Programación de Controladores para Linux

Edgardo E. Hames Julio A. Bianco Franco M. Luque

2013-09-09

1. Introducción

Linux es un clon del sistema operativo Unix, escrito desde cero por Linus Torvalds con ayuda de un grupo lejano de *hackers* en la Red. Linux tiene todas las características de un sistema operativo moderno incluyendo multitarea, memoria virtual, bibliotecas compartidas, carga en demanda, una correcta administración de la memoria y soporte de red para IPv4 e IPv6 entre otros protocolos. La mayor parte de Linux es independiente del hardware donde se ejecuta. Sin embargo, para cada dispositivo soportado por Linux, alguien ha escrito el correspondiente controlador que lo hace interactuar con el resto del sistema. Sin estos controladores, ningún sistema es capaz de funcionar.

Los controladores de dispositivos (*device drivers*) desempeñan un papel muy importante en el núcleo de Linux: son las cajas negras que hacen que cierto hardware responda a una interfaz bien definida de software y ocultan completamente los detalles de cómo funciona el dispositivo. Las actividades desde el espacio de usuario se realizan por medio de un conjunto estandarizado de llamadas al sistema que son independientes del controlador: **el rol del controlador es asociar esas llamadas a las operaciones específicas del hardware**. Esta interfaz de programación es tal que los controladores pueden ser construidos en forma separada del resto del núcleo, y enlazados y activados en tiempo de ejecución cuando sean necesarios. Un controlador de Linux programado para ser cargado y activado sobre un núcleo activo se denomina "módulo".

El propósito de este material es presentar una introducción sobre cómo desarrollar controladores para el núcleo de Linux enfatizando la construcción de módulos.

2. Utilidades

Cuando trabajamos con módulos para el núcleo de Linux, el conjunto de utilidades *modutils* nos permiten manipularlos desde su compilación hasta que deseemos removerlos del sistema. Aquí se presenta una breve reseña de cada una, pero se recomienda al lector que consulte las páginas del manual de cada una de estas aplicaciones.

- depmod: Crea una dependencia intermodular al estilo de Makefile, basado en los símbolos que encuentra en los módulos mencionados en la línea de comandos o en los directorios especificados en el archivo de configuración. Este archivo es utilizado por *modprobe* para cargar la pila correcta de módulos.

  -a Buscar los módulos en todos los directorios especificados en el archivo de configuración */etc/modules.conf*.

  -e Muestra los símbolos no resueltos de cada módulo.

  -n Escribe el archivo de dependencia en la salida estándar en vez de en el árbol de */lib/modules*.

- modinfo: Muestra información sobre un módulo.

  -a Muestra el autor del módulo.

  -d Muestra la descripción del módulo.

  -l Muestra la licencia del módulo.

  -p Muestra los parámetros del módulo.

  -n Muestra el path completo del archivo que corresponde al módulo.

- lsmod: Muestra la lista de módulos cargados. Esta información se obtiene de */proc/modules*.

- insmod: Instala un módulo en el núcleo en ejecución.

  -f Carga el módulo aunque pertenezca a una versión distinta del núcleo.

  -p Prueba si el módulo puede ser cargado.

- rmmod: Desinstala un módulo del núcleo en ejecución.

- modprobe: Instala o desinstala módulos del núcleo en ejecución.

  -r Descarga un módulo y todos los que lo referencian.

- dmesg: Permite examinar los mensajes del núcleo. Los dos usos más frecuentes son:

```
[usuario@localhost]$ dmesg > boot.messages
[usuario@localhost]$ dmesg | less
```

3. Construcción de Módulos

¡Ya es hora de comenzar a programar! En esta sección se presentan conceptos sobre módulos y programación del núcleo de Linux 2.6. Mostraremos el código de un módulo completo (aunque poco útil) y veremos el código que comparten muchos módulos.

3.1. Preparación del Sistema

La construcción de módulos para el núcleo de Linux 2.6 requiere que tenga el árbol de fuentes de un núcleo configurado y construido en su sistema. Este requerimiento es un cambio desde versiones anteriores del núcleo en las cuales alcanzaba con tener los encabezados que correspondieran a la versión en uso. Los módulos del núcleo 2.6 se enlazan con archivos objetos encontrados en árbol de fuentes del núcleo.

En los sistemas GNU/Linux basados en Debian, esto significa instalar el paquete
`linux-headers-generic` o `linux-kernel-headers`.

3.2. Primer Módulo

Un módulo para el núcleo de Linux agrega una funcionalidad al sistema, la cual puede corresponder al controlador de un componente de hardware o no. Es importante notar este punto ya que nuestro primer ejemplo es una clara muestra de que no es necesario que un módulo haga algo con el hardware:

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

MODULE_LICENSE("GPL")

static int hello_init(void)
{
        printk(KERN_INFO "Hello, world\n");
        return 0;
```

```
}

void hello_exit(void) {
        printk(KERN_INFO "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Este módulo define dos funciones, una para ser invocada cuando se carga y activa el módulo (*hello_init*) y otra para cuando el módulo es removido y desactivado (*hello_exit*). Las macros *module_init* y *module_exit* sirven para indicar el rol de estas dos funciones. La macro MODULE_LICENSE es usada para indicar que este módulo tiene una licencia libre; sin esta declaración, el núcleo se mancha (*taint*) con código que no tiene licenciamiento adecuado cuando se carga el módulo.

La función señalada por *module_init* debe contener todo el código de inicialización del módulo. Como ya veremos más adelante, en caso de que estemos implementando un controlador, aquí debemos realizar la registración del dispositivo.

La función indicada por *module_exit* debe contener el código de terminación del módulo. La posibilidad de descargar un módulo con el sistema en ejecución es una de las características más apreciadas por los desarrolladores, ya que reduce el tiempo de desarrollo; uno puede probar distintas versiones del controlador sin necesidad de reiniciar el equipo cada vez que se realiza una nueva versión.

La función *printk* está definida en el núcleo de Linux y se comporta de manera similar a la función *printf* de la biblioteca estándar de C. La constante *KERN_INFO* indica la prioridad del mensaje.

¿Para qué necesitamos otra implementación de una función que ya está en la biblioteca de C? Dado que el núcleo se ejecuta por sí mismo y sin la ayuda de bibliotecas externas, entonces debe definir todas las funciones que le hagan falta. Por lo tanto, al no estar enlazado con ninguna biblioteca, el código fuente de los módulos *nunca* debe incluir los archivos de cabecera comunes.

Ahora podremos probar el módulo utilizando las utilidades *insmod* y *rmmod*. Notar que sólo el superusuario puede cargar y descargar módulos.

```
[usuario@localhost]$ make
make -C /lib/modules/2.6.20-15-generic/build M=/path/to/src modules
make[1]: Entering directory ‘/usr/src/linux-headers-2.6.20-15-generic’
 CC [M]  /path/to/src/hello.o
```

```
 Building modules, stage 2.
 MODPOST 1 modules
 CC      /path/to/src/hello.mod.o
 LD [M]  /path/to/src/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-2.6.20-15-generic'

[root@localhost]# insmod ./hello.ko
Hello, World
[root@localhost]# rmmod hello
Goodbye, cruel world
```

El lector atento descubrirá que hemos hecho uso de la herramienta *make* para poder construir el módulo. En la próxima subsección encontrará los detalles sobre cómo escribir Makefiles para el núcleo y sus módulos. Para concluir, en los cuadros 1, 2, 3 y 4 se presenta un resumen de las funciones usadas hasta el momento.

Cuadro 1: función de incialización

Función	`static int <nombre a elección>(void)`
Sinopsis	Punto de entrada de módulos del núcleo.
Descripción	Esta función debe estar implementada en todo módulo y define el punto de entrada al realizarse su carga. En los controladores es la encargada de inicializar el dispositivo que maneja.
Retorna	En caso de éxito: 0. En caso de error: Código de error apropiado.

Cuadro 2: función de finalización

Función	`static void <nombre a elección>(void)`
Sinopsis	Punto de salida de módulos del núcleo.
Descripción	Esta función debe estar implementada en todo módulo y define el punto de salida al realizarse su descarga. En los controladores es la encargada de realizar el proceso de terminación del dispositivo que maneja.
Retorna	Nada

3.3. Makefiles para Módulos

El proceso de construcción de un módulo difiere significativamente de la construcción de un programa tradicional. El núcleo es un programa autónomo

Cuadro 3: module_init

Macro	`module_init`
Sinopsis	Señala el punto de entrada de módulos del núcleo.
Retorna	Nada

Cuadro 4: module_exit

Macro	`module_exit`
Sinopsis	Señala el punto de salida de módulos del núcleo.
Retorna	Nada

con ciertos requerimientos sobre cómo juntar todas sus piezas. El nuevo sistema de construcción es más sencillo y simple que la versión 2.4. Mostramos a continuación un Makefile de ejemplo, apto para módulos con uno o más archivos objeto.

```
KERNELDIR ?= "/lib/modules/$(shell uname -r)/build"

obj-m := hello.o
hello-objs := hello.o  # Agregar otros archivos objeto

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

Este archivo Makefile es muy flexible y fácilmente reusable ya que no tiene dependencias sobre particularidades del sistema donde se invoca.

3.4. Parámetros de Módulos

Así como los programas de espacio usuario pueden tomar parámetros para modificar su comportamiento, el núcleo de Linux permite que los módulos también sean ajustados mediante el paso de parámetros durante su carga (con *insmod* o *modprobe*):

```
[root@localhost]$ insmod mimodulo count=42
```

En el ejemplo anterior, el parámetro *count* del modulo *mimodulo* tomará el valor 42. En el código, ese parámetro se declarará agregando estas líneas (además de <*linux/stat.h*>):

```
static int count = 1; /* valor por defecto */
module_param(count, int, S_IRUGO);
                        /* permiso de lectura para todo el mundo */
```

3.5. Creación de Dispositivos

Los dispositivos que el núcleo controla pueden corresponderse con artefactos físicos o lógicos. Sin embargo, todos están mapeados a algún nodo o archivo especial normalmente en el directorio */dev*. Por ejemplo, */dev/hda* corresponde a la unidad de disco duro IDE master primaria del equipo y */dev/urandom* es un dispositivo virtual que nos permite obtener números aleatorios generados a partir del *"ruido"* del equipo.

Mediante el comando *ls* podemos ver los atributos de estos dispositivos:

```
[usuario@localhost]$ ls -l /dev/hda /dev/urandom
brw------- 1 root root 3, 0 2007-09-26 08:45 /dev/hda
crw-r--r-- 1 root root 1, 9 2007-09-26 08:45 /dev/urandom
```

En la primera columna podemos observar los permisos de acceso de los archivos y notamos una primer diferencia entre ambos. En el caso de */dev/hda* hay una *b* que nos indica que se trata de un dispositivo de bloques; la *c* en */dev/urandom* nos dice que es un dispositivo de caracteres.

En la quinta columna, encontramos lo que se ha dado en llamar el *major number* de un dispositivo. Este número es el que le indica al núcleo cuál controlador es el encargado de manejar este dispositivo. Cada controlador en el sistema registra un *major number* y el núcleo se encarga de llevar una tabla que los asocia. Así, el sistema sabe fácilmente qué operaciones se pueden realizar sobre un dispositivo.

Finalmente, en la sexta columna se halla el *minor number* del dispositivo. Este número indica la instancia del dispositivo que se está accediendo. Por ejemplo, dos particiones de un disco duro son manejadas por el mismo controlador, pero son distintas instancias del dispositivo:

```
[usuario@localhost]$ ls -l /dev/hda*
brw------- 1 root root 3, 1 2007-09-26 08:45 /dev/hda1
brw------- 1 root root 3, 2 2007-09-26 08:45 /dev/hda2
```

Para crear las entradas en */dev* se utiliza el programa *mknod*. Su uso se da de la siguiente manera:

```
[usuario@localhost]$ mknod -m 0666 /dev/nombre c M m
```

El modificador *-m* indica el modo de acceso; la *c*, que es dispositivo de caracteres. *M* y *m* son el *major* y *minor* del dispositivo respectivamente.

Se puede encontrar una lista completa de la lista de dispositivos y sus *major* y *minor numbers* consultando */usr/src/linux/Documentation/devices.txt*. El listado de los *majors* de dispositivos de bloques y de caracteres que está utilizando actualmente el sistema está en */proc/devices*.

4. Dispositivos de Caracteres

El núcleo de Linux brinda una interfaz de programación que debe ser implementada por cada controlador de un dispositivo de caracteres. Así, los controladores se comportan como verdaderos tipos abstractos y pueden interactuar con el resto del sistema con gran flexibilidad y extensibilidad.

El desarrollador deberá asociar el número *major* del dispositivo en */dev* a una estructura *file_operations* con punteros a cada una de las funciones que soporta el controlador. Dicha asociación se debe realizar al inicializar el controlador con `register_chrdev` (cuadro 5). Cuando el controlador se descarga (con `unregister_chrdev`, cuadro 6), es necesario deshacer esta asociación para que el núcleo no intente invocar funciones que ya no están disponibles.

En los cuadros 7, 8, 9 y 10 vemos las funciones que nos permiten acceder al dispositivo y liberarlo (`open` y `release` resp.), así como las que permiten enviar y recibir datos (`write` y `read` resp.).

5. Espacio de Usuario vs. Espacio de Núcleo

Es muy importante resaltar que el argument `buf` de los métodos `read` y `write` es un puntero de espacio usuario y no puede ser desreferenciado directamente por el núcleo.

Las direcciones de memoria del núcleo están protegidas del alcance de los programas en espacio usuario. Por lo tanto, para leer y escribir datos desde estos programas es necesario usar un par de macros que hacen esta tarea posible: `copy_from_user` (cuadro 11) y `copy_to_user` (cuadro 12).

6. Memoria Dinámica en Espacio de Núcleo

Para el manejo de memoria dinámica en espacio de núcleo se usan las funciones `kmalloc` y `kfree` descriptas en los cuadros 13 y 14.

Cuadro 5: register_chrdev

Función	`int register_chrdev(unsigned int major, const char *name, struct file_operations *fops)`
Cabecera	`#include <linux/fs.h>`
Sinopsis	Realiza el registro de un dispositivo de caracteres.
Descripción	Esta función debe ser llamada por todo controlador para realizar el registro de un dispositivo de caracteres. *major* es el número mayor del dispositivo que se quiere controlar (entre 0 y 255). Si es 0, el número es asignado dinámicamente por el núcleo. *name* es el nombre del dispositivo como aparecerá en */proc/devices*. *fops* es la estructura que contiene referencias a las operaciones que se pueden realizar sobre el dispositivo.
Retorna	En caso de éxito: 0 si *major* es distinto de 0; número de *major* en caso contrario. En caso de error: -EBUSY si *major* ya ha sido solicitado por otro controlador ó -EINVAL si no es número *major* válido.

Cuadro 6: unregister_chrdev

Función	`void unregister_chrdev(unsigned int major, const char *name)`
Cabecera	`#include <linux/fs.h>`
Sinopsis	Realiza el desregistro de un dispositivo de caracteres.
Descripción	Esta función debe ser llamada por todo controlador para realizar el desregistro de un dispositivo de caracteres. *major* es el número mayor del dispositivo controlado (el mismo que se pasó o se obtuvo en *register_chrdev*). *name* es el nombre del dispositivo como aparecía en */proc/devices*.
Retorna	Nada

Cuadro 7: open

Función	`int open(struct inode *ip, struct file *fp)`
Cabecera	`#include <linux/fs.h>`
Sinopsis	Apertura de un dispositivo.
Descripción	Aún cuando es la primera operación que se realiza sobre un dispositivo, su implementación no es obligatoria por parte de un controlador. En caso de no estar definida, el controlador no es avisado sobre la apertura del dispositivo, pero ésta se realiza en forma satisfactoria.
Retorna	En caso de éxito: 0. En caso de error: Código de error apropiado.

Cuadro 8: release

Función	`int release(struct inode *ip, struct file *fp)`
Cabecera	`#include <linux/fs.h>`
Sinopsis	Liberación de un dispositivo.
Descripción	Esta operación se invoca una vez que se libera el dispositivo. Es decir, que no es llamada cada vez que un programa ejecuta la llamada al sistema *close*. Cada vez que una estructura es compartida (por ejemplo, después de un *fork* o *dup*), *release* no será invocado hasta que todas las copias estén cerradas. En caso de no estar definida, el controlador no es avisado sobre el cierre del dispositivo, pero ésta se realiza en forma satisfactoria.
Retorna	En caso de éxito: 0. En caso de error: Código de error apropiado.

Cuadro 9: read

Función	`ssize_t read(struct file *fp, char *buf, size_t length, loff_t *offset)`
Cabecera	`#include <linux/fs.h>`
Sinopsis	Obtiene datos desde un dispositivo.
Descripción	Este método es llamado cada vez que se intenta leer datos desde un dispositivo. Los datos leídos deben ser copiados en *buf*. Se debe prestar especial atención en esto, ya que es un puntero a una dirección de memoria en espacio usuario. *length* es la cantidad de bytes que deben leerse y *offset* el desplazamiento desde el inicio del archivo. Si el método no se define, al ser invocado se retorna -EINVAL.
Retorna	En caso de éxito: La cantidad de bytes leídos ó 0 para indicar EOF. En caso de error: Código de error apropiado.

Cuadro 10: write

Función	`ssize_t write(struct file *fp, const char *buf, size_t length, loff_t *offset)`
Cabecera	`#include <linux/fs.h>`
Sinopsis	Escritura en un dispositivo.
Descripción	Esta operación envía datos al dispositivo. Los datos a escribir están la dirección apuntada por *buf*. Se debe prestar especial atención en esto, ya que es un puntero a una dirección de memoria en espacio usuario. *length* es la cantidad de bytes que deben escribirse y *offset* el desplazamiento desde el inicio del archivo. Si el método no se define, al ser invocado se retorna -EINVAL.
Retorna	En caso de éxito: La cantidad de bytes escritos ó 0 para indicar EOF. En caso de error: Código de error apropiado.

Cuadro 11: copy_from_user

Función	`unsigned long copy_from_user(void *to, const void *from, unsigned long count)`
Cabecera	`#include <linux/uaccess.h>`
Sinopsis	Copia memoria de espacio usuario a espacio núcleo.
Descripción	Esta función se comporta como *memcpy* y copia memoria de espacio usuario (*from*) a espacio núcleo (*to*) *count* bytes. Si el puntero *to* es una referencia inválida, no se realiza copia alguna. Si durante la copia se encuentra una referencia inválida, se devuelve la cantidad de bytes que falta leer.
Retorna	En caso de éxito: 0 En caso de error: La cantidad de bytes que restan por leer.

Cuadro 12: copy_to_user

Función	`unsigned long copy_to_user(void *to, const void *from, unsigned long count)`
Cabecera	`#include <linux/uaccess.h>`
Sinopsis	Copia memoria de espacio núcleo a espacio usuario.
Descripción	Esta función se comporta como *memcpy* y copia memoria de espacio núcleo (*from*) a espacio usuario (*to*) *count* bytes. Si el puntero *to* es una referencia inválida, no se realiza copia alguna. Si durante la copia se encuentra una referencia inválida, se devuelve la cantidad de bytes que falta escribir.
Retorna	En caso de éxito: 0 En caso de error: La cantidad de bytes que restan por escribir.

Cuadro 13: kmalloc

Función	`void *kmalloc(size_t size, int flags)`
Cabecera	`#include <linux/slab.h>`
Sinopsis	Reserva un espacio de memoria de tamaño `size`.
Descripción	Esta función se comporta como *malloc* pero trabaja con memoria en espacio de núcleo y tiene un parámetro adicional `flags`. `flags` es un parámetro de bajo nivel. Usaremos `GFP_KERNEL`.
Retorna	En caso de éxito: El puntero a la memoria. En caso de error: NULL

Cuadro 14: kfree

Función	`void kfree(void *ptr)`
Cabecera	`#include <linux/slab.h>`
Sinopsis	Libera un espacio de memoria previamente reservado con `kmalloc`.
Descripción	Esta función se comporta como *free* pero trabaja con memoria en espacio de núcleo.
Retorna	Nada

7. Sincronización de procesos

Para solucionar los problemas que surgen con las condiciones de carrera, el núcleo nos provee de semáforos. Los semáforos de Linux tienen el tipo `struct semaphore`, definido en <`linux/semaphore.h`>. Los semáforos deben inicializarse antes de su uso pasando un valor numérico a `sema_init` (cuadro 15). Un controlador sólo debe acceder a su estructura utilizando las primitivas provistas.

Si bien la primitiva *P* o *DOWN* está implementada con varios comportamientos, nosotros sólo veremos la función `down_interruptible` (cuadro 16) que permite al proceso ser interrumpido mientras lleva a cabo la operación. Si se interrumpe, el proceso no habrá adquirido el semáforo y entonces no será necesario ejecutar *V* o *UP*. Se puede usar de la siguiente manera:

```
if (down_interruptible(&s)) {
        /* La llamada fue interrumpida */
        return -ERESTARTSYS;
}
```

La operación *V* o *UP* está implementada con un nombre realmente muy sugestivo, `up`, y su comportamiento es el tradicional (cuadro 17).

Cuadro 15: sema_init

Función	`void sema_init(struct semaphore *sem, int val)`
Cabecera	`#include <linux/semaphore.h>`
Sinopsis	Inicializa un semáforo.
Descripción	Inicializa el semáforo *sem* con el valor *val*.
Retorna	Nada

Cuadro 16: down_interruptible

Función	`int down_interruptible(struct semaphore *sem)`
Cabecera	`#include <linux/semaphore.h>`
Sinopsis	Operación P interrumpible.
Descripción	Este método realiza la operación P sobre el semáforo *sem* y permite que éste reciba señales en el transcurso de la llamada.
Retorna	En caso de éxito: 0. En caso de error: -EINTR (llamada interrumpida).

Cuadro 17: up

Función	`void up(struct semaphore *sem)`
Cabecera	`#include <linux/semaphore.h>`
Sinopsis	Operación V.
Descripción	Este método realiza la operación V sobre el semáforo *sem*.
Retorna	Nada

Referencias

[1] Kwan Lowe, "*Kernel Rebuild Guide*", 2004.

[2] Jonathan Corbet, Alessandro Rubini y Greg Kroah-Hartman, "*Linux Device Drivers*", 3rd Edition, O'Rielly, 2005.