Linux Device Driver

(Character Devices)

Amir Hossein Payberah

payberah@yahoo.com

Contents

- Major and Minor number
- Important Structures
- Open and Release
- Read and Write
- Device Filesystem

Major and Minor numbers

- Special files under /dev “c” for char & “b” for block
- Major number identifies driver use at open time
- Minor number is used only by driver to control several devices

```
crw-rw-rw- 1 root   root     1,   3    Feb 23 1999    null
crw-------  1 root   root    10,  1    Feb 23 1999    psaux
crw-------  1 rubini tty      4,   1    Aug 16 22:22   tty1
crw-rw-rw- 1 root   dialout  4,   64   Jun 30 11:19   ttyS0
crw-rw-rw- 1 root   dialout  4,   65   Aug 16 00:00   ttyS1
crw-------  1 root   sys      7,   1    Feb 23 1999    vcs1
crw-------  1 root   sys      7,   129  Feb 23 1999    vcsa1
crw-rw-rw- 1 root   root     1,   5    Feb 23 1999    zero
```

Register a new driver

- int register_chrdev (unsigned int major, const char *name, struct file_operations *fops);
  - □ Tells the kernel to remember the major number and the name of the device driver associated with it.
  - □ fops point to a global structure which kernel finds

Create device node

- mknod /dev/name c major minor
  - The name should be the same
  - Now users can access the device

Dynamic major number

- register_chrdev (major, “name”, *fops)
  - when major = 0, it returns a dynamically allocated major number
- Disadvantage
  - You can’t create the device nodes because the major number assigned to your module can’t be guaranteed to always be the same.

Dynamic major number

- Use /proc/devices

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  6 lp
  7 vcs
  10 misc
  13 input
  14 sound
  21 sg
  180 usb
Block devices:
  2 fd
  8 sd
  11 sr
  65 sd
  66 sd
```

```
major=‘awk "\\$2==\"$module\" {print \\$1}" /proc/devices‘
```

Dynamic major number

```
result = register_chrdev(major, “scull”, &scull_fops);
if (result < 0)
{
    printk(w_level “scull: cannot get a major %d\n” major);
    return result;
}
if (major == 0) //dynamic major allocation
    major = result;
```

Remove a driver

- int unregister_chrdev(unsigned int major, const char *name);

Minor number

- Every time the kernel calls a device driver, it tells the driver which device is being acted upon.
- The major and minor numbers are paired in a single data type that the driver uses to identify a particular device.
  - It resides in the field i_rdev of the inode structure.

dev_t

- Historically, Unix declared dev_t to hold the device numbers.
- It used to be a 16-bit integer value.
- Nowadays, more than 256 minor numbers are needed at times,
  - □ Changing dev_t is difficult

kdev_t

- Within the Linux kernel, a different type, kdev_t, is used.

kdev_t macros

- MAJOR(kdev_t dev);
  - Extract the major number from a kdev_t structure.
- MINOR(kdev_t dev);
  - Extract the minor number.
- MKDEV(int ma, int mi);
  - Create a kdev_t built from major and minor numbers.
- kdev_t_to_nr(kdev_t dev);
  - Convert a kdev_t type to a number (a dev_t).
- to_kdev_t(int dev);
  - Convert a number to kdev_t.

Contents

- Major and Minor number
- Important Structures
- Open and Release
- Read and Write
- Device Filesystem

file_operations structure

- An open device is identified internally by a file structure.
- The kernel uses the file_operations structure to access the driver's functions.
- The structure, defined in <linux/fs.h>.
- It is an array of function pointers.

file_operations structure

```
struct file_operations
{
    loff_t (*llseek) (struct file *, loff_t, int)
    ssize_t (*read) (struct file *, char *, size_t, loff_t *)
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*fsync) (struct inode *, struct dentry *, int);
    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 *);
    struct module *owner;
};
```

file_operations functions

- llseek
  - It is used to change the current read/write position in a file.
- read
  - Used to retrieve data from the device.
- write
  - Sends data to the device.

file_operations functions

- readdir
  - This field should be NULL for device files; it is used for reading directories, and is only useful to filesystems.
- poll
  - Used to inquire if a device is readable or writable or in some special state.
- ioctl
  - It offers a way to issue device-specific commands (like formatting a track of a floppy disk, which is neither reading nor writing).

file_operations functions

- mmap
  - It is used to request a mapping of device memory to a process’s address space.
- open
  - This is always the first operation performed on the device file.
- release
  - This operation is invoked when the file structure is being released.

file_operations functions

- flush
  - □ The flush operation is invoked when a process closes its copy of a file descriptor for a device.
- fsync
  - □ When user calls to flush any pending data.
- fasync
  - □ This operation is used to notify the device of a change in its FASYNC flag.

file_operations functions

- lock
  - It is used to implement file locking.
- readv and writev
  - These system calls allow them to do read or write operation involving multiple memory areas without forcing extra copy operations on the data.
- owner
  - It is a pointer to the module that “owns” this structure.

file_operations sample

```
struct file_operations scull_fops = {
    read: scull_read,
    write: scull_write,
    open: scull_open,
    release: scull_release,
    owner: THIS_MODULE
};
```

file structure

- The file structure represents an open file.
- It is created by the kernel on open and is passed to any function that operates on the file, until the last close.
- It is defined in <linux/fs.h>.

file structure

- An open file is different from a disk file, represented by struct inode.
- A struct file has nothing to do with the FILEs of user-space programs.
  - A FILE is defined in the C library and never appears in kernel code.
  - A struct file is a kernel structure that never appears in user programs.

file structure

```
struct file
{
    mode_t f_mode;
    loff_t f_pos;
    unsigned int f_flags;
    struct file_operations *f_op;
    void *private_data;
    …
};
```

file structure fields

- mode_t f_mode
  - The file mode identifies the file as either readable or writable (or both).
- loff_t f_pos
  - The current reading or writing position.
- unsigned int f_flags
  - These are the file flags, such as O_RDONLY, O_NONBLOCK, and O_SYNC.

file structure fields

- struct file_operations *f_op
  - The operations associated with the file.
- void *private_data
  - The driver can use this field to point to allocated data.

Contents

- Major and Minor number
- Important Structures
- Open and Release
- Read and Write
- Device Filesystem

The open method

- Increment the usage count.
- Check for device-specific errors.
- Initialize the device, if it is being opened for the first time.
- Identify the minor number and update the f_op pointer.
- Allocate and fill any data structure to be put in filp->private_data.

The open method

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

The release method

- Deallocate anything that open allocated in filp->private_data.
- Shut down the device on last close.
- Decrement the usage count.

The release method

- int release(struct inode *inode, struct file *filp);

Contents

- Major and Minor number
- Important Structures
- Open and Release
- Read and Write
- Device Filesystem

Read and Write

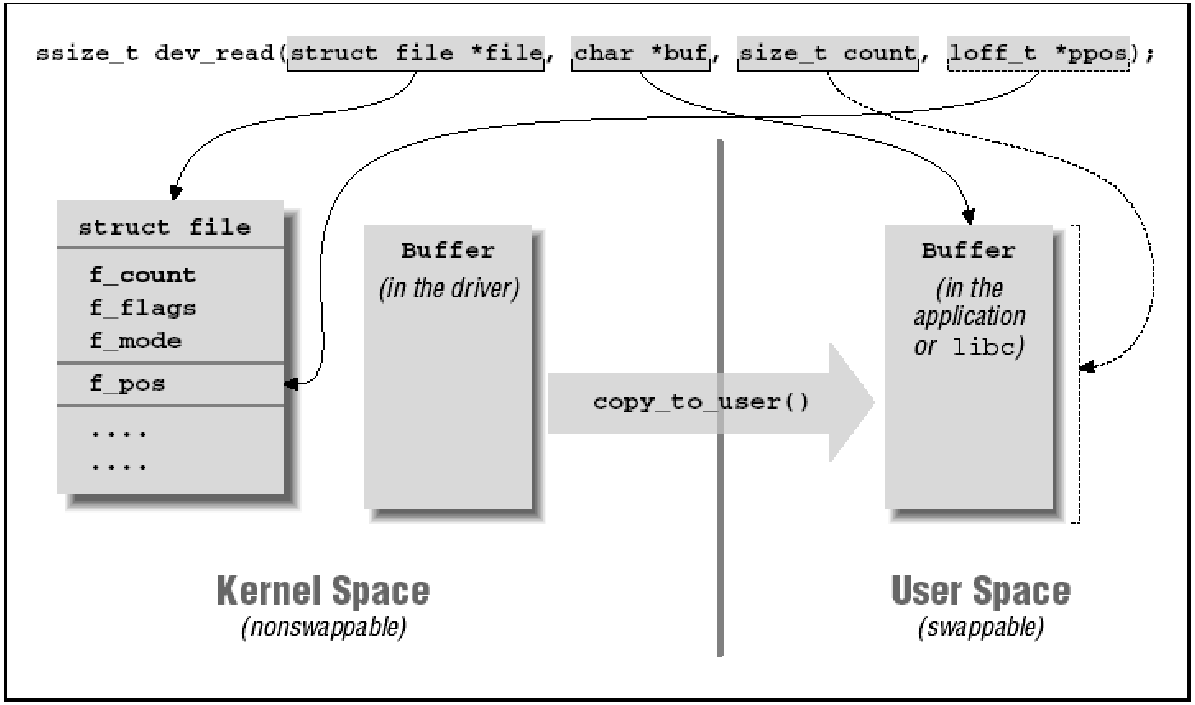
- The read and write methods perform a similar task, that is, copying data from and to application code.

Read and Write

- ssize_t read(struct file *filp, char *buff, size_t count, loff_t *offp);
- ssize_t write(struct file *filp, const char *buff, size_t count, loff_t *offp);
- The buff argument points to the user buffer holding the data.
- offp is a pointer to a “long offset type” object that indicates the file position the user is accessing.

Kernel space to User space

- unsigned long copy_to_user(void *to, const void *from, unsigned long cour



User space to Kernel space

- unsigned long copy_from_user(void *to, const void *from, unsigned long count);

Contents

- Major and Minor number
- Important Structures
- Open and Release
- Read and Write
- Device Filesystem

Device filesystem

- Version 2.4 of the kernel
  - introduced a new (optional) feature, the device file system or devfs.
- If this file system is used, management of device files is simplified and quite different;

Advantage of devfs

- Device entry points in /dev are created at device initialization and removed at device removal.
- There is no need to allocate a major number for the device driver and deal with minor numbers.

Devfs functions

- devfs_handle_t devfs_mk_dir (devfs_handle_t dir, const char *name, void *info);
- devfs_handle_t devfs_register (devfs_handle_t dir, const char *name, unsigned int flags, unsigned int major, unsigned int minor, umode_t mode, void *ops, void *info);
- void devfs_unregister (devfs_handle_t de);

Question?