NOVATEUR PUBLICATIONS
INTERNATIONAL JOURNAL OF RESEARCH PUBLICATIONS IN ENGINEERING AND TECHNOLOGY [IJRPET]
NATIONAL CONFERENCE ON INNOVATIVE TRENDS IN ENGINEERING & TECHNOLOGY-2017
15TH & 16TH MARCH 2017
CONFERENCE PROCEEDINGS ISSN NO - 2454-7875

Paper ID: NITET03

KERNEL MODULES AND DEVICE DRIVERS, DEMYSTIFIED

RAJESH D. NAGAWADE
Head, Learning& Development, SVERI, Pandharpur, rajesh.nagawade@gmail.com

M.A. KHURESHI
Head, CSE Department, AGPIT, Solapur, Haq123.275@gmail.com

S. M. KUMBAR
Assistant Prof. CSE Department, SVERI, Pandharpur,sharan64.kumbar@gmail.com

ABSTRACT

Linux Operating System provides services at various levels; starting from commands to shell, system calls in application level, kernel level calls and function calls at driver level. This is the entire path for a user to reach out to the actual devices like hard disk, keyboard etc.

Kernel modules are conceptually (and in code also) different than ordinary application programs. Device driver programs are basically kernel modules. Also there are other kernel modules which are not device drivers.

This paper attempts to demystify this concept. It starts with explaining the concept of kernel module with an example. It then proceeds with the concepts of writing device drivers. Example source code of character device driver is chosen for illustration.

Objective is to simplify various complex concepts related to kernel modules and device drivers. This should enable the reader to design and write basic device drivers. Audience is expected to be familiar with Linux systems programming.

INTRODUCTION / AGENDA

As new devices are introduced their device drivers need to be developed. Although their look and feel will be similar, minute details need to be worked out for expected results. Also sometimes existing drivers need to be modified for specific purposes. Porting devices to newer operating systems is also a task many a times. Some times kernel modules also need to be customized for newer requirements. This requires many a people to know, design and develop kernel modules and device drivers.

This paper attempts to demystify this concept. It starts with explaining the concept of kernel module with an example. It then proceeds with the concepts of writing device drivers with an example of generic device driver. Source codes are provided for actual exposure.

What to expect: generic concepts of kernel modules, device drivers and their interdependencies.

What not to expect: specific details of any particular device or its driver. Running sample codes for both kernel module and a driver are provided to support concepts.

KERNEL MODULES

A) JOURNEY FROM USER APPLICATION LEVEL TO MODULE LEVEL

Linux Operating System provides services at various levels; starting from commands to shell, system calls in application level, kernel level calls and function calls at driver level. This is the entire path for a user to reach out to the actual devices like hard disk, keyboard etc.

Kernel modules are conceptually (and in code also) different than ordinary application programs. Device driver programs are basically kernel modules. Also there are other kernel modules which are not device drivers.

User types command, shell creates processes to cater for this command. All such processes and shell itself may make use of system calls. This is a user or application level operation.

Shell itself is an application program just like other programs that we develop. The only difference is that it got started by some system process. (i.e. login process). We are at uppermost level then.

Of course shell and application programs need not be referred to as part of kernel albeit they are part of the system.

These programs can make system calls to avail of privileged services from kernel. These services are available only when process runs in kernel mode. These system calls cane be termed as static portion of the kernel.

I would prefer to term these, as static kernel because these are active only when called by a process. And because they do not represent a process, neither these have a main ( ) in their code. These are similar to ordinary functions but run in kernel mode and have access to vital kernel data structures like fd table, process table etc.

Apart from system calls kernel needs to do many activities on periodic basic. Hence it has many processes running in parallel like init 0, init 1, swapper, scheduler, page daemon etc. These are basically programs and are very similar to application programs except that these kernel processes run most of their life in kernel mode. I would prefer to term this as dynamic kernel.

These kernel processes, data structures, system calls are never swapped out of memory unlike application processes which are swappable. Thus this is resident kernel.

Now let us see kernel module in contrast with system calls and kernel processes.

- Kernel modules are not supposed to run forever like kernel processes.
- These are not called by user programs like system calls.
- Their executable / binary code need not exist all the time.
- Kernel module source code has a definite entry and exit point but does not have a main () function.
- If binary can be linked with kernel, it will be part of resident kernel.
- If the binary is not linked with kernel and is loaded when needed, it will not be part of resident kernel.
- In this view of load or link approach – module can be part of static or dynamic kernel.
- Module linked statically with kernel will require recompilation and reboot of kernel.
- Load and remove of module can be done during run of the kernel.

B) MODULE, APPLICATION PROGRAM AND SYSTEM CALLS

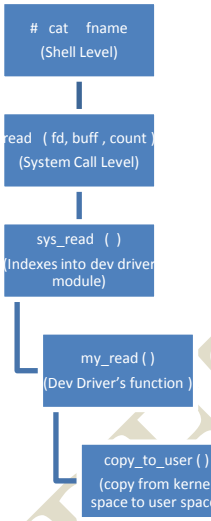

Figure No 1. Hierarchy of calls happening from user application to device driver

Figure no 1 illustrates the programming levels viz. application level, shell level, system call and device driver level.

C) UTILITIES FOR WORKING WITH MODULES

Having logged in as super - user following utilities are available for working with modules.

- **Insmod** allows us to insert a module dynamically into kernel when needed. It will remain in system unless we remove.

  If any other module gets inserted which uses this module, the earlier module's uses count will be incremented.
- **rmmod** decrements the uses count of the given module and removes it if the uses count reaches 0.
- **lsmod** displays all modules currently inserted in the kernel. It is a module variant command for ls.

  For example **mod.c** can be the source code for module. Compiling without linking will produce **mod.ko**. It can be inserted in kernel as follows.

  # insmod mod.ko

  Whether the insertion succeeded or not can be checked by

  # lsmod | grep mod.ko

  This will also list us the size of the module inserted and its uses count.

  D) Makefile for compiling Modules

  Modules may have internal dependencies. Instead of manually keeping track of such dependencies, it is wiser to use a makefile for compiling modules.

Following file namely **Makefile** can be used. It is generic in syntax and uses shell variables. So the same can be used for different modules also, provided the module name in **Makefile** is changed.

With mod.c as module name, mod.ko can be prepared by this file by **make** command.

```
KERNEL_BUILD := /lib/modules/`uname –r'/build
CC := gcc
PROG := app
obj-m := mod.o
modules:
        @$(MAKE) -C $(KERNEL_BUILD) M=$(PWD) $@
$(PROG): $(PROG).c
        @$(CC) -o $(PROG) $(PROG).c
```

Following operations can be thus made.

```
# make                  ( prepares mod.ko )
# insmod mod.ko         ( inserts mod.ko )
# lsmod | grep mo       ( verifies insertion )
```

Let us say an application program **app.c** uses this module. Its binary can be prepared by

```
# make app
```

During the make, inside **Makefile**,

- `uname –r ' will get replaced by the present Linux version name path.
- PROG will get replaced by app
- PWD will get replaced by present working directory.

GENERIC KERNEL MODULE

Let us see what might be the minimal source code for a kernel module that may get repeated for most of the modules. We may have to add up additional functionality into this as per the task for which the module is designed.

A) OBSERVATIONS FOR A GENERIC KERNEL MODULE

- **insmod** internally calls **module_init ( )** function and rmmod internally calls **module_exit ( )** function. So the module must map his

initialization and clean up functions to these functions, respectively.

- In order to be able to give our chosen names to init and clean up function the mapping can be done as follows.
  module_init (mod_init);
  module_exit (mod_clean);
- Module writer is responsible to define mod_init( ) and mod_clean( ). Kernel will make sure to call these during module insertion and clean up.
- The modules cannot make any user level function calls.
- It cannot make system calls or library function calls. For example calling printf ( ), read ( ), fread () from a module will not be permitted.
- Module init and exit functions must have their execution time pretty small.
- A module cannot output on standard output but its output can been seen by # dmesg. This will show all messages logged in by modules till now.
- # dmesg –c will clear the log.

B) ALGORITHM TO WRITE A MODULE

Following algorithm helps one write a generic module.

1) Include the files init.h and module.h from linux folder
2) Announce the license as Dual BSD / GPL
3) Write definition for functions which will do module initialization and module clean up – say F1 and F2, respectively
4) By using module_init () and module_exit() function calls, associate these names F1 and F2 for init and cleanup work.

DEVICE DRIVER AS A KERNEL MODULE

A device driver is a kernel module that can be statically linked with or dynamically inserted in kernel. Kernel reaches out to such driver by treating it as a file in directory /dev. For example /tty represents keyboard – monitor combination. The major number and minor number fields of the inode of such file map into the specific device driver.

The major number decides a class of devices and hence their driver. And minor number differentiates between the various devices that the driver can handle.

When user calls system calls like open (), read (), close ( ), ioctl () kernel refers to the file's inode and internally uses its major and minor numbers.

Kernel uses a data structure called as file operations data structure. This data structure has fields as pointers to functions as shown below.

```
struct file_operations {
        llseek      :   my_llseek,
        read        :   my_read,
        write       :   my_write,
        ioctl       :   my_ioctl,
        open        :   my_open,
        release     :   my_release,
        owner       :THIS_MODULE};
```

Now depending upon the functionality the device needs to provide, the device driver writer needs to define his functions and assign these pointers to his functions. Such file operations structure then needs to be passed to the following function.

```
struct file_operations my_fops;
reg_chrdev ( MAJOR_NO , " mydevice " , &my_fops ) ;
```

This registers the driver's functions with the kernel; thus the driver is registered.

As user calls the functions kernel uses this structure to locate driver's functions that you define.

Here MAJOR_NO is the major number for the device file namely mydevice. This file represents our driver at kernel file system. This is created as by

```
# mknod /dev/mydevice c 250 0
```

c here indicates it is a character device and 250 is its major number and 0 is minor number.

Kernel thus calls our driver's functions. At that instant it gives us file *, inode * and other information as parameters depending upon the file user wants to work upon.

```
struct file *
struct inode *
```

Using these pointers to structures the driver functions can dig into file relevant details like major number, minor number, size of file, owner, access permissions etc. There are nested structures inside structures giving us more information.

Driver can assign his own buffers for storing transient data during transfer. Ideally this data should be allocated using kalloc ( ) and care should be taken that its use is synchronized between multiple instances of driver calls. That is driver routines should be re entrant.

A GENERIC DEVICE DRIVER

Let us consider a generic device driver, that is the driver functionally that must be common between all the drivers. This code will be architecture independent and should be found in most of the drivers.

Let us say we want to provide my open, myriad, mis write , my close and my IOCtl functions inside device driver module. So we will map these using fops structure.

Also we decide to make a device file entry by name "my device" under / dev directory by using mknod.

The kbuf array and DriverData represents driver level data.

User's read system call will receive data from kbuf during my_read ( ) routine of driver. This is managed by copy_to_user ( ) kernel function. Also from user's data given in write system call kbuf will get filled in during my_write ( ) routine of driver. This is managed by copy_from_user ( ) kernel function.

This will complete the interface from user application to kernel to the device driver. Now depending upon different devices and their functionality the other section of the driver that is interacting with actual device will change.

This other section will demand for studying the actual behavior and architecture of the specific device. Before we discuss such device specific changes, let us see the sample source code for such generic device driver.

A) ALGORITHM TO WRITE A GENERIC DEVICE DRIVER

1) Define MAJOR_NO as 250
2) Include the files module's, init.h, fs.h and uaccess.h from linux folder
3) consider major as integer and kbuf as char array
4) DriverData structure holds val as long and str as char array – as members.
5) struct inode * and struct file * parameters are available to my_open function. Define such function to find minor number using following call
   minor = MINOR(filp->f_dentry->d_inode->i_rdev)
6) In the similar manner define my_close function.
7) In the same manner define my_read function and iit will call the function copy_to_user (buf, kbuf, count)
8) Define my_write function and it will call copy_to_user (buf, kbuf, count)
9) Functions my_read and my_write take following parameters (struct file * filp, char * buf, size_t count, loff_t * offset)
10) Function for control operations take following parameters (struct inode * inode , struct file * fp , unsigned int cmd , unsigned long arg)
11) Following will find driver data for control data = (struct DriverData *) arg ; after this data -> val integer and data -> str string is available for use.
12) Now define file operations structure where we assign read to my_read, write to my_write, open to my open, release to my close, IOCtl to my control and owner to THIS_MODULE

B) ALGORITHM FOR PROGRAM USING THIS DRIVER

Following algorithm may use such driver code.

1) Ubuff is character array
2) User Data is structure with val as integer and str as string – say we take data as variable of this structure
3) We open the file /dev/mydevice in read write mode
4) We proceed to assign value say 100 to data. Val
5) We assign some string say "I Came From Application" to data.str
6) Now we call ioctl (fd, 0, &data)
7) Then we call read (fd, ubuff, 15)
8) This should give us value for ubuff
9) Now we may change ubuff to say "Take this from application "
10) And we may then call write (fd , ubuff , strlen (ubuff));
11) This should copy it to kernel and device through our driver functions. We can then close file.

- It calls ioctl and passes over the string "I came from application" to the driver. In a specific driver other information needed for controlling device can be passed over here.
- It calls read ( ) and accepts the string given by the driver. This can be the actual data that has been read from the device by the driver.
- It calls write and passes over the string "Take this from application", to the driver. This can then be written to the data space of the device at hand.

C) OBSERVATIONS FOR SUCH GENERIC DEVICE DRIVER

This sample device driver implements read, write, ioctl, open calls for the device represented by the file name mydevice and major number of 250.

When an application calls

read ( ) driver's my_read ( ) gets called. Being a generic driver it simply copies it's kbuf's data into the application to cater his read request.

write ( ) driver's my_write ( ) gets called. Being a generic driver it simply retrieves data from the application and copies it into the driver's kbuf.

D) HOW TO TACKLE SPECIFIC DEVICES

Herein we have not tackled where does kbuf gets data from before giving to application, and also what does driver do with the kbuf data after taking it from the user application.

Answers to these questions depend upon what device driver we are designing and what are the ways to interact with the device.

CONCLUSION:

Device drivers have a generic section that is similar in most of the drivers. As per its interaction with devices the other section will change

It works as a lower layer to kernel system calls.

It is considered as a sub layer below kernel file system.

Keeping it static and loadable during testing phase is advisable. It can be linked with kernel once tested completely.Future work in this can be as follows.

Obeying the same skeleton various other device drivers can be tackled. These will have similar approach except few device specific changes. In order to achieve this, following areas needs to be focused namely PCI Architecture, Port IO Provision, Memory IO Provision. Modules can be developed on Linux and cross compiled for working on other architectures like ARM.

REFERENCES

1) Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, Linux Device Drivers, Third Edition
2) Maurrice J. Bach, Design of UNIX
3) W. R. Stevens, UNIX Network Programming