# Exercise 8. Write on how a SNULL(Simple Network Utility for Loading Localities) works, need not execute.

The network interfaces fit in with the rest of the Linux kernel and provides examples in the form of a memory-based modularized network interface, which is called *snull*.

The interface uses the Ethernet hardware protocol and transmits IP packets. The knowledge you acquire from examining *snull* can be readily applied to protocols other than IP, and writing a non-Ethernet driver is different only in tiny details related to the actual network protocol.

1. **Entry and exit functions**

The Linux network device driver basically makes a fuss about the net_device structure. The entry function of the program is mainly to allocate and register the net_device structure. Correspondingly, the exit function needs to cancel the logout and release the net_device structure. This experiment will eventually use the ping command to test the sending and receiving of two network devices, so first define two pointers to net_device:

struct net_device *snull_devs[2];

1.1 **Entry function:**

```
static __init int snull_init_module(void)
{
  int result, i, ret = -ENOMEM;
  /* Allocate the devices */
        snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
        snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
        if (snull_devs[0] == NULL || snull_devs[1] == NULL)
                goto out;
  ret = -ENODEV;
        for (i = 0; i < 2;  i++)
                if ((result = register_netdev(snull_devs[i])))
                        printk("snull: error %i registering device \"%s\"\n",
                                        result, snull_devs[i]->name);
                else
                        ret = 0;
out:
        if (ret)
                snull_cleanup();
        return ret;
}
module_init(snull_init_module);
```

Alloc_netdev() function is used to allocate net_device structure:

```
struct net_device *alloc_netdev(int sizeof_priv, const char *name,
                        void (*setup)(struct net_device *));
```

The first parameter of the function is the size of the private data, the second parameter is the device name, and the third parameter setup points to a function that takes struct net_device* as a parameter and is used to initialize some members of net_device.

1.2 The private data of the program defines snull_priv (the usage of spin lock members is not repeated):

```
struct snull_priv {
        struct net_device_stats stats;
        int status;
        struct snull_packet *ppool;
        struct snull_packet *rx_queue;
        int rx_int_enabled;
        int tx_packetlen;
        u8 *tx_packetdata;
        struct sk_buff *skb;
        spinlock_t lock;
};
```

1.2.1 The snull_packet structure is used to save the sent and received data:

```
struct snull_packet {
        struct snull_packet *next;
        struct net_device *dev;
        int     datalen;
        u8 data[ETH_DATA_LEN];
};
```

The series of operation functions defined for struct snull_packet *pool are as follows:

```
int pool_size = 8;
module_param(pool_size, int, 0);
```

```
/*
 * Allocate 8 snull_packets and add them to a linked list
*/
void snull_setup_pool(struct net_device *dev)
{
        struct snull_priv *priv = netdev_priv(dev);
        int i;
        struct snull_packet *pkt;
        priv->ppool = NULL;
        for (i = 0; i < pool_size; i++) {
                pkt = kmalloc (sizeof (struct snull_packet), GFP_KERNEL);
                if (pkt == NULL) {
                        printk (KERN_NOTICE "Ran out of memory allocating packet pool\n");
                        return;
                }
                pkt->dev = dev;
                pkt->next = priv->ppool;
                priv->ppool = pkt;
        }
}
/*
 * Release the linked list composed of snull_packet
*/
void snull_teardown_pool(struct net_device *dev)
{
        struct snull_priv *priv = netdev_priv(dev);
        struct snull_packet *pkt;

        while ((pkt = priv->ppool)) {
                priv->ppool = pkt->next;
                kfree (pkt);
        }
}
/*
 * Take a snull packet from the linked list for sending
*/
struct snull_packet *snull_get_tx_buffer(struct net_device *dev)
{
        struct snull_priv *priv = netdev_priv(dev);
	unsigned long flags;
	struct snull_packet *pkt;

	spin_lock_irqsave(&priv->lock, flags);
	pkt = priv->ppool;
	priv->ppool = pkt->next;
	if (priv->ppool == NULL) {
		printk (KERN_INFO "Pool empty\n");
		netif_stop_queue(dev);
	}
	spin_unlock_irqrestore(&priv->lock, flags);
	return pkt;
}
/*
 * Put a snull packet back to the linked list
 */
void snull_release_buffer(struct snull_packet *pkt)
{
	unsigned long flags;
	struct snull_priv *priv = netdev_priv(pkt->dev);

	spin_lock_irqsave(&priv->lock, flags);
	pkt->next = priv->ppool;
	priv->ppool = pkt;
	spin_unlock_irqrestore(&priv->lock, flags);
	if (netif_queue_stopped(pkt->dev) && pkt->next == NULL)
		netif_wake_queue(pkt->dev);
}
```

1.2.2 rx_queue is the receiving queue of the device, the operation functions are as follows:

```
void snull_enqueue_buf(struct net_device *dev, struct snull_packet *pkt)
{
	unsigned long flags;
	struct snull_priv *priv = netdev_priv(dev);
	spin_lock_irqsave(&priv->lock, flags);
	pkt->next = priv->rx_queue;  /* FIXME - misorders packets */
	priv->rx_queue = pkt;
	spin_unlock_irqrestore(&priv->lock, flags);
}
```

```
struct snull_packet *snull_dequeue_buf(struct net_device *dev)
{
        struct snull_priv *priv = netdev_priv(dev);
        struct snull_packet *pkt;
        unsigned long flags;
        spin_lock_irqsave(&priv->lock, flags);
        pkt = priv->rx_queue;
        if (pkt != NULL)
                priv->rx_queue = pkt->next;
        spin_unlock_irqrestore(&priv->lock, flags);
        return pkt;
}
```

1.2.3 rx_int_enabled is used to enable or disable the receiving interrupt (the interrupt is not a real hardware interrupt, but is simulated by software), the operation function is as follows:

```
/*
* Enable and disable receive interrupts.
*/
static void snull_rx_ints(struct net_device *dev, int enable)
{
        struct snull_priv *priv = netdev_priv(dev);
        priv->rx_int_enabled = enable;
}
```

1.2.4 struct skb_buff socket buffer, this structure is critical, used to transfer data between different layers of the network protocol, the skb_buff structure will be further explained later;

1.3 The setup parameter of alloc_netdevice() points to a function that is used to initialize other members of net_device. The snull_init function is used in the program:

```
static int timeout = SNULL_TIMEOUT; //SNULL_TIMEOUT is defined as 5 in snull.h
module_param(timeout,int,0);
static const struct net_device_ops snull_dev_ops = {
   .ndo_open = snull_open,
   .ndo_stop = snull_release,
   .ndo_start_xmit = snull_tx,
   .ndo_do_ioctl = snull_ioctl,
   .ndo_get_stats = snull_stats,
    .ndo_tx_timeout = snull_tx_timeout,
};
static const struct header_ops snull_header_ops= {
    .create     = snull_header,
    .rebuild = snull_rebuild_header,
    .cache = NULL,
};
void snull_init(struct net_device *dev)
{
    struct snull_priv *priv = NULL;
    ether_setup(dev);

    dev->netdev_ops = &snull_dev_ops;
    dev->header_ops = &snull_header_ops;
    dev->watchdog_timeo = timeout;
        dev->flags |= IFF_NOARP; //Forbid ARP
    dev->features |= NETIF_F_NO_CSUM;
    priv = netdev_priv(dev);
    memset(priv, 0, sizeof(struct snull_priv));
    spin_lock_init(&((struct snull_priv *)priv)->lock);
            snull_rx_ints(dev, 1);                  /* enable receive interrupts */
            snull_setup_pool(dev);

    return;
}
```

snull_init() first initializes the members in the net_device structure: call the ether_setup() function to assign default values to many members in the net_device; Next, set the device method, because the kernel version is different, the device method is directly included in the net_device structure, which is required here Modified, from the code, you can see that the operations such as open and release are encapsulated in the netdev_ops, header_ops structure, and the specific implementation of the device methods such as open, send, and receive will be further explained later. The watchdog_timeo member sets the transmission timeout period, followed by some flags, indicating that ARP is forbidden and force packet verification.

After setting up the net_device member, use netdev_priv() to obtain the private data of the device and set the spin lock, receive enable and data buffer. netdev_priv() is dedicated to access private data of network devices:

```
void *netdev_priv(struct net_device *dev);
```

1.4 After allocating and initializing the net_device structure, we also need to register this device with register_netdev, and register the function prototype for unregistering the network device:

```
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
```

1.5 **Export function:**

```
static __exit void snull_cleanup(void)
{
  int i;

  for (i=0; i<2;i++)
  {
     if(snull_devs[i])
     {
        unregister_netdev(snull_devs[i]);
                        snull_teardown_pool(snull_devs[i]);
        free_netdev(snull_devs[i]);
     }
  }
  return;
}
module_exit(snull_cleanup);
```

The operation of the exit function is the opposite of the entry, mainly to unregister the device, release the data buffer and release the memory occupied by the device.

The explanation for the parameters used in 1.3:

Initializing the net_device member is mainly to implement some device operation methods, as follows:

```
static const struct net_device_ops snull_dev_ops = {
  .ndo_open = snull_open,
  .ndo_stop = snull_release,
  .ndo_start_xmit = snull_tx,
  .ndo_do_ioctl = snull_ioctl,
  .ndo_get_stats = snull_stats,
  .ndo_tx_timeout = snull_tx_timeout,
};
```

```
static const struct header_ops snull_header_ops= {
  .create    = snull_header,
  .rebuild = snull_rebuild_header,
  .cache = NULL,
};
```

<u>The following specifically describes how to implement these functions:</u>

1. snull_open: open the device

```
int snull_open(struct net_device *dev)
{
  memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
  if(dev == snull_devs[1])
     dev->dev_addr[ETH_ALEN-1] ++;
  netif_start_queue(dev);
  return 0;
}
```

The open function assigns snull_devs[0] and snull_devs[1] addresses, and starts the transmission queue of the interface. The transmission queue operations include:

```
netif_start_queue(struct net_device *dev); //Start transmission queue
netif_stop_queue(struct net_device *dev); //Stop transmission queue
netif_wake_queue(struct net_device *dev); //Restart the transmission queue
```

2. snull_release: Called when the interface is closed, it simply stops the transmission queue:

```
int snull_release(struct net_device *dev)
{
  netif_stop_queue(dev);
  return 0;
}
```

3. snull_tx: transmit data packets

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
        int len;
        char *data, shortpkt[ETH_ZLEN];
        struct snull_priv *priv = netdev_priv(dev);

        data = skb->data;
        len = skb->len;
        if (len < ETH_ZLEN) {
                memset(shortpkt, 0, ETH_ZLEN);
                memcpy(shortpkt, skb->data, skb->len);
                len = ETH_ZLEN;
                data = shortpkt;
        }
        dev->trans_start = jiffies; /* save the timestamp */
        /* Remember the skb, so we can free it at interrupt time */
        priv->skb = skb;
        /* actual deliver of data is device-specific, and not shown here */
        snull_hw_tx(data, len, dev);
        return 0; /* Our simple device can not fail */
}
```

skb is a pointer to sk_buff, get the data and data length to be sent from skb, and then call snull_hw_tx function to send, snull_hw_tx code is as follows:

```
void snull_hw_tx(char *buf, int len, struct net_device *dev)
{
    struct iphdr *ih;
    struct net_device *dest;
    struct snull_priv *priv = netdev_priv(dev);
    u32 *saddr, *daddr;
        struct snull_packet *tx_buffer;
    if (len < sizeof(struct ethhdr) + sizeof(struct iphdr))
    {
        printk("snull: Hmm... packet too short (%i octets)\n",len);
        return;
    }
    ih = (struct iphdr *)(buf+sizeof(struct ethhdr));
    saddr = &ih->saddr;
    daddr = &ih->daddr;

        if (dev == snull_devs[0])
        printk(KERN_INFO"before change: %08x:%05i --> %08x:%05i\n",
            ntohl(ih->saddr),ntohs(((struct tcphdr *)(ih+1))->source),
            ntohl(ih->daddr),ntohs(((struct tcphdr *)(ih+1))->dest));
    else
        printk(KERN_INFO"before change: %08x:%05i <-- %08x:%05i\n",
            ntohl(ih->daddr),ntohs(((struct tcphdr *)(ih+1))->dest),
            ntohl(ih->saddr),ntohs(((struct tcphdr *)(ih+1))->source));

((u8 *)saddr)[2] ^= 1;
((u8 *)daddr)[2] ^= 1;

ih->check = 0;
ih->check = ip_fast_csum((unsigned char *)ih,ih->ihl);
if (dev == snull_devs[0])
    printk(KERN_INFO"after change: %08x:%05i --> %08x:%05i\n",
            ntohl(ih->saddr),ntohs(((struct tcphdr *)(ih+1))->source),
            ntohl(ih->daddr),ntohs(((struct tcphdr *)(ih+1))->dest));
else
    printk(KERN_INFO"after change: %08x:%05i <-- %08x:%05i\n",
            ntohl(ih->daddr),ntohs(((struct tcphdr *)(ih+1))->dest),
            ntohl(ih->saddr),ntohs(((struct tcphdr *)(ih+1))->source));

        dest = snull_devs[dev == snull_devs[0] ? 1 : 0];
        priv = netdev_priv(dest);
        tx_buffer = snull_get_tx_buffer(dev);
        tx_buffer->datalen = len;
        memcpy(tx_buffer->data, buf, len);
        snull_enqueue_buf(dest, tx_buffer);
        if (priv->rx_int_enabled) {
                priv->status |= SNULL_RX_INTR;
                snull_interrupt(0, dest, NULL);
        }
        priv = netdev_priv(dev);
priv->status = SNULL_TX_INTR;
priv->tx_packetlen = len;
priv->tx_packetdata = buf;
if (lockup && ((priv->stats.tx_packets + 1) % lockup) == 0)
{
    netif_stop_queue(dev);
    printk(KERN_INFO"Simulate lockup at %ld, txp %ld\n", jiffies,(unsigned long) priv->stats.tx_packets);
}
else
    snull_interrupt(0, dev, NULL);
return;
```

```
}
```

This function modifies the source address and destination address of the data packet to produce the effect of pinging two network interfaces. For example, setting several IP addresses is as follows:

192.168.0.1：local0

192.168.0.2：remote0

192.168.1.2：local1

192.168.1.1：remote1

When sending ping -c 1 remote0, the source address is local0, the destination address is remote0, and the sending function modifies it to remote1 and local1. After local1 receives the data, it sends a response to remote1 The sending function changes the source and destination addresses to remote0 and local0, and finally local0 receives the data, causing the illusion of pinging. Here add the data packet to the receive queue, call snull_interrupt to imitate a hardware interrupt, and set the state of the two network interfaces. The snull_interrupt function is:

```
void snull_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
  int statusword;
  struct snull_priv *priv;
        struct snull_packet *pkt = NULL;

        struct net_device *dev = (struct net_device *)dev_id;

  if (!dev)
     return;
  /* Lock the device */
        priv = netdev_priv(dev);
        spin_lock(&priv->lock);

  statusword = priv->status;
  priv->status = 0;
  if (statusword & SNULL_RX_INTR)
  {
                if (dev_id == snull_devs[0])
                        printk(KERN_INFO"sn0 rx\n");
```

```
            if (dev_id == snull_devs[1])
                printk(KERN_INFO"sn1 rx\n");
    /* send it to snull_rx for handling */
            pkt = priv->rx_queue;
            if (pkt) {
                priv->rx_queue = pkt->next;
                snull_rx(dev, pkt);
            }
}
if (statusword & SNULL_TX_INTR)
{
            if (dev_id == snull_devs[0])
                printk(KERN_INFO"sn0 tx\n");
            if (dev_id == snull_devs[1])
                printk(KERN_INFO"sn1 tx\n");
    priv->stats.tx_packets++;
    priv->stats.tx_bytes += priv->tx_packetlen;
    dev_kfree_skb(priv->skb);
}
spin_unlock(&priv->lock);
        if (pkt)
            snull_release_buffer(pkt); /* Do this outside the lock! */

return;
}
```

If it is to receive an interrupt, call the snull_rx function: the main process is to allocate a sk_bull socket buffer, write data and set some members, and submit it to the upper layer software for processing with the netif_rx function. For a detailed description, refer to the description in the book.

```
void snull_rx(struct net_device *dev, struct snull_packet *pkt)
{
   struct sk_buff *skb;
   struct snull_priv *priv = netdev_priv(dev);
   skb = dev_alloc_skb(pkt->datalen + 2);
   if (!skb)
   {
       printk("snull rx: low on mem - packet dropped\n");
       priv->stats.rx_dropped++;
       return;
}
skb_reserve(skb, 2);
        memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
/* Write metadata, and then pass to the receive level */
        skb->dev = dev;
        skb->protocol = eth_type_trans(skb, dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
        priv->stats.rx_packets++;
        priv->stats.rx_bytes += pkt->datalen;
        netif_rx(skb);
return;
}
```

There are two ways to receive, based on interrupt or polling, here only retain the use of interrupt receiving, based on the description of the polling method reference book.

If it is a sending interrupt, release the socket buffer after setting the relevant state.

4. snull_ioctl: it will be called when using the ifconfig command, but here to simplify the procedure, it simply returns 0 and does nothing:

```
int snull_ioctl(struct net_device *dev, struct ifreq *rq, int cmd)
{
   return 0;
}
```

5. Snull_stats: returns the current status of the device

```
struct net_device_stats *snull_stats(struct net_device *dev)
{
   struct snull_priv *priv = netdev_priv(dev);
   return &priv->stats;
}
```

6. snull_tx_timeout: send timeout processing, just simulate sending interrupt again and restart the transmission queue

```
void snull_tx_timeout (struct net_device *dev)
{
   struct snull_priv *priv = netdev_priv(dev);
   printk(KERN_INFO"Transmit timeout at %ld, latency %ld\n", jiffies,jiffies - dev->trans_start);
```

```
    priv->status = SNULL_TX_INTR;
    snull_interrupt(0, dev, NULL);
    priv->stats.tx_errors++;
    netif_wake_queue(dev);
    return;
}
```

7. Snull_header: Called before hard_start_xmit, it is used to build a hardware header based on the retrieved source and target hardware addresses.

```
int snull_header(struct sk_buff *skb, struct net_device *dev,
                unsigned short type, const void *daddr, const void *saddr,
                unsigned int len)
{
    struct ethhdr *eth = (struct ethhdr *)skb_push(skb,ETH_HLEN);
    eth->h_proto = htons(type);
    memcpy(eth->h_source, saddr ? saddr : dev->dev_addr, dev->addr_len);
    memcpy(eth->h_dest, daddr ? daddr : dev->dev_addr, dev->addr_len);
    eth->h_dest[ETH_ALEN-1] ^= 0x01;
    return (dev->hard_header_len);
}
```

8. snull_rebuild_header: used to re-establish the hardware header after completing the ARP parsing before transmitting the data packet

```
int snull_rebuild_header(struct sk_buff *skb)
{
    struct ethhdr *eth = (struct ethhdr *) skb->data;
    struct net_device *dev = skb->dev;

    memcpy(eth->h_source, dev->dev_addr, dev->addr_len);
    memcpy(eth->h_dest, dev->dev_addr, dev->addr_len);
    eth->h_dest[ETH_ALEN-1] ^= 0x01;
    return 0;
}
```

After implementing these device methods, basically the driver is complete.