Taller de drivers

Sistemas Operativos

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

06 de Octubre de 2015

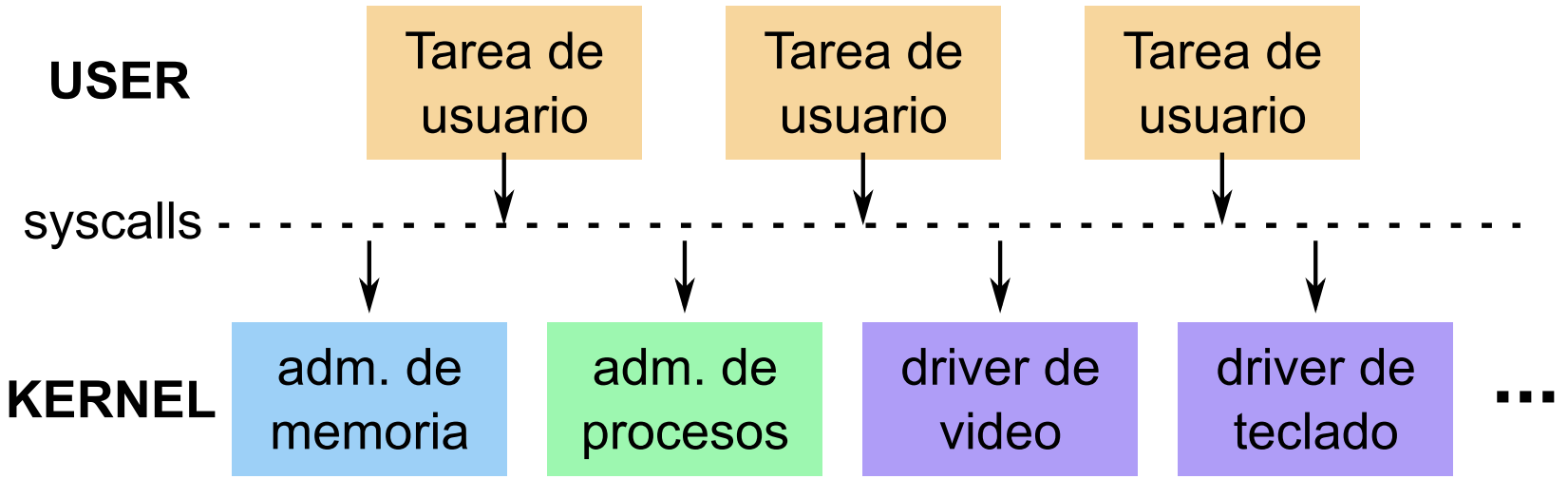
Agenda para hoy

Hoy nos vamos a concentrar en estudiar todo lo necesario para escribir y cargar correctamente un driver en el kernel, y luego vamos a escribir un driver nosotros mismos.

1. Breve repaso teórico
2. Módulos
   - escritura de módulos
   - carga y liberación de módulos
3. Devices
   - tipos de devices
   - device numbers
   - operaciones de *char* devices
   - la estructura `cdev`
   - reserva de nodos
   - memoria dinámica
   - sincronización
4. Taller

La estructura de Linux

En **linux** hay sólo **dos** espacios: user y kernel.



- Kernel **monolítico**: administradores, memoria, drivers, todo junto.
- **El** kernel tiene acceso a todo:
  - ejecuta en un **único** espacio de memoria
  - ejecuta en el **máximo** nivel de privilegio

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video
- Decenas de drivers de otros dispositivos físicamente instalados

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video
- Decenas de drivers de otros dispositivos físicamente instalados

Situación: Enchufo un pendrive USB. Necesita un driver nuevo.

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video
- Decenas de drivers de otros dispositivos físicamente instalados

Situación: Enchufo un pendrive USB. Necesita un driver nuevo.

- ¿Tengo que reiniciar la máquina y cargar de nuevo el kernel?
- ¿Tengo que recompilar el kernel?

La estructura de Linux (3)

Si el kernel está todo contenido en **un** gran archivo binario:

- ¿Qué pasa si quiero agregar funcionalidad cuando ya estoy usando la máquina?
- ¿Qué pasa si incluyo funcionalidad "por las dudas"?

La estructura de Linux (3)

Si el kernel está todo contenido en **un** gran archivo binario:

- ¿Qué pasa si quiero agregar funcionalidad cuando ya estoy usando la máquina?
- ¿Qué pasa si incluyo funcionalidad "por las dudas"?

Solución: (de linux)
Linux soporta la carga y descarga de **módulos** al kernel en tiempo de ejecución.

Módulos

- ¿Qué cosas componen a un **módulo**?

Módulos

- ¿Qué cosas componen a un **módulo**?
  - Funciones relacionadas
  - Datos
  - Puntos de entrada y salida

Módulos

- ¿Qué cosas componen a un **módulo**?
  - Funciones relacionadas
  - Datos
  - Puntos de entrada y salida
- ¿A causa de qué podría ejecutarse el código de un módulo?

Módulos

- ¿Qué cosas componen a un **módulo**?
  - Funciones relacionadas
  - Datos
  - Puntos de entrada y salida

- ¿A causa de qué podría ejecutarse el código de un módulo?
  1. Llamada al sistema
  2. Atención de interrupción

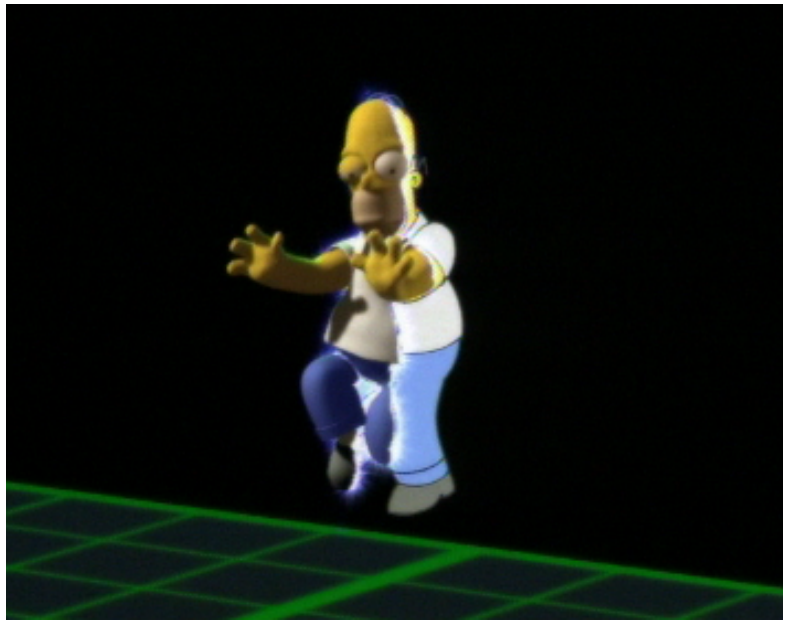
Módulos

- ¿Qué cosas componen a un **módulo**?
  - Funciones relacionadas
  - Datos
  - Puntos de entrada y salida
- ¿A causa de qué podría ejecutarse el código de un módulo?
  1. Llamada al sistema
  2. Atención de interrupción
- ¿Qué funcionalidades podría brindar un módulo?

Módulos

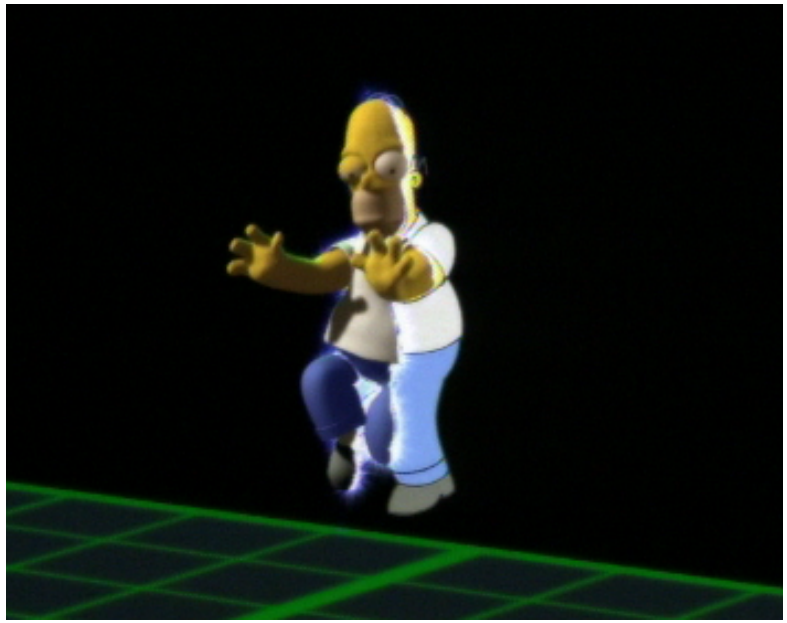
- ¿Qué cosas componen a un **módulo**?
  - Funciones relacionadas
  - Datos
  - Puntos de entrada y salida
- ¿A causa de qué podría ejecutarse el código de un módulo?
  1. Llamada al sistema
  2. Atención de interrupción
- ¿Qué funcionalidades podría brindar un módulo?
- Hoy vamos a escribir nuestro primer módulo...

Un nuevo mundo...



- Estamos ejecutando en el **nivel de máximo privilegio**
- El kernel no está enlazado a la `libc`
- Hacer **operaciones de punto flotante** es más complicado
- Tenemos un **stack fijo y limitado** (y tenemos que compartirlo con el resto del kernel)
- Hay varias fuentes de posibles **condiciones de carrera**

Un nuevo mundo...



- Estamos ejecutando en el **nivel de máximo privilegio**
- El kernel no está enlazado a la `libc`
- Hacer **operaciones de punto flotante** es más complicado
- Tenemos un **stack fijo y limitado** (y tenemos que compartirlo con el resto del kernel)
- Hay varias fuentes de posibles **condiciones de carrera**

¿Qué pasa si hacemos un **acceso indebido a memoria**?


DANGER

Nuestro primer módulo

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void) {
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_ALERT "Adios, mundo cruel...\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Juan de los Palotes");
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

Nuestro primer módulo (2)

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
```

- `init.h` contiene la definición de las macros `module_init()` y `module_exit()`
- `module.h` contiene varias definiciones necesarias para la gran mayoría de los módulos (por ejemplo, varios `MODULE_*`)
- `kernel.h` contiene la declaración de `printk()`

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Leandro Lera Romero");
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

- `MODULE_AUTHOR()` y `MODULE_DESCRIPTION()` son meramente informativos
- `MODULE_LICENSE()` indica la licencia del módulo;
  - algunos valores posibles son:
    - `GPL`
    - `Dual BSD/GPL`
    - `Proprietary`
  - un módulo con una licencia propietaria "mancha" el kernel

Nuestro primer módulo (4)

```
static int __init hello_init(void) {
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");
    return 0;
}

module_init(hello_init);
```

- `static` indica que la función es local al archivo (opcional)
- `__init` le indica al kernel que la función sólo se usará al momento de la inicialización, y que puede olvidarla una vez cargado el módulo (opcional)
- `printk()` se comporta de manera similar a la función `printf()` de la *libc*, pero permite indicar niveles de prioridad:
  - `KERN_ALERT` – problema de atención inmediata
  - `KERN_INFO` – mensaje con información
  - `KERN_DEBUG` – mensaje de *debug*

Nuestro primer módulo (5)

```
static int __init hello_init(void) {
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");
    return 0;
}

module_init(hello_init);
```

- Con `module_init()` se indica dónde encontrar la **función de inicialización** del módulo
- La función de inicialización es llamada:
  - al arrancar el sistema
  - al insertar el módulo
- Su rol es registar recursos, inicializar hardware, reservar espacio en memoria para estructuras de datos, etc.
- Si todo salió bien, tiene que devolver 0; si no, tiene que volver atrás lo que cambió y devolver algo distinto de cero.

Nuestro primer módulo (6)

```
static void __exit hello_exit(void) {
    printk(KERN_ALERT "Adios, mundo cruel...\n");
}

module_exit(hello_exit);
```

- Con `module_exit()` se indica dónde encontrar la **función de "limpieza"** del módulo
- La función de "limpieza" es llamada antes de quitar el módulo
- Se ocupa de deshacer/limpiar todo lo que la función de inicialización y el resto del módulo usaron

Inyectando módulos al kernel

¿Cómo cargamos nuestro módulo al kernel?

- `insmod` carga el código y los datos de nuestro módulo al kernel
- el kernel usa su tabla de símbolos para enlazar todas las referencias no resueltas del módulo
- una vez cargado, se llama a su función de inicialización
- `rmmod` permite quitar el módulo del kernel si esto es posible (por ejemplo, falla si el módulo está siendo usado)
- `modprobe` es una alternativa más inteligente que `insmod` y `rmmod` (tiene en cuenta dependencias entre módulos)

Inyectando módulos al kernel (2)

1. Necesitamos
   - `make`
   - `module-init-tools`
   - `linux-headers-<version>`
     (`<version>` sale de `uname -r`)
2. crear un Makefile con el siguiente contenido:

```
obj-m := hello.o
KVERSION := $(shell uname -r)

all:
    make -C /lib/modules/$(KVERSION)/build SUBDIRS=$(shell pwd) modules

clean:
    make -C /lib/modules/$(KVERSION)/build SUBDIRS=$(shell pwd) clean
```

3. ejecutar `make clean` y `make`
4. usar `insmod` y `rmmod`

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**
  - pueden accederse como una tira de bytes
  - suelen no soportar *seeking*
  - se los usa directamente mediante un nodo en el filesystem
  - tienen un subtipo interesante: **misc devices**
- **block devices**
  - direccionables de a "cachos" definidos
  - suelen soportar *seeking*
  - generalmente, su nodo es montado como un filesystem
- **network devices**
  - proveen acceso a una red
  - no son accedidos a través de un nodo en el filesystem, sino de otra manera (usando sockets, por ejemplo)

Podemos ver ejemplos con `ls -l /dev`

devices y *drivers*

```
lrwxrwxrwx  1 root root           3 2010-10-08 20:00 cdrom -> sr0
...
crw-rw-rw-  1 root root      1,   8 2010-10-08 20:00 random
...
brw-rw----  1 root disk      8,   0 2010-10-08 20:00 sda
brw-rw----  1 root disk      8,   1 2010-10-08 20:00 sda1
...
```

El primer caracter de cada línea representa el tipo de archivo:

- `l` es un *symlink* (enlace simbólico)
- `c` es un *char device*
- `b` es un *block device*

Los *devices* tienen un par de números asociados:

- **major**: está asociado a un driver en particular
- **minor**: identifica a un dispositivo específico que el driver maneja

Construcción de un *char device*

Vamos a construir un *char device*. ¿Qué necesitamos?

Construcción de un *char device*

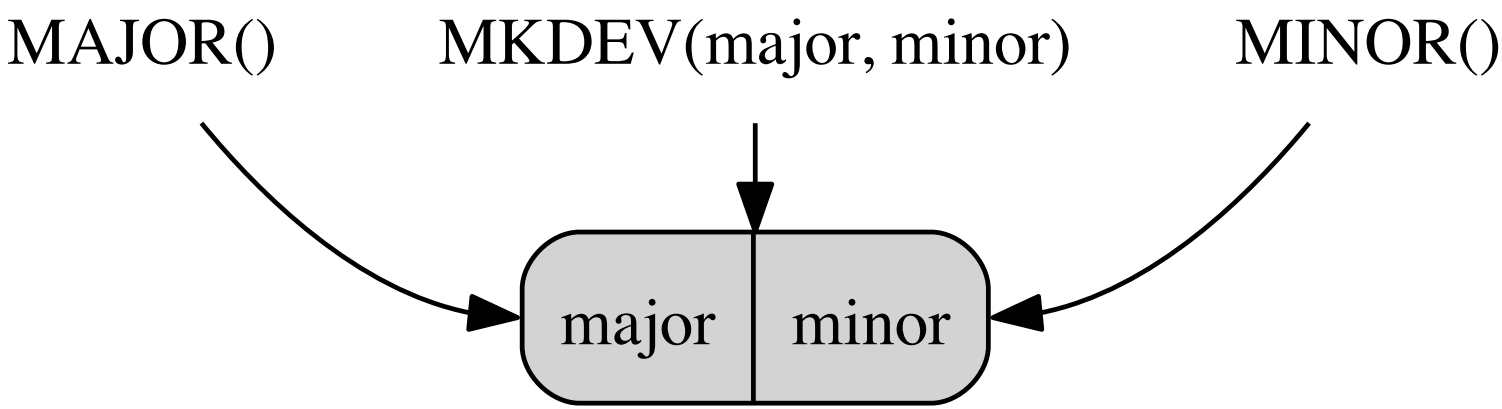
Vamos a construir un *char device*. ¿Qué necesitamos?

- crear un nodo en el filesystem para interactuar con nuestro módulo
- conseguir los *device numbers* que precisemos
- registrar al *device* como un *char device*
- registrar las funciones de cada operación que queramos realizar sobre el *device*

¿Qué parte del módulo debería encargarse de lo anterior?

Device numbers

En el código del kernel, el par *major-minor* se representa con el tipo `dev_t`:



Reservando *device numbers*

¿Cómo reservamos los *device numbers* que necesitamos?

- pedimos un rango específico (puede ser problemático)
- pedimos al kernel que nos asigne un rango dinámicamente

Para reservarlos dinámicamente y para liberarlos, tenemos:

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

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

Las operaciones

```
struct file_operations {
    struct module *owner;
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t,
        loff_t *);
    ...
}
```

- la estructura `file_operations` representa las operaciones que las aplicaciones pueden realizar sobre los *devices*
- cada campo apunta a una función en nuestro módulo que se encarga de la operación, o es `NULL`
- si el campo es `NULL` tiene lugar una operación por omisión distinta para cada campo

Las operaciones (2)

```
struct file_operations {
    struct module *owner;
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t,
        loff_t *);
    ...
}
```

- `owner`: un puntero al módulo "dueño" de la estructura (generalmente `THIS_MODULE`)
- `read()`: para recibir datos desde el *device*; retorna el número de bytes leídos
- `write()`: para enviar datos al *device*; retorna el número de bytes escritos

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario
- el puntero a espacio de usuario puede:
  - ser inválido: puede no haber nada mapeado en esa dirección, o puede haber basura;
  - no estar en memoria (paginado), y el kernel no puede incurrir en *page faults*;
  - ser erróneo o malicioso
- para estar tranquilos, hay que usar:

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

La abstracción cdev

- el kernel representa internamente a los *char devices* mediante la estructura `struct cdev`
- antes de que el kernel llame a nuestras operaciones, tenemos que inicializar y registrar al menos una de estas estructuras

Para inicializar, podemos:

1. pedir al kernel que dinámicamente nos reserve y otorgue una estructura:

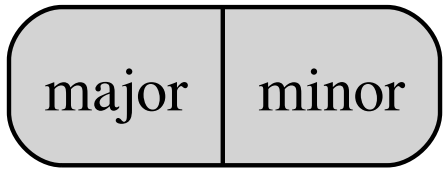
```
struct cdev *mi_cdev = cdev_alloc();
mi_cdev->ops = &mi_fops;
```

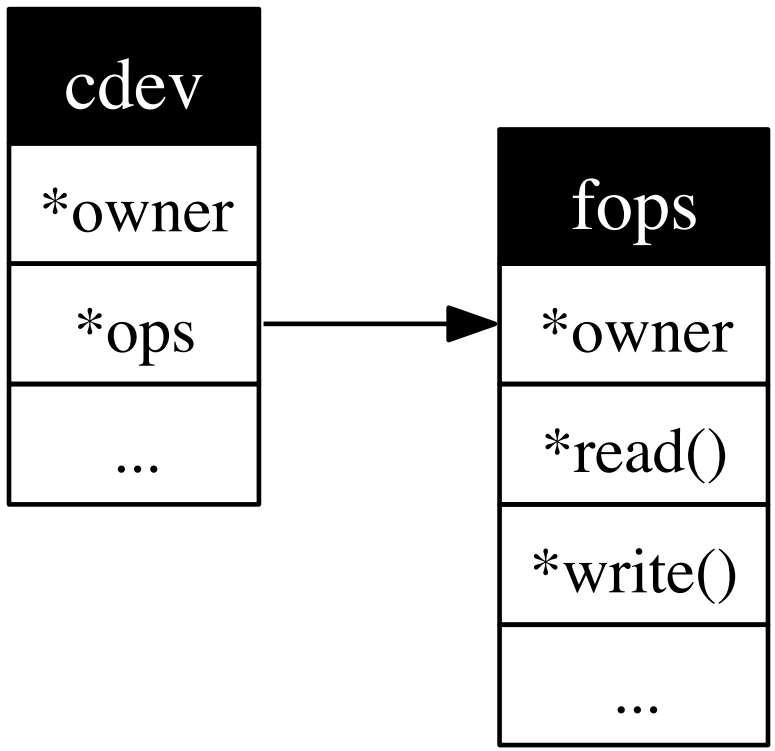
2. inicializar una estructura ya reservada:

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

En los dos casos, hay que inicializar `cdev.owner` a `THIS_MODULE`

La abstracción cdev (2)





La abstracción cdev (3)

Ahora, registramos con:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

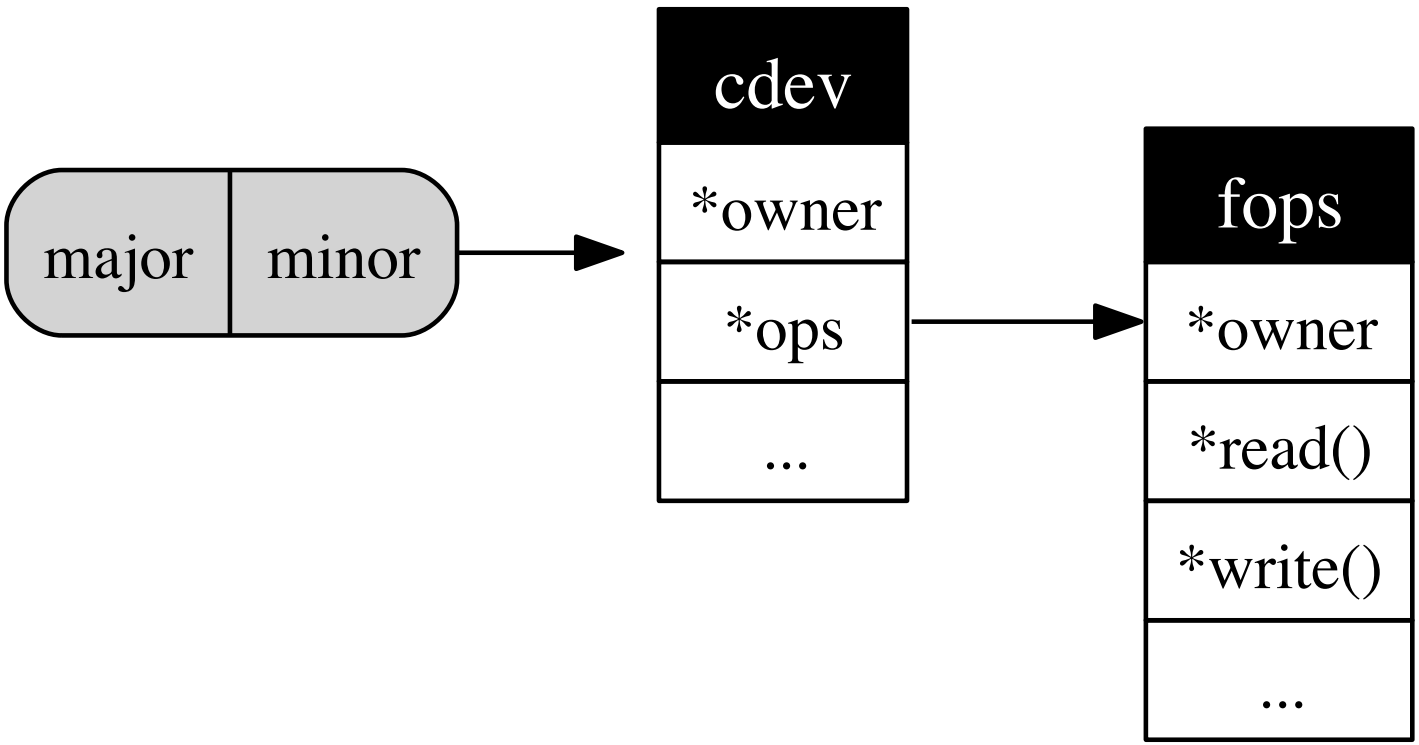
Tener en cuenta que:

- `cdev_add()` puede fallar
- si no falló, las operaciones de nuestro módulo ya pueden ser llamadas

Para quitar al *char device* del sistema, usar:

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

La abstracción cdev (4)



Creando nodos

- Una vez que el device está registrado, podemos crear los nodos en el filesystem
- Sin embargo, esto debe hacerse enteramente desde espacio de usuario
- ¿Por qué no desde el kernel?

Creando nodos

- Una vez que el device está registrado, podemos crear los nodos en el filesystem
- Sin embargo, esto debe hacerse enteramente desde espacio de usuario
- ¿Por qué no desde el kernel?

"Provide mechanism, not policy"

Una distinción muy importante en el mundo UNIX:

Mechanism : las capacidades y funcionalidad que se provee

Policy : el uso que se le da a esas capacidades

Creando nodos (2)

Tenemos, a priori, dos opciones:

- crear los nodos, una vez se haya insertado el módulo, usando `mknod <nodo> c <major> <minor>` ,
- que desde el módulo se genere algún tipo de aviso a alguien, en espacio de usuario, que se encargue de crear el nodo

Para lo segundo:

```
#include <linux/device.h>

static struct class *mi_class;

mi_class = class_create(THIS_MODULE, DEVICE_NAME);
device_create(mi_class, NULL, mi_devno, NULL, DEVICE_NAME);
```

```
device_destroy(mi_class, mi_devno);
class_destroy(mi_class);
```

Ejercicio

Se pide completar la implementación del módulo `sopipe` que funciona como *una especie de* `pipe`.
El funcionamiento debe ser el siguiente:

- Cuando un proceso escribe en el `sopipe`, la información se guarda en un *buffer* en memoria.
- Cuando un proceso lee se le devuelve, si hay, el siguiente byte que aún no se ha leído.
- La lectura debe ser bloqueante.
- El módulo debe soportar lecturas y escrituras concurrentes.
- La escritura debe tener éxito incluso ante un agotamiento del espacio del *buffer*. Es decir que, en caso de ser necesario, debemos pedir más memoria.

Ejemplos de lectura y escritura:

- `cat /path/al/nodo/sopipe` ,
- `echo -n "17"| sudo tee /path/al/nodo/sopipe`

Memoria dinámica (kernel)

Existen diversas formas de pedir memoria al sistema cuando estamos ejecutando en modo kernel. Nosotros vamos a ver dos:

kmalloc

Funciones: `void * kmalloc(size_t size, int flags)`, `kfree(void * ptr)`. Solicita un espacio de memoria **físicamente contiguo** de `size` bytes. Devuelve un puntero *virtual* accesible sólo en modo kernel.

vmalloc

Funciones: `void * vmalloc(size_t size)`,`vfree(void * ptr)`. Solicita un espacio de memoria **virtualmente contiguo** de `size` bytes. Devuelve un puntero *virtual* accesible sólo en modo kernel.

Sincronización (kernel)

Diversos mecanismos de sincronización. Entre ellos semáforos y *mutexes*.

`sempahore`

Tipo de datos: `struct semaphore`. Funciones: `sema_init(struct semaphore * sem, int val)`, `down(struct semaphore * sem)`, `down_interruptible(struct semaphore * sem)`,...,`up(struct semaphore * sem)`

`spinlock`

Son *una suerte de* mutex. Lo que varía es cómo se implementan. Más eficientes que los *mutexes* en varios contextos. Conceptualmente hacen lo mismo. Tipo de datos: `spinlock_t` Funciones: `spin_lock_init(spinlock_t * lock)`, `spin_lock(spinlock_t * lock)`, `spin_unlock(spinlock_t * lock)`, etc.

Montando filesystems remotos

Vamos a acceder a nuestros archivos en la computadora *física* desde la *virtual*.

```
# apt-get install sshfs
# mkdir remoto
# sshfs MI_USUARIO@IP_HOST:/home/MI_USUARIO remoto
```

- Reemplazar *MI_USUARIO* por su cuenta de los labos e *IP_HOST* por la dirección IP de la máquina que están usando.
- El comando *ifconfig* es su amigo.
- Este comando logra abstraer la idea de que los archivos están en una ubicación remota, para el SO son archivos comunes y corrientes.
- Ahora pueden editar desde su usuario de los labos y solo compilar para probar en la consola de la máquina virtual.