By:
Ritvik Raj Singh
CED17I047
5-04-2021

Exercise 8. Write on how a SNULL(Simple Network Utility for Loading Localities) works, need not execute, just soft copy is sufficient.

Step 1 : <Header Files>

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/moduleparam.h>

#include <linux/sched.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/errno.h>  /* error codes */
#include <linux/types.h>  /* size_t */
#include <linux/interrupt.h> /* mark_bh */

#include <linux/in.h>
#include <linux/netdevice.h>   /* struct device, and other headers */
#include <linux/etherdevice.h> /* eth_type_trans */
#include <linux/ip.h>          /* struct iphdr */
#include <linux/tcp.h>         /* struct tcphdr */
```

```
#include <linux/skbuff.h>
#include <linux/version.h>      /* LINUX_VERSION_CODE  */

#include "snull.h"

#include <linux/in6.h>
#include <asm/checksum.h>


MODULE_LICENSE("Dual BSD/GPL");
```

Step 2 :<private structures>

```
/*
 * A structure representing an in-flight packet.
 */
struct snull_packet {
        struct snull_packet *next;
        struct net_device *dev;
        int        datalen;
        u8 data[ETH_DATA_LEN];
};

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

/*
 * This structure is private to each device. It is used to pass
 * packets in and out, so there is place for a packet
 */
```

```
struct snull_priv {
        struct net_device_stats stats;
        int status;
        struct snull_packet *ppool;
        struct snull_packet *rx_queue;  /* List of incoming packets */
        int rx_int_enabled;
        int tx_packetlen;
        u8 *tx_packetdata;
        struct sk_buff *skb;
        spinlock_t lock;
        struct net_device *dev;
        struct napi_struct napi;
};
```

Step3 :<Snull_open-IRQ's>

```
/*
 * Open and close
 */

int snull_open(struct net_device *dev)
{
        /* request_region(), request_irq(), ....  (like fops->open) */

        /*
         * 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);
        if (dev == snull_devs[1])
                dev->dev_addr[ETH_ALEN-1]++; /* \0SNUL1 */
        if (use_napi) {
```

```
                struct snull_priv *priv = netdev_priv(dev);
                napi_enable(&priv->napi);
        }
        netif_start_queue(dev);
        return 0;
}
```

Step 4 : <Snull_release>

```
int snull_release(struct net_device *dev)
{
   /* release ports, irq and such -- like fops->close */

        netif_stop_queue(dev); /* can't transmit any more */
      if (use_napi) {
            struct snull_priv *priv = netdev_priv(dev);
            napi_disable(&priv->napi);
      }
        return 0;
}
```

Step 5 :<ifconfig operations>

```
/*
 * Configuration changes (passed on by ifconfig)
 */
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;
}
```

Step 6 : <receive packets , snull_interrupt>

```
/*
 * Receive a packet: retrieve, encapsulate and pass over to upper levels
 */
void snull_rx(struct net_device *dev, struct snull_packet *pkt)
{
        struct sk_buff *skb;
        struct snull_priv *priv = netdev_priv(dev);

        /*
         * The packet has been retrieved from the transmission
         * medium. Build an skb around it, so upper layers can handle it
         */
        skb = dev_alloc_skb(pkt->datalen + 2);
        if (!skb) {
                if (printk_ratelimit())
                        printk(KERN_NOTICE "snull rx: low on mem - packet dropped\n");
                priv->stats.rx_dropped++;
                goto out;
        }
```

```
        skb_reserve(skb, 2); /* align IP on 16B boundary */
        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);
  out:
        return;
}




/*
 * The typical interrupt entry point
 */
static void snull_regular_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        int statusword;
        struct snull_priv *priv;
        struct snull_packet *pkt = NULL;
        /*
         * As usual, check the "device" pointer to be sure it is
         * really interrupting.
         * Then assign "struct device *dev"
         */
        struct net_device *dev = (struct net_device *)dev_id;
        /* ... and check with hw if it's really ours */

        /* paranoid */
        if (!dev)
                return;

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

        /* retrieve statusword: real netdevices use I/O instructions */
        statusword = priv->status;
        priv->status = 0;
```

```
        if (statusword & SNULL_RX_INTR) {
                /* 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) {
                /* a transmission is over: free the skb */
                priv->stats.tx_packets++;
                priv->stats.tx_bytes += priv->tx_packetlen;
                dev_kfree_skb(priv->skb);
        }

        /* Unlock the device and we are done */
        spin_unlock(&priv->lock);
        if (pkt) snull_release_buffer(pkt); /* Do this outside the lock! */
        return;
}

/*
 * A NAPI interrupt handler.
 */
static void snull_napi_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        int statusword;
        struct snull_priv *priv;

        /*
         * As usual, check the "device" pointer for shared handlers.
         * Then assign "struct device *dev"
         */
        struct net_device *dev = (struct net_device *)dev_id;
        /* ... and check with hw if it's really ours */

        /* paranoid */
        if (!dev)
                return;

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

```
        /* retrieve statusword: real netdevices use I/O instructions */
        statusword = priv->status;
        priv->status = 0;
        if (statusword & SNULL_RX_INTR) {
                snull_rx_ints(dev, 0);  /* Disable further interrupts */
                napi_schedule(&priv->napi);
        }
        if (statusword & SNULL_TX_INTR) {
        /* a transmission is over: free the skb */
                priv->stats.tx_packets++;
                priv->stats.tx_bytes += priv->tx_packetlen;
                if(priv->skb) {
                        dev_kfree_skb(priv->skb);
                        priv->skb = 0;
                }
        }

        /* Unlock the device and we are done */
        spin_unlock(&priv->lock);
        return;
}
```

Step 7 : <snull_hw_tx,changing third octet>

```
/*
 * Transmit a packet (low level interface)
 */
static void snull_hw_tx(char *buf, int len, struct net_device *dev)
{
        /*
         * This function deals with hw details. This interface loops
         * back the packet to the other snull interface (if any).
         * In other words, this function implements the snull behaviour,
         * while all other procedures are rather device-independent
         */
        struct iphdr *ih;
        struct net_device *dest;
        struct snull_priv *priv;
        u32 *saddr, *daddr;
        struct snull_packet *tx_buffer;
```

```
/* I am paranoid. Ain't I? */
if (len < sizeof(struct ethhdr) + sizeof(struct iphdr)) {
        printk("snull: Hmm... packet too short (%i octets)\n",
                        len);
        return;
}

if (0) { /* enable this conditional to look at the data */
        int i;
        PDEBUG("len is %i\n" KERN_DEBUG "data:",len);
        for (i=14 ; i<len; i++)
                printk(" %02x",buf[i]&0xff);
        printk("\n");
}
/*
 * Ethhdr is 14 bytes, but the kernel arranges for iphdr
 * to be aligned (i.e., ethhdr is unaligned)
 */
ih = (struct iphdr *)(buf+sizeof(struct ethhdr));
saddr = &ih->saddr;
daddr = &ih->daddr;

((u8 *)saddr)[2] ^= 1; /* change the third octet (class C) */
((u8 *)daddr)[2] ^= 1;

ih->check = 0;         /* and rebuild the checksum (ip needs it) */
ih->check = ip_fast_csum((unsigned char *)ih,ih->ihl);

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

/*
 * Ok, now the packet is ready for transmission: first simulate a
 * receive interrupt on the twin device, then  a
 * transmission-done on the transmitting device
 */
dest = snull_devs[dev == snull_devs[0] ? 1 : 0];
priv = netdev_priv(dest);
```

```
        tx_buffer = snull_get_tx_buffer(dev);

        if(!tx_buffer) {
                PDEBUG("Out of tx buffer, len is %i\n",len);
                return;
        }

        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->tx_packetlen = len;
        priv->tx_packetdata = buf;
        priv->status |= SNULL_TX_INTR;
        if (lockup && ((priv->stats.tx_packets + 1) % lockup) == 0) {
        /* Simulate a dropped transmit interrupt */
                netif_stop_queue(dev);
                PDEBUG("Simulate lockup at %ld, txp %ld\n", jiffies,
                                (unsigned long) priv->stats.tx_packets);
        }
        else
                snull_interrupt(0, dev, NULL);
}
```

Step 8:<snull_tx , transmit>

```
/*
 * Transmit a packet (called by the kernel)
 */
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;
        }
        netif_trans_update(dev);

        /* 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 */
}
```

Step 9:<snull_ioctl , statistics,debug>

```
/*
 * Ioctl commands
 */
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 = netdev_priv(dev);
        return &priv->stats;
}
```

Step 10:<snull_rebuild_header >

```
/*
 * This function is called to fill up an eth header, since arp is not
 * available on the interface
 */
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;
}


int snull_header(struct sk_buff *skb, struct net_device *dev,
            unsigned short type, const void *daddr, const void *saddr,
            unsigned 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);
}
```

Step 11:<snull_change_mtu,largest packet 1500 bytes>

```
/*
 * The "change_mtu" method is usually not needed.
 * If you need it, it must be like this.
 */
int snull_change_mtu(struct net_device *dev, int new_mtu)
{
        unsigned long flags;
        struct snull_priv *priv = netdev_priv(dev);
        spinlock_t *lock = &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 */
}

static const struct header_ops snull_header_ops = {
       .create  = snull_header,
};
```

Step 12 : <main functions , calling all functions>

```
void snull_init(struct net_device *dev)
{
        struct snull_priv *priv;
#if 0
        /*
         * Make the usual checks: check_region(), probe irq, ...  -ENODEV
         * should be returned if no device found.  No resource should be
         * grabbed: this is done on open().
         */
#endif

        /*
         * Then, assign other fields in dev, using ether_setup() and some
         * hand assignments
         */
        ether_setup(dev); /* assign some of the fields */
        dev->watchdog_timeo = timeout;
        dev->netdev_ops = &snull_netdev_ops;
        dev->header_ops = &snull_header_ops;
        /* keep the default flags, just add NOARP */
        dev->flags           |= IFF_NOARP;
        dev->features        |= NETIF_F_HW_CSUM;

        /*
         * Then, initialize the priv field. This encloses the statistics
         * and a few private fields.
         */
        priv = netdev_priv(dev);
        memset(priv, 0, sizeof(struct snull_priv));
        if (use_napi) {
                netif_napi_add(dev, &priv->napi, snull_poll,2);
        }
        spin_lock_init(&priv->lock);
        priv->dev = dev;

        snull_rx_ints(dev, 1);          /* enable receive interrupts */
        snull_setup_pool(dev);
}
```

Step 13:

```
struct net_device *snull_devs[2];
```

Step 14 : <Init_Module>

```
int snull_init_module(void)
{
        int result, i, ret = -ENOMEM;

        snull_interrupt = use_napi ? snull_napi_interrupt : snull_regular_interrupt;

        /* Allocate the devices */
        snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
                        NET_NAME_UNKNOWN, snull_init);
        snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
                        NET_NAME_UNKNOWN, 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);
module_exit(snull_cleanup);
```

Step 15 :<free buffers,clean module>

```
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]); //will call netif_napi_del()
                }
        }
        return;
}
```

The above code snippets are taken from :
https://raw.githubusercontent.com/martinezjavier/ldd3/master/snull/snull.c