Periféricos Interfaces y Buses

I. Arquitectura de E/S

II. Programación de E/S

Aspectos básicos de la programación de E/S. Arquitectura y programación de la E/S en el sistema operativo. Manejadores de dispositivos (drivers) y su programación (interrupciones).

III. Interfaces de E/S de datos

IV. Dispositivos de E/S de datos

V. Buses

VI. Controladores e interfaces de dispositivos de almacenamiento

VII. Sistemas de almacenamiento

II. Programación de E/S

Bloque I

- Aspectos básicos de la programación de E/S
- Arquitectura y programación de la E/S en el sistema operativo
- Módulos de núcleo en Linux
- Organización de manejadores de dispositivos (*drivers*)
- Gestión de memoria

Aspectos básicos de la programación de entrada/salida

Tres tipos de programación de la E/S:

- Entrada/salida por consulta o programada
- Entrada/salida por interrupciones
- Entrada/salida por acceso directo a memoria

El control de la entrada/salida recae fundamentalmente en el sistema operativo:

- la programación se realiza mediante los *drivers* de dispositivos (*device drivers*)
- las interfaces que ofrecen los sistemas operativos para la programación de drivers no son uniformes

Drivers de dispositivos

Un driver para un dispositivo es una interfaz entre el sistema operativo y el hardware de un dispositivo

Los drivers forman parte del kernel y tienen acceso restringido a estructuras del sistema operativo

El objetivo de un driver debe ser flexibilizar el uso de los dispositivos, proporcionando un mecanismo de uso y no una política de uso (idea que proviene del diseño de Unix):

- mecanismo: capacidades que debe proporcionar
- política: cómo utilizar las capacidades que proporciona

Drivers de dispositivos (cont.)

Ej. 1: los gráficos en Unix se dividen en:

- el *X server* que maneja el hardware y ofrece una interfaz única a los programas de usuario, y
- los *managers* de windows y sesión que implementan unas políticas concretas sin conocer absolutamente nada del hardware

Ej. 2: para los niveles TCP/IP de red el sistema operativo ofrece:

- la abstracción del *socket* que no implementa política alguna sobre el uso de la red, y
- son los servicios que están por encima los que implementan las políticas de uso

Drivers de dispositivos (cont.)

En la implementación de drivers intentaremos observar esta separación en la medida de lo posible (escribir el código de acceso al hardware pero no forzar un uso particular)

Otros aspectos a tener en cuenta:

- la concurrencia en el uso de los drivers
- operación síncrona o asíncrona de los drivers
- la capacidad de ser abiertos muchas veces

Nos vamos a centrar en los sistemas operativos:

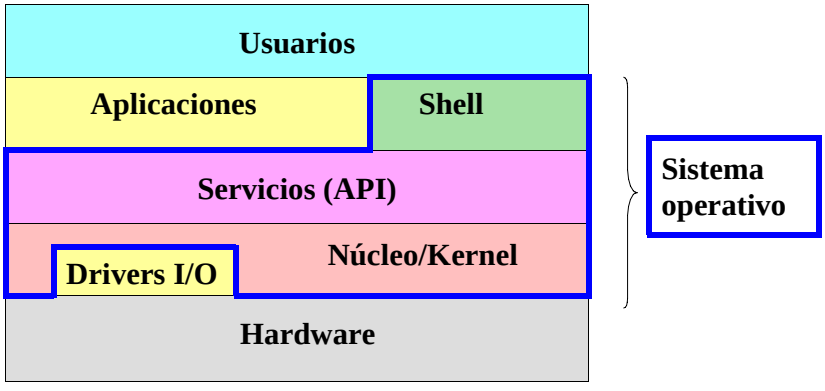
- Linux: es un SO libre con interfaz POSIX, no es de tiempo real
- MaRTE OS: es un SO de tiempo real que implementa el perfil mínimo de POSIX

II. Programación de E/S

Bloque I

- Aspectos básicos de la programación de E/S
- Arquitectura y programación de la E/S en el sistema operativo
- Módulos de núcleo en Linux
- Organización de manejadores de dispositivos (*drivers*)
- Gestión de memoria

El driver en el contexto del sistema operativo



Partición del kernel de Linux

Manejo de procesos:

- creación y destrucción de procesos, entrada/salida de los mismos, comunicación entre procesos (señales, pipes, o primitivas de intercomunicación), planificación de los procesos (cómo comparten el uso de la CPU)

Manejo de memoria:

- el kernel implementa un espacio de direcciones virtuales para cada uno de los procesos encima de los recursos disponibles que son limitados
- diferentes partes del kernel interaccionan a través de un conjunto de llamadas del sistema correspondientes al subsistema de manejo de memoria (ej.: *malloc/free*)

Partición del kernel de Linux (cont.)

Sistema de ficheros:

- casi todo en Unix puede ser tratado como un fichero
- el kernel construye una sistema de ficheros estructurado encima de hardware desestructurado
- Linux soporta varios sistemas de ficheros

Control de dispositivos:

- casi todas las operaciones del sistema se pueden mapear en un dispositivo físico
- salvo el procesador, la memoria, y algún otro elemento, las operaciones de control de los dispositivos se realizan mediante drivers de dispositivos; así cada dispositivo presente en el sistema debe tener su driver

Partición del kernel de Linux (cont.)

Redes:

- las redes deben ser manejadas por el SO porque la mayoría de las operaciones de red no son específicas de un proceso
- la llegada de paquetes es un evento asíncrono
- el paquete debe ser recogido, identificado y despachado antes de que el proceso correspondiente lo reciba

Linux tiene la capacidad de extender la funcionalidad del kernel en tiempo de ejecución mediante lo que denomina módulo (*module*):

- objetos software con una interfaz bien definida, que se cargan dinámicamente en el núcleo del sistema operativo
- el propio kernel ofrece servicios para su instalación y desinstalación

Organización de drivers de entrada/ salida

Los drivers se implementan mediante módulos del núcleo

Los drivers forman parte del núcleo y tienen

- acceso completo al hardware
- acceso restringido a estructuras del sistema operativo

El objetivo del driver es ofrecer un mecanismo de uso general, con operaciones como:

- abrir (*open*) y cerrar (*close*)
- leer (*read*)
- escribir (*write*)
- controlar (*ioctl*)

Tipos de dispositivos y módulos

En Linux 2.6 se distinguen tres tipos de dispositivos y módulos:

- *de caracteres*: E/S directa o por interrupciones
- *de bloques*: E/S por acceso directo a memoria
- *de red*: E/S por dispositivos de comunicaciones

Esta clasificación no es rígida y se podría considerar otra ortogonal como:

- módulos USB, módulos serie, módulos SCSI (*Small Computer Systems Interface*), etc.
- cada dispositivo USB estaría controlado por un módulo USB, pero el dispositivo en sí mismo se comportaría como de caracteres (puerto serie USB), de bloques (tarjeta de memoria USB), o una interfaz de red (interfaz Ethernet USB)

Dispositivos de caracteres

Se puede utilizar como si se tratara de un fichero, como una tira o conjunto de bytes

Normalmente implementa las llamadas al sistema: *open*, *close*, *read* y *write*

Ejemplos de dispositivos de caracteres:

- consola (`/dev/console`)
- puertos serie (`/dev/ttyS0`)

El acceso a estos dispositivos se realiza a través de nodos (*nodes*) del sistema de ficheros

Dispositivos de caracteres (cont.)

La diferencia fundamental con los ficheros es que en ellos te puedes desplazar hacia delante y hacia atrás, mientras que los drivers son normalmente canales de datos a los que se accede secuencialmente

Puede haber dispositivos de caracteres en los que aparecen áreas de datos, como por ejemplo en la adquisición de imágenes por trozos:

- cada trozo puede tener varios elementos, bytes, accesibles
- podría necesitar la implementación de otras funciones de la interfaz de drivers

Dispositivos de bloques

Al igual que los dispositivos de caracteres se acceden a través de nodos del sistema de ficheros en el directorio `/dev`

Pueden contener y controlar un sistema de ficheros (como por ejemplo un disco)

La información se maneja en bloques normalmente de 512 bytes o alguna otra potencia de 2

Las operaciones de lectura y escritura funcionan como en un disposivo de caracteres permitiendo la transferencia de cualquier cantidad de bytes:

- sólo difieren en el modo en que el kernel maneja los datos internamente y en la interfaz software hacia el kernel

Interfaces de red

Cualquier transacción de red se hace a través de una interfaz para el intercambio de información con otros equipos

Normalmente una interfaz es un dispositivo hardware, pero podría ser únicamente software puro

Una interfaz de red se encarga de enviar y recibir paquetes, y también de encaminarlos por el subsistema de red del kernel:

- con independencia de las aplicaciones emisora y receptora
- por tanto, un driver de red en principio no debe saber nada de las conexiones individuales, sólo debe manejar paquetes

Interfaces de red (cont.)

Por ejemplo:

- FTP o Telnet pueden usar el mismo dispositivo para transmitir sus cadenas de datos (*streams*)
- el dispositivo sólo ve paquetes de datos a transmitir y no los *streams*

Los driver de red no tienen acceso a través de un fichero especial en */dev*:

- en lugar de *read* o *write* tienen operaciones de transmisión de paquetes

II. Programación de E/S

Bloque I

- Aspectos básicos de la programación de E/S
- Arquitectura y programación de la E/S en el sistema operativo
- Módulos de núcleo en Linux
- Organización de manejadores de dispositivos (*drivers*)
- Gestión de memoria

Módulos de Linux

El módulo en Linux debe tener al menos dos operaciones:

- *init_function*: para instalarlo
  - debe preparar el módulo para la posterior ejecución del resto de las funciones o puntos de entrada del módulo
- *cleanup_function*: para desinstalarlo
  - debe eliminar todo rastro del módulo

El código de un módulo sólo debe llamar a funciones incluidas en el kernel y no en librerías, ya que el módulo se enlaza directamente con el kernel.

Módulos de Linux (cont.)

Los pasos para construir un módulo de Linux serían:

1. editar el un fichero con las operaciones de inicialización y finalización del módulo
   - tenemos un `fichero.c`
2. compilar el módulo
   - se puede editar un *Makefile* para ahorrar trabajo
   - obtenemos el `fichero.ko`
3. cargar el módulo
   - con `insmod`

Para descargar el módulo se hace con `rmmod`

Ejemplo de módulo Linux

Fichero *sencillo.c*:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");

static int instala (void)
{
  printk(KERN_ALERT "Sencillo instalado\n");
  return 0;
}

static void desinstala (void)
{
  printk (KERN_ALERT "Sencillo desinstalado\n");
}

module_init(instala);
module_exit(desinstala)
```

Estructura de un módulo del núcleo

- "*Includes*" para las funciones de los módulos de núcleo
- `MODULE_LICENSE`: Macro con el tipo de licencia (un texto)
- *Dos funciones*: una de instalación y otra de desinstalación, con los prototipos como en el ejemplo
- `module_init`: con esta macro declaramos cuál de nuestras funciones es la de instalación
- `module_exit`: con esta macro declaramos cuál de nuestras funciones es la de desinstalación

Compilación de módulos de Linux

La compilación:

- Se hace con un `make`, creando un fichero `Makefile` con la siguiente línea (`sencillo.o` es el nombre del fichero objeto):

```
obj-m := sencillo.o
```

- La orden `make` debe incluir el contexto del kernel y una orden que le indique el directorio de trabajo:

```
make -C /lib/modules/linux-2.6.15.23/build M=`pwd` modules
```

- donde hay que reemplazar `/lib/modules/linux-2.6.15.23` por el directorio donde se hallan los fuentes del kernel
  - es necesario disponer de las cabeceras de las fuentes del kernel
- el módulo recibe el nombre `sencillo.ko`

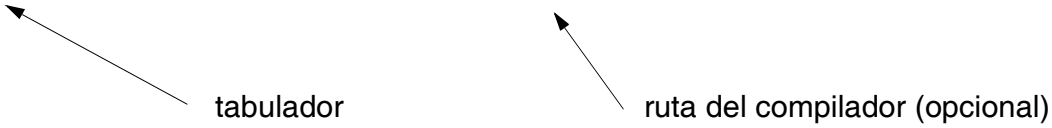
Compilación de módulos de Linux (cont.)

Es posible facilitar la compilación indicando la información anterior dentro del fichero `Makefile`:

```
obj-m := sencillo.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
CC = /usr/bin/gcc-4.0

default:
    $(MAKE) -C $(KDIR) CC=$CC SUBDIRS=$(PWD) modules
```



Cabeceras de las fuentes del kernel

Se suelen encontrar en un paquete aparte

- `linux-headers-2.6.xx.xx`

Comprobar también que están instaladas las utilidades para carga y descarga dinámica de módulos

- `module-init-tools`

Instalación o desinstalación de módulos

- se debe hacer en modo superusuario (*root*)
- se instala con `insmod` (que, además, ejecuta la `init_function`):
- se desinstala mediante `rmmod` (ejecuta la `cleanup_function`):

```
/sbin/insmod sencillo.ko
/sbin/rmmod sencillo
```

- la orden `lsmod` muestra los módulos actualmente instalados
  - lee el fichero `/proc/modules`
  - muestra la siguiente información: nombre, tamaño, contador de uso y lista de módulos referidos

Depuración en módulos de Linux

No se pueden usar las llamadas normales al sistema operativo desde dentro del núcleo

- `printk`: similar al `printf`, pero ejecutable desde el núcleo; escribe en la consola y en `/var/log/messages`
  - la macro `KERN_ALERT` le da prioridad alta al mensaje (linux/kernel.h)
  - se puede consultar con

```
tail /var/log/messages
dmesg
```

Las funciones que se pueden usar desde el kernel se declaran en los directorios: include/linux y en include/asm

situados en las fuentes del kernel, normalmente en:

- /usr/src/linux-xx-xx

II. Programación de E/S

Bloque I

- Aspectos básicos de la programación de E/S
- Arquitectura y programación de la E/S en el sistema operativo
- Módulos de núcleo en Linux
- Organización de manejadores de dispositivos (*drivers*)
- Gestión de memoria

Drivers de Linux

Se crean como un caso particular de los módulos

Los drivers utilizan números especiales para su identificación que tienen que ser registrados en la instalación y borrados en la desinstalación

La instalación de un driver no da acceso a ningún dispositivo concreto:

- el acceso se obtiene con la creación posterior de un *fichero de dispositivo*:
  - forma parte del sistema de ficheros
  - sobre él se podrán ejecutar las operaciones ofrecidas por el driver

Identificación de dispositivos y drivers

Los dispositivos se representan mediante nombres especiales en el sistema de ficheros (habitualmente en `/dev`)

Usan números de identificación

- *Número mayor* (*major*): identifica a un driver, para el acceso a una unidad funcional separada
- *Número menor* (*minor*):
  - identifica subunidades de un dispositivo mayor
  - o diferentes modos de uso del dispositivo

Drivers: se identifican con un nombre y un dato del tipo `dev_t`

- El fichero `/proc/devices` muestra los dispositivos mayores

El tipo `dev_t`

Los números de dispositivo se guardan en un dato del tipo `dev_t`, para el que hay dos macros:

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

- la primera nos devuelve el número mayor, y la segunda el menor

Se puede construir un valor del tipo `dev_t` con la macro:

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

Los números mayores se pueden asignar de forma estática o dinámica.

Asignación estática de números mayores

Si deseamos un número mayor concreto, podemos usar:

```
int register_chrdev_region
  (dev_t first,        //número mayor y primer menor
   unsigned int count,  //cuántos núm. menores
   char *name,          //nombre del dispositivo
```

- Reserva el número mayor indicado y unos cuantos números menores
- Si hay error, retorna un entero distinto de cero

Peligro de colisiones de número

Asignación dinámica de números mayores

La asignación de número mayor para un dispositivo de caracteres se hace en la `init_function` mediante la función definida en `<linux/fs.h>`:

```
int alloc_chrdev_region
  (dev_t *dev,          //retorna número mayor y menor
   unsigned int firstminor, // primer número menor
   unsigned int count,  //cuántos núm. menores
   char *name,          //nombre del dispositivo
```

- Reserva un número mayor para el driver y unos cuantos números menores
- Devuelve en `dev` el número mayor y el primer número menor
- Si hay error, retorna un entero distinto de cero

Creación del fichero de dispositivo

El nombre del dispositivo aparecerá en `/proc/devices` y `sysfs`

Luego creamos el fichero especial de dispositivo

- habitualmente en `/dev`
- se crean mediante la utilidad `mknod`

Por ejemplo, así se crea un fichero para números mayor y menor 123 y 0, respectivamente

```
mknod /dev/nombre_disp c 123 0
```

Puede ser necesario cambiar los permisos de acceso, para el uso por usuarios normales

Desinstalación del driver

Se hace desde la `cleanup_function`:

```
void unregister_chrdev_region
  (dev_t first,
   unsigned int count);
```

- `First` indica el número mayor y el primer número menor
- `Count` indica cuántos números menores hay que desinstalar

Programación de drivers de entrada/ salida

El driver es un módulo software que debe responder a una interfaz bien definida con el kernel, con determinados *puntos de entrada* (funciones C):

`open`	abrir el dispositivo para usarlo
`release`	cerrar el dispositivo
`read`	leer bytes del dispositivo
`write`	escribir bytes en el dispositivo
`ioctl`	orden de control sobre el dispositivo

Las que no se usen no se implementan

Programación de drivers de entrada/ salida (cont.)

Otros puntos de entrada menos frecuentes:

`llseek`	posicionar el puntero de lectura/escritura
`aio_read`	lectura asíncrona (en paralelo al proceso)
`aio_write`	escritura asíncrona
`readdir`	leer información de directorio (no se usa en dispositivos)
`poll`	preguntar si se puede leer o escribir
`mmap`	mapear el dispositivo en memoria
`flush`	volcar los buffers al dispositivo
`fsync`	sincronizar el dispositivo
`aio_fsync`	sincronizar el dispositivo en paralelo al proceso
`fasync`	notificación de operación asíncrona

Programación de drivers de entrada/ salida (cont.)

Otros puntos de entrada menos frecuentes:

`lock`	reservar el dispositivo
`readv`	leer sobre múltiples áreas de memoria
`writev`	escribir sobre múltiples áreas de memoria
`sendfile`	enviar un fichero a un dispositivo
`sendpage`	enviar datos, página a página
`get_unmapped_area`	obtener zona de memoria
`check_flags`	comprobar las opciones pasadas a fcntl()
`dir_notify`	fcntl() requiere aviso de modificaciones a directorio

Interfaces de los puntos de entrada más frecuentes

Convenio de nombres: anteponer el nombre del dispositivo al del punto de entrada; por ejemplo, para el dispositivo `ptr`

```
int ptr_open (struct inode *inode, struct file *filp)

void ptr_release (struct inode *inode, struct file *filp)

ssize_t ptr_read (struct file *filp, char __user *buff,
                  size_t count, loff_t *offp)

ssize_t ptr_write (struct file *filp, const char __user *buff,
                   size_t count, loff_t *offp)

int ptr_ioctl (struct inode *inode, struct file *filp,
               unsigned int cmd, unsigned long arg)
```

La notación `__user` indica que el dato no puede usarse directamente desde el driver; es ignorada por el compilador

Estructuras básicas de los drivers

Definidas en `<linux/fs.h>`:

- **`struct file_operations`**
  - contiene una tabla de punteros a las funciones del driver
- **`struct file`**
  - sirve para representar en el kernel un fichero abierto
- **`struct inode`**
  - estructura con la que el kernel representa internamente los ficheros

Tabla de punteros

```
struct file_operations {
    struct module *owner;
    ... // punteros a puntos de entrada
}
```

El campo `owner` es un puntero al módulo propietario del driver

- Se suele poner al valor **`THIS_MODULE`**

Tabla de punteros (cont.)

Los restantes campos de esta estructura son punteros a las funciones del driver, en este orden:

`llseek`	`mmap`	`readv`
`read`	`open`	`writev`
`aio_read`	`flush`	`sendfile`
`write`	`release`	`sendpage`
`aio_write`	`fsync`	`get_unmapped_area`
`readdir`	`aio_fsync`	`check_flags`
`poll`	`fasync`	`dir_notify`
`ioctl`	`lock`	

Estructura de fichero

La información relevante para los driver es la siguiente:

```
struct file {
  mode_t f_mode;
  loff_t f_pos;
  unsigned int f_flags;
  struct file_operations *f_op;
  void *private_data;
  struct dentry *f_dentry;   ...
}
```

- La crea el kernel en la operación `open` (puede haber muchas para el mismo dispositivo)
- Se le pasa a cualquier función que opera con el dispositivo

Estructura de fichero (cont.)

La descripción de los campos de `struct file` es:

- `f_mode`: Modo de lectura (`FMODE_READ`), escritura (`FMODE_WRITE`) o ambos
- `f_pos`: posición actual para lectura o escritura
  - valor de 64 bits, `long long`
  - se puede leer pero no se debería modificar
  - si `read` o `write` necesitan actualizar la posición deberían usar el parámetro que se les pasa
- `f_flags`: Opciones del fichero usadas al abrir
  - Ej.: `O_RDONLY`, `O_NONBLOCK`, `O_SYNC`
  - los permisos de lectura escritura se deberían chequear en `f_mode`

Estructura de fichero (cont.)

- `f_op`: tabla de punteros a los puntos de entrada del driver (ver página 42)
  - esta tabla se podría cambiar completamente en el `open` dependiendo por ejemplo del número menor para identificar un modo de funcionamiento
- `private_data`: puntero para guardar información específica del driver
  - al principio vale `null`; el `open` debe asignarle memoria
  - el `release` debe liberar esa memoria
- `f_dentry`: datos internos del sistema operativo
  - el punto más importantede esta estructura es que se puede acceder al inodo
    `filp->f_dentry->d_inode`

Estructura de inodos

Contiene información útil para los ficheros, tal como el estado del fichero, hora de creación y modificación, etc., pero la información importante para los drivers es:

```
struct inode {
    dev_t i_rdev;
    struct cdev *i_cdev;
    ...
}
```

- `i_rdev`: identificador del dispositivo
- `i_cdev`: apunta a la estructura interna del kernel para dispositivos de caracteres

Estructura de inodos (cont.)

Es única para el dispositivo

Podemos usar estas macros para averiguar los números mayor y menor:

```
unsigned int imajor(struct inode *inode)
unsigned int iminor(struct inode *inode)
```

Registro del driver de caracteres

Antes de usar el driver, es preciso alojar en memoria, inicializar y registrar la estructura `cdev`

- con las funciones de `<linux/cdev.h>`

Alojar:

```
struct cdev *cdev_alloc(void);
```

Inicializar

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

Además, hay que inicializar el campo `owner`:

```
cdev->owner=THIS_MODULE;
```


Registro del driver de caracteres (cont.)

Añadir el driver al kernel

```
int cdev_add(struct cdev *dev,
             dev_t num,            // id. dispositivo
             unsigned int count); // cuántos disp.
```

- si falla, retorna un número distinto de cero, y no se añade el dispositivo al sistema
- si va bien, el dispositivo se dice que está *vivo*

Eliminarlo

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

Punto de entrada *open*

Interfaz:

```
int ptr_open (struct inode *inodep,
              struct file *filp)
```

Abre el dispositivo:

- Chequea errores del dispositivo
- Inicializa el dispositivo si es la primera vez que se abre
- Crea y rellena si hace falta `filp->private_data`
- Identifica el número menor y actualiza `f_op` si procede

Punto de entrada *release*

Interfaz:

```
void ptr_release (struct inode *inodep,
                  struct file *filp)
```

Cierra el dispositivo:

- Libera si hace falta la memoria asignada a `filp->private_data`
- Cierra el dispositivo si es preciso

Punto de entrada *read*

Interfaz:

```
ssize_t ptr_read (struct file *filp,
                  char __user *buff,
                  size_t count, loff_t *offp)
```

Lee datos del dispositivo:

- `count` es la cantidad de bytes a leer
- `buff` es el buffer de usuario en el que se depositan los datos
  - está en el espacio del usuario
- `offp` la posición del fichero a la que se accede
- retorna el número de bytes leídos o un valor negativo si hay algún error

Punto de entrada *write*

Interfaz:

```
ssize_t ptr_write (struct file *filp,
                   const char __user *buff,
                   size_t count, loff_t *offp)
```

Escribe datos del dispositivo:

- `count` es la cantidad de bytes a escribir
- `buff` es el buffer de usuario en el que están los datos
  - está en el espacio del usuario
- `offp` la posición del fichero a la que se accede
- retorna el número de bytes escritos o un valor negativo si hay algún error

Punto de entrada *ioctl*

Interfaz:

```
int ptr_ioctl (struct inode *inodep,
               struct file *filp,
               unsigned int cmd, unsigned long arg)
```

Envía una orden de control al dispositivo:

- `cmd` es la orden
- `arg` es un argumento para ejecutar la orden

La interpretación de la orden es dependiente del dispositivo

Retorna cero si va bien, o un código de error si va mal.

Ejemplo: Driver de prueba

```
//  basico.h

int basico_open(struct inode *inodep, struct file *filp);

int basico_release(struct inode *inodep, struct file *filp);

ssize_t basico_read (struct file *filp, char *buff,
                     size_t count, loff_t *offp);

ssize_t basico_write (struct file *filp, const char *buff,
          size_t count, loff_t *offp);
```

Driver de prueba: includes

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include "basico.h"

MODULE_LICENSE("GPL");
```


Driver de prueba: variables globales

```
static struct file_operations basico_fops = {
  THIS_MODULE, // owner
  NULL,        // lseek
  NULL,        // read
  NULL,        // aio_read
  NULL,        // write
  NULL,        // aio_write
  NULL,        // readdir
  NULL,        // poll
  NULL,        // ioctl
  NULL,        // mmap
  NULL,        // open
  NULL,        // flush
  NULL,        // release
  NULL,        // fsync
  // lo mismo para todos los demás puntos de entrada
  NULL         // dir_notify
};
```

Driver de prueba: variables globales (cont.)

```
// Datos del dispositivo, incluyendo la estructura cdev

struct basico_datos {
  struct cdev *cdev;  // Estructura para dispositivos de caracteres
  dev_t dev;          // información con el numero mayor y menor
  int dato_cualquiera;
};

static struct basico_datos datos;
```


Driver de prueba: instalación y desinstalación del driver

```
static int modulo_instalacion(void) {
  int result;

  // ponemos los puntos de entrada
  basico_fops.open=basico_open;
  basico_fops.release=basico_release;
  basico_fops.write=basico_write;
  basico_fops.read=basico_read;

  // reserva dinamica del número mayor del módulo
  // numero menor = cero, numero de dispositivos =1
  result=alloc_chrdev_region(&datos.dev,0,1,"basico");
  if (result < 0) {
    printk(KERN_WARNING "basico> (init_mod) fallo con major %d\n",
       MAJOR(datos.dev));
    return result;
  }
```

Driver de prueba: instalación y desinstalación del driver (cont.)

```
  // iniciamos datos
  datos.dato_cualquiera=33; // por ejemplo
  datos.cdev=cdev_alloc();
  cdev_init(datos.cdev, &basico_fops);
  datos.cdev->owner = THIS_MODULE;
  result= cdev_add(datos.cdev, datos.dev, 1);
  if (result!=0) {
    printk(KERN_WARNING "basico> (init_module) Error%d al añadir",
           result);
    // deshacer la reserva y salir
    unregister_chrdev_region(datos.dev,1);
    return result;
  }

  // todo correcto: mensaje y salimos
  printk(KERN_INFO "basico> (init_module) OK con major %d\n",
         MAJOR(datos.dev));
  return 0;
}
```


Driver de prueba: instalación y desinstalación del driver

```
static void modulo_salida(void) {
  cdev_del(datos.cdev);
  unregister_chrdev_region(datos.dev,1);
  printk( KERN_INFO "basico> (cleanup_module) descargado OK\n");
}


module_init(modulo_instalacion);
module_exit(modulo_salida);
```

Driver de prueba: puntos de entrada

```
int basico_open(struct inode *inodep, struct file *filp) {
  int menor= iminor(inodep);
  printk(KERN_INFO "basico> (open) menor= %d\n",menor);
  return 0;
}


int basico_release(struct inode *inodep, struct file *filp) {
  int menor= iminor(inodep);
  printk(KERN_INFO "basico> (release) menor= %d\n",menor);
  return 0;
}
```


Driver de prueba: puntos de entrada (cont.)

```
ssize_t basico_read (struct file *filp, char *buff,
         size_t count, loff_t *offp)
{
  printk(KERN_INFO "basico> (read) count=%d\n",(int) count);
  return (ssize_t) 0;
}


ssize_t basico_write (struct file *filp, const char *buff,
          size_t count, loff_t *offp)
{
  printk(KERN_INFO "basico> (write) count=%d\n",(int) count);
  return (ssize_t) count;
}
```

Programa de prueba para el driver

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

int main() {
  int fd,err;
  char *mistring="Hola yo soy un mensaje";
  char strleido[100];
  ssize_t leidos, escritos;
  strleido[0]=0; // lo hago vacío

  fd=open("/dev/basico0", O_RDWR);
  if (fd==-1) {
    perror("Error al abrir el fichero"); exit(1);
  }
```


Programa de prueba (cont.)

```
  printf("Fichero abierto\n");

  escritos=write(fd,mistring,strlen(mistring)+1);
  printf("Bytes escritos: %d\n",escritos);

  leidos=read(fd,&strleido,100);
  strleido[leidos]='\0';
  printf("Bytes leidos: %d\n",leidos);
  printf("String leido: %s\n",strleido);

  err=close(fd);
  if (err==-1) {
    perror("Error al cerrar el fichero");
    exit(1);
  }
  printf("Fichero cerrado\n");

  exit(0);
}
```

II. Programación de E/S

Bloque I

- Aspectos básicos de la programación de E/S
- Arquitectura y programación de la E/S en el sistema operativo
- Módulos de núcleo en Linux
- Organización de manejadores de dispositivos (*drivers*)
- Gestión de memoria

Gestión de memoria

Reserva de memoria para variables dinámicas: en `<linux/slab.h>`

- dentro del kernel debe usarse `kmalloc` y `kfree`:

```
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

- `kmalloc` reserva un número de bytes >= `size` y nos retorna su dirección; `flags` controla el comportamiento de la función
- `kfree` libera la memoria reservada con `kmalloc`

El tamaño máximo que puede ser asignado por `kmalloc` está limitado (siempre menor que 128KB)

- existen otras funciones para conseguir más memoria como `vmalloc`

Flags de asignación de memoria

Tradicionalmente se han llamado prioridades de asignación y se definen en `<linux/gfp.h>`:

- `GPF_KERNEL`: La función que hace esta asignación debe ser reentrante, y es la que se usa normalmente
- `GFP_BUFFER`: asigna memoria de buffers intermedios y previene deadlocks si los propios subsistemas de I/O necesitan memoria
- `GFP_ATOMIC`: usada para asignar memoria en los manejadores de interrupción
- `GFP_USER`: usada para asignar memoria en la parte de usuario
- `__GFP_DMA`: memoria para transferencias DMA

Copia de datos del espacio del usuario al del kernel

El espacio de direcciones del kernel es diferente al de los procesos (espacio de usuario)

Las funciones *read* y *write* necesitan copiar datos respectivamente hacia y desde el espacio de usuario

Las funciones para estas copias están en `<asm/uaccess.h>`:

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

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

- retornan el número de bytes no copiados
- o un error, si el puntero de usuario no es válido por ejemplo

Ejemplo: buffer virtual

```
//  buffervirtual.h

#define MAX 100

int buffervirtual_open(struct inode *inodep, struct file *filp);

int buffervirtual_release(struct inode *inodep,
                          struct file *filp);

ssize_t buffervirtual_read (struct file *filp, char *buff,
                            size_t count, loff_t *offp);

ssize_t buffervirtual_write (struct file *filp, const char *buff,
                             size_t count, loff_t *offp);
```


Buffer virtual: includes

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include "buffervirtual.h"
#include <linux/slab.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");
```

Buffer virtual: variables globales

```
static struct file_operations buffervirtual_fops = {
  THIS_MODULE, // owner
  NULL,        // lseek
  NULL,        // read
  NULL,        // aio_read
  NULL,        // write
  NULL,        // aio_write
  NULL,        // readdir
  NULL,        // poll
  NULL,        // ioctl
  NULL,        // mmap
  NULL,        // open
  NULL,        // flush
  NULL,        // release
  NULL,        // fsync
  // lo mismo para todos los demás puntos de entrada
  NULL         // dir_notify
};
```


Buffer virtual: variables globales

```
// Datos del dispositivo, incluyendo la estructura cdev

struct bv_datos {
  struct cdev *cdev; // Estructura de dispositivos de caracteres
  dev_t dev;         // información con el numero mayor y menor
  char *buffer;      // este sera nuestro buffer
};

static struct bv_datos datos;
```

Buffer virtual: instalación y desinstalación del driver

```
static int modulo_instalacion(void) {
  int result;

  // ponemos los puntos de entrada
  buffervirtual_fops.open=buffervirtual_open;
  buffervirtual_fops.release=buffervirtual_release;
  buffervirtual_fops.write=buffervirtual_write;
  buffervirtual_fops.read=buffervirtual_read;

  // reserva dinamica del número mayor del módulo
  // numero menor = cero, numero de dispositivos =1
  result=alloc_chrdev_region(&datos.dev,0,1,"buffervirtual");
  if (result < 0) {
    printk(KERN_WARNING "bv> (init_module) fallo con major %d\n",
           MAJOR(datos.dev));
    return result;
  }
```


Buffer virtual: instalación y desinstalación del driver (cont.)

```
  // reservamos MAX bytes de espacio para el buffer
  datos.buffer = kmalloc(MAX, GFP_KERNEL);
  if (datos.buffer==NULL) {
    result = -ENOMEM;
    unregister_chrdev_region(datos.dev,1);
    printk( KERN_WARNING "bv> (init_module) Error, no hay mem.\n");
    return result;
  }

  // instalamos driver
  datos.cdev=cdev_alloc();
  cdev_init(datos.cdev, &buffervirtual_fops);
  datos.cdev->owner = THIS_MODULE;
  result= cdev_add(datos.cdev, datos.dev, 1);
  if (result!=0) {
    printk(KERN_WARNING "bv> (init_module) Error %d al añadir",
           result);
    // deshacer la reserva y salir
    kfree(datos.buffer);
```

Buffer virtual: instalación y desinstalación del driver (cont.)

```
    unregister_chrdev_region(datos.dev,1);
    return result;
  }

  // todo correcto: mensaje y salimos
  printk(KERN_INFO "bv> (init_module) OK con major %d\n",
         MAJOR(datos.dev));
  return 0;
}

static void modulo_salida(void) {
  kfree(datos.buffer);
  cdev_del(datos.cdev);
  unregister_chrdev_region(datos.dev,1);
  printk( KERN_INFO "bv> (cleanup_module) descargado OK\n");
}

module_init(modulo_instalacion);
module_exit(modulo_salida);
```


Buffer virtual: puntos de entrada del driver (cont.)

```
int buffervirtual_open(struct inode *inodep, struct file *filp) {
  int menor= iminor(inodep);
  printk(KERN_INFO "bv> (open) menor= %d\n",menor);
  return 0;
}


int buffervirtual_release(struct inode *inodep, struct file *filp) {
  int menor= iminor(inodep);
  printk(KERN_INFO "bv> (release) menor= %d\n",menor);
  return 0;
}
```

Buffer virtual: puntos de entrada del driver (cont.)

```
ssize_t buffervirtual_read (struct file *filp, char *buff,
                            size_t count, loff_t *offp)
{
  ssize_t cuenta;
  unsigned long not_copied;

  printk(KERN_INFO "bv> (read) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }
  not_copied=copy_to_user(buff,datos.buffer,
                          (unsigned long)cuenta);
  if (not_copied!=0) {
    printk(KERN_WARNING "bv> (read) AVISO, no se leyeron datos\n");
    return(-EFAULT);
  }
  return cuenta;
}
```


Buffer virtual: puntos de entrada del driver (cont.)

```
ssize_t buffervirtual_write (struct file *filp, const char *buff,
                             size_t count, loff_t *offp)
{
  ssize_t cuenta;
  unsigned long not_copied;

  printk(KERN_INFO "bv> (write) count=%d\n",(int) count);
  cuenta=(ssize_t) count;
  if (cuenta>MAX) {
    cuenta=MAX;
  }
  not_copied=copy_from_user(datos.buffer,buff,
                            (unsigned long)cuenta);
  if (not_copied!=0) {
    printk(KERN_WARNING "bv> (write) AVISO, no escribió bien\n");
    return(-EFAULT);
  }
  return cuenta;
}
```

Paso de parámetros en la instalación de un módulo

Se le pueden pasar parámetros al `insmod` con las macros definidas en `<linux/moduleparam.h>`

El paso de tipos enteros se realiza con la macro `module_param()`:

- Ejemplo de uso

```
int param = 5;
...
module_param(param, int, 0)
```

  - los tres parámetros son: la variable, el tipo y los permisos
- En la instalación el parámetro se pasaría:

```
insmod buffervirtual.ko param=20
```

También se pueden pasar arrays y strings.

Bibliografía

[1] Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini, "Linux Device Drivers", 3rd Ed., O'Reilly, 2005.