Lab 5: Introduction to Custom Kernel Modules

Abhinav Subramanian

ECEN 449-502

Due date: 2/25/2020

Introduction:

The aim of this lab was to learn how to integrate a custom module into the Linux kernel such that we can print messages to the console and access the custom Multiply IP. We also learned about how the Linux file system functions on the ZYBO.

Procedure:

- Boot Linux and experiment with mounting and unmounting the SD card file system from the ZYBO.
- Create the hello.c file and makefile to generate the custom kernel module that will print “Hello World!” to the console. Generate the hello.ko file and move it to the SD card.
- Mount the SD card on the ZYBO and insert the module into the kernel. If it worked correctly, you should see “Hello world!” being printed to the console.
- Now, write code for a custom kernel module to multiply two numbers using the custom Multiply IP we made in Lab 3. Change the makefile, and generate the .ko file as done before.
- Insert the multiply.ko module into the ZYBO kernel, and observe the output.

Results:

```
rcS Complete
zynq> mount /dev/mmcblk0p1 /mnt/
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt.
 Please run fsck.
zynq> ls
bin          lib          lost+found   proc         sys          var
dev          licenses     mnt          root         tmp
etc          linuxrc      opt          sbin         usr
zynq> cd mnt
zynq> ls
BOOT.bin             hello.ko             uImage
devicetree.dtb       multiply.ko          uramdisk.image.gz
zynq> insmod hello.ko
Hello Andrew!
zynq> insmod multiply.ko
Mapping virtual address...
Physical address: 43C00000
Physical address: 608C6000
Writing a 7 to register 0
Writing a 2 to register 1
Read 7 from register 0
Read 2 from register 1
Read 14 from register 2
zynq>
```

Conclusion:

In this lab, I learned how to use custom modules on Linux. It’s an incredibly useful feature that makes it easy to access custom peripherals. I also learned the importance of unmounting the SD card before I remove it from the ZYBO, as I forgot to do that once and the SD card got stuck on read only mode.

Postlab questions:

1. The ZYBO would have nothing to reboot from and would throw an error. Therefore, you’d likely have to turn off the zybo, which would kill picocom. So then, put in the SD, turn the board on, restart picocom, then reboot linux, and finally, remount the SD card. What a headache!
2. I found that the SD card was mounted at the path run/media/<my netID>/<sd card name> on the CentOS machine.

3. In the makefile, you’d have to change the .o file name to that of the new .c file. If the directory were changed to lab4 instead of lab5, nothing would happen, as this lab still uses the same kernel files and peripherals used in lab 4. In fact, I actually did this myself because it took forever to copy the lab4 folder into a new one for lab 5.

Appendix:

multiply.c

```
#include <linux/module.h>   // Needed by all modules

#include <linux/kernel.h>   // Needed for KERN_* and printk

#include <linux/init.h>     // Needed for __init and __exit macros

#include <asm/io.h>         // Needed for IO reads and writes

#include "xparameters.h"    // Needed for IO reads and writes

#include <linux/ioport.h>   // Used for io memory allocation


// From xparameters.h, physical address of multiplier

#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR

// Size of physical address range for multiply

#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR + 1


// virtual address pointing to multiplier

void* virt_addr;


/* This function is run upon module load. This is where you setup data

   structures and reserve resources used by the module */

static int __init my_init(void)

{

    // Linux kernel's version of printf

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

```
    // map virtual address to multiplier physical address
    // use ioremap, print the physical and virtual address
    virt_addr=ioremap(PHY_ADDR,12);
    printk(KERN_INFO "Physical address: %X \n",PHY_ADDR);
    printk(KERN_INFO "Physical address: %X \n",virt_addr);

    // write 7 to register 0
    printk(KERN_INFO "Writing a 7 to register 0\n");
    iowrite32(7, virt_addr + 0);     // base address + offset
    // write 2 to register 1
    printk(KERN_INFO "Writing a 2 to register 1\n");
    // use iowrite32
    iowrite32(2,virt_addr+4);

    printk("Read %d from register 0\n", ioread32(virt_addr+0));
    printk("Read %d from register 1\n", ioread32(virt_addr+4));
    printk("Read %d from register 2\n", ioread32(virt_addr+8));

    // A non 0 return means init_module failed; module can't be loaded
    return 0;
}

/* This function is run just prior to the module's removal from the system.
   You should release ALL resources used by your module here (otherwise be
   prepared for a reboot). */
static void __exit my_exit(void)
```

```
{
    printk(KERN_ALERT "unmapping virtual address space...\n");
    iounmap((void*)virt_addr);
}

// These define info that can be displayed by modinfo
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN449 Student (and others)");
MODULE_DESCRIPTION("Simple multiplier module");

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

Makefile for multiply:

```
obj-m += multiply.o
all:
	make -C ~/lab4/linux-3.14/ M=$(PWD) modules
clean:
	make -C ~/lab4/linux-3.14/ M=$(PWD) clean
```