ECEN 749 LAB REPORT

Excercise #9:Linux Kernel: Built-in Modules

Tanmay Verma

12.4.2017

INTRODUCTION

The objective of this lab was to adapt the drivers(multiplier and ir_demod) into built-in modules in the linux image so that they can be loaded at boot time.

PROCEDURE

The following steps were carried out:

I. **Part A: Built-in Module : MULTIPLIER_DRIVER**

1. Copied the linux source code into lab_9 directory.
2. Created a directory in "driver" with name multiplier_driver and copied the driver source code there.
3. Edited the Makefiles and Kconfig entries
4. Configured the kernel to include multiplier driver at boot time
5. Compiled the Source code to generate the uImage.
6. This uImage alongwith other files from Lab_6 were copied to the SD_Card and zybo board was booted.
7. The multiplier messages were seen in the boot logs and mknod command was used to created an entry.
8. A successful operation was carried out with the multiplier devtest executable.

II. **Part B: Built-in Module : IR_DEMOD_DRIVER**

1. Created a new directory in "driver" with the name ir_demod_driver and copied the source code of driver from lab_8 into this directly.
2. Added and edited the Makefile and Kconfig for this device as well
3. Configured the Kernel to include ir_demod driver at boot.
4. Compiled the linux source and created the uImage.
5. Booted the zybo board with this new uImage and observed the module load messages at the boot time.
6. The image size was seen to have slightly increased than using only multiplier.

III. **Part C: Trimmed down Kernel Image**

1. Opened the menuconfig and deselected Networking Support, Device Drivers/Multimedia Support and Device Drivers/Sound Card support.
2. Compiled linux with this new configuration and observed the reduced uImage size.

RESULTS

Following were the uImages sizes that were obtained on the system:

1. Multiplier Driver: **3368.04 KB**
2. IR_Demod Driver: **3369.48 KB**
3. Trimmed Down Kernel: **2444.67 KB**

CONCLUSION

Linux image was configured with different built-in modules and the working operation of multiplier was verified.

Drivers Source Code

The Same driver source code from lab_6 and lab_8 was used. The init and exit functions alongwith virt_addr were renamed in ir_demod.

multiplier.c:

```
#include "multiplier.h"



/* This structure defines the function pointers to our functions for
 * opening, closing, reading and writing the device file.  There are
 * lots of other pointers in this structure which we are not using,
 * see the whole definition in linux/fs.h */
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
      .open = device_open,
      .release = device_release
};




static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
                           char *buffer,        /* buffer to fill with data */
                           size_t length,       /* length of the buffer     */
                           loff_t * offset)
{
    /*
     * Number of bytes actually written to the buffer
     */
    int bytes_read = 0;
    int i;

    for(i = 0; i < 3; i++) {
            *((int *)msg_bf_Ptr + i) = ioread32(virt_addr + i * 4);
            printk("The read data is Got = %d  read = %d", *((int *)msg_bf_Ptr + i),
ioread32(virt_addr + i * 4));
    }
    /*
             * Actually put the data into the buffer
         */
      cur_Ptr = msg_bf_Ptr;


      while (length) {
            /*
     * The buffer is in the user data segment, not the kernel
     * segment so "*" assignment won't work.  We have to use
     * put_user which copies data from the kernel data segment to
     * the user data segment.
         */
             put_user(*(cur_Ptr++), buffer++); /* one char at a time... */
     length--;
     bytes_read++;
    }
    return bytes_read;
}



/*
 * Called when a process tries to open the device file, like "cat
 * dev/my_chardev_mem".  Link to this function placed in file operations
 * structure for our device file.
 */
static int device_open(struct inode *inode, struct file *file)
```

```
{
    printk( KERN_INFO "The %s is opened", DEVICE_NAME);
    return 0;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
    printk( KERN_INFO "The %s is released", DEVICE_NAME);
    return 0;
}

static ssize_t device_write(struct file *file, const char __user * buffer, size_t length, loff_t * offset)
{
    int i;
    /* printk(KERN_INFO "device_write(%p,%s,%d)", file, buffer, (int)length); */
    /* get_user pulls message from userspace into kernel space */
    for (i = 0; i < length; i++)
            get_user(msg_bf_Ptr[i], buffer + i);
    /* write data1 to register 0 */
    printk(KERN_INFO "Writing %d to register 0 \n", *((int *)msg_bf_Ptr));
    iowrite32(*((int *)msg_bf_Ptr), virt_addr + 0 ); //base_address + offset
    /* write data2 to register 1 */
    printk(KERN_INFO "Writing %d to register 1 \n", *((int *)msg_bf_Ptr + 1));
    iowrite32(*((int *)msg_bf_Ptr + 1), virt_addr + 4); //base_address + offset

    /*
            * Again, return the number of input characters used
           */
      return i;
}

/*
 * This function is run upon module load. This is where you setup data structures
 * and reserves resources used by the modules.
 */
static int __init my_init(void)
{
    /* We need to allocate the memspace _BEFORE_ registering the device
            * to avoid any race conditions */
    msg_bf_Ptr = (char *)kmalloc(BUF_LEN*sizeof(char), GFP_KERNEL);

    /* Note: kmalloc can fail, even on a non-borked kernel, always exit
            * gracefully.  In the event of a failure pointer will be set to
            * NULL. */
```

```
  if (msg_bf_Ptr == NULL) {
                /* Failed to get memory, exit gracefully */
                printk(KERN_ALERT "Unable to allocate needed memory\n");
        return 10;              /* Defining error code of 10 for
                                        "Unable to allocate memory" */
  }
  cur_Ptr = msg_bf_Ptr;



 /* Linux kernel's version of printf */
 printk(KERN_INFO "Mapping virtual address to the multiplier\n");
 /* map virtual address to multiplier physical address */
 virt_addr = ioremap(PHY_ADDR, MEMSIZE);
 printk(KERN_INFO "VA %p gets mapped to PA %x", virt_addr, PHY_ADDR);
 /* write 7 to register 0 */



 /* This function call registers a device and returns a major number
  * associated with it.  Be wary, the device file could be accessed
  * as soon as you register it, make sure anything you need (ie
        * buffers ect) are setup _BEFORE_ you register the device.*/
 Major = register_chrdev(0, DEVICE_NAME, &fops);
 /* Negative values indicate a problem */
```

```
    if (Major < 0) {
              /* Make sure you release any other resources you've already
                  * grabbed if you get here so you don't leave the kernel in a
                  * broken state. */
              printk(KERN_ALERT "Registering char device failed with %d\n", Major);
              /* We won't need our memory so make sure to free it here... */
                  kfree(msg_bf_Ptr);
              return Major;
               }

    printk(KERN_INFO "Registered a device with dynamic Major number of %d\n", Major);

    printk(KERN_INFO "Create a device file for this device with this command:\n'mknod
/dev/%s c %d 0'.\n", DEVICE_NAME, Major);

    return 0;                  /* success */
}

/* This function is run just before the module removal. All resources must be released
here which were held by the module other wise prepare for reboot.*/
static void __exit my_exit(void)
{
    printk(KERN_ALERT "removing the driver... \n");
      /*
```

```
             * Unregister the device
     */
    unregister_chrdev(Major, DEVICE_NAME);

    kfree(msg_bf_Ptr);            /* free our memory (note the ordering
                                  here) */
    iounmap((void *)virt_addr);
}

/* ehese define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN749 Student (and others)");
MODULE_DESCRIPTION("Simple Hello World Module");

/*
 * Here we define whihc functions we want to use for initialization
 * and cleanup
 */
module_init(my_init);
module_exit(my_exit);
```

Ir_demod.c

```
/* Moved all prototypes and includes into the header file */
#include "ir_demod.h"

/* This structure defines the function pointers to our functions for
   opening, closing, reading and writing the device file.  There are
   lots of other pointers in this structure which we are not using,
   see the whole definition in linux/fs.h */
static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .release = device_release
};

/*
 * This function is called when the module is loaded and registers a
 * device for the driver to use.
 */
int my_init_1(void)
{

  init_waitqueue_head(&queue);    /* initialize the wait queue */

  /* Initialize the semaphor we will use to protect against multiple
            users opening the device  */
  sema_init(&sem, 1);
```

```
        printk(KERN_INFO "Mapping virtual address...\n");

        virt_addr_1 = ioremap(PHY_ADDR, MEMSIZE);           /* map virtual address to
multiplier physical address */

        /* print the virtual and physical addresses */

        printk(" %x Physical address of the ir_demod peripheral is mapped to  %p\n",
PHY_ADDR, virt_addr_1);

  Major = register_chrdev(0, DEVICE_NAME, &fops);

  if (Major < 0) {

        printk(KERN_ALERT "Registering char device failed with %d\n", Major);

                printk(KERN_ALERT "unmapping virtual address space...\n");

                iounmap((void*)virt_addr_1);

        return Major;

  }

  printk(KERN_INFO "Registered a device with dynamic Major number of %d\n", Major);

  printk(KERN_INFO "Create a device file for this device with this command:\n'mknod
/dev/%s c %d 0'.\n", DEVICE_NAME, Major);


  return 0;      /* success */

}


/*

 * This function is called when the module is unloaded, it releases

 * the device file.

 */

void my_cleanup_1(void)

{

  /*

   * Unregister the device

   */
  unregister_chrdev(Major, DEVICE_NAME);

}



/*
 * Called when a process tries to open the device file, like "cat
 * /dev/irq_test".  Link to this function placed in file operations
 * structure for our device file.
 */
static int device_open(struct inode *inode, struct file *file)
{
  int irq_ret;

  if (down_interruptible (&sem))
    return -ERESTARTSYS;

  /* We are only allowing one process to hold the device file open at
           a time. */
  if (Device_Open){
           up(&sem);
           return -EBUSY;
  }
  Device_Open++;

  /* OK we are now past the critical section, we can release the
           semaphore and all will be well */
  up(&sem);

  /* request a fast IRQ and set handler */
  irq_ret = request_irq(IRQ_NUM, irq_handler, 0 /*flags*/ , DEVICE_NAME, NULL);
  if (irq_ret < 0) {         /* handle errors */
        printk(KERN_ALERT "Registering IRQ failed with %d\n", irq_ret);
        return irq_ret;
  }

  try_module_get(THIS_MODULE);    /* increment the module use count
                                     (make sure this is accurate or you
                                     won't be able to remove the module
                                     later. */

  return 0;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
  Device_Open--;         /* We're now ready for our next caller */

  free_irq(IRQ_NUM, NULL);

  /*
```

```
   * Decrement the usage count, or else once you opened the file,
   * you'll never get get rid of the module.
   */
  module_put(THIS_MODULE);

  return 0;
}

/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,    /* see include/linux/fs.h   */
                           char *buffer,    /* buffer to fill with data */
                           size_t length,    /* length of the buffer        */
                           loff_t * offset)
{
  int bytes_read = 0;

  /*
   * Actually put the data into the buffer
   */
  while (length && (tail_ptr != head_ptr)) {

        /*
        * The buffer is in the user data segment, not the kernel
        * segment so "*" assignment won't work.  We have to use
        * put_user which copies data from the kernel data segment to
        * the user data segment.
        */
        if(tail_ptr == (msg+BUF_LEN)){
        tail_ptr = NULL;
        }
        if(tail_ptr == NULL){
        tail_ptr = msg;
        }
        put_user(*(tail_ptr++), buffer++); /* one char at a time… */

        length--;
        bytes_read++;
  }

  /*
   * Most read functions return the number of bytes put into the buffer
   */
  return bytes_read;
}

/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 * Next time we'll make this one do something interesting.
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
```

```
{

  /* not allowing writes for now, just printing a message in the
           kernel logs. */
  printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
  return -EINVAL;       /* Fail */
}

irqreturn_t irq_handler(int irq, void *dev_id) {
  static int counter = 0;    /* keep track of the number of
                                                interrupts handled */

    printk("IRQ Num %d called, interrupts processed %d times\n", irq, counter++);

  wake_up_interruptible(&queue);   /* Just wake up anything waiting
                                                for the device */
   if(head_ptr == NULL){
            head_ptr = msg;
    }
           *(head_ptr) = ioread8(virt_addr_1);
           head_ptr++;
           *(head_ptr) = ioread8(virt_addr_1+1);
           head_ptr++;
   iowrite8(1, virt_addr_1+10);
   if(head_ptr == (msg+BUF_LEN)){
    head_ptr = NULL;
   }
  return IRQ_HANDLED;

}




/* These define info that can be displayed by modinfo */

MODULE_LICENSE("GPL");

MODULE_AUTHOR("Paul V. Gratz (and others)");

MODULE_DESCRIPTION("Module which creates a character device and allows user interaction with it");


/* Here we define which functions we want to use for initialization

   and cleanup */

module_init(my_init_1);

module_exit(my_cleanup_1);
```

PICOCOM OUTPUT

Multiplier:

```
Mapping virtual address to the multiplier
VA 608e0000 gets mapped to PA 43c00000
Registered a device with dynamic Major number of 245
Create a device file for this device with this command:
'mknod /dev/multiplier c 245 0'.
TCP: cubic registered
NET: Registered protocol family 17
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
zynq_pm_ioremap: no compatible node found for 'xlnx,zynq-ddrc-a05'
zynq_pm_late_init: Unable to map DDRC IO memory.
Registering SWP/SWPB emulation handler
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
RAMDISK: gzip image found at block 0
mmc0: new high speed SDHC card at address aaaa
mmcblk0: mmc0:aaaa SL08G 7.40 GiB
 mmcblk0: p1
EXT2-fs (ram0): warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 212K (40627000 - 4065c000)
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
random: dropbear urandom read with 1 bits of entropy available
rcS Complete
zynq> mknod /dev/multiplier c 245 0
zynq> mount /dev/mmcblk0p1  /mnt/
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
zynq> ./devtest
The multiplier is opened
Writing 0 to register 0
Writing 0 to register 1
 written 0 * 0
The read data is Got = 0  read = 0The read data is Got = 0  read = 0The read data is Got = 0
read = 00 * 0 = 0
Results Correct !


Writing 0 to register 0
Writing 1 to register 1
 written 0 * 1
The read data is Got = 0  read = 0The read data is Got = 1  read = 1The read data is Got = 0
read = 00 * 1 = 0
Results Correct !


Writing 0 to register 0
Writing 2 to register 1
 written 0 * 2
The read data is Got = 0  read = 0The read data is Got = 2  read = 2The read data is Got = 0
read = 00 * 2 = 0
Results Correct !


Writing 0 to register 0
Writing 3 to register 1
 written 0 * 3
The read data is Got = 0  read = 0The read data is Got = 3  read = 3The read data is Got = 0
read = 00 * 3 = 0
Results Correct !
.
.
.
.
```

<u>IR Demod + Multiplier:</u>

```
Mapping virtual address to the multiplier
VA 608e0000 gets mapped to PA 43c00000
```

```
Registered a device with dynamic Major number of 245
Create a device file for this device with this command:
'mknod /dev/multiplier c 245 0'.
Mapping virtual address...
43c00000 Physical address of the ir_demod peripheral is mapped to  608e0000
Registered a device with dynamic Major number of 244
Create a device file for this device with this command:
'mknod /dev/ir_demod c 244 0'.
TCP: cubic registered
NET: Registered protocol family 17
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
mmc0: new high speed SDHC card at address aaaa
zynq_pm_ioremap: no compatible node found for 'xlnx,zynq-ddrc-a05'
zynq_pm_late_init: Unable to map DDRC IO memory.
Registering SWP/SWPB emulation handler
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
RAMDISK: gzip image found at block 0
mmcblk0: mmc0:aaaa SL08G 7.40 GiB
 mmcblk0: p1
EXT2-fs (ram0): warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 212K (40627000 - 4065c000)
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
random: dropbear urandom read with 1 bits of entropy available
rcS Complete
zynq>
```

QUESTIONS

[2 points.] Answers to the following questions:

(a) What are the advantage and disadvantages of loadable kernel modules and built-in modules?

Advantage: The user doesn't have to everytime load the module manually. It occurs at the boot time automatically and can be used by any application.

Disadvantage: Sometimes it loads modules which gets never used by the system. This causes unnecessary modules to be loaded and and increase in uImage size. They also increase the booting time of the machine