Contenido

Índice

1. Introducción

Qué es

Así que queréis escribir módulos del kernel. Sabéis C, habéis escrito algunos programas que corren como procesos y ahora queréis acción de verdad donde un simple puntero loco te puede destrozar el sistema de ficheros y un segmentation fault significa reiniciar. ¿Pero qué leches es exactamente un módulo del kernel?

¿Qué es un módulo del Kernel?

- Trozo de código que se carga en el núcleo bajo demanda.
- Amplian sus capacidades sin necesidad de reiniciar.
- Un driver es un modulo del kernel, por ejemplo ETTY.

Sin módulos tendríamos que construir núcleos monolíticos y meter las nuevas funcionalidades directamente en la imagen. Eso haría tener núcleos muy grandes y la necesidad de rehacer el núcleo entero y reiniciar la máquina cada vez que queremos una nueva funcionalidad.

El ejemplo que se verá en esta presentación, ETTY (emulated TTY) surgió por la necesidad de poder tener puertos serie dentro de una máquina virtual. Qemu te limita a 4, yo necesitaba al menos 8.

Por qué

¿Por qué escribir módulos?

- Por diversión.
- Para ganar dinero
- Para aprender cómo funcionan los entresijos.
- Por que puedes.

El desarrollo del kernel está avanzando mucho últimamente, sobre todo en sistemas embebidos. Cada vez hay mas hardware al que hay que dar soporte. Además puede surgir la necesidad, no es extraño de querer tener un módulo de kernel que está para una versión moderna en una más antigua. Se podría pensar que bastaría con recompilarlo pero el kernel de Linux es ente que cambia constantemente y se reescriben partes de código tan importantes como la pila TCP desde cero prácticamente en cada versión. Además en la rama 2.6 ha habido varios cambios en las APIs. El cambio más grande fue entre las versiones 2.6.17 y 2.6.18. Llevarse módulos de versiones superiores a la 18 a una inferior a la 17 o viceversa puede ser doloroso sino se conocen los entresijos de cómo funcionan los módulos.

¿Vale pero por qué de Linux?

- Porque es libre y el código fuente está disponible
- Por la inmensa comunidad que tiene alrededor
- ¿He dicho ya que es divertido?

Además es el único sistema operativo para el que he aprendido a hacerlos así que pocas opciones tenía. Y no hay que olvidar que esto es el Grupo de Usuarios de Linux, así que dudo que me hubieran dejado dar una charla para hacer módulos para Windows, si es que tiene algo parecido.

Cómo

¿Cómo se meten los módulos en el kernel?

- ¿Qué modulos estoy usando ahora mismo? `lsmod`
- ¿Cómo cargo uno? `modprobe`
- ¿Y para descargarlo? `rmmod`

Para saber los módulos que hay cargados en este momento se usa la orden `lsmod`, que lee el fichero `/proc/modules`, los trata y lo muestra de manera más legible. Lo que muestra es tanto el nombre del módulo cargado como su tamaño y el número de procesos u otros modulos que lo están usando.

La herramienta `modprobe` recibe por parámetro el nombre del núcleo que se quiere usar y lo carga junto con sus dependencias.

¿Y cómo sabe dónde ir a buscar?

Acude al fichero `/lib/modules/$(uname -r)/modules.dep` que es quien define qué módulos hay y sus dependencias.

¿Y cómo genero ese fichero?

Con la orden `depmod -a`.

También se puede usar la orden `insmod` que recibe por parámetro directamente el fichero del módulo y lo carga.

Para descargar un módulo se usa la orden `rmmod` y el nombre del módulo a descargar. Es importante recalcar que no se puede descargar un módulo que esté siendo usado. Esto no es trivial ya que sino tenemos cuidado podemos hacer creer al kernel que ese módulo está siendo usado y no nos dejará descargarlo y eso implica reiniciar, lo veremos más adelante.

2. Módulo de ejemplo

Hola Mundo

Hola Mundo

```
#include <linux/module.h>
#include <linux/kernel.h>
int init_module(void){
  printk(KERN_INFO 'Hola Mundo.\n');
  return 0;
}
void cleanup_module(void){
  printk(KERN_INFO 'Adios mundo cruel.\n');
}
```

Este código es el módulo más simple que hay. Se incluye `module.h` ya que es esencial en todos los módulos. También kernel.h para tener acceso a algunas constantes.

La función `init_module` es la que se ejecuta cuando se carga el módulo y tiene que hacer todas las acciones necesarias para que se pueda usar. En caso de que haya algún problema iniciando el módulo, se retorna con un estado distinto de cero para que se pueda informar al usuario.

La función `cleanup_module` es la opuesta a la anterior. Se ejecuta al descargar el módulo. Es *importantisimo* que esta función deshaga todas las acciones que se hicieron en `init_module` y que deje el sistema tal y como lo encontró.

Como se observa en el código, no se usa la función `printf`. Básicamente porque como estamos programando para el kernel no tenemos `libc` ni podemos usar bibliotecas. Aunque lo parezca, `printk` no está pensada para comunicarse con el usuario. Es un sistema de log y alarmas para el kernel. Cada llamada a `printk` comienza con la prioridad que se le quiere dar al mensaje. Hay 8 prioridades y el kernel tiene macros para cada una (por eso incluimos `kernel.h`). Si no se selecciona ninguna de ellas se usará `DEFAULT_MESSAGE_LOGLEVEL`.

Si la prioridad es menor que `int console_loglevel` el mensaje se mostrará en la consola además de ser añadido a `/var/log/messages`. Todos los mensajes se añaden a `/var/log/messages` se muestren o no.

Cómo se compila

Cómo se compila

```
obj-m += holamundo.o

all:
 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
 make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Los módulos del kernel se compilan de manera distinta a las aplicaciones de usuario. Con los kernel antiguos había que tener mucho cuidado con los parámetros que usualmente se guardaban en `Makefiles`. Al final esto era un lío de narices. Al final y por suerte hay una nueva forma de hacer las cosas llamada `kbuild`. El proceso de creación de módulos externos está ahora integrado en el mecanismo de compilación del kernel.

La información completa sobre cómo compilar módulos externos se puede encontrar en `linux/Documentation/kbuild/modules.txt`.

Por mi parte voy a usar siempre el `Makefile` expuesto arriba.

Ver ejemplo 1

Cómo se le pasan argumentos

Cómo se le pasan argumentos

- `argc`/`argv` han pasado a mejor vida
- `module_param`(nombre,tipo,permisos)
  - Nombre de la variable
  - Tipo de variable

    bool/invbool un bit verdadero o falso.
    charp cadena
    int entero
    long entero largo
    short entero corto
    uint... sin signo
  - Permisos `sysfs` (ver `man 2 open`)

```
insmod mimodulo.ko miparametro=5
```

Los módulos pueden tomar argumentos pero no de la forma en que estamos acostumbrados. Para poder pasarle argumentos a nuestros módulos hay que declarar las variables como globales y entonces usar la función `module_param` para que el mecanismo haga su magia. En tiempo de ejecución, insmod rellenará la variable con lo que se le indique. Es recomendable poner estas declaraciones al principio del código para que se lea mejor.

Ver ejemplo 2

3. Cómo funcionan los módulos

Cómo se ejecutan los módulos

- No hay `main()`
- No tienen un flujo determinado.

Los programas de usuario normalmente comienzan con una función `main()`, se ejecutan un montón de instrucciones y termina cuando ese conjunto de instrucciones se acaba.

Los módulos de kernel trabajan de forma un poco distinta. Comienzan con la función `init_module` o la función que se indique en con la macro `module_init`. Esta función le dice al kernel la funcionalidad que proporciona y lo configura para que se ejecuten las funciones del módulo cuando sea necesario. Una vez hecho esto el módulo no hace nada hasta que el kernel quiera hacer algo con el código que ofrece el módulo.

Funciones disponibles

- No hay `printf`, `malloc`, `strcmp`, `atoi`...
- No hay `libc`!!!!
- ¿Entonces me tengo que picar toooodo lo que haga?Sí.Bueno, casi todo.
- Se pueden usar todas las funciones que exporta el kernel.
- `/proc/kallsyms`

Normalmente los programadores usan funcione que no implementan el típico ejemplo es `printf`. Se suele usar la biblioteca estándar de C `libc`. El código de estas funciones no entra en el programa hasta la fase de enlazado donde el sistema se asegura que el código para, por ejemplo, `printf` está disponible. Entonces se fija la llamada a la función por un puntero a este código.

En este sentido los módulos de kernel son también distintos. En nuestro módulo de ejemplo se uso la función `printk` pero no se incluyó ninguna biblioteca de entrada/salida. Esto es así porque los módulos son objetos cuyos simbolos se resuelven cuando se insertan (`insmod`). Las únicas funciones que se pueden usar son las que exporta el kernel. La lista la puedes comprobar en el fichero `/proc/kallsyms`.

Memoria dinámica en el kernel

- `kmalloc`
  - `#include <linux/slab.h>`
  - `void *kmalloc(size_t size, int flags)`
  - flags: `GFP_KERNEL`
  - flags: `GFP_ATOMIC`
  - flags: ...
- `kfree(void *ptr)`

A veces en un módulo de kernel puede ser necesario gestionar dinámicamente alguna zona de memoria. En el kernel, como no tenemos `libc` no podemos usar las llamadas `malloc`. Por suerte existe una parecida.

Se llama kmalloc.

Su sintaxis es la siguiente. Devuelve el puntero pedido; requiere por parámetro el tamaño deseado y unos `flags`.

Estos flags pueden ser: `GFP_KERNEL` significa que el kernel está pidiendo un espacio de memoria en nombre de un proceso de usuario. Esto significa que el proceso se pone a dormir hasta que la memoria es obtenida.

A veces se requiere llamar a `kmalloc` fuera del contexto de un proceso, como en las interrupciones, temporizadores, ...En estos casos es necesario que el proceso no se duerma esperando la memoria. Así que se debe usar `GFP_ATOMIC`. El kernel cuando se inicia prepara unas cuantas páginas para ser devueltas con esta orden. Si este conjunto de páginas se termina, la llamada falla.

Hay más flags, pero estos son los más importantes y básicamente los únicos que se usan.

`kmalloc` no es la forma más eficiente de obtener mucha memoria, pero sí es de las más sencillas. Más información en la bibliografía.

La memoria pedida con `kmalloc` se tiene que liberar con `kfree`. No se debe pasar a `kfree` una dirección que no se solicitó con `kmalloc`. Pero sí se le puede pasar `NULL`.

Espacio de Kernel vs Espacio de Usuario

- Objetivo del kernel:gestión de los recursos.
  - Tarjeta de video
  - Disco duro
  - Memoria
  - ...
- Competición por los recursos
- Modos de ejecución

    - Modo supervisor
    - Modo usuario

Hay que diferenciar entre funciones de biblioteca y llamadas a sistema. Las funciones de biblioteca son de alto nivel, corre completamente en espacio de usuario y proporcionan una interfaz más sencilla para la funcionalidad que hace el trabajo real (llamadas al sistema).

Las llamadas al sistema se ejecutan en modo kernel en el entorno del usuario y las proporciona el núcleo mismo. Las llamadas a bibioteca como `printf` puede paracer una llamada muy genérica para pintar pero en realidad lo que hace es formatear los datos en cadenas y pintar la cadena usando la llamada al sistema de bajo nivel `write`; que es quien envía los datos a la salida estándar.

Si se desea ver qué llamadas al sistema hace un programa, se puede usar `strace`. Que recibe como parámetro el programa a ejecutar. `strace` muestra por la salida de error todas las llamadas al sistema que hace un ejecutable.

El objetivo del kernel es la gestión de los recursos, estos van desde una tarjeta de vídeo, pasando por un disco duro, o incluso la memoria. Normalmente los programas compiten por el mismo recurso, por ejemplo mi sesión de `vim` con la que he creado esta presentación y el programa `amarok` con el que estoy escuchando música están continuamente peleando por acceder al disco duro a la vez. El núcleo es quien se encarga de mantener las cosas en orden y de no dar acceso a los recursos cuando a los programas se les antoje. Para ello el procesador puede funcionar en diferentes modos; cada uno da un grado de libertad en cuanto a lo que se puede hacer en el sistema.

Los procesadores con arquitectura Intel80386 tienen 4 de estos modos, que se llaman anillos (`rings`). Los sistemas Unix sólo usan dos de estos anillos. El anillo más alto, el 0, también conocido como modo supervisor, donde puede pasar de todo; y el más bajo, llamado modo usuario, el cual está bastante limitado.

Volviendo al tema de las llamadas a biblioteca y las llamadas al sistema. Normalmente uno hace una llamada a bibioteca en modo usuario. La función hace una o varias llamadas al sistema; esas llamadas al sistema se ejecutan en el contexto de la función de la biblioteca. Pero lo hacen en modo supervisor ya que son parte el kernel en sí mismo. Una vez que la llamada al sistema termina su tarea, retorna y la ejecución vuelve al modo usuario.

Espacio de nombres

- Variables locales
- Variables globales
    - Corrupción del espacio de nombres (*namespace pollution*)
- El kernel es un único espacio de nombres
- `/proc/kallsyms`

Cuando se escribe se usan variables con nombre legibles para el programador. Si, por otro lado, estamos escribiendo rutinas que son parte de un problema más grande, cualquier variable global que declaremos estará junto a las de otros y es posible que algunos nombres choquen. Cuando un programa tiene un montón de variables globales que no son suficientemente distinguibles se consigue una corrupción del espacio de nombre (*namespace pollution*). En proyectos grandes hay que buscar formas de desarrollar un esquema para el nombrado correcto de variables y símbolos.

Cuando se escribe código del kernel incluso el método más pequeño se enlaza con el núcleo entero. Así que esto es un problema. La mejor manera de lidiar con esto es declarar las variables como estáticas y usar un prefijo bien definido para los símbolos. Por convención los prefijos del kernel van en minúsculas.

En el fichero `/proc/kallsyms` están todos los símbolos que el kernel conoce y que por ende son accesibles por nuestros módulos ya que comparten el mismo espacio de código del kernel.

Espacio de código

- Gestión de memoria uffff
- Violación de Segmento
- Punteros = Direcciones de memoria físicas?NO
- Punteros = Direcciones de memoria lógicas.
- Espacio de código del kernel = módulos
- *Segfault* del módulo $\Rightarrow$ *Segfault* del kernel

La gestión de la memoria es bastante complicado, casi todo el libro *Understanding The Linux Kernel* trata sobre la gestión de la memoria. Esto queda fuera del ámbito de esta presentación pero es necesario saber algunas cosas.

Si no habéis pensado nunca lo que significa realmente violación de segmento. Te sorprenderá saber lo siguiente.

Normalmente pensamos en las direcciones de memoria de nuestros programas como direcciones físicas de memoria, pero eso no es cierto.

Las direcciones que usamos en nuestros programas son direcciones lógicas. Cuando se crea un proceso se reserva una porción de memoria física para él. Esta memoria comienza en la dirección 0 y se extiende hasta donde se necesite. Como los espacios de memoria de dos procesos no se solapan, ambos pueden acceder a la direccion $0xbfffff978$ pero estarán accediendo a direcciones físicas diferentes. Existen formas para desde un programa acceder al espacio de otro pero eso es otra historia.

El núcleo tiene su propio espacio de memoria también. Como un módulo es un trozo de código que se inserta y elimina dinámicamente en el kernel, éstos comparten el mismo espacio de código.

Esto implica que si tenemos una violación de segmento en un módulo, tenemos una violación de segmento en el núcleo. Y si empezamos a escribir sobre zonas de memoria por ahí perdidas entonces estás pisoteando el código del kernel.

Y tienes un problema, esto es peor de lo que imaginas así que ya puedes tener cuidado.

4. Drivers de dispositivos

Los nodos

Nodos de dispositivo

- Qué son? Ficheros
- *major*
  - `/proc/devices`
- *minor*
- Tipos de dispositivos
  - Caracteres
  - Bloques
- Para crear uno
  - mknod <nombre> bc<b|c> *major minor*
- `/usr/src/linux/Documentation/devices.txt`

Un tipo de módulo es el driver de dispositivo. Un driver de dispositivo proporciona la funcionalidad de un hardware como puede ser una tarjeta de televisión o un puerto serie. En los sistemas Unix cada pieza de hardware está representada por un fichero en `/dev` que es el punto a través del cual nos comunicamos con el hardware. Por ejemplo driver de la tajeta de sonido `es1370.o` conecta `/dev/sound` con la tarjeta de sonido Ensoniq IS 1370. Un programa de usuario puede usar `/dev/sound` sin ni siquiera saber qué tarjeta de sonido está instalada.

Los nodos, como he dicho, son ficheros. Si se listan las los nodos de las 3 primeras particiones de un disco se puede ver algo como:

```
brw-rw---- 1 root disk 8, 1 feb 24 07:19 /dev/sda1
brw-rw---- 1 root disk 8, 2 feb 24 07:19 /dev/sda2
brw-rw---- 1 root disk 8, 3 feb 24 07:19 /dev/sda3
```

El número que está antes de la coma se denomina `major`. El `major` te dice qué driver tienes que usar para manejar ese dispositivo.

Para vér a qué driver se corresponde hay que ir al fichero `/proc/devices`. En este caso habrá una línea así:

```
8 sd
```

Esto indica que se usará el driver sd.

Como se puede observar, los 3 nodos usan el mismo driver, ¿cómo sabe el sistema a qué dispositivo me estoy refiriendo? Para esto está el número de después de la coma, que se le conoce como `minor`. El `minor` es usado por el driver para distinguir entre el hardware que controla. Siguiendo con el ejemplo, cada partición tiene un `minor` distinto para poder hacer referencia a cada partición independientemente.

Hay dos tipos de dispositivos, de caracteres y de bloques. La diferencia es que los de bloques tienen un buffer para peticiones, de tal manera que las puedan ordenar para responderlas mejor y más rápido. Esto es importante en el caso de los discos duros, por ejemplo, donde es más rápido leer y escribir sectores que esté cercanos entre sí que aquellos que están muy lejos. Otra diferencia es que los dispositivos de bloques sólo reciben y envían bloques (el tamaño depende del dispositivo), mientras que con un dispositivo de caracteres se trabaja con desde 1 hasta tantos bytes como se desee. La mayoría de dispositivos son de caracteres ya que no necesitan de ese buffer y no trabajan con tamaños de datos fijos.

Se puede saber el tipo del dispositivo por la primera letra que aparece en el ejemplo. Una `b` para los dispositivos de bloques y una `c` para los de caracteres.

Para crear un nodo se usa la orden `mknod <nombre> <b|c> <major> <minor>` y para saber la lista de dispositivos ya asginados se puede consultar:
`/usr/src/linux/Documentation/devices.txt`.

Dispositivos de caracteres

Estructura `file_operations`

```
struct file_operations {
 struct module *owner;
 loff_t (*llseek) (struct file *,loff_t, int);
 ssize_t (*read) (struct file *,char __user *,size_t,loff_t *);
 ssize_t (*write) (struct file *,const char __user *,size_t, loff_t *);
 int (*readdir) (struct file *,void *,filldir_t);
 unsigned int (*poll) (struct file *,struct poll_table_struct *);
 int (*ioctl) (struct inode *,struct file *,unsigned int,unsigned long);
 int (*open) (struct inode *,struct file *);
 int (*flush) (struct file *,fl_owner_t id);
 int (*fsync) (struct file *,struct dentry *,int datasync);
};
```

La estructura `file_operations` definida en `linux/fs.h` guarda los punteros a las funciones definidas por el driver. Esta es la forma que se usa para hacer "polimorfismo" en C. Aquí se están mostrando un subconjunto de las funciones existentes. Algunas operaciones no son implementadas por un driver. Por ejemplo el driver de una controladora de vídeo no necesita implementar la función `readdir`. Las funciones no implementadas se rellenan con `NULL`.

Uso de `file_operations`

```
struct file_operations fops = {
  .owner = THIS_MODULE,
  .read = midispositivo_read,
  .write = midispositivo_write,
  .open = midispositivo_open,
```

```
  .release = midispositivo_release
};
```

En las versiones modernas de `gcc` la declaración de la variable y su relleno se puede hacer de la forma que indica la diapositiva. Los campos que no se indiquen serán rellenador con `NULL` por `gcc`.

Otras estructuras importantes

- Estructura `file`
- Estructura `cdev`

Cada dispositivo en el kernel está representado por una estructura `file`. Que está definida en `linux/fs.h`. Esta estructura nunca llega al usuario y no es la misma que la `FILE` de `glibc`. La cual nunca llega al kernel. Su nombre puede confundir. Esta estructura representa un fichero abstracto abierto, no un fichero de un disco. Los ficheros de disco se representan por la estructura llamada `inode`. No entro a fondo a ver esta esrtuctura porque los drivers no la rellenan directamente sino que se usan las estructuras que contiene `file` y que son creadas en otros sitios.

Con `cdev` ocurre exactamente lo mismo pero representa un dispositivo de caracteres.

Registrar un dispositivo

- `cdev_alloc`
- `cdev_add`
  - `int cdev_add(struct cdev *dev, dev_t num, unsigned int count);`
- `cdev_del`

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->owner = THIS_MODULE;
my_cdev->ops = &my_fops;
ret=cdev_add(my_cdev, my_major, my_count);
...
cdev_del(my_cdev);
```

Para registrar un dispositivo en el kernel, primero es necesario inicializar algunas estructuras. La primera de todas es la estructura `cdev` que representa al dispositivo. Se crea con la función `cdev_alloc`.

Una vez tenemos la estructura `cdev` como deseamos pasamos a registrarla en el kernel, para ello se usa la llamada `cdev_add`.Que tiene la siguiente forma: `int cdev_add(struct cdev *dev, dev_t num, unsigned int count);`. donde `dev_t` es el `major` con el que queremos registrar el dispositivo, y `count` es el número de dispositivos a los que estará escuchando(los `minor`).

Cuando hemos terminado de usar el dispositivo y queremos desregistrarlo, llamamos a `cdev_del`.

Una secuencia de uso puede ser como la siguiente. En ella una vez inicializada la estructura `cdev` se asocian las funciones que deseamos que se utilices al su campo correspondiente en la estructura, además de decir que el dueño de este dispositivo es este mismo módulo, y pasamos a registrarlo.

Una vez se ha terminado de usar, se desregistra.

Es importante hacer notar que una vez que la llamada `cdev_add` ha terminado exitosamente el dispositivo está "corriendo" y sus operaciones pueden ser usadas por el kernel, así que para cuando llames a esta función asegúrate de que está todo listo.

Ver ejemplo 3

5. Comunicación espacio kernel ⇔ espacio usuario

Usuario a Kernel

Pasar datos del usuario al kernel

- `ssize_t (*write) (struct file *,const char __user*,size_t, loff_t *);`
- `unsigned long copy_from_user(void *to,const void __user *from,unsigned long count);`

Anteriormente en la llama write había algo que no he explicado, y es la palabra `__user`. Esto significa que ese puntero está en espacio de usuario. Cuando un programa realiza la llamada al sistema write, el puntero del contenido que pasa es del espacio de memoria del programa. Como he dicho anteriormente el kernel tiene su propio espacio de memoria, así que para no liarla lo que hay que hacer es copiar el contenido del espacio de usuario al del kernel. Si la liamos, lo menos que puede pasar es que tengamos una violación del segmento del espacio de usuario y se genere un "`oops`" que resulta en la muerte del proceso que nos ha llamado. Además como ese puntero nos lo ha pasado el usuario no podemos fiarnos de lo que contiene, ya que puede contener una dirección maligna creada a propósito para modificar una zona que no le corresponde.

Esto se realiza con la función `copy_from_user`. Que devuelve los bytes copiados y recibe los parámetros como un `memcpy`. Y que hace comprobaciones para asegurarse que el acceso es legal.

Es importante saber que estas llamadas pueden hacer que el proceso se ponga a dormir porque ese espacio de memoria está en `swap` o por cualquier otra razón. Así que la función debe poder ejecutarse concurrentemente con otras funciones del driver

Kernel a Usuario

Pasar datos del kernel al usuario

- `ssize_t (*read) (struct file *,char __user*,size_t,loff_t *);`
- `unsigned long copy_to_user(void __user *to,const void *from,unsigned long count);`

El paso de datos del espacio de kernel al de usuario es igual e inverso al anterior.

Ver ejemplo 4

6. Sistema de ficheros `/proc`

Creación y uso

Cómo crear un fichero en `/proc`

- `<linux/proc_fs.h>`
- `int (*read_proc)(char *page, char **start, off_t offset, int count,int *eof, void *data);`
- `struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode, struct proc_dir_entry *base, read_proc_t *read_proc, void *data);`
- `remove_proc_entry`

Para crear una entrada en `/proc` voy a poner como ejemplo un pequeño fichero sólo para leer. Digo pequeño porque si la salida ocupa más de una página de memoria la cosa se complica bastante. Pero como `/proc` se suele usar para dar algunos datos de estado del driver, no suele hacer falta tanta indormación.

Lo que hay que crear es el método que devolverá los datos a mostrar. Los datos se guardan en `page` y el resto de parámetros se usan para cuando la salida requiere de varias páginas de memoria, así que no los explicaré.

Una vez tenemos la función hay que crear el fichero. Esto se hace con `create_proc_read_entry`. Que recibe por parámetro la ruta del fichero, el modo, la base (que puede ser `NULL`), la función `read` que hemos implementado y unos datos que suelen ser `NULL` también.

El uso de `/proc` está ya desaconsejado, ahora se suele usar el sistema `sysfs`, normalmente montado en `/sys`. Pero esto todavía no lo he usado así que no lo puedo explicar.

Ver ejemplo 5

7. Cómo hablar con dispositivos

La forma normal

`Read` y `Write`

- Dispositivos lógicos ⇔ Dispositivos físicos
- Para leer: `read` ó `midispositivo_read`
- Para escribir: `write` ó `midispositivo_write`
- ¿Y si sólo quiero hablar con el dispositivo lógico?
  - Configurar baudios, paridad, bloquear una puerta, autodestrucción, . . .

Como ya he dicho, y si no lo digo ahora, la idea es que los nodos de los dispositivos sean lo más parecido a los dipositivos físicos reales. Además la idea es que usarlos sea lo más sencillo posible. Por ello para hablar con un dispositivo se usan las funciones `read` y `write` como si fuera un fichero.

Pero esto tiene una limitación para cuando por ejemplo en un puerto serie, no se quieren enviar datos, sino cambiar la velocidad de transferencia, la paridad...

Para poder hacer estas acciones existe lo siguiente.

La forma avanzada

Ioctl

- IOCTL:Input Output ConTroL
- Usuario: `int ioctl(int fd, unsigned long cmd, ...????);`
- Cada `ioctl` es distinta, desestructurado, no documentado, no portable, ...
- En resumen: no usar
- Mejor otros métodos; sobre todo `sysfs`

La llamada `ioctl` viene de `Input Output ConTroL`.

La forma en que la llama el usuario es como se muestra. Con el descriptor del fichero sobre el que se quiere realizar, la orden, que va codificada como un número y ...

¿Puntos suspensivos? ¿Eso quiere decir que puede recibir una lista variable de argumentos? No. Las funciones del núcleo tienen que estar bien definidas (well-defined) porque los programas de usuario sólo pueden llegar a ellas a través de ciertas puertas. Así que los puntos suspensivos no significan argumentos variables sino un único argumento opcional normalmente definido como `char *argp`.

La desestructura natural de esta llamada, la poca o nula documentación, la poca portabilidad, ... hacen que no sea recomendable el uso de esta función. Se recomiendan otras formas para realizar las mismas funciones. Ahora mismo la más recomendable es el uso de `sysfs`.

Ver ejemplo 6

8. Una TTY

Qué es

¿Qué es una TTY?

- Originalmente
  - TTY: *Teletypewriter* teletipo
  - Conexión física o virtual

- Actualmente
  - Cualquier puerto serie
  - Terminales
  - USB-to-serial
  - Modems
  - Bluetooth

Las `tty` toman su nombre de los antiguos teletipos. Una cinta de papel de una línea escrita. Con la llegadas de los ordenadores eso pasó a ser una cinta perforada y luego un cable. De tal forma que una `tty` era una conexión física o virtual con la máquina.

Actualmente una `tty` es cualquier puerto serie, los terminales y pseudoterminales, los drivers USB-to-serial, los modems o las conexiones Bluetooth.

El núcleo del driver `tty` vive justo debajo del de los dispositivo de caracteres y proporciona una serie de características enfocadas en dar una interfaz para los dispositivos de estilo terminal. El núcleo de `tty` se encarga se controlar el flujo de datos y formatearlos. Eso permite a los drivers `tty` centrarse en los datos desde y hasta el *hardware* en vez de preocuparse de la interacción con el usuario.

Para ver qué drivers `tty` se estan usando se puede consultar el fichero `/proc/tty/drivers`.

Cómo se crea

Cómo se hace un driver `tty`

- `<linux/tty_driver.h>`
- `struct tty_driver`
- `alloc_tty_driver(tty_minors)`
- `struct tty_operations`
- `tty_set_operations`
- `tty_registrer_driver`
- `tty_registrer_device`

Para hacer un driver de `tty` hay que incluir el `.h` correspondiende. Ahora un rápido vistazo sobre estructuras y funciones básicas.

La estructura `tty_driver` es la que contiene la información para ser registrada en el kernel.

Se crea con la función `alloc_tty_driver(tty_minors)`. El parámetro indica el número de *minors* donde escuchará.

La estructura `struct tty_operations` se parece mucho a la `file_operations`. Su significado es el mismo pero cambian algunos campos.

La función `tty_set_operations` enlaza una estructura `struct tty_driver` con una `struct tty_operations`.

Por último la función `tty_registrer_driver` registra nuestro driver inicilizado en el núcleo. Y una vez registrado él mismo debe registrar los dispositivos que controla con la función `tty_registrer_device`.

Ejemplo de `tty_operations`

```
static struct tty_operations serial_ops = {
  .open = tiny_open,
  .close = tiny_close,
  .write = tiny_write,
  .write_room = tiny_write_room,
  .set_termios = tiny_set_termios,
};
```

Ejemplo de inicialización de driver

```
tiny_tty_driver->owner = THIS_MODULE;
tiny_tty_driver->driver_name = "tiny_tty";
tiny_tty_driver->name = "ttty";
tiny_tty_driver->major = TINY_TTY_MAJOR,
tiny_tty_driver->type = TTY_DRIVER_TYPE_SERIAL,
tiny_tty_driver->subtype = SERIAL_TYPE_NORMAL,
tiny_tty_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS,
tiny_tty_driver->init_termios = tty_std_termios;
tiny_tty_driver->init_termios.c_cflag = B9600 | CS8;
tty_set_operations(tiny_tty_driver, &serial_ops);
```

`struct termios`

```
struct termios tty_std_termios = {
  .c_iflag =ICRNL | IXON,
  .c_oflag =OPOST | ONLCR,
  .c_cflag =B38400 | CS8 | CREAD | HUPCL,
  .c_lflag =ISIG | ICANON | ECHO | ECHOE
        | ECHOK |ECHOCTL | ECHOKE | IEXTEN,
  .c_cc = INIT_C_CC
};
```

- Configuración inicial de la tty
- `man termios`

La variable `init_termios` en el `tty_driver` es una estructura `termios`. Esta variable se usa para proporcionar al usuario una configuración en caso de que lo use antes de tiempo. Los valores se copian de `tty_std_termios` que está definida en el núcleo de `tty` como se muestra. Más información en `man termios`.

¿Y la función `read`?

¿No hay función `read`?

- No es necesaria
- Drivers tty envían datos al hardware
- Datos recibidos $\rightsquigarrow$ `tty_flip_buffer`
- `tty_flip_buffer_push`

En el ejemplo anterior, cuando he rellenado las funciones del driver, no he marcado la función `read`.¿Eso significa que no se puede leer?

En realidad no es necesario implementarla. Los drivers `tty` se basan en enviar datos al hardware. La capa `tty` te permite abstraerte de muchas cosas que sería complicadas, como por ejemplo la interacción con el usuario.

Lo que el driver tiene que hacer es que en cuanto el dispositivo le envíe información la tiene que guardar en un buffer llamado `tty_flip_buffer`.

Esta estructura contiene dos arrays de datos. Cuando se recibe información del dispositivo se va guardando en el primero. Cuando se llena, se notifica a un usuario que esté esperando leer que ya hay datos disponibles. Mientras el usuario está leyendo, cualquier otro dato nuevo se va guardando en el segundo buffer. Cuando este segundo array se llena, la información es de nuevo lanzada al usuario. Básicamente, la información recibida salta (*flips*) de un buffer a otro con la esperanza de no saturar ambos.

Para prevenir que se pierdan datos un driver `tty` puede saber cómo de grande son los datos que están llegando y si no caben en ese momento puede ”forzar“ el vaciado del buffer en ese preciso momento en vez de esperar a que haya otra ocasión. Esto se hace con la llamada `tty_flip_buffer_push`.

Ejemplo

```
for (i = 0; i < data_size; ++i) {
  if (tty->flip.count >= TTY_FLIPBUF_SIZE)
    tty_flip_buffer_push(tty);

  tty_insert_flip_char(tty, data[i], TTY_NORMAL);
}
tty_flip_buffer_push(tty);
```

En este ejemplo se ve lo antes indicado. Aquí se hace uso de la función `tty_insert_flip_char` que es la que agrega un carácter a la estructura `flip_buffer` de la `tty` sin tener que preocuparnos del buffer en concreto. El último parámetro indica el tipo de carácter introducido. Si es un tipo especial de caracter que indique un error al recibir datos entonces debería ponerse `TTY_BREAK`, `TTY_FRAME`, `TTY_PARITY`, o `TTY_OVERRUN` en función del error.

Un punto importante y que cuando hice el driver `etty` me dió problemas es que si el driver puede recibir datos muy rápidamente entonces hay que activar el flag `tty->low_latency`. Que hace que la llamada `tty_flip_buffer_push` sea inmediatamente ejecutada cuando es llamada. Sino la llamada se encola para lanzar los datos al usuario y vaciar el buffer más tarde, en un futuro cercano.

Resto de `TTY`

Más sobre una `tty`

- Tan compleja como se desee
- Funciones `tiocmget` y `tiocmset`
- `ioctl`s propias de una `tty`.

Una `tty` y en general cualquier módulo tiene muchas otras cosas sobre todo a la hora de hablar directamente con el hardware. Habría estado muy bien poder ver el desarrollo completo incluyendo un aparato. Pero se requerirían muchas horas para ello.

Entre las cosas que tiene una `tty` son las funciones `tiocmget` y `tiocmset` que se usan para configurar los baudios, paridad, etc.

También las `ioctl` propias de las `tty` (ver `man ioctl_list`)

Ver ejemplo etty

Bibliografía

Bibliografía

Referencias

[1] [Peter Jay Salzman, 2005] Linux Kernel Module Programming Guide. *The Linux Documentation Project* (www.tldp.org).

[2] [O'Reilly, 2005] Linux Device Drivers Third Edition

[3] [Imágenes] Tux: tux.crystalxp.net Logo: www.gul.es

Agradecimientos

¡Gracias por venir!

Agradecimientos

- Gracias al gul por organizar estas jornadas
- Gracias a David por convencerme de dar esta charla
- Gracias a todos vosotros por venir