Sistemas Operativos

Gestión de Entrada y Salida

Agenda

Agenda

Introducción

El corazón de una computadora lo constituye la CPU, pero no serviría de nada sin:

- Dispositivos de almacenamiento no volátil:
  - Secundario: discos
  - Terciario: cintas
- Dispositivos periféricos que le permitan interactuar con el usuario (teclado, ratón, micrófono, cámara, etc.)
- Dispositivos de comunicaciones: permiten conectar a la computadora con otras a través de una red

Velocidad de los dispositivos

- La CPU procesa instrucciones a >1GHz (<1ns/ciclo)
- La CPU sólo puede leer de RAM (realmente L1)

Operation[1]	**Latency**		
L1 cache reference	0.5 ns		
Branch mispredict	5 ns		
L2 cache reference	7 ns		14xL1
Mutex lock/unlock	25 ns		
Main memory reference	100 ns		20xL2, 200xL1
Send 1K bytes over 1 Gbps network	10,000 ns	0.01 ms	
Read 1 MB sequentially from memory	250,000 ns	0.25 ms	
Read 1 MB sequentially from SSD	1,000,000 ns	1 ms	4xMem.
Disk seek	10,000,000 ns	10 ms	
Read 1 MB sequentially from disk	20,000,000 ns	20 ms	80xMem, 20xSSD
Send packet CA→Netherlands→CA	150,000,000 ns	150 ms	

[1] Latency Numbers Every Programmer Should Know

Visión del sistema de E/S

- La visión del sistema de E/S puede ser muy distinta dependiendo del nivel de detalle necesario en su estudio
  - Para los programadores: el sistema de E/S es una caja negra que lee y escribe datos en dispositivos externos a través de una funcionalidad bien definida
  - Para los fabricantes de dispositivos: un dispositivo es un instrumento muy complejo que incluye cientos o miles de componentes electrónicos o electro-mecánicos.
- Los diseñadores de sistemas operativos y drivers se encuentran en un lugar intermedio entre los dos anteriores:
  - Les interesa la funcionalidad del dispositivo, aunque a un nivel de detalle mucho mayor que el requerido por el programador de apps.
    - Necesitan información sobre su comportamiento interno para poder optimizar los métodos de acceso a los mismos
  - Requieren conocer la arquitectura del software de E/S del SO para poder exponer cada dispositivo al usuario

Funciones del sistema de E/S

- Facilitar el manejo de los dispositivos periféricos.
  - Ofreciendo una **interfaz** sencilla y estandarizada entre los dispositivos y el resto del sistema.
- Facilitar el desarrollo y mantenimiento de drivers.
  - Permitiendo una implementación gerárquica de drivers
  - Proporcionando dispositivos **virtuales** que permitan conectar cualquier tipo de dispositivo físico sin que sea necesario remodelar el sistema de E/S del sistema operativo
- Permitir la conexión de nuevos dispositivos E/S en caliente
  - Solventando de forma automática su instalación usando mecanismos del tipo **plug&play**
- Optimizar el sistema de E/S del sistema
  - Proporcionando mecanismos de incremento de prestaciones donde sea necesario

Agenda

Drivers (1/2)

- Componente software del SO destinado a gestionar un tipo específico de dispositivo de E/S
  - También llamado controlador SW o manejador de dispositivo
- Cada driver se divide en dos partes:
  - Código independiente del dispositivo para dotar al nivel superior del SO de una interfaz
    - Interfaz similar para acceso a dispositivos muy diferentes
    - Simplifica la labor de portar SSOO y aplicaciones a nuevas plataformas hardware
  - Código dependiente del dispositivo necesario para interactuar con dispositivo de E/S a bajo nivel
    - Interacción con controlador HW
    - Manejo de interrupciones
- Acciones comunes dispositivos E/S:
  - Un dispositivo puede generar y/o recibir datos
    - Operaciones de L/E en el driver
  - Algunos dispositivos pueden necesitar un control específico
    - Ejemplo: rebobinar una cinta
    - Operación de control en el driver
  - Muchos dispositivos generan interrupciones
    - Driver debe realizar procesamiento ligado a una interrupción

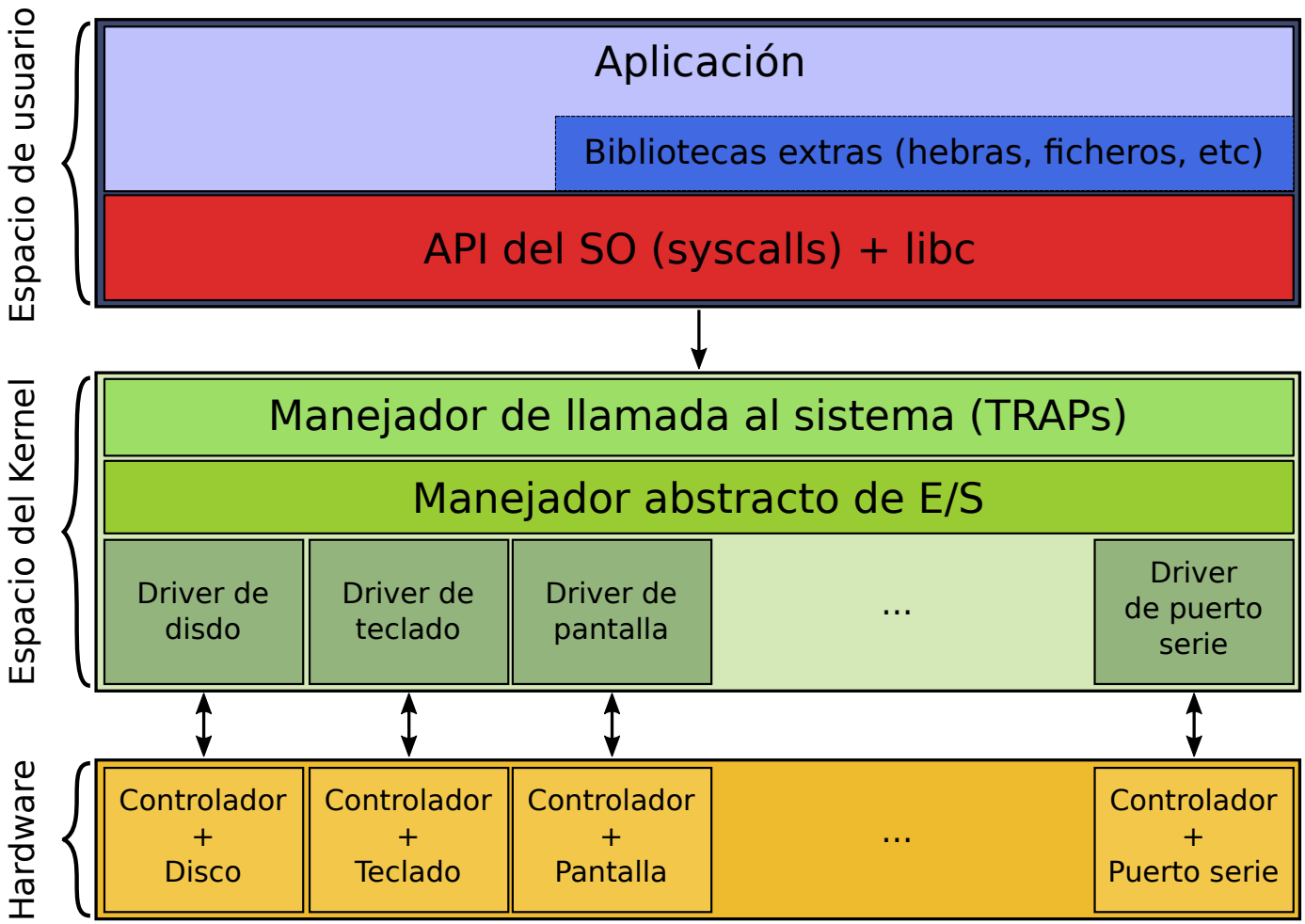
Drivers (2/2)

- Sin embargo, no podemos ser muy, muy genéricos
  - Por ejemplo, no podemos acceder a nivel de byte a un disco (acceso a nivel de bloque)
- Al final, los dispositivos se tienen que dividir en un pequeño número de clases:
  - Dispositivos orientados a/de carácter
    - puerto serie, teclado, ratón, ...
  - Dispositivos orientados a/de bloque
    - disco, pantalla, ...
- Estas clases o categorías se deben incluir para definir diferentes protocolos en la interfaz abstracta o genérica del driver y así ganar en rendimiento de la E/S

Nodos/Ficheros de dispositivo (1/2)

- Casi todos los dispositivos de E/S se exponen al programador como ficheros especiales llamados *nodos* o *ficheros de dispositivo*
  - Por convenio alojados en /dev: `/dev/sda1`, `/dev/tty0`, ...
- Permiten interaccionar con el driver del dispositivo utilizando el interfaz POSIX de ficheros
  - `open()`, `read()`, `write()`, `close()` e `ioctl()`
  - Pueden restringir el acceso al propietario, grupo y otros
- Un driver puede gestionar uno o más dispositivos
  - Implementa las operaciones del interfaz POSIX para estos dispositivos
- Los dispositivos se agrupan en clases
  - Identificadas con un número llamado **major** (Kernel.org)
- Para diferenciar entre los dispositivos de la misma clase gestionados por el driver se les asigna otro número, llamado **minor**
- Cada fichero de dispositivo tiene asociado un par **(major, minor)** que lo vincula de forma biunívoca con el driver que lo gestiona
  - El SO puede así delegar en el driver las operaciones del intrefaz POSIX realizadas sobre el fichero de dispositivo

Nodos/Ficheros de dispositivo (2/2)



Por cada fichero especial hay asociado un *driver* que realiza la tarea solicitada (ej. `read()`)

Dispositivos como ficheros

Los nodos son una potente abstracción del sistema

- Podemos trabajar con los dispositivos como si fuesen ficheros regulares
- Podemos utilizar con muchos dispositivos programas diseñados para operar sobre ficheros regulares sin ningún cambio

Por ejemplo, la utilidad `dd` está diseñada para hacer copias de un fichero a otro y, si queremos, realizar algún cambio sobre los datos que se copian

- Podemos usar la utilidad `dd` para leer los primeros 512 bytes desde el comienzo de un disco duro, el Master Boot Record (MBR), y almacenarlos en el archivo `mbr.bin`

```
# dd if=/dev/hda of=mbr.bin bs=512 count=1
```

- o para escribir ceros en todo el disco duro, eliminando toda la información existente

```
# dd if=/dev/zero of=/dev/hda
```

Creación de nodos/ficheros de dispositivo

Se puede crear un nodo usando el comando `mknod`:

```
# mknod /dev/<nombre> <tipo> <num_major> <num_minor>
```

- `<nombre>`: nombre de archivo de dispositivo
- `<tipo>`: `c` para dispositivos tipo carácter y `b` para tipo bloque
- `<num_major>`: número **major**, que identifica a la clase de dispositivo
- `<num_minor>`: número **minor**, que identifica a un dispositivo concreto de la clase

`stat` permite conocer los atributos de un nodo

```
$  stat /dev/tty1
File: '/dev/tty1'
Size: 0         Blocks: 0          IO Block: 4096  character special file
Device: 6h/6d   Inode: 20          Links: 1        Device type: 4,1
Access: (0620/crw--w----)  Uid: (    0/    root)   Gid: (    5/     tty)
Access: ...
$ stat /dev/sda1
File: '/dev/sda1'
Size: 0         Blocks: 0          IO Block: 4096  block special file
Device: 6h/6d   Inode: 167         Links: 1        Device type: 8,1
Access: (0660/brw-rw----)  Uid: (    0/    root)   Gid: (    6/    disk)
Access: ...
```

Asociación entre driver y major

- La asociación entre el `driver` del dispositivo y el `major` asignado puede consultarse en `/proc/devices`

- La mayor parte de los `drivers` de dispositivo se implementan como módulo cargable del kernel

Terminal

```
$ cat /proc/devices
Character devices:
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
6 lp
...
136 pts
180 usb
189 usb_device
...
Block devices:
2 fd
259 blkext
7 loop
8 sd
11 sr
65 sd
66 sd
67 sd
```

Agenda

Módulos

- Hay dos formas de incluir un driver en el kernel:
  - Enlazarlo estáticamente con el resto del kernel
    - Forma parte del binario del kernel
    - Para cambiarlo hay que recompilar todo el kernel
  - Compilarlo como módulo cargable del kernel, demorando el enlazado hasta el momento de la carga
    - Puede cargarse y/o descargarse bajo demanda y en caliente
    - Más conveniente que el enlazado estático, ya que permite añadir nueva funcionalidad al kernel cuando se necesite
- Gestión de módulos en Linux
  - `lsmod`: lista los módulos cargados actualmente.
  - `modinfo`: nos da información sobre un módulo.
  - `insmod`: carga un módulo. Interfaz de bajo nivel.
  - `rmmod`: descarga un módulo.
  - `modprobe`: interfaz de alto nivel para cargar módulos.
    - Busca en `/etc/modprobe` información y ruta de los módulos

Módulos vs. Aplicaciones

Módulos:

- Modo kernel
- Sólo símbolos exportados por el kernel:
  - `/proc/kallsyms`
  - `libc` no disponible (`printf`)
  - `printk()`: permite escribir mensajes en los ficheros de log y por terminal (tty), no pseudo terminal (pty).
- Ejecutan su función de inicio `'init_module'` al registrarse y quedan residentes para dar servicio

Aplicaciones:

- Modo usuario
- Cualquier función de biblioteca disponible
- Realizan su función de principio a fin (`main`)

Ejemplo: "Hello world"

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

MODULE_LICENSE("GPL");

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

printk

- En el kernel no podemos hacer E/S estándar con printf
- El kernel ofrece la función `printk`
  - Recibe como argumento una cadena con formato (tipo printf)
  - El primer carácter de la cadena indica el nivel de gravedad del mensaje
    - Se usan macros definidas como cadenas de caracteres con un código de log
    - El compilador concatena automáticamente dos literales de cadena que aparecen en el código consecutivamente
  - Los mensajes se imprimen en los ficheros de log del sistema
    - /var/log/messages en Debian
    - /var/log/syslog en (*)Ubuntu
  - El fichero `/proc/sys/kernel/printk` permite configurar un nivel por debajo del cual los mensajes serán mostrados también por la consola de inicio (no el terminal virtual)

```
Nivel de log máximo para mostrar el menaje por consola
# sudo sh -c "echo 8 > /proc/sys/kernel/printk"
```

Nivel de log

- Indicado con el primer carácter de la cadena de formato de printf
- Usado por el kernel para determinar la importancia del mensaje
- Se usan macros definidas en las librerías de cabecera del kernel (`linux/kern_levels.h`[1]):

```
#define KERN_SOH        "\001"          /* ASCII Start Of Header */
#define KERN_SOH_ASCII  '\001'

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

[1] Consultar directorio `/usr/src/linux-headers-$(uname -r)/include` para distribuciones tipo Debian con cabeceras instaladas

Makefile para compilación del módulo

Los módulos del kernel deben compilarse usando el makefile de las cabeceras del kernel instaladas en el sistema:

- Directorio `/lib/modules/$(shell uname -r)/build`
- Se añade a la variable obj-m el objeto correspondiente al módulo a compilar
- Se dispara la regla `modules` del makefile del kernel, poniendo la variable M al valor del path donde se encuentran las fuentes del módulo

```
obj-m += hello.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
```

Compilación y carga del módulo hello

Terminal 1

```
$ make
make -C /lib/modules/3.14.1.lin/build M=/mnt/hgfs/P4/FicherosP4/Hello modules
make[1]: se ingresa al directorio '/usr/src/linux-headers-3.14.1.lin'
  CC [M]  /mnt/hgfs/P4/FicherosP4/Hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M]  /mnt/hgfs/P4/FicherosP4/Hello/hello.ko
make[1]: se sale del directorio '/usr/src/linux-headers-3.14.1.lin'
$ sudo insmod hello.ko
$ lsmod | grep hello
hello                   836  0
$ sudo rmmod hello.ko
$ lsmod | grep hello
$
```

Terminal 2

```
sudo tail -f /var/log/kern.log
...
May  4 14:14:34 debian kernel: [  140.860137] Hello world.
May  4 14:20:14 debian kernel: [  481.981066] Goodbye world.
```

Agenda

Driver de un dispositivo orientado a caracteres

En la función de inicio el módulo debe:

- Crear una estructura `cdev`
- Registrar en esta estructura las funciones que implementarán las operaciones del interfaz POSIX de ficheros
- Reservar un rango `(major, minor)`
- Vincular dicho rango con la estructura `cdev`

En la función de descarga el módulo debe:

- Destruir estructura `cdev`
- Liberar el rango `(major, minor)`

Crear una estructura `cdev`

Puede ser declarada como una variable global estática:

```
struct cdev chardev;
```

O reservada dinámicamente:

```
struct cdev *chardev;
...
chardev = cdev_alloc();
```

Registro de operaciones

Para registrar las operaciones del interfaz POSIX necesitamos:

- Tener creada la estructura `cdev`
- Tener inicializada la escrutura `file_operations`

El registro se realiza con la función `cdev_init`:

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

Operaciones POSIX: `struct file_operations`

Para registrar las funciones que implementan cada una de las operaciones del interfaz POSIX se utiliza un `struct file_operations`:

- Definido en el fichero `linux/fs.h`
- Los campos más relevantes para nosotros son los punteros a función para cada una de las operaciones del interfaz POSIX de ficheros:

```
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 *)
     ;
  ...
  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 *, loff_t, loff_t, int datasync);
    ...
} __randomize_layout;
```

Campo owner

Marca el módulo propietario del dispositivo `cdev` correspondiente

- Puede usarse la macro `THIS_MODULE`
- El SO invocará la función `try_module_get` antes de invocar `open`
- Y la función `module_put` tras ejecutar la operación `release`

`try_module_get`: incrementa el contador de uso del móduo

- El SO impide descargar el módulo mientras ese contador sea mayor que 0
- Mientras algún proceso tenga abierto el fichero de dispositivo no se puede descargar el módulo del kernel

`module_put`: decrementa el contador de uso del móduo

El valor actual del contador de uso de un módulo podemos conocerlo ejecutando el comando `lsmod`

Reserva del rango (major,minor[s])

Dispositivos con major/minor[s] conocidos (Kernel.org):

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

- first: Primer par (major,minor) del driver
- count: Número de minor numbers a reservar para el driver
- name: Nombre del driver (cadena de caracteres arbitraria)
  - Es el valor que aparecerá en /proc/devices al cargar el driver

Dispositivos sin major/minors[s] conocidos:

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

- first: Parámetro de retorno. Primer par (major,minor) que el kernel reserva para el driver.
- firstminor: Menor minor number a reservar dentro del rango consecutivo que otorga el kernel
- count: Número de minor numbers a reservar para el driver
- name: Nombre del driver

Ambas funciones devuelven 0 o un código de error negativo.

Estructura `dev_t`

La estructura `dev_t` (32 bits) representa un par `(major, minor)`

- 12 bits para el `major`
- 20 bits para el `minor`

El empaquetamiento es complejo, se utilizan macros:

- Acceso: `MAJOR(dev_t dev)`, `MINOR(dev_t dev)`
- Construcción: `MKDEV(int major, int minor)`

Vinculación del (major,minor)

Para vincular un rango (major, minor[s]) a un dispositivo de caracteres, representado por la estructura cdev, usamos la función:

```
int cdev_add(struct cdev *p, dev_t first, unsigned count);
```

- p: la dirección de la estructura cdev
- first: el primer par (major, minor) que se quiere vincular
- count: el número de minors a vincular

A partir de este momento, las operaciones del interfaz POSIX serán delegadas en las operaciones registradas en el cdev

Desvinculación y liberación de `major/minor[s]`

Para desvincular el rango de `major/minor[s]` de la estructura `cdev` y liberar la memoria asociada a la estructura usaremos:

```
void cdev_del(struct cdev *p);
```

Para liberar el rango `major/minor[s]` reservado debemos usar:

```
#include <linux/fs.h>
int unregister_chrdev_region (dev_t first, unsigned int count)
```

- `first`: Primer par `(major,minor)` que el driver había reservado previamente.
- `count`: Número de minor numbers consecutivos que el driver había reservado.

La función devuelve 0 o un código de error negativo.

Ejemplo: dispositivo orientado a carácter

```
/*
 *  chardev.c: Creates a read-only char device that says how many times
 *  you've read from the dev file
 */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>  /* for copy_to_user */
#include <linux/cdev.h>

MODULE_LICENSE("GPL");
/*
 *  Prototypes
 */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *,const char *,size_t,loff_t *);
```

Ejemplo: dispositivo orientado a carácter

```
#define SUCCESS 0
#define DEVICE_NAME "chardev"/* Dev name as in /proc/devices   */
#define BUF_LEN 80           /* Max length of the message */

static dev_t start;
static struct cdev* chardev=NULL;
static int Device_Open = 0;/* Is device open?
                            * To prevent multiple access to device */
static char msg[BUF_LEN];  /* The msg the device will give when asked */
static char *msg_Ptr;      /* This will be initialized every time the
                              device is opened successfully */
static int counter=0;      /* Tracks the number of times the character
                            * device has been opened */
static struct file_operations fops = {
    .owner   = THIS_MODULE; // para que se haga el try_module_get
    .read    = device_read,
    .write   = device_write,
    .open    = device_open,
    .release = device_release
};
```

Ejemplo: dispositivo orientado a carácter

```
int init_module(void){
    ...
    /* Get available (major,minor) range */
    if (ret = alloc_chrdev_region (&start, 0, 1,DEVICE_NAME)) {
    ...
    }
    /* Create associated cdev */
    if ((chardev = cdev_alloc()) == NULL) {
    ...
    }
    cdev_init(chardev,&fops);
    if (ret = cdev_add(chardev,start,1)) {
    ...
    }
    ...
    major=MAJOR(start);
    minor=MINOR(start);
    printk(KERN_INFO "I was assigned major number %d.\n", major);
    printk(KERN_INFO "To talk to the driver, create a dev file with\n");
    ...
    return SUCCESS;
}
```

Ejemplo: Operación open

Si hay alguna otra apertura del nodo se devuelve error, dispositivo ocupado.

```
static int device_open(struct inode *inode, struct file *file){
  if (Device_Open)
    return -EBUSY;
  Device_Open++;
  /* Initialize msg */
  sprintf(msg, "I already told you %d times Hello world!\n", counter++);
  /* Initially, this points to the beginning of the message */
  msg_Ptr = msg;
  return SUCCESS;
}
static int device_release(struct inode *inode, struct file *file){
  Device_Open--;    /* We're now ready for our next caller */
  return 0;
}
```

Recordar, como el campo `.owner` se puso a `THIS_MODULE` el SO invocará `try_module_get` antes de ejecutar `device_open`.

Copias modo kernel - modo usuario

El módulo debe tener cuidado al copiar a datos a/desde direcciones suministradas por los procesos de usuario:

- El módulo ejecuta en modo kernel
- Por tanto puede acceder a direcciones que el proceso en modo usuario no podría acceder
  - Por ejemplo una página del kernel

El kernel proporciona dos funciones que antes de hacer la copia comprueban si en modo usuario se tendría permiso, y fallan si no es así:

```
//Copia n bytes del buffer de kernel from al buffer de usuario to
unsigned long copy_to_user(void *to,const void *from,unsigned long n)

//Copia n bytes del buffer de usuario from al buffer de kernel to
unsigned long copy_from_user(void *to,const void *from,unsigned long n)
```

Ambas funciones devuelven el número de bytes que **NO** pudieron copiar

Para copiar una sola variable pueden usarse también las macros:

```
get_user(x, ptr); // ptr dirección de usuario, x variable de kernel
put_user(x, ptr); // ptr dirección de kernel, x variable de usuario
```

Ejemplo: Operación read

```
static
ssize_t device_read(struct file *filp, char *buffer, size_t length,
                    loff_t *offset)
{
    int bytes_to_read = length;

    if (*msg_Ptr == 0)
        return 0;

    if (bytes_to_read > strlen(msg_Ptr))
        bytes_to_read = strlen(msg_Ptr);

    if ( copy_to_user(buffer, msg_Ptr, bytes_to_read) )
        return -EFAULT;

    msg_Ptr+=bytes_to_read;

    return bytes_to_read;
}
```

Ejemplo: carga y uso del módulo chardev

Compilación, carga y uso del módulo

```
$ make
...
$ sudo su
[sudo] password for usuarioso:
# insmod chardev.ko
# mknod -m 666 /dev/chardev c 250 0
# cat /proc/devices | grep chardev
250 chardev
# cat /dev/chardev
I already told you 0 times Hello world!
# cat /dev/chardev
I already told you 1 times Hello world!
# echo "Hello" > /dev/chardev
bash: echo: error de escritura: Operación no permitida
# rmmod /dev/chardev
```

Mensajes de log del módulo

```
$ sudo tail -f /var/log/kern.log
debian kernel: [  282.604598] I was assigned major number 250. To talk to
debian kernel: [  282.604602] the driver, create a dev file with
debian kernel: [  282.604605] 'sudo mknod -m 666 /dev/chardev c 250 0'.
debian kernel: [  282.604606] Try to cat and echo to the device file.
debian kernel: [  282.604608] Remove the device file and module when done.
debian kernel: [  354.964761] Sorry, this operation isn't supported.
```