Sistemas Operativos

Gestión de Entrada y Salida

Organización y jerarquía de drivers

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. Para ello debe ofrecer una **interfaz** entre los dispositivos y el resto del sistema que sea sencilla y fácil de utilizar
- **Optimizar** la E/S del sistema, proporcionando mecanismos de incremento de prestaciones donde sea necesario
- Proporcionar 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 dispositivos nuevos de E/S, solventando de forma automática su instalación usando mecanismos del tipo **plug&play**

Drivers I

- 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

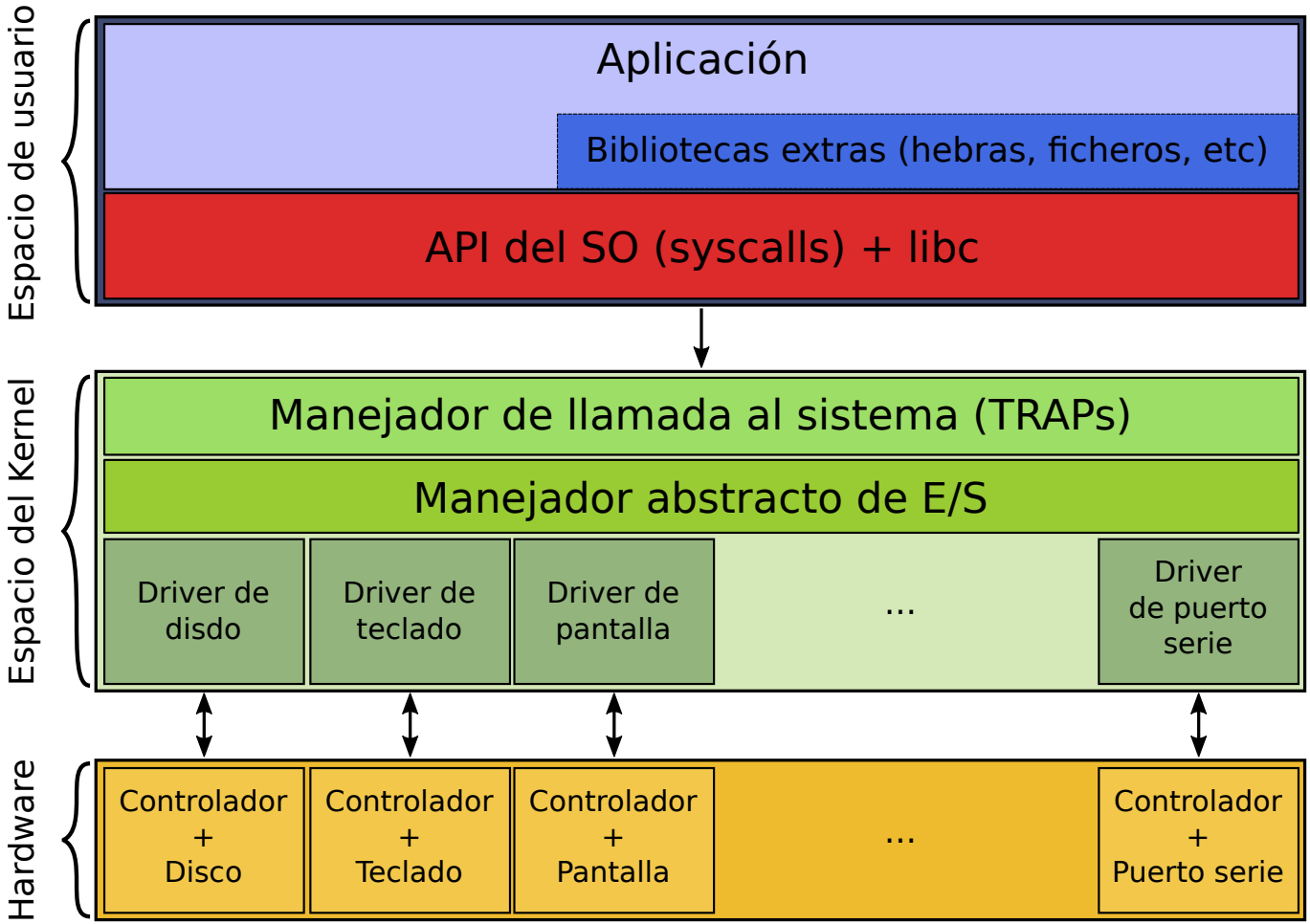
Drivers II

- 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 de carácter
    - puerto serie, teclado, ratón, ...
  - Dispositivos 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

Ficheros de dispositivo I

- Casi todos los dispositivos de E/S se representan como ficheros especiales (que pueden ser de bloque o carácter)
  - `/dev/sda1` para la primera partición del primer disco SATA o USB
  - `/dev/tty0` para el primer terminal/consola de texto
  - `/dev/lp0` para la impresora
- El acceso a estos ficheros especiales se realiza mediante las llamadas al sistema `open()`, `read()`, `write()` y `close()`
  - Un programa de usuario puede acceder al dispositivo de E/S, siempre y cuando el usuario tenga permisos de acceso al fichero especial
  - Excepcionalmente puede requerirse `ioctl()` para realizar operaciones de control

Ficheros de dispositivo II



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

Ficheros de dispositivo III

- Los ficheros de dispositivo se alojan *por convenio* en el directorio /dev
- Para ver los ficheros de dispositivo presentes en el sistema basta con listar el contenido del directorio /dev

Terminal

```
$ ls -l /dev
...
brw-r----- 1 root disk   3,  0 Nov 19 10:20 hda
brw-r----- 1 root disk   3,  1 Nov 19 10:20 hda1
brw-r----- 1 root disk   3,  2 Nov 19 10:20 hda2
...
crw-rw---- 1 root uucp   4, 64 Nov 19 10:20 ttyS0
crw-rw---- 1 root uucp   4, 65 Nov 19 10:20 ttyS1
...
crw-rw---- 1 root audio 14,  3 Dec  2 00:31 dsp
crw-rw---- 1 root audio 14,  4 Dec  2 00:31 audio
...
crw-rw-rw- 1 root root   1,  8 Nov 19 10:20 random
```

Ficheros de dispositivo IV

- Los ficheros de dispositivo son un potente mecanismo de trabajo con los dispositivos hardware tal y como si fuesen ficheros ordinarios:

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

  - **Descripción**: El comando leerá los primeros 512 bytes desde el comienzo del disco duro, el Master Boot Record (MBR), y lo almacenará en el archivo `mbr.bin`

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

  - **Descripción**: Escribe ceros en todo el disco duro, eliminando toda la información existente
- ¿Cómo sabe el SO a qué dispositivo está asociado un fichero de dispositivo? $\Rightarrow$ `(major, minor)`

Ficheros de dispositivo V

- Se puede crear un nuevo fichero de dispositivo usando el comando `mknod`:

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

- donde:
  - `<nombre>`: nombre de archivo de dispositivo
  - `<tipo>`: `c` para dispositivos tipo carácter y `b` para tipo bloque
  - `<num_major>` y `<num_minor>`: `major` y `minor` del `driver` del dispositivo al que este fichero queda asociado
- Se pueden crear ficheros de dispositivo con cualquier `major` y `minor`, sólamente útil si existe un driver asociado con los mismos números.

Major and minor numbers I

- Los dispositivos se agrupan en clases. Cada clase tiene un número de dispositivo principal (`major`) que la identifica:
  - Se pueden consultar en Kernel.org
  - Cada fichero de dispositivo tiene asociado un par (`major, minor`) que lo identifica de forma biunívoca
  - `major`: ID de la clase de dispositivos a la que pertenece
  - `minor`: ID local para que el driver pueda distinguir al dispositivo en caso de gestionar varios
- Ejemplo: un driver que gestiona 2 discos duros
  - Discos representados mediante ficheros de dispositivo
    - `/dev/sda`, `/dev/sdb`
  - Ambos ficheros especiales tendrán el mismo `major` number pero distinto `minor number`

Major and minor numbers II

- El comando `stat` permite consultar el tipo de dispositivo asociado al fichero así como el `major` y `minor number` del mismo:

Terminal

```
$  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: ...
```

Representación de (major, minor)

- En el kernel Linux el par (major,minor) está representado mediante el tipo dev_t
  - dev_t: número de 32 bits (12 bits major, 20 bits minor)
- Por motivos históricos el empaquetamiento es complejo, se utilizan macros:
  - Acceso a números:
    - MAJOR(dev_t dev), MINOR(dev_t dev)
  - Construcción de par:
    - MKDEV(int major, int minor)

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
```

Módulos

- Un driver puede ser estáticamente enlazado "dentro" del kernel o compilado como módulos del kernel.
  - Un módulo es una parte del kernel que puede cargarse y descargarse del kernel bajo demanda.
  - Resulta 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");
}
```

Niveles de `log` I

- El nivel de log es usado por el kernel para determinar la importancia del mensaje. Están definidos 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

Niveles de `log` II

- Los niveles de log (y consola no virtual) pueden ser consultados y modificados a través `/proc`:

Terminal

```
$ cat /proc/sys/kernel/printk
4           4           1           7
#current    default     minimum     boot-time-default
```

  - El primer entero es `console_loglevel`. Los mensajes con valor menor que `console_loglevel` aparecerán en la consola (no virtual).
- Para que todos los mensajes aparezcan por consola podemos hacer:

Terminal

```
# sudo sh -c "echo 8 > /proc/sys/kernel/printk"
```

Ejemplo: compilación

- Los módulos del kernel deben compilarse de forma diferente a los ficheros C habituales

```
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
```

Ejemplo: ejecución

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.
```

Implementar un driver como módulo

- Crear un módulo del kernel con funciones `init_module()` y `cleanup_module()`
- Implementar las operaciones de la interfaz del dispositivo de caracteres: `struct file_operations`
- En la función de inicialización:
  - Reservar `major number` y rango de `minor numbers` para el driver
    - `alloc_chrdev_region()`
  - Crear una estructura `cdev_t` y asociarle las operaciones y el rango de `major/minor`
    - Usar `cdev_alloc()`, `cdev_init()` y `cdev_add()`
- En la función de descarga:
  - Destruir estructura `cdev_t`: `cdev_del()`
  - Liberar el rango `(major, minor)`: `unregister_chrdev_region()`

Estructura `file_operations`

- La encontramos en el fichero `linux/fs.h`.
- Los campos más relevantes para nosotros son:

```
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;
```

Operaciones soportadas

- Sólo es necesario inicializar los campos de la estructura `file_operations` que se corresponden con operaciones soportadas por el dispositivo, por ejemplo:

```
struct file_operations fops = {
  .read    = device_read,
  .write   = device_write,
  .open    = device_open,
  .release = device_release
};
```

Registro de `major/minor[s]`

- Para dispositivos con `major/minor[s]` conocidos (Kernel.org) podemos usar:

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

  - Parámetros
    - `first`: Primer par (`major`,`minor`) que el driver desea reservar.
    - `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
  - Valor de retorno
    - 0 en caso de éxito.
    - En caso de fallo, devuelve valor negativo que codifica el error.

Reserva de `major/minor[s]`

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

- Parámetros
  - `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 (cadena de caracteres arbitraria.) Es el valor que aparecerá en `/proc/devices` al cargar el driver
- Valor de retorno
  - 0 en caso de éxito.
  - En caso de fallo, devuelve valor negativo que codifica el error.

Liberación de `major/minor[s]`

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

- Parámetros
  - `first`: Primer par `(major,minor)` que el driver había reservado previamente.
  - `count`: Número de minor numbers consecutivos que el driver había reservado.
- Valor de retorno
  - 0 en caso de éxito.
  - En caso de fallo, devuelve valor negativo que codifica el error.

Estructura `cdev`

- Necesaria para que el driver pueda recibir peticiones de los programas de usuario. Utilidades:
  - Crear estructura `cdev` (retorna un puntero no nulo a la misma en caso de éxito):

    ```
    struct cdev *cdev_alloc(void);
    ```
  - Asociar interfaz de operaciones del driver a estructura `cdev`:

    ```
    void cdev_init(struct cdev *p,struct file_operations *
    fops);
    ```
  - Permitir que peticiones de programas de usuario sobre el rango de (`major`,`minor`) especificado mediante parámetros `first` y `count` sean redirigidas al driver que gestiona estructura `cdev`:

    ```
    int cdev_add(struct cdev *p, dev_t first, unsigned
    count);
    ```
  - Eliminar asociaciones de estructura `cdev` (rangos de `major`/`minor`) y libera memoria asociada a la estructura:

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

Dispositivo tipo carácter I

```
/*
 *  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
    *);
```

Dispositivo tipo carácter II

```
#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices
       */
#define BUF_LEN 80            /* Max length of the message from the
    device */
/*
 * Global variables are declared as static, so are global within the
    file.
 */
dev_t start;
struct cdev* chardev=NULL;
static int Device_Open = 0; /* Is device open?
                             * Used 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 = {
    .read    = device_read,
    .write   = device_write,
```

Dispositivo tipo carácter III

```
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. To talk to\n",
     major);
    printk(KERN_INFO "the driver, create a dev file with\n");
    printk(KERN_INFO "'sudo mknod -m 666 /dev/%s c %d %d'.\n",
     DEVICE_NAME,
                                                        major,
     minor);
    printk(KERN_INFO "Try to cat and echo to the device file.\n");
    printk(KERN_INFO "Remove the device file and module when done.\n");

    return SUCCESS;
}
```

Dispositivo tipo carácter IV

```
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;
    /* Increase the module's reference counter */
    try_module_get(THIS_MODULE);
    return SUCCESS;
}
static int device_release(struct inode *inode, struct file *file){
    Device_Open--;    /* We're now ready for our next caller */
    /*
     * Decrement the usage count, or else once you opened the file, you'
     ll
     * never get get rid of the module.
     */
    module_put(THIS_MODULE);
    return 0;
}
```

Dispositivo tipo carácter V

```
static ssize_t device_read(struct file *filp, /* see include/linux/fs.h
      */
                           char *buffer,      /* buffer to fill with
    data */
                           size_t length,     /* length of the buffer
      */
                           loff_t * offset){
   int bytes_to_read = length;  // bytes actually written to the buffer

   if (*msg_Ptr == 0) //If we're at the end of the message return 0 (
    EOF)

   if (bytes_to_read > strlen(msg_Ptr)) // we don't read more chars
       bytes_to_read=strlen(msg_Ptr);   // than those remaining to read

   // Actually transfer the data onto the userspace buffer.
   // For this task we use copy_to_user() due to security issues
   if ( copy_to_user( buffer, msg_Ptr, bytes_to_read) )
       return -EFAULT;

   msg_Ptr+=bytes_to_read; // Update the pointer for the next read
    operatio
```

Dispositivo tipo carácter VI

```
/*
 * Called when a process writes to dev file: echo "hi" > /dev/chardev
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t *
     off){
    printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
    return -EPERM;
}

/*
 * This function is called when the module is unloaded
 */
void cleanup_module(void){
    /* Destroy chardev */
    if (chardev)
        cdev_del(chardev);
    /*
     * Unregister the device
     */
    unregister_chrdev_region(start, 1);
}
```

Copia segura del/al espacio de usuario

- Las operaciones `read` y `write` de un fichero de dispositivo tienen como parámetro un puntero `buffer` del espacio de usuario
- No debemos confiar en los punteros del espacio de usuario (puede pertenecer a una región de memoria a la que el proceso asociado al driver no tenga acceso)
- Siempre se ha de trabajar con una copia privada de los datos en el espacio del kernel, usando:

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

  - Devuelven el número de bytes que NO pudieron copiarse

Ejecución

Chardev

```
$ 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
```

Log

```
$ 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.
```