

Taller de drivers

Sistemas Operativos

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

11 de Mayo de 2023

Tabla de Contenidos

1 Intro

2 Drivers

3 Módulos

- ¿Qué es un módulo?
- Escribiendo nuestro primer módulo
- Compilando y ejecutando el módulo

4 Devices

- ¿Cómo se representan los devices?
- Creación de un device
- Acceder al device

5 Recomendaciones

Tabla de Contenidos

1 Intro

2 Drivers

3 Módulos

- ¿Qué es un módulo?
- Escribiendo nuestro primer módulo
- Compilando y ejecutando el módulo

4 Devices

- ¿Cómo se representan los devices?
- Creación de un device
- Acceder al device

5 Recomendaciones

¿Qué vimos?

- Primera parte de la materia:

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling
 - ▶ IPC (y threads)

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling
 - ▶ IPC (y threads)
 - ▶ Memoria

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling
 - ▶ IPC (y threads)
 - ▶ Memoria
- Segunda parte de la materia:

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling
 - ▶ IPC (y threads)
 - ▶ Memoria
- Segunda parte de la materia:
 - ▶ Entrada / Salida (orga 1) y Repaso de diseño E/S

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling
 - ▶ IPC (y threads)
 - ▶ Memoria
- Segunda parte de la materia:
 - ▶ Entrada / Salida (orga 1) y Repaso de diseño E/S
 - ▶ Drivers

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling
 - ▶ IPC (y threads)
 - ▶ Memoria
- Segunda parte de la materia:
 - ▶ Entrada / Salida (orga 1) y Repaso de diseño E/S
 - ▶ Drivers
- ¿Con qué están atrasados? ☺

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling
 - ▶ IPC (y threads)
 - ▶ Memoria
- Segunda parte de la materia:
 - ▶ Entrada / Salida (orga 1) y Repaso de diseño E/S
 - ▶ Drivers
- ¿Con qué están atrasados? 😊
 - ▶ Práctica de E/S

¿Qué vimos?

- Primera parte de la materia:
 - ▶ Syscalls
 - ▶ Scheduling
 - ▶ IPC (y threads)
 - ▶ Memoria
- Segunda parte de la materia:
 - ▶ Entrada / Salida (orga 1) y Repaso de diseño E/S
 - ▶ Drivers
- ¿Con qué están atrasados? 😊
 - ▶ Práctica de E/S
 - ▶ TP

Tabla de Contenidos

1 Intro

2 Drivers

3 Módulos

- ¿Qué es un módulo?
- Escribiendo nuestro primer módulo
- Compilando y ejecutando el módulo

4 Devices

- ¿Cómo se representan los devices?
- Creación de un device
- Acceder al device

5 Recomendaciones

¿Qué vamos a hacer hoy?

Hoy vamos a aprender a hacer un driver.

¿Qué vamos a hacer hoy?

Hoy vamos a aprender a hacer un driver.

- ¿Qué es un driver?

¿Qué vamos a hacer hoy?

Hoy vamos a aprender a hacer un driver.

- ¿Qué es un driver?
-

subsistema-es.pdf

¿Qué es un driver?

¿Qué es un driver?

- Un dispositivo de E/S va a tener, conceptualmente, dos partes:
 - ▶ El dispositivo físico.
 - ▶ Un controlador del dispositivo que interactúa con el SO mediante algún tipo de bus o registro.
- Los drivers son componentes de software muy específicos. Conocen las particularidades del HW contra el que hablan.

¿Los drivers son parte del SO? (1)

¿Los drivers son parte del SO? (1)

- Cuando enciendo la computadora, se carga el kernel del SO.

¿Los drivers son parte del SO? (1)

- Cuando enciendo la computadora, se carga el kernel del SO.
- Luego se cargan los distintos *módulos*, entre ellos, los drivers.

¿Los drivers son parte del SO? (1)

- Cuando enciendo la computadora, se carga el kernel del SO.
- Luego se cargan los distintos *módulos*, entre ellos, los drivers.
 - ▶ Administrador de memoria
 - ▶ Administrador de procesos (scheduler, etc.)
 - ▶ Sistema de archivos
 - ▶ Driver del teclado
 - ▶ Driver del mouse
 - ▶ Driver de video
 - ▶ Driver de ...

¿Los drivers son parte del SO? (1)

- Cuando enciendo la computadora, se carga el kernel del SO.
- Luego se cargan los distintos *módulos*, entre ellos, los drivers.
 - ▶ Administrador de memoria
 - ▶ Administrador de procesos (scheduler, etc.)
 - ▶ Sistema de archivos
 - ▶ Driver del teclado
 - ▶ Driver del mouse
 - ▶ Driver de video
 - ▶ Driver de ...



¿Los drivers son parte del SO? (2)

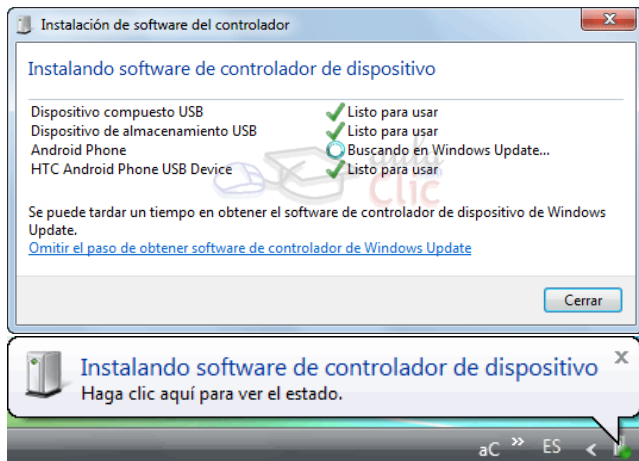
- Necesito un driver nuevo.

¿Los drivers son parte del SO? (2)

- Necesito un driver nuevo.
 - ▶ ¿Ya existía el código en el kernel?
 - ▶ ¿Tengo que reiniciar la máquina y cargar de nuevo el kernel?
 - ▶ ¿Tengo que recompilar el kernel?

¿Los drivers son parte del SO? (2)

- Necesito un driver nuevo.
 - ▶ ¿Ya existía el código en el kernel?
 - ▶ ¿Tengo que reiniciar la máquina y cargar de nuevo el kernel?
 - ▶ ¿Tengo que recompilar el kernel?



Solución

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

Solución

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

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.

Tabla de Contenidos

1 Intro

2 Drivers

3 Módulos

- ¿Qué es un módulo?
- Escribiendo nuestro primer módulo
- Compilando y ejecutando el módulo

4 Devices

- ¿Cómo se representan los devices?
- Creación de un device
- Acceder al device

5 Recomendaciones

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

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

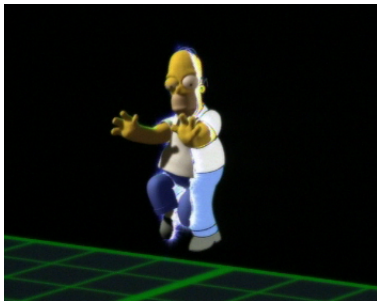
- ¿Qué cosas componen a un **módulo**?
 - ▶ Puntos de entrada y salida
 - ▶ Datos
 - ▶ Funciones
- ¿A causa de qué podría ejecutarse el código de un módulo?

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

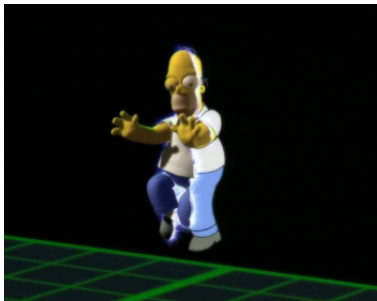
- ¿Qué cosas componen a un **módulo**?
 - ▶ Puntos de entrada y salida
 - ▶ Datos
 - ▶ Funciones
- ¿A causa de qué podría ejecutarse el código de un módulo?
 - 1 Llamada al sistema
 - 2 Atención de interrupción
- ¿Qué funcionalidades podría brindar un módulo?
- Hoy vamos a escribir nuestro primer módulo...

Un nuevo mundo...



- 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...



- 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**?

DANGER

Nuestro primer módulo (1)

```
#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()`
- `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("Juan de los Palotes");  
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)
- `__init` le indica al kernel que la función sólo se usará al momento de la inicialización, y que puede olvidarla 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
- 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
- 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

Compilando nuestro módulo

❶ Necesitamos (en la vm ya está instalado).

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

❷ 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
```

❸ ejecutar `make clean` y `make`

Ejecutando nuestro módulo

- Ya lo compilamos, ¿Cómo lo ejecutamos?
- `ls`
 - ▶ `hello.c hello.ko hello.mod.c hello.mod.o hello.o`
`Makefile modules.order Modules.symvers`

Ejecutando nuestro módulo

- Ya lo compilamos, ¿Cómo lo ejecutamos?
- `ls`
 - ▶ `hello.c hello.ko hello.mod.c hello.mod.o hello.o`
`Makefile modules.order Modules.symvers`
- Los módulos no los ejecutamos nosotros, se ejecutan:
 - 1 Al cargarlo en el sistema
 - 2 Cuando se ejecuta una llamada al sistema
 - 3 Cuando se atiende una rutina de atención de interrupción
 - 4 Al descargarlo del sistema

Ejecutando nuestro módulo

- Ya lo compilamos, ¿Cómo lo ejecutamos?
- `ls`
 - ▶ `hello.c hello.ko hello.mod.c hello.mod.o hello.o`
`Makefile modules.order Modules.symvers`
- Los módulos no los ejecutamos nosotros, se ejecutan:
 - 1 Al cargarlo en el sistema
 - 2 Cuando se ejecuta una llamada al sistema
 - 3 Cuando se atiende una rutina de atención de interrupción
 - 4 Al descargarlo del sistema
- Ok, carguemos el módulo en el sistema.

Cargando módulos al kernel (1)

¿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)
- `modprobe` es una alternativa más inteligente que `insmod` y `rmmmod` (tiene en cuenta dependencias entre módulos)
- `lsmod` lista los módulos cargados

Cargando módulos al kernel (2)

- `sudo insmod hello.ko`
- `lsmod | grep hello`
- `sudo rmmod hello`
- `lsmod | grep hello`

Cargando módulos al kernel (2)

- `sudo insmod hello.ko`
- `lsmod | grep hello`
- `sudo rmmod hello`
- `lsmod | grep hello`

Genial, ya sabemos escribir, compilar y cargar módulos pero ¿qué pasó con los drivers?

Tabla de Contenidos

1 Intro

2 Drivers

3 Módulos

- ¿Qué es un módulo?
- Escribiendo nuestro primer módulo
- Compilando y ejecutando el módulo

4 Devices

- ¿Cómo se representan los devices?
- Creación de un device
- Acceder al device

5 Recomendaciones

Tipos de *devices*

Dijimos que los drivers se comunican con los dispositivos. ¿De qué tipo pueden ser esos dispositivos? 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)

Devices y Drivers

Con `ls -l /dev` podemos ver los *drivers* del sistema.

```
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
- **minor**: identifica a un dispositivo específico que el driver maneja

Construcción de un *char device*

- ➊ Conseguir los *device numbers* (el *major* y el *minor*)
- ➋ Definir las funciones de cada operación del *device*
- ➌ Inicializar el *device* como un *char device*
- ➍ Registrar el *device* como un *char device*
- ➎ Crear un nodo en el filesystem para interactuar con el *device*

Construcción de un *char device*

- ➊ Conseguir los *device numbers* (el *major* y el *minor*)
- ➋ Definir las funciones de cada operación del *device*
- ➌ Inicializar el *device* como un *char device*
- ➍ Registrar el *device* como un *char device*
- ➎ Crear un nodo en el filesystem para interactuar con el *device*

¿En qué parte del módulo se hace todo esto?

(1) Conseguir los *device numbers*

¿Cómo reservamos los *device numbers* que necesitamos?

- Asignamos uno específico (puede ser problemático)
- Pedimos al kernel que nos asigne uno dinámicamente

Para reservarlos y liberarlos dinámicamente tenemos:

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor,
    unsigned int count, char *name);

void unregister_chrdev_region(dev_t first, unsigned int count);
```

Recibe:

- `dev_t *dev`: parámetro de salida (una estructura en donde se van a guardar los *device numbers*)
- `unsigned int firstminor`: primer *minor* a ser usado (0)
- `unsigned int count`: cantidad de *device numbers* contiguos (1)
- `char *name`: nombre del device asociado al rango

(2) Definir las operaciones (1)

```
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

(2) Definir 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

(3) Inicializar el *char device*

- el kernel representa internamente a los *char devices* mediante la estructura `struct cdev`

```
#include <linux/cdev.h>

struct cdev hello_dev;
```

- antes de que el kernel llame a nuestras operaciones, tenemos que inicializar y registrar al menos una de estas estructuras
- Llamar a la función `cdev-init` en el `init`

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

(4) Registrar el *char device* (1)

(4) Registrar el *char device* (2)

Ahora, registramos con:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Tener en cuenta que:

- `cdev_add()` puede fallar
- si no falló, las operaciones de nuestro módulo ya pueden ser llamadas

Para quitar al *char device* del sistema, usar:

```
void cdev_del(struct cdev *dev);
```

(4) Registrar el *char device* (3)

Módulo terminado

Listo, ya creamos el módulo, definimos sus funciones y lo registramos como un driver de dispositivo.

Módulo terminado

Listo, ya creamos el módulo, definimos sus funciones y lo registramos como un driver de dispositivo.

Ahora ¿cómo probamos el código?

Módulo terminado

Listo, ya creamos el módulo, definimos sus funciones y lo registramos como un driver de dispositivo.

Ahora ¿cómo probamos el código?

Necesitamos una interfaz para que los programas de usuario puedan acceder al código.

(5) Crear nodos (1)

Tenemos, a priori, dos opciones:

- Crear los nodos, una vez se haya insertado el módulo, usando `mknod <nodo> c <major> <minor>` ,
- Que desde el módulo se genere algún tipo de aviso a alguien, en espacio de usuario, que se encargue de crear el nodo

Para lo segundo:

```
#include <linux/device.h>

static struct class *hello_class;

hello_class = class_create(THIS_MODULE, "hello");
device_create(hello_class, NULL, hello_devno, NULL, "hello");

device_destroy(hello_class, hello_devno);
class_destroy(hello_class);
```

(5) Crear nodos (2)

- La próxima vez que carguemos nuestro módulo, se generará un nuevo dispositivo virtual en `/dev/`.
- Podemos verlo con `ls /dev/ | grep hello`
 - ▶ El nombre con el que figura es el que le dieron al llamar a `device_create`, podría ser distinto al nombre del módulo.
- Para leer el dispositivo:
 - ▶ `sudo head -n 1 /dev/hello`
- Para escribir el dispositivo:
 - ▶ `echo -n "1" | sudo tee /dev/hello`

Tabla de Contenidos

1 Intro

2 Drivers

3 Módulos

- ¿Qué es un módulo?
- Escribiendo nuestro primer módulo
- Compilando y ejecutando el módulo

4 Devices

- ¿Cómo se representan los devices?
- Creación de un device
- Acceder al device

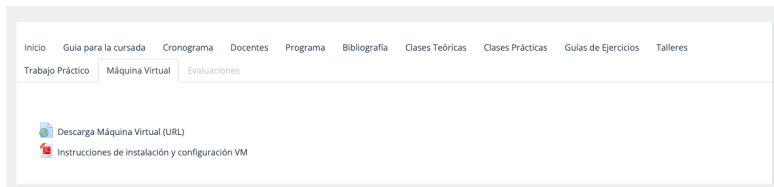
5 Recomendaciones

Antes de empezar el taller...

- Descargar y configurar la VM de la materia.
- Descargar instrucciones desde el campus.

Antes de empezar el taller...

- Descargar y configurar la VM de la materia.
- Descargar instrucciones desde el campus.



Memoria de Usuario vs Memoria de Kernel

- 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:

```
#include <linux/uaccess.h>

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);
```

Memoria dinámica (kernel)

Existen diversas formas de pedir memoria al sistema cuando estamos ejecutando en modo kernel. Nosotros vamos a ver dos:

kmalloc

Funciones: `void * kmalloc(size_t size, int flags), kfree(void * ptr)`. Solicita un espacio de memoria **físicamente contiguo** de `size` bytes. Devuelve un puntero *virtual* accesible sólo en modo kernel.

vmalloc

Funciones: `void * vmalloc(size_t size), vfree(void * ptr)`. Solicita un espacio de memoria **virtualmente contiguo** de `size` bytes. Devuelve un puntero *virtual* accesible sólo en modo kernel.

Para usarlas, incluir `linux/slab.h`.

Memoria dinámica (kernel)

Existen diversas formas de pedir memoria al sistema cuando estamos ejecutando en modo kernel. Nosotros vamos a ver dos:

kmalloc

Funciones: `void * kmalloc(size_t size, int flags), kfree(void * ptr)`. Solicita un espacio de memoria **físicamente contiguo** de `size` bytes. Devuelve un puntero *virtual* accesible sólo en modo kernel.

Ejemplo de `kmalloc`:

```
#include <linux/slab.h>

// pido 9 bytes
unsigned char* buffer = kmalloc(9, GFP_KERNEL);
```

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)`

spinlock

Tipo de datos: `spinlock_t` Funciones: `spin_lock_init(spinlock_t * lock)`, `spin_lock(spinlock_t * lock)`, `spin_unlock(spinlock_t * lock)`, etc.

- Device drivers: <https://www.kernel.org/doc/html/v4.19/driver-api/index.html>
- Kernel locking: <https://www.kernel.org/doc/html/v4.19/kernel-hacking/locking.html>
- Libro: <https://lwn.net/Kernel/LDD3/>