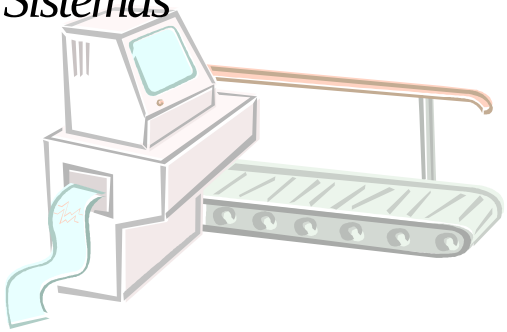
Eliseo García García
e-mail: eliseo.garcia@uah.es
Despacho: E-232
Departamento de Automática

Practica 7

Manejadores de dispositivos en GNU/Linux

Desarrollo de Aplicaciones para Sistemas Industriales



Manejadores de Dispositivos en GNU/Linux

Introducción

Linux es un sistema operativo con enfoque monolítico en donde todas las partes que lo componen están enlazadas en una única imagen que se carga y ejecuta al arrancar el sistema. Por lo tanto, el kernel de Linux está constituido por múltiples componentes que no todos son necesarios para los usuarios en todo momento. Por ello, es por lo que al crearse el kernel, Linux solicita al usuario que especifique los elementos que desea incluir en el kernel. Evidentemente, el objetivo de esta operación es insertar únicamente los gestores necesarios en función de la configuración de la máquina y de su uso. De este modo, el tamaño del kernel es lo más reducido posible en función de la máquina. Cuanto menor es el tamaño del kernel, queda más memoria disponible para el usuario. Además, el arranque de una máquina con un kernel especialmente adaptado, es decir, únicamente con los gestores de dispositivos que posee la máquina, es mucho más rápido. Sin embargo, cualquier modificación del kernel, como la inclusión o la supresión de un gestor de dispositivo, un sistema de archivos, implica la recompilación del kernel. Esto era cierto en las primeras versiones de Linux, hasta la implementación de los *módulos cargables*.

En efecto, a partir de la versión 2.0, Linux incluye una funcionalidad importante que evita que sea necesario recompilar el kernel cada vez que se le quiera añadir cierta funcionalidad. Esta funcionalidad puede añadirse con los "módulos cargables". Los módulos cargables permiten la incorporación "en caliente" de nueva funcionalidad al sistema, es decir, sin parar el sistema o reiniciar la máquina, y por supuesto, sin tener que recompilar el kernel. Estos elementos permiten la inserción y extracción del código en el nucleo que gestiona dispositivos hardware que se pueden ir conectando o desconectando en cada momento.

El objetivo de los *módulos cargables* es, pues, generar en un primer momento un kernel mínimo, y cargar los gestores de manera dinámica en función de las necesidades. Esto permite, en un momento dado, tener un kernel "extendido". Sin embargo, hay que destacar que el kernel debe estar configurado para que gestione los módulos (Loadable module support). Este sistema de módulos cargables existe también en otros sistemas operativos UNIX como por ejemplo Solaris, pero bajo una forma diferente.

Usando esta técnida, se permite al superusuario cargar y descargar manualmente los módulos según las necesidades. Esto es bastante pesada de manipular porque toda operación debe efectuarse manualmente. El usuario normal no tiene derechos suficientes para efectuar esta operación.

Preparación del sistema para el uso de módulos cargables

Como se desprende de lo anterior, los modulos cargables permiten la creación de código al que se le permitirá la ejecución en estrecha coordinación con el sistema operativo. Por lo tanto, es necesario que para la creación de estos módulos necesitemos infomacion respecto de estructuras, funciones, variables, … que utiliza el Sistema Operativo, y que nos la pone a nuestra disposición por si la necesitamos para crear nuestro codigo. Esta información es la que se conoce como Modulos del Sistema Operativo, y Linux nos la pone a nuestra disposición a través de un fichero conocido como Modules.symvers.

Algunas distribuciones oficiales ofrecen junto con el Sistema Operativo el fichero anterior para poder crear los modulos cargables. En nuestro caso, en la distribución de Ubuntu instalada en el laboratorio

tenemos dicho fichero en el directorio /usr/src/, que contine los modulos del sistema operativo del equipo huésped, es decir, del sistema operativo de nuestro PC del laboratorio.

En caso de que no dispongamos del fichero (como es nuestro caso para el Sistema Operativo Raspbian que se usa en la Raspberry Pi), se puede crear realizando una compilación del código del Sistema Operativo para obtener la imagen del Sistema Operativo y los módulos que utiliza. El sistema operativo Linux es un sistema Open Source, lo que significa que el código fuente está disponible para que cualquier usuario lo pueda obtener, por lo que podemos acceder a dicho código, modificarlo si queremos que se comporte de forma distinta a la de la distribución oficial, y realizar la compilacion para crear el ejecutable del sistema operativo (fichero .img que veíamos en la practica anterior).

A continuación, se va a proceder a realizar este proceso. Lo haremos, tal como vimos en la practica anterior, mediante un entorno de compilacion cruzada, usando las herramientas instaladas previamente.

En primer lugar, decir que para la compilación del Sistema Operativo a instalar en la Raspberry Pi se necesita tanto el compilador de la maquina objeto (entorno de compilacion cruzada), asi como el compilador de la maquina huésped. En la distribución del Ubuntu instalado en las maquinas de laboratorio, se encuentran instalados las versiones 9, 10 y 11 del compilador gcc, como podemos comprobar listando el directorio /usr/bin/

```
$ ls –latr /usr/bin/gcc*
```

Estas versiones del compilador no son compatibles con el uso del entorno de compilacion cruzada. Por lo tanto, antes de comenzar el proceso de compilacion, deberemos instalar una versión anterior del compilador del PC, como por ejemplo la versión 4.8. Esta versión la podemos obtener en la dirección http://mirrors.kernel.org/ubuntu/pool/universe/g/gcc-4.8/ . A continuación, se muestran los comandos para la instalación de la versión 4.8 del compilador en nuestra maquina, bajándonos los ficheros mediante el comando wget y realizando su instalación mediante el comando sudo apt install.

```
$ mkdir CompiladorGccAntiguo/
$ cd CompiladorGccAntiguo/
$ mkdir install_g++-4.8
$ cd install_g++-4.8/
$ sudo apt update
$ wget http://mirrors.kernel.org/ubuntu/pool/universe/g/gcc-4.8/g++-4.8_4.8.5-4ubuntu8_amd64.deb
$ wget http://mirrors.kernel.org/ubuntu/pool/universe/g/gcc-4.8/libstdc++-4.8-dev_4.8.5-
4ubuntu8_amd64.deb
$ wget http://mirrors.kernel.org/ubuntu/pool/universe/g/gcc-4.8/gcc-4.8-base_4.8.5-
4ubuntu8_amd64.deb
$ wget http://mirrors.kernel.org/ubuntu/pool/universe/g/gcc-4.8/gcc-4.8_4.8.5-4ubuntu8_amd64.deb
$ wget http://mirrors.kernel.org/ubuntu/pool/universe/g/gcc-4.8/libgcc-4.8-dev_4.8.5-
4ubuntu8_amd64.deb
$ wget http://mirrors.kernel.org/ubuntu/pool/universe/g/gcc-4.8/cpp-4.8_4.8.5-4ubuntu8_amd64.deb
$ wget http://mirrors.kernel.org/ubuntu/pool/universe/g/gcc-4.8/libasan0_4.8.5-
4ubuntu8_amd64.deb
$ sudo apt install ./gcc-4.8_4.8.5-4ubuntu8_amd64.deb ./gcc-4.8-base_4.8.5-4ubuntu8_amd64.deb
./libstdc++-4.8-dev_4.8.5-4ubuntu8_amd64.deb ./cpp-4.8_4.8.5-4ubuntu8_amd64.deb ./libgcc-4.8-
dev_4.8.5-4ubuntu8_amd64.deb ./libasan0_4.8.5-4ubuntu8_amd64.deb ./g++-4.8_4.8.5-
4ubuntu8_amd64.deb
```

Una vez ejecutados los comandos, se observa que en el directorio /usr/bin ha aparecido la versión 4.8 del compilador que anteriormente no estaba.

```
$ ls –latr /usr/bin/gcc*
```

Esto confirma que hay una nueva version de gcc instalado en el Sistema. No es la que se encuentra activa (véase el enlace del fichero gcc en la salida del comando anterior). Se debe especificar que queremos que se utilice la versión 4.8 del compilador gcc cuando se invoque el comando. Eso se consigue mediante la ejecución del siguiente comando:

```
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 80
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 80
```

```
$ sudo update-alternatives --config gcc
```

De esta forma, cuando invoquemos el gcc dentro de nuestro equipo para la compilacion del programa, se estara utilizando la version 4.8. Compruebese la versión de cada uno de los compiladores, el del sistema huésped y el del sistema objetivo, mediante los comandos

```
$ gcc –v
$ ${CCPREFIX}gcc -v
```

Una vez configuradas las herramientas necesarias para la compilacion, pasamos a realizarla.

Cree un directorio para almacenar los ficheros de Raspberry Pi mencionados, y a continuación, ejecutando el siguiente comando en dicho directorio, los ficheros fuente del kernel serán copiados desde el servidor al subdirectorio "linux" (aproximadamente 2 Gbytes)

```
$ mkdir raspberry
$ cd raspberry
$ git clone https://github.com/raspberrypi/linux.git
```

Compruébese que se ha creado un directorio linux dentro de la carpeta raspberrypi, y que en su interior existen un conjunto de carpetas y ficheros que contienen ficheros con codigo fuente, configuraciones, Makefiles,… , que son necesarios para construir el Sistem Operativo.

Con el comando anterior, se ha bajado a nuestro equipo los fuentes de la ultima versión de Linux. Podemos conocer la rama git de la versión que tenemos en el equipo introduciendo el comando

$ git status

Lamentablemente, esta versión no está soportada por nuestra tarjeta Raspberry Pi b+, por lo que tenemos que cambiarlos por los ficheros de una versión anterior (nosotros vamos a utilizar la versión 4.1 de linux). Para obtenerla, dentro del directorio raspberry/linux, ejecutaremos el siguiente comando

$ git switch rpi-4.1.y

$ git pull origin rpi-4.1.y

$ git status

De esta forma, vemos que la rama ha cambiado y el código que tenemos es el de la versión 4.1.

De la practica anterior, se tendrá ya instaladas las herramientas para realizar la compilación de programas para la plataforma Raspberry Pi desde nuestra maquina de desarrollo. Por lo tanto, únicamente fijaremos la variable de entorno CCPREFIX al directorio que contiene la versión de compilador que vamos a usar y, a continuación testearemos su correcto funcionamiento invocando el comando de compilación del este compilador cruzado (gcc) utilizando dicho prefijo:

```
$     export     CCPREFIX=~/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-
gnueabif-raspbian/bin/arm-linux-gnueabihf-
$  ${CCPREFIX}gcc –v
```

Tras la ejecución de este último comando, en la última línea debería aparecer un mensaje con el número de versión de la herramienta que se encuentra instalada. En el punto en el que nos encontramos, tenemos todos los ficheros fuentes en el subdirectorio "linux", las herramientas de compilación instaladas en el directorio "tools", y el sistema configurado para que pueda acceder a dichas herramientas. El siguiente paso, antes de la compilación, es configurar el modo en el que se realizará dicha compilación. Para eso, vamos a partir de la configuración que tiene el kernel que actualmente esta ejecutándose en el sistema objetivo. Dicha configuración se encuentra en un fichero comprimido del sistema objetivo, el fichero

/proc/config.gz. Para obtener dicho fichero, accederemos a nuestro sistema Raspberry y lo copiaremos en el subdirectorio "Linux" del sistema anfitrión.

En función de la versión de Raspbian que se este utilizando, es posible que en el sistema objetivo no este creado el fichero /proc/config.gz. En versiones anteriores se encontraba por defecto, mientras que en las versiones más actuales no se encuentra por defecto, pero se puede generar.

Para conocer en concreto nuestra situación, nos conectamos al sistema objetivo mediante:

```
$ ssh usuario@ip_raspberry
```

y listamos el contenido del directorio /proc

```
$ ls /proc
```

En el caso de que no se encuentre el fichero config.gz en ese directorio, habría que crearlo mediante el comando

```
$ sudo modprobe configs
```

y podremos comprobar como el fichero ya se encuentra en el directorio. Abandonamos el sistema obejtivo mediante

```
$ exit
```

y una vez en el sistema anfitrión, procedemos a copiar dicho archivo en el directorio adecuado.

```
$ cd ~/raspberry/linux
$ scp usuario@ip_raspberry:/proc/config.gz .
```

Con estos comandos, en el subdirectorio linux tendremos un fichero comprimido config.gz que contiene la configuración del kernel que se esta ejecutando actualmente en el sistema objetivo. Procedemos a continuación a su descompresión.

```
$ gunzip –c config.gz > .config
```

El fichero .config es el fichero donde se configura los parámetros del kernel a construir. Se puede editar de forma manual, pero existe un comando que permite modificar dicho fichero para que recoja las características configuradas para la nueva compilacion partiendo de la configuración que actualmente tiene:

```
$ ARCH=arm CROSS_COMPILE=${CCPREFIX} make oldconfig
```

Mediante este comando ejecutado en el subdirectorio linux impulsamos la nueva configuración del kernel partiendo de la configuración que se encuentra actualmente en el fichero .config (que es la configuración que previamente obtuvimos de nuestro sistema Raspberry). En nuestro caso, como vamos a construir un Sistema Operativo básico, negamos todas las opciones de configuración opcionales que nos ofrece la ejecución del comando anterior.

Por otro lado, resulta importante destacar que en el comando anterior hemos fijado el parámetro ARCH a arm (arquitectura de Raspberry Pi). Si no se hubiera especificado, el kernel se habría configurado para su construcción en el sistema anfitrión, cuya arqueitectura es x86, y no se habría podido ejecutar el kernel en el sistema objetivo.

El siguiente paso que vamos a llevar a cabo es configurar el comportamiento de nuestro sistema respecto a los símbolos de depuración. En primer lugar reseteamos dicho comportamiento en el archivo .config,

```
$ grep –v DEBUG_INFO  < .config  >newconfig
$ mv newconfig .config
```

Y a continuación, volvemos a construir nuestro fichero de configuración .config partiendo de él mismo con la característica de la depuración ya incluida. Para tal fin ejecutamos el comando que ya conocemos

```
$ ARCH=arm CROSS_COMPILE=${CCPREFIX} make oldconfig
```

En el momento en el que nos solicite la habilitacion de DEBUG_INFO, accederemos a su habilitacion, y cuando nos solicite la habilitación DEBUG_INFO_REDUCED procederemos a su negación para deshabilitar dicha opción.

Con todo lo realizado anteriormente, ya nos encontramos en disposición de realizar la construcción del nuevo kernel. Para tal fin, ejecutaremos el comando

```
$ ARCH=arm CROSS_COMPILE=${CCPREFIX} make
```

La construcción del kernel puede llevar días si la hubiéramos realizado directamente sobre la plataforma Raspberry Pi sin compilador cruzado, pero en nuestro sistema llevará un tiempo inferior a una hora. Aun así, esta construcción puede ser acelerada en caso de que nuestra maquina contenga varios procesadores o cores mediante la ejecución paralela del comando anterior, para lo cual se debería añadir el modificador –j<numerodeprocesadores>.

Una vez que la construcción del kernel ha concluido, si todo se ha realizado de forma correcta, un nuevo fichero ejecutable vmlinux debe estar en nuestro directorio. Es el fichero resultado de la compilacion del nucleo, a partir del cual se creará el fichero .img imagen del Sistema Operativo.

El siguiente paso es la construcción del directorio que contenga los modulos que Raspberry Pi cargará cuando los necesite durante su ejecución normal. Para tal fin, se puede ejecutar el siguiente comando que realiza la copia de los modulos en el directorio "raspberry/modules"

```
$ ARCH=arm CROSS_COMPILE=${CCPREFIX} INSTALL_MOD_PATH=../modules make
modules_install
```

Ya tenemos el sistema operativo y los modulos construidos en nuestro sistema anfitrion y unicamente quedaría llevarlos a nuestro sistema objetivo. Para eso, en primer lugar, respecto al kernel, creamos una imagen del kernel descomprimida y lo llevamos al directorio temporal de nuestra Raspberry Pi

```
$ cd ~/raspberry/tools/mkimage
```

Para crear la imagen su usa un script de la versión 2 de Python (extensión .py), que por defecto no viene instalado en la distribución del sistema operativo huésped, por lo que procedemos a su instalación con la herramienta apt-get

```
$ sudo apt-get install python2
```

Y, finalmente, realizamos creación de la imagen del kernel y su copya en el sistema objetivo

```
$ ./imagetool-uncompressed.py ../../linux/arch/arm/boot/zImage
$ scp kernel.img usuario@ip_raspberry:/tmp/.
```

Nos encargamos ahora de los modulos. Nos situamos en el subdirectorio donde se han creado, los comprimimos en un fichero tar y los copiamos en el sistema objetivo

```
$ cd ~/raspberry/tools/modules
$ tar czf modules.tgz *
$ scp modules.tgz usuario@ip_raspberry:/tmp/.
```

Ya tenemos en el directorio temporal tanto la imagen del kernel como un fichero comprimido con todos los modulos, con lo que el siguiente paso sería instalar estos ficheros en el lugar conveniente para su uso en el posterior arranque. Con este fin, pasamos a conectarnos en la Raspberry Pi:

```
$ ssh usuario@ip_raspberry
```

Una vez conectados mediante el comando anterior, todo lo que ejecutemos se llevara a cabo en la plataforma objetivo, es decir, en nuestra Raspberry Pi. Instalamos el kernel

```
$ sudo cp /boot/kernel.img /tmp/kernel_ant.img
$ sudo mv /tmp/kernel.img /boot/.
```

Y finalmente descomprimirmos los modulos y eliminamos los ficheros temporales

```
$ cd /
$ sudo tar xzf /tmp/modules.tgz
$ rm /tmp/modules.tgz
```

Podemos comprobar que los modulos se encuentran bien instalados listando el directorio /lib/modules donde estarán los modulos del nuevo kernel. Pues en este momento, para que entre en ejecución el nuevo sistema construido, únicamente quedaría resetear y, tras dicho reset, este nuevo kernel ya seria el que entraría en ejecución.

```
$ sudo shutdown –r now
```

Podemos comprobar que el nuevo kernel esta en ejecución en la Raspberry Pi conectándonos a esta plataforma y preguntando por la versión del nuevo kernel. Dicha versión debe coincidir con la versión de los ficheros fuente del nucleo que nos bajamos al inicio y los cuales utilizamos para la construcción del kernel.

```
$ ssh usuario@ip_raspberry
$ uname -r
```

Como construimos nuestro kernel con informacion de depuración, podríamos ser capaces de depurar cualquier error que el sistema operativo en ejecución pudiera producir.

Funcionamiento de los módulos cargables

Linux proporciona principalmente tres ordenes que permiten manipular los módulos: *insmod, lsmod* y *rmmod.*

La orden *insmod* permite efectuar la carga del módulo. *lsmod* se limita a mostrar el contenido del archivo */proc/modules*. *rmmod* descarga el módulo deseado. La orden *lsmod* indica no sólo el nombre de cada uno de los módulos cargados en memoria, sino también el número de páginas de memoria ocupadas por el

módulo, así como el número de procesos que utilizan este módulo. Una página de memoria ocupa 4 KB, por lo que resulta fácil calcular la memoria economizada cuando estos módulos no están cargados.

El sistema de carga dinámica permite automatizar las cargas de los distintos módulos en función de la demanda. Su implementación precisa la activación en la compilación de la opción CONFIG_KERNELD y los IPC System V. El demonio *kerneld* utiliza las colas de mensajes para comunicarse con el kernel: la carga o descarga de un módulo la realiza *kerneld,* pero las órdenes se envían desde el kernel mediante una cola de mensajes especial. La implementación de esta técnica en una máquina implica el lanzamiento del programa *kerneld* al arrancar el sistema. También es necesario efectuar ciertas operaciones para construir la lista de módulos cargables instalados. Los módulos cargables dinámicamente necesitan dos programas al inicializarse la máquina: (1) *depmod,* que permite generar un archivo de dependencias basado en los símbolos encontrados en el conjunto de los módulos. Estas dependencias se usarán posteriormente en la segunda orden; (2) *modprobe,* que permite cargar un módulo o un grupo de módulos pero también cargar los módulos básicos necesarios para el correcto inicio de la máquina (como NFS, etc.).

La estructura del kernel de Linux, si no contase con la filosofía de los modulos cargables, daria lugar a un sistema poco flexible, ya que cualquier funcionalidad que se le quisiera añadir al kernel del sistema requeriría una recompilación completa del mismo. Aún así, la filosofía de *open source* (fuentes abiertos) hace Linux mucho más flexible que otros sistemas operativos en la que los fuentes no están disponibles. No obstante, la recompilación total del kernel puede resultar engorrosa en las fases de desarrollo de nuevos manejadores de dispositivo, ampliaciones no oficiales del kernel, etc.

Esta limitación desapareció con la incorporación, en la versión 2.0 de Linux, del soporte para la carga dinámica de módulos en el kernel. Esta nueva característica permite la "incorporación en caliente" de nuevo código al kernel del sistema operativo, sin necesidad de reinicializar el sistema.

Los módulos son "trozos de sistema operativo", en forma de archivos objeto (.ko), que se pueden insertar y extraer en tiempo de ejecución (forma dinámica). Dichos archivos .ko se pueden obtener directamente como resultado de la compilación de un archivo .c *(gcc −c prog.c)*, o como la unión de varios archivos .ko enlazados *(ld −r f1.o f2.o)*. La única característica especial que deben tener estos archivos, es la de incorporar las funciones *init_module* y *cleanup_module*. Más adelante veremos su utilidad.

Una vez desarrollado un módulo e insertado en el kernel, su código pasa a ser parte del propio kernel, y por lo tanto se ejecuta en el modo supervisor del procesador (nivel de privilegio 0 en la arquitectura i386), con acceso a todas las funciones del kernel, a las funciones exportadas por módulos previamente insertados, y a todo el hardware de la máquina sin restricciones.

La única diferencia con código enlazado en el kernel es la posibilidad de extraer el módulo una vez ha realizado su labor o ha dejado de ser útil, liberando así todos los recursos utilizados. Naturalmente, para insertar un módulo en el kernel se debe hacer en modo superusuario (supervisor), puesto que si un usuario normal pudiera insertar módulos, esto se convertiría en un serio y evidente problema de seguridad.

Dado que el código de un módulo puede utilizar cualquier función del kernel, pero no ha sido enlazado en tiempo de compilación con él, las referencias a las funciones del kernel no están resueltas. Por tanto, cuando se inserta un módulo en el kernel se deben seguir los siguientes pasos:

- Obtener las referencias a funciones ofrecidas por el módulo.
- Incorporar dichas referencias al kernel, como referencias temporales, que desaparecerán con la extracción del módulo.
- Resolver las referencias a las funciones no resueltas en el módulo, ya sean llamadas a funciones del kernel, como a llamadas a funciones de otros módulos.
- Insertar el módulo en la zona de memoria correspondiente al kernel.
- Finalmente, invocar a la función *xxx_init* registrada a través de la macro *module_init()* como punto de entrada del nuevo módulo.

La extracción de un módulo del kernel se realiza mediante una secuencia similar a la anterior, pero en orden inverso, donde antes de extraer el módulo se invoca a la función *xxx_exit*, registrada a través de la macro *module_exit()* como punto de salida del módulo.

Programación de los módulos cargables

La función *xxx_init* nos van a permitir inicializar el módulo al insertarlo en el kernel (equivaldría a la función *main* de un programa en C). Por otro lado, *xxx_exit* se usará para liberar los recursos utilizados cuando se vaya a extraer.

A continuación, vamos a estudiar la forma de generar el código correspondiente al módulo. Naturalmente, el kernel deberá tener compilado el soporte para módulos, o no podremos insertarlos. Un módulo lo generaremos a partir de un archivo fuente C, del estilo del siguiente.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
int n = 1;
module_param (n, int, 0644);
static int ejemplo_init(void)
{
printk("Entrando. n = %d\n", n);
return 0;
}
static void ejemplo_exit(void)
{
printk("Saliendo.\n");
}
module init(ejemplo init);
module_exit(ejemplo_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sr. X");
```

Este programa podría estar almacenado en el archivo ejemplo.c. Se debe proceder a continuación a la compilación. Ésta no se realiza como un fichero fuente habitual, sino que se compila de forma distinta por dos motivos, el resultado será un módulo y, por otro lado, estamos compilando para otra arquitectura.

Compilación del módulo

Para realizar la compilación se debe realizar un archivo Makefile como el siguiente, que indica el nombre del módulo a generar (observe que tiene extensión .o y no .ko):

```
obj-m += ejemplo.o

all:
   make -C /…/directorio_del_codigo_del_kernel M=${PWD} modules

clean:
   make -C /…/directorio_del_codigo_del_kernel M=${PWD} clean
```

A continuación compilaríamos el programa empleando el programa make en el directorio donde se encuentre el fichero ejemplo.c.

Es necesario indicar que el directorio directorio_del_codigo_del_kernel/… en el ejemplo anterior deberá adecuarse al lugar exacto donde tengamos descomprimido el código fuente del kernel, que además deberá estar configurado (es decir, en esa ubicación deberá estar el fichero Module.symvers, que contiene los símbolos del kernel que podrán ser utilizados por el módulo a crear). Esta sentencia genera el archivo ejemplo.ko, que ya es un módulo insertable. Además, resulta imprescindible asegurarse que la versión del código fuente del kernel que se encuentra en ese directorio coincide exactamente con la versión del kernel que se está ejecutando en el sistema donde se va a utilizar el nuevo módulo creado, ya que en caso contrario puede haber problemas de compatibilidad. Para conocer la versión del kernel que se ejecuta en un sistema Linux basta con ejecutar el comando

```
$ uname -r
```

En el programa ejemplo se emplea la función printk(). Esta función es similar a printf(), salvo que en lugar de imprimir en la salida estándar imprime en un buffer de mensajes del kernel o kernel ring buffer. En este buffer escribe normalmente el kernel todo tipo de errores, notificaciones o eventos que ocurren a nivel del kernel. Se puede ver su contenido ejecutando el comando dmesg, consultando /proc/kmsg o visualizando el fichero /var/log/messages. Por ejemplo podemos emplear la orden tail –f /var/log/messages para ver cómo progresa dicho buffer de forma constante. El manejo de los mensajes del kernel brinda otras posibilidades, algunas de ellas muy complejas. Una propiedad de interés es que se puede indicar la prioridad o la importancia del mensaje mediante una indicación con tres caracteres al principio de la cadena de la forma <n>, donde n es un número de 0 a 7 que indica el tipo de mensaje. Normalmente el núcleo se configura para que sólo se muestren por consola los mensajes de prioridad superior a 6.

- KERN_EMERG System is unusable
- KERN_ALERT Action must be taken immediately
- KERN_CRIT Critical conditions
- KERN_ERR Error conditions
- KERN_WARNING Warning conditions
- KERN_NOTICE Normal but significant condition
- KERN_INFO Informational
- KERN_DEBUG Debug−level messages

Normalmente, el kernel está configurado para mostrar por la consola activa los mensajes de prioridad superior a 6. (Los terminales gráficos no son consolas, a no ser que se hayan lanzado explícitamente como tales).

Compilación cruzada del módulo

En nuestro caso, estamos generando el código en una maquina host (el PC del laboratorio) que, como conocemos, no será la maquina donde se ejecutará el modulo cargable, sino que éste se empleará en el sistema objetivo Raspberry Pi. Es por eso, que al invocar a make, se deben activar las herramientas del compilador cruzado. Esto lo podemos conseguir modificando nuestro fichero Makefile para que se ejecute la herramienta make asociada al compilador cruzado.

```
obj-m += ejemplo.o

all:
   make ARCH=arm CROSS_COMPILE=${CCPREFIX} -C /home/usuario/…/dir_del_codigo_del_kernel_objetivo
M=${PWD} modules

clean:
   make -C /home/usuario/…/dir_del_codigo_del_kernel_objetivo M=${PWD} clean
```

Utilización de los módulos cargables

Como ya hemos comentado anteriormente, para insertar o extraer módulos del kernel debemos tener permisos de superusuario o supervisor. La inserción (carga) de un módulo se lleva a cabo mediante la orden *insmod*, que realizará todas las acciones comentadas antes para insertar el código en el kernel. Ejecute, desde una consola, la siguiente orden y observe qué ocurre en el buffer de salida del kernel tras la inserción:

```
$ sudo insmod ejemplo.ko
```

Observe qué ocurre en el buffer de salida del kernel tras la inserción. Acabamos de instalar ejemplo y ejecutar su función *init_module()*. Si se le pasa a *insmod* un nombre de archivo sin ruta ni extensión, se busca en los directorios estándar (ver *insmod(8)*). Para ver los módulos que están cargados y cierta información sobre ellos, emplearemos la orden *lsmod*. Y finalmente, para extraer un módulo del kernel, emplearemos *rmmod*.

```
$ sudo rmmod ejemplo
```

Para pasarle parámetros a un módulo, hay que asignar valores a las variables globales que hemos declarado como parámetros en la macro *module_param*. Esta macro recibe como primer argumento la variable, como segundo argumento el tipo de dicha variable y finalmente como tercer argumento los permisos del archivo correspondiente en *sysfs*, aspecto que de momento obviaremos.

Por ejemplo, en el caso que nos ocupa podríamos cargar el módulo de la siguiente forma.

```
$ sudo insmod ejemplo.ko n=4
```

Empleando la orden *modinfo ejemplo.ko* podemos averiguar los datos necesarios sobre los parámetros del módulo.

Dependencia e interacción entre modulos cargables

A continuación se muestra la **implementación** de dos módulos cargables en el kernel de Linux, para observar ciertas particularidades de las dependencias entre módulos y su interacción. Estos módulos, que denominaremos *acumulador* y *cliente*, deberán atender al siguiente comportamiento:

- Ambos módulos (*acumulador* y *cliente*) han de mostrar cuando los insertemos (carga) o extraigamos (descarga), un mensaje informativo indicando el instante de la inserción y de la extracción. Podemos obtener el instante actual usando la función del kernel *do_gettimeofday*(definida en el fichero kernel/time/timekeeping.c) o, adicionalmente, consultando la variable *xtime* del kernel. El tipo de variable que usa la función *do_gettimeofday* es *struct timeval* (que almacena el número de segundos desde el uno de enero de 1970) y está definida en include/linux/time.h.

- El módulo *acumulador* tendrá definida una función *void acumular(int i)*, que vaya sumando a una variable global del módulo el valor que se le pase como parámetro. Asimismo deberá ofrecer una función *int llevamos(void)* que permita consultar el valor de dicha variable global. La variable empieza valiendo cero, y cuando se extraiga el módulo, se mostrará el valor acumulado. Es importante que en el módulo acumulador se exporten las funciones para que puedan ser utilizadas por otros módulos, ya que por defecto todas las funciones que definamos serán privadas del módulo. Para ello, se empleará la macro *EXPORT_SYMBOL(simbolo)*.

- El módulo *cliente*, al ser insertado, llamará a la función *acumular()* del módulo acumulador, pasándole un valor igual al que le pasemos al módulo cliente al insertarlo. Cuando se extraiga este módulo, mostrará cuánto llevamos acumulado hasta el momento, haciendo uso de la función *llevamos()*. Es decir, el módulo *cliente*, al ser insertado (cargado), debe llamar a la función *acumular()* del módulo

acumulador, pasándole un valor a acumular igual al parámetro que le pasemos al módulo *cliente* al insertarlo (cargarlo). El módulo *cliente*, al ser extraído, debe llamar a la función *llevamos()* del módulo *acumulador* e imprimir el resultado acumulado en su mensaje de salida.

Módulo acumulador

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

int suma ;
void acumulador(int i){
printk("suma antes de acumular= %d\n", suma);
suma += i ;
printk("suma despues de acumular= %d\n", suma);
}
void consultar(void){
printk("suma = %d\n",suma);
}

static int   init acumulador init(void){
suma = 0 ;
acumulador(8);
return 0;
}
static void __exit acumulador_exit(void){
consultar() ;
}
EXPORT_SYMBOL(acumulador);
EXPORT SYMBOL(consultar);
module_init(acumulador_init);
module_exit(acumulador_exit);
MODULE LICENSE("GPL");
MODULE_AUTHOR("Sr X.");
```

Módulo cliente

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

extern void acumulador(int);
extern void consultar(void);

static int __init cliente_init(void){
acumulador(8);
return 0;
}

static void   exit cliente exit(void){
consultar();
}

module_init(cliente_init);
module exit(cliente exit);
MODULE LICENSE("GPL");
MODULE_AUTHOR("Sr X.");
```

Como tenemos dos módulos será necesario un Makefile que permita construir ambos módulos. En este caso se especificará de la siguiente manera:

```
obj-m := modulo1.o
obj-m += modulo2.o
```

Para insertarlos en el sistema huésped:

```
obj-m += acumulador.o
obj-m += cliente.o
```

```
all:
   make -C /…/dir del codigo del kernel M=${PWD} modules

clean:
   make -C /…/dir_del_codigo_del_kernel M=${PWD} clean
```

Para insertarlos en el sistema objetivo

```
obj-m += acumulador.o
obj-m += cliente.o

all:
   make ARCH=arm CROSS_COMPILE=${CCPREFIX} -C /home/usuario/…/dir_del_codigo_del_kernel_objetivo
M=${PWD} modules

clean:
   make -C /home/usuario/…/dir_del_codigo_del_kernel_objetivo M=${PWD} clean
```

Compruebe que los módulos implementados se compilan correctamente. A continuación, inserte el módulo *acumulador*, y el módulo cliente varias veces, observando que el comportamiento es el correcto. Finalmente, extraiga el módulo *acumulador*, compruebe el mensaje y observe que el resultado final también es el correcto.

Después, observe qué sucede si:

- Intentamos insertar el *cliente* sin que esté el *acumulador* insertado.
- Intentamos extraer el acumulador estando el cliente insertado.

y razone lo ocurrido en ambos casos.

Use *lsmod* cada vez que inserte y extraiga los módulos, para asegurarse de que todo funciona correctamente. Observe también el estado del sistema cuando los módulos están insertados, empleando la misma orden.

Dispositivos

El uso de modulos cargables resulta muy útil a la hora de implementar el código que pueda gestionar elementos hardware que se puedan insertar y extraer en el sistema (dispositivos Plug and Play). De ese modo, cuando se detecte la inserción de un dispositivo se puede cargar el modulo cargable que gestiona su funcionalidad, poniendo el conjunto de funciones que lo manejan a disposición de aplicaciones que puedan interaccionar con el dispositivo.

Otra de las misiones del sistema operativo es abstraer a las aplicaciones del hardware al que nunca deberían acceder directamente, a excepción de las áreas de memoria mapeada en el espacio lógico de los procesos. El objetivo es poder administrar el hardware entre las aplicaciones.

Los sistemas estilo UNIX emplean un mecanismo de comunicación con el hardware que considera a los dispositivos como ficheros pertenecientes a un sistema de archivos. De esta forma se puede disponer de una interfaz normalizada que permite interactuar con ellos haciendo uso de un reducido número de funciones estándar. Además esto permite beneficiarse de algunas características interesantes de los archivos como el control de seguridad contra accesos no autorizados. Casi todos los SS.OO. modernos han adoptado algún método similar para acceder al hardware.

Una de las características más destacables en UNIX es que los dispositivos se encuentran visibles a los usuarios a través del directorio /dev. La comunicación con los dispositivos se realiza a través de controladores de dispositivo (device drivers). Estos son pequeños programas que se encargan de mostrarnos los dispositivos a través de una interfaz estándar en forma de llamadas al sistema.

La abstracción que ofrece UNIX de los dispositivos físicos conectados al ordenador es la de archivo especial de dispositivo. Los archivos especiales de dispositivo suelen localizarse en el directorio /dev. Es decir, Unix trata de manejar cualquier dispositivo como un fichero situado en el directorio /dev. Los archivos especiales (recordemos que en UNIX hay tres tipos de archivo: archivos normales o "regulares", archivos especiales y directorios) ocultan una funcionalidad especial bajo la apariencia de archivos convencionales, con su ruta de acceso, sus atributos, etc.

Existen dos tipos de dispositivos:

- **Dispositivos de tipo carácter.**- Todos aquellos cuyas lecturas o escrituras se pueden realizar mediante un flujo de bytes de datos serie. Ejemplos: puertos serie y paralelo, terminales, y tarjetas de sonido.
- **Dispositivos de modo bloque.**- Representan a dispositivos que realizan lecturas y escrituras en bloques indivisibles de tamaño fijo. Permiten el acceso aleatorio a cualquier dato. Una lectura exige que se recupere todo el bloque de información que la contiene. Una escritura parcial a cualquier parte de un bloque implica que se debe escribir todo el bloque, por lo que previamente debería haber sido leído. El ejemplo más característico es un controlador de disco.

Para listar los dispositivos soportados de forma nativa por el sistema, introducimos la orden:
$ ls -l /dev

No todas las entradas se corresponden con dispositivos reales, sino que son más bien puntos de entrada para dispositivos potenciales. Como se puede ver en la lista, a cada dispositivo le corresponden dos números: uno mayor y otro menor, que utiliza el S.O. para reconocerlos internamente.

Observando los resultados de la ejecución del comando ls −l /dev se pùede compruebar que el primer carácter de los atributos de cada archivo varia entre cinco posibilidades.

- "-" indica que es un archivo normal o regular
- "d" indica que es un archivo de tipo directorio
- "b" indica que es un archivo especial de dispositivo en modo bloque
- "c" indica que es un archivo especial de dispositivo en modo caracter.
- "l" indica que es un enlace a otro archivo

Un detalle importante a tener en cuenta es que el nombre de un archivo especial no influye para nada en su relación con uno u otro dispositivo físico. Por tanto, podemos copiar o renombrar cualquier archivo de este tipo sin ningún problema.

De hecho, estos archivos no contienen información ni ocupan espacio si los duplicamos. Los únicos datos útiles que contienen son el major number y el minor number, que identifican unívocamente el dispositivo que representan. En el listado producido por ls −l, las dos columnas anteriores a la fecha son el major y minor number.

- El major number indica el tipo de dispositivo (por ejemplo, 3 es un disco del primer IDE, 22 uno del segundo IDE, 2 es un disco flexible, etc.)
- El minor number indica un número de orden dentro de los dispositivos de cada tipo (p.ej., entre los discos de un IDE, el 0 es el primer disco, el 64 el segundo, el 1 es la primera partición del primer disco, el 2 la segunda, etc.).

Llamadas al sistema para el manejo de archivos

Los dispositivos pueden ser tratados como archivos ordinarios en muchos sentidos, al compartir las mismas llamadas al sistema básicas.

La biblioteca estándar de C proporciona funciones para que las aplicaciones puedan leer y escribir archivos. Tal es el caso de las conocidas `fopen()`, `fclose()`, `fread()`, y `fwrite()`, que sirven para: abrir o crear, cerrar, leer, y escribir archivos respectivamente.

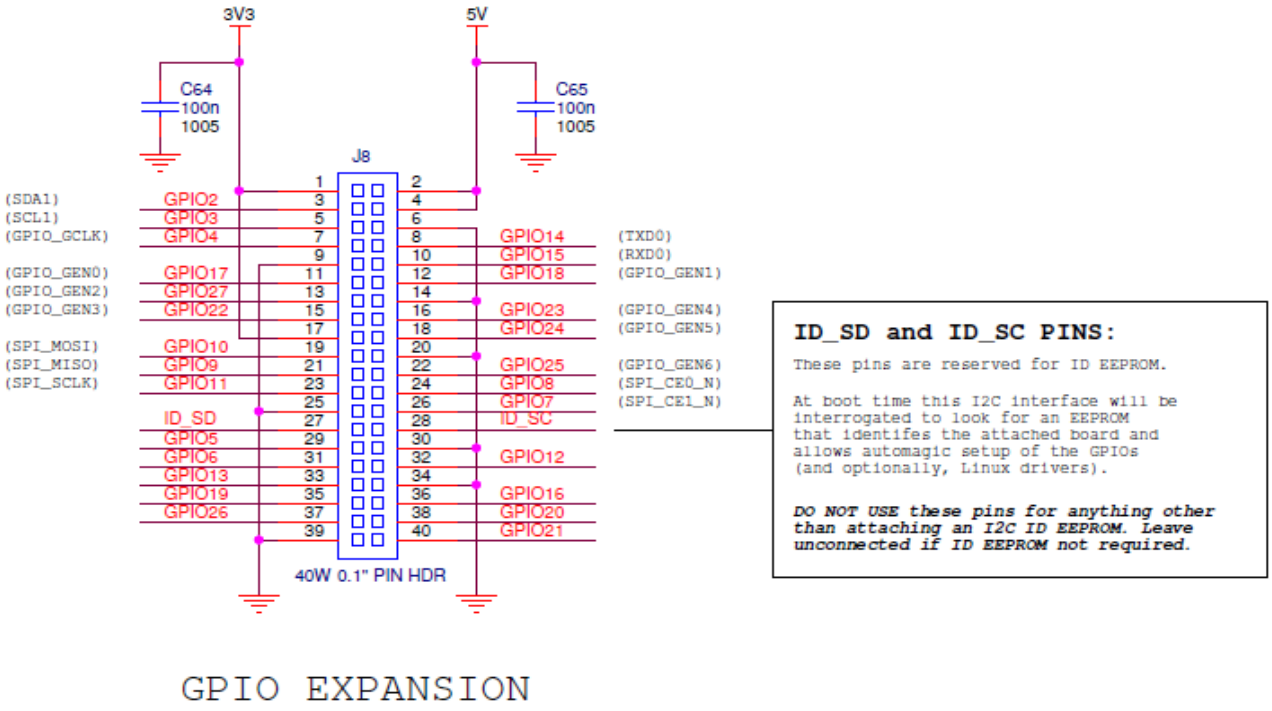
Los sistemas UNIX cuentan además con otras funciones de más bajo nivel con una funcionalidad equivalente. Se trata de las llamadas al sistema `open()`, `close()`, `read()`, y `write()`. Utilizan descriptores de dispositivo en lugar de punteros a estructuras de tipo FILE. Llamada al sistema equivalente a `fflush()` es `fsync()`.

Para poder hacer un seguimiento de las llamadas al sistema que realicen nuestras aplicaciones, podemos usar el programa `strace`.

```
$ strace ./programa_a_ejecutar
```

Dispositivo GPIO en Raspberry Pi

El modelo B+ de Raspberry Pi ofrece el conector J8 como puerto de entrada salida de propósito general. Este conector consta de 40 pines, de los cuales un total de 12 corresponden a alimentaciones y masas. Además, existe una serie de pines que comparten su uso como puertos GPIO con otras funcionalidades, de tal forma que en el caso de que la placa las utilice no se podrán destinar esos pines a GPIO. Así tenemos 5 pines que se usan para la comunicación SPI, 2 pines para la comunicación I2C, 3 pines para la comunicación mediante uart, 7 pines para manejar un display de 7 segmentos,… En dicho conector, únicamente se tienen 9 pines cuya única funcionalidad sea GPIO.



Como su nombre indica, GPIO son pines de propósito general que se pueden configurar como entrada ó salida, y cuyo valor puede ser controlado mediante software. Por defecto, un pin GPIO no tiene definido un propósito especial, y puede ser usado para:

- Fijar un nivel alto o nivel bajo en dicho pin, si se configura como salida

- Leer un nivel lógico digital, en caso de que se configure como entrada
- Recibir peticiones de interrupción, en caso de que se configure como entrada de interrupciones.

Siguiendo el esquema de creación de código para modulos cargables, hemos desarrollado el driver que maneja una de las patillas como GPIO (se muestra en el siguiente apartado). Como cualquier otro driver en linux, este intenta que las aplicaciones traten al dispositivo (en este caso, cada pin de GPIO) como un fichero, y se buscará que las aplicaciones realicen sobre él las operaciones que habitualmente se realizan a ficheros. Para eso, dentro del código del driver se detallan las acciones que deberán realizarse cuando las aplicaciones soliciten la apertura del dispositivo (.open = raspi_gpio_open) el cierre del dispositivo (.release = raspi_gpio_release), la lectura desde el dispositivo (.read = raspi_gpio_read) y la escritura en el dispositivo (.write = raspi_gpio_write). Observese esa estructura en el código desarrollado para el driver de la siguiente sección.

Ejemplo de un driver que maneja un pin

A continuación se muestra un ejemplo de un driver desarrollado mediante modulos cargables que permite el manejo del pin GPIO2 (patilla 3 del conector J8 de la plataforma Raspberry Pi Modelo B+). Obsérvese que, en la zona de definición de variables, se encuentra la sentencia #define NUMGPIO 2, que hace que todo el código anterior se aplique sobre la patilla GPIO2. Para realizar el mismo driver para otra patilla, únicamente se debería cambiar el valor del #define.

```c
/*
* raspi_gpio_driver.c - GPIO Linux device driver for Raspberry Pi B
*/
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <linux/gpio.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <uapi/asm-generic/errno-base.h>
#include <linux/string.h>
#include <linux/spinlock.h>
#include <linux/interrupt.h>
#include <linux/time.h>

/* User-defined macros */
#define DEVICE_NAME "raspi-gpio"
#define BUF SIZE 512
#define NUMGPIO 2

/* User-defined data types */
enum state {low, high};
enum direction {in, out};
/*
* struct raspi_gpio_dev - Gpio pin data structure
* @cdev: instance of struct cdev
* @pin: instance of struct gpio
* @state: logic state (low, high) of a GPIO pin
* @dir: direction of a GPIO pin
* @lock: used to protect atomic code section
*/
struct raspi gpio dev {
   struct cdev cdev;
   struct gpio pin;
   enum state state;
   enum direction dir;
   spinlock t lock;
};

/* Declaration of entry points */
static int raspi_gpio_open(struct inode *inode, struct file *filp);
static ssize_t raspi_gpio_read ( struct file *filp, char *buf, size_t count, loff_t *f_pos);
static  ssize t  raspi gpio write  (struct  file  *filp,  const  char  *buf,  size t  count,  loff t
*f_pos);
static int raspi_gpio_release(struct inode *inode, struct file *filp);

/* File operation structure */
static struct file_operations raspi_gpio_fops = {
   .owner = THIS MODULE,
   .open = raspi_gpio_open,
   .release = raspi_gpio_release,
   .read = raspi gpio read,
   .write = raspi_gpio_write,
};

/* Forward declaration of functions */
static int raspi_gpio_init(void);
```

```
static void raspi gpio exit(void);

/* Global varibles for GPIO driver */
struct raspi gpio dev raspi gpio devp;
static dev_t first;
static struct class *raspi gpio class;

/*
* raspi gpio open - Open GPIO device node in /dev
*
* This function open GPIO
*/

static int raspi_gpio_open (struct inode *inode, struct file *filp) {
struct raspi gpio dev *raspi gpio devp;
unsigned int gpio;

gpio = iminor(inode);
printk(KERN_INFO "GPIO[%d] opened\n", gpio);
raspi_gpio_devp = container_of(inode->i_cdev, struct raspi_gpio_dev,cdev);
filp->private data = raspi gpio devp;
return 0;
}

/*
* raspi_gpio_release - Release GPIO pin
*
* This functions releases GPIO resource when the device is
* last closed.
*/

static int raspi_gpio_release (struct inode *inode, struct file *filp)
{
unsigned int gpio;
struct raspi gpio dev *raspi gpio devp;

raspi_gpio_devp = container_of(inode->i_cdev, struct raspi_gpio_dev, cdev);
gpio = iminor(inode);
printk(KERN_INFO "Closing GPIO %d\n", gpio);
return 0;
}

/*
* raspi gpio read - Read the state of GPIO pins
*
* This functions allows to read the logic state of input GPIO pins
* and output GPIO pins. Since it multiple processes can read the
* logic state of the GPIO, spin lock is not used here.
*/

static ssize_t raspi_gpio_read ( struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
unsigned int gpio;
ssize_t retval;
char byte;

gpio = iminor(filp->f_path.dentry->d_inode);
for (retval = 0; retval < count; ++retval) {
   byte = '0' + gpio_get_value(gpio);
   if(put_user(byte, buf+retval))
          break;
   }
return retval;
```

```
}

/*
* raspi gpio write - Write to GPIO pin
*
* This function allows to set GPIO pin direction (input/out) and
* to set GPIO pin logic level (high/low)
* Set logic level (high/low) to an input GPIO pin is not permitted
* The command set for setting GPIO pins is as follows
* Command Description
* "out" Set GPIO direction to output via gpio_direction_ouput
* "in" Set GPIO direction to input via gpio direction input
* "1" Set GPIO pin logic level to high
* "0" Set GPIO pin logic level to low
*/
static ssize_t raspi_gpio_write ( struct file *filp,const char *buf,size_t count,loff_t *f_pos)
{
unsigned int gpio, len = 0, value = 0;
char kbuf[BUF_SIZE];
struct raspi_gpio_dev *raspi_gpio_devp = filp->private_data;
unsigned long flags;

gpio = iminor(filp->f_path.dentry->d_inode);
len = count < BUF SIZE ? count-1 : BUF SIZE-1;
if(copy_from_user(kbuf, buf, len) != 0)
   return -EFAULT;
kbuf[len] = '\0';
printk(KERN_INFO "Request from user: %s\n", kbuf);
// Check the content of kbuf and set GPIO pin accordingly
if (strcmp(kbuf, "out") == 0) {
   printk(KERN_ALERT "gpio[%d] direction set to ouput\n", gpio);
   if (raspi_gpio_devp->dir != out) {
           spin_lock_irqsave(&raspi_gpio_devp->lock, flags);
           gpio_direction_output(gpio, low);
           raspi gpio devp->dir = out;
           raspi_gpio_devp->state = low;
           spin_unlock_irqrestore(&raspi_gpio_devp->lock, flags);
   }
} else if (strcmp(kbuf, "in") == 0) {
   if (raspi gpio devp->dir != in) {
           printk(KERN INFO "Set gpio[%d] direction: input\n", gpio);
           spin_lock_irqsave(&raspi_gpio_devp->lock, flags);
           gpio direction input(gpio);
           raspi gpio devp->dir = in;
           spin_unlock_irqrestore(&raspi_gpio_devp->lock, flags);
   }
} else if ((strcmp(kbuf, "1") == 0) || (strcmp(kbuf, "0") == 0)) {
   sscanf(kbuf, "%d", &value);
   if (raspi gpio devp->dir == in) {
           printk("Cannot set GPIO %d, direction: input\n", gpio);
           return -EPERM;
   }
   if (raspi_gpio_devp->dir == out) {
           if (value > 0) {
                   spin lock irqsave(&raspi gpio devp->lock, flags);
                   gpio_set_value(gpio, high);
                   raspi_gpio_devp->state = high;
                   spin unlock irqrestore(&raspi gpio devp->lock, flags);
           } else {
                   spin_lock_irqsave(&raspi_gpio_devp->lock, flags);
                   gpio set value(gpio, low);
                   raspi_gpio_devp->state = low;
                   spin_unlock_irqrestore(&raspi_gpio_devp->lock, flags);
```

```
            }
    }
} else {
    printk(KERN ERR "Invalid value\n");
    return -EINVAL;
}
*f pos += count;
return count;
}
/*
* raspi_gpio_init - Initialize GPIO device driver
*
* This function performs the following tasks:
* Dynamically register a character device major
* Create "raspi-gpio" class
* Claim GPIO resource
* Initialize the per-device data structure raspi_gpio_dev
* Initialize spin lock used for synchronization
* Register character device to the kernel
* Create device nodes to expose GPIO resource
*/

static int __init raspi_gpio_init(void)
{
int ret;
if (alloc_chrdev_region(&first,0,1,DEVICE_NAME) < 0) {
    printk(KERN_DEBUG "Cannot register device\n");
    return -1;
    }
if ((raspi_gpio_class = class_create( THIS_MODULE,DEVICE_NAME)) == NULL) {
    printk(KERN_DEBUG "Cannot create class %s\n", DEVICE_NAME);
    unregister_chrdev_region(first, 1);
    return -EINVAL;
    }

if (gpio_request_one(NUMGPIO, GPIOF_OUT_INIT_LOW, NULL) < 0) {
    printk(KERN_ALERT "Error requesting GPIO %d\n", NUMGPIO);
    return -ENODEV;
}
raspi gpio devp.dir = out;
raspi gpio devp.state = low;
raspi_gpio_devp.cdev.owner = THIS_MODULE;
spin lock init(&raspi gpio devp.lock);
cdev init(&raspi gpio devp.cdev, &raspi gpio fops);
if ((ret = cdev_add(&raspi_gpio_devp.cdev,(first + NUMGPIO),1))) {
    printk (KERN ALERT "Error %d adding cdev\n", ret);
    device_destroy (raspi_gpio_class,MKDEV(MAJOR(first),MINOR(first) + NUMGPIO));
    class_destroy(raspi_gpio_class);
    unregister chrdev region(first, 1);
    return ret;
}
if(device create(raspi gpio class,NULL,MKDEV(MAJOR(first),MINOR(first)+NUMGPIO),NULL,"raspiGpio%
d",NUMGPIO) == NULL) {
    class_destroy(raspi_gpio_class);
    unregister chrdev region(first, 1);
    return -1;
}
printk("RaspberryPi GPIO driver initialized\n");
return 0;
}

/*
* raspi_gpio_exit - Clean up GPIO device driver when unloaded
```

```
*
* This functions performs the following tasks:
* Release major number
* Release device nodes in /dev
* Release per-device structure arrays
* Detroy class in /sys
* Set GPIO pin to output, low level
*/
static void   exit raspi gpio exit(void)
{

unregister chrdev region(first, 1);
gpio_direction_output(NUMGPIO, 0);
device_destroy ( raspi_gpio_class,MKDEV(MAJOR(first), MINOR(first) + NUMGPIO));
gpio free(NUMGPIO);
class_destroy(raspi_gpio_class);
printk(KERN_INFO "RaspberryPi GPIO driver removed\n");
}

module_init(raspi_gpio_init);
module exit(raspi gpio exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Me");
MODULE_DESCRIPTION("led driver for GPIO2 in Raspberry Pi B+ platform);
```

A continuación se muestra el código fuente de una aplicación que hace uso del modulo anterior para configurar un pin como salida al que se le permite introducir como parámetro el nivel que tendrá el pin asociado.

```c
/*
* Name : output_test.c
* Description : This is a test application which is used for testing
* GPIO output functionality of the raspi-gpio Linux device driver
* implemented for Raspberry Pi revision B+ platform. The test
* application first sets the GPIO pin on the Raspberry Pi to
* output, then it sets the GPIO pin to "high"/"low" logic
* level based on the options passed to the program from the command
* line
* Usage example:
* ./output_test 1 // Set all GPIO pins to output, high state
* ./output test 0 // Set all GPIO pins to output, low state
*/
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#define NUMGPIO 2
#define BUF_SIZE 4
#define PATH_SIZE 20

int main(int argc, char **argv)
{
int value;
int fd;
char path[PATH_SIZE];
char buf[BUF_SIZE];

if (argc != 2) {
   printf("Option low/high must be used\n");
   exit(EXIT_FAILURE);
}

// Open GPIO pin
snprintf(path, sizeof(path), "/dev/raspiGpio%d", NUMGPIO);
fd = open(path, O_WRONLY);
if (fd < 0) {
   perror("Error opening GPIO pin");
   exit(EXIT_FAILURE);
}

// Set direction of GPIO pins to output
printf("Set GPIO pins to output, logic level :%s\n", argv[1]);
strncpy(buf, "out", 3);
buf[3] = '\0';

if (write(fd, buf, sizeof(buf)) < 0) {
   perror("write, set pin output");
   exit(EXIT FAILURE);
}

// Set logic state of GPIO pins low/high
if(strcmp(argv[1],"low")==0)
   value = 0;
else if (strcmp(argv[1],"high")==0)
   value = 1;
else
   value = -1;
```

```
if (value == 1) {
   strncpy(buf, "1", 1);
   buf[1] = '\0';
} else if (value == 0) {
   strncpy(buf, "0", 1);
   buf[1] = '\0';
} else {
   printf("Invalid logic value\n");
   exit(EXIT FAILURE);
}

if (write(fd, buf, sizeof(buf)) < 0) {
   perror("write, set GPIO state of GPIO pins");
   exit(EXIT_FAILURE);
}

return EXIT_SUCCESS;
}
```

Finalmente se muestra el código fuente de una aplicación que hace uso del modulo del dispositivo para configurar un pin como entrada, y se accede al dispositivo para conocer el nivel lógico del pin asociado.

```c
/*
* Name : input_test.c
* Description : This is a test application which is used for testing
* GPIO input functionality of the raspi-gpio Linux device driver
* implemented for Raspberry Pi revision B+ platform. The test
* application first sets the GPIO pin on the Raspberry Pi to
* input, then it reads the GPIO pin logic level and print thee
* value to the terminal.
*/
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#define NUMGPIO 2
#define BUF SIZE 3
#define PATH_SIZE 20

int main(int argc, char **argv)
{
int fd;
char path[PATH SIZE];
char buf[BUF_SIZE];
char readBuf[2];

// Open all GPIO pins
snprintf(path, sizeof(path), "/dev/raspiGpio%d", NUMGPIO);
fd = open(path, O_RDWR);
if (fd < 0) {
   perror("Error opening GPIO pin");
   exit(EXIT_FAILURE);
}

// Set direction of GPIO pins to input
printf("Set pins to input\n");
strncpy(buf, "in", 2);
buf[2] = '\0';
if (write(fd, buf, sizeof(buf)) < 0) {
   perror("write, set pin input");
   exit(EXIT_FAILURE);
}

// Read logic level of GPIO pins and display them to the terminal
if (read(fd, readBuf, 1) < 1) {
   perror("write, set pin input");
   exit(EXIT_FAILURE);
}
readBuf[1] = '\0';
printf("GPIO pin: %d Logic level: %s\n", NUMGPIO, readBuf);
return EXIT SUCCESS;
}
```

Enunciado de la práctica

1. Realícese la compilación y ejecución de los modulos cargables mostrados en el anunciado (programas ejemplo, acumulador y cliente), y anótese las conclusiones que se obtienen de su ejecución en el sistema huésped y en el sistema cliente.

2. Conectese un circuito (diodo led y resistencia en serie a masa) a la patilla GPIO2 (patilla 3 del conector J8). Utilizando como referencia el driver mostrado en la guía de la práctica que gestiona un pin GPIO, y las aplicaciones de usuario creadas para su gestión, realícese una aplicación que haga parpadear el diodo con una determinada frecuencia de parpadeo. La frecuencia de parpadeo del diodo conectado a la patilla GPIO2 se debe incrementar cuando el usuario introduzca el carácter 'x' por la consola, y se decrementará cuando introduzca el carácter 'y'.

3. Realicese la conexión de otro circuito idéntico a la patilla GPIO3 (patilla 5 del conector J8). Realícese una aplicación que haga parpapear ambos diodos con una frecuencia de parpadeo distinta. La gestión del parpadeo de cada diodo la debe realizar un hilo distinto. Como en el apartado anterior, la frecuencia de parpadeo del diodo conectado a la patilla GPIO2 se debe incrementar cuando el usuario introduzca el carácter 'x' por la consola, y se decrementará cuando introduzca el carácter 'y'. La misma operación debe ocurrir con el diodo conectado a GPIO3 cuando se introduzcan los caracteres 'a' y 'b'.

Como referencia para la resolución de este apartado, dentro de Linux se soporta el estándar POSIX, que dota al sistema de un conjunto de funciones para el desarrollo de aplicaciones sobre este sistema operativo. Entre estas funciones está la que se encarga de la creación de hilos, que es la siguiente

```
#include <pthread.h>

int pthread_create (pthread_t *tid, const pthread_attr_t *attr,
                    void *(*start_routine)(void *), void *arg);
```

- **tid**: dirección de una variable de tipo `pthread_t` que será rellenada con un identificador que referencia al hilo que se crea
- **attr**: dirección de una variable de tipo `pthread_attr_t` que contiene los atributos del hilo que se va a crear. Con `NULL` se toman los atributos por defecto
- **start_routine**: dirección de una función de tipo `void* (*)(void *)` en donde comenzará la ejecución del hilo.
- **arg**: puntero a void que será pasado como argumento de la función `start_routine`.

En el programa siguiente vemos un ejemplo del uso de `pthread_create()`, en el que se ejecuta la función `func()` asincrónicamente. El hilo principal después de haber creado al hijo llamará a la función `sleep()`, pidiendo ser bloqueado durante dos segundos, pasados los cuales recuperará el estado activo; se dice que el hilo se ha echado a dormir. Lo que se pretende es que el hijo haya terminado antes de que el padre se despierte.

```
/*
 * Descripción:
 * Crea un hilo y comprueba que corre.
 */

#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

static int washere = 0;

void * func(void * arg)
{
  printf("Estoy en otro hilo diferente al principal.\n");

  washere = 1; // Las variables globales
               //  son visibles por todos los hilos.
  return 0;
}
```

```c
int main()
{
  pthread t t;

  printf("Estoy en el hilo principal.\n");

  pthread create(&t, NULL, func, NULL);

  sleep(2); // El hilo principal duerme durante 2s

  if(washere == 1)
    printf("Otro hilo diferente al principal ha modificado "
           "la variable global.\n");

  return 0;
}
```

Para compilar el programa necesitamos enlazar con la biblioteca `libpthread.so` como aparece en la siguiente orden:

```
$ gcc -lpthread -o ejemplo_hilos ejemplo_hilos.c
```

Si todo fuera bien, tras la ejecución, la salida de este programa sería:

```
Estoy en el hilo principal.
Estoy en otro hilo diferente al principal.
Otro hilo diferente al principal ha modificado la variable global.
```

A continuación se muestra una representación grafica que muestra el flujo de ejecución del programa: