Lab 09 - Device Drivers

Objectives

1. Lean about the role of a device driver.
2. Learn how devices drivers are organized under Linux.
3. Implement a simple Linux device driver for the keyboard LEDs.

Pre-reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

Overview

The various hardware components and peripherals that an operating system communicates with are of a great variety and different communication protocols must be employed when interacting with each one of them (keyboard, display, mouse disk, serial ports, parallel ports, network card...). In all modern operating systems the interaction with hardware devices is handled by a part of the operating system called a device driver. The device driver encapsulates the particularities of a specific hardware device and exposes a canonical set of operations through a specific and standardized interface.

In this lab you will get familiar with what device drivers are and how they are organized and implemented under Linux. At the end you will put all this knowledge to use by implementing a device driver that controls the LEDs on the keyboard.

Device Drivers

A device driver is a collection of functions and data structures whose purpose is to implement a simple interface for managing a device. Device drivers run as part of the kernel and aren't directly accessible to user processes. Linux provides a mechanism though which processes can communicate with the device drivers and through them with the hardware devices: the device files. The device files are objects that appear in the file system and look just like ordinary files: they can be opened, read from and written to. They are not however normal files, the data that is written to a device file is not stored on the disk (as it would be in case of a normal file), but it is communicated to the appropriate device driver, which in turn communicates it to the underlying hardware. There are two types of device files depending on the type of hardware they manage:

- **Character device files** – represent hardware devices that read or write serial streams of data bytes, one byte at a time. Example of character devices: serial and parallel ports, sound cards, tape drives, terminals.

- **Block device files** – represent hardware devices that read and write data in blocks. Unlike character devices they provide random access to the data stored on the device. Hard drives are a good example of block devices.

The devices files are stored by convention in the /dev directory, however that is just a convention, a device file can be stored in any directory; it can be copied and moved around just as a normal file. To see the device files present at one time on a system just list the contents of the /dev directory:

```
# ls -l /dev
...
brw-r----- 1 root disk    3,   0 Nov 19 10:20 hda
brw-r----- 1 root disk    3,   1 Nov 19 10:20 hda1
brw-r----- 1 root disk    3,   2 Nov 19 10:20 hda2
...
crw-rw---- 1 root uucp    4,  64 Nov 19 10:20 ttyS0
crw-rw---- 1 root uucp    4,  65 Nov 19 10:20 ttyS1
...
crw-rw---- 1 root audio  14,   3 Dec  2 00:31 dsp
crw-rw---- 1 root audio  14,   4 Dec  2 00:31 audio
...
crw-rw-rw- 1 root root    1,   8 Nov 19 10:20 random
```

In the above example the hda device file represents the hard drive, hda1 and hda2 represent the first and the second partition, ttyS0 and ttyS1 represent the first and second serial port, dsp and audio represent the sound card, random is virtual device that when read returns random bytes.

The device files provide a very powerful way of working with the various hardware devices just as if they were normal files. For example the following command:

```
# dd if=/dev/hda of=mbr.bin bs=512 count=1
```

will read the first 512 bytes from the beginning of the hard drive (the master boot record) and store them in the mbr.bin file (dd is a command that just copies part of one file into another file).

The command:

```
# dd if=/dev/zero of=/dev/zero
```

will write zeros over your entire hard drive, effectively erasing everything you have on the hard drive (so don't try it!).

You can also read and write to device files from programs just as if they were normal files. So if you want to write a program that works physically with the hard drive you can just read and write to /dev/hda from within the program.

One question that remains is how does the operating system knows which file is associated with which device driver. This association is not done through the name of the device file, but through a set of two numbers called device numbers. Each device driver is referenced by a major number and a minor number. The major number represents the class of devices that the driver can manage. For example floppy disks have the major number 2, hard disk have the major number 3, serial ports have the major number 4, and sound cards have the major number 14. The minor number references a specific device of a particular class (major number).

Each device file also has two numbers associated with it (you can see this from the output the “ls -l /dev” command). Through these numbers a device file is associated with a device driver having the same device numbers. A new device file can be created using the mknod command:

```
# mknod /dev/<name> <type> <major_number> <minor_number>
```

where <name> represents the name of the device file, <type> is c for character devices or b for block devices, <major_number> and <minor_number> are the major and the minor numbers of the device driver this device file is associated with. Although a device file with any major and minor numbers can be created, it is only useful if there exists a corresponding device driver with the same numbers present inside the kernel.

Kernel Modules

Under Linux, a device driver can be either linked statically inside the kernel or compiled as kernel modules. So what exactly is a kernel module? A kernel module is part of the kernel that can be loaded and unloaded from kernel on demand. This is much more convenient than the alternative of being statically linked inside the kernel, because this way new functionality can be added to the kernel only when needed (when a new device is attached to the computer for example), without the need to reboot the system.

As you already know, a normal program begins with the main() function, executes this function and terminates when the main() function returns. Kernel modules work a bit differently. A kernel modules begins with the execution of the function init_module() when the module is first loaded into the kernel. This function is the entry point for any module and it tells the kernel what functionality the module provides, so that the kernel will know to run the module's functions when they are needed. Once the function init_module() returns, the modules does nothing until one of its functions is called by the kernel. Every modules also has an exit function called cleanup_module(). This function undoes everything that init_module() did, it unregisters the module's functionality from the kernel.

The simplest kernel modules (one that doesn't really do anything useful) looks like this:

File: hello.c

```c
/* The simplest kernel module. */
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */

MODULE_LICENSE("GPL");

int init_module(void)
{
      printk(KERN_INFO "Hello world.\n");

      /* A non 0 return means init_module failed; module can't be loaded. */
      return 0;
}

void cleanup_module(void)
{
      printk(KERN_INFO "Goodbye world.\n");
}
```

Kernel modules are also different from normal programs through the external functions they can call. In normal programs you typically use many functions that you didn't actually written, but they are part of some external library. A typical example is the printf() function used to display information on the console, function that is part of the C standard library. Since kernel modules execute in kernel space (or 'superuser mode') all these library functions cannot be called. The only external functions you can call from within a kernel module are those provided by the kernel. To see all the symbols exported by the kernel take a look at /proc/kallsyms. In the above example you may notice the function printk() that is very similar to printf(), but instead of printing the message on the console it sends it to the system log (you may see it by looking at the /var/log/messages file).

Kernel modules need to be compiled differently than regular programs. Fortunately you don't need to care much about how a kernel module must be compiled; the kernel build mechanism is also capable to compile external modules. All you need is a make file that defines what the module name is and calls the kernel build mechanism.   Such a make file looks like this:

File: makefile

```makefile
MODULE_NAME=hello

obj-m := ${MODULE_NAME}.o

all:
      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

After creating the make file you just need to type make in order to compile the module. When the compilation process finishes you will notice that several new files have just appeared in the directory with the modules source file. One of these files, hello.ko, is the finished compiled module, all the rest are just temporary files created by the compilation process.

To insert the newly compiled module into the running kernel you must run the following command (as root):

```
# insmod hello.ko
```

After the module was inserted you may see that the init_module() function was executed by examining the system log file /var/log/messages or the output of the dmesg command:

```
# tail /var/log/messages
# dmesg | tail
```

To see all the modules loaded into the kernel at one particular time you may use the lsmod command. If the insertion of the hello.ko module was successful you should see it in this list.

To remove a module from the kernel you must use the rmmod command:

```
# rmmod hello
```

A great source of information on how to write kernel modules is the 'Linux Kernel Modules Programming Guide' available at the following address:

http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html

Kernel Modules as Device Drivers

As we already mentioned, device drivers can be implemented as kernel modules so that they may be easily inserted and removed from a running kernel whenever they are needed. We also know by now that a user-space program communicates with a device driver through a device file and the association between a device file and a device driver is done by matching the major number their major numbers.

One of the first thing that every device driver must do when is first loaded is to register itself with the kernel by specifying which is the major number and the operations the device driver supports. This is accomplished for character devices by calling the register_chrdev() function (there is a similar register_blkdev() function for block devices, however writing a block device driver is more complicated than writing a character device driver and we will not discuss it in this lab):

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
```

The register_chrdev() function takes 3 arguments: unsigned int major is the major number you want to request (or 0 if you want it to be automatically assigned), const char *name is the name of the device as it will appear is /proc/devices and struct file_operations *fops is pointer to a file_operations structure indicating which operations the driver can handle.

This structure holds the addresses of the module functions that perform various operations supported by the module, such as read, write, flush... some operations might not be implemented by the driver (for example a driver for a temperature sensor might only need to be read, no written to) in which case the corresponding entries will be set to NULL. The definition of the file_operations structure looks like this:

```
struct file_operations {
       struct module *owner;
        loff_t(*llseek) (struct file *, loff_t, int);
        ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
        ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
       int (*readdir) (struct file *, void *, filldir_t);
       unsigned int (*poll) (struct file *, struct poll_table_struct *);
       int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
       int (*mmap) (struct file *, struct vm_area_struct *);
       int (*open) (struct inode *, struct file *);
       int (*flush) (struct file *);
       int (*release) (struct inode *, struct file *);
       int (*fsync) (struct file *, struct dentry *, int datasync);
       int (*aio_fsync) (struct kiocb *, int datasync);
       int (*fasync) (int, struct file *, int);
       int (*lock) (struct file *, int, struct file_lock *);
        ssize_t(*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t(*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void __user *);
        ssize_t(*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
       unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                                                            unsigned long, unsigned long,
                                                            unsigned long);
};
```

As already mentioned, not all the fields of this structure need to be initialized, only those corresponding to operations supported by the device driver. An initialization of this structure might look like this:

```
struct file_operations fops = {
            .read = device_read,
            .write = device_write,
            .open = device_open,
            .release = device_release
};
```

Below is presented the source code of a fully functional character device driver. This device driver doesn't actually control any underlying hardware device; it just sends back the number of time it has been read. The source of this driver is taken from the 'Linux Kernel Module Programming Guide' mentioned above.

File: chardev.c

```
/*
 *    chardev.c: Creates a read-only char device that says how many times
 *    you've read from the dev file
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>      /* for put_user */

/*
 *    Prototypes - this would normally go in a .h file
 */
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 *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices
*/
#define BUF_LEN 80            /* Max length of the message from the device */

/*
 * Global variables are declared as static, so are global within the file.
 */

static int Major;       /* Major number assigned to our device driver */
static int Device_Open = 0;   /* Is device open?
                                          * Used to prevent multiple access to
device */
static char msg[BUF_LEN];     /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops = {
      .read = device_read,
      .write = device_write,
      .open = device_open,
      .release = device_release
};

/*
 * This function is called when the module is loaded
 */
int init_module(void)
{
            Major = register_chrdev(0, DEVICE_NAME, &fops);

      if (Major < 0) {
        printk(KERN_ALERT "Registering char device failed with %d\n", Major);
        return Major;
      }

      printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
      printk(KERN_INFO "the driver, create a dev file with\n");
      printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
      printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
      printk(KERN_INFO "the device file.\n");
      printk(KERN_INFO "Remove the device file and module when done.\n");

      return SUCCESS;
}

/*
 * This function is called when the module is unloaded
 */
void cleanup_module(void)
{
      /*
       * Unregister the device
       */
      unregister_chrdev(Major, DEVICE_NAME);
}

/*
 * Methods
 */

/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
      static int counter = 0;

      if (Device_Open)
            return -EBUSY;

      Device_Open++;
      sprintf(msg, "I already told you %d times Hello world!\n", counter++);
      msg_Ptr = msg;
      try_module_get(THIS_MODULE);

      return SUCCESS;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
      Device_Open--;          /* We're now ready for our next caller */

      /*
       * Decrement the usage count, or else once you opened the file, you'll
       * never get get rid of the module.
       */
      module_put(THIS_MODULE);

      return 0;
}
```

```
/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h
      */
                     char *buffer,  /* buffer to fill with data */
                     size_t length, /* length of the buffer       */
                     loff_t * offset)
{
      /*
       * Number of bytes actually written to the buffer
       */
      int bytes_read = 0;

      /*
       * If we're at the end of the message,
       * return 0 signifying end of file
       */
      if (*msg_Ptr == 0)
            return 0;

      /*
       * Actually put the data into the buffer
       */
      while (length && *msg_Ptr) {

            /*
             * The buffer is in the user data segment, not the kernel
             * segment so "*" assignment won't work.  We have to use
             * put_user which copies data from the kernel data segment to
             * the user data segment.
             */
            put_user(*(msg_Ptr++), buffer++);

            length--;
            bytes_read++;
      }

      /*
       * Most read functions return the number of bytes put into the buffer
       */
      return bytes_read;
}

/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
      printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
      return -EINVAL;
}
```

There are several things that should catch your attention when looking over this source:

- The first one is the use of the functions try_module_get(THIS_MODULE) and module_put(THIS_MODULE). The role of this functions is to prevent the removal of the module while it is in use. These functions work is by incrementing and decrementing a counter internal to the kernel and when this counter not zero the module cannot be removed from the kernel. You may see the value of this counter by looking at the third field in the corresponding line in /proc/devices or with the lsmod command.

- The second observation is the call put_user() in the device_read() function (when a read operation is performed). The put_user() must be used when transferring some data between kernel-space and user-space. The buffer address cannot be used directly because it is an user-space address from the address space of the program reading from the device file, not from the kernel space. The appropriate address translation has to be performed when accessing an user-space address (remember virtual memory from class!) and this address translation is not necessary for kernel-space addresses.

To compile the above code you may use the make file shown above, you only need to update the name of the kernel module in the make file. After compiling the module you may insert it into the kernel with the command insmod chardev.ko (you must be logged in as root to be able to insert the module). To verify that the module was successfully inserted you may check the system log file /var/log/messages or the output of the dmesg command. From the log file or by looking inside /proc/devices you can find out the major number assigned to the module and create a device file for this module:

```
#  mknod /dev/chardev c 252 0
```

Once the device file is created you may read from the device driver like this:

```
# cat /dev/chardev
I already told you 0 times Hello world!
# cat /dev/chardev
I already told you 1 times Hello world!
# cat /dev/chardev
I already told you 2 times Hello world!
```

If you try to write to the device driver you will get an error message because this device driver doesn't support the write operation;

```
# echo "Hello" > /dev/chardev
bash: echo: write error: Invalid argument
```

Congratulations, you have just created your first Linux device driver!

Controlling the Keyboard LEDs

The keyboard LEDs can be controlled directly through IO ports. The IO ports are one of the mechanisms through which the CPU communicates with almost all the other controllers and peripherals inside a computer. Each port has an address through which it can be read or written to (not unlike a location of memory). Each device inside a computer has a allocated a range of IO ports that may be used to communicate with it. To see which ports are allocated for various devices you can take a look at /proc/ioports. If you examine this file you will notice that the keyboard is using the range 0x60-0x6f. For our problem, to control the keyboard LEDs, we will only need to use only port 0x60.

To control the keyboard LEDs one must perform the following actions:

- Write command 0xED to port 0x60. This instructs the keyboard controller that a command on how to set the LEDs will follow.
- Keyboard controller sends on port 0x60 the acknowledgment 0xFA when it's ready to accept the command.
- Keyboard controller expects a byte value on port 0x60 with the configuration of the keyboard LEDs:
    - bit 0 : scroll lock
    - bit 1 : num lock
    - bit 2 : caps lock
    - bits 3-7 : ignored

There are specialized CPU instructions for accessing the IO ports. However these instructions are privileged instructions that can only be used when the CPU is in superuser mode. What this means is that a normal program is not allowed to access the IO ports, it must be done from kernel space, for example form within a kernel module. To read from and write to the IO ports you may use the following functions:

- outb(value,port) – Write a byte values to a port
- inb(port) – Read a byte value from a port

There are similar functions for reading and writing word and long values to ports (outw, outl, inw, inl).

Using these functions the code that controls the keyboard LEDs should look something like this:

```
retries = 5;
timeout = 1000;
state = ...;       // the LEDs configuration
outb(0xed,0x60);  // tell the keyboard we want to set the LEDs configuration
udelay(timeout);
while (retries!=0 && inb(0x60)!=0xfa) { // wait for the keyboard controller
      retries--;
      udelay(timeout);
}
if (retries!=0) {  //check is keyboard is ready to accept command
      outb(state,0x60);
}
```

Procedure

Part A

Write a new device driver that controls the LEDs on a typical keyboard: the num-lock, caps-lock and scroll-lock LEDs. The device driver must be implemented as a kernel module which registers itself as a character device when loaded. You must be able to interact with the driver through a new device file, /dev/leds, like this: writing 1 to this device file should light up the num-lock led, 2 should light up the caps-lock led, 3 should light up the scroll-lock led, 12 should light up both num-lock and caps lock led and so on.

Example:

Command	Num Lock	Caps Lock	Scroll lock
`echo 1 > /dev/leds`	on	off	off
`echo 123 > /dev/leds`	on	on	on
`echo 32 > /dev/leds`	off	on	on
`echo "" > /dev/leds`	off	off	off

Part B:

Write a user-space program that controls the keyboard LEDs by using the device driver you wrote in part A. The program will need to write to the /dev/leds device file the corresponding strings such as the keyboard LEDs are turn on and off in some predetermined sequence. It's up to you to choose the sequence, it can be something as simple as a circular order, a bit counter or it can be something fancier, like lighting up a keyboard led every time there is a write operation to the hard-drive (imitating the hard-drive led). Bonus points will be awarded for a creative and ingenious solution.

Deliverables

You must submit the source code for both the kernel driver and the user-space program together with a make file for each of them.