Character Device Drivers

Praktikum „Kernel Programming“

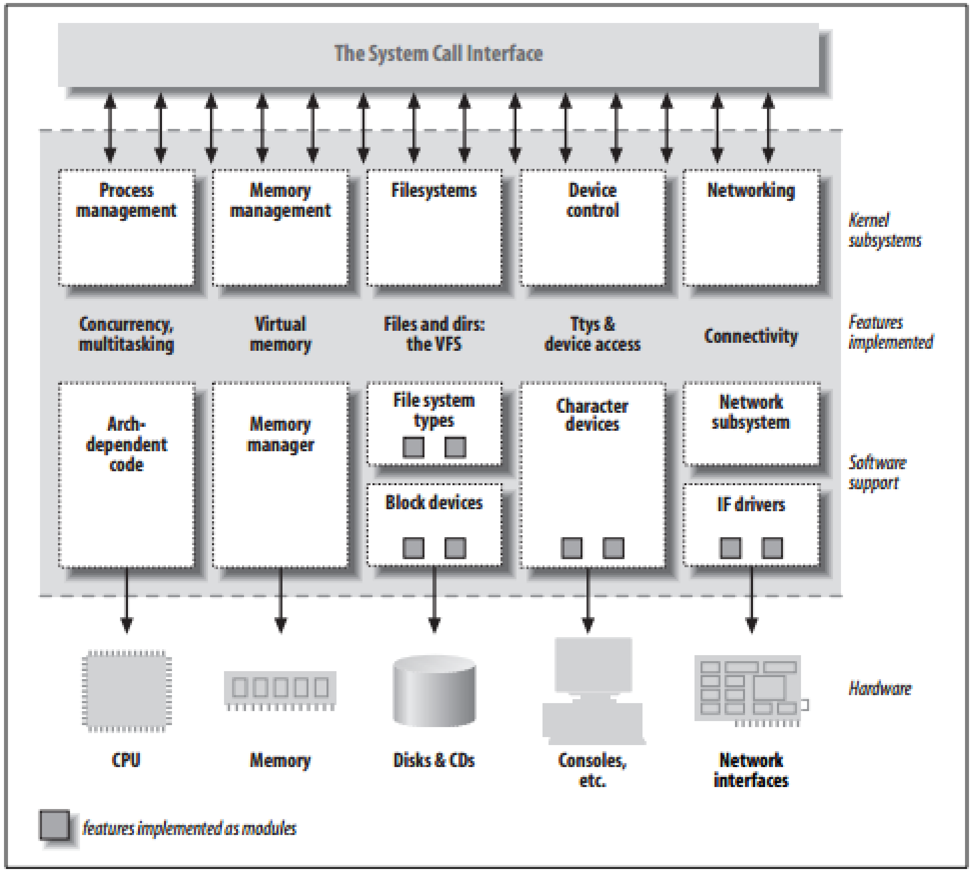
Johannes Coym

December 2, 2015

Outline

- What are character device drivers
- Example of the connection between application and character device
- Major and minor numbers
- File operations
- ioctl (Input/Output control)
- Blocking I/O
- Access control

What are character device drivers

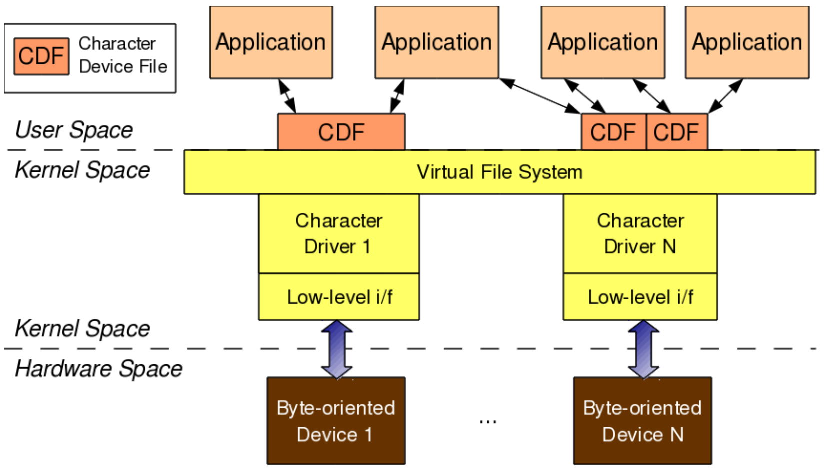


What are character device drivers

- Character devices can be accessed as a stream of bytes
- Character device drivers implement *open*, *close*, *read* and *write* most of the time and grant access to the data stream for the user space
- Examples for character devices:
  - Serial Ports (/dev/ttyS0)
  - Console (/dev/console)
  - Mouse (/dev/input/mouse0)
  - (all devices that are neither storage nor network devices)

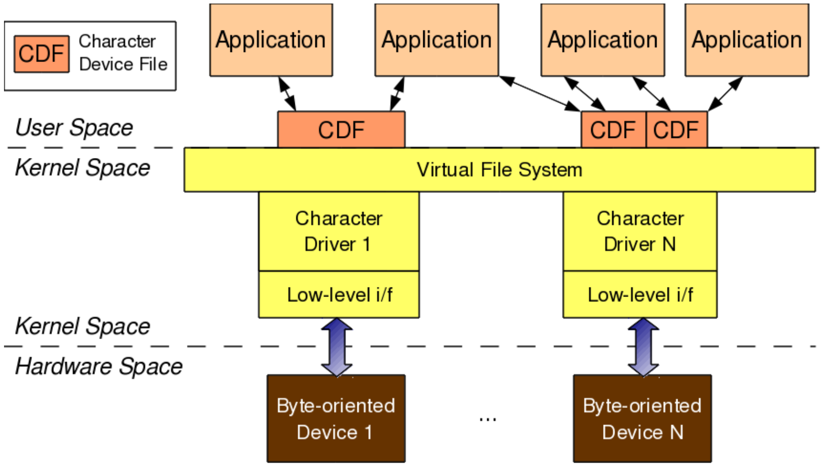
What are character device drivers

- Connection between application and the device in 4 steps:
  - Application
  - Character device file
  - Character device driver
  - Character device



Example of the connection between application and character device

- The music player writes the music to play into the CDF
- The character device driver takes the music from the CDF and sends it as a byte stream to the character device



Major and minor numbers

- Access to device driver from user space through device file
- Kernel needs to know to which driver and which device the device file belongs

➢ Device files mapped by the kernel to a major and a minor number
  - ➢ Major number refers top the driver, each driver has its own
  - ➢ Minor number refers to the device which is managed by the driver

Major and minor numbers

- Limit of 255 major and 255 minor numbers
- Each combination of major and minor number is unique and mapped to a device file
- Some functions need to know the major number
- In the Kernel the type dev_t contains major and minor number of a device

Major and minor numbers

- To get the major or minor number from a dev_t:
  - `MAJOR(dev_t dev);`
  - `MINOR(dev_t dev);`
- To get a dev_t from the major and the minor number:
  - `MKDEV(int major, int minor);`

Major and minor numbers

- Two types of major and minor number region allocation:
  - `int register_chrdev_region(dev_t first, unsigned int count, char *name);`
    - Static allocation where it’s not sure if you’ll get the requested region
    - If the minor numbers exceed the 255 it will automatically assign the next major too, if it’s free
  - `int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);`
    - Dynamic allocation of the device numbers by the kernel
    - You will definitely get a free major number assigned
- To free the assigned major and minor numbers in the exit function:
  - `void unregister_chrdev_region(dev_t first, unsigned int count);`

Major and minor numbers

- Allocating only a major number with it's full 256 minor numbers:
  - `int register_chrdev (unsigned int major, const char * name, const struct file_operations * fops);`
    - Will try to allocate the given major
    - Setting major to 0 will change the functions behavior to dynamically allocate a major number
- Free the assigned major number:
  - `void unregister_chrdev(unsigned int major, const char * name);`

File Operations

```
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 (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, 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 (*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);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
                          loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
};
```

File Operations

- Structure defined in linux/fs.h
- Contains pointers to the common file operations by the drivers
- Usage:

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

File Operations – open/release

- `int (*open) (struct inode *, struct file *);`
- `int (*release) (struct inode *, struct file *);`
- Return value: 0 for success, negative numbers for failure
- Struct inode * is a struct defined in linux/fs.h and includes information about the device
- Struct file * is a struct defined in linux/fs.h and references to the device file

File Operations – read/write

- `ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);`
- `ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);`
- Return value: the size read or written
- Struct file * is a struct defined in linux/fs.h and references to the device file
- Char __user * is the buffer we receive from user space
- Size_t is the size of the requested transfer
- Loff_t is the long offset type indicating the position in the file the user is accessing

File Operations - llseek

- `loff_t (*llseek) (struct file *, loff_t, int);`
- Return value: New position in the file
- Struct file * is a struct defined in linux/fs.h and references to the device file
- Loff_t is the value defining how much the position will be changed
- Int defines where it should start (0 from beginning, 1 at current position, 2 at end)

ioctl (Input/Output control)

- Used for device control of the driver
- Can include software commands like receiving error logs
- Can also include hardware commands like opening a CD drive
- Some command-oriented character devices like terminals use commands instead of ioctl
  - It’s also possible to use only ioctl instead of read and write, you just have to implement the read and write operations as ioctl commands

ioctl (Input/Output control)

- Prototype definition:
  - `int ioctl(int fd, unsigned long cmd, ...);`
  - … stands for an optional argument `char *argp`
- Each ioctl command is defined by one 8 bit Type number for the driver and an additional 8 bit Number for the actual command
- Should return –ENOTTY when an undefined ioctl command is called

ioctl (Input/Output control)

- Each module can define its own ioctl commands
- The ioctl commands should be defined in a header file in combination with the major number
  - Most of the time static major number allocation, when working with ioctl
- The header file should be referenced by any programs using the ioctl commands

ioctl (Input/Output control)

- Usually with a switch-case
- Selecting in switch case which command was sent to him
- Default should just return –ENOTTY
- On success of one command 0 or an answer to the user space program should be returned
- Arguments can be given as a pointer or value and can be received as a return value or pointer

Blocking I/O

- If the driver gets a request which can‘t handle right now he puts the process to sleep
- Reasons for the driver to be not able to handle the request:
  - Receiving a read request when there is no data to read available
  - Receiving a write request when the buffer is already full

Blocking I/O

- To send a process to sleep, we need a wait queue
  - Static initialized wait queue; initialized at compile time:

    ```
    DECLARE_WAIT_QUEUE_HEAD (my_queue);
    ```

  - Dynamic initialized wait queue; intialized at runtime

    ```
    wait_queue_head_t my_queue;

    init_waitqueue_head (&my_queue);
    ```

Blocking I/O

- Several ways to send a process to sleep:
  - `sleep_on(wait_queue_head_t *queue);`
  - `interruptible_sleep_on(wait_queue_head_t *queue);`
  - `sleep_on_timeout(wait_queue_head_t *queue, long timeout);`
  - `interruptible_sleep_on_timeout(wait_queue_head_t *queue, long timeout);`
  - `void wait_event(wait_queue_head_t queue, int condition);`
  - `int wait_event_interruptible(wait_queue_head_t queue, int condition);`

Blocking I/O

- All variants of sleep_on can be woken up using this commands:
  - `wake_up(wait_queue_head_t *queue);`
  - `wake_up_interruptible(wait_queue_head_t *queue);`
  - `wake_up_sync(wait_queue_head_t *queue);`
  - `wake_up_interruptible_sync(wait_queue_head_t *queue);`
- The wait_event variants don‘t need a wake_up call, but wake up automatically on the condition

Access Control

- Single-open lock:
  - Device file can only be opened by one process at the same time
  - Usually implemented with an Integer which is 0 when no process is using the driver and 1 when it‘s busy
- Single-user lock:
  - Device file can be opened by all processes owned by one user
  - Usually implemented using a field saving the owner of the first process opening the file

Access Control

- „Blocking-open“:
  - Device file can be opened by any process at any time but if another process is using the device, the calling process will have to wait
  - Usually implemented with a wait queue
- Cloning the device
  - When open is called by a process, it gets it‘s own copy of the device file as a virtual device file

Literature

„Major and Minor Numbers“
http://www.makelinux.net/ldd3/chp-3-sect-2

Corbet, Rubini, Kroah-Hartmann(2005). „Linux Device Drivers“
https://static.lwn.net/images/pdf/LDD3/ch01.pdf

„Device Drivers, Part 4: Linux Character Drivers“
http://opensourceforu.efytimes.com/2011/02/linux-character-drivers/

„Character Device Files“
http://www.tldp.org/LDP/lkmpg/2.4/html/c577.htm

„Enhanced Char Driver Operations“
http://www.xml.com/ldd/chapter/book/ch05.html

„Writing a Linux Kernel Module — Part 2: A Character Device”
http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/