Linux Kernel Module and Device Driver Development

Thomas Zink

University of Konstanz
thomas.zink@uni-konstanz.de

Abstract. Linux makes life easy for the device driver developer. The Kernel is open source and highly modular. However, albeit these good preconditions there are also some downsides. Finding good and up-to-date documentation is not always easy since some things often change from one kernel release to another. The following examples have been taken from different sources and modified and tested under Ubuntu 7.10 with Kernel 2.6.21.17. To start Kernel hacking one should be familiar with c/c++.

1 Introduction

Before diving into kernel hacking a deeper look into the kernel itself is necessary. The Linux operating system basically consists of two spaces, the user space, where applications live and the kernel space, where all the operating system processes reside, like memory and process management and drivers (Fig.1 [1]). This distinction is important for the kernel developer, since user and kernel space

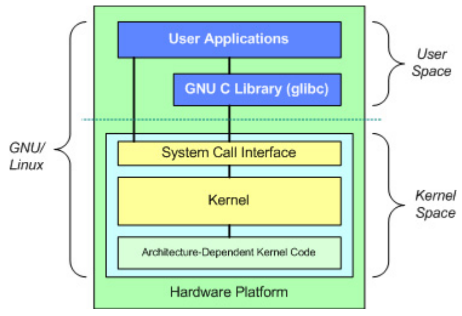

Fig. 1. GNU/Linux

have their own address spaces and one must take special care when moving data from one space to another. This will be covered later in more detail.

[1] image taken from [1]

The Kernel itself consists of three levels or layers. The top layer is the System Call Interface (SIC). Every user space process must make use of this well defined interface to communicate with the kernel. The POSIX standard currently defines more than 300 system calls, a complete list of system calls implemented in Linux can be found in `<linux/include/asm/unistd.h>`.
Below the SIC lies the kernel code which can be separated into a number of subsystems as shown in Fig.2 [2]. The significant properties of the kernel code are that it is architecture independant and modular. The kernel allows code insertion during runtime in form of loadable modules. As we will see later, device drivers are often realized as kernel modules. A number of these subsystems have already

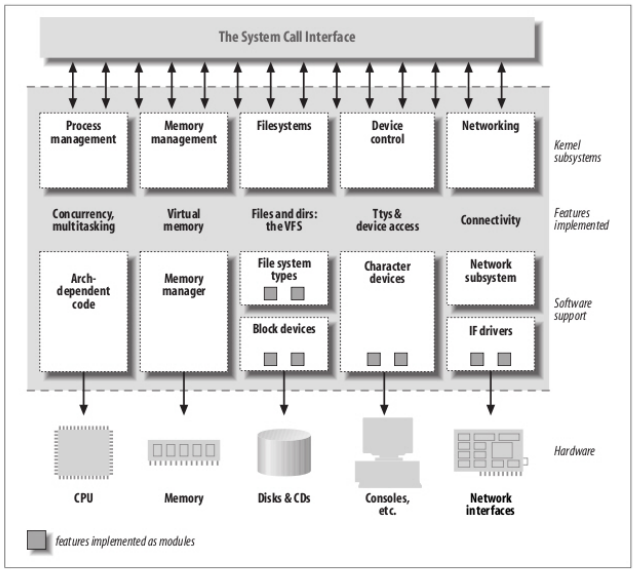

Fig. 2. Linux Kernel

been discussed in other presentations, so there will be no recapitulation here. The focus of this work lies on character devices. Block and network devices will not be covered in detail.
At the lowest layer the architecture-dependent code resides which talks directly to the specific hardware architectures.
Now that the basics of the Linux Kernel have been discussed, we can take a deeper look at kernel modules and device drivers.

[2] image taken from [2]

2 Modules and Drivers

2.1 Kernel Modules

Modules are kernel extensions that can be loaded and unloaded during runtime. They are implemented as object files (kernel objects, extension .ko) and are dynamically linked to the kernel.
This special feature allows smaller kernels, since only code necessary to boot the machine needs to be integrated and all additional code can be inserted later. This is one of the reasons why Linux has this high degree of portability and scalability. Modules are often present in the form of device drivers but not limited to those. Any feature can be implemented as a kernel module which is also done for filesystems and network features, like VPN to name a few. In addition a modular design allows faster kernel development and testing, the system has not to be rebooted after every change, instead the code can be loaded and unloaded as development and testing progresses.
When working with modules the following tools and paths are of special interest.

`</lib/modules/‘uname -r‘>`	location of all the modules
`lsmod, insmod, rmmod` `</proc/modules>`	list, insert and remove modules. contains all loaded modules
`modprobe`	intelligently adds or removes a module from the Linux kernel
`</etc/modprobe.conf>` `</etc/modprobe.d/>`	allow special options, requirements, aliases, blacklists for module handling
`modinfo`	print information about specified module like path/filename, licence, versionmagic, author

2.2 Device Drivers

Device Drivers control access to all devices. Physical devices are accessed through interfaces and are represented by one or more device files. They provide functions to communicate with the device but hide all the details how communication works.
As can be seen in Fig.2 device drivers basically implement one of the following fundamental device types:

1. **Character Devices**
   Accessed (in general) sequentially via byte streams.
2. **Block Devices**
   Can hold a filesystem. Can handle (block) I/Os. Can be accessed randomly.
3. **Network Interfaces**
   Handles network transactions. Can communicate with other hosts. Usually packet oriented.

Hardware devices can be attached via different bus systems. To simplify things and to speed up driver development Linux uses a layered or stacked driver system with standardized interfaces. Every interface implements it's own data structures, and I/O-controls. The levels are:

1. **High-Level-Driver**
   Communicates with the user application, responsible for the actual device. It does not talk directly to the device, this is done by the low-level driver. The casual driver developer will mostly work at this level.
2. **Core-Driver**
   Device and driver management. The core-level driver provides functions that are specific for certain device types. It identifies connected devices, loads the specific driver and manages multiple devices attached to a bus.
3. **Low-Level-Driver**
   Communicates with hardware, handles data transfers. This is the driver that actually talks to specific chips or controllers by accessing it's registers directly.

The trend certainly is to implement device drivers as modules. However, some drivers have to be integrated into the kernel especially those that are needed at system startup. Kernel modules must be compiled for the exact same kernel version they are used with.

3 Kernel Hacking

Apart from theoretical knowledge the ongoing kernel developer must also master development tools and take care of practical conventions. Kernel coding is best done on a dedicated development system, running the kernel for which you would like to develop. One reason for this is that the Linux kernel allows some nice development features to be enabled which are disabled for most standard kernels delivered by distributors. Also, one would like to install additional development man pages which include pages for system calls and the like. They can be accessed with `man 2 <command>`. Then a useful editor must be chosen, for a nice comparison see [5]. It is common practice in the community to use versioning systems for code maintenance. The system used for the linux kernel is Git (http://git.or.cz/), those serious in becoming a kernel hacker should take a closer look at it. The code itself should also be documented well and be written according to the kernel coding style. This can be found in `/usr/src/linux/Documentation/CodingStyle`. Another document of note is `/usr/src/linux/Documentation/HOWTO` the "HOWTO do Linux kernel development". Other documentation can be found in this directory too, however, often pretty outdated. The kernel sources should be installed in `/usr/src/linux/`, you will also need `modtils`. It is advisable to activate the "Forced module unloading" option in the kernel, with this option a broken module can be forced to unload with `rmmod -f`. You will need `gcc` and `make`.
Now that theory and practical issues are covered we dive into programming.

3.1 Writing Kernel Modules

Let's start with a first and famous Hello World module.

```
#include <linux/version.h>     // define kernel version
#include <linux/module.h>     // dynamic loading of modules into the kernel
#include <linux/kernel.h>     // contains often-used function prototypes etc

MODULE_LICENSE("GPL");

// called on insmod
int init_module(void) {
        printk(KERN_INFO "hellomod: init_module called\n");
        return 0;
}
// called on rmmod
void cleanup_module(void) {
        printk(KERN_INFO "hellomod: cleanup_module called\n");
}
```

Here we see some specialties of modules and differences to application. A module always needs the two functions `init_module()` and `exit_module()`. They can be renamed but then a new header must be included and special macros used as a reference to the init and exit functions.

```
/* other headers here */
#include <linux/init.h>    // for init and exit
static int __init hellomod_init(void) { /* code here */ }
static void __exit hellomod_exit(void) { /* code here */ }
module_init(hellomod_init);
module_exit(hellomod_exit);
```

Of the three `#include` statements only `module.h` is really needed for all modules, but the other two are unavoidable when it comes to serious programming, so they have been included in this example (`kernel.h` is in fact needed here for `KERN_INFO`). Another specialty for modules is the macro `MODULE_LICENSE`. This is not really needed but it affects what kernel functions the module can use and how much support you can expect from the community. It is suggested to use "GPL", for a complete list of supported license models and other module macros see the header file itself.
The module can be compiled using `gcc` but it is recommended to use a makefile. We won't go deeper into writing makefiles, a documentation can be found in `<Documentation/kbuild/makefiles.txt>`.
A simple makefile for the above module:

```
obj-m += hellomod.o
KDIR := /lib/modules/`uname -r`/build
PWD := `pwd`
```

```
default:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
        rm -rf *.o *.mod.* *.sym* *.ko .*.cmd .tmp*
```

This file must be saved in the same directory as the module with the name `Makefile`. Then the module can be compiled using `make`. `make clean` cleans up again. Modules are compiled as object files, so called kernel objects with the extension .ko. When loaded they are dynamically linked to the kernel. They run in kernel space and thus share a common address space and namespace with the rest of the kernel.
After compiling the module it can be loaded using `insmod hellomod.ko`. Output will be written to `/var/log/messages`. This concludes the short introduction in module writing, a deeper manual can be found in [4]

3.2 Writing Device Drivers

To become a driver the module must be associated with a device. Recall that a device is represented as a file in linux. Device files are located in `/dev` and look like this:
`brw-rw---- 1 root disk 8, 2 yyy-mm-dd hh:mm /dev/sda2`
The first character indicates the type of device, in this case a block device (character devices have c). After owner and group come two numbers of significant importance, the major number (8) and the minor number (2) . The major number is used to identify the driver responsible for this device. The minor number is used to distinguish between different instances of the same driver, in this case one driver called `sd` handles different disks and partitions, they all will have major number 8 and different minor numbers.
Device drivers have the following prerequisits:

1. **Driver Name**
   Each driver must have a unique name, which is used to identify it. The name will appear in `/proc/devices`. Don't confuse the driver name with the name of the device file or the driver source file. They can be the same but one can't count on that. It is tender to define the driver name as a constant.
2. **Register Driver and Device**
   The driver must be registered at the appropriate device subsystem depending on the device type (block, character, network, SCSI, PCI, USB etc). The kernel provides appropriate functions for registering and unregistering the driver. The registration needs the driver name, the major number (in fact the major number is not needed prior to registration as seen later) and a data structure to hold the results.
   In addition the device itself must also be registered or initialized. This comes with allocating resources, requesting i/o queues etc.
3. **Data Structures**
   Depending on the device type and driver subsystem, some data structures

are necessary. They hold device specific information (like ID), map system calls to driver functions (like open, read, write etc) and map resources to the specific device (like request queues).

A bare template for a simple character device driver could look like the following listing:

```
// hello_driver_barebone.c
#include <linux/kernel.h>    // often used prototypes and functions
#include <linux/module.h>    // all modules need this
#include <linux/fs.h>        // file structures

MODULE_LICENSE("GPL");

// Prototypes
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

// device name as it appears in /proc/devices
#define DRIVER_NAME "hellodrv"

// Major number assigned to device driver
static int Major;

// fops must be declared after functions or the prototypes
static struct file_operations fops = {
    .open    = device_open,      // open device file
    .release = device_release,   // close device file
    .read    = device_read,      // read from device
    .write   = device_write,     // write to device
};

// Functions below
```

When inserting this driver, the function `init_module` will be called. It is here, that driver registration takes place. The template for registration looks like this:
`register_{device}(Major, DRIVER_NAME, &fops);`
{device} is dependent of the device type at hand. If a major number is statically assigned this one can be passed. If one wants a dynamically assigned major number then 0 must be passed, in this case the register function returns the major number for this driver. So a legal registration of the character device in the upper example would be:
`Major = register_chrdev(0, DRIVER_NAME, &fops);`

The function `cleanup_module` is called on exiting the module. Here unregistration is done.

```
int ret = unregister_chrdev(Major, DRIVER_NAME);
```

Again the major number and driver name are needed. The unregister function returns 0 on success or a negative failure number. A number of other functions have been added. They resemble operations applied to the device file and are invoked by the user application through system calls. The names in the file operations structure must be identical to the names of the functions. They can be chosen freely by the developer. `device_open` is called upon opening the device file, it returns 0 on success or a negative error number. `device_release` is called when the file is closed and behaves like `device_open`. `device_read` and `device_write` are called on a read or write to the device file. In the argument list we find the number of bytes to read/write and the offset within the file. They return the actual number of bytes read/written or a negative error.
In order to actually use the driver a device file is needed. In this example it must be created manually by using the command `mknod /dev/<devfile> c <major> 0` . The filename can be freely chosen, but must not be already present or occupied by another driver. The major number must be the one assigned to the driver on registration. To retrieve it one could use `printk` to have the driver write the major number to `/var/log/messages`.

Reading and Writing. As already noted applications run in user space, drivers in kernel space. They have their own memory address spaces. So in order to read from/write to a device there must be functions that provide moving data from user to kernel space and vice versa. The header `<asm/uaccess.h>` defines four functions that do this.

```
unsigned long copy_from_user(void *to, const void *from, unsigned
long bytes_to_copy);
unsigned long copy_to_user(void *to, const void *from, unsigned
long bytes_to_copy);
```

As the names suggest, these functions copy data from or to the user space. The arguments are self explaining. They return the number of bytes not copied, so 0 in case of success. `int get_user(local, ptr);`
Get any data from the user-space address pointed to by ptr and put into local variable.

```
int put_user(datum, ptr);
```

Put the data to the user-space address pointed to by ptr.

The relation between `device_read()` and `copy_to_user()` is depicted in Fig.3 [3].

Input / Output Control. The functions `read` and `write` are sufficient to move or copy data between the module (kernel space) and the application (user space). However, physical devices typically have controllers which can be talked

[3] image taken from [2]

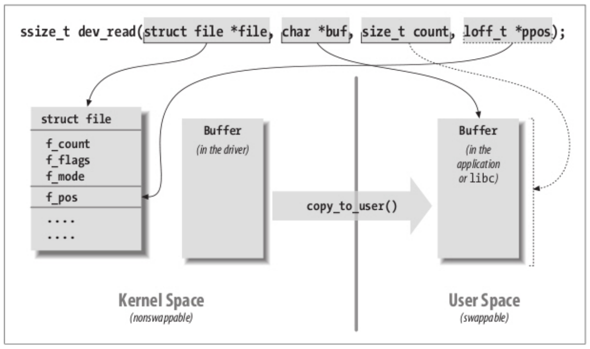

Fig. 3. Copy data from kernel to user space

to, eg for changing settings etc. To do this, we need a special function.
`int device_ioctl(struct inode *i, struct file *f, unsigned int cmd, unsigned long arg);`
With `ioctl` a command and an argument can be passed to the hardware. Examples for commands could be to set or get messages and the argument would be the address where to find or put the message. Commands are usually evaluated using switch statements. The user application must know which commands can be sent to the hardware.

Resources. Before the driver can access any hardware resources they must be requested. Depending on the type of hardware, different resources can or must be requested.

Resource	**Request**	**Release**
IO	`request_region`	`release_region`
Memory	`request_mem_region` `kmalloc`	`release_mem_region` `kfree`
IRQ	`request_irq`	`free_irq`
DMA	`request_dma`	`free_dma`

Requesting resources is usually done after registering the driver and before registering the device. Don't forget to clean up in the release and cleanup functions or else the driver won't unload and the system becomes unstable.

Hardware Detection and Probing. Today hardware detection is mostly done automatically, so hardware detection will not be covered in detail. Bus systems like pci and usb are hotpluggable, if a new device is connected the core driver recognizes the device and looks for a suitable driver. Every device must pass a unique identifier which is a combination of the vendor id and the device id that allows identification. The high level driver for that device must implement a `device_probe` function which identifies the correct hardware and reserves resources.

Depending on the bus system the device is connected to, there are different functions for probing, registering and unregistering the device and driver. So in addition to registering at the IO-Management as a char or block device the driver must also register the device with the corresponding bus subsystem.

4 Discussion

This has only been a very shallow introduction to device driver development. The topic is complex in nature and thus many things must remain unsaid. However, a lot of good material can be found on the web for further reading, including the two books on which this introduction is heavily based, see the reference section for links.

References

1. Jones, M. T.: Anatomy of the Linux kernel. http://www.ibm.com/developerworks/linux/library/l-linux-kernel/. (2007)
2. Corbet, J., Rubini, A. and Kroah-Hartman; G.: Linux Device Drivers. O'Reilly 2005, 3rd Ed. http://lwn.net/Kernel/LDD3/, http://www.oreilly.de/german/freebooks/linuxdrive2ger/book1.html
3. Quade, J., Kunst, E.-K.: Linux-Treiber entwickeln. dpunkt.Verlag 2004. http://ezs.kr.hs-niederrhein.de/TreiberBuch/html/
4. Salzman, P., Burian, M., Pomerantz, O.: The Linux Kernel Module Programming Guide. http://www.tldp.org/LDP/lkmpg/2.6/html/index.html. ver 2.6.4. 2007
5. http://en.wikipedia.org/wiki/Comparison_of_text_editors