Introducción a Linux Device Drivers

Capítulo 1: Linux Device Drivers, Third Edition
by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

Temario

1 - Introducción a Linux Device Drivers

2 - Conceptos generales sobre la creación de módulos

3 - Drivers de caracteres

4 - Timers y tiempos en GNU/Linux

Introducción

- Los controladores son “cajas negras” que permiten al usuario ocultar los detalles de cómo funciona el dispositivo.
- El usuario puede interactuar utilizando un conjunto de llamadas estandarizadas, que son independientes del controlador específico.
- El papel fundamental de los controladores de dispositivo es mapear estas llamadas a operaciones específicas propias del dispositivo, que actúan directamente sobre el hardware.
- Modularidad.

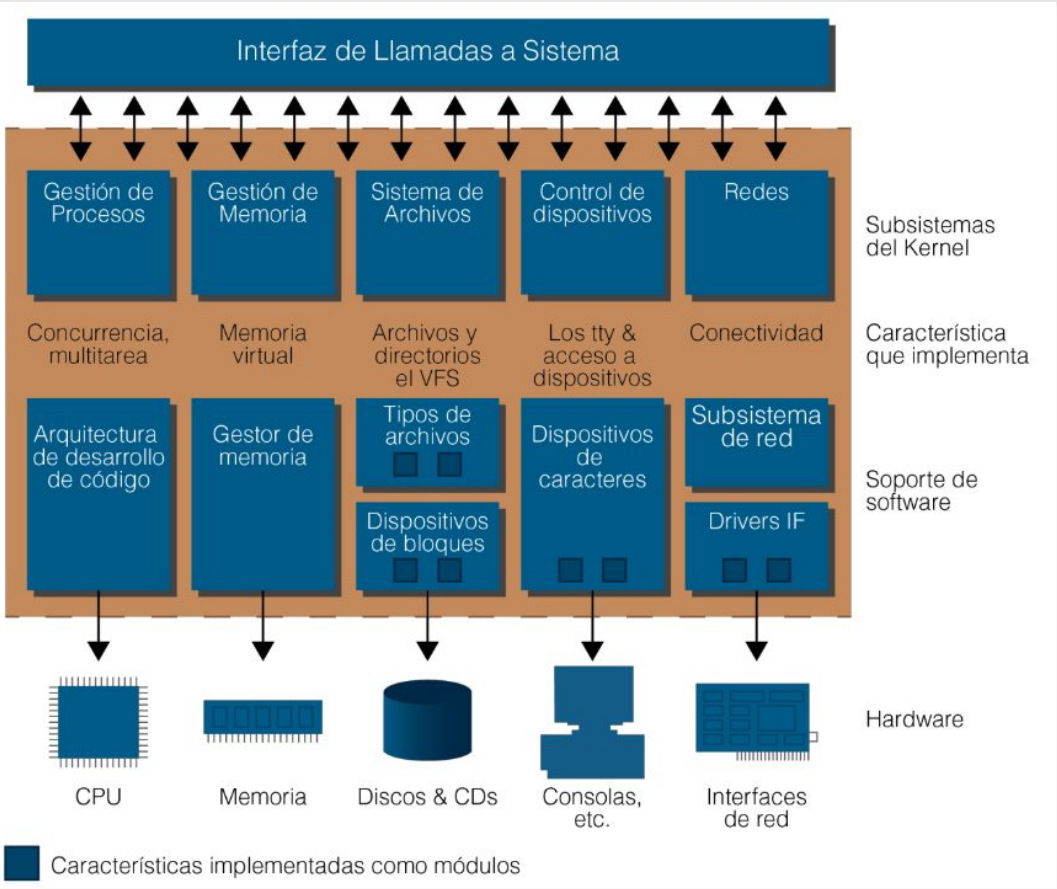
El rol de los drivers

El Rol del Drivers es el **MECANISMO**

NO LA POLITICAS

Splitting the Kernel

- Process management
- Memory management
- Filesystems
- **Device control**
- **Network**

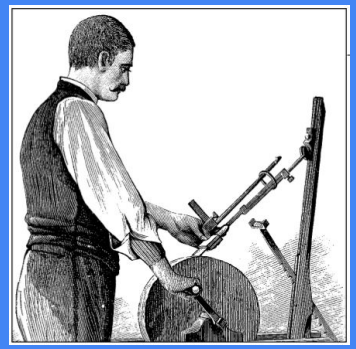


Modules

- Loadable Modules
- Precauciones:
  - Security Issues; bugs, Overflows,
  - Versión numbering
- License Terms
- Tipos de módulos:
  - Character devices
  - Block devices
  - Network

"If the kernel has security holes, then the system as a whole has holes"

Conceptos generales sobre la creación de módulos



Capítulo 2: Linux Device Drivers, Third Edition
by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

Introducción

- El módulo es dependiente de la versión de kernel y de la arquitectura
- Posee como mínimo dos funciones:
  - *module_init()*
  - *module_exit()*
- La macro *MODULE_LICENSE* define el tipo de licencia.
- Uso de bibliotecas del kernel *<include/linux>*
- *insmod* o *modprobe, lsmod* y *rmmod*
- Concurrencia: Múltiples usuarios pueden estar usando un mismo módulo -> *CODE must be reentrant*

The Hello World Module

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
printk(KERN_ALERT "Hola mundo!\n");
return0;
}

static void hello_exit(void)
{
printk(KERN_ALERT "Adiós mundo cruel\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Forma correcta de inicialización y apagado

- __init

```
static int __init initialization_function(void)
{
    /* Initialization code here */
}
module_init(initialization_function);
```

- __exit

```
static void __exit cleanup_function(void)
{
    /* Cleanup code here */
}

module_exit(cleanup_function);
```

user vs kernel space

- Interrupts are not available in user space.
- Direct access to memory is possible
- Access to I/O (/dev/port slow)
- Response time
- Swapped to disk

Drivers de Caracteres

Capítulo 3: Linux Device Drivers, Third Edition
by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

Los números Major and Minor

- El usuario accede a todo dispositivo a través de un *device file.*
- Todos los device files estan en */dev*
- Si se ejecuta el comando ls -l se observa:
  - primer letra: indica el tipo de contrlador (c - caracteres, b - bloques, etc).
  - número major y número minor
- Major: id controlador asociado al dispositivo
- Minor: usado por el kernel para id a qué dispositivo hace referencia.
  - se dev_t definida en <linux/types.h>, MAJOR(dev_t dev) y MINOR(dev_t dev);
- MKDEV(int major, int minor);

Script de asignación dinámica de Major and Minor

```
#!/bin/sh
module="scull"
device="scull"
mode="664"

# invoke insmod with all arguments we got
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod ./$module.ko $* || exit 1

# remove stale nodes
rm -f /dev/${device}[0-3]

major=$(awk "\\$2==\"$module\" {print \\$1}" /proc/devices)

mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

# give appropriate group/permissions, and change the group.
# Not all distributions have staff, some have "wheel" instead.
group="staff"
grep -q '^staff:' /etc/group || group="wheel"

chgrp $group /dev/${device}[0-3]
chmod $mode  /dev/${device}[0-3]
```

Estructuras de datos: file_operations

- file_operations(), definida en <linux/fs.h>
- consider the file to be an “object” and the functions operating on it to be its “methods,”

```
struct file_operations scull_fops = {
        .owner = THIS_MODULE,
        .llseek = scull_llseek,
        .read = scull_read,
        .write = scull_write,
        .ioctl = scull_ioctl,
        .open = scull_open,
        .release = scull_release,
};
```

Estructuras de datos: file

- definida en <*linux/fs.h*> y representa un archivo abierto
- creada por el kernel ante un llamado a *open()*, hasta un *close()*
- *filep* puntero a la estructura
- Campos más importantes:
  - *mode_t f_mode: FMODE_READ // FMODE_WRITE*
  - *unsigned loff_t f_pos: The current reading or writing position.*
  - *struct file_operations *f_op*
  - *f_flags: O_RDONLY, O_NONBLOCK y O_SYNC.*

Estructuras de datos: inode

- Used by the kernel internally to represent files. (**Not Open**)
- dev_t i_rdev:
    - para inodos que representan *device files*, contiene el número de dispositivo.
- struct cdev *i_cdev;
    - Puntero a la estructura interna del kernel que representa los dispositivos de caracteres.

Registro de Char Device

- Debe incluir <linux/cdev.h>, donde se define *struct cdev*
- Dos formas de inicializar y asignar esta estructura:
  - en tiempo de ejecución:

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

  - Llamando a cdev_init

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

- Una vez iniciada, se debe comunicar al kernel
-

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

  - num is the first device number to which this device (MAYOR), count la cantidad de MINORS. **void cdev_del(struct cdev *dev);**

Ejemplo

Iniciar un driver

```
struct scull_dev {
        structscull_qset*data;          /* Puntero al primer quantum set */
        intquantum;                     /* tamaño del quantum */
        intqset;                        /* tamaño del array */
        unsigned long size;             /* cantidad de datos almacenados aquí */
        unsigned int access_key;        /* usado por sculluid y scullpriv */
        struct semaphore sem;           /* semáforo de exclusión mutua */
        struct cdev cdev;               /* estructura de dispositivo de caracteres*/
};
```

```
static void scull_setup_cdev(struct scull_dev *dev, int index){
        int err, devno = MKDEV(scull_major, scull_minor + index);
        cdev_init(&dev->cdev, &scull_fops);
        dev->cdev.owner = THIS_MODULE;
        dev->cdev.ops = &scull_fops;
        err = cdev_add (&dev->cdev, devno, 1);
        /* Por si ocurre un error */
        If(err)
                printk(KERN_NOTICE "Error %d adding scull %d", err, index);
}
```

El método open

- Debe verificar si hay errores en el dispositivo, inicializarlo si es abierto por primera vez, actualizar el puntero *f_op* de ser necesario y llenar toda la estructura de datos apuntada por *filep->private_data*.
-
```
int(*open)(struct inode *inode, struct file *filp);
```
- No necesitamos el Inode, por eso usamos
-
```
container_of(pointer, container_type, container_field);
```
- Ej:

```
int scull_open (struct inode *inode, struct file *filp) {
        struct scull_dev *dev;              /* información del dispositivo */
        dev= container_of(inode->i_cdev, struct scull_dev, cdev);
        filp ->private_data = dev;          /* para otros métodos */
        /* poner a 0 la longitud del dispositivo si fue abierto para sólo-escritura */
        if( (filp->f_flags &O_ACCMODE) == O_WRONLY) {
                        scull_trim (dev);   /* ignorar errores */
        }
        return 0;                           /* si todo salió bien */
}
```

El método release

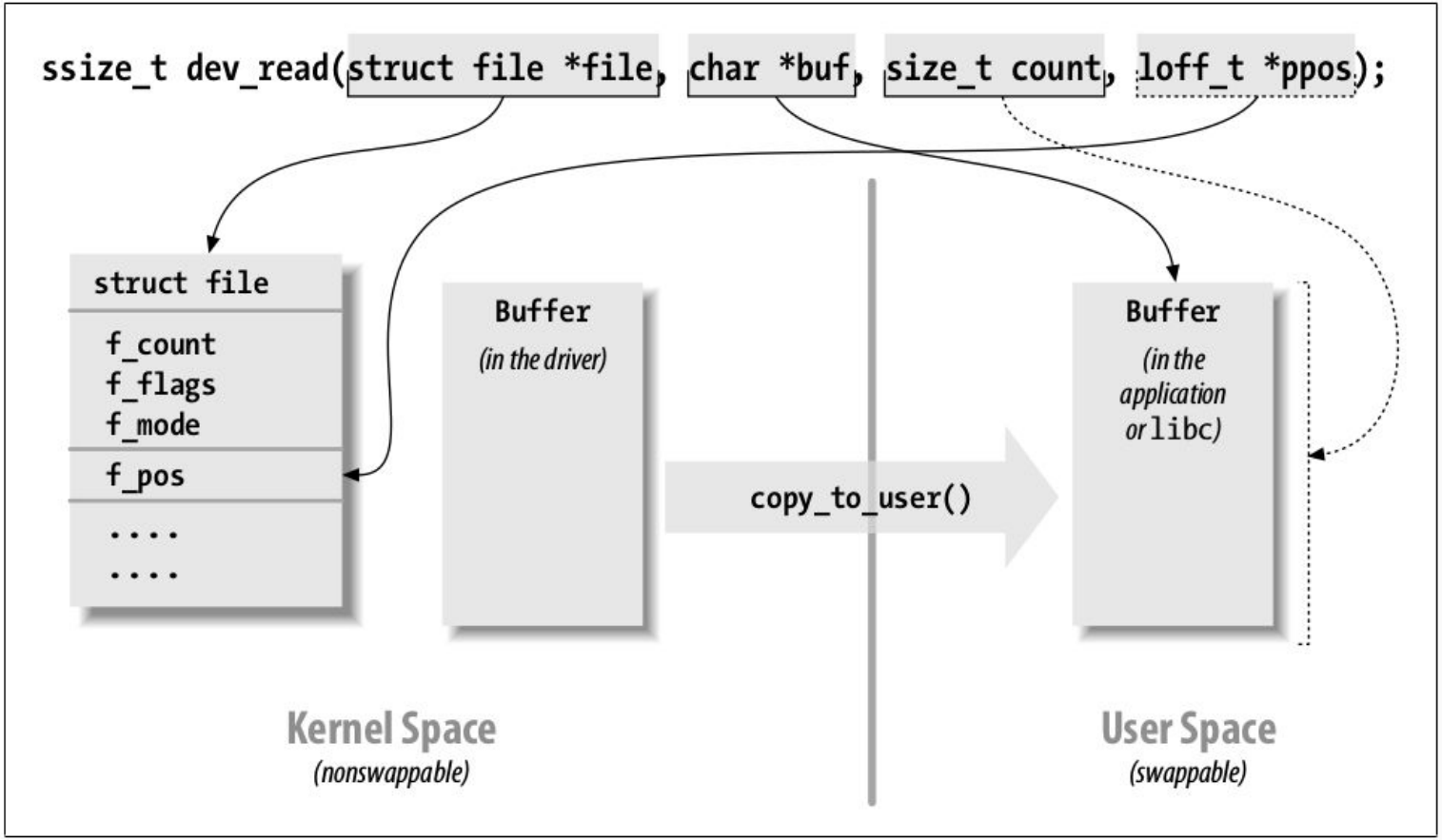
- Este método tiene que deshacer todo lo que hizo el open, es decir, liberar lo reservado en filp->private_data y apagar el dispositivo si es el último en cerrar.
- Es importante aclarar que el método release solo se llama si el contador que posee el kernel con la cantidad de veces que la estructura file está siendo usada es cero.

```
int scull_release (struct inode *inode, struct file *filp){
        return0;
}
```

Los métodos read y write

- Copying data from and to application code:
  - `ssize_t read(struct file *filp, char __user *buff,size_t count, loff_t *offp);`
  - `ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp);`
- *buff* argument is a user-space!!!

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


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)

El método read

```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos){
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr;                        /* the first listitem */
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset;                  /* Cuanto bytes */
        int item, s_pos, q_pos, rest;
        ssize_t retval = 0;
        if(down_interruptible(&dev->sem))
                        return -ERESTARTSYS;
        if(*f_pos >= dev->size)
                        goto out;
        if(*f_pos + count >dev->size)
                        count= dev->size - *f_pos;
        item= (long)*f_pos / itemsize;
        rest= (long)*f_pos % itemsize;
        s_pos = rest / quantum; q_pos = rest % quantum;
        dptr= scull_follow(dev, item);
        if(dptr == NULL || !dptr->data || ! dptr->data[s_pos])
                        goto out;                       /* don't fill holes */
        /* Leer hasta el final */
        if(count >quantum - q_pos)
                        count= quantum - q_pos;
        if(copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
                        retval= -EFAULT;
                        goto out;
        }
        *f_pos += count;
        retval= count;
        out:
        up(&dev->sem);
        return retval;
}
```

El método write

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos){
        struct scull_dev *dev = filp ->private_data;
        struct scull_qset *dptr;
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset;
        int item, s_pos, q_pos, rest;
        ssize_t retval = -ENOMEM;                    /* Condición de error */
        if(down_interruptible(&dev->sem))
                return -ERESTARTSYS;
        item= (long)*f_pos / itemsize;
        rest= (long)*f_pos % itemsize;
        s_pos = rest / quantum; q_pos = rest % quantum;
        dptr= scull_follow(dev, item);

        if (dptr == NULL)
                goto out;
        if(!dptr->data) {
                dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
                if(!dptr->data)
                        goto out;
                memset(dptr->data, 0, qset * sizeof(char *));
        }
        if (!dptr->data[s_pos]) {
                dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
                if (!dptr->data[s_pos])
                goto out;
        }
        if (count >quantum - q_pos)
                count= quantum - q_pos;
        if(copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
                retval= -EFAULT;
                gotoout;
        }
        *f_pos += count;
        retval= count;
        /* actualizar el tamaño */
        if (dev->size < *f_pos)
                dev->size = *f_pos;
        out:
        up(&dev->sem);
        return retval;
}
```

readv and writev

- Versiones vectoriales de read y write
- struct iovec definida en <linux/uio.h>
- Los prototipos son:

```
ssize_t (*readv) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t
*ppos);
ssize_t (*writev) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t
*ppos);
```

/proc Filesystem

- The /proc filesystem is a special, software-created filesystem that is used by the kernel to export information to the world. Each file under /proc is tied to a kernel function that generates the file's "contents" on the fly when the file is read. We have already seen some of these files in action; /proc/modules, for example, always returns a list of the currently loaded modules.

Timers y Tiempos en GNU/Linux

Capítulo 7: Linux Device Drivers, Third Edition
by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

El origen del tiempo

- Se define en *<linux/param.h>*
- Fuertemente dependiente de la arquitectura.
- La variable *HZ* define el incremento de TODA variable de tiempo, por segundo.
- Al modificar su valor, se debe compilar el *kernel*.
- *jiffies* y *jiffies_64* acumuladores de “ticks” desde que se booteo, que, al desbordar, emite una interrupción.
- Disponible en espacio de kernel, para acceso desde el espacio de user usar *timespec*.

Ejemplo de un delay

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;
j = jiffies;
stamp_1 = j + HZ; /* 1 segundo en el futuro*/
stamp_n = j + n * HZ / 1000; /* n milisegundos */
```

kernel timers

- el kernel provee una API para trabajar con los temporizadores (timers)
- n+1 timers
- standard (4 ms) y high-resolution (1 ns)

La API de temporizadores

- Permite crear, cancelar y gestionar temporizadores.
- Temporizadores definidos en la estructura *timer_list*
  - tiempo de expiración
  - callback function
  - contexto
- Inicializar un temporizador
  - *void init_timer(struct timer_list *timer);*
  - *void setup_timer(struct timer_list *timer, void (*function)(unsigned long), unsigned long data);*
- Establecer la expiración (en jiffies) y cancelarlo
  - *int mod_timer(struct timer_list *timer, unsigned long expires);*
  - *int del_timer(struct timer_list *timer);*
- Saber cuánto falta para la expiración
  - *int timer_pending(const struct timer_list *timer);*

La API de temporizadores

- Ejemplo: apiTimer.c

API de temporizadores de alta resolución (*hrtimer*)

- Utiliza la variable *ktime* en vez de *jiffies*
- Temporizadores definidos en la estructura *hrtimer*
  - tiempo de expiración
  - callback function
  - contexto
- Inicializar un temporizador, clocks definidos en *<include/linux/time.h>*
  - *void hrtimer_init(struct hrtimer *time, clockid_t which_clock, enum hrtimer_mode mode);*
  - *int hrtimer_start(struct hrtimer *timer, ktime_t time, const enum hrtimer_mode mode);*
- Establecer la expiración (en jiffies) y cancelarlo
  - *int hrtimer_cancel(struct hrtimer *timer);*
  - *int hrtimer_try_to_cancel(struct hrtimer *timer);*
- Saber si activó su función de callback
  - *int hrtimer_callback_running(struct hrtimer *timer);*

Mayor precisión

- Si la API de temporizadores es simple y eficiente, pero no brinda precisión para RTOS:
  - Sistemas externo (NTP).
  - Registros o mecanismos brindados por la arquitectura.

Lecturas

- *Linux Device Drivers*, Third Edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman
- https://www.ibm.com/developerworks/library/l-timers-list/