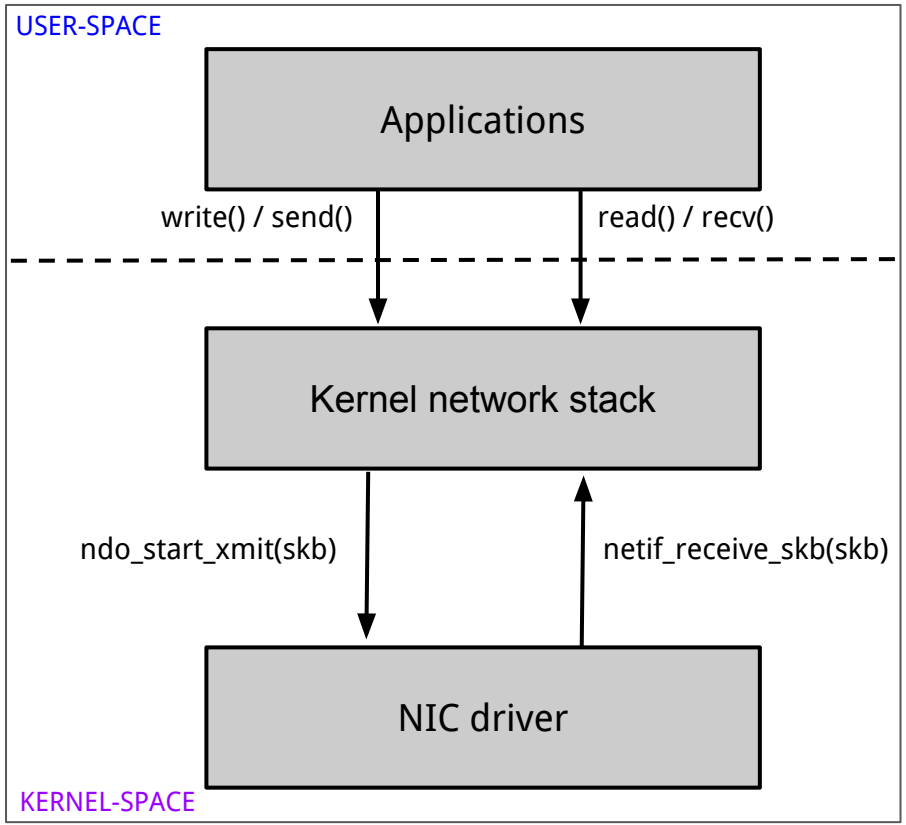
1. `ndo_open(dev)` - The kernel asks the driver to bring the interface up. The driver will initialize NIC rings for transmission and reception and enable the NIC.
2. `ndo_close(dev)` - The kernel asks the driver to bring the interface down. The driver will disable the NIC and clean up NIC rings (e.g. free allocated sk_buffs in TX rings).
3. `ndo_start_xmit(dev, skb)` - The kernel asks the driver to transmit the packet contained in the `skb` object. The ownership of skb is passed to the driver, which means that the driver is in charge to free it.



How Linux NIC drivers work (4)

When the driver receives packets (e.g. after an RX interrupt), it can push them up to the kernel using the `netif_receive_skb(skb)` function or other variants. Depending on the packet contents, the kernel will take the appropriate action:

- Appending the skb to the receive queue of an application socket
- Forward it to another NIC
- Drop it
- ...



NAPI (1)

Inside TX/RX interrupt routines, NIC drivers should do as little work as possible. This principle holds because CPU interrupts are masked while it is executing an interrupt routine, and the latencies of a system can only worsen if interrupts are kept disabled for non-very-short periods. Moreover, interrupt context is a very limited execution environment (e.g. sleeping functions cannot be invoked).

However, a NIC driver needs to respond to TX/RX interrupts by cleaning up TX rings and extracting/replenishing from RX rings, respectively. Since these operations can be quite long, they could be performed in a deferred execution context (e.g. a softirq, a tasklet, workqueue, etc.). An interrupt routine can restrict itself to very simple operations - i.e. acknowledge the interrupt with a register access - and defer the real work.

However, due to the problematic nature of network drivers, the so called *New API* (NAPI) mechanism has been introduced. A NIC driver interrupt routine disables NIC interrupts* and defer its work to a per-CPU NAPI kernel thread. When scheduled, the kernel thread invokes a driver-specific *poll* callback to perform the work. The poll callback enables NIC interrupts only when there is no more pending work.

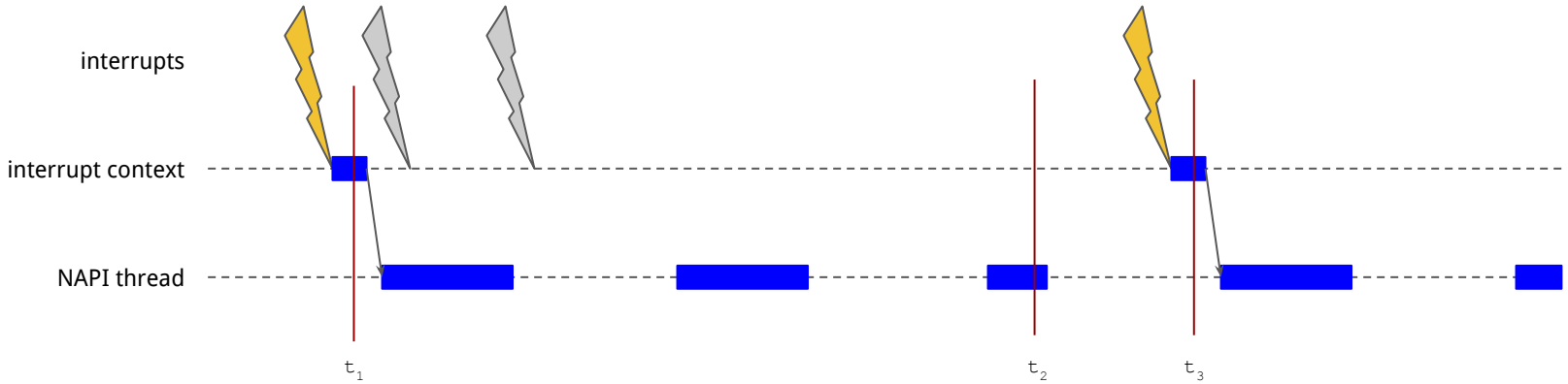
(*) n.b. CPU can still handle interrupts originating from other sources.

NAPI (2)

Each poll callback must limit the amount of work to be done - e.g. number of RX packets processed - in order to ensure fair scheduling among different poll callbacks belonging to different NIC devices. If more work is pending when budget is exhausted, the NAPI kernel thread will reschedule the poll callback as soon as possible.

Temporarily disabling NIC interrupts means *de facto* temporarily switching from interrupt-driven I/O to polled I/O. This has a very beneficial effect on throughput with minimum impact on receive latency.

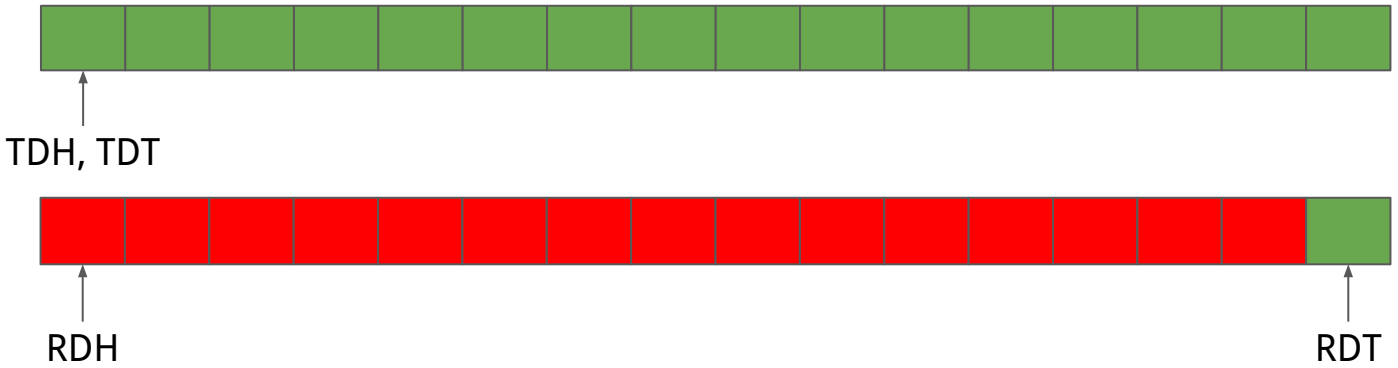
In the example below, polled mode starts at t_1, ends at t_2 and restarts at t_3.



e1000 driver

Among (a lot of) other things, the e1000 `ndo_open` implementation (`e1000_open`):

- Allocates TX and RX rings in physically contiguous DMA-able memory.
- Tells the NIC the physical addresses and length of the allocated TX and RX rings, writing to the `TDBAH, TDBAL, TDLEN, RDBAH, RDBAL, RDLEN` registers.
- Inits `TDH, TDT, RDT, RDH` to zero. At this point TX ring is in the start-up configuration.
- Fills the RX ring with freshly allocated skbs, advancing RDT for each descriptor. At this point RX ring is in the start-up configuration.
- Allocates interrupt vectors and enables interrupts.



e1000 driver transmission (1)

The e1000 `ndo_start_xmit` implementation (`e1000_xmit_frame`) is in charge of mapping the skb received as an input argument into one or more consecutive descriptors of the TX rings. More descriptors are used if the packet referenced by the skb is scattered in memory - this normally happens with GSO packets.

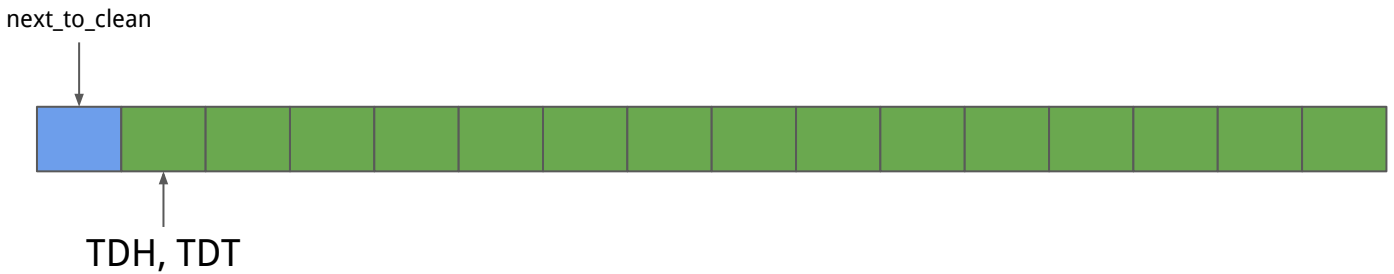
Each descriptor is filled with the physical address and length of a fragment of the packet to be transmitted, plus flags and status fields. The driver must also make sure that all packet memory fragments, allocated in virtual memory, are accessible by the NIC through DMA access, even in the presence of IOMMUs. This is achieved through dynamic DMA mapping*.

Once the descriptors have been prepared (the example assumes a single-fragment packet), the driver can advance `TDT`. Note that `TDT` is updated for each packet transmitted.

(*) https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

e1000 driver transmission (2)

When the NIC has transmitted the submitted packet, it will send an TX interrupt. The driver interrupt routine - shared among all the e1000 interrupts, since the interrupt pin is shared - will just acknowledge the interrupt and schedule the NAPI.

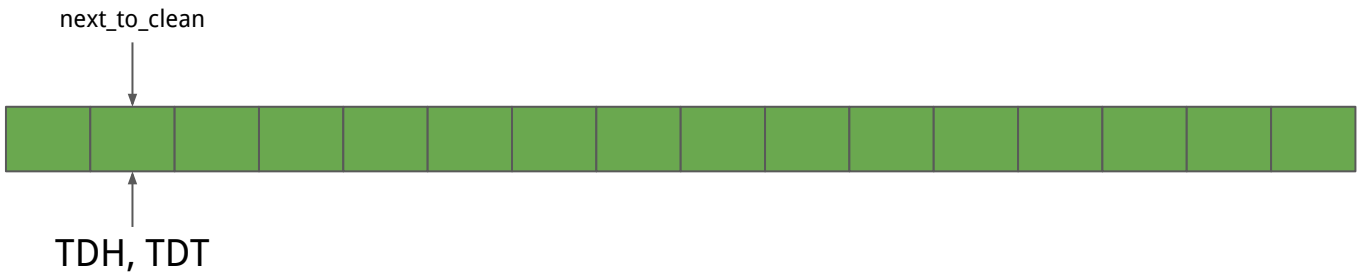


e1000 driver transmission (3)

The e1000 poll callback (`e1000_clean`) takes care of processing consumed descriptors for both rings. The `e1000_clean_tx_irq` subroutine scans the TX ring looking for consumed descriptors - i.e. completed transmissions. A consumed descriptor has the DD bit set in the status field.

A consumed descriptor is cleaned up by destroying the dynamic DMA mapping for the packet fragment, and free the transmitted skb.

A ring index variable (next_to_clean) keeps track of the next descriptor to be cleaned.

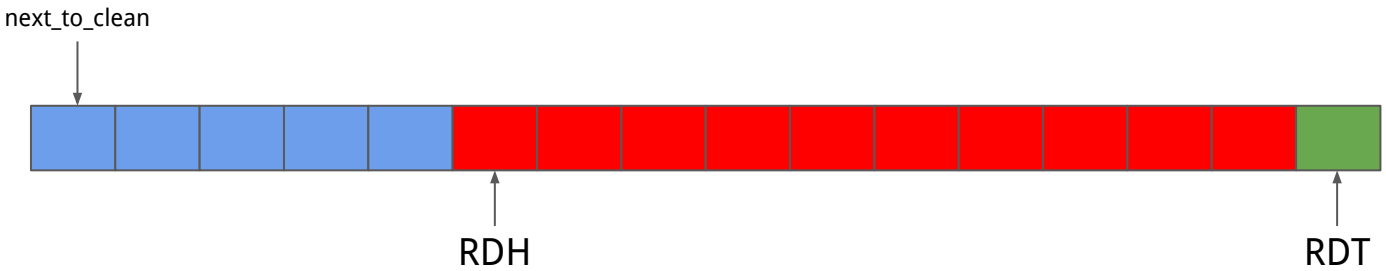


(*) This clearly happens just once for each multi-fragment packets.

e1000 driver reception (1)

When the NIC receives one or more packets, it raises an RX interrupt. The interrupt routine schedules the NAPI, that will invoke the e1000 poll callback.

In the example, blue descriptors contain received packets ready to be processed by the driver.

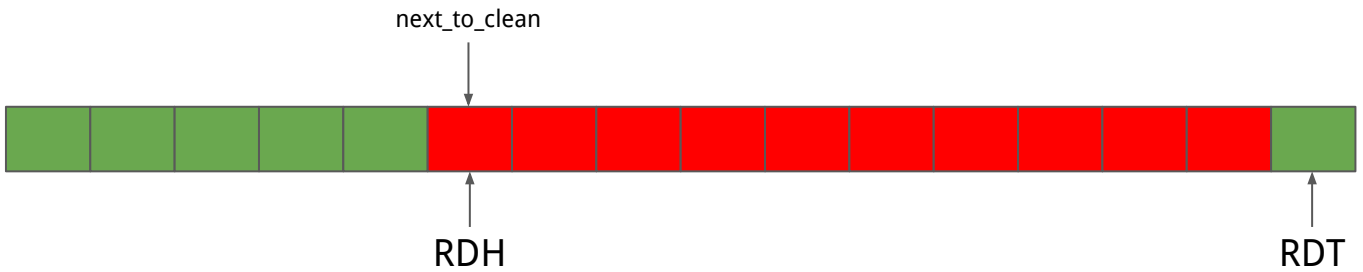


e1000 driver reception (2)

The `e1000_clean_rx_irq` subroutine scans the RX ring looking for consumed descriptors - i.e. completed receptions. A consumed descriptor has the DD bit set in the status field.

A consumed descriptor is cleaned up by destroying the dynamic DMA mapping for associated skb and pushing the skb up to the kernel where it will find its destination.

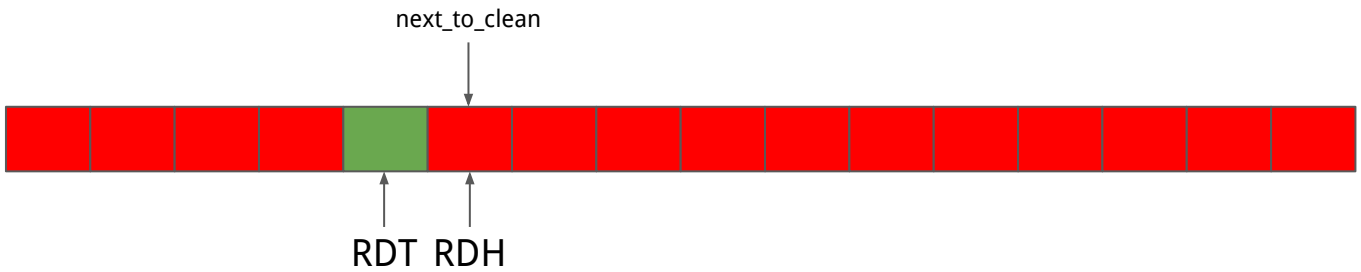
Similarly to what happens in the TX ring, a ring index variable (next_to_clean) keeps track of the next descriptor to be cleaned.



e1000 driver reception (3)

Once consumed RX descriptors have been cleaned, they can be prepared again to receive new packets. Each descriptor is refilled with the physical address and length of a buffer contained into a freshly allocated skb, plus flags and status fields. The driver will also setup a dynamic DMA mapping for the new skb's buffer, similarly to what happens for transmission.

Note that these refill operations (skb allocations and DMA mapping) are the same executed at start-up time, to make the RX ring ready to receive some packets.



e1000 emulation

The difference between RAM accesses and register accesses is that the latter ones may have side-effects - registers are not merely storage. RAM accesses from the guest code do not require any hypervisor intervention.

Register accesses, instead, have a meaning that is dependent on the specific device, and that's why register accesses for emulated devices must be emulated in software.

Assuming hardware-based virtualization, how does QEMU (or in general type-1 hypervisors) emulate an e1000 NIC?

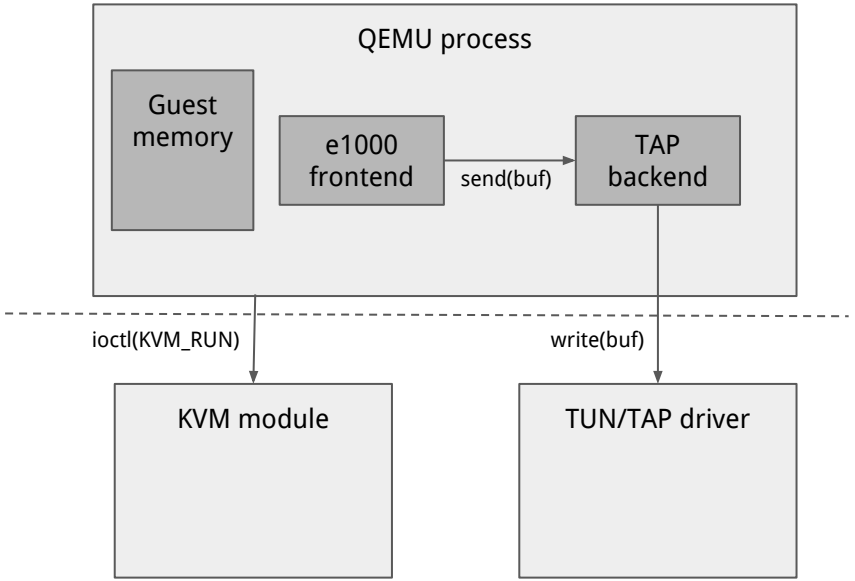
In QEMU terminology, the piece of software implementing device emulation is known as *front-end*, so let's see how the e1000 front-end works, keeping in mind that

- A QEMU process can access all the guest system memory of the guest, since from its point of view it's just a set of malloc()ed buffers.
- QEMU keeps internal data structures (not visible to the guest) to represent I/O devices, CPUs, memory regions, and lots of other things.

e1000 transmission emulation (1)

When an host thread* is natively executing guest code, a write to the TDT register - or to any other register - causes the CPU to exit from the native emulation mode (VM exit) and return in the KVM module (host kernel-space) that was executing an `ioctl(KVM_RUN)` syscall for QEMU.

When the ioctl returns, QEMU figures out that the VM exit was due to a register access for an e1000 device and invokes the e1000 frontend.



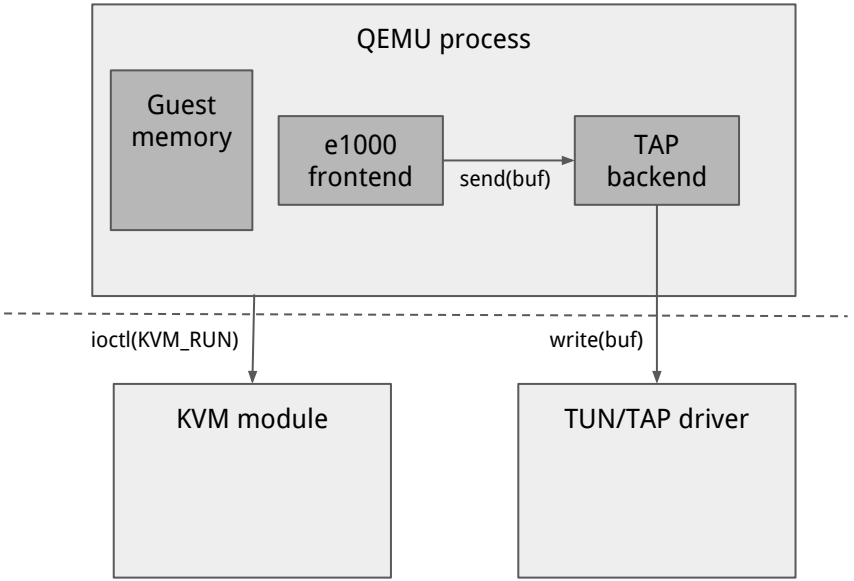
(*) This is also referred to as *vCPU thread*.

e1000 transmission emulation (2)

The e1000 handler for `TDT` writes collects all the produced TX descriptors (from `TDH` to `TDT-1`). For each one, translates the guest physical address of the Ethernet frame into host virtual address, sends the frame to a network *back-end*, writes back the descriptor (i.e. DD bit) and increments `TDH`.

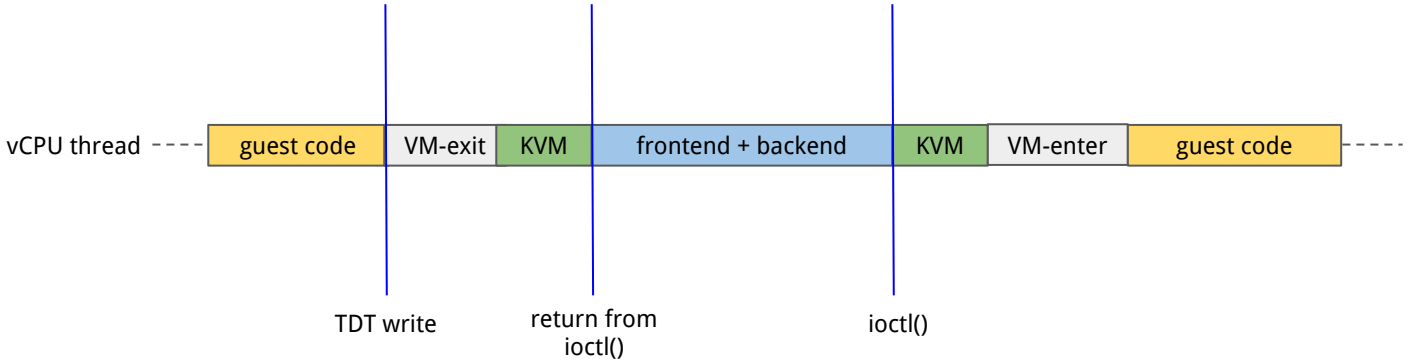
QEMU supports different back-ends: netuser (NAT), TAP, netmap, etc. TAP is the most commonly used in Data Centre deployments. Frames written to a TAP file descriptor are injected in the host network stack as they were received on a physical interface.

Once all descriptors have been a TX interrupt is emulated by means of the KVM module.



e1000 transmission emulation (3)

All the TX processing here described is executed by the same vCPU thread that was executing the guest code. The e1000 emulated TX processing is then completely synchronous, differently from what happens with a real e1000 NIC, where the NIC hardware runs in parallel to the CPU running the sending application and the driver code.



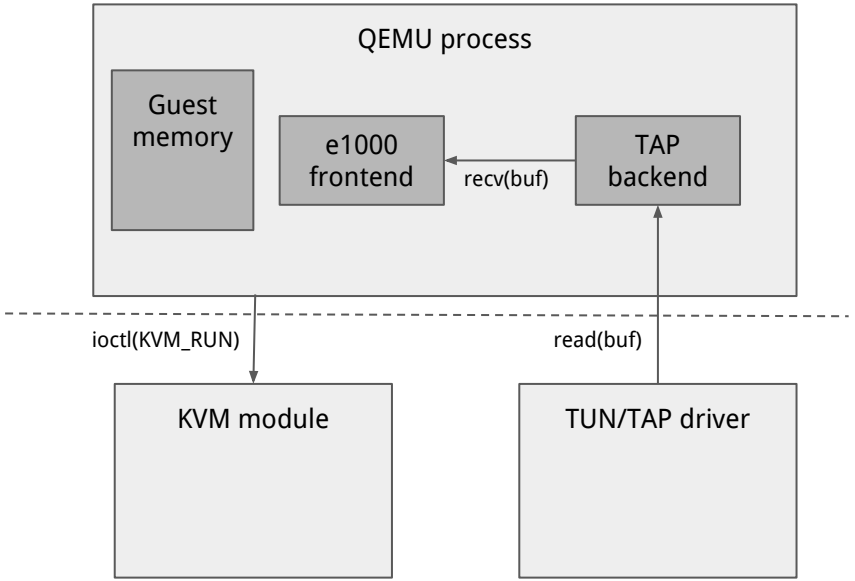
e1000 reception emulation (1)

When the host kernel transmit a packet on the TAP network interface, the Ethernet frame becomes available to be read from the TAP file descriptor.

A QEMU thread executing the main event-loop (the *I/O thread*) wakes up and invokes the TAP backend. The backend, in its turn, reads the frame from the TAP file descriptor and sends it to the e1000 frontend.

The e1000 frontend - assuming `RDH != RDT` - uses the buffer referenced by next available RX descriptor to copy the frame to the guest memory, writes back the descriptor, advances `RDH` and emulates an RX interrupt.

Similarly to what happens with transmission, a guest-physical to host-virtual address translation is necessary.

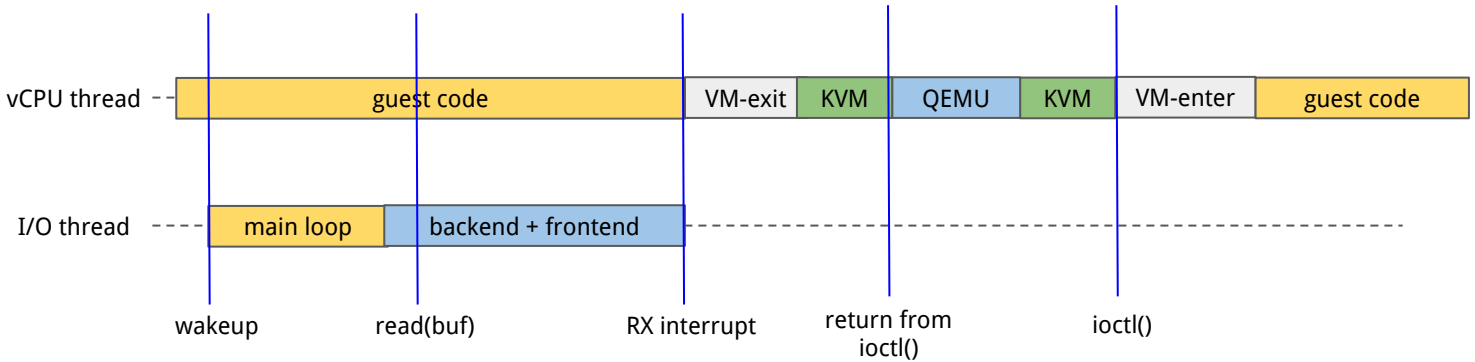


e1000 reception emulation (2)

Differently from TX datapath, RX emulation can run in parallel to guest code (e.g. guest driver and application), since the frontend and backend processing is executed in the I/O thread. This is a simple consequence of the QEMU architecture.

The side effect of writes to `RDT` is to re-enable the backend if it was temporarily disabled because of the consumer stop condition (`RDH == RDT`).

In this example we didn't assume posted interrupts: on interrupt, the vCPU thread is forced to VM exit.



Performance of emulated e1000

Some numbers, assuming recent i5/i7 CPUs (3th and 4th generation):

- Register write are very expensive when executed by a guest, since they cause a VM exit (with involved KVM and QEMU processing and the subsequent VM enter). The overhead is in the order of 1-10 *us*.
- Interrupts are extremely expensive in both host and guest environment - they have high hardware and OS overhead. They becomes even more expensive with VMs, since they hardware overhead (context switch) is replaced by a VM exit*. Overhead is in the order of 5-40 *us*.

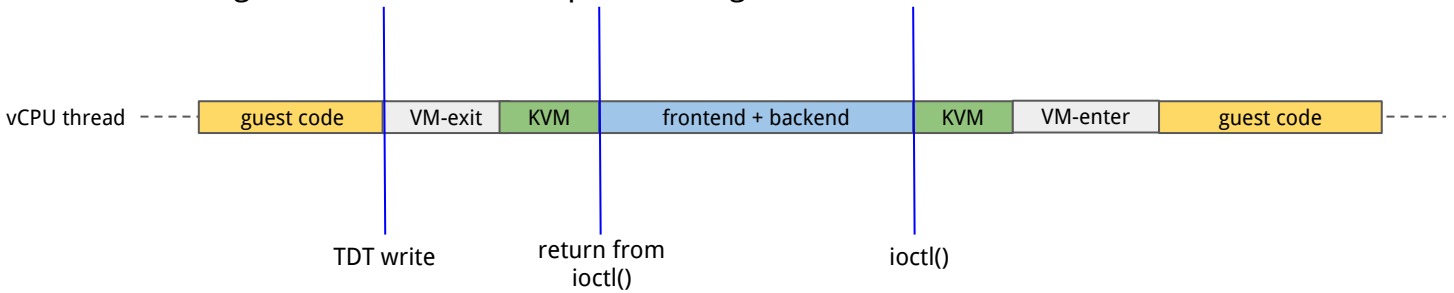
These numbers may look acceptable to you, but they actually aren't in Data Centre environment. A 20 *us* per-packet overhead implies a 50 Kpps packet-rate upper bound. 40Gbit hardware NIC can do over 50 Mpps (with short packets).

(*) Posted interrupt mitigate this problem, but cost remains high.

Transmission throughput performance (1)

Some observations:

- For each packet transmitted - i.e. for each `ndo_start_xmit` call - there is one producer notification (`TDT` write) and, with one vCPU*, one consumer notification (TX interrupt).
- The interrupt routine needs 5 register accesses
  - ICR (Interrupt Cause Read) read to get the interrupt reason and acknowledge it.
  - IMC (Interrupt Mask Clear) write to disable the interrupts.
  - STATUS register read to flush the previous register write.
  - At NAPI polling exit, IMS (Interrupt Mask Set) write to enable the interrupts.
  - STATUS register read to flush the previous register write.



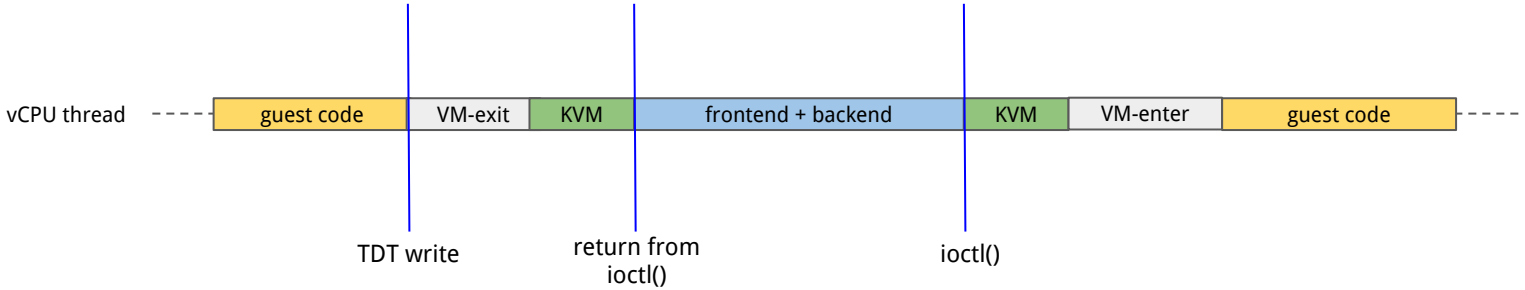
(*) With multiple vCPUs, some ndo_start_xmit call may go in parallel with the NAPI poll callback, which runs with TX interrupts disabled, and therefore some interrupts are coalesced.

Transmission throughput performance (2)

Short packet experiments are good at evaluating the per-packet fixed overhead, without interference due to size-dependent copy overhead. We measure guest-to-host transmission of UDP packets. UDP is chosen over TCP to avoid interferences due to ACKs, in such a way to better evaluate the TX datapath.

Performance is really poor, being killed by notification overhead - interrupts and register accesses.

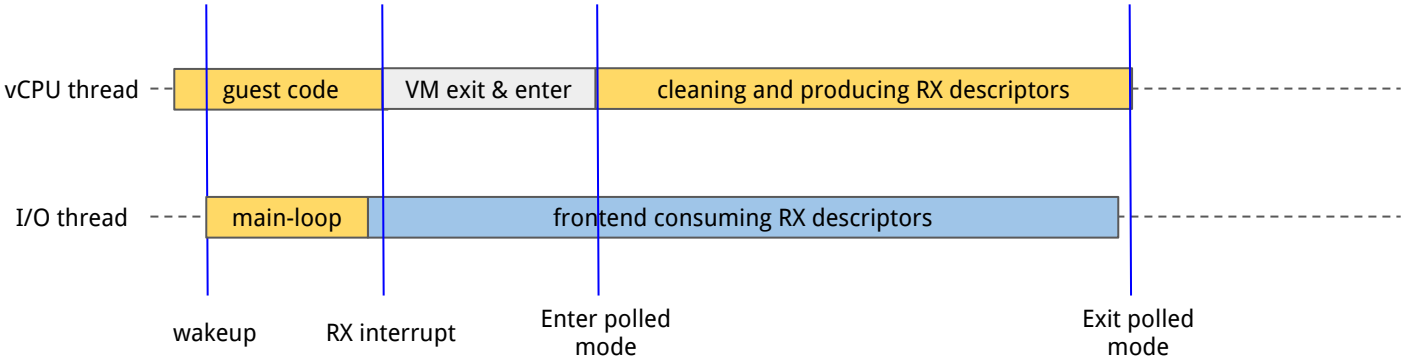
UDP payload	Packet rate	Throughput
20 B	32 Kpps	2.5 Mbps
500 B	32 Kpps	126 Mbps
1.5 KB	17 Kpps	206 Mbps
15 KB	3.6 Kpps	438 Mbps
64 KB	910 pps	476 Mbps



Receive throughput performance (1)

Some observations:

- With the vCPU (NAPI) thread running in parallel to the QEMU IO Thread, RX interrupt coalescing is possible, that is very beneficial to performance. However, how coalescing shows up really timing-dependent (that is machine-dependent), so measurements are often very difficult to reproduce
- For each interrupt processed by the driver, there is one producer notification (RDT write).
- When coalescing happens, interrupt cost (and RDT write) is amortized over a large batch of packets.



Receive throughput performance (2)

With host transmitting UDP packets to guest application we can measure RX datapath performance. Because of interrupt coalescing, packet rates are way higher.

Unfortunately, high-packet rates may cause *receiver livelock**, with the application not having enough time to drain its user-space socket buffer, and driver forced to drop lots of packets because the socket buffer is full.

UDP payload	Sender rate	Receiver rate
20 B	740 Kpps	600 Kpps
500 B	700 Kpps	590 Kpps
1.5 KB	470 Kpps	136 Kpps
15 KB	127 Kpps	15 Kpps
64 KB	33 Kpps	17 Kpps

(*) It's called livelock because the RX datapath is doing a lot of useless work, since packets will be dropped at the very end.

Latency performance

People like large bandwidth, but latency is maybe even more critical*. Among the other things, it really affects TCP performance, especially for bursty traffic (e.g. HTTP). With bursty traffic, we cannot count on batch of packets to amortize notification costs, since traffic is sparse. Latency performance is therefore limited by interrupt overhead and multiple register accesses.

Guest-to-host or host-to-guest ping-pong experiments, carried out using netperf UDP request-response tests, reports a maximum transaction rate of about 17000. This means that the average round trip time between an application running on the guest and one running on the host is about 60 *us*. Most of this overhead is due to the two interrupts (RX and TX) and the 12 register accesses (`TDT` and `RDT` write plus 5 accesses per interrupt).

(*) The *bufferbloat* problem comes from this misconception.

I/O paravirtualization ideas (1)

Our e1000 NIC is emulated, it's not real hardware. So why do we emulate?

Emulation is just a nice way to achieve software compatibility. By means of emulation, a guest can simply reuse unmodified e1000 driver, network stack and all the rest of software appliance. This situation is also referred as *full virtualization*.

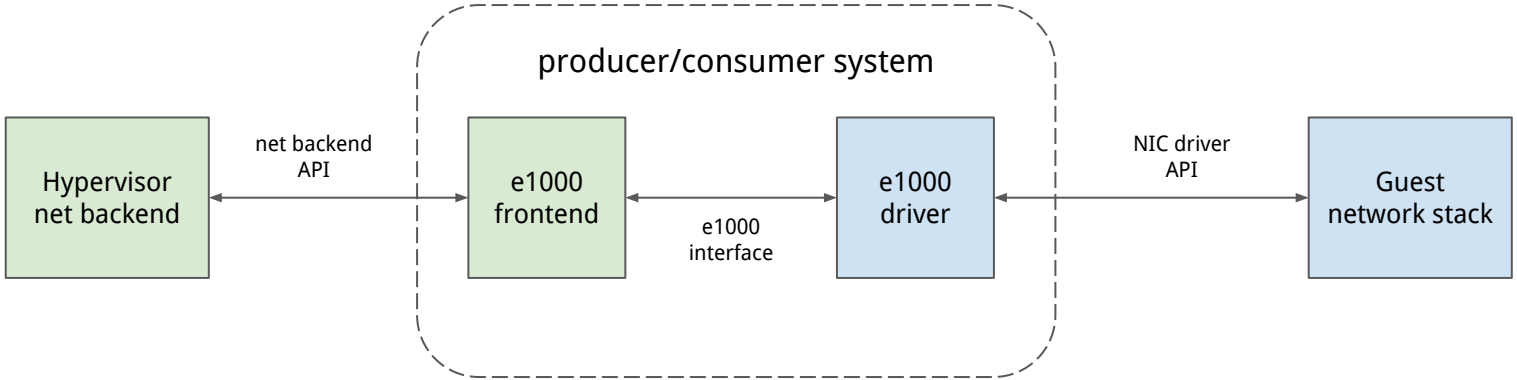
Questions follow:

1. Do we really need 5 register accesses per interrupt?
2. Do we really need to to emulate things like link auto-negotiation, on-board flash or EEPROM?
3. Do we need to emulate plenty of registers that control hardware details that do not even make sense in software, and that we're not required to emulate?
4. Do we really need that many producer/consumer notifications (register access, interrupts)?

I/O paravirtualization ideas (2)

In the very end, we can see the e1000 driver and the e1000 frontend, together, as a producer consumer system that transfers buffers back and forth between the guest kernel and the the hypervisor backend. The driver produces output buffers to be transmitted and input buffers to be filled in. The frontend consumes output buffers passing them to the backend and consumes input buffers by filling them in. The e1000 rings are two queues that decouple producer from consumer.

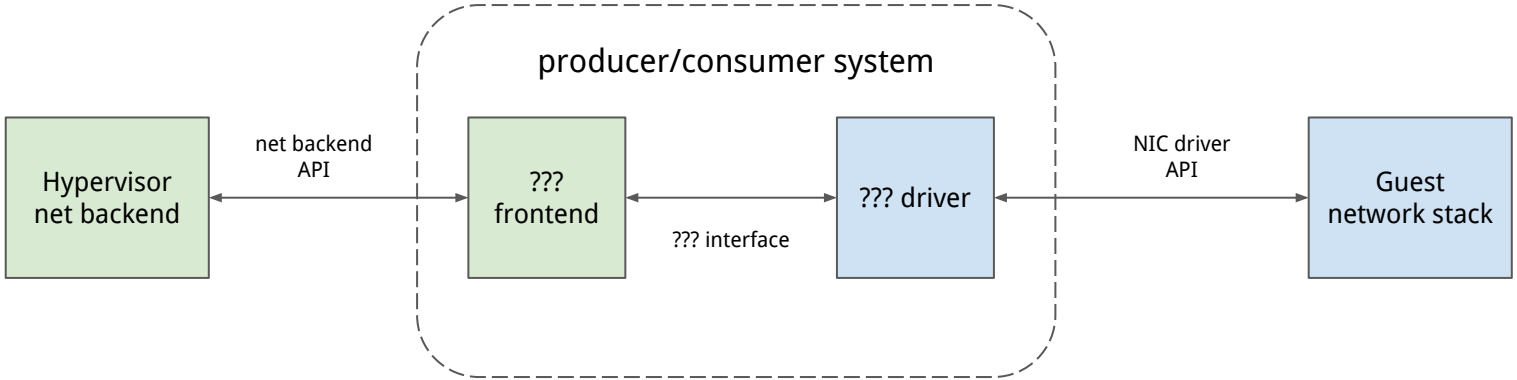
Reformulating the last questions: is it possible to build a better - simpler and more efficient - producer/consumer system that matches the same interfaces as the composite e1000 driver + frontend?



I/O paravirtualization ideas (3)

A better producer/consumer system should follow these principles:

A. Registers used only by the producer to notify the consumer that a queue is not empty anymore. They should not be used to store producer and consumer state (e.g. ring indices).
B. Interrupts used by consumer to notify producer that a queue is not full anymore.
C. Producer and consumer state should be stored in memory, so that both producer and consumer can read it without VM exit overhead.
D. Producer and consumer should be run in separate threads, so that they can work in parallel. They should both try to do as much processing as possible (polling) before going to sleep again, to amortize notification and wake-up costs.
E. Notifications should not be used when not necessary, i.e. when the other party is actively processing (not sleeping).
F. Busy waiting (uncontrolled polling) is not an acceptable general-purpose solution.

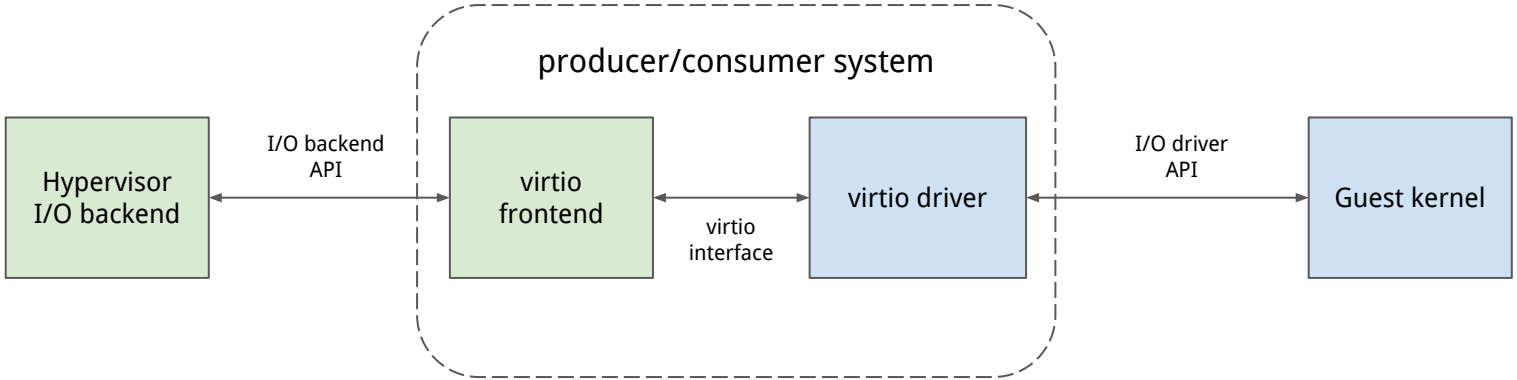


The VirtIO interface (1)

These principles are the foundation of I/O *paravirtualization*, which means that the guest device driver is aware of running in a VM, while the rest of the guest kernel is not.

To get things right, we should note that the same principles can be applied also to other forms of virtualized I/O (block storage, serial ports, …), since all forms of I/O can be seen as producer/consumer systems that exchange messages.

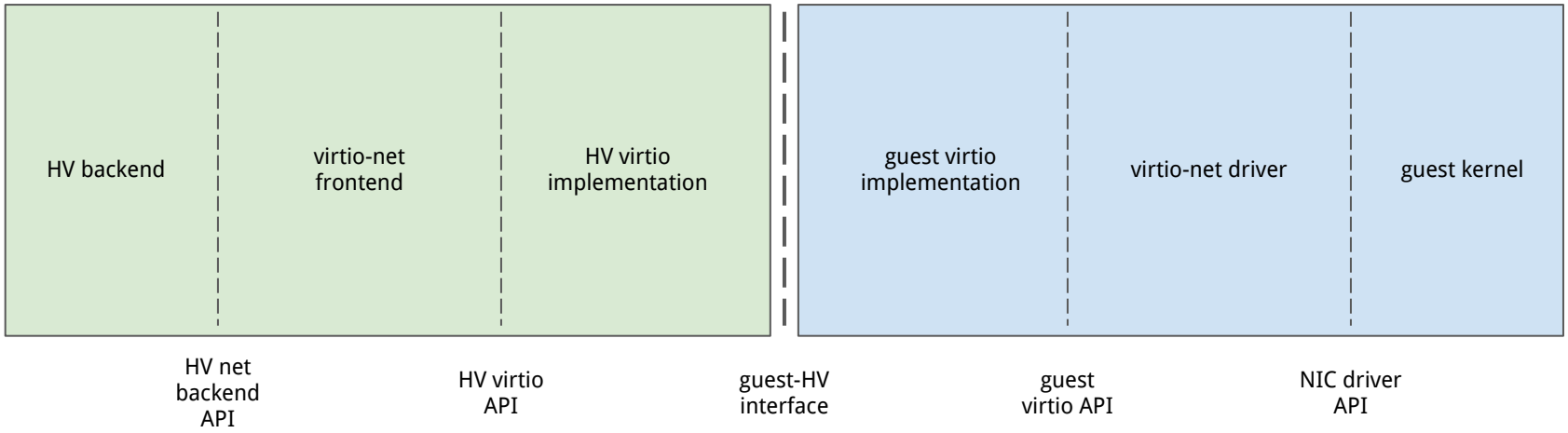
In any case we need to define a reasonable interface that is compatible with them. The VirtIO standard has been introduced to take this role.



The VirtIO interface (2)

VirtIO aims at high performance I/O through device paravirtualization. It's an effort to establish a standard message passing API between drivers and hypervisors. Different drivers and frontends (e.g. a network I/O and block I/O) can use the same API, which implies code reuse of the API implementation.

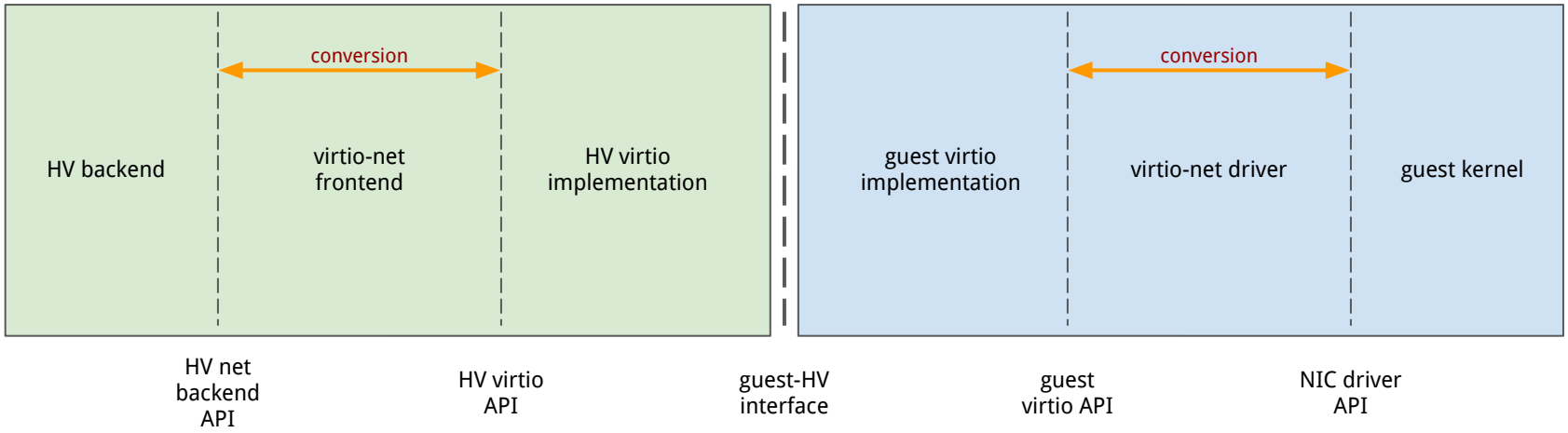
Moreover, adopting the standard avoids introducing ad-hoc I/O paravirtualization solutions (like Xen netfront/netback). VirtIO is currently adopted by QEMU, VirtualBox and bhyve.



The VirtIO interface (3)

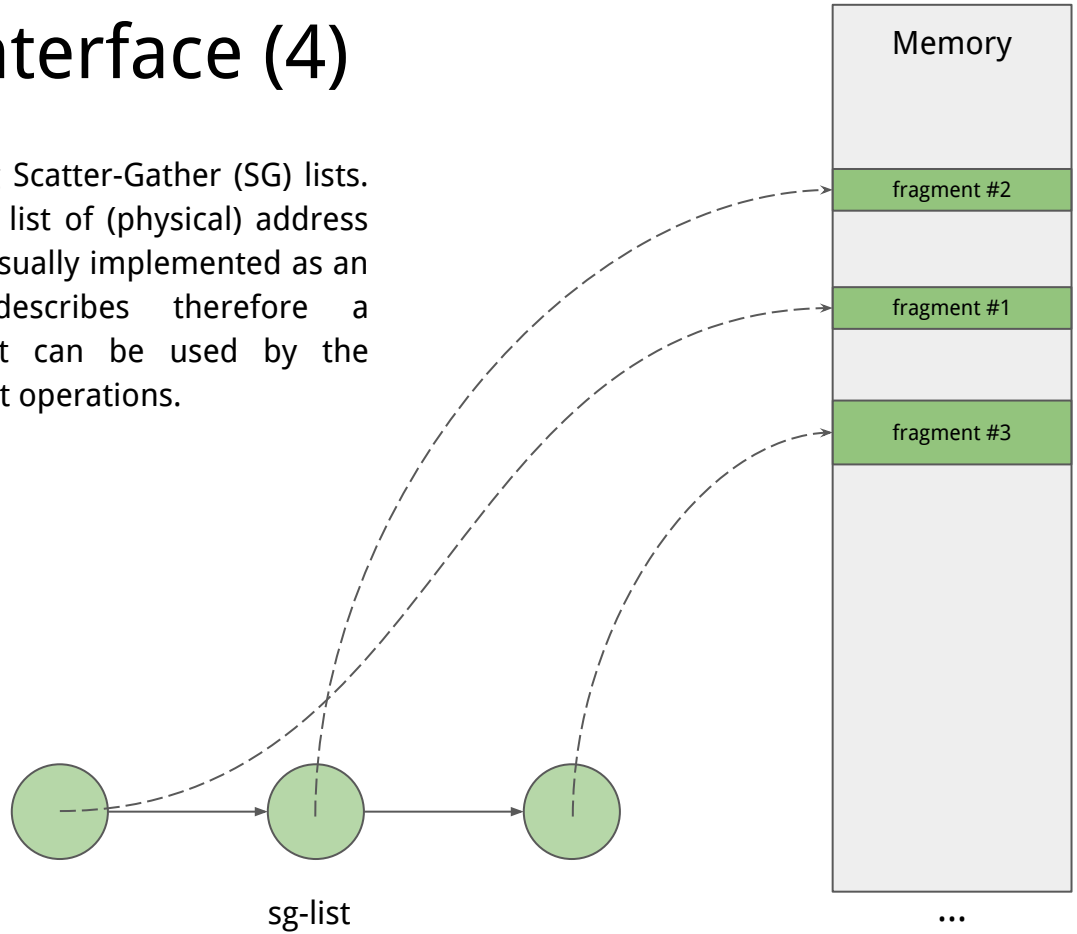
The task of a VirtIO driver is to convert the OS-specific representation of the message (e.g. a skb object for a Linux network driver) to the VirtIO message format, and the other way around.

The virtio-net frontend performs the same task on the hypervisor side, converting VirtIO messages from/to formats understandable to the backend.



The VirtIO interface (4)

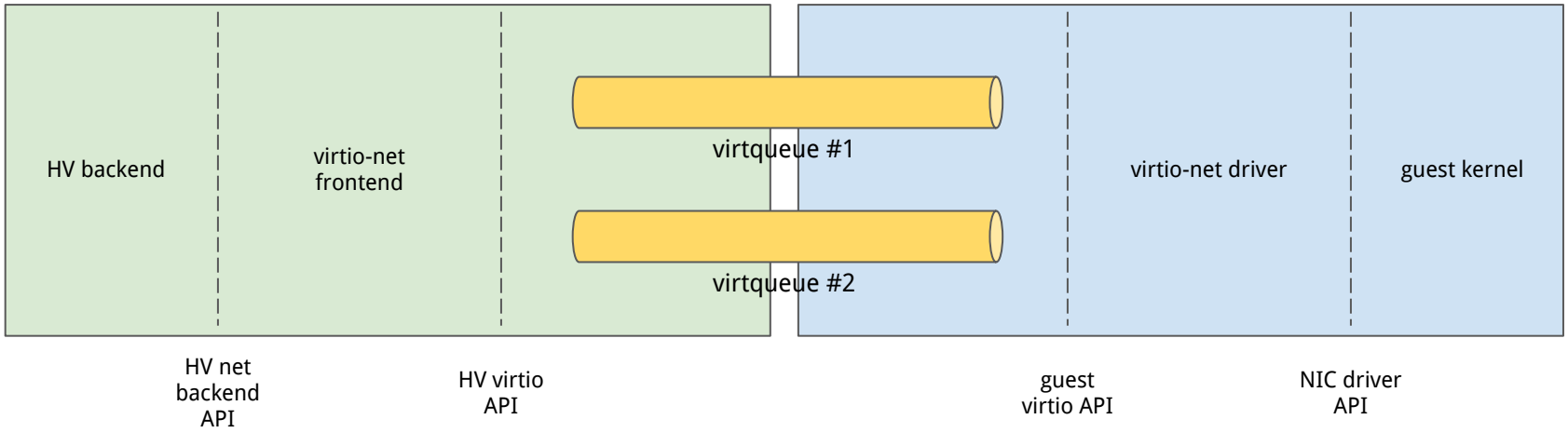
VirtIO exchanges data using Scatter-Gather (SG) lists. An SG list is conceptually a list of (physical) address and length couples, and is usually implemented as an array. Each SG list describes therefore a multi-fragment buffer, that can be used by the consumer for input or output operations.



Virtqueues (1)

Central to VirtIO is the *Virtqueue* (VQ) abstraction. A VQ is a queue where SGs are posted by the guest driver to be consumed by the hypervisor. Output SGs are used to send data to the hypervisor, while input SGs are used to receive data from the hypervisor.

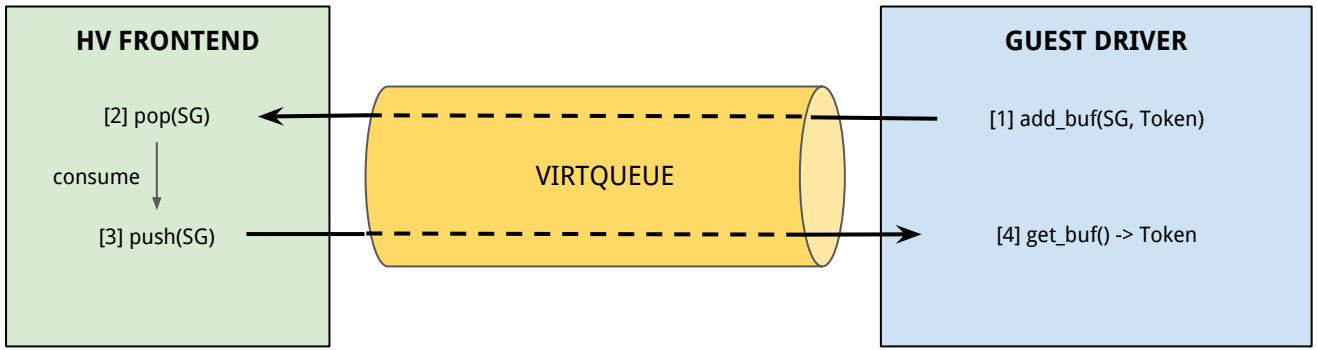
A device can use one or more queues, and the number of queues may be negotiated.



Virtqueues (2)

When a guest driver wants to produce a SG, it calls the `add_buf` VQ method, also passing a token. On the other side, the HV pops the SG, consumes it (interacting with a backend), and pushes it back in the VQ.

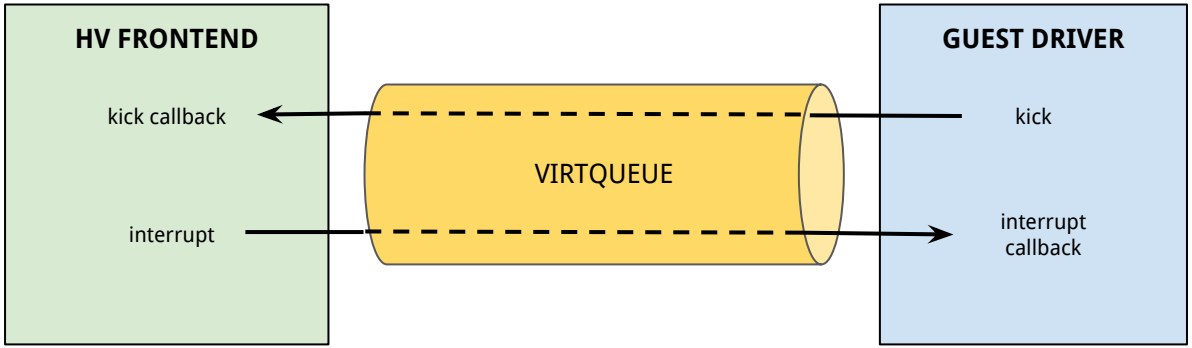
The guest polls for used SGs by calling the `get_buf` VQ method, so that can perform cleanup operations. The token - which is opaque for the VQ and is not passed to the HV - can be used by the driver to match produced SGs (requests) against consumed ones (responses). Tokens allow for out-of-order SG consumption (e.g. useful with block I/O).



Virtqueues (3)

When the driver wants the HV to start consuming SGs, it notifies it using a VQ *kick* (implemented with a register write). Similarly, when the HV wants to notify the driver about consumed SGs, it uses a VQ *interrupt*.

To avoid busy waiting on `get_buf`, the driver can register a per-VQ callback function that is invoked on VQ interrupt. Similarly, to avoid busy waiting on pop, the HV can register a per-VQ callback function that is invoked on VQ kick.

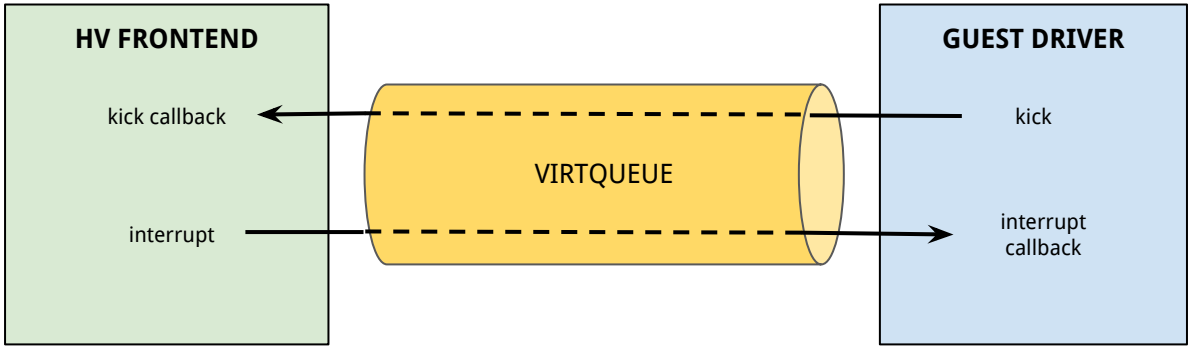


Minimizing notifications (1)

The *kick* and *interrupt* operations are part of the VirtIO interface. The driver should produce as many SGs as possible before kicking (principle E). Similarly, the HV should consume as many SGs as possible before sending an interrupt. In this way, notification costs are amortized over many SGs.

This strategy is usually difficult to deploy. Most of the times the producer doesn't know when the next SG will come, so it's forced to kick after each `add_buf`. The HV, instead, interrupts after each consumed packet to minimize latency.

Luckily, drivers can temporarily disable interrupts and HV can temporary disable kicks!



Minimizing notifications (2)

If we follow principle D, we have to run the HV `pop/consume/push` processing in a separate thread from the vCPU running the guest producer code. The HV thread can be waken up by the VQ kick callback.

Similarly, the `get_buf/cleanup` processing should run in a separate guest thread, to be waken up by the VQ interrupt callback.

Key observations follows:

- Once the HV consumer thread has been waken up, VQ kicks can be disabled to temporarily switch to polled mode. When all the pending SGs have been consumed, VQ kicks can be enabled again.
- Once the guest cleanup thread has been waken up, VQ interrupts can be disabled to temporarily switch to polled mode. When all the consumed SGs have been cleaned up, VQ interrupts can be enabled again.

This strategy (similar to Linux NAPI) allows both producer and consumer to temporarily switch from interrupt mode to polled mode. In this way, notifications (kicks and interrupts) may be amortized over many packets. Does this actually happen? It depends on timings.

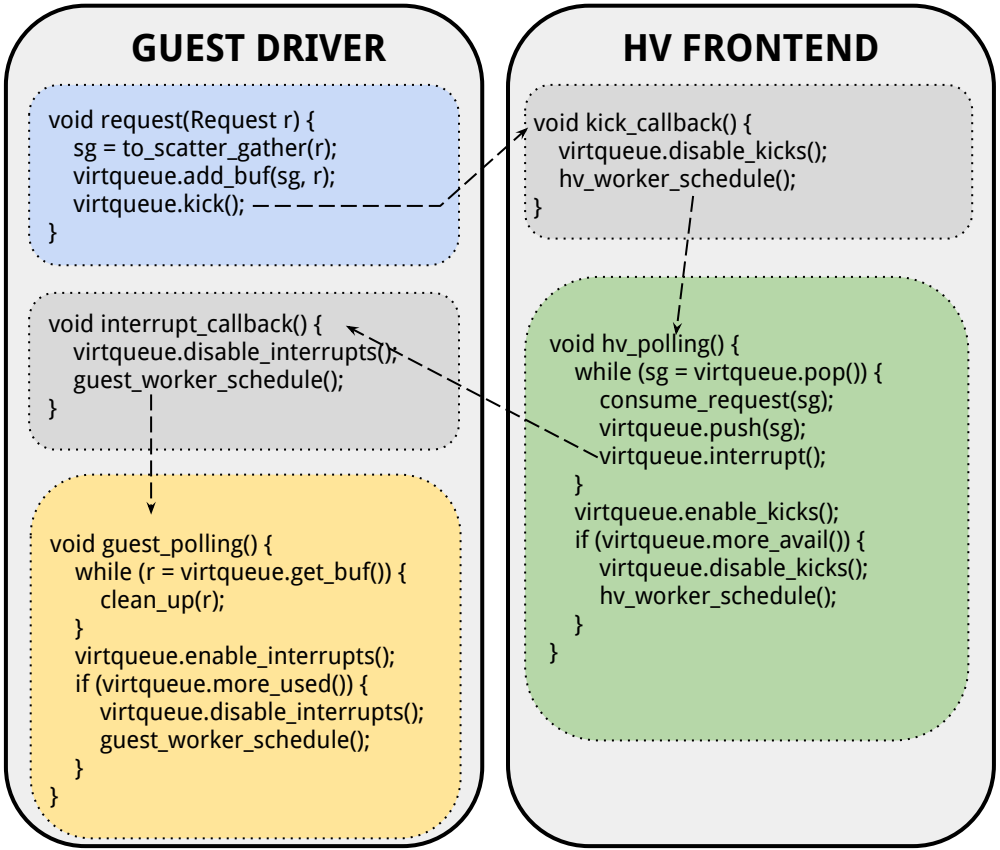
Minimizing notifications (3)

Wrapping up the previous considerations, we can build a producer/consumer system that tries to minimize notifications and perform processing in parallel.

The blue thread produces requests in parallel to the green thread consuming them. The green thread returns consumed requests in parallel to the yellow thread cleaning them up.

While the threads run, notifications are disabled.

This scheme is general and can be applied to any paravirtualized I/O device.

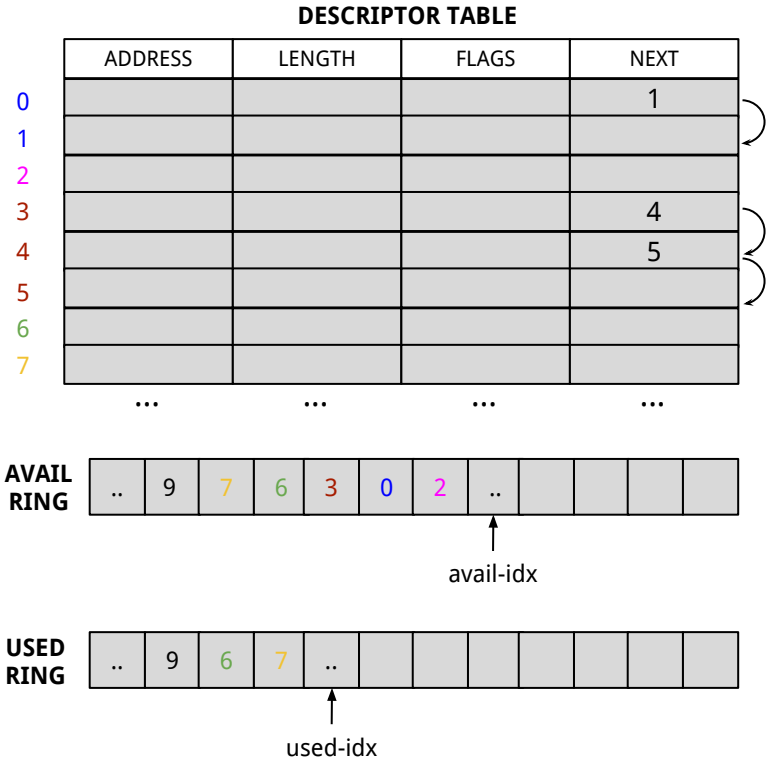


The VRing implementation (1)

Not surprisingly, the current VQ implementation is ring-based. To cope with multi-fragments SGs, out-of-order SGs consumption, and optimize cache usage, a VQ is more complex than a NIC ring (e.g. and e1000 TX ring).

A VQ is implemented with three shared data structures: the descriptors table, the *avail* ring and the *used* ring.

Other data structures are used, but they are private to the guest or the HV.



The VRing implementation (2)

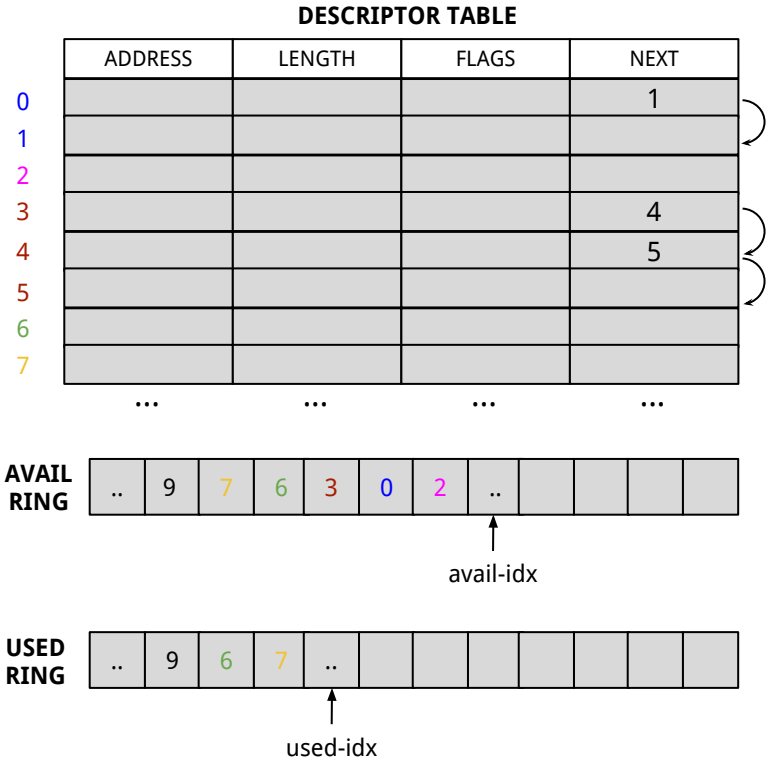
A VRing descriptor is not very different from the e1000 TX or RX ones. It contains

- the physical address and length of a buffer
- a next field for descriptor chaining
- flags (e.g. to indicate if the buffer is for input or output).

An SG entry with *N* fragments is mapped into *N* descriptors.

The descriptor table is written by the guest only, with the HV only reading it. This removes cache invalidation effects between producer and consumes threads.

Descriptors in the table are not necessarily used sequentially (like e1000 rings) because the consumer may process produced SGs out-of-order.



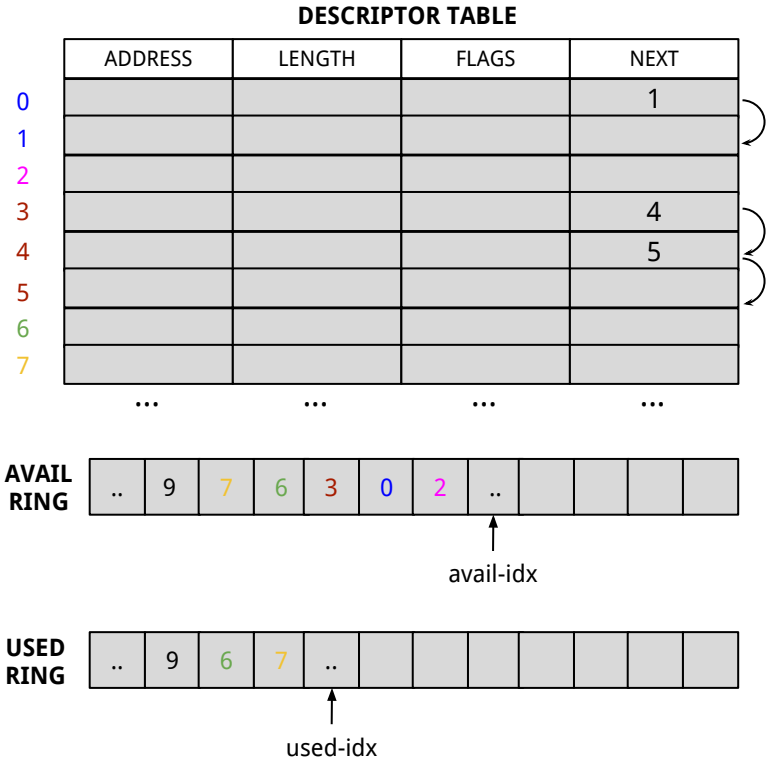
The VRing implementation (3)

The avail ring it's an *indirect* ring used by the guest to expose produced SGs.

Each slot contains an *head*, which is the index - in the descriptor table - to the first descriptor of an SG list produced by the guest.

The avail ring also contains a free running index, the *avail-idx*, that indicates the next avail slot to use. It is incremented by one for each produced SG list. The avail-idx has a similar purpose as e1000 TDT/RDT registers, but is stored in memory (principle C).

The descriptor table is written by the guest only, with the HV only reading it.



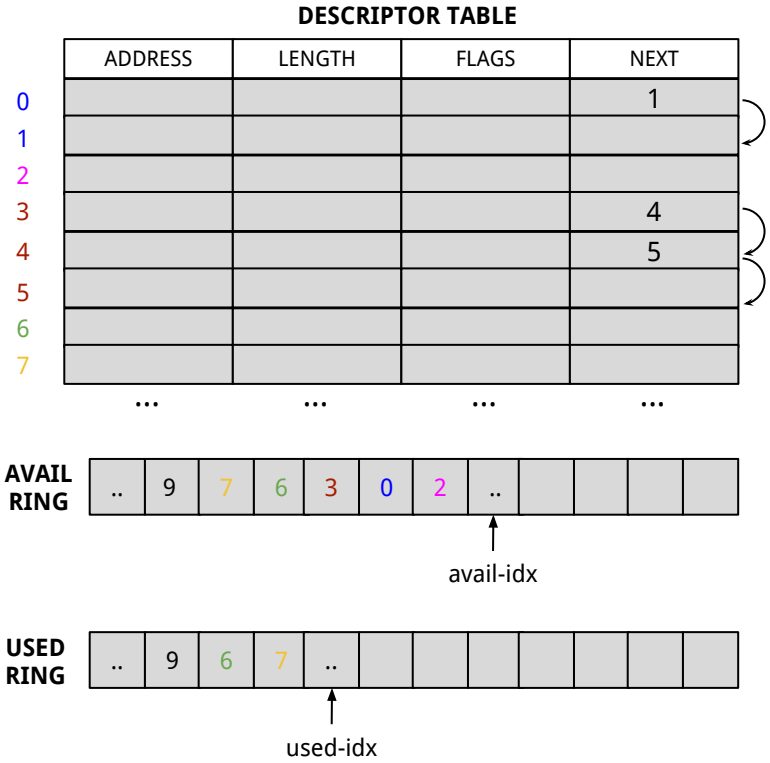
The VRing implementation (4)

The used ring it's an *indirect* ring used by the HV to return consumed SGs to the guest.

Each slot is a pair, containing the head of a consumed SG (taken from the avail ring) and the a length field. For an output SG, the length field is 0. For an input SG it is the total length written into the SG buffers.

The used ring also contains a free running index, the *used-idx*, that indicates the next used slot to use. It is incremented by one for each consumed SG list. The used-idx has a similar purpose as e1000 TDH/RDH registers, but is stored in memory (principle C).

The descriptor table is written by the HV only, with the guest only reading it.

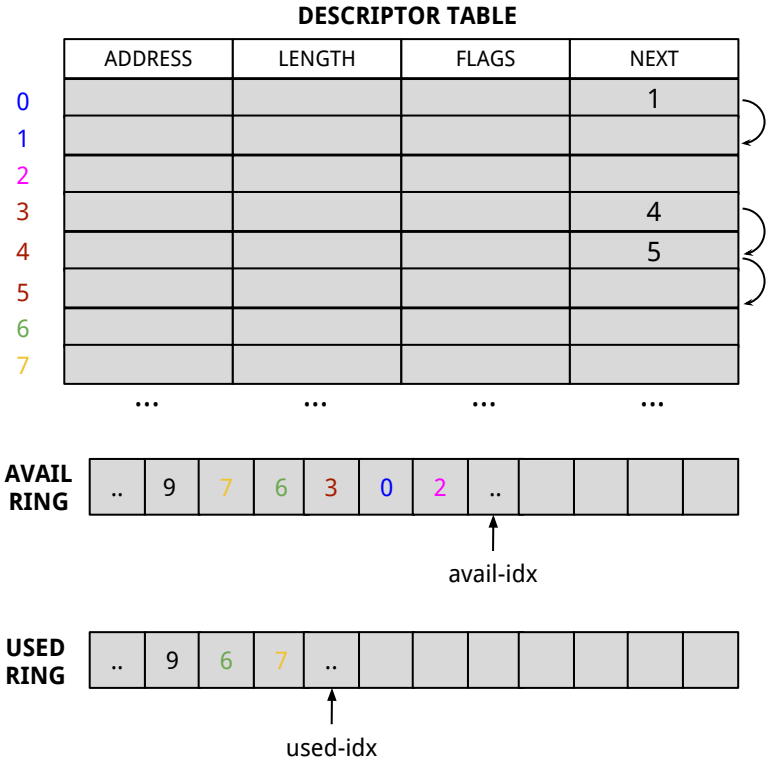


The VRing implementation (5)

VQ `add_buf` implementation (guest):

- Put the SG list into free descriptor table entries, chaining them through the next fields. If not enough descriptors are available, return an error. Free descriptors are chained together for efficient allocation/deallocation.
- Put the head of descriptor chain in the avail ring (in the slot indexed by avail-idx) and increment avail-idx.
- The token is stored into a private ring parallel to the avail ring.

The avail ring cannot wrap around, because the number of ring slots is the same as the number of descriptors, and allocation of free descriptors happens before updating the avail ring.

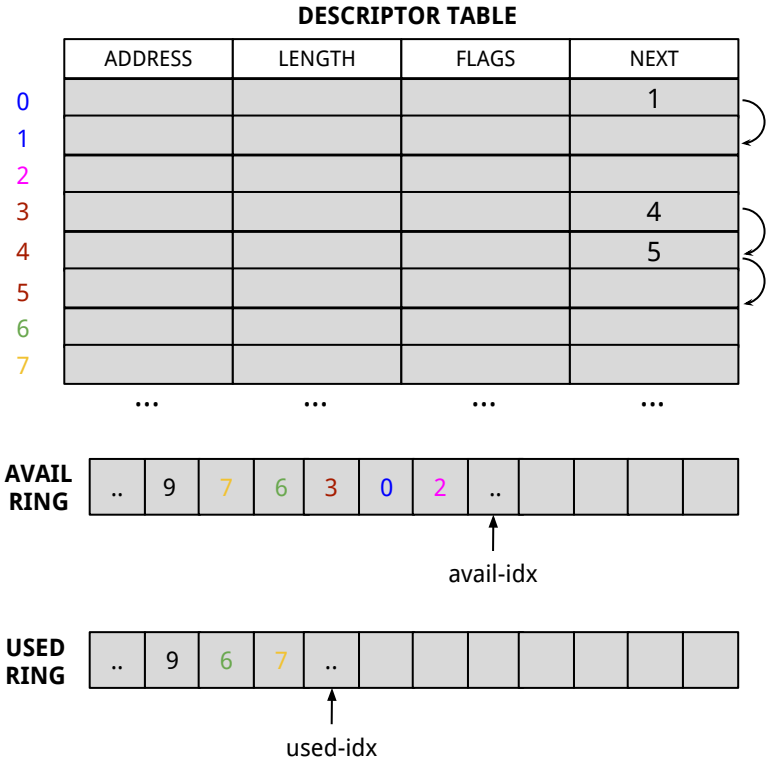


The VRing implementation (6)

VQ pop implementation (HV):

- Read the next slot in the avail ring. An internal index is kept to track the last processed slot.
- Read the referenced SG list from the descriptor table and return it to the caller.

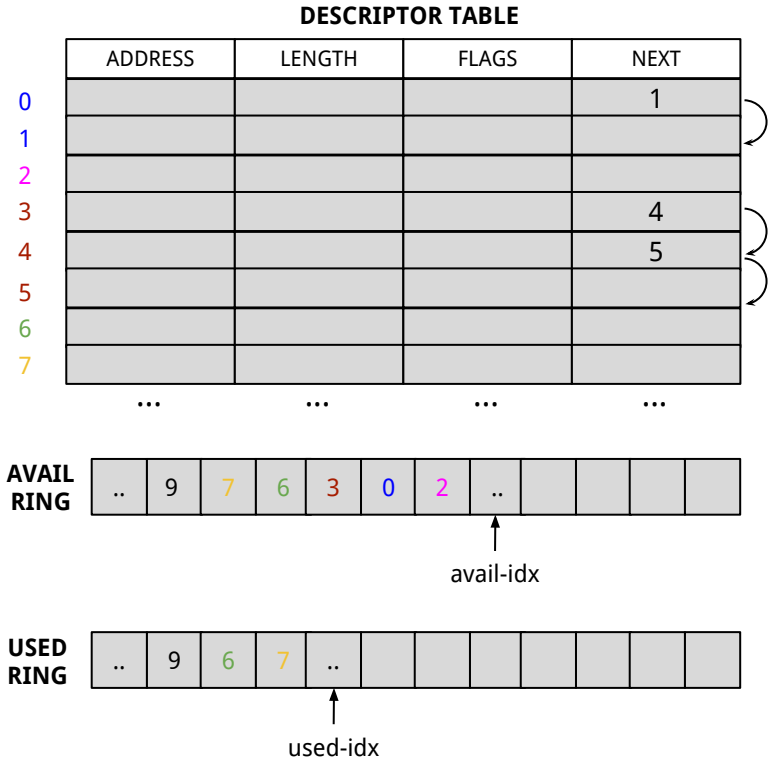
Note that the HV does not modify the descriptor table.



The VRing implementation (7)

VQ push implementation (HV):

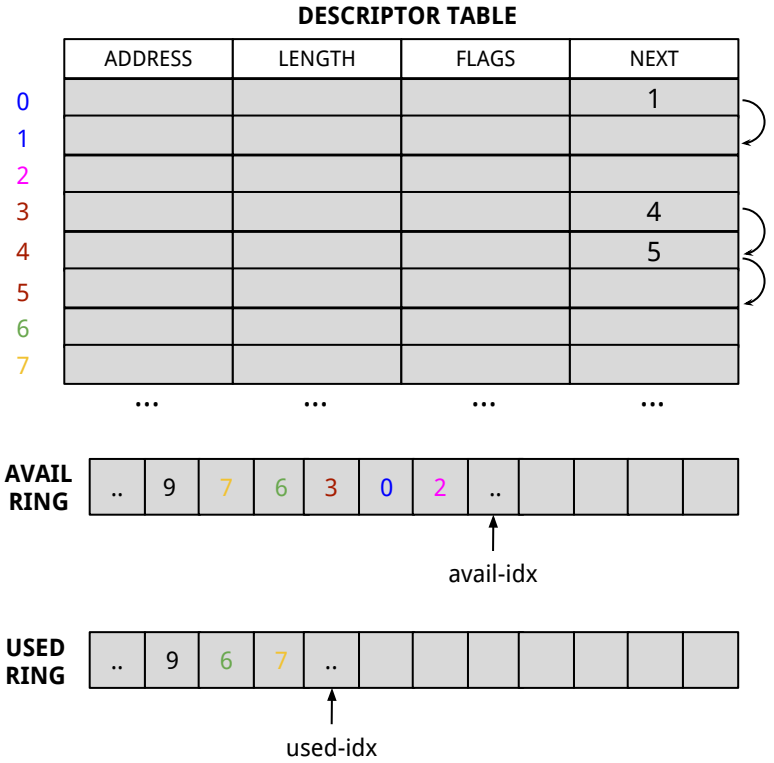
- In a real implementation, this method receives as an argument the head of the descriptors chain representing a consumed SG, instead of the SG itself.
- Put the head in the next slot used ring (in the slot indexed by used-idx) and increment used-idx.



The VRing implementation (8)

VQ `get_buf` implementation (guest):

- Read the next slot in the used ring. An internal index is kept the track the last processed slot.
- Deallocate the descriptors for the used SG list, so that they can be allocated again by `add_buf`.
- Return the used length to the caller together with the associated token.



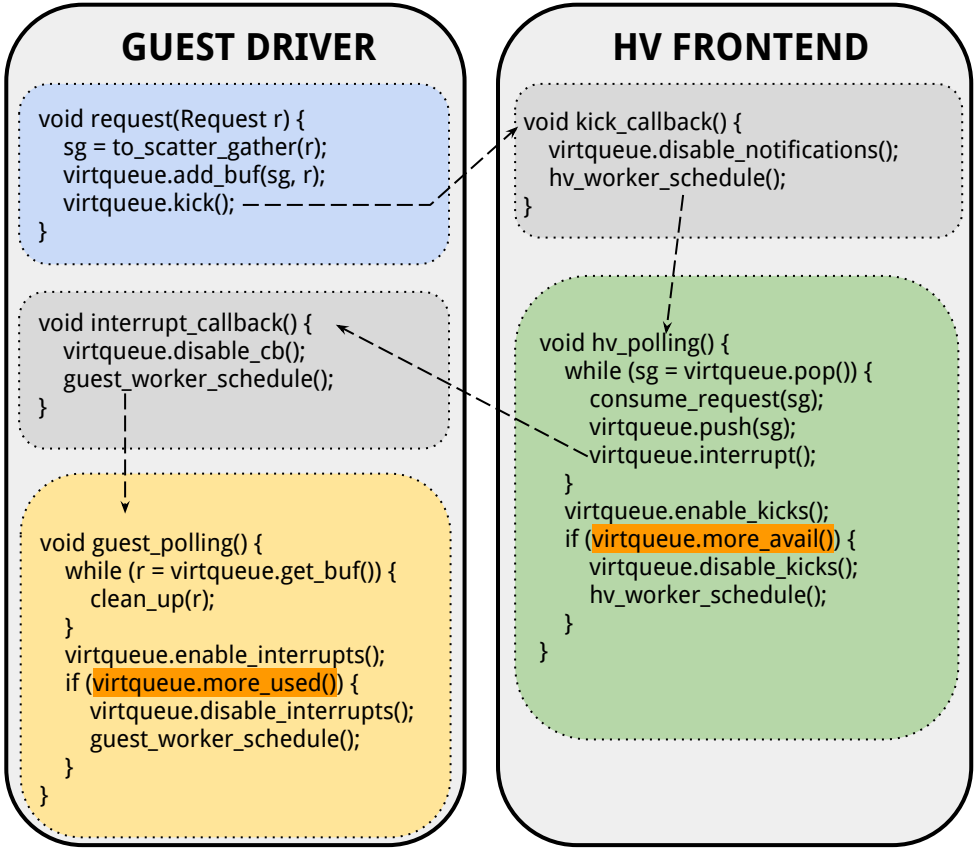
VRing notifications (1)

In addition to slots and avail-idx, the avail ring contains a flag to be used to suppress VQ interrupts. When the producer sets this flag, the consumer *should* not interrupt, and in fact the VQ `interrupt` method has no effect in this case.

A similar flag stored in the used ring is used by the consumer to suppress VQ kicks. When the consumer sets this flag, the producer *should* not kick, and in fact the VQ `kick` method has no effect.

These are a very useful optimization, but the producer/consumer cannot count on it, notifications may still arrive.

But why do we need to double check (highlighted code)?



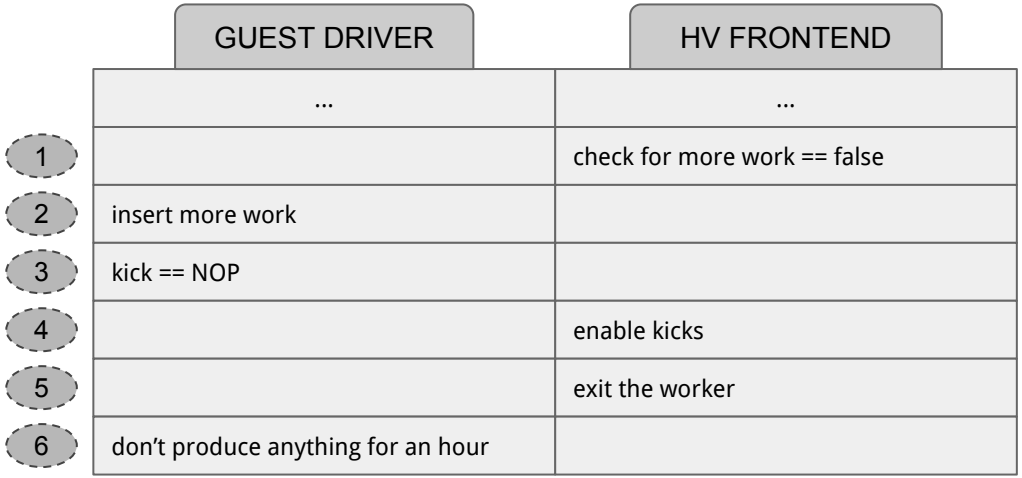
VRing notifications (2)

A race condition is lurking over there.

Assuming consumer does not double check, here is an example of what can happen. Consumer exits the polling cycle (1), but new data is produced (2) before consumer has the chance to enable kicks again (4).

If producer stops for a while (e.g. no more network packets to transmit), the request submitted at (2) does not get executed until next kick.

To solve this, consumer has to double check for more work after enabling VQ kicks.

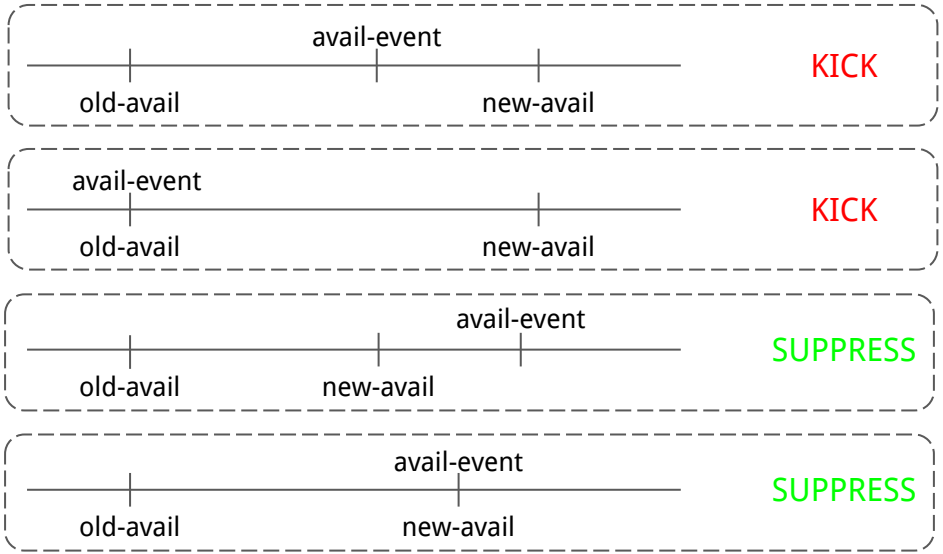


VRing notifications (3)

VirtIO defines an optional advanced feature for VQ kick suppression, replacing the flag-based suppression, referred to as *event-idx*. At the end of the used ring, the consumer stores an *avail-event-idx* field, to indicate *when* the next kick is desired. The producer should kick when *avail-idx* goes beyond *avail-event-idx* from the last time kick was invoked, which means that the producer has just filled the avail slot at index *avail-event-idx*.

In the examples, the producer has called `add_buf` many times, so that avail-idx transitions from old-avail to new-avail since the last time the `kick` method was invoked.

A dual interrupt suppression mechanism is also defined, based on an *used-event-idx* field stored at the end of the avail ring.

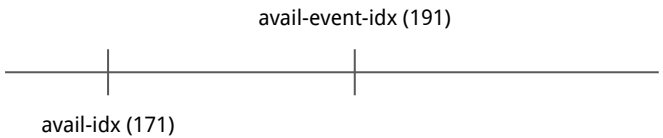


VRing notifications (4)

Why is event-idx useful? The avail-event-idx may be used by the consumer to ask for kicks as soon as there is more to consume, by setting avail-event-idx to the current value of avail-idx*. By doing this, we obtain the very same behaviour of of flag-based suppression, with the advantage that the consumer does not have to reset the flag in the VQ kick callback.

However, event-idx may be used to ask for delayed notifications. If the consumer sets `avail-event-idx = avail-idx + 20`, the producer will kick only when 21 more SGs will have been produced. In this way, the notification cost is amortized over 21 requests.

Consequently, event-idx is an additional mechanism to suppress notifications. Should we always use delayed notifications? Not in practice, we should use them only when we are sure latency will be under control.

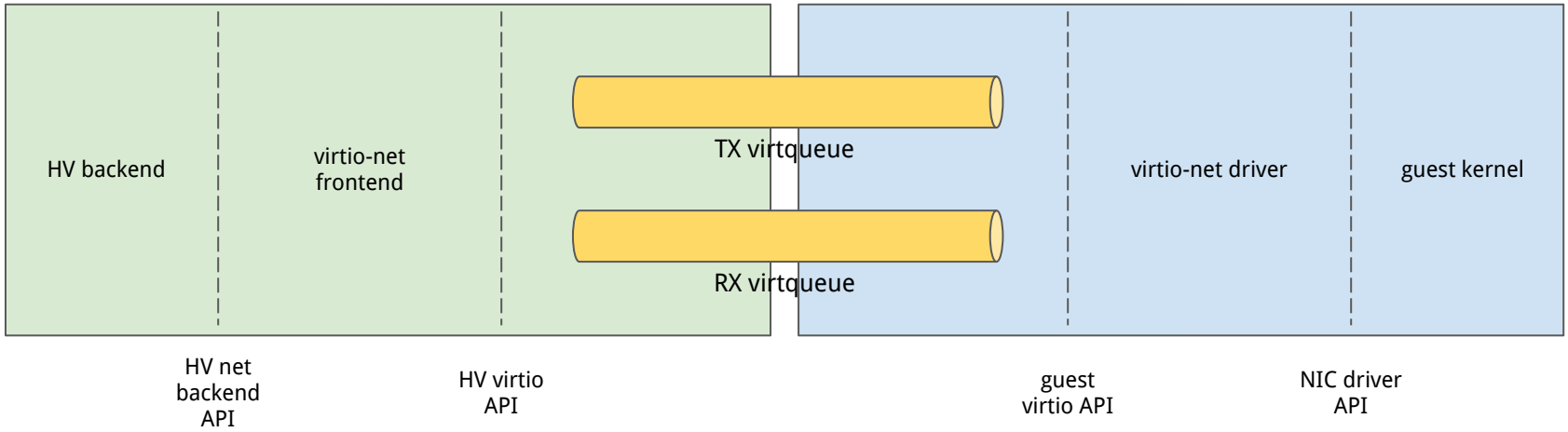


(*) Double check is still necessary to avoid race conditions.

Virtio networking

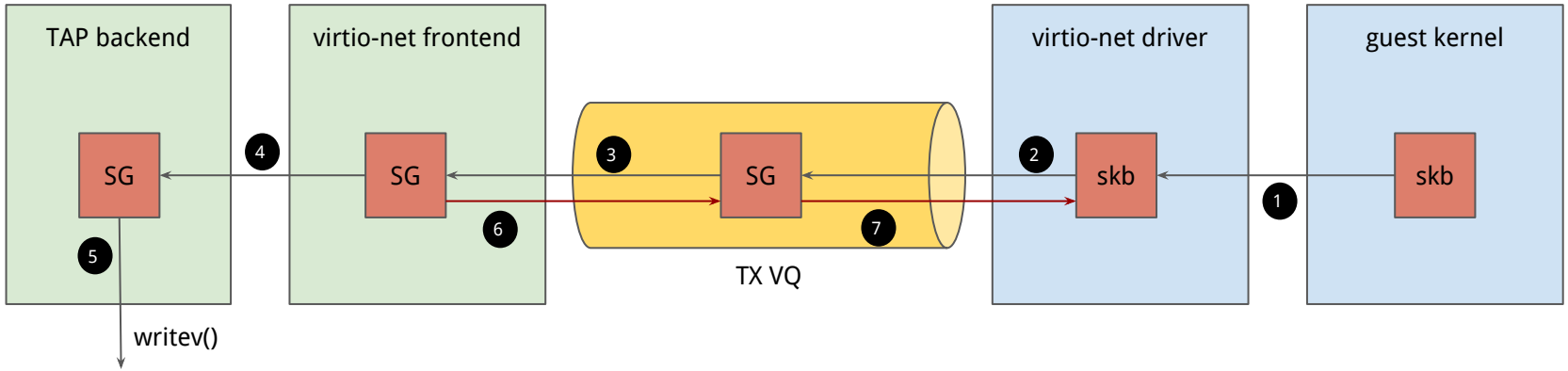
By means of a solid I/O paravirtualization scheme, a VirtIO network adapter (virtio-net) can be designed. The workflow of driver and front-end is not much different from the one used with e1000, but here we will also apply our paravirtualization principles.

Let's make a concrete example, assuming Linux as guest OS, and QEMU as Hypervisor. The virtio-net adapter has at least one VQ for TX and one VQ for RX. More TX/RX VQ couples can be negotiated at initialization time, to spread network processing over multiple CPUs.



Virtio-net transmission

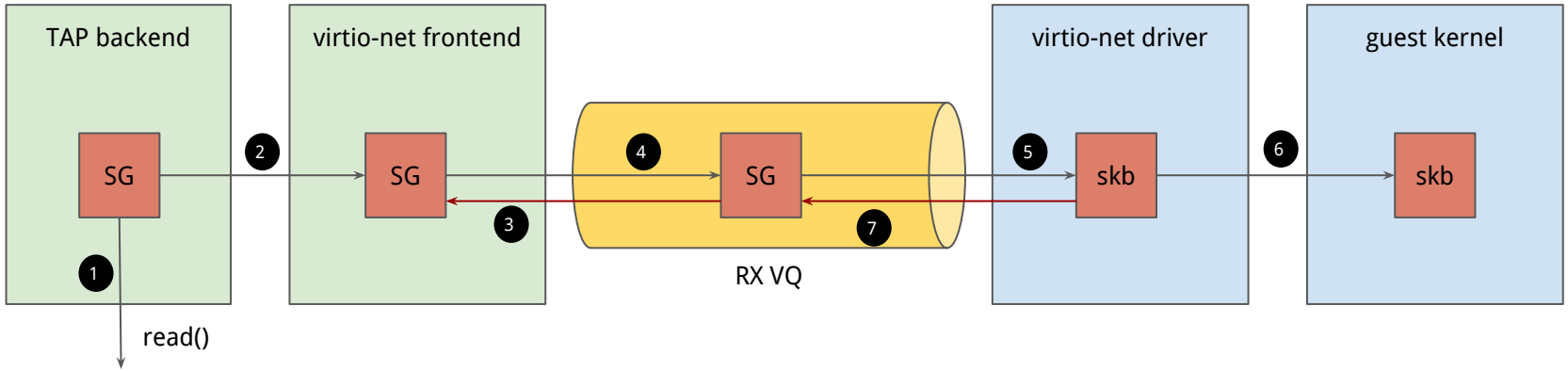
The virtio-net driver translates each `skb` objects into a multi-fragment SG that matches the packet fragments. A GSO packet (up to 64KB) may need up to 17 VirtIO descriptors. The QEMU virtio-net front-end consumes a virtio-net TX SG list by passing it to a net backend (e.g. a TAP device). The TX consumer runs into a deferred context within the QEMU IO Thread. Driver performs TX cleanup operations of previous TX SGs already consumed (`get_buf`) opportunistically, right before calling the `add_buf`. This is possible because TX cleanup does not affect latency. Since `get_buf` and `add_buf` are called by the same thread, this strategy does not require locking the TX VQ.



Virtio-net reception

At initialization time, the virtio-net driver pre-allocates `skb` objects and `add_buf`s their internal buffer to the RX VQ.

The QEMU virtio-net frontend uses (consumes) an RX SG list to copy in a packet received from the net backend, and pushes it back to the driver. On interrupt, the driver defers the `get_buf` work to the NAPI kernel thread, where consumed `skb` objects are pushed up to the kernel stack and replaced by freshly allocated `skb`s (with new `add_buf` calls). Similarly to the transmission datapath, `add_buf` and `get_buf` are called by the same thread, therefore no locking is required on the RX VQ.

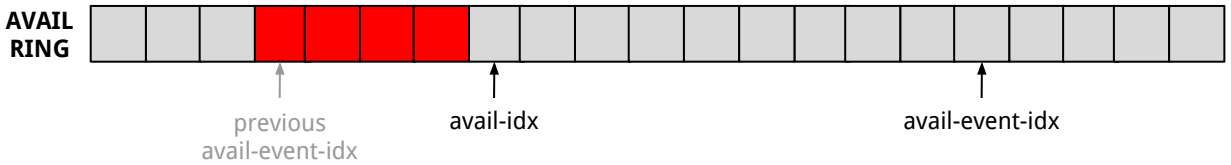


Virtio-net and delayed notifications (1)

The virtio-net driver does not normally know when the guest kernel will ask to transmit the next packet. Therefore, the virtio-net frontend cannot use avail-event-idx to ask for delayed TX kicks without incurring in unbounded latency (TX SGs stall indefinitely in the VQ).

Similarly, the virtio-net frontend does not know when the HV net backend will receive the next packet. Therefore, the virtio-net driver cannot use used-event-idx to ask for delayed RX interrupts, since RX used SGs may stall indefinitely in the VQ.

In the example, red TX SGs may stall in the VQ indefinitely, since nobody knows when avail-idx will advance beyond avail-event-idx.

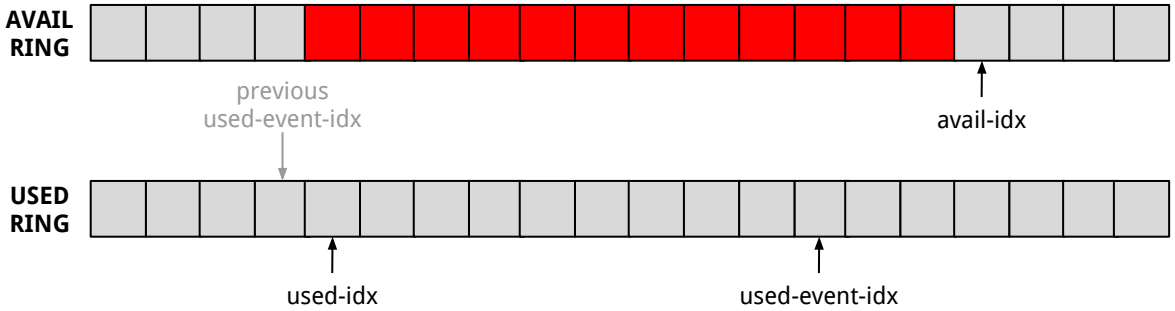


Virtio-net and delayed notifications (2)

Conversely, delayed TX interrupts are normally used, since the driver can assume the frontend will process pending TX SGs as soon as possible, and therefore knows that it will receive an interrupt soon. Moreover, introducing a bit of controlled delay on the TX cleanup processing does not affect latency.

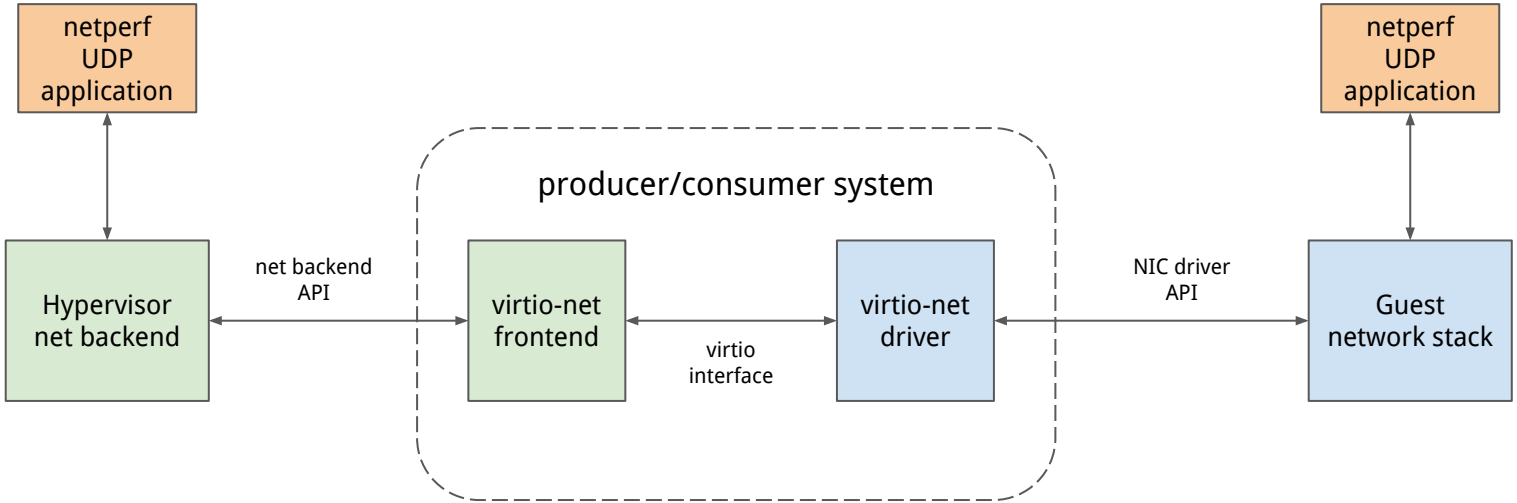
This strategy is useful when TX producer is faster than TX consumer. In this situation, without delayed TX interrupts, descriptor table is always almost full (just one or a few descriptors available), so that producer fills the free one(s) and goes to sleep, waiting for a VQ interrupt. When an interrupt comes, only one or a few descriptors have been used, and the cycle restarts. On average, we have thus approximately one TX interrupt per TX SGs, which is bad. If we use used-event-idx, we can tell the consumer to delay the TX interrupt until enough SGs have been consumed, so that the interrupt is amortized over many slots.

In the example, red TX SGs are pending in the VQ. Producer sets used-event-idx in such a way that consumer will interrupt once 10 SGs will have been used.



Virtio-net performance evaluation

We now redo the same performance *unit tests* done with e1000, to check that paravirtualization principles lead to a better producer/consumer system.



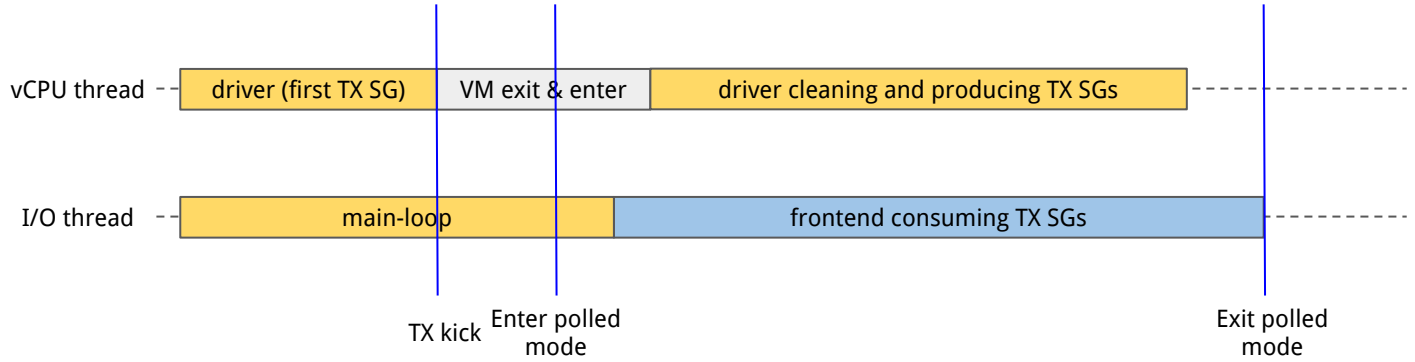
Virtio-net transmission throughput

Performance benefits from parallel frontend TX processing, TX kick suppression and delayed TX interrupts.

For tests over 1.5 KB e1000 performs TSO emulation, while virtio-net just passes the TSO packet unaltered to the TAP.

Paravirtualization principles lead to huge improvements over e1000.

UDP payload	Packet rate	Throughput	Improvement over e1000
20 B	270 Kpps	43.3 Mbps	17.3 x
500 B	270 Kpps	1,08 Gbps	8.5 x
1.5 KB	240 Kpps	2.9 Gbps	14 x
15 KB	162 Kpps	19.5 Gbps	44 x
64 KB	70 Kpps	36.6 Gbps	76 x



Virtio-net receive throughput

The table shows measurements for the same reception experiments performed over e1000. Performance benefits from lighter interrupt management, interrupt suppression, and RX kick suppression.

For short packets, netperf receiver is able to perform better with e1000. This is not contradictory, it's a consequence of livelock. Higher sender rates means more wasted work, and so less time for the receiver to drain its socket receive queue. So virtio-net loses because it's faster.

In general, a good improvement, but e1000 is still competitive thanks to NAPI.

UDP payload	**Sender rate**	**Improvement over e1000**	**Receiver rate**	**Improvement over e1000**
20 B	1010 Kpps	+ 36%	560 Kpps	- 7%
500 B	970 Kpps	+ 38%	430 Kpps	- 28%
1.5 KB	870 Kpps	+ 85%	382 Kpps	2.8 x
15 KB	460 Kpps	3.6 x	143 Kpps	9.5 x
64 KB	204 Kpps	6.2 x	38.3 Kpps	2.3 x

Virtio-net latency

The virtio-net interrupt routine (differently from e1000), does not require lots of register accesses. In the most legacy configuration, one access is necessary to acknowledge the interrupt. However, in practice, virtio devices use per-queue MSI-X interrupts, that do not require a register access as acknowledge. This greatly improves latency.

To be fair, since consumer (HV) TX processing happens in a different thread than the producer (guest) TX processing, on TX kick virtio-net has to pay the overhead context switch (if the consumer was not already running), while e1000 doesn't.

Ping-pong experiments (netperf UDP request-response tests) reports a maximum transaction rate of about 29000. Thus the average round trip time between an application running on the guest and one running on the host is about 35 *us*, which is roughly half of what measured with e1000.

Some references

1. Rusty Russell - virtio: Towards a De-Facto Standard For Virtual I/O Devices
2. Virtio 1.0 standard. Available online at http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf
3. Speeding Up Packet I/O in Virtual Machines. Available online at http://info.iet.unipi.it/~luigi/papers/20130903-rizzo-ancs.pdf
4. The QEMU project. http://www.qemu.org.