Operating Systems

Complutense University of Madrid

2019-2020

Unit 4: Input/Output Management

Juan Carlos Sáez

OS

Contents

OS

ArTeCS

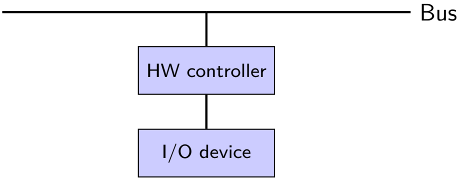
Contents

Introduction

- The CPU constitutes the core part of a computer, but it would be worthless without:
  - Non-volatile storage devices
    - Secondary: disks
    - Tertiary: tapes
  - I/O devices for human interaction (keyboards, mice, microphones, cameras, etc.)
  - Communication devices: enable to connect a computer with other computers by means of a network

Connecting I/O devices to a computer

- In a computer's I/O system two major types of components can be found:
  1. **Peripheral devices or I/O devices:** components connected to the CPU through hardware controllers
     - Mechanical components connected to the computer
  2. **Hardware controllers** are responsible for transferring data between main memory and I/O devices
     - Electronic component making it possible for the I/O device to establish communication with the CPU
     - Feature two connections: one for the computer bus (USB,PCI,..) and another for the I/O device
     - Warning: HW controller $\neq$ Software controller or *Driver*



Objectives of I/O management in the OS

- **Enable user programs to interact with I/O devices seamlessly**
  - Expose I/O devices to programs by means of high-level abstractions and simple APIs
- **Allow the connection of new types of devices** without having to remodel the I/O subsystem in the OS
  - Device classes
- **Handle I/O operations efficiently**
  - Apply device-specific and device-independent optimizations when necessary
- **Enable to detect and setup new I/O devices automatically**, by using *plug & play* mechanisms

Drivers (I)

- Driver: software component of the OS in charge of managing a specific kind of I/O device
  - Also known as software controller
- The code of a driver is typically divided into two parts:
  1. **Device-independent code**: implements a well-defined interface enabling the OS and user programs to interact with the device
     - Drivers handling very different devices may implement the same interface of operations
     - This approach simplifies porting applications to other OSes and different HW platforms
  2. **Device-specific code**: interacts with the I/O device at the low level
     - Direct interaction with the HW controller
     - Interrupt handling
     - ...

Drivers (II)

Other features of I/O devices

- We must also take into account inherent features of I/O devices:
  - Some of them support access at the byte level whereas others do not
  - Example: We cannot read/write an individual byte on a hard disk
    - Instead an entire disk sector or block must be read/written

Typically, devices are divided into 2 broad categories:

1. Character devices
   - keyboard, mouse, serial port,...
2. Block devices
   - disk, network card, screen, ...

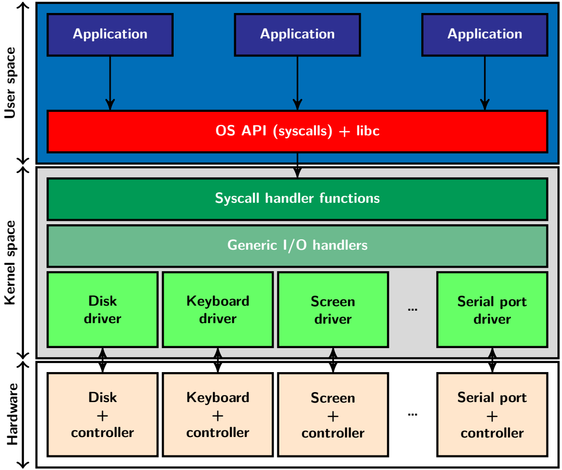
Contents

I/O management on Linux

- Most I/O devices are exposed to the user as special files (block/character special files)
  - `/dev/sda1` represents the first partition of the first SATA or USB disk
  - `/dev/tty0` represents the first terminal or text console
  - `/dev/ttyS0` represents the first serial port
  - `/dev/lp0` for the printer
- Access to those special files is performed via file-related system calls: `open()`, `read()`, `write()` and `close()`
  - User program can access an I/O device as long as they have permissions to access the associated special device file
  - If a device driver supports control operations on the device, the `ioctl()` system call must be used to perform control operations
    - `man 2 ioctl`

Overview

- Each device file has a driver associated with it



When a user program invokes a system call on a special file (e.g. `read()`), a function of the driver's code is invoked to perform the associated processing

Major and minor number

- Each special device file has a unique pair of numbers associated with it: (*major*, *minor*)
  - *major*: ID of the associated device class
  - *minor*: Local ID enabling the driver to distinguish between different devices (in the event it manages several of them)
- On Linux, I/O devices are grouped into device classes
  - Each device class has a major number (ID) associated with it
  - `https://www.kernel.org/doc/Documentation/admin-guide/devices.txt`
- Example: Driver that manages 2 SATA hard disks
  - Disks exposed to the user as `/dev/sda` and `/dev/sdb` respectively
  - Both special files would have the same *major number* but a different *minor number*
- Several *drivers* on Linux may manage different devices with the same major number, but drivers always work with devices in non-overlapping ranges of (major,minor) pairs

Major and minor number

- The `stat` command makes it possible to retrieve the file type as well as the value for the major and minor number for special device files

terminal

```
osuser@debian:~$ stat /dev/tty1
  File: «/dev/tty1»
  Size: 0         Blocks: 0      IO Block: 4096   character special file
Device: 5h/5d   Inode: 1259        Links: 1     Device type: 4,1
Access: (0600/crw-------)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2016-02-24 08:18:59.663968939 +0100
Modify: 2016-02-24 08:18:59.523954207 +0100
Change: 2016-02-24 08:18:59.523954207 +0100
Birth: -
osuser@debian:~$ stat /dev/sda1
  File: «/dev/sda1»
  Size: 0              Blocks: 0           IO Block: 4096   block special file
Device: 5h/5d   Inode: 1708        Links: 1     Device type: 8,1
Access: (0660/brw-rw----)  Uid: (    0/    root)   Gid: (    6/    disk)
Access: 2016-02-24 08:18:59.663968939 +0100
Modify: 2016-02-24 08:18:59.523954207 +0100
Change: 2016-02-24 08:18:59.523954207 +0100
Birth: -
```

Drivers on Linux and kernel modules

terminal

```
osuser@debian:~$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
108 ppp
128 ptm
136 pts
180 usb
189 usb_device
226 drm
...

Block devices:
259 blkext
  8 sd
 11 sr
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
...
```

- The association between a *device driver* and its corresponding major number can be obtained by reading from the `/proc/devices` file

- Most *device drivers* on Linux are implemented as loadable kernel modules
  - Implement a special interface
  - Register themselves as a character or block *driver*

Contents

Linux kernel modules (I)

What is a loadable kernel module?

- A "code fragment" that can be loaded/unloaded into/from the OS kernel's address space on demand
- Its functions are executed in kernel mode (privileged level)
  - Any fatal error in the code may "freeze" the OS
  - Less sophisticated debugging tools available
    - `printk()`: Print a message into the kernel's log file
    - `dmesg` : dumps the contents of the kernel's log file

Loadable kernel modules also exist on other UNIX and UNIX-like systems (BSD, Solaris) as well as on MS Windows

Linux kernel modules (II)

Major benefits

1. Enable to reduce the memory *footprint* of the OS kernel
   - Overall idea: load just the necessary software components (modules)
2. Make it possible to extend the kernel's functionality at run time (without having to restart the system)
   - Preferred mechanism to deploy *device drivers*
3. Lead to a more modular OS design

Linux kernel modules (III)

- The modules available for our kernel are stored in predefined directory
  - /lib/modules/$KERNEL_VERSION
    - $KERNEL_VERSION for the currently running kernel can be obtained with `uname -r`
- The `lsmod` command shows what kernel modules are currently loaded

Terminal

```
osuser@debian:~$ lsmod
Module                 Size  Used by
mperf                   935  0
cpufreq_stats          2139  0
bluetooth             55448  2
cpufreq_powersave       650  0
cpufreq_userspace      1464  0
cpufreq_conservative    3791  0
binfmt_misc            4994  1
uinput                 5172  1
fuse                  49890  3
acpiphp               12757  0
loop                  10809  0
tpm_tis                5725  0
...
```

Anatomy of a Linux kernel module

- A module does not implement a `main()` function, but instead the `init_module()` and `cleanup_module()` functions
  - `init_module()` is invoked when the kernel module is loaded
  - `cleanup_module()` is invoked when the module is removed from the kernel

- When the module is built, an object file (`.ko` ext.) is generated
  - Special ELF (*Executable and Linkable Format*) file

Loading and unloading modules

- To load a kernel module, the `insmod` command must be used:

```
$ insmod my_module.ko
```

- A module can be removed by using the `rmmod` command:

```
$ rmmod my_module
```

- Only the system administrator (*root*) can invoke these commands
  - On the virtual machine, use `sudo <command>` (+ password of `osuser`)

“Hello World” kernel module

hello.c

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

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");
}
```

Building a kernel module

- **Build process managed by a *Makefile***
  - The header files for the currently running kernel must be installed on the machine
    - Already installed in the virtual machine
  - The *Makefile* must be placed **in the same directory as the module sources**
    - To build a module, type "`make`"
    - To remove files generated by the build process, type "`make clean`"
    - The full pathname of the directory with the module sources must not contain spaces

Makefile (module consisting of a single .c file)

```
obj-m =  hello.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
```

Example: building, loading and unloading (I)

Terminal

```
osuser@debian:~/A4Files/Hello$ ls
Makefile  hello.c
osuser@debian:~/A4Files/Hello$ make
make -C /lib/modules/4.9.111-lin/build M=/home/osuser/A4Files/Hello modules
make[1]: Entering directory `/usr/src/linux-headers-4.9.111-lin'
  CC [M]  /home/osuser/A4Files/Hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/osuser/A4Files/Hello/hello.mod.o
  LD [M]  /home/osuser/A4Files/Hello/hello.ko
make[1]: Leaving directory `/usr/src/linux-headers-4.9.111-lin'
osuser@debian:~/A4Files/Hello$ sudo insmod hello.ko
[sudo] password for osuser:
osuser@debian:~/A4Files/Hello$ lsmod | head
Module                  Size  Used by
hello                    836  0
binfmt_misc             6160  1
uinput                  6879  1
nfsd                  198017  2
auth_rpcgss            37914  1 nfsd
oid_registry            2051  1 auth_rpcgss
exportfs                3400  1 nfsd
nfs_acl                 2175  1 nfsd
nfs                   152253  0
```

ArTeCS

Example: building, loading and unloading (II)

Terminal

```
osuser@debian:~/A4Files/Hello$ sudo dmesg | tail
[ 4229.560018] usb 1-2.1: Product: Virtual Bluetooth Adapter
[ 4229.560022] usb 1-2.1: Manufacturer: VMware
[ 4229.560026] usb 1-2.1: SerialNumber: 000650268328
[ 4645.867113] hello: module license 'unspecified' taints kernel.
[ 4645.867117] Disabling lock debugging due to kernel taint
[ 4645.867748] Hello world.
osuser@debian:~/A4Files/Hello$ sudo rmmod hello
osuser@debian:~/A4Files/Hello$ sudo dmesg | tail
[ 4229.396166] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 4229.560005] usb 1-2.1: New USB device found, idVendor=0e0f, idProduct=0008
[ 4229.560013] usb 1-2.1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 4229.560018] usb 1-2.1: Product: Virtual Bluetooth Adapter
[ 4229.560022] usb 1-2.1: Manufacturer: VMware
[ 4229.560026] usb 1-2.1: SerialNumber: 000650268328
[ 4645.867113] hello: module license 'unspecified' taints kernel.
[ 4645.867117] Disabling lock debugging due to kernel taint
[ 4645.867748] Hello world.
[ 4741.556845] Goodbye world.
osuser@debian:~/A4Files/Hello$ lsmod | head
Module                  Size  Used by
binfmt_misc             6160  1
uinput                  6879  1
nfsd                  198017  2
auth_rpcgss            37914  1 nfsd
...
```

Kernel API for modules

- In the Linux kernel there is no libc
  - Just a few functions from the libc are implemented
- Kernel modules can invoke functions to allocate memory, manage timers, etc.
  - Advanced usage of the API for modules: **Optional course “Linux and Android Internals”**

String-related functions

`strlen`, `sprintf`, `strcmp`, `strncmp`, `sscanf`, `strcat`, `memset`, `memcpy`, `strtok`, ...

Allocate/deallocate memory `<linux/vmalloc.h>`

```
void *vmalloc( unsigned long size );
void vfree( void *addr );
```

Other useful functions

Increment/decrement module's reference counter

- `try_module_get(THIS_MODULE);`
- `module_put(THIS_MODULE);`

Transfer data between user and kernel `<linux/uaccess.h>`

- `copy_from_user()`, `copy_to_user()`, `get_user()`, `put_user()`

Contents

Character device drivers

Interface for character device drivers

```
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 *);
....
};
```

Implementing a simple character driver

1. Create a kernel module with functions `init_module()` and `cleanup_module()`
2. Implement operations in the aforementioned interface
   - `struct file_operations`
3. In the `init_module()` function:
   - Reserve range of (major,minor) pairs for the driver
     - `alloc_chrdev_region()`
   - Create a cdev structure and initialize it appropriately
     - The operations in the interface must be then assigned to this structure
     - Use `cdev_alloc()`, `cdev_init()` and `cdev_add()`
4. In the `cleanup_module()` function:
   - Destroy the cdev structure
     - `cdev_del()`
   - Release the (major, minor) pair range reserved in `init_module()`
     - `unregister_chrdev_region()`

Representation of `(major,minor)` in the kernel

- In the Linux kernel, a *(major,minor)* pair is represented via the `dev_t` data type
  - `dev_t`: 32-bit number (12 bits for major, 20 bits for minor)
- Due to historical reasons the internal encoding is complex
- For simplicity/portability across kernel versions, macros are used to manipulate `dev_t` variables:
  - Accessing major/minor: `MAJOR(dev_t dev)`, `MINOR(dev_t dev)`
  - Build a *(major,minor)* pair: `MKDEV(int major, int minor)`
    - Example: `dev_t pair=MKDEV(4,1);`

API description (I)

`alloc_chrdev_region()`

```
#include <linux/fs.h>
int alloc_chrdev_region(dev_t *first, unsigned int firstminor,
                        unsigned int count, char *name);
```

- **Description:**
  - Reserves a consecutive range of (major,minor) pairs when the *major* number is unkown
- **Parameters**:
  - `first`: Output parameter. First *(major,minor)* pair reserved for the driver in the range
  - `firstminor`: Smallest *minor number* to be reserved inside the consecutive range assigned by the kernel
  - `count`: Number of *(major,minor)* pairs to be reserved for the driver
  - `name`: Driver's name (arbitrary string). This will be the name displayed in `/proc/devices` after loading the driver
- **Return value**: 0 on success. Upon failure, it returns a negative number that encodes the error.

API description (II)

`unregister_chrdev_region()`

```
#include <linux/fs.h>
int unregister_chrdev_region(dev_t first, unsigned int count);
```

- **Description:**
  - Free up a consecutive range of (major,minor) pairs that the driver allocated previously
- **Parameters**:
  - `first`: First (major,minor) pair that the driver allocated previously
  - `count`: Number of *(major,minor)* pairs to be freed up
- **Return value**: 0 on success. Upon failure, it returns a negative number that encodes the error.

ArTeCS

cdev structure

- In order for the driver to receive requests from user programs, it must create a cdev structure and initialize it appropriately

Operations on struct cdev <linux/cdev.h>

```
struct cdev *cdev_alloc(void);
```

- Creates a cdev structure. On success, it returns a non-NULL pointer to the created structure

```
void cdev_init(struct cdev *p, struct file_operations *fops);
```

- Associates the driver's interface of operations to a cdev structure

```
int cdev_add(struct cdev *p, dev_t first, unsigned count);
```

- Associates a set of (count) consecutive *(major, minor)* pairs to a cdev structure
  - Upon invoking this function, any operation performed by a user program on a device file with a *(major, minor)* in the specified set will be redirected to the driver

cdev structure

Operations on `struct cdev <linux/cdev.h>`

```
void cdev_del(struct cdev *p);
```

- Frees up a cdev structure and removes the existing associations with *(major, minor)* pairs (if any)

Example: `chardev.c` (1/4)

```c
#include <linux/kernel.h>

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 DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices */
...
dev_t start;            /* Starting (major,minor) pair for the driver */
struct cdev* chardev=NULL; /* Cdev structure associated with the driver */

...

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

Example `chardev.c` (2/4)

```c
/* This function is called when the module is loaded */
int init_module(void) {
    int major;     /* Major number assigned to our device driver */
    int minor;     /* Minor number assigned to the associated character device */
    int ret;

    /* Get available (major,minor) range */
    if ((ret=alloc_chrdev_region(&start, 0, 1,DEVICE_NAME)){
        ... Error handling ...
        return ret;
    }

    /* Create associated cdev */
    if ((chardev=cdev_alloc())==NULL)
        return -ENOMEM;

    cdev_init(chardev,&fops);

    if ((ret=cdev_add(chardev,start,1))){
        ... Error handling ...
        return ret;
    }

    ...
}
```

Example: `chardev.c` (3/4)

```
/* This function is called when the module is loaded */
int init_module(void) {

   ....

   major=MAJOR(start);
   minor=MINOR(start);

   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 "'sudo mknod -m 666 /dev/%s c %d %d'.\n", DEVICE_NAME, major,
        minor);
   printk(KERN_INFO "Try to cat and echo to the device file.\n");
   printk(KERN_INFO "Remove the device file and module when done.\n");

   return SUCCESS;
}
```

Example: `chardev.c` (4/4)

```
/* This function is called when the module is unloaded */
void cleanup_module(void)
{
    /* Destroy chardev */
    if (chardev)
        cdev_del(chardev);

    /*
     * Unregister the device
     */
    unregister_chrdev_region(start, 1);
}
```

operations read()/write()

```
static ssize_t device_read(struct file *file, char *buff, size_t len, loff_t *off);
static ssize_t device_write(struct file *file, const char *buff, size_t len, loff_t
      *off);
```

Relevant parameters

- buff: buffer of characters (or bytes) where the user program passes the data (for write()) or where the module must return the data (for read())
- len
  - For read → maximum number of bytes/characters we can write in buff
  - For write → number of bytes/characters stored in buff

Return value

- Returns the number of bytes that the kernel reads from buff (on write()) o writes in buff (on read())
  - 0 →end of file on read() (there is nothing else to read)
- < 0 → error

Dealing with user-space pointers (I)

- **User-space pointer**: pointer passed as a parameter to a system call (e.g., `read()` or `write()`)
  - The `read` and `write` operations of a character driver accept a pointer to a buffer of the user program (user-space pointer)
- In the kernel, we should never trust the address stored in a user-space pointer
  - NULL pointer
  - Memory region that the user process is not allowed to access

Dealing with user-space pointers (II)

- We should always work with a private copy of the data in kernel space
  - For example, declare a local array `char kbuf[MAX_CHARS]` in the function
  - In `read()`: work with `kbuf` + copy contents of `kbuf` to user space buffer with `copy_to_user()`
  - In `write()`: copy contents of user space buffer to `kbuf` with `copy_from_user()` + do the necessary processing with `kbuf`

<linux/uaccess.h>

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

- Similar semantics to `memcpy()`
- Both functions return the number of bytes that could not be copied

Example: chardev (Building and Loading)

Terminal

```
osuser@debian:~/A4Files$ ls
Chardev  Hello
osuser@debian:~/A4Files$ cd Chardev/
osuser@debian:~/A4Files/Chardev$ ls
Makefile  chardev.c
osuser@debian:~/A4Files/Chardev$ make
make -C /lib/modules/4.9.111-lin/build M=/home/osuser/A4Files/Chardev modules
make[1]: Entering directory `/usr/src/linux-headers-4.9.111-lin'
  CC [M]  /home/osuser/A4Files/Chardev/chardev.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/osuser/A4Files/Chardev/chardev.mod.o
  LD [M]  /home/osuser/A4Files/Chardev/chardev.ko
make[1]: Leaving directory `/usr/src/linux-headers-4.9.111-lin'
osuser@debian:~/A4Files/Chardev$ sudo insmod chardev.ko
[sudo] password for osuser:
osuser@debian:~/A4Files/Chardev$ lsmod | head
Module                  Size  Used by
chardev                 2208  0
binfmt_misc             6160  1
uinput                  6879  1
nfsd                  198017  2
auth_rpcgss            37914  1 nfsd
oid_registry            2051  1 auth_rpcgss
exportfs                3400  1 nfsd
```

Example: chardev (Listing drivers)

Terminal

```
osuser@debian:~/A4Files/Chardev$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
108 ppp
128 ptm
136 pts
180 usb
189 usb_device
226 drm
248 chardev
249 hidraw
250 bsg
251 watchdog
252 rtc
...
```

Invoking the driver's function

To issue requests to the driver...

1. Create a character device file with `mknod` (as *root*)
   - `sudo mknod -m 666 <pathname_char_file> c <major> <minor>`
     - "`-m 666`": grant read/write permissions to everyone
   - The *major number* for the driver can be found in `/proc/devices`
2. Manipulate the character device file:
   - From a user program: `open()`, `read()`, `write()`, `close()`
   - From the shell: `cat`, `echo`
     - `cat <char_file>` → Reads data from the device until reaching EOF (several invocations to `read()` may be necessary) and displays retrieved content on the screen
     - `echo "hello" > <char_file>` → Writes the "hello\n" string (without '\0' at the end) to the device

Example: chardev (Creating the device file)

Terminal

```
osuser@debian:~/A4Files/Chardev$ sudo dmesg | tail
....
[13165.925127] I was assigned major number 248. To talk to
[13165.925130] the driver, create a dev file with
[13165.925132] 'sudo mknod -m 666 /dev/chardev c 248 0'.
[13165.925133] Try to cat and echo to the device file.
[13165.925134] Remove the device file and module when done.
osuser@debian:~/A4Files/Chardev$ sudo mknod -m 666 /dev/chardev c 248 0
osuser@debian:~/A4Files/Chardev$ stat /dev/chardev
  File: `/dev/chardev'
  Size: 0            Blocks: 0          IO Block: 4096   character special file
Device: 4h/4d   Inode: 32859       Links: 1     Device type: f8,0
Access: (0666/crw-rw-rw-)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2018-12-2 19:49:25.720129709 +0100
Modify: 2018-12-2 19:49:25.720129709 +0100
Change: 2018-12-2 19:49:25.720129709 +0100
 Birth: -
osuser@debian:~/A4Files/Chardev$ ls -l /dev/chardev
crw-rw-rw- 1 root root 248, 0 Dec 2 19:49 /dev/chardev
osuser@debian:~/A4Files/Chardev$
```

Example: chardev (Manipulate the device file)

Terminal

```
osuser@debian:~/A4Files/Chardev$ cat /dev/chardev
I already told you 0 times Hello world!
osuser@debian:~/A4Files/Chardev$ cat /dev/chardev
I already told you 1 times Hello world!
osuser@debian:~/A4Files/Chardev$ cat /dev/chardev
I already told you 2 times Hello world!
osuser@debian:~/A4Files/Chardev$ echo hello > /dev/chardev
-bash: echo: write error: Operation not permitted
osuser@debian:~/A4Files/Chardev$
```

Contents

Lab assignment

Requirements

1. Using the virtual machine is mandatory for this assignment:
   - Specific version of the Linux kernel (4.9.x-4.15.x)
   - Root access required (administrator permissions)
     - Run command as root via `sudo` + password "osuser" (Debian VM):

       ```
       $ sudo <command>
       ```

     - Open root shell via `sudo`:

       ```
       $ sudo -i
       ```

2. A computer with a standard PC keyboard must be used
   - On laptops without standard LEDs (e.g., Macbook), you can connect a USB keyboard to the computer and let the VMware VM to manage it
     - Connecting a second mouse or keyboard directly to a hosted VM
     - Connect USB HIDs to a Virtual Machine

Working on the assignment

1. Read the lab assignment instruction sheet carefully
2. Do exercises found there
3. Complete the code of the assignment (2 parts):
   - (**Part A**) Write a device driver (`chardev_leds.c`) that controls the LEDs of a standard PC keyboard
     - Reuse code of the various examples provided
   - (**Part B**) Write a user program (`leds_user.c`) that interacts with the driver created in Part A

Handing in the assignment

- The assignment must be submitted at the end of the lab session (Dec 16, at 10:50am)
  - No extra part. To get extra marks during the session
    - demonstrate that the assignment's code works (show it to the professor)
    - Take the test on the assignment
- Upload a compressed file
  - L<lab_number>_P<PC_number>_A4.tar.gz
  - Must include a *Makefile* to build the kernel module
    - The leds_user.c program must be compiled manually:

```
$ gcc -Wall -g leds_user.c -o leds_user
```

Contents of the compressed file (.zip or .tar.gz)