Scrivere drivers in linux

Linux Device Driver
Cap. 1,3,10

Università
degli Studi
della Campania
Luigi Vanvitelli

Meccanismi e politiche

- Driver ➔ meccanismi
  - Quali capacità devono essere fornite
- Politiche:
  - Come queste capacità devono essere usate

- In un sistema l'ideale sta nel poter separare queste caratteristiche
  - Es: XServer – Finestre/Session manager
  - Es: Socket – Applicazione ftp

Funzioni del SO

- Process Management
- Memory Management
- Filesystems
- Device control
- Networking

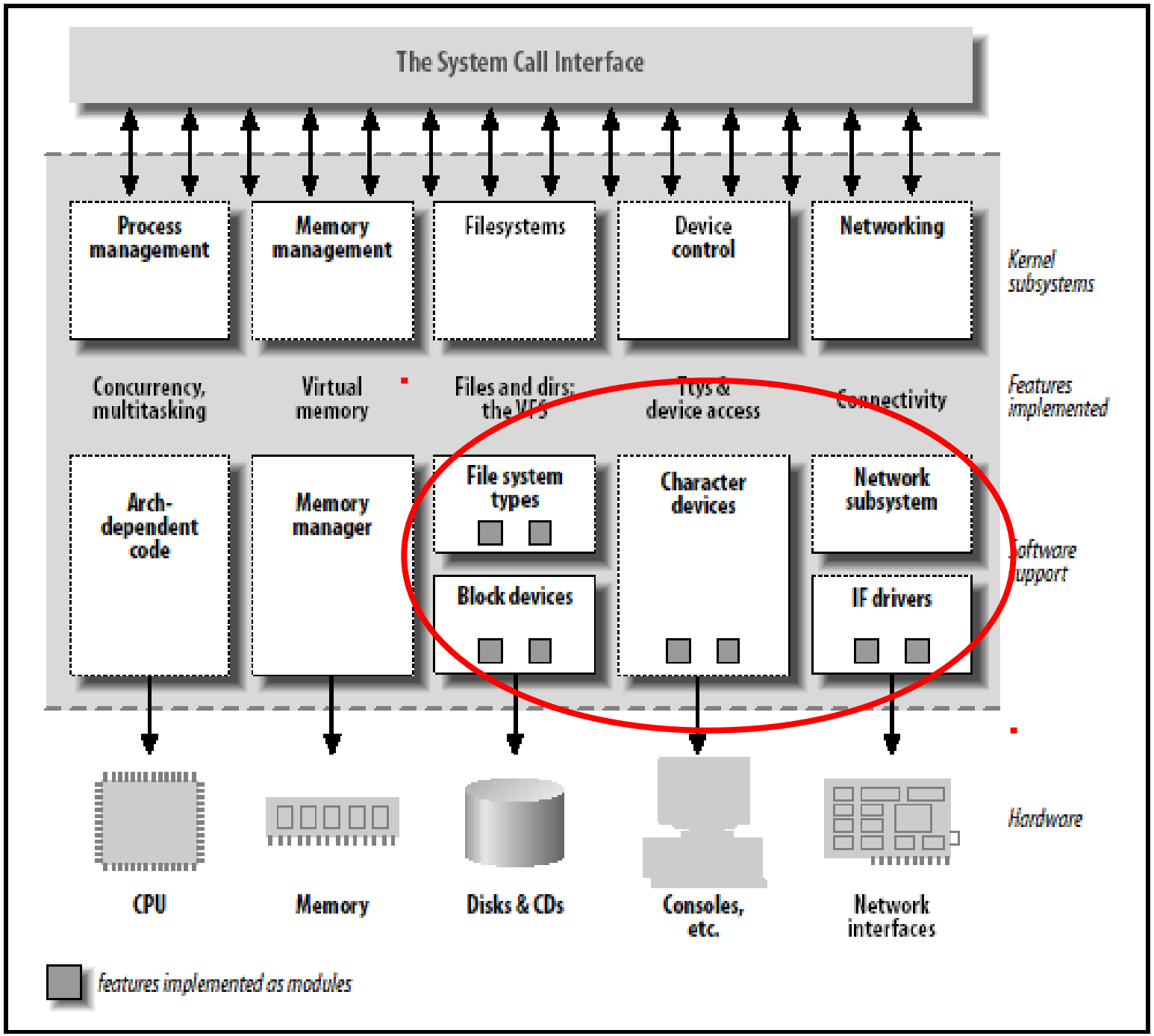
Estensibilità del kernel linux

- Linux consente di estendere il kernel caricando moduli dinamicamente:
  - insmod: carica il nuovo modulo
  - rmmod: rimuove il modulo
- Non occorre riavviare il sistema
- I drivers non sono i soli moduli di sistema operativo

Tipi di device

In linux un device può essere:

- a caratteri
- a blocchi
- di rete


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs; the VFS
Toys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
Hardware
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
features implemented as modules

Device a caratteri

- Si dialoga con il device come con uno stream di byte (come un file).
- Il driver deve implementare le funzioni ***open***, ***close***, ***write***, ***read***.
- L'astrazione di console e di una porta seriale è un file: */dev/console*; */dev/ttyS0*
- Unica differenza è che la maggiorparte delle volte si tratta di canali, non di veri e propri files
- Raramente possono essere visti come aree di memoria all'interno delle quali posso spostarmi

Devices a blocchi

- Sono visti come directory del filesystem
- Un block device è un dispositivo (Es: disco) che può ospitare un filesystem.
- Per l'utente non cambia niente: usa il device scrivendo e leggendo una qualunque quantità di byte (non 512 o potenze più grandi di 2)
- Per quanto riguarda l'implementazione del driver, cambia molto l'interfaccia verso il S.O.

Network devices

- Si distinguono dai precedenti
- Il driver scrive e legge *pacchetti*
- Non può essere visto come un file, ma viene identificato da un nome univoco (es: eth0)
- Non si usano read e write, ma funzioni completamente differenti

Altri devices

- Ci sono dispositivi che fuoriescono dalla classificazione appena fatta
- Essi sono virtualizzati per mezzo di diversi strati software
- USB, SCSI, …
  - Una penna USB è vista come un device a blocchi
  - Una scheda wireless USB è vista come una interfaccia di rete

Realizzare un driver

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

MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
printk(KERN_ALERT "Hello, world\n");
return 0;
}

static void hello_exit(void)
{
printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Osservazioni

- *module_init* e *module_exit* sono due macro, come anche *module_license*
- *printk* è una funzione del kernel che funziona come printf (che però appartiene alla libreria C)
- KERN_ALERT è la priorità del messaggio (massima affinché venga sicuramente visualizzato)

Testare il modulo

% **su**

root# **insmod ./hello.ko**

Hello, world

root# **rmmod hello**

Goodbye cruel world

root#

N.B.: i messaggi potrebbero non essere visualizzati se stiamo eseguendo un emulatore di terminale. Occorre controllare nei file di log come /var/log/messages

Drivers e applicazioni

- Un modulo del kernel registra le operazioni che è in grado di fare (e gli si può chiedere solo di eseguire queste)
- È event driven
- Alla rimozione deve necessariamente rilasciare le risorse (cosa che non è necessario fare quando termina un'applicazione)
- In unix esistono due livelli di privilegio. I moduli del kernel eseguono in ***kernel_mode***, le applicazioni in ***user_mode***.

Compilare il driver

- Occorre avere i file headers del Kernel
- Occorre controllare in *Documentation/Changes* che si abbia la versione necessaria del compilatore
- È utile, ma non necessario che si stia eseguendo la stessa versione del kernel che si sta utilizzando
- Nel nostro semplice caso è sufficiente preparare un makefile con la seguente riga:
  - obj-m := hello.o
- Se il modulo fosse sviluppato su due files invece che su uno solo (file1.c e file2.c):
  - obj-m := module.o
  - module-objs := file1.o file2.o
- Il risultato è un file *.ko

Compilare il kernel

- Supponendo che il sorgente del kernel si trovi in una directory: *~/kernel-2.6*:

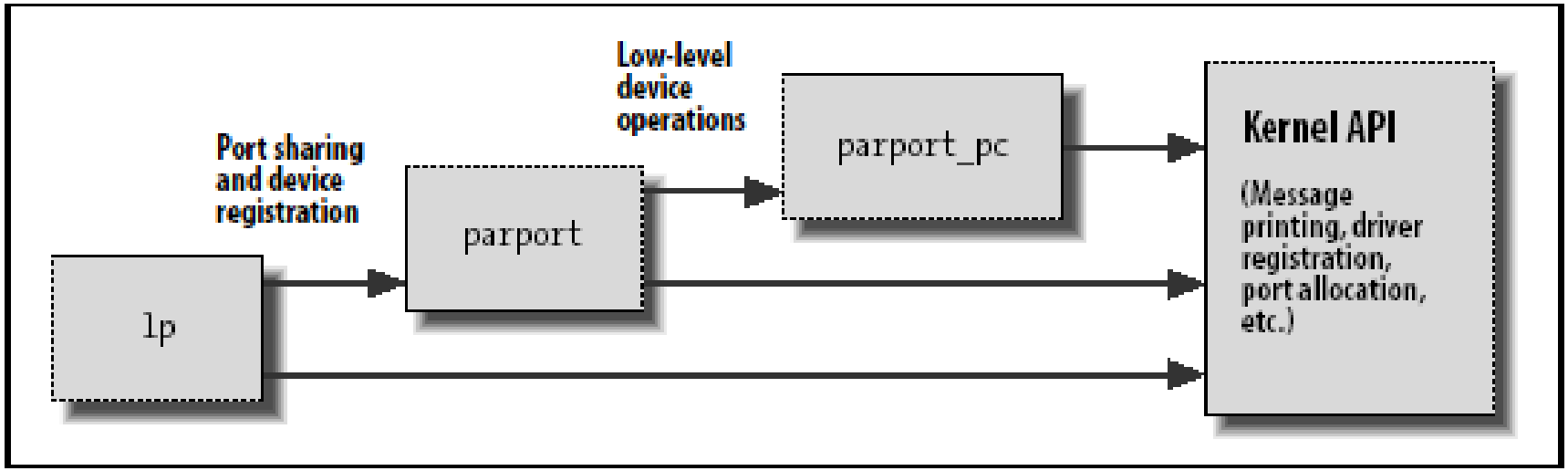
  make -C ~/kernel-2.6 M=`pwd` modules

Compilare il kernel

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
    obj-m := hello.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

Moduli in stack

- Quando un modulo esporta simboli o funzioni questi vengono a far parte del kernel
- Altri moduli possono utilizzarli
- Uno stack di moduli è utile per costruire sistemi complessi
- ***modeprobe*** invece di **insmod** carica tutte le dipendenze



Errori nella fase di registrazione

```
int __init my_init_function(void)
{
int err;
/* registration takes a pointer and a name */
err = register_this(ptr1, "skull");
if (err) goto fail_this;
    err = register_that(ptr2, "skull");
if (err) goto fail_that;
    err = register_those(ptr3, "skull");
if (err) goto fail_those;
    return 0; /* success */
fail_those: unregister_that(ptr2, "skull");
fail_that: unregister_this(ptr1, "skull");
fail_this: return err; /* propagate the error */
}
```

Cleanup

```
void __exit my_cleanup_function(void)
{
unregister_those(ptr3, "skull");
unregister_that(ptr2, "skull");
unregister_this(ptr1, "skull");
return;
}
```

```
struct something *item1;
struct somethingelse
*item2;
int stuff_ok;

void my_cleanup(void)
{
if (item1)
   release_thing(item1);
if (item2)
   release_thing2(item2);
if (stuff_ok)
   unregister_stuff( );
return;
}
```

```
int __init my_init(void)
{
      int err = -ENOMEM;
      item1 = allocate_thing(arguments);
      item2 = allocate_thing2(arguments2);
      if (!item2 || !item2)
         goto fail;
      err = register_stuff(item1, item2);
      if (!err)
      stuff_ok = 1;
      else
            goto fail;
      return 0; /* success */
      fail:
           my_cleanup( );
           return err;
}
```

Passare parametri

- Un driver può aver bisogno di parametri nel momento in cui viene caricato
  - Es: l'indirizzo del porto
- È possibile passare parametri:
  - A linea di comando
  - Attraverso il file */etc/modprobe.conf*
- Vediamo come è possibile farlo per l'esempio precedente

- Passiamo un numero ed una stringa:

  insmod hellop howmany=10 whom="Mom“

- Recuperiamoli nel modulo:

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

User space vs Kernel Space

- User Space Vantaggi:
  - È possibile utilizzare l'intera libreria C. Evitando di ricorrere a programmi esterni (Come le utilities distribuite con il driver stesso)
  - Il programmatore può usare debugger convenzionali.
  - È possibile effettuare il kill del solo driver senza che ne risenta il S.O.
  - La memoria utente è swappabile e quindi il driver non usato di frequente non occupa la RAM
  - Un driver ben programmato può ancora consentire accesso concorrente al dispositivo come succede per i drivers in kernel-space.
  - Se il driver è proprietario si evitano problemi di licenza e problemi quando cambiano le interfacce del kernel.

User space vs Kernel Space

- Le interruzioni non sono disponibili in user space.
- L'utilizzo del DMA è possibile *mmap*ping */dev/mem* e può farlo un utente privilegiato.
- L'accesso ai porti I/O è possibile solo dopo aver usato metodi come *ioperm* o *iopl*. Inoltre non tutte le piattaforme supportano queste system calls e l'acesso ai porti può risultare troppo lento.
- La risposta può essere troppo lenta perché potrebbe essere richiesto un context switch
- Ancora peggio il driver potrebbe essere wappato sul disco.
- *mlock* system call potrebbe bloccare le pagine, ma un programma utente utilizza molta memoria ed inoltre questa call è riservata a utenti privilegiati.
- Molti dispositivi non possono essere gestiti in user space. Incluso alcune interfacce di rete e devices a blocchi.

Un driver per la porta parallela

- Appunti: Ing. El. Dipl. ETHZ Roberto Bucher1
  - Scuola universitaria professionale della Svizzera Italiana (Dipartimento di informatica ed elettrotecnica)
- Linux device drivers:
  - Chap. 1,3,9

Allocare un porto

- struct resource *request_region(unsigned long first, unsigned long n, const char *name);
- void release_region(unsigned long start, unsigned long n);
- int check_region(unsigned long first, unsigned long n);

- *La lista dei porti allocati è accessibile tramite /proc/ioports*

Barrier

- Alcuni porti sono mappati in memoria nonostante l'hardware utilizzi un isolated I/O
- Occorre garantirsi che le scritture non rimangano in cache
- Le letture provengano direttamente dalla RAM

- Per far ciò si usano delle istruzioni dette barrier che bloccano il processo fino a quando la memoria non è stata aggiornata
  - #include <asm/system.h>
  - void rmb(void);
  - void wmb(void);

Major and minor numbers

```
crw-rw-rw-    1 root     root       1,   3 Apr 11  2002 null
crw-------    1 root     root      10,   1 Apr 11  2002 psaux
crw-------    1 root     root       4,   1 Oct 28 03:04 tty1
crw-rw-rw-    1 root     tty        4,  64 Apr 11  2002 ttyS0
crw-rw----    1 root     uucp       4,  65 Apr 11  2002 ttyS1
crw--w----    1 vcsa     tty        7,   1 Apr 11  2002 vcs1
crw--w----    1 vcsa     tty        7, 129 Apr 11  2002 vcsa1
crw-rw-rw-    1 root     root       1,   5 Apr 11  2002 zero
```

Major and minor numbers

- Major number
  - Identifica il driver
- Minor number
  - Identifica univocamente il dispositivo

Allocazione minor number

- Occorre allocare un certo numero di minor number:
  - int register_chrdev_region(dev_t first, unsigned int count,char *name);
- Occorre deallocare i minor numbers quando non verranno più usati
  - void unregister_chrdev_region(dev_t first, unsigned int count);
- Il sistema operativo non sa niente dei minor numbers, questi sono utilizzati solo dalle applicazioni per indirizzare i devices

Allocare major number

- Alcuni major numbers sono dedicati
  - *La lista dei numeri riservati si trova in Documentation/devices.txt*
  - I devices allocati si trovano in: /proc/devices
- È possibile una allocazione dinamica, ma:
  - In tal modo non è possibile creare nodi in anticipo dal momento che questo varia

Esistono già implementati

- Le porte parallele vengono mappate su (tre locazioni da un byte: dato, stato, controllo) :
  - 0x378 la prima
  - 0x278 la seconda
    - *Ameno che gli indirizzi non vengano cambiati a load time*
- /dev/short0
  - Scrive e legge un byte all'indirizzo base 0x378
- /dev/short1
  - Scrive a base +1 … fino a base +7

Creazione file speciale

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 20 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.

Utilizzo del device da programma

```
#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);
}
```

Realizzazione ed utilizzo

- Utilizzo in scrittura:
  - echo -n "any string" > /dev/short0

- Realizzazione:

```
while (count--) {
outb(*(ptr++), port);
wmb( );
}
```

Apertura/Chiusura “Device”

```
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;
}
```

Lettura/Scrittura da Device

```
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;
}
```

```
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;
}
```

Associazione funzioni/operazioni

L'nterfaccia verso il sistema operativo viene inizializzata mediante una struttura dati definita in ”linux/fs.h”.

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

Inizializzazione Modulo

Registrazione mayor number e associazione funzioni/ operazioni

```
static int __init init_epp(void)
{
  int result;
  if ((result = register_chrdev(20, "eppio", &epp_fops)) < 0)
    goto fail;
  printk("EPPIO driver loaded...\n");
  return 0;
 fail:
  return -1;
}
```

Rimozione Modulo

```
static void __exit end_epp(void)
{
  unregister_chrdev(20, "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);
```