Lab 5: Introduction to Kernel Modules on Zynq Linux System

Samuel Huang
ECEN 449

1. Introduction

The purpose of this lab is to build simple kernel modules that are loaded into the Linux kernel on the Zybo Z7-10 board. By creating a simple “Hello World!” and one that prints messages to the kernel’s message buffer and moderates kernel access to the multiplication peripheral. Ultimately the goal of this lab is to expand on earlier work with implementing a linux Kernel onto a FPGA, by adding modules that expand functionality and control created peripherals.

2. Procedure

The first part of the lab sets up the “Hello World!” module. By repeating the process in lab 4 to set up the Linux Kernel with Petalinux, a Linux Software Development Kit, allows us to add a hello world module. The module consists of C code which taps into the kernel and generates a Kernel alert consisting of the message “Hello World!”.

The second part of the lab builds off of the first part and creates a second module which reads and writes to the multiplication peripheral and prints the results to the kernel message buffer. This can be done by repeating the build process and adding the module before creating the Zynq Boot Image, or adding the module during the first set up with the hello world module. The multiply C module code gets the physical and virtual address of the multiplier peripheral to map the addresses and write to the two registers, and read the result into the kernel buffer.

With these two modules added, the modules are then built to create the Boot Image, which then gets loaded onto the SD card to program the ZYBO Z7-10 board. Once loaded the board, the modules are then checked for functionality.

3. Results

The lab builds off of the previous lab which creates a Linux operating system for a FPGA, and adds modules which perform tasks within the Linux Kernel. The first part of the lab creates a module which simply writes “Hello World!” into the kernel buffer. This illustrates the ability to add features to the linux kernel and start to control processes and peripherals. The second part demonstrates this ability to utilize the peripherals we added to the linux system, by mapping the location, inserting values and reading the results. The lab produces two different modules, one which writes text to the linux buffer, another which controls a peripheral.

4. Conclusion

The overall process for this lab was very similar to the previous lab. The most complicated part was making sure each step making the petalinux was correct. The addition of kernel modules in this process showed how C language programs can interact

with the kernel and start driving processes. The lab taught how to specifically connect to the linux kernel, and allow for future creation of modules that can generate meaningful processes. Ultimately the most complicated part of the lab was the time it took to build the project, and ensure that the created modules were properly working with the linux kernel.

5. Questions

If the board was reset before 2.f very few steps would be needed in 2.g since the whole linux kernel can be reloaded onto the board after the modules are integrated using Petalinux, and the modules can then be added into mnt. So the board must be reprogrammed and the mnt must be re accessed.

The mount point for the SD card on the CentOS machine is in the mnt directory, since this accesses the non volatile memory which is located in the SD card.

If we change the name of our hello.c file we would have to update the make file to reflect the new name so that it can properly build the program. If the wrong kernel directory was called, there would be problems since Lab 4 does not contain the modules we created in lab 5. Along with this the petalinux build would not be able to properly create the image boot, stopping us from creating the necessary files.

6. Appendix

Multiply.c:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <asm/io.h>
#include "xparameters.h"

/* physical address of the 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)

void *virt_addr; // Virtual address pointing to multiplier
```

```
/* This function is run upon module load toset up data
 * structures and reserve resources used by the module.
 */
static int __init my_init(void) {
   printk(KERN_INFO "Mapping virtual address...\n");

   /* map virtual address to multiplier physical address */
   virt_addr = ioremap(PHY_ADDR, MEMSIZE);
   if (!virt_addr) {
      printk(KERN_ALERT "Failed to map virtual address\n");
      return -ENOMEM;
   }

   /* 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");
   iowrite32(2, virt_addr + 4); // base address + offset

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

   return 0;
}

 * Resource release here
 */
```

```
static void __exit my_exit(void) {
   printk(KERN_ALERT "Unmapping virtual address space...\n");
   iounmap((void*)virt_addr);
}

/* modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN449 Student (and others)");
MODULE_DESCRIPTION("Simple multiplier module");

/*functions for initialization and cleanup */
module_init(my_init);
module_exit(my_exit);
```

Hello.c

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

/*
 * This function is run upon module load. This is where you set up data
 * structures and reserve resources used by the module.
 */
static int __init my_init(void) {
   /*
    * Linux kernel's version of printf
    */
   printk(KERN_INFO "Hello world!\n");
   // A non-zero 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 "Goodbye world!\n");
}

/*
 * These define info that can be displayed by modinfo
 */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN449 Student (and others)");
MODULE_DESCRIPTION("Simple Hello World Module");

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