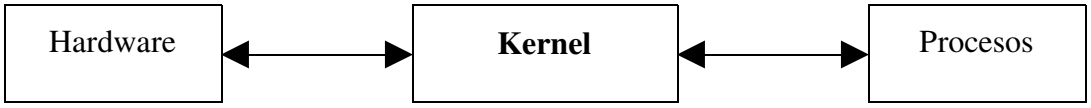
Módulos en Linux

Año 2006 – Kernel 2.6

Introducción

El kernel, debe realizar varias tareas, entre ellas responder al hardware y a los procesos.



Antiguamente para agregarle funciones al kernel, por ejemplo un driver, la opción era el relinkeo estático, el problema era que si el driver funcionaba mal se debia repetir todo el proceso.

Por eso es que se desarrollaron la capacidad de agregar modulos, al Kernel, cuyo codigo corre en el anillo cero.

Un módulo tiene que tener al menos dos funciones: i*nit_module( ) y cleanup_module ( ).* Que son llamdas cuando el modulo se inserta en el kernel y cuando es removido respectivamente.

init_module ( ) generalmente "registra", un handler, que es luego el que será utilizado, o reemplaza una función del kernel con su propio codigo.(y luego llama a la anterior).

El objetivo de *cleanup_module()* es deshacer todo lo que hizo la anterior.
Ambas se encuentran en *module.h*

Un module es un archivo objeto que se linkea con el kernel en tiempo de ejecucion.

Las forma de compilar los módulos varia en las diferentes versiones de kernel, en nuestro caso utilizaremos el kernel 2.6

Comando *lsmod*

Se utiliza para ver los modulos cargados. Este comando simplemente imprime en pantalla en contenido de */proc/modules*. (Se vera mas adelante que lo que hace es leer el archivo y el kernel le informara mediante éste, los módulos que se encuentran corriendo)

No todo tipo de código puede ser cargado como un modulo. Por ejemplo si se quiere modificar la estructura *task_struct* para agregarle un campo nuevo, se tendrá que recompilar todo el kernel.

El kernel tiene dos tareas al insertar un modulo:

1- Asegurarse de que todo el kernel pueda ver los símbolos globales del modulo (como ser el punto de entrada). Asi tambien el modulo debe conocer las direcciones de los símbolos globales del kernel y de otros modulos.

2- Rastrear el uso de los modulos. No se puede "bajar" un modulo cuando alguien lo esta utilizando.

Comando *insmod*

Los modulos son archivos .o de formato ELF. Se deben cargar en memoria con el comando *insmod*. A partir del kernel 2.6 se busca estandarizar los objetos de modulo, por lo tanto se utiliza la extension .ko, para diferenciar a los módulos.

Las funciones que se hayan utilizados en el modulo (por ejemplo *printk)* , se resuelven cuando se hace el *insmod* del modulo.

Si alguna función utilizada no existe dentro del kernel, se vera un error y no se insertará el modulo.

Para los ejemplos de esta guia el nombre del modulo lleva la extensión **.ko**.

Comando *rmmod*

Se utiliza para remover el módulo insertado, por ejemplo *rmmod ej1*.
Al desinstalarlo correrá la función *cleanup_module().*

Funciones externas

Todo lo que el kernel tiene declarado como externo esta en */proc/kallsyms*. Al editar el archivo *kallsyms* se puede ver que al principio están las variables y funciones de los modulos y luego una gran cantidad de funciones exportadas por el kernel.
Por ejemplo: *printk, sprintf, etc.*.

Versión de Kernel

Todos los ejemplos de este tutorial fueron compilados con el kernel 2.6.17-1, la distribución utilzada es Fedora Core 5, pero con upgrade de kernel.

Makefile

Para todos los ejemplos se utilizo el mismo Makefile.

```
obj-m   := ej1.o

all:
	make -C /lib/modules/2.6.17-1.2174_FC5/build M=$(PWD) modules
clean:
	make -C /lib/modules/2.6.17-1.2174_FC5/build M=$(PWD) clean
```

Al compilar obtendremos un archivo **.ko** que será el módulo a instalar en el kernel

Ejemplos

Las salida estandar de un módulo, por ejemplo al utilizar *printk* se producirá en el archivo /var/log/messages. Por lo tanto se recomienda chequear este archivo con el comando “tail -f /var/log/messagges”

Ejemplo 1

(hola.c)

Modulo que solo tiene init_module ( ) y cleanup_module ( ) y utiliza printk ( ).

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

int init_module( )
{
        printk("Este mensaje lo escribe el modulo.\n");
        return 0;
}

void cleanup_module( )
{
        printk("Se desinstala el modulo.\n");
}
```

Ejemplo 2

Obtiene el nombre (del ejecutable) y el PID del proceso actual

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

int init_module()
{
        printk("El nombre es: %s  El PID es: %i \n",current->comm,current->pid);

        return 0;
}

void cleanup_module()
{
        printk("Se desinstala current.\n");
}
```

Se debe incluir *sched.h* para poder utilizar *current*. En realidad *current* esta definido en *current.h*, es un puntero a *task_struck*.

```
printk("The process is \"%s\" (pid %i)\n", current->comm, current->pid);
```

Desde el punto de vista de un modulo *current* es igual a *printk*, es decir, es una referencia externa.

En este caso imprimirá el nombre y el PID de insmod, pero si la misma linea de printk se coloca dentro de un handler de system call se puede obtener el pid del proceso que llamo a la system call.

Task_struct

Esta definida en *sched.h,* algunos de los datos que se pueden obtener del proceso son:

- pid
- nombre (comm)
- priority
- p_pptr (puntero a task_struc del padre).

Para pasar argumentos

Los modulos pueden recibir argumentos a traves de la linea de comandos.
Por ejemplo:

Insmod hello.ko entero=33 mensaje=Pepe

Dichos parámetros no se manejan con argv y argc, sino que se debe declarar variables globales en el módulo y luego utilizar la macro *module_param()* para utilizarlas dentro del módulo.

module_param recibe 3 argumentos: El nombre de la variable, el tipo, y permisos derivados del sysfs.

Ejemplo 3

```
/* param.c
   Ejemplo de modulo que recibe dos parametros: entero y string
   Declara una fucion global para que pueda ser utlizada por otros
   modulos
*/

#include<linux/kernel.h>
#include<linux/module.h>
#include<linux/moduleparam.h>

int entero=12;
char* mensaje="monos";

module_param(entero,int,0);    // declara a entero como parametro y como int
module_param(mensaje,charp,0);   // declara a mensaje como parametro y como string

void fglobal ( char* string )
{
        printk(string);
}

int init_module()
{
        printk("El entero es : %d \n", entero );
        printk("El mensaje es: %s \n",mensaje);
        return 0;
}

void cleanup_module()
{
        printk("Se desinstala el modulo param.\n");
}
```

Con el Ejemplo 3 se puede probar:

- insmod ej3.ko
- insmod ej3.ko entero=55
- insmod ej3.ko mensaje=cualquiera

Recuerde siempre remover el módulo antes de volver a instalarlo.

El file system /proc

Es un file system que se encuentra en memoria y se utiliza para que el kernel se pueda comunicar con los procesos.

Si hacemos un *ls –l* al directorio */proc* veremos que en su mayoria son archivos de 0 bytes, esto es porque al realizar el system call *read* sobre ellos el kernel entregará los valores pedidos.

- Correr los siguiente comandos:
  - cat /proc/cpuinfo
  - cat /proc/modules

En el ultimo ejemplo podemos ver que es la misma salida que produce el comando *lsmod*.

Al leer los archivos del filesystem /proc, lo que hacemos es requerir información actual del kernel a traves de un punto de entrada controlado.

Al escribir el los archivos del filesystem /proc, lo que hacemos es enviar información al kernel. (Tener en cuenta que esto lo estamos haciendo desde del espacio Usuario !)

Ejemplo 4

```
/*
 *  procfs.c -  create a "file" in /proc
 */

#include <linux/module.h>       /* Specifically, a module */
#include <linux/kernel.h>       /* We're doing kernel work */
#include <linux/proc_fs.h>      /* Necessary because we use the proc fs */

struct proc_dir_entry *Our_Proc_File;

int
procfile_read(char *buffer,
              char **buffer_location,
              off_t offset, int buffer_length, int *eof, void *data)
{
```

```
        printk(KERN_INFO "inside /proc/test : procfile_read\n");

        int len = 0;                    /* The number of bytes actually used */
        static int count = 1;

        if (offset > 0) {
                printk(KERN_INFO "offset %d : /proc/test : procfile_read, \
                     wrote %d Bytes\n", (int)(offset), len);
                *eof = 1;
                return len;
        }

        /*
         * Fill the buffer and get its length
         */
        len = sprintf(buffer,
                        "For the %d%s time, go away!\n", count,
                        (count % 100 > 10 && count % 100 < 14) ? "th" :
                        (count % 10 == 1) ? "st" :
                        (count % 10 == 2) ? "nd" :
                        (count % 10 == 3) ? "rd" : "th");
        count++;

        /*
         * Return the length
         */
        printk(KERN_INFO  "leaving /proc/test : procfile_read, wrote %d Bytes\n", len);
        return len;
}

int init_module()
{
        int rv = 0;

/* Se pide la entrada al fs /proc con el nombre y los permisos deseados */
        Our_Proc_File = create_proc_entry("test", 0644, NULL);

/* Se completa con el puntero a función la función que correrá al ser leido el archivo */
        Our_Proc_File->read_proc = procfile_read;

        Our_Proc_File->owner = THIS_MODULE;
        Our_Proc_File->mode = S_IFREG | S_IRUGO;
```

```
        Our_Proc_File->uid = 0;
        Our_Proc_File->gid = 0;
        Our_Proc_File->size = 37;

        printk(KERN_INFO "Trying to create /proc/test:\n");

        if (Our_Proc_File == NULL) {
                rv = -ENOMEM;
                remove_proc_entry("test", &proc_root);
                printk(KERN_INFO "Error: Could not initialize /proc/test\n");
        } else {
                printk(KERN_INFO "Success!\n");
        }

        return rv;
}


void cleanup_module()
{
        remove_proc_entry("test", &proc_root);
        printk(KERN_INFO "/proc/test removed\n");
}
```

Explicación del Ejemplo 4

Analizaremos en un principio la carga del módulo en la función *init_module()*, donde podemos ver que se intenta crear la entrada en /proc. Como en este caso es solamente un archivo de lectura solo completamos el campo *read_proc* de la estrucutra. Al querer leer del archivo esta será la fución que correrá.

Dentro de la función procfile_read podemos observar los dos tipos diferentes de salida, una con printk y la otra con sprintf.

Ademas debemos tener en cuenta que la funcion *procfile_read()* debe retornar la cantidad de bytes leidos en el buffer.

De los argumentos que recibe, los mas importantes son:

- (buffer) Un buffer que es devuelto a la aplicacion que consulta. (Por ejemplo el comando “cat”)
- (offset) La posicion actual dentro del archivo. Observar la salida /var/log/messages al ejecutar varias veces el comando “cat /proc/test”.

Pruebas a realizar con el el Ejemplo 4

- Realizar lecturas sucesivas sobre el archivo /proc/test. Por ejemplo con “cat /proc/test”
- Observe el archivo /var/log/messages mientras realiza las pruebas del punto anterior. ¿ Que observa ?
- Intente escribir en el archivo /proc/test. Por ejemplo “echo 1 > /proc/test/
- Intente cargar dos veces el modulo sin haberlo descargado
- ¿ A que se debe el compartamiento de los dos puntos anteriores ?
- Que sucede si Ud intenta borrar el archivo /proc/test. ¿ Y si le cambia los permisos para poder hacerlo ?

Modulos en el Kernel

Al hacer un *insmod* el kernel ubica en memoria al modulo junto con un objeto que lo describe:

- Tamaño.
- Contador de uso. (usage counter)
- Tabla de símbolos exportados.
- Lista de modulos a los cuales referencia.(dependencia)
- Lista de modulos por los cuales es referenciado. (quien depende de el).
- Forma de inicio (Initialization method).
- Forma de fin. (Clean up method).

Dependencia de modulos

A nivel de objetos es una lista doblemente enlazada que se actualizan dinámicamente cada vez que un modulo es cargado o removido.

Un módulo B puede necesitar una funcion o variable de un módulo A, esto se ve reflejado en la lista del objeto. Para saber si se puede remover se utiliza el usage counter.

Al ejecutar el comando *lsmod*, se pueden ver lineas como esta:

Module	*Size*	*Used by*
vfat	*12481*	*1*
fat	*51933*	*1 vfat*

En este caso podemos ver que el módulo *vfat* está utilizando objetos del módulo *fat.*
Si quisiemos desinstalar el módulo fat obtendriamos un mensaje del tipo:

[root@pampero]# rmmod fat
ERROR: Module fat is in use by vfat

Modulos por demanda

Al iniciarse el sistema corre */sbin/depmod* (se puede ver "Finding modules dependencies [OK]") que examina todos los modulos compilados que se encuentran en */lib/modules*. Analiza las dependencias y las refleja en el archivo *modules.dep.*
Por ejemplo el modulo *vfat.o* necesita que este cargado el modulo *fat.o*.

El programa modprobe

Es similar a *insmod* , puede cargar un modulo por la linea de comando, con la diferencia que *modprobe* chequea las dependencias del modulo a cargar y si hace falta carga los mudulos que se necesiten.

Insmod solo carga el modulo y trata de resolver las llamadas con los símbolos exportados por otros modulos.

Modprobe es invocado por el modulo demonio *kmod* que detecta cuando se necesita de algun otro modulo.