Exercise 8

Simple Network Utility for Loading Localities

SNULL is similar to loopback interface but sends the same packet back to the sender as though it received from a different outside network.

The steps of SNULL program is given below :

Headers :

Including header files required for the execution of SNULL program,like *<linux/slab.h>* for kmalloc,*<linux/error.h>* for error codes and other necessary headers.

Creating a private structure for each device :

This structure is private to each device. It is used to pass packets in and out, so there is a place for a packet.The structure also includes, among other things, an instance of struct net_device_stats, which is the standard place to hold interface statistics.

```
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];
};
```

Open Device :

The snull_open() function opens the interface. The interface is opened whenever *ifconfig* activates it. The *open* method should register any system resource it needs (I/O ports, IRQ, DMA, etc.), turn on the hardware, and perform any other setup your device requires.

```
int snull_open(struct net_device *dev)
{
    MOD_INC_USE_COUNT;
    /* request_region(), request_irq(), ....  (like fops->open) */
#if 0 && defined(LINUX_20)
    /*
     * We have no irq line, otherwise this assignment can be used to
     * grab a non-shared interrupt. To share interrupt lines use
     * the dev_id argument of request_irq. See snull_interrupt below.
     */
    irq2dev_map[dev->irq] = dev;
#endif
    /*
     * Assign the hardware address of the board: use "\0SNULx", where
     * x is 0 or 1. The first byte is '\0' to avoid being a multicast
     * address (the first byte of multicast addrs is odd).
     */
    memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
    dev->dev_addr[ETH_ALEN-1] += (dev - snull_devs); /* the number */

#ifndef LINUX_24
    dev->start = 1;
#endif
    netif_start_queue(dev);
    return 0;
}
```

Close/Release the Device :

It stops the interface. The interface is stopped when it is brought down. This function should reverse operations performed at open time.

```
int snull_release(struct net_device *dev)
{    /* release ports, irq and such -- like fops->close */
    netif_stop_queue(dev); /* can't transmit any more */
#ifndef LINUX_24
    dev->start = 0;
#endif
    MOD_DEC_USE_COUNT;
    /* if irq2dev_map was used (2.0 kernel), zero the entry here */
    return 0;
}
```

Configure the device :

To make configuration changes as shown below.

```
int snull_config(struct net_device *dev, struct ifmap *map)
{
    if (dev->flags & IFF_UP) /* can't act on a running interface */
        return -EBUSY;

    /* Don't allow changing the I/O address */
    if (map->base_addr != dev->base_addr) {
        printk(KERN_WARNING "snull: Can't change I/O address\n");
        return -EOPNOTSUPP;
    }

    /* Allow changing the IRQ */
    if (map->irq != dev->irq) {
        dev->irq = map->irq;
        /* request_irq() is delayed to open-time */
    }

    /* ignore other fields */
    return 0;
}
```

Receive Packets :

To retrieve, encapsulate and pass over to upper levels.

snull_rx receives a pointer to the data and the length of the packet.It sends the packet and some additional information to the upper layers of networking code.Also the packet reception is performed by netif_rx, which hands off the socket buffer to the upper layers.

```
void snull_rx(struct net_device *dev, int len, unsigned char *buf)
{
    struct sk_buff *skb;
    struct snull_priv *priv = (struct snull_priv *) dev->priv;
    /*
     * The packet has been retrieved from the transmission
     * medium. Build an skb around it, so upper layers can handle it
     */
    skb = dev_alloc_skb(len+2);
    if (!skb) {
        printk("snull rx: low on mem - packet dropped\n");
        priv->stats.rx_dropped++;
        return;
    }
    skb_reserve(skb, 2); /* align IP on 16B boundary */
    memcpy(skb_put(skb, len), buf, len);
    /* 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++;
#ifndef LINUX_20
    priv->stats.rx_bytes += len;
#endif
    netif_rx(skb);
    return;
}
```

Create IP/TCP headers – AND – third Octet is changed

As the packets are not modified and just only the address is modified,to receive the same packet from a different network the third octet is changed by the program before transmitting/sending the packet.This is the core of SNULL and is used to test the interface and to test if the packets are sent properly.

Send / Transmit Packet :

Whenever the kernel needs to transmit a data packet, it calls the driver's hard_start_transmit method to put the data on an outgoing queue. Each packet handled by the kernel is contained in a socket buffer structure (struct sk_buff).

The transmission function performs some sanity checks on the packet and transmits the data through the hardware-related function.

The return value from hard_start_transmit will be 0 on success; the driver has taken responsibility for the packet, should make its best effort to ensure that transmission succeeds, and must free the skb at the end. A nonzero return value indicates that the packet could not be transmitted and the kernel will retry later.

```
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 delivery of data is device-specific, and not shown here */
```

```
    snull_hw_tx(data, len, dev);
    return 0; /* Our simple device can not fail */
}
```

To deal with Transmission timeout :

When a transmission timeout happens, the driver must mark the error in the interface statistics and arrange for the device to be reset to a sane state so that new packets can be transmitted. When a timeout happens in snull, the driver calls snull_interrupt to fill in the "missing" interrupt and restarts the transmit queue with netif_wake_queue.

```
void snull_tx_timeout (struct net_device *dev)
{
    struct snull_priv *priv = (struct snull_priv *) dev->priv;

    PDEBUG("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;
}
```

Ioctl commands :

Ioctl commands are used for statistics and debugging

```
int snull_ioctl(struct net_device *dev, struct ifreq *rq, int cmd)
{
    PDEBUG("ioctl\n");
    return 0;
}
/*
 * Return statistics to the caller
 */
struct net_device_stats *snull_stats(struct net_device *dev)
{
    struct snull_priv *priv = (struct snull_priv *) dev->priv;
    return &priv->stats;
}
```

Rebuild headers :

Used to rebuild the hardware header after ARP resolution completes but before a packet is transmitted. The default function used by Ethernet devices uses the ARP support code to fill the packet with missing information.

```
#ifndef LINUX_20
```

```
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;   /* dest is us xor 1 */
    return 0;
}
#else /* LINUX_20 */
int snull_rebuild_header(void *buff, struct net_device *dev, unsigned long dst,
                    struct sk_buff *skb)
{
    struct ethhdr *eth = (struct ethhdr *)buff;

    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;   /* dest is us xor 1 */
    return 0;
}
#endif /* LINUX_20 */


int snull_header(struct sk_buff *skb, struct net_device *dev,
                unsigned short type, void *daddr, 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;   /* dest is us xor 1 */
    return (dev->hard_header_len);
}
```

Fix MTU :

This method changes the maximum transmission unit.

```
int snull_change_mtu(struct net_device *dev, int new_mtu)
{
    unsigned long flags;
    spinlock_t *lock = &((struct snull_priv *) dev->priv)->lock;

    /* check ranges */
    if ((new_mtu < 68) || (new_mtu > 1500))
        return -EINVAL;
    /*
     * Do anything you need, and the accept the value
     */
```

```
    spin_lock_irqsave(lock, flags);
    dev->mtu = new_mtu;
    spin_unlock_irqrestore(lock, flags);
    return 0; /* success */
}
```

Main program :

The core of this function (*snull_init*) is as follows:

```
    ether_setup(dev); /* assign some of the fields */

    dev->open            = snull_open;
    dev->stop            = snull_release;
    dev->set_config      = snull_config;
    dev->hard_start_xmit = snull_tx;
    dev->do_ioctl        = snull_ioctl;
    dev->get_stats       = snull_stats;
    dev->change_mtu      = snull_change_mtu;
    dev->rebuild_header  = snull_rebuild_header;
    dev->hard_header     = snull_header;
#ifdef HAVE_TX_TIMEOUT
    dev->tx_timeout     = snull_tx_timeout;
    dev->watchdog_timeo = timeout;
#endif
    /* keep the default flags, just add NOARP */
    dev->flags           |= IFF_NOARP;
#ifndef LINUX_20
    dev->hard_header_cache = NULL;      /* Disable caching */
```

It is a routine initialization of the net_device structure; it is mostly a matter of storing pointers to our various driver functions.The initialization code also sets a couple of fields (tx_timeout and watchdog_timeo) that relate to the handling of transmission timeouts. The following lines in snull_init allocate and initialize dev->priv:

```
priv = netdev_priv(dev);
memset(priv, 0, sizeof(struct snull_priv));
spin_lock_init(&priv->lock);
snull_rx_ints(dev, 1);      /* enable receive interrupts */
```

init the module :

```
int snull_init_module(void)
{

    int result, i, device_present = 0;
```

```
    snull_eth = eth; /* copy the cfg datum in the non-static place */

    if (!snull_eth) { /* call them "sn0" and "sn1" */
        strcpy(snull_devs[0].name, "sn0");
        strcpy(snull_devs[1].name, "sn1");
    } else { /* use automatic assignment */
#ifdef LINUX_24
        strcpy(snull_devs[0].name, "eth%d");
        strcpy(snull_devs[1].name, "eth%d");
#else
        snull_devs[0].name[0] = snull_devs[1].name[0] = ' ';
#endif
    }

    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 device_present++;
#ifndef SNULL_DEBUG
    EXPORT_NO_SYMBOLS;
#endif

    return device_present ? 0 : -ENODEV;
}
```

Cleanup the module :

The snull_cleanup function simply unregisters the interfaces, performs whatever internal cleanup is required, and releases the net_device structure back to the system.The call to *unregister_netdev* removes the interface from the system; *free_netdev* returns the net_device structure to the kernel. If a reference to that structure exists somewhere, it may continue to exist, but the driver need not care about that. Once the interface is unregistered,the kernel no longer calls its methods.

```
void snull_cleanup(void)
{
    int i;
    for (i=0; i<2;  i++) {
        kfree(snull_devs[i].priv);
        unregister_netdev(snull_devs + i);
    }
    return;
}
```