Systems Software/Programming

Device Driver

https://lwn.net/Kernel/LDD3/

Linux Architecture

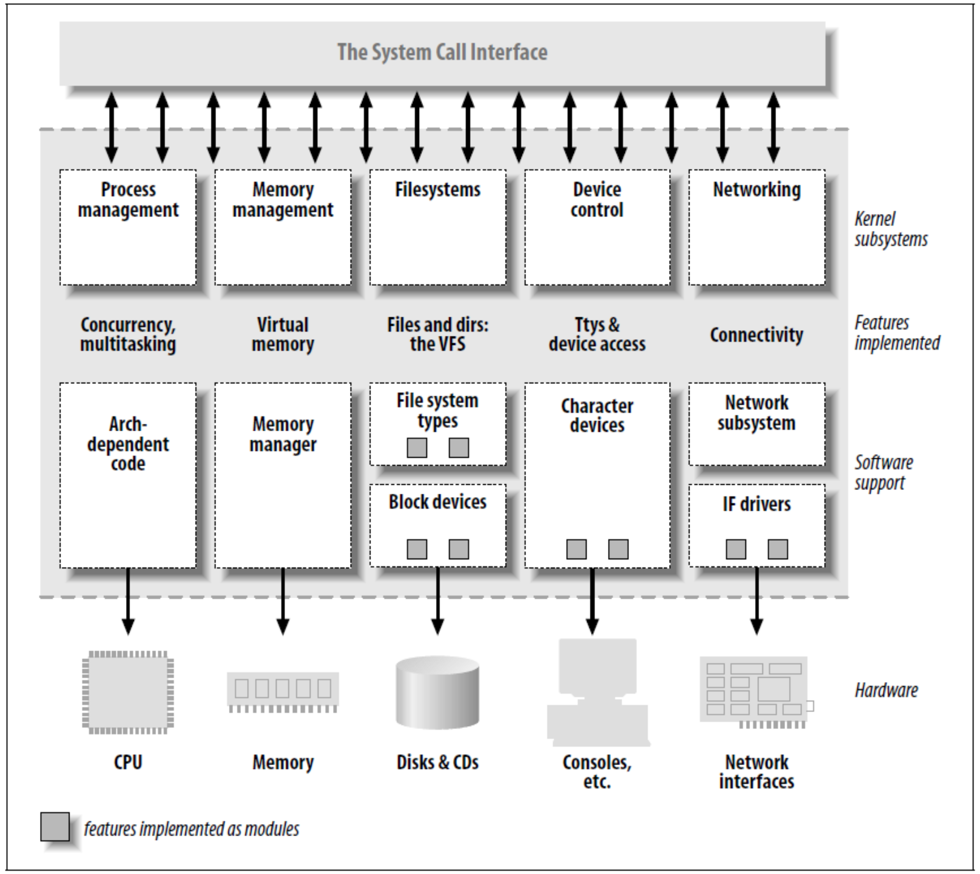


Figure 1-1. A split view of the kernel

- Linux Kernel has several modules:
  - Process Management (studied)
  - Memory Management – create and manage of virtual address space
  - Filesystems – Manages files and directories stored in physical disk by dividing it into logical components.
  - Device Control/Driver – Software designed to communicate and operate specific hardware so that kernel use the hardware without having to know the details.
  - Networking – Protocol implementation

Device Driver Types

- Character Device
  - Read/write character at a time and implements open, read, write, close system calls
  - Character devices are synchronous and only one process can use it at a time
  - Examples: console, keyboard, mouse etc
- Block Device
  - Read/write block of data at a time
  - Writing is done asynchronously so multiple process can write at the same time
  - Examples: CDROM, HDD, USB etc
- Network Device
  - Sends or receives data in form of packets.
  - Example: Hardware device: Ethernet card, Software device: Loopback adapter

Device Driver Module Information

- License:
  - Specified using MODULE_LICENSE(“<license>") macro
  - <license> : “GPL”, “GPL v2”, “Dual BSD/GPL”, “Dual MIT/GPL”, "Proprietary“ etc
- Author:
  - Specified using MODULE_AUTHOR("Author") macro
- Module Description:
  - Specified using MODULE_DESCRIPTION(“My Driver") macro
- Module Version:
  - Format [<epoch>:]<version>[-<extra-version>]
  - Specified using MODULE_VERSION("2:1.0") macro

Constructor and Exit functions

- Init function: module_init(* func()) macro
  - Called when driver module is loaded e.g. using insmod

```
static int __init mydriver_init(void) /* Constructor */
{
    return 0;
}
module_init(mydriver_init);
```

- Exit function: module_exit(* func()) macro
  - Called when driver module is unloaded e.g. using rmmod

```
void __exit mydriver_exit(void)
{

}
module_exit(mydriver_exit);
```

printk()

- printk() kernel level function similar to printf() which is user level
- Except, printk() allow assigning loglevel/priority of messages
    - KERN_EMERG: emergency
    - KERN_ALERT: requiring immediate attention
    - KERN_CRIT: critical condition related to hardware or software
    - KERN_ERR: error reporting
    - KERN_WARNING: warning reporting
    - KERN_NOTICE: take a note
    - KERN_INFO: informational messages
    - KERN_DEBUG: debugging information

Simple Driver

- Source DeviceDriver\mydriver1.c and makefile DeviceDriver\mydriver1_makefile
- Compile: $sudo make
- List module: $lsmod | grep mydriver1
- Load module: $sudo insmod mydriver1.ko
- Unload module: $ sudo rmmod mydriver1
- To see messages from printk(): $dmesg

Passing Parameters to Driver Code DeviceDriver\mydriver2.c DeviceDriver\mydriver2_makefile

- module_param(name, type, perm): Initialize primitive type variable
  - creates the sub-directory under /sys/module
  - Type: bool, invbool, charp (char ptr), int, long, short, uint, ulong, ushort
  - Perm: S_I(W/R)(USR/GRP/OTH), can be combined with bit OR (|)
- module_param_array(name, type, num, perm): For arrays
- module_param_cb(): register callback function which is called when change in value is detected in parameter.
- Load module: $ $ sudo insmod mydriver2.ko value=13 name="DAIICT" arr_value=10,20,30,40
- Files for each parameter with its values is created in /sys/module/mydriver2/parameters/ → "cat name" or "cat cb_value"
- To change the value:
  - First login as root user by $su
  - Change value : $sudo bash –c 'echo new_value > /sys/module/mydriver2/parameters/param_name'
    - e.g. sudo bash -c 'echo 20 > /sys/module/mydriver2/parameters/cb_value'
  - Check messages in system log using dmesg

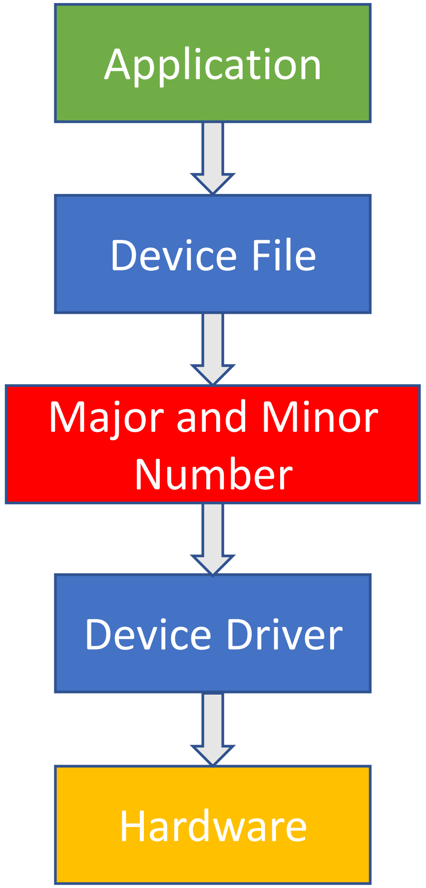
All devices have Major and Minor Numbers

- Major number represents which driver will be used to manage a particular device. It allows multiple devices to have the same major number if they are managed by the same driver.
- Minor number represents a particular device that is managed by the driver identified using Major number.
- In the example below there are 4 tty devices represented as minor numbers 0, 10, 11, 12 but they are all managed by same driver number 4. Similarly 2 disk sda devices with minor numbers 0 and 1 are managed by driver major number 8.

```
$ ls -ltr /dev
crw--w----  1 root     tty      4,   0 May 15 12:46 tty0
crw--w----  1 root     tty      4,  10 May 15 12:46 tty10
crw--w----  1 root     tty      4,  12 May 15 12:46 tty12
crw--w----  1 root     tty      4,  11 May 15 12:46 tty11
brw-rw---- 1 root disk 8, 0 May 15 12:46 /dev/sda
brw-rw---- 1 root disk 8, 1 May 15 12:46 /dev/sda1
```

C stands for char device

B stands for block device



Character Device Driver → Create Major and Minor Numbers [DeviceDriver\mydriver3.c](DeviceDriver\mydriver3.c) [DeviceDriver\mydriver3_makefile](DeviceDriver\mydriver3_makefile)

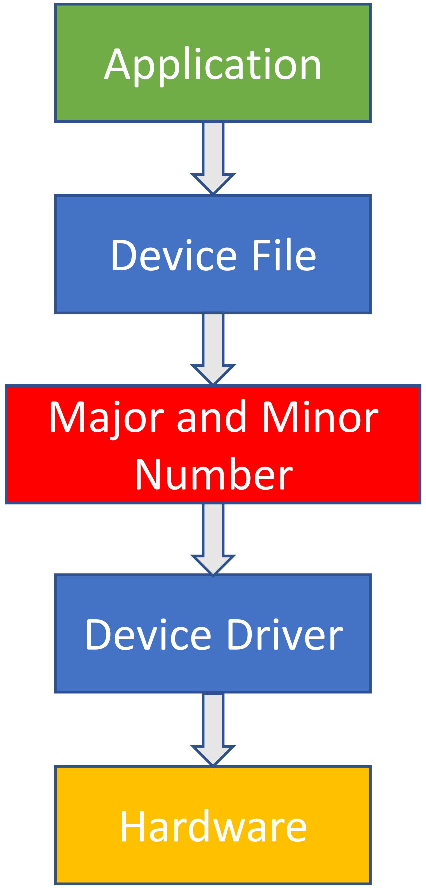
- dev_t struct from linux/types.h used to represent <major>:<minor>
- Create device's major and minor nos using macro MKDEV(int major, int minor) that will return dev_t structure.

dev_t dev = MKDEV(201, 0)

- Statically Register the Device → you provide value for major number, can have conflict even if different drivers are used.

int register_chrdev_region(dev_t first, unsigned int count, char *name);

  - first → starting device number; dev_t dev = MKDEV(201, 0)
  - count → number of devices requested i.e. number of minor numbers
  - name → device name representing number range [first, first+count]



Character Device Driver → Create Major and Minor Numbers DeviceDriver\mydriver3.c DeviceDriver\mydriver3_makefile

- Dynamically Register the device → Major number is allocated to you by kernel

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

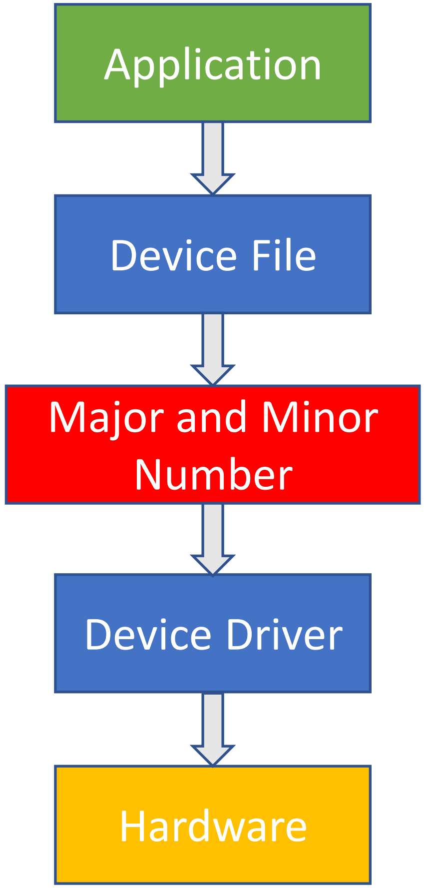
  - dev → major number is returned by kernel which is free to use
  - Firstminor → you assign starting minor number

- Unregister the device

void unregister_chrdev_region(dev_t first, unsigned int count);

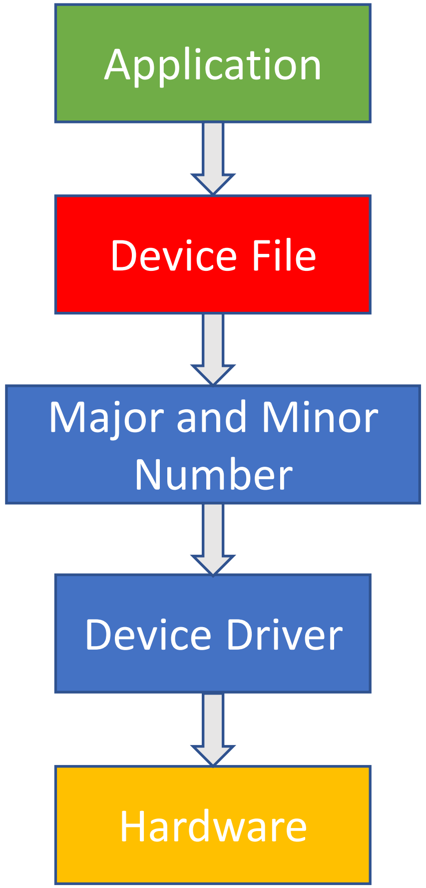
After driver is loaded using insmod, check major number for your device in “cat /proc/devices”

You will not find physical device in /dev directory yet since we have not created the actual device which will be done next.



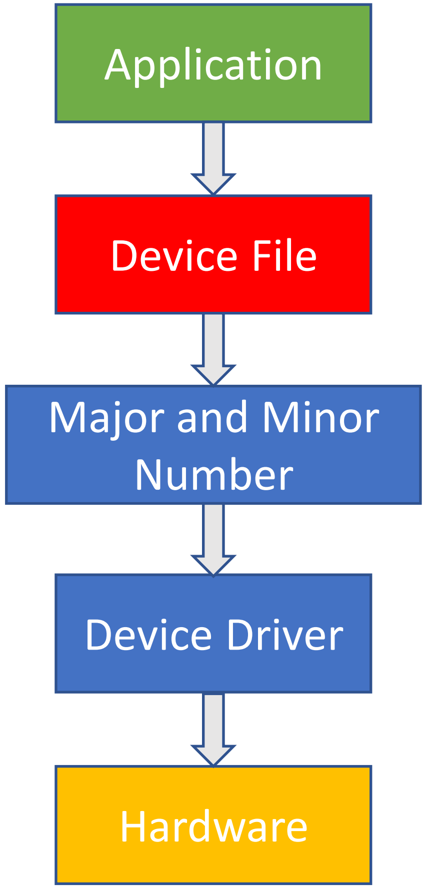
Character Device Driver → Create Device File

- All devices are treated as files and are stored in /dev directory in Unix-based OS.
- Create the device using shell command mknod:
  - mknod -m <permissions> <name> <device type> <major> <minor>
    - name – full path e.g. /dev/mydev
    - device type – c for char device, b for block device
    - Example: $sudo mknod -m 666 /dev/mydev c 201 0
  - Check device creation using → ls -l /dev/ | grep <devname>
- Better approach is to create the device as part of the driver code itself.



Character Device Driver → Create Device File

- Automatically (Programmatically) [DeviceDriver\mydriver4.c](DeviceDriver\mydriver4.c) [DeviceDriver\mydriver4_makefile](DeviceDriver\mydriver4_makefile)
  - header file linux/device.h and linux/kdev_t.h
  - Create structure class for our driver under /sys/class:
    - struct class * class_create (struct module *owner, const char *name);
    - void class_destroy (struct class * cls); // destroy once done
  - Create device:
    - struct device *device_create (struct class * cls, struct device *parent, dev_t dev, const char *fmt, ...);
    - void device_destroy (struct class * class, dev_t devt); // destroy when done
  - Check device creation using → ls -l /dev/ | grep <devname>



cdev Structure linux/cdev.h

- cdev structure is used by linux kernel to represent char device. Inode of this device will have pointer to cdev structure

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

- owner = THIS_MODULE defined in linux/module.h
- ops = list of function pointers which implements the operations for this device
- Kobj = information about driver kernel object
- Dev = your device

inode structure has pointer to cdev structure. In the virtual box installation at /usr/src/linux-headers-5.8.0-48-generic/include/linux/fs.h

```
struct inode {
        umode_t                 i_mode;
        unsigned short          i_opflags;
        kuid_t                  i_uid;
        kgid_t                  i_gid;
        unsigned int            i_flags;
.
.
.
.
.
union {
        struct pipe_inode_info  *i_pipe;
        struct block_device     *i_bdev;
        struct cdev             *i_cdev;
        char                    *i_link;
        unsigned                i_dir_seq;
};
```

File Operation Structure defined in linux/fs.h

struct file_operations mydev_fops = {

.owner = THIS_MODULE,

.read = mydev_read,

.write = mydev_write,

.ioctl = mydev_ioctl,

.open = mydev_open,

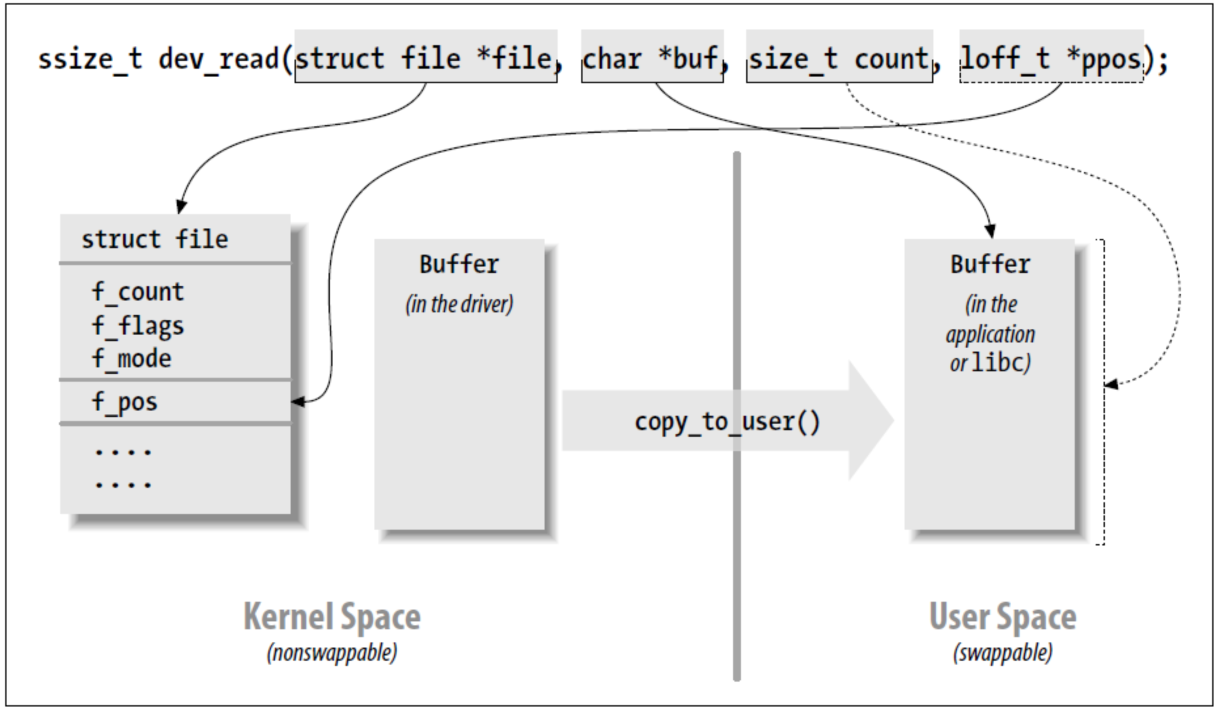
.release = mydev_release,

};

There are many other file_operations structure members but have listed important ones only.

File Operation Structure defined in linux/fs.h

- .owner : struct module *owner → owner of the structure (THIS_MODULE)
- mydev_read: ssize_t (*read) (struct file *file, char _ _user * buf, size_t count, loff_t *ppos); → function pointer which implements reading of data from device.
    - char _ _user * is pointer to user mode buffer to which data need to be read
    - size_t is number of bytes to be read
    - loff_t represents current file position for reading and writing.



File Operation Structure defined in linux/fs.h

- mydev_write: ssize_t (*write) (struct file *, const char _ _user *, size_t, loff_t *); → function pointer which implements writing of the data to device. loff_t represents current file position for reading and writing.
- mydev_ioctl : int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long); → ioctl function implementation such as formatting etc
- mydev_open : int (*open) (struct inode *, struct file *); → open the device and allocate file structure
- mydev_close: int (*release) (struct inode *, struct file *); → close device and release file structure

Useful memory function inside driver code

- void *kmalloc(size_t size, gfp_t flags); defined in linux/slab.h
    - GFP_USER – Allocate memory on behalf of user. May sleep.
    - GFP_KERNEL – Allocate normal kernel ram. May sleep.
    - GFP_ATOMIC – Allocation will not sleep. May use emergency pools. For example, use this inside interrupt handlers.
    - GFP_HIGHUSER – Allocate pages from high memory.
    - GFP_NOIO – Do not do any I/O at all while trying to get memory.
    - GFP_NOFS – Do not make any fs calls while trying to get memory.
    - GFP_NOWAIT – Allocation will not sleep.
    - __GFP_THISNODE – Allocate node-local memory only.
- void kfree(const void *objp) → free allocated memory
- unsigned long copy_from_user(void *to, const void __user *from, unsigned long n); → copy data from user space buffer to kernel space buffer
- unsigned long copy_to_user(const void __user *to, const void *from, unsigned long n); → copy data from kernel space buffer to user space buffer

Cdev and File Operations

DeviceDriver\mydriver5.c

DeviceDriver\mydriver5_makefile

- Create file_operation structure with function pointers for function that you implement

```
static struct file_operations fops =
{
.owner        = THIS_MODULE,
.read         = mydev_read,
.write        = mydev_write,
.open         = mydev_open,
.release      = mydev_release,
};
```

- Initialize cdev structure with specific file_operations

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

- Tell kernel which device uses this cdev structure

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

Test the driver Code using Application

- You device will by default have permission to the user who created the device which is generally root because we use sudo

$ ls -ltr /dev/mydevice

crw------- 1 root root 243, 0 Mar 28 15:41 mydevice

- Enable the read-write permission for other users too (otherwise only root user can access the device)

$ ls -ltr /dev/mydevice

crw-rw-rw- 1 root root 243, 0 Mar 28 15:41 mydevice

Now you can test device with your application
DeviceDriver\mydriver5_test.c