SISTEMAS OPERATIVOS

Tecnológico de Monterrey, Campus Querétaro

Laboratorio 4 - Estructuras del sistema operativo

1. Introducción

En este laboratorio aprenderás cómo crear un módulo kernel y cargarlo en el kernel de Linux. El laboratorio se puede completar utilizando la máquina virtual Linux que está disponible en la plataforma. Deberás usar un editor de texto para escribir los programas en C (`nano`, `vim`)y un compilador de c (`gcc`) para compilar los programas.

Como descubrirás, la ventaja de desarrollar este tipo de módulos es que es un método relativamente fácil de interactuar con el kernel, lo que te permite escribir programas que invocan funciones del kernel directamente. Es importante que tengas en cuenta que estás escribiendo un código que interactúa directamente con el kernel de Linux. ¡Eso normalmente significa que cualquier error en el código podría bloquear el sistema! Sin embargo, dado que usarás una máquina virtual, cualquier falla en el peor de los casos solo requerirá reiniciar el sistema.

2. Creando Módulos Kernel

La primera parte de este laboratorio implica seguir una serie de pasos para crear y cargar un módulo en el kernel de Linux.

Puede enumerar todos los módulos del kernel que están actualmente cargados al ingresar el comando:

```
lsmod
```

Este comando mostrará la lista de los módulos que actualmente se encuentran cargados en el núcleo en tres columnas: nombre, tamaño y dónde se usa el módulo.

El siguiente programa (llamado `simple.c`) ilustra un módulo de kernel muy básico que imprime mensajes apropiados cuando el módulo kernel se carga y descarga.

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

/* This function is called when the module is loaded. */
int simple_init(void)
{
printk(KERN_INFO "Loading Module\n");

return 0;
}

/* This function is called when the module is removed. */
```

```
void simple_exit(void) {
printk(KERN_INFO "Removing Module\n");
}

/* Macros for registering module entry and exit points. */
module_init( simple_init );
module_exit( simple_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

La función `simple_init()` es el punto de entrada del módulo, que representa la función que se invoca cuando el módulo se carga en el kernel. De manera similar, la función `simple_exit()` es el punto de salida del módulo, la función que se llama cuando el módulo se elimina del kernel.

La función de entrada del módulo debe regresar un valor entero, donde 0 representa el éxito y cualquier otro valor que representa el error. La función de salida del módulo regresa `void`. Ni la función de entrada, ni la función de salida reciben ningún parámetro. Las dos siguientes macros se utilizan para registrar los puntos de entrada y salida del módulo con el kernel:

```
module_init( simple_init );
module_exit( simple_exit );
```

Observa cómo las funciones de entrada y salida del módulo hacen llamadas a la función `printk()`. `printk()` es el equivalente del núcleo de `printf()`, sin embargo, su salida se despliega en un búfer de registro del kernel cuyos contenidos pueden leerse con el comando `dmesg`. Una diferencia entre `printf()` y `printk()` es que `printk()` nos permite especificar un indicador de prioridad cuyos valores se dan en el archivo header <linux / printk.h>. En este caso, la prioridad es `KERN_INFO`, que se define como un mensaje informativo.

Las líneas finales - `MODULE_LICENSE()`, `MODULE_DESCRIPTION()` y `MODULE_AUTHOR()` - describen detalles sobre la licencia del software, la descripción del módulo y el autor. Realmente no es necesaria esta información, pero la incluimos porque es una práctica estándar en el desarrollo de módulos kernel.

Este módulo kernel `simple.c` se compila utilizando el archivo `Makefile` que acompaña al código fuente de este laboratorio. Para compilar el módulo, ingresa la siguiente línea de comando:

```
make
```

La compilación produce varios archivos. El archivo `simple.ko` es el módulo del kernel compilado. El siguiente paso ilustra la inserción del módulo en el kernel de Linux.

2.1. Carga y eliminación de módulo

Los módulos kernel se cargan usando el comando `insmod`, que se ejecuta de la siguiente manera:

```
sudo insmod  simple.ko
```

Para verificar si el módulo se ha cargado, ejecuta el comando `lsmod` y busca el módulo `simple`. Recuerda que la función de entrada del módulo se invoca cuando el módulo se carga en el kernel. Para verificar el contenido de este mensaje en el buffer del registro del kernel, ejecuta el comando

```
dmesg
```

Deberías ver el mensaje `"Loading Module."`

La eliminación del módulo kernel implica invocar el comando `rmmod` (observa que el `.ko` no es necesario):

```
sudo rmmod simple
```

Debido a que el buffer del registro del kernel se llena rápidamente, a menudo tiene sentido borrar el buffer periódicamente. Esto se logra de la siguiente manera:

```
sudo dmesg -c
```

2.2. Ejercicio a realizar

Sigue los pasos descritos anteriormente para crear el módulo kernel y cargar y descargarlo. Asegúrate de verificar el contenido del buffer del registro del kernel usando `dmesg` para saber si seguiste los pasos correctamente.

3. Estructuras de Datos Kernel

La segunda parte de este laboratorio implica la modificación del módulo kernel para que use la estructura de datos de la lista enlazada definida en el kernel.

El kernel de Linux proporciona varias estructuras. Aquí exploramos la lista circular doblemente enlazada, que está disponible para los desarrolladores del kernel. Mucho de lo que hablamos está disponible en el código fuente de Linux - en este caso, el archivo header `<linux/list.h>` -, te recomendamos que examines este archivo a medida que avances.

Inicialmente, debes definir una estructura que contenga los elementos que se insertarán en la lista enlazada. La siguiente estructura C define un cumpleaños:

```
struct birthday {
  int day;
  int month;
  int year;
  struct list_head list;
}
```

Observa el campo `struct list_head list`. La estructura `list_head` se define en el archivo header `<linux/types.h>`. Su objetivo es insertar este nodo dentro de la lista enlazada. La estructura `list_head` es bastante simple: solo tiene dos campos, `next` y `prev`, que apuntan a los siguientes y anteriores en la lista. Al incorporar la lista enlazada dentro de la estructura, Linux hace posible administrar la estructura de datos con una serie de funciones de macro.

3.1. Insertar elementos en la lista vinculada

Podemos declarar un objeto `list_head`, que usamos como referencia al encabezado de la lista usando la macro `LIST_HEAD()`:

```
static LIST_HEAD(birthday_list);
```

Esta macro define e inicializa la variable `birthday_list`, que es del tipo `struct list_head`.

Creamos e iniciamos instancias de la estructura `birthday` de la siguiente manera:

```
struct birthday *person;

person = kmalloc(sizeof(*person), GFP_KERNEL);
person->day = 2;
person->month = 8;
person->year = 1995;
INIT_LIST_HEAD(&person->list);
```

La función `kmalloc()` es el equivalente del kernel de la función `malloc()` de nivel de usuario para asignar memoria, excepto que está asigna la memoria del kernel. (El indicador `GFP_KERNEL` indica la asignación rutinaria de la memoria del kernel.) La macro `INIT_LIST_HEAD()` inicializa el campo lista en la estructura `bithday`. A continuación, podemos agregar esta instancia al final de la lista enlazada utilizando la macro `list_add_tail()`:

```
list_add_tail(&person->list, &birthday_list);
```

3.2. Atravesando la lista enlazada

Recorrer la lista implica utilizar la macro `list_for_each_entry()`, que recibe tres parámetros:

- Un apuntador a la estructura que se recorre.
- Un apuntador al inicio de la lista que se recorre.
- La variable que contiene la estructura `list_head`.

El siguiente código ilustra esta macro:

```
struct birthday *ptr;

list_for_each_entry(prt, &birthday_list, list) {
  /* en cada iteraciOn, ptr indica cuAl es */
  /* el nodo actual de la lista */
}
```

3.3. Eliminar elementos de la lista enlazada

La eliminación de elementos de la lista implica el uso de la macro `list_del()`, que recibe un apuntador a una estructura `list_head`.

```
list_del(struct list_head *element);
```

Esto elimina el elemento de la lista mientras mantiene la estructura del resto de la lista.

Quizás el enfoque más simple para eliminar todos los elementos de una lista enlazada es eliminar cada elemento a medida que se recorre la lista. La macro `list_for_each_entry_safe()` se comporta de forma muy parecida a `list_for_each_entry()`, excepto que recibe un argumento adicional que mantiene el valor del siguiente apuntador del elemento que se está

eliminando. (Esto es necesario para preservar la estructura de la lista.) El siguiente ejemplo de código ilustra esta macro.

```
struct birthday *ptr, *next;

list_for_each_entry_safe(ptr, next, &bithday_list, list) {
  /* prt apunta al nodo actual de la lista */
  list_del(&prt->list);
  kfree(ptr);
}
```

Tenga en cuenta que después de eliminar cada elemento, debemos devolver la memoria que previamente había sido asignada con `kmalloc()` al kernel con la llamada `kfree()`. La administración cuidadosa de la memoria, que incluye la liberación de memoria para evitar fugas de memoria, es crucial al desarrollar código a nivel kernel.

3.4. Ejercicio a realizar

En el punto de entrada del módulo, crea una lista vinculada que contenga cinco elementos de cumpleaños de estructura. Recorre la lista enlazada y muestra su contenido en el búfer de registro del kernel. Invoca el comando `dmesg` para asegurarse de que la lista esté construida correctamente una vez que se haya cargado el módulo kernel.

En el punto de salida del módulo, elimina los elementos de la lista enlazada y devuelve la memoria libre al kernel. Nuevamente, invoca el comando `dmesg` para verificar que la lista haya sido eliminada una vez que el módulo kernel haya sido descargado.