ECEN 449: Microprocessor System Design
Department of Electrical and Computer Engineering
Texas A&M University

Prof. Sunil P. Khatri

Lab exercise created and tested by:
Abbas Fairouz, Ramu Endluri, He Zhou, Andrew Douglass and Sunil P. Khatri

Laboratory Exercise #5

Introduction to Kernel Modules on Zynq Linux System

Objective

The objective of this week's lab is to cross compile a simple 'Hello World!" kernel module on the CentOS 7 workstations, and load the module into the Linux kernel on the ZYBO Z7-10 board.

You will then create a kernel module, which prints messages to the kernel's message buffer. However, this kernel module will also moderate kernel access to the multiplication peripheral created in Lab 3.

System Overview

The hardware system you will use this week is identical to the one built in Lab 4. The Linux kernel, will be utilized to allow dynamic loading of kernel modules.

Procedure

1. If you have not already done so, make a copy of your 'lab4" directory and name it 'lab5". This directory should include the Linux kernel source. Boot Linux on the ZYBO Z7-10 board and mount the SD card so we can read and write files to it.

(a) From within the ZYBO Z7-10 Linux serial console (i.e. PICOCOM on the CentOS machine), run the following command:
>mount /dev/mmcb1k0p1 /mnt/

(b) The above command mounts the SD card to this location. There might be some warnings saying that the device is not properly unmounted.

(c) Test out the mount operation by running the following commands in PICOCOM:
>cd /mnt/
>ls -la

The first command changes the current directory to the SD card mount point. The next command lists the contents of the current directory. Do the files in the /mnt directory look familiar?

(d) Now test the write capabilities of the SD card by creating a directory within the /mnt directory. Run the following commands in PICOCOM:
>mkdir test
>ls -lae
Note the date stamp on the 'test' directory. Demonstrate your progress to the TA.

(e) We can now use the SD card to transfer files between the CentOS workstation where we will compile our kernel modules and the ZYBO Z7-10 board where we will run our kernel modules. Before removing the SD card from the ZYBO Z7-10 Linux system, we must un-mount the FAT partition on the SD card to avoid corrupting the file system. Run the following commands in PICOCOM:

>cd /
>umount /mnt

The first command changes the current directory to the root directory. The purpose of this command is to move out of the mount point prior to un-mounting. The second command un-mounts the FAT partition.

(f) Change the current directory to '/mnt' using 'cd' and examine the contents of the directory using 'ls'. What do you see?

2. At this point in lab, the ZYBO Z7-10 board has read/write access to a removable non-volatile file system. We will now cross-compile a simple kernel module on the CentOS machine and load it onto the SD card.

(a) Without turning off the power to the ZYBO Z7-10 board, remove the SD card and insert it into the CentOS machine. Examine the contents of the SD card. Note that the CentOS machine automatically mounts the SD card when inserted.

(b) Create a directory under 'lab5' called 'modules' and copy the text below into a file called 'hello.c'.

```
/*  hello.c - Hello World kernel module
 *
 *  Demonstrates module initialization, module release and printk.
 *
 *  (Adapted from various example modules including those found in the
 *  Linux Kernel Programming Guide, Linux Device Drivers book and
 *  FSM's device driver tutorial)
 */


#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 */


/* 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 "Hello world!\n");

    // 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 "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);
```

(c) Now we need to create a Makefile. In your 'modules' directory, create a file called 'Makefile' and fill it with the text below where <kernel_source_directory> is the root directory of the Linux kernel source code for lab 5. Make sure you use a tab before the make keywords.
Note: When compiling the kernel module, information about the original kernel configuration must be known because the kernel module will become an integral part of the kernel when installed. Thus, we must provide the Makefile with a path to the original kernel source. Do not forget to source settings64.sh

```
obj-m += hello.o

all:
	make -C <kernel_source_directory> M=$(PWD) modules

clean:
	make -C <kernel_source_directory> M=$(PWD) clean
```

(d) Then type the following command in the terminal window under the 'modules' directory:
>make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-

(e) If successful, you should see a hello.ko file appear in your 'modules' directory. Copy this file to the SD card.

(f) Properly remove the SD card from the CentOS machine and insert the card into the ZYBO Z7-10 board. Do not power cycle the the ZYBO Z7-10 board.

(g) From picocom, execute the command >mount /dev/mmcb1k0p1 /mnt/

(h) At this point, we have cross-compiled our Linux kernel module and provided access to it via the ZYBO Z7-10 board. It is now time to load the module into the Linux kernel on the ZYBO Z7-10 board. This is done with the following command:
>insmod hello.ko

(i) To see the output of the 'printk' statement that is called when the kernel module is loaded run the following command in the terminal:
>dmesg | tail

Demonstrate this output to the TA.

(j) We can see what modules are loaded by running 'lsmod' in the terminal window. Do this and ensure the 'hello' module is listed.

(k) Create a modules directory /lib/modules/`uname -r`using the following command. The` symbol can be found on the same key as the ~ symbol.

>mkdir -p /lib/modules/`uname -r`
uname is a kernel command used to get the name and information about the current kernel.

(l) To remove the module, run 'rmmod hello', and then run 'lsmod' to ensure that the module was removed. Also run 'dmesg' again to examine the output of the module during removal.

3. Compile a kernel module that reads and writes to the multiplication peripheral and prints the results to the kernel message buffer.

(a) On the CentOS machine, create a lab5b directory and copy the 'modules' directory from your lab5 directory to your lab5b directory.

(b) Navigate to the 'modules' directory and ensure that the 'Makefile' points to the Linux kernel directory.

(c) Create a new kernel module source file called 'multiply.c'.

(d) Copy the 'xparameters.h' and 'xparameters_ps.h' files in to modules directory. These files should be located in your lab5/lab4.sdk/FSBL_bsp folder.

(e) Copy the following skeleton code into '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 physical address of multiplier*/

/*from xparameters.h*/
#define PHY_ADDR  XPAR_MULTIPLY_0_S00_AXI_BASEADDR //physical address of multiplier
/*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. 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
    /*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
    printk("Read %d from register 0\n", ioread32(virt_addr+0));
    printk("Read %d from register 1\n", \\use ioread32);
    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);
```

(f) Some required function calls have been left out of the code above. Fill in the necessary code. Consult *Linux Device Drivers, 3rd Edition* for help.

Note: When running Linux on the ARM processor, your code is operating in virtual memory, and 'ioremap' provides the physical to virtual address translation required to read and write to hardware from within virtual memory. 'iounmap' reverses this mapping and is essentially the clean-up function for 'ioremap'.

(g) Add a 'printk' statement to your initialization routine that prints the physical and virtual addresses of your multiplication peripheral to the kernel message buffer.

(h) Edit your Makefile to compile 'multiply.c'. Compile your kernel module. Do not forget to source the 'settings64.sh' file.

(i) Load the 'multiply.ko' module onto the SD card and mount the card within the ZYBO Z7-10 Linux system using '/mnt' as the mount point. Consult step 1 if this is unclear.

(j) Use 'insmod' to load the 'multiply.ko' kernel module into the ZYBO Z7-10 Linux kernel. Demonstrate your progress to the TA.

Deliverables

1. [5 points.] Demo the working kernel module to the TA.

   Submit a lab report with the following items:

2. [5 points.] Correct format including an Introduction, Procedure, Results, and Conclusion.

3. [4 points.] The output of the terminal (picocom) for part 2 of lab.

4. [6 points.] Answers to the following questions:

   (a) If prior to step 2.f, we accidentally reset the ZYBO Z7-10 board, what additional steps would be needed in step 2.g?

   (b) What is the mount point for the SD card on the CentOS machine?
   Hint: Where does the SD card lie in the directory structure of the CentOS file system.

   (c) If we changed the name of our hello.c file, what would we have to change in the Makefile? Likewise, if in our Makefile, we specified the kernel directory from lab 4 rather than lab 5, what might be the consequences?