

Auxiliar 10

Sistemas Operativos

Prof: Luis Mateu
Aux: Javier Bustos

Módulos en kernel 2.6.X

La primera pregunta que nace al hablar de *módulos del núcleo* (kernel) de un sistema operativo Linux es: ¿qué es un módulo de kernel?. La respuesta es simple: son aplicaciones en C que uno puede cargar o descargar del núcleo en demanda, sin necesidad de reiniciar el computador.

Luego nace la pregunta: ¿cómo diablos cargo o descargo los módulos del núcleo?. La respuesta es aún más fácil: existen comandos ya creados en el núcleo para hacerlo. Por ejemplo, si tengo el programa compilado como módulo `mimodulo.ko`, entonces las operaciones que se pueden hacer sobre él son:

- `insmod mimodulo.ko`: Cargar el módulo en el núcleo
- `rmmod mimodulo`: Descargar el módulo desde el núcleo
- `lsmod`: ver los módulos ya cargados en el núcleo
- `depmod -a`: crea el archivo `/lib/modules/version/modules.dep` el cual contiene las dependencias de todos los módulos
- `modprobe -a mimodulo.ko`: Primero ve las dependencias del módulo `mimodulo.ko`, si alguna falta la carga y luego carga `mimodulo.ko`

Hola Mundo

El primer programa que uno realiza en todo aprendizaje de ciencias de la computación, es el “hola mundo”. En la programación de módulos, debemos considerar lo siguiente:

Los módulos deben tener al menos dos funciones: `init_module()` y `cleanup_module()`. La primera de ellas es llamada al momento de hacer `insmod` y la segunda al momento de hacer `rmmod`. Típicamente, `init_module()` registra handlers para realizar acciones con el núcleo, o bien reemplaza alguna función del núcleo con un nuevo código. Por último, todo módulo de núcleo debe imperativamente incluir la biblioteca `linux/module.h`.

Nuestro primer programa será entonces:

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_ALERT */
```

```

MODULE_LICENSE("GPL"); /* Evitamos warnings de copyright */

int init_module(void)
{
    printk("<1>Hello world 1.\n");

    // A non 0 return means init_module failed; module can't be loaded.
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_ALERT "Goodbye world 1.\n");
}

```

Compilando el módulo

Para los que sabían programar módulos en la versión 2.4.X del kernel existe una buena noticia: *ahora los flags no son necesarios!*. Un Makefile típico para módulos versión 2.6.X es como el siguiente:

```

obj-m      := mimodulo.o

KDIR      := /lib/modules/$(shell uname -r)/build
PWD       := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

```

Y luego se hace simplemente:

```
make
```

Asignándole parámetros al módulo

Los módulos pueden tomar parámetros desde la línea de comando, pero no utilizando `argc/argv`. Para permitir que los parámetros sean pasados a su módulo, debe declarar las variables que tomarán los valores como globales y después utilizan la macro `MODULE_PARM()`, (definida en `linux/module.h`) para realizar la asociación. En tiempo de ejecución, `insmod` llenará las variables con los valores adquiridos desde la línea de comando. Comúnmente las declaraciones de variables y macros se escriben al principio del código por claridad.

La macro `MODULE_PARM()` utiliza dos argumentos: el nombre de la variable y su tipo. Los tipos validos por el momento son: `b`: *single byte*, `h`: *short int*, `i`: *integer*, `l`: *long int* y `s`: *string*. Strings deben ser declarados como `"char *"` e `insmod` pedirá memoria para ellos. Es recomendable asignarle a cada variable un valor "por defecto", para evitar problemas de variables no definidas (en caso de no recibir el parámetro respectivo).

El siguiente ejemplo muestra la utilización de las macros:

```

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Javier Bustos");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "blah";

MODULE_PARM (myshort, "h");
MODULE_PARM (myint, "i");
MODULE_PARM (mylong, "l");
MODULE_PARM (mystring, "s");

int init_module(void)
{
    printk(KERN_ALERT "Hello, world... again\n=====\\n");
    printk(KERN_ALERT "myshort is a short integer: %hd\\n", myshort);
    printk(KERN_ALERT "myint is an integer: %d\\n", myint);
    printk(KERN_ALERT "mylong is a long integer: %ld\\n", mylong);
    printk(KERN_ALERT "mystring is a string: %s\\n", mystring);
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_ALERT "Goodbye, world 5\\n");
}

```

Otras macros útiles

Si uno desea utilizar sus propias funciones de carga y descarga de módulos, entonces debe utilizar las macros `module_init()` y `module_exit()`, utilizando como parámetro el nombre de sus funciones, por ejemplo:

```

#include <linux/module.h>    // Needed by all modules
#include <linux/kernel.h>    // Needed for KERN_ALERT
#include <linux/init.h>      // Needed for the macros
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Javier Bustos");

static int hello_init(void)
{

```

```

    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, world\n");
}

module_init(hello_2_init);
module_exit(hello_2_exit);

```

Device Drivers

Una clase del módulo es el driver de dispositivo, que proporciona la funcionalidad para el hardware como una tarjeta de la TV o un puerto serial (también los hay no asociados a hardware, como las tareas de sistemas operativos). En unix, cada dispositivo es representado por un archivo situado en `/dev`, y proporciona los medios de comunicación con el hardware. El driver de dispositivo proporciona la comunicación a nombre de un programa de usuario.

Números Major y Minor

Miremos algunos archivos del dispositivo. Aquí están los archivos del dispositivo que representan las primeras tres particiones en el HDD:

```

# ls -l /dev/hda[1-3]
brw-rw---- 1 root disk 3, 1 Jul 5 2000 /dev/hda1
brw-rw---- 1 root disk 3, 2 Jul 5 2000 /dev/hda2
brw-rw---- 1 root disk 3, 3 Jul 5 2000 /dev/hda3

```

En la columna de los números separados por una coma, el primer número se llama el número *major* del dispositivo. El segundo número es el *minor*. El primero indica qué *driver* se utiliza para el hardware y el segundo se utiliza para distinguir todos los dispositivos de ese hardware que es manejado por el mismo *driver*. En el ejemplo anterior, el driver de manejo de HDD es uno sólo, y para él cada partición es un HDD distinto.

Los dispositivos se dividen en dos tipos: dispositivos en modo carácter y dispositivos en modo bloque. La diferencia es que los dispositivos en modo bloque tienen un *buffer* para las peticiones, así que pueden elegir la mejor orden en la cual responder a ellas. Esto es importante en el caso de dispositivos de almacenaje, donde es más rápido leer o escribir los sectores cercanos que los separados. Otra diferencia es que los dispositivos en modo bloque pueden aceptar solamente la entrada y la salida en bloques (cuyo tamaño puede variar según el dispositivo), mientras que los dispositivos en modo carácter permiten utilizar más o menos bytes según se desee.

Usted puede decir si un archivo del dispositivo es para un dispositivo en modo bloque o un dispositivo en modo carácter mirando el primer carácter en la salida de `ls -l`. Si es `b` entonces es un dispositivo en modo bloque, y si es `c` entonces es un dispositivo en modo carácter. Los dispositivos que usted ve arriba son dispositivos en modo bloque. Los siguientes (los puertos seriales) son de carácter:

```
crw-rw----  1 root  dial  4,  64 Feb 18 23:34 /dev/ttyS0
crw-r-----  1 root  dial  4,  65 Nov 17 10:26 /dev/ttyS1
crw-rw----  1 root  dial  4,  66 Jul  5  2000 /dev/ttyS2
crw-rw----  1 root  dial  4,  67 Jul  5  2000 /dev/ttyS3
```

Cuando el sistema fue instalado, todos esos *device driver* fueron creados por el comando `mknod`. Para crear un nuevo dispositivo llamado *cafe* (dispositivo que controla la máquina de café del DCC) con *major/minor* 12 y 2, llame simplemente a `mknod /dev/cafe c 12 2`. En todo caso, cuando un *device driver* es utilizado, el núcleo usa sólo el *major* del archivo para determinar el *driver* a utilizar. Es el *driver* quien debe preocuparse de administrar los *minor* y utilizarlos para distinguir entre los diversos dispositivos de hardware.

Character Device Drivers

Las operaciones a realizar sobre un CDD están definidas en una estructura llamada `file_operations` y definida en `linux/fs.h`. Esta estructura le indica al compilador cuál de las funciones creadas para el módulo deben asignarse a las funciones del *device*. Por ejemplo:

```
struct file_operations fops = {
    read: device_read,
    write: device_write,
    open: device_open,
    release: device_release
};
```

Asigna las funciones `device_` a las operaciones de lectura, escritura, abrir y liberar. No todos los módulos utilizan las 4 funciones, si ese es el caso, la no utilizada se marcará como `NULL`. Jamás olvidar registrar el dueño del *device* con la macro `SET_MODULE_OWNER(&fops)`.

Para registrar el módulo de un dispositivo en el núcleo (es decir, crear el `/dev/algo`), se utiliza la función:

```
int register_chrdev(unsigned int major, const char *name,
    struct file_operations *fops);
```

Donde `unsigned int major` es *major* que uno desea (0 implica un número dinámico), `const char *name` es el nombre del dispositivo tal y como aparecerá en `/proc/devices` y `struct file_operations *fops` es un puntero a la tabla `file_operations` para su *driver*. Un valor de retorno negativo indica que el registro ha fallado.

Para desregistrarlo, deberíamos primero asegurarnos que nadie lo esté utilizando, para eso se utilizan las macros definidas en `linux/modules.h`, que sirven para aumentar, decrementar y leer el número de procesos que utilizan el módulo, las macros son:

- `MOD_INC_USE_COUNT` : incrementa el contador
- `MOD_DEC_USE_COUNT` : decrementa el contador
- `MOD_IN_USE` : informa el valor del contador

Cada vez que el kernel llama a un módulo de dispositivo, le indica cuál de éstos es el que ha producido (o al que va) la llamada. El par mayor/minor reside combinado en el campo `i_rdev` de la estructura `inode` (representación en el kernel de un archivo en disco). Este campo es del tipo `kdev_t` (caja negra en los kernels linux) y las macros que se utilizan para obtener (y crear) los valores son:

- `MAJOR(kdev_t dev);`
- `MINOR(kdev_t dev);`
- `MKDEV(int ma, int mi);`

struct file

La segunda estructura más importante en la programación de módulos para el núcleo es `struct file`, la cual se encuentra definida en `linux/fs.h`. Es el equivalente en kernel a la estructura `FILE`, y representa un “archivo” (abstracto, no físico) abierto. Sus campos más importantes son:

- `mode_t f_mode`: indica si el archivo es apto para lectura y/o escritura, indicado en los bits `FMODE_READ` y `FMODE_WRITE` respectivamente.
- `loff_t f_pos`: posición actual en el archivo
- `unsigned int f_flags`: flags tipo `O_RDONLY`, `O_NONBLOCK` y `O_SYNC`. Un driver debiera chequearlas en operaciones no bloqueantes.
- `struct file_operations *f_op`: Operaciones asociadas a ese archivo.
- `void *private_data`: El llamado de sistema `open` pone este puntero `NULL` justo antes del llamado a la función `open` desde el driver. Así, el módulo puede hacer uso de este puntero o ignorarlo
- `struct dentry *f_dentry`: Puntero a la estructura de directorio asociada a este archivo.

Semáforos

Para el manejo de secciones críticas en este nivel se utilizan semáforos del kernel, los cuales se definen en `asm/semaphore.h` mediante la estructura `struct semaphores`. El uso es similar a lo visto a lo largo del curso y la API es:

- `sema_init(sem, val)`: Inicializa el semáforo `sem` con `val` tickets.
- `int down_interruptible(&sem)`: Verifica si el número de tickets del semáforo `sem` es mayor que 0, en caso de ser cierto toman un ticket y retorna, en caso contrario el proceso dormirá y le dará paso a otros procesos. El valor de retorno es 0 en caso de ejecución sin problemas y distinto de 0 en caso de algún error o interrupción.
- `up(&sem)`: Devuelve un ticket al semáforo `sem`.

Memoria

Para pedir memoria utilice, similarmente a malloc, la función `kmalloc(int size, int flags)`, donde flags puede ser el conjunto de los siguientes valores definidos en `linux/mm.h`:

- `GFP_KERNEL` : Petición normal de memoria
- `GFP_BUFFER` : Usado para la administración de un buffer cache.
- `GFP_ATOMIC` : Usado para pedir memoria de handlers y otras funciones fuera del contexto del programa.
- `GFP_USER` : Pide más memoria para el usuario (no para el módulo)

Similarmente, se utiliza `kfree` para liberar la memoria.

El gran ejemplo

```
/* chardev.c: Creates a read-only char device that says how many times
 * you've read from the dev file */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h> /* for put_user */

int init_module(void);
void cleanup_module(void);
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 *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev" /* Dev name as it appears in /proc/devices */
#define BUF_LEN 80 /* Max length of the message from the device */

/* Global variables are declared as static, so are global within the file. */

static int Major; /* Major number assigned to our device driver */
static int Device_Open = 0; /* Is device open? Used to prevent multiple
                             access to the device*/
static char msg[BUF_LEN]; /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops =
{
read : device_read,
```

```

write : device_write,
open : device_open,
release : device_release
};

int init_module(void)
{
Major = register_chrdev(0, DEVICE_NAME, &fops);

if (Major < 0) {
printk ("Registering the character device failed with %d\n", Major);
return Major;
}

printk("<1>I was assigned major number %d. To talk to\n", Major);
printk("<1>the driver, create a dev file with\n");
printk("'mknod /dev/hello c %d 0'.\n", Major);
printk("<1>Try various minor numbers. Try to cat and echo to\n");
printk("the device file.\n");
printk("<1>Remove the device file and module when done.\n");

return 0;
}

void cleanup_module(void)
{
/* Unregister the device */
int ret = unregister_chrdev(Major, DEVICE_NAME);
if (ret < 0) printk("Error in unregister_chrdev: %d\n", ret);
}

/* Called when a process tries to open the device file, like
* "cat /dev/mycharfile"
*/
static int device_open(struct inode *inode, struct file *file)
{
static int counter = 0;
if (Device_Open) return -EBUSY;
Device_Open++;
sprintf(msg, "I already told you %d times Hello world!\n", counter++);
msg_Ptr = msg;
MOD_INC_USE_COUNT;

return SUCCESS;
}

```



```

/* Called when a process closes the device file. */
static int device_release(struct inode *inode, struct file *file)
{
    Device_Open --;    /* We're now ready for our next caller */

    /* Decrement the usage count, or else once you opened the file, you'll
    never get get rid of the module. */
    MOD_DEC_USE_COUNT;

    return 0;
}

/* Called when a process, which already opened the dev file, attempts to
read from it. */
static ssize_t device_read(struct file *filp,
char *buffer,    /* The buffer to fill with data */
size_t length,   /* The length of the buffer */
loff_t *offset) /* Our offset in the file */
{
    /* Number of bytes actually written to the buffer */
    int bytes_read = 0;

    /* If we're at the end of the message, return 0 signifying end of file */
    if (*msg_Ptr == 0) return 0;

    /* Actually put the data into the buffer */
    while (length && *msg_Ptr) {

        /* The buffer is in the user data segment, not the kernel segment;
        * assignment won't work. We have to use put_user which copies data from
        * the kernel data segment to the user data segment. */
        put_user(*(msg_Ptr++), buffer++);

        length--;
        bytes_read++;
    }

    /* Most read functions return the number of bytes put into the buffer */
    return bytes_read;
}

/* Called when a process writes to dev file: echo "hi" > /dev/hello */

```

```
static ssize_t device_write(struct file *filp,
const char *buff,
size_t len,
loff_t *off)
{
printk (<1>Sorry, this operation isn't supported.\n");
return -EINVAL;
}
```

Nota

Si desean profundizar aún más sus conocimientos, por favor dirigirse a:

<http://www.xml.com/ldd/chapter/book/index.html>

o bien a:

<http://jamesthornton.com/linux/lkmpg/index.html>