Práctica 1: Módulos del kernel Linux

Juan Carlos Sáez Alcaide

Índice

1 Objetivos

Los principales objetivos de esta práctica son los siguientes:

1. Familiarizarse con las siguientes abstracciones de Linux:
   - Módulos cargables
   - Sistema de ficheros /proc
   - Listas enlazadas del kernel
   - Gestión básica de memoria dinámica en el kernel
2. Afrontar las dificultades de la programación en espacio de kernel
3. Conocer los fundamentos de la depuración en Linux mediante trazado con *Ftrace*

La mayor parte de las prácticas de este curso constan de uno o varios proyectos de desarrollo que se describen en la sección *Desarrollo de la Práctica* (sección 3 en este guión). Como fase preparatoria para el proyecto (o proyectos) es preciso realizar previamente una serie de ejercicios sencillos que se describen a continuación.

2 Ejercicios

Ejercicio 1

Muchas funciones de la API del kernel Linux, como por ejemplo `alloc_chrdev_region()` o la propia función de carga de un módulo del kernel, devuelven un número entero que indica si se produjo o no un error al ejecutar la función. El valor cero indica que la función se ha ejecutado con éxito. El retorno de un número negativo refleja que se ha

producido un error, que se codifica con el propio valor de retorno (número comprendido entre -1 y -511). La definición de estos códigos de error (como números positivos) se encuentra en los siguientes ficheros de cabecera del kernel:

- `<uapi/asm-generic/errno-base.h>` (errores clásicos)
- `<uapi/asm-generic/errno.h>`

Consulta los citados ficheros de cabecera usando el buscador de las fuentes del kernel *Elixir Bootlin*, usando el siguiente enlace: https://elixir.bootlin.com/linux/v6.2.16/source. Estos ficheros se encuentran en el directorio `include/uapi/asm-generic`

Modifica alguno de los módulos del kernel de ejemplo para que la función de carga devuelva uno de estos errores (p.ej., `return -EINVAL;`) ¿Qué sucede al intentar cargar el módulo cuando esta función devuelve un número negativo? ¿Es posible descargar el módulo a continuación con `rmmod`?

Ejercicio 2

Estudiar el mecanismo de paso de parámetros a módulos del kernel, que se ilustra en el módulo de ejemplo `Hello5`. Para más información sobre el paso de parámetros, se han de consultar las siguientes fuentes:

1. Fichero de cabecera `<linux/moduleparam.h>`
2. Sección 4.5 de "The Linux Kernel Module Programming Guide"

Ejercicio 3

Estudiar la implementación del módulo de ejemplo "Clipboard", que exporta una entrada /proc. Al cargar/descargar el módulo se creará/eliminará el fichero especial `/proc/clipboard`, que puede emplearse como un portapapeles (*clipboard*) del sistema.

Nota: Antes de comenzar a estudiar el código del ejemplo se recomienda cargar el módulo y realizar escrituras y lecturas sucesivas sobre `/proc/clipboard` (con `echo <cadena> > /proc/clipboard` y `cat /proc/clipboard`). Esto permitirá averiguar rápidamente la funcionalidad del módulo `Clipboard`.

Ejercicio 4 (Introducción a Ftrace)

Ftrace es una herramienta que permite realizar depuración e inspección del funcionamiento del kernel Linux mediante trazado (*tracing*). Fundamentalmente, se utiliza realizando lecturas y escrituras en un conjunto de entradas en el sistema de ficheros *debugfs*. Estas entradas se encuentran bajo `/sys/kernel/debug/tracing` y sólo son accesibles directamente por el usuario *root* .

La herramienta está construida de forma modular, y su funcionalidad está encapsulada en un conjunto de *tracers*, como `nop`, `function` y `function_graph`.

En este breve tutorial analizaremos la funcionalidad más básica de Ftrace mediante una introducción al uso de los *tracers* más sencillos.

Entradas básicas de ftrace

Las entradas de Ftrace sólo estarán accesibles en un sistema si se cumplen los siguientes requisitos:

1. El kernel en ejecución ha sido compilado con soporte de ftrace (`CONFIG_FTRACE=y`)
2. Y `debugfs` está montado (en muchas distribuciones de Linux, como Debian, se monta automáticamente durante el proceso de arranque).
    - En caso de que no estuviera montado se ha de emplear el siguiente comando:

```
$ mount -t debugfs nodev /sys/kernel/debug
```

Comenzaremos accediendo al directorio de Ftrace como `root`:

```
kernel@debian:~$ sudo -i
[sudo] password for kernel:
root@debian:~$ cd /sys/kernel/debug/tracing
root@debian:/sys/kernel/debug/tracing$ ls
available_events            options                 stack_trace
available_filter_functions  per_cpu                 stack_trace_filter
available_tracers           printk_formats          timestamp_mode
buffer_percent              README                  trace
buffer_size_kb              saved_cmdlines          trace_clock
buffer_total_size_kb        saved_cmdlines_size     trace_marker
current_tracer              saved_tgids             trace_marker_raw
dynamic_events              set_event               trace_options
dyn_ftrace_total_info       set_event_notrace_pid   trace_pipe
enabled_functions           set_event_pid           trace_stat
error_log                   set_ftrace_filter       tracing_cpumask
events                      set_ftrace_notrace      tracing_max_latency
free_buffer                 set_ftrace_notrace_pid  tracing_on
function_profile_enabled    set_ftrace_pid          tracing_thresh
instances                   set_graph_function      uprobe_events
kprobe_events               set_graph_notrace       uprobe_profile
kprobe_profile              snapshot
max_graph_depth             stack_max_size
```

La siguiente tabla describe el propósito de las entradas básicas de Ftrace:

Entrada	Descripción
`tracing_on`	Permite activar/desactivar ftrace o consultar estado actual. Escribir la cadena "1" para activar ftrace o "0" para desactivarlo.
`trace`	Al leer de esta entrada se muestran los mensajes almacenados en los *buffers* de `ftrace` (un *buffer* por CPU)
`trace_pipe`	Similar a `trace`, pero además los buffers se vacían al mostrar su contenido (semántica productor/consumidor)
`available_tracers`	Lista el conjunto de *tracers* disponibles
`current_tracer`	Permite consultar/modificar el *tracer* activo leyendo/escribiendo en la entrada
`available_filters`	Lista el conjunto de funciones del kernel o de los módulos cargados que pueden "filtrarse" al usar el tracer `function`
`set_ftrace_filter`	Permite establecer la función (o funciones) para las que ftrace insertará un mensaje en el buffer cuando éstas sean invocadas.

`nop` tracer y `trace_printk()`

El *tracer* por defecto de Ftrace es `nop`. Captura únicamente los mensajes que el kernel o los módulos imprimen con la función `trace_printk()`. Esta función se usa de forma similar a `printf()`, pero con la salvedad de que mensaje impreso se almacena en un buffer interno de `ftrace`.

```
/* Defined at <linux/kernel.h> */

#define trace_printk(fmt, ...)              \
do {                                \
    char _______STR[] = __stringify((__VA_ARGS__)); \
    if (sizeof(_______STR) > 3)         \
        do_trace_printk(fmt, ##__VA_ARGS__);    \
```

```
    else                            \
        trace_puts(fmt);            \
} while (0)
```

La implementación de `trace_printk()` es mucho más eficiente que la de `printk()`, por lo que su uso es más aconsejable que `printk()` para la depuración del kernel. Además, si ftrace está desactivado, no tiene efecto (modo *silencioso*) .

Para ilustrar el uso de `trace_printk()` realizaremos una modificación en el módulo de ejemplo `Clipboard`. El objetivo es hacer que este módulo muestre un mensaje con `trace_printk()` y capturar dicho mensaje con `ftrace`. La modificación consiste en (1) incluir el fichero de cabecera `<linux/ftrace.h>` y (2) añadir una llamada a `trace_printk()` al final de la función `clipboard_write()`:

```
#include <linux/vmalloc.h>
#include <asm-generic/uaccess.h>
#include <linux/ftrace.h>
...
static ssize_t clipboard_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
  ...

  clipboard[len] = '\0'; /* Add the `\0' */
  *off+=len;            /* Update the file position indicator */

  trace_printk("Current value of clipboard: %s\n",clipboard);

  return len;
}
...
```

Ahora procederemos a compilar y cargar el módulo del kernel:

```
kernel@debian:~/LIN/Modulos/Clipboard$ make
make -C /lib/modules/6.2.16-lin/build M=/home/kernel/LIN/Modulos/Clipboard modules
make[1]: se entra en el directorio /usr/src/linux-headers-6.2.16-lin
  CC [M]  /home/kernel/LIN/Modulos/Clipboard/clipboard.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/kernel/LIN/Modulos/Clipboard/clipboard.mod.o
  LD [M]  /home/kernel/LIN/Modulos/Clipboard/clipboard.ko
make[1]: se sale del directorio /usr/src/linux-headers-6.2.16-lin
kernel@debian:~/LIN/Modulos/Clipboard$ sudo insmod clipboard.ko
[sudo] password for kernel:
kernel@debian:~/LIN/Modulos/Clipboard$
```

Para ver el efecto de las modificaciones realizadas abriremos dos ventanas de terminal. En la primera ventana, en la que iniciaremos una sesión como *root*, nos aseguraremos de que `ftrace` y el *tracer* `nop` están activos, y realizaremos una lectura del fichero `trace_pipe` con `cat`(lectura bloqueante). En la segunda ventana, donde no es necesario iniciar sesión como *root*, escribiremos la cadenas “Test” y “Something” (una detrás de la otra) al fichero especial `/proc/clipboard`. Las acciones realizadas en la segunda ventana de terminal harán que se muestren mensajes por la primera (salida de `cat trace_pipe`) .

Terminal 1

```
root@debian:/sys/kernel/debug/tracing$ cat current_tracer
nop
root@debian:/sys/kernel/debug/tracing$ cat tracing_on
1
root@debian:/sys/kernel/debug/tracing$ cat trace_pipe
            bash-16182  [000] ....  1065.269409: clipboard_write:
```

```
            Current value of clipboard: Test

            bash-16182  [000] ....  1100.023458: clipboard_write:
            Current value of clipboard: Something
```

Terminal 2

```
kernel@debian:~/LIN/Modulos/Clipboard$ echo "Test" > /proc/clipboard
kernel@debian:~/LIN/Modulos/Clipboard$ echo "Something" > /proc/clipboard
kernel@debian:~/LIN/Modulos/Clipboard$
```

Tracer `function`

El *tracer* `function` de Ftrace imprime automáticamente un mensaje en el buffer de ftrace cuando se ejecuta cierta función del kernel. Esto permite ver qué funciones núcleo se invocan sin modificar el código del kernel o de un módulo cargable que desee estudiarse o depurarse.

El *tracer* `function` soporta la creación de filtros de funciones. Para crear un filtro se ha de escribir el nombre o nombres de las funciones a trazar en la entrada `set_ftrace_filter` de Ftrace. El listado de funciones que pueden seleccionarse se puede obtener leyendo del fichero especial `available_filter_functions`. Cabe destacar que por defecto no hay ningún filtro configurado para el l *tracer* `function`, por lo que el comportamiento predeterminado es trazar todas las funciones del kernel. Como esto introduce mucha sobrecarga en el sistema, la creación de un filtro *ad-hoc* es la opción recomendada. Además durante la configuración de dicho filtro es aconsejable desactivar temporalmente ftrace mediante`echo 0 > tracing_on`.

Para ilustrar el uso del *tracer* `function` configuraremos Ftrace para que imprima un mensaje cuándo se invoca la función `clipboard_read()` del módulo de ejemplo `Clipboard`. Esto no requiere modificar el código del módulo del kernel. Para configurar el *tracer* `function` como deseamos será necesario seguir los siguientes pasos, ejecutando como *root* los comandos que se muestran a continuación (se asume que el directorio de trabajo es `/sys/kernel/debug/tracing`:

1. Desactivar temporalmente Ftrace

   ```
   $ echo 0 > tracing_on
   ```

2. Activar function tracer y comprobar que se activó correctamente:

```
$ echo function > current_tracer ; cat current_tracer
function
```

3. Preparar filtro de ftrace para trazar las invocaciones de la función `clipboard_read()`

   ```
   $ echo clipboard_read > set_ftrace_filter
   ```

4. Reactivar ftrace

   ```
   $ echo 1 > tracing_on
   ```

Una vez configurado el *tracer* `function` procederemos a abrir dos ventantas de terminal. En una de ellas (como *root*) procederemos a leer de la entrada `trace_pipe` de Ftrace con `cat`. En la segunda ventana, ejecutaremos `cat /proc/clipboard`:

Terminal 1

```
root@debian:/sys/kernel/debug/tracing$ cat trace_pipe
    cat-16406  [000] ....  3166.842845: clipboard_read <-proc_reg_read
    cat-16406  [000] ....  3166.844485: clipboard_read <-proc_reg_read
```

Terminal 2

```
kernel@debian:~/LIN/Modulos/Clipboard$ cat /proc/clipboard
Something
kernel@debian:~/LIN/Modulos/Clipboard$
```

Como se puede observar `clipboard_read()` se invoca dos veces al ejecutar `cat /proc/clipboard`. ¿A qué se debe esto?

3 Desarrollo de la práctica

En esta práctica se ha de implementar un módulo del kernel `modlist.c` que gestione una lista enlazada de enteros, empleando las siguientes definiciones globales:

```c
struct list_head mylist; /* Nodo fantasma (cabecera) de la lista enlazada */

/* Estructura que representa los nodos de la lista */
struct list_item {
    int data;
  struct list_head links;
};
```

Cuando el módulo se cargue/descargue se creará/eliminará automáticamente el fichero especial `/proc/modlist`. Esta entrada /proc permitirá insertar, eliminar y consultar los elementos de la lista desde espacio de usuario (leyendo o escribiendo en dicho fichero).

La memoria asociada a los nodos de la lista debe gestionarse de forma dinámica empleando `kmalloc()` y `kfree()`. Al descargar el módulo, éste ha de ocuparse de borrar y liberar la memoria de todos los elementos de la lista, si la lista no está vacía.

Para realizar modificaciones de la lista enlazada el usuario escribirá cadenas de caracteres (comandos) con un formato específico, como se indica a continuación:

1. Inserción de número (ejemplo 10) al final de la lista:
   - `echo add 10 > /proc/modlist`
2. Eliminación de número de la lista (número 7 en el ejemplo):
   - `echo remove 7 > /proc/modlist`
   - *Borra todas las ocurrencias de ese elemento en la lista*
3. Borrado de todos los elementos de la lista:
   - `echo cleanup > /proc/modlist`

Para la implementación de estos "comandos" en la *write* callback de la entrada `/proc/modlist` se aconseja utilizar la función `sscanf()`. Para más infomación sobre esta función se han de consultar las páginas de manual: `man 3 sscanf`.

Finalmente el módulo del kernel permitirá mostrar los elementos de la lista leyendo de `/proc/modlist` con el comando `cat`.

3.1 Ejemplo de ejecución

```
# Módulo del kernel se supone ya compilado, procedemos a cargarlo
kernel@debian$ sudo insmod modlist.ko

# Inicialmente la lista está vacía.
# No se ha de mostrar nada al ejecutar cat por primera vez
kernel@debian$ cat /proc/modlist
kernel@debian$
```

```
## Inserción de distintos números en la lista y consulta de su contenido
kernel@debian$ echo add 10 >  /proc/modlist
kernel@debian$ cat /proc/modlist
10
kernel@debian$ echo add 4 >  /proc/modlist
kernel@debian$ echo add 4 >  /proc/modlist
kernel@debian$ cat /proc/modlist
10
4
4
kernel@debian$ echo add 2 >  /proc/modlist
kernel@debian$ echo add 5 >  /proc/modlist
kernel@debian$ cat /proc/modlist
10
4
4
2
5

## Eliminación de elemento de la lista
kernel@debian$ echo remove 4 > /proc/modlist
kernel@debian$ cat /proc/modlist
10
2
5

## Ejecución de operación de borrado
kernel@debian$ echo cleanup > /proc/modlist
kernel@debian$ cat /proc/modlist
```

3.2 Partes opcionales de la práctica

Parte opcional 1. Modificar el módulo de la práctica para que la lista gestionada sea de cadenas de caracteres, y su memoria se reserve con `kmalloc()`. Para esta variante de la práctica se aconseja la inclusión de sentencias de compilación condicional para mantener en un mismo fichero fuente las implementaciones del módulo con lista de enteros (básica) y lista de cadenas de caracteres (opcional), como en el siguiente ejemplo:

```
#ifdef PARTE_OPCIONAL
... Fragmento de código específico para lista de cadenas de caracteres ...
#else
... Fragmento de código específico para lista de enteros...
#endif
```

Emplear este tipo de sentencias de preprocesador permite escoger la versión de la práctica a compilar, definiendo el símbolo `PARTE_OPCIONAL` u omitiendolo en tiempo de compilación. Los símbolos de preprocesador se especifican con la opción -D, y, en el caso de los módulos cargables, a través de la variable de entorno EXTRA_CFLAGS que reconoce el sistema de construcción de módulos. Así por ejemplo, para activar la compilación de la parte opcional debería usarse el siguiente comando.

```
make EXTRA_CFLAGS=-DPARTE_OPCIONAL
```

Parte opcional 2. Reimplementar la read callback de la entrada /proc empleando `seq_printf()` mediante la abstracción de `seq_files` de Linux. **Nota:** Se aconseja mantener un contador con el número de elementos de la lista.

El uso de `seq_file` en Linux ofrece las siguientes ventajas:

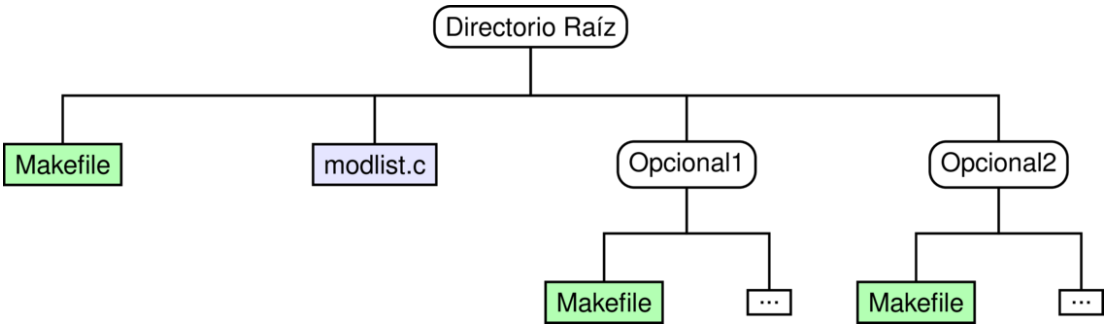
- Soporte especial para recorrido de secuencias de elementos
- Gestión implícita del buffer de usuario
  - No requiere usar `copy_to_user()` ni `sprintf()`

Para más información sobre el uso de `seq_files` pueden consultarse las siguientes fuentes:

1. Documentación sobre `seq_files` para versiones de Linux anteriores a la v5.7.x
   - Sección 10.2 "Professional Linux Kernel Architecture"
   - "The seq_file Interface" (kernel docs)
2. Ejemplo de implementación para Linux 6.2.16 de entrada `/proc/cpuinfo`
   - https://elixir.bootlin.com/linux/v6.2.16/source/fs/proc/cpuinfo.c
   - https://elixir.bootlin.com/linux/v6.2.16/source/arch/x86/kernel/cpu/proc.c#L172
   - Con respecto a versiones previas del kernel (documentación del punto 1), encontramos los siguientes cambios en la API:
     1. Uso de `struct proc_ops` en lugar de `struct file_operations` para entrada `/proc`
     2. Se define la callback `read_iter` en lugar de `read` en `struct proc_ops`

3.3 Entrega de la práctica

La práctica debe entregarse a través del Campus Virtual en un fichero comprimido (.tar.gz o .zip) con la siguiente estructura de directorios:



Consideraciones adicionales:

1. Es aconsejable mostrar el funcionamiento antes de hacer la entrega
2. Las partes opcionales solo puntuarán si la entrega se realiza dentro del plazo establecido