Trabajo Práctico 0
Device Driver

Docentes:

Chuquimango Chilon Luis Benjamin

Echavarri Alan Pablo

Alumno:

Ruiz Barbas Yair Maximiliano

Modulo Hola Mundo

A continuación vamos a hacer funcionar el módulo hola mundo https://bitbucket.org/sor2/tp0.

Este módulo es un módulo sin lógica, por lo cual la tarea de hacerlo funcionar se reduce a poder agregarlo.

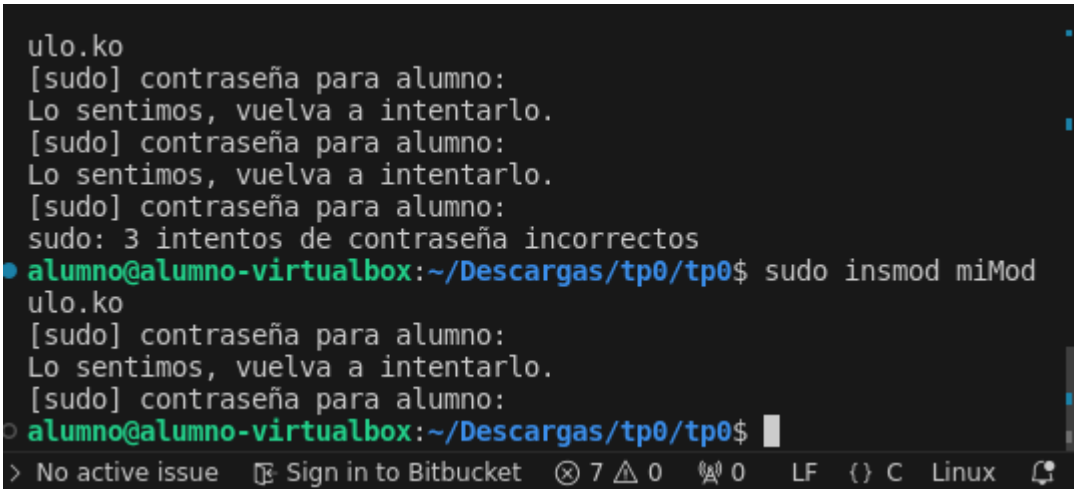
Para lograr esto, lo primero que debemos realizar es cargar el módulo. Esto se logra con el **make**.

```
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$ make
make -C /lib/modules/5.4.0-70-generic/build M=/home/alumno/D
escargas/tp0/tp0 modules
make[1]: se entra en el directorio '/usr/src/linux-headers-5
.4.0-70-generic'
  CC [M]  /home/alumno/Descargas/tp0/tp0/miModulo.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/alumno/Descargas/tp0/tp0/miModulo.mod.o
  LD [M]  /home/alumno/Descargas/tp0/tp0/miModulo.ko
make[1]: se sale del directorio '/usr/src/linux-headers-5.4.
0-70-generic'
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$
```

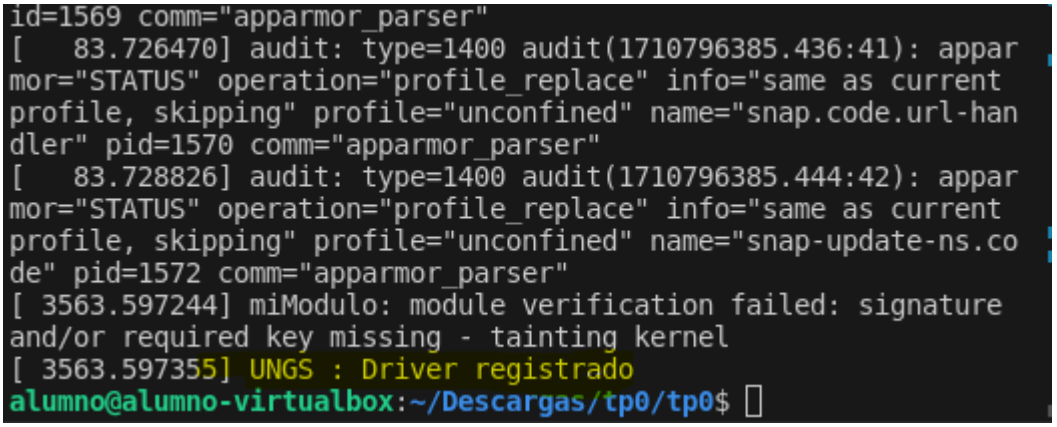
ModInfo de nuestro archivo generado **.ko**:

```
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$ modinfo miModulo.
ko
filename:       /home/alumno/Descargas/tp0/tp0/miModulo.ko
description:    Un primer driver
author:         UNGS
license:        GPL
srcversion:     1CD920ACDE13DD71D14A866
depends:
retpoline:      Y
name:           miModulo
vermagic:       5.4.0-70-generic SMP mod_unload modversions
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$
```

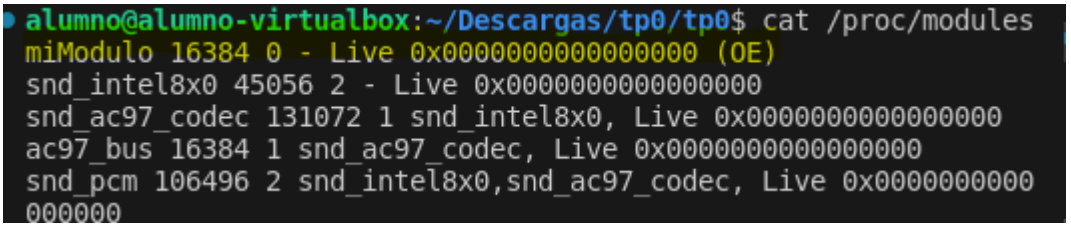
Con el módulo compilado podemos realizar la instalación del mismo utilizando el comando **insmod:**



Para chequear que está instalado podemos realizar lo siguiente. Vamos a ejecutar el comando **dmesg | tail** para ver los mensajes del kernel. Si el módulo se instaló bien entonces deberíamos ver el mensaje *UNGS : Driver registrado*:



Según la bibliografía también tenemos otra forma de chequear la instalación del módulo correctamente, tenemos que realizar un **cat** del archivo **modules: cat /proc/modules:**



Con esto podemos confirmar que **nuestro módulo fue cargado y funciona**.

Módulo Char Device

Para esta sección se nos pide elaborar un kernel module para un char device.

Funciones init module y cleanup module

La función init_module se ejecuta cuando se carga el módulo en el kernel. Esta función debería encargarse de preparar todo lo necesario para nuestro módulo, particularmente el registro de nuestro char device. También va a dejar la información necesaria al cargar el módulo para que podamos hablar con él. Por otro lado, la función *cleanup_module* es invocada antes de **rmmod** y debe encargarse de deshacer todo lo que hicimos durante *init_module*.

Imprimiendo con nuestro Char Device

Para lograr esto debemos implementar la función write, de esta manera podemos recibir la información enviada por el usuario y escribir en el kernel.

```
static ssize_t device_write(struct file *filp, const char *user_buffer,
size_t length, loff_t *offset) {
    ssize_t bytes_written = 0;

    if (*offset >= MAX_BUFFER_SIZE) {
        return -ENOSPC; // not enough space
    }

    // check buffer space so I can avoid overflow
       ssize_t bytes_to_write = min(length, (size_t)(MAX_BUFFER_SIZE -
*offset));

    // copy data to kernel
    if (copy_from_user(buffer + *offset, user_buffer, bytes_to_write)) {
        return -EFAULT; // Error de copia
    }

    // Update offset
    *offset += bytes_to_write;
    // update bytes to write
    bytes_written = bytes_to_write;

    // Print data to the kernel
```

```
    printk(KERN_INFO "YRUIZ: Mensaje recibido: %.*s\n", bytes_written,
buffer);
    return bytes_written;
}
```

Utilizando el comando **make** podemos entonces compilar nuestro device para lograr que imprima en el kernel. A continuación instalamos el módulo con el comando **insmod**. Estos pasos son iguales a la primera parte detallada en el informe aunque es importante constatar que el módulo se instaló correctamente. Al ejecutar el comando **dmesg | tail** veremos lo siguiente:

```
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$ dmesg | tail
[    6.128783] Huh? What family is it: 0x19?!
[    7.489377] snd_intel8x0 0000:00:05.0: white list rate for 1028:0177 is 48000
[ 5945.280160] miModulo: module verification failed: signature and/or required key
 missing - tainting kernel
[ 5945.280406] YRUIZ : Driver registrado
[ 5945.280408] I was assigned major number 240. To talk to
[ 5945.280408] the driver, create a dev file with
[ 5945.280408] 'mknod /dev/YRUIZ c 240 0'.
[ 5945.280408] Try various minor numbers. Try to cat and echo to
[ 5945.280409] the device file.
[ 5945.280409] Remove the device file and module when done.
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$
```

De esta manera tenemos el *major number* que será necesario. Además, la instalación del módulo deja un mensaje con las instrucciones necesarias para crear un dev file y comunicarse con el driver.

Si tenemos todos los permisos necesarios entonces podemos hablar con nuestro device. Ejecutando el comando **dmesg | tail** podemos ver si escribió correctamente:

```
!" > /dev/YRUIZ
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$ echo "Hola, este es char device!" >
/dev/YRUIZ
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$ dmesg | tail
[ 5945.280408] I was assigned major number 240. To talk to
[ 5945.280408] the driver, create a dev file with
[ 5945.280408] 'mknod /dev/YRUIZ c 240 0'.
[ 5945.280408] Try various minor numbers. Try to cat and echo to
[ 5945.280409] the device file.
[ 5945.280409] Remove the device file and module when done.
[ 6700.117721] YRUIZ: Mensaje recibido: Hola, este es char device!

[ 6712.657491] YRUIZ: Mensaje recibido: Hola, este es char device!

alumno@alumno-virtualbox:~/Descargas/tp0/tp0$
```

Devolviendo lo último que fue escrito

A partir de este punto nuestro módulo necesita una nueva función, *device_read*. De esta manera nuestro módulo será capaz de leer y devolver el último mensaje que fue escrito. Dado que la consigna nos llevará a realizar futuras modificaciones el código para esta tarea se encuentra a continuación:

```
static ssize_t device_read(struct file *filp, char *buffer, size_t
```

```
length, loff_t *offset) {
    ssize_t bytes_read = 0;

    if (message_length == 0)
        return 0; //no messages

    //Copy message
        bytes_read  =  simple_read_from_buffer(buffer,  length,  offset,
kernel_buffer, message_length);

    message_length = 0;

    return bytes_read;
}
```

Esta versión de nuestro código puede escribir y leer:

```
[14602.587307] the device file.
[14602.587307] Remove the device file and module when done.
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$ echo "Hola, este es char device comp
leto, escribe y lee!" > /dev/YRUIZ
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$ dmesg | tail
[14602.587306] I was assigned major number 240. To talk to
[14602.587306] the driver, create a dev file with
[14602.587307] 'mknod /dev/YRUIZ c 240 0'.
[14602.587307] Try various minor numbers. Try to cat and echo to
[14602.587307] the device file.
[14602.587307] Remove the device file and module when done.
[14641.936246] YRUIZ: Dispositivo abierto
[14641.936250] YRUIZ: Mensaje recibido: Hola, este es char device completo, escrib
e y lee!

[14641.936251] YRUIZ: Dispositivo cerrado
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$ cat /dev/YRUIZ
Hola, este es char device completo, escribe y lee!
alumno@alumno-virtualbox:~/Descargas/tp0/tp0$
```

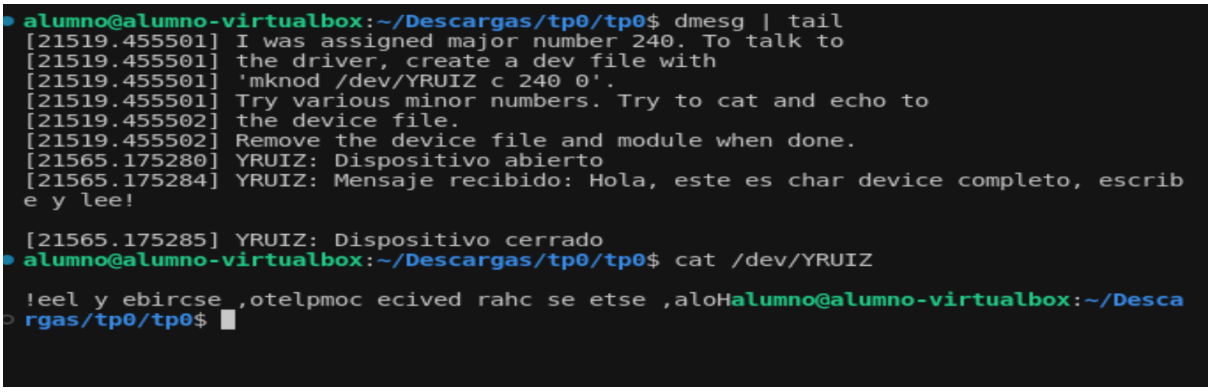
file_operations

La estructura file_operations contiene las operaciones que pueden ser realizadas en el char device, permitiendo que se pueda interactuar con el dispositivo. Para este trabajo se cubrieron varias operaciones de manera que se cumplieran todos los requisitos.

```
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

Devolviendo lo último que fue escrito al revés

Este apartado sólo requiere un ajuste a nuestro código, aunque su naturaleza permanece casi igual a como se ha trabajado a lo largo del documento. Por otro lado, esta implementación requirió otro enfoque ya que estaba utilizando la función simple *read_from_buffer*() pero para lograr escribir al revés la implementación utiliza *put_user()* dentro de un bucle for.



Device open

La función *device_open* se va a ejecutar cuando abramos el device. Para el primer caso, donde solo queremos imprimir en el Kernel, no requerimos de una implementación real ya que no necesitamos realizar operaciones.

static int device_open(struct inode *inode, struct file *file)

```
{
  printk(KERN_INFO "YRUIZ: Dispositivo abierto\n");
  return 0;
}
```

Device release

Similar a lo que sucede con *device_open*, en *device_release* debemos encargarnos de cerrar las operaciones necesarias cuando cerremos el dispositivo. Particularmente para este caso no estamos realizando algo importante entonces las funciones quedan prácticamente vacías.

static int device_release(struct inode *inode, struct file *file)

```
{
  printk(KERN_INFO "YRUIZ: Dispositivo cerrado\n");
  return 0;
}
```

consideraciones y cierre

Lo trabajado a lo largo de este informe representó un desafío no por la complejidad de la tarea sino por la complejidad de las estructuras sobre las que operamos. Es importante destacar que cuando se realizan este tipo de trabajos hay que tener cuidado con los permisos de escritura y la memoria. Por otro lado, la posibilidad de automatizar algunas partes con un *bash* es una gran ventaja.