Taller de drivers

Sistemas Operativos

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

30 de Mayo de 2017

El kernel

Prendo la computadora, se carga **EL** kernel.

- Administrador de memoria ✓
- Administrador de procesos ✓
- Sistema de archivos ✓
- Driver teclado, mouse, video ✓

- ¿Ya existía el código en el kernel?
- ¿Tengo que reiniciar la máquina y cargar de nuevo el kernel?
- ¿Tengo que recompilar el kernel?

Solución (linux): Módulos

- Linux soporta la carga y descarga de **módulos** al kernel en **tiempo de ejecución.**
- «*Los únicos privilegiados son los módulos*», JDP

Nuestro primer módulo

```
//Incluir module_init y module_exit
#include <linux/init.h>
//Incluir MODULE_*mod
#include <linux/module.h>


static int __init hello_init(void) {
  return 0;
}

static void __exit hello_exit(void) {

}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("La banda de SO");
MODULE_DESCRIPTION("Nuestro primer modulo");
```

Cargar nuestro módulo

1. Requisitos previos
   - `make`
   - `module-init-tools`
   - `linux-headers-<version>`
     (`<version>` sale de `uname -r`)
2. Makefile especial:

```
obj-m := hello.o hello0.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. Cargarlo con `insmod` (se saca con `rmmod`)

¿Cómo lo *uso*?

- Ver módulos cargados con `lsmod`
- ¿Y cuándo se ejecuta el código?
  1. Al cargar el módulo
  2. Llamada al sistema
  3. Atención de interrupción
  4. Al descargar el módulo

Advertencia

Advertencia

«*Today we have learned in the agony of war that* **great power involves great responsibility**», Franklin D. Roosevelt

Advertencia: Los peligros de ser el cero

- Bugs.
- Punto flotante.
- Libc. Solución: `<linux/kernel.h>`
- Espacio de nombres.
- Condiciones de carrera.
- Accesos a memoria.

Hola mundo

```
#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("La banda de SO");
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

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)

Devices y drivers

`ls -l /dev/`

```
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 (primer número luego del grupo)
- **minor**: identifica a un dispositivo específico que el driver maneja (segundo número luego del grupo)

Construcción de un *char device*

1. Registrarlo
2. Conseguir los *device numbers*
3. Definir las funciones de cada operación del *device*
4. Crear un nodo en el filesystem para interactuar con el *device*

¿En qué parte del módulo se hace todo esto?

(1) Registrar el *char device*

- Llamar a la función `cdev-init` en el init

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

- No olvidarse de los include

```
#include <linux/fs.h>
#include <linux/cdev.h>
```

(2) Conseguir los *device numbers*

¿Cómo reservamos los *device numbers* que necesitamos?

- Asignamos uno específico (puede ser problemático)
- Pedimos al kernel que nos asigne uno dinámicamente

Para reservarlos dinámicamente y para liberarlos, tenemos:

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

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

Para (des)asignárlo al *char device*:

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


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

(3) Definir 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

(3) Definir las operaciones: un ejemplo

```
static struct file_operations mis_operaciones = {
    .owner = THIS_MODULE,
    .read = mi_operacion_lectura,
};

ssize_t mi_operacion_lectura(struct file *filp, char __user *data,
 size_t s, loff_t *off) {
    ....
    return 0;
}
```

(3) Definir las operaciones: advertencia

- 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);
```

(4) Crear nodos

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

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 1

Implementar el módulo `/dev/nulo` que replique exactamente la funcionalidad de `/dev/null`. Es decir, descartar toda la información que se escribe en él y, a su vez, no leer ningún caracter cuando se intenta leer.

Ejercicio 2

Escribir el módulo `/dev/letras123` que debe realizer lo siguiente:

- Asignar uno de los tres espacios libres al proceso cuando este abra el dispositivo con `open`. Fallar con -EPERM si no hay más lugar.
- Liberar el espacio asignado cuando el usuario haga `close` (o fallar con -EPERM si no se había realizado el `open` correspondiente).
- La primera vez que el usuario hace un `write` (y si tiene un espacio libre asignado) guardar el primer caracter al espacio libre del usuario. Los siguientes writes deben ser ignorados.
- Cada vez que el usuario lea del dispositivo, devolverle tantas copias del caracter guardado como haya pedido leer. Si no se hizo `write` previamente, fallar con -EPERM.

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

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.