Introduction to Drivers

Zilogic Systems

1. Device Abstraction

One of the objectives of an OS is to abstract away the differences in the hardware from the applications running in the OS. An application using the mouse for example, should not be bothered about whether it is a USB mouse, wireless mouse, optical mouse, or whether the mouse has legs!!! The application should not be bothered what chip is located within the mouse, whether it is from IBM, or from Microsoft.

Each device has a software called a driver, that knows about the details of the device hardware. These drivers provide a consistent interface towards the application. For example if you use the `read()` function to read from a Microsoft USB keyboard, you can use the same `read()` function to read from an IBM PS/2 keyboard. Here `read()` is the consistent interface towards the application. And there could be different `read()` implementations provided by different drivers.

2. Device Abstraction in Linux

The designers of Unix observed the similarity between accessing I/O devices and acessing files. Data from I/O devices can be treated as a stream of bytes, just like files.

In Unix, everything is a file! Mouse is a file, printer is a file, harddisk is a file ... Files can be broadly classified as plain files, directories and special or device file. Physical devices are accessed through device files. Device files are located in `/dev`.

Here are some device files and the associated hardware type.

```
/dev/hda, /dev/hdb - IDE Hardisk, CDROM
/dev/sda, /dev/sdb - SCSCI or SATA Hardisk
/dev/fd0           - Floppy Drive
/dev/audio         - Sound Card
/dev/input/mice    - Mouse
/dev/mem           - RAM
```

Reading and writing to the device files, will result in reading and writing to the devices. Let's try out. Read the mouse device using the following command.

```
# cat /dev/input/mice
```

Move the mouse, you should be able to see wierd characters getting dumped on the screen. The only thing we can make out now is that there is some correlation between the mouse movements and the dump from the device file. We will write a small program to interpret these wierd characters.

For each mouse event 3 bytes are produced. We will focus on the first byte for now. The first byte contains bits to indicate which of the 3 mouse buttons was pressed.

```
Bit 0 - Left Button
Bit 1 - Right Button
Bit 2 - Middle Button
```

The following program prints which button was clicked, using the data read from `/dev/input/mice`.

Reading Mouse Events.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <error.h>
#include <errno.h>

int main()
{
        int fd;
        char buf[3];
        int ret;

        fd = open("/dev/input/mice", O_RDONLY);
        if (fd == -1) {
                error(1, errno, "error opening mouse device");
        }

        while (1) {
                ret = read(fd, buf, sizeof(buf));
                if (ret == -1) {
                        error(1, errno, "error reading mouse device");
                }

                if (buf[0] & 0x01) {
                        printf("Left Button Clicked\n");
                } else if (buf[0] & 0x02) {
                        printf("Right Button Clicked\n");
                } else if (buf[0] & 0x04) {
                        printf("Middle Button Clicked\n");
                }
        }
}
```

Now, how does this work? How does the operating system know that mouse events should be reported when `/dev/input/mice` is read? Below is the output of `ls` on device files and ordinary files.

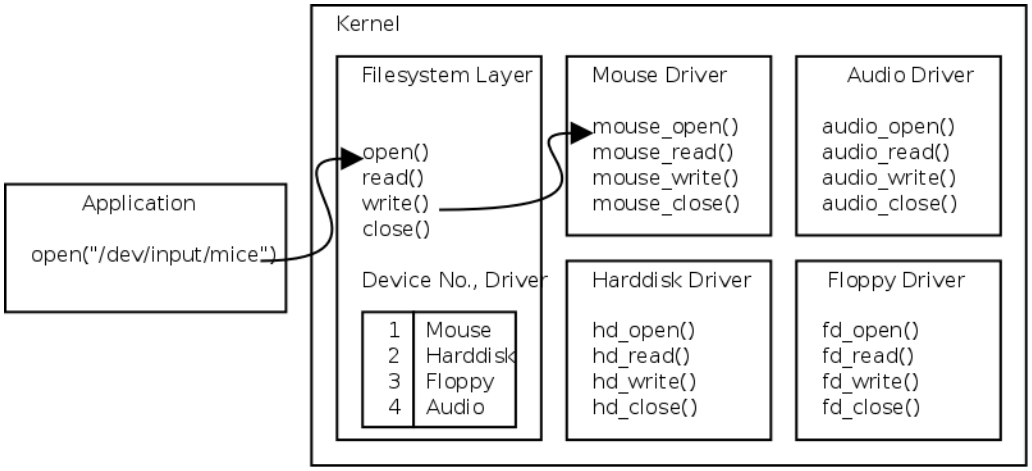
```
[~]$ ls -alh /dev/input/mice /dev/audio
crw-rw---- 1 root audio 14,  4 2007-08-21 10:02 /dev/audio
crw-rw---- 1 root root  13, 63 2007-08-21 10:01 /dev/input/mice

[~]$ ls -alh *.pdf
-rw-r--r-- 1 vijay vijay  323K 2007-07-28 21:40 lcd-datasheet.pdf
-rw-r--r-- 1 vijay vijay 1023K 2007-08-19 18:05 radmind-solaris.pdf
-rw-r--r-- 1 vijay vijay  3.2M 2007-08-11 09:40 thinkpad.pdf
```

If you are very cautious, you will notice that the device files have the type flag set to `c`. `c` stands for character device. We will come to the details of what a character device is shortly. But apart from the file type, you will also notice that the plain files have a size specified in column 5, but the device files have 2 no.s printed instead. These numbers together represent a single no. called device no.

The device no. associates a device file with a device driver.

When you open a device file using the `open()` system call, the file system layer in the kernel will find out from the type flag that this is a device file. The file system layer then uses the device no. to find the appropriate driver and will pass on handling of the system call to the driver. The driver implements the system call by appropriately accessing the device, using port I/O or memory mapped I/O instructions.

Figure 1. Driver access through device file



3. Character Drivers and Block Drivers

Depending upon the interface provided, device drivers are broadly classified as character drivers, block drivers and network drivers. Character drivers provide an interface through which device data is accessed as a stream of bytes. Examples are mouse driver and audio driver. Block drivers provide an interface through which blocks of data can be addressed and accessed randomly. Examples are harddisk driver and floppy driver. Network drivers will be dealt with later.

Figure 2. Character and block driver application interface

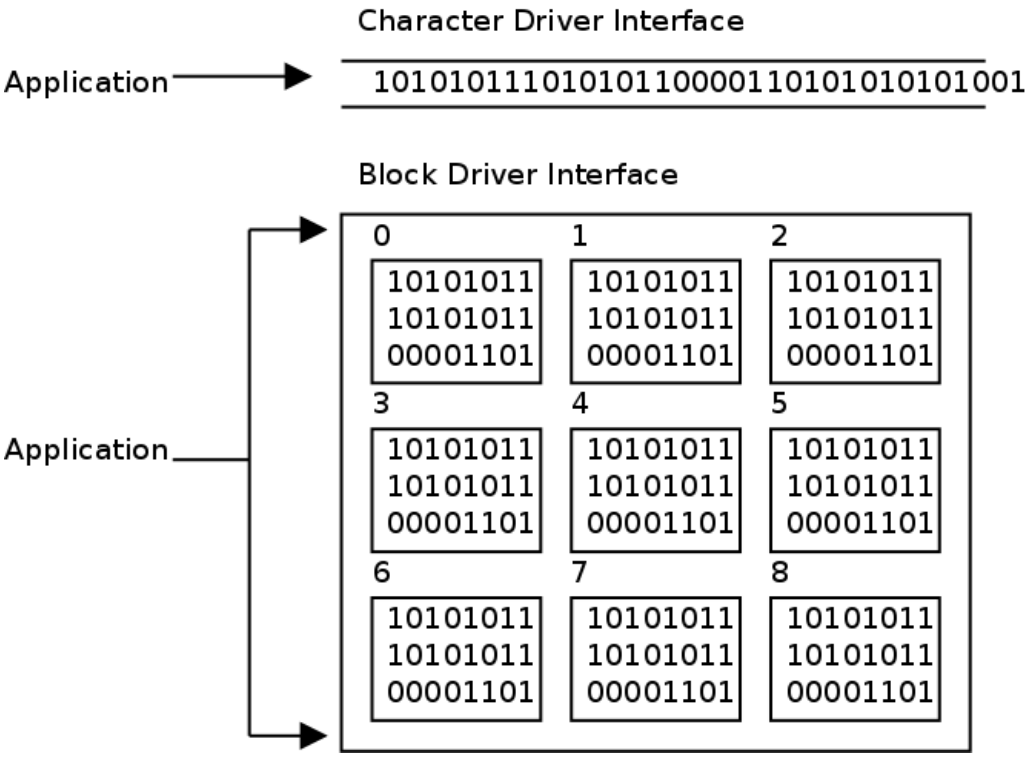
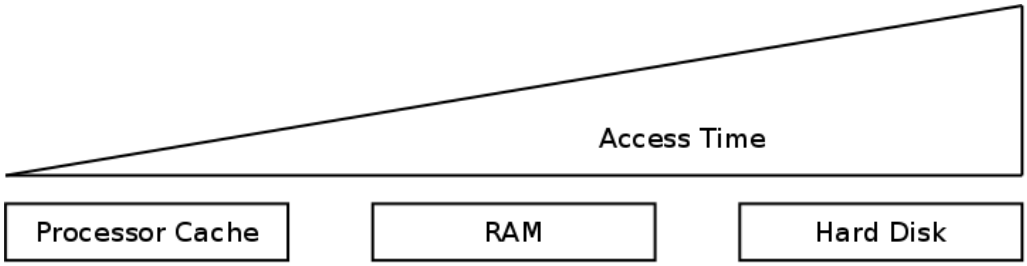


Figure 3. Need for buffer cache



Buffering	Buffer cache is a cache implemented in software that helps speed up disk access by holding recently accessed data in RAM. Block drivers write data from the buffer cache into disk, and read data from disk into the buffer cache. Character drivers do not use the buffer cache.
Seeking	Character drivers may or may not be seekable. Seeking does not make sense for devices like mouse, audio. Block drivers must be seekable, so that any sector on a harddisk can be accessed at any time.
Size	Character devices do not have a size. Block devices have a fixed size.
Filesystem	Because of its characteristics, a filesystem can be used on top of a block driver. It is not possible to use filesystem on top of a character driver.
Device File Type	Both block drivers and character drivers can be accessed through device files. But the device files have different types.

```
[~]$ ls -al /dev/audio /dev/hda
crw-rw---- 1 root audio 14, 4 2007-08-21 10:02 /dev/audio
brw-rw---- 1 root disk   3, 0 2007-08-21 10:01 /dev/hda
```

Block drivers and character drivers have separate device no. space. Meaning that a block driver and character driver can have same device no. and refer to different drivers. Internally, the kernel maintains two device no. to driver mapping tables, one for character drivers and another for block drivers.

```
[~]$ ls -al /dev/hda /dev/ttyp0
brw-rw---- 1 root disk 3, 0 2007-08-21 10:01 /dev/hda
crw-rw-rw- 1 root tty  3, 0 2007-08-21 10:01 /dev/ttyp0
```

4. More fun with device files

We will try accessing other devices through device files, to get familiar with device file interface.

4.1. Accessing the Audio Device

As mentioned before audio devices are also character devices, and can be accessed through `/dev/audio`. The `.au` file encoded in 8-bit mu-law with a sampling rate of 8kHz, exactly matches the default format required by `/dev/audio`. (If you are not very familiar with audio file formats, just understand these are bunch of parameters required for proper audio recording and playback.) To play such a `.au` file you can directly do

```
[~]$ cat mysound.au > /dev/audio
```

If you do not have such a file you can use the SOund eXchange utility `sox`, to convert any MP3 file to the required format, as shown below.

```
[~]$ sox ~/sivaji/athiradee.mp3 -b -u -c 1 -r 8000 athiradee.au
```

Here `-b` specifies 8-bit, `-u` specifies mu-law, `-c` specifies mono (single-channel), `-r 8000` specifies 8kHz sampling rate.

4.2. Accessing the Hard Disk

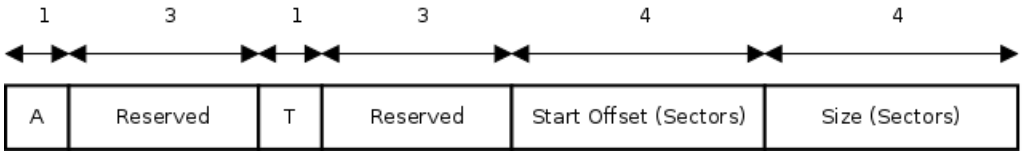
The hard disk is usually accessed through the file system. But it is also possible to raw access the hard disk through the device files `/dev/hda`, `/dev/hdb`, etc. The sectors in the hard disk appear to be layed out sequentially when accessed through `/dev/hda`.

To gain familiarity with the device file, we will write a program to print the partition table of the harddisk.

Background

- Each sector is of 512 bytes.
- The first sector of the hard disk is called the MBR.
- The MBR contains a part of the boot loader, and the partition table.
- The first 446 bytes contains the boot loader code.
- The next 64 bytes contains the partition table.
- The last 2 bytes contain magic nos. `55`, `AA`.
- The partition table has four records, each of size 16 bytes.
- Each record has the following format.

Figure 4. Partition table entry format



The program to print the partition table is shown below

Reading the partition table.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <error.h>
#include <errno.h>
#include <stdint.h>

struct part_record {
        uint8_t active;
        uint8_t discard1[3];
        uint8_t type;
        uint8_t discard2[3];
        uint32_t start;
        uint32_t size;
} __attribute__((packed));

enum {
```

```
        KB = 1000,
        MB = 1000 * KB,
        GB = 1000 * MB
};

void print_human_readable(uint64_t val)
{
        if (val > GB) {
                printf("%.1f GB", ((double)val)/GB);
        } else if (val > MB) {
                printf("%.1f MB", ((double)val)/MB);
        } else if (val > KB) {
                printf("%.1f KB", ((double)val)/KB);
        } else {
                printf("%lld bytes", val);
        }
}

int main()
{
        int fd;
        off_t offset;
        ssize_t size;
        char buf[64];
        struct part_record *pr;
        int i;

        fd = open("/dev/sda", O_RDONLY);
        if (fd == -1) {
                error(1, errno, "error opening harddisk device file");
        }

        offset = lseek(fd, 446, SEEK_SET);
        if (offset == -1) {
                error(1, errno, "error seeking harddisk");
        }

        size = read(fd, buf, sizeof(buf));
        if (size == -1) {
                error(1, errno, "error reading harddisk");
        }

        pr = (struct part_record *) buf;

        for (i = 0; i < 4; i++) {
                printf("%d\t", i+1);
                print_human_readable(pr[i].start * 512LL);
                printf("\t\t");
                print_human_readable(pr[i].size * 512LL);
                printf("\n");
        }

        return 0;
```

```
}
```

To verify the program, the output can be compared with the output of `parted`.

5. Controlling Devices

We have so far seen that, reading and writing to device files translate really well to device I/O. But there are limits to the abstraction. It is not possible to model all device operations as reading and writing to files. Sometimes it is necessary to control and configure the device, and the file read and write abstraction does not work well. Examples of such control operations are ejecting a CDROM drive, or controlling the volume of the sound card.

Linux has a separate system call called `ioctl()` to deal with this. The system call has the following declaration.

```
int ioctl(int fd, int request, ...)
```

`fd` is the device file fd, on which the ioctl operation is to be performed.

`request` is the operation to be performed. It is usually a macro defined in the kernel header files.

A third optional argument specifies a pointer which can be used to send or receive data associated with the request.

An example of `ioctl()` usage is CDROM eject. The macro name for the request is `CDROMEJECT`. It is defined in `linux/cdrom.h`. `CDROMEJECT` does not use the third argument.

The following program shows the use of `ioctl()` to perform a CDROM eject.

Ejecting the CD-ROM drive.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <error.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <linux/cdrom.h>

int main()
{
        int fd;
        int ret;

        fd = open("/dev/sr0", O_RDONLY | O_NONBLOCK);
        if (fd == -1) {
                error(1, errno, "error opening CDROM device file");
        }

        ret = ioctl(fd, CDROMEJECT);
        if (ret == -1) {
                error(1, errno, "error eject CDROM");
        }

        close(fd);
}
```

If `O_NONBLOCK` is not specified then opening the device file will fail, if there is no CD-ROM in the drive.

Some CD-ROM drives support programmatic closing of the CD-ROM drive. The `ioctl()` that is to be use is `CDROMCLOSETRAY`.

We will look at another example with ioctls. Every PC has an RTC (Real Time Clock) built-in into it, to maintain the system time, even when the system is powered down. The RTC is backed up by a battery, and when the system powers up it reads the current time from the RTC.

The date/time is read from the RTC using ioctls. The device file for the RTC is `/dev/rtc0`. The ioctl macro name is `RTC_RD_TIME` and is defined in `linux/rtc.h`. The third argument is passed a pointer to RTC time structure, into which driver fills in the current time.

Reading time from the RTC.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/rtc.h>

#include <error.h>
#include <errno.h>

int main()
{
        int fd;
        struct rtc_time rtc_tm;
        int ret;

        fd = open("/dev/rtc0", O_RDONLY);
        if (fd == -1) {
                error(1, errno, "error opening RTC");
        }

        ret = ioctl(fd, RTC_RD_TIME, &rtc_tm);
        if (ret == -1) {
                error(1, errno, "error reading time");
        }

        printf("Time: %02d:%02d:%02d\n", rtc_tm.tm_hour,
               rtc_tm.tm_min, rtc_tm.tm_sec);

        return 0;
}
```

So far we have seen two examples one without the 3rd argument, another with the 3rd argument to retrieve information from the device. We will now look at another example that is used to transmit information through the 3rd argument.

We already saw how audio files can be played through the device file. But what if we want to control the volume. This can be done through ioctls. There is a special device file for such audio control operations `/dev/mixer`. The ioctl macros to be used are `SOUND_MIXER_READ_VOLUME` and `SOUND_MIXER_WRITE_VOLUME`. The 3rd argument passed is a pointer to an integer. Byte 0 (LSB)

represents the left volume, and byte 1 represents the right volume. The macros are defined in `linux/soundcard.h`.

Setting the volume of the sound card.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/soundcard.h>
#include <error.h>
#include <errno.h>

int main()
{
        int fd;
        unsigned int volume;
        int left;
        int right;
        int ret;

        fd = open("/dev/mixer", O_RDWR);
        if (fd == -1) {
                error(1, errno, "error opening mixer");
        }

        ret = ioctl(fd, SOUND_MIXER_READ_VOLUME, &volume);
        if (ret == -1) {
                error(1, errno, "error reading volume");
        }

        left = volume & 0xff;
        right = (volume & 0xff00) >> 8;

        left += 20;
        right += 20;

        volume = left | (right << 8);

        ret = ioctl(fd, SOUND_MIXER_WRITE_VOLUME, &volume);
        if (ret == -1) {
                error(1, errno, "error writing volume");
        }

        return 0;
}
```

In this session we have described the role of a driver and the Unix abstraction of devices. We have accessed various devices using device files. We have also seen the need for a control interface which does not fit well into the file abstraction. And that the control interface is implemented using ioctls.

6. Further Reading

- Linux: Rute User's Tutorial and Exposition: Device Files http://rute.2038bug.com/node21.html.gz

- Linux Kernel Hackers' Guide: Device Drivers - http://www.tldp.org/LDP/khg/HyperNews/get/devices/devices.html