Taller Drivers

Tomás Melli

Noviembre 2024

Índice

1. Introducción

En este taller, construiremos tres sencillos drivers para el sistema operativo Linux. Es fundamental que tengan presentes los conceptos vistos en las clases teórica y práctica, que les deberían permitir responder a preguntas como:

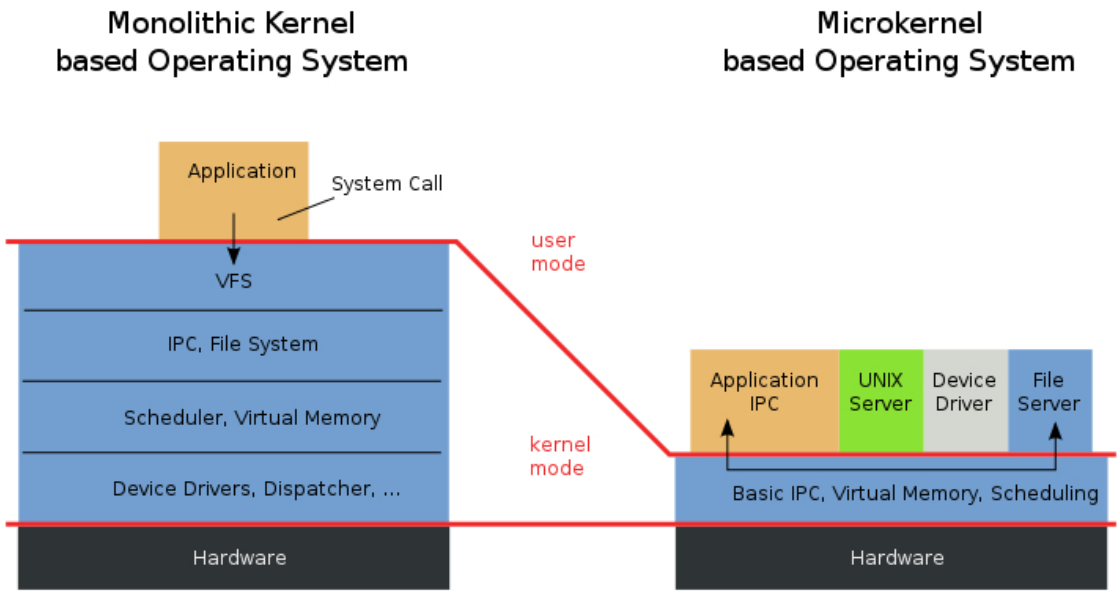
- ¿Qué es un driver? ¿Qué función desempeña?
  Un driver es un componente de software, un programa responsable de permitirle al SO interactuar con un dispositivo de I/O. Este programa tiene la particularidad de que conoce los detalles de la implementación del hardware (precisamente del controlador del dispositvo) con el que interactúa. Esto garantiza cierta capa de abstracción para que el código del usuario del dispositivo no se vuelva loco con detalles específicos del aparato. Es una pieza que se carga como módulo de kernel.

- ¿En qué se diferencia un driver de una pieza de software convencional?
  La diferencia que tiene con una pieza de software convencional es que labura a bajo nivel. Además, como dijimos antes, es específico para el producto, por tanto, el fabricante debe poner a disposición toda la información necesaria para que no haya problemas de compatibilidad.

- En un sistema UNIX, ¿cómo interactúan con un driver los usuarios?
  En un sistema UNIX, se provee cierta API para operar con un dispositivo I/O. Esto incluye un conjunto de funciones como open(int device id), close(int device-id), read(int device-id, int* data), write(int device-id, int* data) las cuales devuelven constantes para saber si se pudo interactuar correctamente. Estas son IO_OK o IO_ERROR.

- ¿Qué tipo de interfaz debe brindar el driver?
  Un driver debe proveer una interfaz simple para que el código del usuario con un par de funciones pueda operar con el dispositivo. Como se dijo anteriormente, generando esa capa de abstracción.

2. Módulos del Kernel



2.1. ¿Los drivers, forman parte del *kernel* ?

Para un sistema tipo UNIX de kernel monolítico sí. Esto se debe a que se ejecutan con máximos privilegio y tienen acceso a las estructuras del kernel y a su código. El tema viene cuando nos ponemos a pensar en la cantidad diferente de dispositivos y fabricantes que existen. Con esto en mente, el kernel no podría traer el soporte necesario para todos ellos por las de que el usuario en algún momento haga uso de uno u otro. Por ello, los drivers se encuentran en forma de MÓDULOS capaces de anexarse al

kernel. Esto es, a la hora de hacer uso de algún dispositivo I/O, se inicializa el driver ("se carga como módulo del kernel") para poder hacer uso de él.
Dicho esto, si queremos practicar programar alguno de estos tendremos que aislarnos de nuestra compu para no romper nada.
Detalles a tener en cuenta cuando laburamos con drivers :

- **Macros definidas en** $< linux/module.h >$
  module_init : esta macro se usa para declarar la función que se ejecutará la momento de carga del driver al kernel. Esta logra iniciar las estructuras que use el driver entre otras cosas.
  module_exit : esta macro se usa para declarar la función que se ejecutará al momento de la descarga del módulo. Esta libera los recursos asociados. Las **macros de metadatos** son aquellas que describen : las funciones del módulo (**MODULE_DESCRIPTION()**) para que otros programadores sepan qué es lo que hace; la licencia (**MODULE_LICENSE("GPL")**) para cumplir con las normas del kernel ; y el nombre del autor (**MODULE_AUTHOR("Tonius")**) .

- **Funciones de** $< linux/kernel.h >$ Vamos a hacer uso de funciones que escriben en el log del kernel. Esto se debe a que en este entorno de trabajo no se tiene acceso a la biblioteca estadar de C. Vamos a utilizar **printk(KERN_INFO .Este es un mensaje informativo");** para definir el nivel de severidad del mensaje como KERN_INFO. También existen otros dos niveles de log : KERN-WARNING y KERN-ERR.
  Tenemos también para pedir memoria la función **kmalloc(size_t size, gfp_t flags)** y para liberarla **kfree(void *ptr)**
  Aquellas vinculadas a la gestión de interrupciones, también están definidas aquí. Para solicitar las líneas de manejo de interrupciones para un handler tenemos **request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev)** y para liberarlas **free-irq(unsigned int irq, void *dev)**.

- **MakeFile distinto del usual**
  El Makefile para un módulo de kernel le indica al sistema de construcción del kernel cómo compilar el módulo, usando el entorno y los archivos de cabecera correspondientes a la versión específica del kernel. Esta dependencia es esencial para garantizar que el módulo funcione correctamente con el kernel en ejecución y para evitar errores de compatibilidad.

- **Comandos necesarios**
  Para poder visualizar qué comandos están cargados en el kernel en cierto momento, su tamaño, entre otras cosas, se utiliza **lsmod**.
  Para poder cargar un módulo se utiliza **insmod** $< nombredelmdulo >$ Para descargarlo, **rmmod** $< nombredelmdulo >$

3. Preparación del driver

Como queremos que el driver sea útil, necesitamos que el usuario pueda *acceder* a él. En el caso de LINUX, esto se logra a través del File System. Ya que cada dispositivo es representado por un archivo. Este archivo está almacenado en el directorio /**dev**. Allí encontraremos un sinfín de archivos. Para distinguirlos, dentro de /dev con *ls -l* vamos a ver algo como :

```
$ ls -l /dev
lrwxrwxrwx  1 root root           3 2010-10-08 20:00 cdrom -> sr0
...
crw-rw-rw-  1 root root      1,   8 2010-10-08 20:00 random
...
brw-rw----  1 root disk      8,   0 2010-10-08 20:00 sda
brw-rw----  1 root disk      8,   1 2010-10-08 20:00 sda1
...
```

El primer caracter indica qué tipo de archivo es. *l : enlace simbólico*, *c : char device (es decir que son dispositivos que se comunican con un flujo constante de datos. Para acceder a ellos, lo hacemos byte a byte, de forma secuencial)* o *b : block device (es decir que son dispositivos que se comunican mediante bloques de datos. Para acceder lo hacemos de a bloque y puede ser aleatorio).*
Un detalle adicional son los **números asociados luego de los permisos de grupo**, el primero se llama **major** y es un número que le asigna el SO para saber qué controlador lo maneja. El segundo se llama **minor** y se utiliza para identificar un dispositivo específico, es decir, una instancia particular controlada por el *major*.

3.1. Pasos para figurar en el sistema

- **Inicializarse como dispositivo**
  Para inicializar un device, varía si se trata de un *char device* o un *block device*. Para nuestro contexto de taller, vamos a trabajar con *char device* y por ello, hacemos el #**include** $< linux/cdev.h >$ para poder llamar a la función **void cdev_init(struct cdev* cdev, struct file_operation* fops);** que sucederá en el momento en que se inicialice el módulo.
  Desglosemos un poco esto. El primer parámetro : **struct cdev* cdev** es la estructura ue representará al dispositivo. El segundo parámetro : **struct file_operation* fops** es una estructura definida en $< linux/fs.h >$ que nos va a permitir definir la *las operaciones que realiza el device* en la sección 4 vamos a hablar más de esta estructura.
  ** Ambas estructuras deberán ser definidas como *static*, esto garantiza seguridad ya que al restringir la visibilidad se reduce la posibilidad de que otras partes del código las modifiquen accidentalmente. Entre otras buenas prácticas.

- **Major y Minor**
  Estamos en condiciones de pedirle al Kernel que nos asigne un *major* y un *minor* de manera dinámica. Para ello, usamos la función
  int alloc_chrdev_region(dev_t* num, unsigned int firstminor, unsigned int count, char* name);. Sus parámetros son : *dev_t* num* que básicamente es el *major*; *firstminor* y *count* pueden setearse en 0 y 1 respectivamente ya que asignaremos cualquier número disponible menor a 0 y luego la cantidad de devices será sólo 1 y finalmente, el *name* del device.
  Posteriormente, tendremos que asignar números al dispositivo que acabamos de inicializar. Para ello, tenemos la función **int cdev_add(struct cdev* cdev, dev_t num, unsigned int count)**.
  Al momento de la descarga del módulo, tendremos que liberar los números, para ello llamamos a **void unregister_chrdev_region(dev_t num, unsigned int count)** y a **void cdev_del(struct cdev* dev)**.

- **Creación de los nodos en el file system**
  Desde el espacio de usuario, por consola podemos escribir el comando : **mknod** $< nodo >$ **c** $< major > < minor >$. De todos modos, por cuestiones de prolijidad se prefiere que *el módulo mismo se encargue* y para ello, necesitamos incluir #**include** $< linux/device.h >$ que contiene clases de dispositivos, operaciones para manejar dispositivos, etc. El procedimiento es el siguiente:

  - Declarar la variable estática **static struct class* mi_clase**
  - Durante la inicialización del módulo, ejecutamos
    mi_clase = class_create(THIS_MODULE, DEVICE_NAME) que básicamente lo que hace es *crear una clase de dispositivo* en base al módulo que la creó. Por ello, se usa la macro *THIS_MODULE* y luego se le pasa el *DEVICE_NAME*. Esta función devuelve un puntero que no será útil en el próximo paso.
  - Ahora tenemos que crear el device, si no hubo error en la creación de la clase. Para ello, llamamos a la función **device_create(mi_clase, NULL, num, NULL, DEVICE_NAME)** que crea el dispositivo per se. Sus parámetros son : *mi_clase* que es el puntero a la clase de dispositivo; el segundo parámetro está en *NULL* porque este es para pasarle un puntero al dispositivo padre; para el tercer parámetro tenemos la variable *num* que representa el número de dispositivo (el major); el cuarto representa un puntero a datos adicionales que

se pueden asociar con el dispositivo; y finalmente el *DEVICE_ NAME* que será el nombre que tendrá el device en el File System.

- Cuando tengamos que descargar el módulo, como tenemos que liberar sus recursos asociados, solicitaremos la destrucción de los nodos llamando a **device_destroy(mi_clase, num)** para desahacerse del device primero, y luego de la clase con **class_destroy(mi_clase)**.

4. Operaciones de archivos

El módulo debe implementar una cantidad de operaciones para que el usuario pueda interactuar con el dispositivo. Estas operaciones deberán ser definidas dentro del **struct file_operations**. Ya vimos que esta estructura es inicializada en el cdev-init. Ahora es momento de completar la información contenida en ella. Este struct contiene *punteros a funciones ue gestionan las operaciones que se pueden realizar en el device*. La pinta que tiene es :

```
struct file_operations {
    struct module *owner;
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*open) (struct inode *, struct file *);
    ssize_t (*release) (struct inode *, struct file *);
    ...
}
```

Y la forma correcta de implementar ciertas operaciones, sigue la siguiente estructura de ejemplo :

```
static struct file_operations mis_operaciones = {
    .owner = THIS_MODULE,
    .read = mi_operacion_lectura,
};

ssize_t mi_operacion_lectura(struct file *filp, char __user *data, size_t s,
    loff_t *off) {
    return 0;
}
```

En el ejemplo provisto, la función de lectura es implementada por mi-operación-lectura que toma como parámetros : primero *el puntero a la instancia actual del archivo abierto* (esto cambia según las llamadas a open()); después en segundo lugar, se recibe el *puntero al buffer en la memoria del usuario* en donde debemos escribir lo leído, es decir que tendremos que copiar desde el kernel hacia el usuario; en tercer lugar, tenemos *el tamaño de bytes que se desea leer*; y finalmente, se pide un *puntero al offset actual de lectura* que para los casos de *char devices* esto no aplica ya que la lectura es secuencial y no aleatoria.

5. Ejercicios

5.1. Ejercicio 1

Para el ejercicio 1, se nos pide completar la estructura del hello_exit(void) y del hello_exit(void) de el driver llamado *nulo*

```
static struct cdev nulo_device;
static dev_t major;
static struct class *nulo_class;
// nombre
#define DEVICE_NAME "nulo_device"
```

```
static int __init hello_init(void) {

  /* Completar */
  printk(KERN_INFO "Se␣está␣cargando␣el␣driver...\n");

  // inicializamos el device
  cdev_init(&nulo_device, &nulo_operaciones);

  // major y minor
  alloc_chrdev_region(&major, 0, 1, DEVICE_NAME);

  // asignamos los números al device
  cdev_add(&nulo_device, major, 1);

  // creamos los nodos en el fs
  nulo_class = class_create(THIS_MODULE, DEVICE_NAME);
  device_create(nulo_class, NULL, major, NULL, DEVICE_NAME);

  // mensaje diciendo que ya todo está seteado correctamente
  printk(KERN_INFO "Todo␣listo␣campeón␣\n");
  return 0;
}

static void __exit hello_exit(void) {
  /* Completar */
  printk(KERN_INFO "Se␣está␣descargando␣el␣driver...\n");

  // liberamos los números asociados al device
  unregister_chrdev_region(major, 1);
  cdev_del(&nulo_device);

  // destruimos los nodos asociados al device
  device_destroy(nulo_class, major);
  class_destroy(nulo_class);
}
```

5.2. Test

Para poder testear nuestros drivers hay que seguir ciertos pasos. Esto se debe a que no es un programa común y corriente. Necesitamos tener instalado en la VM :

- make
- module-init-tools
- linux-headers-< *version* >

Tenemos que crear un **Makefile** con :

```
obj-m := hello.o
KVERSION := $(shell uname -r)

all:
    make -C / lib / modules / $ ( KVERSION )/ build SUBDIRS =
    $ (shell pwd ) modules

clean:
```

```
make -C / lib / modules / $ ( KVERSION )/ build SUBDIRS =
$ ( shell pwd ) clean
```

Una vez compilado llega la parte de la **Ejecución**. Si hacemos un **ls** deberíamos ver algo como :
hello.c hello.ko hello.mod.c hello.mod.o hello.o Makefile modules.order Modules.symvers
** Recordar que nosotros **NO EJECUTAMOS LOS MÓDULOS**. Se ejecutan en los siguientes momentos :

- al ser cargados en el sistema.
- cuando se hace una llamada al sistema.
- cuando se atiende una interrupción.
- al descargarlo.

Dicho esto, podemos hacer la **carga al sistema** con el comando **insmod** que cargará el código y los datos del módulo al kernel.
Existe otro comando, **modprobe** que es una alternativa que tiene en cuenta dependencias entre módulos.
Vamos a lo interesante :

5.2.1. Test Ejercicio 1 : para afianzarse con el test

root@so-taller:/home/taller3/taller3/alumnos/nulo# **make**
make -C /lib/modules/5.4.0-200-generic/build M=/home/taller3/taller3/alumnos/nulo modules make[1]: Entering directory '/usr/src/linux-headers-5.4.0-200-generic'
CC [M] /home/taller3/taller3/alumnos/nulo/nulo.o
Building modules, stage 2.
MODPOST 1 modules
CC [M] /home/taller3/taller3/alumnos/nulo/nulo.mod.o
LD [M] /home/taller3/taller3/alumnos/nulo/nulo.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-200-generic'
root@so-taller:/home/taller3/taller3/alumnos/nulo# **ls**
Makefile Module.symvers modules.order nulo.c nulo.ko nulo.mod nulo.mod.c nulo.mod.o nulo.o
root@so-taller:/home/taller3/taller3/alumnos/nulo# **sudo insmod nulo.ko**
root@so-taller:/home/taller3/taller3/alumnos/nulo# **lsmod | grep nulo**
nulo 16384 0
root@so-taller:/home/taller3/taller3/alumnos/nulo# **ls /dev/ | grep nulo**
nulo

Para ver si funciona realmente, tenemos un script de pyhton para testearlo. Nos movemos a la carpeta de los tests y ...
root@so-taller:/home/taller3/taller3/tests# **python3 nulo_test.py**
== Verificando lectura /dev/nulo ==
* Abriendo dispositivo
* Intentando lectura
* Cerrando lectura
.== Verificando escritura en /dev/nulo ==
* Abriendo dispositivo
* Intentando escribir
* Intentando escribir de nuevo
* Intentando lectura
* Cerrando lectura
. ———————————————————————-
Ran 2 tests in 0.000s
OK

5.3. Ejercicio 2

En el ejercicio 2, ya tenemos toda la lógica de inicialización de la clase y el device. Ahora se nos pide :

- Cuando el usuario invoque a **write**, se interprete la data del buffer como un string al que hay que agregarle *el caracter de fin de string* y luego deberá representar un entero. En caso de error, se devuelve *-EPERM*.
- Cuando el usuario invoque a **read**, se debe devolver un número random entre 0 y el valor que usuario haya escrito previamente (en la función de escritura) y un caracter de fin de línea. Si no escribrió nada antes, entonces falla con *-EPERM*.

** Funciones recomendadas :

- **kmalloc** es una función para pedir memoria dinámicamente en el espacio del kernel. Toma como parámetros : **size_t size** que es el tamaño que necesitamos y **int flags** que son constantes para indicar cómo se debe realizar la asignación de memoria : **GFP_KERNEL** que puede ser bloqueante, es decir que puede esperar a que se libere algo de memoria ; **GFP_ATOMIC** para asignaciones no-bloqueantes que pueden suceder en contexto de interrupciones. Esta función está presente en $< linux/slab.h >$ al igual que **kfree(void* puntero)**.
- **kstrtoint** de la librería $< linux/kernel.h >$. Como su nombre indica, es una función del kernel y nos permite convertir un string a un entero. La función toma tres parámetros : **const char* s** que es la cadena a convertir; **unsigned int base** que es la base numérica; **int* res** que es el puntero a donde guardaremos el resultado. La función devuelve *0 : si fue todo bien* o *-EINVAL : si hubo error en la conversión*
- **get_random_bytes** de la librería $< linux/random.h >$. También función del kernel, capaz de generar aleatoriamente y guardarlos en un buffer. La función toma dos parámetros : **void* buffer** que es el puntero al buffer donde se guarda el resultado; **int nbytes** el número de bytes aleatorios a generar. Esta función no retorna nada.
- **snprintf** de la librería $< linux/kernel.h >$. Esta función permite formatear un string y almacenarlo en un buffer de manera segura ya que limita la cantidad de caracteres que se escriben, evitando desbordamiento. Los parámetros que toma son : **char* str** el puntero al buffer donde se resguardará la data; **size_t tamaño** el tamaño del buffer; **const char* formato**. Esta función añade el caracter de terminación al final. Esta función retorna *un entero positivo sin contar el terminador, en caso de éxito (denotando la cantidad de caracteres escritos en el buffer* en el caso de *fallar, la función puede retornar un número negativo cuando falla el formato; o puede devolver un numero mayor o igual al tamaño del buffer indicando que no se pudo almacenar toda la data porque el espacio fue insuficiente.*

1. **Paso 1 : write**:

```
static ssize_t azar_write(struct file *filp, const char __user *data,
size_t size, loff_t *offset) {
        // variable
        unsigned int str_to_int;

        // pedimos memo
        char* buffer = kmalloc(size+1, GFP_KERNEL);

        // copiamos del espacio de memoria del usuario
        copy_from_user(buffer, data, size);

        // agregamos el caracter nulo
        buffer[size] = 0;

        // usamos kstrtoint para convertir todo esto a entero
```

```
        str_to_int  = kstrtoint(buffer, 10, &upper_bound);

        // chequeamos si fue bien la conversión
        if(str_to_int == 0 && upper_bound > -1)
        {
                kfree(buffer);
                return size;
        }
        // liberamos memo
        kfree(buffer);
        return -EPERM;
    }
```

2. **Paso 2 : read** :

```
static ssize_t azar_read(struct file *filp, char __user *data,
size_t size, loff_t *offset) {
        // variables (van todas arriba porque sino tira un error de ISO 90 )
        unsigned char* res;
        unsigned int nro_random;
        unsigned int resultado;
        size_t formateo, real;

        // si no se escribió nada :: ret
        if (upper_bound > 0)
        {
                // hay que devolver un string
                res = kmalloc(size, GFP_KERNEL);

                // generamos en nro_ranfom
                get_random_bytes(&nro_random, sizeof(nro_random));

                // resultado > 0 y menor al guardado
                resultado =  nro_random % upper_bound;

                // tenemos que guardan en res
                formateo = (size_t) snprintf(res, size, "%d\n", resultado);

                // tam escritura real
                real = min(size, formateo);

                // copiamos al usuario
                copy_to_user(data, res, real);

                // liberamos
                kfree(res);

                // ret
                return size;
        }
        return -EPERM;
}
```

5.4. Ejercicio 3

En el tercer y último ejercicio del taller se debe implementar la sincronización en el contexto del kernel. La idea es conocer un poco las herramientas de sincronización a la hora de enfrentar potenciales

condiciones de carrera en este entorno de desarrollo. Contamos con:

- **Semáforos** de la forma **struct semaphore** con funciones como : *sema_ init(struct semaphore* sem, int value)* para la inicialización del mismo; tenemos *down(struct semaphore* sem)* que es para decrementar el valor y su opuesto *up(struct semaphore* sem)* para aumentar en una unidad su valor; el *down_ interruptible(struct semaphore* sem)* que permite al proceso ser interrumpido por señales.
- **Spinlocks** básicamente es un busy-waiting. Las funciones que lo implementan son : *spin_ lock_ init(spinlock_ t* lock)*, *spin_ lock(spinlock_ t*lock)* y *spin_ unlock(spinlock_ t* lock)*

Para este ejercicio tenemos que hacer uso de la estructura **file**. La particularidad que tiene es que posee un atributo privado a lo OOP, en donde podemos guardar la data que nos pinte. Esto vale porque ese atributo privado, llamado **private_data** es un puntero a void. Cómo hacemos para usarlo ? Acá ponemos el ejemplo del taller :

```
typedef struct foo {
  char bar;
} foo;

static int your_driver_open(struct inode *inod, struct file *filp) {
  filp->private_data = kmalloc(sizeof(struct foo), GFP_KERNEL); // Reservamos
  memoria para la estructura
  ((foo *) filp->private_data)->bar = 4; // Como es puntero a void, necesitamos
   castearlo para poder usarlo
  return -EPERM;
}

static ssize_t your_driver_read(struct file *filp, char __user *data, size_t size
  , loff_t *offset) {
  foo *data = (foo *) filp->private_data; // Como es puntero a void,
  necesitamos castearlo para poder usarlo
  return data->bar
}
```

1. **Paso 1 : open**

```
static int en_uso = 0;

static int letras_open(struct inode *inod, struct file *filp) {

        // pedimos el "mutex" para SLOT_COUNT
        spin_lock(&lock);

        // chequeamos si hay espacio dispo
        if (en_uso >= SLOT_COUNT)
        {
                spin_unlock(&lock);
                return -EPERM;
        }

        // si hay ==> la cantidad de slots en uso ++
        en_uso++;

        // esto es lo del file y el atributo private_data : asignamos espacio
        filp->private_data = kmalloc(sizeof(struct user_data), GFP_KERNEL);
```

```
        // le marcamos como inválida la data
        ((user_data *) filp->private_data)->valid = false;

        // liberamos el "mutex"
        spin_unlock(&lock);
        return 0;

}
```

2. **Paso 2 : close**

```
static int letras_release(struct inode *inod, struct file *filp) {
        // si el usuario no hizo el open() ==> fallamos ... cómo ?
        viendo si se le asignó memo a filp->private_data
        user_data *udata = (user_data *) filp->private_data;
        if (!udata) return -EPERM;

        // si pasó por el open ( donde le seteamos false ) ==>
        spin_lock(&lock);

        // restamos la #slots en uso porque este proceso se va
        en_uso--;

        // liberamos
        spin_unlock(&lock);

        // hacemos el free de lo que pedimos en el open
        kfree(filp->private_data);

        return 0;
}
```

3. **Paso 3 : write**

```
static ssize_t letras_write(struct file *filp, const char __user *data,
size_t size, loff_t *offset) {
        // variables (recordar que siempre van primero por ISO 90 )
        user_data *udata = (user_data *) filp->private_data;

        // letra que traemos del espacio de memo del usuario
        unsigned char letra;

        // si nunca escribió es porque la data está invalidada
        if ( !udata->valid )
        {
                // copiamos una letra del usuario
                copy_from_user(&letra, data, 1);

                // la asignamos
                udata->input = letra;

                // confirmamos que la info es válida
                udata->valid = true;
        }

        return size;
}
```

4. **Paso 4 : read**

```
static ssize_t letras_read(struct file *filp, char __user *data,
size_t size, loff_t *offset) {
        // variables
        user_data *udata = (user_data *) filp->private_data;

        // buffer para guardar las copias del char
        unsigned char* buffer;

        // it
        unsigned int i = 0;

        // chequeamos si la data está válida
        if ( !udata->valid ) return -EPERM;

        // pedimos memo para el buffer
        buffer = kmalloc(size, GFP_KERNEL);

        // copiamos size-veces el char en ese buffer
        // error: for loop initial declarations are only allowed in C99 or
        C11 mode ( no vale usar for )
        while ( i < size )
        {
                buffer[i] = udata->input;
                i++;
        }
        // le copiamos al espacio del usuario el buffer
        copy_to_user(data, buffer, size);

        // liberamos espacio
        kfree(buffer);

        return size;
}
```

6. Conclusión

Fin ! Pasan todos los tests.