Desarrollo de módulos de kernel

Mg. Ing. Pablo Slavkin
Mg. Ing. Hanes N. Sciarrone
MSE - 2024

Implementación de Manejadores de Dispositivos

Módulos de kernel

- Utilización de módulos de kernel
- Particularidades del kernel
- Desarrollo de módulos de kernel

Utilización de módulos de kernel

Utilización de módulos de kernel

- La utilización de módulos de kernel tiene algunas ventajas:
  - El desarrollo y modificación se pueden hacer sin un reboot (load, test, unload, rebuild, load…).
  - Mantienen el tamaño de la imagen de kernel en un mínimo.
  - Reduce el tiempo de booteo.
- Al cargar un módulo se deben tener en cuenta:
  - Una vez cargado el módulo, tiene privilegios nivel kernel (solo el usuario ROOT puede cargar módulos).
  - Existe la posibilidad de solo permitir módulos firmados para incrementar la seguridad.

Utilización de módulos de kernel

- Recordar que algunos módulos tienen dependencia de otros.
- Ejemplo: el módulo *ubifs* depende de *ubi* y *mts*. (link wiki)
- Las dependencias se pueden ver en **/lib/modules/<kernel-version>/modules.dep**
- Este archivo junto con **modules.bin.dep** se generan al ejecutar ***make module_install***
- NOTA: Recordar que esto no se puede hacer directamente en nuestro caso (sistema embebido).

Utilización de módulos de kernel

- Al cargar un módulo, el kernel log se carga con información relacionada.
- Los mensajes de kernel se almacenan en un buffer circular.
- Disponibles a través del comando **dmesg**.
- Mensajes del kernel log también se muestran por medio de la consola de sistema.
- Se filtran según el valor que tenga la variable **loglevel**.

```
console=ttyS0 root=/dev/mmcblk0p2 loglevel=5
```

Utilización de módulos de kernel

- Existen varios comandos útiles a la hora de trabajar con un módulo de kernel.
- Ejemplo: sea un módulo **mi_modulo.ko**
  - **modinfo** lista la información del módulo.
  - **insmod** trata de cargar el módulo. Se debe proporcionar el path completo.

```
$ modinfo <module_path>/mi_modulo.ko
$ modinfo mi_modulo   //(para modulos en /lib/modules)
$ sudo insmod <module_path>/mi_modulo.ko
```

Utilización de módulos de kernel

- Cuando **insmod** falla, no da muchos detalles.
- Es necesario ver el kernel log para estos detalles.
- Ejemplo:

```
$ sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1 Device or resource busy

$ dmesg
[17549774.552000] Failed to register handler for irq channel 2
```

Utilización de módulos de kernel

- Otro comando útil es **modprobe**.
- Trata de cargar todos los módulos de los que depende el módulo pasado como argumento.
- Luego de cargar las dependencias, carga el módulo indicado.

```
$ sudo modprobe mi_modulo.ko
```

Utilización de módulos de kernel

- Comando **lsmod** lista todos los módulos cargados.
- Idéntico a ejecutar **cat /proc/modules**, pero en formato más legible.

```
$ lsmod
```

- Este comando puede ser ejecutado en cualquier directorio.
- No necesita permisos de superusuario (solo lista información).

Utilización de módulos de kernel

- Para remover un módulo se utiliza **rmmod**.
- Trata de remover el módulo indicado.
- Solo permitido si el modulo no esta en uso.

```
$ sudo rmmod mi_modulo
```

- Lo mismo puede lograrse con **modprobe**.
- modprobe también remueve dependencias cargadas.

```
$ sudo modprobe -r mi_modulo
```

Utilización de módulos de kernel

- Pueden pasarse argumentos a los módulos al momento de cargarlos.
- Para ver que parámetros tiene un módulo disponible se utiliza **modinfo**.
- Ejemplo:

```
$ modinfo usb-storage
```

- Uno de los parámetros es **delay_use**.

Utilización de módulos de kernel

- Este parámetro puede ser pasado como argumento de distintas formas:
  - Mediante **insmod**.
  - Mediante **modprobe**.
  - Mediante la línea de comandos del kernel, si el driver se compila estáticamente junto con el kernel.
- Caso **insmod**:

```
$ sudo insmod ./usb-storage.ko delay_use=0
```

Utilización de módulos de kernel

- Caso **modprobe**:
  - Seteo de parámetros en archivo **/etc/modprobe.conf**
  - Seteo de parámetros en cualquier archivo dentro de **/etc/modprobe.d/**

```
options usb-storage delay_use=0
```

- Caso línea de comandos del kernel:

```
usb-storage.delay_use=0
```

Utilización de módulos de kernel

- Para el caso que se desee encontrar y cambiar parámetros de módulos ya cargados:
  - Checkear /sys/module/<name>/parameters.
  - Hay un archivo por parámetro, que contiene el valor del mismo.
  - Si el archivo tiene permisos de escritura puede cambiarse este valor.

```
$ echo 0 > /sys/module/usb_storage/parameters/delay_use
```

Particularidades del kernel

Particularidades del kernel

- A través de los años, los desarrolladores propusieron que el kernel de linux siguiera reglas estándar.

- Citamos las más importantes:
  - Estilo de programación.
  - Asignación e inicialización de memoria para estructuras de kernel.
  - Clases, objetos y OOP (programación orientada a objetos).

Particularidades del kernel

- El Estilo de programación determina si nuestro código se evalúa para ingresar al kernel o no.
- Hay muchas reglas, solo citaremos las más populares.
- Indentación
  - Se utilizan tabs de 8 caracteres y líneas de 80 columnas.
  - Si la indentación evita que se escriba la función completa, es porque tiene muchos niveles de anidamiento.
  - Para checkear esto se utiliza un script dentro de las fuentes del kernel

```
$ ./scripts/cleanfile my_module.c
```

Particularidades del kernel

- Funciones y variables
  - Cualquiera que no sea exportada, debe ser declarada como static.
  - No se aceptan espacios alrededor de expresiones dentro de paréntesis.
  - Son aceptados espacios antes o después de los paréntesis

```
s = sizeof (struct file);   //aceptado
s = sizeof( struct file );  //No aceptado
```

  - La utilización de **typedef** está prohibida.

Particularidades del kernel

- Funciones y variables (cont.)
  - Los comentarios debe ser de la forma /* */
  - Comentarios en línea con doble barra no son aceptados.

```
/* A Desarrolladores de kernel les gusta esto */
// Quien utiliza esto no es digno del Olimpo
```

  - Las macros deben capitalizarse (MAYÚSCULAS). Macros funcionales pueden estar en minúsculas.
  - Un comentario jamás debe reemplazar código legible. Se prefiere re-escribir código que agregar comentarios.

Particularidades del kernel

- La asignación e inicialización de memoria para estructuras de kernel puede hacerse de dos formas: estática y dinámica.
- Todos los inicializadores dinámicos se declaran como macros.
- Existe una diferencia fundamental en el ámbito (scope) de cada objeto:
  - Objetos estáticos tienen un alcance sobre todo el driver. Todos los dispositivos manejados por este driver ven el objeto.
  - Objetos dinámicos solo son visibles por el dispositivo que está utilizando una instancia determinada del driver.

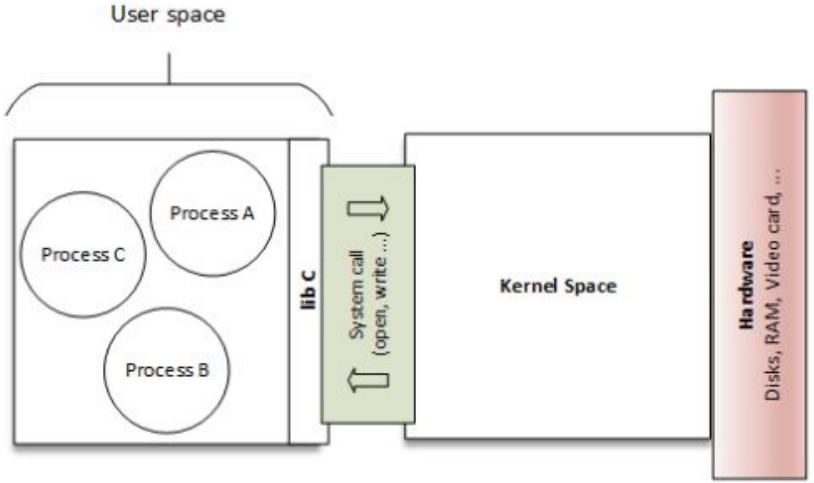
Particularidades del kernel

- Con respecto a la OOP, el kernel implementa esto mediante **dispositivos** y **clases**.
- Los subsistemas del kernel se abstraen como **clases**.
- Hay aproximadamente la misma cantidad de subsistemas que directorios en **/sys/class/**.
- Cada dispositivo que cae dentro de un subsistema tiene un puntero a una estructura de operaciones.
- Esta estructura expone las operaciones que puede efectuar el dispositivo.

Desarrollo de módulos de kernel

Desarrollo de módulos de kernel

- Lo primero a saber en el desarrollo de módulos de kernel es cómo interactúa el espacio usuario con el espacio kernel.
- Tiene que ver con sectores de memoria que se pueden acceder (o no) dependiendo de los derechos de acceso (privilegios).



Desarrollo de módulos de kernel

- Espacio de kernel:
  - Conjunto de direcciones donde se almacena y corre el kernel.
  - Protegido por flags de acceso, previniendo que el usuario lo corrompa.
  - El kernel por lo tanto accede a cualquier posición de memoria, por su nivel de privilegio.
- Espacio usuario:
  - Conjunto de direcciones donde corren los programas normales.
  - Menor privilegio, no puede acceder a memoria de otra aplicacion (sandbox)

Desarrollo de módulos de kernel

- La única manera de que algun codigo de espacio usuario se ejecute a nivel de kernel es mediante system calls.
- System calls: read, write, open, close, ……., etc.
- Cuando un proceso efectúa un system call, se envía una interrupción de software al kernel.
- En la interrupción, se le dan privilegios al código hasta tanto se vuelva de esa interrupción.

Desarrollo de módulos de kernel

- Comencemos identificando los puntos de entrada y salida.
- Todos los módulos de kernel tienen **entry** y **exit** points.
- Son las funciones llamadas con **insmod** y **rmmod** respectivamente.
- Puede ser confuso, pero los módulos no siguen las reglas que conocemos de un programa estándar en C.
- El entry point no necesita ser *main*(), puede tener cualquier nombre.

Desarrollo de módulos de kernel

- Asimismo, en un programa en C, el programa retorna al final de la función *main*().
- En los módulos se define el punto de salida en otra sección (otra función).
- Solo se necesita indicar al kernel cual es la función **entry** y cual la función **exit**.
- Esto se hace mediante las macros **module_init**() y **module_exit**().

Desarrollo de módulos de kernel

- Las macros **module_init**() y **module_exit**() indican las funciones que se ejecutan cuando se carga y descarga el módulo.

- Solo se ejecutan estas funciones una única vez, al momento de efectuar la carga/descarga del módulo.

```
module_init(helloworld_init);
module_exit(helloworld_exit);
```

Desarrollo de módulos de kernel

- Además de las mencionadas macros, se deben indicar dos atributos en la definición de estas funciones.
  - **__init**
  - **__exit**
- En realidad estos atributos son macros definidos en **include/linux/init.h**

```
#define __init __section(.init.text)
#define __exit __section(.exit.text)
```

Desarrollo de módulos de kernel

- Se observa que estas macros indican en qué sección deben ser cargadas estas funciones.
- Estas secciones son conocidas de antemano por el kernel.
- En el caso de **__init** para drivers compilados con el kernel, la memoria asociada se libera luego de inicializar el módulo.
- Esto se da porque al ser built-in, no puede ser desmontado.
- La función init solo se ejecutara en el próximo boot.

Desarrollo de módulos de kernel

- En el ejemplo citado, la sección **__exit** se omite, porque no puede ser desmontado.
- También se omite cuando el kernel se compila sin soporte para desmontar módulos.
- En el caso de módulos que pueden ser cargados, **__exit** no tiene efecto alguno.
- Se pueden observar las secciones de un módulo en un archivo ELF:

```
$ objdump -h module.ko
```

Desarrollo de módulos de kernel

```
static int __init helloworld_init(void) {
    pr_info("Hola Mundo!\n");
    return 0;
}

static void __exit helloworld_exit(void) {
    pr_info("Fin del mundo\n");
}
```

Desarrollo de módulos de kernel

- Sin leer el código del módulo, se debería poder obtener información básica (autor, descripción de parámetros, licencia).
- Los módulos de kernel utilizan su sección **.modinfo** para almacenar esta información.
- La información en un módulo se indica mediante macros:
  - MODULE_DESCRIPTION().
  - MODULE_AUTHOR().
  - MODULE_LICENSE().

Desarrollo de módulos de kernel

- Estas macros en realidad envuelven la única macro que escribe información en el módulo: **MODULE_INFO**(*tag*, *info*).
- Esto quiere decir que además de los campos mencionados, también podemos agregar campos personalizados.

```
MODULE_INFO(mse_imd, "Esto no es para simples mortales");
```

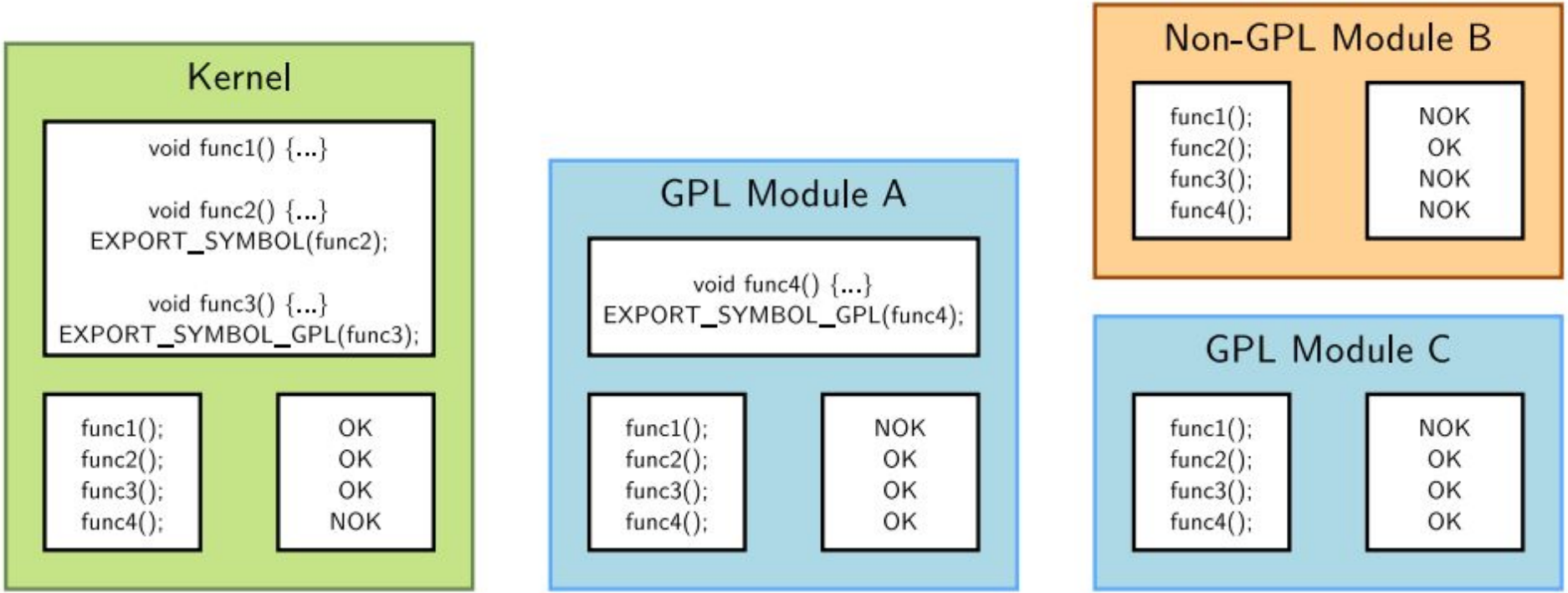
- Estos campos son los que muestra el comando **modinfo**, devolviendo el contenido de la sección **.modinfo**

Desarrollo de módulos de kernel

- Algo muy importante en los módulos es la licencia por la cual se comparte (o no).
- El macro MODULE_LICENSE() da esta información al kernel, y tiene un efecto sobre cómo se comporta el módulo.
- Una licencia no compatible con GPL provoca que el módulo no pueda ver/utilizar algunos servicios y funciones.

```
MODULE_LICENSE ("GPL");
```

Desarrollo de módulos de kernel



Desarrollo de módulos de kernel

- Cargar un módulo no GPL da como resultado un tainted kernel.
- Esto significa que el kernel cargo código que no es open source o que no procede de una fuente confiable.
- Implica no tener soporte de la comunidad.
- Al omitir la definición de MODULE_LICENSE() se asume que el código no es open source.

Desarrollo de módulos de kernel

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int __init helloworld_init(void) {
    pr_info("Hola Mundo!\n");
    return 0;
}
static void __exit helloworld_exit(void) {
    pr_info("Fin del mundo\n");
}
module_init(helloworld_init);
module_exit(helloworld_exit);
MODULE_AUTHOR("Cosme Fulanito <cosme.fulanito@gmail.com>");
MODULE_LICENSE("GPL");
```

Desarrollo de módulos de kernel

- En el código de ejemplo se utiliza una macro que imprime mensajes en consola: **pr_info**.
- Esta junto con otras macros son wrappers de la función **printk**().
- La función **printk**() es para el espacio kernel lo que la función **printf**() es para el espacio usuario.
- **printk**() siempre necesita un log level como parámetro.
- Estos niveles se definen en **include/linux/kern_levels.h**

Desarrollo de módulos de kernel

- Para módulos nuevos se recomienda la utilización de las macros wrappers:
  - pr_info()
  - pr_notice()
  - pr_warning()
  - pr_error()
  - pr_debug()

Desarrollo de módulos de kernel

- A este punto con el esqueleto básico definido, podemos compilar nuestro primer módulo.
- Hay dos posibilidades: dentro del kernel tree y out-of-tree
- **out-of-tree**: El código está fuera del source tree del kernel, en un directorio diferente.
  - Ventaja: puede resultar más fácil las modificaciones.
  - Desventajas: Al no estar integrado con las configuraciones de compilación del kernel, hay que compilarlo por separado.
  - Al ser out-of-tree no puede ser compilado de forma estática.

Desarrollo de módulos de kernel

- **Dentro del kernel tree**: El código está dentro del source tree del kernel.
  - Ventajas: Bien integrado con el proceso de configuración y compilación del kernel.
  - Puede ser compilado de forma estática si se desea.
- Comenzaremos con una compilación out-of-tree.
- Se necesita un makefile para poder compilar el archivo source.

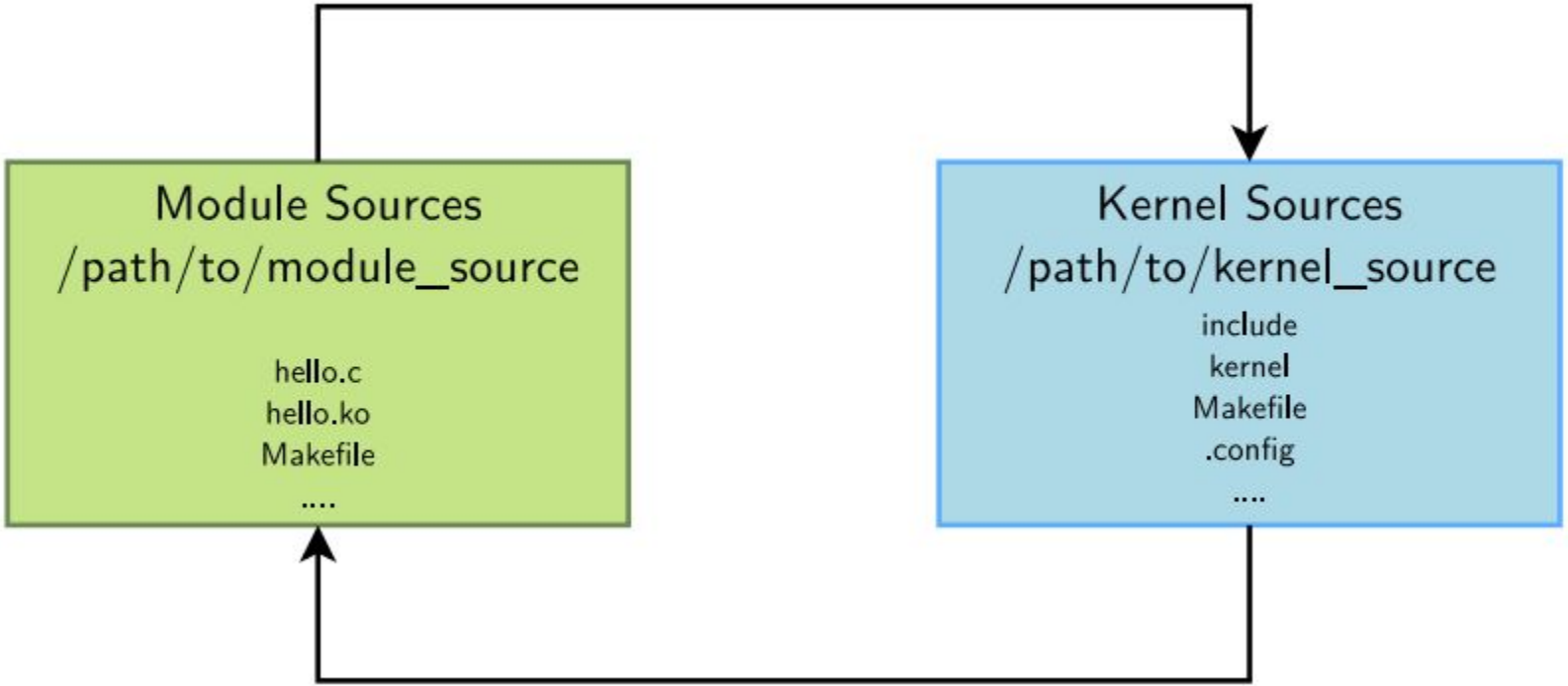
Desarrollo de módulos de kernel

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources
all:
    $(MAKE) -C $(KDIR) M=$$PWD
endif
```

Desarrollo de módulos de kernel

- Este Makefile mínimo es suficiente para compilar cualquier módulo compuesto por un solo archivo fuente.
- En el caso out-of-tree es interpretado con la macro **KERNELRELEASE** sin definir.
- Esto provoca una llamada al Makefile del kernel.
- El directorio donde esta el modulo se pasa en la variable **M**.
- El Makefile del kernel ahora tiene visibilidad del directorio del módulo, y llama al Makefile nuevamente.

Desarrollo de módulos de kernel



Desarrollo de módulos de kernel

- En esta nueva llamada, el Makefile del módulo es invocado con la macro **KERNELRELEASE** definida.
- Se tiene acceso a la definición de **obj-m**.
- En casi todos los Makefiles del kernel se pueden ver instancias del patrón **obj-<X>**.
- **<X>** puede tomar los valores **m**, **y**, **n**, o ser dejado en blanco.
- Estas opciones se utilizan para indicar al kernel opciones de compilación.

Desarrollo de módulos de kernel

- Para clarificar, citamos distintos casos:
  - Si **<X> = m**, se utiliza la variable **obj-m** y el archivo objeto **mi_modulo.o** se compila como un módulo.
  - Si **<X> = y**, se utiliza la variable **obj-y** y el archivo objeto **mi_modulo.o** se compila como parte del kernel (built-in).
  - Si **<X> = n**, el archivo objeto no es compilado.
- Por lo tanto, se utiliza frecuentemente el patrón **obj-$(CONFIG_XXX)**.
- **CONFIG_XXX** es una opción de configuración de kernel.

Desarrollo de módulos de kernel

- Para poder compilar un módulo de kernel, se necesita acceso a los headers del kernel.
- Se puede lograr de dos maneras:
  - Teniendo el kernel completo (caso de esta materia).
  - Solo descargar los headers.
- **IMPORTANTE:** Módulos de kernel compilados con headers de una versión **X** no podrán ser cargados en un kernel version **Y**.
- **modprobe** e **insmod** retornan *Invalid module format*.

Desarrollo de módulos de kernel

- Para compilar in-tree, se debe identificar en qué directorio dentro de **/drivers** debe estar contenido el módulo.
- En este caso pondremos nuestro archivo **.c** en el directorio **/drivers/misc**
- Cada subdirectorio en **/drivers** tiene un archivo **Makefile** y un **Kconfig**.
- Para que la configuración pueda verse con menuconfig, se debe modificar el archivo **Kconfig**.

Desarrollo de módulos de kernel

- Ejemplo: compilar un módulo **hello_world.c**
- Agregar al archivo **/drivers/misc/Kconfig** lo siguiente:

```
config MSE_HELLOWORLD
    tristate "Modulo hola mundo para IMD - MSE"
    default m
    help
        Utilice la opcion Y para compilar de manera built-in.
        Por defecto se compila como modulo in-tree
```

Desarrollo de módulos de kernel

- Luego de modificar **Kconfig**, se debe modificar el **Makefile**
- Agregar al archivo **/drivers/misc/Makefile** lo siguiente:

```
obj-$(CONFIG_MSE_HELLOWORLD) += hello_world.o
```

- Nuevamente verificar que la ruta del archivo fuente sea **/drivers/misc/hello_world.c**
- Verificar funcionamiento ejecutando **make menuconfig**.
- En el caso de compilar con opción **m**, es necesario ejecutar **make modules**.

Desarrollo de módulos de kernel

- Una parte importante del desarrollo de software es el manejo y presentación de errores.
- No es tan importante en el desarrollo de espacio kernel, pero la propagación de un error incorrecto puede generar problemas.
- Los códigos de error son útiles para hacer print de ellos a la hora de hacer debug.
- El kernel provee algunos códigos de error que cubren la mayoría de los errores que pueden ocurrir.

Desarrollo de módulos de kernel

- Los mencionados códigos de error pueden encontrarse en *include/uapi/asmgeneric/errno-base.h*
- En el mismo archivo pueden verse sus significados.
- El resto de los códigos puede encontrarse en *include/uapi/asmgeneric/errno.h*.
- La forma clásica de comunicar un error es mediante el código de error precedido de un signo menos.
- Esto se aplica mayormente en respuesta a system calls

Desarrollo de módulos de kernel

```
dev = init(&ptr);
if(!dev)
    return -EIO;
```

- Este ejemplo corresponde a un error de I/O.

Desarrollo de módulos de kernel

- Dependiendo la situación los errores pueden propagarse al espacio usuario.
- En el caso de una llamada de syscall (open, read, ioctl, mmap) el valor se asigna automáticamente a la variable **errno**.
- **errno** es una variable global de espacio usuario.
- En estos casos se puede utilizar **strerror(errno)** para transformar el error a texto legible.
- NOTA: Recuerde que **errno** y **strerror()** son de espacio usuario.

Desarrollo de módulos de kernel

- En la ocurrencia de un error, se debe deshacer todo lo que se hizo antes del mismo.

- Aplica para cuando se asigna memoria dinámica.

- Es común utilizar la instrucción **goto** para estos casos.

Desarrollo de módulos de kernel

```
ptr = kmalloc(sizeof (device_t));
if(!ptr) {
    ret = -ENOMEM;
    goto err_alloc;
}
dev = init(&ptr);
if(!dev) {
    ret = -EIO
    goto err_init;
}
return 0;
```

```
err_init:
    free(ptr);
err_alloc:
    return ret;
```

Desarrollo de módulos de kernel

- Se utiliza goto al gestionar errores porque se debe hacer en el orden inverso de las operaciones previas.

- Podría usarse estructuras if anidadas, pero tiende a ser confuso y a llevar a errores de indentación.

- La utilización de goto hace que el código sea legible y tener un control de flujo directo (sin saltos FWD-BKWD).

Desarrollo de módulos de kernel

```
if (ops1() == ERR) // |
    goto error1;   // |
if (ops2() == ERR) // |
    goto error2;   // |
if (ops3() == ERR) // |
    goto error3;   // |
if (ops4() == ERR) // V
    goto error4;
```

```
[...]
error4:
    [...]
error3:
    [...]
error2:
    [...]
error1:
    [...]
```

Desarrollo de módulos de kernel

- En el caso que se tengan funciones que retornan punteros, en caso de error retornan NULL.
- Es un mecanismo simple y efectivo, pero no da información de porque se dio el error.
- Para esto el kernel provee tres funciones:
  - void *ERR_PTR(long error);
  - long IS_ERR(const void *ptr);
  - long PTR_ERR(const void *ptr);

Desarrollo de módulos de kernel

- **void *ERR_PTR(long error)**
  - Devuelve el error como un puntero. Generalmente utilizada para funciones que retornan ENOMEM.
  - Ejemplo: return ERR_PTR(-ENOMEM);
- **long IS_ERR(const void *ptr)**
  - Chequea si el valor de puntero corresponde a un error.
- **long PTR_ERR(const void *ptr)**
  - Retorna el error que fue devuelto como puntero.

Desarrollo de módulos de kernel

```
static struct iio_dev *indiodev_setup(){
     [...]
     struct iio_dev *indio_dev;
     indio_dev = devm_iio_device_alloc(&data->client->dev, sizeof(data));
     if (!indio_dev)
        return ERR_PTR(-ENOMEM);
     [...]
     return indio_dev;
    }
```

Desarrollo de módulos de kernel

```
static int foo_probe([...]){
      [...]
      struct iio_dev *my_indio_dev = indiodev_setup();
      if (IS_ERR(my_indio_dev))
         return PTR_ERR(data->acc_indio_dev);
      [...]
}
```

Desarrollo de módulos de kernel

- Así como un programa espacio usuario, los módulos pueden recibir parámetros al ser cargados.
- Útil a la hora de desarrollar para evitar ciclos de corrección/compilación/carga/verificación.
- Para definir parámetros de módulos, primero se deben instanciar las variables que van a recibir los valores.
- Luego utilizar en cada una de ellas la macro **module_param()** definida en *include/linux/moduleparam.h*

Desarrollo de módulos de kernel

- module_param(name, type, perm);
  - **name**: el nombre de la variable a utilizar.
  - **type:** El tipo de datos del parametro. *No son los mismos que en C*. (bool, charp, byte, short, ushort, int, uint, long, ulong).
  - **perm:** representa los permisos del archivo /sys/module/<module>/parameters/<param>.
    Puede ser S_IWUSR, S_IRUSR, S_IXUSR, S_IRGRP, S_WGRP, S_IRUGO
  - **S_I** es solo un prefijo.
  - **R:** read, **W:** write, **X:** execute.
  - **USR:** user, **GRP:** group, **UGO:** user, group, others.

Desarrollo de módulos de kernel

- Se puede utilizar el operador OR (|) para obtener permisos específicos.
- Al utilizar parámetros en los módulos, es buena práctica emplear la macro **MODULE_PARM_DESC**.
- Esta macro asigna una descripción a cada parámetro.

```
MODULE_PARM_DESC(myint,"Esto es un int innecesario");
```

Desarrollo de módulos de kernel

HANDS ON

1. Escribir un modulo “hola mundo”.
2. Compilar out of tree, testear el módulo.
3. Compilar in-tree de manera built-in.
4. Guía de práctica I

Gracias.