OS Primer

By Rich West, Boston University.

Part 1: Invoking Kernel Services using Loadable Modules in Linux

The aim of this part of the primer assignment is for you to learn how to write a simple kernel loadable module. In the past (i.e., when Linux 2.4.x kernels were still the latest and greatest) this primer included hints on how to write your own system calls. As of the 2.6.x Linux kernels, steps were taken to make it difficult for developers to write their own system calls. This is primarily due to the fact that the system call table (identified by the "sys_call_table" symbol) is no longer exported for use in modules. Notwithstanding, practice at writing a simple kernel module, along with code to activate functionality in that module is useful when developing kernel projects (and a module can be used as the basis for a device driver you may one day write!).

A helpful reference for this work is "Linux Device Drivers", by A. Rubini O'Reilly.

Before You Start

You will need to setup a virtual disk for use with a PC emulator, such as VirtualBox, QEMU, BOCHS, VMplayer or similar. Further information will be provided on this.

- ### Writing a Simple Module

The first part of this assignment requires you to implement a simple kernel loadable module that simply prints two strings S1 and S2 to the console. S1 is a string such as "Loading Module..." that is output to the console when your module is first loaded into the kernel, while S2 is is a string such as "Unloading module..." that is output when the module is removed from the kernel.

Up to Linux version 2.4.x it was acceptable to use init_module() and cleanup_module() as the names of the initialization and cleanup functions in your module. The init_module() function was invoked when a module was first loaded into the kernel address space using the `insmod` (or `modprobe`) command. Likewise, cleanup_module() was called when a module was removed from the kernel using `rmmod modulename`.

As of Linux 2.6.x, two new functions were introduced: module_init() and module_exit() in place of the now deprecated init_/cleanup_module() functions. Both of these functions take a single argument, which is the name of the initialization and cleanup functions that you write in your module.

Further information can be found in "The Linux Kernel Module Programming Guide":

- http://tldp.org/LDP/lkmpg/2.6/html/index.html
- NOTE: the above link is to the old kernel version 2.6.x but this is adequate for educational purposes.

Other useful links include:

- The somewhat old Kernel Build HOWTO,
- The Linux Cross Reference (Originally, lxr.linux.no, then http://lxr.free-electrons.com, and now: https://elixir.bootlin.com/linux/latest/source), which is useful for searching around the kernel source tree for various kernel versions

To begin, you should try to write the following (simple) kernel module in a file called "test_module.c":

```
/*
  *     test module.
  */

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

MODULE_LICENSE("GPL");

static int __init initialization_routine(void) {

   printk ("Hello, world!\n");

   return 0;
}

static void __exit cleanup_routine(void) {

   printk ("Unloading module!\n");
}

module_init(initialization_routine);
module_exit(cleanup_routine);
```

This file can be placed in its own directory along with a simple Makefile (assuming the kernel is 2.6.x), having just the following line:

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

Before you can use your module you will need to build the source code against the kernel source tree. If the kernel source tree is in the directory such as /usr/src/linux-`uname -r`, you can issue the following make command to generate a module called "test_module.ko":

```
make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
```

NOTE: If the above line does not work, try:

```
make -C /lib/modules/`uname -r`/build M=$PWD modules
```

Once you have successfully created your kernel module, you can load it using a command such as `insmod test_module.ko`. If everything works, you'll be able to use the command `dmesg` to see any kernel messages printed by your module functions. To remove your module, issue the shell command `rmmod test_module` (without the .ko ending).

After completing Part 1, you should try the examples found at:

- http://tldp.org/LDP/lkmpg/2.6/html/index.html (See "List of Examples" at the bottom of the page)

You should also practice configuring and building a new kernel from source, using the information provided in the pointers above.

To make sure you know how to tackle this basic primer assignment before we write more complex kernel code, you should demonstrate your code to the TF in one of the lab sessions.

Note that, in general, you can use any symbols (i.e., functions, global variables etc) that are exportable to modules. In older kernel versions, these symbols were visible via kernel/xxxx_ksyms.c (where xxxx refers to a specific architecture such as "i386"). It used to be the case that you could try something like `cat /proc/ksyms | more` to see the available symbols. Nowadays, you can see all symbol names via /proc/kallsyms. Additionally, you can use the printk() kernel function to print your messages to the console. If they do not appear on the console, this may be due to an insufficiently high enough priority for the message to appear on the console (or logging of kernel messages has been disabled entirely). You may need to edit /etc/syslog.conf and insert a line such as:

kern.*   /dev/console

to enable console logging of kernel messages. Your mileage may vary depending on the logging configuration of your system. On more recent system versions it seems that syslog.conf has been replaced with syslog-ng.conf, or with a file whose full pathname is /etc/sysconfig/syslog. You should check arounf the /etc directory to find the appropriate syslog configuration file. Note that when you reboot your (virtual) machine after editing this file, you may see many additional text messages, which should not be cause for alarm.This all, however, requires you to have permissions to do this. At the very least, the system administrators should grant you guest privileges to load and unload kernel modules using `insmod` and `rmmod`. Please let the instructor know if you wish to have further privileges to edit system configuration files and build new kernels, if those privileges are not enabled by default.

If you do not modify /etc/syslog.conf, you can always look at the messages in /var/log/messages or simply issue the command `dmesg`, as stated earlier. Alternatively, you might want to insert afunction such as my_printk() (in the provided file) in your module and invoke it instead of using printk().

Again, the exact implementation of my_printk will vary with kernel versions. Tread carefully and use the Linux cross reference as a guide.

- ## Invoking Kernel Services

In this part of the primer assignment, you will want to communicate with your module from user-space, so that it may perform some privileged service on your behalf. Here, we will use ioctls to communicate with the kernel. You should use the template code (ioctl_module.c found via this link, along with the corresponding Makefile that you place in a directory of your choosing) to build a module that sets up the ioctl wrapper code for use by a user-level program. You then need to write a user-level routine that makes an ioctl call to your kernel module to print to the active tty (terminal device) a string that you pass as an argument to the ioctl routine. The format of the ioctl call from your user-level program will be:

ioctl (fd, IOCTL_CMD, &ioctl_args);

Here, `fd' is a file descriptor used for a (pseudo) device file that you create in /proc, IOCTL_CMD is the numeric ioctl command to perform in the body of your kernel module and the final argument is a pointer to a structure containing the arguments that your ioctl code will use. A simple ioctl_test.c file can be found here, for the ioctl_module.c file mentioned above.

NOTE: To print to the active tty you will need to use the my_printk() code described above in your kernel module, along with the string argument passed from your user-level program.

- ## Simple Keyboard Driver

In this part of the assignment, you are required to write yet another ioctl call called `my_getchar()`, that simply polls the PC keyboard for characters. When a key has been pressed (indicated by bit 0 of the keyboard controller status register accessed via port 0x64, using port-based I/O) you want to return from a busy-waiting loop with the appropriate character. Here is some sample code to help:

```
char my_getchar ( void ) {

  char c;

  static char scancode[128] = "\0\e1234567890-=\177\tqwertyuiop[]\n\0asdfghjkl;'`\0\\zxcvbnm,./\0*\0
\0\0\0\0\0\0\0\0\0\0\0\0\000789-
456+1230.\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0";


  /* Poll keyboard status register at port 0x64 checking bit 0 to see if
   * output buffer is full. We continue to poll if the msb of port 0x60
   * (data port) is set, as this indicates out-of-band data or a release
   * keystroke
   */
  while( !(inb( 0x64 ) & 0x1) || ( ( c = inb( 0x60 ) ) & 0x80 ) );

  return scancode[ (int)c ];

}
```

In the above code, inb() and outb() are calls to inline assembly routines to read a byte from a port address, and to write a byte to a port address, respectively, for a given device. These are written as follows and must be executed in your kernel module:

```
static inline unsigned char inb( unsigned short usPort ) {

    unsigned char uch;

    asm volatile( "inb %1,%0" : "=a" (uch) : "Nd" (usPort) );
    return uch;
}

static inline void outb( unsigned char uch, unsigned short usPort ) {

    asm volatile( "outb %0,%1" : : "a" (uch), "Nd" (usPort) );
}
```

Further information about the syntax of the above inline assembly can be found on the main class web-page.
Similarly, further information about how the PC keyboard works can be found via on osdev.org.
Additional details about keyboard scancodes can be found here.

NOTE: For the keyboard driver, you should write a simple user-level program that calls my_getchar() and prints the character read from the keyboard to the screen/shell. You should then extend this user-level program to repeatedly call my_getchar() to read strings of characters.

Adding Support for Keyboard Modifiers and Interrupt-Driven Keyboard Handling

While a basic version of my_getchar() is provided, it does not erase characters on the screen when backspace is pressed, and nor does it support keyboard modifiers such as the shift key followed by another keystroke (e.g., to enable capitalization). You should attempt to add this support. Additionally, you need to disable IRQ1 in the Linux kernel to prevent the Linux keyboard interrupt handler from responding to keystrokes. Finally, you should attempt to implement a keyboard driver solution that uses interrupts rather than polling.

Submission Guidelines

Please submit all your source files, along with a README file to explain how to build and run your code. You should include all provided files that you used or modified. You should also provide sources of information that you used on the web or elsewhere to tackle this assignment. Please use gsubmit on a CS machine and submit to a directory called "primer".

Part 2: Bootloaders and Virtual Disks

One of the most useful ways to develop an OS is to setup a virtual disk image with your OS binary and bootable file system partition. A virtual disk is nothing more than a file that represents a physical storage device. There are many ways to setup a virtual disk but one of the easiest starting points is to use 'dd' or 'qemu-img'. We'll assume you are going to use 'dd' because it is the cleanest way to copy raw bytes from an input file to a given target. Stay tuned for Part 2 in a follow up assignment.