Taller de Tiempo Real para Control Robótico

Mi Introducción A los Drivers

Dr. Nelson Acosta – nacosta@exa.unicen.edu.ar
INCA / INTIA-Facultad Ciencias Exactas-UNCPBA
Tandil - Argentina

(des)Organización del curso

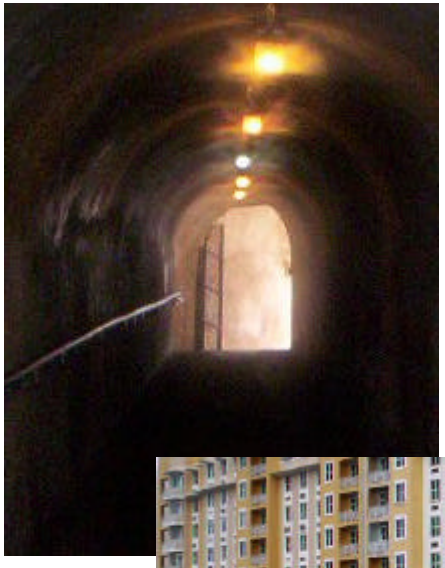
- **Drivers de Linux.**
  - Interfaz con el Sistema Operativo.
  - Dispositivos y módulos.
  - Medición del tiempo, Entrada/Salida e Interrupciones.
  - Mecanismos de transferencia de datos.
- **Linux de Tiempo Real.**
  - Características.
  - Llamadas al sistema.
  - Asignación dinámica.

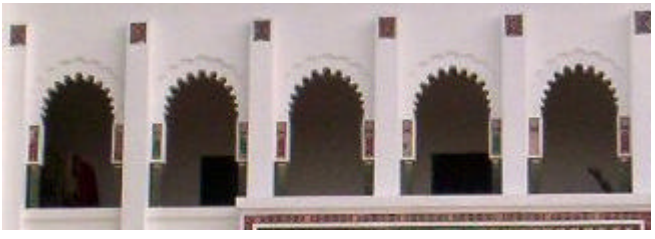
Tabla de Contenido

Tabla de Contenido

- **Background.**
  - Sistema de I/O. Selección de dispositivo. Mapa de I/O. Sincronización. Consulta de estado de un dispositivo. Interrupciones en el PC. Mecanismos de transferencia de datos. Organización.
- **El Linux.**
  - Características de los drivers. Carga y descarga de módulos. Configuración módulos. Init. Espacio del USR. Dispositivos de carácter. Major y Minor, asignación dinámica. Release. Open. Registrado de módulos. Close. Transferencia Usr-Kernel. Valor retorno lectoescritura. Read. Write.

Sistemas de I/O





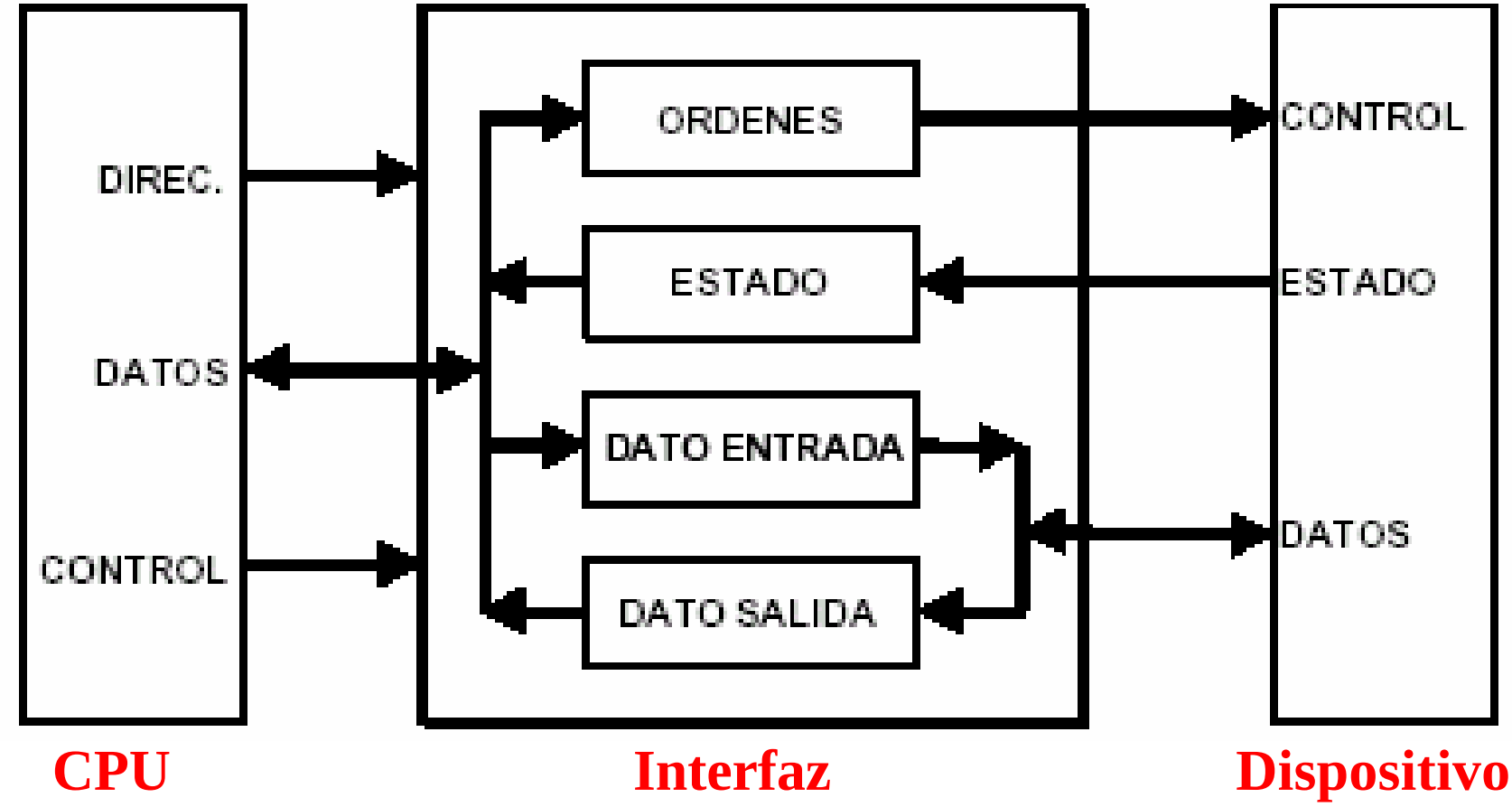
Funciones sistema de E/S

Direccionamiento → **selección de un dispositivo**

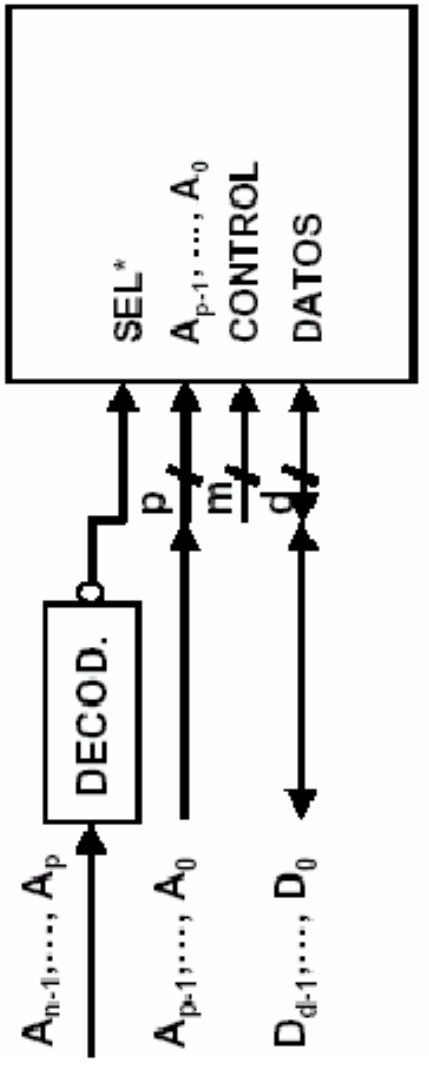
Sincronización → **inicio de la transferencia**

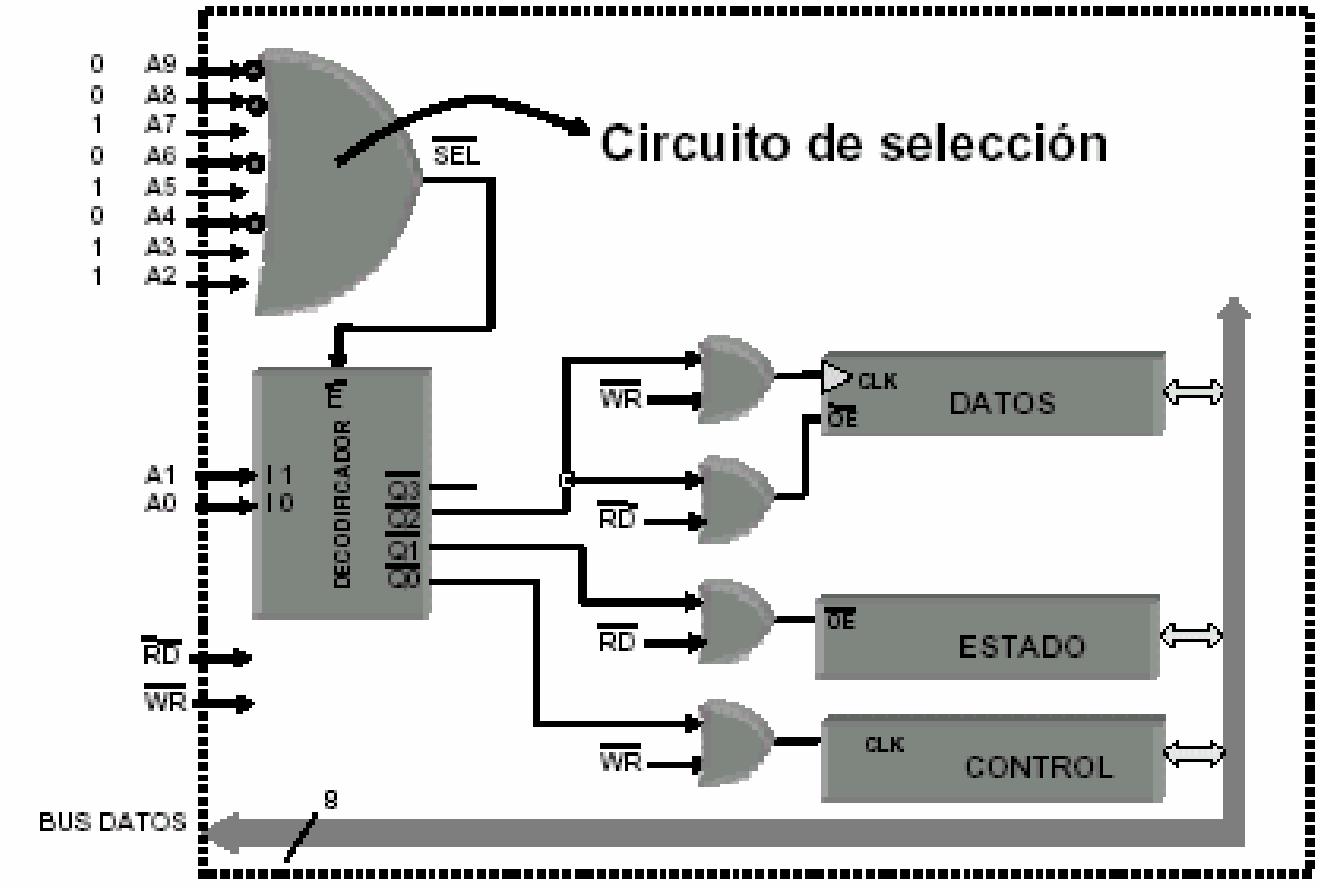
Transferencia → **método de transferencia**

Diagrama simplificado de E/S



Selección de un dispositivo





DIRECCIÓN BASE = 0010101100 (09Ch) REGISTRO DATOS
---------- 01 (09Dh) REGISTRO ESTADO
---------- 10 (09Eh) REGISTRO CONTROL

Mapa de **I/O** del **IBM/PC**

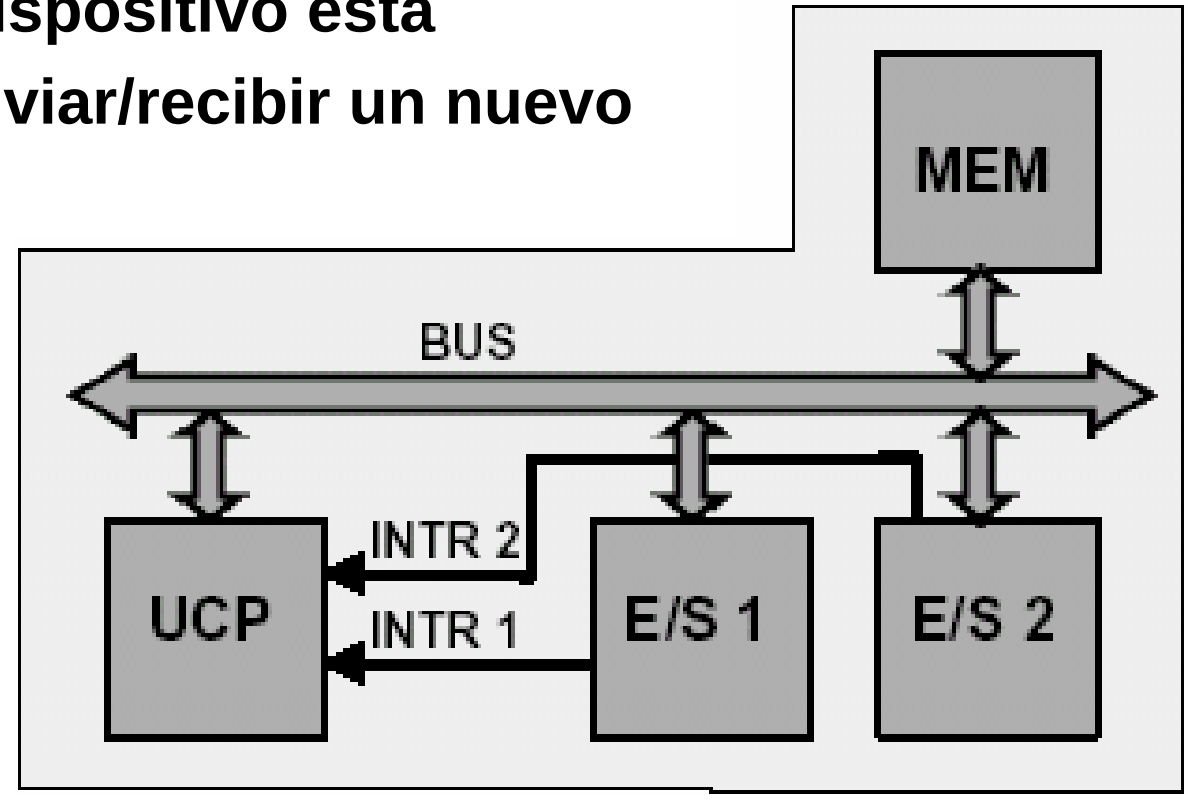
```
000-01F: Registros ADM1 (8237)
020-03F: Registro control INTR 1 (8259)
040-05F: Registro temporizador (8254)
060-06F: Registro control TECLADO (8042)
070-07F: Reloj tiempo real
080-08F: Registro página ADM
0A0-0BF: Registro control INTR 2 (8259)
0C0-0DF: Registros ADM2 (8237)
0F0-0FF: Registro Coprocesador matemático
1F0-1F8: Controlador disco duro
200-20F: Adaptador juegos
210-277: No usado
278-27F: Puerto paralelo 3
280-2F7: No usado
2F8-2FF: Puerto serie 2
300-31F: Placa prototipo USR
360-36F: Reservado
378-37F: Puerto paralelo 2
380-38F: Comunicación SDLC o Bisync 2
3A0-3AF: Comunicación Bisync 1
3B0-3BF: Placa MDA monocromo
3C0-3CF: Placa video color (EGA/VGA)
3D0-3DF: Placa video color (CGA/MCGA)
3F0-3F7: Control diskettes
3F8-3FF: Puerto serie 1
```

Sincronización. Consulta de estado

La CPU debe leer constantemente el registro de estado del la interfaz, para saber cuando el dispositivo esta preparado para enviar/recibir un nuevo dato.

polling

Buzy wait



Sincronización. Consulta de estado

Inconvenientes:

- Mientras la CPU esta leyendo el registro de estado de la interfaz, **no puede** realizar otras tareas de mayor interés.

Ventajas:

- Periférico solicita la petición sólo cuando está preparado. La CPU puede realizar otras tareas.
- El periférico avisa a la CPU de su disponibilidad mediante una línea de Petición de Interrupción (INTR, IRQ,..).

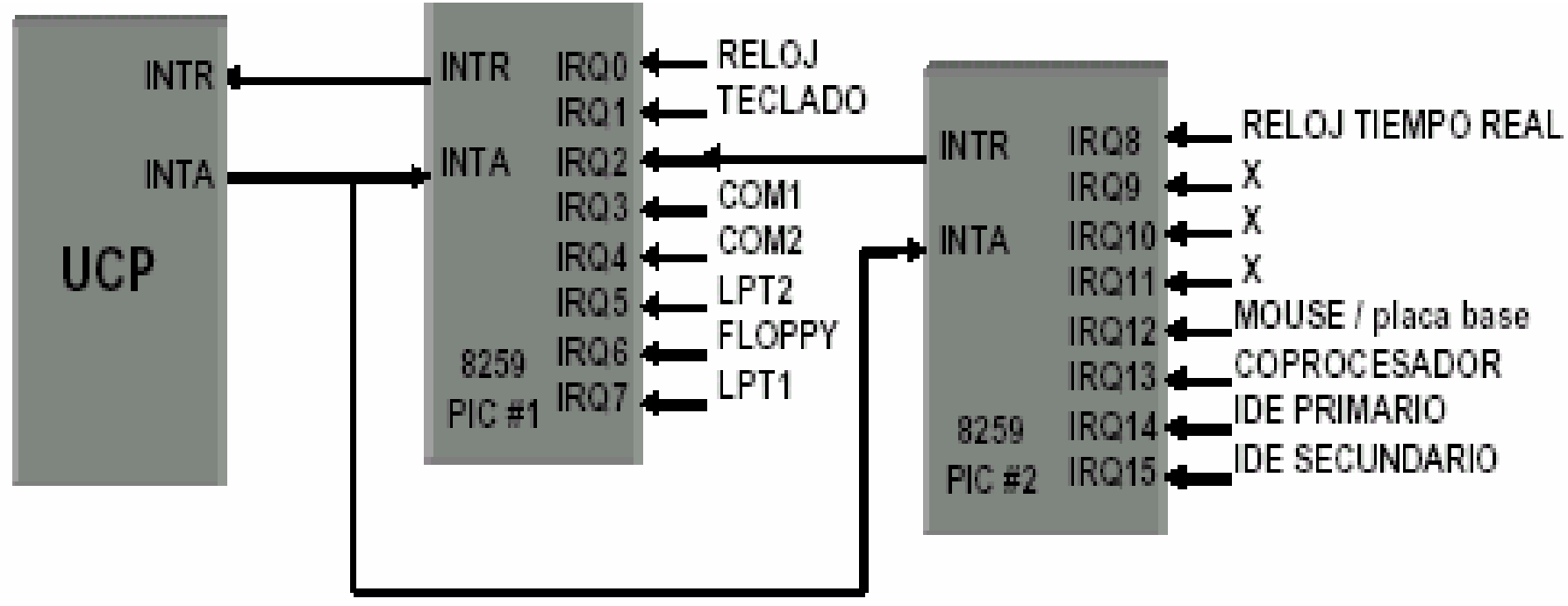
Sincronización. Consulta de estado

Según el evento que produjo la interrupción:

INTERRUPCIONES

- **HARDWARE**
  - **HARDWARE INTERNAS (CPU)**
  - **HARDWARE EXTERNAS (Dispositivos de I/O)**
- **SOFTWARE (Ejecución de instrucciones: INT n)**

Interrupciones en el PC:
diagrama ejemplo



Transferencia de Datos

- **Programada:** la CPU se encarga de llevar o traer datos escribiendo o leyendo los registros de datos de la interfaz del periférico.
- **Acceso directo a memoria:** Se usa un dispositivo auxiliar (Controlador de Acceso Directo a Memoria-DMA), que programado previamente por la CPU es el encargado de realizar la transferencia, dejando libre a la CPU para realizar otras tareas. Este mecanismo se utiliza con dispositivos periféricos que transfieren grandes cantidades de datos (discos, tarjetas de red, etc.).

Mecanismos de I/O

- **Instrucciones especiales.**
  - Incremento del conjunto de instrucciones.
  - Instrucciones tipo INTEL:

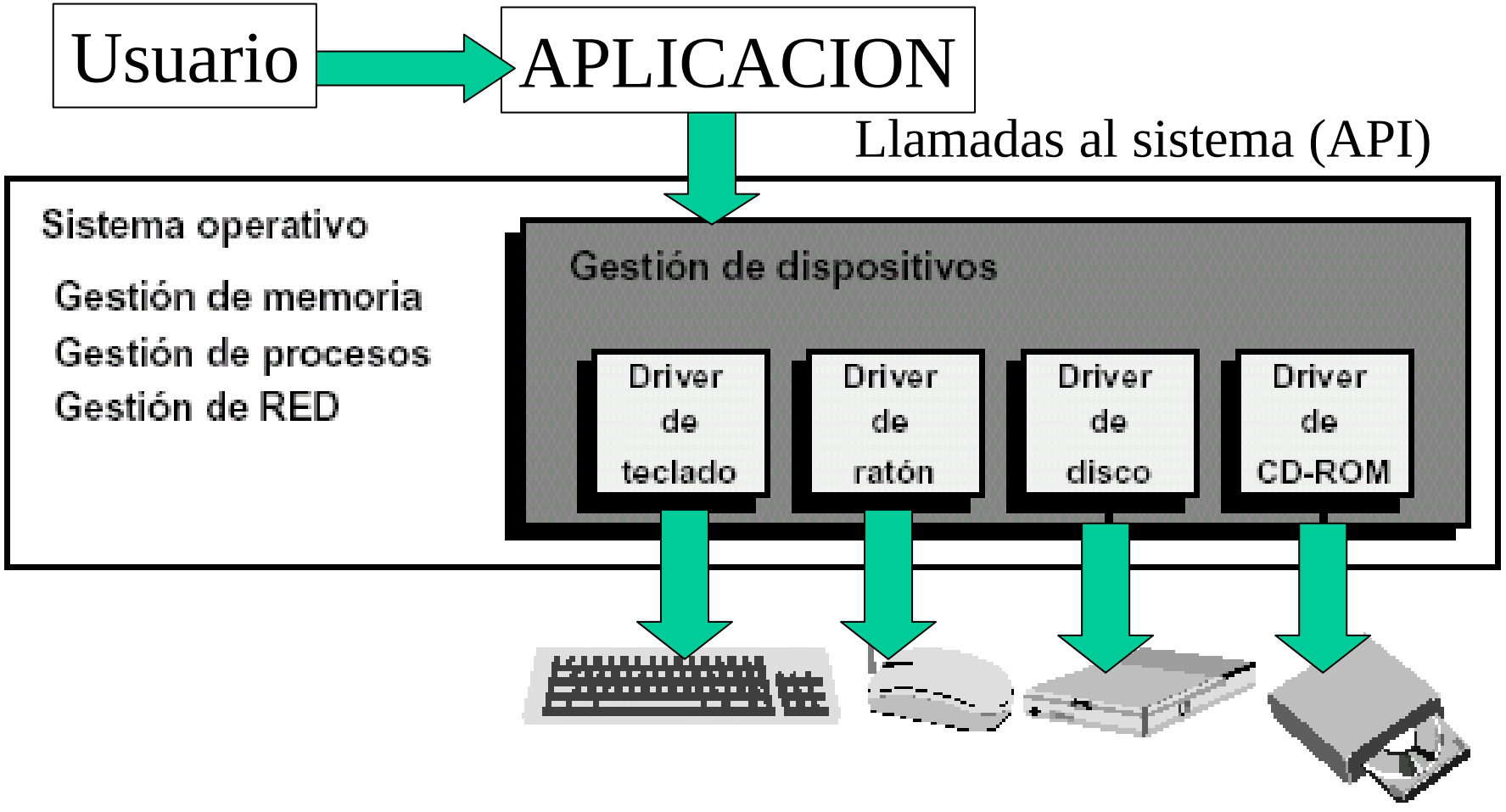
```
inb(int port); / inw(int port);
outb(char val, int port); / outw(char val, int port);
```

- **Memoria Mapeada.**
  - Redireccionamiento de accesos a memoria.
  - Simplicidad del diseño.
  - **Problema**: diferentes velocidades de acceso.

Usando los puertos de I/O

- **Puertos de 8 bits.**
  - unsigned inb(unsigned port);
  - void outb(unsigned char byte, unsigned port);
- **Puertos de 16 bits.**
  - unsigned inw(unsigned port);
  - void outw(unsigned short word, unsigned port);
- **Puertos de 32 bits.**
  - unsigned inl(unsigned port);
  - void outl(unsigned double word, unsigned port);

Organización: Diagrama

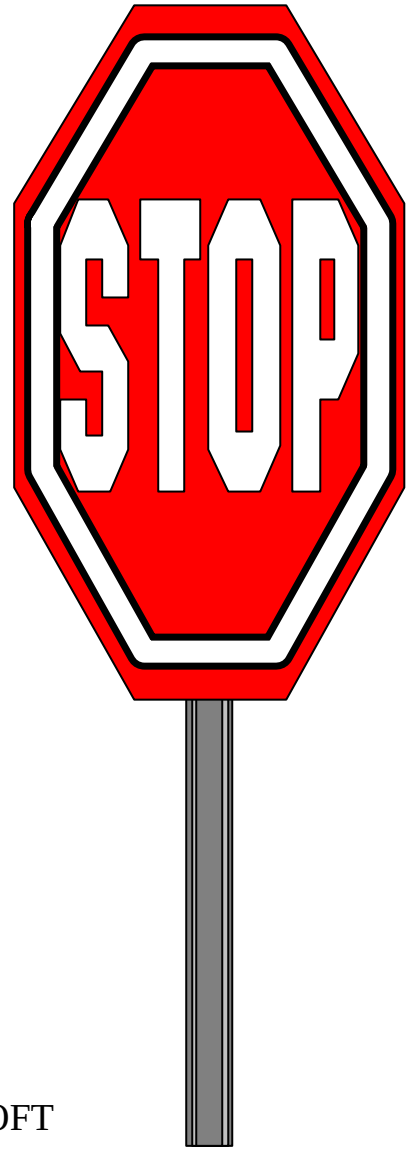


Linux … 1

VENTAJA: la carga dinámica de módulos.

- Permite al programador escribir partes del núcleo (drivers) como objetos separados y cargarlos (o descargarlos) mientras el sistema esta en ejecución.
- Un módulo es un archivo objeto (*.o) que contiene rutinas y datos que serán **enlazados** con el núcleo.
- Una vez cargado el código del módulo reside en el espacio de direcciones del núcleo y se ejecuta enteramente dentro del contexto del núcleo, es decir con los **máximos privilegios**.

Linux ... 2



- **Windows95:** un driver es un módulo conocido como virtual device driver (VxD).
- **Windows 98/2000:** nuevo modelo de drivers conocido como Windows Driver Model (WDM).
- Para implementar un driver en WINDOWS es necesario disponer de del paquete de MICROSOFT W98/2000 **Driver Development Kit** (DDK) y un entorno integrado de desarrollo.

Linux

DRIVERs

Concepto de Drivers:

Un driver es una colección de subrutinas y datos del kernel, que constituye la interfaz software con un dispositivo hardware de entrada/salida.

Tipos de Drivers:

- Drivers para dispositivos de **carácter**.
- Drivers para dispositivos de **bloques**.
- Drivers para dispositivos de **red**.
- Drivers para **pseudo-dispositivos**.

Características de los Drivers

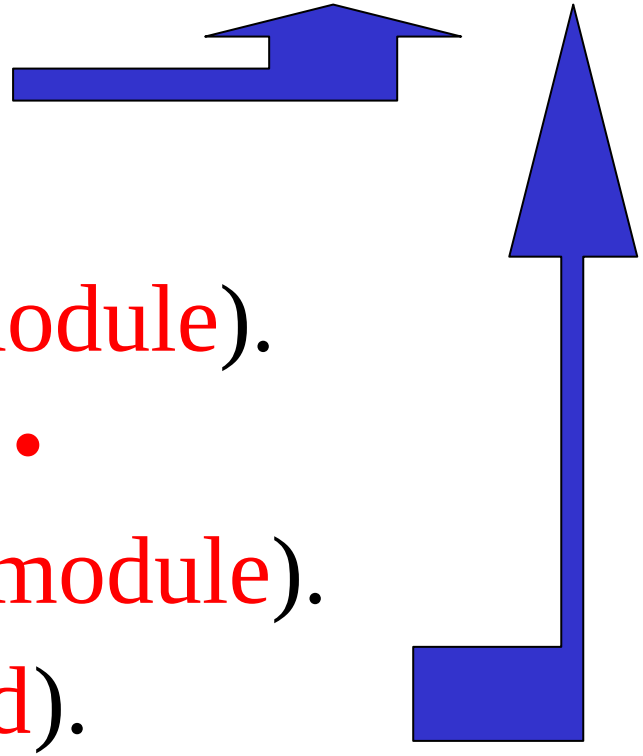
- Registrados como integrantes del sistema.
- Pertenecientes al **espacio** del kernel.
- Extienden la funcionalidad del sistema.
- Pueden emplear todos los recursos del sistema.
- Limitados por la rigidez del kernel.
- Desarrollo a nivel “**super**-usuario”.
- Permiten configuraciones específicas para cada kernel.
- Permanecen en **espera** hasta ser requeridos.

Carga/Descarga de Módulos

- **Escribir** el código del módulo (vi / emacs).
- **Compilar** el módulo (gcc).
- **Cargar** el módulo (insmod).
- **Inicializar** el módulo (init_module).

• • • • • • • • •

- **Cerrar** el módulo (cleanup_module).
- **Descargar** el módulo (rmmod).



Edición, compilación, (des)carga de módulos

Se debe compilar con las llaves adecuadas

hola.c

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

init_module (void) {
    printk("Hola MUNDO\n");
    return 0;
}

void cleanup_module (void) {
     printk("ADIOS mundo cruel\n");
}
```

```
oot3#  vi hola.c
root3#  gcc -c hola.c
root3#  insmod hola.o
Hola MUNDO
root3#  rmmod hola
ADIOS mundo cruel
root3#
```

Características de los módulos

- Permiten la configuración dinámica del sistema.
- Emplean únicamente funciones del sistema.
- Gran número de funciones y variables globales.
- Llamados desde el espacio del usuario.
- Ejecutados en el espacio del kernel.
- Tranferencia desde el espacio de usuario al del kernel mediante llamadas al sistema e interrupciones.
- Código ejecutado en forma continua.

Aspectos destacables de los módulos

- Relación entre **espacio** del kernel y del usuario.
- Concurrencias en el kernel (**sleep**).
- Dependencia con las versiones.
- Registrando tablas de símbolos.
- Manejador de errores.
- Contador en uso.
- Uso de recursos del sistema (*memoria*, *puertos*, *interrupciones*).
- Autodetección y configuración manual.

Funciones básicas de los módulos

- **Init_module**. Se ejecuta al instalarse en el kernel, es donde el diseñador solicita los recursos para la ejecución a Linux.

- **Cleanup_module**. Se ejecuta al desinstalarse del kernel, es donde el diseñador devuelve a Linux los recursos que solicitó para la ejecución.

Funcionamiento: Init_Module

- **Registrar las capacidades del módulo**
  - Determinar las funciones del módulo accesibles a cualquier aplicación por medio de llamadas a funciones del **kernel**.
  - Se entregan como argumento un puntero a una estructura de datos compuesta por punteros al código de las funciones del módulo.
- **Contador de uso**
  - Incrementar el contador (MOD_INC_USE_COUNT).
  - Decrementar el contador (MOD_DEC_USE_COUNT).
  - Evaluar si el contenido es nulo (MOD_IN_USE).

Funcionamiento: Cleanup_Module

- Llamar a **sys_delete_module**.
- Chequear **MOD_IN_USE**.
- Llamar a **cleanup_module** si el contador en uso tiene contenido nulo.
- Eliminación manual del **registrado** de las capacidades del módulo (*estructura de datos*).
- **Eliminación** automática de la tabla de símbolos.

Configuración del módulo

MEDIANTE:

- Parámetros especificados por el **usuario** en forma explícita en ***tiempo de carga***.
- Parámetros obtenidos por **autodetección** en tiempo de ***inicialización***.
- Parámetros obtenidos por **autodetección** con posibilidad de ***modificación manual***.

Trabajo en el espacio del usuario

- **VENTAJAS**
  - Empleo de **librerías externas**.
  - Mejora en las técnicas de **depuración**.
  - Mayor control sobre los posibles **errores de ejecución**.
  - **Asignación dinámica** de la memoria (reutilizable).
  - **Concurrencia** en el acceso a los dispositivos.
- **DESVENTAJAS**
  - **Sin Interrupciones** (no permitidas).
  - Acceso directo a memoria y puertos de I/O **restringido** (superusuario).
  - Tiempos de respuesta **lentos** (cambios de contexto).
  - Necesidad de **bloquear páginas de memoria** (lentitud).

Drivers para dispositivos de caracter

CARACTERISTICAS:

- **Comunicación directa** con programas de usuario.
- Accesibles a través de **archivos especiales** identificados unívocamente (minor, major).
- Se realiza la **transferencia** de datos entre bloques de memoria y dispositivos a través de estos archivos.
- Llevan a cabo la **transferencia de datos**.
- Realizan la transferencia de datos sin empleo de buffers intermedios.

Características de los major y minor

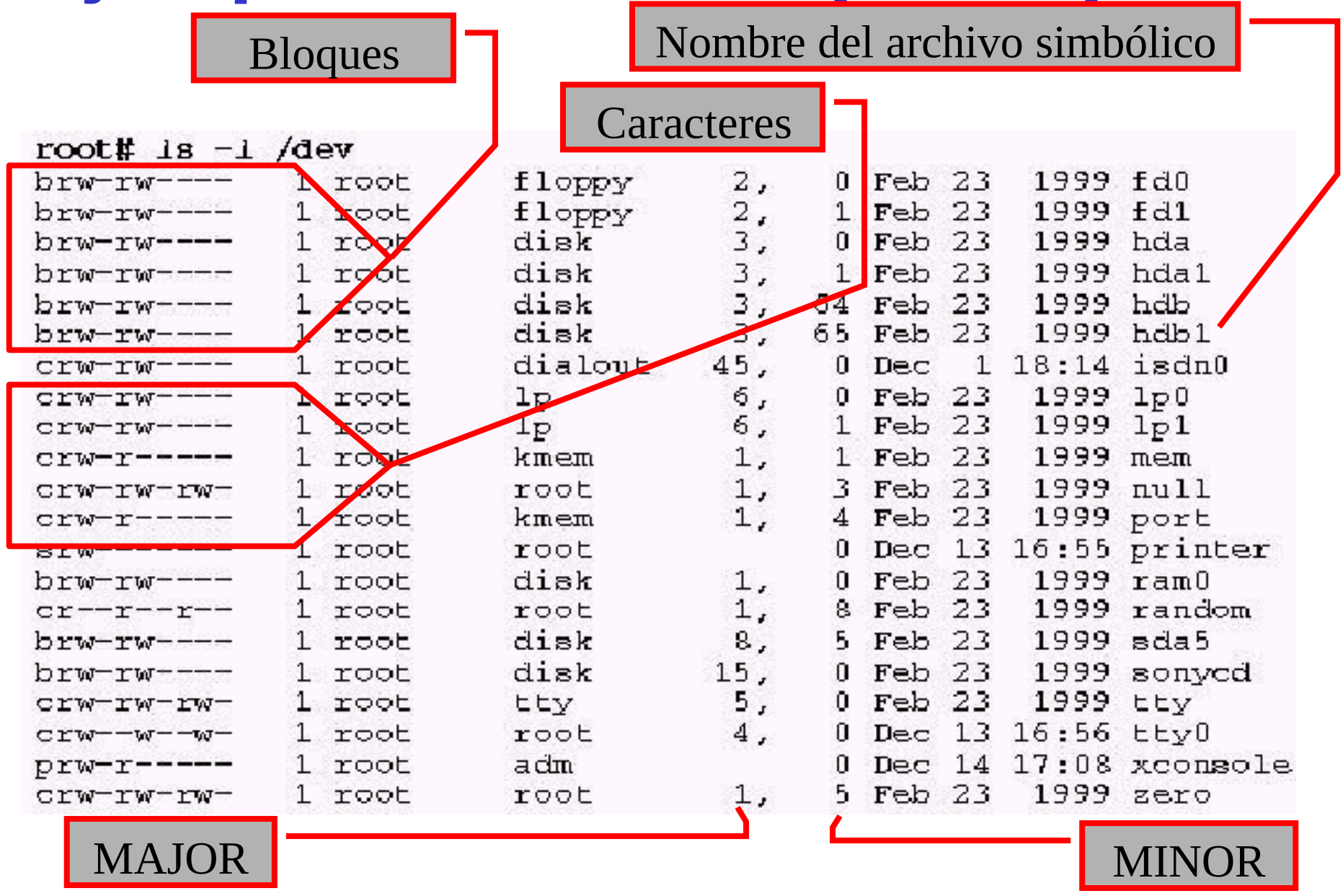
MAJOR

- Utilizado **por el kernel**.
- Permite **asociar** a cada dispositivo su driver apropiado.
- Posibilita el **registro del driver**.
- Asignado en la inicialización del driver.
- Empleado como **índice en un arreglo estático de drivers**.

MINOR

- Utilizado **por el driver**.
- Usual en drivers que controlan **varios dispositivos**.
- Permite diferenciar entre dispositivos con igual MAJOR.
- Determina el comportamiento del dispositivo.

Ejemplos de drivers (/dev/)




```
root# ls -l /dev
brw-rw----   1 root     floppy      2,   0 Feb 23  1999 fd0
brw-rw----   1 root     floppy      2,   1 Feb 23  1999 fd1
brw-rw----   1 root     disk        3,   0 Feb 23  1999 hda
brw-rw----   1 root     disk        3,   1 Feb 23  1999 hda1
brw-rw----   1 root     disk        3,  64 Feb 23  1999 hdb
brw-rw----   1 root     disk        3,  65 Feb 23  1999 hdb1
crw-rw----   1 root     dialout    45,   0 Dec  1 18:14 isdn0
crw-rw----   1 root     lp          6,   0 Feb 23  1999 lp0
crw-rw----   1 root     lp          6,   1 Feb 23  1999 lp1
crw-r-----   1 root     kmem        1,   1 Feb 23  1999 mem
crw-rw-rw-   1 root     root        1,   3 Feb 23  1999 null
crw-r-----   1 root     kmem        1,   4 Feb 23  1999 port
srw-------   1 root     root             0 Dec 13 16:55 printer
brw-rw----   1 root     disk        1,   0 Feb 23  1999 ram0
cr--r--r--   1 root     root        1,   8 Feb 23  1999 random
brw-rw----   1 root     disk        8,   5 Feb 23  1999 sda5
brw-rw----   1 root     disk       15,   0 Feb 23  1999 sonycd
crw-rw-rw-   1 root     tty         5,   0 Feb 23  1999 tty
crw--w--w-   1 root     root        4,   0 Dec 13 16:56 tty0
prw-r-----   1 root     adm              0 Dec 14 17:08 xconsole
crw-rw-rw-   1 root     root        1,   5 Feb 23  1999 zero
```

Registro de un driver de caracter

- Para que el sistema operativo sepa del driver, este debe registrarse.
- Se registra a través de **dos** llamadas al sistema.
- En el registro se vincula el **File-System** con el archivo simbólico, a través del **MAJOR** y la estructura **FOPS**.
- Con el registro se actualiza el acceso en *"/dev*".

Registro de un driver de caracter

```
int register_chrdev(unsigned int major,
      const char *name, struct file_operations
                                          *fops);
int unregister_chrdev(unsigned int major,
                                const char *name);
```

- Si MAJOR >= que MAX_C**G**RDEV entonces retorna el código de error -EINVAL.
- Si MAJOR corresponde con un número ya asignado entonces retorna el código de error -EBUSY.
- Si la operación se realizó correctamente, entonces se retorna el valor 0.

Driver: uso de uno genérico (1)

- La CPU no es el único dispositivo **inteligente** conectado a un sistema, cada dispositivo físico o periférico posee un controlador hardware.
- Teclado, mouse, puertos paralelos y puertos serie son controlados por el circuito integrado conocido como "**SuperIO** chip".
- Discos IDE por un controlador IDE.
- Dispositivos SCSI por un controlador SCSI.

Driver: uso de uno genérico (2)

- Cada controlador hardware dispone de
  - registros de **control**,
  - registros de **estado** y
  - registros de **datos**,
- Utilizados para inicializar el dispositivo, hacer diagnósticos, intercambiar datos, etc.
- No se pone el código para manejar un dispositivo dentro de cada aplicación que lo necesite, dicho código forma parte del núcleo de sistema operativo.

Driver: uso de uno genérico (3)

- El software encargado del manejo de un determinado dispositivo se conoce como manejador o **driver** de dispositivo.
- En LINUX los drivers son un conjunto de rutinas residentes en memoria y que se ejecutan con los máximos privilegios, ya que forman parte del núcleo.
- Los drivers de dispositivos pueden ser **insertados en el núcleo** en forma de módulos, utilizando el comando ***insmod*.**

Driver: uso de uno genérico (4)

- **Característica de Linux**: abstracción en el manejo de los dispositivos físicos.
- Los dispositivos son tratados como archivos (se abren, cierran, leen y escriben usando las mismas *llamadas al sistema* ***que para manipular archivos***).
- Cada dispositivo es representado por un **archivo de dispositivo (*device file*),** por ejemplo, el **primer disco duro** del sistema es representado por ***/dev/hda***.

Driver: uso de uno genérico (5)

- Todos los ficheros de dispositivo presentes en el sistema se encuentran en el directorio ***/dev***.

```
crw-rw-rw- 1 root root 10,  3, Nov 30 1993 tty0
brw-rw-rw- 1 root disk  2,  0, May  1 1999 fd0
brw-rw-rw- 1 root disk  2,  1, May  1 1999 fd1
brw-rw-rw- 1 root disk  3,  0, May  1 1999 hda
brw-rw-rw- 1 root disk  3, 64, May  1 1999 hdb
crw-rw---- 1 root lp    6,  0, May  1 1999 lp0
crw-rw---- 1 root lp    6,  1, May  1 1999 lp1
crw-rw-rw- 1 root uucp  4, 64, May  1 1999 ttyS0
crw-rw-rw- 1 root uucp  4, 65, May  1 1999 ttyS1
```

Driver: uso de uno genérico (6)

- Cada **archivo de dispositivo** tiene asociado un número que lo identifica: el ***major number***.

  /dev/f0 el *major_number* **2**, ***/dev/lp0*** el **6**.

- Dispositivos del **mismo tipo** tienen asociado el mismo *major_number* (los diskettes *fd0* y *fd1* el **2**), porque son atendidos por el **mismo driver**.

- Muestran *minor_numbers* diferentes (0, 1), indican al driver cuál de los dispositivos que solicita servicio (cuando un proceso accede a ese archivo de dispositivo).

Driver: uso de uno genérico (7)

- Algunos de los archivos de dispositivos son identificados por una "**c**", indicando que ese archivo especial esta asociado a un dispositivo de **carácter**.
- Otros son identificados por "**b**" en la primer columna, indicando que dicho archivo esta asociado a un dispositivo de **bloque**.
- **Linux distingue entre dos tipos básicos de dispositivos:** dispositivos de **caracteres** y dispositivos de **bloques**.

Driver:

Dispositivos de Carácter ó Bloques

Driver: Dispositivos de caracter

- Se accede como un archivo normal.
- El **driver** asociado al dispositivo es el que implementa ese proceder.
- El driver implementa las llamadas al sistema ***open***, ***close***, ***read*** *y* ***write***.
- La **consola** y el **puerto paralelo** son dos ejemplos de este tipo de dispositivos, que pueden ser accedidos a través los archivos especiales ***/dev/tty*** y ***/dev/lp***.

Driver: Dispositivos de bloques

- Un dispositivo de bloque puede albergar un **sistema de archivos** (disco duro).
- En UNIX un dispositivo de bloque puede ser accedido en múltiplos del tamaño de bloque (normalmente de 1.024 bytes).
- Se puede leer y escribir dispositivos de bloque igual que para dispositivos de caracteres, pero transfiriendo varios bytes (**variable**) por vez.
- Tienen acceso aleatorio (por bloque).

Dispositivos de almacenamiento masivo: diskette, disco duro, etc.

Operaciones Asociadas al Driver

Operaciones asociadas al driver (1)

1) Posición actual del archivo

int (*__lseek__) (struct inode *, struct file *, off_t, int);

Tipo de archivo

2) Leer datos desde el dispositivo

int (*__read__) (struct inode *, struct file *, char *, int);

3) Escribir datos en el dispositivo

int (*__write__) (struct inode *, struct file *, const char *, int);

4) Apertura del dispositivo

int (*__open__) (struct inode *, struct file *);

5) Cierre del dispositivo

int (*__release__) (struct inode *, struct file *);

Buffer escritura

Descriptor de archivo

Buffer lectura

Operaciones asociadas al driver (2)

6) Determinar si el dispositivo es de r, w, rw

int (***select**) (struct inode *, struct file *, int,
select_table *);

7) Especificar comandos del driver

int (***ioctl**) (struct inode *, struct file *, unsigned int,
unsigned long);

8) Peticiones de mapeo de memoria

int (***mmap**) (struct inode *, struct file *,
struct vm_area_struct *);

9) Sincroniza el dispositivo

int (***fsync**) (struct inode *, struct file *);

struct file_operations

```
struct file_operations nombre_modulo_fops = {
        nombre_modulo_lseek,
        nombre_modulo_read,
        nombre_modulo_write,
        NULL, /*  readdir */     NULL, /*  select */
        nombre_modulo_ioctl,
        NULL,                   /*  mmap */
        nombre_modulo_open,
        nombre_modulo_release,
        NULL, /*  fsync */       NULL, /*  fasync */
        NULL, /*  check media change */
        NULL, /*  revalidate */         };
```

```
struct file_operations scull_fops = {
            read: nombre_modulo_read,
           write: nombre_modulo_write,
            open: nombre_modulo_open,
         release: nombre_modulo_release  };
```

Información asociada al archivo: **struct_file**

Modo archivo:

mode_t **f_mode**;

Posición actual dentro del archivo:

loff_t **f_pos**;

Flags del archivo:

unsigned short **f_flags**;

Inode asociado al archivo:

struct inode ***f_inode**;

Operaciones asociadas con el archivo:

struct file_operations ***f_op**;

Puntero a datos asociados al archivo:

void ***private_data**;

Apertura del archivo

Apertura del archivo (open)

- Chequear posibles errores relacionados con la especificación del dispositivo.
- Inicializar el dispositivo (en la primera apertura).
- Identificar el número minor.
- Actualizar el puntero **f_op**.
- Asignar y completar todas las estructuras de datos de:

  filp->private_data.

- Incrementar el contador de uso.

Acceso a varios dispositivos:
open en varios modos

```
#define TYPE (dev) (MINOR (dev) >> 4)
#define NUM  (dev) (MINOR (dev) & 0xf)

struct file_operations *nom_mod_fop_array[]={
     &nom_mod_fops,          /* tipo 0 */
     &nom_mod_prov_fops,     /* tipo 1 */
     &nom_mod_pipe_fops,     /* tipo 2 */
     &nom_mod_sngl_fops,     /* tipo 3 */
     &nom_mod_user_fops,     /* tipo 4 */
     &nom_mod_wusr_fops,     /* tipo 5 */
};

#define NOMBRE_MODULO_MAX_TYPE 5
```

OPEN: Ejemplo de implementación típica

```
int nom_mod_open(struct inode,
                                        struct file *filp){
   int type = TYPE (inode -> i_rdev);
   int num = NUM (inode->i_rdev);
   nom_mod_dev *dev;
   if(type){
      if(type>NOMBRE_MODULO_MAX_TYPE) return -ENODEV;
      filp->f_op = nom_mod_fop_array[type];
      return filpe->f_op->open(inode,filp);
   }
   if(num>=nom_mod_nr_devs) return -ENODEV;
   dev = &nom_mod_devices[num];
   if((filp->f_flags & O_ACCMODE) == O_WRONLY)
                                        nom_mod_tim(dev);

   filp->private_data = dev;
   MOD_INC_USE_COUNT;
   return 0;
}
```

Declaraciones

Verificación

TipoAcceso

Funciones mínimas a programar

Cierre del archivo asociado al driver:

RELEASE

RELEASE: Cierre del archivo asociado al driver

- Decrementar el contador de uso.
- Liberar cualquier estructura de datos puesta en: filp->private_data.
- Eliminar el dispositivo en la última operación de cierre que se realice después de tantas aperturas como hayan existido.

```
int nom_mod_release(struct inode, struct file *filp)
{
      MOD_DEC_USE_COUNT;
}
```

Implementación:

READ & WRITE

READ & WRITE: Implementación (1)

int (*__read__) (struct inode *, struct file *, char *, int);
int (*__write__) (struct inode *, struct file *, const char *, int);

Argumentos:

- **struct inode * inode:** Puntero a el inode del archivo especial que fue accedido.
- **struct file * file:** Puntero a la estructura file para este dispositivo.
- **char * buf:** Es un buffer de caracteres a ser leido o escrito. Está localizado en el espacio de memoria del usuario y debe ser accedido usando las macros get_fs*(), put_fs*(), and memcpy*fs(). El espacio del usuario es inaccesible durante una interrupción.
- **int count:** Es la cantidad de caracteres a ser leidos o escritos en el buffer. Puede ser del tamaño del buffer.

READ & WRITE: Implementación (2)

Si no hay función **read()** o **write()** Registrada en la estructura **file_operations**, y el dispositivo es de:

- **Caracteres** retornará –EINVAL;
- **Bloques**, serán manejadas por el VFS.

READ: Implementación típica (1)

```
read_write_t nom_mod_read(struct inode,
                          struct file *filp,
                                char *buf,
                             count_t count){

nom_mod_dev *dev = filpe->private_data;
int quantum = dev->quantum;
int qset = dev->qset;
int itemsize = quantum * qset;
unsigned long f_pos = (unsigned long)
                                 (filp->f_pos);
int item, s_pos, q_pos, rest;
```

READ: Implementación típica (2)

```
...
if(f_pos > dev->size)
   return 0;

if(f_pos+count > dev->size)
   count = dev-size - f_pos;

item = f_pos / itemsize;
rest = f_pos % itemsize;
s_pos = rest / quantum;
q_pos = rest % quantum;
...
```

Verificación de tamaño y posición.

Cálculos varios

READ: Implementación típica (3)

```
...
dev = nom_mod_follow( dev, item );
if(!dev->data)
   return 0;
if(!dev->data[s_pos])
   return 0;
if(count>quantum-q_pos)
   count = quantum -q_pos;
dev->usage++;
memcpy_tofs(buf,
            dev->data[s_pos] +qpos,
                                count);
filp->f_pos+=count;
return count;
}
```

Carga y verificación de datos del dispositivo.

Uso operación

Copia y actualización de posición

WRITE: Implementación típica (1)

```
read_write_t nom_mod_write(struct inode,
                           struct file *filp,
                             const char *buf,
                              count_t count){
nom_mod_dev *dev = filpe->private_data;
nom_mod_dev *dptr;
int quantum = dev->quantum;
int qset = dev->qset;
int itemsize = quantum * qset;
unsigned long f_pos = (unsigned long)
                                 (filp->f_pos);
int item, s_pos, q_pos, rest;
...
```

WRITE: Implementación típica (2)

```
...
item = f_pos / itemsize;
rest = f_pos % itemsize;
s_pos = rest / quantum;
q_pos = rest % quantum;
dptr = nom_mod_follow(dev,item);
if(!dptr->data){
   dptr->data =
      kmalloc( qset *sizeof(char *),
               GFP_KERNEL );
   if(!dptr->data)
      return -ENOMEM;
   memset(dptr->data,
      0, qset *sizeof(char *));
}
...
```

Cálculos varios

Carga de datos disp.

Manejos varios

WRITE: Implementación típica (3)

```
...

if(!dptr->data[s_pos]){
   dptr->data[s_pos] =
      kmalloc(quantum,GFP_KERNEL);
   if(!dptr->data[s_pos])
      return -ENOMEM;
}

if(count>quantum -q_pos)
   count = quantum -q_pos;

...
```

Manejos varios

WRITE: Implementación típica (4)

```
...
dev->usage++;
memcpy_fromfs(
   dptr->data[s_pos] +q_pos,
   buf,
   count  );
dev->usage--;
if(dev-size < f_pos+count)
   dev->size=f_pos+count;
filp->f_pos+=count;
return count;
}
```

Cuenta de uso write y copia.

Operación y cuenta.

Implementación típica:

LSEEK

LSEEK: Implementación típica (1)

```
int (*lseek)
 (struct inode *, struct file *, off_t, int);
```

Argumentos:

- **struct inode * inode:** Puntero a la estructura inode del dispositivo.
- **struct file * file:** Puntero a la estructura file del dispositivo.
- **off_t offset:** Desplazamiento desde el orígen donde se debe ir.
- **int origin:**
  - 0 = Toma el desplazamiento absoluto desde el orígen (donde 0 es el principio).
  - 1 = Desplazamiento relativo a la posición actual.
  - 2 = Desplazamiento relativo a la posición final.

Retorno:

- -errno
- (>= 0): posición absoluta luego del lseek.

LSEEK: Implementación típica (2)

Si no hay **lseek()** definido, el kernel realiza la acción por defecto, la cual puede modificar el **file->f_poselement**.

Si el **origin** es 2, la acción por defecto retornará -EINVAL si **file->f_inode** es NULL,de otra forma setea **file->f_pos** a **file->f_inode->i_size** + **offset**.

Implementación típica:

SELECT

SELECT: Implementación típica (1)

```
int (*select) ( struct inode *,
                                  struct file *,
                                              int,
                                 select_table *);
```

Argumentos:

- **struct inode * inode:** Puntero a la estructura inode del dispositivo.
- **struct file * file:** Puntero a la estructura file del dispositivo.
- **int sel_type:** Define el tipo de selección que se debe realizar: SEL_IN para leer, SEL_OUT para escribir, SEL_EX para una excepción.
- **select_table * wait:** ...

SELECT: Implementación típica (2)

...

select_table * wait: Si no es NULL y no hay condición de error, dormirá el proceso, y se despertará cuando el dispositivo esté listo, usualmente a través de una interrupción. Si es NULL, el driver chequea inmediatamente que el dispositivo esté listo.

Esto no requiere del uso de la función **sleep_on*()**, se usa **select_wait()**. Este estado de dormido tiene el mismo comportamiento que **sleep_on_interruptible()**, y **wake_up_interruptible()** será el proceso que lo despierta.

SELECT: Implementación típica (3)

El proceso no es puesto a dormir hasta la llamada al sistema **sys_select()**, la cual originalmente es llamada por **select()**. **select_wait()** agrega el proceso a la cola de espera, pero **do_select()** (llamado desde **sys_select()**) pone el proceso a dormir cambiando el estado a **TASK_INTERRUPTIBLE** y llamando a **schedule()**.

select_wait() retorna: (0) si el dispositivo no está listo if the device is not ready, (1)si el dispositivo está listo.

No se puede usar timeout junto con timers.

Transferencia de **datos**

entre el espacio

del **kernel**

y

del **usuario**

Transferencia de datos entre el espacio del kernel y del usuario

- **Del USUARIO al DISPOSITIVO (write)**

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

- **Del DISPOSITIVO al USUARIO (read)**

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

Valor **retorno** en lectura/escritura

```
VP = Valor Pedido
VR = Valor Resultado
```

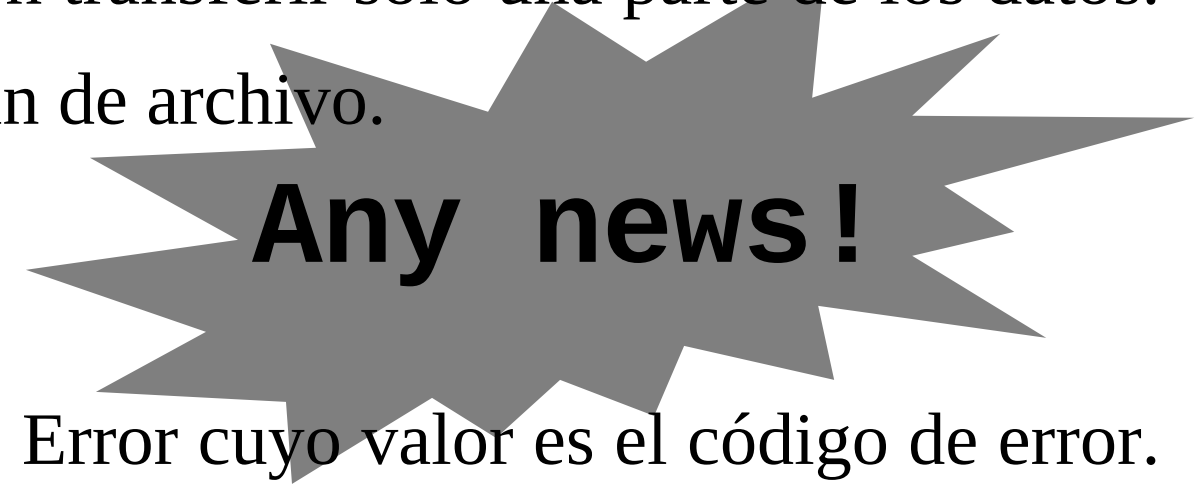
- **VP = = VR.**

  Transferencia realizada **satisfactoriamente**.

- **VP > VR.**

  Se lograron transferir sólo una parte de los datos.

- **VR = = 0**. Fin de archivo.



- **VR < 0.**

  Error cuyo valor es el código de error.

Esto es todo ...