Module 5

Device drivers

A device driver is a computer program that operates or controls a particular type of device that is attached to a computer. A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

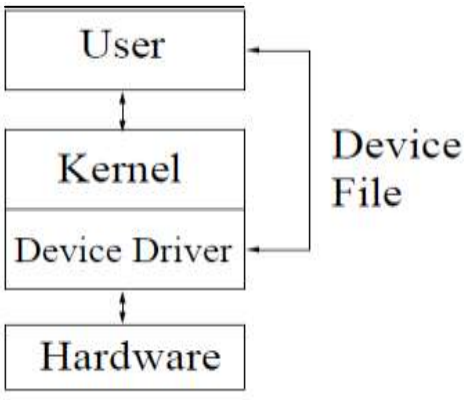
Device Control

Almost every system eventually maps to a physical device. With the exception of the processor, memory, and a few other enetities , any and all device control operations are performed by cod ethat is specific to the device being addressed . That code is code device driver.

The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive.

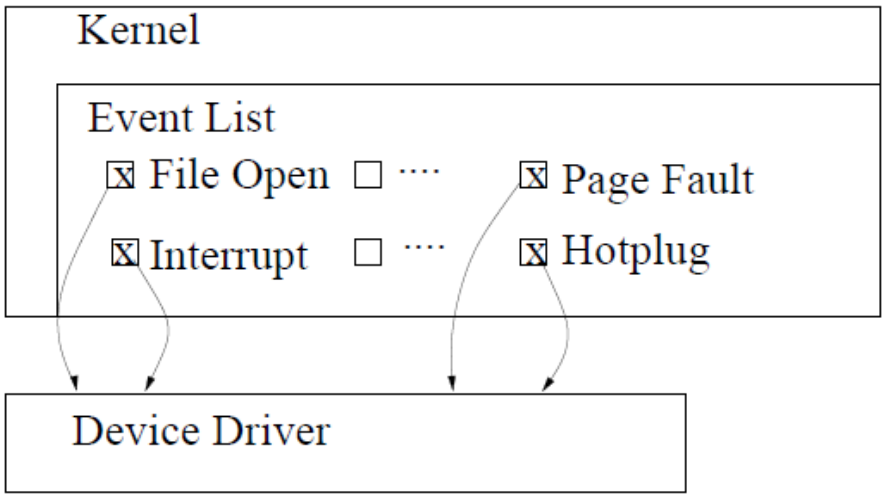
Anatomy of a Device Driver

A device driver has three sides :one side talks to the rest of the kernel ,one talks the hardware, and one talks to the user.



Kernel Interface of a Device Driver

In order to talk to the kernel, the driver registers with subsystems to respond to events. Such an event might be the opening of a file, a page fault, the plugging in of a new USB device, etc.



User Interface of a Device driver

Since Linux follows the UNIX model, and in UNIX everything is a file, users talk with device drivers through device files. Device files are a mechanism, supplied by the kernel, precisely for this direct User-Driver interface.

Characteristics of a device driver

There are many different device drivers in the Linux kernel (that is one of Linux's strengths) but they all share some common attributes:

kernel code

Device drivers are part of the kernel and, like other code within the kernel, if they go wrong they can seriously damage the system. A badly written driver may even crash the system, possibly corrupting file systems and losing data,

Kernel interfaces

Device drivers must provide a standard interface to the Linux kernel or to the subsystem that they are part of. For example, the terminal driver provides a file I/O interface to the Linux kernel and a SCSI device driver provides a SCSI device interface to the SCSI subsystem which, in turn, provides both file I/O and buffer cache interfaces to the kernel.

Kernel mechanisms and services

Device drivers make use of standard kernel services such as memory allocation, interrupt delivery and wait queues to operate,

Loadable

Most of the Linux device drivers can be loaded on demand as kernel modules when they are needed and unloaded when they are no longer being used. This makes the kernel very adaptable and efficient with the system's resources,

Configurable

Linux device drivers can be built into the kernel. Which devices are built is configurable when the kernel is compiled,

Dynamic

As the system boots and each device driver is initialized it looks for the hardware devices that it is controlling. It does not matter if the device being controlled by a particular device driver does not exist. In this case the device driver is simply redundant and causes no harm apart from occupying a little of the system's memory.

Classes of Devices and Modules (Types of Device Drivers)

The Unix way of looking at devices distinguishes between three device types. Each module usually implements one of these types, and thus is classifiable as a *char module*, a *block module*, or a *network module*. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability.

The three classes are the following:

Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the *open*, *close*,*read*, and *write* system calls. The text console (*/dev/console*) and the serial ports (*/dev/ttyS0* and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as */dev/tty1* and */dev/lp0*. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using *mmap* or *lseek*.

Block devices

Like char devices, block devices are accessed by filesystem nodes in the */dev* directory. A block device is something that can host a filesystem, such as a disk. In most Unix systems, a block device can be accessed only as multiples of a block, where a block is usually one kilobyte of data or another power of 2. Linux allows the application to read and write a block device like a char device -- it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node and the difference between them is transparent to the user. A block driver offers the kernel the same interface as a char driver, as well as an additional block-oriented interface that is invisible to the user or applications opening the */dev* entry points. That block interface, though, is essential to be able to *mount* a filesystem.

Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Though both Telnet and FTP connections are stream oriented, they transmit using the same device; the device doesn't see the individual streams, but only the data packets.

Not being a stream-oriented device, a network interface isn't easily mapped to a node in the filesystem, as */dev/tty1*is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of *read* and *write*, the kernel calls functions related to packet transmission.

Major and Minor Numbers

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the */dev*directory. Special files for char drivers are identified by a "c" in the first column of the output of *ls -l*. Block devices appear in */dev* as well, but they are identified by a "b." The focus of this chapter is on char devices, but much of the following information applies to block devices as well.

If you issue the *ls -l* command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor

device number for the particular device. The following listing shows a few devices as they appear on a typical system. Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

```
crw-rw-rw-   1 root     root       1,   3 Apr 11  2002 null
crw-------   1 root     root      10,   1 Apr 11  2002 psaux
crw-------   1 root     root       4,   1 Oct 28 03:04 tty1
crw-rw-rw-   1 root     tty        4,  64 Apr 11  2002 ttys0
crw-rw----   1 root     uucp       4,  65 Apr 11  2002 ttyS1
crw--w----   1 vcsa     tty        7,   1 Apr 11  2002 vcs1
crw--w----   1 vcsa     tty        7, 129 Apr 11  2002 vcsa1
crw-rw-rw-   1 root     root       1,   5 Apr 11  2002 zero
```

Traditionally, the major number identifies the driver associated with the device. For example, */dev/null* and */dev/zero* are both managed by driver 1, whereas virtual consoles and serial terminals are managed by driver 4; similarly, both *vcs1* and *vcsa1* devices are managed by driver 7. Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the one-major-one-driver principle.

The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written (as we will see below), you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.

The Internal Representation of Device Numbers

Within the kernel, the dev_t type (defined in <*linux/types.h*>) is used to hold device numbers—both the major and minor parts. As of Version 2.6.0 of the kernel, dev_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. Your code should, of course, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in <*linux/kdev_t.h*>.

To obtain the major or minor parts of a dev_t, use:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

If, instead, you have the major and minor numbers and need to turn them into a dev_t, use:

```
MKDEV(int major, int minor);
```

Note that the 2.6 kernel can accommodate a vast number of devices, while previous kernel versions were limited to 255 major and 255 minor numbers. One assumes that the wider range will be sufficient for quite some time, but the computing field is littered with erroneous assumptions of that nature. So you should expect that the format of dev_t could change again in the future; if you write your drivers carefully, however, these changes will not be a problem.

Character Device Drivers

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is **register_chrdev_region**, which is declared in <linux/fs.h>:

int register_chrdev_region(dev_t first, unsigned int count,char *name);

Here, first is the beginning device number of the range you would like to allocate. The minor number portion of first is often 0, but there is no requirement to that effect. count is the total number of contiguous device numbers you are requesting. Note that, if count is large, the range you request could spill over to the next major number; but everything will still work properly as long as the number range you request is available. Finally, name is the name of the device that should be associated with this number range; it will appear in /proc/devices and sysfs.

As with most kernel functions, the return value from register_chrdev_region will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

register_chrdev_region works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use; there is a constant effort within the Linux kernel development community to move over to the use of dynamically-allocated device numbers. The kernel will happily allocate a major number for you on the fly, but you must request this allocation by using a different function:

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

With this function, dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range. firstminor should be the requested first minor number to use; it is usually 0. The count and name parameters work like those given to request_chrdev_region. Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

void unregister_chrdev_region(dev_t first, unsigned int count);

The open Method

The *open* method is provided for a driver to do any initialization in preparation for later operations. In most drivers, *open* should perform the following tasks:

- Check for device-specific errors (such as device-not-ready or similar hardware problems)
- Initialize the device if it is being opened for the first time
- Update the f_op pointer, if necessary
- Allocate and fill any data structure to be put in filp->private_data

The first order of business, however, is usually to identify which device is being opened. Remember that the prototype for the *open* method is:

int (*open)(struct inode *inode, struct file *filp);

The release Method

The role of the *release* method is the reverse of *open*. Sometimes you'll find that the method implementation is called *device*_close instead of *device*_release. Either way, the device method should perform the following tasks:

- Deallocate anything that *open* allocated in filp->private_data
- Shut down the device on last close

The basic form of *scull* has no hardware to shut down, so the code required is minimal:The other flavors of the device are closed by different functions because *scull_open* substituted a different filp->f_op for each device. We'll discuss these as we introduce each flavor.

```
int scull_release(struct inode *inode, struct file *filp)
{
   return 0;
}
```

Block Device Drivers

The first step taken by most block drivers is to register themselves with the kernel. The function for this task is **register_blkdev** (which is declared in <linux/fs.h>):

int register_blkdev(unsigned int major, const char *name);

The arguments are the major number that your device will be using and the associated name (which the kernel will display in /proc/devices). If major is passed as 0, the kernel allocates a new major number and returns it to the caller. As always, a negative return value from register_blkdev indicates that an error has occurred.

The corresponding function for canceling a block driver registration is:

int unregister_blkdev(unsigned int major, const char *name);

Here, the arguments must match those passed to register_blkdev, or the function returns -EINVAL and not unregister anything.

Read/ Write Operations

2 methods –

- Polling
- Interrupt based

- *Direct I/O with polling* – the device management software polls the device controller status register to detect completion of the operation; device management is implemented wholly in the *device driver*, if interrupts are not used

- *Interrupt driven direct I/O* – interrupts simplify the software's responsibility for detecting operation completion; device management is implemented through the interaction of a *device driver* and interrupt routine

Pooling

CPU is used(eg. by a device driver) , in a program code loop, to continuously check a device to see if it is ready to accept data or commands , or produce output

Loop (until device ready)

Check device

End loop

CPU is tied up in communication with device until I/O operation is done. No other useful work can be accomplished by the CPU. Typically one CPU in the system , but many devices.

Interrupt based

- I/O via software polling can be inefficient
- Too much time spent by CPU waiting for devices.
- ie., amount of time between I/O operations is large
- Only one CPU; polling ties up this resources
- Better to design differently
- Since CPU is rare resource, we need to keep it busy processing jobs
- Handle I/O as different type of event
- CPU has wire called an interrupt request line
- Instead of I/o controller setting status [busy] =0 control[command –ready]=0, I/O controller sets the interrupt request line to 1
- CPU, as part of instruction processing cycle, automatically checks interrupt request line.
- If it is set to 1, this generates an interrupt.