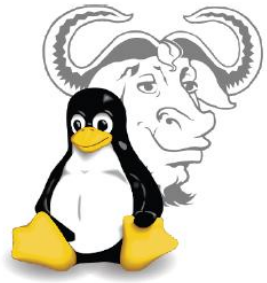
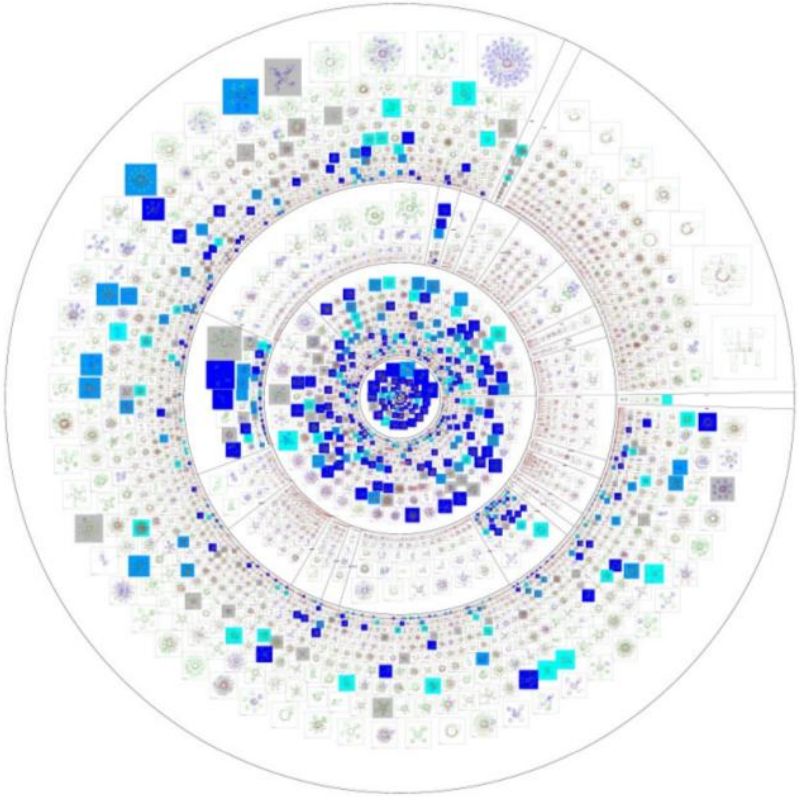
Introducción a los Device Drivers

(CC) Por Daniel A. Jacoby

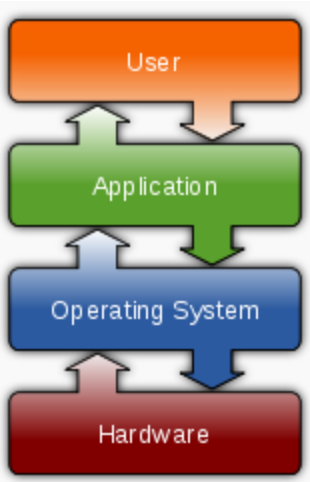




ITBA
INSTITUTO TECNOLOGICO DE BUENOS AIRES
•UNIVERSIDAD PRIVADA•

Que son los DD ?

- En informática un DD es un programa que permite que las aplicaciones de alto nivel interactúen con el hardware.
- Son dependientes tanto del Sistema operativo como del hardware usado.



Porque son necesarios los DD?

Un Sistema Operativo debe poder evolucionar

- Incorporación de nuevos dispositivos
- Flexibilidad para la corrección de errores
- Permitir la optimización del funcionamiento

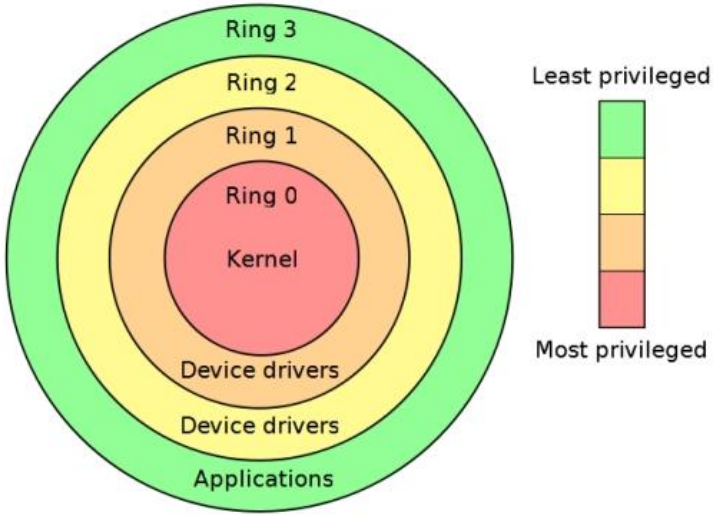
Como ???

LKM

- Programación de código abierto (Open Source)
- LKM (Loadable Kernel Modules)

Acceso al Hardware

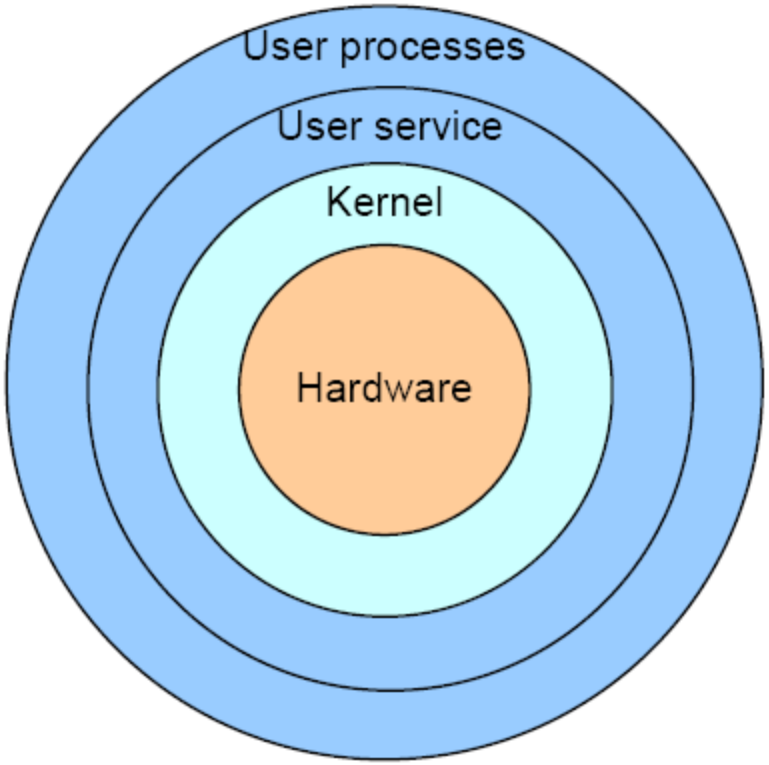
- Protección del OS y Aplicaciones !!!
- Windows & Linux → supervisor/user-mode



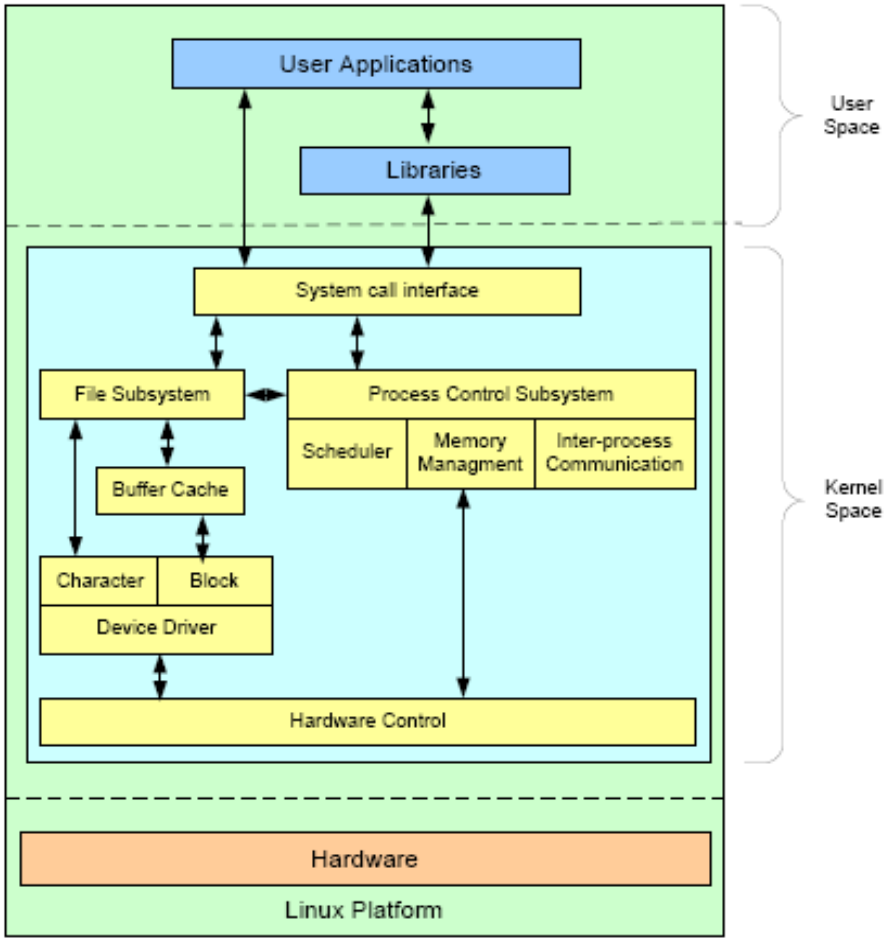
Kernel y User space

- **Kernel Space** : Acceso directo al hardware de manera organizada. Impedir que el usuario acceda a recursos del hardware de cualquier forma
- **User space**: Aplicaciones del usuario que deberán estar controladas para evitar hacer daño al Sistema operativo u otras aplicaciones (Ring3)

Interfaz US-KS



Interfaz US-KS



Interfaz básica de un LKM

Event	User function	Kernel function
Load Module	insmod	module_init()
Open device	fopen	file_operation:open
Read Device	fread	file_operation:read
Write Device	fwrite	file_operation;write
Close Device	fclose	file_operation:release
Remove Device	rmmod	module _exit()

Interfaz mínima

- int **init_module**(void);

  Es invocada durante la instalación del modulo

- void **cleanup_module**(void);

  Es invocada durante la remoción del modulo

Interfaz mínima (usando macros)

- **module_init** (x);

  Es invocada durante la instalación del modulo

- **module_exit**(x);

  Es invocada durante la remoción del modulo

PRINTK

La funcion printk() es usada en la programacion de device drivers para enviar mensajes al log del kernel y sirve para depurar el driver

Se puede ver usando el comando dmesg desde la línea de comandos

Sintaxis :printk ("log level" "message", <arguments>);

Log level:

```
#define KERN_EMERG "<0>" /* system is unusable*/
#define KERN_ALERT "<1>" /* action must be taken immediately*/
#define KERN_CRIT "<2>" /* critical conditions*/
#define KERN_ERR "<3>" /* error conditions*/
#define KERN_WARNING "<4>" /* warning conditions*/
#define KERN_NOTICE "<5>" /* normal but significant condition*/
#define KERN_INFO "<6>" /* informational*/
#define KERN_DEBUG "<7>" /* debug-level messages*/
```

Ejemplo Printk()

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

int init_module(void)
{ printk("<1>Hello, world\n"); return 0; }

void cleanup_module(void)
{ printk("<1>Goodbye cruel world\n"); }
```

Ejemplo

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

/*
 * hello_init  the init function, called when the module is loaded.
 * insmod drv1.ko
 * Returns zero if successfully loaded, nonzero otherwise.
 */
static int hello_init(void)
{
        printk(KERN_ALERT "Hello minimal driver.\n");
        return 0;
}

/*
 * hello_exit  the exit function, called when the module is removed.
 * rmmod drv1
 */
static void hello_exit(void)
{
        printk(KERN_ALERT "Bye,bye, Device Driver!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Daniel A. Jacoby");
```

Compilacion

- Make File : mmake (crea el makefile y compila)

$>./mmake mydriver (solo la primera vez)

mydriver.c → mydriver.ko

$> make (once mmake was run this also does the job)

mydriver.c → mydriver.ko

Nota: Esto crea el makefile asi que para subsecuentes compilaciones solo hace falta ejecutar make.

Compilacion: Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m := drv1.o

else

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
	rm -r .tmp_versions *.mod.c .*.cmd *.o

endif
```

Instalacion/Remocion

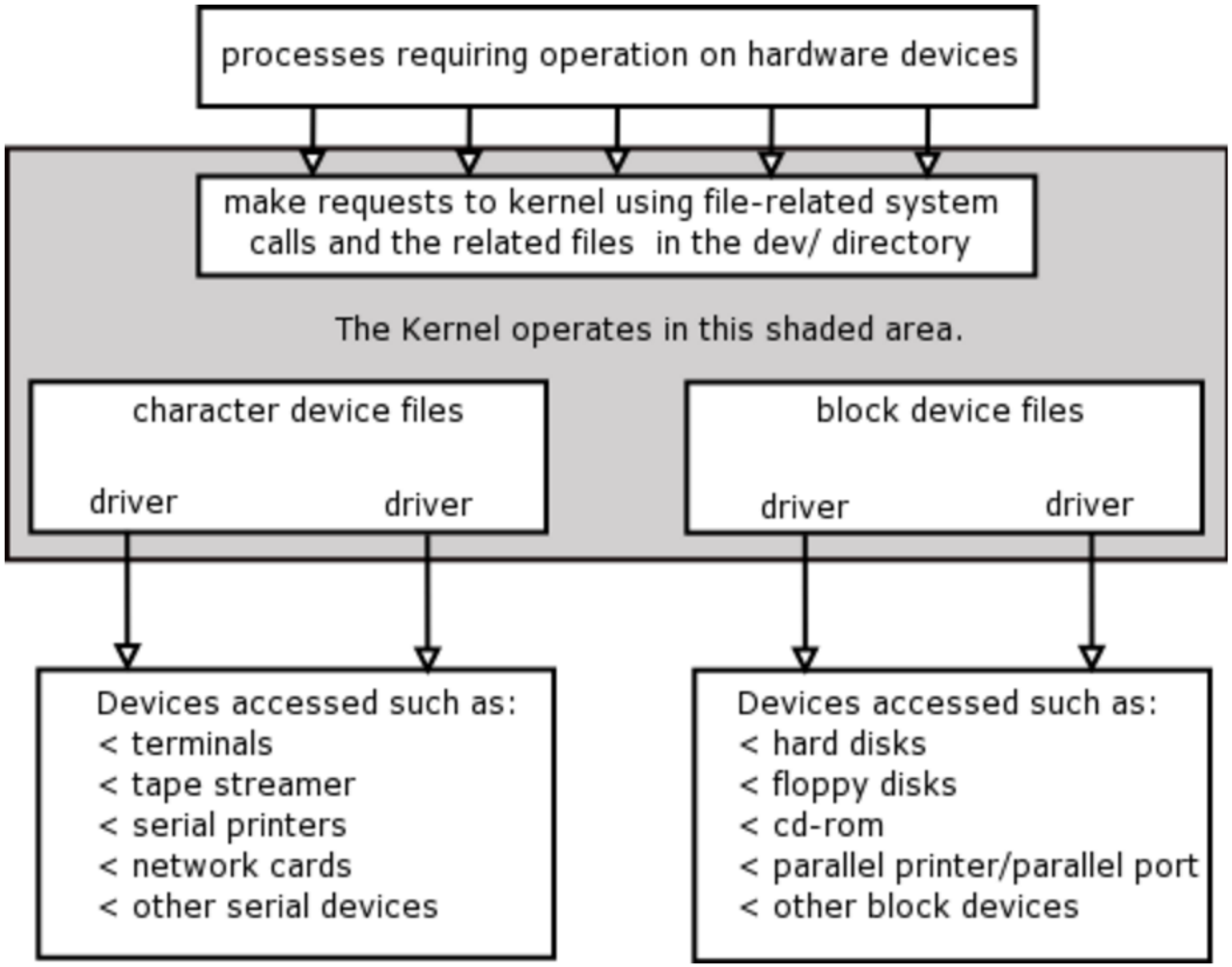
- sudo insmod mydriver.ko
- sudo rmmod mydriver

Try It !!

- Uncompress drv1.zip
- Compile ./mmake drv1.c (..or make)[1]
- Install sudo insmod drv1.ko
- See installed modules lsmod | more (or much better lsmod | grep drv1)
- See module info modinfo ./drv1.ko
- See kernel log dmesg | tail -3
- sudo rmmod drv1
- See kernel log dmesg | tail -3

Note1 simply run make if mmake was already executed

Character vs Block Device Drivers



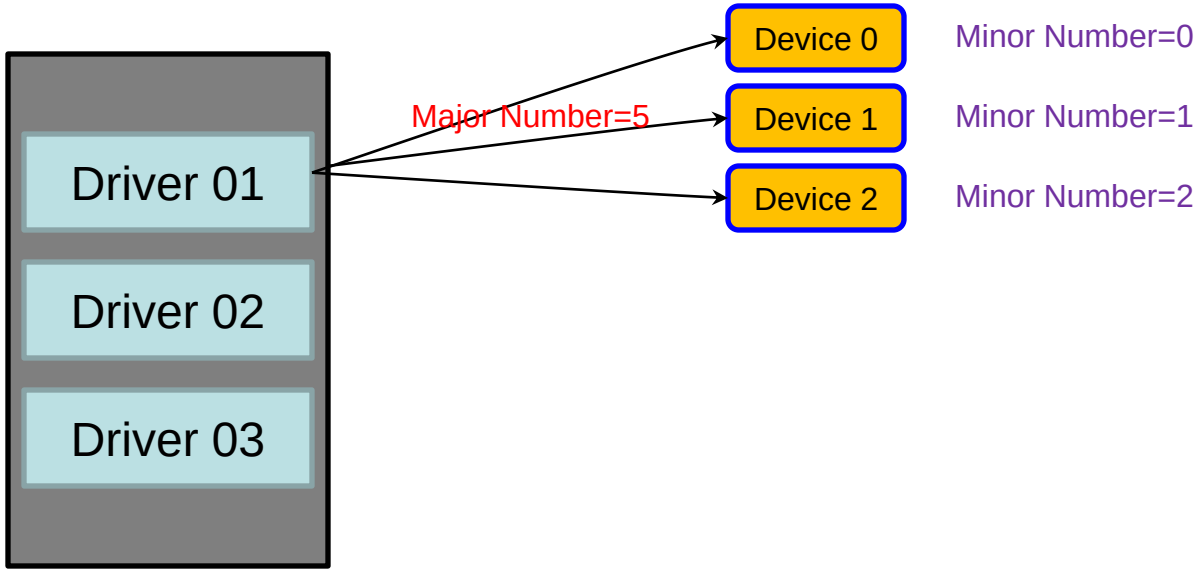
Character Devices

- Son dispositivos de hardware que acceden de un byte a la vez en forma secuencial (seriales) a los datos.
- No usan Buffers (Unbuffered).
- Se encuentran en /dev .
- ls -l /dev   c = char b=block l= symlink d=directory

```
crw-rw-rw-   1 root root        1,   8 jun 11 11:06 random
crw-rw-r--+  1 root root       10,  62 jun 11 11:06 rfkill
lrwxrwxrwx   1 root root            4 jun 11 11:06 rtc -> rtc0
crw-------   1 root root      254,   0 jun 11 11:06 rtc0
brw-rw----   1 root disk        8,   0 jun 11 11:06 sda
brw-rw----   1 root disk        8,   1 jun 11 11:06 sda1
brw-rw----   1 root disk        8,   2 jun 11 11:06 sda2
brw-rw----   1 root disk        8,   5 jun 11 11:06 sda5
crw-rw----   1 root disk       21,   0 jun 11 11:06 sg0
crw-rw----+  1 root cdrom      21,   1 jun 11 11:06 sg1
lrwxrwxrwx   1 root root            8 jun 11 11:06 shm -> /run/shm
crw-------   1 root root       10, 231 jun 11 11:06 snapshot
drwxr-xr-x   4 root root          320 jun 11 11:06 snd
brw-rw----+  1 root cdrom      11,   0 jun 11 11:06 sr0
lrwxrwxrwx   1 root root           15 jun 11 11:06 stderr -> /proc/self/fd/2
lrwxrwxrwx   1 root root           15 jun 11 11:06 stdin -> /proc/self/fd/0
lrwxrwxrwx   1 root root           15 jun 11 11:06 stdout -> /proc/self/fd/1
crw-rw-rw-   1 root tty         5,   0 jun 12 00:17 tty
crw--w----   1 root tty         4,   0 jun 11 11:06 tty0
crw-rw----   1 root tty         4,   1 jun 11 11:06 tty1
crw--w----   1 root tty         4,  10 jun 11 11:06 tty10
crw--w----   1 root tty         4,  11 jun 11 11:06 tty11
```

- cat /proc/devices   // Muestra todos los dispositivos existentes

Major and Minor Numbers

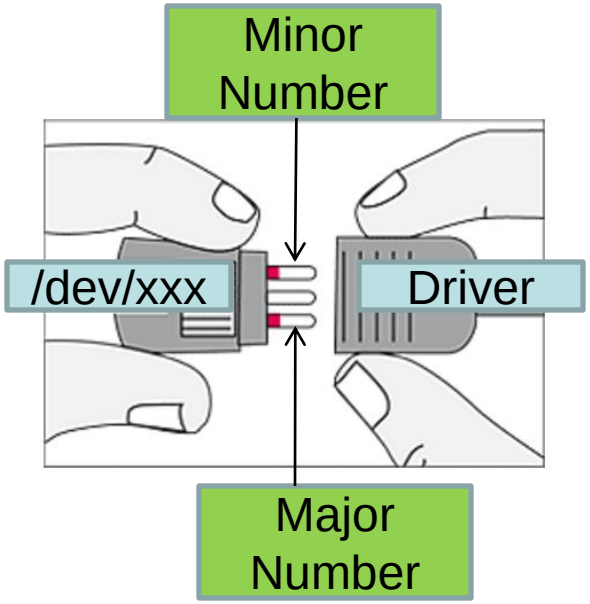


Cada driver tiene un “Major Number” que lo identifica en forma unica dentro del kernel. A su vez cada driver controla por lo menos a un dispositivo. Para numerar cada dispositivo se usa un numero llamado “Minor Number”.

Major and Minor Numbers

Ejemplo ls -l /dev

```
crw--w----  1 root tty      4,  0 jun 11 11:06 tty0
crw-rw----  1 root tty      4,  1 jun 11 11:06 tty1
crw--w----  1 root tty      4, 10 jun 11 11:06 tty10
crw--w----  1 root tty      4, 11 jun 11 11:06 tty11
crw--w----  1 root tty      4, 12 jun 11 11:06 tty12
crw--w----  1 root tty      4, 13 jun 11 11:06 tty13
crw--w----  1 root tty      4, 14 jun 11 11:06 tty1
crw--w----  1 root tty      4, 15 jun 11 11:06 tty1
crw--w----  1 root tty      4, 16 jun 11 11:06 tty1
crw--w----  1 root tty      4, 17 jun 11 11:06 tty1
crw--w----  1 root tty      4, 18 jun 11 11:06 tty1
```



Es responsabilidad del programador crear una entrada (inode) en /dev Esta entrada sera algo asi como el gateway entre el dispositivo (/dev/michardriver) y el driver (modulo). Este vinculo se hace efectivo al corresponder el par “Major Number” y el “Minor Number” del dispositivo (/dev/xxx) y el modulo o device driver.

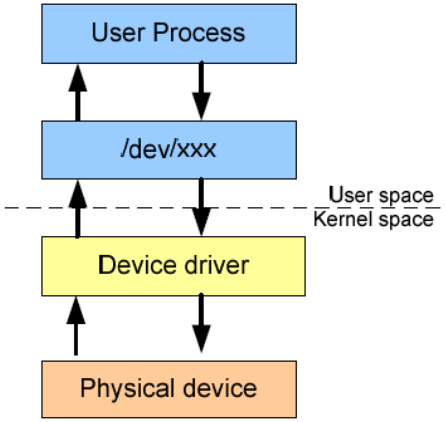
Device Nodes

Como crear un nodo

Un Nodo se crea de la siguiente forma:

```
sudo mknod /dev/mydev  c  61 0     // c =char device
chmod ugo+rw /dev/mydev            // Set permission
```

El dispositivo “mydev” tiene asignado un major=61 y el minor=0



(sudo rm /dev/mydev // remover)

Registrar el driver

Para registrar un driver tenemos las siguientes funciones:

```
/*For static Major of Number */
int register_chrdev_region(dev_t first, unsigned int count, char *name)

/*For dynamic allocation of Major Number */
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name)
```

La funcion **register_chrdev_region()** es usada cuando conocemos de antemano el "Major number" . En caso de usar asignacion dinamica del "Major number" usamos la funcion **alloc_chrdev_region().**

Nota: Existe tambien una funcion **register_chrdev()** que es mas antigua y debe ser evitada en nuevas implementaciones.El motivo es que no puede ver un numero expandido de dispositivos, el numero maximo se limita a 255 (si se solicita un valor superior retorna error) Ver Apendice.

Registrar el driver

Macros útiles:

Los siguentes macros permiten obtener los valores de major y minor a partir de un numero de Dispositivo

```
int MAJOR(dev_t dev);
int MINOR(dev_t dev);
```

La siguiente macro nos permite obtener el numero de dispositivo dentro del kernel a partir de Major y Minor number

```
dev_t MKDEV(unsigned int major, unsigned int minor);
```

Nota: dev_t es un tipo definido para representar el numero de un dispositivo dentro del kernel

register_chrdev_region()

Parámetros

/*For static Major of Number */

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

first : Es el numero del primer dispositivo del driver

count : Es el numero dispositivos del driver

name: Es el nombre con que aparece el driver en /proc/devices

Retorna 0 si no hubo problemas y <0 si hubo error

Ejemplo:

```
static int mydevice_major = 61;
int result;
dev_t dev = 0;
dev = MKDEV(mydevice_major, 0);   // Major , Minor
result = register_chrdev_region(dev, 1, "mydevname");
```

alloc_chrdev_region()

Parametros

/*For dynamic allocation of Major Number */

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

dev: Este es un parámetro de salida que quedara inicializado con el numero del primer dispositivo del driver

firstminor : Es el valor del primer dispositivo del driver

count : Es el numero dispositivos del driver

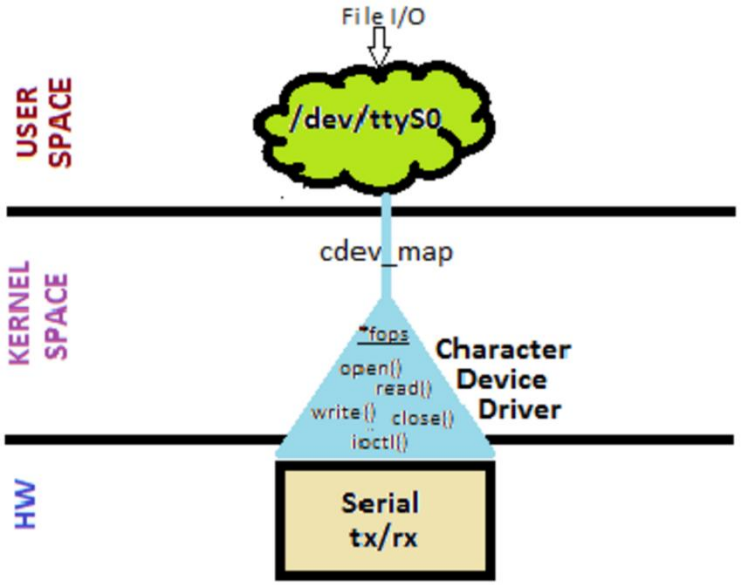
name: Es el nombre con que aparece el driver en /proc/devices

Retorna 0 si no hubo problemas y <0 si hubo error

Ejemplo:

```
static dev_t mydev;
alloc_chrdev_region(&mydev,0,1,"mydevname");
```

The file_operations structure



struct file_operations

```
struct file_operations my_fops =
    {
    owner:      THIS_MODULE,
    read:       my_read,
    write:      my_write,
    open:       my_open,
    release:    my_release,
    };
```

Todos los dispositivos con transferencia serial (streaming ej.: Teclado.Terminales) son Character devices y son accesibes en User Space como un nodo en el sistema de archivos (device node file) ej.; /dev/ttyS0. Las aplicaciones acceden a estos dispositivos mediante operaciones de archivos de uso regular en el User Space (fopen(),fputc(), ….), La funcion del driver es transladar las operaciones de archivos en el User Space y operaciones del dispositivo (fops: open() read() write() ioctl() close()) implementadas mediante la estructura **file_operations**

The cdev structure

La Estructura cdev encapsula a fops y otros elementos necesarios para describir al driver por ejemplo minor y major numbers
La estructura cdev es accedida en el kernel por la siguiente API:

cdev_init() - used to initialize struct cdev with the defined file_operations
cdev_add() - used to add a character device to the system.
cdev_del() - used to remove a character device from the system

Despues de inicializar la estructura cdev (cdev_init()) y agregarla al kernel (cdev_add()) el driver cobra vida!!!

The cdev API

Parametros

void cdev_init(struct cdev *cdev, struct file_operations *fops);
cdev: es al estructura a inicializar
fops: debe apuntar a la estructura file_operations de este modulo

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

dev: es la estructura cdev del dispositivo
num: es el numero del primer dispositivo del modulo
count: numero total de dispositivos del modulo

Init driver Example 1/2

```
static int hello_init(void)
{
int result;
dev_t dev = 0;

    printk(KERN_ALERT "Hello minimal driver.\n");



        /*
         * Dynamic major if not set otherwise.
         */
      if (hello_major) {
              dev = MKDEV(hello_major, 0);
              result = register_chrdev_region(dev, 1, "hello");
      } else {
              result = alloc_chrdev_region(&dev, 0, 1, "hello");  // Dynamic major
              hello_major = MAJOR(dev);
      }
      if (result < 0) {
              printk(KERN_WARNING "hello: can't get major %d\n", hello_major);
              return result;
      }

        hello_setup_cdev(helloDevs, 0, &hello_fops);



    printk(KERN_INFO   hello_DRIVER_NAME " " hello_DRIVER_VERSION " registered\n");
        dprintk("dev major = %d\n", hello_major);
        dprintk("Dany %d\n", 2);



        return 0;

}
```

Init driver Example 2/2

```
/*
 * Set up the cdev structure for a device.
 */
static void hello_setup_cdev(struct cdev *dev, int minor,
                             struct file_operations *fops)
{
        int err, devno = MKDEV(hello_major, minor);

        cdev_init(dev, fops);
        dev->owner = THIS_MODULE;
        dev->ops = fops;
        err = cdev_add(dev, devno, 1);
        /* Fail gracefully if need be */
        if (err)
                printk(KERN_NOTICE "Error %d adding hello %d", err, minor);
}


/*
 * hello_exit  the exit function, called when the module is removed.
 * rmmod drv2
 */
static void hello_exit(void)
{

        cdev_del(helloDevs);
        unregister_chrdev_region(MKDEV(hello_major, 0), 1);
        printk(KERN_ALERT "Bye,bye, Hello Device Driver!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Daniel A. Jacoby");
```

File operations

```
static struct file_operations hello_fops = {
        .owner = THIS_MODULE,
        .open = hello_open,
        .release = hello_release,
        .write  = hello_write,
        .read = hello_read,

#if LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 35)
        .ioctl  = hello_ioctl,
#else
        .unlocked_ioctl = hello_ioctl,
#endif
};
```

File operations open() and release()

Habitualmente se usa para inicializar (liberar) recursos (buffers,punteros ,etc)

```
static int hello_open(struct inode *ino, struct file *filep);
static int hello_release(struct inode *node, struct file *file);
```

Se invoca open() cuando una aplicación abre el nodo

Ej.: $ cat /dev/midriver

Y al finalizar se llama a release()

File operations write() and read()

Mediante estas funciones se intercambian datos entre la applicacion (User space) y el dispositivo.

```
static ssize_t hello_read(struct file *filp,  char *msgusr, size_t length, loff_t *offset );
static ssize_t hello_write(struct file *file, const char *buf,size_t n, loff_t *ppos);
```

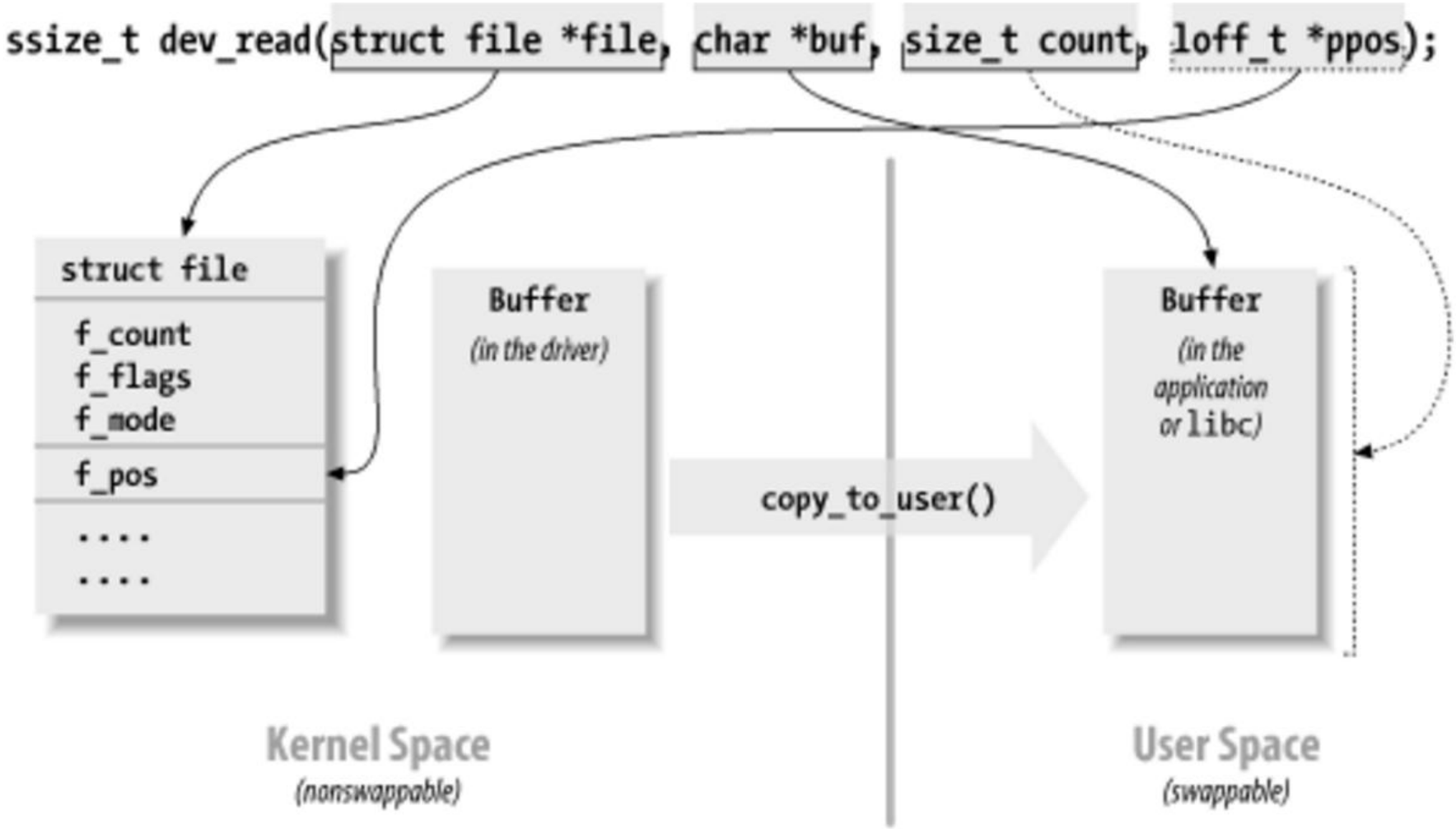
Se invoca read() cuando una aplicación lee el nodo

Ej.: fgetc(fd) cat /dev/mydevice

o write() cuando se escribe datos al nodo

Ej:. fputc('c',fd); echo "1" > /dev/mydevice

read()


ssize_t dev_read(struct file *file, char *buf, size_t count, loff_t *ppos);
struct file
f_count
f_flags
f_mode
f_pos
....
....
Buffer
(in the driver)
copy_to_user()
Buffer
(in the application or libc)
Kernel Space
(nonswappable)
User Space
(swappable)

read()

static ssize_t read(struct file *filp, char *msgusr, size_t length, loff_t *offset)

filp: file pointer
msgusr: puntero al buffer con datos para el usuario
length: cantidad de datos a transferir
offset: puntero al desplazamiento respecto del filp

Para acceder a los datos NO se debe usar el puntero msgusr pues no es seguro

-El puntero del usuario no es confiable
-En el momento de la llamada (system call) la memoria del usuario esta en una pagina differente

Para no comprometer al kernel se usa una funcion segura de copia del espacio de usario al espacio de kernel

unsigned long copy_to_user(void __user* to ,const void * from, unsigned long count);

write()

static ssize_t write(struct file *filp, const char *msgusr, size_t length, loff_t *offset)

filp: file pointer
msgusr: puntero al buffer con datos del usuario
length: cantidad de datos a transferir
offset: puntero al desplazamiento respecto del filp

Para acceder a los datos NO se debe usar el puntero msgusr pues no es seguro

-El puntero del usuario no es confiable
-En el momento de la llamada (system call) la memoria del usuario esta en una pagina differente

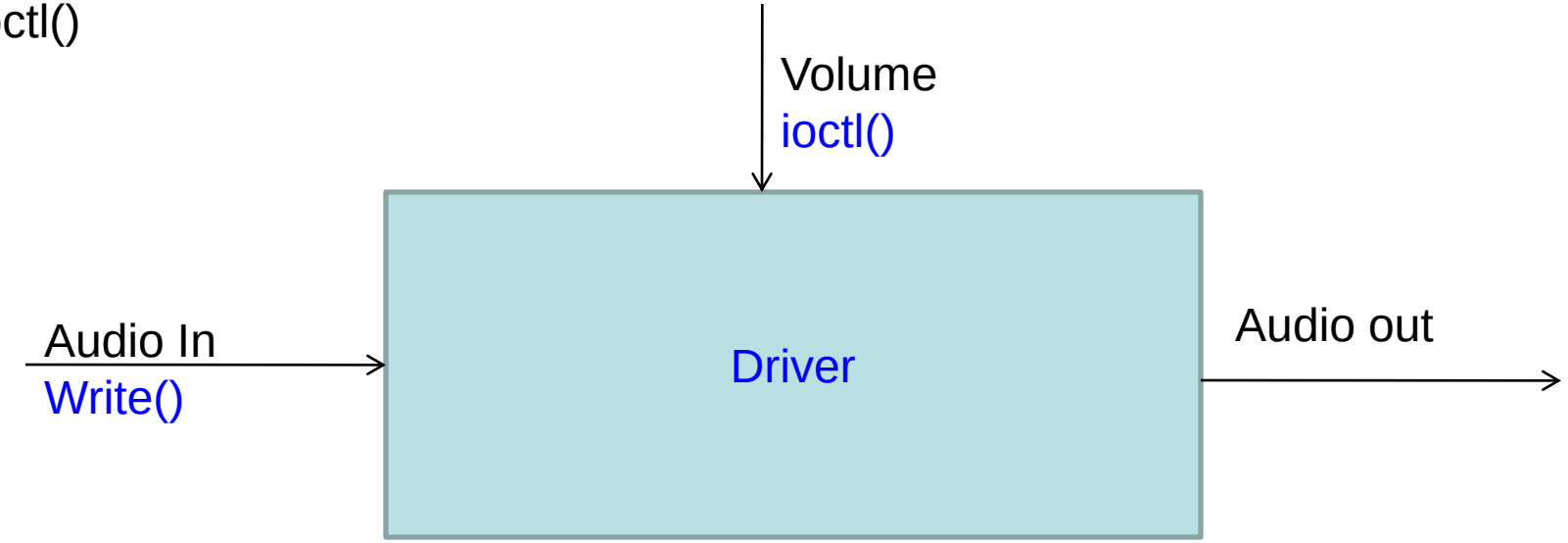
Para no comprometer al kernel se usa una funcion segura de copia del espacio de usario al espacio de kernel

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

IOCTL() (I/O control)

ioctl() es una funcion que nos pemite parametrizar el funcionamiento del driver.

Supongamos que tenemos un repoductor de mp3 funcionando en la PC
El flujo de datos sale del disco hacia el driver del reproductor (usando la funcion write() del driver del reproductor). Si quisieramos modificar el volumen de salida u otro parametro deberemos usar una entrada auxiliar al driver para hacerlo usamos ioctl()



IOCTL() (Ejemplo)

```
char buf[200];

long device_ioctl(struct file *filep, unsigned int cmd, unsigned long arg) {


        int len = 200;

        printk("CMD:%ul Rd:%ul Wr:%ul",cmd,READ_IOCTL,WRITE_IOCTL);
        switch(cmd) {
        case READ_IOCTL:
                copy_to_user((char *)arg, buf, 200);
                break;

        case WRITE_IOCTL:
                copy_from_user(buf, (char *)arg, len);
                break;

        default:
                return -ENOTTY;
        }
        return len;

}
```

IOCTL() (Invocacion)

```c
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>

#define MY_MACIG 'G'
#define READ_IOCTL _IOR(MY_MACIG, 0, int)
#define WRITE_IOCTL _IOW(MY_MACIG, 1, int)


int main(){
        char buf[200];
        int fd = -1;
        if ((fd = open("/dev/cdev_example", O_RDWR)) < 0) {
                perror("open");
                return -1;
        }
        if(ioctl(fd, WRITE_IOCTL, "hello world") < 0)
                perror("first ioctl");
        if(ioctl(fd, READ_IOCTL, buf) < 0)
                perror("second ioctl");

        printf("message: %s\n", buf);
        return 0;

}
```

GPIO Support for Kernel

GPIO request solicita pines al Kernel

```
err = gpio_request_one(tx_pin, GPIOF_OUT_INIT_LOW, "PIN_NAME");  // request one pin
err = gpio_request_array(leds_gpios, ARRAY_SIZE(leds_gpios));    // request many pins at once
```

Constantes útiles

```
#define GPIOF_DIR_OUT  (0 << 0)
#define GPIOF_DIR_IN   (1 << 0)

#define GPIOF_INIT_LOW (0 << 1)
#define GPIOF_INIT_HIGH        (1 << 1)

#define GPIOF_IN               (GPIO_DIR_IN)
#define GPIOF_OUT_INIT_LOW     (GPIO_DIR_OUT | GPIO_INIT_LOW)
#define GPIOF_OUT_INIT_HIGH    (GPIO_DIR_OUT | GPIO_INIT_HIGH)
```

Exportar pines al User space:

gpio_export(tx_pin,0);

GPIO Support for Kernel

GPIO free libera pines que fueron solicitados por el Kernel

```
gpio_free(tx_pin);                                    // Free one pin
gpio_free_array(leds_gpios, ARRAY_SIZE(leds_gpios));  // Free many pins at once
```

Unexport pin

gpio_unexport(tx_pin);

comandos utiles

- lsmod
- cat /proc/devices
- cat /proc/ioports