

Taller de drivers

Sistemas Operativos

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

12 de Abril de 2016

Agenda para hoy

Vamos a hacer un driver !

Agenda para hoy

Vamos a hacer un driver !

- 1 Breve repaso teórico

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices
 - ▶ device numbers

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices
 - ▶ device numbers
 - ▶ operaciones de *char* devices

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices
 - ▶ device numbers
 - ▶ operaciones de *char* devices
 - ▶ la estructura *cdev*

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices
 - ▶ device numbers
 - ▶ operaciones de *char* devices
 - ▶ la estructura *cdev*
 - ▶ reserva de nodos

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices
 - ▶ device numbers
 - ▶ operaciones de *char* devices
 - ▶ la estructura *cdev*
 - ▶ reserva de nodos
 - ▶ memoria dinámica

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices
 - ▶ device numbers
 - ▶ operaciones de *char* devices
 - ▶ la estructura *cdev*
 - ▶ reserva de nodos
 - ▶ memoria dinámica
 - ▶ sincronización

Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices
 - ▶ device numbers
 - ▶ operaciones de *char* devices
 - ▶ la estructura *cdev*
 - ▶ reserva de nodos
 - ▶ memoria dinámica
 - ▶ sincronización

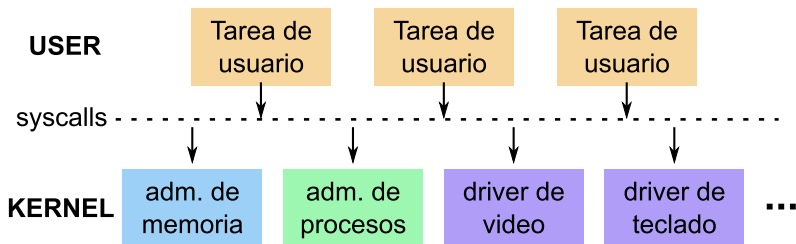
Agenda para hoy

Vamos a hacer un driver !

- ① Breve repaso teórico
- ② Módulos
 - ▶ escritura de módulos
 - ▶ carga y liberación de módulos
- ③ Devices
 - ▶ tipos de devices
 - ▶ device numbers
 - ▶ operaciones de *char* devices
 - ▶ la estructura *cdev*
 - ▶ reserva de nodos
 - ▶ memoria dinámica
 - ▶ sincronización
- ④ Taller

La estructura de Linux

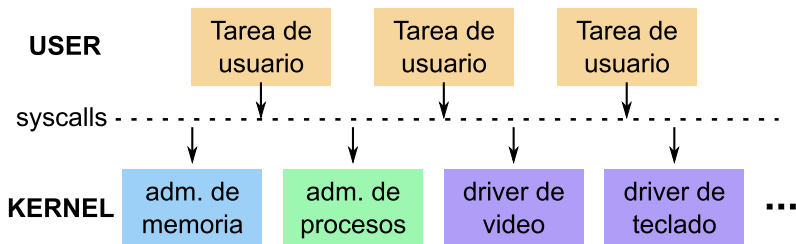
En **linux** hay sólo **dos** espacios: user y kernel.



- Kernel **monolítico**: administradores, memoria, drivers, todo junto.

La estructura de Linux

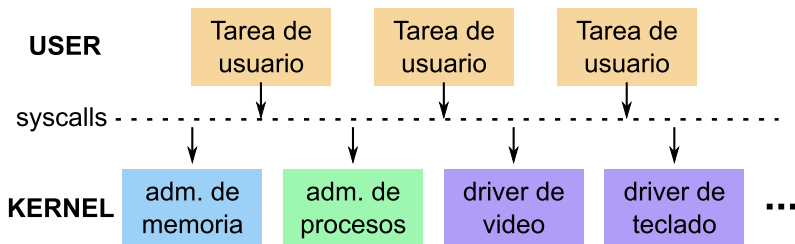
En **linux** hay sólo **dos** espacios: user y kernel.



- Kernel **monolítico**: administradores, memoria, drivers, todo junto.
- **El** kernel tiene acceso a todo:

La estructura de Linux

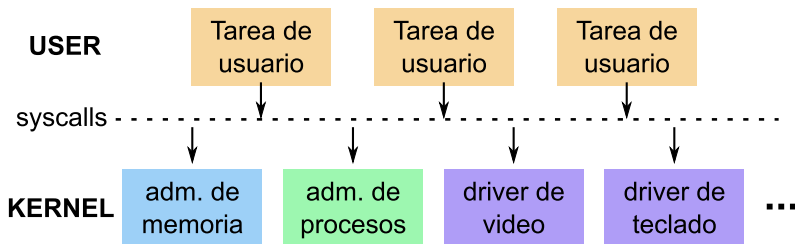
En **linux** hay sólo **dos** espacios: user y kernel.



- Kernel **monolítico**: administradores, memoria, drivers, todo junto.
- **El** kernel tiene acceso a todo:
 - ▶ ejecuta en un **único** espacio de memoria

La estructura de Linux

En **linux** hay sólo **dos** espacios: user y kernel.



- Kernel **monolítico**: administradores, memoria, drivers, todo junto.
- **El** kernel tiene acceso a todo:
 - ▶ ejecuta en un **único** espacio de memoria
 - ▶ ejecuta en el **máximo** nivel de privilegio

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video
- Decenas de drivers de otros dispositivos físicamente instalados

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video
- Decenas de drivers de otros dispositivos físicamente instalados

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video
- Decenas de drivers de otros dispositivos físicamente instalados

Situación: Enchufo un pendrive USB. Necesita un driver nuevo.

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video
- Decenas de drivers de otros dispositivos físicamente instalados

Situación: Enchufo un pendrive USB. Necesita un driver nuevo.

- ¿Tengo que reiniciar la máquina y cargar de nuevo el kernel?

La estructura de Linux (2)

El kernel se carga al iniciar. Entonces se debería cargar:

- Administrador de memoria
- Administrador de procesos (scheduler, etc.)
- Driver del teclado
- Driver de la placa de video
- Decenas de drivers de otros dispositivos físicamente instalados

Situación: Enchufo un pendrive USB. Necesita un driver nuevo.

- ¿Tengo que reiniciar la máquina y cargar de nuevo el kernel?
- ¿Tengo que recompilar el kernel?

La estructura de Linux (3)

Si el kernel está todo contenido en **un** gran archivo binario:

- ¿Qué pasa si quiero agregar funcionalidad cuando ya estoy usando la máquina?

La estructura de Linux (3)

Si el kernel está todo contenido en **un** gran archivo binario:

- ¿Qué pasa si quiero agregar funcionalidad cuando ya estoy usando la máquina?
- ¿Qué pasa si incluyo funcionalidad “por las dudas”?

La estructura de Linux (3)

Si el kernel está todo contenido en **un** gran archivo binario:

- ¿Qué pasa si quiero agregar funcionalidad cuando ya estoy usando la máquina?
- ¿Qué pasa si incluyo funcionalidad “por las dudas”?

La estructura de Linux (3)

Si el kernel está todo contenido en **un** gran archivo binario:

- ¿Qué pasa si quiero agregar funcionalidad cuando ya estoy usando la máquina?
- ¿Qué pasa si incluyo funcionalidad “por las dudas”?

Solución: (de linux)

Linux soporta la carga y descarga de **módulos** al kernel en tiempo de ejecución.

- ¿Qué cosas componen a un **módulo**?

- ¿Qué cosas componen a un **módulo**?
 - ▶ Puntos de entrada y salida
 - ▶ Datos
 - ▶ Funciones relacionadas

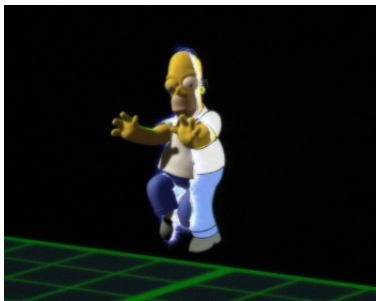
- ¿Qué cosas componen a un **módulo**?
 - ▶ Puntos de entrada y salida
 - ▶ Datos
 - ▶ Funciones relacionadas
- ¿A causa de qué podría ejecutarse el código de un módulo?

- ¿Qué cosas componen a un **módulo**?
 - ▶ Puntos de entrada y salida
 - ▶ Datos
 - ▶ Funciones relacionadas
- ¿A causa de qué podría ejecutarse el código de un módulo?
 - 1 Llamada al sistema
 - 2 Atención de interrupción

- ¿Qué cosas componen a un **módulo**?
 - ▶ Puntos de entrada y salida
 - ▶ Datos
 - ▶ Funciones relacionadas
- ¿A causa de qué podría ejecutarse el código de un módulo?
 - 1 Llamada al sistema
 - 2 Atención de interrupción
- ¿Qué funcionalidades podría brindar un módulo?

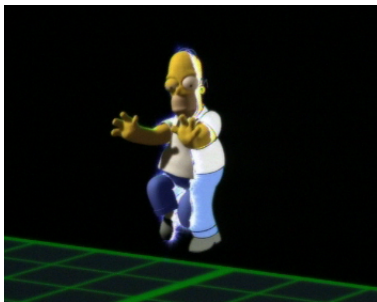
- ¿Qué cosas componen a un **módulo**?
 - ▶ Puntos de entrada y salida
 - ▶ Datos
 - ▶ Funciones relacionadas
- ¿A causa de qué podría ejecutarse el código de un módulo?
 - 1 Llamada al sistema
 - 2 Atención de interrupción
- ¿Qué funcionalidades podría brindar un módulo?
- Hoy vamos a escribir nuestro primer módulo...

Un nuevo mundo...(parecido al de orga 2 pero no)



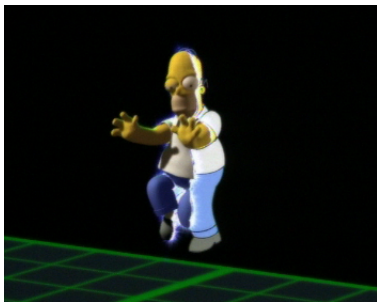
- Estamos ejecutando en el **nivel de máximo privilegio**

Un nuevo mundo...(parecido al de orga 2 pero no)



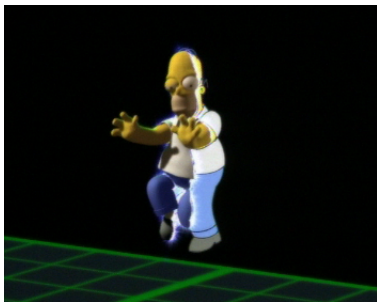
- Estamos ejecutando en el **nivel de máximo privilegio**
- El kernel no está enlazado a la libc

Un nuevo mundo...(parecido al de orga 2 pero no)



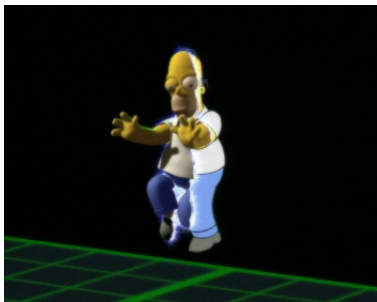
- Estamos ejecutando en el **nivel de máximo privilegio**
- El kernel no está enlazado a la libc
- Hacer **operaciones de punto flotante** es más complicado

Un nuevo mundo...(parecido al de orga 2 pero no)



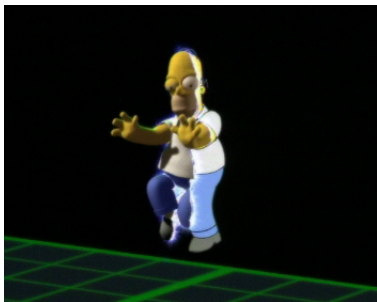
- Estamos ejecutando en el **nivel de máximo privilegio**
- El kernel no está enlazado a la libc
- Hacer **operaciones de punto flotante** es más complicado
- Tenemos un **stack fijo y limitado** (y tenemos que compartirlo con el resto del kernel)

Un nuevo mundo...(parecido al de orga 2 pero no)



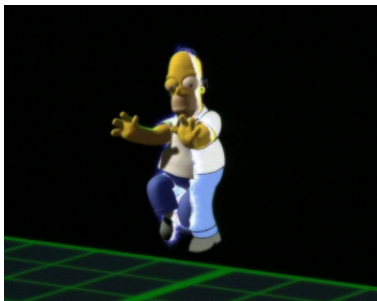
- Estamos ejecutando en el **nivel de máximo privilegio**
- El kernel no está enlazado a la libc
- Hacer **operaciones de punto flotante** es más complicado
- Tenemos un **stack fijo y limitado** (y tenemos que compartirlo con el resto del kernel)
- Hay varias fuentes de posibles **condiciones de carrera**

Un nuevo mundo...(parecido al de orga 2 pero no)



- Estamos ejecutando en el **nivel de máximo privilegio**
- El kernel no está enlazado a la libc
- Hacer **operaciones de punto flotante** es más complicado
- Tenemos un **stack fijo y limitado** (y tenemos que compartirlo con el resto del kernel)
- Hay varias fuentes de posibles **condiciones de carrera**

Un nuevo mundo...(parecido al de orga 2 pero no)



- Estamos ejecutando en el **nivel de máximo privilegio**
- El kernel no está enlazado a la libc
- Hacer **operaciones de punto flotante** es más complicado
- Tenemos un **stack fijo y limitado** (y tenemos que compartirlo con el resto del kernel)
- Hay varias fuentes de posibles **condiciones de carrera**

¿Qué pasa si hacemos un **acceso indebido a memoria**?



Nuestro primer módulo

Nuestro primer módulo

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

static int __init hello_init(void) {
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_ALERT "Adios, mundo cruel...\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Juan de los Palotes");
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

Nuestro primer módulo (2)

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

- `init.h` contiene la definición de las macros `module_init()` y `module_exit()`

Nuestro primer módulo (2)

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

- `init.h` contiene la definición de las macros `module_init()` y `module_exit()`
- `module.h` contiene varias definiciones necesarias para la gran mayoría de los módulos (por ejemplo, varios `MODULE_*`)

Nuestro primer módulo (2)

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

- `init.h` contiene la definición de las macros `module_init()` y `module_exit()`
- `module.h` contiene varias definiciones necesarias para la gran mayoría de los módulos (por ejemplo, varios `MODULE_*`)
- `kernel.h` contiene la declaración de `printk()`

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Leandro Lera Romero");  
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

- `MODULE_AUTHOR()` y `MODULE_DESCRIPTION()` son meramente informativos

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Leandro Lera Romero");  
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

- MODULE_AUTHOR() y MODULE_DESCRIPTION() son meramente informativos
- MODULE_LICENSE() indica la licencia del módulo;

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Leandro Lera Romero");  
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

- `MODULE_AUTHOR()` y `MODULE_DESCRIPTION()` son meramente informativos
- `MODULE_LICENSE()` indica la licencia del módulo;
 - ▶ algunos valores posibles son:

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Leandro Lera Romero");  
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

- `MODULE_AUTHOR()` y `MODULE_DESCRIPTION()` son meramente informativos
- `MODULE_LICENSE()` indica la licencia del módulo;
 - ▶ algunos valores posibles son:
 - ★ GPL

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Leandro Lera Romero");  
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

- `MODULE_AUTHOR()` y `MODULE_DESCRIPTION()` son meramente informativos
- `MODULE_LICENSE()` indica la licencia del módulo;
 - ▶ algunos valores posibles son:
 - ★ GPL
 - ★ Dual BSD/GPL

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Leandro Lera Romero");  
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

- `MODULE_AUTHOR()` y `MODULE_DESCRIPTION()` son meramente informativos
- `MODULE_LICENSE()` indica la licencia del módulo;
 - ▶ algunos valores posibles son:
 - ★ GPL
 - ★ Dual BSD/GPL
 - ★ Proprietary

Nuestro primer módulo (3)

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Leandro Lera Romero");  
MODULE_DESCRIPTION("Una suerte de 'Hola, mundo'");
```

- `MODULE_AUTHOR()` y `MODULE_DESCRIPTION()` son meramente informativos
- `MODULE_LICENSE()` indica la licencia del módulo;
 - ▶ algunos valores posibles son:
 - ★ GPL
 - ★ Dual BSD/GPL
 - ★ Proprietary
 - ▶ un módulo con una licencia propietaria “mancha” el kernel

Nuestro primer módulo (4)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- `static` indica que la función es local al archivo (opcional)

Nuestro primer módulo (4)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- `static` indica que la función es local al archivo (opcional)
- `__init` e `__initdata` le indican al kernel que sólo se usará al momento de la inicialización, y que puede olvidarse una vez cargado el módulo (opcional)

Nuestro primer módulo (4)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- `static` indica que la función es local al archivo (opcional)
- `__init` e `__initdata` le indican al kernel que sólo se usará al momento de la inicialización, y que puede olvidarse una vez cargado el módulo (opcional)
- `printk()` se comporta de manera similar a la función `printf()` de la *libc*, pero permite indicar niveles de prioridad:

Nuestro primer módulo (4)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- `static` indica que la función es local al archivo (opcional)
- `__init` e `__initdata` le indican al kernel que sólo se usará al momento de la inicialización, y que puede olvidarse una vez cargado el módulo (opcional)
- `printk()` se comporta de manera similar a la función `printf()` de la *libc*, pero permite indicar niveles de prioridad:
 - ▶ `KERN_ALERT` – problema de atención inmediata

Nuestro primer módulo (4)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- `static` indica que la función es local al archivo (opcional)
- `__init` e `__initdata` le indican al kernel que sólo se usará al momento de la inicialización, y que puede olvidarse una vez cargado el módulo (opcional)
- `printk()` se comporta de manera similar a la función `printf()` de la *libc*, pero permite indicar niveles de prioridad:
 - ▶ `KERN_ALERT` – problema de atención inmediata
 - ▶ `KERN_INFO` – mensaje con información

Nuestro primer módulo (4)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- `static` indica que la función es local al archivo (opcional)
- `__init` e `__initdata` le indican al kernel que sólo se usará al momento de la inicialización, y que puede olvidarse una vez cargado el módulo (opcional)
- `printk()` se comporta de manera similar a la función `printf()` de la *libc*, pero permite indicar niveles de prioridad:
 - ▶ `KERN_ALERT` – problema de atención inmediata
 - ▶ `KERN_INFO` – mensaje con información
 - ▶ `KERN_DEBUG` – mensaje de *debug*

Nuestro primer módulo (5)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- Con `module_init()` se indica dónde encontrar la **función de inicialización** del módulo

Nuestro primer módulo (5)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- Con `module_init()` se indica dónde encontrar la **función de inicialización** del módulo
- La función de inicialización es llamada:

Nuestro primer módulo (5)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- Con `module_init()` se indica dónde encontrar la **función de inicialización** del módulo
- La función de inicialización es llamada:
 - ▶ al arrancar el sistema

Nuestro primer módulo (5)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- Con `module_init()` se indica dónde encontrar la **función de inicialización** del módulo
- La función de inicialización es llamada:
 - ▶ al arrancar el sistema
 - ▶ al insertar el módulo

Nuestro primer módulo (5)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- Con `module_init()` se indica dónde encontrar la **función de inicialización** del módulo
- La función de inicialización es llamada:
 - ▶ al arrancar el sistema
 - ▶ al insertar el módulo
- Su rol es registrar recursos, inicializar hardware, reservar espacio en memoria para estructuras de datos, etc.

Nuestro primer módulo (5)

```
static int __init hello_init(void) {  
    printk(KERN_ALERT "Hola, Sistemas Operativos!\n");  
    return 0;  
}  
  
module_init(hello_init);
```

- Con `module_init()` se indica dónde encontrar la **función de inicialización** del módulo
- La función de inicialización es llamada:
 - ▶ al arrancar el sistema
 - ▶ al insertar el módulo
- Su rol es registrar recursos, inicializar hardware, reservar espacio en memoria para estructuras de datos, etc.
- Si todo salió bien, tiene que devolver 0; si no, tiene que volver atrás lo que cambió y devolver algo distinto de cero.

Nuestro primer módulo (6)

```
static void __exit hello_exit(void) {  
    printk(KERN_ALERT "Adios, mundo cruel...\n");  
}  
  
module_exit(hello_exit);
```

- Con `module_exit()` se indica dónde encontrar la **función de “limpieza”** del módulo

Nuestro primer módulo (6)

```
static void __exit hello_exit(void) {  
    printk(KERN_ALERT "Adios, mundo cruel...\n");  
}  
  
module_exit(hello_exit);
```

- Con `module_exit()` se indica dónde encontrar la **función de “limpieza”** del módulo
- La función de “limpieza” es llamada antes de quitar el módulo

Nuestro primer módulo (6)

```
static void __exit hello_exit(void) {  
    printk(KERN_ALERT "Adios, mundo cruel...\n");  
}  
  
module_exit(hello_exit);
```

- Con `module_exit()` se indica dónde encontrar la **función de “limpieza”** del módulo
- La función de “limpieza” es llamada antes de quitar el módulo
- Se ocupa de deshacer/limpiar todo lo que la función de inicialización y el resto del módulo usaron

Inyectando módulos al kernel

¿Cómo cargamos nuestro módulo al kernel?

- `insmod` carga el código y los datos de nuestro módulo al kernel

Inyectando módulos al kernel

¿Cómo cargamos nuestro módulo al kernel?

- `insmod` carga el código y los datos de nuestro módulo al kernel
- el kernel usa su tabla de símbolos para enlazar todas las referencias no resueltas del módulo

Inyectando módulos al kernel

¿Cómo cargamos nuestro módulo al kernel?

- `insmod` carga el código y los datos de nuestro módulo al kernel
- el kernel usa su tabla de símbolos para enlazar todas las referencias no resueltas del módulo
- una vez cargado, se llama a su función de inicialización

Inyectando módulos al kernel

¿Cómo cargamos nuestro módulo al kernel?

- `insmod` carga el código y los datos de nuestro módulo al kernel
- el kernel usa su tabla de símbolos para enlazar todas las referencias no resueltas del módulo
- una vez cargado, se llama a su función de inicialización
- `rmmmod` permite quitar el módulo del kernel si esto es posible (por ejemplo, falla si el módulo está siendo usado)

Compilando nuestro módulo

- 1 Necesitamos

Compilando nuestro módulo

1 Necesitamos

- ▶ `make`

Compilando nuestro módulo

1 Necesitamos

- ▶ `make`
- ▶ `module-init-tools`

Compilando nuestro módulo

1 Necesitamos

- ▶ make
- ▶ module-init-tools
- ▶ linux-headers-<version>
(<version> sale de `uname -r`)

Compilando nuestro módulo

1 Necesitamos

- ▶ make
- ▶ module-init-tools
- ▶ linux-headers-<version>
(<version> sale de `uname -r`)

2 crear un Makefile con el siguiente contenido:

Compilando nuestro módulo

1 Necesitamos

- ▶ make
- ▶ module-init-tools
- ▶ linux-headers-<version>
(<version> sale de `uname -r`)

2 crear un Makefile con el siguiente contenido:

```
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
```

Compilando nuestro módulo

1 Necesitamos

- ▶ make
- ▶ module-init-tools
- ▶ linux-headers-<version>
(<version> sale de `uname -r`)

2 crear un Makefile con el siguiente contenido:

```
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
```

3 ejecutar `make clean` y `make`

Compilando nuestro módulo

1 Necesitamos

- ▶ make
- ▶ module-init-tools
- ▶ linux-headers-<version>
(<version> sale de `uname -r`)

2 crear un Makefile con el siguiente contenido:

```
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
```

3 ejecutar `make clean` y `make`

4 usar `insmod` y `rmmod`

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

- ▶ direccionables de a “cachos” definidos

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

- ▶ direccionables de a “cachos” definidos
- ▶ suelen soportar *seeking*

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

- ▶ direccionables de a “cachos” definidos
- ▶ suelen soportar *seeking*
- ▶ generalmente, su nodo es montado como un filesystem

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

- ▶ direccionables de a “cachos” definidos
- ▶ suelen soportar *seeking*
- ▶ generalmente, su nodo es montado como un filesystem

- **network devices**

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

- ▶ direccionables de a “cachos” definidos
- ▶ suelen soportar *seeking*
- ▶ generalmente, su nodo es montado como un filesystem

- **network devices**

- ▶ proveen acceso a una red

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

- ▶ direccionables de a “cachos” definidos
- ▶ suelen soportar *seeking*
- ▶ generalmente, su nodo es montado como un filesystem

- **network devices**

- ▶ proveen acceso a una red
- ▶ no son accedidos a través de un nodo en el filesystem, sino de otra manera (usando sockets, por ejemplo)

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

- ▶ direccionables de a “cachos” definidos
- ▶ suelen soportar *seeking*
- ▶ generalmente, su nodo es montado como un filesystem

- **network devices**

- ▶ proveen acceso a una red
- ▶ no son accedidos a través de un nodo en el filesystem, sino de otra manera (usando sockets, por ejemplo)

Tipos de *devices*

En UNIX, comúnmente:

- **char devices**

- ▶ pueden accederse como una tira de bytes
- ▶ suelen no soportar *seeking*
- ▶ se los usa directamente mediante un nodo en el filesystem
- ▶ tienen un subtipo interesante: **misc devices**

- **block devices**

- ▶ direccionables de a “cachos” definidos
- ▶ suelen soportar *seeking*
- ▶ generalmente, su nodo es montado como un filesystem

- **network devices**

- ▶ proveen acceso a una red
- ▶ no son accedidos a través de un nodo en el filesystem, sino de otra manera (usando sockets, por ejemplo)

Podemos ver ejemplos con `ls -l /dev`

devices y drivers

```
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
...
```

devices y drivers

```
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 de cada línea representa el tipo de archivo:

- l es un *symlink* (enlace simbólico)
- c es un *char device*
- b es un *block device*

devices y drivers

```
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 de cada línea representa el tipo de archivo:

- l es un *symlink* (enlace simbólico)
- c es un *char device*
- b es un *block device*

Los *devices* tienen un par de números asociados:

- **major**: está asociado a un driver en particular (primer número luego del grupo)
- **minor**: identifica a un dispositivo específico que el driver maneja (segundo número luego del grupo)

Operando un *char device*

Vamos a construir un driver de un *char device*. ¿Qué necesitamos?

Operando un *char device*

Vamos a construir un driver de un *char device*. ¿Qué necesitamos?

- un char device (fakeserial provisto por la catedra)

Operando un *char device*

Vamos a construir un driver de un *char device*. ¿Qué necesitamos?

- un char device (fakeserial provisto por la catedra)
- registrar al *device* como un *char device*

Operando un *char device*

Vamos a construir un driver de un *char device*. ¿Qué necesitamos?

- un *char device* (fakeserial provisto por la catedra)
- registrar al *device* como un *char device*
- registrar las funciones de cada operación que queramos realizar sobre el *device*

Operando un *char device*

Vamos a construir un driver de un *char device*. ¿Qué necesitamos?

- un char device (fakeserial provisto por la catedra)
- registrar al *device* como un *char device*
- registrar las funciones de cada operación que queramos realizar sobre el *device*

Operando un *char device*

Vamos a construir un driver de un *char device*. ¿Qué necesitamos?

- un *char device* (fakeserial provisto por la catedra)
- registrar al *device* como un *char device*
- registrar las funciones de cada operación que queramos realizar sobre el *device*

¿Qué parte del módulo debería encargarse de lo anterior?

Registrar el device

En el código del kernel,

```
misc_register(miscdevice device);  
misc_deregister(miscdevice device);  
struct miscdevice {  
    MISC_DYNAMIC_MINOR, //esto nos asigna un MINOR  
    DEVICE_NAME, //nombre del dispositivo  
    &fakeserial_fops,  
};
```

Las operaciones

```
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 *);  
    ...  
}
```

- la estructura `file_operations` representa las operaciones que las aplicaciones pueden realizar sobre los *devices*

Las operaciones

```
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 *);  
    ...  
}
```

- la estructura `file_operations` representa las operaciones que las aplicaciones pueden realizar sobre los *devices*
- cada campo apunta a una función en nuestro módulo que se encarga de la operación, o es `NULL`

Las operaciones

```
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 *);  
    ...  
}
```

- la estructura `file_operations` representa las operaciones que las aplicaciones pueden realizar sobre los *devices*
- cada campo apunta a una función en nuestro módulo que se encarga de la operación, o es `NULL`
- si el campo es `NULL` tiene lugar una operación por omisión distinta para cada campo

Las operaciones (2)

```
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 *);  
    ...  
}
```

- owner: un puntero al módulo “dueño” de la estructura (generalmente THIS_MODULE)

Las operaciones (2)

```
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 *);  
    ...  
}
```

- owner: un puntero al módulo “dueño” de la estructura (generalmente THIS_MODULE)
- read(): para recibir datos desde el *device*; retorna el número de bytes leídos

Las operaciones (2)

```
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 *);  
    ...  
}
```

- `owner`: un puntero al módulo “dueño” de la estructura (generalmente `THIS_MODULE`)
- `read()`: para recibir datos desde el *device*; retorna el número de bytes leídos
- `write()`: para enviar datos al *device*; retorna el número de bytes escritos

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario
- el puntero a espacio de usuario puede:

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario
- el puntero a espacio de usuario puede:
 - ▶ ser inválido: puede no haber nada mapeado en esa dirección, o puede haber basura;

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario
- el puntero a espacio de usuario puede:
 - ▶ ser inválido: puede no haber nada mapeado en esa dirección, o puede haber basura;
 - ▶ no estar en memoria (paginado), y el kernel no puede incurrir en *page faults*;

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario
- el puntero a espacio de usuario puede:
 - ▶ ser inválido: puede no haber nada mapeado en esa dirección, o puede haber basura;
 - ▶ no estar en memoria (paginado), y el kernel no puede incurrir en *page faults*;
 - ▶ ser erróneo o malicioso

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario
- el puntero a espacio de usuario puede:
 - ▶ ser inválido: puede no haber nada mapeado en esa dirección, o puede haber basura;
 - ▶ no estar en memoria (paginado), y el kernel no puede incurrir en *page faults*;
 - ▶ ser erróneo o malicioso
- para estar tranquilos, hay que usar:

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario
- el puntero a espacio de usuario puede:
 - ▶ ser inválido: puede no haber nada mapeado en esa dirección, o puede haber basura;
 - ▶ no estar en memoria (paginado), y el kernel no puede incurrir en *page faults*;
 - ▶ ser erróneo o malicioso
- para estar tranquilos, hay que usar:

Las operaciones (3)

- tanto `read()` como `write()` escriben en o leen de la memoria de usuario
- el puntero a espacio de usuario puede:
 - ▶ ser inválido: puede no haber nada mapeado en esa dirección, o puede haber basura;
 - ▶ no estar en memoria (paginado), y el kernel no puede incurrir en *page faults*;
 - ▶ ser erróneo o malicioso
- para estar tranquilos, hay que usar:

```
unsigned long copy_to_user(void __user *to, const void *from,  
    unsigned long count);  
unsigned long copy_from_user(void *to, const void __user *from,  
    unsigned long count);
```

Sincronización (kernel)

Diversos mecanismos de sincronización. Entre ellos semáforos y *mutexes*.

`semaphore`

Tipo de datos: `struct semaphore`. Funciones: `sema_init(struct semaphore * sem, int val)`, `down(struct semaphore * sem)`, `down_interruptible(struct semaphore * sem)`, ..., `up(struct semaphore * sem)`

Montando filesystems remotos

Vamos a acceder a nuestros archivos en la computadora *física* desde la *virtual*.

```
# apt-get install sshfs
# mkdir remoto
# sshfs MI_USUARIO@IP_HOST:/home/MI_USUARIO remoto
```

- Reemplazar *MI_USUARIO* por su cuenta de los labos e *IP_HOST* por la dirección IP de la máquina que están usando.
- El comando *ifconfig* es su amigo.
- Este comando logra abstraer la idea de que los archivos están en una ubicación remota, para el SO son archivos comunes y corrientes.
- Ahora pueden editar desde su usuario de los labos y solo compilar para probar en la consola de la máquina virtual.