ECEN 749

Laboratory Exercise #5

Introduction to Kernel Modules on Zynq Linux System

Name: Pranav Anantharam

UIN: 734003886

Course: ECEN 749 Microprocessor System Design

Section: 603

Introduction:

The objective of the 5th laboratory exercise was to construct a basic "Hello World!" kernel module on the CentOS 7 workstations and integrate it into the Linux kernel on the ZYBO Z7-10 board. This exercise aimed to familiarize participants with kernel module creation and loading procedures. Additionally, it served to demonstrate practical kernel-level interactions and the management of peripheral access within the system, as illustrated by moderating kernel access to the multiplication peripheral introduced in a previous lab session. Moreover, we compiled a kernel module that reads and writes to the multiplication peripheral and prints the results to the kernel message buffer. Understanding kernel module development and manipulation is crucial for comprehending system-level programming and enhancing proficiency in operating system customization and optimization.

Procedure:

1) Boot Linux on the ZYBO Z7-10 board via the SD card.
2) Using the picocom serial console, run commands given in the manual to mount the SD card.
3) Test out the mount operation by running commands in picocom.
4) Next, we test the write capability of the SD card by creating a directory within the /mnt directory.
5) Run the umount /mnt command on the serial console to un-mount the FAT partition on the SD card and avoid corrupting the file system.
6) Remove the SD card from the Zybo Z7-10 board and insert it into the CentOS machine.
7) Insert a USB drive and open the PetaLinux project directory used for Lab 4.
8) Source the PetaLinux environment using the command given in the manual.
9) Create a module named ‘hello’ within the PetaLinux project directory.
10) Copy the code given in the lab manual into the hello.c driver file in the module directory.
11) Build the hello module and copy the compiled hello.ko file into the SD card.
12) Remove the SD card from the CentOS machine and insert the card into the Zybo Z7-10 board.
13) Within the serial console, run the mount command to mount the FAT partition of the SD card.
14) Next, load the module into the Linux kernel by running the insmod command.
15) To view the output of the printk statements, run the dmesg | tail command in the terminal.
16) Next, to build a kernel module that reads and writes to multiplication peripheral, create a module named ‘multiply’.
17) Copy the parameter.h and xparameters.h files to the multiply module directory.
18) Modify multiply.bb to include the header files.
19) Copy the skeleton code given in the lab manual into multiply.c and add the function calls to map virtual address to multiplier physical address, read and write into registers.
20) Build the multiply module and copy the compiled multiply.ko file into the SD card.
21) Insert the SD card into the Zybo Z7-10 board, mount the FAT partition and load the module using insmod command.
22) View the output using the dmesg | tail command.

(Source code and Output Screenshots given in Appendix Section)

Results:

Linux was booted up on the Zybo Z7-10 board. We were able to build and compile kernel modules on the CentOS workstations and load the kernel modules onto the Zybo Z7-10 board. A‘hello world’ kernel module was built, along with a kernel module to interact with the multiplier peripheral and print results to the kernel message buffer. Outputs were obtained and displayed on a serial console using picocom application. The outputs were demonstrated to the TA as well.

The following changes were made in the source code (multiply.c):

- Added ioremap function call to provide physical to virtual address space translation required to read and write data to hardware from within virtual memory.
- Added iowrite32 and ioread32 function calls to write to / read from registers that can hold 32-bit integers.
- Added printk statements to print results to the kernel message buffer.

(Source Code and Output Screenshots given in Appendix Section)

Conclusion:

In conclusion, the fifth laboratory exercise provided valuable insights into kernel module development and its integration into the Linux kernel ecosystem. We gained hands-on experience crucial for understanding system-level programming intricacies, including constructing a "Hello World!" kernel module and compiling kernel modules to interact with peripherals, such as the multiplier peripheral, and print results to the kernel message buffer. These skills enhanced proficiency in operating system customization and optimization, establishing a solid foundation for addressing more advanced system-level programming challenges.

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?

Answer:
In the event of accidentally resetting the Zybo board before reaching step 2.f., we would need to reset the Zybo board to load the Linux kernel again from the SD Card and mount the SD Card in the Linux filesystem. Additionally, acquiring superuser privileges using the “sudo su -” command will be imperative for executing any subsequent actions requiring elevated privileges.

(b) What is the mount point for the SD card on the CentOS machine?

Answer:

The mount point for the SD card on the CentOS machine is located at:

/run/media/<Net-ID>/<SD_card_name>

For example, the mount point for my SD card on my CentOS machine was:

/run/media/pranav_anantharam/ECEN_749

(ECEN_749 is the name of the partition created on the SD card)

(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?

Answer:

We need to update the first line of the Makefile as follows -

```
obj-m += multiply.o
```

Specifying the same kernel directory from Lab 4 in the Makefile would have no consequences since we have used the same Linux folder copied from Lab 4 to Lab 5.

Appendix:

Hello World Kernel Module Console Output Screenshots:

```
linux_boot:/mnt# ls
BOOT.BIN  boot.scr  hello.ko  image.ub  test
linux_boot:/mnt# cd ../
linux_boot:/# ls
bin    boot   dev    etc    home   init   lib    media  mnt    proc   root   run    sbin   sys    tmp    usr    var
linux_boot:/# insmod /mnt/hello.ko
hello: loading out-of-tree module taints kernel.
Hello world!
linux_boot:/# dmesg | tail
mmcblk0: mmc0:aaaa SS08G 7.40 GiB
 mmcblk0: p1
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
mmc0: card aaaa removed
mmc0: new high speed SDHC card at address aaaa
mmcblk0: mmc0:aaaa SS08G 7.40 GiB
 mmcblk0: p1
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
hello: loading out-of-tree module taints kernel.
Hello world!
linux_boot:/# lsmod
    Tainted: G
hello 16384 0 - Live 0xbf005000 (O)
uio_pdrv_genirq 16384 0 - Live 0xbf000000
linux_boot:/#
```

Multiply Kernel Module Console Output Screenshots:

```
ls
bin    boot   dev    etc    home   init   lib    media  mnt    proc   root   run    sbin   sys    tmp    usr    var
linux_boot:/# mount /dev/mmcblk0p1 /mnt/
linux_boot:/# insmod /mnt/multiply.ko
Mapping virtual address...
Physical Address = 34e60ced
Virtual Address = 53e30bab
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
linux_boot:/# dmesg | tail
 mmcblk0: p1
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
Mapping virtual address...
Physical Address = 34e60ced
Virtual Address = 53e30bab
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
linux_boot:/#
```

Hello World Kernel Module Source Code:

```c
/* hello.c - Hello World kernel module
 *
 * Demonstrates module initialization, module release and printk.
 */
#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 information that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN749 Student");
MODULE_DESCRIPTION("Simple Hello World Module");

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

Multiply Kernel Module Source Code:

```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
```

```c
#include <linux/ioport.h>   // IO memory allocation

// 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

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

    // map virtual address to multiplier physical address
    // use ioremap
    virt_addr = ioremap(PHY_ADDR, MEMSIZE);

    // print the physical and virtual address
    printk(KERN_INFO "Physical Address = %p \n", PHY_ADDR);
    printt(KERN_INFO "Virtual Address = %p \n", virt_addr);

    // Write register 7 to register 0
    printk(KERN_INFO "Writing a 7 to register 0 \n");
    // use iowrite32
    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);
```

```
}

/* Display information that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN749 Student");
MODULE_DESCRIPTION("Simple multiplier module");

/* Functions to use for initialization and cleanup */
module_init(my_init);
module_exit(my_exit);
```