

Programación de módulos del kernel

ADIÓS A LOS MONOLITOS

Originalmente el kernel era monolítico, es decir, todas las funcionalidades residían en un gran todo indivisible. Esto lo hacía más pesado y lento. Poco a poco se fue pasando a un desarrollo modular de los componentes, haciendo el núcleo lo más ligero posible.

POR DAVID SANTOS

En sus inicios el kernel era monolítico, es decir, que se nos obligaba a recompilarlo completamente cada vez que se requería algún tipo de función o controlador para un dispositivo nuevo. Hoy en día eso no es así, prefiriéndose un diseño modular. El uso de un kernel modular nos brinda la ventaja de poder añadir y dividir las distintas funcionalidades que necesitamos, a pesar de que siguen existiendo algunos componentes del sistema que deben estar compilados dentro del propio kernel.

Crear un módulo para el kernel no es magia negra ni tan difícil como se cree. A continuación veremos cómo preparar el entorno que necesitamos para poder trabajar y los pasos necesarios para crear un módulo para nuestro kernel. Concretamente nos centraremos en la rama 2.6 del kernel, que es la más usada actualmente.

¿Qué Necesitamos?

Normalmente las distribuciones Linux que utilizan la mayoría de los usuarios traen paquetes precompilados. Esto significa que cuando nuestra distro instala un kernel en el sistema, a pesar de que puede cubrir todas nuestras necesidades gracias a herramientas de las distintas distribuciones para la detección de hardware, en algunos casos puede que necesitemos tener soportado otro com-

ponente u otra funcionalidad que ésta no trae. Si uno es curioso y pretende sacarle el máximo rendimiento a su sistema, es un buen ejercicio compilar el kernel para adaptarlo a las necesidades propias, que seguro no son las mismas del que lo compiló.

Sin embargo, este no es el propósito del presente artículo, sino que lo que pretendemos es explicar cómo es la creación de módulos para el kernel, algo que en realidad no es tan difícil como pueda parecer. Lo principal para poder desarrollar nuestro módulo o poder compilar algún módulo externo es el código fuente del kernel. Éste se encuentra disponible en los respectivos repositorios de cada distribución, aunque es buena idea usar las fuentes oficiales de [1].

Una vez dispongamos de las fuentes del kernel en nuestro sistema, hemos de asegurarnos de que están en el directorio `/usr/src/linux`. Probablemente, cuando descomprimamos el tarball del kernel, nos cree un directorio con un nombre parecido, pero con la versión del kernel al final, es decir, algo como `linux_2-6-22.8`. Copiamos el directorio a `/usr/src/` y hacemos un enlace simbólico que apunte a `/usr/src/linux`:

```
# mv linux_2-6-22.8 /usr/src/
```

```
# ln -s /usr/src/linux-$(uname -r) linux
```

Si estamos usando un kernel precompilado por nuestra distribución hay que descargarse de [1] exactamente la

Listado 1: hola1.c

```
01 #include <linux/module.h>
02 #include <linux/kernel.h>
03
04 int init_module()
05 {
06     printk(KERN_INFO "Hola mundo
07     1.\n");
08     return 0;
09 }
10 void cleanup_module()
11 {
12     printk(KERN_INFO "Adios
13     mundo 1.\n");
14 }
```

Listado 2: Makefile

```
01 obj-m += hola1.o
02 all:
03     make -C
04     /lib/modules/$(shell uname
05     -r)/build M=$(PWD) modules
06 clean:
07     make -C
08     /lib/modules/$(shell uname
09     -r)/build M=$(PWD) clean
```

Listado 3: hola2.c

```

01 #include <linux/module.h>
02 #include <linux/kernel.h>
03 #include <linux/init.h>
04 #define AUTOR "Linux-Magazine
    <info@linux-magazine.es>"
05 #define DESCRIPCION
    "Aprendiendo a programar
    módulos del kernel."
06
07 static int __init
    init_hola2(void)
08 {
09     printk(KERN_INFO "Hola mundo
    2.\n");
10     return 0;
11 }
12 static void __exit
    cleanup_hola2(void)
13 {
14     printk(KERN_INFO "Adios
    mundo 2.\n");
15 }
16 module_init(init_hola2);
17 module_exit(cleanup_hola2);
18
19 MODULE_LICENSE("GPL");
20 MODULE_AUTHOR(AUTOR);
21
22 MODULE_DESCRIPTION(DESCRIPCION);
23
24 MODULE_SUPPORTED_DEVICE("mundo");

```

misma versión de las fuentes del kernel, de lo contrario tendremos muchos problemas.

Una vez realizada la descarga y enlazado el directorio, podemos echar un vistazo dentro para ver los diferentes subdirectorios que contiene. En cada uno de ellos se almacenan distintas partes necesarias para construir el kernel. Para poder construirlo, es decir, compilarlo, necesitaremos un compilador, así que es imprescindible tener GCC instalado en el sistema. Es recomendable usar la versión precompilada que tengamos en los repositorios de la distribución. Seguidamente, con todo instalado, podremos disponernos a crear nuestro módulo.

Relación Amor/Odio

El kernel está situado en un sitio concreto, que claramente es el más privilegiado. Él es el que trabaja directamente con la máquina y sus distintos dispositivos, usando el espacio denominado KernelLand o Kernel Space. Los usuarios no

trabajamos al nivel que opera el kernel, sino que hacemos peticiones al núcleo para que lo haga por nosotros y lleve a cabo las tareas que nosotros no podemos hacer directamente. El espacio que se reserva para los usuarios es conocido como UserLand o User Space.

Aunque las denominaciones importan poco, sí que es importante conocer la diferencia a la que operamos usuario y kernel, ya que en el momento en el que se cometa un fallo desde uno u otro, puede que tengamos que enfrentarnos a unos cuantos errores *kernel panic*.

El Módulo más Simple

Comenzaremos con un módulo extremadamente simple que nos permita saber cómo funciona cualquiera de ellos y cómo cargarlo en nuestro sistema. Haremos uso del famoso "Hola Mundo" para explicar distintos aspectos básicos necesarios para escribir un módulo. Como entorno de desarrollo podemos usar cualquier editor de textos. Si este editor además dispone de coloreado de código,

mejor aún. Creamos el archivo *hola1.c*, cuyo contenido debe ser igual al del Listado 1.

Podemos apreciar que cargamos dos ficheros en la cabecera *module.h* (necesario para todos los módulos) y *kernel.h* (necesario para poder disponer de *KERN_INFO*). Todos los módulos suelen tener por lo menos dos funciones: *init_module()*, llamada cuando el módulo es cargado en el kernel con *insmod*, y una función de finalización, *cleanup_module()*, a la que se llama en el momento que descargamos el módulo con *rmmod*. No es obligatorio usar estas funciones, ya que pueden usarse macros que las llamen. Sin embargo, la mayoría de los desarrolladores siguen haciéndolo. Una pequeña recomendación antes de cargar el módulo o descargarlo es hacer *sync* para volcar los datos en memoria a disco.

Puede que a muchos *printk()* les recuerde a *printf()* del lenguaje C. Esta función no fue creada para comunicar el kernel con el usuario, sino para hacer un registro de los mecanismos del kernel y recoger advertencias. Cada *printk()* viene acompañado de una prioridad. Hay 8 prioridades, y el kernel tiene macros para todas ellas. Si no se especifica un nivel de prioridad, se omite y se usa la prioridad por defecto, *DEFAULT_MESSAGE_LOGLEVEL*. En nuestro caso hemos usado *KERN_INFO*. Como se ha comentado, no nos mostrará directamente nada en pantalla, sino que se registra en *dmesg*. Podremos ver los mensajes haciendo *dmesg | tail*, que nos presenta las últimas líneas del log del kernel. Si estamos usando algún sistema de registros, como puede ser

Tabla 1: Macros y números de prioridad

Mensaje	Nivel	Significado
KERN_EMERGE	<0>	El sistema es inutilizable
KERN_ALERT	<1>	Se debe tomar alguna medida inmediatamente
KERN_CRIT	<2>	Condiciones críticas
KERN_ERR	<3>	Condiciones de error
KERN_WARNING	<4>	Condiciones de advertencia
KERN_NOTICE	<5>	Condición normal, pero significativo
KERN_INFO	<6>	Informativo
KERN_DEBUG	<7>	Mensajes de depuración

Listado 4: Salida de modinfo

```

01 root@satania:~/1km/02# modinfo
    hola2.ko
02 filename:          hola2.ko
03 description:      Aprendiendo a
    programar módulos del kernel.
04 author:           Linux-Magazine
    <info@linux-magazine.es>
05 license:         GPL
06 depends:
07 vermagic:
    2.6.22-gentoo-r5 mod_unload
    PENTIUM4

```

Listado 5: inicio.c

```

01 #include <linux/module.h>
02 #include <linux/kernel.h>
03
04 int init_module()
05 {
06     printk(KERN_INFO
07         "Iniciando...\n");
08     return 0;
09 }

```

Listado 6: fin.c

```

01 #include <linux/module.h>
02 #include <linux/kernel.h>
03
04 void cleanup_module()
05 {
06     printk(KERN_INFO
07         "Cerrando...\n");
08 }

```

Listado 7: Makefile modificado

```

01 obj-m += infin.o
02 startstop-objs := inicio.o
03 fin.o
04 all:
05     make -C
06     /lib/modules/$(shell uname
07     -r)/build M=$(PWD) modules
08 clean:
09     make -C
10     /lib/modules/$(shell uname
11     -r)/build M=$(PWD) clean

```

syslogd, el resultado también se verá reflejado en el fichero `/var/log/messages`. Para nuestro propósito, se puede omitir el uso de `KERN_INFO` y usar el nivel por defecto. Si examinamos el fichero `/usr/src/linux/include/linux/kernel.h`, veremos que muestra todas las prioridades por orden. Un advertencia: nunca se debe usar como prioridad un número directamente, ya que sólo se puede pasar la macro que acompaña el número de la prioridad. Las macros, sus prioridades y sus significados se reseñan en la Tabla 1.

Compilando

Normalmente, cuando hacemos un pequeño programa en C se suele compilar a mano con pocos parámetros. Sin embargo, para compilar nuestro módulo crearemos *Makefile*, mostrado en el Listado 2. ¡Ojo! Debe transcribirse tal y como está en el ejemplo, incluyendo el sangrado que aparece, pues de lo contrario es bastante probable que no funcione (en todo caso, se pueden descargar todos los archivos de este artículo del sitio web de Linux Magazine en [3]). Este tipo de fichero, aunque esté organizado jerárquicamente, puede llegar a ser difícil de mantener.

Una vez tenemos listo nuestro fichero, podemos comenzar a compilar usando *make* directamente desde nuestra terminal hasta obtener el módulo compila debe ser parecida a la de la Figura 1.

Para cargar nuestro módulo bastará con usar el comando *insmod* como usuario root seguido del nombre del

módulo, en este caso *hola1.ko*, y para descargarlo lo mismo, pero con el comando *rmmod*. Si miramos el log del kernel podemos comprobar que nos ha imprimido la cadena que le pasamos anteriormente cuando lo carga y cuando lo descarga.

Macros e Información

Como comentábamos más arriba, se pueden usar macros propios para llamar a las funciones *init_module()* y *cleanup_module()*. En el ejemplo que mostramos en el Listado 3 creamos un fichero llamado *hola2.c* y hacemos uso de las macros, añadiremos además información adicional para el módulo.

La información extra que añadiremos la pasaremos usando cuatro macros definidas en *module.h* que ya tenemos prediseñadas. Éstas nos brindan información sobre quien creó el módulo, una corta descripción, qué dispositivo atiende y, bastante importante, qué licencia usa. En nuestro caso usaremos la licencia GPL. Esta información se extrae ejecutando *modinfo*, cuyo contenido aparece en el Listado 4.

Aunque podría ayudar a una configuración automática de los módulos, la macro *MODULE_SUPPORTED_DEVICE()* está actualmente en desuso, pero aquí la usamos para que nos dé información sobre el dispositivo que usa, que en nuestro caso es */dev/mundo*.

Para compilar el nuevo módulo *hola2.c* procederemos de la misma forma que en el caso del ejemplo anterior usando un archivo *Makefile*, sustituyendo el fichero *hola1.o* por *hola2.o*.

Módulos Multi-archivo

A continuación trataremos de coger el primer ejemplo y partirlo en dos ficheros, e indicaremos al fichero *Makefile* cómo debe enlazarlos para que los compile como un solo módulo.

Crearemos un fichero llamado *inicio.c* y otro llamado *fin.c*, los cuales tendrán el aspecto que presentan en los Listados 5 y 6 respectivamente. Ahora nos centramos en nuestro *Makefile*, en el que eliminaremos la línea *obj-m += hola2.o* hasta dejarlo tal y como aparece en el Listado 7.

Si ahora ejecutamos *make* tendremos de nuevo un módulo funcionando, aunque creado de otra manera. Ni en este

Tabla 2: Permisos sobre sysfs

S_IRWXU	Lectura, escritura, ejecución/búsqueda por parte del dueño
S_IRUSR	Lectura por parte del dueño
S_IWUSR	Escritura por parte del dueño
S_IXUSR	Ejecución/búsqueda por parte del dueño
S_IRWXG	Lectura, escritura, ejecución/búsqueda por parte del grupo
S_IRGRP	Lectura por parte del grupo
S_IWGRP	Escritura por parte del grupo
S_IXGRP	Ejecución/búsqueda por parte del grupo
S_IRWXO	Lectura, escritura, ejecución/búsqueda por parte de otros
S_IROTH	Lectura por parte de otros
S_IWOTH	Escritura por parte de otros
S_IXOTH	Ejecución/búsqueda por parte de otros
S_ISUID	Establecer user-ID en el momento de la ejecución
S_ISGID	Establecer group-ID en el momento de la ejecución
S_ISVTX	Para directorios, flag de restricción de eliminación

ejemplo ni en otro un poco más complicado se puede apreciar su potencial, pero si tenemos la oportunidad de mirar los distintos directorios donde se almacenan los módulos del kernel (aún sin compilar), y nos fijamos en los archivos *Makefile*, observaremos que muchos módulos están divididos en varios ficheros. Si no fuera así, mantenerlos sería una tarea mucho más ardua.

Pasando Argumentos

Los módulos pueden adquirir datos pasándoselos mediante argumentos ya declarados. Pero aquí no funciona lo de *argc/argv*. Para poder pasarle argumentos a un módulo desde la línea de comandos hay que declararlos previamente y usar la macro *module_param()*, definida en *linux/moduleparam.h*, para poder usar este mecanismo.

La macro *module_param()* recoge 3 argumentos: el nombre de la variable, el tipo y los permisos que tiene en *sysfs* (es un sistema de archivos virtual que viene en los kernels de la rama 2.6. Exporta información de dispositivos y drivers del kernel a User Space). Si preferimos usar una matriz para los enteros o para las cadenas, podemos usar *module_param_array()* y *module_param_string()*.

Para ilustrar cómo funciona el paso de argumentos a un módulo, escribiremos uno que acepta como parámetro el nom-

bre de otro módulo y nos indicará si está cargado o no. Algo así como *lsmmod*, pero desde el mismo kernel. Crearemos un archivo llamado

lkm_s.c, que tiene el aspecto que mostramos en el Listado 8. El argumento *S_IRUSR* es el último argumento que recoge

ule_param() e indica los permisos sobre *sysfs*. Con esta cadena en concreto le estamos dando permisos de lectura al propietario. Para otros permisos, ver la Tabla 2.

Editamos el fichero *Makefile* y seguimos el mismo proceso que en el anterior ejemplo, pero utilizando *lkm_s.o*. Compilamos y cargamos el módulo, y, en este caso, *modinfo* nos mostrará mas información. En él aparecerán todos los parámetros declarados con sus valores por defecto. Vamos a cargar el módulo pasándole el único parámetro que hemos añadido. Éste sirve para pasarle a *request_module* la cadena con el nombre

```
sefokuma@satania lkm $ make
make -C /lib/modules/2.6.22-gentoo-r5/build M=/home/sefokuma/lkm modules
make[1]: se ingresa al directorio '/usr/src/linux-2.6.22-gentoo-r5'
CC [M] /home/sefokuma/lkm/hola1.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/sefokuma/lkm/hola1.mod.o
LD [M] /home/sefokuma/lkm/hola1.ko
make[1]: se sale del directorio '/usr/src/linux-2.6.22-gentoo-r5'
sefokuma@satania lkm $
```

Figura 1: Proceso de compilación.

del módulo que queremos comprobar si está cargado. Si ahora miramos el registro, comprobamos que la salida nos dice si el módulo buscado está cargado o no. Para poder usar *request_module* necesitamos añadir en los archivos de cabecera *kmod.h*. Éste es el que comprueba la cadena que le pasamos, de modo que si está cargado devuelve un 0 y está descargado devuelve 256.

Llamada a la Acción

Como hemos podido ver, la escritura de módulos para el kernel en principio no es algo extremadamente complejo, ni entraña magia negra ni nada por el estilo. Y si hay algo de lo que se quejan los creadores y mantenedores del código del kernel es de la poca gente de la que disponen. Así que animamos a todos los programadores a los que les haya picado el gusanillo con este artículo a seguir documentándose y a que se unan al esfuerzo colectivo de mejorar el núcleo de Linux. ■

Listado 8: lkm_s.c

```
01 #include <linux/module.h>          17 int mod=request_module("%s",
02 #include <linux/init.h>           modulo);
03 #include <linux/kernel.h>         18
04 #include <linux/kmod.h>           19 if (mod == 0)
05                                     20     printk(KERN_INFO "El
06 MODULE_LICENSE("GPL");           modulo está cargado.\n");
07 MODULE_AUTHOR("Linux-Magazine     21 else
    <info@linux-magazine.es>");     22     printk(KERN_INFO "El
08 MODULE_DESCRIPTION("Y si no      modulo no está cargado.\n");
    existiese lsmmod...");         23     return 0;
09                                     24 }
10 char *modulo;                      25
11                                     26 static void __exit
12 module_param(modulo, charp,       lkm_s_exit(void)
    S_IRUSR);                       27 {
13 MODULE_PARM_DESC(modulo,         28     printk(KERN_INFO "Fin.\n");
    "Módulo a buscar.");           29 }
14                                     30
15 static int __init                31 module_init(lkm_s_init);
    lkm_s_init(void)                32 module_exit(lkm_s_exit);
16 {
```

RECURSOS

- [1] Fuentes del kernel: <http://www.kernel.org>
- [2] The Linux Kernel Module Programming Guide: <http://en.tldp.org/LDP/lkmpg/>
- [3] Ficheros fuente de este artículo: <http://www.linux-magazine.es/Magazine/Downloads/32/Kernel>
- [4] Documentación muy interesante sobre kernel: <http://jungla.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html>