Práctica sobre el sistema de señales de Linux. Sistemas Operativos (2005-2006).

Junio, 2006

Abstract. Este guión incluye la documentación relativa a las tareas a realizar y las herramientas recomendadas.

1. Introducción.

El trabajo a realizar se divide en dos partes. La primera se refiere a la utilización de las señales en la programación de procesos de usuario, mientras que el objetivo de la segunda es acceder al tratamiento de las señales entregadas a los procesos desde el kernel. Para la realización de las tareas se facilitan un conjunto de herramientas que también se describen en este guión.

2. Descripción de las tareas a realizar.

2.1. Utilización de señales en procesos de usuario.

Construir un programa en C que realice las siguientes funciones [Wall, 2001]:

- Habrá un proceso padre que lance un conjunto de NCHILD procesos hijo mediante la llamada del sistema fork(). Cada proceso hijo tendrá un identificador propio definido por el usuario. El proceso padre guardará en un archivo la relación de procesos generados mediante el par (identificador propio, PID del proceso). Puesto que nos vamos a concentrar en la utilización de las señales más que en una aplicación concreta,

para esta práctica, implementa cada proceso hijo como un bucle infinito:

```
while(1);
```

- El proceso padre deberá interceptar la señal SIGINT (Ctrl-C en una terminal) mediante la llamada del sistema sigaction(). La función de interceptación enviará una señal SIGTERM a cada uno de los procesos hijo y acabará con NCHILD llamadas a la llamada del sistema wait() para asegurarnos que los recursos utilizados por los procesos hijo son liberados. Más abajo veremos otra forma de gestionar desde el proceso padre, la liberación de recursos utilizados por los procesos hijo que terminan.

- La ejecución de los procesos hijo será controlada por un archivo de configuración. En dicho archivo se establecerá qué procesos deben estar suspendido y cuales en ejecución. Para ello cada linea de este archivo consistirá en el par (identificador propio del proceso, 1 (suspender) o 0 (pasar a ejecución)). El proceso padre leerá este archivo y colocará a los procesos hijo en la situación especificada, mediante el envío de las señales SIGSTOP y SIGCONT. Este proceso lo realizará al comienzo de la ejecución y mediante una función de interceptación de la señal SIGHUP.

- Añadir una tercera opción al archivo de configuración. Si un proceso tiene asignado el número de opción 2, pasará a estar suspendido y bloqueará la señal SIGCONT. Una vez que el proceso no tenga la opción 2 establecida, SIGCONT quedará desbloqueada.

- Añadir al proceso padre la posibilidad de monitorizar la terminación de los procesos hijo. Cuando un proceso hijo termine, el proceso padre dará cuenta de este evento en un fichero, incluyendo además información sobre si la terminacin se produjo de forma normal o porque el proceso hijo recibió una señal. Pista: estudia el funcionamiento de la señal SIGCHLD.

2.1.1. Ayudas para la realización de esta parte de la práctica.

Estos son algunos consejos que pueden ayudarte en la realización de la práctica.

- Las siguientes llamadas del sistema pueden serte útiles: fork(), getpid(), sigaction(), kill(), wait(), waitpid(), fopen(),feos(),fprintf(),fscanf().

- El pid de los procesos puede ser consultado desde la shell mediante:

  ```
  ps -a -f -H
  ```

- El estado de un proceso, , las señales bloqueadas y las señales capturadas lo puedes obtener desde la shell mediante:

  ```
  cat /proc/$(PID)/status
  ```

  siendo PID el identificador del proceso que se desea consultar.

- Ten cuidado con la forma en la que se generan los procesos hijo. Separa bien el código que debe ser ejecutado exclusivamente por el padre y el código exclusivo de los hijos, especialmente si ese código incluye la generación de procesos hijo.

- Veamos un código ejemplo de la utilización de fork():

```
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>


int main(void)
{
int i;
pid_t child;
FILE *fid;

// imprimimos pid del proceso padre

printf("Proceso principal %d\n",getpid());
// asignamos los identificadores...
child=0;
child=fork();
if(child==-1)
{
perror("Error en fork");
exit(EXIT_FAILURE);
```

```
}
if(child==0)
{

//Proceso hijo
//Este es el código que ejecutaría el proceso hijo
while(1);

}
else
{

// en el proceso padre child es diferente de 0 así que este es el código
// ejecutado en el proceso padre
....


exit(EXIT_SUCCESS);
}
}
```

- Realicen pruebas exhaustivas para comprobar la correcta funcionalidad del programa. No olviden que debe funcionar incluso cuando un tercer proceso pudiera terminar con los procesos hijo.

Programación en el kernel.

En esta parte de la práctica utilizaremos un kernel de Linux 2.6, modificado para exportar la tabla de llamadas del sistema y monitorizar la interceptación de las señales.

2.1.2. Herramientas básicas.

El trabajo a realizar se desarrollará como un módulo para el kernel de Linux 2.6. Vamos a utilizar las siguientes herramientas:

- Un sistema de virtualización con una distribución Linux "mini" preparada con el kernel 2.6.16. Los sistemas de virtualización son especialmente adecuados para este trabajo pues permiten realizar la programación del kernel sin "riesgos" y con mayor comodidad. Usaremos

el sistema qemu (www.qemu.org) en sus versiones 0.7 y superiores. El kernel para el sistema de pruebas debe ser compilado con la misma versión de gcc que la usada en el sistema de desarrollo. El sistema que se suministra con la práctica fue compilado con gcc 3.4.4.

- El sistema de desarrolo estará formado por gcc 3.4.4, make y el árbol del fuente compilado del kernel 2.6.16 que se ejecuta en el sistema de pruebas.
- El fuente del kernel 2.6.16 fue parcheado para exportar la variable sys_call_table, que es la tabla de llamadas del sistema. Esto facilita bastante la interceptación posterior. Los kernel previos a la versión 2.6 exportaban esta variable por defecto, pero esto ya no es así.

Utilizaremos el sistema de módulos del kernel para añadir nuestro código y poder interceptar la llamada del sistema. A continuación se muestra el esqueleto de un módulo simple [Corbet et al., 2005]:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
static char *quien="mundo";
static int howmany=1;
extern int sys_call_table[];
MODULE_LICENSE("Dual BSD/GPL");
module_param(howmany,int,S_IRUGO);
module_param(quien,charp, S_IRUGO);
static int hello_init(void)
{
int i;
for(i=0;i<howmany;i++)
printk(KERN_ALERT "Hola, %s\n",quien);
return 0;
}

static void hello_exit(void)
{


printk(KERN_ALERT "Adios\n");
}
```

```
module_init(hello_init);
module_exit(hello_exit);
```

Este código representa un módulo simple que admite dos parámetros. Una vez compilado adecuádamente, como usuario root en el sistema de pruebas lo cargaríamos del siguiente modo:

```
insmod hello.ko quien="yo" howmanytimes=3
```

Al cargarse el módulo se ejecutaría la función de inicialización, declarada como tal en modulo_init(). En el ejemplo se trataría de la función hello_init.

Para descargar el módulo, haríamos:

```
rmmod hello.ko
```

En este caso se ejecuta la función de descarga, declarada como tal en modulo_exit(). En el ejemplo se trataría de la función hello_exit.

Es importante ver en el listado además como se realiza la declaración de parámetros con module_param.

La compilación de este módulo se debe realizar con make, mediante la definición de un Makefile. A continuación se muestra un ejemplo, de un Makefile preparado para compilar el módulo hello.ko (este Makefile no vale para versiones anteriores del kernel):

```
ifeq ($(KERNELRELEASE),)

    KERNELDIR ?= /mnt/cdrom/linux-source-2.6.16
    # The current directory is passed to sub-makes as argument
    PWD := $(shell pwd)

modules:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

modules_install:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install

clean:
rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions

.PHONY: modules modules_install clean
```

```
else
    # called from kernel build system: just declare what our modules are
    obj-m := hello.o

endif
```

En este Makefile, es esencial establecer en KERNELDIR el path al árbol de la fuente del kernel compilado. Cuando make lee este Makefile, se encuentra en primer lugar que KERNELRELEASE no ha sido definido y entra por el if a realizar el objetivo modules del primer make. Antes de realizarlo se cambia al directorio KERNELDIR (opción -C de make) y sigue haciendo make a partir de ahí (de esta manera carga las variables de los Makefile del fuente del kernel). Cuando ha terminado vuelve al path del directorio donde nos encontramos (opción -M) y lee nuevamente el Makefile. La diferencia es que esta vez KERNELRELEASE ya está definido, por lo que pasa a realizar la parte correspondiente al else. En este caso se trata de construir un módulo a partir de hello.o. Este Makefile lo podemos usar como patrón con el kernel Linux 2.6, pero no con kernels anteriores [Corbet et al., 2005].

2.1.3. Utilización del sistema de pruebas.

Nuestro sistema de pruebas, virtualiza un ordenador Pc Pentium II con un mini sistema Linux 2.6.16. El kernel ha sido configurado para soportar varios sistemas de ficheros: ext2, ext3, vfat entre otros. El fichero qcim representa la imagen de un disco de almacenamiento con el sistema de ficheros ext2 y el mini sistema Linux. Para construirlo, se usó el mini sistema Linux ttylinux 5.0, y se adaptó para nuestro sistema de pruebas. Se trata de un sistema reducido, por lo que no esperen encontrar todas las opciones habituales en las herramientas Linux.

El siguiente comando arranca el sistema virtual:

```
qemu -hda qcim -hdb fat:./modulos -snapshot
```

En este caso estamos arrancando el sistema virtual con un /dev/hda master representado por qcim, y un sistema de ficheros vfat en /dev/hdb1, que es una copia del directorio ./modulos.

Una vez en el sistema de pruebas, entramos como root (usuario root, password root) y montamos el sistema vfat:

```
mount -t vfat /dev/hdb1 /mnt
```

De esta manera podemos desarrollar y compilar fuera del sistema de pruebas, colocar los módulos y archivos ejecutables en un directorio del ordenador y luego arrancar el sistema de pruebas para probarlo.

La opción -snapshot en el comando de arranque del sistema de pruebas también es importante. Con esta orden, los directorios y ficheros que creemos en las unidades de almacenamiento del sistema de pruebas, o las modificaciones que realicemos realmente no son permanentes, ya que son virtualizadas. Una combinación de teclas puede realizar el volcado y hacerlas permanentes, pero si esto no se realiza, estaremos seguros que qcim no ha sido modificado, por lo que si rearrancamos el sistema de pruebas estará siempre en la misma situación. Esto es lo recomendable, si se está utilizando el sitema virtual para pruebas.

2.1.4. Interceptación de llamadas del sistema.

La interceptación de llamadas del sistema es una técnica sencila si disponemos de la tabla de llamadas del sistema exportada. En el siguiente ejemplo, se muestra como podemos dejar sin funcionalidad la llamada que permite crear directorios en un sistema de archivos:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
extern void* sys_call_table[];
int (*orig_mkdir)(const char *path); /*la llamada original*/


int hacked_mkdir(const char *path)
{
 return 0;
}
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
printk(KERN_ALERT "Hola\n");
orig_mkdir=sys_call_table[39];
sys_call_table[39]=hacked_mkdir;
return 0;
}
```

```
static void hello_exit(void)
{
sys_call_table[39]=orig_mkdir;

printk(KERN_ALERT "Adios\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Como vemos la técnica es sencilla. Basta sustituir en la función de carga del módulo, el puntero a la función de servicio en la tabla de llamadas del sistema por la función que queremos realice la sustitución. En la función de descarga, volvemos a dejar las cosas como estaban. La posición de la llamada del sistema en la tabla se puede obtener desde el código fuente del kernel.

En nuestro caso, no queremos realizar una sustitución sino ampliar la funcionalidad de la llamada del sistema. Si quisiéramos ampliar la funcionalidad del mkdir, imprimiendo además un mensaje de información, haríamos lo siguiente:

```
int hacked_mkdir(const char *path)
{
 printk("Estamos creando un directorio\n");
 orig_mkdir(path);
}
```

Es importante destacar que se trata de una técnica útil para estudiar el funcionamiento del kernel, pero no es una práctica recomendable en general. De hecho, como se ha comentado más arriba, a partir del kernel 2.6, ya no se exporta la tabla de llamadas del sistema, precisamente para evitar esta práctica.

Monitorización mediante un módulo del kernel de señales interceptadas.

Como hemos explicado en teoría, la interceptación o captura de una señal implica una serie de acciones realizadas por el kernel que finaliza con la llamada del sistema sigreturn(). Esta llamada se encarga de restaurar el contexto del usuario para poder seguir el curso del proceso de usuario que se estaba ejecutando cuando tuvo lugar la entrega de la señal [Bar, 2000].

Por ello, si se realiza la interceptación de esta llamada es posible, monitorizar las capturas de señales en nuestro sistema. El objetivo en esta práctica

es producir un mensaje por pantalla cada vez que se produzca una interceptación. En dicho mensaje se dará cuenta del número de interceptaciones contabilizadas, del PID del proceso que está realizando la interceptación y de las señales de dicho proceso que han sido configuradas por éste para ser interceptadas.

Veamos ahora algunas ayudas útiles para realizar el trabajo:

- Puedes utilizar el programa desarrollado en la primera parte de la práctica para probar el módulo.
- Realiza el trabajo comenzando por conseguir sacar un mensaje cada vez que se detecte la llamada a la rutina de servicio sys_sigreturn.
- La información a la que puedes acceder desde el módulo está en el descriptor del proceso, al que puedes acceder mediante la variable current (por ejemplo,

  ```
  current->pid
  ```

  da el pid del proceso). Para averiguar, qué partes de esta estructura te proporcionan la información necesaria puedes utilizar la documentación que menciona en la bibliografía o el código fuente del kernel: por ejemplo, el archivo signal.c en archi386kernel. Fíjate como se usa el puntero current, para obtener información sobre el proceso. Es relevante especialmente la estructura del descriptor sighand, así como la estructura ka_sigaction [Bover y Cesati, 2005], [Love, 2005]
- Realiza un script que te permita probar el módulo de forma sistemática.

3. Evaluación de la práctica.

Los objetivos establecidos son obligatorios. Las mejoras que se proponen, como trabajo adicional optativo son:

- Utilizar el sistema /proc para mostrar la información monitorizada de la interceptación de señales.
- Explorar la realización de la monitorización, sin la interceptación de la llamada del sistema. Para ello, hay que revisar las funciones del kernel involucradas en la entrega de la señal que son exportadas.

Estas son propuestas, pero se valorarán todas las mejoras que presenten.

4. Bibliografía

Referencias

[Bar, 2000] Bar, M. (2000). The linux signals handling model. Linux Journal,2000.

[Bover y Cesati, 2005] Bover, D. y Cesati, M. (2005). *Understanding de Linux Kernel, 3rd edition.* O'Reilly.

[Corbet et al., 2005] Corbet, J., Rubini, A., y Kroah-Hartman, G. (2005). *Drivers en Linux.* O'Reilly.

[Love, 2005] Love, M. (2005). The linux kernel process management. capitulo del libro Linux Kernel Development, 2nd Edition.

[Wall, 2001] Wall, K. (2001). *Programación en Linux.* Prentice-Hall.