MPS

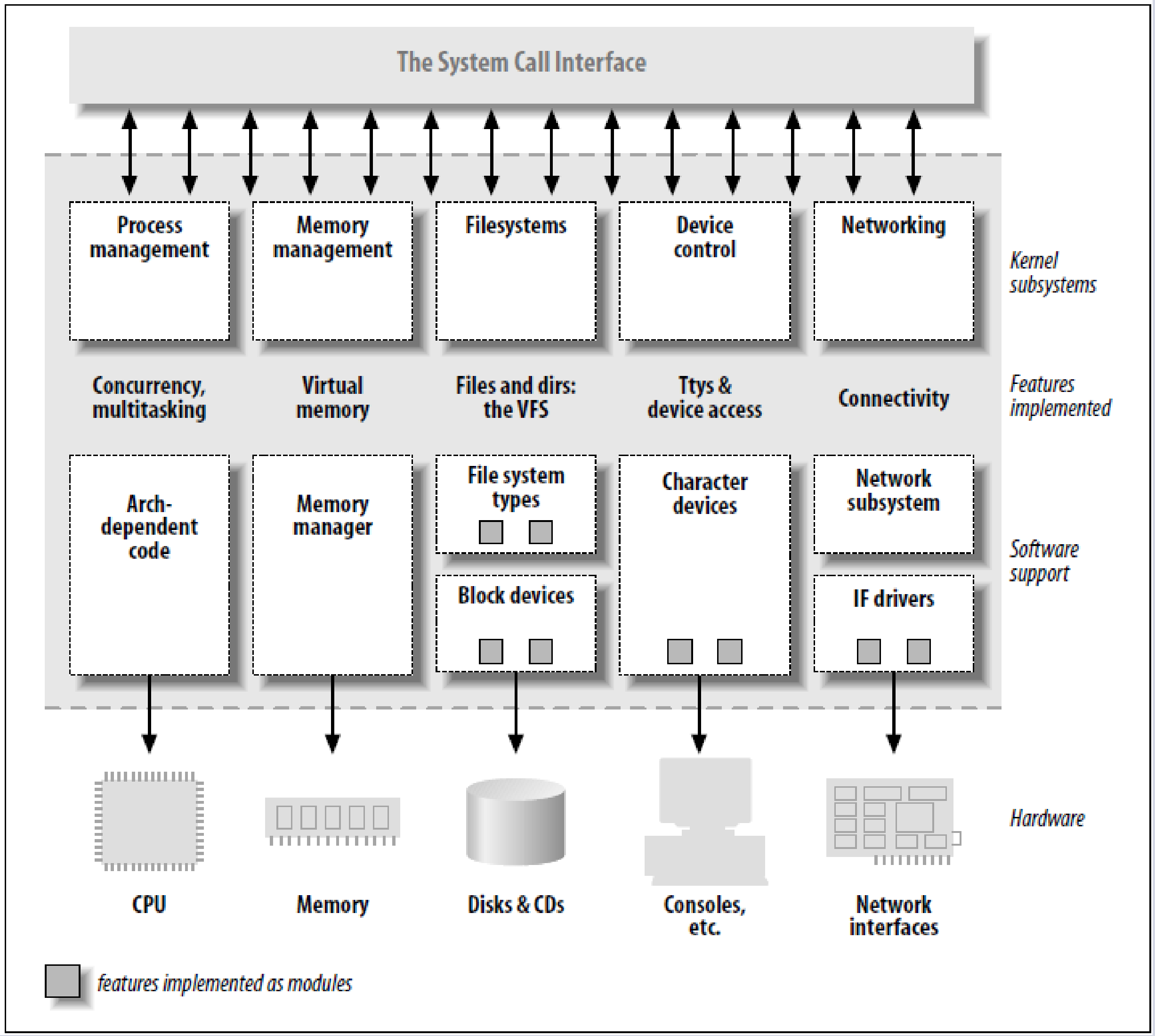
Lesson 05

Linux
Character Drivers

Todays Lesson

12:15		Last Lesson
		Presentation of last exercise
		Char driver intro and Major/Minor Numbers
12:50		Break (10 min)
		Nodes
		File Operations
		Device Registration
13:25		Break (10 min)
		Driver implementation of Fops
13:40		Exercise: GPIO Linux Device Driver

The Linux Kernel



Linux Char Devices

- A device where
  - Data is accessed as a stream
  - Random read/write to different areas is not required
- Examples
  - Serial ports (RS-232/RS-485), GPIO, A/D Converter, etc
  - Devices not being memory devices or network interfaces

ASR33 Teletype
Origin of the abbreviation tty

/dev/

```
root@DevKit8000:~# ls -l /dev
crw-rw----    1 root     root     252,   0 Jan  1  1970 DW9710
crw-------    1 root     root       5,   1 Jan 27 10:20 console
crw-rw----    1 root     video     29,   0 Jan  1  1970 fb0
```

Type "tty" to get your current terminal

Major Numbers

- Says nothing about the functionality
- Can be allocated statically in the driver or dynamically during module insertion
- Character / Block Devices has each their set of numbers
- Typically one major number per ***driver***

Minor Numbers

- Specifies the device that uses the driver
- *One minor number per* ***device***
- Examples
  - Terminal driver
    - Major Number: 4
    - Minor Numbers: 0-xx for tty0, tty1…ttyS2 etc
    - The minor numbers may cover different chips
  - A/D Converter 8-channel (one chip!) driver
    - Major Number: 63
    - Minor Numbers: 0-7
    - Gives us to one (logical) device per channel
    - Adding a chip, could be implemented by adding a major number or just by adding minor numbers.
- Gives us a unified interface to hardware

Major / Minor Numbers

ads7870driver.c

MAJOR = 63
MINOR = 0

```
root@DevKit8000:~# ls -l /dev
crw-rw----    1 root    root    63, 0 Jan  1  1970 ADS7870
```

Hand-On Exercise

What Major / Minor Numbers are assigned to the Serial Ports on the Add-on board (ttyS0-ttyS2)?

Static Allocation of Major Numbers

- Drivers for non-general platforms, can use statically allocated numbers
  - This can be the case for a small embedded platforms where you have control of all drivers
- Example:

```
#define ADS7870_MAJOR 64
#define ADS7870_MINOR  0
#define ADS7870_CH     8   // 8-ch ADC
...
// Allocating Device Numbers
devno = MKDEV(ADS_7870_MAJOR, ADS7870_MINOR);
err   = register_chrdev_region(devno, ADS7870_CH,
                            “ads7870”);
// Freeing Device Numbers
unregister_chrdev_region(devno, ADS7870_CH);
```

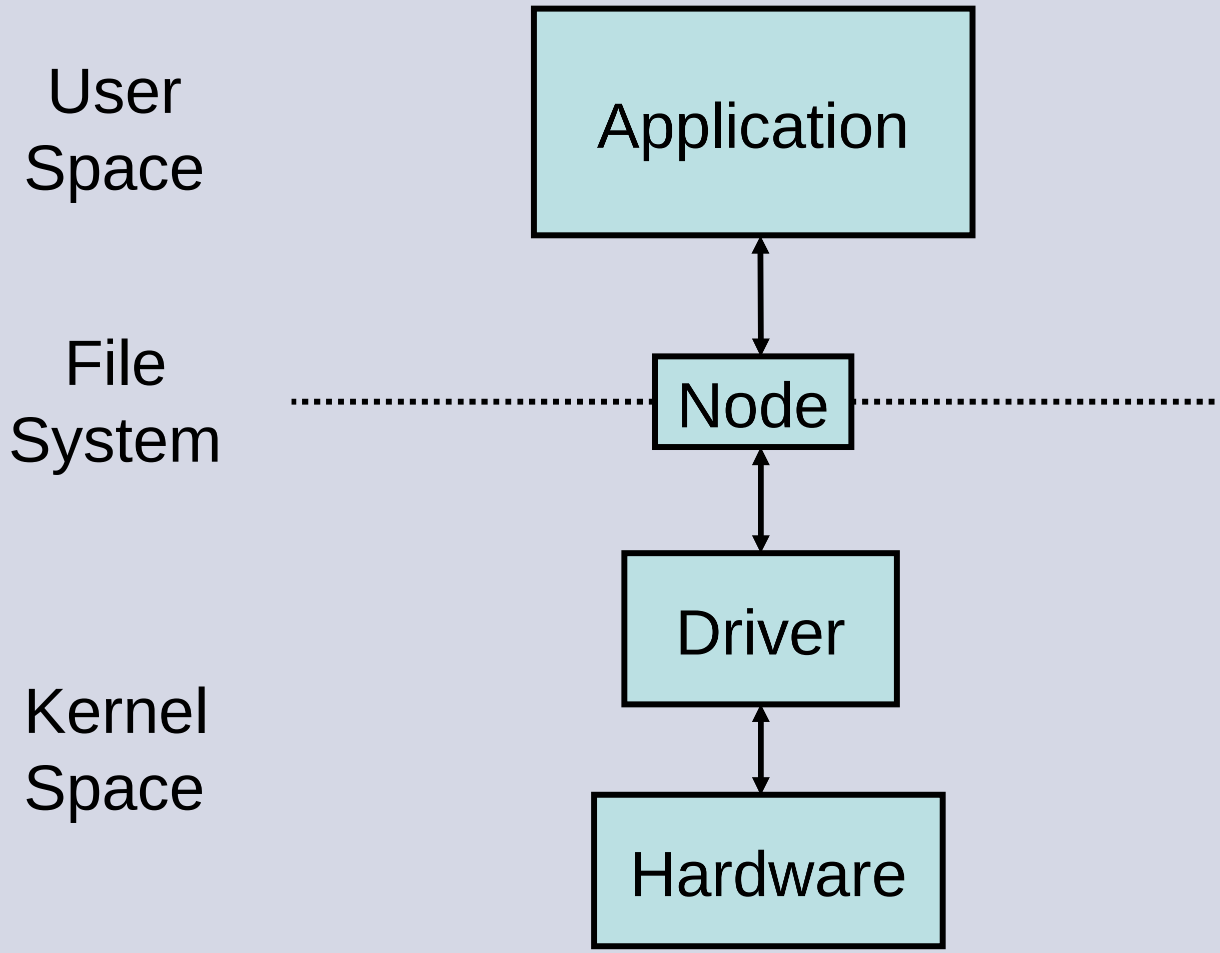
Dynamic Allocation of Major Numbers

- Drivers for general platforms, ex a PC **must** use dynamically allocated numbers
- Example:

```
#define ADS7870_MINOR  0
#define ADS7870_CH     8   // 8-ch ADC
...
// Allocating Device Numbers
err = alloc_chrdev_region(pDev, ADS7870_MINOR,
                          ADS7870_CH, ads7870”);

// Freeing Device Numbers
unregister_chrdev_region(devno, ADS7870_CH);
```

Nodes



- To let the driver appear in the file system, you have to create nodes.

Making Nodes

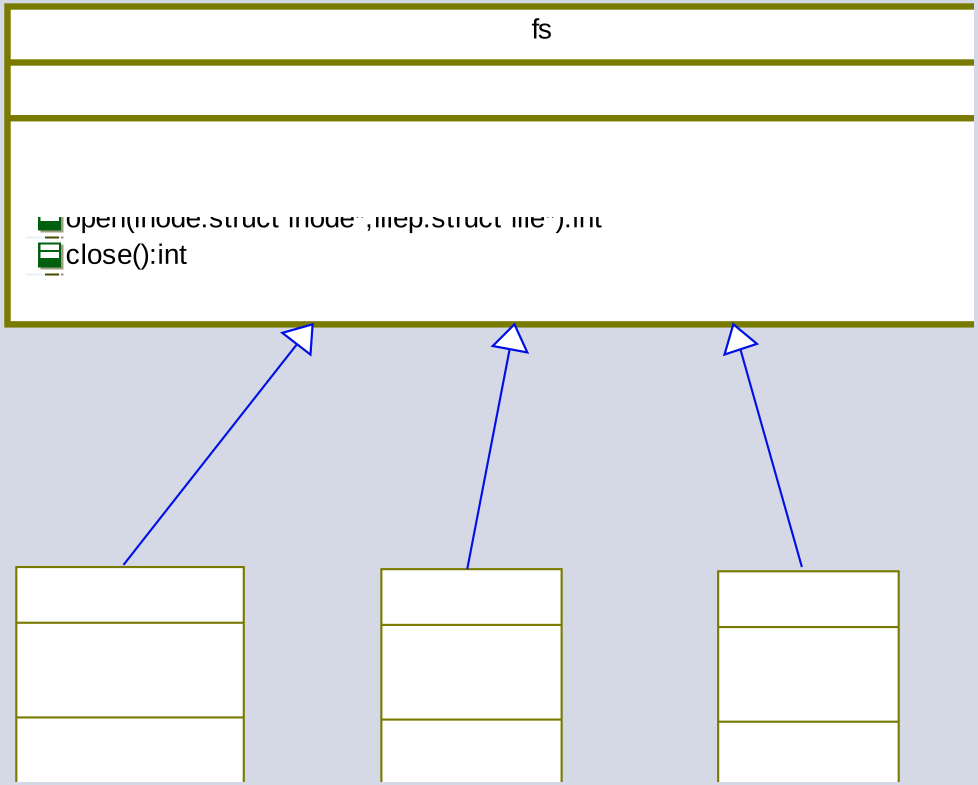
- Examples
  - Driver with statically bound numbers:

```
root@DevKit8000:~# mknod /dev/adc0 c 63 0
```

Desired device name **Char Driver** **Major / Minor**

  - Driver with dynamically assigned numbers:
    - Insert the module with “`insmod`”
    - look up the major number assigned using: “`cat /proc/devices`”
    - Create nodes with “`mknod`” using the major number found
    - See script p.47 in *ldd3*

Linux File Operations (1)



- We must implement the concrete read/write/open/release operations
- Object Oriented C

Linux File Operations (2)

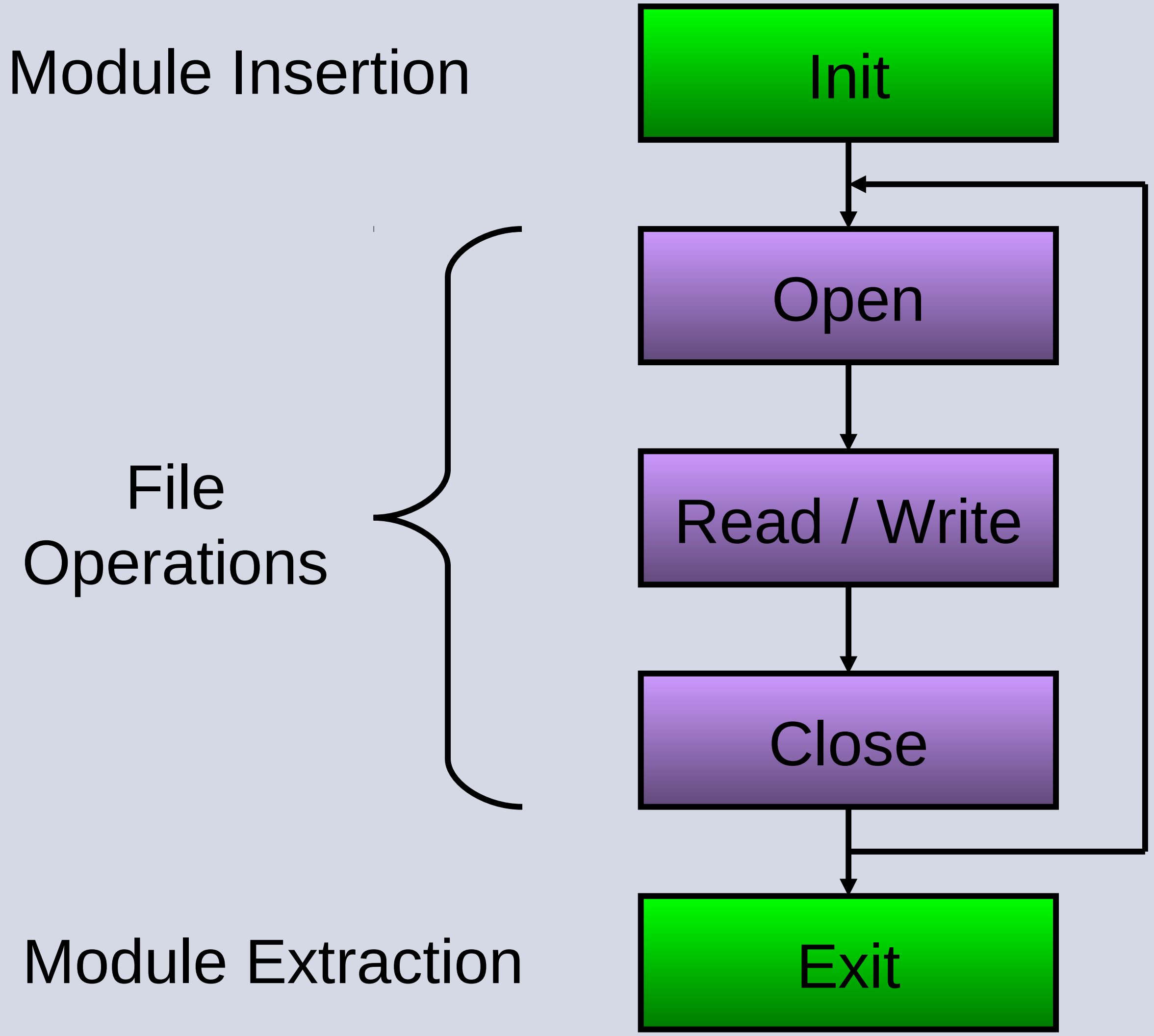
- We must specify our actual implementation of the file system operations:

```
struct file_operations myDriver_fops = {
    .owner = THIS_MODULE,
    .read  = myDriverRead,
    .write = myDriverWrite,
    .open  = myDriverOpen,
    .release = myDriverRelease,
};
...
err = myDriverOpen(struct inode *inode,
                     struct file *filep) {
...
```

attribute

operations

Driver Life-Cycle



Device Registration (1)

- After allocating major / minor numbers, the driver must be registered in the kernel
- Several helper methods are provided in cdev.h
- For a stand-alone c-dev structure use:

```
#include <linux/cdev.h>

Struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

Device Registration (2)

- For the structure to be located in the driver (typical):

```
#include <linux/cdev.h>

static struct cdev my_cdev;
...
devno = MKDEV(...
register_chrdrv_region(...

cdev_init(&my_cdev, &my_fops); // Get c-dev
my_cdev.owner = THIS_MODULE;   // Set c-dev owner
my_cdev.ops = &my_fops;        // Set c-dev file op

err = cdev_add(my_cdev, devno, no_devices);
```

The Open Method

- One of the methods inherited from fs is open
- You must implement the concrete tasks:
  - Check for device not ready errors
  - Initialize device if opened for the first time
  - Allocate memory to be used
  - Check if module is already in use (concurrency)

```
int my_cdev_open(stuct inode *inode, struct *filep){
  if (MAJOR(inode->i_rdev) != my_drv_MAJOR)
    return –ENODEV;
  if (MINOR(inode->i_rdev) != my_drv_MINOR)
    return –ENODEV;
  if (!try_module_get(my_drv_fops.owner))
    return –ENODEV;
  return 0; }
```

The Release Method

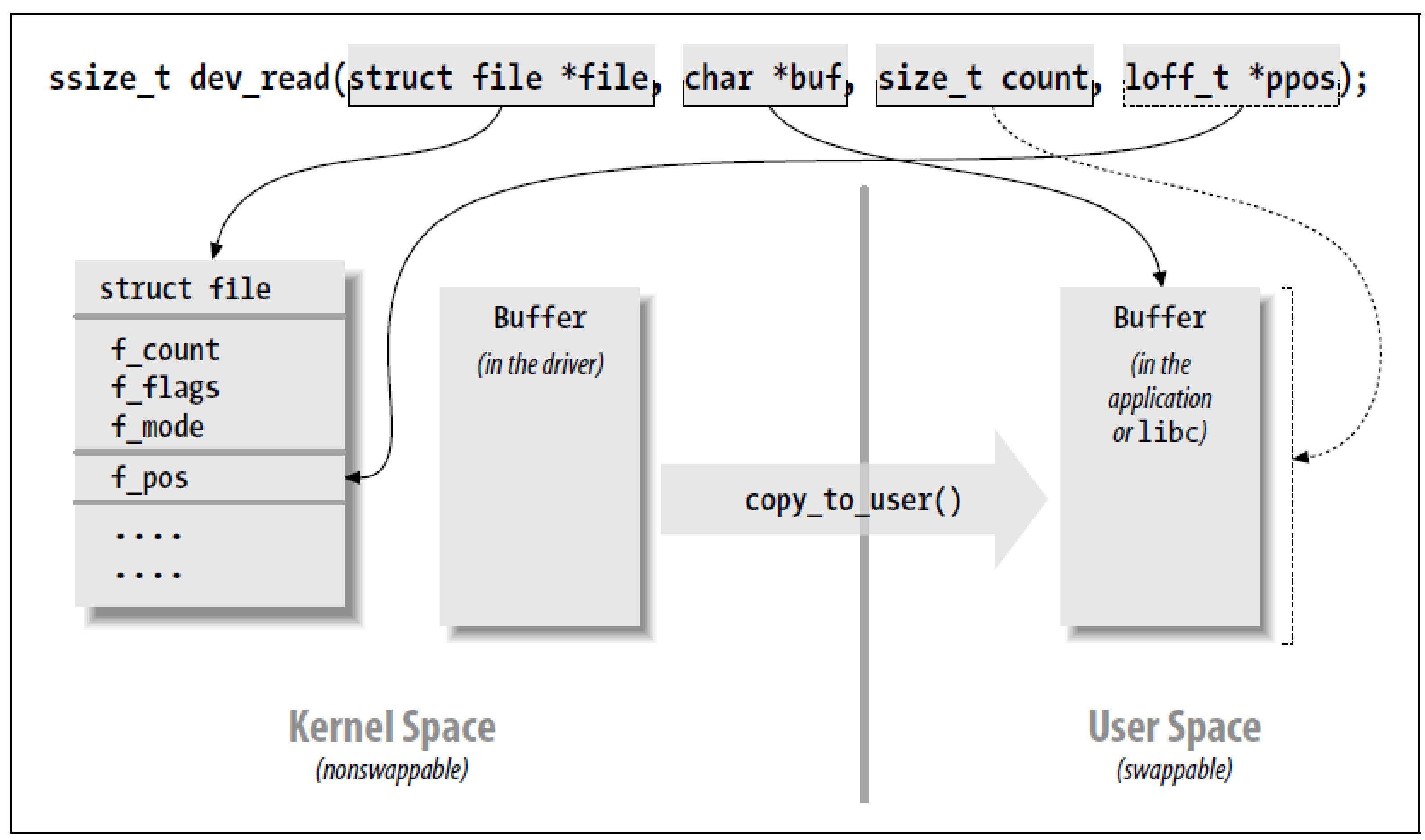
- The Release method is in charge of releasing the module, to make it available for other clients (applications)
- Things to be done during release are:
  - De-allocation of memory allocated during open
  - Shut down the device on the last close
  - Freeing the module semaphore (concurrency)

```
int my_cdev_release(stuct inode *inode,
                     struct *filep){
  <... check major/minor ...>
  module_put(my_drv_fops.owner);
  return 0; }
```

User- / Kernel Space Data (1)

- The kernel- and user space has separate memory locations
- User space has only user access, no access to kernel memory. Trying ➔Page Fault
- User space memory is typically virtual and maybe swappable
- Copy from Kernel- to User Space:
  - unsinged long **copy_to_user**(void __user *to,
    const void *from, unsigned long count);
- Copy from User- to Kernel Space:
  - unsinged long **copy_from_user**(void *to,
    const void __user *from, unsigned long count);

User- / Kernel Space Data (2)



- Note that buf is a char pointer!

The Read Method

- The “read” method is the driver method called, when an application performs a read (ex: cat /dev/my_dev) on the device
- “read” return the number of bytes read, zero if EOF reached or a negative value on errors

```
ssize_t my_cdev_read(struct file *filep,
                  char __user *buf,
                     size_t count,
                     loff_t *f_pos){

  char my_buf[12] = “hello_world”;
  char my_buf_len = sizeof(my_buf);
  copy_to_user(buf, my_buf, my_buf_len);
  *f_pos += my_buf_len;
return my_buf_len; }
```

requested size

The Write Method

- The “write” method is the driver method called, when an application performs a write (ex: echo “hello” > /dev/my_dev) to the device
- “write” return the number of bytes written, zero if none written or a negative value on errors

```
ssize_t my_cdev_write(struct file *filep,
                      char __user *buf,
                      size_t count,
                      loff_t *f_pos){
  // my_buf is a local variable
  copy_from_user(my_buf, buf, count);
  *f_pos += count;
return count; }
```