Exercise for Lecture
Software Defined Networking

Prof. Dr. David Hausheer, Julius Rückert
Christian Koch, Jeremias Blendin, Leonhard Nobach, Matthias Wichtlhuber

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter Term 2016/17
Exercise No. 6
Published: 10.01.2017
Submission exclusively via Moodle, Deadline: 24.01.2017
Contact: Please use the Moodle forum to post questions and remarks on the exercise.
Web: `http://www.ps.tu-darmstadt.de/teaching/ws1617/sdn/`
Submission: `https://moodle.tu-darmstadt.de/course/view.php?id=8385`

Surname (Nachname):	Chen
First name (Vorname):	Zhen
ID# (Matrikelnummer):	2665935

Team member: Letian Feng(2255840), Chunyuan Yu(2587628)

Problem 6.1 - DPDK Basics

For the following tasks, we refer to the DPDK documentation, especially

- https://en.wikipedia.org/wiki/New_API (Not part of DPDK, but of the Linux kernel)
- http://dpdk.org/doc/guides-2.1/prog_guide/poll_mode_drv.html
- http://dpdk.org/doc/guides/prog_guide/mbuf_lib.html
- http://dpdk.org/doc/api/rte__byteorder_8h.html

You can answer in free text to solve this problem, writing code is optional / not necessary.

a) Interrupts vs. Poll-Mode Drivers

The "New API" (not part of DPDK) is a Linux-kernel effort to speed up network processing. It achieves this by finding a trade-off between interrupts and polling.

I What are the advantages of polling, and the advantages of interrupts?

Advantages of Interrupts: CPU responses immediately when a task arrives instead of wasting time for asking devices one by one. This feature suits for real time tasks.
Advantages of PMD: In PMD it's the CPU that determines when to handle requests. It's simpler and no special hardware is needed. Further more, since polling policy is determined by CPU, it prevents the situation that CPU is flooded by too many requests.

II Define one network load situation in which you would prefer poll-mode-drivers, and one in which you would prefer interrupts.

Interrupts is suitable for tasks which require low delay.
PMD is suitable for steady traffic with high amount of packets.

b) Mempools / MBuf Structures

Incoming packets are stored in `mbuf` structures. You want to implement a (non-encrypting) tunnel endpoint, which must append an additional header between the Ethernet and the IP header before forwarding the packet. Assume a maximum-sized packet (1500 bytes). **How do you efficiently solve this in DPDK without shifting the entire payload?**

According to Fig. 6.1 & 6.2, headers are saved in the RTE_PKTMBUF_HEADROOM, and multiple rte_mbufs could be segmented together(buffer chaining). So we could allocate a new mbuf with a headroom and a segment of length 0, then we use the functions "prepend/append data before" and "remove data at the beginning of the buffer", which are described in section 6.5, to move the Ethernet header & other control information before it into the first mbuf and append an additional header after it, finally move IP header and the requiement is fulfilled.

c) DPDK and Endianness

Explain why the functions of `rte_byteorder.h` are useful if you want to manipulate or do calculations on IP packet headers on an x86-based system.

rte_byteorder.h contains functions which define a generic API for byte swap operations. Since x86 is small endian and IP packet is designed to big endian, swap functions declared in rte_byteorder.h are needed to swap the position of low and high bytes.

d) NFV Performance

Assume a packet of 1500 bytes.

- NF 1 checks if the payload contains a specific signature of at least 16 bytes anywhere in the payload and, in case, drops the packet.
- NF 2 encapsulates a packet in a VLAN if it originates from a list of 20.000 exact, well-known malicious IP addresses.
- NF 3 performs generic routing encapsulation (GRE) of all input traffic.

Estimate the relative performance of the three NFs you can achieve when implementing it in DPDK, based on the expected processing complexity. Order them by your estimation, and provide the reasons for your answer. ***(Example: NFx should achieve the highest performance, because ..., NFy achieves medium performance, because ...)***

NF3 should achieve the highest performance, because the only task is the encapsulation of GRE and IP and Ethernet(or aother 2-layer protokol) headers, the performance depends almost only on NIC;
NF1 achieves medium performan, because except for decapsulation of all headers before payload(e.g. Ethernet, IP, TCP), it has to execute a search program for a 16 bytes signature, the performance depends on NIC and CPU;
NF3 achieves the lowest performance, because except for decapsulation of Ethernet and IP headers, it has to compare the source ip address to 20.000 ip addresses(32 bits * 20.000 = 640.000 bits = 80.000bytes), the performance depends on NIC, CPU and storage/memory(much slower).

Problem 6.2 - DPDK Code Study

```c

\\...packet receive code omitted...

const struct ether_addr* intAddr2set = {.addr_bytes={0x52,0x54,0x00,0x00,0x00,0x01}}
uint32_t my_maskv4 = IPv4(255,255,255,0);
uint32_t my_ipv4 = IPv4(192,168,0,1);

struct arp_hdr* arp_hdr = rte_pktmbuf_mtod_offset(packet, struct arp_hdr*, sizeof(struct ether_hdr) + vlan_offset);

if (__bswap_16(arp_hdr->arp_op) == 1) {

  struct arp_ipv4* arpdata = &arp_hdr->arp_data;

  uint32_t swapped_tip = __bswap_32(arpdata->arp_tip);

  if (swapped_tip == my_ipv4) {

    if ( /* TODO */) {
      RTE_LOG(INFO, USER1, "   Sent from a wrong subnet...\n");
      return 0;
    }

    memcpy(&l2hdr->d_addr, &arpdata->arp_sha, sizeof (struct ether_addr));
    memcpy(&l2hdr->s_addr, my_mac, sizeof(struct ether_addr));
    memcpy(&arpdata->arp_tha, &arpdata->arp_sha, sizeof(struct ether_addr));
    memcpy(&arpdata->arp_sha, my_mac, sizeof(struct ether_addr));
    arpdata->arp_tip = arpdata->arp_sip;
    arpdata->arp_sip = __bswap_32(my_ipv4);
    arp_hdr->arp_op = __bswap_16(2);

    struct rte_mbuf* pkt2send_buf[1];
    pkt2send_buf[0] = packet;

    int retrn = rte_eth_tx_burst(IF_IDENTIFIER, 0, pkt2send_buf, 1);
    if (retrn == 1) {
      printf("    Successful.\n");
      return 1;
    } else {
      printf("    NOT successful.\n");
      return -1;
    }
  }
}
```

I This code works on an incoming ARP packet. What is this code supposed to do with it?

This code is supposed to check if the ARP request's destination ip is "my_ipv4" and it's from the same subnet as "my_ipv4&my_maskv4". If yes then modify the request packet's mac & ip addresses in arp header, and the mac address in ethernet header, and finally send it back.

II Add the missing `if` statement checking whether the ARP was sent from a wrong subnet.

__bswap_32(arpdata >arp_sip)&my_maskv4 != my_ipv4&my_maskv4