Scrittura di moduli (driver) sotto Linux 2.6.x
P r o v v i s o r i o

Ing. El. Dipl. ETHZ Roberto Bucher[1]
Scuola universitaria professionale della Svizzera Italiana
Dipartimento di informatica ed elettrotecnica

27 febbraio 2009

[1]Galleria 2, CH-6928 Manno, Email: roberto.bucher@supsi.ch

Sommario

Questo documento ad uso interno chiarisce la programmazione e l'utilizzo di driver per il kernel di Linux, scritti quali moduli che possono essere caricati separatamente. È il risultato di una ricerca abbastanza laboriosa su diversi siti Linux, visto che pochi documenti aiutano lo sviluppatore dandogli immediatamente le informazioni utili allo sviluppo di questi moduli.
Quale esempio viene portato un semplice driver che permette di fare I/O sulla porta parallela.

Capitolo 1

Basi

Il kernel di Linux fa largo uso di moduli che non sono programmati internamente al kernel, ma che possono essere caricati nel sistema in caso di necessità. Questo permette di poter scrivere e modificare driver senza dover continuamente ricompilare il kernel. Questi moduli permettono l'accesso anche agli utenti a zone protette della memoria o porte I/O del sistema.
Per lo sviluppo, la compilazione e il caricamento di un driver occorre entrare nel sistema in qualità di "superuser", con la password di "root".
Un modulo deve avere obbligatoriamente almeno due funzioni:

- La funzione di inizializzazione del modulo (eseguita al momento di "insmod")
- La funzione di uscita del modulo (eseguita al momento di "rmmod")

Vediamo un semplicissimo esempio di modulo (modulo1.c).

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");

static int __init init_mod(void)
{
  printk("Modulo1 started...\n");
  return 0;
}

static void __exit end_mod(void)
{
  printk("Modulo1 ended...\n");
}
```

```
module_init(init_mod);
module_exit(end_mod);
```

Per compilare questo esempio occorre usare il "Makefile" seguente

```
obj-m:= modulo1.o

KDIR = /lib/modules/$(shell uname -r)/build
PWD  = $(shell pwd)

default:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
rm -f *.mod.* *.o *.ko
```

Per caricare il modulo è sufficiente chiamare il programma "insmod", mentre per scaricarlo occorre dare il comando "rmmod".

```
insmod ./modulo1.ko
dmesg
rmmod modulo1
dmesg
```

Il comando "dmesg" permette di visualizzare l'output del modulo, stampato mediante la funzione "printk" che è la funzione analoga a "printf" per il kernel. Verrà quindi visualizzato la prima volta il messaggio della procedure "init_module" ("Modulo1 started") e la seconda volta quello della procedura "cleanup_module" ("Modulo1 unloaded").

Capitolo 2

Perché un driver

Normalmene si scrivono dei driver per permettere ad un utente non privilegiato di avere accesso a parti protette del sistema, quali porte o zone di memoria. Occorre quindi arricchire il nostro codice di parti che permettano di gestire diverse operazioni, tra cui quelle di apertura del device, scrittura, lettura e chiusura.
Un device fisico quale una porta I/O o una scheda driver accessibile ad un certo indirizzo, viene vista dall'utente come un file, ed è quindi accessibile con i normali comandi di accesso (open, read write, close).
Per poter fare questo lavoro occorre che esista un device ben definito nella directory "/dev", con un major-char-number e un minor-char-number ben specifico, a cui, nell'inizializzazione del modulo, viene associato il modulo stesso. Inoltre deve essere inizializzata una tabella (struct file_operations) che contiene la lista delle procedure di accesso.
Nel prossimo capitolo ci occupiamo di un semplicissimo driver che scrive un byte sulla porta parallela e lo legge, analizzando in dettaglio tutte le procedure necessario per un corretto funzionamento dello stesso.

Capitolo 3

Il driver per la porta parallela

Nella scrittura delle varie funzioni del modulo occorre osservare strettamente la sintassi dei parametri in entrata e in uscita, che devono coincidere con quelli delle operazioni su files di "C". I file che viene mostrato è salvato con il nome "eppio_drv.c".

3.1 Apertura del driver

Questa procedura viene chiamata al momento che l'utente userà la funzione "open". In questa procedura, nel nostro esempio, ci limitiamo alla stampa di un messaggio.

```
static int epp_open(struct inode *inode, struct file *filep)
{
  printk("Device open\n");
  return 0;
}
```

3.2 Chiusura del driver

Questa procedura verrà associata al comando "close" delle operazioni su files. Anche qui ci limitiamo ad un semplice messaggio.

```
static int epp_release(struct inode *inode, struct file *filep)
{
  printk("Device closed\n");
  return 0;
}
```

3.3 Scrittura sul driver

Questa procedura viene associata alla procedura "write" utilizzata dall'utente. È importante notare che non occorrono particolari comandi per settare delle "permission" per accedere alle porte (comando "ioperm"), ma è sufficiente utilizzare i comandi di output (macro "outb" ed ev. "outw")

```
static ssize_t epp_write(struct file *filep, const char *buf, size_t count, loff_t *
{
        outb(0x00,0x37A);          /* EPP in uscita */
        outb(buf[0],0x37B);        /* Scrivi su registro ADDRESS */
        return 0;
}
```

3.4 Lettura dal driver

Questa procedura viene assegnata alla procedura "read" da file.

```
static ssize_t epp_read(struct file *filep, char *buf, size_t count, loff_t *f_pos)
{
        outb(0x20,0x37A);          /* EPP: Input */
        buf[0]=inb(0x37B);        /* leggi da registro ADDRESS */

        return 0;
}
```

3.5 Associare le procedure alle funzioni dei files

Per associare le funzioni scritte e necessarie al buon funzionamento del driver, quest'ultime devono essere esportate al sistema. Questo lavoro viene svolto riempendo una struttura particolare di tipo "file_operations", definita nell'header file "linux/fs.h". O

```
struct file_operations epp_fops =
{
  owner:      THIS_MODULE,
  open:       epp_open,
  release:    epp_release,
  read:       epp_read,
  write:      epp_write,
};
```

3.6 Inizializzazione del modulo

Durante l'inizializzazione del modulo occorre anche associare un ben definito device al nostro driver. Questo viene fatto con la chiamata della funzione "register_chardev", passando come parametri, nell'ordine, il major-char-number, una stringa con il nome del driver e la struttura con registrate le funzioni specifiche di accesso, vista nel capitolo precedente. In questo esempio il device associato al driver avrà il major-cher-number pari a 20.

```
static int __init init_epp(void)
{
  int result;
  if ((result = register_chrdev(22, "eppio", &epp_fops)) < 0)
    goto fail;
  printk("EPPIO driver loaded...\n");
  return 0;
 fail:
  return -1;
}
```

3.7 Chiusura del driver

Qui è sufficiente deregistrare il driver.

```
static void __exit end_epp(void)
{
  unregister_chrdev(22, "eppio");
  printk("EPPIO driver unloaded...\n");
}
```

In ultimo occorre aggiungere la linea in testa

```
MODULE_LICENSE("GPL");
```

e le due linee seguenti in fondo al al programma

```
module_init(init_mod);
module_exit(end_mod);
```

Capitolo 4

Compilazione e installazione del driver

La compilazione del driver che contiene le procedure viste nel capitolo precedente deve essere fatta con il "Makefile" seguente

```
obj-m:= eppio_drv.o

KDIR = /lib/modules/$(shell uname -r)/build
PWD  = $(shell pwd)

default:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
rm -f *.mod.* *.o *.ko
```

Una volta compilato, il driver può essere caricato con il comando

```
insmod ./eppio_drv.ko
```

Per poterlo utilizzare occorre ancora definire un device associato al driver tramite il major-char-number 20, con i comandi

```
mknod -m 0666 /dev/eppio c 22 0
```

Viene così creato un device "/dev/eppio" con major-char-number 20 e minor-char-number 0, accessibile da parte di tutti gli utenti.

Capitolo 5

Chiamata delle funzioni del driver da programma

L'utilizzo delle funzioni del driver da parte dell'utente è molto semplice. Il prossimo programma mostra come utilizzare il driver precedente per scrivere e leggere un byte attraverso la porta parallela.

```
#include <fcntl.h>
#include <stdio.h>
int main()
{
char buffer[1];
int fd;

  fd=open("/dev/eppio",O_RDWR);
  buffer[0]=0x00;
  write(fd,buffer,1,NULL);
  read(fd,buffer,1,NULL);
  printf("Value : 0x%02x\n",buffer[0]);
  close(fd);
}
```

Appendice A

Il driver completo

```
#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>
#include <linux/fs.h>

MODULE_LICENSE("GPL");

static int epp_open(struct inode *inode, struct file *filep)
{
  printk("Device open\n");
  return 0;
}

static int epp_release(struct inode *inode, struct file *filep)
{
  printk("Device closed\n");
  return 0;
}

static ssize_t epp_read(struct file *filep, char *buf, size_t count, loff_t *f_pos)
{
  printk("Reading...\n");
  return 0;
}

static ssize_t epp_write(struct file *filep, const char *buf, size_t count, loff_t *
{
  printk("Writing...\n");
  return 0;
}
```

```
struct file_operations epp_fops =
{
  owner:      THIS_MODULE,
  open:       epp_open,
  release:    epp_release,
  read:       epp_read,
  write:      epp_write,
};

static int __init init_mod(void)
{
  int result;
  if ((result = register_chrdev(22, "testmod", &epp_fops)) < 0)
    goto fail;
  printk("Modulo1 started...\n");
  return 0;
 fail:
  return -1;
}

static void __exit end_mod(void)
{
  unregister_chrdev(20, "testmod");
  printk("Modulo1 ended...\n");
}

module_init(init_mod);
module_exit(end_mod);
```