# Clase Taller de Módulos

Gastón Aguilera

Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

# ¿Que haremos hoy?

Temas:

- Device drivers
  - Los modulos y los device drivers
  - Las estructuras que exporta el kernel via fs virtuales
  - Algunas estructuras internas del kernel
  - Ejemplos de modulos que hacen uso de dichas estructuras
  - Ejercicio para entregar

# Módulos y Drivers

- Recordemos que los modulos de un kernel puede ser agregados a la imagen del kernel en forma estática o en forma dinámica en tiempo de ejecución.
- Cabe señalar que esta segunda formula de agregar funcionalidad al kernel, se trata de una solución implementativa y no de diseño del kernel.
- Esto hace que se deba tener especial cuidado en como se hace uso de los recursos del kernel. No existe tipo alguno de protección. Y para cargarlos hace falta tener privilegios de root.
- Se hizo una revisión de la estructura de un modulo de kernel y la forma de generarlo.
- Veamos a continuación algunos ejemplos ...
  - Módulo de manejo de luces de teclado
  - Módulo de sobrecarga de system-call
  - Módulo de manejo de device driver

# Módulos - Luces de teclado

- Con este módulo, se verá la forma que se accede a las funciones de otros módulos y a las del kernel en sí.
- Una forma que el kernel y los modulos tienen para comunicarse con el usuario es a traves de los system-calls y a traves de sistemas de archivos virtuales como ser /proc.
- En el sistema de archivos /proc se pueden crear archivos que son accesibles por las funciones generales de archivos (open, close, read, write).
- Dichas funciones pueden ser mapeadas a funciones propias del módulo, con solo respetar la definición de las funciones sobre los parámetros formales.

## Código fuente del Driver

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kd.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/ioctl.h>
#include <linux/syscalls.h>
#include <linux/proc_fs.h>
#include <linux/tty.h>
#include <linux/vt_kern.h>
#include <linux/console_struct.h>
#include <asm/uaccess.h>

/* Prototipos de las funciones de inicializacion y destruccion */
static int __init luces_init(void);
static void __exit luces_exit(void);

// Prototipos de funciones de leds
static void setLeds(long int);
static long int getLeds(void);
static void restaurarLeds(void);

/* Informamos al kernel que inicialice el modulo usando luces_init
* y que antes de quitarlo use luces_exit
*/

module_init(luces_init);
module_exit(luces_exit);
```

## Código fuente del Driver

```
# Definicion de las variables globales

#define FILE_SIZE 4 // Un caracter por cada led, y un end of line al final.
#define procfs_name "keyboardLeds" // Nombre del archivo en /proc

long int originalLeds;
char miBuffer[4];
const long int LED_FLAGS[4] = {LED_NUM,LED_CAP,LED_SCR};
const char LED_CHARACTERS[4] = {'n','c','s'};

struct tty_driver *mi_driver; // Driver de la consola
struct proc_dir_entry *procFile; // Informacion de nuestro archivo en /proc
```

## Código fuente del Driver

```
/* Inicializacion */
static int __init luces_init()
{
        printk(KERN_ALERT "Se carga el modulo de control de LEDs\n");

        // Inicializamos lo relativo al driver de la consola.
        mi_driver = vc_cons[fg_console].d->vc_tty->driver;

        // Al momento de cargar el modulo, mostramos el estado de los LEDs
        mostrarLEDs(originalLeds = getLeds());

        // Inicializamos el archivo en /proc
        procFile = create_proc_entry(procfs_name, 0666,&proc_root);
        if (procFile == NULL)
        {
                remove_proc_entry(procfs_name, &proc_root);
                printk(KERN_ALERT "Error: Could not initialize /proc/%s\n",procfs_name);
                return -ENOMEM;
        }

        // Asociamos las funciones de lectura y escritura al archivo creado
        procFile->read_proc  = procFileRead;
        procFile->write_proc = procFileWrite;
        procFile->owner      = THIS_MODULE;
        procFile->mode       = S_IFREG | S_IRUGO | S_IWUGO; // rw-rw-rw-
        procFile->uid        = 0;
        procFile->gid        = 0;
        procFile->size       = FILE_SIZE;
        return 0;
}
```

## Código fuente del Driver

```
/* Destruccion */

static void __exit luces_exit()
{
        printk(KERN_ALERT "Se descarga el modulo de control de LEDs.\n");
        restaurarLeds();
        remove_proc_entry(procfs_name,&proc_root);
}


/* Declaramos que este codigo tiene licencia GPL.
*/
MODULE_LICENSE("GPL");
```

## Código fuente del Driver

```
static void mostrarLEDs(long int leds)
{
        if (leds & LED_NUM) {printk(KERN_ALERT "Num Lock LED is ON\n");}
        if (leds & LED_CAP) {printk(KERN_ALERT "Caps Lock LED is ON\n");}
        if (leds & LED_SCR) {printk(KERN_ALERT "Scroll Lock LED is ON\n");}
}

static long int getLeds(void)
{
        mm_segment_t old_fs;
        long int res;

        // Tenemos que cambiar de segmento de usuario a segmento de kernel,
        // porque ioctl esta preparado para ser invocado por el usuario,
        // y por lo tanto interpreta los punteros como punteros en el
        // espacio de direcciones del usuario, mientras que nosotros lo
        // invocamos desde el kernel.
        old_fs = get_fs();
        set_fs(KERNEL_DS);
        mi_driver->ioctl(vc_cons[fg_console].d->vc_tty,NULL,KDGETLED,(int)&res);
        set_fs(old_fs);
        return res;
}

static void setLeds(long int leds)
{ mi_driver->ioctl(vc_cons[fg_console].d->vc_tty,NULL,KDSETLED,leds); }

static void restaurarLeds(void) { setLeds(originalLeds); }
```

## Código fuente del Driver

```
void updateBuffer(void)
{
        int i;
        int leds = getLeds();
        for(i=0;i<3;i++)
                miBuffer[i] = ((leds&LED_FLAGS[i])?LED_CHARACTERS[i]:'-');
        miBuffer[3] = '\n';
}

int procFileRead(char *buffer,char **buffer_location,off_t offset,
                                int bufferLength, int *eof, void *data)
{
        if (offset>=FILE_SIZE || offset < 0) return 0;
        else
        {
                updateBuffer();
                *buffer_location = miBuffer+offset;
                return FILE_SIZE-offset;
        }
}
```

## Código fuente del Driver

```
int procFileWrite(struct file *file, const char *buffer, unsigned long count, void *data)
{
        long int leds;
        unsigned long oCount;
        typeof(file->f_pos) i;
        char writeBuffer[4];

        if (file->f_pos < 0) return count; // No se modifica nada si el offset es negativo.
        oCount = count;
        leds = getLeds() & (LED_NUM | LED_CAP | LED_SCR);

        copy_from_user(writeBuffer,buffer,min(3UL,count));
        if (count > 0 && writeBuffer[0] == 'X') {
                // Si se escribe cualquier cosa que tenga a 'X' como
                // primer caracter, se interpreta que se debe restaurar
                // los leds al estado original al cargar el modulo.
                restaurarLeds();
                return oCount;
        }
        for (i=file->f_pos; i < 3 && count > 0; i++,count--) {
                if (writeBuffer[oCount-count] == LED_CHARACTERS[i])
                        leds |= LED_FLAGS[i];    // Se Prende el led
                else if (writeBuffer[oCount-count] == '-')
                        leds &= ~(LED_FLAGS[i]); // Se Apaga el led
        }
        setLeds(leds);
        return oCount;
}
```

# Módulos - syscall mkdir

- Con este módulo, se verá la forma que se accede a las estructuras internas del kernel.
- Hace varias versiones atras(2.4.x), el acceso a la tabla de las funciones de syscall se encontraba desprotegida y cualquier modulo podia realizar cambios en ella. Esto atentaba contra la proteccion del kernel y permitía hacer mas simple el trabajo de instalación de root kits.
- En las versiones 2.6 esto fue solucionado haciendo que dicha tabla no esté exportada y poniendola en una zona de memoria de solo lectura.

# Módulos - syscall mkdir

## Código fuente del Driver

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kd.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/ioctl.h>
#include <linux/syscalls.h>
#include <linux/proc_fs.h>
#include <linux/fs.h>
#include <linux/tty.h>
#include <linux/vt_kern.h>
#include <linux/console_struct.h>
#include <linux/random.h>
#include <linux/kdev_t.h>
#include <asm/uaccess.h>
#include <asm/agp.h>

/* Prototipos de las funciones de inicializacion y destruccion */
static int __init sysmk_init(void);
static void __exit sysmk_exit(void);

/* Informamos al kernel que inicialice el modulo usando hello_init
* y que antes de quitarlo use hello_exit
*/

module_init(sysmk_init);
module_exit(sysmk_exit);
```

## Código fuente del Driver

```
#define SUCCESS 0
#define SYS_CALL_TABLE_BASE 0xc0318500

static const void **sys_call_table = (void *) SYS_CALL_TABLE_BASE;

asmlinkage int (*sys_call_orig)(const char *pathname);
static struct page *pg;
static pgprot_t prot;

int miMkdir(const char *path) {        return -EPERM; } // Siempre falla

/* Inicializacion */
static int __init sysmk_init()
{
        printk(KERN_ALERT "Se carga el modulo nomkdir\n");

        sys_call_orig = sys_call_table[__NR_mkdir];

        pg = virt_to_page(SYS_CALL_TABLE_BASE);
        prot.pgprot = VM_READ | VM_WRITE | VM_EXEC;
        change_page_attr(pg,1,prot);

        sys_call_table[__NR_mkdir] = miMkdir;

        return SUCCESS;
}
```

# Módulos - syscall mkdir

## Código fuente del Driver

```
/* Destruccion */
static void __exit sysmk_exit()
{
        printk(KERN_ALERT "Se descarga el modulo nomkdir\n");

        sys_call_table[__NR_mkdir] = sys_call_orig;
}

MODULE_LICENSE("GPL");
```

# Módulos - device driver probabilidad

- Con este módulo, se verá la forma en que se registran los drivers de cualquier tipo y en particular del tipo misc.
- Haciendo una revision existen varios tipos de drivers para: misc device, character device, block device.
- Dichos drivers existen en un directorio o file system particular /dev.
- El directorio /dev ha sido modificado varias veces en cuestion de las formas en que se registran los devices.

# Modulos - device driver probabilidad

## Código fuente del Driver

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kd.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/ioctl.h>
#include <linux/syscalls.h>
#include <linux/proc_fs.h>
#include <linux/fs.h>
#include <linux/tty.h>
#include <linux/vt_kern.h>
#include <linux/console_struct.h>
#include <linux/random.h>
#include <linux/kdev_t.h>
#include <asm/uaccess.h>

/* Prototipos de las funciones de inicializacion y destruccion */
static int __init proba_init(void);
static void __exit proba_exit(void);

// Prototipos de las funciones de manejo del dispositivo
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);

module_init(proba_init);
module_exit(proba_exit);
```

# Modulos - device driver probabilidad

## Código fuente del Driver

```
#define SUCCESS 0
#define PROC_FILE_SIZE sizeof(int) // Un entero
#define SEMILLA_DEFAULT 1238732
#define procfs_name "probabilidad" // Nombre del archivo en /proc
#define DEVICE_NAME "probabilidad"
static int Device_Open = 0;
static struct proc_dir_entry *procFile; // Informacion de nuestro archivo en /proc
static int cantidadLecturas;
static int Major; // Device major number
// Funciones de manejo del dispositivo en /dev
static struct file_operations fops =
{
        .read    = device_read,
        .open    = device_open,
        .release = device_release
};
```

## Código fuente del Driver

```
// Maneja aperturas del archivo
static int device_open(struct inode *inode, struct file *file)
{
        if (Device.Open)
            return -EBUSY;
        Device_Open++;
        try_module_get(THIS_MODULE);
        return 0;
}

// Maneja el evento de cierre del archivo
static int device_release(struct inode *inode, struct file *file)
{
        Device_Open--;
        module_put(THIS_MODULE);
        return 0;
}

static ssize_t device_read(        struct file *filp, char *buffer, size_t length,
                                loff_t *offset)
{
        size_t i;
        cantidadLecturas++;
        for(i = 0; i < length; i++)
        {
                unsigned int r = random32();
                buffer[i] = 'A' + r%26;
        }
        return length;
}
```

## Código fuente del Driver

```
// Funciones de manejo de la entrada en /proc
// Devuelve al usuario la cantidad de lecturas realizadas.
int procFileRead(char *buffer,char **buffer_location,off_t
                offset,        int bufferLength, int *eof, void *data)
{
        if (offset>=PROC_FILE_SIZE || offset < 0) return 0;
        else
        {
                sprintf(buffer,"%d",cantidadLecturas);
                return PROC_FILE_SIZE-offset;
        }
}

// Lee la semilla del usuario y la utiliza.
int procFileWrite(struct file *file, const char *buffer,
                unsigned long count, void *data)
{
        u32 semilla = 0;
        // No se modifica nada si el offset esta fuera de rango.
        if (file->f_pos < 0 || file->f_pos >= PROC_FILE_SIZE) return count;
        copy_from_user(&semilla,buffer,min((unsigned long)sizeof(semilla),count));
        srandom32(semilla);
        printk(KERN_ALERT "Se cambia la semilla por %u\n",semilla);
        return count;
}
```

# Modulos - device driver probabilidad

## Código fuente del Driver

```
/* Inicializacion */
static int __init proba_init()
{
        printk(KERN_ALERT "Se carga el modulo probabilidad\n");
        cantidadLecturas = 0;
        srandom32(SEMILLA_DEFAULT);

        // Inicializamos el dispositivo en /dev
        Major = register_chrdev(0,DEVICE_NAME,&fops);
        if (Major < 0)
        {
                printk(KERN_ALERT "Registering char device failed with %d\n",Major);
                return Major;
        }
        printk(KERN_ALERT "Se registro el driver con Major number = %d\n",Major);
```

# Modulos - device driver probabilidad

## Código fuente del Driver

```
        // Inicializamos el archivo en /proc
        procFile = create_proc_entry(procfs_name, 0666,NULL);
        if (procFile == NULL)
        {
                printk(KERN_ALERT "Error: Could not initialize /proc/%s\n",procfs_name);
                unregister_chrdev(Major, DEVICE_NAME);
                return -ENOMEM;
        }

        procFile->read_proc  = procFileRead;
        procFile->write_proc = procFileWrite;
        procFile->owner      = THIS_MODULE;
        procFile->mode       = S_IFREG | S_IRUGO | S_IWUGO; // rw-rw-rw-
        procFile->uid        = 0;
        procFile->gid        = 0;
        procFile->size       = PROC_FILE_SIZE;

        /* Si devolvemos un valor distinto de cero significa que
         * hello_init fallo y el modulo no puede ser cargado.
         */
        return 0;
}
```

# Modulos - device driver probabilidad

## Código fuente del Driver

```
/* Destruccion */
static void __exit proba_exit()
{
        printk(KERN_ALERT "Se descarga el modulo probabilidad\n");
        remove_proc_entry(procfs_name,NULL);
        unregister_chrdev(Major,DEVICE_NAME);
}

MODULE_LICENSE("GPL");
```

# Módulos - Ejercicio

- Para una aplicacion de estadistica, 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. Solo se puede leer de dicho driver. Este driver ha sido visto en la clase, pero se pide que se reescriba el codigo 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 traves de /proc/probabilidad,
- En resumen:
  - Para leer la proxima letra se usara: dd if=/dev/probabilidad count=1 bs=1.
  - Para cambiar la semilla inicial: echo "45" > /proc/probabilidad

# Módulos - Ejercicio

- Para un proyecto de ciencias, los alumnos de primaria, necesitan tener implementado la sucesion de fibonacci. No como una función sino como un device, al cual se le pueda indagar por el siguiente número.
- Para lograr este objetivo se nos ocurrió que se genere un módulo, que maneje el dispositivo /dev/fibonacci del tipo character device. Al cual se le puede escribir los dos números por medio de la función de usuario write, y obtener el próximo número fibonacci mediante la funcion de usuario read.
  - Para leer el próximo numero se usara: dd if=/dev/fibonacci count=1 bs=1.
  - Para escribir los dos numeros iniciales: echo "4 5" > /dev/fibonacci
- Como un agregado a este problema, se pide que en /proc/fibocount, se informe la cantidad de lecturas y escrituras realizadas al dispositivo /dev/fibonacci.
- Y para hacerles la vida mas facil se debe utilizar un device miscelaneo. (Ver http://www.linuxjournal.com/article/2920).

# Modulos - device driver fib

## Código fuente del Driver

```
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/module.h>
#include <linux/vmalloc.h>
#include <linux/time.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");

// Variables globales
int acum_lecturas = 0;
static struct proc_dir_entry *proc_entry;
static unsigned long prev_fib, prev_fib2;
static char *input_fibos;

//Funcion generadora de numeros fibonacci.
unsigned long get_fibona(unsigned long m_w, unsigned long m_z)
{
    return (m_z + m_w);
}

//Funciones de lectura y escritura invocadas por /proc fs.
int cant_lecturas(char * page, char **start, off_t off, int count, int *eof, void *data)
{
    int len;
    len = sprintf(page, "Lecturas/Escrituras realizadas: %d\n", acum_lecturas);
    return len;
}
```