Devices Drivers
Desarrollo de un char device en Linux

Alejandro Furfaro

Abril 2011

Agenda

1. Conceptos Básicos
2. Hands on
3. Char Devices

“We are back to the times when men where men and wrote their own device drivers. . . ”

Linus Torvalds.

Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Es la mediación entre los dispositivos de hardware y los procesos del sistema o de usuario.
- Se ocupa de resolver el mecanismo de acceso al hardware.
- No se concentra en la política de manejo de la información, aspecto que queda para el software de usuario.
  - El Driver de disco se ocupa del acceso físico al disco.
  - El File System Manager formatea los datos para el usuario (política de manejo de información).

Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Es la mediación entre los dispositivos de hardware y los procesos del sistema o de usuario.
- Se ocupa de resolver el mecanismo de acceso al hardware.
- No se concentra en la política de manejo de la información, aspecto que queda para el software de usuario.
  - El Driver de disco se ocupa del acceso físico al disco.
  - El File System Manager formatea los datos para el usuario (política de manejo de información).

Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Es la mediación entre los dispositivos de hardware y los procesos del sistema o de usuario.
- Se ocupa de resolver el mecanismo de acceso al hardware.
- No se concentra en la política de manejo de la información, aspecto que queda para el software de usuario.
  - El Driver de disco se ocupa del acceso físico al disco.
  - El File System Manager formatea los datos para el usuario (política de manejo de información).

Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Es la mediación entre los dispositivos de hardware y los procesos del sistema o de usuario.
- Se ocupa de resolver el mecanismo de acceso al hardware.
- No se concentra en la política de manejo de la información, aspecto que queda para el software de usuario.
  - El Driver de disco se ocupa del acceso físico al disco.
  - El File System Manager formatea los datos para el usuario (política de manejo de información).

Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Es la mediación entre los dispositivos de hardware y los procesos del sistema o de usuario.
- Se ocupa de resolver el mecanismo de acceso al hardware.
- No se concentra en la política de manejo de la información, aspecto que queda para el software de usuario.
  - El Driver de disco se ocupa del acceso físico al disco.
  - El File System Manager formatea los datos para el usuario (política de manejo de información)

Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Es la mediación entre los dispositivos de hardware y los procesos del sistema o de usuario.
- Se ocupa de resolver el mecanismo de acceso al hardware.
- No se concentra en la política de manejo de la información, aspecto que queda para el software de usuario.
  - El Driver de disco se ocupa del acceso físico al disco.
  - El File System Manager formatea los datos para el usuario (política de manejo de información)

Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Es la mediación entre los dispositivos de hardware y los procesos del sistema o de usuario.
- Se ocupa de resolver el mecanismo de acceso al hardware.
- No se concentra en la política de manejo de la información, aspecto que queda para el software de usuario.
  - El Driver de disco se ocupa del acceso físico al disco.
  - El File System Manager formatea los datos para el usuario (política de manejo de información)

Clasificación

1. Char Devices
2. Block Devices
3. Network Devices
4. Miscelaneous Devices

Char Devices

- Son los mas simples.
- Generalmente en los Sistemas Operativos modernos se acceden como un stream de bytes. Por ejemplo en Linux, se acceden tal como si fuesen nodos del File System. Ejemplos en Linux: TTYs (/dev/console). Serial ports (/dev/ttyS0).
- A diferencia de los archivos comunes, no nos podemos desplazar hacia atrás y hacia adelante (función lseek).
- Acceden a los datos en forma secuencial.
- Generalmente registran sus prestaciones en el Sistema Operativo a través de objetos del FS que responden a las funciones standard de acceso a archivos. Si nuestro propósito es desarrollar un SO POSIX compatible, estas funciones serán `open()`, `read()`, `write()`, `close()`, etc.

Char Devices

- Son los mas simples.
- Generalmente en los Sistemas Operativos modernos se acceden como un stream de bytes. Por ejemplo en Linux, se acceden tal como si fuesen nodos del File System. Ejemplos en Linux: TTYs (/dev/console). Serial ports (/dev/ttyS0).
- A diferencia de los archivos comunes, no nos podemos desplazar hacia atrás y hacia adelante (función lseek).
- Acceden a los datos en forma secuencial.
- Generalmente registran sus prestaciones en el Sistema Operativo a través de objetos del FS que responden a las funciones standard de acceso a archivos. Si nuestro propósito es desarrollar un SO POSIX compatible, estas funciones serán `open()`, `read()`, `write()`, `close()`, etc.

Char Devices

- Son los mas simples.
- Generalmente en los Sistemas Operativos modernos se acceden como un stream de bytes. Por ejemplo en Linux, se acceden tal como si fuesen nodos del File System. Ejemplos en Linux: TTYs (/dev/console). Serial ports (/dev/ttyS0).
- A diferencia de los archivos comunes, no nos podemos desplazar hacia atrás y hacia adelante (función lseek).
- Acceden a los datos en forma secuencial.
- Generalmente registran sus prestaciones en el Sistema Operativo a través de objetos del FS que responden a las funciones standard de acceso a archivos. Si nuestro propósito es desarrollar un SO POSIX compatible, estas funciones serán `open()`, `read()`, `write()`, `close()`, etc.

Char Devices

- Son los mas simples.
- Generalmente en los Sistemas Operativos modernos se acceden como un stream de bytes. Por ejemplo en Linux, se acceden tal como si fuesen nodos del File System. Ejemplos en Linux: TTYs (/dev/console). Serial ports (/dev/ttyS0).
- A diferencia de los archivos comunes, no nos podemos desplazar hacia atrás y hacia adelante (función lseek).
- Acceden a los datos en forma secuencial.
- Generalmente registran sus prestaciones en el Sistema Operativo a través de objetos del FS que responden a las funciones standard de acceso a archivos. Si nuestro propósito es desarrollar un SO POSIX compatible, estas funciones serán `open ( )`, `read ( )`, `write ( )`, `close ( )`, etc.

Char Devices

- Son los mas simples.
- Generalmente en los Sistemas Operativos modernos se acceden como un stream de bytes. Por ejemplo en Linux, se acceden tal como si fuesen nodos del File System. Ejemplos en Linux: TTYs (/dev/console). Serial ports (/dev/ttyS0).
- A diferencia de los archivos comunes, no nos podemos desplazar hacia atrás y hacia adelante (función lseek).
- Acceden a los datos en forma secuencial.
- Generalmente registran sus prestaciones en el Sistema Operativo a través de objetos del FS que responden a las funciones standard de acceso a archivos. Si nuestro propósito es desarrollar un SO POSIX compatible, estas funciones serán `open ( )`, `read ( )`, `write ( )`, `close ( )`, etc.

Char Devices

- Son los mas simples.
- Generalmente en los Sistemas Operativos modernos se acceden como un stream de bytes. Por ejemplo en Linux, se acceden tal como si fuesen nodos del File System. Ejemplos en Linux: TTYs (/dev/console). Serial ports (/dev/ttyS0).
- A diferencia de los archivos comunes, no nos podemos desplazar hacia atrás y hacia adelante (función lseek).
- Acceden a los datos en forma secuencial.
- Generalmente registran sus prestaciones en el Sistema Operativo a través de objetos del FS que responden a las funciones standard de acceso a archivos. Si nuestro propósito es desarrollar un SO POSIX compatible, estas funciones serán `open ( )` , `read ( )` , `write ( )` , `close ( )` , etc.

Block Devices

- Agregan complejidad a su interfaz.
- Al igual que los char devices, se mapean como Nodos del File System. Si miramos en Linux en el directorio /dev, encontraremos los dispositivos de nuestro disco rígido, DVD, etc.
- A diferencia de los Char Devices registran sus prestaciones en estructuras propias del kernel.
- La diferencia pasa por como el kernel maneja internamente los datos. Por lo regular es de a bloques (512 o 1024 bytes).
- Transmiten o reciben bloques de bytes a demanda del kernel mediante la función request. Algo diferente de la simple interfaz de los char devices.
- Son dispositivos que pueden hostear un File System. Ej.: Discos, Cintas, DVDs, etc.

Block Devices

- Agregan complejidad a su interfaz.
- Al igual que los char devices, se mapean como Nodos del File System. Si miramos en Linux en el directorio /dev, encontraremos los dispositivos de nuestro disco rígido, DVD, etc.
- A diferencia de los Char Devices registran sus prestaciones en estructuras propias del kernel.
- La diferencia pasa por como el kernel maneja internamente los datos. Por lo regular es de a bloques (512 o 1024 bytes).
- Transmiten o reciben bloques de bytes a demanda del kernel mediante la función request. Algo diferente de la simple interfaz de los char devices.
- Son dispositivos que pueden hostear un File System. Ej.: Discos, Cintas, DVDs, etc.

Block Devices

- Agregan complejidad a su interfaz.
- Al igual que los char devices, se mapean como Nodos del File System. Si miramos en Linux en el directorio /dev, encontraremos los dispositivos de nuestro disco rígido, DVD, etc.
- A diferencia de los Char Devices registran sus prestaciones en estructuras propias del kernel.
- La diferencia pasa por como el kernel maneja internamente los datos. Por lo regular es de a bloques (512 o 1024 bytes).
- Transmiten o reciben bloques de bytes a demanda del kernel mediante la función request. Algo diferente de la simple interfaz de los char devices.
- Son dispositivos que pueden hostear un File System. Ej.: Discos, Cintas, DVDs, etc.

Block Devices

- Agregan complejidad a su interfaz.
- Al igual que los char devices, se mapean como Nodos del File System. Si miramos en Linux en el directorio /dev, encontraremos los dispositivos de nuestro disco rígido, DVD, etc.
- A diferencia de los Char Devices registran sus prestaciones en estructuras propias del kernel.
- La diferencia pasa por como el kernel maneja internamente los datos. Por lo regular es de a bloques (512 o 1024 bytes).
- Transmiten o reciben bloques de bytes a demanda del kernel mediante la función request. Algo diferente de la simple interfaz de los char devices.
- Son dispositivos que pueden hostear un File System. Ej.: Discos, Cintas, DVDs, etc.

Block Devices

- Agregan complejidad a su interfaz.
- Al igual que los char devices, se mapean como Nodos del File System. Si miramos en Linux en el directorio /dev, encontraremos los dispositivos de nuestro disco rígido, DVD, etc.
- A diferencia de los Char Devices registran sus prestaciones en estructuras propias del kernel.
- La diferencia pasa por como el kernel maneja internamente los datos. Por lo regular es de a bloques (512 o 1024 bytes).
- Transmiten o reciben bloques de bytes a demanda del kernel mediante la función request. Algo diferente de la simple interfaz de los char devices.
- Son dispositivos que pueden hostear un File System. Ej.: Discos, Cintas, DVDs, etc.

Block Devices

- Agregan complejidad a su interfaz.
- Al igual que los char devices, se mapean como Nodos del File System. Si miramos en Linux en el directorio /dev, encontraremos los dispositivos de nuestro disco rígido, DVD, etc.
- A diferencia de los Char Devices registran sus prestaciones en estructuras propias del kernel.
- La diferencia pasa por como el kernel maneja internamente los datos. Por lo regular es de a bloques (512 o 1024 bytes).
- Transmiten o reciben bloques de bytes a demanda del kernel mediante la función request. Algo diferente de la simple interfaz de los char devices.
- Son dispositivos que pueden hostear un File System. Ej.: Discos, Cintas, DVDs, etc.

Block Devices

- Agregan complejidad a su interfaz.
- Al igual que los char devices, se mapean como Nodos del File System. Si miramos en Linux en el directorio /dev, encontraremos los dispositivos de nuestro disco rígido, DVD, etc.
- A diferencia de los Char Devices registran sus prestaciones en estructuras propias del kernel.
- La diferencia pasa por como el kernel maneja internamente los datos. Por lo regular es de a bloques (512 o 1024 bytes).
- Transmiten o reciben bloques de bytes a demanda del kernel mediante la función request. Algo diferente de la simple interfaz de los char devices.
- Son dispositivos que pueden hostear un File System. Ej.: Discos, Cintas, DVDs, etc.

Network Devices

- Los network devices parecen ser iguales a los block devices. *Pero solo en apariencia*.
- Controlan las interfaces durante las transacciones de paquetes de datos en red contra un equipo remoto, pero sin conocer en detalle la composición de las transacciones que conforman esos paquetes.
- No siempre se relacionan con dispositivos de hardware (loopback por ejemplo).
- No constituyen dispositivos orientados a stream. Por esta razón no son mapeables como nodos en el File System del sistema. Por ejemplo en Linux, si miramos /dev, no hay ningún dispositivo relacionado con la interfaz de red.

Network Devices

- Los network devices parecen ser iguales a los block devices. *Pero solo en apariencia*.
- Controlan las interfaces durante las transacciones de paquetes de datos en red contra un equipo remoto, pero sin conocer en detalle la composición de las transacciones que conforman esos paquetes.
- No siempre se relacionan con dispositivos de hardware (loopback por ejemplo).
- No constituyen dispositivos orientados a stream. Por esta razón no son mapeables como nodos en el File System del sistema. Por ejemplo en Linux, si miramos /dev, no hay ningún dispositivo relacionado con la interfaz de red.

Network Devices

- Los network devices parecen ser iguales a los block devices. *Pero solo en apariencia*.
- Controlan las interfaces durante las transacciones de paquetes de datos en red contra un equipo remoto, pero sin conocer en detalle la composición de las transacciones que conforman esos paquetes.
- No siempre se relacionan con dispositivos de hardware (loopback por ejemplo).
- No constituyen dispositivos orientados a stream. Por esta razón no son mapeables como nodos en el File System del sistema. Por ejemplo en Linux, si miramos /dev, no hay ningún dispositivo relacionado con la interfaz de red.

Network Devices

- Los network devices parecen ser iguales a los block devices. *Pero solo en apariencia*.
- Controlan las interfaces durante las transacciones de paquetes de datos en red contra un equipo remoto, pero sin conocer en detalle la composición de las transacciones que conforman esos paquetes.
- No siempre se relacionan con dispositivos de hardware (loopback por ejemplo).
- No constituyen dispositivos orientados a stream. Por esta razón no son mapeables como nodos en el File System del sistema. Por ejemplo en Linux, si miramos /dev, no hay ningún dispositivo relacionado con la interfaz de red.

Network Devices

- Los network devices parecen ser iguales a los block devices. *Pero solo en apariencia*.
- Controlan las interfaces durante las transacciones de paquetes de datos en red contra un equipo remoto, pero sin conocer en detalle la composición de las transacciones que conforman esos paquetes.
- No siempre se relacionan con dispositivos de hardware (loopback por ejemplo).
- No constituyen dispositivos orientados a stream. Por esta razón no son mapeables como nodos en el File System del sistema. Por ejemplo en Linux, si miramos /dev, no hay ningún dispositivo relacionado con la interfaz de red.

Miscelaneous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Algunos autores clasifican en esta categoría especial, a los drivers de los controladores de buses, ya que son bastante particulares.
  - PCI
  - USB
  - SCSI

Miscelaneous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Algunos autores clasifican en esta categoría especial, a los drivers de los controladores de buses, ya que son bastante particulares.
  - PCI.
  - USB.
  - SCSI

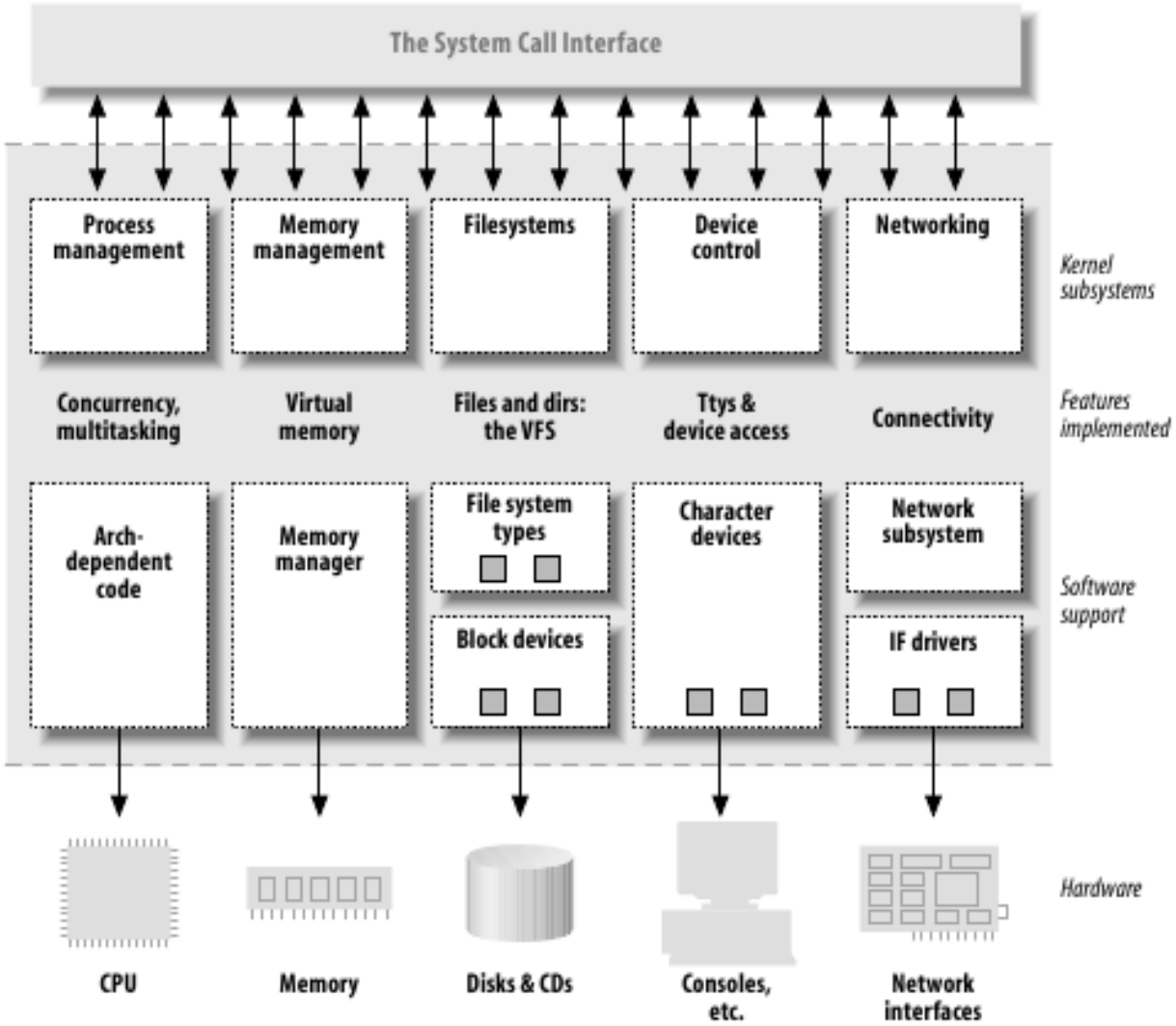
Miscelaneous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Algunos autores clasifican en esta categoría especial, a los drivers de los controladores de buses, ya que son bastante particulares.
  - PCI.
  - USB.
  - SCSI

Miscelaneous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Algunos autores clasifican en esta categoría especial, a los drivers de los controladores de buses, ya que son bastante particulares.
  - PCI.
  - USB.
  - SCSI

Inserción en el kernel



Concurrencia

- Un kernel que se precie, debe ser concurrente.
- Por lo tanto un driver debe estar escrito con la idea que en un mismo instante ocurren varias cosas. Debe ser re entrante.
- El kernel de LINUX, por ejemplo, es concurrente, .
- Desde el kernel no tenemos los recursos que usamos en las aplicaciones
  - No se accede a las system call standard
  - No hay mecanismos para Inter Process Communication
  - Por ejemplo, en Linux, para averiguar desde cierto punto de un driver, cual es el proceso que invocó alguna de sus funciones hay que escribir este bloque:

```
printk("The process is \"%s\" (pid %)\n", current->comm, current->pid);
```

Concurrencia

- Un kernel que se precie, debe ser concurrente.
- Por lo tanto un driver debe estar escrito con la idea que en un mismo instante ocurren varias cosas. Debe ser re entrante.
- El kernel de LINUX, por ejemplo, es concurrente, .
- Desde el kernel no tenemos los recursos que usamos en las aplicaciones
  - No se accede a las syscall standard
  - No hay mecanismos para IPC, Process Communication
  - Por ejemplo, en Linux, para averiguar desde qué modulo driver, cual es el proceso que invocó alguna de sus funciones hay que escribir este bloque:

```
printk("The process is \"%s\" (pid %i)\n", current->comm, current->pid);
```

Concurrencia

- Un kernel que se precie, debe ser concurrente.
- Por lo tanto un driver debe estar escrito con la idea que en un mismo instante ocurren varias cosas. Debe ser re entrante.
- El kernel de LINUX, por ejemplo, es concurrente, .
- Desde el kernel no tenemos los recursos que usamos en las aplicaciones
  - No se accede a las system call standard
  - No hay mecanismos para IPC, Process Communication
  - Por ejemplo, en Linux, para averiguar desde qué modulo o driver, cual es el proceso que invocó alguna función o funcionalidad y proceder en base a esto.

```
printk("The process is \"%s\" (pid %)\n", current->comm, current->pid);
```

Concurrencia

- Un kernel que se precie, debe ser concurrente.
- Por lo tanto un driver debe estar escrito con la idea que en un mismo instante ocurren varias cosas. Debe ser re entrante.
- El kernel de LINUX, por ejemplo, es concurrente, .
- Desde el kernel no tenemos los recursos que usamos en las aplicaciones
  - No se accede a las system call standard.
  - No hay mecanismos para Inter Process Communication
  - Por ejemplo, en Linux, para averiguar desde qué proceso un driver, cual es el proceso que invocó alguna de sus funciones hay que buscar este dato.

```
printk("The process is \"%s\" (pid %)\n", current->comm, current->pid);
```

Concurrencia

- Un kernel que se precie, debe ser concurrente.
- Por lo tanto un driver debe estar escrito con la idea que en un mismo instante ocurren varias cosas. Debe ser re entrante.
- El kernel de LINUX, por ejemplo, es concurrente, .
- Desde el kernel no tenemos los recursos que usamos en las aplicaciones
  - No se accede a las system call standard.
  - No hay mecanismos para Inter Process Comunication.
  - Por ejemplo, en Linux, para averiguar desde dentro de un driver, cual es el proceso que invocó alguna de sus funcioneshay que escribir esta línea:

```
printk("The process is \"%s\" (pid %)\n", current->comm, current->pid);
```

Concurrencia

- Un kernel que se precie, debe ser concurrente.
- Por lo tanto un driver debe estar escrito con la idea que en un mismo instante ocurren varias cosas. Debe ser re entrante.
- El kernel de LINUX, por ejemplo, es concurrente, .
- Desde el kernel no tenemos los recursos que usamos en las aplicaciones
  - No se accede a las system call standard.
  - No hay mecanismos para Inter Process Comunication.
  - Por ejemplo, en Linux, para averiguar desde dentro de un driver, cual es el proceso que invocó alguna de sus funcioneshay que escribir esta línea:

```
printk("The process is \"%s\" (pid %)\n", current->comm, current->pid);
```

Concurrencia

- Un kernel que se precie, debe ser concurrente.
- Por lo tanto un driver debe estar escrito con la idea que en un mismo instante ocurren varias cosas. Debe ser re entrante.
- El kernel de LINUX, por ejemplo, es concurrente, .
- Desde el kernel no tenemos los recursos que usamos en las aplicaciones
  - No se accede a las system call standard.
  - No hay mecanismos para Inter Process Comunication.
  - Por ejemplo, en Linux, para averiguar desde dentro de un driver, cual es el proceso que invocó alguna de sus funcioneshay que escribir esta línea:

```
printk("The process is \"%s\" (pid %)\n", current->comm, current->pid);
```

Concurrencia

- Un kernel que se precie, debe ser concurrente.
- Por lo tanto un driver debe estar escrito con la idea que en un mismo instante ocurren varias cosas. Debe ser re entrante.
- El kernel de LINUX, por ejemplo, es concurrente, .
- Desde el kernel no tenemos los recursos que usamos en las aplicaciones
  - No se accede a las system call standard.
  - No hay mecanismos para Inter Process Comunication.
  - Por ejemplo, en Linux, para averiguar desde dentro de un driver, cual es el proceso que invocó alguna de sus funcioneshay que escribir esta línea:

```
printk("The process is \"%s\" (pid %)\n", current->comm, current->pid);
```

Vamos a un caso práctico que funciona bien: Linux

- Linux genera drivers en una estructura de programación llamada módulo

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

static int hello_init(void)
{
   printk(KERN_ALERT "Hola, mundo!\n");
   return 0;
}
static void hello_exit(void)
{
   printk(KERN_ALERT   Adios   , Mundo cruel!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

No luce como un programa en C normal ¿no?

- No usa función main!!! (¡¿con que se come entonces?!)
- Compilarlo es toda una cuestión. Lo veremos mas adelante.
- Asumiendo que ya está compilado, (no linkea solo compila), el producto es un archivo llamado hola.ko (ko por kernel object)
- La forma de uso es (atención con el prompt ;)):

```
# insmod ./hello.ko
  Hola mundo!
# rmmod hello
  Adios mundo cruel!
#
```

No luce como un programa en C normal ¿no?

- No usa función main!!! (¡¿con que se come entonces?!)
- Compilarlo es toda una cuestión. Lo veremos mas adelante.
- Asumiendo que ya está compilado, (no linkea solo compila), el producto es un archivo llamado hola.ko (ko por kernel object)
- La forma de uso es (atención con el prompt ;)):

```
# insmod ./hello.ko
  Hola mundo!
# rmmod hello
  Adios mundo cruel!
#
```

No luce como un programa en C normal ¿no?

- No usa función main!!! (¡¿con que se come entonces?!)
- Compilarlo es toda una cuestión. Lo veremos mas adelante.
- Asumiendo que ya está compilado, (no linkea solo compila), el producto es un archivo llamado hola.ko (ko por kernel object)
- La forma de uso es (atención con el prompt ;)):

```
# insmod ./hello.ko
  Hola mundo!
# rmmod hello
  Adios mundo cruel!
#
```

No luce como un programa en C normal ¿no?

- No usa función main!!! (¡¿con que se come entonces?!)
- Compilarlo es toda una cuestión. Lo veremos mas adelante.
- Asumiendo que ya está compilado, (no linkea solo compila), el producto es un archivo llamado hola.ko (ko por kernel object)
- La forma de uso es (atención con el prompt ;)):

```
# insmod ./hello.ko
  Hola mundo!
# rmmod hello
  Adios mundo cruel!
#
```

No luce como un programa en C normal ¿no?

- No usa función main!!! (¡¿con que se come entonces?!)
- Compilarlo es toda una cuestión. Lo veremos mas adelante.
- Asumiendo que ya está compilado, (no linkea solo compila), el producto es un archivo llamado hola.ko (ko por kernel object)
- La forma de uso es (atención con el prompt ;)):

```
# insmod ./hello.ko
  Hola mundo!
# rmmod hello
  Adios mundo cruel!
#
```

Por favor que alguien nos ayude!!!!

- insmod y rmmod, se utilizan para manejar nuestro módulo. insmod lo instala, y quedará disponible hasta que se ejecute rmmod.
- insmod hace que se ejecute la función module_init ().
- rmmod hace que se ejecute la función module_exit ().
- Cualquier parecido con las funciones constructora y destructora de un lenguaje orientado a objetos. . . es deliberado :) (al menos todo indica que es la tendencia: un kernel orientado a objetos).

Por favor que alguien nos ayude!!!!

- insmod y rmmod, se utilizan para manejar nuestro módulo. insmod lo instala, y quedará disponible hasta que se ejecute rmmod.
- insmod hace que se ejecute la función module_init ().
- rmmod hace que se ejecute la función module_exit ().
- Cualquier parecido con las funciones constructora y destructora de un lenguaje orientado a objetos. . . es deliberado :) (al menos todo indica que es la tendencia: un kernel orientado a objetos).

Por favor que alguien nos ayude!!!!

- insmod y rmmod, se utilizan para manejar nuestro módulo. insmod lo instala, y quedará disponible hasta que se ejecute rmmod.
- insmod hace que se ejecute la función module_init ().
- rmmod hace que se ejecute la función module_exit ().
- Cualquier parecido con las funciones constructora y destructora de un lenguaje orientado a objetos. . . es deliberado :) (al menos todo indica que es la tendencia: un kernel orientado a objetos).

Por favor que alguien nos ayude!!!!

- insmod y rmmod, se utilizan para manejar nuestro módulo. insmod lo instala, y quedará disponible hasta que se ejecute rmmod.
- insmod hace que se ejecute la función module_init ().
- rmmod hace que se ejecute la función module_exit ().
- Cualquier parecido con las funciones constructora y destructora de un lenguaje orientado a objetos. . . es deliberado :) (al menos todo indica que es la tendencia: un kernel orientado a objetos).

Por favor que alguien nos ayude!!!!

- insmod y rmmod, se utilizan para manejar nuestro módulo. insmod lo instala, y quedará disponible hasta que se ejecute rmmod.
- insmod hace que se ejecute la función module_init ().
- rmmod hace que se ejecute la función module_exit ().
- Cualquier parecido con las funciones constructora y destructora de un lenguaje orientado a objetos. . . es deliberado :) (al menos todo indica que es la tendencia: un kernel orientado a objetos).

Relación del módulo con el kernel

- Una aplicación común y corriente realiza una función de principio a fin, luego de lo cual, simplemente finaliza su ejecución y es removida de la memoria del sistema
- El módulo al “ejecutarse”, solo se registra en el sistema y queda “instalado” en la memoria del sistema para futuros requerimientos. Para removerlo de la memoria es necesario rmmod. Que lo que hace entonces es desinstalarlo.
- Este tipo de enfoque se parece bastante a programación event driven, ya que las funcionalidades del módulo serán invocadas ya sea por un programa de aplicación o responderá a interrupciones, de acuerdo a las necesiades de operación.

Relación del módulo con el kernel

- Una aplicación común y corriente realiza una función de principio a fin, luego de lo cual, simplemente finaliza su ejecución y es removida de la memoria del sistema
- El módulo al “ejecutarse”, solo se registra en el sistema y queda “instalado” en la memoria del sistema para futuros requerimientos. Para removerlo de la memoria es necesario rmmod. Que lo que hace entonces es desinstalarlo.
- Este tipo de enfoque se parece bastante a programación event driven, ya que las funcionalidades del módulo serán invocadas ya sea por un programa de aplicación o responderá a interrupciones, de acuerdo a las necesiades de operación.

Relación del módulo con el kernel

- Una aplicación común y corriente realiza una función de principio a fin, luego de lo cual, simplemente finaliza su ejecución y es removida de la memoria del sistema
- El módulo al “ejecutarse”, solo se registra en el sistema y queda “instalado” en la memoria del sistema para futuros requerimientos. Para removerlo de la memoria es necesario rmmod. Que lo que hace entonces es desinstalarlo.
- Este tipo de enfoque se parece bastante a programación event driven, ya que las funcionalidades del módulo serán invocadas ya sea por un programa de aplicación o responderá a interrupciones, de acuerdo a las necesiades de operación.

Relación del módulo con el kernel

- Una aplicación común y corriente realiza una función de principio a fin, luego de lo cual, simplemente finaliza su ejecución y es removida de la memoria del sistema
- El módulo al “ejecutarse”, solo se registra en el sistema y queda “instalado” en la memoria del sistema para futuros requerimientos. Para removerlo de la memoria es necesario rmmod. Que lo que hace entonces es desinstalarlo.
- Este tipo de enfoque se parece bastante a programación event driven, ya que las funcionalidades del módulo serán invocadas ya sea por un programa de aplicación o responderá a interrupciones, de acuerdo a las necesiades de operación.

Relación del módulo con el kernel

- Otra diferencia es que una aplicación se linkea contra librerías para llamar a esas funciones. El Módulo se linkea en forma dinámica con el kernel, y solo podrá invocar aquellas funciones exportadas por el kernel.
- Por otra parte el módulo informa al kernel sus funciones y los punteros a las mismas de modo que el kernel al recibir llamados desde las aplicaciones para las funciones del módulo las pueda ubicar.

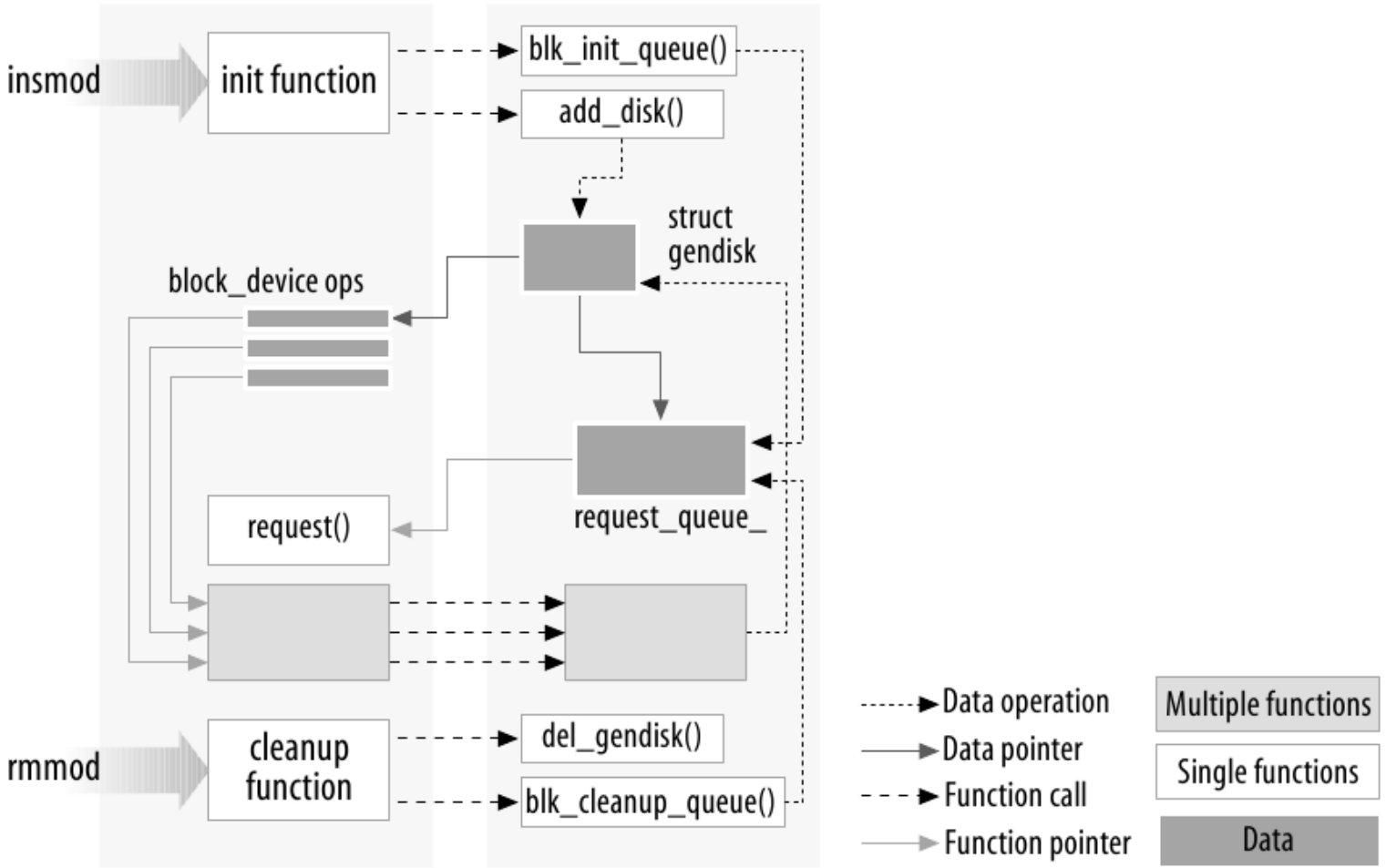
Relación del módulo con el kernel

- Otra diferencia es que una aplicación se linkea contra librerías para llamar a esas funciones. El Módulo se linkea en forma dinámica con el kernel, y solo podrá invocar aquellas funciones exportadas por el kernel.
- Por otra parte el módulo informa al kernel sus funciones y los punteros a las mismas de modo que el kernel al recibir llamados desde las aplicaciones para las funciones del módulo las pueda ubicar.

Relación del módulo con el kernel

- Otra diferencia es que una aplicación se linkea contra librerías para llamar a esas funciones. El Módulo se linkea en forma dinámica con el kernel, y solo podrá invocar aquellas funciones exportadas por el kernel.
- Por otra parte el módulo informa al kernel sus funciones y los punteros a las mismas de modo que el kernel al recibir llamados desde las aplicaciones para las funciones del módulo las pueda ubicar.

Agregan funcionalidad al kernel existente.



Generalidades

- Son dispositivos que deben estar relacionados con uno nodo en /dev.
- Se los crea con un comando desde el shell

  mknod <nombre> <type> <Mn> <mn>

- Donde Mn se denomina Número Mayor y mn Número Menor.

Generalidades

- Son dispositivos que deben estar relacionados con uno nodo en /dev.
- Se los crea con un comando desde el shell

  mknod <nombre> <type> <Mn> <mn>

- Donde Mn se denomina Número Mayor y mn Número Menor.

Generalidades

- Son dispositivos que deben estar relacionados con uno nodo en /dev.
- Se los crea con un comando desde el shell

```
mknod <nombre> <type> <Mn> <mn>
```

- Donde Mn se denomina Número Mayor y mn Número Menor.

Generalidades

- Son dispositivos que deben estar relacionados con uno nodo en /dev.
- Se los crea con un comando desde el shell

```
mknod <nombre> <type> <Mn> <mn>
```

- Donde Mn se denomina Número Mayor y mn Número Menor.

Números Mayor y Menor

- Veamos un estracto del listado del /dev

```
crw-rw-rw- 1 root    root     1,  3 Feb 23 1999  null
crw------- 1 root    root    10,  1 Feb 23 1999  psaux
crw------- 1 root    tty      4,  1 Aug 16 22:22 tty1
crw-rw-rw- 1 root    dialout  4, 64 Jun 30 11:19 ttyS0
crw-rw-rw- 1 root    dialout  4, 65 Aug 16 00:00 ttyS1
crw------- 1 root    sys      7,  1 Feb 23 1999  vcs1
crw------- 1 root    sys      7,129 Feb 23 1999  vcsa1
crw-rw-rw- 1 root    root     1,  5 Feb 23 1999  zero
```

- El kernel usa el Major number para despachar la ejecución del driver correcto en el momento en que se ejecuta la función open () desde el proceso que lo desea acceder.
- El Minor number es usado por el driver. El kernel solo lo pasa al driver para que este lo utilice si lo necesita.

Números Mayor y Menor

- Veamos un estracto del listado del /dev

```
crw-rw-rw- 1 root   root     1,  3  Feb 23 1999  null
crw------- 1 root   root     10, 1  Feb 23 1999  psaux
crw------- 1 root   tty      4,  1  Aug 16 22:22 tty1
crw-rw-rw- 1 root   dialout  4,  64 Jun 30 11:19 ttyS0
crw-rw-rw- 1 root   dialout  4,  65 Aug 16 00:00 ttyS1
crw------- 1 root   sys      7,  1  Feb 23 1999  vcs1
crw------- 1 root   sys      7,  129 Feb 23 1999 vcsa1
crw-rw-rw- 1 root   root     1,  5  Feb 23 1999  zero
```

- El kernel usa el Major number para despachar la ejecución del driver correcto en el momento en que se ejecuta la función open () desde el proceso que lo desea acceder.
- El Minor number es usado por el driver. El kernel solo lo pasa al driver para que este lo utilice si lo necesita.

Números Mayor y Menor

- Veamos un estracto del listado del /dev

```
crw-rw-rw- 1 root   root     1,  3 Feb 23 1999  null
crw------- 1 root   root    10,  1 Feb 23 1999  psaux
crw------- 1 root   tty      4,  1 Aug 16 22:22 tty1
crw-rw-rw- 1 root   dialout  4, 64 Jun 30 11:19 ttyS0
crw-rw-rw- 1 root   dialout  4, 65 Aug 16 00:00 ttyS1
crw------- 1 root   sys      7,  1 Feb 23 1999  vcs1
crw------- 1 root   sys      7,129 Feb 23 1999  vcsa1
crw-rw-rw- 1 root   root     1,  5 Feb 23 1999  zero
```

- El kernel usa el Major number para despachar la ejecución del driver correcto en el momento en que se ejecuta la función open () desde el proceso que lo desea acceder.
- El Minor number es usado por el driver. El kernel solo lo pasa al driver para que este lo utilice si lo necesita.

Números Mayor y Menor

- Veamos un estracto del listado del /dev

```
crw-rw-rw- 1 root   root     1,  3  Feb 23 1999  null
crw------- 1 root   root     10, 1  Feb 23 1999  psaux
crw------- 1 root   tty      4,  1  Aug 16 22:22 tty1
crw-rw-rw- 1 root   dialout  4,  64 Jun 30 11:19 ttyS0
crw-rw-rw- 1 root   dialout  4,  65 Aug 16 00:00 ttyS1
crw------- 1 root   sys      7,  1  Feb 23 1999  vcs1
crw------- 1 root   sys      7,  129 Feb 23 1999 vcsa1
crw-rw-rw- 1 root   root     1,  5  Feb 23 1999  zero
```

- El kernel usa el Major number para despachar la ejecución del driver correcto en el momento en que se ejecuta la función open () desde el proceso que lo desea acceder.
- El Minor number es usado por el driver. El kernel solo lo pasa al driver para que este lo utilice si lo necesita.

Relación entre el mayor Number y el driver

- El módulo de un char device, para obtener del kernel un número mayor, desde la función module_init(), invocará alguna de las siguientes funciones del kernel Definidas en *linux/fs.h*
  - Reserva un rango de major numbers:

```
int register_chrdev_region (dev_t first,
    unsigned int count, char *name);
```

  - Si conocemos exactamente el major number a utilizar:

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

- Cuando es removido, desde la función module_exit (), devolverá al kernel el o los Major Number obtenido por medio de cualquiera de las dos funciones anteriores

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

Relación entre el mayor Number y el driver

- El módulo de un char device, para obtener del kernel un número mayor, desde la función module_init(), invocará alguna de las siguientes funciones del kernel Definidas en *linux/fs.h*
  - Reserva un rango de major numbers.

```
int register_chrdev_region (dev_t first,
    unsigned int count, char *name);
```

  - Si conocemos exactamente el major number a utilizar:

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

- Cuando es removido, desde la función module_exit (), devolverá al kernel el o los Major Number obtenido por medio de cualquiera de las dos funciones anteriores

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

Relación entre el mayor Number y el driver

- El módulo de un char device, para obtener del kernel un número mayor, desde la función module_init(), invocará alguna de las siguientes funciones del kernel Definidas en *linux/fs.h*
  - Reserva un rango de major numbers.

```
int register_chrdev_region (dev_t first,
    unsigned int count, char *name);
```

  - Si conocemos exactamente el major number a utilizar:

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

- Cuando es removido, desde la función module_exit (), devolverá al kernel el o los Major Number obtenido por medio de cualquiera de las dos funciones anteriores

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

Relación entre el mayor Number y el driver

- El módulo de un char device, para obtener del kernel un número mayor, desde la función module_init(), invocará alguna de las siguientes funciones del kernel Definidas en *linux/fs.h*
  - Reserva un rango de major numbers.

```
int register_chrdev_region (dev_t first,
    unsigned int count, char *name);
```

  - Si conocemos exactamente el major number a utilizar:

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

- Cuando es removido, desde la función module_exit (), devolverá al kernel el o los Major Number obtenido por medio de cualquiera de las dos funciones anteriores

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

Relación entre el mayor Number y el driver

- El módulo de un char device, para obtener del kernel un número mayor, desde la función module_init(), invocará alguna de las siguientes funciones del kernel Definidas en *linux/fs.h*
  - Reserva un rango de major numbers.

```
int register_chrdev_region (dev_t first,
    unsigned int count, char *name);
```

  - Si conocemos exactamente el major number a utilizar:

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

- Cuando es removido, desde la función module_exit (), devolverá al kernel el o los Major Number obtenido por medio de cualquiera de las dos funciones anteriores

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

Good save File Operations

- Una vez asignado el número mayor, el driver es identificable.
- Solo resta declarar sus funcionalidades al kernel junto con la referencia a cada una para que éste las pueda invocar cada vez que una aplicación requiera uso de funciones de este driver.
- Para ello se define una estructura del kernel denominada *File Operations*.
- Esta estructura es conjunto de punteros a función
- Para cada uan de las clases de device drivers el kernel define los métodos disponibles, y luego cada mulo los instancia con el nombre que le parezca mas conveniente, exportando al kernel la referencia al punto de entrada del método, es decir, un puntero a la función que lo implementa.

Good save File Operations

- Una vez asignado el número mayor, el driver es identificable.
- Solo resta declarar sus funcionalidades al kernel junto con la referencia a cada una para que éste las pueda invocar cada vez que una aplicación requiera uso de funciones de este driver.
- Para ello se define una estructura del kernel denominada *File Operations*.
- Esta estructura es conjunto de punteros a función
- Para cada uan de las clases de device drivers el kernel define los métodos disponibles, y luego cada mulo los instancia con el nombre que le parezca mas conveniente, exportando al kernel la referencia al punto de entrada del método, es decir, un puntero a la función que lo implementa.

Good save File Operations

- Una vez asignado el número mayor, el driver es identificable.
- Solo resta declarar sus funcionalidades al kernel junto con la referencia a cada una para que éste las pueda invocar cada vez que una aplicación requiera uso de funciones de este driver.
- Para ello se define una estructura del kernel denominada *File Operations*.
- Esta estructura es conjunto de punteros a función
- Para cada uan de las clases de device drivers el kernel define los métodos disponibles, y luego cada mulo los instancia con el nombre que le parezca mas conveniente, exportando al kernel la referencia al punto de entrada del método, es decir, un puntero a la función que lo implementa.

Good save File Operations

- Una vez asignado el número mayor, el driver es identificable.
- Solo resta declarar sus funcionalidades al kernel junto con la referencia a cada una para que éste las pueda invocar cada vez que una aplicación requiera uso de funciones de este driver.
- Para ello se define una estructura del kernel denominada *File Operations*.
- Esta estructura es conjunto de punteros a función
- Para cada uan de las clases de device drivers el kernel define los métodos disponibles, y luego cada mulo los instancia con el nombre que le parezca mas conveniente, exportando al kernel la referencia al punto de entrada del método, es decir, un puntero a la función que lo implementa.

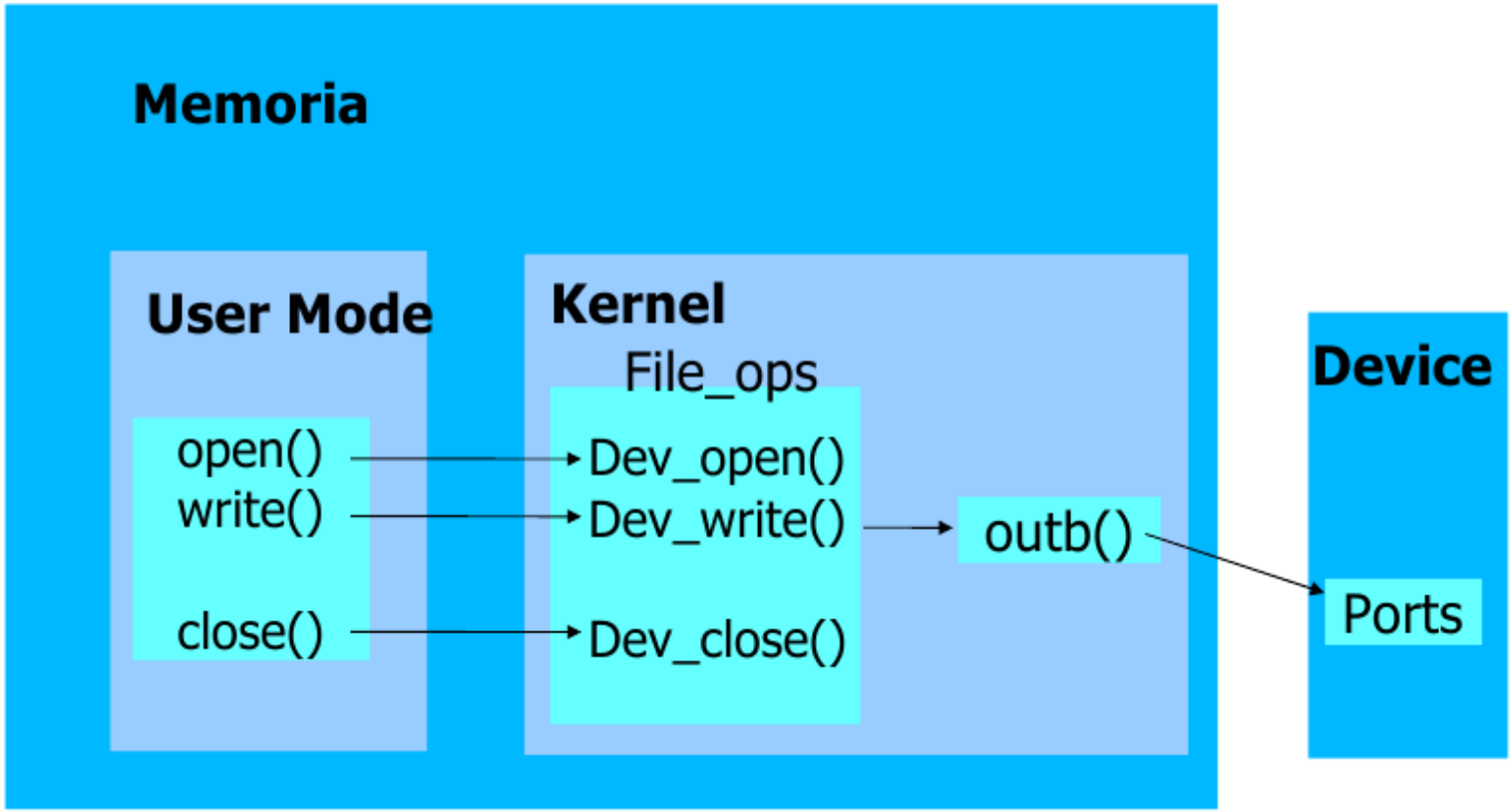
Good save File Operations

- Una vez asignado el número mayor, el driver es identificable.
- Solo resta declarar sus funcionalidades al kernel junto con la referencia a cada una para que éste las pueda invocar cada vez que una aplicación requiera uso de funciones de este driver.
- Para ello se define una estructura del kernel denominada *File Operations*.
- Esta estructura es conjunto de punteros a función
- Para cada uan de las clases de device drivers el kernel define los métodos disponibles, y luego cada mulo los instancia con el nombre que le parezca mas conveniente, exportando al kernel la referencia al punto de entrada del método, es decir, un puntero a la función que lo implementa.

Good save File Operations

- Una vez asignado el número mayor, el driver es identificable.
- Solo resta declarar sus funcionalidades al kernel junto con la referencia a cada una para que éste las pueda invocar cada vez que una aplicación requiera uso de funciones de este driver.
- Para ello se define una estructura del kernel denominada *File Operations*.
- Esta estructura es conjunto de punteros a función
- Para cada uan de las clases de device drivers el kernel define los métodos disponibles, y luego cada mulo los instancia con el nombre que le parezca mas conveniente, exportando al kernel la referencia al punto de entrada del método, es decir, un puntero a la función que lo implementa.

Diagrama de llamadas.



Métodos de File Operations

struct module *owner Es el primer campo de file_operations.
No es en sí mismo una operación sino un puntero al módulo “dueño“ de la estructura.
Se usa para evitar que el módulo sea cargado mientras sus operaciones están en uso.
A menudo se lo inicializa sencillamente con la macro THIS_MODULE, definida en linux/module.h.

Métodos de File Operations

loff_t (*llseek) (struct file *, loff_t, int); El método llseek se usa para cambiar la posición actual de lectura/escritura en un archivo. La nueva posición se retorna como un valor positivo.
loff_t es un “long offset” y tiene al menos un ancho de 64 bits aún en plataformas de 32-bit.
Si se produce algún error en su ejecución retorna un valor negativo. Si este puntero se inicializa en NULL en file_operations, seek () modificará el contador de posición en la estructura

Métodos de File Operations

ssize_t (*read) (struct file *, char __user *, size_t, loff_t *); Lee datos desde un archivo o device. Un puntero NULL en esta posición hace que la system call read () sobre este device devuelva -EINVAL (“Invalid argument”). Un valor de retorno no negativo representa el número de bytes leídos.

Métodos de File Operations

ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);

Envía datos a un archivo o device. Si este puntero es NULL, la system call write () retorna -EINVAL al programa que la invoca Un valor de retorno, no negativo, es el número de bytes escritos.

Métodos de File Operations

ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);

Inicia una lectura asincrónica (puede no completarse antes de retornar). Si es NULL, todas las operaciones serán ejecutadas en forma sincrónica por read ().

ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);

Inicia una operación de escritura asincrónica sobre el device.

Métodos de File Operations

`int (*readdir) (struct file *, void *, filldir_t);` Se usa para leer directorios. Solo lo usan los file systems. Debe ser NULL para cualquier device.

Métodos de File Operations

unsigned int (*poll) (struct file *, struct poll_table_struct *); El método poll es el back end de tres system calls: poll (), epoll (), y select ().
Se usa para saber si un read () o un write () a uno o mas descriptores de archivo va a bloquear.
El método poll () debe retornar una máscara de bits que indica si son factibles lecturas o escrituras no bloqueantes.
El kernel con esta información pone un proceso en estado sleeping hasta que sea posible la operación de E/S.
Si un driver deja NULL este método, se asume que puede ser leído o escrito sin bloqueo.

Métodos de File Operations

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

La system call ioctl () envía comandos device específicos. El kernel generalmente procesa ioctl () por medio del método definido en file_operations. Si no hay un method ioctl (), la system call retorna error para cualquier requerimiento no predefinido (-ENOTTY, “No such ioctl for device”).

Métodos de File Operations

(*mmap) (struct file *, struct vm_area_struct *); mmap requiere el mapeo de un device de memoria al espacio de direcciones del proceso.
Si este método es NULL, la system call mmap () retorna -ENODEV.

Métodos de File Operations

`int (*open) (struct inode *, struct file *);` Como siempre, es la primer operación realizada sobre el archivo o device, no es necesario declararlo. Si es NULL, el device siempre se abre, pero no se notifica al driver.

Métodos de File Operations

int (*flush) (struct file *); La operación flush () se invoca cuando un proceso cierra su copia del file descriptor de un device.
Ejecuta (y espera por) cualquier operación excepcional sobre el device.
No confundir con la operación fsync () requerida por un programa.
flush () se usa en muy pocos drivers: el driver SCSI de cinta lo use, por ejemplo, para asegurar que todos los datos escritos estén en la cinta antes de cerrar el dispositivo.
Si es NULL, el kernel simplemente ignora el requerimiento.

Métodos de File Operations

int (*release) (struct inode *, struct file *); Se invoca cuando se desea liberar la estructura.
Igual que open () puede ser NULL.
release () no se invoca cada vez que un proceso llama a close (). Si una estructura file se comparte (como resultado de fork () o dup()), release () se invoca cuando todas las copias ejecutan close ().

Métodos de File Operations

`int (*fsync) (struct file *, struct dentry *, int);` Es el back end de la system call fsync (), que es llamada por un programa para flushear cualquier dato pendiente. Si es NULL, retorna -EINVAL.

`int (*aio_fsync)(struct kiocb *, int);` Es la versión asincrónica del método fsync.

`int (*fasync) (int, struct file *, int);` Se usa para notificar al device que cambió su flag FASYNC. Puede ser NULL si el driver no soporta notificación asincrónica.

`int (*lock) (struct file *, int, struct file_lock *);` Se usa para implementar file locking.
Es indispensable en archivos, pero rara vez se usa en drivers.

Métodos de File Operations

ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *

ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t

Implementan operaciones de lectura escritura fragmentada, que ocasionalmente necesitan involucrar múltiples áreas de memoria.
Estas system calls fuerzan operaciones extra de copia sobre los datos.
Si estos punteros se dejan NULL, se llaman en su lugar los métodos read () y write () (quizá mas de una vez).

Métodos de File Operations

ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);

Implementa el lado read de la system call sendfile (), que mueve los datos desde un file descriptor hacia otro con mínima copia.
Se usa por ejemplo en un web server que necesita enviar los contenidos de un archivo fuera hacia la red. Los device drivers normalmente la dejan en NULL.

Métodos de File Operations

ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);

Implementa el lado read de la system call sendfile (), que mueve los datos desde un file descriptor hacia otro con mínima copia.
Se usa por ejemplo en un web server que necesita enviar los contenidos de un archivo fuera hacia la red. Los device drivers normalmente la dejan en NULL.

Métodos de File Operations

ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);

sendpage es la otra mitad de sendfile. El kernel la llama para enviar datos al archivo correspondiente, una página a la vez. Los device drivers normalmente no implementan sendpage.

unsigned long (*get_unmapped_area) (struct file *, unsigned long, uns

El objetivo de este método es encontrar una ubicación adecuada en el espacio de direcciones del proceso para mapearla sobre un segmento de memoria del device. Normalmente es el código de manejo de la memoria quien realiza esta tarea Este método permite a los drivers forzar los requerimientos de alineamiento que pueda tener cualquier device. La mayoría de los drivers dejan este método NULL.

Métodos de File Operations

int (*check_flags)(int) Permite al módulo chequear los flags que se le pasan en una llamada fcntl (F_SETFL...).

int (*dir_notify)(struct file *, unsigned long); Se invoca cuando una aplicación usa fcntl () para pedir modificaciones en un directorio.
Sólo es útil en file systems.
Los drivers no necesitan implementar dir_notify.

Declaración de File Operations en un char device

- Dentro del código del módulo en el que se va a implementar el char device debemos declarar la estructura file operations, que para el caso de un char device solo recesita cinco métodos

```
struct file_operations  midriver_fops =
{
    .owner = THIS_MODULE,
    .read = my_read,
    .write = my_write,
    .ioctl = my_ioctl,
    .open = my_open,
    .release = my_release,
};
```

- Los métodos comienzan siempre con el caracter '.', y el nombre propio de cada función está del otro lado de la igualdad.

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode; //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)*
  - *loff_t f_pos; //Puntero de 64 bits offset dentro del archivo*
  - *unsigned int f_flags; //O_RDONLY, O_NONBLOCK, O_SYNC.*
  - *struct file_operations *f_op;*
  - *void *private_data;*

    open () la carga con NULL antes de llamar al método open propio del driver.

    Se puede utilizar para guardar datos propios del driver
  - *struct dentry *f_dentry;*

    Directory entry. Normalmente no es necesario tenerla en cuenta salvo si necesitan acceder al inodo del d

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC
  - *struct file_operations *f_op;*
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.
    Se puede utilizar para guardar datos propios del driver.
  - *struct dentry *f_dentry;*
    Directory entry. Normalmente no es necesario tocarlo. Importante para sistemas de archivos virtuales.

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC
  - *struct file_operations *f_op;*
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.
    Se puede utilizar para guardar datos propios del driver.
  - *struct dentry *f_dentry;*
    Directory entry. Normalmente no es necesario tocarlo. Contiene información acerca del archivo físico.

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC.
  - *struct file_operations *f_op*;
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.
    Se puede utilizar para guardar datos propios del driver
  - *struct dentry *f_dentry;*
    Directory entry. Normalmente no es necesario tenerla en cuenta, salvo si necesitan acceder al inodo del directorio

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC.
  - *struct file_operations *f_op*;
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.
    Se puede utilizar para guardar datos propios del driver
  - *struct dentry *f_dentry;*
    Directory entry. Normalmente no es necesario tenerla en cuenta, salvo si necesitan acceder al inodo del directorio

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC.
  - *struct file_operations *f_op*;
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.
    Se puede utilizar para guardar datos propios del driver
  - *struct dentry *f_dentry;*
    Directory entry. Normalmente no es necesario tenerla en cuenta, salvo si necesitan acceder al inodo del directorio

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC.
  - *struct file_operations *f_op*;
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.
    Se puede utilizar para guardar datos propios del driver
  - *struct dentry *f_dentry;*
    Directory entry. Normalmente no es necesario tenerla en cuenta, salvo si necesitan acceder al inodo del directorio

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC.
  - *struct file_operations *f_op*;
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.
    Se puede utilizar para guardar datos propios del driver
  - *struct dentry *f_dentry;*
    Directory entry. Normalmente no es necesario tenerla en cuenta, salvo si necesitan acceder al inodo del directorio

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC.
  - *struct file_operations *f_op*;
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.

    Se puede utilizar para guardar datos propios del driver
  - *struct dentry *f_dentry;*
    Directory entry. Normalmente no es necesario tenerla en cuenta, salvo si necesitan acceder al inodo del directorio

Estructuras auxiliares: File

- Definida en `linux/fs.h`.
- Contiene la información lógica de un archivo abierto con open ().
- Campos de interés para un char device:
  - *mode_t f_mode;* //Modo en que se abrió el archivo (FMODE_READ, FMODE_WRITE)
  - *loff_t f_pos;* //Puntero de 64 bits offset dentro del archivo
  - *unsigned int f_flags;* //O_RDONLY, O_NONBLOCK, O_SYNC.
  - *struct file_operations *f_op*;
  - *void *private_data;*
    open () la carga con NULL antes de llamar al método open propio del driver.

    Se puede utilizar para guardar datos propios del driver
  - *struct dentry *f_dentry;*

    Directory entry. Normalmente no es necesario tenerla en cuenta, salvo si necesitan acceder al inodo del directorio.

Estructuras auxiliares: i-node

- Definida en `linux/fs.h`.
- Contiene la información de un nodo del file system (no de un archivo abierto).
- Campos de interés para un char device:
  - *dev_t i_rdev*: contiene el número de device (32 bits: 12 major number, 20 minor number).
  - *struct cdev *i_cdev*: es una estructura del LKM que representa a un char device. Si el inodo no contiene un char device este campo es NULL.
  - Para obtener el major y el minor number a partir de *inode*

Estructuras auxiliares: i-node

- Definida en `linux/fs.h`.
- Contiene la información de un nodo del file system (no de un archivo abierto).
- Campos de interés para un char device:
  - *dev_t i_rdev*: contiene el número de device (32 bits: 12 major number, 20 minor number)
  - *struct cdev *i_cdev*: es una estructura del LKM que representa a un char device. Si el inodo no contiene un char device este campo es NULL
  - Para obtener el major y el minor number a partir de inode: `unsigned int iminor(struct inode *inode);` `unsigned int imajor(struct inode *inode);`

Estructuras auxiliares: i-node

- Definida en `linux/fs.h`.
- Contiene la información de un nodo del file system (no de un archivo abierto).
- Campos de interés para un char device:
  - *dev_t i_rdev;* contiene el número de device (32 bits: 12 major number 20 minor number)
  - *struct cdev *i_cdev;* es una estructura interna del VFS que representa a un char device. Si el inodo no contiene un char device este campo es NULL
  - Para obtener el major y el minor number a partir de *dev_t*: *unsigned int iminor(struct inode *inode);* *unsigned int imajor(struct inode *inode);*

Estructuras auxiliares: i-node

- Definida en `linux/fs.h`.
- Contiene la información de un nodo del file system (no de un archivo abierto).
- Campos de interés para un char device:
  - *dev_t i_rdev;*//contiene el número de device (32 bits: 12 major number 20 minor number)
  - *struct cdev *i_cdev;* //es una estructura del LDM que representa a un char device. Si el inodo no contiene un char device este campo es NULL.
  - Para obtener el major y el minor number a partir de inode

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

Estructuras auxiliares: i-node

- Definida en `linux/fs.h`.
- Contiene la información de un nodo del file system (no de un archivo abierto).
- Campos de interés para un char device:
  - *dev_t i_rdev;*//contiene el número de device (32 bits: 12 major number 20 minor number)
  - *struct cdev *i_cdev;* //es una estructura del LDM que representa a un char device. Si el inodo no contiene un char device este campo es NULL.
  - Para obtener el major y el minor number a partir de inode

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

Estructuras auxiliares: i-node

- Definida en `linux/fs.h`.
- Contiene la información de un nodo del file system (no de un archivo abierto).
- Campos de interés para un char device:
  - *dev_t i_rdev;*//contiene el número de device (32 bits: 12 major number 20 minor number)
  - *struct cdev *i_cdev;* //es una estructura del LDM que representa a un char device. Si el inodo no contiene un char device este campo es NULL.
  - Para obtener el major y el minor number a partir de inode

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

Estructuras auxiliares: i-node

- Definida en `linux/fs.h`.
- Contiene la información de un nodo del file system (no de un archivo abierto).
- Campos de interés para un char device:
  - *dev_t i_rdev;*//contiene el número de device (32 bits: 12 major number 20 minor number)
  - *struct cdev *i_cdev;* //es una estructura del LDM que representa a un char device. Si el inodo no contiene un char device este campo es NULL.
  - Para obtener el major y el minor number a partir de inode

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

Manejo de Interrupciones!

*int request_irq (unsigned int irq, void (*handler) (int, void *, struct pt_regs *), unsigned long flags, const char *dev_name, void *dev_id);*

Retorna 0 o un código negativo de error. (-EBUSY si otro driver está usando la IRQ pedida por ejemplo).
Argumentos:

- *unsigned int irq*: Número de la IRQ requerida.
- *void (*handler) (int, void *, struct pt_regs *)*: Puntero a la función que se desea instalar como handler.

Manejo de Interrupciones!

- *unsigned long flags*: Máscara de bits relacionada al manejo de interrupciones.
  1. *SA_INTERRUPT*, Indica que es un “fast“ interrupt handler: Se ejecutará aún con las interupciones deshabilitadas
  2. *SA_SHIRQ*: Indica que la interrupción puede compartirse entre varios devices.
- *const char *dev_name*: Nombre del device en /dev. Lo usa en /proc/interrupts para mostrar el dueño de la IRQ.
- *void *dev_id*: Puntero usado para compartir IRQ’s. Cuando no se comparte IRQ se lo deja en NULL

*void free_irq (unsigned int irq, void *dev_id);*