Kernel-bypass techniques for high-speed network packet processing

CS 744

Presenters: Rinku Shah, Priyanka Naik

{rinku, ppnaik}@cse.iitb.ac.in

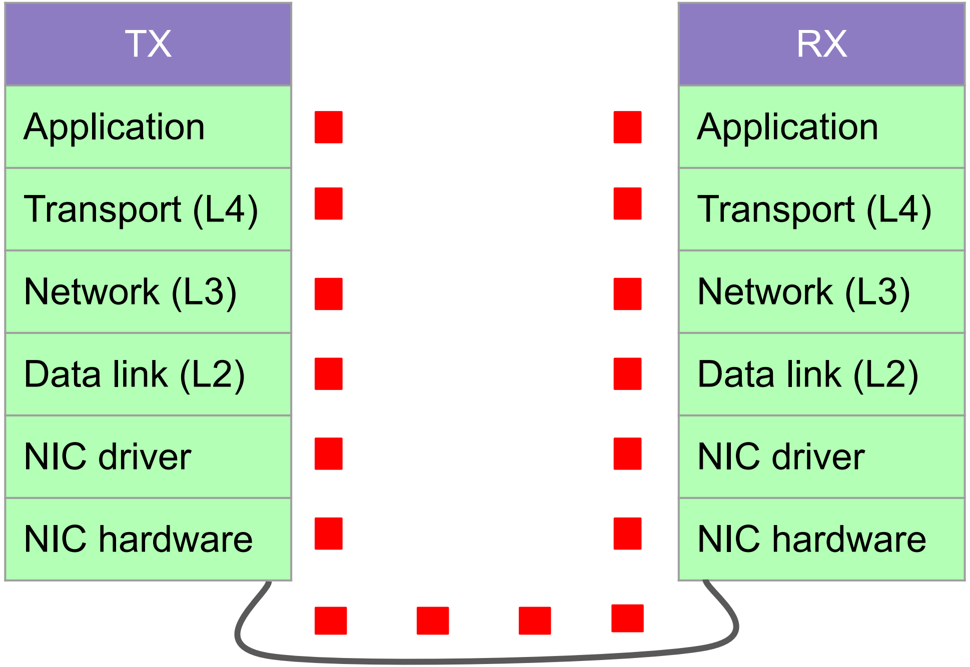
Course Instructor: Prof. Umesh Bellur

Department of Computer Science & Engineering
Indian Institute of Technology Bombay

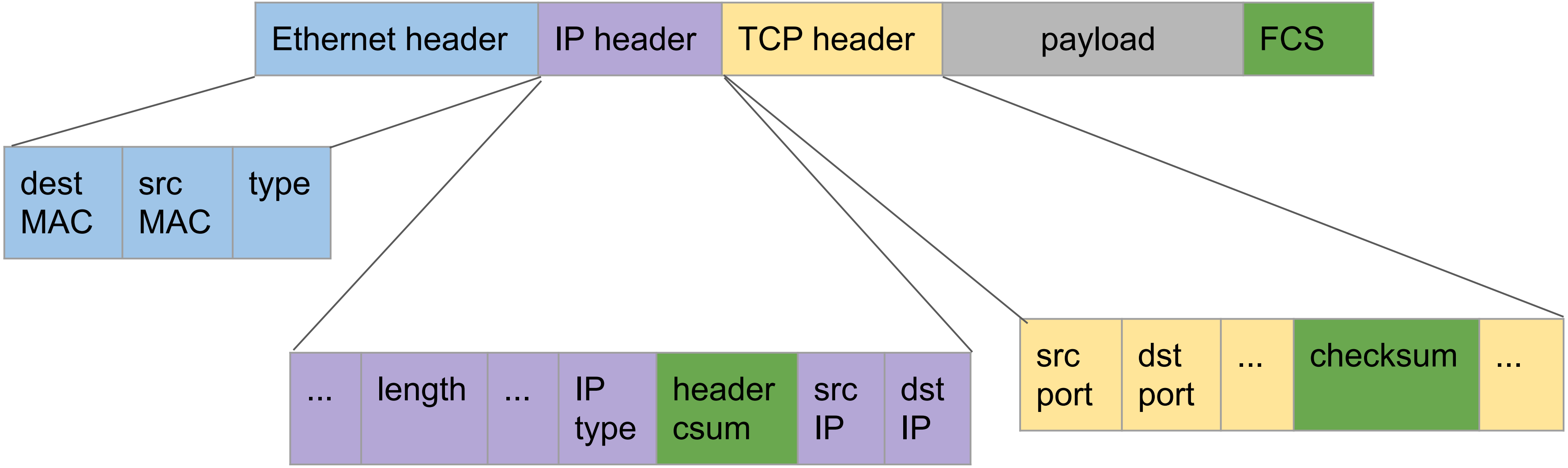
Outline

- The journey of a packet through the Linux network stack
- Need for kernel bypass techniques for packet processing
- Kernel-bypass techniques
  - User-space packet processing
    - **Data Plane Development Kit (DPDK)**
    - **Netmap**
  - User-space network stack
    - **mTCP**
- What's trending?

Typical packet flow



What does a packet contain?

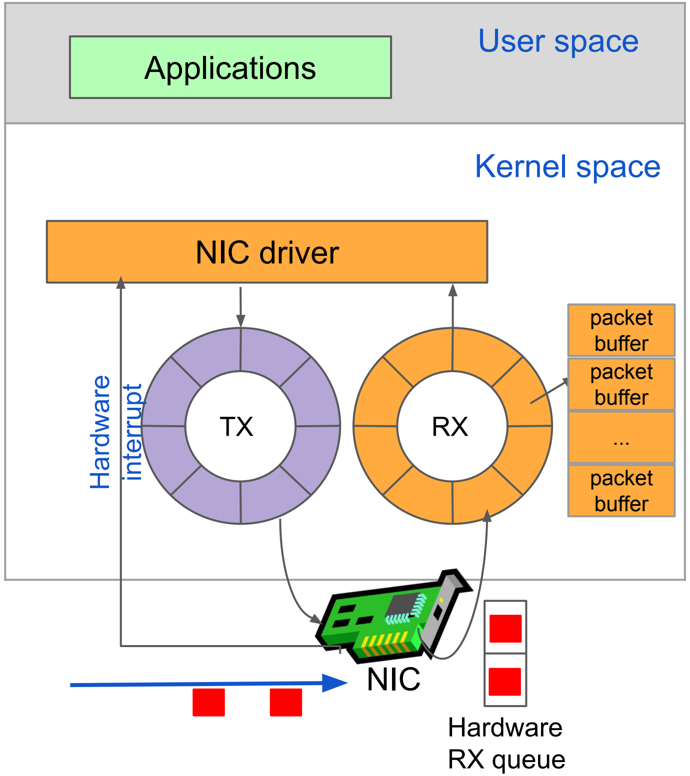


FCS: Frame Check Sequence

Outline

- **The journey of a packet through the Linux network stack**
- Need for kernel bypass techniques for packet processing
- Kernel-bypass techniques
  - User-space packet processing
    - **Data Plane Development Kit (DPDK)**
    - **Netmap**
  - User-space network stack
    - **mTCP**
- What’s next??

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

RX
Application
Transport (L4)
Network (L3)
Data link (L2)
NIC driver
NIC hardware

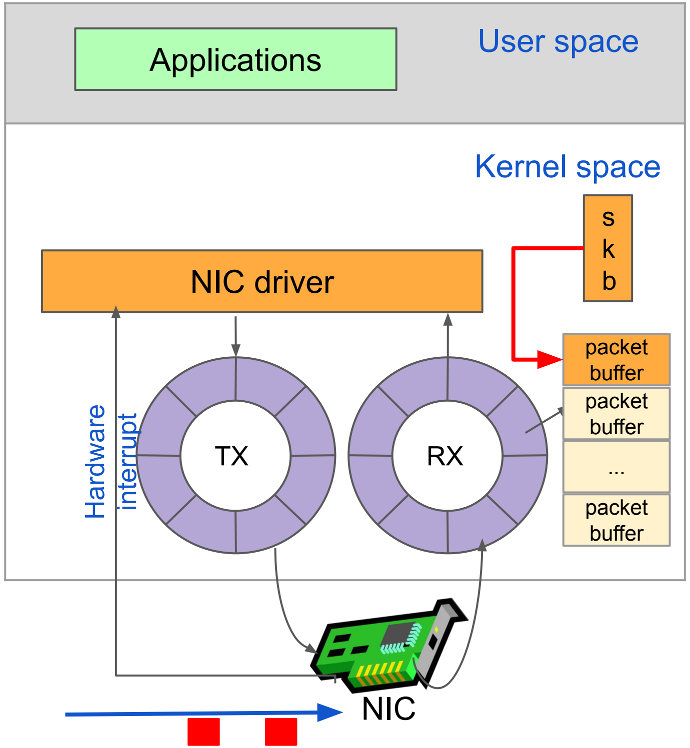
CPU interrupts the process in execution

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)

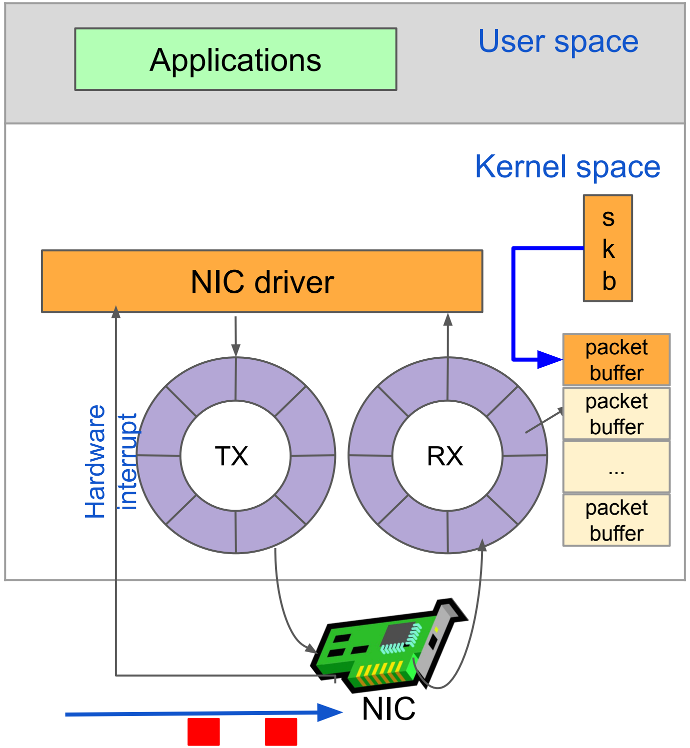
Oops!!

sk-buff (sk-buff tutorial link)

In-memory data structure that contains packet metadata

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

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

NIC hardware

Common processing

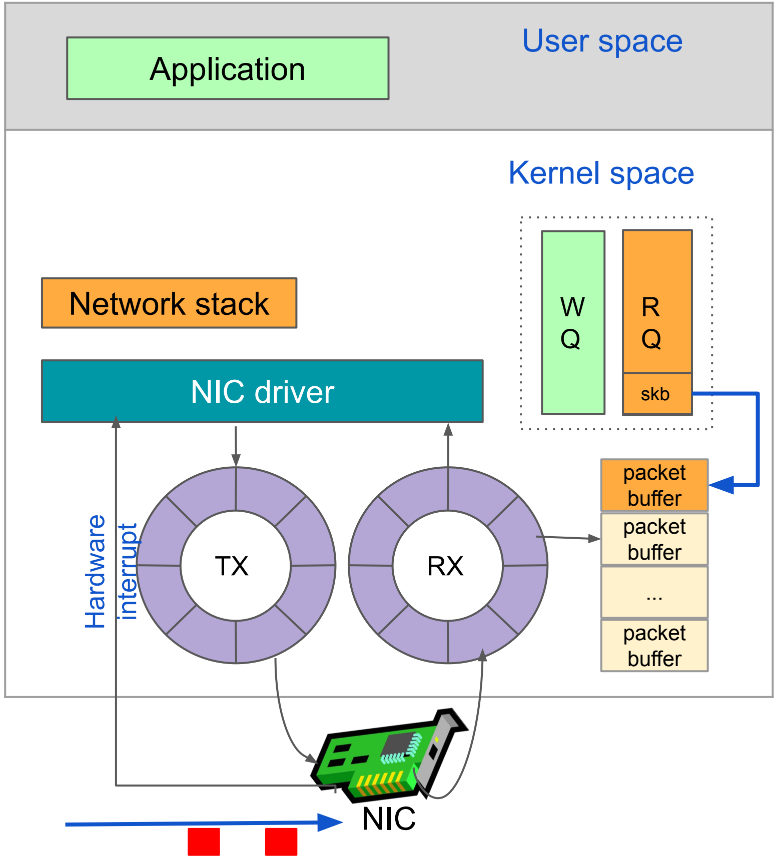
1. Match destination IP/socket
2. Verify checksum
3. Remove header

L3-specific processing

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

L4-specific processing

L3/L4 processing



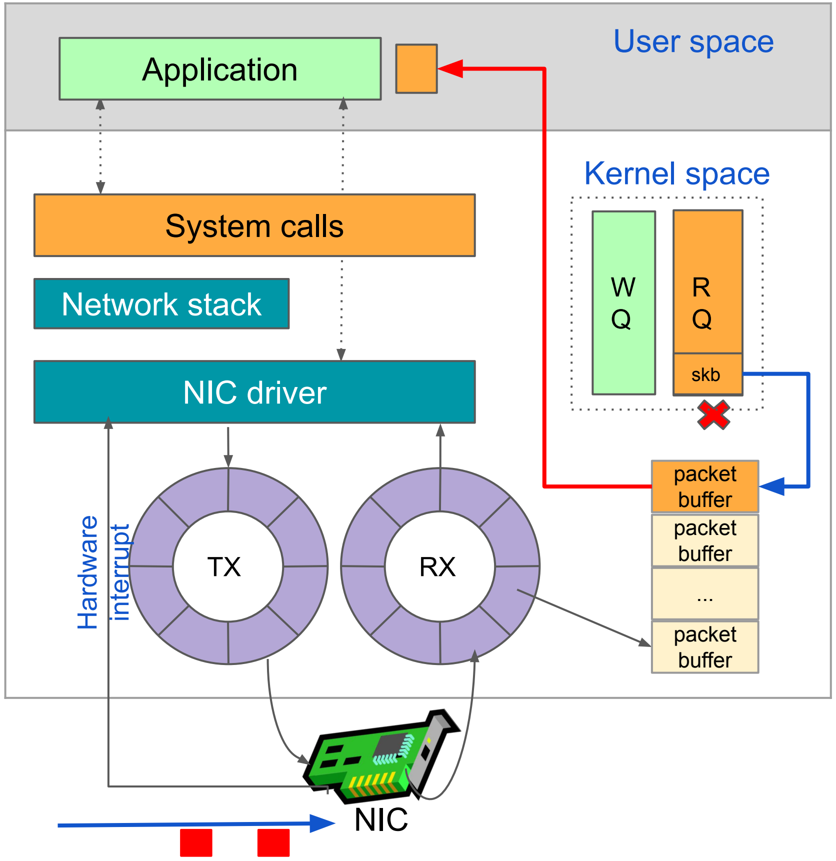
L3-specific processing

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

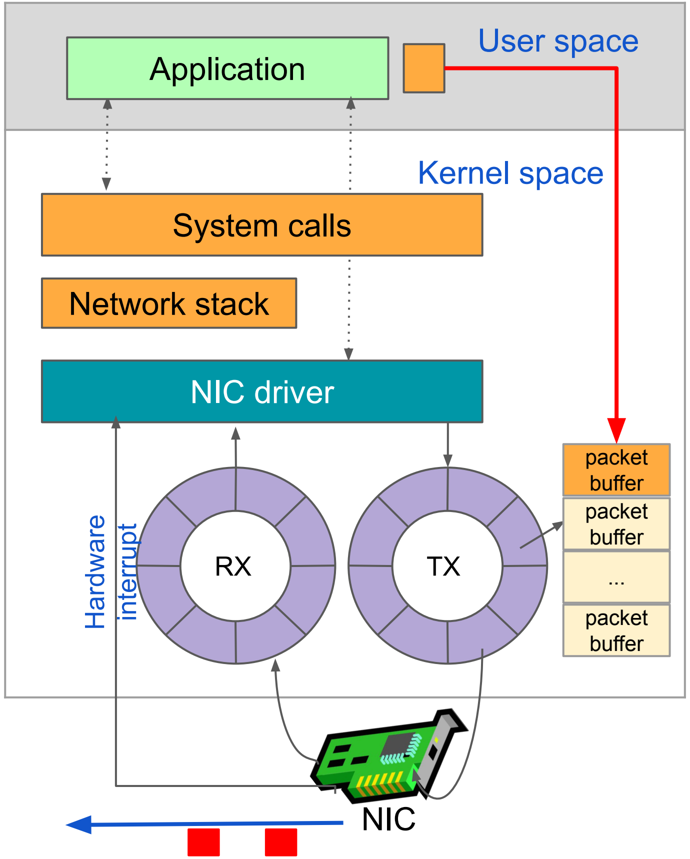


On socket read: user space to kernel space

- 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

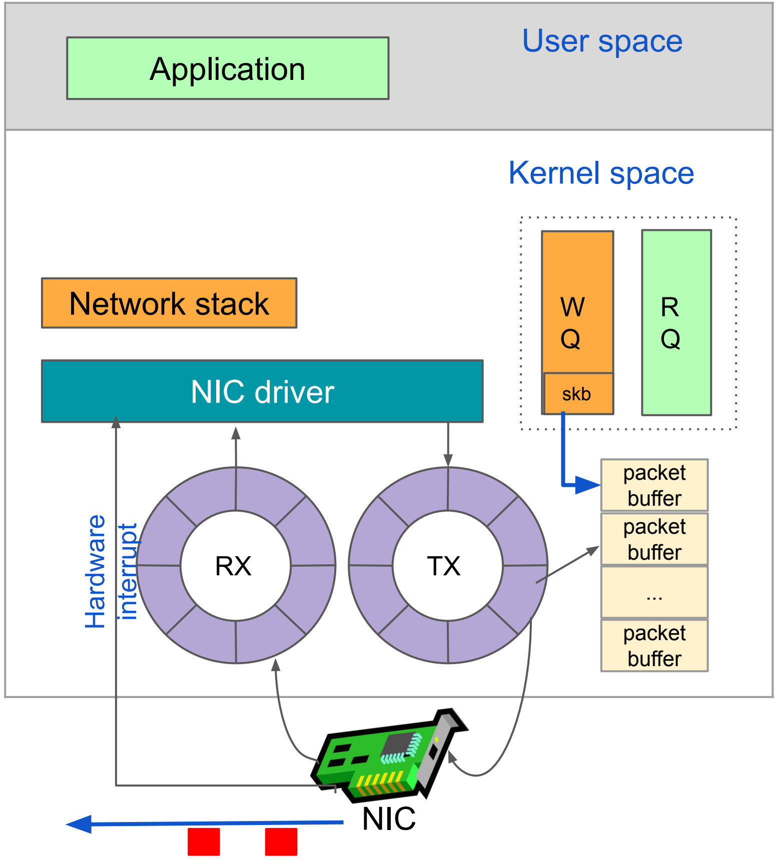
Transmit path of an application packet



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

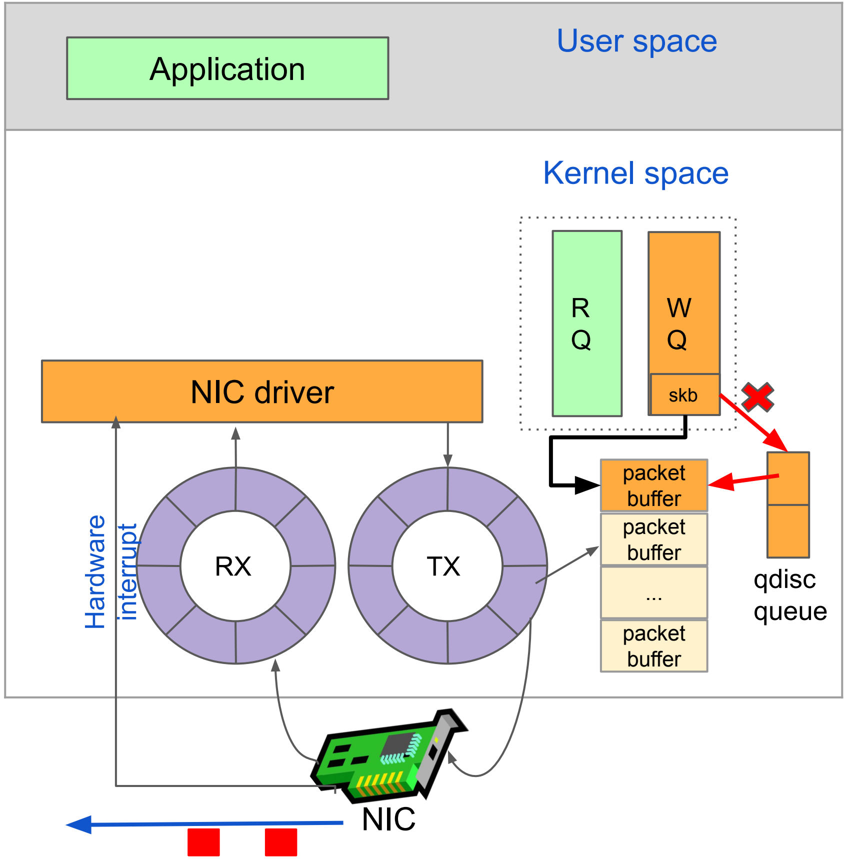
Common processing

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



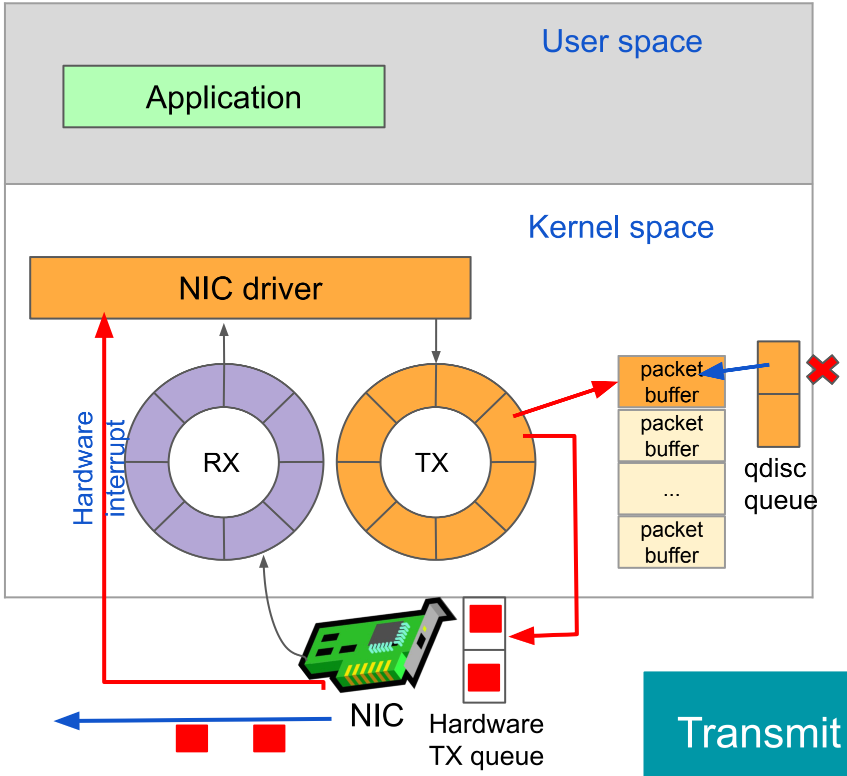
Enqueue packet to queue discipline (qdisc)

- 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
  - Tells 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)

Optimizations to accelerate kernel packet processing

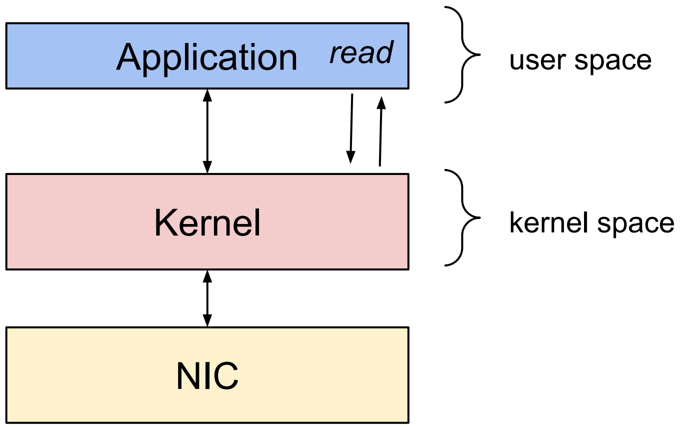
- NAPI (New API) Reading link
- GRO (Generic Receive Offload) GRO+GSO
- GSO (Generic Segmentation Offload) GRO+GSO with DPDK
- Use of multiple hardware queues Multiqueue NIC, Supplement: RSS+RPS+...
- ...

Outline

- The journey of a packet through the Linux network stack
- **Need for kernel bypass techniques for packet processing**
- Kernel-bypass techniques
    - User-space packet processing
        - **Data Plane Development Kit (DPDK)**
        - **Netmap**
    - User-space network stack
        - **mTCP**
- What's trending?

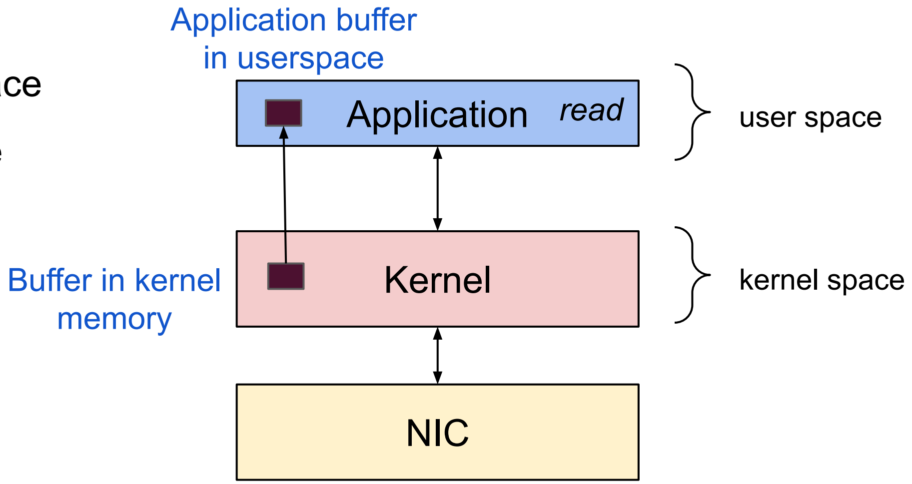
Packet Processing Overheads in Kernel

- Context switch between kernel and userspace



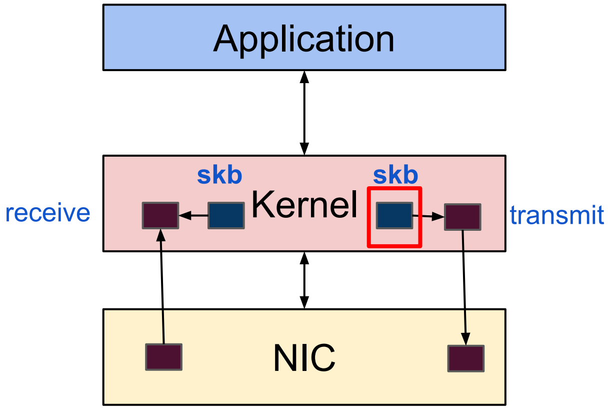
Packet Processing Overheads in Kernel

- Context switch between kernel and userspace
- Packet copy between kernel and userspace

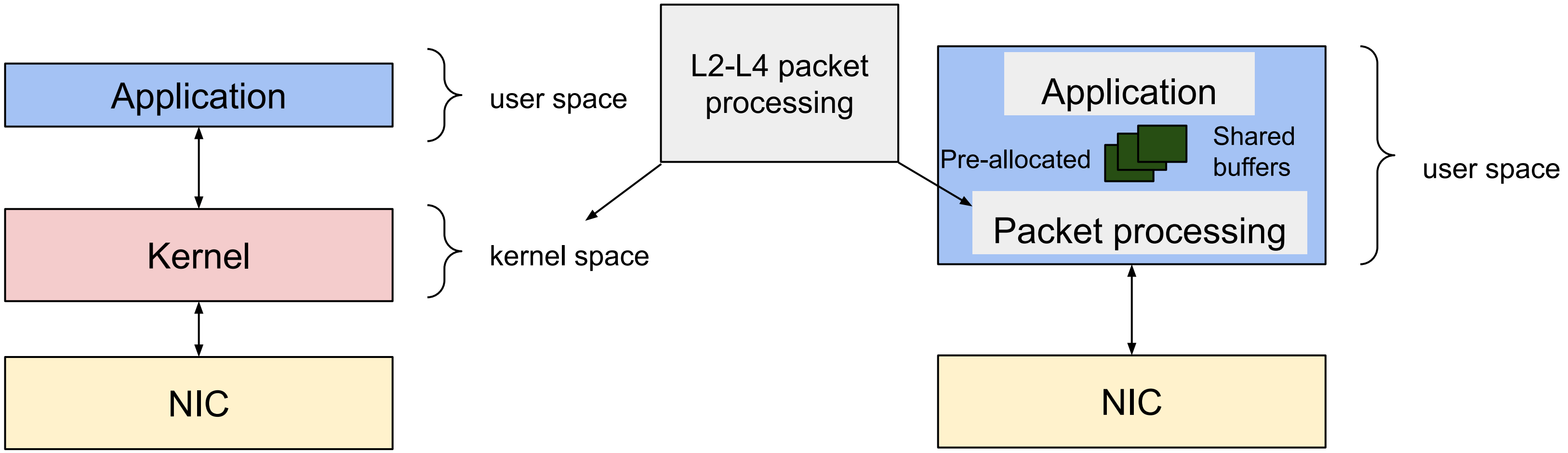


Packet Processing Overheads in Kernel

- Context switch between kernel and userspace
- Packet copy between kernel and userspace
- Dynamic allocation of sk_buff
- Per packet interrupt
- Shared data structures



Overcome Overheads in Kernel: Bypass the kernel



- ✖ Context switch between kernel and userspace
- ✖ Packet copy between kernel and userspace
- ✖ Dynamic allocation of sk_buff

Interrupt vs Poll Mode

Interrupt Mode

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

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

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

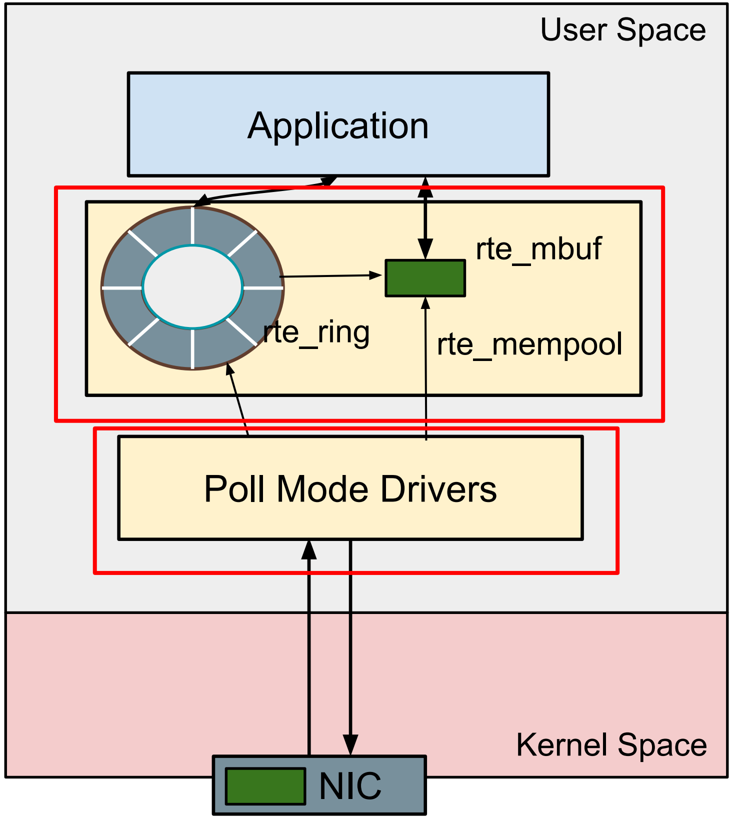
DPDK

Outline

- The journey of a packet through the Linux network stack
- Need for kernel bypass techniques for packet processing
- **Kernel-bypass techniques**
    - User-space packet processing
        - **Data Plane Development Kit (DPDK)**
        - **Netmap**
    - User-space network stack
        - **mTCP**
- What’s trending?

Intel Data Plane Development Kit (DPDK)

- Poll mode user space drivers (uio)
  - Unbinds NIC from kernel
- Mempool: HUGE pages to avoid TLB misses.
- Rte_mbuf: metadata+ pkt buffer
- Cooperative multiprocessing
  - Safe for trusted application

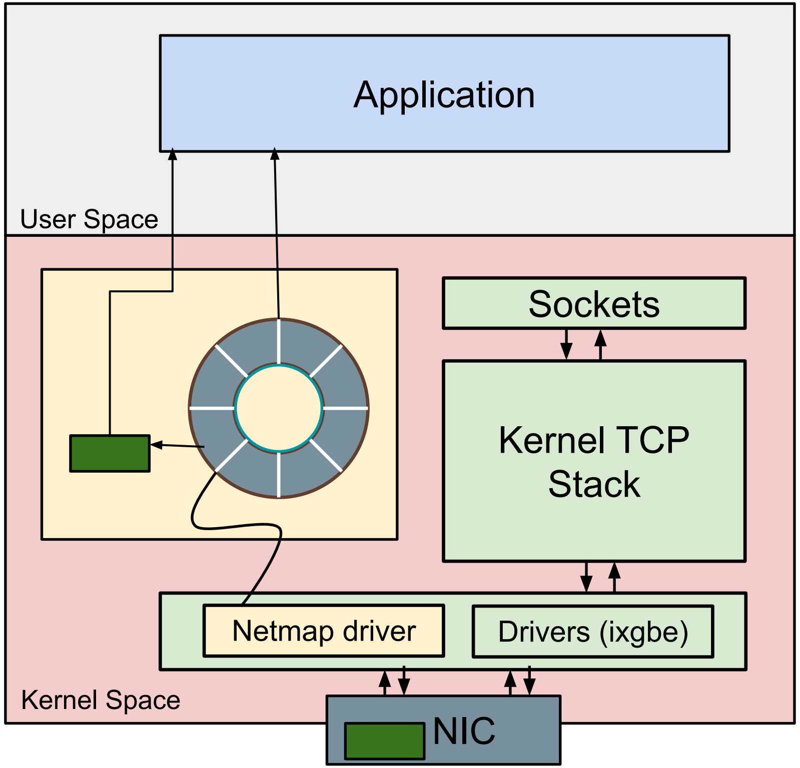


DPDK

Netmap

- Netmap Rings are memory regions in kernel space shared between application and kernel
- No extra copy of a packet
- NIC can work with netmap as well as kernel drivers (transparent mode)

DPDK, netmap manage processing till L2 of network stack



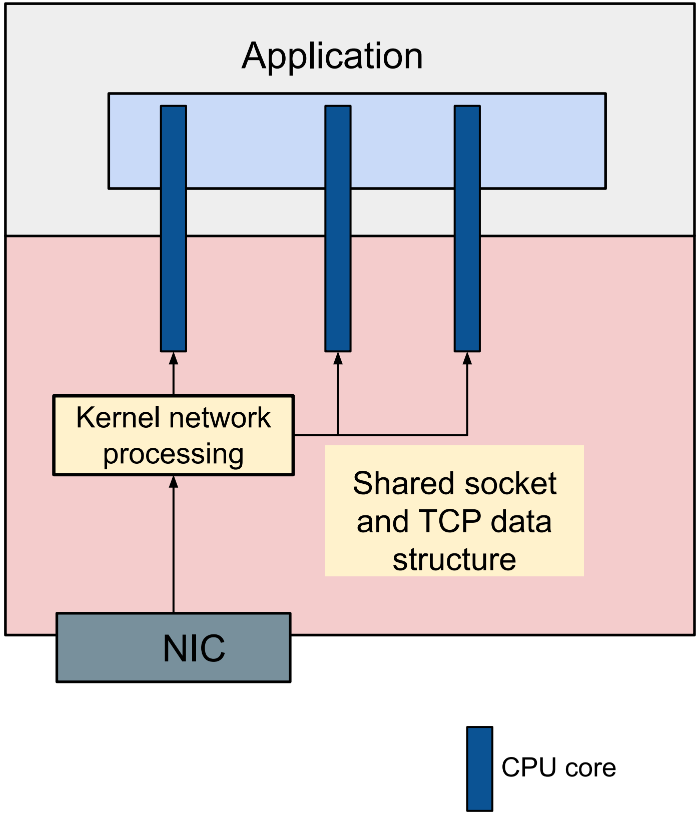
netmap

What about L3-L7 processing?

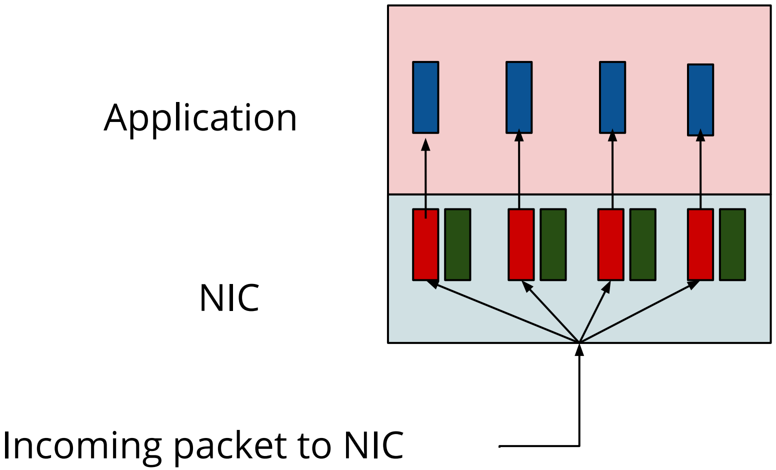
- Overheads with L3-L7 processing in kernel
  - Shared data structure

- Userspace network stack
  - Over netmap or DPDK

- mTCP: multicore TCP

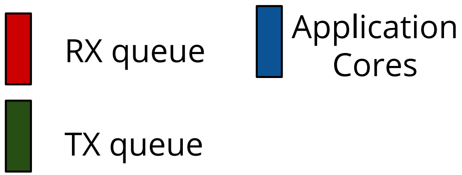


Multiqueue NIC



Receive Side Scaling (RSS)

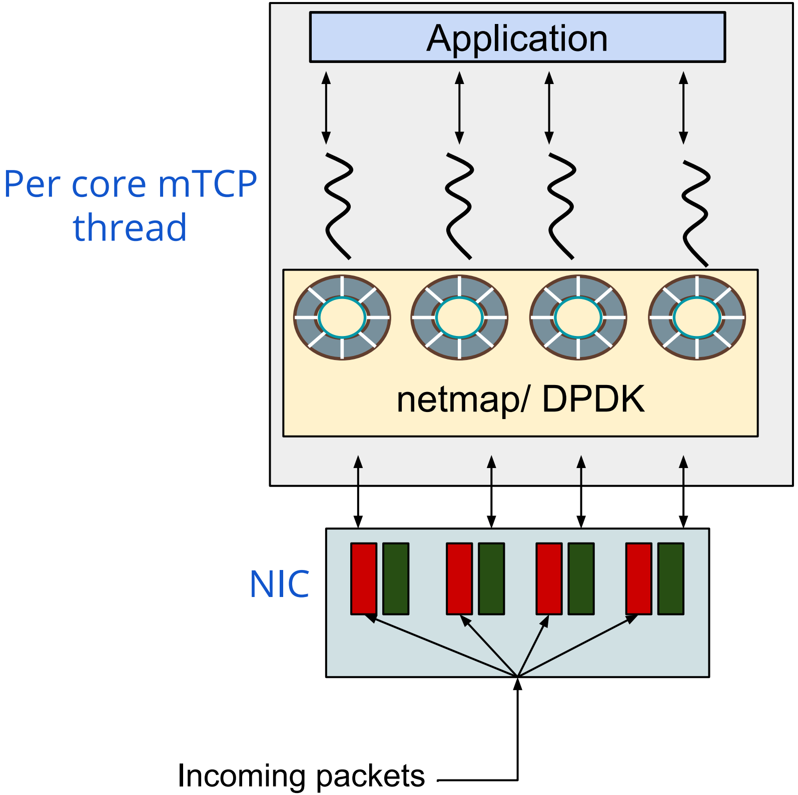
Hash of (src_ip, dst_ip, src_port, dst_port)



mTCP: Userspace network stack

- 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

✖ Shared data structures



[mTCP]

Outline

- The journey of a packet through the Linux network stack
- Need for kernel bypass techniques for packet processing
- Kernel-bypass techniques
  - User-space packet processing
    - **Data Plane Development Kit (DPDK)**
    - **Netmap**
  - User-space network stack
    - **mTCP**
- **What's trending?**

What’s trending?

- Offload application processing to the kernel
  - BPF (Berkeley Packet Filter)
  - eBPF (eXtended BPF) BPF+eBPF+XDP link-1, BPF+eBPF+XDP tutorial link-2
- Offload application processing to the NIC driver
  - XDP (eXpress DataPath) Sample apps for eBPF + XDP
- Offload application processing to programmable hardware
  - Programmable SmartNICs (NPU/DPU)
    - Netronome, Mellanox, Bluefield, Pensando Video on smartNIC architecture + Netronome NIC specifics
  - Programmable FPGAs
    - Xilinx, Altera
  - Programmable hardware ASICs Programmable network: Intro video , Detailed video link
    - Barefoot Tofino, Cisco’s Doppler, Intel Flexpipe, Cavium’s Xpliant