Kernel Bypass

Mainack Mondal
Sandip Chakraborty

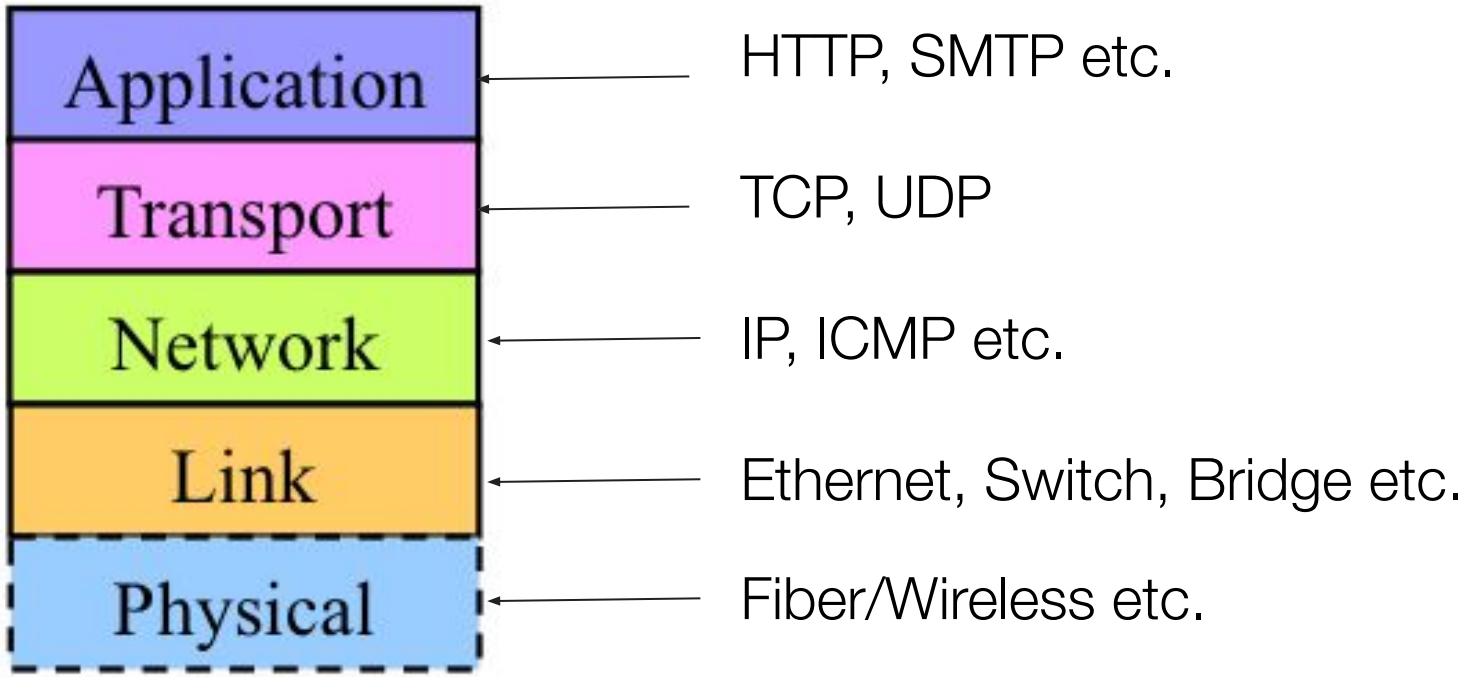
CS 60203
Autumn 2024

Outline

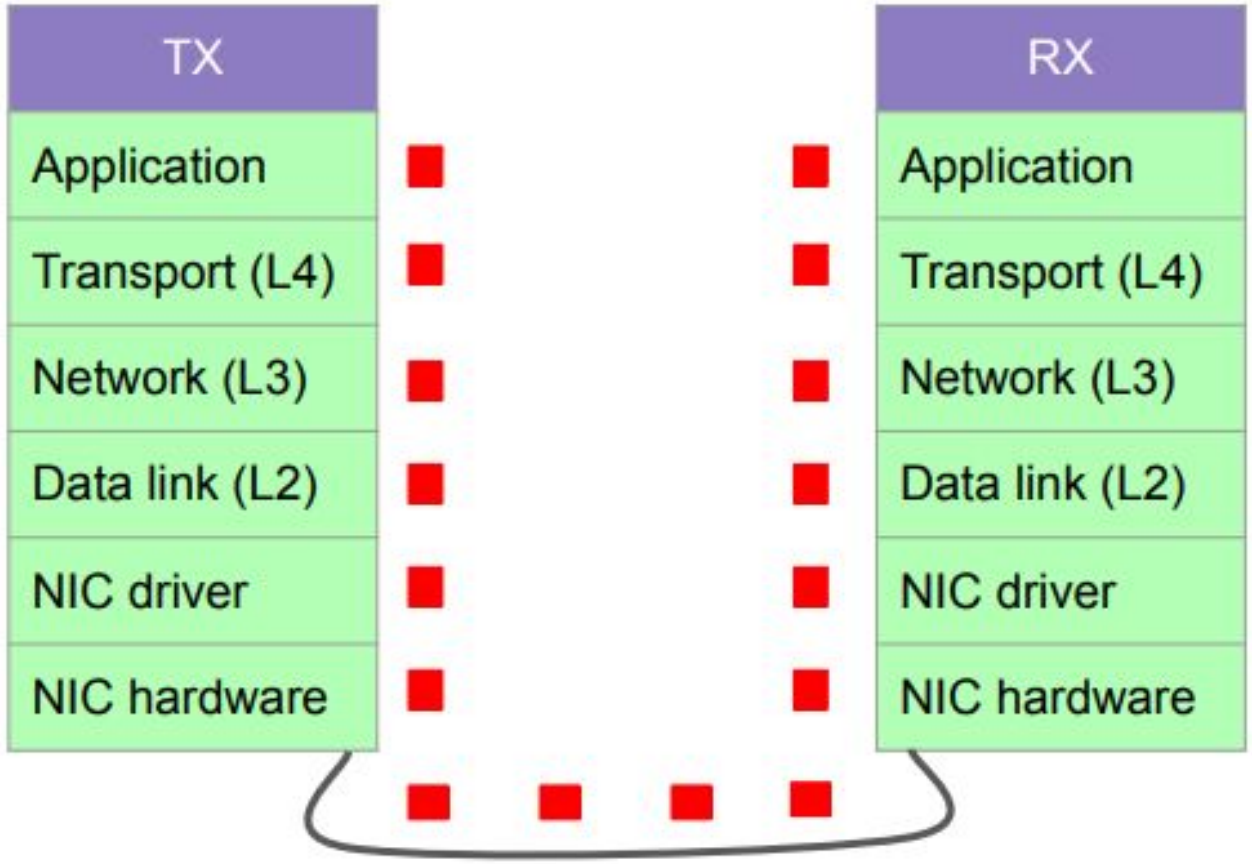
- Linux Network Stack
- Need for Kernel Bypass
- Kernel Bypass Techniques
  - User-space packet processing
    - Netmap and DPDK
  - User-space network stack
    - mTCP

Linux Network Stack (Slides credits: Mythili Vutukuru, IIT Bombay)

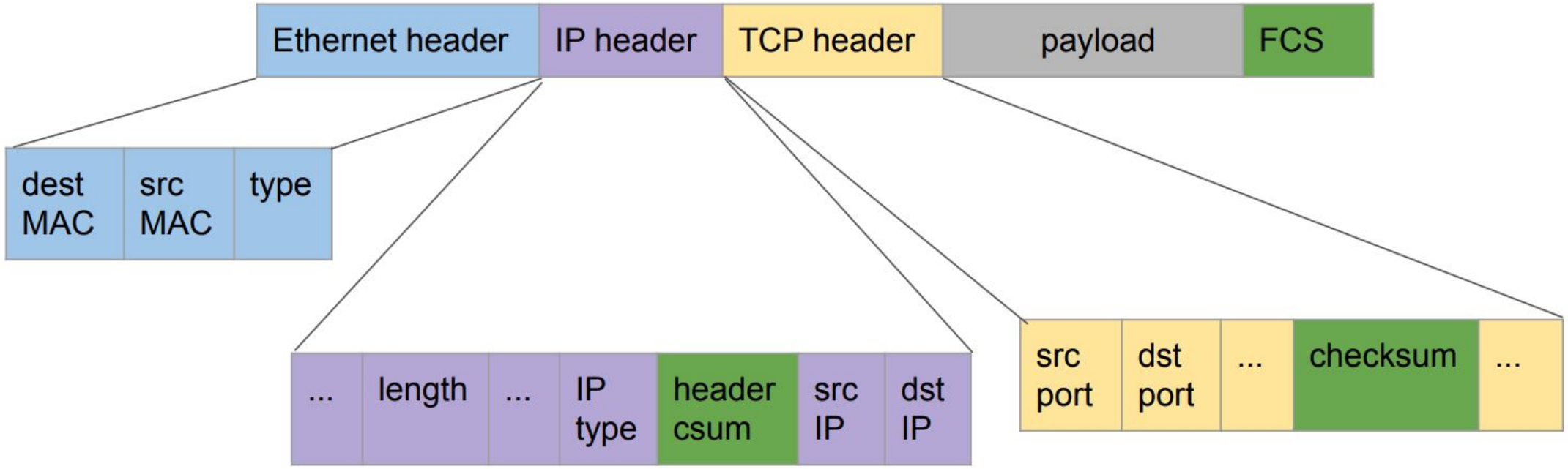
TCP/IP Layers



Typical Packet Flow

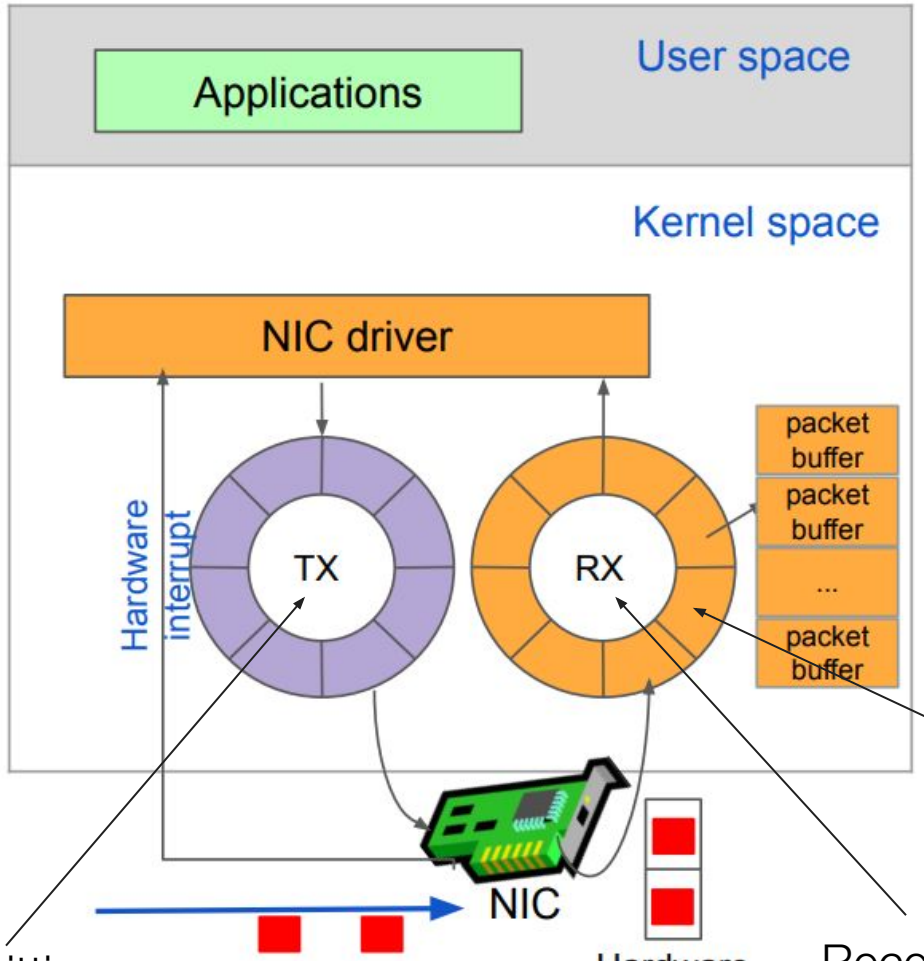


Packet Contents



Let’s see the journey of a packet through the Linux network stack

Packet Arrives at NIC



NIC receives the packet

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

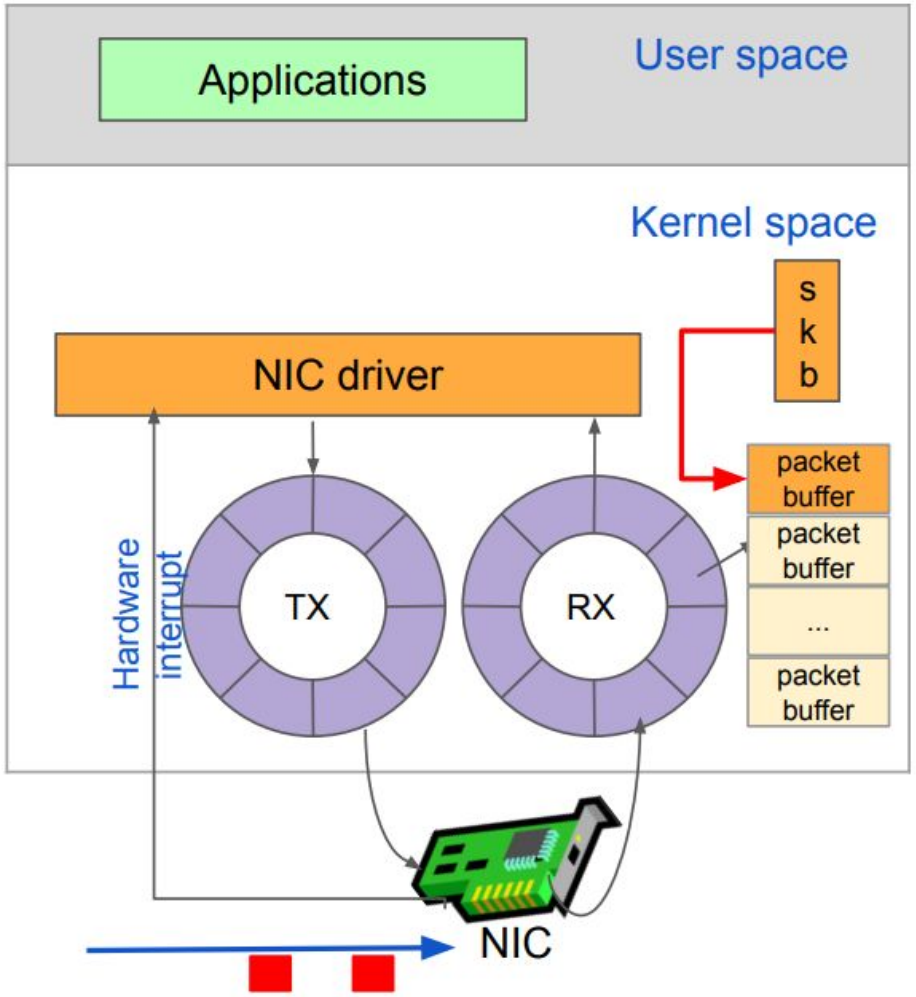
Packets accepted at the NIC

- 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

Processing the Packet in Kernel



Driver dynamically allocates an `sk-buff(skb)`

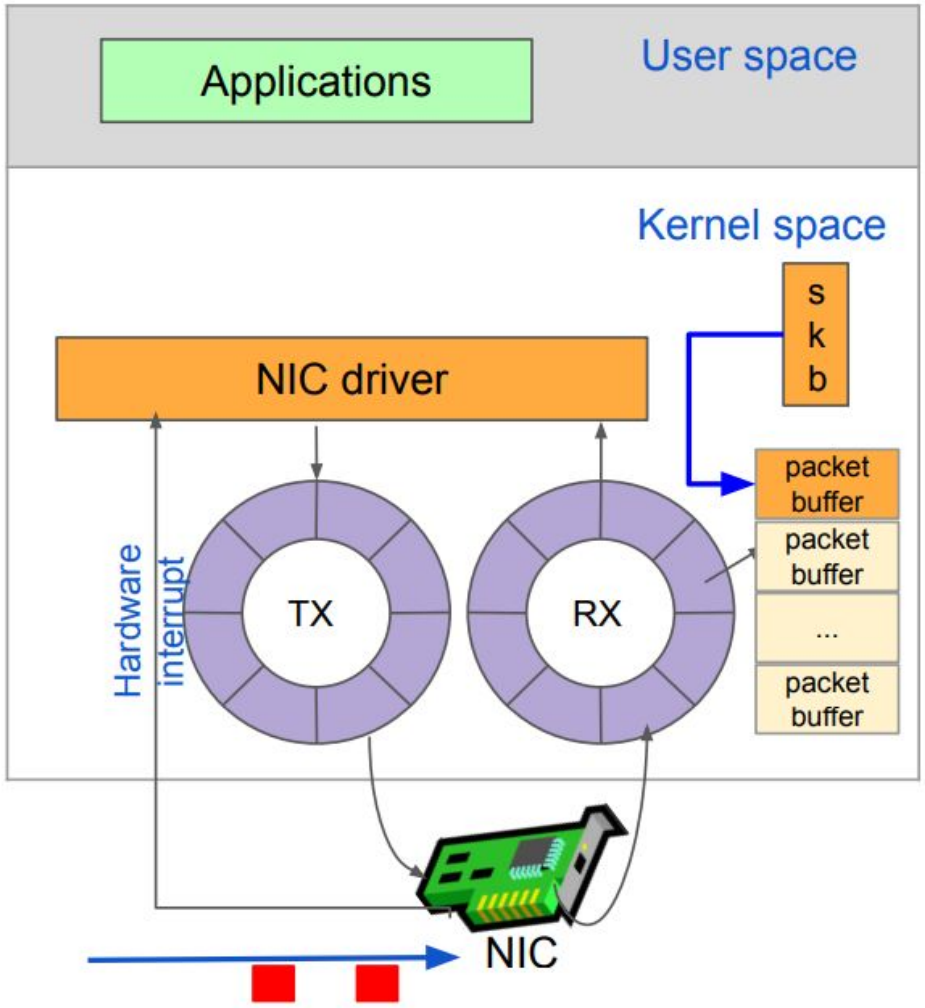
`sk-buff`

In-memory data structure that contains packet metadata

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

To know more about sk_buff: Read Link

Packet Processing (Contd.)



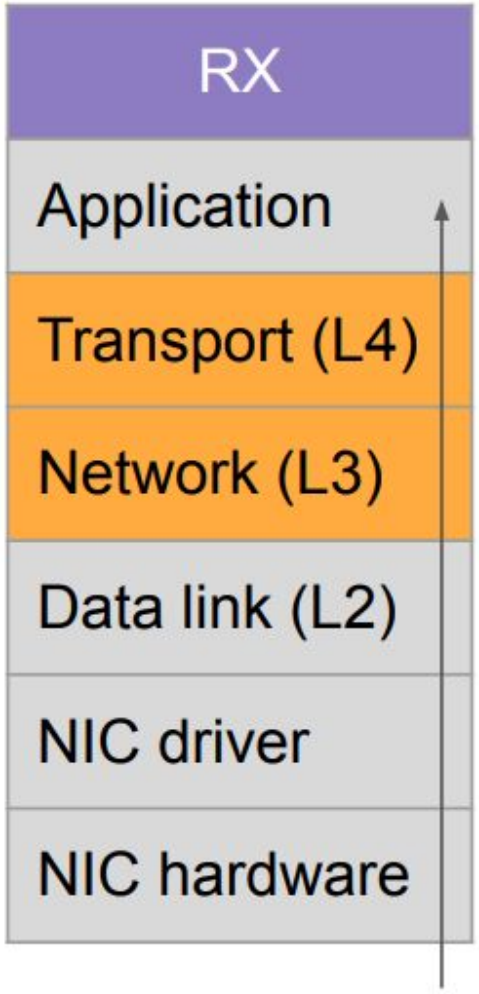
NIC driver processing

- Driver dynamically allocates an `sk-buff`
- Update `sk-buff` with packet metadata
- Remove the Ethernet header
- Pass `sk-buff` to the network stack

(for all packets in buffer)

Call L3 protocol handler

L3/L4 Processing



Common Processing

- Match destination IP/socket
- Verify checksum
- Remove header

+

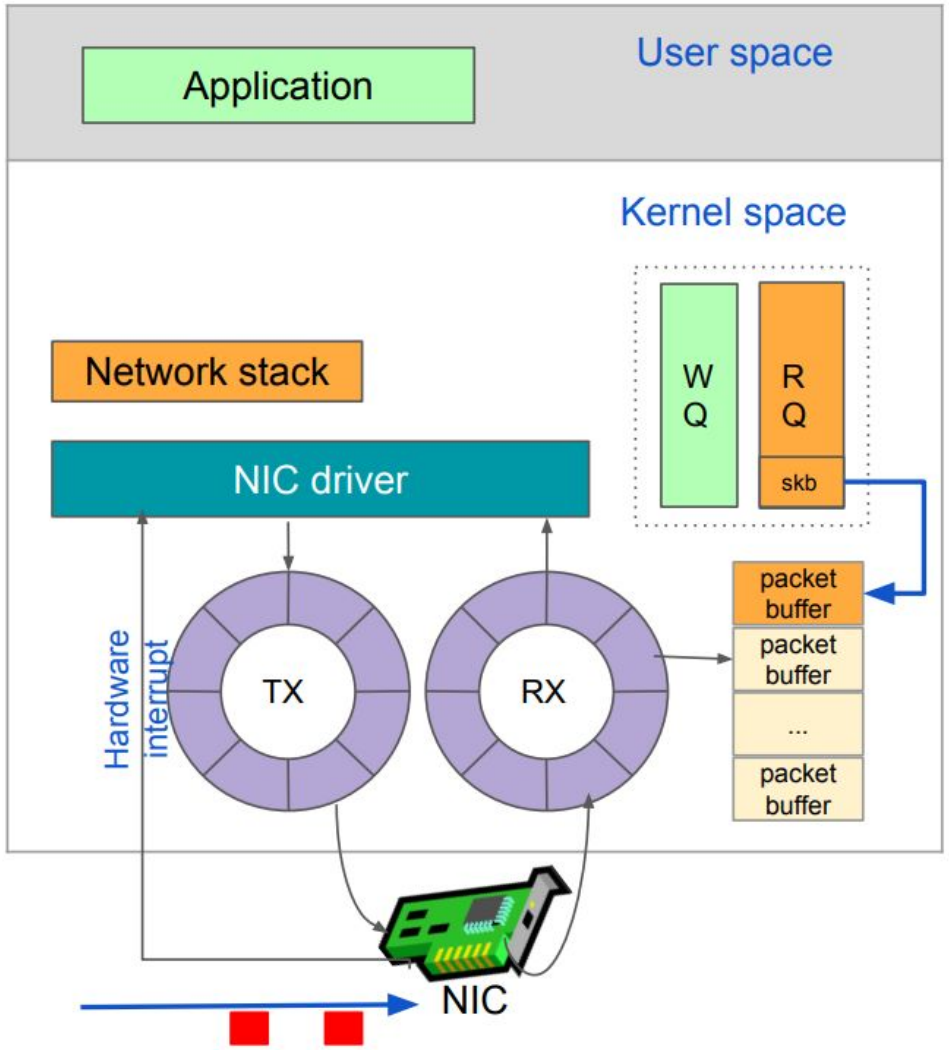
L3-specific processing

- Route lookup
- Combine fragmented packets
- Call L4 protocol handler

L4-specific processing

- Handle TCP state machine
- Enqueue to socket read queue
- Signal the socket

L3/L4 Processing (Contd.)



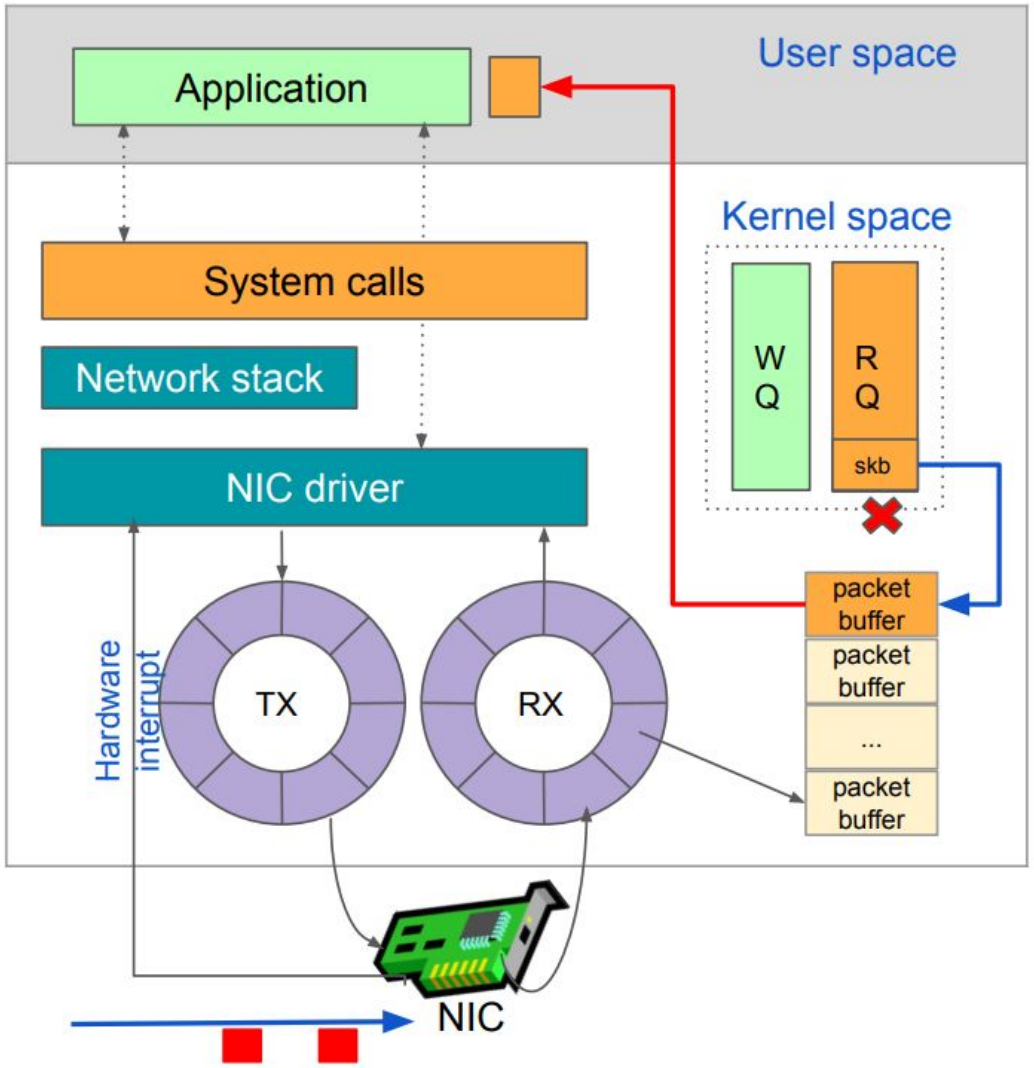
L3-specific processing

- Route lookup
- Combine fragmented packets
- Call L4 protocol handler

L4-specific processing

- Handle TCP state machine
- Enqueue to socket read queue
- Signal the socket

Application Layer Processing



user space to kernel space

On socket read:

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

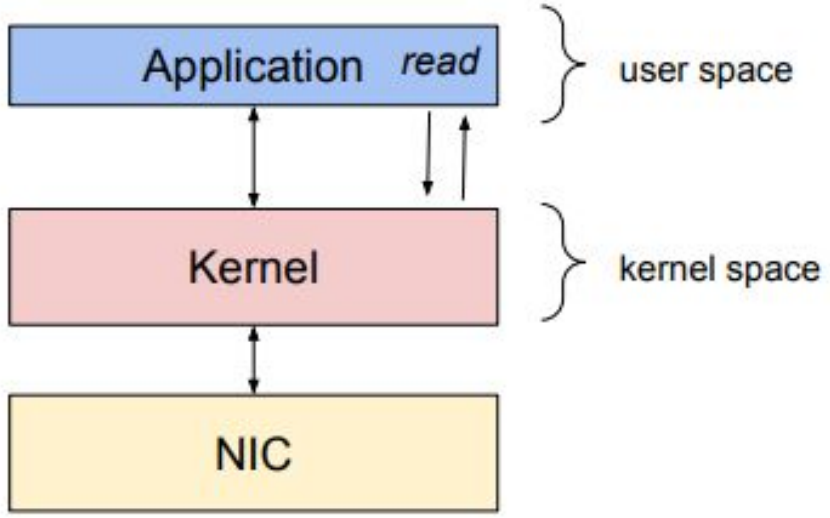
kernel space to user space

And the process goes on …

Need for Kernel Bypass

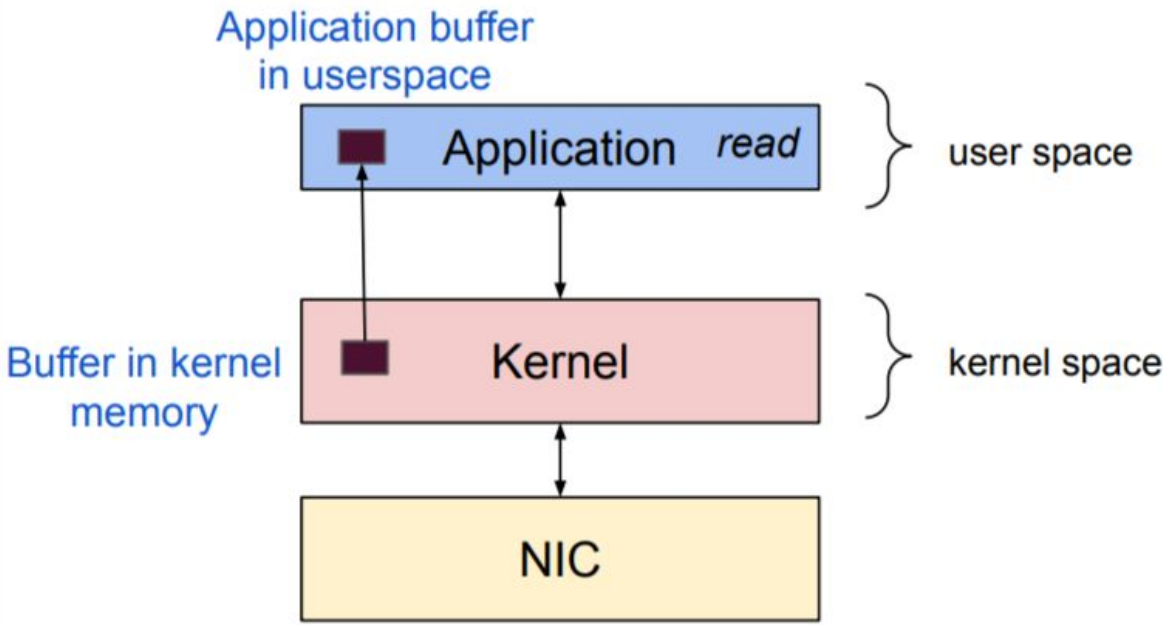
Packet Processing Overheads in Kernel

Context switch between kernel and userspace



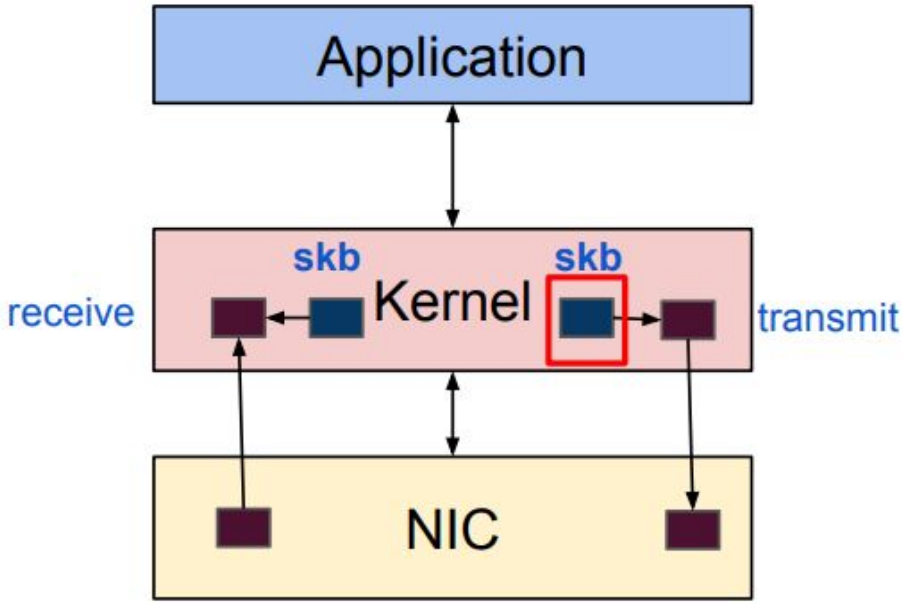
Packet Processing Overheads in Kernel

Packet copy between kernel and userspace



Packet Processing Overheads in Kernel

- Dynamic allocation of sk_buff
- Per packet interrupt
- Shared data structures



Summary

- Too many context switches!! → Pollutes CPU cache
- Per-packet interrupt overhead
- Queuing delays
- Dynamic allocation of sk-buff
- Packet copy between kernel and user space
- Shared data structures
- Too Bad!! in multicore

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

Kernel Bypass to the Rescue

Image credits: Matt Brown

Outline

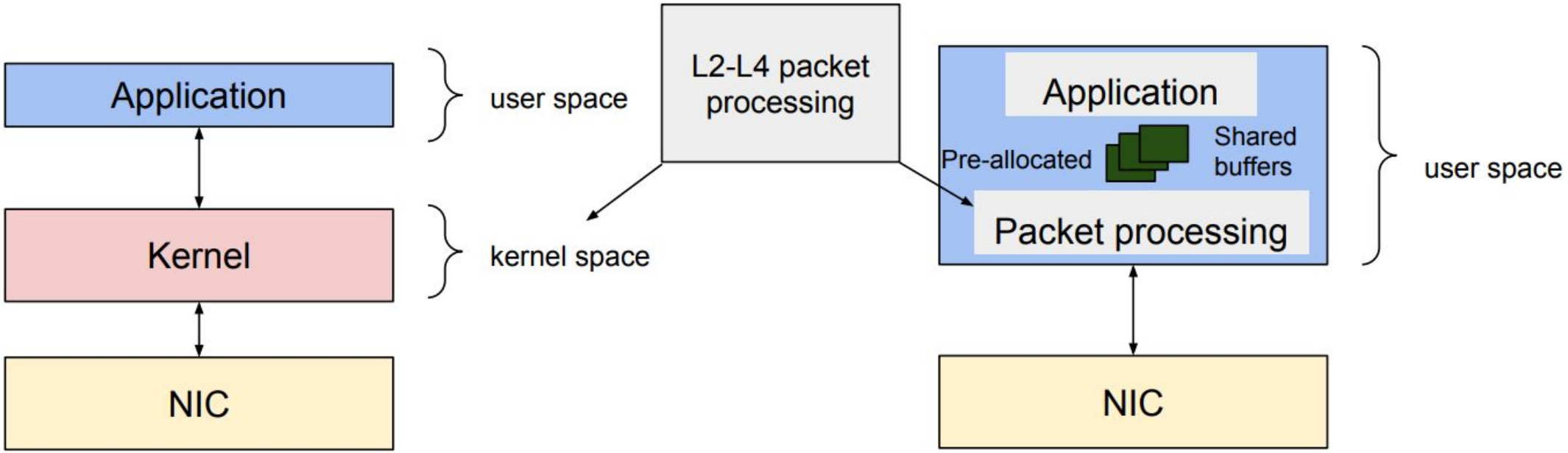
- Linux Network Stack

- Need for Kernel Bypass

- **Kernel Bypass Techniques**
  - User-space packet processing
    - Netmap and DPDK
  - User-space network stack
    - mTCP

Kernel Bypass Techniques

Overcome Overheads in Kernel: Kernel Bypass



- **No** Context switch between kernel and userspace
- **No** Packet copy between kernel and userspace
- **No** Dynamic allocation of sk_buff

Yay !!

But how does your userspace programs know when a packet has arrived?

Interrupt vs Poll Mode: Kernel Bypass Techniques

Interrupt Mode

- NIC notifies it needs servicing
- Interrupt is a hardware mechanism
- Handled using interrupt handler
- Interrupt overhead for high speed traffic

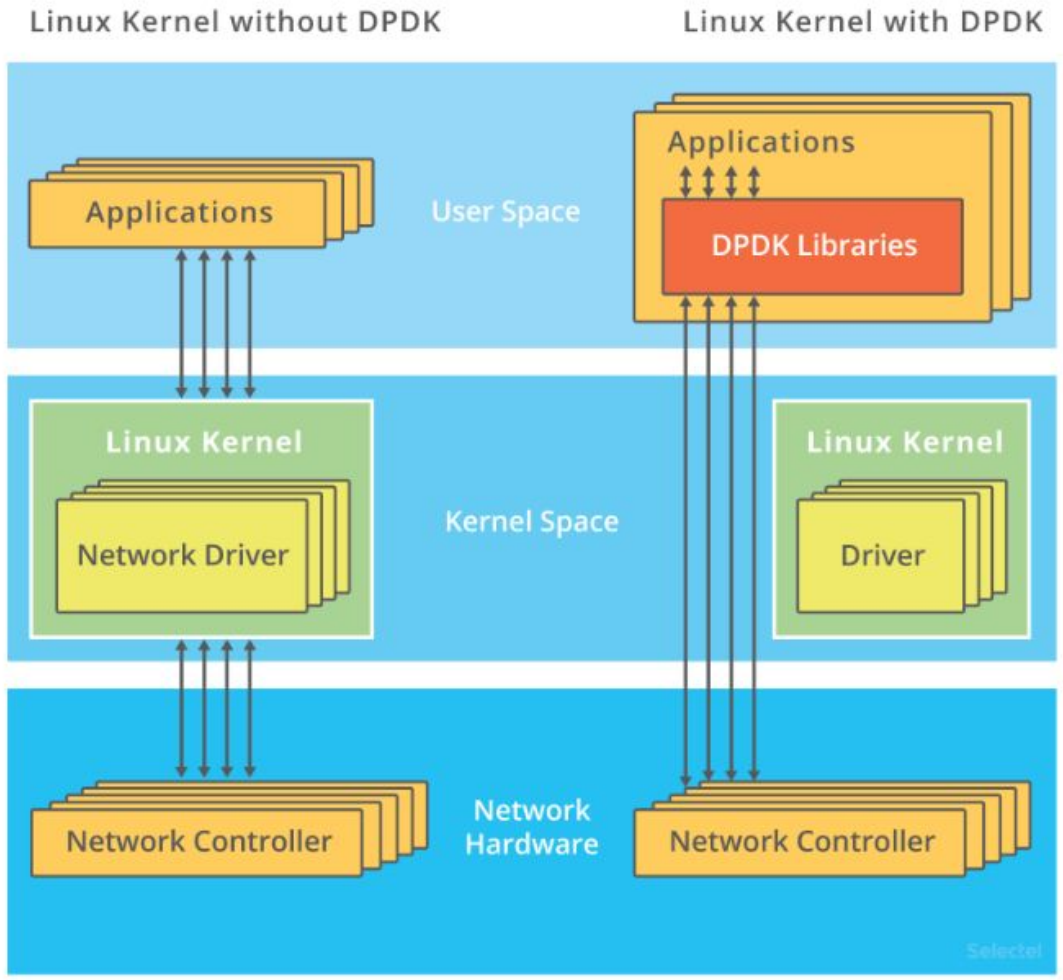
Netmap

Poll Mode

- CPU keeps checking the NIC
- Polling is done with help of control bits(**Command-ready** bit)
- Handled by the CPU
- Consumes CPU cycles but handles high speed traffic

DPDK

DPDK: Dataplane Development Kit



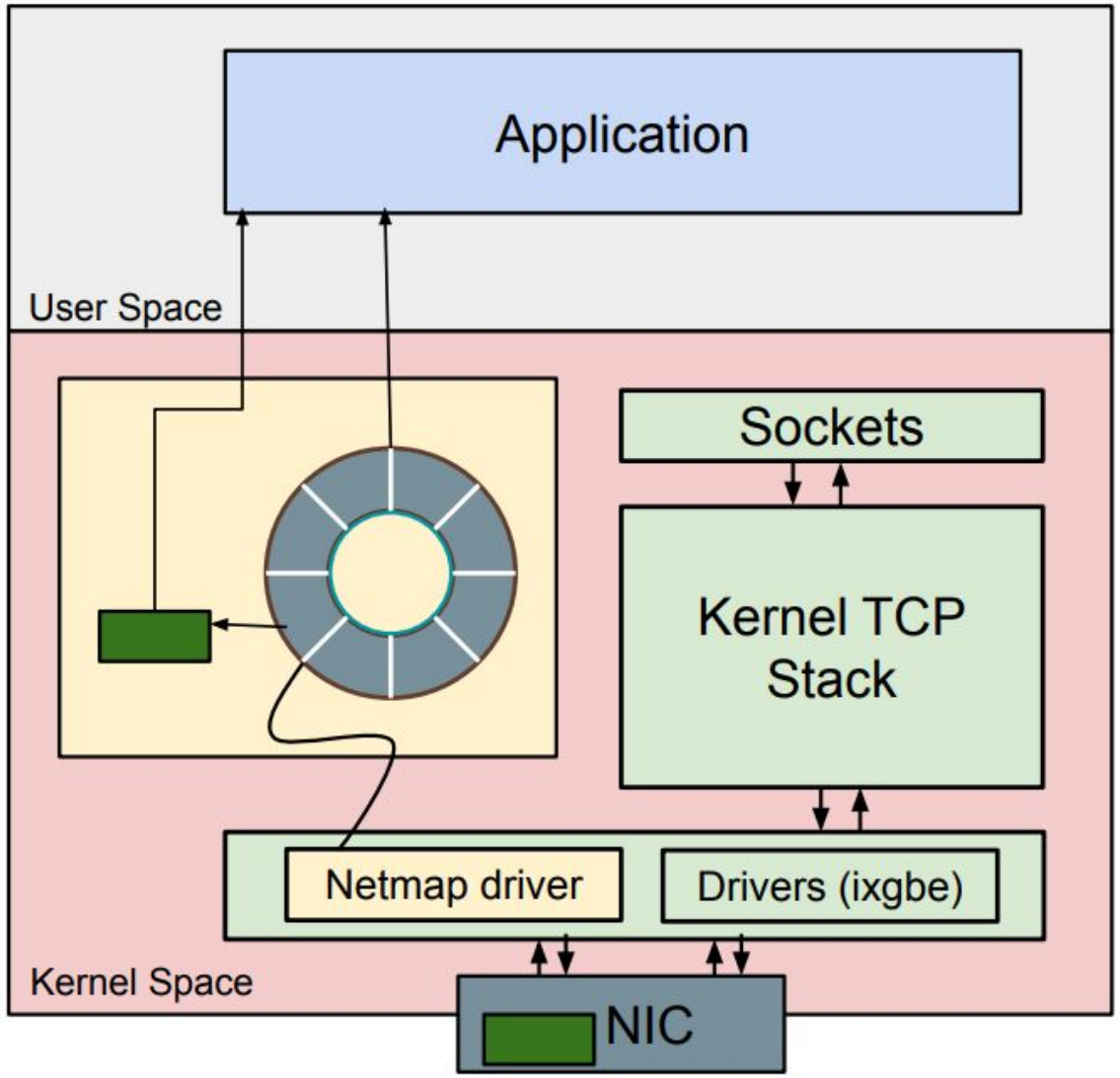
- User-space packet processing
  - Avoid context switching overhead
- Poll Mode Driver (PMD)
  - Avoid interrupt processing overhead
  - Keeps a core busy
- Memory usage optimizations
  - Light-weight mbufs
  - Memory pools that use hugepages, cache alignment, etc
  - Lockless ring buffers

Source: https://blog.selectel.com/introduction-dpdk-architecture-principles/

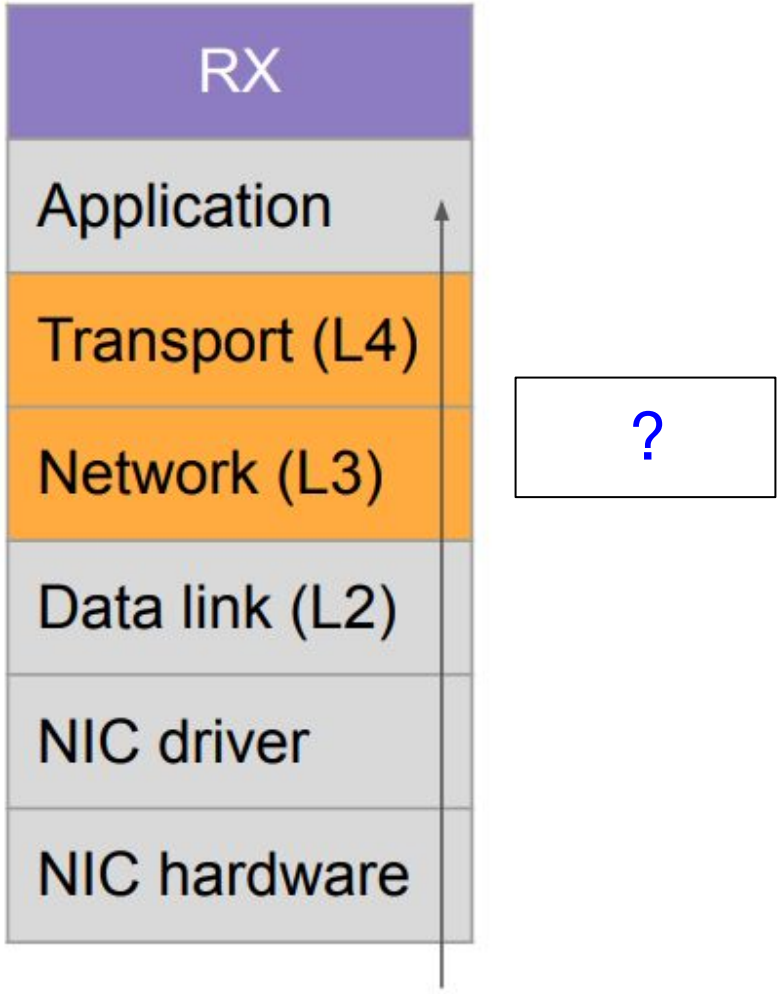
Netmap

- Netmap Rings are memory regions in kernel space shared between application and kernel
- Fast Interface for packet sniffing
- Light-weight packet buffers
- Fewer memory copies
- NIC can work with netmap as well as kernel drivers (transparent mode)

DPDK, netmap manage processing till L2 of network stack

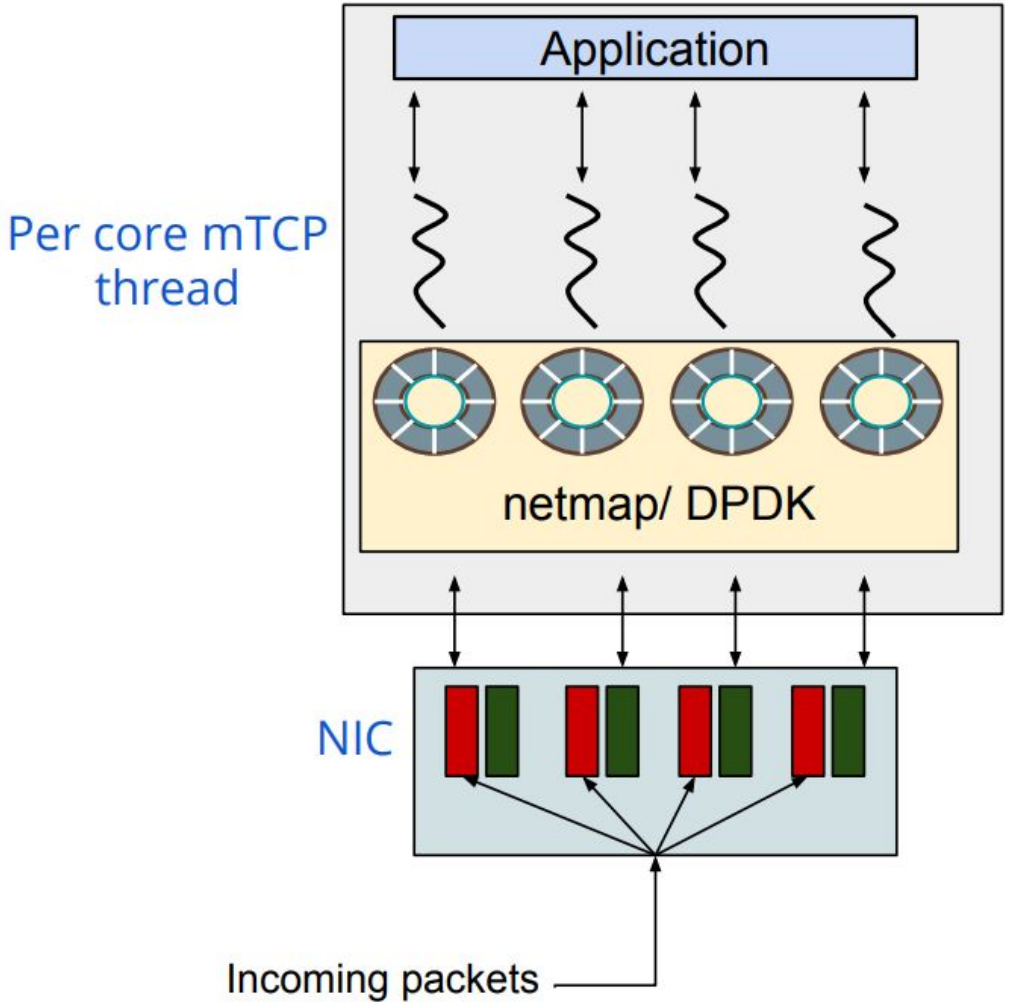


L3/L4 Processing

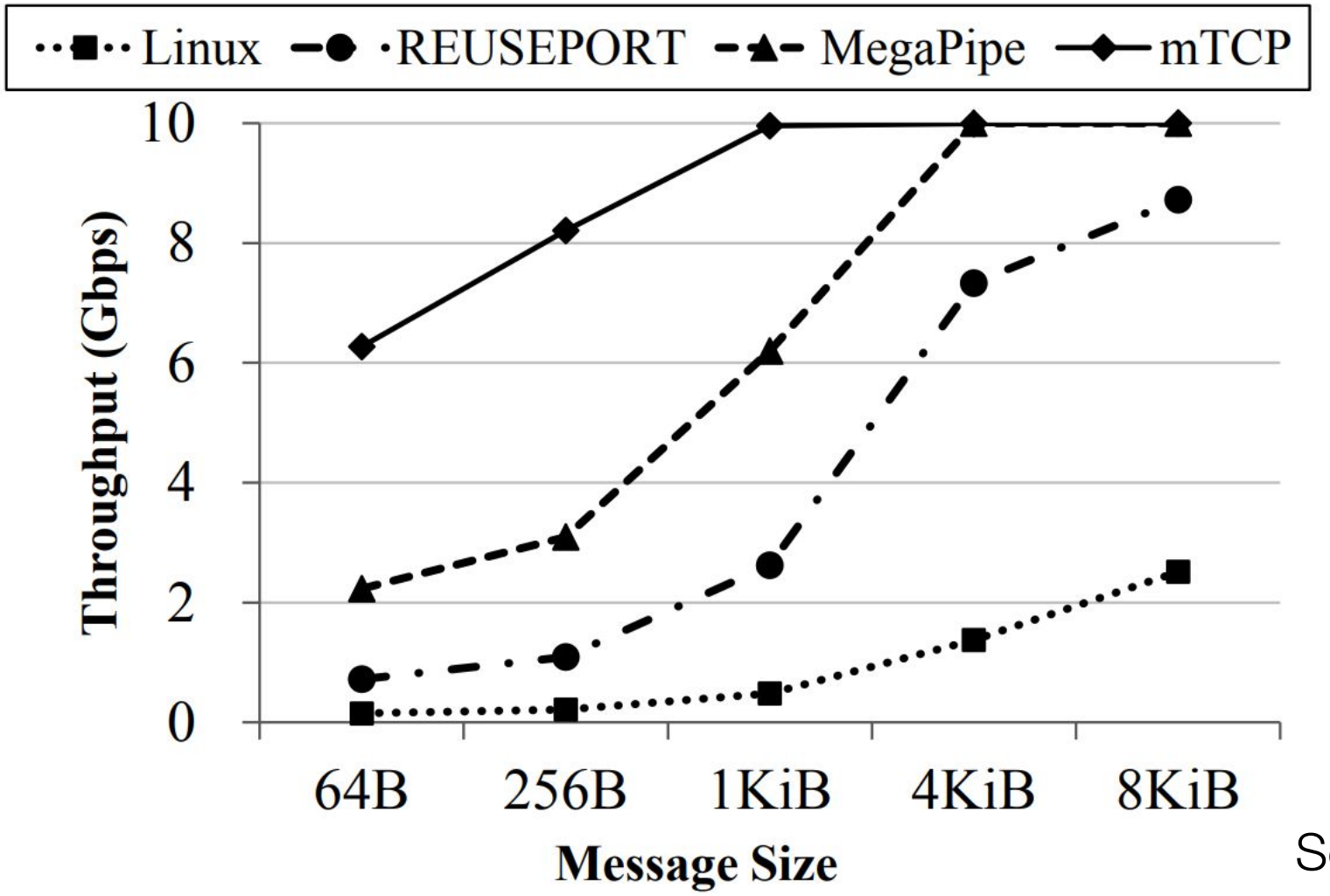


mTCP: Userspace network stack

- User-space TCP/IP stack built over kernel bypass packet I/O engines(e.g. DPDK)
- Designed for multicore scalable application
- Per core TCP data structures
  - E.g. accept queue, socket list
  - Lock free
  - Connection locality
- Leverages multiqueue support of NIC
- No shared data structures

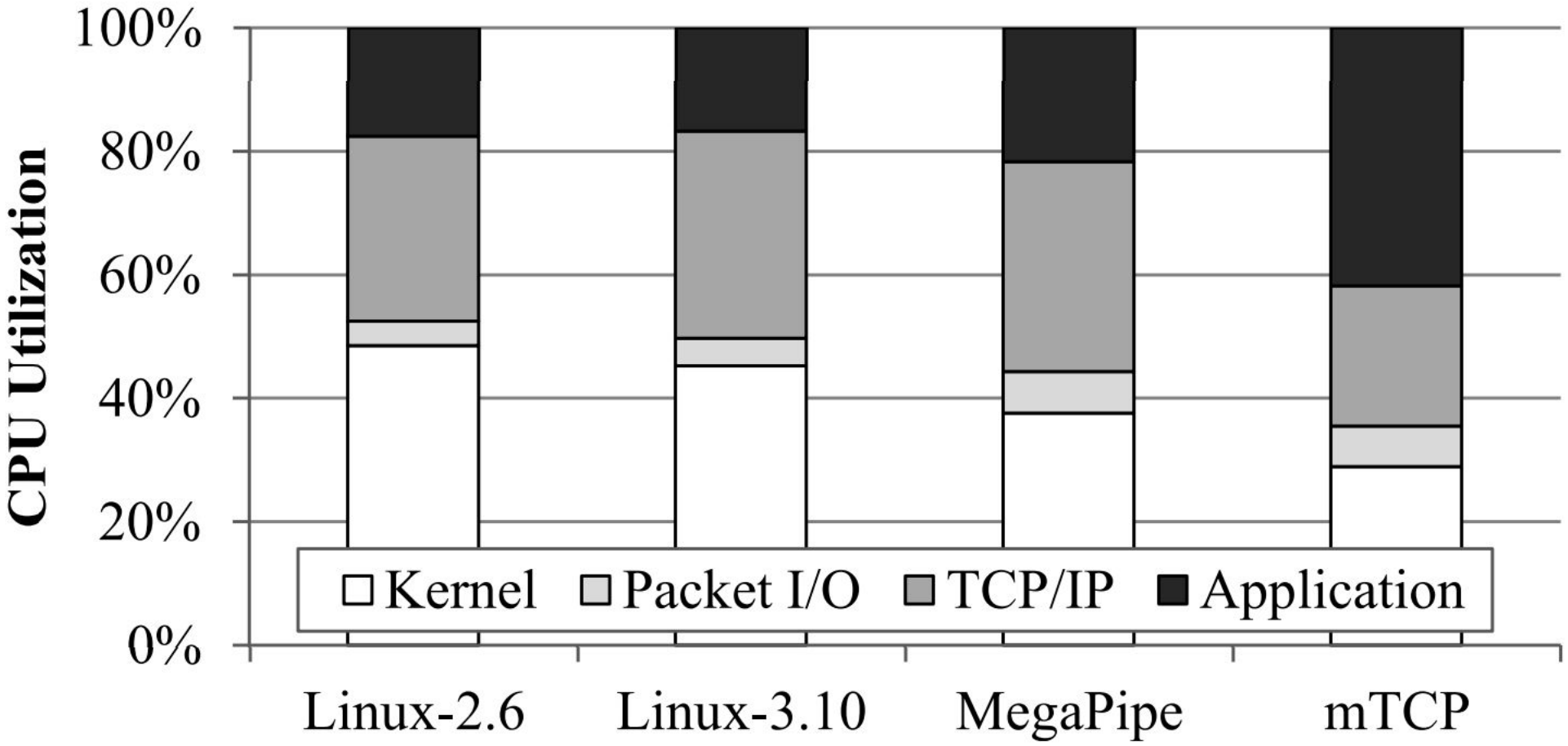


mTCP Performance



Source: mTCP paper

mTCP Performance (Contd.)



Source: mTCP paper

References

- More about DPDK: Link
- Netmap Presentation (Usenix ATC 2012): Link
- Netmap Paper: Link, Implementation: Link
- mTCP Paper: Link, Implementation: Link
- Cloudflare blog on Kernel Bypass: Link