

Taller de drivers

Sistemas Operativos

Alejandro Deymonnaz¹

¹Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

24 de Mayo de 2011

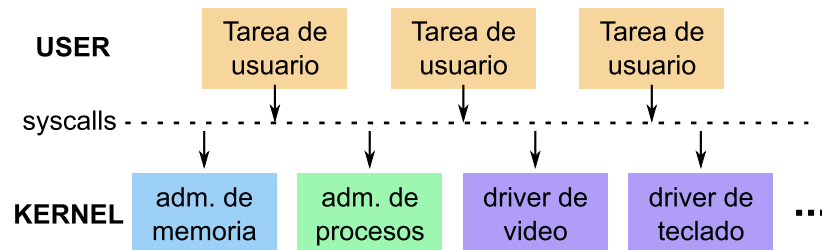
Menú del día

Hoy veremos:

- 1 Kernel de Linux: La teoría
 - Estructura del Sistema Operativo
 - Módulos
- 2 Kernel de Linux: La práctica
 - Manos a la obra
- 3 Taller

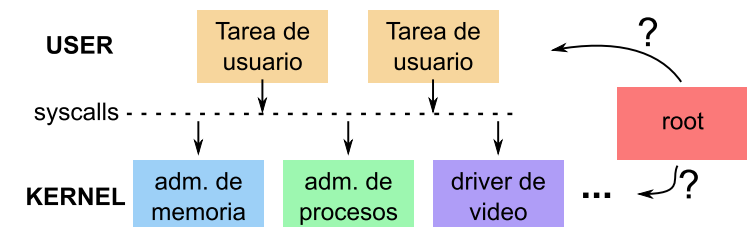
La estructura de Linux (1)

En **linux** se divide en sólo **dos** espacios: user y kernel.

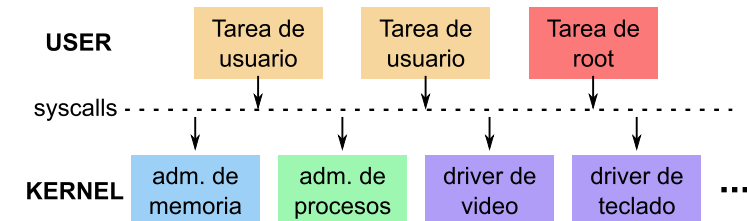


- kernel **monolítico**: administradores, memoria, drivers, todo junto.
- el kernel tiene acceso a todo:
 - ejecuta en un **único** espacio de memoria
 - ejecuta en el **máximo** nivel de privilegio

¡Momento!



Los programas de root ¿dónde están?



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?

Módulos (1)

Un **módulo** se compone comúnmente de:

- Funciones (auxiliares, internas, etc)
- Datos
- Puntos de entrada y salida

Situación: Está ejecutando una tarea de usuario. (ej: `while(1);`). ¿Cómo pasa a ejecutarse el código de un módulo del kernel? (o código del kernel en general)

- 1 Llamada al sistema (pedido **voluntario** del usuario)
- 2 Atención de interrupción (pedido **involuntario**)

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 (2)

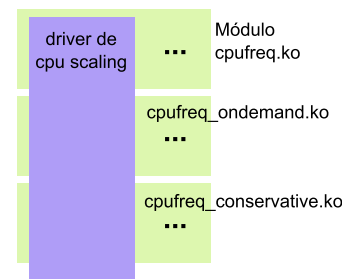
Puede tener **varios** drivers de dispositivos muy similares.

Módulo nvidia.ko

driver de
nvidia 8600GT

driver de
nvidia 8800GT

...



Un mismo **driver** puede estar implementado repartido en distintos **módulos**.

Nuevo Mundo



Sabi (DC - FCEyN - UBA)

Taller de drivers

1er cuat. 2011 9 / 34

Nuestro primer módulo (1)

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

static int __init hello_init(void) {
    printk(KERN_ALERT "Aloha honua!\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("Sabi");
MODULE_DESCRIPTION("'Aloha honua' desde el kernel.");
```

Sabi (DC - FCEyN - UBA)

Taller de drivers

1er cuat. 2011 11 / 34

Nuevo mundo

- Estamos ejecutando en el nivel de máximo privilegio
- ¿Qué pasa si hacemos un acceso indebido a memoria?
- El kernel no está enlazado a la `libc`
- Hacer operaciones de punto flotante es complicado
- Tenemos un stack fijo y limitado (y tenemos que compartirlo con el resto del kernel, pero no con el usuario)
- Hay varias fuentes de posibles condiciones de carrera

Sabi (DC - FCEyN - UBA)

Taller de drivers

1er cuat. 2011 10 / 34

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()`

Sabi (DC - FCEyN - UBA)

Taller de drivers

1er cuat. 2011 12 / 34

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sabi");
MODULE_DESCRIPTION("'Aloha honua' desde el kernel.");
```

- `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
- `MODULE_AUTHOR()` y `MODULE_DESCRIPTION()` son meramente informativos

Nuestro primer módulo (5)

```
static int __init hello_init(void) {
    printk(KERN_ALERT "Aloha honua!\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 registrar 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 (4)

```
static int __init hello_init(void) {
    printk(KERN_ALERT "Aloha honua!\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 (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 el **destructor** del módulo o función de limpieza.
- el destructor es llamado 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)

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`

Manos a la obra

- 1 instalar los paquetes para
 - `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`

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 podría hacer nuestra función de inicialización?

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

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, tenemos `alloc_chrdev_region()`.

Recibe:

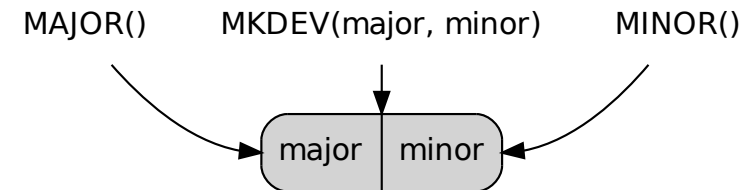
- `dev_t *dev`: parámetro de salida
- `unsigned int firstminor`: primer *minor* a ser usado
- `unsigned int count`: cantidad de *device numbers* contiguos
- `char *name`: nombre del device asociado al rango

Para liberarlos, `unregister_chrdev_region()`. Recibe:

- `dev_t *first`
- `int count`

Device numbers

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



- las macros `MAJOR(dev_t dev)` y `MINOR(dev_t dev)` nos dan cada número
- `MKDEV(int major, int minor)` nos da un `dev_t` para el par de números

Las operaciones (1)

```

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 *);
    ...
}
  
```

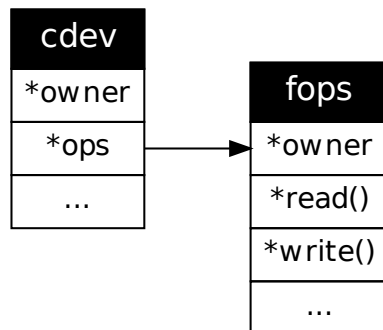
- ya tenemos nuestros *device numbers*, pero todavía no podemos realizar operaciones con ellos.
- la estructura `file_operations` representa las operaciones que las aplicaciones pueden realizar sobre los *devices*
- cada campo apunta a una función en nuestro módulo que se encarga de la operación, o es `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
- si el puntero de **read()** o **write()** es `NULL`, se retorna `-EINVAL` al tratar de realizar la operación

La abstracción cdev (2)



La abstracción cdev (1)

- 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, del siguiente modo:

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

- 2 inicializar una estructura ya reservada usando:

```
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 (3)

Ahora, registramos con:

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

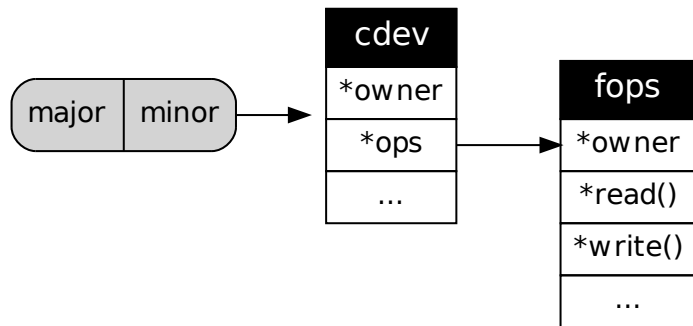
Tener en cuenta que:

- `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 (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 señal 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);
```

Creando nodos (1)

- 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

Misc devices

¿Qué pasa si no queremos tanta flexibilidad?

- los *misc devices* son *char devices* simples que tienen varias características y una API en común
- todos comparten el *major* 10, pero pueden elegir su propio *minor*
- una llamada a `misc_register()` simplifica los pasos de:
 - pedir *device numbers*
 - crear nodos en `/dev`
 - registrar el *char device* con `cdev_init()` y `cdev_add()`

```
static struct miscdevice mi_dev = {
    MI_MINOR,
    "midriver",
    &mi_fops,
};

misc_register(&mi_dev);
```


Módulos - Ejercicio

- Para una aplicación de estadística, se necesita tener un driver de tipo caracter que retorne una letra entre A y Z en forma aleatoria cada vez que se lea de él. Sólo se puede leer de dicho driver. Este driver ha sido visto en la clase, pero se pide que se reescriba el código teniendo en cuenta las operaciones indicadas en clase.
- Como un agregado a este problema, se pide que se pueda cambiar la semilla de random a través de `/dev/probchar`,
- En resumen:
 - Para leer la próxima letra se usará: `dd if=/dev/probchar count=1 bs=1`.
 - Para cambiar la semilla inicial: `echo "45" > /dev/probchar`

Entrega del taller

Deben enviar el código debidamente comentado, a sisopdc@gmail.com con subject:

Taller Drivers: grupo Apellido1, Apellido2

Reemplazando `Apellidoi` por los apellidos de los **2** integrantes del grupo.
La fecha límite de entrega es el Lunes 30/05/2011 a las 23:59 GMT-0300