Universidad Autónoma de Chihuahua

Facultad de Ingeniería

Sistemas Operativos I

-- Módulo de Kernel --

Luis Julián Lozoya Navarrete

Módulo de Kernel:

Para introducirnos un poco más en el desarrollo de este trabajo hay que comprender en que consiste un módulo de kernel, el cual no es más que nada un fragmento de código que se puede cargar y descargar en el kernel del sistema operativo a voluntad, sin necesidad de reiniciar el sistema. Estos módulos amplían la funcionalidad del kernel y permiten agregar características adicionales de manera dinámica.

Los módulos permiten que el kernel sea más modular. Esto significa que solo se carga en memoria el código necesario, reduciendo el uso de recursos del sistema y permitiendo actualizaciones y cambios sin necesidad de reiniciar.

Como veremos más adelante, la compilación de un módulo generalmente implica el uso del make con un Makefile que define cómo construir el módulo.

- **Explicación del programa.**

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/miscdevice.h>
```

El kernel al ser un tipo de embebido no puede depender de bibliotecas del sistema, por lo que es necesario instalar las cabeceras del kernel, diciéndole que debe de estar instaladas en la ruta default.

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Luis Julian");
MODULE_DESCRIPTION("First Module!");
MODULE_VERSION("0.01");


#define DEVICE_NAME "whisper"
```

Aquí tenemos algunas macros y el #define que es una macro también, que no es más que una directiva del preprocesador del lenguaje C, en otras palabras, es lo que se realiza antes de compilar.

Las macros que tenemos son para declarar la licencia, autor, descripción, versión y el nombre del dispositivo.

```
static int major;
struct cdev char_device;
```

Contamos con un par de variables globales, que nos serán de ayuda en nuestro código.

```
static int whisper_open(struct inode*, struct file*);
static int whisper_release(struct inode*, struct file*);
static ssize_t whisper_read(struct file*, char*, size_t, loff_t*);
static ssize_t whisper_write(struct file*, const char*, size_t, loff_t*);
int write_to_user(char *, char *);
```

Y unas cuantas funciones que con ayuda del file_operations le asignamos a cada elemento apuntadores a funciones que declaramos previamente

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = whisper_open,
    .release = whisper_release,
    .read = whisper_read,
    .write = whisper_write,
};
```

El código cuenta con una función de inicialización, esto es lo primordial para un módulo de kernel. Como queremos que nuestro modulo se relacione con un archivo de dispositivo de tipo carácter (comunicación de byte en byte), le asignamos a la variable global major la función register_chrdev, en otras palabras, le estamos diciendo que nos registre un dispositivo de carácter que se llame whisper y le pasamos la referencia a la estructura file_operations, que es a la que le asignamos que funciones usará cuando lo abra, pudiendo realizar cualquier operación básica de un archivo.

```
static int __init whisper_init(void) {
    major = register_chrdev(0, DEVICE_NAME, &fops);

    if (major < 0) {
        printk(KERN_ALERT "whisper Module load failed\n");
        return major;
    }
```

De igual forma estamos validando el valor de major, el cual no puede ser menor que cero.

El major number es un número que asigna el kernel y es lo que nos aparece cuando listamos el archivo de dispositivo, indicando la relación del módulo con el dispositivo.

```
    printk(KERN_INFO "whisper Module has been loaded\n");

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

Aquí estamos utilizando printk, que lo que hace es escribir hacia la salida estándar del kernel, la cual está redirigida regularmente a la bitácora del sistema. Esto lo hacemos para saber cual es el major number que nos asigno el kernel para poder establecer la relación a mano con ayuda del comando mknod

```
static void __exit whisper_exit(void) {
    printk(KERN_INFO "Aún no hay whisper, removiendo módulo!!\n");
}

static int whisper_open(struct inode *inodep, struct file *filep) {
   printk(KERN_INFO "Super Module device opened\n");
   return 0;
}
```

También contamos con funciones de salida y abrir, que lo único que hacen es mandar un mensaje, que se mostrará en la bitácora, cuando abren y cierran un archivo.

```
static ssize_t whisper_write(struct file *filep, const char *buffer,
                        size_t len, loff_t *offset) {
    //int errors = 0;
    static char message[256] = {0};
    if (copy_from_user(message, (void *)buffer, len))
        return -EFAULT;
    /*if (len > 1024) {
        printk(KERN_INFO "The message must be 1024 bytes or less");
        return -EFAULT;
    }*/
    strcpy(user_message, message);
    printk(KERN_INFO "From userland: %s\n", (char *)message);
    return len;
}
```

Está función es cuando queremos escribir algo en nuestro archivo, que para objetivos del trabajo no utilizaremos a gran medida, pero nos permite agarrar mensajes en espacio de usuario y ponerlos en espacio de kernel.

```c
static int whisper_release(struct inode *inodep, struct file *filep) {
   printk(KERN_INFO "Super Module device closed\n");
   return 0;
}

static ssize_t whisper_read(struct file *filep, char *buffer, size_t len, loff_t *offset) {
    int errors = 0;
    char *message = "Tonight the music seems so loud\nI wish tha we could lose this crowd...\nMaybe is better this way\n";
    int message_len = strlen(message);
    errors = copy_to_user(buffer, message, message_len);
    printk(KERN_INFO "user_message: %s\n", message);
    return errors == 0 ? message_len : -EFAULT;
}


module_init(whisper_init);
module_exit(whisper_exit);
```

Por último, contamos con release que es lo que se ejecuta cuando quitamos el módulo con rmmod, y read que es lo ejecutado cuando le pedimos lectura al módulo con ayuda de un comando como lo puede ser cat.

Para compilar el código ocuparemos la ayuda de un archivo llamado **Makefile**, el cual nos permite construir programas de manera compleja y mantener cierto versionamiento de lo que compilamos, por lo que, si el archivo ya lo hemos compilado anteriormente y lo queremos volver a compilar sin realizarle ningún cambio al código, este ya no compilará, todo esto para usar lo menos de recursos de cómputo posibles. A continuación, se muestra el archivo Makefile utilizado.

```
obj-m += whisper.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


old:
	make -C /lib/modules/6.1.13-200.fc37.x86_64/build M=$(PWD) modules
```

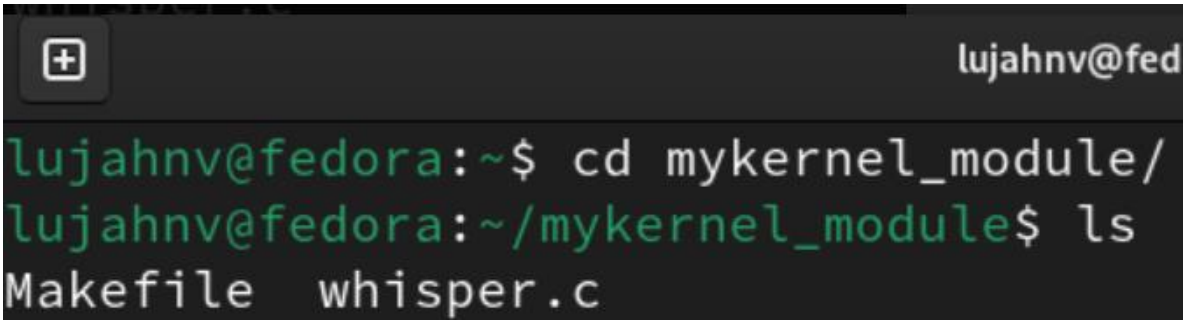
El archivo Makefile cuenta con tres targets (all, clean, old), o etiquetas que asignamos para actividades que va a realizar make. En el caso de all, ejecuta el comando make con el switch -C (le dice al comando make que la raíz de búsqueda para los archivos de encabezado es en la ruta default para los archivos de encabezado).

M = $(PWD) → Hace que se cambie de directorio hacia donde están los archivos que el kernel tiene designados para construir un módulo y considere el Working Directory donde se encuentra el Código del módulo actual… PWD = Print Working Directory.

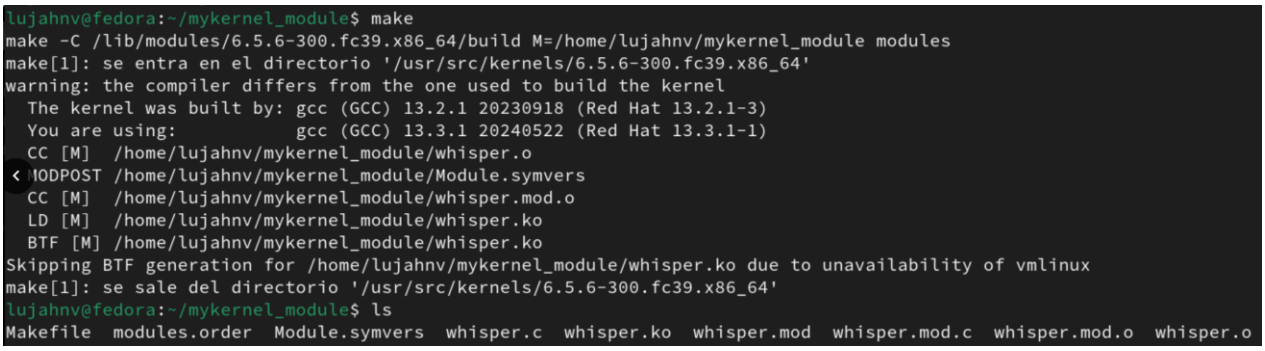
/$(shell uname -r) nos da la versión del kernel que estamos utilizando en ese momento.

- **Compilación y ejecución del código:**

Primero que nada, nos dirigimos a la ruta donde se encuentra nuestro módulo y el archivo Makefile.



Después con ayuda del comando make compilamos nuestro módulo, después de haberlo compilado exitosamente nos deberían aparecer múltiples archivos, entre ellos el archivo whisper.ko



Ahora toca insertar el modulo con ayuda del comando insmod para integrarlo al espacio de kernel.

Una vez insertado, podemos visualizar la bitácora del sistema con el comando journalctl -k y ver en el siguiente mensaje:

```
jun 04 22:31:17 fedora kernel: whisper: module verification failed: signature and/or required key missing - tainting kernel
jun 04 22:31:17 fedora kernel: whisper Module has been loaded
jun 04 22:31:17 fedora kernel: I was assigned major number 239. To talk to
jun 04 22:31:17 fedora kernel: the driver, create a dev file with
jun 04 22:31:17 fedora kernel: 'mknod /dev/whisper c 239 0'.
jun 04 22:31:17 fedora kernel: Try various minor numbers. Try to cat and echo to
jun 04 22:31:17 fedora kernel: the device file.
jun 04 22:31:17 fedora kernel: Remove the device file and module when done.
```

Ejecutamos ese comando para (mknod /dev/whisper c 239 0), que nos permite crear un archivo de dispositivo para poder interactuar con nuestro módulo

```
lujahnv@fedora:~/mykernel_module$ sudo mknod /dev/whisper c 239 0
```

Para ser más específicos, este comando lo que hace es crear un archivo de dispositivo en la ruta /dev/whisper, de tipo carácter con el major number 239

Por último, para que nuestro módulo nos cante el éxito romántico careless whisper ejecutamos el comando cat /dev/whisper y podremos visualizar lo siguiente en la terminal.

```
Tonight the music seems so loud
I wish tha we could lose this crowd...
Maybe is better this way
Tonight the music seems so loud
I wish tha we could lose this crowd...
Maybe is better this way
Tonight the music seems so loud
I wish tha we could lose this crowd...
Maybe is better this way
```

Nota: Para liberar el módulo del sistema se utiliza el comando rmmod.

-- De esta manera concluimos de manera exitosa con la creación de un módulo de kernel --