姓名：胡昊源

学号：518021910269

Part1 Get familiar with DPDK

Q1: What's the purpose of using hugepage?

hugepages 的出现是因为摩尔定律导致了主机内存越来越大，最后不得不对于标准 pages 的改进以提高效率，在需要相同内存的情况下减少页表项，从而能减少 TLB miss 率，提升性能。

Q2: Take examples/helloworld as an example, describe the execution flow of DPDK programs?

以 hello world 为例，其程序源代码如下：

```
{
    int ret;
    unsigned lcore_id;

    ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_panic( “Cannot init EAL\n” );

    /* call lcore_hello() on every slave lcore */
        RTE_LCORE_FOREACH_SLAVE(lcore_id) {
        rte_eal_remote_launch(lcore_hello, NULL, lcore_id);
    }

    /* call it on master lcore too */
    lcore_hello(NULL);

    rte_eal_mp_wait_lcore();
    return 0;
}
```

对于所有的 DPDK 程序来说，流程如下：

1、初始化基础运行环境

```
int rte_eal_init(int argc, char **argv)
```

该函数读取入门参数，解析并保存作为 DPDK 运行的系统信息，构建一个针对包处理的运行环境。

2、初始化多核运行环境

```
RTE_LCORE_FOREACH_SLAVE（lcore_id）
```

该函数遍历所有 EAL 指定的可以使用的 lcore，然后通过 rte_eal_remote_launch 在每个 lcore 上，启动指定的线程。

```
int rte_eal_remote_launch(int (*f)(void *), void *arg, unsigned slave_id);
```

该函数第一个参数是被征召的线程，第二个参数是传给从线程的参数，第三个参数是指定的逻辑核，是从线程执行的核。

3、执行代码

lcore_hello 的源代码如下图所示：

```
static int
lcore_hello(__attribute__((unused)) void *arg)
{
    unsigned lcore_id;
    lcore_id = rte_lcore_id();
    printf("hello from core %u\n", lcore_id);
    return 0;
}
```


这里运行函数 lcore_hello，它读取自己的逻辑核编号 lcore_id，打印相应的数值。

Q3: Read the codes of examples/skeleton, describe DPDK APIs related to sending and receiving packets.

skeleton 的源代码如下图所示：

```
int main(int argc, char *argv[])
{
    struct rte_mempool *mbuf_pool;
    unsigned nb_ports;
    uint8_t portid;

    /* Initialize the Environment Abstraction Layer (EAL). */
    int ret = rte_eal_init(argc, argv);

    /* Check that there is an even number of ports t send/receive on. */
    nb_ports = rte_eth_dev_count();
    if (nb_ports < 2 || (nb_ports & 1))
        rte_exit(EXIT_FAILURE, "Error: number of ports must be even\n");

    /* Creates a new mempool in memory to hold the mbufs. */
    mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
        MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());

    /* Initialize all ports. */
    for (portid = 0; portid < nb_ports; portid++)
        if (port_init(portid, mbuf_pool) != 0)
            rte_exit(EXIT_FAILURE, "Cannot init port %"PRIu8 "\n",
                    portid);

    /* Call lcore_main on the master core only. */
    lcore_main();
    return 0;
}
```

DPDK 有关收发包的 API：

1、网口初始化

```
port_init(uint8_t port, struct rte_mempool *mbuf_pool)
```

首先我们对指定端口设置队列数，在收发两个方向 上，基于端口与队列进行配置设置，缓冲区进行关联设置。

2、网口设置

```
int rte_eth_dev_configure(uint8_t port_id, uint16_t nb_rx_q, uint16_t nb_tx_q,
const struct rte_eth_conf *dev_conf)
```

对指定端口设置接收、发送方向的队列数目，依据配置信息来指定端口功能。

3、队列初始化

```
int rte_eth_rx_queue_setup(uint8_t port_id, uint16_t rx_queue_id, uint16_t
nb_rx_desc, unsigned int socket_id, const struct rte_eth_rxconf *rx_conf, struct
rte_mempool *mp)

int rte_eth_tx_queue_setup(uint8_t port_id, uint16_t tx_queue_id,uint16_t
nb_tx_desc, unsigned int socket_id, const struct rte_eth_txconf *tx_conf)
```

对指定端口的某个队列，指定内存、描述符数量、报文缓冲区，并且对队列进行配置。

4、网口设置

```
int rte_eth_dev_start(uint8_t port_id)
```

初始化配置结束后，启动端口。

5、收发报文

```
static inline uint16_t rte_eth_rx_burst(uint8_t port_id, uint16_t queue_id,
struct rte_mbuf **rx_pkts, const uint16_t nb_pkts)

static inline uint16_t rte_eth_tx_burst(uint8_t port_id, uint16_t queue_id,
struct rte_mbuf **tx_pkts, uint16_t nb_pkts)
```

这里的参数，含义分别是：端口，队列，报文缓冲区以及收发包数。

Q4: Describe the data structure of 'rte_mbuf'.

rte_mbuf 是一种 buf 管理的结构体。

一般 rte_mbuf 数据都会有分 3 个区域：headroom、data、 tailroom。在保存报文的内存块前后分别保留 headroom 和 tailroom，以方便应用解封报文。Headroom 默认 128 字节，可以通过宏 RTE_PKTMBUF_HEADROOM 调整。整个 buf 的大小，也就是数据结构中的 buf_len 的大小，一般是4096个字节。

headroom：
一般用来存放用户自己针对于 mbuf 的一些描述信息，一般保留给用户使用，可以通过修改 mbuf 头文件，来实现 headroom 的大小；
data_off 的默认值就是 mbuf 的 headroom 的大小，默认是 128。如果要定义超过这个范围的私有字段，可以修改 RTE_PKTMBUF_HEADROOM

data：
data 区域一般指的是地址区间在 buf_addr + data_off 到 buf_add + data_off + data_len 之间的区域。data_len 就是这段数据的长短，这个 data_len 一般都是通过 mbuf 的几个基本操作，或者通过赋值来实现的。

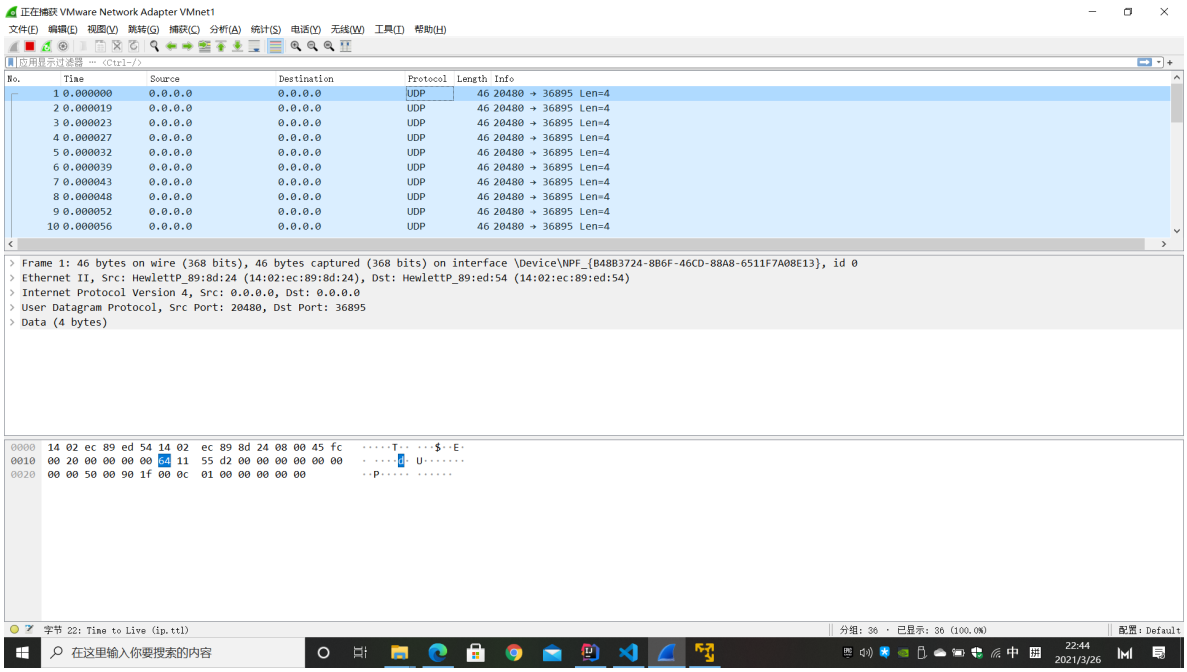
tailroom：
一般指的是，data_len 还未包含的东西。默认其实data_len是0。所以说默认来说tailroom应该是占了很大的空间的；

mbuf 的控制，就是不断的控制这个几个区域的大小，报文数据存放在 data 中，主要控制的就是 data_off 与 data_len；

此外还有一些常量，比如类型，所属的 mempool，端口等等。

Part2 Send UDP

我的 UDP 发送成功的截图



包头格式：必须按照协议规范进行才能正确通过协议栈，并且 Ether/IP/UDP 不能均要具备

包大小限制: UDP 包大小必须小于 IP 的 Payload，大于 UDP 包头长度。

端口监听: 虚拟机的设置中设置的指定 Adapter