ACM India Winter School on
Full-stack Networking

Day 1
Understanding Linux Networking Concepts

Rinku Shah, Assistant Professor, IIIT Delhi
Sumit Kumar, Research Scholar, IIIT Delhi
Najiya Naj, Research Scholar, IIIT Delhi
Maryam Tahira, Research Scholar, IIIT Delhi

Brief Bio

Rinku Shah
Assistant Professor, IIITD
rinku@iiitd.ac.in
https://faculty.iiitd.ac.in/~rinku/

Sumit Kumar
Research Scholar, IIITD
sumitk@iiitd.ac.in

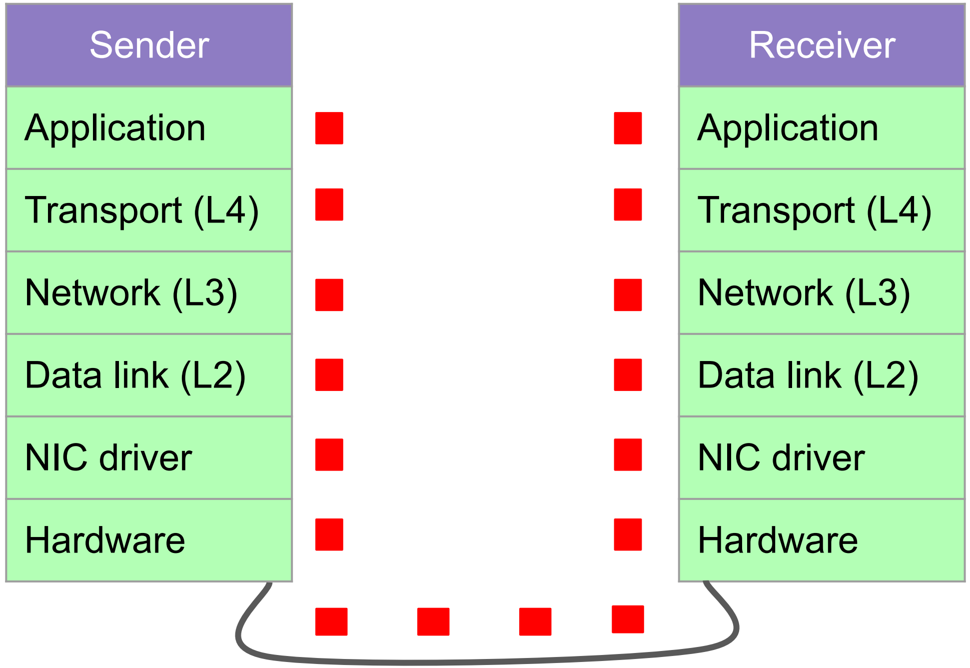
Najiya Naj
Research Scholar, IIITD
najiyan@iiitd.ac.in

Maryam Tahira
Research Scholar,IIITD
maryamt@iiitd.ac.in

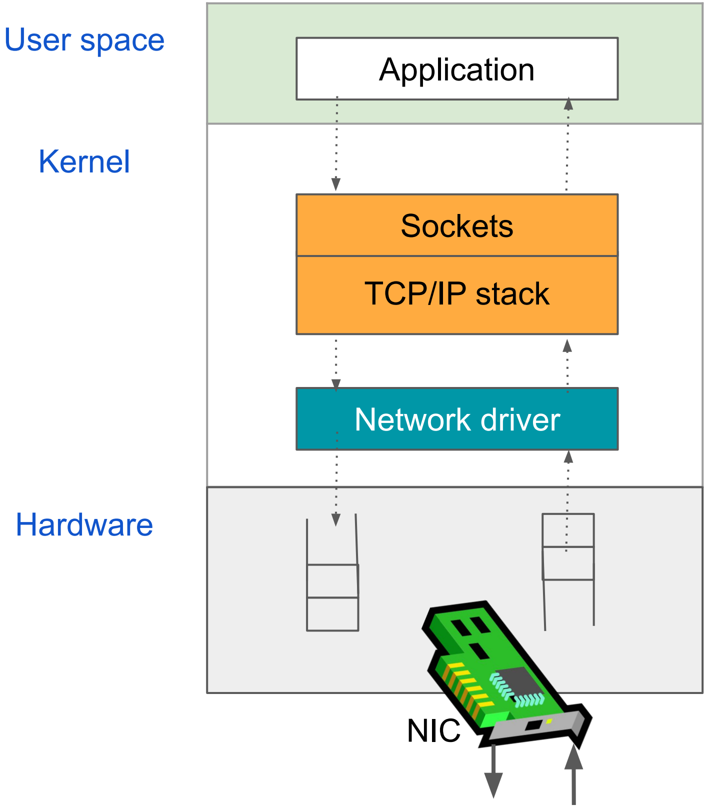
Outline

- Linux networking stack
  - Journey of a packet
- Evolution of NICs
- Evolution of network packet processors
- Hands-on session with Linux system
  - Basic networking commands
  - Configuring network functions

TCP packet flow from source to destination application

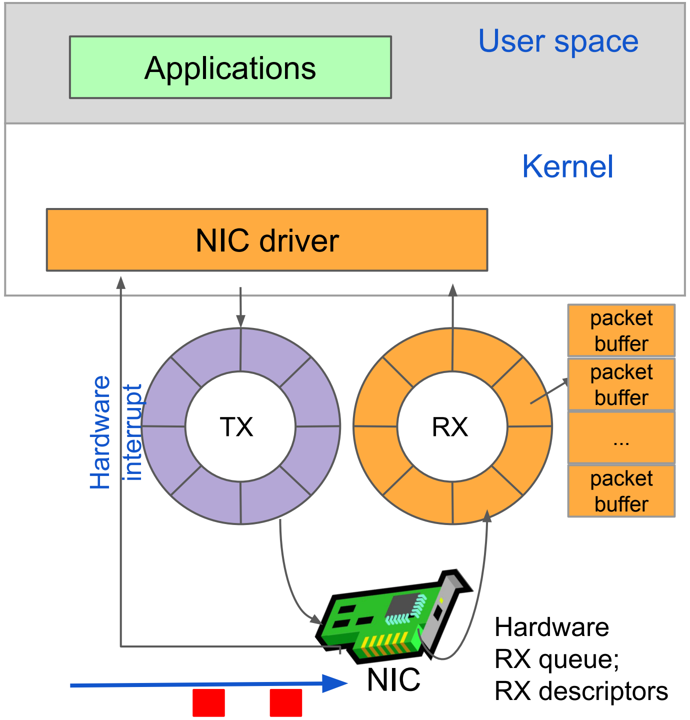


How does the application send and receive packets?



The journey of a packet through the Linux network stack …

RX path: Packet arrives at the destination NIC



NIC receives the packet

- Match destination MAC address
- Verify Ethernet checksum (FCS)

Packets accepted at the NIC

- DMA the packet to RX ring buffer
- NIC triggers an interrupt

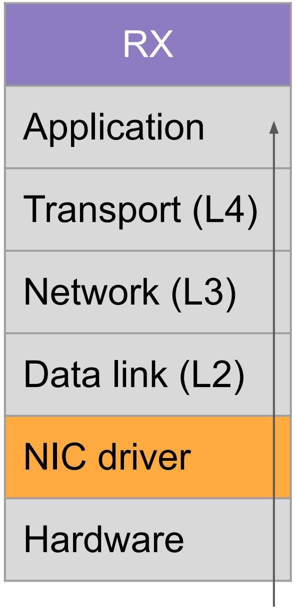
TX/RX rings

- Circular queue
- Shared between NIC and NIC driver
- Content: Length + packet buffer pointer

Interrupt processing in the linux kernel

- Top-half
  - Minimal processing
- Bottom-half
  - Rest of interrupt processing

Top-half interrupt processing



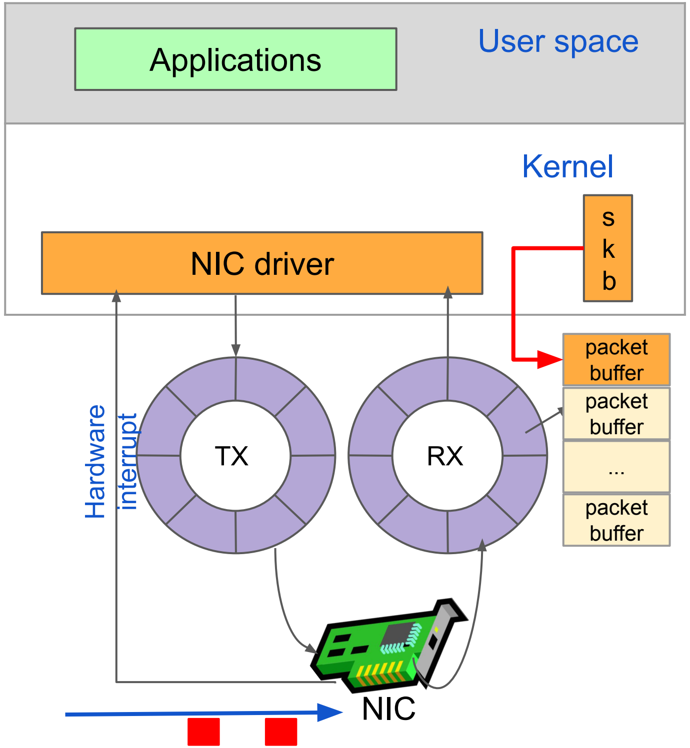
CPU interrupts the process in execution

Switch from user space to kernel space

Top-half interrupt processing

- Lookup IDT (Interrupt Descriptor Table)
- Call corresponding ISR (Interrupt Service Routine)
  - Acknowledge the interrupt
  - Schedule bottom-half processing
- Switch back to user space

Bottom-half processing



CPU initiates the bottom-half when it is free (soft-irq)

Switch from user space to kernel space

Driver dynamically allocates an **sk_buff** (a.k.a., skb)

Oops!!

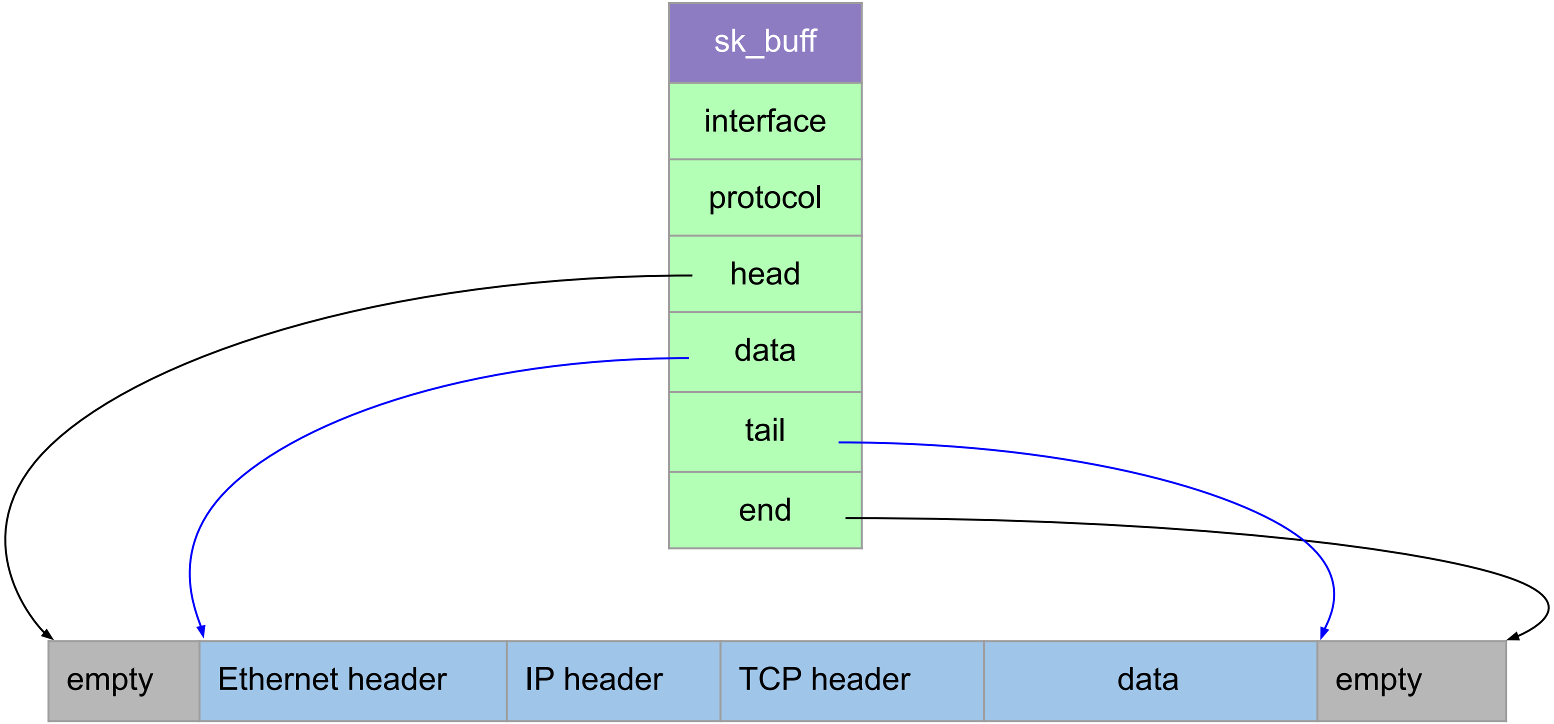
sk_buff (sk_buff tutorial link)

In-memory data structure that contains packet metadata

- Pointers to packet headers and payload
- More packet related information ...

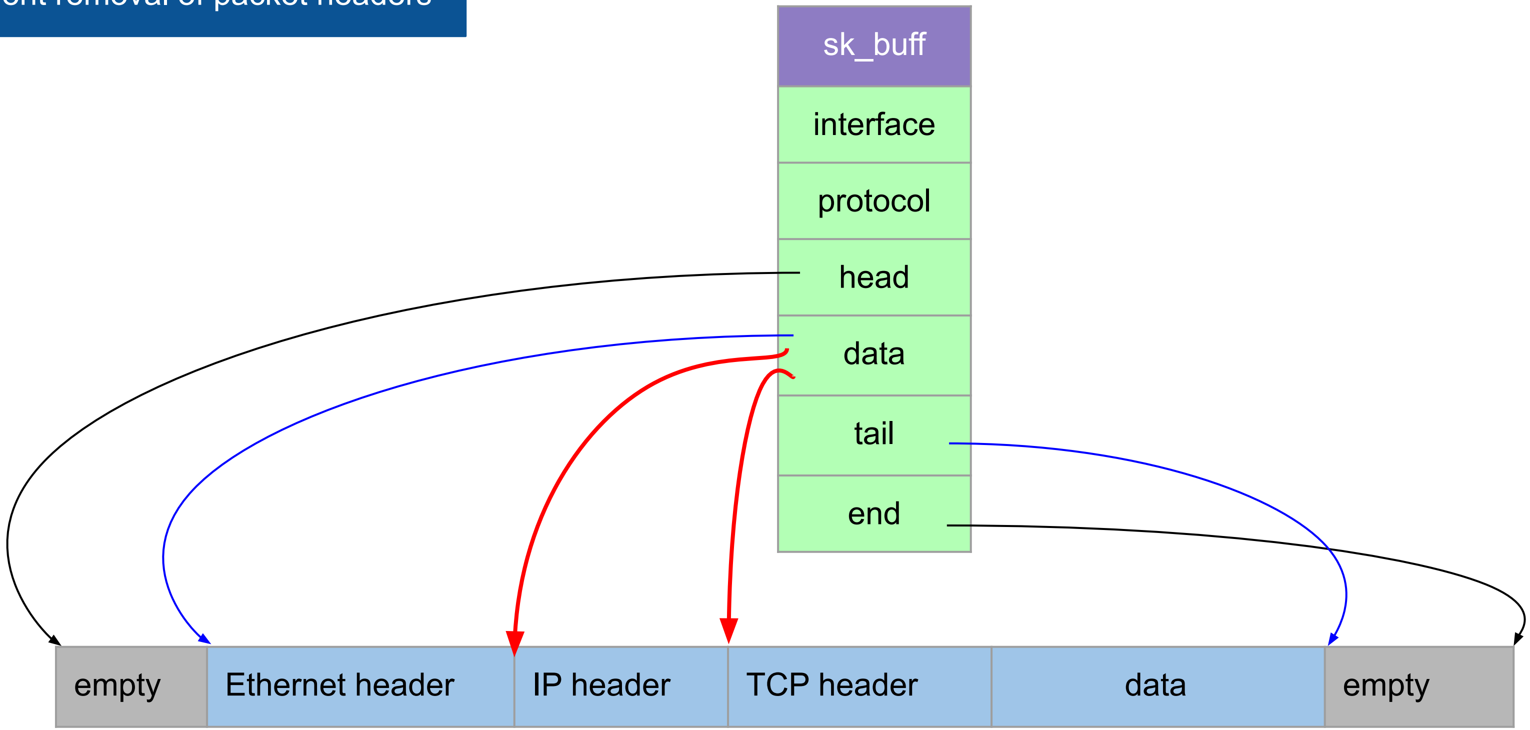
What is “sk_buff” ?

sk_buff: in-memory data structure that contains packet metadata



Advantages of sk_buff

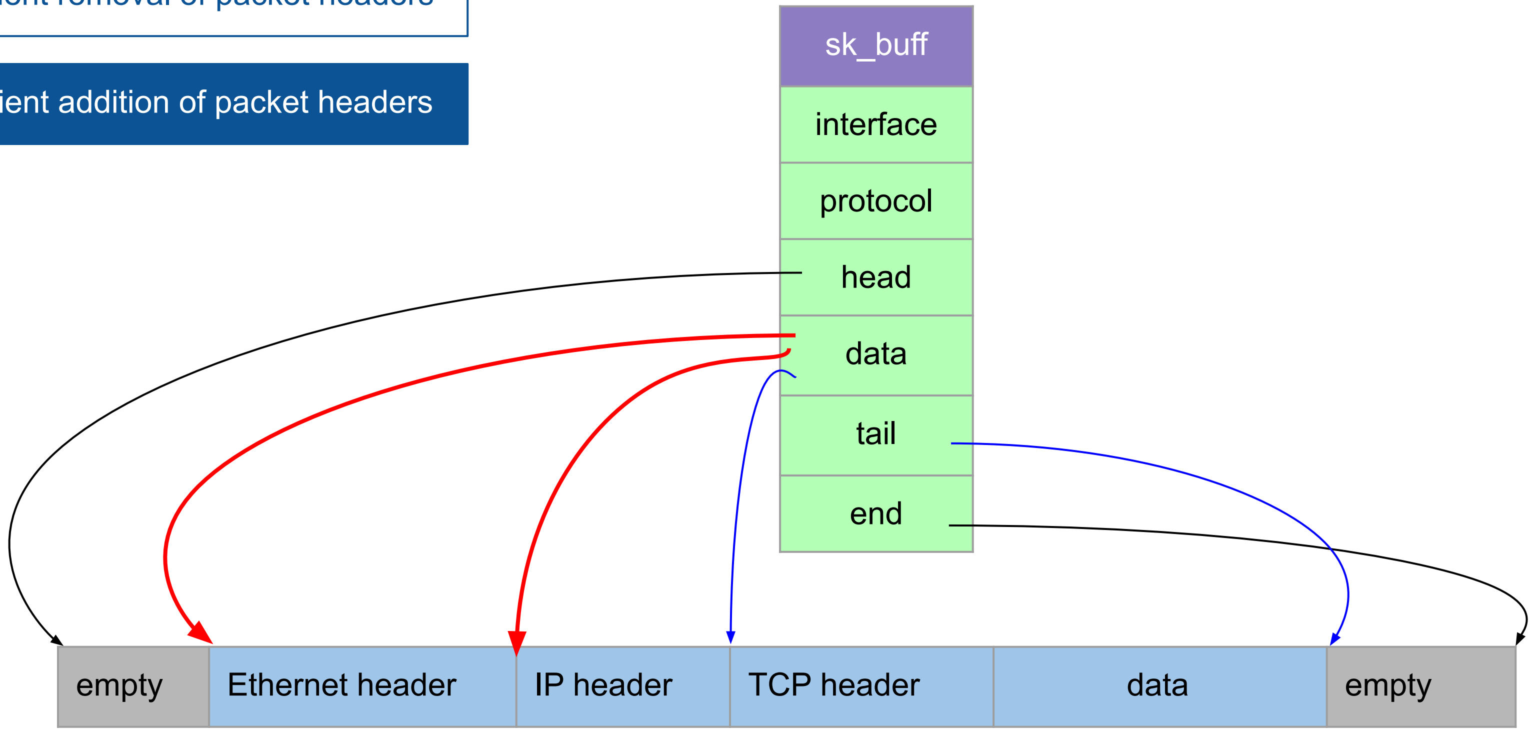
Efficient removal of packet headers



Advantages of sk_buff

Efficient removal of packet headers

Efficient addition of packet headers

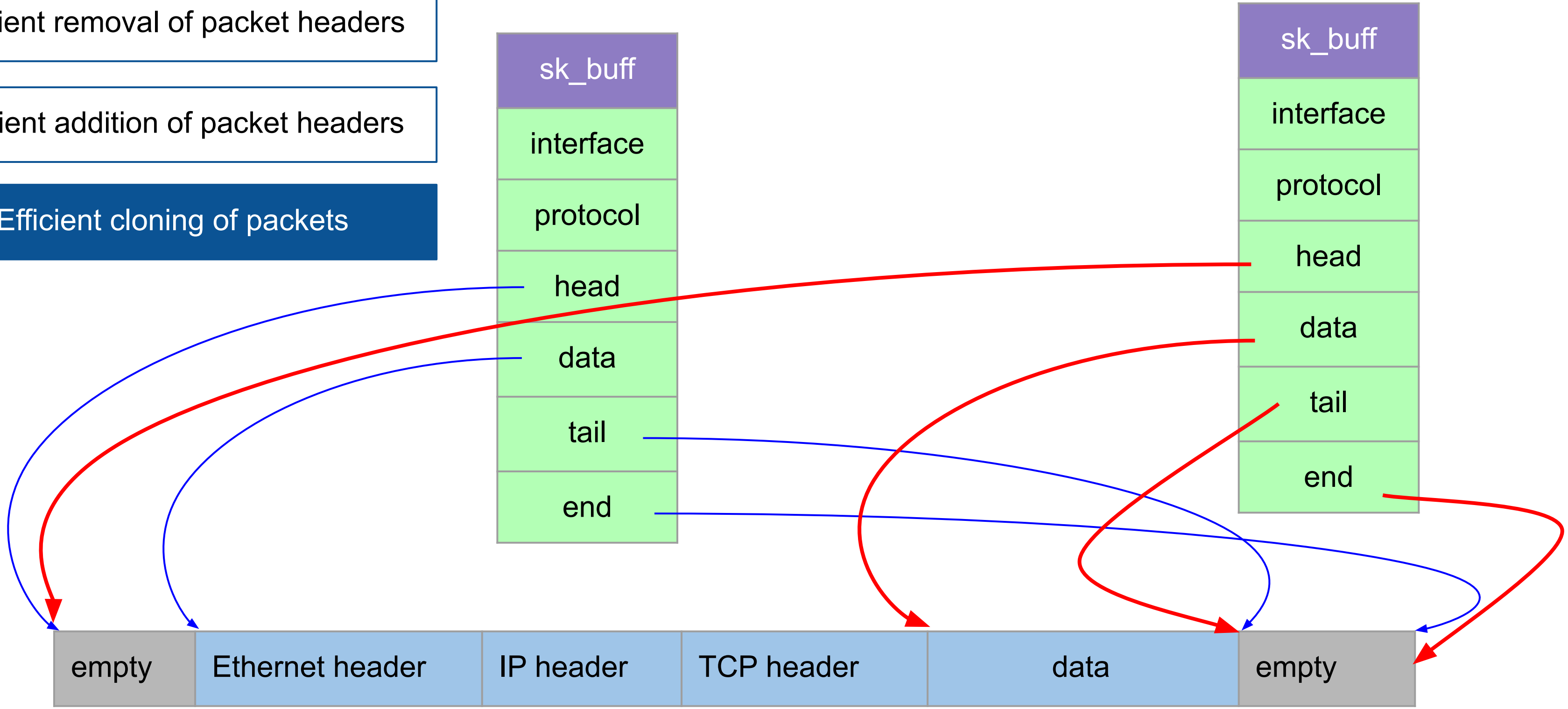


Advantages of sk_buff

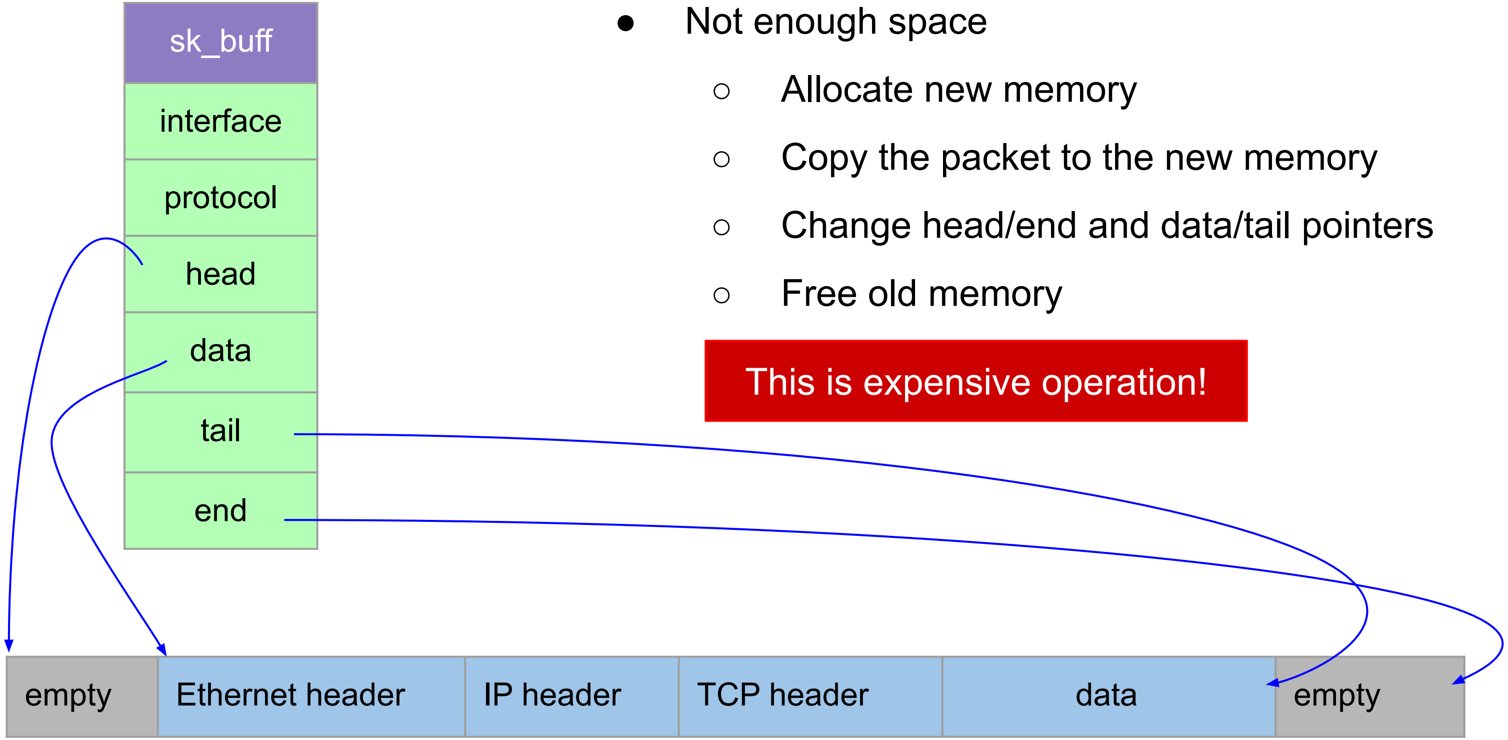
Efficient removal of packet headers

Efficient addition of packet headers

Efficient cloning of packets



Disadvantage of sk_buff

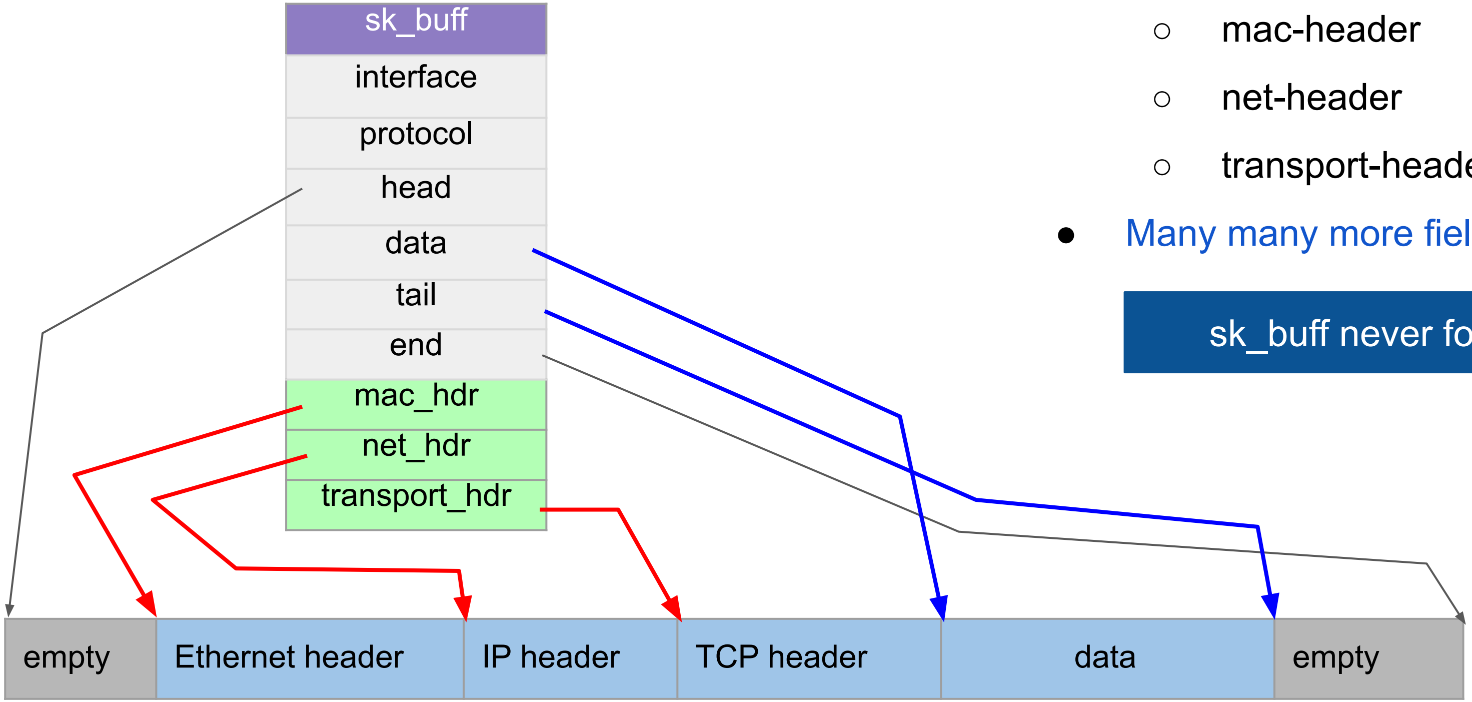


Add packet header at the beginning --- encapsulation

- Not enough space
  - Allocate new memory
  - Copy the packet to the new memory
  - Change head/end and data/tail pointers
  - Free old memory

This is expensive operation!

More about skb structure



- More pointers
  - mac-header
  - net-header
  - transport-header
- Many many more fields ...

sk_buff never forgets!

After alloc_skb

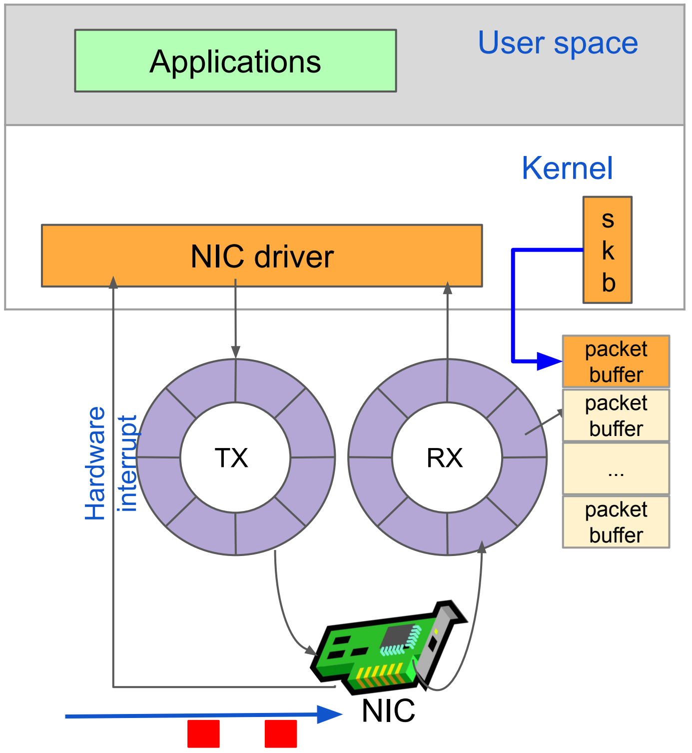
After reserve_skb

skb containing data

skb_put called on buffer

Skb_push has occurred on previous buffer

Bottom-half processing



For all packets in buffer

NIC driver processing

1. Driver dynamically allocates an **sk_buff**
2. Update sk_buff with packet metadata
3. Remove the Ethernet header
4. Pass sk_buff to the network stack

Call L3 protocol handler

L3/L4 processing

RX

Application

Transport (L4)

Network (L3)

Data link (L2)

NIC driver

Hardware

Common processing

1. Match destination IP/socket
2. Verify checksum
3. Remove header

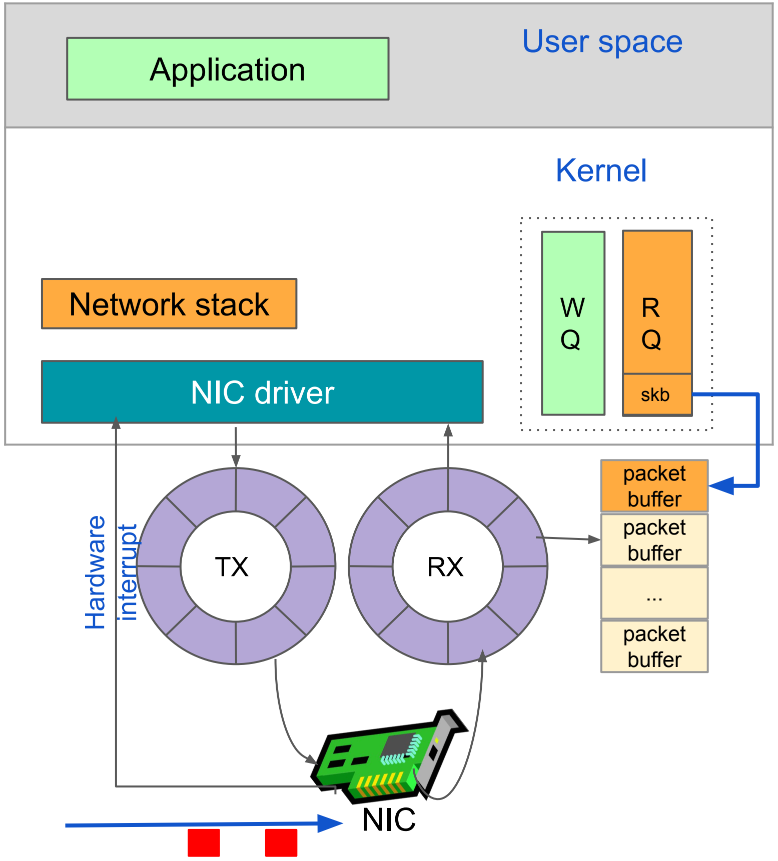
+

L3-specific processing

1. Route lookup
2. Combine fragmented packets
3. Call L4 protocol handler

L4-specific processing

L3/L4 processing



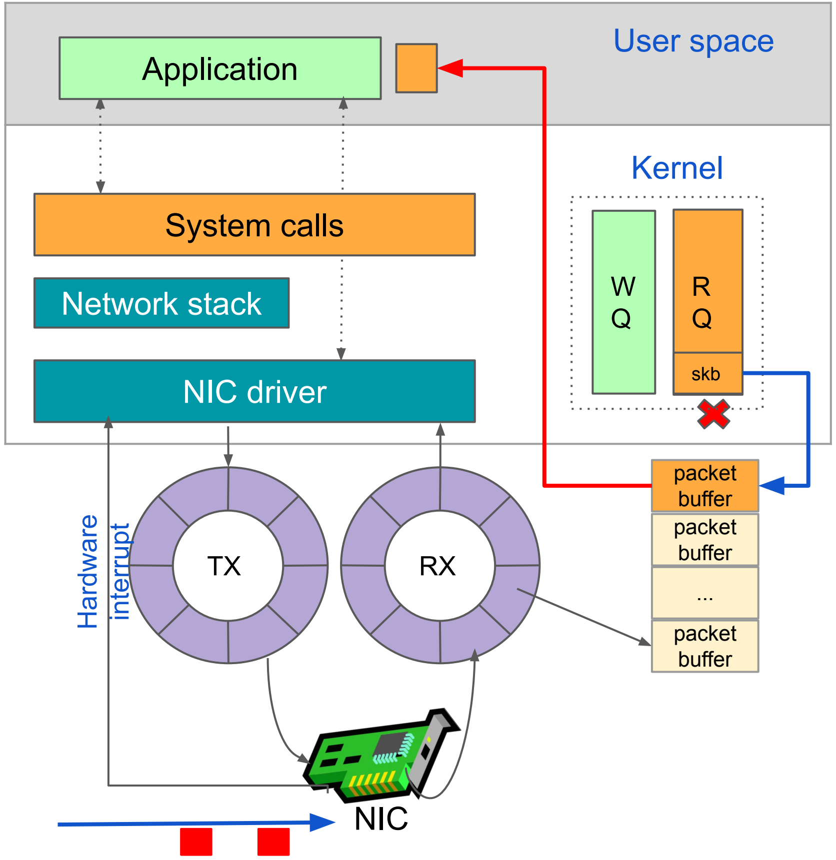
L3-specific processing

1. Route lookup
2. Combine fragmented packets
3. Call L4 protocol handler

L4-specific processing

1. Handle TCP state machine
2. Enqueue to socket read queue
3. Signal the socket

Application processing

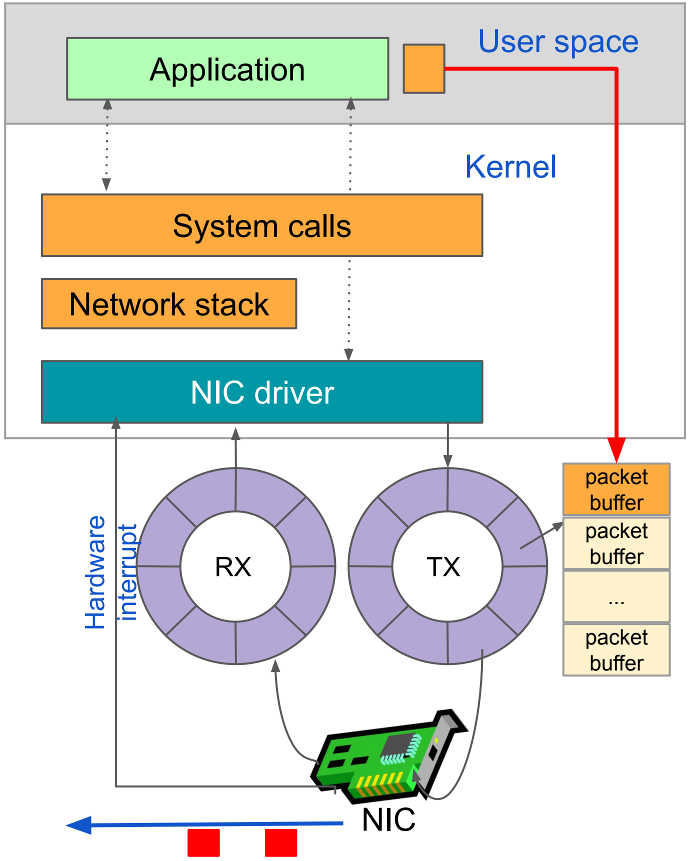


On socket read: **user space to kernel space**

- Dequeue packet from socket receive queue (kernel space)
- Copy packet to application buffer (user space)
- Release sk_buff
- Return back to the application

kernel space to user space

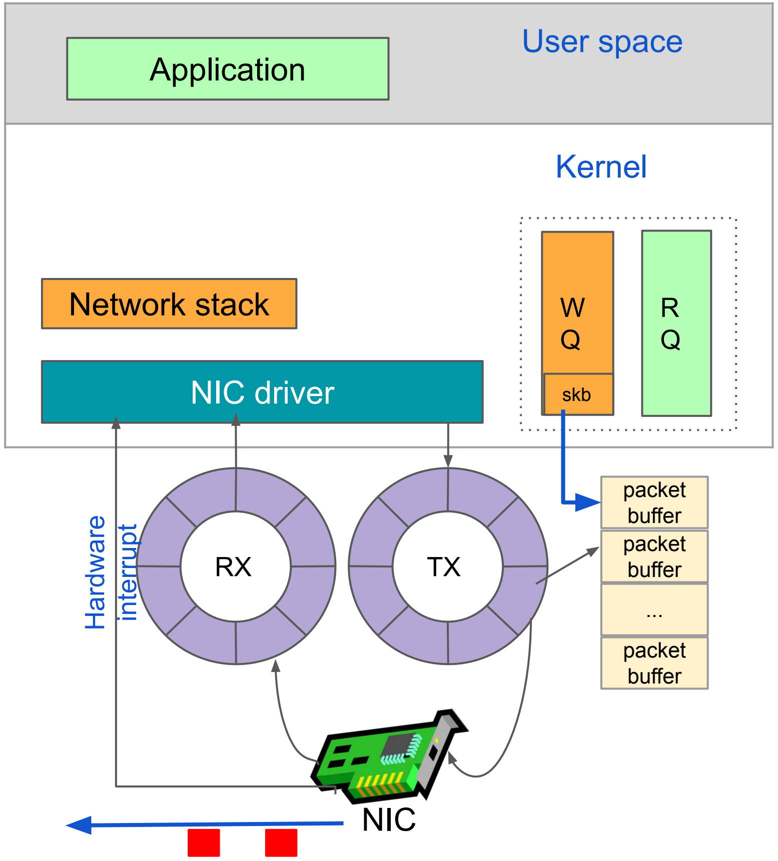
Transmit path of an application packet



On socket write: **user space to kernel space**

- Writes the packet to the kernel buffer
- Calls socket's send function (e.g., sendmsg)

L4/L3 processing



L4-specific processing

1. Allocate sk_buff
2. Enqueue sk_buff to socket write queue
3. Call L3 protocol handler

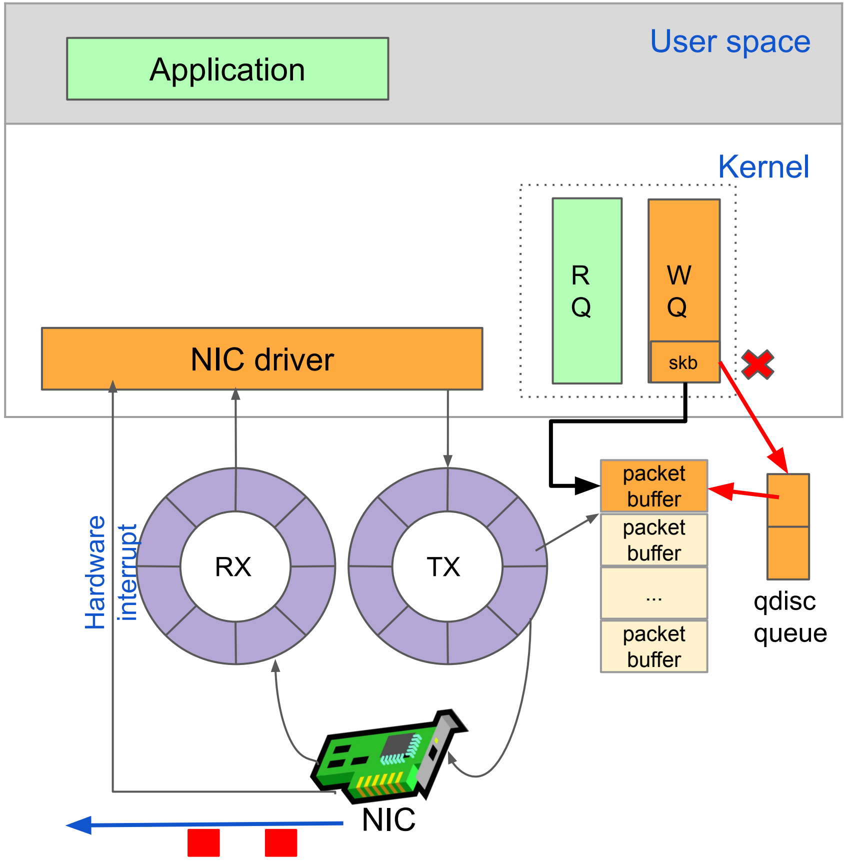
Common processing

1. Build header
2. Add header to packet buffer
3. Update sk_buff

L3-specific processing

1. Fragment, if needed
2. Call L2 protocol handler

L2 processing



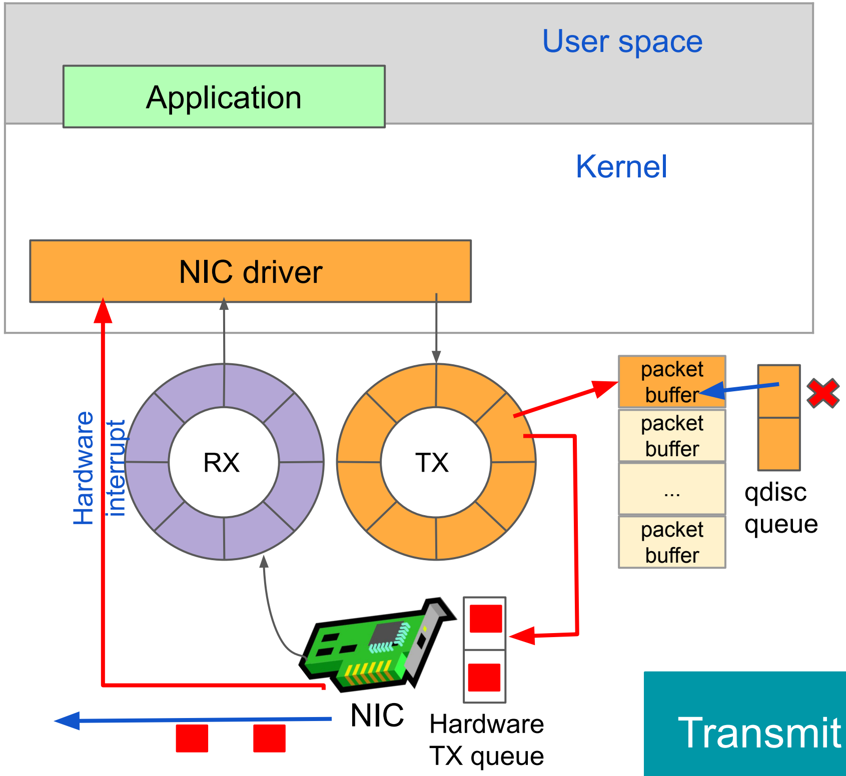
Enqueue packet to queue discipline (qdisc)

- Hold packets in a queue
- Apply scheduling policies (e.g. FIFO, priority)

qdisc

- Dequeue sk_buff (if NIC has free buffers)
- Post process sk_buff
  - Calculate IP/TCP checksum
  - … (tasks that h/w cannot do)
- Call NIC driver's send function

NIC processing



NIC driver

- If hardware transmit queue full
  - Stop qdisc queue
- Otherwise:
  - Map packet data for DMA
  - **T**ells NIC to send the packet

NIC

- Calculates ethernet frame checksum (FCS)
- Sends packet to the wire
- **Sends an interrupt "Packet is sent" (kernel space to user space)**
- Driver frees the sk_buff; starts the qdisc queue

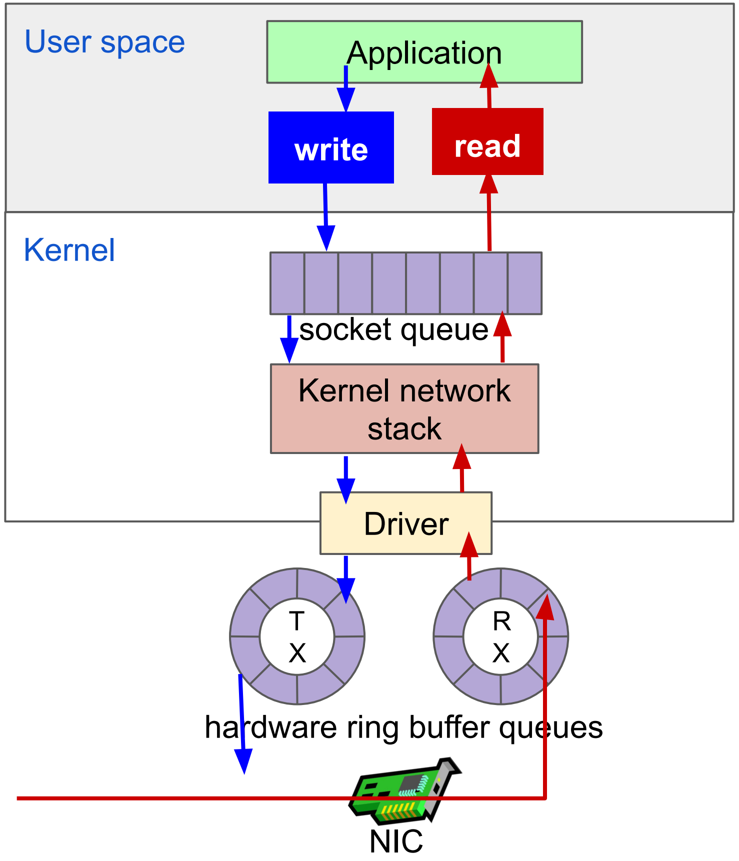
Transmit and receive packet processing pipeline DONE!!

Packet processing overheads in the kernel

- Too many context switches!!
  - Pollutes CPU cache
- Per-packet interrupt overhead
- Dynamic allocation of sk_buff
- Packet copy between kernel and user space
- Shared data structures

Cannot achieve line-rate for recent high speed NICs!! (40Gbps/100Gbps)

Packet processing overheads with the TCP/IP network stack



Sources of overhead

Mode switching

Context switching
Lock/unlock
Packet copy

sk_buff's dynamic alloc/dealloc

Per packet interrupts

100 Gbps NIC[1]
- **RX: 20 CPUs**
- **TX: 10 CPUs**

[1] Understanding Host Network Stack Overheads

* Slide abstracts details for simplicity

NIC Offloads? Why?

- Free up server CPU cycles for application
- Specialized processing can be efficient
- Scaling performance
  - Low latency
  - High throughput
- Power savings

Evolution of Network Interface Cards

Ref: IETF 105 Technology Deep Dive: How Network Interface Cards (NICs) Work Today

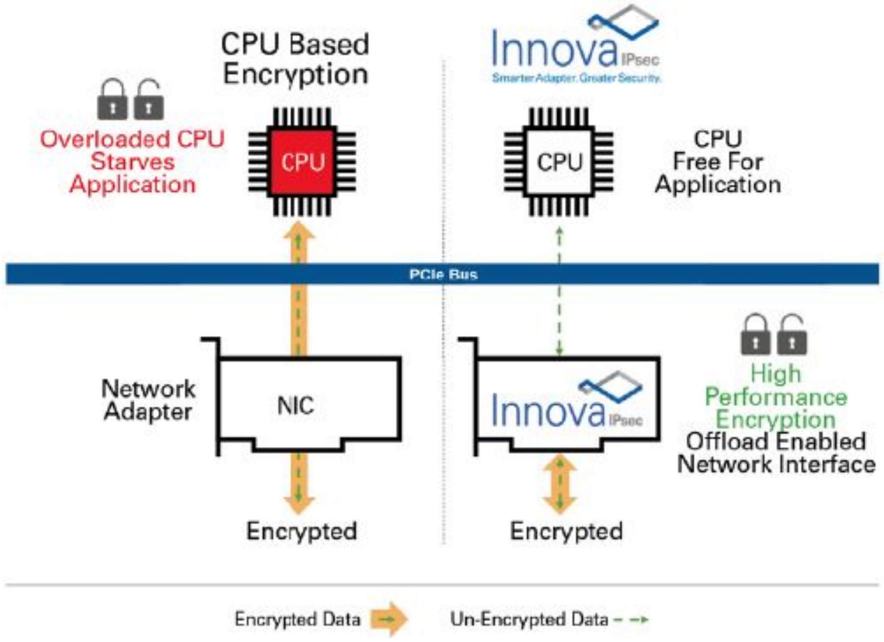
Improve NIC performance: What should be offloaded?

Basic offloads

- High frequency, specialized, compute-intensive tasks **Checksum offload**
- Reduce per packet overheads → Increase packet sizes?? **Segmentation offload**
- Parallel packet processing **Multiqueue offload**

Advanced offloads: Data plane in Hardware

Fixed function (minimally configurable) pipeline



- 6x throughput
- 10x CPU savings

Ref: https://network.nvidia.com/pdf/whitepapers/WP_Innova_IPsec.pdf

Advanced offloads: Data plane in Hardware

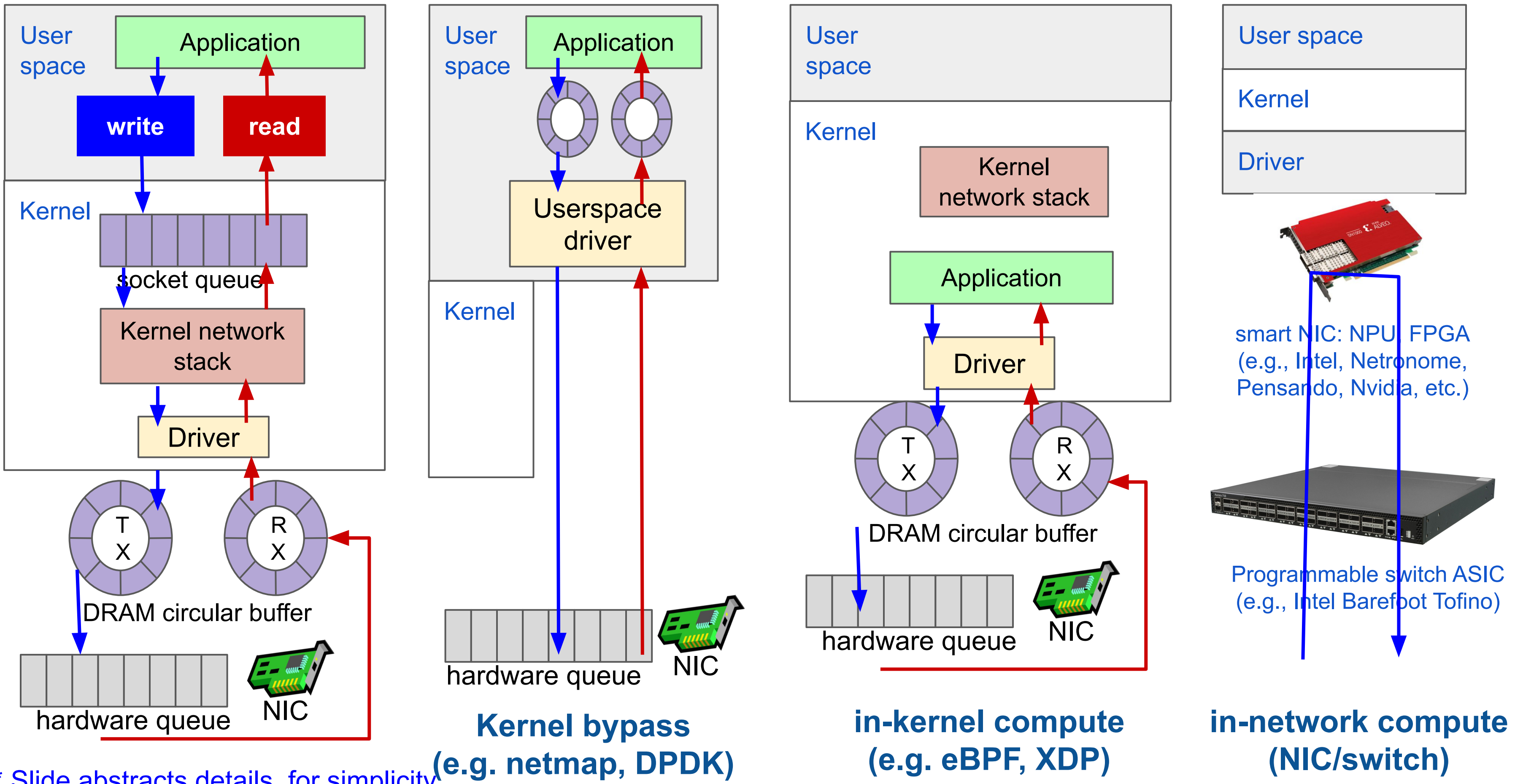
Programmable pipeline

- Network processing unit (NPU)
  - Multithreaded or many-cores
  - Some domain-speciifc instructions
- FPGA
  - Gate-level programmable
- General purpose processor

Netronome smart NIC CX4000

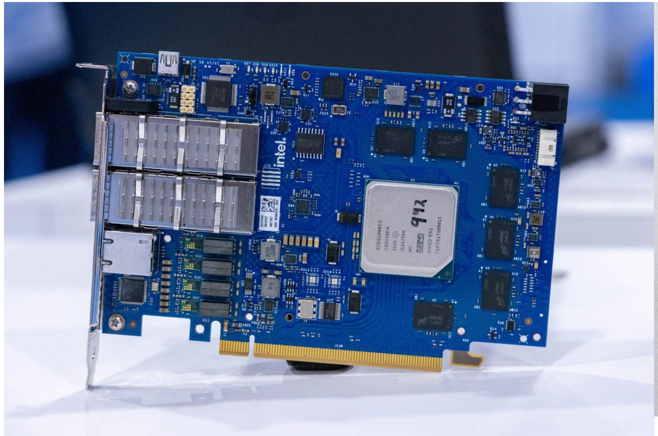
[Xilinx SN1022 FPGA smart NIC]

Evolution of network packet processing



* Slide abstracts details for simplicity

Industry trends



[Intel Mounts Evan IPU (ASIC+FPGA)]

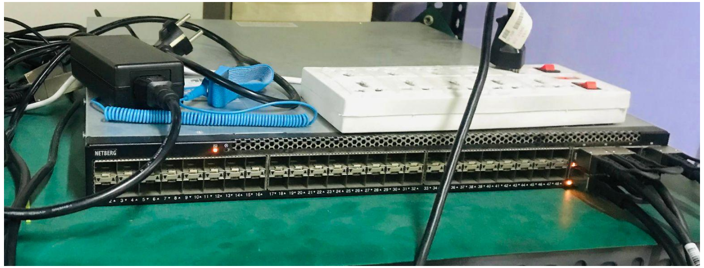
[Pensando DPU DSC-200]

[Xilinx Alveo 280 FPGA smart NIC]

Other vendors: Broadcom, Nvidia, …

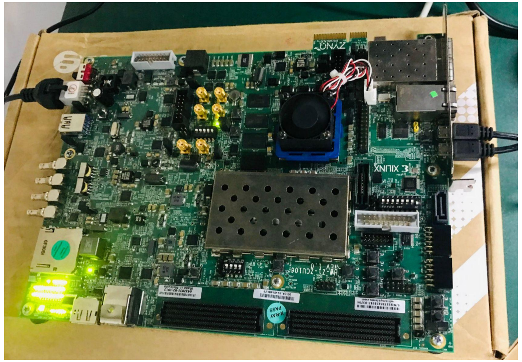
Programmable Netwoking Lab @ IIIT Delhi

https://github.com/pnl-iiitd



Intel Tofino switch

Netronome smart NIC
CX4000



Xilinx MPSoC FPGA board
ZCU106

Xilinx FPGA smart NIC
SN1022

Backup slides

After alloc_skb

After reserve_skb

skb containing data

skb_put called on buffer

Skb_push has occurred on previous buffer