Práctica 2. Módulos del núcleo

J.C. Pérez, S. Sáez, V. Lorente y L. Pascual

Índice

1. Introducción
2. Los módulos cargables en Linux
    2.1. Programación de los módulos
        2.1.1. Imprimiendo mensajes desde el núcleo de Linux
    2.2. Compilación de los modulos
    2.3. Utilización de los módulos
3. Tarea a realizar
4. Comprobación del funcionamiento
5. Entrega de la práctica
6. Notas
7. Bibliografía

1. Introducción

El núcleo de Linux está organizado siguiendo una arquitectura monolítica, en la cual, todas las partes del núcleo del sistema operativo (sistemas de ficheros, manejadores de dispositivos, protocolos de red, etc.) están enlazadas como una sola imagen (normalmente el fichero **`/vmlinuz`**) que es la que se carga y ejecuta en el arranque del sistema.

Esta estructura podría dar lugar a un sistema poco flexible, ya que cualquier funcionalidad que se le quisiera añadir al núcleo del sistema requeriría una recompilación completa del mismo. Aún así, la filosofía de fuentes abiertos hace Linux mucho más flexible que otros sistemas operativos en la que los fuentes no están disponibles. No obstante, la recompilación total del núcleo puede resultar engorrosa en las fases de desarrollo de nuevos manejadores de dispositivo, ampliaciones no oficiales del núcleo, etc.

Esta limitación desapareció con la incorporación, en la versión 2.0 de Linux, del soporte para la carga dinámica de módulos en el núcleo. Esta nueva característica permite la incorporación en caliente de nuevo código al núcleo del sistema operativo, sin necesidad de reinicializar el sistema.

La práctica aborda el desarrollo de dos pequeños módulos.

2. Los módulos cargables en Linux

Los módulos son "trozos de sistema operativo", en forma de ficheros objeto especiales (**`.ko`**), que se pueden insertar y extraer en tiempo de ejecución. Dichos ficheros **`.ko`** se obtienen como resultado de la compilación con un fichero makefile con una sintaxis especial creado por GNU con el fin de facilitar la tarea de compilar los módulos. Podemos construir un módulo a partir de un fichero **`.o`** incluyendo en la primera línea del makefile '**`obj-m = ejemplo.o`**' o bien un único módulo compuesto de dos ficheros con la línea '**`module-objs-m = ejemplo.o ejemplo2.o`**'. Para compilar dos módulos a la vez basta con poner '**`obj-m = ejemplo.o ejemplo2.o`**'. Algunas características especiales que deben tener estos ficheros, son las de incorporar las macros **`module_init`** y **`module_exit`**. Más adelante veremos su utilidad.

Una vez desarrollado un módulo e insertado en el núcleo, su código pasa a ser parte del propio núcleo, y por lo tanto se ejecuta en el modo supervisor del procesador (nivel de privilegio 0 en la arquitectura i386), con acceso a todas las funciones del núcleo, a las funciones exportadas por módulos previamente insertados, y a todo el hardware de la máquina sin restricciones.

La única diferencia con código enlazado en el núcleo es la posibilidad de extraer el módulo una vez ha realizado su labor o ha dejado de ser útil, liberando así todos los recursos utilizados.

Queremos que el código de un módulo pueda utilizar cualquier función del núcleo, pero este código no ha sido enlazado en tiempo de compilación con el kernel, por lo que las referencias a las funciones del núcleo no están resueltas. Así pues, el proceso de inserción de un módulo debe seguir una serie de pasos:

1. Obtener las referencias a funciones ofrecidas por el módulo.
2. Incorporar dichas referencias al núcleo, como referencias temporales (desaparecerán con la extracción del módulo).
3. Resolver las referencias a las funciones no resueltas en el módulo, ya sean a funciones del núcleo, como a funciones de otro módulos.
4. Insertar el módulo en la zona de memoria correspondiente al núcleo.
5. Finalmente, invocar a la macro **`module_init`** del nuevo módulo.

La extracción de un módulo del núcleo se realiza mediante una secuencia similar a la anterior, pero en orden inverso, donde antes de extraer el módulo se invoca a la función **`module_exit`**.

La construcción, carga y modificación del siguiente ejemplo es una buena experiencia para comprender mejor cómo los módulos interactúan y trabajan con el kernel. Dicho ejemplo es compatible con casi cualquier versión del kernel 2.6.X, incluidos aquellos que distribuyen ciertos vendedores. Aunque es recomendable que usemos uno descargado directamente de http://www.kernel.org o de las principales distribuciones.

2.1. Programación de los módulos

La macro **`module_init`** nos van a permitir inicializar el módulo al insertarlo en el núcleo (equivaldría a la función main de un programa en C). La sintaxis a seguir en el código de nuestros módulos será la siguiente:

```
static int __init mi_funcion(void){

/* Código de inicialización */

}

module_init(mi_funcion)
```

La función de inicialización debe ser declarada "**`static`**", ya que no debe ser vista fuera del fichero. La macro **`__init`** indica al kernel que esta es la función que debe usar sólo en la inicialización del módulo, por lo que es obligatorio incluirla si queremos que funcione (esta macro añade una sección especial en el código objeto del módulo). Una vez ha terminado, los recursos que ha utilizado el sistema para caragar el módulo son liberados para otros usos.

Complementariamente, **`module_exit`** se usará para liberar los recursos utilizados cuando se vaya a extraer. Su sintaxis es la que sigue:

```
static void __exit funcion_salida(void){

/*Código de la función de salida*/

}

module_exit(funcion_salida);
```

El modificador **`__exit`** indica que el código se ejecuta sólo en el momento de la descarga del módulo. La declaración de **`module_exit`** es necesaria para habilitar al kernel para encontrar la función de salida o *limpieza*.

A continuación vamos a ver cómo podemos crear, insertar y extraer un módulo. Todo ello acompañado de algunos ejemplos. Los módulos son una característica opcional de Linux que se elige cuando se compila el núcleo, y por lo tanto, nuestro núcleo debe tenerla activada si queremos utilizar está característica del sistema.

Un módulo se crea a partir de un fuente en "C". A continuación tenemos un módulo mínimo:

Fichero **`ejemplo.c`**:

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif
#ifndef MODULE
#  define MODULE
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
MODULE_LICENSE("Dual BSD/GPL");

int n = 1;
module_param(n, int, S_IRUGO);

static int __init entrando(void) {
  printk(KERN_INFO "Entrando.n=%d\n",n);
  return 0;
}

static void __exit saliendo(void) {
  printk(KERN_INFO "Saliendo.\n");
}

module_init(entrando);
module_exit(saliendo);
```

2.1.1. Imprimiendo mensajes desde el núcleo de Linux

El núcleo no dispone de salida estándar, por lo que no podemos utilizar la función **`printf()`**. A cambio, el núcleo ofrece una versión de ésta, llamada **`printk()`**, que funciona casi igual, a excepción de que el resultado lo imprime sobre un buffer circular de mensajes (*kernel ring buffer*).

En el *kernel ring buffer* es donde se escriben todos los mensajes del núcleo. De hecho, son los mensajes que vemos cuando arranca Linux. En cualquier momento podemos ver su contenido reciente con la orden **`dmesg`** o su contenido inmediato consultando el fichero **`/proc/kmsg`**.

El manejo de los mensajes por el kernel es muy flexible y algo complejo (ver **`klogd(8)`** y **`/usr/src/linux/include/linux/kernel.h`**). Concretamente, el primer parámetro de printk debe ser una de las siguientes constantes, dónde **`<n>`** es un número entre 0 y 7, ambos inclusive, que indican el nivel de prioridad del mensaje.

```
KERN_EMERG   <0> System is unusable
KERN_ALERT   <1> Action must be taken immediately
KERN_CRIT    <2> Critical conditions
KERN_ERR     <3> Error conditions
KERN_WARNING <4> Warning conditions
KERN_NOTICE  <5> Normal but significant condition
KERN_INFO    <6> Informational
KERN_DEBUG   <7> Debug-level messages
```

Normalmente, el núcleo está configurado para mostrar por la consola activa los mensajes de prioridad superior a 6. (Los terminales gráficos no son consolas, a no ser que se hayan lanzado explícitamente como

tales).

2.2. Compilación de los modulos

El fichero **`Makefile` necesario** para compilar el módulo de ejemplo sería:

```
KVERSION = $(shell uname -r)

obj-m = ejemplo.o

all:
        make -C /lib/modules/$(KVERSION)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(KVERSION)/build M=$(PWD) clean
```

El nombre del fichero *makefile* debe ser **`Makefile`**.

La primera línea averigua automáticamente qué versión del núcleo estás ejecutando (**`uname -r`**) y supone que los módulos que vas a compilar son para ese núcleo. Si se compilan los módulos para un núcleo distinto del actual, habría que cambiar esa línea poniendo la versión concreta del núcleo (v.g. **`KVERSION = 2.6.20.3-ubuntu1`**).

La opción "**`-C`**" indica la ejecución del makefile desde el directorio indicado por el primer parámetro, en este caso **`/lib/modules/$(KVERSION)/build`** y compila y crea los módulos cuyos ficheros se encuentran en el directorio indicado por el segundo parámetro, **`$(PWD)`**.

La opción "**`M=`**" indica al compilador que lo que se va a compilar y generar son módulos.

Para compilar los módulos especificados en la variable **`obj-m`** se utilizará la orden **`make all`**.

Si no entiendes algo, pregúntalo, después de pensarlo. ¡¡No te quedes con ninguna duda al respecto!!

Si se está utilizando un núcleo diferente al que viene en la distribución, para que estén disponibles los ficheros de cabecera necesarios para compilar, los fuentes del núcleo deben estar en su sitio (**`/usr/src/linux-num_de_version`**) y se debe haber ejecutado al menos **`make menuconfig`** (grabando la configuración) y **`make depend`**, o haber recompilado el núcleo completo (**`make menuconfig`** y **`make depend bzImage modules modules_install`**).

2.3. Utilización de los módulos

La carga de un módulo se lleva a cabo mediante la orden **`insmod`**, que realizará todas las acciones comentadas antes para insertar el código en el núcleo. Haz, desde una consola:

```
# insmod ejemplo.ko
```

Acabamos de instalar ejemplo y ejecutar su macro **`module_init`**. Si se le pasa a **`insmod`** un nombre de fichero sin ruta, se busca en los directorios estándar (ver **`insmod(8)`**).

La orden **`lsmod`** permite listar los módulos que en un momento dado tenemos instalados:

```
# lsmod
```

Y, finalmente, con **`rmmod`** podemos extraer del núcleo el módulo (el nombre puede no incluir la extensión .ko):

```
# rmmod ejemplo
```

Para pasar parámetros a un módulo, no hay más que asignar valores a las variables globales declaradas como parámetros con la macro **`module_param(nombre, tipo, modo)`**. Como hemos visto en el programa de ejemplo, **`module_param`** recibe como primer argumento el nombre de la variable y como segundo argumento, el tipo. Como tercer parámetro tenemos el modo en que puede ser accedida la variable.

El tipo debe ser uno de los siguientes: **`int`**, **`long`**, **`short`**, **`uint`**, **`ulong`**, **`ushort`**, **`charp`** (puntero a caracter) y **`bool`** (un booleano, cuya variale asociada debe ser de tipo entero).

El modo **`S_IRUGO`** indica que este parámetro será leído por todo el mundo pero nadie puede modificarlo (**R**ead **U**ser, **G**roup y **O**thers).

La definición de **`module_param`** puede consultarse en el fichero **`/usr/src/linux/include/linux/moduleparam.h`**.

La sintaxis es muy sencilla, ya que basta con escribir la asignación como parámetro de **`insmod`**.

Por ejemplo (pruébalo mejor desde una consola):

```
# insmod ejemplo.ko n=4
```

Con **`modinfo -p ejemplo.ko`** podemos averiguar qué parámetros puede recibir el módulo.

3. Tarea a realizar

El objetivo de la práctica consiste en la implementación de dos módulos cargables en el núcleo de Linux. Estos módulos, **acumulador** y **cliente**, deberán comportarse según los criterios siguientes:

- Cada uno de ellos ha de mostrar cuando lo insertemos o extraigamos un mensaje informativo indicando el instante de inserción y de extracción (en número de segundos desde el uno de enero de 1970).
- Podemos obtener el instante actual consultando la funcion **`get_seconds()`**, que nos devuelve como un **`unsigned long`** el campo **`tv_sec`** de la variable **`xtime`** declarada en **`include/linux/time.h`** (no es necesario incluir este fichero). El tipo de esta variable es **`struct timespec`** y está definido también en el mismo fichero.
- El módulo **acumulador** debe definir una función **`void acumular(int i)`**, que reciba un parámetro entero y vaya sumando su valor a una variable global. Esta función debe ser exportable (en C, toda función no static es exportable).
- El módulo **acumulador** también debe ofrecer una función **`int llevamos(void)`**, que devuelva cuánto lleva acumulado.
- El módulo **cliente**, al ser insertado, debe llamar a la función **`acumular()`** del módulo acumulador con un valor a acumular igual al parámetro que le pasemos al módulo cliente al insertarlo.
- El módulo **cliente**, al ser extraido, debe llamar a la función **`llevamos()`** del módulo acumulador e imprimir el resultado en su mensaje de salida.
- El módulo **acumulador**, al ser extraido, también debe imprimir el resultado final de la suma en su mensaje de salida.

Para que el cliente pueda ver las funciones **`llevamos()`** y **`acumular()`**, hemos de usar la macro **`EXPORT_SYMBOL(nombre_del_simbolo_a_exportar)`** en el acumulador.

4. Comprobación del funcionamiento

Comprueba que los módulos implementados se compilan correctamente. Después inserta el módulo acumulador y comprueba que imprime el mensaje inicial.

Extrae el módulo **acumulador** y comprueba el mensaje. Después intenta insertar el **cliente** sin que esté el acumulador insertado.

Comprueba el funcionamiento del conjunto acumulando una serie de valores y extrayendo finalmente el módulo **acumulador** para ver el resultado.

Usa **`lsmod`** cada vez que insertes y extraigas los módulos, para asegurarte de que todo funciona. Trabaja en una consola de texto para ver los mensajes. Imprime éstos con nivel de prioridad menor que 6.

5. Entrega de la práctica

Para entregar la práctica deberéis hacer un fichero *tar* comprimido con **`gzip`** que contenga única y exclusivamente los ficheros fuente (**`cliente.c`** y **`acumulador.c`**) y el fichero **`Makefile`**. Un ejemplo de la orden a utilizar sería:

```
$ tar cvzf practica2.tgz cliente.c acumulador.c Makefile
```

No os preocupéis por el nombre del fichero ya que el sistema de entrega lo renombra.

Para entregar la práctica deberéis rellenar el formulario que se encuentra en la página de prácticas. La clave solicitada debe ser introducida por el profesor de prácticas.

6. Notas

- Podéis guardar vuestros ficheros en memorias USB, ya que sólo es necesario conservar de una sesión a otra los ficheros fuente que hayáis modificado, no el resto de ficheros del núcleo, ni ejecutables, ni módulos compilados (**`.ko`**). Los ficheros modificados siempre serán de un tamaño perfectamente manejable.
- Para manejar la memoria podéis montarla con la orden **`mount(8)`**.
- Recordad siempre borrar todo antes de apagar la máquina.

7. Bibliografía

Para más información, consultar el Capítulo 2 de: Linux Device Drivers (3ª Edición). Disponible en http://lwn.net/Kernel/LDD3/