Guía de Programación de Módulos del Núcleo Linux

1999 Ori Pomerantz

Versión 1.1.0, 26 Abril 1999.

Este libro trata sobre cómo escribir Módulos del Núcleo Linux. Es útil, eso espero, para programadores que conocen C y quieren aprender cómo escribir módulos del núcleo. Está escrito como un manual de instrucciones ‘COMO’ (How-To), con ejemplos de todas las técnicas importantes.

Como este libro toca en muchos puntos del diseño del núcleo, no se supone que es tan completo como necesario — hay otros libros con este propósito, impresos y en el proyecto de documentación de Linux.

Names of all products herein are used for identification purposes only and are trademarks and/or registered trademarks of their respective owners. I make no claim of ownership or corporate association with the products or companies that own them.

Ori Pomerantz
Apt. #1032
2355 N Hwy 360
Grand Prairie
TX 75050
USA
E-mail: mpg@simple-tech.com

Los nombres de todos los productos adjuntos se utilizan únicamente con el propósito de identificación y son marcas registradas de sus respectivos propietarios. No he hecho ninguna demanda de propiedad o asociación corporativa con los productos o compañías que las poseen.

Ori Pomerantz
Apt. #1032
2355 N Hwy 360
Grand Prairie
TX 75050
USA
E-mail: mpg@simple-tech.com

Índice General

Capítulo 0

Introducción

Entonces, quieres escribir un módulo del núcleo. Conoces C, has escrito un número de programas normales que se ejecutan como procesos, y ahora quieres estar donde está la verdadera acción, donde un simple puntero salvaje puede destruir tu sistema de ficheros y donde un volcado de memoria (core dump) significa un reinicio de la máquina.

Bien, bienvenido al club. Yo sólo he tenido un golpe de un puntero salvaje en un directorio importante bajo DOS (gracias que, ahora está en el **D**ead **O**perating **S**ystem; Sistema Operativo Muerto), y no veo porque vivir bajo Linux debería ser algo seguro.

Peligro: He escrito esto y verificado el programa bajo versiones 2.0.35 y 2.2.3 del núcleo funcionando en un Pentium. Para la mayor parte, debería funcionar en otras CPUs y en otras versiones del núcleo, ya que ellos son 2.0.x o 2.2.x, pero no puedo prometer nada. Una excepción es el capítulo 11, el cual no debería funcionar en ninguna arquitectura excepto x86.

0.1 Quién debería de leer esto

Este documento es para la gente que quiere escribir módulos del núcleo. Aunque trataré en varios sitios sobre cómo las cosas son realizadas en el núcleo, este no es mi propósito. Hay suficientes buenas fuentes las cuales hacen un trabajo mejor que el que yo pudiera haber hecho.

Este documento también es para gente que sabe como escribir módulos del núcleo, pero que no se han adaptado a la versión 2.2 de éste. Si eres una de estas personas, te sugiero que mires en el apéndice A para ver todas las diferencias que he encontrado mientras actualizaba los ejemplos. La lista no está cerca de ser amplia, pero creo que cubre la mayoría de las funcionalidades básicas y será suficiente para que empieces.

El núcleo es una gran pieza de programación, y creo que los programadores deberían leer al menos algunos ficheros fuentes del núcleo y entenderlos. Habiendo dicho esto, creo en el valor de jugar primero con el sistema y hacer preguntas más tarde. Cuando aprendo un nuevo lenguaje de programación, no empiezo leyendo el código de la biblioteca, pero si escribiendo un pequeño programa ‘Hola, mundo’. No veo por que el jugar con el núcleo tendría que ser diferente.

0.2 A Destacar en el estilo

Me gusta poner tantas bromas como sea posible en mi documentación. Estoy escribiendo porque me gusta, y asumo que la mayoría de los que estais leyendo esto es por el mismo motivo. Si quieres saltarte este punto, ignora todo el texto normal y lee el código fuente. Prometo poner todos los detalles importantes en destacado.

0.3 Cambios

0.3.1 Nuevo en la versión 1.0.1

1. **Sección de Cambios**, 0.3.
2. **Cómo encontrar el número menor del dispositivo**, 2.
3. **Arreglada la explicación de la diferencia entre caracteres y ficheros de dispositivo**, 2
4. **Makefiles para los módulos del núcleo**, 1.1.
5. **Multiprocesamiento Simétrico**, 12.
6. **Un Capítulo de 'Malas Ideas'** , 13.

0.3.2 Nuevo en la versión 1.1.0

1. **Soporte para la versión 2.2 del núcleo**, todo sobre el sitio.
2. **Ficheros de Código fuente multi-núcleo**, 2.1.
3. **Cambios entre 2.0 y 2.2**, A.
4. **Módulos de Núcleo en Múltiples Ficheros Fuente**, 1.2.
5. **Sugerencia de no dejar módulos que implementan llamadas al sistema que pueden ser quitadas**, 7.

0.4 Agradecimientos

Me gustaría dar gracias a Yoav Weiss por muchas ideas útiles y discusiones, también por encontrar fallos en este documento antes de su publicación. Por supuesto, cualesquiera errores remanentes son puramente fallos mios.

El esqueleto TEX para este libro fue disimuladamente robado de la guía 'Linux Installation and Getting Started', donde el trabajo TEX fue realizado por Matt Welsh.

Mi gratitud a Linus Torvalds, Richard Stallman y toda la otra gente que me hizo posible ejecutar un sistema operativo de calidad en mi computadora y obtener el código fuente sin decirlo (si, correcto — ¿entonces por qué lo dije?).

0.4.1 Para la versión 1.0.1

No pude nombrar a todo el mundo que me escribió un email, y si te he dejado fuera lo siento por adelantado. Las siguientes personas fueron especialmente útiles:

- **Frodo Looijaard desde Holanda** Por un montón de sugerencias útiles, y sobre información sobre los núcleos 2.1.x.
- **Stephen Judd de Nueva Zelanda** correcciones de deletreo.
- **Magnus Ahltorp de Suiza** Corrigiendo un fallo como una mina sobre la diferencia de dispositivos de bloque y de carácter.

0.4.2 Para la versión 1.1.0

- **Emmanuel Papirakis de Quebec, Canada** Por portar todos los ejemplos a la versión 2.2 del núcleo.
- **Frodo Looijaard de Holanda** Por decirme cómo crear un módulo del núcleo con múltiples ficheros (1.2).

Por supuesto, cualesquiera errores remanentes son míos, y si piensas que ellos han hecho el libro inutilizable eres bienvenido a aplicar un total reembolso del dinero que has pagado por él.

Capítulo 1

Hola, mundo

Cuando el primer hombre de las cavernas cinceló el primer programa en las paredes de la primera computadora de las cavernas, era un programa para imprimir la cadena de caracteres ‘Hola, mundo’ en las pinturas de los Antílopes. Los libros de texto de los romanos empezaban con el programa ‘Salut, Mundi’. No sé qué le pasa a la gente que rompe con esta tradición, y creo que es más seguro no saberlo.

Un módulo del núcleo tiene que tener por lo menos dos funciones: `init_module` la cual se llama cuando el módulo es insertado en el núcleo, y `cleanup_module` la cual se llama justo antes de ser quitado. Típicamente, `init_module` registra un manejador para algo en el núcleo, o reemplaza una de las funciones del núcleo con su propio código (usualmente código para hacer algo cuando alguien llama a la función original). La función `cleanup_module` se supone que deshace lo que `init_module` ha hecho, por lo tanto el módulo puede ser descargado de una forma segura.

hello.c

```
/* hello.c
 * Copyright (C) 1998 by Ori Pomerantz
 *
 * "Hello, world" - la versión módulo del núcleo.
 */

/* Los archivos de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos realizando trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */



/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif



/* Inicializa el módulo */
int init_module()
{
  printk("Hola, mundo - este es el núcleo hablando\n");
```

```
  /* Si retornamos un valor distinto de cero, significa
   * que init_module falló y el módulo del núcleo
   * no puede ser cargado */
  return 0;
}


/* Limpieza - deshacemos todo aquello que hizo init_module */
void cleanup_module()
{
  printk("La vida de un módulo del núcleo es corta\n");
}
```

1.1 Makefiles para los Módulos del Núcleo

Un módulo del núcleo no es un ejecutable independiente, sino un fichero objeto el cual es enlazado en el núcleo en tiempo de ejecución. Como resultado, deberían de ser compilados con la flag `-c`. También, todos los módulos del núcleo deberían de ser compilados con ciertos símbolos definidos.

- `__KERNEL__` — Esto le dice a los ficheros de cabeceras que este código se ejecutará en modo kernel (núcleo), y no como parte de un proceso de usuario (modo usuario).
- `MODULE` — Esto le dice a los ficheros de cabeceras que le den las apropiadas definiciones para un módulo del núcleo.
- `LINUX` — Técnicamente hablando, esto no es necesario. En todo caso, si quieres escribir un módulo del núcleo de forma seria el cual compile en más de un sistema operativo, serás feliz si lo haces. Esto te permitirá hacer compilación condicional en las partes que son dependientes del Sistema Operativo.

Hay otros símbolos que tienen que ser incluidos, o no, dependiendo de las flags con las que haya sido compilado el núcleo. Si no estás seguro de como fue compilado el núcleo, mira en `/usr/include/linux/config.h`

- `__SMP__` — Procesamiento Simétrico. Esto tiene que estar definido si el núcleo fue compilado para soportar multiprocesamiento simétrico (incluso si sólo se esta ejecutando en una CPU). Si usas Multiprocesamiento Simétrico, hay otras cosas que necesitas hacer (ver capítulo 12).
- `CONFIG_MODVERSIONS` — Si CONFIG_MODVERSIONS estaba habilitado, necesitas tenerlo definido cuando compiles el módulo del núcleo e incluir `/usr/include/linux/modversions.h`. Esto también puede ser realizado por el propio código.

Makefile

```
# Makefile para un módulo básico del núcleo

CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: hello.c /usr/include/linux/version.h
$(CC) $(MODCFLAGS) -c hello.c
echo insmod hello.o para conectarlo
```

```
echo rmmod hello para desconectarlo
echo
echo X y la programación del núcleo no se mezclan.
echo Haz insmod y rmmod desde fuera de X.
```

Entonces, ahora la única cosa que queda es hacer `su` a root (¿no compilaste como root, verdad?) Viviendo en el límite[1]...), y entonces haz `insmod hello` y `rmmod hello` para contentar a tu corazón. Mientras lo haces, nota que tu nuevo módulo del núcleo está en `/proc/modules`.

El motivo por el que Makefile recomienda no hacer `insmod` desde X es porque cuando el núcleo tiene que imprimir un mensaje con `printk`, lo envía a la consola. Cuando no utilizas X, va al terminal virtual que estás usando (el que escogiste con Alt-F<n>) y lo ves. Si utilizas X, en cambio, hay dos posibilidades. Si tienes una consola abierta con `xterm -C`, en cuyo caso la salida será enviada allí, o no, en cuyo caso la salida irá al terminal virtual 7 — el que es 'cubierto' por X.

Si tu núcleo se vuelve inestable será más probable que cojas los mensajes de depuración sin las X. Fuera de X, `printk` va directamente desde el núcleo a la consola. En X, de la otra forma, los `printk`s van a un proceso de modo usuario (`xterm -C`). Cuando este proceso recibe tiempo de CPU, se supone que lo envía al proceso servidor de X. Entonces, cuando el servidor X recibe la CPU, se supone que lo muestra — pero un núcleo inestable usualmente significa que el sistema va a romper o a reiniciar, por lo tanto no quieres que se retrasen los mensajes de error, los cuales quizás expliquen lo que hiciste mal, para que no vuelva a pasar.

1.2 Múltiples Ficheros de Módulos del Núcleo

A veces tiene sentido dividir el módulo del núcleo entre varios ficheros de código. En este caso, necesitas hacer lo siguiente:

1. En todos lo ficheros fuente menos en uno, añade la línea `#define __NO_VERSION__`. Esto es importante porque `module.h` normalmente incluye la definición de `kernel_version`, una variable global con la versión del núcleo para la que es compilado el módulo. Si necesitas `version.h`, necesitas incluirla, porque `module.h` no lo hará por ti con `__NO_VERSION__`.
2. Compila todos los ficheros fuente de la forma normal.
3. Combina todos los ficheros objetos en uno. Bajo x86, hazlo con `ld -m elf_i386 -r -o <nombre del módulo>.o <1 fichero fuente>.o <2 fichero fuente>.o`.

Aquí hay un ejemplo de este módulo del núcleo.

start.c

```
/* start.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hola, mundo" - la versión módulo del núcleo.
 * Este fichero incluye justamente la rutina de comienzo
 */

/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */
```

[1] El motivo por el que prefiero no compilarlo como root es que cuanto menor sea hecho como root más seguro estará el equipo. Yo trabajo en seguridad informática, por lo tanto estoy paranoico

```
/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif



/* Inicializa el módulo */
int init_module()
{
  printk("Hola, mundo - este es el núcleo hablando\n");

  /* Si retornamos un valor distinto de cero, significa
   * que init_module falló y el módulo del núcleo
   * no puede ser cargado */
  return 0;
}
```

stop.c

```
/* stop.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hola, mundo" - la versión módulo del núcleo. Este
 * fichero incluye justamente la rutina de parada.
 */

/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */

#define __NO_VERSION__      /* Este no es "el" fichero
                             * del módulo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

#include <linux/version.h>   /* No incluido por
                              * module.h debido
                              * a __NO_VERSION__ */



/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
```

```
#include <linux/modversions.h>
#endif



/* Limpieza - deshacemos todo aquello que hizo init_module */
void cleanup_module()
{
  printk("La vida de un módulo del núcleo es corta\n");
}
```

Makefile

```
# Makefile para un módulo multiarchivo del núcleo

CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o: start.o stop.o
ld -m elf_i386 -r -o hello.o start.o stop.o

start.o: start.c /usr/include/linux/version.h
$(CC) $(MODCFLAGS) -c start.c

stop.o: stop.c /usr/include/linux/version.h
$(CC) $(MODCFLAGS) -c stop.c
```

Capítulo 2

Ficheros de Dispositivos de Carácter

De este modo, ahora somos valientes programadores del núcleo y sabemos como escribir módulos que no hacen nada. Nos sentimos bien con nosotros mismos y mantenemos nuestros corazones en tensión. Pero de algún modo sentimos que falta algo. Los módulos catatónicos no son muy divertidos.

Hay dos formas principales para que un módulo del núcleo se comunique con los procesos. Uno es a través de los ficheros de dispositivos (como los ficheros en el directorio `/dev`, la otra es usar el sistema de ficheros proc. Porque uno de los principales motivos para escribir algo en el núcleo es soportar algún tipo de dispositivo hardware, empezaremos con los ficheros de dispositivos.

El propósito original de los ficheros de dispositivo es permitir a los procesos comunicarse con los controladores de dispositivos en el núcleo, y a través de ellos con los dispositivos físicos (módems, terminales, etc.). La forma en la que esto es implementado es la siguiente.

Cada controlador de dispositivo, el cual es responsable de algún tipo de hardware, es asignado a su propio número mayor. La lista de los controladores y de sus números mayores está disponible en `/proc/devices`. Cada dispositivo físico administrado por un controlador de dispositivo es asignado a un número menor. El directorio `/dev` se supone que incluye un fichero especial, llamado fichero de dispositivo, para cada uno de estos dispositivos, tanto si está o no realmente instalado en el sistema.

Por ejemplo, si haces `ls -l /dev/hd[ab]*`, verás todas las particiones de discos duros IDE que quizás estén conectadas a una máquina. Nota que todos ellos usan el mismo número mayor, 3, pero el número menor cambia de uno a otro *Nota: Esto asume que estás usando una arquitectura PC. No se como son conocidos otros dispositivos en Linux ejecutándose en otras arquitecturas.*

Cuando tu sistema fue instalado, todos esos ficheros de dispositivos fueron creados por el comando `mknod`. No existe un motivo técnico por el cual tienen que estar en el directorio `/dev`, es sólo una convención útil. Cuando creamos un fichero de dispositivo con el propósito de prueba, como aquí para un ejercicio, probablemente tenga más sentido colocarlo en el directorio en donde compilas el módulo del núcleo.

Los dispositivos están divididos en dos tipos: los dispositivos de carácter y los dispositivos de bloque. La diferencia es que los dispositivos de bloque tienen un búfer para las peticiones, por lo tanto pueden escoger en qué orden las van a responder. Esto es importante en el caso de los dispositivos de almacenamiento, donde es más rápido leer o escribir sectores que están cerca entre si, que aquellos que están más desperdigados. Otra diferencia es que los dispositivos de bloque sólo pueden aceptar bloques de entrada y de salida (cuyo tamaño puede variar de acuerdo al dispositivo), en cambio los dispositivos de carácter pueden usar muchos o unos pocos bytes como ellos quieran. La mayoría de los dispositivos en el mundo son de carácter, porque no necesitan este tipo de *buffering*, y no operan con un tamaño de bloque fijo. Se puede saber cuando un fichero de dispositivo es para un dispositivo de carácter o de bloque mirando el primer carácter en la salida de `ls -l`. Si es 'b' entonces es un dispositivo de bloque, y si es 'c' es un dispositivo de carácter.

Este módulo está dividido en dos partes separadas: La parte del módulo que registra el dispositivo y la parte del controlador del dispositivo. La función `init_module` llama a `module_register_chrdev` para añadir el controlador de dispositivo a la tabla de controladores de dispositivos de carácter del núcleo. También retorna el número mayor a ser usado por el controlador. La función `cleanup_module` libera el dispositivo.

Esto (registrar y liberar algo) es la funcionalidad general de estas dos funciones. Las cosas en el núcleo no funcionan por su propia iniciativa, como los procesos, sino que son llamados, por procesos a través de las llamadas al sistema, o por los dispositivos hardware a través de las interrupciones, o por otras partes del núcleo (simplemente llamando a funciones específicas). Como resultado, cuando añades código al núcleo, se supone que es para registrarlo como parte de un manejador o para un cierto tipo de evento y cuando lo quitas, se supone que lo liberas.

El controlador del dispositivo se compone de cuatro funciones device_<acción>, que son llamadas cuando alguien intenta hacer algo con un fichero de dispositivo con nuestro número mayor. La forma en la que el núcleo sabe como llamarlas es a través de la estructura `file_operations`, `Fops`, la cual se dio cuando el dispositivo fue registrado, e incluye punteros a esas cuatro funciones.

Otro punto que necesitamos recordar aquí es que podemos permitir que el módulo del núcleo sea borrado cuando *root quiera*. El motivo es que si el fichero del dispositivo es abierto por un proceso y entonces quitamos el módulo del núcleo, el uso del fichero causaría una llamada a la posición de memoria donde la función apropiada (read/write) usada tendría que estar. Si tenemos suerte, ningún otro código fue cargado allí, y obtendremos un feo mensaje. Si no tenemos suerte, otro módulo del núcleo fue cargado en la misma posición, lo que significará un salto en el medio de otra función del núcleo. El resultado sería imposible de predecir, pero no sería positivo.

Normalmente, cuando no quieres permitir algo, devuelves un código de error (un número negativo) desde la función que se supone que lo tendría que hacer. Con `cleanup_module` esto es imposible porque es una función void. Una vez que se llama a `cleanup_module`, el módulo está muerto. En todo caso, hay un contador que cuenta cuantos otros módulos del núcleo están usando el módulo, llamado contador de referencia (que es el último número de la linea en `/proc/modules`). Si este número no es cero, `rmmod` fallará. La cuenta de referencia del módulo está disponible en la variable `mod_use_count_`. Como hay macros definidas para manejar esta variable (`MOD_INC_USE_COUNT` y `MOD_DEC_USE_COUNT`), preferimos usarlas, mejor que usar `mod_use_count_` directamente, por lo tanto será más seguro si la implementación cambia en el futuro.

chardev.c

```
/* chardev.c
 * Copyright (C) 1998-1999 by Ori Pomerantz
 *
 * Crea un dispositivo de carácter (sólo lectura)
 */

/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Para dispositivos de carácter */
#include <linux/fs.h>       /* Las definiciones de dispositivos
                             * de carácter están aquí */
#include <linux/wrapper.h>  /* Un envoltorio que
                             * no hace nada actualmente,
                             * pero que quizás ayude para
```

```
                            * compatibilizar con futuras
                            * versiones de Linux */


/* En 2.2.3 /usr/include/linux/version.h incluye
 * una macro para esto, pero 2.0.35 no lo hace - por lo
 * tanto lo añado aquí si es necesario */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif


/* Compilación condicional. LINUX_VERSION_CODE es
 * el código (como KERNEL_VERSION) de esta versión */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for put_user */
#endif



#define SUCCESS 0


/* Declaraciones de Dispositivo ************************** */

/* El nombre de nuestro dispositivo, tal como aparecerá
 * en /proc/devices */
#define DEVICE_NAME "char_dev"


/* La máxima longitud del mensaje desde el dispositivo */
#define BUF_LEN 80

/* >Está el dispositivo abierto correctamente ahora? Usado para
 * prevenir el acceso concurrente en el mismo dispositivo */
static int Device_Open = 0;

/* El mensaje que el dispositivo dará cuando preguntemos */
static char Message[BUF_LEN];

/* >Cuánto más tiene que coger el proceso durante la lectura?
 * Útil si el mensaje es más grande que el tamaño
 * del buffer que cogemos para rellenar en device_read. */
static char *Message_Ptr;


/* Esta función es llamada cuando un proceso
 * intenta abrir el fichero del dispositivo */
static int device_open(struct inode *inode,
       struct file *file)
{
  static int counter = 0;
```

```
#ifdef DEBUG
  printk ("Dispositivo abierto(%p,%p)\n", inode, file);
#endif

  /* Esto es como coger el número menor del dispositivo
   * en el caso de que tengas más de un dispositivo físico
   * usando el controlador */
  printk("Dispositivo: %d.%d\n",
 inode->i_rdev >> 8, inode->i_rdev & 0xFF);

  /* No queremos que dos procesos hablen al mismo tiempo */
  if (Device_Open)
    return -EBUSY;

  /* Si había un proceso, tendremos que tener más
   * cuidado aquí.
   *
   * En el caso de procesos, el peligro es que un
   * proceso quizás esté chequeando Device_Open y
   * entonces sea reemplazado por el planificador por otro
   * proceso que ejecuta esta función. Cuando
   * el primer proceso regrese a la CPU, asumirá que el
   * dispositivo no está abierto todavía.
   *
   * De todas formas, Linux garantiza que un proceso no
   * será reemplazado mientras se está ejecutando en el
   * contexto del núcleo.
   *
   * En el caso de SMP, una CPU quizás incremente
   * Device_Open mientras otra CPU está aquí, correcto
   * después de chequear. De todas formas, en la versión
   * 2.0 del núcleo esto no es un problema por que hay un
   * cierre que garantiza que sólamente una CPU estará en
   * el módulo del núcleo en un mismo instante. Esto es malo
   * en términos de rendimiento, por lo tanto la versión 2.2
   * lo cambió. Desgraciadamente, no tengo acceso a un
   * equipo SMP para comprobar si funciona con SMP.
   */

  Device_Open++;

  /* Inicializa el mensaje. */
  sprintf(Message,
    "Si te lo dije una vez, te lo digo %d veces - %s",
    counter++,
    "Hola, mundo\n");
  /* El único motivo por el que se nos permite hacer este
   * sprintf es porque la máxima longitud del mensaje
   * (asumiendo enteros de 32 bits - hasta 10 dígitos
   * con el signo menos) es menor que BUF_LEN, el cual es 80.
   * <<TEN CUIDADO NO HAGAS DESBORDAMIENTO DE PILA EN LOS BUFFERS,
```

```
   * ESPECIALMENTE EN EL NÚCLEO!!! */

  Message_Ptr = Message;

  /* Nos aseguramos de que el módulo no es borrado mientras
   * el fichero está abierto incrementando el contador de uso
   * (el número de referencias abiertas al módulo, si no es
   * cero rmmod fallará) */
  MOD_INC_USE_COUNT;

  return SUCCESS;
}


/* Esta función es llamada cuando un proceso cierra el
 * fichero del dispositivo. No tiene un valor de retorno en
 * la versión 2.0.x porque no puede fallar (SIEMPRE debes de ser
 * capaz de cerrar un dispositivo). En la versión 2.2.x
 * está permitido que falle - pero no le dejaremos. */

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
  struct file *file)
#else
static void device_release(struct inode *inode,
   struct file *file)
#endif
{
#ifdef DEBUG
  printk ("dispositivo_liberado(%p,%p)\n", inode, file);
#endif

  /* Ahora estamos listos para la siguiente petición*/
  Device_Open --;

  /* Decrementamos el contador de uso, en otro caso una vez que
   * hayas abierto el fichero no volverás a coger el módulo.
   */
  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;
#endif
}


/* Esta función es llamada cuando un proceso que ya
 * ha abierto el fichero del dispositivo intenta leer de él. */

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(struct file *file,
    char *buffer,    /* El buffer a rellenar con los datos */
```

```
    size_t length,   /* La longitud del buffer */
    loff_t *offset)  /* Nuestro desplazamiento en el fichero */
#else
static int device_read(struct inode *inode,
                       struct file *file,
    char *buffer,   /* El buffer para rellenar con
     * los datos */
    int length)     /* La longitud del buffer
                     * (<no debemos escribir más allá de él!) */
#endif
{
  /* Número de bytes actualmente escritos en el buffer */
  int bytes_read = 0;

  /* si estamos al final del mensaje, devolvemos 0
   * (lo cual significa el final del fichero) */
  if (*Message_Ptr == 0)
    return 0;

  /* Ponemos los datos en el buffer */
  while (length && *Message_Ptr)  {

    /* Porque el buffer está en el segmento de datos del usuario
     * y no en el segmento de datos del núcleo, la asignación
     * no funcionará. En vez de eso, tenemos que usar put_user,
     * el cual copia datos desde el segmento de datos del núcleo
     * al segmento de datos del usuario. */
    put_user(*(Message_Ptr++), buffer++);


    length --;
    bytes_read ++;
  }

#ifdef DEBUG
   printk ("%d bytes leidos, quedan %d\n",
     bytes_read, length);
#endif

   /* Las funciones de lectura se supone que devuelven el
    * número de bytes realmente insertados en el buffer */
  return bytes_read;
}




/* Se llama a esta función cuando alguien intenta escribir
 * en nuestro fichero de dispositivo - no soportado en este
 * ejemplo. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
```

```
    const char *buffer,    /* El buffer */
    size_t length,   /* La longitud del buffer */
    loff_t *offset)  /* Nuestro desplazamiento en el fichero */
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  return -EINVAL;
}




/* Declaraciones del Módulo **************************** */

/* El número mayor para el dispositivo. Esto es
 * global (bueno, estático, que en este contexto es global
 * dentro de este fichero) porque tiene que ser accesible
 * para el registro y para la liberación. */
static int Major;

/* Esta estructura mantendrá las funciones que son llamadas
 * cuando un proceso hace algo al dispositivo que nosotros creamos.
 * Ya que un puntero a esta estructura se mantiene en
 * la tabla de dispositivos, no puede ser local a
 * init_module. NULL es para funciones no implementadas. */

struct file_operations Fops = {
  NULL,   /* búsqueda */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* seleccionar */
  NULL,   /* ioctl */
  NULL,   /* mmap */
  device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,   /* borrar */
#endif
  device_release  /* a.k.a. cerrar */
};


/* Inicializa el módulo - Registra el dispositivo de carácter */
int init_module()
{
  /* Registra el dispositivo de carácter (por lo menos lo intenta) */
  Major = module_register_chrdev(0,
                                 DEVICE_NAME,
```

```
                                &Fops);

  /* Valores negativos significan un error */
  if (Major < 0) {
    printk ("dispositivo %s falló con %d\n",
    "Lo siento, registrando el carácter",
    Major);
    return Major;
  }

  printk ("%s El número mayor del dispositivo es %d.\n",
          "El registro es un éxito.",
          Major);
  printk ("si quieres hablar con el controlador del dispositivo,\n");
  printk ("tendrás que crear un fichero de dispositivo. \n");
  printk ("Te sugerimos que uses:\n");
  printk ("mknod <nombre> c %d <menor>\n", Major);
  printk ("Puedes probar diferentes números menores %s",
          "y ver que pasa.\n");

  return 0;
}


/* Limpieza - liberamos el fichero correspondiente desde /proc */
void cleanup_module()
{
  int ret;

  /* liberamos el dispositivo */
  ret = module_unregister_chrdev(Major, DEVICE_NAME);

  /* Si hay un error, lo indicamos */
  if (ret < 0)
    printk("Error en unregister_chrdev: %d\n", ret);
}
```

2.1 Múltiples Versiones de Ficheros Fuente del Núcleo

Las llamadas al sistema, que son las principal interfaces del núcleo mostradas a los procesos, generalmente permanecen igual a través de las versiones. Una nueva llamada al sistema quizás sea añadida, pero usualmente las viejas se comportarán igual que de costumbre. Esto es necesario para la compatibilidad regresiva — una nueva versión del núcleo **no** se supone que romperá con los procesos regulares. En la mayoría de los casos, los ficheros de dispositivo permanecerán igual. En el otro caso, las interfaces internas dentro del núcleo pueden y sufren cambios entre las versiones.

Las versiones del núcleo Linux están divididas entre las versiones estables (n.<número par>.m) y las versiones en desarrollo (n.<número impar>.m). Las versiones en desarrollo incluyen todas las nuevas ideas, incluyendo aquellas que serán consideradas un error, o reimplementadas, en la siguiente versión. Como resultado, no puedes confiar en que la interfaz permanecerá igual en estas versiones (lo cual es porque no las soportamos en este libro, es mucho trabajo y caducarán rápidamente). En las versiones estables, por otro

lado, podemos esperar que al interfaz permanezca sin cambios a través de las versiones de corrección de fallos (el número m).

Esta versión de la GPMNL incluye soporte para la versión 2.0.x y la versión 2.2.x del núcleo Linux. Como hay diferencias entre las dos, esto requiere compilación condicional dependiendo de la versión del núcleo. La forma con la que hacemos esto es usando la macro `LINUX_VERSION_CODE`. En la versión a.b.c. de un núcleo, el valor de esta macro debería de ser $2^{16}a + 2^8b + c$. Para obtener el valor específico de una versión específica del núcleo, podemos usar la macro `KERNEL_VERSION`. Como no está definida en 2.0.35, la definiremos nosotros si es necesario.

Capítulo 3

El Sistema de Ficheros /proc

El Linux hay un mecanismo adicional para que el núcleo y los módulos del núcleo envíen información a los procesos — el sistema de ficheros `/proc`. Originalmente diseñado para permitir un fácil acceso a la información sobre los procesos (de aquí el nombre), es ahora usado por cada parte del núcleo que tiene algo interesante que informar, como `/proc/modules` que tiene la lista de los módulos y `/proc/meminfo` que tiene las estadísticas de uso de la memoria.

El método para usar el sistema de ficheros proc es muy similar al usado con los controladores de dispositivos — creas una estructura con toda la información necesitada por el fichero `/proc`, incluyendo punteros a cualquier función manejadora (en nuestro caso sólo hay una, que es llamada cuando alguien intenta leer desde el fichero `/proc`). Entonces, `init_module` registra la estructura en el núcleo y `cleanup_module` la libera.

El motivo por el que usamos `proc_register_dynamic`[1] es porque no queremos determinar el número de inodo usado para nuestro fichero en adelante, pero permitimos al núcleo determinarlo para prevenir que falle. Los sistemas de ficheros normales están localizados en un disco, en vez de en memoria (que es donde está `/proc`), y en ese caso el número de inodo es un puntero a una posición de disco donde el nodo índice del fichero (abreviadamente inodo) está localizado. El inodo contiene información sobre el fichero, por ejemplo los permisos del fichero, junto con un puntero a la posición o posiciones del disco donde pueden ser encontrados los datos del fichero.

Como nosotros no cogemos la llamada cuando el fichero es abierto o cerrado, no podemos poner `MOD_INC_USE_COUNT` y `MOD_DEC_USE_COUNT` en este módulo, y si el fichero es abierto y entonces el módulo es borrado, no hay forma de impedir las consecuencias. En el siguiente capítulo veremos una implementación más dura, pero más flexible, para tratar con los ficheros `/proc` que permitirá protegernos también de este problema.

procfs.c

```
/* procfs.c -  crea un "fichero" en /proc
 * Copyright (C) 1998-1999 by Ori Pomerantz
 */


/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
```

[1] En la versión 2.0, en la versión 2.2 esto es realizado automáticamente para nosotros si establecemos el inodo a cero.

```
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Necesario porque usamos el sistema de ficheros proc */
#include <linux/proc_fs.h>



/* En 2.2.3 /usr/include/linux/version.h se incluye
 * una macro para eso, pero 2.0.35 no lo hace - por lo
 * tanto lo añado aquí si es necesario */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Ponemos datos en el fichero del sistema de fichero proc.

   Argumentos
   ==========
   1. El buffer donde los datos van a ser insertados, si
      decides usarlo.
   2. Un puntero a un puntero de caracteres. Esto es útil si
      no quieres usar el buffer asignado por el núcleo.
   3. La posición actual en el fichero.
   4. El tamaño del buffer en el primer argumento.
   5. Cero (>para uso futuro?).

   Uso y Valores de Retorno
   ========================
   Si utilizas tu propio buffer, como yo, pon su situación
   en el segundo argumento y retorna el número de bytes
   usados en el buffer.

   Un valor de retorno de cero significa que actualmente no
   tienes más información (final del fichero). Un valor negativo
   es una condición de error.

   Para Más Información
   ====================
   La forma en la que descubrí qué hacer con esta función
   no fue leyendo documentación, sino que fue leyendo el código
   que lo utiliza. Justamente miré para ver para qué usa el campo
   get_info de la struct proc_dir_entry (Usé una combinación
   de find y grep, por si estás interesado), y vi que se usa en
   <directorio del código del núcleo>/fs/proc/array.c.

   Si algo no es conocido sobre el núcleo, esta es la forma
   habitual de hacerlo. En Linux tenemos la gran ventaja
```

```
   de tener el código fuente del núcleo gratis - úsalo.
 */
int procfile_read(char *buffer,
  char **buffer_location,
  off_t offset,
  int buffer_length,
  int zero)
{
  int len;  /* El número de bytes usados realmente */

  /* Esto es static, por lo tanto permanecerá en
   * memoria cuando abandonemos esta función */
  static char my_buffer[80];

  static int count = 1;

  /* Damos toda nuestra información de una vez, por lo tanto
   * si el usuario nos pregunta si tenemos más información
   * la respuesta debería de ser no.
   *
   * Esto es importante porque la función estándar de lectura
   * de la librería debería continuar emitiendo la
   * llamada al sistema read hasta que el núcleo responda
   * que no hay más información, o hasta que el buffer esté
   * lleno.
   */
  if (offset > 0)
    return 0;

  /* Rellenamos el buffer y cogemos su longitud */
  len = sprintf(my_buffer,
    "Para la vez %d%s, vete!\n", count,
    (count % 100 > 10 && count % 100 < 14) ? "th" :
      (count % 10 == 1) ? "st" :
        (count % 10 == 2) ? "nd" :
          (count % 10 == 3) ? "rd" : "th" );
  count++;

  /* Dice a la función que llamamos dónde está el buffer */
  *buffer_location = my_buffer;

  /* Devolvemos la longitud */
  return len;
}


struct proc_dir_entry Our_Proc_File =
  {
    0, /* Número de Inodo - ignóralo, será rellenado por
        * proc_register[_dynamic] */
    4, /* Longitud del nombre del fichero */
    "test", /* El nombre del fichero */
```

```
    S_IFREG | S_IRUGO, /* Modo del fichero - este es un fichero
                        * regular que puede ser leído por su
                        * dueño, por su grupo, y por todo el mundo */
    1,/* Número de enlaces (directorios donde el
         * fichero está referenciado) */
    0, 0,  /* El uid y gid para el fichero - se lo damos
            * a root */
    80, /* El tamaño del fichero devuelto por ls. */
    NULL, /* funciones que pueden ser realizadas en el inodo
           * (enlazado, borrado, etc.) - no soportamos
           * ninguna. */
    procfile_read, /* La función read para este fichero,
                    * la función llamada cuando alguien
                    * intenta leer algo de el. */
    NULL /* Podemos tener aquí un función que rellene el
          * inodo del fichero, para habilitarnos el jugar
          * con los permisos, dueño, etc. */
  };




/* Inicializa el módulo - registra el fichero proc */
int init_module()
{
  /* Tiene éxito si proc_register[_dynamic] tiene éxito,
   * falla en otro caso. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  /* En la versión 2.2, proc_register asigna un número
   * de inodo automáticamente si hay cero en la estructura,
   * por lo tanto no necesitamos nada más para
   * proc_register_dynamic
   */


  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* proc_root es el directorio raiz para el sistema de ficheros
   * proc (/proc). Aqué es dónde queremos que nuestro fichero esté
   * localizado.
   */
}


/* Limpieza - liberamos nuestro fichero de /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
```

```
}
```

Capítulo 4

Usando /proc Para Entrada

Por lo tanto, tenemos dos formas de generar salida para los módulos del núcleo; podemos registrar un controlador de dispositivo y `mknod` el fichero de dispositivo, o podemos crear un fichero `/proc`. Esto permite al módulo del núcleo decirnos cualquier cosa que quiera. El único problema es que no hay forma de que nos responda. La primera forma en que enviaremos entrada a los módulos del núcleo será volviendo a escribir en el fichero `/proc`.

Porque el sistema de ficheros proc fue escrito principalmente para permitir al núcleo informar de su situación a los procesos, no hay medidas especiales para la entrada. La estructura `proc_dir_entry` no incluye un puntero a una función de entrada, de la forma que se incluye un puntero a una función de salida. En vez de esto, para escribir en un fichero `/proc`, necesitamos usar el mecanismo estándar del sistema de ficheros.

En Linux hay un mecanismo estándar para el registro de sistemas de ficheros. Como cada sistema de ficheros tiene que tener sus propias funciones para manejar las operaciones de inodos y ficheros[1], hay una estructura especial para mantener los punteros a todas estas funciones, `struct inode_operations`, la cual incluye un puntero a `struct file_operations`. En /proc, cuando registramos un nuevo fichero, se nos permite especificar que `struct inode_operations` será usada para acceder a el. Este es el mecanismo que usaremos, una `struct inode_operations` que incluya punteros a nuestras funciones `module_input` y `module_output`.

Es importante destacar que los papeles estándar de lectura y escritura son reservados en el núcleo. Las funciones de lectura son usadas para salida, mientras que las funciones de escritura son usadas para la entrada. El motivo de esto es que la lectura y escritura se refieren al punto de vista del usuario — si un proceso lee algo del núcleo, entonces el núcleo necesita sacarlo, y si un proceso escribe algo en el núcleo, entonces el núcleo lo recibe como entrada.

Otro punto interesante aquí es la función `module_permission`. Esta función se llama cuando un proceso intenta hacer algo con el fichero `/proc`, y puede decidir cuándo permitir el acceso o no. Ahora mismo está solamente basado en la operación y el uid del usuario actual (tal como está disponible en `current`, un puntero a una estructura que incluye información del proceso actualmente ejecutándose), pero puede estar basado en cualquier cosa que queramos, como lo que otro proceso se encuentre realizando en el mismo fichero, la hora del día, o la última entrada recibida

El motivo para `put_user` y `get_user` es que la memoria de Linux (bajo la arquitectura Intel, quizás sea diferente bajo otros procesadores) está segmentada. Esto significa que un puntero, por si mismo, no referencia una única posición en memoria, sólo una posición en un segmento de memoria, y necesitas saber en qué segmento de memoria está para poder usarlo. Sólo hay un segmento de memoria para el núcleo, y uno para cada proceso.

El único segmento de memoria accesible a un proceso es el suyo, por lo tanto cuando escribimos programas normales para ejecutarse como procesos no hay necesidad de preocuparse por los segmentos. Cuando escribes un módulo del núcleo, normalmente quieres acceder al segmento de memoria del núcleo, el cual es

[1] La diferencia entre estos dos es que las operaciones de ficheros tratan con el fichero, y las operaciones de inodo tratan con las formas de referenciar el fichero, tales como crear enlaces a el.

manejado automáticamente por el sistema. Entonces, cuando el contenido de un búfer de memoria necesita ser pasado entre el proceso actualmente ejecutándose y el núcleo, la función del núcleo recibe un puntero al búfer de memoria en el cual está el segmento del proceso. Las macros `put_user` y `get_user` nos permiten acceder a esa memoria.

procfs.c

```
/* procfs.c -  crea un "fichero" en /proc, que permite
 * entrada y salida. */

/* Copyright (C) 1998-1999 by Ori Pomerantz */


/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necesario porque usamos el sistema de ficheros proc */
#include <linux/proc_fs.h>


/* En 2.2.3 /usr/include/linux/version.h se incluye
 * una macro para eso, pero 2.0.35 no lo hace - por lo
 * tanto lo añado aquí si es necesario */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* para get_user y put_user */
#endif

/* Las funciones del fichero del módulo ********************** */


/* Aquí mantenemos el último mensaje recibido, para
 * comprobar que podemos procesar nuestra entrada */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];


/* Desde que usamos la estructura de operaciones de fichero.
 * podemos usar las provisiones de salida especiales de proc -
```

```
 * tenemos que usar una función de lectura estándar, y es
 * esta función */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
    struct file *file,   /* El fichero leído */
    char *buf, /* El buffer donde se van a poner los datos
                * (en el segmento de usuario) */
    size_t len,  /* La longitud del buffer */
    loff_t *offset) /* Desplazamiento en el fichero - ignóralo */
#else
static int module_output(
    struct inode *inode, /* El inodo leído */
    struct file *file,   /* El fichero leído */
    char *buf, /* El buffer donde se van a poner los datos
                * (en el segmento de usuario) */
    int len)  /* La longitud del buffer */
#endif
{
  static int finished = 0;
  int i;
  char message[MESSAGE_LENGTH+30];

  /* Retornamos 0 para indicar el final del fichero, que
   * no tenemos más información. En otro caso, los procesos
   * continuarán leyendo de nosotros en un bucle sin fin. */
  if (finished) {
    finished = 0;
    return 0;
  }

  /* Usamos put_user para copiar la cadena de caracteres del
   * segmento de memoria del núcleo al segmento de memoria de
   * proceso que nos llamó. get_user, BTW, es usado para
   * lo contrario. */
  sprintf(message, "Last input:%s", Message);
  for(i=0; i<len && message[i]; i++)
    put_user(message[i], buf+i);


  /* Nota, asumimos aquí que el tamaño del mensaje está por
   * debajo de la longitud, o se recibirá cortado. En una
   * situación de la vida real, si el tamaño del mensaje es menor
   * que la longitud entonces retornamos la longitud y en la
   * segunda llamada empezamos a rellenar el buffer con el byte
   * longitud+1 del mensaje. */
  finished = 1;

  return i;  /* Retornamos el número de bytes "leidos" */
}


/* Esta función recibe la entrada del usuario cuando el
```

```
 * usuario escribe en el fichero /proc. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
    struct file *file,   /* El mismo fichero */
    const char *buf,     /* El buffer con la entrada */
    size_t length,       /* La longitud del buffer */
    loff_t *offset)      /* desplazamiento del fichero - ignóralo */
#else
static int module_input(
    struct inode *inode, /* El inodo del fichero */
    struct file *file,   /* El mismo fichero */
    const char *buf,     /* El buffer con la entrada */
    int length)          /* La longitud del buffer */
#endif
{
  int i;

  /* Pone la entrada en Message, donde module_output
   * posteriormente será capaz de usarlo */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buf+i);
  /* En la versión 2.2 la semántica de get_user cambió,
   * no volverá a devolver un carácter, excepto una
   * variable para rellenar como primer argumento y
   * un puntero al segmento de usuario para rellenarlo
   * como segundo.
   *
   * El motivo para este cambio es que en la versión 2.2
   * get_user puede leer un short o un int. La forma
   * en la que conoce el tipo de la variable que
   * debería de leer es usando sizeof, y para lo que
   * necesita la variable.
   */
#else
    Message[i] = get_user(buf+i);
#endif
  Message[i] = '\0';  /* queremos un estándar, cadena
                       * de caracteres terminada en cero */

  /* Necesitamos devolver el número de caracteres de entrada
   * usados */
  return i;
}



/* Esta función decide si permite una operación (retorna cero)
 * o no la permite (retornando distinto de cero, lo cual indica porqué
 * no está permitido).
 *
 * La operación puede ser uno de los siguientes valores:
```

```
 * 0 - Ejecuta (ejecuta el "fichero" - sin sentido en nuestro caso)
 * 2 - Escribe (entrada en el módulo del núcleo)
 * 4 - Lee (salida desde el módulo del núcleo)
 *
 * Esta es la función real que chequea los permisos del
 * fichero. Los permisos retornados por ls -l son sólo
 * para referencia, y pueden ser sobreescritos aquí.
 */
static int module_permission(struct inode *inode, int op)
{
  /* Permitimos a todo el mundo leer desde nuestro módulo, pero
   * sólo root (uid 0) puede escribir en el */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /* Si es algún otro, el acceso es denegado */
  return -EACCES;
}




/* El fichero está abierto - realmente no nos preocupamos de
 * esto, pero significa que necesitamos incrementar el
 * contador de referencias del módulo. */
int module_open(struct inode *inode, struct file *file)
{
  MOD_INC_USE_COUNT;

  return 0;
}


/* El fichero está cerrado - otra vez, interesante sólo por
 * el contador de referencias. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* realizado con éxito */
#endif
}


/* Estructuras para registrar el fichero /proc, con
 * punteros a todas las funciones relevantes. ********** */
```

```
/* Las operaciones del fichero para nuestro fichero proc. Es aquí
 * donde colocamos los punteros a todas las funciones llamadas
 * cuando alguien intenta hacer algo en nuestro fichero. NULL
 * significa que no queremos tratar con algo. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "lee" desde el fichero */
    module_input,   /* "escribe" en el fichero */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,    /* Alguien abrió el fichero */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* borrado, añadido aquí en la versión 2.2 */
#endif
    module_close,    /* Alguien cerró el fichero */
    /* etc. etc. etc. ( son todas dadas en
     * /usr/include/linux/fs.h). Ya que no ponemos nada
     * más aquí, el sistema mantendrá los datos por defecto
     * que en Unix son ceros (NULLs cuando cogemos
     * punteros). */
  };
```

```
/* Las operaciones del inodo para nuestro fichero proc. Las necesitamos,
 * por lo tanto tendremos algún lugar para especificar las
 * estructuras de operaciones del fichero que queremos usar. También es
 * posible especificar funciones a ser llamadas para cualquier cosa
 * que pudiera ser hecha en un inodo (como no queremos molestar,
 * las ponemos a NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* crear */
    NULL, /* lookup */
    NULL, /* enlazar */
    NULL, /* desenlazar */
    NULL, /* enlace simbólico */
    NULL, /* mkdir */
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* renombrar */
    NULL, /* leer enlace */
    NULL, /* seguir el enlace */
    NULL, /* leer página */
    NULL, /* escribir página */
    NULL, /* bmap */
    NULL, /* cortar */
```

```
    module_permission /* chequeo para permisos */
  };


/* Entrada de directorio */
static struct proc_dir_entry Our_Proc_File =
  {
    0, /* Número de inodo - ignóralo, será automáticamente rellenado
        * por proc_register[_dynamic] */
    7, /* Longitud del nombre del fichero */
    "rw_test", /* El nombre del fichero */
    S_IFREG | S_IRUGO | S_IWUSR,
    /* Modo del fichero - este es un fichero normal el cual
     * puede ser leído por su dueño, su grupo, y por todo el
     * mundo. También, su dueño puede escribir en él.
     *
     * Realmente, este campo es sólo para referencia, es
     * module_permission el que hace el chequeo actual.
     * Puede usar este campo, pero en nuestra implementación
     * no lo hace, por simplificación. */
    1,  /* Número de enlaces (directorios donde el fichero
         * está referenciado) */
    0, 0,  /* El uid y gid para el fichero -
            * se lo damos a root */
    80, /* El tamaño del fichero reportado por ls. */
    &Inode_Ops_4_Our_Proc_File,
    /* Un puntero a la estructura del inodo para
     * el fichero, si lo necesitamos. En nuestro caso
     * lo hacemos, porque necesitamos una función de escritura. */
    NULL
    /* La función de lectura para el fichero. Irrelevante
     * porque lo ponemos en la estructura de inodo anterior */
  };



/* Inicialización del módulo y limpieza ****************** */

/* Inicializa el módulo - registra el fichero proc */
int init_module()
{
  /* Tiene éxito si proc_register[_dynamic] tiene éxito,
   * falla en otro caso */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  /* En la versión 2.2, proc_register asigna dinámicamente un número de
   * inodo automáticamente si hay un cero en la estructura, por lo
   * tanto no se necesita más para proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
```

```
}


/* Limpieza - liberamos nuestro fichero de /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

Capítulo 5

Hablando con los Ficheros de Dispositivo (escrituras y IOCTLs)

Los ficheros de dispositivos se supone que representan dispositivos físicos. La mayoría de los dispositivos físicos son usados para salida y para entrada, por lo tanto tiene que haber algún mecanismo para los controladores de dispositivos en el núcleo para obtener la salida a enviar desde el dispositivo a los procesos. Esto es realizado abriendo el fichero del dispositivo para salida y escribiendo en el, justamente como escribir a un fichero. En el siguiente ejemplo, esto es implementado por `device_write`.

Esto no es siempre suficiente. Imagínate que tienes un puerto serie conectado a un módem (incluso si tienen un módem interno, todavía es implementado desde la perspectiva de la CPU como un puerto serie conectado a un módem, por lo tanto no tienes que hacer que tu imaginación trabaje mucho). Lo natural sería usar el fichero del dispositivo para escribir cosas al módem (comandos del módem o datos a ser enviados a través de la linea telefónica) y leer cosas desde el módem (respuestas a comandos o datos recibidos a través de la linea telefónica). Entonces, esto deja abierta la pregunta de qué hacer cuando necesitas hablar con el puerto serie, por ejemplo para enviarle la velocidad a la que los datos son enviados y recibidos.

La respuesta en Unix es usar una función especial llamada `ioctl` (abreviatura para **i**nput **o**utput **c**on**t**ro**l**). Cada dispositivo tiene sus propios comandos `ioctl`, los cuales pueden leer `ioctl`'s (para enviar información desde un proceso al núcleo), escribir (`ioctl`'s (para devolver información a un proceso), [1] ambas o ninguna. La función se llama con tres parámetros; el descriptor del fichero del dispositivo apropiado, el número de ioctl, y un parámetro, el cual es de tipo long y al que le puedes hacer una conversión (cast) para pasarle cualquier cosa. [2]

El número ioctl codifica el número mayor del dispositivo, el tipo de la ioctl, el comando, y el tipo del parámetro. Este número ioctl es usualmente creado por una llamada a una macro (`_IO`, `_IOR`, `_IOW` o `_IOWR` — dependiendo del tipo) en el fichero de cabeceras. Este fichero de cabeceras debería ser incluido (`#include`) en ambos programas que usarán `ioctl` (para que ellos puedan generar el ioctl apropiado) y por el módulo del núcleo (para que lo entienda). En el ejemplo siguiente, el fichero de cabeceras es `chardev.h` y el programa que lo usa es `ioctl.c`

Si quieres usar `ioctl`'s en tus propios módulos del núcleo, es mejor recibir un asignación `ioctl` oficial, por que si accidentalmente coges los ioctls de alguien, o si alguien coge los tuyos, sabrás que algo está mal. Para más información, consulta el árbol de código fuente del núcleo en '`Documentation/ioctl-number.txt`'.

chardev.c

```
/* chardev.c
 *
```

[1] Destacar que aquí los papeles de leer y escribir se han intercambiado *otra vez*, por lo tanto en las lecturas `ioctl` se envía información al núcleo y las escrituras reciben información desde el núcleo.

[2] Esto no es exacto. Tu no podrás pasarle una estructura, por ejemplo, a través de un ioctl — pero podrás pasarle un puntero a la estructura.

```
 * Crea un dispositivo de entrada/salida de carácter
 */


/* Copyright (C) 1998-99 por Ori Pomerantz */



/* Los ficheros de cabeceras necesarios */

/* Estándar en módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Para dispositivos de carácter */

/* Las definiciones de dispositivo de carácter están aquí */
#include <linux/fs.h>

/* Un envoltorio el cual no hace nada en la
 * actualidad, pero que quizás ayude para compatibilizar
 * con futuras versiones de Linux */
#include <linux/wrapper.h>


/* Nuestros propios números ioctl */
#include "chardev.h"


/* En 2.2.3 /usr/include/linux/version.h se incluye una
 * macro para esto, pero 2.0.35 no lo hace - por lo tanto
 * lo añado aquí si es necesario. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* para get_user y put_user */
#endif



#define SUCCESS 0
```

```
/* Declaraciones de Dispositivo ****************************** */


/* el nombre de nuestro dispositivo, tal como aparecerá en
 * /proc/devices */
#define DEVICE_NAME "char_dev"


/* La máxima longitud del mensaje para nuestro dispositivo */
#define BUF_LEN 80


/* >Está el dispositivo correctamente abierto ahora? Usado
 * para evitar acceso concurrente al mismo dispositivo */
static int Device_Open = 0;

/* El mensaje que el dispositivo nos dará cuando preguntemos */
static char Message[BUF_LEN];

/* >Cuanto puede coger el proceso para leer el mensaje?
 * Útil si el mensaje es más grande que el tamaño del
 * buffer que tenemos para rellenar en device_read. */
static char *Message_Ptr;


/* Esta función es llamada cuando un proceso intenta
 * abrir el fichero de dispositivo */
static int device_open(struct inode *inode,
                       struct file *file)
{
#ifdef DEBUG
  printk ("device_open(%p)\n", file);
#endif

  /* No queremos hablar con dos procesos a la vez */
  if (Device_Open)
    return -EBUSY;

  /* Si esto era un proceso, tenemos que tener más cuidado aquí,
   * porque un proceso quizás haya chequeado Device_Open correctamente
   * antes de que el otro intentara incrementarlo. De cualquier forma,
   * estamos en el núcleo, por lo tanto estamos protegidos contra
   * los cambios de contexto.
   *
   * Esta NO es la actitud correcta a tomar, porque quizás estemos
   * ejecutándonos en un sistema SMP, pero trataremos con SMP
   * en un capítulo posterior.
   */

  Device_Open++;
```

```
  /* Inicializa el mensaje */
  Message_Ptr = Message;

  MOD_INC_USE_COUNT;

  return SUCCESS;
}


/* Esta función se llama cuando un proceso cierra el
 * fichero del dispositivo. No tiene un valor de retorno
 * porque no puede fallar. Sin pérdida de consideración de
 * lo que pudiera pasar, deberías de poder cerrar siempre un
 * dispositivo (en 2.0, un fichero de dispositivo 2.2 puede
 * ser imposible de cerrar). */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
                          struct file *file)
#else
static void device_release(struct inode *inode,
                           struct file *file)
#endif
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif

  /* Ahora estamos listos para la siguiente llamada */
  Device_Open --;

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;
#endif
}



/* Esta función se llama cuando un proceso que ya
 * ha abierto el fichero del dispositivo intenta leer
 * de él. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(
    struct file *file,
    char *buffer, /* El buffer para rellenar con los datos */
    size_t length,     /* La longitud del buffer */
    loff_t *offset) /* desplazamiento en el fichero */
#else
static int device_read(
    struct inode *inode,
    struct file *file,
```

```
    char *buffer,   /* El buffer para rellenar con los datos */
    int length)     /* La longitud del buffer
                     * (<no debemos de escribir más allá de él!) */
#endif
{
  /* Número de bytes actualmente escritos en el buffer */
  int bytes_read = 0;

#ifdef DEBUG
  printk("device_read(%p,%p,%d)\n",
    file, buffer, length);
#endif

  /* Si estamos al final del mensaje, retornamos 0
   * (lo cual significa el final del fichero) */
  if (*Message_Ptr == 0)
    return 0;

  /* Realmente ponemos los datos en el buffer */
  while (length && *Message_Ptr)  {

    /* Como el buffer está en el segmento de datos del usuario
     * y no en el segmento de datos del núcleo, la asignación
     * no funcionará. En vez de ello, tenemos que usar put_user
     * el cual copia datos desde el segmento de datos del núcleo
     * al segmento de datos del usuario. */
    put_user(*(Message_Ptr++), buffer++);
    length --;
    bytes_read ++;
  }

#ifdef DEBUG
   printk ("Leídos %d bytes, quedan %d\n",
     bytes_read, length);
#endif

   /* Las funciones de lectura se supone que devuelven el número
    * de bytes realmente insertados en el buffer */
  return bytes_read;
}


/* Esta función se llama cuando alguien intenta
 * escribir en nuestro fichero de dispositivo. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
                            const char *buffer,
                            size_t length,
                            loff_t *offset)
#else
static int device_write(struct inode *inode,
                        struct file *file,
```

```
                        const char *buffer,
                        int length)
#endif
{
  int i;

#ifdef DEBUG
  printk ("device_write(%p,%s,%d)",
    file, buffer, length);
#endif

  for(i=0; i<length && i<BUF_LEN; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buffer+i);
#else
    Message[i] = get_user(buffer+i);
#endif

  Message_Ptr = Message;

  /* De nuevo, retornamos el número de caracteres de entrada usados */
  return i;
}


/* Esta función es llamada cuando un proceso intenta realizar
 * una ioctl en nuestro fichero de dispositivo. Cogemos dos
 * parámetros extra (en adición al inodo y a las estructuras
 * del fichero, los cuales cogen todas las funciones de dispositivo): el
 * número de ioctl llamado y el parámetro dado a la función ioctl.
 *
 * Si el ioctl es de escritura o de lectura/escritura (significa
 * que la salida es devuelta al proceso que llama), la llamada ioctl
 * retorna la salida de esta función.
 */
int device_ioctl(
    struct inode *inode,
    struct file *file,
    unsigned int ioctl_num,/* El número de ioctl */
    unsigned long ioctl_param) /* El parámetro a él */
{
  int i;
  char *temp;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  char ch;
#endif

  /* Se intercambia de acuerdo al ioctl llamado */
  switch (ioctl_num) {
    case IOCTL_SET_MSG:
      /* Recibe un puntero al mensaje (en el espacio de usuario)
       * y establece lo que será el mensaje del dispositivo. */
```

```
      /* Coge el parámetro dado a ioctl por el proceso */
      temp = (char *) ioctl_param;

      /* Encuentra la longitud del mensaje */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, temp);
      for (i=0; ch && i<BUF_LEN; i++, temp++)
        get_user(ch, temp);
#else
      for (i=0; get_user(temp) && i<BUF_LEN; i++, temp++)
;
#endif

      /* No reinventa la rueda - llama a device_write */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      device_write(file, (char *) ioctl_param, i, 0);
#else
      device_write(inode, file, (char *) ioctl_param, i);
#endif
      break;

    case IOCTL_GET_MSG:
      /* Da el mensaje actual al proceso llamador - el parámetro
       * que damos en un puntero, lo rellena. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      i = device_read(file, (char *) ioctl_param, 99, 0);
#else
      i = device_read(inode, file, (char *) ioctl_param,
                      99);
#endif
      /* Peligro - asumimos aquí que la longitud del buffer es
       * 100. Si es menor de lo que tenemos quizás desborde el
       * buffer, causando que el proceso vuelque la memoria.
       *
       * El motivo por el que permitimos hasta 99 caracteres es
       * que el NULL que termina la cadena de caracteres también
       * necesita sitio. */

      /* Pone un cero al final del buffer, por lo
       * tanto estará correctamente terminado */
      put_user('\0', (char *) ioctl_param+i);
      break;

    case IOCTL_GET_NTH_BYTE:
      /* Este ioctl es para entrada (ioctl_param) y
       * para salida (el valor de retorno de esta función) */
      return Message[ioctl_param];
      break;
  }

  return SUCCESS;
```

```
}


/* Declaraciones del Módulo ************************** */


/* Esta estructura mantendrá las funciones a ser llamadas
 * cuando un proceso realiza algo al dispositivo que hemos
 * creado. Desde que un puntero a esta estructura es mantenido
 * en la tabla de dispositivos, no puede ser local a init_module.
 * NULL es para funciones no implementadas. */
struct file_operations Fops = {
  NULL,   /* búsqueda */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* selección */
  device_ioctl,   /* ioctl */
  NULL,   /* mmap */
  device_open,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,  /* borrar */
#endif
  device_release  /* cerrar */
};


/* Inicializa el módulo - Registra el dispositivo de carácter */
int init_module()
{
  int ret_val;

  /* Registra el dispositivo de carácter (por lo menos lo intenta) */
  ret_val = module_register_chrdev(MAJOR_NUM,
                                   DEVICE_NAME,
                                   &Fops);

  /* Valores negativos significan un error */
  if (ret_val < 0) {
    printk ("%s falló con %d\n",
            "Lo siento, registrando el dispositivo de carácter ",
            ret_val);
    return ret_val;
  }

  printk ("%s El número mayor del dispositivo es %d.\n",
          "El registro es un éxito",
          MAJOR_NUM);
  printk ("si quieres hablar con el controlador del dispositivo,\n");
  printk ("tienes que crear el fichero del dispositivo. \n");
  printk ("Te sugerimos que uses:\n");
  printk ("mknod %s c %d 0\n", DEVICE_FILE_NAME,
```

```
          MAJOR_NUM);
  printk ("El nombre del fichero del dispositivo es muy importante, porque\n");
  printk ("el programa ioctl asume que es el\n");
  printk ("fichero que usarás.\n");

  return 0;
}


/* Limpieza - libera el fichero apropiado de /proc */
void cleanup_module()
{
  int ret;

  /* libera el dispositivo */
  ret = module_unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

  /* Si hay un error,informa de ello*/
  if (ret < 0)
    printk("Error en module_unregister_chrdev: %d\n", ret);
}
```

chardev.h

```
/* chardev.h - el fichero de cabeceras con las definiciones ioctl.
 *
 * Aquí las declaraciones tienen que estar en un fichero de cabeceras,
 * porque necesitan ser conocidas por el módulo del núcleo
 * (en chardev.c) o por el proceso llamando a ioctl (ioctl.c)
 */

#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>



/* El número mayor del dispositivo. No podemos dejar nada más
 * en el registro dinámico, porque ioctl necesita conocerlo. */
#define MAJOR_NUM 100


/* Establece el mensaje del controlador del dispositivo */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/* _IOR significa que estamos creando un número de comando ioctl
 * para pasar información desde un proceso de usuario al módulo
```

```
 * del núcleo.
 *
 * El primer argumento, MAJOR_NUM, es el número mayor de
 * dispositivo que estamos usando.
 *
 * El segundo argumento es el número del comando
 * (puede haber varios con significado distintos).
 *
 * El tercer argumento es el tipo que queremos coger
 * desde el proceso al núcleo
 */

/* Coge el mensaje del controlador de dispositivo */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
 /* Este IOCTL es usado para salida, para coger el mensaje
  * del controlador de dispositivo. De cualquier forma, aún
  * necesitamos el buffer para colocar el mensaje en la entrada,
  * tal como es asignado por el proceso.
  */


/* Coge el byte n'esimo del mensaje */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
 /* El IOCTL es usado para entrada y salida. Recibe
  * del usuario un número, n, y retorna Message[n]. */


/* El nombre del fichero del dispositivo */
#define DEVICE_FILE_NAME "char_dev"


#endif
```

ioctl.c

```
/* ioctl.c - el proceso para usar las ioctls para controlar
 * el módulo del núcleo
 *
 * Hasta ahora podíamos usar cat para entrada y salida.
 * Pero ahora necesitamos realizar ioctls, los cuales
 * requieren escribir en nuestro proceso.
 */

/* Copyright (C) 1998 by Ori Pomerantz */
```

```
/* específico del dispositivo, tales como números ioctl
 * y el fichero del dispositivo mayor. */
#include "chardev.h"


#include <fcntl.h>      /* abrir */
#include <unistd.h>     /* salir */
#include <sys/ioctl.h>  /* ioctl */



/* Funciones para las llamadas ioctl */

ioctl_set_msg(int file_desc, char *message)
{
  int ret_val;

  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);

  if (ret_val < 0) {
    printf ("ioctl_set_msg fallido:%d\n", ret_val);
    exit(-1);
  }
}



ioctl_get_msg(int file_desc)
{
  int ret_val;
  char message[100];

  /* Peligro - esto es peligroso porque no decimos al
   * núcleo cuanto le está permitido escribir, por lo
   * tanto, quizás desborde el buffer. En la creación
   * de un programa real, deberemos usar dos ioctls - uno
   * para decir al núcleo la longitud del buffer y otro para
   * darle el buffer a rellenar
   */

  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);

  if (ret_val < 0) {
    printf ("ioctl_get_msg fallido:%d\n", ret_val);
    exit(-1);
  }

  printf("get_msg mensaje:%s\n", message);
}
```

```
ioctl_get_nth_byte(int file_desc)
{
  int i;
  char c;

  printf("get_nth_byte mensaje:");

  i = 0;
  while (c != 0) {
    c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);

    if (c < 0) {
      printf(
      "ioctl_get_nth_byte fallo en el byte %d'esimo:\n", i);
      exit(-1);
    }

    putchar(c);
  }
  putchar('\n');
}
```

```
/* Principal - Llama a las funciones ioctl */
main()
{
  int file_desc, ret_val;
  char *msg = "Mensaje pasado por ioctl\n";

  file_desc = open(DEVICE_FILE_NAME, 0);
  if (file_desc < 0) {
    printf ("No se puede abrir el fichero del dispositivo: %s\n",
            DEVICE_FILE_NAME);
    exit(-1);
  }

  ioctl_get_nth_byte(file_desc);
  ioctl_get_msg(file_desc);
  ioctl_set_msg(file_desc, msg);

  close(file_desc);
}
```

Capítulo 6

Parámetros de Inicio

En muchos ejemplos previos, tuvimos que codificar algo en el módulo del núcleo, tal como el nombre del fichero para los ficheros `/proc` o el número mayor del dispositivo para el dispositivo para que pudiéramos hacer ioctls en el. Esto va en contra de la filosofía de Unix, y Linux, la cual es escribir un programa flexible que el usuario pueda configurar.

La forma de decirle a un programa, o a un módulo del núcleo, algo que necesitan antes de empezar a trabajar es mediante los parámetros de la linea de comandos. En el caso de los módulos del núcleo, nosotros no tenemos `argc` y `argv` — en vez de esto, tenemos algo mejor. Podemos definir variables globales en el módulo del núcleo e `insmod` las rellenará por nosotros.

En este módulo del núcleo, definimos dos de ellas: `str1` y `str2`. Todo lo que necesitas hacer es compilar el módulo del núcleo y entonces ejecutar `insmod str1=xxx str2=yyy`. Cuando `init_module` es llamado, `str1` apuntará a la cadena de caracteres 'xxx' y `str2` a la cadena de caracteres 'yyy'.

En la versión 2.0 no hay chequeo de tipos de estos argumentos[1]. Si el primer carácter de `str1` o `str2` es un dígito, el núcleo rellenara la variable con el valor del entero, en vez de con un puntero a la cadena de caracteres. En una situación de la vida real tienes que verificar esto.

En cambio, en la versión 2.2 usas la macro `MACRO_PARM` para decir a `insmod` lo que esperas como parámetros, su nombre *y su tipo*. Esto resuelve el problema de los tipos y permite a los módulos del núcleo recibir cadenas de caracteres que empiezan con un dígito, por ejemplo.

param.c

```
/* param.c
 *
 * Recibe en linea de comandos los parámetros en la instalación del módulo
 */

/* Copyright (C) 1998-99 by Ori Pomerantz */




/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */
```

[1] No puede haberlos, ya que bajo C el fichero objeto sólo tiene la localización de las variables globales, no de su tipo. Esto es por lo que los ficheros de cabeceras son necesarios

```
/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


#include <stdio.h>  /* Necesito NULL */


/* En 2.2.3 /usr/include/linux/version.h se incluye
 * una macro para esto, pero 2.0.35 no lo hace - por lo
 * tanto lo añado aquí si es necesario */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Emmanuel Papirakis:
 *
 * Los nombres de parámetros son ahora (2.2)
 * manejados en una macro.
 * El núcleo no resuelve los nombres de los
 * símbolos como parecía que tenía que hacer.
 *
 * Para pasar parámetros a un módulo, tienes que usar una macro
 * definida en include/linux/modules.h (linea 176).
 * La macro coge dos parámetros. El nombre del parámetro y
 * su tipo. El tipo es una letra entre comillas.
 * Por ejemplo, "i" debería de ser un entero y "s" debería
 * de ser una cadena de caracteres.
 */


char *str1, *str2;


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(str1, "s");
MODULE_PARM(str2, "s");
#endif


/* Inicializa el módulo - muestra los parámetros */
int init_module()
{
  if (str1 == NULL || str2 == NULL) {
    printk("La próxima vez, haz insmod param str1=<algo>");
    printk("str2=<algo>\n");
  } else
    printk("Cadenas de caracteres:%s y %s\n", str1, str2);
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  printk("Si intentas hacer insmod a este módulo dos veces,");
  printk("(sin borrar antes (rmmod)\n");
  printk("al primero), quizás obtengas el mensaje");
  printk("de error:\n");
  printk("'el símbolo para el parámetro str1 no ha sido encontrado'.\n");
#endif

  return 0;
}


/* Limpieza */
void cleanup_module()
{
}
```

Capítulo 7

Llamadas al Sistema

Hasta hace poco, la única cosa que hemos hecho era usar mecanismos bien definidos del núcleo para registrar ficheros `proc` y manejadores de dispositivos. Esto está bien si quieres hacer algo a través de lo que los programadores del núcleo quisieran, tal como escribir un controlador de dispositivo. Pero, ¿y si quieres escribir algo inusual, para cambiar el comportamiento del sistema de alguna forma? Entonces, te encuentras sólo.

Aquí es dónde la programación del núcleo se vuelve peligrosa. Escribiendo el ejemplo siguiente eliminamos la llamada al sistema `open`. Esto significa que no podría abrir ningún fichero, no podría ejecutar ningún programa, y no podría `apagar` la computadora. Tengo que pulsar el interruptor. Afortunadamente, no se muere ningún fichero. Para asegurarse de que no pierdes ningún fichero, por favor ejecuta `sync` correctamente antes de hacer el `insmod` y el `rmmod`.

Olvídate de los ficheros `/proc`, olvídate de los ficheros de los dispositivos. Son sólo detalles menores. El mecanismo *real* de comunicación entre los procesos y el núcleo, el que es usado por todos los procesos, son las llamadas al sistema. Cuando un proceso pide un servicio al núcleo (tal como abrir un fichero, duplicarse en un nuevo proceso, o pedir más memoria), este es el mecanismo usado. Si quieres cambiar el comportamiento del núcleo de formas interesantes, este es el sitio para hacerlo. Si quieres saber ver qué llamadas al sistema usa un programa, ejecuta `strace` ⟨`comando`⟩ ⟨`argumentos`⟩.

En general, un proceso no se supone que pueda acceder al núcleo. No puede acceder a la memoria del núcleo y no puede llamar a las funciones del núcleo. El hardware de la CPU fuerza esto (este es el motivo por el que es llamado 'modo protegido'). Las llamadas al sistema son una excepción a esta regla general. Lo que sucede es que el proceso rellena los registros con los valores apropiados y entonces llama a una instrucción especial, la cual salta a una posición previamente definida en el núcleo (por supuesto, la posición es leíble por los procesos de usuario, pero no se puede escribir en ella). Bajo las CPUs Intel, esto es realizado por medio de la interrupción 0x80. El hardware sabe que una vez que saltas a esta localización, no te estarás ejecutando más en modo usuario restringido, pero sí como núcleo del sistema operativo — y entonces se te permite hacer todo lo que quieras.

A la posición en el núcleo a la que un proceso puede saltar es llamada `system_call`. El procedimiento en la posición verifica el número de la llamada al sistema, el cual dice al núcleo qué servicio ha pedido el proceso. Entonces, mira en la tabla de llamadas al sistema (`sys_call_table`) para ver la dirección de la función del núcleo a llamar. A continuación llama a la función, y después retorna, hace unos pocos chequeos del sistema y regresa al proceso (o a un proceso diferente, si el tiempo del proceso ha finalizado). Si quieres leer este código, está en el fichero fuente `arch/`<`architecture`>`/kernel/entry.S`, después de la linea `ENTRY(system_call)`.

Por lo tanto, si queremos cambiar la forma en la que una cierta llamada al sistema trabaja, lo que necesitamos hacer es escribir nuestra propia función para implementarla (usualmente añadiendo un poco de nuestro código) y entonces cambiar el puntero en `sys_call_table` para que apunte a nuestra función. Porque quizás la borremos después y no queremos dejar el sistema en un estado inestable, es importante que `cleanup_module` restaure la tabla a su estado original.

El código fuente aquí es un ejemplo de un módulo del núcleo. Queremos 'espiar' a un cierto usuario, e

imprimir un mensaje (con printk) cuando el usuario abra un fichero. Hasta aquí, nosotros reemplazamos la llamada al sistema para abrir un fichero con nuestra propia función, llamada `our_sys_open`. Esta función verifica el uid (identificación del usuario) del proceso actual, y si es igual al uid lo espiamos, llamando a `printk` para mostrar el nombre del fichero a ser abierto. Entonces, de todas formas, llama a la función original `open` con los mismos parámetros, para realmente abrir el fichero.

La función `init_module` reemplaza la localización apropiada en `sys_call_table` y mantiene el puntero original en una variable para restaurar todo cuando regresemos a la normalidad. Esta aproximación es peligrosa, por la posibilidad de que dos módulos del núcleo cambien la misma llamada al sistema. Imagínate que tenemos dos módulos del núcleo, A y B. La llamada al sistema de A será A_open y la llamada al sistema de B será B_open. Ahora, cuando A está insertada en el núcleo, la llamada al sistema es reemplazada con A_open, la cual llamará a la sys_open original cuando esté hecho. A continuación, B es insertado en el núcleo, el cual reemplaza la llamada al sistema con B_open, que ejecutará la llamada al sistema que él piensa que es la original, A_open, cuando esté hecho.

Ahora, si B se quita primero, todo estará bien — simplemente restaurará la llamada al sistema a A_open, la cual llamará a la original. En cambio, si se quita A y después se quita B, el sistema se caerá. El borrado de A restaurará la llamada al sistema original, sys_open, sacando a B fuera del bucle. Entonces, cuando B es borrado, se restaura la llamada al sistema que **él** piensa que es la original, A_open, la cual no está más en memoria. A primera vista, parece que podemos resolver este problema particular verificando si la llamada al sistema es igual a nuestra función open y si lo es no cambiándola (por lo tanto B no cambiará la llamada al sistema cuando es borrado), lo que causará un problema peor. Cuando A es borrado, a él le parece que la llamada al sistema fue cambiada a B_open y por lo tanto no apuntará más a A_open, y por lo tanto no la restaurará a sys_open antes de que sea borrado de memoria. Desgraciadamente, B_open aún intentará llamar a A_open la cual no estará mas allí, por lo que incluso sin quitar B el sistema se caerá.

Pienso en dos formas de prevenir este problema. El primero es restaurar la llamada al valor original, sys_open. Desgraciadamente, sys_open no es parte de la tabla del sistema del núcleo en `/proc/ksyms`, por lo tanto no podemos acceder a ella. La otra solución es usar un contador de referencias para prevenir que root borre el módulo una vez que está cargado. Esto es bueno para la producción de módulos, pero malo para un ejemplo de aprendizaje — que es por lo que no lo hice aquí.

syscall.c

```
/* syscall.c
 *
 * Ejemplo de llamada al sistema "robando"
 */


/* Copyright (C) 1998-99 por Ori Pomerantz */


/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.h>  /* La lista de llamadas al sistema */
```

```
/* Para el actual estructura (proceso), necesitamos esto
 * para conocer quién es el usuario actual. */
#include <linux/sched.h>



/* En 2.2.3 /usr/include/linux/version.h se incluye
 * una macro para esto, pero 2.0.35 no lo hace - por lo
 * tanto lo añado aquí si es necesario */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>
#endif



/* La tabla de llamadas al sistema (una tabla de funciones).
 * Nosotros justamente definimos esto como externo, y el
 * núcleo lo rellenerá para nosotros cuando instalemos el módulo
 */
extern void *sys_call_table[];


/* UID que queremos espiar - será rellenado desde la
 * linea de comandos */
int uid;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(uid, "i");
#endif

/* Un puntero a la llamada al sistema original. El motivo para
 * mantener esto, mejor que llamar a la función original
 * (sys_open), es que alguien quizás haya reemplazado la
 * llamada al sistema antes que nosotros. Destacar que esto
 * no es seguro al 100%, porque si otro módulo reemplaza sys_open
 * antes que nosotros, entonces cuando insertemos llamaremos
 * a la función en ese módulo - y quizás sea borrado
 * antes que nosotros.
 *
 * Otro motivo para esto es que no podemos tener sys_open.
 * Es una variable estática, por lo tanto no es exportada. */
asmlinkage int (*original_call)(const char *, int, int);
```

```
/* Por algún motivo, en 2.2.3 current-uid me da cero, en vez de
 * la ID real del usuario. He intentado encontrar dónde viene mal,
 * pero no lo he podido hacer en un breve periodo de tiempo, y
 * soy vago - por lo tanto usaremos la llamada al sistema para
 * obtener el uid, de la forma que un proceso lo haría.
 *
 * Por algún motivo, después de que recompilara el núcleo este
 * problema se ha ido.
 */
asmlinkage int (*getuid_call)();



/* La función con la que reemplazaremos sys_open (la
 * función llamada cuando llamas a la llamada al sistema open).
 * Para encontrar el prototipo exacto, con el número y tipo de
 * argumentos, encontramos primero la función original (es en
 * fs/open.c).
 *
 * En teoría, esto significa que estamos enlazados a la versión
 * actual del núcleo. En la práctica, las llamadas al sistema nunca
 * cambian (se destruirían naufragando y requerirían que los programas
 * fuesen recompilados, ya que las llamadas al sistema son las
 * interfaces entre el núcleo y los procesos).
 */
asmlinkage int our_sys_open(const char *filename,
                            int flags,
                            int mode)
{
  int i = 0;
  char ch;

  /* Checkea si este es el usuario que estamos espiando */
  if (uid == getuid_call()) {
   /* getuid_call es la llamada al sistema getuid,
    * la cual nos da el uid del usuario que ejecutó
    * el proceso que llamó a la llamada al sistema
    * que tenemos. */

    /* Indica el fichero, si es relevante */
    printk("Fichero abierto por %d: ", uid);
    do {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, filename+i);
#else
      ch = get_user(filename+i);
#endif
      i++;
      printk("%c", ch);
    } while (ch != 0);
    printk("\n");
```

```
  }

  /* Llamamos a la sys_open original - en otro caso, perdemos
   * la habilidad para abrir ficheros */
  return original_call(filename, flags, mode);
}



/* Inicializa el módulo - reemplaza la llamada al sistema */
int init_module()
{
  /* Peligro - muy tarde para él ahora, pero quizás
   * la próxima vez... */
  printk("Soy peligroso. Espero que hayas hecho un ");
  printk("sync antes de insertarme.\n");
  printk("Mi duplicado, cleanup_module(), es todavía");
  printk("más peligroso. Si\n");
  printk("valoras tu sistema de ficheros, será mejor ");
  printk("que hagas \"sync; rmmod\" \n");
  printk("cuando borres este módulo.\n");

  /* Mantiene un puntero a la función original en
   * original_call, y entonces reemplaza la llamada al sistema
   * en la tabla de llamadas al sistema con our_sys_open */
  original_call = sys_call_table[__NR_open];
  sys_call_table[__NR_open] = our_sys_open;

  /* Para obtener la dirección de la función para la
   * llamada al sistema foo, va a sys_call_table[__NR_foo]. */

  printk("Espiando el UID:%d\n", uid);

  /* Coje la llamada al sistema para getuid */
  getuid_call = sys_call_table[__NR_getuid];

  return 0;
}


/* Limpieza - libera el fichero apropiado de /proc */
void cleanup_module()
{
  /* Retorna la llamada al sistema a la normalidad */
  if (sys_call_table[__NR_open] != our_sys_open) {
    printk("Alguien más jugó con la llamada al sistema ");
    printk("open\n");
    printk("El sistema quizás haya sido dejado ");
    printk("en un estado iniestable.\n");
  }

  sys_call_table[__NR_open] = original_call;
```

```
}
```

Capítulo 8

Procesos Bloqueantes

¿Qué puedes hacer cuando alguien te pregunta por algo que no puedes hacer en el acto? Si eres un humano y estás siendo molestado por un humano, la única cosa que puedes decir es: 'Ahora no. Estoy ocupado. *¡Vete!*". Pero si eres un módulo del núcleo y estás siendo molestado por un proceso, tienes otra posibilidad. Puedes poner el proceso a dormir hasta que lo puedas atender. Después de todo, los procesos son puestos a dormir por el núcleo y todos son despertados al mismo tiempo (esta es la forma en la que múltiples procesos aparentan ejecutarse a la vez en una simple CPU).

Este módulo del núcleo es un ejemplo de esto. El fichero (llamado `/proc/sleep`) sólo puede ser abierto por un sólo proceso a la vez. Si el fichero ya está abierto, el módulo del núcleo llama a `module_interruptible_sleep_on`[1]. Esta función cambia el estatus de la tarea (una tarea es la estructura de datos del núcleo que mantiene información sobre un proceso y la llamada al sistema en la que está, si es que está en alguna) a `TASK_INTERRUPTIBLE`, lo que significa que la tarea no se ejecutará hasta que sea despertada de alguna forma, y se añade a `WaitQ`, la cola de tareas esperando para acceder al fichero. Entonces, la función llama al planificador para hacer un cambio de contexto a un proceso diferente, el cual tenga algún uso para la CPU.

Cuando un proceso ha acabado con el fichero, lo cierra, y se llama a `module_close`. Esta función despierta a todos los procesos en la cola (no hay un mecanismo para despertar a sólo uno de ellos). Entonces retorna y el proceso que acaba de cerrar el fichero puede continuar ejecutándose. A la vez, el planificador decide que ese proceso ya tuvo suficiente y le da el control de la CPU a otro proceso. Eventualmente, a uno de los procesos que estaba en la cola se le dará el control de la CPU por el planificador. Él empieza en el punto justo después de la llamada a `module_interruptible_sleep_on` [2]. Puede proceder a establecer un variable local para decir a todos los otros procesos que el fichero aún está abierto y seguir con su vida. Cuando otros procesos obtienen un poco de CPU, verán la variable global y volverán a dormir.

Para hacer nuestra vida más interesante, `module_close` no tiene un monopolio en el despertar de los procesos que están esperando para acceder al fichero. Una señal, tal como Ctrl-C (`SIGINT`) también puede despertar a un proceso[3] En este caso, queremos regresar inmediatamente con `-EINTR`. Esto es importante para que los usuarios puedan, por ejemplo, matar el proceso antes de que reciba el fichero.

Hay un punto más que recordar. Algunas veces los procesos no quieren dormir, quieren coger lo que quieren inmediatamente, o dirán que no pueden hacerlo. Tales procesos usan la flag `O_NONBLOCK` cuando abren el fichero. El núcleo se supone que responde retornando con el código de error `-EAGAIN` de las operaciones que en otro caso bloquearían, tales como abrir un fichero en este ejemplo. El programa cat_noblock, disponible en el directorio fuente para este capítulo, puede ser usado para abrir el fichero con `O_NONBLOCK`.

sleep.c

[1] La forma más fácil de mantener un fichero abierto es con `tail -f`.

[2] Esto significa que el proceso aún está en modo núcleo — en tanto que el proceso es concerniente, él emite la llamada al sistema `open` y la llamada al sistema no ha regresado todavía. El proceso no conoce a nadie que usara la CPU durante la mayoría del tiempo entre el momento en el que hizo la llamada y el momento en el que regresó.

[3] Esto es porque nosotros usamos `module_interruptible_sleep_on`. Podíamos haber usado `module_sleep_on` en vez de ella, pero tendríamos que el resultado serían usuarios extremadamente enfadados cuyos control Cs son ignorados.

```
/* sleep.c - crea un fichero /proc, y si varios procesos
 * intentan abrirlo al mismo tiempo, los pone
 * a todos a dormir */

/* Copyright (C) 1998-99 por Ori Pomerantz */


/* Los ficheros de cabeceras necesarios */


/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necesario porque usamos el sistema de ficheros proc */
#include <linux/proc_fs.h>

/* Para poner los procesos a dormir y despertarlos */
#include <linux/sched.h>
#include <linux/wrapper.h>



/* En 2.2.3 /usr/include/linux/version.h se incluye una
 * macro para esto, pero 2.0.35 no lo hace - por lo tanto
 * lo añado aquí si es necesario. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* para get_user y put_user */
#endif


/* Las funciones de fichero del módulo ********************** */

/* Aquí mantenemos el último mensaje recibido, para probar
 * que podemos procesar nuestra entrada */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];


/* Como usamos las estructuras de operaciones de ficheros, no
 * podemos usar las provisiones de salida de proc especiales - tenemos
```

```
 * que usar una función estándar de lectura, que es esta*/
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
    struct file *file,   /* El fichero a leer */
    char *buf, /* El buffer donde poner los datos (en el
                * segmento de usuario) */
    size_t len,  /* La longitud del buffer */
    loff_t *offset) /* Desplazamiento en el fichero - ignóralo */
#else
static int module_output(
    struct inode *inode, /* El inodo a leer */
    struct file *file,   /* El fichero a leer */
    char *buf, /* El buffer donde poner los datos (en el
                * segmento de usuario) */
    int len)  /* La longitud del buffer */
#endif
{
  static int finished = 0;
  int i;
  char message[MESSAGE_LENGTH+30];

  /* Retorna 0 para significar el final del fichero - que no
   * tenemos nada más que decir en este punto. */
  if (finished) {
    finished = 0;
    return 0;
  }

  /* Si no entiendes esto ahora, eres un
   * programador del núcleo sin esperanza. */
  sprintf(message, "Ultima entrada:%s\n", Message);
  for(i=0; i<len && message[i]; i++)
    put_user(message[i], buf+i);

  finished = 1;
  return i;  /* Retorna el número de bytes "leídos" */
}


/* Esta función recibe la entrada del usuario cuando
 * el usuario escribe el fichero /proc. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
    struct file *file,   /* El mismo fichero */
    const char *buf,     /* El buffer con la entrada */
    size_t length,       /* La longitud del buffer */
    loff_t *offset)      /* desplazamiento del fichero - ignóralo */
#else
static int module_input(
    struct inode *inode, /* El inodo del fichero */
    struct file *file,   /* El mismo fichero */
    const char *buf,     /* El buffer con la entrada */
```

```
    int length)          /* La longitud del buffer */
#endif
{
  int i;

  /* Pone la entrada en Message, donde module_output
   * más tarde será capaz de usarlo */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
     get_user(Message[i], buf+i);
#else
  Message[i] = get_user(buf+i);
#endif
/* queremos un estándar, cadena de caracteres terminada en cero */
  Message[i] = '\0';

  /* Necesitamos devolver el número de caracteres
   * de entrada usados */
  return i;
}

/* 1 si el fichero está actualmente abierto por alguien */
int Already_Open = 0;

/* Cola de procesos que quieren nuestro fichero */
static struct wait_queue *WaitQ = NULL;


/* Llamado cuando el fichero /proc se abre */
static int module_open(struct inode *inode,
                       struct file *file)
{
  /* Si las banderas del fichero incluyen O_NONBLOCK, esto
   * significa que el proceso no quiere esperar al fichero.
   * En este caso, si el fichero ya está abierto, deberemos
   * fallar con -EAGAIN, significando que "tienes que intentarlo
   * otra vez", en vez de bloquear un proceso que tendría que
   * estar despierto. */
  if ((file->f_flags & O_NONBLOCK) && Already_Open)
    return -EAGAIN;

  /* Este es el sitio correcto para MOD_INC_USE_COUNT
   * porque si un proceso está en el bucle, que
   * está dentro del módulo, el módulo del núcleo no
   * debería ser quitado. */
  MOD_INC_USE_COUNT;

  /* Si el fichero ya está abierto, espera hasta que no lo esté */
  while (Already_Open)
  {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    int i, is_sig=0;
```

```
#endif

    /* Esta función pone el proceso actual,
     * incluyendo algunas llamada al sistema, como nosotros,
     * a dormir. La ejecución será retomada correctamente después
     * de la llamada a la función, o porque alguien
     * llamó a wake_up(&WaitQ) (sólo module_close hace esto,
     * cuando el fichero se cierra) o cuando una señal, como
     * Ctrl-C, es enviada al proceso */
    module_interruptible_sleep_on(&WaitQ);

    /* Si despertamos porque tenemos una señal no estamos
     * bloqueando, retornamos -EINTR (falla la llamada al
     * sistema). Esto permite a los procesos ser matados o
     * parados. */


/*
 * Emmanuel Papirakis:
 *
 * Esta es una pequeña actualización para trabajar con 2.2.*. Las
 * señales son ahora contenidas en dos palabras (64 bits) y son
 * almacenadas en una estructura que contiene un array de dos
 * unsigned longs. Ahora tenemos que realizar 2 chequeos en nuestro if.
 *
 * Ori Pomerantz:
 *
 * Nadie me prometió que no usarían nunca más de 64 bits, o
 * que este libro no sería usado para una versión de Linux
 * con un tamaño de palabra de 16 bits. En cualquier caso este
 * código debería de funcionar.
 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)

    for(i=0; i<_NSIG_WORDS && !is_sig; i++)
      is_sig = current->signal.sig[i] &
        ~current->blocked.sig[i];
    if (is_sig) {
#else
    if (current->signal & ~current->blocked) {
#endif
      /* Es importante poner MOD_DEC_USE_COUNT aquí.
       * porque los procesos dónde open es interrumpido
       * no tendrán nunca un close correspondiente. Si
       * no decrementamos el contador de uso aquí, lo dejaremos
       * con un valor positivo el cual no nos dará
       * la oportunidad de llegar hasta 0, dándonos un módulo inmortal,
       * que sólo se puede matar reiniciando la máquina. */
      MOD_DEC_USE_COUNT;
      return -EINTR;
    }
  }
```

```
  /* Si estamos aquí, Already_Open debe ser cero */

  /* Abre el fichero */
  Already_Open = 1;
  return 0;  /* Permite el acceso */
}



/* Llamado cuando el fichero /proc se cierra*/
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  /* Establece Already_Open a cero, por lo tanto uno de los procesos
   * en WaitQ será capaz de establecer Already_Open otra vez a uno y
   * abrir el fichero. Todos los otros procesos serán llamados cuando
   * Already_Open vuelva a ser uno, por lo tanto volverán a
   * dormir. */
  Already_Open = 0;

  /* Despertamos a todos los procesos en WaitQ, por lo tanto si
   * alguien está esperando por el fichero, lo puede tener. */
  module_wake_up(&WaitQ);

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* finalizado con éxito */
#endif
}



/* Esta función decide cuando permite una operación (retorna cero)
 * o no la permite (retorna distinto de cero lo cual indica porque
 * no es permitida).
 *
 * Las operaciones pueden ser una de los siguientes valores:
 * 0 - Ejecuta (ejecuta el "file" - sin pérdida de significado en
 *     nuestro caso)
 * 2 - Escribe (entrada al módulo del núcleo)
 * 4 - Lectura (salida desde el módulo del núcleo)
 *
 * Esta es la función real que chequea los permisos del
 * fichero. Los permisos retornados por ls -l son sólo
 * para referencia, y pueden ser sobreescritos aquí.
 */
```

```
static int module_permission(struct inode *inode, int op)
{
  /* Permitimos a todo el mundo leer de nuestro módulo, pero
   * sólo root (uid 0) puede escribir en el */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;

  /* Si es otro, el acceso es denegado */
  return -EACCES;
}


/* Estructuras para registrar como fichero /proc, con
 * punteros a todas las funciones relevantes. ********** */

/* Operaciones de fichero para nuestro fichero proc. Aquí es
 * donde colocamos los punteros a todas las funciones llamadas
 * cuando alguien intenta hacer algo a nuestro fichero. NULL
 * significa que no queremos tratar con algo. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "lee" del fichero */
    module_input,   /* "escribe" al fichero */
    NULL,  /* readdir */
    NULL,  /* seleccionar */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,/* llamado cuando el fichero /proc es abierto */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* borrado */
#endif
    module_close      /* llamado cuando es cerrado */
  };



/* Las operaciones de inodo para nuestro fichero proc. Las necesitamos
 * para tener algo donde especificar la estructura
 * de operaciones del fichero que queremos usar, y las funciones que
 * usamos para los permisos. También es posible especificar funciones que
 * pueden ser llamadas por alguien más, lo cual se puede realizar en un
 * inodo (como no queremos ninguna, ponemos NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* crear */
    NULL, /* lookup */
    NULL, /* enlazar */
    NULL, /* desenlazar */
    NULL, /* enlace simbólico */
    NULL, /* mkdir */
```

```
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* renonmbrar */
    NULL, /* leer enlace */
    NULL, /* seguir enlace */
    NULL, /* lee página */
    NULL, /* excribe página */
    NULL, /* bmap */
    NULL, /* corta */
    module_permission /* chequea los permisos */
  };

/* Entrada del directorio */
static struct proc_dir_entry Our_Proc_File =
  {
    0, /* Número de inodo - ignóralo, será rellenado por
        * proc_register[_dynamic] */
    5, /* Longitud del nombre del fichero */
    "sleep", /* El nombre del fichero */
    S_IFREG | S_IRUGO | S_IWUSR,
    /* Modo del fichero - este es un fichero normal que
     * puede ser leído por su dueño, su grupo, y por
     * todo el mundo. Además, su dueño puede escribir en él.
     *
     * Realmente, este campo es sólo para referencia, es
     * module_permission el que realiza el chequeo actual.
     * Puede usar este campo, pero en nuestra implementación no
     * lo hace, por simplificación. */
    1,  /* Número de enlaces (directorios donde el fichero
         * es referenciado) */
    0, 0,  /* El uid y gid para el fichero - se los damos
            * a root */
    80, /* El tamaño del fichero indicado por ls. */
    &Inode_Ops_4_Our_Proc_File,
    /* Un puntero a la estructura de inodos para
     * el fichero, si lo necesitamos. En nuestro caso
     * lo hacemos, porque necesitamos una función write (de escritura). */
    NULL  /* La función read para el fichero.
           * Irrelevante, porque lo ponemos
           * en la estructura del inodo anterior */
  };



/* Inicialización y Limpieza del módulo *************** */


/* Inicializa el módulo - registra el fichero proc */
int init_module()
{
  /* Tiene éxito si proc_register_dynamic tiene éxito,
   * falla en otro caso */
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* proc_root es el directorio raiz para el sistema de
   * ficheros proc (/proc). Es decir, donde queremos que sea
   * localizado nuestro fichero. */
}


/* Limpieza - libera nuestro fichero en /proc. Esto puede
 * ser peligroso si aún hay procesos esperando en WaitQ, porque
 * ellos están dentro de nuestra función open, la cual será
 * descargada. Explicaré que hacer para quitar un módulo
 * del núcleo en tal caso en el capítulo 10. */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

Capítulo 9

Reemplazando printk's

Al principio (capítulo 1), dije que X y la programación de módulos del núcleo no se mezclaban. Esto es verdad mientras se está desarrollando el módulo del núcleo, pero en el uso actual quieres ser capaz de enviar mensajes a cualquier tty[1] el comando que viene del módulo. Esto es importante para identificar errores después de que el módulo del núcleo es liberado, porque será usado por todos ellos.

La forma en la que esto se hace es usando `current`, un puntero a la actual tarea ejecutándose, para obtener la estructura tty de la tarea actual. Entonces, miramos dentro de la estructura tty para encontrar un puntero a una función de escritura de cadenas de caracteres, la cual usamos para escribir una cadena de caracteres a la tty.

printk.c

```
/* printk.c - envía salida textual al tty en el que estás
 * ahora, sin importarle cuando es pasado
 * a través de X11, telnet, etc. */


/* Copyright (C) 1998 por Ori Pomerantz */


/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necesarios aquí */
#include <linux/sched.h>    /* Para el actual */
#include <linux/tty.h>      /* Para las declaraciones de tty */
```

[1] **T**ele**ty**pe, originalmente una combinación de teclado-impresora usado para comunicarse con un sistema Unix, y hoy una abstracción para el flujo de texto usado para un programa Unix, cuando es un terminal físico, un xterm, una pantalla X, una conexión de red usada con telnet, etc.

```
/* Imprime la cadena de caracteres al tty apropiado, el
 * que usa la tarea actual */
void print_string(char *str)
{
  struct tty_struct *my_tty;

  /* La tty para la tarea actual */
  my_tty = current->tty;

  /* Si my_tty es NULL, significa que la actual tarea
   * no tiene tty en la que puedas imprimir (esto es posible, por
   * ejemplo, si es un demonio). En este caso, no hay nada
   * que se pueda hacer. */
  if (my_tty != NULL) {

    /* my_tty->driver es una estructura que mantiene las funciones
     * de tty, una de las cuales (write) es usada para
     * escribir cadenas de caracteres a la tty. Puede ser usada
     * para coger una cadena de caracteres del segmento de memoria
     * del usuario o del segmento de memoria del núcleo.
     *
     * El primer parámetro de la función es la tty en la que
     * hay que escribir, porque la misma función puede
     * ser normalmente usada para todas las ttys de un cierto
     * tipo. El segundo parámetro controla cuando la función
     * recibe una cadena de caracteres de la memoria del núcleo
     * (falsa, 0) o desde la memoria del usuario (verdad, distinto
     * de cero). El tercer parámetro es un puntero a la cadena
     * de caracteres, y el cuarto parámetro es la longitud de la
     * cadena de caracteres.
     */
    (*(my_tty->driver).write)(
        my_tty, /* La misma tty */
        0, /* No cogemos la cadena de caracteres de la memoria de usuario */
str, /* Cadena de caracteres */
strlen(str));  /* Longitud */

    /* Las ttys fueron originalmente dispositivos hardware, las
     * cuales (usualmente) se adherían estrictamente al estándar
     * ASCII. De acuerdo con ASCII, para mover una nueva linea
     * necesitas dos caracteres, un retorno de carro y un salto
     * de linea. En Unix, en cambio, el salto de linea ASCII
     * es usado para ambos propósitos - por lo tanto no podemos
     * usar \n, porque no tendrá un retorno de carro y la siguiente
     * linea empezará en la columna siguiente
     *                                      después del paso de linea.
     *
     * BTW, este es el motivo por el que el formato de un fichero de
     * texto es diferente entre Unix y Windows. En CP/M y sus derivados,
     * tales como MS-DOS y Windows, el estándar ASCII fue estrictamente
     * adherido, y entonces una nueva linea requiere un salto de linea
     * y un retorno de carro.
```

```
     */
    (*(my_tty->driver).write)(
      my_tty,
      0,
      "\015\012",
      2);
  }
}


/* Inicialización y Limpieza del módulo ***************** */


/* Inicializa el módulo - registra el fichero proc */
int init_module()
{
  print_string("Módulo insertado");

  return 0;
}


/* Limpieza - libera nuestro fichero de /proc */
void cleanup_module()
{
  print_string("Módulo borrado");
}
```

Capítulo 10

Planificando Tareas

Muy frecuentemente, tenemos tareas ‘domésticas’ que tienen que ser realizadas en un cierto tiempo, o todas frecuentemente. Si la tarea es realizada por un proceso, lo haremos poniéndolo en el fichero `crontab`. Si la tarea es realizada por un módulo del núcleo, tenemos dos posibilidades. La primera es poner un proceso en el fichero `crontab` el cual despertará el módulo con una llamada al sistema cuando sea necesario, por ejemplo abriendo un fichero. Esto es terriblemente ineficiente, de cualquier modo — ejecutamos un proceso fuera de `crontab`, leer un nuevo ejecutable de memoria, y todo esto para despertar al módulo del núcleo el cual está de todas formas en memoria.

En vez de hacer esto, podemos crear una función que será llamada una vez en cada interrupción del reloj. La forma en la que hacemos esto es creando una tarea, mantenida en una `estructura tq_struct`, la cual mantendrá un puntero a la función. Entonces, usamos `queue_task` para poner esta tarea en una lista de tareas llamada `tq_timer`, que es la lista de tareas a ser ejecutadas en la siguiente interrupción de reloj. Porque queremos que se mantenga la función siendo ejecutada, necesitamos ponerla otra vez en `tq_timer` cuando es llamada, para la siguiente interrupción del reloj.

Hay un punto más que necesitamos recordar aquí. Cuando un módulo es quitado por `rmmod`, primero se verificado su contador de referencias. Si es cero, se llama a `module_cleanup`. Entonces, se quita el módulo de memoria con todas sus funciones. Nadie controla si la lista de tareas del reloj contiene un puntero a una de estas funciones, las cuales no estarán más disponibles. Años después (desde la perspectiva de la computadora, para la perspectiva de un humano no es nada, menos de una milésima de segundo), el núcleo tiene una interrupción de reloj e intenta llamar a la función en la lista de tareas. Desgraciadamente, la función ya no está más allí. En la mayoría de los casos, la página de memoria donde estaba está sin utilizar, y obtiene un feo mensaje de error. Pero si algún otro código está ahora situado en la misma posición de memoria, las cosas se puede poner **muy** feas. Desgraciadamente, no tenemos una forma fácil de eliminar una tarea de una lista de tareas.

Como `cleanup_module` no puede retornar con un código de error (es una función void), la solución es no dejar que retorne. En vez de ello, se llama a `sleep_on` o `module_sleep_on`[1] para poner el proceso `rmmod` a dormir. Antes de eso, informa a la función llamada por la interrupción del reloj para que pare de apuntarse estableciendo una variable global. Entonces, en la siguiente interrupción del reloj, el proceso `rmmod` será despertado, cuando nuestra función no está más en la cola y es seguro quitar el módulo.

sched.c

```
/* sched.c - planifica una función para ser llamada en
 * cada interrupción del reloj */


/* Copyright (C) 1998 por Ori Pomerantz */
```

[1] Ambas son realmente lo mismo.

```
/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Necesario porque usamos el sistema de ficheros proc */
#include <linux/proc_fs.h>

/* Planificamos tareas aquí */
#include <linux/tqueue.h>

/* También necesitamos la habilidad para ponernos a dormir
 * y despertanos más tarde */
#include <linux/sched.h>

/* En 2.2.3 /usr/include/linux/version.h se incluye una
 * macro para esto, pero 2.0.35 no lo hace - por lo tanto
 * lo añado aquí si es necesario. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* El número de veces que la interrupción del reloj
 * ha sido llamada */
static int TimerIntrpt = 0;


/* Esto lo usa cleanup, para evitar que el módulo
 * sea descargado mientras intrpt_routine está
 * aún en la cola de tareas */
static struct wait_queue *WaitQ = NULL;

static void intrpt_routine(void *);


/* La estructura de cola de tareas para esta tarea, de tqueue.h */
static struct tq_struct Task = {
  NULL,   /* Próximo elemento en la lista - queue_task hará
           * esto por nosotros */
  0,      /* Una bandera significando que todavía no hemos
   * insertado en la cola de tareas */
  intrpt_routine, /* La función a ejecutar */
  NULL    /* El parámetro void* para esta función */
```

```
};



/* Esta función será llamada en cada interrupción de reloj.
 * Nótese que el puntero *void - funciones de la tarea
 * puede ser usado para más de un propósito, obteniendo
 * cada vez un parámetro diferente. */

static void intrpt_routine(void *irrelevant)
{
  /* Incrementa el contador */
  TimerIntrpt++;

  /* Si cleanup nos quiere matar */
  if (WaitQ != NULL)
    wake_up(&WaitQ);   /* Ahora cleanup_module puede retornar */
  else
    /* Nos vuelve a poner en la cola de tareas */
    queue_task(&Task, &tq_timer);
}



/* Pone datos en el fichero del sistema de ficheros proc. */
int procfile_read(char *buffer,
                  char **buffer_location, off_t offset,
                  int buffer_length, int zero)
{
  int len;  /* Número de bytes usados actualmente */

  /* Esto es estático por lo tanto permanecerá en memoria
   * cuando deje esta función */
  static char my_buffer[80];

  static int count = 1;

  /* Damos toda nuestra información de una vez, por lo
   * tanto si alguien nos pregunta si tenemos más
   * información la respuesta debería de ser no.
   */
  if (offset > 0)
    return 0;

  /* Rellena el buffer y obtiene su longitud */
  len = sprintf(my_buffer,
                "Timer fue llamado %d veces\n",
                TimerIntrpt);
  count++;

  /* Dice a la función que nos ha llamado dónde
```

```
   * está el buffer */
  *buffer_location = my_buffer;

  /* Retorna la longitud */
  return len;
}


struct proc_dir_entry Our_Proc_File =
  {
    0, /* Número de inodo - ignóralo, será rellenado por
        * proc_register_dynamic */
    5, /* Longitud del nombre del fichero */
    "sched", /* El nombre del fichero */
    S_IFREG | S_IRUGO,
    /* Modo del fichero - este es un fichero normal que puede
     * ser leido por su dueño, su grupo, y por todo el mundo */
    1,  /* Número de enlaces (directorios donde
         * el fichero es referenciado) */
    0, 0,  /* El uid y gid para el fichero - se lo damos
            * a root */
    80, /* El tamaño del fichero indicado por ls. */
    NULL, /* funciones que pueden ser realizadas en el
           * inodo (enlace, borrado, etc.) - no
           * soportamos ninguna. */
    procfile_read,
    /* La función read para este fichero, la función llamada
     * cuando alguien intenta leer algo de él. */
    NULL
    /* Podemos tener aquí una función para rellenar
     * el inodo del fichero, para permitirnos jugar con
     * los permisos, dueño, etc. */
  };


/* Inicializa el módulo - registra el fichero proc */
int init_module()
{
  /* Pone la tarea en la cola de tareas tq_timer, por lo
   * tanto será ejecutado en la siguiente interrupción del reloj */
  queue_task(&Task, &tq_timer);

  /* Tiene éxito si proc_register_dynamic tiene éxito.
   * falla en otro caso */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}
```

```
/* Limpieza */
void cleanup_module()
{
  /* libera nuestro fichero /proc */
  proc_unregister(&proc_root, Our_Proc_File.low_ino);

  /* Duerme hasta que intrpt_routine es llamada por última
   * vez. Esto es necesario, porque en otro caso, desasignaremos
   * la memoria manteniendo intrpt_routine y Task mientras
   * tq_timer aún las referencia. Destacar que no permitimos
   * señales que nos interrumpan.
   *
   * Como WaitQ no es ahora NULL, esto dice automáticamente
   * a la rutina de interrupción su momento de muerte. */
 sleep_on(&WaitQ);
}
```

Capítulo 11

Manejadores de Interrupciones

Excepto para el último capítulo, todo lo que hicimos en el núcleo lo hemos hecho como respuesta a un proceso preguntando por él, bien tratando con un fichero especial, enviando un `ioctl`, o a través de una llamada al sistema. Pero el trabajo del núcleo no es sólo responder a las peticiones de los procesos. Otro trabajo, que es un poco más importante, es hablar con el hardware conectado a la máquina.

Hay dos tipos de interacción entre la CPU y el resto del hardware de la computadora. El primer tipo es cuando la CPU da órdenes al hardware, el el otro es cuando el hardware necesita decirle algo a la CPU. La segunda, llamada interrupción, es mucho más difícil de implementar porque tiene que tratar con el hardware cuando sea conveniente, no la CPU. Los dispositivos hardware típicamente tienen una pequeña cantidad de ram, y si no quieres leer su información cuando está disponible, se pierde.

Bajo Linux, las interrupciones hardware son llamadas IRQs (abreviatura para **I**nterrupt **Re**q**u**ests)[1]. Hay dos tipos de IRQs, pequeñas y grandes. Una IRQ pequeña es una que se espera que dure un **muy** corto periodo de tiempo, durante el cual el resto de la máquina será bloqueada y ninguna otra interrupción será manejada. Una IRQ grande es una que puede durar mucho, y durante la cual otras interrupciones pueden ocurrir (pero no interrupciones desde el mismo dispositivo). Cuando sea posible, es mejor declarar un manejador de interrupciones para que sea grande.

Cuando la CPU recibe una interrupción, para lo que quiera que esté haciendo (a menos que se encuentre procesando una interrupción más prioritaria, en cuyo caso tratará con esta interrupción sólo cuando la más prioritaria se haya acabado), salva ciertos parámetros en la pila y llama al manejador de interrupciones. Esto significa que ciertas cosas no están permitidas en un manejador de interrupciones por si mismo, porque el sistema está en un estado indefinido. La solución a este problema es que el manejador de interrupciones hace lo que necesite hacer inmediatamente, usualmente leer algo desde el hardware o enviar algo al hardware, y entonces planificar el manejo de la nueva información en un tiempo posterior (esto se llama 'bottom half') y retorna. El núcleo está garantizado que llamará al bottom half tan pronto como sea posible — y cuando lo es, todo lo que está permitido en los módulos del núcleo será permitido.

La forma de implementar esto es llamar a `request_irq` para tener un manejador de interrupciones preparado cuando la IRQ relevante es recibida (hay 16 de ellas en las plataformas Intel). Esta función recibe el número de IRQ, el nombre de la función, flags, un nombre para `/proc/interrupts` y un parámetro para pasarle al manejador de interrupciones. Las flags pueden incluir `SA_SHIRQ` para indicar que estás permitiendo compartir la IRQ con otro manejador de interrupciones (usualmente porque un número de dispositivos hardware están en la misma IRQ) y `SA_INTERRUPT` para indicar que esta es una interrupción rápida. Esta función sólo tendrá éxito si no hay ya un manejador para esta IRQ, o si ya estás compartiéndola.

Entonces, desde el manejador de interrupciones, nos comunicamos con el hardware y entonces usamos `queue_task_irq` con `tq_immediate` y planificamos el bottom half. El motivo por el que no podemos usar la `queue_task` estándar en la versión 2.0 es que la interrupción quizás suceda en el medio de una `queue_task` de alguien [2]. Necesitamos `mark_bh` porque en las primeras versiones de Linux sólo había un array de 32 bottom halves, y ahora uno de ellos (`BH_IMMEDIATE`) es usado para la lista enlazada de bottom

[1] Esta es una nomenclatura estándar de la arquitectura Intel donde Linux se originó.

[2] `queue_task_irq` está protegida de esto por un bloqueo global — en 2.2 no hay `queue_task_irq` y `queue_task` está protegida por un bloqueo.

halves de los controladores que no tienen una entrada bottom half asignada.

11.1 Teclados en la Arquitectura Intel

Atención: El resto de este capítulo es completamente específico de Intel. Si no estás trabajando en una plataforma Intel, no funcionará. Ni siquiera intentes compilar el siguiente código.

Yo tuve un problema escribiendo el código de ejemplo para este capítulo. Por una parte, para un ejemplo, es útil que se ejecutara en las computadores de todo el mundo con resultados significativos. Por otra parte, el núcleo ya incluye controladores de dispositivos para todos los dispositivos comunes, y aquellos controladores de dispositivos no coexistirán con lo que voy a escribir. La solución que encontré fue escribir algo para la interrupción del teclado, y deshabilitar antes el manejador normal de interrupción del teclado. Como está definido como un símbolo estático en los fichero fuentes del núcleo (específicamente, `drivers/char/keyboard.c`), no hay forma de restaurarlo. Antes de instalar este código, haz en otro terminal `sleep 120 ; reboot` si es que valoras tu sistema.

Este código se registra para la IRQ 1, la cual es la IRQ controlada por el teclado bajo las arquitecturas Intel. Entonces, cuando recibe una interrupción de teclado, lee el estado del teclado (que es el propósito de `inb(0x64)`) y examina el código, que es el valor devuelto por el teclado. Tan pronto como el núcleo piensa que es factible, ejecuta `got_char` que da el código de la tecla usada (los siete primeros bits del código leído) y cuando está siendo presionado (si el octavo bit es cero) o soltado (si es uno).

intrpt.c

```
/* intrpt.c - Un manejador de interrupciones. */


/* Copyright (C) 1998 por Ori Pomerantz */



/* Los ficheros de cabeceras necesarios */

/* Estándar en los módulos del núcleo */
#include <linux/kernel.h>   /* Estamos haciendo trabajo del núcleo */
#include <linux/module.h>   /* Específicamente, un módulo */

/* Distribuido con CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/sched.h>
#include <linux/tqueue.h>

/* Queremos una interrupción */
#include <linux/interrupt.h>

#include <asm/io.h>


/* En 2.2.3 /usr/include/linux/version.h se incluye una
 * macro para esto, pero 2.0.35 no lo hace - por lo tanto
 * lo añado aquí si es necesario. */
```

```
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Bottom Half - esto será llamado por el núcleo
 * tan pronto como sea seguro hacer todo lo normalmente
 * permitido por los módulos del núcleo. */
static void got_char(void *scancode)
{
  printk("Código leído %x %s.\n",
    (int) *((char *) scancode) & 0x7F,
    *((char *) scancode) & 0x80 ? "Liberado" : "Presionado");
}


/* Esta función sirve para las interrupciones de teclado. Lee
 * la información relevante desde el teclado y entonces
 * planifica el bottom half para ejecutarse cuando el núcleo
 * lo considere seguro. */
void irq_handler(int irq,
                 void *dev_id,
                 struct pt_regs *regs)
{
  /* Estas variables son estáticas porque necesitan ser
   * accesibles (a través de punteros) por la rutina bottom
   * half. */
  static unsigned char scancode;
  static struct tq_struct task =
        {NULL, 0, got_char, &scancode};
  unsigned char status;

  /* Lee el estado del teclado */
  status = inb(0x64);
  scancode = inb(0x60);

  /* Planifica el bottom half para ejecutarse */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  queue_task(&task, &tq_immediate);
#else
  queue_task_irq(&task, &tq_immediate);
#endif
  mark_bh(IMMEDIATE_BH);
}



/* Inicializa el módulo - registra el manejador de IRQs */
int init_module()
{
  /* Como el manejador de teclado no coexistirá con
```

```
   * otro manejador, tal como nosotros, tenemos que deshabilitarlo
   * (liberar su IRQ) antes de hacer algo. Ya que nosotros
   * no conocemos dónde está, no hay forma de reinstalarlo
   * después - por lo tanto la computadora tendrá que ser reiniciada
   * cuando halla sido realizado.
   */
  free_irq(1, NULL);

  /* Petición IRQ 1, la IRQ del teclado, para nuestro
   * irq_handler. */
  return request_irq(
    1,  /* El número de la IRQ del teclado en PCs */
    irq_handler,  /* nuestro manejador */
    SA_SHIRQ,
    /* SA_SHIRQ significa que queremos tener otro
     * manejador en este IRQ.
     *
     * SA_INTERRUPT puede ser usado para
     * manejarla en una interrupción rápida.
     */
    "test_keyboard_irq_handler", NULL);
}


/* Limpieza */
void cleanup_module()
{
  /* Esto está aquí sólo para completar. Es totalmente
   * irrelevante, ya que no tenemos forma de restaurar
   * la interrupción normal de teclado, por lo tanto
   * la computadora está totalmente inservible y tiene que
   * ser reiniciada. */
  free_irq(1, NULL);
}
```

Capítulo 12

Multi–Procesamiento Simétrico

Una de las formas más fáciles (léase, baratas) de aumentar el rendimiento es poner más de una CPU en la placa. Esto se puede realizar haciendo que CPUs diferentes tengan trabajos diferentes (multi–procesamiento asimétrico) o haciendo que todos ellos se ejecuten en paralelo, realizando el mismo trabajo (multi–procesamiento simétrico, a.k.a. SMP). El multi–procesamiento asimétrico, efectivamente, requiere conocimiento especializado sobre las tareas que la computadora debería de ejecutar, lo cual no está disponible en un sistema operativo de propósito general como Linux. Sin embargo, el multi–procesamiento simétrico es relativamente fácil de implementar.

Por relativamente fácil, quiero decir exactamente — que no es *realmente* fácil. En un entorno de multi–procesamiento simétrico, las CPUs comparten la misma memoria, y como resultado, el código ejecutándose en una CPU puede afectar a la memoria usada por otro. Ya no puedes estar seguro de que una variable que has establecido a un cierto valor en la linea anterior todavía tenga el mismo valor — la otra CPU quizás haya jugado con ella mientras nosotros estábamos mirando. Obviamente, es imposible programar algo como esto.

En el caso de la programación de procesos esto no suele ser un problema, porque un proceso normalmente sólo se ejecutará en una CPU a la vez[1]. El núcleo, sin embargo, podría ser llamado por diferentes procesos ejecutándose en CPUs diferentes.

En la versión 2.0.x, esto no es un problema porque el núcleo entero está en un gran spinlock. Esto significa que si una CPU está en el núcleo y otra CPU quiere entrar en el, por ejemplo por una llamada al sistema, tiene que esperar hasta que la primera CPU haya acabado. Esto hace que Linux SMP sea seguro[2], pero terriblemente ineficiente.

En la versión 2.2.x, varias CPUs pueden estar en el núcleo al mismo tiempo. Esto es algo que los escritores de módulos tienen que tener en cuenta. Espero que alguien me de acceso a un equipo SMP, por lo tanto espero que la siguiente versión de este libro incluya más información.

[1] La excepción son los procesos con hilos, que pueden ejecutarse en varias CPUs a la vez.

[2] Queriendo decir que es seguro para usar con SMP

Capítulo 13

Problemas comunes

Antes de enviarte al mundo exterior y escribir módulos del núcleo, hay unas pocas cosas sobre las que te tengo que avisar. Si me equivoco al avisarte y algo malo sucede, por favor envíame el problema para una total devolución de la cantidad que te tengo que pagar por tu copia del libro.

1. **Usando bibliotecas estándar.** No puedes hacer esto. En un módulo del núcleo sólo puedes usar las funciones del núcleo, que son las funciones que puedes ver en `/proc/ksyms`.

2. **Deshabilitando las interrupciones.** Quizás necesites hacer esto durante un rato y es correcto, pero si no las habilitas posteriormente, tu sistema estará muerto y tendrás que apagarlo.

3. **Meter tu cabeza dentro de un gran carnívoro.** Probablemente no tenga que avisarte sobre esto.

Apéndice A

Cambios entre 2.0 y 2.2

No conozco todo el núcleo suficientemente para documentar todos los cambios. En el transcurso de la conversión de los ejemplos (o actualmente, adaptando los cambios de Emmanuel Papirakis) me encontré con las siguientes diferencias. Las listo aquí para ayudar a los programadores de módulos, especialmente aquellos que aprendieron de versiones previas de este libro y que están más familiarizados con las técnicas que utilizo, y para convertirlos a la nueva versión.

Un recurso adicional para la gente que quiera convertirse a 2.2 está en `http://www.atnf.csiro.au/~rgooch/linux/docs/porting-to-2.2.html`.

1. **asm/uaccess.h** Si necesitas `put_user` o `get_user` tienes que incluir sus ficheros de cabeceras (#include)

2. **get_user** En la versión 2.2, `get_user` recibe el puntero a la memoria de usuario y la variable en la memoria del núcleo para rellenarla con la información. El motivo por el que esto es así es que `get_user` ahora puede leer dos o cuatro bytes al mismo tiempo si la variable que leemos es de una longitud de dos o cuatro bytes.

3. **file_operations** Esta estructura tiene una función de borrado entre las funciones `open` y `close`.

4. **close en file_operations** En la versión 2.2, la función close retorna un entero, por lo tanto se permite que falle.

5. **read y write en file_operations** Las cabeceras de estas funciones cambiaron. Ahora retornan `ssize_t` en vez de un entero, y su lista de parámetros es diferente. El inodo ya no es un parámetro, y en cambio está el desplazamiento en el fichero.

6. **proc_register_dynamic** Esta función ya no existe. En vez de ello, llamas a `proc_register` normalmente y pones cero en el campo inodo de la estructura.

7. **Señales** Las señales en la estructura de tareas ya no son un entero de 32 bits, sino un array de enteros `_NSIG_WORDS`.

8. **queue_task_irq** Incluso si quieres planificar una tarea para que suceda dentro de un manejador de interrupciones, usa `queue_task`, no `queue_task_irq`.

9. **Parámetros del Módulo** No tienes que declarar los parámetros del módulo como variables globales. En 2.2 también puedes usar `MODULE_PARM` para declarar su tipo. Esto es un gran avance, porque permite que el módulo reciba parámetros de cadenas de caracteres que empiezan con un dígito, por ejemplo, sin ser confuso.

10. **Multi–Procesamiento Simétrico** El núcleo ya no está dentro de un gran spinlock, lo que significa que los módulos del núcleo tienen que tener en cuenta el SMP.

Apéndice B

¿Desde aquí hasta dónde?

Yo, fácilmente podría haber introducido unos pocos capítulos más en este libro. Podría haber añadido un capítulo sobre cómo crear nuevos sistemas de ficheros, o sobre como añadir nuevas pilas de protocolos (como si hubiera necesidad de esto — tendrías que excavar bajo tierra para encontrar una pila de protocolos no soportados por Linux). Podría haber añadido explicaciones sobre mecanismos del núcleo que no hemos tocado hasta ahora, tales como el arranque o la interfaz de discos.

De cualquier forma, he escogido no hacerlo. Mi propósito al escribir este libro era suministrar una inicialización en los misterios de la programación de módulos del núcleo y enseñar técnicas comunes para este propósito. Para gente seriamente interesada en la programación del núcleo, recomiendo las lista de recursos del núcleo en `http://jungla.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html`. También, como dijo Linus, la mejor forma de aprender el núcleo es leer tú mismo el código fuente.

Si estás interesado en más ejemplos de módulos cortos del núcleo, te recomiendo la revista Phrack. Incluso si no estás interesado en seguridad, y como programador deberías de estarlo, los módulos del núcleo son buenos ejemplos de lo que puedes hacer dentro del núcleo, y son suficientemente pequeños para no requerir mucho esfuerzo para entenderlos.

Espero haberte ayudado en tu misión de convertirte en un mejor programador, o al menos divertirte a través de la tecnología. Y, si escribes módulos del núcleo útiles, espero que los publiques bajo la GPL, para que yo también pueda utilizarlos.

Apéndice C

Beneficios y Servicios

Espero que nadie piense en descaradas promociones aquí. Son todas cosas que son buenas en el uso para los programadores principiantes de módulos del núcleo Linux.

C.1 Obteniendo este libro impreso

El grupo Coriolis va a imprimir este libro varias veces en el verano del 99. Si ya es este verano, y quieres este libro impreso, puedes fácilmente ir a tu impresor y comprarlo de una forma agradablemente encuadernada.

Apéndice D

Mostrando tu gratitud

Este es un documento libre. No tienes obligaciones posteriores a aquellas dadas en la GNU Public License (Appendix E). En todo caso, si quieres hacer algo como recompensa por la obtención de este libro, aquí hay algunas cosas que puedes hacer.

- Envíame una tarjeta postal a
  ```
  Ori Pomerantz
  Apt.  #1032
  2355 N Hwy 360
  Grand Prairie
  TX 75050
  USA
  ```
  Si quieres recibir un gracias de mi parte, incluye tu dirección de correo electrónico.
- Contribuye con dinero, o mejor todavía, con tiempo, a la comunidad de software libre. Escribe un programa o un documento y publícalo bajo la GPL. Enseña a otra gente cómo usar el software libre, tal como Linux o Perl.
- Explica a la gente como el ser egoistas *no* es incompatible con el vivir en sociedad o ayudando a otra gente. Yo he disfrutado escribiendo este documento, y creo que publicándolo me ayudará en el futuro. Al mismo tiempo, he escrito un libro el cual, si has llegado tan lejos, te ayudará. Recuerda que la gente feliz es habitualmente más útil que aquella gente que no lo es, y hace que la gente sea *mejor* con la gente con menos habilidades.
- **Sé feliz**. Si consigo conocerte, será el mejor encuentro para mi, te hará más útil para mi ;-).

Apéndice E

La GNU General Public License

Lo que está impreso a continuación es la GNU General Public License (la *GPL* o *copyleft*), bajo la cual está licenciado este libro.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The 'Program', below, refers to any such program or work, and a 'work based on the Program' means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term 'modification'.) Each licensee is addressed as 'you'.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and ‘any later version’, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM ‘AS IS’ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

APPENDIX: HOW TO APPLY THESE TERMS TO YOUR NEW PROGRAMS

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the ‘copyright’ line and a pointer to where the full notice is found.

> *one line to give the program’s name and a brief idea of what it does.* Copyright ©19yy *name of author*
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
 Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type show w.  This is free software, and you are welcome to
redistribute it under certain conditions; type show c for
details.
```

The hypothetical commands show w and show c should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than show w and show c; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a ‘copyright disclaimer’ for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program Gnomovision (which makes passes at compilers) written by James Hacker.
>
> *signature of Ty Coon*, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Apéndice F

Sobre la traducción

Este documento es la traducción de ‘Linux Kernel Module Programing Guide 1.1.0’ y el proceso de traducción ha sido llevado a cabo por:

- Traductor: Rubén Melcón Fariña <melkon@terra.es>
- Revisor: Óscar Sanz Lorenzo <gaaldornik@terra.es>
- Encargado de Calidad: Francisco Javier Fernández <serrador@arrakis.es>

Documento publicado por el proyecto de documentación de Linux (`http://www.es.tldp.org`).

Número de revisión: 0.12 (Septiembre de 2002)

Si tienes comentarios y/o sugerencias sobre la traducción, ponte en contacto con Rubén Melcón <melkon@terra.es>

Índice de Materias