Scrittura di moduli (driver) sotto Linux 2.4.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

21 maggio 2006

[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 "init_module"
- La funzione di uscita "cleanup_module"

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

```
#include "linux/module.h"

int init_module(void){
  printk("Start mymodul\n");
  return 0;
}

void cleanup_module(void){
    printk("mymodul unloaded\n");
}
MODULE_LICENSE("GPL");
EXPORT_NO_SYMBOLS;
```

Per compilare questo esempio occorre dare il comando seguente:

```
gcc -c -I/usr/src/linux/include -DMODULE -D__KERNEL__ -Wall mymodul.c
```

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

```
insmod ./mymodul.o
dmesg
rmmod mymodul
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” (“Start mymodul”) e la seconda volta quello della procedura “cleanup_module” (“mymodul unloaded”).

Capitolo 2

Perché un driver

Normalmene si scrivono dei driver per permettere ad un utente normale 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 gli indirizzi delle procedure di accesso, in posizioni ben determinate.
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".

3.1 Apertura del driver

Questa procedura viene chiamata al momento che l'utente userà la funzione "open", e si occupa soprattutto di tenere un conteggio di quanti utenti hanno aperto il driver e lo stanno utilizzando. Quest'operazione è importante, perché occorre evitare di cancellare il modulo (con rmmod) quando ci sono ancora utenti che lo stanno utilizzando.
Questo conteggio viene fatto tramite la macro "MOD_INC_USE_COUNT".

```
static int epp_open(struct inode* ino, struct file* filep)
{
  MOD_INC_USE_COUNT;
  return 0;
}
```

3.2 Chiusura del driver

Analogamente alla procedura precedente, qui si tratta di decrementare il contatore degli utenti che stanno utilizzando il driver. Questa procedura verrà associata al comando "close" delle operazioni su files.

```
static int epp_close(struct inode* ino, struct file* filep)
{
  MOD_DEC_USE_COUNT;
```

```
  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* buffer,
                         size_t count,
                         loff_t *ppos)
{
        outb(0x00,0x37A);          /* EPP in uscita */
        outb(buffer[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* buffer,
                        size_t count,
                        loff_t *ppos)
{
        outb(0x20,0x37A);         /* EPP: Input */
        buffer[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 "fs.h". Ogni attività di accesso ai files ha una sua posizione ben precisa all'interno di questa tabella. È quindi sufficiente associare ad ogni elemento della tabella la corrispondente funzione del driver specifico.

```
static struct file_operations epp_fops = {
  read      : epp_read,
  write     : epp_write,
  open      : epp_open,
  release   : epp_close,
};
```

Solo le procedure utilizzate vanno immesse nella tabella, mentre le altre vanno inizializzate a "NULL".

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 22.

```
int init_module(void){
  printk("Start eppio Driver\n");
  if (register_chrdev(22,"eppio",&epp_fops)) {
    printk("Loading eppio failed\n");
    return -EIO;
  }
  else
    printk("Driver eppio loaded\n");
  return 0;
}
```

3.7 Chiusura del driver

Come già detto precedentemente, la chiusura del driver avviene unicamente quando il numero di utenti collegati allo stesso è nullo. Questo viene controllato con la macro "MOD_IN_USE".

```
void cleanup_module(void){
  if (MOD_IN_USE)
    printk("Device busy\n");
  if (unregister_chrdev(22,"eppio")!=0)
    printk("Failed to unload eppio driver\n");
  else
    printk("Driver eppio unloaded\n");
}
```

In ultimo occorre aggiungere le due linee

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

Capitolo 4

Compilazione e installazione del driver

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

```
gcc -c /usr/src/linux/include -DMODULE -D__KERNEL__ -Wall -O2 eppio.c
```

Oltre alle definizioni obbligatorie per i moduli del kernel (-DMODULE e -D__KERNEL__) si consiglia di dare un livello completo per i "warning" (-Wall). L'ultimo flag (-O2) è necessario per poter utilizzare le macro "inline" "inb" e "outb".
Una volta compilato, il driver può essere caricato con il comando

```
insmod ./eppio.o
```

Per poterlo utilizzare occorre ancora definire un device associato al driver tramite il major-char-number 22, con i comandi

```
mknod /dev/eppio c 22 0
chmod 0666 /dev/eppio
```

Viene così creato un device "/dev/eppio" con major-char-number 22 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("Valore letto : %x\n",buffer[0]);
  close(fd);
}
```

Appendice A

Il driver completo

```
#include "linux/module.h"
#include "asm/io.h"

static int epp_open(struct inode* ino, struct file* filep)
{
  MOD_INC_USE_COUNT;
  return 0;
}

static int epp_close(struct inode* ino, struct file* filep)
{
  MOD_DEC_USE_COUNT;
  return 0;
}

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

static ssize_t epp_read(struct file* filep,
                        char* buffer,
                        size_t count,
                        loff_t *ppos)
{
```

```
        outb(0x20,0x37A);              /* EPP: Input */
        buffer[0]=inb(0x37B);          /* leggi da registro ADDRESS */

        return 0;
}

static struct file_operations epp_fops = {
  read      : epp_read,
  write     : epp_write,
  open      : epp_open,
  release   : epp_close,
};

int init_module(void){
  printk("Start eppio Driver\n");
  if (register_chrdev(22,"eppio",&epp_fops)) {
    printk("Loading eppio failed\n");
    return -EIO;
  }
  else
    printk("Driver eppio loaded\n");
  return 0;
}

void cleanup_module(void){
  if (MOD_IN_USE)
    printk("Device busy\n");
  if (unregister_chrdev(22,"eppio")!=0)
    printk("Failed to unload eppio driver\n");
  else
    printk("Driver eppio unloaded\n");
}
MODULE_LICENSE("GPL");
EXPORT_NO_SYMBOLS;
```