Linux Device Drivers

- Ivan Gomez Castellanos
- Marc Christian Jolibois
- Victor Rodriguez
- Hector Barajas

Agenda

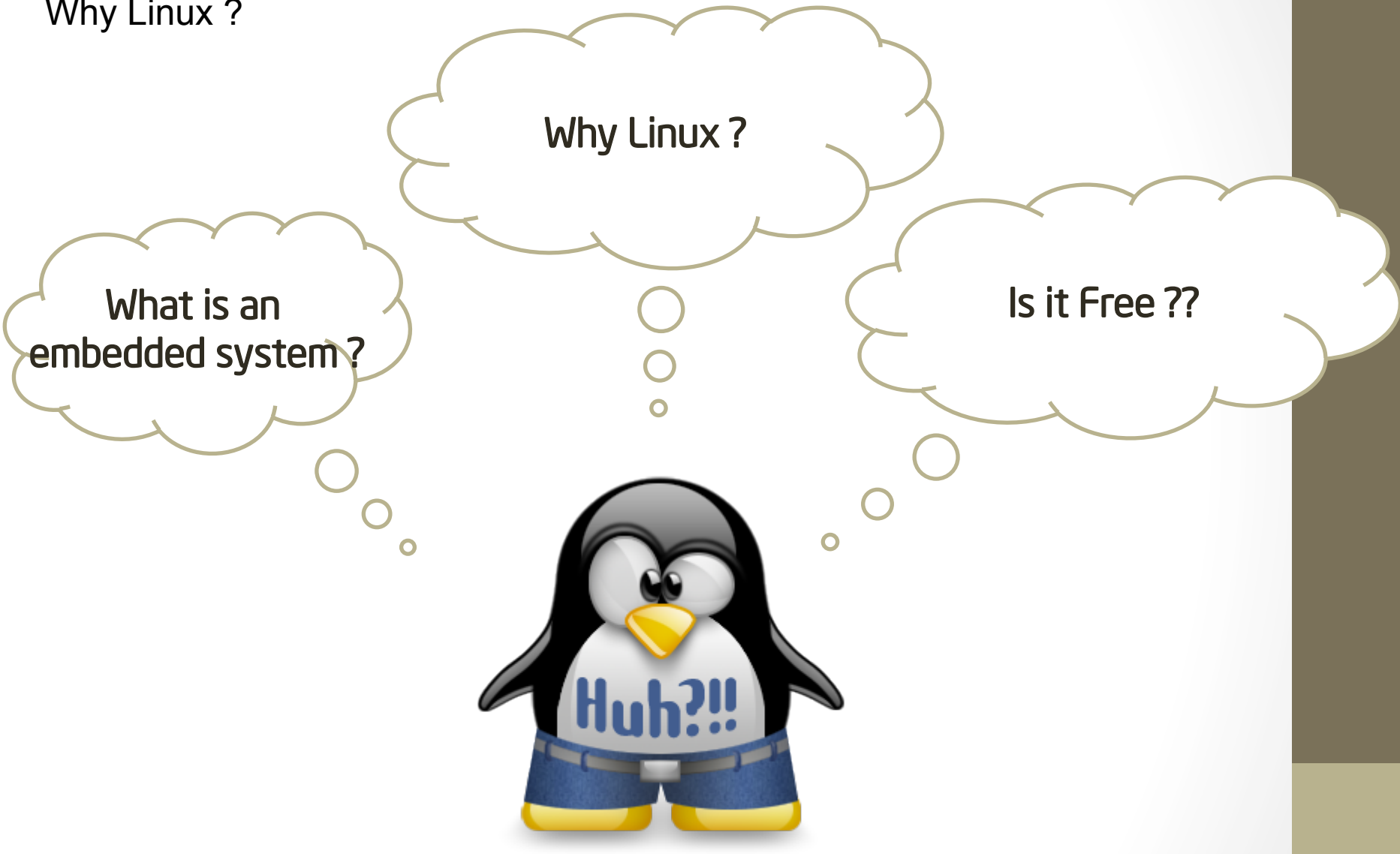
- Linux Introduction
  - Kernel Features
  - Types of drivers
  - System Calls
  - Modules
  - Lab: “Hello World”
- Char Drivers
  - File Operation(Read/Write/ Open/Close)
  - Lab : Create a Character Driver that can copy from user space and to user space
- Interruptions
- Timers
  - Lab: Timers on dmesg
- Communication with HW

Linux Introduction

Agenda

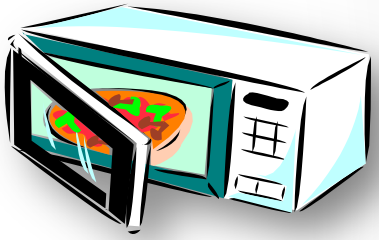
- Why Linux ?
- Kernel Features
- Embedded System Anatomy
- User Space / Kernel Space
- Types of drivers
- System Calls
- Modules
- Lab: “Hello World”

Why Linux ?



- What is an embedded system?

An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is specifically designed for a particular function.



Embedded or not ?

- Contains a processing engine
- Typically design for a specific application
- Includes a simple user interface
- Resource Limited
- Might have power limitations
  Not general purpose computing platforms
- Software built in
- Applications without human intervention

Why Linux ?

Why Linux ?

- Linux supports vast variety of hardware devices
- Linux support a huge variety of applications and networking protocols
- Linux is scalable, from microwave to large switches and routers
- Linux is Free ! Really ? Well kind of …
- Linux has a huge number of active developers
- Increasing number of HW/SW vendors support Linux

Is it free ???

- Linux Kernel is licensed under the terms of the GNU GPL
  - (Generic Public License)

"When you speak of free software , we are referring to freedom not price"

The license is **self-perpetuating**

The license grants the user freedom to **run** the program

The license grants the user the right to **study** and modify the source code

The license grants the user the permission to **distribute** the original code and his modifications

The license is **viral**, it grants these same rights to anyone to any whom you distribute GPL software

Free Vs Freedom

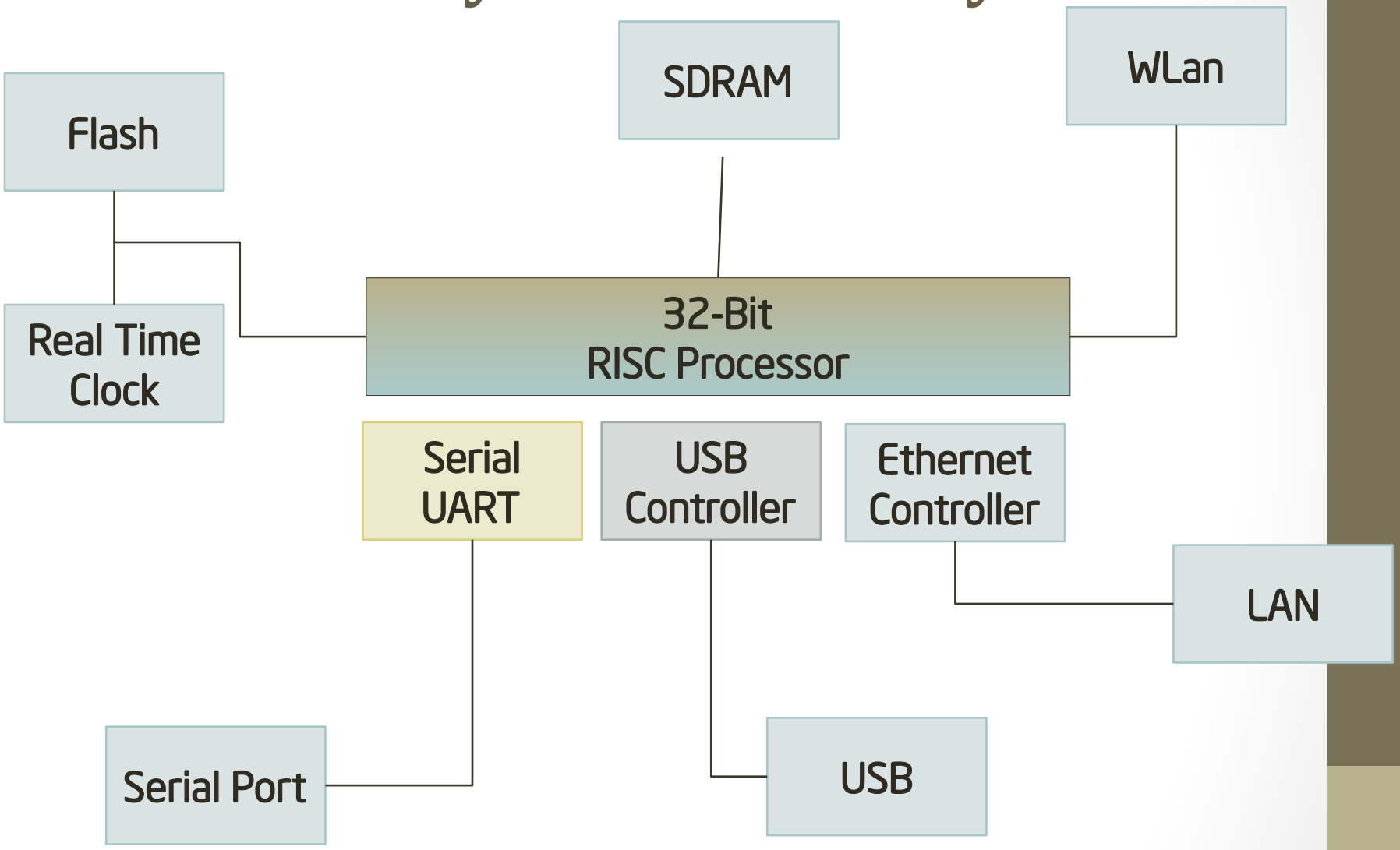
- Free as freedom
  - Download
  - Use
  - Study
  - Change
- Free as in beer
  - Integration
  - Maintenance
  - Support

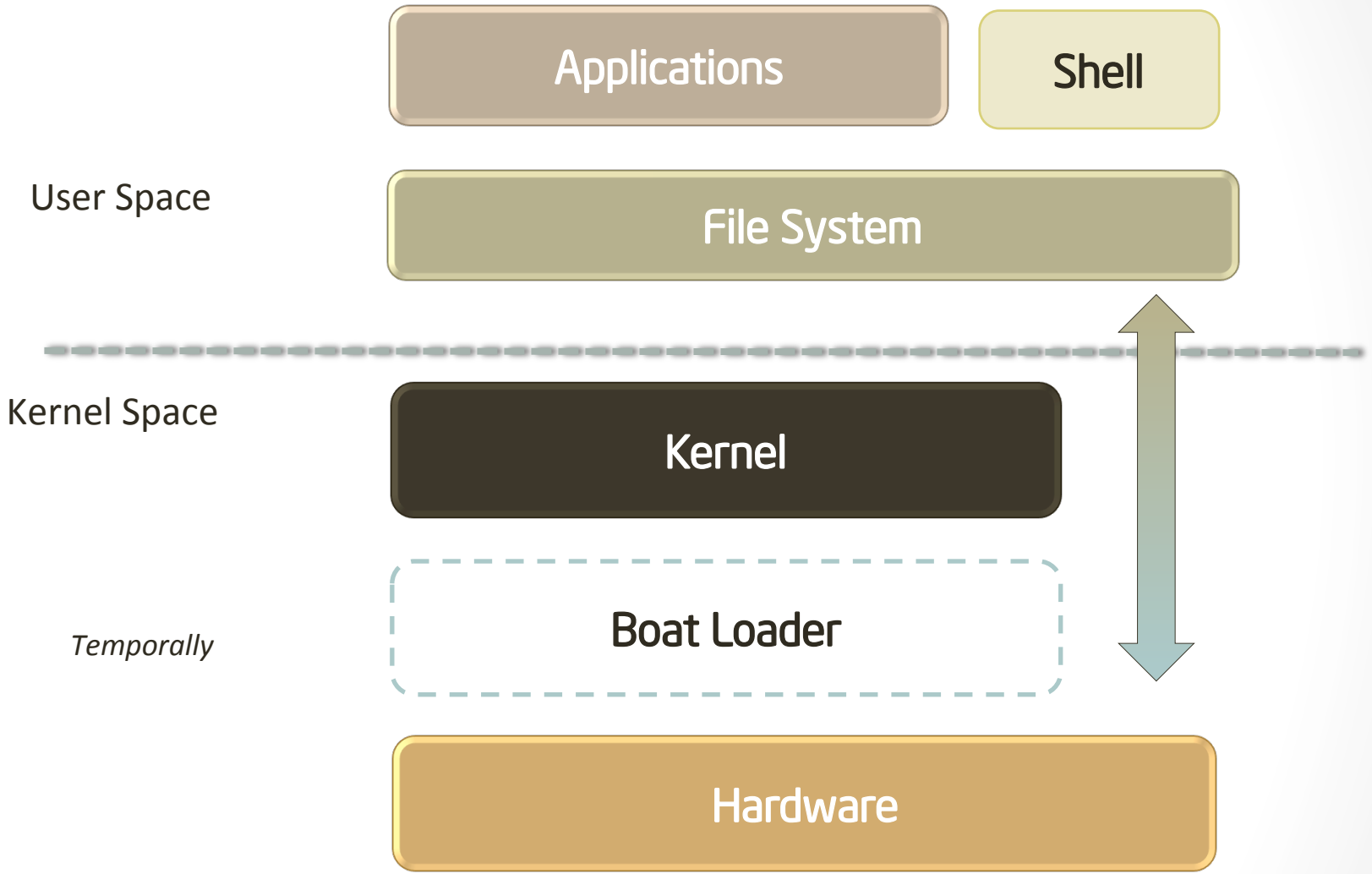
Kernel Features

- Components of the kernel

Process Management	• Creating and Destroying processes • I/O to Processes • Inter-process communication (IPC) and signals and pies • Scheduling
Memory Management	• Build up a virtual addressing space for all processes • Allocating and freeing up memory • Process interaction with memory
Device Management	• Systems operations map to physical device • Device drivers control operations for virtually every peripheral and HW component
Networking	• Networking operations are not process specific • Incoming packets are asynchronous • Processes must be put to sleep and wake for network data

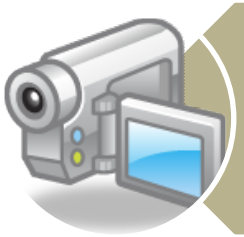
Embedded System Anatomy




Applications
Shell
User Space
File System
Kernel Space
Kernel
Boat Loader
Temporally
Hardware

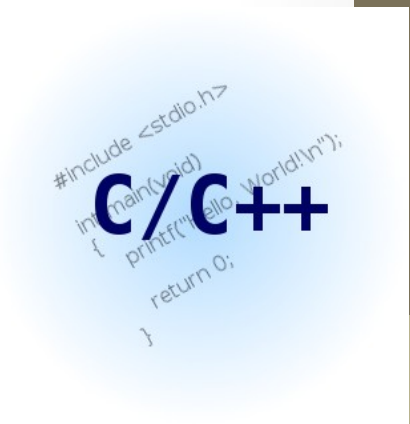
The BIG SW picture

Applications

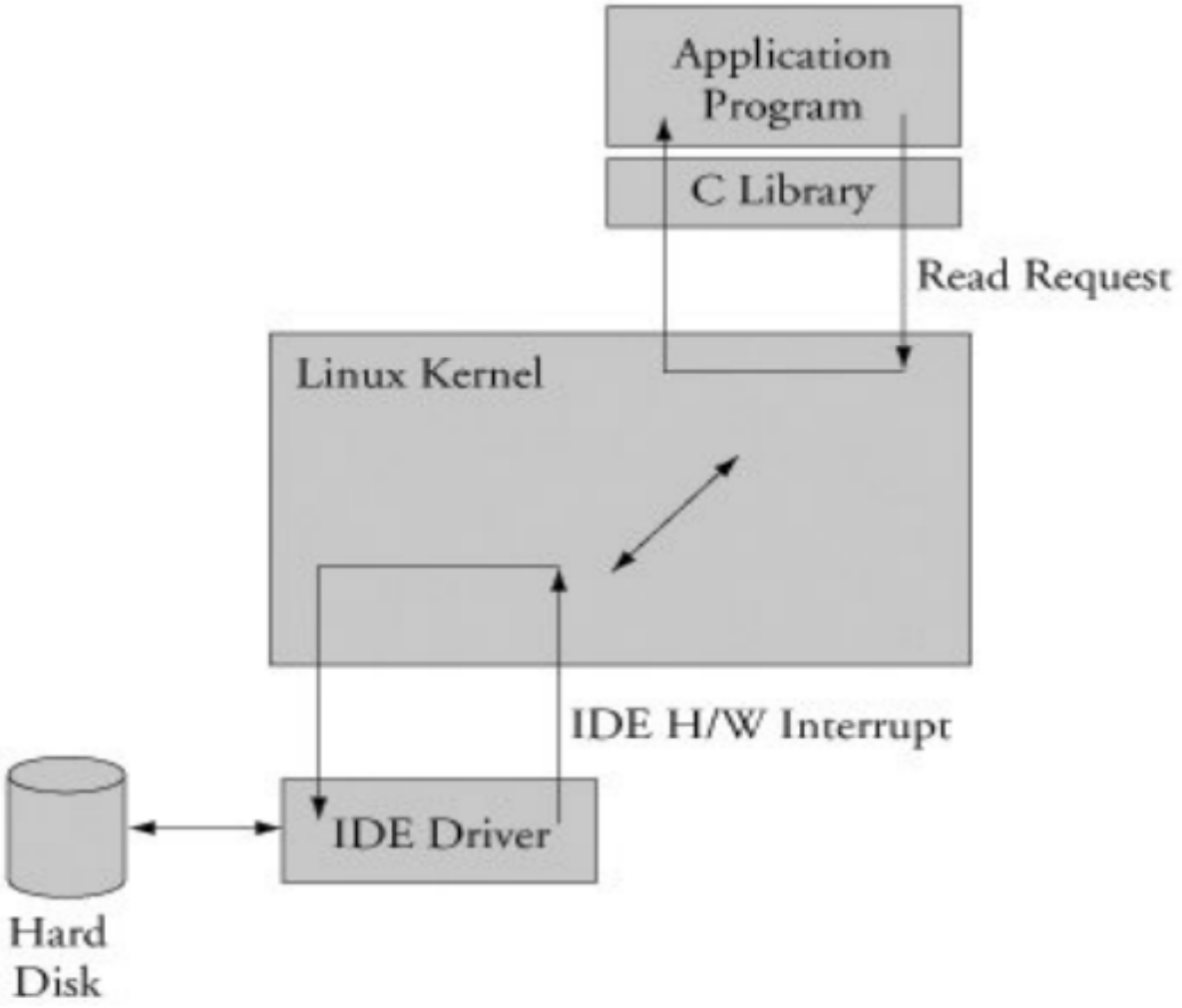


Middleware

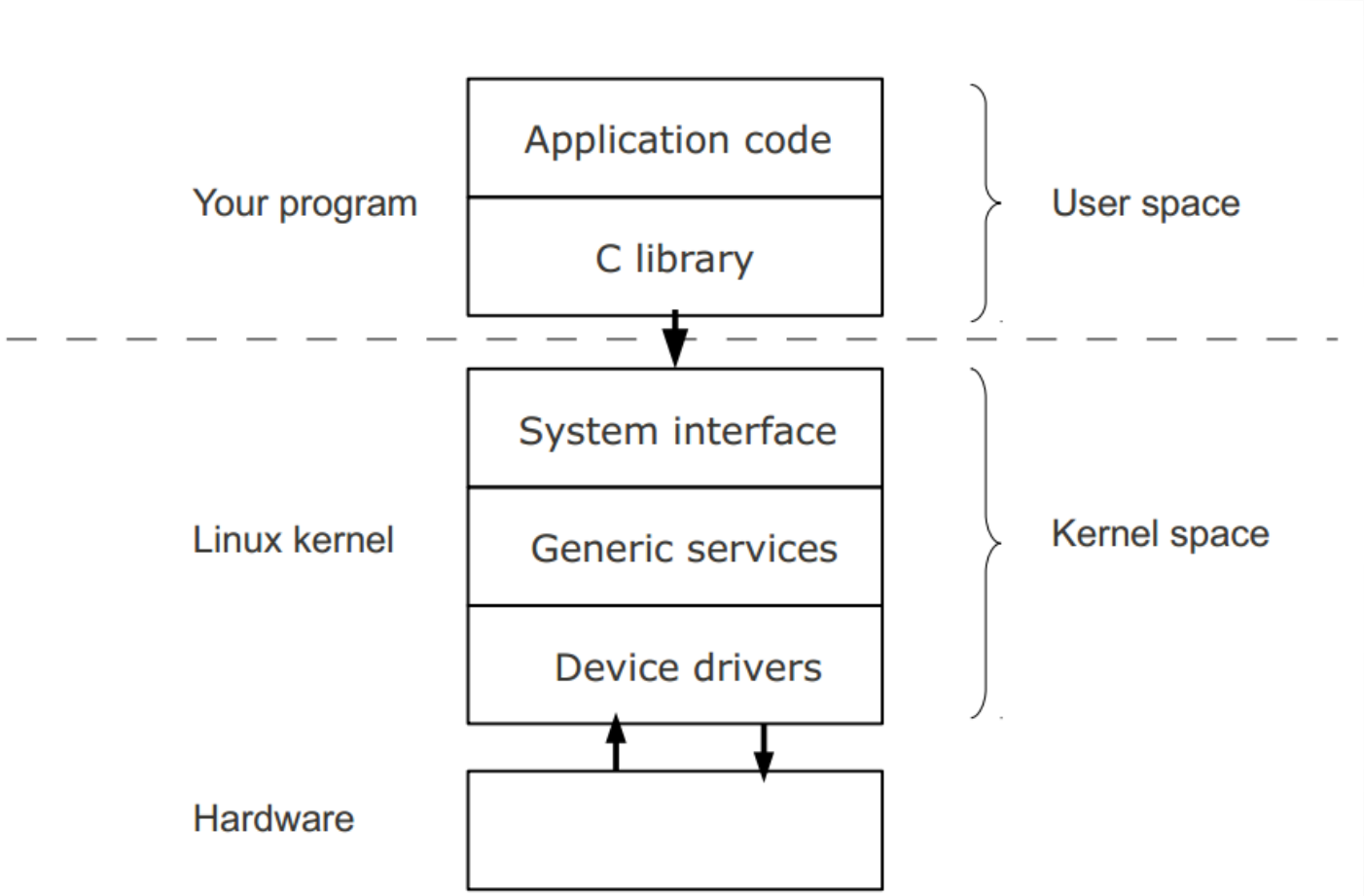
Base Port



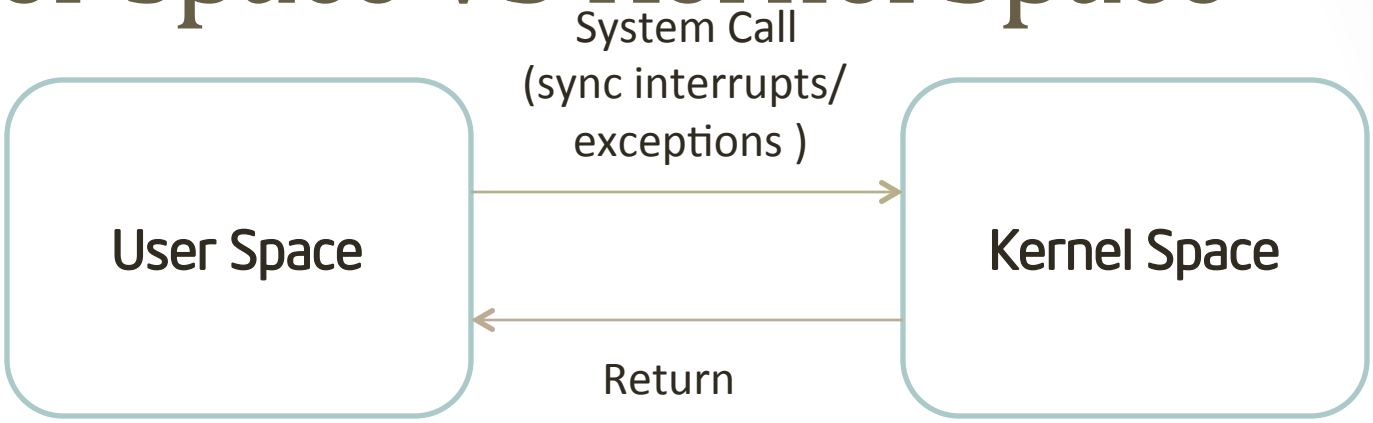
How do we read a file?



User Space / Kernel Space



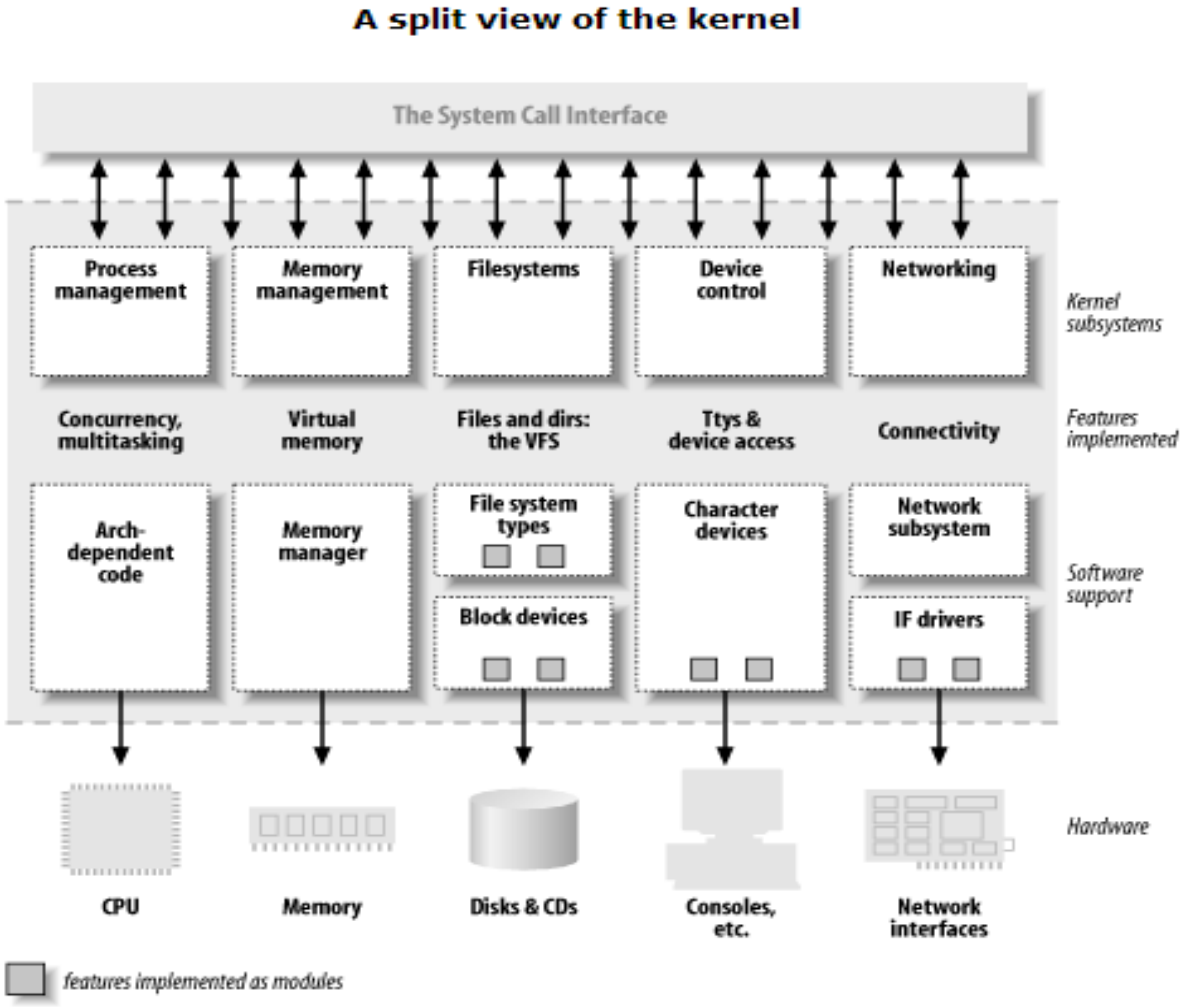
User space VS Kernel Space



- Applications and daemons execute with limited privileges (Ring 3 on x86)
- This is true even if the application has root privileges

- Kernel has direct, privileged access to HW and Memory (Ring 0 on x86) Drivers (and modules) have kernel privileges

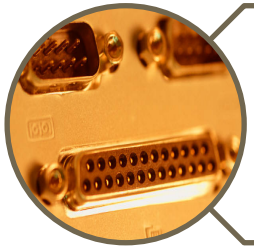
A view of the kernel



The truth about drivers

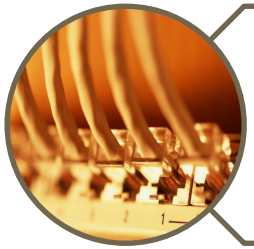
- A device Driver is the lowest level of software as it is directly bound to the hardware features of the device
- Each driver manages one or more piece of HW while the kernel handless process scheduling …
- May be integrated directly into the kernel or loadable modules (not all the modules are device Drivers)
- A driver can be a modular or build in part of the kernel

Drivers ....



Char

Basic Bloc



Network

Character Devices

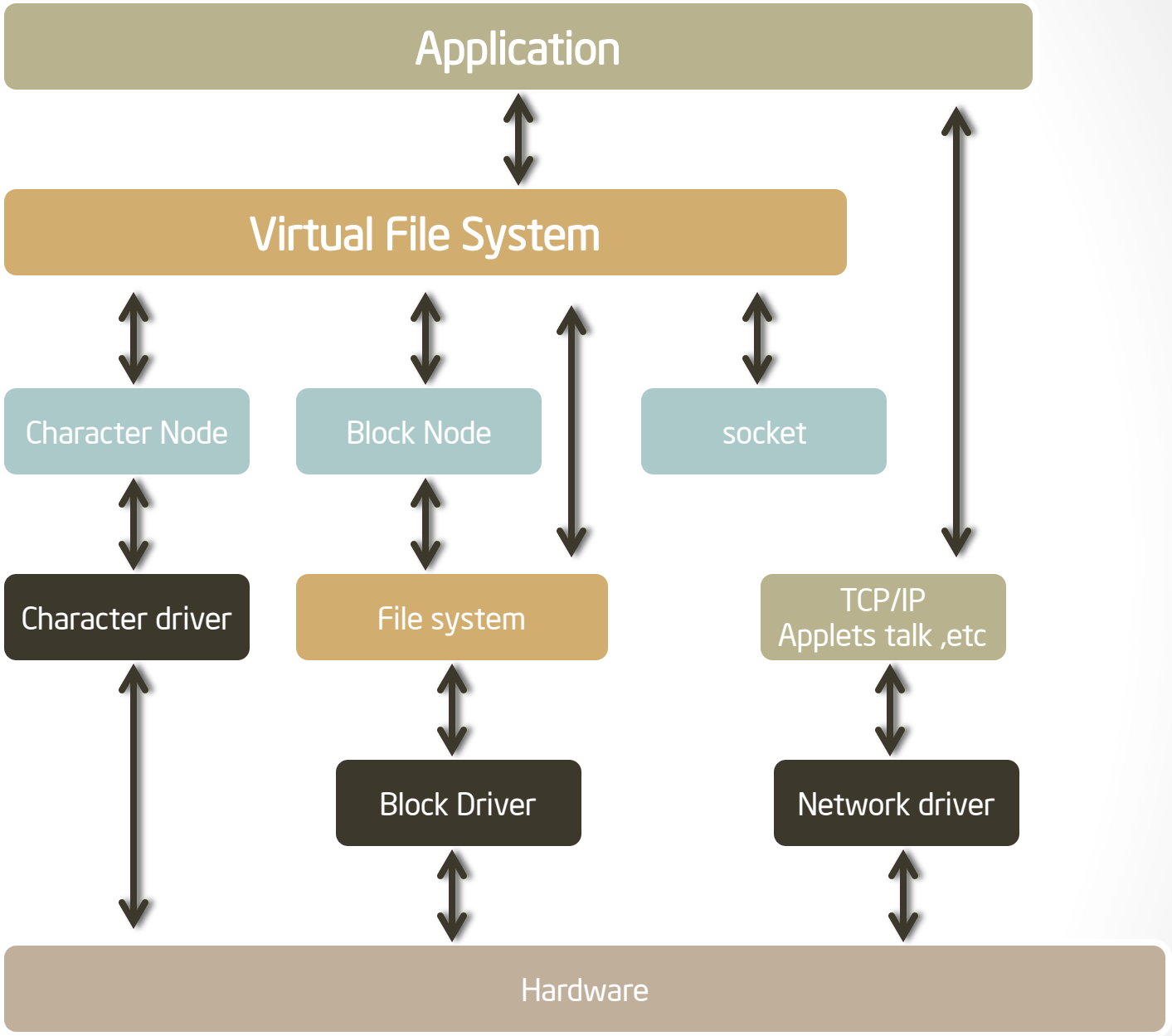
- Can be written to and read from a byte at a time
- Well represented as streams
- Usually permit only sequential access
- Can be considered as files
- Implement : open close read and write functions
- Serial Parallel ports , console (monitor and keyboard) etc
- Examples : /dev/tty0, dev/ttyS0 ..

Block Devices

- Can be written to and read from only in block size multiples; access is usually cached
- Permit random access
- File systems can be mounted on these devices
- In Linux block devices can behave like character devices(transferring a block of bytes)
- Hard drives, cd rooms, etc
- Examples : /dev/sda1./dev/fd0

Network Devices

- Transfer packet of data. Device sees the packets ,not the streams.
- Most often accessed via the BSD socket interface
- Instead of read, write the kernel calls caket reception and transmission functions
- Network interfaces are not mapped to the FS: they are identified by a name
- Examples : eth0,ppp0


Application
Virtual File System
Character Node
Block Node
socket
Character driver
File system
TCP/IP
Applets talk ,etc
Block Driver
Network driver
Hardware

How Apps use device drivers

- User Applications interact with peripheral devices using the same basic system calls

We will forget about network device drivers, because well ….
They are not reached trough file system entries ,
So lets focus on character drivers

For each one of the limited number of these system calls there is a corresponding **entry point** in the driver

release()	read()	write()	write()

System calls (important facts)

- A system call is a request for service that a program makes of the kernel
- There are functions in the GNU C Library to do virtually everything that system calls do.
- And if I want to do it by myself ?
- The GNU C Library provides the *syscall* function !!!!!!!!

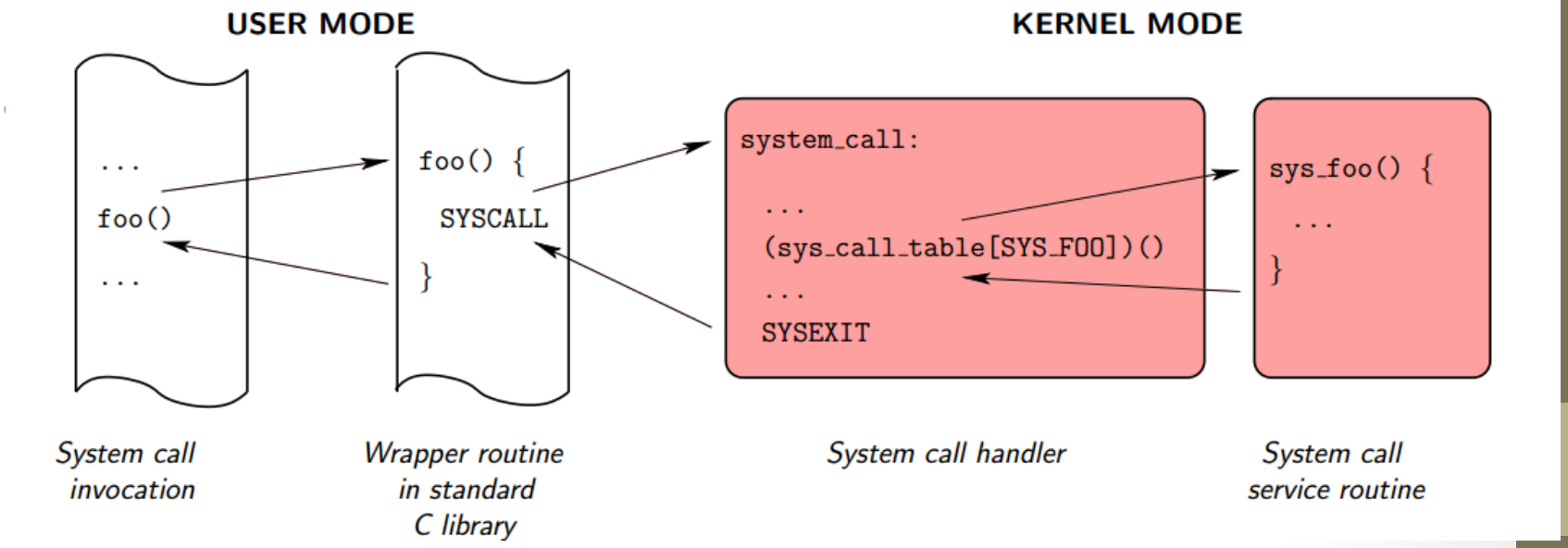
Note that there are other kinds of functions that may exist in a driver, which are not directly reached by user system calls

- Interrupts
- Timed tasks
- Power management

One a driver is loaded, its methods are all registered with the kernel and its event driven

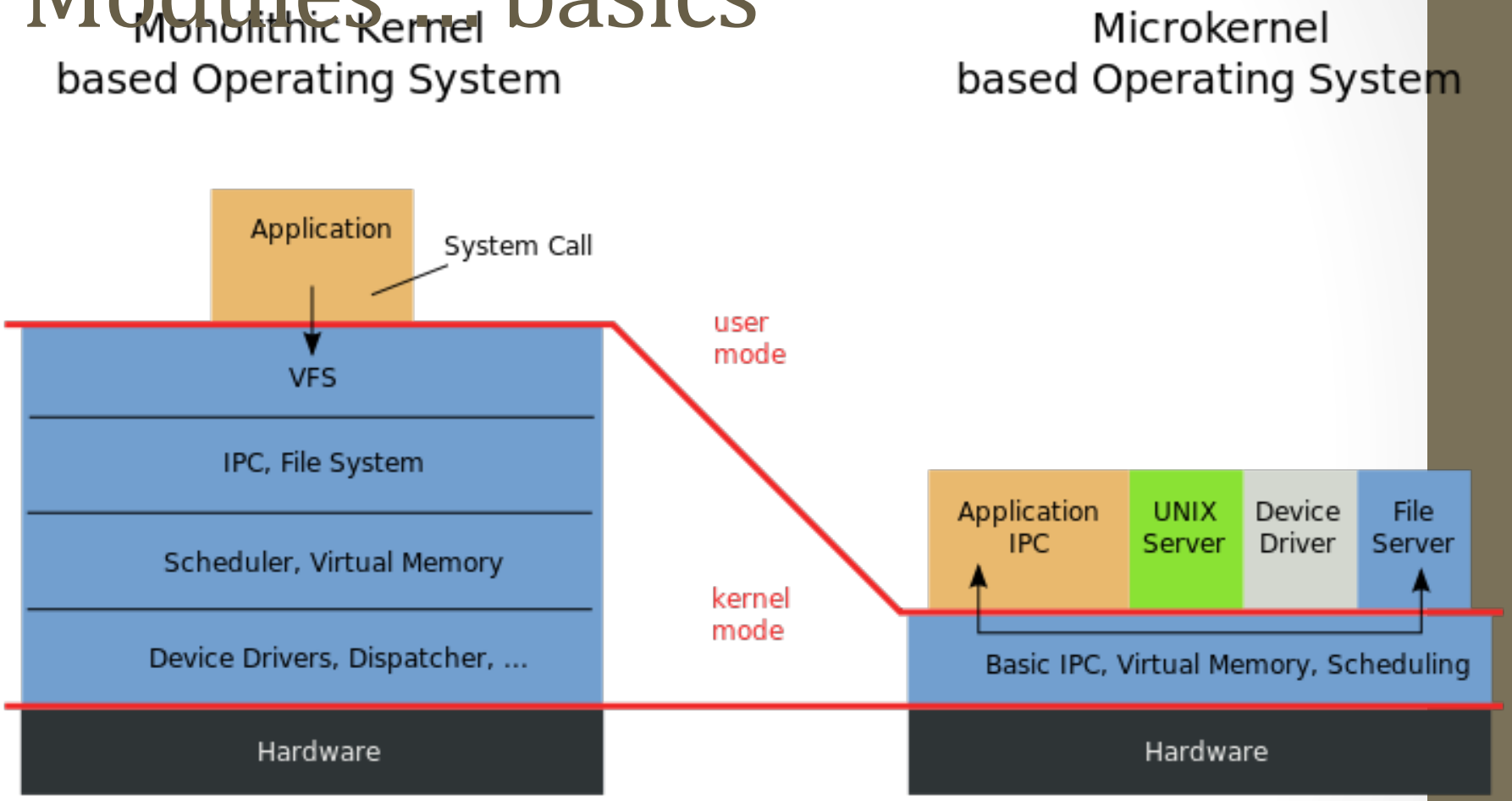
Walking through the system call

1. syscall is declared in unistd.h
2. Function: long int **syscall** (*long int sysno, ...*)
   - *sysno* is the system call number. Each kind of system call is identified by a number. Macros for all the possible system call numbers are defined insys/syscall.h
   -



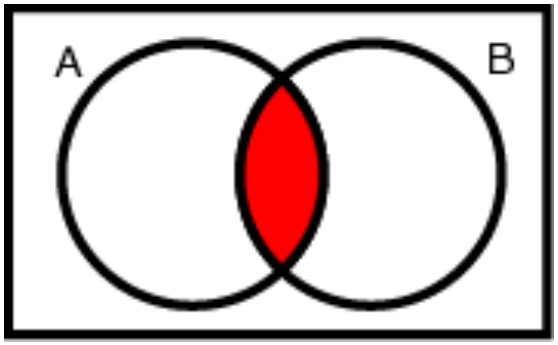
```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
int main(void) {
        long ID1, ID2;
        /*-----------------------------*/
        /* direct system call          */
        /* SYS_getpid (func no. is 20) */
        /*-----------------------------*/
        ID1 = syscall(SYS_getpid);
        printf ("syscall(SYS_getpid)=%ld\n", ID1);
        /*-----------------------------*/
        /* "libc" wrapped system call  */
        /* SYS_getpid (Func No. is 20) */
        /*-----------------------------*/
        ID2 = getpid();
        printf ("getpid()=%ld\n", ID2);
        return(0); }
```

Modules... basics

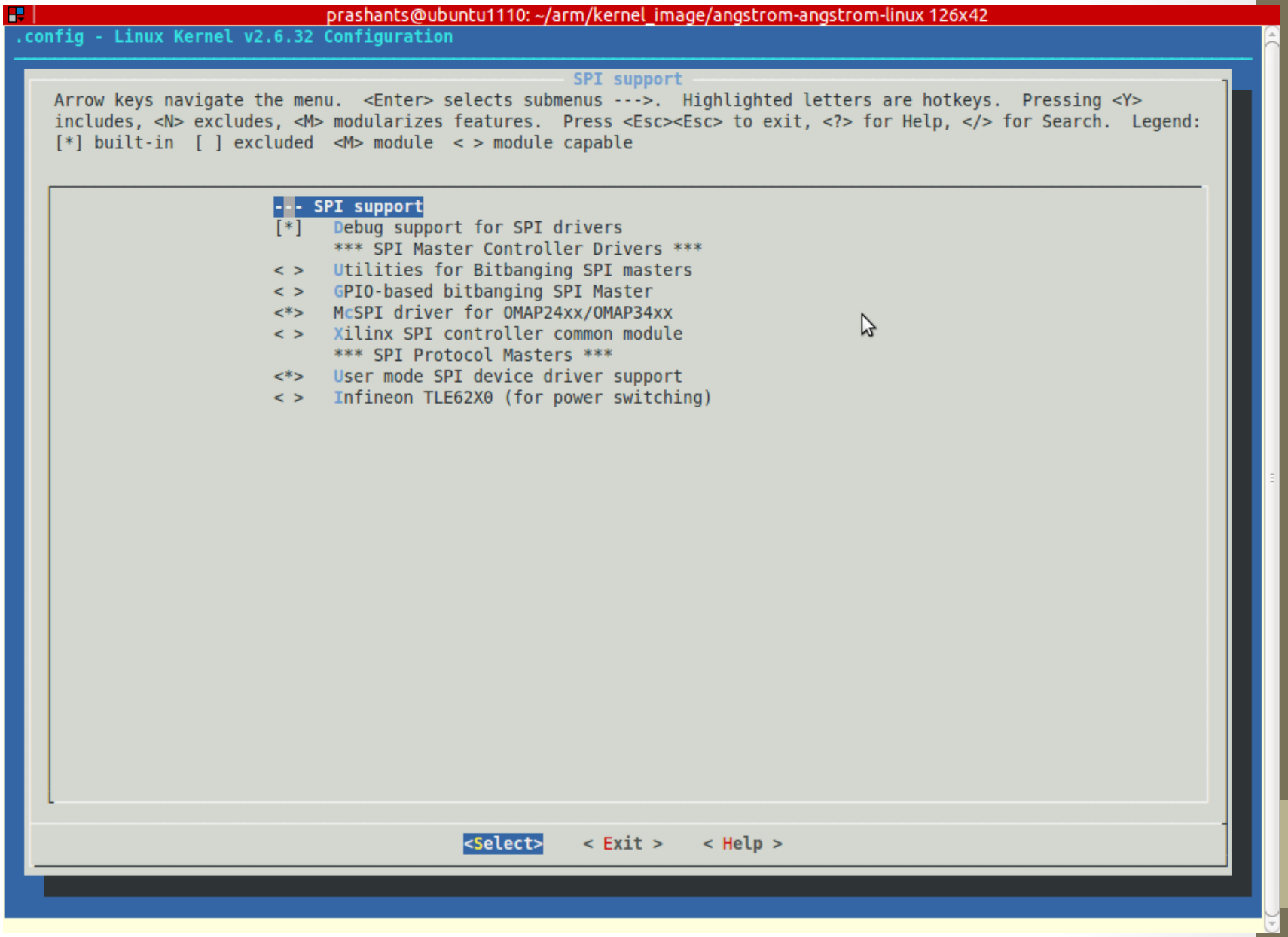


- Is a component that can be loaded/ unloaded into/from an already running kernel
- Once loaded a module has all the capabilities of any other pat of the kernel
- Modules use the kernel as a shared library

modules

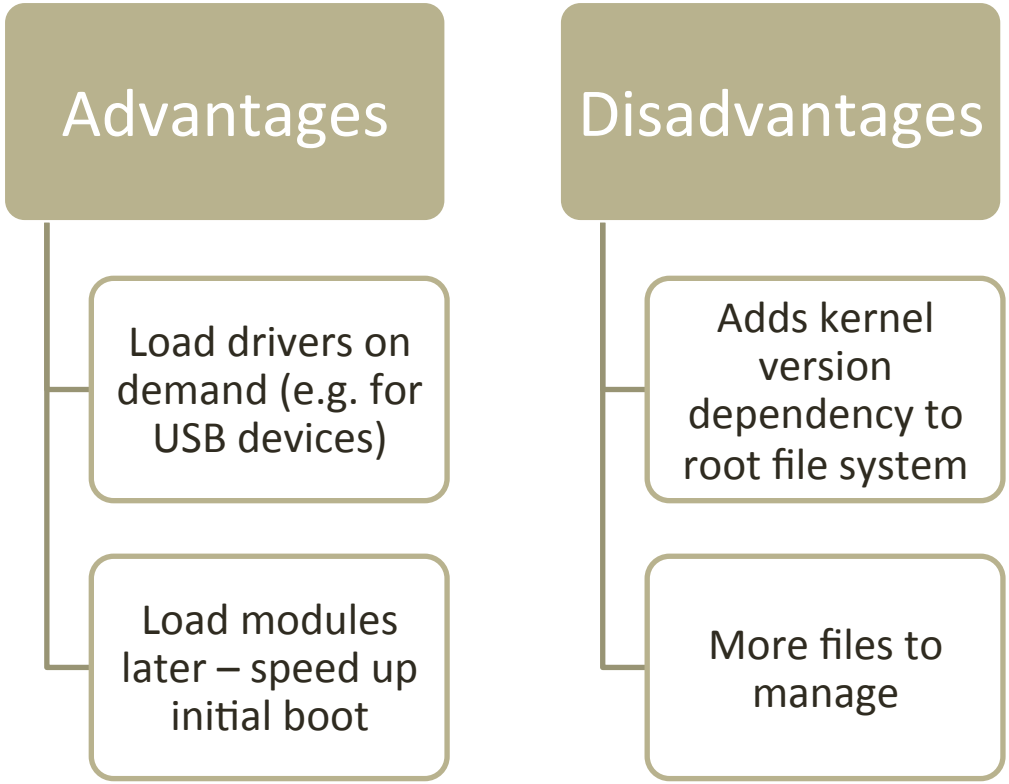


Drivers
A device driver may be built in to the kennel


prashants@ubuntu1110: ~/arm/kernel_image/angstrom-angstrom-linux 126x42
.config - Linux Kernel v2.6.32 Configuration
SPI support
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable
--- SPI support
[*] Debug support for SPI drivers
*** SPI Master Controller Drivers ***
< > Utilities for Bitbanging SPI masters
< > GPIO-based bitbanging SPI Master
<*> McSPI driver for OMAP24xx/OMAP34xx
< > Xilinx SPI controller common module
*** SPI Protocol Masters ***
<*> User mode SPI device driver support
< > Infineon TLE62X0 (for power switching)
<Select> < Exit > < Help >

Modules or drivers ?

Kernel code that is loaded after the kernel has booted



Finally, the Hello World!!!!!

```
/*
 *  hello-1.c - The simplest kernel module.
 */
#include <linux/module.h>         /* Needed by all modules */
#include <linux/kernel.h>         /* Needed for KERN_INFO */

int init_module(void)
{
        printk(KERN_INFO "Hello world 1.\n");

        /*
         * A non 0 return means init_module failed; module can't be loaded.
         */
        return 0;
}

void cleanup_module(void)
{
        printk(KERN_INFO "Goodbye world 1.\n");
}
```

Is that all ?

Kernel modules must have at least two functions:

init_module

"start" (initialization) function called

- Is called when the module is insmoded into the kernel
- Registers a handler for something with the kernel
- It replaces one of the kernel functions with its own code (usually code to do something and then call the original function)

cleanup_module

"end" (cleanup) function called () which

- Is called just before it is rmmoded.
- Undo whatever init_module() did, so the module can be unloaded safely.

Introducing printk()

printk() is similar to C function printf() but has some important differences:

Typical invocation:

- printk(KERN_INFO “Hi there ”);

printk() has no floating point format invocation

Every message in printk() has a log level /usr/src/linux/include/linux/printk.h

If you don't specify a priority level, the default priority, DEFAULT_MESSAGE_LOGLEVEL, will be used.

There are 8 log levels that can be specified in a printk

```
#define KERN_EMERG "&lt;0>"
#define KERN_ALERT "&lt;1>"
#define KERN_CRIT "&lt;2>"
#define KERN_ERR "&lt;3>"
#define KERN_WARNING "&lt;4>"
#define KERN_NOTICE "&lt;5>"
#define KERN_INFO "&lt;6>"
#define KERN_DEBUG "&lt;7>"
```

There are 4 values that define what gets printk'd where:

```
extern int console_printk[];
#define console_loglevel
(console_printk[0])
#define default_message_loglevel
(console_printk[1])
#define minimum_console_loglevel
(console_printk[2])
#define default_console_loglevel
(console_printk[3])
```

Let's compile it !!!

- Documentation in
  - linux/Documentation/kbuild/modules.txt
  - linux/Documentation/kbuild/makefiles.tx

```
obj-m += hello-1.o
all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Tabs

Kernel Module utilities

insmod

- Loads the module code and data into the kernel, which, in turn, performs a function similar to that of *ld*, in that it links any unresolved symbol in the module to the symbol table of the kernel

modprobe

- It will look at the module to be loaded to see whether it references any symbols that are not currently defined in the kernel. If any such references are found, *modprobe* looks for other modules in the current module search path that define the relevant symbols

rmmod

- Removes module from the kernel

lsmod

- Produces a list of the modules currently loaded in the kernel. *lsmod* works by reading the*/proc/modules* virtual file

Ok that's really old

```
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>                 /* Needed for the macros */

static int __init hello_2_init(void)
{
        printk(KERN_INFO "Hello, world 2\n");
        return 0;
}

static void __exit hello_2_exit(void)
{
        printk(KERN_INFO "Goodbye, world 2\n");
}

module_init(hello_2_init);
module_exit(hello_2_exit);
```

__init__ and __exit macros

kernel 2.2 and later !!!

```
#include <linux/module.h>        /* Needed by all modules */
#include <linux/kernel.h>        /* Needed for KERN_INFO */
#include <linux/init.h>          /* Needed for the macros */

static int hello3_data __initdata = 3;

static int __init hello_3_init(void)
{
        printk(KERN_INFO "Hello, world %d\n", hello3_data);
        return 0;
}

static void __exit hello_3_exit(void)
{
        printk(KERN_INFO "Goodbye, world 3\n");
}

module_init(hello_3_init);
module_exit(hello_3_exit);
```

The __init macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers, but not loadable modules

The __exit macro causes the omission of the function when the module is built into the kernel, and like __exit, has no effect for loadable modules

Licensing and Module Documentation

- In kernel 2.4 and later, a mechanism was devised to identify code licensed under the GPL (and friends) so people can be warned that the code is non open-source
- This license mechanism is defined and documented in linux/module.h
- MODULE_LICENSE("GPL");
- MODULE_AUTHOR(DRIVER_AUTHOR);
- MODULE_DESCRIPTION(DRIVER_DESC);

Modules Spanning Multiple Files

- Sometimes it makes sense to divide a kernel module between several source files.

```
#include <linux/kernel.h>
#include <linux/module.h>
int init_module(void) {
        printk(KERN_INFO "Hello, world - this is the kernel speaking\n");
        return 0;
}
```

```
#include <linux/kernel.h>
#include <linux/module.h>
void cleanup_module() {
printk(KERN_INFO "Short is the life of a kernel module\n");
 }
```

```
obj-m += startstop.o
startstop-objs := start.o stop.o
```

Char Drivers

Agenda

- Device Nodes
- Major and minor numbers
- Accessing the device node
- *Udev*

PRESENTS...

Device Nodes

- Character and block devices have file system entries associated with them.
- These nodes can be used by user-level programs to communicate with the device, whose driver can manage more than one device node
- ls –la /dev/
- Device nodes are made with
  - mknod [-m mode] /dev/name <type> <major> <minor>
  - mknod –m 666 /dev/mycdrv c 254 1
  - mknod() system call

Maior and minor numbers

The mayor and minor numbers identify associated with the device

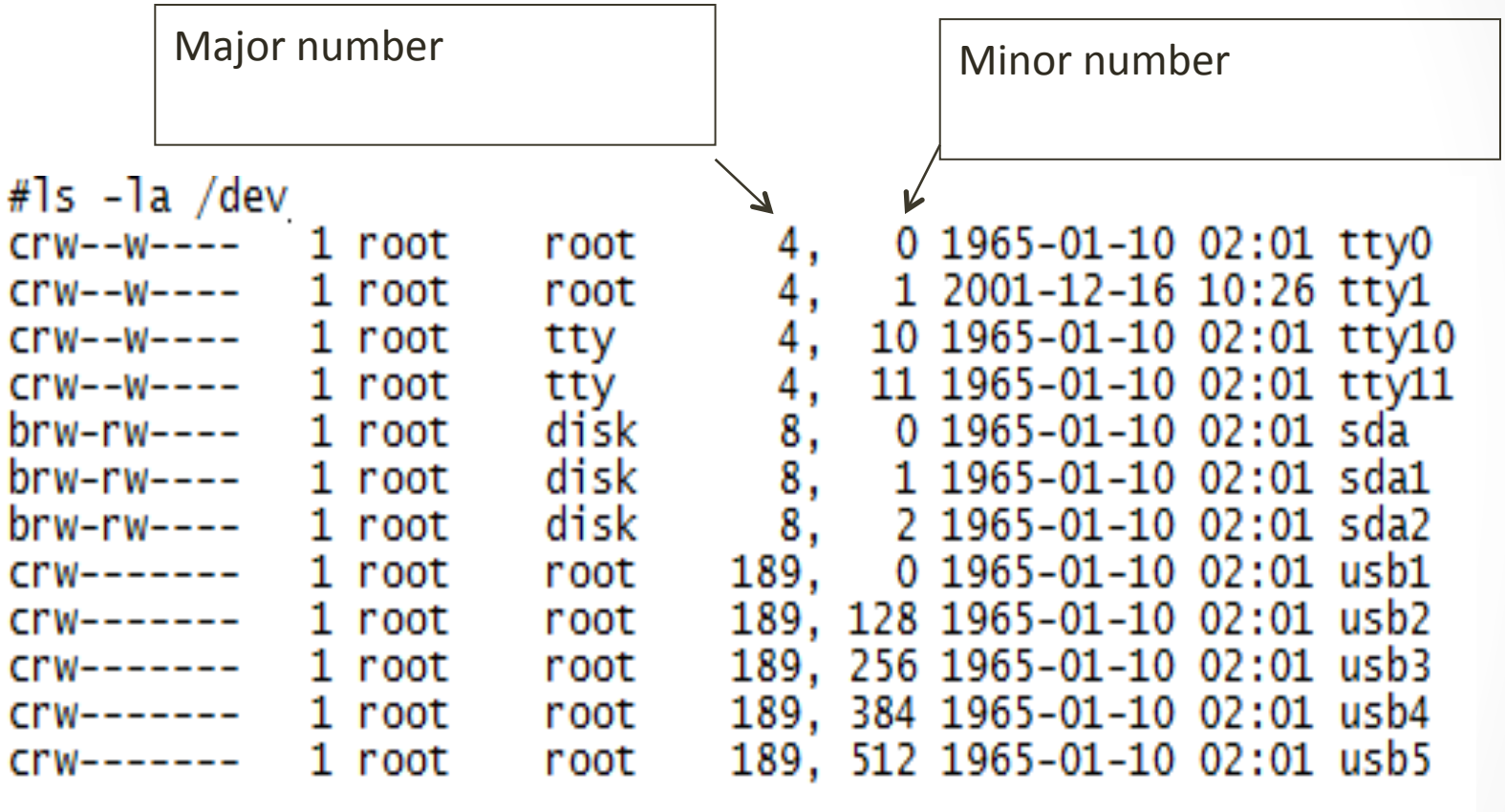
The minor number is used only within the device driver to either differentiate between instance of the same kind of device :

- First and second sound card\
- Hard disk partition

Operation of a given device

- Density floppy driver

Major and minor numbers



Reserving Mayor/Minor numbers

- List of major and minor numbers pre associated with devices can be found in /usr/src/linux/Documentation/devices.txt
- Adding a new driver to the system (i.e. registering it) means assigning a major number to it , usually during the device's initialization routine

```
#include <linux/fs.h>
        int register_chrdev_region ( dev_t first, unsigned int count, char
        *name); // name = /proc/devices
        void unregister_chrdev_region( dev_t first,unsigned int count) ;
```

Dynamic Allocation of Mayor Number

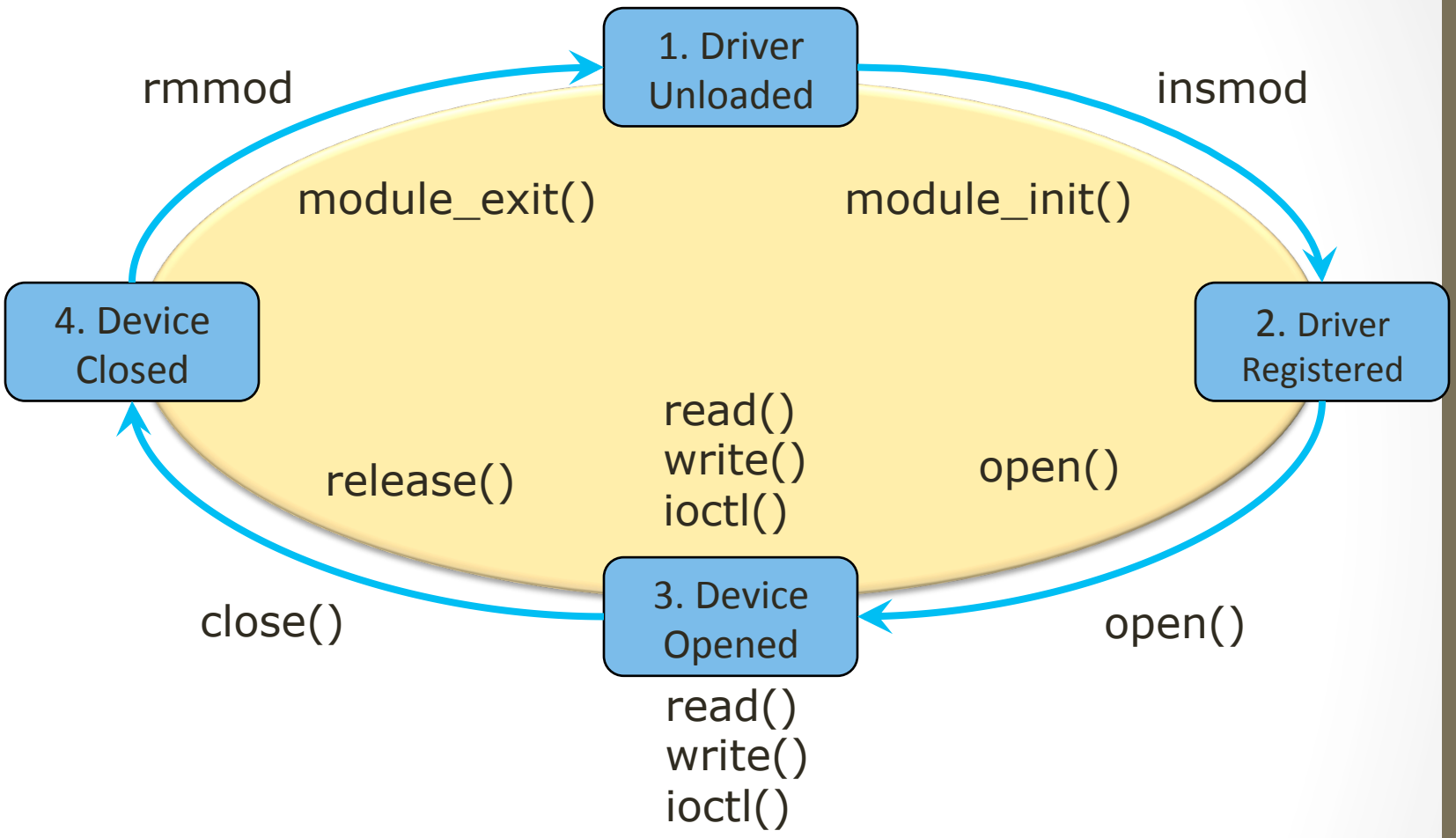
- Choosing a unique major number may be difficult , lets make it dynamically

```
#include <linux/fs.h>
int alloc_chrdev_region(dev_t *first,
                        unsigend int firstminor, // usually 0
                        unsigned int count,
                        char *name)
```

- Disadvantage : Require script !!!!!!

Summary

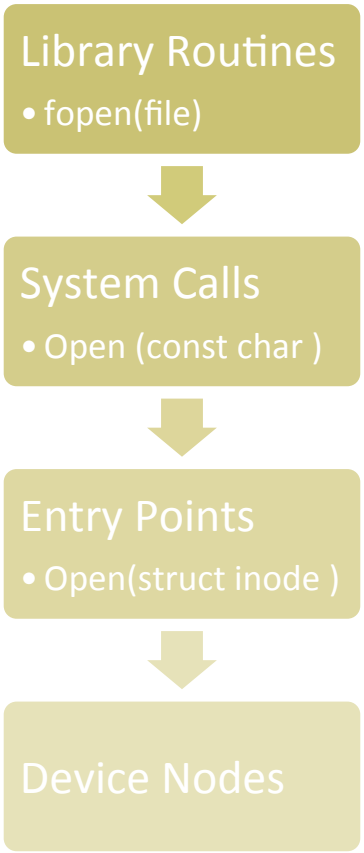
`dev_t`	type used to represent device numbers within the kernel.
`MKDEV()`	Builds a `dev_t` object given major and minor numbers.
`MAJOR()` `MINOR()`	Extract the major and minor numbers from a dev_t object.
`register_chrdev_region()`	Registers a range of device numbers (when the major number is known in advance).
`alloc_chrdev_region()`	Dynamically allocates a range of device numbers.
`unregister_chrdev_region()`	Releases a range of device numbers, regardless if `alloc_chrdev_region` or `unregister_chrdev_region` was used.



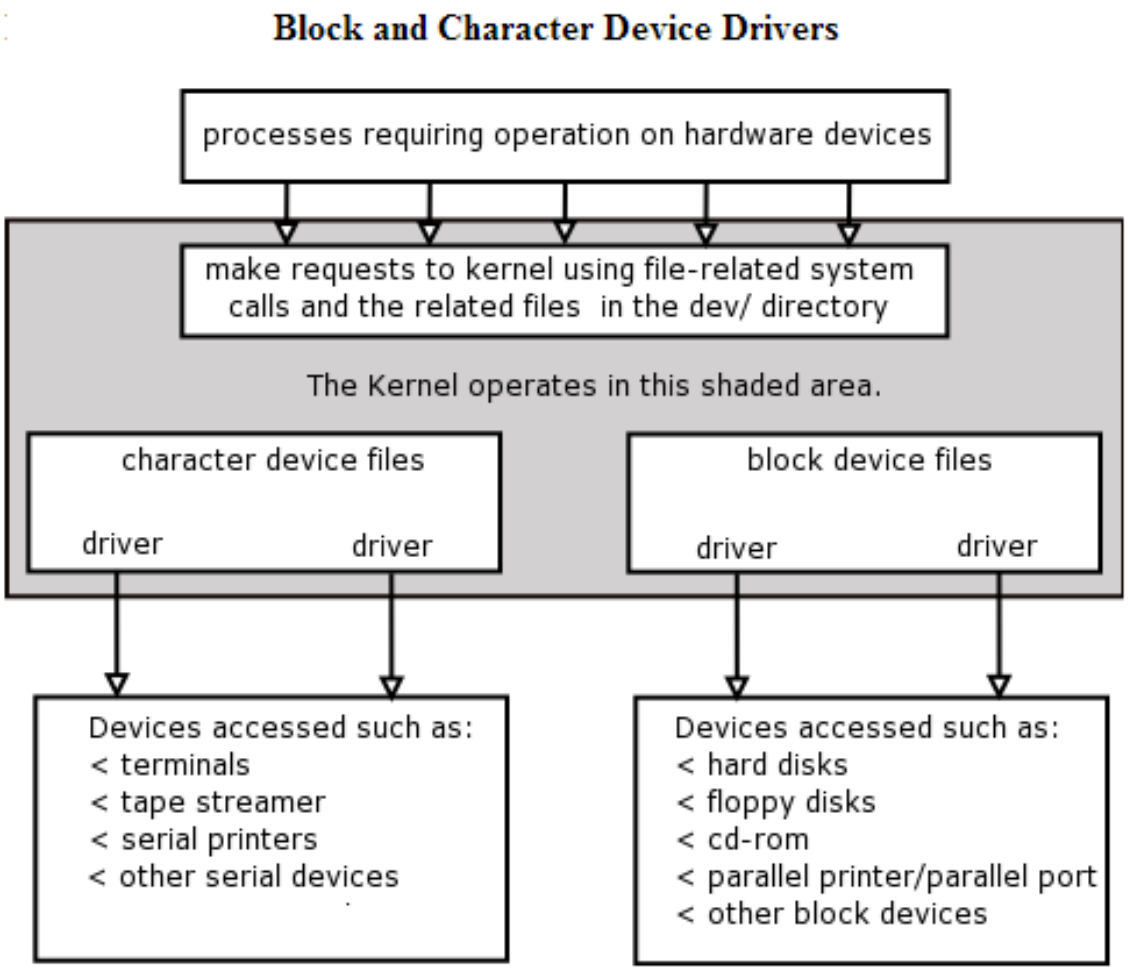
Notes:
Inside Circle: **Kernel functions**
Outside Circle: **User Functions**

Accessing the device node

- Unix like operational systems , such as Linux , applications access peripheral devices using the same function as they would for ordinary files



Calls from user space



Initialize and register cdev

- So far all we have done is reserve a range of device numbers!!!
- The kernel uses structures of type cdev to represent char devices internally. It is defined in <linux/cdev.h>.
- Before invoking device's operations, the code should allocate and register these structures.
- To initialize the struct cdev, to register, and to remove a char device from the system:

Relevant functions

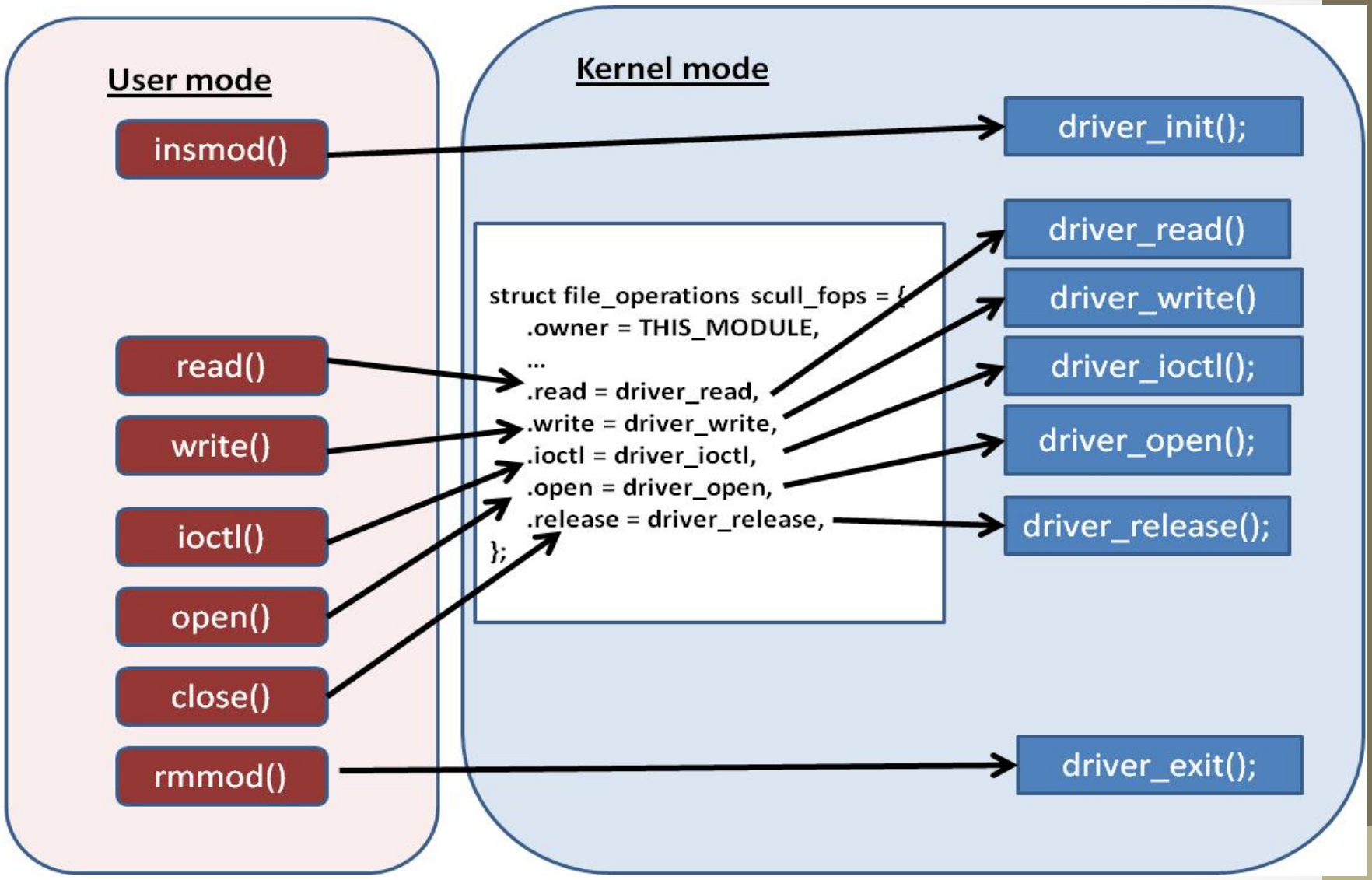
struct cdev *mycdev =cdev_alloc();	Allocate device in kernel
cdev_init(mycdev,&fops);	Initialization of module
cdev_add(mycdev,first,count);	Turn it on !!!!!
cdev_del(mycdev)	Turn it off !!!!!

Fops ????

Char Driver Entry Points (FOPS)

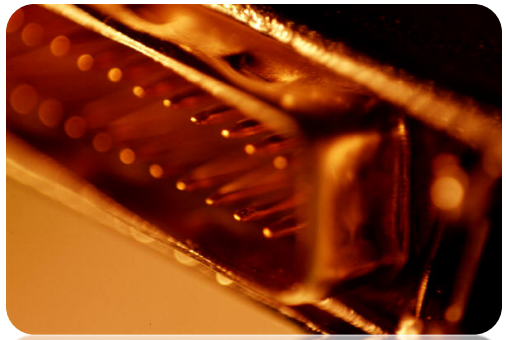
- Entry points are described by the file_operations structure in <linux/fs.h>
- The first field of the file_operations is owner, which is a pointer to the module that owns the structure.
  - It is used to prevent the module from being unloaded while its operations are in use.
  - It is usually set to THIS_MODULE.

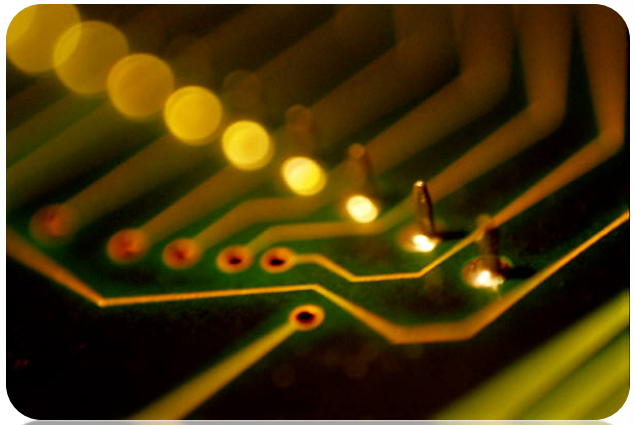
```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        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 *, struct dentry *, 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 **);
};
```


User mode
Kernel mode
insmod()
driver_init();
read()
write()
ioctl()
open()
close()
rmmod()
struct file_operations scull_fops = {
.owner = THIS_MODULE,
...
.read = driver_read,
.write = driver_write,
.ioctl = driver_ioctl,
.open = driver_open,
.release = driver_release,
};
driver_read()
driver_write()
driver_ioctl();
driver_open();
driver_release();
driver_exit();

Reality !!!

- /dev reached 15,000 ~20, 000 in 2.4 kernel !!!!
- Nodes by default = maybe never used = need to remove
- Potentially error prone task





Udev came to save the day !!!

The udev method creates device nodes on the fly as they

Udev does not handle the discovery of devices or management of them (HAL)

#include <linux/device.h>

- Struct class *class_create
- Struct device *device_create
- Struct device_destroy
- Struct class_destroy

Steps to write a char driver

1. Create the init() function.

2. Create the exit() function to clean up everything that was done in the init() function.

3. Declare some functions to implement the file operations for your driver (you can declare empty read() and write() functions).

4. Statically initialize the file_operations structure, pointing to your previously created functions.

For the init() function do:

As optional step, set up a parameter to enable users to manually enter the major number.

Set the major and minor numbers dynamically or statically, with alloc_chardev_region() or register_chardev_region() , respectively.

Declare and initialize a cdev structure using your struct file_operations, with the function cdev_init(). Set the owner field to THIS_MODULE.

Register the struct cdev. The cdev_add() function register the services of the driver with the kernel.

Add code to register with the sysfs, using the functions class_create() and device_create(). This will create a file in the /dev directory.

For the `exit()` function do:

Unregister services from kernel: cdev_del().

Release major and minor numbers: unregister_chrdev_region().

Unallocate memory allocated by the init() function.

Unregister device from sysfs: device_destroy() and class_destroy(). The file in /dev directory will be removed.

Steps to write a char driver

- Until this point the interfaces can be created and destroyed when the module is loaded and unloaded.
- The last part is to implement the functions declared for the file_operations structure.

```
static ssize_t myread(struct file *filp, char __user *buf,
                                                size_t nbuf, loff_t
*offs)
{
}
static ssize_t mywrite(struct file *filp, const char __user *buf,
                                                size_t nbuf, loff_t
*offs)
{
}
<...>
static const struct file_operations mydev_fops = {
         .owner = THIS_MODULE,
         .read = myread,
         .write = mywrite
         <...>
};
```

Implemented File Operations

Entry Point

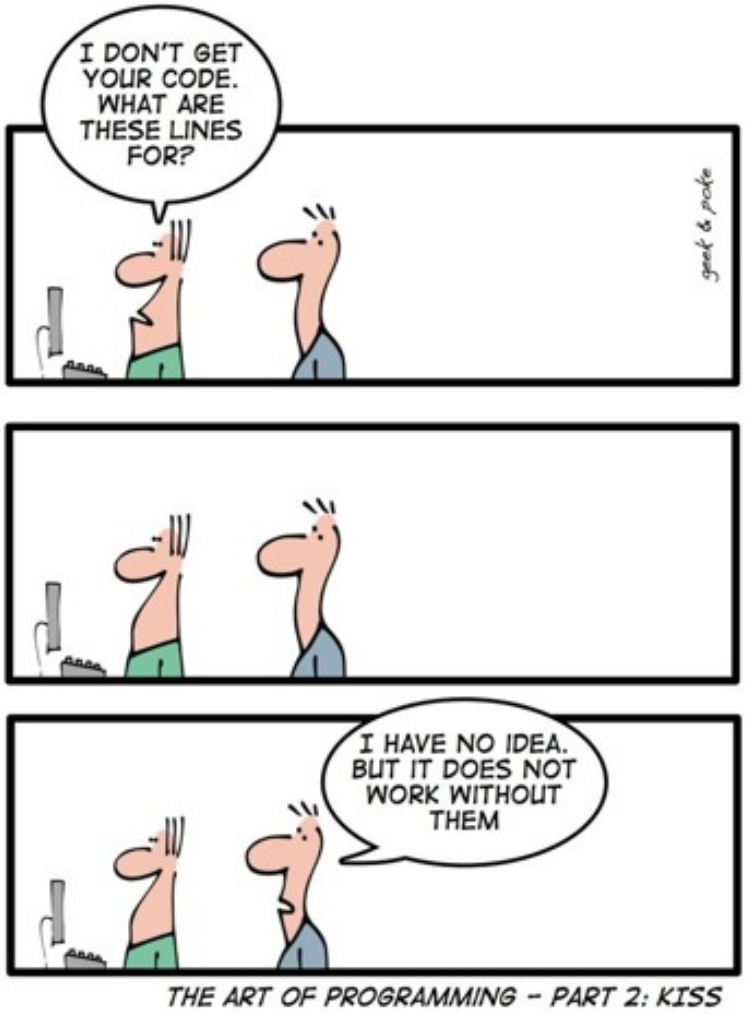
```
static int __init mymodule_init (void)
{
        /* Allocate major/minor numbers */
        /* Initialize cdev structure and register it with the kernel*/
        /* Do any initialization required, such as allocating memory */
}

static void __exit mymodule_exit(void)
{
        /* Deregister services from kernel */
        /* Release major/minor numbers */
        /* Undone everything initialized in init function */
}

module_init(mymodule_init);
module_exit(mymodule_exit);

MODULE_AUTHOR("");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(“Char driver skeleton");
```

Init function

Exit function

Module info

Lab Part 1 , lets create a terminal



References

- The Linux Kernel Module Programming Guide
- Linux Device Drivers 3rd Edition
- Linux Tutorial
- Linux Kernel Development Second Edition
- LF331 Developing Linux Device Drivers
-