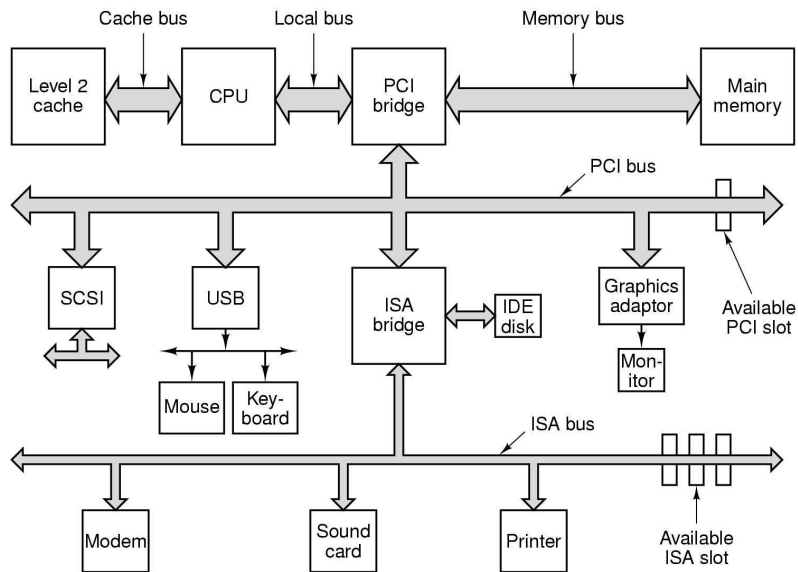


## Input.output (I/O)



## Input-output (I/O)

Grande varietà di dispositivi

Due grandi classi:

- ▶ *block devices* (es. dischi)
  - accesso a un blocco di dati per volta,
  - posso indirizzare il singolo blocco
- ▶ *character devices* (es. mouse, modem)
  - accesso a un byte per volta,
  - non posso indirizzare il singolo byte

Alcuni dispositivi non ricadono esattamente in nessuna delle due (es. clock, video controller)

## BUS

Un *bus* è

- un insieme di fili
- un protocollo

## Device controllers

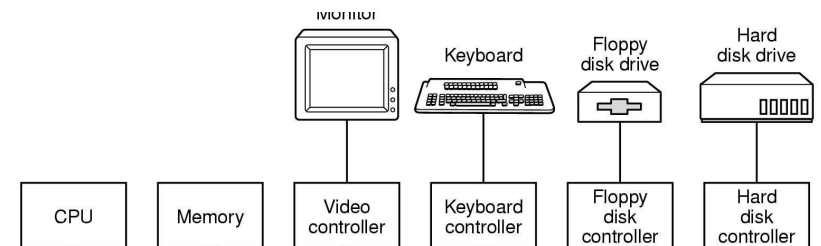
Un device si compone di:

- parti meccaniche
- parti elettroniche

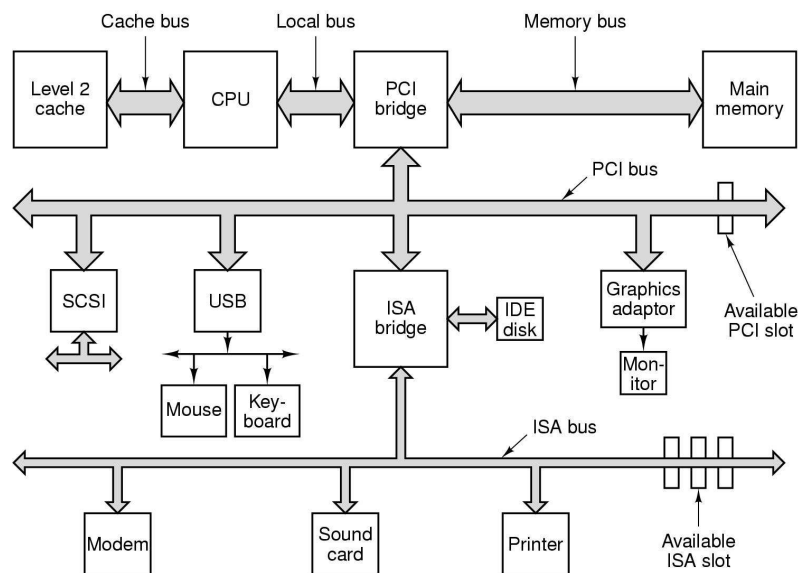
La parte elettronica è il *device controller*

Può in alcuni casi pilotare più dispositivi (es. IDE, SCSI, USB)

Compito del controller: interfacciare il protocollo di basso livello utilizzato dalla parte meccanica del device con il protocollo di alto livello usato dal bus



## Architettura di un tipico PC



## Tipi di bus

Un tipico PC possiede diversi tipi di bus

- Memory bus (il più veloce)
- Backplane bus (es. PCI, ISA)
- Specialized bus (es. SCSI, USB)

Il bus ISA è presente per offrire compatibilità con vecchie schede

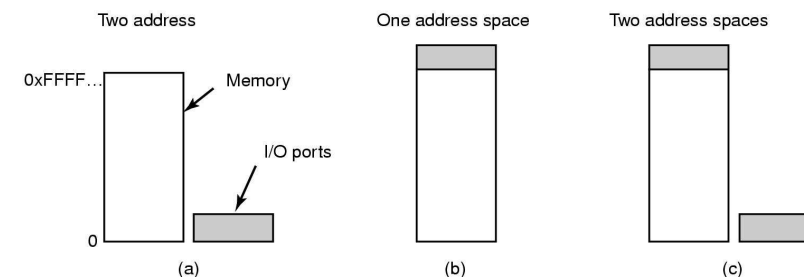
## Comunicazione con il device

Il controller possiede

- registri di stato
- registri di comando
- buffer di I/O

Come accedere a questi registri?

Due alternative:



- Spazi di indirizzi separati per I/O e memoria
- Memory-mapped I/O
- Ibrido

## Spazi di indirizzi separati per I/O e memoria

Richiedono istruzioni speciali per accedere agli indirizzi di I/O

Es. i386: istruzioni IN, OUT

Gli indirizzi di I/O si chiamano *ports*

Lo spazio disponibile è di 64KB

## Esempio: il file /proc/ioports in Linux

Mostra gli intervalli di port riservati dai vari *device drivers*

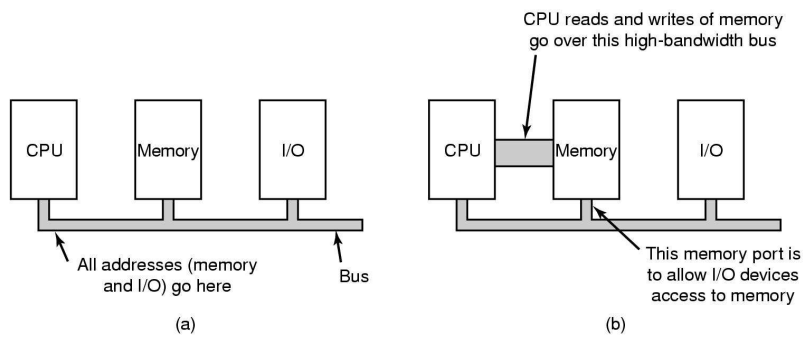
```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
03c0-03df : vga+
```

[...]

## Memory-mapped I/O

Il controller è connesso al bus della memoria

Ciascun dispositivo (di memoria o di I/O) confronta gli indirizzi che appaiono sul bus con i propri



## Esempio: il file /proc/iomem

Mostra gli intervalli di RAM riservati dai device driver

```
00000000-0009f7ff : System RAM
0009f800-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000fffff : System ROM
00100000-13feffff : System RAM
    00100000-0028031c : Kernel code
    0028031d-002eade3 : Kernel data
13ff0000-13ffffbfff : ACPI Tables
13fffc00-13ffffff : ACPI Non-volatile Storage
14000000-14000fff : Texas Instruments PCI1410 PC card Cardbus Controller
```

[...]

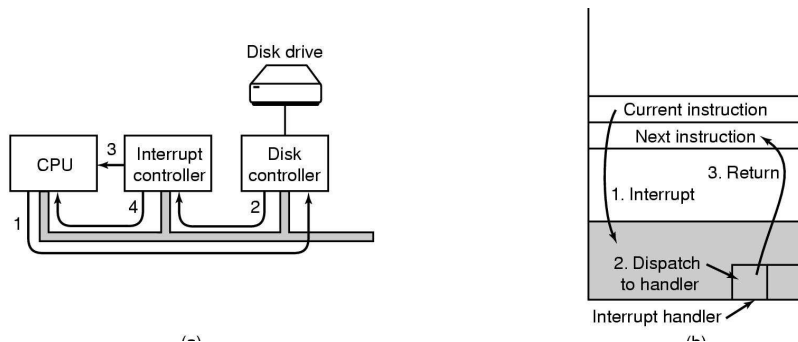
## Memory-mapped I/O: vantaggi e svantaggi

- Non occorrono istruzioni speciali: i device driver possono essere scritti in C senza assembler
- La protezione dei device è fornita dal meccanismo di memoria virtuale
- posso usare direttamente tutte le istruzioni che accedono alla memoria (quindi è più veloce che usare IN, OUT)

Ma:

- il caching della memoria può essere un problema
- il *probing* per trovare i dispositivi collegati diventa più difficile
- occorre programmare opportunamente il PCI bridge

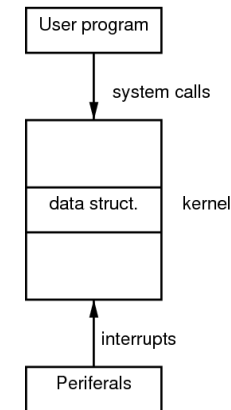
### Uso degli interrupt



1. Il SO manda una richiesta al controller
2. Quando i dati sono disponibili il controller manda un interrupt all'*interrupt controller*
3. L'IC manda un interrupt alla CPU
4. Il SO legge il numero del dispositivo sul bus

Lo scopo è evitare il *nolling*

## Il modello del kernel



## Interrupt Controller

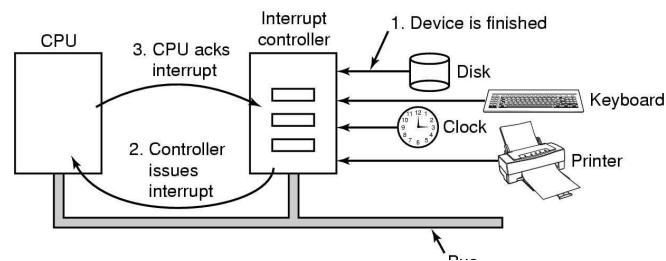
Il numero del dispositivo è usato come indice in una tabella detta *interrupt vector*

Viene selezionato un interrupt handler

Lo handler restituisce un ACK all'Interrupt Controller

L'IC non manda un secondo interrupt fino a che non riceve l'ACK  
⇒ si evitano race conditions

## Esempio: il file /proc/pci



Le linee di interrupt sono incorporate nel backplane bus

Nel caso PCI abbiamo 16 IRQ connessi a due Programmable Interrupt Controller chip (PIC)

PCI devices found:

[...]

Bus 0, device 7, function 2:

USB Controller: Intel Corp. 82371AB PIIX4 USB (rev 1).

IRQ 11.

Master Capable. Latency=32.

I/O at 0x1480 [0x149f].

[...]

Bus 0, device 18, function 0:

Ethernet controller: Intel Corp. 82557 [Ethernet Pro 100] (rev 8).

IRQ 11.

Master Capable. Latency=66. Min Gnt=8. Max Lat=56.

Non-prefetchable 32 bit memory at 0xf4100000 [0xf4100fff].

I/O at 0x1440 [0x147f].

Non-prefetchable 32 bit memory at 0xf4000000 [0xf40fffff].

[...]

## Principi della gestione I/O (i)

### Device indpendence

Il SW applicativo deve poter leggere dati da un floppy, da un disco IDE, un CD-ROM in maniera *trasparente*

Esempio: la shell di Unix

```
sort < pippo > pluto
```

Il programma “sort” vede due “file”, uno di input e uno di output.  
Non sa a quali dispositivi siano connessi

## Principio di Unix

Everything is a file

## Principi della gestione I/O (ii)

### Uniform naming

In Unix, il FS del floppy e del CD-ROM fanno parte di un singolo, unico file system

## Buffering

Il SO deve leggere i dati che arrivano in tempo reale da un dispositivo veloce e salvarli in un buffer

Es. telecamere, schede di rete

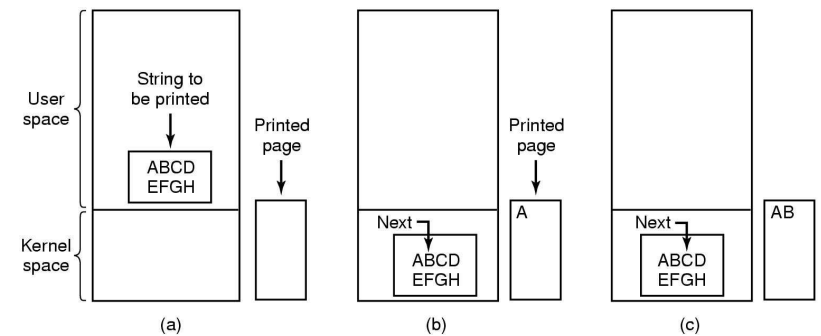
## Blocking vs. non-blocking I/O

Dal punto di vista applicativo, l'I/O bloccante è più semplice

Dal punto di vista del SO, il polling è uno spreco di tempo

Compito del SO: rendere bloccanti le operazioni di I/O per i processi utente senza sprecare tempo

## Programmed I/O (i)



Passi necessari per stampare una stringa

## Programmed I/O (ii)

```
copy_from_user(buffer, p, count);          /* copia dalla memoria
                                             utente in un buffer
                                             del kernel 'p' */

for (i=0; i<count; i++) {
    while (*printer_status_register != READY) /* aspetta che la
                                             stampante sia pronta */
        ;
    *printer_data_register = p[i];          /* stampa un carattere */
}
return_to_user();
```

## Interrupt-driven I/O

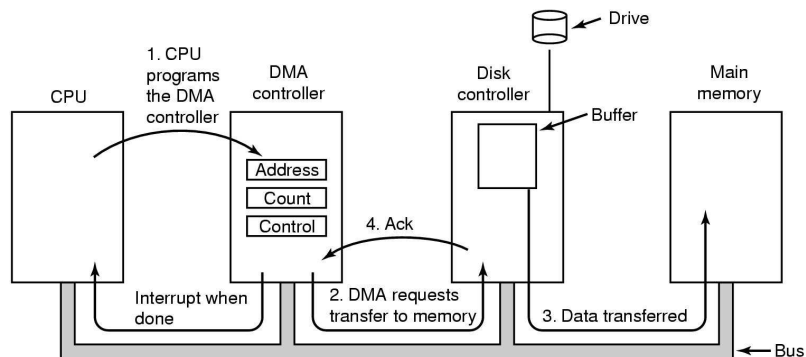
### syscall

```
copy_from_user(buffer, p, count);
while (*printer_status_register
      != READY)
    /* wait */ ;
*printer_data_register = p[0];
current->status = BLOCKED;
schedule();
return_from_syscall();
```

### interrupt handler

```
if (i == count) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    i++;
}
acknowledge_interrupt();
return_from_interrupt();
```

## Direct Memory Access (DMA)



## I/O con DMA

### syscall

```
copy_from_user(buffer, p, count);
setup_DMA();
current->status = BLOCKED;
schedule();
```

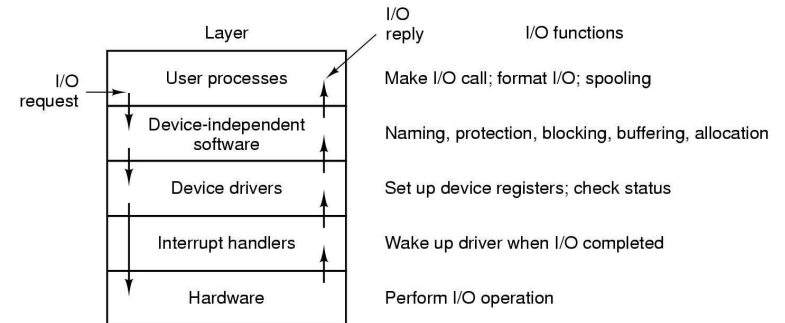
### interrupt handler

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

## Strati di software nella gestione dell'I/O

0. User-level I/O software (ex. "stdio")
1. Device-independent OS software
2. Device drivers (ex. driver SCSI)
3. Interrupt handler (chiama una procedura all'interno del driver opportuno)
4. Hardware

## Strati del software di I/O



## Interrupt handlers

Sono installati dal device driver

Quando un device driver richiede I/O si blocca

- ▶ down su semaforo
- ▶ wait su cond variable
- ▶ receive message

Quando arriva l'interrupt atteso l'**interrupt handler**

- ▶ up sul semaforo
- ▶ wakeup su cond variable
- ▶ send message

## Device drivers

Dispositivi specializzati per la gestione di

- classi di dispositivi (es. SCSI)
- specifici dispositivi (es. ncr53c8xx)

Spesso sono realizzati da terze parti

Eseguono in modo kernel

Si interfacciano al kernel per mezzo di *interfacce standard*

In Unix, abbiamo due interfacce:

- block drivers
- character drivers



## Device drivers: handling a request

Get request from device-independent sw (es. *file system*)

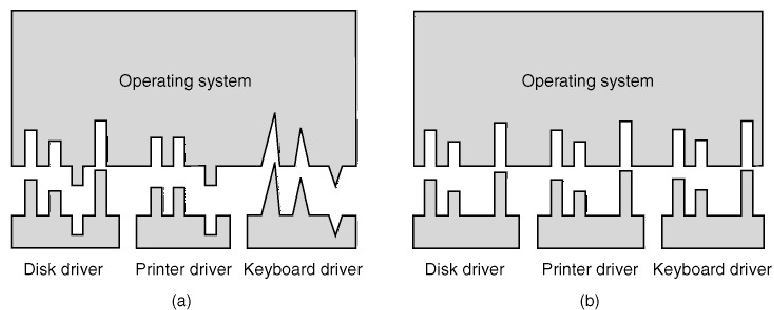
- ▶ Check request is valid
- ▶ Translating from abstract (*read*) to concrete (*start motor*)
- ▶ Send commands to registers
- ▶ Block if necessary
- ▶ Send answer to device-independent sw

## Device-independent I/O software

### Responsibilities

- ▶ Uniform interfacing for device drivers
- ▶ Buffering
- ▶ Error reporting
- ▶ Allocating and releasing dedicated devices
- ▶ Provi

## Uniform interfacing for device drivers



## In Unix

I driver sono divisi in due categorie: *block* o *character*

Ciascun driver è identificato dal *tipo* (c o b) e un *major number* 0-255

Ciascun dispositivo gestito da un certo driver è identificato da un *minor number* 0-255

La mappa dei major e minor number di Linux si trova in [Documentation/devices.txt](#)

## Esempio da Documentation/devices.txt

```
4 char      TTY devices
             0 = /dev/tty0      Current virtual console
             1 = /dev/tty1      First virtual console
             ...
             63 = /dev/tty63     63rd virtual console
             64 = /dev/ttyS0     First UART serial port
             ...
             255 = /dev/ttyS191  192nd UART serial port
```

## La chiamata di sistema open(2) II

```
O_RDONLY    open for reading only
O_WRONLY    open for writing only
O_RDWR      open for reading and writing
O_NONBLOCK  do not block on open or for data to become available
O_APPEND    append on each write
O_CREAT     create file if it does not exist
O_TRUNC     truncate size to 0
O_EXCL      error if create and file exists
O_SHLOCK    atomically obtain a shared lock
O_EXLOCK    atomically obtain an exclusive lock
```

Esempio:

```
int fd = open('pippo.txt', O_RDONLY);
```

## La chiamata di sistema open(2) I

### NAME

open - open or create a file for reading or writing

### SYNOPSIS

```
#include <fcntl.h>
```

```
int
open(const char *path, int flags, mode_t mode);
```

### DESCRIPTION

The file name specified by path is opened for reading and/or writing as specified by the argument flags and the file descriptor returned to the calling process.

## La chiamata di sistema close(2)

### NAME

close - delete a descriptor

### SYNOPSIS

```
#include <unistd.h>
```

```
int
close(int d);
```

### DESCRIPTION

The close() call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated.

[...]

When a process exits, all associated file descriptors are freed ...

## La chiamata di sistema read(2)

### NAME

read, readv, pread - read input

### SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
```

ssize\_t

```
read(int d, void *buf, size_t nbytes);
```

### DESCRIPTION

Read() attempts to read nbytes of data from the object referenced by the descriptor d into the buffer pointed to by buf.

### RETURN VALUES

If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable errno is set to indicate the error.

## Esempio di I/O con le syscall di Unix I

Conta tutte le occorrenze del carattere 'x'

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char ** argv) {
    int count = 0;
    if (1 == argc) {
        printf(Usage: %s file [file ...]\n);
        exit(EXIT_FAILURE);
    }
    for (int i=1; i<argc; i++) {
        count += count_occurrences('x', argv[i]);
    }
    printf("Il numero di occorrenze e' %d\n", count);
    exit(0);
}
```

## Esempio di I/O con le syscall di Unix II

```
int count_occurrences(char ch, char * filename) {
    int result = 0;
    char buffer[1];
    int fd = open(filename, O_RDONLY);
    if (-1 == fd) {
        printf("Non posso aprire %s: %d\n", filename, errno);
        return 0;
    }
    while (1 == read(fd, buffer, 1)) {
        if (ch == buffer[0]) {
            result++;
        }
    }
    close(fd);
    return result;
}
```

## User-space I/O SW

```
int count_occurrences(char ch, char * filename) {
    int result = 0;
    int chread;
    FILE *f = fopen(filename, "r");
    if (0 == f) {
        printf("Non posso aprire %s: %d\n", filename, errno);
        return 0;
    }
    while (chread = fgetc(f)) {
        if (EOF == chread) break;
        if (ch == chread) result++;
    }
    fclose(f);
    return result;
}
```

## Installazione dei driver

In Unix tradizionale: sono linkati staticamente al kernel

In Linux: i *moduli* possono essere installati dinamicamente

quindi posso evitare il reboot!

lista dei moduli caricati: comando `lsmod`, oppure

`cat /proc/modules`

```
> cat /proc/modules
ds                6624      1
i82365            22416     1
pcmcia_core       40896     0 [ds i82365]
eeepro100         17264     1
esssolo1          25504     1
gameport          1308     0 [esssolo1]
soundcore         3236     4 [esssolo1]
usbserial         17888     0 (unused)
usb-uhci           20708     0 (unused)
usbcore           48000     0 [usbserial usb-uhci]
```

## Il comando `mknod(1)`

`mknod [options] name {bc} major minor`

*make block or character special files*

*A special file is a triple (boolean, integer, integer) stored in the filesystem. The boolean chooses between character special file and block special file. The two integers are the major and minor device number.* — manuale di `mknod(1)`

I file speciali non si possono creare con `open(2)`

Il comando `mknod(1)` usa la syscall `mknod(2)`

## Special files in Unix

Ciascun dispositivo può essere associato a un file detto *special file*

Es. `/dev/fd0` è associato al primo floppy disk

Quindi posso comunicare “direttamente” col dispositivo

(in realtà comunico con il driver)

```
$ ls -l /dev/tty1
crwx-w-- 1 matteo tty 4, 1 Apr 28 10:53 /dev/tty1
```

## Alcuni file speciali “buffi”

```
crw-rw-rw- 1 root sys 1, 3 Jul 18 1994 /dev/null
crw-rw-rw- 1 root sys 1, 5 Jul 18 1994 /dev/zero
```

Il driver per il dispositivo “memoria” (da `Documentation/devices.txt`)

1 char	Memory devices	
	1 = <code>/dev/mem</code>	Physical memory access
	2 = <code>/dev/kmem</code>	Kernel virtual memory access
	3 = <code>/dev/null</code>	Null device
	4 = <code>/dev/port</code>	I/O port access
	5 = <code>/dev/zero</code>	Null byte source
	6 = <code>/dev/core</code>	OBSOLETE - replaced by <code>/proc/kcore</code>
	7 = <code>/dev/full</code>	Returns <code>ENOSPC</code> on write
	8 = <code>/dev/random</code>	Nondeterministic random number gen.
	9 = <code>/dev/urandom</code>	Faster, less secure random number gen.
	10 = <code>/dev/aio</code>	Asynchronous I/O notification interface

## In che cosa consiste un device driver

Un device driver è *un insieme di procedure*

(Se vogliamo, è l'implementazione di un "interfaccia" nel senso di Java)

Per ogni operazione che posso fare su un file, c'è una corrispondente procedura nel driver

Device	Open	Close	Read	Write	Ioctl
Null	null	null	null	null	null
Memory	null	null	mem_read	mem_write	null
Keyboard	k_open	k_close	k_read	error	k_ioc
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioc
Printer	lp_open	lp_close	error	lp_write	lp_ioc