

Costruire un device driver per Linux

Testo di riferimento:

Alessandro Rubini, Jonathan Corbet, *Linux Device Drivers*, seconda edizione, O'Reilly 2001

Scaricabile liberamente: vedi <http://matteo.vaccari.name/so/#testo>

1

Un semplice modulo

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

MODULE_LICENSE("GPL");

int init_module() {
    printk("<1>Hello, world!\n");
    return 0;
}

int cleanup_module() {
    printk("<1>Goodbye, world!\n");
}
```

2

Esempio di uso

```
$ gcc -c -I/usr/src/linux/include/ hello.c
$ su
Password:
# insmod hello.o
Hello, world!
# rmmod hello
Goodbye, world!
#
```

3

Per vedere quali moduli sono caricati

```
$ lsmod
Module                Size  Used by    Not tainted
ds                     6624   1
i82365                22416   1
eepro100              17264   1
esssolo1              25504   0 (unused)
apm                    9148   1
```

Notare l'*usage count*

Un modulo può modificare il suo usage count

Quando l'usage count è a 0, il modulo può essere scaricato

4

Le due componenti fondamentali di un modulo

```
int init_module(void);
```

- chiamata quando il modulo è caricato
- serve a dichiarare al kernel quello che il modulo sa fare
- restituisce 0 per indicare successo

```
void cleanup_module(void);
```

- chiamata quando il modulo è scaricato
- dobbiamo ritirare tutte le dichiarazioni fatte in `init_module()`

5

printk

Non possiamo usare `stdio` nel kernel

Non possiamo usare nessuna libreria!

`printk` è la versione kernel di `printf`

“<1>” è la priorità del messaggio (vedi `/usr/include/sys/syslog.h`)

6

Simboli di preprocessor

Perché `#define __KERNEL__`?

⇒ perché i file di include sono usati occasionalmente nei programmi applicativi; il simbolo `__KERNEL__` dice che stiamo compilando codice che eseguirà in modo kernel

Perché `#define MODULE`?

⇒ perché alcuni driver possono essere compilati sia come moduli che linkati staticamente nel kernel; il simbolo `MODULE` dice che stiamo compilando codice che eseguirà come modulo

7

Perché `MODULE_LICENSE("GPL")`?

Dobbiamo dichiarare una licenza *libera*

Altrimenti installare il modulo *contamina* (taint) il kernel

```
# insmod hello.o
Warning: loading hello.o will taint the kernel: no license
See http://www.tux.org/lkml/#s1-18 for information about tainted modul
```

8

Registrare un character driver

```
int register_chrdev(unsigned int major, const char *name,  
    struct file_operations *fops);
```

- major: il major number desiderato
- name: il nome del device
- fops: insieme di procedure
- restituisce: 0 se ha successo, < 0 se fallisce

Va chiamato in `init_module()`;

Il major number può essere scelto fra quelli a disposizione per esperimenti (p. es. 42)

(punti extra per chi sa perché proprio 42... :-)

9

Il contraltare

```
int unregister_chrdev(unsigned int major, const char *name);
```

- major, name: devono avere gli stessi valori passati a `register_chrdev()`
- restituisce: 0 se ha successo, < 0 se fallisce

Va chiamata in `cleanup_module()`;

Cosa succede se scarichiamo il modulo con `rmmod(1)`, ma dimentichiamo di eseguire `unregister_chrdev()` nella `cleanup_module()` ?

⇒ disastro!

(occorre fare un reboot)

10

Esempio di uso di `register_chrdev()`

Eseguire in `init_module()`

```
#define MY_MAJOR 42  
  
result = register_chrdev(MY_MAJOR, "mydevice", &my_fops);  
if (result < 0) {  
    printk(KERN_WARNING "mydevice: can't get major %d\n", MY_MAJOR);  
    /* restituiamo un risultato negativo;  
    il modulo non verrà caricato */  
    return result;  
}
```

11

File Operations

Linux ragiona in modo object-oriented

Un "file" è una classe astratta

I suoi "metodi" sono raccolti nella struct `file_operations`

struct `file_operations` ha il ruolo di una *interface* di Java

Abbiamo un metodo per ogni syscall relativa a un file

Un driver può lasciare NULL i puntatori a metodi che non implementa

12

In <linux/fs.h>

```
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    [...]
};
```

13

Utilizziamo una variabile globale

```
struct file_operations my_fops;
```

La inizializziamo in init_module():

```
my_fops.open = my_open;
my_fops.read = my_read;
[...]
```

Tutti gli altri campi restano a NULL

14

La struct file

Non centra niente con FILE di stdio.h

Rappresenta un *file aperto*

Contiene i dati su cui operano le file_operations

Definita in <linux/fs.h>; alcuni campi sono:

```
loff_t f_pos;
posizione corrente nel "file"
```

```
struct file_operations *f_op;
contiene un puntatore alle fops passate con register_chrdev()
```

```
void *private_data;
può essere usata dal driver per conservare dati privati
```

15

La open()

```
int my_open(struct inode *inode, struct file *filp);
```

- inode: identifica il file speciale sul disco che è stato aperto
- filp: dati allocati dal kernel per questa open

cosa deve fare la open() di un driver:

- incrementa l'usage count
- inizializza il dispositivo se necessario
- alloca dati privati e li assegna a filp->private_data

16

Usare il minor number

```
int my_open(struct inode *inode, struct file *filp);
```

Il minor number si trova in `inode->i_rdev` (Occorre usare la macro `MINOR()`)

Il minor number può essere usato dal driver in due modi

- identificare dispositivi diversi
- aprire lo stesso dispositivo in modi diversi

es.

`/dev/hda1` e `/dev/hda2` sono lo stesso device, ma partizioni diverse
`/dev/hda1` e `/dev/hdb1` sono device diversi

```
brw-rw---- 1 root    disk      3,   1 Jun  9 2002 /dev/hda1
brw-rw---- 1 root    disk      3,   2 Jun  9 2002 /dev/hda2
brw-rw---- 1 root    disk     3,  65 Jun  9 2002 /dev/hdb1
brw-rw---- 1 root    disk     3,  66 Jun  9 2002 /dev/hdb2
```

17

I nostri "device"

```
/dev/scull0, /dev/scull1, /dev/scull2, /dev/scull3
```

con `major = 42` e `minor ∈ {0, 1, 2, 3}`

("scull": vedi il Rubini)

creati con

```
# mknod /dev/scull0 c 42 0
# mknod /dev/scull1 c 42 1
# mknod /dev/scull2 c 42 2
# mknod /dev/scull3 c 42 3
```

Ciascun *scull* permette di accedere a una zona di memoria *globale* e *persistente* di max 4000 byte

18

Implementazione di Scull_Dev

```
typedef struct Scull_Dev {
    void * data;          /* punta alla nostra zona di memoria */
    int size;            /* lunghezza della nostra zona di memoria */
    struct_semaphore sem; /* per la mutua esclusione */
} Scull_Dev;
```

19

Allochiamo memoria staticamente

Ci serve una struttura dati per ogni "device"

```
#define NR_SCULL_DEVICES 4
Scull_Dev scull_devices[NR_SCULL_DEVICES];
```

I semafori vanno inizializzati: mettiamo questo codice nella nostra `module_init()`

```
for (i=0; i<NR_SCULL_DEVICES; i++) {
    sema_init(&scull_devices[i].sem, 1);
}
```

20

La nostra open()

```
int scull_open(struct inode *inode, struct file *filp) {
    int    minor = MINOR(inode->i_rdev);
    Scull_Dev *dev = (Scull_Dev *) filp->private_data;

    if (!dev) {
        if (minor >= NR_SCULL_DEVICES) return -ENODEV;
        dev = &scull_devices[minor];
        filp->private_data = dev;
    }

    /* se siamo stati aperti write-only, tronca il contenuto */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY) {
        scull_trim(dev);
    }

    return 0; /* success */
}
```

C'è una race condition in questo codice; riuscite a vederla?

21

Trova la race condition

Una race condition si verifica in generale quando manipolo variabili globali

In questo caso ho una race condition su `scull_trim(dev)`

Per evitarla uso il semaforo:

```
if ((filp->f_flags & O_ACCMODE) == O_WRONLY) {
    if (down_interruptible(&dev->sem)) return -ERESTARTSYS;
    scull_trim(dev);
    up(&dev->sem);
}
```

22

Alcune parole sui semafori del kernel

```
if (down_interruptible(&dev->sem)) return -ERESTARTSYS;
```

Se la `down_interruptible()` restituisce non zero, vuol dire che abbiamo ricevuto un signal

La `down()` invece non permette di ricevere signal

(Ma rischio di creare processi che non possono essere uccisi...)

`-ERESTARTSYS` dice al chiamante che la operazione è stata interrotta da un signal

23

read() e write()

```
ssize_t read(struct file *filp, char *buf, size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char *buf, size_t count, loff_t *offp);
```

- `filp`: puntatore a file
- `buf`: area dati da trasferire
- `count`: dimensione del buffer
- `offp`: posizione di I/O

Il succo di `read()` e `write()` è copiare dati fra spazio kernel e spazio utente

Non possiamo usare `memcpy()`: cosa succede se accedendo a `*buf` abbiamo un page fault?

24

Come copiare dati fra spazio kernel e spazio utente?

Linux offre due funzioni che risolvono il problema:

```
unsigned long copy_to_user(void *to, const void *from, unsigned long count);
```

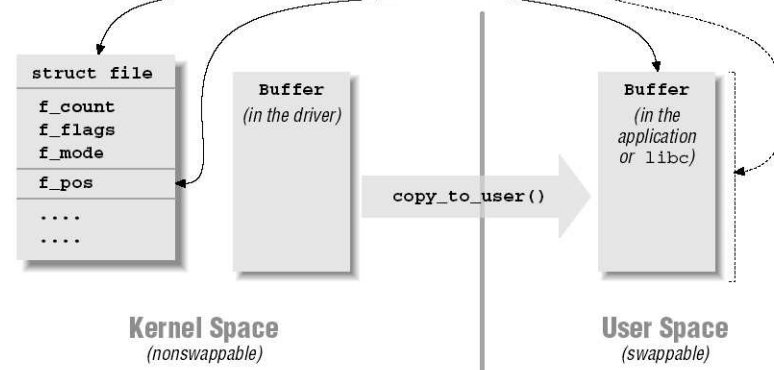
```
unsigned long copy_from_user(void *to, const void *from, unsigned long count);
```

Queste funzioni controllano anche che l'indirizzo sia corretto (appartiene allo spazio di indirizzamento del processo, ecc.)

25

Come vengono usati gli argomenti di read()

```
ssize_t dev_read(struct file *file, char *buf, size_t count, loff_t *ppos);
```



26

Il valore restituito da read()

- positivo: il numero di byte letti (può essere meno di count)
- zero: siamo a end-of-file
- negativo: condizione di errore. Il valore indica il motivo dell'errore (es. -EFAULT: bad address). Vedi <linux/errno.h> per i valori possibili

Nota:

Le applicazioni utente ricevono sempre -1 in caso di errore; il valore specifico è nella variabile `errno`

27

```
ssize_t scull_read(struct file *filp, char *buf, size_t count,
                 loff_t *f_pos)
{
    Scull_Dev *dev = (Scull_Dev *) filp->private_data;
    int ret = 0;

    if (down_interruptible(&dev->sem)) return -ERESTARTSYS;

    if (*f_pos >= dev->size) goto out;
    if (*f_pos + count > dev->size) count = dev->size - *f_pos;
    if (copy_to_user(buf, dev->data + *f_pos, count)) {
        ret = -EFAULT; /* bad address */
        goto out;
    }

    *f_pos += count;
    ret = count;

out:
    up(&dev->sem);
    return ret;
}
```

28

Il valore restituito da write()

- positivo: il numero di byte scritti (può essere meno di count)
- zero: non ha scritto nulla. Non è un errore.
- negativo: condizione di errore. Il valore indica il motivo dell'errore

29

```
ssize_t scull_write(struct file *filp, const char *buf, size_t count,
                    loff_t *f_pos)
{
    Scull_Dev *dev = (Scull_Dev *) filp->private_data;
    int ret = -ENOMEM; /* default in caso di errore: out of memory */

    if (down_interruptible(&dev->sem)) return -ERESTARTSYS;

    if (*f_pos + count > MAX_SCULL_SIZE) {
        ret = -ENOSPC; /* no space left on device */
        goto out;
    }

    /* se la scrittura va oltre la dimensione corrente, devo riallocare
       il buffer del device */
    if (*f_pos + count > dev->size) {
        void *p = kmalloc(*f_pos + count, GFP_KERNEL);
        if (!p) goto out;
        memcpy(p, dev->data, dev->size);
        kfree(dev->data);
        dev->data = p;
    }
}
```

30

```
if (copy_from_user(dev->data, buf, count)) {
    ret = -EFAULT; /* bad address */
    goto out;
}
*f_pos += count;
ret = count;

if (*f_pos > dev->size) dev->size = *f_pos;

out:
up(&dev->sem);
return ret;
}
```

31

```
int release(struct inode *, struct file *);
```

Viene chiamata quando il file viene chiuso (da tutti i processi che l'hanno aperto)

Potremmo usarla per rilasciare la memoria ... se scull non fosse persistente

Invece il rilascio lo dobbiamo fare in module_cleanup()

32

Interrupt service routines

Per poter ricevere un interrupt, il driver deve "registrarlo" al sistema operativo (durante la "open")

```
result = request_irq(my_irq, my_interrupt,
                    SA_INTERRUPT, "my_device", NULL);
if (result) {
    printk(KERN_INFO "my_device: can't get assigned irq %i\n", short_irq);
    short_irq = -1;
}
else { /* actually enable it -- assume this *is* a parallel port */
    outb(0x10,short_base+2);
}
```

33

```
int request_irq(unsigned int irq,
               void (*handler)(int, void *, struct pt_regs *),
               unsigned long flags,
               const char *dev_name,
               void *dev_id);
```

```
unsigned int irq
    This is the interrupt number being requested.
void (*handler)(int, void *, struct pt_regs *)
    The pointer to the handling function being installed. We'll discuss the arguments to this function later in this chapter.
unsigned long flags
    As you might expect, a bit mask of options (described later) related to interrupt management.
const char *dev_name
    The string passed to request_irq is used in /proc/interrupts to show the owner of the interrupt (see the next section).
void *dev_id
    This pointer is used for shared interrupt lines. When no sharing is in force, dev_id can be set to NULL, but it a good idea
```

34

Osservare quali interrupt sono riservati

```
$ cat /proc/interrupts
    CPU0          CPU1
0:   34584323    34936135   IO-APIC-edge  timer
1:    224407     226473   IO-APIC-edge  keyboard
2:         0         0         XT-PIC      cascade
5:   5636751    5636666   IO-APIC-level eth0
9:         0         0   IO-APIC-level acpi
10:   565910     565269   IO-APIC-level aic7xxx
12:   889091     884276   IO-APIC-edge  PS/2 Mouse
13:         1         0         XT-PIC      fpu
15:  1759669    1734520   IO-APIC-edge  ide1
NMI:  69520392    69520392
LOC:  69513717    69513716
ERR:         0
$
```

35

Come faccio a sapere qual'è la mia linea di interrupt?

a) probing

b) mi affido ai default "ben noti" per questo device, se esistono

```
switch(short_base) {
    case 0x378: my_irq = 7; break;
    case 0x278: my_irq = 2; break;
    case 0x3bc: my_irq = 5; break;
}
```

36