Exercise - 8

05.04.2021

CED17I017

FIROZ MOHAMMAD

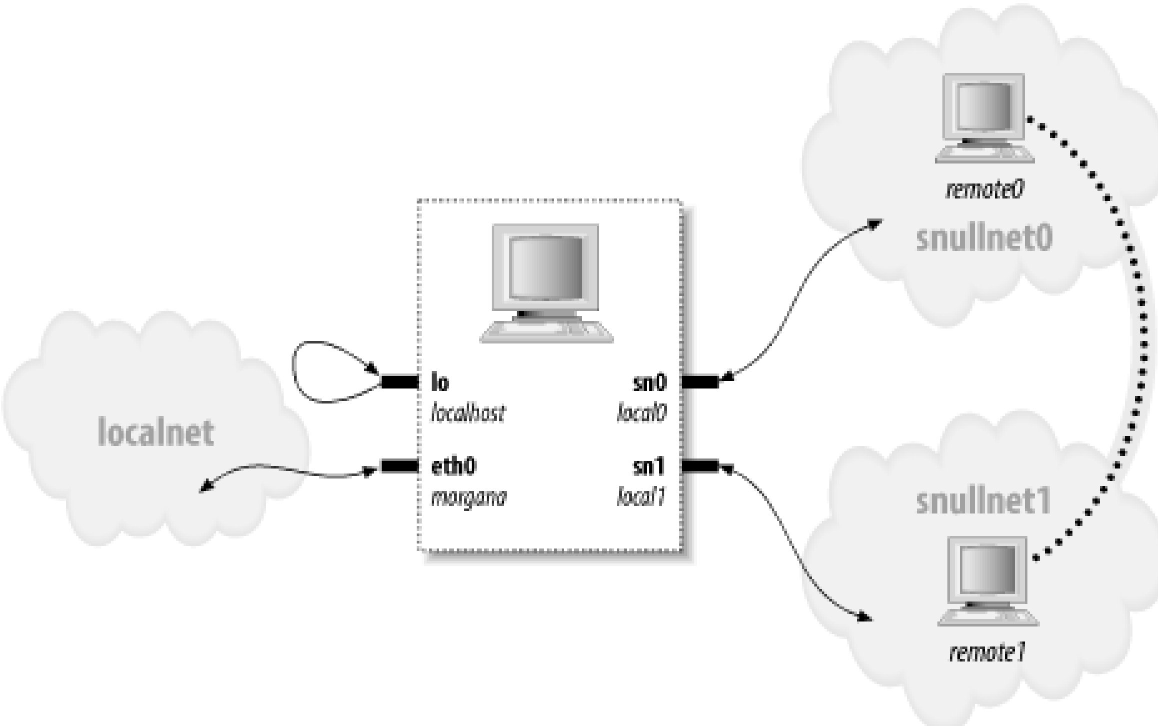
Write on how a SNULL(Simple Network Utility for Loading Localities) works, need not execute, just soft copy is sufficient.

SNULL – Simple Network Utility for Loading Localities,

- driver of the network device
- driver that does not talk to the ”actual” devices
- works like a loopback device

How snull driver work ?

- The *snull* module creates two interfaces.
- snullnet0 - the network associated with the sn0 interface,
- snullnet1 - the network associated with the sn1 interface,
- snullnet0 is the network that is connected to the sn0 interface. Similarly, snullnet1 is the network connected to sn1. The addresses of these networks should differ only in the least significant bit of the third octet. These networks must have 24-bit netmasks.
- local0 is the IP address assigned to the sn0 interface; it belongs to snullnet0. The address associated with sn1 is local1. local0 and local1 must differ in the least significant bit of their third octet and in the fourth octet.
- remote0 is a host in snullnet0, and its fourth octet is the same as that of local1. Any packet sent to remote0 reaches local1 after its network address has been modified by the interface code. The host remote1 belongs to snullnet1, and its fourth octet is the same as that of local0.



Here are possible values for the network numbers. Once you put these lines in */etc/networks*, you can call your networks by name. The values were chosen from the range of numbers reserved for private use.

snullnet0 192.168.0.0

snullnet1 192.168.1.0

he following are possible host numbers to put into */etc/hosts*:

192.168.0.1 local0

192.168.0.2 remote0

192.168.1.2 local1

192.168.1.1 remote1

Now , we can ping :

user: ~$ ping -c 2 remote0

64 bytes from 192.168.0.99: icmp_seq=0 ttl=64 time=1.6 ms

64 bytes from 192.168.0.99: icmp_seq=1 ttl=64 time=0.9 ms

2 packets transmitted, 2 packets received, 0% packet loss

user: ~$ ping -c 2 remote1

64 bytes from 192.168.1.88: icmp_seq=0 ttl=64 time=1.8 ms

64 bytes from 192.168.1.88: icmp_seq=1 ttl=64 time=0.9 ms

2 packets transmitted, 2 packets received, 0% packet loss

Let's deep dive to SNULL module working :

Entry and Exit function :

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];
```

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.

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;
};
```

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 *ppool 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);
}
```

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;
}
```

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;
}
```

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 in the book 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);
```

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);
```

Exit 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.