SOFTWARE DEFINED NETWORKS

LAB 3

AKHILA NAIR

STUDENT ID: 01743358

HAND IN & DUE DATE: 23rd March 2018

Estimated Hours of work: 20

1. **Purpose:**
    - The purpose of lab 3 is that since now we know how to use DPDK to send and receive packets, now we have to combine both Tx and Rx in the same code.
    - Other important task is that we not only need to use hardware Tx and Rx queue but also software queue to transfer data between threads.
    - Next task is to learn how to use ring buffer to design such software queue.

2. **Setup:**
    - The setup remains the same as lab2, so it goes as following.
    - Connect the laptop to DPDK box using the Ethernet cable via RJ45 Adapter.
    - Go to network centre and change the settings if not already set.
    - Now SSH into the system to log in the DPDK box.
    - Now use putty to change the directory by using "cd /home/test.dpdk/lab3".
    - Choose the file you want to run for Ex: TX, then "cd TX"
    - Enter the "make" command.
    - Now to build the code use " sudo ./build/tx_demo -w 0000:03:00.0 --socket-mem 64 --file-prefix tx -c 0x02".

    This will run the main.c file from the TX folder.

3. **Software Design:**

    Major software and code changes is explained shortly in the section below,

    - **To initialize a ring structure:**

      *struct rte_ring *RxtoTx;*

      This API is initializing the ring structure. Producer and consumer has head and tail index. This index is in the range between 0 and 2^{32}. This means that we are masking their value when we access the ring field.

    - **To create a ring:**

      ```
      int main(int argc, char **argv)
      {
         app_init(argc, argv);
         RxtoTx = rte_ring_create("RX to TX",2048,0, 1);
         if(RxtoTx == NULL)
        {rte_exit(EXIT_FAILURE, "%s\n", rte_strerror(rte_errno));}
      ```

      In this, we are creating a new ring named Rx to Tx in memory.
      This Function uses memzone_reserve() to allocate memory. It calls rte_ring_init() to initialize an empty ring.
      The parameters passed here are:
      RxtoTx → Name of Ring
      2048 → Size of ring (Count)
      0 → Socket ID

1 → Flags, RING_F_SP_ENQ & RING_F_SC_DEQ

- **Ethernet device Configuration:**

struct app_pktq_hwq_in_params {

uint32_t size;

struct rte_eth_rxconf conf;

};

This is setting up the ethernet device for RX configuration.
The size describes the number of ring descriptor of each RX queue.

struct app_pktq_hwq_out_params {

uint32_t size;

uint32_t burst;

struct rte_eth_txconf conf;

};

This is setting up the ethernet device for TX configuration.
The size describes the number of ring descriptor of each TX queue.

status = rte_eth_dev_configure(pmd_id, 1, 1, &link_temp.conf);

This function is used to configure an Ethernet device.
This function must be invoked first before any other function in Ethernet API.
Parameters passed are
Port ID – Port identifier of ethernet device to configure.
1 – nb_rx_queue, Number of received queue to setup.
1 – nb_tx_queue, Number of transmit queue to setup.
Eth_conf - &link_temp_conf, Pointer to the configuration data to be used for Ethernet device.

- **Setting up the Queue:**

Int rte eth_queue_setup

This function allocates a contiguous block of memory from a memory zone associated with socket ID and initializes each received descriptor with the network buffer allocated from memory pool.

- **Creating a memory pool:**

```
struct rte_mempool * mp;
mp = rte_mempool_create(name,
mempool_params_default.pool_size,
mempool_params_default.buffer_size,
mempool_params_default.cache_size,
sizeof(struct rte_pktmbuf_pool_private),
```

A memory pool is an allocator of fixed size object. It uses ring to store free objects. A pointer is then assigned to point it to the memory pool structure.

- **Promiscous mode:**

```
if (link_temp.promisc)
{
rte_eth_promiscuous_enable(pmd_id);
}
```

This function will enable the promiscuous mode. This means that this port will receive all the packets.

- **To start the link:**

static void app_init_link()
rte_eth_dev_configure():
This function is used to configure the number of queues for a port. For each port there is only one RX queue. The number of TX queues depend on the number of lcores.

```
ret = rte_eth_dev_configure((uint8_t) portid, 1, 1, &port_conf);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Cannot configure device: err=%d, port=%u\n", ret,
                                                                    portid);
```

- **To set up the EAL:**

  *static void app_init_eal(int argc, char **argv)*
  {
  int status;
  status = rte_eal_init(argc, argv);
  if (status < 0) { rte_panic("EAL init error\n");
  }

- **Enqueue Process:**

  rte_ring_sp_enqueue(RxtoTx, pkts[i])

  This function allows you to enqueue one object on ring.
  Parameters passed are:
  r: A pointer to the ring structure
  obj: A pointer to the object to be the added
  So RxtoTx is pointer to the ring structure.
  And pkts [i] because we have to enqueue each packets.

- **Retrieving input packets from Ethernet device:**

  n_pkts = rte_eth_rx_burst(0, 0, pkts, 1);

  Retrieve a burst of input packets from a receive queue of an Ethernet device. The retrieved packets are stored in *rte_mbuf* structures whose pointers are supplied in the *rx_pkts* array.

  The rte_eth_rx_burst() function loops, parsing the RX ring of the receive queue, up to *nb_pkts* packets, and for each completed RX descriptor in the ring, it performs the following operations:
  Initialize the *rte_mbuf* data structure associated with the RX descriptor according to the information provided by the NIC into that RX descriptor.
  Store the *rte_mbuf* data structure into the next entry of the *rx_pkts* array.
  Replenish the RX descriptor with a new *rte_mbuf* buffer allocated from the memory pool associated with the receive queue at initialization time.

- **Mbuff:**

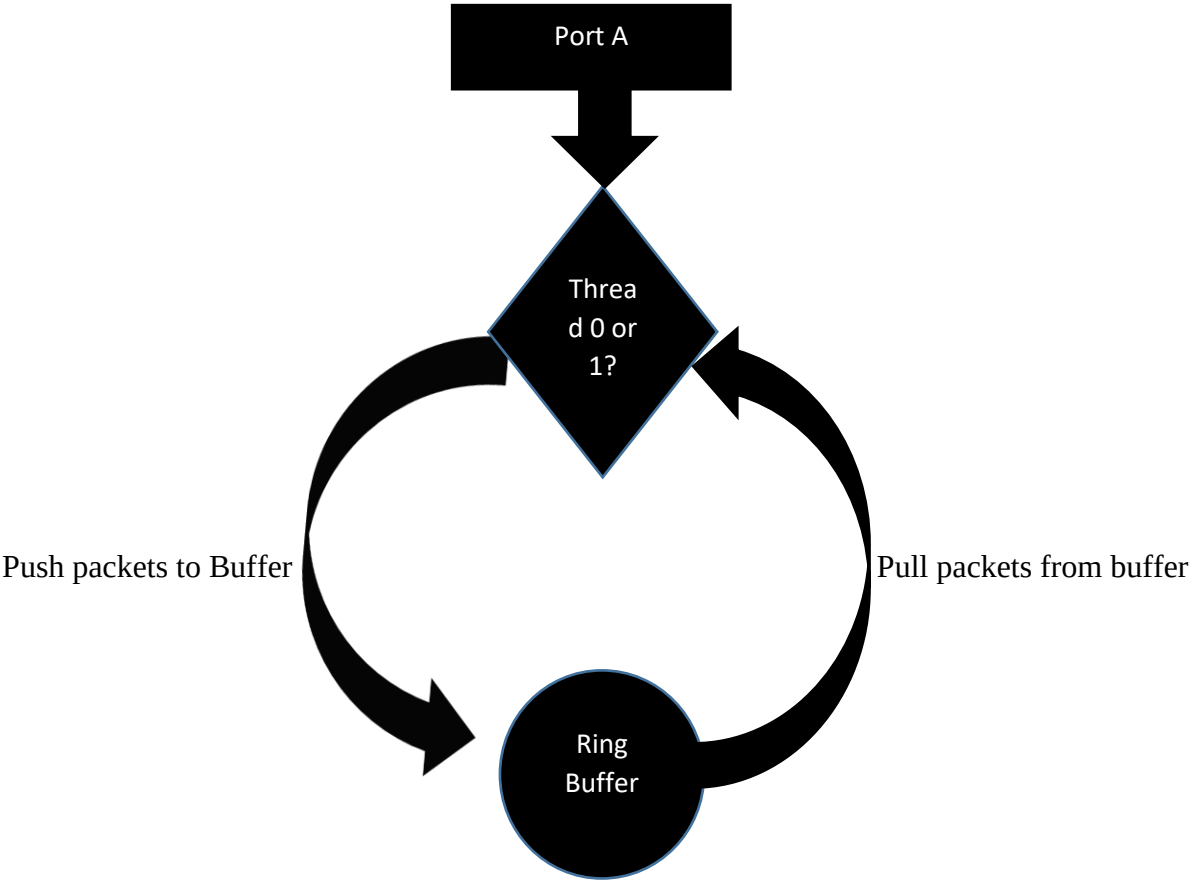
  uint8_t packet = rte_pktmbuf_mtod(pkts, uint8_t*)*

A macro that points to the start of the data in the mbuf. The returned pointer is cast to type t. Before using this function, the user must ensure that the first segment is large enough to accommodate its data.

Parameters passed are:
M → the packet mbuf
t → to type to cast the result into.

FLOWCHART:

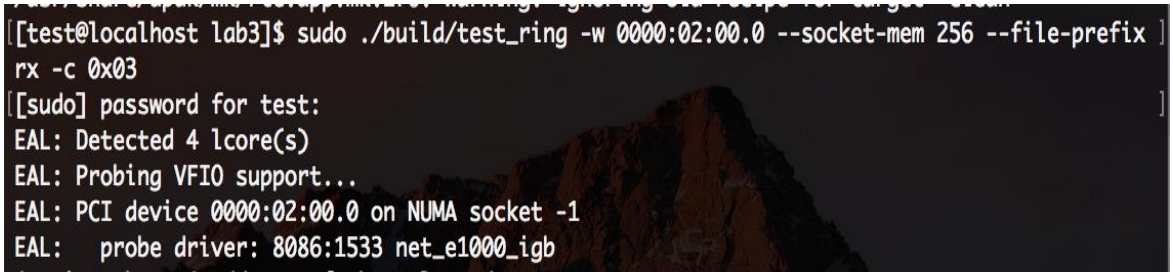
The flowchart below explains the concept of the code.



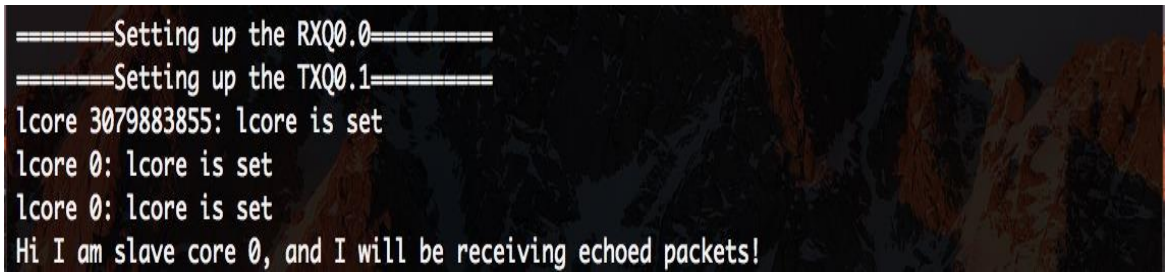
4. **Troubleshooting:**
   - I had difficulty in understanding the basic ring principle. And how the Tx and rx combine into code.
     I later saw some videos and papers to get a fair understanding.
   - There was a problem with the TX and RX enqueing process. After much trial and error method I could solve it.
   - The receiver was receiving only 1 packet.

5. **Results:**

- **Compiling the Tx code and running –**



- **Setting up the TX and RX queue –**



- **The packets being echoed -**

6. **Conclusion:**
   A DPDK code was developed which has both RX and TX functionality. The code has 2 threads. Thread 0 uses the RX function to receive the packets from port A and push it to ring buffer. Thread 1 pull packets from ring buffer and push it to the same port A.