LINUX KERNEL AND DEVICE DRIVER PROGRAMMING

A SIMPLER APPROACH TO LINUX KERNEL

LINUX KERNEL AND DEVICE DRIVER PROGRAMMING

A SIMPLER APPROACH TO LINUX KERNEL

By

Mohan Lal Jangir
Development Manager
Samsung India Software Operations
Bangalore (Karnataka)

UNIVERSITY SCIENCE PRESS

(An Imprint of Laxmi Publications (P) Ltd.)

BANGALORE ● CHENNAI ● COCHIN ● GUWAHATI ● HYDERABAD
JALANDHAR ● KOLKATA ● LUCKNOW ● MUMBAI ● RANCHI
NEW DELHI ● BOSTON, USA

Published by :

UNIVERSITY SCIENCE PRESS

(An Imprint of Laxmi Publications (P) Ltd.)
113, Golden House, Darya Ganj,
New Delhi-110 002
Phone : 011-43 53 25 00
Fax : 011-43 53 25 28
www.laxmipublications.com
info@laxmipublications.com

Price : ₹ **175.00** *Only.* *First Edition* : 2014

OFFICES

✆ **Bangalore**	080-26 75 69 30	✆ **Jalandhar**	0181-222 12 72
✆ **Chennai**	044-24 34 47 26	✆ **Kolkata**	033-22 27 43 84
✆ **Cochin**	0484-237 70 04, 405 13 03	✆ **Lucknow**	0522-220 99 16
✆ **Guwahati**	0361-254 36 69, 251 38 81	✆ **Mumbai**	022-24 91 54 15, 24 92 78 69
✆ **Hyderabad**	040-24 65 23 33	✆ **Ranchi**	0651-220 44 64

ULK-9717-175-LINUX KER DEV DRI PROG-JAN **C—**

Typeset at : *Excellent*, Delhi. *Printed at* :

Contents

Preface

Being an electrical engineering student at my undergraduate level, my exposure to Linux started rather late and was limited to C programming on Linux PC. My true interaction with Linux started during my post graduation at IIT DELHI, wherein I started learning Linux shell commands, Vi editor etc. Still my experience remained limited to user space work like programming, networking and basic system administration.

My first encounter with kernel programming happened when I was given an assignment to write a device driver for mouse. Although I could succeed in getting mouse events, I could not complete the assignment. But it left an interesting area open to me which I continued to explore till date.

The idea of sharing the experience started when I finished a self assignment to write a network device driver on 2.4 kernel. I wrote an article and published it on Linux Malta group. The article turned out to be an immensely popular which increased my faith in sharing my experience. Later, the same article was re-published in Linux Gazette magazine because Linux Malta group website went down.

Based on the responses from the readers I realized that while many good, in-fact very good, reference books are available about Linux device driver programming, still there is a void in teaching an end to end driver development. Specifically a sample and real hardware made users to do the real development from scratch.

Encouraged from this article's success, I decided to write similar articles for block, USB, char, UART drivers as well and put them together along with kernel programming articles so that a reader finds all of them in one place. That's how this book has originated. All the code samples in this book are written or re-written for 2.6.21 kernel, however wherever needed, reference to older kernel are also been given so that readers may know the difference if they have to work on older kernels.

The examples sited throughout this book are trimmed versions of the self assignments done at home on Linux running on AMD AthlonTM processor. Specifically the error handling part is either removed or is minimal for the sake of brevity. Also the driver design is kept very simple and minimal basic functions are implemented for easy understanding. This will help readers to quickly understand and then modify them to add more features, error handling and bug fixing to grasp better understanding.

Audience of This Book

This book is written for students or professionals who quickly want to learn Linux Kernel programming and device driver development. Each chapter in this book is associated with code samples and code commentary so that the readers may quickly understand.

The readers are expected to have C programming experience and minimum know how of Linux. Every chapter in this book has been written from basic so that readers have no difficulty in understanding them.

Organization of This Book

This book can be classified in three parts. The first part is from chapter 1 to chapter 6 which is about kernel programming. These chapters explain kernel functionalities, debugging mechanisms, communication mechanisms etc.

The second part is from chapter 7 to chapter 9 which is dedicated for device driver development. These chapters pick a real hardware and explain device driver development from scratch. These chapters also include references to specifications, registers, and protocols etc. which are used by the devices, so that readers find all needful data at one place.

The third part is chapter 10 which explains about some common mechanism likes memory allocation, kernel threads, inter processor communications etc.

Readers are encouraged to modify, enhance and fix the sample code specifically on non-x86 and multi-processor architectures. This will give readers more confidence in kernel and device driver programming.

—Author

Acknowledgement

At the outset, I would like to thank the publishers and editors to make this book in reality.

This book is an outcome of my learning experiments done with Linux kernel. I thank my parents and family for their blessings and support. Specially my elder brother Dr. Manoj Jangid who has been an encouraging, supporting and inspiring figure in my life.

Writing this book was a long process and I must thank my wife Devyani for providing encouragement and support throughout even though she was completely held up by our one year old son Vatsal. She patiently managed things alone when I spent hours to complete the experiments and to convert them into this book.

I feel gratitude to Linux open source community who has answered many questions on mailing lists. It was a great help from them whenever I stuck somewhere during experiments.

Lastly, I thank my article readers whose feedback and response have inspired me to write this book. It has been a pleasant experience to share my experiments with them.

—Author

1 Beginning Kernel Programming

Linux being an open source operating system gives flexibility to customize the kernel based on user needs. This means, that a user is free to include whatever needed and exclude everything else to keep the kernel size small. In this chapter, we begin with kernel compilation and then develop first kernel module.

1.1 MAKING A NEW KERNEL

Linux kernel programmers very often need to build new kernel for different hardware or for different products. This section is for beginners who are compiling kernel for the first time. If you have compiled and run new kernel previously, you can safely skip this section.

1.1.1 Know Your System

For compiling new kernel, you have to first gather information about what to include in the kernel and what can be excluded. If you are working with a new system, the hardware reference manual is your best bet. To begin with, you need to gather at least the following information:

- Processor—Manufacturer, number of cores/processors
- Bus (PCI, ISA, PCMCIA, USB etc)
- Network devices
- Display devices
- File systems to be used
- Peripheral devices

Next, you can configure the kernel to include support for required hardware.

1.1.2 Configuring and Compiling Kernel

You can download kernel source from http://www.kernel.org/. There are many ways to configure the kernel. If you are working in graphical mode, you can use “make xconfig” or “make gconfig” which provides very easy and convenient way to configure the kernel. If you don’t have graphical mode but can run Tcl/Tk, you can use “make menuconfig”, else you can use “make config” which works completely in text mode.

You can choose any feature to be included in kernel (built-in), or can choose to compile as module (runtime loadable module). Therefore, it is possible to keep necessary features to be built in kernel while optional features can be configured as module, and can be loaded on demand. Your configuration is saved in ".config" file in kernel tree.

Alternatively, if you have an already prepared configuration file, you can directly copy that file to .config, and can run "make oldconfig".

In older kernels, "make dep" was required after configuration to generate dependencies. However, 2.6 kernel does not need "make dep".

To compile the kernel you need the following commands:

- make—compiles compressed kernel image. If you know target name for compressed kernel image, you can use that command also. For example, for x86 architecture, you can use "make bzImage" image.
- make modules—compiles kernel modules.
- make modules_install—installs kernel modules in /lib/modules/'uname –r'/ directory. If you want to install in other directory, you can use "make modules_install INSTALL_MOD_PATH=/path/to/modules/dir" command.

The compressed kernel image is created either in main kernel tree or some architecture specific directory. For example, for x86 architecture you will find compress kernel image at arch/i386/boot/bzImage location.

1.1.3 Making initrd Image and Installing Kernel

After compiling the kernel, you need to make initrd image (initial ram disk). You can use mkinitramfs command to build initrd image. The initrd image name can be specified with –o option. Kernel version must be specified at the end. Below is an example to create initrd for 2.6.30 kernel.

```
mkinitramfs -o /boot/initrd.img-2.6.30 2.6.30
```

Next you need to copy kernel compressed image and initrd image to boot directory which can be /boot or /boot/grub based on the boot loader. Finally you need to edit boot loader configuration file (lilo.conf, grub.conf etc) to include the new kernel. Some boot loaders (like lilo) requires reinstalling boot loader after modifying boot loader configuration file.

You can reboot the system now. During next boot, you can choose to boot into the new kernel from boot loader menu.

1.2 PATCHING THE KERNEL

Sometimes it is required to patch the kernel to include source code of some work in progress or for some external feature. Two utilities 'diff' and 'patch' are useful in generating and applying patches. First let's understand how to generate a patch.

Assume that you have original kernel source in a directory named linux-old. You copy the contents of the linux-old into linux-new directory and work in linux-new directory. After finishing your changes are in linux-new directory, and you want to deliver the changes to somebody else. However, it is not a good idea to send entire linux-new directory. To deliver only the changes you made, you can create a patch and send the patch file.

The following command generates a patch for above mentioned directories.

```
diff –Nur linux-old/ linux-new/ > changes.patch
```

The N option instructs diff to treat absent files as empty. The u option instructs to generate unified format diff (which is used by patch utility). The r option instructs to look into directories recursively.

Now you can send file changes.patch to somebody else, who wants to use these changes. They can use patch utility to apply changes available in changes.patch file to linux-old directory.

The patch can be applies as follows:

```
patch –p0 < changes.patch
```

The p option instructs to strip the smallest prefix containing num (digit following p option) leading slashes from each file name found in the patch file. For example, if you want to apply the patch from inside linux-old directory, you must choose –p1 option.

However, before applying a patch, it is always advisable to use —dry-run option which acts as applying patch but it really does not apply. This gives you a view of how the patch would be applied.

1.3 HELLO WORLD MODULE AND MAKEFILE

Let's write our first kernel module. Similar to main function in C application, kernel modules have init_module function which is executed when the module is inserted into the kernel using insmod command. In addition to that, modules have an exit function cleanup_module which is called when module is removed from kernel using rmmod command.

Table 1.1 illustrates a hello world kernel module.

Table 1.1

```
/* hello.c */
#include <linux/kernel.h>
#include <linux/module.h>
int init_module(void)
{
        printk("hello world\n");
        return 0;
}
void cleanup_function(void)
{
        printk("good bye\n");
        return;
}
MODULE_LICENSE("GPL");
```

The MODULE_LICENSE macro specifies kind of module; which can be GPL if you are willing to release the module under GPL license.

To compile kernel modules, you need to use kernel build. Table 1.2 illustrates a Makefile for above shown module.

Table 1.2

```
obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
```