Character device drivers

Mg. Ing. Gonzalo E. Sanchez
MSE - 2022

Implementación de Manejadores de Dispositivos

Char device drivers

- Introducción
- Concepto de major & minor
- Operaciones sobre file devices
- Asignar y registrar un Char Device
- Operaciones de escritura

Introducción

Introducción

- Un char device transfiere datos desde o hacia una aplicación en forma de caracteres.
- Lo hace como un stream de datos, como un puerto serie.
- Un char device driver expone las funcionalidades y propiedades de un dispositivo mediante un archivo especial en **/dev**.
- Este archivo especial se utiliza para intercambiar información entre la aplicación y el dispositivo.
- Utiliza el concepto básico de linux en donde todo es un archivo.

Introducción

- Los char device drivers representan los drivers más básicos que posee las fuentes del kernel.
- Podemos ver la estructura de este tipo de drivers en **include/linux/cdev.h**
- Se rigen por la estructura **struct cdev**.

Introducción

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

Introducción

- Los char devices se manejan como si fueran archivos.
- Por esto utilizan operaciones básicas definidas en la estructura **struct file_operations**:
  - Read
  - Write
  - Select
  - Open
  - Close

Concepto de major & minor

Major & Minor

- Como se mencionó antes, los char devices al registrarse van poblando el directorio **/dev**.
- Puede determinarse un archivo correspondiente a un char device utilizando el comando **ls -l** en el directorio **/dev**.
- Los char devices siempre tienen una letra **c** delante.
- Los números **major** y **minor** identifican y vinculan los dispositivos con sus respectivos drivers.
- Observar que estos solo aparecen en archivos asociados a un char device o dispositivos block.

Major & Minor

- La primer letra de los archivos dentro de **/dev** puede ser:
  - **c**: Archivos de Char devices.
  - **b**: Archivo de block device.
  - **l**: Link simbólico.
  - **d**: Directorios.
  - **s**: Sockets.
  - **p**: Pipe con nombre.

Major & Minor

- Para tipos de archivo **b** y **c**, la quinta y sexta columna tienen un formato **<X, Y>**.
- **X** representa el número major.
- **Y** representa el número menor.
- El kernel almacena los números que identifican un dispositivo en variables tipo **dev_t** que no son más que **u32**.
- El número **major** se representa con 12 bits.
- El número **minor** se representa con los 20 bits restantes.

Major & Minor

- En alguna ocasión podría ser necesario extraer los números **major** y **minor** de una variable **dev_t**.
- El kernel proporciona dos macros para esto:
  - MAJOR(dev_t dev);
  - MINOR(dev_t dev);
- Por otro lado, para pasar los números a una variable tipo **dev_t**:
  - MKDEV(int major, int minor);

Major & Minor

- El número **major** podría verse como un offset en la tabla de device drivers.
- Quiere decir que varios dispositivos utilizando el mismo driver tendrán el mismo número **major**.
- No es una regla, pero se trata de tener un número **major** por driver.
- El número **minor** puede ser utilizado como un índice de vector para los dispositivos manejados por un driver.

Major & Minor

- Existen dos maneras de asignar estos números:
- Estáticamente: Consiste en adivinar (?) un número **major** que todavía no fue utilizado. Se utiliza la función **register_chrdev_region()**.

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

- Retorna 0 si es exitosa, o un código de error negativo en caso contrario.

Major & Minor

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

- El argumento **first** debe construirse con la macro **MKDEV(major,minor)**.
- El argumento **count** es la cantidad de dispositivos consecutivos necesarios a registrar.
- El argumento **name** debería ser el nombre del dispositivo asociado o driver.

Major & Minor

- Dinámicamente: El número es asignado por el kernel utilizando la función **alloc_chrdev_region().**

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

- Retorna 0 si es exitosa, o un código de error negativo en caso contrario.
- Es la forma recomendada de obtener un valor **dev_t**.

Major & Minor

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

- El argumento ***dev** es el valor de retorno del número asignado.
- **firstminor** es el primer numero **minor** del rango que se desea registrar.
- **count** es la cantidad de dispositivos consecutivos necesarios a registrar.
- El argumento **name** debería ser el nombre del dispositivo asociado o driver.

Major & Minor

- En términos de funcionalidad no hay diferencia.
- La forma estática era utilizada tiempo atrás, pero no es lo que se utiliza hoy en día para obtener números **major** y **minor.**
- La utilizaremos de forma didáctica, y luego pasaremos al método dinámico.
- De todas maneras, lo usual es que estas funciones no sean llamadas directamente desde el driver.
- Son enmascaradas por el framework que utiliza el driver.

Major & Minor

- Antes de linux 2.6.32 los device files tenían que crearse de modo manual mediante **mknod**.
- Corresponde a la forma estática (registrar) de obtener los números **major** y **minor**.
- La coherencia entre los dispositivos y los device files era librada al desarrollador del sistema.
- Luego del release estable del kernel 2.6 se incorporó un nuevo filesystem virtual **sysfs** que se monta en **/sys**.

Major & Minor

- El trabajo de **sysfs** es exportar una vista de la configuración de hardware del sistema al espacio usuario.
- Los drivers que se compilan con el kernel directamente registran sus objetos con **sysfs** cuando son detectados.
- Driver compilados como módulos se registran en **sysfs** cuando el módulo se carga (**insmod**).

Major & Minor

- Una vez montado **sysfs** en **/sys** lo registrado por los drivers se hace disponible para los procesos en espacio usuario.
- El kernel utiliza **sysfs** para exportar nodos de dispositivos para ser utilizados por **udev**.
- **udev** es el device manager de linux.
- Administra los dispositivos presentes en el directorio **/dev**.

Major & Minor

- Los device files son creados por el kernel a través del filesystem **devtmpfs**.
- Cualquier driver que quiera registrar un nodo (device node) debe pasar por **devtmpfs**.
- Esta es la manera dinámica para crear los device files.
- Para que **devtmpfs** cree un nodo se necesita una entrada en **/sys/class**.
- Esta es una forma de agrupar los drivers según clases (network, gpio, i2c-device, etc).

Operaciones sobre file devices

Operaciones sobre file devices

- Las operaciones sobre los file devices dependen de los drivers que manejan esos archivos.
- Estas operaciones (las posibles) están determinadas por la estructura **struct file_operations** en el kernel.
- Esta estructura expone en espacio usuario un set de callbacks que terminan invocando system calls sobre un archivo.
- La estructura debe ser poblada con los callbacks.
- Si se desea que esté al alcance del usuario la función **write()** hay que implementarla en el driver.

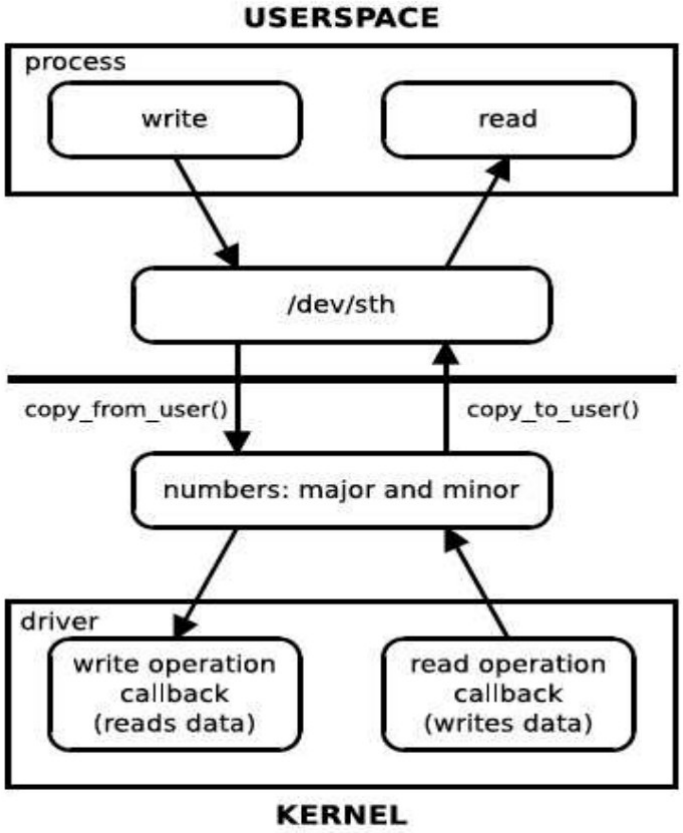
Operaciones sobre file devices

- Todos los campos de la estructura **file_operations** pueden verse en **include/linux/fs.h**.
- Implementaremos solo las más básicas:
  - Open
  - Read
  - Write
  - Release (se invoca con **close()**).
  - Llseek (invocada mediante **lseek**()).
  - Ioctrl

Operaciones sobre file devices

- Un programa en espacio usuario puede hacer una system call relacionada a archivos.
- El kernel busca el driver responsable de ello, en especial el que creó el archivo.
- Al localizar la estructura **file_operations** correspondiente pueden suceder dos cosas:
  - Que la función requerida esté implementada, entonces solo se corre.
  - Que la función no esté implementada. Se devuelve un código de error acorde a la system call.

Operaciones sobre file devices



Operaciones sobre file devices

- Existe un “inconveniente” al querer intercambiar información entre el espacio usuario y el espacio kernel.
- Recordar que el espacio kernel no es accesible al espacio usuario (se requiere de privilegios).
- Asimismo, el espacio usuario (para el kernel) no es una fuente de confianza.

Operaciones sobre file devices

- No está permitido para el kernel copiar directamente datos de espacio usuario mediante memcpy() por algunas razones:
  - No funciona en ciertas arquitecturas.
  - Si la dirección que otorga la aplicación es inválida, produce un segfault.
- El espacio usuario no es de confianza porque una aplicación maliciosa podría pasar una dirección de memoria de kernel.
- Así podría reemplazarse datos desde el dispositivo (**read()**) o hacer un dump hacia el dispositivo (**write()**).

Operaciones sobre file devices

- Para mantener el kernel seguro y portable el driver debe utilizar funciones especiales para intercambiar datos.
- En el caso de un solo valor:
  - get_user(value, address): la variable en espacio kernel **value** se actualiza con el contenido de la dirección apuntada **address**.
  - put_user(value, address): la dirección de espacio usuario apuntada **address** se carga con el valor **value**.

Operaciones sobre file devices

- En el caso de un buffer:

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

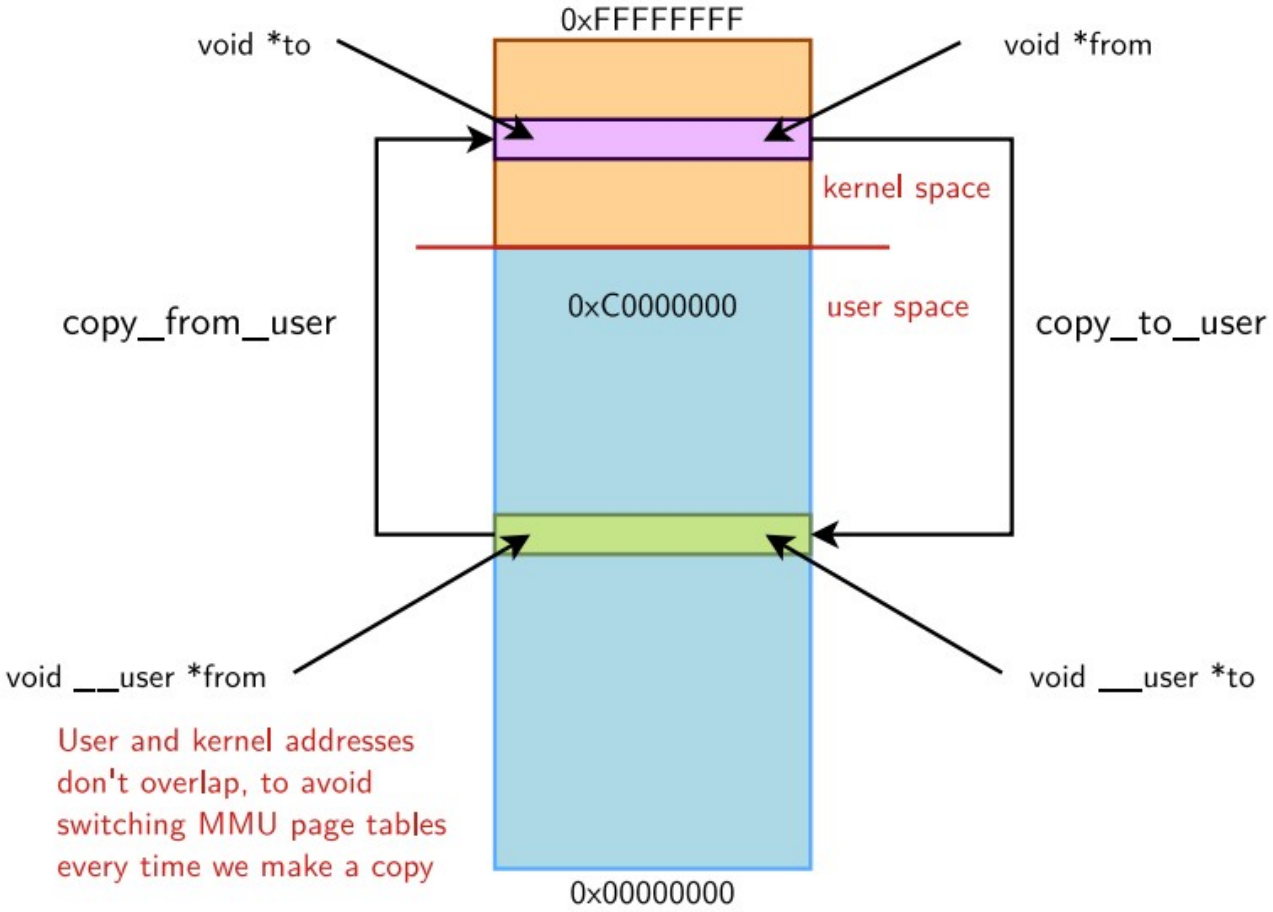
```
unsigned long copy_from_user(void *to, const void __user
*from,
                             unsigned long n);
```

- Cuando el puntero es a una dirección de espacio usuario se utiliza el macro **__user**.
- Es utilizado por **sparse** para indicar al desarrollador que está accediendo a memoria no confiable.

Operaciones sobre file devices

- El valor del argumento **n** indica la cantidad de bytes a copiar.
- El resultado de estas funciones debe ser chequeado.
- Retorno de la función:
  - Cero en caso de éxito.
  - La cantidad de bytes que no pudieron ser copiados, en caso de fallar.
- **NOTA:** si **copy_to_user()** no puede copiar algunos datos, efectúa un padding con ceros para lograr los **n** bytes en destino.

Operaciones sobre file devices



Operaciones sobre file devices

- **Método open()**
- Invocado cada vez que una aplicación quiere abrir un file device.
- Usualmente utilizado para hacer inicializaciones varias

```
int (*open)(struct inode *inode, struct file *filp);
```

- Por cada open() se le dará a la función callback un **inode** como parámetro.
- **inode** es la representación a bajo nivel del kernel para un archivo.

Operaciones sobre file devices

- **Método release()**
- Invocado al cerrar el archivo mediante función **close()**.
- Se debe deshacer todo lo que se hizo al ejecutar **open()**.

```
int (*release)(struct inode *inode, struct file *filp);
```

- Normalmente se libera la memoria asignada en **open()**.
- Se apaga el dispositivo (si está soportado) y se descartan buffers.

Operaciones sobre file devices

- **Método write()**
- Utilizado para enviar datos al dispositivo.
- Cuando una aplicación llama al método **write()** sobre el file device, se invoca la implementación de kernel.

```
ssize_t(*write)(struct file *filp, const char __user
*buf,
                                size_t count, loff_t *pos);
```

Operaciones sobre file devices

- El valor de retorno es la cantidad de bytes escritos.
- El argumento ***buf** representa el buffer de datos proveniente del espacio usuario.
- **count** es el tamaño solicitado de la transferencia.
- ***pos** indica la posición inicial a partir de la que deben ser escritos los datos en el archivo (dispositivo).

Operaciones sobre file devices

- Pasos para efectuar una operación **write** (*Linux device driver development, página 101*):
  - Chequear las solicitudes de espacio usuario (bad or invalid - solo para dispositivos que exponen su memoria).
  - Ajustar **count** en el caso que se extienda más allá del tamaño del archivo (memoria disponible en el dispositivo).
  - Adecuar la posición en memoria a partir de la cual se escribe (no necesariamente comienza en 0x00 en el dispositivo)
  - Copiar los datos desde el espacio usuario a espacio kernel de forma adecuada.

Operaciones sobre file devices

- Pasos para efectuar una operación **write** (continuación):
  - Incrementar la posición del cursor en el archivo según la cantidad de bytes escritos.
  - Retornar la cantidad de bytes escritos.
- Estos pasos no son universales, sólo dan una guia.
- Consultar el citado libro para un ejemplo completo con código (muy útil para la propia implementación).

Operaciones sobre file devices

- **Método read()**
- Utilizado para recibir datos desde el dispositivo.
- Cuando una aplicación llama al método **read()** sobre el file device, se invoca la implementación de kernel.

```
ssize_t(*read)(struct file *filp, char __user *buf,
                          size_t count, loff_t *pos);
```

Operaciones sobre file devices

- El valor de retorno es la cantidad de bytes leídos.
- El argumento ***buf** representa el buffer de datos del espacio usuario, recipiente de los datos leídos.
- **count** es el tamaño solicitado de la transferencia (tamaño del buffer en espacio usuario).
- ***pos** indica la posición inicial a partir de la que deben ser leídos los datos en el archivo (dispositivo).

Operaciones sobre file devices

- Pasos para efectuar una operación **read** (*Linux device driver development, página 104*):
  - Prevenir la lectura más allá del tamaño del file device, retornando EOF en ese caso.
  - Ajustar **count** en el caso que se extienda más allá del tamaño del archivo (memoria disponible en el dispositivo).
  - Adecuar la posición en memoria a partir de la cual se lee (no necesariamente comienza en 0x00 en el dispositivo)
  - Copiar los datos en el espacio usuario de forma adecuada (**copy_to_user()**).

Operaciones sobre file devices

- Pasos para efectuar una operación **read** (continuación):
  - Incrementar la posición del cursor en el archivo según la cantidad de bytes leídos.
  - Retornar la cantidad de bytes leídos.
- Al igual que en el caso de **write**, estos pasos no son universales, sólo dan una guia.
- Consultar el citado libro para un ejemplo completo con código (muy útil para la propia implementación).

Operaciones sobre file devices

- **Método llseek()**
- Utilizado para mover el cursor de posición sobre un archivo.
- Cuando una aplicación llama al método **lseek()** sobre el file device, se invoca la implementación de kernel.

```
loff_t(*llseek) (struct file *filp, loff_t offset,
                                    int whence);
```

Operaciones sobre file devices

- El valor de retorno es la nueva posición del cursor en el archivo.
- **offset** es un offset relativo a la posición actual en el archivo.
- **whence** define desde donde iniciar el desplazamiento. Los valores posibles son:
  - SEEK_SET: Posición relativa al inicio del archivo.
  - SEEK_CUR: Posición relativa al valor actual del cursor.
  - SEEK_END: Posición relativa al final del archivo.

Operaciones sobre file devices

- Pasos para efectuar una operación **llseek** (*Linux device driver development, página 106*):
  - Chequear todos los casos definidos por **whence** y ajustar la nueva posición acorde a estos estados.
  - Chequear si el valor de la nueva posición es válido.
  - Actualizar el valor del cursor del archivo con la nueva posición.
  - Retornar la nueva posición del cursor.

Operaciones sobre file devices

- Como en los casos anteriores, estos pasos no son universales, sólo dan una guía.
- Consultar el citado libro para un ejemplo completo con código (muy útil para la propia implementación).

Operaciones sobre file devices

HANDS ON

1. Descargar fuentes de práctica del campus virtual.

2. Seguir la guía de práctica II Character device drivers

Operaciones sobre file devices

HANDS ON

1. Seguir el laboratorio 4.2 de la bibliografía “Linux driver development for embedded processors”.
   Páginas 105-109

Operaciones sobre file devices

HANDS ON

1. Seguir el laboratorio 4.3 de la bibliografía “Linux driver development for embedded processors”.
   Páginas 111-113

Gracias.