Engineering excellence. Sourced.

Linux Device Driver Basics

By

-HEMAKUMAR

Session-2

Character Drivers

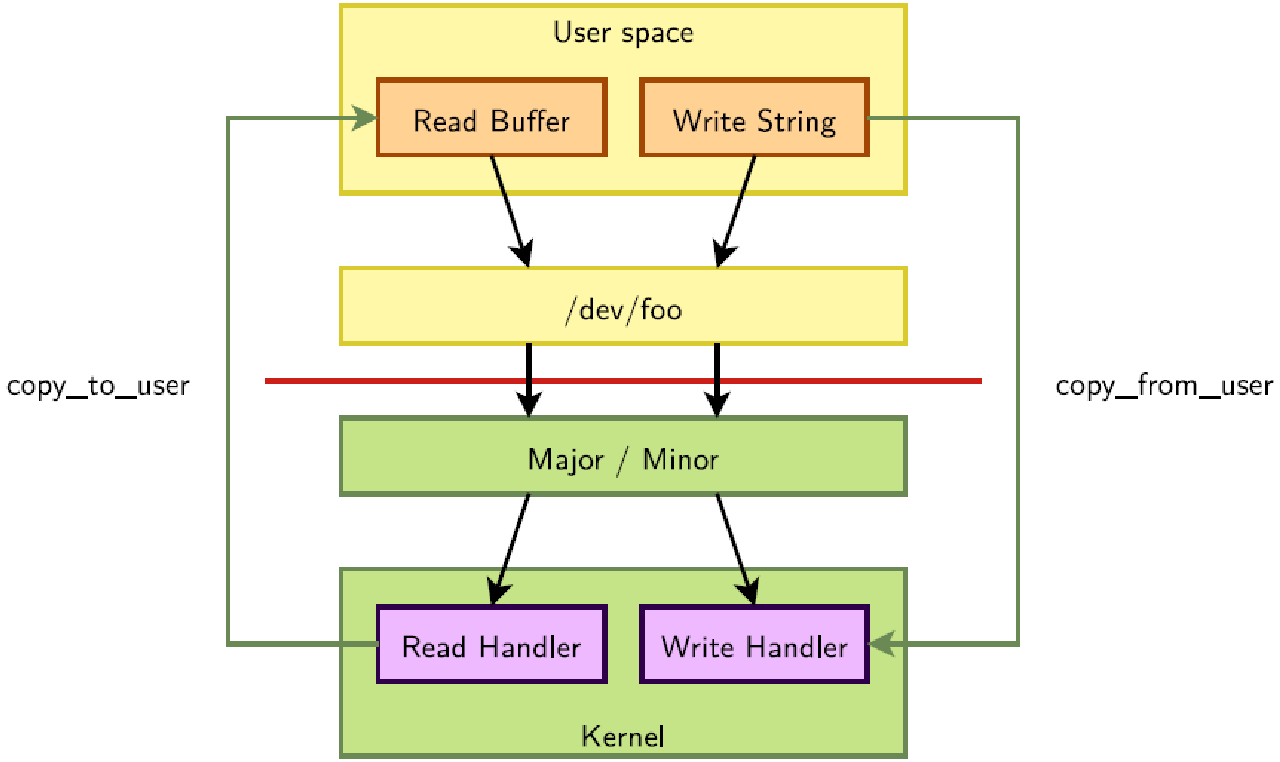
What to Expect?

- After this session, you would know
  - W's of Character Drivers
  - Major & Minor Numbers
  - Registering & Unregistering Character Driver
  - File Operations of a Character Driver
  - Writing a Character Driver
  - Linux Device Model
  - udev & automatic device creation

Character Driver in Kernel

- From the point of view of an application, a character device is essentially a file.
- The driver of a character device must therefore implement operations that let applications think the device is a file: open, close, read, write, etc.
- In order to achieve this, a character driver must implement the operations described in the struct file_operations structure and register them.
- The Linux filesystem layer will ensure that the driver's operations are called when a user space application makes the corresponding system call.

From user space to the kernel: character devices



File operations

- Here are the most important operations for a character driver. All of them are optional.

```
#include <linux/fs.h>
struct file_operations
{
    ssize_t (*read) (struct file *, char_user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char_user *, size_t, loff_t
    *);    long (*unlocked_ioctl) (struct file *, unsigned int,
    unsigned long);    int (*mmap) (struct file *, struct
    vm_area_struct *);
    int (*open) (struct inode *, struct file
    *);  int (*release) (struct inode *, struct
    file *);
};
```

open() and release()

- int foo_open(struct inode *i, struct file *f)

  ▶Called when user space opens the device file.

  ▶struct inode is a structure that uniquely represents a file in the system (be it a regular file, a directory, a symbolic link, a character or block device)

  ▶struct file is a structure created every time a file is opened. Several file structures can point to the same inode structure.
    - ▶Contains information like the current position, the opening mode, etc.
    - ▶Has a void *private_data pointer that one can freely use.
    - ▶A pointer to the file structure is passed to all other operations

- int foo_release(struct inode *i, struct file *f)
  - ▶Called when user space closes the file.

read()

- ssize_t foo_read(struct file *f, char__user *buf, size_t sz, loff_t *off)
  - ▶Called when user space uses the read() system call on the device.
  - ▶Must read data from the device, write at most sz bytes in the user space buffer buf, and update the current position in the file off. f is a pointer to the same file structure that was passed in the open() operation
  - ▶Must return the number of bytes read. 0 is usually interpreted by userspace as the end of the file.
  - ▶On UNIX, read() operations typically block when there isn't enough data to read from the device

Write()

- ssize_t foo_write(struct file *f, const char__user *buf, size_t sz, loff_t *off)

    ▶Called when user space uses the write() system call on the device

    ▶The opposite of read, must read at most sz bytes from buf, write it to the device, update off and return the number of bytes written.

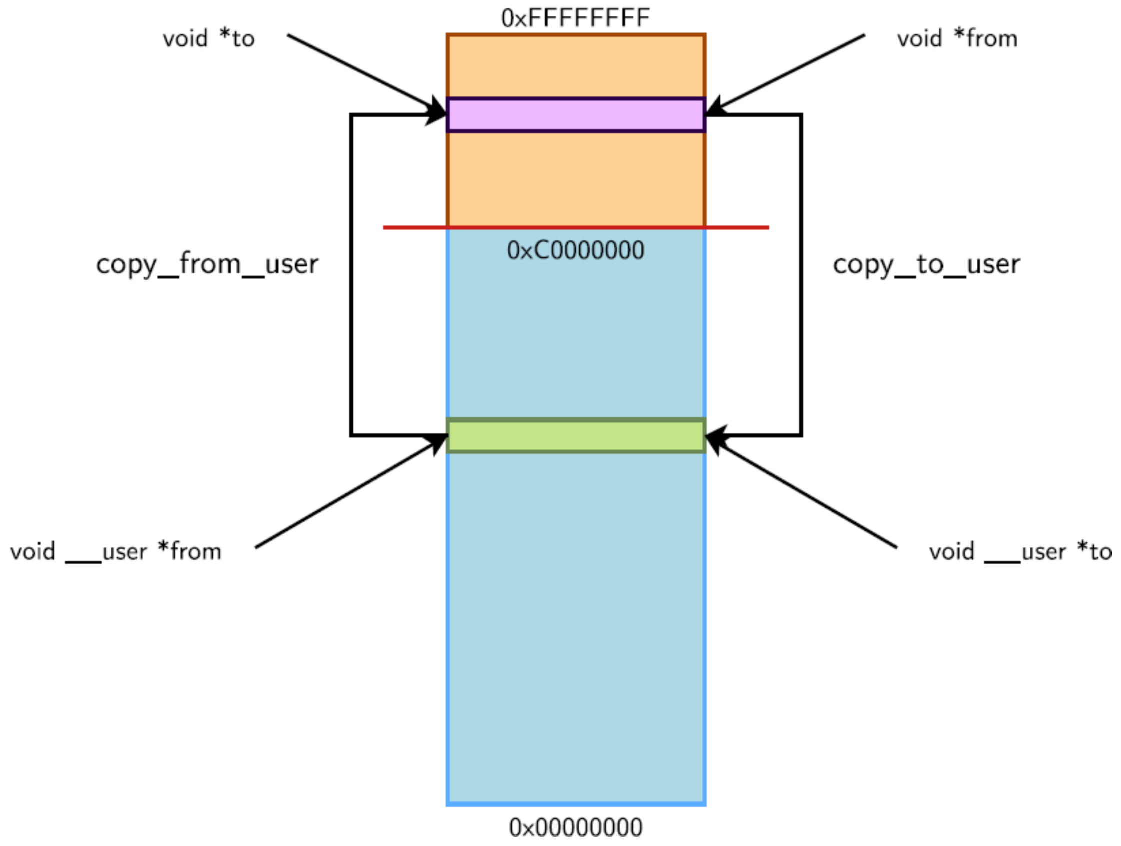
Exchanging data with user space

- Kernel code isn't allowed to directly access user space memory, using memcpy() or direct pointer dereferencing.
  - ▶Doing so does not work on some architectures.
  - ▶If the address passed by the application was invalid, the application would segfault.
  - ▶ Never trust user space. A malicious application could pass a kernel address which you could overwrite with device data (read case), or which you could dump to the device (write case).
- To keep the kernel code portable, secure, and have proper error handling, your driver must use special kernel functions to exchange data with user space.

Exchanging data with user space.

- A single value
  - ▶get_user(v, p);
    - ▶The kernel variable v gets the value pointed by the user space pointer p
  - ▶put_user(v, p);
    - ▶The value pointed by the user space pointer p is set to the contents of the kernel variable v.
- A buffer
  - ▶ unsigned long copy_to_user(void__user *to, const void *from, unsigned long n);
  - ▶ unsigned long copy_from_user(void *to, const void_user *from, unsigned long n);
- The return value must be checked. Zero on success, non-zero on failure. If non-zero, the convention is to return -EFAULT.

Exchanging data with user space ..



Zero copy access to user memory

- Having to copy data to or from an intermediate kernel buffer can become expensive when the amount of data to transfer is large (video).

- Zero copy options are possible:
  - ▶mmap() system call to allow user space to directly access memory mapped I/O space. See our mmap() chapter.
  - ▶get_user_pages_fast() to get a mapping to user pages without having to copy them.

unlocked_ioctl()

- long unlocked_ioctl(struct file *f, unsigned int cmd, unsigned long arg)

  ▶Associated to the ioctl() system call.

  ▶Called unlocked because it didn't hold the Big Kernel Lock (gone now).

  ▶Allows to extend the driver capabilities beyond the limited read/write API.

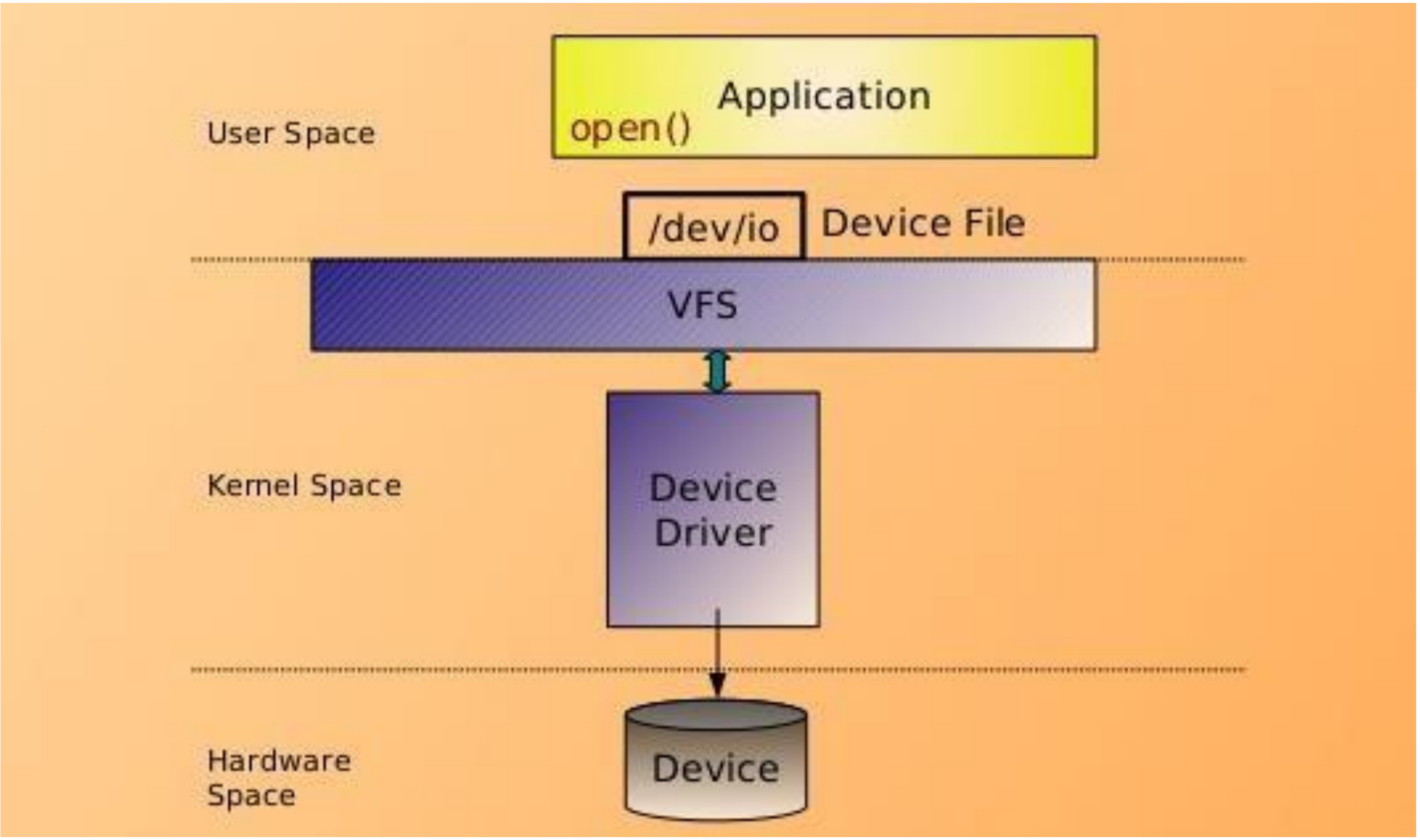
  ▶For example: changing the speed of a serial port, setting video output format, querying a device serial number...

  ▶cmd is a number identifying the operation to perform

  ▶arg is the optional argument passed as third argument of the ioctl() system call. Can be an integer, an address, etc.

  ▶The semantic of cmd and arg is driver-specific.

3 Entities in 3 Spaces



The /dev/null read & write

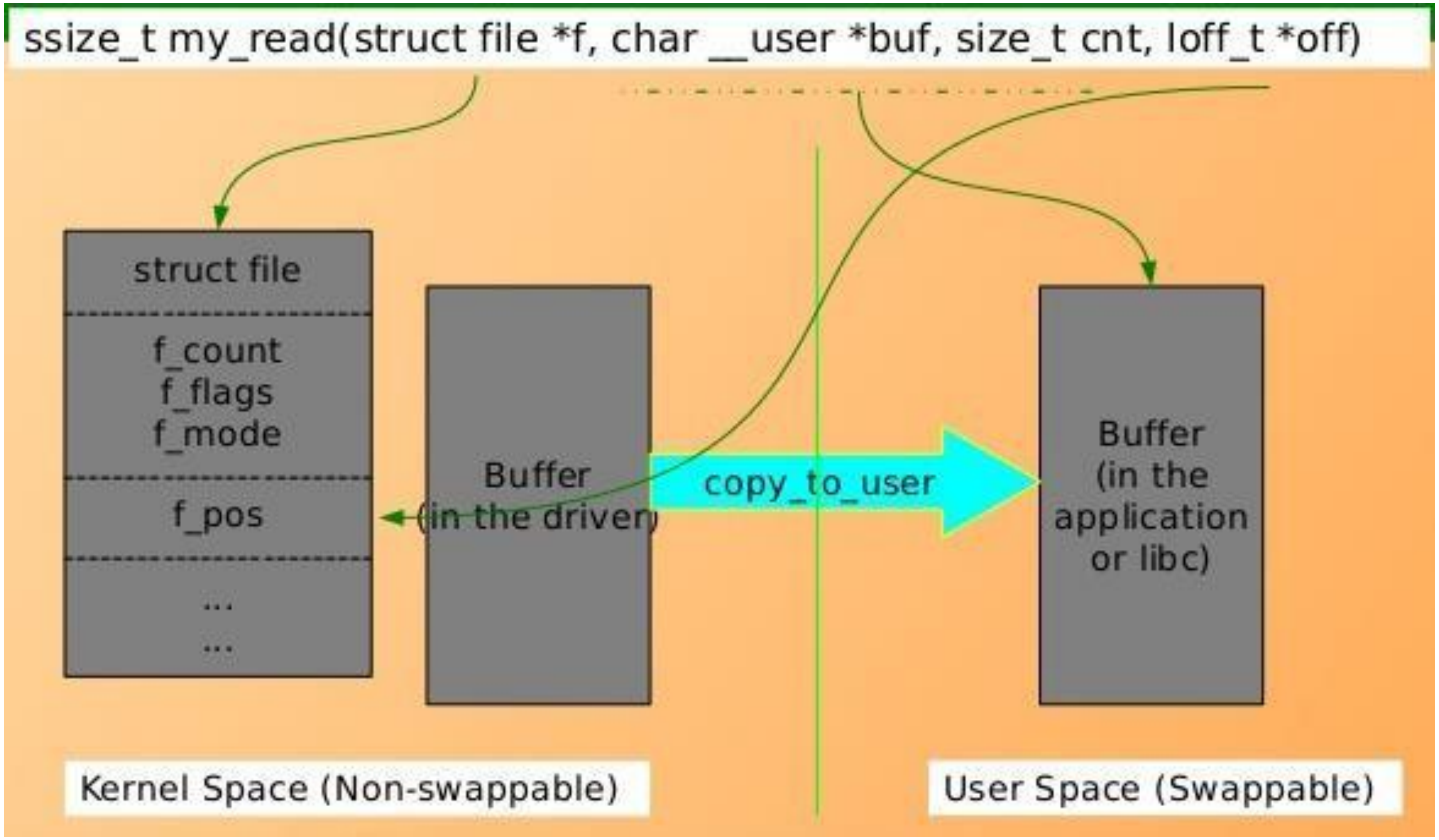
```
ssize_t my_read(struct file *f, char __user *buf, size_t cnt, loff_t *off)
{
    ...
    return read_cnt;
}
ssize_t my_write(struct file *f, char __user *buf, size_t cnt, loff_t *off)
{
    ...
    return wrote_cnt;
}
```

The mem device read

```
#include <asm/uaccess.h>

ssize_t my_read(struct file *f, char __user *buf, size_t cnt, loff_t *off)
{
    ...
    if (copy_to_user(buf, from, cnt) != 0)
    {
        return -EFAULT;
    }
    ...
    return read_cnt;
}
```

The read flow

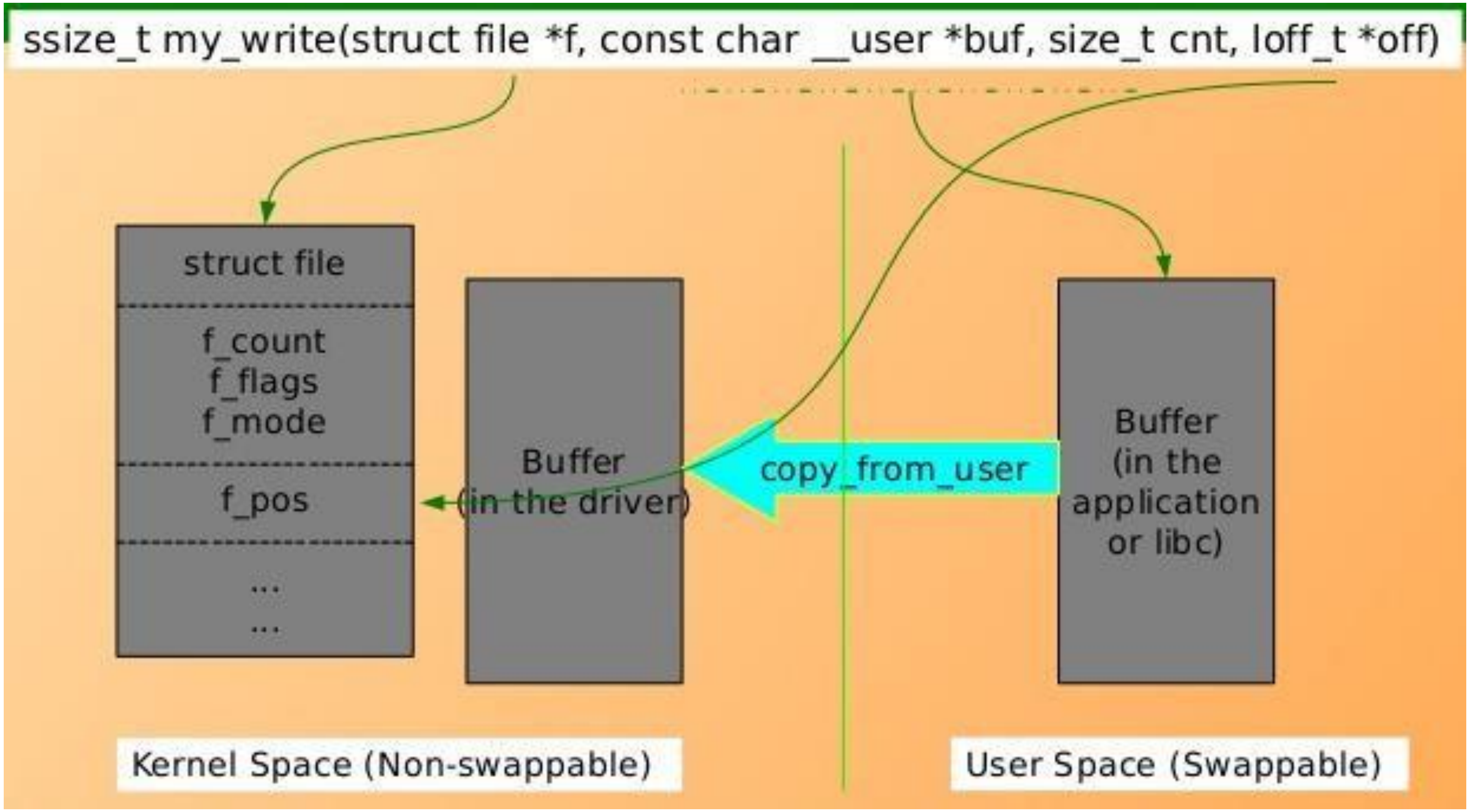


The mem device write

```
#include <asm/uaccess.h>

ssize_t my_write(struct file *f, const char __user *buf, size_t cnt, loff_t *off)
{
    ...
    if (copy_from_user(to, buf, cnt) != 0)
    {
        return -EFAULT;
    }
    ...
    return wrote_cnt;
}
```

The write flow



The mem device write

```
#include <asm/uaccess.h>


ssize_t my_write(struct file *f, const char __user *buf, size_t cnt, loff_t *off)
{
    ...
    if (copy_from_user(to, buf, cnt) != 0)
    {
        return -EFAULT;
    }
    ...
    return wrote_cnt;
}
```

W's of Character Drivers

- What does “Character” stand for?
- Look at entries starting with 'c' after
  - ls -l /dev
- Device File Name
  - User Space specific
  - Used by Applications
- Device File Number
  - Kernel Space specific
  - Used by Kernel Internals as easy for Computation

Major & Minor Number

- ls -l /dev
- Major is to Category; Minor is to Device
- Data Structures described in Kernel C in object oriented fashion
- Type Header: <linux/types.h>
  - Type: dev_t – 12 bits for major & 20 bits for minor
- Macro Header: <linux/kdev_t.h>
  - MAJOR(dev_t dev)
  - MINOR(dev_t dev)
  - MKDEV(int major, int minor)

Major & Minor Number

Major and Minor Number for a driver:

- Every device driver is identified by a unique ID, which is of 2 parts. Major number and Minor number.

- Major Number: A number indicating which device driver should be used to access a particular device.

- All devices controlled by the same device driver have a common major device number. The minor device numbers are used to distinguish between different devices and their controllers.

- Minor Number: will come in to the picture when there are same kind of multiple devices.

Registering & Unregistering

- Registering the Device Driver
  - int register_chrdev_region(dev_t first, unsigned int count, char *name);
  - int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int cnt, char *name);
- Unregistering the Device Driver
  - void unregister_chrdev_region(dev_t first, unsigned int count);
- Header: <linux/fs.h>
- Kernel Window: /proc/devices

Registering the file operations

- The Registration
  - int cdev_add(struct cdev *cdev, dev_t num, unsigned int count);
- The Unregistration
  - void cdev_del(struct cdev *cdev);
- Header: <linux/cdev.h>

Initialization for Registration

- 1st way initialization
  - struct cdev *my_cdev = cdev_alloc();
  - my_cdev->owner = THIS_MODULE;
  - my_cdev->ops = &my_fops;
- 2nd way initialization
  - struct cdev my_cdev;
  - cdev_init(&my_cdev, &my_fops);
- Header: <linux/cdev.h>

The file operations

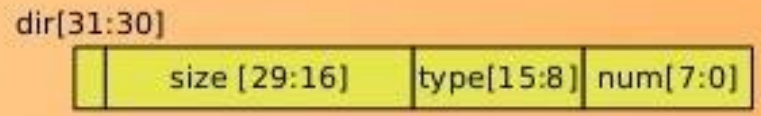
- struct file_operations
  - struct module owner = THIS_MODULE; /* <linux/module.h> */
  - int (*open)(struct inode *, struct file *);
  - int (*release)(struct inode *, struct file *);
  - ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
  - ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
  - loff_t (*llseek)(struct file *, loff_t, int);
  - int (*unlocked_ioctl)(struct file *, unsigned int, unsigned long);
- Header: <linux/fs.h>

The file & inode structures

- Important fields of struct file
  - mode_t f_mode
  - loff_t f_pos
  - unsigned int f_flags
  - struct file_operations *f_op
  - void *private_data
- Important fields of struct inode
  - unsigned int iminor(struct inode *);
  - unsigned int imajor(struct inode *);

The I/O Control API

- API
  - int (*unlocked_ioctl)(struct file *, unsigned int cmd, unsigned long arg)
- Command
  - Macros
    - _IO, _IOW, _IOR, _IOWR
  - Parameters
    - type (character) [15:8]
    - number (index) [7:0]
    - size (param type) [29:16]
- Header: <linux/ioctl.h> →...→ <asm-generic/ioctl.h>



SIMPLE CHARACTER DRIVER - Implementation of LCD driver

CHARACTER DRIVER of LCD driver:

```
#include <linux/modules.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include  <linux/uaccess.h>

#define LCD_MAJOR 254  static
unsigned char inuse = 0;  static
int  nbytes;

int mylcd_open(struct inode * inode, struct file * filp)
{
    printk("my_LCD _ open Invoked \n");
    if(inuse)
        return EBUSY;
    inuse =  1;
    return 0;
}
int   mylcd_release(struct inode * inode, struct file * filp)
{
    inuse =  0;
    return  0;
}
```

SIMPLE CHARACTER DRIVER - Implementation of LCD driver..

```
Size_t mylcd_write(stuct file * file, const char *buf, size_t count, loff_t *f_pos)
{     char data[10];
      nbytes = copy_from_user(data, buf, count);
      printk("\n data = %s", data);
      return nbytes;
}
Static struct file_operations fops = {
          write:    mylcd_write,
          open :    mylcd_open,
          release:  mylcd_release,
};
Int   mylcd_init(void)
{
      int result =  0;
      inuse =  0;
      result = register_chardev(LCD_MAJOR,      "mylcd" ,  &fops);
      return  0;
}
Void mylcd_exit(void)
{
      unregister_chrdev(LCD_MAJOR,  "mylcd");
}
MODULE_DESCRIPTION("TEST LCD DRIVER");
MODULE_AUTHOR("MY_TEAM");
```

Test App to SIMPLE CHARACTER DRIVER - Implementation of LCD driver…

Test application in U-space:

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

Main()
{
    int  fd,  n;
    char  buf[12]  = “Hello World”;
    fd  =  open(“/dev/mylcd”, O_RDWR);
    Printf(“\n fd  =  %d”,  fd);
    getchar();
    n  = write(fd, buf, 10);
    printf(“ No. of bytes read %d\n”, n);
}
```

SIMPLE CHARACTER DRIVER - Implementation of LCD driver….

Compilation & Execution of above driver:

[root@testkerdrv]# **make**

[root@testkerdrv]# **insmod mylcd.ko**

[root@testkerdrv]# **cat /proc/devices | more** -- to check driver inserted successfully or not.

char devices:

1 mem

4 /dev/vc/0

254 mylcd

256 pcidev

[root@testkerdrv]# **mknod /dev/mylcd c 254 0**

[root@test]# ./a.out

fd = 3; -- open call got executed

[root@testkerdrv]# **dmesg**

mylcd open invoked.

data = hello world.

Ioctl() example: Kernel side

```
static long phantom_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
 {struct phm_reg r;
   void __user *argp = (void __user *)arg;

  switch (cmd) {
  case PHN_SET_REG:
         if (copy_from_user(&r, argp,
             sizeof(r)))  return -EFAULT;
         /*  Do something
         */   break;
   case PHN_GET_REG:
       if (copy_to_user(argp, &r,
           sizeof(r)))  return -EFAULT;
       /*  Do something
       */   break;
  default:
      return -ENOTTY;
 }
 return 0; }
```

Ioctl() Example: Application Side

```
int main(void)
 {
    int fd, ret;
    struct phm_reg reg;
    fd =
    open("/dev/phantom");
    assert(fd >0);
    reg.field1 =42;
     reg.field2 =67;
    ret =ioctl(fd, PHN_SET_REG, &
    reg);  assert(ret ==0);
    return 0;
}
```

Linux Device Model (LDM)

- struct kobject - <linux/kobject.h>
  - kref object
  - Pointer to kset, the parent object
  - kobj_type, type describing the kobject
- kobject instantiation → sysfs representation
- Parent object guides the entries under /sys/
  - bus – the physical buses
  - class – the device categories
  - device – the actual devices

udev & LDM

- Daemon: udevd
- Configuration: /etc/udev/udev.conf
- Rules: /etc/udev/rules.d/
- Utility: udevinfo [-a] [-p <device_path>]
- Receives uevent on a change in /sys
- Accordingly, updates /dev &/or
- Performs the appropriate action for
  - Hotplug
  - Microcode / Firmware Download
  - Module Autoload

Device Model & Classes

- Latest way to create dynamic devices
  - Create or Get the appropriate device category
  - Create the desired device under that category
- Class Operations
  - struct class *class_create(struct module *owner, char *name);
  - void class_destroy(struct class *cl);
- Device into & out of Class
  - struct class_device *device_create(struct class *cl, NULL, dev_t devnum, NULL, const char *fmt, ...);
  - void device_destroy(struct class *cl, dev_t devnum);

What all have we learnt?

- W's of Character Drivers
- Major & Minor Numbers
- Registering & Unregistering Character Driver
- File Operations of a Character Driver
- Writing a Character Driver
- Linux Device Model
- udev & automatic device creation

Any Queries?

Books for Ref..

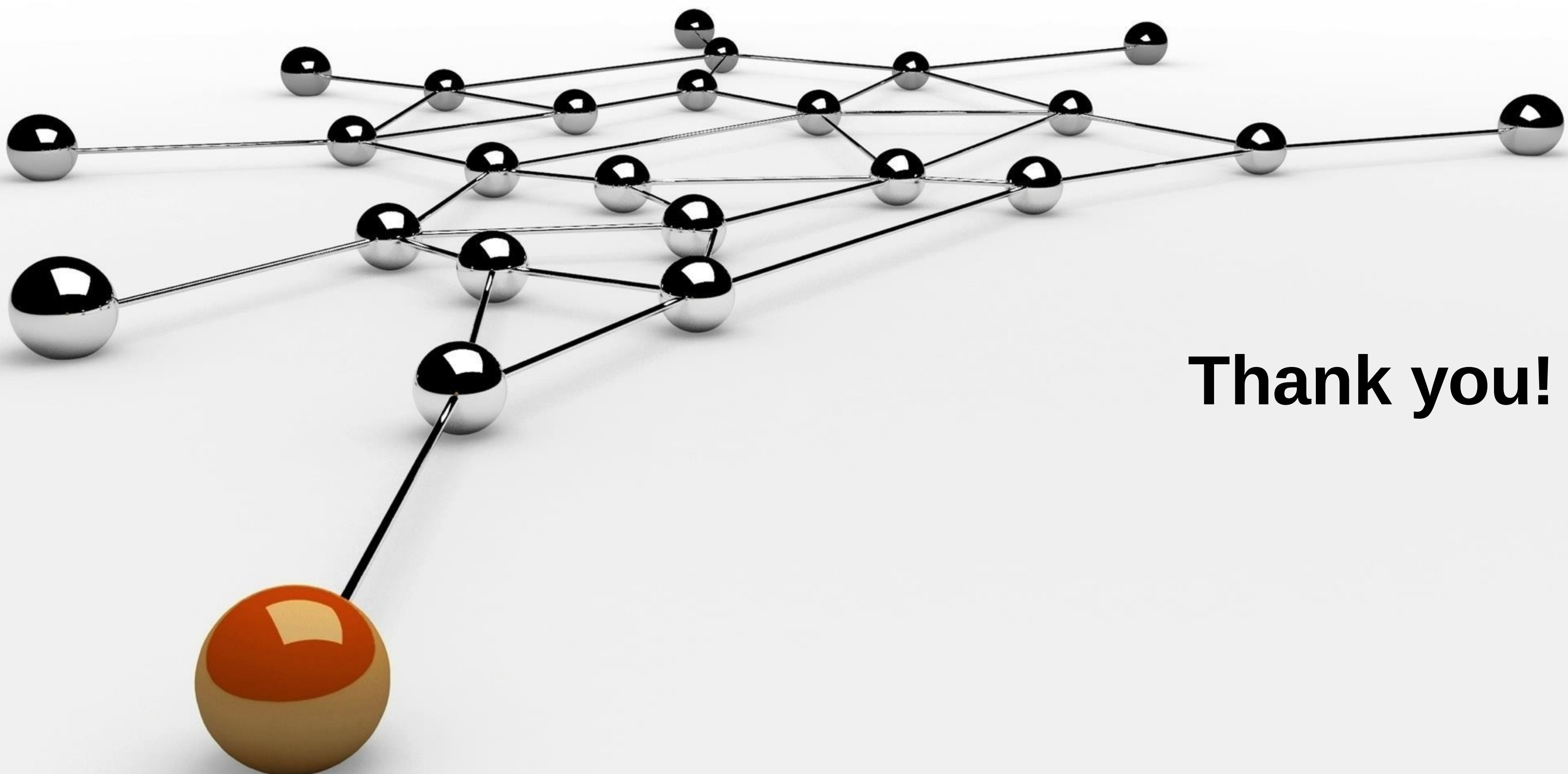
Books:

- Understanding the Linux Kernel, D. P. Bovet and M. Cesati, O'Reilly & Associates, 2000.
- Linux Core Kernel – Commentary, In-Depth Code Annotation, S. Maxwell, Coriolis Open Press, 1999.
- The Linux Kernel, Version 0.8-3, D. A Rusling, 1998.
- Linux Kernel Internals, 2nd edition, M. Beck et al., Addison-Wesley, 1998.
- Linux Kernel, R. Card et al., John Wiley & Sons, 1998.
- Linux Device Drivers, 3rd Edition, Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman Published by O’Reilly Media, Inc., 1005

Questions ??

Aricent

Engineering excellence. Sourced.



Thank you!

www.aricent.com