http://lsi.ugr.es

Diseño de Sistemas Operativos
5º Curso de Ingeniería Informática

Guión de Prácticas – Curso 2010-11

Práctica 2:

Programación de módulos kernel en Linux

Objetivo de la práctica:

Programar varios módulos sencillos para el kernel de Linux al objeto de estudiar las posibilidades que nos ofrecen, y conocer las herramientas que suministra Linux para realizar de forma dinámica la carga/descarga de módulos.

1 Introducción

Un **módulo** es un archivo objeto ELF reubicable que resuelve sus símbolos cuando se carga en el kernel. Una vez cargado, es parte del kernel y su invocación tiene los mismos puntos de entrada que el kernel: interrupciones y/o excepciones. Los manejadores de dispositivos se desarrollan como módulos. Los módulos utilizan una interfaz clara entre el kernel y el dispositivo, lo que hace que el módulo sea fácil de escribir y a la vez mantiene el código del kernel libre de desorden.

El módulo debe ser compilado (no enlazado), y se carga en el kernel en ejecución con ***insmod***, que es un enlazador dinámico, que se utiliza para resolver los símbolos indefinidos del módulo en direcciones del kernel mediante la tabla de símbolos del kernel.

Esto significa que podemos escribir un módulo de forma muy similar a un programa C, y que podemos utilizar funciones no definidas por nosotros. La diferencia es que sólo podemos utilizar un conjunto reducido de funciones externas, que son las *funciones públicas* definidas por el kernel. Si tenemos dudas de si una función kernel es pública, o no, podemos buscarla por su nombre en el archivo fuente */usr/src/linux/kernel/ksyms.c*, o en la tabla de ejecución */proc/ksyms*. En teoría, podemos escribir cualquier cosa dentro de un módulo. En la práctica, debemos recordar que un módulo es código kernel y debe poseer una interfaz bien definida con el resto del sistema.

Recordar que Unix transfiere la ejecución desde modo usuario a modo kernel cuando se realiza una llamada al sistema o se produce una interrupción. El código kernel que ejecuta una llamada al sistema trabaja en el contexto del proceso actual – trabaja en beneficio del proceso actual y puede acceder a las estructuras de datos del espacio de direcciones del proceso. El código que maneja interrupciones, por otro lado, es asíncrono respecto de los procesos y no esta relacionado con ningún proceso.

El objeto de un módulo es extender la funcionalidad del kernel. Un código modularizado se ejecuta en espacio del kernel. Normalmente un manejador realiza las dos tareas explicadas anteriormente: algunas funciones del módulo se ejecutan como parte de una llamada al sistema, y otras están a cargo del manejador de interrupciones.

Cómo el resto del kernel, los módulos y manejadores de dispositivos deben ser reentrantes, es decir, pueden ser ejecutados en más de un contexto a la vez. Las estructuras de

datos deben ser diseñadas cuidadosamente para mantener separadas múltiples hebras de ejecución, y el código debe cuidar el acceso a datos compartidos para evitar la corrupción de los datos. Un error común entre programadores de manejadores es asumir que la concurrencia no es un problema mientras que un segmento de código particular no se bloquee. Si bien es cierto que Linux es no apropiativo hasta la versión 2.4 (el 2.6 es apropiativo), debemos tener presentes que el manejador que diseñemos se puede ejecutar en un sistema SMP.

La presente descripción hacer referencia al desarrollo de módulos en la versión 2.4 del kernel, que es la que tenemos en prácticas. Las principales diferencias con su desarrollo en la versión 2.6 del kernel las hemos visto en clase de teoría.

2 Desarrollo de la práctica

La práctica consta de dos bloques en los cuales se van a tratar dos aspectos separados:

- **Bloque I:** Realizaremos dos módulos sencillos, relativamente aislados del resto del kernel, para estudiar las características de programación de módulos y la dependencia de módulos. En concreto, las tareas a realizar en este bloque son dos módulos, denominados *acumulador* y *cliente*, y que se atienen al siguiente comportamiento:
  1. Ambos módulos deben mostrar cada vez que se insertan o extraigan el instante de tiempo de la operación. Se mostrará el número de segundos desde la Época (1 de enero de 1970). Para lo cual, consultaremos la variable del kernel `xtime` declarada en *kernel/sched.h*. Esta variable es de tipo `struct timeval`, tipo que esta definido en *include/linux/time.h* y que será necesario incluir en los módulos. Recuerde que para poder utilizar esta variable en los módulos hay que declararla como `extern` que el compilador no conoce ni su tipo ni su tamaño.
  2. El módulo *acumulador* define una función `void acumular(int i)`, que suma en una variable global del módulo el valor que se le pasa como parámetro. También, implementa la función `int consultar(void)` que permite consultar el valor de dicha variable. La variable tiene valor inicial cero, y cuando se extraiga el módulo mostrará el valor que se haya acumulador.
  3. El módulo *cliente*, al insertarse, llamará a la función `acumular()` del módulo *acumulador* pasándole un valor igual al que le pasamos al módulo al insertarlo. Cuando se extraiga, mostrará cuanto llevamos acumulado hasta el momento haciendo uso de la función `llevamos()`.

  Para comprobar el funcionamiento, inserta el módulo acumulador una vez, y el módulo cliente varias veces, observando que el comportamiento es correcto. Finalmente, extrae el módulo acumulador y observa que el resultado final también es correcto. A continuación, observa que pasa si insertamos el módulo cliente sin haber insertado el acumulador, o intentamos extraer el acumulador estando insertado aún el cliente. Explica lo ocurrido en ambos casos.

- **Bloque II:** Ahora vamos a implementar un módulo para ver la interacción con el resto del kernel. El los ejercicios anteriores, los módulos solo realizan su función cuando se insertan o retiran. Como indicábamos en la introducción, el kernel se activa a través de una llamada al sistema, o mediante una interrupción. Pues bien, en este bloque, vamos a desarrollar un módulo que intercepte las llamadas al sistema `geteuid()`, `getuid()`, `getgid()`, y `getegid()` al objeto de cambiar el funcionamiento de la orden `whoami` por un nuevo comportamiento de tu elección. Además, podemos interceptar las llamadas y tras dar un mensaje nuestro, la orden diga dando también la información que ofrecía antes del cambio.

3 Material necesario

A continuación mostramos el código completo de un módulo "Hola, Mundo". Este módulo debe compilarse y ejecutarse bajo las versiones 2.0 a 2.4 del kernel de Linux:

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

int init_module(void) {printk("<1>Hola, Mundo\n"); return 0;}
void cleanup_module(void) {printk("<1>Adios mundo cruel\n");}
```

La función `printk` esta definida en el kernel de Linux y se comporta de forma similar a la función `printf` de la biblioteca estándar de C. El kernel necesita esta función pues no dispone de la ayuda de la biblioteca C. El módulo puede invocar a `printk` ya que después de su instalación queda enlazado al kernel y puede acceder a los símbolos públicos del kernel (variables y funciones que detallaremos en la siguiente sección). La cadena `<1>` indica la prioridad del mensaje. Especificamos la prioridad más alta (menor número cardinal) para que se muestre en la consola, dependiendo de la versión del kernel que ejecutemos, la versión del demonio *klog*, y nuestra configuración.

Podemos probar el módulo invocando a *insmod* y *rmmod*. Recordar que solo el superusuario puede cargar y descargar un módulo. Para ejecutar el código anterior:

```
root# gcc –c hola.c
root# insmod ./hola.o
Hola, Mundo
root# rmmod hola
Adios mundo cruel
root#
```

En nuestro sistema, deberemos hacer una instalación de los módulos "forzada" debido a las diferencias entre versiones del kernel de nuestro sistema de prácticas. Es decir, debemos invocar a *insmod* de la siguiente forma:

```
root# insmod –f ./hola.c
```

En base al mecanismo que use nuestro sistema para escribir las líneas de mensajes, la salida puede ser diferente. La salida anterior se tomó de una consola de texto; si ejecutamos *insmod* y *rmmod* desde un *xterm*, la salida se producirá en los archivos de log del sistema (como */var/log/messages*). En breve, veremos como escribir desde un xterm.

Módulos kernel frente a aplicaciones

Antes de seguir, conviene subrayar las diferencias entre un módulo kernel y una aplicación:

1. Mientras que una aplicación se ejecuta desde el principio hasta el final, un módulo se registra a sí mismo para servir solicitudes futuras, y su función principal termina inmediatamente. Es decir, la tarea de la función `init_module` es preparar las futuras invocaciones de las funciones del módulo.
2. Una aplicación puede invocar funciones externas pertenecientes a las bibliotecas. Un módulo se enlaza sólo con el kernel por lo que sólo puede invocar a las funciones exportadas por el kernel. La figura 1 muestra cómo se utilizan las llamadas a función y los punteros a funciones para añadir nueva funcionalidad al kernel en ejecución.
3. Actualmente, las cabeceras de espacio de usuario están separadas de las cabeceras de espacio kernel. Algunas veces las aplicaciones incluyen cabeceras kernel, ya sea porque se utilice una vieja biblioteca o porque se necesita información no incluida en las cabeceras usuario. Sin embargo, muchas declaraciones de las cabeceras kernel son relevantes solo al propio kernel y no deben verse por aplicaciones en espacio de usuario. Estas declaraciones se protegen mediante el bloque `#ifdef __KERNEL__`. Esta es la razón por la cual los manejadores, y el código kernel, debe compilarse con el símbolo del procesador `__KERNEL__` definido. El papel de las cabeceras kernel individuales se introducirán según sean necesarias.
4. El trabajo en el desarrollo de sistemas software grandes debe evitar la *polución del espacio de nombres* (fenómeno que ocurre cuando existen muchas funciones y variables globales cuyos nombres no son lo suficientemente significativos para que se distingan fácilmente). Las colisiones en el espacio de nombres requiere un esfuerzo extra del programador para

recordar los nombres reservados y puede acarrean problemas que pueden ir desde la carga del módulo hasta fallos raros. La mejor forma de evitar este problema es declarar todos nuestros símbolos como `static`[1] y utilizar un prefijo único dentro del kernel para los símbolos que sean globales. Como escritores de módulos, podemos controlar la visibilidad externa de nuestros símbolos como describiremos en el apartado "La tabla de símbolos del kernel".

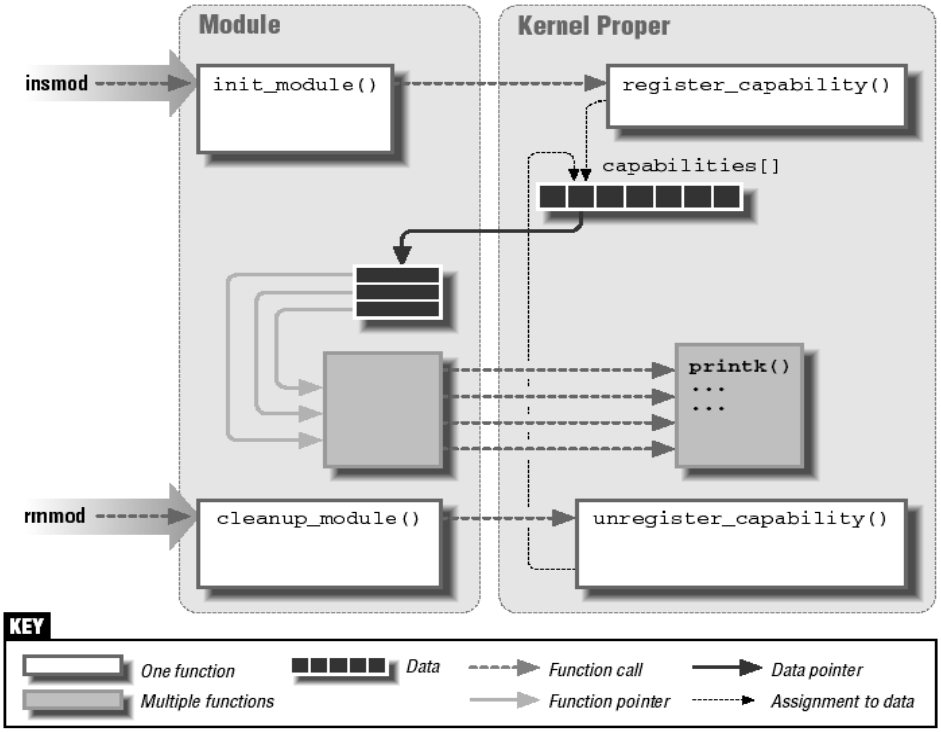

Figura 1.- Enlazando un módulo con el kernel.

5. La última diferencia es cómo cada entorno gestiona las faltas: mientras que una falta de segmentación en una aplicación es inofensiva durante el desarrollo de una aplicación y podemos utilizar un depurador para detectar el error, una falta kernel es fatal al menos para el proceso actual, si no para el sistema completo.

El proceso actual

El código del kernel puede conocer el proceso actual (en el contexto del cual se ejecuta) accediendo al item `current`, un puntero a una `struct task_struct`, que en la versión 2.4 del kernel se declara en `<asm/current.h>`, incluida por `<linux/sched.h>`. El puntero `current` referencia al proceso de usuario ejecutándose actualmente y nos puede servir para obtener información específica del proceso, si la necesitamos. Un ejemplo de esta técnica lo veremos en el Capítulo 5, "Control de acceso a un archivo de dispositivo". Por ejemplo, la siguiente declaración imprime el ID del proceso y el nombre de la orden del proceso actual:

```
printk("El proceso es \"%s\" (pid %i)\n", current->comm, current->pid);
```

Compilación y carga

Vamos a dedicar el apartado a escribir un módulo completo aunque sin tipos, es decir, un módulo que no pertenece a ninguna de las clases siguientes: de caracteres, de bloques, o de red. El

[1] La mayoría de las versiones de *insmod* (pero no todas) exportan todos los símbolos como no-static si no encuentra instrucciones específicas en el módulo; por ello es sabio declarar como static todos los símbolos que no deseemos exportar.

módulo que veremos de denomina *skull* (Simple Kernel Utility for Loading Localities). Podemos reutiliza el módulo para nuestros propios módulos o manejadores para lo cual solo deberemos eliminar el código que no nos sirva.

Antes de comenzar con las funciones de inicialización y limpieza del módulo, construiremos un makefile para construir el código objeto que debe cargar el kernel.

Primero, debemos definir el símbolo `__KERNEL__` en el procesador antes de incluir alguna cabecera. Como mencionábamos antes, gran parte de los contenidos específicos del kernel en los archivos de cabecera no estarían disponibles sin este símbolo.

Otro símbolo importantes es `MODULE`, que debe definirse antes de incluir `<linux/module.h>` (excepto para los manejadores que están enlazados directamente en el kernel, pero estos no están cubiertos en estas notas).

Si estamos compilando para una máquina SMP, necesitamos definir `__SMP__` antes de incluir las cabeceras kernel. En la versión 2.2 la opción multiprocesador o uniprocesador fue ascendida a un item de configuración propio, así utilizando estas líneas muy al principio del módulo realizaremos la tarea:

```
#include <linux/config.h>
#ifdef CONFIG_SMP
# define __SMP__
#endif
```

En la compilación debemos utilizar el indicador `-O`, ya que muchas funciones están declaradas en línea[2] en los archivos de cabecera. *gcc* no expande las funciones en línea salvo que se habilite la optimización, pero puede aceptar las opciones –g y -O, permitiéndonos depurar código que utiliza funciones en línea. Es importante la expansión adecuada de estas funciones pues el kernel hace un uso intensivo de ellas.

Debemos comprobar que el compilador que utilizamos iguala al kernel para el que estamos compilando, ver el archivo *Documentation/Changes* en el árbol de fuentes del kernel. El kernel y el compilador se desarrollan a la vez por diferentes grupos, así algunas veces los cambios en una herramienta revelan errores en otra. Algunas distribuciones entregan una versión del compilador demasiado nueva para construir el kernel de forma fiable. En este caso, estas suelen suministrar un paquete separado (a menudo denominado *kgcc*) con un compilador pensado para la compilación del kernel.

Finalmente, al objeto de evitar errores desagradables, es recomendable utilizar el indicador `-Wall` (todos los avisos), y también fijar todas las características de nuestro código que provocan avisos del compilador, incluso si esto requiere cambios en nuestro estilo de programación. Cuando se escribe código kernel, el estilo de codificación preferido es el propio de Linus. En Documentación/CodingStyle encontramos una lectura amena y una lección obligatoria para cualquier persona interesada en los detalles del kernel.

Todas las definiciones e indicadores que hemos introducido hasta ahora tienen su mejor ubicación en la variable `CFLAGS` utilizada por *make*.

Si el módulo que vamos a construir esta dividido en diferentes archivos fuentes, lo cual es usual, el makefile necesita una regla para unir los diferentes archivos objetos. Los archivos objeto se unen con la orden `ld -r`, que realmente no es una operación de enlazado aunque la realice el enlazador. La salida de esta orden es otro objeto módulo que incorpora todo el código de los archivos de entrada. La opción –r significa que el archivo de salida es reubicable.

El siguiente makefile es un ejemplo mínimo de cómo se construye un módulo compuesto de dos archivos fuentes. Si nuestro módulo solo tiene un único archivo fuente, debemos saltar la entrada que contiene `ld -r`:

[2] Sin embargo, es arriesgado el uso de una optimización mayor que –O2, dado que el compilador podría poner funciones en línea que no esta declaradas como inline en el código. Esto puede ser un problema con el código kernel, dado que algunas funciones esperan encontrar una estructura de pila estándar cuando son invocadas.

```
# Change it here or specify it on the "make" command line
KERNELDIR = /usr/src/linux

include $(KERNELDIR)/.config

CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include \
  -O -Wall

ifdef CONFIG_SMP
 CFLAGS += -D__SMP__ -DSMP
endif

all: skull.o

skull.o: skull_init.o skull_clean.o
  $(LD) -r $^ -o $@

clean:
  rm -f *.o *~ core
```

Después de construir el módulo, el siguiente paso es cargarlo en el kernel. Como hemos comentado *insmod* realiza este trabajo. Este programa es similar a *ld*, enlaza los símbolos no resueltos en el módulo con los de la tabla de símbolos del kernel en ejecución. A diferencia del enlazador, esta no modifica el archivo en disco sino una copia en memoria del mismo. *insmod* acepta cierto número de opciones en la línea de órdenes (ver la página de manual), y puede asignar valores a variables enteras o cadenas en el módulo antes de ser enlazado con el kernel. Así, si se diseña correctamente un módulo, puede configurase en tiempo de carga; esto da mayor flexibilidad al usuario que la configuración en compilación. La configuración en tiempo de carga la explicaremos en el apartado "Configuración manual y automática".

Dependencias de versión

Teniendo en mente que el código de nuestro módulo debe recompilarse para cada versión del kernel con la que debe ser enlazado. Cada módulo define un símbolo denominado `__module_kernel_version__`, que *insmod* iguala con el número de versión del kernel actual. Este símbolo se situa en la sección `.modinfo` del ELF. Esta descripción se aplica sólo a las versiones 2.2 y 2.4 del kernel; Linux 2.0 hace lo mismo pero de otra forma.

El compilador define este símbolo por nosotros cuando incluimos `<linux/module.h>`. Esto significa que si nuestro módulo esta compuesto de varios archivos, solo debemos incluirlo en uno de ellos (salvo que utilicemos `__NO_VERSION__`, que introduciremos en breve).

Si las versiones no coinciden, podemos carga el módulo especificando el conmutador `-f` (forzado), pero esta operación no es segura y puede fallar. La carga puede fallar porque simplemente no coincidan los símbolos, lo que provoca un mensaje de error, o debido a cambios internos del kernel, en cuyo caso podemos provocar un error serio y posiblemente un pánico de sistema. La no coincidencia de versiones se puede tratar de una manera más elegante utilizando versiones en el kernel. Esta práctica no trataremos en profundidad en control de versiones.

Si deseamos compilar el módulo para una versión particular del kernel, debemos incluir los archivos de cabecera específicos para ese kernel (por ejemplo, declarando un `KERNELDIR` diferente) en el makefile anterior.

Cuando solicitamos cargar un módulo, *insmod* sigue su propio camino de búsqueda para encontrar el archivo objeto, buscando en los directorios dependientes de la versión bajo */lib/modules*. Aunque versiones antiguas buscaban en el directorio actual, este comportamiento esta ahora deshabilitado por seguridad (el mismo problema que ocurre con la variable de entorno

PATH). Así, si necesitamos cargar un módulo desde nuestro directorio de trabajo debemos utilizar *./module.o*, que funciona con todas las versiones de la herramienta.

A veces, encontraremos interfaces kernel con un comportamiento diferente entre las versiones 2.0.x y 2.4.x. En este caso, necesitamos recurrir a las macros que definen el número de versión del árbol de fuentes actual, que están definidas en `<linux/versión.h>`. En lugar de iniciar una larga discusión específica de la versión 2.4, apuntaremos algunos cambios en los ejemplos que lo requieran.

La cabecera, automáticamente incluida por *linux/module.h*, define las siguientes macros:

`UTS_RELEASE` La macro se expande por una cadena que describe al versión de este árbol del kernel. Por ejemplo, "2.3.48".

`LINUX_VERSION_CODE` La macro expande a la representación binaria de la versión del kernel, un byte por cada parte del número de liberación de la versión. Por ejemplo, el código para 2.3.48 es 131888 (esto es, 0x020330). Con esta información, podemos determinar (casi) fácilmente la versión del kernel con la que tratamos.

`KERNEL_VERSION(major, minor, release)` Esta es la macro utilizada para construir un "código_versión_kernel" a partir de los números individuales de constituyen el número de versión. Por ejemplo, KERNEL_VERSION(2, 3, 48) se expande por 131888. Esta macro es muy útil cuando necesitamos comparar la versión actual y un punto de comprobación conocido.

El archivo *version.h* esta incluido por el módulo *module.h*, por lo que no tenemos que incluirlo explícitamente. Por otro lado, podemos evitar esta inclusión declarando antes `__NO_VERSION__`. Utilizaremos __NO_VERSION__ si necesitamos incluir <linux/module.h> en varios archivos fuentes que serán enlazados juntos para formar un único módulo, por ejemplo, si necesitamos preprocesar macros declaradas en *module.h*. La declaración `__NO_VERSION__` de antes de incluir module.h evita la declaración automática de la cadena `__module_kernel_version__` o su equivalente en los archivos fuentes donde no lo deseemos (*ld –r* podría quejarse sobre las múltiples definiciones del símbolo). Los módulos ejemplos que vamos a utilizar lo hacen así.

La mayoría de las dependencias basadas en la versión del kernel pueden solventarse con condicionales del preprocesador explotando `KERNEL_VERSION` y `LINUX_VERSION`. Las dependencias de versión deberían, sin embargo, confundir el código del manejador con multitud de #ifdef; la mejor forma de tratar con incompatibilidades es confinarlas en una cabecera específica.

La tabla de símbolos del kernel

Comentábamos cómo *insmod* resuelve los símbolos no definidos contra la tala de símbolos públicos del kernel. La tabla contiene las direcciones de los ítems globales del kernel –funciones y variables- que son necesitados para implementar un manejador modularizado. La tabla de símbolos puede leerse en formato texto desde el archivo */proc/ksyms*.

Cuando se carga un módulo, cualquier símbolo exportado por el módulo se convierte en parte de la tabla de símbolos del kernel, y podemos verlo en */proc/ksyms* o en la salida de la orden *ksyms*.

Nuevos módulos pueden utilizar símbolos exportados por nuestro módulo, y podemos apilar nuevos módulos sobre otros módulos. El apilamiento de módulos es ampliamente implementado en el kernel: el sistema de archivos *msdos* descansa sobre los símbolos exportados por el módulo *fat*, y cada entrada al módulo de dispositivo de USB se apila sobre los módulos *usbcore* e *input*.

El apilamiento de módulos es útil en proyectos complejos. Si se implementa una nueva abstracción en la forma de manejador de dispositivos, esta podría ofrecer un enchufe para implementaciones específicas de hardware. Por ejemplo, el conjunto de manejadores video-for-linux esta dividido en un módulo genérico que exporta símbolos utilizados por los manejadores de dispositivos de más bajo nivel para hardware específico. En función de nuestra configuración,

cargamos el módulo de video genérico y el módulo específico para el hardware instalado. El soporte para puertos paralelos y una amplia variedad de dispositivos ligables se gestionan de la misma manera, como es el subsistema kernel USB. La Figura 2 muestra el apilamiento del subsistema del puerto paralelo; las fechas muestran las comunicaciones entre los módulos y con la interfaz de programación del kernel.

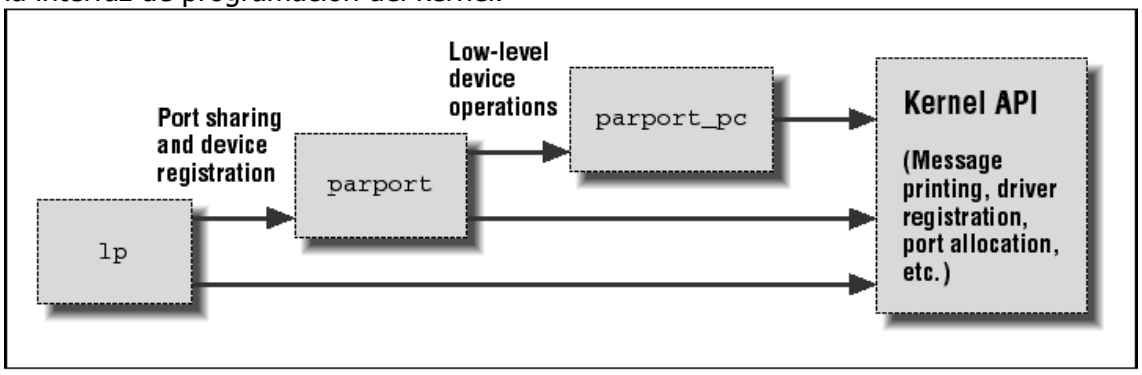

Figure 2-2. Stacking of parallel port driver modules

Cuando se utilizan módulos apilables, es útil la utilidad *modprobe*. Esta funciona de forma muy similar a *insmod*, pero también carga cualesquiera otros módulos que sean necesarios por el módulo que deseamos cargar. Así, una orden *modprobe* puede a veces reemplazar varias invocaciones de *insmod* (si bien necesitamos aún *insmod* cuando cargamos nuestros módulos desde el directorio actual, dado que *modprobe* solo mira en el árbol de módulos instalados).

La modularización por capas puede ayudar a reducir el tiempo de desarrollo al simplificar cada capa. Esto es similar a la separación entre mecanismo y política ampliamente utilizado.

En el caso usual, un módulo implementa su propia funcionalidad sin necesidad de exportar ningún símbolo. Sin embargo, necesitaremos exportar los símbolos cuando queramos que otros módulos puedan beneficiarse de ellos. También, necesitaremos incluir instrucciones específicas para evitar exportar todos los símbolos no-static, como la mayoría de las versiones de *modutils*, pero no todas, exporta todos por defecto.

Los archivos de cabecera del kernel de Linux suministran una forma conveniente de manejar la visibilidad de nuestros símbolos, reduciendo así la polución del espacio de nombres y promover el ocultamiento adecuado de la información. Este mecanismo funciona con los kernel 2.1.18 y posteriores; el kernel 2.0 tiene un mecanismo completamente diferente, que describiremos al final.

Si nuestro módulo no exporta ningun símbolo, podemos desear hacerlo explícito colocando una línea con la llamada a la macro:

```
EXPORT_NO_SYMBOLS;
```

La macro se expande en una directiva en ensamblador y puede aparecer en cualquier lugar del módulo. Sin embargo, el código portable debe situarla en la inicialización del módulo (*init_module*), dado que la versión de esta macro definida en *sysdep.h* para viejos kernels solo funcionará allí.

Si, por de otro lado, necesitamos exportar un subconjunto de símbolos desde nuestro módulo, el primer paso es definir la macro del preprocesador `EXPORT_SYMTAB`. Esta macro debe estar definida antes de incluir *module.h*. Es común definirla en tiempo de compilación con el indicador –D del compilador en el *Makefile*.

Si esta definido `EXPORT_SYMTAB`, los símbolos individuales se exportan con la pareja de macros:

```
EXPORT_SYMBOL (nombre);
EXPORT_SYMBOL_NOVERS (nombre);
```

Cualquiera de las dos versiones de la macro hará que el símbolo dado este disponible fuera del módulo; la segunda versión exporta el símbolo sin información de versión. Los símbolos deben

exportarse fuera de cualquier función ya que las macros expanden por la declaración de la variable.

Inicialización y cierre

Como ya hemos mencionado, `init_module` registra la funcionalidad ofrecida por el módulo. Para cada funcionalidad, existe una función específica del kernel que realiza la operación de registro. Los argumentos pasados a las funciones de registro del kernel son normalmente un puntero a una estructura de datos que describe la funcionalidad y el nombre de la función registrada. La estructura de datos suele empotrar punteros a las funciones del módulo, que es como las funciones del cuerpo del módulo son invocadas. Entre los ítems que se pueden registrar podemos encontrar manejadores de dispositivos, archivos /proc, dominios ejecutables, y disciplinas de línea. Muchos de estos ítems registrables soportan funciones que no están directamente relacionadas con el hardware si no que son abstracciones software. Estos ítems pueden registrarse dado que se pueden integras de cualquier forma en la funcionalidad de los manejadotes.

Manejo de errores en *init_module*

Si se produce un error cuando registramos la utilidad, debemos deshacer cualquier operación de registro realizada antes del fallo. Se puede producir un error, por ejemplo, si no hay suficiente memoria en el sistema para asignar una nueva estructura de datos o porque un recurso que esta siendo solicitado es utilizado por otro manejador.

Linux no mantiene un registro de funcionalidad por módulo que ha sido registrado, así el módulo debe deshacer todo por él mismo si *init_module* falla. Si fallamos en la operación de des-registrar el módulo, se deja al kernel en un estado inseguro: no poedemos volver a registrar nuestra función recargando el módulo dado que aparecería como ocupado, y no podemos des-registrarlo porque necesitamos el mismo puntero que utilizamos para registrarlo. La recuperación de esta situación es complicada, y normalmente nos veremos forzados a re-arrancar el sistema.

La recuperación de errores es a menudo bien gestionada por la declaración `goto`. Normalmente, odiamos el goto, pero en situación es útil. En el kernel, `goto` se utiliza a menudo para tratar con los errores.

El siguiente código ejemplo, que utiliza funciones de registro y des-registro ficticias, se comporta correctamente si falla la inicialización en cualquier punto.

```
int init_module(void)
{
      /* el registro toma un puntero y un nombre */
      err = register_esto(ptr1, "skull");
      if (err) goto fallo_esto;
      err = register_eso(ptr2, "skull");
      if (err) goto fallo_eso;
      err = register_aquello(ptr3, "skull");
      if (err) goto fallo_aquello;

      return 0; /* exito */

      fallo_aquello: unregister_eso(ptr2, "skull");
      fallo_eso: unregister_esto(ptr1, "skull");
      fallo_esto: return err; /* propaga el error */
}
```

El código intenta registrar tres funciones ficticias. La declaración `goto` se utiliza en caso de fallo para provocar la des-registración sólo de las funciones que han sido registradas con éxito antes de que las cosas fuesen mal.

Otra opción, que no requiere la declaración `goto`, es mantener la pista de lo que se ha registrado con éxito y llamar a *cleanup_module* en caso de cualquier error. La función de limpieza sólo deshará los pasos que se han finalizado con éxito. Sin embargo, esta alternativa requiere más código. El valor devuelto por *init_module*, `err`, es un código de error. En el kernel de Linux, los códigos de error son número negativos que pertenecen al conjunto definido en <linux/errno.h>. So deseamos generar nuestros propios códigos de error en lugar de retornar lo que obtenemos de otras funciones, deberemos incluir <linux/errno.h> al objeto de utilizar los valores simbólicos tales como `-ENODEV`, `-ENOMEM`, etc. Siempre es buena práctica retornar los códigos de error adecuados, ya que los usuarios de los programas pueden traducirlos a cadena con sentido utilizando la función *perror* o similar.

Obviamente, *cleanup_module* deshace cualquier operación de registro realizada por init_module, y es costumbre (no obligación) des-registrar funciones en el orden inverso usado para registrarlas (como vimos en el fragmento de código anterior).

Si nuestras inicializaciones y limpiezas son más complejas que tratar unos cuantos ítems, el enfoque de gotos puede no se práctico. Para evitar la duplicidad de código, podemos llamar a *cleanup_module* desde dentro de *init_module* si se produce un error.

El contador de uso

El sistema mantiene un contador de uso para cada módulo para poder determinar cuando este puede ser eliminado de forma segura. En los kernels modernos, el sistema mantiene automáticamente este contador, utilizando un mecanismo que veremos en el próximo capítulo. Sin embargo, hay veces que necesitamos ajustar manualmente el contador de uso. El código portable para kernels más antiguos debe utilizar un contador de uso manual. Para trabajar con el contador de uso, utilizamos las tres macros:

`MOD_INC_USE_COUNT`	Incrementa el contador para el módulo actual.
`MOD_DEC_USE_COUNT`	Decrementa el contador.
`MOD_IN_USE`	Se evalúa a cierto si el contador no es cero.

Las macros están definidas en `<linux/module.h>` y actúan cono estructuras de datos internas que no deben ser accedidas directamente por el programador.

Observar que no hay que comprobar `MOD_IN_USE` dentro de *cleanup_module*, ya que la comprobación la realiza la llamada al sistema *sys_delete_module*.

La gestión adecuada del contador de uso es crítica para la estabilidad del sistema. Recordar que el kernel puede decidir intentar descargar nuestro módulo en cualquier instante. Un error común de la programación de módulos es iniciar una serie de operaciones (por ejemplo, en respuesta de una solicitud a *open*) e incrementar después el contador de uso. Si el kernel descarga el módulo entre las operaciones, el caos esta asegurado. Para evitar esta clase de problemas, debemos llamar a `MOD_INC_USE_COUNT` antes de hacer cualquier otra cosa en el módulo.

No seremos capaces de descargar un módulo si perdemos la pista del contador de uso. Esta situación puede ocurrir con frecuencia durante el desarrollo, por lo que debemos mantenerlo en mente. Por ejemplo, si se destruye un proceso porque nuestro manejador desreferencia un puntero NULL, el manejador no será capaz de cerrar el dispositivo y el contador de uso no volverá a cero. Una posible solución es deshabilitar completamente el contador de uso durante el ciclo de depuración redefiniendo tanto `MOD_INC_USE_COUNT` como `MOD_DEC_USE_COUNT` a no-ops. Otra solución es el uso de cualquier otro método para forzar el contador a cero. Las comprobaciones razonables nunca deben soslayarse en un módulo en producción. Para depuración, sin embargo, el uso de la fuerza bruta en ocasiones ayuda a ahorrar esfuerzo y tiempo de desarrollo.

El valor actual del contador de uso se encuentra en el tercer campo de cada entrada de */proc/modules*. Este archivo muestra los módulos cargados actualmente en el sistema, con una

entrada para cada módulo. Los campos son el nombre del módulo, el número de bytes de memoria que utiliza, en el valor actual de contador de uso. Un archivo */proc/modules* típico:

```
autofs                  11264   1 (autoclean)
pcnet32                 12048   1 (autoclean)
ipchains                38976   0 (unused)
usb-uhci                20720   0 (unused)
usbcore                 49664   1 [usb-uhci]
BusLogic                89696   3
sd_mod                  11680   3
scsi_mod                95072   2 [BusLogic sd_mod]
```

Aquí vemos, entre otras cosas, que el módulo del puerto paralelo se a cargado de forma apilada. El marcador (`autoclean`) identifica a los módulos gestionados por kmod o kerneld; el marcador (`unused`) significa exactamente eso. En Linux 2.0, el tamaño esta expresado en páginas (4 KB).

Descarga

Para descargar un módulo utilizamos la orden rmmod. Su tarea es mucho más simple que la carga, ya que no se realiza enlazado. La orden invoca la llamada al sistema *delete_module*, que invoca a *cleanup_module* en el propio módulo si el contador de uso es cero o, en otro caso, devuelve error.

La implementación de *cleanup_module* tiene a su cargo des-registrar cada item que fue registrado por el módulo. Sólo los símbolos exportados se eliminan de forma automática.

Funciones explícitas de inicialización y limpieza

Como hemos visto, el kernel invoca a *init_module* y *cleanup_module* para realizar las labores de inicialización y limpieza, respectivamente. En kernels más modernos, sin embargo, estas funciones tienen a menudo nombres diferentes. En el kernel 2.3.13, existe un mecanismo para la designación explícita de estas rutinas; el uso de este mecanismo sigue el estilo de programación preferido.

Por ejemplo, si deseamos denominar a estas funciones como *my_init* y *my_cleanup*, bastaría con hacer

```
module_init(mi_init);
module_exit(mi_limpieza);
```

Observar que nuestro código debe incluir <linux/init.h> para utilizar *module_init* y *module_exit*. La ventaja de realizar esto es que cada función de inicialización y limpieza tiene su propio nombre en el kernel, lo que ayuda a la depuración.

Si escarbamos en los fuentes del kernel (versiones 2.2 y posteriores), probablemente veremos una forma diferente de declaración para la inicialización y limpieza de módulos, que se parece a:

```
static int __init mi_init(void)
{
  …
}
static void __exit mi_limpieza(void)
{
  …
}
```

El atributo `__init` (e `initdata`, para los datos), cuando se utiliza de esta forma, provoca que se descarte la función de inicialización, y se reclame su memoria, después la inicialización. Sin embargo, sólo funciona para manejadote empotrados; no tiene efecto en módulos. En vez de eso, `__exit` provoca la omisión de la función marcada cuando el manejador no esta construido como un módulo; de nuevo, no tiene efecto en módulos. El uso de `__init` puede reducir la cantidad de memoria utilizada por el kernel. No existe daño en marcar la función de

inicialización del módulo con `__init`, incluso pensando que no hay tampoco beneficio. La gestión de la inicialización de módulos no ha sido implementada aún, pero existe una posible mejora en el futuro.

Utilizando recursos

Un módulo no puede realizar su tarea sin el uso de recursos del sistema tales como memoria, puertos de E/S, memoria para E/S, y líneas de interrupción, además de canales DMA si utilizamos los viejos controladores DMA como ISA.

Nuestros programas obtienen memoria utilizando *kmalloc* y la liberan utilizando *kfree*. Estas funciones son similares a *malloc* y *free*, excepto que *kmalloc* tiene un argumento adicional, la prioridad. Normalmente, una prioridad de `GFP_KERNEL` o `GPF_SUER` bastará. El acrónimo GFP proviene de "get free page".

Puertos de E/S y memoria E/S

El papel de un manejador típico es, en la mayoría de los casos, es leer y escribir en los puertos de E/S y memoria de E/S. El acceso a los puertos de E/S y memoria de E/S[3] (colectivamente denominado *regiones de E/S*) se produce en la inicialización y durante el funcionamiento normal.

Desafortunadamente, no todas las arquitecturas de bus ofrecen una forma clara de identificar las regiones de E/S que pertenecen a cada dispositivo, y a veces el manejador debe adivinar dónde habita su región de E/S, o incluso probar el dispositivo leyendo o escribiendo en los "posibles" rangos de direcciones. Este problema es especialmente cierto en el bus ISA.

A pesar de las características (o falta de características) del bus que utiliza el dispositivo hardware, el manejador del dispositivo debe garantizar el acceso exclusivo a sus regiones de E/S al objeto de evitar las interferencias con otros manejadores.

Los diseñadores de Linux eligieron implementar un mecanismo de solicitud/liberación para las regiones de memoria, principalmente como medio para evitar colisiones entre diferentes dispositivos. El mecanismo ha sido ampliamente utilizado para puertos de E/S y se generalizó recientemente para la gestión general de asignación de recursos. Observar que este mecanismo es sólo una abstracción software que ayuda al mantenimiento del sistema, y puede, o no, ser aplicada por las características hardware. Por ejemplo, el acceso no autorizado a un puerto de E/S no produce ninguna condición de error equivalente a una "falta de segmentación" –el hardware no puede asegurar el registrado de puertos.

Podemos encontrar información sobre recursos registrados en los archivos */proc/ioports* y */proc/iomem*.

▪ Puertos

Un archivo */proc/ioports* típico sobre PC con el kernel 2.4 tiene el siguiente aspecto:

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : serial(set)
0300-031f : NE2000
0376-0376 : ide1
...
```

[3]Denominamos memoria de E/S a las áreas de memoria que residen en el dispositivo periférico para diferenciarlas de la RAM del sistema.

Cada entrada en el archivo especifica (en hexadeximal) un rango de los puertos bloqueados por el manejador o propiedad del dispositivo hardware. Este archivo se puede utilizar para evitar colisiones cuando se añade un dispositivo al sistema y se debe seleccionar un rango de E/S adecuando los puentes (*jumpers*). Aunque el hardware más moderno no utiliza puentes, el tema es aún relevante para dispositivos específicos o componentes industriales.

Pero lo que es más importante de este archivo es la estructura de datos que hay tras él. Cuando el manejador software para un dispositivo se inicializa, este puede conocer que rango de puertos están en uso; si el manejador necesita probar los puertos de E/S para detectar el nuevo dispositivo, será capaz de evitar probar los puertos que están ya en uso por otro dispositivo.

La prueba de ISA es una tarea arriesgada, varios manejadores distribuidos con el kernel oficial de Linux rechazan realizar pruebas cuando se cargan como módulos, para evitar el riesgo de destruir en sistema en ejecución al metiendo algo en los puertos donde puede existir hardware desconocido. Afortunadamente, las arquitecturas de bus modernas (y algunas antiguas) son inmunes a estos problemas.

La interfaz de programación utilizada para acceder a los registros de E/S se realiza con tres funciones:

```
int check_region(unsigned long start, unsigned long len);
struct resource *request_region(unsigned long start,
unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
```

La función *check_region* puede llamarse para ver si un rango de puertos esta disponible para asignarlo; retorna un código de error negativo (tal como `-EBUSY` o `-EINVAL`) si la respuesta es no. *request_region* asignará realmente el rango de puertos, devolviendo un puntero no nulo si la asignación tiene éxito. Los manejadores no necesitan usar o salvar el puntero real[4] devuelto – todo lo que necesitamos comprobar es que no es `NULL`. El código que sólo funciona con kernels 2.4 no necesitan para nada la función *check_region*; de hecho, es mejor no hacerlo, pues la cosas pueden cambiar entre las llamadas *chech_region* y *request_region*. Si deseamos generar código portable con viejos kernels debemos utilizar *chech_region* dado que *request_region* devuelve un void en versiones anteriores a 2.4. Nuestro manejador deberá llamar a *release_region* para liberar los puertos cuando finalice con ellos. Las tres funciones son realmente macros, y están definidas en `<linux/ioport.h>`.

La secuencia típica de registrado de puertos es la siguiente, como aparece en el ejemplo del manejador *skull* (la función skull_probe_hw no se muestra ya que contiene código específico del dispositivo):

```
#include <linux/ioport.h>
#include <linux/errno.h>
static int skull_detect(unsigned int port, unsigned int range)
{
 int err;

 if ((err = check_region(port,range)) < 0) return err; /* busy */
 if (skull_probe_hw(port,range) != 0) return -ENODEV; /* not found */
 request_region(port,range,"skull");     /* "Can't fail" */
 return 0;
}
```

Este código mira primer si el rango solicitado de puertos esta disponible; si no pueden ser asignados, no tiene sentido buscar el hardware. La asignación real de los puertos se difiere hasta que se conoce la existencia del dispositivo. La llamada *request_region* no debería fallar nunca; el kernel sólo carga un único módulo a la vez, por lo que no debería haber problemas con otros módulos "slipping in" y robando los puertos durante la fase de detección.

[4] El puntero actual sólo es utilizado cuando la función se invoca internamente por el subsistema gestor de recursos del kernel.

Cualquier puerto de E/S asignado al manejador debe finalmente ser liberado; skull realiza esto dentro de *cleanup_module*:

```
static void skull_release(unsigned int port, unsigned int range)
{
 release_region(port,range);
}
```

El enfoque solicitud/liberación de recursos es similar a la secuencia registro/desregistro descrita en las funciones anteriores y encaja bien en el esquema de implementación basado en `goto` antes esbozado.

Memoria

La información de memoria de E/S esta disponible en el archivo */proc/iomem*. A continuación se muestra una porción de este archivo tal como aparecen en un PC:

```
00000000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000fffff : System ROM
00100000-03feffff : System RAM
 00100000-0022c557 : Kernel code
 0022c558-0024455f : Kernel data
20000000-2fffffff : Intel Corporation 440BX/ZX - 82443BX/ZX Host bridge
68000000-68000fff : Texas Instruments PCI1225
68001000-68001fff : Texas Instruments PCI1225 (#2)
e0000000-e3ffffff : PCI Bus #01
e4000000-e7ffffff : PCI Bus #01
 e4000000-e4ffffff : ATI Technologies Inc 3D Rage LT Pro AGP-133

 ...
```

De nuevo, los valores aparecen en rangos en hexadecimal, y la cadena tras los dos puntos es el nombre del propietario de la región de E/S.

El registro de memoria de E/S se realiza de la misma forma que los puertos de E/S, dado que están realmente basados en el mismo mecanismo interno.

Para obtener y liberar el acceso a ciertas regiones de memoria de E/S, el manejador debe realizar las siguientes llamadas:

```
int check_mem_region(unsigned long start, unsigned long len);
int request_mem_region(unsigned long start, unsigned long len,
   char *name);
int release_mem_region(unsigned long start, unsigned long len);
```

Un manejador típico podrá conocer su propio rango de memoria de E/S, y la secuencia mostrada con anterioridad para puertos de E/S se reducirá a lo siguiente:

```
if (check_mem_region(mem_addr, mem_size)) { printk("drivername:
 memory already in use\n"); return -EBUSY; }
 request_mem_region(mem_addr, mem_size, "drivername");
```

Asignación de recursos en Linux 2.4

El mecanismo actual de asignación de recursos se introdujo en Linux 2.3.11 y suministra una forma flexible de controlar los recursos del sistema. Esta sección describe brevemente. Sin embargo, las funciones básicas para la asignación de recursos (*request_region* y otras) se implementan aún (mediante macros) y son utilizadas por razones de compatibilidad hacia atrás. La mayoría de programadores de módulos no necesitan conocer sobre lo que realmente esta ocurriendo debajo, pero será necesario en el desarrollo de manejadores más complejos.

La gestión de recursos es capaz de controlar recursos arbitrarios, y puede realizarlo de manera jerárquica. Los recursos conocidos globalmente (el rango de puertos de E/S, por ejemplo) puede subdividirse en subconjuntos más pequeños –por ejemplo, los recursos asociados con una ranura particular. Los manejadores individuales pueden a su vez subdivididas más.

Los rangos de recursos están descritos mediante una estructura `resource`, declarada en `<linux/ioport.h>`:

```
struct resource {
 const char *name;
 unsigned long start, end;
 unsigned long flags;
 struct resource *parent, *sibling, *child;
};
```

Los recursos del nivel más alto (raíz) se crean en tiempo de arranque. Por ejemplo, la estructura de recursos describen el rango de puertos de E/S se crea como sigue:

```
struct resource ioport_resource =
   { "PCI IO", 0x0000, IO_SPACE_LIMIT, IORESOURCE_IO };
```

Así, el nombre del recurso es `PCI IO`, y cubre un rango desde cero a `IO_SPACE_LIMIT`, que de acuerdo a la plataforma hardware en la que se ejecuta, puede ser 0xffff (16 bits del espacio de direcciones, como ocurre en el x86, IA-64, Alpha, M68k, y MIPS), `0xffffffff` (32 bits: SPARC, PPC, SH) ó `0xffffffffffffffff` (64 bits: SPARC64) .

Los subrangos de un recurso dado pueden crearse con *allocate_resource*. Por ejemplo, durante la inicialización de PCI se crea un nuevo recurso para una región que esta realmente asignada a un dispositivo físico. Cuando el código PCI lee esos puertos o asignaciones de memoria crea un nuevo recurso para estas regiones, y las asigna bajo *ioport_resource* ó *iomem_resource*.

Un manejador puede solicitar un subconjunto de un recurso particular (realmente un subrango del recurso global) y lo marca como ocupado llamando a `__request_region`, que devuelve un puntero a una nueva estructura de datos `struct resource` que describe el recurso que se esta solicitando (o devuelve `NULL` en cado de error). La estructura en siempre parte ya del árbol global de recursos, y el manejador no puede utilizarlo a voluntad.

Un lector interesado puede mirar los detalles en el código *kernel/resource.c*. La mayoría de los escritores de manejadores, sin embargo, están más que servidos con *request_region* y las otras funciones introducidas con anterioridad.

El mecanismo de capas produce un par de beneficios. Uno es que hace la estructura de E/S del sistema evidente dentro de las estructuras de datos del kernel. El resultado se muestra en /proa/ioports, por ejemplo:

```
e800-e8ff : Adaptec AHA-2940U2/W / 7890
e800-e8be : aic7xxx
```

El rango `e800-e8ff` esta asignado a una tarjeta Adaptec, que se ha identificado ella misma al manejador del bus PCI. El manejador `aic7xxx` ha solicitado gran parte del rango –en este caso, la parte correspondiente a los puertos reales de la tarjeta.

La otra ventaja de controlar los recursos de esta forma es que particiona el espacio de puertos en distintos subrangos que reflejan el hardware del sistema subyacente. Dado que el distribuidor de recursos no permite una asignación que cruce subrangos, puede bloquear a un manejador erróneo (o uno que busque hardware que no existe en el sistema) de la asignación de puertos que pertenecen a más rangos –incluso so algunos de estos puertos están desasignados en aquel momento.

Configuración automática y manual

Algunos parámetros que un manejador necesita pueden cambiar de un sistema a otro. Por ejemplo, el manejador debe conocer las direcciones de E/S reales del hardware, o los rangos de memoria (esto no es un problema con las interfaces de bus bien diseñadas y sólo se aplica a dispositivos ISA). A veces necesitamos pasar parámetros al manejador para ayudarlo a encontrar su propio dispositivo o para habilitar/deshabilitar ciertas características especiales.

Dependiendo del dispositivo, existen otros parámetros además de las direcciones de E/S que afectan al comportamiento de manejador, tales como la marca y versión del dispositivo. Ajustar el manejador con los valores correctos, es decir, configurarlo, es una de las tareas complicadas que necesitamos realizar durante la inicialización del dispositivo.

Básicamente hay dos formas de obtener los valores correctos: bien el usuario los especifica explícitamente o el manejador los autodetecta. Si bien, la autodetección es indudablemente el menor enfoque, la configuración por el usuario es el más fácil de implementar. Un equilibrio adecuado para un escritor de dispositivos es implementar la configuración automática cuando sea posible, mientras que permite la configuración de usuario como una opción para sobrescribir la autodetección. Una ventaja adicional de este enfoque es que el desarrollo inicial se puede realizar sin autodetección, especificando los parámetros en tiempo de carga, y se puede implementar la autodetección más tarde.

Muchos manejadores tienen también opciones de configuración que controlas aspectos de su funcionamiento. Por ejemplo, un manejador para adaptadores SCSI tiene a menudo opciones para controlar el uso "tagged command queuing", y los manejadores IDE permiten el control de usuario de las operaciones DMA. Así, incluso si nuestro manejador descansa enteramente en la autodetección para localizar el harware, podemos querer poner a disposición del usuario otras opciones de configuración.

Los valores de los parámetros pueden asignarse en tiempo de cargo con *insmod* o *modprobe*; la última puede también leer los parámetros asignados en un archivo de configuración (típicamente */etc/modules.conf*). Las órdenes aceptan la especificación de valores enteros o cadenas en la línea de órdenes. Así, si nuestro módulo puede soportar un parámetro entero denominado skull_ival y un parámetro skull_sval, los parámetros pueden ajustarse en tiempo de carga del módulo de la forma:

```
insmod skull skull_ival=666 skull_sval="el mejor"
```

Sin embargo, antes de que *insmod* pueda cambiar los parámetros del módulo, este debe hacerlos disponibles. Los parámetros se declaran con la macro `MODULE_PARM`, que esta definida en *module.h*. `MODULE_PARM` toma dos parámetros: el nombre de la variable, y una cadena que describe el tipo. La macro debe situarse fuera de cualquier función y se encuentra típicamente cerca de la cabecera del archivo fuente. Los dos parámetros mencionados anteriormente pueden declararse con las siguientes líneas:

```
int skull_ival=0;
char *skull_sval;

MODULE_PARM (skull_ival, "i");
MODULE_PARM (skull_sval, "s");
```

Actualmente, hay cinco tipos soportados para parámetros del módulo: `b`, un byte; `h`, dos bytes (short); `i`, un entero; `l`, un long; y `s`, una cadena. En el caso de valores cadena, se puede definir una variable puntero; *insmod* puede asignar la memoria para el parámetro suministrado por el usuario y ajustar adecuadamente la variable. Un valor entero precediendo el tipo indica una matriz de una longitud dada; dos números, separados por un guión, dan los valores mínimo y máximo. Un ejemplo, podemos declarar una matriz que deba tener al menos dos, y no más de cuatro, como:

```
int skull_array[4];
MODULE_PARM (skull_array, "2-4i");
```

También existe la macro `MODULE_PARM_DESC`, que permite al programador suministrar una descripción para un parámetro del módulo. Esta descripción se almacena en el archivo objeto; puede verse como una herramienta como *objdump*, y puede visualizarse mediante herramientas de administración de sistemas automáticas. Un ejemplo:

```
int base_port = 0x300;
MODULE_PARM (base_port, "i");
```

```
MODULE_PARM_DESC (base_port,"Puerto E/S base (defecto 0x300)");
```

Todos los parámetros del módulo deben de tener un valor por defecto; *insmod* podrá cambiar los valores sólo si se lo indica el usuario explícitamente. El módulo puede comprobar los parámetros explícitos probando los parámetros frente a los valores por defecto. La configuración automática, entonces, puede diseñarse para funcionar de esta forma: si las variables de configuración tienen sus valores por defecto, realizan autodetección; en otro caso, mantiene el valor actual. Al objeto de que funciones esta técnica, el valor "por defecto" deber ser tal que el usuario nunca desee realmente especificarlo en tiempo de carga.

El siguiente código muestra cómo *skull* autodetecta la dirección del puerto de un dispositivo. En este ejemplo, la autodetección se utiliza para mirar múltiples dispositivos, mientras que la configuración manual esta restringida a un único dispositivo. La función *skull_detect* la vimos en "Puertos", mientras que *skull_init_board* esta a cargo de la inicialización específica del dispositivo y no se muestra:

```
/*
 * port ranges: the device can reside between
 * 0x280 and 0x300, in steps of 0x10. It uses 0x10 ports.
 */
#define SKULL_PORT_FLOOR 0x280
#define SKULL_PORT_CEIL 0x300
#define SKULL_PORT_RANGE 0x010

/*
 * the following function performs autodetection, unless a specific
 * value was assigned by insmod to "skull_port_base"
 */

static int skull_port_base=0; /* 0 forces autodetection */
MODULE_PARM (skull_port_base, "i");
MODULE_PARM_DESC (skull_port_base, "Base I/O port for skull");


static int skull_find_hw(void) /* returns the # of devices */
{
 /* base is either the load-time value or the first trial */
 int base = skull_port_base ? skull_port_base
        : SKULL_PORT_FLOOR;
 int result = 0;

 /* loop one time if value assigned; try them all if autodetecting */
 do {
        if (skull_detect(base, SKULL_PORT_RANGE) == 0) {
         skull_init_board(base);
         result++;
        }
        base += SKULL_PORT_RANGE; /* prepare for next trial */
 }
 while (skull_port_base == 0 && base < SKULL_PORT_CEIL);

 return result;
}
```

Si las variables de configuración son utilizadas sólo dentro del manejador (no son publicadas en la tabla de símbolos del kernel), el escritor del manejador puede hacer la vida un poco más fácil para el usuario eliminado los prefijos de los nombres de las variables. Los prefijos no tienen mucho sentido para el usuario excepto tecleado extra.

Por completitud, existen otras tres macros que sitúan documentación dentro del archivo objeto. Estas son:

`MODULE_AUTHOR (nombre)` Pone el nombre del autor en el archivo objeto.

`MODULE_DESCRIPTION (des)` Pone una descripción del módulo.

`MODULE_SUPPORTED_DEVICE (dev)` Sitúa una entrada que describe qué dispositivo esta soportado por este módulo. Los comentarios en las fuentes del kernel sugieren que este parámetro puede utilizarse finalmente para ayudar con la carga automatizada del módulo, pero en este instante no se realiza tal uso.

Cambios en la gestión de recursos

El nuevo esquema de gestión de recursos trae unos pocos problemas de portabilidad si deseamos escribir un manejador que se ejecute en versiones anteriores a la 2.4. Esta sección discute los problemas de portabilidad que podemos encontrar y cómo la cabecera *sysdep.h* intenta esconderlos.

El cambio más visible en el nuevo código de gestión de recursos es la inclusión de la función *request_mem_region* y relacionadas. Su papel esta limitado al acceso a la base de datos de memoria de E/S, si realizar operaciones específicas sobre cualquier hardware. Así, lo que podemos hacer con kernels anteriores es simplemente no invocar las funciones. La cabecera *sysdep.h* consigue simplemente esto definiendo las funciones como macros que retornan 0 para los kernel anteriores al 2.4.

Otra diferencias entre el 2.4 y los anteriores es el prototipo real de *request_region* y las funciones relacionadas.

Los kernel anteriores al 2.4 declara *request_region* y *release_region* como funciones que retornan `void` (así fuerza el uso de *check_region* de antemano). La nueva implementación, más correctamente, tiene funciones que devuelven un puntero de forma que puede señalarse una condición de error (haciendo así a *check_region* bastante inútil). EL valor del puntero real no será normalmente útil para el código de manejador salvo para probar el valor `NULL`, que significa que la solicitud ha fallado.

Si deseamos ahorrar algunas líneas de código y no estamos preocupados por la compatibilidad hacia atrás, podemos explotar las nuevas llamadas al sistema y evitar utilizar *check_region*, liberando la región de memoria y devolviendo éxito si la solicitud se satisface; la sobrecarga es despreciable dado que ninguna de esas funciones se invoca nunca desde una sección de código crítica en tiempo.

Si deseamos transportabilidad, podemos poner la secuencia de llamadas que sugerimos anteriormente en el capítulo e ignorar los valores de retorno de *request_region* y *release_region*. De todas formas, *sysdep.h* declara ambas funciones como macros para devolver 0 (éxito), así podemos hacer tanto que sea portable y comprobar el valor de retorno de cada función.

La última diferencia en el registro de E/S entre la versión 2.4 y anteriores es que los tipos de datos utilizados para los argumentos `start` y `len`. Mientras los nuevos kernel siempre utilizan `unsigned long`, los viejos kernels utilizan tipos más cortos. Este cambio no tiene efecto sobre la portabilidad del manejador.

Sustituyendo los printk's

Al inicio de esta apartado, veiamos como printk imprime mensajes en la consola del sistema pero no, por ejemplo, en una terminal X. En ocasiones, no interesa que un módulo pueda enviar mensajes a cualquier tty desde la que se instale el módulo.

La forma de hacer esto es utilizar el puntero al proceso actual, `current`, para obtener la estructura `tty` del proceso actual. Después, miramos en esta estructura el puntero que apunta a la función de escritura, que nosostros utilizaremos para escribir una cadena en el tty. Un ejemplo:

```
/*print_string.c - Send output to the tty you're running on, regardless of
whether it's through X11, telnet, etc.  We do this by printing the string to
the tty associated with the current task.
 */
#include <linux/kernel.h>
#include <linux/module.h>
```

```
#include <linux/sched.h>    // For current
#include <linux/tty.h>      // For the tty declarations
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman");

void print_string(char *str)
{
   struct tty_struct *my_tty;
   my_tty = current->tty;           // The tty for the current task

/* If my_tty is NULL, the current task has no tty you can print to (this is
possible, for example, if it's a daemon).  If so, there's nothing we can do.*/
   if (my_tty != NULL) {

      /* my_tty->driver is a struct which holds the tty's functions, one of
which (write) is used to write strings to the tty.  It can be used to take a
string either from the user's memory segment or the kernel's memory segment.
The function's 1st parameter is the tty to write to, because the same function
would normally be used for all tty's of a certain type.  The 2nd parameter
controls whether the function receives a string from kernel memory (false, 0)
or from user memory (true, non zero).  The 3rd parameter is a pointer to a
string. The 4th parameter is the length of the string.
       */
      (*(my_tty->driver).write)(
         my_tty,                 // The tty itself
         0,                      // We don't take the string from user space
         str,                    // String
         strlen(str));           // Length

      /* ttys were originally hardware devices, which (usually) strictly
followed the ASCII standard.  In ASCII, to move to a new line you need two
characters, a carriage return and a line feed.  On Unix, the ASCII line feed is
used for both purposes - so we can't just use \n, because it wouldn't have a
carriage return and the next line will start at the column right after the line
feed. BTW, this is why text files are different between Unix and MS Windows.
In CP/M and its derivatives, like MS-DOS and MS Windows, the ASCII standard was
strictly adhered to, and therefore a newline requirs both a LF and a CR.
       */
      (*(my_tty->driver).write)(my_tty, 0, "\015\012", 2);
   }
}

int print_string_init(void)
{
   print_string("The module has been inserted.  Hello world!");
   return 0;
}

void print_string_exit(void)
{
   print_string("The module has been removed.  Farewell world!");
}

module_init(print_string_init);
module_exit(print_string_exit);
```

¿Cómo interceptar las llamadas al sistema?

En este apartado vamos a "abusar" del esquema de módulos kernel. Normalmente, estos se utilizan para extender la funcionalidad, en especial de los manejadores de dispositivos. En este apartado, vamos a hacer algo diferente como es interceptar una llamada al sistema y modificarla de forma que el sistema cambie el comportamiento de diversas órdenes o llamadas al sistema.

Las llamadas al sistema están definidas en el archivo */usr/include/sys/syscall.h*. A continuación mostramos un estracto de este archivo:

```
#ifdef _SYS_CALL_H
#define _SYS_CALL_H

#define SYS_setup     0     /*utilizada solo por init*/
#define SYS_exit1
#define SYS_fork2
#define SYS_read3
…
#endif          /*<sys/syscall.h> */
```

Cada llamada esta definida por un número (ver listado) que es el que se utiliza realmente para realizar la llamada. El kernel utiliza la interrupción software 0x80 para manejar las llamadas. El número de la llamada y sus argumentos se deposita en los registros apropiados de la máquina. El número de la llamada se utiliza como índice en un vector del kernel denominado `sys_call_table[]`. Esta función hace corresponder a cada número de llamada la correspondiente función kernel que lo realiza.

Vamos a ver como se puede interceptar una llamada al sistema. Para ellos lo que tenemos que capturar el puntero de la correspondiente entrada de `sys_call_table[]` y ajustarlo por el puntero a la función nueva que sustituye a la llamada. El siguiente módulo hace imposible a cualquier usuario del sistema donde se instala la creación de un directorio.

```
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

extern void* sys_call_table[];       /*para acceder a sys_call_table*/

int (*orig_mkdir)(const char *path); /*la llamada original*/


int hacked_mkdir(const char *path)
{
 return 0;                           /*la nueva llamada no hace nada*/
}

int init_module(void)                /*iniciar modulo*/
{
 orig_mkdir=sys_call_table[SYS_mkdir];
 sys_call_table[SYS_mkdir]=hacked_mkdir;
 return 0;
}

void cleanup_module(void)            /*retirar modulo*/
{
 sys_call_table[SYS_mkdir]=orig_mkdir; /*establece mkdir a la original */
}
```

Consideraciones importantes de nuestro sistema

Para la realización de las prácticas, vamos a usar los equipos de la Escuela que tiene instalada distribución Red Hat. Esta distribución, por motivos de seguridad, no exporta la tabla de llamadas al sistema.

Para solventar este inconveniente deberemos exportarla nosotros mismos. Los pasos a seguir son:

1. Editamos */usr/src/linux-2.4/kernel/ksyms.c* y añadimos la línea
   `EXPORT_SYMBOL(sys_call_table)`
2. Recompilamos e instalamos el nuevo kernel.
3. Arrancamos en local con la nueva imagen del núcleo.

Además en nuestro sistema, en lugar de las llamadas al sistema `getpid`, etc. deberemos usar las funciones `getpid32`, etc.

4 Bibliografía y referencias de interés

- Versión 2.4 del kernel:

- P. J. Salzamn, y O. Pomerantz, "The Linux Kernel Module Programming Guide", 2001, en http://tldp.org/LDP/lkmpg/.
- Pragmatic /THC, "LKM – Loadable Linux Kernel Modules", 1999, en http://www.thc.org/papers/LKM_HACKING.html.
- A. Rubini y J. Corbet, "*Linux Device Drivers* (2nd ed.)", O`Reilly, 2001. Podemos verlo en la dirección http://www.xml.com/ldd/chapter/book/.

- Versión 2.6 del kernel:

– B. Henderson, Linux Loadable Kernel Modules HOWTO, 2006-09-24, disponible en http://tldp.org/HOWTO/Module-HOWTO/.
– A. Rubini, J. Corbet, y G. Kroah-Hartman, *Linux Device Drivers (3/e)*, O'Reilly, 2005, disponible en http://lwn.net/Kernel/LDD3/.
– S. Venkateswaran, *Essential Linux Device Drivers*, Prentice Hall, 2008. Disponible en ftp://air.zz.com/_manuals/Linux/Venkateswaran%20-%20Essential%20Linux%20Device%20Drivers%20%28Prentice,%202008%29.pdf.