Device drivers

Una brevissima descrizione dell'astrazione dell'accesso a periferiche tramite device files

Marco Prandini

Device driver e kernel modules

- Alcuni sono cablati nel kernel, la maggior parte sono implementati da *moduli* del kernel dinamicamente caricabili
  - esplorare **/lib/modules**
  - approfondimento proposto: comandi **insmod**, **modprobe**, **lsmod**, **modinfo**
- Il codice del modulo definisce
  - come "farsi trovare"
  - come sono implementate le versioni specifiche di system call per il dispositivo gestito dal modulo
  - approfondimento proposto (hard): navigare nel codice di Linux https://elixir.bootlin.com/linux/latest/source

Caricamento dei moduli

- Step 1: viene rilevato un dispositivo fisico
- Step 2: il controller I/O manda un interrupt
- Step 3: la CPU esegue l'interrupt handler, che è parte del kernel, che identifica l'evento e lo scrive su **dbus** (un canale pub-sub per tutti gli eventi di sistema)
- Step 4: **udev** riceve l'evento, e consultando `/lib/modules/<kernel_version>/modules.alias` individua il modulo in grado di gestire il dispositivo

```
# modinfo psmouse
filename:       /lib/modules/4.13.0-37-generic/kernel/drivers/input/mouse/psmouse.ko
license:        GPL
description:    PS/2 mouse driver
author:         Vojtech Pavlik <vojtech@suse.cz>
srcversion:     16F6FEC23F72FA71FF21E33
alias:          serio:ty05pr*id*ex*
alias:          serio:ty01pr*id*ex*
depends:
intree:         Y
name:           psmouse
vermagic:       4.13.0-37-generic SMP mod_unload
signat:         PKCS#7
```

in ogni modulo sono dichiarate le stringhe identificative dei dispositivi fisici gestibili

`depmod` le raccoglie tutte e le scrive nel file `modules.alias`

Funzionamento dei moduli

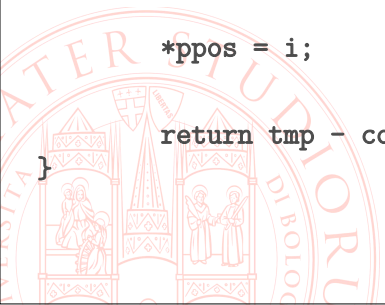
- Il modulo definisce in che modo vanno implementate le system call previste per la macro-categoria di dispositivi
  - **dispositivi a blocchi**: utilizzano buffer e cache per ottimizzare il trasferimento di blocchi di dimensione data,
    - si comportano un po' come un normale file, nel senso che "conservano" un elenco di byte singolarmente indirizzabili
    - es: dischi di vario tipo (ide, scsi, sata, usb, virtuali, …)
  - **dispositivi a caratteri**: gestiscono il trasferimento dati un carattere/byte alla volta
    - possono consumare caratteri e farci qualcosa, ad esempio mostrarli su di un terminale
    - possono fornire caratteri se disponibili, o lasciare il consumatore in attesa se non ce ne sono, es. tastiera
    - in ogni caso non supportano la ricerca random (seek)
  - dispositivi vari (misc)

Un esempio di modulo

- Nelle due slide seguenti e nelle slide 10 e 11 sono riportati frammenti di un modulo molto semplice
  - non è un vero e proprio device driver
    - non tutti i moduli del kernel lo sono, in generale!
  - strutturalmente è identico
- Il modulo alloca nel kernel una variabile *counter*
  - implementa una propria versione di *write* che accetta qualsiasi stringa, la ignora, e incrementa counter
  - implementa una propria versione di *read* che restituisce il valore di counter
  - è quindi un semplice ma completo esempio di come le syscall consentono a codice user-space di accedere a dati e funzioni kernel-space
- Il codice è scaricabile da
  http://lia.disi.unibo.it/Courses/AmmSistemi1718/counter.tar
  - include un file README con le indicazioni d'uso

Implementazione delle system call

```
static ssize_t counter_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
        unsigned char contents[COUNTER_BUFFER];
        unsigned i = *ppos;
        unsigned char *tmp;
        int size;

        size = scnprintf(contents,COUNTER_BUFFER,"%d", counter);

        for (tmp = contents; count-- > 0 && i < size; ++i, ++tmp)
                *tmp = contents[i];

        if (copy_to_user(buf, contents, tmp - contents))
                return -EFAULT;

        *ppos = i;

        return tmp - contents;
}
```



Implementazione delle system call

```
static ssize_t counter_write(struct file *file, const char __user *buf, size_t count,
loff_t *ppos)
{
        unsigned char contents[COUNTER_BUFFER];


        if (count > COUNTER_BUFFER)
                return -EFAULT;


        if (copy_from_user(contents, buf, count))
                return -EFAULT;


        printk(KERN_INFO "Happily discarding message %s", contents);

        /* This is the only useful thing: to increase a line counter */
        counter++;


        return count;
}
```

Funzionamento dei moduli

- Dov'è il codice di queste funzioni? In linea di principio c'è una tabella di puntatori, ad esempio

device	open	close	read	write	seek	...
psmouse	→	→	→	→	→	
counter	→	→	→	→	→	
disk	→	→	→	→	→	
...						

funzione che legge dal buffer della porta PS2 caratteri che rappresentano la posizione del mouse

funzione che manda incrementa counter per ogni riga ricevuta

funzione che comanda l'elettronica del disco per posizionarsi su di uno specifico blocco

- serve un sistema di nomi per dire "voglio invocare la *read* di XXX"!

Device files

- Nella cartella /dev si trovano molti file speciali

```
brw-rw---- 1 root   disk      8,  1 Mar 20 11:14 /dev/sda1
crw--w---- 1 las    tty     136,  0 Mar 20 11:17 /dev/pts/0
```

- Sono punti di accesso alle periferiche, astratti come file
  - sono memorizzati sul filesystem come normali inode, ma non hanno data block associati
  - su di essi, invocando le system call tipiche dei file si scatenano operazioni definite nel corrispondente *device driver*
  - quale d.d. usare è definito dal major number
  - l'istanza di dispositivo di quella classe è indicato dal minor number
  - possono essere di tipo block o character

Lo scheletro di un modulo

```
static const struct file_operations counter_fops = {
        .owner          = THIS_MODULE,
        .read           = counter_read,
        .write          = counter_write,
        .open           = counter_open,
        .release        = counter_release,
};

static struct miscdevice counter_dev = {
        COUNTER_MINOR,
        "counter",
        &counter_fops
};
```

registrazione dei puntatori alle funzioni che implementano le syscall (fops=file operations)

descrittore del driver:
- minor supportati
- nome del file in /dev
- puntatore alle syscall

Lo scheletro di un modulo

```
static int __init counter_init(void)
{
        int ret;

        ret = misc_register(&counter_dev);
        if (ret) {
                printk(KERN_ERR "counter: can't misc_register on minor=%d\n",
COUNTER_MINOR);
        } else {
                counter = 0;
                printk(KERN_INFO "Useless string counter version 1.0\n");
        }
        return ret;
}

static void __exit counter_cleanup(void)
{
        misc_deregister(&counter_dev);
}

module_init(counter_init);
module_exit(counter_cleanup);
```

2. si esegue una funzione ...

3. che dice al kernel che descrittore usare (v. slide precedente)

1. al caricamento del modulo ...

Device file → device driver

- Quindi al caricamento del modulo
  - il modulo dichiara che slot nella tabella dei puntatori vuole occupare, per mezzo di major e minor number
    - oppure chiede al kernel uno slot libero
  - registra nelle celle i puntatori alle proprie implementazioni delle system call
  - udev crea un device file col nome specificato dal modulo, a cui associa major e minor number
- Ora qualsiasi operazione fatta sul device file scatena la specifica azione registrata dal device driver

```
1 brw-rw---- 1 root  disk    8, 1 Mar 20 11:14 /dev/sda1
                             2
```

```
open("/dev/sda1",O_RDWR)
```

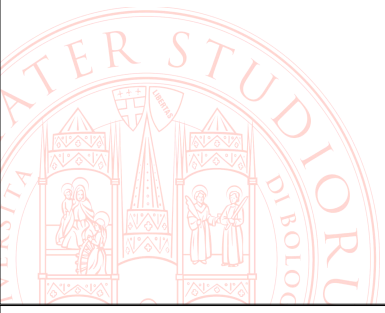
1) il kernel riconosce sda1 come file speciale
2) ricava dall'inode major e minor
3) effettua il lookup nella tabella
4) esegue la funzione registrata

device	open	close	read	write	seek	...
10,1	→	→	→	→	→	
10,100	→	→	→	→	→	
3 8,1	4 →	→	→	→	→	
...						

Device files di uso comune

- Alcuni device files non rappresentano vere periferiche, sono implementati dal kernel e sono utili per lavorare coi processi

`/dev/zero`	produce uno stream infinito di zeri (binari)
`/dev/null`	ogni read restituisce EOF, ogni write viene scartata
`/dev/random`	produce byte casuali ad alta entropia → bloccante se non ce n'è a sufficenza
`/dev/urandom`	produce uno stream pseudocasuale illimitato



Device files di uso comune

- Alcuni device files notevoli che rappresentano vere periferiche:

`/dev/tty*`	terminali fisici del sistema
`/dev/pts/*`	pseudo-terminali (dentro finestre del sistema grafico)
`/dev/sd*`	dischi e partizioni