BY: DR B. THANGARAJU

Kernel Corner

A Simple Read and Write Pseudo Character Device Driver

The second part of this series on device drivers discusses some device entry points and the creation of a device file. Sample code for a device driver is also given.

In the first article on Linux device drivers, we learnt how to write a simple module, register or unregister a character device, allocate or deallocate a major number and insert or remove the module. In this article, we extend the same module with some more functionality. In order to understand the device driver concepts, we will focus on a hardware-independent, memory-based pseudo device using only an array in main memory (RAM). The advantage of treating memory as a device is that it is hardware-independent and you can concentrate on device driver principles without being distracted by hardware issues. Once we are familiar with the concepts, it is simple to include any hardware-specific information about an actual device while most of the other parts of your driver code remain the same.

When an application program invokes a system call, the corresponding driver function (known as an entry point) is called internally. These entry points are registered in a *file_operations* structure in the driver code. In our example, we use some of the entry points, the syntax and intricacies of which are discussed in the next section. The macros used for device maintenance and the functions used to move data from and to the device are also discussed. The actual pseudo device driver code has also been provided along with an application program which explains how to send and receive data to/from the device.

THEORY OF ENTRY POINTS

Before we discuss the device entry points, three important structures namely, *file_operations*, *file* and *inode* structures should be clearly understood. First, the *file_operations* structure is an array of function pointers, which is used by the kernel to access the driver's functions. Our sample device driver records the read, write, open and release functions in the file_operations structures as follows:

```
struct file_operations fops = {
read:              my_read,
write:             my_write,
open:              my_open,
release:           my_release
};
```

The function pointers *my_read*, *my_write*, *my_open*, and *my_release* used in the above declaration are assigned to the corresponding *file_operations* device entry points. The structures file and inode are described in the following sections.

open:

```
int (*open)(struct inode *inode, struct file *fp)
```

Two arguments are passed in the open function. The first argument is the inode structure, which is used to retrieve information such as user id, group id, access time, size, major and minor number of a particular device. The

second argument—the file structure—characterises an open device. It has many fields and some of the important ones are:

f_mode: Designates hether the file is read only or write only or both read and write.

f_pos: Points to current file position.

f_flags: Typically used to check whether the file is open in non-blocking (O_NONBLOCK or O_NDELAY) mode.

file_operations: Pointer to file operations structure used to invoke entry points of a particular device, which depends on the minor number.

***private_data**: This points to allocated data for a device.

The complete list of members of the file_operations, file and inode structures can be seen in the *<linux/fs.h>* header file.

To support dynamic module loading, the kernel maintains a usage count for each device. The *MOD_INC_USE_COUNT* macro enables this usage count. Without such usage count, chaos will result if one process accesses the device while another process tries to remove the module. Whenever a process invokes the open function, the count value will be incremented.

release:

```
int (*release)(struct inode *inode, struct file *fp)
```

This function is called when an application program invokes the close system call. Any memory space allocated for the device (e.g., via *private_data) must be freed. **MOD_DEC_USE_COUNT** is then issued to decrement the usage count of the device. If any process asks to remove the driver module, the kernel first checks whether the usage count is zero. Only then will it allow the process to remove the module. Otherwise, the 'Device or Resource Busy' error will be displayed. So the function closes the device only when the last process closes the device.

read:

```
ssize_t (*read)(struct file *fp, char *buff, size_t n, loff_t
*offset)
```

Four arguments have to be passed in the read function—the first and second are pointers to the file structure and user buffer, respectively. The third is the size of the data to be read while the fourth is the file pointer position. Data types *ssize_t*, *size_t*, and *loff_t* designate signed size type, size type and long offset type respectively. When an application program issues the read system call, this function is invoked and the data is to be copied from kernel space to user space using the *copy_to_user* function. The syntax for the function is as follows:

```
copy_to_user(buff, array, n);
```

copy_to_user accepts three arguments—a pointer to the user buffer, a pointer to the kernel buffer and the size of the data to be transferred.

write:

```
ssize_t (*write)(struct file *fp, const char *buff, size_t n,
loff_t *offset)
```

The explanation of the function arguments is the same as that of read. Here, the data should be transferred from user address space to kernel address space. *copy_from_user* function will do this...

```
copy_from_user(array, buff, n);
```

copy_to_user and copy_from_user functions are declared in the *<asm/uacessess.h>* header file.

Most of these functions return 'zero' for success and a negative value for error. If we are only interested in reading from and writing to a device in the following driver code, declarations for open and release functions are not necessary. The driver will work even without these two functions but remember, the kernel may remove the module even though the module is in use by other processes. Therefore, the other processes accessing the device will get the 'OOPS' message with segmentation fault error. But if the open and release functions are included and utilise the usage count macro, the kernel will throw the 'device or resource busy' error and will not remove the module. So it is better to use open and release functions with the usage count to block any removal of the module when it is in use.

WRITING A READ-WRITE CHARACTER DEVICE DRIVER

```
/* This driver module registers a character device named as
'my_device'
 * The device can be accessed like open, close, read and write.
 * printk statement is used in every function to know whether
that
 * function is invoked by a process or not?
 */

#define MODULE     // preprocessor symbol. Should be (cont.)
#define __KERNEL__ // defined to compile modularized kernel
code.
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

static char array[80];  // size allocated for the device
static int Major;

int my_open(struct inode *inode, struct file *fp)
{
       printk("open entry point is invoked\n");
       MOD_INC_USE_COUNT;
       return 0;
}

int my_release(struct inode *inode, struct file *fp)
```

```
{
       printk (“close entry point is invoked\n”);
       MOD_DEC_USE_COUNT;
       return 0;
}

ssize_t my_read(struct file *fp, char *buff, size_t n, loff_t
*offset)
{
       printk(“Inside read entry point:\n”);
       copy_to_user(buff, array, n);
       return 0;
}

ssize_t my_write(struct file *fp, const char *buff, size_t n,
loff_t *offset)
{
       printk(“ Inside write entry point:\n”);
       copy_from_user(array, buff, n);
       return 0;
}

struct file_operations fops = {
read:    my_read,
write:   my_write,
open:    my_open,
release:           my_release
};

int init_module (void)
{
       Major = register_chrdev(0, “my_device”, &fops);
         if (Major == -1)
       {
        printk(“Major number allocation is failed\n”);
          return -1;
       }

   printk (“ Module is inserted successfully \n”);
   printk (“ The major number for your device is %d\n”, Major);
   return 0;
}

void cleanup_module(void)
{
       unregister_chrdev(Major, “my_device”);
       printk(“Module is removed successfully \n”);
}
```

After compiling and loading this module, the output of the printk statement in the *init_module* is displayed. The major number allocated for the *my_device* can be seen in the output or check the */proc/devices* file, where it can be seen in the character device list. In my system, major number 253 was allocated for the *my_device*. Bear in mind that the module only registers the device. You also need to create a device file for accessing the device. Usually, the device file is created in */dev* directory, but in this experiment I don't want to disturb the */dev* directory since root permission is needed to create a device file there. Moreover, if you create file in */dev* directory and, if without your knowledge, you delete some files in the directory, it will affect the normal operations of the system. You can create the device file in the present working directory using the following command.

```
$ mknod my_device c 253 0
```

The *mknod* command is used to create device special files. Four parameters are passed with this command, such as device name, device type (either character or block), major number and minor number. The following application program is used to send and receive data from the device.

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

main () {
       int fd;
       char ch, read_buf[80], write_buf[80];

       fd = open(“my_device”, O_RDWR);
       if (fd == -1)

           {
             printf(“ Error in opening file\n”);
             exit(-1);
           }

      printf (“ What do you want? PRESS r for reading or w for
                                                  writing: “);
      /* First time, should choose writing */
      scanf (“%c”, &ch);

      switch (ch) {
             case ‘w’:
                    printf (“ Enter the message to device: “);
                      scanf (“ %[^\n]”, write_buf);
                      //read until to press ENTER

                      write(fd, write_buf, sizeof(write_buf));
                      break;

          case ‘r’:
             read(fd, read_buf, sizeof(read_buf));
             printf (“ The message from the device is %s\n”,
                                                  read_buf);
             break;

       default:
            printf (“ Press either r or w\n”);
             break;
      } // for switch
} // main end
```

This article has summarised the advantages of a memory-based device, along with some of the device entry points. It discusses the important file operations, file and inode structures and the creation of a device file. A sample device driver program has been given with an application program to access the device, such as using open, close, read and write functions. In the next article, we will enhance this driver code with some more entry points such as *llseek* and *ioctl*. And we will see how to modify the module to handle more than one device. LFY

The author is manager—Talent Transformation at Wipro Technologies, Bangalore. He has been working in the fields of Linux internals, Linux kernel, Linux device drivers, embedded and realtime Linux for the past few years.