Programación de Módulos en Linux

En este documento intentaremos aprender acerca de los módulos del kernel de Linux y como poder programarlos. Para ello trataremos de resumir y explicar que es el núcleo y como está formado, seguiremos con los módulos y para finalizar hablaremos específicamente de los módulos controladores de dispositivos.

Tabla de contenido

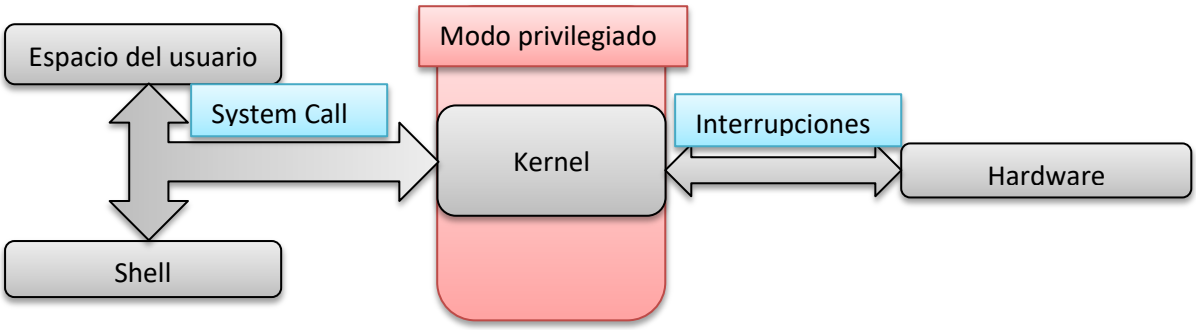
Núcleo o Kernel

Definición

El Kernel es un programa informático o software que forma parte de cualquier Sistema Operativo

Utilidad

El Kernel es el encargado de que usuario y máquina se comuniquen sin que el primero tenga que hablar el mismo lenguaje que el segundo.



Es decir, el Kernel se encarga de acceder y gestionar los recursos del ordenador como la CPU, dispositivos E/S y otros recursos. También es el encargado de gestionar la memoria y los dispositivos.

Por tanto hay procesos que solo pueden ser ejecutados por el kernel (modo Kernel o privilegiado) como la gestión de memoria para un proceso y otras puede hacerlas directamente el usuario como dar distintos procesos a realizar a la CPU.

Tipos

Kernel Monolíticos

El Kernel engloba todos los servicios del S.O. No es modular, por lo que al implementar cualquier cambio requiere la recompilación del núcleo y reinicio del sistema.

Micro-núcleos

Núcleos de pequeño tamaño que abarcan las necesidades más básicas del sistema. Para añadir más funcionalidades se han de añadir módulos externos al núcleo.

Núcleos Híbridos

Arquitectura basada en la "fusión" de núcleos monolíticos y micro-núcleos. Por lo que los controladores de los dispositivos y las extensiones se pueden cargar y descargar fácilmente como módulos mientras el sistema sigue funcionando.

Exonúcleos

Toda la funcionalidad del sistema deja de estar en memoria para encontrarse en librerías dinámicas

Aunque hay más tipos de núcleos estos son los principales. El núcleo de Linux es híbrido, como la mayoría en todos los S.O.

Módulos del Núcleo

Los módulos son fragmentos de código que pueden vincularse dinámicamente al núcleo después de que el sistema haya arrancado. También pueden desvincularse y eliminarse cuando ya no sean necesarios.

Si nos centramos en los núcleos de Linux, estos suelen ser controladores de dispositivos, pseudocontroladores de dispositivos (Ej. Controladores de red) o sistemas de archivos.

Los módulos son útiles para probar nuevo código del Kernel sin tener que reconstruir y reiniciar el núcleo. Por otro lado estos implican más código y estructuras de datos adicionales por lo que penaliza memoria y rendimiento.

Para cargar o descargar módulos en el núcleo de Linux se pueden utilizar los comandos *insmod* y *rmmod* o el propio Kernel puede cargar o descargar los que necesite. Una vez cargado un nuevo módulo, este tiene los mismos derechos que cualquier otro código del núcleo, por lo que puede llegar a bloquear el núcleo o los controladores de dispositivos.

Ahora que sabemos que es un módulo, profundicemos en su inserción y eliminación en el núcleo.

Cargar un Módulo

Como comentamos anteriormente hay dos formas de insertar un módulo. Primero nos centraremos en la forma “manual” de hacerlo, es decir, con un comando. En esta ocasión nos centraremos en *insmod,* aunque hay otros comandos parecidos como *modprobe*.

Si nos vamos a su manual, podremos ver que toma como argumentos el nombre del archivo de un módulo y una opción especifica del módulo. Tras realizar su llamada, el comando intentará conectar el módulo con el núcleo en ejecución resolviendo los símbolos de la tabla de símbolos exportados del núcleo.

Profundicemos más en esto último. Tras realizar la llamada, *insmod* llama a *init_module()* para informar al Kernel de que se intenta cargar un módulo. Ahora el Kernel toma el control y ejecuta *sys_init_module().* Después de verificar que el usuario tiene permisos para cargar un módulo, se llama a la función *load_module* la cual asigna memoria temporal y copia el módulo ELF (Formato ejecutable y vinculable) del espacio del usuario a la memoria del Kernel. Tras comprobar si el módulo es un archivo ELF apropiado, se genera un offset en el espacio de memoria temporal asignado. El estado de modulo se actualiza a *MODULE_STATE_COMING* y se asigna la ubicación en el núcleo utilizando *SHF_ALLOC*. Se realiza la resolución de símbolos y *load_module* devuelve una referencia al módulo del núcleo. Esta referencia se inserta en una tabla que contiene todos los módulos cargados del sistema. Para finalizar se llama a la función *module_init()* en el código del módulo y este actualiza su estado a *MODULE_STATE_LIVE*.

Ahora nos centraremos en la parte “automática”. Esta manera es más “eficiente”, pues solo se cargan los módulos cuando son necesarios, es decir, bajo demanda. Cuando el Kernel necesita un módulo, este solicitará que el *daemon* (término antiguo para referirse a un hilo o proceso

en segundo plano que hace algo útil) del núcleo intente cargar el módulo adecuado. El *daemon* del núcleo es un proceso normal, pero con privilegios de superusuario. Su función principal es cargar y descargar módulos del Kernel aunque también puede realizar otras tareas.

Descargar un Módulo

Saber cómo cargar un módulo implica, casi de manera obligada, tener que saber cómo descargarlo, ya que nadie querría saber arrancar un coche sin saber cómo apagarlo posteriormente. Al igual que para cargar, para eliminar módulos hay dos formas de hacerlo. Para borrar módulos mediante comandos podemos utilizar *rmmod*.

Al igual que insmod, rmmod toma como argumentos el nombre del archivo y una opción específica.

Si analizamos detenidamente cómo elimina un módulo, vemos que realiza pasos muy similares (con instrucciones "inversas") que insmod. Es decir, *rmmod* llama a *delete_module*(), la cual indica al Kernel que ha llegado una petición para borrar un módulo. Tras delegar el control al Kernel, este ejecuta *sys_delete_module().* Después de comprobar si el usuario tiene permisos para descargar un módulo, se verifica si hay algún otro módulo cargado que dependa del módulo que se va a eliminar (modules_which_uses_me). Mediante *MODULE_STATE_LIVE*, se constata que el módulo esté realmente cargado en el Kernel y se llama a *module_exit()* escrita en el código del módulo. Posteriormente se ejecuta *free_module*, la cual elimina las referencias del módulo, ya sean *sysfs* o a objetos del propio módulo. También realiza, si es necesario, una limpieza de la arquitectura. Y por último, descarga el módulo del núcleo, actualiza el estado de este a *MODULE_STATE_GOING* y libera la memoria utilizada por los argumentos del espacio de usuario.

Los módulos que son cargados bajo demanda, son eliminados automáticamente del sistema cuando ya no se utilizan. Cada vez que un temporizador de inactividad, establecido al iniciar el Kernel, expira, el *deamon* del Kernel realiza una llamada para eliminar todos los módulos cargados que no se utilicen. Los módulos pueden ser eliminados solo si ningún otro en uso depende de ellos.

Ejemplos

Ahora que ya sabemos algo de como insertar y eliminar módulos, veamos algunos ejemplos.

"Hello, world"

Al igual que en el mundo de la programación, el primer programa que todos hacemos es el "Hello Word", en el mundo de los módulos, el primero que haremos será el módulo "Hello Word" aunque lo llamaré "hello" para abreviar.

```
/*
 * hello.c - Módulo de ejemplo
 */
#include <linux/module.h
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_start(void){
        printk(KERN_INFO "Hello, world\n");
        return 0;
```

```
}
static void __exit hello_end(void){
        printk(KERN_INFO "Goodbye, world\n");
}
module_init(hello_start);
module_exit(hello_end);
```

Este es todo el código que constituye al módulo así que expliquémoslo por partes antes de aprender a compilarlo. Cabe destacar que el módulo carece de la documentación, en la cual nos centraremos más adelante, para poder explicar el código de una forma más clara.

Para empezar, a continuación del primer comentario nos encontraremos con tres *"include"*, estos son necesarios para cualquier módulo. El primero, *"module.h",* se encarga de definir un montón de macros necesarios para el funcionamiento del módulo. El siguiente*,"kernel.h*", se usa para obtener información del núcleo. Y la última define los macros *module_init()* y *module_exit().*

Seguimos con la función *hello_start* y *hello_end*, que no hace falta inicializar pues *__init* y *__exit* pues habiendo incluido la *init.h*, estos fragmentos de código se encargar de ubicar algunas partes del código de Linux y al arrancar el núcleo el código se inicializa.

Dentro de las funciones, utilizamos *printk()* para comunicarnos con el usuario, pero esta función en realidad está pensada para registrar información o dar avisos.

Las dos últimas líneas de código son a las que hice referencia anteriormente en cargar y descargar módulos.

Ahora que hemos visto y entendido el ejemplo, probémoslo. Para ello primero hay que compilarlo

Compilar un módulo

Para compilar un módulo, necesitamos crear su fichero Makefile. Estos archivos son los ficheros de texto que utiliza *make*(Herramienta de generación de código que utilizaremos más tarde) para llevar la gestión de la compilación de programas. El Makefile del módulo hello luce tal que así:

```
obj-m += hello.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Tras crear el Makefile, en el terminal ejecutaremos el comando *make*, y el modulo estará compilado. Si realizamos un ls podremos ver como ahora hay, entre otros, un archivo llamado hello.ko. Para comprobar si el módulo funciona correctamente insertaremos este módulo en el kernel con el comando *"insmod hello.ko".* Ahora para ver si se ha introducido y ejecutado correctamente en el kernel, ejecutaremos *dmesg | tail -1* para ver el último acontecimiento ocurrido en el sistema o simplemente *dmesg* para verlos todos y buscar el mensaje del módulo

hello. Después borraremos el módulo del sistema con *rmmod* y repetiremos *dmseg* para ver el otro mensaje del módulo.

Documentación de un módulo

Para documentar de una forma adecuada un módulo necesitaremos utilizar una serie de macros. Por ejemplo, para definir el tipo de licencia del módulo debemos utilizar el macro *MODULE_LICENSE().* Para agregar autor, *MODULE_AUTHOR(),* para una descripción *MODULE_DESCRIPTION()* o para la versión del módulo *MODULE_VERSION().* Si lo añadimos a nuestro módulo quedaría así.

```
/*
 * hello.c – Módulo de ejemplo
 */
#include <linux/module.h
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE(“GPL”);
MODULE_AUTHOR(“Alejandro Pulido”);
MODULE_DESCRIPTION(“Un modulo simple que imprime hello world cuando se
ejecuta y goodbye cuando se extrae del kernel”);
MODULE_VERSION(“0.1”);

static int __init hello_start(void){
        printk(KERN_INFO "Hello, world\n");
        return 0;
}
static void __exit hello_end(void){
        printk(KERN_INFO "Goodbye, world\n");
}
module_init(hello_start);
module_exit(hello_end);
```

Las descripciones de los módulos son mejores hacerlas en inglés pues es el idioma dominante de la informática, y asi más usuarios podrán entender para qué sirve el módulo. La licencia de este módulo es GLP, es decir General Public License. Para saber acerca de otras licencias se pueden consultar páginas web como esta.

Esta documentación es irrelevante en el funcionamiento del módulo pero bastante útil para otros usuarios que a lo mejor quieren usar el módulo o simplemente saber más sobre él.

Pasar argumentos a un módulo

Los módulos, al igual que un programa en C, también pueden utilizar argumentos, pero no con argc o argv. Primero tenemos que incluir el header <linux/moduleparam.h>, después crearemos una variable como la que le pasaremos al módulo, es decir del mismo tipo. Lo siguiente será utilizar una macro para registrar el parámetro que se ha pasado. Esta macro es *“module_param()”* cuyos argumentos son el nombre de la variable, el tipo y los permisos de esta. Los permisos pueden ser S_IRUSR, S_IWUSR, S_IXUSR, S_IWGRP, S_IRGRP, donde la R, W o X cambia dependiendo si se dan permisos para leer, escribir o ejecutar respectivamente. Las últimas tres letras indican si el permiso es para usuario (USR) o grupo (GRP). Veamos un ejemplo:

```
/*
 * hello.c – Módulo de ejemplo
 *
 */
#include <linux/module.h
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/moduleparam.h>

MODULE_LICENSE(“GPL”);
MODULE_AUTHOR(“Alejandro Pulido”);
MODULE_DESCRIPTION(“Un modulo simple que imprime hello world cuando se
ejecuta y goodbye cuando se extrae del kernel”);
MODULE_VERSION(“0.1”);

int param_prueba = 0;

module_param(param_prueba, int, S_IRUSR | S_IWUSR);

static int __init hello_start(void){
        printk(KERN_INFO "Hello, world\n");
        printk(KERN_INFO “param = %d” ,param_prueba);
        return 0;
}
static void __exit hello_end(void){
        printk(KERN_INFO "Goodbye, world\n");
}
module_init(hello_start);
module_exit(hello_end);
```

Después de volver a ejecutar el comando make debemos insertar el módulo escribiendo en la consola *“insmod hello.ko param_prueba=(int)”.* Para ver si se ha ejecutado correctamente volvemos a llamar a dmesg.

En este último caso el argumento es un entero, pero si fuese un array de enteros (o de otro tipo) la sintaxis cambia.

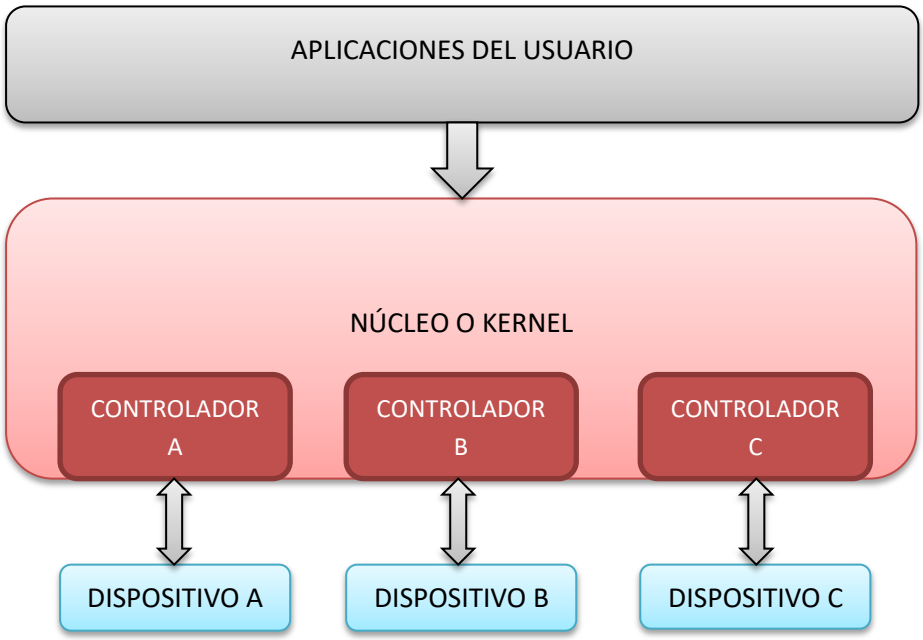
```
/*
 * hello.c – Módulo de ejemplo
 */
#include <linux/module.h
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/moduleparam.h>

MODULE_LICENSE(“GPL”);
MODULE_AUTHOR(“Alejandro Pulido”);
MODULE_DESCRIPTION(“Un modulo simple que imprime hello world cuando se
ejecuta y goodbye cuando se extrae del kernel”);
MODULE_VERSION(“0.1”);

int param_prueba[4] = {0,0,0,0};
int i = 0;

module_param_array(param_prueba, int, NULL, S_IRUSR | S_IWUSR);

static int __init hello_start(void){
        printk(KERN_INFO "Hello, world\n");
```

```
        while(param_prueba[i]){
                printk(KERN_INFO "param = %d" ,param_prueba[i]);
        i++;
        }
        return 0;
}
static void __exit hello_end(void){
        printk(KERN_INFO "Goodbye, world\n");
}
module_init(hello_start);
module_exit(hello_end);
```

Para ejecutarlo se harán los mismos pasos que para paso de variables normales, solo que los valores separados por comas., es decir *"insmod hello.ko param_prueba=(int, int, int)"*.

Controladores

Como comentamos anteriormente, los módulos podían ser controladores de dispositivos, pseudocontroladores de dispositivos (Ej. Controladores de red) o sistemas de archivos. Pues de forma general, los módulos suelen ser controladores de dispositivos, es decir software que controla alguna parte especifica del hardware.



Tipos de dispositivos

Hay dos tipos de dispositivos, los dispositivos de carácter y los de bloques

Dispositivos de carácter

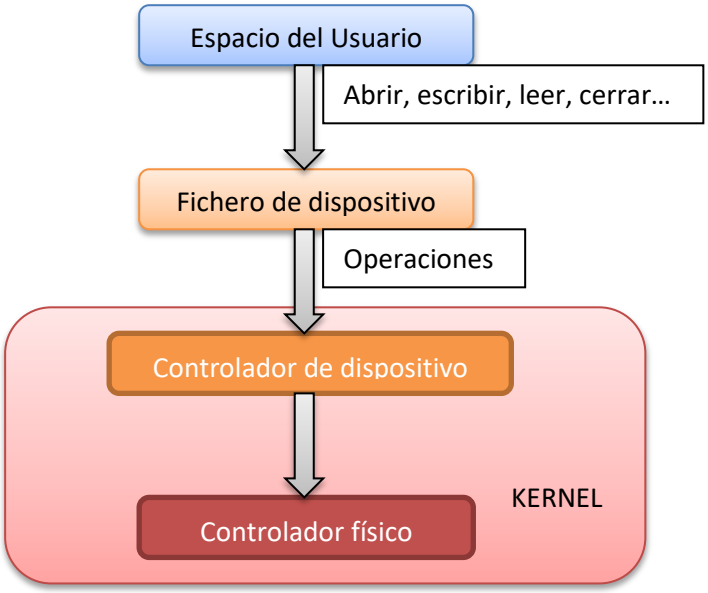
Son los que usan comunicación mediante bytes u octetos (caracteres) como la mayoría de dispositivos de entrada salida de un ordenador. Ej. Teclado, ratón…

Dispositivos de bloques

Aquellos que la comunicación la realizan mediante conjuntos de mayor tamaño que los caracteres. Estos son capaces de alojar una caché para los accesos, mientras que los de carácter no.

Para el usuario, si son de tipo c o b no importa, pero si para el sistema. Cabe destacar que las operaciones en modo c se hacen al ejecutarlas, es decir son sincronizadas mientras que las de bloques son asíncronas.

Como ya sabemos, todos los dispositivos en un SO tipo UNIX son ficheros. Cuando se trata de un fichero de dispositivo, que normalmente se encuentran en el directorio dev, sea de cualquier tipo, los datos leídos o escritos serán gestionados por el controlador respectivo.



Se podría decir que los ficheros de dispositivos son un puente entre el espacio del usuario y el kernel.

Si nos vamos al directorio dev y realizamos un ls –l podremos ver que delante de los permisos del archivo encontraremos el tipo de archivo (carácter o bloques) y después del nombre habrá dos números. El primero es el número mayor, este es usado por el kernel para identificar el controlador correcto cuando se accede al archivo. El segundo es el menor, el uso de este depende de del archivo.

Ejemplos

Vamos a ver un ejemplo de controlador.

https://github.com/derekmolloy/exploringBB/blob/master/extras/kernel/ebbchar/ebbchar.c

En este ejemplo podemos ver que el controlador puede abrir, leer, escribir y liberar el dispositivo al que maneja.

Si analizamos poco a poco el código de este nos daremos cuenta de que tiene todos los elementos que debe tener un módulo. Si obviamos la "burocracia" del código (registrar la clase del dispositivo, su controlador, escoger el número mayor…) y nos centramos en las funciones que dan utilidad al controlador nos daremos cuenta de que la función *dev_open()* se encarga de contar el número de veces que se ha abierto el controlador y enviar un mensaje al núcleo informando de estas. Si seguimos con *dev_release()* esta se encarga de enviar un mensaje de despedida al kernel. Ahora vamos a lo "importante", nos fijamos en *dev_write(),* en donde se guarda en "message" el mensaje enviado y su longitud. También guarda la longitud en una variable y la devuelve. Como es obvio, *dev_read*() se usa para leer del dispositivo los mensajes guardados, usando *copy_to_user()* para copiar el mensaje en el espacio del usuario. Si modificamos estos cuatro métodos podemos cambiar la funcionalidad del controlador, es decir, en vez de almacenar el mensaje y su longitud podríamos hacer que almacenase solo las vocales, o el mensaje al revés. De todas maneras será mejor probar el driver para ver si funciona, para ello nos valdremos de un programa muy básico de c.

```
#include<stdio.h>
#include<stdlib.h>
#include<fcntl.h>
#include<string.h>
#include<unistd.h>

int main(){
   int ret, fd;
   char buffer[256];
   char envio[256];

   fd = open("/dev/ebbchar", O_RDWR); //Abriendo con permiso de lectura y escritura
   if (fd < 0){
      perror("Fallo al abrir el dispositivo...");
      return -1;
   }
   printf("Escribe el mensaje que será enviado al dispositivo:\n");
   scanf("%[^\n]%*c", envio);                // Read in a string (with spaces)
   printf("Enviando mensaje... [%s].\n", envio);
   ret = write(fd, envio, strlen(envio)); // Send the string to the LKM
   if (ret < 0){
      perror("Error al escribir el mensaje en el dispositivo");
      return -1;
   }

   printf("Pulsa la tecla Enter para leer del dispositivo...\n");
   getchar();

   printf("Leyendo del dispositivo...\n");
   ret = read(fd, buffer, 256);
   if (ret < 0){
      perror("Error al leer del dispositivo");
      return -1;
   }
   printf("El mensaje recibido es: %s\n", buffer);
   printf("Final de programa\n");
   return 0;
}
```

(Este programa es una modificación del encontrado en http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/)

Este pequeño programa abre el controlador y escribe el mensaje pedido por consola en el dispositivo ficticio que maneja el controlador, tras pulsar “enter” lee de este y lo imprime por consola.

No olvidemos que el controlador es un módulo, por lo que para probarlo habrá que tener su *Makefile*, insertarlo en el núcleo y así podremos utilizarlo con nuestro programa.

Conclusión

Hemos visto por encima que es el Kernel, sus tipos más utilizados y para qué sirve. Después de esta introducción hemos hablado sobre los módulos del núcleo, como insertarlos y descargarlos utilizando *insmod* y *rmmod* respectivamente. Continuamos con un ejemplo de un módulo muy sencillo, el cuál fuimos completando poco a poco la funcionalidad de su código, añadiendo su documentación, viendo su *Makefile*…. Para finalizar nos centramos en un tipo de módulo, el que generalmente es más usado, los controladores de dispositivos, viendo un ejemplo y ejecutando un programa para ver su funcionamiento.

BIBLIOGRAFIA

Información sobre el Kernel:

- https://afteracademy.com/blog/what-is-kernel-in-operating-system-and-what-are-the-various-types-of-kernel
- https://www.youtube.com/watch?v=-O6GsrmOUgY

Información sobre módulos:

- https://tldp.org/LDP/lkmpg/2.6/lkmpg.pdf
- https://tldp.org/LDP/tlk/modules/modules.html

Información sobre controladores:

- http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/

Lista de videos informativos:

- https://www.youtube.com/watch?v=-O6GsrmOUgY&list=PL16941B715F5507C5&index=4

Libros:

- Building and running modules
- Linux Kernel Development
- Understanding the Linux Kernel