Seminario. Módulos cargables del kernel (LKM)

Duración: 1 sesión

1. Objetivos del seminario

Los objetivos concretos de este seminario son:

- Aprender cómo funcionan los módulos cargables del kernel (LKM).
- Configurar nuestro sistema para construir, cargar y descargar dichos módulos.
- Comprobar el funcionamiento de un módulo LKM sencillo.

2. Introducción

En computación, un “módulo cargable del núcleo” (*loadable kernel module*, LKM) es un archivo que contiene código objeto que puede extender el kernel en ejecución de un sistema operativo. La mayoría de los sistemas operativos modernos soportan módulos cargables en el kernel, habitualmente llamados "módulos cargables" o "extensión del núcleo". Un ejemplo típico de módulo cargable son los controladores de dispositivo.

Los módulos cargables en el kernel son generalmente utilizados para brindar soporte a nuevos dispositivos de hardware (periféricos) y sistemas de archivos, así como para agregar llamadas al sistema, extendiéndolo. Cuando la funcionalidad provista por un módulo del kernel deja de ser requerida, normalmente éste puede ser descargado, liberando su memoria.

En este seminario queremos escribir un módulo del kernel de Linux muy sencillo que podría dar servicio a un dispositivo (periférico o sistema de archivos en general).

Adicionalmente, un ejemplo avanzado que se sale del ámbito de este seminario podría ser el desarrollo de controladores de dispositivos de caracteres y la creación de aplicaciones en el espacio de usuario que se comuniquen con estos módulos LKM.

3. ¿Qué es un módulo del kernel?

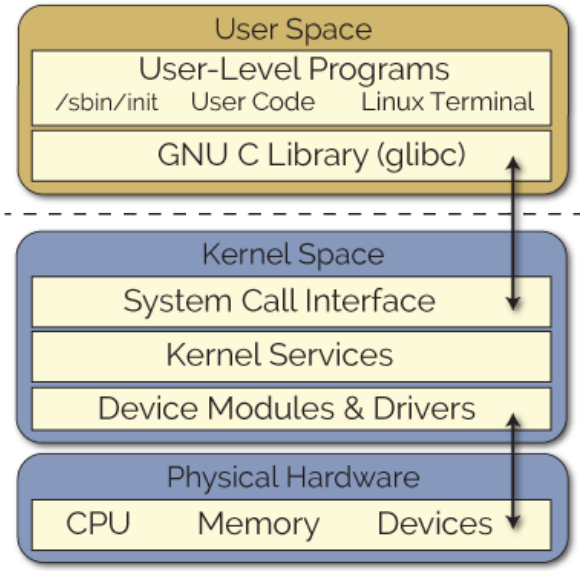
Un LKM es un mecanismo para añadir código al kernel de Linux, o eliminarlo, en tiempo de ejecución. Es una técnica ideal para dar soporte a nuevos dispositivos periféricos y sistemas de archivos, ya que permiten que el kernel se comunique con el hardware sin tener que saber cómo funciona éste. Vienen a ser lo que habitualmente conocemos como software de controlador del dispositivo. La alternativa a los LKMs sería incluir el código para todos los posibles controladores de dispositivos en el núcleo de Linux, lo que daría lugar a un núcleo extremadamente complejo e inmanejable.

Sin esta capacidad modular, el kernel de Linux sería muy grande, ya que tendría que soportar todos los controladores que se pudieran llegar a necesitar. También habría que reconstruir el núcleo cada vez que se quisiera añadir nuevo hardware o actualizar un controlador de dispositivo.

Los LKM se cargan en tiempo de ejecución, pero no se ejecutan en el espacio de usuario, ya que son esencialmente parte del kernel. Ese código se ejecuta en el espacio del kernel, mientras que las aplicaciones se ejecutan en el espacio de usuario.

Tanto el espacio del kernel como el del usuario tienen sus propios espacios de direcciones de memoria que no se pueden solapar.

Este enfoque garantiza que las aplicaciones que se ejecutan en el espacio de usuario tengan una visión coherente del hardware, independientemente de las características físicas de los diferentes periféricos que podamos tener. Así, los servicios del kernel se ponen a disposición del espacio de usuario de forma controlada usando llamadas al sistema. A continuación se muestra el esquema de funcionamiento y relaciones entre espacio del kernel y espacio de usuario:



Es conveniente usar módulos del kernel porque así la interacción con dispositivos periféricos se realiza mediante una interfaz homogénea siempre (usando ***sysfs***) y operaciones de archivo sencillas tipo apertura, lectura, escritura y cierre.

4. Preparar el sistema para construir LKMs

Para desarrollar LKMs el sistema operativo debe estar preparado para compilar el código del kernel, y para ello hay que tener instaladas las cabeceras de Linux:

```
sudo apt-get update
sudo apt-cache search linux-headers-$(uname -r)
sudo apt-get install linux-headers-$(uname -r)
```

Debemos asegurarnos de que instalamos las cabeceras para la versión correcta de nuestro kernel (podemos usar `uname -a` para identificar la versión concreta).

Puesto que es relativamente fácil bloquear (y dañar) el sistema operativo cuando cargamos nuevos LKMs, se recomienda **no** usar nuestra máquina Linux principal, sino usar máquinas virtuales en las que hacer todas las pruebas.

5. El código del módulo

Debemos tener en cuenta que un módulo del kernel no es una aplicación. Como veremos más adelante, no tendremos una función `main()` como en otros programas. En general, los módulos del kernel tienen las siguientes características generales:

- No se ejecutan secuencialmente (un módulo del kernel se registra a sí mismo para recibir y atender las peticiones definidas dentro del código del módulo).
- No tienen recolector de basura automático (debemos reservar y liberar la memoria, cuando sea necesario).
- No se pueden usar funciones `printf()` (el código del núcleo no puede acceder a las librerías de funciones).
- Pueden ser interrumpidos.
- Se ejecutan con nivel de privilegios más alto que las aplicaciones de usuario.
- No tienen soporte para operaciones de punto flotante.

Para este seminario usaremos como código de ejemplo el facilitado por "*derekmolloy*" en su repositorio:

https://github.com/derekmolloy/exploringBB/tree/master/extras/kernel/

El siguiente listado proporciona el código de un primer ejemplo de LKM muy sencillo. El código utiliza la función `printk()` para mostrar "¡Hola mundo!..." en los registros de log del kernel:

```
/**
 * @file    hello.c
 * @author  Derek Molloy
 * @date    4 April 2015
 * @version 0.1
 * @brief  An introductory loadable kernel module (LKM).
 * @see http://www.derekmolloy.ie/ for a full description
*/

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

MODULE_LICENSE("GPL");          ///< The license type
MODULE_AUTHOR("Derek Molloy"); ///< The author
MODULE_DESCRIPTION("A simple Linux driver."); ///< description
MODULE_VERSION("0.1");          ///< The version of the module

static char *name = "world";
module_param(name, charp, S_IRUGO);
MODULE_PARM_DESC(name, "The name to display in log");

/** @brief The LKM initialization function
 *  @return returns 0 if successful
 */
static int __init helloBBB_init(void){
   printk(KERN_INFO "EBB: Hello %s from the BBB LKM!\n", name);
   return 0;
}

/** @brief The LKM cleanup function
*/
static void __exit helloBBB_exit(void){
   printk(KERN_INFO "EBB: Goodbye %s from BBB LKM!\n", name);
}

/** @brief Identify the initialization function at insertion
time and the cleanup function
 */
module_init(helloBBB_init);
module_exit(helloBBB_exit);
```

Algunas aclaraciones previas:

- La declaración `MODULE_LICENSE("GPL")` proporciona información (a través de `modinfo`).
- Debemos evitar siempre el uso de variables globales en los módulos del núcleo. Debe usar la palabra clave `static` para restringir el alcance de una variable dentro del módulo.
- La macro `module_param(name, type, permissions)` tiene tres parámetros: `name` (el nombre del parámetro mostrado al usuario y el nombre de la variable en el módulo), `type` (el tipo del parámetro) y `permissions` (son los permisos de acceso).
- Las funciones incluidas en el LKM pueden tener el nombre que se desee (por ejemplo, `helloBBB_init()` y `helloBBB_exit()`), sin embargo, se deben pasar los mismos nombres a las macros especiales `module_init()` y `module_exit()`
- El uso de `printk()` es muy similar al de la función `printf()`, pero debe especificar un nivel de registro (`KERN_EMERG`, `KERN_ALERT`, `KERN_CRIT`, `KERN_ERR`, `KERN_WARNING`, `KERN_NOTICE`, `KERN_INFO`, `KERN_DEBUG` y `KERN_DEFAULT`).

Cuando se cargue este módulo, se ejecutará la función `helloBBB_init()`, y cuando se descargue el módulo se ejecutará la función `helloBBB_exit()`.

Para compilar el módulo usaremos un Makefile, haciendo el proceso de compilación muy sencillo:

```
obj-m+=hello.o

all:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD)
clean
```

Como resultado, obtendrá un LKM llamado `hello.ko` en el directorio de trabajo (el archivo con extensión **.ko**):

```
[pedro@server part1$ uname -a
 Linux server 5.0.0-38-generic #41-Ubuntu SMP Tue Dec 3 00:27:35 UTC 2019 x86_64 x86_64
[pedro@server part1$
[pedro@server part1$ make
 make -C /lib/modules/5.0.0-38-generic/build/ M=/home/pedro/derekmolloy/part1 modules
 make[1]: se entra en el directorio '/usr/src/linux-headers-5.0.0-38-generic'
   CC [M]  /home/pedro/derekmolloy/part1/hello.o
   Building modules, stage 2.
   MODPOST 1 modules
   CC      /home/pedro/derekmolloy/part1/hello.mod.o
   LD [M]  /home/pedro/derekmolloy/part1/hello.ko
 make[1]: se sale del directorio '/usr/src/linux-headers-5.0.0-38-generic'
[pedro@server part1$
[pedro@server part1$ ls -l
 total 36
 -rw-r--r-- 1 pedro usuario 2424 may 11 11:57 hello.c
 -rw-r--r-- 1 pedro usuario 5224 may 11 11:59 hello.ko
 -rw-r--r-- 1 pedro usuario  646 may 11 11:59 hello.mod.c
 -rw-r--r-- 1 pedro usuario 2728 may 11 11:59 hello.mod.o
 -rw-r--r-- 1 pedro usuario 3416 may 11 11:59 hello.o
 -rw-r--r-- 1 pedro usuario  154 may 11 11:57 Makefile
 -rw-r--r-- 1 pedro usuario   46 may 11 11:59 modules.order
 -rw-r--r-- 1 pedro usuario    0 may 11 11:59 Module.symvers
 -rw-r--r-- 1 pedro usuario  407 may 11 11:57 proceso.txt
 pedro@server part1$
```

5. Probando el módulo LKM

Este módulo puede ser cargado ahora usando las herramientas del módulo del kernel como se muestra en las siguientes capturas. Primero veamos cómo insertar el nuevo módulo en el kernel (`insmod`):

```
[pedro@server part1$
 pedro@server part1$ sudo insmod hello.ko
 pedro@server part1$ lsmod
 Module                  Size  Used by
 hello                  16384  0
 nls_iso8859_1          16384  0
 uas                    24576  0
 usb_storage            69632  1 uas
 nvidia_uvm            831488  0
 snd_hda_codec_hdmi     53248  1
```

A continuación solicitaremos información sobre el módulo (`modinfo`):

```
[pedro@server part1$
 pedro@server part1$ modinfo hello.ko
 filename:       /home/pedro/derekmolloy/part1/hello.ko
 version:        0.1
 description:    A simple Linux driver for the BBB.
 author:         Derek Molloy
 license:        GPL
 srcversion:     0DD9FE0DE42157F9221E608
 depends:
 retpoline:      Y
 name:           hello
 vermagic:       5.0.0-38-generic SMP mod_unload
 parm:           name:The name to display in /var/log/kern.log (charp)
[pedro@server part1$
[pedro@server part1$
[pedro@server part1$ sudo rmmod hello.ko
[pedro@server part1$ _
```

Finalmente, lo eliminamos del kernel (`rmmod`) y revisaremos la salida de la función `printk()` en el registro de log del kernel:

```
pedro@server part1$
pedro@server part1$ sudo su -
root@server:~# cd /var/log
root@server:/var/log# tail -f kern.log

May 11 12:00:35 kernel: [6649143.075501] EBB: Hello world from the BBB LKM!
May 11 12:01:40 kernel: [6649208.426850] EBB: Goodbye world from the BBB LKM!
```

5. Conclusiones

Si hemos seguido correctamente las instrucciones del tutorial habremos desarrollado un módulo del kernel cargable (LKM) muy simple, lo habremos cargado en el kernel y habremos comprobado su funcionamiento.

Teniendo en cuenta la complejidad del tema del desarrollo de módulos/drivers para dispositivos, así como su integración en el kernel de Linux, resulta más adecuado comenzar con un ejemplo lo más sencillo posible. En cualquier caso, nos ha servido para aprender cómo funcionan los módulos cargables del kernel; también para configurar el sistema operativo para desarrollar este tipo de módulos; y finalmente para probar su funcionamiento en una distribución Linux real.

Cuestiones a resolver

El objetivo principal es conocer cómo funciona el sistema de módulos cargables del kernel de Linux y hacer un módulo sencillo.

El estudiante debe estudiar el sistema de LKM de Linux. A continuación desarrollará un módulo sencillo en lenguaje C y lo cargará en el kernel usando las herramientas estudiadas. Comprobará su correcto funcionamiento inspeccionando los logs del sistema y finalmente descargará el módulo.

Como resultado **se mostrará** al profesor el funcionamiento correcto del LKM desarrollado así como el proceso de carga en el kernel.

En el documento a entregar se describirá cómo se ha creado y cargado el módulo, y se incluirán varias capturas de pantalla mostrando el proceso y el resultado.

Normas de entrega

La práctica o seminario podrá realizarse de manera individual o por grupos de hasta 2 personas.

Se entregará como un archivo de texto en el que se muestre la información requerida. También se puede utilizar la sintaxis de Markdown para conseguir una mejor presentación e incluso integrar imágenes o capturas de pantalla. La entrega se realizará subiendo los archivos necesarios al repositorio "**PDIH**" en la cuenta de GitHub del estudiante, a una carpeta llamada "**S-LKM**".

Toda la documentación y material exigidos se entregarán en la fecha indicada por el profesor. No se recogerá ni admitirá la entrega posterior de las prácticas/seminarios ni de parte de los mismos.

La detección de copias implicará el suspenso inmediato de todos los implicados en la copia (tanto de quien realizó el trabajo como de quien lo copió).

Las faltas de ortografía se penalizarán con hasta 1 punto de la nota de la práctica o seminario.

Referencias

https://es.wikipedia.org/wiki/Exchangeable_image_file_format
https://www.thegeekstuff.com/2013/07/write-linux-kernel-module/
https://es.wikipedia.org/wiki/M%C3%B3dulo_de_n%C3%BAcleo
https://es.wikipedia.org/wiki/N%C3%BAcleo_Linux
http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/
https://github.com/derekmolloy/exploringBB/tree/master/extras/kernel/
http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/
http://junyelee.blogspot.com/2018/10/linux-device-drivers.html
https://www.geeksforgeeks.org/linux-kernel-module-programming-hello-world-program/
https://www.thegeekstuff.com/2012/04/linux-lkm-basics/
https://sysplay.github.io/books/LinuxDrivers/book/Content/Part09.html
http://embeddedguruji.blogspot.com/search/label/linux-device-driver
http://embeddedguruji.blogspot.com/2013/06/simple-usb-driver-in-linux.html
https://stackoverflow.com/questions/31052993/why-is-the-probe-function-in-my-kernel-module-not-being-called
https://en.wikipedia.org/wiki/Human_interface_device
https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/introduction-to-hid-concepts
https://embeddedguruji.blogspot.com/2019/04/learning-usb-hid-in-linux-part-1-basics.html

Anexo. Ejemplo de driver USB bajo Linux

Un driver USB bajo Linux puede crearse de diferentes maneras. Una de ellas es usar usando la especificación USB HID (Human Interface Device). La otra alternativa es usar la plantilla anterior, añadiendo varias estructuras de datos y funciones especiales.

Por un lado, cuando el dispositivo se inserta en el bus USB, si el identificador de proveedor y el identificador de producto coinciden con el controlador registrado en el núcleo, se llama a la función de sondeo (*probe*). Para eso, la función recibe la estructura `usb_device`, el número de interfaz y el identificador de interfaz.

Además, el controlador USB se registra en el subsistema de Linux, dándole información sobre los dispositivos que soporta y la función que debe ser llamada cuando el dispositivo se inserta o se retira del sistema. Toda esta información se pasa con la ayuda de la estructura `usb_driver`. Veamos a continuación un código de ejemplo muy sencillo para crear un driver de este tipo:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/usb.h>
#include <linux/usb/input.h>
#include <linux/hid.h>
```

```
#define VENDOR_ID 0x0930
#define DEVICE_ID 0x6545

MODULE_LICENSE("GPL");
MODULE_AUTHOR("dev");
MODULE_DESCRIPTION("eusb driver");

static struct usb_device_id eusb_table[] = {
        { USB_DEVICE(VENDOR_ID, DEVICE_ID) },
        { } /* Terminating entry */
};

MODULE_DEVICE_TABLE (usb, eusb_table);

static int eusb_probe(struct usb_interface *interface, const
struct usb_device_id *id) {
        pr_debug("USB probe function called\n");
        return 0;
}

static void eusb_disconnect(struct usb_interface *interface) {
        pr_debug("USB disconnect function called\n");
}

static struct usb_driver eusb_driver = {
    //.owner =  THIS_MODULE,
    .name =     "eusb",
    .probe =    eusb_probe,
    .disconnect =   eusb_disconnect,
    .id_table =     eusb_table
};

static int __init eusb_init(void) {
    int result = 0;

    pr_debug("Hello world!\n");
    result = usb_register(&eusb_driver);
    if(result){
        pr_debug("error %d while registering usb\n", result);}
    else{pr_debug("no error while registering usb\n");}

    return 0;
}

static void __exit eusb_exit(void) {
    usb_deregister(&eusb_driver);
    pr_debug("Exiting module\n");
}

module_init(eusb_init);
module_exit(eusb_exit);
```

El contenido del fichero `Makefile` para compilar el módulo es:

```
obj-m := eusb.o
CFLAGS_eusb.o := -DDEBUG
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```