Lab 2: Kernel Module

Yu-Lun Huang

2011-07-20

1 Objective

- Learning kernel programming skills.
- Be familiar with `insmod()`, `lsmod()`, `rmmod()`, etc.

2 Prerequisite

- Read man pages of the above system calls.

3 What is a Kernel Module?

Modules are pieces of code that can be loaded and unloaded into the kernel on demand. Modules extend the functionality of the kernel without rebooting the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

A program usually begins with a `main()` function, executes a bunch of instructions and terminates upon completion of those instructions. Kernel modules work a bit differently. A module always begin with the function you specify with `module_init()` call. This is the entry function for modules; it tells the kernel what functionality the module provides and sets up the kernel to run the module's functions when they are needed. Once it does this, the entry function returns and the module does nothing until the kernel wants to do something with the code that the module provides. All modules end by calling the function you specify with the `module_exit()` call. This is the exit function for modules; it undoes whatever entry function did. It unregisters the functionality that the entry function registered. The following sample code is the simplest module, hello world.

```
1  /*
2   * hello.c -- the simplest module
3   */
4
5  #include <linux/init.h>    /*  Needed for the macros */
6  #include <linux/kernel.h>  /* Needed for KERN_ALERT */
7  #include <linux/module.h>  /* Needed by all modules */
8
9  int hello_init(void)
10 {
11     printk(KERN_ALERT "Hello world.\n");
12
```

```
13      // A non 0 return means init_module failed; module can't be loaded.
14      return 0;
15  }
16
17  void hello_exit(void)
18  {
19      printk(KERN_ALERT "Goodbye world.\n");
20  }
21
22  module_init(hello_init);
23  module_exit(hello_exit);
24
25  MODULE_LICENSE("GPL");
```

Kernel module also prints messages in a different way. It uses `printk()` function to write all levels of messages to `/proc/kmsg`, rather than calls the standard C library (libc) function `printf()`. That's because modules are object files whose symbols get resolved upon calling `insmod()`. The definition for the symbols comes from the kernel itself, so `printk()` is the only external functions you can use provided by the kernel.

4 Compiling a Kernel Module

Kernel modules need to be compiled a bit differently from regular user-space applications. In most case, we write a "Makefile" for easy compilation. Let's look at a simple Makefile for compiling a module named `hello.c`:

```
1  obj-m += hello.o
2
3  all:
4      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6  clean:
7      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Use `make` command to start the compilation.

```
SHELL> make
```

Now, you can load your modules with `insmod()`.

```
SHELL> insmod hello.ko
```

You can see your module loaded into the kernel by:

```
SHELL> lsmod
```

Or, you can remove your module by:

```
SHELL> rmmod hello
```

The `printk()` function writes all levels of messages to `/proc/kmsg` . You can check the messages by executing "`dmesg`".

```
SHELL> dmesg
```

5 LED Control on PXA270

There is an 8-bit LED lamps on the motherboard of PXA270, numbered frome D9(1) to D16(8) We use the `creator-pxa270-lcd.ko` module to control the LED lamps, it is also used to drive the LCD, 7-segement LED, KeyPAD, and DIP Switch.

5.1 Compile and load modules

Rebuild your kernel and rootfile system to support `creator-pxa270-lcd.ko` module:

- Get the source code of `creator-pxa270-lcd.ko`:
  Download `Creator_PXA270_LCD_Device_Driver.src.tar.gz` from E3 website and decompressed it to your kernel source.

```
SHELL> cd ~
SHELL> tar xzvf Creator_PXA270_LCD_Device_Driver.src.tar.gz
```

- Configure kernel source:

```
SHELL> cd ~/microtime/linux
SHELL> make mrproper
SHELL> make menuconfig
```

  - In the window of "Linux Kernel Configuration", select "Load an Alternate Configuration from File" and load the configuration file `arch/arm/configs/creator_pxa270_defconfig`.
  - Select "Device Drivers" → "Character devices" and mark "Creator-pxa270 LCD" as `[M]`.
  - Save and exit kernel configuration.

- Make Image:
  Compile Linux kernel and `creator-pxa270-lcd.ko` module.

```
SHELL> make clean
SHELL> make
```

  The `creator-pxa270-lcd.ko` module will be placed at `microtime/linux/drivers/char/`.

- Make new root filesystem:
  Copy `creator-pxa270-lcd.ko` module into root filesystem.

```
SHELL> cp ~/microtime/linux/drivers/char/creator-pxa270-lcd.ko
~/microtime/rootfs/lib/modules/2.6.15.3/kernel/drivers/char/
```

  Then, rebuild and flash root filesystem.

- Load modules

  Type the following command to load the `creator-pxa270-lcd.ko` on PXA270.

```
> insmod lib/modules/2.6.15.3/kernel/drivers/char/creator-pxa270-lcd.ko
```

5.2 Control LED

- LED programming guide

Header file:

```
#include "asm-arm/arch-pxa/lib/creator_pxa270_lcd.h"
```

Commands:

```
LED_IOCTL_SET          // set the specified LED (D9 - D16)
LED_IOCTL_CLEAR        // clear the specified LED (D9 - D16)
```

Values:

```
LED_ALL_ON          0xFF
LED_ALL_OFF         0x00
LED_D9_INDEX        1
LED_D10_INDEX       2
LED_D11_INDEX       3
LED_D12_INDEX       4
LED_D13_INDEX       5
LED_D14_INDEX       6
LED_D15_INDEX       7
LED_D16_INDEX       8
```

Sample code:

```
/*
 * led.c -- the sample code for controlling LEDs on Creator.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include "asm-arm/arch-pxa/lib/creator_pxa270_lcd.h"

int main(int argc, char *argv[])
{
    int fd;          /* file descriptor for /dev/lcd */
    int retval;

    unsigned short data;

    /* Open device /dev/lcd */
    if((fd = open("/dev/lcd", O_RDWR)) < 0)
    {
        printf("Open /dev/lcd faild.\n");
        exit(-1);
    }

```

```
26      /* Turn on all LED lamps */
27      data = LED_ALL_ON;
28      ioctl(fd, LED_IOCTL_SET, &data);
29      printf("Turn on all LED lamps\n");
30      sleep(3);
31
32      /* Turn off all LED lamps */
33      data = LED_ALL_OFF;
34      ioctl(fd, LED_IOCTL_SET, &data);
35      printf("Turn off all LED lamps\n");
36      sleep(3);
37
38      /* Turn on D9 */
39      data = LED_D9_INDEX;
40      ioctl(fd, LED_IOCTL_BIT_SET, &data);
41      printf("Turn on D9 \n");
42      sleep(3);
43
44      /* Turn off D9 */
45      data = LED_D9_INDEX;
46      ioctl(fd, LED_IOCTL_BIT_CLEAR, &data);
47      printf("Turn off D9 \n");
48      sleep(3);
49
50      /* Close fd */
51      close(fd);
52
53      return 0;
54  }
```

Add the header search path when compile `led.c`.

```
SHELL> arm-unknown-linux-gnu-gcc -o hello hello.c
-L /opt/arm-unknown-linux-gnu/arm-unknown-linux-gnu/lib/
-I /opt/arm-unknown-linux-gnu/arm-unknown-linux-gnu/include/
-I /home/lab616/microtime/linux/include/
```

Lab 2 Assignments

1. Rewrite the sample codes to print your student id and name to /proc/kmsg on your PC.
2. Rewrite the sample codes to print your student id and name to /proc/kmsg on PXA270.
3. Write a simple program to show the number input on LED. The program reads number from console, converts the number into binary code, and shows the binary code on LED.