A deep dive into Linux Networking

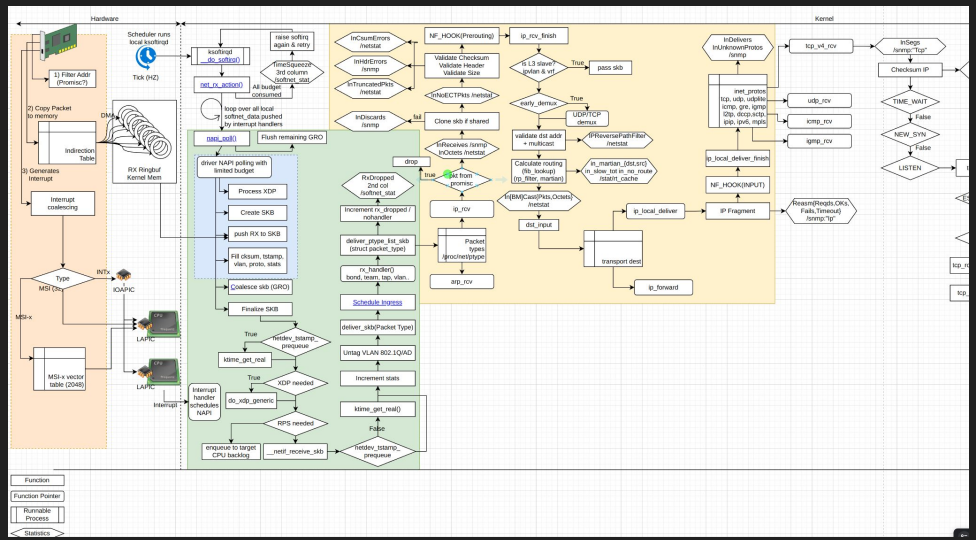
Mohamed Elsakhawy

Agenda

- Why this presentation ?
- The network stack
- Core concepts
- The NIC driver
- The network stack structs
- The transmit and receive paths
- The challenge of small packets

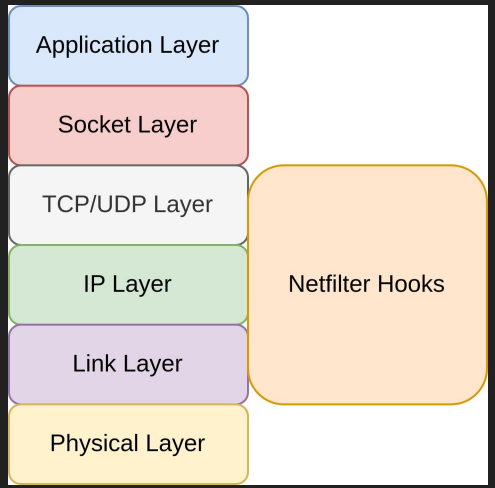
Why this presentation

- It’s complicated
  - Very complicated
  - https://github.com/Saruspete/LinuxNetworking ⇒ Adrien Mahieux
  - That’s just part of it
- As a Linux engineer
  - You don’t need all of that
  - But knowing always helps



The Linux Network stack

- Responsible for Network transmit/receive
- Provides an abstraction layer
  - Reachable by syscalls
- Can be divided into 7 components
- Packets flow down
- Netfilter is engaged using hooks



Concepts

- User space vs Kernel space
- NIC TX and RX rings
- IRQ
- DMA
- NAPI vs Interrupt driven receive
- RPS and RSS
- RFS
- QDISC

User space vs Kernel space

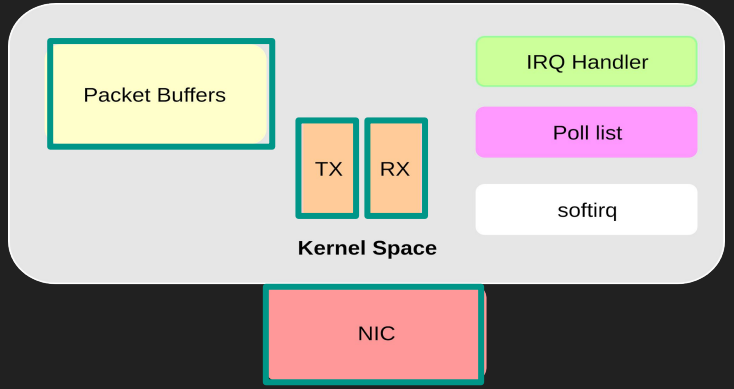
- Logical segregation of privilege
- Application run in user space
  - Limited access
- NIC drivers run in the kernel space
  - Privileged mode
  - Direct access to hardware
  - Memory management, CPU..etc
  - Access to memory regions
    - controlled , e.g DMA

IRQs and softIRQs

- Interrupt request
  - Has Associated Interrupt Service Routines (ISR)
- ISRs essentially
  - Specify how to handle this interrupt
- Interrupts have a problem
  - The CPU stops doing what it’s doing
- Comes the softirq (raised by the kernel itself)
  - Not blocking

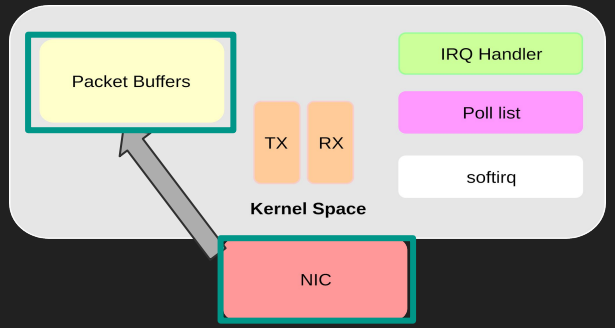
NIC TX and RX rings

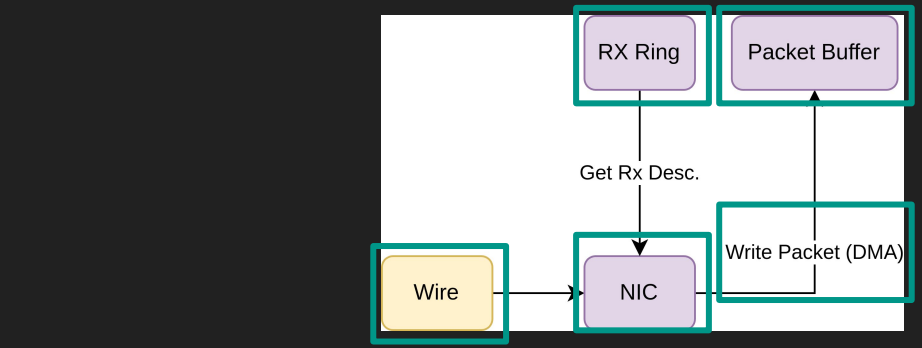
- Have pointers to start/end addresses of packet contents
  - Not the packet data
  - Hence they are called “descriptor rings”
- Packet contents are in packet buffers
- The memory areas DMA’able



DMA

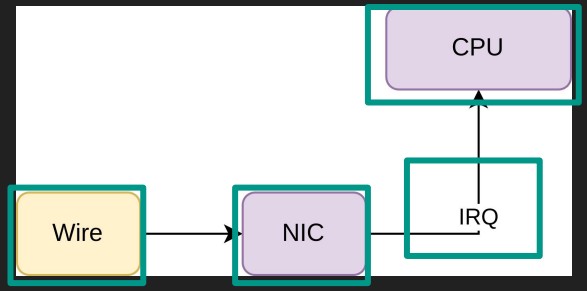
- The DMA engine
  - RX ring contains pointers to RAM locations to write the packet contents to
  - We saved the host CPU, DMA is executed by the DMA controller

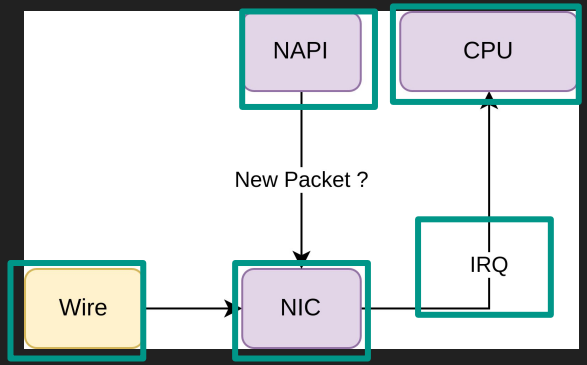




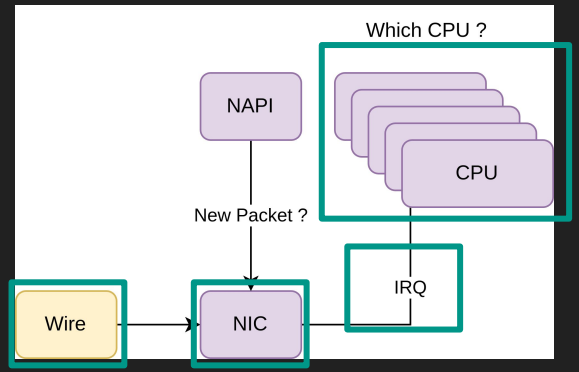
NAPI and Interrupt driven receive

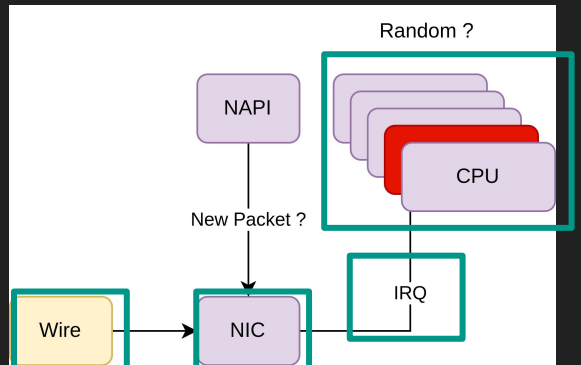
- Before NAPI ⇒ Interrupt per packet
  - High overhead
- NAPI ⇒ New API
  - Interrupt and polling
  - An interrupt generated at first packet
    - Interrupts disabled
    - Device put into polling mode
    - No packet
      - Interrupt re-enabled
- In a single queue NIC
  - By default, the CPU that services the interrupt
    - Processes the packet up the network stack
    - That is a problem !





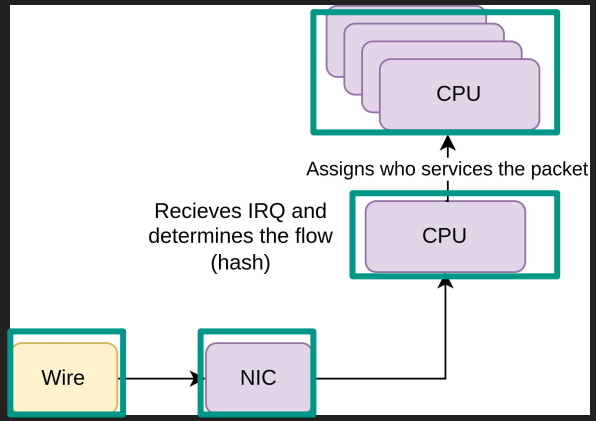
- Systems have more than one core
- And NICs have multiple queues
- Better performance can be achieved by
  - Assigning more cores to handle the multiple queues
  - This can achieve better performance ?
    - But …..
      - Order of the packets is important
- We need to assign flows to CPU cores
  - At a high level
    - Same source, same destination





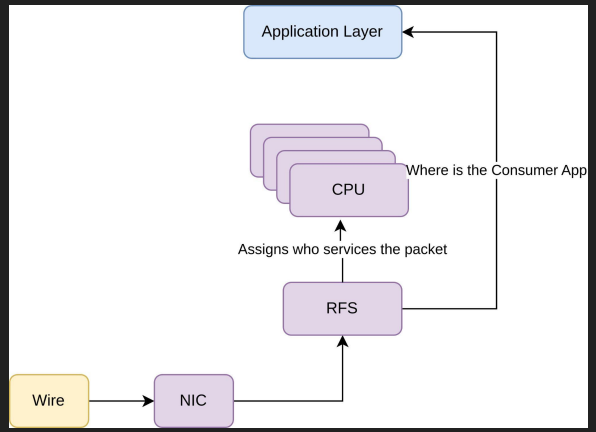
RPS and RSS

- RSS: Receive-Side Scaling
  - multi queue-receive
  - NIC has multiple queues
  - Hash computation / Flow determination on the NIC
- RPS: Receive Packet Steering (RPS)
  - Directs packets to specific CPU for processing
  - One CPU receives the interrupts and computes the hash
    - Determine the flow, and the CPU that services the packet



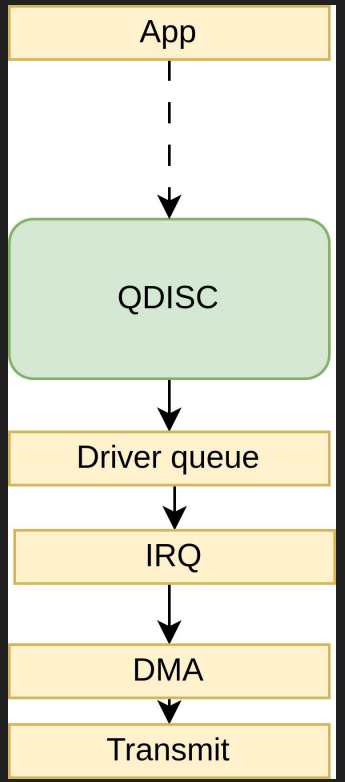
Receive Flow Steering

- Goes a step further
- RFS tracks consumer app in the userspace
  - Associates with flow
- Flow processed on the same CPU core that the consumer app runs on
  - High cache hit
  - Better network performance



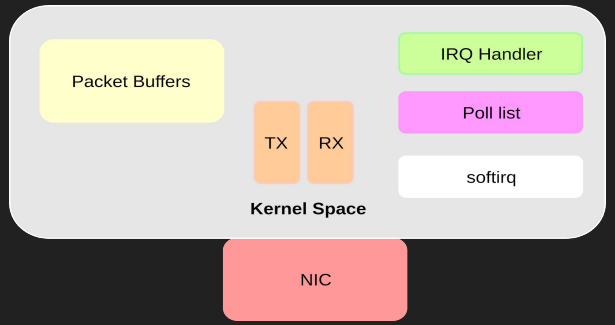
QDISC

- Part of the Traffic control (TC)
  - Packet transmission
- Queueing disciplines
  - Think of it a policies for packet transmissions
  - FIFO is a policy
    - And the most common one



The NIC driver

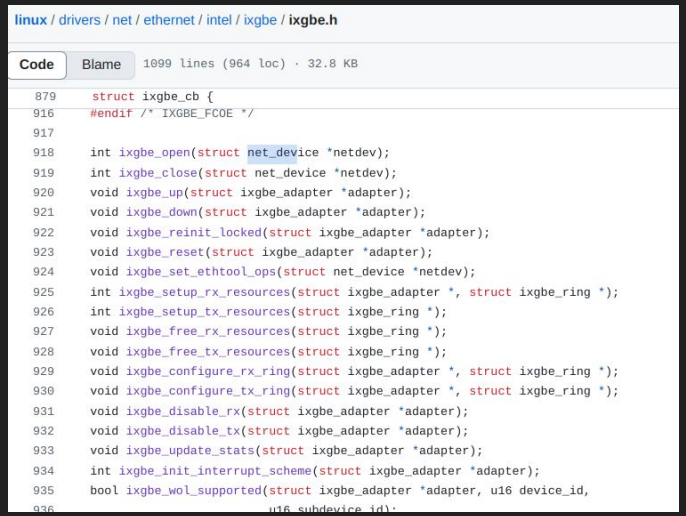
- The NIC driver
  - Runs in the kernel space
- Three core functions
  - Allocate RX/TX queues
    - RX/TX descriptor rings
  - Initialize the NIC
    - Set the value of the NIC registers properly
  - Handling interrupts
    - For TX and RX
    - register “Interrupt service handlers/routines”



The structs

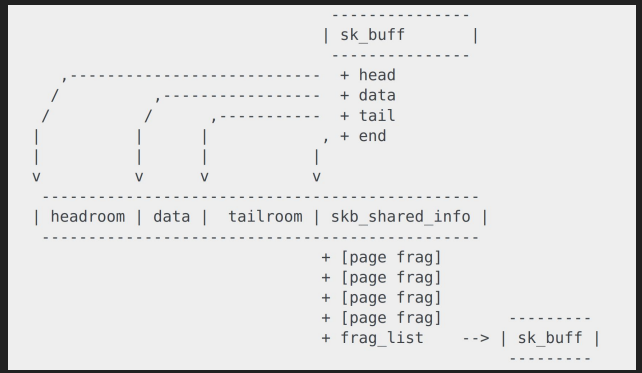
- The *net_device struct*
  - struct used to define a network device in the kernel
  - Defined at the kernel side

https://github.com/torvalds/linux/tree/master/drivers/net/ethernet/intel/ixgbe



The sk_buff struct

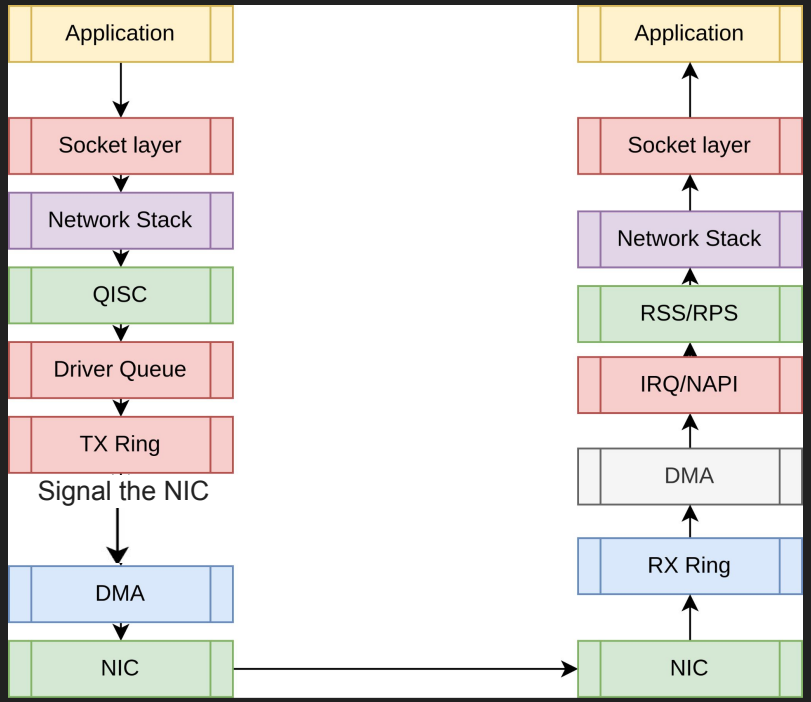
- The main structure representing a packet
  - Only packet metadata , not the contents
  - Exists at the implementation of all layers of the OSI model
    - Unique
    - Pointers to next sk_buff
    - Shared between network stack layers
- So who creates the first sk_buff for a packet ?
  - You guessed it right , the driver !



https://docs.kernel.org/networking/skbuff.html

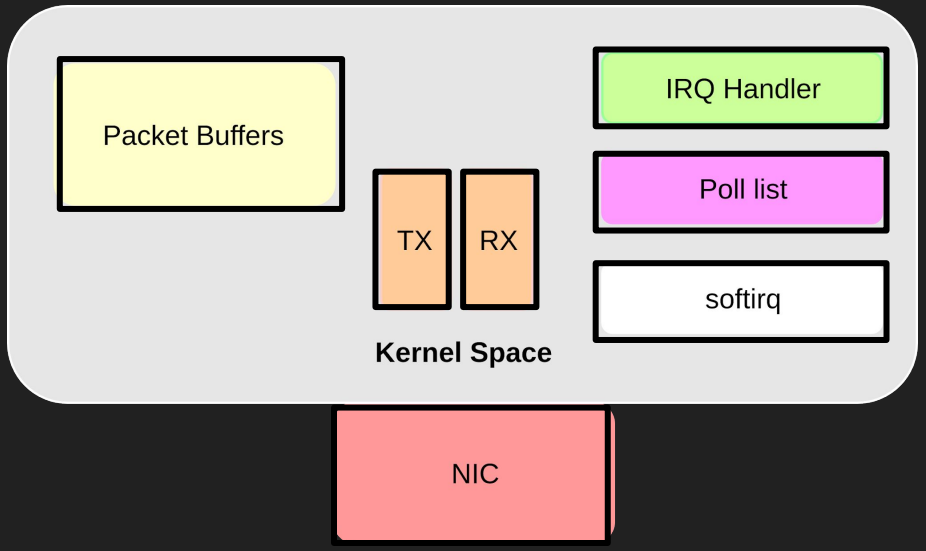
The transmit vs receive paths

-



NIC Receive

- When a NIC receives a packet:
  - Fetch a descriptor
  - DMA Packet to Packet buffers
  - RX Interrupt
  - Poll_list
  - RPS/RFS
  - Softirq



The transmit path

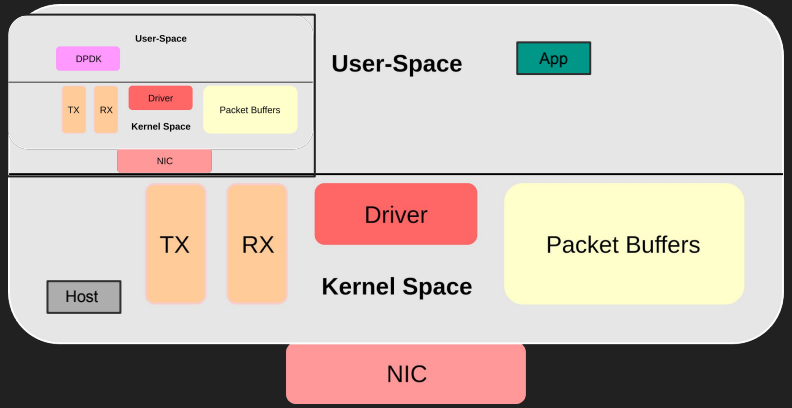
- Application runs in userspace
- When a packet is ready to be sent, the *sendto* syscall is used
- sk_buff created, contents set in packet buffer
- QDISC controls the queue
- TX Descriptor ring set
- NIC picks up the TX descriptor
    - DMAs the packet contents from packet buffer
    - Sends packet on wire

The challenge of small packets

- A Packet:
  - Smallest unit in the IP layer
  - Too many packets
    - Lower latency, and less throughput
    - NFV loves small packets
  - Jumbo frames
    - Larger packets
    - Higher throughput and higher latency
- Cost-benefit analysis
- For small packets, we want the least network stack engagement

In a virtualized environment

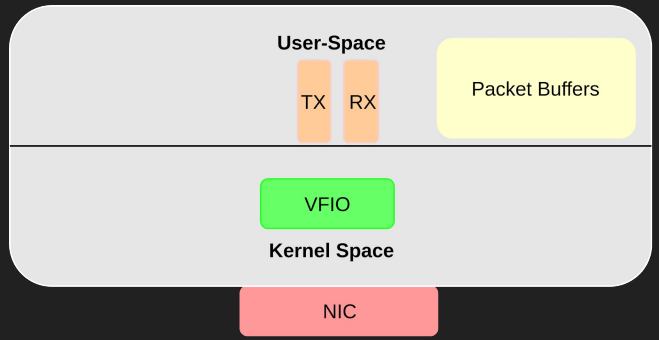
- Network stack on the host and;
- Network stack inside the VM
- Very expensive overhead
  - For every packet
  - And small packets means higher overhead

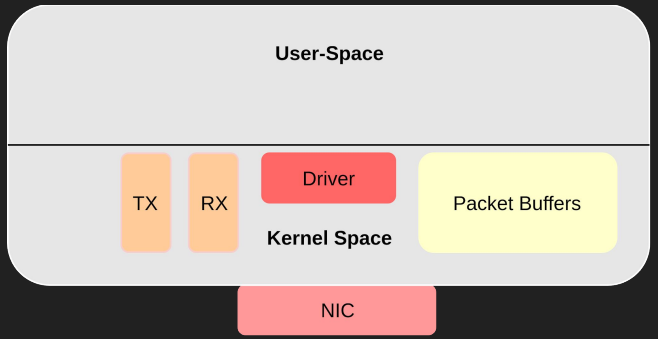


- Saturating a 10-Gbit NIC over 65-byte MTU inside a VM
  - Traditional layout, very hard to do
- Some solutions bypass the host network stack
  - VFIO/IOMMU
- Moving the NIC to userspace
- Something needs to manage the NIC in userspace
- We can’t have a physical NIC for every VM
  - DPDK-enabled OVS
  - SR-IOV

Moving NIC to userspace

- In user-space, we need to:
  - Access NIC's onboard registers
  - Handle Interrupts
  - Allocate memory to DMA from/to
- VFIO & IOMMU
  - Bypass the network stack

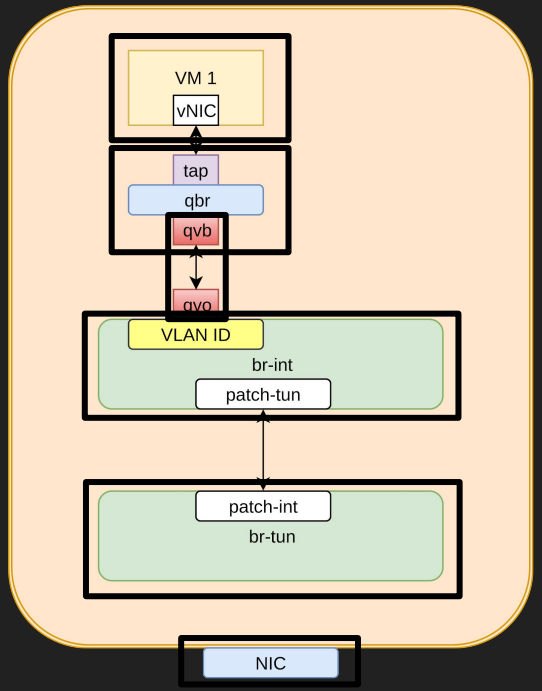


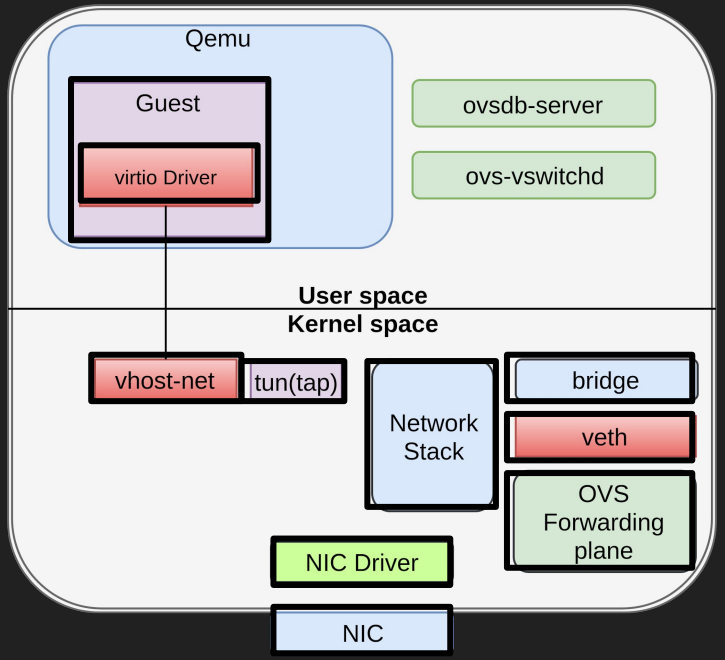


VFIO & IOMMU

- VFIO
  - Passthrough style driver
  - Maps user-space memory regions to DMA
  - Allows direct device access in userspace
  - NIC ownership to user space process
  - Security ?
- IOMMU provides Isolation & Address translation
  - Ensures requestor NIC has rights to DMA memory region
  - DMA regions are Virtual addresses (IOVA)
- VFIO pokes a hole through the kernel space

Traditional layout of VM Network in Openstack

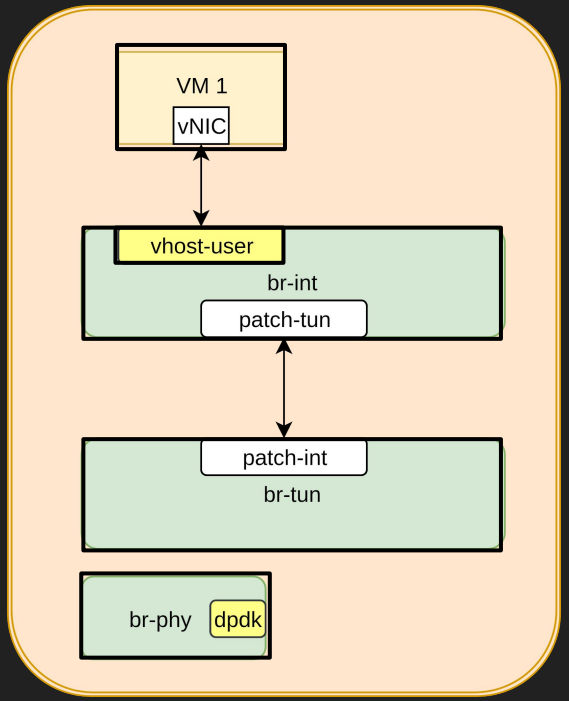


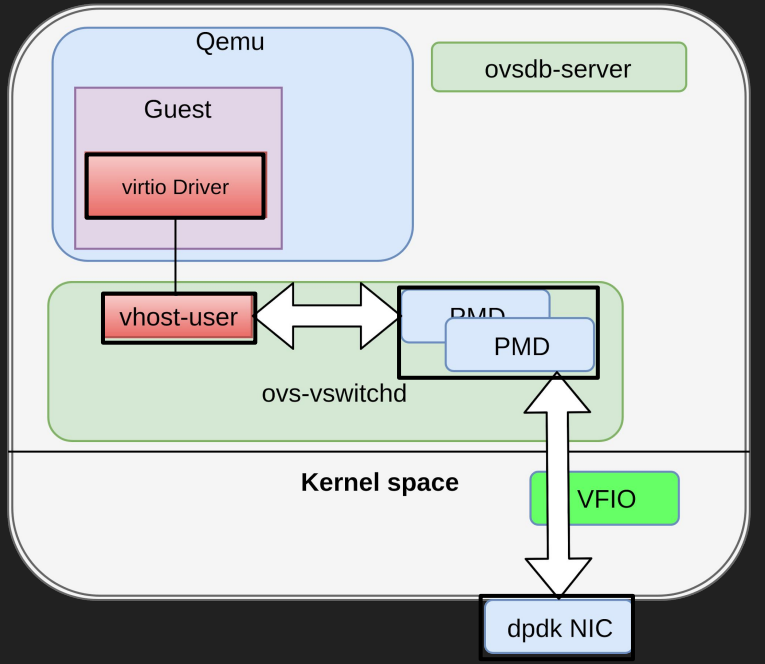


DPDK

- Data plane development kit
- Even with VFIO/IOMMU (VT-d for intel)
  - We can’t have a NIC per application
  - We need something to do the functionality of the network stack
    - But faster
- At a high-level
  - DPDK provides provides a framework for faster packet processing in userspace
    - Customer memory management
    - Polling based sending/receiving
- OVS-DPDK
  - Switching in the userspace
  - Based on openvswitch

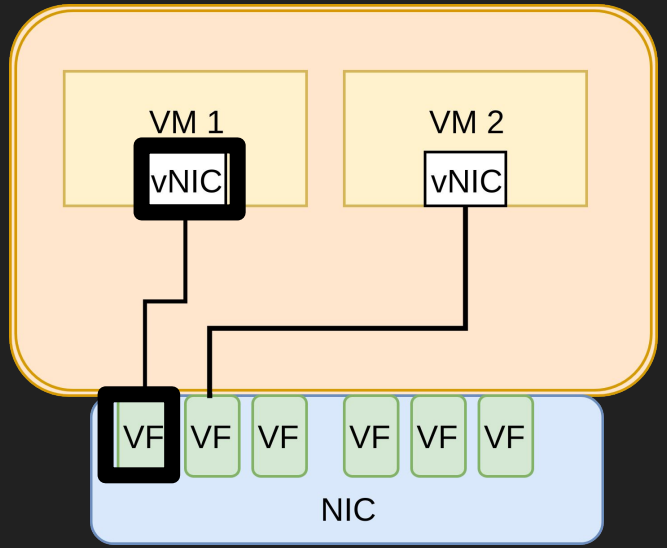
DPDK-enabled Neutron OVS deployment

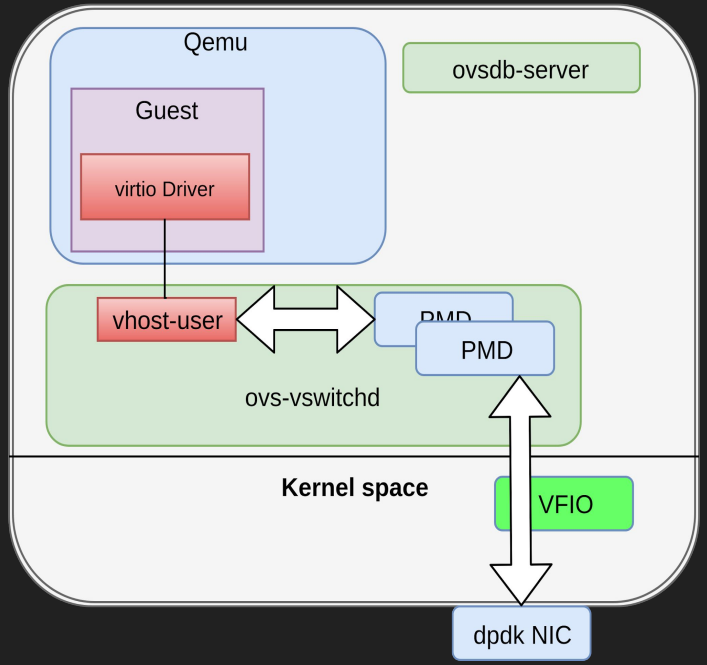


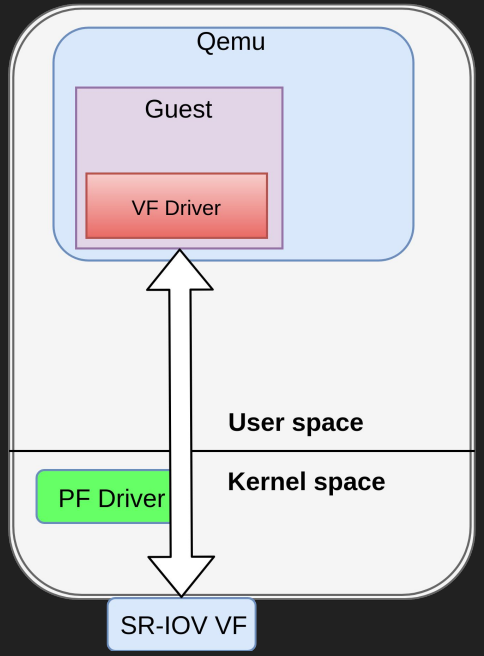


SR-IOV

Virtualization of Input/Output devices
Physical Functions “PF” vs Virtual Functions “VF”
Single PF, many VFs
VMs have direct access to VFs
No host/switching overhead, i.e. no OVS
VM communicates directly with physical network




Qemu
ovsdb-server
Guest
virtio Driver
vhost-user
PMD
ovs-vswitchd
Kernel space
VFIO
dpdk NIC


Qemu
Guest
VF Driver
User space
Kernel space
PF Driver
SR-IOV VF

Thank you, Questions