Sistemas Operativos y Redes 2

Taller: Linux Device Drivers

Docentes:
Agustín Alexander- aalexander@campus.ungs.edu.ar

Miércoles 25 de Marzo de 2019

Cómo cargamos un device driver en linux

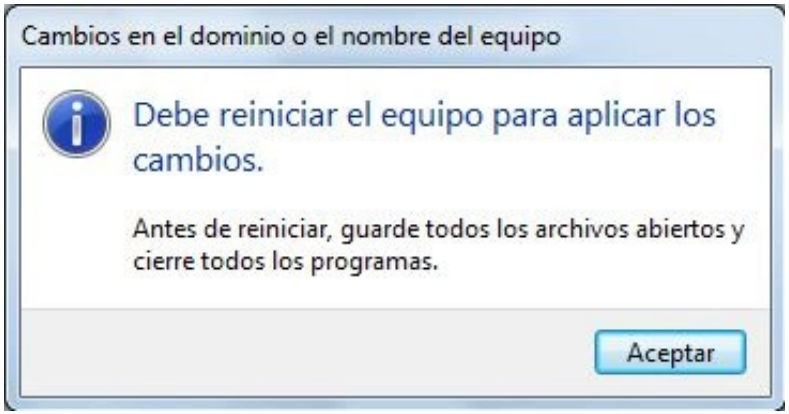


Figure: ¿Y si no quiero?..

1. No es necesario reiniciar para instalar tanto como para desinstalar

Cómo cargamos un device driver en linux

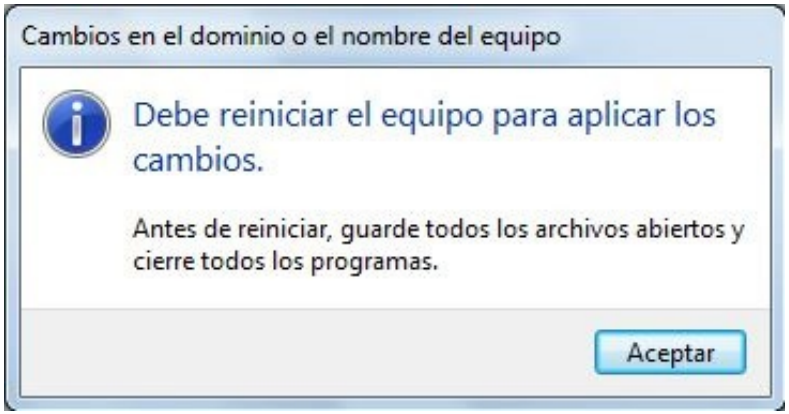


Figure: ¿Y si no quiero?..

1. No es necesario reiniciar para instalar tanto como para desinstalar
2. Se utilizan kernel modules, archivos en C de tipo .ko

Cómo cargamos un device driver en linux

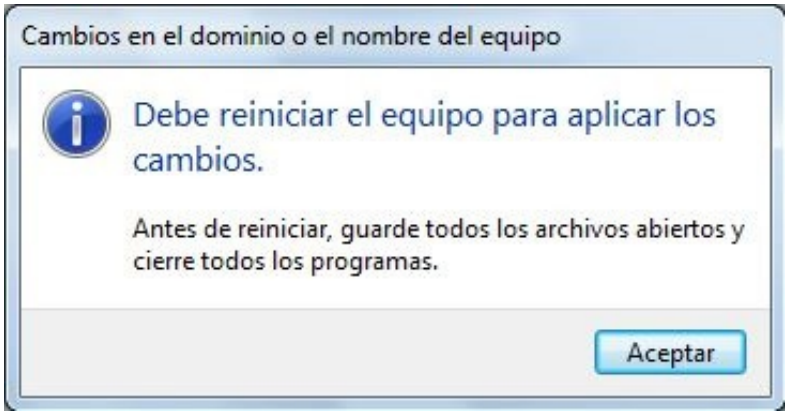


Figure: ¿Y si no quiero?..

1. No es necesario reiniciar para instalar tanto como para desinstalar
2. Se utilizan kernel modules, archivos en C de tipo .ko
3. Se encuentran en **/lib/modules/$KernelVersion**

Cómo cargamos un device driver en linux

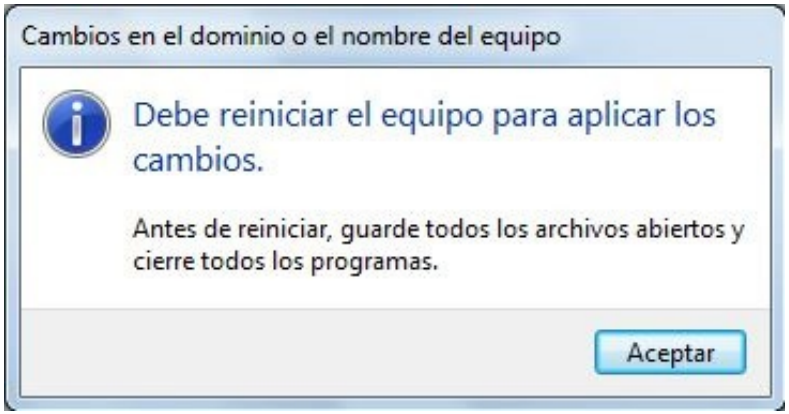


Figure: ¿Y si no quiero?..

1. No es necesario reiniciar para instalar tanto como para desinstalar
2. Se utilizan kernel modules, archivos en C de tipo .ko
3. Se encuentran en **/lib/modules/$KernelVersion**
4. Permiten agregarle funcionalidad al kernel en runtime

Cómo cargamos un device driver en linux

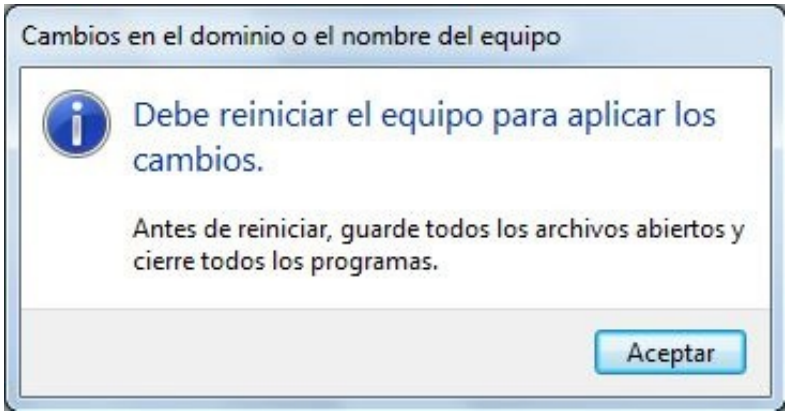


Figure: ¿Y si no quiero?..

1. No es necesario reiniciar para instalar tanto como para desinstalar
2. Se utilizan kernel modules, archivos en C de tipo .ko
3. Se encuentran en **/lib/modules/$KernelVersion**
4. Permiten agregarle funcionalidad al kernel en runtime
5. Pero... ¿que pasa si hay un bug en un modulo del kernel?

Tolerancia a fallas

1. Errores en programas son menos graves que errores en el kernel

Tolerancia a fallas

1. Errores en programas son menos graves que errores en el kernel
2. Un error en el kernel puede resultar en un **oops** o **panic**

Tolerancia a fallas

1. Errores en programas son menos graves que errores en el kernel
2. Un error en el kernel puede resultar en un **oops** o **panic**
3. Un Kernel Panic guarda en un log y reinicia el sistema.

Tolerancia a fallas

1. Errores en programas son menos graves que errores en el kernel
2. Un error en el kernel puede resultar en un **oops** o **panic**
3. Un Kernel Panic guarda en un log y reinicia el sistema.
4. En cambio un **oops** solamente descarga el modulo y logea

BSOD

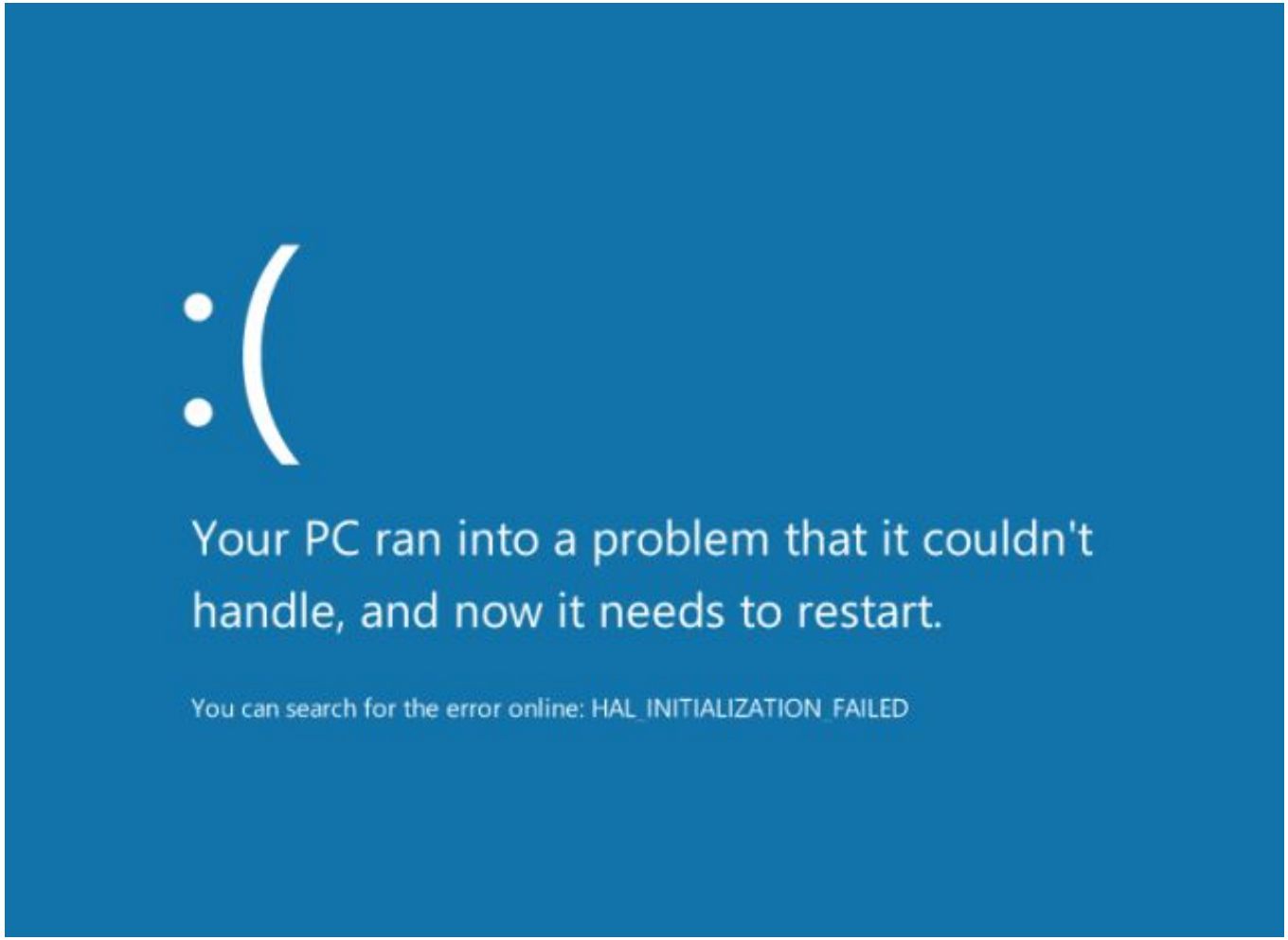


Figure: BSOD o pantalla azul de la muerte

No todos los drivers son creados iguales

1. No todos los modulos del kernel son drivers

No todos los drivers son creados iguales

1. No todos los modulos del kernel son drivers
2. Por ejemplo en linux los schedulers pueden ser modulos

No todos los drivers son creados iguales

1. No todos los modulos del kernel son drivers
2. Por ejemplo en linux los schedulers pueden ser modulos
3. Tambien no todos los drivers se cargan en forma de modulos

No todos los drivers son creados iguales

1. No todos los modulos del kernel son drivers
2. Por ejemplo en linux los schedulers pueden ser modulos
3. Tambien no todos los drivers se cargan en forma de modulos
4. Existen drivers compilados dentro del kernel x ej. **FreeSync**

No todos los drivers son creados iguales

1. No todos los modulos del kernel son drivers
2. Por ejemplo en linux los schedulers pueden ser modulos
3. Tambien no todos los drivers se cargan en forma de modulos
4. Existen drivers compilados dentro del kernel x ej. **FreeSync**
5. Para cargar y descargar este tipo de drivers **si** hace falta recompilar el kernel y reiniciar

No todos los drivers son creados iguales

1. No todos los modulos del kernel son drivers
2. Por ejemplo en linux los schedulers pueden ser modulos
3. Tambien no todos los drivers se cargan en forma de modulos
4. Existen drivers compilados dentro del kernel x ej. **FreeSync**
5. Para cargar y descargar este tipo de drivers **si** hace falta recompilar el kernel y reiniciar
6. Hoy veremos como crear y cargar drivers del tipo Kernel Modules

Codigo de un kernel module

1. Viven en kernel space que tiene su propio espacio de memoria separado del user space

Codigo de un kernel module

1. Viven en kernel space que tiene su propio espacio de memoria separado del user space
2. Los recursos asociados no se limpian cuando se descarga.

Codigo de un kernel module

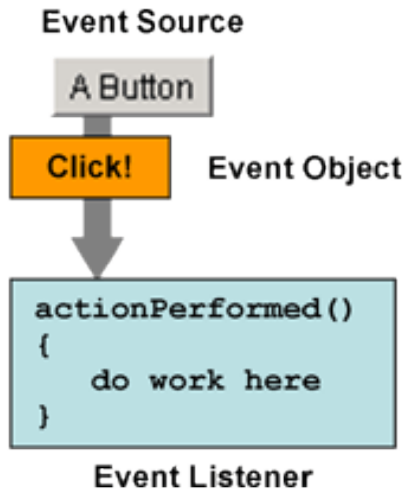
1. Viven en kernel space que tiene su propio espacio de memoria separado del user space
2. Los recursos asociados no se limpian cuando se descarga.
3. Pueden ser interrumpidos y accedidos por mas de un proceso.

Codigo de un kernel module

1. Viven en kernel space que tiene su propio espacio de memoria separado del user space
2. Los recursos asociados no se limpian cuando se descarga.
3. Pueden ser interrumpidos y accedidos por mas de un proceso.
4. Tienen un nivel mayor de privilegio de ejecucion. Roban mas ciclos de CPU

Codigo de un kernel module

1. Viven en kernel space que tiene su propio espacio de memoria separado del user space
2. Los recursos asociados no se limpian cuando se descarga.
3. Pueden ser interrumpidos y accedidos por mas de un proceso.
4. Tienen un nivel mayor de privilegio de ejecucion. Roban mas ciclos de CPU
5. No se ejecuta secuencialmente, similar a event-driven programming.



6

Kernel Space & User Space

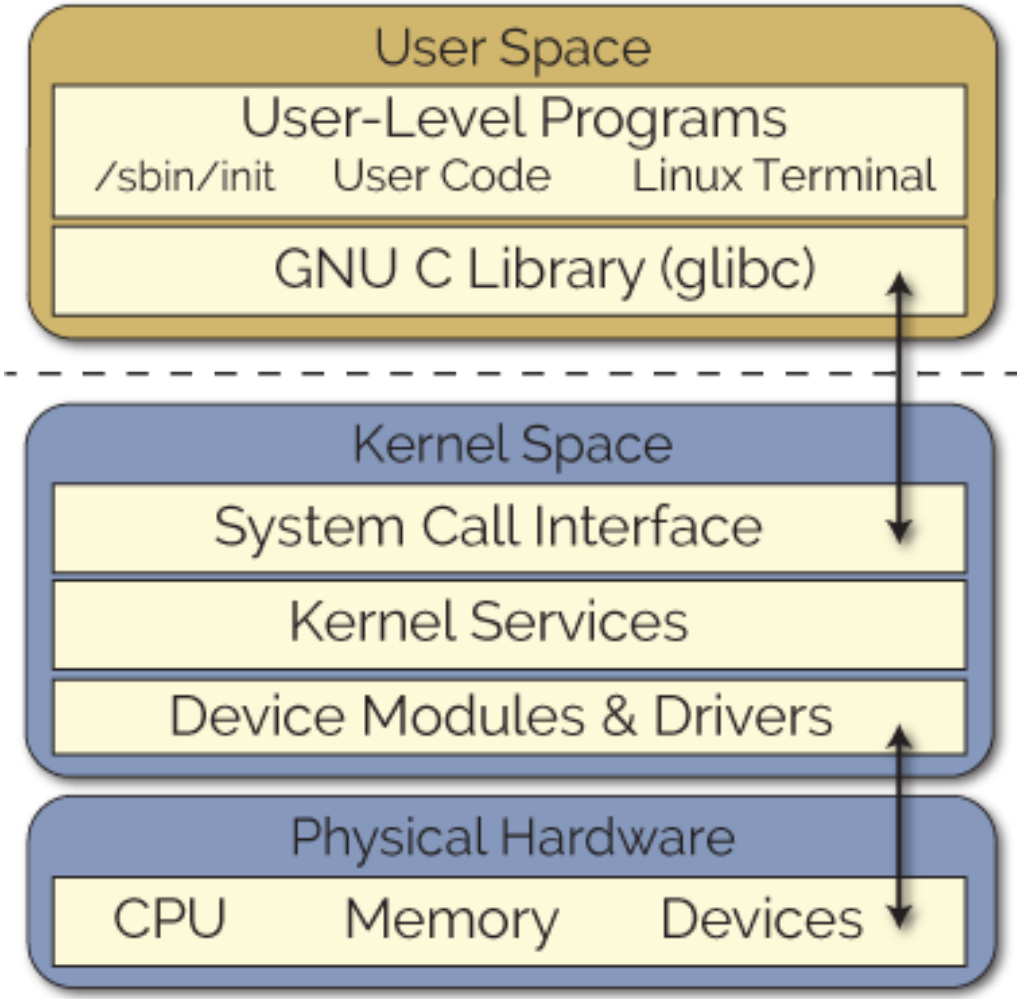


Figure: Kernel Space & User Space.

Comandos utiles

1. *lsmod*: Lista los módulos.

Comandos utiles

1. *lsmod*: Lista los módulos.
2. *insmod* $<$ *nombredemodulo* $>$: Carga el módulo.

Comandos utiles

1. *lsmod*: Lista los módulos.
2. *insmod* $<$ *nombredemodulo* $>$: Carga el módulo.
3. *modprobe* $<$ *nombredemodulo* $>$: Carga el módulo y dependencias.

Comandos utiles

1. *lsmod*: Lista los módulos.
2. *insmod* $<$ *nombredemodulo* $>$: Carga el módulo.
3. *modprobe* $<$ *nombredemodulo* $>$: Carga el módulo y dependencias.
4. *rmmod* $<$ *nombredemodulo* $>$: Descarga el módulo.

Comandos utiles

1. *lsmod*: Lista los módulos.
2. *insmod* $<$ *nombredemodulo* $>$: Carga el módulo.
3. *modprobe* $<$ *nombredemodulo* $>$: Carga el módulo y dependencias.
4. *rmmod* $<$ *nombredemodulo* $>$: Descarga el módulo.
5. Con **dmesg** vemos el log del kernel.

Módulo "hola mundo"

1. Para hacer nuestro driver primero haremos un modulo

Módulo "hola mundo"

1. Para hacer nuestro driver primero haremos un modulo
2. No podemos utilizar las funciones de user space (**stdio.h**)

Módulo "hola mundo"

1. Para hacer nuestro driver primero haremos un modulo
2. No podemos utilizar las funciones de user space (**stdio.h**)
3. Utilizamos **kernel.h** (¿Que funciones nos proporciona ?)

Módulo "hola mundo"

1. Para hacer nuestro driver primero haremos un modulo
2. No podemos utilizar las funciones de user space (**stdio.h**)
3. Utilizamos **kernel.h** (¿Que funciones nos proporciona ?)
4. No tenemos **main**, tenemos **init_module** y **cleanup_module**.

Módulo "hola mundo"

1. Para hacer nuestro driver primero haremos un modulo
2. No podemos utilizar las funciones de user space (**stdio.h**)
3. Utilizamos **kernel.h** (¿Que funciones nos proporciona ?)
4. No tenemos **main**, tenemos **init_module** y **cleanup_module**.
5. Son llamadas en el insmod y el rmmod

module.c

```c
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{ /* Constructor */
    printk(KERN_INFO "UNGS: Driver registrado\n");
    return 0;
}

void cleanup_module(void)
{/* Destructor */
    printk(KERN_INFO "UNGS: Driver desregistrado\n");
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("UNGS");
MODULE_DESCRIPTION("Un primer driver");
```

Para comenzar

1. Averiguar version del kernel con **uname -r**

Para comenzar

1. Averiguar version del kernel con **uname -r**
2. Instalar module-init-tools linux-headers-(**version del kernel**)

Para comenzar

1. Averiguar version del kernel con **uname -r**
2. Instalar module-init-tools linux-headers-(**version del kernel**)
3. Crear un archivo llamado Makefile

Para comenzar

1. Averiguar version del kernel con **uname -r**
2. Instalar module-init-tools linux-headers-(**version del kernel**)
3. Crear un archivo llamado Makefile
4. Ejecutar *make clean & make;*

Makefile

```
obj-m := miModulo.o

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

Char Devices

1. ¿Como funciona?
   - Transfiere datos bloque por bloque

Char Devices

1. ¿Como funciona?
   - Transfiere datos bloque por bloque
   - Se comportan como pipes, lectura en serie ej. lectora CD
2. ¿Que operaciones tiene ?

Char Devices

1. ¿Como funciona?
   - Transfiere datos bloque por bloque
   - Se comportan como pipes, lectura en serie ej. lectora CD
2. ¿Que operaciones tiene ?

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *)
    ;
```

https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

Char Devices

1. ¿Como funciona?
   - Transfiere datos bloque por bloque
   - Se comportan como pipes, lectura en serie ej. lectora CD
2. ¿Que operaciones tiene ?

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *)
    ;
```

https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

4. A diferencia de un block device no poseen **seek**

Char Devices

1. ¿Como funciona?
   - Transfiere datos bloque por bloque
   - Se comportan como pipes, lectura en serie ej. lectora CD
2. ¿Que operaciones tiene ?

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *)
    ;
```

https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

4. A diferencia de un block device no poseen **seek**
5. Un device como vimos tiene asociado un archivo en **/dev**

Char Devices

1. ¿Como funciona?
   - Transfiere datos bloque por bloque
   - Se comportan como pipes, lectura en serie ej. lectora CD
2. ¿Que operaciones tiene ?

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *)
    ;
```

https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

4. A diferencia de un block device no poseen **seek**
5. Un device como vimos tiene asociado un archivo en **/dev**
   - El FS para saber que driver usar necesita un **Major Number**

Char Devices

1. ¿Como funciona?
   - Transfiere datos bloque por bloque
   - Se comportan como pipes, lectura en serie ej. lectora CD
2. ¿Que operaciones tiene ?

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *)
    ;
```

https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

4. A diferencia de un block device no poseen **seek**
5. Un device como vimos tiene asociado un archivo en **/dev**
   - El FS para saber que driver usar necesita un **Major Number**
   - También poseen un **Minor Number**, este es interno del driver

Char Devices

1. ¿Como funciona?
   - Transfiere datos bloque por bloque
   - Se comportan como pipes, lectura en serie ej. lectora CD
2. ¿Que operaciones tiene ?

```
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *)
    ;
```

https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

4. A diferencia de un block device no poseen **seek**
5. Un device como vimos tiene asociado un archivo en **/dev**
   - El FS para saber que driver usar necesita un **Major Number**
   - También poseen un **Minor Number**, este es interno del driver
   - Para crearlo *mknod* **/dev/nombre** nombre **major minor**

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*
   - *copy_from_user(to,from,bytes);*

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*
   - *copy_from_user(to,from,bytes);*
   - *put_user(value,destination);*

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*
   - *copy_from_user(to,from,bytes);*
   - *put_user(value,destination);*
   - *get_user(value,source);*

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*
   - *copy_from_user(to,from,bytes);*
   - *put_user(value,destination);*
   - *get_user(value,source);*
3. Para estas funciones necesito linux/uaccess.h ojo no ASM

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*
   - *copy_from_user(to,from,bytes);*
   - *put_user(value,destination);*
   - *get_user(value,source);*
3. Para estas funciones necesito linux/uaccess.h ojo no ASM
4. Y como operamos con el FS linux/fs.h

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*
   - *copy_from_user(to,from,bytes);*
   - *put_user(value,destination);*
   - *get_user(value,source);*
3. Para estas funciones necesito linux/uaccess.h ojo no ASM
4. Y como operamos con el FS linux/fs.h
   - *try_module_get(THIS_MODULE);*

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*
   - *copy_from_user(to,from,bytes);*
   - *put_user(value,destination);*
   - *get_user(value,source);*
3. Para estas funciones necesito linux/uaccess.h ojo no ASM
4. Y como operamos con el FS linux/fs.h
   - *try_module_get(THIS_MODULE);*
   - *module_put(THIS_MODULE);*

Punto de partida

1. ¿Si los módulos viven en el kernel, como puedo hacer read?
2. Para poder leer/escribir tengo que poner el buffer en el userspace/kernelspace
   - *copy_to_user(to,from,bytes);*
   - *copy_from_user(to,from,bytes);*
   - *put_user(value,destination);*
   - *get_user(value,source);*
3. Para estas funciones necesito linux/uaccess.h ojo no ASM
4. Y como operamos con el FS linux/fs.h
   - *try_module_get(THIS_MODULE);*
   - *module_put(THIS_MODULE);*
5. Esta todo en The linux kernel Module Programing Guide (LKMPG) https://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf.

Manos a la obra