Trabajo Práctico - Sistemas de Computación

Autores
Maero Facundo - 38479441
Colazo Agustín - 38986764
Gonzalez Gustavo - 7721064

        - API de Timers de Linux
        - Apertura y Cierre del Device
        - Lectura y Escritura del Device
        - Interrupciones y Procesos Bloqueados
    - Desarrollo de la Interfaz de Usuario
    - Makefile y uso
    - Bibliografía
- CppCheck

Consignas del Trabajo

- **Ejercicio 1** El usuario, desde el espacio de usuario, ingresa un número en el programa, el cual debe inicializar un timer en espacio de kernel, utilizando la API de timers de Linux. Una vez que el timer finalice, se llama a una función que le envíe un mensaje, notificando al usuario.
- **Ejercicio 2** Utilizando la API de interrupciones de GNU/Linux, el programa debe generar una interrupción al finalizar el conteo. La interrupción debe indicar al usuario que ha ocurrido.

El desarrollo debe realizarse sobre sobre la placa de desarrollo Intel Galileo/Raspberry Pi o similar compatible.

Guía de Desarrollo

Introducción y Hello World

Para desarrollar este proyecto es necesaria una PC con Sistema Operativo Linux y los Kernel Headers instalados. La notebook utilizada para el desarrollo estaba ejecutando una versión limpia de Xubuntu 17.04 y no fue necesario instalar ningun header.

Es buena práctica comprobar que todas las herramientas funcionen correctamente, compilando y ejecutando el driver más simple posible, un driver "Hello World". Para ello se escribe el siguiente código en un archivo .c:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
printk(KERN_ALERT "Hello, world\n");
return 0;
}
static void hello_exit(void)
```

```
{
printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Y lo guardamos en la carpeta `src` como `hello-1.c`, junto con un archivo `Makefile` para poder compilar el proyecto. El Makefile contiene lo siguiente:

```
obj-m += hello-1.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
```

Luego debemos dirigirnos con una terminal a la carpeta del proyecto con `cd` y ejecutar `make` para compilar nuestro primer driver. Veremos que hay varios archivos más en la carpeta. El que nos interesa es `hello-1.ko`.

Para instalarlo en el kernel solo debemos ejecutar `sudo insmod hello-1.ko`. Asimismo, para eliminarlo se usa `sudo rmmod hello-1`.

Este simple driver imprime en los logs del kernel un mensaje al instalarlo, y uno al eliminarlo. Estos pueden verse ejecutando el comando `dmesg -wH`, que muestra por la terminal todo el log del kernel desde que se encendió la PC, en formato legible para el usuario, y de manera continua para ver los cambios en el log.

Conceptos Útiles

Antes de comenzar a escribir código es conveniente aclarar algunos conceptos claves para comprender el desarrollo del módulo y su funcionamiento.

Los módulos del Kernel de Linux pueden ser de dos tipos: de bloques o de caracteres. El desarrollado aquí es un driver por caracteres.

Sin importar su tipo, todos los drivers desarrollados son ampliamente dependientes de la versión del Kernel utilizada para compilarlos. En realidad es posible compilar un módulo para otra versión del núcleo (para ello son necesarios los headers de la misma), pero trabajo explica la compilación para la versión de núcleo en uso. Por este motivo, si se quisiera instalar un driver precompilado, es muy probable que no funcione.

Majors y Minors

Los dispositivos por caracteres se acceden mediante su nombre en el sistema de archivos. Se encuentran en la ruta `/dev`, y se identifican con la letra "c" en la primer columna de la salida de `ls -l` Con este comando también es posible ver más información. Los números que aparecen a la izquierda de la fecha de última modificación son los **major** y **minor** numbers.

- El major indica el driver asociado al dispositivo. Así, por ejemplo, todos los dispositivos

de disco tendrán asociado el mismo driver.

- El minor identifica, dentro de un mismo major, a cada dispositivo en particular. Siguiendo el anterior ejemplo, si una PC tiene instalados más de un disco duro, tendrán asignados diferentes minor numbers.

Desarrollo del driver

Funciones principales

En un driver las dos funciones principales son `module_init()` y `module_exit()`, que se ejecutan al instalar y desinstalar el módulo.

La función de inicialización creará todas las estructuras de control necesarias para que el módulo funcione, se registrará con el Sistema Operativo para ser utilizable por el usuario, iniciará el módulo de Timer y avisará de su estado en los logs del Kernel.

Estructura file_operations

Deben definirse las funciones que va a realizar, en una estructura especial llamada `file_operations`, que contiene punteros a las funciones que puede realizar un device driver. En nuestro caso se asignan las 4 funciones básicas a utilizar, y luego se definirá su implementación. La estructura, llamada `my_fops` tiene esta forma:

```
static struct file_operations my_fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};
```

Tipo de dato dev_t

Este tipo de dato permite alojar los número major y minor de un dispositivo. Está definido en `<linux/types.h>`

Estructura cdev

El primer paso es inicalizar una estructura de tipo `cdev` como un puntero a una estructura especial del Kernel, que identifica a los dispositivos de caracteres. Esta estructura contiene una referencia a las `file_operations` del módulo, y una referencia al módulo mismo.

Se la define como:

```
static struct cdev *my_cdev;
```

Y se la inicializa con las siguientes instrucciones:

```
my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

```
my_cdev->owner = THIS_MODULE;
```

Reservar y desalojar Device Numbers

Un driver debe solicitar un major y minor number al sistema. Esto puede realizarse de manera estática (solicitando un major puntual), o dinámica (su major es asignado por el Sistema Operativo, con un número libre). Se prefiere la segunda metodología.

Una vez listo el paso anterior, se solicita al Kernel que nos asigne un major number:

```
alloc_chrdev_region(&my_dev_t, 0, 1, "my_timer1");
```

Donde se pasa como argumento la estructura de tipo `my_dev_t` que alojará los números asignados, `0` es el primer minor solicitado, `1` es la cantidad de dispositivos, y finalmente se pasa el nombre del mismo.

Ahora es posible ver los números de nuestro dispositivo con las macros `MAJOR(my_dev_t)` y `MINOR(my_dev_t)`.

Device Registration

Lo que sigue es hacerle saber al Kernel sobre la existencia de nuestro módulo, con la siguiente función:

```
cdev_add(my_cdev, my_dev_t, 1);
```

Aquí el módulo ya está corriendo en el sistema, y puede ser llamado por otros módulos, o programas de usuario.

Otro paso a seguir es registrar el Device y Class de nuestro dispositivo. Esto permite que aparezca en el archivo /proc/devices. Una class es una abstracción que permite englobar diversos drivers bajo un mismo tipo, por ejemplo, una unidad SATA y una IDE son manejados por diferentes drivers, pero en esencia son del tipo driver de disco. Teniendo inicializadas las estructuras de antemano:

```
static struct class*  myCharClass  = NULL;
static struct device* myCharDevice = NULL;
```

Se las registra así, donde CLASS_NAME es una macro con el nombre de nuestro driver:

```
myCharClass = class_create(THIS_MODULE, CLASS_NAME);
myCharDevice = device_create(myCharClass, NULL, my_dev_t, NULL, DEVICE_NAME);
```

Device cleanup

Para limpiar el driver de nuestro sistema, es decir realizar una desinstalación correcta, es necesario deshacer todos los pasos que se siguieron al instalarlo. Esto engloba desalojar los major y minor solicitados, eliminar la estructura `cdev`, desregistrar el device y su clase, y borrar las estructuras correspondientes:

```
unregister_chrdev_region(my_dev_t, 1);
cdev_del(my_cdev);
device_unregister(myCharDevice);
device_destroy(myCharClass, major);
class_unregister(myCharClass);
class_destroy(myCharClass);
```

API de Timers de Linux

La API que nos proporciona el sistema nos permite definir un timer, indicarle el tiempo que deberá contar, e iniciar su ejecución. Linux cuenta con dos timers: el primero, de baja resolución (~4 ms), y uno de alta resolución, que discrimina intervalos en el orden de los nanosegundos.

Para el práctico se va a utilizar la API convencional, ya que su resolución se considera suficiente. Lo primero que debe hacerse es crear una estructura de tipo timer, e inicializarla. Esto se realiza en la función init del driver:

```
static struct timer_list my_timer;
setup_timer( &my_timer, my_timer_callback, 0 );
```

Donde `my_timer_callback` es la función que se ejecutará al finalizar la cuenta del timer. Aquí el timer ya se encuentra inicializado y listo, solo hace falta pasarle el tiempo que queremos que cuente y empezará a trabajar. Esto se hace con la función `mod_timer` y un instante en el futuro, dado por el valor actual de `jiffies` más el tiempo deseado `timer_value`:

```
mod_timer( &my_timer, jiffies + msecs_to_jiffies(timer_value) );
```

Luego simplemente se define la función de trigger, donde simplemente se avisa que el timer finalizó.

```
void my_timer_callback( unsigned long data )
{
    printk( "my_timer_callback called (%ld).\n", jiffies );
}
```

Apertura y Cierre del Device

Como el driver es muy simple y no hay datos dinámicos para los que haya que reservar memoria, la función `device_open()` simplemente retorna. Asimismo, la función `device_close()` no tiene que cerrar ni apagar ningún hardware, por lo que también retorna.

Lectura y Escritura del Device

Para la lectura y escritura si es necesaria la realización de varias acciones. En este caso en particular, cuando el usuario escribe en el driver, le pasa el tiempo en milisegundos para el timer. El control de este valor se realiza en la parte de usuario, por lo que se supone un valor correcto.

Por lo tanto, se copia el contenido del buffer de usuario `buff` en el string `msg` en espacio de Kernel. Luego se apunta `msg_Ptr`, un char pointer, al mensaje, para luego devolverlo al usuario en un `read()`.

La variable `long timer_value` guarda el tiempo a pasar al timer. Se extrae esta cantidad con la función `kstrtol()`, similar a `atoi()` en espacio de usuario. Finalmente se inicia el timer con la función wrapper `my_timer_startup()`, que simplemente llama a `mod_timer()`.

```
copy_from_user (msg, buff, len);
msg_Ptr = msg;

res = simple_strtol(msg, 10, &timer_value);
if(res){
    printk("my_timer1 -> parsing error\n");
    return res;
}

my_timer_startup(timer_value);
```

La función de lectura está estructurada de la siguiente manera:

```
error_count = copy_to_user(buffer, msg, length);
return error_count;
```

Se copia al string de usuario `buffer` el contenido de `msg`, y se retorna `error_count`. En caso exitoso su valor será cero.

Interrupciones y Procesos Bloqueados

El callback del timer de Linux se ejecuta en contexto de interrupción, por lo que no puede incluir funciones como sleep o relacionadas.

Esto puede comprobarse de manera simple con la función `in_interrupt()`, incluída en la librería `<asm/hardirq.h>` Esta función permite saber si un bloque de código del Kernel se encuentra ejecutándose en contexto de interrupción.

Por lo tanto, nuestro `my_timer_callback` se está ejecutando en contexto de interrupción, lo cual es muy útil para avisar a procesos relacionados sobre la ocurrencia de un evento.

Utilizaremos una estructura del Kernel, llamada **Wait Queue**, que consiste en una lista de procesos que esperan un evento. Se la declara de manera estática con la macro:

```
DECLARE_WAIT_QUEUE_HEAD(wq);
```

Luego, para incorporar la versión interrumpible a nuestro driver, agregamos la siguiente función, que agrega el proceso actual a la wait queue `wq` hasta que sea despertado.

```
wait_event_interruptible(wq, timer_done != 0);
```

El segundo argumento es una condición, una expresión booleana evaluada antes y después de dormir. Hasta que no evalúe a `True`, el proceso seguirá durmiendo.

Como se explicó anteriormente, se aprovechará la naturaleza de la ejecución de `my_timer_callback()` para despertar al proceso esperando. Esto se realiza explícitamente con la función:

```
wake_up_interruptible(&wq);
```

Así, el proceso bloqueado en la mitad de la ejecución de una función del driver continuará directamente donde fue bloqueado sin ningún problema.

Desarrollo de la Interfaz de Usuario

El programa que actúa como User Interface consiste en un código en C que solicita un valor a pasarle al driver, lo controla, y muestra por consola los resultados arrojados.

Primero se intenta abrir el device driver como lectoescritura (`O_RDWR`). Si esto falla, el programa avisa el error y finaliza.

```
open("/dev/my_timer1", O_RDWR);
```

Luego se solicita al usuario que ingrese el modo de espera por el timer. Se incorporaron 3 modalidades:

- **Polling:** intenta leer el driver cada 500 ms hasta que el timer haya finalizado, pudiendo configurarlo con `POLLING_INTERVAL`.

  

- **Sleep:** el proceso de usuario duerme la cantidad de tiempo ingresada, y luego intenta leer el driver.

  

- **Interrupción:** El proceso lee el driver, se bloquea y espera ser despertado cuando el timer finalice.

  

  En los primeros dos escenarios el proceso duerme utilizando la función `usleep()`, que acepta valores en microsegundos, por lo que las constantes definidas se encuentran en esta unidad. Una vez seleccionado esto, se solicita el tiempo para configurar el timer. Se aceptan valores positivos, pudiendo configurar un valor máximo `MAX_VALUE`.

Cuando ya se tiene el número deseado, se lo traslada a una variable string, agregándole un caracter `\n` al final, como lo indica la documentación de `simple_strtol()`, y se lo envía al driver con:

```
write (fd, time_to_sleep_str, strlen(time_to_sleep_str));
```

La lectura se realiza con la función `read`:

```
ret = read(fd, recieve, STRING_LEN);
```

Makefile y uso

El Makefile utilizado para compilar el proyecto se encuentra en la carpeta `src/`. El primer driver tendrá el nombre `my_timer1.ko`.

Para ejecutar el proyecto se debe:

- Compilar el código fuente:
  - `$ make`
- Instalar el módulo en el Kernel
  - `$ sudo insmod my_timer1.ko`
- Cambiar los permisos del archivo en /dev para ejecutar el programa de interfaz de usuario sin permisos de administrador
  - `$ sudo chmod 666 /dev/my_timer1`
- Ejecutar el programa
  - `$ ./ui`

Bibliografía

El trabajo se realizó consultando las siguientes fuentes:

- https://www.ibm.com/developerworks/library/l-timers-list/ (https://www.ibm.com/developerworks/library/l-timers-list/)
- https://technologyasilearn.wordpress.com/2015/08/30/demystifying-linux-kernel-timers/ (https://technologyasilearn.wordpress.com/2015/08/30/demystifying-linux-kernel-timers/)
- Linux Device Drivers, Alessandro Rubini. 3ra Edición (http://shop.oreilly.com/product/9780596005900.do)

CppCheck

Al compilar y linkear, se genera un archivo donde se guardan los posibles errores y advertencias que encuentre el programa CppCheck al realizar el análisis estático del código. Este archivo se encuentra en:

Si desea más información, remítase a la documentación proporcionada, que se encuentra en la ruta `doc/html/index.html`